Carnegie Mellon University, Electrical & Computer Engineering Department

Term Project: Report

Fast Convolutional Neural Networks (CNN) for Image Recognition with TensorFlow and GPU Acceleration

18-645: How to Write Fast Code, Spring 2017

Date: 5th May, 2017

Yiang Hua, Vikram Bhattacharjee, Sibo Wang

{yiangh, vbhattac, sibow}@andrew.cmu.edu

Abstract

Convolutional Neural Network (CNN) is the current state-of-the-art deep learning model for image recognition tasks. However, the workloads of CNN are computationally intensive and costly so that GPU parallel acceleration becomes substantial in practical implementation of CNN. In this project, we adopt TensorFlow with its underlying CUDA platform to accelerate the training process of a 9-layer CNN by harnessing the parallel computing power of GPU. The main optimization technique is to lower convolutions to matrix multiplication which has a high ratio of floating-point operations per transferred byte. The experiment uses CIFAR-10 dataset and is performed with NVIDIA GeForce GTX 750 Ti and Intel(R) Core(TM) i7-6700K separately. The result shows that the runtimes are 13,378s and 51,972s for GPU and CPU version respectively. Therefore, the optimization with GPU can achieve **3.88x** speedup compared to CPU. In conclusion, deep learning models such as CNN can achieve much higher training speed with appropriate GPU optimization techniques.

Keywords: CNN, TensorFlow, CUDA, GPU Acceleration, Matrix Multiplication, CIFAR-10

1. INTRODUCTION

Convolutional Neural Networks (CNN) [1] are being widely used in image processing and pattern recognition applications since they have a number of advantages over the generic methods of image recognition. Convolutional neural networks have specialized connectivity structure and they incorporate learning of feature hierarchy. These aspects offer more advantages when compared to general techniques. However, the workloads of convolutional neural networks are computationally intensive and costly so that GPU parallel acceleration becomes substantial in practical implementation of convolutional neural networks. Therefore, this report addresses a fast implementation of the convolutional neural network in an image processing applications through TensorFlow which is an open source library supporting GPU computation for imaging and related recognition tasks and GPU acceleration via CUDA. In addition, we outline the results achieved for the CIFAR-10 [2] dataset. We find that deep learning models such as convolutional neural networks can achieve much higher training speed with proper GPU optimization techniques.

2. RELATED WORK

Convolutional neural networks are gradually becoming popular in the application of image processing and pattern recognition and the focus has shifted to devise faster and efficient applications of the same. Due to very large size of the dataset needed for training purpose, the neural networks need to be optimized from a “method” perspective. Several researchers have tried to improve the efficiency of the convolutional neural networks. Maxime et.al [3] have reported the work that efficiently transfers image representations learned with convolutional neural networks into other visual recognition tasks with limited training data so that it can reduce training work. To achieve this, they designed a method which reused a trained layer to on image dataset to form a mid-level image. Based on this method, they achieved a great improvement in fields of object classification. Other than optimization of the training layers, researchers also tried to improve this class of neural networks by utilizing Fully Convolution Network (FCN). Jonanthan et al. [4] have shown that a fully convolution network trained on semantic segmentation can exceed the state-of-the-art without implementing further machinery. This method has been approved to be an efficient approach both in asymptotical and absolute mannerisms because it precludes the need for the complications in other works, such as patch-wise training etc. In addition, Cordel [5] reported their study on filter bank's influence on convolutional neural networks by using the MNIST dataset. They revealed that when the spatial correlation information among pixels is decreased, the correct rate of recognizing handwritten digits also correspondingly decreases. This increases further scope for research in fields of improved optimization for convolutional neural networks. Wang et al. [6] have explored an optimization of convolutional neural networks based on introducing explicit logical relations between filters in the convolutional layers of the architecture. With this optimized version, the performance on the MNIST dataset is better than many state-of-the-art convolutional neural networks optimization procedures presented so far.

In addition to improve the efficiency of algorithm for convolutional neural networks, some researchers have implemented a more efficient way by fully utilize hardware resource and fast platforms. Alex et al. [7] implemented a deep convolutional neural networks ImageNet on a GPU platform. This is because convolutional neural networks can be implemented in a parallel way and GPU is better at dealing with massive parallel computing. Also, they used 2-D convolution to optimize the GPU implementation. As a result, their network achieves top-1 and top-5 test set error rates of 37.5% and 17.0%. Daniel et al. [8] demonstrated scalable convolutional neural networks based on GPU platform and showed that the computing time can be improved by 2 to 24 times by using the GPU acceleration depending on the type of network training and classification. Other than hardware optimization, there are some researches on optimize convolutional neural networks through architecture perspective. Zhang et al. [9] have reported their implementation of convolutional neural networks with Caffe to accelerate them via FPGAs. The result shows that this hardware/software co-designed library can efficiently get 7.3 times and 43.5 times performance and energy gains compared with traditional implementation. Besides, other people have tried to use Spark platform to accelerate convolutional neural networks. They extracted features from pre-trained convolutional neural networks and returned those features to another full connected neutral network, and they got a very good performance speedup.

All those works fall into several major approaches to optimize convolutional neural networks, either through algorithm optimization, or through platform and hardware optimization, or through both way. It inspired us to further explore a new way to optimize convolutional neural networks by exploiting both hardware and software level optimization. Therefore, we choose the

TensorFlow as our platform designed by Google and has a great potential in deep learning field. Also, we will put forward the gained knowledge on GPU many core optimization into practice since GPU will be an essential hardware platform for image processing in future machine learning.

3. TENSORFLOW AND GPU BASED CONVOLUTIONAL NEURAL NETWORKS

3.1 Fast Platforms

3.1.1 TensorFlow

TensorFlow is an open source software library for deep learning using computational graphs. The nodes in the dynamic graphs represent mathematical operations, while the graph edges represent the multidimensional data arrays (also known as tensors) communicated between them. The flexible architecture allows deployment of computation to one or more CPUs or GPUs with a single API.

3.1.2 The Graphics Processing Unit (GPU)

A graphics processing unit (GPU) is an electronic circuit designed to alter memory involving parallel architecture for accelerating and increasing the total number of floating point operations per second. GPU accelerates the sequential part of the programs through its parallel architecture thereby reducing the memory latency issues invoked during high performance computing in CPUs. GPU consists of streaming microprocessors (SMs) each of which are allocated with local memory for reducing latency in memory operations. They further support instruction-level parallelism. GPU is designed to cover both execution and memory latency for a large amount of clock cycles. GPU invokes a kernel on a grid of parallel thread blocks. Each thread within a block executes an instance on the kernel. Each thread has a unique thread ID within its block and has provisions for corresponding program counters, registers, input and output results.

3.1.3 CUDA

CUDA is a fast parallel programming model and a computing platform devised by NVIDIA which enables dramatic increase in computing performance by harnessing the power of graphics processing unit (GPUs) [10]. The CUDA platform extends basic industry standard C and C++

programming as its interface and passes the instructions directly from the host (CPU) to the device (GPU).

3.1.4 Integration of TensorFlow and CUDA

The CUDA platform supports TensorFlow and can be easily integrated for computing on the GPU platform. This report makes full use of TensorFlow with its underlying CUDA platform. Their relation of integration is illustrated in Figure 1.

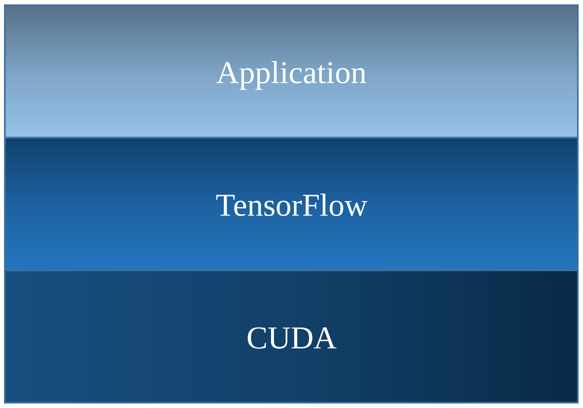

Figure 1. Relation of Integration of TensorFlow and CUDA

3.2 Optimization Techniques

3.2.1 Convolutional Neural Networks

The approach for the recognition techniques is based a feature extractor, the output of which is fed into a neural network. The feature extract is independent of the training procedure and the neural network. Convolutional neural networks (CNN) are the state-of-the-art model topology for image classification tasks. Convolutional Neural Networks apply a series of filters to the raw data of an image to extract and learn higher-level features which in turn can be used by the model for classification [11]. They are composed of three different types of layers namely: 1) Convolutional layers; 2) Pooling layers; 3) Fully connected layers.

A. Convolutional Layers

The convolutional layers are responsible for the feature extraction like edges, corners, end points and non-visual features in other signals. A convolutional layer consists of a number of two-dimensional planes of neurons which represent the feature maps. Each neuron of a feature map is connected to a small subset of neurons inside the feature maps of the previous layer, the so called receptive fields. The receptive fields of neighboring neurons overlap and the weights of these receptive fields are shared through all the neurons of the same feature map. First, the convolution between each input feature map and the respective kernel is computed. Corresponding to the connectivity between the convolutional layer and its preceding layer these convolution outputs are then summed up together with a trainable scalar, known as the bias term. Finally, the result is passed through an activation function. The output $y_n^{(l)}$ of a feature map n in a convolutional layer l is given by (1).

$$y_n^l(x,y) = f^l\left(\sum_{m\in M}\ \sum_{i,j\in R^2|i\le k_x^l;j\le k_y^l} w_{mn}^l(i,j).y_m^{l-1}(x.h^l+i, y.v^l+j)+b_n^l\right) \tag{1}$$

k_x^l and k_y^l are the width and the height of the convolution kernels w_{mn}^l of layer l, and b_n^l is the bias of feature map n in layer l. The set M_n^l contains the feature maps in the preceding layer $l-1$ that are connected to feature map n in layer l. The values h^l and v^l describe the horizontal and vertical step size of the convolution in layer l (usually 1), while f^l is the activation function of layer l.

*B. Pooling Layer*s

To reduce the size of consecutive feature maps a pooling layer is usually placed between two convolutional layers. This type of layer reduces the outputs of a certain number of adjacent neurons (normally a square of 2×2 neurons) of a feature map in the previous layer to a single value. Afterwards it multiplies a single weight to this value, adds a bias and passes the result through an activation function to obtain the result of the output feature map. Pooling layers have the same number of feature maps as the preceding convolutional layer, where each feature map of the pooling layer is always connected to the corresponding one in the previous convolutional layer (1-to-1 connection). The output $y_n^{(l)}$ of a feature map n in the pooling layer l is calculated according to (2).

$$y_n^l(x,y) = f^l\left(w_n^l.\sum_{i,j\in R^2|i\le s_x^l;j\le s_y^l} y_m^{l-1}(x.s_x+i, y.s_y+j)+b_n^l\right) \tag{2}$$

s_x^l and s_y^l are the width and the height of the convolution kernels w_{mn}^l of layer l, and b_n^l is the bias of feature map n in layer l. The value w_n^l is the weight of feature map n in layer l and f^l the activation function of layer l.

C. Fully Connected Layers

After the convolutional and pooling layers one or more fully connected layers follow. These layers are always used to perform classification. In those layers the outputs of all neurons in layer l−1 are connected to every neuron in layer l. The output $y_j^{(l)}$ of neuron j in a fully connected layer l is given by (3).

$$y_n^l(j) = f^l\left(\sum_{i=1}^{N^{l-1}} y^{(l-1)}(i).w^l(i,j) + b^l(j)\right) \tag{3}$$

N^{l-1} is the number of neurons in the preceding layer l−1, $w^l(i,j)$ is the weight for the connection from neuron i in layer l−1 to neuron j in layer l, and $b^l(j)$ is the bias of neuron j in layer l. As for the other two layers, f^l represents the activation function of layer l.

The architecture of the convolutional neural networks model consists of 9 layers (conv1, pool1, norm1, conv2, norm2, pool2, local3, local4, softmax_linear). To elaborate, conv layer is for convolution and rectified linear activation, pool layer is for max pooling, norm layer is for local response normalization. After two similar combinations of conv, pool and norm layers, there are two local layers that are fully connected layer with rectified linear activation. Finally, there is a softmax_linear layer for linear transformation to produce logits. The architecture of convolutional neural networks which we used in the project is illustrated in Figure 2.

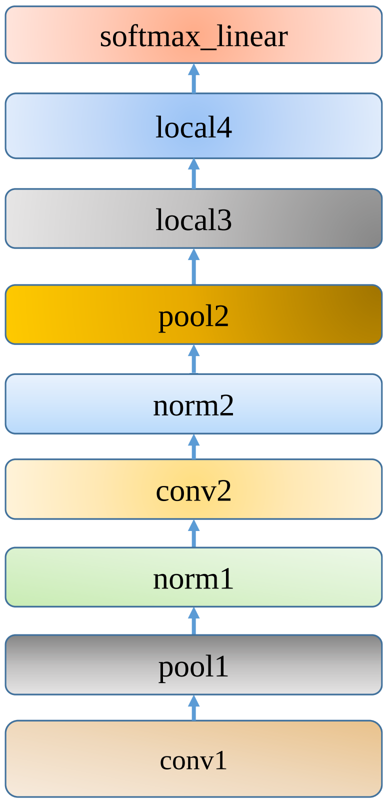

Figure 2. Our CNN Architecture

3.2.2 Lower Convolutions to Matrix Multiplication

The workloads of convolutional neural networks are computationally intensive so that GPU parallel acceleration becomes more and more substantial in practical implementation of convolutional neural networks. Convolutional neural networks are computed by dense kernels which is different from traditional dense linear algebra subroutines. Therefore, providing a highly portable library for optimizing these dense routines can benefit the training of deep neural networks. In this project, we adopt TensorFlow along with its underlying cuDNN library [12]. The goal of optimization is to achieve performance as fast as matrix multiplication without auxiliary memory. There are several approaches to achieve this goal.

One approach is to use Fast Fourier Transform (FFT) to compute convolutions. Fast Fourier Transform can reduce the time and space complexity of convolutions efficiently. However, because of padding filters to inputs, it makes use of a large amount of temporary memory. In addition, it's very costly if the filters are small compared to the inputs in the first few layers of a convolutional neural networks.

Another common way is to compute all convolutions directly, which will be very efficient but need a large amount of customized implementations to deal with corner cases. The most weakness of this approach is its lack of portability, it is well-optimized for convolutions in some parameter spaces but poorly for others.

The final approach we adopt is to lower convolutions to matrix multiplication, which achieves a significant fraction of floating-point operations. It reshapes the filter tensor F into a matrix F_m with dimensions $K \times CRS$ and the image data D into matrix D_m with dimensions $CRS \times NPQ$. The output of this matrix multiplication is matrix O_m with dimensions $K \times NPQ$. An example of the whole procedure is illustrated in Figure 3.

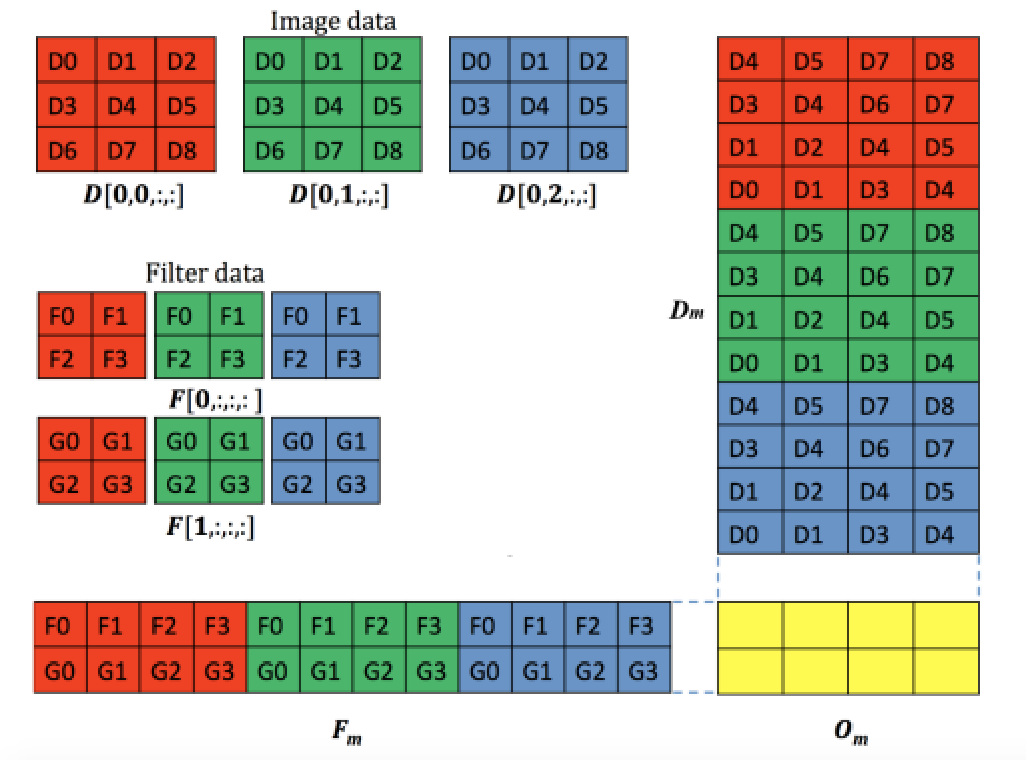

Figure 3. Lowering Convolution to Matrix Multiplication

As shown in figure 3, different colors represent different input feature maps. The filter matrix F_m has the dimensions of $K \times CRS = 2 \times 12$ and image data matrix D_m has the dimensions of $CRS \times NPQ = 12 \times 4$. And the output matrix O_m has the dimensions of $K \times NPQ = 2 \times 4$.

Lowering convolutions to matrix multiplication will bring a huge improvement because the kernels of matrix multiplication are highly optimized. It has a high ratio of floating-point operations per transferred byte. The sizes of F_m and D_m rely on the products of the parameters in the convolution rather than themselves. The approach only requires the product is large enough so that the performance will be consistent. In addition, fixed sized input matrices are read into on-chip memory successively and compute the output matrix. The tiles of input matrices are computed while the next tiles of input matrices are fetched from off-chip memory into on-chip memory, which allows the computation of matrix multiplication to be decided only by the arithmetic time and hide the memory latency with data transfer. Furthermore, this method avoids the problem of costly materialization by materializing image data matrices in on-chip memory only lazily instead of materializing them in off-chip memory before matrix multiplication. Lastly, additional indexing arithmetic is required and it need repeated calculations of modulus and integer division operations. The optimization is to transform these costly modulus and integer division operations into integer multiplication and shift so that the overhead of indexing arithmetic can be reduced. When the computation finishes, the tensor transposition is performed to store the result to the layout as user required.

For future work, the performance of lowering convolutions to matrix multiplication can be further improved up to that attained by complete matrix multiplication. And for machines with multiple GPUs, more speedup can be expected with TensorFlow framework.

4. ANALYSIS OF RESULTS

4.1 Experiments Description

We use TensorFlow 1.1 with GPU to tackle the CIFAR-10 problem, which is a common benchmark in deep learning. The CIFAR-10 dataset is made up of 60000 32x32 color images in 10 classes, with 6000 images per class. The number of training images are 50000 while the number of testing images are 10000. The dataset has been divided into five training sets and one test set, each with 10000 images. The test set contains 1000 randomly-selected images from each class. The training batches contain remaining images in random order, but some of the training sets may contain more images from one class than another. Between them, the training sets contain 5000 images from each class. We installed TensorFlow with GPU on a single machine. The specification of this machine and relevant GPU information is illustrated in Table 1.

Table 1. Hardware Information

Configuration	Detail
CPU	Intel(R) Core(TM) i7-6700K
Cores	4
Frequency	4.00 GHz
Memory	16 G
Disk	256 G
GPU	NVIDIA GeForce GTX 750 Ti
CUDA Version	8.0
CUDA Capability	5.0
Global Memory	2001 MB
GPU Max Clock Rate	1.15 GHz
Multiprocessors	5
Bandwidth	128 bit

The program can be divided into three modules: 1) model inputs; 2) model prediction; 3) model training. The model inputs module first reads images from CIFAR-10 binary data files in disk. The images are cropped into 24x24 pixels for randomly training and approximately whitened to make the model intensive to dynamic range. In addition, a several types of random distortions are applied to images for increasing the data size artificially including randomly flip the image from left to right, randomly distort the image brightness and randomly distort the image contrast.

The model prediction module adds inference operations to compute the logits of the predictions. This module reflects the architecture of the convolutional neural networks which consists of 9 layers (conv1, pool1, norm1, conv2, norm2, pool2, local3, local4, softmax_linear) as we mentioned above.

The model training module adopts softmax regression applying a softmax nonlinearity to the output of the convolutional neural networks and computes the cross entropy between the labels and the normalized predictions. The normal weight decay losses are applied to all learned parameters. The final objective function is the sum of the cross entropy and all L2 regularization terms. For optimization algorithm, batch gradient descent is used with a learning rate decaying exponentially over time. Our goal is to minimize the objective function by computing the gradient and updating the parameters constantly. It should be noted that we execute train.py file to train the model and eval.py file to evaluate the model. For clearly observing the GPU optimization effect, we compared the runtime and the evaluation results for both GPU and CPU versions.

4.2 Results Comparison and Discussion

The comparison of runtime and speedup are essential for our analysis, we present some metrics including runtime and training speed for both GPU and CPU versions in Table 2.

Table 2. System Metrics

Metrics	GPU Version	CPU Version
Runtime	13378s	51972s
Number of examples per second	1890.4	491.3
Number of seconds per batch	0.068	0.261

Figure 4 illustrates the runtime for both GPU and CPU versions more clearly. Because of the significant effect of lowering convolutions to matrix multiplication, the speedup for GPU version compared to CPU version is **3.88x**.

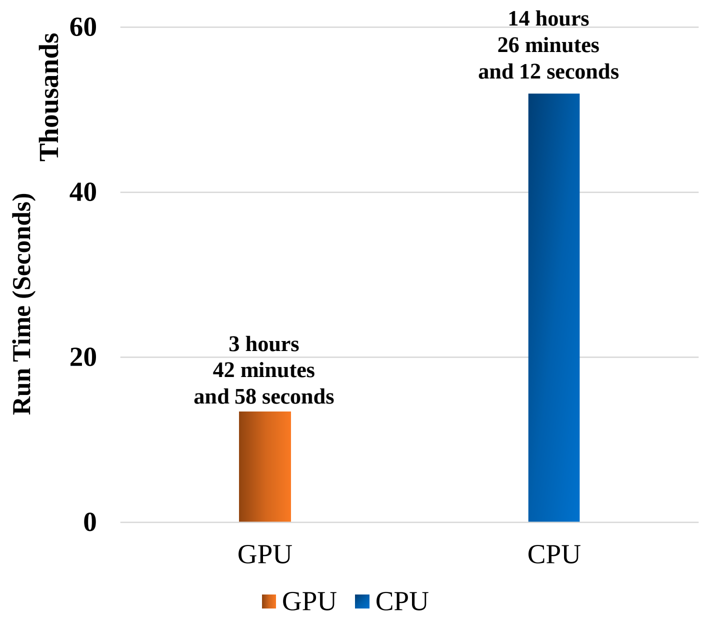

Figure 4. Runtime for Both GPU and CPU Versions

Apart from runtime and speedup, we also care about the ability of the model, i.e. prediction accuracy. After running evaluation program on the 10000 test images, we got 87.3% for GPU version and 87.0% for CPU version, which is shown in Figure 5.

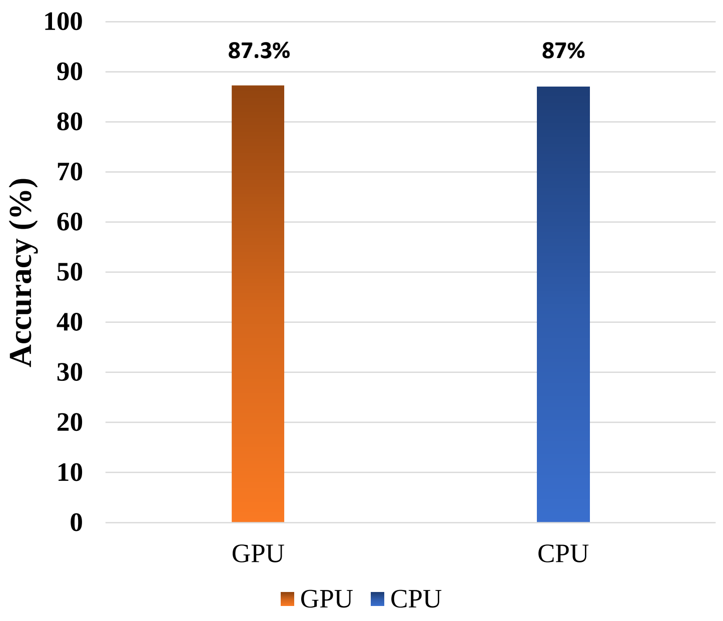

Figure 5. Accuracy for Both GPU and CPU Versions

The descent processes of the objective loss function during training for both versions are illustrated in Figure 6. The shapes of loss function for both versions are similar and all decrease from approximately 3.5 to 0.3 after 200k training steps.

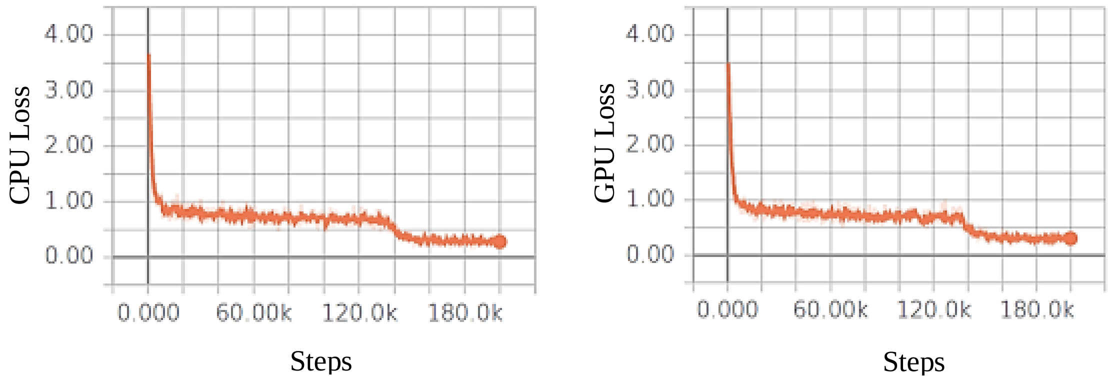

Figure 6. Loss Function for Both GPU and CPU Versions

5. CONCLUSION

Convolutional Neural Networks are being widely used in image processing and pattern recognition applications. However, the workloads of convolutional neural networks are computationally intensive so that GPU parallel acceleration becomes significant in implementation of convolutional neural networks. In this project, we adopt TensorFlow with GPU to accelerate the training process of a 9-layer convolutional neural. The main optimization technique is to lower convolutions to matrix multiplication which has a high ratio of floating-point operations per transferred byte. In addition, the tiles of input matrices are computed while the next tiles of input matrices are fetched from off-chip memory into on-chip memory, which hides the memory latency with data transfer. The result shows that the optimization with GPU can achieve 3.88x speedup compared to CPU. Therefore, deep learning models such as convolutional neural networks can achieve much higher training speed with appropriate optimization techniques such as transforming computationally intensive operations into highly-optimized GPU routines.

REFERENCES

[1]. Convolutional Neural Network. Source:https://en.wikipedia.org/wiki/Convolutional_neural_network

[2]. CIFAR-10 Dataset. Source: http://www.cs.toronto.edu/~kriz/cifar.html

[3]. Oquab, Maxime, et al. "Learning and transferring mid-level image representations using convolutional neural networks." Proceedings of the IEEE conference on computer vision and pattern recognition. 2014.

[4]. Long, Jonathan, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2015.

[5]. Cordel, I. I., O. Macario, and Arnulfo P. Azcarraga. "Measuring the contribution of filter bank layer to performance of convolutional neural networks." International Journal of Knowledge-based and Intelligent Engineering Systems 21.1 (2017): 15-27.

[6]. Wang, Hanli, Peiqiu Chen, and Sam Kwong. "Building Correlations Between Filters in Convolutional Neural Networks." IEEE Transactions on Cybernetics (2016).

[7]. Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

[8]. Strigl, Daniel, Klaus Kofler, and Stefan Podlipnig. "Performance and scalability of GPU-based convolutional neural networks." Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on. IEEE, 2010.

[9]. Zhang, Chen, et al. "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks." Proceedings of the 35th International Conference on Computer-Aided Design. ACM, 2016.

[10]. NVIDIA GPU. Source: http://www.nvidia.com/object/cuda_home_new.html#sthash.vD6KlbYg.dpuf.

[11]. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient Based Learning Applied to Document Recognition," in Proc. of the IEEE, vol. 86, no. 11, 1998, pp. 2278–2324.

[12]. Chetlur, Sharan, et al. "cudnn: Efficient primitives for deep learning." arXiv preprint arXiv:1410.0759 (2014).