Elastic Pipeline Load Balancing for Dynamic DNNs

by

Muhammet Abdullah Soytürk

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

January 27, 2023

Elastic Pipeline Load Balancing for Dynamic DNNs

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Muhammet Abdullah Soytürk

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assoc. Prof. Didem Unat (Advisor)

_____

Assoc. Prof. Tolga Ovatman

_____

Assist. Prof. Gözde Gül Şahin

Date: _____

Dedicated to Hatice Soytürk and Şükrü Soytürk whose sacrifices made this possible.

ABSTRACT

Elastic Pipeline Load Balancing for Dynamic DNNs
Muhammet Abdullah Soytürk
Master of Science in Computer Science and Engineering
January 27, 2023

Training of dynamic models is gaining traction in DNNs as it reduces computational and memory requirements of large-scale training. Gradual pruning, one of the prominent approaches for dynamic training, prunes (or sparsifies) the parameters of a model during training. However, one of the side effects of gradual pruning is that sparsification introduces an imbalanced workload across accelerators, which in turn affects the pipeline parallelism efficiency. This work introduces DynPipe which dynamically load balances the stages of the pipeline to offset the negative performance effects of pruning. On top of load balancing dynamic models, DynPipe can dynamically pack work into fewer GPUs, while sustaining performance. DynPipe works on single nodes with multi-GPUs and also on systems with multi-nodes. Experimental results show that DynPipe can speed up the training up to 5.64% in a single node, and 8.43% in a multi-node setting, over state-of-the art solutions used in training production large language models. DynPipe is available at `https://anonymous.4open.science/r/DynPipe-1EC5`

ÖZETÇE

Yüksek Lisans Tez Başlığı
Muhammet Abdullah Soytürk
Bilgisayar Bilimleri ve Mühendisligi, Yüksek Lisans
January 27, 2023

Dinamik modellerin eğitimi, büyük ölçekli eğitimin hesaplama ve bellek gereksinimlerini azalttığı için DNN'lerde ilgi görüyor. Dinamik eğitim için öne çıkan yaklaşımlardan biri olan kademeli budama, eğitim sırasında bir modelin parametrelerini budar (veya seyreltir). Bununla birlikte, kademeli budamanın yan etkilerinden biri, seyrekleştirmenin hızlandırıcılar arasında dengesiz bir iş yükü getirmesi ve bunun da boru hattı paralellik verimliliğini etkilemesidir. Bu çalışma, budamanın olumsuz performans etkilerini gidermek için boru hattındaki yükleri dinamik olarak dengeleyen DynPipe'ı tanıtmaktadır. DynPipe, dinamik modellerde yük dengelemeye ek olarak, toplam yükü performansı düşürmeden daha az sayıda GPU'ya sığdırabilir. DynPipe, çok GPU'lu tek düğümlerde ve çok düğümlü sistemlerde çalışır. Deneysel sonuçlar, DynPipe'ın büyük dil modeli eğitiminde kullanılan son teknoloji çözümlere göre tek bir düğümde eğitimi %5,64'e ve çok düğümlü bir ortamda %8,43'e kadar hızlandırabildiğini göstermektedir. DynPipe aşağıdaki adreste mevcuttur:
https://anonymous.4open.science/r/DynPipe-1EC5

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor *Asst. Prof. Didem Unat* for her guidance in every step of the way. I thank *Mohamed Wahib* for his constant guidance throughout this research. Our discussions with the members of ParCoreLab *Ismayil Ismayilov, Mandana Bagheri Marzijarani, Javid Baydamirli, Muhammad Aditya Sasongko, Ilyas Turimbetov, Erhan Tezcan, Doğan Sağbili, Aydın Özcan*, and *Endi Merkuri* have always been a source inspiration.

I would like to thank *Beyzanur Köseoğlu, Furkan Serper, İrem Sultan Dilbaz, Hande Nur Şahin, Burak Duman*, and *Sevde Zülal Uysal* who made Istanbul a home for me during my time here. I also thank *Aladdin Demirkan, Merve Demirci*, and *Beyza Ünal* for their support and encouragement.

I am grateful to the RIKEN Center for Computational Science, Simula, National Institute of Advanced Industrial Science and Technology, European Research Council, and Koc University for providing the resources and support necessary for this research. I would like to thank the researchers at RIKEN CCS High Performance Artificial Intelligence Systems Research Team for their assistance and for providing access to the necessary materials and equipment.

Finally, I feel privileged to have my family's support for my decisions. The sacrifices they made allowed me to come to this point in my life and I can never thank them enough for that.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

BERT	Bidirectional Encoder Representations from Transformers
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DNN	Deep Neural Network
GPT	Generative Pretraining
GPU	Graphics Processing Unit
HPC	High Performance Computing
LLM	Large Language Model
MIG	Multi Instance GPU
MPI	Message Passing Interface
MoE	Mixture of Experts
NCCL	NVIDIA Collective Communications Library
PaLM	Pathways Language Model
P2P	Peer-to-Peer
SKU	Stock Keeping Unit
SpMM	Sparse Matrix-Matrix Multiplication

Chapter 1

INTRODUCTION

Motivation: The size of neural networks used to train language models has been growing exponentially since the perception of the first attention-based model [Vaswani et al., 2017]. This growth in model sizes requires larger memory and more computing power. Yet, neither the memory capacity nor the compute capability of a single accelerator increases at the same rate [Sevilla et al., 2022]. This makes it inevitable to employ model parallel training schemes with multiple accelerators. Hence, state-of-the-art HPC centers and cloud providers employ a combination of model and data parallelism to train large models.

Model parallelism can be broadly grouped into three categories: intra-layer parallelism, channel/filter parallelism, and pipeline parallelism [Kahira et al., 2021]. In intra-layer parallelism, the operators of a layer are distributed across multiple accelerators [Shoeybi et al., 2019, Narayanan et al., 2021, Smith et al., 2022]. In channel/filter parallelism [Dryden et al., 2019b, Dryden et al., 2019a], the input and/or output channels for all/some layers are distributed among accelerators. In pipeline parallelism, the most commonly used form of parallelism in large model training [Narayanan et al., 2021], consecutive layers of a model are packed into stages where each stage is typically assigned to one accelerator, and the input mini-batch is split into micro batches (chunks) to improve accelerator utilization by overlapping the computation of different chunks in a pipeline fashion [Huang et al., 2019, Harlap et al., 2018, Fan et al., 2021, Li and Hoefler, 2021].

In traditional language model training schemes, the workload for each stage is known in advance and remains static throughout the training. To reduce computational and memory costs, a new line of training schemes that introduce dynamic

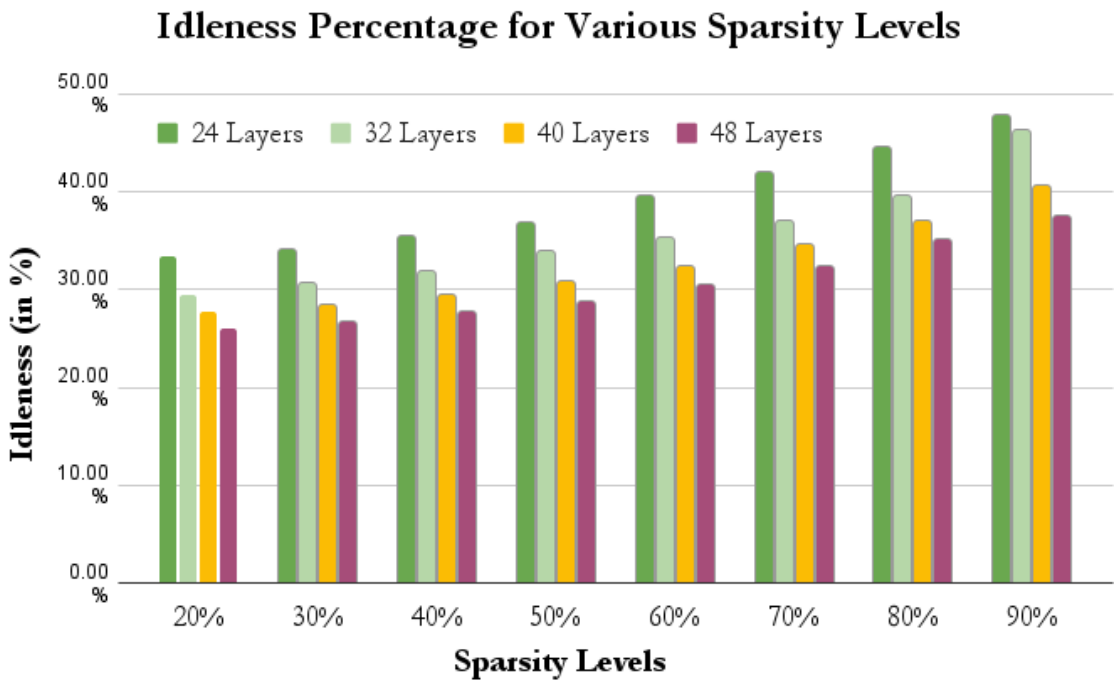

Figure 1.1: Idleness percentage of GPUs for a single iteration of BERT models [Devlin et al., 2018] with 24, 32, 40 and 48 layers running on 8 NVIDIA A100 GPUs. Idleness, due to load imbalance, increases the more we prune (sparsify) the model.

training workloads is rapidly emerging. This includes gradual pruning where the parameters of a model are pruned (i.e. sparsified) during training [Gale et al., 2019], freeze training where some of the layers of a model are frozen [Raghu et al., 2017], and gated neural networks at which the pathway through layers changes based on the input [Shazeer et al., 2017]. Other than efficiency, there are other forms and reasons for using dynamic models to improve the model attributes, such as explainability and generalization. We refer the reader to the survey by Han et al. [Han et al., 2021] on dynamic models.

One of the unforeseen side effects of these dynamic models is that they introduce a load imbalance in pipeline parallelism, which effectively decreases the throughput of training [Zhou et al., 2022, He et al., 2022]. For example, Figure 1.1 shows the maximum idleness of GPUs for BERT language models with different numbers of layers, where the models are sparsified from 20% to 90%. The load imbalance in-

creases as the model is pruned more, which reduces the pipeline utilization. Load imbalance can manifest itself as bubbles that appear in the pipeline due to a stalling accelerator that is waiting for work to pass on from its neighbor. At 90% sparsification, the idleness ratio of a pipeline can be up to 48% (Figure 1.1). Since a pipeline is as fast as its slowest stage, load balancing is crucially important for resource utilization.

Limitation of state-of-art approaches: State-of-the-art production frameworks implement a static load balance at the beginning of the training and preserve the same load distribution throughout the training. Megatron-LM [Shoeybi et al., 2019] evenly distributes all transformer layers to the accelerators. DeepSpeed [Smith, 2023] currently provides three partitioning methods to distribute the layers of a model. *Uniform* balances the number of layers, *param* balances the number of parameters in each stage, and *regex* balances the layers whose name matches the given regex. Since this approach is based on the assumption that loads of the accelerators are approximately the same throughout the training, it fails to solve the pipeline stalls introduced by dynamic models which decreases the computational efficiency.

Key insights and contributions: Considering the increase in dynamic training workloads, this work aims to reduce the pipeline stalls introduced by dynamic training. Since the computational efficiency directly affects the cost and time to train a model, we propose *DynPipe*, a pipeline dynamic load-balancing framework for dynamic models, to balance the pipeline stages during training. DynPipe dynamically load balances the stages of the pipeline whenever there is an imbalance in the workload of accelerators during training, which in turn increases the computational efficiency and results in cost savings. DynPipe includes the implementation for two different dynamic balancers. Our experiments show that DynPipe can scale in both single-node multi-GPU environments and also multi-node multi-GPU environments. DynPipe improves the end-to-end training time of the BERT model by up to 5.64% in single node and 8.43% in multi node over the state-of-the-art approaches on a gradual pruning training scheme.

DynPipe does not just improve the performance by dynamic load balancing, it also has the capability of adapting the GPU resources elastically. More specifically,

Table 1.1: Training cost estimation of several LLMs and the cost saving for %1 speedup. Note that O(10s) training runs are typically required to test and tune the hyperparamters of a production model.

Model	**Cost Est.**	**Savings w\ %1 Speedup**
GPT-3 [Brown et al., 2020]	$5M-$12M [Morgan, 2022, Li, 2022]	$50k-$120k
PaLM [Chowdhery et al., 2022]	$9M-$23M [Heim, 2022]	$90k-$230k
Chincilla [Hoffmann et al., 2022]	$2.5M-$3.1M [Morgan, 2022]	$25k-$31k

as the aggregate amount of work drops during training due to the sparsification, the load balancer re-packs the work to fewer GPUs (subject to memory capacity constraints) while sustaining performance. The GPUs that are no more required in training can then potentially be released back to the job scheduler (in environments where the job schedulers can re-acquire released resources. For instance, in single-node multi-GPP systems Nvidia Multi-Instance GPU (MIG) [Nvidia, 2023] support partitioning the node to multi-tenant; released GPUs can be returned to MIG to be allocated to other tenants. For multi-nodes, cloud schedulers are capable of acquiring released resources and reassigning to other jobs by using elastic Kubernetes [Elastic, 2023], for instance.

To the best of our knowledge, this is the first work that studies the pipeline stalls introduced by unstructured sparsity during dynamic training, and offers a solution to the issue. Table 1.1 shows the training cost estimation for several LLM models and the potential cost savings for every %1 speedup in the training time. Considering the 10s of millions of dollars reported for every single run to train LLMs (Large Language Models) [Li, 2022, Heim, 2022, Morgan, 2022], and the 10s of training runs needed to develop and tune a model, even a single-digit improvement in efficiency percentage is of significant importance. Moreover, elastically in reducing the number of resources used in training without impacting the overall training throughput provides additional savings. Finally, we emphasize that DynPipe sits as a solution on top of the pipelining and pruning schemes. In other words, DynPipe has no affect on the model accuracy, since it just redistributes the workload and does not interfere with the mechanics of the learning process.

Finally, DynPipe is designed such that the load balancing method is orthogonal to the sparsification approach. In other words, DynPipe can be used for dynamic load balancing models that are being sparsified with a different criterion, structure, schedule or even models that are dynamically changing due to a reason other than sparsification (e.g. layer freezing [Wang et al., 2022], and manufacturing variability of the computing units [Sinha et al., 2022]).

Our contributions are summarized below:

- We propose a dynamic load-balancing framework, DynPipe, to alleviate the negative effects of dynamic models on pipeline utilization.
- We show the benefits of dynamic load balancing with a gradual training scenario in both single node and multi node settings.
- We compare the effectiveness of different balancing mechanisms based on the number of parameters and layer execution times.

Limitations of the proposed approach:

This work aims to alleviate the negative effects of the pipeline bubbles introduced by the change in the workload during training, yet there are also pipeline bubbles that are inherent because of the pipeline scheduling technique itself. DynPipe was not designed to solve these inherent bubbles. We explain the difference between the inherent bubbles in the pipeline and the extra bubbles introduced by the dynamic model training in Section 2.0.2.

The components of DynPipe are configurable according to the needs of the dynamic workload but these configurations currently should be done by the user. We leave the automation of the process of finding the best configuration, based on the given dynamic model, to future work.

Chapter 2

BACKGROUND

With ever-growing Deep Neural Network (DNN) models, in terms of the number of parameters, and datasets, parallel training has become necessity. This exponential increase in in compute and memory requirements has made DNN training one of the prominent workloads in high performance computing [Zhang et al., 2022].

In a typical DNN training schedule, the workload is known in advance and remains relatively static throughout the training. For instance, there are many studies [Harlap et al., 2018, Fan et al., 2021, Li and Hoefler, 2021] that aim to improve the pipeline design, and all of them assume a static workload throughout the training. However, new research directions for dynamic neural network training have started to emerge for various reasons such as computational efficiency, generality, and adaptiveness. Han et al. [Han et al., 2021] provides additional background on why dynamic models might be preferred.

2.0.1 Neural Network Pruning

Pruning during training leads to dynamic models. Deep neural network models are over-parameterized [Denil et al., 2013]. This means that there exists sub-networks that can train to the same accuracy [Frankle and Carbin, 2018, Gale et al., 2019]. Network pruning is a sparsification procedure that removes a fraction of the parameters to achieve the same performance with a smaller network. There are three main considerations that need to be taken into account when applying network pruning: criterion, structure, and schedule of the pruning.

Pruning Criterion: Every pruning scheme needs to define a criterion to choose which parameters to prune. A non-exhaustive list of pruning criteria used in the literature includes: weight magnitude [Li et al., 2016, Renda et al., 2020], gradient

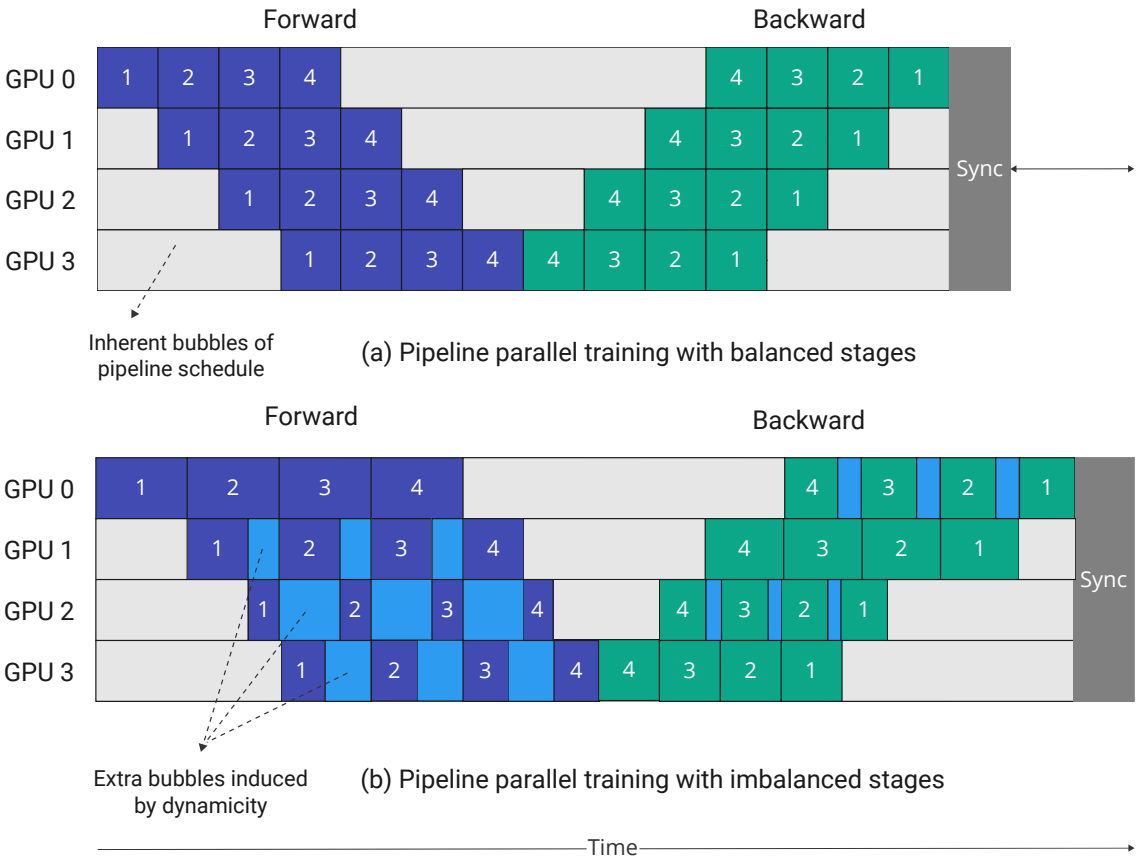

Figure 2.1: Bubble types in a pipeline with 4 microbatches. Inherent bubbles in the pipeline are shown in gray and bubbles introduced by dynamicity (e.g. sparsity) are shown in light blue.

magnitude [Cun et al., 1990, Mozer and Smolensky, 1989], Bayesian statistics-based criteria [Dai et al., 2018, Molchanov et al., 2017], and reinforcement learning based criteria [Lin et al., 2017, He et al., 2018]. These criteria can be applied either locally (i.e. considering each layer's weights separately) or globally (i.e. considering weights in all layers).

Pruning Structure: Parameters in a model can be removed in a structured or unstructured way. Structured sparsity [Kruschke and Movellan, 1991] enforces a pattern to be applied while choosing the parameters to be pruned. This can range from removing filters in a convolution layer to removing attention heads in a multi-headed attention layer. On the other hand, unstructured sparsity [Han et al., 2015] is not under the constraint of a pattern (i.e can remove parameters freely), hence,

offers a finer granularity. Even though unstructured sparsity offers better flexibility, structured sparsity is more prevalent since it is difficult to implement efficient kernels for sparse data structures in unstructured sparsity and deep learning software packages have limited support for sparse computations. However, it has been shown that the enforcement of a certain structure for the pruning of parameters can result in significant degradation in model quality compare to unstructured sparsity [Kalchbrenner et al., 2018, Elsen et al., 2020].

Pruning Schedule: After choosing the criterion and the structure of the pruning, one must decide when to prune and how often to prune. The most popular schedule in the literature consists of pruning after training is over, and then fine tune the model to recover the loss introduced by the pruning [Han et al., 2015]. Another effective approach is to remove a certain percentage of weights progressively during the training until the target sparsity is reached [Zhu and Gupta, 2017], which eliminates the fine-tuning process. There are also schedules that enforce a constant rate of sparsity throughout the training [Mocanu et al., 2018].

For a more comprehensive analysis of various sparsification procedures which are applied in deep learning, we refer the reader to Hoefler et al. [Hoefler et al., 2021]

2.0.2 Load Imbalance in Dynamic Models

To reduce computational and memory costs, training schemes that introduce dynamic training workloads have started to emerge. One of the dynamic training schemes is gradual pruning to reduce the model size. In a gradual pruning scheme, the number of parameters used in training changes during training based on a pruning strategy. If this pruning technique does not prune each layer uniformly (e.g. global magnitude pruning [Hagiwara, 1993]), the workload of each stage may be significantly different, which may introduce extra bubbles (stalls) in the pipeline [Zhu and Gupta, 2017, Frankle and Carbin, 2018, Bellec et al., 2017].

Another emerging dynamic training scheme is freeze training which relies on the idea that some layers of a network might converge faster than others, and hence can be frozen and excluded from the model during training [Shen et al., 2020]. If the

frozen layers are not evenly distributed among accelerators, this can act as a source of imbalance in the pipeline as reported by Shen et al. [Shen et al., 2020].

Networks at which the pathway through layers changes based on the input (e.g. gated neural networks) are prone to load imbalance as well. For instance, in mixture-of-experts models, most inputs might follow the same route which causes remaining experts to be underutilized [Zhou et al., 2022].

Recent work [Sinha et al., 2022] shows that a source of imbalance can also be the computing units themselves. For example, GPUs that have the same architecture and SKU (stock keeping unit) exhibit performance variations of up to 20% due to manufacturing variability and the chip's power management. This dynamic training workload might cause extra pipeline stalls due to performance differences between identical computing units.

It is important to note that there are two types of bubbles in pipeline parallelism: inherent bubbles of the pipeline schedule (e.g. bubbles at the beginning and at the end of each training step in GPipe [Huang et al., 2019]), and bubbles introduced by the dynamic models during training (e.g. bubbles introduced by pruning during training). Figure 2.1 illustrates the difference between the inherent pipeline bubbles in the pipeline schedule and the bubbles introduced by the dynamicity of the workload. We aim to reduce the latter type of bubbles by carefully redistributing the layers among stages to minimize the workload imbalance in the pipeline.

Algorithm 1 End-to-end Training of Dynamic Models with DynPipe

Input: model, train_iters, rank

Input: prune_args, balance_args, pack_args

prune_rat, prune_region, prune_freq ← prune_args
prune_idx ← 0
prune_iter ← NULL
is_load_balance, balancer = balance_args
is_pack, num_gpus_to_pack = pack_args
profile ← 0
for iter ← 0 **to** train_iters **do**
    train_step(model, profile)
    if iter **in** prune_region & iter % prune_freq == 0 **then**
        g_prune(model, prune_rat[prune_idx], rank) ▷ Algo. 2
        prune_idx += 1
        prune_iter = iter
        if is_load_balance **then**
            profile = 1
        end if
    end if
    if is_load_balance & iter == prune_iter + 1 **then**
        load_balance(model, balancer) ▷ Algo. 3
        profile ← 0
    end if
    if is_pack & iter == prune_iter + 1 **then**
        pack_workload(model, num_gpus_to_pack) ▷ Algo. 4
    end if
end for

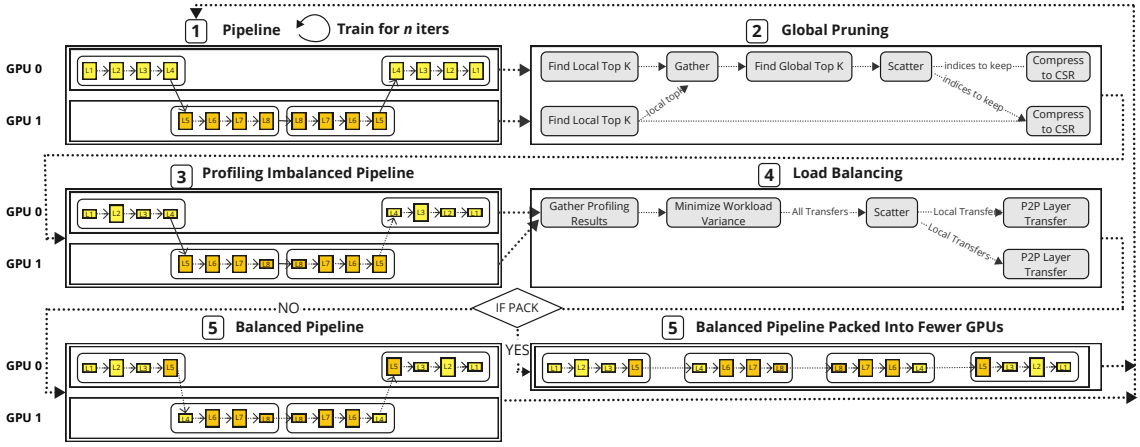

Figure 3.1: Overview of DynPipe. The flow in the figure (top to bottom) is repeated until the target sparsity is reached or training is completed. Each rectangle represents a transformer layer (i.e. encoder or decoder layer). The size of a rectangle illustrates the amount of work that the transformer layer has. (1) shows the pipeline before pruning and trains the model for n iterations (2) performs global gradual pruning, (3) profiles the pipeline to check if there is any imbalance between stages, (4) performs load balancing based on the profiling results, (5) trains the balanced pipeline until the next pruning, optionally it reduces the number of resources (GPUs) used in training by packing.

Chapter 3

DESIGN

3.0.1 Overview

In this work, we take pipeline parallelism with gradual pruning (sparsification during training) as an example case of dynamic models, for which current execution systems in DNN training are not ready to handle efficiently. Even though we show the efficiency of our load balancing system for dynamic DNNs by employing a pipeline parallel training scheme with gradual pruning, it can be a basis for expanding to other forms of dynamic models, such as freeze training.

Algorithm 1 shows the overall flow of operations for dynamic load balancing with

DynPipe. The algorithm takes as input a model, the number of training iterations, the rank of the accelerator, and several arguments for pruning, balancing, and packing the model's workloads. We start the training with the original model and train it until a user-specified pruning region (an iteration range e.g. 3000-7000) is reached (line 7-8). The model is pruned only if the training is in this pruning region. Once the training is in the pruning region, the model parameters are gradually pruned every $prune\_freq$ iteration (e.g. every 1000 iterations) where $prune\_freq$ is the frequency of pruning until the sparsity of the model reaches the given target sparsity (line 9-16). The first iteration after each pruning operation is used for profiling the time it takes to execute each layer in the pruned model and the memory usage of all accelerators in the pipeline. Next, DynPipe collects the profiling information and decides on balancing the workload by moving layers across pipeline stages based on the execution times to minimize the pipeline stalls while respecting the memory capacity of the accelerators (line 17-20) and attempts to pack the total workload into fewer number of GPUs if the packing is enabled (line 21-23). Once the training is out of the pruning region, the pipeline continues to execute with the pruned model and balanced pipeline.

Figure 3.1 also illustrates the overview of DynPipe with its steps. The implementation of individual steps of global pruning, load balancing, and packing can be found in their related sections.

3.0.2 Gradual Global Magnitude Pruning

For our pruning design, we use the gradual pruning schedule proposed in [Zhu and Gupta, 2017] which is formulated as:

$$s_t = s_f + (s_i - s_f)(1 - \frac{t - t_0}{n\Delta t})^3 \text{ for } t \in \{t_0, t_0 + \Delta t, ..., t + n\Delta t\} \tag{3.1}$$

where s_i, s_f, n, t_0, and Δt are initial sparsity, final sparsity, number of pruning steps, initial pruning step, and pruning frequency, respectively. The aim of this schedule is to prune the model rapidly in the initial pruning steps since there are

many irrelevant connections, then reduce the pruning rate as the number of parameters in the network gets smaller.

Algorithm 2 Global Pruning Algorithm

Input: model, sparsity, rank

Output: model

```
1: params ← concat_params(model)
2: k ← num_params × (1 - sparsity)
3: local_topk, local_topk_indices ← topk(abs(params), k)
4: topk_values ← gather(local_topk)
5: if rank == 0 then
6:     global_topk_indices ← topk(abs(topk_values), k)
7: end if
8: indices_to_keep ← scatter(global_topk_indices)
9: model = compress_model(model, indices_to_keep)
10: return model
```

We employed an unstructured magnitude pruning technique as opposed to a structured one since unstructured magnitude pruning typically retains better accuracy under high sparsity rates [Prasanna et al., 2020]. Unstructured magnitude pruning is applied globally (taking all parameters in the model into account) instead of locally since it has been empirically shown that global pruning yields better accuracy under high compression ratios [Blalock et al., 2020].

To our knowledge, there is no deep learning framework that supports global pruning on a distributed model at the time of this writing. Thus we implemented our own global pruning algorithm as shown in Algorithm 2. The global pruning method takes three arguments, namely the model, target sparsity, and the rank of the device. Note that each rank[1] has only its own portion of the model. First, each rank finds its own local top-k parameters in terms of magnitude (line 3). Then, rank 0 gathers the top-k parameters of each rank (line 4). When rank 0 receives all

[1]We use one MPI rank per GPU.

top-k parameters, it calculates the indices of global top-k parameters to keep (line 6), and sends the indices that belong to each rank (line 8). Finally, after each rank receives its indices to keep, they prune (discard) parameters with all other indices in their local parameters (line 9).

3.0.3 Load Balancing

A well-balanced pipeline is crucial for high performance in parallel language model training. Therefore, DynPipe collects the layer execution times after each pruning step to assess whether moving layers between accelerators would be beneficial for the pipeline. The first iteration after pruning is used for profiling the layer execution times and memory usage. Although we prefer to perform load balancing after each pruning step for this use case, the frequency of load balancing can be altered by the end user depending on the requirements of the execution.

DynPipe implements two load balancing algorithms, and can be extended to support other algorithms. The first one is the parameter-based partitioning method that balances partitions based on the number of parameters. We also implemented a version where the same algorithm is used for balancing partitions based on the layer execution times instead of the number of parameters. This algorithm with two variants is built on top of DeepSpeed's load balancing utility functions for partitioning in model parallelism [Smith, 2023]. The second algorithm is a diffusion-based algorithm that aims to minimize the variance between the workload of each rank by attempting to move layers from overloaded ranks to underloaded ranks in an iterative way. The workload can either be the layer execution times or the parameter counts as in the DeepSpeed-based algorithms. The number of iterations to decide on the final load distribution is a user-defined parameter.

Algorithm 3 shows the pseudo-code for the diffusion-based load balancing algorithm. After rank 0 gathers the loads (i.e. layer execution times or the number of parameters for each layer) from all ranks, it discovers all layer transfers between ranks by calling a diffusion re-balance function. The number of iterations to minimize the variance is an argument that can be tuned according to the workload.

For each iteration of balancing, the total load of each rank, variance, and average load are calculated (lines 3-5). Then, each rank is assigned a status: *overloaded* or *underloaded* (lines 6-7). After the status of each rank is assigned, each overloaded rank attempts to send its least loaded layer to the least loaded rank (lines 7-24). Every time an overloaded rank attempts to send a layer to an underloaded rank, new loads and variance are calculated (lines 12-14). If the new variance is smaller than the current variance and it satisfies the memory constraints of the destination rank, the transfer is accepted and added to the transfers list in the format of (source, destination, layer id) (lines 17-22). When rank 0 discovers all layer transfers from source ranks to destination ranks, it distributes the information to other ranks and the sparse format data structures, CSR, of the layers to be transferred are sent to their new destinations.

3.0.4 Packing

Workload packing is the process of merging the total workload into a smaller number of GPUs with the purpose of using the available resources more efficiently, i.e. unused resources can be released. This can be achieved with simple algorithms (in small scale) such as first-fit, best-fit, and round-robin as well as complex optimization problems (for large scale) such as ant colony optimization [Dorigo et al., 2006] or genetic algorithms [Dasgupta et al., 2013]. Workload packing aims to increase GPU utilization and reduce the overall number of GPUs employed to continue the training process. For long training schedules that are common in LLM training, workload packing can result in substantial cost savings. It may also provide improved performance due to less number of cross-GPU communication calls and smaller pipeline bubbles.

Algorithm 4 shows a simple first-fit algorithm that we used for workload consolidation. It simply iterates over all the available GPUs (lines 2-3) and checks if the combined memory usage of the two GPUs is less than the maximum memory capacity of a single GPU and the number of active GPUs is greater than the target number of GPUs *target_num_gpus* for packing (lines 4-5). If that is the case, it will

transfer all layers of the source GPU to the destination GPU (lines 7-8). Then, it updates the memory usage and the number of layers on the destination GPU accordingly. This process continues until all the available GPUs have been checked and processed. The goal of this algorithm is to reduce the number of active GPUs to the *target_num_gpus*, while also ensuring that the total memory usage remains within device limits.

3.0.5 Implementation

The DynPipe load balancing system was developed on top of Megatron-LM v3.0 [2]. Each component of DynPipe, namely pruning, load balancing and packing is implemented in a separate package for ease of use and extension.

Unstructured pruning requires a sparse storage format to compactly store, train, and transfer the pruned model. One of the most commonly used sparse formats is the compressed sparse row (CSR) format. Using a sparse matrix format requires dense matrix multiplication (DMM) operations to be converted to sparse counterparts (SpMM). Since PyTorch does not support the derivative of SpMM operation for backpropagation on a CSR tensor, we evaluated CSR-based SpMM implementations available for use on GPUs, namely cuSPARSE by Nvidia and Sputnik [Gale et al., 2020]. Figure 3.2 shows the performance of cuSPARSE and Sputnik against the dense counterpart (cuBLAS). SpMM kernel of Sputnik outperforms cuSPARSE in all sparsity levels. This is mainly because Sputnik kernels were implemented by specifically considering the deep learning workloads, unlike cuSPARSE kernels that mainly target the HPC workloads, which often have more than 99% sparsity. It is also worth noticing that Sputnik starts to outperform cuBLAS after 75% sparsity. Thus, for sparse operations, we implemented PyTorch bindings for the CUDA kernels of Sputnik [3].

The gather and scatter operations in global pruning were implemented by em-

[2] `https://github.com/NVIDIA/Megatron-LM/releases/tag/v3.0.2`

[3] The Sputnik bindings are made available at the following link: `https://anonymous.4open.science/r/Torch-Sputnik-E926/README.md`.

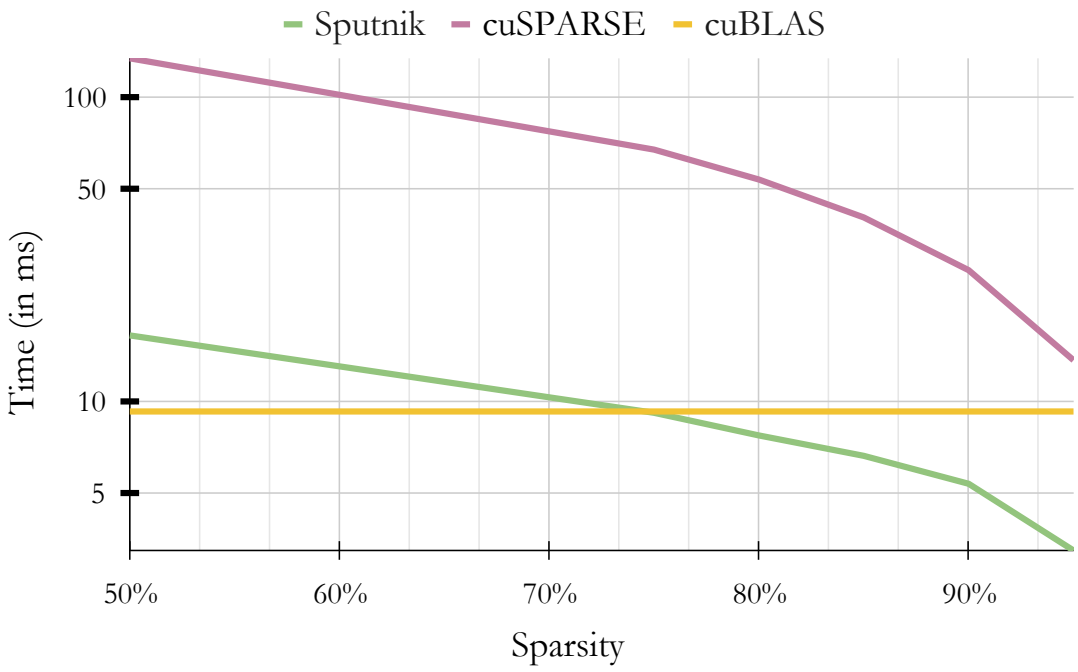

Figure 3.2: Sparse (Sputnik [Gale et al., 2020] and cuSPARSE) vs Dense (cuBLAS) matrix multiplication performance comparison for M=N=K=4096 on Nvidia A100. Starting at 75% sparsity level, sparse kernels using Sputnik have performance advantages over dense kernels.

ploying NCCL Peer-to-Peer (P2P) send-receive operations instead of collective communication operations since the sizes of the objects to be sent (local_topk) and received (indices_to_keep) from each rank are different and other ranks do not have this size information to participate in the collective call.

The necessary information for load balancers such as layer execution times and memory usage comes from the profiling iteration after each pruning iteration. The execution time profiling is implemented by extending the built-in timers of MegatronLM. The memory consumption of each pipeline stage is gathered with PyTorch's memory statistics for CUDA.

Algorithm 3 Diffusion-based Load Balancing Algorithm

Input: loads, num_ranks, max_iters, times, mem_info

Output: transfers (list)

```
transfers ← []
for iter ← 0 to max_iters do
    total_loads ← [sum(t) for t in times]
    avg_load ← average(total_loads)
    var ← variance(total_loads)
    status ← ["Overload" if l > avg_load
                  else "Underload" for l ∈ loads]
    for src ← 0 to num_ranks do
        if status[src] == "Overload" then
            dst ← get_least_loaded_rank(loads)
            lyr_idx ← get_least_loaded_layer(src, times)
            new_loads ← update_loads(src,dst,lyr_idx,loads)
            new_total_loads ← [sum(l) for l ∈ new_loads]
            new_var ← variance(new_total_loads)
            mem_req = sum(mem_info[dst]) +
                      mem_info[src][lyr_idx]
            if new_var < var && mem_req ¡ MAX_MEM then
                var ← new_var
                loads ← new_loads
                update_mem_info(src, dst, lyr_idx, mem_info)
                transfers.append((src, dst, lyr_idx))
            end if
        end if
    end for
end for
return transfers
```

Algorithm 4 Pack Layers into Fewer GPUs

Input: active_gpus, mem_usage

Input: target_num_gpus, num_layers

Output: transfers (list)

```
transfers ← []
for src in range(num_ranks) do
    for dst in range(src + 1, num_ranks) do
        if mem_usage[src] + mem_usage[dst] ¡ MAX_MEM
        && sum(active_gpus) ¿ target_num_gpus then
            active_gpus[src] = 0
            for lyr_idx in range(num_layers[src]) do
                transfers.append((src, dst, lyr_idx))
            end for
            mem_usage[dst] += mem_usage[src]
            num_layers[dst] += num_layers[src]
        end if
    end for
end for
return transfers
```

Chapter 4

EVALUATION

This section contains empirical results and an analysis of our dynamic load balancer. Experiments were mainly conducted on compute nodes each of which contains an Intel Xeon Gold 6148 CPU, and eight 40GB NVIDIA A100 GPU. The GPUs in the same node communicate through 12 NVLink3 and an NVSwitch. The compute nodes are connected by 4 Infiniband HDR (200 Gbps). We used CUDA 11.3, OpenMPI 4.0.5 and PyTorch 1.12 with NCCL 2.9.9 distributed backend.

The models used for experiments are trained on the Wikipedia dataset [Foundation, 2023]. All models used for training have a sequence length of 512, a hidden size of 1024, 16 attention heads, and the models are trained with a micro-batch size of 2 and batch size of 64 for 10000 iterations unless specified otherwise.

We have experimented with two dynamic load balancing algorithms each of which has two different configurations. The first algorithm is based on a DeepSpeed [Rajbhandari et al., 2020] API which uses a combination of a binary search and a linear probe to find the best partitioning given the parameter counts of the encoder/decoder layers. We call this algorithm *Partition by Param* throughout the evaluation. Another variation of this balancer uses encoder/decoder layer execution times as input instead of parameter counts. Hence, this variation is called *Partition by Time*. The second algorithm is a global diffusion-based load balancing algorithm that aims to minimize the variance between loads of the accelerators in an iterative way. This balancer has two variations similar to DeepSpeed, namely *Diffusion by Param* and *Diffusion by Time*.

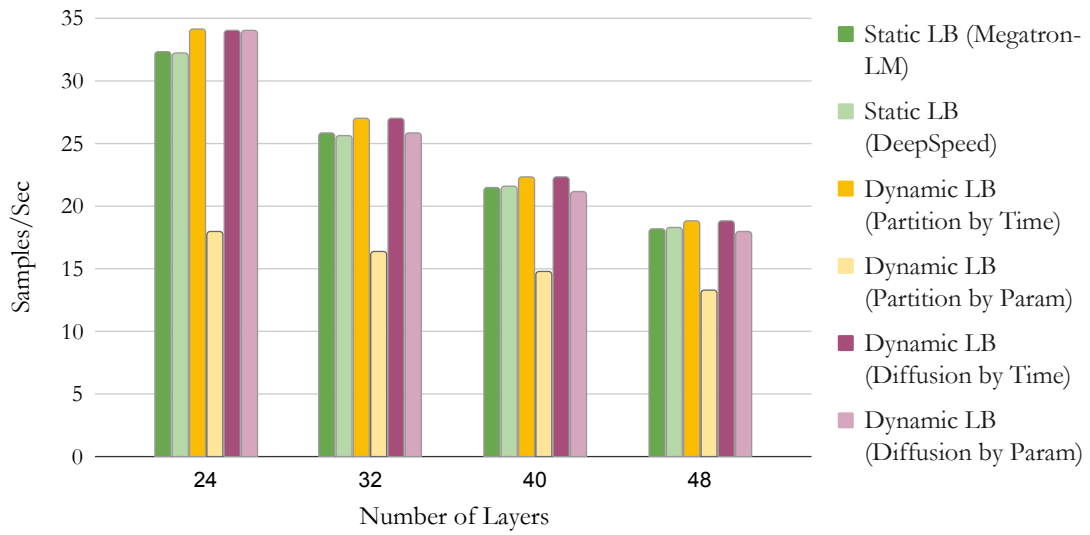

Figure 4.1: End-to-end training throughput (samples/sec) comparison for different balancer types where the target sparsity is 90% in a gradual pruning setting. Time-based dynamic load balancers outperform the baseline static load balancers and dynamic parameter-based load balancers in all model sizes. Higher is better.

4.0.1 End-to-end Training

We trained BERT models [Devlin et al., 2018] having different numbers of layers with eight A100 GPUs in a single node. The pruning region starts from iteration 3000 and continues until iteration 7000 and the model is pruned every 1000 iterations until the 90% target sparsity is reached. This corresponds to a sparsity levels of 52%, 79%, and 90% after each pruning step which is achieved with pruning steps of 52%, 56%, and 46% in order. All other hyperparameters are the same as Megatron-LM.

In Figure 4.1 we report the throughput of two state-of-art static load balancers and four dynamic load balancers. The static balancers are Megatron-LM [Shoeybi et al., 2019] which distributes the layers evenly across accelerators and DeepSpeed [Microsoft, 2023] which distributes the layers by balancing the number of parameters before the training starts. On the other hand, dynamic balancers, *Partition by Time, Partition by Param, Diffusion by Time*, and *Diffusion by Param* are redistributing the layers after each pruning step if it is necessary. While parameter-based balancers require the profiling step after the pruning step just for memory usage information, time-based balancers require it for both memory usage and the layer execution time

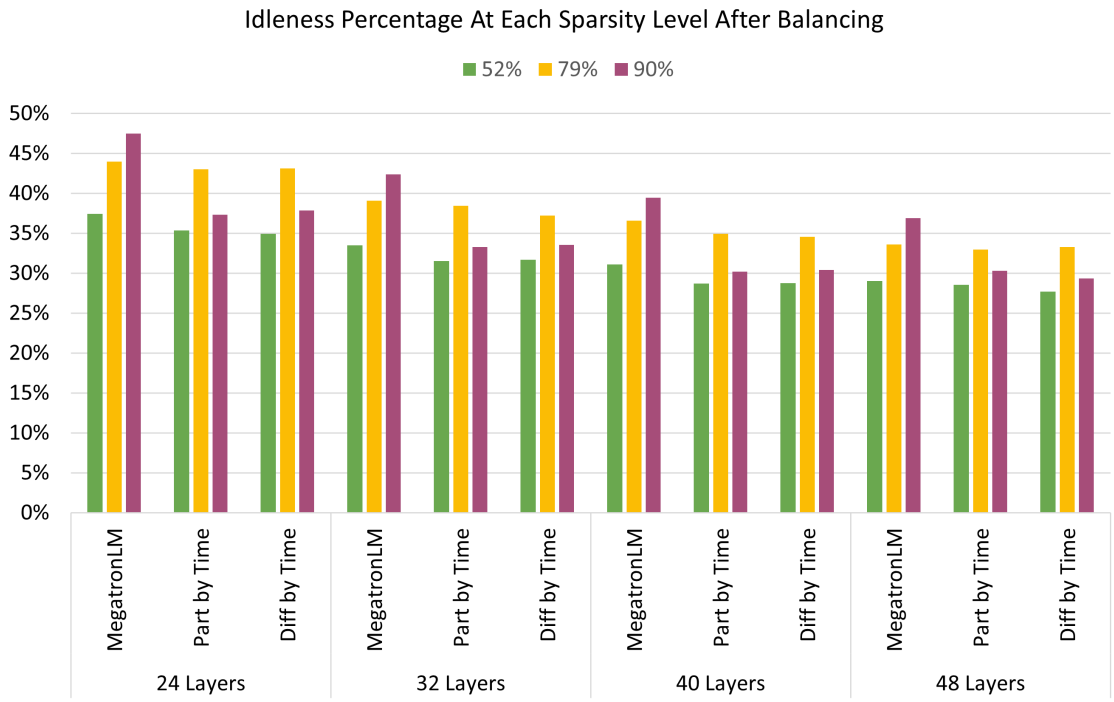

Figure 4.2: Idleness percentage of GPUs for a single iteration after each sparsity level (52%, 79%, and 90%). The target sparsity is 90%. Time-based dynamic load balancers lead to fewer pipeline bubbles. Lower is better.

information.

As seen in the figure, using layer execution time for diffusion or partitioning dynamic load balancing outperforms the parameter count-based implementations in each scale. While execution time-based dynamic balancers outperform the baseline static balancers in every scale, parameter-based dynamic balancers sometimes slow down the training (e.g. *Partition by Param* in every scale). This can be attributed to the fact that as the transformer layers are pruned, parameters in the embedding layer of the first GPU and the post-processing layers of the last GPU start to dominate the parameter counts. This leads the parameter count-based algorithms to aggressively move the transformer layers of the first and last GPU to other GPUs (more than necessary in most cases). In conclusion, time-based load-balancing algorithms result in higher throughput in all cases.

Figure 4.2 shows the maximum idleness percentage of GPUs in the pipeline after each sparsity level for the baseline model MegatronLM and time-based load bal-

Table 4.1: Time for load balancing (load balancing overhead) in terms of number of training iterations. Lower is better. Note: models train to 10,000s of iterations.

# of Layers	Partition by Time	Partition by Param	Diffusion by Time	Diffusion by Param
24	25	61	12	18
32	9	55	7	20
40	12	56	11	18
48	13	54	4	13

ancers *Diffusion by Time* and *Partition by Time*. Dynamic load balancers alleviates the pipeline bubbles in all sparsity levels and in certain cases (e.g. 24 layers at 90% sparsity), the benefit of dynamic load balancing may be up to %10 less pipeline bubbles compared to the static load balancing of MegatronLM.

4.0.2 Overhead of Load Balancing

The time spent to load balance the model is negligible in deep neural networks since they are typically trained for days if not months [Chowdhery et al., 2022, Hoffmann et al., 2022]. Table 4.1 shows the time spent while load balancing for different balancers in terms of the number of iterations. The maximum number of iterations for the diffusion algorithm is set to 5 but the experiments showed that it usually converges after two iterations. The reported load balancing times include both the load balancing decision and the actual transfer of the parameters and index data structures (i.e. row offsets and column indices in CSR format) of the layers to be sent or received. Among all balancers, *Diffusion by Time* has the least overhead. Considering the fact that the frequency of pruning is in the order of 1000s-10000s to recover the accuracy after pruning [Gale et al., 2019, Zhu and Gupta, 2017], the load balancing overhead is easily amortized. All our throughput and speedup results include the load balancing overhead unless specified otherwise.

4.0.3 Vertical Scaling

In single-node vertical scaling experiments, the number of layers in the model and the number of GPUs used in the pipeline are changed. In Table 4.2, we report throughputs of the static baseline balancer (Megatron-LM) and the best-performing dynamic load balancers from end-to-end training experiments (*Diffusion by Time* and *Partition by Time*). The dynamic load balancers speed up the training in various degrees up to 5.64% for different numbers of GPUs.

One important observation is that as the number of GPUs used in the pipeline increases, the speed-up gained by the usage of a dynamic balancer builds up. This suggests that the importance of load balancing increases as the pipeline gets deeper because the additional bubbles that are introduced by the dynamic nature of the model affect the efficiency of the pipeline more. This is important when considering the fact that the model size of large language models doubles approximately every 3.9 months [Zhang et al., 2022] which leads to deeper pipelines.

4.0.4 Packing

In the packing experiments, the training starts with 8 GPUs and after each pruning step, DynPipe attempts to pack the total workload into fewer number of GPUs by while satisfying the memory capacity constraints of the devices. Figure 4.3 reports the throughput/number of GPUs for each model size where the model is packed into 6, 4, and 2 GPUs. The 8 GPU setting for each model size serves as a baseline where there is no packing. This measurement also corresponds to the performance per dollar metric as the cost is directly proportional to the number of GPUs used in training.

It can be observed that in all model scales (e.g. 24 or 32 layers), packing can allow the training to be continued with fewer GPUs which may result in significant cost savings. For example, in Figure 4.3, reducing the GPU count from 6 to 4 results in almost the same throughput while the resource usage cost is reduced by %50 for 32 layer case. The benefits of packing are not limited to the cost savings. For instance, packing the number of GPUs to 6 from 8 in 24 layer setting also increases

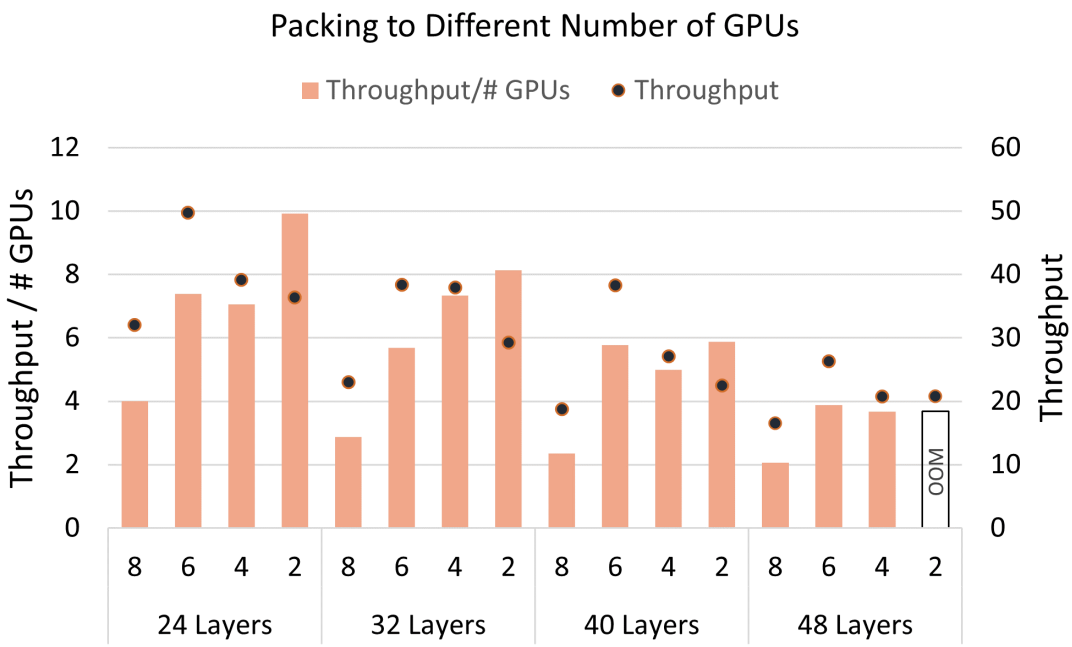

Figure 4.3: Packing the workload into fewer GPUs as the model gets smaller due to gradual pruning. The target sparsity is 90%. Left y-axis: throughput/#GPUs. Right y-axis: throughput (tokens/second). Higher is better.

the throughput which results in faster training time. In other words, packing the workload into fewer GPUs after pruning may lead to faster or comparable training time with fewer resources.

4.0.5 Multi-node Weak Scaling

For multi-node weak scaling experiments, we trained the BERT models having different numbers of layers and batch sizes on up to 8 nodes each of which contains a single A100 GPU (one GPU/node). The pruning region starts from iteration 30 and continues until iteration 70 and the model is pruned every 10 iterations until the 90% target sparsity is reached. The pruning and load balancing overheads are excluded from the measurements since the number of iterations to do this scaling experiment is not sufficient enough to amortize the overheads; in actual training (1000s to 10,000s iterations) the pruning and load balancing overheads would be negligible (Table 4.1).

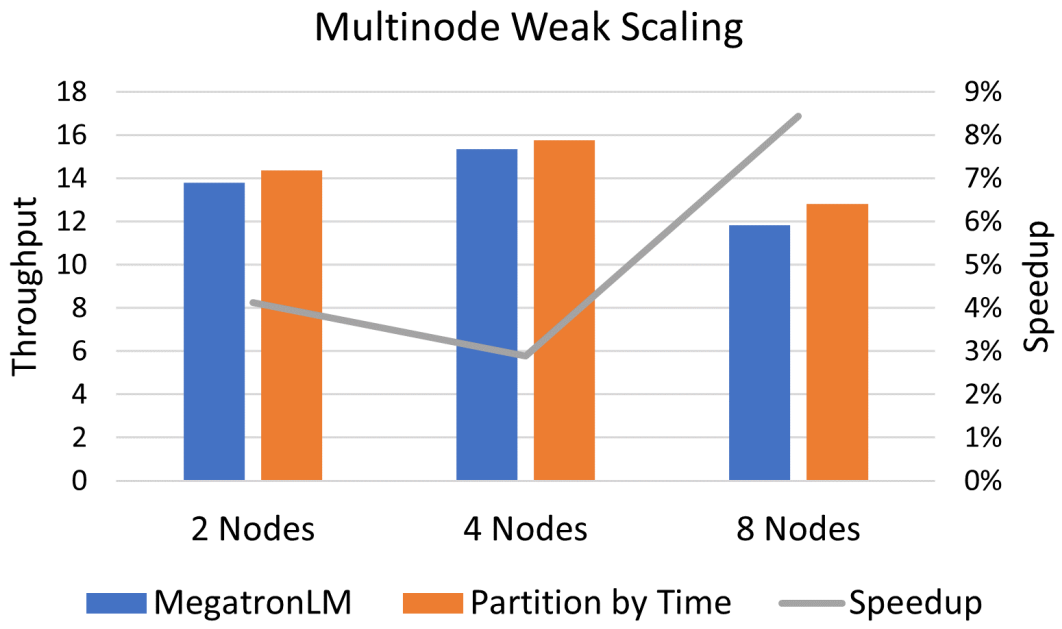

Figure 4.4: Weak scaling throughput (tokens/sec) comparison of baseline static load balancing with MegatronLM and dynamic load balancing with *Partition by Time* algorithm of DynPipe. Left y-axis: throughput. Right y-axis: speed up of *Partition by Time* over MegatronLM. Higher is better.

The settings that are used for multi-node experiments is listed in Table 4.3. As we increase the number of GPUs and layers, we also increase the batch size to fix the number of micro batches to four times the number of GPUs in the pipeline, as suggested in [Huang et al., 2019] to achieve good pipeline utilization.

Figure 4.4 shows that the pipeline that is dynamically balanced with *Partition by Time* algorithm of DynPipe reaches higher throughputs in all scales and it provides speed up over baseline MegatronLM up to 8.43%.

4.0.6 Dynamic Minibatch/Microbatch Size

In cases where the total load of the pipeline decreases such as gradual sparsification and freeze training, carefully changing the minibatch and microbatch size according to the needs of the new pipeline after load balancing may increase the efficiency of the training. For instance, GPipe [Huang et al., 2019] suggests the number of micro batches to be greater than four times the number of GPUs in the pipeline

for optimal overlapping. Since the packing decreases the number of GPUs in the pipeline, adjusting the number of micro batches in the pipeline after packing could be beneficial. In addition, minibatch size can be increased after the pruning operations since the memory requirement for execution is less after the pruning. DynPipe currently does not support this feature.

Table 4.2: Vertical scaling experiments show the throughputs (samples/sec) of baseline Megatron-LM, and time-based algorithms, namely *Diffusion by Time* and *Partition by Time* where the target sparsity is 90%. The speed is calculated for the best-performing balancer in each case. The benefits of dynamic load balancing increase as the number of GPUs in the pipeline increases.

# Layers	# GPUs	Megatron LM	Diff by Time	Part by Time	Speed Up
24	2	11.58	11.75	**11.75**	1.50%
	4	20.23	**20.86**	20.82	3.12%
	8	32.25	33.94	**34.07**	5.64%
32	2	8.83	**8.96**	8.93	1.49%
	4	15.69	16.06	**16.13**	2.81%
	8	25.81	26.90	**26.93**	4.36%
40	2	7.14	7.24	**7.25**	1.53%
	4	12.84	13.11	**13.11**	2.06%
	8	21.43	22.26	**22.30**	4.06%
48	2	OOM	OOM	OOM	OOM
	4	10.89	**11.09**	11.08	1.77%
	8	18.13	**18.72**	18.71	3.30%

Table 4.3: Experiment settings for weak scaling

# GPUs	# Layers	# Microbatches
2	12	8
4	24	16
8	48	32

Chapter 5

RELATED WORK

5.0.1 Load Balancing Model-Parallel Deep Neural Networks

Layer-wise load balancing

Layer-wise balancing techniques work on layer granularity instead of operators. DeepSpeed [Microsoft, 2023] offers three partitioning methods to balance the workload of stages: parameters, uniform, and regex. While the parameters method is trying to balance the number of parameters in each stage, the uniform aims to distribute the layers evenly. Regex only distributes the layers that match the given regex (e.g. transformer layers). Similar to the parameters method of DeepSpeed, He et al. [He et al., 2021] balance the stages based on the number of parameters in each stage. Narayanan et al. [Narayanan et al., 2021] assign each stage the same number of transformer layers to balance the load. None of the aforementioned studies use the actual execution time of the layers to decide on the distribution of layers. DynPipe supports DeepSpeed's partitioning scheme with both parameters and layer execution times to guide load balancing, as well as a diffusion-based load balancing algorithm out of the box.

Load balancing via graph partitioning

Graph partitioning-based load balancing schemes find atomic operations in the model and consider them as nodes in a directed acyclic graph (DAG). Edges in the graph represent the dependencies between operations. Tanaka et al. [Tanaka et al., 2021] partition the DAG in three phases at which they first find atomic operations, then group these operations into blocks according to their computation times, and finally, they combine blocks into final partitions by using a dynamic programming-based algorithm. Qararyah et al. [Qararyah et al., 2021] create disjoint clusters from

the nodes of the graph by finding critical paths and mapping these clusters to devices based on a mapping algorithm that takes both critical-communication minimization and temporal load balancing into account. Both studies perform profiling before the actual training and partition the graph once.

Load balancing in Mixture of Experts Models

The mixture of experts (MoE) [Jacobs et al., 1991] models contain many sub-networks (experts) where a router allocates inputs to top-k experts. At scale, experts are distributed across devices. [Lepikhin et al., 2020] defines an expert's capacity to limit the maximum number of tokens that can be processed by an expert to achieve workload balance. [Fedus et al., 2021] routes each token to only one expert and uses the same expert capacity restriction. Lewis et al. [Lewis et al., 2021] employ an auction algorithm [Bertsekas, 1992] to solve the token-to-expert assignment problem. This line of work is different from ours in the sense that their aim is to balance workload in the feed-forward network while our work aims to balance all layers of the transformer model.

5.0.2 Packing

In dynamic neural network models, packing the total workload into fewer number accelerators can provide significant cost-saving benefits. PipeTransformer [He et al., 2021] offers an elastic pipelining system for freeze training where some of the layers of the model are frozen during the training. PipeTransformer packs the remaining active layers into fewer GPUs and creates pipeline replicas if possible. When PipeTransformer receives a notification for layer freezing, it attempts to divide the number of GPUs by 2 subject to the memory capacity constraints. On the other hand, our work DynPipe can pack to an arbitrary number of GPUs specified by the user. Another difference between the packing mechanism of DynPipe and PipeTransformer is that PipeTransformer uses the parameter size as a proxy to estimate the memory usage while DynPipe uses the actual memory usage from the profiling step before load balancing. Finally, PipeTransformer is only capable of packing layers to

fewer GPUs, and not load balancing. DynPipe, on top of being capable of packing when deemed beneficial, it can also redistribute the workload to achieve a better load balance.

5.0.3 Dynamic Pruning

Model pruning is a fast-paced research area. Since the optimization problem has many dimensions, there are many approaches to prune a model. We mainly focus on the schedule of the pruning rather than the decision of how to prune (e.g. magnitude pruning, variational dropout etc.) and what kind of structure (e.g. unstructured pruning, structured pruning) to be applied while pruning.

One of the commonly used sparsification technique is sparsification during training (i.e. gradual pruning) where the pruning starts before the model is trained until convergence. While some studies [Wortsman et al., 2019, Lin et al., 2020] use a binary mask to specify whether a parameter is pruned, which enables them to apply better weight regrowth or selection, others [Gale et al., 2020] delete the pruned parameters to reduce the memory usage and number of operations. There are also many works on how fast to prune. For instance, Zhu and Gupta [Zhu and Gupta, 2017] prune the model rapidly in the first pruning steps when there are many abundant parameters in the model, and then reduce the pruning ratio as the number of parameters in the model are getting less and less. Dai et al. [Dai et al., 2019] employ a three phase schedule (birth-brain, baby-brain, and adult-brain) similar to the human brain development. Mostafa et al. [Mostafa and Wang, 2019] uses magnitude pruning as criterion to prune the parameters and regrows parameters to comply with the training budget.

Chapter 6

CONCLUSION

DynPipe offers a load-balancing system for dynamic models where the loads of the accelerators change during training. DynPipe provides better load balance among stages than state-of-the-art static load balancing approaches which results in better efficiency and faster end-to-end training time. Empirical results for large language models with a gradual pruning training show that DynPipe significantly improves the training throughput over the counterparts. We foresee that dynamic models will be more prominent in the future and dynamic load distribution will be of utmost importance.

BIBLIOGRAPHY

[Bellec et al., 2017] Bellec, G., Kappel, D., Maass, W., and Legenstein, R. (2017). Deep rewiring: Training very sparse deep networks. *arXiv preprint arXiv:1711.05136*.

[Bertsekas, 1992] Bertsekas, D. P. (1992). Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications*, 1(1):7–66.

[Blalock et al., 2020] Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., and Guttag, J. (2020). What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146.

[Brown et al., 2020] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

[Chowdhery et al., 2022] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. (2022). Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

[Cun et al., 1990] Cun, Y. L., Denker, J. S., and Solla, S. A. (1990). *Optimal Brain Damage*, page 598–605. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Dai et al., 2018] Dai, B., Zhu, C., Guo, B., and Wipf, D. (2018). Compressing neural networks using the variational information bottleneck. In *International Conference on Machine Learning*, pages 1135–1144. PMLR.

[Dai et al., 2019] Dai, X., Yin, H., and Jha, N. K. (2019). Nest: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers*, 68(10):1487–1497.

[Dasgupta et al., 2013] Dasgupta, K., Mandal, B., Dutta, P., Mandal, J. K., and Dam, S. (2013). A genetic algorithm (ga) based load balancing strategy for cloud computing. *Procedia Technology*, 10:340–347. First International Conference on Computational Intelligence: Modeling Techniques and Applications (CIMTA) 2013.

[Denil et al., 2013] Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and De Freitas, N. (2013). Predicting parameters in deep learning. *Advances in neural information processing systems*, 26.

[Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[Dorigo et al., 2006] Dorigo, M., Birattari, M., and Stutzle, T. (2006). Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39.

[Dryden et al., 2019a] Dryden, N., Maruyama, N., Benson, T., Moon, T., Snir, M., and Essen, B. V. (2019a). Improving strong-scaling of CNN training by exploiting finer-grained parallelism. In *IPDPS*, pages 210–220. IEEE.

[Dryden et al., 2019b] Dryden, N., Maruyama, N., Moon, T., Benson, T., Snir, M., and Essen, B. V. (2019b). Channel and filter parallelism for large-scale CNN training. In *SC*, pages 10:1–10:20. ACM.

[Elastic, 2023] Elastic (2023). Elastic cloud on kubernetes (eck). [Retrieved 22 January 2023].

[Elsen et al., 2020] Elsen, E., Dukhan, M., Gale, T., and Simonyan, K. (2020). Fast

sparse convnets. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 14629–14638.

[Fan et al., 2021] Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., et al. (2021). Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445.

[Fedus et al., 2021] Fedus, W., Zoph, B., and Shazeer, N. (2021). Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity.

[Foundation, 2023] Foundation, W. (2023). Wikimedia downloads.

[Frankle and Carbin, 2018] Frankle, J. and Carbin, M. (2018). The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*.

[Gale et al., 2019] Gale, T., Elsen, E., and Hooker, S. (2019). The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*.

[Gale et al., 2020] Gale, T., Zaharia, M., Young, C., and Elsen, E. (2020). Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE.

[Hagiwara, 1993] Hagiwara, M. (1993). Removal of hidden units and weights for back propagation networks. In *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, volume 1, pages 351–354 vol.1.

[Han et al., 2015] Han, S., Pool, J., Tran, J., and Dally, W. (2015). Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28.

[Han et al., 2021] Han, Y., Huang, G., Song, S., Yang, L., Wang, H., and Wang, Y. (2021). Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

[Harlap et al., 2018] Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. (2018). Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*.

[He et al., 2021] He, C., Li, S., Soltanolkotabi, M., and Avestimehr, S. (2021). Pipetransformer: Automated elastic pipelining for distributed training of transformers. *arXiv preprint arXiv:2102.03161*.

[He et al., 2022] He, J., Zhai, J., Antunes, T., Wang, H., Luo, F., Shi, S., and Li, Q. (2022). Fastermoe: Modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 120–134, New York, NY, USA. Association for Computing Machinery.

[He et al., 2018] He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., and Han, S. (2018). Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)*, pages 784–800.

[Heim, 2022] Heim, L. (2022). Estimating palm's training cost.

[Hoefler et al., 2021] Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. (2021). Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *The Journal of Machine Learning Research*, 22(1):10882–11005.

[Hoffmann et al., 2022] Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., et al. (2022). Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

[Huang et al., 2019] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. (2019). Gpipe: Efficient training of

giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32.

[Jacobs et al., 1991] Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87.

[Kahira et al., 2021] Kahira, A. N., Nguyen, T. T., Bautista-Gomez, L., Takano, R., Badia, R. M., and Wahib, M. (2021). An oracle for guiding large-scale model/hybrid parallel training of convolutional neural networks. In *HPDC*, pages 161–173. ACM.

[Kalchbrenner et al., 2018] Kalchbrenner, N., Elsen, E., Simonyan, K., Noury, S., Casagrande, N., Lockhart, E., Stimberg, F., Oord, A., Dieleman, S., and Kavukcuoglu, K. (2018). Efficient neural audio synthesis. In *International Conference on Machine Learning*, pages 2410–2419. PMLR.

[Kruschke and Movellan, 1991] Kruschke, J. and Movellan, J. (1991). Benefits of gain: speeded learning and minimal hidden layers in back-propagation networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(1):273–280.

[Lepikhin et al., 2020] Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. (2020). Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*.

[Lewis et al., 2021] Lewis, M., Bhosale, S., Dettmers, T., Goyal, N., and Zettlemoyer, L. (2021). Base layers: Simplifying training of large, sparse models. In *International Conference on Machine Learning*, pages 6265–6274. PMLR.

[Li, 2022] Li, C. (2022). Openai's gpt-3 language model: A technical overview.

[Li et al., 2016] Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. (2016). Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*.

[Li and Hoefler, 2021] Li, S. and Hoefler, T. (2021). Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14.

[Lin et al., 2017] Lin, J., Rao, Y., Lu, J., and Zhou, J. (2017). Runtime neural pruning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 2178–2188, Red Hook, NY, USA. Curran Associates Inc.

[Lin et al., 2020] Lin, T., Stich, S. U., Barba, L., Dmitriev, D., and Jaggi, M. (2020). Dynamic model pruning with feedback. *arXiv preprint arXiv:2006.07253*.

[Microsoft, 2023] Microsoft (2023). Microsoft/deepspeed: A deep learning optimization library that makes distributed training and inference easy, efficient, and effective.

[Mocanu et al., 2018] Mocanu, D. C., Mocanu, E., Stone, P., Nguyen, P. H., Gibescu, M., and Liotta, A. (2018). Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 9(1):1–12.

[Molchanov et al., 2017] Molchanov, D., Ashukha, A., and Vetrov, D. (2017). Variational dropout sparsifies deep neural networks. In *International Conference on Machine Learning*, pages 2498–2507. PMLR.

[Morgan, 2022] Morgan, T. P. (2022).

[Mostafa and Wang, 2019] Mostafa, H. and Wang, X. (2019). Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. In *International Conference on Machine Learning*, pages 4646–4655. PMLR.

[Mozer and Smolensky, 1989] Mozer, M. C. and Smolensky, P. (1989). *Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment*, page 107–115. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Narayanan et al., 2021] Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. (2021). Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15.

[Nvidia, 2023] Nvidia (2023). Nvidia multi-instance gpu (mig). [Retrieved 18 January 2023].

[Prasanna et al., 2020] Prasanna, S., Rogers, A., and Rumshisky, A. (2020). When bert plays the lottery, all tickets are winning. *arXiv preprint arXiv:2005.00561*.

[Qararyah et al., 2021] Qararyah, F., Wahib, M., Dikbayır, D., Belviranli, M. E., and Unat, D. (2021). A computational-graph partitioning method for training memory-constrained dnns. *Parallel computing*, 104:102792.

[Raghu et al., 2017] Raghu, M., Gilmer, J., Yosinski, J., and Sohl-Dickstein, J. (2017). Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. *Advances in neural information processing systems*, 30.

[Rajbhandari et al., 2020] Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. (2020). Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE.

[Renda et al., 2020] Renda, A., Frankle, J., and Carbin, M. (2020). Comparing rewinding and fine-tuning in neural network pruning. *arXiv preprint arXiv:2003.02389*.

[Sevilla et al., 2022] Sevilla, J., Heim, L., Ho, A., Besiroglu, T., Hobbhahn, M., and Villalobos, P. (2022). Compute trends across three eras of machine learning. *arXiv preprint arXiv:2202.05924*.

[Shazeer et al., 2017] Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*.

[Shen et al., 2020] Shen, S., Baevski, A., Morcos, A. S., Keutzer, K., Auli, M., and Kiela, D. (2020). Reservoir transformers. *arXiv preprint arXiv:2012.15045*.

[Shoeybi et al., 2019] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. (2019). Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

[Sinha et al., 2022] Sinha, P., Guliani, A., Jain, R., Tran, B., Sinclair, M. D., and Venkataraman, S. (2022). Not all gpus are created equal: Characterizing variability in large-scale, accelerator-rich systems. *arXiv preprint arXiv:2208.11035*.

[Smith, 2023] Smith, S. (2023). Pipeline parallelism.

[Smith et al., 2022] Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., et al. (2022). Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*.

[Tanaka et al., 2021] Tanaka, M., Taura, K., Hanawa, T., and Torisawa, K. (2021). Automatic graph partitioning for very large-scale deep learning. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1004–1013. IEEE.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

[Wang et al., 2022] Wang, Y., Sun, D., Chen, K., Lai, F., and Chowdhury, M. (2022). Efficient dnn training with knowledge-guided layer freezing. *arXiv preprint arXiv:2201.06227*.

[Wortsman et al., 2019] Wortsman, M., Farhadi, A., and Rastegari, M. (2019). Discovering neural wirings. *Advances in Neural Information Processing Systems*, 32.

[Zhang et al., 2022] Zhang, H., Zheng, L., Li, Z., and Stoica, I. (2022). Welcome to the "big model" era: Techniques and systems to train and serve bigger models.

[Zhou et al., 2022] Zhou, Y., Lei, T., Liu, H., Du, N., Huang, Y., Zhao, V., Dai, A., Chen, Z., Le, Q., and Laudon, J. (2022). Mixture-of-experts with expert choice routing. *arXiv preprint arXiv:2202.09368*.

[Zhu and Gupta, 2017] Zhu, M. and Gupta, S. (2017). To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*.