TRAINING AND APPLYING A FEEDFORWARD
MULTILAYER NEURAL NETWORK IN GPU

Klaus Raizer
klaus@fem.unicamp.br

Hugo Sakai Idagawa
hugoidagawa@gmail.com

Prof. Dr. Eurípedes Guilherme de Oliveira Nóbrega
egon@fem.unicamp.br

Prof. Dr. Luiz Otávio Saraiva Ferreira
lotavio@fem.unicamp.br

Department of Computational Mechanics, Mechanical Engineering Faculty,
State University of Campinas, Campinas, SP, Brazil

Abstract. This work presents the implementation of Feedforward Multi-Layer Perceptron (FFMLP) Neural Networks on General-purpose computing graphics processing units (GPU) by using CUDA programming language (Compute Unified Device Architecture). The paper presents an introduction to how data is decomposed and processed in a GPU in order to take advantage of its powerful parallel processing capabilities. A feedforward multi-layer perceptron is briefly presented and explained. Since a neural network must be trained in order to solve a particular problem, this work implements a variation of the gradient descent optimization method called backpropagation. The results for both the feedforward implementation and backpropagation training are then compared with their CPU versions.

Keywords: Neural Networks, GPU, CUDA, backpropagation, ECG
1. INTRODUCTION

Motivated by an ever increasing demand for high definition and real-time 3D graphics, GPUs (Graphics Processing Units) have become a platform with huge processing capacity due to their inherent parallelism and broad-band memory. The solutions of certain classes of problems are specially affected by the high parallelism available in a GPU. Some of them are Cryptography, FFT, 2D and 3D segmentation, image processing, computer vision, many problems in the bio-informatics field and Artificial Intelligence techniques like Genetic Algorithms and Artificial Neural Networks (ANN). This work relates the implementation in GPU of a specific, but with broad applications, type of Artificial Neural Network called Feedforward Multilayer Perceptron (FFMLP). (Owens et al., 2007)

1.1 Processing in Graphic Boards

GPU - Graphics Processing Unit A high demand for faster processing of 3D and high definition graphics induced the production of the current programmable GPUs, which have huge processing power and high parallelism. As a matter of fact, the increase in Floating-Point Operations per Second (FPOS) for the GPU is faster than for CPU. Recent numbers show that GPU has a performance of more than seven times GFlops/second than a current CPU (NVIDIA, 2008). The reason for this discrepancy is that GPUs are specialized in highly parallel and computationally intensive processes. This means that the GPU is projected in a way that more transistors are used to process data instead of being used to store data or for flux control. The structural difference between a CPU and a GPU can be seen on Fig. 1.

![Figure 1: Structural differences between a CPU and a GPU](image)

The superiority of GPUs for solving highly parallel problems becomes evident from Fig. 1, which suggests that an arithmetically intensive problem strongly benefits from the GPU’s structure.

CUDA - Compute Unified Device Architecture CUDA (Compute Unified Device Architecture) is a language that extends the C programing language, allowing the programmer to use functions named kernel to call a process a number of times concurrently, being each data processed by an individual thread. A diagram of how the inner structure of a GPU is hierarchically organized can be seen on Fig. 2.

Each thread runs a distinct process or program. Threads are organized inside Blocks, and Blocks are in turn grouped inside Grids.
Calling a process N times  The kernel is called using a global declaration and the number of threads and blocks is defined using the syntax seen on Fig. 3. Each process runs on a thread, and each thread has a number for it’s identification. This number becomes available through the variable threadID from within the kernel. An example can be seen on Fig. 3, which shows the code used to add two vectors.

A common CPU solution for the above mentioned problem would be running through each element of vectors A and B, adding them and storing the result at the same position in C. However, the sum of each position is independent of the other, which suggests that processing all positions at the same time would produce the same result considerably faster. This parallel process is what happens within the GPU when the code from Fig. 3 is processed. However, it is important to remember that threads are organized within blocks, which in turn are organized within grids. The variable blockId returns the number of the block in question while its size can be acquired with blockDim.

Figure 4 for instance shows the sum of two matrices. notice that the position of each thread depends on its position inside its Block (threadIdx) and the position of its Block within the Grid (blockIdx * blockDim). A block size defined as being 16x16 for instance contains 256 threads and this size should be defined taking into account the GPU’s hardware limitations.

1.2 Artificial Neural Networks

Artificial Neural Networks are composed of simple elements that, working together and in parallel in a net, are able to perform approximations, pattern recognition, control of systems and also complex classifications. Each element in this net is called neuron, and its structure is based on how a biological neuron works, like the one from Fig. 5.

In a biological neuron, connections between neurons occur through structures called synapses.
All input connections are summed and, if this value reaches a certain limit, the signal propagates through the axon. (Freeman and Skapura, 1991)

Network structure It is possible to determine a mathematical model that behaves like the original biological neuron, such as the one in Fig. 6. These artificial neurons can then be organized into layers producing a structure that behaves similarly to a biological neural network.

The artificial neuron is nothing more than a weighted sum of its inputs and subsequent application of an activation function, as can be seen on Fig. 6. The weights are adjusted by an iterative process called training. After adding the weighted inputs, the result goes through an activation function, which can be of many kinds; like sigmoid 7(b), or linear 7(a). Another commonly used activation function is the hyperbolic tangent as seen on Fig. 7(c).

From Fig. 8 it becomes clear that it is possible to represent those operations basically as matrices and vectors multiplications. The output of one layer can be written as:

\[ s = f(W \cdot i + b) \]  

(1)

Being \( f \) the activation function, \( i \) the input vector for the current layer, \( b \) the bias vector and \( W \) the weight matrix.

\[
W = \begin{bmatrix}
W_{1,1} & W_{1,2} & \cdots & W_{1,nInputs} \\
W_{2,1} & W_{2,2} & \cdots & W_{2,nInputs} \\
\vdots & \vdots & \ddots & \vdots \\
W_{nNeurons,1} & W_{nNeurons,2} & \cdots & W_{nNeurons,nInputs}
\end{bmatrix}
\]

Bias is an input fixed as 1 and has its own weight. As can be seen on the weight matrix, lines on \( W \) represent weights for a particular neuron, being the number of columns equivalent to the number of neurons and the number of lines equivalent to the number of inputs.
When more than one layer of neurons are linked in series like in Fig. 8 it forms a structure called Feedforward Multilayer Perceptron (FFMLP). In this kind of Artificial Neural Network all outputs of one layer are connected to all inputs of the following layer.

2. METHODS

As previously explained in section 1.2, a neural network can be implemented as a series of matrix multiplications and applications of activation functions. These matrices are called weight matrices and contain the weights that are to be multiplied by the Input vector. These weights are calculated through an iterative process called training which aims at reducing the Output error.

Since the bias is much like an input set as unit, it is possible to treat it like an extra input in the input vector. Doing so allows including the extra sum process, as seen in equation 1, into the vector/matrix multiplication. The result of this is that the weight matrix will go from having a dimension of (nNeurons,nInputs) to (nNeurons,nInputs+1). The program also needs an Input vector, I with an extra element at its end set as 1 to be processed.
Figure 6: Artificial model of a Neuron.

Figure 7: Activation Functions Linear (a), Sigmoid (b) and Hyperbolic Tangent (c).

The current example was performed using the sigmoid activation function for all neurons.

2.1 The feedforward phase

Once we have the weight matrices (with the bias vectors included) and an Input vector the steps to implement the neural network can be described as follows:

1. Multiply the first layer’s weight matrix by the Input vector (with last element set as unit for bias).

2. Apply the activation function.

3. Multiply the output of the previous layer (with last element set as unit) by the next weight matrix.

4. Apply the activation function.

Implementing the previous algorithm in C renders a code that is basically a loop for each neuron that calculates the weighted sum of its inputs, adds its correspondent bias and applies the activation function. This process happens once for each layer and produces the final Output vector.

The costs of looping through each weight-input multiplication would be extremely high for a neural network with many neurons and layers. In applications such as computer vision
and image recognition for instance, the number of neurons can be hundreds and the application
might demand the neural network to perform these calculations many times per second.

Notice however that most calculations are completely independent from one another. Par-
allelization of these operations produces a huge gain in performance.

The same process was then implemented also in GPU. For step 1 the original matrixMul
kernel from cuda SDK was used, the result being processed by another kernel that applies the
activation function as in step 3. The final effect was that each multiplication was done by an
independent thread in the GPU. In the kernel responsible for applying the activation function,
every neuron was treated by an individual thread as well. Achieving then a high level of weight
and node parallelism (Omondi and Rajapakse, 2006).

The tests were then performed at the float level of arithmetic precision for both platforms.

2.2 The backpropagation algorithm

Before using a Neural Network, however, it is vital to train it so it can perform some
classification, function approximation or solve any other problem it was meant to solve. As
previously mentioned, training consists of reducing the overall error of the network by fine-
tuning its many weights. A common algorithm for doing this is a variation of the gradient
descent method known as backpropagation.

Figure 9 considers the specific case of a single neuron with only two weights. For each
possible value they might assume a different result comes out from the output for a given in-
put. This result is initially different from the expected output, resulting in an error, as seen in
equation 2, that can be measured and evaluated.

\[
error_k(n) = expectedOutput_k(n) - producedOutput_k(n) \tag{2}
\]

The overall objective is to minimize the total error of the output vector so the produced
output gets closer to the expected. The usual function to be minimized can be seen in equation
3 (Haykin, 1998), with N being the total number of elements in the output vector.

\[
E(n) = \frac{1}{2} \cdot \sum_{k=1}^{N} error_k(n)^2 \tag{3}
\]

The weights that form the network must be corrected in order to make the error seen in
equation 3 as small as possible. One way of doing it is by taking the gradient of the error
surface, like the one seen in Fig. 9 and correcting all the weights in the opposite direction, which constitutes a *delta descent* optimization technique. The total weight correction can be found by using the equation 4 (Haykin, 1998).

\[
\Delta W_{kj}(n) = \eta \cdot error_k(n) \cdot \frac{df(wSum)}{dwSum} \cdot I_j(n)
\]  

(4)

Being \( \eta \) a positive constant called *learning rate*, \( wSum \) the weighted sum at the given neuron, \( k \) the current neuron, \( j \) the current input and the term \( \frac{df(wSum)}{dwSum} \) the derivative of the activation function.

Since there is no direct access to the output error at a given hidden neuron, we must estimate it by applying equation 5.

\[
error_k(n) = \sum_{l=1}^{L} error_l(n) \cdot W_{kl}(n)
\]  

(5)

With \( k \) as the neuron of the previous layer, \( l \) the neuron of the current layer and \( L \) the total number of neurons in the current layer. In other words, the estimated error of a given neuron at the first layer will be defined by the weighted sum of the errors existent in all the neurons of the next layer that this particular neuron is connected with.

An algorithm that implements this method is as follows:

1. Start weights with random values.
2. Apply the input to the network.
3. Estimate the errors.
4. Calculate $\Delta W$ for all weights.

5. Correct the original weights.

The program should loop through steps 2,3,4,5 until it reaches a satisfactory total error, each loop constituting one epoch, as defined by equation 3. This sort of training is called incremental or online training, and the weights are altered each time a particular input is presented to the neural network. Another way of updating the weights is doing so only at the end of each epoch, accumulating all the changes for each weight and using them only after all the inputs have been presented to the neural network. This mode of training is also called batch training, and would be the same as looping through steps 2,3,4 and leaving step 5 as the last thing to be done in the current epoch.

2.3 Implementation

The algorithm is performed both in the CPU and GPU as a sequence of matrix and vector operations as the following description, being $O$ the expected output and $I$ the input:

1. Feedforward phase
   
   (a) $wSum1 = W1 \cdot I$
   
   (b) $wSum2 = W2 \cdot S1$
   
   (c) $S1 = f(wSum1)$
   
   (d) $S2 = f(wSum2)$

2. Calculate errors
   
   (a) $errorL2 = O - S2$
   
   (b) $errorL1 = W2^T \cdot errorL2$

3. $totalError = totalError + \sum_{i=1}^{N} error^2$

4. Weigth update
   
   (a) $\delta L1 = \eta \cdot errorL1 \cdot f'(wSum1)$
   
   (b) $\delta L2 = \eta \cdot errorL2 \cdot f'(wSum2)$
   
   (c) $\delta W1 = \delta L1 \cdot I^T$
   
   (d) $\delta W2 = \delta L2 \cdot S1^T$
   
   (e) $W1 = W1 + \delta W1$
   
   (f) $W2 = W2 + \delta W2$
There is a difference however in how these operations are performed in the CPU and GPU. While in the CPU version these steps are processed in C functions with serial operations, the GPU version processes each of them in a particular kernel.

As an example, the matrix multiplication at 1a is performed by the \texttt{matrixMul} kernel, while the error calculation 2a is performed by a kernel like the one in picture 3.

While calculating the errors’ propagation on item 2b we were able to avoid transposing the weight matrix \( W_2 \) by performing \( \text{errorL}_1 = W_2^T \cdot \text{errorL}_2 \) instead of \( \text{errorL}_1 = W_2 \cdot \text{errorL}_2 \), which renders the same result since \( \text{errorL}_1 \) is always a vector, and as such is always stored in the same linear way in the memory.

3. BENCHMARK

The two versions of the algorithms were implemented in C for the CPU hardware and in CUDA for the GPU. The CPU configuration was:

- OS: Linux Ubuntu version 9.04: kernel 2.6.28-15
- Memory: 3.8GB RAM
- Processor: Intel Core 2 Dual E8400@3.00GHz

The GPU configuration used for this project was the following:

- GeForce 8800GT
- Memory: 1024Mb
- GPU clock: 600MHz
- Memory clock: 900MHz

In order to test the training performance for a meaningful range of network sizes we have tested our algorithm to solve a particular classification problem which aims at identifying a number of heart diseases in an ECG (electrocardiogram) signal (Hampton, 2003).

The signals were obtained at the PhysioBank (physiologic signal archives for biomedical research) database and were taken from the MIT-BIH Arrhythmia set of signals (Goldberger et al., 13).

The objective is to classify each heart pulse as being either normal, premature ventricular contraction, right bundle branch Block or left bundle branch block (Hamde, 2002).

From each pulse a set of 43 features were extracted using the discrete wavelet transform (Mallat, 1989) and served as input pattern for training the neural network. The training set is composed of 512 samples in total.

The above mentioned problem renders a network with 44 inputs, considering the bias which is automatically inserted into the input vector, and 4 neurons at the output, so each neuron would be responsible for classifying a particular cardiopathy. The size of the hidden layer was set from 8, 16, 32, 64, 128, 256, 512 and 1024 neurons to test the training performance in both CPU and GPU.

As the problem becomes less computationally intensive the advantage of the GPU over the CPU not only reduces but also becomes inferior. In order to test even smaller problems and investigate this phenomenon, we used the standard 2 inputs XOR gate with a variable number of hidden layer inputs and checked at which point using the GPU ceased to be an advantage. All tests were performed with a learning rate of \( \eta = 0.1 \).
3.1 Results

Before using the GPU it is important to perform a "warm-up" computation outside the measured time to remove the CUDA startup overhead from the performance measurements Podlozhnyuk (2007). All results were recorded in files by the program and produced what can be seen in Tab. 3 and 1. The tables show the results for both problems with different number of neurons in the hidden layer which is associated to a total number of weights at each network.

<table>
<thead>
<tr>
<th>Hidden layer</th>
<th>Total Weights</th>
<th>CPU [ms]</th>
<th>CPU epochs</th>
<th>GPU [ms]</th>
<th>GPU epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9</td>
<td>17.35</td>
<td>454</td>
<td>78.80</td>
<td>296</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>27.30</td>
<td>428</td>
<td>69.27</td>
<td>256</td>
</tr>
<tr>
<td>8</td>
<td>33</td>
<td>69.41</td>
<td>605</td>
<td>98.40</td>
<td>351</td>
</tr>
<tr>
<td>16</td>
<td>65</td>
<td>60.87</td>
<td>281</td>
<td>43.22</td>
<td>144</td>
</tr>
<tr>
<td>32</td>
<td>129</td>
<td>159.32</td>
<td>380</td>
<td>43.35</td>
<td>133</td>
</tr>
<tr>
<td>64</td>
<td>257</td>
<td>257.64</td>
<td>312</td>
<td>33.81</td>
<td>83</td>
</tr>
<tr>
<td>128</td>
<td>513</td>
<td>2798.40</td>
<td>1712</td>
<td>49.11</td>
<td>89</td>
</tr>
<tr>
<td>256</td>
<td>1025</td>
<td>709.02</td>
<td>217</td>
<td>59.16</td>
<td>69</td>
</tr>
</tbody>
</table>

While measuring the time for the case when the allocation is considered there was a slight variation in the values, around a few microseconds. In those cases the average of the most significant values was taken. It was noted from the tests that there was no significant difference in performance between taking and not taking into account the memory allocation time.

As expected, a gain in performance was obtained while running the algorithm in GPU. That happened because the training algorithm makes intensive use of matrix operations. Comparing Fig. 11 and Fig. 12 it is possible to notice that gain in performance for the XOR case was more expressible than the ECG one.
Table 2: XOR gate training results counting the memory allocation time

<table>
<thead>
<tr>
<th>Hidden layer</th>
<th>Total Weights</th>
<th>CPU [ms]</th>
<th>CPU epochs</th>
<th>GPU [ms]</th>
<th>GPU epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9</td>
<td>17.35</td>
<td>454</td>
<td>79.50</td>
<td>296</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>27.31</td>
<td>428</td>
<td>69.47</td>
<td>256</td>
</tr>
<tr>
<td>8</td>
<td>33</td>
<td>69.37</td>
<td>605</td>
<td>98.47</td>
<td>351</td>
</tr>
<tr>
<td>16</td>
<td>65</td>
<td>60.86</td>
<td>281</td>
<td>43.07</td>
<td>144</td>
</tr>
<tr>
<td>32</td>
<td>129</td>
<td>159.32</td>
<td>380</td>
<td>44.30</td>
<td>133</td>
</tr>
<tr>
<td>64</td>
<td>257</td>
<td>257.69</td>
<td>312</td>
<td>33.94</td>
<td>83</td>
</tr>
<tr>
<td>128</td>
<td>627</td>
<td>2796.97</td>
<td>1712</td>
<td>48.95</td>
<td>89</td>
</tr>
<tr>
<td>256</td>
<td>1025</td>
<td>709.37</td>
<td>217</td>
<td>59.40</td>
<td>69</td>
</tr>
</tbody>
</table>

Table 3: ECG pulse classification training results without counting the memory allocation time

<table>
<thead>
<tr>
<th>Hidden layer</th>
<th>Total Weights</th>
<th>CPU [ms]</th>
<th>CPU epochs</th>
<th>GPU [ms]</th>
<th>GPU epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>388</td>
<td>2793.15</td>
<td>425</td>
<td>19831.98</td>
<td>425</td>
</tr>
<tr>
<td>16</td>
<td>772</td>
<td>5542.02</td>
<td>495</td>
<td>24629.01</td>
<td>495</td>
</tr>
<tr>
<td>32</td>
<td>1540</td>
<td>17142.92</td>
<td>841</td>
<td>46207.35</td>
<td>841</td>
</tr>
<tr>
<td>64</td>
<td>3076</td>
<td>32726.14</td>
<td>842</td>
<td>24391.39</td>
<td>367</td>
</tr>
<tr>
<td>128</td>
<td>6148</td>
<td>64399.37</td>
<td>843</td>
<td>28623.60</td>
<td>322</td>
</tr>
<tr>
<td>256</td>
<td>12292</td>
<td>126853.82</td>
<td>844</td>
<td>75526.91</td>
<td>532</td>
</tr>
<tr>
<td>512</td>
<td>24580</td>
<td>1792943.87</td>
<td>5996</td>
<td>1542597.37</td>
<td>5995</td>
</tr>
<tr>
<td>1024</td>
<td>49156</td>
<td>3002.38</td>
<td>4</td>
<td>2372.53</td>
<td>4</td>
</tr>
<tr>
<td>2048</td>
<td>98308</td>
<td>7232.48</td>
<td>5</td>
<td>16356.61</td>
<td>17</td>
</tr>
<tr>
<td>4096</td>
<td>196612</td>
<td>7553.40</td>
<td>2</td>
<td>10735.38</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 11: Comparison of the performance between CPU and GPU for the XOR problem

That happened because the ECG classification problem requires a considerable number of samples (512 samples in this particular case) compared to the XOR problem (only 4 samples). Considering this, a batch version of this algorithm would produce a performance gain even more expressive.
4. CONCLUSIONS

The implementation of Neural Networks in GPU has show a considerable improvement in processing speed compared to its implementation in CPU. Real time applications such as vision systems and pattern recognition would greatly benefit from such an impressive performance.

Highly parallel applications such as neural networks are specially benefited by GPU’s structure not only for processing time but also during its time-demanding training phase.

It is important to realise however that for the use of GPU to be justified, the problem must be sufficiently computationally intensive to compensate for the memory allocation and data transfer costs inherent to using the graphic card instead of processing it all in the CPU.
Currently the batch training is being implemented in both CPU and GPU versions of the algorithm what would considerably improve the performance in GPU since weight updates could be made in parallel. The gradient descent algorithm is also being improved by adding an adaptive learning rate and a momentum factor, increasing the training speed and helping to avoid falling in local minima respectively.

In the near future other algorithms such as the conjugate gradient will be implemented and added to form a GPU neural network tool box.

Acknowledgements

We would like to thank the team at the Mechanical Computation Department for the support and also the Mechanical Engineering Faculty of the State University of Campinas for the opportunity to develop this work. We would also like to thank CAPES for the financial support.

REFERENCES


