Linnæus University
Sweden

Degree project

Accelerated Deep Learning using Intel Xeon Phi

Author: André Viebke
Supervisor: Sabri Pllana
External Supervisor: Narges Khakpour

Date: 2015-07-15

Course Code: 4DV01E, 15 credits
Level: Master

Department of Computer Science

Abstract

Deep learning, a sub-topic of machine learning inspired by biology, have achieved wide attention in the industry and research community recently. State-of-the-art applications in the area of computer vision and speech recognition (among others) are built using deep learning algorithms. In contrast to traditional algorithms, where the developer fully instructs the application what to do, deep learning algorithms instead learn from experience when performing a task. However, for the algorithm to learn require training, which is a high computational challenge. High Performance Computing can help ease the burden through parallelization, thereby reducing the training time; this is essential to fully utilize the algorithms in practice. Numerous work targeting GPUs have investigated ways to speed up the training, less attention have been paid to the Intel Xeon Phi coprocessor.

In this thesis we present a parallelized implementation of a Convolutional Neural Network (CNN), a deep learning architecture, and our proposed parallelization scheme, *CHAOS*. Additionally a theoretical analysis and a performance model discuss the algorithm in detail and allow for predictions if even more threads are available in the future. The algorithm is evaluated on an Intel Xeon Phi 7120p, Xeon E5-2695v2 2.4 GHz and Core i5 661 3.33 GHz using various architectures and thread counts on the MNIST dataset.

Findings show a **103.5x, 99.9x, 100.4x** speed up for the large, medium, and small architecture respectively for 244 threads compared to 1 thread on the coprocessor. Moreover, a **10.9x - 14.1x** (large to small) speed up compared to the sequential version running on Xeon E5. We managed to decrease training time from *7 days* on the Core i5 and *31 hours* on the Xeon E5, to *3 hours* on the Intel Xeon Phi when training our large network for 15 epochs.

Sammanfattning

deep learning, ett delämne inom maskininlärning, har på den senaste tiden fått stor uppmärksamhet både i näringslivet och forskarsamhället. *State-of-the-art* applikationer inom *computer vision* och *speech recognition* är byggda på *deep learning* algoritmer. I jämförelse med traditionella algoritmer där utvecklaren instruerar vad som ska utföras i detalj, lär sig *deep learning* algoritmer genom erfarenhet att uträtta uppgiften som den ska bli bra på. Hursomhelst, för att algoritmen ska kunna lära sig krävs mycket träning vilket är en stor beräkningsbörda. *High Performance Computing* kan minska bördan genom att parallellisera träningen och därmed minska träningstiden; vilket är vitalt för att kunna använda tekniken fullt ut i praktiken. Flertalet arbeten undersöker potentialen av grafikprocessorer (GPU), men få har undersökt Intel Xeon Phi.

I denna tes presenterar vi en parallell implementation av *Convolutional Neural Network (CNN)*, en *deep learning* arkitektur, och vår förslagna paralleliseringsschema, *CHAOS*. Därutöver presenteras en teoretisk analys och en *performance model* som tillåter diskutera algoritmen och grovt uppskatta exekveringstiden om fler trådar skulle bli tillgängliga i framtiden. Algoritmen utvärderas på Intel Xeon Phi 7120p, Xeon E5-2695v2 2.4 GHz och Core i5 661 3.33 GHz genom olika arkitekturer och antal trådar på datamängden MNIST.

Resultaten påvisar en **103.5x, 99.9x, 100.4x** *speed up* för stor, mellan och liten arkitektur respektive, för 244 trådar jämfört med 1 tråd på Xeon Phi. Dessutom **10.9x - 14.1x** (stor till liten) *speed up* i jämförelse med den sekventiella versionen på Xeon E5. Vi lyckades minska träningstiden från *7 dagar* på Intel Core i5 till *31 timmar* på Xeon E5, och *3 timmar* på Intel Xeon Phi när vi tränade vårt stora nätverk 15 epoker.

Keywords: Machine Learning, Deep Learning, Supervised Deep Learning, Intel Xeon Phi, Convolutional Neural Network, CNN, High Performance Computing, CHAOS, parallel computing, coprocessor, MIC, speed up, performance model, evaluation

Preface

First of all I would like to express my thankfulness for the opportunity to perform this project. I would like to share my gratitude with my supervisor, Sabri, who helped guide me throughout the project. I also want to thank my future wife for enduring the late evenings and weekends I spent on this project.

Both deep learning and Intel Xeon Phi were topics completely new to me before initiating this study. It is now afterwards I recognize what I have missed out by not study them earlier. I am convinced that deep learning algorithms will drive the future of applications. As these applications require computational resources, there is no doubt such algorithms need to be parallelized or otherwise make use of high computational platforms to meet their computational needs.

We began this project with a wondering: how can Intel Xeon Phi facilitate deep learning Algorithms? Although several papers have been published during the last couple of years targeting GPUs, few have been published for the Intel Xeon Phi. This wondering is what initiated, and drove this study.

Förord

Först och främst vill jag framföra min tacksamhet för att ha fått möjlighet att utföra detta projekt. Jag vill tacka min handledare Sabri, som vägledde mig genom projektet. Jag vill också tacka min framtida fru för att stått ut med långa kvällar och helger som spenderats på projektet.

Både *deep learning* och *Intel Xeon Phi* var ämnen helt nya för mig innan jag påbörjade arbetet. Det är först nu efteråt som jag förstår vad jag missat genom att inte studera dem tidigare. Jag är övertygad om att *deep learning* kommer att vara en del av framtidens applikationer. Eftersom dessa kräver mycket beräkningskraft råder det inget tvivel om att dessa behöver parallelliseras eller på något sätt använda sig av plattformar med hög beräkningskapacitet som möter deras behov.

Vi började projektet med en undran: hur kan *Intel Xeon Phi* underlätta *deep earning* algoritmer? Mycket material har publicerats relaterade till grafikprocessorer (GPU) men väldigt få för *Intel Xeon Phi*. Denna undran initierade, och drev vår studie.

Publications

A. Viebke and S. Pllana, "The potential of the Intel Xeon Phi for supervised Deep Learning," in IEEE HPCCC 2015. New York: IEEE, 2015.

Contents

List of Figures

List of Tables

Listings

Glossary

A short glossary with terms used in the study.

- **Machine learning** - A group of algorithms learning from experience in performing a task.
- **Deep neural network (DNN)** - Non-linear, complex, hierarchic models inspired by biology solving complex problems in computer science.
- **Convolutional Neural Network (CNN)** - A specific type of DNNs introducing convolutional and pooling layers inspired the behaviour of the visual cortex of animals.
- **High Performance Computing (HPC)** - The use of parallel processing power beyond the TFLOPS boundary facilitating high computational loads.
- **Intel Xeon Phi** - A HPC device from Intel with up to 61 cores (Knights Corner) and in total 244 threads performing 1.2 TFLOPS.
- **VPU** - Vector Processing Unit is a part of modern processing units allowing several scalar operations to be merged into vector operations and thus lower the number of instructions (and cycles) required.
- **OpenMP** - An API/library facilitating the use of thread-, and data-parallelism by using pragmas and library routines, removing the necessity for developers to manage threading and SIMD instructions manually.
- **Speed Up** - The fraction of how much faster the parallel version is compared to a base lined version.
- **Data- and model-parallelism** - Two concepts used in parallelization of CNNs to define how the problem is divided over threads in the host system. Data-parallelism denote that the problem is divided by the input space, as in the case of this study. Model-parallelism define that several workers work on different parts of the model. Data-parallelism can also refer to SIMD instructions in some literature. To avoid confusion we will denote SIMD instructions as SIMD parallelism, however the context will also determine the semantics.
- **Processing Unit** - A processing unit is an electronic circuitry that carry out arithmetic, logical, control and I/O operations. In this study we generally used the term processing unit incorrectly to mean hardware threads, however, in many cases the number of hardware threads equals the number of processing units, i.e. cores on the coprocessor.
- **Performance Model** - A performance model is a model expressing the characteristics of a system in terms of resources consumed, resource contention, delays, etc. Its main purpose is to model a realistic system in order to provide insight of how a proposed system may behave.
- **Theoretical Analysis** - A way of expressing the behaviour of an algorithm theoretically. That is, how the algorithm is expected to behave in different configurations.

Chapter 1

Introduction

Traditional applications are created by engineers who strictly feed the computer with instructions. Throughout the history of computers, the abstraction level have shifted from hardware-near to higher level languages eliminating details from the engineer. Nevertheless, it is still the engineer who instructs the computer what actions to take, although through a higher level of abstraction.

Deep learning algorithms learn from their own experience rather than that of the engineer. The necessity of programmers micro-managing the computer becomes less relevant, instead engineers focus on developing and implementing sophisticated deep learning models that are able to *learn* to solve complex problems.

We find these techniques intriguing, their characteristics offer a new way to think of software engineering. Therefore we decided to investigate them further, focusing on how to lower the learning time. This thesis introduces the reader to the concepts of deep learning and the Intel Xeon Phi. In the study we design, implement and evaluate a parallelization scheme applied to a CNN. Moreover, a theoretical analysis is carried out and a performance model is developed and evaluated.

This chapter is organized as follows. First deep learning and the Intel Xeon Phi are introduced. Thereafter the motivation, problem, goals and research question are discussed. Lastly, the approach is described together with the contributions and limitations of the study.

1.1 Deep Learning and the Intel Xeon Phi

Machine learning is the science of making computers perform actions without explicitly programming them. By teaching computers how to perform a task they can gain experience and improve their skills in performing it [6].

Deep learning is a sub-field of machine learning which in contrast to machine learning comprises multiple layers of representations. In essence stacking of multiple layers allows deep learning algorithms to solve more complex problems than those of shallow, traditional machine learning algorithms [7]. These deep hierarchical models are inspired by the visual cortex of mammals where higher level of representations are based on lower representations [8].

A core component of deep learning, Deep Neural Networks, or DNNs, is the underlying representation model of deep learning algorithms. The first network that deserves the name deep was developed by Fukushima in 1979, which interestingly was a CNN, however, neural networks dates back even further. It was first in 1981 (Werbos et al.), the first efficient implementation of the important back-propagation algorithm was applied to

neural networks. Later, in 1989 LeCun et al. applied the techniques to CNNs and in 1998 published the famous LeNet [9].

Deep Neural Networks (DNNs) can be visualized as weighted graphs as depicted in *Figure 1.1.1*. In a nutshell DNNs are able to make predictions by forward propagating an input through the network. After a forward pass through the network, the output layer contains a vector comprising the prediction. For instance, an image forwarded through the network results in a vector comprising classifications at the output layer [10].

Back-propagation is the process of learning the network. Back-propagation is a technique used in supervised learning to update the weights of the network based on the calculated loss in the forward pass. The ultimate goal of learning is to optimize the network such that the deviation between the prediction of each sample seen so far, and the desired output (label) of that sample is as low as possible [11].

In this study we focus on supervised deep learning of CNNs using the back-propagation algorithm. Supervised learning uses large labelled datasets to train the network. Each prediction is compared to the label and weight parameters are adjusted according to the errors in prediction [9].

A Convolutional Neural Network (CNN) is a variant of a DNN introducing two new layer types: *convolutional-* and *pooling-layers*. Inspired by the visual cortex of animals, CNNs are applied to state-of-the-art applications, including computer vision and speech recognition [7].

A performance model provides insight of how a system is expected to perform in terms of resources consumed and contentions. A performance model can model the intended behaviour of a system without the system at hand [12].

Theoretical analysis aims to predict the resources required by an algorithm. The results of a theoretical analysis allows to express and discuss the algorithm [13].

High Performance Computing (HPC) and larger datasets have paved the way for the success of deep learning algorithms. Although the current hype, not much work target the Intel Xeon Phi coprocessor. The Intel Xeon Phi comprises up to 61, 1.2 GHz cores, and each core can switch between 4 hardware threads in a round-robin manner. Even if each core has its own Level 2 cache, cores can share cache data internally. Therefore the coprocessor can be thought of a shared-memory, many-core coprocessor [14]. The Intel Xeon Phi used in our experiments is of type 7120p.

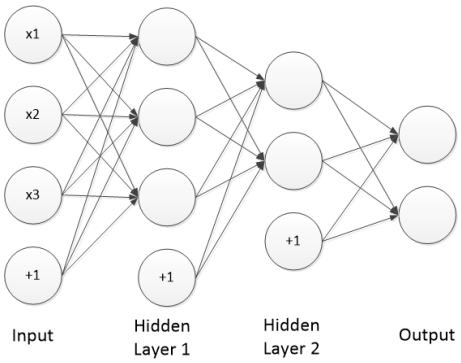

Figure 1.1.1: A DNN.

Deep learning have emerged from universities to industry in a rapid pace. Applications of deep learning can be found everywhere, one example of a deep learning application is the Android voice recognition [15]. Other applications include face recognition, self-driven cars, predicting liver diseases, image classification, and Skype Translate [16, 17, 18, 3, 15] - and the future will certainly yield even more.

There is, undoubtedly, a growing interest for deep learning, several large companies, including Google, Microsoft and Nvidia have announced their interest. Google acquired DeepMind in 2013 [19], a company focusing solely on deep learning. Researchers at Google also participated (and won some of the sub-challenges) in the latest ImageNet competition (2014) [20]. Microsoft perform research in the area as well. In an article published in February 2015 (this year), researchers at Microsoft claim that they outperformed a human on the 2012 version of the ImageNet dataset [21]. At a conference in March 2015 the co-founder of Nvidia, Jen-Hsun Huang, presented their new gears in conjunction with deep learning and CNNs [22]. There is no doubt deep learning is a hot topic in computer science.

Supervised learning is computational costly and time consuming - in many cases several weeks are required to complete a training session; large neural networks comprise millions, if not billions of computations. Unfortunately, large delays in training highly limit their usage in practice as many parameters often needs to be tested, and each test requires a full session of training. Intel bring High Performance Computing to consumers in form of the *Intel Xeon Phi* coprocessor. Numerous work previously target Graphical Processing Units GPUs, e.g. [23, 24], fewer target the coprocessor, leaving the potential of the *Xeon Phi* unexplored. Adopting deep learning algorithms for the *Xeon Phi* could decrease the training time from weeks to days, or from days to hours, allowing the coprocessor to be considered for supervised deep learning. Moreover, this allows consumers already invested in a *Xeon Phi* to use it for deep learning applications.

The main goal of this thesis is to investigate and position the *Intel Xeon Phi* in the context of supervised deep learning in general and CNNs in specific. To achieve this, the following sub-goals have been identified,

- Find an appropriate implementation targeting an unexplored deep learning algorithm (later narrowed down to CNNs);
- Design a parallelization scheme and adopt it to the selected implementation;
- Perform evaluation, collect data required for analysis and analyse the results;
- Perform a theoretical analysis, in a hardware independent model, to understand and describe the algorithm. The theoretical analysis should help understand the resources required by the algorithm and what theoretical bounds that can be expected. E.g. how much work is required to train the network and how much faster (in terms of execution time) is the algorithm expected to perform when using multiple cores instead of one core;
- Design and evaluate a performance model extending the theoretical model (as derived in the theoretical analysis) by including hardware characteristics. A performance model answers *what-if* questions with respect to the number of threads that goes beyond the number of hardware threads supported by the coprocessor. The performance model can also be used to predict the execution time for varying number of inputs, epochs and architectures;

This thesis will answer the research question:

RQ1: What is the potential of Intel Xeon Phi for supervised deep learning algorithms?

In this thesis we design a parallelization scheme, named *CHAOS*, which we adapt to be used with an existing implementation. *CHAOS* is evaluated experimentally on the MNIST [25] dataset using varying CNN architectures and thread counts. The evaluation is performed on three different platforms. We thoroughly analyse the algorithm theoretically and conclude its theoretical bounds. The theoretical model is further extended into a performance model accounting for hardware characteristics of the coprocessor. The performance model is evaluated on the coprocessor and used to predict for varying thread counts beyond that of the Intel Xeon Phi.

Our main contributions of this thesis include,

- our parallelization scheme *CHAOS* for the Intel Xeon Phi;
- experimental evaluation of the implementation using the MNIST dataset;
- development and validation of a corresponding performance model;

The scope of this study is limited to deep learning in general and CNNs in specific. For the evaluation we used one dataset (MNIST), one implementation yet several CNN architectures, platforms and thread counts. The reader should be aware of that the study is an empirical evaluation of deep learning algorithms, and do not intend to re-implement the algorithm for the coprocessor. Neither is the intention to cover all deep learning algorithms or datasets available. The project is time-boxed, time is a vital factor that cannot be neglected - training networks takes severe amount of time.

We limit ourselves to Intel Xeon Phi even though optimizations implicitly apply to other Intel architectures as well, however, no comparison will be done for GPUs (or other competing architectures). The main intentions of the theoretical model and the performance model are to discuss the algorithm and may therefore have some limitations in practice.

The thesis is intended for anyone interested in deep learning and High Performance Computing (HPC) in general. Readers searching for performance results related to deep learning and the coprocessor should especially be interested in this study as it contributes to an otherwise sparse set of work in the area. Knowledge of basic concepts in computer science are expected, an introduction to the important topics is carried out in *chapter 2* . Readers new to deep learning and/or Intel Xeon Phi can consider this thesis a nice introduction to the topics. Researchers in the area should especially find the empirical results, the parallelization scheme *CHAOS* and the performance model to be of interest. We also recognize the industry as a target group of this work as more applications use deep learning algorithms in business, and companies may already been invested in a coprocessor.

1.2 Related Work

This chapter discusses previous research related to machine learning, deep learning and CNNs. Additionally, work targeting Intel Xeon Phi outside the scope of machine learning, deep learning and CNNs is also included. Overall, we find the research for the coprocessor and deep learning to be sparse, and that of GPUs to be dense.

Error rates define the number of images incorrectly predicted by the network and helps conclude the prediction accuracy of the network.

One epoch is an iteration of training, networks are trained for a set of epochs. In each iteration all inputs of the training dataset is considered once by the network.

1.2.1 Datasets in Related Work

NORB [26] contains 50 toy objects in 5 categories captured in different conditions and angles. Its main intention is for 3D object recognition. CIFAR 10 [27] consists of 60,000 colour images divided into 10 classes. The MNIST [25] dataset of handwritten digits contains 60,000 training images, and 10,000 testing images. The ImageNet [20] dataset contains over 15,000,000 images divided in 22,000 categories. The number of images and categories increase by the year, earlier competitions may therefore have a smaller set of image (1.2 million and 1,000 categories in the set at 2010).

1.2.2 Applications of Intel Xeon Phi

Previous work related to machine learning for the coprocessor is sparse if compared to other architectures such as GPUs. However, progress have been made for both deep- and shallow-models. Shallow models have shown to solve simple well-contained problems. However, complicated problems cause difficulties for shallow models, and many-layered (deep) models (e.g. deep neural networks) have shown more successful for the task. Whereas traditional models comprise one or two layers, deep models comprise multiple layers [7].

In this chapter we introduce previous work for Support Vector Machines (SVMs), Restricted Boltzmann Machines (RBMs), sparse auto encoders and the Brain-State-in-a-Box (BSB) model. In addition some related work outside the context of deep learning is included.

In [28], Yang You et al. present a library for parallel Support Vector Machines (SVMs), MIC-SVM, which facilitates the use of SVMs on many- and multi-core architectures including Intel Xeon Phi. Previous SVM libraries target serial execution paradigms and GPUs. One such library is the LIBSVM which is a sequential library that eases the use of SVMs for programmers [29]. Experiments performed on several known datasets showed up to 84x speed up on the Intel Xeon Phi compared to the sequential execution of LIBSVM. Their work target machine learning, our work target deep learning.

In a paper by Lei Jin et al. [30] training of sparse auto encoders and restricted Boltzmann machines were carried out on the Intel Xeon Phi. The authors managed to speed up the algorithm with a factor of 7 to 10 times compared to the Xeon CPU and more than 300 times compared to the un-optimized version executed on one thread on the coprocessor. The Xeon CPU was of type E5620 with a clock frequency of 2.4 GHz and 4 cores, and the Xeon Phi of type 5110p. The work carried out in their study target unsupervised deep learning of restricted Boltzmann machines and sparse auto encoders, our work target supervised deep learning of CNNs.

The performance gain on Intel Xeon Phi 7110p for a model called Brain-State-in-a-Box (BSB) used for text recognition is studied by Ahmed et al. in [31]. Focus lies in the recall function, i.e. the function that matches the input towards the trained patterns. According to the author, the speed up is about two-fold for the coprocessor compared to a CPU with 16 cores when parallelizing the algorithm. Their work target another architecture than ours, which target CNNs.

Studies have investigated the benefits and drawbacks in general for Intel Xeon Phi. Jianbin Fang et al. [32] investigated empirically the performance of the coprocessor. They concluded that it is possible to achieve the promised performance if the developer selects a proper parallelization strategy and puts in the hours to fully optimize the code. Existing applications will rarely utilize the coprocessor without this additional work.

In a technical report [33] carried out at Linnaeus University in the context of DNA sequencing, the authors managed to speed up their algorithm by 10x on a Intel Xeon Phi 7120p compared to a Xeon E5-2695v2 using the balanced affinity mode.

Work on the map-reduce framework for the Xeon Phi in [34] by Mian Lu et al. resulted in an optimized map-reduce framework which can be executed both for single and multiple Intel Xeon Phis. Map-reduce is a well known pattern used to distribute work over several nodes to speed up computations.

George Teodoro et al. [35] investigated the potential of large scale clusters of Intel Xeon Phis to facilitate the analysis of images retained from whole slide tissue specimens, allowing for the intense computations to be divided over several nodes. Intel Xeon Phi SE10P (5100 series) was used for the experiments.

A library to ease the burden for developer to create offloaded applications for the Xeon Phi is presented in [36] by Jiri Dokulil et al. In [37] the authors extend their work by implementing automatic tuning of algorithms to better make use of the resources of the coprocessor.

Kai-Cheung Leung et al. [38] investigated pattern matching of images on the Xeon Phi (of model 5110p) and achieved an up to 140x speed up compared to one thread on the coprocessor. The pattern matching uses k-nearest neighbours, a well-known classification method.

1.2.3 Related Work Targeting CNNs

Numerous previous work target deep learning in general and CNNs in specific. We will not attempt to cover work related to deep learning in general as this scope is too wide, instead this chapter will focus on CNNs for GPUs in the context of computer vision (image classification). Work related to MNIST [25] dataset are of most interest, also NORB [26] and CIFAR 10 [27] is considered. Additionally work done in speech recognition and document processing is briefly introduced. The purpose of this chapter is to introduce the state-of-the-art in the area of GPUs and CNNs.

Work presented in [23] by Dan Cireșan et al. target a CNN implementation raising the bars for the CIFAR10 (19.51% error rate), NORB (2.53% error rate) and MNIST (0.35% error rate) datasets. The training was performed on GPUs (Nvidia GTX 480 and GTX 580) where the authors managed to decrease the training time severely - up to 60 times compared to sequential execution on a CPU - and decrease the error rates to an, at the time, state-of-the-art accuracy level.

Jordan Vrtanoski et al. [39] showed a 25.8x speed up on an ATI 5870 GPU compared to a Xeon W3530 CPU when training the model on the MNIST dataset.

Work carried out by Dan Cireșan et al. [40] performed almost human-like on the MNIST dataset, achieving a best error rate of 0.23%. To be compared to the performance of humans, about 0.20%. The authors trained the network on a GPU.

The ImageNet challenge aims to evaluate algorithms for large-scale object detection and image classification based on the ImageNet dataset. Work by Alex Krizhevsky et al. [41] entered the 2012 challenge. The authors reduced the error rate of the test set to 15.3% from the second best 26.2% using a CNN with 5 convolutional layers. For

the experiments, two GPUs (Nvidia GTX 580) were used only communicating in certain layers. The training lasted for 5 to 6 days.

In a later challenge, ILSVRC 2014, a team from Google entered the competition with GoogleNet, a 22-layer deep CNN and won the classification challenge with a 6.67% error rate. The training was carried out on CPUs. The authors state that the network could be trained on GPUs within a week, illuminating the limited amount of memory to be one of the major concerns [3].

Work by Omry Yadan et al. [24] used multiple GPUs to train CNNs on the ImageNet dataset using both data- and model-parallelism, i.e. either the input space is divided into mini-batches where each GPU train its own batch (data parallelism) or the GPUs train one sample together (model parallelism). The work does not compare to CPUs, however, using 4 GPUs (Nvidia Titan) and model- and data-parallelism, the network was trained for 4.8 days.

Inchul Song et al. [42] constructed a CNN to recognize face expressions and developed a smart-phone app in which the user can capture a picture and send it to a server hosting the network. The network, predicts a face expression and sends the result back to the user. With the help of GPUs (Nvidia Titan), the network was trained in a couple of hours on the ImageNet dataset.

Experiments carried out on the NORB [26] dataset performed by Dominik Scherer et al. [43] showed up to 115x speed up trained on an Nvidia GTX 285 compared to a CPU implementation (Core i7 940). After training the network for 360 epochs, an error rate of 8.6% was achieved.

Dan Cireșan et al. [1] combined multiple CNNs to classify German traffic signs and achieved a 99.15% recognition rate (0.85 % error rate). The training was performed using an Intel Core i7 and 4 GPUs (2 x GTX 480 and 2 x GTX 580).

Researchers have also found CNNs successful for speech tasks. Large vocabulary continuous speech recognition deals with translation of continuous speech for languages with large vocabularies. Tara N. Sainath et al. [44] investigated the advantages of CNNs performing speech recognition tasks and compared the results with previous DNN approaches. Results indicated on a 12-14% relative improvement of word error rates compared to a DNN trained on GPUs.

Kumar Chellapilla et al. [45] investigated GPUs (Nvidia Geforce 7800 Ultra) for document processing on the MNIST [25] dataset and achieved a 4.11x speed up compared to the sequential version executed on a CPU (Intel Pentium 4, 2.5 GHz).

1.2.4 Example of Implementations and Libraries

This section enumerates some useful libraries and implementations found during our research study. For each library we present the name, platform, and programming language.

Name	Architecture Support	Programming Language
Cuda-Convnet2[1]	GPU	C++/CUDA
Tiny CNN[2]	CPU	C++
Eblearn[3]	CPU/GPU	C++
C++ training of CNN[4]	CPU	C++
NNforge[5]	CPU/GPU	C++
RaPyDLI[6]	GPU/MIC[11]	Python/C++/Java
Caffe[7]	CPU/GPU	C++
Torch[8]	CPU/GPU	C/CUDA [12]
Theano[9]	CPU/GPU	Python
Fnnlib[10]	CPU	C++

Table 1.2.1: Some useful implementations and libraries for deep learning and CNNs.

1.3 Outline

The thesis is organized as follows. In *chapter 2* the reader is introduced to the topics covered in the thesis. *Chapter 3* covers the software development and experiments followed by the results of the evaluation in *chapter 4*. *Chapter 5* performs a theoretical analysis of the algorithm. Based on the analysis, a performance model is designed and evaluated in the *section 5.4*. An analysis of both the theoretical- and experimental findings, and answer to the research question, is presented in *chapter 6*. The thesis is closed with reflections, conclusions and future work in *chapter 7*.

[1] https://code.google.com/p/cuda-convnet/source/
[2] https://github.com/nyanp/tiny-cnn
[3] http://eblearn.sourceforge.net/
[4] http://people.idsia.ch/ ciresan/
[5] http://milakov.github.io/nnForge/
[6] http://salsaproj.indiana.edu/RaPyDLI/
[7] http://caffe.berkeleyvision.org/
[8] http://torch.ch/
[9] https://github.com/Theano/Theano/
[10] http://sourceforge.net/projects/fnnlib/
[11] Many Integrated Core - an architecture from Intel which combines several cores on one chip. The Intel Xeon Phi is the brand name used for Intel's MIC architecture.
[12] CUDA is a parallel computing platform and API facilitating the use of GPUs for general purpose processing, also known as GPGPU.

Chapter 2

Background

The theory chapter introduces the core topics of the thesis including: High Performance Computing, Intel Xeon Phi, machine learning, deep learning, Deep Neural Networks (DNNs), and Convolutional Neural Networks (CNNs). Moreover, some optimization techniques for the coprocessor and parallelization schemes are discussed.

2.1 Intel Xeon Phi

High Performance Computing, devices pushing the computations boundaries beyond 10^{12} floating points operations per second (TFLOPS) are not only used by industry but are also made available to consumers as off-the-shelf solutions. As the increasing performance of single processing units have plateaued, more cores haven been added to the same chip to increase performance. Moreover, several nodes can be connected in a network to further increase the computational capabilities. This introduce new advantages for developers if they are able to fully make use of the hardware and deal with the demands high parallel computing entails [46].

The Intel Xeon Phi used in this study is of model 7120p, and facilitates 61 cores, each with a clock frequency of 1.2 GHz. The coprocessor can achieve up to *1.2 TFLOPS* performance. It facilitates 16 GB memory with a maximum theoretical bandwidth of 352 GB/s. Each core has its own private L1 (Level 1) and L2 (Level 2) cache. The L1 cache is 32KB wide for instructions and data respectively. The L2 cache combines the data and instructions into a 512KB wide space[47, 48]. *Figure 2.1.1* shows an overview of the coprocessor. More details can be found in *Appendix A*.

Intel emphasize the simplicity of porting applications to the coprocessor: "Moving a code to Intel Xeon Phi might involve sitting down and adding a couple lines of directives that takes a few minutes. Moving a code to a GPU is a project." [14]. Even if changes to the code is required to fully make use of the coprocessors' capabilities, the fundamental thinking's of the implementation does not have to change. If written in a language supported (such as FORTRAN or C(++)) the Intel compiler will take care of the compilation for the coprocessor automatically. Additionally, time spent optimizing for the Intel Xeon Phi will result in an optimized version for the Xeon processor as well [14].

The coprocessor runs its own Linux operating system, through *ssh* it is possible to communicate to it from the host over the network stack. Programs can either be executed in native mode, where they execute on the device, or in offload mode, executed on the host where work is offloaded to the coprocessor [48]. The coprocessor used in our study runs a μOS of version 2.6.38.8 and a software stack (MPSS) of 3.1.1.

Each core can switch between 4 hardware threads, each thread can only execute every

second cycle in a round-robin fashion, and hence at least two threads need to be running for full utilization. However, in most cases even more threads can be beneficial due to memory latencies - threads not executing can perform memory fetches while waiting. To minimize memory latencies, the coprocessor automatically perform pre-fetching of data, pre-fetch commands can also be issued programmatically by the developer [47, 49].

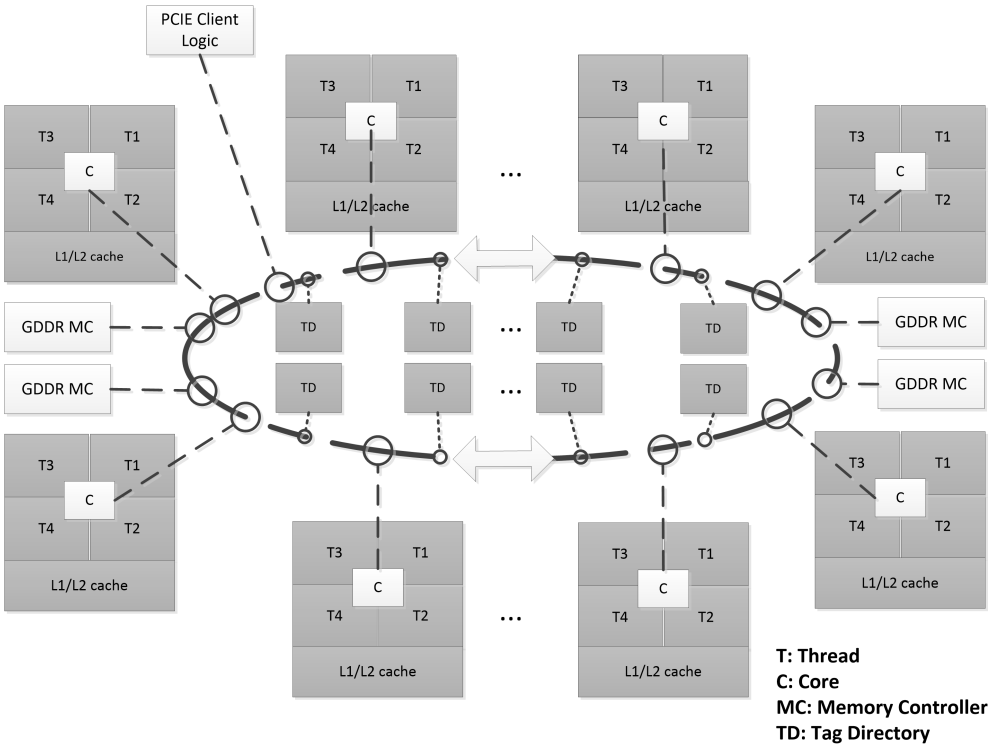

Figure 2.1.1: The architecture of Intel Xeon Phi.

The interconnect ring (in the center of *figure 2.1.1*) is bidirectional and consist of three sub-rings. The largest ring carries data, the next second largest ring carry instructions and the smallest carry data flow commands. When a memory access occur to the L1 cache and misses, the L2 cache will be queried. If the data is not contained in the core's L2 cache, a request will be made to the Tag Directory, TD. The TD contain the memory addresses of data in L2 cache for all the cores on the MIC, hence data can be transferred between cores, over the ring, and do not need to be fetched from the main memory. If data is found in the L2 cache about 250 clock cycles of waiting is omitted and if found in the L1 cache it 20 clock cycles of wait-time is omitted - therefore data locality is essential. The compiler issues hardware pre-fetches to mitigate memory wait times for future instructions which is essential to fully make use of the vector processing unit (VPU). The vector processing unit execute SIMD (Single Instruction Multiple Data) instructions which allows for multiple operations to be carried out simultaneously on different parts of the data [49, 47].

Each cache (both L1 and L2) at each core has a Translation Lookaside Buffer (TLB) which maps virtual and physical memory addresses. It also have its own vector processing unit (VPU) which can perform 8 (8x64) double precision or 16 (16x32) single precision operations in one cycle. The level of precision depends on the number of decimal points, i.e. the accuracy of the operations. Vectorization of code, i.e. code utilizing the VPU, can highly improve the application performance. Careful consideration of vectorization, data-locality and scalability is essential to fully utilize the Intel Xeon Phi [49, 48, 47].

In contrast, the Intel Xeon E5-2695v2 has a shared high level cache based on Intel Smart Cache technology, where all cores share 30 MB of memory. This allows the cores

to dynamically share the space as needed [50].

2.2 DNNs and CNNs

A short introduction to machine learning, and deep learning is carried out in this chapter. Deep neural networks (DNNs) and convolutional neural networks (CNNs) will be highlighted with the specifics required to fully understand the study - the section1 will make no attempt to cover the full aspect of deep learning.

Tom M. Mitchell provides a formal definition of machine learning algorithms in his book *Machine Learning* [51]: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [51, p. 2]

A deep neural networks is the underlying model used in deep learning. Many variations of deep neural networks exist, roughly divided into three categories: supervised, unsupervised and hybrid. The categorization is based on how the network learn. An unsupervised configuration learn patterns in data without any assistance, i.e. it has no prior knowledge of the labels of the data. On the contrary, supervised learning requires large datasets of labelled data. Hybrid networks is a combination of the both. Examples of unsupervised configurations are restricted Boltzmann machines and auto encoders. Recurrent Neural Networks and CNNs are both examples of supervised. The latter is the focus of this study [7].

Deep learning and DNNs are not new concepts, perhaps the first neural network that deserve the name deep was introduced by Fukushima in 1979, and this was also a CNN. However, due to larger datasets, better computational abilities and practical achievements, deep learning have gain an increase in popularity lately. In essence they are multi-layer models constructed to learn various levels of representations where higher level representations are described based on the lower level ones [9].

The mammal visual processing system is hierarchical (deep) in nature. Higher level features are abstractions of lower level ones. E.g. to understand speech, waveforms are translated through several layers until reaching a linguistic level. A similar analogy can be drawn for images, where edges, and corners are lower level abstractions translated into more spatial patterns on higher levels. Moreover, it is also known that the animal cortex consists of both simple and complex cells firing on certain visual inputs in their receptive fields. Moreover, simple cells detect edge-like patterns whereas complex cells are locally invariant, spanning larger receptive fields. These are the very fundamental properties of the animal brain inspiring DNNs and CNNs. *Figure 2.2.1* depicts a DNN consisting of one input layer, one output layer and two hidden layers, CNNs are covered in more detail later in this chapter [7, 8].

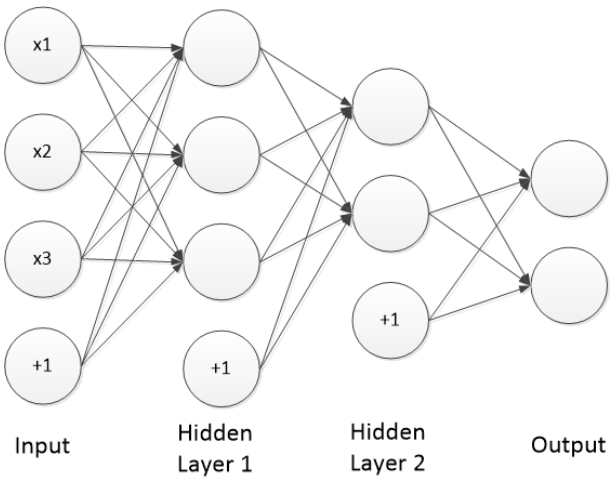

Figure 2.2.1: A deep neural network.

2.2.1 DNN Architecture

The architecture of a DNN consists of multiple layers of neurons. Neurons are connected to each other through edges (weights) (*figure 2.2.1*). The network can simply be thought of as a weighted graph; a directed acyclic graph represent a feed-forward network. The depth and breadth of the network differs as may the layer types. Independent of depth, a network have at least one input- and one output-layer. A neuron have a set of incoming weights attached which have corresponding outgoing edges attached to neurons in the previous layer. Also a bias term is used at each layer as an intercept term. The goal of the learning process is to adjust the network weights and find a global minimum by reducing the overall error, i.e. the deviation between the predicted and the desired outcome of all the samples. The resulting weight parameters can thereafter be used to make predictions of inputs not yet seen [52].

2.2.2 Forward Propagation

Forward propagation proceeds by performing calculations at each layer until reaching the output layer. At the output layer it is common to apply a soft max function (or similar) to squash the output vector and hence derive the prediction. The output on each layer is calculated as the weighted sum of the outputs of connected neurons at the previous layer sent through an activation function. Calculating the output given the input can be done using the equation $y_i^l = \sigma(x_i^l) + I_i^l$ where y_i^l is the output value of neuron i at layer l and x_i^l is the input value of the same neuron. The I is used for the input layer when the output y is non-existing, i.e. there is no previous layer. The input x_i^l can be calculated as $x_i^l = \sum_j (w_{ji}^l y_j^{l-1})$ where w_{ji}^l denotes the weight between neuron i in the current layer l, and j in the previous layer, and y_j^{l-1} the output of the jth neuron at the previous layer. To clarify: the input of a neuron i in the current layer is the weighted sum of the connected neurons j in the previous layer. In this example the *sigmoid* function is used as an activation function - *tanh* is another popular choice. The goal of activation functions are to return a normalized value (*sigmoid* return [0,1] and *tanh* [-1,1]) [53, 52].

2.2.3 Back-propagation

Back-propagation is the process of propagating errors, i.e. the loss calculated as the deviation between the predicted and the desired output, backward in the network, by adjusting

the weights at each layer. The stochastic gradient descent seeks to find the global minimum of the weight parameters by adjusting the weights in relation to the error (loss) of the prediction. After the forward propagation has been performed, the activations for the neurons at the output layer are known; in the concrete example of image classification, the output vector contains the prediction of the classification. Based on the predicted output, the loss as compared to the desired output can be calculated. Back-propagation begins by calculating the loss of the prediction using a loss function; examples of loss functions are *mean square error* and *cross entropy*. Partial derivatives of the output layer are computed and propagated backward in the network. Partial derivatives of a neuron are made relative to the error at the output layer, i.e. how much the neuron participated in the faulty prediction. The equation: $\frac{\delta E}{\delta y_i^l} = \sum w_{ij}^l \frac{\delta E}{\delta x_j^{l+1}}$ denotes that the partial derivative of neuron i at the current layer l is the sum of the derivatives of connected neurons at the next layer multiplied with the weights, assuming w^l denotes the weights between the maps. Consequently, weights are updated by subtracting a gradient from each weight parameter (more on this when considering CNNs below). Additionally, a decay is commonly used to control the impact of the updates, which is omitted in the above calculations. More concretely, the algorithm can be thought of as updating the weights on each layer based on "how much it was responsible for the errors in the output" [53, 11].

2.2.4 Overfitting, Dropout and Early Stopping

Overfitting a network leads to good performance on the training set however bad performance on the test set, i.e. the network rather memorizes than learn features. Overfitting occur when the model have too many parameters relative to the number of samples and hence many combinations of the parameters can model the same relationship between samples. Traditionally the problem was mitigated by training several networks, and then average them or otherwise validate them against the test set before continue training. However, this process is expensive, and hence a new method called dropout was introduced in 2012 by Hinton et al. who discovered the possibility to leave out every node with a probability of 0.5 at back-propagation - a cheap way of averaging a neural network. Results show that the need to stop training earlier due to bad convergence was reduced as well [54]

2.2.5 CNNs

This chapter will cover the basics of CNNs and their properties, extending the discussion in the previous section. CNNs achieve state-of-the-art performance in many applications including: speech recognition, image classification, natural language processing and machine translation [7]. The ImageNet competition push the boundaries of image recognition every year in the annual competition [20].

This section will introduce two new layer types: *convolutional layers* and *pooling layers*, and briefly discuss forward- and back-propagation.

The convolutional layer consists of several feature maps where neurons in each map connects to a grid of neurons in maps in the previous layer through overlapping kernels (*figure 2.2.2*). The kernels are tiled to cover the whole input space. The approach is inspired by the receptive fields of the mammal visual cortex. All neurons of a map extract the same features from a map in the previous layer as they share the same set of weights. Different skipping factors can be used to determine the total amount of neurons required to scan the entire image. The convolution also removes the high density of an otherwise

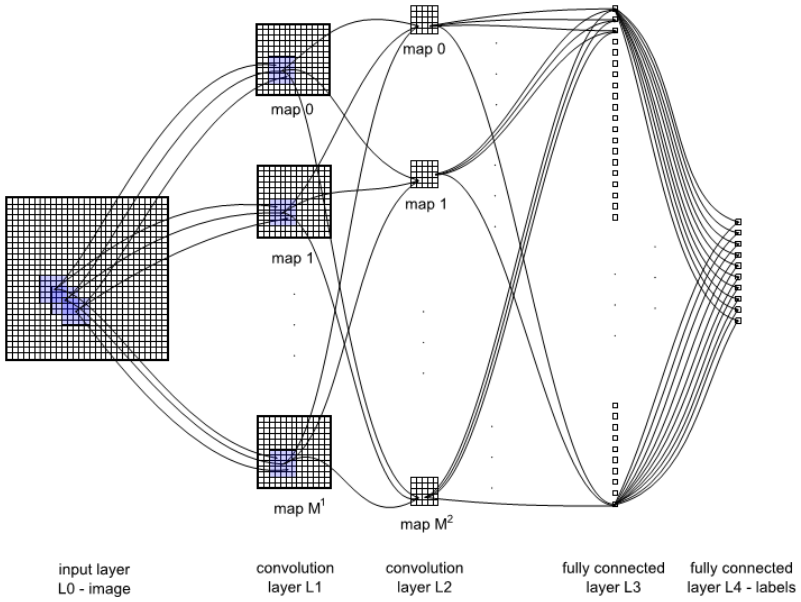

Figure 2.2.2: A CNN with 5 layers emphasizing the kernels. Courtesy to Dan Cireșan et al [1].

fully connected graph, which allows for faster training since the network contains less parameters. Feature maps introduce automatic feature extraction as each feature map learns to extract features from the layer below, e.g. edges or corners (in the case of an image). *Figure 2.2.2* shows an architecture with fully connected convolutional layers, emphasizing the kernels. Feature maps differ both in size and number of neurons (map size), commonly less and larger maps are used in the lower layers and more and smaller in the top layers, which is also the case for the traditional LeNet-5 (*figure 2.2.3*) [23].

Pooling layers intervene convolutional layers and have shown to lead to faster convergence. Each neuron in a pooling layer outputs the (maximum/average) value of a partition of neurons in the previous layer, and hence only activates if the underlying grid contains the sought feature. Besides from lowering the computational load, it also enables position invariance and down samples the input by a factor relative to the kernel size [2].

CNNs are commonly constructed similar to the LeNet-5 (*figure 2.2.3*) beginning with an input layer, followed by several convolutional/pooling combinations, ending with a fully connected layer and an output layer. Recent networks are much deeper and/or wider, the GoogleNet shown in *figure* 2.2.4 consists of *22* layers [2].

Forward propagation in the convolutional layer is similar to that of a fully connected layer. The input of a neuron is calculated as the weighted sum of all connected neurons over the kernel spanning the previous layer, as shown in *figure 2.2.2*. The output is the input sent through an activation function. The pooling layer is merely computing the output as the (max/average) value of a non-overlapping grid in the previous layer and map [23].

Back-propagation in convolutional layers can be divided into computing partial derivatives and computing weights. At a convolutional layer, calculations span over the set of connected maps in the previous layer. Fixating on one neuron, i, neurons in the previous layer over the kernel are updated with the derivative of i multiplied with the weight parameter connecting the two neurons. Each neuron j at the previous layer will therefore be updated several times, each time it is touched by a kernel. The resulting partial derivative

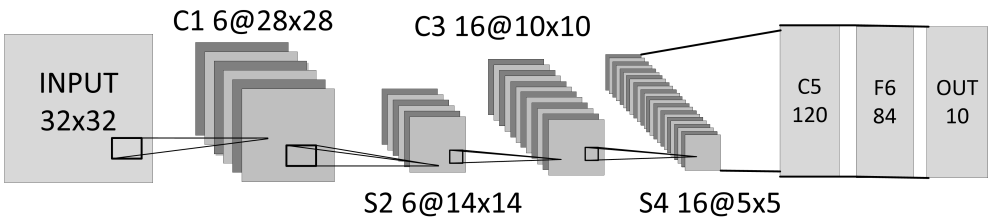

Figure 2.2.3: The traditional LeNet-5 architecture [2].

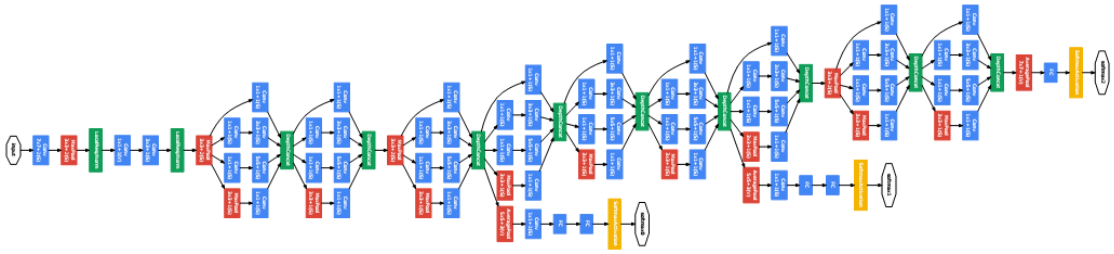

Figure 2.2.4: GoogleNet [3].

propagated to the neuron j at the previous layer can be thought of as $\frac{\delta E}{\delta y_j^{l-1}} = \sum(w_{ji}^l \frac{\delta E}{\delta y_i^l})$ where i and j are neurons connected over the kernel. Hence, the derivative of neuron j at the previous layer is the result of each neuron i propagating its delta values to neurons it touched when performing forward propagation [23, 55].

Weights of the convolutional layers are computed by subtracting a gradient from each weight parameter. Similar to propagating the partial derivatives, the gradient is calculated based on the derivative of neuron i at the current layer multiplied with the output of neuron j at the previous layer over the kernel. A weight is updated for every kernel in a map, since weights are shared between all kernels in the same map. The goal is to minimize the error produced by the weights in the forward propagation [23, 55].

When computing partial derivatives for the pooling layer, all neurons in the layer is iterated, and the partial derivative is pushed to the derivatives of the previous layer contributing to the (max/average) value in forward propagation: $y_{max(n)}^{l-1} = \frac{\delta E}{\delta y_n^l}$ where $max(n)$ is the neuron providing the maximum value in forward propagation. Pooling layers do not have any weights [23, 55].

2.3 Intel Xeon Phi Optimization Techniques

This chapter discusses some optimization techniques for the Intel Xeon Phi. It considers both algorithmic optimization and micro-architectural optimizations. The former targets the design of the algorithm and its data structures whereas the latter is tightly bound the underlying hardware.

2.3.1 Algorithmic Optimization Techniques

The major algorithmic considerations relates to parallelism (threading, vectorization), I/O, scalability and data structures.

To fully utilize the coprocessor one should make use of both thread- and SIMD-parallelism. Thread parallelism can be achieved by using, e.g. OpenMP [56], Cilk [57] or

Thread Building Blocks[58]. SIMD parallelism make use of the wide vector processing unit (VPU) of the coprocessor [59].

A short list of features facilitating parallelism:

- *OpenMP*: A set of compiler directives, environment variables and API functions to ease shared-memory parallelism [56].
- *Threading Build Blocks (TBB)*: Is a high performance parallel library from Intel, allowing for defining logical tasks to be performed in parallel rather than explicitly working with threads. Intel position it as a "template library for task parallelism" for C++ [58].
- *Cilk (+)*: Intel describe the library as an extension including reducers, array notation, SIMD instructions and keywords for spawning and syncing. It is available for C and C++ [57].
- *Compiler Directives*: The compiler can help improve the code through directives to increase the speed and make the memory access more efficient. The default optimization level is O2 which automatically add optimizations to the code at compile time, including vectorization. The O3 level use a more aggressive loop transformation compared to O2 which may not always be beneficial [60]
- *SIMD*: AVX (Advanced Vector Extensions) instructions helps optimize the execution through efficient vectorization. SIMD can be issued through Cilk, OpenMP or as native commands for the Intel compiler [57, 56].
- *Intel Math Kernel Library*: A highly optimized math computing library supporting BLAS and LAPACK routines in C (++), FORTRAN [61].
- *Intel Integrated Performance Primitives (IPP)*: A highly optimized library for multimedia, data-processing and communication applications. The built-in functions use Intel Streaming SIMD Extensions and Advanced Vector Extensions instruction sets [62].

The compiler performs automatic vectorization if possible - in many cases it is not and the developer needs to assist the compiler vectorize the code. Using an array notation, as defined in Cilk is one option. Another is to make use of the #pragma SIMD instruction in OpenMP. The compiler will try to unroll loops if possible, the developer can enforce the compiler if necessary [56, 59].

The scalability of the application is important in order to fully utilize the cores of the coprocessor. When spawning threads it is also necessary to be aware of the overhead that comes with joining the threads at the end of the work-sharing construct. If only a small subset of the threads exit later than the majority, the execution time (span) will increase since the total time is no better than that of the worst worker. Moreover, the time to spawn and synchronize threads will be abundant. Additionally, locks to shared data also have to be reconsidered when increasing the number of threads [59].

Other scalability issues include system calls and false-sharing; *malloc* and *gettimeofday* can be replaced by more efficient methods or omitted if possible. False-sharing occur when several threads work on data originating from the same cache line. The problem arises when threads work on data that resides nearby in memory since an update of one value forces other threads to re-fetch if working on data on the same cache lines [59].

One should try to reduce the number of memory accesses and work with data residing in the registers. For this, tiling or blocking can be used to work on cached data as far as possible [59].

Alignment of data should be done to 64 bytes in order to facilitate the memory access for the coprocessor. This can be achieved by allocating the memory with *_mm_malloc()*. Also, aligned memory in a structure of arrays (SoA) rather than an array of structures (AoS) is preferred as it results in a coalesced access pattern. It is also necessary to inform the compiler of the aligned access by instrumenting the code with aligned attributes. In case the data is not organized in a coalesced manner, the coprocessor can issue gather and scatter instructions which makes the loading and storing of data to and from memory more efficient. The goal is to work on local data as far as possible, and when memory access is required, optimize the access pattern by fetching larger chunks of data required by the operations. In some cases pre-fetching directives can help the compiler fetch data before-hand and alleviate the memory penalties [59].

2.3.2 Micro Architectural Optimization Techniques

This chapter will introduce some hardware-specific optimizations. A couple of metrics will be discussed in short to get an understanding of what to analyse and how to perform the fine-grained adaptations for the Intel Xeon Phi [49].

- *CPI*: To decrease the execution time, the Cycles Per Instruction (CPI) needs to be reduced, i.e. the number of cycles required to retire one instruction. It is beneficial to aim for a low CPI per core, as each hardware thread is only allowed to execute in a round-robin fashion. Moreover, this pattern hides some of the memory penalties, therefore even if the theoretical minimum number of CPIs per core is 0.5, by increasing the thread count penalties can be hidden. Also, SIMD instructions comprise several sub-instructions which infer a higher CPI. Finding hot spots with high CPI facilitates optimization of the code [49].

- *TLB Misses*: Both L1 and L2 for each core have a Translation Lookaside Buffer (TLB). Misses to the L1 TLB infer a penalty up to 25 cycles, and misses to the L2 TLB up to 100 cycles. Therefore it is important to minimize the misses. In cases were the L1 to L2 ratio is high, large pages can be used to reduce wait times. The goal should be to avoid spatiality and aggregate cache lookups as far as possible, which will also infer a better cache locality [49].

- *Vectorization Processing Unit (VPU) Usage*: Indicates on the magnitude of vectorization. The aim should be to utilize the VPU as far as possible [49].

- *Memory Bandwidth*: A low usage of bandwidth of data transfers between L2 caches and memory is essential to fully utilize the cores. Reducing the memory bandwidth used should be the goal of any application [49].

- *Cache Usage:* Efficient use of the caches are critical, a L2 cache miss infer a 250-cycle-penalty and a L1 cache miss 20-cycle penalty. Thus, data locality is important to fully utilize the cores. Software pre-fetching, blocking and alignment help mitigate these problems and should be utilized in all cases possible [49].

2.4 Parallelization Schemes for Stochastic Gradient Descent

On-line stochastic gradient descent have the advantage of instant updates of weights for each sample. However, the sequential nature of the algorithm yields impediments as the number of multi- and many core platforms are emerging. Simply training images in a sequential order will certainly not yield any parallelism and will not fully make use of the advantages modern architectures provide [63].

To derive the constructed parallelization schemes we searched the literature for existing successful parallelization schemes targeting on-line stochastic gradient descent. That is, approaches using supervised learning and the on-line stochastic gradient descent optimization function. This section is organized as follows. Firstly differences between model- and data-parallelism is highlighted. Thereafter strategies *A-D* discuss the approaches in more detail. Lastly, we explain model parallelism.

2.4.1 Model- and Data-Parallelism

The parallelism can be either divided data-wise, i.e. workers process several inputs concurrently, or model-wise, i.e. several workers share the computational burden of one input. The data parallelism divides the training set into on a batch of inputs, learning from several samples concurrently updating the weight vector with some delay. On the contrary, model parallelism aims to speed up the time for computations at each layer [64]. Whether one approach can be advantageous over the other mainly depends on the synchronization overhead of the weight vectors and how well it scales with the number of processing units.

2.4.2 Strategy A: Hybrid

Figure 2.4.1 visualizes data- and model parallelism, data parallelism is applied in convolutional layers, and model parallelism in fully connected layers. In [4] Krizhevsky argues that both data- and model-parallelism have their advantages, however in different contexts. He concludes that because of layer characteristics one should consider using data parallelism for the convolutional layers and model parallelism for the fully connected layers. The argument is based on computation- and representation size. Convolutional layers are most computational costly and require about 95% of the computations whereas the fully connected layers have smaller representations and are less computational costly. Krizhevsky proposes three variations of the algorithm, however in general the process is as follows: each worker is assigned a set of images and performs calculations on the convolutional layers. However, at the fully connected layers the algorithm switch to model parallelism and the workers help each other carry out the computations [4].

2.4.3 Strategy B: Averaged Stochastic Gradient

Dividing the input into batches and feeding each batch to a node, was presented to work well for the MNIST dataset when training restricted Boltzmann machines on 40 nodes in [64]. The algorithm proceeds as follows:

1. Initialize the weights of the learner by randomization.
2. Split the training data into n equal non-overlapping chunks and send them to the learners together with the weight parameters.

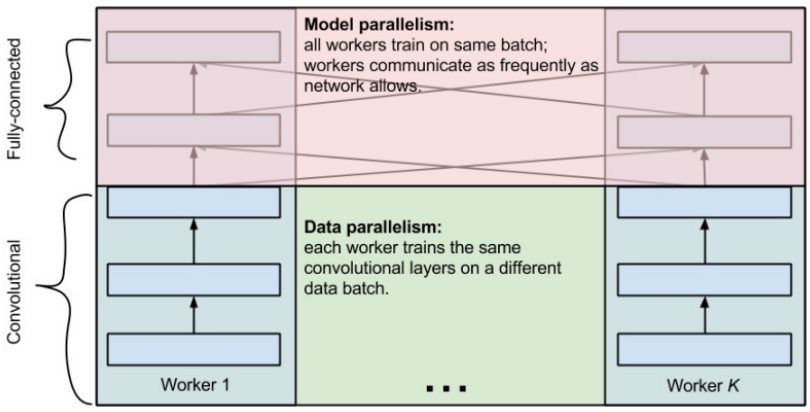

Figure 2.4.1: The hybrid approach of data- and model-parallelism. Courtesy to Alex Krizhevsky [4].

3. Each learner process the data and calculates the weight gradients for its subset of inputs.
4. Calculated gradients are sent back to the master.
5. The master computes the new weights as the mean of the proposed updates and updates the weights.
6. The master sends the new weights to the nodes and a new epoch/iteration begins.

Results show that the convergence speed is slightly worse than for the sequential approach, however the training time is heavily reduced. The authors acknowledge that larger mini-batches sent to the learners reduce the convergence and at the same time speeds up the computations [64]. A similar study by Zhao You et al. in [65] also uses average stochastic gradient descent, showing that larger mini-batches leads to slower convergence.

2.4.4 Strategy C: Delayed Stochastic Gradient

In [66] John Langford et al. recognize that the update of weight parameters can be done in a round-robin fashion by the workers. One proposed solution is to divide samples into n chunks (where n may be the total number of threads used) and let each thread work on its own distinct chunk of samples, only sharing a common weight vector. However threads are only allowed to update the weight vector in a round-robin fashion, and hence each update will be delayed [66].

2.4.5 Strategy D: HogWild!

Strategy D is based on the HogWild! algorithm proposed in 2011 by Feng Niu et.al. HogWild! is stochastic gradient descent without locks. The approach is applicable for sparse optimization problems (threads/core updates do not conflict to a great extent). The authors theoretically and experimentally show that the algorithm outperforms other stochastic gradient descent parallelization schemes, and mathematically prove the approach to be valid [63].

2.4.6 Model Parallelism

By assigning several threads to work on the same layer it is possible to speed up the calculations of each layer and hence speed up the total training time. For forward propagation the work is divided over a set of workers, each working on a set of neurons. The back-propagation work either by pulling or pushing deltas. Pulling deltas facilitates parallelism as the delta of each neuron can be calculated in atomicity. Pushing deltas are easier to implement, however it leads to concurrent updates by workers. Similar, the weight updates are divided over the set of kernels shared between maps in consecutive layers [23].

Chapter 3

The Approach

This study empirically evaluates the performance of supervised deep learning using CNNs on the Intel Xeon Phi. First, an implementation was selected and modified into a sequential non-optimized version. Thereafter, we designed *CHAOS* (explained in section *3.2*) and implemented it on the sequential implementation, constructing a parallel, optimized version. The parallel implementation was evaluated experimentally and data was collected for different architectures and threads. Results were processed and presented in *chapter 4*, and further analysed and depicted in *chapter 6*. Lastly, a theoretical study was conducted and presented in *chapter 5* and combined with the experimental study to conclude the answer to our research question *RQ1* in *section 6.3*.

3.1 Selection of Implementation

Firstly a selection procedure was carried out searching a feasible implementation to be used in the study. During the literature study several implementations were found, c.f. section *1.2.4*. Primarily the inclusion criteria comprised six items qualifying the implementation: good reputation of author, simplicity, no (or few) dependencies, dynamic - supporting various architectures, C++ programming language, supports training of CNN. Other attributes not required however requested: academic relation, validity (correct implementation), example datasets provided in the package.

CNNs were not the only architecture considered, other supervised learning models were included in the search as well, including recurrent neural networks. However, at some point in the process it became clear that CNNs was of most interest, and hence became a requirement in the selection procedure. No previous research have been found for CNNs during our literature study and Intel Xeon Phi (no supervised learning for that matter). Moreover CNNs are applied to state-of-the-art applications, including computer vision and speech recognition [7]. Moreover, choosing an existing implementation enables us to reach the goals of the study, spending time on evaluation rather than implementation; numerous implementations already exists, this would not bring anything new to the research community.

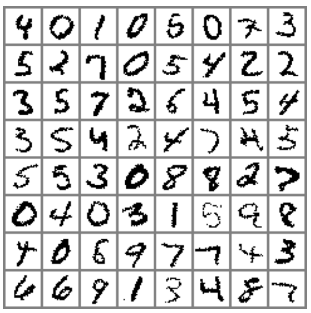

Figure 3.1.1: Extract from MNIST dataset. Courtesy to Y. Tang et al. [5].

We found a project written by Dan Cireșan to meet all the requirements. The paper [23] cover a similar, although slightly different, implementation. The project implements a trainer for CNNs targeting the MNIST [25] dataset of handwritten digits, packaged with the project. An extract of MNIST is shown in *figure 3.1.1*. MNIST consists of 60,000 training/validation images and 10,000 testing images. The network architecture can be defined in a text file dynamically, however, two predefined networks are already packaged with the project. The project is implemented in C++ and developed in Visual Studio. There are no dependencies other than *Boost*.

Other implementations, e.g. *Eblearn*, *Torch*, and *Caffe* had several dependencies, and/or are reasonably large. Others are highly integrated with GPUs, e.g. *Cuda-Convnet2*. *Tiny CNN* provide a library for CNNs with example of MNIST however we could not find any academic relation, otherwise this is a promising library.

First a development environment was prepared to favor code completion and code locality on the host. The project was created on the host to which the coprocessor is connected. The communication with the host system was done through *ssh*, both to the host and to the coprocessor. Compilation was performed using the Intel c compiler, *icc* 15.0.0, and various parameters, shown in *Appendix D*.

For the experiments, two CPUs and the Intel Xeon Phi were used. The Linux host comprises an Intel Xeon CPU E5-2695v2 with a clock frequency of 2.40 GHz with 12 physical cores and 2 threads per core using hyper-threading, resulting in 48 logical cores. The desktop computer comprises an Intel Core i5 661 with a clock frequency of 3.33 GHz and 4 logical cores. The coprocessor is of model 7120p comprising 61 cores and 4 threads per core with a clock frequency of 1.2 GHz. Detailed information of the platforms can be found in *Appendix A*.

3.2 CHAOS: Controlled HogWild with Arbitrary Order of Synchronization

We introduce *CHAOS*, **C**ontrolled **H**ogWild with **A**rbitrary **O**rder of **S**ynchronization, a parallelization scheme constructed and used in this study. By combining parts of strategies A to D, defined in *section 2.4*, we came up with a data parallel, controlled version of HogWild!, with delayed updates. The key aspects of *CHAOS* are:

- Data parallelism - Multiple identical network instances (workers) are created, sharing weight parameters; other variables are thread private allowing for several images to be processed concurrently. *Figure 3.2.1* shows an overview of the scheme, ignoring superfluous details. After creating network instances, and preparing images, the training starts. Each epoch carries out three major steps. The first step, *Training*, proceeds with each worker picking an image, forward propagates it through

the network, calculates the error, and back-propagates the partial derivatives, adjusting the weight parameters. Since each worker picks a new image from the set as long as more images are available, other workers does not have to wait for significantly slow workers. After *Training*, each worker participates in *Validation* and *Testing* evaluating the accuracy of the network by predicting images in the validation and test set accordingly. Adoption of data parallelism was inspired by Alex Krizhevsky in [4], promoting data parallelism for convolutional layers as they are computational intense.

- **C**ontrolled **H**ogwild - Updates of weight parameters during back-propagation are not instant nor significantly delayed. To avoid unnecessary invalidation of cache lines and align memory writes, updates of shared weights are delayed to the end of each layer's computations. Intermediate updates are applied to local weight parameters, thus calculating the gradients before sharing them with other workers. The approach was inspired by *HogWild!* [63] proposing instant updates of weights, and delayed updates as proposed by John Langford et al. in [66]. We use neither and both, the gradients are calculated and saved locally first, however, workers can update the global set of gradients at any time - they do not have to wait for other workers to finish update before sharing their contributions.

- **A**rbitrary **O**rder of **S**ynchronization - Because all workers share weight parameters, there is no need for explicit synchronization, however, an implicit synchronization is performed in an arbitrary order, since writes are controlled by a fist-comes-first schedule and reads are performed on demand.

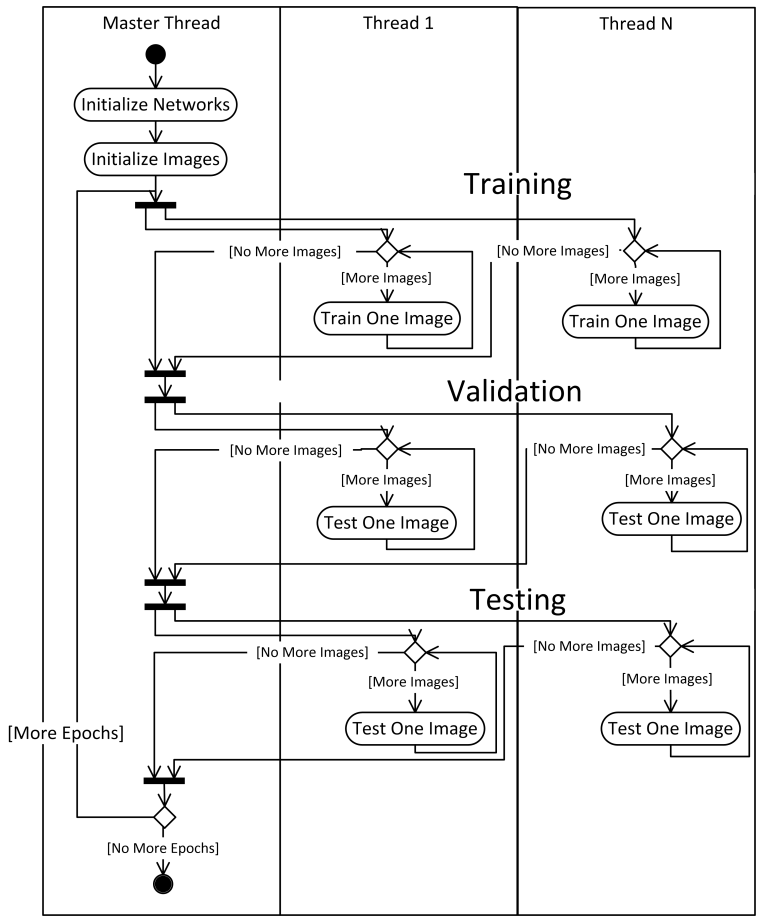

Figure 3.2.1: Activity diagram of *CHAOS*.

The arbitrary order of updates entails non-deterministic results, however, theory and practice show this deviation to be negligible. To detain theoretical ground in our work, we choose to combine the strategies that suits well with our current implementation. The main goal is to minimize the time spent in the convolutional layers which can be done through data parallelism, adapting the knowledge presented in *strategy A* (defined in *section 2.4.2*).

In *strategy B* (*section 2.4.3*), the synchronization is performed as a result of averaging workers gradient calculations. Since work is distributed, computations are performed on stale parameters. The strategy can be applied in distributed and non-distributed settings. The division of work over several distributed workers was adapted in *CHAOS*.

In *strategy C* (*section 2.4.4*), the updates are postponed using a round-robin-fashion where each thread gets to update when it is its turn. The difference compared to *strategy B* is that instances train on the same set of weights and no averaging is performed. The advantage is that all instances train on the same weights. The disadvantage of this approach is the delayed updates of the weight parameters as they are performed on stale data. Training on shared weights and delaying the updates are adopted in *CHAOS*.

Strategy D (*section 2.4.5*) present a *lock-free approach* of updating the weight parameters, updates are performed instantly without any locks. Our updates are not instant, however, after computing the gradients there is nothing prohibiting a worker contributing to the shared weights, the notion of instant inspired *CHAOS*.

Some of the above strategies introduce mini-batch training, i.e. dividing the inputs to a batch size > 1. This has shown to lead to slower convergence as the batch sizes increase [64]. Therefore we neglect this approach in our consideration.

In *HogWild!* weight updates are instant, using no delays. However, some limitations of the coprocessor rules out this approach - invalidation of cache lines infer large memory penalties. Hence update of gradients are delayed until after the weight computations of the entire layer. Also *#atomic* pragmas are used to ensure data races are avoided, initial tests show that without it, some penalties were introduced. At the end of each back-propagation there is no synchronization required since all weights of all layers are shared among all network instances. The major benefit is the reduced delay of updates (they are almost instant); parameter updates are delayed only with a small fraction. We recognize that weights may change rapidly (even during calculations at one layer) and impact the training for other workers. Although this seems to be a theoretical problem, it does not seem to be a problem in practice.

3.3 Implementation

We performed a series of modifications to the implementation in order to run our experiments, resulting in two versions, which we refer to as sequential and parallel. The sequential version is the original implementation supplemented with a *Reporter* and *Helper* class to facilitate results collection and training. The parallel version is, as the name suggests, parallelized, and optimized using several optimization techniques. This chapter explains the development of the application used in the experiments. The main requirement is to facilitate the many cores of the coprocessor efficiently to lower the execution time, at the same time maintaining low deviation in error rates, especially on the test set. Moreover, the quality of the implementation is verified using errors and error rates on the validation and test set.

3.3.1 Sequential Implementation

The sequential version is a minor justification of the original version. To ease profiling, a *Reporter* class was added to serialize execution results. The instrumentation should not add any time penalties in practice. However, if these penalties occur in the sequential version they are likely to imply corresponding penalties in the parallel version, therefore it should not impact the results. *Figure 3.3.1* shows a class diagram for the sequential version. Parameters and variables are omitted to avoid superfluous information, otherwise almost all methods are included (with some exceptions of *Idx*) as well as their visibility. The *NeuralNetwork* class provides the core functionality including forward- and back-propagation. The *Main* class interacts with the *NeuralNetwork* to initiate the training process. The *Idx* class manages the images and labels and the *Helper* class manages common tasks such as generating date-time strings for the file names.

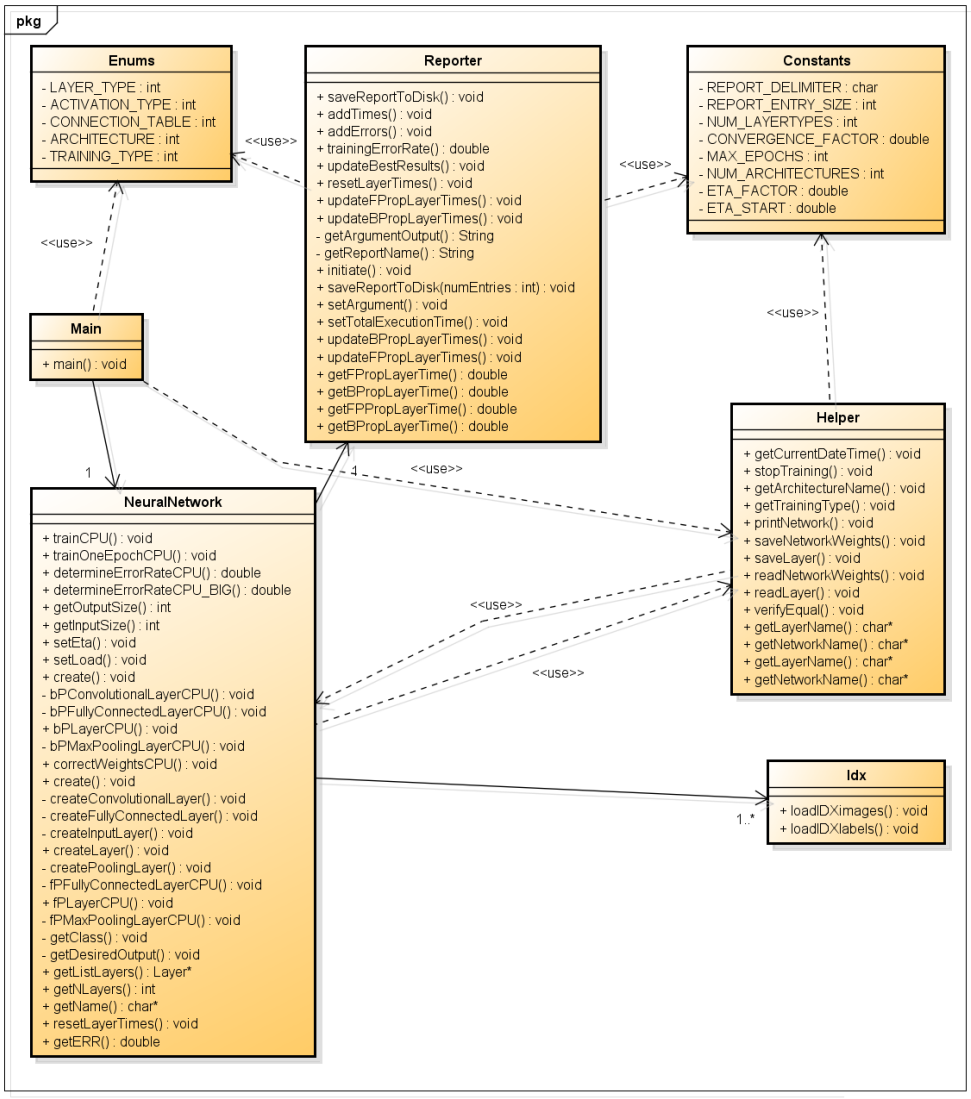

Figure 3.3.1: Class diagram for the sequential version.

3.3.2 Parallel Implementation

The main goal of the parallel version is to lower the execution time of the sequential implementation and to scale well with the number of processing units on the coprocessor. To facilitate this, it is essential to fully consider the characteristics of the underlying hardware. From results derived in the sequential execution we found the hotspots of the application to be predominantly the convolutional layers. The time spent in both forward- and back-propagation occupies about 94% of the total time of all layers (up to 99% for the larger network), which is depicted in the *table 3.3.1*. Interestingly, similar measurements are discussed by Alex Krizhevsky in [4]. Moreover, the on-line stochastic gradient descent is sequential in its nature, making it hard to parallelize the algorithm. To speed up the execution, we adopted the parallelization scheme designed in this study, *CHAOS*.

Our proposed strategy, *CHAOS*, presented earlier in this chapter, only infer smaller modifications to the implementation. A set of N network instances are created and assigned to T threads. We assume $T == N$, i.e. one thread per network instance. T threads are spawned, each responsible for its own instance.

The overview of the algorithm was shown previously in *figure 3.2.1*. In *figure 3.3.2*

Layer type	Forward propagation	Back-propagation	% of total
Fully connected	40.9 s	30.9 s	1.4 %
Convolutional	3,241 s	1,438 s	93.7 %
Max pooling	188.3 s	8.2 s	3.9 %

Table 3.3.1: Execution times at each layer for the sequential version on the Xeon E5 using the small CNN architecture.

the training, testing and back-propagation details are shown. Training (left) picks an image, forward propagates it, determines the loss and back-propagates the partial derivatives (deltas) in the network - this process is done simultaneously by all workers, each worker processing one image. Each worker participating in testing (center), picks an image, forward propagates it and then collects errors and error rates. The results are cumulated for all threads. Perhaps the most interesting part is the back-propagation (right). The shared weights are used when propagating the deltas, however, before updating the weight gradients, the pointers are set to the local weights. Thereafter the algorithm proceeds by updating the local weights first. When a worker have contributions to the global weights it is allowed to update in a controlled manner, avoiding data races. Updates immediately affects other workers in their training process. Hence the update is delayed slightly, to decrease the invalidation of cache lines, yet almost instant and workers do not have to wait for a longer period of time before contributing with their knowledge.

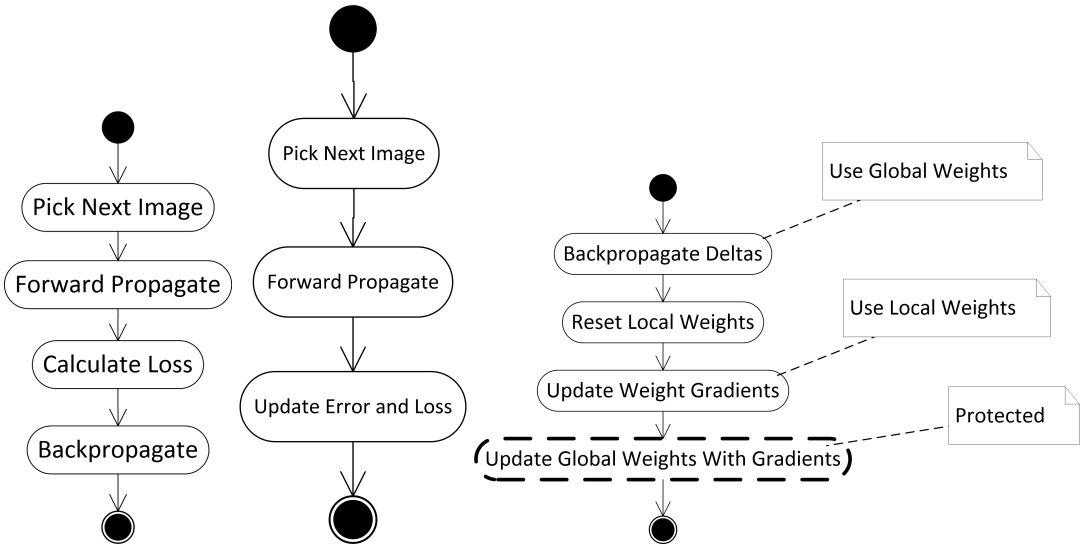

Figure 3.3.2: The training (left), testing (center), and back-propagation (right) of one image.

Too see why delays are important, consider the following scenario: If training several network instances concurrently, they share the same weight vectors, other variables are thread private. The major consideration lies in the weight updates. Let W_j^l be the *j*the weight on the *l*th layer. In accordance with the current implementation, a weight is updated several times since neurons in a map (on the same layer) share the same weights, and the kernel is shifted over the neurons. Further assume that several threads work on the same weight W_j^l at some point in time. Even if other threads only read the weights, their local data, as saved in the Level 2 cache, will be invalidated and a re-fetch is required to assert their integrity. This happens due to cache lines are shared between cores (as

explained in *chapter 2*). The approach of slightly delaying the updates and forcing one thread to update in atomicity leads to fewer invalidations. Still a major disadvantages is that the shared weights does not infer any data locality (data cannot retain completely in Level 2 cache for a longer period of time).

In the study, OpenMP was selected to facilitate thread- and SIMD parallelism utilizing the AVX (Advanced Vector Extension) features. The rationale of choosing OpenMP are two-fold: firstly, OpenMP is owned and developed by a non-profit organization, supported by many large vendors, which is the opposite of other techniques, e.g. Intel TBB and Intel Cilk. Secondly, we have previous positive experience of OpenMP.

A fork of the sequential implementation created the base for the parallel. For version management we also created a new *Git* project, and a new project on the host machine.

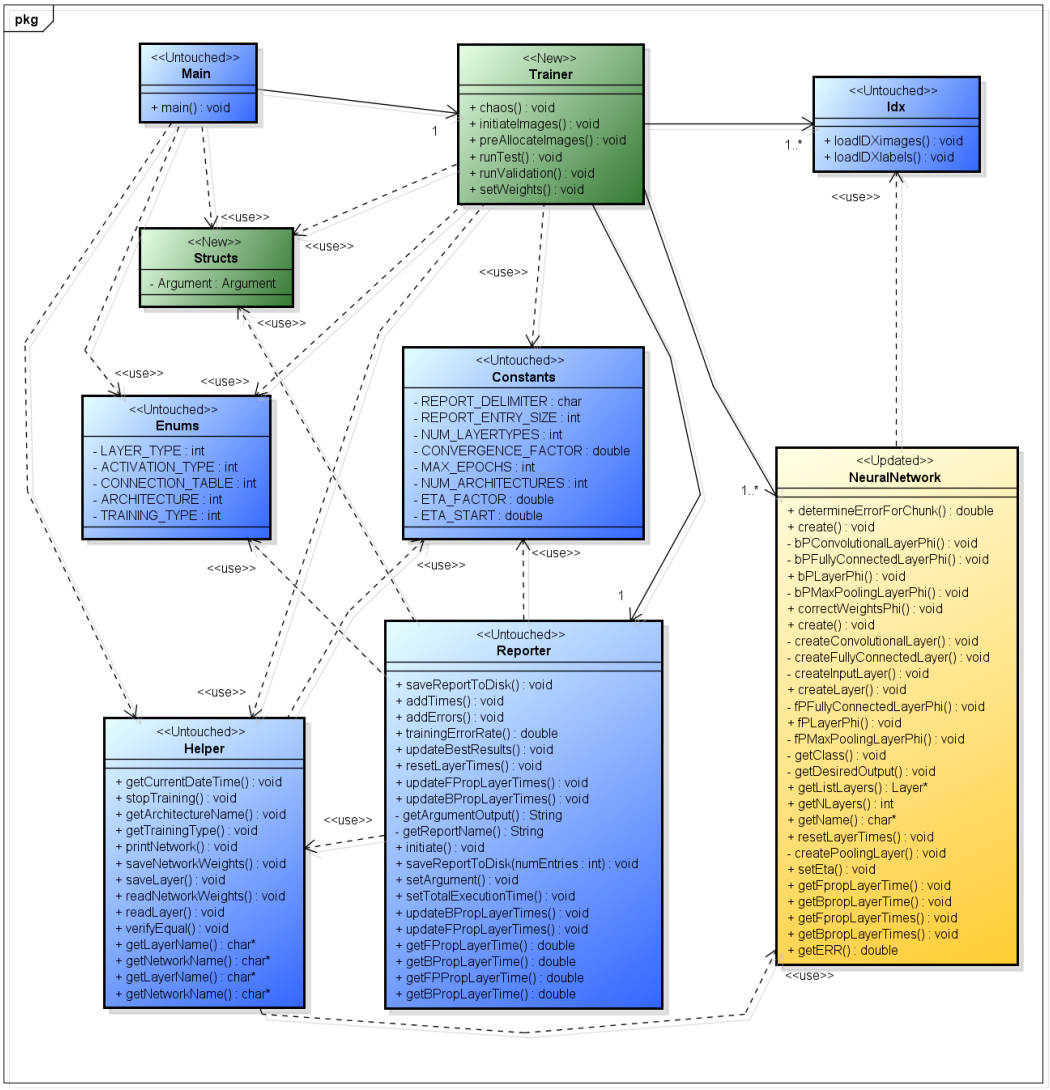

Figure 3.3.3: Class diagram for the parallel implementation.

The class diagram for the *parallel* version is depicted in *figure 3.3.3*. Classes unchanged are marked blue. Green are classes that have been added, and yellow classes have been changed. Additionally, stereotypes annotates the state for colorless print outs. The *Trainer* class was introduced including the *chaos* method, the starting point of the training. It also includes functions to execute the tests and validation in parallel: *runTest* and *runValidation*. The *setWeights* method sets the pointers of all network instances to

the weights of the default instance. The *initiateImages* and *preAllocateImages* methods prepares the images in memory for easier access. *Structs* contains the *Argument* struct encapsulating the input parameters.

The *NeuralNetwork* class have minor changes. Methods to retrieve the layer times are added and the *determineErrorRate** methods are substituted with *determineErrorForChunk*.

The main method is parameterized with the training type, CNN architecture size, network size and thread count. Also compiler directives can be used to inform the compiler to use *LOCALWEIGHTS*.

A code snippet of the local weight updates is shown in *listings 3.1*. First, the weights pointer is set to work on the local weights, and reset their values. After the update (from line 10) the local gradients contribute to the global weight parameters.

```
1  // Point to local weights and reset them
2  #ifdef LOCALWEIGHTS
3      Wmap = LocalW;
4      memset(LocalW, 0, nWeights * sizeof (double));
5  #else
6      Wmap = W;
7  #endif
8
9  // Update the shared weights from local weights
10 #ifdef LOCALWEIGHTS
11     for (int w = 0; w < nWeights; w++) {
12     #pragma omp atomic
13         W[w] += LocalW[w];
14     }
15 #endif
```

Listing 3.1: Code snippet for the update of weight parameters.

To further decrease the time spent in convolutional layers, loops were vectorized to facilitate the vector processing unit of the coprocessor. Data was allocated using *_mm_malloc()* with 64 byte alignment increasing the accuracy of memory requests. The vectorization was achieved by adding *#pragma omp simd* instructions and explicitly informing the compiler of the memory alignment using *__assume_aligned()*. Some unnecessary overhead is added through the lack of data alignment of the deltas and weights. A vectorization report for the back-propagation of convolutional layers can be found in *listings 3.2*, showing that the estimated potential speed up is 2.250 when facilitating the vector processing unit, and that the vectorization suffer from unaligned stores. More reports for the forward- and back-propagation functions can be found in *Appendix C*.

```
1  LOOP BEGIN at neuralnetwork.cpp(504,21)
2          remark #15399: vectorization support: unroll factor set to 2
3          remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
4          remark #15451: unmasked unaligned unit stride stores: 1
5          remark #15458: masked indexed (or gather) loads: 1
6          remark #15475: --- begin vector loop cost summary ---
7          remark #15476: scalar loop cost: 160
8          remark #15477: vector loop cost: 70.750
9          remark #15478: estimated potential speedup: 2.250
10         remark #15479: lightweight vector operations: 47
11         remark #15480: medium-overhead vector operations: 2
12         remark #15482: vectorized math library calls: 1
13         remark #15488: --- end vector loop cost summary ---
14 LOOP END
```

Listing 3.2: Vectorization report for forward propagation in the convolutional layer.

Many of the system calls made per thread basis were removed, e.g. the images are loaded into a pre-allocated memory location instead of allocating new memory when requesting an image. Likewise for the desired output. This removes the wait times otherwise required by system calls. Compared to loops, experiments show that the overhead

was negligible and that using *memset* to reset the weights did not infer a noticeable time penalty.

Hardware pre-fetching was discussed in *chapter 2*, a technique applied by the coprocessor to mitigate the shortcomings of the in-order-execution scheme. Pre-fetching loads data to the L2 cache in order to make it available for future computations. In addition to the hardware pre-fetcher, a software pre-fetcher can be used with the *#pragma pre-fetch* command. Pre-fetching did however in our case not infer a noticeable speed up and was therefore not included in the final version.

We compile the code with optimization level *03* enabling automatic optimizations of the compiler, including *Block-Unroll-And-Jam*. However, identical results was derived using the *O2* option.

The default thread affinity for OpenMP were used as no noticeable speed up were encountered for other schemes. For the Xeon Phi, the default setting is *scatter*. We also run the experiments with *balanced* and *compact*, *compact* showed worse performance and *balanced* equal to the ones derived for *scatter*. Reduction is used for summarizing the errors in the *RunTest*, and *RunValidation* function.

Reducing the spawning and synchronization of threads increase the scalability of the algorithm. Moreover, letting workers pick images instead of assigning images to workers, allow for a smaller overhead at the end of a work-sharing-construct. The number of locks are minimized as far as possible. Additionally, the *CHAOS* parallelization scheme mitigates the problem of false-sharing.

The locality of data was achieved by making most of the variables thread private. Although the weight parameters are not thread private, they can be shared between cores via the interconnect ring.

To facilitate scalability, minimize memory contention, and synchronization overhead, *CHAOS* was adopted to the implementation. Through thread parallelism, dividing the input space over available workers, training was distributed. *CHAOS* uses the vector processing units to improve performance and tries to retain local variables in local cache as far as possible. The delayed updates decrease the invalidation of cache lines. Since weight parameters are shared among threads, there is a possibility that data can be fetched from another core's cache instead of main memory, reducing the wait times. Also memory was aligned to 64 bytes and unnecessary system calls was removed from the parallel work.

3.4 Evaluation Approach

The evaluation of the implementation was performed on three different platforms (Core i5, Xeon E5, and Xeon Phi 7120p) described earlier in this chapter. Also a more detailed explanation can be found in *Appendix A*. The evaluation used the MNIST [25] dataset of handwritten digits, comprising 60,000 training/validation images and 10,000 test images using varying thread counts, and CNN architectures. This section aims to describe the details of the evaluation process.

Data collection was done through observations using bat and bash scripts for Windows (desktop) and Linux (host and coprocessor) respectively. An execution scheme was constructed as an overview of the intended configurations to execute. *Table 3.4.1* shows an overview of the scheme, execution reports can be found in *Appendix D*. All configurations were run three times with some exceptions (refer to *Appendix D* for more information), and the results were averaged - executions lasted for week(s) hence more executions were not possible to perform. We selected the thread counts based on previous experience.

Platform	Version	#Threads
Host	Seq.	1
Host	Par.	1
Host	Par.	12
Host	Par.	24
Host	Par.	48
Phi	Par.	1
Phi	Par.	15
Phi	Par.	30
Phi	Par.	60
Phi	Par.	120
Phi	Par.	180
Phi	Par.	240
Phi	Par.	244
Desktop	Seq.	1

Table 3.4.1: Planned execution scheme.

Results from each execution were copied by the scripts into a corresponding result directory and later imported to Excel, averaged and processed for further calculations. To create the resulting artefacts, data from each sheet was aggregated into a common results document. The same process was performed repeatedly for each CNN architecture considered.

Architectures are presented in *table 3.4.2*. As can be seen, at each step, the size and hence the number of computations increase. The small and medium architectures were packaged with the implementation, however, the large architecture was inspired by an architecture used by Cireșan et al. in [23].

Each execution used a starting *eta* (decay) of *0.001* and a factor *0.9*. For more accurate comparison between executions the scrambling of images at each epoch was ignored, and the weights were calculated once and then read into the proceeding executions. The small and medium architectures were trained for 70 epochs and the large for 15.

The data collected, serialized to disk included:

- Training time per epoch - The total time for training, i.e. forwarding and back-propagating the images through the network.
- Validation time per epoch - The total time for validation, i.e. forwarding the validation images through the network and calculate the loss and error rate.
- Testing time per epoch - The total time for testing, i.e. forwarding the testing images through the network and calculate the loss and error rate.
- Validation error per epoch - The total loss of images during validation in one epoch.
- Testing error per epoch - The total loss of images during testing in one epoch.
- Validation error rate per epoch - The error rate at validation, i.e. the number of images recognized incorrectly/total number of images per epoch.
- Testing error rate per epoch - The error rate at testing, i.e. the number of images recognized incorrectly/total number of images per epoch.

- Epoch time - The execution time for the whole epoch, including training, validation and testing.
- Layer times per network instance - The total time spent per layer averaged over the number of network instances.
- Total execution time - The total time for the whole training excluding reading images to memory and creating instances.
- Total training time - The total time for training.
- Total validation time - The total time for validation.
- Total testing time - The total time for testing.
- Total epoch time - Total time spent in epochs, i.e. the total execution time minus the sequential work, similar to the total execution time excluding some minor initialization work.
- Arguments used for execution - CNN Architecture, number of threads, number of network instances, strategy, name of file, and compiler directives (LOCAL-WEIGHTS).

	Type	Maps	Map Size	Neurons	Kernel Size	Weights
Small	Input	-	29x29	841	-	-
	Conv	5	26x26	3,380	4x4	85
	Max	5	13x13	845	2x2	-
	Conv	10	9x9	810	5x5	1,260
	Max	10	3x3	90	3x3	-
	Full	-	50	50	-	4,550
	Output	-	10	10	-	510
Medium	Input	-	29x29	841	-	-
	Conv	20	26x26	13,520	4x4	340
	Max	20	13x13	3,380	2x2	-
	Conv	40	9x9	3,240	5x5	20,040
	Max	40	3x3	360	3x3	-
	Full	-	150	150	-	54,150
	Output	-	10	10	-	1,510
Large	Input	-	29x29	841	-	-
	Conv	20	26x26	13,520	4x4	340
	Max	20	26x26	13,520	1x1	-
	Conv	60	22x22	29,040	5x5	30,060
	Max	60	11x11	7,260	2x2	-
	Conv	100	6x6	3,600	6x6	216,100
	Max	100	2x2	900	3x3	-
	Full	-	150	150	-	135,150
	Output	-	10	10	-	1,510

Table 3.4.2: CNN architectures used in evaluation.

The parallel version is compared to the sequential by the error and error rates collected during evaluation. This lets us deduce how much accuracy is lost (or gained) compared to the sequential (base line) version.

3.5 Analysis Approach

Results collected is analysed in *chapter 6*, in which the execution time and speed up for Xeon Phi is discussed for varying number of threads and CNN architectures. Also, errors, error rates (incorrect predictions), and time spent in different layer types are explained in more detail. The analysis will also derive conclusions related to the theoretical analysis (*chapter* 5) and performance model (*section 5.4*) to conclude the research question *RQ1*.

The errors and error rates are used to validate our implementation, comparing the errors of the parallel version with that of the sequential and original version. Errors and error rates for the validation and test set were collected for each epoch, and are depicted in graphs for each CNN architecture in *chapter 4*. We discuss the deviation in number of incorrectly predicted images and also a ratio of incorrectly predicted images and deviation in units for the error. This is further related to the state-of-the-art performance on MNIST.

Chapter 4

Results

In this chapter results collected in the empirical study are presented. The training was performed for 70 epochs for the small- and medium-architecture and 15 epochs for the large, using a starting decay (eta) of *0.001* and factor of *0.9*. An epoch is an iteration in the training process. The evaluation was performed on three different platforms: Intel Xeon E5-2695v2 with a clock frequency of 2.4 GHz, 12 physical- and 48 logical cores, Intel Core i5 661 with a clock frequency of 3.33 GHz, 4 logical (2 physical) cores, and the Intel Xeon Phi 7120p comprising 61 cores with a clock frequency of 1.2 GHz, and a total of 244 hardware threads. The evaluation was performed using varying thread counts on Xeon E5 and Intel Xeon Phi. The MNIST dataset of handwritten digits was used as input.

To ease comparison, network instances were initialized with the same weight parameters and scrambling of images was disabled. This results in worse error rates, nevertheless evaluation is the main goal of this study, error rates secondary. Each execution was performed three times (with some exception) and then averaged to form the results presented in this chapter.

4.1 Execution Time, Speed Up and Prediction Accuracy

The execution time is the total time the algorithm executes, excluding initializing the network instances and images, this is true for both the sequential and parallel version. The stop criteria used is max epochs, i.e. the network trains until reaching the maximum number of epochs defined. Other techniques have been investigated, however not used to collect the results, however, in the results analysis we provide an overview if stopping at a common error rate. The speed up is measured as the relativeness between two execution times, with the sequential execution times of E5 and Xeon Phi as the base. The error rate is the fraction of images the network was unable to predict and the error the cumulated loss from the loss function.

In the figures and tables we use some abbreviations due to limited space. *Par* is the parallel version, *Seq* the sequential. Also *T* denote threads. E.g. *Phi Par. 1 T* is the parallel version and one thread on the Xeon Phi.

4.1.1 Results on the Small CNN Architecture

The small network consists of 7 layers: 1 input (29x29 neurons), 2 convolutional (5,10 feature maps, 26x26,9x9 map size), 2 max pooling (5,10 feature maps, 13x13,3x3 map size), 1 fully connected (4,550 neurons) and 1 output (510 neurons). The following section shows the results of the small CNN architecture.

In *figure 4.1.1* the total execution times for the different configurations are depicted. The sequential version is less efficient than the parallel version. The *Core i5 Seq* is about 5 times worse than *Xeon E5 Seq*. For the *Xeon Phi Par*, more threads imply lower execution time in any case. The time is halved between 15, 30 and 60 threads which is not the case between 60, 120, 180, 240 and 244 threads where the performance only is a fraction better.

The sequential version executed on the Xeon E5 spends about *83 minutes* training the network, and *Phi Par. 244 T* about *6 minutes*. Additionally, the Core i5 require about *408 minutes, 7 hours* to complete the training.

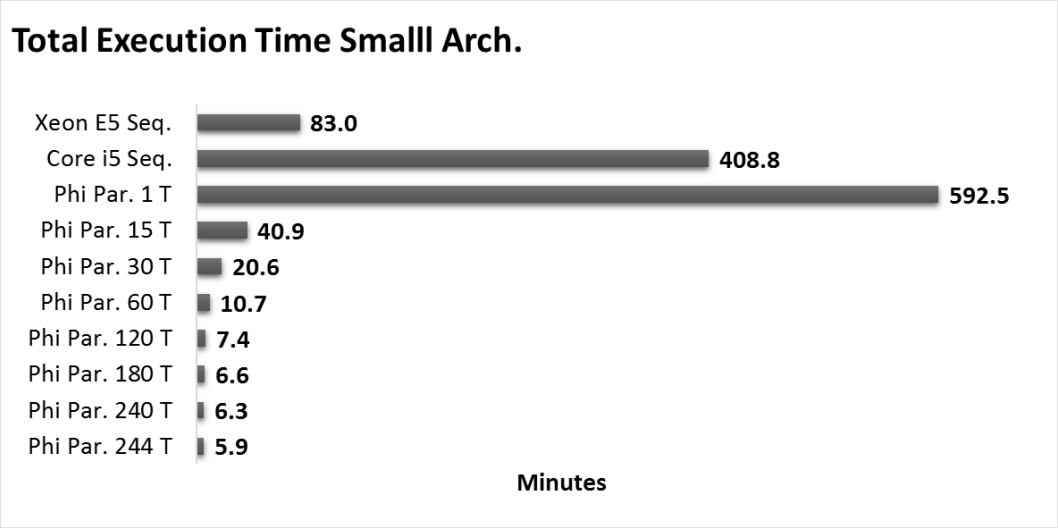

Figure 4.1.1: Total execution time for the small CNN architecture.

Figure 4.1.2 emphasize the facts shown in *figure 4.1.1* in terms of speed up, compared to *Xeon E5 Seq*; values in parentheses are relative measurements compared to *Xeon E5 Par. 1 T*. As can be seen, "throwing more cores at the problem" results in increased speed up in all cases. 240 threads on the *Xeon Phi Par.* infer a *94.6x* speed up compared to *Phi Par. 1 T* and *13.26x* compared to *Xeon E5 Seq*. 244 threads on the coprocessor yield even higher speed up, *100.4x* compared to *Phi Par. 1 T* and *14.07x* compared to *Xeon E5 Seq*.

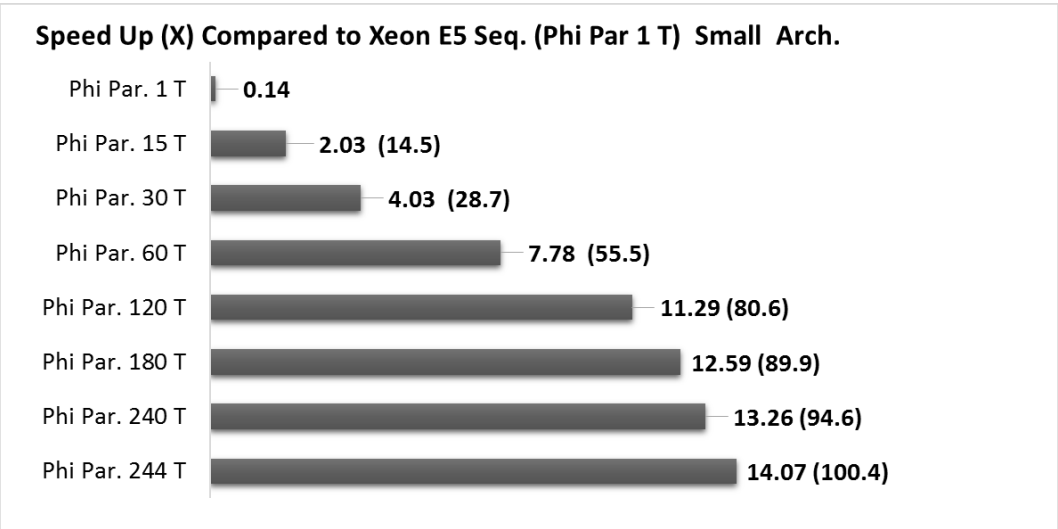

Figure 4.1.2: Speed up for the small CNN architecture compared to *Xeon E5 Seq* (*Phi Par. 1 T*).

Error and error rates are recorded for each epoch and set of images (training/validation and testing). The errors are cumulated results of the loss function, and hence describe the magnitude of false predictions of all images. The error rates are the fractions of incorrectly predicted images. Therefore the error rates on the validation set is deemed to be lower than for the test set since this is the set of images the network adjusts to.

The error rate on the validation set can be seen in *figure 4.1.3*. *Phi Par. 180 T* has the lowest ending error rate and *Phi Par. 15 T* the highest. The optimal as derived from *Phi Par. 1 T* should be used as a base line. *Phi Par. 180 T* differ by 0.03% (about 17 images less) and *Phi Par. 15 T* with 0.007% (about 4 images).

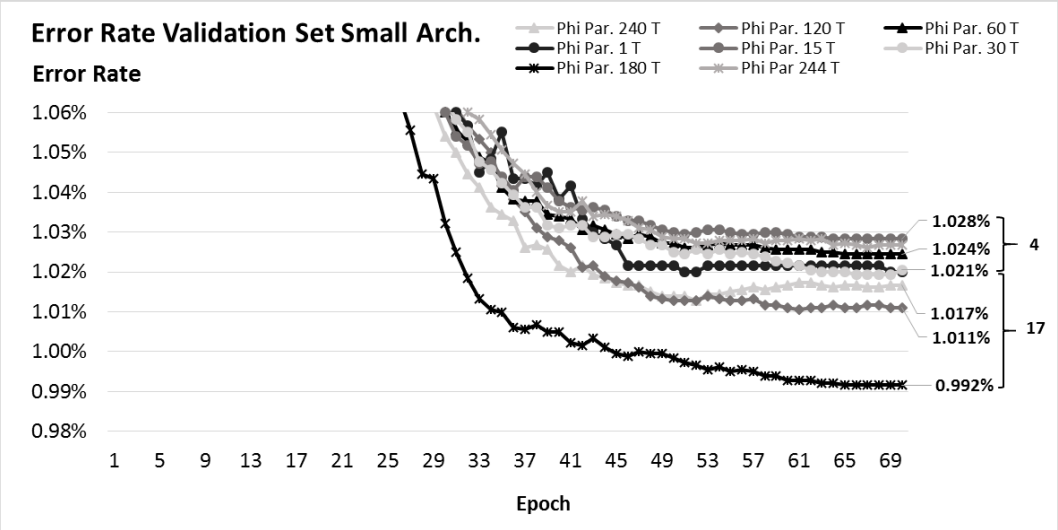

Figure 4.1.3: Error rate of the validation set for the small CNN architecture.

The error rate for the test set can be seen in *figure 4.1.4*. In general the curves fluctuates heavily in the beginning and stabilizes at the end. *Phi Par. 1 T* has the lowest ending error rate closely followed by *Phi Par. 15 T*. Worse is the *Phi Par. 120 T*, the difference is 0.063% (about 6 images).

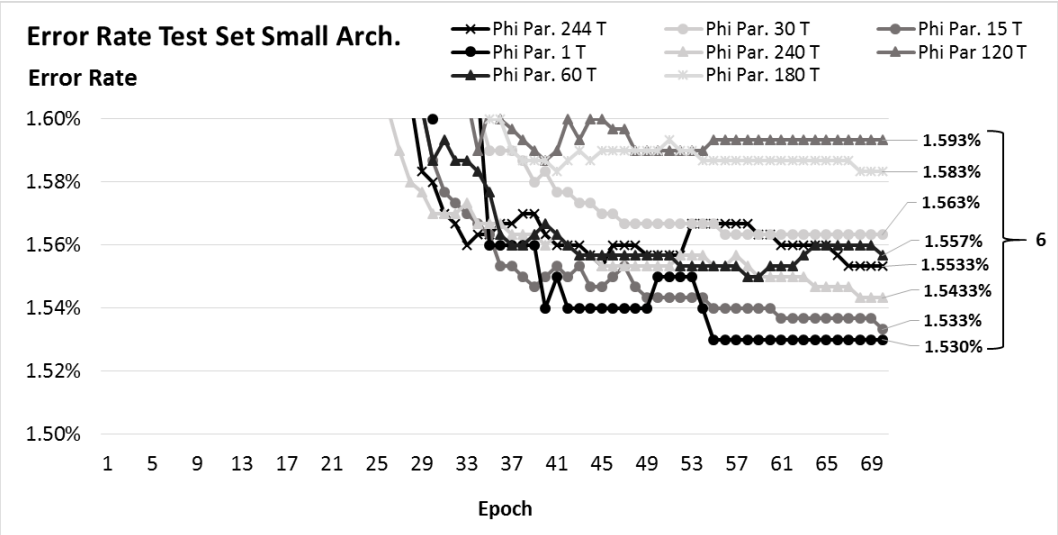

Figure 4.1.4: Error rate of the test set for the small CNN architecture.

The error of the validation set (figure *4.1.5*) decrease as the training proceeds even if the curve diminish at the end. The optimal error from *Phi Par. 1 T* is 2,092 which can

be compared to 2,105 for the *Phi Par. 240 T* (13 units) and 2,077 for *Phi Par. 120 T* (15 units).

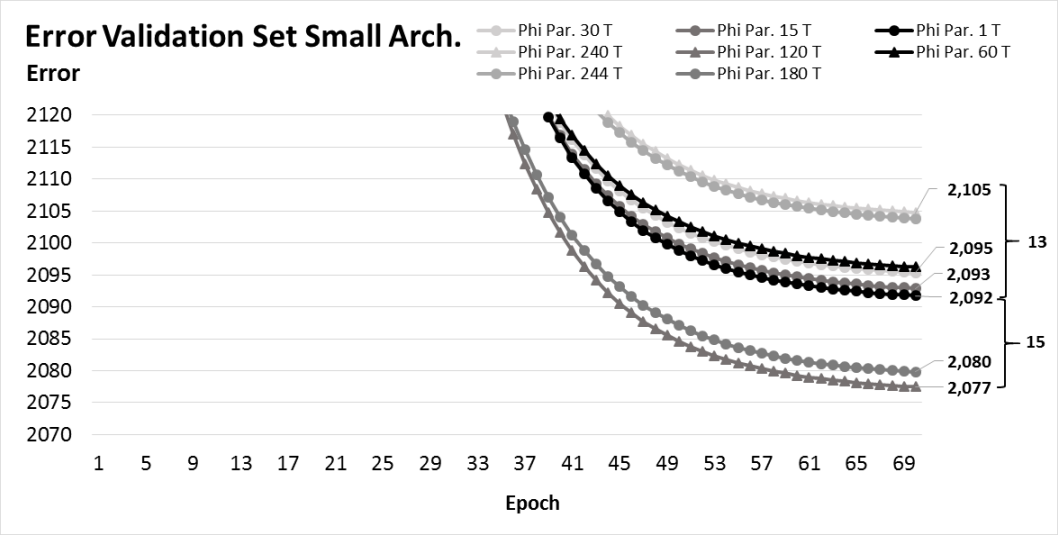

Figure 4.1.5: Error of the validation set for the small CNN architecture.

The error on the test set can be seen in *figure 4.1.6*. The lowest ending error has *Phi Par. 1 T*, at each step, the error gets worse, and hence *Phi Par. 244 T* ends with the highest error (about 22 units worse than optimal).

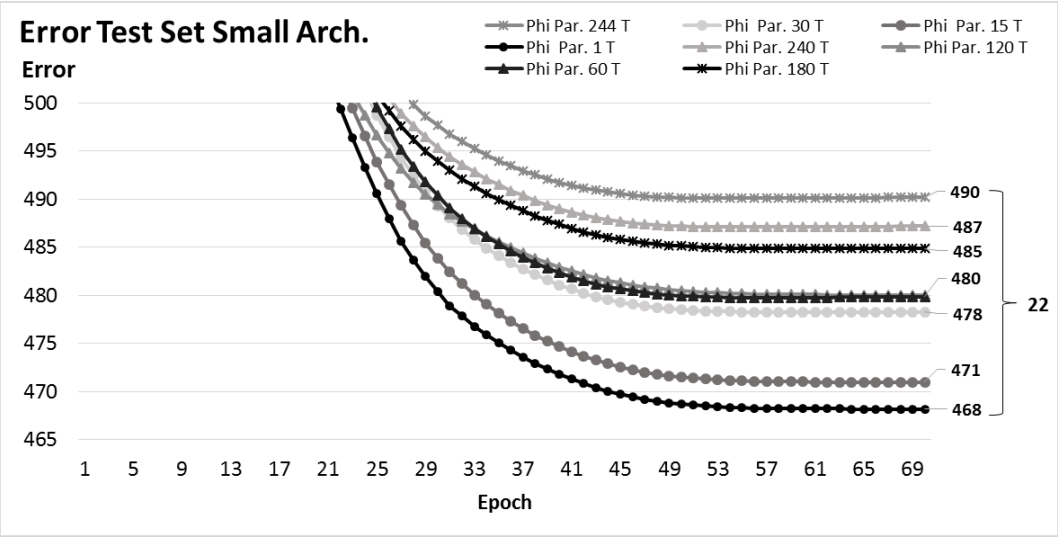

Figure 4.1.6: Error of the test set for the small CNN architecture.

The reader should be aware of the collection strategy; results collected in executions were averaged, which could infer some implicit fluctuations, we provide the standard deviation in *Appendix D*.

4.1.2 Results on the Medium CNN Architecture

The medium CNN architecture consists of equal amount of layers as the small. However, the number of maps for the convolutional (and max-pooling) layers are increased - from 10 to 20 and 20 to 40 - which also increase the number of neurons and weights. More details of the CNN architectures can be found in *Appendix B*.

The total execution time continues to decrease with the number of threads (*figure 4.1.7*), although it increases almost linearly with the number of threads for up to 60 threads for the *Phi Par.* it is only lowered by a fraction for 60, 120, 180, 240, and 244 threads. *Core i5 Seq.* spends about *72 hours* for the whole training compared to *Xeon E5 Seq.* about *13 hours*, and *Phi Par. 244 T* about *1 hour*. Additionally, *Phi Par. 1 T* spend about *114 hours*.

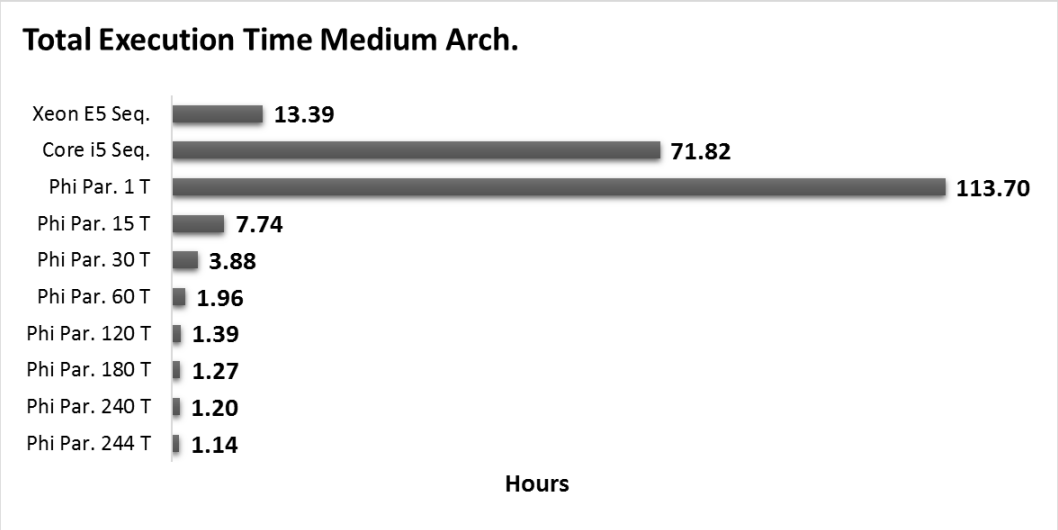

Figure 4.1.7: The total execution time for the medium CNN architecture.

In figure *4.1.8* it can be seen that there is almost a linear speed up for *Phi Par. 15, 30 and 60 T* when compared to *Phi Par. 1 T*. Additionally, *Phi Par. 244 T* has a speed up of *99.9x*, compared to *Phi Par. 1 T*. When increasing the number of threads, the speed up increase as well. *Phi Par 1 T.* is about 8 times slower than the sequential version on Xeon E5.

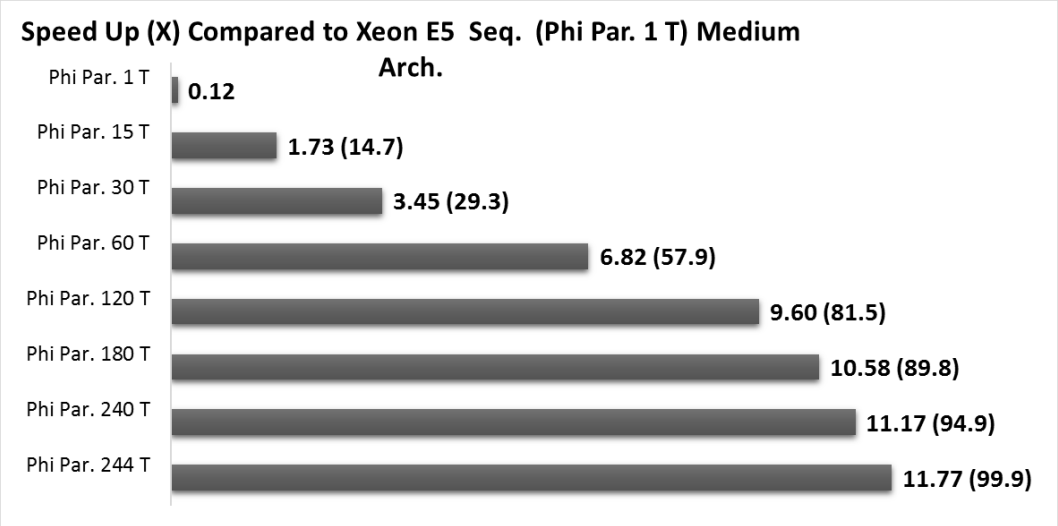

Figure 4.1.8: The speed up for the medium CNN architecture compared to *Xeon E5 Seq* (*Phi Par. 1 T*).

The error rates of the validation set is shown in *figure 4.1.9*. The lowest error rate has *Phi Par. 60 T* (about 3 images from optimal) and worse *Phi Par. 180 T* (about 3 images from optimal).

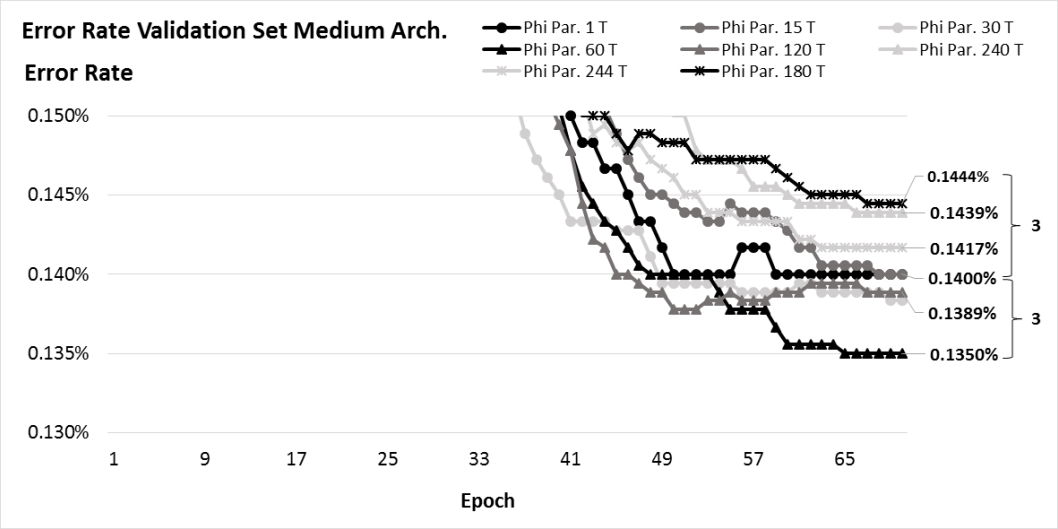

Figure 4.1.9: Error rate of the validation set for the medium CNN architecture.

Figure 4.1.10 contains the error rates for the test set. All lines fluctuates a bit before paving out in the end. It can be seen that the resulting error rates for the *Phi Par.* differs slightly from the optimal values - *Phi Par. 240 T* about 1 image and *Phi Par. 15 T* about 5 images.

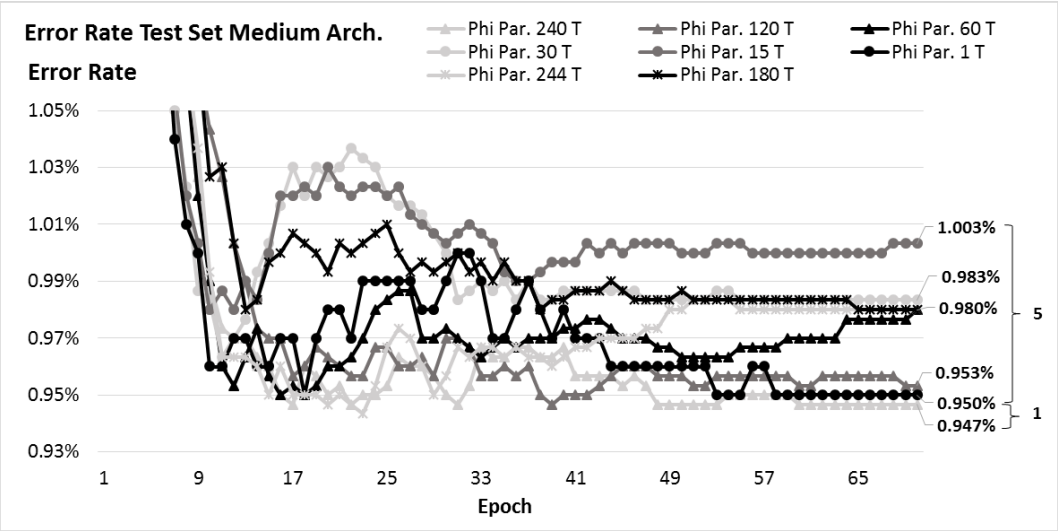

Figure 4.1.10: Error rate of the test set for the medium CNN architecture.

The validation errors are shown in *figure 4.1.11*. The lowest ending error is that of *Phi Par. 30 T* and the highest is *Phi Par. 240 T*. The optimal error (*Phi Par. 1 T*) is 508.55, and hence the worst differs with about one unit compared to the base line and the best two units.

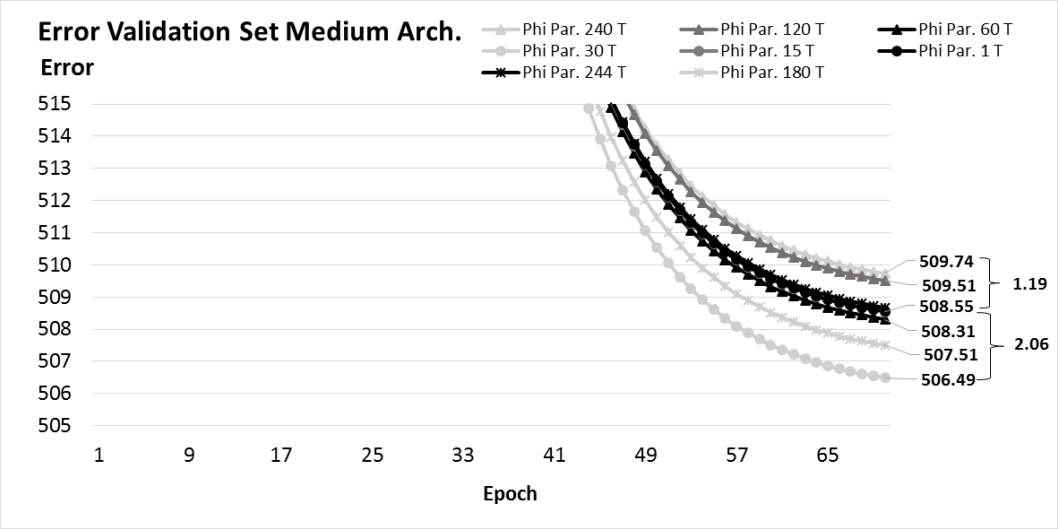

Figure 4.1.11: Error of the validation set for the medium CNN architecture.

For the test set (*figure 4.1.12*) the values are decreasing rapidly until reaching about 20 epochs where the values stabilizes and slightly increase. This is true for all configurations, even for the *Phi Par. 1 T*. Although the resulting error is higher than for 30 epochs, this is only by a fraction. The ending errors only differs by a couple of units, *Phi Par. 180 T* being the lowest, and *Phi Par. 1 T* the highest.

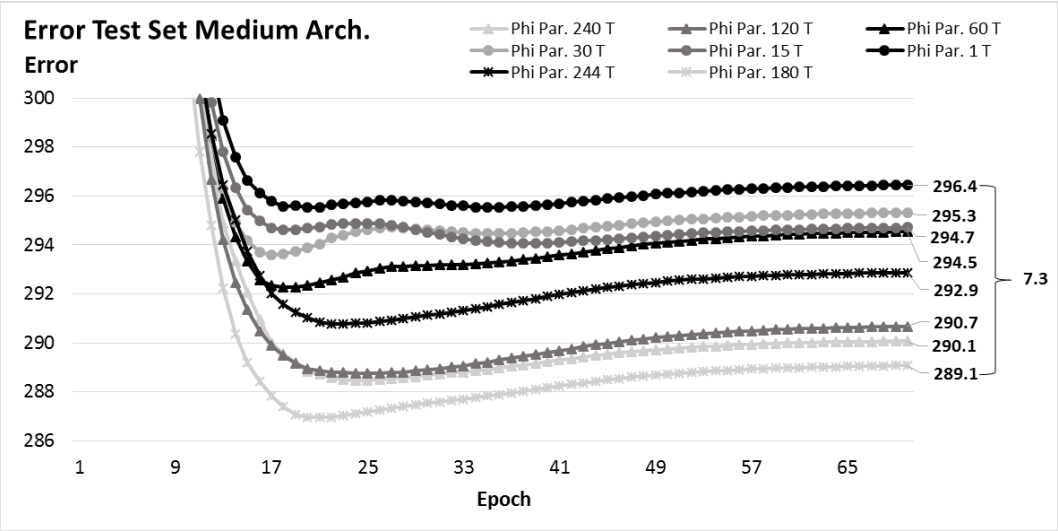

Figure 4.1.12: Error of the test set for the medium CNN architecture.

4.1.3 Results on the Large CNN Architecture

The following section present the results for the large CNN architecture including the speed up, errors and total execution time.

As can be seen in *figure 4.1.13* more threads decrease the running time for *Phi Par*. The execution time for *Phi Par. 1 T* is far more than for *Core i5 Seq*, however, already after *Phi Par. 15 T* the execution time is less. It takes *167.5 hours* to train the large network on Core i5 661, *31.1 hours* on Xeon E5-2695v2 and *3 hours* on the Xeon Phi when training for 15 epochs.

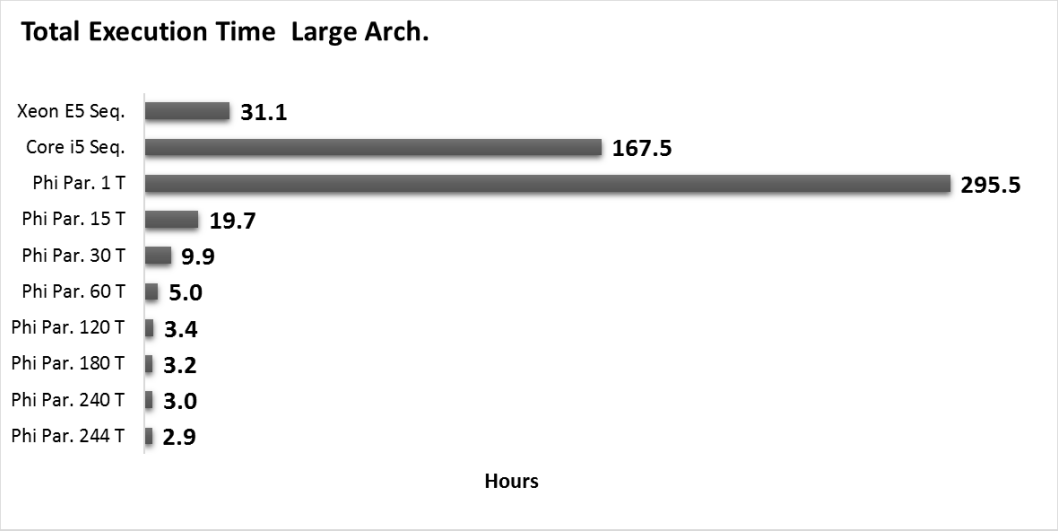

Figure 4.1.13: Total execution time for the large CNN architecture.

The speed up scales well when adding more threads (*figure 4.1.14*) and is doubled as the threads are doubled up until 60 threads. The best speed up is encountered by *Phi Par. 244 T*, *103.5x* compared to *Phi Par. 1 T* and *10.91x* compared to the *Xeon E5 Seq*. *Phi Par 1 T* is about 9 times worse than the sequential version on Xeon E5.

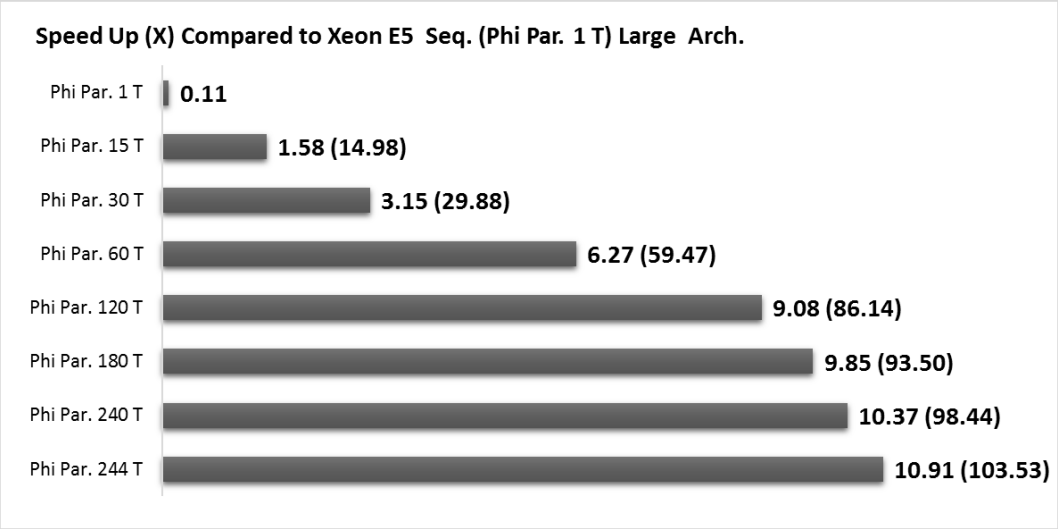

Figure 4.1.14: Speed up for the large CNN architecture as compared to *Xeon E5 Seq* (*Phi Par. 1 T*).

In *figure 4.1.15* it can be seen that *Xeon E5 Seq.* ends with an error rate of 0.0167% (about 10 images), which it shares with both *Phi Par. 15 T* and *Phi Par. 30 T*. The worst error rate is shown by *Phi par. 244 T* which ends at 0.021% (about 13 images).

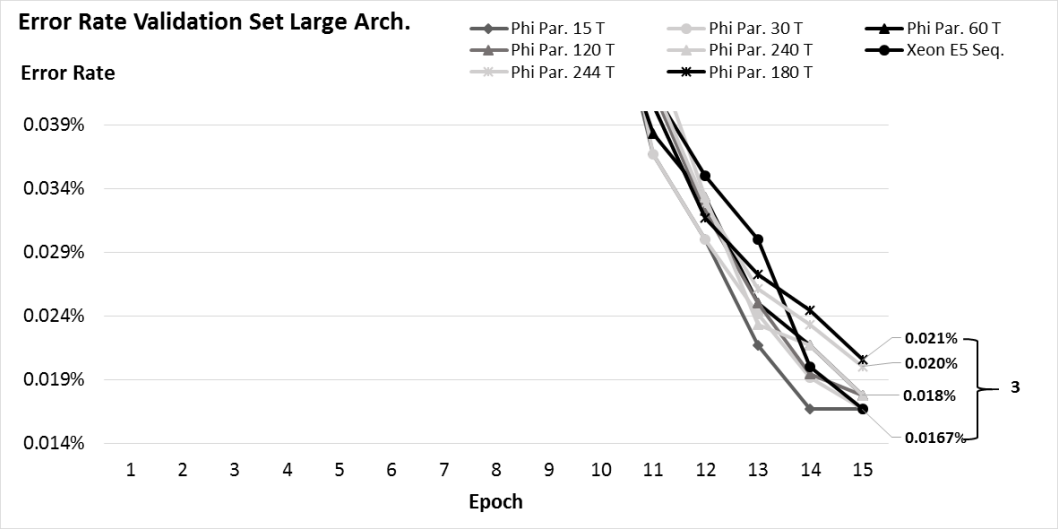

Figure 4.1.15: Error rate of the validation set for the large CNN architecture.

For the test set, in *figure 4.1.16*, the *Phi Par. 15 T* ends with 0.84 % error (about 84 images), 10 images better than optimal. *Phi Par. 244 T* ends at 0.95% (about 95 images), 1 image worse than optimal.

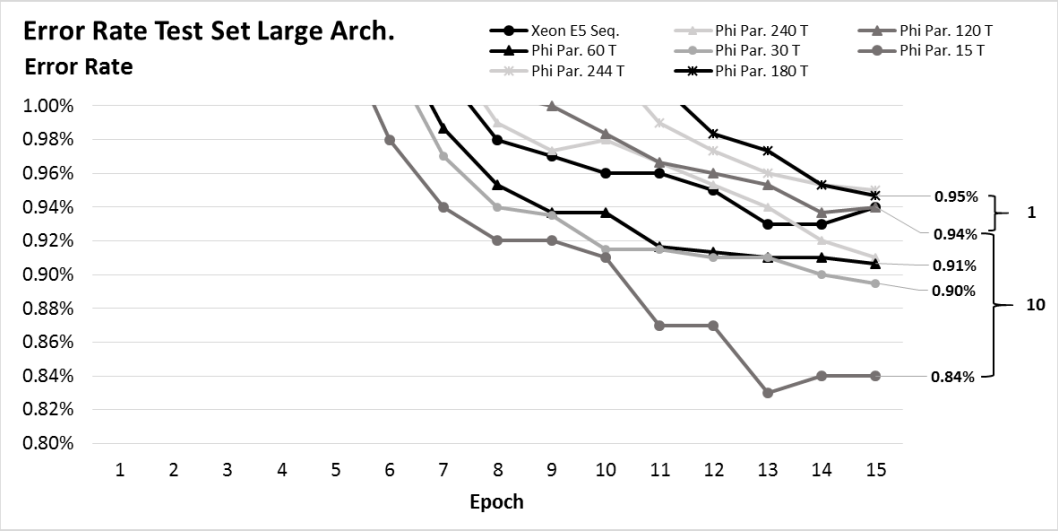

Figure 4.1.16: Error rate of the test set for the large CNN architecture.

In *figure* 4.1.17, the errors on the validation set is shown. As can be seen, the resulting error is about 180.14 for all configurations. It seems that the resulting error can be even lower as the curve is rather steep at the end.

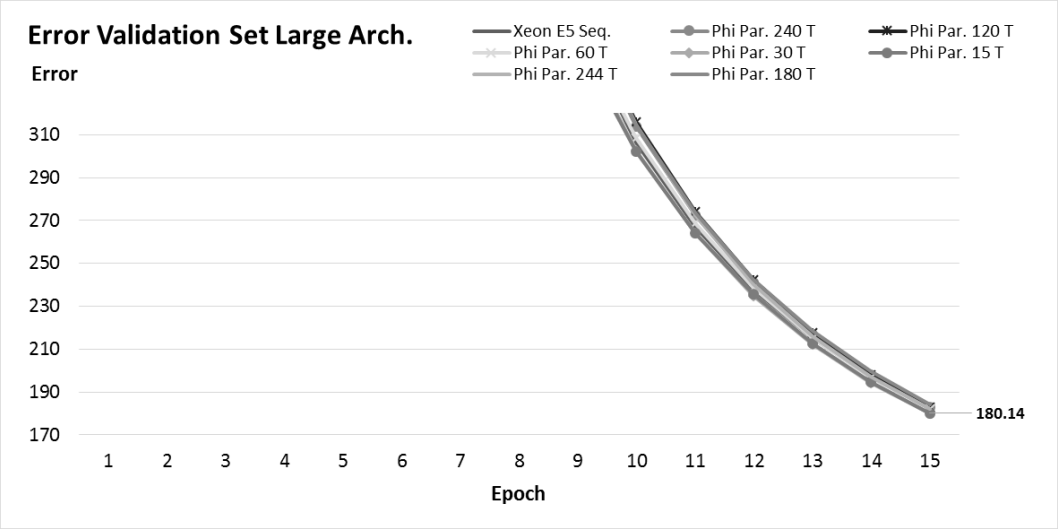

Figure 4.1.17: Error of the validation set for the large CNN architecture.

The error on the test set can be seen in *figure 4.1.18*. *Xeon E5 Seq.* ends at 290.0. The lowest error has *Phi Par. 15 T* with 281.2, about 9 better than the base line. Worst is *Phi Par. 120 T*, at 294.9, 4.9 worse than optimal.

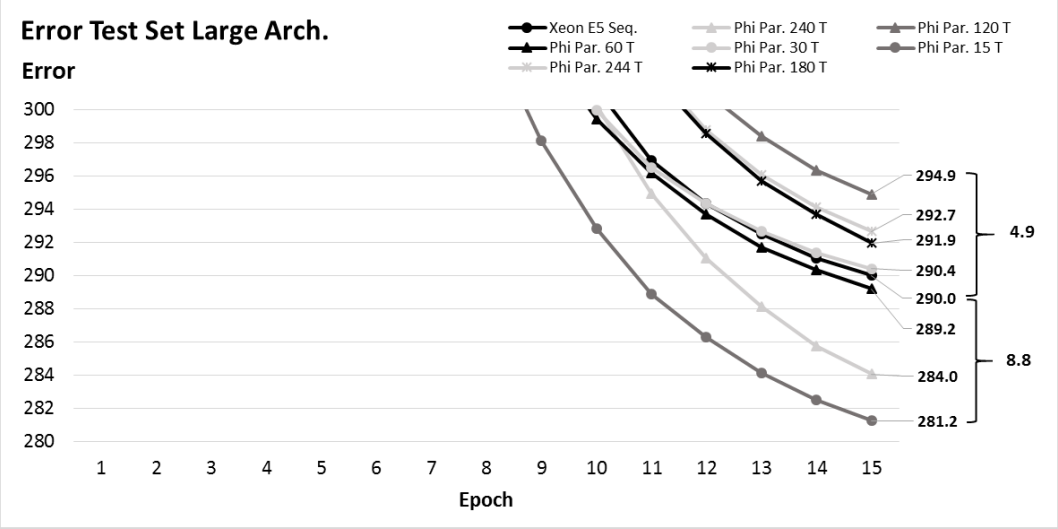

Figure 4.1.18: Error of the test set for the large CNN architecture.

4.2 Time Spent on Each Layer

In this section the layer times are presented, i.e. the time spent per layer, epoch and network instances, on average. The results were gathered as the total time spent for all network instances on all layers together. Dividing the total time by the number of network instances and later the number of epochs, yield the number of seconds spent on each layer per network instance and epoch. A lower time spent on each layer per epoch and instance indicates on a speed up.

In *table 4.2.1*, layer times for the small architecture is depicted. It can be seen that the convolutional layers are far more costly than the others, about 84-85 % of the time is spent here. Back-propagation in the fully connected layer is also time consuming, requiring up

to 10% of the time. Perhaps more interesting is that the time is decreasing as adding more threads and that this is true even for the convolutional layers.

	BPFull		BPConv		FPConv		FPFull	
	s	%	s	%	s	%	s	%
Core i5 Seq.	4.476	1.3%	127.7	37.1%	199.2	57.8%	4.691	1.4%
Xeon E5 Seq.	0.441	0.6%	20.6	29.1%	46.3	65.5%	0.585	0.8%
Phi Par. 244 T	0.364	9.1%	3	74.2%	0.4	10.6%	0.046	1.2%
Phi Par. 240 T	0.376	8.9%	3.1	74.2%	0.5	10.7%	0.050	1.2%
Phi Par. 180 T	0.411	9.2%	3.3	73.8%	0.5	10.9%	0.053	1.2%
Phi Par. 120 T	0.475	9.5%	3.7	73%	0.6	11.4%	0.058	1.2%
Phi Par. 60 T	0.713	9.6%	5.3	72%	0.9	12%	0.090	1.2%
Phi Par. 30 T	1.238	8.7%	10.4	72.7%	1.7	12.2%	0.173	1.2%
Phi Par. 15 T	2.264	8.0%	20.6	73.1%	3.5	12.3%	0.352	1.2%
Phi Par. 1 T	22.160	5.5%	302.9	74.9%	51.9	12.8%	5.057	1.3%

Table 4.2.1: Average layer times for the small CNN architecture.

For the medium layer times (*figure 4.2.2*), a similar pattern as for the small architecture can be seen. Even more time is spent in the convolutional layers. Up to 90% of the time is spent here. Up to 10% of the time is spent in the fully connected layers.

	BPFull		BPConv		FPConv		FPFull	
	sec	%	sec	%	sec	%	sec	%
Core i5 Seq.	51.2	1.0%	1,626	30.6%	1,927	36.3%	44.19	0.8%
Xeon E5 Seq.	6.2	0.9%	267.5	38.9%	399.1	58.0%	4.61	0.7%
Phi Par. 244 T	3.7	7.5%	39.8	81.3%	4.5	9.2%	0.18	0.4%
Phi Par. 240 T	3.8	7.3%	41.9	81.4%	4.8	9.2%	0.19	0.4%
Phi Par. 180 T	4.2	7.6%	44.1	80.8%	5.2	9.6%	0.20	0.4%
Phi Par. 120 T	5.0	8.3%	47.9	79.4%	6.1	10.2%	0.23	0.4%
Phi Par. 60 T	7.8	9.0%	67.1	77.9%	9.3	10.8%	0.37	0.4%
Phi Par. 30 T	14.3	8.4%	133.2	78.4%	18.5	10.9%	0.73	0.4%
Phi Par. 15 T	27.6	8.1%	266.4	78.6%	37.1	10.9%	1.49	0.4%
Phi Par. 1 T	333.2	6.7%	3,946	79.6%	558.6	11.3%	23.32	0.5%

Table 4.2.2: Average layer times for the medium CNN architecture.

The large architecture spends almost all time in the convolutional layer and almost no time in the other layers, as can be seen in *table 4.2.3*. For *Phi Par. 240 T* about 88% is spent in the back-propagation of convolutional layers and about 10% in forward propagation. Again, the time spent on each layer is non-increasing for larger amount of threads.

	BPFull		BPConv		FPConv		FPFull	
	sec	%	sec	%	sec	%	sec	%
Core i5 Seq.	103.9	0.31%	16,778	49.32%	16,952	49.83%	86.73	0.25%
Xeon E5 Seq.	30.2	0.19%	7,097	44.51%	8,714	54.66%	17.04	0.11%
Phi Par. 244 T	7.8	1.36%	506.2	88.48%	54.7	9.56%	0.23	0.04%
Phi Par. 240 T	8.1	1.34%	532.2	88.45%	57.8	9.61%	0.24	0.04%
Phi Par. 180 T	9.0	1.41%	557.9	87.78%	64.8	10.20%	0.26	0.04%
Phi Par. 120 T	11.3	1.63%	598.4	86.82%	75.4	10.94%	0.28	0.04%
Phi Par. 60 T	19.5	1.91%	877.7	86.19%	114.4	11.23%	0.47	0.05%
Phi Par. 30 T	34.7	1.71%	1,749	86.36%	228.3	11.27%	0.94	0.05%
Phi Par. 15 T	60.8	1.50%	3,495	86.52%	456.9	11.31%	1.90	0.05%
Phi Par. 1 T	836.7	1.38%	52,387	86.60%	6,859	11.34%	29.75	0.05%

Table 4.2.3: Average layer times for the large CNN architecture.

For all architectures it can be seen that the more threads involved in training the more percentage of the total time each thread spends in the back-propagation of the convolutional layer, and less time in the others. Overall, the time spent at each layer is decreased per thread when increasing the number of threads. Therefore there is an interesting relationship between the layer times and the speed up of the algorithm.

Chapter 5

Theoretical Study

This chapter carries out a theoretical analysis of the algorithm. First the algorithm is described as pseudocode, annotated with the number of times each line is executed in its context. For each function, the total running time, T, is calculated from the amount of work per line and the number of times a line is executed. The results are later used when discussing the speed up and parallelism of the algorithm, and when extending the model into a performance model. Some functions have been moved to *Appendix E* to avoid superfluous information in this chapter. The aim of the theoretical study is to understand, and discuss the potential of the algorithm.

5.1 Pseudocode and Annotation

The analysis was carried out on the parallel version of the algorithm. This section presents pseudocode for the algorithm, each line is annotated with the number of invocations in its context. The pseudocode can be somewhat close to the implementation language *C++* as the analysis was performed on an existing implementation. Approaches and terminology were inspired by [13]. The terminology used in short:

- *for-loops* are written as: *for .. to ...* and *while conditions* as *while condition*. For-loops are inclusive and hence *-1* is often use to set correct bounds.
- Indentation is used to define code blocks, white-space to separate statements.
- Function calls are denoted as *object.function*. We intentionally omit arguments when appropriate, *this* is used to denote the current object and *CamelCase.function* to denote static calls.
- Plain English will be used when convenient.
- *#atomic* will be used to denote atomic operations.
- *break* denotes exiting the current loop.
- *error* indicates on an error.
- *print* "some text" indicates print outs.
- All variables are assumed to be local, objects are initialized and given a name.
- Declaration of variables are omitted.

- Switch statements are translated to corresponding *if-else* statements.
- // denote comments.
- When discussing the time complexity, operations having their own calculations are denoted as camel case, e.g. *SomeComplexOperation*, denoting the work required to carry out the operation. Also, $T_{SomeComplexOperation}$ is used to define the time required by the operation. In most cases the function also have a corresponding listing.

To annotate the code, we use a top-down approach, where each function is calculated in atomicity. For simplicity, details are omitted as often as possible, names for complex operations will replace the specifics. The analysis will focus on the *Main*, *Trainer* and *NeuralNetwork* classes. Functions explained in more detail are the most important ones; because they either occur repeatedly (once per epoch) or are entry points of the application. We define $c_{lineNumber}$ to be an arbitrary amount of work, targeting a specific line; in the calculations, constants are used initially and omitted/aggregated when no longer needed. The constants can either be thought of as a number of operations requiring a set of cycles to be retired, or as the approximated time to perform the work once. The semantics does not matter in the general context, and when it does, it will be expanded lazily.

It is further assumed that each operation takes a constant amount of time respectively, i.e. the constant time is atomic for each operation and multiplied by the number of times it is issued. Moreover, all *if-statements* are assumed to be visited, however only one of the branches in *if-else* statements are included in the calculations. Translation from complex operations are done lazily to ease the perception of the formulas, e.g. *arch* denotes an arbitrary architecture.

Table 5.1.1 shows the variables used. The variables i, it, ns and ep are used to parameterize the formulas later on and are hence the outer-most parameters. Other hidden variables exists, however are only used to describe the forward- and back-propagation shown in *Appendix E*.

Variable	Meaning
i	Number of training/validation images.
it	Number of test images.
ns	Number of networks instances.
ep	Number of epochs.
p	Number of processing units.

Table 5.1.1: Parameters used in the theoretical analysis.

Some parts of the algorithm are easier described in plain English and some using programming-alike-syntax, whatever suits the situation is used within the frames of the stated terminology. As the annotation was constructed after the implementation, it makes little sense to simplify some expressions further away from the actual implementation.

In some cases, a translation to achieve functions in functions are desirable, and will be done lazily when needed. Also, sometimes more detailed information is required for the underlying variables to make sense. E.g. by knowing the architecture under consideration it is possible to extend and perform the discussions and computations, without a known architecture it makes little sense.

The viewpoint of the analysis is on the scalability of the parallelization scheme rather than the CNN architecture. The rationale is that the parallelization scheme *CHAOS* is highly related to the network instances and images, whereas the CNN architectures is not. Two viewpoints are considered: *a)* keeping the architecture fixed, how will the algorithm scale with different network instances (processing units), epochs and images? *b)* keeping the number of images, epochs and network instances fixed, how will different CNN architectures impact the execution time? The main viewpoint is *a)*, and hence discussions in this chapter will be performed with this in mind.

In the performance model presented in *section 5.4* we investigate how varying the number of processing units and CNN architecture size affects the running time, and compare it to the measured values. In that case we replace the *arch* placeholder with more realistic values. The term processing unit is used implicitly to mean hardware thread if nothing else is stated.

5.1.1 Main Function

The main function presented in *listings 5.1* yields no parallelism, this is the entry point of the application. In the algorithm, the training can be altered by selecting the type of training, CNN architecture and number of network instances/threads.

```
1  main()
2
3  1 Let inputArgument be a struct holding the input arguments
4  1 Set default values for inputArgument
5  1 Check argument input
6  1 Read arguments from command line
7  1 Validate the arguments
8
9  // Initiate training
10 1 Let Trainer be a newly created Trainer instance
11 1 if trainingType == CHAOS
12 1   print "CHAOS"
13 1   trainer.CHAOS(inputArgument)
14 1 else
15 1   print "training␣type␣not␣implemented"
16 1   error
17
18 1 return
```

Listing 5.1: Listings of the main function.

$$T(i, it, ep, ns) = T_{Trainer}(i, it, ep, ns) + c$$

The total execution time of the algorithm is the execution time of the *Trainer* plus a constant amount of work in *main*.

5.1.2 Trainer Function

The *Trainer* class carries out the training, pseudocode is presented in *listings 5.2*. Currently it holds the *chaos* function and some helper functions (including *runTest*). The *Trainer* trains the network for a set of epochs (*ep*) and evaluates the training effort on the test (*it*) and validation set (*i*) for each epoch. Sequentially, the training is performed on one image at the time, when parallelizing the algorithm, one network instance trains a subset of the images, picking a new image from the set in each iteration as long as more images exist in the set of images not yet processed. Note that workers train a non-overlapping subset of images.

The number of executions is divided by ns in the listings as the context is the worker. Therefore some terms are abbreviated, instead of denoting ns/ns we ignore the term entirely.

```
1  chaos()
2
3  1 Let reporter be a newly initiated Reporter object
4
5  // Set ETA and name
6  1 etaStart = ETA_START // 0.001
7  1 etaFactor = ETA_FACTOR // 0.9
8  1 nName = Helper.getNetworkName(architecture)
9
10 // Create networks with shared weights
11 1 Let def be a new NeuralNetwork created for architecture of type nName with
       CreateNetwork
12 1 ReadNetworkWeights(def, nName)
13
14 1   Let nns be a new vector of size numberOfNetworks
15 ns+1 for i = 0 to nns.Length-1
16 ns     Let nn be a new NeuralNetwork created with nName with CreateNetwork
17 ns     SetNetworkWeights(nn, def)
18 ns     Add nn to nns
19
20 // Initiate and pre allocate images
21 1 Let idxh_train, idxh_validation, idxh_test be IDX handlers for the datasets
22 1 InitiateImages(idxh_train); InitiateImages(idxh_validation); InitiateImages(idxh_test);
23
24 // Loads all images to a buffer easier to manage
25 1 Allocate space for buffers trainImages, testImages and validationImages
26 1 PreAllocateImages(trainImages)
27 1 PreAllocateImages(validationImages)
28 1 PreAllocateImages(testImages)
29
30 1 PreAllocateDesiredOutput
31 1 Initialize variables
32
33 ep+1 while contEpochs == true
34 ep     set count = 0
35
36 ep*(ns+1) parallel for n = 0 to nns.Length-1
37
38 ep           nns[n].setEta(eta, etaStart, etaFactor)
39 ep           nns[n].resetLayerTimes()
40
41 ep*(i/ns+1) while true
42                  #atomic
43 ep*(i/ns+1)      myCount = count++;
44
45 ep*(i/ns+1)   if myCount >= numberTrainingImages
46 ep              break
47
48 ep*i/ns       Let label be the label associated to myCount
49 ep*i/ns       Let image be the image associated to myCount
50 ep*i/ns       Set image to input layer
51
52 // Forward propagate the image through the network
53 ep*i/ns*l     for j = 1 to numberOfLayers-1
54 ep*i/ns*(l-1)      nns[n].ForwardPropagate(j)
55
56 ep*i/ns       Let t be the desired output
57 ep*i/ns       Let y be the output of the last layer
58 ep*i/ns       Let delta be the deltas of the last layer
59
60 ep*i/ns(nll+1)for k = 0 to numberOfNeuronsLastLayer-1
61 ep*i/ns*nll     if activationTypeLastLayer == soft_max
62 ep*i/ns*nll       if y[k] < FLOAT_MIN //The float minimum
63 ep*i/ns*nll           y[k] = FLOAT_MIN
64 ep*i/ns*nll       delta[k] = y[k] - t[k]
65 ep*i/ns*nll     else if activationTypeLastLayer == scaled_tanh
66 ep*i/ns*nll       delta[k] = tanh(y[k]) * (y[k] - t[k])
67 // End for
68
```

```
69 // Backward propagate the image through the network
70 ep*i/ns*(l-1)   for l = numberOfLayers -2 to 0
71 ep*i/ns*(l-2)      nns[n].BackwardPropagate(l)
72
73 // End while true
74
75 ep  Update layer times in Reporter
76 // End for networks
77
78 ep RunTest(i,ns)
79 ep RunTest(it,ns)
80 ep contEpochs = Helper.stopTraining()
81 ep Update reporter times and errors
82 ep eta = eta * etaFactor
83 ep epoch++
84 // End while epochs
85
86 1  Update total execution times in reporter
```

Listing 5.2: Pseudocode for the trainer.

The *Trainer* needs some explanation:

- (3) - A reporter object is created to serialize execution data to disk.
- (6-7) - ETA (decay) is set for training. The ETA controls the factor of learning, i.e. how much the deltas (partial derivatives) should affect the weight parameters. The value is decreased by a factor in each epoch.
- (11-12) - A default network instance is created to hold the shared weights. The default instance has the same CNN architecture as the other instances. Weights are read from disk to the shared memory. Each instance is created from the configuration file defining the layers.
- (14-18) - Networks are created and initiated with the shared weights. The pointer in memory is set to the pointer of the weights of the default network instance.
- (21-22) - Images and labels are loaded from disk to memory.
- (25-28) - Images are pre-allocated in the memory for easier access. Even if the images and labels are accessible in the *Idx* class, their memory is allocated lazily. Instead images and labels are loaded in such a way that they are faster and easier to access, although with the penalty of more memory required.
- (30) - Desired output vectors are generated and saved in global memory for fast and easy access.
- (33-83) - The training process is carried out for the number of epochs predefined.
- (36-66) - The learning is performed in parallel by *ns* network instances, each processing a chunk of images. The chunk is not evenly divided, instead each instance picks images as long as more images are available.
- (41) - Loops until no more images to process. After increasing the current image count at line *43* each instance checks if all images are processed, and exits if that is the case.
- (53-54) - Forward propagates the image through the network, calculating the output of the final layer. Forward propagation calculates the weighted sum through an activation function at each neuron until the output layer.

- (60-66) - Calculates partial derivatives (deltas) of the last layer.
- (70-71) - Back-propagates the error through the network to adjust the weight parameters.
- (78-79) - Runs the test for the validation set and test set respectively.
- (80) - Determines whether to continue training or not.

$$
\begin{aligned}
&T_{Trainer}(i, it, ep, ns) = (T_{CreateNetwork}(arch) + c_{15})(ns + 1) \\
&+ T_{ReadNetworkWeights}(arch) + (T_{SetNetworkWeights}(arch) + C_{18}) * ns \\
&+ 2 * T_{InitateImages}(i) + T_{InitiateImages}(it) + 2 * T_{PreAllocateImages}(i) \\
&+ T_{PreAllocateImages}(it) + T_{PreAllocateDesiredOutput}(arch) \\
&+ (a + T_{RuntTest}(i, ns) + T_{RunTest}(it, ns)) * ep + b * ep * ns \\
&+ (c - 2 * T_{BPropOneLayer}(arch) - T_{FPropOneLayer}(arch)) * ep * i/ns \\
&+ (T_{FpropOneLayer}(arch) + c_{70} + T_{BPropOneLayer}(arch)) * ep * i/ns * l \\
&+ d * ep * i/ns * nll + e
\end{aligned}
$$

The above formula is the result of simplifying the summation of work on each line. Some rewrites are performed to ease perception, such as using *arch* to denote any arbitrary CNN architecture, and *FPropOneLayer* to define the forward propagation at one layer. The variable ns, i.e. the number of network instances/workers depicts that the number of images per worker will decrease as the number of workers increase. The time complexity for the complex operations are defined in their respective formula, some defined in this chapter (*main, chaos, runTest, fowardPropagation* and *backPropagation*), others defined in *Appendix E*.

In *section 5.2* our goal is to, on a high level, discuss the speed up and parallelism of the algorithm. The problem is that the execution time of our algorithm now depends on several parameters prohibiting us to safely use the standard *Big Oh* notation as discussed in [67]. Therefore we did not further simplify the formula at this point, instead we lazily do so when necessary to avoid misconceptions.

5.1.3 Forward Propagate Function

The *forwardPropagate* function, defined in *listings 5.3*, determines the target-layer function to invoke depending on the layer type at the given index.

```
1  forwardPropagate()
2  1  type = layers[idx].type
3
4  1  if type == fully_connected
5  1    ForwardPropagateFullyConnectedLayer(idx)
6  1  else if type == convolutional
7  1    ForwardPropagateConvolutionalLayer(idx)
8  1  else if type == pooling
9  1    ForwardPropagateMaxPoolingLayer(idx)
10 1  else
11 1    print "Layer type no implemnented"
12 1    error
13
14 1  Update layer times
```

Listing 5.3: Pseudocode for the forwardPropagate function.

$$T_{ForwardPropagate}(arch, idx) = c_2 + c_4 + c_6 + c_8 + c_{10} + T_{ForwardPropagateLayer}(arch, idx) + c_{14} + = T_{ForwardPropagateLayer}(arch, idx) + c'$$

where $T_{ForwardPropagateLayer}(arch, idx)$ is one of:
$T_{ForwardPropagateFullyConnectedLayer}(arch, idx)$,
$T_{ForwardPropagateConvolutionalLayer}(arch, idx)$,
$T_{ForwardPropagateMaxPoolingLayer}(arch, idx)$ or
$c_{11} + c_{12}$. These are further defined in *Appendix E*.

5.1.4 Back-propagate Function

Back-propagate, shown in *listings 5.4*, is responsible for determining the sub-routine to be invoked for the layer type at a specific index. As the algorithm determines the underlying layer type based on the index of the next layer we will use variable names accordingly, e.g. we will use *nnl* to mean neurons in layer *idx+1* when discussing calculations at a specific layer.

```
1  backpropagate()
2
3  1type = layers[idx + 1].type
4
5  1if type == fully_connected
6  1   BackpropagateFullyConnectedLayer(idx)
7  1else if type == convolutional
8  1   BackpropagateConvolutionaLayer(idx)
9  1else if type == pooling
10 1   BackpropagateMaxPoolingLayer(idx)
11 1else
12 1   print "Layer type not implemented"
13 1   error
14
15 1 Update layer times
```

Listing 5.4: Pseudocode for the backPropagate function.

$$T_{Backpropagate}(arch, idx) = c_3 + c_5 + + c_7 + c_9 + c_{11} + T_{BackpropagateLayer}(arch, idx) + c_{17} = T_{BackpropagateLayer}(arch, idx) + c$$

where $T_{BackpropagateLayer}(arch, idx)$ is one of:
$T_{BackpropagateFullyConnectedLayer}(arch, idx)$,
$T_{BackpropagateConvolutionalLayer}(arch, idx)$,
$T_{BackpropagateMaxPoolingLayer}(arch, idx)$ or
$c_{12} + c_{13}$

5.1.5 Run Test and Validation Functions

The *runTest* function (*listings 5.5*) encapsulates the *determineErrorForChunk* function which computes the error and error rates of a set of images. In our case, the chunk size is fixated to 1. The error and error rates are calculated through the *determinerError-ForChunk* function and reduced by all threads participating in the process.

```
1  runTest()
2
3  1     count = myCount = err_loc = ERR_loc = 0
4
5  ns+1   parallel for n = 0 to nns.Length-1
6  i/ns+1   while true
7               #atomic
8  i/ns+1       myCount = count++;
9
10 i/ns+1       if myCount >= numberOfImages
11 1              break
12
13 i/ns          err_loc += nns[n].DetermineErrorForChunk()
14 i/ns          ERR_loc += nns[n].getError()
15 // End while
16 //End for
17
18 1     return err_loc, ERR_loc
```

Listing 5.5: Pseudocode for the runTest function.

$$
\begin{aligned}
&T_{RunTest}(i, ns) = c_3 + c_5 + c_6 + c_8 + c_{10} + c_{11} + c_{18} + c_5 * ns \\
&+ (c_6 + c_8 + c_{10} + T_{DetermineErrorForChunk} + c_{14}) * i/ns \\
&= a * ns + (b + T_{DetermineErrorForChunk}) * i/ns + c
\end{aligned}
$$

5.2 Work, Span, Speed Up and Parallelism of the Algorithm

This section discusses the theoretical bounds of the algorithm in terms of work, span, speed up and parallelism. The work is defined as T_1, i.e. the time it takes to perform all work on one processing unit, or the sum of work on all processing units. The span is the longest of any of the paths through the algorithm denoted as T_∞; it can also be thought of the execution time given an infinite number of processing units. The running time on *p* processing units is denoted as T_p. $p == ns$ is used, i.e. one network instance per processing unit. The speed up is denoted as T_1/T_p and the parallelism as T_1/T_∞. The speed up yields a theoretical upper bound of expectation, and the parallelism how well suited the algorithm is for parallelization, i.e. what variable(s) is the parallelism bound to. A linear speed up occur when $T_1/T_p = p = ns$ [13].

The goal is to understand how well the algorithm scales with an increasing number of processing units, and how it responds when varying the number of epochs and images. More importantly, the performance model allows us to assess the implementation.

Details of the CNN architecture is omitted (a more detailed discussion is carried out in the *section 5.3*), simplifications are made by denoting the architecture as *arch*; the viewpoint *a)* as discussed previously is of interest in this analysis. Workers can carry out their work independently of each other except for the weight updates which will be shared by all workers.

Our algorithm uses *parallel for* statements to indicate parallelization, there is no need for recursive calls, or more complex syntax. The spawning and synchronization of threads are assumed to be negligible since they only contribute to the running time once per epoch and since no hardware characteristics are present, the overhead is hard to calculate.

The execution time for the *main* function was calculated previously and repeated here:

$$T(i, it, ep, ns) = T_{Trainer}(i, it, ep, ns) + c$$

The main function is simply a constant amount of work plus the work carried out by the *Trainer* for *i* training/validation images, *it* test images, *ep* epochs and *ns* network

instances. The formula was denoted earlier and repeated below to base our working formula:

$$
\begin{aligned}
&T_{Trainer}(i, it, ep, ns) = (T_{CreateNetwork}(arch) + c_{15})(ns + 1) \\
&+ T_{ReadNetworkWeights}(arch) + (T_{SetNetworkWeights}(arch) + C_{18}) * ns \\
&+ 2 * T_{InitateImages}(i) + T_{InitiateImages}(it) + 2 * T_{PreAllocateImages}(i) \\
&+ T_{PreAllocateImages}(it) + T_{PreAllocateDesiredOutput}(arch) \\
&+ (a + T_{RuntTest}(i, ns) + T_{RunTest}(it, ns)) * ep + b * ep * ns \\
&+ (c - 2 * T_{BPropOneLayer}(arch) - T_{FPropOneLayer}(arch)) * ep * i/ns \\
&+ (T_{FpropOneLayer}(arch) + c_{70} + T_{BPropOneLayer}(arch)) * ep * i/ns * l \\
&+ d * ep * i/ns * nll + e
\end{aligned}
$$

Where $T_{BpropOneLayer}(arch)$ and $T_{FPropOneLayer}(arch)$ is the time for back- and forward-propagation respectively for CNN architecture *arch* on an arbitrary layer. $T_{CreateNetwork}(arch)$ is performed one time per network instance and one time for the default instance. $T_{ReadNetworkWeights}(arch)$ is the time to read the network weights from disk into memory. $T_{SetNetworkWeights}(arch)$ is the time to set the pointer of each layer to the memory address of the shared weights, and $T_{InitiateImages}$ and $T_{PreAllocateImages}$ to prepare the images in memory. $RunTest$ performs validation on the test- and validation-sets.

5.2.1 Amount of Work Required by the Algorithm

To ease discussion, we simplify the original formula to a working formula, removing superfluous information. The variables used below have no direct relation to the ones in the original formula. We define the working formula as:

$$
\begin{aligned}
&T_p(i, it, ep) = a * p + 4 * b * i + 2 * c * it \\
&+ d * ep + e * ep * p + \left(\frac{(f + T_{FProp} + T_{BProp}) * i}{p_i}\right) * ep \\
&+ \left(\frac{(g + T_{FProp}) * i}{p_i}\right) * ep + \left(\frac{(h + T_{FProp}) * it}{p_{it}}\right) * ep + w
\end{aligned}
$$

Terms performing a constant amount of work are aggregated and hidden to ease the calculations. Remember that restrictions were put on the CNN architecture being fixed and hence operations related to the architecture requires a constant amount of work even when changing the input parameters; although the constant is certainly large, it is still a constant. The formula is also extended to forward- and back-propagation level, T_{FProp} and T_{BProp} denotes the time to forward- and back-propagate an image through the network, respectively. $T_{CreateNetwork}$, $T_{SetNetworkWeights}$ and some other terms are aggregated into a. $T_{InitiateImages}$ and $T_{PreAllocateImages}$ is exchanged with constants b and c to denote that the time to prepare images are 2 times each set, where each image require a constant amount of work. We introduce p_i and p_{it} to denote that p cannot exceed i or it, i.e. $p_i \leq i \wedge p_{it} \leq it$ and thus $p_i = min(p, i) \wedge p_{it} = min(p, it)$. This also put restrictions on that the work over epochs cannot be lowered by adding more threads as this still needs to be carried out sequentially for ep epochs. Also we exchanged ns with p to denote the number of processing units. Even though it is possible to simplify further, we see no point

in doing so. From the working, formula T_1 is denoted as:

$$T_1(i, it, ep) = a + 4 * b * i + 2 * c * it \\ + d * ep + e * ep + (f + T_{FProp} + T_{BProp}) * i * ep \\ + (g + T_{Fprop}) * i * ep + (h + T_{FProp}) * it * ep + w$$

As the formula is annotated with $p = ns$ it is possible to use $p = ns = 1$ to determine the amount of work required by the algorithm. The algorithm carries out a constant amount of work a, w, some minor work per epoch, and work dependent on the number of images in the test- and validation sets. That is, the preparation of the images and labels and forward- and back-propagation of images. The total amount of work is independent of the number of processing units, as more processing units only share the total amount of work. The asymptotic notation can be defined as: $\mathcal{O}(i)$ if $it < i$ for a fixed ep, indicating that the execution time grows by the number of images.

5.2.2 Span of the Algorithm

Before defining the span, it can be seen that the parallelism cannot exceed the number of images *i* - the speed up is likely to decrease after reaching $p > it$. Another observation is as *p* increases, the time for *a* which is mainly the creation of network instances also increases. At the same time the $e * ep * p$ increase, however, this is really small in comparison. The time spent in each epoch for the forward- and back-propagation is decreasing when adding more processing units since each instance trains a smaller set of images. Without detailed calculations, the time required for the creation of the instances is far less than the time to forward propagate i images $2 * ep$ times, back-propagate them ep times, and forward propagate it images ep times. It would require really large and wide networks training a small set of images for a low number of epochs. Moreover, although not currently implemented, the creation of network instances could be parallelized as well and hence each processing unit would create its own instance - an easy modification to the existing algorithm. We could mitigate the time to create instances by just transforming $a * ns$ to $a * ns/ns = a$, and therefore this will be omitted further on in this chapter, if not stated otherwise.

In *figure5.2.1*, an overview of the algorithm is shown using call outs to denote the time complexity for different operations. Dashed lines denote the critical path through the algorithm. As each processing unit carries out equal amount of work, doing so in parallel reduces the overall computations required per worker, the shortest execution time depends on the slowest worker. Here, the creation of network instances is not parallelized. The span can be thought of as the sequential amount of work required to initialize images and labels, and other variables necessary, plus the maximum time for each network instances to carry out its intended amount of work in training, validation and testing. If applying infinite number of processing units, what remains are the initial amount of work and the maximum time spent by each processing unit to process its chunk of images. The call outs makes the figure easier to relate to the simplified working formula.

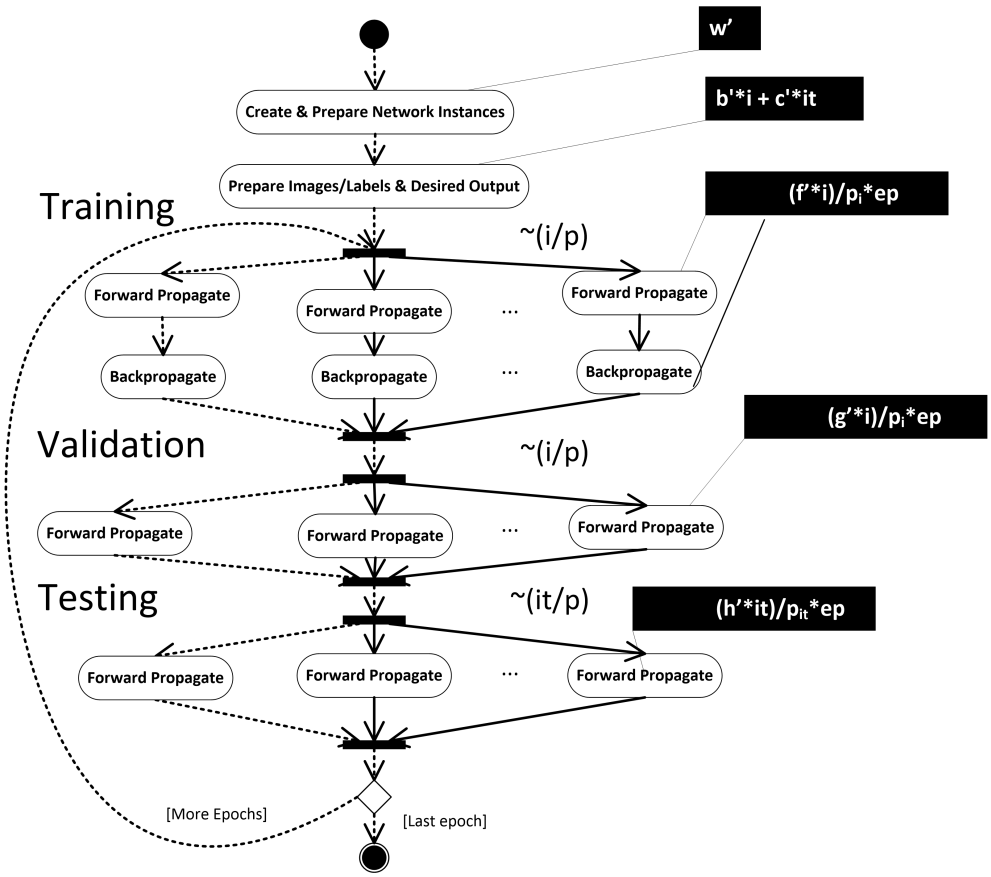

Figure 5.2.1: An overview of *CHAOS*.

Before making an attempt to define the span we first simplify T_p one last step. It interests us how the execution time for the algorithm changes with the number of processing units primarily, yet keeping the epochs and images in the equation. We do not remove anything, simply exchange terms with abbreviations to avoid excessive information:

$$
\begin{aligned}
&T_p(i, it, ep) = a * p + 4 * b * i + 2 * c * it \\
&+ d * ep + e * ep * p + \left(\frac{(f + T_{FProp} + T_{BProp}) * i}{p_i}\right) * ep \\
&+ \left(\frac{(g + T_{Fprop}) * i}{p_i}\right) * ep + \left(\frac{(h + T_{FProp}) * it}{p_{it}}\right) * ep + w \\
&= (b' * i + c' * it + d * ep) + \left(\frac{f' * i}{p_i}\right) * ep + \left(\frac{g' * i}{p_i}\right) * ep + \left(\frac{h' * it}{p_{it}}\right) * ep + w'
\end{aligned}
$$

The work by a is aggregated in to w' as each instance creates itself. The work divisible by the number of processing units (forward- and back-propagation) are aggregated into f', g', h'. We remove $e * ep * p$ as this is only the work to spawn the threads and does not impact the execution time as we omitted the synchronization in our theoretical model.

As each processing unit clearly cannot process less than one image, denoted above, the ideal case for the parallel parts of the algorithm would occur for $p == i$, one worker process one image. This infers that the critical path is the sequential amount of work plus the work by each network instance: train one image, test one image, and validate one image, as these cannot be separated. From the simplified working formula the span is

defined as:

$$T_\infty(i, it, ep) = (b' * i + c' * it + w') + (d + f' + g' + h') * ep$$

i/p_i and it/p_{it} can be removed since each instance trains, validates and tests one image each. Also, as can be seen, some work is carried out once $(b' * i + c' * it + w')$ and some per epoch $(d + f' + g' + h') * ep$. The equation shows that if infinite number of threads are available, the sequential amount of work will depend on the number of images, and the parallel part will decrease to the time of processing one images for ep epochs.

5.2.3 Speed Up of the Algorithm

The speed up is the ratio between execution times of T_1 and T_p, i.e. how much faster p processing units are than one processing unit to carry out the work. From the simplified T_p we derive the simplified T_1:

$$T_1(i, it, ep) = (b' * i + c' * it + w') + (d + f' * i + g' * i + h' * it) * ep$$

The speed up, T_1/T_p is the total amount of work divided by the work as carried out by p processing units:

$$S_p = \frac{T_1}{T_p} = \frac{(b' * i + c' * it + w') + (d + f' * i + g' * i + h' * it) * ep}{(b' * i + c' * it + w') + \left(d + \left(\frac{f' * i}{p_i}\right) + \left(\frac{g' * i}{p_i}\right) + \left(\frac{h' * it}{p_{it}}\right)\right) * ep}$$

Grouping the terms and factor out ep clarify the speed up - if $p_i == p_{it} == 1$, the speed up is one. Increasing the number of processing units will decrease the denominator and therefore increase the speed up. However, the speed up will not be linear as the overhead from the sequential amount of work will be a limiting factor. If increasing the number of images, we could expect the right term to outgrow the left term as the constants f', g' and h' are larger than b' and c' and thereby diminish the impact of the left term. Another observation is that ep adds a factor to the right term, indicating that the left term will have less impact for larger epoch counts.

To visualize the theoretical speed up, the formula was parameterized with: *a)* ep=*70*, i=*60,000*, it=*10,000* and *b)* ep=*70*, i=*120,000*, it=*20,000*, and increasing *p* values. Moreover, the values of the constants were set to values approximately relative to each other. As can be seen, the speed up diminishes after *60,000* and *120,000* images respectively as it is no longer possible to divide the images over available instances. Moreover, it can be seen that an almost linear speed up is encountered up to 10,000 threads for *a)* and 20,000 threads for *b)*. After this point only the training and validation can be sped up, not the testing, and hence the speed up increase more slowly. The values used are approximations and number of threads overestimated in reality. However, the plot depicts the expected behaviour when increasing the number of threads. More importantly, the algorithm seem to scale well with the number of threads until reaching i. If focusing on less than it threads, which is a more realistic scenario, the algorithm seem to encounter an almost linear speed up, in theory.

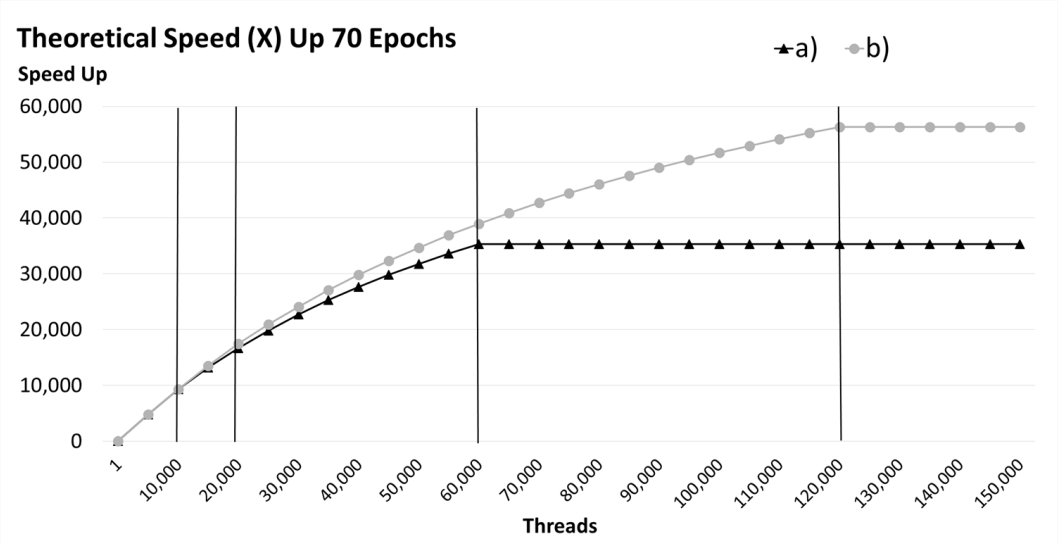

Figure 5.2.2: Instantiation of the theoretical speed up model.

5.2.4 Parallelism of the Algorithm

We end this section by denoting the parallelism as:

$$\frac{T_1}{T_\infty} = \frac{(b' * i + c' * it + w') + (d + f' * i + g' * i + h' * it) * ep}{(b' * i + c' * it + w') + (d + f' + g' + h') * ep}$$

The parallelism is bound to the number of images. The left term is present in both the denominator and numerator, when increasing the number of images they will grow in a similar fashion, however, the right term in the numerator will grow faster as f', g' and h' are larger than b' and c', and also include ep. For large i and it, the left term in the numerator will diminish. Again assuming $it \leq i$ we can asymptotically state the parallelism as a function that is bound by the number of images: $\theta(i)$.

Note that the magnitude of forward- and back-propagation hidden in f', g' and h' does not seem to impact the parallelism theoretically, just increase the amount of work, and hence the total execution time. To conclude: given an infinite number of processing units, the maximum achievable speed up in theory is highly bound to the number of images i.

5.3 CNN Architectures

In our original viewpoint, referred to as *a)* we omitted the underlying architecture entirely. In this section we will briefly discuss what we could expect from the different architectures. For each architecture we will define the aggregated amount of operations for forward- and back-propagation on the different layers. These are calculated from the number of maps, weights, map sizes and kernel sizes from the detailed listings in *Appendix E*. Further on we will assume each operation to consume an arbitrary factor of instructions in order to compare architectures. This is far from accurate as operations will infer different number of instructions, however it allows for comparison of architectures. The *table 5.3.1* shows the results of inserting the parameters for the different architectures into the forward propagation formula. The ratio shows the increase as compared to the previous architecture. About 10 times more operations are carried out at each step when increasing the architecture size.

	Max Pooling	Fully Connected	Convolutional	Total	Ratio
Small	7k	5k	46k	58k	-
Medium	29k	56k	474k	559k	9.64
Large	99k	137k	5,113k	5,349k	9.57

Table 5.3.1: Number of operations when forward propagating one image for different CNN architectures.

In *table 5.3.2* the approximated number of operations needed to backpropagate one image is shown. At each step the number of operations increase by a factor of about 12.

	Max Pooling	Fully Connected	Convolutional	Total	Ratio
Small	2k	10k	512k	524k	-
Medium	4k	112k	6,003k	6,119k	11.68
Large	8k	274k	72,896k	73,178k	11.96

Table 5.3.2: The number of operations when back-propagating one image for different CNN architectures.

5.4 Performance Model

The theoretical model presented in the previous *section 5.2* have some practical limitations. In this section we make an attempt to include the wait times (memory latency, synchronization overhead) in a performance model, escalating the theoretical model one step closer to the reality. Adding memory latencies, and synchronization overhead to the theoretical model allow to plot the measured and predicted execution times and thereby evaluate the theoretical model and the implementation of the parallelization scheme. Additionally it allow us to claim the execution time for increasing thread counts, and changing number of epochs and images.

5.4.1 Design of the Performance Model

The performance model was derived by extending the theoretical model. Below is the non-simplified working formula repeated:

$$
\begin{aligned}
&T_p(i, it, ep) = a * p + 4 * b * i + 2 * c * it \\
&+ d * ep + e * ep * p + \left(\frac{(f + T_{FProp} + T_{BProp}) * i}{p_i}\right) * ep \\
&+ \left(\frac{(g + T_{Fprop}) * i}{p_i}\right) * ep + \left(\frac{(h + T_{FProp}) * it}{p_{it}}\right) * ep + w
\end{aligned}
$$

The goal is to construct a parameterized model with ep, i, it and p as parameters. The ambition is not to create a model to scale well over several nodes and a large number of threads, rather to evaluate the theoretical model and relate it to the actual measurements.

The total execution time depends on several factors including: speed, number of processing units, communication costs (such as network latency), and memory contention.

As the current algorithm is not implemented to facilitate message-passing between processing units, this is omitted. Most interesting is contentions causing wait times, including memory latencies and synchronization overhead. A time penalty referred to as T_{mem} is added to the model, adding memory and synchronization overhead. The contention is measured through an experimental approach by executing a small script on the coprocessor for different thread counts, weights and layers. The full set of variables are shown in *table 5.4.1*.

Variable	Explanation
	Parameters
p	Number of processing units
i	Number of training/validation images
it	Number of test images
ep	Number of epochs
	Constants - hardware dependent
CPI	Best theoretical CPI/thread
s	Speed of processing unit
OperationFactor	Operation factor
	Measured - hardware dependent
MemoryContention	Memory contention
T_{Fprop}+	Forward propagation / image (ms)
T_{Bprop}+	Back-propagation / image (ms)
T_{Prep}+	Time for preparations
	Calculated - hardware independent
*FProp**	# FProp Operations / image
*BProp**	# BProp Operations / image
*Prep**	# Operations carried out for preparations
* The parameter is only used in prediction a)	
+ The parameter is only used in prediction b)	

Table 5.4.1: Variables used in the performance model.

Term	Value
Epochs (ep)	70 (small, medium), 15 (large)
Images (i)	60,000
Images (it)	10,000
Processing units/threads (p)	1 - 3,840
FProp	See *table 5.3.1*
BProp	See *table 5.3.2*
Prep	Small: 10^9 Medium: 10^{10} Large: 10^{11}

Table 5.4.2: Hardware independent parameters used in the performance model.

Parameter	Intel Xeon Phi
s	1.238 GHz
Max processing units available (p)	244 (240 used for prediction)
T_{Fprop}(ms)	Small: 1.45 Medium: 12.55 Large: 148.88
T_{Bprop}(ms)	Small: 5.3 Medium: 69.73 Large: 859.19
T_{Prep}(s)	Small: 12.56 Medium: 12.7 Large: 13.5
CPI	1-2 threads: 1; 3 threads: 1.5; 4 threads: 2
MemoryContention	Table *5.4.4*
OperationFactor	Small: 15 Medium: 15 Large: 15

Table 5.4.3: Hardware specific parameters used in the performance model.

# Threads	Small	Medium	Large
1	$7.10 * 10^{-6}$	$1.56 * 10^{-4}$	$8.83 * 10^{-4}$
15	$6.40 * 10^{-4}$	$2.00 * 10^{-3}$	$8.75 * 10^{-3}$
30	$1.36 * 10^{-3}$	$3.97 * 10^{-3}$	$1.67 * 10^{-2}$
60	$3.07 * 10^{-3}$	$8.03 * 10^{-3}$	$3.22 * 10^{-2}$
120	$6.76 * 10^{-3}$	$1.65 * 10^{-2}$	$6.74 * 10^{-2}$
180	$9.95 * 10^{-3}$	$2.50 * 10^{-2}$	$1.00 * 10^{-1}$
240	$1.40 * 10^{-2}$	$3.83 * 10^{-2}$	$1.38 * 10^{-1}$
480*	$2.78 * 10^{-2}$	$7.31 * 10^{-2}$	$2.73 * 10^{-1}$
960*	$5.60 * 10^{-2}$	$1.47 * 10^{-1}$	$5.46 * 10^{-1}$
1,920*	$1.12 * 10^{-1}$	$2.95 * 10^{-1}$	1.09
3,840*	$2.25 * 10^{-1}$	$5.91 * 10^{-1}$	2.19

* Predicted values.

Table 5.4.4: Measured and predicted memory contention (s) for the Intel Xeon Phi.

We define $T_{mem}(ep, i, p) = \frac{MemoryContention * ep * i}{p}$ where $MemoryContention$ is the measured memory contention when p threads are fighting for the weights I/O concurrently, the values are depicted in *table 5.4.4*. In *table 5.4.1* parameters used in the performance model are depicted, some are hardware dependent others independent of the underlying hardware. Each parameter is either measured, calculated, a constant or a parameter in the model. *Table 5.4.3* shows the parameters used specific for the Intel Xeon Phi and *table 5.4.2* shows parameters independent of the hardware.

In essence two strategies are provided: strategy *a)* is not concerned with any practical measurements except for T_{mem}. An approximation of the number of operations required for the calculations are done, w.r.t. the calculated values in *section 5.3* - each operation can be thought of as an arbitrary amount of instructions. Along with the *CPI* and *OperationFactor* it is possible to derive the number of instructions (theoretically) per cycle

that each thread can perform. Strategy *b)* applies the measurements for the sequential work, and the forward- and back-propagation when measured sequentially executing the algorithm.

Strategy *a)* of the formula is defined as:

$$
\begin{aligned}
&T(i, it, ep, p, s) = T_{comp}(i, it, ep, p, s) + T_{mem}(ep, i, p) \\
&= \Bigg(\frac{Prep + 4 * i + 2 * it + 10 * ep}{s} \\
&+ \Bigg(\Big(\Big(\frac{FProp + BProp}{s} \Big) * \frac{i}{p_i} * ep \Big) + \Big(\Big(\frac{FProp}{s} \Big) * \frac{i}{p_i} * ep \Big) \\
&+ \Big(\Big(\frac{FProp}{s} \Big) * \frac{it}{p_{it}} * ep \Big) \Bigg) * CPI \Bigg) * OperationFactor + T_{mem}(ep, i, p)
\end{aligned}
$$

Note that the constants are high approximations, they are relative to each other yet far from precise. We use $Prep$ to be different for each CNN architecture (10^9, 10^{10} and 10^{11} for small, medium and large architecture respectively) and to denote the number of operations required to create network instances, prepare weights, etc. Additionally, b', c' can be thought of as having a factor of 1, i.e. the work per epoch (not concerned with network instances) is 10 times more costly than the work to prepare/initialize one image. Constants f, g, h are omitted as they diminish with $Fprop$ and $BProp$. The $OperationFactor$ is adjusted to closely match the measured value for 15 threads, and mitigate the approximations done for instructions in the first place, at the same time account for vectorization.

When one hardware thread is present per core, one instruction per cycle can be assumed. For 4 threads per core, only 0.5 instructions per cycle can be assumed per thread - each thread gets to execute two instructions every fourth cycle (CPI of 2) and hence we use the CPI factor to control the best theoretical amount of instructions a thread are able to retire. The speed s is defined in *table 5.4.3*. $FProp$ and $BProp$ are placeholders for the actual number of operations shown in *tables 5.3.1* and *5.3.2* respectively.

For strategy *b)* the following formula is used:

$$
\begin{aligned}
&T_p(i, it, ep) = T_{comp}(i, it, ep, p) + T_{mem}(ep, i, p) = \\
&T_{prep} + \Bigg(\Big(\big(T_{FProp} + T_{Bprop} \big) * \frac{i}{p_i} * ep \Big) + \Big(T_{FProp} * \frac{i}{p_i} * ep \Big) \\
&+ \Big(T_{FProp} * \frac{it}{p_{it}} * ep \Big) \Bigg) * CPI + T_{mem}(ep, i, p)
\end{aligned}
$$

Where T_{prep} is the measured time it takes to prepare the training (small:12.56s, medium 12.7s and large 13.5s) and T_{Fprop}, T_{BProp} the time it takes to forward- and back-propagate one image through the network.

5.4.2 Evaluation of the Performance Model

This section aims to evaluate the predicted values using the performance model w.r.t. the measured values in the experimental study. Also, the number of epochs and images are scaled to predict execution time for larger datasets and longer training.

In *figure 5.4.1* the predicted and measured execution times are depicted. Values for *b)* have a dark-gray background, measured values have a black background, and *a)* has a light-gray background. For the small network, the predictions are pretty close to the measured values with a slight deviation at the end. The prediction models seem to overestimate the execution time a with a small factor.

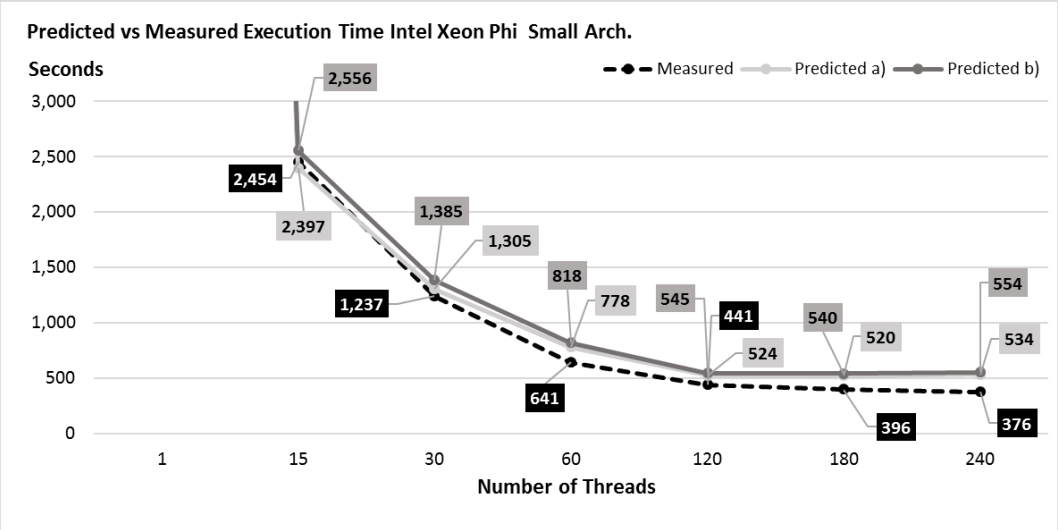

Figure 5.4.1: Predicted vs Measured execution times for Intel Xeon Phi using the small CNN architecture.

For the medium CNN architecture, depicted in *figure 5.4.2*, the measured execution time is similar to both predicted *a)* and predicted *b)*. At 120 threads, the measured and predicted values starts to deviate, which is recovered at 240 threads. Overall the prediction follow the measured values closely, although it underestimates the execution time slightly, in contrast to the small, which made an overestimation.

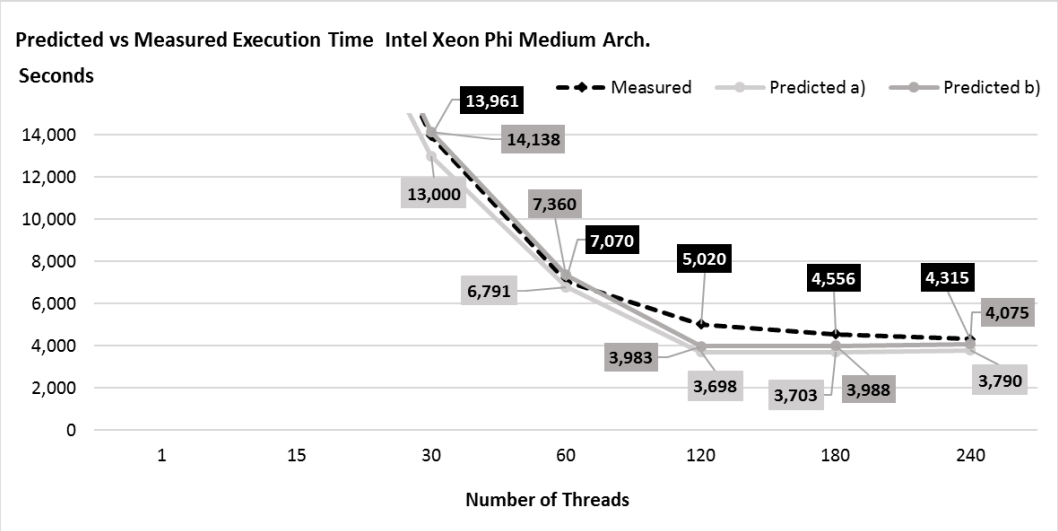

Figure 5.4.2: Predicted vs Measured execution times for the Intel Xeon Phi using the medium CNN architecture.

The large CNN architecture (*figure 5.4.3*) yields similar results as the medium. As can be seen, the measured values are slightly higher than the predictions, however, the

predictions follow the measured values. As can be seen for 120 threads there is a deviation which is recovered, more or less, for 240 threads. Also, the predictions increase between 120 and 180, and 180 and 240 threads for both predictions whereas the actual execution time is lowered. This is most probably due to the *CPI* factor that is added when 3 or more threads are present on the same core. If adding more steps in the model (more thread counts), the transition is expected to be smoother. With larger steps in the graph, the values become disjointed.

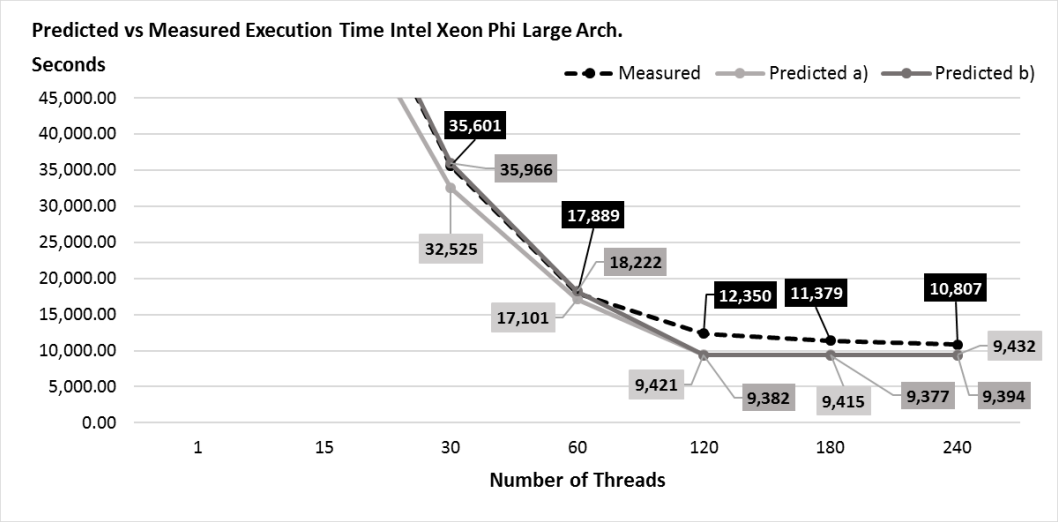

Figure 5.4.3: Predicted vs Measured execution times for Intel Xeon Phi using the large CNN architecture.

By calculating the difference between measured and predicted values and divide the difference over the predicted value yields the relative error in prediction. The averaged value over all measured thread counts (1, 15, 30, 60, 120, 180, 240 and 244 threads) are depicted in table *5.4.5*. Mathematically we formulate $x = \frac{|m-p|}{p}$ to derive the deviation in prediction where x denote the deviation, m the measured value, and p the predicted value.

Small		Medium		Large	
a)	b)	a)	b)	a)	b)
14.57%	16.35%	14.76%	7.48%	15.36%	10.22%

Table 5.4.5: Averaged deviation in predictions for both prediction models and all considered CNN architectures.

In *table 5.4.6*, the same prediction models were used to predict the execution times for *480, 960, 1,920* and *3,840* threads for the different CNN architectures, otherwise using the same parameters. According to the predictions, if *3,840* threads were available, the small network should take about *4.6* minutes to train, the medium *14.2-14.5* and the large *18-36.8* minutes. It can be seen that the predictions for the large CNN architecture are not as well aligned when increasing to larger thread counts as for small and medium. The speed up for prediction *b)* is more prominent than that of prediction *a)*. However, in these predictions we did not account of the deviation as calculated above.

	Small		Medium		Large	
	a)	b)	a)	b)	a)	b)
480	6.6	6.7	36.8	39.1	92.9	82.6
960	5.4	5.5	23.9	25.1	60.8	45.7
1,920	4.9	4.9	17.4	18.0	44.8	27.2
3,840	4.6	4.6	14.2	14.5	36.8	18.0

Table 5.4.6: Predicted execution times (min) for 480, 960, 1,920 and 3,840 images using the performance models.

Additionally we evaluated the execution time for varying image counts, and epochs, for 240 and 480 threads for the small CNN architecture and prediction *a)*. As can be seen in *table 5.4.7* doubling the number of images or epochs, approximately double the execution time. However, doubling the number of threads does not halve the execution time.

240 Threads					
Images x1000		Epochs			
i	it	70	140	280	560
60	10	8.9	17.6	35.0	69.7
120	20	17.6	35.0	69.7	139.3
240	40	35.0	69.7	139.3	278.3
480 Threads					
Images x1000		Epochs			
i	it	70	140	280	560
60	10	6.6	12.9	25.6	51.1
120	20	12.9	25.6	51.1	101.9
240	40	25.6	51.1	101.9	203.6

Table 5.4.7: Execution time (minutes) when scaling epochs and images for 240 and 480 threads using the small CNN architecture.

Chapter 6

Results Analysis

The aim of this chapter is to analyse the results derived experimentally and theoretically in an objective manner to answer the research question *RQ1*.

6.1 Experimental Analysis

The experimental results are presented in *chapter 4* including speed up, error and error rates for the small, medium and large CNN architectures. Additionally the average time spent on each layer for each network instance was presented in *section 4.2*.

All experiments were carried out on the MNIST [25] dataset of handwritten digits, comprising 60,000 training/validation images and 10,000 test images. The details of the CNN architectures can be found in *table 6.1.1*. The small and medium networks were trained for 70 epochs and the large for 15, using a starting decay of *0.9* and factor of *0.001*. The affinity for OpenMP was set to the default of the coprocessor, *scatter*.

For the experiments, three platforms were used with different characteristics and performance: The Core i5 661 has the lowest performance with 3.33 GHz, 4 logical (2 physical) cores and 4GB of memory. The Intel Xeon Phi 7120p comprises 61 cores, each able to handle 4 hardware threads, running at 1.2 GHz, with 16 GB memory. The Xeon E5-2695v2 has 12 physical cores, in total 48 logical, with a clock frequency of 2.4 GHz and memory size of 132GB.

Each parallel configuration were executed three times for 1, 15, 30, 60, 120, 180, 240 and 244 threads on Intel Xeon Phi and Xeon E5 (with some exceptions, refer to *Appendix D*) and results were averaged. Additionally, the sequential version were evaluated on the Xeon E5 and Core i5 661.

In the figures and tables we use some abbreviations due to limited space. *Par* is the parallel version, *Seq* the sequential. Also *T* denote threads. E.g. *Phi Par. 1 T* is the parallel version and one thread on the Xeon Phi.

	Type	Maps	Map Size	Neurons	Kernel Size	Weights
	Input	-	29x29	841	-	-
	Conv	5	26x26	3,380	4x4	85
	Max	5	13x13	845	2x2	-
Small	Conv	10	9x9	810	5x5	1,260
	Max	10	3x3	90	3x3	-
	Full	-	50	50	-	4,550
	Output	-	10	10	-	510
	Input	-	29x29	841	-	-
	Conv	20	26x26	13,520	4x4	340
	Max	20	13x13	3,380	2x2	-
Medium	Conv	40	9x9	3,240	5x5	20,040
	Max	40	3x3	360	3x3	-
	Full	-	150	150	-	54,150
	Output	-	10	10	-	1,510
	Input	-	29x29	841	-	-
	Conv	20	26x26	13,520	4x4	340
	Max	20	26x26	13,520	1x1	-
	Conv	60	22x22	29,040	5x5	30,060
Large	Max	60	11x11	7,260	2x2	-
	Conv	100	6x6	3,600	6x6	216,100
	Max	100	2x2	900	3x3	-
	Full	-	150	150	-	135,150
	Output	-	10	10	-	1,510

Table 6.1.1: CNN architectures used in evaluation.

6.1.1 Execution Time and Speed Up

The primary goal of the study is to lower the execution time for the working algorithm through parallelism. Perhaps most interesting is the total execution time of the algorithm as seen in *figure 6.1.1*. It shows that the execution time increase for each CNN architecture size for all thread counts on the Xeon Phi, however, adding more threads to a fixed architecture size decreases the execution time. Therefore, the scalability of the algorithm seems to be promising.

In *figure 6.1.1* it can be seen that Xeon E5 spends *31 hours* training the large network. The Phi only need about *3 hours* to complete the training. Training the large network on Xeon Phi is possible during an *afternoon*, on the Xeon E5 it requires *one-and-a-half-day*. In addition, not shown in the chart, the Core i5 trains the network *for a week*. Moreover, the small network requires *6 minutes* (a coffee break) training for 244 threads on the Xeon Phi and *1 hour and 24 minutes* on the Xeon E5.

It should be considered that the small, medium and large CNN architecture was not trained for the same amount of epochs, and that larger networks tend to produce better predictions (better ending error rates). A more fair comparison would be to compare the execution times until reaching a specific error rate on the test set. In *figure 6.1.2* the total execution times for the different CNN architectures and threads on the Xeon Phi is shown, setting the stop criteria as the *error rate* $\leq 1.54\%$, the ending error rate of the test set for the small architecture. It can be seen that the large network executes for a longer period of time even if it converges in fewer epochs, and that the medium network actual needs less time to reach an equal (or better) ending error rate than the small and large network. Many different stop criterion could be used for comparison, however, we will stick to the number of epochs in the analysis from now on.

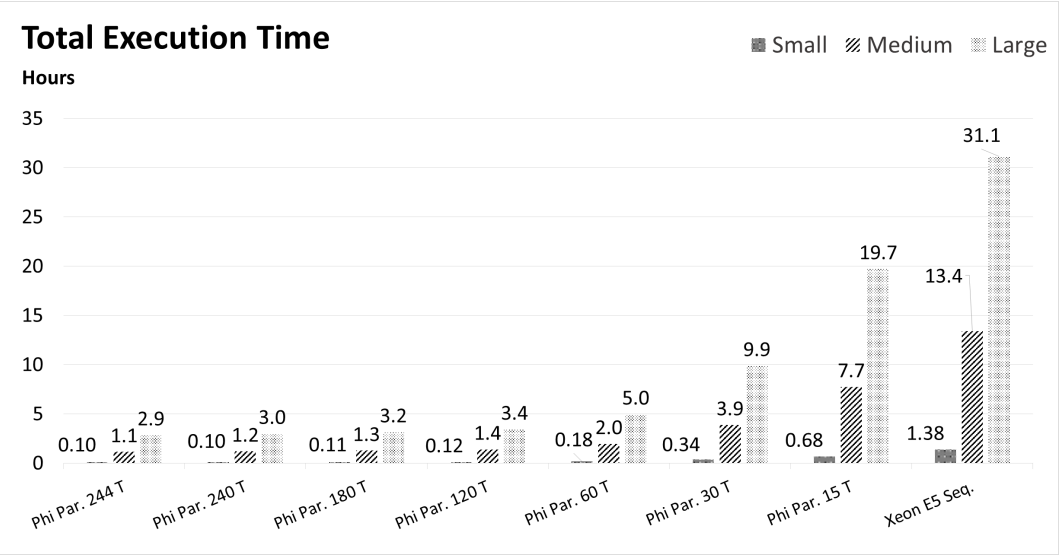

Figure 6.1.1: Total execution times for all CNN architectures on the Xeon Phi, and *Xeon E5 Seq.*

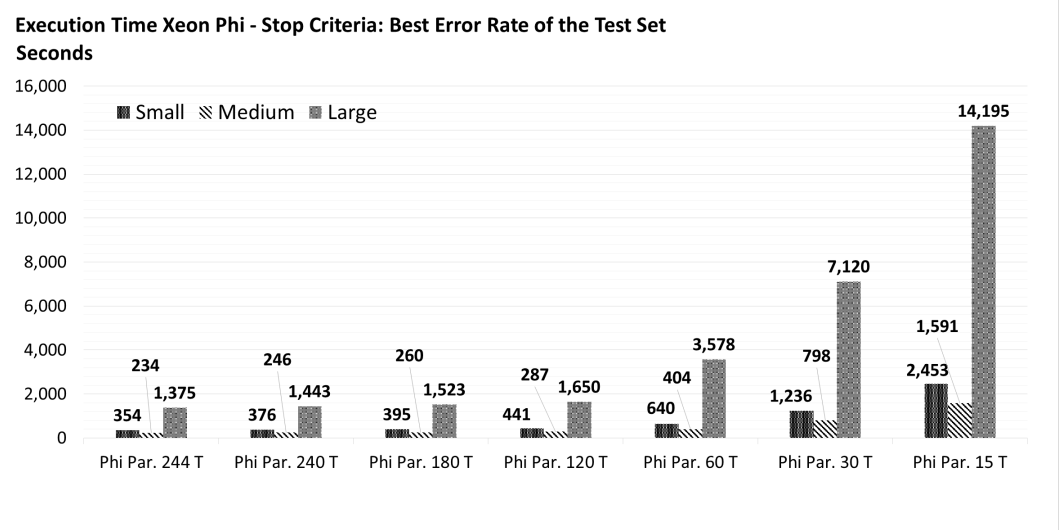

Figure 6.1.2: Execution times on the Xeon Phi for all architectures, using early stopping.

Note that several other factors impact training, including the starting decay, the factor which the decay is decreased, dataset, loss function, preparation of images, initial weight values. Therefore several combinations of parameters needs to be tested before finding a balance. In this study we focus on the number of epochs as the stop criteria and draw conclusions from this, taking into account the deviation of the error and error rates.

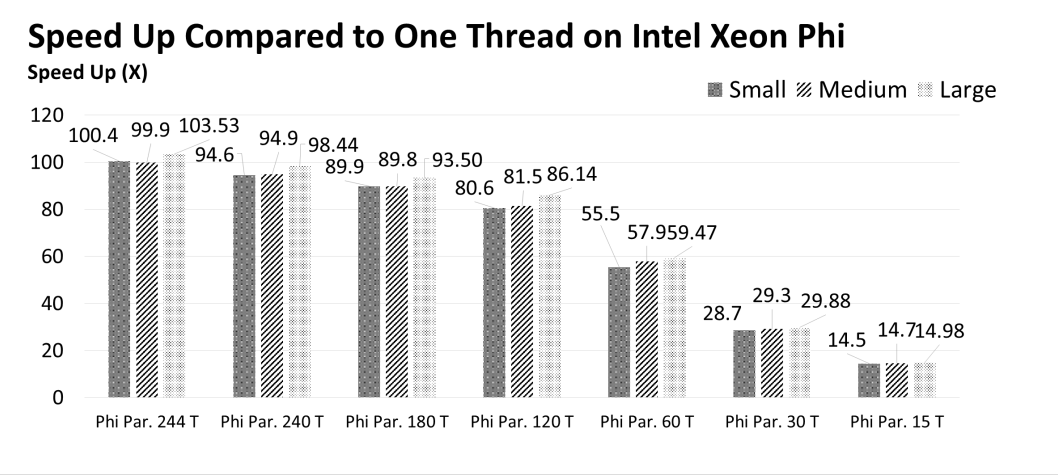

Figure 6.1.3: Speed up compared to *Phi Par. 1 T* for all architectures.

If considering the speed-up-viewpoint, a speed up (even if only by a small factor) is encountered for all CNN architectures when applying more threads on the Xeon Phi (*figure 6.1.3*). Moreover it can be seen that when keeping the number of threads fixed and increasing the architecture size, the speed up increases with a small factor as well, except for 244 threads. It seems like larger architectures are beneficial. However, it could also be the case that *Phi Par. 1 T* executes relatively slower for larger architectures than for smaller ones, which could be seen in the results as well. To illuminate this, *figure 6.1.4*, shows that the speed up compared to *Xeon E5 Seq.* decreases for larger architectures, and hence the *Xeon E5 Seq.* seem to do a better job for larger architectures than *Phi Par. 1 T* does. On the positive, more threads still increase the speed up when fixating the architecture size.

One should keep in mind that even if the speed up is rather constant for different architecture sizes, the total execution time is still decreasing by the speed up factor - only depicting the speed up can be confusing. In the end, what is interesting in most cases is the total execution time.

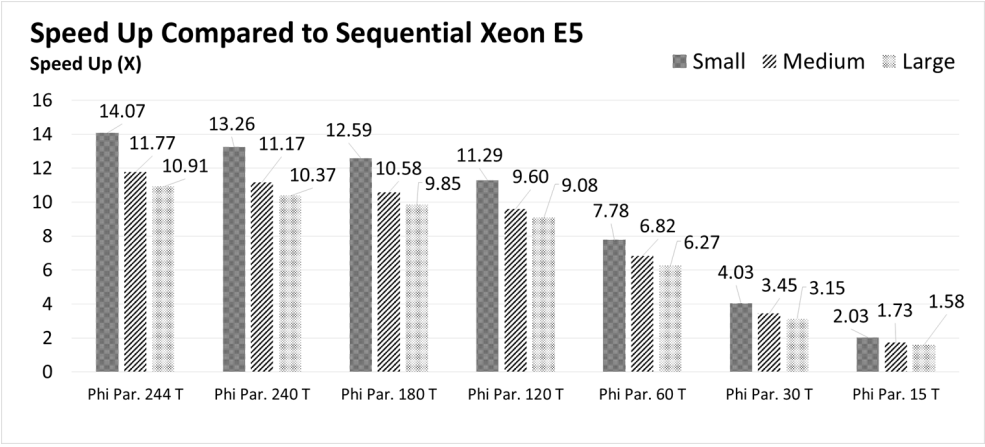

Figure 6.1.4: Speed up compared to *Xeon E5 Seq* for all CNN architectures when executed on the Xeon Phi.

6.1.2 Time Spent at Each Layer

The total execution time is highly bound to the forward- and back-propagation in the network, and as earlier concluded, the convolutional layers is the most computational

heavy ones. *Table 6.1.2* presents the speed up relative to the *Phi Par. 1 T* for the different architectures on the convolutional layer. The times are collected by each network instance (through instrumentation of the forward- and back-propagate function) and averaged over the number of network instances and epochs. As can be seen, in almost all cases there is an increase in speed up when increasing the network size, more importantly, the speed up does not decrease. Maybe the most interesting phenomena is that the speed up per layer have an almost direct relationship to the speed up of the algorithm, especially if compared to the back-propagation part. This emphasizes the importance of reducing the time spent in the convolutional layers.

	[1]BPC S	BPC M	BPC L	[2]FPC S	FPC M	FPC L
Phi Par. 244 T	102.0	99.3	103.5	122.3	124.2	125.4
Phi Par. 240 T	96.5	94.1	98.4	114.3	117.3	118.7
Phi Par. 180 T	91.8	89.5	93.9	106.3	107.0	105.8
Phi Par. 240 T	82.7	82.4	87.5	91.0	91.0	91.0
Phi Par. 60 T	56.9	58.9	59.7	58.6	60.1	60.0
Phi Par. 30 T	29.2	29.6	29.9	29.8	30.2	30.1
Phi Par. 15 T	14.7	14.8	15.0	14.9	15.1	15.0

Table 6.1.2: Averaged layer speed up compared to the *Phi Par. 1 T*.

6.1.3 Prediction Accuracy

In order to validate the implementation, the error and error rates were collected for each epoch and configuration. This section aims to analyse them in more detail. The results are presented in *chapter 4*.

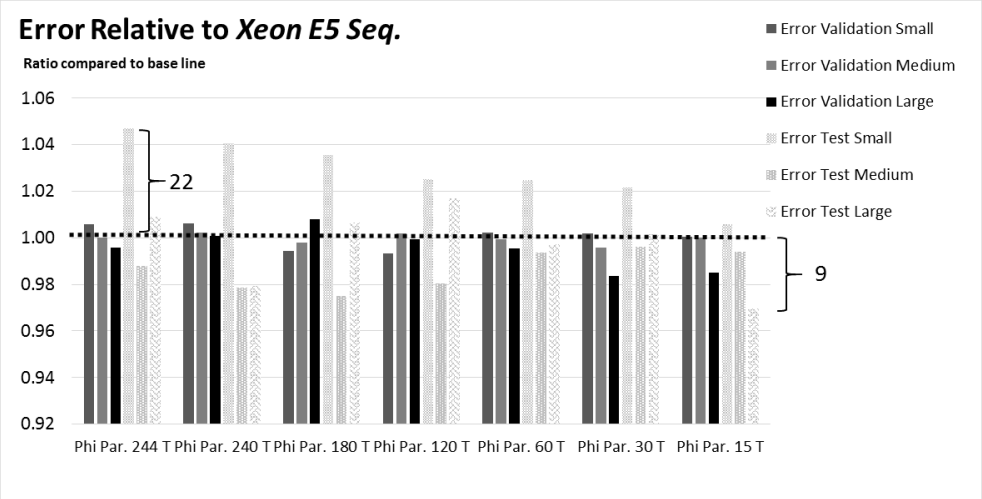

Figure 6.1.5: Relative cumulative error (loss) compared to *Xeon E5 Seq*.

In *figure 6.1.5* the ending errors for all CNN architectures and thread counts are presented, for both the validation and test set. The black dashed line emphasize the base line,

[1] *BP* is an abbreviation of back-propagation, *C* for convolutional layer. Additionally, *S, M* and *L* denote the Small, Medium and Large architecture respectively.

[2] *FP* is an abbreviation of forward propagation.

i.e. a ratio of 1. Values below the line is considered better, and values above worse than for *Xeon E5 Seq*. We use Xeon E5 as the base line, however, identical results are derived executing the sequential version on any platform. The largest difference is encountered by *Phi Par. 244 T*, about 22 units (0.05%) worse than the base line. On the contrary, *Phi Par. 15 T* has 9 units lower error compared to the base line for the large test set. It can be seen that the validation sets are rather stable whereas the test sets fluctuates more heavily. Although one should consider the deviation in error respectfully, they are not abundant in this case. The reader should be aware of that the diagram has a high zoom factor, the differences are therefore magnified.

In *table* 6.1.3 the number of incorrectly predicted images are shown for each CNN architecture. The number of images were derived by multiplying the error rates collected in the experiments with the number of images in the set and then rounded to the nearest integer. The reason for rounding is the loss of precision in the collection procedure occurring when dividing the number of incorrectly predicted images over the size of the set.

For each architecture, the total (Tot) amount of images and the difference (Diff) compared to the optimal numbers of *Xeon E5 Seq.* are shown. Negative values indicate that the ending error rate was better than optimal, less images were incorrectly predicted. Positive values indicate that more images than optimal was *not* predicted correctly. Values annotated with bold fonts are the worst values for the set and italic fonts the best. No obvious pattern can be found, however, increasing the number of threads does not lead to worse prediction in general. *Phi Par. 180 T* stands out as it was 17 images better than optimal for the small architecture on the validation set, a similar value can be found for *Phi Par. 15 T* on the large architecture on the test set. *Phi Par. 15 T* also performs worst on the small architecture on the validation set. The overall worst performance is achieved by 120 threads on the test set for the small architecture. One should consider that the total number of images in the validation set is 60,000 and 10,000 for the test set. Overall, the number of wrongly predicted images and the deviation from the base line is not abundant.

	Validation						Test					
	Small		Medium		Large		Small		Medium		Large	
	Tot	Diff	Tot	Diff	Tot	Diff	Tot	Diff	Tot	Diff	Tot	Diff
Phi Par. 244 T	616	4	85	1	12	2	155	2	98	3	**95**	**1**
Phi Par. 240 T	610	-2	86	2	11	1	154	1	*95*	*0*	91	-3
Phi Par. 180 T	*595*	*-17*	**87**	**3**	**12**	**2**	158	5	98	3	95	1
Phi Par. 120 T	607	-5	83	-1	11	1	**159**	**6**	95	0	94	0
Phi Par. 60 T	615	3	*81*	*-3*	11	1	156	3	98	3	91	-3
Phi Par. 30 T	612	0	83	-1	*10*	*0*	156	3	98	3	90	-5
Phi Par. 15 T	**617**	**5**	84	0	*10*	*0*	*153*	*0*	**100**	**5**	*84*	*-10*

Table 6.1.3: The number of incorrectly predicted images for the different CNN architectures.

It should also be considered that training could be stopped earlier or other techniques such as dropout (refer to *section 2*) could be used to lower the error rates; the test error rates seem to slightly increase after a given amount of epochs at which point training will not improve the prediction accuracy of the network any further. Therefore, if stopping at the lowest error rate, the ending error rates could be lowered. However, the main goal of this study is not to increase the performance of the prediction, instead we use the errors and error rates in order to validate the correctness of our approach.

When training the small architecture using the original implementation an ending error rate on the test set of 1.41% was achieved. To be compared to 1.55% using 244 threads on the Xeon Phi, and 1.53% for the sequential version (not randomizing weights and images). Hence, when randomizing images, a 12-14 (0.12-0.14%) images better prediction could be expected. It should be considered that the original version was trained for 65 epochs as default, not 70 as the results presented in this section. Nevertheless, a small deviation in error insure the quality of the parallelization scheme and the implementation.

6.2 Theoretical Analysis

In *chapter 5*, a theoretical analysis of the algorithm was carried out concluding the parallelism and speed up. This model did not account for hardware characteristics - in essence it evaluated the thread parallelism of the algorithm. The same model was later extended in the *section 5.4* into a performance model, in which hardware characteristics were accounted for. This model was compared to the measured values allowing for validation of the theoretical model and predictions of varying epochs, images and processing units. In addition, adjusting the number of operations/time spent in forward-and back-propagation allow for predictions of other architectures as well.

Our theoretical model had some obvious impairment, it almost stated a linear speed up except for the small fraction of sequential work required. Nevertheless, we ended up concluding the parallelism as:

$$\frac{T_1}{T_\infty} = \frac{(b' * i + c' * it + w') + (d + f' * i + g' * i + h' * it) * ep}{(b' * i + c' * it + w') + (d + f' + g' + h') * ep}$$

in which it can be seen that the parallelism is bounded by the number of images (*i* and *it*), assuming the number of training/validation images i are larger than the test set it, and fixating ep we stated the asymptotic notation as $\theta(i)$. Additionally, the number of epochs is expected to affect the execution time, however not the parallelism per se. The constant amount of work carried out in the left terms b', c', d, and w' is both in the numerator and the denominator. The constants f', g' and h' are considerable larger than the left term, and hence an increasing amount of images and epochs will diminish the left term. Although, even theoretically the speed up will not be linear as it the sequential amount of work add some overhead.

However, the theoretical model is not completely true since it excludes several hardware specific factors, including memory latencies and synchronization overhead. Therefore we created a performance model using two strategies. The model is parameterized by the number of epochs (ep), testing images (it), training images (i) and processing units (p). Also other parameters can be used, both related to the hardware and not.

In prediction *a)* we approximated the number of instructions required by each part of the algorithm (using major simplifications). We plotted a graph pretty close to the measured values in our results. In prediction *b)* we inserted measured times for the sequential execution of the different parts of the algorithm. The results were similar to the ones from prediction *a)*, again closely mimicking the measured values. We do not replicate the formulas here, please refer to the *section 5.4* for more information. Additionally predictions were made for increasing epoch counts, images and processing units. The algorithm scaled well with the number of processing units, showing promising results if more units are available in future experiments.

To conclude, the theoretical model was promising with the number of training/validation images as an upper bound of the maximum speed up of the algorithm. When

applying hardware specific parameters, the model turned out to be pretty accurate, and encouraging to facilitate the many cores of the coprocessor. The sequential work and memory contention limited the algorithm to scale linearly, however scaled well even for larger number of threads. Errors found in experiments were not abundant, in fact for some configurations the number of incorrectly predicted images were even better than the base line. Hence, even if the current parallelization scheme has its shortcomings, it indicates on speed up both in theory and in practice for the coprocessor. Moreover, a lower execution time did not decrease the prediction accuracy noticeably.

6.3 RQ1: What is the Potential of Intel Xeon Phi for Supervised Deep Learning Algorithms?

Results derived in experiments and theoretical analysis highlighted the capabilities of the Intel Xeon Phi coprocessor. Results showed an increase in speed up for increasing thread counts, at the same time lowering the execution time significantly. Using on-line stochastic gradient with back-propagation to train CNNs, supervised, was proven successful on the Xeon Phi. By exploiting thread- and SIMD-parallelism, memory alignment and other optimization techniques, the training was increased many-fold - *103.5x* compared to a single thread on the coprocessor (large architecture) and *14.07x* compared to the sequential version on the Xeon E5-2695v2 (small architecture). Training lasted for *31.1 hours* on the Xeon E5-2695v2 was lowered to *2.9 hours* on the Xeon Phi (244 threads) when training the large network for 15 epochs.

The speed up brought some deficiency in terms of fluctuation and deviation of errors and error rates, however, as shown in the analysis these were not prominent, and in some cases even better than those of the compared sequential version. The absence of image randomization, randomization of weights and presence of shared weights should also be taken into account when considering the errors. Nevertheless, a small difference in error to gain salient speed up is in our opinion an acceptable trade-off.

Additionally, the theoretical model and performance model showed promising results in terms of scaling, and if, in the future, more threads become available, it could be expected that the algorithm has an increased speed up, lowering the execution time even further.

On the contrary, optimizing algorithms to run smoothly on the Xeon Phi requires work. The same phenomena has been found by others, e.g. Jianbin Fang et al. [32]. The time spent optimizing for the Xeon Phi pays of two-fold as the algorithm becomes optimized for other Intel architectures implicitly. In order to fully utilize the many cores on the coprocessor, one needs to carefully consider the characteristics of the underlying hardware.

The scope of deep learning algorithms is wide and deep, generalizing our findings to all algorithms, architectures, datasets, and implementation would be naive. Nevertheless, there are similarities allowing for some generalization, including the back-propagation algorithm. Moreover, experiments showed promising results for larger architectures as well and larger, more complex datasets commonly require larger architectures. Perhaps the most important factor prohibiting generalization is related to the implementation, i.e. data structures and how propagation is carried out at each layer. Bad alignment of data, and loops impossible to vectorize prohibit an efficient use of the vector processing units. Additionally, markedly large datasets which will not fit in main memory are also problematic. Sparse deep learning implementations able to adopt *CHAOS* successfully are expected to work well with the coprocessor, in accordance with our findings.

As a result, *the potential of Intel Xeon Phi for supervised deep learning algorithms* is certainly *promising* - promising results were derived for CNNs and in many aspects they can be generalized to other supervised deep learning models as well.

Previous work targeting deep learning and the Intel Xeon Phi is sparse. To our best knowledge this is the first work targeting supervised deep learning algorithms and CNNs on the Intel Xeon Phi many core coprocessor.

Chapter 7

Discussion

This chapter is organized as follows. The section social impact and ethical considerations discusses the relation to the society as a whole. The validity and reliability section illuminates the actions taken to increase the quality of the work. In contributions we share the major achievements of the study and how they contribute to the community. A personal reflection on the method discusses decisions made in the approach. Some explicit limitations of the study are mentioned in its own section. The chapter ends with a conclusion and suggestions of future work.

7.1 Social Impact and Ethical Considerations

Findings of this study have several applications in the society. First, illuminating the coprocessor's ability to speed up supervised deep learning algorithms widen the scope for its usage; individuals or companies who already own a device, can make use of the many advantages of deep learning, understanding the potential of the Xeon Phi to perform the task. Secondly, individuals or companies that do not own a device, can with the help of this study understand the advantages and shortcomings in the context of deep learning algorithms, making wiser decisions whether or not to invest in a device. Moreover, the coprocessor have been shown to be very energy efficient, to reduce power consumption it could be a feasible alternative - the Xeon Phi is used in a super computer positioned in the top 20 on the GREEN list [68].

Furthermore, with the help of the developed performance model, the community can, roughly, get an understanding of what to expect in future settings and the capabilities of the parallelization scheme.

This work position the Xeon Phi as a target for research, and allows both industry and consumers to make use of it for deep learning. All data used in our experiments were publicly available and no privacy considerations were necessary.

7.2 Reliability and Validity of the Results

Reliability of the results are important to ensure testability and repeatability. Therefore, we perform several executions for each configuration and provide their average as the final results. It should also be noted that training for several epochs also removes some of the deviations as an equal amount of work is carried out by each worker in each epoch. In addition, the standard deviation is included in *Appendix D*, errors are documented in our results (*chapter 4*), and the analysis (*chapter 6*) covers the prediction accuracy. Moreover, *Appendix D* also describes the compilation and execution parameters needed to repeat the

experiments. The source code can be distributed at request, please send an email [1] to announce your interest.

The sequential and parallel versions executed on one thread were proven equal. Additionally, the original implementation of the code was validated towards our sequential version to be equal when including the randomization of weights. Therefore it is expected that the deviation originates from the lack of randomization and the non-deterministic update scheme.

It should be noticed that the sequential and the parallel version should not be compared when executing the parallel version for network/thread sizes > 1. This since the unordered sequence of updates when several networks are trained using the same weights will not be deterministic. Nevertheless, we showed in the analysis of the results (*chapter 6*) that the errors of the algorithm did not deviate markedly from the optimal values. A small deviation in error and a considerable speed up, is in our opinion an acceptable trade-off. Moreover, the transparency of all of the results collected, and the collection process, emphasize the absence of bias in the study.

The external validity of our experiments, measured by generalization is limited by the scope defined. Due to the limited scope it is not possible to prove the results to be valid for all implementations, datasets, architectures, and deep learning algorithms, nor for all supervised deep learning algorithms. We acknowledge a limited external validity of the study as we did not randomize, or investigate several datasets, CNN architectures or implementations - yet again, this was not the goal of the study, however the reader should be aware of the limitations and the quality of the work.

7.3 Contributions

The evaluation of the Intel Xeon Phi for deep learning algorithms is our main contribution in this thesis. Findings illuminate the potential of the Xeon Phi in the context of CNNs and shows significant speed up for increasing thread counts on the coprocessor. Results also indicate on increased performance for the Intel E5-2695v2 processor. We designed, implemented and evaluated **C**ontrolled **H**ogWild with **A**rbitrary **O**rder of **S**ynchronization, *CHAOS*, taking advantage of positive qualities from existing stochastic gradient descent schemes.

Additionally, a performance model was designed, and evaluated. Except from evaluating the theoretical model, and positioning the measurements of the evaluation, the performance model allows to predict varying epochs, images and thread counts. It is also possible to adapt it to changing CNN architecture sizes and hardware properties. Using the performance model we made predictions of the execution time for larger thread counts and various CNN architectures. Additionally, we made predictions for varying number of images and epochs.

We believe that the study contribute to an otherwise sparse set of work targeting deep learning and the Intel Xeon Phi, and minimize the gap of knowledge mentioned in the problem context of this study.

Previous studies, as discussed in *section 1.2*, investigated unsupervised deep learning and machine learning. We found no work targeting supervised deep learning explicitly for the Xeon Phi, nor any work targeting CNNs and the Xeon Phi. Work done by Lei Jin et al. in [30] present work for Intel Xeon Phi and *unsupervised* training of sparse auto encoders and restricted Boltzmann machines. Moreover, a library for *Support Vector*

[1] av22cj@student.lnu.se

Machine (SVM), MIC-SVM, was developed and later evaluated on the Xeon Phi by Yang You et al. [28].

To our best knowledge this is the first study investigating the potential of Intel Xeon Phi for supervised deep learning and CNNs.

7.4 Personal Reflections on the Method

In this section we aim to subjectively discuss the method, the rationale of decisions and what could have been done differently, i.e. lessons learned.

7.4.1 Personal Reflections on the Selection of Implementation

First off, the selection of an existing implementation is a core decision in order to be successful with the study. As described in *chapter 3*, the selection procedure was based on inclusion criterion, and resulted in code written by Dan Cireșan. Limitations of the host system demanded few dependencies, otherwise most of the time would have been spent solving dependencies, and otherwise get the code running on the host and the coprocessor. In future work a clean implementation, or an implementation based on a well-known library would be to prefer, e.g. *Tiny CNN* [2].

7.4.2 Personal Reflections on the Parallelization Scheme

The choice of parallelization scheme was not obvious. The main concern was not to deviate too much from the original implementation. The literature was searched for promising, and simple, schemes that would require only smaller modifications to the code. Additionally, if possible to generalize the scheme to other implementations that would be beneficial.

A couple of schemes were evaluated before developing *CHAOS*. Among them were mini-batch stochastic-gradient-descent shown to have a bad convergence, especially for larger batch sizes, similar results have been presented by other researchers [64]. Additionally, we also tried asynchronous stochastic gradient descent, averaging the weight updates from several instances. In addition, we applied model parallelism with little success, the overhead of spawning threads and inseparable loops prohibited the use of this approach. Hence we focused on the data parallel approach.

CHAOS is far from perfect, to further improve it we would like to dig deeper into the hotspots, using *VTune Amplifier* or similar analysis tool to derive a more optimized version. Without the performance metrics available we propose the following improvements to *CHAOS*:

- Data locality is essential for the coprocessor to operate smoothly. Many variables are thread private and together they require much storage, several threads sharing the same caches increase this problem eve further. One idea is to divide the work on each core model-wise letting all threads work on the same data. Another approach is to group cores to train one instance of the network and average gradients at specific points in time, similar to asynchronous stochastic gradient descent. In general, the distribution of network instances and the synchronization of weight parameters could be altered to minimize wait times.

[2] https://github.com/nyanp/tiny-cnn

- The alignment of data structures could be improved to reduce the number of memory requests and increase the amount of relevant data fetched. The current data structures are not optimized fully for the Xeon Phi even if allocated using alignment and annotated SIMD instructions are used. Additionally, higher utilization of the vector processing (VPU) unit can be achieved.

- Combining *CHAOS* with model parallelism have been discussed thoroughly in [4] by Alex Krizhevsky where data- and model-parallelism are combined using different approaches on different layers.

Moreover, other optimization techniques such as using the Math Kernel Library (MKL) was not introduced and can be added in a future update.

7.4.3 Personal Reflections on the Evaluation

The data collection procedure was carried out based on a set of epochs, where each epoch produced similar measurements in execution time. However, more executions lead to better predictability (in many cases). There is a balance of execution time and prediction accuracy which have to be weighed into the resulting execution time. Other research have shown, e.g. [40], that training for a large amount of epochs results in the best error rates. However, at some point the error does not longer decrease. On the contrary training for a fraction of epochs may yield "good enough" results. Therefore the comparison in execution time should be relative to the accepted predictability in the specific case and one should be careful when comparing execution time of CNN architectures.

In this study we do not consider all viewpoints, we focus on the total execution time. Additionally, we provide some insight in the results analysis (*chapter 6*) when training is stopped at a fixed error rate. Nevertheless, as the error rates did not deviate much from the base line, whatever architecture provides best results for the problem should be compared in the specific case. Moreover, the training should preferably stop when the error rate seem to converge, i.e. do not decrease in consecutive epochs.

7.5 Limitations of the Study

We acknowledge the scope of the study to be limited to the MNIST [25] dataset, the CNN architectures used, the parallelization scheme, and the implementation selected. Due to the limited scope, we cannot claim the results to be valid for all architectures, datasets or implementations used in conjunction with supervised deep learning algorithms. Careful implementation allowing all common optimization functions, activation functions, architecture types, datasets, etc. would allow for a wider scope; however, this was not our goal of the study.

We assume the scalability, and speed up of the algorithm to be reasonably good even for larger CNN architectures and datasets since we recognized a speed up for an increasing number of threads for large architectures. The limiting factor may be the memory size of the coprocessor. The input and architecture size affects the number of calculations not their characteristics, although more variables may lead to worse data locality. Moreover, we recognize no restrictions in using other activation functions or loss functions. Nevertheless, the scope should be restricted to on-line stochastic gradient descent and the back-propagation algorithm applied to sparse deep learning models. As *CHAOS* divide the work on a high level, it should be applicable to other deep learning algorithms that have the possibility to divide the work by the input domain.

The performance model should not be considered generic for all platforms, its intended scope is the algorithm used in this study and the Intel Xeon Phi, codenamed Knights Corner.

7.6 Conclusion

Based on an existing implementation of a CNN we developed a highly parallel algorithm optimized for the Intel Xeon Phi. Results show a *103.5x, 99.9x* and *100.4x* speed up (large, medium, small) for 244 threads compared to one thread on the Xeon Phi and *14.07x* (small) compared to the sequential version executed on the host, Xeon E5-2695v2. Training the large network for 15 epochs on the coprocessor requires about *3 hours*, a sequential execution on the host, Xeon E5-2695v2 requires *31 hours* and on the Core i5 661 it takes *a week*. Additionally, one thread on the Xeon Phi takes *12 days*. The parallel version showed only a smaller deviation in error rates compared to the sequential version.

Our contributions also include *CHAOS*, a parallelization scheme using thread- and SIMD-parallelism to facilitate the training, and a performance model, allowing for evaluation of the theoretical claims and future predictions. Based on the results derived from experiments and theoretical deduction we answered the research question with the pleasing answer: *promising* - the potential of the Intel Xeon Phi for supervised deep learning algorithms is promising, and theoretically the algorithm should scale well even beyond the cores of the coprocessor. Results pave the way for future research in the area. The Intel Xeon Phi position itself as a consumer-load-balancer to be used in conjunction with CPUs, providing off-the-shelf, high computational power.

7.7 Future Work

Except from an improved version of *CHAOS*, mentioned in *section 7.4.2* we also promote future work to focus on larger architectures and datasets, e.g. GoogleNet and ImageNet to further analyse the Xeon Phi. Also, a dynamic implementation, presumably a fork of an existing well known implementation should be created and optimized to be used with the coprocessor, facilitating multiple nodes as well. With such scalable model, the evaluation could cover a larger extent of the scope - larger architectures, different models and other datasets. The implementation should not suffer from too many third-party libraries, to ease the execution on the coprocessor. Additionally, an offload version of *CHAOS* (or similar) would make use of both the host CPU and the coprocessor and facilitate the use of several coprocessors as well. A promising library called HyPhi [36] can be used to enable this. Another promising project not yet finished, RaPyDLI [69] aims to create a library addressing many of the concerns mentioned above. Furthermore, future work should target the second generation of Intel Xeon Phi, codenamed Knight Landing.

Bibliography

[1] D. C. Ciresan, U. Meier, J. Masci, and J. Schmidhuber, "A committee of neural networks for traffic sign classification." in *Proceedings of the IEEE*, 2011, pp. 1918–1921.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, no. 11, 1998, pp. 2278–2324.

[3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions." *CoRR*, vol. abs/1409.4842, 2014.

[4] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks." *CoRR*, vol. abs/1404.5997, 2014.

[5] Y. Tang and I. Sutskever, "Data normalization in the learning of restricted boltzmann machines," Department of Computer Science, University of Toronto, Technical Report UTML-TR-11-2, 2011.

[6] A. Ng, "Coursera - free online courses from top universities," 2015. [Online]. Available: https://www.coursera.org/learn/machine-learning/home/info

[7] L. Deng and D. Yu, "Deep learning: Methods and applications." Microsoft, Tech. Rep., 2014.

[8] "Convolutional neural networks (lenet) — deeplearning 0.1 documentation," 2015. [Online]. Available: http://deeplearning.net/tutorial/lenet.html

[9] J. Schmidhuber, "Deep learning in neural networks: An overview." *Neural Networks*, vol. 61, pp. 85–117, 2015.

[10] "Training an artificial neural network - intro," 2015. [Online]. Available: http://www.solver.com/training-artificial-neural-network-intro

[11] "Backpropagation algorithm - ufldl," 2015. [Online]. Available: http://deeplearning.stanford.edu/wiki/index.php/Backpropagation_Algorithm

[12] J. DAINTITH, "performance model – dictionary definition of performance model | encyclopedia.com: Free online dictionary," 2015. [Online]. Available: http://www.encyclopedia.com/doc/1O11-performancemodel.html

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[14] "Intel® xeon phi™ product family: Product brief," 2015. [Online]. Available: http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html

[15] R. McMillan, "Siri will soon understand you a whole lot better | wired," 2015. [Online]. Available: http://www.wired.com/2014/06/siri_ai/

[16] K. Wu, X. Chen, and M. Ding, "Deep learning based classification of focal liver lesions with contrast-enhanced ultrasound," *Optik - International Journal for Light and Electron Optics*, vol. 125, no. 15, pp. 4057–4063, 2014.

[17] N. T. Adithya Rao, "Recognition of facial expressions with autoencoders and convolutional-nets." Master's thesis, Université de Montréal, 2014.

[18] "Self-driving car test: Steve mahan," 2012. [Online]. Available: https://www.youtube.com/watch?v=cdgQpa1pUUE

[19] C. Shu, "Google acquires artificial intelligence startup deepmind for more than $500m," 2015. [Online]. Available: http://techcrunch.com/2014/01/26/google-deepmind/

[20] "Imagenet," 2015. [Online]. Available: http://image-net.org/

[21] J. Novet, "Microsoft researchers say their newest deep learning system beats humans — and google," 2015. [Online]. Available: http://venturebeat.com/2015/02/09/microsoft-researchers-say-their-newest-deep-learning-system-beats-humans-and-google

[22] J. H. Huang, "Leaps in visual computing," 2015.

[23] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "High-performance neural networks for visual object classification," *CoRR*, vol. abs/1102.0183, 2011.

[24] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato, "Multi-gpu training of convnets." *CoRR*, vol. abs/1312.5853, 2013.

[25] C. C. Yann LeCun and C. Burges, "Mnist handwritten digit database," 2015. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[26] L. C. Yann, "Norb object recognition dataset," 2015. [Online]. Available: http://www.cs.nyu.edu/~ylclab/data/norb-v1.0/

[27] A. Krizhevsky, "Cifar-10 and cifar-100 datasets," 2015. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[28] Y. You, S. L. Song, H. Fu, A. Marquez, M. M. Dehnavi, K. J. Barker, K. W. Cameron, A. P. Randles, and G. Yang, "Mic-svm: Designing a highly efficient support vector machine for advanced modern multi-core and many-core architectures." in *IPDPS*. IEEE, 2014, pp. 809–818.

[29] C.-C. Chang and C.-J. Chung, "Libsvm – a library for support vector machines," 2015. [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/libsvm/

[30] L. Jin, Z. Wang, R. Gu, C. Yuan, and Y. Huang, "Training large scale deep neural networks on the intel xeon phi many-core coprocessor." in *IPDPS Workshops*. IEEE, 2014, pp. 1622–1630.

[31] K. Ahmed, Q. Qiu, P. Malani, and M. Tamhankar, "Accelerating pattern matching in neuromorphic text recognition system using intel xeon phi coprocessor." in *IJCNN*. IEEE, 2014, pp. 4272–4279.

[32] J. Fang, H. J. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving intel xeon phi." in *ICPE*. ACM, 2014, pp. 137–148.

[33] S. Memeti and S. Pllana, "Accelerating dna sequencing using intel xeon phi," Linnaeus University, Department of Computer Science, Tech. Rep., 2015.

[34] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh, "Optimizing the mapreduce framework on intel xeon phi coprocessor," *CoRR*, vol. abs/1309.0215, 2013.

[35] G. Teodoro, T. M. Kurç, J. Kong, L. A. D. Cooper, and J. H. Saltz, "Comparative performance analysis of intel xeon phi, gpu, and cpu: A case study from microscopy image analysis." in *IPDPS*. IEEE, 2014, pp. 1063–1072.

[36] J. Dokulil, E. Bajrovic, S. Benkner, M. Sandrieser, and B. Bachmayer, "Hyphi - task based hybrid execution c++ library for the intel xeon phi coprocessor." in *ICPP*. IEEE Computer Society, 2013, pp. 280–289.

[37] J. Dokulil and S. Benkner, "Automatic tuning of a parallel pattern library for heterogeneous systems with intel xeon phi." in *ISPA*. IEEE, 2014, pp. 42–49.

[38] K.-C. Leung, D. M. Eyers, X. Tang, S. Mills, and Z. H. 0001, "Investigating large-scale feature matching using the intel® xeon phi™ coprocessor." in *IVCNZ*. IEEE, 2013, pp. 148–153.

[39] J. Vrtanoski and T. D. Stojanovski, "Pattern recognition with opencl heterogeneous platform." in *Telecommunications Forum (TELFOR), 2012 20th*. IEEE, 2012, p. 701.

[40] D. C. Ciresan, U. Meier, J. Masci, and J. Schmidhuber, "Multi-column deep neural network for traffic sign classification." *Neural Networks*, vol. 32, pp. 333–338, 2012.

[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks." in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 1097–1105.

[42] I. Song, H.-J. Kim, and P. Jeon, "Deep learning for real-time robust facial expression recognition on a smartphone," in *Consumer Electronics (ICCE), 2014 IEEE International Conference on*, Jan 2014, pp. 564–567.

[43] D. Scherer, H. Schulz, and S. Behnke, "Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors." in *ICANN (3)*, vol. 6354. Springer, 2010, pp. 82–91.

[44] T. N. Sainath, B. Kingsbury, G. Saon, H. Soltau, A. rahman Mohamed, G. E. Dahl, and B. Ramabhadran, "Deep convolutional neural networks for large-scale speech tasks." *Neural Networks*, vol. 64, pp. 39–48, 2015.

[45] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, Oct. 2006.

[46] N. R. Council, *The Rise of Games and High-performance Computing for Modeling and Simulation.* National Academies Press, 2010.

[47] "Intel® xeon phi™ coprocessor - the architecture | intel® developer zone," 2012. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

[48] J. Rendeirs, "An overview of programming for intel xeon processors and intel xeon phi coprocessors," 2015.

[49] "Optimization and performance tuning for intel® xeon phi™ coprocessors, part 2: Understanding and using hardware events | intel® developer zone," 2012. [Online]. Available: https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding

[50] "Ark | intel® xeon® processor e5-2695 v2 (30m cache, 2.40 ghz)," 2015. [Online]. Available: http://ark.intel.com/products/75281/Intel-Xeon-Processor-E5-2695-v2-30M-Cache-2_40-GHz

[51] T. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[52] "Neural networks - ufldl," 2015. [Online]. Available: http://deeplearning.stanford.edu/wiki/index.php/Neural_Networks

[53] A. Gibiansky, "Fully connected neural network algorithms - andrew gibiansky," 2015. [Online]. Available: http://andrew.gibiansky.com/blog/machine-learning/fully-connected-neural-networks/

[54] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[55] A. Gibiansky, "Convolutional neural networks - andrew gibiansky," 2015. [Online]. Available: http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/

[56] "Openmp application program interface," 2015.

[57] "Cilk plus tutorial," 2012. [Online]. Available: https://www.cilkplus.org/cilk-plus-tutorial

[58] "Intel® threading building blocks (intel® tbb) user guide," 2015. [Online]. Available: https://software.intel.com/en-us/node/506045

[59] "Optimization and performance tuning for intel® xeon phi™ coprocessors - part 1: Optimization essentials | intel® developer zone," 2012. [Online]. Available: https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization

[60] "Step by step performance optimization with intel® c++ compiler," 2014. [Online]. Available: https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler

[61] "Intel® math kernel library (intel® mkl) | intel® developer zone," 2015. [Online]. Available: https://software.intel.com/en-us/intel-mkl

[62] "Details about intel® integrated performance primitives (intel® ipp) | intel® developer zone," 2015. [Online]. Available: https://software.intel.com/en-us/intel-ipp/details

[63] B. Recht, C. Re, S. J. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent." in *NIPS*, 2011, pp. 693–701.

[64] M. D. F. D. Grazia, I. Stoianov, and M. Zorzi, "Parallelization of deep networks." in *ESANN*, 2012.

[65] Z. You and B. Xu, "Improving training time of deep neural networks with asynchronous averaged stochastic gradient descent," in *Chinese Spoken Language Processing (ISCSLP), 2014 9th International Symposium on*, Sept 2014, pp. 446–449.

[66] M. Zinkevich, A. J. Smola, and J. Langford, "Slow learners are fast." in *NIPS*. Curran Associates, Inc., 2009, pp. 2331–2339.

[67] R. R. Howell, "On asymptotic notation with multiple variables," Dept. of Computing and Information Sciences, Tech. Rep., 2008.

[68] "The green500 list - november 2014 | the green500," 2015. [Online]. Available: http://www.green500.org/lists/green201411&green500from=1&green500to=100

[69] "Rapydli - high performance open source deep learning," 2015. [Online]. Available: http://salsaproj.indiana.edu/RaPyDLI/index.html

Appendix A

Platforms

This chapter discusses the details of the platforms used in evaluation.

Xeon E5 @ Host

- CPU: Intel Xeon CPU E5-2695v2 @ 2.40GHz with 12 physical cores, using hyper-threading resulting in 48 logical cores in total. Intel Smart Cache [1]: 30 MB.
- OS: Linux 2.6.32431.el6.x86_64
- Memory: 132 GB

Core i5 661 @ Desktop

- CPU: Intel Core i5 661 3.33 GHz, 2 physical cores, 4 logical. Intel Smart Cache: 4 MB.
- OS: Windows 7 64bit
- Memory: 4GB

Intel Xeon Phi

- Model: 7120P
- Speed: 1.238 GHz / core
- OS: Linux 2.6.38.8, MPSS 3.1.1
- Memory: 16 GB
- Memory bandwidth: 352 GB/s
- L1 cache: 32 KB data, 32 KB instructions
- L2 cache: 0.5 MB /core, 30.5 MB in total

The Host and coprocessor was accessed using *ssh* and the Desktop using Remote Desktop.

[1] The Intel Smart Cache technology from Intel allow cores to share high level cache dynamically, i.e. each core has no dedicated cache space[50].

Appendix B

Details of CNN Architectures

The small network used at training is shown in *figure B.0.1*. The input layer has 841 neurons in a 29x29 grid. The input of images is 28x28 using a stride of 1 pixel. The first convolutional layer has 5 maps, 3380 neurons, uses a kernel size of 4x4, a map size of 26x26 and 85 weights. The first max-pooling layer consists of five maps, 845 neurons, a 2x2 kernel, 13x13 map size. The second convolutional layer consists of 10 maps, 810 neurons, 5x5 kernel size, 9x9 map size and 1260 weights. The last max-pooling layer has 10 maps, 90 neurons, 3x3 kernel and 3x3 map size. The fully connected layer has 50 neurons and 4,550 weights. Ending with the output layer with 10 neurons and 510 weights.

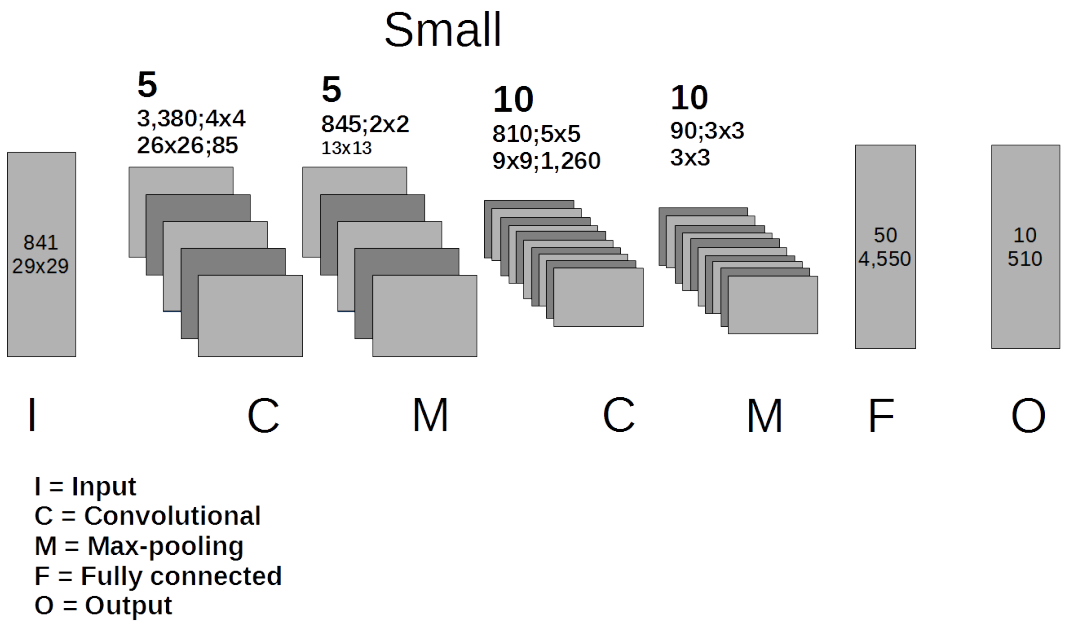

Figure B.0.1: Details of the small CNN architecture.

The medium network used at training can be seen in *figure B.0.2*. The input layer has 841 neurons in a 29x29 grid. The first convolutional layer has 20 maps, 13,520 neurons, uses a kernel size of 4x4, a map size of 26x26 and 340 weights. The first max-pooling layer consists of 20 maps, 3,380 neurons, a 2x2 kernel, 13x13 map size. The second convolutional layer consists of 40 maps, 3,240 neurons, 5x5 kernel size, 9x9 map size and 20,040 weights. The last pooling layer has 40 maps, 360 neurons, 3x3 kernel and 3x3 map size. The fully connected layer has 150 neurons and 54,150 weights. Ending with the output layer with 10 neurons and 1,510 weights.

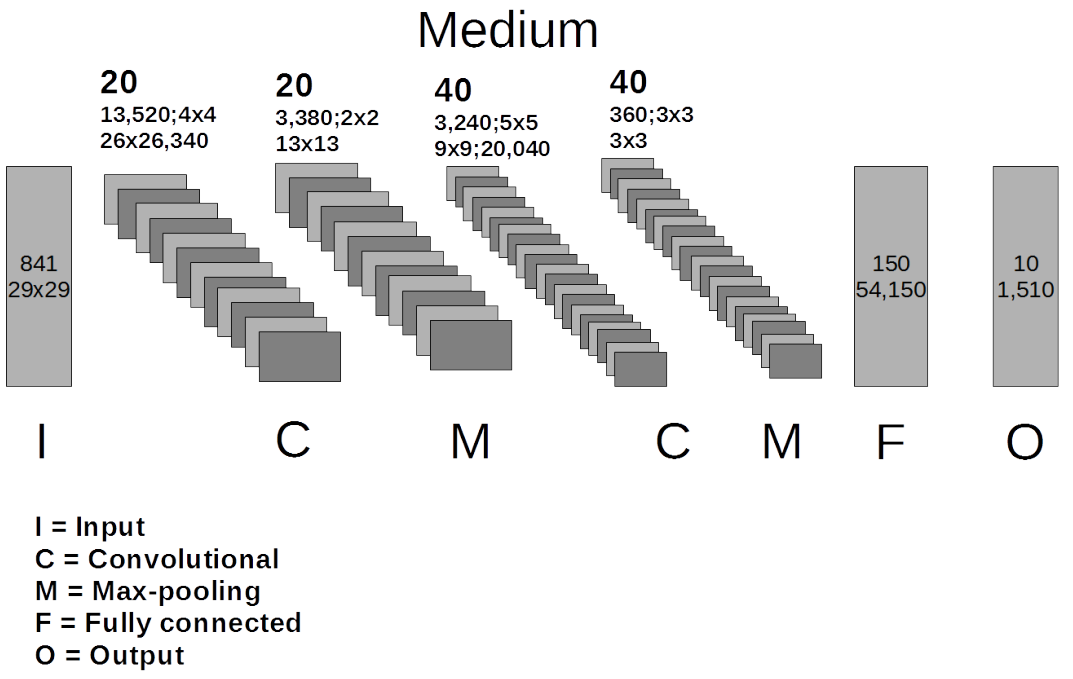

Figure B.0.2: Details of the medium CNN architecture.

The large network used at training in *figure B.0.3*. The input layer has 841 neurons in a 29x29 grid. The first convolutional layer has 20 maps, 13,520 neurons, uses a kernel size of 4x4, a map size of 26x26 and 340 weights. The first max-pooling layer consists of 20 maps, 13,520 neurons, a 1x1 kernel, 26x26 map size. The second convolutional layer consists of 60 maps, 29,040 neurons, 5x5 kernel size, 22x22 map size and 30,060 weights. The middle max-pooling layer has 60 maps, 7,260 neurons, 2x2 kernel and 11x11 map size. The last convolutional layer has 100 maps, 3,600 neurons, a 6x6 kernel, a map size of 6x6 and 216,100 weights. The last pooling layer has 100 maps, 900 neurons, 2x2 kernel size and 3x3 map size. The fully connected layer has 150 neurons and 135,150 weights. Ending with the output layer with 10 neurons and 1,510 weights.

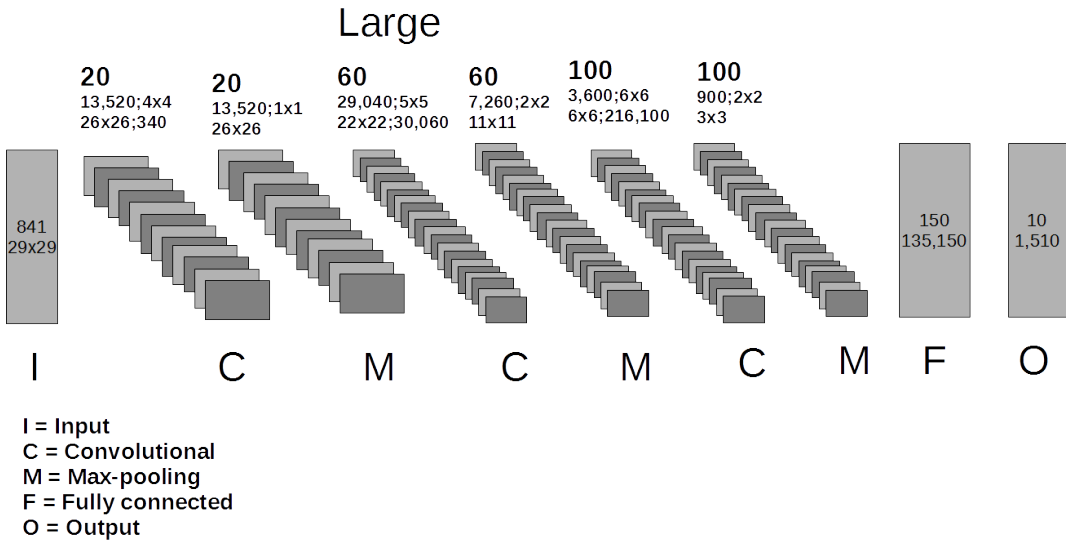

Figure B.0.3: Details of the large CNN architecture.

All maps in consecutive layers are fully connected, no image processing was used, no

pooling at convolutional layers, and *tanh* was used as activation function.

Appendix C

Vectorization Reports

This section includes vectorization reports for the convolutional layer and the back-propagation of the fully connected layer, superfluous information has been removed, only inner vectorized loops are retained.

```
1  LOOP BEGIN at neuralnetwork.cpp(504,21)
2          remark #15399: vectorization support: unroll factor set to 2
3          remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
4          remark #15451: unmasked unaligned unit stride stores: 1
5          remark #15458: masked indexed (or gather) loads: 1
6          remark #15475: --- begin vector loop cost summary ---
7          remark #15476: scalar loop cost: 160
8          remark #15477: vector loop cost: 70.750
9          remark #15478: estimated potential speedup: 2.250
10         remark #15479: lightweight vector operations: 47
11         remark #15480: medium-overhead vector operations: 2
12         remark #15482: vectorized math library calls: 1
13         remark #15488: --- end vector loop cost summary ---
14 LOOP END
```

Listing C.1: Vectorization report for forward propagation in the convolutional layer.

Listings C.1 shows the forward propagation in the convolutional layer. The report relates to the part calculating the input of a neuron by iterating connected neurons in the previous layer. As can be seen the unroll factor is set to 2 and unaligned stores are issued, the estimated speed up is 2.250.

```
1  LOOP BEGIN at neuralnetwork.cpp(630,5)
2     LOOP BEGIN at neuralnetwork.cpp(634,9)
3        remark #15399: vectorization support: unroll factor set to 4
4        remark #15300: LOOP WAS VECTORIZED
5        remark #15442: entire loop may be executed in remainder
6        remark #15448: unmasked aligned unit stride loads: 2
7        remark #15449: unmasked aligned unit stride stores: 1
8        remark #15475: --- begin vector loop cost summary ---
9        remark #15476: scalar loop cost: 15
10       remark #15477: vector loop cost: 16.000
11       remark #15478: estimated potential speedup: 3.570
12       remark #15479: lightweight vector operations: 6
13       remark #15480: medium-overhead vector operations: 1
14       remark #15488: --- end vector loop cost summary ---
15    LOOP END
16 LOOP END
```

Listing C.2: Vectorization report for back-propagation in the fully connected layer.

Listings C.2 shows the back-propagation in the fully connected layer. The weight gradient calculation is vectorized. According to the vectorization report it encounters a potential speed up of 3.570.

```
1  LOOP BEGIN at neuralnetwork.cpp(683,9)
2    LOOP BEGIN at neuralnetwork.cpp(694,25)
3       remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
4       remark #15460: masked strided loads: 2
5       remark #15462: unmasked indexed (or gather) loads: 1
6       remark #15475: --- begin vector loop cost summary ---
7       remark #15476: scalar loop cost: 30
8       remark #15477: vector loop cost: 7.500
9       remark #15478: estimated potential speedup: 3.980
10      remark #15479: lightweight vector operations: 6
11      remark #15480: medium-overhead vector operations: 1
12      remark #15481: heavy-overhead vector operations: 1
13      remark #15488: --- end vector loop cost summary ---
14   LOOP END
15 LOOP END
16
17 LOOP BEGIN at neuralnetwork.cpp(706,9)
18    remark #15399: vectorization support: unroll factor set to 4
19    remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
20    remark #15448: unmasked aligned unit stride loads: 3
21    remark #15449: unmasked aligned unit stride stores: 1
22    remark #15475: --- begin vector loop cost summary ---
23    remark #15476: scalar loop cost: 22
24    remark #15477: vector loop cost: 32.000
25    remark #15478: estimated potential speedup: 2.680
26    remark #15479: lightweight vector operations: 12
27    remark #15480: medium-overhead vector operations: 1
28    remark #15488: --- end vector loop cost summary ---
29 LOOP END
30
31  LOOP BEGIN at neuralnetwork.cpp(728,21)
32    remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
33    remark #15460: masked strided loads: 2
34    remark #15462: unmasked indexed (or gather) loads: 1
35    remark #15475: --- begin vector loop cost summary ---
36    remark #15476: scalar loop cost: 26
37    remark #15477: vector loop cost: 7.500
38    remark #15478: estimated potential speedup: 3.450
39    remark #15479: lightweight vector operations: 6
40    remark #15480: medium-overhead vector operations: 1
41    remark #15481: heavy-overhead vector operations: 1
42    remark #15488: --- end vector loop cost summary ---
43 LOOP END
```

Listing C.3: Vectorization report for back-propagation in the convolutional layer.

Listings C.3 shows the vectorization report for the back-propagation of the convolutional layer. The report has three sections, one for each loop. The loop beginning at row *1* is responsible for back-propagating partial derivatives. The compiler performs a good job and manage to speed it up by almost 4 times. The loop beginning in row *17* is simple, it iterates the collected derivatives and multiplies them with the *tanh* function. Finally, the loop beginning at row *31* updates the weight parameters over the current kernel, estimating a speed up of 3.450.

Appendix D

Execution and Validation Details

This chapter includes detailed execution information including the standard deviation of the results and errors, compilation and execution options, as well as execution reports.

D.1 Compilation and Execution of the Application

To ease compilation and execution, scripts were created with different parameters. For the coprocessor the *mmic* parameter was added and all dependent libraries were copied manually. To prepare environment variables a *.profile* script was added on the coprocessor. *LOCALWEIGHTS* was added as a compiler directive to imply that local weights should be used to delay the updates. All compilation was done using the Intel compiler *icc* 15.0.0.

The parameters used as input to the application (in order):

1. Training type - currently only support 0 (CHAOS).
2. CNN Architecture (0: Small, 1: Medium, 2: Large).
3. Number of network instances ≥ 1 && ≤ 244.
4. Number of threads ≥ 1 && ≤ 244. Preferably equal to number of network instances.

Compilation command for *sequential* version: *icc main.ccp neuralnetwork.cpp helper.cpp trainer.cpp reporter.cpp idx.cpp -o network.out*

Compilation command for *parallel* version: *icc main.ccp neuralnetwork.cpp helper.cpp trainer.cpp reporter.cpp idx.cpp -o network.out -O3 -openmp -DLOCALWEIGHTS*

To generate code for the coprocessor the following was added: *-mmic -I/usr/include*.

D.2 Standard Deviation of the Results

For error and error rates the standard deviation of each epoch were calculated over the executions and averaged. The results are presented in table *D.2.1, D.2.2 and D.2.3*. Some results are incomplete due to less than two executions were performed for that configuration. Overall, it can be seen that very small deviations were encountered. The most interesting results are the execution times for *Core i5* showing that the execution time differs with a large factor between executions. To reduce the deviation more executions should be carried out and hence derive a better average. Nevertheless, the deviation did not affect the final results abundantly.

The standard deviation was calculated over the set of values in each execution, e.g. test error for an epoch, and averaged. All values were calculated using Excel, and it should be considered that a deviation is presented even if all values are identical. For the executions performed on one thread, the deviation is small, however not equal between executions, although results showed identical results over all executions. Moreover, in general the standard deviation is not abundant and is not expected to affect the results negatively. Nevertheless, we do not change the output of Excel, the values to pay most attention to is the ones with higher values.

	Error Rate		Error		
	Test	Validation	Test	Validation	Tot. Exec. Time
Xeon E5 Seq.	$3.72 * 10^{-19}$	$3.72 * 10^{-19}$	$1.30 * 10^{-14}$	$1.95 * 10^{-14}$	6.76
Core i5 Seq.	$3.66 * 10^{-19}$	$3.66 * 10^{-19}$	$1.30 * 10^{-14}$	$1.95 * 10^{-14}$	$1.66 * 10^{2}$
Phi Par. 244 T	$1.93 * 10^{-4}$	$9.93 * 10^{-5}$	1.86	3.65	$1.02 * 10^{1}$
Phi Par. 240 T	$1.83 * 10^{-4}$	$6.62 * 10^{-5}$	1.10	4.15	$2.27 * 10^{-1}$
Phi Par. 180 T	$1.76 * 10^{-4}$	$7.36 * 10^{-5}$	$6.65 * 10^{-1}$	4.40	9.41
Phi Par. 120 T	$1.90 * 10^{-4}$	$1.26 * 10^{-4}$	1.98	3.80	$3.13 * 10^{-1}$
Phi Par. 60 T	$2.54 * 10^{-4}$	$8.48 * 10^{-5}$	1.13	3.75	$1.89 * 10^{-1}$
Phi Par. 30 T	$2.61 * 10^{-4}$	$1.10 * 10^{-4}$	2.15	6.63	$5.48 * 10^{-1}$
Phi Par. 15 T	$2.17 * 10^{-4}$	$4.88 * 10^{-5}$	2.11	5.97	2.96
Phi Par. 1 T	$3.66 * 10^{-19}$	$3.66 * 10^{-19}$	$1.28 * 10^{-14}$	$1.92 * 10^{-14}$	$2.86 * 10^{1}$
Xeon E5 Par. 48 T	$2.52 * 10^{-4}$	$1.08 * 10^{-4}$	2.50	7.83	$1.55 * 10^{1}$
Xeon E5 Par. 24 T	$2.66 * 10^{-4}$	$8.11 * 10^{-5}$	1.55	5.55	8.38
Xeon E5 Par. 12 T	$2.31 * 10^{-4}$	$7.98 * 10^{-5}$	1.44	5.85	2.69
Xeon E5 Par. 1 T	$3.72 * 10^{-19}$	$3.72 * 10^{-19}$	$1.30 * 10^{-14}$	$1.95 * 10^{-14}$	7.12

Table D.2.1: Average standard deviation for the error, error rates and execution time for the small CNN architecture.

	Error Rate		Error		
	Test	Validation	Test	Validation	Tot. Exec. Time
Xeon E5 Seq.	$9.91 * 10^{-20}$	$7.43 * 10^{-20}$	0	$4.87 * 10^{-15}$	$1.57 * 10^{1}$
Core i5 Seq.	$1.08 * 10^{-19}$	$8.13 * 10^{-20}$	0	$5.33 * 10^{-15}$	$3.49 * 10^{3}$
Phi Par. 244 T	$1.49 * 10^{-4}$	$4.02 * 10^{-5}$	1.94	1.10	$2.65 * 10^{-1}$
Phi Par. 240 T	$2.07 * 10^{-4}$	$3.38 * 10^{-5}$	$5.99 * 10^{-1}$	2.49	1.68
Phi Par. 180 T	$9.35 * 10^{-5}$	$3.86 * 10^{-5}$	1.43	1.38	$2.96 * 10^{-1}$
Phi Par. 120 T	$1.32 * 10^{-4}$	$4.88 * 10^{-5}$	2.72	2.06	$8.54 * 10^{-1}$
Phi Par. 60 T	$2.63 * 10^{-4}$	$5.46 * 10^{-5}$	1.26	$9.55 * 10^{-1}$	$1.99 * 10^{-1}$
Phi Par. 30 T	$1.45 * 10^{-4}$	$2.63 * 10^{-5}$	1.52	2.10	9.80
Phi Par. 15 T	$1.58 * 10^{-4}$	$4.64 * 10^{-5}$	2.21	2.64	$2.41 * 10^{1}$
Phi Par. 1 T	X	X	X	X	X
Xeon E5 Par. 48 T	$2.02 * 10^{-4}$	$5.70 * 10^{-5}$	1.83	3.58	$1.82 * 10^{1}$
Xeon E5 Par. 24 T	$2.17 * 10^{-4}$	$3.80 * 10^{-5}$	1.50	1.93	5.11
Xeon E5 Par. 12 T	$1.78 * 10^{-4}$	$3.12 * 10^{-5}$	1.73	3.34	$1.58 * 10^{1}$
Xeon E5 Par. 1 T	$9.91 * 10^{-20}$	$7.43 * 10^{-20}$	0	$4.87 * 10^{-15}$	$9.07 * 10^{-1}$

Table D.2.2: Average standard deviation for the error, error rates and execution time for the medium CNN architecture.

	Error Rate		Error		
	Test	Validation	Test	Validation	Tot. Exec. Time
Xeon E5 Seq.	$1.16 * 10^{-19}$	$2.33 * 10^{-19}$	$1.14 * 10^{-14}$	$1.14 * 10^{-14}$	5.40
Core i5 Seq.	0	0	0	0	$3.73 * 10^{4}$
Phi Par. 244 T	$1.65 * 10^{-4}$	$7.55 * 10^{-5}$	2.98	$1.21 * 10^{1}$	$5.10 * 10^{3}$
Phi Par. 240 T	$1.86 * 10^{-4}$	$7.20 * 10^{-5}$	2.85	9.95	7.34
Phi Par. 180 T	$2.56 * 10^{-4}$	$8.30 * 10^{-5}$	2.78	8.19	7.99
Phi Par. 120 T	$2.00 * 10^{-4}$	$6.43 * 10^{-5}$	4.44	9.60	$4.42 * 10^{-1}$
Phi Par. 60 T	$1.61 * 10^{-4}$	$1.37 * 10^{-4}$	4.11	$1.85 * 10^{1}$	5.31
Phi Par. 30 T	$9.67 * 10^{-5}$	$4.72 * 10^{-5}$	3.95	7.63	$6.41 * 10^{1}$
Phi Par. 15 T	X	X	X	X	X
Phi Par. 1 T	X	X	X	X	X
Xeon E5 Par. 48 T	$2.26 * 10^{-4}$	$8.31 * 10^{-5}$	4.23	$1.22 * 10^{1}$	$1.29 * 10^{1}$
Xeon E5 Par. 24 T	$1.70 * 10^{-4}$	$4.96 * 10^{-5}$	2.20	5.27	$1.42 * 10^{1}$
Xeon E5 Par. 12 T	$1.68 * 10^{-4}$	0	2.10	5.18	4.36
Xeon E5 Par. 1 T	0	0	0	0	$8.23 * 10^{-1}$

Table D.2.3: Average standard deviation for the error, error rates and execution time for the large CNN architecture.

D.3 Execution Reports for the Experiments

In this section we present the executions as performed in experiments, most of the configurations were carried out 3 times (with some deviations due to time restrictions) and the average was used to derive the final results. The small and medium networks were trained for 70 epochs and the large for 15.

Version	#Threads	Platform	#Executions
Sequential	1	Host	3
Sequential	1	Desktop	3
Parallel	244	Phi	3
Parallel	240	Phi	3
Parallel	180	Phi	3
Parallel	120	Phi	3
Parallel	60	Phi	3
Parallel	30	Phi	3
Parallel	15	Phi	3
Parallel	1	Phi	3
Parallel	48	Host	3
Parallel	24	Host	3
Parallel	12	Host	3
Parallel	1	Host	3

Table D.3.1: Execution report for the small architecture.

Version	#Threads	Platform	#Executions
Sequential	1	Host	3
Sequential	1	Desktop	3
Parallel	244	Phi	3
Parallel	240	Phi	3
Parallel	180	Phi	3
Parallel	120	Phi	3
Parallel	60	Phi	3
Parallel	30	Phi	3
Parallel	15	Phi	3
Parallel	1	Phi	1
Parallel	48	Host	3
Parallel	24	Host	3
Parallel	12	Host	3
Parallel	1	Host	3

Table D.3.2: Execution report for the medium architecture.

Version	#Threads	Platform	#Executions
Sequential	1	Host	3
Sequential	1	Desktop	2
Parallel	244	Phi	3
Parallel	240	Phi	3
Parallel	180	Phi	3
Parallel	120	Phi	3
Parallel	60	Phi	3
Parallel	30	Phi	2
Parallel	15	Phi	1
Parallel	1	Phi	1
Parallel	48	Host	3
Parallel	24	Host	3
Parallel	12	Host	3
Parallel	1	Host	3

Table D.3.3: Execution report for the large architecture.

Appendix E

Additional Theoretical Analysis

This chapter discusses the running time of the complex functions related to the forward- and back-propagation, and initialization, excluded in the *chapter 5*. In *table E.0.1* variables used in the formulas are enumerated.

Variable	Meaning
arch	CNN Architecture (small, medium, large)
l	Number of layers
labels	Number of labels
nll	Number of neurons last layer
npl	Number of neurons previous layer (idx)
ntl	Number of neurons this layer (idx)
nnl	Number of neurons next layer (idx)
msx	Map size x this layer (idx)
msy	Map size y this layer (idx)
msyn	Map size y next layer (idx)
msxn	Map size x next layer (idx)
ksx	Kernel size x this layer (idx)
ksy	Kernel size y this layer (idx)
ksxn	Kernel size x next layer (idx)
ksyn	Kernel size y next layer (idx)
nwl	Number of weights this layer (idx)
nwnl	Number of weights next layer (idx)
mtl	Number of maps this layer (idx)
mpl	Number of maps previous layer (idx)
mnl	Number of maps next layer (idx)

Table E.0.1: Abbreviations used in the theoretical analysis.

E.1 Forward Propagate Function for the Max Pooling Layers

Forward propagation in the max pooling layer calculates the maximum value inside each kernel spawning neurons in the previous layer. The goal is to aggregate the maximum values held by neurons in the previous layer to increase the spatiality and reduce the number of neurons required in later layers.

```
forwardPropagateMaxPoolingLayer()

mtl+1       for thisLayerMap = 0 to numberOfMapsThisLayer-1
mtl(msy+1)      for neuronY = 0 to numberOfNeuronsThisLayerY-1
mtl*msy(msx+1)    for neuronX = 0 to numberOfNeuronsThisLayerX-1
mtl*msy*msx           Let max be smallest double value
mtl*msy*msx(ksy+1)    for kernelY = 0 to kernelSizeY-1
mtl*msy*msx*ksy(ksx+1)    for kernelX = 0 to kernelSizeX-1
mtl*msy*msx*ksy*ksx         Let idx be the index of the neuron at prev.layer through kernel
mtl*msy*msx*ksy*ksx         if max < outputsPreviousLayer[idx]
mtl*msy*msx*ksy*ksx           max = outputsPreviousLayer[idx]
mtl*msy*msx*ksy*ksx           idx_max = idx

mtl*msy*msx           Let idx be index of neuron: (neuronX, neuronY) in thisLayerMap
mtl*msy*msx           outputsThisLayer[idx] = max
mtl*msy*msx           maxPositionsThisLayer[idx] = idx_max
```

Listing E.1: Pseudocode for forward propagation in the max-pooling layer.

$$
\begin{aligned}
&T_{ForwardPropagateMaxPoolingLayer}(mtl, msx, msy, ksx, ksy) \\
&= c_3 + (c_3 + c_4)mtl + (c_4 + c_5)mtl * msy + (c_5 + c_6 + c_7 + c_{14} \\
&+ c_{15} + c_{16})mtl * msy * msx + (c_7 + c_8)mtl * msy * msx * ksy \\
&+ (c_8 + c_9 + c_{10} + c_{11} + c_{12})mtl * msy * msx * ksy * ksx \\
&= a * mtl + b * (mtl * msy) + c * (mtl * msy * msx) \\
&+ d * (mtl * msy * msx * ksy) + e * (mtl * msy * msx * ksy * ksx) + f
\end{aligned}
$$

E.2 Forward Propagate Function for the Fully Connected Layers

The fully connected layer either uses the *tanh* or the *soft max* activation function. Therefore, the calculations are separated. The output is retrieved by passing the input through the activation function.

```
forwardPropagateFullyConnectedLayer()

1   if activationTypeLayer == tanh
1       idx = 0
ntl+1   for neuronThisLayer = 0 to numberOfNeuronsThisLayer-1
ntl         Initialize sum to bias
ntl*npl     for c = 1 to numberOfNeuronsPreviousLayer-1
ntl*(npl-1)     Increase sum by multiplying weight and output of related neuron c at
      previous layer

ntl         Set output of neuron neuronThisLayer to tanh(sum)
ntl         Move weight index
// End for

1   else if activationTypeLayer == soft_max
1       Let a be the input of this layer
ntl+1   for i = 0 to a.Length-1
ntl       a[i] = 0

// Calculate inputs
ntl+1   for neuronThisLayer = 0 to numberOfNeuronsThisLayer-1
ntl         Initialize a[neuronThisLayer] to bias
ntl*npl     for c = 1 to numberOfNeuronsPreviousLayer-1
ntl*(npl-1)       Add output * weight of connected neuron to a[neuronThisLayer]

// Calculate outputs
1       cut = 700
1       denominator = 0
ntl+1   for neuronThisLayer = 0 to numberOfNeuronsThisLayer-1
ntl       if a[neuronThisLayer] > cut
ntl         a[neuronThisLayer] = cut
ntl       else if a[neuronThisLayer] <- cut
ntl         a[neuronThisLayer] = -cut
```

```
34 ntl       denominator = denominator + eExponent(a[neuronThisLayer])
35 // End for
36
37 ntl+1   for neuronThisLayer = 0 to numberOfNeuronsThisLayer-1
38 ntl       yThisLayer[neuronThisLayer] = eExponent(a[neuronThisLayer]) / denominator
```

Listing E.2: Pseudocode for forward propagation in the fully connected layer.

$$
\begin{aligned}
&T_{ForwardPropagateFullyConnectedLayerTanh}(ntl, npl) = c_3 + c_4 + c_5 + c_{14} \\
&+ (c_5 + c_6 + c_{10} + c_{11})ntl + (c_7 + c_8)ntl * npl \\
&= a * ntl + b * ntl * npl + c \\
&T_{ForwardPropagateFullyConnectedLayerSoftMax}(ntl, npl) = \\
&c_3 + c_{14} + c_{15} + c_{16} + c_{20} + c_{26} + c_{27} + c_{28} + c_{37} \\
&+ (c_{16} + c_{17} + c_{18} + c_{20} + c_{21} - c_{23} + c_{28} + c_{29} + c_{30} + c_{31} + c_{32} + c_{37} + c_{38})ntl \\
&+ (c_{22} + c_{23})ntl * npl = a * ntl + b * ntl * npl + c
\end{aligned}
$$

E.3 Forward Propagate Function for the Convolutional Layers

The convolutional layer calculates the output of each neuron as *tanh* of the input. The input is calculated over all connected maps and kernels for each neuron in the current map and layer.

```
1  forwardPropagateConvolutionalLayer()
2
3  1   if activationTypeThisLayer == scaled_tanh
4  mtl+1     for thisLayerMap = 0 to numberOfMapsThisLayer-1
5  mtl           Let numberOfMapsConnectedPreviousLayer be the number of connected maps in
       previous layer
6  mtl(msy+1)      for neuronY = 0 to numberOfNeuronsThisLayerY-1
7  mtl*msy(msx+1)      for neuronX = 0 to numberOfNeuronsThisLayerX-1
8  mtl*msy*msx           Let sum equal bias
9  mtl*msy*msx*(mpl+1)     for previousLayerMap = 0 to numberOfMapsConnectedPreviousLayer-1
10 mtl*msy*msx*mpl*(ksy+1)    for kernelY to kernelSizeY-1
11 mtl*msy*msx*mpl*ksy(ksx+1)    for kernelX to kernelSizeX-1
12 mtl*msy*msx*mpl*ksy*ksx         Add to sum, the output * weight of the connected neuron
       through the kernel
13
14 mtl*msy*msx           Let output of neuron (neuronX, neuronY) equal tanh(sum)
15 mtl           Move weights pointer to next connected map
```

Listing E.3: Pseudocode for forward propagation in the convolutional layer.

$$
\begin{aligned}
&T_{FPropConvLayer}(mtl, mpl, msy, msx, ksx, ksy) \\
&= c_3 + c_4 + (c_4 + c_5 + c_6 + c_{15})mtl + (c_6 + c_7)mtl * msy \\
&+ (c_7 + c_8 + c_9 + c_{14})mtl * msy * msx + (c_9 + c_{10})mtl * msy * msx * mpl \\
&+ (c_{10} + c_{11})mtl * msy * msx * mpl * ksy \\
&+ (c_{11} + c_{12})mtl * msy * msx * mpl * ksy * ksx \\
&= a * mtl + b * (mtl * msy) + c * (mtl * msy * msx) \\
&+ d * (mtl * msy * msx * mpl) + e * (mtl * msy * msx * mpl * ksy) \\
&+ f * (mtl * msy * msx * mpl * ksy * ksx) + g
\end{aligned}
$$

E.4 Back-propagate Function for the Max Pooling Layers

Backpropagation in the max pooling layer simply pushes deltas from this layer to the previous one through the max filter as determined in the forward propagation.

```
1   backpropagateMaxPoolingLayer()
2
3   1         if idx > 0 // Ignore input layer
4   1          Set previous layer deltas to 0
5
6   nnl+1     for n = 0 to numberOfNeuronsThisLayer-1
7   nnl        deltaPreviousLayer[max_pos[n]] = deltaThisLayer[n]
```

Listing E.4: Pseudocode for back-propagation in the max-pooling layer.

$$T_{BackwardPropagationMaxPoolingLayer}(ntl) = \\ c_3 + c_4 + c_6 + (c_6 + c_7) * nnl = a * nnl + b$$

E.5 Back-propagate Function for the Fully Connected Layers

The propagation of values at the fully connected layer is similar to the one of the convolutional layer. First deltas are pulled from the next layer by multiplying them with connecting weight values and decay. Then weights are updated: each weight parameter is decreased by the output of the neuron at the current layer multiplied with the delta of the connected neuron at the next layer. This is performed for each combination of weights connected neurons in the layers.

```
1   backpropagateFullyConnectedLayer()
2
3   // Propagate deltas to this layer
4   1     if idxLayer > 0 // Ignore input layer
5   ntl+1     for neuronThisLayer = 0 to numberOfNeuronsThisLayer-1
6   ntl           deltaThisNeuron = 0
7   ntl           weightIdx = neuronThisLayer + 1
8   ntl(nnl+1)     for neuronNextLayer = 0 to numberOfNeuronsNextLayer-1
9   ntl*nnl         Add delta of neuron in nextLayer * weight to deltaThisNeuron
10  ntl*nnl         Move weightIdx
11
12  ntl         Multiply deltaThisNeuron with tanh(outputThisNeuron)
13
14  // Update weights of next layer by iterating neuron pairs
15  1     Set Weights pointer LocalWeights to delay updates
16  1     Set LocalWeights to 0 at all positions
17
18  nnl+1   for neuronNextLayer = 0 to numberOfNeuronsNextLayer-1
19  nnl         Let eta_nextDelta be eta * deltaNeuronNextLayer
20  nnl         Pre-compute weightIdx
21  nnl         Decrease weight with bias
22  nnl(ntl+1)    for neuronThisLayer = 0 to numberOfNeuronsThisLayer-1
23  nnl*ntl         Subtract eta_nextDelta * outputThisNeuron from localWeights[weightIdx]
24
25
26  // Update with delay
27  nwnl+1   for w = 0 to numberOfWeightsNextLayer-1
28             #atomic
29  nwnl       weightsNextLayer[w] = weightsNextLayer[w] + localWeightsNextLayer[w]
```

Listing E.5: Pseudocode for back-propagation in the fully connected layer.

$$T_{BackpropagateFullyConnectedLayer}(nnl, ntl, nwnl) \\ = c_4 + c_5 + c_{15} + c_{16} + c_{18} + c_{27} + (c_5 + c_6 + c_7 + c_8 + c_{12})ntl \\ + (c_8 + c_9 + c_{10} + c_{22} + c_{23})ntl * nnl \\ + (c_{18} + c_{19} + c_{20} + c_{21} + c_{22})nnl + c_{27} * nwnl + c_{28} * nwn \\ = a * ntl + b * (ntl * nnl) + c * nnl + d * nwnl + e$$

E.6 Back-propagate Function for the Convolutional Layers

First deltas are propagated to the previous layer based on the deltas on the current layer. Thereafter, the weights are calculated and the gradients are added to the local weights. For each neuron in the current map, all weights that are touched by this neuron is updated accordingly over all connected maps. After the local calculations of weights are done, the global weights are updated with the calculated values.

```
1  backpropagateConvolutionalLayer()
2
3  // Propagate deltas to previous layer
4  1   if idx > 0 // Ignore input layer
5  1     Set deltas on previous layer to 0
6  mnl+1       for mapthisLayer = 0 to numberOfMapsThisLayer-1
7  mnl               Let numberOfMapsConnectedPreviousLayer be the number of connected maps in
       the previous layer
8  mnl*(msyn+1)       for neuronY = 0 to numberOfNeuronsThisMapY-1
9  mnl*msyn*(msxn+1)     for neuronX = 0 to numberOfNeuronsThisMapX-1
10 mnl*msyn*msxn             Let delta be the delta of neuron (neuronX, neuronY)
11 mnl*msyn*msxn*(mpl+1)     for mapPreviousLayer = 0 to numberOfMapsConnectedPreviousLayer-1
12 mnl*msyn*msxn*mtl             Pre-calculate indices
13 mnl*msyn*msxn*mtl*(ksyn+1)     for kernelY = 0 to kernelSizeY
14 mnl*msyn*msxn*mtl*ksyn*(ksxn+1)       for kernelX = 0 to kernelSizeX
15 mnl*msyn*msxn*mtl*ksyn*ksxn               Add delta * Weights[(neuronX,neuronY), (kernelX,
       kernelY)] to deltaPreviousLayer[(kernelX, kernelY)]
16
17
18 mnl               Move pointer to weights for next connected maps
19 // End outer for
20
21 ntl+1for i = 0 to numberOfNeuronsPreviousLayer-1
22 ntl     previousLayerDelta[i] = previousLayerDelta[i] * tanh(previousLayerOutput[i])
23
24 // Update weights
25 1   Set Weights pointer to LocalWeights for delaying updates
26 1   Set all LocalWeights to 0
27
28 mnl+1for mapthisLayer = 0 to numberOfMapsThisLayer-1
29 mnl      Let numberOfMapsConnectedPreviousLayer be the number of connected maps in the
       previous layer
30 mnl*(msyn+1)     for neuronY = 0    to numberOfNeuronsThisMapY-1
31 mnl*msyn*(msxn+1)     for neuronX = 0 to numberOfNeuronsThisMapX-1
32 mnl*msyn*msxn*(mnl+1)       for mapPreviousLayer = 0 to numberOfMapsConnectedPreviousLayer-1
33 mnl*msyn*msxn*mnl               Pre-calculate indices
34 mnl*msyn*msxn*mtl*(ksyn+1)       for kernelY = 0 to kernelSizeY-1
35 mnl*msyn*msxn*mtl*ksyn*(ksxn+1)       for kernelX = 0 to kernelSizeX-1
36 mnl*msyn*msxn*mtl*ksyn*ksxn               Decrease the gradient with the delta of neuron (
       neuronX,neuronY) * the output of the connected neuron (kernelX,kernelY) over the
       kernel
37
38 mnl      Move weights pointer to next map
39 // End outer for
40
41 // Update weights with a delay from values in LocalWeights
42 nwl+1     for w = 0 to numberOfWeightsThisLayer-1
43             #atomic
44 nwl         WeightsThisLayer[w] = WeightsThisLayer[w] + LocalWeightsThisLayer[w]
```

Listing E.6: Pseudocode for back-propagation in the convolutional layer.

$$
\begin{aligned}
&T_{BackpropagateConvolutionalLayerD}(mnl, mtl, msyn, msxn, ksyn, ksxn) \\
&= c_4 + c_5 + c_6 + (c_6 + c_7 + c_8)mnl + (c_8 + c_9)mnl * msyn \\
&+ (c_9 + c_{10} + c_{11})mnl * msyn * msxn \\
&+ (c_{11} + c_{12} + c_{13})mnl * msyn * msxn * mtl \\
&+ (c_{13} + c_{14})mnl * msyn * msxn * mtl * ksyn + (c_{14} \\
&+ c_{15})mnl * msyn * msxn * mtl * ksyn * ksxn \\
&= a * mnl + b * (mnl * msyn) + c * (mnl * msyn * msxn) \\
&+ d * (mnl * msyn * msxn * mtl) + e * (mnl * msyn * msxn * mtl * ksyn) \\
&+ f * (mnl * msyn * msxn * mtl * ksyn * ksxn) + g
\end{aligned}
$$

$$
\begin{aligned}
&T_{BackpropagateConvolutionalLayerW}(mnl, mtl, msyn, msxn, ksyn, ksxn, nwnl) \\
&= c_{25} + c_{26} + c_{28} + c_{42} + (c_{28} + c_{29} + c_{30} + c_{38})mnl \\
&+ (c_{30} + c_{31})mnl * msyn + (c_{31} + c_{32})mnl * msyn * msxn \\
&+ (c_{32} + c_{33} + c_{34})mnl * msyn * msxn * mtl \\
&+ (c_{34} + c_{35})mnl * msyn * msxn * mtl * ksyn \\
&+ (c_{35} + c_{36})mnl * msyn * msxn * mtl * ksyn * ksxn + (c_{42} + c_{44})nwnl \\
&= a * mnl + b * (mnl * msyn) + c * (mnl * msyn * msxn) \\
&+ d * (mnl * msyn * msxn * mtl) + e * (mnl * msyn * msxn * mtl * ksyn) \\
&+ f * (mnl * msyn * msxn * mtl * ksyn * ksxn) + g * nwnl + h
\end{aligned}
$$

$$
\begin{aligned}
&T_{BackpropagateConvolutionalLayer} \\
&= T_{BackpropagateConvolutionalLayerDelta} \\
&+ T_{BackpropagateConvolutionalLayerWeights}
\end{aligned}
$$

E.7 Determine Error For Chunk Function

The *DetermineErrorForChunk* function calculates the error for a chunk of images. In the current implementation it is used with a single image.

```
1   determineErrorForChunk()
2
3   1    currentERR = 0
4   1    Set pointers for image and label
5   1    Point image to input layer
6
7   l    for i = 1 to numberOfLayers-1
8   (l-1)    ForwardPropagate(i)
9
10  1    Set image to NULL
11  1    Set desired output to y
12  1    digit = 0
13
14  nll  for i = 1 to numberOfNeuronsLastLayer-1
15  (nll-1)    if y[i] > y[digit]
16  (nll-1)      digit = i
17
18  (nll+1)    for i = 0 to numberOfNeuronsLastLayer-1
19  nll          if lastLayerActivationType == soft_max
20  nll            Set y[i] bound to DOUBLE_MIN
21  nll          Update currentERR
22
23  nll          else if lastLayerActivationType == scaled_tanh
24  nll            Update currentERR
25
26  1      totalERR = totalERR + currentERR
27  1      return 1 if wrong prediction, 0 if corrects
```

Listing E.7: Pseudocode for the determineErrorForChunk function.

$$
\begin{aligned}
&T_{DetermineErrorForChunk} = c_3 + c_4 + c_5 - T_{FPropOneLayer}(arch) + c_{10} + c_{11} \\
&+ c_{12} - c_{15} - c_{16} + c_{18} + c_{26} + c_{27} + T_{FPropOneLayer}(arch) * l \\
&+ (c_{14} + c_{15} + c_{16} + c_{18} + c_{19} + c_{20} + c_{21} + c_{23} + c_{24})nll \\
&= T_{FPropOneLayer}(arch) * l - T_{FPropOneLayer}(arch) + b * nll + c
\end{aligned}
$$

E.8 Other Functions

This chapter cover other non-trivial operations included in the calculations. Each complex function is approximated and summarized.

$$
\begin{aligned}
&T_{PreAllocateDesiredOutput}(arch) = a * nll^2 = \mathcal{O}(nll^2) \\
&T_{PreAllocateImages}(i) = a * i * ss = \mathcal{O}(i)
\end{aligned}
$$

The number of images *i* will always grow faster than the sample size *ss* as the sample size is fixed.

$$
T_{InitiateImages}(i) = a + 3i + i * labels + log(labels) + labels = \mathcal{O}(i)
$$

The number of labels are constant, in our case *10*, and hence we can omit them in the asymptotic notation.

$$
\begin{aligned}
&T_{ReadNetworkWeights}(nwl) = \mathcal{O}(nwl) \\
&T_{SetNetworkWeights}(l) = \mathcal{O}(l) \\
&T_{CreateNetwork}(l) = \mathcal{O}(T_{CreateLayer} * l)
\end{aligned}
$$

where $T_{CreateLayer}$ is one of: $T_{CreateInputLayer}$, $T_{CreateFullyConnectedLayer}$, $T_{CreateConvolutionalLayer}$ or $T_{CreateMaxPoolingLayer}$

$$
\begin{aligned}
&T_{CreateInputLayer} = \mathcal{O}(ksx * ksy) \\
&T_{CreateMaxPoolingLayer} = \mathcal{O}(1) \\
&T_{CreateFullyConnectedLayer} = \mathcal{O}(nwl) \\
&T_{CreateConvolutionalLayer} = \mathcal{O}(mtl * mpl)
\end{aligned}
\tag{E.1}
$$

Appendix F

Additional Results

In this chapter we provide additional results related to the Xeon E5 and Core i5 excluded in the chapter *Results*.

F.1 Results for Intel Xeon E5

This section aims to describe data collected for the Xeon E5 including execution time, speed up and error. In *figure F.1.1* it can be seen that *Xeon E5 Par.* lower the execution time for 12 and 24 threads, however when further increasing the count to 48 threads it takes slightly longer time. The training of the large network takes *1 hour and 45 minutes* for 24 threads on the Xeon E5 and *about 31 hours* for the sequential version.

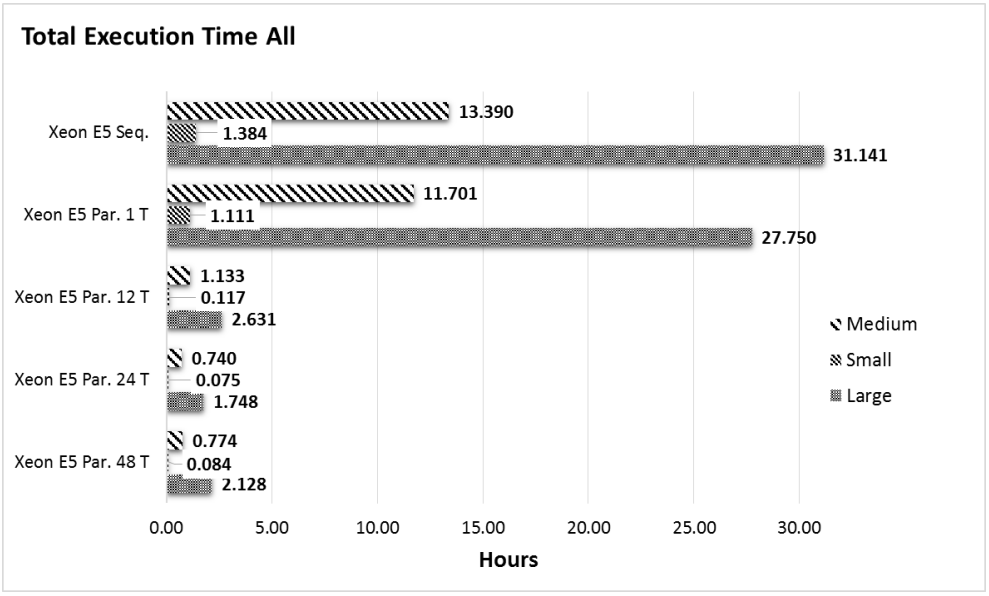

Figure F.1.1: The total execution time for all CNN architectures on the Xeon E5, for various thread counts.

In *figure F.1.2* the speed up for Xeon E5 compared to *Xeon E5 Seq.* is shown. The speed up is increasing from 12 to 24 threads for all CNN architectures. From 24 to 48 threads the speed up only increases for the medium architecture, for the large and small, it decreases. Except for the medium architecture on *Xeon E5 Par 48 T*, smaller architectures seem to infer a better speed up than larger ones. When executing the experiments again for 48 threads, the same values were encountered, however, execution times fluctuates highly for all architectures, which is not the case for 24 and 12 threads on the Xeon E5.

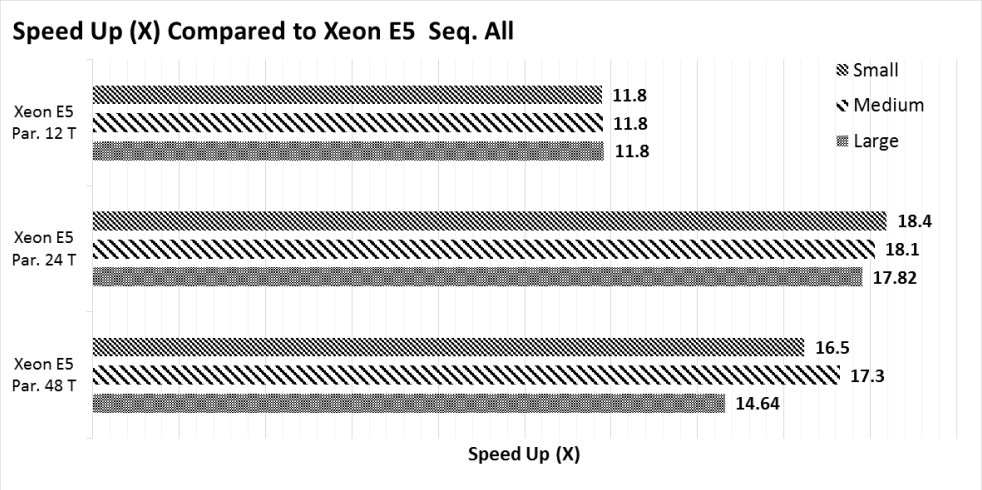

Figure F.1.2: Speed up compared to *Xeon E5 Seq.* for all architectures and thread counts on the Xeon E5.

Optimizing the code for the Xeon Phi also optimize it for the Xeon CPU as it share similar properties and the compilation is similar (just ignoring the *mmic* switch). The ending error rates are shown for the Xeon E5 in table F.1.1.

	Error						Error Rate					
	Validation			Test			Validation			Test		
	Small	Medium	Large	Small	Medium	Large	Small	Medium	Large	Small	Medium	Large
48 T	2085	512	183	477	293	295	1.03%	0.14%	0.02%	1.57%	0.99%	0.92%
24 T	2093	510	181	475	292	285	1.02%	0.14%	0.02%	1.58%	0.99%	0.91%
12 T	2090	506	180	471	297	290	1.01%	0.13%	0.02%	1.51%	1.01%	0.90%
1 T	2092	509	183	468	296	290	1.02%	0.14%	0.02%	1.53%	0.95%	0.94%
Seq.	2092	509	183	468	296	290	1.02%	0.14%	0.02%	1.53%	0.95%	0.94%

Table F.1.1: The ending error and error rates for the Xeon E5 on all CNN architectures compared to the base line *Seq*.

F.2 Speed Up compared to Intel Core i5

Figures F.2.1, F.2.2, and F.2.3 show the *speed up* compared to *Core i5*. It can be seen that the same pattern is found as for *Xeon E5 Seq.* however, the values are a fraction higher.

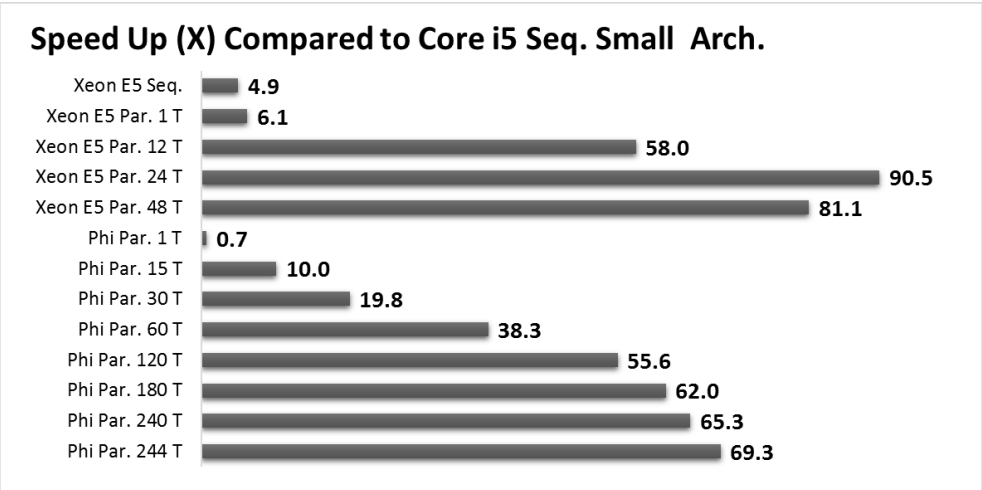

Figure F.2.1: Speed up compared to *Core i5 Seq.* for the small architecture.

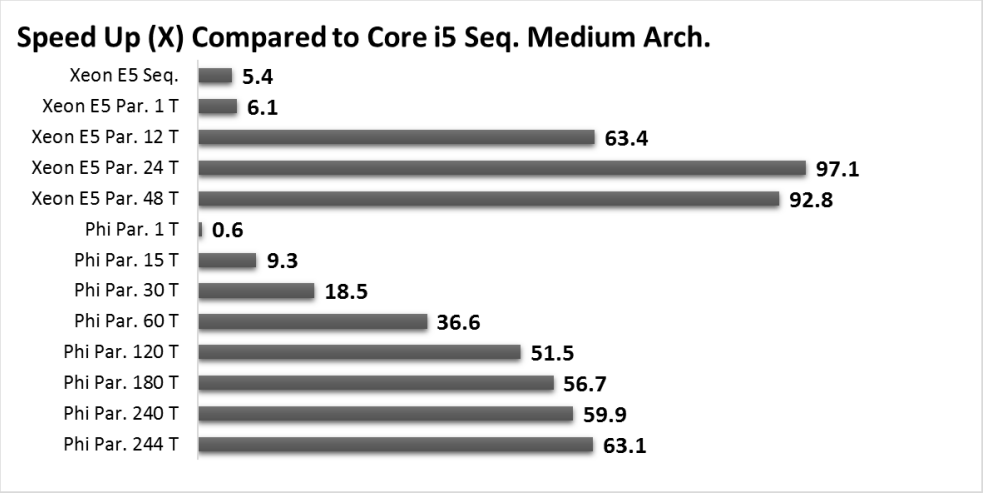

Figure F.2.2: Speed up compared to *Core i5 Seq.* for the medium architecture.

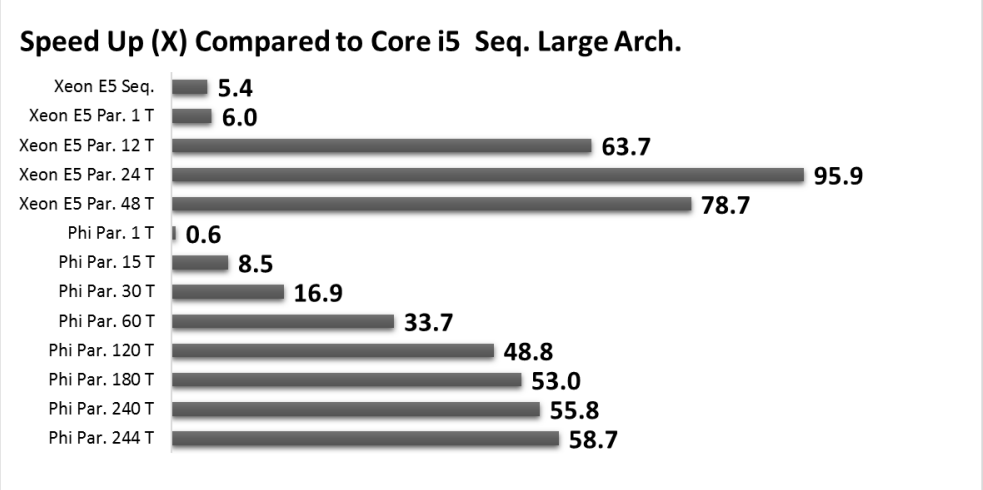

Figure F.2.3: Speed up compared to *Core i5 Seq.* for the large architecture.

Lnu.se

Linnæus University
Sweden

Faculty of Technology
SE-391 82 Kalmar | SE-351 95 Växjö
Phone +46 (0)772-28 80 00
teknik@lnu.se
Lnu.se/faculty-of-technology?l=en