VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Hardwarově akcelerované neuronové sítě

Hardware accelerated neural networks

2011 Bc. Stanislav Toman

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava*.

V Ostravě 28. dubna 2011 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2011 .....

I would like to thank my supervisor Ing. Dušan Fedorčák for supporting me with advices and leading the process of creation this thesis. Without his help, this thesis wouldn't have been possible.

Abstrakt

Tato diplomová práce ukazuje možnosti hardvérové akcelerace algoritmů neuronových síti. V první teoretické části představuje základní prvky neuronových síti, váhy, aktivační funkci a trénovací proces. Vícevrství perceptron, používán pro rozpoznávaní vzoru je pak popsán detailně z pohledu různých trénovacích algoritmů. Detailně je také popsána metoda zpětného šířeni chyby. Druha část práce popisuje možnosti paralelizace metody zpětného šířeni chyby, jsou představeny dva implementační přístupy, implementace na CPU a implementace na GPU. Praktická část pak detailně popisuje zvolenou implementaci na GPU využívající technologii CUDA a ukazuje výsledky výkonnostních testů algoritmu na výkonném CUDA kompatibilním zařízeni. Práce obsahuje i stručný popis momentálně dostupných softwarových řešeni pro neuronové sítě.

Klíčová slova: Vícevrství perceptron, rozpoznávaní vzoru, zpětné šíření chyby, paralelizace, GPU, CUDA

Abstract

This thesis shows the possibilities of hardware acceleration for neural network algorithms. First theoretical part introduces the basics of neural network units, weights, the activation function and the training process. Multi-layer perceptron used in pattern recognition is then deeply analyzed from the view of training algorithms. The error back-propagation process is described in details. Second part of the thesis discusses the options in parallelization of the error back-propagation process, where two implementation approaches, CPU implementation and GPU implementation, are proposed. The practical part then, includes detailed description of chosen GPU implementation utilizing CUDA and shows the results of performance testing on a high end CUDA device. Work also includes brief description of currently available software solutions for neural networks.

Keywords: Multi-layer perceptron, pattern recognition, error back-propagation, parallelization, GPU, CUDA

Seznam použitých zkratek a symbolů

ANN	–	Artificial Neural Network
API	–	Application Programming Interface
CPU	–	Central Processing Unit
CUDA	–	Compute Unified Device Architecture
GPU	–	Graphics Processing unit
MSE	–	Mean Squared Error
RMS	–	Root Mean Square

Contents

1 Introduction

Artificial neural network is a group of interconnected artificial neurons. The basic aim of ANN was to create a model simulating the activity of human brain. After nearly 50 years of research, neural networks are very popular and useful in many fields, as pattern recognition, optimization or prediction. However, we are still not able to recreate fully operating model comparable to the human brain. Due to an amount of neural cells in human brain and the speed they are communicating, there are still obstacles that have to be overcome.

The problem, that we will be discussing in this thesis, is the amount of time needed for the training process of the neural networks. It is known that specially in cases of pattern recognition where the goal is to create a model capable of seeing and classifying graphical patterns in their natural state, that is recognized by human brain, we are presenting a relatively big amount of training input data to the network. As the training is working as an iterative process of refining the structure of neural network, pattern by pattern, it is significantly performance demanding. Thus we will take a deeper look at the possibility of accelerating the training process with available hardware equipment.

As the training needs a lot of simple iterative steps, we will try to parallelize the algorithm and create a robust process that will produce a trained neural network in a competitive time. On the hardware side of the problem, we need to find the best fit for the massively parallel algorithm. For this task, we will use the parallel computing power of GPU. Because the GPU was designed with the aim to do a lot of computing steps in parallel, especially for working with a lot of matrix type data structures.

Another thing is recently published architecture for creating massively parallel algorithms that are computed on a GPU with the acronym CUDA. The CUDA (Compute Unified Device Architecture) is created as an extension to C programing language, so we will be using C/c++ for the implementation. As the computation power of GPU is directly connected with the amount of cores included in the process, we will use a currently available Tesla C2050 hardware platform.

It is believed that with this computational power, we will be able to produce a massively accelerated training algorithm that will be capable of training the neural network structures in competitive time, and this way making the neural network algorithms more usable even in the commercial field.

2 Theoretical introduction to artificial neural networks

Artificial neural network consists of a pool of simple processing units, which communicate by sending signals to each other over a large number of weighted connections. We can distinguish a set of major aspects of this model:

- Neuron - a single processing unit
- Y_k - state of activation for every unit, equivalent to the output of the unit
- W_{jk} - connection between the units $_j$ and $_k$, which has a weight determining the effect which the signal of unit $_j$ has on unit $_k$
- F_k - determines the new level of activation based on the effective input and current activation (the update)
- θ_k - threshold, bias, offset, an external input defined for each unit
- propagation rule - determines the effective input of a unit from its external inputs
- learning rule - a method for information gathering
- environment - space within which the system must operate, providing input signals and error signals

2.1 Neurons

Each unit performs a relatively simple job. Receive input from neighbors or external sources and use this to compute an output signal which is propagated to other units. Apart from this processing, a second task is the adjustment of the weights. The system is inherently parallel in the sense that many units can carry out their computations at the same time. Within neural networks, it is useful to distinguish three types of units.

- input units (indicated by $_i$) which receive data from outside the neural network
- output units (indicated by $_o$) which send data out of the neural network
- hidden units (indicated by $_h$) whose input and output signals remain within the neural network

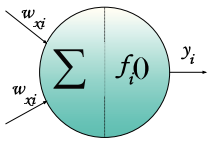

Figure 1: Neuron unit

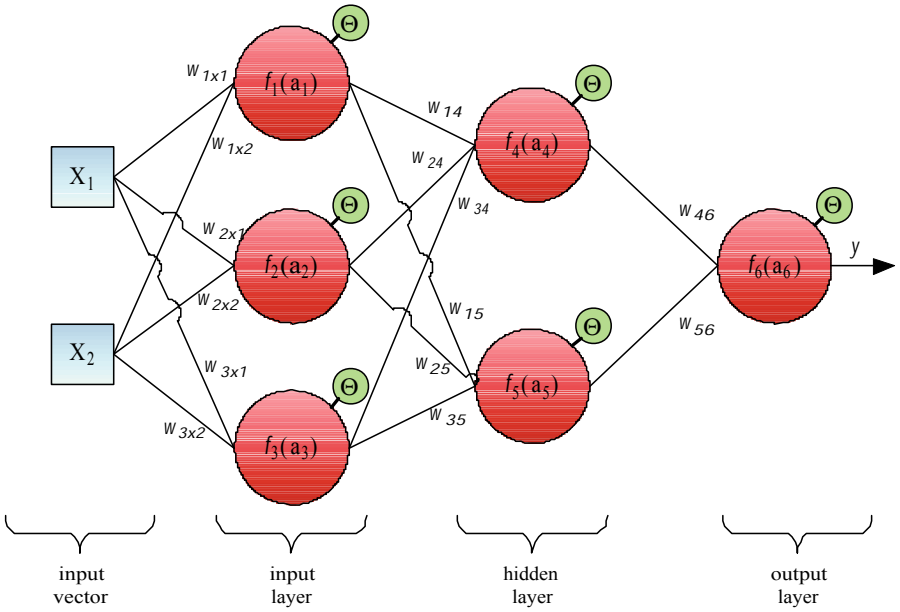

Figure 2: Feed-Forward network topology

During operation, units can be updated either synchronously or asynchronously. With synchronous updating, all units update their activation simultaneously. With asynchronous updating, each unit has a (usually indexed) probability of updating its activation at a time $_t$, and usually only one unit will be able to do this at a time. [**?**, **?**, **?**]

2.2 Weights

In most cases, we assume that each unit provides an additive contribution to the input of the unit with which it is connected. The total input to the unit $_k$ is simply the weighted sum of the separate outputs from each of the connected units plus a bias or offset.

$$\sum x_j w_{jk} + \theta_k \tag{1}$$

The contribution for positive w is considered as an excitation and for negative w as inhibition. In some cases more complex rules for combining inputs are used, in which a distinction is made between excitatory and inhibitory inputs. We call units with a propagation rule sigma units.

2.3 Activation and output rules

We also need a rule which gives the effect of the total input on the activation of the unit. We need a function F which takes the total input $\sum wx$ and the current activation a_t and produces a y_k new value of the activation of the unit $_k$.

Sigmoid function is the most widely used type of activation function.

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

$$f'(x) = f(x)(1 - f(x)) \tag{3}$$

2.4 Network topologies

In the previous section, we discussed the properties of the basic processing unit in an artificial neural network. This section focuses on the pattern of connections between the units and the propagation of data.

As for this pattern of connections, the main distinction we can make is between feed-forward and recurrent networks.

Feed-forward networks, where the data flow from input to output units is strictly feed-forward. The data processing can extend over multiple layers of units, but no feed-back connections are present. That is, connections extending from outputs of units to inputs of units in the same layer or previous layers. Recurrent networks do contain feed-back connections. Contrary to feed-forward networks, the dynamical properties of the network are important. In some cases, the activation values of the units undergo a relaxation process such that the network will evolve to a stable state in which these activations do not change anymore. In other applications, the change of the activation values of the output neurons is significant, such that the dynamical behavior constitutes the output of the network. Classical examples of feed-forward networks are the perceptron, multi-layer perceptron and adaline network. In further discussion we will only consider feed-forward networks. [**?**, **?**]

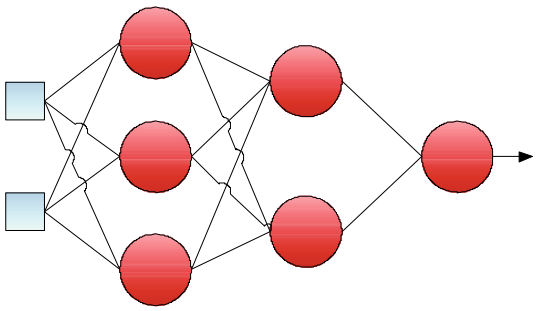

Figure 3: Feed-Forward network topology

2.5 Training of artificial neural networks

A neural network has to be configured such that the application of a set of inputs produces (either direct or via a relaxation process) the desired set of outputs. Various methods to set the strengths of the connections exist. One way is to set the weights explicitly. Another way is to train the neural network by feeding it teaching patterns and letting it change its weights according to some learning rule.

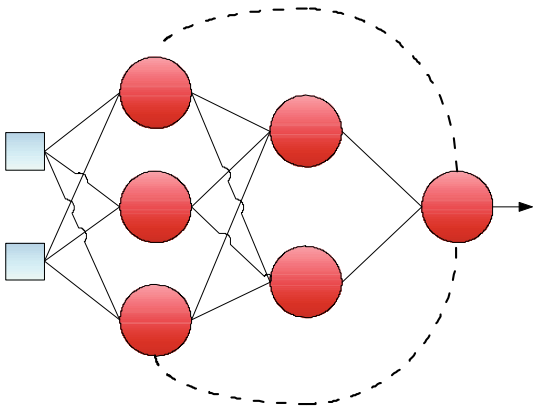

Figure 4: Recurrent network topology

2.5.1 Paradigms of learning

We can categorize the learning situations in two distinct sorts. These are:

- Supervised learning or associative learning in which the network is trained by providing it with input and matching output patterns. These input-output pairs can be provided by an external teacher, or by the system which contains the network (self-supervised).
- Unsupervised learning or self-organization in which an output unit is trained to respond to clusters of pattern within the input. In this paradigm, the system is supposed to discover statistically salient features of input population. Unlike the supervised learning paradigm, there is no a priori set of categories into which the patterns are to be classified, rather the system must develop its own representation of the input stimuli. [**?**]

2.5.2 Modifying unit connections

Both learning paradigms discussed above result in an adjustment of the weights of the connections between units, according to modification rule. Virtually, all learning rules for models of this type can be considered as a variant to Hebbian learning rule suggested by Hebb. The basic idea is that if two units j and k are active simultaneously, their interconnection must be strengthened. If j receives input from k, the simplest version of Hebbian learning describes, how to modify the weight w_{jk} with

$$\Delta w_{jk} = \eta y_j y_k \tag{4}$$

where Y is a positive constant of proportionality representing the learning rate. Another common rule uses not the actual activation of unit k but the difference between the actual and desired activation for adjusting the weights:

$$\Delta w_{jk} =_j y(d_k - y_k) \tag{5}$$

in which d_k is the desired activation provided by a teacher. This is often called the Widrow-Hoff rule or the delta rule.

2.5.3 Incremental algorithm

When talking about processing the training data from the dataset, we can distinguish to the incremental algorithm and the batch algorithm. The idea of incremental approach is that we are processing only one pattern in an epoch. Each pattern then changes the structure of a weight matrix. We can describe the process steps:

1. Read one pattern input vector.
2. Calculate output for current pattern.
3. Update weights.

This describes one epoch of incremental algorithm. In the learning process this algorithm is cycled times the patterns in dataset.

2.5.4 Batch algorithm

Other option is to use the batch algorithm. The difference is that each step of the algorithm is processed in a loop for all patterns in the dataset as follows:

1. Read input vectors for all patterns.
2. Calculate output for all patterns.
3. Update weights.

This describes one epoch of batch learning process. In practice is this approach used quite often. The disadvantage is that it is more memory demanding than the incremental algorithm.

3 Analysis of learning algorithms in neural networks

Many of the fundamental concepts of pattern recognition can be displayed on a simple, yet difficult, problem of recognizing hand written characters of digits. Images are captured by a camera with good resolution or with a scanner. This way we have an dataset containing our patterns. In order to use them as an input to the neural network we need to find a suitable mapping from a picture to a float or better double precision number. For this purpose we must extract each pattern, means a digit or a letter, from the document to an equally sized pixel map. Every pixel in the map has its double precision value of black typically ranging from 0 to 1 according to the fraction of pixel occupied by black ink. Number of pixels in the capture map represents a dimension of the pattern. In other words, a handwritten digit captured on a square map of 20x20 pixel would be converted into a double precision vector with length 400. To feed such a pattern to the structure of neural network, we will need 400 neurons in the input layer, if any preprocessing of the inputs will not be done.

3.1 Multi-layer perceptron

It is known that a single layer neural network can only adapt to a linear separable datasets. This means that the capability of a single layer neural network is limited. For this reason, we introduce multi-layer networks. Multi-layer neural network, also known as multi-layer perceptrons, allow us to apply such a structure to a more complex functions. Although, it is proven that a two layer neural network can approximate any dataset, we can experience topologies considering three even four layers. [**?**] It is due to the search of the optimum between a number of neurons in each layer and a number of layers, which can have an effect on the time demand during the learning process. Basic type of a multi-layer network is a feed-forward network. In this network the output can be calculated as an explicit function of inputs and weights. With the feedback loops, there is a recurrent network which unit's outputs are looped. For networks with differentiable activation functions there, is a powerful and efficient method called error backpropagation for finding derivatives of error in respect to weights in the network. This is an important feature of the multi-layer perceptron which plays a role in the majority of training algorithms. [**?**, **?**]

3.1.1 Error back-propagation

During the learning process of neural network, we are trying to minimize the error in respect to the weights and biases. If we consider first the network of threshold units, we can spot limitations in calculation of the error. Although we can use the perceptron learning rule for the update of weights between the last hidden layer and the output layer, we don't have an algorithm to calculate the errors for earlier layers. If the output layer produces an incorrect output we have no procedure to determine which unit of hidden layers is responsible for the error. So there is no way of determining which weight should be updated. This problem is known as a credit assignment problem. However, we can

find a solution to this problem quit easily. By using a differentiable activation functions. When considering a neural network with differentiable activation functions, the units become a differentiable to the input variables, weights and biases. So if we define an error function such as sum-of-squares or root-mean-square error, which definitely is a differentiable function, then this function is differentiable to the weights. So we are able to find derivatives of this function for the weights, and we can determine which weight is the contributor to the error and how much. This way we can find the values that minimize the error applying the gradient descent, or other optimization method. This method is called error back-propagation. [**?**]

The first step of the algorithm is the forward propagation of input to calculate the output of network for current pattern. Output of each unit is calculated as a function of the sum of input weights. Output of previous layer is presented as an input to the next layer. [**?**]

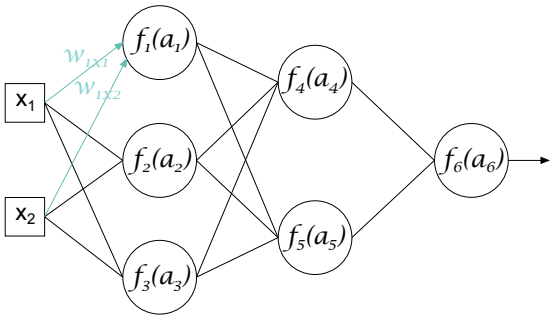

Figure 5: Step 1 : Forward propagation

$$y_1 = f_1\left(w_{1x1}x_1 + w_{1x2}x_2\right) \tag{6}$$

Second step is the calculation of error at output layer. We do this by comparing the actual output y and the desired output d. After calculation, we back-propagate the error to the network. We obtain the δ values for all neurons. [**?**]

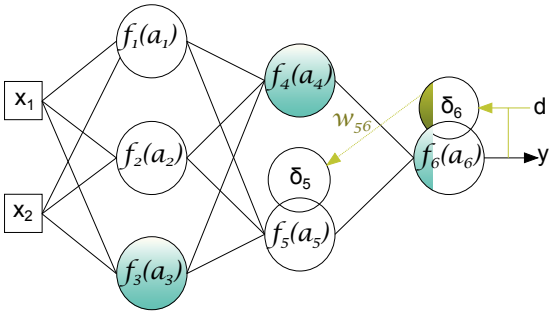

Figure 6: Step 2 : Error back propagation

$$\delta_5 = w_{56}\delta_6 \tag{7}$$

Third step, after back-propagating to all units, we calculate the partial derivative of the activation function of a unit and multiply it with the error value δ, input value x and learning rate η. This multiplication, known as a local gradient, is added to the current weight value. Therefore, the weight is updated. We loop this process for all the units until the output layer.[**?**]

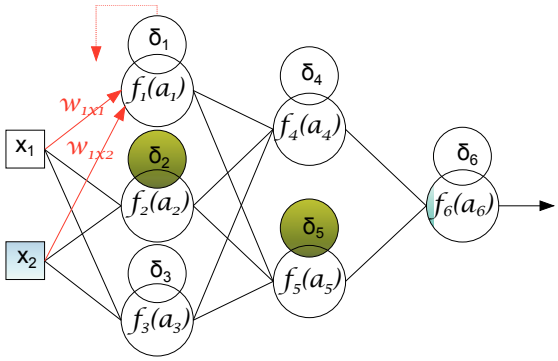

Figure 7: Step 3 : Weights update with local gradient

$$w'_{1x1} = w_{1x1} + \eta\delta_1 \frac{\partial f_1(a_1)}{\partial a_1} x_1 \tag{8}$$

$$w'_{2x1} = w_{2x1} + \eta\delta_1 \frac{\partial f_1(a_1)}{\partial a_1} x_2 \tag{9}$$

The most important aspect of the back-propagation algorithm is its computational efficiency. Let's think about how many computer operations are needed to evaluate the derivatives of the error function in scale of the size of neural network. If W is the total number of weights and biases, then a full calculation of all derivatives would require $\sigma(W)$ operations. Each unit needs one addition operation, for computing the sum and one multiplication for computing the output of activation function. If we took the expression for the error function and wrote down the explicit formula for the derivatives and then evaluated them numerically by forward propagation, we would have to evaluate W number of terms (one for each weight and bias) each requiring $\sigma(W)$ operations. So the overall computational effort would be quadratic $\sigma(W^2)$. In comparison the computational complexity of backpropagation is (W). So the backpropagation algorithm reduced the computational complexity from $\sigma(W^2)$ to $\sigma(W)$ for each input vector. Since we know that the training of a multi-layer perceptron can be time consuming, this efficiency gain is very important. It is also understood that for total computation of the error function of N training patterns the number of computational steps would be N times larger. [**?**]

3.1.2 Jacobian matrix

We have seen how the back-propagation algorithm is used for calculating the derivatives or the error function. However, this method can be used for calculating other derivatives.

Here comes the Jacobian matrix. The elements of Jacobian matrix are the derivatives of the network outputs in respect to the inputs.

$$J(f) = \begin{bmatrix} \frac{\partial f(x_1,w)}{\partial w_1} & \cdots & \frac{\partial f(x_1,w)}{\partial w_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(x_n,w)}{\partial w_1} & \cdots & \frac{\partial f(x_n,w)}{\partial w_n} \end{bmatrix} \tag{10}$$

3.1.3 Hessian matrix

Second derivatives of the function form the Hessian matrix.

$$H(f) = \begin{bmatrix} \frac{\partial^2 f(x_1,w)}{\partial w_1^2} & \frac{\partial^2 f(x_2,w)}{\partial w_1\,\partial w_2} & \cdots & \frac{\partial^2 f(x_n,w)}{\partial w_1\,\partial w_n} \\ \frac{\partial^2 f(x_1,w)}{\partial w_2\,\partial w_1} & \frac{\partial^2 f(x_2,w)}{\partial w_2^2} & \cdots & \frac{\partial^2 f(x_n,w)}{\partial w_2\,\partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x_1,w)}{\partial w_n\,\partial w_1} & \frac{\partial^2 f(x_2,w)}{\partial w_n\,\partial w_2} & \cdots & \frac{\partial^2 f(x_n,w)}{\partial w_n^2} \end{bmatrix} \tag{11}$$

Hessian matrix is used in many applications dependent to the neural networks. Non-linear optimization algorithms used for training neural networks second order properties of the error surface. The Hessian forms the basis of a fast procedure for re-training a feed-forward network. Inverse form of this matrix is used to identify the least significant weights in a network as a part of network *pruning* algorithm and the determinant of the Hessian is used to compare the relative probabilities of a different model. However the Hessian can be calculated exactly, there are many techniques to approximate the values. [**?**]

3.2 Error functions

In the previous discussions, we have been using the sum-of-squares error function mainly due to its analytical simplicity. However, in practice there are many other error functions, which are suitable for use in neural networks.

3.2.1 Sum-of-squares error

Sum of squares is a concept that permeates much of inferential statistics and descriptive statistics. More properly, it is the sum of squared deviations. Mathematically, it is an unscaled, or unadjusted measure of dispersion (also called variability). When scaled for the number of degrees of freedom, it estimates the variance, or spread of the observations about their mean value. [**?**] The distance from any point in a collection of data, to the mean of the data, is the deviation.

$$\sum_{i=1}^{n} (y_i - t_i)^2 \tag{12}$$

This can be written as , where y_i is the $i - th$ data point, and is the estimate of the mean. If all such deviations are squared, then summed, as in , this gives the sum of squares for these data.

3.2.2 Mean squared error

In statistics, the mean squared error (MSE) of an estimator is one of many ways to quantify the difference between values implied by an estimator and the true values of the quantity being estimated. MSE is a risk function, corresponding to the expected value of the squared error loss or quadratic loss. The MSE is the second moment (about the origin) of the error, and thus incorporates both the variance of the estimator and its bias. For an unbiased estimator, the MSE is the variance. Like the variance, MSE has the same units of measurement as the square of the quantity being estimated. In an analogy to the standard deviation, taking the square root of MSE yields the root mean square error or root mean square deviation (RMS or RMSD), which has the same units as the quantity being estimated; for an unbiased estimator, the RMS is the square root of the variance, known as the standard error. The basic formula for the MSE is:

$$E_{MSE} = \frac{1}{2N} \sum_{i=1}^{n} (y_i - t_i)^2 \tag{13}$$

The MSE thus assesses the quality of an estimator in terms of its variation and unbiasedness. Note that the MSE is not equivalent to the expected value of the absolute error.

3.2.3 Root mean squared error and normalization

The root mean square error (RMS) is a frequently-used measure of the differences between values predicted by a model or an estimator and the values actually observed from the thing being modeled or estimated. RMS is a good measure of precision. These individual differences are also called residuals, and the RMS serves to aggregate them into a single measure of predictive power. The RMSD of an estimator with respect to the estimated parameter is defined as the square root of the mean square error:

$$E_{RMS} = \sqrt{\frac{1}{2N} \sum_{i=1}^{n} (y_i - t_i)^2} \tag{14}$$

In some cases we need to normalize the RMS. Normalized RMS is divided by the range of observed values as follows

$$E_{NRMS} = \frac{\mathrm{E_{RMS}}}{x_{max} - x_{min}} \tag{15}$$

The RMS error has the advantage, that its value does not grow with the size of the data set. If it has a value of unity, then the network is predicting the best data 'in the mean' while a value of zero means perfect prediction of the data.

3.3 Parameter optimization algorithms

The main problem in neural networks, the minimizing of the error function, has been widely studied. Through the time, many of the approaches have been successful, and some are directly applicable. In this chapter, we will discuss various approaches and their pros and cons. The simplest method is using of gradient descent. Then we have few heuristic modifications to the gradient descent method which improve its performance. We will also review a conjugate gradient method as another approach, slightly different from the gradient descent. Next we will take a look at the powerful Levenberg-Marquardt algorithm which is applicable specifically to the sum-of-squares function.

3.3.1 Error surfaces

The problem of error surfaces approach is to find a weight vector which minimizes an error function. For a single layer of weights with sum-of-squares error and output-unit activation function, the error function will be a quadratic function of the weights. In this case, the error function will have a general multidimensional parabolic form. There is a single minimum which can be located by a sequence of linear equations. But for more typical general networks, that means with more than one layer, the function will be non-linear. For such functions there will be many minims. All of these minim satisfies the

$$\Delta E = 0 \tag{16}$$

where ΔE denotes the gradient of E in the weight space. For some algorithms such as conjugate gradient or quasi-newton method the error function is guaranteed not to increase as result of the change to the weights. The potential disadvantage of such algorithm is that if they reach such minimum they will remain there forever, as there is no mechanism for them to escape, as it is understood this would require a temporary increase of the error. The initial choice of weights therefore determine which minimum the algorithm will converge to. Different algorithm can exhibit different behavior in the neighborhood of a minimum. [**?**]

3.3.2 Gradient descent

Gradient descent is one of the simplest network training algorithms. As for other training algorithms, we start with guessing a weight vector, which is often chosen by random. We then update the weight vector, in the way, that we move a short distance in the direction of the greatest rate of decrease of the error, that means in the direction of the negative gradient. We can run the procedure in sequential, or batch version. In the batch version, the error function gradient is evaluated after all patterns are processed, and the weights are updated by the formula:

$$\Delta w^i = -\eta \Delta E \tag{17}$$

In the sequential version, the error gradient is re-evaluated after each pattern, and the weights are updated by the formula:

$$\Delta w^i = -\eta \Delta E^n \tag{18}$$

where the different patterns in the training set can be considered in sequence or randomly. As we already discussed the parameter η is a learning rate. When the learning rate is selected sufficiently small we expect the value of E will decrease at each successive step. Because the gradient descent algorithm is reminiscent of the Robbins-Monro procedure for finding zero of a regression function we are assured of convergence when the learning rate is made to decrease at each step of the algorithm. According to this, learning rate can be modified during the learning process, although in practice often it is chosen to be of a constant value even though the guarantee of convergence is lost. The serious difficulty of this approach is that when the learning rate is too large, the algorithm may overshoot, leading to an increase in E and possibly to divergent oscillations resulting in a complete breakdown of the algorithm. On the other hand, if the parameter is chosen too low the learning process may proceed very slowly, leading to large computational time. [**?**] The significant advantage of the sequential approach against the batch approach is in cases where the data set has a lot of redundant information. For example, when we extend a dataset simply by replicating its patterns 10 times and putting them after each other to form a single dataset, the computation time for one evaluation of E in batch learning would take 10 times longer and so the whole process would take 10 times longer. On the other hand the sequential algorithm updates the weights after each pattern and so it will stay unaffected by the replication of data. Another advantage of the sequential approach is that, it has a possibility of escape from the local minimum. The gradient descent procedure we have described so far involves taking a succession of finite steps through weight space. We can instead imagine the evolution of the weight vector taking place continuously as a function of time τ. The gradient rule is then replaced by a set of coupled non-linear ordinary differential equations of the form:

$$\frac{dw_i}{dw_\tau} = -\eta \frac{dE}{dw_i} \tag{19}$$

The simple gradient descent formula represents a fixed-step forward Euler technique for solving the problem, which is a particularly inefficient approach for stiff equations. Application of specialized techniques can give a significant improvement in convergence time. One very simple technique for dealing with the problem is to add a momentum term to the gradient descent formula:

$$\Delta w^\tau = -\eta \Delta E^\tau + \mu \Delta w^{\tau - 1} \tag{20}$$

where μ is called the momentum parameter. To understand the effect of momentum on the learning process, consider a motion through the error surface. If we make the approximation that the gradient is unchanging, we can apply this formula iteratively to a long series of weight updates.

$$\Delta w = -\eta \Delta E \{1 + \mu + \mu^2 + \ldots\} = -\frac{\eta}{1 - \mu} \Delta E \tag{21}$$

The result of the momentum is to increase the effective learning rate from η to $\frac{\eta}{(1-\mu)}$. By contrast, in the error region where the curvature is oscillatory. The successive contributions from the momentum term will tend to cancel and the effective learning rate will

stay eta. This inclusion of momentum generally leads to a significant improvement in the performance of gradient descent. However, the algorithm still stays relatively inefficient. One obvious probe with simple gradient descent plus momentum algorithm is that it contains two parameters, μ and η , whose values must be selected by trial and error. The optimum values will depend on the particular problem, and will typically vary during the training. We seek a procedure for setting these automatically as part of the training algorithm. One such approach is to use the bold driver technique. Consider the situation without a momentum first. The idea is to check whether the error function has actually decreased after each step of the gradient descent. If it has increased then the algorithm must overshoot the minimum and so the learning rate parameter must have been too large. In this case the weight change is undone, and the learning rate is decreased. This process is repeated until a decrease in error is found. If, of course, the error decreased the new values are accepted. But the learning rate might be too small so its value is increased. This leads to a following formula for updating the learning rate.

$$\eta_{new} = \begin{cases} \rho\, \eta_{old}, if \Delta E < 0 \\ \sigma\, \eta_{old}, if \Delta E > 0 \end{cases} \tag{22}$$

The parameter ρ is chosen to be slightly larger than unity (example 1.1) in order to avoid frequent occurrences of an error increase, since in such cases the error evaluation is wasted. The parameter σ is taken to be significantly less then unity (example 0.5) so that the algorithm quickly reverts to find a step to decrease the error, to minimize wasted computation. Many heuristic variations are possible, such as increasing the learning rate η incrementally by a fixed value. If we consider momentum term in this situation, we can set it to a fixed value since the weight update in a negative gradient calculation is wasted so this has an effect of setting the momentum temporarily to zero. We have noted that the gradient vector need not always point towards the error function minimum. And we have seen that the long narrow valleys in the error function, can lead to very slow progress down the valley as a consequence of the need to keep the learning rate small in order to avoid divergence oscillations. An approach that has been introduced to deal with this problem is to use separate learning rates for each weight parameter and procedure an updates of these learning rates during the learning process. The gradient descent formula then becomes:

$$\Delta w_i^\tau = -\eta_i^\tau \frac{dE}{dw_i^\tau} \tag{23}$$

We might want to increase a particular learning rate when the derivative of E in respect to corresponding parameter has the same sign on consecutive steps, since this indicates that the weight is moving in steadily downhill direction. In contrast, when the derivative of E in respect to corresponding weight changes in consecutive steps, this signals an oscillation, and we want to decrease the learning parameter. To implement this we take:

$$\Delta \eta_i^\tau = g_i^\tau g_i^{\tau-1} \tag{24}$$

where

$$g_i^\tau = \frac{dE}{dw_i^\tau} \tag{25}$$

and $\eta > 0$ is the step size parameter. This is called delta-delta rule.This algorithm appears to work good in practice but the drawback is that we now have four parameters(η, ϕ, κ and μ).

3.3.3 Conjugate gradients

Another enhanced approach is to use conjugate gradient algorithm. For a general error function, the error in the neighborhood of a given point will be approximately quadratic, and so we may hope that repeated application will lead to the error function convergence to the minimum. The step length of this procedure is governed by the coefficient α_j, and the search direction is determined by the coefficient β_j. These expressions depend on the Hessian matrix. For a non-quadratic error function, the Hessian matrix will depend on the current weight vector, and so will need to be re-evaluated at each step of the algorithm. Since the evaluation of H is computationally costly for a non-linear neural network, and since its evaluation would have to be done repeatedly, we would like to avoid having to use the Hessian. In fact, it turns out that the co-efficient α_j and β_i can be found without explicit knowledge of H. This leads to the conjugate gradient algorithm.

$$\beta_j = \frac{g_{j+1}^T \left(g_{j+1} - g_j\right)}{d_j^T \left(g_{j+1} - g_j\right)} \tag{26}$$

This is known as a Hestenes - Stiefel expression. It is also true that

$$d_j^T g_j = -g_j^T g_j \tag{27}$$

which together allows to be written in the Polak-Ribiere form

$$\beta_j = \frac{g_{j+1}^T \left(g_{j+1} - g_j\right)}{g_j^T g_j} \tag{28}$$

Similarly, we can use the orthogonality property for the gradients to simplify further resulting in the Fletcher-Reeves form.

$$\beta_j = \frac{g_{j+1}^T g_{j+1}}{g_j^T g_j} \tag{29}$$

Note that these three expressions for a β_j are equivalent provided the error function is exactly quadratic. In practice, the error function will not be quadratic, and these different expressions for β_j can give different results. The Polak-Ribiere form is generally found to give slightly better results than the other expressions. This is probably due to the fact that, if the algorithm is making little progress, so that successive gradient vectors are very similar, the Polak-Ribiere form gives a small value for β_j so that the search direction tends to be reset to the negative gradient direction, which is equivalent to restarting the conjugate gradient procedure. We also wish to avoid the use of the Hessian matrix to evaluate α_j. In fact, in the case of a quadratic error function, the correct value of α_j

can be found by performing a line minimization along the search direction. To see this consider a quadratic error as a function of the parameter α along the search direction given by:

$$E(w_j + \alpha d_j) = E_0 + b^T(w_j + \alpha d_j) + \frac{1}{2}(w_j + \alpha d_j)^T H(w_j + \alpha d_j) \tag{30}$$

If we set the derivative of this expression with respect to α equal to zero we obtain:

$$\alpha_j = -\frac{d_j^T g_j}{d_j^T H d_j} \tag{31}$$

where we have used the expression for the local gradient in the quadratic approximation. The conjugate gradient algorithm has been derived on the assumption of a quadratic error function with a positive-definite Hessian matrix. For general non-linear error functions the Hessian matrix need not to be positive-definite. The search directions defined by the conjugate gradient algorithm need not then be descent directions. In practice, the use of robust linear minimization ensures that error can not increase at any step, and such algorithms have good performance generally. As we have seen the conjugate gradient provides a minimization technique which requires only the evaluation of the error function and its derivatives, and which, for a quadratic error function, is guaranteed to find the minimum in at most W steps. Since the derivation has been relatively complex, we now summarize the key steps of the algorithm:

1. Choose an initial weight vector w_1.
2. Evaluate the gradient vector g_1, and set the initial search direction $d_1 = -g_1$.
3. At step j, minimize $E(wj + \alpha d_j)$with respect to α to give $w_{j+1} = w_j + \alpha_{min} d_j$.
4. Test to see if the stopping criterion is satisfied.
5. Evaluate the new gradient vector g_{j+1}
6. Evaluate the new search direction in which β_j is given by the Hestenes-Stiefel formula, the Polak-Ribiere formula or the Fletcher-Reeves formula.
7. Set $j = j + 1$ and go to step 3.

The batch form of gradient descent with momentum involves two arbitrary parameters λ and μ where λ determines the step length, and μ controls the momentum. A major problem with the conjugate gradient is how to determine values of λ and μ, since their optimum will typically vary during training. This method is a gradient descent algorithm in which the parameters λ and μ are determined at each operation automatically. **[?]**

3.3.4 Levenberg-Marquardt algorithm

All of the algorithms, that we have discussed, were general-purpose methods designed to be used with a wide range of error functions. Levenberg-Marquardt algorithm is a method designed to be used only with the sum-of-squares error. Let's consider the sum-of-squares error: **[?]**

$$E = \frac{1}{2}\sum (\epsilon^n)^2 \tag{32}$$

where $\epsilon\hat{n}$ is the error for the nth pattern, and ϵ is a vector with elements ϵ^n. Suppose we are currently at a point w_{old} in weight space and we move to a point w_{new}. If the difference between old weight setting and new weight setting is small we can expand the error vector ϵ to first order of Taylor series

$$\epsilon(w_{new}) = \epsilon(w_{old}) + J(w_{new} - w_{old}) \tag{33}$$

where we have used matrix of fist derivatives J from previous chapters known as Jacobian

$$(J)_{ni} \equiv \frac{d\epsilon^n}{dw_i} \tag{34}$$

The error function can be then written as

$$E = \frac{1}{2}\left\|\epsilon(w_{old} + J(w_{new} - w_{old})\right\|^2 \tag{35}$$

If we minimize this error with respect to the new weights w_{new} we obtain

$$w_{new} = w_{old} - (J^T J)^{-1} J^T \epsilon(w_{old}) \tag{36}$$

For the sum-of-squares error function, the elements of the Hessian matrix take the form

$$(H)_{ik} = \frac{d^2 E}{dw_i dw_k} = \sum \left(\frac{d\epsilon^n}{dw_i}\frac{d\epsilon^n}{dw_k} + \epsilon^n \frac{d^2\epsilon^n}{dw_i dw_k} \right) \tag{37}$$

In general we do not need to compute exact value of Hessian. An suitable approximation can be calculated from Jacobian in form

$$H = J^T J \tag{38}$$

For a linear network, it would be exact. For a non-linear network, it represents an approximation, although we note that in the limit of an infinite data set the expression is exact at the global minimum of the error function. Recall that in this approximation the Hessian is relatively easy to compute, since first derivatives with respect to network weights ca be obtained very efficiently using backpropagation. In principle, the update formula 35 could be applied iteratively in order to try to minimize the error function. The problem with such an approach is that the step size which is given by 35 could turn to be large, in which case the approximation would be no longer valid. This problem is addressed by seeking to minimize the error function while at the same time trying to keep the step

size small so as to ensure that the linear approximation remain valid. This is achieved by considering a modified error function of the form:

$$E = \frac{1}{2} \|\epsilon(w_{old}) + J(w_{new} - w_{old})\|^2 + \lambda \|w_{new} - w_{old}\|^2 \tag{39}$$

where the parameter λ governs the step size. For large values of λ the quadratic difference of old weight vector and new weight vector will tend to be small. If we minimize the modified error with respect to w_{new}, we obtain

$$w_{new} = w_{old} - (J^T J + \lambda I)^{-1} J^T \epsilon(w_{old}) \tag{40}$$

where I is the unit matrix. For very small values of the parameter λ we recover the Newton formula, while for large values of λ we recover standard gradient descent. In latter case the step length is determined by λ^{-1}, so that it is clear that, for sufficiently large values of λ, the error will necessarily decrease then generates a very small step in the direction of the negative gradient. We can summarize the key steps of the algorithm as follows:

1. Compute the Jacobian.
2. Compute the error gradient.
3. Approximate the Hessian using the cross product Jacobian.
4. Calculate new weights and update the weights.
5. Recalculate the sum of squared errors using the updated weights.
6. If the sum of squared errors has not decreased.
   (a) Discard the new weights, increase λ using factor and go to step 4.
7. Else decrease lambda using factor and stop.

This algorithm is an example of a model trust region approach in which the model is trusted only within some region around the current search point. The size of this region is governed by the value of λ. In practice a value must be chosen for λ and this value should vary appropriately during the minimization process. **[?]**

4 Research of currently available neural network solutions

4.1 FANN

Fast Artificial Neural Network Library is a popular and free open source neural network library. It is written in C supports multilayer networks with full connection as well as with non-full connection. Cross-platform execution in both fixed and floating point are supported and, it includes a framework for easy handling of training data sets. The algorithm routines are written with low level C and take advantage of bitwise operations. There are also available bindings to more than 15 programming languages. Library uses RPROP, Quickprop, Batch, Incremental algorithm for learning process. [**?**]

4.2 Flood

Flood is an implementation of the multilayer perceptron neural network in the C++ programming language. Algorithms include gradient methods, newton methods and also evolutionary algorithms. Root mean square, normalized root mean square and basic sum of squares error functions are implemented for a choice. Framework for processing dataset is also available. The package is very well documented and includes examples. [**?**]

4.3 Encog

Encog is a neural network library developed in .NET and Java. Library has extensive support of various data models, like XML, SQL, CVS or image down sampling. User can choose from a range of training algorithms as backpropagation, scaled conjugate gradient, Levenberg-Marquardt algorithm or genetic algorithms. Activation functions like a sigmoid, bipolar, tangent, softmax or linear are also included in the library with a range of supported network structures. Self-organizing maps, Hopfield, Boltzmann, Adaline and of course feed forward network. [**?**]

4.4 NeuroSolutions

NeuroSolutions is a commercially used software bundle with a graphic interface. Includes all widely used network topologies like self-organizing maps, multilayer perceptron, radial basis functions. Levenber-Marquardt, gradient methods, quickprop, delta-bar-delta are available algorithms. Software is developed for windows platform, written in ANSI C++. [**?**]

4.5 SNNS

SNNS (Stuttgart Neural Network Simulator) is a well known neural network simulator developed at the University of Stuttgart. SNNS is usable on Unix platform and includes a graphical user interface if needed. Package uses a range of algorithms like backpropagation, quickprop, RPROP or kohonnen learning and others. [**?**]

5 Analysis of parallelization in neural networks

There are many approaches we can use to make the neural network learning algorithm parallel. Basic differentiation is a parallelization of a network structure and a parallelization of a training dataset. In this chapter, we will take a closer look at each one of them.

5.1 Parallelization of dataset

The principle of dataset parallelism is to create partitions of the training dataset. Each processing unit then needs to have its own copy of weight matrix. These matrices needs to be consistent, because the weights are then updated in a global operation according to change values of local matrices.

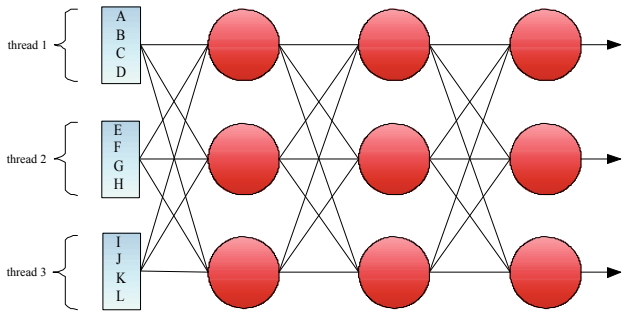

Figure 8: Dataset parallelization

The dimension degree of dataset parallelization algorithm is the number of patterns in the training set. [**?**]

5.2 Parallelization of network structure

Parallelizing the structure we are dividing the network into blocks that are able to operate on their own.

5.2.1 Layer parallelization

This approach is also called pipelining in some publications. The idea is to slice the structure horizontally, which means we get a layer object that holds all the weights (for that current layer). Each layer is then computed by a separate computing unit.

As we can see this approach mix the forward direction with the backward direction. The dimension degree of layer parallelism is the number of weight layers.

5.2.2 Node parallelization

When we are talking about the node parallelism we actually can make decision. One is to parallelize the neurons, the other is parallelization of weights. Vertical slicing or neuron

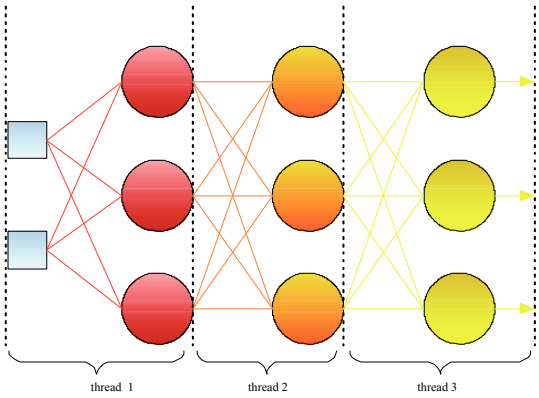

Figure 9: Layer parallelization

parallelism is the most common principle. As we can see, the idea is that all incoming

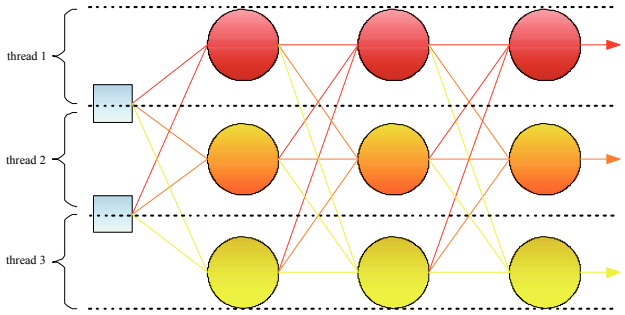

Figure 10: Neuron parallelization

weights of one neuron are calculated by one computing unit. The output of the neurons is then computed by the matrix vector product.

First each computing unit computes an output for hidden layer neurons. Then the computational units exchange the computed values and continue to compute neuron outputs of the output layer and also the error. The hidden layer error is computed based on a hidden layer error based on a vector-matrix product. In cases the number of neurons is larger than the number of computational units, each computational unit computes more than one neuron. The dimension degree of neuron parallelization is the number of neurons in hidden and in output layer. Synapse parallelism is the other kind of node parallelization. The difference is that instead of mapping the rows of weights, we are mapping the columns. So this way each computing unit computes only a partial sum (that which is in its column). Advantage of this approach is that hidden layer error can be computed without communication. The dimension degree of synapse parallelism is the number of neurons in input layer and in hidden layer. [**?**]

5.3 Parallelization of atomic operations

The deepest approach in the sense of structure of the neural network we can take is to parallelize the atomic operations. This approach, also called fine grained parallelization is a method where each cycle in the algorithm is executed in parallel in different threads with the data stored in shared memory. For such an approach, a synchronization of threads is needed, which is done in barriers. [**?**]

5.4 CPU implementation approach

Originally, algorithms programmed for CPU have been designed most of the time as a serial stream of instructions. However, there are approaches how to make algorithms run parallel on CPU.

5.4.1 Hardware requirements

We can classify parallel CPU according to their level of hardware support.

Multi-core processor is a processor, that includes multiple execution units on a same chip. This processor can issue a multiple instructions per cycle from multiple instruction streams.

Symmetric multiprocessing computer is a computer system with multiple identical processors that share memory and connect via a bus. Bus contention prevents bus architectures from scaling. As a result, Symmetric multiprocessing computers generally do not comprise more than 32 processors. Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are cost-effective, provided that a sufficient amount of memory bandwidth exists.

A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. The most common type of cluster is the Beowulf cluster, which is a cluster implemented on multiple identical commercial off-the-shelf computers connected with a TCP/IP Ethernet local area network.

A massively parallel processor is a single computer with many networked processors. Massively parallel processors have many of the same characteristics as clusters, but they have specialized interconnect networks (whereas clusters use commodity hardware for networking). They also tend to be larger than clusters, typically having far more than 100 processors.

Grid computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the Internet, grid computing typically deals only with embarrassingly parallel problems.

5.4.2 Software requirements

For creating parallel algorithms, we need some API which is capable of sending multiple instructions for multiple processing units and also process the results. Such an

API is Message Passing Interface. Message Passing Interface is the most widely used message-passing system interface. It is based on a language-independent communications protocol. Both point-to-point and collective communication are supported. The implementation language is not constrained to match the language or languages it seeks to support at runtime. Most implementations combine C, C++ and assembly language, and target C, C++, and Fortran.

Apart from the message API we also need to manage memory resources. OpenMP is one of most widely used shared memory API. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. It's an implementation of multi threading, a method of parallelization whereby the master thread (a series of instructions executed consecutively) forks a specified number of slave threads and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

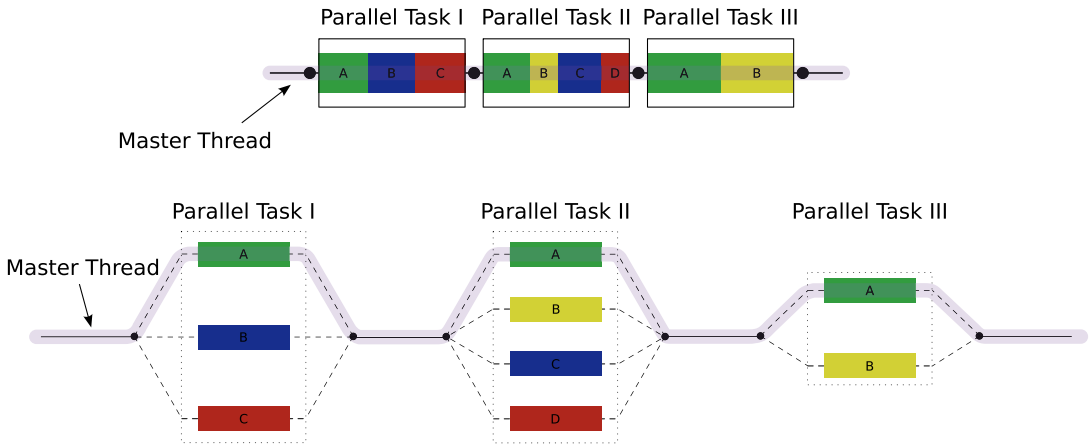

Figure 11: Diagram of parallel task execution on CPU

The CPU approach is a known method to parallelize algorithm, which of course includes the training algorithms for neural network structures. However the significant hardware requirements are not available for a reasonable price, therefore not widespread, what is the main drawback and the reason why we will not use the CPU approach for the implementation.

5.5 GPU implementation approach

Last couple years have GPU computing became more popular even for general purpose computations. The idea behind the GPU is in its multi-core architecture, which makes it a parallel instruction execution hardware. For this well known capability of GPU units vendors of GPU started to invest in software abstraction. Compute Unified Device Architecture (CUDA) is an abstraction of a programming language capable to send parallel instructions and manage memory resources.

5.5.1 Programming model

CUDA is implemented as an extension to C programming language. This allows the programmer to define special C functions called kernels. Kernel is executed N times by a CUDA device in comparison with a sequential code which is executed only once in a regular C function.

Kernel is defined with a keyword **__global__**. The number of threads, that means how many times the kernel will be executed is specified using <<< **...** >>> execution configuration syntax. Configuration specifies grid dimensions, block dimensions, number of bytes per block and CUDA stream, which executes the kernel. Last two specifications are zero in default. For the dimensions, a new variable type is specified **dim3**, which is a three component vector. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in **threadIdx** variable. As an illustration, the following sample code adds two vectors, A and B of size N and stores the result into vector C.

```
1  // Kernel definition
2  __global__ void VecAdd(float* A, float* B, float* C)
3  { int i = threadIdx.x;
4  C[i] = A[i] + B[i];
5  }
6
7  int main()
8  {
9  ... // Kernel invocation with N threads
10 VecAdd<<<1, N>>>(A, B, C);
11 }
```

Výpis 1: Kernel definition and invocation

Vector **threadIdx** can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same. For a two-dimensional block of size *(Dx, Dy)*, the thread ID of a thread of index *(x, y)* is *(x + y Dx)*. For a three-dimensional block of size *(Dx, Dy, Dz)*, the thread ID of a thread of index *(x, y, z)* is *(x + y Dx + z Dx Dy)*. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core.

```
1  VecAdd<<<dim3 GridDimensions, dim3 BlockDimensions, size_t BytesPerBlock,
       cudaStream_t>>>(A, B, C);
```

Výpis 2: Kernel configuration parameters

CUDA programming model assumes that the threads are executed on a separate device, which fulfills the function of a co-processor to the host which is running the program. All the serial code is executed on the host, and all the parallel code is executed on

the device. From this also comes the feature, that the host and the device maintain their own memory spaces separately.

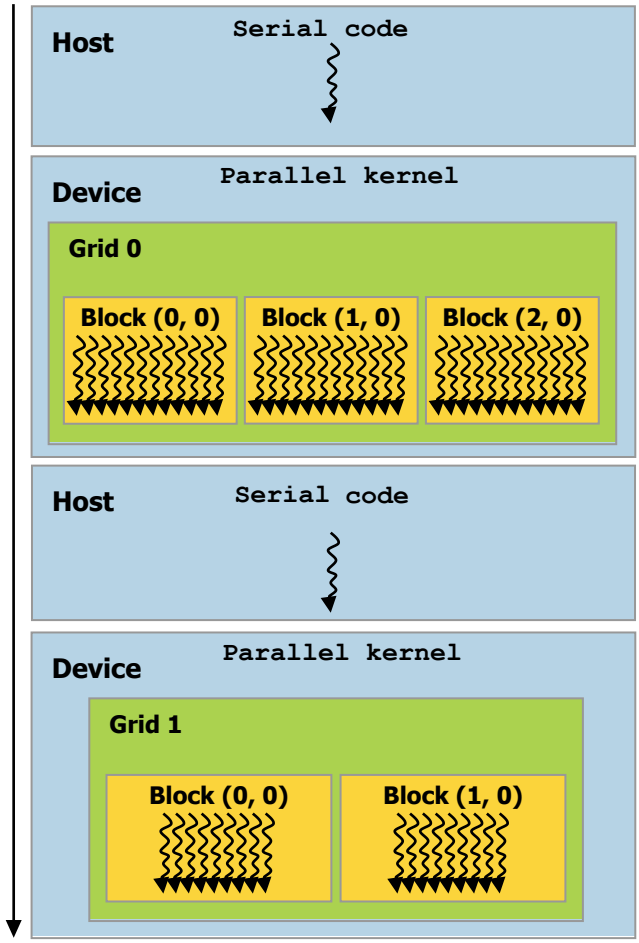

Figure 12: CUDA programming model

[?, ?, ?]

5.5.2 Memory management

CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels can only operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory. Device memory can be allocated either as linear memory or as CUDA arrays. CUDA

arrays are opaque memory layouts optimized for texture fetching. Linear memory exists on the device in a 32-bit address space for devices of compute capability 1.x and 40-bit address space of devices of compute capability 2.0, so separately allocated entities can reference one another via pointers, for example, in a binary tree. Linear memory is typically allocated using **cudaMalloc()** and freed using **cudaFree()** and data transfer between host memory and device memory are typically done using **cudaMemcpy()**. It is understood that transfer operations between device and global memory are time demanding, therefore we are always trying to minimize these transfers. [**?**, **?**, **?**]

```
1  int main()
2  {
3  size_t size = sizeof(float);
4
5  // Allocate variable h_A host memory
6  float* h_A = (float*)malloc(size);
7
8  // Allocate variable d_A device memory
9  float* d_A;
10 cudaMalloc((void**)&d_A, size);
11
12 // Copy device memory to host memory
13 cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
14
15 // Free host memory ...
16 cudaFree(d_A);
17 }
```

Výpis 3: Memory allocation and transfer

5.5.3 Global Memory

Global memory resides in device memory and device memory is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e. whose first address is a multiple of their size) can be read or written by memory transactions. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. For example, if a 32-byte memory transaction is generated for each thread.s 4-byte access, throughput is divided by 8. How many transactions are necessary and how throughput is ultimately affected varies with the compute capability of the device. For devices of compute capability 1.0 and 1.1, the requirements on the distribution of the addresses across the threads to get any coalescing at all are very strict. They are much more relaxed for devices of higher compute capabilities. For devices of compute capability 2.0, the memory transactions are cached, so data locality is exploited to reduce impact on throughput. [**?**, **?**, **?**]

5.5.4 Local Memory

Local memory accesses only occur for some automatic variables as mentioned. Automatic variables that the compiler is likely to place in local memory are: Arrays for which it cannot determine that they are indexed with constant quantities or large structures or arrays that would consume too much register space or any variable if the kernel uses more registers than available (this is also known as register spilling). The local memory space resides in device memory, so local memory accesses have same high latency and low bandwidth as global memory accesses and are subject to the same requirements for memory coalescing. Local memory is however organized such that consecutive 32-bit words are accessed by consecutive thread IDs. Accesses are therefore fully coalesced as long as all threads in a warp access the same relative address. On devices of compute capability 2.0, local memory accesses are always cached in L1 and L2 in the same way as global memory accesses. [**?**, **?**, **?**]

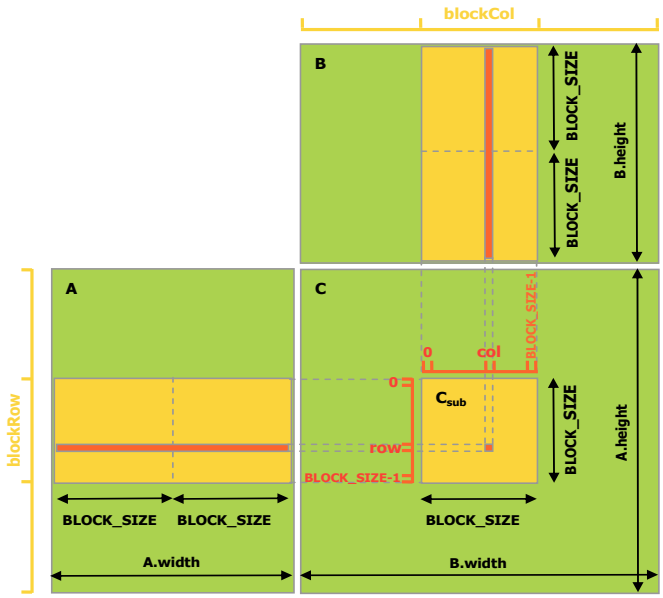

Figure 13: Computation of matrix multiplication in shared memory

5.5.5 Shared Memory

Shared memory is allocated using the **__shared__** qualifier. Because it is on-chip, the shared memory space is much faster than the local and global memory spaces. In fact, for all threads of a warp, accessing shared memory is fast as long as there are no bank conflicts between the threads, as detailed below. To achieve high bandwidth, shared

memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause $n - way$ bank conflicts. To get maximum performance, it is therefore important to understand how memory addresses map to memory banks in order to schedule the memory requests to minimize bank conflicts. Picture shows the computation of matrix multiplication in shared memory. [**?**, **?**, **?**]

5.5.6 Compilation

If it is needed, kernels for CUDA can be written in the instruction set called PTX. However usually programmers use high order languages as C or C++. In order to run, kernels must be compiled to binary form. Compiler *nvcc* is a tool which compiles the C program to PTX code and maintains the links. [**?**, **?**, **?**]

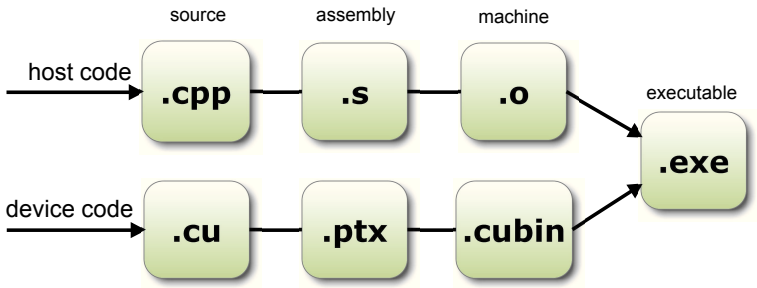

Figure 14: Compilation process of CUDA and C code

Source files compiled with *nvcc* can include a mix of host code and device code. Basic workflow consists in separating device code from host code and compiling the device code into an assembly form (PTX code) and/or binary form. The generated host code is output either as C code that is left to be compiled using another tool or as object code directly by letting *nvcc* invoke the host compiler during the last compilation stage. The CUDA can also be emulated to be used with a systems which do not have a compatible GPU. This option can be defined either using macro in the code or as a parameter to compiler. [**?**, **?**, **?**, **?**]

5.5.7 CUDA device

CUDA represents a scalable programming model which allows wide processor and memory scaling. The GPU will set the method to utilize available cores of the device, that means we can run CUDA programs on any NVIDIA CUDA capable device, but the overall performance will vary with the device. Simply a device with more cores will execute the multi-threaded program in less time than a device with fewer cores.

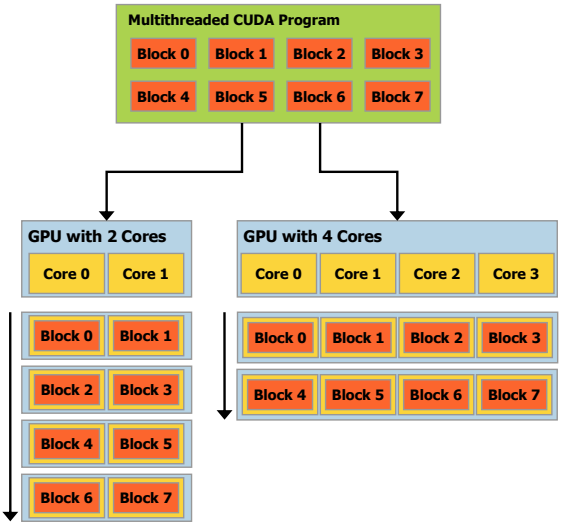

Figure 15: Parallel kernel invocation on multiple devices

For our implementation we will be using the powerful Tesla C2050 multi core platform. The Tesla platform utilize 32 cores per a multiprocessor and has following specifications:

Processor clock rate	1.15 GHz
Amount of global memory	2.81 Gb
Number of multiprocessors	14
Number of processor cores	448
Shared memory per block	49152 bytes
Maximum block dimensions	(1024, 1024, 64)
Maximum grid dimensions	(65535, 65535, 1)
Warp size	32
Compute capability	2.0

Table 1: Tesla C2050 specifications

[?]

6 Implementation of parallel algorithm in CUDA

6.1 Overview

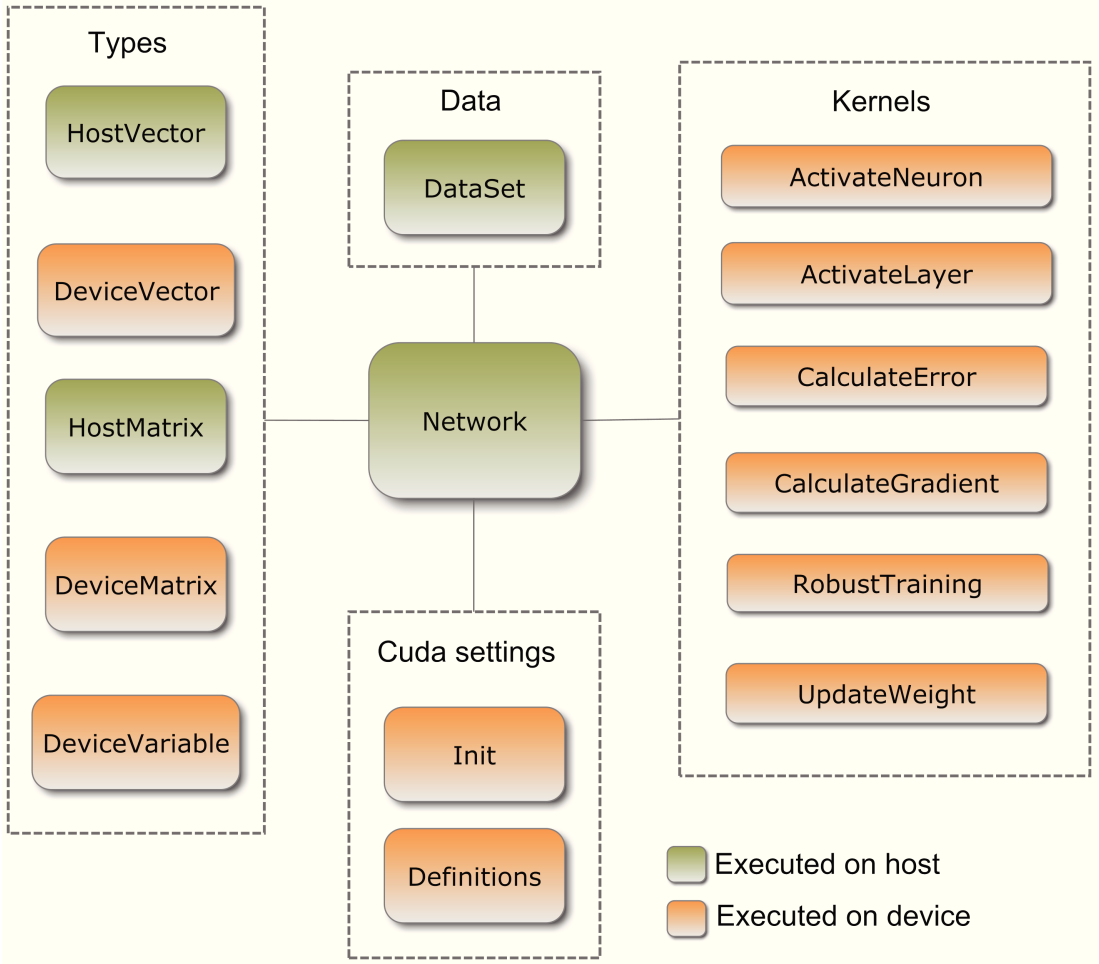

Figure 16: Implementation overview

The implementation is written in CUDA C language combining the parallel kernels executed on the GPU device and sequential instructions executed on the host. Design is centered to class *Network* which uses all CUDA definitions, kernels, types, and data model. *Definitions* define macros to set the algorithm on the environments for any capable device and also for the emulation. There are also macros which switch between types to be used in the algorithm according to the computing capability of the device. Initial check of device properties is done at the initialization and so are retrieved the properties of the device according to which are the definitions chosen. Initialization also checks for the total number of threads allowed by the device, and set the value to be used in the training. The device kernels are divided to the stages of backpropagation algorithm.

Each kernel works in shared device memory to maximize the performance. Transfers between host memory and device memory are minimal, since whole dataset is loaded to the device memory at the initialization of training. The last overall error value (RMS) is stored at each epoch in the *DeviceVariable* type variable, which is accessible to the device. This type is designed to be updated asynchronously. Therefore it can be accessed multiple times and would not slow down the training process. The *DataSet* class holds the implementation of I/O functionality. The input is presented in a plain text format. The device and the host have their own data structures derived from base classes. That means, for computation at a device side we use *DeviceVector* and *DeviceMatrix* classes and to read the actual values on host side we use *HostVector* and *HostMatrix* classes. Implementation represents the batch backpropagation algorithm. The parameter optimization is done by a gradient descent algorithm with variable stepping and learning rate memory for each weight. Overall error is evaluated after each epoch and for testing purposes there is also timer, which holds the elapsed time for particular stages of one epoch as well as a timer calculating overall elapsed time of the training process. Methods for displaying the neural network outputs is also implemented and as a default it calculates outputs for all the patterns of the dataset after the training process is successfully finished. The parameters of the network structure can be saved to a document, represented as a plain text values of each weight in a particular layer. By a default, algorithm looks for these documents holding the parameter values, and if they are found, the weight values are loaded. The training process continues when the overall error of the network is still higher than expected. This way we are able to execute training process for a specific network and dataset continuously. The file containing the dataset values should be in following format:

```
1  #Training set
2  #Number of inputs
3  2
4  #inputs description <name> <min> <max>
5  x1 0.0 1.0
6  x2 0.0 1.0
7  #Number of outputs
8  1
9  #outputs description <name> <min> <max>
10 out 0.0 1.0
11 #Number of patterns
12 140
13 #Patterns <inputs> | <outputs>
14 0.6200877192982455 0.5163709677419356 | 0.0
15 .
16 .
17 .
```

Výpis 4: Dataset format

Deep insight to the implementation of each stages of backpropagation will be discussed in next pages.

6.2 Single epoch

First stage of the backpropagation algorithm is calculation of output. The cycle loops through all network layers, when in each layer a kernel for layer activation is executed. The kernel calculates the outputs of all neurons in the layer. Outputs are calculated for all patterns.

```
1 void Network::train()
2 {
3   for(int l = 0; l < num_layers; l++)
4   {
5     layers[l].KERNEL_activate_layer(...);
6   }
```

Výpis 5: Output calculation

In second stage the overall network error is calculated. This error represents RMS for all patterns in the dataset. The value is then asynchronously updated in the host memory. A decision block decides whether to use robust process.

```
1     KERNEL_calculate_error(...);
2
3     rms.update_value_async(...);
4
5     KERNEL_robust_training(...);
```

Výpis 6: Overall error calculation and robust algorithm

Third stage is the calculation of local gradient values for all the units in a network layer. The loop runs backwards through all network layers till the input layer. Gradient values are calculated for all patterns.

```
1   for(int l = num_layers - 2; l >= 0; l--)
2   {
3     layers[l].KERNEL_calculate_gradient(...);
4   }
```

Výpis 7: Local gradient calculation

Fourth and last stage of the backpropagation algorithm is weight update process. Cycle loops through all layers and updates the weights. Finally, the epoch count is incremented.

```
1   for(int l = num_layers - 1; l >= 0; l--)
2   {
3     layers[l].KERNEL_update_weights(...);
4   }
5   epoch++;
6 }
```

Výpis 8: Weights update

6.3 Neuron activation kernel

Neuron activation kernel calculates output of a current neuron for all patterns in the dataset. The multiplication of weight vector and input vector is done in the device shared memory. Output of the neuron is stored in array *outputs[n]* where n is the pattern.

```
1  __global__ activate_layer_neurons(...)
2  {
3     extern __shared__ double iw[];//iw is in shared memory
4
5     iw[threadIdx.x] = 0.0;//all cells are cleared with zero
6
7     for (int i = threadIdx.x; i <= num_inputs; i += blockDim.x) //for all
          weights, stepping by neurons
8     {
9
10       double i_w = weights[blockIdx.x * (num_inputs + 1) + i]; //get
             weight index
11
12       if (i > BIAS) i_w *= inputs[blockIdx.y * num_inputs + (i - 1)]; //
             mutliplication of weight and input value i from pattern
13
14       iw[threadIdx.x] += i_w; //storing in iw[]
15
16    }
17    __syncthreads();
18     if (blockSize >= 1024)
19    {
20       if (threadIdx.x < 512) iw[threadIdx.x] += iw[threadIdx.x + 512];//
             summation of the deltas according to the blocksize
21        __syncthreads();
22    }
23
24    if (blockSize >= 512)
25    {
26       if (threadIdx.x < 256) iw[threadIdx.x] += iw[threadIdx.x + 256];//
             sum of connections according to the blocksize
27       __syncthreads();
28    }
29
30    if (blockSize >= 256)
31    {
32       if (threadIdx.x < 128) iw[threadIdx.x] += iw[threadIdx.x + 128];//
             sum of connections according to the blocksize
33       __syncthreads();
34    }
35
36    if (blockSize >= 128)
37    {
38       if (threadIdx.x < 64) iw[threadIdx.x] += iw[threadIdx.x + 64];//sum
             of connections according to the blocksize
39       __syncthreads();
40    }
41
```

```
42      if (threadIdx.x < 32)
43      {
44          if (blockSize >= 64) iw[threadIdx.x] += iw[threadIdx.x + 32];//sum
                of connections according to the blocksize
45          if (blockSize >= 32) iw[threadIdx.x] += iw[threadIdx.x + 16];//sum
                of connections according to the blocksize
46          if (blockSize >= 16) iw[threadIdx.x] += iw[threadIdx.x + 8];//sum of
                 connections according to the blocksize
47          if (blockSize >= 8) iw[threadIdx.x] += iw[threadIdx.x + 4];//sum of
                connections according to the blocksize
48          if (blockSize >= 4) iw[threadIdx.x] += iw[threadIdx.x + 2];//sum of
                connections according to the blocksize
49          if (blockSize >= 2) iw[threadIdx.x] += iw[threadIdx.x + 1];//sum of
                connections according to the blocksize
50
51          if (threadIdx.x == 0)
52          {
53              int n = blockIdx.y * gridDim.x + blockIdx.x;//get neuron index
54
55              double output = sigmoid(iw[0]);//calculate output for this neuron
56
57              outputs[n] = output; //store output in outputs[]
58          }
59      }
60  }
```

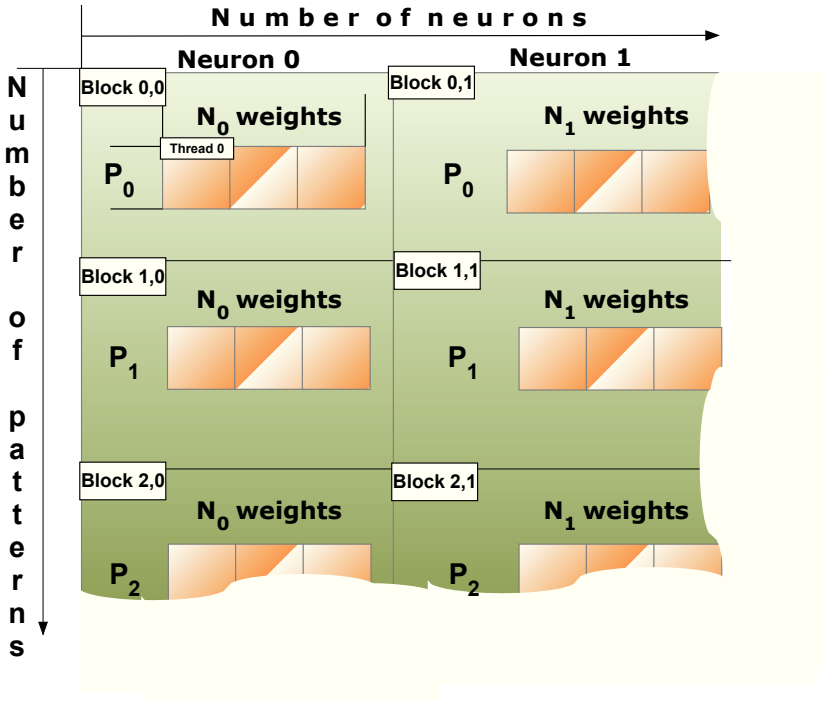

Figure 17: Neuron activation kernel

6.4 Layer activation kernel

This kernel calculates the outputs of all neurons in a current layer. Position of input i for pattern p is determined by (index of pattern * number of inputs + index of input). Position of weight between neuron j and neuron k is determined by (index of neuron j * (number of inputs + 1) + index of neuron k + 1). Position of output of neuron n for pattern p is determined by (index of pattern * number of neurons + index of neuron)

```
1  __global__ activate_layer(...)
2  {
3     extern __shared__ double iw[]; // iw is allocated in shared memory
4
5     int connection = threadIdx.y * blockDim.x + threadIdx.x;//get the index
          of a connection for specific neuron and input
6
7     sum_input_weight(connection, inputs, weights); //sum input and weight
          for this connection
8
9
10    if (INPUT == 0) //iw[0]stores the sum (the activation) of the neuron
11    {
12       int n = blockIdx.x * blockDim.y + threadIdx.y;//gets index of neuron
             in the layer
13
14       double output = sigmoid(iw[connection]); // calculates output of
             neuron n, input to the function is the sum stored in cell iw[0]
15
16       outputs[n] = output;//stores output of neuron n in array outputs
17    }
18 }
```

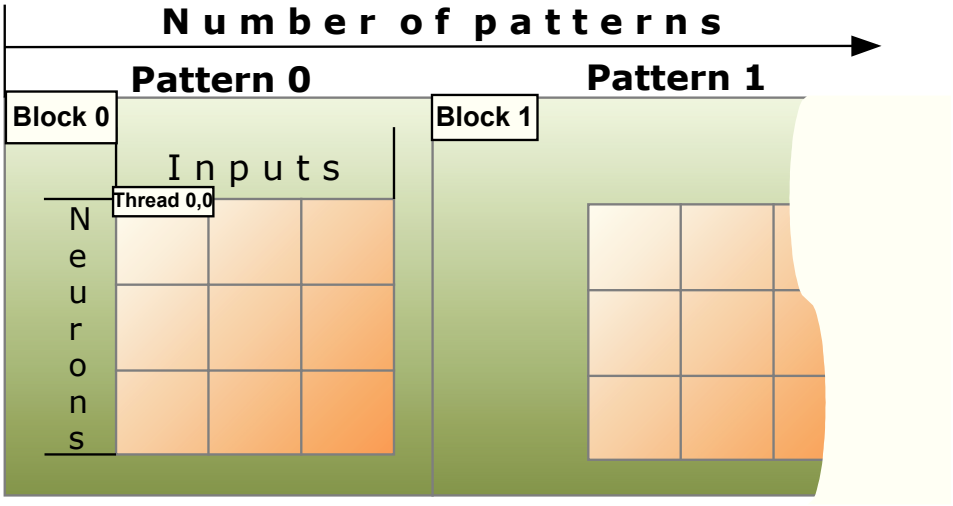

Figure 18: Layer activation kernel

For an output layer, we have defined another kernel, which calculates the same outputs of all units for all patterns but also calculates the local gradients of all the units. The kernel also does a partial calculation of the root mean square error for the output layer neurons. Position of local gradient for a neuron n and pattern p is determined as (index of pattern * number of neurons + index of neuron). There is also parameter which holds an array of desired outputs and a position of an output a neuron n and pattern p is determined as (index of pattern * number of neurons + index of neuron).

```
1  __global__ activate_output_layer(...)
2  {
3     extern __shared__ double iw[]; //iw is in shared memory
4
5     int connection = threadIdx.y * blockDim.x + threadIdx.x;//get the index
          of a connection for specific neuron and input
6
7     sum_input_weight(connection, inputs, weights);//sum input and weight
          for this connection
8
9     double * shared_rms = (iw + (blockDim.x * blockDim.y));//sets the
          pointer of share_rms to point just after the array with neurons*
          inputs
10
11    if (threadIdx.x == 0)//iw[0] stores the sum (the activation) of the
          neuron
12    {
13       int n = blockIdx.x * blockDim.y + threadIdx.y;//get the index of the
             neuron
14
15       double output = sigmoid(iw[connection]);//calculate the output of
             the neuron n
16
17       double error = (desired_outputs[n] - output);//calculate the error
             as a difference between desired and actuall output
18
19       outputs[n] = output;//stores output of neuron n in outputs[n]
20
21       local_gradient[n] = error * sigmoid_derivate(output);//calculates
             local gradient for the neuron n
22
23       shared_rms[threadIdx.y] = error * error; //stores RMS of current
             neuron in the shared_rms[] array
24    }
25
26    if (blockDim.y > 1) // if there is more than one output neuron the
          overall RMS for the pattern will be the sum of partial RMS
27    {
28       __syncthreads();
29
30
31       if (threadIdx.x == 0 && (threadIdx.y & 1) == 0 && threadIdx.y + 1 <
             blockDim.y) //if (INPUT == 0 && (threadIdx.y & 1) == 0 &&
             threadIdx.y + 1 < blockDim.y)//if input index is 0 and neuron
```

```
            index is other (higher) than 1 and lower then number of all
            neurons
32          {
33              shared_rms[threadIdx.y] += shared_rms[threadIdx.y + 1];//sum all
                    partial rms of all neurons
34          }
35          __syncthreads();
36
37          int next_interval;
38
39          for (int interval = 2; interval < blockDim.y; interval =
                next_interval)
40          {
41              next_interval = interval << 1; //left bitwise = multiplication by
                     2
42
43              if (threadIdx.x == 0 && (threadIdx.y & (next_interval - 1)) == 0
                    && threadIdx.y + interval < blockDim.y)
44              {
45                  shared_rms[threadIdx.y] += shared_rms[threadIdx.y + interval];
                        //summation of rms by intervals of 2 cells
46              }
47              __syncthreads();
48          }
49      }
50
51      if (threadIdx.y == 0 && threadIdx.x == 0)
52      {
53          rms[blockIdx.x] = shared_rms[0]; //overall RMS which is summed in
                shared_rms[0] for a pattern is stored in rms[pattern]
54      }
55  }
```

6.5 Gradient calculation kernel

This kernel calculates the local gradients for all the units in hidden layers. The initial input is the gradient of output layer calculated in the kernel of the output layer. Kernel has the address of the best RMS value. If the current RMS value is higher than the actual best RMS value multiplied by a maximum error growth factor, the algorithm applies the principle of robust training and does not continue to calculate the gradient for hidden layers. This means the last weight update values are then discarded. The step size is reduced and the algorithm recalculates the output of network with the reduced step size. If the robust training is not applied the best RMS value is NULL, and the algorithm continues to calculate the gradient for all the hidden layers for all the patterns.

```
1  __global__ calculate_gradient(...)
2  {
3     extern __shared__ double lg[]; //lg resides in shared memory
4
5     if (best_rms != NULL) //checks we have obtained best rms value already
6     {
7        __shared__ double rms;
8
9        __shared__ double b_rms;
10
11       rms = *rms_f; //get last calculated RMS
12
13       b_rms = *best_rms; //get best rms
14
15       if (rms >= b_rms * max_error_growth) return; // if the last
             calculated rms is higher than best rms multiplied by allowed
             growth factor do not continue with the calculation of gradients
16    }
17
18    double * lg_next_layer = (lg + (blockDim.x * blockDim.y)); //get
          pointer of the local gradient array for next layer
19
20    if (threadIdx.y == 0)
21    {
22       lg_next_layer[threadIdx.x] = local_gradient_next_layer[blockIdx.x *
             blockDim.x + threadIdx.x];//get value of gradient of a neuron in
              next layer
23    }
24
25    int connection = threadIdx.x * blockDim.y+1 + threadIdx.y + 1; //get
          the index of connection of neuron from current layer to previous
26
27    int thread_id = (threadIdx.y * blockDim.x + threadIdx.x); //get index
          of output neuron for the gradient calculation
28
29    __syncthreads();
30
31    lg[thread_id] = weights[connection] * lg_next_layer[threadIdx.x]; //
          multiplication of weight with next layer neuron gradient gives the
          gradient of the current neuron
```

```
32
33      __syncthreads();
34
35      int number_elem_sum = blockDim.x;
36
37      for (int sum_up_to = (number_elem_sum >> 1); number_elem_sum > 1;
            sum_up_to = (number_elem_sum >> 1))//weight values summation with
            bitwise shift array division
38      {
39          int next_number_elem_sum = sum_up_to;//value will be used in next
                iteration
40
41          if (number_elem_sum & 1) //for last cell in array the value is 1
42          {
43              next_number_elem_sum++; // so must be doubled to be able to added
                    to the sum
44          }
45
46          if (threadIdx.x < sum_up_to)
47          {
48              lg[thread_id] += lg[thread_id + next_number_elem_sum];//gradient
                    values from the next layer are summed
49          }
50
51          number_elem_sum = next_number_elem_sum; // value will be used in
                next iteration
52
53          __syncthreads();
54      }
55
56      if (threadIdx.x == 0)
57      {
58          int n = blockIdx.x * blockDim.y + threadIdx.y; //get index of neuron
59
60          double Fh = outputs[n]; //get output value of the neuron n
61
62          double lgn = lg[thread_id]; //get the gradient value for the neuron
                n
63
64
65          local_gradient[n] = lgn * sigmoid_derivate(Fh); // calculate the
                partial derivation and therefore local gradient of neuron n
66      }
67  }
```

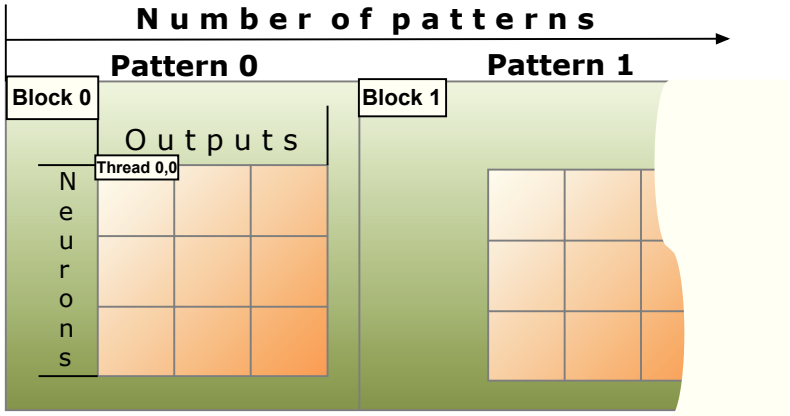

Figure 19: Gradient activation kernel

6.6 Error calculation kernel

This kernel provides the operation of calculating the overall RMS error of network for all patterns, using the partial RMS errors calculated in the output layer kernel.

```
1  __global__ calculate_rms(...)
2  {
3     extern __shared__ double shared_rms[]; //rms resides in shared memory
4
5     shared_rms[threadIdx.x] = CUDA_VALUE(0.0); //clear the cells with zeros
6
7     for (int p = threadIdx.x; p < number_patterns; p += blockDim.x) //for
          all patterns
8     {
9        shared_rms[threadIdx.x] += rms[p]; //get error from rms[] for
             pattern p
10    }
11    __syncthreads();
12     if (blockSize >= 1024)
13    {
14       if (threadIdx.x < 512)
15       {
16          shared_rms[threadIdx.x] += shared_rms[threadIdx.x + 512];//
                summation of the deltas according to the blocksize
17       }
18       __syncthreads();
19    }
20
21    if (blockSize >= 512)
22    {
23       if (threadIdx.x < 256)
24       {
25          shared_rms[threadIdx.x] += shared_rms[threadIdx.x + 256];//
                summation of partial errors according to the max block size
26       }
```

```
27          __syncthreads();
28      }
29
30      if (blockSize >= 256)
31      {
32          if (threadIdx.x < 128)
33          {
34              shared_rms[threadIdx.x] += shared_rms[threadIdx.x + 128];//
                    summation of partial errors according to the max block size
35          }
36          __syncthreads();
37      }
38
39      if (blockSize >= 128)
40      {
41          if (threadIdx.x < 64)
42          {
43              shared_rms[threadIdx.x] += shared_rms[threadIdx.x + 64];//
                    summation of partial errors according to the max block size
44          }
45          __syncthreads();
46      }
47
48      if (threadIdx.x < 32)
49      {
50          if (blockSize >= 64) shared_rms[threadIdx.x] += shared_rms[threadIdx
                .x + 32];//summation
51          if (blockSize >= 32) shared_rms[threadIdx.x] += shared_rms[threadIdx
                .x + 16];//summation
52          if (blockSize >= 16) shared_rms[threadIdx.x] += shared_rms[threadIdx
                .x + 8];//summation
53          if (blockSize >= 8) shared_rms[threadIdx.x] += shared_rms[threadIdx.
                x + 4];//summation
54          if (blockSize >= 4) shared_rms[threadIdx.x] += shared_rms[threadIdx.
                x + 2];//summation
55          if (blockSize >= 2) shared_rms[threadIdx.x] += shared_rms[threadIdx.
                x + 1];//summation
56          if (threadIdx.x == 0)
57          {
58
59              double f_rms = sqrt(shared_rms[0] / number_patterns_neurons) /
                    CUDA_VALUE(2.0); // calculate overall RMS error of the
                    network for the whole dataset
60
61              if (isnan(f_rms) || isinf(f_rms))//if result is not number or is
                    infinity
62              {
63                 f_rms = number_patterns_neurons;
64              }
65
66              *rmsF = f_rms;//updates the value of current rms
67          }
68      }
69  }
```

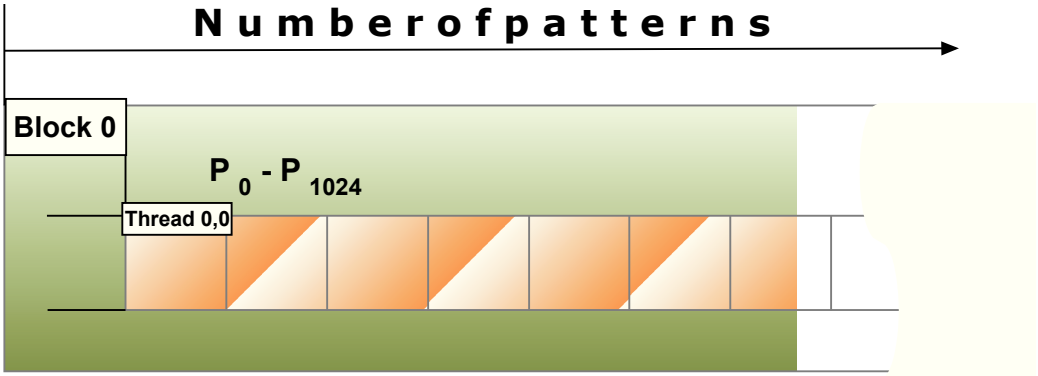

Figure 20: Error calculation kernel

6.7 Weight correction kernel

The kernel for updating the layer weights of a specific layer, also updates the step sizes. First there is a check if the robust training is used and whether the criteria is met. Then the delta values for all the units and patterns are calculated. Calculated delta value is then divided by the number of patterns and applied to the weight as a multiplication of the learning rate and direction factor with the addition of a momentum of the delta value from last update.

```
1  __global__ update_weights(...)
2  {
3     extern __shared__ double deltas[]; //reside in shared memory
4
5     if (best_rms != NULL)
6     {
7        __shared__ double rms;
8        __shared__ double b_rms;
9
10       rms = *rms_f; //last calculated rms
11
12       b_rms = *best_rms; //best rms
13
14       if (rms >= b_rms * max_error_growth) return; // if the last
             calculated rms is higher than best rms by allowed growth factor
             the kernel does not continue in updating the weights
15    }
16
17    deltas[threadIdx.x] = CUDA_VALUE(0.0); //clears the deltas array
18
19    for (int p = threadIdx.x; p < number_patterns; p += blockDim.x)
20    {
21       double delta = local_gradient[p * gridDim.y + blockIdx.y]; //get
             local gradient for a pattern p and current neuron
```

```
22
23          if (blockIdx.x > 0)
24          {
25              delta *= inputs[p * gridDim.x-1 + (blockIdx.x - 1)]; //
                    multiplication of the gradient with the input value
26          }
27
28          deltas[threadIdx.x] += delta;//store the delta to deltas[] array
29      }
30      __syncthreads();
31      if (blockSize >= 1024)
32      {
33          if (threadIdx.x < 512)
34          {
35              deltas[threadIdx.x] += deltas[threadIdx.x + 512];//summation of
                    the deltas according to the blocksize
36          }
37          __syncthreads();
38      }
39
40      if (blockSize >= 512)
41      {
42          if (threadIdx.x < 256)
43          {
44              deltas[threadIdx.x] += deltas[threadIdx.x + 256];//summation of
                    the deltas according to the blocksize
45          }
46          __syncthreads();
47      }
48
49      if (blockSize >= 256)
50      {
51          if (threadIdx.x < 128)
52          {
53              deltas[threadIdx.x] += deltas[threadIdx.x + 128];//summation of
                    the deltas according to the blocksize
54          }
55          __syncthreads();
56      }
57
58      if (blockSize >= 128)
59      {
60          if (threadIdx.x < 64)
61          {
62              deltas[threadIdx.x] += deltas[threadIdx.x + 64];//summation of
                    the deltas according to the blocksize
63          }
64          __syncthreads();
65      }
66
67      if (threadIdx.x < 32)
68      {
69          if (blockSize >= 64) deltas[threadIdx.x] += deltas[threadIdx.x +
                32];//summation
```

```
70        if (blockSize >= 32) deltas[threadIdx.x] += deltas[threadIdx.x +
              16];//summation
71        if (blockSize >= 16) deltas[threadIdx.x] += deltas[threadIdx.x + 8];
              //summation
72        if (blockSize >= 8) deltas[threadIdx.x] += deltas[threadIdx.x + 4];
              //summation
73        if (blockSize >= 4) deltas[threadIdx.x] += deltas[threadIdx.x + 2];
              //summation
74        if (blockSize >= 2) deltas[threadIdx.x] += deltas[threadIdx.x + 1];
              //summation
75
76        if (threadIdx.x == 0)
77        {
78           int connection = blockIdx.x * gridDim.x + blockIdx.x; //get the
                 index of connection for current neuron
79
80           double delta = deltas[0] / number_patterns; //divide the sum of
                 all deltas for all patterns with the paterns number
81
82           double learn_rate = learning_rate[connection];//get the learn
                 rate for a connection
83
84           double factor = same_direction(
                 last_delta_without_learning_momentum[connection], delta) ? u
                 : d; // set the reducing factor according to comparision of
                 the last delta and current delta of a connection
85
86           learn_rate *= factor; //update the leaning rate with the reducing
                  factor
87
88           if (learn_rate > max_step_size) //if the resulting learn rate is
                 higher then maximum step size
89           {
90              learn_rate = max_step_size; //set the learn rate to the
                    maximum
91           }
92
93           learning_rate[connection] = learn_rate; //store the learn rate
                 for a current connection in learn_rate[]
94
95           last_delta_without_learning_momentum[connection] = delta;//
                 update the last delta
96
97           delta += momentum * last_delta[connection]; // add the momentum
                 of last delta to current delta
98
99           last_delta[connection] = delta; //store the delta for current
                 connection in last_delta[]
100
101          double w = weights[connection] + (learn_rate * delta);// multiply
                  the delta with learning rate and update the last weight
                 value for current connection
102
```

```
103             if (isnan(w) || isinf(w)) //if the result is not number or it si
                    infinity
104             {
105                 last_delta[connection] = CUDA_VALUE(0.0); //clear the momentum
                        memories for the current connection
106
107                 last_delta_without_learning_momentum[connection] = CUDA_VALUE
                        (0.0); //clear the delta for the current connection
108
109                 if (best_rms != NULL) //if we have obtained the best rms value
110                 {
111                     learn_rate *= r; // update the learn ret with the reducing
                            factor
112
113                     learning_rate[connection] = learn_rate; //store the learn
                            rate for the current connection in learning_rate[]
114                 }
115             }
116             else
117             {
118                 weights[connection] = w; // if everything is alright update
                        the weight value for the current connection
119             }
120         }
121     }
122 }
```

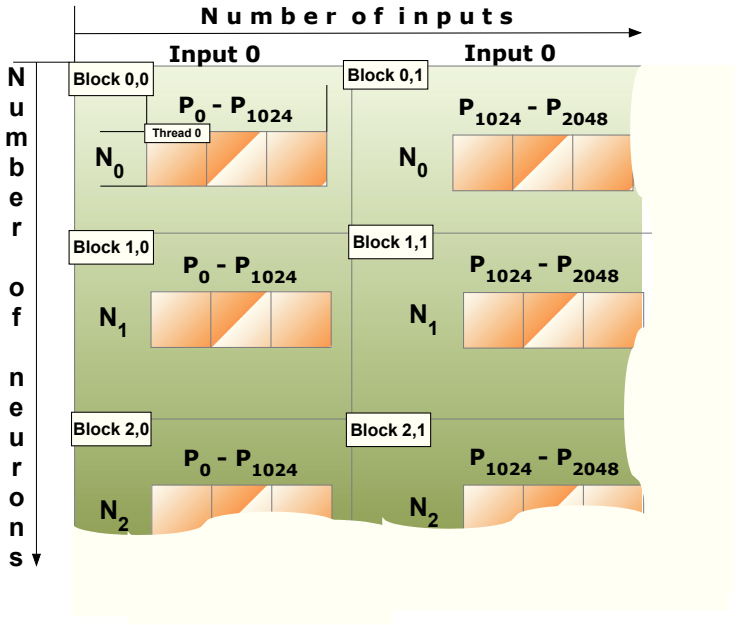

Figure 21: Weights update kernel

6.8 Robust training kernel

This kernel applies the principle of robust training. First it checks if the actual RMS is lower than the minimum obtained so far. If that is true, the minimum value is adjusted and the weights of network are stored. Otherwise kernel checks if the actual RMS exceeded the best RMS by a given tolerance and in affirmative case, the best weights are restored, the learning rate is reduced by a reducing factor, and the momentum memories are set to zero.

```
1  __global__ robust_training(...)
2  {
3     __shared__ double rms;
4
5     __shared__ double b_rms;
6
7     rms = *rms_f;//last calculated RMS
8
9     b_rms = *best_rms; // best rms
10
11    if (rms < b_rms) // if the actual rms is lower then the best rms
12    {
13       for (int layer = 0; layer < layers_number; layer++) //for all layers
14       {
15          if (threadIdx.x < number_weights[layer])
16          {
17             best_weights[layer][threadIdx.x] = weights[layer][threadIdx.x
                  ]; // than the actual weights are stored as best weights
18          }
19       }
20
21       if (threadIdx.x == 0) *best_rms = rms; //best rms is updated with
             the current
22    }
23    else if (rms >= b_rms * max_error_growth) //if the actual rms is higher
           than the best rms
24    {
25       for (int layer = 0; layer < layers_number; layer++) //for all layers
26       {
27          if (threadIdx.x < number_weights[layer])
28          {
29             weights[layer][threadIdx.x] = best_weights[layer][threadIdx.x
                  ]; //restore the best weights
30             learning_rate[layer][threadIdx.x] *= r; //update the learning
                  rate with the reducing factor
31             last_delta_without_learning_momentum[layer][threadIdx.x] =
                  CUDA_VALUE(0.0); // last delta values are cleared
32
33             lastDelta[layer][threadIdx.x] = CUDA_VALUE(0.0); // also
                  momentum memories are cleared
34          }
35       }
36    }
37 }
```

7 Testing and performance comparison

For testing the performance, we need a significant amount of data to present to our network structure. MNIST handwritten digit database is a reasonable source. The testing dataset includes ten thousand patterns of digits 0 to 9. The training process of the parallel algorithm was compared to the sequential algorithm of the FANN library. Both algorithms were executed on the Tesla C2050 machine, and ran 265 epochs. Below are the results of the comparison tests.

Network configuration	784-25-15-10
Initial learning rate	0.5
Momentum	0.7
Maximum error growth factor	0.02
Robust factor	0.5
Maximum step size	10
Up step	1.9
Down step	0.1
Desired error	0.04

Table 2: Parameter configuration

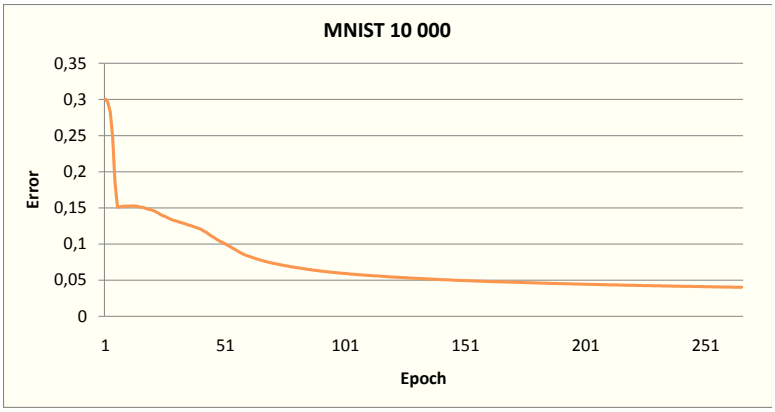

Figure 22: Error minimization of parallel algorithm

Final error	0.0399
Total epochs	265
Total elapsed time(s)	167

Table 3: Results of parallel algorithm

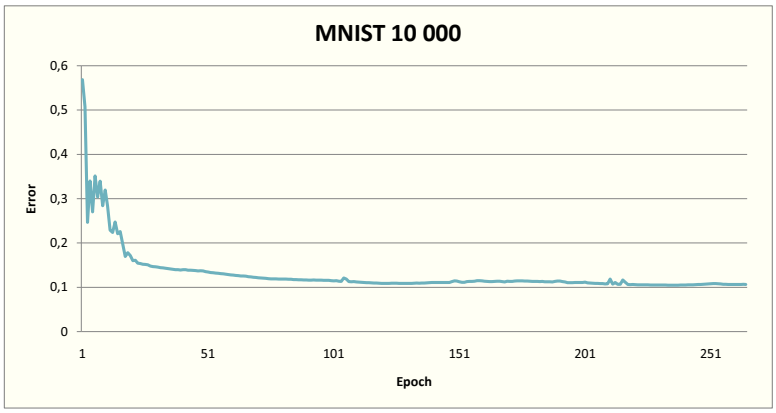

Figure 23: Error minimization of sequential FANN algorithm

Final error	0.106
Total epochs	265
Total elapsed time(s)	914

Table 4: Results of sequential FANN algorithm

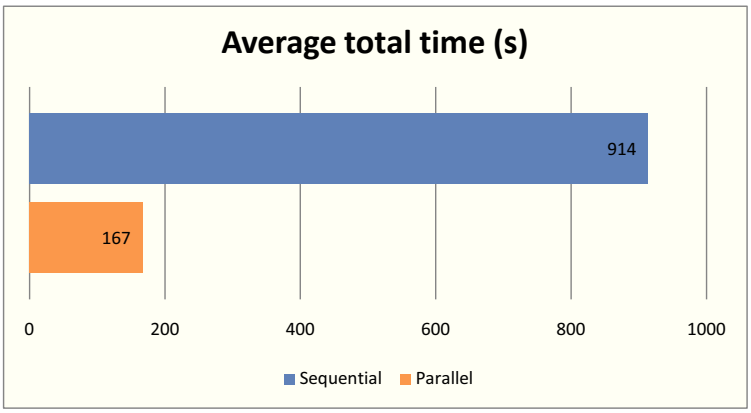

Figure 24: Total elapsed time for 265 epochs

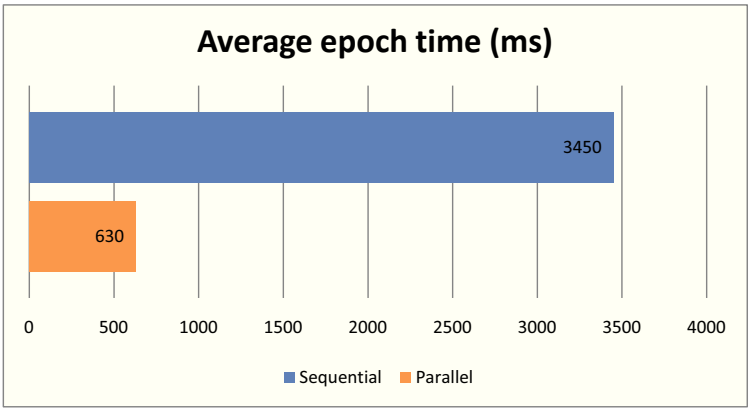

Figure 25: Elapsed time for 1 epoch

Sequential algorithm	3450
Parallel algorithm	630

Table 5: Elapsed time for 1 epoch

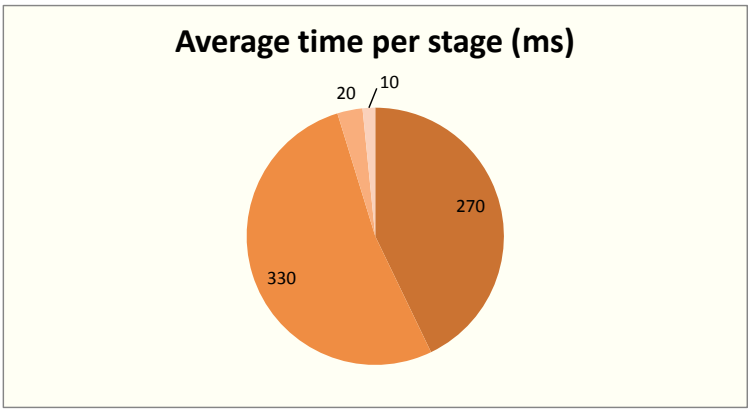

Figure 26: Stages time duration for 1 epoch of parallel algorithm

Output calculation	270
Error calculation	330
Gradient calculation	20
Weights update	10

Table 6: Stages time duration for 1 epoch of parallel algorithm

8 Conclusion

Writing of this thesis helped me to get knowledge and experience in the field of neural network algorithms which I was interested in. The pattern recognition problem, which I chose is a well known chapter of neural network applications. Therefore, the problem with excessive time demand of training backpropagation algorithms for such a neural structure is discussed with great interest of research community. Most of the documents I have gained the knowledge from were scientific articles from international science journals, while I have experienced a slight lack of relevant materials in local resources, especially in the chapter of parallelization of network structure. However, the consultations with my supervisor were very helpful and led me the right way through the whole process. All the sources I have used are listed in the *References* section.

Parallelization of dataset combined with parallelizing the atomic operations of the backpropagation algorithm in CUDA framework showed to be the right solution to accelerate the learning process of neural networks used for pattern recognition. The only limit is the number of neurons in the neural structure as the algorithm is dataset parallel. However, in most problem cases we deal with large dataset we want to present to the neural structure. Therefore, this parallel algorithm is applicable to various other applications.

For testing purposes and to measure the performance of the algorithm. I used the well known MNIST pattern database. The dataset represented ten thousand patterns of handwritten digits. The Tesla multi core platform which I used to execute the algorithm, managed to reduce the time of one epoch of the batch learning process to 630 milliseconds, which is a very satisfying result. Overall error minimization to 0.04 was achieved in average time 167 seconds. Such a result is comfortably applicable not only in the research but also in the real solutions.

With utilizing the GPU to execute the algorithm, the solution is also much affordable, because even the high end GPU devices are significantly cheaper, in comparison to cluster CPU devices.

The results proved that there is definitely a solution of accelerating the neural network learning algorithms using parallelization on GPU which can be used in various applications.

Stanislav Toman

9 References

[1] Bishop M. Christopher, Neural Networks for Pattern Rrecognition, Clarendon Press Oxford, 1995

[2] Jeff Heaton, Introduction to Neural Networks with C #, Heaton Research Inc., 2008

[3] Ben Krose, Patrick Van der Smagt, An Introduction to Neural Networks, The University of Amsterdam, 1996

[4] prof. Ing. Ivo Vondrák CSc., Ing. Dušan Fedorčák Neuronové síte, Katedra Informatiky VŠB - Technická univerzita Ostrava

[5] Mariusz Bernacki, Przemyslaw Wlodarczyk, Principles of training multi-layer neural network using backpropagation, Akademia Górniczo-Hutnicza im. Stanislawa Staszica Krakow, 2004

[6] Samrasinghe Sandhya, Neural Networks for Applied Sciences and Eengineering, Auerbach Publications, 2006

[7] Macek Rudolf, Hardware Accelerated Computational Intelligence, Faculty of Electrical Engineering of Czech Technical University Prague, 2008

[8] Sheetal Lahabar, Pinky Agrawal, P J Narayanan, High Performance Pattern Recognition on GPU, Centre for Visual Information Technology International Institute of Information Technology Hyderabad, 2008

[9] Petrowski A., Choosing Among Several Parallel Implementations of the Backpropagation Algorithm, Departmen Informatique Institut nationals des Telecommunications France,1994

[10] Chris Oei, Gerald Friedland, Adam Janin, Parallel Training of a Multi-Layer Perceptron on a GPU, International Computer Science Institute, 2009

[11] Eckel Bruce, Thinking in C++, MindView, Inc., 2000

[12] Jim Torresen and Shinji Tomita. A Review of Parallel Implementations of Backpropagation Neural Networks. Chapter 2 in Parallel Architectures for Artificial Neural Networks, IEEE CS Press, 1998

[13] Bernardete Ribeiro Lopez Noel, GPU Implementation Of Multiple Backpropagation Algorithm, Center of Informatics and Systems of University of Coimbra Portugal, 2009

[14] Sincak Peter, Neurónové Siete - Inžiniersky Prístup, Technická Univerzita Košice, 1996

[15] NVIDIA Corporation, TESLA C2050 AND TESLA C2070 COMPUTING PROCESSOR BOARD - Board Specification, NVIDIA Corporation, 2010

[16] NVIDIA Corporation, CUDA C Programming guide, NVIDIA Corporation, 2010

[17] NVIDIA Corporation, CUDA-GDB Debugger User Manual, NVIDIA Corporation, 2010

[18] NVIDIA Corporation, CUDA Reference Manual, NVIDIA Corporation, 2010

[19] NVIDIA Corporation, CUDA Best Practices Guide, NVIDIA Corporation, 2010

[20] Jason Sanders, Edward Kandrot, CUDA by Example, NVIDIA Corporation, 2011

[21] Nissen Steffen, The NaturalDocs documentation for the FANN C library and the C++ wrapper in HTML form 2010

[22] Lopez Roberto, Flood: An Open Source Neural Networks C++ Library, International Center for Numerical Methods in Engineering, 2010

[23] Heaton Research, Inc., Introduction to Encog 2.5 for C#, Heaton Research, Inc. 2010

[24] NeuroDimension, Inc., NeuroSolutions Getting Started Manual, NeuroDimension, Inc. 2010

[25] Stutgart Neural Network Simulator, SNNS User Manual, University of Stuttgart, 2010

[26] Cybenko George, Approximations by superpositions of sigmoidal functions. Mathematics of Control, Signals, and Systems, Springer-Verlag New York, 1989

A Content of included compact disc

- Digital form of this thesis with source code
- Source code of the implementation
- Excel sheets with testing results