Scaling Distributed DNN Training for Large Images

(Technical Research Report)

Team:
Rayan Hamza
Nawras Alnaasan
Zhengqi Dong
Arpan Jain

The Ohio State University
4/12/2021

Table of Contents

Introduction

Advances in high performance computing and systems facilitated the rise of Deep Learning (DL). DL is a subset of Machine Learning (ML), which is a subset of Artificial Intelligence (AI). DL requires more computation power and large datasets compared to machine learning as it extracts features from raw data automatically. DL uses deep neural networks (DNNs) for classification and regression tasks. The effectiveness of DNNs has made deep learning a default method for AI problems like object detection and recognition, automatic speech recognition (ASR), and natural language processing (NLP). With such advantages, the scientific community has started using DL models in more challenging problems like mesh tangling, hydrodynamics, molecular engineering, and digital pathology. However, deep learning models have already saturated the peak potential of modern architectures like CPUs and GPUs. Therefore, there is a need to explore novel parallelization strategies to exploit multiple GPUs for DNN training.

Distributed DNN training has become the primary approach to train large and complex DNNs that can take weeks or months to train on a single GPU. Broadly, distributed DNN training can be divided into three categories; 1) Data Parallelism, 2) Model-Parallelism, and 3) Hybrid Parallelism. Data parallelism provides an excellent speedup on a large number of GPUs by replicating the DNN on each GPU and performing forward and backward pass simultaneously on all GPUs. Allreduce operation is used to synchronize the training by aggregating the gradients from the model replicas. However, state-of-the-art DL models like Amoebanet and GPT3 cannot be trained on a single GPU due to the memory requirement as these models (out-of-core models) cannot fit inside the memory of a single GPU. Researchers have proposed model-parallelism to overcome the limitation of data parallelism by distributing the deep learning model across multiple GPUs, thereby, reducing the memory requirement per GPU. However, model-parallelism suffers from underutilization of resources and cannot deliver good speedup on a large number of GPUs. Therefore, hybrid parallelism is proposed in the literature to scale the training of out-of-core models to a large number of GPUs. Hybrid parallelism combines data and model-parallelism for out-of-core DL model training.

Background

DNN training

A Deep Neural Network (DNN) is a set of layers, which is composed of several neurons. A neuron can be regarded as a basic computation that combines weighted summation and activation function. An activation function is a nonlinear function that introduces nonlinearity into DNNs. DNN training entails minimizing the loss function by adjusting the weights of the neurons. DNN training is a two-step process; 1) Forward Pass and 2) Backward Pass. In the forward pass, prediction is calculated by feed forwarding the input through the layers of neurons. Error is calculated by comparing prediction and actual output. Partial error is required at every neuron to determine the direction and magnitude of adjustment. This is achieved by backpropagating the calculated error backward using the chain rule of differentiation. Figure 1 shows the forward and backward pass in DNN training.

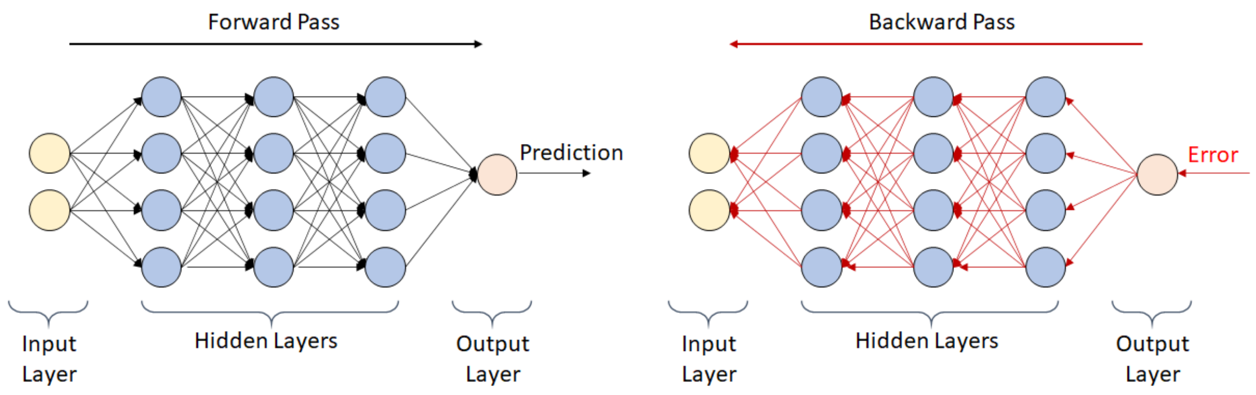


Figure 1. Forward and backward pass for DNN

Distributed DNN training

Training DNN on multiple processing units like CPUs and GPUs is known as distributed DNN training. Broadly, it fits into one of the following categories: 1) Data Parallelism, 2) Model-Parallelism, and 3) Hybrid Parallelism.

Data Parallelism

Data parallelism distributes the data among processing units and replicates the model on all processing units to perform forward and backward pass simultaneously on all processing units. There are two ways to synchronize the DNN training using data parallelism: 1) Parameter Server and 2) Allreduce operation. In parameter-based approaches, a central parameter server is created that holds the parameters of trained DNN. All workers/processing units send calculated gradients to the parameter server and wait for the latest parameters from the parameter server. On the other hand, global gradients can be calculated by aggregating local gradients using allreduce communication primitive, which is an element-wise summation operation and makes the result available to all the participating workers. Distributed DNN training has moved to allreduce operation to synchronize the training as parameter server has several issues like congestion and a single point of failure.

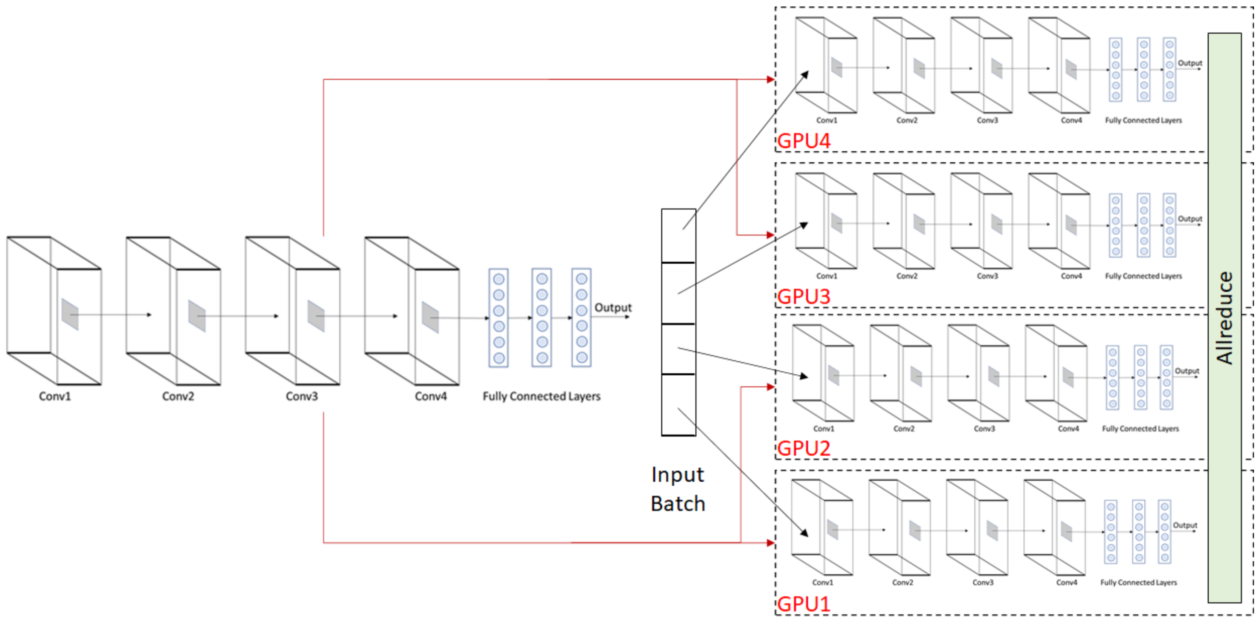


Figure 2. Input batch split over multiple GPUs

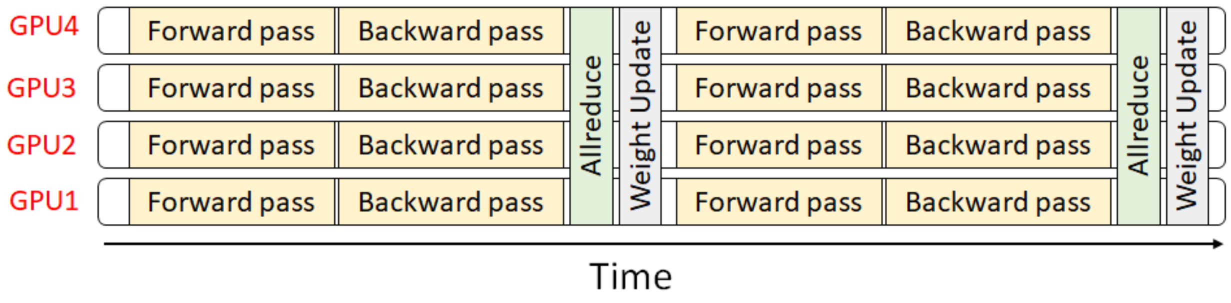


Figure 3. Data parallelism process over time

Model-Parallelism

Instead of distributing the data, model-parallelism splits the DNN among multiple processing units like GPUs or CPUs. It solves the limitation of data parallelism by partitioning the model and decreasing the overall memory requirement on a single processing unit. There are several ways to split the model. One of the most common approaches is to split the model at the finest granularity of a layer. Figure 4 shows the partitioning of a 7 layer convolution neural network (CNN) across four GPUs and the need to preserve connections between the layers. Model-parallelism needs distributed implementation of forward and backward pass as layers are spread across multiple processing units, which makes implementing model-parallelism a challenging task. Figure 5 shows the forward and backward pass computation over time on 4 GPUs for the model in Figure 4. Model-parallelism suffers from the under-utilization of resources as only one processing unit performs computation at any given time due to sequential data dependency in forward and backward pass.

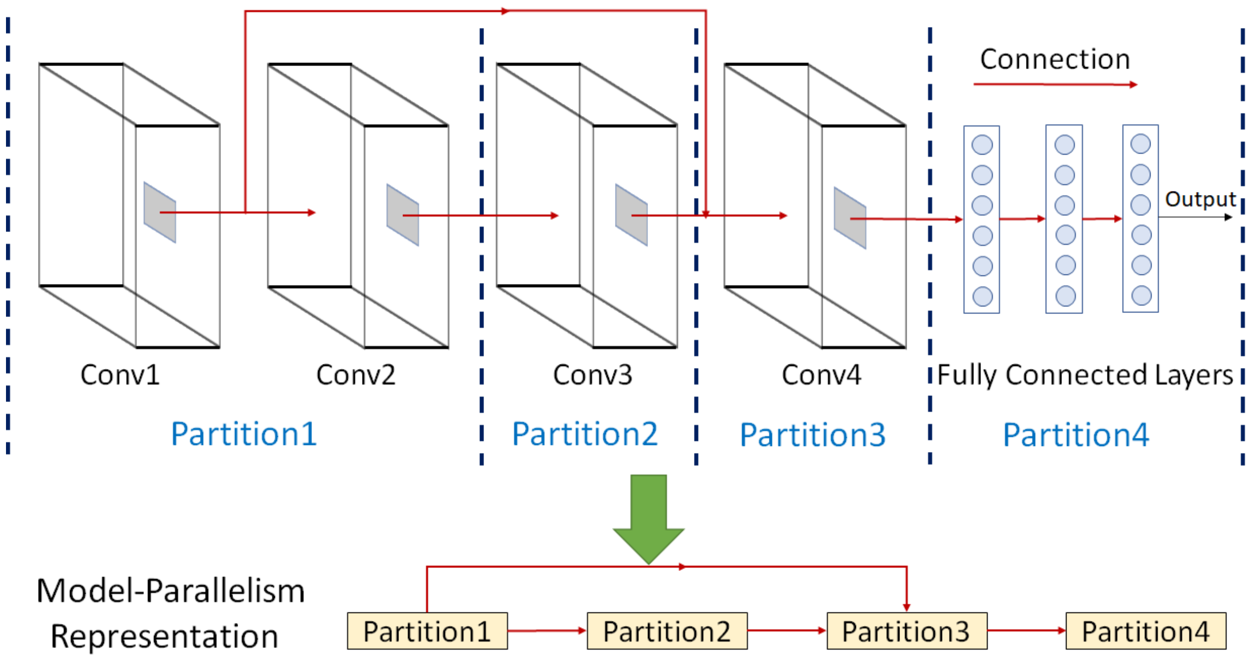


Figure 4. Model partitioning in model parallelism

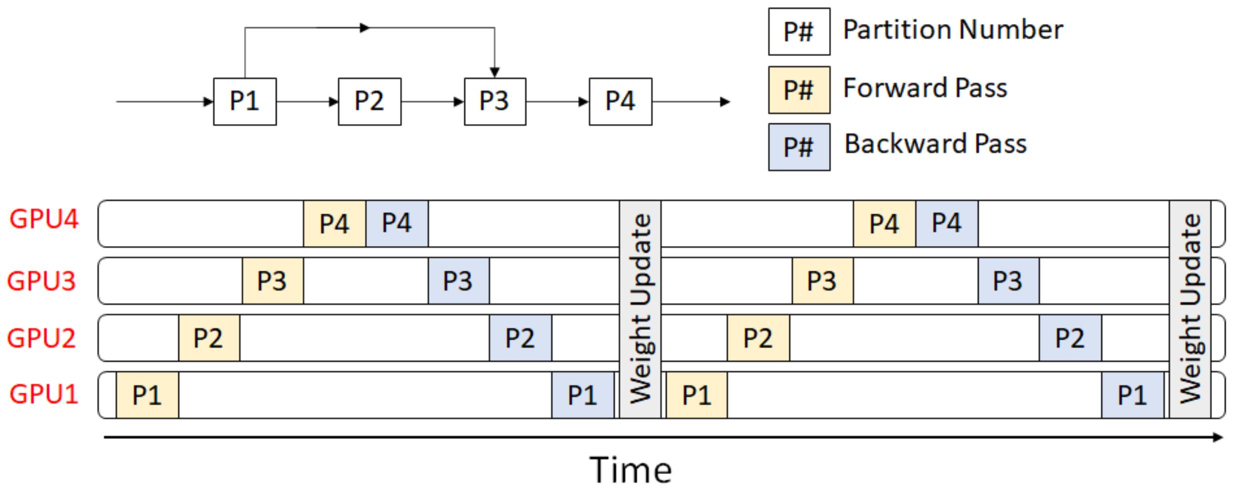


Figure 5. Model parallelism process over time

Hybrid Parallelism

Hybrid Parallelism aims to solve the scaling issue of out-of-core models by combining model-parallelism with data parallelism. Model-parallelism is used to train the out-of-core model by splitting the model across multiple processing units. TO scale the DNN training to a large scale, data parallelism is used to replicate the model partitions and allreduce operation to synchronize the training. Figure 6 shows the distributed DNN in hybrid parallelism and Figure 7 shows the forward and backward pass over time in hybrid parallelism.

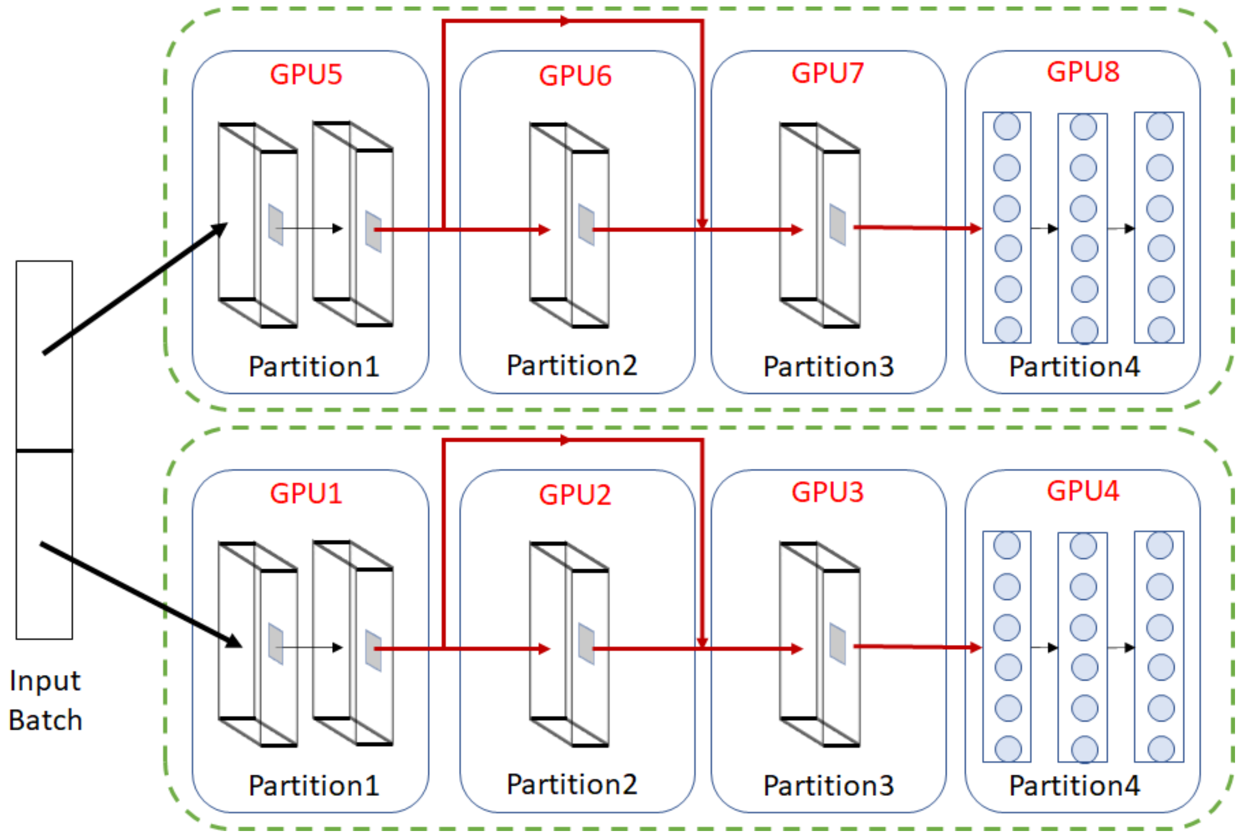


Figure 6. Model and batch splitting in hybrid parallelism

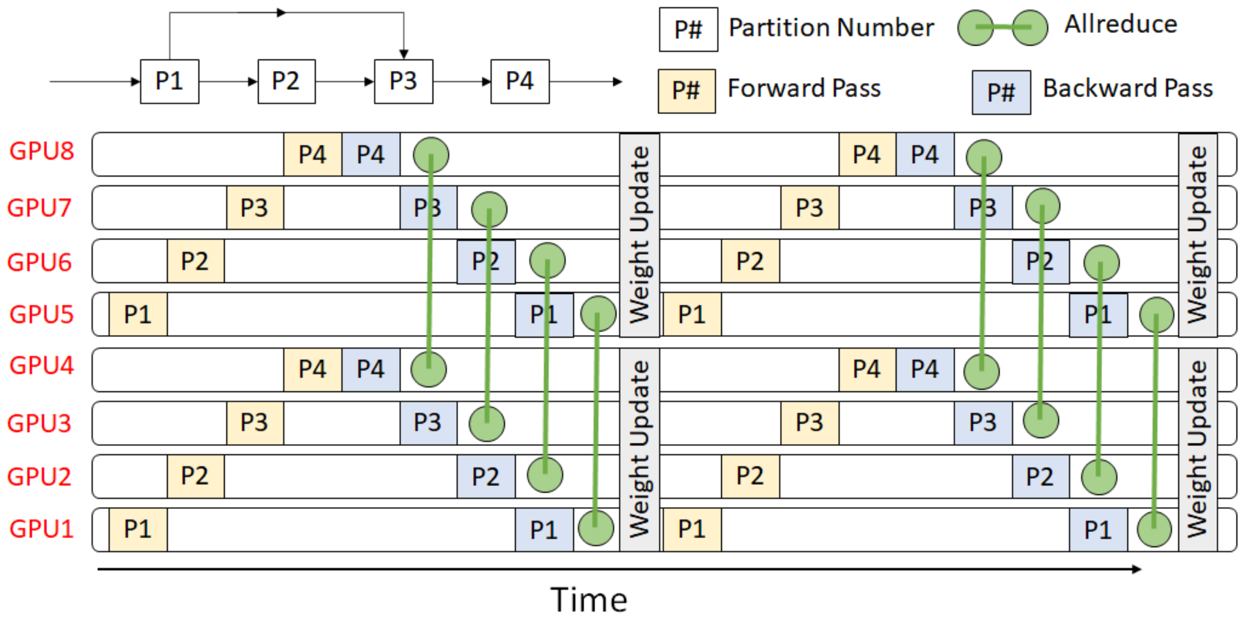


Figure 7. Hybrid parallelism process over time

Literature Review

Using various resources to find related literature, 6 relatively recent research papers have been chosen and extensively reviewed. Below is a brief summary of each:

1. PipeDream-2BW: Memory-Efficient Pipeline-Parallel DNN Training

Problem to solve: Out-of-core DNN model, too large to fit into single GPU memory. Moreover, we not only want to solve the problem but also want efficient training of large models, with both high throughput and low memory footprint.
Solution: PiepeDream-2BW(double-buffered weight updates), a system that performs memory-efficient pipeline-parallel training of DNN models with billions of parameters and is able to achieves high throughput, low memory footprint, and data parallelism-like semantics through a innovative weight update double buffering strategy. (Particularly granted to the its weight gradient coalescing strategy and planning algorithm)
Innovative techniques:

a. **2BW's weight gradient coalescing strategy**: a technique that reduces the memory footprint of training while avoiding pipeline flushes.
b. **2BW partitioning planning algorithm**: PipeDream-2BW's planner determines how to split a model over the available compute devices by exhaustively searching over the reduced search space of all possible parallel-pipeline configurations.

Result/Impact: Compared to the hybrid parallelism baseline model, PipeDream-2BW is 6.9x faster for BERT-192, and 5.4x faster for GPT-2 by using 64 GPUs. Compared to the GPipe model, PipeDream-2BW outperforms corresponding GPipe configurations at the same global minibatch size by up to 1.9x due to its lack of periodic pipeline flushes. Compared to PipeDream, PipeDream-2BW is up to 6.2x faster.

2. ZeRO-Memory: ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

Problem to solve: Being able to train large DNN models (billions to trillions of parameters), while retaining high computational granularity, low communication overhead, and eliminate the memory redundancy.
Solution: Zero Redundancy Optimizer(ZeRO): It can optimize memory and training speed while increasing the model size and eliminates memory redundancies in data- and model-parallel training while retaining low communication overhead and high computational granularity.
Innovative techniques:

a. **ZeRO-DP**: A type of ZeRO that builds upon DP(Data Parallelism). It aims at reducing the per-device memory footprint of a model while retraining the memory efficiency. In another word, it aims to retain the training efficiency of DP while achieving the memory efficiency of MP, through partitioning them -- optimizer states, gradients and parameters -- across data parallel processes.

b. **ZeRO-R:** Another type of ZeRO that builds upon MP(Model Parallelism). It targets at increasing the memory availability for even larger models by reducing the residual memory consumption/redundancies in MP and the time it takes for the memory allocator to find free contiguous memory.

Result/Impact:

c. Experimented on ZeRO-100B(a model with up to 170B parameters), enables 8x increase in model sizes, 10x in throughput improvement(41.4 TFlops/GPU), achieves super-linear speedups on modern GPU clusters, and trained the largest model in the world.
d. No model refactoring is necessary, and it is as easy to use as standard data-parallelism.
e. Has the potential to scale beyond 1 trillion parameters by using today's hardware.
f. Declared as a prime candidate for future investigation on large model training.

Relevance: All the papers were recently published at the super computing conference -- SC '20: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis -- which is one of the top conferences in the field of HPC.

3. GEMS: GPU-Enabled Memory-Aware Model-Parallelism System for Distributed DNN Training

Problem to solve: Model parallelism for out-of-core models (more than 1 machine) that has a higher speedup than earlier model parallelism solutions. Currently targeting pathology and large images.

Solution: GEMS - removes the memory footprint in basic model parallelism by introducing a replica of the model, and switching propagation operations between the two.

Innovative techniques:

a. **GEMS MAST**: use a replica of the model to fill the memory footprint in MP-basic; synchronize parameters of the models via allreduce operation
b. **GEMS MASTER**: GEMS MAST, but using allreduce half as much (stacking more compute onto the allreduce operation)
c. **GEMS HY**: Can be implemented from GEMS MAST or GEMS MASTER, gives each GPU (model partition) an allreduce operation

Result/Impact: observed 1.36x speedup in GEMS: MAST and a 1.83x speedup in GEMS: MASTER compared to alternative methods. GEMS introduces more feasible tools to solve overarching problems in digital pathology and other domains that require heavy memory usage.

4. Training Large Neural Networks with Constant Memory using a New Execution Algorithm

Problem to solve: Maintain desired depth of neural networks (NN) while reducing the memory usage during training.
Solution: Layer-to-layer (L2L): can run very large models independent of depth by applying graph reduction and cross mixed precision to receive results on par with standard NN training.
Innovative techniques:

a. **Graph Reduction**: abstracts encoder layer architecture so that only one encoder layer is needed, reduces memory footprint from O(n) to O(1) (only 3 layers in model graph needed: one embed layer, one encoder layer, and one class layer).
b. **Cross Mixed Precision:** Keeping floating point (FP) 32 precision on master copy of model while running reduced graph at FP16 precision. Weights updated with FP32 precision, propagations done at FP16 precision

Result/Impact: The depth of models no longer affects the memory available on GPUs as drastically as it has before; this can bring model parallelism problems back to data parallelism potentially, or even reduce the amount of resources needed for model parallelism techniques.

5. GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism

Problem to solve: Finding a way to handle model parallelism with task-indepence and efficiency while supporting any large scale neural network overcoming barriers like memory limitations and communication overhead.
Solution: GPipe: a flexible pipeline parallelism library which handles scaling of any network expressed as a sequence of layers with efficacy regardless of the network size and architecture.
Innovative techniques:

a. **Batch-splitting:** GPipe introduces a batch-splitting pipeline algorithm. This algorithm starts by partitioning each layer into cells placed on separate accelerators then splits mini-batches into micro-batches and executes the micro-batches over a cell.
b. **Synchronous Gradient Descent:** Synchronous gradient descent is used for training where gradients are accumulated across micro-batches and then applied at the end of the mini-batch which is consistent regardless of the number of partitions (cells).

Result/Impact: GPipe provides flexibility and efficiency when used on large neural networks. By scaling up the networks, the model is capable of achieving linear speedup with the number of devices used to train a network. Moreover, GPipe provides flexibility to support any DNN that can be represented as a sequence of layers and reliability by

using synchronous gradient descent that is consistent regardless of the number of partitions.
Relevance: The final version of this paper was published in 2019 by a team of researchers at Google at the gpu technology conference GTC20. This paper provides a great insight on an efficient technique on model parallelism that may inspire other or similar techniques to be implemented in this project.

6. Channel and Filter Parallelism for Large-Scale CNN Training

Problem to solve: Accelerating large-scale training of CNN on complex models and large datasets via model parallelism while keeping low communication overhead and enabling strong scaling. Also, overcoming some limitations of data-parallelism such memory limitations and mini-batch size with wide CNNs.
Solution: This paper introduces three algorithms that allow for strong scaling, reducing communication overhead with weak scaling, and requiring no hyperparameter tuning. Each algorithm is differentiated by the data movement and computation patterns it uses. The three algorithms are listed in the innovative techniques section below.
Innovative techniques:

a. **Stationary-x**: Avoids communication of input data during forward propagation.
b. **Stationary-y:** Symmetric to Stationary-x swapping forward and backward propagation. It avoids communication of input data during backpropagation as well as the gradient computations.
c. **Stationary-w:** More general algorithm combining the communication pattern of both Stationary-y and Stationary-x algiorhtms. It controls the amount of communication during forward/backward propagation.

All three algorithms partition the parameters of convolutional layers instead of replicating them for each processor (which is both data and model parallelism).
Result/Impact: The algorithms provided in this paper improve strong and weak training including 4.1x reduction in training time of residual networks and 4x reduction in allreduce overhead. They reduce communication overhead and memory pressure. Moreover, the wider models introduced in this paper provide more accuracy on training/testing the ImageNet dataset.
Relevance: This paper was published in 2019 at the super computer conference SC19. It provides insight on how to accelerate large-scale CNN training using multiple techniques including channel and filter partitioning. The ideas behind these techniques might come in handy in this project especially when dealing with wide models.

Objectives

We outline two concrete objectives for this project:

- User-friendly interface for distributed DNN training using model-parallelism in PyTorch.
- Explore CPU-Offloading schemes to decrease memory requirement in model-parallelism.

We split the first objective into two tasks required in automatically splitting the model: 1) Analytical model to predict computation for each layer and 2) Finding connections between layers in DNN.

Hypothesis

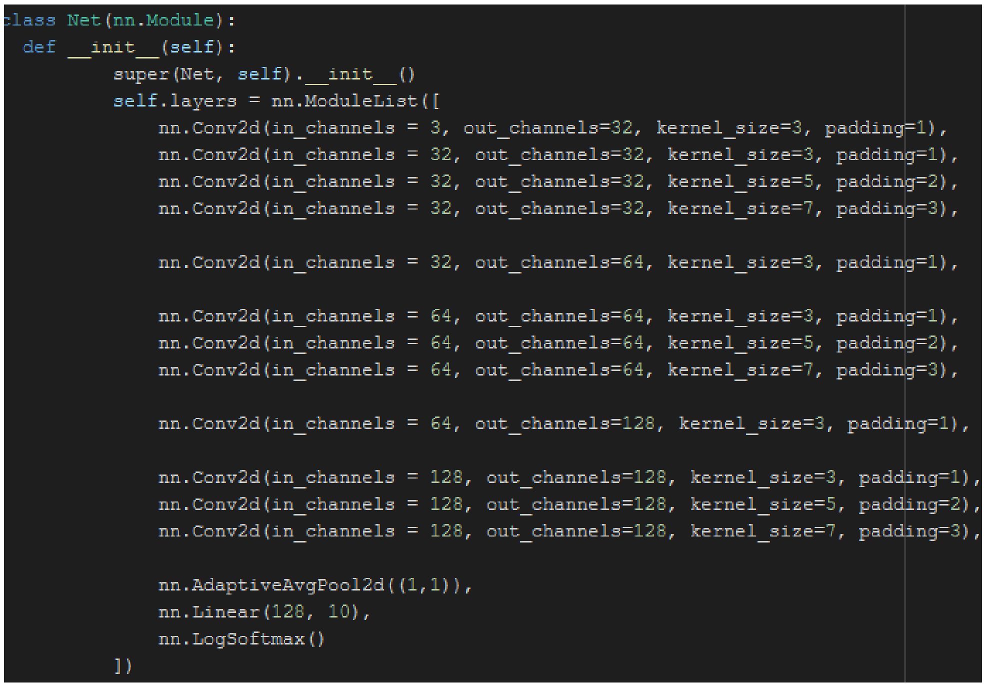
For this research project, three research hypotheses are tested:

1. Use analytical models to estimate execution time for a model split to efficiently split the DNNs across multiple GPUs
2. Use PyTorch's Model and API to understand data flow in DNNs written in PyTorch to implement user transparent model-splitting
3. Use CPU offloading mechanism to optimize GEMS-MASTER design

Methodology

Analytical Model

The purpose of the analytical model is to offer an informed approach to splitting models out-of-core. While achieving a network representation allows us to view which dependencies to preserve in doing the splitting, the analytical model seeks to associate each layer with its expected pass/propagation times, which are based on various model parameters. Our main focus was on the time estimation of the convolutional layer (nn.Conv2D), a layer that has propagation times dependent on dimensions such as image size, channel size (in and out), kernel size, and batch size. We were able to get some actual propagation times by running a test model modified from a model designed for the MNIST dataset, and ran it on torchvision.FakeData datasets for square images of different sizes (256px, 512px, 1024px).



```
class Net(nn.Module):
  def __init__(self):
        super(Net, self).__init__()
        self.layers = nn.ModuleList([
            nn.Conv2d(in_channels = 3, out_channels=32, kernel_size=3, padding=1),
            nn.Conv2d(in_channels = 32, out_channels=32, kernel_size=3, padding=1),
            nn.Conv2d(in_channels = 32, out_channels=32, kernel_size=5, padding=2),
            nn.Conv2d(in_channels = 32, out_channels=32, kernel_size=7, padding=3),

            nn.Conv2d(in_channels = 32, out_channels=64, kernel_size=3, padding=1),

            nn.Conv2d(in_channels = 64, out_channels=64, kernel_size=3, padding=1),
            nn.Conv2d(in_channels = 64, out_channels=64, kernel_size=5, padding=2),
            nn.Conv2d(in_channels = 64, out_channels=64, kernel_size=7, padding=3),

            nn.Conv2d(in_channels = 64, out_channels=128, kernel_size=3, padding=1),

            nn.Conv2d(in_channels = 128, out_channels=128, kernel_size=3, padding=1),
            nn.Conv2d(in_channels = 128, out_channels=128, kernel_size=5, padding=2),
            nn.Conv2d(in_channels = 128, out_channels=128, kernel_size=7, padding=3),

            nn.AdaptiveAvgPool2d((1,1)),
            nn.Linear(128, 10),
            nn.LogSoftmax()
        ])
```

Figure 8. Convolutional layers definitions

We decided to go with a less complicated approach and predict the time using multivariate polynomial curve fitting. In testing curves of different degrees (linear, quadratic, and cubic), we found very similar performance in the predicting of layer-times on the forward and backward pass, the figure below compares predicted and actual propagation times from training the sample model (which had 9 different convolutional layers, all with different parameters) on different datasets (for different image sizes). Each sample simply had 5 dimensions of data that indicated the parameters of the convolutional layer as described earlier. Given these results on a linear polynomial, it is safe to say that the splitting will be accurately informed of the propagation-time of any convolutional layers it may have.

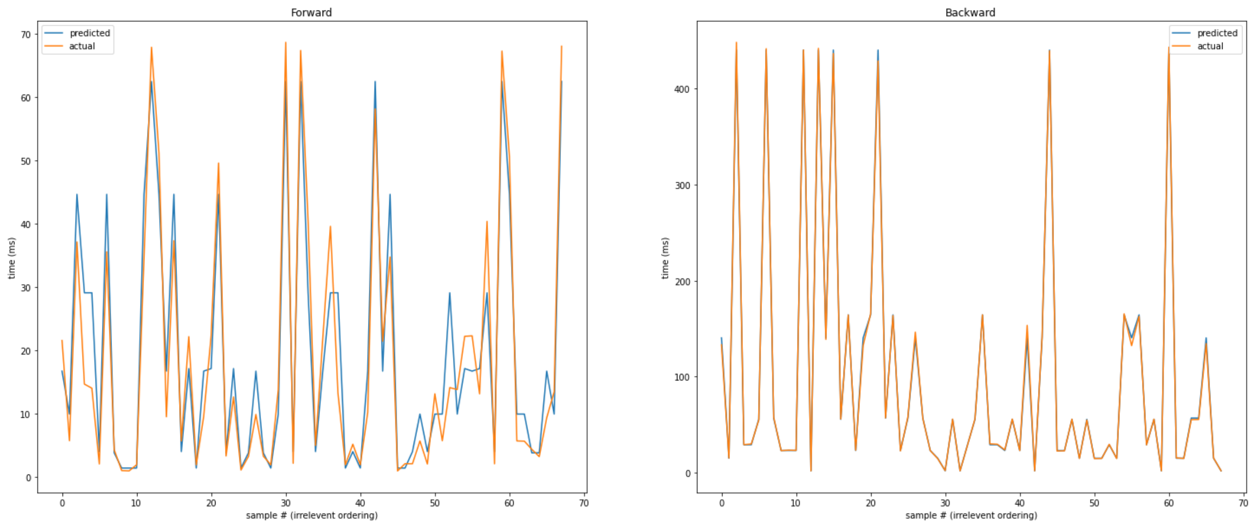


Figure 9. Curve fitting for forward/backward time predictions

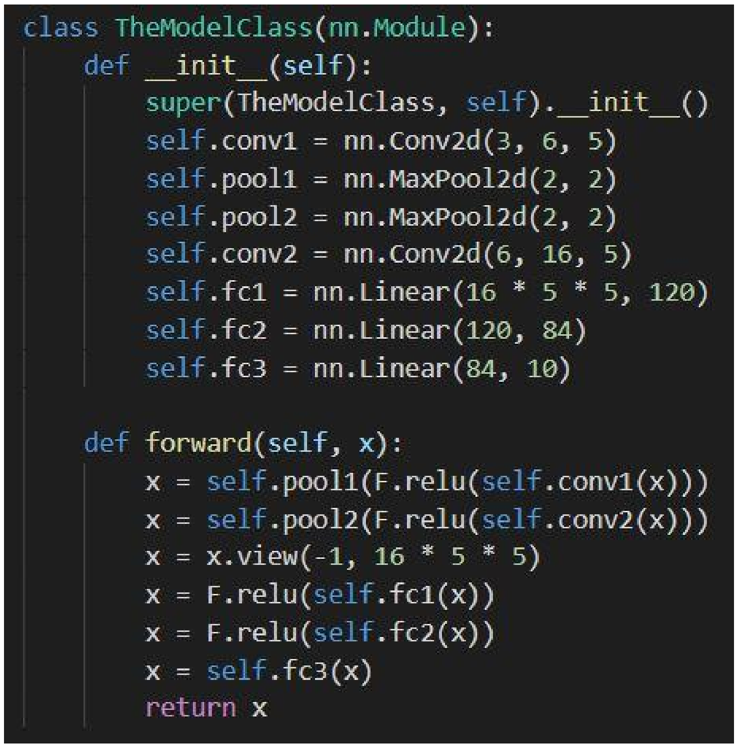
Network Representation

Purpose

To be able to analyze and split a model, it must be representable in an appropriate data structure. The needed information is the names and types of layers/blocks and the connections between them. PyTorch does not provide such functionality. There are some tools that come close to what's needed (TensorBoard and TorchViz), but as shown in the comparison below, they do not carry the needed information for effective analysis and splitting. Therefore, a new method that represents a network in the described terms must be developed. The task becomes more complicated by introducing complex designs where layers are interconnected (layer has more than one connection).

Design

1. **PyTorch model**: Starting with any PyTorch model, we need to extract the graph representation of its layers/blocks and the connections between them. Below is a simple convolutional neural network to show the different steps of the network representation design.

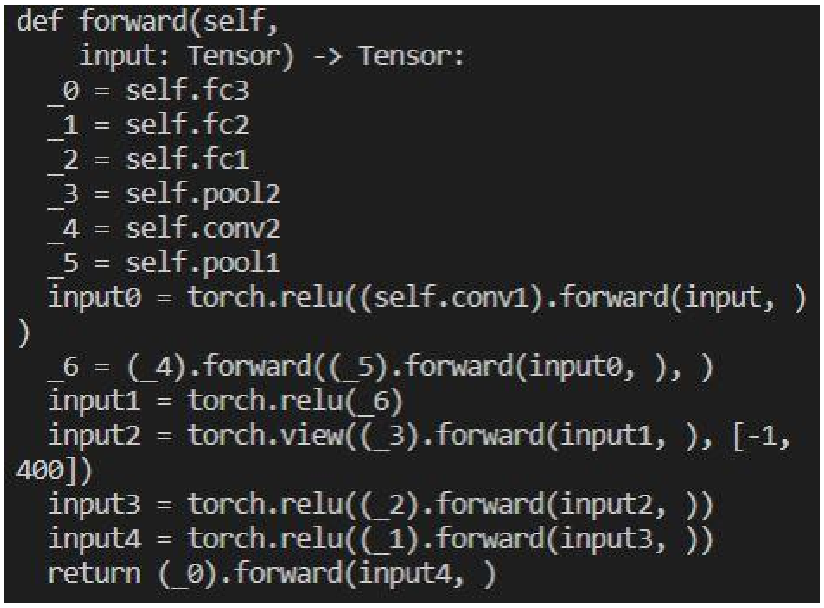


```python
class TheModelClass(nn.Module):
    def __init__(self):
        super(TheModelClass, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Figure 10. Example CNN definition

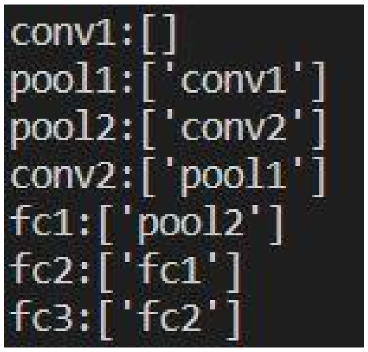
2. **Trace output:** PyTorch has a function torch.jit.trace that returns a string representation of the forward function of a model. It works by passing a data instance (or dummy data) into the model and tracing it through the forward function. Below is the output of the tracing function for the example CNN:



```
def forward(self,
    input: Tensor) -> Tensor:
  _0 = self.fc3
  _1 = self.fc2
  _2 = self.fc1
  _3 = self.pool2
  _4 = self.conv2
  _5 = self.pool1
  input0 = torch.relu((self.conv1).forward(input, )
)
  _6 = (_4).forward((_5).forward(input0, ), )
  input1 = torch.relu(_6)
  input2 = torch.view((_3).forward(input1, ), [-1,
400])
  input3 = torch.relu((_2).forward(input2, ))
  input4 = torch.relu((_1).forward(input3, ))
  return (_0).forward(input4, )
```

Figure 11. Torch.jit.trace output

3. **Parsing:** a string is an inconvenient representation for analyzing and splitting the network. While it might be helpful for manual review of a model, it does not help in performing automated processes. Therefore, the string is parsed to extract information and store it in an adjacency list. Each layer in the network is associated with a list to indicate the inputs for that specific layer. The adjacency list below represents the example CNN:



```
conv1:[]
pool1:['conv1']
pool2:['conv2']
conv2:['pool1']
fc1:['pool2']
fc2:['fc1']
fc3:['fc2']
```

Figure 12. Adjacency list for CNN

4. **Graph representation:** from the adjacency list, the network can be visualized for better clarity. While this step is not necessary for the automated tasks, it is very useful to confirm the results of the network representation. Using python graphing packages like graphviz can visualize the network structure by slightly manipulating the adjacency list. Below is the resulting graph from the example CNN:

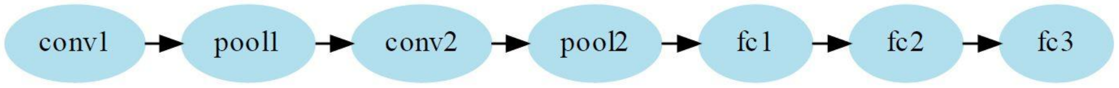
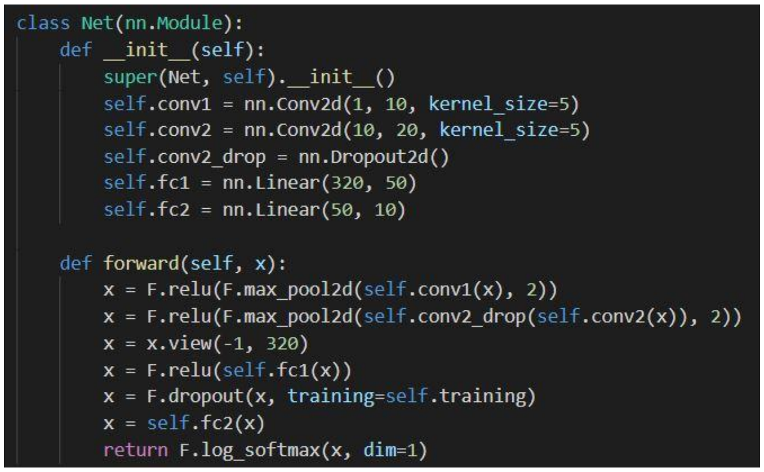


Figure 13. Graph representation of example CNN

Validation

To validate the results of the design, it was tested on multiple network structures to see the resulting graphs and confirm the results. Below are the resulting graphs for multiple networks:

1. Simple CNN



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Figure 14. Definition of simple CNN model

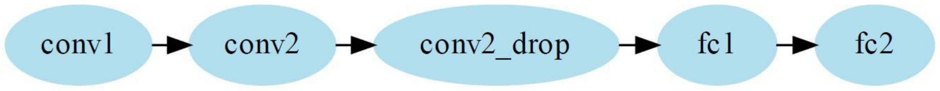
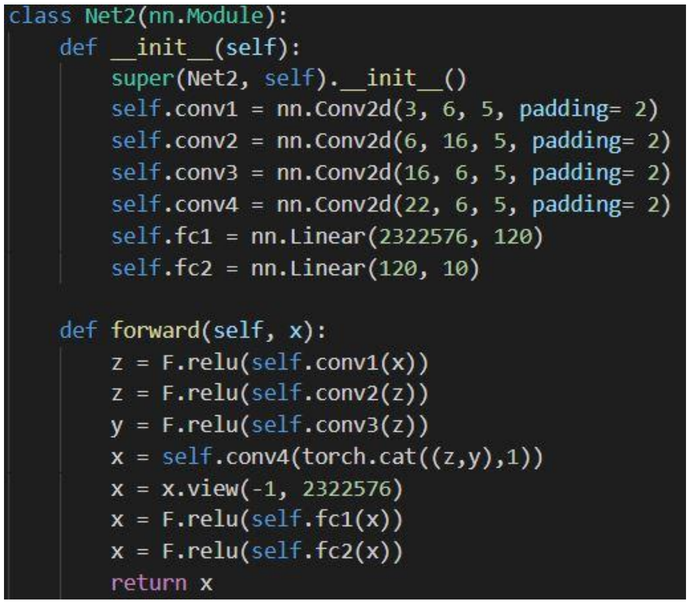


Figure 15. Graph representation of simple CNN model

Going through the forward function, the layer conv1 is the input of conv2, and conv_2 is the input of conv2_drop and so on with the rest of the linear layers. Running the code gives an accurate representation of the specific network showing all the connections and layer names. The representation carries out just the right amount of information that is needed to perform our analysis and later the splitting.

2. Interconnected CNN:



```
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5, padding= 2)
        self.conv2 = nn.Conv2d(6, 16, 5, padding= 2)
        self.conv3 = nn.Conv2d(16, 6, 5, padding= 2)
        self.conv4 = nn.Conv2d(22, 6, 5, padding= 2)
        self.fc1 = nn.Linear(2322576, 120)
        self.fc2 = nn.Linear(120, 10)

    def forward(self, x):
        z = F.relu(self.conv1(x))
        z = F.relu(self.conv2(z))
        y = F.relu(self.conv3(z))
        x = self.conv4(torch.cat((z,y),1))
        x = x.view(-1, 2322576)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return x
```

Figure 16. Interconnected CNN definition

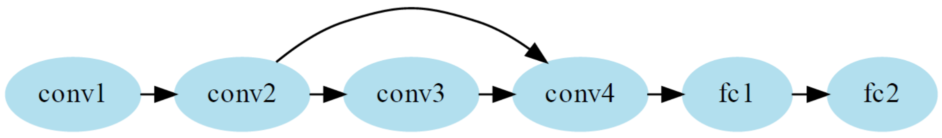
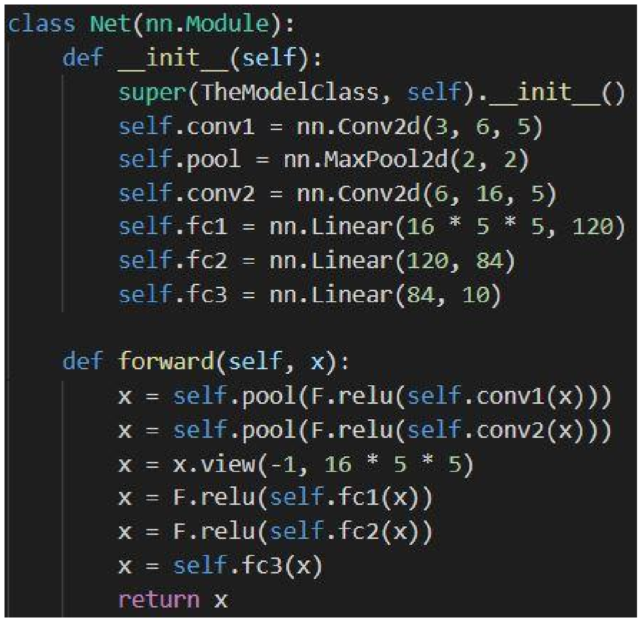


Figure 17. Graph representation of interconnected CNN

This is an interconnected model, where a layer can take input from two different layers. The reason this is happening is that the input for layer conv4 is the concatenation of layers conv2 and conv3. The graph represents this concatenation by adding the arrow from layer conv2 to layer conv4.

3. Duplicate Layer:



```python
class Net(nn.Module):
    def __init__(self):
        super(TheModelClass, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Figure 18. Network with duplicate layer

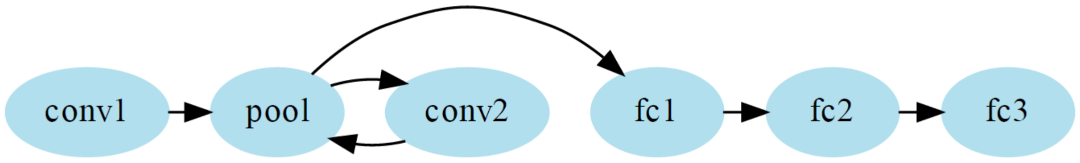


Figure 19. Incorrect graph representation of duplicate layer

The problem with such a graph is the cycle created between layers pool and conv2. The output of layer conv1 is the input of the layer pool. The output of the layer pool is input for conv2, and output of conv2 as input for the layer pool. This sequence is logically and technically viable; however, it creates a cycle in the graph that makes the graph somewhat inefficient to use for analysis or splitting. A proposed solution is to automatically duplicate and rename the problematic layer to resolve any cycles. The results of such a fix should look something like the second graph where layer pool is replaced by pool1 and pool2.

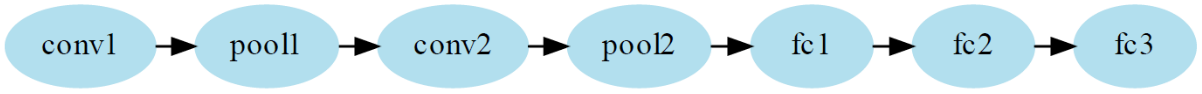
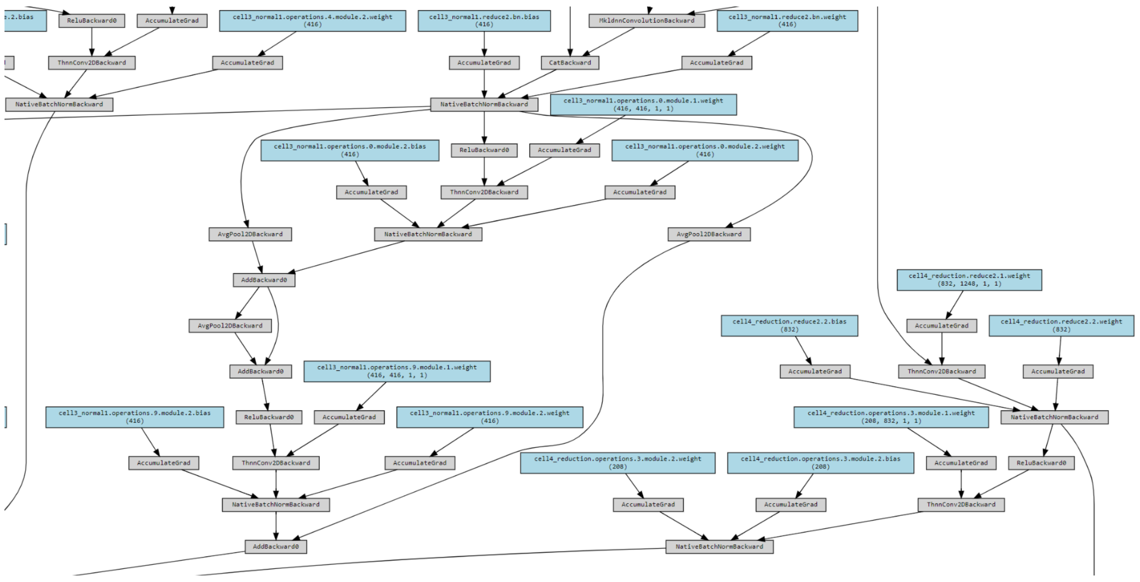


Figure 20. Possible fix for representing duplicate layer

Solution Comparison

As mentioned earlier, there are packages that provide a graph representation of a network. For instance, for the AmoebaNet model, the TorchViz library generates the following graph:



ReluBackward0
AccumulateGrad
cell13_normal1.operations.4.module.2.weight (416)
cell13_normal1.reduce2.bn.bias (416)
MkldnnConvolutionBackward
cell13_normal1.reduce2.bn.weight (416)
ThnnConv2DBackward
CatBackward
NativeBatchNormBackward
cell13_normal1.operations.0.module.1.weight (416, 416, 1, 1)
cell13_normal1.operations.0.module.2.bias (416)
cell13_normal1.operations.0.module.2.weight (416)
AvgPool2DBackward
AddBackward0
cell13_normal1.operations.9.module.1.weight (416, 416, 1, 1)
cell13_normal1.operations.9.module.2.bias (416)
cell13_normal1.operations.9.module.2.weight (416)
cell14_reduction.reduce2.1.weight (832, 1248, 1, 1)
cell14_reduction.reduce2.2.bias (832)
cell14_reduction.reduce2.2.weight (832)
cell14_reduction.operations.3.module.1.weight (208, 832, 1, 1)
cell14_reduction.operations.3.module.2.weight (208)
cell14_reduction.operations.3.module.2.bias (208)

Figure 21. TorchViz visualization of AmoebaNet

This is only a portion of the full graph. There are multiple issues with such representation: a) No one-to-one mapping with layers. b) Introduces nodes for weights and biases c) Shows graph for backward propagation. d) No block-level abstraction.

On the other hand, we can notice a much cleaner and simpler representation generated by the new design. This design specifically includes the right amount of information needed. For the same model, which is AmoebaNet, the graph representations on the block level of 6- and 18-layer AmoebaNet models are generated. Almost every block has two inputs from the two previous blocks. A block is a unit that has a lot of layers in it. The new representation is very convenient for splitting, because it simplifies this complicated model, and it eliminates a lot of internal connections. Still, if needed the design can recursively run on each block to find the different layers and connections within the blocks providing more options on how to split the model.

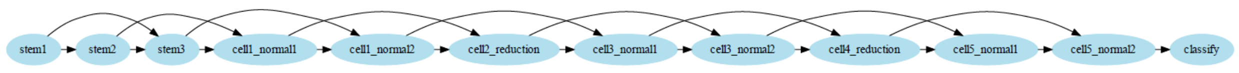


Figure 22. Graph of 6-layers AmoebaNet (block level)

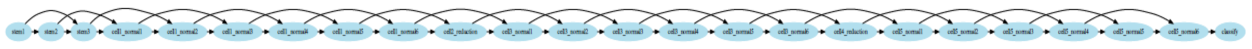


Figure 23. Graph of 18-layers AmoebaNet (block level)

CPU Offloading

As we saw in the area of Natural Language Processing (NLP), the size of deep learning models is becoming larger and larger each year, Bert-large (0.3B), GTP-2 (1.5B), Megatron-LM (8.3B), T5 (11B) [3]. Training a large deep learning model becomes an unstoppable trend to improve the accuracy, but training a model with a billion or trillions number of parameters is daunting.

In the past, the traditional approach to train such large deep learning models often involved the support from either data parallelism or model parallelism. However, those approaches have the fundamental requirement to fit these models into a limited device memory, and this requirement sometimes can prevent the training of larger batch size or deeper neural networks. The CPU-Offloading strategy enables the potential to train a large deep learning model with over billions of parameters by offloading the memory pressure from GPU to CPU. Since the amount of memory available in the CPU can be much larger than the GPU, we will be able to accommodate a larger model with more complicated design or structure.

By exploiting the PyTorch library, we discovered two possible ways to enhance the performance of CPU-Offloading: 1) Pin_memory. According to the official PyTorch documentation, “For data loading, passing pin_memory=True to a DataLoader will automatically put the fetched data Tensors in pinned memory, and thus enables faster data transfer to CUDA-enabled GPUs” [1]; 2) Non-blocking. Once we pinned a tensor or storage with pin_memory, we can use asynchronous GPU copies by passing an additional non_blocking=True argument to a to() or a cuda() call, which will allow the overlap of data transfers with computation [2].

CPU-Offloading provides the potential to offload the memory to the CPU and allows us to train some out-of-core models that require much larger memory than what is available in a single GPU. However, this approach is not free and requires certain cost on the I/O communication. Beside the baseline model without any CPU-offloading support, there are four different comparative experimental groups we have conducted to discover the factors that can pose effect on CPU-Offloading and the most efficient way to apply the CPU-Offloading during training: 1) CPU-Offloading without any optimization; 2) CPU-Offloading with pin_memory; 3) CPU-Offloading with non-blocking; 4) CPU-Offloading with pin_memory and non-blocking. For evaluating the performance of these four different approaches, we applied them on four different sizes of deep learning models: VGG19, AlexNet, ResNet50, InceptionV3. The result of these four different pre-trained model are demonstrated in the table 1- 5:

Approaches	Total Time(sec)	Best Val ACC	CtoG (mean)	CtoG (max)	CtoG (min)	CtoG (medium)	GtoC (mean)	GtoC (max)	GtoC (min)	GtoC (medium)
Baseline on GPU	94.4539	0.9739								
Naïve CPU-offloading	179.9194	0.9804	0.1069	0.1450	0.0646	0.1173	0.4836	0.5842	0.4358	0.4825
with pin-memory	194.5803	0.9739	0.1131	0.1414	0.0651	0.1184	0.4816	0.6069	0.4293	0.4805
non-blocking	179.8007	0.9739	0.0863	0.1329	0.0628	0.0682	0.4880	0.6082	0.4291	0.4891
pin-memory and non-blocking	190.4785	0.9804	0.0713	0.1310	0.0631	0.0657	0.4917	0.6061	0.4429	0.4931

Table 1. CPU-Offloading evaluation of vgg19 on RI2 sky-k80 (139,578,434 params in total)

Approaches	Total Time(sec)	Best Val ACC	CtoG (mean)	CtoG (max)	CtoG (min)	CtoG (medium)	GtoC (mean)	GtoC (max)	GtoC (min)	GtoC (medium)
Baseline on GPU	30.0673	0.9673								
Naïve CPU-offloading	150.5116	0.9739	0.0678	0.0935	0.0584	0.0644	0.2148	0.2672	0.1952	0.2082
with pin-memory	166.7242	0.9673	0.0640	0.0993	0.0591	0.0615	0.2346	0.2758	0.1966	0.2378
non-blocking	153.9791	0.9608	0.0665	0.0916	0.0575	0.0628	0.2159	0.2644	0.1968	0.2089
pin-memory and non-blocking	160.9983	0.9608	0.0648	0.0939	0.0584	0.0630	0.2100:	0.2634	0.1949	0.2068

Table 2. CPU-Offloading evaluation of vgg19 on RI2 bdw-v100 (139,578,434 params in total)

Approaches	Total Time(sec)	Best Val ACC	CtoG (mean)	CtoG (max)	CtoG (min)	CtoG (medium)	GtoC (mean)	GtoC (max)	GtoC (min)	GtoC (medium)
Baseline on GPU	24.1323	0.9085								
Naïve CPU-offloading	42.2092	0.9150	0.0285	0.0383	0.0243	0.0270	0.0762	0.0888	0.0680	0.0735
with pin-memory	47.4363	0.9346	0.0262	0.0384	0.0241	0.0251	0.0804	0.0924	0.0675	0.0811
non-blocking	41.5072	0.9346	0.0276	0.0572	0.0237	0.0261	0.0761	0.0898	0.0679	0.0729
pin-memory and non-blocking	47.5516	0.9216	0.0252	0.0360	0.0235	0.0242	0.0816	0.0933	0.0683	0.0812

Table 3. CPU-Offloading evaluation of AlexNet on RI2 bdw-v100(57,012,034 params in total)

Approaches	Total Time(sec)	Best Val ACC	CtoG (mean)	CtoG (max)	CtoG (min)	CtoG (medium)	GtoC (mean)	GtoC (max)	GtoC (min)	GtoC (medium)
Baseline on GPU	26.1702	0.9477								
Naïve CPU-offloading	113.5793	0.9608	0.0250	0.0539	0.0216	0.0238	0.0461	0.0714	0.0292	0.0423
with pin-memory	110.0957	0.9608	0.0252	0.0430	0.0221	0.0229	0.0463	0.0721	0.0311	0.0445
non-blocking	113.6041	0.9608	0.0205	0.0323	0.0170	0.0197	0.0474	0.0771	0.0307	0.0446
pin-memory and non-blocking	109.3674	0.9608	0.0208	0.0400	0.0172	0.0183	0.0457	0.0746	0.0310	0.0425

Table 4. CPU-Offloading evaluation of ResNet50 on RI2 bdw-v100 (23,512,130 params in total)

Approaches	Total Time(sec)	Best Val ACC	CtoG (mean)	CtoG (max)	CtoG (min)	CtoG (medium)	GtoC (mean)	GtoC (max)	GtoC (min)	GtoC (medium)
Baseline on GPU	32.2195	0.9281								
Naïve CPU-offloading	124.2807	0.9477	0.0352	0.0471	0.0317	0.0339	0.0555	0.0890	0.0390	0.0523
with pin-memory	120.2433	0.9346	0.0358	0.0575	0.0317	0.0328	0.0562	0.0990	0.0393	0.0534
non-blocking	123.1403	0.9477	0.0280	0.0394	0.0241	0.0271	0.0605	0.0973	0.0407	0.0602
pin-memory and non-blocking	119.2649	0.9281	0.0283	0.0620	0.0241	0.0254	0.0574	0.1029	0.0402	0.0532

Table 5. CPU-Offloading evaluation of InceptionV3 on RI2 bdw-v100 (24,348,900 params in total)

From the above result, there are several key observations had discovered, and we provided some analysis below to explain those finding:

1) By comparing the result in Table 1 and Table 2, we can see that the total amount of time taken by the CPU-Offloading was affected by the underlying hardware architecture. The VGG19 that was trained on sky-k80, skylake microarchitecture with NVIDIA TESLA K80, takes 94.4358 sec on Baseline model and around 170-190 sec for different optimized approaches, and the VGG19 that was trained on bdw-v100, broadwell microarchitecture with NVIDIA V100, takes 30.0673 sec on Baseline model and around 150-170 sec for different optimized approaches. TESLA K80 belongs to the Nvidia Kepler architecture that was fabricated/manufactured by TSMC on 28nm process and was introduced in 2012 [4], and NVIDIA V100 is based on Volta microarchitecture that was fabricated/manufactured by TSMC on a 12nm process and was introduced in 2017 [5]. The Nvidia Volta microarchitecture superseded Kepler over three generations, and, therefore, the total training time of the baseline model that was trained on VGG19 on

bdw-v100 is roughly three times faster than sky-k80. However, this improvement is not significant after applied CPU-Offloading, and this signifies that the majority of time cost for applying CPU-Offloading is primarily contributed by the I/O communication, in moving data between CPU and GPU.

2) In Table 3, we applied CPU-Offloading on AlexNet, a simple DL model with 8 layers. The result showed that AlexNet has produced the lowest total training time on the baseline group, and this improvement is much more significant on groups that applied with certain CPU-Offloading techniques. The AlexNet is a simple model and does not have a lot of layers as other models such as ResNet or InceptionNet, but the total number of parameters produced by the AlexNet (~57M params) still about twice the size of ResNet or InceptionNet (23-24M params), which coerce the size of the tensor on each layer to be vast and condensed. Performing CPU-Offloading is actually achieved by moving tensor between CPU and GPU. Since the number of tensors that are required to put on the I/O bus were reduced, the amount of time taken for I/O communication is also reduced. Therefore, we believe that performance of CPU-Offloading can be greatly improved by compressing the total number of parameters needed to transfer a model into a smaller number of condensed tensors.
3) The results in Table 4 and 5 were trained on ResNet and InceptionV3, which are comparatively similar in terms of the model size and the number of layers contained in the model. There is an important finding that we observed in the result. Among all the experiments that we conducted across all five tables, we noticed the total training time is only reduced after applying the CPU-Offloading optimization techniques on these two models. Therefore, we believe the optimizations techniques (pin_memory or/and non_blcoking) are helpful only for the models that contain more number of layers and parameters.

Conclusion

In this project, we worked on the automatic splitting of DNN for model-parallelism and explored CPU-Offloading to decrease memory requirement on a single processing unit like CPU or GPU. We divided model splitting into 2 tasks: 1) Estimating the computation time for each layer to equally divide the total computation among different processes and 2) Finding connections between layers to insert send/recv communication primitives between processes (network representation). We evaluated polynomial regression to estimate computation time for the convolution layer using input size, kernel size, in channels, and out channels. We developed a module to find connections between layers in a PyTorch model, which is impossible using existing methods and libraries. Further, we explored the feasibility of CPU-Offloading in model-parallelism and evaluated the performance of different CPU-Offloading approaches.

In the future, we would like to integrate network representation with the analytical model and develop new analytical models for other layers like pooling, ReLU, and dense layers. We further want to overlap computation with CPU-Offload to make it efficient and improve the performance of model-parallelism and GEMS.

References

[1] “TORCH.UTILS.DATA in PyTorch”, 2019, https://pytorch.org/docs/stable/data.html#torch.utils.data.get_worker_info
[2] “CUDA SEMANTICS in PyTorch”, 2019, https://pytorch.org/docs/stable/notes/cuda.html
[3] Rajbhandari, Samyam, et al. "Zero: Memory optimization towards training a trillion parameter models." arXiv preprint arXiv:1910.02054 (2019), https://arxiv.org/pdf/1910.02054.pdf
[4] Kepler (microarchitecture) in Wikipedia, https://www.wikiwand.com/en/Kepler_(microarchitecture)
[5] Volta (microarchitecture) in Wikipedia, https://www.wikiwand.com/en/Volta_(microarchitecture)
[6] Narayanan, Deepak, et al. "Memory-efficient pipeline-parallel dnn training." arXiv preprint arXiv:2006.09503 (2020).
[7] A. Jain, *et al.*, "GEMS: GPU-Enabled Memory-Aware Model-Parallelism System for Distributed DNN Training," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Atlanta, GA, US, 2020 pp. 621-635. doi: 10.1109/SC41405.2020.00049
[8] Pudipeddi, Bharadwaj et al. "Training Large Neural Networks with Constant Memory using a New Execution Algorithm" arXiv preprint arXiv:arXiv:2002.05645 (2020).
[9] Huang, Yanping, et al. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism" arXiv:1811.06965 [cs.CV] (2019).
[10] Dryden, Nikoli, et al. “Channel and Filter Parallelism for Large-Scale CNN Training” In The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC ’19), https://doi.org/10.1145/3295500.3356207 (2019).