DistCNN: Mid Semester Evaluation

ABHISHEK GUPTA and SIDDHARTH YADAV

DistNN, Distributed Neural Networks, is a framework for modeling and training Neural Network over a cluster of GPU.

CCS Concepts: • **Deep Learning → Machine Learning**; • **Distributed Systems**; • **Parallel Computing**;

Additional Key Words and Phrases: neural networks, distributed systems, model parallelism

ACM Reference Format:
Abhishek Gupta and Siddharth Yadav. 2019. DistCNN: Mid Semester Evaluation. 1, 1 (April 2019), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep Learning is blooming these days. With time, the size of these networks and the data is increasing and it is becoming harder to train a network on a single GPU machine. The number of research papers published using distributed GPU in the area of Deep Learning is increasing with time. This can be clearly seen in Introduction. The numbers of nodes in pre-2013 era is because GPU weren't popular in Machine Learning community at that time. Moreover in production, where high throughput and availability is our main goal, distributed Deep Learning system play a great role.

Having established the importance of distributed deep learning, we plan to implement a naive distributed neural network framework by using CUDA, cuBLAS, Thrust and CUDA-aware MPI. This project is challenging because:

- Making a general purpose API for modeling and training neural networks in C++
- We have only used low-level API like CUDA and cuBLAS instead libraries like cuDNN used by

2 LITERATURE

The idea of using GPUs to train networks faster is not a new one. There is a lot of literature discussing how the learning process can be made faster in general. If we do not focus on neural networks specifically, then papers like cuSVM[5] work on writing highly parallel algorithms for SVM learning.

We looked at parallel implementation of neural networks in CUDA[3] to look at how the speedups can be achieved in a simple neural network. There is detailed discussion about the backpropagation algorithm which is to be used. The algorithms consists of various matrix operations which can be parallelized. They identify two types of parallel operations, vector and matrix products and arithmetic operations. When to use cuBLAS is a topic that they have covered as well. Experiments on cancer and mushroom data show how fast they converge.

Since our main focus was on deciding the algorithm for our upcoming goals we looked at some articles on data parallelism[1] and moel parallelism[2] to help us decide which parallelization strategy would be the best for us.

Authors' address: Abhishek Gupta, abhishek16004@iiitd.ac.in; Siddharth Yadav, siddharth16268@iiitd.ac.in.

XXXX-XXXX/2019/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

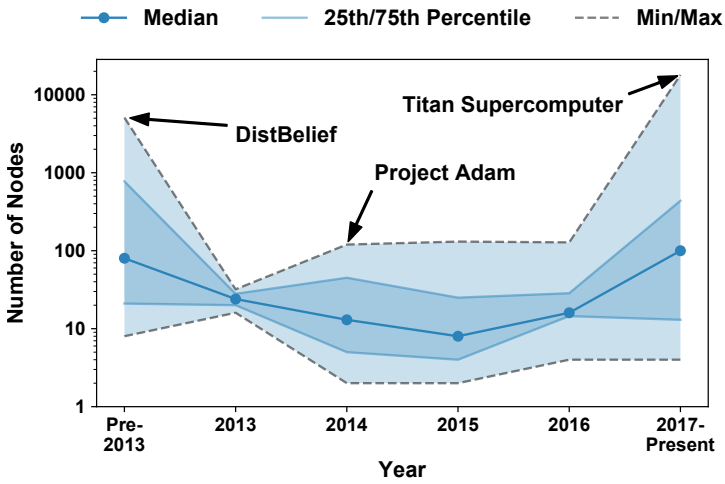

Fig. 1. Training with Single vs. Multiple Node

We also looked at some in-depth analysis for parallel deep learning[4]. This is by far the most holistic analysis we could find on the topic of parallelizing neural networks. It starts off with descriptions of basics required for understanding any kind of learning. In section 2.3 they focus on things like single machine parallelism vs multiple machine parallelism. Further discussions about reductions and SGD follow in section 2.
Section 3 focuses on the various parts of a deep learning architecture. It talks about neurons in terms of how they are used during a forward pass etc. They even touch upon some harder parts of deep learning such as recurrent layers and focus on some case studies which includes very popular architectures namely AlexNet, GoogleLeNet, ResNet etc.
Section 4 is a concurrency analysis of popular layer types that are used. These layers are

(1) Fully Connected Layers
(2) Convolution Layers
(3) Recurrent Layers

For fully connected layers since the only operation happening is essentially matrix matrix multiplication, we can use optimizations for the Linear Algebra libraries to improve performance of our deep learning models.
For convolution operations there is focus on increasing data redundancy to utilize as many resources as possible. There is discussion about various mathematical techniques used to modify the algorithm, well beyond the scope of people reading and writing this report.

We will not be discussing recurrent units as they are not an area of focus for us in this project.

3 SHARED MEMORY PARALLELISM

A serial neural network can be heavily parallelized on a multi-core platform. This is achieved in the following way

3.1 Mathematical Operations Parallelism

Majority computational operations performed in Neural Networks are Matrix Operations, which cab be performed efficiently in parallel on GPU. We have utilized several parallelized kernels in this projects. Using them alone in place of serial matrix operations results in a significant speedup.

- Matrix Multiplication

Table 1. Work-Depth Analysis of Convolution Implementations

Method	Work (W)	Depth (D)
Direct	$N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$	$\lceil \log_2 C_{in} \rceil + \lceil \log_2 K_y \rceil + \lceil \log_2 K_x \rceil$
im2col	$N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$	$\lceil \log_2 C_{in} \rceil + \lceil \log_2 K_y \rceil + \lceil \log_2 K_x \rceil$
FFT	$c \cdot HW \log_2(HW) \cdot (C_{out} \cdot C_{in} + N \cdot C_{in} + N \cdot C_{out}) + HWN \cdot C_{in} \cdot C_{out}$	$2\lceil \log_2 HW \rceil + \lceil \log_2 C_{in} \rceil$
Winograd ($m \times m$ tiles, $r \times r$ kernels)	$2^*\alpha(r^2 + \alpha r + 2\alpha^2 + \alpha m + m^2) + C_{out} \cdot C_{in} \cdot P$ $(\alpha \equiv m - r + 1, \quad P \equiv N \cdot \lceil H/m \rceil \cdot \lceil W/m \rceil)$	$2^*2\lceil \log_2 r \rceil + 4\lceil \log_2 \alpha \rceil + \lceil \log_2 C_{in} \rceil$

- Matrix Scaling
- Matrix Addition/Subtraction
- Convolution

3.2 Neuron Parallelism

This result from the fact that each neuron of a particular layer can be processed independently. Due to this fact, we are able to victories most of neural network operation. It is also observed in CNNs, where each filter in a convolution layer can be applied in parallel. There are usually 32-128 filters in each convolutions layers and processing them independently results in a big speed-up.

4 DISTRIBUTED MEMORY PARALLELISM

Here we discuss parallelism possibilities in distributed GPU setting. We can scale the network by either using Data Parallelism or Model parallelism.

4.1 Data Parallelism

Consider working with a large dataset. While we would like to go through the whole dataset quickly, it has been shown that large batch sizes lead to poor performance. Thus, we would like to distribute the data over different GPUs and share what the network has learned afterwards. This can be done by replicating the network over different GPUs with small portions of data training these different network independently. At the end we would combine these learnt parameters.
The problem with this parallelism is the immense communication overhead that it incurs. The parameters have to be copied several times from different GPUs to the central GPU where they are averaged out.

4.2 Model Parallelism

Model parallelism is the idea that we divide the model and all the GPUs train different parts of the model over the same data. This provides us with the obvious benefit of having very large networks. There is data sharing that happens here as well when we are trying to do matrix multiplication for forward pass or backward pass. However the amount of data to be transferred is usually lower here.
Model parallelism suffers from the problem of synchronization. The output of layers are dependent on previous layers so in a divided model we need to wait for all GPUs to finish working on a layer to move on.

4.3 Combination

There is no one-method-fits-all solution to this problem. We often need to make decisions based on the problem at hand. One way to tackle the problem is to use a combination of these. If the network is small then shift to Data Parallelism since not much sharing needs to happen now. Otherwise work with Model Parallelism.

5 OPTIMIZATION ALGORITHM PARALLELISM

In this section, we discuss techniques that requires modification of the training algorithms in Deep Learning to increase or assist parallelism while maintaining the accuracy.

5.1 Pipeline Parallelism

This is a layer oriented parallelism done in a pipelining fashion. It is similar to instruction level pipelining in CPUs. In general, each layer is just dependent on its previous layer. Therefore, the whole model can be run in a pipeline. In other words, the first layer will start working on the second input right after doing computation on first input and it will not wait for the first input to be passed through the entire network. The implementation of this parallelism non-trivial because weights in each layer needs to be updated after evaluation of each batch/input. This problem is solved by making changes to the batching algorithm. This detail will be discussed in the final report.

5.2 Batching

In batching we apply gradient update after evaluating the model on a batch, i.e., a group of training example. Therefore, in a single forward pass of the model, we process to more than one input data point. This technique can also be applied on CPU, but results the speed up in case of GPU implementation are more significant as GPUs are better in multiplication of larger matrix.

5.3 Parallelization Strategy

Since we are working with a small network we would be focusing on Data Parallelization as the most basic technique. We would also be looking at pipeline parallelism while optimizing the code.

6 RESULTS

We were supposed to produce comparison charts with python libraries but we couldn't do that in light of some unfixed bugs in our code. We will be able to do this part in the near future.
We have however gone through the theoretical parts and decided on how to move forward with parallelization.

6.1 Current Status

We have done the following work:

- Naive implementation of Neural Network on a GPU

We are a little behind as we couldn't fix some bugs in the code. This would be done as soon as possible.

6.2 Future Work

We would be implementing the following things in the final submission of the project:

- Fixing bugs in the code
- A network distributed over multiple GPUs
- Handle node failure over the network
- Optimize the code

REFERENCES

[1] How to Parallelize Deep Learning on GPUs Part 1/2: Data Parallelism, http://timdettmers.com/2014/10/09/deep-learning-data-parallelism/
[2] How to Parallelize Deep Learning on GPUs Part 2/2: Model Parallelism, http://timdettmers.com/2014/11/09/model-parallelism-deep-learning/

[3] Parallel Training of a Back-Propagation Neural Network Using CUDA,
https://intranet.matematicas.uady.mx/personal/mramirez/web/madera/PID1501721.pdf
[4] Demystifying Parallel and Distributed Deep Learning: An In-DepthConcurrency Analysis,
http://htor.inf.ethz.ch/publications/img/distdl-preprint.pdf
[5] CUSVM: A CUDA IMPLEMENTATION OF SUPPORT VECTOR CLASSIFICATION AND REGRESSION
http://www.patternsonascreen.net/cuSVMDesc.pdf

.