Distributed Deep Learning Methods

Eric Tang, Matthew Harvill

1 Introduction

Since we implemented fully-sharded data parallelism and tensor model parallelism for part 1, we attempted to speed up our leaderboard submission's model inference with quantization, and we were able to speed up noticeably on CPU. We discuss our implementation and findings in 5.

2 Background

For this project we implemented fully-sharded data parallelism, tensor model parallelism, and iterative magnitude based pruning.

Fully-Sharded Data Parallel Distributed data parallelism (DDP) is a method where multiple GPUs train with the same model weights, different data, and all-reduce gradients on each iteration to keep weights the same. Full-sharded data parallelism (FSDP) is an extension of DDP that also splits the model weights and optimizer states, and then shares them during forward and backward passes to achieve the same results as DDP. Although FSDP increases communication costs due to sharing weights and optimizer states between GPUs, it is an effective method because it reduces memory footprint, allowing for the training of larger models and/or batch sizes.

Tensor Model Parallel Tensor model parallel (TP) is a technique that partitions model weights across GPUs to compute matrix multiplication independently for the MLP and CausalSelfAttention modules in each transformer layer. After each of these modules, results/gradients are aggregated to maintain mathematical equivalence to single GPU training. This method is effective because it requires trivial changes to model architecture (splitting of some linear layer weights), but reduces memory requirements linearly with respect to number of GPUs. Like, FSDP, it increases communication costs, but these come from outputs, and not model weights. Unlike FSDP, default tensor model parallelism uses the same data for all GPUs.

Iterative Magnitude Based Pruning Iterative magnitude based pruning is a technique to reduce model size by removing weights with the smallest magnitudes, fine-tuning on the resulting model, and repeating until a desired model size is reached. Two common implementations involve (1) taking the smallest weights across entire matrices and removing them, and (2) finding rows with the smallest L2 norms and removing them. While the first method is more selective, it requires some overhead from storing the sparse representation, compared to the second method, which just removes rows from all linear layers, such that the model runs normally with smaller embedding dimensions.

3 Experimental Setup

Dataset, Models, Benchmarks For this project we are training GPT2-Medium and running inference with it on text from the Wikitext103 dataset. Shown below in 1 are the default train and validation losses on Wikitext-103 with GPT2-Medium.

Iteration	**Train Loss**	**Validation Loss**
After 10 iterations	3.6146	-
After 50 iterations	3.0950	3.1292
After 100 iterations	2.9568	-
Validation Loss (initial)	-	3.7432

Table 1: GPT2-Medium Benchmark Losses

4 Results

4.1 Part 1

Calculations For the following throughput graphs, we calculated throughput as:

$$\text{Throughput} = \frac{grad\_acc\_steps * batch\_size * block\_size * iters}{total\_time(sec)} = \frac{40 * batch\_size * 128 * 20}{total\_time}$$

Fully-Sharded Data Parallel As shown in 1, with a fixed sequence length of 128 tokens, and 16GB of memory per GPU, we see logarithmically increasing throughput as we increase the batch size. Also, as shown in 2, training the model requires a fixed amount of memory for the model (around 2.5GB per GPU), and otherwise increases linearly with batch size.

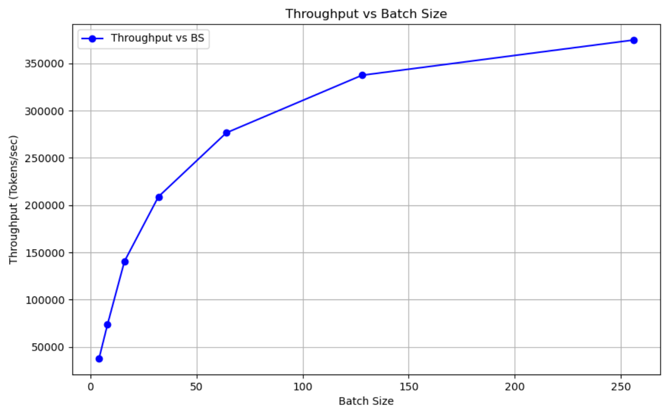

Figure 1: FSDP Throughput (Tokens/Sec) vs Batch Size

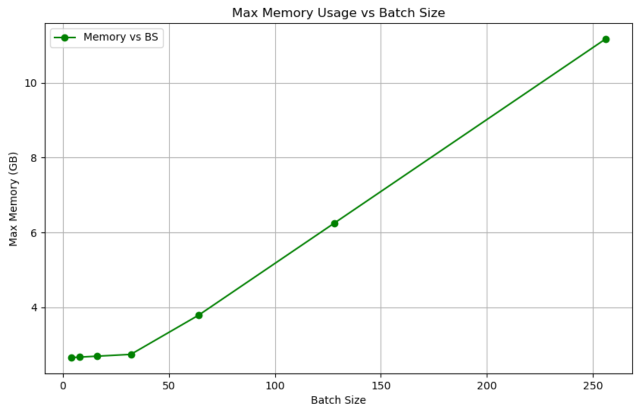

Figure 2: FSDP Max Memory Usage (GB) vs Batch Size

Tensor Model Parallel Similar to FSDP, in 3, with a fixed sequence length of 128 tokens, and 16GB of memory per GPU, we see logarithmically increasing throughput as we increase the batch size. Also, as shown in 4, training the model requires a fixed amount of memory for the model (around 2.5GB per GPU), and otherwise increases linearly with batch size.

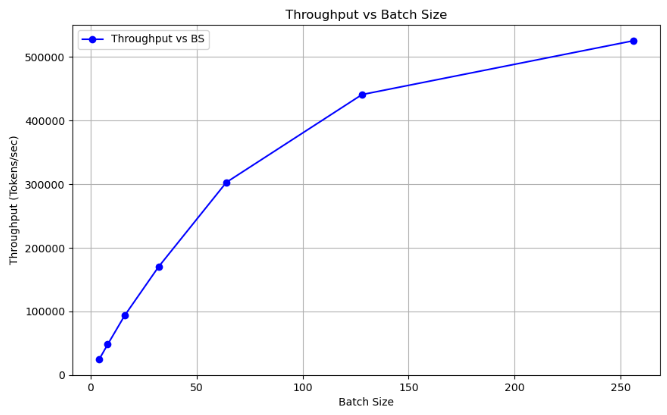

Figure 3: Tensor Parallel Throughput (Tokens/Sec) vs Batch Size

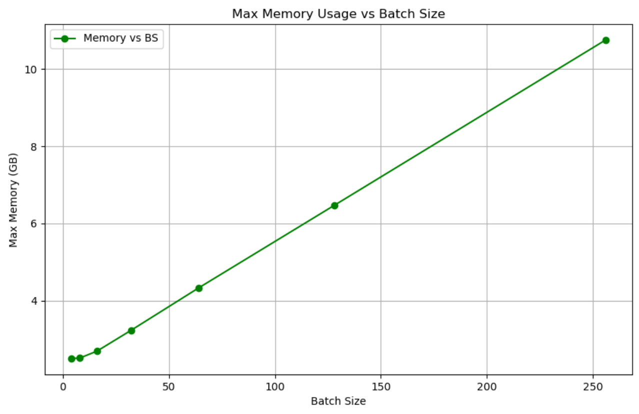

Figure 4: Tensor Parallel Max Memory Usage (GB) vs Batch Size

FSDP vs Tensor Model Parallel As we touched on in 2, FSDP shards model weights and optimizer states across layers, whereas TP shards every weight matrix. Therefore, both methods have a similar fraction of the model across GPUs (potential slight difference based on how layers are partitioned in FSDP). Unsurprisingly, we see that the max memory both methods use is almost identical across batch sizes. Since these different sharding techniques require different communication - FSDP shares weights, optimizer states, and all-reduces gradients, while TP all-reduces forwarded ouputs and gradients - we see differences in throughput as batch size increases. We see TP outpace FSDP as batch size increases because it doesn't have to share model weights and optimizer states in addition to all-reducing gradients.

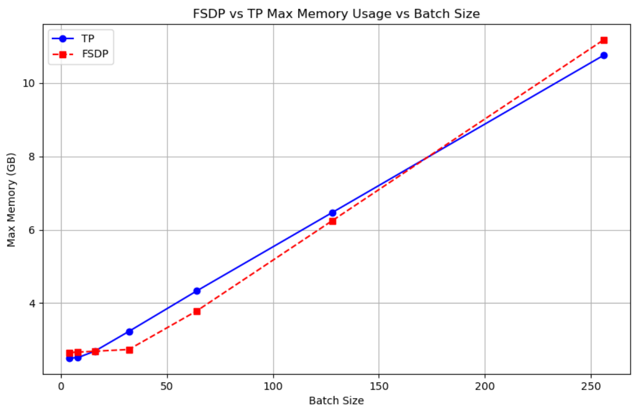

Figure 5: TP vs FSDP Max Memory Usage vs Batch Size

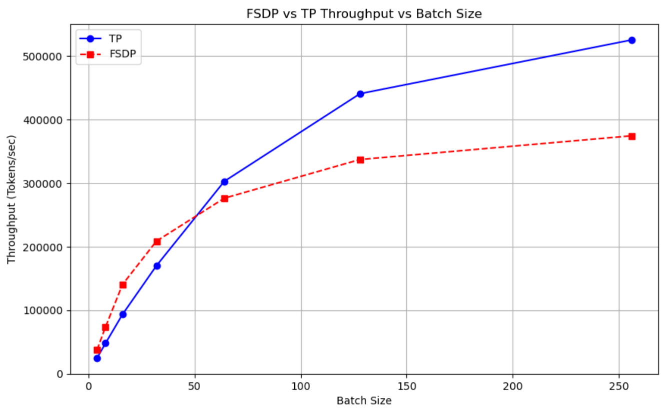

Figure 6: TP vs FSDP Throughput vs Batch Size

4.2 Part 2

Iterative Magnitude Based Pruning As shown in 7, we see the validation loss on WikiText-103 slowly increasing as we prune up to 80% of the model weights, and then spiking when pruning to 90%. We also see similar behavior in 9, except the row pruning starts off worse, increases quicker, and takes a lot more iterations to achieve similar results. This is most likely due to the fact that pruning individual weights is more selective, and doesn't remove as many salient weights until reaching the end (where all weights become important since the model has become low-rank).

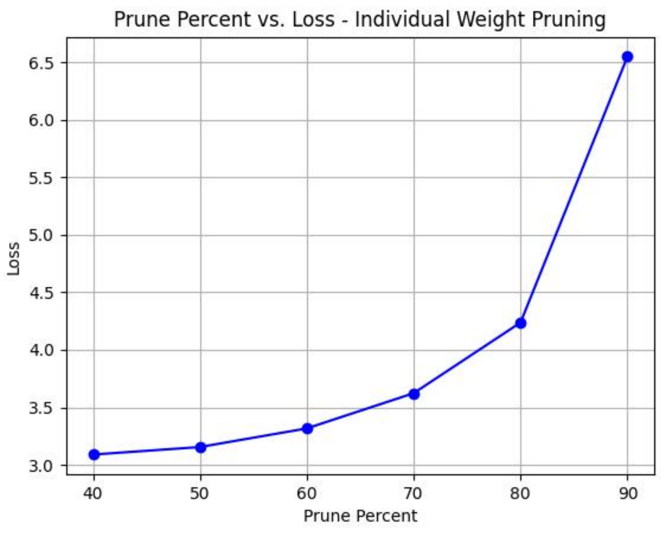

Figure 7: Individual Weight Prune Percentage vs Val Loss

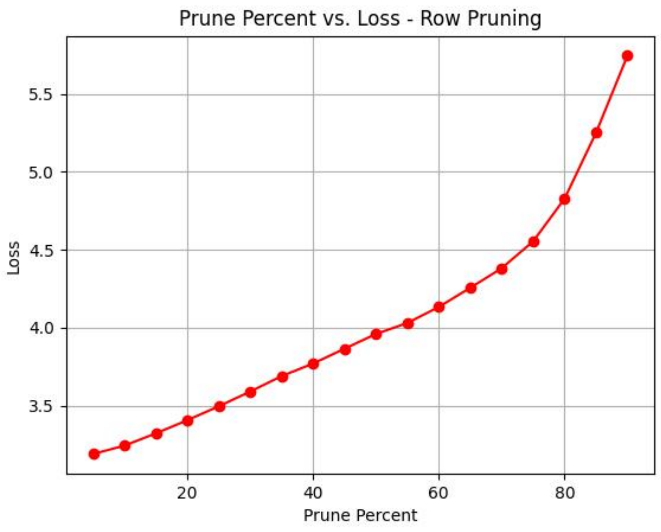

Figure 8: Row Prune Percentage vs Val Loss

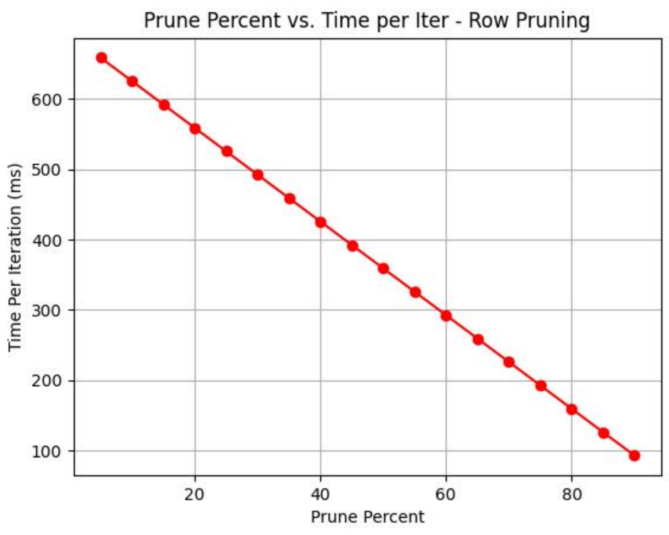

Figure 9: Row Prune Percentage vs Expected Time per Iteration

For our implementation of pruning rows, we zeroed out weights by keeping a mask for each set of weights that we pruned. This resulted in the speed across different pruning percentages to remain constant, since the masking operation is equivalent no matter the number of zeroes or ones in the mask weights. Therefore, in the above plot of row prune percentage vs expected time per iteration, we project the time per iteration as a linear speedup w.r.t the percentage of weights pruned.

Comparing results between pruning individual weights and pruning entire rows, we find that for most pruning percentages, pruning individual weights results in a lower validation loss. However, there is a tradeoff between the speedup and keeping the validation loss low, since for individual weight pruning, it is more difficult to actually reduce the computation being done within the model - we essentially need to keep a mask of weights that we no longer want to be active for individual pruning, while for row pruning, we can discard rows of the weights and the input, resulting in less computation.

4.3 Extension

Quantization We attempted using the built in PyTorch quantization library to reduce inference speed for our model. We initially started experiments on gpt2-small in order to iterate on the implementation more quickly, and used a reduced block-size of 64 rather than 1024. We ran into the issue of the PyTorch quantization library backend not supporting GPU inference, and so we were limited to benchmarking model inference speed on CPU resources. By quantizing linear layers in our model, we were able to observe a more than 2x speedup in inference time on full batches of data (i.e. timing the estimate_loss function).

Value	Quantized	Time
32	No	128.7186198234558
32	Yes	56.72795653343201
64	No	216.6015682220459
64	Yes	96.8292019367218

Table 2: Inference with and without Quantization

5 Analysis

5.1 Quantization

In addition to adding quantization in train.py, we added quantization for speedup in ‘sample.py‘ for generating text. However, due to the lack of support for int8 speedups on v100, and issues with PyTorch quantization running on backends other than CPU, our final results are reported without quantization. In addition, inspecting the outputs of the quantized vs the full model, we see a significantly less coherent output from the quantized model. Given more time, we would attempt to fix this by introducing Quantization Aware Training to help allow the quantized model to have stronger performance.

Without Quantization: *In 1957 , the United States Patent Office issued a patent application for " Method and Appar*

With Quantization: *wcombe that sql healthy Spec matters% fallacyETA tremendously Ravens editor Punjab pci thrivefing paragraph Shockzynski*

6 Leaderboard

Our final results are as follows:

```
{
    "loss": 2.837290620803833,
    "inference_latency_1": 50.69934152686839,
    "inference_latency_12": 206.2308219912759,
    "training_throughput_4": 37779.8200479,
    "training_throughput_12": 107243.990557
}
```

For training throughput, we follow our calculations from measuring FSDP and Tensor Parallel throughput, using

$$\text{Throughput} = \frac{grad\_acc\_steps * batch\_size * block\_size * iters}{total\_time(sec)} = \frac{40 * batch\_size * 128 * 20}{total\_time}$$

The units of our throughput results are samples per second.

For inference latency, we use the following calculation to produce our results:

$$\text{Latency} = \frac{batch\_size * max\_new\_tokens}{total\_time(sec)} = \frac{batch\_size * 500}{total\_time}$$

Thus the units of our inference latency results are in tokens per second.