Karan Vora (kv2154)
High Performance Machine Learning Assignment 6

Q1):

```
Epoch 1: Loss 1.5493, Accuracy 43.96%
Batch size: 32, Training time for epoch: 59.82 seconds
Epoch 2: Loss 1.0319, Accuracy 63.52%
Batch size: 32, Training time for epoch: 43.42 seconds
Files already downloaded and verified
Epoch 1: Loss 1.4499, Accuracy 46.88%
Batch size: 128, Training time for epoch: 39.07 seconds
Epoch 2: Loss 0.9548, Accuracy 66.09%
Batch size: 128, Training time for epoch: 36.85 seconds
Files already downloaded and verified
Epoch 1: Loss 1.6006, Accuracy 40.54%
Batch size: 512, Training time for epoch: 41.69 seconds
Epoch 2: Loss 1.0881, Accuracy 60.74%
Batch size: 512, Training time for epoch: 40.25 seconds
```

From the response mentioned above, we can see that increasing batch size does reduce the epoch time but only to an extent after that for huge batches, the training time is plateaued or higher.

==

Q2):

	Batch-size 32 per GPU		Batch-size 128 per GPU		Batch-size 512 per GPU	
	Time	Speedup	Time	Speedup	Time	Speedup
1-GPU	41.30	1	33.64	1	33.95	1
2-GPU	23.15	1.86	17.87	1.88	17.50	1.94
4-GPU	11.65	3.54	9.23	3.64	9.01	3.76

With larger batchsize we can see better scaling and speedup and it is in strong scaling as the problem is of same size but we are increasing the available resources.

==

Q3.1):

	Batch size 32 on GPU		Batch size 128 on GPU		Batch size 512 on GPU	
	Compute	Comm	Compute	Comm	Compute	Comm
2 GPU	23.15	2.9463	17.87	0.7441	17.50	0.2936
4 GPU	11.65	1.5360	9.23	0.4007	9.01	0.1639

From Q3 its apparent that with larger batchsize the communication time reduces drastically along with compute time as the GPUs spend more time computing the results and less time on interprocess communications

Q3.2):

First, let's define the formula to calculate how long it takes to finish an all-reduce operation.

1. Time taken to finish an all-reduce operation:

The time to finish an all-reduce operation can be calculated as:

`T_allreduce = (T_transfer + T_compute) * log2(P)`

where:
- T_transfer: Time taken to transfer data between GPUs
- T_compute: Time taken for reduction and broadcast computation
- P: Number of GPUs

T_transfer can be calculated as:

`T_transfer = (message_size / bandwidth)`

where:
- message_size: Size of the message being transferred (in bytes)
- bandwidth: Bandwidth of the interconnect between GPUs (in bytes/second)

Assuming the computation time is negligible compared to the data transfer time, we can approximate the all-reduce time as:

`T_allreduce ≈ (message_size / bandwidth) * log2(P)`

2. Bandwidth utilization formula:

The bandwidth utilization can be calculated as:

`Utilization = (total_data_transferred / T_allreduce) / bandwidth`

where:
- total_data_transferred: Total amount of data transferred during the all-reduce operation (in bytes)

Since the data is transferred in both reduction and broadcast phases, we have:

`total_data_transferred = 2 * (P - 1) * message_size`

Now you can calculate the bandwidth utilization for each multi-GPU/batch-size-per-GPU setup.

3. Calculated results in Table 3:

To calculate the results for each configuration in Table 3, you'll need to know the message_size and bandwidth. The message_size depends on the model and the batch size per GPU. The bandwidth depends on your system's GPU interconnect (e.g., NVLink, PCIe).

	Batch size 32	Batch size 128	Batch size 512
	Bandwidth (MB/s)	Bandwidth (MB/s)	Bandwidth (MB/s)
2-GPU	14.61	58.88	129.64
4-GPU	28.61	116.35	250.10

==

Q4.1):

```
Epoch 5: Loss 1.0156, Accuracy 63.37%
Batch size: 512, Training time for epoch: 9.61 seconds
Batch size: 512, Communication time for epoch: 0.1787 seconds
Bandwidth utilization for 4 GPUs and batch size 512: 250.10 MB/sec
```

For Lab2 Batch size 128 and 1 GPU

```
Epoch 5: Loss 0.5143, Accuracy 82.22%
Batch size: 128, Training time for epoch: 36.97 seconds
Batch size: 128, Communication time for epoch: 1.4798 seconds
Bandwidth utilization for 1 GPUs  and batch size 128: 30.20 MB/sec
```

Q4.2):

When using a large batch size, the model may suffer from reduced training accuracy due to a few reasons, such as less noise in the gradient updates, which can lead to overfitting and optimization challenges. Here are some strategies to improve training accuracy with large batch sizes:

1. Learning rate scaling: Increase the learning rate linearly with the batch size. For example, if you double the batch size, double the learning rate as well. This is known as the linear scaling rule.

2. Warm-up: Start training with a smaller learning rate and gradually increase it to the target learning rate during the first few epochs. This warm-up phase can help the model converge more smoothly.

3. Learning rate schedule: Use a learning rate schedule, such as step decay, cosine annealing, or cyclic learning rates, to adapt the learning rate during training. Reducing the learning rate over time can help the model converge to a better minimum.

4. Gradient clipping: Apply gradient clipping to prevent the gradients from becoming too large during training, which can cause instability or divergence.

5. Batch normalization: Use batch normalization layers in your neural network. Batch normalization helps in maintaining a stable distribution of activations and can improve the generalization performance of the model.

==

Q5):

One needs to set up the epoch ID in the Distributed Data Parallel (DDP) case because DDP uses a distributed sampler to divide the dataset across multiple processes. Each process works on a unique subset of the dataset. The epoch ID is crucial for synchronizing these processes and ensuring that they work on different portions of the dataset in each epoch.

In the case of DDP, each process runs on a separate server or GPU, and thus it is essential to keep them in sync. By setting the epoch ID at the beginning of each epoch, you ensure that:

1. Each process works on a non-overlapping subset of the dataset during each epoch, avoiding duplicate work and improving efficiency.
2. The distributed sampler shuffles the dataset consistently across all processes, maintaining the randomness of data samples and preventing overfitting or bias.

==

Q6):

No, gradients are not the only messages communicated across learners in a distributed training setup. Besides gradients, other messages that may be communicated include:

1. Model parameters: In some distributed training algorithms like parameter server-based methods, model parameters are communicated between the server and the workers. The workers receive the latest parameters from the server, compute gradients using their local data, and send the gradients back to the server, which then updates the global model.

2. Optimizer state: For certain optimizers like Adam or RMSprop, there is additional state information (e.g., running averages of gradients and squared gradients) that may need to be communicated and synchronized across learners.

3. Loss values and evaluation metrics: During training, it is common to aggregate loss values and evaluation metrics (such as accuracy, F1 score, etc.) across all learners to monitor the overall training progress and performance.

4. Control messages: In a distributed training environment, control messages are used to coordinate and synchronize the processes. These messages can include instructions for starting a new epoch, adjusting the learning rate, or stopping the training process based on predefined criteria.

===

Q7):

No, it might not be sufficient to communicate only gradients across 4 GPUs in the 512 batch size per GPU case. Here's why:

1. Optimizer state: As mentioned earlier, some optimizers like Adam or RMSprop require additional state information to be communicated and synchronized across learners. This information is crucial for updating the model parameters in a consistent and efficient manner.

2. Synchronization: In a distributed training setup, especially when using Distributed Data Parallel (DDP), it is important to maintain synchronization across all GPUs. This might involve exchanging information about the learning rate schedule, epoch progress, and other control messages to ensure all GPUs are working in a coordinated fashion.

3. Loss values and evaluation metrics: To monitor the overall training progress and performance, it is necessary to aggregate loss values and evaluation metrics across all GPUs. This allows detecting convergence, overfitting, or other issues, and taking appropriate actions during training.

4. Load balancing: In some cases, the workload might not be evenly distributed among the GPUs. To avoid a situation where some GPUs become bottlenecks, it is essential to communicate information about the workload distribution and, if necessary, redistribute the workload to maintain a balanced training process.