CMSC416: Introduction to Parallel Computing

Topic: Deep Learning in Parallel
Date: March 02, 2024

Neural Network

A neural network is a family of parameterized approximator functions. Neural Networks (NN) work well with high-dimensional data such as Large Language Models.

A NN takes input x (such as images) and parameters theta. The NN starts with random parameters and gradually learns them. The computation is organized in a sequence of layers. In slide 4, we see an example of a NN with 3 layers where each layer takes its own parameter theta_i for i=1,2,3 (the parameter theta for the NN is a vector with element theta_i corresponding to the ith layer). Each layer takes an input from the previous layer, does a computation (such as Matrix Multiple) on the data, and outputs into the next layer.

Terminology

Learning/training is the task of selecting better parameters (weights) that lead to a more accurate output of the NN.

Loss is a scalar proxy that when mathematically minimized, leads to higher accuracy of the NN. I.e. the process of learning/training is minimizing the loss function to obtain parameters.

Gradient Descent is the method used to minimize the loss function. Gradient Descent is usually done iteratively in batches, where each batch represents a subset of the dataset.

An epoch constitutes one pass over all batches.

What is parallel deep learning?

The goal of parallel deep learning is to train a NN on multiple GPUs.

In the last decade, we have seen that the sizes of NN have grown exponentially (as seen in slide 6). It seems that the more parameters there are in a NN, the better it can perform (such as LLN). So naturally this leads to a growth in parameters in state-of-the-art NN.

For example, LLaMA has around 70 billion parameters. How long would this take to train on a single A11 GPU on Zaratan? 172 Years! Which costs 40,000 dollars. Thus there is a need to speed up training by using parallel computation across 1000s of GPUs.

Parallel/Distributed Training:

There are many batches, epochs, and layers that can be exploited in parallel algorithms across GPUs.

We start with the simplest method, data parallelism.

Data Parallelism

Data parallelism aims to divide training data among the workers (GPUs). On slide 9, we see an example with two GPUs that split the total batch into two sub-batches. Each GPU has a copy of the entire NN in Data Parallelism. This leads to a ridiculous parallel algorithm where each GPU can compute on its sub-batch without communication of the other GPU. After the computation of the gradients, the workers use an all-reduce (an average operation) to synchronize the gradients. Each GPU does gradient descent locally.

The Pros and Cons of Data Parallelism:

The pros are that it is simple and embarrassingly parallel (no communication between GPUs). A lot of libraries like Pytorch have some implication of data parallelism. But, data parallelism is restricted to NNs that can fit on a single GPU as each GPU needs its own complete copy of the NN. This motivates other methods as state-of-the-art models are too large to fit onto a single GPU.

To address this, we think about Inter-Layer Parallelism

Inter-Layer Parallelism

Inter-Layer Parallelism distributes the layers of the NN to different procs/GPUs. Unlike Data Parallelism, now each GPU only works on a fraction of the model allowing models to exceed the memory of a single GPU. On the other hand, inter-layer parallelism is no longer embarrassing parallel as there is point-to-point communication between procs/GPUs. The communication arises from the dependency of a layer on the output of the previous layer. For example, layer one needs to pass the activations to layer 2, which will pass to layer 3 and so at some point, one GPU needs to pass information to another GPU. This also shows a sequential dependency between the layers (layer 1 -> layer 2 -> … -> layer n, and layer 2 can only compute once it has data from 1 and so on). So this approach is not fully parallel which defeats the point of using multiple GPUs.

The solution is pipelining in inter-layer parallelism.

Pipelining in inter-layer Parallelism

Pipelining takes advantage of breaking the batch into multiple shards (microbatches) and processing them in a pipelined fashion. On slide 12, we see an example of pipelining with a batch broken into 8 shards. We see how the shards can be processed in parallel. I.e., GPU 0 starts with shard 0 on layer 0 and then passes to GPU 1. While GPU 1 works on shard 0 on layer 1, GPU 0 concurrently begins working on shard 1 on layer 0. This allows the GPUs to work in parallel.

To summarize Inter-layer naively, has sequential dependencies so break the batch into shards to create a pipeline. (Notice that NN needs the pipeline to go both ways since NN needs to do a backpass)

Intra-Layer Parallelism

Intra-Layer Parallelism divides each layer between GPUs. Often in machine learning, the layers are computing large matrix multiples, so a lot of intra-layer parallelism takes advantage of parallel matrix multiplication. Often when you read papers on Intra-layer parallelism, you see parallel matrix multiplication methods, such as Cannons 2D and Agarwal's 3D .

Hybrid Parallelism

Typically state of the art methods use a combination of the methods described above. A common hybrid parallelism technique is 3D parallelism which combines all three of the methods. 3D parallelism is currently thought to be the fastest/best for NN. 3D parallelism is used on LLN. Some examples of these frameworks are DDP, FSDP, ZeRO, Megatron-LM (MEgatron-LM is thought to be the fastest for large data).

How are these combined? A hand-wavey explanation is as follows. Let's say the GPUs are organized in a 2d array. The rows of the array can use one method while the columns can use another. If the GPUs are organized in 3D, each dimension can be used to perform a different method of parallelism. We can continue to increase the dimensions and use each dimension for different modes of parallelism. See the Megatron-LM paper which exhibits this behavior nicely.

Parallel Deep Learning @ PSSG

GPUs are extremely good at parallel matrix multiplication, so in practice, computation is not a problem. A problem arises in communication between GPUs/procs, especially for large models like LLN. So the primary task becomes how to minimize the communication overhead. On slide 15, we see the weak scaling on AxoNN which is an in-house model (open source) on Frontier.

Another motivation in parallel deep learning is creating a user-friendly application. That is, designing a parallel process that can be implemented by others with minimal changes to their code. In a perfect world, a user would only have to add one line of code to implement a parallel design.