Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism

Xupeng Miao[*†‡]
Peking University
xupeng.miao@pku.edu.cn

Yujie Wang[*†]
Peking University
alfredwang@pku.edu.cn

Youhe Jiang[*†]
Peking University
youhejiang@gmail.com

Chunan Shi[†]
Peking University
spirited_away@pku.edu.cn

Xiaonan Nie[†]
Peking University
xiaonan.nie@pku.edu.cn

Hailin Zhang[†]
Peking University
z.hl@pku.edu.cn

Bin Cui[†§]
Peking University
bin.cui@pku.edu.cn

ABSTRACT

Transformer models have achieved state-of-the-art performance on various domains of applications and gradually becomes the foundations of the advanced large deep learning (DL) models. However, how to train these models over multiple GPUs efficiently is still challenging due to a large number of parallelism choices. Existing DL systems either rely on manual efforts to make distributed training plans or apply parallelism combinations within a very limited search space. In this approach, we propose Galvatron, a new system framework that incorporates multiple popular parallelism dimensions and automatically finds the most efficient hybrid parallelism strategy. To better explore such a rarely huge search space, we 1) involve a decision tree to make decomposition and pruning based on some reasonable intuitions, and then 2) design a dynamic programming search algorithm to generate the optimal plan. Evaluations on four representative Transformer workloads show that Galvatron could perform automatically distributed training with different GPU memory budgets. Among all evaluated scenarios, Galvatron always achieves superior system throughput compared to previous work with limited parallelism.

PVLDB Reference Format:
Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, Bin Cui. Galvatron: Efficient Transformer Training over Multiple GPUs Using Automatic Parallelism . PVLDB, 16(3): 470 - 479, 2022.
doi:10.14778/3570690.3570697

PVLDB Availability Tag:
The source code of this research paper has been made publicly available at https://github.com/PKU-DAIR/Hetu/tree/main/tools/Galvatron.

1 INTRODUCTION

Transformer models have achieved great success in a wide range of deep learning (DL) applications in recent years, such as computer vision (CV) [11, 46], natural language processing (NLP) [6, 44, 47], graph learning [33, 51] and recommendation systems [42]. For example, many Transformer variants (e.g., BERT [10], GPT-2 [35], T5 [36]) are leading the state-of-the-art performance in various NLP tasks such as machine translation and question answering. Transformers are also applicable to image recognition (e.g, ViT [11], Swin Transformer [21]) and multimodal tasks (e.g, CLIP [34], DALL-E [38]). Due to their superior performance, Transformers are becoming increasingly important in modern web companies.

Empirical evidence indicates that scaling model parameters is an effective path towards model performance improvements [17]. For instance, the original Transformer only has millions of model parameters while GPT-2 has 1.5 billion with superior performance [35]. Such large amounts of parameters also incur high computational and memory costs even for emerging accelerators like GPUs. With the increasing model scales, building and designing Transformers demand more system optimizations, and *how to perform efficient Transformers training* is becoming more challenging.

Distributed DL systems adopt data and model parallelism to improve the training efficiency by utilizing multiple GPU devices. Data parallelism divides the large volume of input data into multiple parts and each device is only responsible for partial data [9, 22, 53]. It requires each device to store a whole model replica, suffering from large model scales. Model parallelism is a more promising direction that partitions the model from different *parallelism dimensions* and makes each device store a subset of model parameters, such as tensor parallel [29] and pipeline parallel [13, 27, 28, 50]. Various choices of the parallelism strategies lead to distinct memory consumption, communication overheads and execution efficiency.

However, directly applying these techniques to scaling Transformers is facing crucial challenges in both system efficiency and usability. Some recent advanced methods have been proposed to automatically find the parallelism strategies through the fine-grained combination of data and model parallelism for individual operators in the model. For example, OptCNN [14], FlexFlow [15, 43], Tofu [45], and TensorOpt [7] consider both data and tensor parallelism and use different search algorithms to optimize the execution plans. PipeDream [27] and DAPPLE [12] combine pipeline parallelism with data parallelism to improve the efficiency. Unfortunately, existing approaches only support limited parallelism dimensions (i.e., data parallelism and rare model parallelism dimensions) or rely on strong model and hardware configurations (i.e., expert-designed

[*]Equal contribution.
[†]School of Computer Science & Key Lab of High Confidence Software Technologies (MOE), Peking University
[‡]Computer Science Department, Carnegie Mellon University
[§]Institute of Computational Social Science, Peking University (Qingdao), China

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 16, No. 3 ISSN 2150-8097.
doi:10.14778/3570690.3570697

parallelism strategy) and result in sub-optimal performance in practice. To the best of our knowledge, there is few prior work considering the automatic parallelism for large-scale Transformers with a complex search space including multiple parallelism dimensions.

In this approach, we propose Galvatron, a novel automatic parallel training system for Transformer models over multiple GPUs. Our target is to integrate data parallelism with a variety of model parallelism dimensions, provide a rarely larger search space (compared with previous approaches), and find the optimal hybrid parallelism strategies in an efficient manner. However, such an integration brings an explosive growth of the search space and cannot be directly explored as usual. Therefore, we are interested in the following question: *How can we exploit as many parallelism dimensions as possible and efficiently explore the search space in the meanwhile?*

We study four popular parallelism paradigms in the distributed training of Transformer models, including data parallelism (DP), sharded data parallelism (SDP) [16, 37], tensor parallelism (TP) and pipeline parallelism (PP). They have distinct memory consumption and communication overheads and no single paradigm could beat the others on both sides. The search space of automatic parallelism should include the arbitrary combinations of them. Inspired by some key intuitions from our observations and analysis, we first propose a decision-tree structure to decompose the search space and perform pruning to remove the inefficient combinations. To determine the final distributed execution plan, we then propose a dynamic programming search algorithm to utilize the optimal substructure property of this problem. It is worth mentioning that the cost estimation in Galvatron considers the GPU performance slowdown from computation and communication overlapping, which has been ignored for a long time in previous approaches. We provide an implementation of Galvatron over PyTorch. Unlike existing toolbox-like systems (e.g., DeepSpeed [40], Megatron [29]) relying on users' expertise and significant tuning efforts, Galvatron's automatic parallelism only requires a few lines' modifications on the original training script. Our evaluation selects four representative Transformers, including both NLP (i.e., BERT and T5) and CV (i.e., ViT, Swin Transformer). The experimental results show that Galvatron could significantly outperform the four pure parallelisms and existing automatic parallelisms with limited dimensions (i.e., DP+TP and DP+PP) under various device memory budgets.

We summarize our contributions as follows: First, we enlarge the explored dimension of automatic parallelism for Transformer training, and introduce a novel decision-tree abstraction to decompose the large search space. Second, we design a novel parallelism optimization method to automatically find the most efficient hybrid parallelism strategy based on the estimated costs. Finally, we build Galvatron system that supports larger models' training and achieves up to 338% and 55% throughput speedups compared to state-of-the-art pure and hybrid parallelism methods respectively.

2 PRELIMINARY

2.1 Transformer Models

Transformers are first proposed to solve sequence modeling and transduction problems such as language modeling and machine translation [44]. The self-attention and point-wise feed-forward modules are the basic components in each Transformer layer. Most operations are dense algebras like matrix multiplications, resulting in huge computation costs and memory consumption.

Transformers in NLP. Different manners of using Transformer layers in NLP incur three mainly Transformer architectures, including encoder-only (for text classification, e.g., BERT and RoBERTa [20]), decoder-only (for text generation, e.g., GPT-2 and Transformer-XL [8]), and encoder-decoder (for sequence-to-sequence tasks, e.g., T5 and BART [18]). They have similar basic model components and some slight differences on the structures. For example, the decoder has an additional self-attention layer compared to the encoder. What's more, the encoder-decoder architecture combines encoders and decoders symmetrically (i.e., the same number of layers) together. These differences bring some distinct system workload characteristics in both computation and memory.

Transformers in CV. Transformers are also becoming increasingly attractive in computer vision areas. Vision Transformer (ViT) first replaces the tokens in languages with patches in images and the patches are fed to the encoder for the image classification task. Standard ViTs have a fixed number of patches and the same hidden dimension across different layers. Swin Transformer proposes a multi-stage hierarchical architecture with a shifted window-based attention to encode multi-scale patches. However, such multi-scale architectures also uneven computation and memory across layers.

2.2 Parallelism in Distributed Training

Data parallelism. Data parallelism approaches are widely used to scale up the distributed training for large input datasets. It refers to distribute the data samples across multiple workers to compute and synchronize the model updates (e.g., gradients). Each worker should maintain a replica of the model which implies that the model should be fit into the device memory. To alleviate the redundant memory consumption, DeepSpeed ZeRO [37] (also named by FSDP in FairScale [5]) has been proposed to partition the model states instead of replicating them. It is similar to model parallelism but still follows the data parallelism computation process except involving additional communications to share the model states.

Model parallelism. Model parallelism divides the model into multiple parts and each worker is only responsible for the computation of the partial model. Due to the complexity of DL model architectures, a variety of model parallelism approaches have been proposed with different model partition techniques. There are mainly two kinds of paradigms commonly used for large-scale Transformers training, including distributed tensor parallelism (TP) and layer-wise pipeline parallelism (PP). For example, Megatron-LM [29] uses TP, partitions the feed-forward and self-attention modules in Transformers to multiple devices and inserts communication operations (e.g., All-Reduce) to guarantee consistent results. GPipe [13] first proposes PP, treats each model as a sequence of layers and partitions the model into multiple composite layers across the devices. The workers are organized as a pipeline and transfer intermediate results at the partition boundaries between neighboring partitions.

Automatic parallelism. Recent approaches propose to integrate both data and model parallelism and search for better distributed training strategies. For example, FlexFlow, OptCNN, Tofu and TensorOpt consider both tensor parallelism and data parallelism. PipeDream and DAPPLE extend pipeline parallelism and enable data parallelism to replicate each pipeline stage. However,

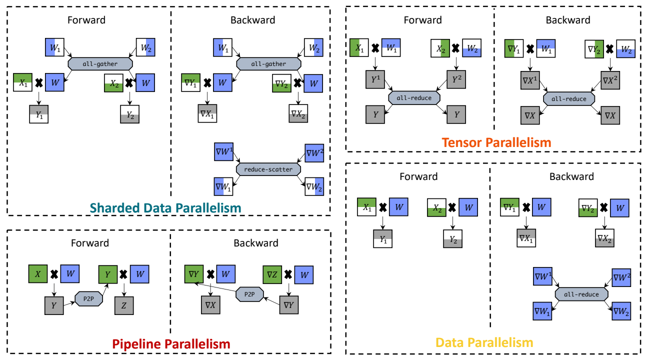


Figure 1: Illustration of different basic parallelisms in Galvatron. We use the green and gray colors to denote the input and output activations for both forward and backward computation. The model parameters and gradients are in blue.

these approaches only explore the combination of data parallelism and at most one single model parallelism dimension. Such limited decision spaces cannot generate efficient enough parallelization plan for many workloads. In fact, industrial companies have taken great efforts to explore better parallelism combinations when training large Transformers on their clusters, such as Turing-NLG [41] from Microsoft and GPT-3 [6] from OpenAI. These evidences suggest that it is necessary to design an automatic parallelization system covering as many parallelism decisions as possible, without relying on strong system tuning experience from human experts.

Task parallelism. Some approaches involve multiple training tasks simultaneously. For example, Cerebro [26] targets the model selection problem in AutoML scenarios and each task has similar model architecture with individual configurations (e.g, model size, batch size, learning rate). This line of approaches is orthogonal to our problem and they ignore the parallel training of single task.

3 GALVATRON DESIGN

The goal of Galvatron is to automatically search within the composite parallelism space and generate the optimal parallelization plan for the given Transformer model and the distributed environment. The key challenge comes from the large search space when considering multiple parallelism strategies and making fine-grained decisions for the model parameters. In this section, we first introduce the search space and then describe our detailed solutions.

3.1 Search Space Analysis

We first take an example environment with two GPUs to better illustrate the large search space, optimization target and necessary constraints. Then we extend the problem to multi-GPU cases.

3.1.1 Two-GPU Example. A Transformer model can be treated as a sequence of L layers, and each layer L_i contains a set of model parameters $\mathbf{w}_i$. Due to the back propagation, the forward computation results (i.e., activations) $\mathbf{f}_i$ should be kept inside the device memory before it calculates the gradients $\mathbf{g}_i$ in backward. The problem is to select the optimal parallelism strategy for each layer individually from a large search space, which is a composition of DP, SDP, PP, and TP. As illustrated in Figure 1, all these parallelism strategies could split the computation workloads into multiple devices. But they have distinct memory consumption and communication overheads, finally leading to different system efficiency.

Data parallelism. In DP, each GPU has a model replica and half of the input data samples. Since the size of activations is proportional to the number of data samples, each GPU only needs to store half of the forward activations. After the backward computation, the GPUs should synchronize their gradients (i.e., `all-reduce`) before updating the model, which has the sample size as model parameters.

Sharded Data parallelism. In SDP, each GPU has half of model parameters and half of the input data samples. However, it requires two times `all-gather` to collect the sharded model parameters for forward and backward computation and once `reduce-scatter` to update gradients. Since an `all-reduce` operation is equivalent to the combination of once `all-gather` and once `reduce-scatter`, the communication cost of SDP is 1.5× larger than DP.

Pipeline parallelism. In PP, the layer L_i could be placed on either GPU 0 or GPU 1, resulting in two possible memory costs: (1, 0) and (0, 1). The communication cost is mainly determined by whether the neighboring layers are on the same device. We select GPipe as the default PP in this approach and the rest (e.g., PipeDream) are left as future work. The efficiency is also affected by the pipeline bubbles (i.e., idle time), which can be reduced by splitting micro-batches.

Tensor parallelism. In TP, each GPU also has half of model parameters. Unlike SDP, TP allows each device to perform the forward computation (e.g., matrix multiplications and self-attentions) with half model. It requires to synchronize the activations with the `all-reduce` operations for both forward and backward computation. Due to the intermediate synchronization, TP has some additional replications of the activations.

3.1.2 Multi-GPU Extension. When extending to multi-GPU, the problem becomes more complicated. For example, for two nodes with 4 GPUs in total, it is easy to integrate 2-way TP within a node and 2-way PP across nodes. Alternatively, using 2-way PP within a node and 2-way DP across nodes is also possible. Moreover, there exist hundreds of candidate strategies when scaling to 8 GPUs for a single layer. For a given model, the entire search space is much larger and exponentially growing with the number of layers.

3.2 Decision-tree-based Decomposition

Considering for such as large search space, it is impossible to brute-force search all the combinations of the four parallelism paradigms within a feasible time budget. Therefore, to explore the search space more efficiently, we introduce the following key intuitions from empirical observations or theoretical analysis.

Takeaway #1. PP prefers to be applied across device "islands". Each island is a set of devices with higher-bandwidth interconnects (e.g., NVLink, PCIe) and should be in charge of a stage in the pipeline. Compared to other parallelisms, PP has much less communication overheads especially for large models. Because each stage typically has multiple layers but only requires to communicate the activations from the boundary layers. It is sensible to perform PP partition first across slower inter-island links (e.g., QPI, Ethernet).

Takeaway #2. Suppose the devices are homogeneous, these parallelism strategies prefer to divide the devices into groups with equal

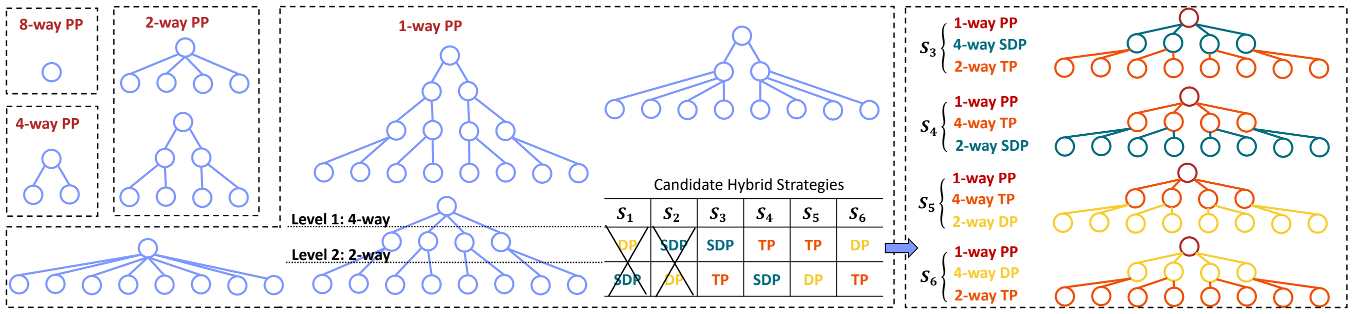

8-way PP
2-way PP
1-way PP
4-way PP
Level 1: 4-way
Level 2: 2-way
Candidate Hybrid Strategies
S_1 S_2 S_3 S_4 S_5 S_6
~~DP~~ ~~SDP~~ SDP TP TP DP
~~SDP~~ ~~DP~~ TP SDP DP TP
S_3 1-way PP 4-way SDP 2-way TP
S_4 1-way PP 4-way TP 2-way SDP
S_5 1-way PP 4-way TP 2-way DP
S_6 1-way PP 4-way DP 2-way TP

Figure 2: Illustration of the decision trees for 8 GPUs under different PP degrees (i.e., 8/4/2/1). We select one of them to introduce how to use the tree to describe the candidate hybrid parallelism strategies. We remove S_1 and S_2 as suggested by *Takeaway #3* and illustrate the left four hybrid strategies on the right part. There are 22 candidate hybrid strategies for all trees in total.

size. For example, a 2-way DP on 4 GPUs means two 2-GPU groups, rather than a single GPU and one 3-GPU group. Consequently, the optimal hybrid parallelism strategy on one group should be also consistent with those of the other groups. Note that, it could fail for PP since the model partitions may have different computation operations, resulting in different optimal parallelism strategies.

Based on the above important intuitions, we design a decision-tree to decompose the search space and represent the candidate hybrid parallelism strategies. We next present the details.

Insights Underpinning Decision-tree. We find that most existing automatic parallelism approaches only involve two parallelism dimensions (e.g., OptCNN and FlexFlow), which is easily to enumerate all possible parallelism configurations for a single layer. After involving pipeline parallelism (e.g., PipeDream), they often partition the model into different stages first and each stage is then assigned to a subset of devices. Such kind of observation suggests us to explore the hierarchical search space by utilizing a decision-tree. Another motivation is that we need the tree structure to capture the orders when applying parallelism even inside a stage. Due to the device topology and hierarchical bandwidth, it is necessary to consider the permutations of hybrid strategies since they may have different communication efficiencies.

Decision-tree construction. Given a Transformer model, Galvatron first applies PP to partition the model into multiple stages. In the meanwhile, the devices are also divided into multiple groups with the same size. As suggested by Takeaway #1, it prefers grouping between devices with higher bandwidth. For an 8-GPU scenario, Galvatron will attempt 1/2/4/8-way PP respectively. Suppose the model is partitioned evenly by PP, based on Takeaway #2, the size of the corresponding device group should be 8/4/2/1 respectively after applying PP, which directly determines the number of leaf nodes in our decision-trees. As shown in Figure 2, given the number of leaf nodes, there might exist multiple possible tree structures. We define the decision-tree construction rules as follows:

- Each decision-tree denotes a sub-search-space and its height is the number of available parallelism paradigms.
- Any one of the parallelisms cannot be applied repeatedly in different levels of a decision-tree.
- The degree of non-leaf nodes should be selected from $\{2, 4, 8, \cdots\}$.

With the above rules, the constructed trees could represent the arbitrary combinations of these parallelisms in a non-overlap manner. The guidance from *Takeaway #1* and #2 significantly helps Galvatron to avoid the unnecessary and inefficient parallelism combinations. For a single layer with 8-GPUs, it produces 34 different candidate hybrid parallelism strategies, which reduces the original combinational search space including hundreds of strategies by one order of magnitude. It could be further optimized as follows:

Takeaway #3. Using SDP is always better than integrating DP and SDP. We make a comparison with N-way DP, N-way SDP and the combination of N_1-way DP and N_2-way SDP ($N_1 \times N_2 = N$). First, SDP always has fewer model parameters than DP+SDP since $N_2 \leq N$. Second, integrating DP and SDP will lead to two rounds of communication including $2(N_1 - 1)/N_1$ for N_1-way DP and $3(N_2 - 1)/N_2$ for N_2-way SDP. Given $N_1 \times N_2 = N$, we can prove that the minimum value of its cost is still larger than that of pure SDP. Therefore, we exclude such combinations from our search space. After applying *Takeaway #3*, we could further reduce the number of candidate strategies to 22 for a single layer with 8-GPUs.

3.3 Parallelism Optimization

The target of Galvatron is to generate the optimal hybrid parallelism strategy for the input DL model with the given devices.

Problem Formulation. We define the optimization problem as follows. Given model M (with L layers) and N devices (with memory capacity of E), the object is to find the largest throughput Tpt and return the corresponding parallelism strategy, which is made up of the fine-grained layer-level parallelism strategies.

Optimization Workflow. Basically, the system throughput equals to the ratio between the batch size and the iteration time (i.e., per-batch execution time). Tuning the batch size could lead to distinct memory consumption, computation costs and communication overheads. Scaling the model training with hybrid parallelism strategies could reduce the memory consumption and enlarge the batch size. But it could also bring significant communication overheads. In other words, the highest training throughput does not have to come with the largest batch size. Therefore, we design the optimization workflow of Galvatron as illustrated in Algorithm 1. It gradually increases the explored batch size (line 2) until exceeding the device memory for all possible parallelism strategies.

Given a candidate batch size B, Galvatron then utilizes *Takeaway #1* to apply PP at first. We suppose the total number of devices N is the power of two (e.g., 4, 8, 16), which is common in dedicated GPU training clusters. So we only explore the 2-th powered PP degrees (line 4). With a P-way PP, the model is evenly partitioned into P

Algorithm 1: Galvatron Optimization

```
Input: model: $M$, #devices: $N$, device memory: $E$
Output: maximum system throughput $Tpt$
$Tpt \leftarrow 0$;
for Batch size $B \leftarrow 1, 2, ...$ do
    Time costs set $C \leftarrow \{\}$;
    for PP degree $P \in \{1, 2, 4, 8, ..., N\}$ do
        Time cost $C_P \leftarrow 0$;
        Model stages $\{M_i\}_{i=1}^{P} \leftarrow$ Pipeline_Partition($M, P$);
        Strategies set $S \leftarrow$ Construct_Decision_Tree($N/P$);
        for $i \in \{1, 2, ..., P\}$ do
            $C_P$ += Dynamic_Programming($E, M_i, B, S$);
        end
        $C$.append($C_P$);
    end
    $C_{opt} \leftarrow \min(C)$;
    if $C_{opt}$ is not $\infty$ then
        $Tpt \leftarrow \max(B/C_{opt}, Tpt)$;
    else
        return $Tpt$ ;                /* Out-Of-Memory */
    end
end
```

stages (line 6). Note that, we support several load balancing guidelines for PP partitioning, such as the number of layers/parameters, the maximum memory usage and the execution time. It is also possible to co-optimize by repeatedly interacting with the search inside each stage like Unity [43] and Alpa [52]. All devices are also evenly divided into P groups. Then we can construct the corresponding decision tree that represents the candidate hybrid parallelism strategies composed of DP, SDP and TP. After obtaining the strategies set S, we make the dynamic programming search for each model stage M_i to determine how to parallelize each layer in M_i while minimizing the execution time under the limited device memory budget E. The search algorithm returns the minimum time cost if not exceeding the device memory, which is then accumulated for all stages (line 9). Here we exclude the boundary layers' activation transferring costs in PP as they are usually quite small. By comparing the results from all possible PP degrees (line 13) and batch sizes, Galvatron obtains the maximum throughput (line 15).

Dynamic Programming Search. For a given model stage including L layers, we suppose the function $C(L, E)$ represents the total execution time of these L layers under the device memory budget E. We define $c(L, S_j)$ to denote the execution time of the L-th layer applying S_j, one of the parallelism strategies from the candidates S. Before applying the dynamic programming, we first prove that the problem follows the optimal substructure property. To obtain the minimum execution time $C(L, E)$, we clarify that the solution must contain the sub-problem solution $C(L', E')$, which represents the minimum execution time for the sub-model, i.e., first L' layers ($L' \leq L$), within a smaller device memory budget E' ($E' \leq E$). This clarification holds because if the optimal solution $C(L, E)$ does not contain a specific $C(L', E')$, we can always reduce the total execution time by replacing the sub-problem solution to $C(L', E')$. Due to the linear sequence model structure, the parallelization plan of the first L' layers will not affect the rest $L - L'$ layers given the same memory budget $E - E'$. Therefore, the problem satisfies the optimal substructure property for dynamic programming. During the search process, we start with $C(0, \cdot) = 0$ and $C(\cdot, 0) = \infty$, then we can derive the following state transition formula:

$$C(L, E) = \min_{S_j \in S}\{C(L-1, E - O(L, S_j)) + c(L, S_j) + R(L, S_i, S_j)\}, \quad (1)$$

where $O(L, S_j)$ is the memory consumption of the L-th layer applying S_j and $R(L, S_i, S_j)$ is the transformation cost between the L-th layer applying S_i and its former layer applying S_j. If two neighboring layers have different parallelism strategies, the former layer's output should be transformed to the required data layout to facilitate the next layer's parallelism. For example, if the former layer uses the combination between 2-way DP and 2-way TP and the current layer attempts to use 4-way DP, a transformation step is necessary to prepare the full model replica and the 1/4 forward activation at each device for the current layer. During the state transition process, if the memory usage exceeds the budget E, the cost function C should return infinity.

Complexity Analysis. The proposed dynamic programming search formula in E.q. (1) has a computation complexity of $O(LE|S|)$. As we can see, the size of the single layer's decision space is crucial for the entire complexity and our proposed decision-tree significantly reduces the space and makes it feasible. The number of layers L and the memory budget E also affect the complexity. For extreme cases with thousands of layers or huge memory capacity, we can further reduce the complexity by taking coarse-grained explorations, e.g., fusing multiple layers, using large memory granularity.

3.4 Cost Estimation

Galvatron provides a cost estimator to estimate the computation and communication costs and memory consumption during the optimization process. Existing approaches mainly adopt two techniques for the estimation, including *profiling* and *simulating*. In Galvatron, we take advantages from both sides and design a cost model to make the estimations cheap, efficient and accurate. Specifically, for the memory consumption, we use the shape of a tensor and its data type to calculate its memory. For the computation time, we suppose it could be estimated by the product of the batch size and the per-sample computation time. The latter could be measured by profiling real layer execution time on a single device. Note that, the Transformers are mainly composed by matrix multiplication operations, so the backward computation is usually twice of the forward computation. For the communication time, we can obtain the approximate communication time by using the size of tensor to be transferred divided by the inter-device connection's bandwidth.

With the above computation and communication cost estimations, $c(l, s)$ (i.e., the cost of a given layer l using a specific parallelism strategy $s \in S$) could be calculated by simulating the execution process. It consists of two steps, e.g., forward and backward computation. The simulation for the forward computation is simple and directly sums up the computation and communication costs (i.e., `all-gather` in SDP and `all-reduce` in TP). However, during

Table 1: Comparison with 8 GPUs under different memory constraints. The maximum throughput (samples/s) of each strategy is given, along with the corresponding batch size in the bracket, and OOM denotes Out-Of-Memory.

Memory	Strategy	BERT-Huge-32	BERT-Huge-48	ViT-Huge-32	ViT-Huge-48	T5-Large-32	T5-Large-48	Swin-Huge-32	Swin-Huge-48
8G	PyTorch DDP (DP)	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
	Megatron (TP)	OOM	OOM	16.16 (24)	10.65 (16)	OOM	OOM	13.47 (24)	8.41 (8)
	PyTorch GPipe (PP)	OOM	OOM	20.57 (56)	16.59 (32)	OOM	OOM	23.61 (40)	16.42 (24)
	FSDP/ZeRO-3 (SDP)	4.65 (8)	OOM	33.25 (64)	15.71 (40)	5.97 (8)	OOM	24.86 (48)	11.92 (32)
	DeepSpeed 3D	7.79 (8)	OOM	30.56 (40)	14.59 (16)	8.12 (8)	OOM	26.22 (32)	14.27 (16)
	Galvatron (DP+TP)	OOM	OOM	29.4 (32)	15.76 (16)	OOM	OOM	26.18 (24)	14.76 (16)
	Galvatron (DP+PP)	OOM	OOM	31.79 (48)	**20.93** (24)	**9.37** (8)	OOM	27.18 (40)	17.71 (24)
	Galvatron (ours)	**8.16** (8)	OOM	**36.58** (56)	**20.93** (24)	**9.37** (8)	OOM	**31.33** (48)	**21.64** (32)
12G	PyTorch DDP (DP)	OOM	OOM	14.22 (16)	OOM	OOM	OOM	OOM	OOM
	Megatron (TP)	5.72 (8)	OOM	16.71 (48)	10.99 (32)	5.14 (8)	OOM	13.68 (40)	9.62 (24)
	PyTorch GPipe (PP)	9.22 (8)	**6.2** (8)	25.13 (104)	16.62 (64)	9.09 (8)	**6.83** (8)	26.07 (72)	19.82 (48)
	FSDP/ZeRO-3 (SDP)	8.91 (16)	3.15 (8)	47.41 (112)	24.24 (72)	11.26 (16)	4.11 (8)	37.38 (88)	21.98 (64)
	DeepSpeed 3D	7.79 (8)	5.35 (8)	37.88 (80)	22.68 (48)	8.12 (8)	5.76 (8)	34.14 (72)	20.07 (40)
	Galvatron (DP+TP)	8.92 (8)	5.35 (8)	42.21 (64)	17.2 (32)	9.53 (8)	OOM	37.26 (56)	20.18 (32)
	Galvatron (DP+PP)	9.22 (8)	**6.2** (8)	**50.69** (72)	24.01 (56)	11.95 (16)	**6.83** (8)	35.87 (56)	21.69 (48)
	Galvatron (ours)	**11.39** (16)	**6.2** (8)	**50.69** (72)	**26.63** (72)	**14.49** (16)	**6.83** (8)	**41.69** (64)	**25.42** (64)
16G	PyTorch DDP (DP)	6.39 (8)	OOM	44.40 (64)	OOM	7.79 (8)	OOM	28.61 (40)	OOM
	Megatron (TP)	6.06 (16)	3.88 (8)	16.81 (72)	11.02 (40)	5.14 (8)	OOM	13.83 (56)	9.71 (40)
	PyTorch GPipe (PP)	12.96 (16)	6.2 (8)	25.26 (144)	17.24 (96)	9.09 (8)	6.83 (8)	28.23 (104)	20.11 (64)
	FSDP/ZeRO-3 (SDP)	12.47 (24)	6.06 (16)	59.93 (160)	32.15 (104)	14.95 (24)	7.16 (16)	49.68 (136)	26.46 (88)
	DeepSpeed 3D	8.50 (16)	5.35 (8)	41.67 (128)	25.45 (72)	11.52 (16)	5.76 (8)	37.13 (104)	24.12 (64)
	Galvatron (DP+TP)	12.59 (16)	6.19 (8)	46.02 (88)	23.97 (48)	14.52 (16)	6.84 (8)	44.65 (80)	26.51 (48)
	Galvatron (DP+PP)	13.00 (16)	6.2 (8)	54.05 (120)	28.01 (56)	14.64 (16)	6.83 (8)	44.15 (96)	25.82 (56)
	Galvatron (ours)	**15.05** (24)	**7.46** (16)	**63.25** (160)	**35.74** (104)	**16.50** (24)	**8.36** (16)	**54.06** (136)	**29.21** (72)
20G	PyTorch DDP (DP)	11.57 (16)	OOM	61.54 (112)	17.02 (32)	14.3 (16)	5.43 (8)	42.82 (80)	11.8 (24)
	Megatron (TP)	6.06 (16)	3.88 (8)	16.11 (88)	11.02 (56)	5.47 (16)	3.55 (8)	13.84 (72)	9.79 (48)
	PyTorch GPipe (PP)	13.52 (24)	7.05 (16)	28.64 (192)	17.96 (128)	9.53 (16)	8.13 (16)	29.75 (128)	20.73 (88)
	FSDP/ZeRO-3 (SDP)	17.06 (40)	7.8 (24)	63.75 (216)	38.29 (136)	17.93 (32)	7.16 (16)	55.22 (176)	32.63 (120)
	DeepSpeed 3D	8.50 (16)	5.35 (8)	43.36 (168)	27.82 (104)	13.14 (24)	7.96 (16)	40.60 (136)	26.09 (96)
	Galvatron (DP+TP)	14.65 (24)	8.05 (16)	61.54 (112)	28.69 (72)	15.35 (24)	6.84 (8)	54.87 (104)	30.59 (72)
	Galvatron (DP+PP)	15.52 (24)	8.11 (16)	61.54 (112)	34.88 (96)	17.27 (24)	**10.33** (16)	50.19 (136)	31.62 (80)
	Galvatron (ours)	**18.21** (40)	**8.95** (24)	**70.5** (152)	**41.2** (136)	**18.64** (32)	**10.33** (16)	**60.06** (144)	**37.75** (120)

the backward process, DP and SDP enable the computation and communication overlapping, which may bring estimation errors. A typical choice is to take the maximum value from the computation and communication costs (e.g., PipeDream [27]). Existing automatic parallelism approaches barely notice that modern GPUs simultaneously performing compute kernels and communication primitives (e.g., NCCL [2]) lead to slowdown for both sides. The performance degradation is mainly from the resource contention of thread warps in GPU streaming multiprocessors. We find that such contention could slow down the computation and communication by 1.3× in our evaluations, which is consistent with some recent observations [39]. By considering the overlapping slowdown, Galvatron makes more accurate estimations and better optimizations.

4 IMPLEMENTATION

Galvatron is an automatic parallel training framework especially for Transformer models (open sourced at [4]), as a part of a novel distributed DL system Hetu [23–25, 31]. We provide a simple and efficient interface to Galvatron users by making a few lines' modifications on the PyTorch training programs [19, 32].

Communication group. We implement all communication primitives with PyTorch NCCL functions. As Galvatron supports complex hybrid parallelism strategies, there could exist many communication groups among the GPUs in the generated parallelization plan. To avoid the expensive NCCL groups construction overheads, Galvatron maintains a global communication group pool which is created in advance and contains all groups that might be used.

Transformation optimization. We propose an efficient *Slice-Gather* step to perform the transformations automatically between two neighboring layers with different parallelism strategies. Given the previous layer with strategy A and the current layer with strategy B, the main idea of *Slice-Gather* is to ensure the input activations for the current layer are placed on the devices according to the requirement of strategy B, which has been extensively studied [49, 52]. There exists some special cases that the *Slice-Gather* step brings no communication costs (e.g., strategy A is 4-way TP and strategy is 4-way DP). Galvatron will automatically recognize such cases and finish the transformation without any overheads.

5 EXPERIMENTS

5.1 Experimental Setups

In this section, we compare Galvatron with 4 pure parallelism strategies implemented by the state-of-the-art systems including PyTorch DDP [19] for DP, Megatron [29] for TP, PyTorch GPipe [3] for PP, and FairScale FSDP [48] (similar to DeepSpeed ZeRO Stage-3 [37]) for SDP. We also compare with DeepSpeed 3D which is an expert-designed baseline [1] integrating DP, TP, and PP globally. Besides, we further provide two auxiliary versions of Galvatron to verify the

Table 2: Statistics of Models

Model	Layer Num	Hidden Size	Param. Num	Acti. Size/sample
BERT-Huge-32	32	1280	672M	3149.39MB
BERT-Huge-48	48	1280	987M	4657.51MB
BERT-xHuge	128	2560	10.2B	24210.05MB
ViT-Huge-32	32	1280	632M	646.5MB
ViT-Huge-48	48	1280	947M	968.59MB
ViT-xHuge	128	2560	10.1B	5313.9MB
T5-Large-32	16 Enc.+16 Dec.	1024	502M	4119.66MB
T5-Large-48	24 Enc.+24 Dec.	1024	737M	6107.75MB
Swin-Huge-32	2/2/26/2	320/640/1280/2560	701M	726.59MB
Swin-Huge-48	2/2/42/2	320/640/1280/2560	1016M	1016.8MB

training efficiency of previous automatic parallelism approaches with limited parallelism dimensions (i.e., DP+TP and DP+PP). To focus on automatic parallelism, we disable some memory optimizations (e.g., recompute [30]) and leave them as our future work. We select NLP models BERT, T5 as well as CV models ViT, Swin Transformer as our experimental models. The statistics of models are listed in Table 2. We select the Adam optimizer and use the English Wikipedia and ImageNet-1K as input datasets for them respectively. Most experiments are evaluated on a single node equipped with 8 Nvidia RTX TITAN 24 GB GPUs using PCIe 3.0. For PP, we manually tune the number of micro-batches to minimize the bubbles and estimate its costs. All results are averaged over 100 iterations.

5.2 End-to-End Comparison

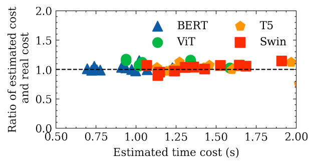
Table 1 shows the overall system throughput results of different models under different strategies with different memory constraints, along with the corresponding batch size. As we can see, under different model scales and memory budgets, Galvatron always outperforms all baselines in multiple regards. Surprisingly, on ViT models, Galvatron promotes the overall system throughput by up to 338%, and achieves a maximum of 55% acceleration compared with hybrid strategies. Similarly, on the other three models, Galvatron still achieves a maximum of 200%-334% and 28%-52% compared with single and hybrid strategies respectively.

We can also find that different models may have different preferences on the parallelism strategies. For example, under different memory budgets, BERT almost always prefers DP+PP among all baselines. Similar observations could be also found on some cases of T5. For ViT and Swin Transformer, the preferences change to SDP when increasing the memory budgets. The reason mainly comes from that NLP models have larger activation while CV models have larger model parameters, thus the latter could benefit more from sharding the model parameters across the GPUs. Here DeepSpeed 3D uses an officially suggested strategy [1] combining 2-way DP/TP/PP together. Such a fixed strategy outperforms three pure parallelisms but fails to beat SDP in most cases.

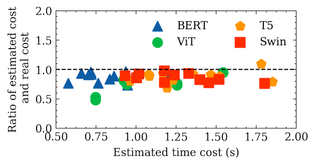
Another interesting finding is that the hybrid parallelisms like DP+TP and DP+PP may perform worse than pure SDP (e.g., ViT-Huge-32 with 8G, Swin-Huge-32 with 16G). It further indicates that existing automatic parallelism approaches focusing on limited model parallelism dimensions are suffering from these limitations.

5.3 Estimation Performance

Figure 3 demonstrates the cost estimation errors with and without considering the overlapping slowdown. It can be observed that our estimation results are very close to the real execution costs

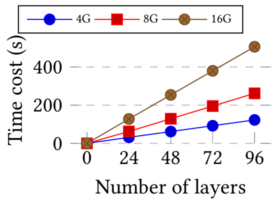


(a) w. overlapping slowdown

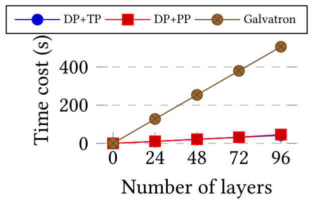


(b) w.o. overlapping slowdown

Figure 3: Estimation errors with and without considering the overlapping slowdown.



(a) Different memory budgets



(b) Galvatron vs DP+TP vs DP+PP

Figure 4: Search time costs with different numbers of layers.

for all experimental models. The average prediction error is less than 5%. However, when ignoring the slowdown, the estimations become obviously lower, resulting in an average prediction error of more than 15%, which compromises the promised efficiency of the generated execution strategy.

5.4 Optimization Efficiency

The efficiency of our dynamic programming search algorithm varies according to different number of model layers, overall strategies and memory constraints. As shown in Figure 4 (a), when the number of model layers and memory limit increase linearly, the search time of our algorithm increases linearly as excepted, only hundreds of seconds is required to generate the optimal execution plan, which is acceptable and negligible relative to the extremely long model training time. Figure 4 (b) demonstrates the impact of total parallelism dimensions on the search time, both DP+TP and DP+PP have a total of 4 alternate strategies on 8 GPUs, while Galvatron has 22 overall candidates. In this case, the search time of DP+TP and DP+PP is consistent and much less than that of Galvatron.

5.5 Optimal Parallelism Plan

We list some examples of the optimal parallelism plans suggested by Galvatron. We choose two models, BERT-Huge-32 and Swin-Huge-32, and two memory constraints, 8 GB and 12 GB to analyze.

For BERT-Huge-32 with 8 GB memory, Galvatron provides an optimal plan containing two strategies S_1^A, a combination of PP, TP and DP, and S_2^A, a combination of PP, TP, DP. This optimal plan combines all four basic parallelisms, making it possible to train this model within 8 GB memory and achieves a 75% acceleration compared to other strategies. Under the memory limitation of 12 GB, Galvatron gives a mixture of S_1^B, TP+DP, and S_2^B, TP+SDP. As we can see, Galvatron incorporates SDP and thus reduces memory costs and enlarges the batch size as well as the throughput.

For Swin-Huge-32, the optimal plans given by Galvatron is rather complex, as it has four different layers which have different strategy preference. In Swin Transformer, shallower layers have larger

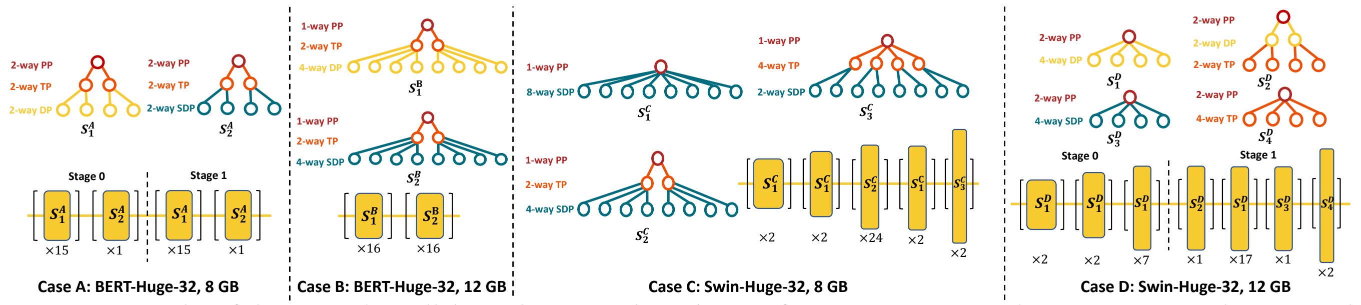


Figure 5: Examples of the optimal parallelism plans given by Galvatron for BERT-Huge-32 and Swin-Huge-32 under 8 GB and 12 GB memory budgets. Each yellow rectangle denotes an encoder layer, and its height and width represent parameter size and activation size respectively. The number $\times N$ under the rectangle means applying an strategy for consecutive N layers.

Table 3: Comparison with 16 GPUs.

Memory	Strategy	BERT-Huge-32	BERT-Huge-48	ViT-Huge-32	ViT-Huge-48
8G	PyTorch DDP (DP)	OOM	OOM	OOM	OOM
	Megatron (TP)	OOM	OOM	16.86 (32)	10.86 (16)
	PyTorch GPipe (PP)	13.79 (16)	5.88 (8)	50.70 (128)	27.96 (80)
	FSDP/ZeRO-3 (SDP)	8.95 (16)	6.12 (16)	69.48 (128)	34.92 (96)
	DeepSpeed 3D	**15.24** (16)	6.43 (8)	57.14 (64)	29.92 (40)
	Galvatron (DP+TP)	OOM	OOM	54.43 (64)	24.56 (32)
	Galvatron (DP+PP)	13.91 (16)	5.88 (8)	68.56 (128)	35.02 (72)
	Galvatron (ours)	**15.24** (16)	**8.43 (16)**	**76.74** (128)	**38.32** (88)
16G	PyTorch DDP (DP)	12.14 (16)	OOM	88.06 (128)	OOM
	Megatron (TP)	6.12 (16)	4.23 (16)	17.11 (64)	11.26 (48)
	PyTorch GPipe (PP)	23.29 (40)	12.92 (24)	69.72 (320)	50.23 (208)
	FSDP/ZeRO-3 (SDP)	30.37 (64)	11.74 (32)	123.95 (320)	61.49 (224)
	DeepSpeed 3D	23.92 (48)	13.03 (24)	91.56 (256)	53.81 (152)
	Galvatron (DP+TP)	23.01 (32)	10.50 (16)	99.22 (160)	49.82 (96)
	Galvatron (DP+PP)	23.73 (40)	13.12 (40)	115.88 (224)	61.38 (208)
	Galvatron (ours)	**32.67** (64)	**14.74** (40)	**131.15** (320)	**72.74** (208)

Table 4: Comparison with 64 GPUs.

Memory	Strategy	BERT-xHuge	ViT-xHuge
16G	PyTorch DDP (DP)	OOM	OOM
	Megatron (TP)	0.68 (3)	1.94 (12)
	PyTorch GPipe (PP)	9.74 (16)	61.95 (96)
	FSDP/ZeRO-3 (SDP)	OOM	OOM
	DeepSpeed 3D	8.44 (16)	64.91 (96)
	Galvatron (DP+TP)	1.73 (4)	5.07 (2)
	Galvatron (DP+PP)	9.74 (16)	64.83 (104)
	Galvatron (ours)	**13.77** (24)	**68.35 (136)**
32G	PyTorch DDP (DP)	OOM	OOM
	Megatron (TP)	0.77 (7)	2.11 (28)
	PyTorch GPipe (PP)	21.38 (48)	94.84 (288)
	FSDP/ZeRO-3 (SDP)	OOM	OOM
	DeepSpeed 3D	21.28 (40)	91.19 (256)
	Galvatron (DP+TP)	1.73 (4)	5.51 (68)
	Galvatron (DP+PP)	23.64 (48)	110.98 (232)
	Galvatron (ours)	**27.49** (64)	**114.55 (328)**

activation size and smaller parameter size. To reduce memory consumption and communication overhead, shallower layers prefer data parallel which splits input activations and communicates parameter gradients, while deeper layers prefer tensor parallel which splits model parameters and communicates activations.

5.6 Scalability Study

We conduct further comparisons on large clusters. We first extend our experiments to 16 Nvidia RTX TITAN GPUs over two servers connected by 100 Gb InfiniBand network. Table 3 illustrates the results on BERT and ViT models. Not surprisingly, Galvatron achieves the best performance with different memory budgets. Compared with the results on 8 GPUs, Galvatron and the hybrid parallelism methods could obtain more than 2× speedups for many cases. For example, Galvatron enlarges the batch size from 160 to 320 for ViT-Huge-32 under 16 GPUs with 16 GB memory, and the throughput increases from 63.25 to 131.15 samples per second. The 2.07× speedup comes from the flexible fine-grained layer-level parallelism strategy, which helps to reduce the communication costs and improve the training efficiency. We also manually search for the optimal DeepSpeed 3D parallelism configurations but they are still unsatisfactory. We then extend to an industrial GPU cluster including 64 Nvidia A100 GPUs, where each server has 8 GPUs equipped with NVLink and the servers are connected by 100 Gb InfiniBand network. Since the environment scale is significantly larger than before, we also increase the model sizes to 10 billion parameters (i.e., BERT-xHuge and ViT-xHuge, details are in Table 2). As we can see in Table 4, even on such a large GPUs cluster, Galvatron still outperforms these baseline methods. Besides, based on our observations, the search time costs do not exponentially grow (i.e., 2.2× and 9.2× for 16 GPUs and 64 GPUs respectively compared with 8 GPUs), which is still tolerable.

6 CONCLUSION

Large-scale Transformer training is becoming increasingly important due to its expensive training costs. Existing data and model parallelism approaches are suffering from the system efficiency problem. To address the problem, we presented Galvatron, a novel automatic parallel Transformer training system over multiple GPUs. Through the carefully designed search space decomposition and exploration algorithm, Galvatron significantly outperforms the state-of-the-art baselines on the training throughput. We hope the open source release of Galvatron will facilitate the future research directions on more challenging scenarios, e.g., heterogeneous environments and large DL models with complex and dynamic structures.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No. 61832001 and U22B2037) and PKU-Tencent joint research Lab. Bin Cui is the corresponding author.

REFERENCES

[1] 2021. DeepSpeed 3D Parallelism. https://github.com/microsoft/Megatron-DeepSpeed/blob/main/examples/pretrain_bert_distributed_with_mp.sh.

[2] 2021. NVIDIA collective communications library (NCCL). https://developer.nvidia.com/nccl.

[3] 2021. PyTorch GPipe. https://pytorch.org/docs/stable/pipeline.html.

[4] 2022. Galvatron. https://github.com/PKU-DAIR/Hetu/tree/main/tools/Galvatron.

[5] Mandeep Baines, Shruti Bhosale, Vittorio Caggiano, Naman Goyal, Siddharth Goyal, Myle Ott, Benjamin Lefaudeux, Vitaliy Liptchinsky, Mike Rabbat, Sam Sheiffer, et al. 2021. Fairscale: A general purpose modular pytorch library for high performance and large scale training.

[6] 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. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

[7] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. 2022. TensorOpt: Exploring the Tradeoffs in Distributed DNN Training With Auto-Parallelism. *IEEE TPDS* 33, 8 (2022), 1967–1981.

[8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In *ACL*. 2978–2988.

[9] 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. 2012. Large Scale Distributed Deep Networks. In *NeurIPS*. 1232–1240.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.

[11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *ICLR*.

[12] 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. 2021. DAPPLE: a pipelined data parallel approach for training large models. In *PPoPP*. ACM, 431–445.

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*. 103–112.

[14] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *ICML*, Vol. 80. PMLR, 2279–2288.

[15] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *MLSys*.

[16] Youhe Jiang, Xupeng Miao, Xiaonan Nie, and Bin Cui. 2022. OSDP: Optimal Sharded Data Parallel for Distributed Deep Learning. *ICML Hardware Aware Efficient Training (HAET) Workshop* (2022).

[17] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. (2020). arXiv:2001.08361

[18] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *ACL*. 7871–7880.

[19] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *PVLDB* 13, 12 (2020), 3005–3018.

[20] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. (2019). arXiv:1907.11692

[21] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. In *ICCV*. IEEE, 9992–10002.

[22] Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui. 2021. Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce. In *SIGMOD*. ACM, 2262–2270.

[23] Xupeng Miao, Xiaonan Nie, Hailin Zhang, Tong Zhao, and Bin Cui. 2022. Hetu: A highly efficient automatic parallel distributed deep learning system. *Sci. China Inf. Sci.* (2022). https://doi.org/10.1007/s11432-022-3581-9

[24] Xupeng Miao, Yining Shi, Hailin Zhang, Xin Zhang, Xiaonan Nie, Zhi Yang, and Bin Cui. 2022. HET-GMP: A Graph-based System Approach to Scaling Large Embedding Model Training. In *SIGMOD*. 470–480.

[25] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2022. HET: Scaling out Huge Embedding Model Training via Cache-enabled Distributed Framework. *PVLDB* 15, 2 (2022), 312–320.

[26] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *PVLDB* 13, 11 (2020), 2159–2173.

[27] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *SOSP*. 1–15.

[28] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *ICML*. 7937–7947.

[29] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *SC*. ACM, 58:1–58:15.

[30] Xiaonan Nie, Xupeng Miao, Zhi Yang, and Bin Cui. 2022. TSPLIT: Fine-grained GPU Memory Management for Efficient DNN Training via Tensor Splitting. In *ICDE*. IEEE, 2615–2628.

[31] Xiaonan Nie, Pinxue Zhao, Xupeng Miao, Tong Zhao, and Bin Cui. 2022. HetuMoE: An Efficient Trillion-scale Mixture-of-Expert Distributed Training System. *arXiv:2203.14685* (2022).

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *NeurIPS* (2019).

[33] Yun Peng, Byron Choi, and Jianliang Xu. 2021. Graph Learning for Combinatorial Optimization: A Survey of State-of-the-Art. *Data Sci. Eng.* 6, 2 (2021), 119–141.

[34] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *ICML*, Vol. 139. PMLR, 8748–8763.

[35] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[36] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *JMLR* (2020).

[37] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *SC*. IEEE/ACM.

[38] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-Shot Text-to-Image Generation. In *ICML*, Vol. 139. PMLR, 8821–8831.

[39] Saeed Rashidi, Matthew Denton, Srinivas Sridharan, Sudarshan Srinivasan, Amoghavarsha Suresh, Jade Nie, and Tushar Krishna. 2021. Enabling Compute-Communication Overlap in Distributed Deep Learning Training Platforms. In *ISCA*. IEEE, 540–553.

[40] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *SIGKDD*. 3505–3506.

[41] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zheng, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. (2022). arXiv:2201.11990

[42] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. 2019. BERT4Rec: Sequential Recommendation with Bidirectional Encoder Representations from Transformer. In *CIKM*. 1441–1450.

[43] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *OSDI*. 267–284.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*. 5998–6008.

[45] Minjie Wang, Chien-Chin Huang, and Jinyang Li. 2019. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *EuroSys*. ACM, 26:1–26:17.

[46] Hua-Peng Wei, Ying-Ying Deng, Fan Tang, Xing-Jia Pan, and Wei-Ming Dong. 2022. A Comparative Study of CNN-and Transformer-Based Visual Style Transfer. *Journal of Computer Science and Technology* 37, 3 (2022), 601–614.

[47] Yuemei Xu, Han Cao, Wanze Du, and Wenqing Wang. 2022. A Survey of Cross-lingual Sentiment Analysis: Methodologies, Models and Evaluations. *Data Sci. Eng.* 7, 3 (2022), 279–299.

[48] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. 2020. Automatic cross-replica sharding of weight update in data-parallel training. *arXiv:2004.13336* (2020).

[49] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. 2021. GSPMD: general and scalable parallelization for ML computation graphs. *arXiv:2105.04663* (2021).

[50] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R. Aberger, and Christopher De Sa. 2021. PipeMare: Asynchronous Pipeline Parallel DNN Training. In *MLSys 2021*.

[51] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. 2021. Do Transformers Really Perform Bad for Graph Representation? *arXiv:2106.05234* (2021).

[52] 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. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *OSDI*. USENIX Association, 559–578.

[53] Martin Zinkevich, M. Weimer, Alex Smola, and L. Li. 2010. Parallelized Stochastic Gradient Descent. In *NeurIPS*.