Efficient Distributed LLM Inference with Dynamic Partitioning

Isaac Ong

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-108
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-108.html

May 16, 2024

Efficient Distributed LLM Inference with Dynamic Partitioning

by

Isaac Ong

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Joseph E. Gonzalez

Spring 2024

Abstract

Efficient Distributed LLM Inference with Dynamic Partitioning

by

Isaac Ong

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Ion Stoica, Chair

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.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my research advisor Professor Ion Stoica for his constant guidance and feedback. This report is based on my broader research work with Woosuk Kwon on investigating tensor parallelism for distributed LLM inference, and I would like to sincerely thank him for his continued mentorship. Finally, I would also like to thank my family for their unwavering support throughout my studies and pursuits.

Chapter 1

Introduction

Recent advances in machine learning have enabled an exponential increase in model size of large language models (LLMs), from BERT[10] with 340 million parameters to GPT-2[27] with 1.5 billion parameters, and GPT-3[4] with 175 billion parameters. The emergence of these large language models have led to myriad new use cases such as chatbots like ChatGPT[24] and coding assistants like GitHub Copilot[12].

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 [4] 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 [11]. Therefore, model inference must be parallelized across multiple GPUs to be served efficiently.

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 accelerate 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 [18], HuggingFace's TGI [13], and NVIDIA's TensorRT-LLM [23] make use of the approach proposed by Megatron-LM[30], 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 [19], 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 [27] to over 100,000 tokens supported by Claude [2], 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, input length should be considered when determining an optimal partitioning strategy. At the same time, the performance of different partitioning strategies can also vary greatly 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 develop a system capable of conducting an exhaustive search over all partitioning strategies for distributed Transformer inference considering their performance with respect to FLOPs, communication overhead, and memory requirements. We then identify 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 inference-time dynamic partitioning on both L4 and A100 NVIDIA GPUs using the Llama 2 7B, 13B, and 70B models [33]. Our evaluation results show that as compared to using the static partitioning strategy from Megatron-LM [30], 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 develop a system capable of conducting an exhaustive search over all feasible partitioning strategies for distributed LLM inference and identify the Pareto frontier of partitioning strategies based on 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, making it a promising future direction for optimizing distributed LLM inference.

Chapter 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

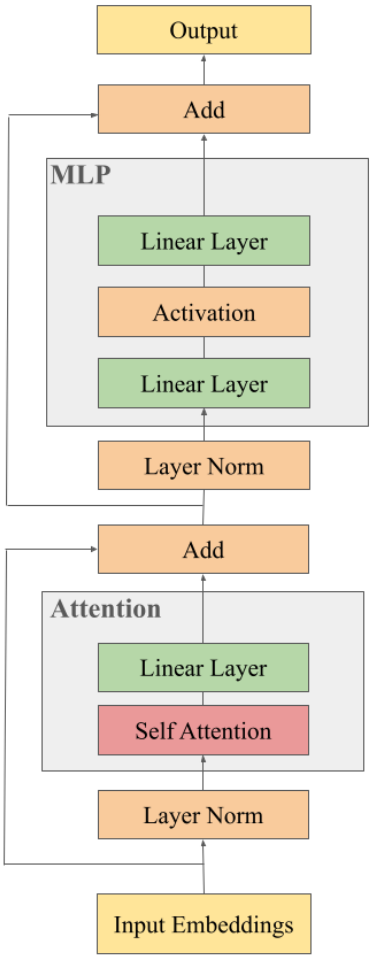

Figure 2.1: The architecture for a single Transformer layer

LLMs are based on the auto-regressive Transformer architecture [34], show in a simplified manner in Figure 2.1. 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.

Model Parallelism

In general, there are two main approaches to model parallelism: pipeline parallelism and tensor parallelism. Both of these approaches aim to distribute the computational and memory requirements of large models across multiple GPUs or accelerators, but they have different trade-offs and limitations.

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 like in a pipeline. This allows for efficient utilization of GPU resources by overlapping computation and communication, as each GPU can start processing the next batch of data as soon as it has finished its assigned operations and passed the results to the next GPU.

Tensor parallelism Tensor parallelism is an orthogonal approach to pipeline parallelism whereby which involves distributing the computation of individual tensors across multiple GPUs, allowing for parallel processing of different parts of the tensor. Tensor parallelism can be applied to various operations, such as matrix multiplications, convolutions, and element-wise operations. By partitioning the computation, tensor parallelism can enable the training and inference of larger models that would otherwise not fit into the memory of a single GPU. However, this approach can lead to increased communication costs due to the need for synchronization and data transfer between GPUs, which can become a bottleneck for large models with high communication volumes. Therefore, careful design of communication patterns and synchronization mechanisms are crucial for achieving strong performance with tensor parallelism.

2.2 Transformer Model Parallelism

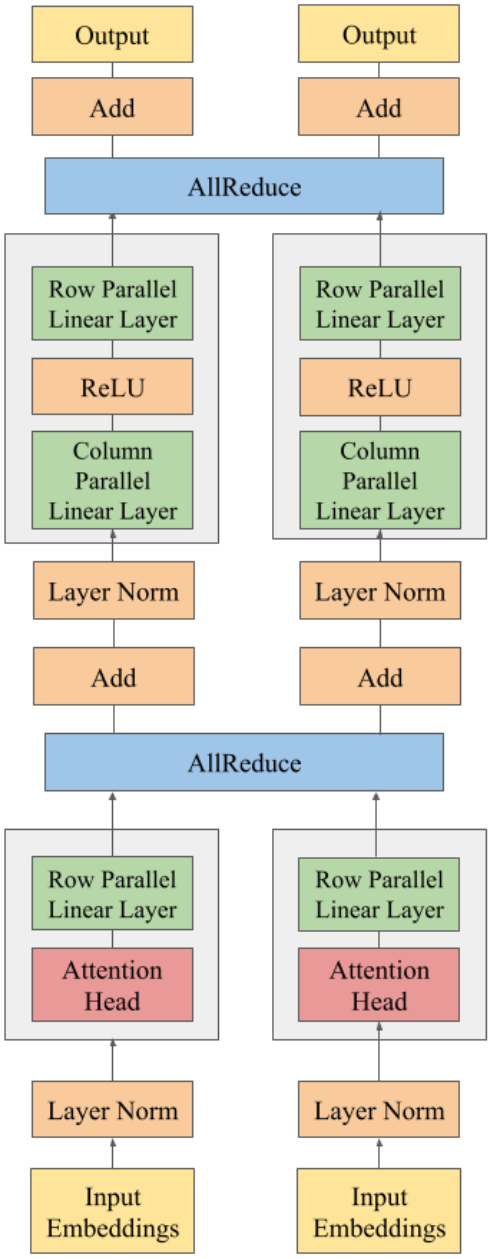

Figure 2.2: The partitioning strategy used by Megatron-LM for 2 GPUs

For Transformer models specifically, Megatron-LM [30] introduced a tensor parallelism strategy for both the self-attention blocks and MLP blocks, as shown in Figure 2.2.

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.

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 [30] 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.

Chapter 3

Formulation

We formalize the problem of identifying valid partitioning strategy given a model architecture by specify the set of states that tensor can belong to when distributed across GPUs and the set of operations that can be performed on them. Note that for the purposes of the formulation, we assume that the row and column dimensions of activation tensors corresponds to the sequence dimension and hidden dimension respectively.

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 3.1:

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

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

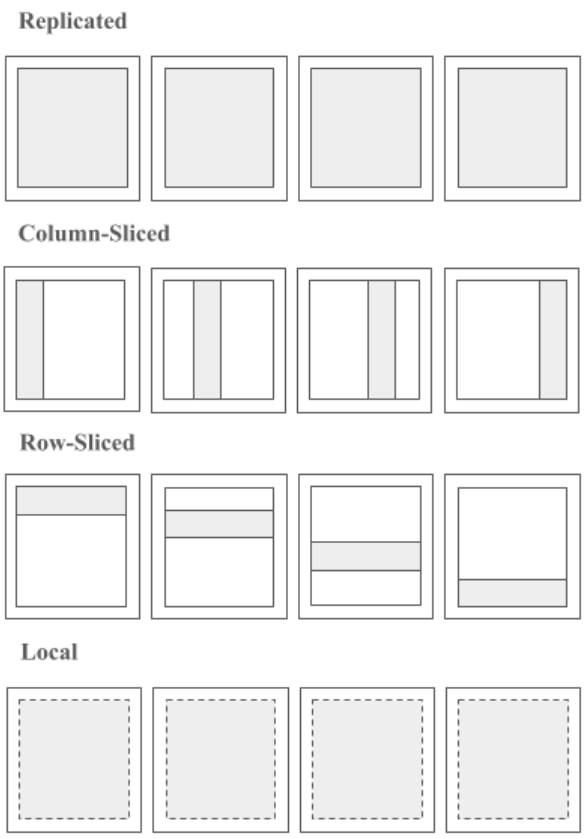

Figure 3.1: The valid states for a given tensor A when distributed across GPUs

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 operations on tensors:

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

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 operations for LayerNorm are as follows:

- $\text{LayerNorm}(A_R) = [LayerNorm(A)]_R$

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

Self-Attention For the attention operator in the Transformer model, we treat it as a single operator to simplify the search process. Moreover, the architecture of the attention mechanism makes it particularly suited for the parallelization scheme described by Megatron-LM [30], which is why we treat self-attention as a single operation for this problem. There are two supported attention operations: the first takes in the concatenated query, key, value tensors for a single attention head and returns the resulting attention tensor, while the second takes in the full query, key, value tensors for all attention heads, and returns the full attention tensor:

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

Collective Communication For tensors distributed across GPUs, collective communication operations can be used to manipulate tensors on multiple devices at the same time. The AllGather operation gathers part of the tensor from all ranks and distributes that full tensor to all ranks, while the ReduceScatter operation reduces the tensor such that the result is equally scattered across all ranks. AllReduce is equivalent to an ReduceScatter followed by an AllGather operation. Finally, we also define an AllToAll operation where each rank sends a section of the tensor it stores such that a *column-sliced* tensor converted to a *row-sliced* tensor, and vice versa.

- AllGather$(A_{CS}) = A_R$
- AllGather$(A_{RS}) = A_R$
- ReduceScatter$(A_L) = A_{CS}$
- ReduceScatter$(A_L) = A_{RS}$
- AllReduce$(A_L) = A_R$
- AllToAll$(A_{RS}) = A_{CS}$
- AllToAll$(A_{CS}) = A_{RS}$

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

Based on Figure 2.1, let A be the input tensor into a Transformer layer, QKV be the query, key, value weights 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 operations of a single Transformer layer like so:

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

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.

Chapter 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 System

Based on the formulation in Section 3, we developed PartitionSearch, a system for searching across all valid partitioning strategies for any Transformer model architecture given a specific memory allocation. PartitionSearch keeps track of the tensor state and sizes symbolically during the search process. Using this data, for each valid discovered 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.

We developed this system in about 1300 lines of Python using SymPy for symbolic manipulation. We utilize multiprocessing to perform recursive backtracking across all tensor states, and aggressively prune paths that do not lead to the target Transformer state, allowing us to search across hundreds of thousands of possibilities efficiently.

4.2 Results

By substituting specific symbol values for all discovered partitioning strategies, PartitionSearch ranks and identifies the Pareto frontier across all valid partitioning strategies in terms of the FLOPs required and communication overhead.

We consider a partitioning strategy to dominate another strategy if it better or equal in all objectives, and strictly better 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.

Figure 4.1 shows the results obtained by PartitionSearch for the Llama 2 7B model [38] as an example, when the number of input tokens is 1024 and when the model is par-

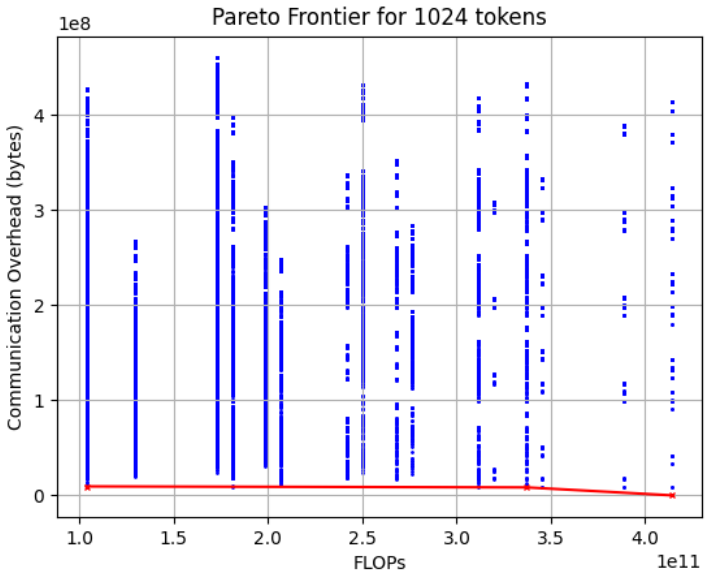

Figure 4.1: Partitioning strategies for Llama 2 7B discovered for 1024 the Llama 2 7B model parallelized across 4 GPUs. Each blue dot represents a partitioning strategy. The red line denotes the Pareto frontier for partitioning strategies across weight FLOPs and communication volume.

allelized across four GPUs. The results indicate that there are many valid partitioning strategies for distributed LLM inference apart from Megatron-LM, while the Pareto frontier (denoted in red) is comprised of a small subset of all valid partitioning strategies.

We observe that there is a wide variation in the performance of these partitioning strategies, with the values for weight FLOPs communication volume 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, while there are a large number of valid strategies, the actual set of practical partitioning strategies is much smaller. Using PartitionSearch, we identified three viable Pareto optimal partitioning schemes for distributed LLM inference across different input lengths. Each of these partitioning schemes exhibits different characteristics and performs differently depending on the model, input length and types of GPUs used.

Because these strategies share similar sub-strategies across both the self-attention block and MLP block, we first describe these sub-strategies below.

4.3 Sub-Strategies

Megatron-style Attention This refers to the partitioning scheme used for the self-attention block in Megatron-LM (§2.2). 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 output projection 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). 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 [38] with 40 layers and a hidden dimension of 5120, using FP16 precision, 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.

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

Table 4.1: 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.

4.4 Overall Strategies

Using the above sub-strategies as building blocks, we now detail the Pareto optimal partitioning strategies for a single Transformer layer identified by PartitionSearch: Megatron, ProjectionReplicated, and WeightGathered. Table 4.1 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). Out of the three partitioning 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 required for synchronization is smaller.

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 when the input length is sufficiently large in the prefill phase of LLM inference (§2.1) since the entire set of *input prompt* tokens have to be processed at once, and the synchronization overhead is significant. GPU characteristics, namely the inter-GPU communication bandwidth, also affect the degree to which communication becomes a bottleneck. 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 PROJECTION-REPLICATED exceeds that of this strategy. Therefore, in such a scenario, this strategy is the most efficient out of all three strategies.

Chapter 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 across all Transformer layers are partitioned in a column-parallel manner just as in Megatron-LM 2.2 since the self-attention mechanism is unchanged for all strategies. However, the weights for the output projection linear layer in the self-attention block are fully replicated on all GPUs. Even though Megatron-style 2.2 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 with minimal overhead. Therefore, we trade-off increased memory usage 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): 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 4.1, for an OPT model [38], 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 [33], 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 for switching.

Chapter 6

Implementation

We developed a prototype GPU-based LLM inference library for dynamic partitioning in about 2000 lines of code in Python. The inference library is based on a minimal version of vLLM [18] that includes specific features for speeding up inference such as PagedAttention and iteration-level scheduling. For the model executor, we implemented support for Llama 2 [33] using PyTorch [26]. For the collective communication of tensors across GPUs, we use NCCL [21]. 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 [14].

Chapter 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 [33]. The Llama 2 LLMs are the most popular open-source models based on an online LLM leaderboard [25], 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, following convention, 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 [30], which is the state-of-the-art approach to model parallelism used in modern LLM inference engines such as vLLM [18] and TGI [13].

7.2 Results

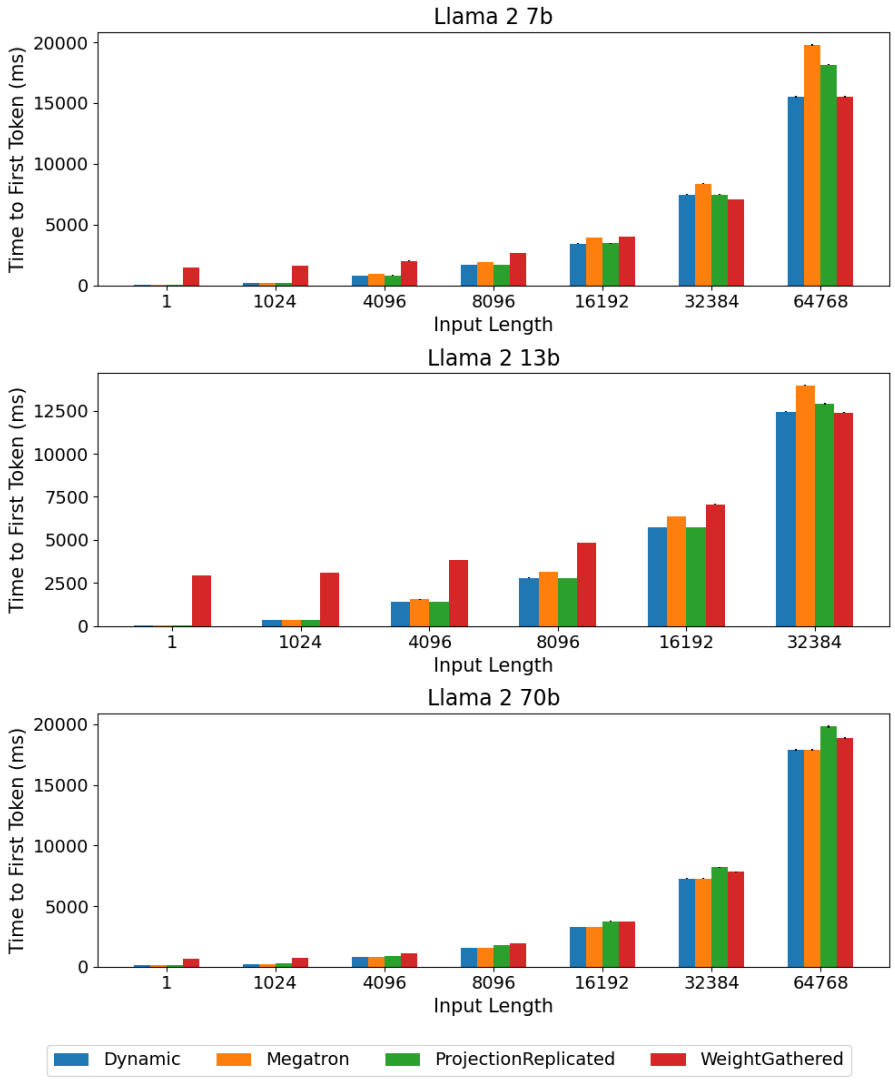

Figure 7.1: Time to first token for Llama 2 models. 95% confidence interval computed over 10 total iterations.

Time to first token Figure 7.1 shows the time to first token for varying prompt lengths on Llama 2 [33] 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 rather than communication 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 appears to become the bottleneck when the input length is greater than 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 theoretical threshold due to overlapping of computation and communication in WeightGathered in our library implementation 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 with minimal switching overhead. 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.

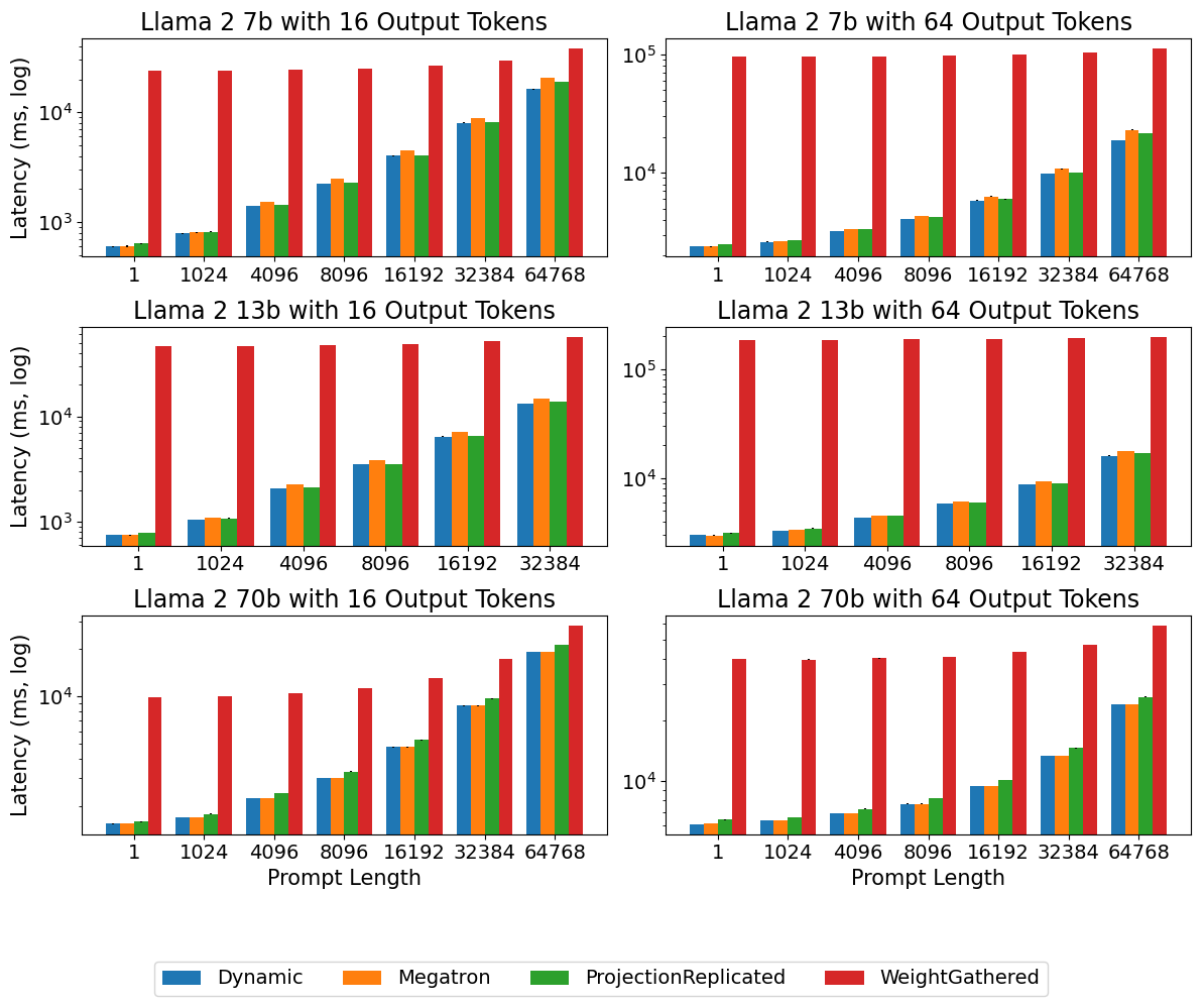

Figure 7.2: Latency for Llama 2 model generation on short and long outputs (log scale). 95% confidence interval computed over 10 total iterations.

Latency Figure 7.2 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 (§4.3), 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 all 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 static Megatron-based 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 wide 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).

Chapter 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.

Chapter 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 [20] 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 [5] 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 [22] 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 preprocessing and post-processing, as well as other optimizations such such as FlashAttention [8] and the 8-bit floating point data type. DeepSpeed Inference [1] 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 [36] 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 [7, 28, 17], quantization techniques to reduce the memory required by LLMs [9, 37, 39], as well as model distillation [29, 31], where smaller specialized models are trained using larger models. FlashAttention [8] 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 [18] significantly reduces memory fragmentation and duplication in the key-value-cache for Trans-

former 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 [6] 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 [32] 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 [16] 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 [3] 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.

Chapter 10

Conclusion

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 developed a system to perform 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 [33] as compared to the de-facto approach introduced by Megatron-LM [30]. In particular, we find that dynamic partitioning is most effective on smaller LLMs and GPUs with lower interconnect bandwidth and compute.

10.1 Future Work

There are several directions for future work. Our current analysis of different partitioning strategies (§4.4) calculates a theoretical value for the weight FLOPs and communication volume. 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, the extent of our search space in PartitionSearch is limited by the set of the operations that we define, meaning that we might miss potentially better partitioning strategies due to our design. For example, we make the decision to treat self-attention as a single operation for simplicity - however, this may prevent us from discovering new partitioning strategies for self-attention that are perform better than existing strategies. Adding new collective communication operations may also expand the space of partitioning

strategies. Therefore, we hope to further expand the search space with the goal of discovering new Pareto-optimal strategies.

Next, 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.

This work currently 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 new model architectures might lead to further opportunities for optimizing inference via new Pareto-optimal strategies for dynamic partitioning.

Bibliography

[1] Reza Yazdani Aminabadi et al. *DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale*. arXiv:2207.00032 [cs]. June 2022. DOI: 10.48550/arXiv.2207.00032. URL: http://arxiv.org/abs/2207.00032 (visited on 12/11/2023).

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

[3] Sanjith Athlur et al. *Varuna: Scalable, Low-cost Training of Massive Deep Learning Models*. arXiv:2111.04007 [cs]. Nov. 2021. DOI: 10.48550/arXiv.2111.04007. URL: http://arxiv.org/abs/2111.04007 (visited on 12/11/2023).

[4] Tom B. Brown et al. *Language Models are Few-Shot Learners*. arXiv:2005.14165 [cs]. July 2020. DOI: 10.48550/arXiv.2005.14165. URL: http://arxiv.org/abs/2005.14165 (visited on 12/10/2023).

[5] ByteDance. *EffectiveTransformer*. https://github.com/bytedance/effective$_t ransformer$.

[6] Zhenkun Cai et al. "TensorOpt: Exploring the Tradeoffs in Distributed DNN Training with Auto-Parallelism". In: *IEEE Transactions on Parallel and Distributed Systems* 33.8 (Aug. 2022). arXiv:2004.10856 [cs, stat], pp. 1967–1981. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2021.3132413. URL: http://arxiv.org/abs/2004.10856 (visited on 12/11/2023).

[7] Krzysztof Choromanski et al. *Rethinking Attention with Performers*. arXiv:2009.14794 [cs, stat]. Nov. 2022. DOI: 10.48550/arXiv.2009.14794. URL: http://arxiv.org/abs/2009.14794 (visited on 12/11/2023).

[8] Tri Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. arXiv:2205.14135 [cs]. June 2022. DOI: 10.48550/arXiv.2205.14135. URL: http://arxiv.org/abs/2205.14135 (visited on 12/11/2023).

[9] Tim Dettmers et al. *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. arXiv:2208.07339 [cs]. Nov. 2022. DOI: 10.48550/arXiv.2208.07339. URL: http://arxiv.org/abs/2208.07339 (visited on 12/11/2023).

[10] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv:1810.04805 [cs]. May 2019. DOI: 10.48550/arXiv.1810.04805. URL: http://arxiv.org/abs/1810.04805 (visited on 12/10/2023).

[11] William Fedus, Barret Zoph, and Noam Shazeer. *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. arXiv:2101.03961 [cs]. June 2022. DOI: 10.48550/arXiv.2101.03961. URL: http://arxiv.org/abs/2101.03961 (visited on 12/10/2023).

[12] GitHub. *GitHub Copilot*. https://github.com/features/copilot. 2022.

[13] HuggingFace. *Text Generation Interface*. https://github.com/huggingface/text-generation-inference.

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

[15] Abhinav Jangda et al. *Breaking the Computation and Communication Abstraction Barrier in Distributed Machine Learning Workloads*. arXiv:2105.05720 [cs]. Mar. 2022. DOI: 10.48550/arXiv.2105.05720. URL: http://arxiv.org/abs/2105.05720 (visited on 12/10/2023).

[16] Zhihao Jia, Matei Zaharia, and Alex Aiken. *Beyond Data and Model Parallelism for Deep Neural Networks*. arXiv:1807.05358 [cs]. July 2018. DOI: 10.48550/arXiv.1807.05358. URL: http://arxiv.org/abs/1807.05358 (visited on 12/11/2023).

[17] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. *Reformer: The Efficient Transformer*. arXiv:2001.04451 [cs, stat]. Feb. 2020. DOI: 10.48550/arXiv.2001.04451. URL: http://arxiv.org/abs/2001.04451 (visited on 12/11/2023).

[18] Woosuk Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. arXiv:2309.06180 [cs]. Sept. 2023. DOI: 10.48550/arXiv.2309.06180. URL: http://arxiv.org/abs/2309.06180 (visited on 12/10/2023).

[19] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. arXiv:2005.11401 [cs]. Apr. 2021. DOI: 10.48550/arXiv.2005.11401. URL: http://arxiv.org/abs/2005.11401 (visited on 12/10/2023).

[20] NVIDIA. *FasterTransformer*. https://github.com/NVIDIA/ FasterTransformer. 2023.

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

[22] NVIDIA. *TensorRT LLM*. 2023.

[23] NVIDIA. *TensorRT-LLLM*. https://github.com/NVIDIA/TensorRT-LLM.

[24] OpenAI. *ChatGPT*. en-US. https://openai.com/blog/chatgpt. 2022. (Visited on 12/10/2023).

[25] LMSYS Org. *Chatbot Arena*. https://chat.lmsys.org/.

[26] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: https://papers.nips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html (visited on 12/10/2023).

[27] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: 2019. URL: https://www.semanticscholar.org/paper/Language-Models-are-Unsupervised-Multitask-Learners-Radford-Wu/9405cc0d6169988371b2755e573cc28650d14dfe (visited on 12/10/2023).

[28] Aurko Roy et al. *Efficient Content-Based Sparse Attention with Routing Transformers*. arXiv:2003.05997 [cs, eess, stat]. Oct. 2020. DOI: 10.48550/arXiv.2003.05997. URL: http://arxiv.org/abs/2003.05997 (visited on 12/11/2023).

[29] Victor Sanh et al. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. arXiv:1910.01108 [cs]. Feb. 2020. DOI: 10.48550/arXiv.1910.01108. URL: http://arxiv.org/abs/1910.01108 (visited on 12/11/2023).

[30] Mohammad Shoeybi et al. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. arXiv:1909.08053 [cs]. Mar. 2020. DOI: 10.48550/arXiv.1909.08053. URL: http://arxiv.org/abs/1909.08053 (visited on 12/10/2023).

[31] Siqi Sun et al. *Patient Knowledge Distillation for BERT Model Compression*. arXiv:1908.09355 [cs]. Aug. 2019. DOI: 10.48550/arXiv.1908.09355. URL: http://arxiv.org/abs/1908.09355 (visited on 12/11/2023).

[32] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. "Piper: Multidimensional Planner for DNN Parallelization". In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 24829–24840. URL: https://proceedings.neurips.cc/paper/2021/hash/d01eeca8b24321cd2fe89dd85b9beb51-Abstract.html (visited on 12/11/2023).

[33] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. arXiv:2307.09288 [cs]. July 2023. DOI: 10.48550/arXiv.2307.09288. URL: http://arxiv.org/abs/2307.09288 (visited on 12/10/2023).

[34] Ashish Vaswani et al. *Attention Is All You Need*. arXiv:1706.03762 [cs]. Aug. 2023. DOI: 10.48550/arXiv.1706.03762. URL: http://arxiv.org/abs/1706.03762 (visited on 12/11/2023).

[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*. arXiv:1807.08887 [cs]. Mar. 2019, pp. 1–17. DOI: 10.1145/3302424.3303953. URL: http://arxiv.org/abs/1807.08887 (visited on 12/11/2023).

[36] Yuanzhong Xu et al. *GSPMD: General and Scalable Parallelization for ML Computation Graphs*. arXiv:2105.04663 [cs]. Dec. 2021. DOI: 10.48550/arXiv.2105.04663. URL: http://arxiv.org/abs/2105.04663 (visited on 12/11/2023).

[37] Ofir Zafrir et al. "Q8BERT: Quantized 8Bit BERT". In: *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*. arXiv:1910.06188 [cs]. Dec. 2019, pp. 36–39. DOI: 10.1109/EMC2-NIPS53020.2019.00016. URL: http://arxiv.org/abs/1910.06188 (visited on 12/11/2023).

[38] Susan Zhang et al. *OPT: Open Pre-trained Transformer Language Models*. arXiv:2205.01068 [cs]. June 2022. DOI: 10.48550/arXiv.2205.01068. URL: http://arxiv.org/abs/2205.01068 (visited on 12/10/2023).

[39] Yijia Zhang et al. *Integer or Floating Point? New Outlooks for Low-Bit Quantization on Large Language Models*. arXiv:2305.12356 [cs]. May 2023. DOI: 10.48550/arXiv.2305.12356. URL: http://arxiv.org/abs/2305.12356 (visited on 12/11/2023).

[40] Lianmin Zheng et al. *Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning*. arXiv:2201.12023 [cs]. June 2022. DOI: 10.48550/arXiv.2201.12023. URL: http://arxiv.org/abs/2201.12023 (visited on 12/11/2023).

Appendix A

Raw Data

A.1 Time to First Token

Input Length	Dynamic	Megatron	ProjectionReplicated	WeightGathered
1	39.8125	39.4615	41.868	1491.6495
1024	210.797	236.253	210.2415	1612.1665
4096	834.992	949.788	838.32	2031.913
8096	1680.804	1912.5425	1681.312	2669.958
16192	3452.317	3910.1625	3457.254	3993.292
32384	7489.2195	8381.168	7485.6335	7080.7215
64768	15533.145	19795.499	18170.0725	15503.617

Table A.1: Mean time to first token (ms) for Llama 2 7B generation using 4 L4 GPUs on varying input lengths. Results are computed over 10 total iterations.

Input Length	Dynamic	Megatron	ProjectionReplicated	WeightGathered
1	50.017	49.96	51.6165	2915.8775
1024	336.1615	371.455	335.1855	3094.4945
4096	1387.8365	1540.623	1388.0525	3827.5665
8096	2801.636	3129.5115	2799.9325	4851.285
16192	5734.265	6356.4685	5731.1065	7064.468
32384	12439.6755	13975.2355	12890.9215	12380.7165

Table A.2: Mean time to first token (ms) for Llama 2 13B generation using 4 L4 GPUs on varying input lengths. Results are computed over 10 total iterations.

Input Length	Dynamic	Megatron	ProjectionReplicated	WeightGathered
1	103.395	104.6745	107.9335	623.489
1024	228.499	228.499	258.539	714.442
4096	790.8735	790.8735	905.532	1110.536
8096	1567.402	1567.402	1798.3365	1948.2325
16192	3289.319	3289.319	3764.8925	3743.2965
32384	7286.418	7286.418	8202.5445	7824.277
64768	17882.6555	17882.6555	19821.264	18844.9195

Table A.3: Mean time to first token (ms) for Llama 2 70B generation using 4 A100 GPUs on varying input lengths. Results are computed over 10 total iterations.

A.2 Latency

Input Length	Dynamic	Megatron	ProjectionReplicated	WeightGathered
1	601.018	605.1045	637.7185	23915.922
1024	787.017	808.8685	817.848	24057.652
4096	1410.1405	1525.2325	1435.382	24384.5525
8096	2253.943	2487.0835	2286.748	25226.935
16192	4023.1075	4485.6245	4052.88	26483.4375
32384	8052.1335	8941.0005	8099.2305	29736.8855
64768	16229.7495	20525.0665	18986.3305	38411.0555

Table A.4: Mean latency (ms) for Llama 2 7B generation using 4 L4 GPUs on varying input lengths with output length 16. Results are computed over 10 total iterations.

Input Length	Dynamic	Megatron	ProjectionReplicated	WeightGathered
1	752.487	748.815	784.293	46731.9205
1024	1049.593	1090.699	1082.3695	46665.2615
4096	2092.0305	2253.1955	2132.351	47769.972
8096	3516.1085	3840.5545	3551.033	48713.033
16192	6448.107	7079.9645	6499.876	51310.8015
32384	13273.346	14865.2685	13876.233	56668.946

Table A.5: Mean latency (ms) for Llama 2 13B generation with 4 L4 GPUs on varying prompt lengths with output length 16. Results are computed over 10 total iterations.

Input Length	Dynamic	Megatron	ProjectionReplicated	WeightGathered
1	1537.3495	1548.138	1584.501	9930.974
1024	1690.1795	1690.1795	1771.6765	10051.501
4096	2242.8395	2242.8395	2415.515	10486.6895
8096	3031.56	3031.56	3310.555	11348.0745
16192	4744.893	4744.893	5261.584	13157.1655
32384	8723.573	8723.573	9701.099	17269.2285
64768	19304.882	19304.882	21278.1705	28264.1985

Table A.6: Mean latency (ms) for Llama 2 70B with 4 A100 GPUs on varying input lengths with output length 16. Results are computed over 10 total iterations.

Input Length	Dynamic	Megatron	ProjectionReplicated	WeightGathered
1	2397.1395	2378.9705	2506.775	95577.469
1024	2614.25	2649.693	2722.746	95615.469
4096	3232.572	3354.0675	3373.04	95738.8595
8096	4079.557	4323.9655	4214.7865	96850.989
16192	5846.15	6316.9285	5967.3465	98457.1095
32384	9882.714	10765.9815	10035.116	102219.6265
64768	18727.5475	23000.609	21598.289	112136.7735

Table A.7: Mean latency (ms) for Llama 2 7B generation with 4 L4 GPUs on varying input lengths and output length 64. Results are computed over 10 total iterations.

Input Length	Dynamic	Megatron	ProjectionReplicated	WeightGathered
1	3017.577	2982.553	3117.518	186537.522
1024	3302.9155	3354.8175	3465.633	186361.053
4096	4364.9895	4529.4995	4508.622	187721.6065
8096	5807.213	6100.304	5946.9375	189355.1315
16192	8749.101	9379.7135	8917.9215	192468.647
32384	16067.8065	17612.402	16895.4365	198486.297

Table A.8: Mean latency (ms) for Llama 2 13B generation with 4 L4 GPUs on varying input lengths and output length 64. Results are computed over 10 total iterations.

Input Length	Dynamic	Megatron	ProjectionReplicated	WeightGathered
1	6092.361	6143.1665	6410.4785	40099.8135
1024	6373.007	6373.007	6603.2165	39965.5685
4096	6903.916	6903.916	7255.7595	40460.074
8096	7689.4505	7689.4505	8193.087	41279.3945
16192	9408.911	9408.911	10098.6335	43431.0275
32384	13349.288	13349.288	14505.7775	47423.058
64768	23947.8545	23947.8545	26079.156	58811.546

Table A.9: Mean latency (ms) for Llama 2 70B generation with 4 A100 GPUs on varying input lengths and output length 64. Results are computed over 10 total iterations.