Faster Distributed LLM Inference with Dynamic Partitioning

Isaac Ong, Woosuk Kwon

Introduction

Recent advances in machine learning have enabled an exponential increase in model size of large language models, from BERT with 340 million parameters to GPT-2 with 1.5 billion parameters, and GPT-3 with 175 billion parameters. The emergence of these large language models (LLMs) have led to myriad new use cases such as chatbots like ChatGPT and coding assistants like GitHub Copilot. However, the size of these LLMs poses challenges for serving them as they exceed the memory limits of modern processors. Therefore, model inference has to be parallelized across GPUs. The performance characteristics of distributed LLM inference are highly dependent on the input length, which can vary from 1 token to tens of thousands of tokens. Moreover, the optimal partitioning scheme for tensor parallelism may also differ based on the input length. However, existing LLM inference engines do not account for this and apply a static partitioning scheme regardless of input size. Therefore, we propose dynamic partitioning, an approach to tensor parallelism for distributed LLM inference that dynamically switches between different optimal partitioning schemes based on the input size.

Background

LLMs are based on the autoregressive transformer architecture, which generates new tokens one at a time based on the *input prompt* tokens and the previously-generated *output* tokens. Figure 1 shows a diagram of the transformer architecture. A transformer layer consists of a self-attention block, followed by a two-layer multi-layer perceptron (MLP) as shown in the figure.

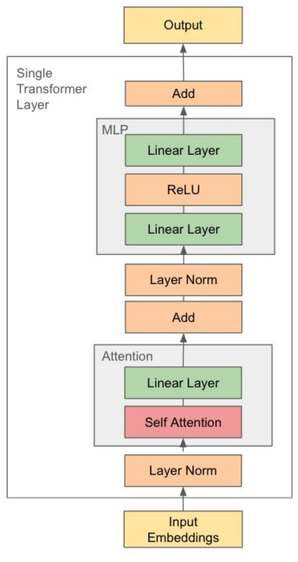


Figure 1. Transformer Architecture

There are two main approaches to model parallelism:

1. Pipeline parallelism performs operations on one GPU before the outputs of these operations are passed onto the next GPU where a new set of operations are performed, and so on.
2. Tensor parallelism, which is the focus of this work, partitions tensor operations across multiple GPUs so as to either increase the speed of computation or to reduce the amount of memory required on each GPU.

The current state of the art approach to tensor parallelism is Megatron-LM:

- Self attention block is partitioned across attention heads in a column parallel manner so that the matrix operations corresponding to every attention head are done locally. The output projection linear layer is partitioned in a row parallel manner so that it takes the output of the attention operation directly. Finally, an all-reduce is required to synchronize the tensors before moving to the MLP layer.
- For the MLP layer, the first matrix operation is partitioned in a column parallel manner while the second matrix operation is partitioned in a row parallel manner so that it can take the output of the first matrix operation without any synchronization operation. Finally, an all-reduce is again required to synchronize the resulting tensors.

Partitioning Scheme Search

To facilitate the identification of alternative optimal partitioning schemes, we developed a program that exhaustively searches all viable distributed partitioning schemes for LLM architectures to determine the Pareto frontier for partitioning schemes based on FLOPs, communication overhead, and weights memory.

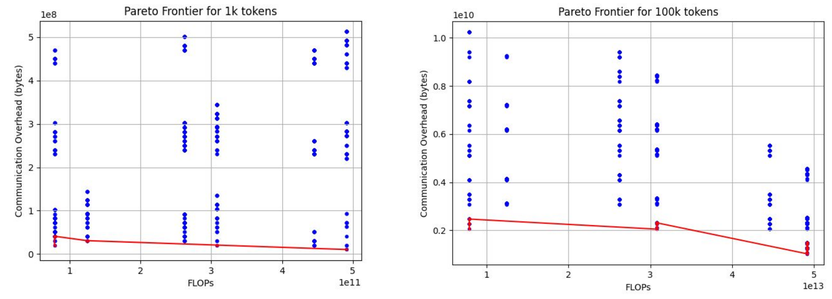
We formalize the state of parallelized tensors into 4 types:

1. Replicated tensor (X_R): Fully replicated across all GPUs
2. Column-sliced tensor (X_{CS}): Sliced along the column dimension across GPUs
3. Row-sliced tensor (X_{RS}): Sliced along the row dimension across GPUs
4. Local tensor (X_L): Same size as the full tensor, but does not contain the full values

Next, we formalize the following rules for operations on such tensors:

1. $A_{CS} @ B_{RS} = AB_L$
2. $A_R @ B_{CS} = AB_{CS}$
3. $A_{RS} @ B_R = AB_{RS}$
4. $A_R @ B_R = AB_R$
5. $\text{AllGather}(A_{CS} \text{ or } A_{RS}) = A_R$
6. $\text{ReduceScatter}(A_L) = A_{CS} \text{ or } A_{RS}$
7. $\text{Split}(A_R) = A_{CS} \text{ or } A_{RS}$
8. From 5 and 6: $\text{AllReduce}(A_L) = A_R$

Using the above axioms, we can run the search across an entire transformer layer and identify the Pareto frontier for partitioning schemes, as well as the symbolic FLOPs, communication overhead, and weights memory for each partitioning scheme.



Pareto Optimal Partitioning Schemes

Based on the search above, we identified 3 viable Pareto optimal partitioning schemes for distributed LLM inference. Each of these partitioning schemes have different characteristics depending on the model and input length.

1. **Megatron Attention / Megatron MLP:** This is the same partitioning scheme used in Megatron-LM. It requires the least FLOPs out of the identified partitioning schemes, and is therefore optimal for small sequence lengths when compute, and not communication overhead, is the bottleneck.
2. **Output Projection Replicated Attention / Megatron MLP:** In output projection replicated attention, an all-gather operation is performed after the self attention operation to obtain the full tensor, and the output projection linear layer is fully replicated. An all-reduce is no longer required after the attention block, reducing the communication overhead. This partitioning scheme is more optimal than the first scheme when communication is the bottleneck. This is often the case in the prefill phase of LLM generation, when the entire list of *input prompt* tokens have to be processed at once.
3. **Megatron Attention / Weight-gathered MLP:** In weight-gathered MLP, the weights in the MLP layer are gathered before each matrix operation, and discarded after. This makes the communication overhead for the MLP layer independent of the sequence length, unlike the first two schemes. Therefore, this partitioning scheme is optimal when communication overhead is the bottleneck, and when the sequence length is long enough such that the communication overhead of other schemes exceeds the communication overhead of this scheme.

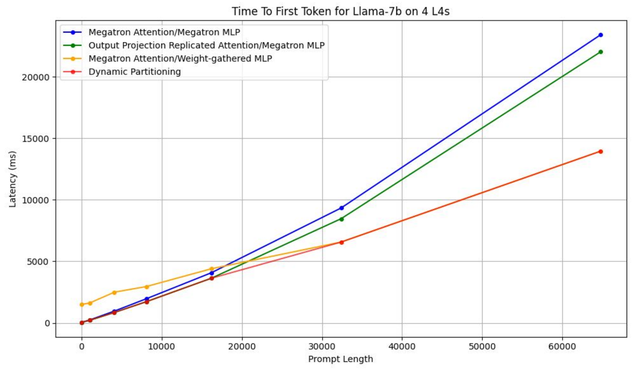
Design & Implementation

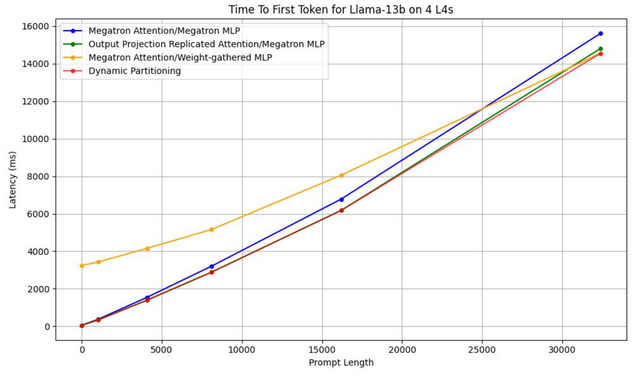
With the identified partitioning schemes, it is possible to dynamically switch between them at inference time because they share the same weight layout. Therefore, we can load the model weights at initialization time regardless of the chosen partitioning scheme at inference time.

To determine the threshold for which to switch between different partitioning schemes, we calculate the theoretical prompt length at which each scheme becomes optimal, taking into account both the communication overhead and FLOPs of each scheme. However, the theoretical threshold is an overestimate as it does not account for GPU characteristics as well as the fact that certain schemes, such as weight-gathered MLP, can more easily overlap computation and communication on the GPU.

We developed a LLM inference library using PyTorch in Python that supports Llama-based models and added support for dynamic partitioning using the identified partitioning schemes. Moreover, we took advantage of torch.compile to accelerate inference speed by JIT compiling the PyTorch code into custom CUDA kernels.

Evaluation





Using dynamic partitioning, we observe reductions of **10% - 40%** in the time to first token for varying prompt lengths as compared to the scheme used by Megatron-LM in the inference library we developed. These benchmarks were executed on 4 L4 GPUs using the Llama-7b and Llama-13b models. This indicates marked improvements in the speed of distributed LLM inference using dynamic partitioning as compared to a static partitioning scheme.