HPML Assignment 6

sss9772@nyu.edu

Shashvat Shah

Q1.

Logs:
main
Files already downloaded and verified
Starting training with batch size 32

training epoch: 0 started
training epoch: 1 started
Training Time for Each Epoch: 14.172584776002054

Starting training with batch size 128

training epoch: 0 started
training epoch: 1 started
Training Time for Each Epoch: 4.044042579004099

Starting training with batch size 512

training epoch: 0 started
training epoch: 1 started
Training Time for Each Epoch: 2.411188792998928

Starting training with batch size 2048

training epoch: 0 started
training epoch: 1 started
Training Time for Each Epoch: 0.8340566200004105

Starting training with batch size 8192

training epoch: 0 started
GPU memory full at batch size : 8192, use 2048.0 for 1 GPU

Index	Compute time	Batch size
0	14.172584776002054	32
1	4.044042579004099	128
2	2.411188792998928	512
3	0.8340566200004105	2048

Table 1

As we can see that the code runs out of memory for batchsize 8192. So the largest batch size we can handle is 2048.

Machine configuration as per srun command

srun --mem=8GB --gres=gpu:v100:1 --ntasks-per-node=8 --account=ece_gy_9143-2023sp --partition=n1s8-v100-1 --time=02:00:00 --pty /bin/bash

8GB RAM V100 GPU with 16GB GPU VRAM and 8 CPU cores

Explanation for Table 1 results:

On NVIDIA V100 GPU, it takes 14.2 seconds to train one epoch (w/o data-loading) using mini-batch size 32, 4.1 seconds using batch-size 128 and so on. When batch size is larger, it takes longer or shorter time to train, because with larger batch sizes, the model handles a greater number of examples in each forward and backward pass, thereby reducing the number of iterations needed to process the entire dataset. Consequently, this optimizes GPU utilization and minimizes the overhead of launching CUDA kernels repeatedly for each iteration.

Q.2

	Batch size 32		Batch size 128		Batch size 512	
	Time	Speedup	Time	Speedup	Time	Speedup
1 GPU	19.885911484999	1	18.946140470000046	1	20.40828916800001	1
2 GPU	16.182581389999	1.18	11.05438008100009	1.42	11.1001814009999	1.45
4 GPU	11.779135971999	1.41	8.547305760999961	1.55	8.642169798000054	1.56

Table 2

Explanation for results in Table 2 results:

The measured data indicates that we are conducting strong-scaling experiments. In strong-scaling, the problem size remains constant while the computational power is increased. The data reveals that as the number of GPUs is increased, the processing time decreases for each batch size.

However, the speedup numbers suggest that strong-scaling is not highly efficient in this particular case, as there is no increase in speedup when using 4 GPUs. Although a significant speedup is observed in the 2-GPU scenario, the presence of communication overhead actually reduces the speedup in the 4-GPU scenario.

Q 3.1

	Batch size 32		Batch size 128		Batch size 512	
	Compute Time (s)	Comm Time(s)	Compute Time (s)	Comm Time(s)	Compute Time (s)	Comm Time(s)
2 GPU	16.1825	0.97720	11.054380	6.47988	11.100181	8.0618213
4 GPU	11.77913	0.06612	8.5473057	1.73996	8.64216979	3.1194091

Table 3

To measure the communication time I use the formula:

Formula:
communication_time = (total_epoch_time -- compute_time_per_gpu – data_loading_time)

total_epoch_time : time to perform data IO + cpu_gpu transfer + compute time

compute_time_per_gpu : as calculated in Q1, the time to do cpu_gpu transfer + forward and backward pass

data_loading_time : time to load data from disk to memory or IO time

From table 3 we see that the communication time increases with decreases with increase in number of GPU per fixed batch size. This is because we need to perform less iterations per GPU and it results in less forward and backward passes. Less forward and backward passes results in overall less communication time.

Q3.2

	Batch size 32	Batch size 128	Batch size 512
	Bandwidth util (GB/s)	Bandwidth util (GB/s)	Bandwidth util (GB/s)
2 GPU	0.1627382	0.050176	0.054911
4 GPU	0.212511	0.0637588	0.143226

Table 4

Bandwidth formula for DP algorithm

```
bandwidth = (device_count * total_params * 4 * num_forward_iters) / (cpu_gpu_time * (1024**3))
```

device_count : number of GPU allocated
total_params : number of gradients in the model
num_forward_iters: number of iterations it takes in 1 epoch
cpu_gpu_time : total time consumed in doing memory transactions

We see that the bandwidth utilizations drop with increase in batch size from 32 to 128 in both cases but remains similar while changing from 128 to 512.

Also by increasing the GPU count for a fixed batch size we see a overall increase in bandwidth utilization as more memory transactions are done because we have more gradient sharing to be done.

Q. 4.1

	Avg Loss	Avg Train Accuracy (%)
1 GPU (Batch size 128)	0.05	61.9
4 GPU (Batch size 2048)	1.2	30.63

Table 5

Explanation:

For the larger batch size and 4-GPU configuration, we observe higher values of loss and accuracy. This can be attributed to the lower number of updates made to the model, which is a result of using a larger batch size with the same learning rate. Although the model processed the same number of examples over the course of five epochs, the disparity in loss and accuracy results stems from the variance in the number of gradient updates.

Q 4.2

1. Scaling the learning rate: To ensure optimal optimization without divergence, it is recommended to proportionally increase the learning rate as the batch size increases. This scaling is justified by the fact that larger batches provide more accurate gradient estimates.
2. Adaptive rate scaling per layer (LARS): LARS is a technique that adjusts the learning rate for individual layers in a neural network based on the gradient norm of each layer. This approach has proven to be effective in training deep neural networks with sizable batch sizes.

Q 5

No, it is not necessary to communicate the entire model across GPUs repeatedly during training. Initially, the model is replicated across each GPU, and this involves communicating the parameters across the learners. However, once the initial replication is done, only the gradients need to be communicated between the GPUs. This is because the gradients can be applied to the model individually and in a deterministic manner. By communicating only the gradients, the overhead of repeatedly transmitting the entire model is avoided, leading to more efficient and streamlined training.

Q 6

As stated in the PyTorch documentation, when operating in distributed mode, it is crucial to invoke the "set_epoch()" method at the commencement of every epoch, prior to creating the DataLoader iterator. This step is essential for ensuring the proper functioning of shuffling

across multiple epochs. In distributed mode, each process possesses its own duplicate of the dataset and performs shuffling independently. By setting the epoch ID, it guarantees that each process shuffles its unique copy of the dataset in a distinct order for each epoch. This procedure guarantees the desired randomness and variety in the training data across epochs and facilitates improved model training and generalization. Without setting the epoch ID, the shuffling process might lead to repetitive patterns or ineffective distribution of the data, potentially hindering the model's ability to learn and adapt. Hence, the inclusion of the "set_epoch()" method in distributed mode is crucial for maintaining the desired level of randomness and achieving reliable performance during training.

Q 7

In the study, the authors introduced a technique called "gradient accumulation" that enables efficient communication across 4 GPUs. The main idea behind this approach is to accumulate gradients over multiple iterations before conducting a single communication step. By doing so, the researchers were able to effectively reduce the communication overhead associated with training large models.

The significance of this technique lies in its ability to streamline the training process by minimizing the frequency of communication between GPUs. Instead of transmitting gradients after every iteration, the gradients are accumulated over a certain number of iterations, and the communication step is performed once. This consolidation of gradient updates greatly reduces the overall communication time, allowing for more efficient training.

By employing gradient accumulation, the authors demonstrated that it is possible to achieve satisfactory results by communicating only the gradients across the 4 GPUs. This finding implies that the communication bandwidth is effectively utilized, as the accumulation of gradients over multiple iterations minimizes the need for frequent data exchange. As a result, training large models becomes more feasible and efficient, as the communication overhead is significantly reduced.

In summary, the paper presents the utilization of "gradient accumulation" as a technique to optimize the communication process in distributed training across 4 GPUs. The authors successfully demonstrated that this approach enables efficient training of large models by reducing the communication overhead and improving overall training performance.