Implementation of Neural Network Backpropagation in CUDA

Jinfeng Liu* and Lei Guo

School of Automation, Northwestern Polytechnical University, Xian 710129, China
ljf@sjtu.org

Abstract. Training a multilayer Neural Network with Backpropagation algorithm is usually a very time consuming processing. In this paper, we propose an approach which uses CUDA programming model and exploits the computing power of Graphic Processing Units (GPUs) to accelerate the Backpropagation process. Experiments show that this method can achieve up to 7 times of speedup over the CPU counterpart.

Keywords: Neural Network, Backpropagation, GPU, CUDA.

1 Introduction

An Artificial Neural Network (ANN) is a mathematical model or computational model that tries to simulate the structure and/or functional aspects of biological neural networks. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. In most cases an ANN is an adaptive system that changes its structure based on external or internal information that flows through the network during the learning phase. Modern neural networks are non-linear statistical data modeling tools. They are usually used to model complex relationships between inputs and outputs or to find patterns in data.

Forward propagation is the process whereby each of the neurons calculates its output value, based on inputs provided by the output values of the neurons that feed it. Backpropagation is an iterative process that starts with the last layer and moves backwards through the layers until the first layer is reached. Assume that for each layer, we know the error in the output of the layer. If we know the error of the output, then it is not hard to calculate changes for the weights, so as to reduce that error. The problem is that we can only observe the error in the output of the very last layer.

Backpropagation gives us a way to determine the error in the output of a prior layer given the output of a current layer. The process is therefore iterative: start at the last layer and calculate the change in the weights for the last layer. Then calculate the error in the output of the prior layer.

Since a Neural Network requires a considerable number of vector and matrix operations to get results, it is very suitable to be implemented in a parallel programming

* Corresponding author.

Z. Du (Ed.): Intelligence Computation and Evolutionary Computation, AISC 180, pp. 1021–1027.
springerlink.com

model and run on Graphics Processing Units (GPUs). Our goal is to utilize and unleash the power of GPUs to boost the performance of a Neural Network.

2 A Brief Introduction to CUDA

The extraordinary GPU computing power is very attractive to general-purpose system development, which is referred to as general-purpose computing on GPUs (GPGPU). The first generation of GPGPU requires that non-graphics application must follow the general flow of the graphics pipeline (consisting of vertex, geometry and pixel processors), and memory access also has many restrictions, so it is very difficult to program.

The Compute Unified Device Architecture (CUDA) [1] from NVIDIA is a new technique for GPGPU. CUDA provides useful features for implementing generic data-parallel computing:

(1) Code is ANSI C extended with some keywords and data structures that eliminates the need of mapping the application to graphics API;
(2) Code can read from arbitrary addresses in memory;
(3) Full support for integer and bitwise operations.

In CUDA, the GPU is regarded as a coprocessor capable of executing a great number of threads in parallel. A single source program consists of the *host code* to be executed on the CPU and the *kernel code* to be executed on the GPU. The kernel code is usually computational intensive and data-parallel and is executed as many different *threads*. In CUDA, threads are organized into *thread blocks*, a thread block can have at most 512 threads, and threads belonging to the same thread block can share data through the shared memory and can perform barrier synchronization.

The hardware implementation of the device is a set of Streaming Multiprocessors (SMs). Each SM has a Single Instruction Multiple Data (SIMD) architecture; typically it has 8 processors that execute the same instruction on different data (threads). The blocks being processed in a multiprocessor are divided into SIMD groups of threads called *warps*.

The GPU's memories have different types in CUDA; they are specialized and have different access times and throughput limitations.

Global memory is a large, long-latency memory that exists physically as off-chip dynamic RAM (DRAM); it can be accessed by all threads in all blocks. *Shared memory* is a low latency, limited-capacity memory which can be only accessed by threads in a block. It is useful for data that can share and reuse to eliminate redundant accesses to the global memory. The *constant memory* uses a small cache of a few kilobytes optimized for high temporal locality and accesses by large numbers of threads across multiple thread blocks. The texture memory space uses the GPU's texture caching and filtering capabilities, and is best utilized with data access patterns exhibiting 2-D locality.

The properties of the different types of memories and the Suitability of CUDA have been summarized in [2][3].

3 Implementation of Backpropagation in CUDA

3.1 The Structure of ANN Used in This Paper

Fig.1 is an illustration of the neural network[4][5][6]. The input layer (Layer #1) is the grayscale image of 29x29 pixels. There are 29x29 = 841 neurons in the input layer. Layer #2 is a convolution layer with six feature maps. Each feature map is sized to 13x13 neurons. Each neuron in each feature map is a 5x5 convolution kernel of the input layer. Layer #3 is also a convolution layer with 50 feature maps. Each feature map is 5x5, and each unit in the feature maps is a 5x5 convolution kernel. Layer #4 is a fully-connected layer with 100 units. Each of the 100 neurons in the layer is connected to all 1250 neurons in the previous layer. Layer #5 is the final, output layer. This layer is a fully-connected layer with 10 units.

Altogether, there are a total of 3215 neurons in the neural network, 134066 weights and 184974 connections. The object is to train all 134066 weights so that, for an arbitrary input at the input layer, there is exactly one neuron at the output layer whose value is +1 whereas all other nine neurons at the output layer have a value of -1.

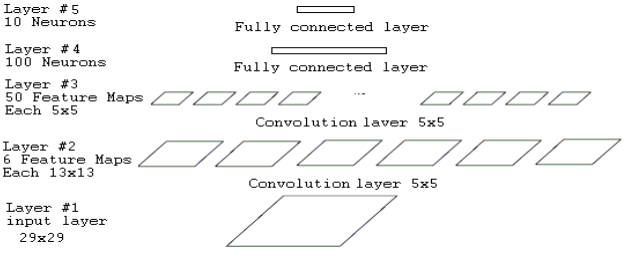

Fig. 1. The structure of ANN

3.2 General Description of Backpropagation Algorithm

Backpropagation algorithm consists of two phases: in the feed forward pass, an input vector is presented to the network and propagated forward to the output; in the backpropagation phase, the network output is compared to the desired output; network weights are then adjusted in accordance with an error-correction rule.

Steps of Backpropagation algorithm:

1) Initialization. Assign random numbers to synaptic weights, using a uniform random distribution.

2) Forward Propagation. Forward propagation is the process whereby each neuron calculates its output value based on inputs and the weight of connection indicated in equation (1)

$$x_n^i = F(y_n^i) = F\left(\sum_{j=0}^{C_{n-1}} \omega_n^{ij} \cdot x_{n-1}^j\right). \quad (1)$$

Where: x_n^i is the output of the i-th neuron in layer n, x_{n-1}^j is the output of the j-th neuron in layer n-1, ω_n^{ij} is the weight that the i-th neuron in layer n applies to the output of the j-th neuron from layer n-1. F()is the activation function.

3) Backpropagation computation:

The error due to a single pattern is calculated as follows:

$$E_n^p = \frac{1}{2} \cdot \sum (x_n^i - T_n^i)^2 \tag{2}$$

Where: E_n^p is the error due to a single pattern P at the last layer n; T_n^i is the desired output at the last layer; and x_n^i is the actual value of the output at the last layer.

Given equation (1), then taking the partial derivative yields:

$$\frac{\partial E_n^p}{\partial x_n^i} = x_n^i - T_n^i. \tag{3}$$

Equation (3) calculates numeric values for the derivative. We calculate the changes in the weights by applying the following two equations (4) and then (5):

$$\frac{\partial E_n^p}{\partial y_n^i} = G(x_n^i) \cdot \frac{\partial E_n^p}{\partial x_n^i} \tag{4}$$

Where: $G(x_n^i)$ is the derivative of the activation function.

$$\frac{\partial E_n^p}{\partial \omega_n^{ij}} = \frac{\partial E_n^p}{\partial y_n^i} \cdot \frac{\partial y_n^i}{\partial \omega_n^{ij}} = x_{n-1}^j \cdot \frac{\partial E_n^p}{\partial y_n^i} \tag{5}$$

We calculate the error for the previous layer, using the following equation:

$$\frac{\partial E_{n-1}^p}{\partial x_{n-1}^k} = \sum_i \omega_n^{ik} \cdot \frac{\partial E_n^p}{\partial y_n^i} \tag{6}$$

We take the numeric values obtained from equation (6), and use them in a repetition of equations (4), (5) and (6) for the immediately preceding layer. Meanwhile, the values from equation (5) tell us how much to change the weights in the current layer n, we update the value of each weight according to the formula:

$$(\omega_n^{ij})_{new} = (\omega_n^{ij})_{old} - \eta \cdot \frac{\partial E_n^p}{\partial \omega_n^{ij}} \tag{7}$$

Where: η is the "learning rate", typically a small number that is gradually decreased during training.

3.3 Implementation Details of Backpropagation in CUDA

3.3.1 Forward Propagation Computing

Because of the inherent structure limitation of ANN, parallelism can only be achieved within one level. For each network level, there is a CUDA function handling the

parallelism computation of neuron values of that level, the connections of the Neural Network are implicitly defined in CUDA functions.

Five arrays are defined to save the value of input layer and the weights between each layer, and transfer these arrays from host memory to device memory. The neuron values of layer 2 to layer 5 are allocated on device memory. The first kernel function calculates neuron values of layer 2. The second kernel function calculates neuron values of layer 3, and so on. So there are four kernel functions on Forward Propagation stage.

The pseudo kernel code of compute layer #2 is shown in Fig.2. Each block deal with one feature map and each thread compute one neuron value. This code is called by the host code by setting parameter of block with 6 and thread with 169.

Line 1 in Fig.2 gets the block ID and the thread ID, Line 3 defines a shared memory array s1, line 4 puts all elements of array gn1 to array s1, line 5 calculate the activation value by adding the weighted inputs together, it has two benefits to increase speed: first, loop unrolling eliminates the branch instruction and the loop counter update, second, array s1 is defined in shared memory and array gn1 is in global memory, using s1 instead of gn1 can effectively reduce access time. Line 6 applies an activation calculation. Line 7 puts result to array gn2, which contains the neuron value of layer #2.The other three kernel functions are basically similar with the first kernel function.

```
input：array gn1 contain neuron value of input layer, array gw1 contain
       weight between input layer and layer #2.
output：array gn2 is neuron value of layer #2.
```

```
1：bid=blockIdx.x;  tx=threadIdx.x;  ty=threadIdx.y;
2：wt=ty*2*29+tx*2;  result=0;
3：__shared__ float s1[29*29];
4：s1[] = gn1[]；//put all elements of array gn1 to array s1
5 ： result = s1[wt]*gw1[bid*26+1]+ s1[wt+1]*gw1[bid*26+2]+
s1[wt+2]*gw1[bid*26 +3]+⋯ // a total of 25 multiplication
6： result=(1.7159*tanhf(0.66666667*result));
7： gn2[13*13*bid+ty*13+tx]=result;
```

Fig. 2. Pseudo kernel code of calculating layer #2

3.3.2 Backpropagation Computation

1) Calculate output error of layer #5 using equations (4). Fig.3 is the pseudo kernel code which is called in the host code by setting block (10, 1) and thread (1, 1)

2) Adjust the weights in layer #4. Fig.4 is the pseudo kernel code which is called in the host code by setting block(10,1)and thread(100,1).

Line 1 in Fig.4 gets the block ID and the thread ID, line 3 and line 4 calculate the changes of weights in layer #4 according equations (5), line 5 adjust weights according equations (7), eta is the learning rate.

3) Calculate output error of #4 according equation (6). Fig. 5 is the pseudo kernel code which is called in the host code by setting block(100,1)and thread(10,1).

Line 1 in Fig.5 gets the block ID and the thread ID, line 2 defines a shared memory array st, in line 3 each thread in one block calculate one part of result and store in

array st, line 4 lets these threads synchronized. In line 5 means if the thread ID is 0, line 6 add ten parts of result together.

4) Adjust the weights in layer #3.Fig. 6 is the pseudo kernel code which is called in the host code by setting block(1251,1) and thread(100,1).

```
input：gn5 is the actual output value of layer #5,tn5 is the desired output of layer #5
output：dx5 is the output error of layer #5
1:  bid=blockIdx.x ;
2:  dx5[bid]= DSIGMOID(gn5[bid])*(gn5[bid]-tn5[bid]);
```

Fig. 3. Pseudo kernel code of calculate output error of layer #5

```
input：dx5 is the output error of layer #5,gn4 is the neuron value of layer #4,gw4 is the weights of layer #4
output：newdw4 is adjusted weights of layer #4
1: bid=blockIdx.x;   tx=threadIdx.x;
2: wbegin= bid *101;
3: dw4[wbegin]+=dx5[bid];
4: dw4[wbegin+tx+1]+=dx5[tx]*gn4[tx];
5: newgw4[wbegin+tx+1]=gw4[wbegin+tx+1] –eta* dw4[wbegin+tx+1]
```

Fig. 4. Pseudo kernel code of adjust weights in layer #4

```
input：gw4 is the weights of layer #4,gn4 is the neuron value of layer #4,dx5 is the output error of layer #5
output：dx4 is the output error of layer #4
1: bid=blockIdx.x;   tx=threadIdx.x;
2: __shared__ float st[10];
3: st[tx]=dx5[tx]*gw4[tx*101+1+ bid];
4: __syncthreads();
5: if(tx==0){
6:     dx4[bid]= st[0]+st[1]+…st[9];
7:     dx4[bid]= DSIGMOID(gn4[bid])*dx4[bid];   }
```

Fig. 5. Pseudo kernel code of calculate output error of layer #4

```
input：dx4 is the output error of layer #4,gn3 is the neuron value of layer #3.gw3 is the weights of layer #3.
output：newgw3 is adjusted weights in layer #3
1: bid=blockIdx.x ;
2: tx=threadIdx.x；
3: dw3[1251*tx+bid]=dx4[tx]*gn3[bid-1];
4: newgw3[1251*tx+bid]= gw3[1251*tx+bid] –eta* dw3[1251*tx+bid];
```

Fig. 6. Pseudo kernel code of adjust weights in layer #3

5) The rest calculations

Calculate output error of layer #3; the kernel code is similar to Fig.5. Adjust the weights in layer #2; the kernel code is similar to Fig.6. Calculate output error of layer #2; the kernel code is similar to Fig. 5. Adjust the weights in layer #1; the kernel code is similar to Fig. 6.

4 Experimental Results

We have implemented the Neural Network Backpropagation described above on NVIDIA GeForce GTX9800+.This GPU has 16 SMs, 128 1.836GHz processing cores, with 896MB onboard memory. The GPU is installed on a desktop computer equipped with a 2.8GHz Core 2 Duo CPU E7400.

One pass of Backpropagation computation spend CPU 7.457ms, and spend GPU 1.091ms. The GPU gain about 7 times speedup than CPU.

5 Conclusions

Calculation the weight of multilayer complex Neural Network using backpropagation is a very time consuming processing, it may take several hours or more than one day if uses common algorithm executed on CPU. In this paper, we mainly describe an approach of calculating backpropagation using GPU in CUDA. Experiments show that this approach can greatly reduce computation time. This paper both has practical use in backpropagation calculation and reference value for general GPU computation.

References

[1] NVIDIA Corporation. NVIDIA CUDA programming Guide Version 2.2 (April 2009), http://developer.nvidia.com/cuda

[2] Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.-M.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of ACM PPOPP 2008, pp. 73–82 (2008)

[3] Hwu, W.-M.W., Rodrigues, C., Ryoo, S., Stratton, J.: Compute Unified Device Architecture Application Suitability. Computing in Science and Engineering 11(3), 16–26 (2009)

[4] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE 86(11), 2278–2324 (1998)

[5] Simard, P.Y., Steinkraus, D., Platt, J.: Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In: International Conference on Document Analysis and Recognition, pp. 958–962. IEEE Computer Society (2003)

[6] O'Neill, M.: Neural Network for Recognition of Handwritten Digits (2006), http://www.codeproject.com/KB/library/NeuralNetRecognition.aspx