Balanced and Elastic End-to-end Training of Dynamic LLMs

Mohamed Wahib
RIKEN Center for Computational Science
Kobe, Japan
moahmed.attia@riken.jp

Muhammed Abdullah Soyturk
Koç University
Istanbul, Turkey
muhammetabdullahsoyturk@gmail.com

Didem Unat
Koç University
Istanbul, Turkey
dunat@ku.edu.tr

Abstract

To reduce the computational and memory overhead of Large Language Models, various approaches have been proposed. These include a) Mixture of Experts (MoEs), where token routing affects compute balance; b) gradual pruning of model parameters; c) dynamically freezing layers; d) dynamic sparse attention mechanisms; e) early exit of tokens as they pass through model layers; and f) Mixture of Depths (MoDs), where tokens bypass certain blocks. While these approaches are effective in reducing overall computation, they often introduce significant workload imbalance across workers. In many cases, this imbalance is severe enough to render the techniques impractical for large-scale distributed training, limiting their applicability to toy models due to poor efficiency.

We propose an autonomous dynamic load balancing solution, DynMo, which provably achieves maximum reduction in workload imbalance and adaptively equalizes compute loads across workers in pipeline-parallel training. In addition, DynMo dynamically consolidates computation onto fewer workers without sacrificing training throughput, allowing idle workers to be released back to the job manager. DynMo supports both single-node multi-GPU systems and multi-node GPU clusters, and can be used in practical deployment. Compared to static distributed training solutions such as Megatron-LM and DeepSpeed, DynMo accelerates the end-to-end training of dynamic GPT models by up to 1.23x for MoEs, 3.18x for parameter pruning, 2.23x for layer freezing, 4.02x for sparse attention, 4.52x for early exit, and 1.17x for MoDs.

CCS Concepts

• **Computer systems organization → Parallel architectures**.

Keywords

Large Language Models, Load Balancing, Pipeline Parallelism

ACM Reference Format:
Mohamed Wahib, Muhammed Abdullah Soyturk, and Didem Unat. 2025. Balanced and Elastic End-to-end Training of Dynamic LLMs. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'25).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SC'25, St. louis, MO

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Neural network sizes used to train LLMs have grown exponentially since the introduction of Transformers [57], demanding increasingly more memory and compute power. However, the memory and compute capacity of a single accelerator have not kept pace [46]. As a result, combinations of model and data parallelism have been adopted to train large models [36]. Pipeline parallelism is one of the most widely used forms of model parallelism in LLMs, where consecutive layers are grouped into stages, each assigned to an accelerator (worker) [24]. Input mini-batches are split into micro-batches to increase utilization through pipelined execution [9, 17, 20, 29, 41]. Pipeline parallelism is essential in practice, as combining data, tensor/expert, and context parallelism alone is insufficient to scale training to tens of thousands of GPUs. Major production families of models such as OpenAI's GPT, Claude, Mixtral, DeepSeek, Qwen, Gemini etc reportedly rely on pipeline parallelism in combination with other forms of parallelism.

In traditional LLM training, each pipeline stage has a fixed workload known in advance. In contrast, emerging schemes designed to reduce computational demands introduce dynamic workloads. These include: neural networks where different input samples take different pathways through the model layers, e.g. gated neural networks [47], sparsely activated Mixture of Experts (MoEs) [61], Switch Transformers [10], Mixture of Depths (MoDs) [44] etc., gradual pruning where the parameters of a model are pruned (i.e. sparsified) during training [13], freeze training where some of the layers of the model are adapatively frozen during training [59], different schemes to dynamically sparsify the attention matrix [30, 38, 54], and early exit strategies where tokens skip remaining layers based on an exit decision [8, 25, 32, 45]. Beyond computational efficiency, dynamic models are also explored for benefits like improved generalization and explainability. We refer readers to [16, 55] for surveys on dynamic LLMs.

One of the main downsides of using dynamic models is the load imbalance they introduce in pipeline parallelism, which significantly reduces LLM training throughput [19, 60]. For example, Figure 1 shows the average GPU idleness for GPT models with varying depths and different dynamic model types. This imbalance appears as *bubbles* in the pipeline, where an accelerator stalls while waiting for work to be passed from its neighboring worker. As illustrated in the figure, idleness can range from 18% to 5x, depending on the scheme, e.g., layer freezing introduces up to 40% idleness for a 40-layer model. Note that these bubbles are distinct from the inherent gaps in pipeline scheduling, such as idle periods in the pipeline wind-up and wind-down. The former, which stems from dynamic workloads, is the primary focus of this paper.

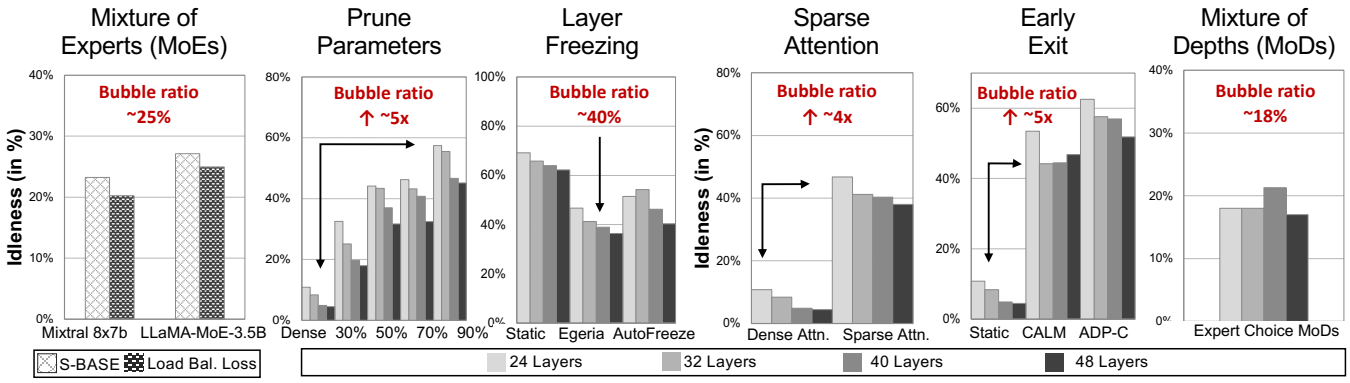

Figure 1: Average idleness percentage of GPUs (per iteration) for training dynamic GPT models [42] on 720 H100 GPUs in total, excluding MoEs which uses 128 H100 GPUs in total. We use models parameterized to have between 24 and 48 layers, except for the MoEs user case for which we report the average idleness percentage for Mixtral 8x7b and LlaMA-MoE-3.5B models. For pipeline parallelism, we use the highest performing (single) pipeline parallelism scheme known to the authors: the "almost zero-bubble pipeline parallelism" scheme [41]. The bubble ratios are measured on a hybrid of pipeline and data parallelism. ❶ MoE: we observe ~25% bubble ratio in the pipeline on Mixtral 8x7b [23] and LLaMA-MoE-3.5b [56], arising from the load imbalance imposed by the routing schemes used in token choice (S-BASE [27] and load imbalance with auxiliary loss [23]). ❷ Gradual pruning of model parameters: we observe almost a five fold increase in idleness at 90% sparsity levels. Note that idleness at *Dense* is the inherent pipeline bubbles of a static model. ❸ Layer freezing: SoTA freezing schemes that incorporate load balancing (Egeria [59] and AutoFreeze [31]) yield ~40% bubble ratio. ❹ Dynamic Sparse Flash Attention: locality sensitive hashing with support for flash attention [38] exhibits a 4x increase in the bubble ratio over the baseline dense attention. ❺ Early exit: SoTA early exit methods (CALM [45] and ADP-C [32]) exhibits up to 5x increase in the bubble ratio over the baseline (w/o early exit), mainly due to the accumulation of bubbles in late layers. ❻ MoD: we observe ~18% bubble ratio in the pipeline, arising from the load imbalance imposed by the routing scheme of expert choices that lacks information about future tokens [44].

Production distributed training frameworks typically apply static load balancing at the start of training and maintain the same distribution throughout. For example, Megatron-LM [50] evenly splits transformer layers across accelerators. DeepSpeed [52] offers three partitioning strategies: *uniform* (equal number of layers), *param* (equal number of parameters), and *regex* (grouping layers by name patterns). These methods assume that workloads remain stable during training. Consequently, they fail to handle the pipeline stalls introduced by dynamic models, leading to reduced computational efficiency.

Innovative designs of dynamic models aim to reduce computational cost, but without effective load balancing, their benefits practically fail to translate into actual performance gains during distributed training [4]. To address this gap, we introduce DynMo, an elastic load-balancing framework tailored for dynamic models. It is the first work to study pipeline stalls caused by training dynamic models. DynMo ensures balanced pipeline stages by dynamically redistributing workloads across accelerators whenever imbalance arises, thereby improving computational efficiency and reducing training costs. The framework incorporates two different dynamic balancers, both proven to converge to the optimal workload balance among workers. Our experiments demonstrate that DynMo incurs negligible overhead and scales effectively in both single-node multi-GPU and multi-node multi-GPU setups.

DynMo also provides the capability to elastically adapt GPU resources. As the total workload decreases during training due to techniques such as gradual pruning or early exit, the load balancer consolidates the work onto fewer GPUs, subject to memory capacity constraints, while maintaining performance. GPUs that are no longer needed can then be released back to the job scheduler. Given the high cost and long duration of LLM training, this elasticity enables significant additional cost savings. Our contributions are:

- We introduce DynMo, a framework for load balancing dynamic LLMs to improves their end-to-end training efficiency, making them practical for real-world use. We invoke DynMo to rebalance at regular intervals without prior knowledge of dynamism, hence balancing the load in a fully automated and transparent manner. DynMo is orthogonal to the underlying pipeline and dynamism schemes, ensuring compatibility with diverse dynamic compute and model reduction methods.
- We propose two load balancing algorithms proven to converge to optimal balancing. Additionally, we introduce a GPU re-packing scheme that reduces the number of GPUs used during training by consolidating work onto fewer devices.
- We demonstrate the benefits of DynMo across six dynamic model scenarios in both single-node and multi-node settings. On multi-node hybrid data and pipeline parallelism with up to 720 H100 GPUs, DynMo delivers speedups of 1.23× (MoEs), 3.18× (parameter pruning), 2.23× (layer freezing), 4.02× (sparse attention), 4.52× (early exit), and 1.17× (MoDs) over static Megatron-LM. Additionally, our re-packing strategy can reduce GPU usage by up to 50% while maintaining comparable performance.

Finally, we emphasize that DynMo has no impact on model accuracy, as it solely redistributes workload without altering the learning process or regime. DynMo functions as a complementary system software layer, operating independently of the underlying dynamic strategies such as parameter pruning, early exit, layer freezing, and expert routing. This design makes DynMo easily extendable and compatible with a wide range of dynamic schemes. In principle, it can also be applied to models that adapt for other reasons, such as hardware variability [51].

2 Dynamic Models

Training schemes reducing work can introduce dynamic training workloads. The irregular control-flow in those dynamics models lead inevitably to load imbalance. This leads to inefficiencies that cause the dynamic model to be slower than the baseline, hence defeating the purpose of using a dynamic model.

The load balancing problem considered can be formally defined as follows. Given a set of workers $\mathcal{N} = \{\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_n\}$ and a set of tasks $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$, each task $t_j \in \mathcal{T}$ is associated with a workload c_j. The total workload is denoted by:

$$C = \sum_{j=1}^{m} c_j$$

Let $A : \mathcal{T} \to \mathcal{N}$ be a load assignment function that maps each task to a worker. The load of a worker $\mathcal{N}_i \in \mathcal{N}$, denoted L_i, is defined as the sum of the workloads of tasks assigned to it:

$$L_i = \sum_{t_j \in A^{-1}(\mathcal{N}_i)} c_j$$

The objective of the load balancing problem is to minimize the maximum load among all workers:

$$\min_{A} \max_{i \in \{1,\ldots,n\}} L_i = \min_{A} \max_{i \in \{1,\ldots,n\}} \left(\sum_{t_j \in A^{-1}(\mathcal{N}_i)} c_j \right)$$

This optimization problem aims to distribute the tasks such that the workload is balanced across the workers, minimizing the worst-case scenario in terms of load.

In the remainder of this section, we build on the formal load balancing definition by extending the model to capture the dynamism introduced by each of our six example cases. In each of the six cases, we define the load on each worker N_j to calculate the load imbalance $\Delta L^{(k)}$ at time step k. Ideally, $L_j^{(k)} \approx L_{j'}^{(k)}$ for all workers, but the dynamic nature of the six cases leads to an imbalance. Defining maximum and minimum loads:

$$L_{\max}^{(k)} = \max_j L_j^{(k)}, \quad L_{\min}^{(k)} = \min_j L_j^{(k)} \tag{1}$$

the imbalance is:

$$\Delta L^{(k)} = \frac{L_{\max}^{(k)} - L_{\min}^{(k)}}{\frac{1}{n}\sum_{j=1}^{n} L_j^{(k)}} \tag{2}$$

2.1 Mixture of Experts

In MoEs [48], input tokens are routed to specialized sub-networks (experts) instead of a single feed-forward network, improving efficiency but introducing load imbalance when those experts are distributed among workers. Empirical results from Mixtral 8x7B [23] show up to 25% imbalance. Prior approaches attempt mitigation via auxiliary losses [23, 27], yet routing remains suboptimal. Recent strategies, such as DeepSeek V3 [6], introduce bias terms in token-to-expert affinity scores [58], achieving up to 8% imbalance per layer in a 3B model. However, imbalance compounds across layers and is expected to be higher in larger models. Load imbalance propagates through training, creating pipeline stalls, especially during `all_to_all` communication. Addressing this remains crucial for efficient MoE training in distributed systems.

Let $\mathcal{E} = \{e_1, e_2, \ldots, e_k\}$ be the set of experts, each assigned to a worker $\mathcal{N}_j \in \mathcal{N}$. A routing function $R : \mathcal{T} \to \mathcal{N}$ maps tokens to workers. The load of a worker L_j aggregating the workload c_{e_j} all tokens it is assigned, at time step k:

$$L_j^{(k)} = \sum_{e_j \in \mathcal{E}_i} \sum_{t_k \in R^{-1}(e_j)} c_{ej}^{(k)}$$

The load per worker defined above can be used with Equations 1 and 2 to get the overall system imbalance introduced by the stochastic, or learnable, routing.

2.2 Parameter Pruning

Global parameter pruning [15] removes a subset of model parameters, creating a sparse network that maintains performance but introduces imbalance due to non-uniform pruning across layers.

Let $\mathcal{L} = \{l_1, l_2, \ldots, l_d\}$ be the set of layers, each assigned to a worker $\mathcal{N}_j$. During distributed training, workload varies as parameters are pruned. The fraction of retained parameters in layer l_i at time step k is $p_i^{(k)}$, the total network compute being $C^{(k)}$, where $A^{(k)} : \mathcal{L} \to \mathcal{N}$ be the layer-to-worker assignment. The load on worker $\mathcal{N}_j$ is:

$$L_j^{(k)} = \sum_{l_i \in (A^{(k)})^{-1}(\mathcal{N}_j)} \sum_{i=1}^{d} p_i^{(k)} \cdot c_i$$

The load per worker defined above can be used with Equations 1 and 2 to get the overall system imbalance introduced by the pipeline stalls occurring while some workers wait for heavily pruned layers to complete [1, 12, 62].

2.3 Layer Freezing

Layer freezing reduces computational costs by halting updates to certain layers once they have converged. While improving efficiency, it introduces load imbalance when frozen layers are unevenly distributed among workers [49].

Let $\mathcal{L} = \{l_1, l_2, \ldots, l_d\}$ be the set of layers in an LLM, each assigned to a worker $\mathcal{N}_i$. Earlier layers often converge faster [59], enabling their freezing to save computation. Let $f_i^{(k)} \in \{0, 1\}$ indicate whether layer l_i is frozen at time step k:

$$f_i^{(k)} = \begin{cases} 1, & \text{if layer } l_i \text{ is frozen at } k, \\ 0, & \text{otherwise} \end{cases}$$

For c_i being the initial compute. If $f_i^{(k)} = 1$, then $c_i^{(k)} = 0$, contributing no computational load. For the total workload of layer l_i at time step k is, where $A^{(k)} : \mathcal{L} \rightarrow \mathcal{N}$ is the layer-to-worker assignment. The load on worker $\mathcal{N}_j$ is:

$$L_j^{(k)} = \sum_{l_i \in (A^{(k)})^{-1}(\mathcal{N}_j)} \sum_{i=1}^{d} (1 - f_i^{(k)}) \cdot c_i$$

The load per worker can be used with Equations 1 and 2 to get the overall system imbalance introduced by uneven freezing.

2.4 Dynamic Sparse Flash Attention

Dynamic sparse flash attention combines hash-based sparse attention with FlashAttention [5, 38], accelerating computations by restricting the attention matrix to blocks determined by hash codes. However, varying sparsification across layers causes load imbalances during distributed training.

Let $\mathcal{L} = \{l_1, l_2, \ldots, l_d\}$ be the set of layers, each using dynamic sparse attention. The workload depends on the sparsity level, which changes dynamically. Let $s_i^{(k)} \in [0, 1]$ be the sparsity factor of layer l_i at time k, representing the fraction of nonzero elements in the attention matrix. For c_i being the initial workload of layer l_i. Let $A^{(k)} : \mathcal{L} \rightarrow \mathcal{N}$ be the layer-to-worker assignment. The load on worker $\mathcal{N}_j$ is: at time k is:

$$L_j^{(k)} = \sum_{l_i \in (A^{(k)})^{-1}(\mathcal{N}_j)} \sum_{i=1}^{d} s_i^{(k)} \cdot c_i$$

The load per worker defined above can be used with Equations 1 and 2 to get the overall system imbalance introduced by sparsity, which varies across layers and time steps.

2.5 Early Exit

Early Exit allows tokens to skip layers once they reach a confident state, reducing computation but causing load imbalance due to uneven token distribution across layers. This can increase bubble ratios by up to 5x due to idle time [45].

Let $\mathcal{L} = \{l_1, l_2, \ldots, l_d\}$ be the set of layers where tokens can exit early. Let $t^{(k)}$ be the total tokens at time k, and $t_i^{(k)}$ as the tokens processed by layer l_i. Typically, $t_i^{(k)}$ decreases with depth [32, 45]. For c_i being the initial workload. For c_i being the initial workload, the total workload of all layers is:

$$C^{(k)} = \sum_{i=1}^{d} \frac{t_i^{(k)}}{t^{(k)}} \cdot c_i$$

Let $A^{(k)} : \mathcal{L} \rightarrow \mathcal{N}$ be the layer-to-worker assignment. The load of worker $\mathcal{N}_j$ is:

$$L_j^{(k)} = \sum_{l_i \in (A^{(k)})^{-1}(\mathcal{N}_j)} c_i^{(k)}$$

The load per worker defined above can be used with Equations 1 and 2 to get the overall system imbalance introduced by reduced work in later layers of the model.

2.6 Mixture of Depths

MoD [44] generalizes early exit by allowing tokens to skip both intermediate and final layers. The MoD variant used in this work employs expert choice via MoEs for improved performance. However, dynamic layer skipping and variability in expert selection introduce significant load imbalances in distributed training. Empirical results show up to 18% imbalance in MoDs [44].

Let $\mathcal{L} = \{l_1, l_2, \ldots, l_d\}$ be the set of layers, with $t^{(k)}$ as the total number of tokens at time k, and $t_i^{(k)}$ as the tokens processed by layer l_i. MoD enables tokens to skip layers based on expert predictions, leading to fluctuating workloads. The routing weight $r_i^{(k)}$ determines whether tokens bypass a layer, and the total effective compute of l_i at time k is:

$$C^{(k)} = r_i^{(k)} \cdot t_i^{(k)} \cdot c_i$$

Let $A^{(k)} : \mathcal{L} \rightarrow \mathcal{N}$ be the load assignment function, then each worker $\mathcal{N}_j$ would have the load:

$$L_j^{(k)} = \sum_{l_i \in (A^{(k)})^{-1}(\mathcal{N}_j)} c_i^{(k)}.$$

The load per worker defined above can be used with Equations 1 and 2 to get the overall system imbalance introduced by: a) the MLP predictor's misestimation of whether a token will be among the top-k selected for the next layer, and b) possible imbalance from the underlying MoE that the MoD sits on top of.

3 DynMo Framework

3.1 Overview

We evaluate our load balancing system on six representative dynamic model cases that static distributed training systems struggle to handle efficiently. While our work focus on these cases, the approach generalizes to other forms of dynamic models.

Figure 2 illustrates the overview of DynMo with all its steps. The implementation of individual steps of model (or control-flow) altering, load balancing, and re-packing can be found in their respective sections. The dynamism depends on the target case. For instance, if the target case is parameter pruning, the dynamism function would apply global pruning on different worker.

DynMo takes as input a model, the number of training iterations, and several dynamism configuring arguments (that have default values if the user wishes to use DynMo out-of-the-box). We start the training with the original model and train it until the dynamism to apply is reached. The frequency varies by the target case, in layer freezing it is as frequent as every 50 iterations, while in parameter tuning the pruning frequency is in the range of 3000-7000 iterations. The model or control flow is altered only if the training is in the dynamism stage. In this stage, the model is adjusted by either being modified or when the control flow inside the model changes, until it meets the training stopping criteria. The first iteration after each dynamism operation is used for profiling the time it takes to execute each layer in the altered model and the memory usage of all workers (accelerators) in the pipeline. Next, DynMo collects the profiling information and decides on balancing the workload by moving layers across pipeline stages based on the execution times of individual layers to minimize the pipeline stalls, subject to the

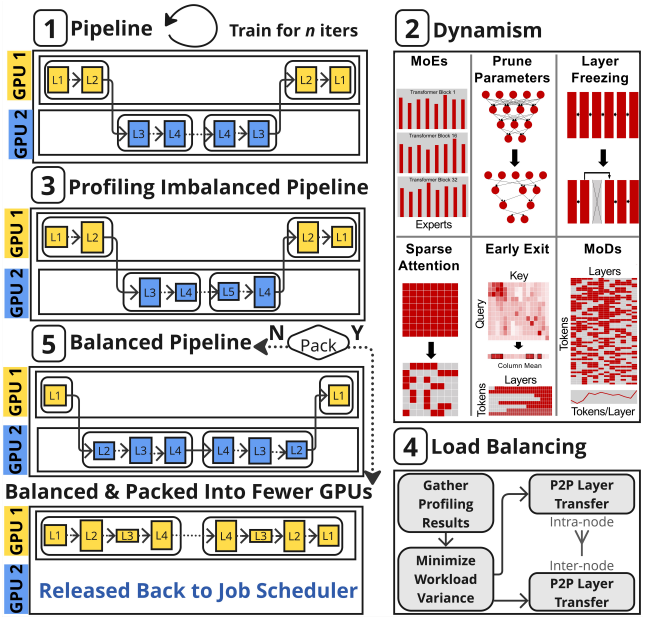

Figure 2: Overview of DynMo. The flow in the figure (top to bottom) is repeated until training is completed. Each yellow and blue rectangle represents a transformer layer. The size of a rectangle illustrates the amount of work in a layer. (1) shows the pipeline before the model starts to change due to dynamism. (2) some action (dynamically) changes the model or the flow of work inside the model changes. (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 time to rebalance, optionally it reduces the number of resources (GPUs) used in training by re-packing.

constraints of memory capacity per worker. DynMo also attempts to re-pack the total workload into fewer number of GPUs if the re-packing feature is enabled by the end user. Once the training is out of the dynamism region, the balanced pipeline continues to execute with the model.

By large, the mechanism for measuring the load imbalance, re-distributing the load, and re-packing (when possible) does not vary from case to case. However, we need to identify the load imbalance arising from dynamism. For each of the six example cases, a representative case is discussed next.

3.2 Imbalance Caused by Dynamism in Model Training

DynMo operates as a black-box approach where the load balancing happens at regular fixed intervals, without any knowledge of whether the model has changed or not. As will be shown in the results section, the very low overhead allows DynMo to be invoked even at the granularity of each iteration. Due to space constraints, we focus here on describing the imbalance due to dynamism for gradual pruning, as a representative of the different cases we cover in this paper.

Algorithm 1 Dynamism (Global Pruning as an Example)

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

3.2.1 ***Gradual Global Magnitude Pruning.*** For our pruning design, we use the gradual pruning schedule proposed in [62] which is formulated as:

$$S_t = S_f + (S_i - S_f)(1 - \frac{t - t_0}{n\Delta t})^3,\ t \in \{t_0, t_0 + \Delta t, ..., t + n\Delta t\} \quad (3)$$

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.

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 [39]. 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 [3].

To our knowledge, there is no deep learning framework that supports global pruning on a distributed model at the time of this writing (support is only for undistributed models). Thus we implemented our own global pruning algorithm as shown in Algorithm 1. 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 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).

3.3 Load Balancing

DynMo implements two load balancing algorithms, and can be extended to support other algorithms. The first is centralized parameter-based partitioning that balances partitions based on the number of parameters. The load balancing algorithm is built on top of DeepSpeed's load balancing utility functions for partitioning in model

[1]We use one MPI rank per GPU.

parallelism [52]. The second algorithm is an iterative decentralized diffusion-based algorithm that aims to minimize the variance between the workload of each rank by attempting to move layers from overloaded workers to underloaded ones in an iterative way. The workload can be described by either the layer execution times, or the parameter counts as in the centralized partitioning method.

We demonstrate that the two load balancing schemes meet the goals for optimal load balancing by using the following lemmas.

Lemma 1. *A centralized load balancer L_c over $\mathcal{N}$ workers satisfies maximum reduction in the imbalance $\mathcal{N}_i$ if and only if $\mathcal{N}_i$ reduces the bubble ratio to minimum.*

Proof. We will prove by contradiction. Suppose a centralized load balancer L_c over $\mathcal{N}$ workers satisfies maximum reduction in the imbalance $\mathcal{N}_i$ when $\mathcal{N}_i$ has a bubble ratio higher then the minimum. By the definition of maximum reduction in load balance, L_c must preserve minimum differential between the loads of workers $\mathcal{N}_i$ and $\mathcal{N}_j$, which $\mathcal{N}_i$ and $\mathcal{N}_j$ have the minimum load and maximum loads in $\mathcal{N}$, respectively. Consequently, increasing the bubble ratio of $\mathcal{N}_j$ changes the load difference between $\mathcal{N}_i$ and $\mathcal{N}_j$. This is in contradictory of L_c achieving maximum reduction on imbalance. □

Lemma 2. *An iterative decentralized diffusion based load balancer L_d over $\mathcal{N}$ workers satisfies maximum reduction in the imbalance $\mathcal{N}_i$ if and only if $\mathcal{N}_i$ reduces the bubble ratio to minimum. Also the load balancer is guaranteed to converge to the maximum reduction in imbalance in the following number of rounds*

$$O\left(\min\left\{\mathcal{N}^2\log\left(\frac{S\mathcal{N}}{\gamma}\right)\log\mathcal{N}, \frac{S\mathcal{N}\log\mathcal{N}}{\gamma}\right\}\right)$$

where $\gamma \in \mathbb{R}_{>0}$ is the convergence factor and $\in \mathbb{R}_{>0}$ is the total number of stages in the pipeline.

Proof. We leverages core ideas from Lyapunov optimization. We first define a potential function, ϕ, that measures at each round the total magnitude of workload gaps in the system:

$$\forall r \geq 0 : \phi(r) = \sum_{u,v \in V} |x_u(r) - x_v(r)|$$

Similar to a Lyapunov function, ϕ maps the system state (in this case, a vector of workloads for $\mathcal{N}$ workers) at any given round to a non-negative scalar value that describes the desirability of the current system state. As ϕ decreases toward 0, the system state becomes more desirable; i.e. the workload is balanced across $\mathcal{N}$. As in a standard Lyapunov optimization, we show below that the modifications to a system state caused by executing a single round of our max neighbor algorithm will drift the value of ϕ toward zero in a non-decreasing manner. We establish a probabilistic lower bound for the amount of drift in a given round to obtain our time bounds.

For a given round $r \geq 0$ and node pair $u, v \in V$, we define $d_{u,v}(r) = |x_u(r) - x_v(r)|$ to describe the gap between u and v's workload at the end of that round. For each such r, we also define: $\{\{u, v\} \,|u$ and v connected and averaged their workloads in round $r\}$, i.e., the set of node pairs that connect and average in r, and $D_r = \sum_{u,v \in A_r} d_{u,v}(r-1)$, i.e., the sum of gaps averaged in r. Finally, we define $t_{max}(r) = max_{u,v \in V}\left\{d_{u,v}(r)\right\}$ to describe the largest gap between any two nodes at the end of round r. From the above analysis that $\phi(r)$ decreases by at least D_r in each round r, we proceed to prove the converge time complexity bound.

For a maximum number of rounds to converge to the minimum imbalance:

$$O\left(\min\left\{\mathcal{N}^2\log\left(\frac{S\mathcal{N}}{\gamma}\right)\log\mathcal{N}, \frac{SN\log\mathcal{N}}{\gamma}\right\}\right)$$

Note that these two bounds essentially coincide at $\tilde{O}(\mathcal{N}^2)$ with $\gamma = \Theta(S/n)$, where the notation $\tilde{O}$ hides logarithmic factors. In other words, if we want all nodes to have the same workload up to a constant factor, the max neighbor strategy uses $\tilde{O}(\mathcal{N}^2)$ rounds. We first note that if we arrive at a round r in which $\phi(r) \leq \gamma$, then the system ends this round γ-converged, i.e. the sum of the gaps is at most γ, and thus clearly any individual gap is at most γ. Since ϕ is monotonically non-increasing, it follows that every round $r' \geq r$ is also γ-converged. So we just need to show that with high probability, ϕ will decrease to γ in the time bound stated by the theorem statement.

For each $r \geq 1$, we call r "good" if and only if $\phi(r-1) - \phi(r) \geq s_{max}(r-1)/(60\ ln(2n))$. We next calculate how many good rounds guarantee that ϕ falls below γ. To do so, we first note that, non-good rounds cannot increase ϕ, so we are safe to focus only on reductions generated by good rounds in calculating our bound.

By the definition of ϕ, for each $r \geq 1$ we know that $\phi(r) < s_{max}(r)n^2$. It follows that if r is a good round, then it decreases $\phi(r-1)$ by a multiplicative factor less than $(1 - \frac{1}{60n^2 ln(2n)})$. Finally, we also observe that $s_{max}(0) \leq S$ and therefore $\phi(0) < Sn^2$. Leveraging these observations, to find the number of good rounds needed to decrease ϕ below γ, we just need to find the minimum s time steps such that

$$Sn^2\left(1 - \frac{1}{60n^2 ln(2n)}\right) \leq \gamma$$

A simple calculation implies that $s_{con} = 60n^2 ln(2n) ln(Sn^2\gamma^{-1})$ is sufficient to satisfy this inequality. We have now established that after s_{con} good rounds the system will be γ-converged for all future rounds. We are left to bound the number of rounds required to generate s_{con} good rounds with high probability.

For each round r, let X_r be the random indicator variable that evaluates to 1 if round r is good and otherwise evaluates to 0. We know a given round r is good with probability at least $1/\mathcal{N}$, regardless of the history of the execution through the round $r-1$. We cannot, however, directly leverage this observation to calculate (and concentrate) the expected sum of X variables for a given execution length, as the distribution determining a given X_r might depend in part on the outcome of previous experiments. To overcome this issue, we define for each round r, a trivial random indicator variable $\hat{X}_r$ that evaluates to 1 with independent probability $1/N$ and otherwise evaluates to 0. Note that for each r, X_r stochastically dominates $\hat{X}_r$, and therefore for each $s > 0$, $Y_s = \sum_{r=1}^{s} X_r$ stochastically dominates $s > 0$, $\hat{Y}_s = \sum_{r=1}^{s} \hat{X}_r$. It follows for any $s > 0$, if $\hat{Y}_s \geq s_{con}$ with some probability p then $Y_s \geq s_{con}$ with probability at least p.

A Chernoff bound applied to $\hat{Y}_s$, for $s = c.s_{con}$ (where $c \geq 1$ is a sufficiently large constant defined with respect to the constants in s_{con} and the constants in the Chernoff form used), provides that $\hat{Y}_s \geq s_{con}$ with high probability, and therefore so is Y_s. To conclude

the proof, we note that $c.s_{con} \in O\left(\mathcal{N}^2 log(\frac{S\mathcal{N}}{\gamma}) log\mathcal{N}\right)$, as required by the theorem γ statement. □

The diffusion-based load balancing algorithm achieves ideal load balancing by iteratively minimizing workload imbalances using a Lyapunov-inspired approach. The potential function ϕ, defined as the sum of workload gaps between workers, serves as a measure of imbalance in the system. Each iteration of the algorithm reduces ϕ by redistributing tasks from overloaded to underloaded workers, prioritizing layer transfers that yield the largest reductions in imbalance while satisfying memory constraints. The algorithm's probabilistic analysis guarantees that ϕ decreases towards a convergence threshold γ, with the rate of convergence bounded by $O(\mathcal{N}^2 \log(S\mathcal{N}/\gamma) \log \mathcal{N})$, where $\mathcal{N}$ is the number of workers and S is the total pipeline size. This systematic reduction ensures that workload imbalances, quantified by the bubble ratio, are minimized, driving the system towards an optimally balanced state. The theoretical guarantees of convergence and imbalance reduction underpin the algorithm's robustness in dynamic environments.

3.3.1 ***Overhead of DynMo and Frequency of Dynamism.*** DynMo incurs negligible overhead (detailed in the evaluation section). Across all model sizes and dynamism scenarios, total overhead stays within a few single-digit percentages. This includes profiling, the rebalancing algorithm, and inter-GPU layer migration.

As a general rule, we apply rebalancing via DynMo whenever the model or control flow changes. Given its low overhead, DynMo can be used as frequently as needed, depending on application requirements. For example, in gradual pruning, dynamism (that is, model changes requiring rebalancing) typically occurs every few thousand iterations. In contrast, for MoEs and MoDs, rebalancing is needed every iteration due to unpredictable imbalances caused by routing decisions taken during the forward pass at each FFN. In these cases, we perform rebalancing during backpropagation, coupling layer migration with the pipeline parallelism scheme by moving layers while the gradients calculation take place, from the last to the first layer.

3.4 Re-packing the Model to Fewer Workers

Workload re-packing is the process of consolidating the total training workload onto a reduced number of workers (GPUs) when the overall compute demand drops, allowing idle GPUs to be released. For long training schedules common in LLM training, this elasticity can yield significant cost savings. It can also lead to improved performance by decreasing the number of cross-worker communications and mitigating pipeline bubbles.

3.4.1 ***Re-packing Algorithm.*** As shown in Algorithm 2, we implement workload consolidation using a first-fit algorithm with the objective of reducing the number of active workers, subject to memory capacity constraints. The algorithm iteratively examines each pair of workers (lines 2-3): if their combined memory usage fits within a single GPU's memory budget (lines 4-5), the workload is migrated to consolidate onto one GPU, freeing the other (lines 7-8). This process continues across the set of workers until no further consolidation is possible. Re-packing is scheduled at the end of each training iteration, leveraging the existing synchronization barrier to avoid the need for introducing additional barriers.

Algorithm 2 Re-pack Layers into Fewer Workers

```
   Input: active_workers, mem_usage
   Input: target_num_workers, num_layers
   Output: transfers (list)
 1: transfers ← []
 2: for src in range(num_ranks) do
 3:     for dst in range(src + 1, num_ranks) do
 4:         if mem_usage[src] + mem_usage[dst] < MAX_MEM
 5:        && sum(active_workers) > target_num_workers then
 6:             active_workers[src] = 0
 7:             for lyr_idx in range(num_layers[src]) do
 8:                 transfers.append((src, dst, lyr_idx))
 9:             end for
10:             mem_usage[dst] += mem_usage[src]
11:             num_layers[dst] += num_layers[src]
12:         end if
13:     end for
14: end for
15: return transfers
```

3.4.2 ***Releasing Unused GPUs after Re-packing.*** To ensure the GPUs are released in a practical manner, we use the NCCL communicator. While the NCCL communicator can not be resized, NCCL supports splitting the communicator via `ncclCommSplit()` into multiple sub-partitions, or even creating a single communicator with fewer ranks [37]. After re-packing, active GPUs are assigned to a new sub-communicator, while idle GPUs are assigned to a different one. Since idle GPUs are excluded from the active communicator, concurrent communicators can safely proceed without the risk of deadlock.

Alternatively, re-packing can be coordinated with checkpointing. Unlike rebalancing, which may occur every few iterations, re-packing is infrequent—typically triggered after thousands of iterations, when the model structure has changed significantly. By combining re-packing with a checkpoint restart, the implementation is simplified since a new NCCL communicator is already created during the restart. Moreover, because the model is reloaded and reshared among the workers during checkpoint recovery, there is no additional overhead for resharding the model to a new set of workers.

DynMo's supports the release of the GPUs to the ECK (Elastic Cloud on Kubernetes) [7]. After re-packing, DynMo, signals the release of GPUs release back to the ECK-managed Kubernetes cluster. In particular DynMo uses the Kubernetes API to send a `PATCH` request to the Kubernetes API server to update the `resources.requests` and `resources.limits` fields in the pod spec, effectively reducing or removing GPU resource claims. ECK, which automates the orchestration of containerized workloads and their underlying resources, then detects these freed GPUs and reassigns them to other pending jobs in the queue. This integration allows for more efficient and elastic resource allocation across workloads, minimizing idle GPU time and improving overall utilization.

Lastly, even though DynMo's speedup mostly comes from dynamic load balancing, re-packing provides 4–11% additional performance benefit from the reduced number of number of ranks and volume of communication.

4 Implementation

The DynMo framework was developed on top of NVIDIA Megatron Core 0.9.0[2]. Each component of DynMo, namely hooking to dynamism point in model training, load balancing, and re-packing is implemented in a separate package for ease of use and extension.

The `gather` and `scatter` operations in global pruning were implemented by employing NCCL Peer-to-Peer (P2P) `send-receive` operations instead of collectives since the sizes of the objects to be sent (`local_topk`) and received (`indices_to_keep`) in Algorithm 1 from each rank are different and other ranks do not have this size information to participate in the collective call. It is worth mentioning that while an `alltoallv` collective could be used as an alternative, using it however would add unnecessary global synchronization and may cause contention. Thus we opt for P2P communication.

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 Megatron-LM. The memory consumption of each pipeline stage is gathered with PyTorch's memory statistics for CUDA.

4.1 Dynamic Reconfiguration of the Pipeline

DeepSpeed's PipelineModule [33] currently offers three partitioning strategies for distributing model layers, which can be set using the `partition_method()` keyword argument passed to the `PipelineModule`. This allows us to move the layers between GPUs, when needed. When a layer is migrated from GPU *A* to GPU *B*, the memory allocated for the layer (weights, activation, gradients, optimizers) is released on GPU *A* and allocated on GPU *B*. As the training resumes (after the layers are migrated), the underlying pipeline scheme starts to pass the new mini-batches along the new distribution of layers in the pipeline.

4.2 Implementations of Dynamic Models

4.2.1 ***Mixture of Experts.*** We build on Mixtral 8x7b weights from Hugging Face [35] and LLaMA-MoE-3.5B [56] in continual training by monitoring the imbalance between layers with respect to number of assigned tokens. We used both the auxiliary load balance scheme adopted in Mixtral 8x7b [23] and the S-BASE load balancer [28]. Since the imbalance is dependent on the point the routing decision is taken, i.e., in the forward pass at each FFN, we rebalance in the back propagation phase where we attach the movement of layers to the pipeline parallelism communication flow.

4.2.2 ***Parameter Pruning.*** Among the six example cases, gradual pruning posed a unique challenge, as it required code changes to integrate with DynMo. Unstructured pruning demands a sparse storage format for efficient training and transfer. A common choice is the compressed sparse row (CSR) format, which necessitates replacing dense matrix multiplications (DMM) with sparse equivalents (SpMM). Since PyTorch does not support computing the derivative of SpMM operations for backpropagation on a CSR tensor, we evaluated GPU-compatible CSR-based SpMM implementations, namely cuSPARSE by Nvidia and Sputnik [14]. Sputnik's SpMM consistently outperformed cuSPARSE across all tested sparsity levels. This is largely due to Sputnik's kernels being tailored for deep learning workloads, unlike cuSPARSE, which is optimized for HPC workloads with extreme sparsity (often >99%). Notably, Sputnik begins to outperform cuBLAS around 75% sparsity. Thus, for sparse operations, we implemented PyTorch bindings for the CUDA kernels of Sputnik.

4.2.3 ***Layer Freezing.*** DynMo sits on top of layer freezing solutions. More specifically, we build on Wang et al. [59] Egeria solution by monitoring the rate by which the training loss changes, freezing layers when they reach the convergence criterion, and drop frozen layers from in both the back propagation phase and gradient exchange when data parallelism is used. It is important to note that Egeria periodically updates the reference model (on the CPU) to drive the layer freezing, yet does not actively try to remedy the load imbalance caused by layer freezing. The effect of load imbalance is particularly pronounced since earlier layers tend to be more frozen than later layers, i.e. the layer freezing is not uniformly occurring across the model. In comparison, DynMo dynamically balances layer placement across GPUs, and in an orthogonal fashion, whenever the reference model used for freezing is updated.

4.2.4 ***Dynamic Sparse Flash Attention.*** We build on Pagliardini et al. dynamic sparse attention [38] in continual training by implementing a binding from PyTorch to the CUDA kernel provided by authors of the paper. The hash-based attention makes the causal attention mask irregular, i.e., we get blocked sparsity that is then leveraged by the Flash Attention. The irregular causal structures caused by the hashing lead to different amount of blocks/tiles in different layers.

4.2.5 ***Early Exit.*** We adapt the early exit methods CALM [45] and ADPC [32] to observe the imbalance by peaking into the confidence measure prediction of CALM and ADPC. Since early exit happens at the later layers, we start our observation from the first layer at which tokens start to exit, and we assume all layers before than to have the same load. Early exit in particular benefits greatly from re-packing. That is since the change in control flow of the model happens in the later layers.

4.2.6 ***Mixture of Depths.*** We build on the MoDs work by Raposo et al. [44] by including in our GPT models we use in testing the small auxiliary MLP predictor that predicts whether that token will be among the top-k or not in the sequence. Similar to the case of MoEs, since the imbalance is dependent on the point the router takes the decision, i.e. in the forward pass, we rebalance in the back propagation phase where we attach the movement of layers to the pipeline parallelism scheme. Since the routing happens around the entire block, i.e., the routing applies to both forward MLPs and multi-head attention, we treat the skipped layers to be *shadow* layers when redistributing the layers on workers.

5 Evaluation

Experiments were primarily conducted on a supercomputer where each compute node consists of 2x AMD EPYC 9654 96-core/2.4GHz CPUs and 4x NVIDIA H100 SXM5 (80GB) GPUs. GPUs communicate with CPUs over PCIe Gen5 x16, and with each other via

[2]https://github.com/NVIDIA/Megatron-LM/releases/tag/core_r0.9.0

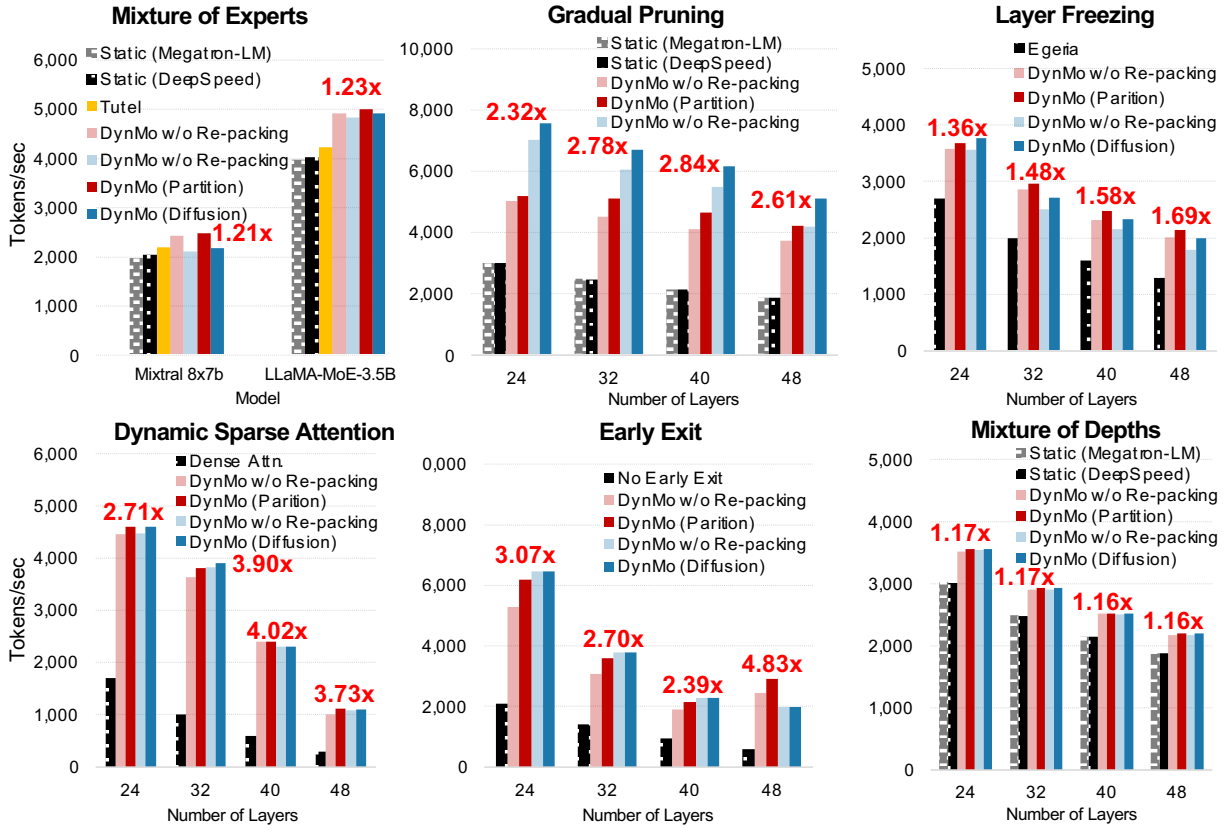

Figure 3: Throughput of end-to-end training for six different example cases. DynMo rebalances at regular intervals w/o prior knowledge of dynamism. MoE, Sparse Attn., and MoDs: invoke DynMo every iteration. Pruning, layer freezing, and early exit: invoke DynMo every 100s to 1,000s iterations. Speedup we report is the highest among balancing by number of parameters or layer execution time, divided by the highest among static Megatron-LM and DeepSpeed (or SoTA baseline, when available). MoEs and MoDs: we use 128 GPUs (16 nodes each with 4x H100s) in a hybrid of 8-way data parallel + 16-way pipeline). For gradual pruning, layer freezing, dynamic sparse attention, and early exit we use a total of 720 H100 GPUs (90 nodes each with 4x H100s) in a hybrid of 30-way data parallel + 24-way pipeline.

NVSwitch (NVLink34 x6). Nodes are interconnected via 4x 200Gbps InfiniBand NDR200 links. We used CUDA 12.6, OpenMPI 4.0.7, and PyTorch 2.3.1 with the NCCL 2.17.1 backend.

Models were trained on the Wikipedia dataset [11]. For Mixtral 8x7B and LLaMA-MoE-3.5B, we perform continual training. All models use a sequence length of 2048, hidden size of 1024, and 32 attention heads. Unless otherwise specified, training runs for 10,000 iterations with micro-batch size 2 and batch size 64. In multi-node experiments, as we scale the number of GPUs, we proportionally increase the batch size to maintain four micro-batches per GPU, following the guidance in [20] for optimal pipeline utilization.

We evaluated two dynamic load balancing algorithms, each with two configurations, across all six dynamic model scenarios. The first, Partition, uses DeepSpeed's [43] API and applies either parameter counts *(Partition: by Param)* or decoder layer execution times *(Partition: by Time)* to determine optimal layer placement using binary search and linear probing. The second, Diffusion, is a decentralized, iterative algorithm that balances load by minimizing variance across workers, with variants using either parameter counts *(Diffusion: by Param)* or execution times *(Diffusion: by Time)*.

5.1 End-to-end Training Throughput and Speedup

All our throughput and speedup results include the load balancing overhead, unless specified otherwise. We trained GPT models [42] having different numbers of layers to determine the training throughput and speedup over static Megatron-LM and DeepSpeed (or SoTA baseline, when available).

Figure 3 for MoEs, gradual pruning, and MoDs presents the highest throughput of two static and four dynamic load balancers. The first static balancer, Megatron-LM [50], evenly distributes layers across accelerators. The second static balancer, DeepSpeed [34], balances the number of parameters before training begins. In contrast, each of the two the dynamic load balancers (*Partition* and *Diffusion)* has two variants to redistribute the layers after each dynamism step: redistribute based on number of parameters or based on the layer execution time. Parameter-based balancers require profiling after the pruning step for memory usage information, while time-based balancers require profiling for memory usage and layer execution time information. We report the highest among both. Figure 3 for layer freezing, dynamic sparse attention, and early exit compare

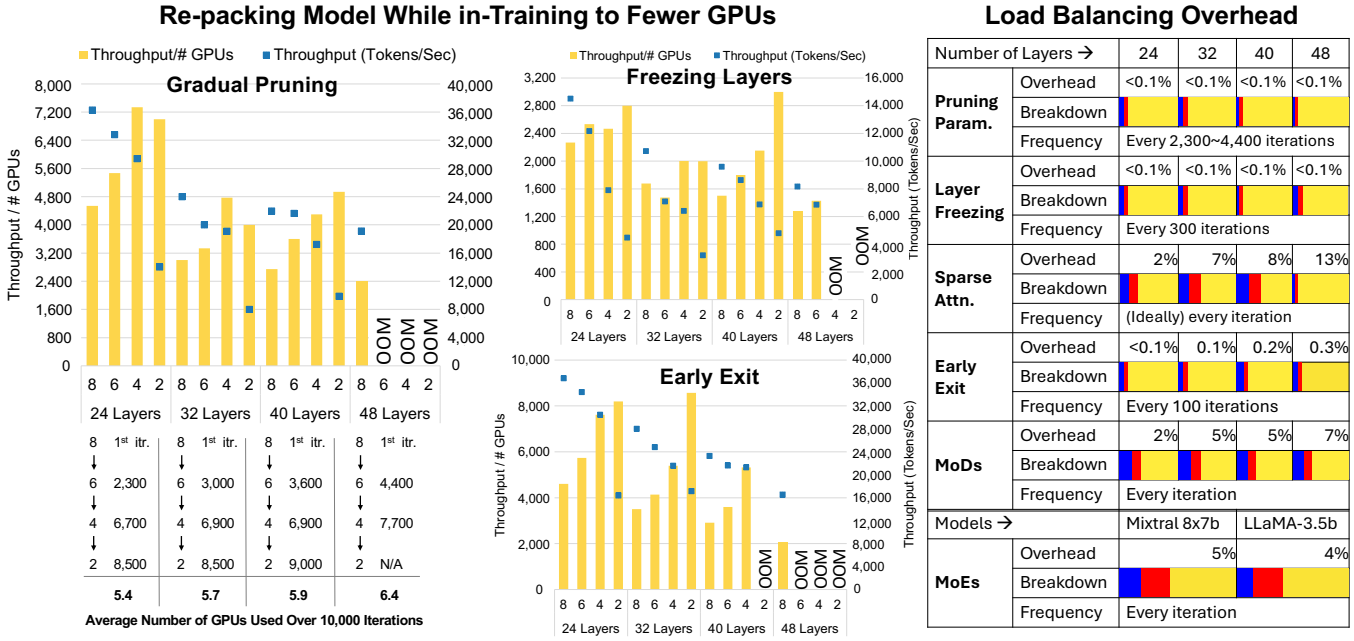

Figure 4: Re-packing the layers of GPT models into fewer GPUs as the model gets smaller due to dynamism. X-axis: Number of GPUs in pipeline parallelism using 90 nodes and up to 8 GPUs per node. Left Y-axis: throughput/number of GPUs shown with yellow bars. Right Y-axis: throughput (tokens/sec) shown with blue squares. Below: we show the average number of GPUs needed throughout the gradual pruning training at which we dynamically re-pack (total 10,000 iterations). Right: load balancing overhead for example cases. The overhead reported is broken down to three things: profiling (in), DynMo load balancing algorithm (in), and migration of layers between GPUs (in).

the two dynamic load balancers (*Partition* and *Diffusion)* over SoTA baseline that exists for those example cases.

We observed that the use of layer execution time for dynamic load balancing, such as diffusion or partitioning, consistently outperforms parameter count-based implementations across all scales. In every scale, execution time-based dynamic balancers surpass the baseline static balancers. As seen in the figure, most of the speedup reported is attributed to balancing the load by redistributing the, and not due to the reduced communication when we re-pack: the speedup gain from re-packing is between 4~11% of the entire speedup gain. In other words, even if we do not re-pack at, the speedup gain remains almost the same. Hence we treat re-packing as just a way to be efficient by using less GPUs, and not as a way to speed up imbalanced dynamic training.

Mixture of Experts. MoEs requires find-grained dynamism since the load vary from iteration to iteration. DynMo shows more than 1.21x improvement on Mixtral 8x7b and LLaMA-MoE-3.5B in continual training. We do not change any of the hyperparameters from the original implementations. In addition to the Megatron-LM and DeepSpeed baselines, we also compare a highly MoE-tailored system: Tutel [21]. DynMo significantly outperforms Tutel: 1.18x on Mixtral 8x7b and 1.21x on LLaMA-MoE-3.5B.

The improvement margins on MoEs and MoDs are in fact among the top in all six use cases, relative to how much for the bubble ratio was eliminated. DynMo reduces the bubble ratios of MoEs and MoDs from 25% to 8% and 18% to 4%, respectively. As a result, the end-to-end training of two production models improve 1.21x on Mixtral 8x7b and 1.23x on LLaMA-MoE-3.5B. Those improvements would translate to significant cost savings, considering the huge cost of training those models (and other similar models).

Gradual Pruning. 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 sparsity levels of 52%, 79%, and 90% after each pruning step. All other hyperparameters are the same as Megatron-LM. Using layer execution time for diffusion or partitioning dynamic load balancing outperforms the parameter count-based implementations in each scale, for up to 3.18x.

Layer Freezing. DynMo ourperforms the SoTA layer freezing tool Egeria [59]. We can observe two main points. First, the speedups of different load balancing algorithms over static algorithms are largely similar. This is mainly because the different algorithms tend to arrive at similar load balancing solutions when entire layers are frozen. Second, DynMo shows increased speedup as the number of layers increases, particularly with diffusion, primarily because Egeria's overhead grows fast with the number of layers, while DynMo overhead remains almost flat.

Dynamic Sparse Attention. DynMo achieves between 2.71x-4.02x speedup over the baseline dense attention (w/o sparsification). Dynamic sparse attention is the example case where DynMo is most efficient at removing the pipeline bubbles. That is since using layer

time execution, which fluctuates a lot in dynamic sparse attention, enables effective redistribution of the layers.

Early Exit. DynMo achieves more than 4x on average over the baseline w/o exit, i.e. when all tokens pass through the entire model. Similar to dynamic sparse attention, early exit benefits the most from DynMo due to the big variance in load between earlier and later layers.

Mixture of Depths. Like MoEs, MoDs layer loads vary from iteration to iteration. DynMo shows 1.17x improvement on the baseline in continual training. We suspect MoDs will in the future be able to benefit more from custom load balancers that leverage the knowledge of how MoEs is used in hybrid with MoDs.

5.2 Overhead of Load Balancing

Figure 4 (right) reports the load balancing overheads. This includes both the load balancing decision and the actual transfer of the parameters and other data of the layers to be sent or received, e.g., row offsets and column indices in CSR format for gradual pruning, and gradients in the case of MoEs and MoDs. The overhead is generally negligible, hence giving the opportunity for the use of DynMo in other forms of dynamic models, and not just the six example cases in this paper.

5.3 Re-packing Models to Fewer GPUs

In the re-packing experiments, the training starts with 8 GPUs per node in pipeline parallelism. After a dynamism step, DynMo attempts to re-pack the total workload into fewer GPUs while satisfying the memory capacity constraints. Figure 4 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 re-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.

We observe that in all model scales, re-packing can allow the training to be continued with fewer GPUs which may result in significant cost savings. For example, in gradual pruning we reduce the GPU count from 8 to an average of 5.8 GPUs while sustaining the training throughout.

6 Related Work

6.1 Load Balancing Model-Parallel DNNs

Layer-wise load balancing. Layer-wise balancing techniques operate at the granularity of layers rather than individual operators. DeepSpeed [34] provides three partitioning methods to balance stage workloads. The parameters method (i) aims to equalize the number of parameters across stages, while the uniform method (ii) distributes layers evenly. The regex method (iii) selectively distributes layers that match a given pattern, such as transformer layers. Similarly, He et al.[18] balance stages based on parameter counts, and Narayanan et al.[36] assign an equal number of transformer layers to each stage. However, none of these approaches use actual layer execution time to guide partitioning. DynMo leverages DeepSpeed's partitioning scheme by supporting both parameter counts and layer execution times for load balancing, and also includes a diffusion-based algorithm as a built-in option.

Load balancing via graph partitioning. Graph partitioning-based load balancing schemes identify atomic operations in the model and represent them as nodes in a directed acyclic graph (DAG), where edges capture data dependencies. Tanaka et al.[53] partition the DAG in three phases: identifying atomic operations, grouping them into blocks based on computation time, and combining these blocks into final partitions using DP. Qararyah et al.[40] form disjoint clusters by identifying critical paths and mapping them to devices using an algorithm that accounts for both communication minimization and temporal load balancing. Both approaches rely on profiling before training and partition only once.

Load balancing in MoE Models. MoE[22] 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. [26] 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. [10] route each token to only one expert and use the same expert capacity for restrictions. Lewis et al. [28] employ an auction algorithm [2] to solve the token-to-expert assignment problem. This line of work is different from ours as their aim is to balance workload in the feed-forward network while our work aims to balance all layers of the transformer model.

6.2 Re-Packing

PipeTransformer [18] 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 DynMo can pack to an arbitrary number of GPUs specified by the user. PipeTransformer uses the parameter size as a proxy to estimate the memory usage while DynMo uses the actual memory usage from the profiling step before load balancing. Finally, PipeTransformer is only capable of re-packing, and not load balancing. DynMo can do both.

7 Conclusion

Research on dynamic models will not deliver practical impact unless there is a platform from which those models can be made efficient. DynMo serves as that platform, offering a load-balancing system and a re-packing solution specifically designed for dynamic models whose worker loads vary during training. This leads to improved efficiency and faster end-to-end training times. Empirical results on LLMs, demonstrated across six representative dynamic model cases, show that DynMo significantly improves training throughput compared to existing approaches. Moreover, it enables elastic GPU utilization, resulting in additional cost savings. As dynamic models become increasingly prevalent, dynamic load distribution will be critical for scalable and efficient training.

Acknowledgments

Authors from Koç University has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 949587).

References

[1] Guillaume Bellec, David Kappel, Wolfgang Maass, and Robert Legenstein. 2017. Deep rewiring: Training very sparse deep networks. *arXiv preprint arXiv:1711.05136* (2017).

[2] Dimitri P. Bertsekas. 1992. Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications* 1, 1 (01 Oct 1992), 7–66. doi:10.1007/BF00247653

[3] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What is the state of neural network pruning? *Proceedings of machine learning and systems* 2 (2020), 129–146.

[4] Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. 2024. A Survey on Deep Neural Network Pruning: Taxonomy, Comparison, Analysis, and Recommendations. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46, 12 (2024), 10558–10578. doi:10.1109/TPAMI.2024.3447085

[5] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2024. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1189, 16 pages.

[6] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[7] Elastic. 2025. Elastic Cloud on Kubernetes (ECK). https://github.com/elastic/cloud-on-k8s [Retrieved 22 February 2025].

[8] Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. 2020. Depth-Adaptive Transformer. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SJg7KhVKPH

[9] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, 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*. 431–445.

[10] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *J. Mach. Learn. Res.* 23 (2022), 120:1–120:39.

[11] Wikimedia Foundation. 2023. Wikimedia Downloads. https://dumps.wikimedia.org

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

[13] Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574* (2019).

[14] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[15] M. Hagiwara. 1993. Removal of hidden units and weights for back propagation networks. In *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, Vol. 1. 351–354 vol.1. doi:10.1109/IJCNN.1993.713929

[16] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. 2021. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[17] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377* (2018).

[18] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. 2021. Pipetransformer: Automated elastic pipelining for distributed training of transformers. *arXiv preprint arXiv:2102.03161* (2021).

[19] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. 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* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 120–134. doi:10.1145/3503221.3508418

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[21] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, Joe Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. 2022. Tutel: Adaptive Mixture-of-Experts at Scale. *CoRR* abs/2206.03382 (June 2022). https://arxiv.org/pdf/2206.03382.pdf

[22] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. 1991. Adaptive mixtures of local experts. *Neural computation* 3, 1 (1991), 79–87.

[23] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG]

[24] Albert Njoroge Kahira, Truong Thao Nguyen, Leonardo Bautista-Gomez, Ryousei Takano, Rosa M. Badia, and Mohamed Wahib. 2021. An Oracle for Guiding Large-Scale Model/Hybrid Parallel Training of Convolutional Neural Networks. In *HPDC*. ACM, 161–173.

[25] Sehoon Kim, Sheng Shen, David Thorsley, Amir Gholami, Woosuk Kwon, Joseph Hassoun, and Kurt Keutzer. 2022. Learned Token Pruning for Transformers. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22)*. ACM. doi:10.1145/3534678.3539260

[26] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).

[27] Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. 2021. BASE Layers: Simplifying Training of Large, Sparse Models. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 6265–6274. https://proceedings.mlr.press/v139/lewis21a.html

[28] Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. 2021. Base layers: Simplifying training of large, sparse models. In *International Conference on Machine Learning*. PMLR, 6265–6274.

[29] Shigang Li and Torsten Hoefler. 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*. 1–14.

[30] Liu Liu, Zheng Qu, Zhaodong Chen, Fengbin Tu, Yufei Ding, and Yuan Xie. 2022. Dynamic Sparse Attention for Scalable Transformer Acceleration. *IEEE Trans. Comput.* 71, 12 (2022), 3165–3178. doi:10.1109/TC.2022.3208206

[31] Yuhan Liu, Saurabh Agarwal, and Shivaram Venkataraman. 2021. AutoFreeze: Automatically Freezing Model Blocks to Accelerate Fine-tuning. arXiv:2102.01386 [cs.LG]

[32] Zhuang Liu, Zhiqiu Xu, Hung-Ju Wang, Trevor Darrell, and Evan Shelhamer. 2022. Anytime Dense Prediction with Confidence Adaptivity. In *ICLR*. OpenReview.net.

[33] Microsoft. 2024. *DeepSpeed User Guide: Pipeline Parallelism*. https://deepspeed.readthedocs.io/en/latest/pipeline.html#deepspeed.pipe.PipelineModule

[34] Microsoft. 2023. Microsoft/DeepSpeed: A deep learning optimization library that makes distributed training and inference easy, efficient, and effective. https://github.com/microsoft/deepspeed

[35] Mistral. 2024. Hugging Face: Mixtral of Experts. https://huggingface.co/papers/2401.04088

[36] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[37] Nvidia. 2024. *Nvidia NCCL User Guide*. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/communicators.html

[38] Matteo Pagliardini, Daniele Paliotta, Martin Jaggi, and François Fleuret. 2023. Fast Attention Over Long Sequences With Dynamic Sparse Flash Attention. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=UINHuKeWUa

[39] Sai Prasanna, Anna Rogers, and Anna Rumshisky. 2020. When bert plays the lottery, all tickets are winning. *arXiv preprint arXiv:2005.00561* (2020).

[40] Fareed Qararyah, Mohamed Wahib, Doğa Dikbayır, Mehmet Esat Belviranli, and Didem Unat. 2021. A computational-graph partitioning method for training memory-constrained DNNs. *Parallel computing* 104 (2021), 102792.

[41] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. 2024. Zero Bubble (Almost) Pipeline Parallelism. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=tuzTN0eIO5

[42] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2018. Language Models are Unsupervised Multitask Learners. (2018). https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

[43] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[44] David Raposo, Sam Ritter, Blake Richards, Timothy P. Lillicrap, Peter Humphreys, and Adam Santoro. 2024. Mixture-of-Depths: Dynamically allocating compute in transformer-based language models. https://api.semanticscholar.org/CorpusID:268876220

[45] Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Q. Tran, Yi Tay, and Donald Metzler. 2022. Confident Adaptive Language Modeling. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=uLYc4L3C81A

[46] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. 2022. Compute trends across three eras of machine learning. *arXiv preprint arXiv:2202.05924* (2022).

[47] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[48] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer.. In *ICLR (Poster)*. OpenReview.net. http://dblp.uni-trier.de/db/conf/iclr/iclr2017.html#ShazeerMMDLHD17

[49] Sheng Shen, Alexei Baevski, Ari S Morcos, Kurt Keutzer, Michael Auli, and Douwe Kiela. 2020. Reservoir transformers. *arXiv preprint arXiv:2012.15045* (2020).

[50] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[51] Prasoon Sinha, Akhil Guliani, Rutwik Jain, Brandon Tran, Matthew D Sinclair, and Shivaram Venkataraman. 2022. Not All GPUs Are Created Equal: Characterizing Variability in Large-Scale, Accelerator-Rich Systems. *arXiv preprint arXiv:2208.11035* (2022).

[52] Shaden Smith. 2023. Pipeline parallelism. https://www.deepspeed.ai/tutorials/pipeline/#load-balancing-pipeline-modules

[53] Masahiro Tanaka, Kenjiro Taura, Toshihiro Hanawa, and Kentaro Torisawa. 2021. Automatic graph partitioning for very large-scale deep learning. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1004–1013.

[54] Yi Tay, Dara Bahri, Liu Yang, Donald Metzler, and Da-Cheng Juan. 2020. Sparse Sinkhorn Attention. arXiv:2002.11296 [cs.LG]

[55] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2022. Efficient Transformers: A Survey. *ACM Comput. Surv.* 55, 6, Article 109 (dec 2022), 28 pages. doi:10.1145/3530811

[56] LLaMA-MoE Team. 2024. LLaMA-MoE: Building Mixture-of-Experts from LLaMA with Continual Pre-training. https://github.com/pjlab-sys4nlp/llama-moe

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[58] Lean Wang, Huazuo Gao, Chenggang Zhao, Xu Sun, and Damai Dai. 2024. Auxiliary-Loss-Free Load Balancing Strategy for Mixture-of-Experts. arXiv:2408.15664 [cs.LG] https://arxiv.org/abs/2408.15664

[59] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. 2022. Efficient DNN Training With Knowledge-guided Layer Freezing. *arXiv preprint arXiv:2201.06227* (2022).

[60] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew Dai, Zhifeng Chen, Quoc Le, and James Laudon. 2022. Mixture-of-Experts with Expert Choice Routing. *arXiv preprint arXiv:2202.09368* (2022).

[61] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M. Dai, Zhifeng Chen, Quoc V. Le, and James Laudon. 2022. Mixture-of-Experts with Expert Choice Routing. In *NeurIPS*.

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