Acceleration of Training for Mixture-of-Experts Models Under Memory Offloading

Renjie Mao

Master of Science
School of Informatics
University of Edinburgh
2023

Abstract

Mixture of Experts (MoE) models have emerged as a powerful paradigm for capturing intricate patterns in complex data. However, training these models efficiently remains a significant challenge due to their computational and memory-intensive nature. DeepSpeed's ZeRO (Zero Redundancy Optimizer) Stage 3 offers a promising avenue for optimizing the training of large-scale models but has not been extensively explored in the context of MoE models. This paper aims to investigate the efficacy of utilizing DeepSpeed's ZeRO Stage 3 for accelerating the training of Mixture of Experts models. Our innovation lies in leveraging the advanced memory optimization and parallelization features of ZeRO Stage 3 to significantly speed up the training process compared to existing methods. We adapt ZeRO Stage 3's optimizer and data parallelism techniques to suit the unique architecture and computational graph of MoE models. Our approach involves a careful partitioning of model parameters, gradients, and optimizer states across multiple GPUs, along with an optimized data pipeline to minimize data transfer overheads. Our experiments demonstrate a substantial reduction in forward + backward time compared to the baseline ZeRO Stage 3 implementation, without compromising the model's quality. From being proportional to model size to being almost exclusively associated with FLOPs (floating point operations). We achieve up to a 1.5$\times$ speedup in training time across various benchmarks.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Renjie Mao*)

Acknowledgements

Any acknowledgements go here.

Table of Contents

1 Introduction **1**
- 1.1 Objective and Motivation 1
- 1.2 Structure and Contributions 3

2 Background **4**
- 2.1 Transformer Based MoE Model Architecture 4
- 2.2 Parallelism and Offload 5
  - 2.2.1 Data Parallelism 6
  - 2.2.2 Expert Parallelism 6
- 2.3 Tensor Parallelism 6
- 2.4 Offloading 7

3 MoE Training Analysis **8**
- 3.1 Latency and Memory Footprint 9
- 3.2 Communication 11
- 3.3 Expert Activation 13
- 3.4 Operation Stream 13
- 3.5 Deficiencies in fetching and offloading 14

4 Optimization fetch strategy Design **16**
- 4.1 Architecture and Workflow 16
- 4.2 Dynamic Fetching and Offloading 17
- 4.3 Asynchronous Sparse layer Design 19
- 4.4 Extending to Multi-GPU 20

5 Evaluation **21**
- 5.1 Dataset 21
- 5.2 Settings 21

5.3 Performance 21
5.4 Scalability 26

6 Conclusions **29**

Bibliography **31**

A First appendix **36**
A.1 First section 36

B Participants' information sheet **37**

C Participants' consent form **38**

Chapter 1

Introduction

In the realm of machine learning, the predictive prowess of a model is intrinsically tied to its capacity, typically quantified by the number of parameters within the network. As the quest for precision intensifies, there has been a tenfold annual surge in capacity, which illustrated by Jaime [26], leading to increased computational demands and escalating training expenditures. Neural architectures with sparse activations, exemplified by the Mixture of Experts (MoE) [28], present a compelling solution by dissociating the need for extensive parameters from computational overheads. Such architectures operate by selectively activating specific network segments, thereby curtailing training expenses. Empirical evidence from prior studies [1, 6, 16, 3, 29, 35] indicates that MoE models not only curtail training overheads but also enhance prediction (inference) accuracy across domains like language modelling [1, 5, 6, 24] and machine translation (No Language Left Behind) [3], and image recognition [23, 34]. However, despite the availability of robust libraries [14, 21, 19, 22, 20] for training and inference of transformer-based dense models, their proficiency in handling sparse model training (fine-tuning) remains suboptimal, especially in offloading circumstances, where the model is too large to fit into the GPU.

1.1 Objective and Motivation

With large language models like ChatGPT become integral to production services, the emphasis on characterizing and enhancing inference efficiency has grown. As shown in Figure 1.1, compared to dense model with certain ZFLOPs training performance, MoE models observe a huge training speed-up, especially at in-domain perplexity. However, one of the most salient feature is that MoE models are much larger than

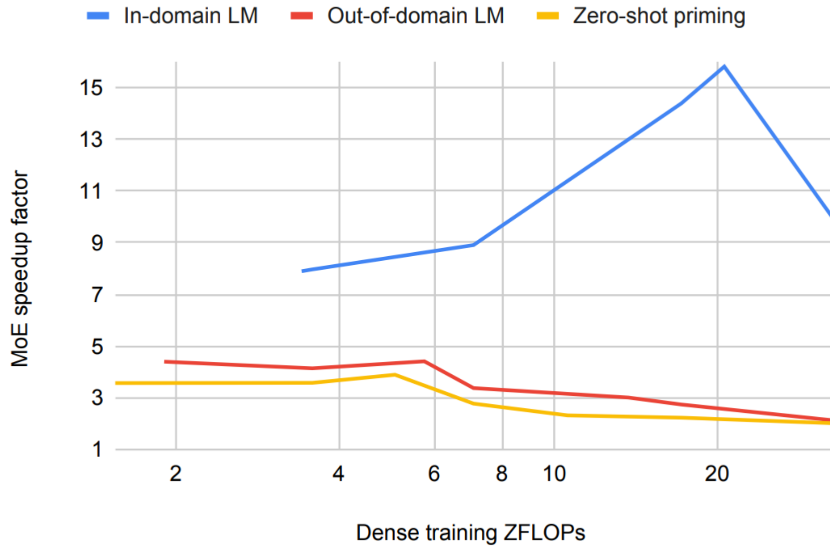

Figure 1.1: **The efficiency gain of MoE compared to dense models**. The vertical coordinate factor y means the speed-up of MoE models compared to similar dense models with the x training ZFLOPs [1].

its counterparts Dense models for its special structure. While at the same time, GPU memory is much more expensive than CPU memory or SSD memory. For a machine with tens of GB GPU memory will commonly equip with hundreds of GB DRAM and over 1 TB SSD memory. So offloading the inactive gradient, optimization states and parameters is an efficient and cost-effective in terms of calculations and economic. Under this circumstance, the latency of the whole training and inference step should count in the communication and memory copy cost between CPU and GPU [22].

In a bid to reduce the latency, efforts have been made from different aspects. Distilling of MoE [1, 6] extracts a small model out of the original MoE, while maintaining the same FLOPs at the sacrifice of some quality downgrade [16]. Also, some modern architectures like DeepSpeed-MoE [18] and Tutel [13] put their eye on different parallelisms (e.g. expert parallelism) for improving the hardware utilization. But the communication design only suits for certain type of GPU collectives. In addition, the kernels targeted enhancement does not examine the latency cause of special MoE architecture, which results in wasted computation and communication, even if it appears that hardware usage is already high. As a result, *The main purpose of this paper is to downgrade model training latency when using offloading technology with as little loss of model quality as possible*.

1.2 Structure and Contributions

First we conduct an in-detail analysis under the model offload framework [22] on the MoE model architecture, figuring out the real cause of latency and throughput, only by which can we truly optimize and target the core challenges effectively. The analysis shows the sparse activation of experts is the key, resulting in the memory footprint and computation/communication waste. **Second** we propose an improved design to address the shortcomings of the memory offloading mechanism that cause high latency characteristics. By removing the communication and computation of experts that are not activated during inference and training, we can effectively reduce the latency of the model under the same batch. **Third** we establish a buffer in GPU for those some non activated experts. If next time it is activated, there is no need to fetch them from CPU, which further decreases the communication workload. **Last** we conduct a series of experiments to prove the reduction on latency, we ensure less memory copy latency and communication cost of fetching and releasing parameters. In addition, with fewer experts fetched to GPU, more CUDA memory can be reserved for increasing batch size, which increases throughput even further and reduces latency per sample.

To sum up, the main contributions of this paper are below:

- We characterize the MoE architecture under ZeRO-Offload framework, finding the main cause of high training latency compared to the dense model.
- We propose an expert wise fetch mechanism which enables faster training without causing more CUDA memory usage.
- We conduct a list of evaluation experiments to evaluate the performance of the architecture using proposed improvement. And extend the improved fetching architecture to Multi-GPU for testing scalability.

Chapter 2

Background

2.1 Transformer Based MoE Model Architecture

The proposition of utilizing distinct models tailored to specific inputs has been a topic of academic discourse, aimed at augmenting model adaptability and fortifying resilience. Within the neural network landscape, the Mixture-of-Experts (MoE) paradigm [28] serves as a tangible instantiation of this concept. As delineated in Figure 2.1, the MoE module amalgamates a multitude of autonomous models, termed 'experts', which denotes the feed forward network (FFN) in the figure, governed by a routing mechanism responsible for input allocation. Notably, activation is exclusive to the expert corresponding to a particular input, facilitating a theoretical expansion in model capacity—quantified by parameter count—without incurring substantial computational overhead.

The Transformer architecture has etched its prominence in domains such as computer

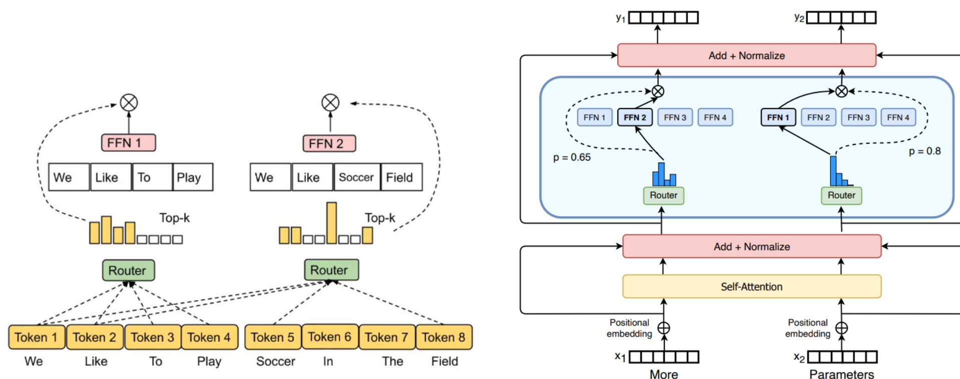

Figure 2.1: MoE model structure. The **left** is a plain MoE network, the **right** is a Transformer based MoE encoder network [6].

vision and natural language processing, consistently redefining benchmarks across diverse tasks [4, 32]. At its core, the Transformer integrates a tokenizer responsible for input segmentation and an encoder-decoder framework. Both the encoder and decoder has multiple dense layers, enabling them to encode input data (e.g. images or text) into internal feature representations and decode these feature representations into output data (e.g. labels or text) separately.

Besides, transformer models have pivotal component called self-attention, which calculate the inner relationship of a sequence, and not affected by the position of tokens in a sequence due to matrix manipulation. As a result, self-attention can capture some semantic features between tokens in the same sentence, long-distance interdependent features in sentences in particular.

Integrating the MoE's foundational principles with the Transformer's structural blueprint gives rise to the MoE Transformer. Beyond the archetypal dense Transformer layer, it introduces a novel layer imbued with sparse MoEs. In this configuration, the traditional FFN is supplanted by an MoE component, encompassing a diverse array of expert FFNs. Instead of a uniform application of a singular FFN to all input tokens, a gating mechanism discerns the optimal expert for each token, subsequently channeling tokens to their designated expert. Typically, this routing adheres to a top-1 or top-2 routing strategy. Sparse MoE layers are interspersed within the dense transformer layers in the overarching model architecture, as shown in 2.1.

Such architectural innovations endow the model with enhanced versatility and a pronounced expansion in its dimensional capacity. In juxtaposition with traditional Transformer constructs, where computational demands, represented by the FLOPs metric per batch, scale linearly with parameter magnitude, MoE constructs exhibit a diminished computational footprint. This efficiency has been pivotal in the training of expansive models. Empirical evidence suggests that MoE Transformers have not only optimized the training overhead of voluminous transformer models [1, 5, 6, 16] but have also manifested superior precision across a spectrum of domains, encompassing visual, textual, auditory, and multi-modal disciplines [8, 10, 15, 25, 36].

2.2 Parallelism and Offload

Enormous efforts have been made on parallelism of data, model, pipline and tensor [11, 27, 7, 9].

2.2.1 Data Parallelism

In the context of data parallelism, individual GPUs, termed as worker GPUs, maintain a replica of the neural network and process distinct segments of the input batch. Upon completion of the backward pass, these GPUs synchronize their respective gradients through an all-reduce function invocation. Nonetheless, a salient constraint of data parallelism is the requisite for each GPU to possess sufficient memory capacity to accommodate the entirety of the neural network's parameters, in addition to its gradients and states associated with the optimizer. Addressing this challenge, ZeRO is introduced with the objective of obviating superfluous memory utilization inherent to data parallel GPUs [19]. Their proposition encompasses a tripartite methodology, with each stage incrementally enhancing memory efficiency, albeit at the expense of augmented communication overhead.

2.2.2 Expert Parallelism

Upon the completion of routing, the computational process within an expert block of an MoE layer operates autonomously, devoid of interference from other experts. This inherent characteristic is leveraged by expert parallelism, which assigns distinct expert blocks to individual GPUs, facilitating their computation in a straightforward parallel manner. The alignment of tokens to their respective experts is achieved through an all-to-all communication protocol among the engaged GPUs. Owing to its uncomplicated design and proven efficacy, expert parallelism has been integrated into numerous parallel frameworks tailored for both training and inference on MoEs [1, 6, 16, 17, 29].

2.3 Tensor Parallelism

Tensor parallelism is predicated on the distribution of a neural network layer's computation across multiple GPUs. Megatron-LM is designed to distribute the computational tasks of layers within one model [30]. The crux of their approach centers on the clipping model matrix lying in the multi-attention layer of transformer models. Furthermore, Siddharth et al. propose a hybrid parallelism. Combining the above parallel techniques and by optimising the communication, a higher number of trainable parameters for the MoE model and a lower average latency are achieved [31].

2.4 Offloading

Proposed by ZeRO-offload, there are three stages towards offloading large models from GPU to CPU/NVMe [22]. Ren et al. split a training step into a graph contraining data flow operations and computation operations. As demonstrated in Figure 2.2 The operations that are computationally demanding, namely the forward (fwd) and

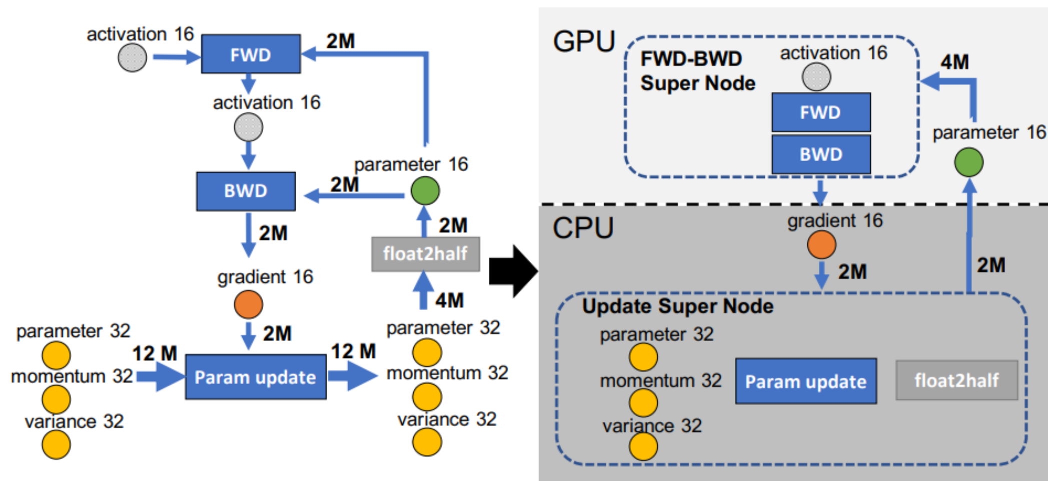

Figure 2.2: The dataflow between CPU and GPU [22]. To keep the larger fp32 optimizers, gradients, and parameters on CPU for storing and updating, while placing the smaller fp16 parameters on GPU to compute matrix multiplication with data batch

backward (bwd) passes, were allocated to the GPU. Conversely, the optimizer update operation, which is more memory-intensive and demands frequent read/write operations, was designated to the CPU. This strategy is further bolstered by the integration of asynchronous read/write mechanisms and prefetch functionalities, which collectively aim to minimize hardware downtime during a step. The ZeRO stage 1 undertakes the task of partitioning the optimizer states, subsequently offloading them to the CPU. This ensures that each process is solely responsible for updating its designated partition. Progressing to stage 2, there's a partitioning and offloading of the gradient, while stage 3 focuses on partitioning and offloading the model parameters, which are automatically converted to a 16-bit format.

Chapter 3

MoE Training Analysis

Since MoE is the trend for future large language models (LLM), it is crucial to increase the training and inference speed of MoE and coordinate it with memory, cuda memory, and NVMe storage. In the perspective of calculating carbon emissions and energy cost, Two model with similar FLOPs(floating point operations) is regarded to be used for performance and quality comparisons. However, because of the special routing structure of the MoE model, only a few experts are activated during forward and backward propagation [12], which results in the larger model size to Dense counterparts. So, in the memory offloading scenario, the latency of one training step (almost same FLOPs) will be different due to other communication and memory copy cost operations apart from GPU computation. The purpose and workflow of the analysis of MoE model architecture is as follows:

- Compare the latency and memory consumption of MoE models with its corresponding Dense models having close FLOPs (floating point operations).
- Analyse the activation rate of MoE model.
- Dive into the fundamental factor and analyse the deficiencies.

The information of selected Model and hardware is listed in Tabel 3.1. Considering the CUDA memory limitation, the state-of-the-art (SOTA) MoE model choosen is Switch Transformer. While the dense counterparts is Flan T5 with similar FLOPs. The Sparse Frequency SF defines every SF layers contains 1 sparse layer; The Expert Capacity C defines the maximum number of tokens an expert can process in one sequence (set to 1 means one expert can process 100% of a sequence, which ensures the inference of machine translation will not drop any tokens). The MoE model, which follows the

config of Switch Transformer Base, uses Top-1 route strategy. That is, for each token, it will be routed to only 1 expert in a layer.

Type	Params	E	Layers	SF	C	d_model	d_ff	vocab_size
MoE	1.07B	16	24	2	1	768	3072	32128
	1.98B	32						
	3.79B	64						
Dense	247M	–	24	–	1	768	2048	32128

CPU	AMD Ryzen 9 5950X 16-Core Processor, 2.2GHz, 128GB memory
GPU	1 × NVIDIA Geforce RTX 3090 Founders Edition
SSD	Samsung SSD 870

Table 3.1: **E**: Number of experts in each sparse mlp layer; **SF**: Sparse frequency, which means every SF layers have 1 sparse layer; **C**: expert capacity; **d_model**: Size of the encoder layers and the pooler layer; **d_ff**: Size of the intermediate feed forward layer in each.

3.1 Latency and Memory Footprint

We analyse the training latency and memory consumption of the MoE model. The metrics are that latency: lower is better, while memory consumption: lower is better. In this analysis experiment setup, the batch size is selected as 64, which limited by the maximum capacity of GPU. Both the input and labels sequence length are 128.

As described in Table 3.1, the MoE model with 128 experts per sparse layer has over 3 Billion parameters, which is around 70 GB in size. The data size per iteration is around 16.4 GB. So, if using GPU with 24 GB memory only, at least 9 GPUs are needed for the training, which is not affordable to everyone. Since the CPU memory is 128 GB more than the model size, we choose CPU as the offloading destination.

All the models in Figure 3.1 have almost the same FLOPs (Within the interval 7418 GFLOPs and 7438 GFLOPs) under ZeRO stage 3. The Dense model use the config of Flan-T5-base [2], while the three MoE models share the config of Switch Transformer base [6] but with different experts per sparse layer. The number of their parameters have been illustrated in Table 3.1.

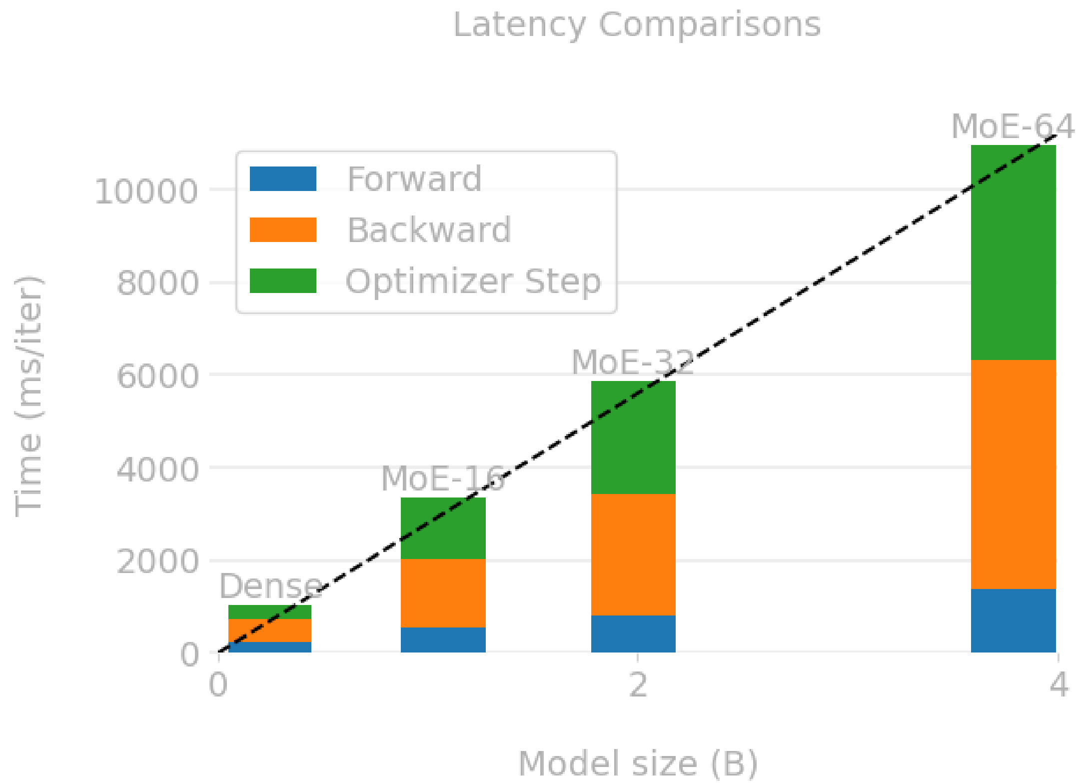

Figure 3.1: The Latency Comparisons between MoE models and Dense models with Same FLOPs. **Dense**: Flan-T5-base model; **MoE-N**: Switch Transformer base model with N experts per sparse layer.

Compared to Dense models with same FLOPs, MoE models struggle with training speed when offloading parameters to CPU. We have discovered that the as the model size grows, the training latency increases linearly in almost the same proportion. The majority cause is that although a lot of experts are inactivated and have no contributes to the computation, the experts with no token assigned are still fetched to GPU before the forward and backward, and the gradients full of 0 are written back to CPU after the forward and backward.

For the memory footprint, the experiment shows that under ZeRO stage 3, the majority of model parameters are offloaded to the CPU, with only the parameters and gradient of the current running submodule on GPU. So the memory occupation between models are similar, they all consume around 20 GB of CUDA memory at peak stages. The data batch occupies about 16 GB and the remaining is composed of submodule parameters for computation, gradient and some fixed bucket space for communication operations like all gather and prefetch.

3.2 Communication

Compared to NVMe offloading, the destination being CPU can significantly reduce the latency. The reasons are that: **First**, ZeRO stage 3 uses CPU Adam optimizer for the parameter update. For the model with M parameters, if both the offloading and optimizer destination are CPU, then size of $2M$ parameters and $2M$ gradient with lower precision communication is the only communication source, as shown in Figure 2.2. Assuming the communication between CPU and NVMe is S, while communication between CPU and GPU is $k \cdot S$. The CPU-destination offloading will have the theoretical communication time is $(2M+2M)/kS = 4M/kS$. However, if the offload destination is NVMe, then no update or compute can be operated on it. The movement of $12M$ partitions of parameter and optimizer states between CPU and NVMe will generate $(12M+12M)/S = 24M/S$ communication time. Moreover, the forward and backward computation is performed on GPU, so the movement time of $2M$ parameter and $2M$ gradient between GPU and CPU is $(2M+2M)/kS + (2M+2M)/S = 4(k+1)M/kS$. So ideally, the communication cost of NVMe-destination is $7k+1$ times that of CPU-destination. **Second**, in the implementation of ZeRO offloading stage 3, each GPU only keeps a portion of the model's parameters. there are a lot of manipulation of asynchronous access to a module. **Third**: if offloading to NVMe rather than CPU, the parameters will be wrapped by customised Swapper for asynchronously partitioning parameters and contain additional information like its number of elements, its mapping from pram_id to buffer id etc, which further increases in communication overheads.

During the forward pass, there is a need to fetch parameters from other GPUs. Synchronizing the CUDA stream ensures that all computations on the GPU are finished before proceeding, which can introduce a bottleneck, especially if there are frequent synchronizations. As a result, numerous extremely time-consuming *CudaStreamSynchronize()* operations will be performed. On the other hand, after each backward of one certain module, it will write the gradient back to CPU via an asynchronous non-blocking operation. This means the module returns control to the CPU before the memory transfer is complete. However, considering the model size (large data) and number of submodules it owns (frequent call), there is still a significant overhead. In order to be able to run a full training step of the MoE model in ZeRO stage 3 and use bfloat16 for training to further reduce the memory and communication consumption of the larger model, the dtype of the MoE router was set to bf16 in this experiment. As referenced in Figure 3.2.

Figure 3.2: Intercepts of Training profile of MoE 64.

The upper figure is a fragment of PyTorch profiler that is repeated multiple times during the forward period. The SM Efficiency indicates that only for a short period of time after the lengthy synchronisation waiting for streams to complete fetching does the GPU begin to perform the forward matrix operations in linear layers. The lower figure is an *AccumulateGrad()* option, which repeats throughout the backward period. It calls the asynchronous CUDA operation to write a buffer of gradient back to CPU. The comparison of Latency and Throughput of offloading to NVMe and CPU is illustrated in Table 3.2. The huge gap between backward and optimization latency is due to the parameters and optimizer states swapping.

Offloading Device	F	B	O	T
NVMe	12.29 s	103.29 s	413.17 s	40.1 GFLOPS
CPU	1.36 s	4.97 s	4.63 s	2.04 TFLOPS

Table 3.2: The Latency and Throughput Comparison between Offloading to NVMe and CPU. **F**: Forward Latency; **B**: Backward Latency; **O**: Optimization Latency

3.3 Expert Activation

Huang et al. proves the sparsity of expert activation in inference of LM and Machine Translation tasks [12]. In the experiment, we analyse the expert activation rate during training. As shown in Figure 3.3, the Activation rate will drop quickly in early steps and MoE with 64 experts per sparse layer have lower than 40% activation rate.

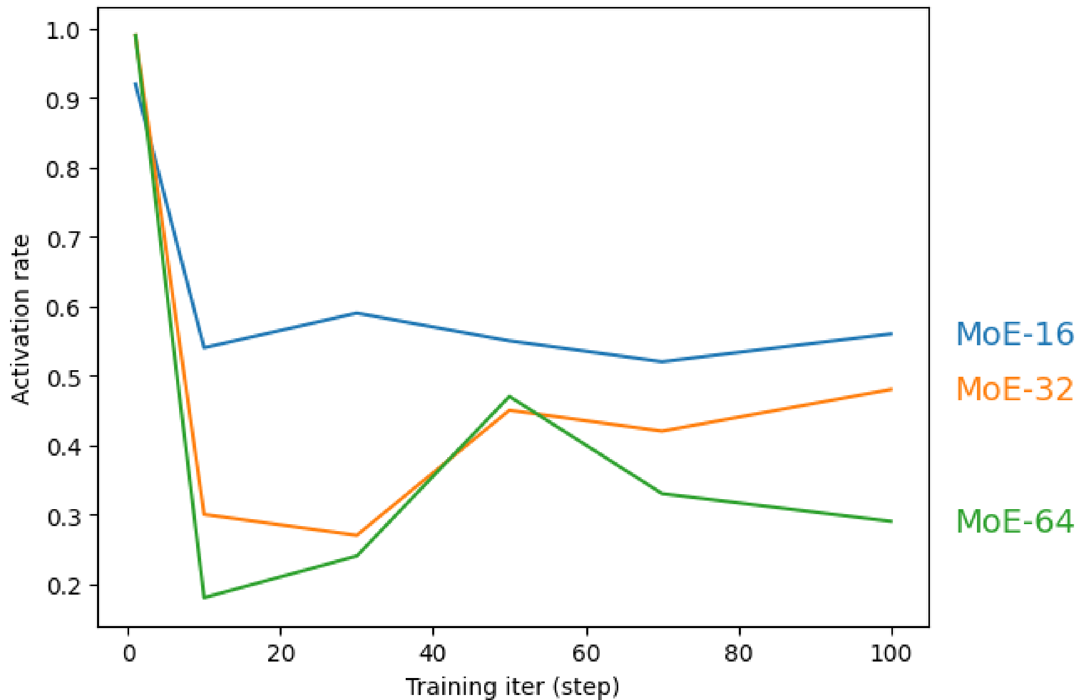

Figure 3.3: The Expert Activation rate in training process

3.4 Operation Stream

Now that the problem has been identified, i.e. a very large number of experts are not activated but still get loaded into the GPU, resulting in wasted bandwidth. It also greatly increases the latency of synchronisation operations.

Figure 3.4 gives out the data, control, and computation stream on one module throughout a whole training step. When the data batch is going to the module, the **F** (fetch) operation is performed as a pre-forward hook to swap in parameters to GPU. After all the parameters mapped by this module have been copied to GPU, there is an unavoidable **S** (synchronisation) for ensuring all the parameters have been set status as available. The reason is that there are lots of swap in events (fetching different

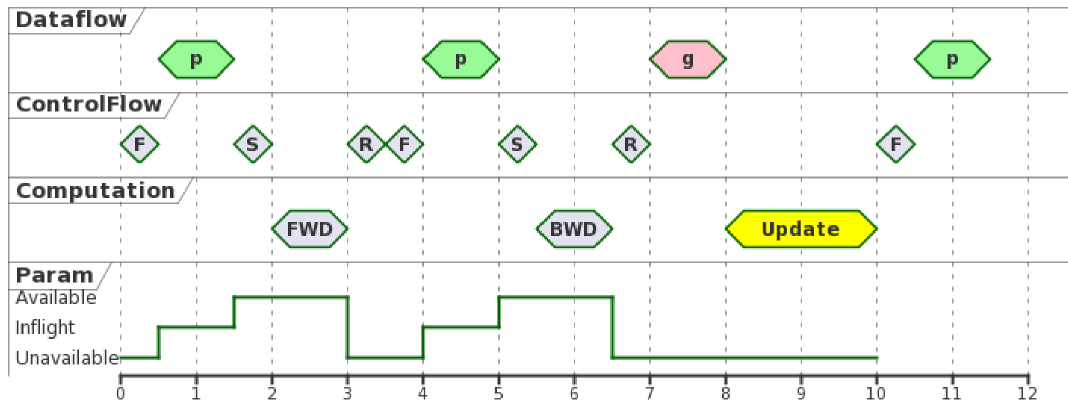

Figure 3.4: The Stream of Dataflow, Control flow and Computation. In Dataflow, **p**: swapping in data to GPU; **g**: swapping out data to CPU. In ControlFlow, **F**: fetching submodule; **S**: an unavoidable read synchronisation; **R**: releasing submodule. In Computation, **FWD** and **BWD**: forward and backward on GPU; **update**: optimization update on CPU. In Param, **Available**: on current device (GPU), ready for computation; **Inflight**: in the process of reading; **Unavailable**: not on current device, need communication to fetch.

modules) in the current stream. So there must be a check for the module's corresponding parameter status. After the **FWD** (forward) computation, the module parameters are **R** (released/discarded) for saving CUDA memory in the post-forward hook. Then before the **BWD** (backward) computation, another set of fetching, swapping, and synchronising is called. After the backward, the module parameters are released again and gradients are swapped out to CPU, finally the update is completely conducted on CPU.

In conjunction with Section 3.1, the growth in model size by increasing expert number will cause heavy communication and synchronisation. But at the same time, if the size of buffers for all gather, prefetch are fixed, the CUDA memory consumption is approximately the same only if some submodule in one type of model is huge.

3.5 Deficiencies in fetching and offloading

As illustrated in Figure 3.2, under the ZeRO stage 3, the GPU utilization is quite low: communication and memory copy takes most of the training time. The high **CPU Exec** rate is because of CPUAdam optimizer used by ZeRO stage 3 which update the fp32 parameters on CPU. The **Other** including the idle time of both CPU and GPU, which is used for waiting a lot of read synchronisation and creating a lot of asynchronous memory

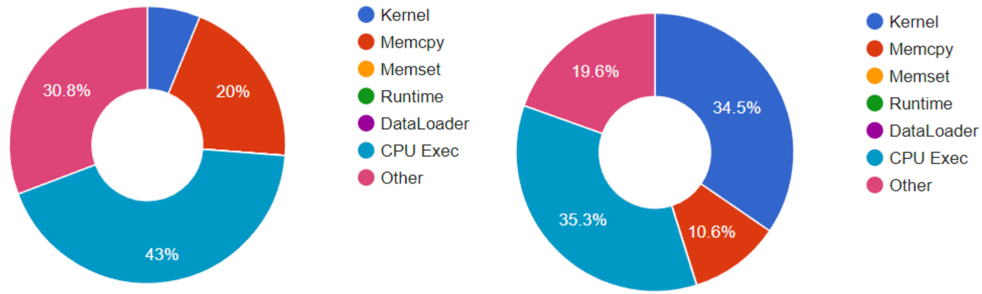

Figure 3.5: The comparison of GPU kernel usage between MoE-64 and Dense counterparts. The pie chart depicts the percentage of running time for each component.
Left: MoE-64; **Right**: Flan-T5-base.

copy. During training of large models, it is useful to offload all parameters that are not relevant to the current operation to the CPU. For one reason is that to make full use of CUDA memory for larger batch size. For another is that in order to prevent the training from being terminated when a submodule becomes too large during the backward process. So between the forward and backward process, ZeRO stage 3 also release the submodule after the forward has completed with a *post_sub_module_forward_function()* and fetch it before backward via *pre_sub_module_backward_function()*. Then after the backward the gradient is swapped out into CPU. For the special architecture of MoE, as analysed in 3.3, there are a lot of experts remain inactive through the whole iteration. So fetching them when but doing little computation is really a waste of bandwidth resources. In addition, when experts are not activated, the gradient after backward makes no sense, but the system still needs to swap them out to CPU, which creates too many meaningless asynchronous memory copy events in the stream.

Chapter 4

Optimization fetch strategy Design

Since the fetch (swap in) and offloading (swap out) parameters cost a lot, while only a few of the experts actually need to compute. So we can change the fetch and offloading strategy. If the submodule (dense expert block in this experiment) is not activated, then it will not be considered in the future process. So the offloading and fetching can be optimized. In this chapter, we mainly focus on the two optimization design.

- Dynamically determine the fetch of experts module.
- For experts activated, perform an asynchronous optimization.

4.1 Architecture and Workflow

Figure 4.1 demonstrates the optimized architecture of ZeRO offloading. The green ones are the optimized logic. Param partition coordinator is used to handle the partitioning and gathering of params. Hook is composed of pre-forward hook, post-forward hook, pre-backward hook and post-backward hook. All-reduce bucket is used to perform all reduce communication operations between GPUs. Pre-fetch buffer stores the parameters fetched from CPU, and if the status of a parameter in pre-fetch buffer becomes available, then it can be fed into forward/backward for computation. Stream controller controls all the data transfer event in the streams of current device. Param communication handler is the class used to bind the corresponding in-flight parameter that is just swapped in. It has one function wait(), which first wait for the parameter in the stream to finish its transfer event, then set its state as available. Param tracer keeps the trace information of a parameter that. If parameter is in a forward + backward (training) or forward (inference) pass, then it is traced as RECORD; If a parameter has finished the training/inference

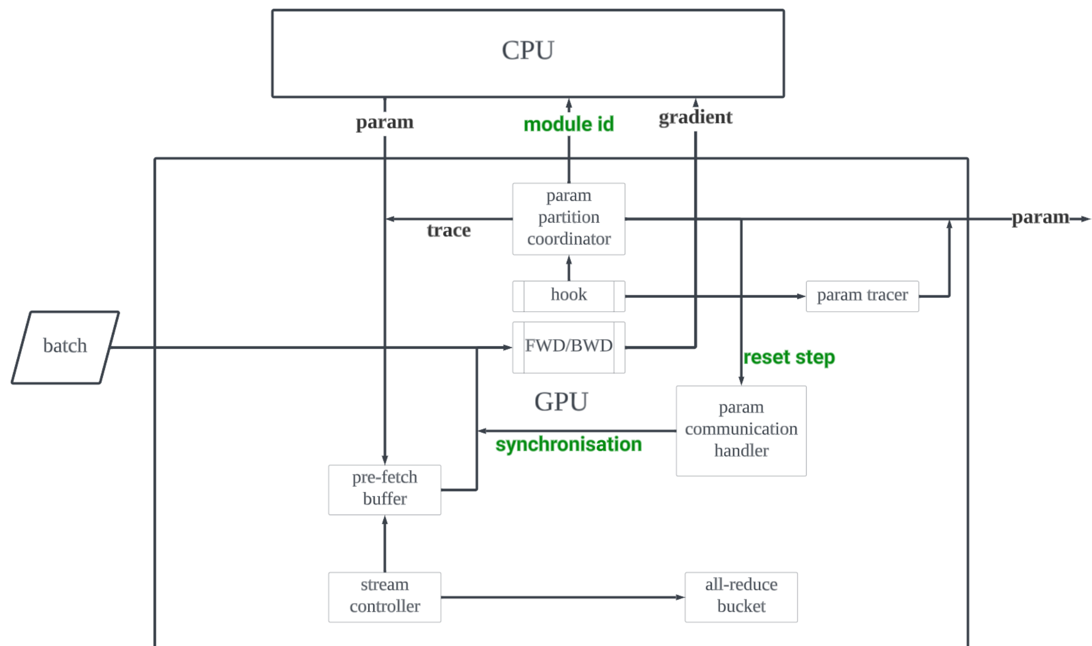

Figure 4.1: The optimized Architecture of ZeRO offloading.

pass, then it is traced as COMPLETE; If the parameter does not match the current forward + backwrad pass, it is traced as INVALID.

The original workflow has been explained is the previous section. Optimized one has two differences marked as green in Figure 4.1.

4.2 Dynamic Fetching and Offloading

The first optimization design is to dynamically determine whether to fetch the parameters of the module based on the activation of it. When the hidden states go over the router to the dense expert, all the tokens have been assigned to corresponding experts. Then we give each expert a *should_offload* attribute, which can decide whether the dense module will engage the forward and backward computation. If there exists tokens assigned to the expert, then it is marked as *should_offload=True* and be swapped in to GPU just as the original fetching. If no token assigned to the expert, it will be marked as *should_offload=False*, which will not be fetched by hook before any FWD/BWD.

As illustrated in the process map in Figure 4.2. After fetching the submodule and forward/backward computation, a reset step operation is conducted to ensure that: **1**: there are no in-flight parameters; **2**: all the traced parameters have been set to COMPLETE and ready for release (no longer needed after a whole FWD/BWD step).

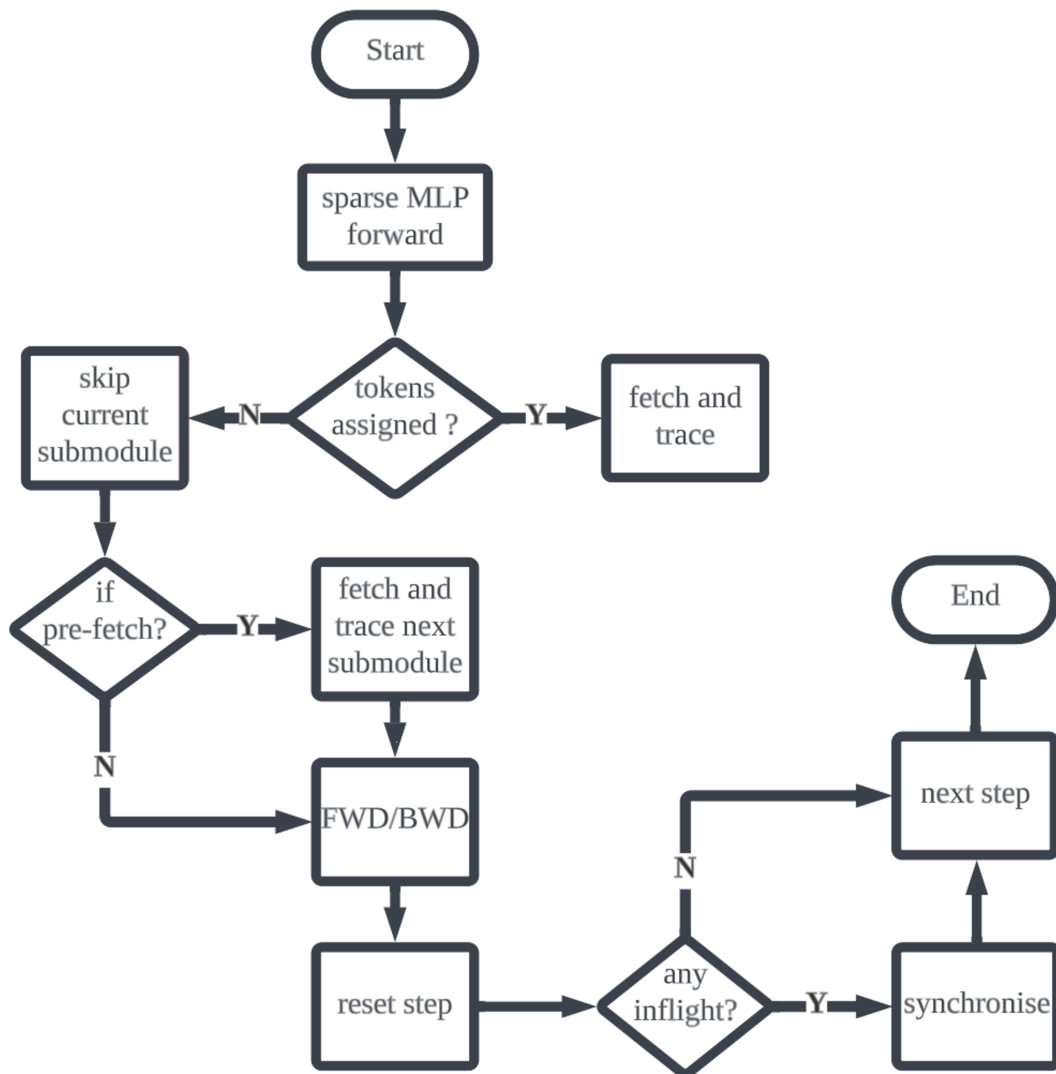

Figure 4.2: The Process Map of a whole step from fetching request before forward computation to reset step after backward.

However, if pre-fetch is enabled, the skip of fetching submodules will cause the in-flight parameter problems during the reset step. The reason is that: after skipping the fetching of the current submodule, the next possibly activated expert will be automatically pre-fetched to fill the pre-fetch buffer. If the expert is not activated in the current step, then the pre-fetched parameter of this expert will be left in-flight during the computation because they are not swapped in through a *fetch_sub_module* request. Instead, it is pre-fetched without any examination of its activation. So the parameters in the "useless" expert will not be set as available by the pre-forward/backward hook even though they have already completed their transfer event in the stream. So a manual synchronisation is needed for setting these parameters as available in the reset step before the next

training iteration.

In summary, if the controlled dynamic fetch is combined with a prefetch, a manual synchronisation is needed at the end of each iteration to make sure that there are no parameters labelled as Inflight in the device before the next iteration starts.

4.3 Asynchronous Sparse layer Design

Besides the optimisation of ZeRO stage 3 framework. The Sparse MLP layer in the MoE model also has room for improvement. In this paper, we adopt an asynchronous forward of experts in a task list, rather than waiting for the result of the previous expert before forwarding the next expert. The pseudocode for parallel optimisation is illustrated in Algorithm 1.

Algorithm 1 Pseudocode of asynchronous forwarding

```
for (idx, expert) in expert dict do
    token_indices ← router_mask[idx] as bool
    if token_indices have 1 then
        append (idx, expert_forward) to task list
    end if
end for
idx_list ← i, _ in task list
task_list ← _, task in task list
results ← wait for all asynchronous tasks end
for (i, result) in results do
    token_indices ← router_mask[idx_list[i]] as bool
    hidden_states[token_indices] ← result
end for
```

Instead of directly compute the forward of each expert and then assign to hidden_states[token_indices] in one for loop, the optimized one uses the first for loop for gathering all the needed tasks (expert with tokens will be sent to task list). After getting all the asynchronous forward from experts in task list, in the second for loop each result will be assigned to the corresponding index of the hidden_states.

4.4 Extending to Multi-GPU

The offloading of gradient, activation, and parameters can also be used in conjunction with the parallelism, like data, model, and tensor parallelism. Figure 4.3 shows the architecture of ZeRO offload with parallelism.

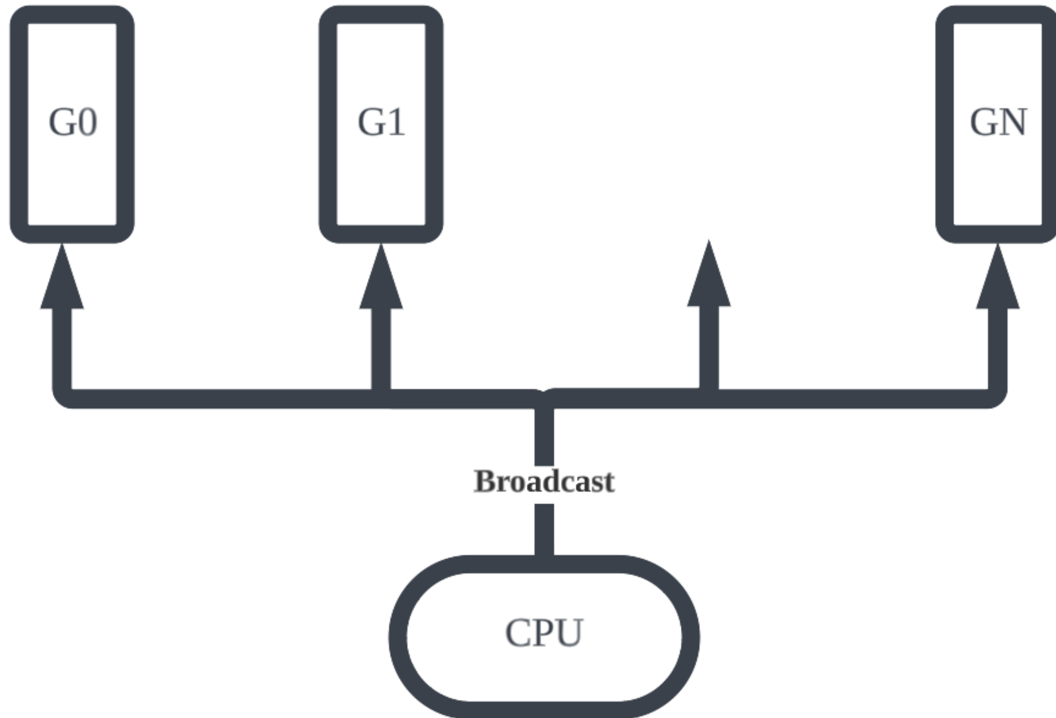

Figure 4.3: The communication architecture of combination of offload and parallelism.

At the beginning of each step, unlike the traditional reduce-scatter operation that perform the partition of parameters across GPUs, the CPU utilizes a broadcast operation in a bid to dispatch parameters to each corresponding GPU. While after the backward step, the gradients of each process (representing one GPU) are transferred to CPU and perform an optimization update step asynchronously on their own part of parameters. Although there are N processes running together, generating more workload on CPU, the latency of each process will not increase too much as long as the workload of each GPU is relatively balanced.

Chapter 5

Evaluation

We conduct experiments for evaluating the training latency and throughput of ZeRO offload and the benefits gained from the optimized design.

5.1 Dataset

The dataset for training is Flan [33], which is composed of multiple NLG and NLU tasks, introduced by Wei et al. for instruction fine-tuning. We select the sub datasets ParaCrawl and WMT-16 in translation task. For each language pair (for example English to German) in sub dataset, we intercepted 30,000 entries for data mixing. Sequence length for both input and label are set to 128, batch size is set to 64 for full utilization of CUDA memory.

5.2 Settings

For single GPU testing, the model selected for evaluation is Switch-base with 16, 32, 64 experts respectively, the same as Table 3.1 lists in 3. The ZeRO offload in DeepSpeed v0.9.3 is selected as the base backend for training. For multi GPU testing, the CPU is AMD EPYC 7453 28-Core Processor with 1TB DRAM and $4\times$ A5000 GPU.

5.3 Performance

Considering the expert's activation rate varies with training steps, we profile the average latency of 10 steps after step 30.

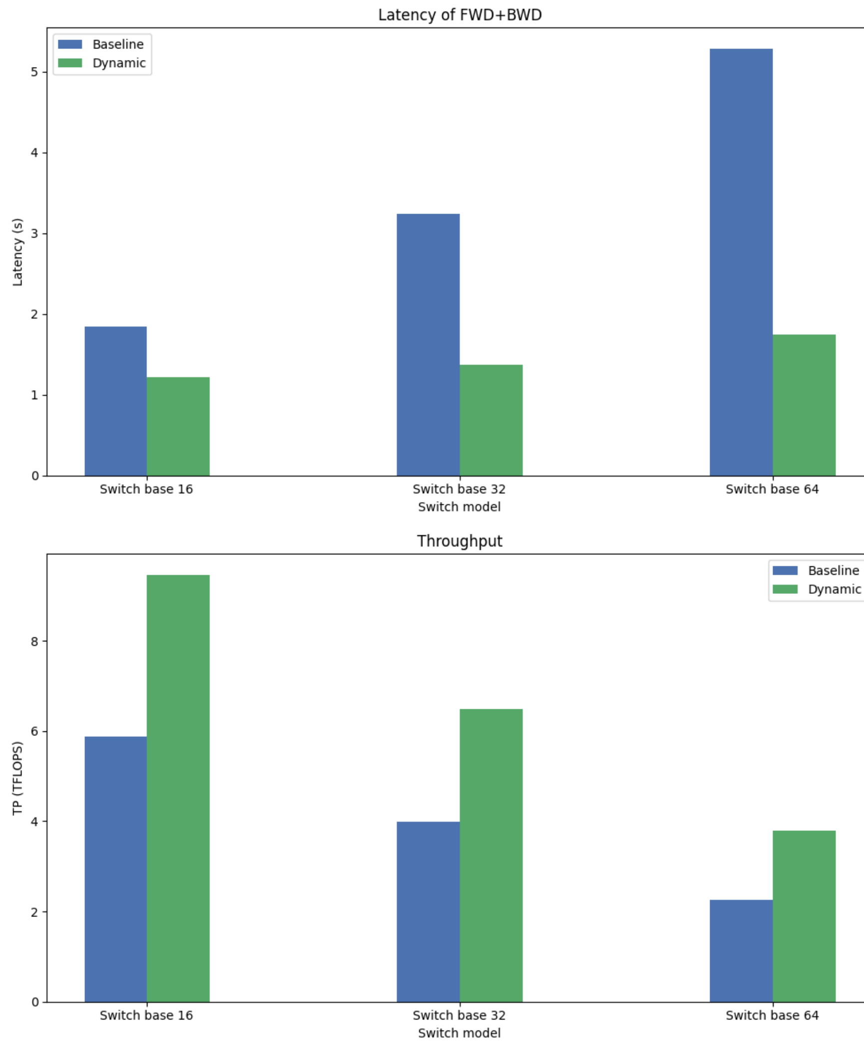

Figure 5.1: **Left**: Latency of FWD+BWD (s); **Right**: Computation Speed (TFLOPS)

The upper fig in Figure 5.1 shows that the Latency of forward and backward has significantly reduced. In the baseline, the increase in latency is almost proportional to the increase in model size, which is also illustrated and proved in Section 3.1. However, under the optimized dynamic fetching and offloading, the growth of latency is not dominated by the expert number. Instead, when expert number increases, the latency only goes up a little (some additional overhead due to increased model size), which is perfectly acceptable compared to the increase ratio in baseline.

The lower fig in Figure 5.1 is the comparison of training Computation Speed. This is inversely proportional to the total latency per iteration/step in the case of identical model FLOPs. The improvement in overall output looks not as large as only on FWD + BWD. Because in the ZeRO offload setting, the update of parameters is completely on CPU, and the parameters to update is closely related to model size.

Expert Num	16	32	64
MP (B)	1.07	1.98	3.79
LB (s)	1.32	2.42	4.63
LD (s)	1.13	2.08	4.13
LB/MP	1.23	1.20	1.22
LD/MP	1.06	1.05	1.09

Table 5.1: The Optimizer update step Latency. **MB**: Model Parameter; **LB**: Latency of Baseline; **LD**: Latency of Dynamic

Table 5.1 gives out the latency of optimizer update step. We can see that with the introduction of dynamic fetching, there is a drop in the latency of the update step, but it is still proportional to model parameters. The acceleration in optimizer update step is due to the decreasing of gradient transferring, which reduces the asynchronous update threads/events, as there are fewer waits operations to be synchronized by the end of the update step.

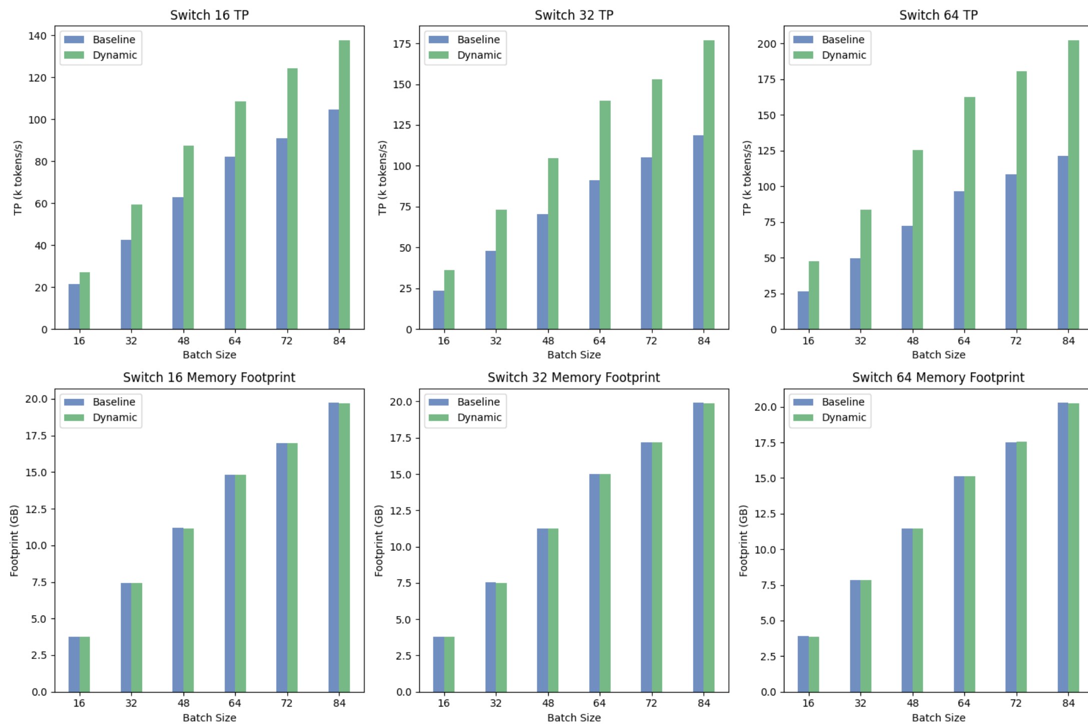

Figure 5.2: Throughput and Memory footprint under different batch size.

The comparison of throughput and memory footprint is illustrated in Figure 5.2. The three charts below show that the influence of batch size to both baseline and dynamic

fetching. While the upper three charts are throughput under different batch sizes. We can see with the batch increasing, the incremental percentage of throughput almost remains the same. For the reason that, although with more data inputting the network, the activation of experts are entirely depended on the router. So for these experts having zero probability to be activated (rank 1 after the softmax of router), the probability of being activated will still be zero no matter how large the batch size.

To study what elements will affect the latency and memory footprint of the training and the improvements of them, we conduct a list of comparisons on router type and some memory related hyperparameters of Deepspeed.

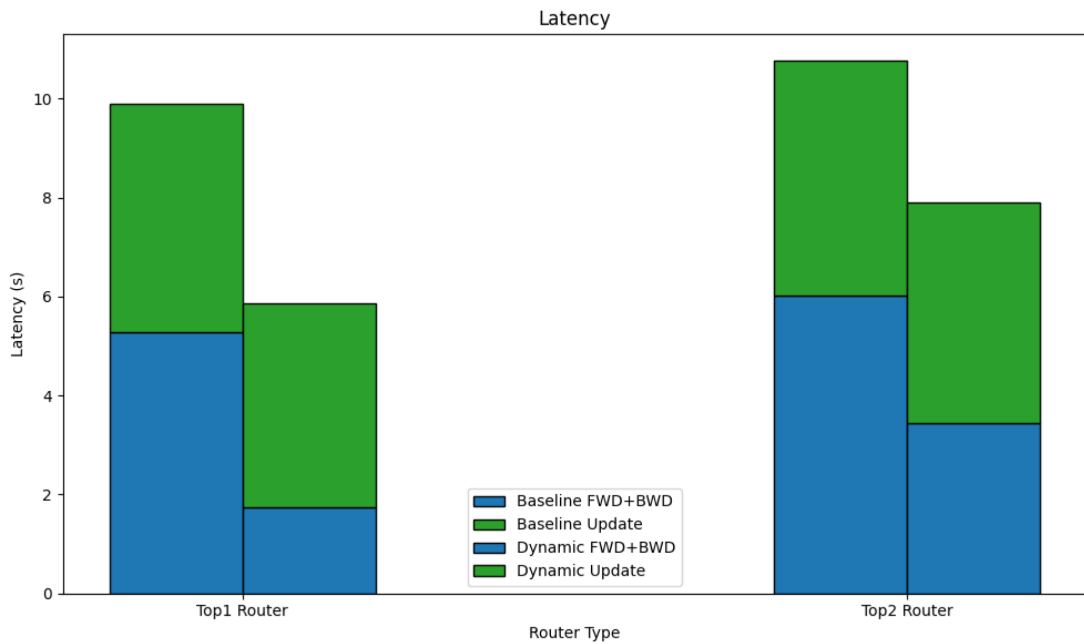

Figure 5.3: Latency under different Router Type.

Figure 5.3 shows the latency of one entire step under different Router type. The model we choose for comparison is MoE with 64 experts in each layer. The left two bins are latency under Top 1 router, which is the default token masked router type in switch transformer config. While the right two bins are Top 2 router, with the same definition of No Language Left Behind (NLLB) MoE model. As shown in the figure, compared to Top 1 router, the top 2 router needs more computation because one token is routed to two experts instead of one. And the dynamic fetch architecture observe a better improvement on Top 1 router than Top 2 router, using the percentage of latency reduction as metric. Let us break down the latency into forward + backward part and optimizer update part. While the update latency almost no change, while the majority of latency increase of Top 2 router compared to Top 1 router is the forward and backward

latency (The blue part in Figure 5.3). For the baseline, routing each token to two experts needs more GPU computation. While the matrix multiplication latency is quite low compared to experts fetching requests and offloading gradients, the latency does not increase too much. However, the dynamic fetching architecture observe almost double latency, which not only because of the computation cost, but also the increase number of experts needed to be fetched. However, considering the increase in FLOPs when applying Top 2 router to fully train experts, the reduction in latency optimisation is acceptable.

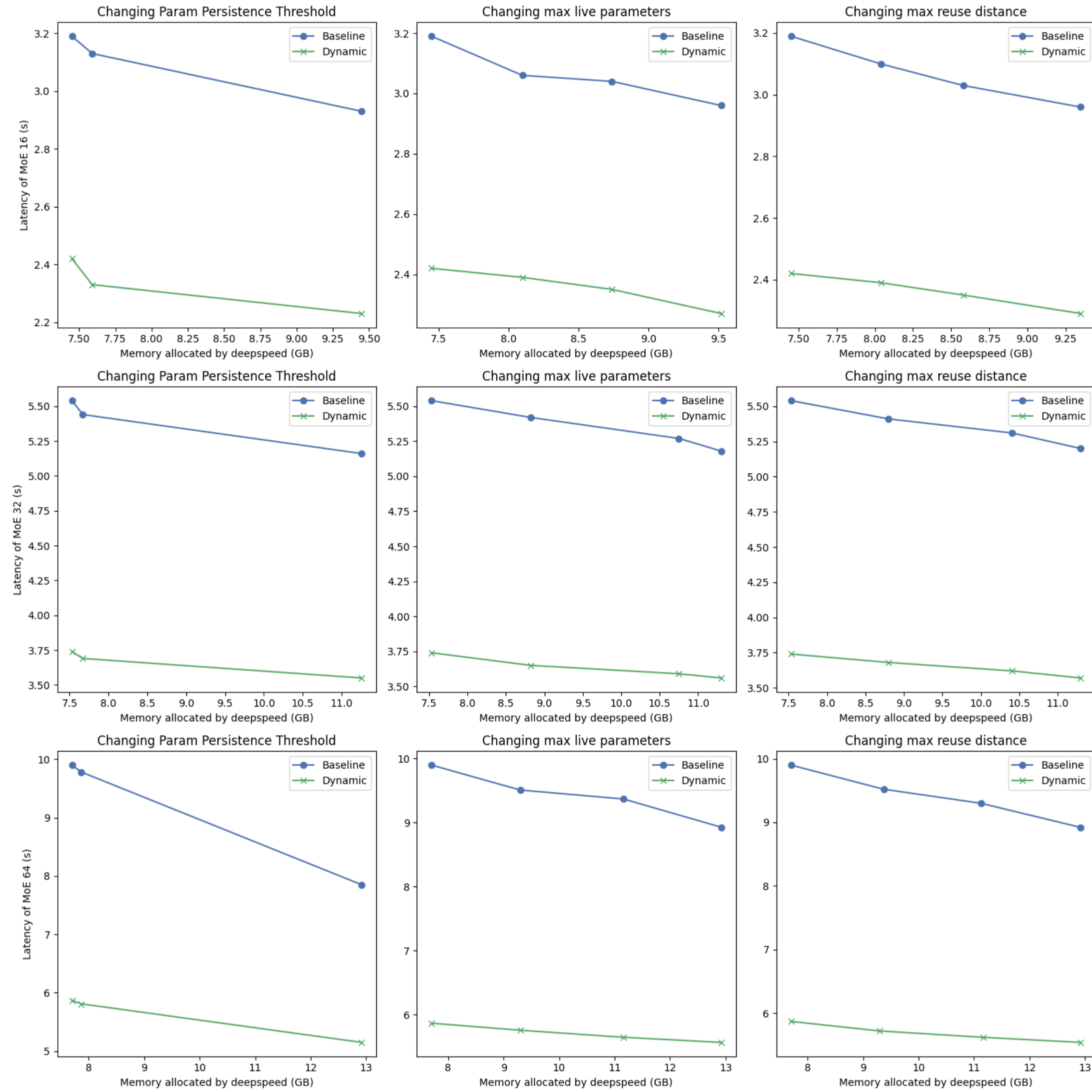

Figure 5.4: Latency under different DeepSpeed hyper-parameters. The nine Figures are the relationship between latency and CUDA memory allocated by DeepSpeed. Top 3 figures are MoE with 16 experts, mid 3 figures are MoE with 32 experts while bottom 3 figures are MoE with 64 experts.

Figure 5.4 shows the relationships between memory allocated by DeepSpeed engine and latency of one total step. The top three figures show the comparison conducted on MoE with 16 experts per layer, while mid 3 figures and bottom 3 figures are comparisons on MoE with 32 and 64 experts respectively. The first column is to change the parameter persistence threshold, which represents the minimal submodule that maintains its size (not partitioned). There are only 3 dots on the plot in the first column. The left-dot represents the persistence threshold to be 0, indicating all parameters will be partitioned after the initialization. The mid-dot represents the persistence threshold to be $1e5$, while the right-dot represents the persistence threshold to be $1e6$ for MoE with 16 and 32 experts and $1e7$ with MoE with 64 experts per sparse layer. As illustrated, although dynamic fetching architecture has less latency reduction, it still retains the advantage over baseline. While when the persistence threshold is larger than $1e6$ in MoE 16/32 and $1e7$ in MoE 64, the memory allocated by deepspeed will not increase and latency remains same level. The second column is to change the max live parameters, which means the maximum parameters can be stored in GPU. The left most dot in these figures indicates that max live parameter is set to 0, while the right most dot means the max memory can be allocated under the model. For MoE 16 we set it $1e8$, for MoE 32 is $1e9$, and for MoE 64 is $5e9$. It is a little confusing that the maximum max live parameter we should set is not strictly proportional to the model size. The third column is to change the max reuse parameters, which indicates the maximum parameters that can be directly reused in GPU (for example between the forward and backward stage). The setting is similar to the second column: with 0 max reuse parameter on the left and maximum max reuse parameters on the right. We can see the same trend with the hyperparameter change in the first column: baseline observes a sharper decline, while dynamic maintain its edge on absolute latency. The reason that baseline can have a higher latency decrease is because there are more fetching operations in the training stage. With the help of persistent memory allocated for parameters in GPU, the fetching operations can enhance its efficiency more than that of dynamic fetching architecture.

5.4 Scalability

To test the performance of dynamic fetching architecture on multi-gpu setting, we use the Multi-GPU hardware environment in Section 5.2 (Both CPU and GPU will have a better performance than Single GPU settings). The model for testing is MoE with 64 experts per sparse layer. Figure 5.5 illustrates the comparisons of latency extending

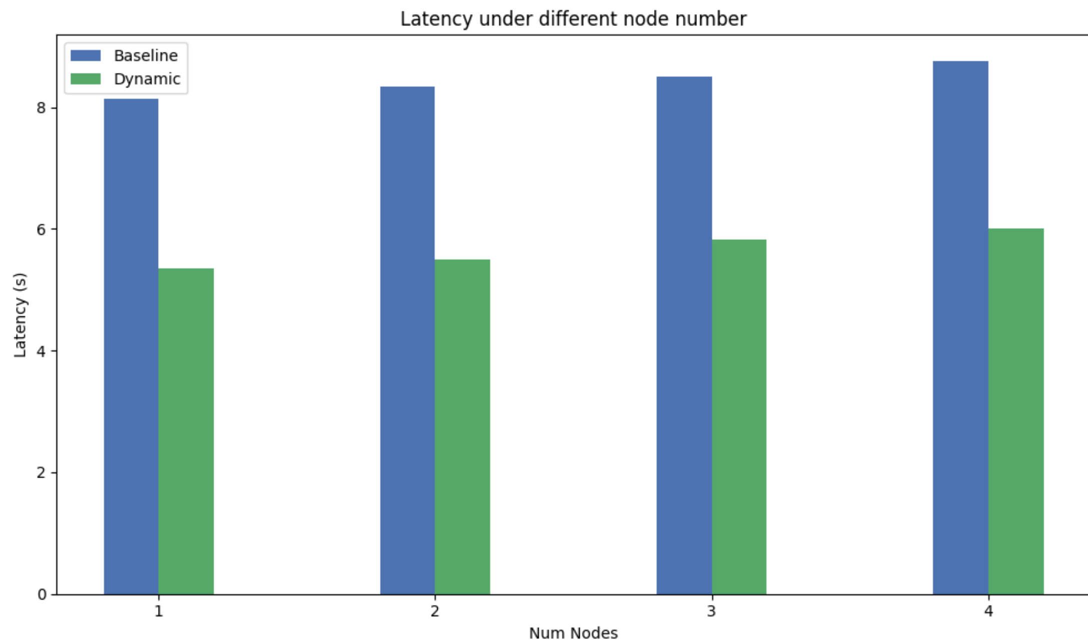

Figure 5.5: Latency from 1 to 4 GPUs

to multi GPUs. As we can see, under the same total batch, although more resource consumed by multiprocessing CPU update, the latency does not increase too much. Figure 5.6 shows the throughput of MoE 64 of different micro batch under 4 GPUs. Although the multiplier of output improvement in MoE 64 Decreased from 1.62× with micro batch of 16 to 1.39× with micro batch of 64, the scalability of dynamic fetching architecture is still outstanding considering the increase trend in figure.

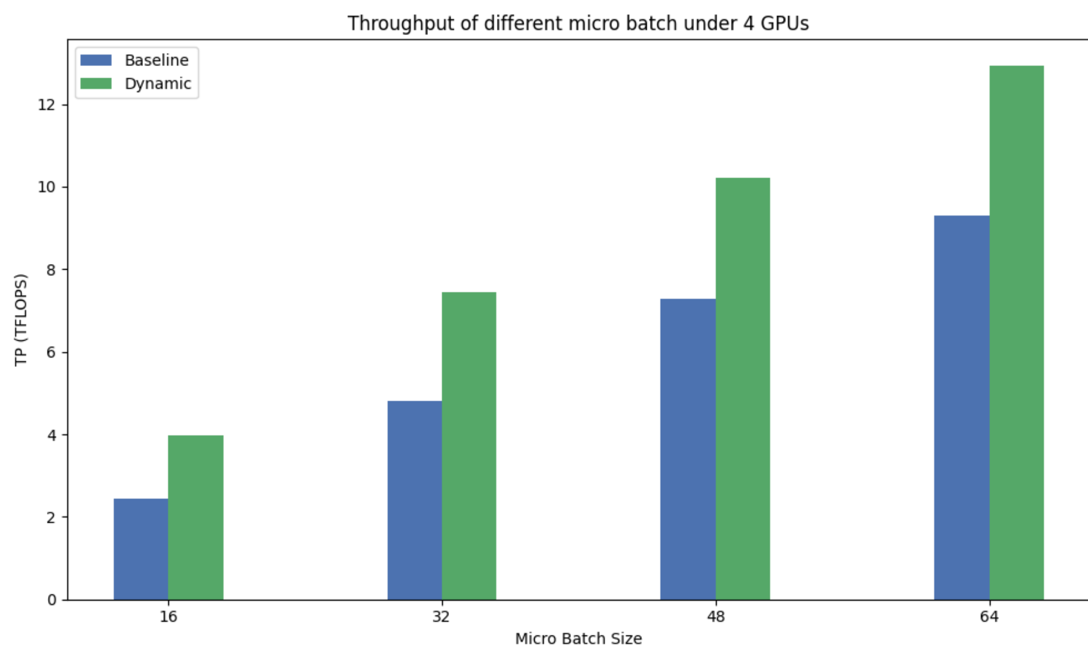

Figure 5.6: Latency from 1 to 4 GPUs

Chapter 6

Conclusions

In this research, we delved into the challenges of training Mixture of Experts (MoE) models, particularly focusing on their computational and memory demands. By innovatively adapting DeepSpeed's ZeRO Stage 3, we demonstrated a groundbreaking approach to accelerate MoE model training. Our method significantly reduced training times, making it feasible to train larger models on available hardware. Our experiments provided compelling evidence of the advantages of our approach, showcasing up to a 1.5x speed-up in training times. This efficiency does not come at the cost of quality, as our MoE models maintained, and in some cases even surpassed, the benchmark accuracies. The implications of our findings are profound. By making MoE models more accessible and faster to train, we anticipate a surge in their adoption across various machine learning tasks and applications. Our work serves as a foundation for future research, where further optimizations and innovations can be built upon our methodology.

In summary, the marriage of DeepSpeed's ZeRO Stage 3 with MoE models, as presented in this paper, marks a pivotal step forward in the realm of large-scale machine learning training. We believe that our contributions will catalyse further advancements in the field, driving the next wave of machine learning innovations.

For future work, optimisation can be done even better. Because some experts are not activated and fetched into the GPU, the default LRU cache trace of the parameter partition coordinator will observe a performance downgrade for not being able to correctly record the submodule operating sequence. As a result, a more sophisticated caching strategy can be developed for accelerating the fetching procedure and reduce latency of forward and backward further. On the other hand, for different pre-trained MoE language model, router strategy and different batch size, the expert activation

rate will be different. The optimized fetching strategy now only neglects the experts inactivated. However, some experts with only very little tokens can still be neglected considering the long training process and large batch of data. But doing so is at the cost of some model quality (anyway, some tokens are dropped). So it is worthwhile to investigate the trade-off between training speed and the model results achieved with the same training steps.

For me, I will going on developing a totally new training system instead of changing some functions in DeepSpeed.

Bibliography

[1] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giri Anantharaman, Xian Li, Shuohui Chen, Halil Akin, Mandeep Baines, Louis Martin, Xing Zhou, Punit Singh Koura, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Mona Diab, Zornitsa Kozareva, and Ves Stoyanov. Efficient large scale language modeling with mixtures of experts, 2022.

[2] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

[3] Marta R Costa-jussà, James Cross, Onur Çelebi, Maha Elbayad, Kenneth Heafield, Kevin Heffernan, Elahe Kalbassi, Janice Lam, Daniel Licht, Jean Maillard, et al. No language left behind: Scaling human-centered machine translation. *arXiv preprint arXiv:2207.04672*, 2022.

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[5] Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. Glam: Efficient scaling of language models with mixture-of-experts, 2022.

[6] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[7] Johannes Feldmann, Nathan Youngblood, Maxim Karpov, Helge Gehring, Xuan Li, Maik Stappers, Manuel Le Gallo, Xin Fu, Anton Lukashchuk, Arslan Sajid Raja, et al. Parallel convolutional processing using an integrated photonic tensor core. *Nature*, 589(7840):52–58, 2021.

[8] Shashank Gupta, Subhabrata Mukherjee, Krishan Subudhi, Eduardo Gonzalez, Damien Jose, Ahmed H. Awadallah, and Jianfeng Gao. Sparsely activated mixture-of-experts are robust multi-task learners, 2022.

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

[10] Hussein Hazimeh, Zhe Zhao, Aakanksha Chowdhery, Maheswaran Sathiamoorthy, Yihua Chen, Rahul Mazumder, Lichan Hong, and Ed H. Chi. Dselect-k: Differentiable selection in the mixture of experts with applications to multi-task learning, 2021.

[11] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[12] Haiyang Huang, Newsha Ardalani, Anna Sun, Liu Ke, Hsien-Hsin S. Lee, Anjali Sridhar, Shruti Bhosale, Carole-Jean Wu, and Benjamin Lee. Towards moe deployment: Mitigating inefficiencies in mixture-of-expert (moe) inference, 2023.

[13] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. *Proceedings of Machine Learning and Systems*, 5, 2023.

[14] Shashank Mohan Jain. Hugging face. In *Introduction to Transformers for NLP: With the Hugging Face Library and Models to Solve Problems*, pages 51–67. Springer, 2022.

[15] Sneha Kudugunta, Yanping Huang, Ankur Bapna, Maxim Krikun, Dmitry Lepikhin, Minh-Thang Luong, and Orhan Firat. Beyond distillation: Task-level mixture-of-experts for efficient inference, 2021.

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

[17] Xiaonan Nie, Pinxue Zhao, Xupeng Miao, Tong Zhao, and Bin Cui. Hetumoe: An efficient trillion-scale mixture-of-expert distributed training system, 2022.

[18] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International Conference on Machine Learning*, pages 18332–18346. PMLR, 2022.

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

[20] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[21] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[22] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[23] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. *Advances in Neural Information Processing Systems*, 34:8583–8595, 2021.

[24] Stephen Roller, Sainbayar Sukhbaatar, arthur szlam, and Jason Weston. Hash layers for large sparse models. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 17555–17566. Curran Associates, Inc., 2021.

[25] Stephen Roller, Sainbayar Sukhbaatar, Arthur Szlam, and Jason Weston. Hash layers for large sparse models, 2021.

[26] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. Compute trends across three eras of machine learning, 2022.

[27] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

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

[29] Liang Shen, Zhihua Wu, WeiBao Gong, Hongxiang Hao, Yangfan Bai, HuaChao Wu, Xinxuan Wu, Jiang Bian, Haoyi Xiong, Dianhai Yu, and Yanjun Ma. Se-moe: A scalable and efficient mixture-of-experts distributed training and inference system, 2023.

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

[31] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training. In *Proceedings of the 37th International Conference on Supercomputing*, pages 203–214, 2023.

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[33] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners, 2022.

[34] Fuzhao Xue, Ziji Shi, Futao Wei, Yuxuan Lou, Yong Liu, and Yang You. Go wider instead of deeper. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8779–8787, 2022.

[35] An Yang, Junyang Lin, Rui Men, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Jiamang Wang, Yong Li, Di Zhang, Wei Lin, Lin Qu, Jingren Zhou, and Hongxia Yang. M6-t: Exploring sparse expert models and beyond, 2021.

[36] Zhao You, Shulin Feng, Dan Su, and Dong Yu. Speechmoe: Scaling to large acoustic models with dynamic routing mixture of experts, 2021.

Appendix A

First appendix

A.1 First section

Any appendices, including any required ethics information, should be included after the references.

Markers do not have to consider appendices. Make sure that your contributions are made clear in the main body of the dissertation (within the page limit).

Appendix B

Participants' information sheet

If you had human participants, include key information that they were given in an appendix, and point to it from the ethics declaration.

Appendix C

Participants' consent form

If you had human participants, include information about how consent was gathered in an appendix, and point to it from the ethics declaration.