Parallel Training of Large Scale Neural Networks: Performance Analysis & Prediction

September 18, 2012

Author: R.P.M. van Doormaal
Supervisor: Prof. Dr. H. Corporaal
Tutor: Ir. M.C.J. Peemen
Abstract

Large scale neural networks, and Convolutional Neural Networks (CNNs) in particular, are a very promising solution for complicated vision applications such as face detection and object classification. CNNs can perform very robust classification under varying circumstances such as lighting conditions, location invariance and small distortions. Their accuracy on these vision applications is comparable to hand-made classification systems, which require a big design effort. CNNs solve such problems by training on big datasets with many examples it may encounter and the desired response it should give during execution. This ensures the robustness of the system during execution and simplifies the design process by integrating most normalization steps in one automatically tuned classifier.

The downside is that training such CNNs takes a very long time, and accelerating the training process by parallelization is non-trivial. For example the current training procedure for a real-world vision task takes several days of processing on a high-end PC. This work introduces a parallel implementation for both the CPU and GPU platform which is able to achieve a speedup 5X, reducing the training time for a speedsign recognition network from 2 days to under 10 hours.

Research on neural network training indicates that performing the training on a GPU platform can improve the training speed over training on a regular CPU platform. This work shows the trade-offs between the GPU and CPU platform and shows the network parameters decide which platform is superior for the training of large scale neural networks.

This work focuses on optimizing the training process and understanding & evaluating trade-offs to select the optimal combination of factors to minimize the training time of large scale neural networks. Performance prediction models are introduced and a design space exploration is performed using different platforms and neural networks. Results of this design space exploration show that choosing the correct platform can increase the training performance 2-4X, and in the same way the choice of the neural network can have a big impact on the training performance.

Besides the network structure and target architecture the learning algorithm also plays a big role in the eventual performance of the network training. Finally the online and batch learning algorithms are compared. Since batch introduces additional parallelism in the training process this could have a positive effect on the network training. This work shows in contrary to published work that with an honest comparison between these algorithms there is no advantage when using batch learning over using online learning for the investigated platforms.
## Contents

1 Introducing neural networks .................................................. 3  
1.1 What are neural networks? .................................................... 3  
1.2 Applications of neural networks .......................................... 4  
1.2.1 Real-life applications .................................................... 4  
1.2.2 Studied applications ...................................................... 5  
1.3 From biological to artificial neural network ............................. 6  

2 Neural network training ........................................................... 8  
2.1 The learning algorithm ....................................................... 8  
2.1.1 Unsupervised learning .................................................... 8  
2.1.2 Supervised learning ...................................................... 8  
2.1.3 The back-propagation algorithm ...................................... 9  
2.2 Training the studied applications ....................................... 10  
2.2.1 Profiling Results .......................................................... 10  
2.2.2 Parallel training results ............................................... 11  
2.3 Problem Description .......................................................... 14  

3 Performance Modeling ............................................................ 15  
3.1 Introducing the roofline model ............................................ 15  
3.2 Composing the roofline model ............................................. 16  
3.2.1 Computational roofline .................................................. 16  
3.2.2 Memory bandwidth roofline .......................................... 17  
3.2.3 Operational intensity .................................................... 17  
3.2.4 Computational performance .......................................... 18  
3.3 Applying the roofline model .............................................. 18  
3.3.1 Determining the theoretical roofline ................................ 18  
3.3.2 Determining the practical roofline .................................. 20  
3.3.3 Performance of the training application ............................ 23  
3.4 Result evaluation .............................................................. 24  
3.4.1 CPU bottlenecks .......................................................... 24  
3.4.2 GPU bottlenecks .......................................................... 25  
3.4.3 Concluding remarks ..................................................... 27  

4 CPU performance prediction of CNN training ............................ 28  
4.1 Deriving the CPU performance prediction model ...................... 28  
4.2 Forward convolution model ................................................ 29  
4.3 Forward convolution micro benchmark kernel .......................... 30  
4.4 Analyzing the forward convolution ...................................... 32  
4.4.1 The innermost loop ‘l’ ................................................... 32  
4.4.2 Adding the ‘k’ loop ...................................................... 33  
4.4.3 Adding the rest of the run convolution layer function .......... 34  
4.4.4 The compute_local_gradients and update_weights functions .. 35
Chapter 1

Introducing neural networks

1.1 What are neural networks?

Neural networks are basically models of the brain and nervous system and are an approach to machine learning, where the goal is to allow computers to improve their performance over time by evolving its behavior. The core objective here is to generalize from what the network has experienced before. Next to Artificial Neural Networks (ANNs) there are other machine learning approaches such as genetic programming or decision tree learning, but these approaches are not the focus of this research.

An illustrative example are the experiments of Wanatabe et al. titled ‘Pigeons as art experts’. [26] In this work pigeons were placed inside a Skinner box and were presented two paintings of Van Gogh and Chagall (see Figure 1.1). A conditional experiment was performed on the pigeons where they were rewarded with food for pecking when presented with a painting of a certain artist.

After a period of training the pigeons were able to discriminate between Van Gogh and Chagall with 95% accuracy when presented with one of the two paintings shown in Figure 1.1. However the discrimination was still 85% when the pigeons were presented with another unseen painting from one of these artists. It seems that the pigeons were able to acquire knowledge about art by not only memorizing the paintings, but extract and recognize patterns of the paintings to be able to generalize from what they have seen and make predictions.

This exactly the behavior that is desired from an ANN, which is a mathematical or computational model inspired by the structure and functional aspects of biological neural networks. ANNs are models consisting of several nodes interconnected by a configurable network. This network is configured by weights defining the importance of a particular connection. These weights can be adjusted to change the behavior of the network. By modifying the weights the network is able to learn a desired behavior, just like the pigeons in the previous example.

ANNs are basically a different way of computing when compared to conventional computers.

Figure 1.1: Impression of the experiments in the work of Wanatabe et al.
Where ANNs are based on the massively parallel architecture of the biological neural system, conventional computers are based on the processing and memory abstraction of human processing. ANNs are used for diverse applications, and they are especially suitable for applications where it is difficult to state explicit rules and more efficient to learn the desired behavior by training on a set of examples.

This work mostly focuses on Convolutional Neural Networks (CNNs). CNNs are deep neural network architectures that are widely used for classification or recognition tasks on images. Their performance is very competitive compared to the alternatives which are generally a tuned pipeline of algorithms such as lighting correction, color thresholding, edge and corner detection or shape recognition. CNNs are inspired by human visual perception and are optimized for two-dimensional pattern recognition by using shared network weights and fewer connections. These constraints reduce the solution space of CNNs, which makes them easier to train than fully-connected deep networks. An example convolutional neural network used for hand-written character recognition is shown in Figure 1.2.

1.2 Applications of neural networks

There are a lot of real-life applications of (convolutional) neural networks. Some application fields will be described in Section 1.2.1 and the applications that are used in this work to study large scale neural networks will be described in Section 1.2.2.

1.2.1 Real-life applications

(a) Security  (b) Industrial  (c) Medical  (d) Automotive  (e) Financial

Figure 1.3: Examples of application fields for (convolutional) neural networks

Neural networks are widely used in real-life applications that have a high data or task complexity that makes it impractical to classify by hand, and an automatic classifier such as a neural network is required. Some example applications are shown in Figure 1.3. Neural networks can
for example perform face recognition in the security domain, component analysis for industrial purposes, detection and evaluation of medical phenomena such as cancer, speed sign detection and recognition in the automotive domain or stock market prediction and fraud detection in the financial domain.

1.2.2 Studied applications

There are two applications that are tested for the GPU and CPU platforms. The first one is a convolutional neural network for speed sign recognition, developed within the faculty of Electrical Engineering at the Eindhoven University of Technology. An overview of the convolutional neural network for speed sign recognition can be seen in Figure 1.4.

The second application is a convolutional neural network for the classification of the small-NORB stereoscopic object dataset. This application has bigger input images (96x96 pixels opposed to 32x32 pixels for the speed sign recognition application) and bigger feature maps within the network. The dataset contains 50 different objects (25 for training and 25 for testing) which are equally distributed over five classes. These objects are shown from different angles with different lighting conditions (see Figure 1.5). The CNN used for this application is similar to the network shown in Figure 1.4, but the size of the feature maps, the number of feature maps and the connection pattern is slightly different.

The training of these networks takes around 2 (speed sign recognition) and 5 days (small-NORB) for a single network configuration on a high-end CPU, so a performance increase for the training is highly desirable to be able to perform design-space exploration or apply generalization techniques like bootstrapping or cross-validation.

Figure 1.4: CNN for speed sign recognition

Figure 1.5: Subset of the visual patterns used for training with small-NORB dataset
1.3 From biological to artificial neural network

Now applications of artificial neural networks have been treated their origin will be elaborated. The human brain contains billions of neural cells that each act like a simple processing element. In comparison artificial neural networks consist of artificial nodes that emulate the properties of biologic neurons. The biological neuron is a cell that transmits information by electrical and chemical signaling. This process is highly complex and the exact working of this is not very relevant to the understanding of the way biological neurons operate. When looking at the neuron at a higher level (abstracting from the exact operation of the chemical processes inside the neuron) the biological neuron it consists of:

- A cell body (acting as the processing node)
- Axons responsible for transmitting signals to other neurons (acting as the output of the neuron)
- Dendrites which carry signals from other neurons to the cell body (acting as the input of the neuron)
- Synapses that connects the axon of one neuron to the dendrite of another neuron

The artificial neuron is inspired by this abstracted model of the biological neuron and serves as basis for every neural network. Both models can be seen in Figure 1.6, where the biological neuron is shown on the left and the artificial neuron on the right. From this figure it can seen that the basic structure of both models is very similar, and the artificial neuron mainly mimics the information flow of the biological neuron. This information flow in biological neurons can be described as:

- Outputs from connected neurons are received at the dendrites through synapses (specialized membranes that transfer signals from one neuron to the next by diffusion of neurotransmitters) and serve as input for the neuron
- The inputs from other neurons are each multiplied with a weight corresponding to the synaptic efficiency of the synapse of the connection
- The signals are then processed into an output of the neuron in the cell body, aggregating the inputs and applying a non-linear activation function
- The output travels along the axon of the neuron, after which the same steps repeat for the connected neurons
Mathematically the artificial neuron can be represented as in Equation 1.3.

\[
v = \sum_{i=0}^{m} w_i x_i \quad (1.1)
\]

\[
y = \varphi(v) \quad (1.2)
\]

\[
y = \varphi\left(\sum_{i=0}^{m} w_i x_i\right) \quad (1.3)
\]

In this equation \(y\) represents the output of the neuron, \(\varphi\) represents the applied activation function (e.g. linear function, threshold function, sigmoidal function) and \(w_0...w_m\) represent the weight connections for the inputs \(x_0...x_m\), where \(x_0\) represents the bias node which is fixed at a value of 1 and \(w_0\) is the weight of the connection of the bias node to the neuron.
Chapter 2

Neural network training

Now the fundamentals and applications of neural networks are clear the focus will shift to the real problem, namely the network training. It is already mentioned in Chapter 1 that the training process takes a long time, especially for large scale networks.

Take for example the speed sign detection CNN from [19]. This network requires 2 days of training on a high end PC, and exploring the design space for the network simply requires too much training time to be beneficial. Because of this long training time the network topology used in most state-of-the-art neural network applications is chosen based on previous successful networks for similar tasks instead of based on a thought-out exploration of the design space.

In this chapter the typically used back-propagation algorithm for network training will be explained in detail. After that the steps taken to optimize the training process for the CPU and GPU platform will be explained, after which the problem statement will be sketched for this work.

2.1 The learning algorithm

There are two main learning strategies for neural networks: supervised learning and unsupervised learning.

2.1.1 Unsupervised learning

Unsupervised learning methods train networks with training data consisting of unlabeled examples (no desired output of the network is provided with the input pattern) which are fed to the network for training. Unsupervised learning methods can detect patterns in the input, and nodes in the network become associated with different patterns in the input data.

Unsupervised learning methods are mostly used for applications where for a certain input the output that the network should produce is unknown. The disadvantage of unsupervised learning methods is that they operate very local within network layers (e.g. neighbor neurons affect each other), and therefore are not suited for learning specific tasks but rather identify correlations in the input data.

For training deep networks the supervised learning is generally applied because the network has to learn a specific task, which is not possible to achieve using solely unsupervised learning. It is possible to use a combination of supervised and unsupervised learning for training a network. For example in the first layers an unsupervised method is used to detect correlations in the input, and a supervised method on the output of these layers is used.

2.1.2 Supervised learning

Supervised learning methods train networks with training data consisting of labeled examples, where a training vector contains the input data and the desired output (or supervisory signal) of the network. The output obtained from the network is observed, and the deviation of the network
output from the desired output is measured by a cost function. The network weights are adjusted according to the magnitude of the output error as defined by the learning algorithm.

There are a lot different learning algorithms that operate in a supervised manner among which stochastic learning, competitive learning or gradient descent learning. With stochastic learning where the weights of the network are adjusted in a probabilistic fashion. With competitive learning (or winners-take-all) neurons in a layer compete and the neuron responding the strongest to the input undergoes weight adjustment. And gradient descent learning calculates the gradient of the error function with respect to every weight in the network. This gradient function indicates how a small change in a certain weight will affect the overall error of the network.

There are different learning algorithms that apply the gradient descent approach, but the most popular learning algorithm that applies the gradient descent is the back-propagation algorithm, which is able to train multi-layer neural networks with non-linear activation functions.

2.1.3 The back-propagation algorithm

The back-propagation algorithm trains a network using gradient descent and can be described in 5 steps [9]. The variable $n$ indicates the current iteration of the training procedure. This variable is only important when updating the weights in step 4 because the weights of the next iteration are adjusted based on results of the current iteration. In all other steps of the algorithm the calculations concern a single iteration, so for simplicity the $n$ will be left out here.

1. Initialization Network weights are randomly picked from a uniform distribution whose mean is zero

2. Presentations of Training Examples Present the network with one (online learning) or multiple ((mini-)batch learning) input vectors, randomly picked from the training data. For each training vector perform the forward and backward computations in point 3 and 4 respectively.

3. Forward Computation Let a training vector be denoted by the tuple $(x(n), d(n))$ where $x(n)$ is the input vector and $d(n)$ is the desired response vector. Compute the induced local fields and function signals of the network by proceeding forward through the network, layer by layer. The neuron potential $v^{(l)}_{j}$ for neuron $j$ in layer $l$ is:

$$v^{(l)}_{j} = \sum_{i} w^{(l)}_{ji} y^{(l-1)}_{i}$$

(2.1)

Here $y^{(l-1)}_{i}$ is the output signal of neuron $i$ in the previous layer $l-1$ at iteration $n$, and $w^{(l)}_{ji}$ is the synaptic weight of neuron $j$ in layer $l$ that is fed from neuron $i$ in layer $l-1$. For $i = 0$ it holds that $y^{(l-1)}_{i} = 1$, and $w^{(l)}_{j0} = b^{(l)}_{j}$ is the bias applied to neuron $j$ in layer $l$. The activation function of the neurons has to be differentiable, because the weight updates of the network depend on the gradient of the output error. Assuming the activation function used is a (simplified) sigmoid function, defined as shown in Equation 2.2.

$$\varphi(x) = \frac{1}{1 + e^{-x}}$$

(2.2)

Since the output of the sigmoid function has asymptotes at 0 and 1, the derivative of the sigmoid approaches zero when the output of the neuron goes to 0 or 1. Using the sigmoid as activation function, the output function of neuron $j$ in layer $l$ becomes:

$$y^{(l)}_{j} = \varphi_{j}(v_{j})$$

(2.3)

Where:
Here $x_j$ is the $j$th element of the input vector $x$, $L$ is the number of layers (depth) of the network and $o_j$ is the output vector of the network. The error signal is computed as follows, which can then be propagated back through the network to adjust the weights:

$$e_j = d_j - o_j$$  \hspace{1cm} (2.6)

Here $d_j$ is an element the desired response vector $d$ from the training data.

4. **Backward Computation** Backward computation means computing the local gradients $\delta$ of the network, defined by:

$$\delta_{j}^{(l)}(n) = \begin{cases} 
\epsilon_{j}^{(L)}(n)\phi'_{j}(v_{j}^{(L)}(n)) & \text{for neuron } j \text{ in output layer } L \\
\phi'_{j}(v_{j}^{(l)}(n)) \sum_{k} \delta_{k}^{(l+1)}(n)w_{kj}^{(l+1)}(n) & \text{for neuron } j \text{ in hidden layer } l
\end{cases}$$  \hspace{1cm} (2.7)

Where the prime in $\phi'_{j}(\cdot)$ denotes differentiation with respect to the argument. When the local gradients of the network are calculated the weights can be updated. The weights in layer $l$ are adjusted according to the delta rule:

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \eta \delta_{j}^{(l)}(n)y_{i}^{(l-1)}(n)$$  \hspace{1cm} (2.8)

5. **Iteration** Iterate the forward and backward computations under point 3 and 4 respectively by presenting new input vectors to the network until the stopping criteria (e.g. when the error of the output is low enough) is met.

### 2.2 Training the studied applications

To train convolutional neural networks the standard back-propagation algorithm is used as explained in the previous section, with some minor adjustments to cope with convolutional networks with shared weights and reduced connections. In this section the applications used in this work, the speed sign recognition network and small-NORB object recognition network, will be profiled first and then the optimized results will be presented for the CPU and GPU platform.

#### 2.2.1 Profiling Results

To summarize, the back-propagation is split up in several steps as can be seen in Tables 2.1 and 2.2. In these tables the CPU results of the training of the speed sign recognition and small-NORB application are shown.

The **Rest** category consists of the overhead of the application like reading the configuration files and converting the input data to floating point values. The **Rest** category contains several functions that are called to initialize the network, weights and input data. This category seems to use a lot of the execution time because the data set for these tests is very small. Once the data set used to train the network becomes larger the execution times of the functions in the **Rest** category will remain the same while the execution time of the other functions will increase with the data set. At this point the **Rest** category is not important for optimization anymore, consuming less than one percent of the execution time of the application.

When looking at the profiling information of the applications, the most important candidate for performance improvement is the **run_convolution_layer** function, since it consumes most
application function | execution time [ms] | execution time (%)
--- | --- | ---
run convolution layer | 0.68 | 38
compute local gradients | 0.15 | 8
update weights | 0.28 | 16
compute error | 0.01 | 0
rest | 0.68 | 38
**total** | 1.80 | 100

Table 2.1: Profiling results of the speed sign recognition application

application function | execution time [ms] | execution time (%)
--- | --- | ---
run convolution layer | 2.03 | 30
compute local gradients | 0.46 | 7
update weights | 1.25 | 19
compute error | 0.01 | 0
rest | 3.01 | 45
**total** | 6.75 | 100

Table 2.2: Profiling results of the small-NORB application

execution time for one image. Next to this, the `update_weights` function is a good candidate as well, consuming 16-19% of the execution time for a single image with both applications. The `compute_local_gradients` function is not the most time-consuming function to start with, but it is the most complex function and will be most difficult to efficiently execute in parallel.

### 2.2.2 Parallel training results
As part of the preparation project and the graduation project the implementations of the small-NORB and speed sign recognition training applications for both the CPU and GPU platform were optimized to reduce the training time. Several algorithmic changes and optimizations were made which are further explained in the preparation project report to optimize the speed of the neural network training.

For example the use of SSE instructions for the CPU required algorithmic changes like aligning the memory footprint of the application to ensure aligned memory access for the SSE vectors in all cases.

Another example in is that the computation of the local gradients (which propagates the error backwards through the network) multiple neurons in layer $l+1$ can have a connection to the same neuron in layer $l$ causing them to write to the same memory address at the same time. This method of calculating the local gradients also has the disadvantage that each layer needs to have knowledge of the subsequent layer, i.e. it needs to know what weights and neurons are involved in the gradient calculation. This dependency prevents parallel execution over the feature maps in layer $l+1$, since individual processes would write to the same address of a neuron in layer $l$. This could be solved in two ways: ‘Pushing’ the local gradients to layer $l$ (Figure 2.1b) instead of ‘Pulling’ them from layer $l+1$ (Figure 2.1a), or ‘Pulling’ the local gradients (like the standard back-propagation algorithm) but preventing simultaneous memory accesses at the same memory address.

For both the CPU and GPU implementation the ‘Pulling’ strategy is applied since it rendered faster execution for the local gradient computation. This strategy exploits the parallelism between feature maps instead of parallelism between neurons within a feature map.

The final results of the optimized applications for CPU and GPU will now be elaborated. First the speedup of the most compute-intensive functions of the application and finally a long-term test is done with 100 epochs of training on the full input data set to see if the measured speedup...
Figure 2.1: Pushing versus pulling the error signals between network layers

at function level is actually present when training the network.

The results for the most compute-intensive functions of the CNN training application are shown in Table 2.3 and 2.4.

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU ref [ms]</th>
<th>GPU [ms]</th>
<th>CPU Optimized [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>run convolution layer</td>
<td>0.68</td>
<td>0.14</td>
<td>0.06</td>
</tr>
<tr>
<td>compute local gradients</td>
<td>0.15</td>
<td>0.13</td>
<td>0.11</td>
</tr>
<tr>
<td>update weights</td>
<td>0.28</td>
<td>0.14</td>
<td>0.09</td>
</tr>
<tr>
<td>compute error</td>
<td>0.01</td>
<td>0.10</td>
<td>0.01</td>
</tr>
<tr>
<td>Total</td>
<td>1.12</td>
<td>0.51</td>
<td>0.27</td>
</tr>
</tbody>
</table>

Table 2.3: Optimization results for the speed sign recognition application

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU ref [ms]</th>
<th>GPU [ms]</th>
<th>CPU Optimized [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>run convolution layer</td>
<td>2.03</td>
<td>0.19</td>
<td>0.16</td>
</tr>
<tr>
<td>compute local gradients</td>
<td>0.46</td>
<td>0.15</td>
<td>0.32</td>
</tr>
<tr>
<td>update_weights</td>
<td>1.25</td>
<td>0.25</td>
<td>0.32</td>
</tr>
<tr>
<td>compute error</td>
<td>0.01</td>
<td>0.11</td>
<td>0.01</td>
</tr>
<tr>
<td>Total</td>
<td>3.75</td>
<td>0.70</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Table 2.4: Optimization results for the small-NORB application
Table 2.5 shows the results of the long test of 100 epochs for the training algorithm to confirm that the function speedups shown in Table 2.3 and 2.4 are actually present when training the CNN for a large amount of epochs.

<table>
<thead>
<tr>
<th>Network</th>
<th>Implementation</th>
<th>Training Time</th>
<th>Expected Speedup</th>
<th>Achieved Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>speed sign</td>
<td>CPU ref</td>
<td>9m34s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>CPU opt</td>
<td>1m53s</td>
<td>5.45X</td>
<td>5.09X</td>
</tr>
<tr>
<td></td>
<td>GPU</td>
<td>4m28s</td>
<td>2.77X</td>
<td>2.42X</td>
</tr>
<tr>
<td>small-NORB</td>
<td>CPU ref</td>
<td>3h22m20s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>CPU opt</td>
<td>37m41s</td>
<td>5.96X</td>
<td>5.37X</td>
</tr>
<tr>
<td></td>
<td>GPU</td>
<td>35m18s</td>
<td>6.49X</td>
<td>5.73X</td>
</tr>
</tbody>
</table>

Table 2.5: Training time comparison of 100 epochs of training

The results from Table 2.5 show that the optimized implementations on both the CPU and GPU incur a big time reduction for the training of 100 epochs for both neural networks. The column with expected speedup values is based on the function speedup shown in Table 2.3 and 2.4, and the achieved speedup values are measured during a test run of 100 epochs of the network. The results are more or less what is expected by the results from Table 2.3 and 2.4 which is shown in the expected speedup column of Table 2.5. There is a deviation between the expected speedup and the actual speedup, which lies around 10%. This is caused by the fact that the function that prepares the input images to be fed to the neural network is not taken into account since it only consumed 1% of the execution time of the reference implementation on the CPU. However in the optimized implementations for CPU and GPU the other functions are faster, making this preparation function more significant which uses about 10% of the execution time in these implementations and closing the gap to the speedup that was expected.

The results in Table 2.3 and 2.4 show that there is a trade-off between using a CPU or GPU for the training of a convolutional neural network. The choice is influenced by the amount of parallelism in the neural network because this determines how much performance can be won by parallel execution. When for example a network contains large feature maps like in the small-NORB application, the GPU is able to exploit more parallelism in the training process and is slightly faster than the CPU implementation (see Table 2.4). However when the amount of parallelism in the network reduces the CPU implementation quickly performs better than the GPU with respect to the training time, like with the speed sign recognition application where the CPU implementation is more than two times faster than the GPU implementation (see Table 2.3).

When a more recent CPU is used that is able to make use of AVX instructions (where the registers for executing operations in an SIMD fashion like with the SSE instructions are 256 bits wide instead of 128 bits) another performance increase for the CPU implementation is expected when there is parallelism available within feature maps like with the small-NORB application. The expectation is that the CPU implementation will be faster than the GPU implementation for this network as well as the speed sign recognition network. Using a GPU for the training would then only be profitable when training networks with large feature maps, even bigger than those of the small-NORB network.

To explore this trade-off between using a CPU or GPU for the training of a convolutional neural network, and to be able to choose the correct platform with respect to a certain network it is necessary to be able to predict the performance of the training on a certain platform. As mentioned the performance of a platform is directly related to the amount of parallelism in the network that is to be trained. Therefore a model for both the CPU and GPU has to be formulated to be able to predict the performance on the respective platforms. The starting point of this performance predictive model is the roofline model [28], which is often used to predict performance of simple computational kernels for both CPU and GPU.
2.3 Problem Description

The previous section has showed that the parallelization of the training process can already provide a great improvement over the standard training procedure on CPU. For example by these optimizations the training time of the speed sign recognition and detection network has reduced from two days to less than 10 hours. However it also showed there is a big trade-off space between the network type, the target platform and the learning algorithm for the training performance. This is shown in Figure 2.2 where a combination between the neural network structure and target platform can influence the performance of the training of the network to great extent. The used learning method also influences the eventual training performance, for example because some algorithms have more opportunities to exploit parallelism than others. Choosing an optimal combination of these factors is important for effective and fast network training. Understanding these trade-offs and exploring them is the goal of this research.

It can be the case for example that you want to find the optimal platform to train a specific network, or you want to find an optimal network structure for a fixed platform. In an early design stage it can even be the case that an optimal training application has to be designed and the network and platform can be chosen freely according to the optimal combination.

![Figure 2.2: Trade-off space for the neural network training application](image)

Chapter three to five elaborate on performance prediction of networks on different target architectures, focusing on CPU and GPU platforms to be able to clarify training performance on different target architectures. In order to do this a performance model has to be utilized that is able to achieve accurate precision (with an accuracy of 25% of the actual performance to be able to explore the design space based on reliable predictions). Besides this it is desirable that the model can be executed as fast as possible in order to explore a realistic design space in reasonable time. The model should also be platform independent to be able to explore and compare performance on different platforms.

Chapter six utilizes these results to gain insight on the trade-off between network variations and different platforms. Chapter seven finally evaluates an alternative algorithm for training the networks, namely the batch method. Other alternative learning algorithms were also researched in the preparation of this project. Genetic algorithms for network training, particle swarm optimization and differential evolution, were studied. Even though they showed promising results with experiments on small networks, they turned out to be very inefficient for the training of large scale networks. Therefore only the online and batch algorithms will be compared in this report.
Chapter 3

Performance Modeling

To explore the trade-off described in Section 2.3 a performance model is required to be able to estimate the performance of the training application on a target platform. As mentioned it should be very accurate for reliable results, have a short execution time and be as simple as possible to maintain insight and transparency. This means that factors that only have a small impact on the prediction should not be taken into account since they only complicate the model and increase the execution time.

The roofline model introduced by Williams et al. [28] is a performance model designed to understand and analyze performance bottlenecks. The roofline model is very simplistic, meaning that the model is transparent and executes very fast. On top of this it can be applied to both the CPU and GPU architectures. This could be a very good candidate for the performance estimation of the neural network training process.

This chapter first introduces the roofline model and how it can be composed, after which it will be applied to the training application in Section 3.3 after which the results will be discussed.

3.1 Introducing the roofline model

The last decade the focus of the processor field has shifted from increasing the performance of a single high frequency processor to integrating multiple processors on a single chip, introducing a parallel computing era where multi- and many-core systems on chip are expected to play an important role in the coming years. Parallelism will become increasingly important to increase the performance of computer architectures. Many-core architectures such as the Graphics Processor Unit (GPU) are suited to execute a class of applications where other applications are more suited for execution on a multi-core processor. This introduces a heterogeneous computing environment where such multi-core processors can be integrated with a many-core processor in a single system or on a single chip. The performance evaluation of certain applications on such architectures becomes increasingly difficult due to the increasing amount of factors that need to be taken into account when using a multi- or many-core architecture.

The roofline model gives an estimation of the performance of an application based on the assumption that performance is limited by either the bandwidth between the off-chip memory and the processor or by peak computational performance of the architecture. This also makes the model processor specific since these properties differ per processor. The roofline model was initially designed to estimate CPU performance, but can easily be applied to a GPU since it presumes a simple architectural model consisting of black boxed computational elements (e.g. CPUs, cores, function units, etcetera) and memory elements (e.g. DRAM, caches, register files) interconnected by a network.

The visual representation of the roofline model is depicted in Figure 3.1. It consists of two roofs, namely a memory roof limited by the bandwidth to the external memory and computational roof limited by the maximum computational performance of the processor. Next to these roofs there
can be multiple ceilings that can bound the performance if the processor or memory bandwidth is not used at its full potential. These ceilings can be crossed by an application by efficiently using the processor or by bounding the memory traffic. There are also two example applications shown in the roofline model. The first application has an operational intensity of 0.9 and its performance is bounded by the memory bandwidth between the processor and the off-chip memory. The second application has an operational intensity of 10 and is bounded by the computational performance of the processor.

![Figure 3.1: Visual representation of the roofline model](image)

The roofline model considers an application kernel as a movement of data from one or more memories to a processor where it can be buffered, duplicated or used for computation. The modified data or new data is then written back to those memories. The movement of data between memory and the processor is bounded by the characteristics of the interconnect between the processor and the memory. The memory traffic of interest is that between the processor and the off-chip memory since that has the greatest latency and is most likely to form a bottleneck in the data traffic from and to the processor. For example when certain data is moved from the off-chip memory to a processor it is assumed the caches of the architecture make sure this data is kept local if it is re-used (unless there is a great amount of data used in the application that could cause the data to be overwritten in any layer of the cache hierarchy). The maximum computational performance of architecture consists of single precision floating-point operations (such as addition, subtraction, multiplication or compare).

### 3.2 Composing the roofline model

Unlike the model itself the parameters of the rooflines itself are architecture dependent. This means the internal architecture of the platform is analyzed to determine the computational and memory bandwidth rooflines. Using analytical equations the theoretical rooflines can be determined. [23] It is also possible to determine a practical roofline for a certain architecture, for example using the STREAM benchmark to determine the maximum practical bandwidth to the off-chip memory. [16]

#### 3.2.1 Computational roofline

Equation 3.1 can be used to determine the computational roofline of an architecture. Here $C_{\text{roof}}$ represents the computational roofline of the architecture, or the absolute peak computational performance. This can be calculated by multiplying the clock frequency of the cores ($C_f$), the number of cores ($\#\text{cores}$) and the number floating point operations that can be executed per core.
per clock cycle ($C_{\text{ops}}$). $C_{\text{ops}}$ takes multiple function units per core and sub-word parallelism into account and is calculated by summing the number of function units and their respected width (see Equation 3.2), where $q$ represents the number of function units per core.

$$C_{\text{roof}} = \#\text{cores} \cdot C_t \cdot C_{\text{ops}} \quad [\text{ops/sec}] \quad (3.1)$$

$$C_{\text{ops}} = \sum_{0}^{q-1} FU_{\text{width}} \quad (3.2)$$

### 3.2.2 Memory bandwidth roofline

The theoretical memory bandwidth roofline $B_{\text{roof}}$ can be calculated using Equation 3.3, where $M_f$ is the memory clock speed at which a memory unit issues transfers, $M_{dt}$ is the number of data transfers per clock cycle, $M_{\text{width}}$ defines the width in bytes of a single memory transfer and $\#\text{channels}$ is the number of memory channels.

$$B_{\text{roof}} = M_f \cdot M_{dt} \cdot M_{\text{width}} \cdot \#\text{channels} \quad [\text{bytes/sec}] \quad (3.3)$$

Using equations 3.1 to 3.3 and the information of the regarded architecture the outline for the roofline model (as shown in Figure 3.1) can be made.

### 3.2.3 Operational intensity

The operational intensity of a kernel is expressed in floating point operations per byte ([flops/byte]). A high operational intensity infers the kernel is more likely to be bound by the computational roofline of the architecture since it performs a large amount of floating point operations per byte transferred from the off-chip memory. On the other hand a low operational intensity suggests the kernel is likely bound by the memory bandwidth since a lot of data is needed from the off-chip memory for few computations. The computational performance of the architecture is expressed in floating point operations per second ([flops/sec]). The operational intensity of a kernel can be estimated by analyzing the kernel. An example kernel is shown in Listing 3.1, which performs a matrix-matrix multiplication.

```c
C[i,j] = 0.0;
for(i=0; i<N; i++){
    for(j=0; j<N; j++){
        for(k=0; k<N; k++){
            C[i,j] += A[i,k] * B[k,j];
        }
    }
}
```

Listing 3.1: Example kernel performing matrix-matrix multiplication

The operational intensity of the kernel in Listing 3.1 can be estimated by analyzing the kernel. Assuming the matrices contain single precision floating point values and the cache is greater than $12 \cdot N^2$: $A[i,j]$, $B[i,j]$ and $C[i,j]$ can be kept in the cache. If this is the case only their initial reference and the write back reference generate traffic to the off-chip memory. The kernel performs 2 floating point operations (flops) per iteration and $2 \cdot N^3$ flops in total. It also requires that the matrices $A[i,j]$, $B[i,j]$ and $C[i,j]$ are read once, and $C[i,j]$ is written back after the computations are finished. This generates a total traffic of $4 \cdot 4 \cdot N^2 = 16 \cdot N^2$ bytes, resulting in an operational intensity of $N/8$. When the value of $N$ is known the exact operational intensity of the kernel can be determined.
3.2.4 Computational performance

When the operational intensity of the kernel has been estimated, the application can be drawn as vertical line on the roofline model (see Figure 3.1). At this moment it is possible to see the maximal performance on a certain architecture. For example when Application 1 in Figure 3.1 is considered it can be concluded that when the application optimally uses the architecture resources it can hit the memory bandwidth roofline and achieve a performance of around $30 \cdot 10^9$ floating point operations per second. The actual computational performance of the application can be extracted by calculating the number of floating point operations the kernel performs and measuring the time required to execute the kernel. When this is known, the application can be depicted as a point on the roofline graph.

3.3 Applying the roofline model

After a short elaboration on the basis of the roofline model it can be applied to the neural network training application to see if it is a good candidate for performance estimation for such large scale networks.

3.3.1 Determining the theoretical roofline

The theoretical memory bandwidth and computational rooflines can be determined from the information about the platforms that are used to implement the application. These are shown in Figure 3.2 for the CPU and GPU respectively.

![Roofline model of the Intel Core i7-960 and GeForce GTX470](image)

Figure 3.2: Roofline models for the CPU and GPU platform used to implement the CNN application

The CPU roofline contains several computational ceilings. The computational ceilings indicate the use multiple threads, the use of SIMD instructions and the balancing factor of multiply and addition operations when using both of these operations. This last ceiling is important because the Nehalem architecture has 6 instruction ports through which it can perform a floating point multiplication and addition in parallel (see Figure 3.3). If the amount of multiplications and additions is balanced it is possible to perform two floating point operations per cycle. If this ratio is imbalanced, for example if the kernel only contains multiplications, the maximum achievable performance is one floating point operation per cycle.

Assuming the operations are perfectly balanced, using SIMD instructions can theoretically increase the performance by a factor of four since four single precision floating point operations can be executed in parallel in a single cycle. Similarly using multiple threads to utilize all four cores of the CPU can increase the floating point performance with a factor of four, which yields a peak performance (for single precision floating point operations) of 102.4 Gigaflops/sec as shown in Equation 3.4 and 3.5.
Figure 3.3: The Intel Nehalem micro-architecture containing 3 instruction ports for instruction handling and 3 instruction ports to handle load & store operations from the memory.

\[
P_{\text{peak}}^{CPU} = 3.2 \cdot 10^9 \text{ [clock frequency]} \cdot 2 \text{ [flops/cycle]} \cdot 4 \text{ [SIMD width]} \cdot 4 \text{ [#Cores]} \\
= 102.4 \text{ Gigaflops/sec} \tag{3.4}
\]

The roofline model of the GPU is shown in the right of Figure 3.2. The used GPU can be programmed using the CUDA language to construct kernels that execute on the GPU. The kernel is executed for each thread that is defined at launch time. These threads can be grouped in thread blocks, and each thread block is assigned to a stream processor (see Figure 3.4).

\[
P_{\text{peak}}^{GPU} = 1.215 \cdot 10^9 \text{ [core clock frequency]} \cdot 448 \text{ [processing cores]} \cdot 2 \text{ [flops/cycle]} \\
= 1088.6 \text{ Gigaflops/sec} \tag{3.7}
\]

This model contains one ceiling representing the peak computational performance when no
fused multiply-add instructions are used, which is a factor two lower than the computational roofline.

3.3.2 Determining the practical roofline

CPU

In this section several micro benchmark kernels are tested and placed upon the roofline model for the CPU to see if it is possible to reach the theoretical roofline or peak performance. Starting point is the convolution kernel of the training application, which has a lot of nested for-loops the control part of the kernel is very big. Besides the control part there is also a large overhead because of the address calculations that need to be carried out. Technically the floating point data path and control data path can be simultaneously utilized (see Figure 3.3 where the floating point data path uses port 0 & 1 and the control data path port 5), but there probably are still some dependencies within these two paths which prevent the application from hitting the roofline. Besides the address calculations use the same data path as the floating point data path (see Figure 3.3 port 0 & 1) which causes further degradation of the floating-point performance. There is also a sequential confinement, since the addresses must be calculated before the floating point calculations can be carried out.

```c
for(int i=0; i<iterations; i++){
    sum0 += a * b[i];
    sum6 += a * b[i+6];
}
sum = sum0 + sum1 + sum2 + sum3 + sum4 + sum5 + sum6;
```

Listing 3.2: Kernel performing the multiply-accumulate operation with the least possible overhead from other calculations like address calculations and control flow calculations

Listing 3.2 shows a micro benchmark kernel that is able to get very close to the single core ceiling without using SSE instructions by performing the multiply-add operation like in the actual CNN application. Although the operations performed in both kernels are similar, there are still several differences. These are mostly in the amount of data that is needed each iteration from off-chip memory and the overhead from address calculations. This is done to be able to make more efficient use of the available registers and see how close the performance can get to the theoretical ceiling of 6.4 Gigaflops/sec. Next to this benchmark kernel, other benchmark kernels were also implemented that were able to get within 10% of the ceiling when using SSE instructions and when enabling multi-threading. This is useful when trying to analyze the behavior of the CNN application with these features enabled and will help explain why the application is not able to exploit the full performance of the platform.
Listing 3.3: Micro benchmark kernel able to achieve the single core ceiling (without using SSE instructions), performing the multiply-add operation

The kernel can be described as in Listing 3.2, and is manually written in assembly language (Listing 3.3) to map as efficient as possible. The achieved performance is 5.85 Gigaflops/sec, which is still about 9% off the theoretical peak performance of 6.4 Gigaflops/sec. The operational intensity of the kernel in Listing 3.3 can be easily derived. Each iteration one new value of the array "b" has to be read from the off-chip memory (the other five will reside in the level 1 cache). This gives a total of 4 bytes of traffic to the off-chip memory per iteration. The kernel performs 14 floating point operations per iteration, yielding an operational intensity of $\frac{14}{4} = 3.5$. When the kernel is placed on the CPU roofline model of Figure 3.2 (see Figure 3.5) it can be concluded that the memory bandwidth is not the bottleneck in this case. The fact that the kernel is not able to reach the peak performance of 6.4 Gigaflops/sec must be caused by the fact that there is some overhead in the kernel by other computations.

Figure 3.5: Multiply-add micro benchmark kernel in the CPU roofline model
When looking at the generated assembly code by the compiler there are three instructions of overhead each iteration for the loop control. These are the counter increment, the compare for the loop condition and the jump instruction. Assuming these instructions can be carried out in parallel to the floating point operations these cannot cause the gap between the kernel’s performance and the peak performance. This gap can be caused by the fact that the data must be prepared in the XMM registers before the calculations can be carried out. It cannot be verified since it is not possible to see how the instructions are actually scheduled when the kernel is executed, but it is the only explanation for the fact that this kernel cannot reach the single core ceiling without using SSE instructions.

Figure 3.5 shows that when introducing parallelism with enabling SSE instructions and multi-threading the gap to the peak performance increases a little, but the micro benchmark kernel is still able to get to 95% of the theoretical peak computational performance of the CPU. When looking at the bandwidth to off-chip memory that can be achieved in practice (by measuring using the STREAM benchmark [16]) it can be seen that the practical bandwidth is about 50% lower than the theoretical memory bandwidth. The theoretical peak bandwidth to off-chip memory is 32 GB/s (8 GB/s per core) for the used Nehalem architecture.

When running the STREAM benchmark for a single core the bandwidth is confirmed to be around 8 GB/s, however when utilizing multiple cores the full bandwidth cannot be exploited, as noticed. This is probably because the Quick Path Interconnect (QPI) ports have to be utilized to access remote memory which have a lower bandwidth of 12.8 GB/s, affecting the achieved bandwidth in practice (also described in [24]).

**GPU**

Similar as the approach to check the practical roofline for the CPU the practical roofline of the GPU will be determined using micro benchmark kernels to see what the maximum performance is that can be achieved in practice. The micro benchmark kernel looks as in Listing 3.4 which consists of three parts, namely a load, processing and store.

```c
__global__ mbKernel<
unsigned int OPS>
Kernel_Multiply(float* input, float* output)
{
    // Thread index calculation
    unsigned int i = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int j = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int id = i+(j*blockDim.x);

    int static_input = 6; // Declare a static value for the multiply–addition
    float result = input[id]; // Fetch the data

    #pragma unroll
    for (unsigned int i=0; i<OPS ; i++)
    {
        result *= result + static_input; // Perform the computation
    }

    output[id] = result; // Store the result
}
```

Listing 3.4: Micro benchmark kernel for the GPU multiply-add performance

Using the OPS parameter in the micro benchmark kernel the number of operations per byte can be controlled, making it possible to move along the x-axis of the roofline model. When a ratio of higher than 16 operations per byte is used the multiply-addition kernel of Listing 3.4 is able to reach a performance that is about 5% lower than the theoretical roofline (see Figure 3.6). This proves that it is also possible for the GPU implementation to get very close to the maximum theoretical performance of the hardware.
Unlike with the CPU, the GPU is actually able to achieve the theoretical bandwidth to the off-chip memory. This is again measured by using the STREAM benchmark able to measure the memory bandwidth. Therefore it is not necessary to introduce a new ceiling as with the CPU roofline model in Figure 3.5. Note that it is necessary to make use of coalesced loads of data from the external memory to get close to the theoretical maximum performance, with uncoalesced loads the performance is not able to get close to the roofline.

3.3.3 Performance of the training application

CPU

Figure 3.7 shows the CPU roofline models for the training application for both the small-NORB and speed sign application.

When looking at the roofline models in Figure 3.7 it can be concluded that the convolutional neural network training is not using the CPU efficiently. For example when looking at the small-NORB roofline it can concluded that the update_weights function performs at a factor of 16-32 times the theoretical peak performance of the CPU. The main problem is that it is unclear where these gaps are coming from. The main problem is that the roofline model only takes the memory bandwidth to off-chip memory and the maximum floating point performance of the cores into account. From the results it can be concluded that there is another bottleneck or multiple bottlenecks that the roofline model does not take into account which prevent the training application to fully utilize the capacity of the CPU.
Figure 3.8 shows the GPU roofline models for the training application for both the small-NORB and speed sign application.

The roofline models in Figure 3.8 look similar to the results that were obtained for the CPU implementation in Section 3.3.3. The performance of the convolutional neural network training is very far from the theoretical peak performance of the hardware of the GPU platform, and for some network layers the performance is more than a factor 100 from this peak performance.

For example when looking at the small-NORB roofline, the update_weights function performs at a factor of 16-32 times the theoretical peak performance of the CPU. The main problem is that it is unclear where these gaps are coming from. Again the conclusion is drawn that the roofline model is not detailed enough to be able to predict the performance of this application. This means it is necessary to come up with a more detailed model that is able to predict the performance with more accuracy, or extend the roofline model with additional ceilings to accurately predict the performance.

The conclusions that were drawn from the training results can be validated by the data in Table 2.3 and 2.4 by comparing the roofline models for the small-NORB network (Figure 3.7a and 3.8a) and the speed sign recognition network (Figure 3.7b and 3.8b). It is clear that for the small-NORB network the GPU in general has a higher performance than the CPU, although the differences are very big. This also follows from the results in Table 2.3. For the speed sign recognition network the CPU has a higher performance than the GPU (Table 2.4) which is also reflected in the corresponding roofline models.

3.4 Result evaluation

The previous section has shown that the roofline model is unable to accurately predict the performance of the neural network application. Because it abstracts away from a lot of architecture specific bottlenecks the prediction deviates too much from the actual performance for a reliable prediction. There are several reasons that can cause the convolutional neural network training application to be unable to fully utilize the CPU and GPU platform. These will be discussed shortly in the upcoming sections. These could very well be the cause of the CNN training application not being able to reach the peak performance of both platforms.

3.4.1 CPU bottlenecks

The Intel CPU architecture is already discussed shortly in Section 3.3.1 and has some pitfalls that could cause the degradation of the performance of the CNN training application as seen in the roofline models in Section 3.3.3. [7]
**Instruction fetch & Predecoding**

The instruction fetch and predecoding stage in the instruction pipeline was improved the least compared to the rest of the pipeline, and therefore could form a bottleneck when it cannot keep up with the speed of the execution units. This will most likely form a problem with CPU-intensive code, with a lot of instructions to be executed.

**Register read stalls**

There are three register read ports on the permanent register file, which is insufficient in many situations. Register read stalls are a likely bottleneck in the Nehalem architecture, and it is possible that this deteriorates the performance of the application. The main cause of register read stalls are values in the register that are often read but seldom written to (for example loop counters).

**Execution units & Execution ports**

Execution ports are also a likely bottleneck when the application generates code that consists of many operations that utilize the same execution port (see Figure 3.3 for an overview of the execution units and execution ports in the Nehalem architecture). Especially when the application is memory-intensive and a lot of load and/or store operations need to be executed it can form a bottleneck because there is only one memory read port and one memory write port.

**Execution latencies**

Execution latencies generally don’t incur any performance drawbacks. Most ALU operations have a latency of one clock cycle, and there is an additional switching latency for moving data between the integer and floating point unit. These only cause a performance problem when there is a long dependency chain in the application.

**Memory bandwidth**

Although the cache and memory bandwidth for the Nehalem architecture has improved significantly compared to its predecessors, it is still a likely bottleneck for memory-intensive applications which require a lot of traffic from/to the memory system.

### 3.4.2 GPU bottlenecks

Just as for the CPU there are a number of possible causes that most probably are causing the CNN training application performance to be much lower than the theoretical peak performance of the GPU hardware.

**#Threads**

Section 3.3.1 mentioned that the GPU architecture is programmed using threads that are organized in thread blocks that are assigned to one of the stream processors (SMs) of the GPU as shown in Figure 3.4. Each streaming processor consists of 32 cuda cores. When a thread block is assigned to a SM it is partitioned into warps of 32 threads per warp and executed on the cuda cores.

The threads are issued in-order, so there is no instruction level parallelism or branch prediction. Reading from the global memory of the GPU has a big latency of hundreds of cycles. This latency is hidden by switching threads and waiting for the data to be available. The same holds for the arithmetic latency, which is 24 cycles on the Fermi architecture (for single precision calculations) due to the pipeline depth. Although making sure there are enough threads available to hide latencies, this is limited by the register and shared memory requirements of the application. Execution of one warp requires 4 cycles, thus can conclude from this that having \( \frac{24}{4} = 6 \) warps per SM to hide the latency of read after write dependencies in the instruction stream, which means theoretically there should at least be \( 32 \cdot 6 = 192 \) threads per SM.
In practice 8 warps per SM are required to fully hide the arithmetic latency [11, 18] which means there should at least be \(32 \cdot 8 \cdot \#SMs\) threads (for the GTX 470 this means there should at least be \(32 \cdot 8 \cdot 14 = 3584\) threads) to be able to achieve the maximum performance with respect to the arithmetic latency. Of course for memory intensive kernels that require a lot of data from global memory the latency is not hidden with this amount of threads and the CUDA core will still be stalled until the data arrives.

**Uncoalesced loads**

When reading from external memory the number of memory transactions that have to be issued for all threads within a warp is first calculated. It is mentioned before that reading from external memory incurs a big latency, so the memory transactions are grouped in as little transactions as possible (called memory coalescing). Whether or not this is possible depends on the size of the memory accesses by each thread and the distributions of the memory addresses across the threads within the warp (see Figure 3.9 for an example of a coalesced memory access). Divergence within a warp does not affect memory coalescing, so even if there are divergent threads within a warp coalesced memory transactions can be utilized. The coalesced memory transactions are 32, 64 or 128-byte wide. The size of the accessed words must be 4, 8 or 16 bytes.

![Figure 3.9: Example of coalesced memory access to external memory in a half-warp](image)

In the CNN training application most communication with global memory consists of 32-bit or 4-byte floating point values. This means that all 16 words accessed in the half warp must lie in the same 64-byte segment of the external memory and they must be accessed in sequence to be able to exploit coalescing of memory transactions. When this is not the case it could have a big impact on the performance, because the amount of required memory transactions to the external memory will increase and the bandwidth to the external memory will become the bottleneck for the application.

**Intra warp diversion**

Conditional branches such as if statements or loops within a kernel can diverge the threads within a single warp. It is also possible that a warp executes a certain operation only for a single thread in that warp (see Listing 3.5). The warp serially executes each branch path taken and disables threads that are not on that path. Eventually when all paths complete the threads converge back to the unified path. Such diversions only occur within a warp and not between different warps, since these are executed independently on the GPU.
Listing 3.5: Example of split execution within a single warp

The consequences of such branches or diversions is limited but still decreases the performance of the kernel since threads are executed that are not important for the application, so if there are a lot of diversions within a kernel it could hurt the performance and cause the peak performance to be infeasible.

3.4.3 Concluding remarks

It is clear that the roofline model is of no use for complex applications such as CNN training without modifications. An option would be to extend the roofline model with additional ceilings for each of the bottlenecks to come up with an accurate prediction. However the dynamic behavior of the training application which takes the network structure as input and can change every execution makes this very hard. Because of this the kernels in the applications represented by the points in the roofline graph would require individual ceilings for each of the discussed bottlenecks in the previous section. For example when the application on the GPU architecture runs with a network with smaller feature maps in the first network layer the number of threads will decrease, decreasing the performance prediction based on the number of concurrent executed threads. However this will only hold for a single network layer, causing a shift for these kernels and their bottlenecks in the roofline graph.

This would result in a chaotic roofline graph with individual ceilings for every kernel (note that the GPU roofline model had about 30 kernels), which is very impractical. With this in mind Chapter 3 and Chapter 4 continue on the idea of the roofline model where a performance model is created based on the bottlenecks described in the previous section to improve the accuracy of the prediction.
Chapter 4

CPU performance prediction of CNN training

Since the roofline model is not able to predict the performance of the CNN application with enough precision due to the complex nature of the CNN training application, an alternative way of analyzing and predicting the performance of the CNN training has to be researched.

A lot of alternative approaches for performance modeling for CPU architectures exist. Most of these use a cycle-accurate processor simulator which provide a detailed insight in the performance, complexity and power consumption of the processor. [13, 5]

Because of the complexity of modern CPU architectures the design space formed by these simulators is very big, composed of the large amount of performance factors such as the processor frequency, function unit issue width, the size and latency of the caches, bandwidth to off-chip memory, branch predictor settings, etcetera. [15] In this case the properties of the neural network also play a big role in the performance on the CPU platform.

Because of this big design space it is impossible to explore all design points before actually implementing the application. It is also an advantage that the application is fixed, because this makes the use of extensive CPU architecture simulators unnecessary because these are too detailed for this case. Disadvantages are that the use of extensive simulators requires more time than desirable for a fast design space exploration for the CNN training application.

To avoid these problems a model is composed after analyzing the bottleneck of the training application that is able to predict the performance on the target CPU by predicting the performance of this bottleneck, and abstracting from other performance factors of the platform. This has the advantage of simplicity which makes the model easily adaptable to other CPU platforms, and extremely fast to enable fast performance prediction and design space exploration.

The idea of this model is very simple. Because the application is generic it uses several input files including an xml file that describes the structure of the neural network. This file contains for example the number of layers, the number of feature maps per layer and the sizes of these feature maps. Using this information in combination with the architecture description and feeding it to a model as shown in Figure 4.1 it is possible to come up with an accurate performance prediction for the application for a specific network.

4.1 Deriving the CPU performance prediction model

To be able to predict the performance of a certain neural network during the training phase the CPU assembly code is studied to derive the exact bottleneck which causes the application not to be able to get to the roofline of the CPU (as shown in Figure 3.3). The training is split up into the three most important functions which affect the performance as shown in Section 2.2, namely run\_convolution\_layer, compute\_local\_gradients and update\_weights. The performance model for
4.2 Forward convolution model

The `run_convolution_layer` function consists of three kernels as shown in Listing 4.1.

```c
for λ := 1 to Λ do
    for r := 0 to R(λ) do
        for m := 0 to M(λ) do {init values at bias}
            for n := 0 to N(λ) do
                Y[λ, r, m, n] := B[λ, r];
        }  
        for q := 0 to R(λ − 1) do
            if qq(λ, r, q) do
                for m := 0 to M(λ) do {MACC kernel convolution}
                    for n := 0 to N(λ) do
                        for k := 0 to K(λ) do
                            for l := 0 to L(λ) do
                                Y[λ, r, m, n] := Y[λ, r, m, n] + V[λ, r, q, k, l] * Y[λ − 1, q, m*S[λ]+k, n*S[λ]+l];
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
for m := 0 to M(λ) do {non-linear activation}
    for n := 0 to N(λ) do
        Y[λ, r, m, n] := sigmoid(Y[λ, r, m, n]);
}  
```

Listing 4.1: The forward convolution function which propagates the neural network inputs forward through the network.

Figure 4.2 shows how the loops in Listing 4.1 conform to the architecture of the convolutional neural network. The only factor that is not shown in this image is the subsample factor $S$. The subsample factor reduces the amount of data by local averaging over a predefined non-overlapping window. The size of this window is described by the subsample factor $S$. 

In the previous chapter is concluded that the application is not getting near the performance roofline, indicating that it is far from the peak performance of the CPU, but the exact reason for this is not yet clear at this point. To analyze this phenomenon another micro benchmark kernel is used that mimics the operation within the two inner loops from Listing 4.1 to study the maximum performance of the forward convolution. Using this as a starting point the entire performance of the *run_convolution_layer* function is derived from which it is possible to create a model to predict the performance for any CNN network. This micro benchmark kernel will be discussed in the next section, and the analysis of the entire convolution function in the following sections.

### 4.3 Forward convolution micro benchmark kernel

The micro benchmark kernel in this section covers the multiply-add operation as performed in the forward convolution phase of the CNN training application (the second kernel in the convolution function shown in Listing 4.1), and is tested to see what the best-case performance will be. For simplicity several things are neglected, such as the complex address calculations and the big nested loop structure. The only thing that is taken into account here is the multiply-add operation and the two innermost loops of the application. The analysis is done without multi-threading enabled because when enabled the assembly code generated by the compiler becomes too complex to analyze by hand. However multi-threading will be taken into account in the final performance prediction model. Listing 4.2 shows source code of this kernel written using SSE intrinsics.

---

Figure 4.2: Visual representation of the nested loops in the *run_convolution_layer* function shown in Listing 4.1
for (int i=0; i<size; i+=16) {
    y1 = _mm_load_ps(layer[1].image[r][i]);
    :
    y4 = _mm_load_ps(layer[1].image[r][i+12]);
    for (k=0; k<ksize; k++)
        for (l=0; l<ksize; l++)
            w = _mm_set_ps1(weight[i+k+l]);
            x1 = _mm_load_ps(layer[0].image[0][i+k+1]);
            y1 = _mm_add_ps(y1, _mm_mul_ps(w, x1));
            :
            x4 = _mm_load_ps(layer[0].image[0][i+k+l+12]);
            y4 = _mm_add_ps(y6, _mm_mul_ps(w, x4));
    _mm_store_ps(&layer[1].image[r][i], y1);
    :
    _mm_store_ps(&layer[1].image[r][i+12], y4);}

Listing 4.2: Micro benchmark kernel performing the multiply-add operation within two nested loops, like in the convolution phase of the CNN training

When analyzing the generated assembly from this micro benchmark kernel, an ideal schedule for the CPU can be composed for scheduling the generated instructions. This schedule is composed by hand and is depicted in Figure 4.3. Table 4.1 shows the instructions that are present in the schedule in Figure 4.3 and future schedules.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>move instruction, moving data from outside the core to the registers</td>
</tr>
<tr>
<td>SL</td>
<td>switching latency, occurs when the output of an instruction is directly used as input for another instruction of another domain (for example from the load to the floating point domain)</td>
</tr>
<tr>
<td>S</td>
<td>shuffle instruction which copies data within an XMM register of the core</td>
</tr>
<tr>
<td>LEA</td>
<td>load effective address instruction</td>
</tr>
<tr>
<td>U</td>
<td>unpack instruction which rearranges data within the 128-bit wide XMM registers of the core</td>
</tr>
<tr>
<td>M</td>
<td>move instruction which moves part of an XMM register (for example the lower 64 bits) to another XMM register</td>
</tr>
<tr>
<td>MUL</td>
<td>multiplies two floating point vectors</td>
</tr>
<tr>
<td>ADD</td>
<td>adds two floating point vectors</td>
</tr>
</tbody>
</table>

Table 4.1: Instruction reference with Figures 4.3 to 4.5

As can be concluded from the schedule in Figure 4.3 the two inner-most loops of the MACC kernel (see Listing 4.1) are not able to reach the maximum performance of the CPU. To be able to execute the multiply-accumulate operation with SSE vector operations there must be several variables packed in the registers on which this vector operation takes place:

1. The output vector 'y1' to 'y4', which are loaded into and written back from the registers every kernel execution (or every execution of the 'k' and 'l' loop)

2. The input vectors 'x1' to 'x4', which elements should be loaded into the registers every iteration of the innermost loop

3. The weight vector 'w', which should be loaded into the registers once every iteration of the innermost loop

Theoretically without using multi-threading the maximum performance is 8 floating point operations (or 25.6 Gigaflops/sec) per cycle when a vector of 4 floating point values is used and
the multiply and addition operations can be carried out in parallel within one core. This kernel achieved a performance of 6.4 floating point operations per cycle (or 20.4 Gigaflops/sec) which is 80% of the maximum theoretical performance. This can be explained by studying the schedule in Figure 4.3. As can be seen, neglecting the loads of "y1" to "y4" of the outer loop, it takes 5 move instructions to prepare the data for the MACC operation each iteration of the inner loop. These instructions use the same execution port on the core (port 2) for loading the data into the registers. Since this port can only handle one load instruction per cycle and the maximum throughput is one cycle, it is not possible to execute these move instructions any faster than 5 cycles. Assuming there is no extra latency from the data loading (for example if it is not present in the cache and it has to be taken from off-chip memory) and the instructions are ideally pipelined, the CPU executes a single iteration of the 'l' loop in 5 cycles. This means that 8 floating point operations are executed in 5 cycles best-case (instead of 4 cycles ideally), yielding a performance decrease of 20% which corresponds to the achieved performance of about 80% of the theoretical maximum performance.

4.4 Analyzing the forward convolution

The micro benchmark kernel in the previous section is not able to reach the peak performance of the CPU, but when analyzing the actual convolution kernel as shown in Listing 4.1 there are a lot of extra factors that decrease the performance, such as address calculations loop overhead or data dependencies.

4.4.1 The innermost loop 'l'

When generating the ideal schedule for the innermost loop 'l' there are some extra instructions that further decrease the performance of the code. Figure 4.4 shows the schedule for the innermost loop when the instructions are ideally scheduled on the target architecture.

The schedule in Figure 4.4 shows that the weight vector has to be loaded once per kernel (once per iteration of the 'k' loop), similar to the micro benchmark kernel from Section 4.3. The difference compared to the micro benchmark kernel is that the application uses a complicated address calculation for the input vector "x" of the MACC operation (as shown in Listing ??) which takes a sub sampling factor into account. This causes non-linear memory addressing which prevents that the input vector can be loaded in a single load instruction, but requires four instructions to load the four floating point values. This can be seen in Figure 4.4 by the four move instructions labeled 'x1' to 'x4'. After these move instructions there are two unpack instructions which combine
these four floating point values into a single floating point vector ‘x’. Only after this the multiply-
add operation can be carried out, which introduces a big dependency chain of load instructions.
The four move instructions to load a single SSE vector with input values is a serious deterioration
of the performance, since four aligned values can be loaded in a single instruction. However it
is possible to load the input values in an aligned fashion and use shuffle and unpack instructions
to manoeuvre the values into a correct SSE vector for the calculations. The problem is that
different sub-sampling values create different access patterns to the memory which can only be
solved by using templates for different sub-sampling values, which is costly to design and therefore
is not done. If this is implemented it could prove to be a good improvement over the current
implementation, but only if the additional instructions to correctly form the SSE vectors do not
become too dominant.

From the results it is concluded that execution port 2 (the yellow blocks in Figure 4.4) is the
bottleneck here since it can only handle a single instruction per cycle. As shown it is utilized 100%
of the time and prevents the other instructions from executing any earlier. From the schedule in
Figure 4.4 is concluded that the maximum throughput of a single iteration of the innermost loop ‘l’
(or the time between two multiply-add operations) is 5 cycles. These 5 cycles cover a multiply
and addition of packed SSE vectors consisting of four floating point values, so there are 8 floating
point calculations done within a single iteration, and therefore 1.6 floating point operations per
cycle. The theoretical maximum for a single core is 8 floating point operations per cycle, so the
performance already drops by a factor of five compared to the theoretical maximum of the CPU
by the large amount of data movement required in the innermost loop of the MACC kernel. It
can also be seen that there is a latency of 10 cycles before the actual floating point calculations
can start for the first iteration, which hurts the performance each time the ‘l’ loop is restarted.
Using this information the number of cycles ($C_l$) for this loop can be predicted using the formula
in Equation 4.1, depending on the size of the kernel (the bound of the ‘l’ loop). Here $ks$ represents
the kernel size, which is equal to the kernel height ($kh$) · kernel width ($kw$).

$$C_l = (kw * 10) + (ks * 5) \quad (4.1)$$

4.4.2 Adding the ‘k’ loop

When taking the ‘k’ loop of the MACC kernel into account there are only some address calculations
that are executed every iteration of the ‘k’ loop that have to be taken into account. However to
be able to execute these address calculations again there is data that has to be loaded into the
registers. Figure 4.5 shows the ideal schedule for the instructions within the ‘k’ loop, and it can be
seen that like with the ‘l’ loop port 2 of the core is the bottleneck due to the data load operations
shown in the yellow blocks.

From the generated assembly code it can be seen that the ‘l’ loop can only start after the last
Figure 4.5: Ideal schedule for the 'k' loop of the MACC kernel in the \textit{run\_convolution\_layer} function shown in Listing 4.1.

load effective address instruction from Figure 4.5 is finished, since the address that is loaded here is used in the first instruction of inner loop 'l'. Therefore the 'k' loop adds an extra delay of 12 cycles to the kernel execution estimation in cycles. This can be taken into account by extending Equation 4.1 with the number of cycles needed for the 'k' loop and form Equation 4.3.

\[
C^{kl} = ((kw \times 10) + (ks \times 5)) + (kw \times 12) \quad (4.2)
\]

\[
C^{kl} = (kw \times 22) + (ks \times 5) \quad (4.3)
\]

4.4.3 Adding the rest of the \textit{run\_convolution\_layer} function

The rest of the loops of the MACC kernel ('q', 'm' and 'n') shown in Listing 4.1 of the \textit{run\_convolution\_layer} function were analyzed in the same way as the two innermost loops. For these loops the bottleneck turned out to be the move instructions that utilize execution port 2 of the core, so the high amount of data movement seems to drastically limit the achievable performance of the training application for the CPU. The other two kernels inside the \textit{run\_convolution\_layer} function which initialize the bias values and applies the activation function (see Listing 4.1) are also taken into account for the final prediction for the number of cycles.

When extending Equation 4.3 with the prediction for other loops inside the MACC kernel Equation 4.5 is formed. The predicted number of cycles for the entire \textit{run\_convolution\_layer} function consists of the three mentioned kernels shown in Equation 4.4 to 4.6. The abbreviations in the formulas are explained in Table 4.2. The reason for the ceiling function is that sub word parallelism is exploited using SSE vectors which are used in the dimension of the 'n' loops that runs over the width of the feature maps. Therefore the number of cycles reduces compared to using scalar vectors containing only a single floating point value by a factor of \( \text{ceil}(\frac{FM}{4}, 1) \).

\[
C^{bias} = #FM \cdot (7 + 7 \cdot (height(FM) \cdot \text{ceil}(\frac{width(FM)}{4}, 1))) \quad (4.4)
\]

\[
C^{conv} = #FM \cdot \text{conn}(FM) \cdot height(FM) \cdot \text{ceil}(\frac{width(FM)}{4}, 1) \cdot ((kw \times 22) + (ks \times 5)) \quad (4.5)
\]

\[
C^{act} = #FM \cdot height(FM) \cdot \text{ceil}(\frac{width(FM)}{4}, 1) \cdot 47 \quad (4.6)
\]

\[
C^{rel} = C^{bias} + C^{conv} + C^{act} \quad (4.7)
\]

Since it is possible to calculate the number of cycles for the \textit{run\_convolution\_layer} function, the application performance can be estimated based on several characteristics of the CNN architecture.
**Abbreviation** | **Explanation**  
---|---  
#FM | number of feature maps within the convolutional network layer  
conn(FM) | number of connected feature maps from the previous network layer  
height(FM) | height of the feature maps in pixels  
width(FM) | width of the feature maps in pixels  
ceil(x, y) | ceiling function which rounds up x to the next multiple of y  
kw | width of the kernel in pixels  
ks | size of the entire kernel in pixels (kernel width \cdot kernel height)  

Table 4.2: Abbreviation overview for Equations 4.4 to 4.7  

(mainly shown in Table 4.2) and characteristics of the CPU used to create an ideal schedule of the generated assembly code. To predict the performance and to be able to compare the predictions to the actual performance the results are first rewritten in a percentage of the theoretical peak performance of the CPU and then rewritten to the predicted execution time. This can be compared to the measured execution time of the actual training application executed on the CPU. To get the predicted percentage of the theoretical peak performance of the CPU Equation 4.8 has to be used. Here $P_{\text{rel}}$ is the predicted performance in GFlops/sec, $FPO_{\text{rel}}$ is the number of floating point operations and $C_{\text{rel}}$ is the predicted number of cycles for execution of the run_convolution_layer function.

\[ P_{\text{rel}} = \frac{FPO_{\text{rel}}}{C_{\text{rel}}} \quad (4.8) \]

The $FPO_{\text{rel}}$ can be calculated from the convolutional neural network information present in the network description file using Equation 4.9 to 4.12.

\[ FPO^{\text{bias}} = 0 \quad (4.9) \]

\[ FPO^{\text{conv}} = #FM \cdot \text{conn}(FM) \cdot \text{height}(FM) \cdot \text{width}(FM) \cdot ks \cdot 2 \quad (4.10) \]

\[ FPO^{\text{act}} = #FM \cdot \text{height}(FM) \cdot \text{width}(FM) \cdot 2 \quad (4.11) \]

\[ FPO^{\text{rel}} = FPO^{\text{bias}} + FPO^{\text{conv}} + FPO^{\text{act}} \quad (4.12) \]

The number of floating point operations for the initialization of the bias values ($FPO^{\text{bias}}$) is zero because the operation in this kernel only sets part of the memory to zero, which is not a floating point operation according to the roofline model. The number of floating point operations for the MACC kernel and activation function kernel conforms to the number of times the statement inside the inner loop of these kernels is executed times two (since there are two floating point operations in these statements). Using Equation 4.7 and 4.12 it is possible to predict the percentage of theoretical peak performance the application will achieve using Equation 4.8, which can easily be rewritten to the amount of time the application will take to execute the run_convolution_layer function.

4.4.4 The **compute_local_gradients** and **update_weights** functions

As mentioned before a lot of the derivation of the model is not reported in this report since this would simply consume too much space in the report, and therefore the derivation of the performance prediction model of the other functions used in the training process, namely the **compute_local_gradients** and **update_weights** functions will not be elaborated further in this report. They are obtained using the same process as with the run_convolution_layer function and yield similar formulas as in Equation 4.4 to 4.12 to predict the performance of these functions.
4.4.5 Adding multi-threading to the model

The performance prediction model up to this point only predicts the performance on a single thread and does not take multi-threading into account. To multi-threading into account each of the functions of the CNN training process have to be modified independently since multi-threading is applied at different levels for each function. For the compute_local_gradients function it is not possible to apply multi-threading because of sequential dependencies in the code structure. This prevents the use of dividing the work over multiple cores since this would require too much communication of data between cores, preventing a speedup in the execution time from this multi-threading. A solution to use multi-threading for the compute_local_gradients function is to invert the direction of the computation of the local gradients. This requires extra computations but enables the possibility to divide the work over several independent threads. However since the maximum number of concurrent threads on the used CPU is 8 (with simultaneous multi-threading enabled) this proved to be no improvement over the single-threaded solution with respect to the execution time of the function. For the run_convolution_layer function it is possible to execute the feature maps within a layer in parallel. Since simultaneous multi-threading (running two threads on a single core concurrently) introduces too much overhead to be profitable the feature maps of a network layer can be split amongst four threads that execute at the same time. When enabling multi-threading the theoretical maximum floating point performance is 102.4 GFlops/sec. To be able to predict performance for a network layer and the estimated execution time for the forward convolution Equation 4.13 is used. This equation multiplies the performance of the single-threaded forward convolution with a factor which represents how efficient the multi-threading is.

\[
P_{MT}^{rel} = \frac{4 \cdot \#FM}{\text{ceil}(\#FM, 4)} \cdot P^{rel}
\]  

(4.13)

For example when a layer consists of 8 feature maps, the run_convolution_function will be around \(4 \cdot \frac{8}{8} = 4\) times faster than the single-threaded approach. And when a layer consists of 10 feature maps it will be around \(4 \cdot \frac{10}{\text{ceil}(10, 4)} = \frac{40}{12} = 3.33\) times faster, because the third warp only uses two out of the possible four threads and therefore two cores remain idle. This estimation turned out to be very accurate, since without SMT enabled there is very little overhead from the multi-threading. In the update_weights function the network layers are divided over the threads when multi-threading is enabled. Since the investigated networks in this study (small-NORB and speed sign network) both consist of four network layers it is easily divided over the four available cores of the CPU. This implies that the slowest network layer by means of computation time determines the execution time of the entire function, and therefore multi-threading is easily introduced in the model for this function by taking the maximum execution time of the layers at the model output and taking this as execution time of the update_weights function.
4.5 Results for the CPU performance prediction model

The results of the performance prediction model for CNN training on CPU are shown in Table 4.3. This table shows the predicted execution time versus the measured execution time of functions for each network layer of the CNN network for both the small-NORB and the speed sign recognition neural networks. The results are with multi-threading enabled, as discussed in Section 4.4.5.

<table>
<thead>
<tr>
<th>CNN Network</th>
<th>Training Function</th>
<th>Network Layer</th>
<th>Predicted E.T. [ms]</th>
<th>Measured E.T. [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>small-NORB</td>
<td>run_convolution_layer</td>
<td>1</td>
<td>0.073</td>
<td>0.075</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.049</td>
<td>0.050</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>0.026</td>
<td>0.027</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.003</td>
<td>0.004</td>
</tr>
<tr>
<td></td>
<td>compute_local_gradients</td>
<td>2</td>
<td>0.256</td>
<td>0.261</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>0.043</td>
<td>0.051</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.002</td>
<td>0.006</td>
</tr>
<tr>
<td></td>
<td>update_weights</td>
<td>1</td>
<td>0.282</td>
<td>0.318</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.175</td>
<td>0.176</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>0.156</td>
<td>0.178</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.004</td>
<td>0.010</td>
</tr>
<tr>
<td>speed sign</td>
<td>run_convolution_layer</td>
<td>1</td>
<td>0.014</td>
<td>0.017</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.017</td>
<td>0.021</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>0.014</td>
<td>0.017</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.004</td>
<td>0.006</td>
</tr>
<tr>
<td></td>
<td>compute_local_gradients</td>
<td>2</td>
<td>0.065</td>
<td>0.070</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>0.020</td>
<td>0.029</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.002</td>
<td>0.009</td>
</tr>
<tr>
<td></td>
<td>update_weights</td>
<td>1</td>
<td>0.032</td>
<td>0.038</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.048</td>
<td>0.058</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>0.073</td>
<td>0.087</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.007</td>
<td>0.012</td>
</tr>
</tbody>
</table>

Table 4.3: Results for the small-NORB and speed sign recognition neural networks for the CPU
As can be seen from the results in Table 4.3 and corresponding Figure 4.6 & 4.7 the model is able to predicts the performance of the training functions with good accuracy. The big errors are mainly in the fourth network layer for both the small-NORB and speed sign recognition network. This is because there is very little parallelism in the fourth layer because it consists of a small amount (5 and 9 respectively) of neurons. This makes it that the performance is hard to predict and often worse because the compiler has trouble generating efficient code for this network layer. It can be seen that the run convolution layer function has a small execution time compared to the other two functions. The reason for this is that this function is suitable for parallel execution, where the other functions (and especially the compute local gradients function) are more difficult to execute in parallel. The update weights and compute local gradients function is require about the same execution time on the CPU (note that the layers of the update weights function are executed concurrently on different threads, so the slowest layer determines the total execution time of the function). Noticeable is that the predicted execution time of the model is faster than the measured execution time for every function and network layer. This is easily explainable by the fact that when composing the model using the generated assembly code an optimal schedule is composed. However it is not certain that the compiler is actually able to create this ideal schedule and might come up with a less efficient schedule (which is probably the main reason for the bad prediction of the fourth network layer). Besides this all load operations in the schedules assume that the data is available in the caches and doesn’t need to be extracted from the off-chip memory, which would introduce an extra latency. This is also not the case in the real situation, causing the predicted performance to be more optimal than the actual performance.
Chapter 5

GPU performance prediction model

As concluded from the GPU roofline models for the CNN training application discussed in Section 3.3.3 the application is not able to fully exploit the performance of the GPU. Just as for the CPU implementation the goal is to be able to predict the performance of the training and find out what the bottleneck is that causes the performance degradation on this platform. To be able to answer these questions it is necessary to investigate the GPU architecture and develop a model that is as simple as possible to maintain simplicity but thorough enough to capture the main performance factors of the GPU platform.

The approach for the GPU performance prediction model is similar to the one for the CPU platform as shown in Figure 4.1. There are already several publications using performance prediction models for the GPU platform. These can be split into roughly three categories.

Analytical models which use a mathematical abstraction of the application in combination with properties of the target architecture, the input application and the input data. [4] Profile-based models consist of two parts. In the first part the application is executed under an instrumental tool generating several statistics such as the execution time and memory accesses. In the second part of the model these statistics are fed to an analysis tool which generates a performance estimation of the application’s run for the target platform. [17] Simulation-based models predict the performance of the GPU by simulating the architecture and running the application on this simulated platform. This way every instruction is simulated and the prediction can be very accurate. [2, 8, 12]

Like with the CPU performance model using a simulation-based approach is not really an option since this is too extensive, requiring too much time for design space exploration based on this model.

The analytical method is preferred (like with the CPU prediction model) because it is as simple and transparent as possible while still assuring good accuracy for the performance prediction. However due to the complex nature of the CNN training application analyzing the generated PTX assembly code turned out to be too time consuming for this project, since it would require weeks of work to implement.

Because of the limited time left to realize the GPU model in this project the profile-based approach is chosen. The profile-based prediction is more accurate than the analytical approach because it uses the exact number of instructions instead of an estimation. However this approach is slightly less efficient because it requires additional steps (profiling the application by an instrumental tool) before being able to predict performance. Upside of this approach is that the time penalty required for this profiling is limited since it only needs to profile when another neural network structure is used and is still architecture independent (opposed to the simulation-based approach).
5.1 GPU performance prediction model components

Figure 5.1: Structure of the GPU performance prediction model

The general structure of the GPU performance prediction model is shown in Figure 5.1. There are three types of inputs that are needed to be able to generate the performance prediction on the GPU platform; programmer and application specific parameters, the application source and architecture specific parameters.

5.1.1 Application source information extraction

The first input is the application source of the CNN training application. Using this source PTX (Parallel Thread eXecution; the assembly language used in Nvidia’s CUDA programming environment) traces are generated using the GPUOcelot tool. [6]

GPUOcelot is an open source compilation framework intended to provide a set of translation tools with the PTX code as starting point (see Figure 5.2 for the structure of the GPUOcelot tool). From this PTX code translations can be made to diverse many-core architectures such as Nvidia or AMD GPUs or multi-core CPUs. Besides these translations the tool is able to emulate the PTX code using a functional emulator for analysis purposes.

The top path in Figure 5.2 is important for the model used in this work. GPUOcelot generates a trace from the PTX code of the application using a functional PTX emulator that computes the complete architectural state of a GPU for each dynamic instruction. This can be used to tune the application for performance optimization or in our case extract useful information of the application to be able to predict the performance.

This useful information consists of the two most important performance indicators for execution on the GPU architecture (which are also the main performance factors in the roofline model): the computational and memory limitations of the architecture. These can be analyzed by the amount of computational and memory instructions in the instruction trace generated by GPUOcelot.

Using a hand-made script the instruction trace generated by GPUOcelot is analyzed and the useful information is extracted. This script checks the trace for the first thread for each thread block, and it is assumed that the rest of the thread block will behave in the same way. This is not the case when threads diverge because of explicit branches within the CUDA kernel, but this is neglected since it does not normally affect the performance. This useful information that is extracted from the instruction trace and required for the prediction model is shown in Table 5.1 under the instruction trace input type.
5.1.2 The prediction model

Approach of the prediction model

The prediction model to predict the performance of the application on the GPU architecture (shown in Figure 5.1) is based on the analytical model for GPUs by Sunpyo Hong and Hyesoon Kim [10]. The introduced model introduces notions CWP (Computational Warp Parallelism) and MWP (Memory Warp Parallelism). CWP represents the number of thread warps that can overlap with the memory accesses to hide the memory latency (similar to the notion of arithmetic intensity), and MWP represents the number of thread warps that can concurrently access the external memory. MWP is computed by the memory bandwidth to external memory, the memory bank parallelism and the number of running warps per SM. Based the model assumes the bottleneck of the application can be categorized in one of three cases:

- **Not enough warps running**
  The CWP and MWP are bounded by the number of active warps per SM. If the application does not have enough concurrently executing warps the available warp parallelism cannot be fully exploited on the GPU architecture.

- **CWP > MWP**
  When CWP is greater than MWP (which is generally the case) there are enough warps that are waiting for the values that need to be communicated to the external memory. This means the computation periods are hidden by the periods that are needed to communicate data from external memory. In this case the kernel is memory-bounded.

- **MWP > CWP**
  When MWP is greater than CWP many memory operations can be overlapped and the computational time is dominant for the kernel. In this case the kernel is compute-bounded.
The approach for this analytical model is roughly the same as the roofline model explained in Chapter 3 since it assumes the main bottlenecks are the bandwidth to external memory and the computational peak performance of the architecture. However this model takes a lot more factors into account that determine the performance of the application. In Section 3.4.2 several causes for performance loss were listed for the GPU platform. These factors such as the number of threads and thread blocks executed or whether or not the load instructions from external memory are executed in a coalesced or uncoalesced matter are part of this analytical model, which makes it a lot more exact than the roofline model. Therefore this model is able to give a much more accurate estimation of the performance and a better insight in the bottlenecks of the CNN training application for the GPU platform.

Zhenyu Ye extended the analytical model of Hong and Kim to increase the performance estimation accuracy with the notions of Pipeline Warp Parallelism (PWP) and kernel launch overhead. The kernel launch overhead takes the time it takes to launch a kernel on the GPU into account, which varies for different numbers of thread blocks, different threads per thread block and threads with a different number of arguments. This overhead is in the margin of $\mu$s, so it can be of great significance for small kernels. The PWP represents the maximum number of warps per SM that can be in the execution pipeline of the SM. For example when the pipeline latency of the GPU architecture is 24 cycles the next instruction cannot be issued until the previous instruction of the same warp writes back the result. When it takes four cycles to issue an instruction on the architecture there should at least be 6 active warps per SM to hide this pipeline latency.

**Inputs of the prediction model**

Figure 5.1 shows that the prediction model takes three sorts of inputs to be able to predict the performance of the CNN training application. There are three types of inputs that are needed for the analytical model treated in the previous section, namely programmer and application specific parameters, the parameters extracted from the instruction trace generated from the application source (see Section 5.1.1) and architecture specific parameters. An overview of these different parameter groups and their contents can be seen in Table 5.1.
<table>
<thead>
<tr>
<th>Input Type</th>
<th>Abbreviation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Trace</td>
<td>#Total_insts</td>
<td>Total number of instructions in the instruction trace</td>
</tr>
<tr>
<td></td>
<td>#Comp_insts</td>
<td>Number of computational instructions in the instruction trace</td>
</tr>
<tr>
<td></td>
<td>#Mem_insts</td>
<td>Number of memory instructions in the instruction trace</td>
</tr>
<tr>
<td></td>
<td>#Uncoal_Mem_insts</td>
<td>Number of uncoalesced memory instructions in the instruction trace</td>
</tr>
<tr>
<td></td>
<td>#Coal_Mem_insts</td>
<td>Number of coalesced memory instructions in the instruction trace</td>
</tr>
<tr>
<td></td>
<td>#Synch_insts</td>
<td>Number of synchronization instructions in the instruction trace</td>
</tr>
<tr>
<td>Architecture</td>
<td>#Threads_per_warp</td>
<td>Number of threads per warp</td>
</tr>
<tr>
<td></td>
<td>Issue_cycles</td>
<td>Number of cycles to execute one instruction</td>
</tr>
<tr>
<td></td>
<td>Freq</td>
<td>Clock frequency of the SM processor</td>
</tr>
<tr>
<td></td>
<td>Mem_Bandwidth</td>
<td>Bandwidth between the GPU cores and external memory</td>
</tr>
<tr>
<td>Programmer-specific</td>
<td>#Threads_per_block</td>
<td>Number of threads per thread block</td>
</tr>
<tr>
<td></td>
<td>#Blocks</td>
<td>Number of thread blocks per kernel</td>
</tr>
<tr>
<td>Application-specific</td>
<td>#Active_SMs</td>
<td>Number of active SMs</td>
</tr>
<tr>
<td></td>
<td>#Active_blocks_per_SM</td>
<td>Number of concurrently running blocks per SM</td>
</tr>
</tbody>
</table>

Table 5.1: List of most important parameters provided to the prediction model

Based on the parameters in Table 5.1 the analytical model is able to predict the performance using the analytical model per GPU kernel of the application.

5.2 Results for the GPU performance prediction model

The results of the performance prediction model for the GPU for CNN training are shown in Figure 5.3 and Figure 5.4, which shows the measured and predicted execution time for each kernel of the training application for both the small-NORB and speed sign recognition neural networks. The explanation of the abbreviations is shown in Table 5.2.
<table>
<thead>
<tr>
<th>Training Function</th>
<th>Kernel Abbreviation</th>
<th>Kernel functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_convolution_layer</td>
<td>IBV</td>
<td>\textit{initialize_bias_values}: initializes the values of neurons in the feature maps of that network layer to the bias value</td>
</tr>
<tr>
<td></td>
<td>CWK</td>
<td>\textit{convolve_weight_kernel}: performs the forward convolution operation of that layer</td>
</tr>
<tr>
<td></td>
<td>SA</td>
<td>\textit{sigmoid_activation}: applies the non-linear activation function</td>
</tr>
<tr>
<td>compute_local_gradients</td>
<td>COE</td>
<td>\textit{compute_output_error}: computes the sum squared error of the outputs of the neural network after the forward convolution</td>
</tr>
<tr>
<td></td>
<td>CLG</td>
<td>\textit{compute_local_gradients}: computes the local gradients for every connection when propagating backwards through the network</td>
</tr>
<tr>
<td>update_weights</td>
<td>SLG</td>
<td>\textit{sum_local_gradients}: calculates the local gradients by back-propagation from the succeeding layers</td>
</tr>
<tr>
<td></td>
<td>SA</td>
<td>\textit{sum_array}: calculates the gradients of the weights by summing the local gradients</td>
</tr>
<tr>
<td></td>
<td>UDW</td>
<td>\textit{update_delta_weights}: updates the weights using the delta rule and the calculated weight gradients</td>
</tr>
</tbody>
</table>

Table 5.2: Abbreviation explanation with Figure 5.3 and 5.4

Figure 5.3: Results for the small-NORB network

Figure 5.4: Results for the speed sign network
The performance prediction has about the same precision as the CPU performance prediction model, and the biggest errors occur for the update_delta_weights kernel (U/DW in the bar graphs). The reason for this is that this kernel is way more complex than the other kernels and uses the shared memory for data storage, which is not possible for most of the other kernels because of the large amount of required data to be communicated. Because of this the predicted execution time is higher than the actual measured execution time.

While generating these results a good insight of the bottlenecks for the GPU implementation is obtained. In almost all kernels the CWP is greater than the MWP (typically with a factor of 1.5 - 5) from which is concluded that the kernels are all bounded by the memory bandwidth. By inspecting the analysis of the instruction trace of the application it becomes clear that there are two main problems that cause the performance degradation.

The first reason is that in most cases it is not possible to use coalesced loads, because it is not properly aligned for coalesced access in external memory. When extending the GPU roofline models with a memory ceiling when using solely uncoalesced loads, the roofline model of Figure 5.5 arises.

Although the introduction of the scattered bandwidth ceiling (where all threads need to access the external memory in an uncoalesced fashion) it can still be seen that the roofline model is not precise enough to accurately predict the performance of the CNN training application. The prediction error when using the roofline model still varies between a factor 2 and 32 based on the scattered bandwidth ceiling and the information from the analytical model on the memory traffic.

The second cause of the bad performance of the GPU is that there is limited parallelism in the application because of the sequential behavior of the neural network training algorithm. Because of this limited parallelism the SMs of the GPU cannot be fully utilized, and for some kernels several SMs are inactive because there are not enough thread blocks to occupy them.

This behavior is different for each kernel and each network layer, so therefore it cannot be taken into account in the roofline model because this would require introducing a new ceiling for every kernel and network layer. This approves the introduction of our GPU model which is able to predict the performance of the training application more precisely because it takes multiple factors into account the roofline model ignores such as the number of thread blocks and size of the thread blocks or the pipeline latency.
Chapter 6

Model verification &
Design space exploration

With the models that were composed for both the CPU and GPU platforms it is possible to perform a Design Space Exploration (DSE) for neural networks with different sizes (i.e. different number of layers, different sizes of feature maps, etcetera) and predict the performance on different architectures. In this chapter first the model is verified for different platforms on the subject of accuracy in Section 6.1, after which a design space exploration is carried out for different networks 6.2.

6.1 Performance model verification

In this section the accuracy of the performance models is verified for different target platforms for the neural network training. Figure 6.1 and Figure 6.2 in the next section shows the results for the Intel core i7-960 and Nvidia GTX-470, along with results for multiple other platforms.

It shows that the small-NORB network training requires more execution time than the speed sign recognition network. It can also be concluded that training the small-NORB network on the GPU platform will be a bit faster than training it on the CPU platform. In contrast the speed sign recognition network will train a lot faster using the CPU platform than when using the GPU platform. These conclusions are already proven to be correct earlier in this report, and are easily extracted from Figure 6.1. A big advantage of this approach is that it is easily expendable to other architectures and neural network topologies, and they can be compared to each other in a similar graph.

<table>
<thead>
<tr>
<th>Platform</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core 2 Quad-Q6600</td>
<td>CPU</td>
</tr>
<tr>
<td>Intel Core i7-960</td>
<td>CPU</td>
</tr>
<tr>
<td>Nvidia GeForce GTX-460</td>
<td>GPU</td>
</tr>
<tr>
<td>Nvidia GeForce GTX-470</td>
<td>GPU</td>
</tr>
<tr>
<td>Nvidia GeForce GTX-570</td>
<td>GPU</td>
</tr>
</tbody>
</table>

Table 6.1: Platform choices for neural network training

The same neural networks (small-NORB object recognition and speed sign recognition) will be used for the verification of the models on different platforms to see how the models that were developed in this study are useful for design choices for the neural network training application. The goal is to train the neural network as fast as possible, but it is not always the case that the

1Platforms used for benchmarking and performance prediction model developing
GPU is the fastest solution to train a certain network (as can be seen from the results in Section 2.3. The different platform choices used for the verification are shown in Table 6.1.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Function</th>
<th>small-NORB network</th>
<th>speed sign network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel C2Q Q6600</td>
<td>run_convolution_layer</td>
<td>195</td>
<td>267</td>
</tr>
<tr>
<td></td>
<td>compute_local_gradients</td>
<td>390</td>
<td>518</td>
</tr>
<tr>
<td></td>
<td>update_weights</td>
<td>453</td>
<td>376</td>
</tr>
<tr>
<td>Intel Core i7-960</td>
<td>run_convolution_layer</td>
<td>153</td>
<td>157</td>
</tr>
<tr>
<td></td>
<td>compute_local_gradients</td>
<td>302</td>
<td>319</td>
</tr>
<tr>
<td></td>
<td>update_weights</td>
<td>285</td>
<td>321</td>
</tr>
<tr>
<td>Nvidia GTX-460</td>
<td>run_convolution_layer</td>
<td>150</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>compute_local_gradients</td>
<td>139</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>update_weights</td>
<td>280</td>
<td>351</td>
</tr>
<tr>
<td>Nvidia GTX-470</td>
<td>run_convolution_layer</td>
<td>184</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td>compute_local_gradients</td>
<td>151</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>update_weights</td>
<td>262</td>
<td>251</td>
</tr>
<tr>
<td>Nvidia GTX-570</td>
<td>run_convolution_layer</td>
<td>155</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>compute_local_gradients</td>
<td>125</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>update_weights</td>
<td>198</td>
<td>160</td>
</tr>
</tbody>
</table>

Table 6.2: Results of the example design space exploration for CNN training

Table 6.2 shows the results of an example design space exploration for the small-NORB and speed sign recognition CNN networks on different CPU and GPU platforms. Figure 6.1 and Figure 6.2 show the same results in a more clear visual representation for the small-NORB and speed sign recognition neural networks respectively.

![Figure 6.1: Results for the design space exploration for the small-NORB neural network for different platforms](image)

Figure 6.1: Results for the design space exploration for the small-NORB neural network for different platforms

47
Some interesting conclusions can be drawn from the results of the model verification for different platforms. First of all, all the prediction of the execution time for each network and platform are very good compared to the roofline model discussed in Chapter 3. The maximum error in the execution time prediction is around 20%, where the roofline model had a maximum prediction error of a factor 50 for the CPU and GPU implementation. This error is within the margin of 25% that was initially taken as sufficient for an accurate prediction of the training performance.

The results of the CPU implementations are comparable, where the model predicts a slightly lower execution time than the actual measured execution time. This is because the model is composed by scheduling the executed instructions in an ideal manner which is probably not how the instructions are actually scheduled in reality. Besides the model assumes all load instructions are cache hits, which makes it slightly optimistic compared to the actual implementation. This also explains the fact that the Core 2 Quad Q6600 CPU platform performance prediction is more optimistic than the prediction of the Core i7-960 CPU platform. The Core 2 Quad is technically two Core 2 Duo processors glued together, which means that there is no level 3 cache shared amongst all processors like in the Core i7 processor. Communication from the lowest cache level to the other two cores is done over a bus connecting the two Core 2 Duo cores, which introduces extra latency and causes a more optimistic performance prediction for this platform.

Besides this, the Core i7 and Core 2 Quad are very alike, where the Core i7 has a higher clock speed (3.2 GHz versus 2.4 GHz) and thus a higher floating point performance, but this can easily be compensated for by modifying a model parameter. Another difference is that the Core 2 Quad has a switching latency of three cycles when switching between the integer and floating point domain, which could also be compensated for by adjusting a model parameter.

Last the Core 2 Quad does not make use of hyper-threading and the latest version of the SSE instruction set. This does not make a difference for the results because in the preparation project it was proven that enabling hyper-threading only decreases the performance because when using four threads the four CPU cores each run one thread. When using five or more threads one or more cores have to run two threads which is not resulting in an increase in performance in this case. The use of the latest SSE instruction set in the Core i7 processor also makes no difference since this only extends the previous instruction set with various instructions that are not used in the convolutional neural network application.

When looking at the GPU results from Figure 6.1 and Figure 6.2 there are also some interesting conclusions that can be drawn. The serial numbers of the GPUs used in the design space exploration look very similar, but there are some big architectural differences between these three
GPUs. The main differences of these architectures are shown in Table 6.4.

<table>
<thead>
<tr>
<th>Platform</th>
<th>#SMs</th>
<th>#CUDA Cores</th>
<th>#Int/FP Processors per SM</th>
<th>Processor Clock [MHz]</th>
<th>Memory B/W [GB/sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nvidia GeForce GTX-460</td>
<td>7</td>
<td>336</td>
<td>48</td>
<td>1350</td>
<td>115.2</td>
</tr>
<tr>
<td>Nvidia GeForce GTX-470</td>
<td>14</td>
<td>448</td>
<td>32</td>
<td>1215</td>
<td>133.9</td>
</tr>
<tr>
<td>Nvidia GeForce GTX-570</td>
<td>15</td>
<td>480</td>
<td>32</td>
<td>1464</td>
<td>152.0</td>
</tr>
</tbody>
</table>

Table 6.3: Most important architectural differences in the GPU architectures used for the design space exploration

Table 6.3 shows that the GTX-460 has more integer and floating point execution units per SM, but also has less SMs than the GTX-470 and GTX-570 GPUs, however the warp size is 32 threads for all three GPUs. The GTX-460 architecture also has two schedulers to schedule warps on the execution units (one for warps with an odd warp ID and one for warps with an even warp ID) where the other architectures have a single scheduler to schedule the warps. This creates an advantage over the GTX-470 and GTX-570 because the warps are scheduled in a superscalar fashion increasing the possible instruction throughput for well-balanced applications. From the results it shows that this does not introduce a big advantage for the neural network application, because the eventual performance is not faster than that of the other GPU platforms. This is because the application requires a lot of communication to the memory through the warp parallelism cannot be exploited optimally, and because the memory bandwidth is lower than for the GTX-470 and GTX-570 platforms the eventual performance is worse. This holds especially for the speed sign recognition application from Figure 6.2 because there are lesser threads and thus warps due to the smaller images that are used in this network, and the warp parallelism is even lower decreasing the performance for this application.

When comparing the GTX-470 and GTX-570 GPUs the GTX-570 has one more SM (32 more CUDA cores), a higher processor speed and a higher memory bandwidth. This causes a significant performance for the small-NORB convolutional neural network in Figure 6.1 and a minor performance increase in the case of the speed sign recognition application in Figure 6.2. This is obviously because of the higher memory bandwidth and processor speed, but in the case of the small-NORB network there are more threads so the extra SM of the GTX-570 can be fully exploited for most functions. However for the speed sign recognition application there are very few threads causing not all SMs to be active, removing the advantage of the extra SM of the GTX-570 and decreasing the performance increase over the GTX-470 GPU.

6.2 Design Space Exploration

The developed models can be used to predict the performance of a convolutional neural network on a certain platform as shown in the previous section. It is also interesting to see what happens if the neural network size is varied and use the models to evaluate the performance on the different platforms. For this experiment both the speed sign recognition network and the small-NORB network were used, and the network characteristics are shown in Table 6.4. See Figure 1.2 for a visual representation of such network and the feature map sizes, convolution kernel sizes and sub sampling sizes.

The performance of different networks can be checked using the models, but there are some constraints on the network characteristics that need to be taken into account:

- The input image is fixed at 96x96 pixels (small-NORB) or 32x32 pixels (speed sign)
- The third and fourth layer consist of single neurons (feature map size of 1x1 pixels)
- The fourth layer consists of 5 neurons (small-NORB) or 9 neurons (speed sign)
Table 6.4: Characteristics of the convolutional neural networks (FM = Feature Map Size, CK = Convolution Kernel Size, SS = Sub sampling Factor)

<table>
<thead>
<tr>
<th></th>
<th>Layer 1</th>
<th>Layer 2</th>
<th>Layer 3</th>
<th>Layer 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Image</td>
<td>FM</td>
<td>CK</td>
<td>SS</td>
<td>FM</td>
</tr>
<tr>
<td>small-NORB network</td>
<td>96x96</td>
<td>23x23</td>
<td>6x6</td>
<td>1x1</td>
</tr>
<tr>
<td>speed- sign recognition network</td>
<td>32x32</td>
<td>14x14</td>
<td>5x5</td>
<td>1x1</td>
</tr>
</tbody>
</table>

To keep the DSE plausible and restricted some extra constraints were introduced:

- The number of layers is fixed
- The number of feature maps per layer is fixed
- The convolution kernel sizes are varied between 2-8
- The sub sampling factors are varied between 2-4

As a consequence of varying the convolution kernel sizes and the sub sampling factors, the sizes of the feature maps change accordingly. Based on these constraints several network configurations were plausible for this DSE which are shown in Table 6.5. Networks that produced similar data are left out to keep the DSE clear. Here the variations on the small-NORB network consist of configuration 1-10 and variations on the speed sign recognition network of configuration 11-17. Finally configuration 18 is added to explore a point in the design space where the trade-off between the CPU and GPU platform is more or less equal. When looking at the number of floating point operations for the networks in the last column of Table 6.5 the points where the performance of the CPU and GPU platforms is close to each other is between the workload of both neural networks, and configuration 18 validates this by showing the performance is indeed very close between the different platforms.
<table>
<thead>
<tr>
<th>Conf.</th>
<th>Input</th>
<th>Layer 1</th>
<th>Layer 2</th>
<th>Layer 3</th>
<th>Layer 4</th>
<th>Flops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FM</td>
<td>CK</td>
<td>SS</td>
<td>FM</td>
<td>CK</td>
</tr>
<tr>
<td>1</td>
<td>96x96</td>
<td>31x31</td>
<td>4x4</td>
<td>3x3</td>
<td>8x8</td>
<td>8x8</td>
</tr>
<tr>
<td>2</td>
<td>96x96</td>
<td>31x31</td>
<td>4x4</td>
<td>3x3</td>
<td>6x6</td>
<td>8x8</td>
</tr>
<tr>
<td>3</td>
<td>96x96</td>
<td>23x23</td>
<td>5x5</td>
<td>4x4</td>
<td>7x7</td>
<td>3x3</td>
</tr>
<tr>
<td>4</td>
<td>96x96</td>
<td>23x23</td>
<td>5x5</td>
<td>4x4</td>
<td>5x5</td>
<td>4x4</td>
</tr>
<tr>
<td>5</td>
<td>96x96</td>
<td>23x23</td>
<td>5x5</td>
<td>4x4</td>
<td>6x6</td>
<td>6x6</td>
</tr>
<tr>
<td>6</td>
<td>96x96</td>
<td>23x23</td>
<td>5x5</td>
<td>4x4</td>
<td>8x8</td>
<td>8x8</td>
</tr>
<tr>
<td>7</td>
<td>96x96</td>
<td>23x23</td>
<td>5x5</td>
<td>4x4</td>
<td>4x4</td>
<td>8x8</td>
</tr>
<tr>
<td>8</td>
<td>96x96</td>
<td>30x30</td>
<td>7x7</td>
<td>3x3</td>
<td>7x7</td>
<td>3x3</td>
</tr>
<tr>
<td>9</td>
<td>96x96</td>
<td>30x30</td>
<td>7x7</td>
<td>3x3</td>
<td>8x8</td>
<td>7x7</td>
</tr>
<tr>
<td>10</td>
<td>96x96</td>
<td>30x30</td>
<td>7x7</td>
<td>3x3</td>
<td>6x6</td>
<td>7x7</td>
</tr>
<tr>
<td>11</td>
<td>32x32</td>
<td>14x14</td>
<td>5x5</td>
<td>2x2</td>
<td>5x5</td>
<td>5x5</td>
</tr>
<tr>
<td>12</td>
<td>32x32</td>
<td>15x15</td>
<td>3x3</td>
<td>2x2</td>
<td>6x6</td>
<td>4x4</td>
</tr>
<tr>
<td>13</td>
<td>32x32</td>
<td>10x10</td>
<td>3x3</td>
<td>3x3</td>
<td>2x2</td>
<td>3x3</td>
</tr>
<tr>
<td>14</td>
<td>32x32</td>
<td>14x14</td>
<td>5x5</td>
<td>2x2</td>
<td>2x2</td>
<td>7x7</td>
</tr>
<tr>
<td>15</td>
<td>32x32</td>
<td>9x9</td>
<td>6x6</td>
<td>3x3</td>
<td>2x2</td>
<td>2x2</td>
</tr>
<tr>
<td>16</td>
<td>32x32</td>
<td>13x13</td>
<td>7x7</td>
<td>2x2</td>
<td>4x4</td>
<td>2x2</td>
</tr>
<tr>
<td>17</td>
<td>32x32</td>
<td>13x13</td>
<td>7x7</td>
<td>2x2</td>
<td>3x3</td>
<td>8x8</td>
</tr>
<tr>
<td>18</td>
<td>80x80</td>
<td>19x19</td>
<td>5x5</td>
<td>4x4</td>
<td>5x5</td>
<td>5x5</td>
</tr>
</tbody>
</table>

Table 6.5: Characteristics of the plausible network configurations for DSE (FM = Feature Map Size, CK = Convolution Kernel Size, SS = Sub sampling Factor)

The data in Table 6.5 shows that changing the key features of the network incurs a drastic change in the amount of calculations that have to be done. Because of this only looking at the predicted execution time that comes out of the models for the different networks is not a fair because the amount of calculations varies per network. Therefore the performance (in GFlops / Sec) is plotted for the different network configurations in Figure 6.3.

Figure 6.3 shows that the performance of the CPU platforms are more or less stable for both the small-NORB and the speed sign recognition networks. The GPU platforms show a lot more variations for different network configuration from Table 6.5. This can be explained by the fact that the CPU uses most of its resources for all network configurations which makes it scale more or less linearly when varying the amount of work to be done. The GPU platforms however have a lot more available resources. It is already concluded that the standard small-NORB network is not able to fully exploit all of these resources, but by enlarging the amount of neurons by modifying the network more work has to be done and the available resources on the GPU platform can be used more effectively. This leads to a higher performance, which is clearly the case for configuration 1, 2, 8, 9 and 10 in Figure 6.3 (which are also the configurations with the highest amount of floating point operations).

It seems that the CPU results for configuration 6 are worse than all other variations of the small-NORB network. This is caused by the fact that this configuration has very small feature maps in the second network layer compared to the other configurations and therefore is not able to reach the same performance by exploiting the parallelism within feature maps like the other configurations.

When comparing the variations of the small-NORB and speed sign networks there is not much difference between the performances of the CPU platforms but there is a drastic performance drop in the performance of the GPU platforms which perform much worse than the CPU platforms for the speed sign network configurations. This is caused by the small feature maps of this network leading to poor utilization of the available resources of the GPU platforms. The CPU can still be effectively used which explains why the performance drop for these platforms is much smaller.

Configuration 18 is a manual composed network with an input image size of 80x80 which shows
Figure 6.3: DSE results for different network configurations

a Break-Even Point (BEP) where the performance of the different platforms is more or less equal. The amount of neurons and floating point operations of this network configuration is in between the speed sign recognition and small-NORB networks. From this it can be concluded that when the neural network has less floating point operations than this network it is very probable that the CPU platform will perform better than the GPU platform, and vice versa.
Chapter 7
Online versus Batch Training

As mentioned in the introduction of this work there is another dimension to the trade-off space for neural network training besides the architecture and neural network, namely the training algorithm. Variations like particle swarm optimization and differential evolution that showed promising when tested on small networks like the family tree neural network (introduced in the book Cognitive Modeling by T. Polk et al. [20]) proved to scale badly when using larger networks like convolutional neural networks due to the large solution space.

Another algorithmic variation on the standard back-propagation algorithm is often discussed in recent publications, namely batch training. This variation looks promising because it introduces more parallelism in the training process and therefore gives rise to more efficient use of the target platforms processing power and possibly faster training of the network.

7.1 Introduction to Batch Training

The trade-off between two approaches to the training of neural networks using the back-propagation algorithm is also explored using the optimized implementation for the CPU platform. So far the online approach has been used to train neural networks. This means that the weights and bias values of the neural network are updated every time a single image is propagated forward and backwards through the network. Another variation is batch training, where the weight changes are accumulated over and the presentation of a batch of input images of the training data before being applied to the network. When this batch consists of a single image the training corresponds with online training, and when this batch consists of all images in the training set it corresponds with full batch training. A batch size between these extremes is mini-batch training [22], providing a medium between the two previous approaches.

Online training uses the local gradient of each training vector to determine what direction the gradient should go, but these local gradients can incur noise and contradictions. The noise can help overcome local minima, but the contradictions can cause a negative effect on the amount of iterations needed for the training of the network.

(Mini-)Batch training is able to compute the direction of the true gradient more accurately because it uses a bigger training set than online training (which only uses a single image) to update the weights. The downside is that the weight adjustments tend to grow large when the size of the training set increases, and this means the updating of the weights occurs in big steps causing unstable learning and overshooting of (local) minima. Batch learning might require more iterations to reach the desired error minimum, but it is more fit for parallel execution since multiple images can be fed through the network. Only the accumulation of the weight updates and the actual updating has to be done sequentially. It is often said batch training is better for neural network training than online training since it follows the true gradient direction instead of lurching around the gradient direction like with online training. [21]

Online training is able to give a more accurate estimation of the gradient direction and the
expected value of the weight adjustment is in the direction of this gradient, whereas for batch training it is in the direction of the gradient at the beginning of the epoch. [29] This means online training is very likely to train in less epochs than batch training due to a more guided training which also shows from the paper of Wilson et al. The question that is answered in the following sections is whether batch training can compete with online training when looking at the total training time since it introduces more parallelism.

7.2 Implementing the Batch Training

Studies on small scale networks have shown that batch training can have a positive effect on the network training time. However for large scale networks this is not necessarily the case, and it would be a valuable addition to review the results of this trade-off for the work in this report.

The paper of D. Randall Willson and Tony R. Martinez shows an overview research publications with respect to the trade-off between batch and online training. [29] The bottom line of this paper is that batch training is very unlikely to converge faster than online training for small scale networks, and for large scale networks the batch training required much more training iterations to reach the same results as the online training.

![Algorithmic comparison of the online and batch training methods](image)

Implementing the batch training requires a couple of small changes compared to the online training method. Figure 7.1 shows the difference between both training methods. The online training method feeds an input image through the network and updates the network weights based on the computed output, local gradients and weight updates of this input image. After this the next image is fed to the network. For our platform a mini-batch size of four is chosen because this maps best to the target CPU architecture since it has four cores that can each execute the forward convolution, computation of the output error and local gradients for one of the input images. After this the weight updates are accumulated and applied to the network, after which four new input images can be fed to the network, repeating the process.

The mini-batch approach is implemented using OpenMP which is able to divide the individual paths for each of the four input images over the four physical cores of the CPU architecture. This required introducing a small overhead compared to the online training method, because every core requires a private copy of the neural network parameters (consisting of the network weights, bias values, network structure, etcetera) to prevent multiple cores writing to the same address. On the
other hand the batch method only requires updating of the weights once per four images, where the online method needs to update the weights for each input image. The effect of this however is probably minimal, because the convolutional neural network uses shared weights for feature maps, reducing the number of weights that need to be updated.

7.3 Batch Training Results

The approach of the experiments to compare the online and batch training methods is as follows. First the speed sign recognition and small-NORB object recognition neural networks were trained using the online method with a certain learning rate $r$. A lot of pre-processing steps like adapting the intensity of the images were omitted to speed up the training process and since obtaining a very high network performance is not the goal here it is permitted. Only the pre-processing step of shifting the input images is used to be able to get an acceptable performance after the network training.

Subsequently the same neural networks are trained using the batch training method with two different learning rates:

- Learning rate: $r$ (equal to the learning rate with the online method)
- Learning rate: $r/4$

When the batch approach uses the same learning rate $r$ it should be able to learn four times as fast per epoch compared to the online method, and the total training time should be about four times faster than the online training method. When the batch approach uses a learning rate of $r/4$ it should follow the same trajectory as the online approach per epoch, since it receives an equal update but only from four input images. This should only make the training curve smoother compared to the online curve because it averages the weight updates over four images instead of one.

The batch approach processes four times the number of input images per epoch compared to the online approach, but because most of the work can be divided over four threads it should be able to process the mini-batch of four images in the same time the online approach processes a single input image (assuming ideal parallel execution). But like it is with most assumptions this is not the case when actually implementing the algorithms. The batch method has extra overhead for copying data, and still has the updating of the weights that has to be executed sequentially on a single core.

Measurements show that a single epoch of the online method is a little bit faster than a single epoch of the batch method with four images, so some performance is already lost here. This is mostly caused by the fact that the convolutional networks used for this research are relatively big and contain a lot of parallelism as is (i.e. inside the feature maps). This parallelism is exploited in the online implementation by dividing feature maps over different CPU cores, and when applying the batch method this is not usable anymore because the cores are already utilized. This means that the sequential part of the forward convolution to the computation of local gradients is slower than with the online method.

The training results are shown in Figure 7.2(a) and 7.2(b) show the Sum Squared Error (SSE) of the output and the amount of misclassifications per epoch of training for the small-NORB network, and Figure 7.3 shows the same for the speed sign recognition network. The training times of the networks in combination with the different training methods are shown in Table 7.1.
Figure 7.2: Results of the online and batch training approaches for the small-NORB object recognition network

Figure 7.3: Results of the online and batch training approaches for the speed sign recognition network

<table>
<thead>
<tr>
<th>Neural Network</th>
<th>Training Method</th>
<th>Learning Rate</th>
<th>Training Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>small-NORB</td>
<td>Online</td>
<td>$r$</td>
<td>34m48s</td>
</tr>
<tr>
<td></td>
<td>Batch</td>
<td>$r$</td>
<td>30m54s</td>
</tr>
<tr>
<td></td>
<td>Batch</td>
<td>$r/4$</td>
<td>2h2m11s</td>
</tr>
<tr>
<td>speed sign</td>
<td>Online</td>
<td>$r$</td>
<td>26m29s</td>
</tr>
<tr>
<td></td>
<td>Batch</td>
<td>$r$</td>
<td>25m55s</td>
</tr>
<tr>
<td></td>
<td>Batch</td>
<td>$r/4$</td>
<td>1h51m36s</td>
</tr>
</tbody>
</table>

Table 7.1: Training times for the online and batch training methods

Figure 7.2 and 7.3 show the same kind of curve for the training process when looking at the sum squared error. When looking at the number of misclassifications there is a difference in the graphs. With the small-NORB network all images are misclassified with an untrained network, where with the speed sign recognition there are some correct classifications with the untrained network and when training this number grows larger before converging to a minimum.

The small-NORB network is trained until a maximum of 15% misclassifications on the test set had been reached. This is not the maximum achievable performance but makes it possible to compare the online and batch methods while keeping the training time relatively low. The performance can be further improved by using more pre-processing steps next to shifting the input.
image and training for a longer time. The speed sign recognition network is trained longer (in terms of epochs) because it is faster than the small-NORB network per epoch (see earlier chapters for the proof of this, for example Section 2.3). Here the network is trained until a maximum of 5% misclassifications on the test set had been reached.

The results show that the online and the batch with learning rate \( r/4 \) curves follow the same trajectory when looking at the number of misclassifications and the SSE. The curve of the batch with learning rate \( r/4 \) is also a bit smoother than the online curve, because it averages the weight updates over four input images where the online method updates for every input image. However when looking at the training times in Table ?? it seems for both networks that the batch method with learning rate \( r/4 \) is about four times slower than the online method. The only way batch training would pay off with respect to the training time is when it uses less epochs to train (since it can only achieve the same speed per epoch as online learning). This can be done by increasing the learning rate for the batch method.

No extensive study is done in this work to see what the highest possible learning rate for both the online and batch method is while still achieving high classification performance of the network. However in the paper of Wilson et al. [29] the learning rate \( r \) used for the online method is reduced by a factor \( \sqrt{N} \) for the batch method, processing \( N \) images in per mini-batch. This is because some of the individual weight changes cancel each other out with online learning, so the learning rate for batch does not have to be a full \( N \) times smaller than online (as used in this research). This will cause the batch method to converge faster than the online method (in fewer epochs) and the possibility arises that the batch approach would be faster than the online learning approach.

In this research the mini-batch approach of size 4 is used with learning rate \( r/4 \), meaning that according to the paper of Wilson et al. the learning rate could be \( r/\sqrt(4) = r/2 \). It has been noted that a learning rate of \( r/4 \) and \( r \) for the mini-batch training scaled linearly, where a learning rate of \( r \) converged four times faster than the learning rate of \( r/4 \). When a learning rate of \( r/2 \) would be used this would thus still be a factor two slower than the online training method for the networks studied in this research.

It is also possible to draw a conclusion with respect to batch training on the GPU platform. When using the mini-batch training method with a mini-batch size of four on a single GPU the time spent per epoch can at most be equal to the online method (since the mini-batch method processes four times the amount of input images per epoch and parallelizes the mini-batch over four paths). This is under the assumptions that there are enough processing elements available to perform the forward convolution, computation of the output error and the local gradients in parallel. Besides there should be enough bandwidth to the off-chip memory to transfer four times the amount of data of the online method to be able to achieve the same performance. If all this is the case the performance of the batch method will be more or less equal to that of the online training method, since it is already shown that the convergence of the batch method is not faster than the online approach but only results in a smoother learning curve. Here the same story holds as for the GPU when assuming the learning rate behavior of the paper of Wilson et al. where choosing a learning rate of \( \sqrt{N} \) with a mini-batch size of \( N \) will most probably not be able to improve on the online training performance. Unless a larger mini-batch size is chosen and the learning rate is further increased the mini-batch approach will be slower, and increasing the mini-batch size means the implementation will be very likely to run into either of the mentioned bottlenecks.

Another possibility is the use of a cluster of GPUs and divide the mini-batch over different GPU nodes. This is what has been done in the work of J. Ward et al. [27] where a speedup of 23% is achieved by using the mini-batch method with 4 GPUs compared to the online method implemented on a single GPU for the small-NORB network training). Here they compare ten input images per epoch for both methods, which is not a fair comparison since the batch method should have four times as much input images per epoch and a learn rate of \( 1/\sqrt(4) \) times the online learning rate to be able to compare the results from this study for the CPU.

To be fair the execution time of the batch results from this study should be multiplied by two (half the online learning rate and four times the number of input images yields a factor two). This would mean that the online training on a single GPU is still faster since the batch method is only
23% faster, which means that also in this case online outperforms the batch method on a GPU cluster with mini-batch size four.

The learning rate could become larger when a larger mini-batch size is used (for example 8, 16 or 32). As mentioned this is not possible to implement efficiently on the CPU due to the limit of four physical cores on the platform, and probably not achievable on a single GPU due to the PE and bandwidth bottlenecks. It is possible using a big cluster of GPUs where it is much less likely to run into the single GPU bottlenecks, however it is very likely that the communication between these GPUs will become the bottleneck since the weights of all members in the mini-batch have to be transferred to a single GPU, accumulated, applied to the network and distributed to all GPUs again.
Chapter 8

Conclusions, Related work & Future work

Concluding this research the work itself is shortly summarized in Section 8.1 with special attention for the conclusions and contributions. The contributions of this work and how these fit with the related research is discussed in Section 8.2. Finally suggestions for future work based on this work are given in Section 8.3.

8.1 Conclusions & Contributions

The basis of this work is the optimization of the training for large scale neural networks such as convolutional neural networks and exploring the trade-off space that determines the performance of this training process. Despite of the hardware progress over the last decades the training of large scale neural networks is still mostly limited by computational speed of modern processors. Because of limited parallelism during the training of these networks caused by sequential behavior of the generally used back-propagation learning algorithm massively parallel architectures such as GPUs are not fully exploited.

Despite the many dependencies that limit the performance of the training process, the convolutional neural networks studied in this work achieve a performance increase of 5X compared to the original C implementation. During the mapping it became clear that the GPU is far from optimal for different neural networks. For networks with a reasonable amount of parallelism such as the small-NORB object recognition network the GPU proves to be a small improvement over an optimized CPU implementation with respect to the training time. However if the network has limited parallelism (mostly caused by fewer and smaller feature maps inside the network layers) the optimized CPU implementation easily outperforms the GPU implementation.

It is desirable to be able to accurately predict the performance of the training process. Motivations are that it is useful to see which platform should be chosen to train a certain network in the shortest time, and to be able to see how small changes in neural network structures can influence the training performance. The roofline model, usually able to predict the performance of small kernels with great accuracy on different platforms, proved to be too inaccurate to predict the performance of a complicated application such as the neural network training algorithm. Extending the roofline model with additional ceilings is also not an option since the training application is dynamic, which would require different ceilings for every kernel in the application.

Analysis of the platforms used in this work with respect to the training application showed the cause of the poor performance of the neural network training. The application required a large amount of load and store operations which proved to be a vast problem on the CPU architecture which can only handle one move operation per clock cycle. The GPU implementation is bound by the large amount of unaligned load operations which are inevitable due to the irregular access pattern of the application making aligned memory accesses infeasible. The small amount
of parallelism in the training application also prevents the use of the entire compute capabilities of the massively parallel architecture resulting in a loss in performance.

After studying these bottlenecks it is possible to compose prediction models to improve on the accuracy of the roofline model. The CPU performance prediction model is based on the single move operation per cycle limit. By analyzing the instructions generated by the application and extracting the neural network information from the input files, the amount of cycles required for training can be estimated using a analytical model. The GPU performance prediction model uses the same approach as the CPU prediction model, but for this scenario a profile-based approach is used for more accurate results.

The results of both prediction models is tested on different platforms and proved to be much more accurate than the roofline model for the training application. The prediction error is at most 20% where the roofline model showed an error of much greater magnitudes.

Using these prediction models a design space exploration is presented, studying the impact of different platforms and neural network configurations on the performance of the training process. The results show that choosing the right platform for training a certain network can results in a 2-4X increase in the training performance. The choice of neural network structure can also influence the training performance significant as shown by the results. The results also show a break even point where the CPU and GPU architecture have similar performance, indicating that smaller neural networks will most probably perform better on a CPU platform and vice versa.

Finally the trade-off between the online and the batch training method is studied. Experiments show that using the batch method can not achieve faster network training than the online method when maintaining the learning speed per epoch unless the batch method is able to increase the learning rate over the online method whilst still achieving the same classification performance. Although no extensive research is done in this work with respect to the learning rate that can be chosen for both methods, the approach for choosing the learning rate for the batch method from the work of Wilson et al. is adopted. From this it is possible to formulate several conclusions with respect to batch training. The experiments show that batch training cannot outperform the online method on a CPU platform. It is reasoned that batch training for a single GPU platform will show a similar trend where batch is very unlikely to outperform the online method. The only way batch training can result in a better performance compared to online training would be on a large compute cluster such as a cluster of GPUs, which would allow a greater learning rate for batch training due to the possibility of a larger batch size. However the use of such compute cluster introduces a new problem which could likely form a bottleneck, namely the communication cost of data between the cluster nodes.

With the set of tools developed in this work it is possible to explore different neural network structures and train larger networks by using the hardware more efficiently. This gives rise to easier design choices for designers to develop such neural networks and the development of more applications to help us in our everyday life, for example in the medical and automotive fields.

8.2 Related Work

There are several publications that focus on implementing the training process of large scale neural networks on a parallel platform like a GPU or multi core CPU like in this work. An interesting publication implementing neural network training on a CPU platform is the work of Alfred et al. [1] where RBF neural network training is performed using SIMD parallelism, and the performance of MMX and SSE (Intel), 3DNow! (AMD), VIS (Sun) and Altivec (Motorola) is compared which shows promising results. The neural network structure used for this work is different, but it is still interesting to see how different types of SIMD parallelism are exploited on different architectures.

In the work of D.C. Cireşan et al. [3] the training of a CNN network is implemented on a GPU platform. The results were very good, with a measured speedup of more than a factor 60 over a compiler-optimized CPU implementation. However there are two big differences with the approach in this work. Firstly the neural network used in the work of D.C. Cireşan et al. is much bigger than the networks used in this study, containing a few hundred feature maps per network layer.
This creates a huge amount of parallelism that can be exploited by the GPU architecture, but will slow down the learning of the network due to the large amount of work to be done. Secondly the GPU implementation is compared to a compiler-optimized CPU implementation, whilst in this work it is shown that a compiler-optimized version leaves a great deal of parallelism unused due to the complexity of the application.

The roofline model introduced by Williams et al. [28] is widely used for performance prediction for small applications. After evaluation it is not practical to use the roofline model to predict the performance of a complicated application such as neural network training as shown in Section ?? in this report.

There are publications that analyze the performance of the neural network training process, but these mostly cover differences between learning algorithms or other influences on the training process. However there are no publications discussing the trade-off space between different platforms with respect to the training time for large-scale neural network training like in this work using the performance prediction models.

The online versus batch training dilemma is still an ongoing battle in the literature. Many researchers are of the opinion that batch training is as fast or faster and more stable than online training (For example in the work of J.C. Principe et al. [21]). As mentioned the paper of D. Randall Willson and Tony R. Martínez [29] shows an overview of current views on online and batch training. The bottom line of this paper is that batch training is almost always slower than online training, and especially when the training incurs a large training set. This work shows that the batch method is indeed very unlikely to train CNNs faster than the online method unless a large amount of compute power is used like in a GPU cluster.

Also optimized implementations of batch training can be found in the literature. The work of V. Tsaregorodtsev [25] implements the back-propagation algorithm with a batched update approach implemented on a multi core CPU. However the result from this work is that the batch implementation utilizing multiple cores is faster than the single-threaded batch implementation which is trivial, and no comparison with the online method has been made.

The work of Ward et al. [27] implemented the batch training method for the small-NORB object recognition network and denoted a speedup of 23% over the online method using a mini-batch size of four. However when using the honest method of comparing batch and online like in this work, the batch version uses a smaller learning rate and learns slower than in their work, making the online method perform better than the batch method.

8.3 Future Work

To be able to get better training performance newer generations of hardware platforms should be supported by the models presented in this work. Since the models are designed to be easily extensible for new hardware platforms (with new performance properties and possibly other bottlenecks for the training application) this should form no problem. However future platforms cannot be taken into account in this work and form a part of future work to extend the models.

Downside of the GPU model in this work is that the it requires a new trace analysis for each different neural network structure, requiring a little more time than the offline method of the CPU model. This can be improved in future research by estimating the instruction information straight from the application (using an analytical approach) without the use of tools like GPUOcelot like with the CPU model, since this step required too much time to wrap up in this work.

Alternatives for or modifications of the typically used back-propagation algorithm still are a very interesting research area for training of large scale neural networks. The particle swarm optimization and batch training method were treated in this work, but there is a large amount of algorithmic modifications and alternatives available that could prove to be an improvement over the regular online back-propagation algorithm. An example that can prove to be of improvement by applying a small algorithmic change to the algorithm is the work of H. Larochelle et al. [14] where the weights of each network layer is pre-trained instead of randomly initialized before training. To evaluate if the optimization process in this work can also have a positive effect on the pre-training
step of this algorithm future research is required.

The conclusions of online versus batch training in this work are mostly based on the assumption that the learning rate for batch can be a factor $\frac{1}{\sqrt{N}}$ times higher than with online training with a batch size of $N$ images, which is taken from the work of Wilson et al. [29]. Based on this the batch method can only train faster than online using massive compute power like a GPU cluster. To prove this additional research is required to determine the relationship between the batch size and learning rate with respect to the classification performance of the network and the training time.

The conclusions that batch training is likely not a big improvement over the online approach can also make one think about how biological neural networks learn. For example it is not likely that people learn in a batched fashion where the networks behavior is adapted after getting multiple learning impulses, but in a more sequential way.
Bibliography


63


