The Impact of Model Architecture and Batch Size on Data Parallel multi-GPU Training

Eric Stevens

Abstract

This paper highlights factors that should be considered when determining if the training speed of an artificial neural network will improve by distributing the workload across multiple graphical processing units (GPUs) instead of using a single GPU. Specifically, this paper explores the impacts of network architecture and training batch size on the performance of data parallel training in PyTorch. Two separate artificial neural networks with dissimilar architectural characteristics are constructed and tested for training time performance. Evaluation of each network architecture is performed by varying the training batch size and number of GPUs used to train, recording the training time for each variation. The disparity in training time performance between the two models is discussed at length in order to give the reader a sense of the primary considerations when determining the effectiveness of applying multi-GUP data parallel training to reduce training time.

1 Introduction

In the last 10 years neural networks have out performed traditional methods of machine learning in many domains. A key advantage that neural networks have over traditional machine learning methods is their ability to continue achieving higher performance on very large datasets. Where traditional learning algorithms performance goes flat with more data, artificial neural networks continue to achieve higher accuracy.

This revolution in machine learning has resulted in the need for machines that can process huge quantities of data quickly. The nature of the training process of artificial neural networks revolves around large Matrix multiplications. CPUs are not the proper tool for this task due to the serial manner in which they execute instructions. Hardware technology from the graphics processing world, however, deals with large matrix multiplications in parallel, enabling video manipulation. These graphics processing units (GPUs) are able to dramatically speed up the training time of artificial neural networks and have become the hardware tool of choice for artificial neural network training.

While hardware for training neural networks continues to improve, individual hardware components are not sufficient for the complexity of the models and the size of the data in the machine learning world of today. Currently, the most effective way to achieve high-throughput artificial neural network training is by distributing training workloads across hardware accelerator nodes such as GPUs.

There are a variety of techniques that can be used to leverage multiple GPUs in the training of artificial neural networks. Data parallel training, for example, is a technique in which the architecture of the model is broadcasted to each GPU and the training data is partitioned so that each GPU can process a portion of it.

In this paper the technique of data parallel artificial neural network training across a cluster of GPUs is explored in detail. Specifically, empirical evidence is gathered to assess the performance of data parallel GPU training with respect to both network architecture and the training hyperparameter, batch size.

2 Background

2.1 Multi GPU Neural Network Training

Using multiple GPUs for artificial neural network training is commonplace in contemporary machine learning workflows. There are multiple ways in which distributing training workloads across hardware nodes can be leveraged.

2.1.1 Considerations For Distribution Of Training

There are two main factors that need to be kept in mind when evaluating the ability for artificial neural network training workloads to be parallelized. The first of these is the forward/backwards pass. These steps involve exposing the entirety of the training data to the entirety of the model architecture. This means that each input instance makes it from the input layer of the network to the output layer, where a loss value can be calculated and be back-propagated through the entirety of the network.

The second factor to consider is the need to update all of the networks training parameters coherently prior to the beginning of the next training batch. To be more explicit, after a single input batch is processed by the network, the gradients calculated from the backward pass need to update the model parameter such that each instance in the entirety of the following input batch is exposed to the same parameter weights as one another.

These two factors act as guide rails for the way data parallelism can be leveraged in the training of artificial neural networks. The meaning of the factors will become more clear in the section below, where two distinct strategies for training across multiple GPUs are discussed.

2.1.2 Model vs Data Parallelism

Techniques for distributing the training workload of an artificial neural network across GPUs generally fall into one of two main categories, model parallelism and data parallelism. Model parallelism is not evaluated thoroughly in this paper and is only discussed here to draw the distinction between it and it's counterpart, which will be discussed in detail.

In model parallelism, the architecture of the artificial neural network itself is segmented into different components that perform different processing tasks. In other words the entire network architecture is broken up into smaller self-contained artificial neural networks. Each of these components is shipped to a different GPU. After the components of the network have been set up on the GPUs, the input data can be passed to the input of the first component, and that components output can be forwarded on to the next GPU as prescribed by the network architecture. In this case forward and backward passes occur across multiple GPUs, which may result in a throughput bottleneck with respect to the size of the training data. However, updating of model parameters can be done locally on GPU during the backward pass, avoiding the need to synchronize gradients.

Data parallelism functions very differently than model parallelism. With data parallelism the entirety of the artificial neural network architecture, which starts on a default GPU, is copied and replicated on all of the other GPUs. After this, input training batches can be split up into smaller mini-batches that can be processed in parallel by each of the GPUs. Dividing the training batches up into mini-batches reduces the throughput bottleneck that results from large data. However, although the same model architecture is sitting on each GPU, the fact that each copy of the model is exposed to a different set of training data results in different loss calculations, and therefore different gradient updates. This situation, in which you have different gradient calculations on each replica of the model, which each sit on a separate GPU, results in the need to synchronize the gradients in a central location. The model parameters must be gathered from each GPU to the default GPU in order to update the entirety of the model. After collecting the gradients and updating the centralized model on the default GPU, the new model needs to be redistributed to all of the GPUs again in order for the next training batch to be processed. This results in a potential bottleneck when passing model parameters back and forth from the default GPU.

3 Methods

In this section the tools and methods utilized to generate the experimental results are discussed in detail for the sake of reproducibility. In order to achieve this goal, several aspects of the methods used in this experiment need to be highlighted. A cloud service provider was used for access to hardware. The components and networking characteristics of the cloud instance used are relevant to the character of the findings in the experiments. A specific implementation of data parallelism is used to perform the experiment. That implementation has its own influence on the performance characteristics of the experimental results. A specific machine learning task is utilized for the experiment with a readily available data set. The artificial neural network architectures that are tested in the experiment are explained in detail, since they are the main variable in the experiment.

3.1 Lambda Labs

Lambda Labs is a workstation building company and cloud service provider of machines targeted at achieving high performance artificial neural network training. For this experiment, the cloud service offered by Lambda Labs is utilized. Lambda Labs allows you to spin up virtual machines with a dedicated virtual CPU, four dedicated NVIDIA GTX 1080ti graphics processing units, 32GB of RAM, and dedicated networking between the components. This choice was made to mimic having dedicated hardware without having to purchase it.

The Nvidia GTX 1080 TI GPUs each have 11 GB of on-board RAM and are connected through a pcie mesh.

3.2 Data Parallelism in PyTorch

PyTorch is the framework that was utilized for this experiment. PyTorch has implementation of data parallelism through their DataParallel class, which may impact performance when compared with other implementations of data parallel multi-GPU training. It is important to discuss the way in which PyTorch implements data parallelism across multiple GPUs.

What follows is a brief walkthrough of the process PyTorch applies to each training batch in the data parallel workflow. Forward Mini batch scatter: A default GPU, usually GPU 0, ingests an entire training batch and segments it into its original size divided by the number of available GPUs. It then ships these segmented mini batches to the other GPUs. Model Broadcast: PyTorch takes the model that is sitting on the default GPU and copies its entire architecture to all of the other GPUs. Parallel Forward Pass: A forward pass on the mini batches performed in parallel on their respective GPUs. Output Gather: The outputs of the forward pass are gathered to the default GPU.

Backward Loss calculation: The individual losses are calculated on the default GPU. Loss scatter: the losses on the default GPU are redistributed to their respective GPUs. Parallel backwards pass: the backwards pass is performed in parallel on each GPU with respect to their individual losses. The gradients for each parameter on each GPU are calculated. Reduce gradients: The gradients from all of the GPUs are returned to the default GPU where they are aggregated and used to update the main model.

This process is repeated one time for each batch in the training set.

3.3 Training Task

The training task selected for this experiment is arbitrary. The main reason a specific training task was selected is to keep the experiment grounded in the context of a practical application. More general experimental results could have been collected with the use of variable models, a subject for the Future Works section.

The task selected for this experiment is a simple automatic speech recognition classification task. The model input instances are 1.5 second audio clips, each of which contains a speaker saying a digit from 0 to 9. The labels corresponding to the training data are one-hot encoded vectors, where the one hot index corresponds to the digit being said in the audio clip.

The total size of the training data set is 92 MB, relatively small for an artificial neural network training task. These 92 MB correspond to 2,000 1.5 second audio clips with a sample rate of 8 kHz, stored as 32 bit floating-point numbers. The relatively small size of the training data serves to amplify the effects that are being studied in the experiment.

The task performance of both network architectures was above 99 percent accuracy on the training data (100 epochs, batch size of 10). In both cases, the models we're likely very overfit to the data. No regularization was attempted and no testing data set performance was examined. This is pointed out to emphasize the fact that what the models are doing is not important to the experiment. What is important is the training time as it relates to the model architecture and batch size.

3.4 Network Architecture

Since the major goal of this experiment is to understand how neural network architectures affect data parallel multi-GPU training speed, two very different model architectures are constructed and their performances are compared. Below, the architectures of each of the models will be discussed in detail. Particular attention should be paid to the discussion of model parameters size and computational complexity.

3.4.1 Linear Model

The first model is a simple sequence of linear layers with rectified-linear and sigmoid activation functions between. The input to the model is an ar-

ray of 12000 32-bit floating point values. 12,000 corresponds to 1.5 seconds worth of audio data multiplied by an 8 kHz sample rate. The linear architecture is a very straightforward four layer step down from a size 12,000 input to a size 10 output. The size of the data is decreased considerably in the first layer, taking it from 12,000 down to 1,000. From there, a layer of 1,000 to 100 followed by 100 to 10 and finishing with a final 10 to 10 layer.

All of these layers are fully connected. The resulting network has 12,102,220 training parameters total. That is the sum of the input and output size of each layer multiplied together plus the sum of the output layer sizes for the biases. Each training parameter is a 32-bit floating-point number, resulting in 46 megabytes worth of training parameter data, where megabyte equals 2 to the power of 20 bytes.

Layer (type)	Output Shape	Param #
Linear-1	[-1, 1, 1000]	12,001,000
ReLU-2	[-1, 1, 1000]	0
Linear-3	[-1, 1, 100]	100,100
ReLU-4	[-1, 1, 100]	0
Linear-5	[-1, 1, 10]	1,010
Sigmoid-6	[-1, 1, 10]	0
Linear-7	[-1, 1, 10]	110

Total params: 12,102,220
Trainable params: 12,102,220
Non-trainable params: 0

Input size (MB): 0.05
Forward/backward pass size (MB): 0.02
Params size (MB): 46.17
Estimated Total Size (MB): 46.23

Figure 1: Linear model results at batch size 20 for all GPUs

3.4.2 Convolutional Model

The convolutional artificial neural network has a much more complicated architecture. Again, the input size is 12000 and the output size is 10. However, here we are using convolutional layers instead of fully connected linear layers. In the first convolutional layer the input is padded by 100 on either side then 50 kernels, each one of them 200 wide, are convolved with the input using a stride of one. This results in an output the same size as the input. A rectified linear unit is applied before a max-pooling step with a size of 4 is applied, bringing the input from the initial 12,000 wide array to a two-dimensional 3000 x 50 array where the 3,000 corresponds to the 12,000 Max pulled at a size of 4, and the 50 corresponds to the 50 kernels.

This General pattern repeats for two more layers. The second layer has 30 kernels with a size of 100 each and a max-pooling layer of size 3. With a padding of 50 on either side of the 3,000 input and a 100 wide kernel, the output of the convolutional layer remains at 3,000. The rectified linear unit is applied and then the max pooling of size 3 brings the shape from an input of 50 by 3000 to an output of 30 kernels by 1,000. The final convolutional layer in the model repeats the exact same process, with 10 kernels in a max-pooling layer of size 5, bringing the size down to 10 kernels by 200. The model then takes the 10 x 200 array and flattens it into a single 2,000 wid vector. A linear layer is applied to get from 2,000 down to 100, and another linear layer is applied to get from 100 down to 10.

Obviously, the architecture of the convolutional model is much more complex. The interesting, yet obvious fact to those who understand convolutional neural network parameter sharing, is that even with all of this model complexity the total number of training parameters is 376,200, which corresponds to 1.4 MB worth of training parameter data. This is small when compared to the 46 Maggie be of training parameters in the fully-connected linear model.

Layer (type)	Output Shape	Param #
Conv1d-1	[-1, 50, 12001]	10,050
ReLU-2	[-1, 50, 12001]	0
MaxPool1d-3	[-1, 50, 3000]	0
Conv1d-4	[-1, 30, 3001]	150,030
ReLU-5	[-1, 30, 3001]	0
MaxPool1d-6	[-1, 30, 1000]	0
Conv1d-7	[-1, 10, 1001]	15,010
ReLU-8	[-1, 10, 1001]	0
MaxPool1d-9	[-1, 10, 200]	0
Linear-10	[-1, 100]	200,100
Sigmoid-11	[-1, 100]	0
Linear-12	[-1, 10]	1,010

Total params: 376,200
Trainable params: 376,200
Non-trainable params: 0

Input size (MB): 0.05
Forward/backward pass size (MB): 12.07
Params size (MB): 1.44
Estimated Total Size (MB): 13.55

Figure 2: Linear model results at batch size 20 for all GPUs

4 Experiment

After setting up these two model architectures the experiment is rather straightforward. To reiterate, the goal of this experiment is to test the influence that model architecture and batch size have over the training time when performing multi-GPU data parallel training. Therefore the experiment involves collecting data across the domain of different num-

bers of utilized GPUs and different batch sizes.

For each of the two models described above we perform a training routine on 1 GPU, 2 GPUs, 3 GPUs and 4 GPUs. On each of the GPU collections, we run the training routine once each for batch sizes 10, 20, 30, 50, 70, 100, 150, 200, 300, 400, 500 and 600. For every combination of number of GPUs and batch size, 20 epochs of training are performed and the time to perform each training epoch is recorded. At the end of the 20 epochs an average time per epoch is calculated and stored.

The results of the experiment is a data set containing the time-per-epoch to train for each combination of number of GPUs and a training batch size. One of these data sets is created for each of the models discussed above. Having this empirical data allows us to evaluate the training characteristics when spreading the models across multiple GPUs, as well as the performance on a single GPU.

5 Experimental Results

Here, the training time performance of each model will be discussed. Interesting characteristic differences of the models training time performance will be highlighted. This section is only intended to provide the empirical results of the experiment. An interpretation of these results will be provided in the following section.

5.1 Linear Model

There are three distinct observations that can be made about the training performance of the linear fully connected model. The first is that, at all batch sizes, a single GPU trains more quickly then multiple GPUs. The second characteristic of training performance with the linear model is that training performance seems to improve as a function of the inverse of the batch size at all GPU counts. Finally, it is noted that while the increase of training time from a single GPU to a second GPU can be several multiples of the single GPU train time, the time performance of a second GPU compared to a third or even a fourth GPU barely changes at all.

5.2 Convolutional Model

The training speed performance of the convolutional model has very different characteristics then it's linear counterpart. At a batch size of 10, it takes almost the same amount of time to train whether 1, 2, 3 or 4 GPUs are used. Above a batch size of 10, we begin to see a consistent relationship with

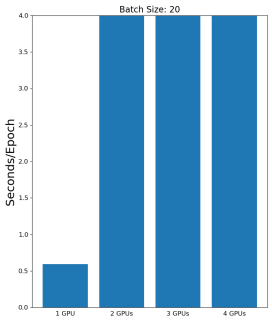

Figure 3: Linear model results at batch size 20 for all GPUs

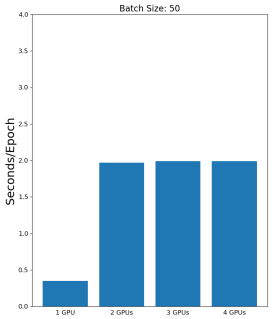

Figure 4: Linear model results at batch size 50 for all GPUs

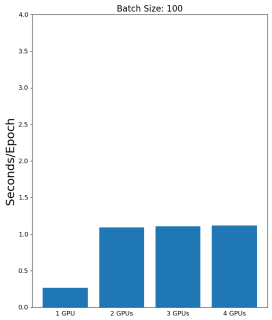

Figure 5: Linear model results at batch size 100 for all GPUs

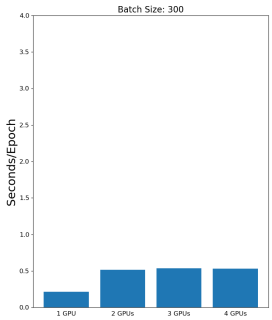

Figure 6: Linear model results at batch size 300 for all GPUs

the number of GPUs resulting in better training time performance. It should be noted that for a single GPU, the smallest batch size get's the best performance. It should also be noted that for all other numbers of GPUs, the best performance is not necessarily at the highest batch size. Different numbers of GPUs perform better at different batch sizes. There is no monotonic relationship between batch size and training time, unlike what was true for the linear model.

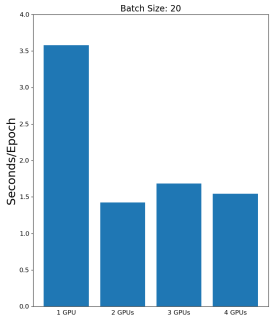

Figure 7: Convolutional model results at batch size 20 for all GPUs

6 Interpretation

The purpose of this section is to make observations about the empirical data that may suggest performance relationships to architectural characteristics and batch size selection. It is important to know that the conclusions drawn in this section are not observed directly. There will be a conversation with regards to the inter-GPU communications network architecture that will not be based on observing traffic on that network, but rather derived from logical induction. There will also be some discussion

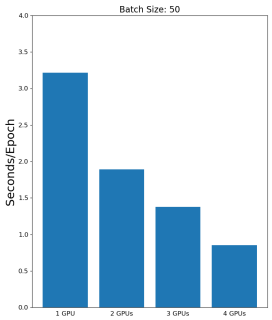

Figure 8: Convolutional model results at batch size 50 for all GPUs

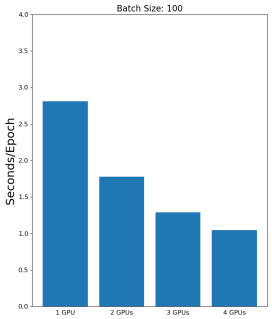

Figure 9: Convolutional model results at batch size 100 for all GPUs

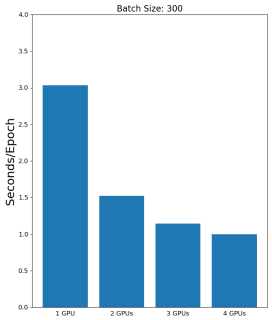

Figure 10: Convolutional model results at batch size 300 for all GPUs

that relates to the GPU interfacing software and GPU algorithms for processing data. It is important to know that this discussion is not based on the study of these concepts. Again, these will be inferences arrived at by looking at the empirical results of the experiment. Care will be taken to point out where conclusions come from and to distinguish conjecture from fact.

6.1 Linear Model

The three interesting observations from the empirical data resulting from applying the experiment to the linear model were as follows. A single GPU always out performed multiple GPUs, regardless of the batch size. The training time is a function of the inverse of the batch size. Finally, the fact that although a single GPU could be multiple times faster than 2 GPUs; 2, 3, and 4 GPUs all trained in a very similar time.

6.1.1 Poor Multi-GPU Performance

Why does the linear model get no benefit from training on multiple GPUs? To answer this question it is important to go back and understand how data parallel works in PyTorch. Looking again at the order of operations in the PyTorch data parallel flow, we see that there are two steps where the entire model must be transferred. In the model broadcast step, the entire model on the default GPU must be sent to all of the other GPUs. In the gradient reduce step, all of the GPUs send their updated weights back to the default GPU. This will be an import part of answering why the performance is so poor for the multi-GPU linear model training.

For the moment let's assume that the GPUs have independent communication channels with one another. Consider a batch size of 20. Since the total number of training instances is 2,000, a batch size of 20 results in the need to process 100 batches per epoch. Now consider the size of the model parameters in the linear fully-connected model. The over 12 million model parameters take up 46 MB of memory, which we will round to 50 MB for simplicity. As mentioned above, the model replication phase and the gradient reduction phase of the data parallel workflow in PyTorch involves transferring the entirety of the model twice for each batch. One transfer from the default GPU to the others (model broadcast) and once from all the GPUs to the default GPU (gradient reduce). So to calculate how much network traffic goes over each of the inter-GPU communication lines in a single epoce: the size of the model, times 2 model transfers per batch, times the number of batches per epoch . Assume a batch size of 20 for a moment. That would be 50 MB of model parameters, times 2 transfers of those parameters per batch, times 100 batches per epoch. This result is 10 GB, a huge amount of network traffic resulting from the combination of a model that is 50 MB in size and a batch size that forces 100 batches to be processed for epoch.

The implication here is that the model size in combination with how many times it is being transferred is what is causing the poor performance. In other words we are facing an inter-GPU networking bottleneck. To expand on this point, let's take another look at the graph of epoch time over the batch size of the linear model. We have calculated that a batch size of 20 will result in 10 GB of transfer. The experiment record is that a batch size of 20 on multiple GPUs roughly corresponds to a five-second epoch training time. At a batch size of 100 and total training instances of 2,000 we have 20 batches per epoch. Using the same math as before, 50 MB of model parameter data, times 2 transfers per batch, times 20 batches results in 2 GB of transfer per epoch. The experimental results show that the batch size of 100 corresponds to a training performance of roughly one second per epoch on multiple gpus. Finally, repeating the math once again, a batch size of 200 results in 10 batches per epoch. So 50 MB times 2 transfers, times 10 batches equals 1 GB, which in our experiment corresponds to a training time of roughly 0.5 seconds per epoch.

The pattern that emerges when you look at this data is the training time appears to be a function that is linearly with respect to the amount of model parameter data being transferred per epoch. This observation makes sense of the data resulting from the experiment, and also explains the relationship between the curve and the inverse of the batch size we see in the data. If the network bandwidth is the limitation due to the size of the model parameters, then we would see this clear pattern that correlates the amount of data being transferred to the training time per epoch.

When training on a single GPU there is no network traffic in terms of model parameters. The single GPU training occurs entirely on that GPU and all time is spent on the data processing rather than on I/O. It is therefore reasonable to conjecture that poor performance in multiple GPU data

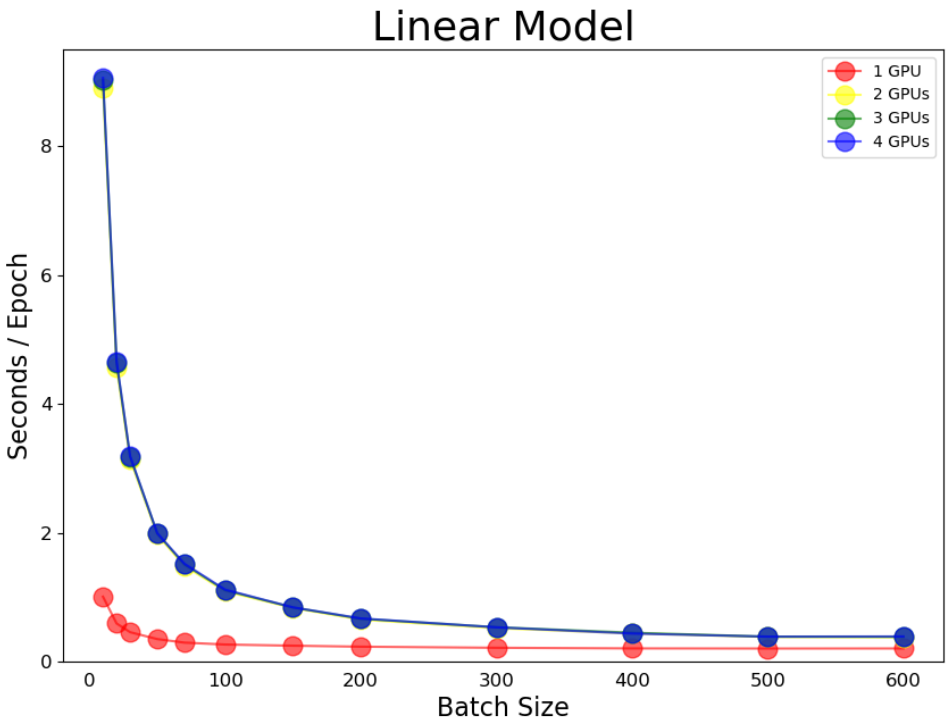

Figure 11: Linear model multiple GPU lines with epoch training time on Y axis and batch size on X axis.

parallel situations is due to the inter-GPU network traffic, caused by the size of the model parameters and exacerbated by small batch sizes. This is network choke, when the bottleneck in training performance is inter GPU communication.

From the discussion above it becomes obvious why this beautiful curve occurs with respect to batch size. The calculations above showed that the training time performance was linearly related to the number of batches per epoch, which multiplies the model parameter size. Since batches per epoch is the inverse of batch size we can say that the relationship between training time and batch size is that the training time is some constant times the inverse of batch size.

6.1.2 Why Do 2, 3, and 4 GPUs Perform Equally

When we observed the graph we noticed that 2, 3, and 4 GPUs have roughly the same training time performance as one another. In our explanation above we made the assumption that each GPU had dedicated communication links to every other GPU. This is not necessarily true, although modern GPU interconnect architectures do allow for it. The performance characteristics observed in the experiment presented do depend on the computing infrastructure on which they are performed. This is why time was taken in the beginning of the paper to describe the hardware and its configuration. It would not be difficult to try a different hardware configuration and get different results then this experiment did. That is why an attempt is made to stress concepts that are generalizable and highlight considerations that need to be taken when evaluating whether an individual hardware configuration will result in different performance characteristics.

The Nvidia system management interface allows us to view the type of communication networks that we have between the GPUs. In the case of the hardware that was used for this experiment, we do have direct GPU to GPU communication, however this communication traverses a PCIe channel as well as a PCIe host bridge, implying that the links between gpus are non dedicated.

So how is it then, with the inter-GPU Network bandwidth being the bottleneck for the training time, that our training performance did not continue to degrade as we went from 2 GPUs to 3 GPUs and 4 GPUs? The answer to this question is not directly observed through the experiments performed in this paper. Rather, a possible explanation was arrived at through researching Nvidia product capabilities.

Nvidia NCCL Is the collective Communications Library provided by Nvidia. Nccl is a library Of collective Computing Primitives that are aware of the topology of the GPU interconnect. the collective Communications Primitives that are supported are as follows

All reduce All Gather Broadcast Reduce Reduce Scatter

This Library allows for very efficient topology aware inter GPU communication, allowing many data transfers to and from multiple GPUs to take virtually the same amount of time as it would to transfer data to a single GPU. While a detailed overview of the NCCL algorithms is out of the scope of this paper, the performance of these types of collective communication algorithms plays an unavoidable role in the types of performance considerations that are central to this discussion. Therefore, the results of the experiment presented here should be viewed with the awareness of libraries like NCLL.

6.2 Convolutional Model

The behavior of the convolutional network during training is drastically different than that of the linear model. The most important factor here is the dramatic difference in model size. The fact that the model size is so small relative to the linear model suggests that the inter-GPU networking bottleneck would not occur unless a very small batch was used. For example, a batch size of 2 on 2 GPUs means that 1,000 batches must be processed. Since the size of the convolutional model is 1.5 MB, this results in a total of 3 GB of transfer, Corresponding to roughly two seconds of training time in the linear model. Comparing the batch size of 20, which we calculated for the linear model would result in 10 GB of transfer, the convolutional model only transfers 300 MB.

While after a certain batch size we do get the hoped for performance increases as the result of adding GPUs, at a certain point no matter the number of GPUs increasing the batch sizes does not increase the training performance.

The empirical data collected in the experiment for the convolutional network is completely unlike that of the linear network. Where the linear network was a smooth curve easily captured with the small mathematical function, the convolutional network has characteristics that seemed almost random. There are dips in the graph in different places but don't seem to be following any particular pattern.

Through this randomness there can be seen an interesting characteristic of these curves. Each GPU line towards the left hand side of the graph experience is a large dip before bouncing off of some bottom and settling at a training time that remains fairly consistent for the rest of the batch sizes. The single-GPU line seems to be coming off of this bounce from the lowest bat size measurement, which is 10. What's interesting about the rest of these lines is that they bounce with the same number of instances per GPU as the single GPU line. So, the 2GPU line bounces at a batch size of 20, corresponding to 10 training instances per GPU; the 3 GPU line bounces at a batch size of 30, corresponding to 10 training instances per GPU; and finally, no measurement was taken at a batch size of 40, but the bounce on the 4 GPU line occurs at a measurement of batch size 50, 12.5 instances per GPU. So it does appear that there is a relationship between the number of training instances on a GPU and the location at which the bounce occurs.

Combining this observation about the location of the bounce for each line with the fact that training time remains the same for all batch sizes thereafter, another conjecture can be made with regard to why this model performs the way it does. A number that we haven't discussed so far that can be seen in the models summary summary of the network is the forward backward pass size of the model. This corresponds to how much memory a single training instance will take up in the GPU when passing through this model. Each of these gpus has 11 GB of RAM. So the fact that the bounce is occurring at 10 instances means that the bottleneck is not due to a lack of memory on the GPU, since 10 instances would only correspond to 120 MB of memory on the GPU. It does appear safe to say that this number is related to the computational complexity of the model. With this model the input layer is creating 50 kernels of size 200, then 30 kernels of size 100, then tent rentals of size 50.

This represents a lot of memory usage per instance while training, but more importantly, a realization that the computational workload is very complex for the multi-layer convolutional network is indicated. So the conjecture about why the bounce occurs is that it occurs where the batch size aligns with the computational throughput of the GPUs. In other words, 10 training instances per

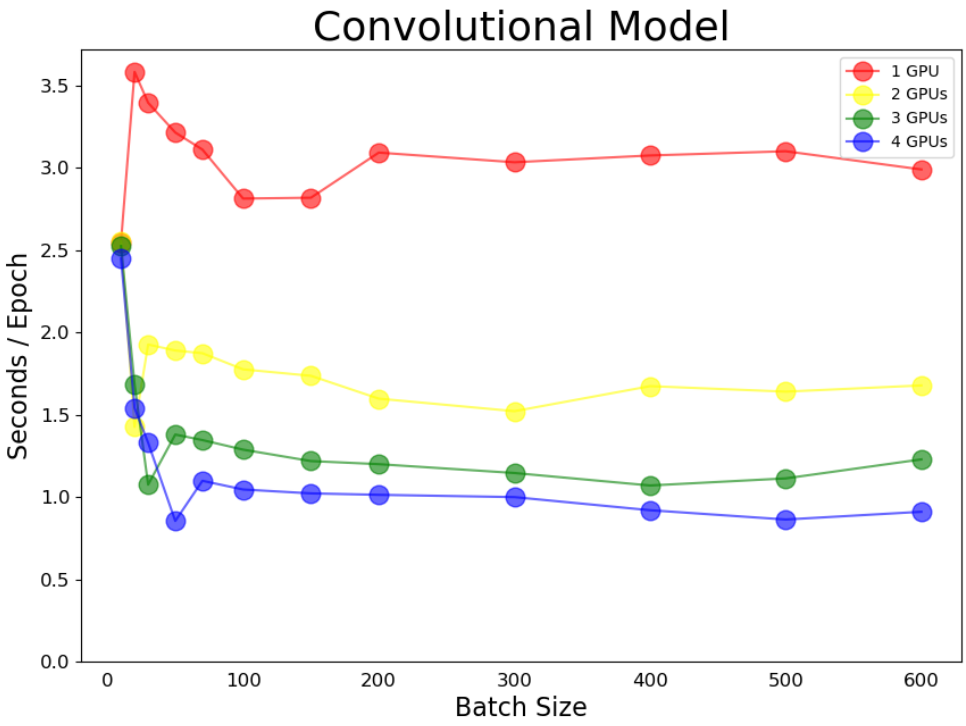

Figure 12: Convolutional model multiple GPU lines with epoch training time on Y axis and batch size on X axis.

GPU is the most the GPU can ingest and process on the fly. After that point, the GPU must buffer training instances and retrieve them as computational resources free up. At this point changing the batch size does not have a major impact on the speed of training because all training instances will be buffered anyway. The sporadic behavior of the GPU lines thereafter is something for which a conjecture will not be presented, but the way in which the GPU handles buffering and retrieving data for processing is suspected.

7 Conclusion

The main take away from the preceding is that adding GPUs to your artificial neural network training workflow is not always advantageous in terms of training time performance. There are a number of factors to consider when determining whether data parallel training on multiple GPUs in PyTorch is the correct choice to speed up the training of a model. These factors include model size, desired batch size, inter-GPU networking bottlenecks, and GPU computational bottlenecks.

In a situation like that of the linear model, adding GPUs may not help and could potentially slow training down. The key consideration to determine whether network choke is occurring is to look at the size of the model parameters relative to the computational complexity of the model. If the model parameter size is very big but the computational task is not very complex then it is likely that you will face network choke, and are better off training on a single GPU. However, if it is determined that the computational workload is heavy for a single GPU, and the size of the model parameters is relatively small, it is likely that adding GPUs and setting up multi-GPU data parallel training will decrease training time..

8 Experiment Source Code

https://github.com/Eric-D-Stevens/multi_GPU_DataParallel_Tasks