IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 2 Issue 11, November 2015.
www.ijiset.com
ISSN 2348 – 7968

Implementing Convolutional Neural Network with Parallel Computing using CUDA

Shashank Bajpai
Assistant Manager - Security Strategy and Cloud Security, Reliance Jio Infocomm Limited (A 4G Telecom Enterprise of RIL)
Researcher in areas of High Performance Computing, Parallel Computing, Cloud Security, Telecom and Cyber Security
Email: shashank.bajpai@ril.com, Website: https://about.me/shashank.bajpai

***Abstract*—Convolutional Neural Network is a multi-layered approach to image stabilization and noise reduction. This can be implied for advanced image analysis and large scale image detection used for various purposes pertaining to National Security. Parallel Environment is a computing scenario wherein Multi-Core architecture of present day CPU and GPU is optimally used for faster and precise calculations. Parallel Programming includes various libraries to divide and execute threads in parallel in different cores. In my research project I aim to implement and execute Convolutional Neural Network in a Parallel Environment. For this a GPU with multi-core hardware is a pre-requisite. Also the corresponding parallel architecture libraries will be required to be installed and their usage well understood.**

Keywords*—Convolutional Neural Network (CNN), GPU, CUDA, Back Propagation Algorithm.

I. INTRODUCTION

Neural Networks (NNs) perform well on pattern recognition tasks with a large amount of training data. For image classification, like Optical Character Recognition (OCR), Convolution Neural Networks (CNNs) deliver optimum performance. The area of applications for CNNs is widespread. They are used for handwriting recognition, face, eye and license plate detection.

The prominent shortcoming of Neural Network is that it takes long time for training. It includes large number of floating point operations and data transfer rate is usually low. So CNN is implemented on GPGPU (General Purpose GPU) computation on current Graphic Processing Units (GPUs)[1], GPU has advantage over CPU of high computational throughput at low cost, achieved through their parallel architecture.

I am developing a complete software solution that harnesses computing capability of Graphic Processing Unit to stabilize and analyze a given image. This solution can be ultimately incorporated as a module to serve federal and international security purposes. I wish to develop a quicker and more precise methodology by combining the concepts of neural network and parallel programming.

II. CUDA ARCHITECTURE

CUDA stands for Compute Unified Device Architecture. CUDA technology leverages parallel processing power of NVIDIA GPUs. The CUDA architecture is a parallel computing architecture that delivers the performance of NVIDIA's graphics processor technology to general purpose GPU Computing [2]. Applications that run on the CUDA architecture can take advantage of an installed base of over one hundred million CUDA-enabled GPUs in desktop and notebook computers, professional workstations, and supercomputer clusters.

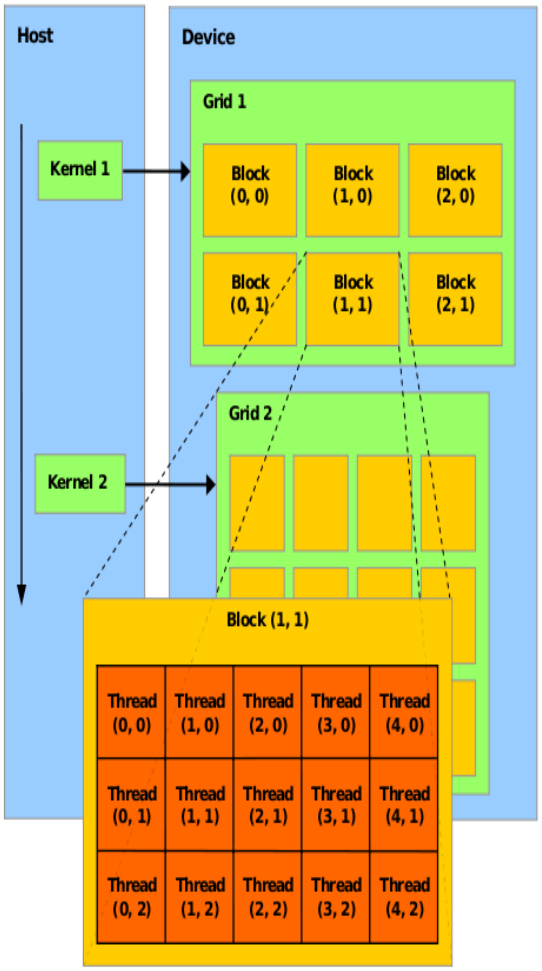

Fig. 1. CUDA Architecture

CUDA Architecture splits the device into grids, blocks and threads in a hierarchical structure as shown in fig. 1. Since there are a number of threads in one block and a number of blocks in one grid and a number of grids in one GPU, the parallelism that is achieved using such a hierarchical architecture. Grid is a group of threads all running the same kernel. These threads are not synchronized. Every call to CUDA from CPU is made through one grid. Starting a grid on CPU is a synchronous operation but multiple grids can run at once [4]. Grids are composed of blocks. Each block is a logical unit containing a number of coordinating threads, a certain

ISSN 2348 – 7968

amount of shared memory. Blocks are composed of threads. Threads are run on the individual cores of the multiprocessors. Threads have a certain amount of register memory. Usually there can be 512 threads per block.

III. CONVOLUTIONAL NEURAL NETWORK (CNN)

CNN is composed of three different types of layers: convolution layers, sub sampling layers (optional), and fully connected layers.

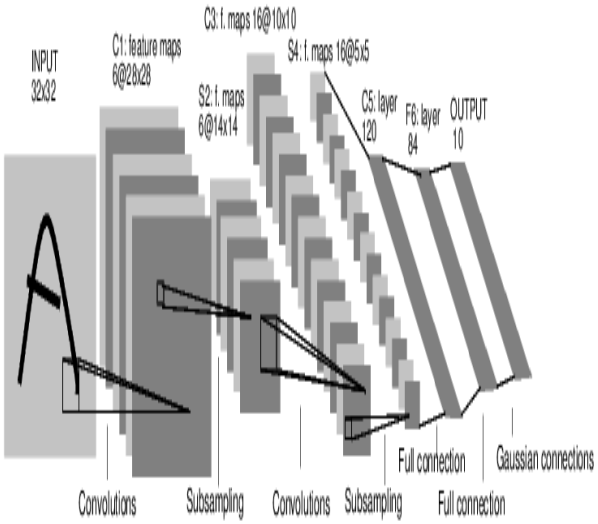

Fig. 2. Architecture of the LeNet5

These layers are arranged in a feed forward Structure.

A. Convolution Layer

The convolution layers are responsible for the feature extraction (edges, corners, end points or non-visual features in other signals), two key concepts of local receptive fields and shared weights Convolution layers apply a small set of filters all over their input "images".

B. Sub sampling Layer

To reduce the size of consecutive feature maps a sub-sampling layer is usually placed between two convolution 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.

C. Fully Connected Layer

After the convolution and sub sampling layers one or more fully connected layers follow. In those layers the outputs of all neurons in layer l - 1 are connected to every neuron in layer. A fully-connected layer simply multiplies its input by a weight matrix. [1] Average is considered for analysis.

IV. BACK PROPAGATION ALGORITHM

The back propagation algorithm trains a given feed-forward multilayer neural network for a given set of input patterns with known classifications. When each entry of the sample set is presented to the network, the network examines its output response to the sample input pattern. The output response is then compared to the known and desired output and the error value is calculated. Based on the error, the connection weights are adjusted. The back propagation algorithm is based on Widrow-Hoff delta learning rule in which the weight adjustment is done through mean square error of the output response to the sample input. The set of these sample patterns are repeatedly presented to the network until the error value is minimized.

We want to train a multi-layer feed forward network by gradient descent to approximate an unknown function, based on some training data consisting of pairs (x, t). The vector x represents a pattern of input to the network, and the vector t the corresponding target (desired output). As we have seen before, the overall gradient with respect to the entire training set is just the sum of the gradients for each pattern; in what follows we will therefore describe how to compute the gradient for just a single training pattern [3].

A. Definitions

1) *the **error** signal for unit j:*

$$\delta_j = -\partial E / \partial net_j$$

2) *the (negative) **gradient** for weight w_{ij}:*

$$\Delta w_{ij} = -\partial E / \partial w_{ij}$$

3) *the set of nodes **anterior** to unit i:*

$$A_i = \{j : \exists\, w_{ij}\}$$

4) *the set of nodes **posterior** to unit j:*

$$P_j = \{i : \exists\, w_{ij}\}$$

B. The gradient

We expand the gradient into two factors by use of the chain rule

$$\Delta w_{ij} = -\frac{\partial E}{\partial net_i}\frac{\partial net_i}{\partial w_{ij}}$$

The first factor is the error of unit I and the second –

$$\frac{\partial net_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_{k \in A_i} w_{ik}\, y_k = y_j$$

Putting the two together, we get:

$$\Delta w_{ij} = \delta_i\, y_j$$

C. Forward activation

The activity of the input units is determined by the network's external input **x**. For all other units, the activity is propagated forward:

$$y_i = f_i\left(\sum_{j \in A_i} w_{ij}\, y_j\right)$$

Note that before the activity of unit i can be calculated, the activity of all its anterior nodes (forming the set Ai) must be

ISSN 2348 – 7968

known. Since feed forward networks do not contain cycles, there is an ordering of nodes from input to output that respects this condition.

D. *Calculating output error*

Assuming that we are using the sum-squared loss

$$E = \frac{1}{2}\sum_{o}(t_o - y_o)^2$$

Error for output unit o is simply

$$\delta_o = t_o - y_o$$

E. *Error back propagation*

For hidden units, we must propagate the error back from the output nodes (hence the name of the algorithm). Again using the chain rule, we can expand the error of a hidden unit in terms of its posterior nodes:

$$\delta_j = -\sum_{i \in P_j} \frac{\partial E}{\partial net_i}\frac{\partial net_i}{\partial y_j}\frac{\partial y_j}{\partial net_j}$$

Of the three factors inside the sum, the first is just the error of node i. The second is

$$\frac{\partial net_i}{\partial y_j} = \frac{\partial}{\partial y_j}\sum_{k \in A_i} w_{ik}\, y_k = w_{ij}$$

While the third is the derivative of node j's activation function:

$$\frac{\partial y_j}{\partial net_j} = \frac{\partial f_j(net_j)}{\partial net_j} = f_j'(net_j)$$

For hidden units h that use the tanh activation function, we can make use of the special identity tanh (u)' = 1 - tanh (u) ^{2}, giving us

$$f_h'(net_h) = 1 - y_h^2$$

Putting all the pieces together we get

$$\delta_j = f_j'(net_j)\sum_{i \in P_j}\delta_i\, w_{ij}$$

Note that in order to calculate the error for unit j, we must first know the error of all its posterior nodes. Again, as long as there are no cycles in the network, there is an ordering of nodes from the output back to the input that respects this condition. For example, we can simply use the reverse of the order in which activity was propagated forward.

V. APPLICATION

The General-purpose computing on graphics processing units [GPGPU] is far more superior in terms of computing and execution of parallel tasks when compared with multi-core CPUs of today. Dark Silicon phase is used to describe the growing gap between area gains and power gains [5]. Thereby predicting an end to multi-core scaling with a follow-up to the end of frequency scaling. These new dimensions are all cumulative in the field of Processor Development. We wish to optimize the use of present day resources not only in the field of software applications but also in system applications and increase the hardware throughput [6].

A. *Code Snippets*

Following are a few snippets of the ConvNet source used in my application [7]. The base source code is in CUDA while a wrapper UI for application usage is built with Python and Shell scripts.

```
class ConvNet(IGPUModel):
    def __init__(self, op, load_dic, dp_params={}):
        filename_options = []
        dp_params['multiview_test'] = op.get_value
('multiview_test')
        dp_params['crop_border'] = op.get_value('crop_border')
        IGPUModel.__init__(self, "ConvNet", op, load_dic,
filename_options, dp_params=dp_params)

    def import_model(self):
        lib_name = "pyconvnet" if is_windows_machine() else
"_ConvNet"
        print "========================="
        print "Importing %s C++ module" % lib_name
        self.libmodel = __import__(lib_name)

    def init_model_lib(self):
        self.libmodel.initModel(self.layers,
self.minibatch_size, self.device_ids[0])
```

Fig. 3. Convnet.py – Python Wrapper

```
#include <vector>
#include <iostream>
#include <string>

#include <nvmatrix.cuh>
#include <nvmatrix_operators.cuh>
#include <matrix.h>
#include <convnet.cuh>
#include <util.cuh>

using namespace std;

/*
 * =======================
 * ConvNet
 * =======================
 */
ConvNet::ConvNet(PyListObject* layerParams, int minibatchSize,
int deviceID) : Thread(false),  _deviceID(deviceID), _data
(NULL) {
    try {
        int numLayers = PyList_GET_SIZE(layerParams);

        for (int i = 0; i < numLayers; i++) {
            PyObject* paramsDict = PyList_GET_ITEM
(layerParams, i);
            string layerType = pyDictGetString(paramsDict,
"type");
```

Fig. 4. Convnet.cu – Cuda source

CONCLUSION

This research document aims at concluding that the GPUs work quite well for implementing convolutional neural networks. The relatively low amount of data to transfer to the GPU for every pattern and the big matrices that have to be handled inside the network seem to be appropriate for GPGPU processing. Furthermore, my experiments showed that the GPU implementation scales much better than the CPU implementations with increasing network size.

ACKNOWLEDGMENTS

I take this opportunity to express my gratitude towards my previous employer organization Center for Development of Advanced Computing (CDAC), Govt. of India, for exposure and support for research in High Performance Computing and Parallel Computing with deployment in distributed Grid Network (GARUDA). I have effectively implemented several complex computational tasks in Tesla GPU with CUDA and

ISSN 2348 – 7968

executed the same within a given time frame. I am also thankful for Reliance Industries Ltd. for continual motivation for me to innovate and research in spite of strict managerial timelines and also the NVIDIA Developer Community for the various updates and continual support.

REFERENCES

[1] Stefan Podlipnig Daniel Strigl, Klaus Kofler, “Performance and scalability of GPU based convolutional neural networks.”, IEEE 2010

[2] Rafia Inam, Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden,” An Introduction to GPGPU Programming - CUDA Architecture”

[3] ErrorBackpropagation http://www.willamette.edu/~gorr/classes/cs449/backprop.html

[4] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, Amit Bawaskar, ”GPGPU Processing in CUDA Architecture”, An International Journal (ACIJ), Vol.3, No.1, January 2012

[5] Hadi Esmaeilzadehy, Emily Blem, Rene St. Amant, Karthikeyan Sankaralingamz, Doug Burger, “Dark silicon and the end of multicore scaling.”, appears in the Proceedings of the 38th International Symposium on Computer Architecture (ISCA ’11)

[6] Naga Vydyanathan, Amit Kale, Bharatkumar Sharma, Rahul Thota, “Towards a robust, real-time face processing system using CUDA-enabled GPUs”, IEEE 2012

[7] Alex Krizhevsky (akrizhevsky@gmail.com) Source of ConvNet https://code.google.com/p/cuda-convnet/