Parallelizing Mixture of Experts Transformers in FlexFlow for Low-Latency Inference

15-618 Final Project Proposal

https://hugolatendresse.github.io/15618-final-project/

Hugo Latendresse (hlatendr@andrew.cmu.edu)
Matthew Katz (mkatz2@andrew.cmu.edu)

Carnegie Mellon University

Summary

We will add support for Mixture of Experts (MoE) Transformer models to the FlexFlow Serve framework. In particular, we will add support for models that use MoE in their MLP layers. Once we are able to successfully run inference with the model on FlexFlow, we will benchmark its per-token latency and throughput on a node with four V100 GPUs against a traditional service setting.

Background

Mixture of Experts (MoE) is a machine learning technique that has surged in popularity as a method to keep latency low for training and serving increasingly large generative DL models. This technique is commonly used for LLMs, which our project will focus on. MoE replaces a functional unit, or layer, of a model with multiple “expert” networks that are each significantly smaller than the original layer. As a result of the training process, each expert specializes in processing a (not necessarily disjoint) subset of the input space, so a given input need only be processed by its corresponding expert(s). A router or gate network is placed at the beginning of the layer and determines which expert(s) to send the input to. In sparse MoE, only a proper subset of the whole model is used for forward passes. This reduces computational requirements and can help improve the per-token generation latency.

MoE is commonly applied to MLP layers of transformers, with the notable example of the Mixtral model family. As most layers of modern LLM architectures, they require significant amounts of computation and memory accesses. Moreover, modern LLMs do not fit on a single GPU. Therefore, parallelization is necessary to serve them. The question is not "whether" to parallelize, but "how" to parallelize, with the goal of achieving the best possible latency and throughput on a given set of hardware resources. Different parallelization strategies will lead to different data movement, amount of redundant computation, etc., leading to different performance.

FlexFlow Serve is a project led by Prof Jia and his research team. It is an open-source compiler and distributed system for highly optimized LLM serving. It utilizes standard forms of multi-GPU parallelism, as well as speculative decoding to accelerate inference. For our research project, we will focus on incremental decoding: speculative decoding is not in scope.

FlexFlow does not yet support serving LLMs with MoE architectures. We propose selecting a baseline MoE-based LLM, implementing what is needed in FlexFlow Serve to support it, and benchmarking its inference performance (per-token latency and throughput) against a traditional, non-accelerated service setting. This will entail recreating the MoE model’s architecture and forward pass in FlexFlow, writing some CUDA kernels to parallelize the MoE router mechanism and MoE MLP layers, and evaluating performance.

The Challenge

The first challenge is to onboard ourselves onto the FlexFlow project. The repository is quite large and nuanced. We do not yet know what components of the architecture we choose are already implemented, and where exactly our starting point will be.

The workload consists of an entire transformer model, though our focus will be on the MoE-MLP layers. Each layer of the model is of course dependent on the previous layer (and not vice-versa, since we are focusing on inference). Within an MoE-MLP layer, the "experts" (FNN models) are dependent on the router.

The core challenge of the parallelization work we will do is figuring out the best way to parallelize the MoE layer. Inferring with a model that cannot fit on a single GPU necessitates expensive communication, increasing the communication-to-computation ratio. There are many possible ways to decompose and schedule the calculation on multiple workers, and experimentation is needed to find the strategy that minimizes communication and latency.

Pure data parallelism will not be possible since the entire model does not fit on a single GPU. We can think of two main strategies to parallelize MoE layers. We call them “inter-expert parallelism” and “intra-expert parallelism”. Inter-expert parallelism means allocating the different experts to different GPUs. It’s similar to model parallelism, except that each sequence (each token) is routed to only one expert.

Intra-expert parallelism means processing the activations within a single expert concurrently. It is similar to the concept of data parallelism, but at a finer granularity (we split across activations within a sample instead of across samples within a batch).

Temporal locality is high for the weights, but very limited for the activations. We expect spatial locality to be high within experts, but possibly lower across experts.

Divergent execution is typical to neural networks due to the non-linear activation functions. In MoE models, an additional layer of divergence is introduced due to the routing to experts. That can introduce load imbalance, as some experts may receive more inputs than others.

Resources

We will be working off of the FlexFlow codebase (https://github.com/flexflow/FlexFlow). We will be writing a combination of CUDA kernels and C++ code.

We plan to use one compute node in Pittsburgh's Supercomputer (Bridges-2) GPU cluster. Each node has four V100 GPUs with 32GB of VRAM, for a total of 128GB for VRAM. We will tentatively base our project on Mixtral 8x7B Instruct with half-precision, which requires 90GB of VRAM in total.

Goals and Deliverables

Plan to Achieve

- [ ] Write a CUDA kernel(s) implementing the key components of our baseline MoE model, namely a MoE MLP layer consisting of routers (gate functional units) and experts (FFNs)
- [ ] Complete the implementation of a full MoE transformer by incorporating our work with existing FlexFlow CUDA kernels for the traditional parts of MoE transformers (self-attention, etc.)
- [ ] Write other CUDA and C++ code to make our baseline model work with the FlexFlow API (inference only).
- [ ] Successfully serve an MoE model with FlexFlow.
- [ ] Benchmark per-token latency and throughput of our baseline model using FlexFlow vs per-token latency of the Hugging Face Transformers package.
- [ ] Create a poster explaining how FlexFlow works, describing the architecture of our chosen MoE model, and showing our process in parallelizing it.
- [ ] Show the speed at which we can output tokens versus Hugging Face Transformers, through a video or in real time.

We will prioritize an inter-expert parallelism strategy (see description in the Challenge section). As time permits, we will try intra-expert parallelism and other strategies.

Hope to Achieve

- [ ] Beat Hugging Face Transformers in terms of per-token latency. We only "hope" to achieve that because that library already makes use of parallelization and is probably optimized to some extent, and because we are not using speculative decoding.
- [ ] Profile our implementation and identify the bottlenecks.
- [ ] Iterate on our implementation to achieve better speedups.
- [ ] Benchmark our implementation against other accelerators like vLLM and FasterTransformer.

Platform Choice

FlexFlow is implemented in C++ and CUDA, so those are the two languages we will use.
We will be working from Linux machines since it is the only OS officially supported by FlexFlow.

Schedule

Week	Activity
November 11 - November 17	● Finalize the project proposal ● Familiarize ourselves with the FlexFlow repo ● Walk through the FlexFlow developer guide ● Meet with a member of Prof Jia's team to onboard onto the project
November 18 - November 24	● Confirm the choice of baseline model based on the resources available on the PSC ● Finalize our initial strategy to parallelize the baseline model (probably "inter-experts"). ● Write a CUDA kernel(s) implementing an MoE MLP layer ● Complete the implementation of a Full MoE Transformer ● Write other C++ and CUDA code to integrate our baseline with FlexFlow ● Complete Milestone report
November 25 - December 1	● Benchmark our implementation with non-accelerated Hugging Face implementation ● Try a different parallelism strategy ("intra-expert" or other).
December 2 - December 8	● Continue iterating on our implementation until desired speedup is achieved ● Complete poster
December 9 - December 15	● Complete final report