Parallel Neural Network Framework

Srajan Garg 140050017
Anuj Mittal 140050024
Sumith Kulal 140050081
Shubham Goel 140050086

***Abstract*—Most of the current computer systems use statistical techniques to 'learn" with data and progressively improve performance on a specific task. Artificial Neural Networks are one such technique that is extensively used. However computation done in neural networks is intensive and can be vastly benefit from parallel techniques. In this paper, we present a parallel neural network framework. We also attempt to support convolution layer for the neural networks. Most image manipulation tasks use convolution routines. Furthermore, recent work on image manipulation is data-driven and uses such frameworks. As an example study, we work on a higher level (learning-based) tasks of Optical Character Recognition on the MNIST dataset. The convolution layer in the framework allows us to tackle a wide range of simple (non-learning) image manipulation problems with a single fixed convolution layer (Eg. Blur, Edge Detection etc.). We have implemented fully connected layers and activation layers in the neural network. All the operations in these layers are inherently parallelizable and we compared the performance of the serial code with parallelized CUDA running on GPU.**

I. Introduction

The size of images is generally huge and a serialized code can be significantly slow when running image manipulation tasks. It becomes further slow when tasks involve machine learning techniques like neural networks which involve processing huge data to train the network. In this project, we show how these processes can run significantly faster with parallelization using GPU. We worked on two problems in this field. We compared GPU parallelized to show the speedup gained by GPU parallelization. We also observed that CPU parallelization using OpenMP did not give us a significant reason and try to reason the cause of it.

II. Neural Network

To begin with, we write a standalone C++ serial neural network library, and provide a user friendly API. We wanted to give the user the flexibility to choose their own network architecture with simple function calls, just like in popular python libraries like Tensorflow[1] and PyTorch[2]. We also provide tunable parameters like learning rate and number of training epochs for the network. Next, we parallelize the network using two popular concurrency techniques from different domains. We use OpenMP for CPU parallelization and CUDA for GPU parallelization. We then discuss the approaches used and results obtained.

A. Layers

The entire network, in essence, is a list of objects of type *Layer*. Each layer has some common operations, and we use pure virtual functions in a base class to specify the operations required. A willing user can even add their own *Layer* by inheriting from the base class and defining the required functions. Each layer has it's own buffer for storing outputs after the forward operation, and also a buffer for storing the gradients that need to be passed to the previous layer. Each layers also stores pointers to the buffers of the previous and next layer to use the data in the forward and backward pass respectively.

The library comes with the following pre-defined layers:

- **Input**: This layer is added to the network by default and acts as a proxy layer for interfacing user supplied data with the network. It has no extraordinary function in the network's learning.
- **Dense**: This layer takes in the number of hidden units as the parameter and adds a fully connected dense layer to the already existing neural network. it borrows the number of the inputs from the existing network, and initializes a weight matrix appropriately.
- **Activation**: This layer adds an activation function on the output of the previous layer. A variety of activation functions have been made available including ReLU and sigmoid.

B. Algorithm

The neural network performs its primary task of learning its tunable parameters when the *train* method is called, which envelopes the three main procedures require for this step in a loop. Apart from these methods, the neural network has some miscellaneous helper functions like *add_training _data* and *initialize*. The main methods in the network's learning procedure are :

- Forward Pass
- Error Calculation
- BackPropogation

Dense Layer Forward:

$$\underset{(outF)}{out} = \underset{(outFXinF)}{Wt} * \underset{(inF)}{in} + \underset{(outF)}{Bias} \tag{1}$$

Dense Layer Backprop:

$$\frac{\partial E}{\partial in} = \frac{\partial out}{\partial in} * \frac{\partial E}{\partial out} \Rightarrow \quad \delta_{in} = Wt^T * \delta_{out} \tag{2}$$

$$\frac{\partial E}{\partial Bias} = \frac{\partial out}{\partial Bias} * \frac{\partial E}{\partial out} \Rightarrow \quad \delta_{in} = I * \delta_{out} \tag{3}$$

$$\frac{\partial E}{\partial Wt} = \frac{\partial out}{\partial Wt} * \frac{\partial E}{\partial out} \Rightarrow \quad \delta_{in} = \delta_{out} * in^T \tag{4}$$

Activation Layer Forward:

$$out = f(in) \tag{5}$$

Activation Layer Backprop:

$$\frac{\partial E}{\partial in} = \frac{\partial out}{\partial in} * \frac{\partial E}{\partial out} \Rightarrow \quad \delta_{in} = f'(\delta_{out}) \tag{6}$$

C. Parallelization

Upon careful review of the architecture of our library, we could think of multitudinous ways to parallelize the program. We had to stick to implementing only a subset of these ideas in the best interest of time. We tried the following ways to parallelize our code.

- **CPU**: We can trivially observe that most of the computation involved in the training of the neural network are matrix multiplications. A simple technique to parallelize the entire library was to parallelize this step, which takes up a majority of the program time. (gprof run and see)
- **GPU**: All heavy operations for a neural network with dense layers rely on matrix multiplications. So we decided to accelerate the code by implementing matrix multiplication in the GPU as follows:

```
matmul(float*a, float*b, float*res, ..)
{
  // Allocate space, Copy data to GPU
  // Call kernel routine
  // Copy data to CPU
}
```

However, this code ran much slower than its CPU counterpart. We realized that was because of extensive data transfer between the Host and Device. Every matrix multiplication operation resulted in 6 transfers. Each dense layer's forward + backward pass required 3 matmul operations. For 10000 data points running on a 3 layer network, this meant 54000 transfers per epoch!
As a remedy, we decided to first move all data to GPU, then operate on it. So ALL operations had to be implemented in CUDA. To facilitate book-keeping and easy handling, we wrapped all data pointers (in both Host and Device) by a common class 'Tensor' with the following members:

```
class Tensor {
  float* data
  bool is_cuda
  int ndims
  float shape[NDIMS_MAX]
}
```

All data was moved to GPU in the very beginning and each operation on Tensors was implemented for both CPU (C++) and GPU (CUDA). Here is what the matmul operation looked like:

```
matmul(Tensor a, Tensor b, Tensor res,..)
{
  if (is_cuda)  // CUDA matmul routine
    Run matmul_cuda( ... )
  else  // CPU matmul routine
    Run matmul_cpu( ... )
}
```

Other Tensor operations that had to be implemented in CUDA include addition, subtraction, and multiplication for scalar and matrix. A complete list has been added in the appendix. This gave considerable speedups over the CPU code, as expected.

D. Solving a Real World Problem

It is time to put our library to the test! We went ahead with a problem which is relatively simple but still maintains practical relevance in the real world. The handwritten digit recognition problem seems like a good fit, and achieves decent accuracy on relatively simple dense networks. We used the classic MNIST dataset[3] for our experiments and benchmarking our neural network speeds. The following architecture is used for the task in hand. This also sheds light on how easy it is to setup an architecture in our framework.

```
NN nn(784);
nn.add_layer(new Dense(800));
nn.add_layer(new
    Activation(Activations::SIGMOID, 800));
nn.add_layer(new Dense(10));
nn.add_layer(new
    Activation(Activations::SIGMOID, 10));

nn.initialize(0.01, Errors::CROSSENTROPY);
```

III. Image Manipulation by fixed Convolution

In image processing, several objectives like blurring, sharpening, edge detection can be achieved by a convolution between a convolution matrix (also known as kernel or mask) and an image. In a convolution, we sweep the kernel over the image and combine pixels and kernel values with multiplication, then summarize the results with addition. To help us with image utilities like read/write of images, we use the CImg library[4].

A. Algorithm

Convolution with some specific kernels leads to very interesting results. For example, if we use a kernel with all elements as 1, we get a blurring effect and this is known as box blur. As a part of our evaluation, we have implemented kernels that perform box blur, gaussian blur, edge detection, sharpening and embossing.

Convolution over an image

```
for x = 0 to image_x
  for y = 0 to image_y
```

```
    for z = 0 to image_z
      perform single pixel convolution
```

Single Pixel Convolution

```
result = 0
for x = 0 to kernel_x
  for y = 0 to kernel_y
    if image_pixel exists at x, y
      result += kernel_value * pixel_value
output_pixel = result
```

B. Parallelization

We observe that computation at every pixel is independent of the computation at other pixels. We also observe that multiplication in a single pixel computation can be parallelized as long as we ensure that all the multiplication is completed before we summarize and do the addition.

1) GPU Parallelization: We parallelize the convolution using CUDA. We adjust the number of blocks and number of threads such that a single pixel computation is done by one block and single multiplication in this computation is done by one thread. We ensure that all threads terminate by using __syncthreads() before the final addition.

2) CPU Parallelization: We used OpenMP to achieve CPU parallelization through multiple threads. Apart from parallelizing the convolution, we also parallelized the generation of kernels which was not adequeate to parallelize on CUDA due to its relatively small size.

IV. Results

We ran our framework on the MNIST dataset as mentioned before. We present the time taken by the serial code, OpenMP code with different number of threads and CUDA code at various training epoch levels. We also present the time taken for applying effects using kernel on a sample image along with the output images.

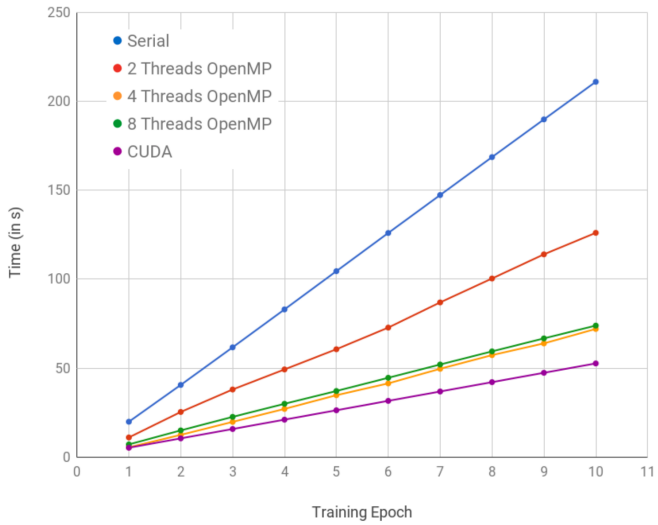

Fig. 1: Neural Network Runtime

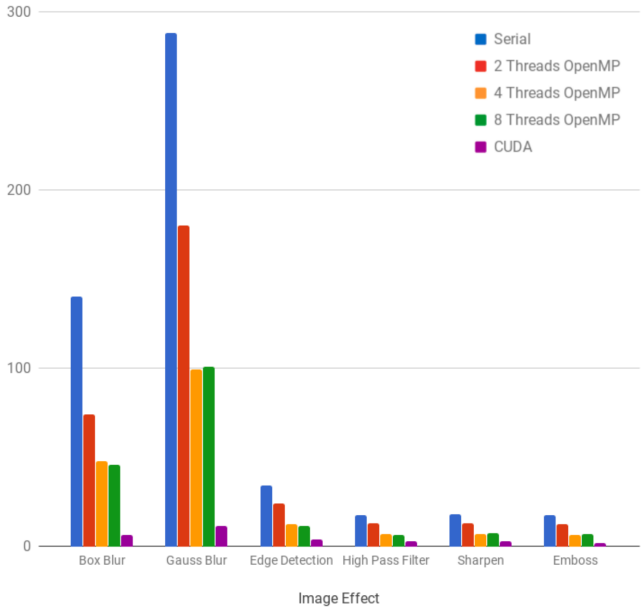

Fig. 2: Convolution Runtime

Epoch	Serial	2 thr	4 thr	8 thr	CUDA	Speedup(CUDA)
1	19.88	11.06	5.51	7.15	5.28	3.77
5	104.51	60.64	34.80	37.16	26.33	3.97
10	210.94	126.06	72.04	73.92	52.66	4.01

Effect	Serial	2 thr	4 thr	8 thr	CUDA	Speedup(CUDA)
Box Blur	140.46	74.24	48.00	45.73	6.53	21.50
Gauss Blur	288.50	180.08	99.60	100.72	11.43	25.24
Edge Detection	34.22	23.98	12.51	11.34	3.79	9.03
High Pass filter	17.77	12.96	7.16	6.46	2.91	6.11
Sharpen	17.85	12.95	6.95	7.32	2.75	6.50
Emboss	17.44	12.27	6.29	7.01	2.13	8.17
Total	517.27	317.23	181.13	179.18	31.19	16.59

Fig. 3: Original Image

Fig. 4: Blur Gauss

Fig. 5: Edge Detection

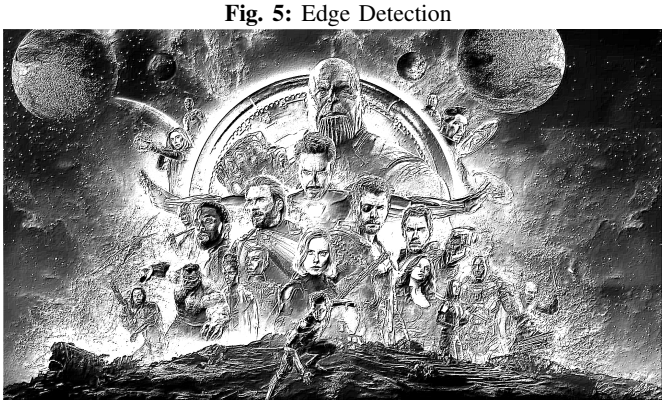

Fig. 6: Emboss

Fig. 7: Highpass Filter

Fig. 8: Sharpen

V. Conclusion

In this paper, we present a small but feature packed neural network framework. This framework was built from scratch and achieves high performance leveraging parallelization from CUDA. We also implemented and evaluated our convolution on a variety of images, producing good results in a small time.

VI. Future Work

A natural immediate extension to our framework would be implementing backprop for our convolution layer. We can also make our convolution and matrix multiplication routines much faster using cache locality. This would open possibilities for our framework to train machine learning models on images. Additionally, there are tons of neural network features that can be added to this framework to bring it closer to full fledged frameworks like PyTorch and Tensorflow. Support for new layers, new activation layers, etc., can be added with no end in sight.

References

[1] K. Keeton and T. Roscoe, Eds., *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/osdi16

[2] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[3] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[4] "The cimg library is a small and open-source c++ toolkit for image processing." https://github.com/dtschump/CImg, accessed: 2010-09-30.