Week 6 Reading Summaries: ML Parallelism + Megatron

Amogh Patankar
University of California, San Diego
3869 Miramar Street Box #3037, La Jolla, CA- 92092
apatankar@ucsd.edu

Abstract

With these readings, we dive deep into distributed training and model parallelism, two paradigms of ML optimization. Within distributed training, there are techniques like DeepSpeed and FSDP. Megatron-LM is a framework that trains multi-billion parameter language models using model parallelism, and they implement it with a few communication operations in PyTorch.

1 ML Parallelism Blog

1.1 Distributed Training Basics

Distributred training deals with maximizing throughput, while using various techniques to decrease model size and memory utilization. Primarily, data, model, and tensor parallelism are used, but a combination of these strategies allows for vast improvements in throughput.

1.2 ZeRO-powered Data Parallelism

Zero Redundancy Optimizer is a form of data parallelism that improves memory efficiency, the idea being that we have data parallelism and residual memory (ZeRO-DP and ZeRO-R). The DeepSpeed team deals with ZeRO-Offland/Infinity and ZeRO++.

1.2.1 Stage 1

Stage 1 of ZeRO consists of only partitioning/sharding across GPU workers, with the same weights and gradients replicated across workers.

1.2.2 Stage 2

In this stage, both the optimizer and gradients are partitioned across workers, allowing each worker to update the partition of the optimizer, essentially performing a reduce-scatter.

1.2.3 Stage 3

Here, each layer of the model is horizontally sliced; this implies that a worker stores partial weight tensors. Different GPU workers exchange the parts they have, and compute gradients + activations. That is, there's no limitation by per-GPU vRAM.

1.2.4 ZeRO-R

This is simply managing memory fragmentation, by reducing activation memory footprint, and it improves data buffers.

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

1.2.5 ZeRO-Offload

ZeRO-Offload is utilized to offload optimizer and computation from GPUs to CPU, however, we run into inefficient computation done on CPU, as opposed to GPU. With ZeRO, only computations under *O(MB)* offload to CPU, and heavy computations are done on GPU.

1.2.6 ZeRO-Infinity

This improvement offloads to disk and has CPU offloading changes, allowing model fitting in excess of 10T parameters for a DGX-2 node. The real difference is that it allows you to offload more data and effectively uses bandwidth utilization and computation efficiently.

1.2.7 ZeRO++

Quantized weights are halved to int8, partitioning becomes hybrid helping in multi-node settings, and quantized gradients allow for communication volume decreases by swapping fp16 by int4 while computing gradient reduce-scatter operations.

1.3 Fully-Sharded Data Parallel

1.3.1 Full Sharding

ZeRO-3 is a method to shard parameters, gradients and optimizer state completely, where workers only hold the subset of weights, and activations + gradients are computed on-demand through parameter communication.

1.3.2 Hybrid Sharding

Hybrid sharding essentially combines replication and sharding, meaning that replication is across subsets, while sharding only happens on subsets of size F. For each pass, all-gather and reduce-scatter operations, on top of which all-gather occurs across nodes for averaged gradients.

1.4 Efficient Finetuning

1.4.1 Mixed Precision

Mixed-precision finetuning means that there all intermediate values are stored in half-precision, with real values in FP32.

1.4.2 Parameter-Efficient Fine Tuning

PEFT is a method to fine tuning a model by essentially freezing model weights; with techniques like LoRA, QLoRA, and IA^3.

1.4.3 Flash Attention

In using flash attention, you save memory, don't approximate, and have fast implementation of the attention mechanism. You can use various precisions with this.

1.4.4 Gradient/Activation Checkpointing

One method of efficient fine tuning involves retaining a portion of activations and recomputing the rest later. This causes memory to decrease by a magnitude of $O\sqrt{N}$.

1.4.5 Quantization

Methods like PTQ, QAT, and others allow for efficient inference, by quantizing weights and activations. PTQ does this after training, not during, in order to have efficient inference, while as QAT does training with quantized parameters, another method of making inference efficient.

1.4.6 Gradient Accumulation

Gradient accumulation allows us to increase batch size, while sacrificing throughput; however, the benefit is that we reduce the # of all-reduce operations, leading to better train times and less memory.

2 Megatron

2.1 Introduction

The NLP domain has had a massive increase in the available compute and dataset size in recent years, and this has enabled LLMs via unsupervised pretraining. SoTA results can be achieved by finetuning these models; but, with this comes memory constraints. Optimizers like ADAM and model reduction techniques can be used, but MEGATRON-LM is a simple and efficient model parallelism approach using model parallelism *within* layers. The authors show the effects of model size scaling on accuracy, but training left-to-right GPT2, and BERT, evaluating them on downstream tasks.

2.2 Background and Challenges

2.2.1 Neural Language Model Pretraining

Using large corpus pretraining has been the prevalent approach to NLP, and more advanced research consisted of learning + transferring models capturing word representations. The SoTA has advanced to transferring multi-billion parameter LMs, and MEGATRON-LM advances upon the SoTA models.

2.2.2 Transformer Language Models and Multi-Head Attention

NLP more recently has focused around the transformer architecture, with encoder/decoder, but with some variants like excluding one from the model. The MEGATRON-LM architecture is a transformer layer, with an Attention Head, and a MLP.

2.2.3 Data and Model Parallelism

Data parallelism refers to the idea of training a minibatch across multiple workers, whereas model parallelism is the idea of distributing memory usage and computation across multiple workers. Large batch training has caused complications that offsets increased traning, and the constraint is that the model *must* fit on just one worker. Within model parallelism, the two paradigms- layer-wise parallelism and tensor computation both require additional logic, changes to the optimizer, or graph recompilation.

2.3 Model Parallel Transformers

The authors use the transformer architecture to add a few primitives. In the MLP block, they parallelism the GEMM + GeLU by splitting the weight matrix A along its rows and input X along its columns, resulting in a synchronization point. Moreover, they split both GEMMs in the MLP, and require only one all-reduce operation in the forward and backward pass each. They also use parallelism in the multihead attention op, partitioning GEMMs with KQV in a column parallel manner. This splits attention head parameters and the workload across GPUs, and the GEMM is parallelized along its rows. The authors duplicate the computation across GPUs, to optimize the model; all in all, their approach is fairly easy to implement, only adding a few add-reduce operations.

2.4 Setup

2.4.1 Training Dataset

The authors use aggregate datasets consisting of Wikipedia articles, remove articles present in their test sets, and combine datasets. They then use preprocessing to remove documents with < 128 tokens in their body.

2.4.2 Training Optimization and Hyperparameters

The authors use mixed-precision training with dynamic loss scaling, Adam optimizer with weight decay, and gradient norm clipping. This was for GPT2, they use a similar process for BERT.

2.5 Experiments

Their experiments show their results, specifically for the GPT-2 and BERT models.

2.6 Conclusion

The authors successfully pass the single GPU per model training paradigm by slightly altering PyTorch transformations. They show that future work can include to increase pretraining scale, and improvement optimizers for memory and efficiency.