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

Automatic parallelism strategy generation with minimal memory redundancy

Yanqi SHI[§], Peng LIANG[§], Hao ZHENG, Linbo QIAO, Dongsheng LI[‡]

National University of Defense Technology, Changsha 410000, China

E-mail: yqshi@nudt.edu.cn, peng_leung@nudt.edu.cn, zhengh@nudt.edu.cn, linboqiao@nudt.edu.cn, lds1201@163.com

Received Oct. 10, 2023; Revision accepted Oct. 17, 2023; Crosschecked

Abstract: Large-scale deep learning (DL) models are trained distributedly due to memory and computing resource limitations. Few existing strategy generation approaches take optimal memory minimization as the objective. To fill this gap, we propose a novel algorithm that generates optimal parallelism strategies with the constraint of minimal memory redundancy. We propose a novel Redundant Memory Cost Model (RMCM) to calculate the memory overhead of each operator in a given parallel strategy. To generate the optimal parallelism strategy, we formulate the parallelism strategy searching problem into an integer linear programming problem and use an efficient solver to find minimal-memory intra-operator parallelism strategies. Furthermore, the proposed algorithm has been extended and implemented in a multi-dimensional parallel training framework and is characterized by the ability of high throughput and minimal memory redundancy. Experimental results demonstrate that our approach achieves significant memory savings of up to 67% compared to the latest Megatron-LM strategies, and has a similar throughput. The principal contribution of the present research lies in its provision of a novel algorithm that optimizes parallelism strategies, reducing memory redundancy in large-scale DL models. In conclusion, our paper introduces a memory-efficient algorithm for generating parallelism strategies, surpassing existing strategies in reducing memory requirements.

unedited

Key words: Deep learning; Automatic parallelism; Minimal memory redundancy
https://doi.org/10.1631/FITEE.2300684 **CLC number:**

1 Introduction

Large-scale deep learning (DL) models have been a tremendously popular area of study in recent years owing to (Krizhevsky et al., 2012; Naumov et al., 2019; Brown et al., 2020; Dan et al., 2023), which is a result of scaling up model sizes and dataset sizes (Zhuang et al., 2017). For example, PaLM with 540 billion parameters is trained with a corpus of 780 billion tokens representing a wide range of natural language use cases (Chowdhery et al., 2022).

As a result, training these models is computationally intensive and time-consuming, requiring plenty of computing resources. However, resources on a single node are limited and insufficient to train these models. (Lan et al., 2018; Guan et al., 2020) To address this challenge, researchers have proposed various parallelism strategies to accelerate the training process of large-scale DL models. However, designing parallel strategies is not a straightforward task, and it often requires considerable expertise and trial-and-error exploration to achieve high performance (Mo, 2018; Xiaobin et al., 2023).

To relieve us from the parallelism design procedure, researchers propose auto-parallelism methods (Jia et al., 2018; Cai et al., 2022; Zheng et al., 2022; Liu et al., 2023a) that can find decent strategies given a specific model and environment. These methods

§ Yanqi SHI and Peng LIANG contributed equally to this manuscript
‡ Corresponding author
ORCID: Yanqi SHI, https://orcid.org/0000-0002-8899-1018
© Zhejiang University Press 2023

model the communication costs of different parallelism strategies and then use a search algorithm to find the optimal strategy. However, existing studies in the literature typically aim to minimize the communication cost for a given model. Under this search objective, the resulting parallelism strategies are often not memory-optimal.

To fill this gap, we propose an algorithm for searching memory-optimized operator-level parallelism strategies by analyzing the memory overhead (MO) of neural networks in distributed training. This enables us to scale up further the hidden or batch size in limited hardware resources with limited memory.

Initially, we instantiate a memory analyzer, enabling the computation of MO for any specified strategy. Subsequently, we proceed to formalize the problem of strategy searching. To accomplish this, we leverage an external solver specifically designed for integer linear programming (ILP) and employ it as a basis for arriving at the decisions, pertaining to optimal multidimensional intra-operator parallelism strategies, that minimize memory usage.

The proposed algorithm developed in this paper offers several vital contributions. First, it allows the identification of strategies with minimal redundant memory. Second, it involves a simplified expression of the strategy searching problem, thus making it more concise and easy to understand. Finally, it allows experiments to be conducted that demonstrate the MO benefits deriving from the present research concept.

The remainder of the paper is organized as follows. Section 2 introduces the background in the field of auto parallelism; Section 3 presents an overview and details of the proposed algorithm, including the methodology and critical components; Section 4 describes the experimental setup and presents the results and analysis; Section 5 concludes the paper and outlines directions for future research.

2 Background

2.1 Intra- and inter-operator parallelism

Alpa (Zheng et al., 2022) catalogs existing parallelism methods into two orthogonal categories: intra-operator and inter-operator parallelism. Intra-operator parallelisms are parallelism schemes that partition an operator's involved tensors along some dimensions, assign the resulting partitioned computation to multiple devices, and let them execute different parts of the computation simultaneously. Inter-operator parallelism, including pipeline parallelism (Harlap et al., 2018; Huang et al., 2019; Liu et al., 2023b), partitions models into several stages with multiple operators. This paper focuses on generating multi-dimensional and memory-optimal intra-operator parallelism strategies.

2.2 Existing intra-operator parallelism schemes

Since Hinton (Krizhevsky et al., 2012) trained AlexNet using two GPUs in 2012, researchers have proposed many intra-operator parallelism schemes, including data parallelism (DP), model parallelism (MP) (Shazeer et al., 2018; Shoeybi et al., 2019), and zero redundancy data parallelism (ZeRO-DP) (Rajbhandari et al., 2020).

2.2.1 Data parallelism

DP partitions and distributes the data across devices with a replicated model. Each device computes the gradients using the split data and uses communication mechanisms such as AllReduce or Broadcast to synchronize the gradients or model parameters with other devices so that after every iteration, the models on all workers are the same.

2.2.2 Model parallelism

MP partitions the model parameters across devices and makes devices process the same data. MP produces partial sum or sliced results when the parameter matrix is partitioned row-wise and column-wise. Row-wise MP (Row-MP) requires synchronization to unify the operator's results on different devices. Column-wise MP (Column-MP) does synchronization only in backward propagation.

2.2.3 Zero redundancy data parallelism

Zero redundancy data parallelism (ZeRO-DP) (Rajbhandari et al., 2020) works by partitioning the model parameters, gradients, and optimizer states into multiple pieces and then distributes these pieces across the nodes in the distributed system in such a way that there is no redundancy in memory usage. ZeRO-DP has three stages: Stage 1 partitions

Unedited

the optimizer states, Stage 2 additionally partitions gradients, and Stage 3 partitions model parameters by introducing another All-Gather to guarantee mathematical equivalence. From stage 2, ZeRO-DP replaces the All-Reduce operation in DP with a Reduce-Scatter on gradients and an All-Gather operation on parameters to reduce the memory redundancy of gradients. Since stage 3 might introduce more communication overheads that lower the throughput, we apply ZeRO-DP stage 2 in this work.

2.3 Strategy searching algorithm and objective

Researchers have proposed methods to automatically search parallelism strategies. ToFu (Wang et al., 2019), TensorOpt (Cai et al., 2022), and Alpa (Zheng et al., 2022) generate intra-operator parallelism strategies by minimizing the overall communication cost of a computation graph under the observation that all different strategies of an operator have the same computation cost. In the cases of ToFu and TensorOpt, to produce better results, the dynamic programming algorithm proposed by OptCNN (Jia et al., 2018) is adapted. Alpa formalizes the searching problem as an integer programming problem and uses a solver to progress the solution. In the present research, we formulate the problem of intra-operator parallelism strategy in a minimization manner, allowing us to utilize an ILP solver to assist in identifying memory-optimal strategies.

To our knowledge, unlike other automated parallelism methods, which adopt the minimization of communication costs as an object, our approach presents a variation in that it endeavors to generate parallelism strategies with a primary objective of memory minimization and a secondary objective of computation of communication volume produced by such strategies, together with presenting a comparison of the performance in terms of transcendence with those of the latest methods.

3 Method design

3.1 Overview

Our approach primarily prioritizes the minimization of memory redundancy as an optimization objective. Diminishing memory redundancy enables us to scale up further the hidden or batch size in limited hardware resources with limited memory.

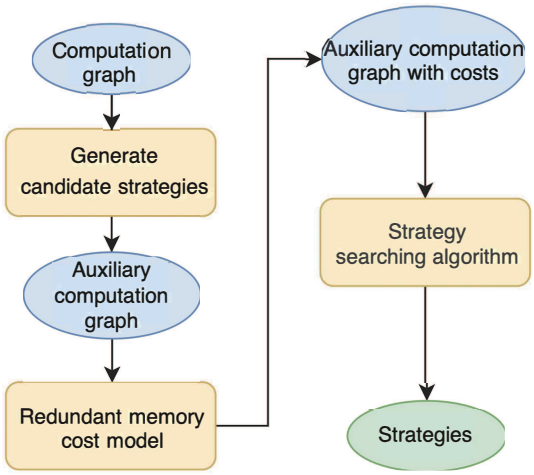

Fig. 1 Framework of automatic parallelism strategy generation with minimal memory redundancy

To minimize memory redundancy, we propose a method to formalize the strategy search problem into an ILP using an auxiliary computation graph and to solve the problem accurately using a third-party ILP solver.

Furthermore, the constrained communication bandwidth has emerged as a bottleneck in amplifying parallel training efficiency. For DP, the impact of communication bandwidth is mainly manifested in parameter synchronization. In MP, training requires parameters such as gradients frequently. Limited communication bandwidth can result in a longer duration of such data exchanges, slowing down parallel training. Thus, to alleviate communication stress, we treat the communication volume generated during training as a secondary optimization objective.

Algorithm 1 provides an overview framework of the research embodied in the present study. In Section 3.2, We build a memory cost model to calculate the MO of arbitrary intra-operator strategies. We generate candidate strategies for each operator in the computation graph and initialize an auxiliary computation graph. Each node in the auxiliary computation graph represents a node in the original graph with a specific candidate strategy. More details about the auxiliary computation graph can be found in Section 3.3. Then, we apply the cost model to the auxiliary graph, assigning redundant memory cost to each node to form a complete graph. In Section 3.4, we illustrate the process of calculating communication costs from two perspectives by analyzing the auxiliary computation graph and subsequently

unedited

Algorithm 1 Generate parallelism strategy with minimal memory redundancy

Require: Computation Graph $\mathbf{G} = (V, E)$, Device Graph $\mathbf{D} = (V_D, E_D)$;
Ensure: Parallelism Strategy $\mathbf{P}$;
$G_A = (V_A, E_A), V_A \leftarrow \emptyset, E_A \leftarrow \emptyset$
// Generate Auxiliary Graph
for $(u, w) \in E$ **do**
  $U_A = \text{GenerateCandidateStrategies}(u, \mathbf{D})$
  $W_A = \text{GenerateCandidateStrategies}(w, \mathbf{D})$
  $V_A = U_A \cup W_A \cup V_A$
  for $u_a \in U_A$ **do**
    for $w_a \in W_A$ **do**
      $E_A = E_A \cup \{(u_A, w_A)\}$
    end for
  end for
end for
$M \leftarrow \emptyset$
// Compute Redundant Memory Cost and Communication Volume
for $(u_A, w_A) \in E_A$ **do**
  $m_{(u_A, w_A)} = \text{ComputeMemory}(u_A, w_A)$
  $M = M \cup \{m_{(u_A, w_A)}\}$
  $c_{(u_A, w_A)} = \text{ComputeResharding}(u_A, w_A)$
  $C = C \cup \{c_{(u_A, w_A)}\}$
end for
for $v_A \in V_A$ **do**
  $c_{u_A} = \text{ComputeParallel}(u_A, w_A)$
  $C = C \cup \{c_{v_A}\}$
end for
// Searching Optimal Strategy
$\mathbf{P} = \text{PathSearching}(C, M, G_A)$

incorporating them into the optimization objective. Finally, we formalize the strategy searching problem as an ILP problem in Section 3.5. The strategy-searching algorithm inputs that complete graph to find memory-minimal strategies.

3.2 Redundant memory cost model

To accurately evaluate the memory cost arising from the use of different parallelism strategies we first build a memory cost model to calculate the MO of arbitrary intra-operator strategies. The memory cost model takes an operator and its candidate parallelism strategy as inputs and returns the corresponding memory cost.

The key to building the memory cost model is understanding the redundancy of different strategies. DP, MP, and ZeRO-DP stage 2 bring memory redundancy to the training cluster. Taking matrix multiplication (MatMul) as an example, its forward computation is shown as Eq. (1), and its backward computation is shown as Eq. (2) and Eq. (3).

$$Y = XW, \tag{1}$$

$$\delta W = X^{\mathrm{T}} \delta Y, \tag{2}$$

$$\delta X = \delta Y W^{\mathrm{T}}, \tag{3}$$

X, W, Y represent the input tensor, weight matrix, and output tensor, respectively. $\delta X, \delta W, \delta Y$ represents for the gradients of X, W, Y, respectively.

As Fig. 2 shows, although DP partitions X and Y along the batch-size axis, it duplicates the model weights across devices within the ZeRO-DP (stage 2) group. Suppose the parallelism degree is N; then the redundancy it brings will be: $(n-1) * \text{size}(W)$. Similarly, the redundancies of Row-MP and Column-MP are $(n-1) * \text{size}(Y)$ and $(n-1) * \text{size}(X)$, respectively. Suppose the DP, Row-MP, and Column-MP degrees are d, r, c, respectively. Then, the memory redundancy a MatMul operator brings would be:

$$\begin{aligned} M = & \frac{1}{d * r * c} * (d(d-1) * \text{size}(W) \\ & + r(r-1) * \text{size}(Y) + c(c-1) * \text{size}(X)). \end{aligned} \tag{4}$$

With these formulations, we construct memory cost models for each operator. Using this Redundant Memory Cost Model, each memory redundancy caused by different strategies can be calculated as the weight of edges on the auxiliary graph.

3.3 Path searching problem modeling and generate candidate strategies

The computation graph is a graphical structure used to describe and organize the computational operations of neural network models. The nodes in the computation graph represent operations, while the edges represent dependencies between different operations.

The nodes in the computation graph are amenable to the use of various potential parallelism strategies that can be employed for each operation. The collection of all possible parallelism strategies a node can adopt is referred to as a candidate strategy. We expand the original computation graph to generate a new auxiliary computation graph based on these candidate strategies.

Each node in the auxiliary computation graph represents a parallelism strategy adopted for the

unedited

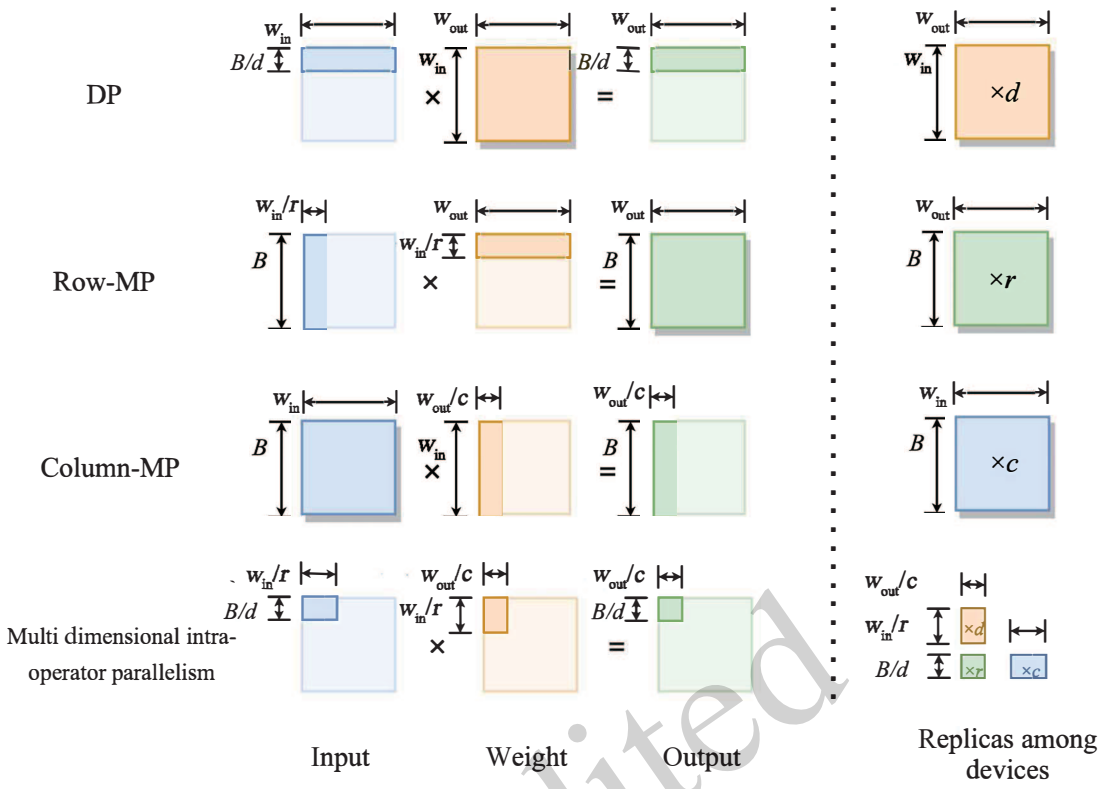

Fig. 2 Illustration of redundancy memory in intra-operator parallelism

unedited

computational operations corresponding to the original computation graph. Similarly to the original computation graph, the auxiliary computation graph is a directed graph where edges represent the data flow. The weight of an edge refers to the MO incurred by choosing a particular parallelism strategy. The value of the weight is determined through the RMCM.

Dissimilarly to the case with the original calculation graph, the data flowing through the nodes are fixed. Different paths can be chosen in the auxiliary calculation graph. By generating an auxiliary calculation graph and calculating memory cost, we model the strategy-searching problem as a path-searching problem.

Fig. 3 shows an example: we first expand the original computation graph $G = (V, E)$ to form an auxiliary computation graph $G_A = (V_A, E_A)$. For node $A \in V$ in Fig. 3a, since it has K_a different strategies, there will be K_a auxiliary nodes in V_A. Each auxiliary node $A_i \in V_A$ represents the i^{th} strategy of node $A \in V$. As seen in Fig. 3c the strategy-searching problem becomes a path-searching problem in the auxiliary computation graph. The red lines in Fig. 3c indicate that we choose the second, first, and K_c^{th} strategies for nodes $A, B, C \in V$, respectively.

3.4 Communication overhead

Considering that communication overhead has become the bottleneck hindering improvement in the performance of parallel training in deep learning, to maximize the throughput of training while generating parallel strategies with minimal memory redundancy, we take the computation of communication volume produced by parallelism strategies as a secondary objective during strategy search.

In executing operator-level parallelism, the principal genesis of training communication overhead stems from the enactment of the parallel operator in conjunction with the resharding procedure. Contemporary automatic parallelism methodologies typically compute the aforementioned two categories of communication volume individually, and the approach employed to measure communication volume is comparatively well-established.

The communication overhead brought about by

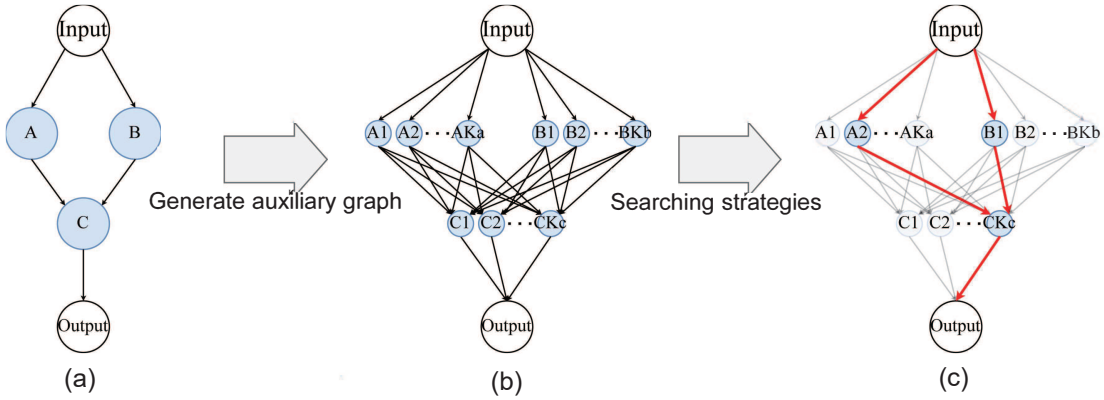

Fig. 3 Illustration of auxiliary computation graph: (a) original computation graph; (b) auxiliary computation graph; (c) strategy decision

the execution of parallel strategies can correspond to each selected node on the auxiliary graph. The specific value of the communication volume can be calculated based on the shape of the tensor, the partitioning method, and the type of operator execution. It is denoted as C_v, here v is a node in the auxiliary Graph.

Since our method uses a combination of different parallelism strategies, it may result in a mismatch between the tensor distribution of the output of the previous operator and the distribution required by the input of the next operator. The extra communication volume brought by resharding is denoted as $C_(i,j)$, where (i,j) is an Edge in the auxiliary graph.

3.5 Strategy searching algorithm

Using the methods in Section 3.3 and Section 3.2, the computational operations during model training, candidate parallelism strategies, and the MO incurred with the use of different strategies have been represented by an auxiliary computation graph with costs. To generate parallelism strategies more efficiently, we formalize the path-searching problem as an ILP problem, as represented below:

$$\min \quad \sum_{(i,j)\in E_A} B_{ij}M_{ij} + \alpha C(V_A, E_A) \tag{5}$$

$$\text{s.t.} \quad \sum_{v_a \in V_A} X_{v_a} = 1, \tag{6}$$

$$\sum_{(i,v_a)\in E_A} B_{iv_a} = X_{v_a} \times in_degree(v), \tag{7}$$

$$\sum_{(v_a,k)\in E_A} B_{v_a k} = X_{v_a} \times out_degree(v), \tag{8}$$

$$B_{ij}, X_{v_a} \in \{0,1\}, \forall (i,j) \in E_A, \forall v_a \in V_A,$$

where X_{v_a}, B_{ij} are to-be-solved bool values indicating the selection of the vertex $v_a \in V_A$ and edge $(i,j) \in E_A$. M_{ij} is the MO of edge $(i,j) \in E_A$. $C(V_A, E_A)$ is the total Communication Volume, which may be calculated as :

$$C(V_A, E_A) = \sum_{v_a \in V_A} X_{v_a} C_{v_a} + \sum_{(i,j)\in E_A} B_{ij} C_{ij},$$

α is a hyperparameter that controls the proportion of communication volume to the objective. The larger the value of alpha, the higher the proportion of communication attributed to the optimization objective. Equation 6 informs the solver that we only select one strategy for all $v \in V$. Equation 7 and equation 8 limit any $v_a \in V_A$ to have the same indegree and outdegree as their original vertex $v \in V$. To avoid selecting multiple strategies for $v \in V$, we set the indegree and outdegree of $v_a \in V_A$ to zero if it is not selected.

Formalizing the problem into ILP formulation endows the generated parallelism strategies with the capability to fully adhere to the current hardware constraints, thereby ensuring the non-occurrence of issues like memory overflow or communication conflicts. Moreover, unlike the case with other solution methods, solving the ILP problem guarantees to obtain the optimal solution, which helps us find the parallelism strategy with minimal MO given the memory constraints.

Additionally, existing high-performance ILP solvers can efficiently accelerate the solving process of integer programming problems, potentially saving time compared to using alternative methods for strategy search.

unedited

Table 1 Parameters of the model in the experiment

Model name	Batch size	Hidden size	Sequence length	Head num
Bert-large	2	1024	512	16
1.7B	16	2304	2048	24
3.6B	16	3072	2048	24
7.6B	16	4096	2048	32

4 Experiment

Our experimental trials were conducted utilizing the PyTorch (version 1.11) framework on a server configuration consisting of eight A100 GPUs, two AMD EPYC 7282 16-core Processors, a memory capacity of 512 GB, and the Ubuntu 20.04 operating system. CUDA (version 11.1) was employed, and COPT was used as an external integer programming solver.

Our approach primarily uses a combination of DP and MP. These parallelism strategies guarantee the accuracy of gradient calculations and parameter modifications. Consequently, parallel training via our strategic framework is mathematically congruent with conventional training methodologies. In congruence with preceding research on automated parallelism, we abstain from contrasting model efficiency, directing our focus solely towards memory expenditure and throughput.

4.1 Megatron-LM, Alpa, and Colossal-Auto

We employed the parallelism strategy provided by Megatron-LM (Shoeybi et al., 2019; Narayanan et al., 2021) as our baseline approach. Megatron-LM is a state-of-the-art (SOTA) system engineered for training transformer-based (Vaswani et al., 2017) language models on GPUs. It combines data, pipeline, and operator parallelism for enhanced performance. The effectiveness of these techniques can be controlled using three integer parameters that dictate the degree of parallelism allocated to each technique. The best configuration is determined by conducting a grid search and following the guidelines outlined in their paper. This approach ensures sufficient utilization of the parallelism techniques, resulting in superior training outcomes for homogeneous transformer-based language models.

We also provide a comparison with another SOTA method, Alpa. The basic situation of Alpa has been described above. Alpa also uses ILP solvers to search intra-operator parallelism strategy; the difference, however, is that Alpa's optimization object is to reduce communication as much as possible.

Finally, we compared our proposed method with Colossal-Auto (Liu et al., 2023a) by using the Colossal-AI (Li et al., 2023)framework, a novel deep learning framework implemented using Python.

4.2 Evaluatiom

Our methodology preserves the integrity of the synchronous gradient descent algorithm, ensuring that the convergence of the model remains unaffected. As a result, our evaluation primarily focuses on quantifying the training MO.

The experiments were conducted using a single-layer transformer model (Vaswani et al., 2017), specifically an encoder layer of the transformer, at four different scales: Bert-Large (Devlin et al., 2019) and 1.7 B, 3.6 B, and 7.5 B Megatron-LM (Narayanan et al., 2021). The specific hyperparameters used for each scale are presented in Table 1.

These hyperparameters were chosen to evaluate the performance and effectiveness of our method with varying model sizes. And α is set to 1.

By examining the results obtained from these experiments, insights can be gained into the scalability and capability of the system when training transformer-based language models at different scales.

Our experiments evaluated the runtime MO of different-sized single-layer transformer models (Bert-Large, 1.7 B, 3.6 B, and 7.5 B); the parameters are shown in Table 1. To assess this metric, we utilized the built-in CUDA interface routines for recording the MO during runtime.

Using a dedicated ILP solver, the parallel strategy generation in the environment of two, four, and eight GPUs can be completed in 0.02 s, 0.05 s, and 0.23 s, respectively.

unedited

Table 2 MO during runtime compared to Megatron-LM, Alpa, and Colossal-AI

Model	Device	Memory overhead(MB)				Relative		
		Megatron	Alpa	Colossal	Ours	Megatron	Alpa	Colossal
Bert-Large	2	216.06	148.81	331.63	185.05	0.86	1.25	0.55
1.7B	2	25948.15	24976.17	15708.42	18825.93	0.73	0.75	1.20
3.6B	2	39620.04	37473.76	16136.24	27634.30	0.70	0.73	1.71
7.5B	2	41473.96	38345.58	42265.51	22369.74	0.54	0.58	0.52
Bert-Large	4	219.22	158.90	331.64	148.70	0.68	0.94	0.45
1.7B	4	31043.46	27606.46	15708.42	10575.90	0.34	0.38	0.67
3.6B	4	41419.42	35070.54	16136.24	13568.87	0.33	0.38	0.84
7.5B	4	43822.80	35633.63	41862.86	18219.26	0.42	0.51	0.44
Bert-Large	8	199.81	177.01	335.73	128.05	0.64	0.72	0.38
1.7B	8	30067.89	47090.08	21731.21	21450.06	0.71	0.46	0.98
3.6B	8	40104.50	57417.39	24169.72	26477.73	0.66	0.46	1.10
7.5B	8	42044.41	54803.73	41877.61	26593.83	0.63	0.49	0.64

unedited

Table 2, Fig. 4 and 5, illustrate our comparative analysis of the parallelism strategy generated by our approach versus Megatron-LM's, Alpa's, and Colossal-Auto's on two, four, and eight devices. Megatron-LM's parallelism strategy serves as the baseline for our evaluation.

The results show that our strategy consistently reduces MO compared to other methods, while the throughput during model training is similar to those to the other three methods.

Specifically, when conducting parallel training experiments on two devices, the MO consistently remains lower than that of the Megatron-LM strategy, with the worst-case scenario being 0.86 times the original and the best-case scenario being 0.54 times the original. When training on four devices, the MO is consistently reduced compared to the original strategy of Megatron-LM, with the worst-case scenario being 0.68 times the original and the best-case scenario being 0.33 times the original. Similarly, on eight devices, our strategy saves memory by around 65%.

Compared with Alpa's strategy, our method generally has a smaller MO when training models of different sizes. When training with two devices, our method achieves at least 0.58 of the Alpa's MO. When training a 1.7 B model with four devices, the memory usage can achieve at least 0.38 times that of Alpa. When training with eight devices, the MO is at least 0.46 times that of Alpa. Only when two devices train a Bert-Large is the training's MO 1.25 times more than of Alpa's. We speculate that the additional MO is brought by frameworks when requesting memory from the system. This is more evident when the model size is not large.

Presumably, owing to the utilization of the Colossal-AI framework, the Colossal-Auto approach manifests a certain performance advantage compared to alternative methods conducted on our self-implemented code. However, its peak memory consumption exhibits some inconsistency across distinct training scenarios. Nonetheless, amidst such circumstances, our method continues to showcase a steady advantage in conserving MO.

Furthermore, we observed that the reduction in redundant memory also leads to a decrease in the amount of data needing to be copied between devices. Hence, in applying a parallel strategy that minimizes memory redundancy, there is also a certain reduction in communication overhead. This is one of the reasons why our method rivals other SOTA approaches in terms of throughput.

As our approach generates parallel strategies with less memory redundancy and communication

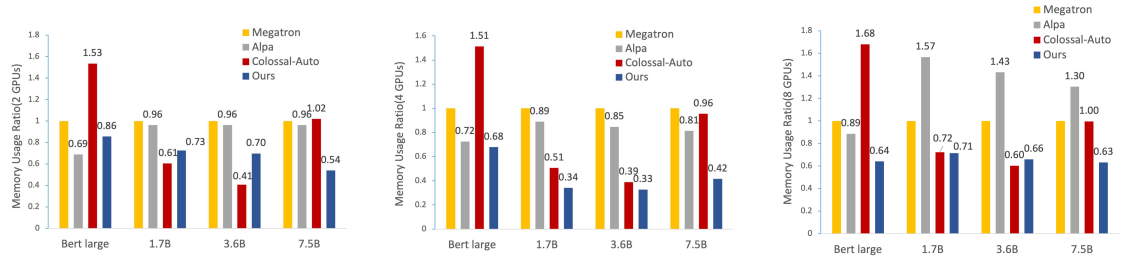

Fig. 4 Memory overhead comparison of 2 devices (a), 4 devices (b) and 8 devices (c)

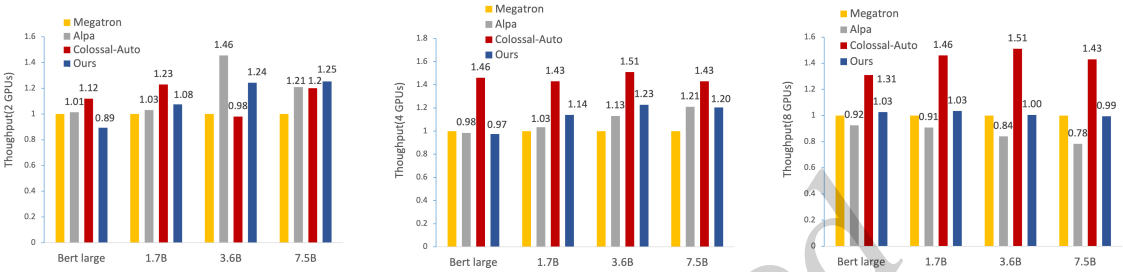

Fig. 5 Thoughput comparison of 2 devices (a), 4 devices (b) and 8 devices (c)

unedited

volume, the load balancing of memory and computing resources is more reasonable. Compared to the parallelism strategies by Megatron-LM and Alpa, our approach significantly reduces MO while achieving similar performance. Moreover, the performance advantage becomes increasingly pronounced as the model size escalates, and maximum memory savings are achieved when four GPUs are computed in parallel.

5 Conclusions

To conclude, we present an algorithm that can generate minimal memory-redundancy parallelism strategies. The algorithm was designed based on minimal memory redundancy, aiming to address the challenges of training huge models on a single Compute node. The effectiveness and efficiency of the proposed algorithm have been verified through extensive experiments and analysis.

The key findings and contributions of this research can be summarized as follows:

1. A novel algorithm is proposed that optimizes parallelism strategies, reducing memory redundancy and improving memory efficiency in large-scale DL models by leveraging the extraction of the model's computational graph and employing auxiliary graphs.

2. By conducting experiments on neural networks with various sizes within the transformer class, the proposed parallel strategy, which aims to minimize memory redundancy, as detailed in the paper, achieves significant reductions in MO of up to 67%. At the time of the performance of these experiments, the proposed strategyâĂŹs throughput was similar to those of other SOTA methods.

However, there are still areas that require further investigation and improvement:

1. While the method's effectiveness has been demonstrated on small-scale machines, the absence of experimental results on larger-scale devices limits this study.

2. Given that the choice of traditional memory optimization methodologies in strategy formulation can significantly enhance the utilization of available memory resources, insufficient integration with memory optimization techniques is also another limitation characterizing the research methodology adopted in the present study.

To summarize, this research advances automatic

parallelism by introducing a novel algorithm that demonstrates superior performance. It explores new possibilities and provides valuable insights for researchers and practitioners training large models on resource-constrained devices. The present research establishes a robust foundation for future investigations and encourages further exploration into the parallelization of training deep learning models.

References

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

Cai ZK, Yan X, Ma KH, et al., 2022. TensorOpt: exploring the tradeoffs in distributed DNN training with auto-parallelism. *IEEE Trans Parallel Distrib Syst*, 33(8):1967-1981. https://doi.org/10.1109/TPDS.2021.3132413

Chowdhery A, Narang S, Devlin J, et al., 2022. Palm: scaling language modeling with pathways. https://arxiv.org/abs/2204.02311.

Dan YH, Lei ZK, Gu YY, et al., 2023. Educhat: a large-scale language model-based chatbot system for intelligent education. https://arxiv.org/abs/2308.02773.

Devlin J, Chang MW, Lee K, et al., 2019. BERT: pre-training of deep bidirectional transformers for language understanding. Proc Conf of the 9th American Chapter of the Association for Computational Linguistics: Human Language Technologies, p.4171-4186. https://doi.org/10.18653/v1/N19-1423

Guan L, Sun T, Qiao LB, et al., 2020. Anefficient parallel and distributed solution to nonconvex penalized linear svms. *Front Inf Technol Electron Eng*, 21(4):587-603. https://doi.org/10.1631/FITEE.1800566

Harlap A, Narayanan D, Phanishayee A, et al., 2018. Pipedream: fast and efficient pipeline parallel DNN training. https://arxiv.org/abs/1806.03377.

He XB, Chen X, Guo H, et al., 2023. Scalability and efficiency challenges for the Exascale supercomputing system: practice of a parallel supporting environment on the Sunway Exascale prototype system. *Front Inf Technol Electron Eng*, 24(1):41-58. https://doi.org/10.1631/FITEE.2200412

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

Jia ZH, Lin SN, Qi CR, et al., 2018. Exploring hidden dimensions in accelerating convolutional neural networks. Proc 35th Int Conf on Machine Learning, p.2274-2283.

Krizhevsky A, Sutskever I, Hinton GE, 2012. ImageNet classification with deep convolutional neural networks. Proc 26th Annual Conf on Neural Information Processing Systems, p.1106-1114.

Lan Q, Qiao LB, Wang YJ, 2018. Stochastic extra-gradient based alternating direction methods for graph-guided regularized minimization. *Front Inf Technol Electron Eng*, 19(6):755-762. https://doi.org/10.1631/FITEE.1601771

Li SG, Liu HX, Bian ZD, et al., 2023. Colossal-AI: a unified deep learning system for large-scale parallel training. Proc 52nd Int Conf on Parallel Processing, p.766-775. https://doi.org/10.1145/3605573.3605613

Liu YL, Li SG, Fang JR, et al., 2023a. Colossal-auto: unified automation of parallelization and activation checkpoint for large-scale models. https://arxiv.org/abs/2302.02599.

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

Mo ZY, 2018. Extreme-scale parallel computing: bottlenecks and strategies. *Front Inf Technol Electron Eng*, 19(10):1251-1260. https://doi.org/10.1631/FITEE.1800421

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

Naumov M, Mudigere D, Shi HJM, et al., 2019. Deep learning recommendation model for personalization and recommendation systems. https://arxiv.org/abs/1906.00091.

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

Shazeer N, Cheng YL, Parmar N, et al., 2018. Mesh-tensorflow: deep learning for supercomputers. Proc 32nd Int Conf on Neural Information Processing Systems, p.10435-10444.

Shoeybi M, Patwary M, Puri R, et al., 2019. Megatron-LM: training multi-billion parameter language models using model parallelism. https://arxiv.org/abs/1909.08053.

Vaswani A, Shazeer N, Parmar N, et al., 2017. Attention is all you need. Proc 31st Int Conf on Neural Information Processing Systems, p.6000-6010.

Wang MJ, Huang CC, Li JY, 2019. Supporting very large models using automatic dataflow graph partitioning. Proc 14th EuroSys Conf, article 26. https://doi.org/10.1145/3302424.3303953

Zheng LM, Li ZH, Zhang H, et al., 2022. Alpa: automating inter-and intra-operator parallelism for distributed deep learning. Proc 16th USENIX Symp on Operating Systems Design and Implementation, p.559-578.

Zhuang YT, Wu F, Chen C, et al., 2017. Challenges and opportunities: from big data to knowledge in AI 2.0. *Front Inf Technol Electron Eng*, 18(1):3-14. https://doi.org/10.1631/FITEE.1601883

Unedited