Faster Distributed LLM Inference with Dynamic Partitioning
(CS 262A)

Isaac Ong with Woosuk Kwon
University of California, Berkeley

Abstract

In light of the rapidly increasing size of large language models (LLMs), this work addresses the challenge of serving these LLMs efficiently given the limitations of modern GPU memory. We observe that the inference of LLMs is unique as compared to other models due to the wide variation in input lengths, a factor not adequately addressed by existing works. Current inference engines typically employ a static partitioning strategy, which is sub-optimal given the variability in input lengths and the diversity of GPU specifications. To overcome these challenges, we propose a dynamic partitioning strategy for distributed LLM inference which dynamically switches between different partitioning strategies at inference time, optimizing for both GPU characteristics and input length. We systematically search for all Pareto optimal partitioning strategies for distributed LLM inference, focusing on their computational requirements, communication overhead, and memory demands. Based on this search, we identify three Pareto optimal strategies that cater to different scenarios and implement an inference engine for dynamic partitioning. Our evaluation, conducted on NVIDIA L4 and A100 GPUs using the Llama 2 family of models, demonstrates significant improvements over existing approaches. We illustrate reductions in the time to the first token of up to 40% and reductions in latency of up to 18%, underlining the effectiveness of dynamic partitioning. Our findings pave the way for more efficient utilization of GPU resources in distributed LLM inference, accommodating the evolving landscape of model sizes and architectures.

1 Introduction

Recent advances in machine learning have enabled an exponential increase in model size of large language models (LLMs), from BERT [1] with 340 million parameters to GPT-2 [2] with 1.5 billion parameters, and GPT-3 [3] with 175 billion parameters. The emergence of these large language models have led to myriad new use cases such as chatbots like ChatGPT [4] and coding assistants like GitHub Copilot [5].

However, the size of these large language models poses challenges for serving them as they exceed the memory limits of modern processors. For instance, with 175 billion parameters, the weights for GPT-3 [3] require over 300GB of GPU memory to store, while the latest NVIDIA H100 GPUs only contain 80GB of memory, meaning that at least four of these GPUs are required to serve GPT-3. Moreover, this only accounts for the model weights and not the additional memory required the store the model activations and inference code. This rapid increase in model size shows no sign of stopping [6]. Therefore, model inference must be parallelized across multiple GPUs to be feasible.

Model parallelism techniques can be mainly classified into two main types: pipeline parallelism and tensor parallelism. Tensor parallelism, which is the focus of this work, partitions tensor operations across multiple GPUs so as to increase the speed of computation or reduce the amount of memory used on each GPU. Such techniques have been well-studied in the literature. Currently, LLM inference engines such as vLLM [7], HuggingFace's TGI [8], and NVIDIA's TensorRT-LLM [9] make use of the approach introduced by Megatron-LM [10], which describes a specific model partitioning strategy to distribute tensor computation across GPUs.

Our key observation is that inference with LLMs is uniquely different from other machine learning models because of the wide variation in input lengths, a difference that is not covered by existing work. We note this is the case specifically for inference and not training because of the wide-ranging applications for LLMs, from chatbot conversations to use cases like retrieval-augmented generation [11], where documents containing tens of thousands of tokens are fed into a LLM for summarization and information retrieval. To this end, there has also been a trend of increasing context length supported by LLMs, from 1024 supported by GPT-2 [2] to over 100,000 tokens supported by Claude [12], a trend which is likely to continue into the future.

We note that partitioning strategies can differ greatly in terms of FLOPs, communication overhead and memory requirements, all of which vary based on the input length. Therefore, the input length should be considered to determine an optimal partitioning strategy. At the same time, the perfor-

mance of different partitioning strategies can also vary a lot based on the type of GPU used, an issue which is exacerbated by the heterogeneity in GPUs today. Currently, major companies such as NVIDIA, Google, and AMD offer GPUs that have a broad range of specifications. For example, the memory bandwidth for NVIDIA GPUs can range from 200GB/s with A2 Tensor Core GPUs to 2TB/s with H100 Tensor Core GPUs. On the other hand, the L4 GPU achieves 36 TFLOPs on half-precision floating point numbers while the H100 GPU achieves 1512 TFLOPs, a difference of over 40 times. These wide disparities in GPU characteristics have to be considered when deciding the optimal partitioning strategy for LLM inference.

Existing works in LLM inference do not account for this and apply a static partitioning scheme for all input lengths and models. Therefore, in this work, we propose using a dynamic partitioning strategy for distributed LLM inference that switches between partitioning strategies at inference time based on the model, GPU characteristics and input length with the goal of minimizing the time to first token and latency. We conduct an exhaustive search over all partitioning strategies for distributed LLM inference considering their performance with respect to FLOPs, communication overhead, and memory requirements. We then identify three Pareto optimal partitioning strategies for LLM inference that perform most efficiently in different scenarios. Based on these partitioning strategies, we develop a inference engine that is capable of dynamically switching between these partitioning strategies at inference time.

We evaluate dynamic partitioning on both L4 and A100 NVIDIA GPUs using the Llama 2 7B, 13B, and 70B models [13]. Our evaluation results show that as compared to using the static partitioning strategy from Megatron-LM [10], using a dynamic partitioning strategy achieves a reduction of up to 40% in the time to first token and a reduction of up to 18% in overall latency.

To summarize, we make the following contributions:

- We formalize the problem of identifying alternative partitioning strategies for distributed LLM inference while ensuring the correctness of these strategies by defining tensor states and operations on these tensors.
- We conduct an exhaustive search over all feasible partitioning strategies for distributed LLM inference and identify the Pareto frontier of partitioning strategies based on weight FLOPs, communication volume, and weights memory.
- We implement an LLM inference library that implements dynamic partitioning, switching between different partitioning schemes at inference time based on the GPU, model architecture, and input length. We show that dynamic partitioning achieves superior performance in terms of overall latency and time to first token as compared to the existing state-of-the-art approach to tensor parallelism.

2 Background

This section explains the existing Transformer architecture used in LLMs today, as well as techniques used to parallelize these models.

2.1 Transformer Architecture

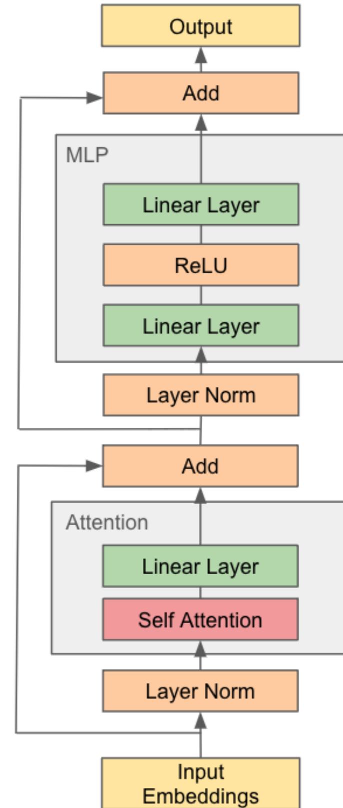

Figure 1: The architecture for a single Transformer layer

LLMs are based on the auto-regressive Transformer architecture [14], show in a simplified manner in Figure 1. A single Transformer layer consists of a self-attention block, followed by a two-layer multi-layer perceptron (MLP), as shown in the figure. These layers are replicated multiple times to form the full Transformer model. Transformer models are auto-regressive, meaning that they generate new tokens one at a time based on the *input prompt* tokens and the previously-generated *output* tokens. Specifically, the process of inference in Transformer models can be broken down into two phases.

Prefill phase The prefill phase takes the entire user prompt and computes the first *output* token. As part of this process, the Transformer also generates the key and values vectors for all the prompt tokens, which are stored for future use. Therefore, for this phase, the size of the input is the length of the entire prompt.

Generation phase The auto-regressive generation phase generates the remaining tokens one at a time. Specifically, at each iteration, the Transformer model takes in the last

generated *output* token and computes the next token in the sequence using all the previously-generated key and value vectors. This process of generation continues until either the sequence reaches a maximum length (as specified by the user or the LLM) or when an end-of-sequence token is returned. For this phase, the size of the input is always one at each iteration, since only the last *output* token generated is used.

2.2 Model Parallelism

In general, there are two main approaches to model parallelism: pipeline parallelism and tensor parallelism.

Pipeline parallelism In pipeline model parallelism, the layers of the model are split between different GPUs. Each GPU performs operations for its assigned portion of the model before the outputs of these operations are passed onto the next GPU, where a new set of operations are performed, just as in a pipeline.

Tensor parallelism Tensor parallelism is an orthogonal approach to pipeline parallelism whereby tensor computation is partitioned across GPUs to either speed up computation or the reduce the memory usage on each GPU.

2.2.1 Transformer Model Parallelism

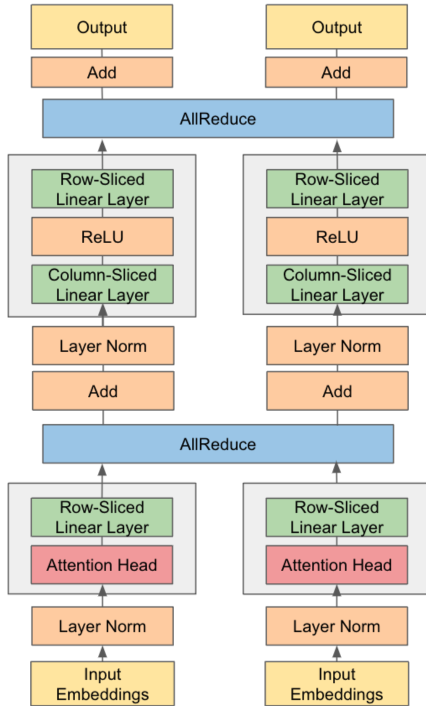

Figure 2: The partitioning strategy used by Megatron-LM for 2 GPUs

For Transformer models specifically, Megatron-LM [10] introduced a model parallelism strategy for both the self-attention blocks and MLP blocks, as shown in Figure 2.

Self-attention Block For the self-attention block, Megatron-LM partitions the multi-headed attention operation in a column-parallel manner such that the matrix operations corresponding to each attention head are done locally on each GPU, allowing the attention operation to be parallelized across GPUs. Next, the output linear layer is partitioned in a row-parallel manner such that it takes the output of the attention operation directly. The resulting tensor after the linear layer is the same size as the full tensor, but only contains partial values for each element. Therefore, an all-reduce operation is performed to synchronize the tensors so that the full tensor is now present on each GPU.

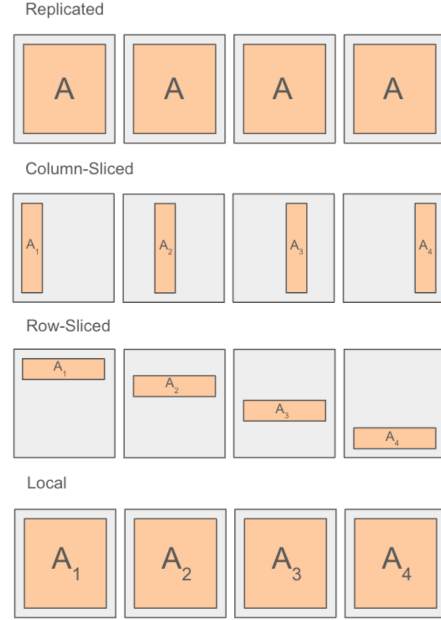

Figure 3: The valid states for a given tensor A when distributed across GPUs

MLP Block For the MLP block, the first linear layer is partitioned in a column-parallel manner while the second linear layer is partitioned in a row-parallel manner, allowing it to take the output of the first linear layer directly without any synchronization. Finally, the resulting tensor again only contains partial values for each element, so an all-reduce operation is required after the MLP block to obtain the full tensor on each GPU.

By partitioning the computation across multiple GPUs, the Megatron-LM [10] partitioning strategy reduces the FLOPs required on each GPU at the expense of increased communication volume, which comes from the all-reduce operations performed after the self-attention block and the MLP block.

3 Formulation

We formalize the problem of identifying valid partitioning strategy given a model architecture by creating rules that specify the state of tensors when distributed across GPUs and the set of operations that can be performed on them.

Tensor States We extend the tensor layout described in CoCoNet [15] such that a given tensor A can classified into one of

four states when used for distributed computation: *replicated*, *column-sliced*, *row-sliced* or *local*, as shown in Figure 3:

- A *replicated* tensor is one which has the same value on all devices.
- A *column-sliced* tensor (denoted by A_{CS}) or *row-sliced* tensor (denoted by A_{RS}) is partitioned equally across all devices in a column-parallel or row-parallel manner respectively.
- A *local* tensor (denoted by A_L) is one that has the same shape on all devices, but contains a different value on each device.

Matrix Operations Based on the above tensor states, the valid matrix operations given two tensors A and B are as follows (with multiplication denoted using @):

- $A_{CS}@B_{RS} = AB_L$
- $A_R@B_{CS} = AB_{CS}$
- $A_{RS}@B_R = AB_{RS}$
- $A_R@B_R = AB_R$

Non-linearities On each GPU, non-linearities such as the rectified linear activation function (or ReLU) can be only be executed on tensors that are not *local* since they require each element in the tensor to be the same value as in the full tensor, leading to the following rules:

- ReLU(A_R) $= [ReLU(A)]_R$
- ReLU(A_{RS}) $= [ReLU(A)]_{RS}$
- ReLU(A_{CS}) $= [ReLU(A)]_{CS}$

LayerNorm The LayerNorm operator can only be applied to *replicated* tensors or *row-sliced* tensors since it is applied along the row dimension. Therefore, the rules for LayerNorm are as follows:

- LayerNorm(A_R) $= [LayerNorm(A)]_R$
- LayerNorm(A_{RS}) $= [LayerNorm(A)]_{RS}$

Self-Attention For the attention operator in the Transformer model, we treat it as a black box to simplify the search process because of the complexity of the attention mechanism. Moreover, the architecture of the attention mechanism makes it particularly suited for the parallelization scheme described by Megatron-LM [10], which is why we only implement a single rule. The Attention operator takes in a single matrix containing the concatenated key, query, and value matrices, and returns the attention matrix:

- Attention(A_{CS}) = $[Attention(A)]_{CS}$

Collective Communication For tensors distributed across GPUs, collective communication operations can be used to manipulate tensors on multiple devices at the same time. The rules that govern how they operate are as follows:

- AllGather(A_{CS}) $= A_R$
- AllGather(A_{RS}) $= A_R$
- ReduceScatter(A_L) $= A_{CS}$
- ReduceScatter(A_L) $= A_{RS}$
- Since AllReduce is equivalent to an ReduceScatter followed by an AllGather, AllReduce(A_L) $= A_R$

Transformer Layer To facilitate the searching of partitioning strategies while ensuring correctness, we also formalize the process of inference through a single Transformer layer using tensors and the operations described above.

Based on Figure 1, let A be the input tensor into a Transformer layer, QKV be the query, key, value matrix for the self-attention operation, W_0 be the weight matrix for linear layer in the self-attention block, and W_1 and W_2 be the weight matrices for the linear layers in the MLP block. We ignore the residuals introduced by the Add operator for simplicity, as these are element-wise operations that do not affect the search space of valid partitioning strategies. We can then formulate the resulting tensor from inference like so:

$$ReLU(LayerNorm(Attn(LayerNorm(A)QKV)W_0)W_1)W_2$$

Given this, any partitioning strategy for LLM inference can be defined as a set of operations on the input and weight tensors. By matching the resulting tensor of the partitioning strategy to the above tensor, we can verify the correctness of the partitioning strategy.

4 Search

In this section, we detail the process of developing a program to discover alternative partitioning strategies as well as our results.

4.1 PartitionSearch

Based on the formulation in Section 3, we developed PARTITIONSEARCH, a program capable of exhaustively searching across all valid partitioning strategies for any LLM model architecture. PARTITIONSEARCH keeps track of the tensor state and sizes symbolically during its search. Using this, for each discovered partitioning strategy, PARTITIONSEARCH symbolically calculates the weight FLOPs, communication volume, and weight memory for the strategy in terms of the input length, model parameters, and number of GPUs used.

By substituting specific values for the weight FLOPs, communication volume, and weight memory used, PARTITIONSEARCH ranks and identifies the Pareto frontier across all valid partitioning strategies in terms of these three objectives.

We consider a partitioning strategy to dominate another strategy if it less or equal in all objectives, and strictly less in at least one objective. A partitioning strategy is considered Pareto optimal it is not dominated by any other strategy and the set of all Pareto optimal strategies form the Pareto frontier.

4.2 Results

Figure 4 shows the results obtained by PARTITIONSEARCH for the OPT 13B model [16] across both small and large input lengths when parallelized across four GPUs. The results show that there are a many valid partitioning strategies for distributed LLM inference apart from Megatron-LM.

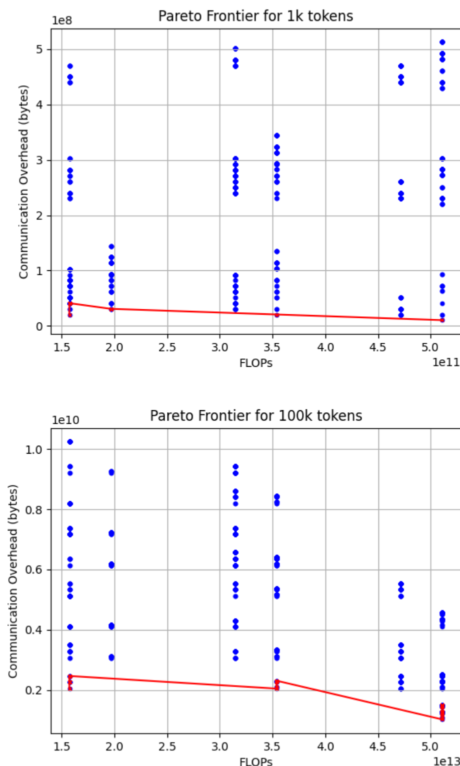

Figure 4: Characteristics of partitioning strategies discovered for 1k and 100k tokens on an OPT 13B model parallelized across 4 GPUs. The red line denotes the Pareto frontier for partitioning strategies across weight FLOPs, communication volume, and weight memory.

We observe that there is a wide variation in the performance of these partitioning strategies, with the values for weight FLOPs, communication volume, and weight memory differing by up to five times. Many of these partitioning strategies are not feasible practically because the FLOPs required or memory requirements are too high to run any model efficiently. Therefore, we prune all partitioning strategies that require more than double the weight FLOPs, communication overhead, or weight memory of the best partitioning strategy for that specific objective.

Based on this, we identified three viable Pareto optimal partitioning schemes for distributed LLM inference. Each of these partitioning schemes perform differently depending on the model and input length. Because these strategies share similar sub-strategies across both the self-attention block and MLP block, we first describe these sub-strategies below.

4.2.1 Sub-Strategies

Megatron-style Attention This refers to the partitioning scheme used for the self-attention block in Megatron-LM (§2.2.1). The communication overhead of this sub-strategy scales with input length because there is an all-reduce operation on the resulting tensor, and this dimensions of this tensor depend on the input length.

Projection Replicated Attention Instead of performing an all-reduce operation after the linear layer, an all-gather operation is performed on the attention matrix after the self attention operation to obtain the replicated tensor, while the weight tensor for the linear layer W_0 is fully replicated. Therefore, an all-reduce is no longer required to obtain the full tensor.

As compared to Megatron-style attention, projection replicated attention has a smaller communication overhead because it only requires an all-gather operation instead of an all-reduce operation. However, this comes at the expense of greater FLOPs and memory usage since the attention matrix and weight matrix are multiplied as fully-replicated tensors instead of sliced tensors.

Megatron-style MLP This refers to the partitioning scheme used for the MLP block in Megatron-LM (§2.2.1). Similar to with Megatron-style attention, the communication overhead of this sub-strategy scales with input length.

Weight-gathered MLP In weight-gathered MLP, the weights for both linear layers (W_1 and W_2) are fully replicated for each Transformer pass. However, instead of the loading the full weights for all Transformer layers on initialization, which would not be feasible in the real-world given the prohibitive memory requirements, an all-gather is executed to gather the weights for each Transformer layer *before* they are required. These gathered weights are then *discarded* after the operations, avoiding having to store the weights for all Transformer layers at once. Moreover, because the weight matrices are replicated, the input tensor to the MLP block is partitioned in a row-parallel manner to accelerate computation.

As compared to Megatron-style MLP, the communication overhead of weight-gathered MLP is independent of the input length because the collective communication operations are only performed on the weight matrices of the linear layers, which are fixed size for a given model. This communication overhead depends solely on the size of the model. We note that this overhead is significant, even for smaller models. For example, for an OPT-13B model [16] with 40 layers and a hidden dimension of 5120, the communication volume for a single weight-gathered MLP block is over 400 MB, which translates to over 16GB for MLP blocks across the entire model.

4.2.2 Overall Strategies

Using the above sub-strategies as building blocks, we now describe the Pareto optimal partitioning strategies for a single Transformer layer identified by PARTITIONSEARCH: MEGATRON, PROJECTIONREPLICATED, and WEIGHTGATHERED. Table 1 illustrates their performance in terms of weight FLOPs, communication volume, and weight memory.

MEGATRON This is the same partitioning strategy used by Megatron-LM (§2.2.1). Out of the three partitioning

	Megatron	Projection-Replicated	Weight-Gathered
Weight FLOPs	$24d^2n/g$	$2d^2n$+$22d^2n/g$	$24d^2n/g$
Communication Vol. (B)	$8dn$	$6dn$	$4dn$+$16d^2$
Weight Memory (B)	$18d^2/g$	$16d^2/g$+$2d^2$	$18d^2/g$

Table 1: Characteristics of each strategy calculated by PARTITIONSEARCH. d denotes the hidden size of the model, n denotes the input length, and g denotes the number of GPUs used.

strategies, it requires the least weight FLOPs, tied with WEIGHTGATHERED, making it most efficient when compute, and not communication, is the bottleneck. This depends on both the specifications of the GPU and the input length, but in general, this is the case for smaller sequence lengths since the amount of communication volume will be lower.

PROJECTIONREPLICATED This strategy is a combination of projection replicated attention and Megatron-style MLP. Since the communication overhead for projection replicated attention is lower as compared to Megatron-style attention, this strategy is more efficient than the MEGATRON when communication is the bottleneck. This is the case in the prefill phase of LLM inference (§2.1) when the input length is sufficiently long since the entire set of *input prompt* tokens have to be processed at once. When communication overhead becomes a bottleneck is also affected by the specifications of the GPUs used, specifically the inter-GPU communication bandwidth. On GPUs with lower interconnect bandwidth such as the L4 GPU, communication becomes a bottleneck at shorter input lengths.

WEIGHTGATHERED This strategy is a combination of Megatron-style attention and weight-gathered MLP. Because weight-gathered MLP expects its input to be partitioned in a column-parallel manner, the all-reduce operation after the original Megatron-style attention is replaced with a reduce-scatter operation instead. Since the communication overhead for weight-gathered MLP does not scale with input length, there is a threshold whereby the input length is sufficiently long such that the communication overhead of PROJECTIONREPLICATED exceeds that of this strategy. Therefore, in such a scenario, this strategy is the most efficient out of all three strategies.

5 System Design

This section details the design of our LLM inference engine based on dynamic partitioning.

5.1 Weight Layout

Importantly, the weights for the LLM are loaded in a fashion that allows the inference engine to switch between different strategies at inference time efficiently. By doing so, we avoid having to move the weights around in GPU memory at inference time, which would incur additional GPU memory and significantly increase latency.

Self-Attention Block The weights for the query, key, and value matrices are replicated fully on each GPU because the self-attention mechanism is unchanged for all strategies. The weights for the output linear layer in the self-attention block are also fully replicated on GPUs. Even though Megatron-style attention only requires these weights to be partitioned in a column-parallel manner, fully replicating these weights allow us to switch between Megatron-style attention and projection replicated attention at inference time. Therefore, we trade-off increased weight memory for reduced switching cost at inference time.

MLP Block Since the weights are partitioned similarly for both Megatron-style MLP and weight-gathered MLP, we reuse the same weight layout as in Megatron-LM (§2.2.1): the weights for the first linear layer are partitioned in a column-parallel manner, while the weights for the second linear layer are partitioned in a row-parallel manner.

5.2 Switching Thresholds

The inference engine switches between the three strategies at inference time using input length thresholds based on calculations detailed below.

From MEGATRON to PROJECTIONREPLICATED MEGATRON is the most efficient for shorter sequence lengths when inference is compute bound because it requires the least FLOPs out of the three strategies. PROJECTIONREPLICATED becomes more efficient when the input length increases to a point where communication, and not compute, becomes the bottleneck instead. Formally, denote the inter-GPU communication bandwidth and FLOPs of the GPUs used as x GB/s and y FLOPs respectively. Given a partitioning strategy with communication volume C GB and F FLOPs, the time taken for communication is x/C while the time taken for computation is y/F.

Therefore, when the time taken for communication exceeds the time taken for computation, the inference engine switches from MEGATRON to PROJECTIONREPLICATED as the partitioning strategy.

From PROJECTIONREPLICATED to WEIGHTGATHERED We calculate the threshold for the input length whereby the communication volume of PROJECTIONREPLICATED exceeds that of WEIGHTGATHERED, making WEIGHTGATHERED more efficient. Let n denote the input length and d denote the hidden dimension of the model. Based on the values in Table 1, for an OPT model [16], the communication volume of PROJECTIONREPLICATED exceeds that of WEIGHTGATHERED when $n > 8d$. Therefore, when the input length is longer than this threshold, the inference engine switches from using PROJECTIONREPLICATED to WEIGHTGATHERED when serving an OPT model. This value differs for other LLM model architectures. For instance, for the Llama 2 model architecture [13], the threshold is $n > 3m$ instead, where m is

the model's intermediate dimension.

We note that this threshold is an overestimate as it is a theoretical value that does not account for the fact that with WEIGHTGATHERED, weight-gathered MLP can overlap computation and communication on the GPU more easily, since the inference engine can queue all-gather operations for the weight tensors before they are required. Accounting for this would lead to a lower input length threshold.

6 Implementation

We developed a prototype LLM inference library in about 2k lines of code in Python that implements dynamic partitioning. The inference library is based on a minimal version of vLLM [7] that includes specific features for speeding up inference such as PagedAttention. For the model executor, we implemented support for Llama 2 [13] using PyTorch [17]. For the collective communication of tensors across GPUs, we use NCCL [18]. We verified the correctness of the inference library by comparing the outputs of the Llama 2 models on a variety of prompts to that of the Transformers library [19].

7 Evaluations

In this section, we evaluate the performance of dynamic partitioning under a variety of workloads.

7.1 Experimental Setup

Model and cluster configuration We evaluate dynamic partitioning on the Llama 2 family of LLMs with 7B, 13B, and 70B parameters [13]. The Llama 2 LLMs are the most popular open-source models based on a LLM leaderboard [20], and the range of model sizes evaluated cover a variety of use cases. We evaluate the 7B and 13B Llama 2 models on four L4 NVIDIA GPUs and evaluate the 70B Llama 2 model on four A100-80GB NVIDIA GPUs consistent with industry norms, all provisioned on Google Cloud Platform. Similarly, we perform inference on half-precision weights to save GPU memory and speed up inference. The L4 NVIDIA GPU is a lower-end GPU with a constrained inter-GPU communication bandwidth of 64 GB/s using PCIe Gen4 and achieves up to 242 TFLOPs of performance on half-precision floating point numbers. On the other hand, the A100-80GB NVIDIA GPU is a higher-end GPU that uses NVIDIA's proprietary interconnect NVLink, allowing it to achieve an inter-GPU communication bandwidth of up to 600 GB/s, while also reaching 624 TFLOPs on half-precision floating point numbers.

Metrics We use time to first token and latency as our main metrics of success. Time to first token refers to how quickly users see the first token after submitting the prompt, and a shorter time to first token translates to better responsiveness, which is critical for interactive use cases. This is affected by how long it takes for the model to process the entire prompt, generate the first output token, and return it to the user. Next, latency refers to the time required to generate the entire output response, and corresponds to the speed of LLM inference perceived by the user. Low latency is essential for a smooth user experience, especially in applications that involve real-time interaction with the user such as chatbots or search. Considering the inference phases (§2.1), time to first token measures the time taken only for the prefill phase, while latency measures the time taken for both the prefill phase and auto-regressive generation phase.

Baselines For each of the model and cluster configurations, we evaluate the time to first token and latency using each partitioning strategies individually, as well as using dynamic partitioning where we switch between the three strategies dynamically. For both of these metrics, we evaluate the models on increasing prompt lengths from 1 to 64678. Because of the constrained GPU memory available on L4 GPUs, we only evaluate up to a prompt length of 32384 for the 13B model. As the main baseline for comparison, we consider MEGATRON, the strategy used by Megatron-LM [10], which is the state-of-the-art approach to model parallelism used in modern LLM inference engines such as vLLM [7] and TGI [8].

7.2 Results

Figure 5 shows the time to first token for varying prompt lengths on Llama 2 [13] using both dynamic partitioning and each strategy individually. We observe that for the 7B and 13B models, there is no single partitioning strategy that achieves the shortest time to first token for all input lengths, highlighting the importance of a dynamic strategy that is able to switch between these individual strategies. For the 7B model evaluated on L4 NVIDIA GPUs, MEGATRON achieves the shortest time to first token for a prompt length of 1, PROJECTIONREPLICATED achieves the shortest time to first token for intermediate prompt lengths up to 16192, while WEIGHTGATHERED achieves the shortest time to first token for longer prompt lengths up to 64678. For the 13B model, MEGATRON achieves the shortest time to first token for prompts containing 1 token, followed by PROJECTIONREPLICATED for prompts up to 32384 tokens. Finally, for the Llama 2 70B model evaluated on A100 NVIDIA GPUs, MEGATRON achieves the shortest time to first token for all prompt lengths.

A100 GPUs have a significantly higher inter-GPU communication bandwidth as compared to L4 GPUs. As such, inference is bottlenecked by compute for all evaluated prompt lengths, which explains why MEGATRON always achieves the shortest time to first token for the 70B model, unlike with the other two model sizes. With the 7B and 13B models evaluated on L4 GPUs, communication becomes the bottleneck when the input length is 1024, leading to PROJECTIONREPLICATED achieving a shorter time to first token than MEGATRON from this point on. Based on the calculation that the threshold for which WEIGHTGATHERED becomes more efficient than PROJECTIONREPLICATED when $n > 3m$ (§5.2), this threshold is approximately 33024 for the 7B model ($m = 11008$). Consistent with these calculations, WEIGHTGATHERED achieves a shorter time to first token than PROJECTIONREPLICATED for the 7B model when the input length is at least 32384. This value is lower than the calculated

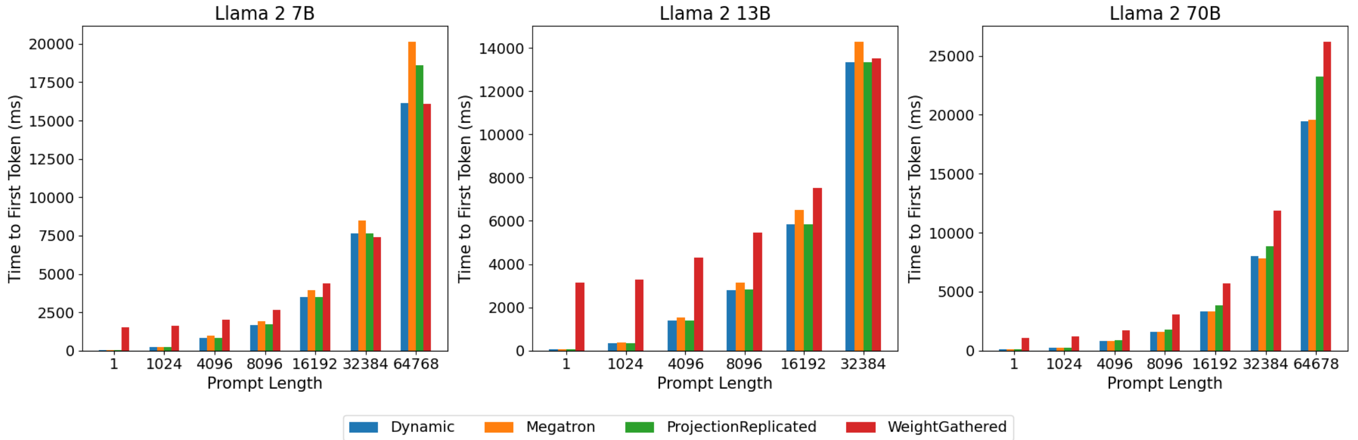

Figure 5: Time to first token for Llama 2 models

theoretical threshold due to overlapping of computation and communication in WEIGHTGATHERED as discussed (§5.2).

For all three models, dynamic partitioning achieves the shortest or close to the shortest time to first token for all prompt lengths, illustrating the effectiveness of this approach as compared to using a static partitioning strategy. Specifically, as compared to MEGATRON, using a dynamic partitioning strategy achieves significant reductions in the time to first token of 10% to 40% for the 7B model, and 6% to 12% for the 13B model. For Llama 2 70B evaluated on A100 GPUs, because MEGATRON is always the optimal strategy, there is no significant improvement or regression from using dynamic partitioning.

Figure 6 shows the latency on varying prompt lengths for both shorter output sequences of 16 tokens and longer output sequences of 64 tokens.We observe that because of the the high communication overhead of WEIGHTGATHERED on short input lengths (§4.2.1), using it as a static partitioning strategy for inference leads to significantly higher latency of up to 50 times, a trend that holds across all models and input lengths. Using WEIGHTGATHERED for the auto-regressive LLM generation phase (§2.1) is extremely inefficient communication-wise since the input length is always one. The highlights the importance of having a dynamic partitioning scheme that is able to use a strategy such as WEIGHTGATHERED purely for the prefill phase (§2.1), and to use more efficient strategies such as MEGATRON or PROJECTIONREPLICATED for the auto-regressive generation phase. Similar to with the time to first token, we observe that for the Llama 2 7B and 13B models, there is no single partitioning strategy that achieves the lowest latency across all input lengths. With a shorter output length of 16, MEGATRON achieves the lowest latency for both the 7B and 13B models when the prompt length is 1, while PROJECTIONREPLICATED achieves the lowest latency for longer prompt lengths. While MEGATRON is indeed a more efficient strategy than PROJECTIONREPLICATED for the generation phase since this phase is compute bound, the lower latency achieved by PROJECTIONREPLICATED for greater prompt lengths can be attributed to the time saved from the more efficient prefill phase (due to the lower communication overhead) outweighing the less efficient generation phase. When the output length is increased to 64, we observe that the threshold whereby PROJECTIONREPLICATED achieves a lower latency than MEGATRON increases to a prompt length of between 4096 and 8192. This is because with a longer output length, the overhead from using the more inefficient PROJECTIONREPLICATED for the generation phase becomes more significant, requiring a longer prompt length to offset this with the prefill phase. For the 70B model, MEGATRON again achieves the lowest latency for all input and output lengths due to the significantly higher inter-GPU communication bandwidth of the A100 GPU.

For all three models, we again observe that dynamic partitioning achieves the lowest or close to the lowest latency for all prompt lengths, highlighting the effectiveness of this strategy. As compared to MEGATRON, using dynamic partitioning leads to a reduction in latency of up to 18% for the 7B model and up to 6% for the 13B model on both short and long output lengths, demonstrating a notable improvement over the approach used in existing inference engines. By taking into the account the input length, model size, and GPU characteristics, dynamic partitioning is able to switch between MEGATRON, PROJECTIONREPLICATED, and WEIGHTGATHERED to optimize for latency, ensuring that performance remains high for a range of prompt lengths. We note when dynamic partitioning is used, because the LLM generation phase always contains one token, MEGATRON is always used for this. Therefore, the main benefit from using dynamic partitioning comes from being able to switch between the three strategies during the prefill phase (§2.1).

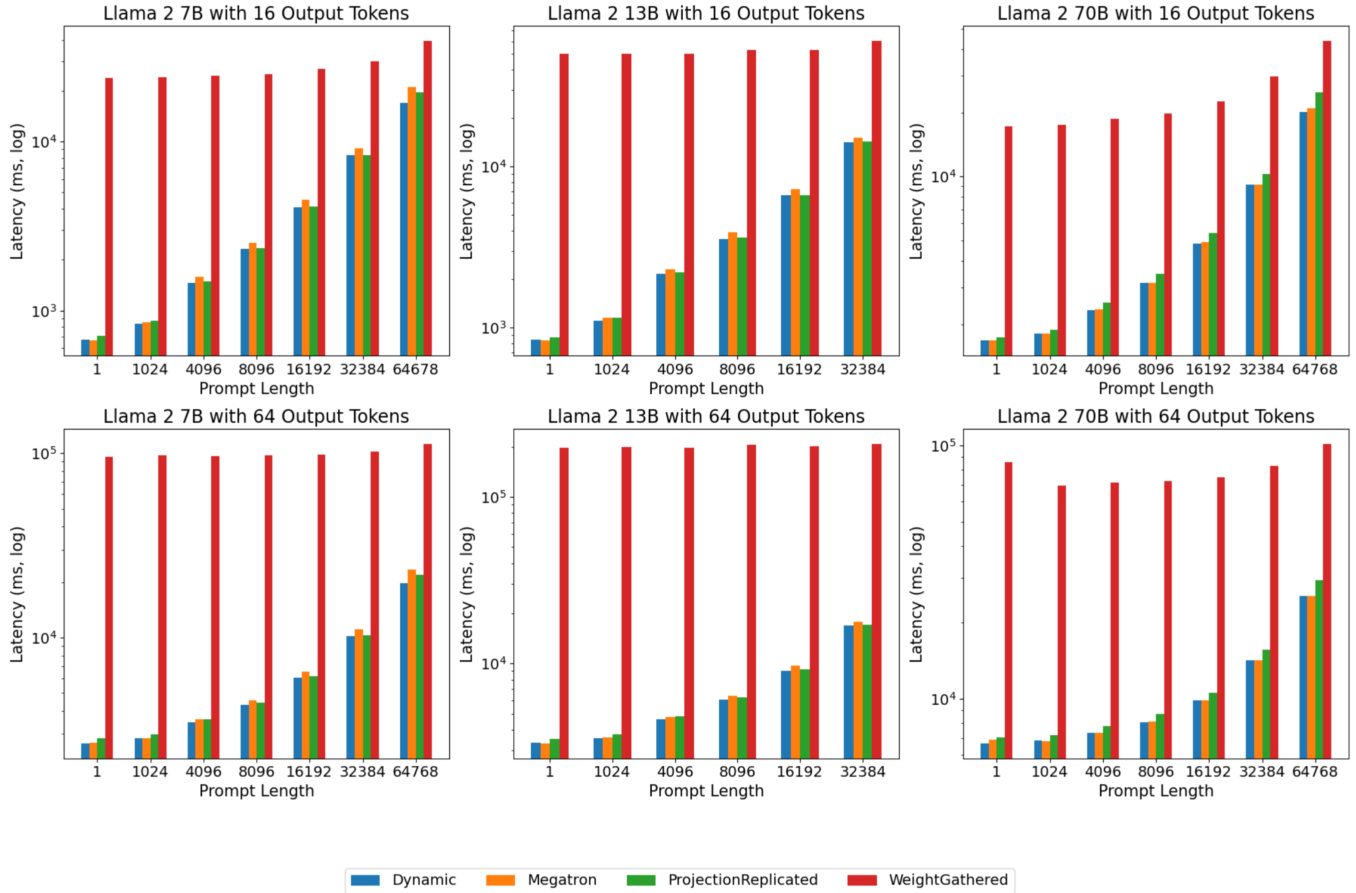

Figure 6: Latency for Llama 2 model generation on short and long outputs (log scale)

8 Discussion

Our findings demonstrate the nuanced relationship between model size, GPU capabilities, and partitioning strategies. The effectiveness of the different partitioning strategies MEGATRON, PROJECTIONREPLICATED, and WEIGHTGATHERED, varies notably across different model sizes and GPUs. The superior interconnect bandwidth of A100 GPUs enables MEGATRON to consistently outperform other strategies for the 70B model whereas with the 7B and 13B models on L4 GPUs, there is no single partitioning strategy that dominates across all input lengths. Instead, there is a clear transition point whereby communication becomes the bottleneck instead of computation, necessitating a shift in partitioning strategy from MEGATRON to PROJECTIONREPLICATED.

The strong observed performance of dynamic partitioning in all these scenarios underscores its ability to optimize the speed of LLM inference across a range of model and hardware configurations. Importantly, when compared to MEGATRON, the current state-of-the-art approach for model parallelism in LLMs, we demonstrate that dynamic partitioning achieves significant improvements in both the time to first token and overall latency for the 7B and 13B models without any performance degradation for the 70B model. This illustrates dynamic partitioning to be a versatile and effective approach to model parallelism for distributed LLM inference. This versatility is especially crucial for real-world LLM applications today, where input lengths and computational demands can vary widely.

9 Related Works

Parallelism for LLM inference Prior works in the space have proposed multiple approaches for training and serving large models more efficiently via specific partitioning strategies. FasterTransformer [21] establishes a suite built using C++, CUDA, and cuBLAS for benchmarking single-GPU and multi-GPU inference for different types of Transformer models across many model sizes. It exploits both tensor parallelism and pipeline parallelism, along with other optimizations such as quantization. EffectiveTransformer [22] is an inference library built on top of FasterTransformer that reduces memory usage while increasing execution speed by dynamically adjusting the padding on intermediate tensors. Similarly, TensorRT LLM [23] is an open-source library built on top of FasterTransformer that optimizes the inference performance of LLMs on NVIDIA's GPUs. It wraps NVIDIA's TensorRT deep learning

compiler that uses the optimized kernels from FasterTransformer, performs pre-processing and post-processing, as well as other optimizations such such as FlashAttention [24] and the 8-bit floating point data type. DeepSpeed Inference [25] aims to further accelerate inference by leveraging ZeRO offload to utilize CPU and NVMe memory on top of GPU memory for models which do not fit in GPU memory. GSPMD [26] is a compiler-based system for model computation whereby users can provide hints for how to partition tensors across devices, based on which the system will automatically distribute tensor computation. This work shares many partitioning strategies introduced by these prior works, but the act of dynamically switching between strategies at inference time is novel.

Improving inference efficiency Several works have also focused on improving the inference efficiency of Transformer models by proposing improvements to the model architecture and inference engine. These include improving the efficiency of the self-attention block [27–29], quantization techniques to reduce the memory required by LLMs [30–32], as well as model distillation [33, 34], where smaller specialized models are trained using larger models. FlashAttention [24] introduces an IO-aware exact attention algorithm that reduces the number of memory IO operations between GPU HBM and SRAM, reducing the memory bottleneck and enabling faster inference for Transformer models. PagedAttention [7] significantly reduces memory fragmentation and duplication in the key-value-cache for Transformer inference, reducing memory usage for inference. Dynamic partitioning is orthogonal to these techniques, and can be used in conjunction with them to further speed up inference. We utilize a number of these techniques such as FlashAttention and PagedAttention in our library implementation.

Optimizing model parallelism automatically Finally, a line of work also focuses on being able to automatically determine the optimal approach to model parallelism for large models. Tofu [35] uses a dynamic programming approach to identify the optimal partitioning strategy for neural network models across multiple GPUs. Similarly, TensorOpt [36] uses a dynamic programming algorithm to identify new model parallelism strategies that trade-off between different objectives such as memory and cost of computation. Piper [37] is an efficient optimization algorithm using dynamic programming to partition models across multiple GPUs when leveraging data parallelism, tensor model parallelism, pipeline model parallelism, and other memory optimizations. FlexFlow [38] focuses on a more comprehensive search of parallelization strategies along the Sample, Operation, Attribute, and Parameter dimensions (SOAP), using guided randomized search of this space to identify the fastest parallelization strategy. Varuna [39] focuses on commodity networking clusters, and determines the optimal pipeline parallelism and data parallelism strategy for training large models across these devices, significantly reducing cost and improving training time. Finally, Alpa [40] generalizes the search for optimal parallelism strategies using both integer linear programming and dynamic programming, supporting a comprehensive search of different strategies for distributed model training. While these works target training for general machine learning models, dynamic partitioning focuses specifically on inference for LLMs, and our key observation about the wide-ranging input lengths for these workloads allows for new optimizations.

10 Conclusions and Future Work

This paper introduced dynamic partitioning, a new approach towards model parallelism for distributed LLM inference where we dynamically switch between partitioning strategies at inference time depending on the model, GPU specifications, and input length. We conducted a systematic search of all viable partitioning strategies and identified three Pareto optimal strategies for parallelizing transformer inference across GPUs. Based on these strategies, we developed an LLM inference library that implements dynamic partitioning, demonstrating significant improvements in the time to first token and latency across the Llama 2 models [13] as compared to the de-facto approach introduced by Megatron-LM [10]. In particular, we find that dynamic partitioning is most effective on smaller LLMs and GPUs with lower interconnect bandwidth and compute.

There are several directions for future work. Our current analysis of different partitioning strategies (§4.2.2) calculates a theoretical value for the weight FLOPs, communication volume, and weight memory. This does not account for how certain strategies can take greater advantage of overlapping communication and computation on the GPU, which might affect the choice of Pareto optimal strategies. Therefore, we plan to investigate how to formalize the ability of specific partitioning strategies to overlap computation and communication more easily. This would allow us to conduct a more extensive search of partitioning strategies using PartitionSearch, which might in turn lead to a larger set of optimal strategies for dynamic partitioning.

Additionally, while this work focuses on NVIDIA GPUs as a reflection of their widespread use, it leaves open the question of how dynamic partitioning would perform on other hardware architectures. Hence, we hope to extend our inference library to work with other accelerators such as AMD GPUs and Google Cloud TPUs, as well as investigate how dynamic partitioning performs on these accelerators.

Finally, this work targets dense LLMs, but it would be interesting to see how dynamic partitioning can be applied to other related model architectures. For example, mixture of expert models decompose LLMs into smaller sub-models that focus on specific aspects of the input data, enabling more efficient inference and resource utilization. Conducting a systematic search of partitioning strategies for such models might lead to new Pareto optimal strategies for dynamic partitioning.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, May 2019. arXiv:1810.04805 [cs].

[2] Alec Radford, Jeff Wu, Rewon Child, D. Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners, July 2020. arXiv:2005.14165 [cs].

[4] OpenAI. ChatGPT, 2022. https://openai.com/blog/chatgpt.

[5] GitHub. GitHub Copilot, 2022. https://github.com/features/copilot.

[6] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity, June 2022. arXiv:2101.03961 [cs].

[7] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention, September 2023. arXiv:2309.06180 [cs].

[8] HuggingFace. Text generation interface. https://github.com/huggingface/text-generation-inference.

[9] NVIDIA. Tensorrt-lllm. https://github.com/NVIDIA/TensorRT-LLM.

[10] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, March 2020. arXiv:1909.08053 [cs].

[11] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks, April 2021. arXiv:2005.11401 [cs].

[12] Anthropic. Claude, 2023. https://www.anthropic.com/index/introducing-claude.

[13] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models, July 2023. arXiv:2307.09288 [cs].

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, August 2023. arXiv:1706.03762 [cs].

[15] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Sarikivi. Breaking the Computation and Communication Abstraction Barrier in Distributed Machine Learning Workloads, March 2022. arXiv:2105.05720 [cs].

[16] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models, June 2022. arXiv:2205.01068 [cs].

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[18] NVIDIA. NCCL: The NVIDIA Collective Communication Library, 2023. https://developer.nvidia.com/nccl.

[19] HuggingFace. Transformers. https://huggingface.co/docs/transformers/index.

[20] LMSYS Org. Chatbot arena. https://chat.lmsys.org/.

[21] NVIDIA. FasterTransformer, 2023. https://github.com/NVIDIA/ FasterTransformer.

[22] ByteDance. Effectivetransformer. https://github.com/bytedance/effective$_{t}$*ransformer.*

[23] NVIDIA. TensorRT LLM, 2023.

[24] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, June 2022. arXiv:2205.14135 [cs].

[25] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale, June 2022. arXiv:2207.00032 [cs].

[26] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. GSPMD: General and Scalable Parallelization for ML Computation Graphs, December 2021. arXiv:2105.04663 [cs].

[27] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking Attention with Performers, November 2022. arXiv:2009.14794 [cs, stat].

[28] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient Content-Based Sparse Attention with Routing Transformers, October 2020. arXiv:2003.05997 [cs, eess, stat].

[29] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The Efficient Transformer, February 2020. arXiv:2001.04451 [cs, stat].

[30] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale, November 2022. arXiv:2208.07339 [cs].

[31] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8BERT: Quantized 8Bit BERT. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, pages 36–39, December 2019. arXiv:1910.06188 [cs].

[32] Yijia Zhang, Lingran Zhao, Shijie Cao, Wenqiang Wang, Ting Cao, Fan Yang, Mao Yang, Shanghang Zhang, and Ningyi Xu. Integer or Floating Point? New Outlooks for Low-Bit Quantization on Large Language Models, May 2023. arXiv:2305.12356 [cs].

[33] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter, February 2020. arXiv:1910.01108 [cs].

[34] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient Knowledge Distillation for BERT Model Compression, August 2019. arXiv:1908.09355 [cs].

[35] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, March 2019. arXiv:1807.08887 [cs].

[36] Zhenkun Cai, Kaihao Ma, Xiao Yan, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. TensorOpt: Exploring the Tradeoffs in Distributed DNN Training with Auto-Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1967–1981, August 2022. arXiv:2004.10856 [cs, stat].

[37] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional Planner for DNN Parallelization. In *Advances in Neural Information Processing Systems*, volume 34, pages 24829–24840. Curran Associates, Inc., 2021.

[38] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks, July 2018. arXiv:1807.05358 [cs].

[39] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: Scalable, Low-cost Training of Massive Deep Learning Models, November 2021. arXiv:2111.04007 [cs].

[40] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning, June 2022. arXiv:2201.12023 [cs].