École Polytechnique Fédérale de Lausanne

Semester Project

Optimizing End to End Inference Time on MoE models

presented by

André Espírito Santo

Supervisors:

Prof. Dr. Anne-Marie Kermarrec

Milos Vujasinovic

Dr. Martijn de Vos

Dr. Rafael Pereira Pires

January 2025

"The function of good software is to make the complex appear to be simple."

– GRADY BOOCH

Contents

Abstract

Mixture of Experts (MoE) models have demonstrated potential for improving the quality of the network's outputs, particularly in language generation and language modeling tasks. Recent research has focused on designing specialized frameworks for training and inference of such models, as their sparse and dynamic nature makes using conventional Hardware Accelerators (HAs) inefficient. However, to the best of my knowledge, no existing work turns its attention to achieving perfect load balancing between HAs, irrespective of the skewness in routing decisions. Moreover, most identify communication as the primary bottleneck. Based on my findings, for the small input sizes typically used during inference and a limited number of HAs, the bottleneck lies in the number of kernels launched.

In this semester project I propose an algorithm that ensures theoretical perfect load balancing among HAs no matter how skewed the routing decisions are, which in turn reduces the idle time of each HA between each communication round, improving hardware utilization. Importantly, this solution is *dropless*, meaning that no tokens are ever dropped. The algorithm is grounded on the concept of Tensor Sharding (TS) applied to experts, wherein expert's weights are distributed across all HAs. This guarantees that every HA performs exactly the same amount of computation. This algorithm was tested on encoder-only and encoder-decoder architectures, where I show that this approach achieves a maximum speedup of 520% compared to my baseline.

Introduction

Machine learning (ML) has been growing rapidly in popularity, particularly with the rise of Large Language Models (LLMs). A phenomenon known as EMERGENCE is observed in LLMs, whereby larger models exhibit capabilities that smaller models do not, despite having the same type of model architecture [1].

This phenomenon has lead to the development of increasingly larger models, which now feature an unprecedented number of parameters [2, 3]. However, as the number of parameters grows, scaling these enormous models presents significant challenges. The computational budget needed for such models is substantial, which also raises concerns about environmental sustainability [4]. This situation has created a demand for conditional computation, which enables a sub-linear relationship between model size and computational cost. Conditional computation facilitates the scaling of models while utilizing only a subset of parameters for each input.

To address these challenges, MoE models have been introduced. In these models, certain layers do not utilize all weights to compute the output for each sample. Instead, the weights are partitioned into groups known as *experts*, and for each sample, only a subset $\mathcal{K}$ of these *experts* is activated [5]. On the other hand, MoE models are less efficient in HA utilization due to the dynamic nature of their computations, variable tensor sizes, and, in most real setups, communication overheads.

To mitigate these issues and to leverage batched matrix multiplication kernels for efficient *expert* computation, many models employ a CAPACITY FACTOR, which ensures an equal number of samples per *expert* [6]. However, this approach leads to some samples not being processed by any *expert* and relying solely on residual connections to progress through the model. This negatively impacts accuracy [7].

Thus, there is a pressing need to optimize both training and inference for MoE models. While many techniques have been proposed to improve training efficiency [8, 9, 10], inference optimization remains relatively underexplored.

In most MoE models, *experts* account for most model parameters (96.776% for encoder-decoder Switch Transformer with 128 *experts* [2]). In real-world inference systems, a cluster of HAs serves request concurrently using Data Parallelism. Hence, a common setup is to have the model replicated on each HA. However, due to the high MoE model sizes, they occupy most of memory of one HA alone. A prevalent solution is to distribute *experts* per HAs and use inter-HA communication to serve requests jointly.

Hence, in this report I develop an inference system for MoE models that minimizes end-to-end inference latency while maximizing throughput. A key requirement of this system is that it should be a *dropless* solution, ensuring that no samples are discarded, thereby maintaining maximum model accuracy.

Preliminaries

3.1 Mixture-of-Experts

Mixture-of-Experts (MoE) [5], short for Sparsely-Gated Mixture-of-Experts layer, consists of up to thousands of feed-forward sub-networks. This design enables conditional computation, where only a subset of these sub-networks processes certain tokens. For this reason, it is occasionally referred to as a sparse layer. A trainable gating network determines the specific sparse combination of *experts* utilized for each token.

This architecture facilitates scaling the number of parameters in the model while theoretically maintaining a sub-linear relationship with execution time.

Denote by $G(x)$ the output of the gating network and by $E_i(x)$ the output of the i-th *expert* network for a given input x. The output y of the MoE module is expressed as follows:

$$y = \sum_{i=1}^{n} G(x)_i E_i(x) \tag{3.1}$$

The sub-linear relationship arises from the sparsity of $G(x)_i$, which implies that the outputs of most *experts* are disregarded and therefore do not need to be computed.

Google demonstrated that employing top-1 routing simplifies the routing process without compromising model quality [2]. This reduces the compu-

tational overhead of routing and simplifies its implementation while lowering communication costs.

Since the routing function is trainable, it sometimes results in an imbalance in the assignment of tokens to inputs, causing a small subset of *experts* to handle the majority of the tokens. To mitigate this, MoE models are trained with an auxiliary loss function that encourages the router to have a uniform distribution of tokens per experts. However, this loss normally harms model accuracy, and sometimes does not provide balanced routing [9, 11].

3.2 Tensor Parallelism / Sharding

Tensor Parallelism / Sharding (TS) refers to the technique of partitioning a tensor into multiple segments and distributing each segment across separate nodes in a distributed computing environment.

This approach is a form of Model Parallelism, in which the model's weights are divided into segments and assigned to different nodes. It is typically employed during model training or inference to increase the scalability of the system by utilizing multiple HAs.

Tensor parallelism is generally effective only when applied to HAs within the same node, as it needs high data transfer throughput and low latency to perform efficiently [12].

3.3 Data Parallelism

Data parallelism is a distributed model training approach in which distinct computing nodes are assigned specific partitions of the dataset.

This method is extensively employed in distributed deep learning and high-performance computing to enhance training and inference speed. Each node operates independently, performing forward and backward propagation on its assigned data subset.

Problem Setup

In this section I introduce the problem hindering MoE inference, specifying under which conditions this problem arises.

4.1 MoE experts placement

When training MoE models, which incorporate MoE layers, most system configurations involve multiple HAs, as they seldom fit entirely in one HA's memory.

As MoE layers often constitute the majority of the model's size (e.g., 96.776% in the encoder-decoder Switch Transformer with 128 *experts* [2]), a common approach is to replicate all non-MoE layers across the HAs while distributing the *experts* among them rather than replicating them [13]. This distribution is defined by a topology, which specifies the placement of *experts* across the HAs.

During computation, HAs use collective communication primitives to share input data. This ensures that inputs assigned to each *expert*, as determined by the router, are processed by the HA holding the corresponding *expert*. Subsequently, the processed outputs are redistributed to the appropriate HAs.

4.2 Imbalance

As mentioned, the router produces imbalanced decisions despite being trained with a loss function designed to prevent such behavior.

When paired with the aforementioned setup, this leads to **idle time for the HAs holding under-utilized experts**, as the HAs holding over-utilized *experts* process more tokens and therefore take more time. This prevents the full computational capacity of the infrastructure from being utilized, leading to longer training and inference times.

4.3 Assumptions

I assume that all of the HAs have the same computing power. Each HA is connected to all other nodes through high speed and throughput links, which is common in intra-node setups. Each HA is also connected to the host device, the CPU.

4.4 Problem

My objective is to reduce the waiting time by balancing the computational load across the HAs. Assuming ideal load balancing, where each HA possesses equal computational capacity, each HA will, on average, require the same amount of time to process the input, resulting in virtually no idle time. This leads to higher HA utilization, which in turn leads to smaller end to end latency and bigger throughput.

Additionally, most of other experiments [8, 12, 9] show that communication between HAs is the performance bottleneck for such systems. In my experiments, where a maximum of 4 HAs are considered and only inference is considered, the main bottleneck turned out to be the amount of launched kernels, which does not allow for an efficient use of the HAs since each kernel performs a small operation. Hence, the process of launching each of these kernels requires synchronization between the CPU and the HAs, turning it into a bottleneck.

Thus, the goal to achieve is optimal load balancing, while minimizing the overhead of launching *kernels*.

Algorithm

In this section, I introduce a basic version of the algorithm devised for achieving perfect load balancing.

Later, I delve into how to optimize the algorithm.

5.1 System

The model's layers are replicated across each HA, with the exception of the MoE layers. All *experts* are identical, with each *expert* consisting of two matrices. These matrices will henceforth be referred to as W_i and W_o, respectively. There are num_exp *experts* and num_HA HAs.

For the MoE layer, the router is replicated across all HAs. Each HA contains all *experts*, but only a slice of each *expert* (achieved through TS). Specifically, each HA holds a slice of W_i and a slice of W_o. No parameters from each *expert* are duplicated. Various methods for slicing these matrices and their respective implications will be discussed later. For now, let us assume that W_i is sliced along the columns and W_o is sliced along the rows.

Each HA processes a distinct batch of data (3.3).

Although this is the proposed configuration, the algorithm is easily adaptable to different *expert* configurations.

5.2 Procedures

PermutePerExpert This function takes a vector of inputs and partitions it into *num_exp* vectors, where each vector corresponds to the input for an individual *expert*.

GroupExpertsPerHA This function accepts a list of lists of vectors containing data (either *expert* input or output). The outer list, indexed by h, represents the data for all *experts* corresponding to HA h, and each element at index e of the inner list holds the data for *expert* e. Additionally, it accepts a vector that specifies the number of inputs per *expert* from each HA. The function groups the data into *num_HA* vectors, each containing the concatenated data for all *experts* for each HA.

AllGather **and** *ScatterReduce* These are standard communication primitives commonly found in distributed HAs systems. They are fully described in `https://pytorch.org/docs/stable/distributed.html#torch.distributed.all_gather`.

5.3 Algorithm

Algorithm 1 illustrates the procedure executed by each HA. The *hidden_states* vector will differ for each HA since they all begin with a distinct batch of data. We refer to the HA executing the code as HA P.

Firstly, the router runs on HA P's inputs to determine the assignment of tokens to *experts* (line 1).

Subsequently, the input is divided into subsets corresponding to each *expert*, and HAs communicate the number of inputs assigned to each *expert* via an *AllGather* operation (lines 4-7).

To minimize idle time by ensuring a single round of communication, HA P concatenates its inputs for all *experts* into one vector, which it then shares with the other HAs through an *AllGather*. Upon receiving the inputs from HA P,

Algorithm 1: Algorithm running in SIMD fashion on each HA

Input: Accepts the *hidden_states* from the layer before
Output: Returns the next *hidden_states* of the model

$r\_mask, r\_probs \leftarrow Router(hidden\_states)$

$Input\_per\_exp \leftarrow PermutePerExpert(hidden\_states, r\_mask)$
$Send\_sizes \leftarrow NumInputsPerExpert(Input\_per\_exp)$

#Recv_sizes shape: (num_HA, num_experts)
$Recv\_sizes \leftarrow AllGather(Send\_sizes)$

$Send \leftarrow Concatenate(Input\_per\_exp)$

$Recv \leftarrow AllGather(Send)$

$Workload \leftarrow PermutePerExpert(Recv, Recv\_sizes)$
for $h \in \{1, \ldots, num\_HA\}$ **do**
    for $e, exp \in experts$ **do**
        $Workload[h][e] \leftarrow exp(Workload[h][e])$

$Send \leftarrow GroupExpertsPerHA(Workload, Recv\_sizes)$
$Recv \leftarrow ScatterReduce(Send)$
$Recv \leftarrow PermutePerExpert(Recv, Send\_sizes)$

$hidden\_states \leftarrow hidden\_states \cdot r\_probs \cdot Recv$

each HA can split P's inputs into expert-specific subsets, using the input sizes received during the metadata *AllGather* operation.

Following the data *AllGather* communication, all HAs hold the same *num_HA* vectors, which contain the inputs from each HA (lines 9-11).

Next, HA P splits the inputs from all HAs into subsets corresponding to each expert, using the input sizes received during the metadata *AllGather* operation (line 13). After this operation, P holds a list of lists of tensors, where entry (h, e) contains the vector containing all inputs from HA h for *expert* e.

Each input is then processed by the corresponding *expert* (the slice held by P). For input $\mathbf{x}$, this corresponds to the operation $\mathbf{x}W_i^P W_o^P$ (lines 14-16).

After processing the inputs, P must return the outputs to the corresponding HA from which the input originated. Moreover, the outputs for the same tokens from different HAs must be summed point-wise to generate the final result, as depicted in Fig. 5.1c. P concatenates the outputs belonging to a single HA from all *experts* into one vector for efficient communication.

The HAs then perform a *ScatterReduce*. This communication primitive is more efficient because it enables P to send the outputs originating from inputs from HA $h \neq P$ exclusively to h, thereby minimizing the total amount of data transferred. Furthermore, it allows for the summing and scattering of vectors in a single operation, which is generally more efficient in most systems (lines 18-19).

The vector resultant from the *ScatterReduce* is the point-wise sum of the concatenated *expert* outputs from all HAs of P's inputs. Therefore, this vector has the same number of outputs per *expert* as line 4. P can then use this information to separate the outputs into subsets corresponding to each expert. Finally, P re-assigns the new values for *hidden_states* (line 22).

5.4 Load balancing

Each HA holds an equal number of *expert* parameters, provided that the sliced dimension of W_i and W_o is divisible by the number of HAs. Even when this condition is not met, a subset of the HAs holds only **one** additional row or column of W_i and/or W_o, which is often negligible given the total size of the sliced dimension. This occurs because each HA contains a slice of all *experts*, and these slices are distributed uniformly among the HAs.

Furthermore, each HA processes an equal amount of input, since all HAs process all inputs from all HAs.

5.5 How to slice W_i and W_o

Consider an input matrix $\mathbf{x}$ of shape (a, b), where each entry occupies 1 Byte. Let W_i have a shape of (b, c) and W_o a shape of (c, d).

The matrices W_i and W_o can be sliced either row-wise or column-wise, resulting in four possible slicing configurations: (1) both matrices are split row-wise, (2) both matrices are split column-wise, (3) W_i is split column-wise and W_o row-wise, and (4) W_i is split row-wise and W_o column-wise.

We first analyze the splitting of W_i. Fig. 5.1 illustrates how the inputs are processed and combined for a row-wise split (5.1a) and a column-wise split (5.1b). In a column-wise split, each HA processes $a \cdot b$ entries of $\mathbf{x}$. Conversely, in a row-wise split, each HA processes only $a \cdot b \div num\_HA$ entries of $\mathbf{x}$.

This distinction directly affects the volume of data transferred during the first data communication round. With a column-wise split of W_i, each HA must send all its input data to every other HA, resulting in a total data transfer of $a \cdot b \cdot (num\_HA - 1)$ bytes per HA. In contrast, a row-wise split requires each HA to send only $a \cdot b \cdot (num\_HA - 1) \div num\_HA$ bytes in total, as each HA transmits a unique segment of $\mathbf{x}$ to the others.

While a row-wise split of W_i may initially seem advantageous, considering

the interaction with W_o reveals a different outcome. A column-wise split of W_i allows both matrix multiplications to proceed without intermediate synchronization if W_o is split row-wise, as illustrated in Fig. 5.1c.

All other slicing combinations would necessitate synchronization between operations. For instance, if W_i is split row-wise, the outputs of $\mathbf{x}W_i^P$ must be summed point-wise across all HAs. Similarly, if W_o is split column-wise, the outputs of $\mathbf{x}W_i^P$ must be concatenated across HAs before the next multiplication can proceed.

The optimal slicing strategy is thus to split W_i column-wise and W_o row-wise. Assuming that $c \bmod num\_HA = 0$, then each HA, will hold $b \cdot c \div num\_HA$ entries from W_i and $c \div num\_HA \cdot d$ entries from W_o.

5.6 Uneven splits across HAs

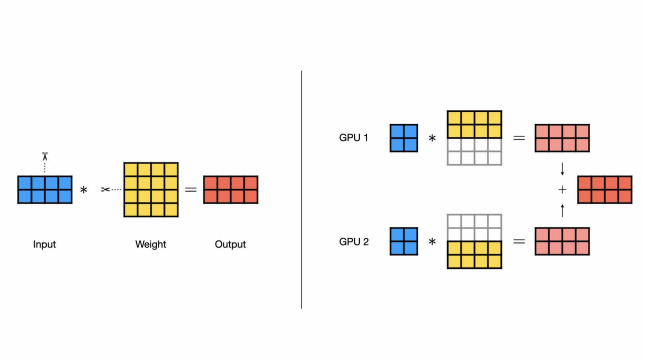
The algorithm accommodates scenarios in which at least one of the sliced dimensions is not evenly divisible by the number of HAs. Assuming the slicing strategy proposed in Section 5.5, all operations, specially concatenation, are defined even if the shapes of W_i and W_o are not the same for every HA, as all inputs are assumed to have the same number of features in all but the first dimension.

5.7 Communication

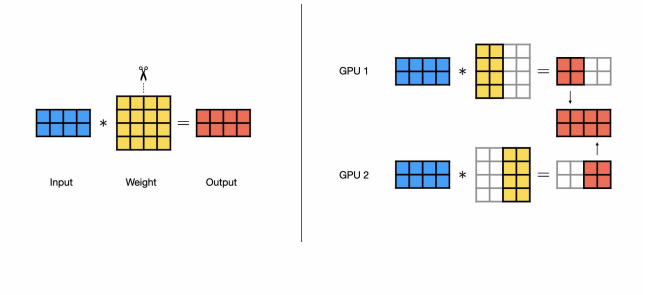
The algorithm is implemented with a minimal amount of communication.

During the initial data *AllGather* operation, each HA must share all of its input data since every input needs to be processed by all other HAs. In the final data *ScatterReduce* operation, P only transmits the output corresponding to the result of processing an *expert* on a specific input from h to h and no other HA.

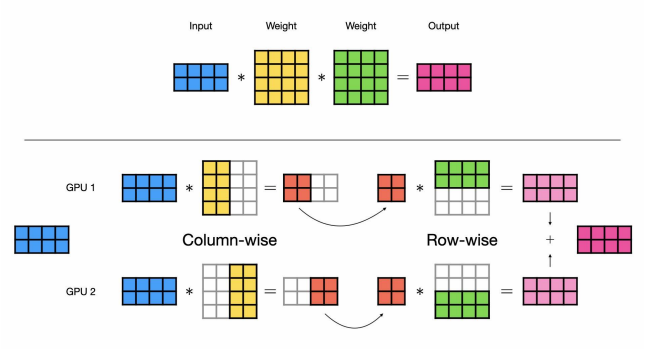
The only significant communication overhead arises from the transfer of



(a) Row-wise



(b) Column-wise



(c) Using a column-wise split on W_i and a row-wise split for W_o allows to perform both operations without synchronization in between them

Figure 5.1: Distributed processing using tensor sharding. Figures from `https://lightning.ai/lightning-ai/studios/tensor-parallelism-supercharging-large-model-training-with-pytorch-lightning?section=featured`.

metadata, which constitutes a relatively small amount of data. Typically, the number of *experts* does not exceed a few thousand . One possible solution to reduce this overhead is padding the inputs for each *expert* to the maximum number of input samples per *expert* (dropping samples is not considered, as the solution needs to be *dropless*). However, this approach was found to be more costly than transmitting the metadata.

It is important to note that each HA might not always handle the same amount of data.

Metadata *AllGather* In this round, each HA transmits a vector containing *num_exp* integers. Consequently, all HAs share vectors of identical size.

Data *AllGather* In this round, HA P sends its input data to every other HA. However, some HAs may have more inputs than others due to: (1) variations in batch sizes across HAs and (2) the router not assigning an *expert* to every sample, resulting in some samples being dropped and not included in these vectors. This can lead to imbalances in vector lengths even if all HAs have identical batch sizes.

Data *ScatterReduce* During this round, each HA transmits *num_HA* vectors. Vector h contains the outputs corresponding to the inputs from HA h for all *experts*, concatenated together. Although these vectors may differ in length from one another for the reasons mentioned above, all HAs transmit *num_HA* vectors with the same size, only with different values resulting from the computation of their slice of the *expert* on that HA's inputs.

Additionally, the two data communication rounds are independent of the number of *experts*, enabling scalability as the number of *experts* increases. Although the metadata communication round depends on the number of *experts*, its small size makes it negligible in terms of overall communication overhead.

Finally, all communication is influenced by the number of participating HAs.

5.8 Optimizing *expert* inference

One significant limitation of this approach is that the algorithm now executes numerous small matrix multiplications, as each *expert* slice is processed independently. In real-world systems, this results in substantial overhead due to frequent kernel launches, which poses scalability challenges as the number of *experts* and HAs increases.

To address this issue, it becomes necessary to fuse some of these matrix multiplications to reduce the number of kernel launches.

A straightforward solution involves concatenating the inputs from different HAs corresponding to the same *expert* into a single vector. This allows the *expert* to process all inputs simultaneously, followed by a post-processing step to separate the outputs back into subsets that correspond to the outputs belonging to each HA. This approach makes the number of kernel launches for matrix multiplications independent of the number of HAs.

A more advanced optimization leverages variable-sized sparse matrix multiplication, enabling the processing of all *expert* outputs in a single step using a large sparse matrix multiplication algorithm, as detailed in [6]. This approach makes the number of kernel launches independent of the number of *experts*. This variation is further explored in Section 6.

Evaluation

In this section, I present a series of experiments designed to evaluate the proposed algorithm and its variation under various conditions.

Two model architectures are considered: an encoder-only model and an encoder-decoder model. The model is configured with 8, 16, 32, 64, 128, or 256 experts. The model is tested with differing numbers of *experts*, different input sizes, different router skew values and different number of skewed *experts*.

The baseline for comparison is the same model, using DeepSpeed's [12] inference engine and DeepSpeed's custom MoE layer.

Two implementations are evaluated: the basic algorithm, referred to as **Sliced**, and a variation leveraging MegaBlocks [6] to process all slices of *experts* at once within each HA while also employing TS. This variation is referred to as **MegaBlocks**.

The performance of these two implementations is compared against each other and against DeepSpeed.

6.1 Setting

DeepSpeed is configured to use a combination of **expert parallelism**, wherein different *experts* are assigned to distinct HAs, and TS, consistent with my implementation. In both DeepSpeed and my implementation, the remainder of the model is replicated across HAs. As this layer inherently drops tokens

due to the use of a capacity factor, this parameter is set to `min(num_exp, 50.0)`, ensuring minimal token loss while adhering to memory constraints. My implementation is *dropless*, meaning that no capacity factor is applied.

A custom router was implemented to regulate the degree of skew in token-to-expert assignments. All experiments were conducted using this router, unless stated otherwise. Additionally, this router allows a truly *dropless* implementation, as the Switch Transformer employs a capacity factor. However, this router is not suitable for real-world scenarios, as it operates entirely probabilistically and lacks a learning mechanism. It has two parameters: the degree of skew in the assignments (*router_skew*) and the number of *experts* receiving a disproportionately larger share of tokens (*num_experts_skew*).

The benchmarks were conducted using 4 HAs. The impact of varying the number of HAs was not tested, as, for HAs ranging from 1 to 8, only configurations with 1, 2, and 4 HAs are compatible with both DeepSpeed and my implementation. This is a soft requirement, as modifying certain matrix multiplication kernel constants (e.g. tile size) from [6] would allow for compatibility with other numbers of HAs.

All HAs are located within the same computing node and share access to a single CPU, specifically an AMD EPYC 7543 32-Core Processor operating at a maximum clock speed of 3.7 GHz.

In this configuration, all Hardware Accelerators (HAs) are NVIDIA A100-SXM4-80GB GPUs utilizing CUDA Version 12.6. Each GPU is equipped with 80 GB of random access memory. The GPUs are interconnected via NVLink technology [14] and are linked to the CPU through PCIe [14] bridges.

6.1.1 Model

For my experiments, I utilized the Google Switch Transformers [2]. The base models are available on HuggingFace at `https://huggingface.co/docs/transformers/model_doc/switch_transformers`. The code for my algorithm is implemented in a separate module, and functions to insert this module in the base model, as

well as copying the original experts' weights, are provided. The code used to run my experiments is available at `https://github.com/pleaky2410/semester-project-moe`.

6.1.2 Datasets

As a custom router that works solely based on probabilities was used, the chosen dataset should not play a major role in the end results. For the purposes of my experiments, I used 1 dataset: *bookcorpus*.

6.2 End-to-End Inference Time (E2EIT)

The objective of this experiment is to analyze the variation in E2EIT across different implementations and iterations. The experiment was conducted using 128 experts, 4 HAs, 100 iterations, a batch size of 250, and a sequence length of 120. The dataset was *bookcorpus*. This experiment is the only one that does not utilize the custom router; instead, it employs the original router to provide an example of a real execution scenario. The capacity factor was set to 50.0 for all runs.

Figure 6.1 illustrates the variation in E2EIT across different implementations and iterations. The full model E2EIT was not used because DeepSpeed optimizes the entire model, whereas my implementation focuses solely on MoE layers. A total of 4 layers are represented: the first and last encoder and decoder layers of the encoder-decoder version of Switch.

It is evident that for encoder layers, the MegaBlocks implementation demonstrates superior performance. For decoder layers, however, this difference is less obvious, as both implementations appear to perform equally well. Additionally, both of my implementations outperform the baseline DeepSpeed implementation for encoder layers but underperform relative to it for decoder layers.

This is expected, given that DeepSpeed is primarily designed for decoder-only architectures and auto-regressive tasks, making it more optimized for decoder layers. Additionally, the performance gains of DeepSpeed from using 1

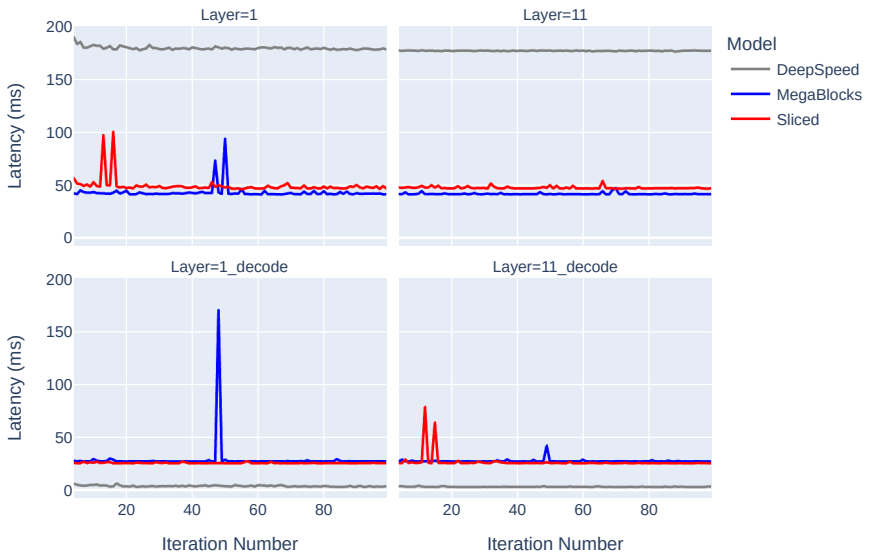


Figure 6.1: E2EIT of the first and last encoder and decoder MoE layer

fused kernel for most operations are more noticeable on decoder layers, where the input size is much smaller. Furthermore, the latency spikes observed in DeepSpeed are significantly smaller - almost negligible - compared to those in my implementation. This is likely due to the use of custom fused kernels for both computation and communication. Moreover, this observation suggests that DeepSpeed may employ tensor padding alongside with the capacity factor, ensuring that tensor sizes remain relatively static across iterations, regardless of the actual token-to-expert assignments.

Finally, it is also evident that the first layer - both for the encoder and decoder - induces greater latency spikes compared to the last layer.

6.3 Average Speedup

I executed experiments using DeepSpeed, as well as the Sliced and MegaBlocks implementations of the model, while systematically varying one system parameter at a time. For each value of the parameter, I computed the average speedup

in E2EIT across all layers and iterations for the Sliced and MegaBlocks implementations relative to the DeepSpeed implementation. This analysis was performed separately for both the encoder-only and encoder-decoder variants of the model, ensuring that the encoder versions of Sliced and MegaBlocks were compared to the encoder variant of DeepSpeed with the same value for the parameter, and similarly for the encoder-decoder variants.

6.3.1 Number of experts

The experiment was conducted using the *bookcorpus* dataset, with a batch size of 250 and a sequence length of 120. 4 HAs were used. The parameter *num_experts_skew* was set to 10% of the total number of *experts*, while *router_skew* was configured to 0.6. Each implementation was executed for 100 iterations.

As illustrated in Fig. 6.2, both implementations initially exhibit an increase in relative speedup, reaching a maximum of 6.84 times faster than DeepSpeed, followed by a decline. Nevertheless, even in one of the least favorable scenarios, the encoder-decoder MegaBlocks implementation remains 26% faster than DeepSpeed with 256 experts, while the encoder-only implementation is 194% faster under the same conditions.

However, drawing definitive conclusions from this plot is challenging due to the capacity factor settings in DeepSpeed, which is defined as `min(num_exp, 50)`. Consequently, after exceeding 32 experts, this factor remains constant. The capacity factor must be within the range [0, *num_exp*] to ensure all operations are well-defined. Setting the capacity factor to 8 for all *expert* configurations would not be a fair comparison, as it would result in a significant number of dropped tokens. This occurs because the **expert capacity**, which determines the number of tokens each *expert* can process, is defined as:

$$expert\_capacity = capacity\_factor \times \frac{batch\_size}{num\_exp} \tag{6.1}$$

Setting the capacity factor to *num_exp* allows each *expert* to process the entire batch, which occurs when $router\_skew = 1$ and $num\_experts\_skew = 1$, leading to a *dropless* solution. However, due to memory constraints, it is not always feasible to set the capacity factor equal to the number of experts.

This further demonstrates that the proposed solution has a smaller memory footprint while always being *dropless*. Consequently, when both solutions are *dropless*, the proposed implementation performs better than DeepSpeed. Conversely, when DeepSpeed ceases to be *dropless*, the performance of the proposed solution degrades as the number of *experts* increases, since some tokens may be dropped in DeepSpeed.

A particularly interesting observation is that the Sliced version outperforms MegaBlocks up to 32 experts, beyond which MegaBlocks surpasses it. This behavior is attributed to the computational overhead associated with MegaBlocks, which outweighs the benefits of fused matrix multiplications when the number of *experts* is low. However, as the number of *experts* increases, MegaBlocks effectively mitigates the issue of excessive kernel launches and fully exploits the benefits of TS, ultimately proving to be a superior solution compared to the Sliced version.

Additionally, notable differences arise between encoder-only and encoder-decoder models. Encoder-only models derive greater benefits from TS than encoder-decoder models, as they comprise more batched operations. In contrast, the structure of decoder modules diminishes the overall performance gains from TS. Furthermore, DeepSpeed is specifically optimized for NLG tasks [12] and decoder-only architectures, leading to superior performance in decoder layers while exhibiting lower efficiency in encoder layers overall.

6.3.2 Input size

For this experiment, dataset, number of HAs, sequence length, *router_skew*, *num_experts_skew* and number of iterations were the same as in Section 6.3.1. 128 *experts* were used, and hence the capacity factor used was 50.0.

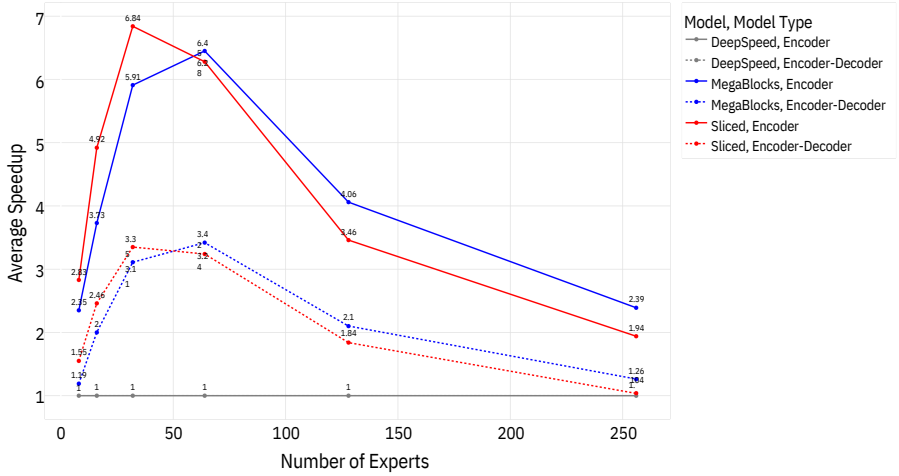


Figure 6.2: Average Speedup compared to DeepSpeed for 8, 16, 32, 64, 128 and 256 Experts

In Fig. 6.3, the average speedup of all MoE layers relative to DeepSpeed is compared between the MegaBlocks and Sliced implementations across across different batch sizes.

The trend is evident: performance increases as the batch size increases. This is expected, as with a bigger batch size, although the percentage of *router_skew* is the same, the difference in number of tokens is bigger between different experts, causing a bigger total idle time for non balanced implementations. If padding is used in DeepSpeed, the tensor sizes will be bigger for a bigger input size, making my solution more efficient, as it leverages dynamic tensor sizes according to router assignments.

For a batch size smaller than 100, my solution is worse than DeepSpeed, even for encoder models. This is also somewhat expected, given that, as mentioned in Section 6.3.1, DeepSpeed is tailored for decoder-only architectures, which deal with smaller inputs due to its auto-regressive nature. A smaller batch size resembles the behavior of these layers.

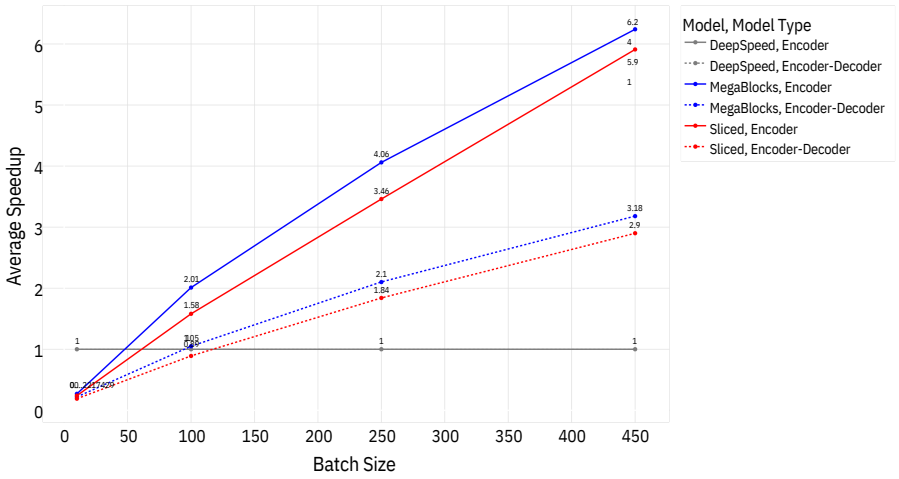


Figure 6.3: Average Speedup compared to DeepSpeed with 128 *experts* for different batch sizes

6.3.3 Router Skew

In Fig. 6.4, the average speedup of all MoE layers relative to DeepSpeed is compared between the MegaBlocks and Sliced implementations across different values of router skew.

This result is somewhat unexpected. Intuitively, as the *router_skew* increases, the speedup of the proposed implementations would be expected to improve, given that theoretical load balancing across HAs should be fully exploited in highly skewed scenarios. However, the observed relationship is approximately linear.

Several factors may help explain this result. First, DeepSpeed also employs TS for experts, which contributes to load balancing, although this is not its primary objective, as indicated in [12]. Expert parallelism itself can enforce a certain degree of load balancing, depending on how *experts* are distributed across HAs. Second, given that the capacity factor is 50.0 and the number of *experts* is 128, the actual token-to-expert assignments in DeepSpeed are not as skewed as in the proposed implementation. Each *expert* in DeepSpeed can process at most $50 \times \frac{250 \times 120}{128} \approx 11718$ tokens, whereas a single *expert* in the

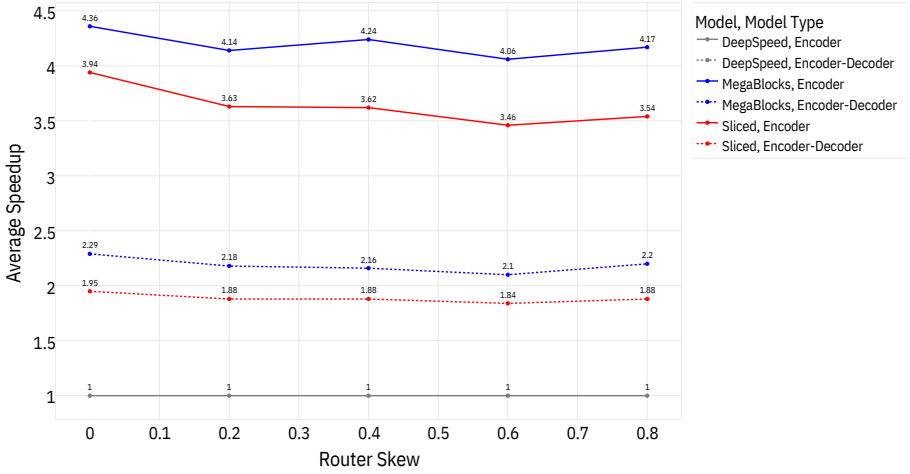


Figure 6.4: Average Speedup compared to DeepSpeed with 128 *experts* for different router skew values

proposed implementation could be assigned up to 24,000 tokens if it receives 80% of the total tokens.

Nevertheless, the primary factor likely relates to DeepSpeed's internal implementation. While speculative, it is plausible that DeepSpeed employs a static tensor size for the inputs to each expert. If an *expert* is assigned more tokens than its capacity, the excess tokens are discarded. Conversely, if an *expert* receives fewer tokens than its capacity, the input tensor may be padded to match the maximum size [2]. This would imply that DeepSpeed's execution time remains constant regardless of token assignment, corresponding to the worst-case scenario for all experts. This hypothesis, paired with the fact that the proposed solution achieves theoretical load balancing regardless of token distribution and thus exhibits a constant execution time, explain this result.

6.3.4 Number of skewed experts

For this experiment, the number of experts, dataset, number of HAs, sequence length, batch size and number of iterations were identical to 6.3.3. The *router_skew* was 0.6, and the capacity factor was set to 90, to test if it is the cause of the

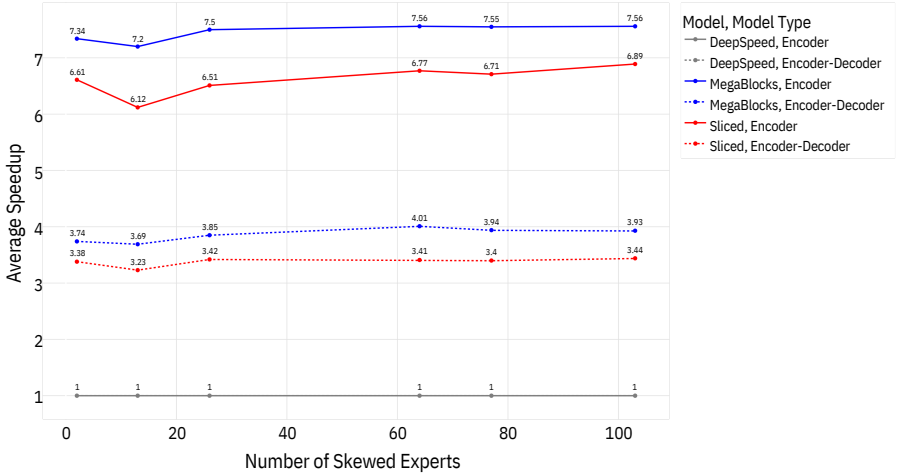


Figure 6.5: Average Speedup compared to DeepSpeed with 128 *experts* for different numbers of skewed experts

results obtained in Section 6.3.3.

In Fig. 6.4, the average speedup of all MoE layers relative to DeepSpeed is compared between the MegaBlocks and Sliced implementations across different values of number of skewed experts.

The relationship is the same as in Section 6.3.3. I would expect my proposed solutions to be increasingly better as the number of skewed *experts* diminishes. The possible reasons for this result are the same as in Section 6.3.3.

6.4 Hardware Accelerator (HA) idle time

In this experiment, the number of experts, dataset, number of HAs, sequence length, batch size, *num_experts_skew*, and number of iterations were identical to those in Section 6.3.3. *router_skew* was set to 0.6. An encoder-decoder version was used.

Figure 6.6 was obtained by computing the mean and standard deviation (std) of the latency of different code sections in Alg. 1 across all HAs for the same layer and iteration. Subsequently, both the mean and standard deviation values were averaged across iterations using the appropriate formula for averaging standard

deviations. Consequently, larger error bars in the bar plot indicate greater variations in execution time for a given code section across different HAs, leading to increased idle time for some HAs.

It is evident that, although the proposed algorithm theoretically ensures load balancing, this property does not fully hold in practice, particularly for decoder layers. This discrepancy is entirely explained by the fact that real hardware and software do not execute an identical number of computations in precisely the same amount of time. For instance, in the section labeled "2nd Data," which corresponds to the *ReduceScatter* operation, all HAs transmit the same volume of data; however, the transmission duration varies across different HAs.

Furthermore, the process of launching kernels needs CPU-HA synchronization, the duration of which depends on numerous factors. Since the model involves conditional control flows, such as if and for statements, the timing of kernel launches is also influenced by the Python interpreter, which may introduce variations in execution time across different processes. Lastly, measurement inaccuracies may also contribute to these discrepancies.

Decoder layers appear to be more affected by these issues, as they process fewer tokens and consequently exhibit a lower E2EIT. As a result, these overheads become more noticeable.

These findings show that, while the proposed algorithm theoretically ensures perfect load balancing, this balance is not always held in practice.

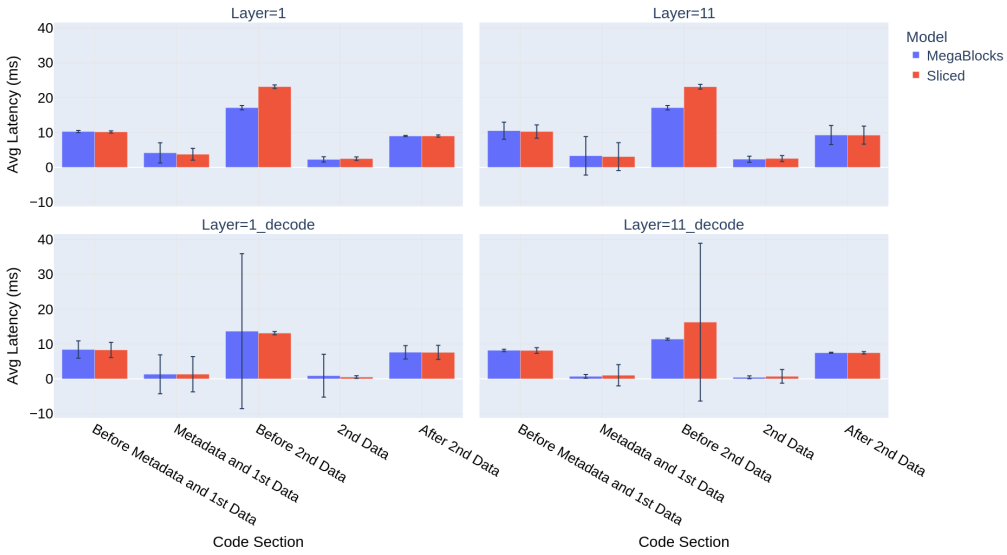


Figure 6.6: Average latency and std of the first and last encoder and decoder layers

Related work

Various authors have been interested in exploring MoE architecture for different applications and how to design systems optimized for MoE training and inference.

MoE models. Shazeer et al. [5] set the foundation for MoE layers and their architecture. Among many other authors, Fedus et al. [2] explored applying MoE layers to the Transformer architecture, proving that top-1 routing produces quality results while keeping computation simpler and more efficient. Chen et al. [15] aims to implement and train different routing functions that are aware of the topology of the system, improving the system performance while running the model. However, this requires to re-train models for different topologies. Thus, it is not fit for models that may run in different systems. Lepikhin et al. [13] presents a top-2 gating function to balance *expert* utilization, ensuring tokens are distributed efficiently amog experts. Aminabadi et al. [12] aims at improving parameter efficiency and developed a Pyramid Residual MoE (PR-MoE), which is an hybrid between a dense and sparse model.

Training systems for MoE models. Lepikhin et al. [13] sets the foundations of how to use *expert* parallelism to train MoE models. However, their solution only works with Google's own HA: TPUs. He et al. [8] delivers a complete API for MoE training implemented in `Pytorch`. It provides optimizations

for MoE models such as running all the inputs for each *expert* at once, implementing specialized *kernels* for grouping / ungrouping inputs per expert. He et al. [9] improves this solution by creating estimations of the duration of each operation, used to choose the best parallelism strategy. Moreover, the authors present shadowing models for improving load-balancing between HAs. Lastly, they improve communication efficiency. Hwang et al. [11] proposes adaptive parallelism that efficiently handles the dynamic training workload of MoE.

Inference systems for MoE models Aminabadi et al. [12] provides the first highly scalable and optimized engine for MoE inference (and also for training). It employs Data Parallelism, Tensor Sharding and many custom *kernels* for their own MoE layer. Their view of Tensor Sharding is not so focused on load-balancing, as each *expert* may be sharded across a subset of all HAs. Their implementation works for both single and multi node setups. However, their implementation always needs a capacity factor to be set, which might result in tokens being dropped. Finally, their implementation is primarily focused on auto-regressive models, while mine works best for encoder or encoder-decoder models. Yao et al. [16] focus on inter-layer optimizations more than intra-layer. They exploit the concept of *expert* affinity, based on the observation that tokens are more likely to be routed to the same *experts* in different MoE layers. Additionally, they propose a novel *expert* parallelism optimization based on context coherence and *expert* affinity, which allows to reduce the number of communication rounds for some layers to 1.

Conclusion

This semester project report proposes an algorithm for optimizing End-to-End Inference Time for MoE models by applying Tensor Sharding, which aims to achieve perfect load balancing between HAs. When coupled with Data Parallelism, it achieves high HA utilization rates.

My experiments showed that TS applied to *experts* with the intent of providing perfect load balancing achieves smaller latency than DeepSpeed. It is tailored for datasets where the skewness of token assignment is high, allowing to keep a truly *dropless* solution without the overhead of setting a high capacity factor. Furthermore, it achieves almost perfect load balancing, which contributes to bigger HA utilization.

My main contribution is an algorithm that merges different already existing approaches with the clear goal of achieving theoretical perfect load balancing and minimizing the number of *kernel* launches to obtain a better solution for inference setups with a relatively small number of HAs.

However, more improvements could be made. The current algorithm does not support heterogeneous *expert* architectures. Additionally, if *experts* are made up of more than 2 matrices, and no further optimizations are employed, there needs to be synchronization in between the multiplication operations. Furthermore, my solution is not performant for small batch sizes or decoder-only models, attaining the best performance for encoder-only models.

Bibliography

[1] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," 2022. [Online]. Available: https://arxiv.org/abs/2206.07682

[2] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," 2022. [Online]. Available: https://arxiv.org/abs/2101.03961

[3] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," 2022. [Online]. Available: https://arxiv.org/abs/2204.02311

[4] M. Rillig, M. Ågerstrand, M. Bi, K. Gould, and U. Sauerland, "Risks and benefits of large language models for the environment," *Environmental science technology*, vol. 57, 02 2023.

[5] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," 2017. [Online]. Available: https://arxiv.org/abs/1701.06538

[6] T. Gale, D. Narayanan, C. Young, and M. Zaharia, "Megablocks: Efficient sparse training with mixture-of-experts," 2022. [Online]. Available: https://arxiv.org/abs/2211.15841

[7] O. Sanseviero, L. Tunstall, P. Schmid, S. Mangrulkar, Y. Belkada, and P. Cuenca, "Mixture of experts explained," 2023. [Online]. Available: https://huggingface.co/blog/moe

[8] J. He, J. Qiu, A. Zeng, Z. Yang, J. Zhai, and J. Tang, "Fastmoe: A fast mixture-of-expert training system," 2021. [Online]. Available: https://arxiv.org/abs/2103.13262

[9] J. He, J. Zhai, T. Antunes, H. Wang, F. Luo, S. Shi, and Q. Li, "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*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 120–134. [Online]. Available: https://doi.org/10.1145/3503221.3508418

[10] M. Zhai, J. He, Z. Ma, Z. Zong, R. Zhang, and J. Zhai, "SmartMoE: Efficiently training Sparsely-Activated models through combining offline and online parallelization," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 961–975. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/zhai

[11] C. Hwang, W. Cui, Y. Xiong, Z. Yang, Z. Liu, H. Hu, Z. Wang, R. Salas, J. Jose, P. Ram, J. Chau, P. Cheng, F. Yang, M. Yang, and Y. Xiong, "Tutel: Adaptive mixture-of-experts at scale," 2023. [Online]. Available: https://arxiv.org/abs/2206.03382

[12] R. Y. Aminabadi, S. Rajbhandari, M. Zhang, A. A. Awan, C. Li, D. Li, E. Zheng, J. Rasley, S. Smith, O. Ruwase, and Y. He, "Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale," 2022. [Online]. Available: https://arxiv.org/abs/2207.00032

[13] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," 2020. [Online]. Available: https://arxiv.org/abs/2006.16668

[14] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, p. 94–110, Jan. 2020. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2019.2928289

[15] C. Chen, M. Li, Z. Wu, D. Yu, and C. Yang, "Ta-moe: Topology-aware large scale mixture-of-expert training," 2023. [Online]. Available: https://arxiv.org/abs/2302.09915

[16] J. Yao, Q. Anthony, A. Shafi, H. Subramoni, D. K., and Panda, "Exploiting inter-layer expert affinity for accelerating mixture-of-experts model inference," 2024. [Online]. Available: https://arxiv.org/abs/2401.08383