Analyzing the performance and scheduling of neural networks that use pipeline parallelism

A Practice School Report submitted to
Manipal Academy of Higher Education
in partial fulfilment of the requirement for the award of the degree of

BACHELOR OF TECHNOLOGY

in

Computer Science & Engineering

Submitted by

Aditya Shankar

170905094

Under the guidance of

Dr. Govindarajan Ramaswamy		**Dr. K.N Manjunath**
Professor Dept. of CSA	**&**	**Associate Professor**
Indian Institute of Science (IISc)		**Dept. of Comp. Sci**

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

June 2021

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Manipal
< Date >

CERTIFICATE

This is to certify that the project titled **Analyzing the performance and scheduling of neural networks that use pipeline parallelism** is a record of the bonafide work done by **Aditya Shankar** (*Reg. No. 170905094*) submitted in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology (B.Tech.) in **COMPUTER SCIENCE & ENGINEERING** of Manipal Institute of Technology, Manipal, Karnataka, (A Constituent Institute of Manipal Academy of Higher Education), during the academic year 2021.

Dr. K.N Manjunath
Associate Professor, Computer Science and Engineering

Prof. Dr. Ashalatha Nayak
HOD, CSE Dept.
M.I.T, MANIPAL

Project / Internship Offer Letter

Supercomputer Education and Research Centre

Indian Institute of Science

Prof. R. Govindarajan

Feb. 3, 2021

To whomever it may concern

Subject: Research Internship

This is to certify that Aditya Shankar (Reg no: 170905094, 8th semester, B.Tech Computer Science and Engineering , Manipal Institute of Technology) is pursuing his Bachelor Thesis Internship in the domain of computer systems at the Indian Institute of Science (IISc), Bangalore. He is working on 'Analyzing the performance and scheduling of neural networks that use pipeline parallelism.' The internship commenced on the 1st of January, 2021, and will continue for a period of 5 months (till June 1st, 2021).

Thanking you,

Yours Sincerely,

(R. Govindarajan)

Super Computer Education and Research Centre
Indian Institute of Science
Bangalore - 560 012

Supercomputer Edn. & Res. Centre, Indian Institute of Science, Bangalore, 560 012
Ph: +91 (80) 2360 0654 • Fax: +91 (80) 2360 2648 • Email: govind@iisc.ac.in

Project Completion Letter

Supercomputer Education and Research Centre

Indian Institute of Science

Prof. R. Govindarajan

May 31, 2021

To whomever it may concern

Subject: Research Internship Certificate

This is to certify that Aditya Shankar (Reg no: 170905094, 8th semester, B.Tech Computer Science and Engineering , Manipal Institute of Technology) has completed his Bachelor Thesis Internship in the domain of computer systems at the Indian Institute of Science (IISc), Bangalore. He has successfully completed the project titled 'Analyzing the performance and scheduling of neural networks that use pipeline parallelism', within the specified end date of 1st June 2021.

Thanking you,

Yours Sincerely,

(R. Govindarajan)

Super Computer Education and Research Centre
Indian Institute of Science
Bangalore - 560 012

Supercomputer Edn. & Res. Centre, Indian Institute of Science, Bangalore, 560 012
Ph: +91 (80) 2360 0654 • Fax: +91 (80) 2360 2648 • Email: govind@iisc.ac.in

ACKNOWLEDGMENTS

I express my gratitude to the Director of Manipal Institute of Technology, Dr. D Srikanth Rao, and the Head of the Department for Computer Science and Engineering, Dr. Ashalatha Nayak. Special thanks to my internal advisor, Dr. K.N Manjunath, who provided me with valuable feedback over the course of my project.

I am very grateful to Dr. Govindarajan Ramaswamy for providing me the opportunity to do an internship under his guidance. The constructive feedback and ideas provided by him have proved to be extremely useful, enabling the completion of the project in a timely manner. His patience and methodical approach to problem-solving will always be a source of inspiration for me.

Finally, I would like to thank Amazon Web Services for providing the resources which have become increasingly hard to obtain during the COVID-19 pandemic.

ABSTRACT

The advancement of deep-learning has necessitated the scaling of computer systems to accommodate large models. Solutions which traditionally consisted of model and data parallelism, now include pipeline parallelism too. GPipe and PipeDream are two libraries implementing pipeline parallelism for training neural nets. My internship at IISc has primarily focused on understanding the GPipe implementation, with PipeDream serving as an additional reference.

This project delves into the intricacies of the GPipe library—it's implementation, and performance analysis when training neural nets. A brief introduction to the concepts of data, model, and pipeline parallelism is provided, followed by the implementation of models utilizing GPipe. It also includes the performance study of the library and concludes by identifying potential bottlenecks and areas for improvement.

The results indicate that the performance of GPipe is highly dependent on optimizing the pipeline partitioning and computational load. Contrary to what one might expect, increasing the degree of parallelism does not improve performance beyond a point. Suggestions for areas to improve upon are mentioned, namely the partition algorithm adopted, and degree of pipeline parallelism involved.

GPipe has been implemented as part of the Lingvo framework. TensorFlow Profiler and TensorBoard have been used to conduct the performance analysis.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Scaling up deep neural network capacity has been known as an effective approach for improving model quality for several different machine learning tasks. In many cases, increasing model capacity beyond the memory limit of a single accelerator has required developing special algorithms or infrastructure. These solutions are often architecture-specific and do not transfer to other tasks.

To address the need for efficient and task-independent model parallelism, Google developed GPipe, a pipeline parallelism library that allows scaling any network that can be expressed as a sequence of layers. By pipelining different sub-sequences of layers on separate accelerators, GPipe provides the flexibility of scaling a variety of different networks to gigantic sizes efficiently. Moreover, GPipe utilizes a novel batch-splitting pipelining algorithm, resulting in almost linear speedup when a model is partitioned across multiple accelerators.

Traditionally, there have been two common approaches to scaling neural networks: data-parallelism and model parallelism. Pipeline-parallelism is an extension of these methods and exploits certain advantages of both these techniques. However, every novel idea has scope for improvement, requiring a detailed study of its performance—the aim of this project.

The existing literature provides a brief analysis of GPipe, but does not give detailed information on the scheduling and parallelization limits. This project gives a detailed description of the implementation of GPipe and its performance, highlighting bottlenecks and areas for performance enhancement.

Upcoming sections cover the related background work, the methodology adopted, its implementation, analysis of the results, and the future scope of this project.

CHAPTER 2
BACKGROUND THEORY / LITERATURE REVIEW

As DNNs have become more widely developed and used, model sizes have grown to increase effectiveness—models today have tens to hundreds of layers totalling 10–20 million parameters. Such growth not only stresses the already time and resource-intensive DNN training processes, but also causes the commonly used parallelization approaches to break down. The most common approach is *data parallelism*, where the DNN model is replicated on multiple worker machines, with each worker processing a subset of the training data. Weight updates computed on individual workers are aggregated to obtain a final weight update that reflects updates across all inputs. The amount of data communicated per aggregation is proportional to the size of the model. Although data-parallel training works well with some popular models that have high computation-to-communication ratios, two important trends threaten its efficacy—First, growing model sizes increase per-aggregation communication. Second, rapid increases in GPU compute capacity further shift the bottleneck of training towards communication across models.

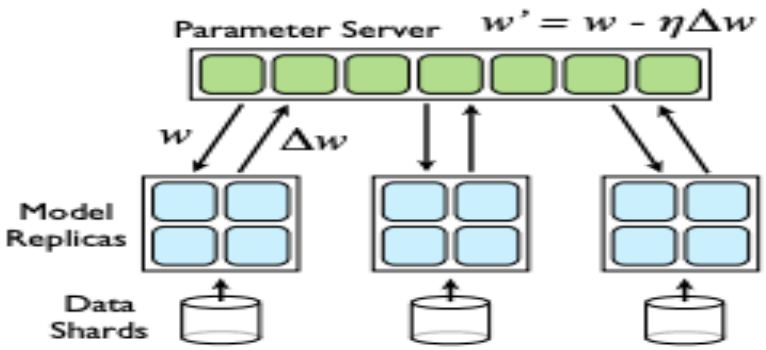


Figure 2.1: Data Parallelism

Another approach to distributed training, *model parallelism,* is used traditionally for models that are too large to keep in a worker's memory or cache during training. Model-parallel training involves partitioning the model among workers such that each worker evaluates and performs updates for only a subset of the model's parameters. However, even though model parallelism enables training of very large models, traditional model parallelism can lead to

severe under-utilization of compute resources since it either actively uses only one worker at a time (if each layer is assigned to a worker) or cannot overlap computation and communication (if each layer is partitioned). In addition, determining how best to partition a DNN model among workers is a challenging task even for the most experienced machine learning practitioners, often leading to additional inefficiency.

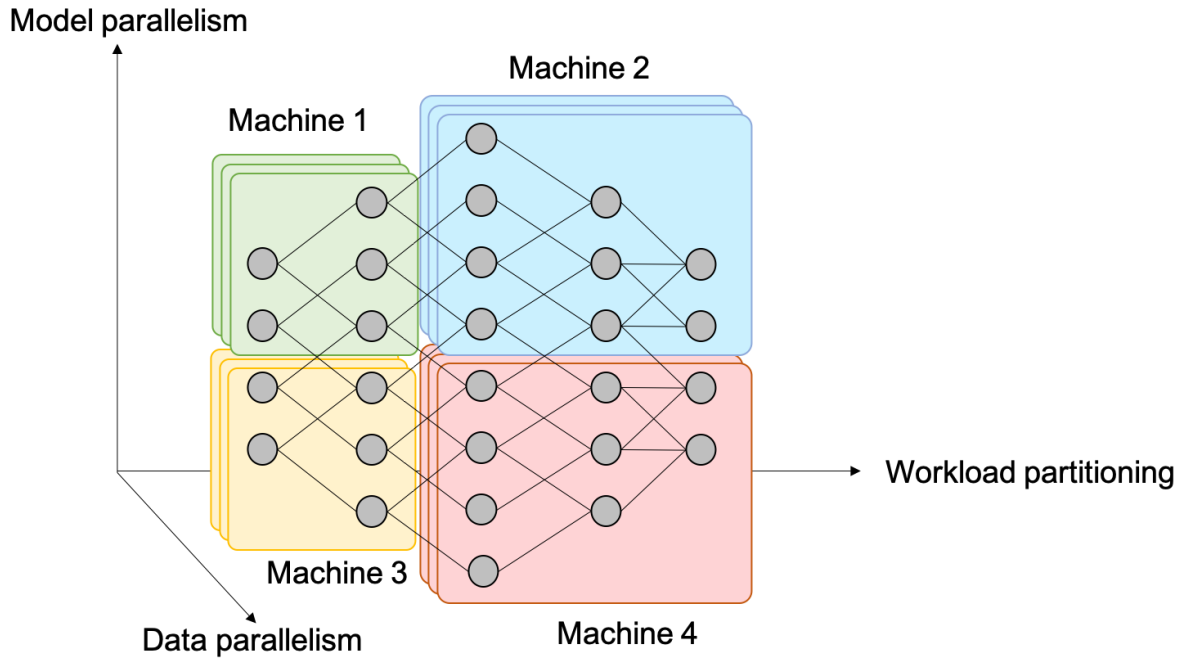


Figure 2.2: Model parallelism

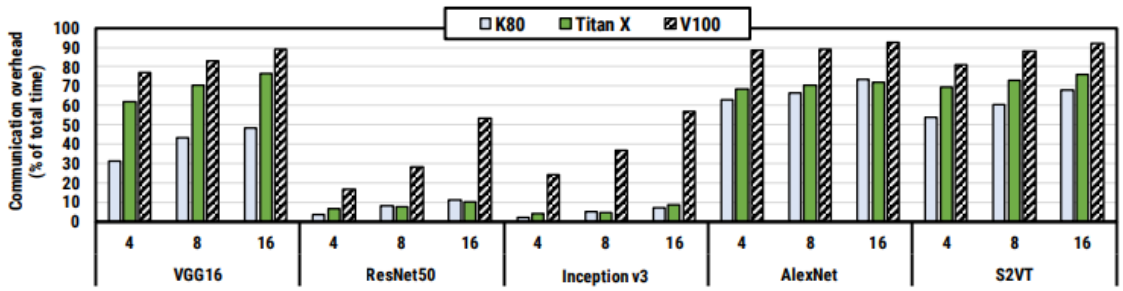


Figure 2.3: Communication costs as percentage of total time

Libraries such as GPipe and PipeDream make use of Pipeline parallelism [1,2] by splitting a model's layers into partitions and assigning each partition to a device or a set of devices. GPipe partitions a minibatch into smaller micro-batches and pipelines these across the GPUS, leading to increased utilization and restricting cross-device communication to the partition boundaries

alone. In addition, GPipe re-computes activations during backprop, to avoid retaining the activations in memory.

GPipe and PipeDream explore novel techniques for improving the scaling of neural networks. However, further research is necessary to improve these frameworks, requiring an understanding of the libraries' implementation. Analysis of details such as the memory usage, cross device communication costs, layer partitioning costs, etc, is required to identify the bottlenecks in the performance. Once this is done, techniques to improve these frameworks can be explored. This project provides a detailed performance analysis of GPipe, to ultimately help improve neural network training.

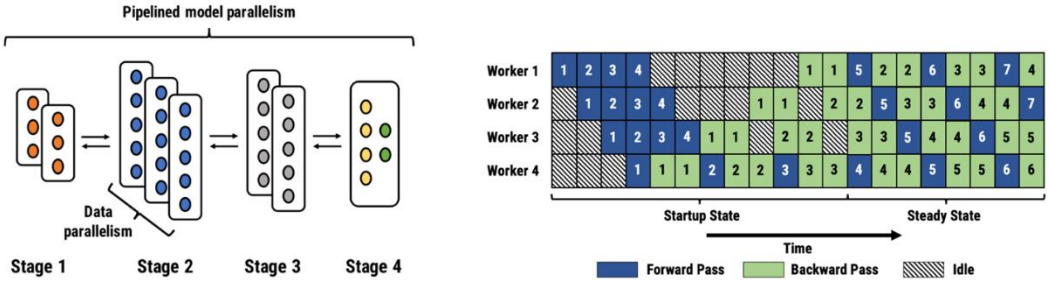


Figure 2.4: Pipeline parallelism in PipeDream (numbers represent minibatches)

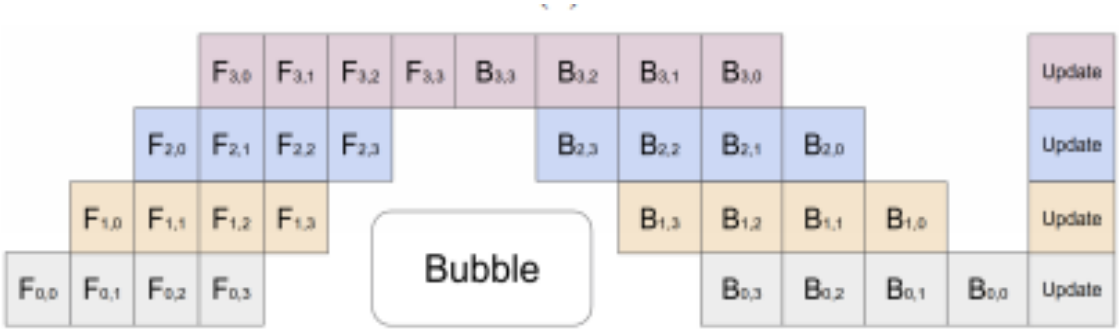


Figure 2.5: Pipeline parallelism in GPipe

CHAPTER 3
METHODOLOGY

3.1 METHODOLOGY

This section provides details on the methodology used to carry out the project. It explains the basic flow of tasks, including the implementation details of GPipe. At the end, a summary of the hardware and software requirements is provided along with a summary of tasks carried out throughout the project.

The following block diagram explains the flow of tasks:

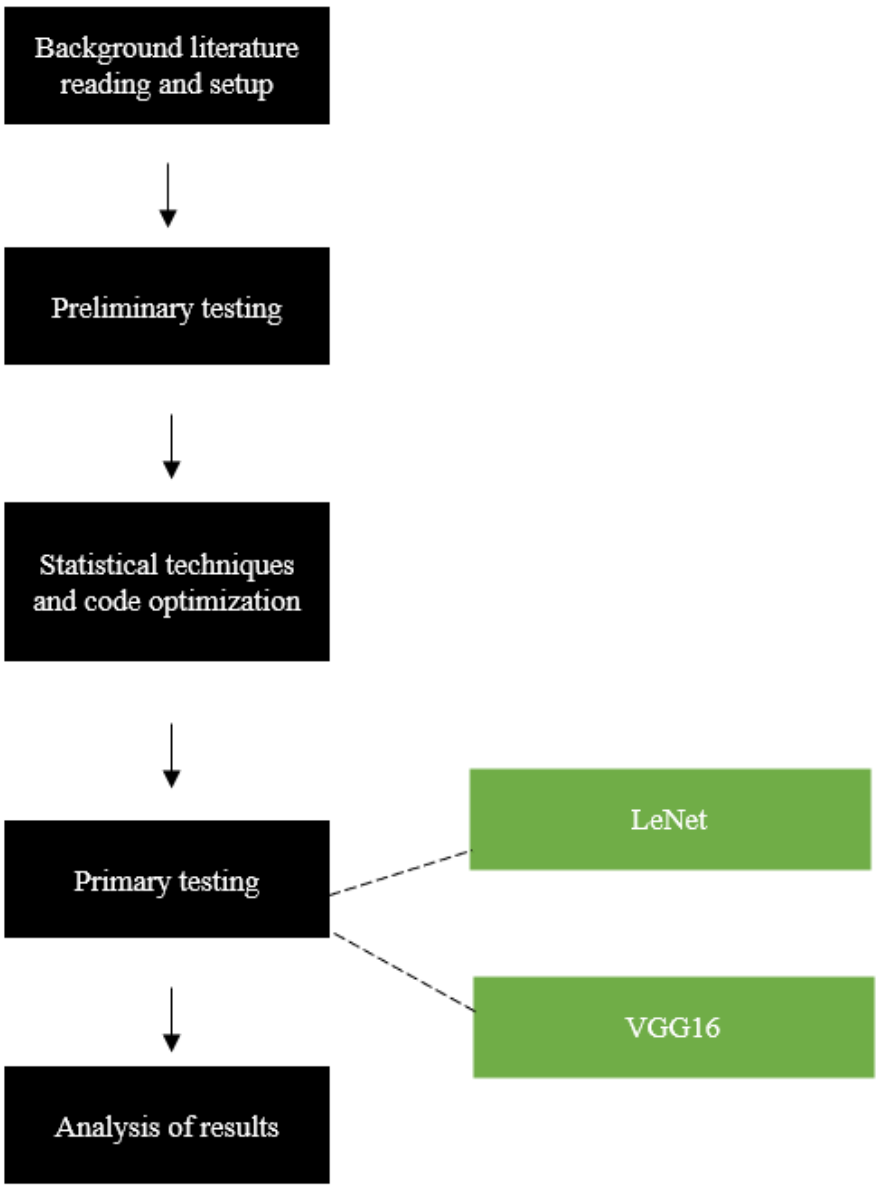


Figure 3.1: Block diagram of methodology

Background literature reading and setup: GPipe [2] and PipeDream [1] explain the concept of pipeline parallelism. SpecTrain [3] and XPipe [4] address some of the drawbacks existing in GPipe and PipeDream. They introduce additional algorithms to avoid problems such as weight stashing in PipeDream.

It was necessary to work with existing implementations of the GPipe and PipeDream. GPipe is open-sourced as part of the Lingvo framework, currently under development by Google. PipeDream is part of Microsoft's Project Fiddle [7], however, it is currently limited in usage, so it was decided to focus on GPipe. The Lingvo framework [3] is built on top of TensorFlow, mainly developed for applications in natural language processing, but it can be extended to other areas.

Setting up the frameworks and resolving dependencies was done prior to carrying out experiments.

Preliminary testing: Preliminary testing of GPipe retrieved basic information on important cost factors and the know-how of implementing custom models on the Lingvo framework. These tests were done using a python script, that utilized the barebones features of GPipe. The model was a dummy neural network consisting of 16 identical layers, testing the basic functionality of GPipe on a synthetic minibatch of dimensions (16,8,8,1). Variations on the number of splits, the partitioning algorithm used, etc. were tested to get a high-level understanding of all the operations involved when using GPipe. Python debugger (pdb) was used to understand the flow of execution, and the main functions used by GPipe.

The following images show a breakdown of the operations involved when using GPipe in the case of an auto-partitioned, 4-split variation of the model (uses the layer partitioning algorithm of GPipe. Manual partitioning is also an option). Please note that the absolute time values are not important as the purpose was to just identify the primary operations. A detailed breakdown of the operations requires the use of profiling tools such as cProf, nvprof, tfprof [12], etc.

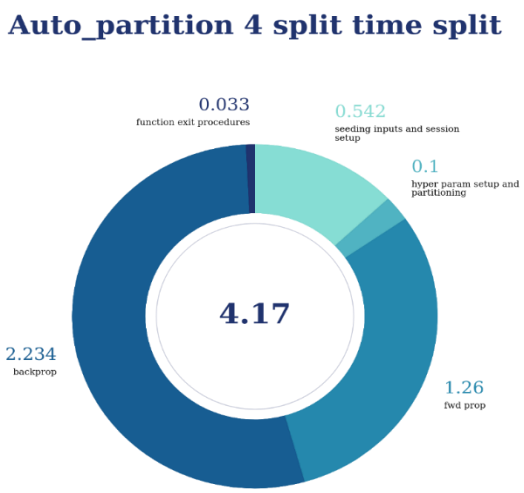


Figure 3.2: Main operations in GPipe based on preliminary testing

Statistical techniques and code optimization: Preliminary testing was done on a single minibatch of data. Relying on a single test run can lead to inaccurate results due to anomalies. To remove these anomalies from the reported values, it was necessary to average across multiple test runs, along with some statistical methods: only the time splits within a bound of +/- 1.5 times the standard deviation was considered. At least 80% of the values were within this bound, so it can be concluded that the results were fairly accurate. The values shown are applicable to a single minibatch that uses GPipe on the previously mentioned dummy neural network. Similar methods were applied for future tests.

Primary testing: Upgrading the Lingvo framework to make it compatible for customized models such as LeNet5 [13] and VGG16 [ ] involved modifications/additions to the Lingvo Library.

The Lingvo framework is primarily focused on language processing applications, hence GPipe has not been fully integrated into the framework for applications such as image processing. Upgrading the framework involved extensive debugging to understand how to create a customized model for different applications. Unmet dependencies required the following modifications to the library code, namely:

1. Creating GPipe compatible layers: Lingvo currently only supports GPipe for certain types of layers (primarily those used in NLP). Layers such as convolutional layers and pooling layers required this integration with GPipe. The following code snippet is an example of a GPipe-compatible SoftMax layer that was added to the framework:

```
class GPipeImageProcessingSoftmaxLayer(layers.SimpleFullSoftmax):
    """GPipe compatible softmax layer for image processing"""
    @classmethod
    def Params(cls):
        p = super().Params()
        return p

    def FProp(self,theta,inputs):
        p = self.params
        if not isinstance(inputs, list):
          inputs = [inputs]

        # If inputs are matrices already, delegate to _FProp2D.
        if inputs[0].shape.ndims == 2:
          return self.FProp2D(theta, inputs)
    def FProp2D(self,theta,inputs):
        p = self.params
        inputs = self._GetInputs(inputs)
        logits = self.Logits(theta, inputs)
        return logits
```

Figure 3.3: A GPipe-compatible SoftMax layer

2. Writing stub functions to enable the working of GPipe operations for the modified layers. GPipe layers depend on a function called FPropMeta () [4], that returns an estimate of the number of meta flop operations required by the layer, and information on the output of that layer. These values are used by the partitioning algorithm of GPipe at a later stage. Some commonly used layers such as convolutional layers, and pooling layers did not have an implementation of FPropMeta (), so a stub function was written to mimic the results. Below is an example of the FPropMeta () included for convolutional layers:

```
class Conv2DLayer(BaseConv2DLayer):
  """Convolution layer, with optional batch-normalization and activation."""

  def _EvaluateConvKernel(self, inputs, filter_w, strides, dilation_rate,
                          padding_algorithm, data_format):
    p = self.params
    return tf.nn.convolution(
        inputs,
        filter_w,
        strides=strides,
        dilations=p.dilation_rate,
        data_format='NHWC',
        padding=padding_algorithm)
  @classmethod #added fix to include meta flops for conv2d
  def FPropMeta(cls, p, inputs,*args):
    py_utils.CheckShapes((inputs,))
    temp = BaseConv2DLayer(p)
    shape = temp.OutShape(inputs)
    outputs = tshape.Shape(shape)
    return py_utils.NestedMap(flops=1, out_shapes=(outputs,))
```

Figure 3.4: A stub function for FPropMeta ()

At the time of testing, the documentation on GPipe was poor, so understanding how to create a customized model was difficult. There were no examples on how to extend GPipe for tasks such as image processing, so a lot of debugging was required to make changes to the library code for a customized model. Ultimately, a GPipe-compatible version of the LeNet5 model was built for further tests.

There are two main considerations required to understand how a partitioned model is created—how to indicate that pipeline parallelism is used, and how to handle communication of tensors/data between partition boundaries. The first is done by creating a wrapper around a standard neural network model called a *pipelining layer*. The pipelining layer, in simple terms, informs the underlying framework that the neural network is GPipe-compatible. The second is achieved through a special layer called the FeatureExtractionLayer [4], that handles device-to-device transfer of tensors/data between partitions. As an example, consider the neural network shown below:

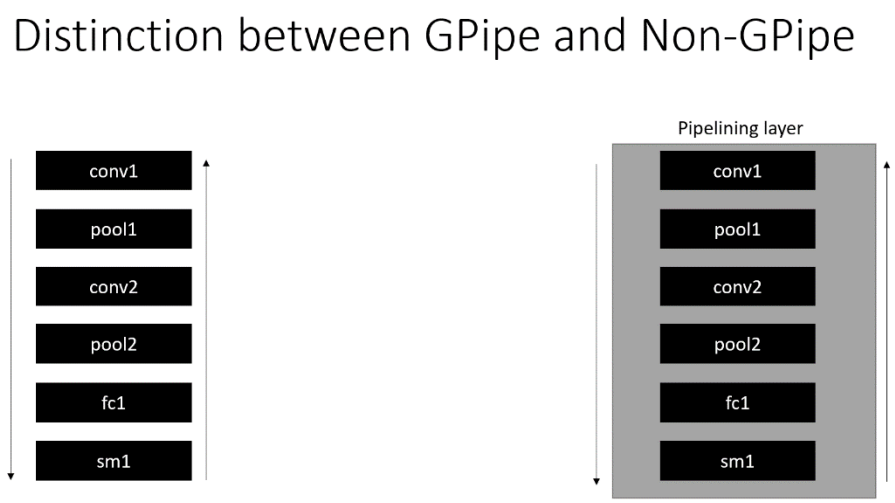
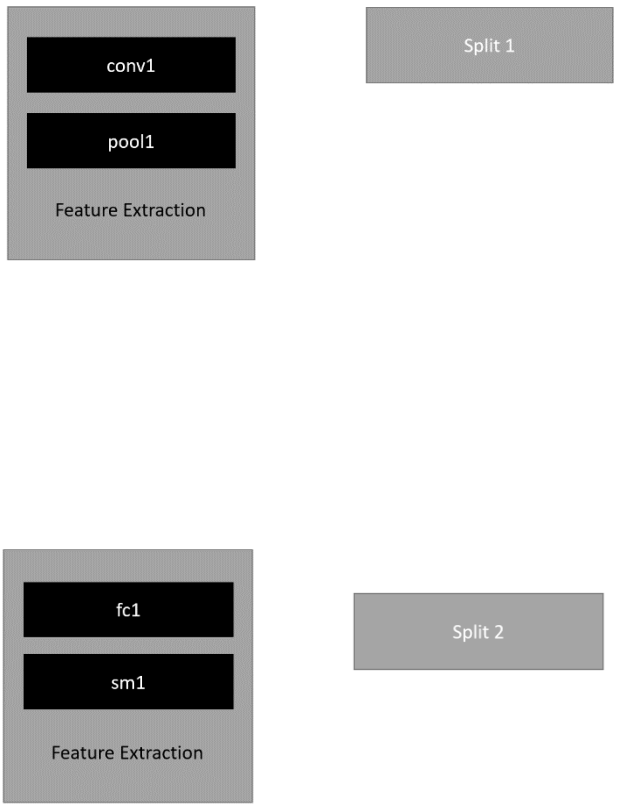


Figure 3.5: The functioning of the Pipelining layer as a wrapper to enable GPipe

The following figure illustrates how the layers are partitioned with the FeatureExtractionLayer handling communication:



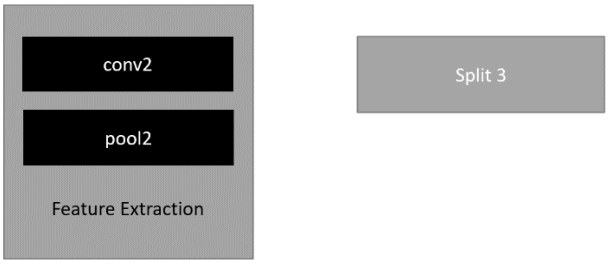


Figure 3.6: Illustration of partitioning in GPipe

GPipe explores pipeline parallelism in neural networks, so it is not meaningful to only test on a single GPU system. An AWS G3 instance with 2 GPUS (M60 Tesla) was used to carry out these tests. Library dependencies had to be resolved because of the modifications/enhancements made to the framework.

[8,9,10,11] shed light on conducting performance analysis for neural networks. Once dependencies were resolved, the final testing of the LeNet model and the VGG16 model were done on a 4-GPU Nvidia Tesla T4 cluster. TensorFlow Profiler was used to measure performance with variations in the number of GPU's, partition layers, micro-batches.

TensorFlow Profiler [12] provides detailed information on the most expensive CUDA Kernel calls, TensorFlow operations, and a trace file containing a timeline of operations, making it a convenient tool for performance measurement. The following figures are examples of the TensorFlow profiler outputs.

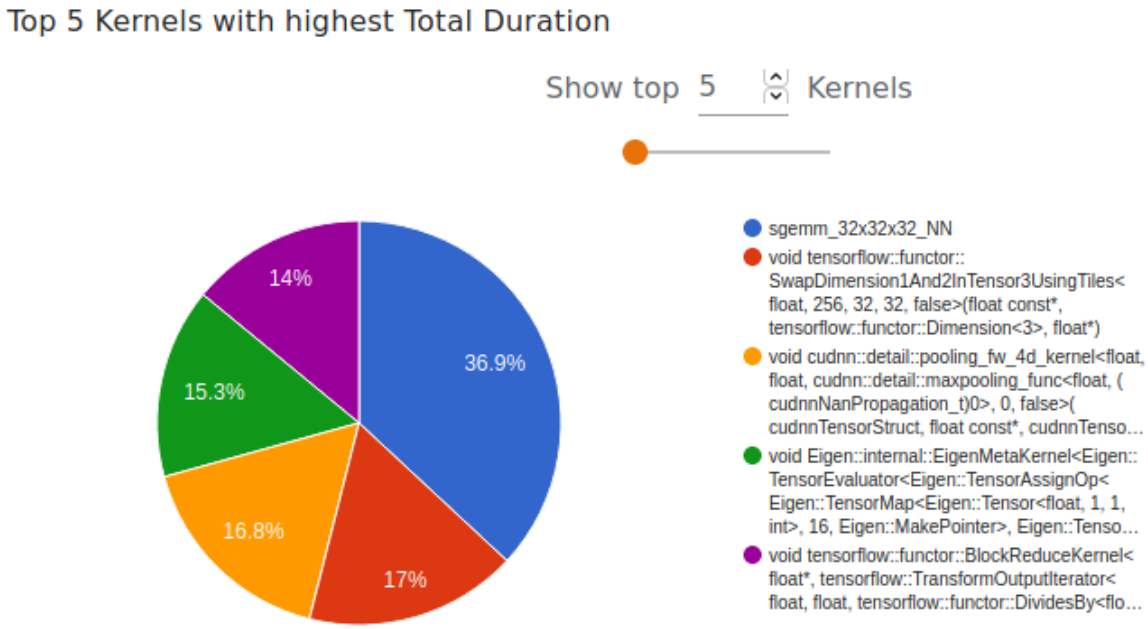


Figure 3.7: Example output of expensive CUDA kernels as measured by TensorFlow Profiler

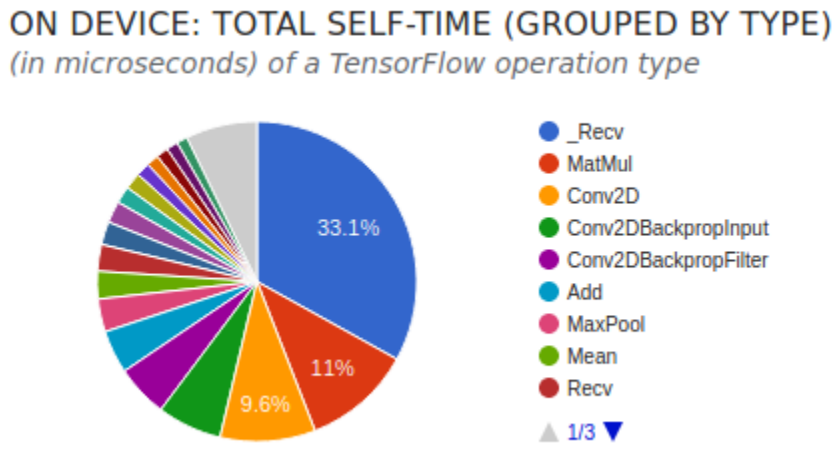


Figure 3.8: An indication of the expensive TensorFlow operations

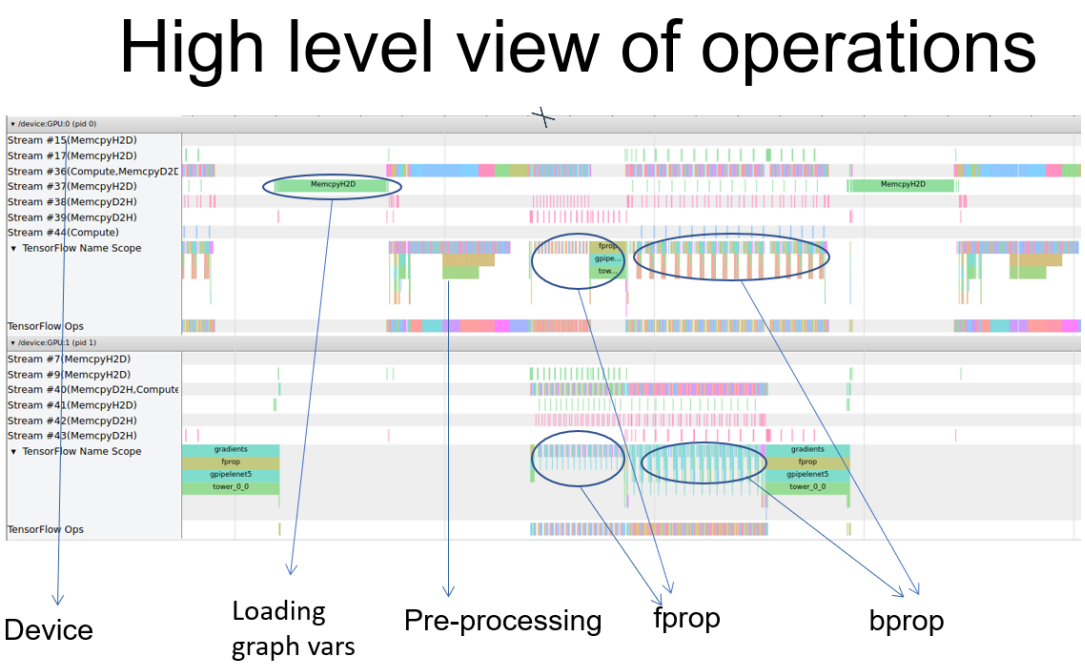


Figure 3.9: Trace file output of profiling which can be used to analyze scheduling info

Data parallelism was tested with GPipe. A detailed understanding of the scheduling of operations was obtained after analysing the trace files. Comparative studies on the variations helped identify potential bottlenecks. Results were documented and have been pushed to GitHub.

Analysis of results and exploring optimizations: After the conduction of the tests on the LeNet and VGG16 models, results were analyzed to understand the performance bottlenecks. These are discussed in the following section.

3.2 HARDWARE AND SOFTWARE REQUIREMENTS

Amazon Web Services was used to carry out the experiments. The below requirements expand on the tools installed/present in the AWS instance (Amazon Deep Learning AMI with EC2)

3.2.1 Software Tools

- Google's Lingvo framework [3] for testing GPipe.

- CUDA drivers and toolkits with corresponding versions >= 10.1
- Python 3.6+
- Python debugger—To understand how execution proceeds in the framework
- TensorFlow version 2.3+
- TensorFlow Profiler [12]
- TensorBoard
- SSH service such as PuTTY
- An operating system running the Linux kernel

3.2.2 Hardware Requirements

- A CUDA-capable GPU with support for CUDA versions >= 10.1. There is no fixed size for the memory size (8 GB recommended) of the GPU as it depends on the particular neural network under test. The experiments were done on the Nvidia M60 Tesla, and T4 Tesla GPU's.

3.3 PROJECT TASKS SUMMARY

Dates	**Summary**
1-1-21 to 11-1-21	Reading of base papers—PipeDream and GPipe
12-1-21 to 18-1-21	Reading of additional papers (XPipe and SpecTrain)
19-1-21 to 25-1-21	Framework setup and high-level understanding of execution (Lingvo framework)
26-1-21 to 2-2-21	Preliminary testing of GPipe

3-2-21 to 9-2-21	Statistical methods to improve preliminary testing
9-2-21 to 22-2-21	Upgrading Lingvo and extending GPipe compatibility for non-NLP tasks. Implemented LeNet5 with GPipe compatibility
22-2-21 to 3-3-21	Implementation of model on multiple GPU cluster using AWS (2 M60 Tesla GPUS, 8GB each)
4-3-21 to 11-3-21	Additional reading of papers on performance analysis
11-3-21 to 15-3-21	Performance analysis using TensorFlow profiler
15-3-21 to 31-3-21	Extended Lingvo to make data-parallel training possible for GPipe. Results documented and pushed to GitHub. Trace files analysed and bottlenecks identified. Detailed scheduling information obtained
1-4-21 to 7-4-21	Tested variations on the number of micro-batches and splits to identify combination with maximum performance
8-4-21 to 12-4-21	Provided project details to AWS for acquiring access to 4-gpu system. Approval came back within 4 days (12th April)
12-4-21 to 15-4-21	Setting up instance and framework for 4-gpu system (T4 Tesla, 16 GB each). Added patches for data-parallelism with GPipe (extension of 2-gpu modifications)
15-4-21 to 22-4-21	Conducted performance tests on 4-GPU system
22-4-21 to 27-4-21	Analysis of profiler output and documenting of results
27-4-21 to 30-4-21	Basic preparation of VGG16 model with GPipe

1-5-21 to 4-5-21	Layer-wise costs in addition to partition costs for LeNet5
4-5-21 to 8-5-21	Fixed bugs in the VGG-16 model
8-5-21 to 15-5-21	Performance analysis of VGG-16
15-5-21 to 31-5-21	Analysis of profiling data for VGG16 model

CHAPTER 4
RESULT ANALYSIS

This chapter provides a comparative study of performance study done on LeNet5 and the VGG16 model. Data-parallel tests were also done. Tests on the LeNet model were done using the MNIST dataset using minibatches of size 1024. For VGG16, synthetic inputs of same dimensions as the ImageNet-12 dataset were used using minibatches of size 128. The use of synthetic inputs is justified because the aim was to study the performance of the model at the systems-level rather than the application level. This is independent of the dataset used.

GPipe partitions the layers based on the forward prop computation time of each layer. The splits are partitioned so as to minimize the variation in the computation time of each split. A similar approach was followed when partitioning the models in the experiments.

4.1 ANALYSIS OF END-TO-END TIMES

4.1.1 End-to-end times in milliseconds (Forward prop + Backward prop)

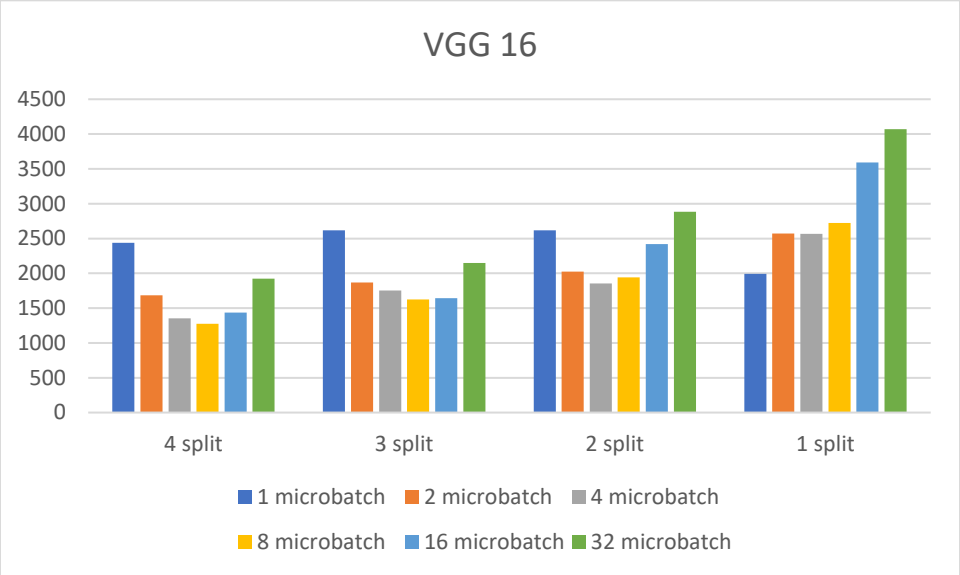


Figure 4.1: End-to-end total times in milliseconds (forward + backprop) for VGG16

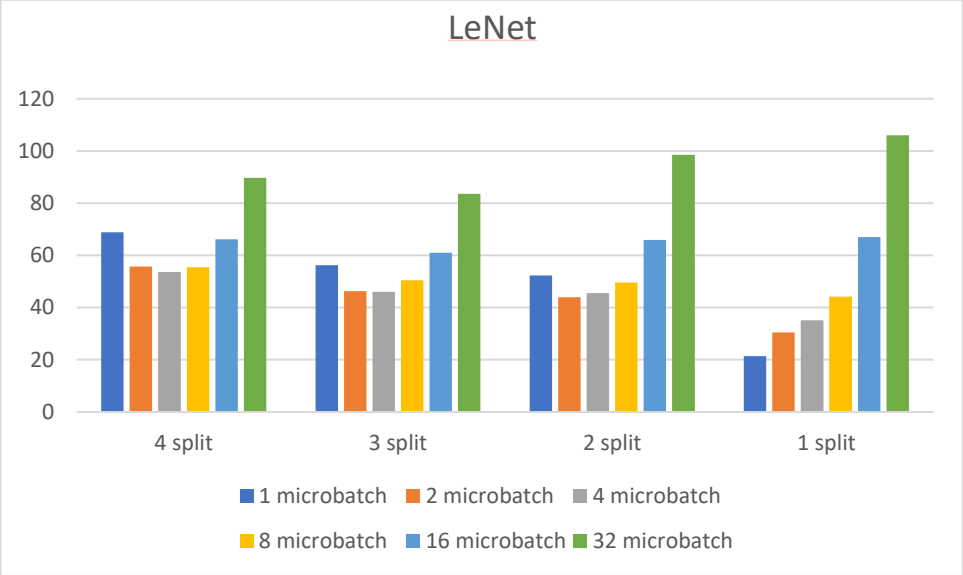


Figure 4.2: End-to-end times (Forward + Backprop) for LeNet5

Both results show a similar trend as shown in the graphs above. When some pipeline parallelism is present (when using more than 1 split), we notice that performance improves with increasing the number of micro-batches up to a point, beyond which it worsens. This is contrary to what one expects, as an increase in the number of micro-batches should improve the extent of pipeline parallelism in the system. This is a recurring observation in the experimental results and an explanation is provided at the end of this section.

In the absence of pipeline parallelism (single split), increasing the number of micro-batches worsens the performance.

4.1.2 Forward prop end-to-end times in milliseconds

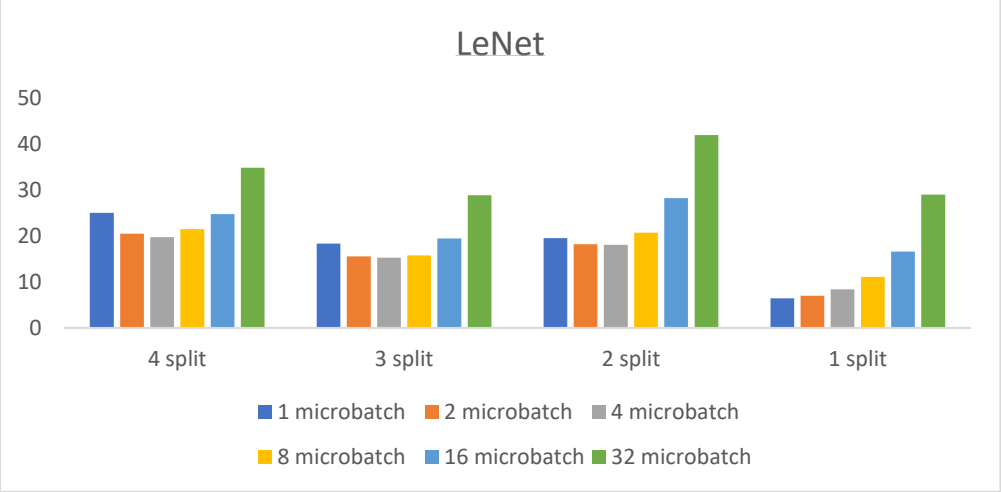


Figure 4.3: End-to-end times forward prop in LeNet

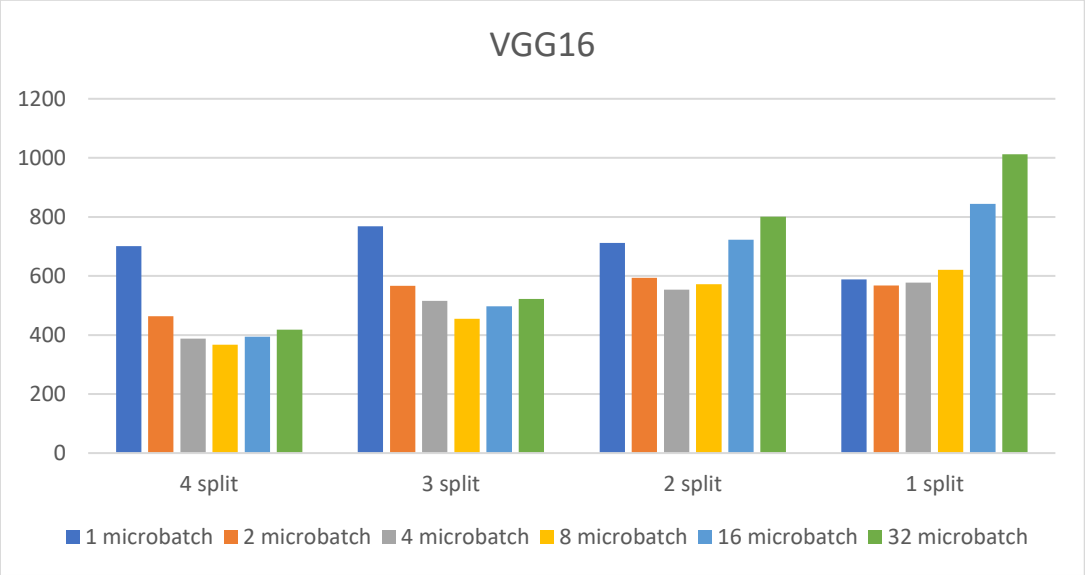


Figure 4.4: End-to-end forward prop times VGG 16

4.1.3 Backprop end-to-end times in milliseconds

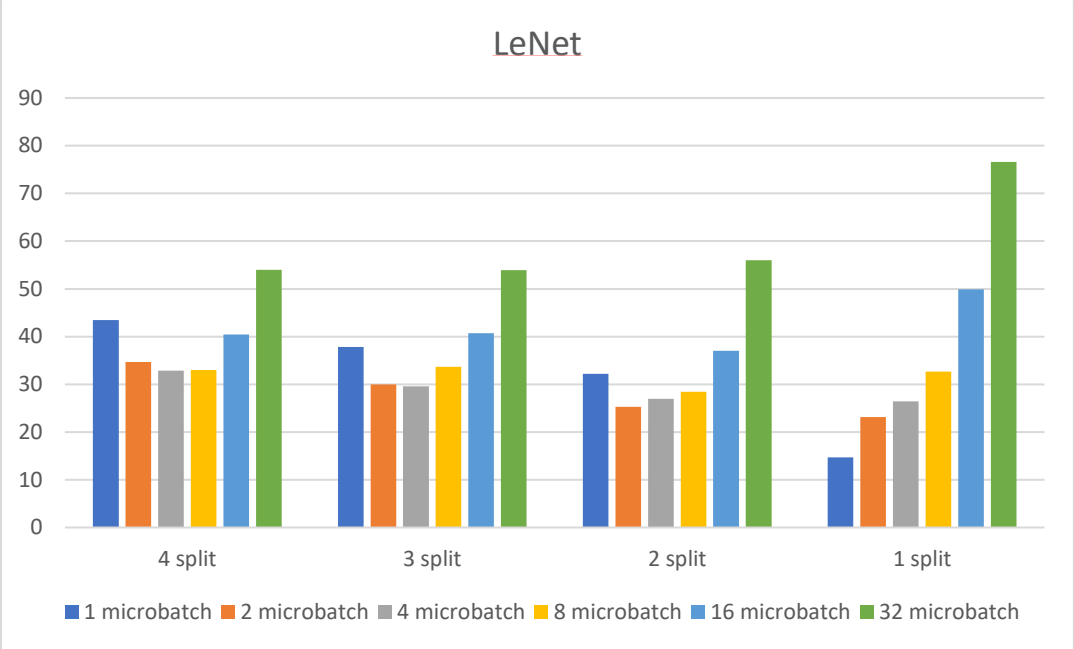


Figure 4.5: End-to-end backward propagation time LeNet

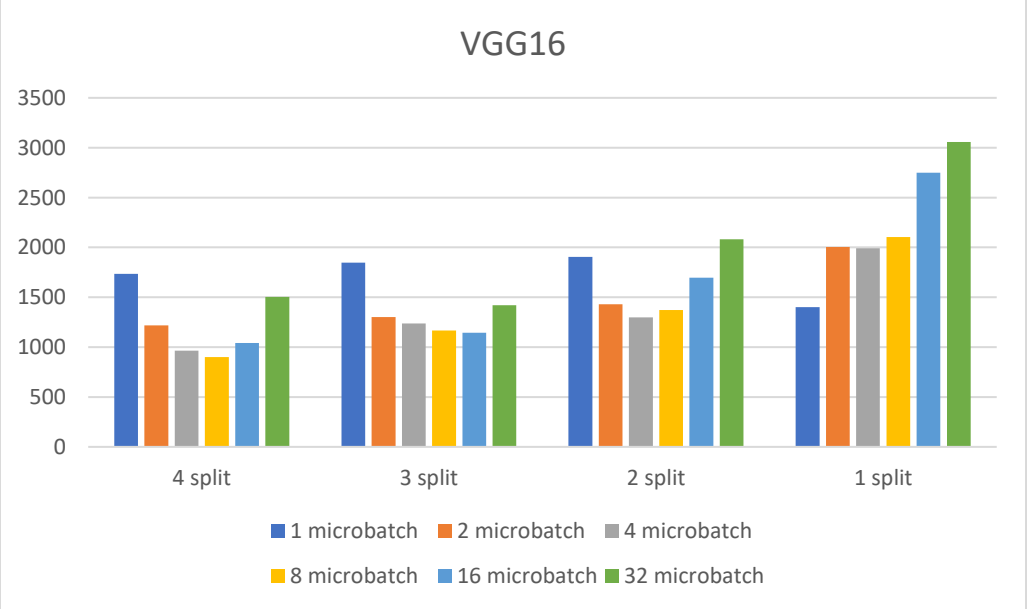


Figure 4.6: End-to-end backprop times VGG16

In all three scenarios, we observe that increasing the number of micro-batches does not improve performance after a point, in fact performance deteriorates. Why does this happen? Consider a

minibatch of size 1024. Assume the total compute time for processing this minibatch is *C*. When dividing this minibatch into 2 equal micro-batches, we expect the compute time of each micro-batch to be *C/2*, under an ideal scenario. However, from Amdahl's law, we know this not to be true, hence the total computation time may be slightly higher than *C/2* for each micro-batch. Therefore, the accumulated compute time for processing both micro-batches end up being greater than *C*. So, we conclude that halving the minibatch/micro-batch further does not double the speedup of processing each micro-batch. This is illustrated in the figure shown below.

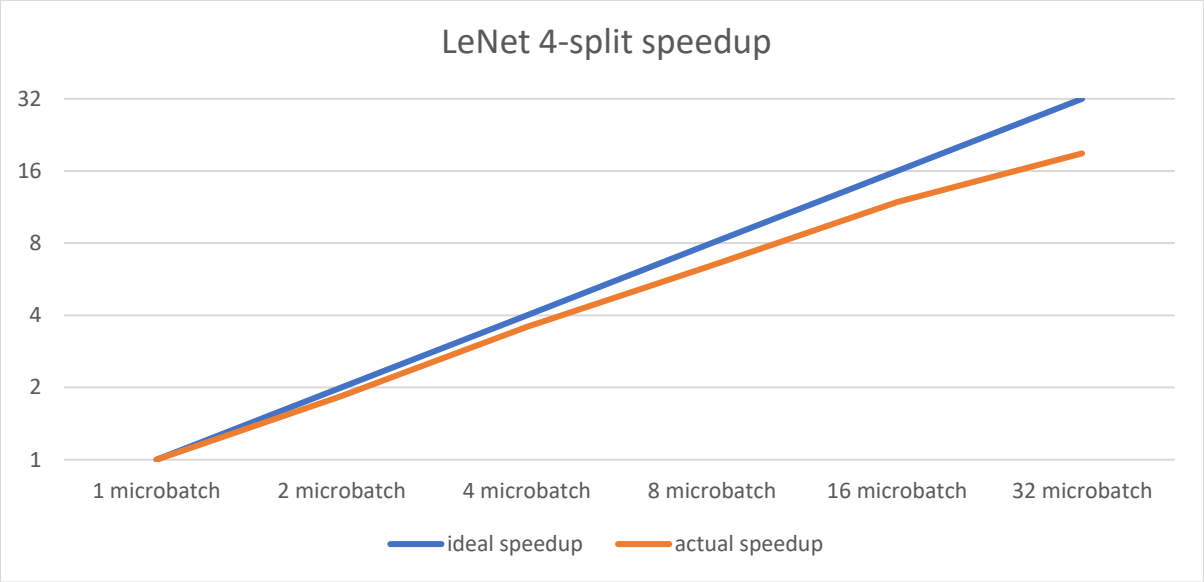


Figure 4.7: Ideal vs observed speedup in LeNet for 4 split partitions

From the above figure we notice that the deviation from the ideal speedup increases with increasing the number of micro-batches, leading to higher total processing times (See table below)

Table 4.1: Total processing time for different number of micro-batches in single split VGG16 model

	2 microbatch	4 microbatch	8 microbatch	16 microbatch	32 microbatch
Processing time (ms)	565.858	573.494	613.48	829.6	867.52

The table above explains why performance deteriorates for large number of micro-batches—it is because of an increase in the workload. For a reasonable increase in the micro-batches, the total workload/processing time does not increase by too much, hence the effect of pipelining still leads to better performance.

4.2 COMMUNICATION COSTS

The figure below shows how communication times vary for different micro-batches.

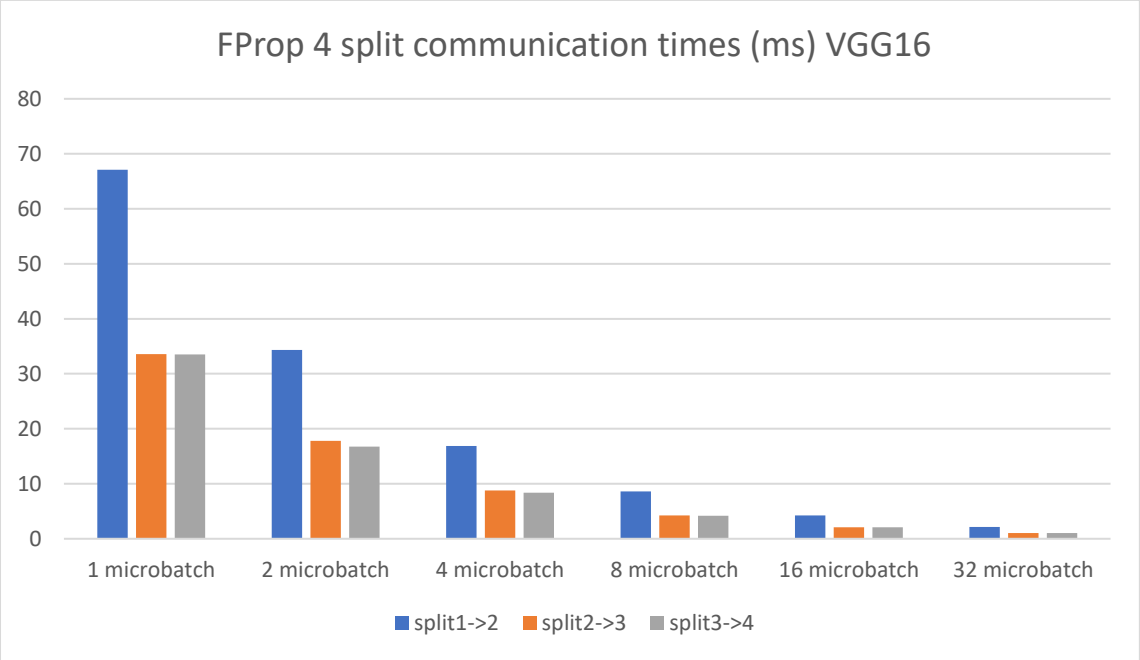


Figure 4.8: Communication times in forward prop for VGG16

Now compare this with the tensor sizes transferred between the boundaries:

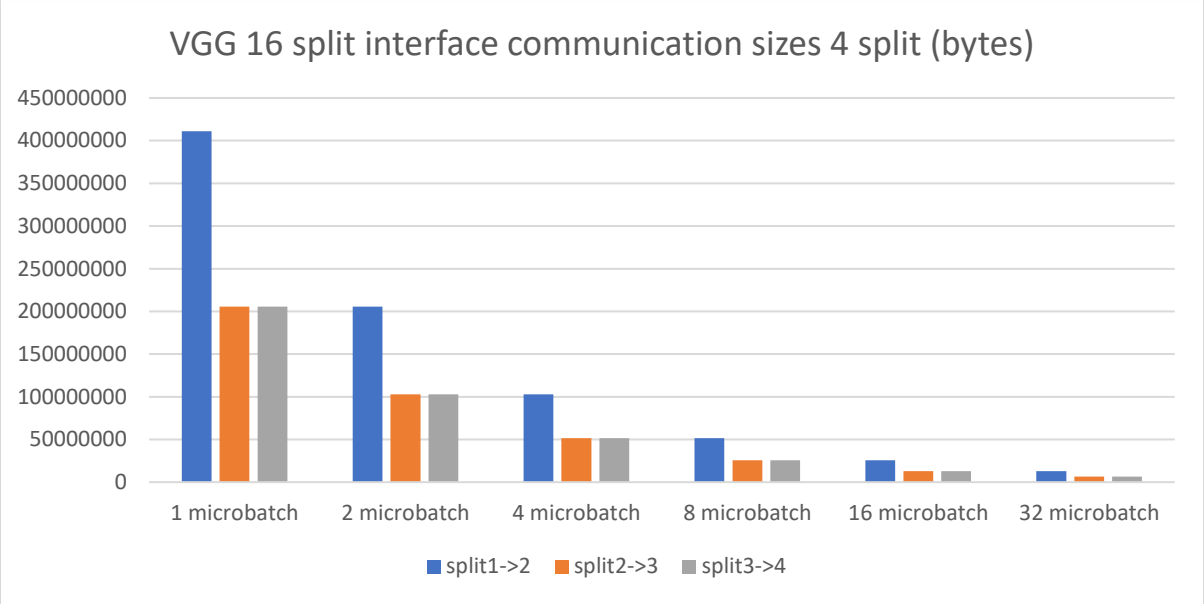


Figure 4.9: Tensor sizes in forward prop communication VGG16

We observe that unlike computation costs, the communication costs vary linearly with the size of the micro-batch, i.e., halving the microbatch size approximately halves the time spent communicating.

4.3 AREAS FOR IMPROVEMENT

Consider an illustration of pipelining in a model with 3 splits and 3 micro-batches:

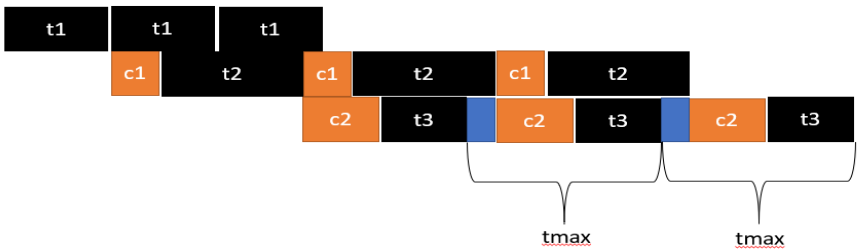


Figure 4.10: Pipelining in GPipe

To estimate the end-to-end processing time for m micro-batches, we use the following formula:

$$time = t1 + c1 + t2 + c2 + t3 + tmax * (m - 1)$$

Where *tmax* is the maximum processing time of a stage in the pipeline.

We observe that the communication cost of sending a tensor from the first to the second split is counted as a part of the 2nd partition. This is because TensorFlow, works on passive sends and active receives—i.e., the sender does not "push" the tensors into the next GPU/receiver, rather, the receiver must "pull" these tensors from the sender when required.

Thus, we notice that the value of tmax depends on the communication costs and the computation costs for each partition. For the most efficient partitioning, all stages must have equal costs, which will lead to the best end-to-end time. Since GPipe currently only partitions based on the computation costs, the resulting partitioning may not be the best.

To understand the extent to which communication influences partitioning, consider the *partition efficiency* defined as follows:

$$Partition\ Efficiency = \frac{\sum_{i=0}^{i=k} S_i}{k \times S_{max}}$$

Where k is the number of stages in the pipeline, S_i, are the costs of the individual stages, and S_{max} is the cost of the most expensive stage in the pipeline.

The table below shows the partition efficiency (%) calculated without taking communication

Table 4.2: Partition efficiency in VGG16 4 split without considering communication costs

Fprop compute-costs per microbatch for 4 split scenario (standard)						
scenario	1 microbatch	2 microbatch	4 microbatch	8 microbatch	16 microbatch	32 microbatch
split1	180.168	97.133	43.58	21.956	10.775	5.426
split2	131.326	65.217	31.875	16.1	9.905	5.049
split3	145.231	70.088	34.1	17.923	13.985	6.141
split4	113.475	50.502	26.592	15.65	13.236	8.59
avg cost	**142.55**	**70.735**	**34.03675**	**17.90725**	**11.97525**	**6.3015**
efficiency	79.12	72.82	78.1	81.56	85.63	73.36

The average partition efficiency comes out to be 78.43%

In contrast, compare this with the scenario taking communication into account:

Table 4.3: Partition efficiency in VGG16 4 split when considering communication costs

Fprop compute-costs per microbatch for 4 split scenario (standard)						
scenario	1 microbatch	2 microbatch	4 microbatch	8 microbatch	16 microbatch	32 microbatch
split1	180.168	97.133	43.58	21.956	10.775	5.426
split2	198.404	99.567	48.741	24.732	14.152	7.183
split3	178.811	87.888	42.892	22.148	16.075	7.18
split4	147.011	67.232	34.964	19.836	15.353	9.636
avg	**176.0985**	**87.955**	**42.54425**	**22.168**	**14.08875**	**7.35625**
efficiency	88.75	90.55	87.07	89.63	87.64	76.34

Now the average partition efficiency comes out to be: 86.66%

Hence, we cannot ignore communication costs when partitioning.

CHAPTER 5
CONCLUSION AND FUTURE SCOPE

5.1 CONCLUSIONS

To summarise, this report covers the implementation of the GPipe library—its functioning, and how models can be made using the Lingvo framework. It provides a detailed performance analysis of the library, highlighting the bottlenecks with suggestions for areas to improve upon.

The following conclusions have been drawn:

1. Contrary to expectations, the performance of GPipe does not increase with increase in the number of micro-batches. Rather, the performance improves up to a point, after which it starts deteriorating.
2. Communication costs, unlike computation, vary linearly with the microbatch size.
3. The algorithm for partitioning can be improved by taking communication costs into account.

5.2 FUTURE SCOPE

In this report, the problem of increasing micro-batches to large numbers has been identified. Analysis on identifying the optimal number of micro-batches for a particular model and dataset, is critical to performance enhancement.

GPipe partitions based only on forward propagation. Partitioning based on backward propagation must also be considered because backprop costs are in general higher than the forward prop costs. In addition, communication costs must be taken into account when developing a mechanism for partitioning.

REFERENCES

Journal / Conference Papers

[1] Aaron Harlap, Deepak Narayanan, “PipeDream: Fast and Efficient Pipeline Parallel DNN Training”, arXiv preprint, June, 2018

[2] Yanping Huang, Youlong Cheng, “GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism”, Advances in Neural Information Processing Systems, 32 103--112, 2019

[3] Jonathan Shen, P Nguyen, “Lingvo: A Modular and Scalable Framework for Sequence-to-Sequence Modelling”, February, 2019

[4] Chi-Chung Chen, Chia Lin Yang, “Efficient and Robust Parallel DNN Training Through Model Parallelism On Multi-GPU Platform”, October, 2019

[5] Lei Guan, Wotao Yin, “XPipe: Efficient Pipeline Model Parallelism for Multi-GPU DNN Training”, November, 2020

[6] Arpan Jain, Ammar Ahmad Awan, “Performance Characterization of DNN Training using TensorFlow and PyTorch on Modern Clusters”, IEEE Xplore, September, 2019

[7] Jie Liu, Jiawen Liu, “Performance Analysis and Characterization of Training Deep Learning Models on Mobile Devices”, September, 2019

[8] Shi Dong, Xiang Gong, “Characterizing the Microarchitectural Implications of a Convolutional Neural Network (CNN) Execution on GPUs”, ICPE’18, Berlin, Germany, April 9-13, 2018

[9] Robert Adolf, Saketh Rama, “Fathom: Reference Workloads for Modern Deep Learning Methods”, October, 2016

Reference / Hand Books

[10] Sue Grace, Phil Gravestock, Inclusion and Diversity: Meeting the Needs of All Students, Taylor & Francis, 2008

[11] Engineering ethics in practice: a guide for engineers, The Royal Academy of Engineering, August 2011

[12] Ibo van de Poel, Lamber Royakkers, Ethics, Engineering and Technology, Wiley-Blackwell Publishers, May 2011

Web

[13] Lingvo repository, https://github.com/tensorflow/lingvo

[14] Microsoft Project Fiddle repository, https://www.microsoft.com/en-us/research/project/fiddle/

[15] TensorFlow Profiler, https://www.tensorflow.org/guide/profiler

[16] LeNet model, https://en.wikipedia.org/wiki/LeNet

[17] VGG 16 model, https://neurohive.io/en/popular-networks/vgg16/

[18] Amdahl's Law, https://en.wikipedia.org/wiki/Amdahl%27s_law

PLAGIARISM REPORT

PROJECT DETAILS

Student Details			
Student Name	**ADITYA SHANKAR**		
Register Number	170905094	Section / Roll No	CSE B, Roll.17
Email Address	aditya.ssr@gmail.com	Phone No (M)	+91 9901651358
Project Details			
Project Title	Analysing the performance and scheduling of neural networks that use pipeline parallelism		
Project Duration	5 months	Date of reporting	January 1st, 2021
Organization Details			
Organization Name	**SERC Lab, IISc Bangalore**		
Full postal address with pin code	Electrical Engineering Department, Indian Institute of Science, Bangalore, Javanica Marg, Mathikere, Bengaluru, Karnataka 560012		
Website address	http://www.serc.iisc.ac.in/		
External Guide Details			
Name of the Guide	**Dr. Govindarajan Ramaswamy**		
Designation	Professor, Dept. of Computer Science and Automation		
Full contact address with pin code	Dept. of Computer Science and Automation/ SERC Lab, Indian Institute of Science, Mathikere, Bengaluru, Karnataka 560012		
Email address	govind@iisc.ac.in	Phone No (M)	+91 (80) 2360 0654 or 2293 2794
Internal Guide Details			
Faculty Name	**Dr. K.N. Manjunath**		
Full contact address with pin code	Dept of Computer Science & Engg, Manipal Institute of Technology, Manipal – 576 104 (Karnataka State), INDIA		
Email address	manjunath.kn@manipal.edu		