Implementation of a Multilayer Perceptron in the Graphics Processing Unit

Ventsislav Nikolov

Abstract. *In this paper, a parallel realization of a multilayer neural network in a Graphics Processing Unit (GPU) is presented. There are different types and architectures of neural networks, but most of them are known as models that strongly incorporate fine-grained parallelism. This is highly applicable to GPUs, which contain a large number of simple Streaming Multiprocessors. The realization is based on Compute Unified Device Architecture which is an extension to the C language that allows GPU code to be written in regular C. Performances are represented according to the time required for the execution of sequential and parallel implementations. Conclusions and future developments are introduced at the end of the paper.*

Keywords: *Parallel Neural Network, Graphics Processing Unit, Compute Unified Device Architecture*

1 INTRODUCTION

Often in computer technologies methods are used based on exact calculations. For example, in searching algorithms the goal is to find a specific element in a given set. Searching for a record in a database is performed by looking for an exact value within a given field, such as a record identifier or a group of fields.

Neural networks, unlike such exact solutions, work with approximations [2]. This makes them convenient for solving problems related to inexact or partial data. There are situations in which it is difficult to find a solution, except by using principles of inexact solutions. Those situations often arise in tasks in which finding a solution – even if it is not an exact one – is more important than the absolute accuracy. Inexact solutions are convenient for finding similar, pre-processed or raw data. Neural networks are able to generalize, which is one of their most important characteristics [3]. Their development has been inspired by biological neural networks and, as such, they represent a very simplified equivalent of natural neural networks [7].

Neural networks can be realized both in hardware and software. The hardware realization is often more effective, as it is designed to solve specific problems. The flexibility of the hardware realization, however, is very poor. Oftentimes, settings for neural network parameters depend on the problem that has to be solved, where the software realization provides significant advantages. Despite it being performed in a general-purpose machine, the software realization is usually much cheaper and significantly easier to modify, even if it is not as fast as the hardware realization. This is the reason why, in this paper, the focus lies on the theoretical basis of the software realization on a general-purpose Graphics Processing Unit (GPU).

2 GPU VERSUS CPU

The type of neural network considered in this work is a multilayer perceptron [4], which is one of the most widely used neural networks in practice. There are different types and architectures of neural networks [5] [8], but most of them are known as models that strongly incorporate fine-grained parallelism [10]. This is appropriate for the GPU, which frequently incorporates a large number of simple Streaming Multiprocessors (SMs). The realization presented here is based on Compute Unified Device Architecture (CUDA) [1] – an extension to the C language that allows the GPU code to be written in regular C. The written code can be executed in both host Central Processing Unit (CPU) and device processor unit (GPU). In Fig. 1, the performances of CPU and GPU are juxtaposed according to the CUDA programming developers guide, dating from a number of years ago [1].

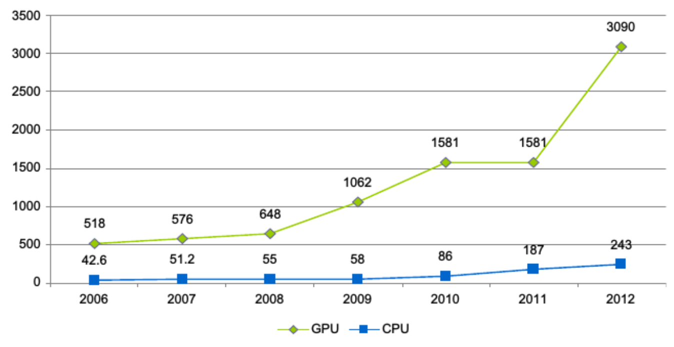


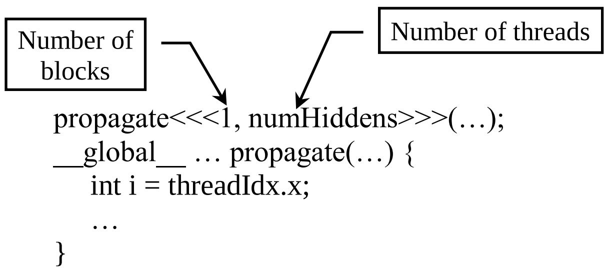
Fig. 1. CPU versus GPU performance in gigaflops

This performance has been measured in 2012, but is quite similar now. Moreover, the difference nowadays is even greater in favour of the GPU.
When it comes to executing a program code in GPU, the main problem is that the algorithms must be adapted to become attuned with fine-grained parallelism. Here, an approach is introduced that is applied for parallel execution in both forward and backward stages on a multilayer perceptron. Independent processing units in each layer work in parallel. In the forward stage, parallel calculations are realized in the hidden and output layers, while in the backward stage, they take place in the hidden and input layer. The proposed approach is described for neural networks with one hidden layer, but is applicable for any number of hidden layers as well. Experimental data, that has been used for the training, is presented in the form of financial data of individual persons. The goal is to determine each individual’s credit rating based on historical examples. The performance of the realization is demonstrated according to the time required for the execution of sequential and parallel implementations.

3 SPECIFIC FEATURES OF CUDA PROGRAMMING

In CUDA programming, entry points are provided to the GPU by C functions called *kernels*. Syntactically, they are invoked as normal functions with two differences:

- Memory management between CPU and GPU. Memory regions, represented in CPU, must be copied in order to be available in GPU prior to invoking the kernels, as well as after that, in the opposite direction, to obtain the results. Thereafter, the allocated GPU memory must be freed.
- By calling the kernel, the number of grids, blocks and threads must be specified. These are 3D dimensional arrays that are physically used for the execution of kernel functions. The kernel function is executed once for every thread, so a specified number of threads determines the number of executions. In kernel functions, appropriate program constructs must be used for the execution of independent code fragments in parallel. Indices of blocks and threads are available by built-in variables, provided by CUDA. The picture below shows an example of calling a kernel function, as well as the kernel function prototype.



```
propagate<<<1, numHiddens>>>(…);
__global__ … propagate(…) {
    int i = threadIdx.x;
    …
}
```

Basically, CUDA hardware and software are organized in devices, grids, blocks and threads, as shown in Fig.2. In kernel entry points, these elements are specified by describing the level of parallelism.

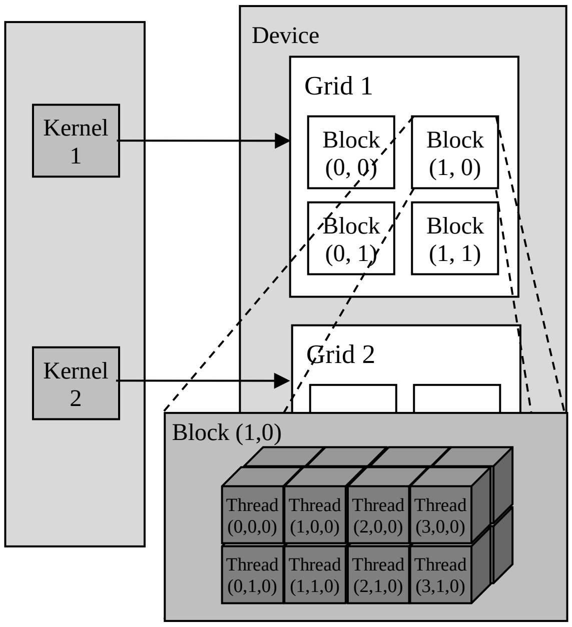


Fig. 2. CUDA device, block and grid organization

4 THE NEURAL NETWORK ALGORITHM

The realized neural network works in two stages: the training that is based on historical data and the generation of output data from new and unknown input data.

4.1 Training

In the training stage, available data is presented in the form of pairs of input-output training vectors $z_1 \rightarrow d_1$, $z_2 \rightarrow d_2, \dots$ $z_p \rightarrow d_p$. Here, input vectors are denoted as z and output vectors as d. The number of input neurons is I, hidden neurons are J and outputs are K. Symbol z_i stands for generated values of input neurons, y_j for hidden neurons and o_k for outputs. Index i is used for input neurons, j for hidden neurons and k for output neurons. Each neuron, from any given layer, is connected to every other neuron in the next layer. For these connections, there are associated weights. Those between input and hidden layer are v_{ji}, and the ones between hidden and output layer are w_{kj}.

I and K are determined according to the number of input and output values in the provided training examples, while J is determined by different approaches [9]. For example, cross-validation can be used by comparing the output error and by choosing the best hidden neuron numbers. In the here presented calculation, J is calculated by using (1) and is rounded to the nearest integer value.

$$J = \sqrt{I * K} \tag{1}$$

Output values, generated from input elements, are equal to their input values z_1, z_2, … z_I, but for other layers, different types of activation functions are used [9]. Thus, data should be transformed in the corresponding definition domain, ranging from min to max, according to the activation function. Some of the most commonly used activation functions are shown below [4].

Bipolar sigmoid:

$$f(x) = \frac{1-e^{-x}}{1+e^{-x}} \tag{2}$$

Hyperbolic tangent:

$$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{3}$$

Softplus:

$$f(x) = \ln(1 + e^{x}) \tag{4}$$

Gaussian:

$$f(x) = e^{-x^2} \tag{5}$$

Bent identity:

$$f(x) = \frac{\sqrt{x^2+1}-1}{2} + x \tag{6}$$

In neural network layers and processing neurons the activation function can vary, but most often, only one type is chosen in the training. The main principle for choosing a function type is that is must be non-linear and its derivative have to be easily computed.
The neural network is trained until the criterion for the completion of the training is met. For example, reaching the given number of epochs or a maximum error for all training patterns [6]. In every epoch, all training patterns are presented to the neural network in random or spatial order and the weights of connections between neurons are modified. In this way, they are iteratively improving the generated outputs to be as close as possible to the given outputs. For every training pattern, the following calculations are performed:

FORWARD STAGE

Input values are calculated for the neurons in the hidden layer:

$$net_j = \sum_{i=1}^{A}\left(x_i v_{ji}\right) \tag{7}$$

Output values are calculated for the hidden neurons:

$$h_j = f(net_j) \tag{8}$$

where f is the activation function.

The propagation from hidden to output layer is done in a similar way. First, the input sum is calculated:

$$net_k = \sum_{j=1}^{B}\left(h_j w_{kj}\right) \tag{9}$$

after that, the outputs are calculated:

$$o_k = f(net_k) \tag{10}$$

where f is one of the functions (2)-(6).

BACKWARD STAGE

Errors are calculated for the output elements:

$$\varepsilon = \frac{1}{2}(d_k - o_k)^2 \tag{11}$$

In order to calculate the modification of weights of the connections between hidden and output neurons, the following equation is used:

$$\Delta w_{kj} = -\eta \frac{\partial \varepsilon}{\partial w_{kj}} \tag{12}$$

where η is the learning rate chosen before the training. Normally, it has a small real value that determines the convergence rate. The term $\frac{\partial \varepsilon}{\partial w_{kj}}$ is calculated as follows:

$$\frac{\partial \varepsilon}{\partial w_{kj}} = \frac{\partial \varepsilon}{\partial o_k}\frac{\partial o_k}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} \tag{13}$$

Taking into account that

$$\frac{\partial \varepsilon}{\partial o_k} = -(d_k - o_k) \tag{14}$$

$$\frac{\partial o_k}{\partial net_k} = f'(net_k) \tag{15}$$

$$\frac{\partial net_k}{\partial w_{kj}} = h_j \tag{16}$$

then

$$\frac{\partial \varepsilon}{\partial w_{kj}} = \frac{\partial \varepsilon}{\partial o_k}\frac{\partial o_k}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = -(d_k - o_k)f'(net_k)h_j \tag{17}$$

and

$$\Delta w_{kj} = \eta(d_k - o_k)f'(net_k)h_j \tag{18}$$

The derivative f'(net$_k$) of the activation function is calculated according to the weighs. Functions (2)-(6) have the following derivatives:

Bipolar sigmoid:

$$f'(x) = f(x)(1 - f(x)) \tag{19}$$

Hyperbolic tangent:

$$f'(x) = 1 - f(x)^2 \tag{20}$$

Softplus:

$$f'(x) = \frac{1}{1+e^{-x}} \tag{21}$$

Gaussian:

$$f'(x) = -2xe^{-x^2} \tag{22}$$

Bent identity:

$$f'(x) = \frac{x}{2\sqrt{x^2+1}} + 1 \tag{23}$$

By taking the following substitution

$$\delta_k = \frac{\partial \varepsilon}{\partial o_k}\frac{\partial o_k}{\partial net_k} \tag{24}$$

the modification of weights of the connection between input and hidden layer neurons can be calculated as:

$$\Delta v_{ji} = -\eta \frac{\partial \varepsilon}{\partial v_{ji}} = \eta\left(\sum_{k=1}^{C}\left(w_{kj}\delta_k\right)\right)f'\left(net_j\right)z_i \tag{25}$$

After applying the modifications Δv_{ji} and Δw_{kj}

$$w_{kj}(\text{new}) = w_{kj}(\text{old}) + \Delta w_{kj} \tag{26}$$

$$v_{ji}(\text{new}) = v_{ji}(\text{old}) + \Delta v_{ji} \tag{27}$$

all previous steps, starting from (7), are repeated for all training patterns. If the criterion for stopping the training is still not met, forward and backward stages are performed again.
Modifications of the main algorithm, that are described above, can be applied by using a momentum constant, in order to avoid being stuck in a local minimum of the error surface ε(w).

$$\Delta w_{kj}(\text{new}) = \Delta w_{kj} + \mu\Delta w_{kj}(old) \tag{28}$$

$$\Delta v_{ji}(\text{new}) = \Delta v_{ji} + \mu\Delta v_{ji}(old) \tag{29}$$

Additionally, a flat spot term can be added in (12) and (26), in order to avoid flat spaces in the error surface function ε(w).

$$\Delta w_{kj} = -\eta\left(\frac{\partial \varepsilon}{\partial w_{kj}} + c\right) \tag{30}$$

$$\Delta v_{ji} = -\eta\left(\frac{\partial \varepsilon}{\partial v_{ji}} + c\right) \tag{31}$$

4.2 Parallel Algorithm

The parallel execution is performed in both forward and backward stages. In the forward stage, independent calculations are carried out for neurons j in (7) and (8) for the hidden layer.

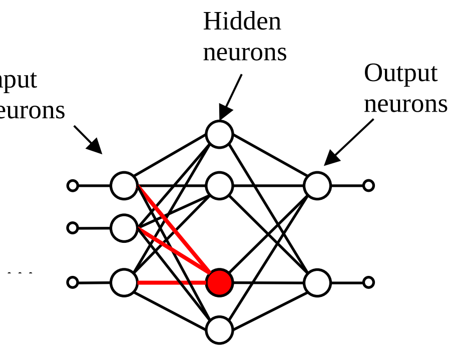


Fig. 3. Independent neuron in the hidden layer in the forward stage

In the same way, calculations for neurons k in (9) and (10) are performed on separate SM (Fig.2).

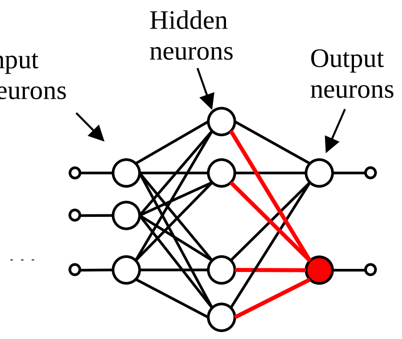


Fig. 4. Independent neuron in the output layer in the forward stage

In the backward stage, calculations in neurons j, in hidden layers, are independent in (19). Neurons i in input layer (26) are also performed in parallel.

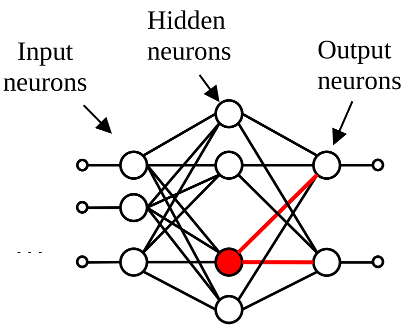


Fig. 5. Independent neuron in the hidden layer in the backward stage

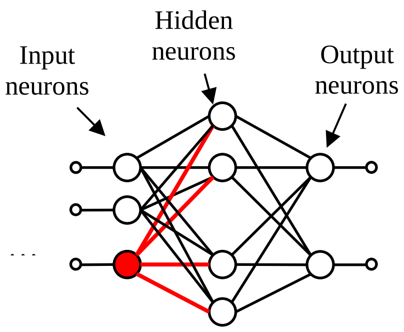


Fig. 6. Independent neuron in the input layer in the backward stage

5 RESULTS

The results presented in this paper are obtained from 100 training examples. Every training example consists of 22 real values. The training is performed in 500 epochs, with a neural network architecture of one hidden layer, 22 input neurons, 30 hidden neurons and one output node. The experiments are repeated 20 times and the average time for the sequential realization in the GPU is 25.89 seconds; the parallel GPU realization lasts 3.73 seconds. If the parallelization is performed only in the forward stage – as it is shown in Fig. 1 and Fig. 2 – the execution time is 18.55 seconds.

6 CONCLUSIONS AND FUTURE WORK

The parallel realization depends on the architecture and the algorithm of the neural network. Other approaches also need to be realized and tested, such as the local approach, according to which training patterns are separated in sub-groups. For every sub-group, a separate neural network is trained, that can be executed in parallel. Additionally, it is possible to develop hierarchical structures with independent sub-structures. As the results show, the investigated approach is promising. If the majority of processing elements can be found in hidden and output layers, the approach is particularly applicable in practical software solutions.

7 REFERENCES

1. Cheng, J., M. Grossman, T. McKercher. Professional CUDA Programming. Wrox, 2014.
2. Du, K.-L., M.N.S. Swamy. Neural Networks in a Softcomputing Framework. Springer, 2006.
3. Fausett, L. Fundamentals of Neural Networks: Architectures, Algorithms, and Applications. Prentice-Hall, ISBN:0-13-334186-0, 1994.
4. Galushkin, A. I. Neural Networks Theory. ISBN 978-3-540-48124-9. Springer, 2007.
5. Hech-Nielsen, R. Theory of the Backpropagation Neural Network. Neural networks for perception (Vol. 2): computation, learning, architectures. Hercourt Brace & Co., ISBN: 0-12-741252-2, 1992, pp. 65-93.
6. J. Zurada, Introduction to artificial neural systems, West Publishing Co., St. Paul, MN, 1992
7. Kasabov, N. K. Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering. The MIT Press, ISBN-10: 0-262-11212-4 ISBN-13: 978-0-262-11212-3, 1998.

8. Kermanshahi, B. Recurrent neural network for forecasting next 10 years loads of nine Japanese utilities. Neurocomputing, Vol. 23, No. 1, 1998, pp.125-133
9. Tarassenko, L. A Guide to Neural Computing Applications. Elsevier, 2004.
10. Touretzky, D. S. 15-486/782: Artificial Neural Networks, Lectures, Carnegie Mellon Univeristy, Fall 2006 - http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15782-f06/syllabus.html.

8 ABOUT THE AUTHOR

Dr. Ventsislav Nikolov
Senior Software Developer
Eurorisk Systems Ltd.
31, General Kiselov Str., 9002 Varna, Bulgaria
E-mail: vnikolov at euroriskssystems dot com