CSE5249 Final Project Report

Name: Zhengqi(Drago) Dong, Tom Ballas, Arpan Jain

1. Introduction

The application of deep learning has garnered considerable attention in many fields of research, such as image classification, speech recognition, autonomous driving, and cancer detection. This can be contributed to not only the attractive property of being able to learn the general feature representations of datasets from scratch but also to massive available datasets and advanced computing powers. With such advantages, the field of deep learning is flourishing in recent decades and has superseded the performance of many traditional machine learning and AI algorithms [2, 3].

The influence of deep learning is pervasive due to its capability in solving many complex tasks while providing state-of-the-art results. However, it faces the challenges of scaling to much larger models and datasets, and the traditional training algorithms are inherently sequential and cannot be trivially parallelized. The current two strategies to exploit the parallelism in learning workload are: 1) Develop parallel and distributed Deep Neural Network (DNN) training models. 2) Develop parallel hardware architecture for DNN [3, 4].

Currently, there are three strategies exploited on distributed and parallel DNN models: 1) Data parallelism: Replicating the model across multiple devices and then distributing the data across the devices. 2) Model parallelism: Instead of partitioning the data, the model will be split among multiple devices, and each device will train a model partition on entirely duplicated data. 3) Hybrid parallelism: Involves integrating the strategies of data and model parallelism, such as GEMS-Hybrid [5]. Data parallelism is the most popular and widely adopted mechanism, but it fails to diminish the memory size beyond what is required for a single image, and therefore is not applicable for very large samples [7]. In many pathological studies, the high-resolution images are required for disease image detection, and the U-Net [6] model is considerably attractive in biomedical segmentation applications. In this project we will introduce an innovative approach for implementing model parallelism on large samples, which is extensible on the U-Net-like architectures.

2. Literature Review

To obtain a better understanding of existing research in distributed DNN models, we conducted a literature review on four relevant papers that successfully implemented distributed approaches. In the first paper [1], the author proposed the Data Parallelism approach that utilizes the optimized communication primitives in Message Passing Interface (MPI) to distribute DNN training across multiple machines/GPUs on TensorFlow. Data parallelism involves replicating a model across multiple devices and then distributing the data across the devices. Figure 1 on the following page illustrates the process of data parallelism. Each device learns the model independently without modifying the standard backpropagation algorithm, and the weights and biases are synchronized/averaged across all devices via allReduced operation.

The allReduced operation is provided by many MPI interfaces, e.g., OpenMPI. Since MPI is specialized for Supercomputers communication interface, the overall communication overhead can be greatly reduced compared to the sockets interface.

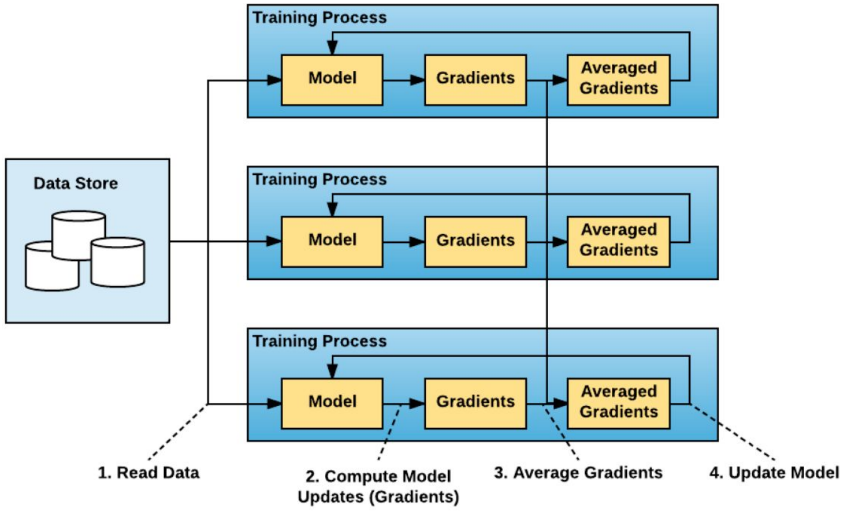

Figure 1: Illustration of basic data parallelism algorithm [8]

As rapid growth of the training dataset and the machine learning model, and the availability of high-performance multi-core GPUs, the performance of primitive versions of TensorFlow cannot satisfy the need of modern business anymore. In the second paper [8], the author introduced a new approach to the distributed deep learning library on TensorFlow, called Horovod. There are several major chances to the old distributed TensorFlow packages. 1) The parameter server approach in the standard distributed TensorFlow packages was replaced by the NCLL's ring-all reduced approach that was originally introduced by Baidu [11]. 2) They converted code into a stand-alone package that is compatible with various releases of TensorFlow. 3) The Horovod library supports the models to run not only on a single GPU, but also on multiple GPUs. 4) The Horovod library support distributed training on various deep learning framework (e.g., PyTorch, MXNet) with minimal modification to the code. As is shown in Figure 2, the less number of communication between worker and parameter server scaled the training efficiency of original distributed TensorFlow on multi-core GPUs.

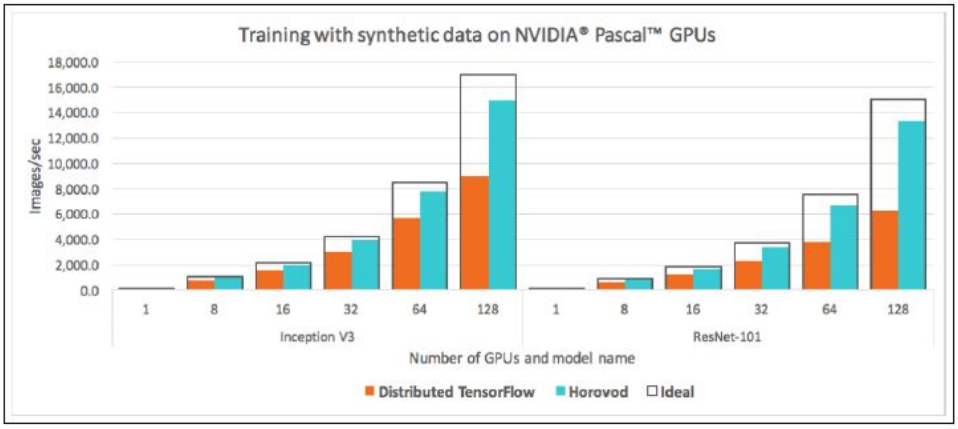

Figure 2: A benchmarks comparison for multi-core GPU scaling performance on TensorFlow [8]

Although approaches like Horovod offer good performance, they must be used for models that reside in the memory of a CPU/GPU. However, there are some larger and deeper models that require more memory than what is available on a single CPU/GPU. This has caused a need for model parallelism which involves distributing a model across multiple GPUs. This means that a layer or multiple layers will be trained on each device. With larger models creating a need for model parallelism approaches, researchers at Google sought to develop an approach for model parallelism to train a large deep learning model. However, one of the major challenges facing them was that traditional model parallelism suffers from under-utilization of GPU resources since at each time only one GPU can perform computations. Figure 3 on the following page illustrates this concept.

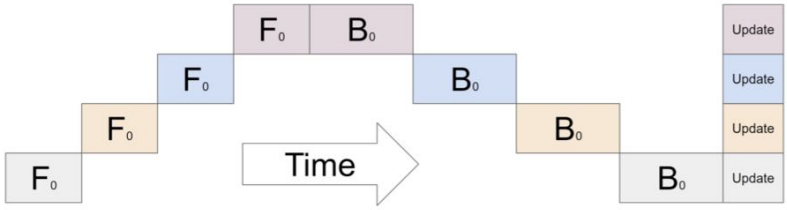

Figure 3: Illustration of the traditional model parallelism approach with both forward (F_0) and backward passes (B_0) on multiple devices (each device is shown as a row). This shows how the resources are underutilized across devices. [9]

To solve this challenge, the researchers wrote an article [9] where they proposed a new approach called pipeline parallelism. As is shown in Figure 4, for pipeline parallelism, the data that is to be trained on each device is split into multiple batches. This allows for a device to train a small batch and pass to the next device for computation, instead of having to train on all the data before the next device can perform training. The researchers compiled their pipeline parallelism approach into a library called GPipe. This approach exhibited success as GPipe was used to train a model that achieved a top-1 accuracy of 84.4% on ImageNet-2012.

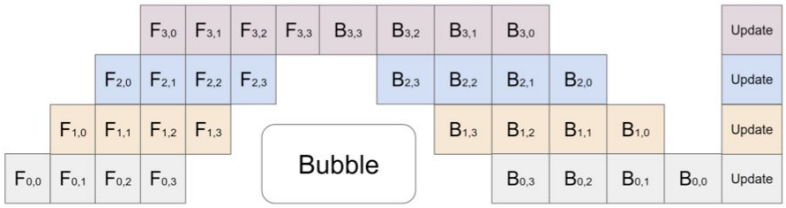

Figure 4: Illustration of the pipeline parallelism approach with both forward ($F_{device,\ batch}$) and backward passes ($B_{device,\ batch}$) on multiple devices (each device is shown as a row). This shows how this approach is able to utilize resources better than traditional model parallelism. [9]

While this approach was successful, it did not exhibit the ability to train state-of-the-art DNNs like ResNet(s) on High-Performance Computing (HPC) systems. In addition, it does not utilize hybrid parallelism which combines data and model parallelism. To solve these challenges, researchers at Ohio State published an article [10] that proposed a hybrid parallelism approach for HPC systems. Their approach called HyPar-Flow utilizes Horovod for data parallelism and implements a pipelining approach for model parallelism. In addition, HyPar-Flow takes advantage of HPC optimizations. HyPar-Flow exhibited success in that it allowed for a hybrid parallelism approach that saw up to a 1.6X speedup over Horovod-based data-parallel training.

However, despite the success of HyPar-Flow, it is unable to operate on multiple GPUs and instead works on multiple CPUs. In addition, the pipeline parallelism approach is limited in performance compared to data parallelism, and the length of the pipeline is limited by the batch size. To mitigate the pipeline parallelism issues and allow for GPU use, researchers at Ohio State published a paper [5] where they propose GEMS (GPU Enabled Memory Aware Model Parallelism System). As was discussed with traditional model parallelism, there is a memory vacuum during forward and backward propagation. Within GEMS, to utilize this memory, two model replicas can be created. The first replica is trained as normal, and the second replica uses the free memory and compute to train in an inverted manner. Following forward and backward passes for both replicas, the parameters of each model are synchronised similar to data parallelism. This entire proposed process is illustrated in Figure 5.

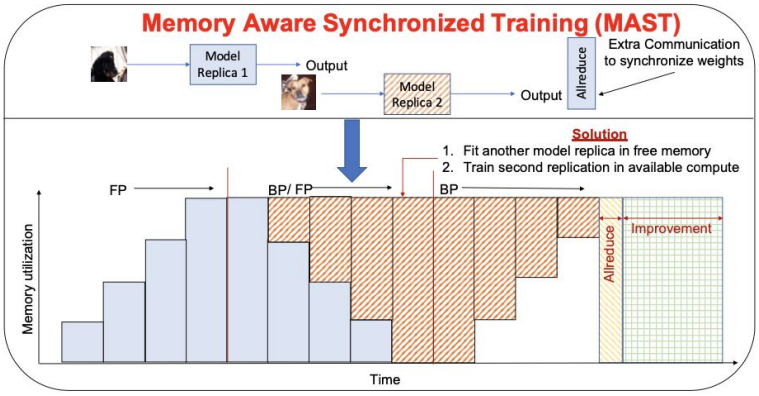

Figure 5: Illustration of GEMS parallelism approach [5]

The GEMS approach exhibited success in that it was used to train a 1000-layer ResNet-1k model with 97.32% scaling-efficiency and reduce the training time for ResNet-110-v2 from seven hours to 28 minutes.

3. Challenges

We now highlight challenges in implementing model parallelism approaches with large images for the UNet architecture.

A. General Model Parallelism Challenges

When implementing model parallelism, partitioning the model and implementing distributed forward and backward passes are two major challenges. Partitioning the model can be challenging as every model is different and the partitioning method needs to support different models. In particular, dealing with skip or residual connections in the DNN topology can be a complicated task. Implementing distributed forward and backward passes is also difficult. This implementation can be especially challenging since existing deep learning frameworks do not provide distributed back propagation implementations.

B. Skip Connections

One of the major challenges for partitioning a DNN for model parallelism is dealing with skip connections in the DNN topology. A skip connection skips some layer or layers in a neural network and feeds the output of one layer as the input to the next layers (instead of only the next one). In a simple sequential neural network, a model can be easily partitioned as each layer only sends an output to one other layer. Therefore, a model can simply be divided into groups of sequential layers to be sent to each GPU, and the output of one partition of layers will provide the input to another partition. However, models with skip connections cannot be partitioned in this manner since some layers must send their output to multiple other layers.

C. Load Balancing

Another challenge when partitioning a DNN is determining how to distribute the layers across the GPUs. This can be particularly challenging because different applications may require a different distribution of layers. Depending on the application, it may be best to balance the load based on memory, compute, or a custom configuration. Providing a flexible load balancing approach that can be customized to the application is necessary for developing a model parallelism approach.

4. Methodologies

To address the challenges discussed in section 3, we performed the following methods to implement a model parallelism approach for UNet.

A. Distributing DNN in PyTorch

In order to implement model parallelism in PyTorch, the DNN layers need to be partitioned to multiple GPUs. In PyTorch, the layers of a neural network can be grouped into Sequential modules. When partitioning, we broke a Sequential module containing all layers into multiple modules of layers. Each module can be passed to a GPU, and the GPU can train those layers. This method was first used to split a simple PyTorch sequential neural network across multiple GPUs. An example illustrating the partitioning of a sequential neural network across four GPUs is shown in section A of Figure 6.

Simply dividing a model evenly by layers is effective for simple sequential models, but this method is unable to deal with skip connections. A popular model that utilizes skip connections is the ResNet model. ResNet utilizes skip connections to allow extremely large DNN's to deal with vanishing gradients. In order to partition ResNet, we took advantage of the fact that skip connections in ResNet only pass over a few layers of the DNN. This enables us to designate the layers passed over by each skip connection as blocks, and the model can be partitioned by these blocks opposed to being partitioned by layers. This assures that all layers within a block are assigned to the same GPU. An example illustrating the concept of partitioning a group of ResNet layers is shown in section B of Figure 6.

Other models such as UNet have far longer skip connections than ResNet. These skip connections can encompass many layers which makes partitioning these models significantly more challenging. We partitioned this model by combining consecutive convolution layers into blocks and vertically partitioning the UNet model by the blocks. Vertical partitioning enables each GPU to consume roughly the same amount of memory. Since the images are cropped as the layers go down the model, if the model was split horizontally, some horizontal partitions would handle much larger sized images which would result in certain GPUs consuming far more memory than others. In addition, the vertical partitioning method used allows long skip connections to be completed as the output of each block can be passed to both the next sequential block and the block across from it when completing a forward pass. Vertical partitioning of the UNet model is illustrated in section C of Figure 6.

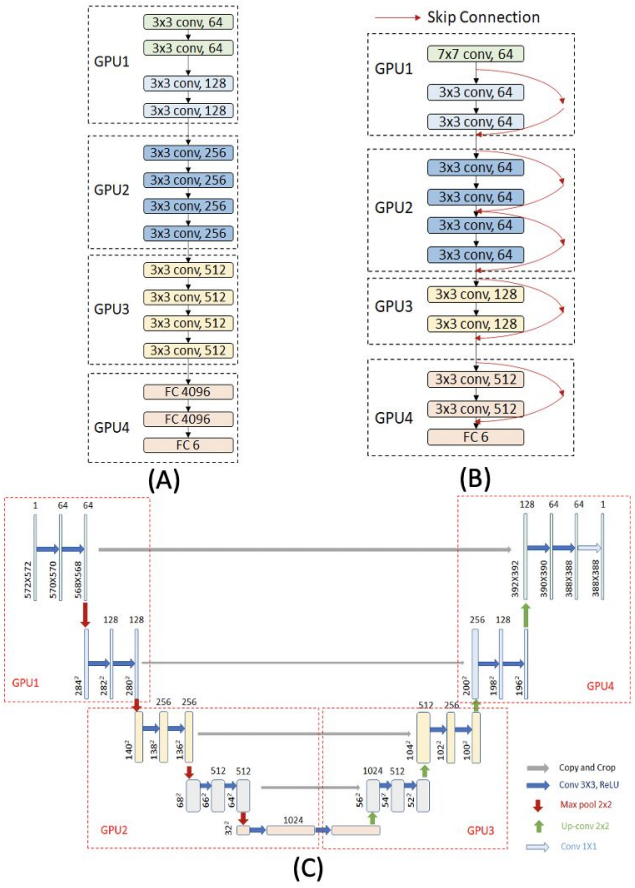

Figure 6: Partitions across 4 GPUs for Basic DNN (A), ResNet (B), and UNet Architectures (C)

When partitioning a DNN, load balancing is one of the key approaches to get better performance. We developed an approach that would partition the DNN to balance either the compute or memory load. Compute load balancing will provide improved performance in memory-aware designs as every GPU will complete the forward and backward passes on their partition in the same amount of time. However, memory-based load balancing will provide better performance when memory consumed by each layer is different and varies a lot. When training a DNN on very-large images, the first few layers will consume more memory compared to the last layers therefore memory-based load balancing is needed to fit a model inside a few GPUs.

B. Use of MPI in Model Parallelism

Message Passing Interface(MPI) is a standardized and portable message passing interface protocol that provides functions on a variety of parallel computers. This protocol standardized the syntax and semantics for many message-passing libraries on many parallelized computers, e.g., MPICH, OpenMPI, Intel MPI libraries. The specification of MPI operations can be roughly categorized to two sets: point-to-point communication and collective communication. Point-to-point communication mechanism enables the transmission of data

between a pair of processes, where the source point will send the data, and the target point will receive. Some exemplary MPI operations that we had used in our project are MPI_Send and MPI_Recv. Point-to-point communication is great for transferring data between two ranks, but it is not effective enough for exchanging the data amount of a group of processes. There are a plethora of collective communication operations available, such as MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Allgather, and etc. MPI_Allreduce is a type of collective communication that allows us to aggregate the reduced result across all processes and distribute the result to all processes. It is an extremely useful operation when implementing the distributed and parallel DNN model, where we need to synchronize the local gradient before updating the weight parameter on each replica [12].

C. Integrating Model Parallelism and UNet

When implementing model parallelism for UNet, we first need to determine how to map the output of a given layer to its next input. For UNet, this is particularly challenging because an output of a layer may be input to multiple other layers. To accomplish this, we implemented this mapping by storing two lists of key value pairs. The first list was used for forward propagation and used a given layer as the key and a list of layers to send the output to as the value. The second list functioned the same way except the mapping was done for backward propagation. In order to communicate between processes, we used asynchronous communication using MPI_Isend and MPI_Irecv. Asynchronous communications allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter. Asynchronous communications are often referred to as non-blocking communications since other work can be done while the communications are taking place" [13]. These operations allow computations and communication to overlap, which leads to improved performance.

As was discussed in Section 4, the U-net model can be partitioned in a vertical fashion, where the first two and last two residual blocks will be separately stored on different GPUs. This vertical partitioning can be performed so that each GPU will train approximately the same number of layers as is shown in Figure 7 below.

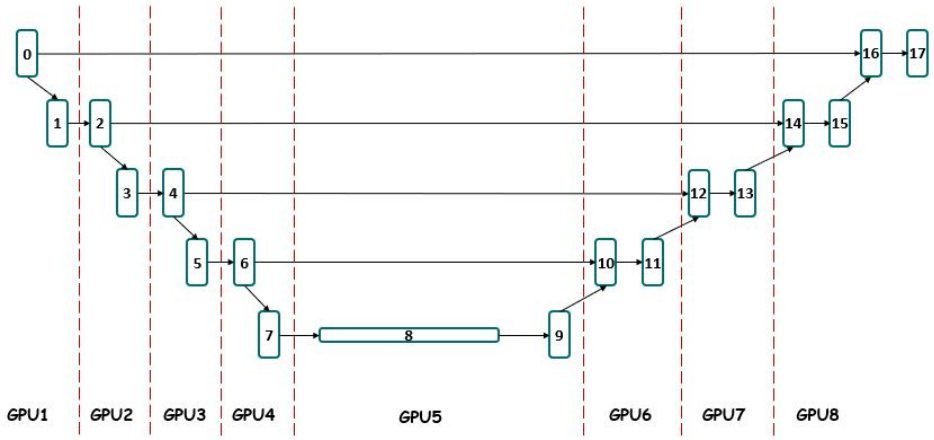

Figure 7: Naive version of model parallelism

In this naive model partitioning approach, we might leave the last four residual blocks in a single GPU, as the number of parameters will decrease as going down to the bottom of the model. However, given the special design of symmetric structure of the U-Net model, the number of parameters are accumulated at the first and last few layers and will lead to a huge memory pressure on a single GPU. Thus, we proposed a more advanced partition method, Model-Aware model parallelism. As shown in Figure 8 below, we leave the last two residual blocks in a single GPU, and bottleneck blocks 6-8 and 9-11 are stored separately on different GPUs. In the following result section, we show that the Model-Aware model parallelism outperformed the naive model partitioning.

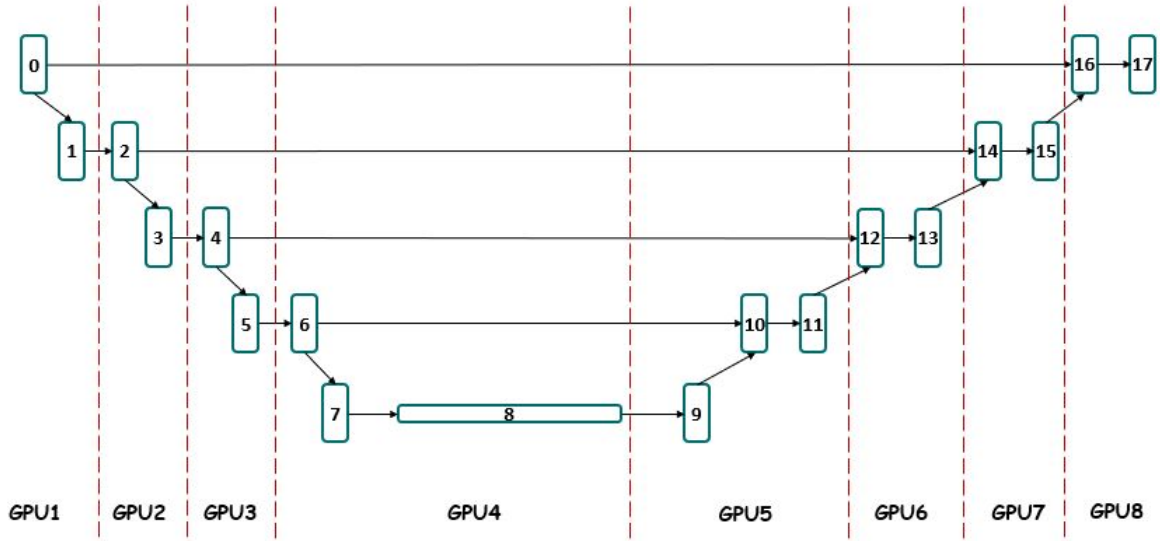

Figure 8: Model-aware model parallelism

5. Results

Our model parallelism approach was evaluated by its ability to train a UNet model on large images. The approach was tested on 8 V100 GPUs with 16 GB HBM (High Memory

Bandwidth). Images with sizes progressing from 64x64 to 2048x2048 pixels were used, and the time in seconds required to perform one forward and backward pass of the model for one image of a given size was recorded. Three different approaches were tested. The first method offered a baseline as it did not involve distributing the model across multiple GPUs. This method resulted in the GPU running out of memory when training on a 640x640 image. The second method involved a naive model parallelism approach. In this approach the UNet model was evenly vertically partitioned across the GPUs. This method was able to train all images up to a 1024x1024 image. The final method tested a model parallelism approach that balanced memory load across GPUs. Memory-based load balancing opposed to evenly partitioning layers is appropriate for the UNet model since the model's first few layers will consume more memory compared to the last layers. This approach was able to train all images up to 2048x2048. In addition to being able to train larger images, the model parallelism approaches also suffered minimal slowdown when compared to the baseline. The results of all tests are shown in Figure 9.

Image Size	Sequential	Model Parallelism (Naive)	Model Parallelism (Model Aware)
64	0.07481	0.19	0.19
128	0.128	0.24	0.236
256	0.346	0.5424	0.49
512	1.3	1.44	1.48
640	OOM	1.9	1.92
768	OOM	2.76	2.81
1024	OOM	OOM	5.96
2048	OOM	OOM	OOM

Figure 9: UNet model training times for one forward and backward pass of DNN

6. Conclusion

In this report, we presented an innovative approach for implementing model parallelism for training UNet models on large images. When evaluating our proposed design, we found that our model parallelism method with memory-based load balancing was able to train UNet models with images over double the size of the maximum image size for non-distributed approaches. In addition, our model parallelism method suffered minimum slowdown compared to the non-distributed approaches. Overall, these results exhibit the capability of model parallelism approaches to train models like UNet with biomedical segmentation applications.

7. References

[1] Vishnu, Abhinav, Charles Siegel, and Jeffrey Daily. "Distributed tensorflow with MPI." arXiv preprint arXiv:1603.02339 (2016).

[2] Zhang, Shuai, et al. "Deep learning based recommender system: A survey and new perspectives." ACM Computing Surveys (CSUR) 52.1 (2019): 1-38.

[3] http://web.cse.ohio-state.edu/~panda.2/5194/slides/overview.pdf

[4] Ben-Nun, Tal, and Torsten Hoefler. "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis." ACM Computing Surveys (CSUR) 52.4 (2019): 1-43.

[5] Arpan Jain, Ammar Ahmad Awan, Asmaa Aljuhani, Jahanzeb Hashmi, Quentin Anthony, Hari Subramoni, Dhabaleswar K. Panda, Raghu Machiraju, Anil Parwani. “GEMS: GPU Enabled Memory Aware Model Parallelism System for Distributed DNN Training.” in SuperComputing 2020.

[6] O. Ronneberger, P.Fischer, & T. Brox (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. In Medical Image Computing and Computer-Assisted Intervention (MICCAI) (pp. 234–241). Springer.

[7] Dryden, Nikoli, et al. "Improving strong-scaling of CNN training by exploiting finer-grained parallelism." 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2019.

[8] Sergeev, Alexander, and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow." arXiv preprint arXiv:1802.05799 (2018).

[9] Huang, Yanping, et al. “GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism”. arXiv preprint arXiv:1811.06965 (2019).

[10] Awan, Ammar Ahmad, et al. "HyPar-Flow: Exploiting MPI and Keras for Scalable Hybrid-Parallel DNN Training using TensorFlow." arXiv preprint arXiv:1911.05146 (2019).

[11] Andrew Gibiansky. Bringing HPC techniques to deep learning. http://research.baidu.com/bringing-hpc-techniques-deep-learning, 2017. [Online; accessed 6-December-2017].

[12] MPI for Python, https://mpi4py.readthedocs.io/en/stable/intro.html

[13] Introduction to Parallel Computing Tutorial, https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial