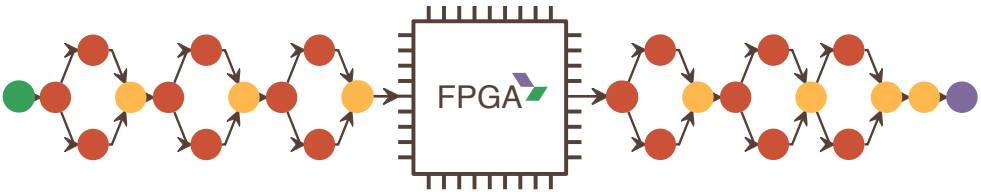
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SCS
super computing systems

Master Thesis Report

ZynqNet:

An FPGA-Accelerated Embedded Convolutional Neural Network



David Gschwend
davidgs@student.ethz.ch

Supervisors: Emanuel Schmid
Felix Eberli

Professor: Prof. Dr. Anton Gunzinger

August 2016, ETH Zürich,

Department of Information Technology and Electrical Engineering

Abstract

Image Understanding is becoming a vital feature in ever more applications ranging from medical diagnostics to autonomous vehicles. Many applications demand for embedded solutions that integrate into existing systems with tight real-time and power constraints. Convolutional Neural Networks (CNNs) presently achieve record-breaking accuracies in all image understanding benchmarks, but have a very high computational complexity. Embedded CNNs thus call for small and efficient, yet very powerful computing platforms.

This master thesis explores the potential of FPGA-based CNN acceleration and demonstrates a fully functional proof-of-concept CNN implementation on a Zynq System-on-Chip. The *ZynqNet Embedded CNN* is designed for image classification on ImageNet and consists of *ZynqNet CNN*, an optimized and customized CNN topology, and the *ZynqNet FPGA Accelerator*, an FPGA-based architecture for its evaluation.

ZynqNet CNN is a highly efficient CNN topology. Detailed analysis and optimization of prior topologies using the custom-designed *Netscope CNN Analyzer* have enabled a CNN with 84.5 % top-5 accuracy at a computational complexity of only 530 million multiply-accumulate operations. The topology is highly regular and consists exclusively of convolutional layers, ReLU nonlinearities and one global pooling layer. The CNN fits ideally onto the FPGA accelerator.

The *ZynqNet FPGA Accelerator* allows an efficient evaluation of ZynqNet CNN. It accelerates the full network based on a nested-loop algorithm which minimizes the number of arithmetic operations and memory accesses. The FPGA accelerator has been synthesized using High-Level Synthesis for the Xilinx Zynq XC-7Z045, and reaches a clock frequency of 200 MHz with a device utilization of 80 % to 90 %.

Organization of this report Chapter 1 gives an overview of the current opportunities and challenges regarding image understanding in embedded systems. The following chapter 2 introduces the central concepts of *Convolutional Neural Networks* (CNNs) and *Field-Programmable Gate Arrays* (FPGAs), as well as a number of CNN topologies and CNN accelerators from prior work. Chapter 3 dives deep into the analysis, training and optimization of CNN architectures, and presents our customized *ZynqNet CNN* topology. Next, chapter 4 shifts the focus onto the design and implementation of our FPGA-based architecture for the evaluation of CNNs, the *ZynqNet FPGA Accelerator*, and reports lessons learned from the application of High-Level Synthesis. Finally, chapter 5 presents the performance results of the overall *ZynqNet Embedded CNN* system, before the conclusion in chapter 6 puts these in a bigger perspective.

Acknowledgement

First and foremost, I would like to thank my supervisor Emanuel Schmid for the pleasant collaboration, the fruitful discussions, the helpful guidance and his excellent support during the project. You offered me full confidence and freedom, yet were always there when I needed feedback, a different point of view or new ideas. I also thank Felix Eberli for arranging this project, for involving me in various interesting meetings and discussions, and for his generous support.

Special thanks also go to professor Dr. Anton Gunzinger for giving me the chance to work on a fascinating project of practical relevance, and to the whole staff at Supercomputing Systems AG for the warm welcome and the pleasant stay.

Finally, I want to express my gratitude to my family, my friends and my fiancée. You've always had my back, and I could not have made it here without your constant and unconditional support. Thank you.

Contents

1 Introduction 1

1.1 Motivation . . . 1

1.2 Contribution . . . 2

2 Background and Concepts 3

2.1 Convolutional Neural Networks . . . 3

2.1.1 Introduction to Neural Networks . . . 3

2.1.2 Introduction to Convolutional Neural Networks . . . 5

2.1.3 Network Topologies for Image Classification . . . 9

2.1.4 Compression of Neural Network Models . . . 11

2.2 Field-Programmable Gate Arrays . . . 13

2.2.1 Introduction to Field-Programmable Gate Arrays . . . 13

2.2.2 Introduction to High-Level Synthesis . . . 14

2.3 Embedded Convolutional Neural Networks . . . 15

2.3.1 Potential Hardware Platforms . . . 15

2.3.2 Existing CNN Implementations on Embedded Platforms . . . 16

2.4 Project Goals and Specifications . . . 18

3 Convolutional Neural Network Analysis, Training and Optimization 21

3.1 Introduction . . . 21

3.2 Convolutional Neural Network Analysis . . . 21

3.2.1 Netscope CNN Analyzer . . . 22

3.2.2 Characteristics of Resource Efficient CNNs . . . 23

3.2.3 Efficiency Analysis of Prior CNN Topologies . . . 23

3.3 Convolutional Neural Network Training . . . 25

3.3.1 Construction of a GPU-based Training System . . . 25

3.3.2 CNN Training with Caffe and DIGITS . . . 26

3.4 Network Optimization . . . 26

3.4.1 Optimizations for Efficiency . . . 26

3.4.2 Optimizations for FPGA Implementation . . . 28

3.4.3 Optimizations for Accuracy . . . 29

3.4.4 Final Results . . . 30

4 FPGA Accelerator Design and Implementation 31

4.1 Introduction . . . 31

4.1.1 Zynqbox Platform Overview . . . 31

4.1.2 Data Type Considerations . . . 31

4.1.3 Design Goals . . . 33

4.2 Algorithm Design . . . 34

4.2.1 Requirements Analysis . . . 34

4.2.2 Algorithmic Options . . . 35

4.2.3 Parallelization . . . 36

4.2.4 Data Reuse . . . 38

4.3 Hardware Architecture and Schedule . . . 40
4.4 High-Level Synthesis Design Flow . . . 42
4.4.1 Introduction to Vivado HLS . . . 43
4.4.2 Coding Style . . . 44
4.4.3 Compiler Directives . . . 46
4.4.4 Limitations and Problems . . . 51
4.5 Post-HLS Design Flow . . . 53
4.5.1 Vivado Design Suite . . . 55
4.5.2 Zynq Driver Development . . . 55
5 Evaluation and Results **59**
5.1 ZynqNet CNN Performance . . . 59
5.1.1 Accuracy . . . 61
5.1.2 Computational Complexity . . . 61
5.1.3 Memory Requirements . . . 61
5.1.4 Resource Efficiency . . . 62
5.2 ZynqNet FPGA Accelerator Performance . . . 62
5.2.1 Resource Utilization . . . 62
5.2.2 Maximum Clock Frequency . . . 63
5.2.3 Operation Schedule . . . 63
5.2.4 Potential Improvements . . . 64
5.3 System Performance . . . 65
5.3.1 Throughput . . . 65
5.3.2 Power Efficiency . . . 65
6 Conclusion **67**
Appendix A Declaration of Originality **69**
Appendix B Task Description **70**
Appendix C Convolutional Neural Network Visualizations **72**
C.1 3D Illustration of Convolutional Layers . . . 72
C.2 Netscope Visualizations of Different CNN Topologies . . . 73
C.3 Advanced Usage Tips and Restrictions for Netscope . . . 75
Appendix D CNN Training Details and Results **76**
D.1 Hardware Components of the CNN Training Workstations . . . 76
D.2 Screenshots from the DIGITS CNN Training Software . . . 77
D.3 Overview of all CNN Training Experiments . . . 78
D.4 Layer Description Table for ZynqNet CNN . . . 79
D.5 Tips and Trick for the Training of CNNs . . . 80
Appendix E FPGA Accelerator Details **83**
E.1 Analysis of the Pipeline Flushing Issue . . . 83
E.2 Detailed Block Diagram for the ZynqNet FPGA Accelerator . . . 84
Bibliography **85**

List of Figures

2.1 Illustration of Biological and Aritificial Neurons 4
2.2 Illustration of a Fully Connected Neural Network 4
2.3 Convolutional Layer Nomenclature and Illustration 6
2.4 Comparison of Non-Linear Activation Functions 8
2.5 Example Images from the ImageNet Dataset 9
2.6 Block Diagram for Microsoft's FPGA-based CNN Accelerator 17
2.7 Illustration of DSE and Roofline Model by Zhang et al. 18
2.8 Illustration of the System-Level Design Approach to this Project 19

3.1 Screenshots of the Netscope CNN Analyzer 22
3.2 Design Space Exploration of CNN Topologies from Prior Work 24
3.3 SqueezeNet and ZynqNet Computational Complexity Analysis 27
3.4 SqueezeNet and ZynqNet Capacity and Dimension Analysis 28

4.1 Schematic of the SCS Zynqbox Platform 32
4.2 Topology Visualization of the ZynqNet CNN 34
4.3 Illustration of the Input Line Buffer in the ZynqNet FPGA Accelerator 39
4.4 Algorithmic Schedule of the ZynqNet FPGA Accelerator 41
4.5 High-Level Block Diagram of the ZynqNet FPGA Accelerator 42
4.6 Example of Different Array Partitioning Modes in Vivado HLS 52
4.7 Vivado Design Suite Block Design of the ZynqNet FPGA Accelerator 55

5.1 Design Space Exploration of CNN Topologies including ZynqNet CNN 60

C.1 3D Illustration of Convolutional Layers in a ZynqNet Fire Module 72
C.2 Netscope Visualizations of CNN Topologies from Prior Work 73
C.3 Detailed Netscope Visualizations of SqueezeNet and ZynqNet CNN 74

D.1 Photos of the GPU-based CNN Training Workstations 76
D.2 Screenshots from the DIGITS CNN Training Software 77

E.1 Detailed Block Diagram for the ZynqNet FPGA Accelerator 84

List of Tables

2.1 Comparison of CNN Topologies for ImageNet Classification from Prior Work . 10

5.1 Comparison of CNN Topologies for ImageNet Classification including ZynqNet 61
5.2 FPGA Utilization Report 63
5.3 System Power Measurement Results 66

D.1 Hardware Components of the CNN Training Workstations 76
D.2 Overview of all CNN Training Experiments 78
D.3 Layer Description Table for ZynqNet CNN 79

E.1 Analysis of the Pipeline Flushing Issue 83

1 Introduction

> *It is clear that humans will soon outperform state-of-the-art image classification models only by use of significant effort, expertise, and time.*
>
> — **Andrej Karpathy**
> (Deep Learning Expert, OpenAI)

1.1 Motivation

Image understanding is a very difficult task for computers. Nevertheless, advanced Computer Vision (CV) systems capable of image classification, object recognition and scene labeling are becoming increasingly important in many applications in robotics, surveillance, smart factories and medical diagnostics. Unmanned aerial vehicles and autonomous cars, which need to perceive their surroundings, are further key applications.

In the last few years, significant progress has been made regarding the performance of these advanced CV systems. The availability of powerful computing platforms and the strong market pull have shaped a very fast-paced and dynamic field of research. Former approaches to image understanding, which mainly relied on hand-engineered features and hard-coded algorithms, are increasingly being replaced by *machine learning* concepts, where computers learn to understand images by looking at thousands of examples. These advanced learning algorithms, which are based on recent high-performance computing platforms as well as the abundance of training data available today, are commonly referred to as *deep learning*.

Convolutional Neural Networks (CNNs) currently represent the most promising approach to image understanding in CV systems. These brain-inspired algorithms consist of multiple layers of feature detectors and classifiers, which are adapted and optimized using techniques from machine learning [1]. The idea of neural networks has been around for almost 80 years [2], yet only the latest generations of high-performance computing hardware have allowed the evaluation and training of CNNs deep and wide enough for good performance in image understanding applications. The progress in these last years has been amazing though, and state-of-the-art convolutional neural networks already rival the accuracy of humans when it comes to the classification of images [3].

This exceptional performance of CNNs comes at the cost of an enormous computational complexity. The real-time evaluation of a CNN for image classification on a live video stream can require billions or trillions of operations per second. The effort for image segmentation and scene labeling is even significantly higher. While this level of performance can be reached with the most recent Graphics Processing Units (GPUs), there is the simultaneous wish to embed such solutions into other systems, such as cars, drones, or even wearable devices, which exhibit strict limitations regarding physical size and energy consumption. Future embedded CNNs thus call for small and efficient, yet very powerful computing platforms.

Different platforms have been considered for efficient high-performance implementations of CNNs, and *Field-Programmable Gate Arrays* (FPGAs) are among the most promising of them. These versatile integrated circuits provide hundreds of thousands of programmable logic blocks and a configurable interconnect, which enables the construction of custom-tailored accelerator architectures in hardware. These have the potential to deliver the computational power required by embedded CNNs within the size and power envelopes dictated by their respective applications.

1.2 Contribution

Initially, this master aimed to explore, benchmark and optimize one or more commercial approaches to the acceleration of convolutional neural networks on FPGAs, with a focus on embedded systems. Multiple FPGA and intellectual property vendors have announced frameworks and libraries that target the acceleration of deep learning systems.[1] However, none of these solutions turned out to be ready and available for testing.

Nevertheless, we decided to further pursue this promising approach by building our own proof-of-concept FPGA-based CNN implementation from scratch, with a special focus on the optimized co-operation between the underlying hardware architecture and the convolutional neural network. The result is the *ZynqNet Embedded CNN*, an FPGA-based convolutional neural network for image classification. The solution consists of two main components:

1. The *ZynqNet CNN*, a customized convolutional neural network topology, specifically shaped to fit ideally onto the FPGA. The CNN is exceptionally regular, and reaches a satisfying classification accuracy with minimal computational effort.
2. The *ZynqNet FPGA Accelerator*, a specialized FPGA architecture for the efficient acceleration of ZynqNet CNN and similar convolutional neural networks.

ZynqNet CNN is trained offline on GPUs using the Caffe framework, while the ZynqNet FPGA Accelerator employs the CNN for image classification, or *inference*, on a Xilinx Zynq XC-7Z045 System-on-Chip (SoC). Both components have been developed and optimized within the six month time frame of this master thesis, and together constitute a fully functional convolutional neural network implementation on the small and low-power Zynq platform.

This report documents the ZynqNet CNN and the ZynqNet FPGA Accelerator and gives insight into their development. In addition, the *Netscope CNN Analyzer* is introduced, a custom tool for visualizing, analyzing and editing convolutional neural network topologies. Netscope has been used to analyze a number of different CNN architectures, and the findings are presented in the form of a *Design Space Exploration* (DSE) of CNN topologies from prior work. Finally, the performance of the ZynqNet Embedded CNN is evaluated and its performance is compared to other platforms.

[1] The following commercial frameworks and libraries target the acceleration of CNNs using FPGAs:

- Auviz Systems AuvizDNN Framework [4], [5], [6]
- Falcon Computing Solutions machine learning libraries based on OpenCL [7]
- MulticoreWare machine learning libraries based on SDAccel [8]

Additionally, Altera OpenCL [9] and Xilinx SDAccel [10] are generic frameworks which allow computation kernels to be offloaded from a host processor onto FPGA-based accelerators. However, these frameworks do not directly accelerate CNNs and were therefore not considered ready-to-use, although both companies mention the acceleration of deep learning algorithms as a major use case.

2 Background and Concepts

> *If I have seen further than others, it is by standing upon the shoulders of giants.*
>
> **— Isaac Newton**

This chapter introduces two of the main concepts behind this thesis: Convolutional Neural Networks (CNNs, section 2.1) and Field-Programmable Gate Arrays (FPGAs, section 2.2). In addition, we present a number of CNN topologies (section 2.1.3) and embedded CNN implementations from prior work (section 2.3), before the final section compiles the requirements and specifications for our own FPGA-accelerated embedded CNN (section 2.4).

2.1 Convolutional Neural Networks

The following sections give a brief overview of neural networks in general, and of convolutional neural networks in particular. First, an intuitive explanation for the inner workings of neural networks is presented, including a high-level description of the network training process (section 2.1.1). The next section dives into convolutional neural networks, an architecture particularly suited for processing images, and gives an overview of the construction of these networks (section 2.1.2). Finally, the most important CNN topologies for image classification are introduced and characterized (section 2.1.3).

For a more conclusive introduction to this rapidly expanding field, the excellent course *CS231n: Convolutional Neural Networks for Visual Recognition* by Andrej Karpathy is highly recommended and publicly available [11]. Further starting points include the *Deep Learning Book* by Goodfellow, Bengio et al. [12], the online course by Nielsen [13] as well as the Caffe tutorials [14].

2.1.1 Introduction to Neural Networks

Biological Inspiration Neural networks are a family of computation architectures originally inspired by biological nervous systems. The human brain contains approximately 86 billion neurons connected by 10^{14}–10^{15} synapses. Each neuron receives input signals at its dendrites and produces output signals along its axon, which branches out and connects to the dendrites of other neurons via synapses. These synapses influence the transfer of information from one neuron to the other by amplifying or attenuating the signals, or even inhibiting the signal transfer at other synapses. Together, the billions of conceptually simple neurons form an incredibly complex interacting network which enables us humans to see, hear, move, communicate, remember, analyze, understand and even to fantasize and dream [11], [15].

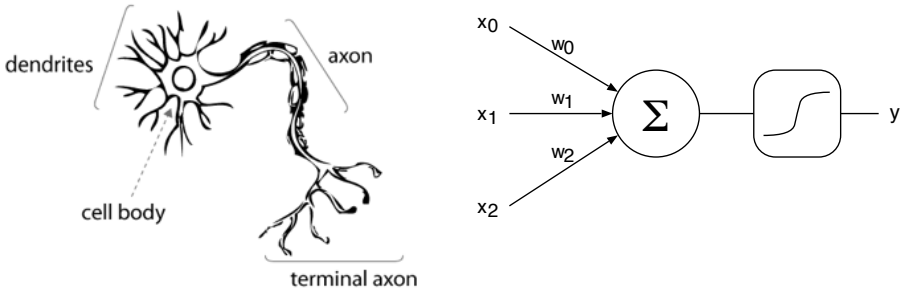


Figure 2.1.: A Biological Neuron and its Artificial Counterpart. (Image adapted from [16])

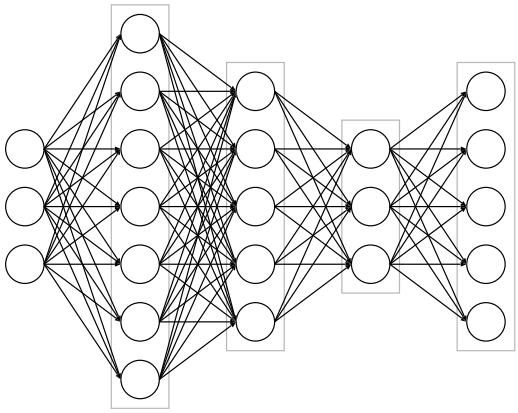


Figure 2.2.: Example of a Neural Network with 4 Fully-Connected Layers, 3 Inputs and 5 Outputs.

Artificial Neurons The basic building block in artificial neural networks is the *artificial neuron*, depicted in fig. 2.1. The artificial neuron receives a number of input signals x_i from other neurons. These input signals are multiplied with weights w_i to simulate the synaptic interaction at the dendrites. The weighed input signals are summed up, biased with a fixed w_b and fed into a non-linear *activation function*, which produces the neuron's output signal $y = f\left(\sum \left[x_i \cdot w_i\right] + w_b\right)$ [11]. The weights w can be seen as the tuning knobs that define the neuron's reaction to a given input signal, and their values can be adjusted in order to learn to approximate a desired output signal [11], [17].[1]

Neural Network Organization A *neural network* is formed by interconnecting many artificial neurons. Usually, the neurons are arranged in a directed acyclic graph to form a *feed-forward neural network*.[2] The neurons are further grouped into layers, and connections are only allowed between neurons of adjacent layers. Figure 2.2 shows an example of a four-layer feed-forward neural network with fully-connected layers[3] and five outputs.

[1] A single artificial neuron can naturally only give a simple approximation. Interestingly, already a two-layer neural network is a *universal approximator* that can approximate any continuous functions to an arbitrary degree of precision using a finite amount of neurons. An intuitive explanation for this theorem is given in [13, ch. 4], which also justifies the application of a non-linear activation function.

[2] Neural networks with directed cycles in their structure are called *Recurrent Neural Networks* (RNNs) and play an important role in speech recognition, text-to-speech synthesis and natural language processing. Thanks to their inherent memory, they are well suited for processing time-dependent signals. However, RNNs are usually difficult to train and have problems with scaling. Some of these difficulties can be mitigated by using *Long Short-Term Memory* (LSTM) networks, which are currently the most popular type of RNNs [18], [19].

[3] In a fully-connected layer, each output from the previous layer is connected to every neuron in the current layer. A feed-forward neural network consisting of fully-connected layers is also called *Multilayer Perceptron* (MLP) [11].

Network Training The parameters in a neural network are not manually chosen, but *learned* during a training phase. The most popular training approach is called *supervised learning* and requires a set of *labeled training examples*. One optimization pass through all training examples is called a *training epoch*. Depending on the type of data and the capacity of the neural network, a complete training session can take anywhere from one to a few hundred epochs. The training starts with small, randomly initialized weights.[4] One by one, the examples are fed through the network (so-called *forward pass*). The resulting outputs are compared to the *ground truth labels* using a *loss function*, which measures how much the output deviates from the expected result. The goal of the learning process is then to minimize this loss (or error) on the training set by optimizing the weight parameters. *Stochastic Gradient Descent* is the most popular optimization method currently used for training neural networks. The gradient descent algorithm computes a gradient vector that describes each weight's influence on the error. These gradients can be efficiently calculated by *backpropagation of the output error* through the network (so-called *backward pass*). The optimization loop repeatedly takes a training example, calculates the current loss (forward pass), derives the gradient vector (backward pass), and adjusts all weights by a small amount in the opposite direction of their respective gradient (update phase). The magnitude of these updates is determined by the so-called *learning rate*. An alternate version of the algorithm, called *Batch Gradient Descent*, defers the weight updates, and first computes and averages the gradients of a *batch* of training examples. This allows the computation to be vectorized and executed more efficiently on platforms which support vector instructions, including GPUs, DSPs and most CPUs [11], [12], [20].

Performance Validation By iteratively adjusting the weights, the network ideally converges towards a solution with minimal loss and thus with a good approximation of the desired output on the training set. Every few epochs, the performance of the model is verified with an array of *validation examples* which were not used during training.[5] If the training set is representative for the actual "real-world" data, the network also delivers good estimations for previously unseen examples. If however the training set is too small, or the network's learning capacity too high, the neural network can memorize examples "by heart" and lose its ability to generalize. Such *overfitting* can be counteracted with enlarged training sets (possibly using *data augmentation* strategies such as mirroring, rotation and color transformations) as well as changes to the network structure (such as the addition of regularization methods) [11].

2.1.2 Introduction to Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a special class of neural networks particularly suited for operation on 2D input data such as images. They are widely used for *image classification*, *object recognition* and *scene labeling* tasks.

Nomenclature The input to each layer in a convolutional neural network consists of a stack of ch_{in} 2D images of dimension $h_{\text{in}} \times w_{\text{in}}$, the so-called *input feature maps*. Each layer produces a stack of ch_{out} 2D images of dimension $h_{\text{out}} \times w_{\text{out}}$, called *output feature maps*. An illustration can be found in fig. 2.3.

[4] Initialization with a constant (e.g. zero) would make all neurons compute exactly the same outputs and would prevent any learning. The exact initialization strategy can be quite important.

[5] The validation examples are usually a fraction of the labeled examples which are set aside from the beginning. It is common to use around 20 % to 25 % of the labeled examples as validation set.

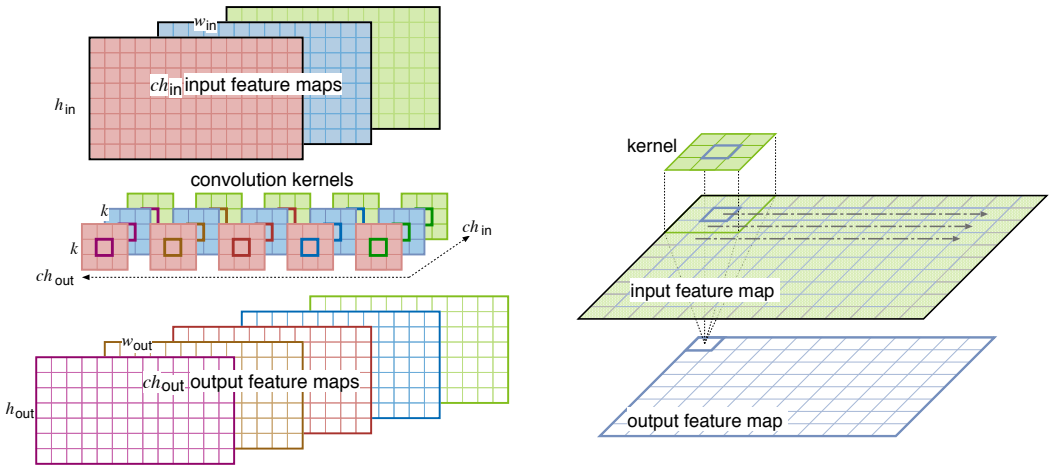


Figure 2.3.: Left: Illustration of the CNN Layer Nomenclature. The ch_{in} input feature maps (solid R, G, B) are transformed into ch_{out} output feature maps (outlined P, B, R, B, G) by applying $ch_{\text{in}} \times ch_{\text{out}}$ filter kernels of size $k \times k$. Right: Illustration of the 2D convolution between a 3×3 kernel and an input feature map by sliding the kernel over the input pixels and performing multiply-accumulate operations at each pixel position.

Motivation When neural networks are employed for image-related tasks, their input usually consists of pixel data. Even for an image with a modest resolution of 256×256 RGB pixels, the resulting input consists of $256 \times 256 \times 3 \approx 200\,000$ elements, and a subsequent fully-connected neural network layer would need billions of weights. Luckily, there is no need for full connectivity when dealing with pixel data thanks to the locality of information in images. In order to decide whether there is a car in the center of an image one does not need to consider the color of the top-right corner pixel — and the bottom-right pixels usually do not influence the class assigned to the top-left pixels. The important information in images can be captured from local neighborhood relations. Strong contrasts indicate edges, aligned edges result in lines, combined lines can result in circles and contours, circles can outline a wheel and multiple nearby wheels can point to the presence of a car [11], [21]. This locality of information in images is exploited in convolutional neural networks by replacing the fully-connected layers with *convolutional layers*.

Weight Sharing by Convolution A convolutional layer contains a $ch_{\text{in}} \times ch_{\text{out}}$ array of kernels, which are small filters of size $k \times k$ (typically 1×1, 3×3, 5×5, 7×7 or 11×11). These kernels are applied to the input feature maps by means of 2D convolution. Each output pixel is thus generated from just a small *local receptive field* in the input image. ch_{out} filter kernels are slid over each input feature map. For each input feature map, this results in ch_{out} partial output feature maps. The final output feature maps are formed by summing the partial output feature maps contributed by all ch_{in} input channels (see fig. 2.3 for an illustration, a mathematical formulation follows in eq. (4.2) in section 4.2.2). Instead of requiring $(h_{\text{in}} \times w_{\text{in}} \times ch_{\text{in}}) \times (h_{\text{out}} \times w_{\text{out}} \times ch_{\text{out}})$ weights, the number of parameters in a convolutional layer is thus reduced to $(k \cdot k) \times (ch_{\text{in}} \times ch_{\text{out}})$. The independence from the input image dimensions also enables large images to be processed without an exploding number of weights [11], [13], [17].

Layer Types Convolutional neural networks are constructed by stacking a number of generic network layers, which transform the input feature maps of dimension $(h_{\text{in}} \times w_{\text{in}} \times ch_{\text{in}})$ into output feature maps of dimension $(h_{\text{out}} \times w_{\text{out}} \times ch_{\text{out}})$ [11], [14]. A typical CNN consists of the following layer types:

Convolutional Layers apply $(ch_{\text{in}} \times ch_{\text{out}})$ filters of size $(k \times k)$ to generate the output feature maps. For filters larger than 1×1, border effects reduce the output dimensions. To avoid this effect, the input image is typically *padded* with $p = \lfloor k/2 \rfloor$ zeros on each side. The filters can be applied with a *stride* s, which reduces the output dimensions to $w_{\text{out}} = w_{\text{in}}/s$ and $h_{\text{out}} = h_{\text{in}}/s$.

Nonlinearity Layers apply a non-linear activation function to each input pixel. The most popular activation function is the *Rectified Linear Unit (ReLU)* which computes $f(x) = \max(0, x)$ and clips all negative elements to zero. Early networks used sigmoidal functions such as $f(x) = 1/(1 + e^{-x})$ or $f(x) = \tanh(x)$, but these are no longer used because of their computational complexity and their slowing effect on convergence during training. More recent ideas include the *Parametric ReLU (PReLU)* $f(x) = \max(\alpha \cdot x, x)$ with learnable parameter α [22], *Maxout* [23] and *Exponential Linear Units (ELU)* [24]. Figure 2.4 shows a comparison of some of these options.

Pooling Layers reduce the spatial dimensions of the input by summarizing multiple input pixels into one output pixel. Two popular choices are *max-pooling* and *avg-pooling*, which summarize their local receptive field by taking the maximum or the average value of the pixels, respectively. They are usually applied to a patch of 2×2 or 3×3 input pixels with a stride $s = 2$, but can also be applied as *global pooling* to the whole input image, in order to reduce the spatial output dimensions to 1×1 pixels.

Fully-Connected Layers are often used as the last layers in a CNN to compute the class scores in image classification applications. Even though the spatial dimensions h_{in} and w_{in} in the last layers are typically heavily reduced, the fully-connected layers often account for most of the weights in these CNNs.

Local Response Normalization (LRN) Layers introduce competition between the neurons of adjacent output channels by normalizing their responses with respect to a certain neighborhood of N channels. LRN layers were introduced in the famous *AlexNet* architecture [25], but are used less often in recent CNNs.

Batch Normalization (BN) Layers were introduced in 2015 by researchers at Google [26]. Batch Normalization is applied after every training batch and normalizes the layer's output distribution to zero-mean, unit-variance. The uniform input distribution to subsequent layers should allow higher learning rates and thereby accelerate the training and improve the accuracy of the network. However, as of this writing, BN layers are not fully supported on all training platforms and can be difficult to employ in practice.

Dropout Layers are a popular method to combat overfitting in large CNNs. These layers randomly drop a selectable percentage of their connections during training, which prevents the network from learning very precise mappings, and forces some abstraction and redundancy to be built into the learned weights.

Softmax Layers are the most common *classifiers*. A classifier layer is added behind the last convolutional or fully-connected layer in each image classification CNN, and squashes the raw class scores z_i into class probabilities P_i according to $P_i = e^{z_i}/\sum_k e^{z_k}$, which results in a vector P that sums up to 1.

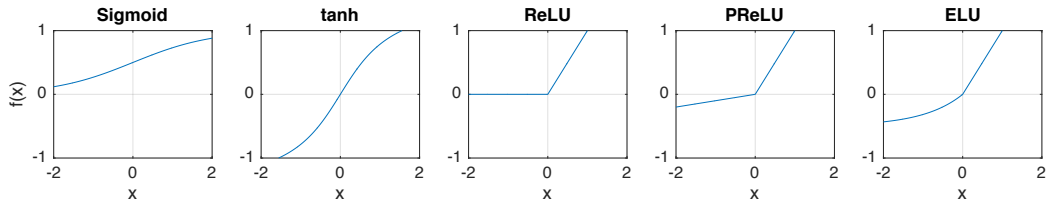


Figure 2.4.: The Non-Linear Activation Functions Sigmoid, tanh, ReLU, PReLU and ELU.

Neural Network Training Frameworks There are many popular software frameworks specifically built for the design and training of neural networks, including, among others, the *Neural Network Toolbox for MATLAB* [27], *Theano* [28] with the extensions *Lasagne* [29] and *Keras* [30], *Torch* [31], *TensorFlow* [32] and *Caffe* [33]. Most of these frameworks can utilize one or multiple GPUs in order to heavily accelerate the training of neural networks. For this thesis, the Caffe framework has been used due to its maturity, its support in the GPU-based training system *NVidia DIGITS*, [34] and most importantly because of the excellent availability of network descriptions and pretrained network topologies in native Caffe format.

Network Specification In order to fully describe a convolutional neural network, the following information is required:

1. a topological description of the network graph
2. a list of layers and their settings
3. the weights and biases in each layer
4. (optionally) a training protocol

In Caffe, the network description and the layer settings are stored in a JSON-like, human-readable text format called `.prototxt`. The weights are saved in binary `.caffemodel` files. The training protocol is also supplied in `.prototxt` format and includes settings such as the base learning rate, the learning rate schedule, the batch size, the optimization algorithm, as well as the random seeds for training initialization. These settings are only needed if the network is to be trained from scratch or *finetuned*, which refers to the process of adapting a trained network to a different dataset. For *inference*, where a fully trained network is utilized for forward-computation on new input data, the network description and the trained weights are sufficient.

Figure 2.5.: Sample Images from the ImageNet Challenge (white shark, banana, volcano, fire engine, pomeranian, space shuttle, toilet paper)

2.1.3 Network Topologies for Image Classification

One of the most interesting, yet also one of the hardest problems in Computer Vision is *Image Classification*: The task of correctly assigning one out of several possible labels to a given image. Examples for this problem include yes-or-no decisions (Is there a person in front of the car? Is this tissue sample cancerous?) but also recognition tasks with a large number of labels (What breed of dog is this? Who is on this photo?). As an extension of image classification, *scene labeling* assigns a class to every pixel of the input image.

ImageNet Challenge The *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* is an annual competition where participants develop algorithms to classify images from a subset of the *ImageNet* database. The ImageNet database consists of more than 14 million photographs collected from the Internet, each labeled with one groundtruth class. The ILSVRC training set consists of approximately 1.2 million images in 1000 different classes, covering a huge variety of objects (from toilet paper, bananas and kimonos to fire trucks, space shuttles and volcanoes), scenes (from valleys and seashores to libraries and monastries) and animals (120 breeds of dogs, but also axolotls, sharks and triceratop dinosaurs). Some sample images from the challenge are shown in fig. 2.5. Participants are allowed to make five predictions. The *top-1 accuracy* tracks the percentage of correct labels assigned at first guess, and the *top-5 accuracy* takes all five predictions into account. Humans can reach approximately 5 % top-5 error rate with explicit training and concentrated effort [3], [35].

CNN Topologies for Image Classification on ImageNet The huge number of training samples and the difficulty of the problem make the ImageNet challenge an ideal playground for machine learning algorithms. Starting with *AlexNet* in 2012, convolutional neural networks have taken the lead in the ILSVRC competition, and the top-1 and top-5 error rates of the winning entries have dropped significantly every since. The most important topologies are summarized in table 2.1, visualized in fig. C.2 and quickly introduced in the following list.[6]

AlexNet by Alex Krizhevsky et al. from the University of Toronto was the first CNN to win the ILSVRC in 2012. AlexNet consists of 5 convolutional layers, has 60 million parameters and requires approximately 1.1 billion *multiply-accumulate* (MACC) operations for one forward pass. The network achieved a groundbreaking top-5 error rate of 15.3 % on ILSVRC 2012, with the second-best entry left behind at 26.2 % [25].

Network-in-Network (NiN) by Min Lin et al. from the National University of Singapore was published as a novel CNN architecture in 2013. The NiN architecture consists of small, stacked multilayer perceptrons which are slid over the respective input just like convolutional filters. Additionally, the authors use global average pooling in the

[6] The top-5 error rates reported in this list correspond to the performance of the ILSVRC submissions, unless otherwise stated. The participants often use multi-net fusion (fusing the predictions of multiple separately trained networks) and multi-crop evaluation (averaging the predictions made on different crops of the input image) to boost their accuracy. The single-net single-crop error rate of these CNNs can differ significantly.

Table 2.1.: Comparison of Different CNN Topologies for Image Classification on ImageNet. The top-5 error rate is listed for single-net, single-crop evaluation. #MACCs is the number of multiply-accumulate operations in one forward pass. #activations is the total pixel count in all output feature maps.

	#conv. layers	#MACCs [millions]	#params [millions]	#activations [millions]	ImageNet top-5 error
AlexNet	5	1 140	62.4	2.4	19.7%
Network-in-Network	12	1 100	7.6	4.0	~19.0%
VGG-16	16	15 470	138.3	29.0	8.1%
GoogLeNet	22	1 600	7.0	10.4	9.2%
ResNet-50	50	3 870	25.6	46.9	7.0%
Inception v3	48	5 710	23.8	32.6	5.6%
Inception-ResNet-v2	96	9 210	31.6	74.5	4.9%
SqueezeNet	18	860	1.2	12.7	19.7%

classifier instead of fully-connected layers. This makes the network much smaller in terms of parameters. NiN never officially participated in ILSVRC, but can be trained on the ImageNet dataset and reaches approximately AlexNet-level accuracy [36], [37].

VGG stands for Visual Geometry Group, University of Oxford, and also names this group's CNN architecture which won part of the ILSVRC 2014 challenge. The researchers experimented with deep CNNs containing up to 19 convolutional layers. The most popular variant *VGG-16* has a depth of 16 layers, and a very regular structure, consisting exclusively of 3×3 convolution and 2×2 max-pooling layers. The spatial dimensions are steadily reduced from 224×224 pixels to 7×7 pixels, while the number of channels is simultaneously increased from 3 to 4096. The network reached a top-5 error of 7.3 %. However, VGG-16 contains almost 140 million weights and one forward pass requires nearly 16 billion MACC operations [38].

GoogLeNet by Christian Szegedy et al. from Google is a milestone CNN architecture published just a few days after the VGG architecture. The 22-layer GoogLeNet set a new ILSVRC classification record with a top-5 error rate of 6.67 %, while requiring only 1.2 million parameters and 0.86 billion MACC operations.[7] The savings are achieved by a more complex architecture which employs so-called *Inception modules*. These modules are a *network-in-network* sub-architecture which first uses a 1×1 convolutional layer to reduce the number of channels, before expanding this compressed representation again using parallel convolutional layers with kernel sizes 1×1, 3×3 and 5×5. The reduction in the channel dimension decreases the number of parameters and MACC operations in both the reducing and the expanding layers, and the composition of multiple layers increases the non-linear expressiveness of the network. To improve training convergence, GoogLeNet makes use of LRN layers [40].

ResNet by Kaiming He et al. from Microsoft Research won the ILSVRC in 2015. Their very deep ResNet-152 model achieved a top-5 error rate of less than 5.7 % by using 152 convolutional layers. Models with a depth of more than 20 convolutional layers were previously very hard to train. The researchers solved this problem by including detours around each batch of two subsequent convolutional layers, summing both the detoured original and the filtered representation together at the junction points. This topology

[7] The ILSVRC-2014 winning entry used multi-crop evaluation on 144 crops for this result. Single-crop performance is rather in the order of 9 % top-5 error [39].

resembles a function $y = F(x) + x$ where the network only needs to learn the *residual function* $F(x)$, merely "adding information" rather than reinventing the wheel every two layers. The smaller version ResNet-50 uses 50 convolutional layers and Batch Normalization, has 47 million parameters and needs 3.9 billion MACC operations per forward pass to reach a top-5 error of 6.7 % [41].

Inception v3 and v4 by Christian Szegedy et al. are Google's latest published image classification CNNs. The GoogLeNet architecture has been thoroughly studied and optimized in the Inception v3 paper [42], with valuable hints on how to design and modify CNNs for efficiency. The Inception v4 paper [43], published in February 2016, studies the positive effects of residual connections in Inception module-based architectures and presents *Inception-ResNet-v2* which reaches a 4.1 % top-5 error rate on the ILSVRC dataset. All recent Inception architectures make heavy use of Batch Normalization layers [42], [43].

SqueezeNet by Forrest Iandola et al. from UC Berkeley, also published in February 2016, differs from the other CNN architectures in this list because the design goal was not record-breaking accuracy. Instead, the authors developed a network with an accuracy similar to AlexNet, but with 50× less parameters. This parameter reduction has been achieved by using *Fire modules*, a reduce-expand micro-architecture comparable to the Inception modules, and careful balancing of the architecture. The 18-layer SqueezeNet uses 7×7, 3×3 and 1×1 convolutions, 3×3 max-pooling, dropout and global average pooling, but neither fully-connected, nor LRN, nor Batch Normalization layers. One forward pass requires only 860 million MACC operations, and the 1.24 million parameters are enough to achieve less than 19.7 % single-crop top-5 error [44].[8]

2.1.4 Compression of Neural Network Models

State-of-the-art CNNs require significant amounts of memory for their weights (e.g. 560 MB for VGG-16 with 32-bit weights) which can be problematic for example regarding over-the-air updates or the deployment on embedded systems. Researchers have been looking for ways to reduce both the number of weights, and the memory required per weight.

Kernel Decomposition and Pruning Denil et al. demonstrate that up to 95 % of all weights in their CNN can be predicted instead of learned, without a drop in accuracy [45]. Denton et al. approximate fully trained convolution kernels using singular value decomposition (SVD) [46], while Jin et al. replace the 3D convolution operation by three consecutive one-dimensional convolutions (across channel, horizontal, vertical) [47]. Similar methods have been used to efficiently deploy CNNs on smartphones [48]. A final idea is *network pruning*, where small or otherwise unimportant weights are set to zero, which effectively removes the corresponding connections [49], [50].

Limited Numerical Precision Reducing the memory consumption of each weight is possible by replacing the typical *32-bit floating-point* weights either with *16-bit floating-point* weights [51], [52] or with *fixed-point approximations* of less than 32 bits [53]. Neural networks have been shown to tolerate this type of quantization very well. Hwang et al. successfully quantized most layers in their CNN to three bits [54], Sung et al. restricted their network

[8] AlexNet also has a top-5 error of 19.7 % with single-crop evaluation. The 15.3 % top-5 error on ILSVRC has been achieved using multi-net fusion and multi-crop evaluation.

to ternary values $(-1,0,1)$ with a negligible drop in accuracy [55], and Courbariaux et al. even train CNNs with binary weights and activations [56], [57]. With Ristretto, Gysel et al. recently published an automated CNN approximation tool which analyzes floating-point networks and condenses their weights to compact fixed-point formats, while respecting a maximally allowed accuracy drop [58].

Deep Compression Finally, Han et al. combine pruning, trained quantization and Huffman coding to reduce the storage requirement of AlexNet by a factor of $39\times$, and that of VGG-16 even $49\times$ without any drop in accuracy [59]. For all methods mentioned, finetuning the network with the compressed weights helps to recover most of the initial accuracy loss.

2.2 Field-Programmable Gate Arrays

This section gives a high-level introduction to *Field-Programmable Gate Arrays* (FPGAs). The first part highlights characteristics, strengths and weaknesses of this hardware platform, before the second part focuses on *High-Level Synthesis* (HLS), a relatively new methodology which makes it possible to program FPGAs in high-level languages such as C and C++.

2.2.1 Introduction to Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) are semiconductor devices consisting of a 2D array of configurable logic blocks (CLBs, or *logic slices*), which are connected via programmable interconnects. The interconnect can be thought of as a network of wire bundles running vertically and horizontally between the logic slices, with switchboxes at each intersection. Modern high-end FPGA generations feature hundreds of thousands of configurable logic blocks, and additionally include an abundance of hardened functional units which enable fast and efficient implementations of common functions.[9] The logic blocks, the fixed-function units as well as the interconnect are programmed electronically by writing a *configuration bitstream* into the device. The configuration is typically held in SRAM memory cells, and the FPGAs can be reprogrammed many times [60], [61].

FPGAs versus General-Purpose Processors The advantage of FPGA-based systems over traditional processor-based systems such as desktop computers, smartphones, most embedded systems, and also over GPUs, is the availability of freely programmable general-purpose logic blocks. These can be arranged into heavily specialized accelerators for very specific tasks, resulting in improved processing speed, higher throughput and energy savings. This advantage comes at the price of reduced agility and increased complexity during the development, where the designer needs to carefully consider the available hardware resources and the efficient mapping of his algorithm onto the FPGA architecture. Further, some algorithmic problems do not map well onto the rigid block structures found on FPGAs [60], [62].

FPGAs versus ASICs *Application-Specific Integrated Circuits* (ASICs) are custom-tailored semiconductor devices. In contrast to FPGAs, they do not suffer any area or timing overhead from configuration logic and generic interconnects, and therefore typically result in the smallest, fastest and most energy-efficient systems. However, the sophisticated fabrication processes for ASICs results in lengthy development cycles and very high upfront costs, which demands a first-time-right design methodology and very extensive design verification. Therefore ASICs are mostly suited for very high-volume, cost-sensitive applications where the non-recurring engineering and fabrication costs can be shared between a large number of devices. FPGAs with their reprogrammability are better suited for prototyping and short development cycles [60].

[9] This includes on-chip SRAM (Block RAM), USB, PCIe and Ethernet Transceivers, Serializer-Deserializer circuits, Digital Signal Processor (DSP) Slices, Cryptographic Accelerators, PLLs, Memory Interfaces and even full ARM processor cores.

2.2.2 Introduction to High-Level Synthesis

Hardware Description Languages and Register Transfer Level Design Traditionally, FPGAs are programmed using a *Hardware Description Language* (HDL) such as VHDL or Verilog. Most designs are described at *Register Transfer Level* (RTL), where the programmer specifies his algorithm using a multitude of parallel processes which operate on vectors of binary signals and simple integer data types derived from them. These processes describe combinational logic, basic arithmetic operations as well as registers, and are driven by the rising and falling edges of a clock signal. RTL descriptions are very close to the logic gates and wires that are actually available in the underlying FPGA or ASIC technology, and therefore the hardware that results from *RTL synthesis* can be closely controlled. However, the process of breaking down a given algorithm into logic blocks, processes and finite state machines on the register transfer level is very tedious and error-prone. Many design decisions have to be made before writing any code, and later changes are difficult and costly. This prevents iterative optimizations and demands a lot of intuition, experience and expert knowledge from designers [60].

Increasing the Level of Abstraction with HLS *High-Level Synthesis* (HLS) tries to lower this barrier to entry by enabling designers to specify their algorithms in a high-level programming language such as C, C++ or SystemC. Many implementation details are abstracted away and handled by the *HLS compiler*, which converts the sequential software description into a concurrent hardware description, usually at RTL level.

Vivado High-Level Synthesis *Vivado High Level Synthesis* (VHLS) by Xilinx Inc. is one of the most popular commercial HLS compilers. With VHLS, designers can use loops, arrays, structs, floats, most arithmetic operations, function calls, and even object-oriented classes. These are automatically converted into counters, memories, computation cores and handshake protocols as well as accompanying state machines and schedules. The compilation can be influenced using scripted *compiler directives* or embedded *compiler pragmas*, which are meta-instructions interpreted directly by the VHLS compiler. Operations are by default scheduled to be executed concurrently and as early as possible. Using the compiler pragmas, the designer can further influence the inference of memories and interfaces, the parallelization of loops and tasks, the synthesis of computation pipelines, etc. [62], [63]

Promises and Difficulties The increased abstraction level in High-Level Synthesis promises faster development cycles, flexible optimization strategies and much higher productivity at the cost of slightly less control on the end result. Especially with regard to every-increasing design complexities, shrinking time-to-market requirements and the abundant resources in modern FPGAs, such a compromise would be very welcome. However, HLS tools have been on the market for more than 12 years now, yet most engineers still use RTL descriptions for their FPGA and ASIC designs. The task of converting sequential, high-level software descriptions into fully optimized, parallel hardware architectures is tremendously complex. Although companies have invested hundreds of millions of dollars and years of research into HLS [64], [65], [66], the results attained are still highly dependent on the coding style and intricate design details. Because flaws and deficiencies in the compiler are only discovered during the design, the decision for HLS is associated with a non-negligble risk [67].

2.3 Embedded Convolutional Neural Networks

The following sections give a short overview of different options for the implementation of convolutional neural networks in embedded systems. All of these embedded implementations focus on *inference* using the CNN, and assume that the training is done offline using e.g. GPU-based training systems. Section 2.3.1 introduces the possible hardware platforms for the computation of CNNs, before section 2.3.2 presents a number of existing CNN implementations from prior work.

2.3.1 Potential Hardware Platforms

Embedded systems typically have very specific requirements and constraints such as limited power and energy budgets, finite battery capacities, small physical sizes resulting in limited heat dissipation capabilities, as well as high reliability requirements and hard real-time constraints. These characteristics make the development of algorithms and systems for the embedded market different from the scientific playground where many neural networks are currently researched. Still, there are a number of different options for the implementation of convolutional neural networks in embedded systems:

Central Processing Units (CPUs) are the processor cores found in most of today's devices, including desktop computers and smartphones. Most of these CPUs are general-purpose, flexibly programmable and built for good performance on a maximally wide range of computational workloads. There exist many different types of processors suitable for embedded systems, with different tradeoffs regarding speed and power requirements. However, CPUs compute results sequentially[10] and are thus not ideally suited for the highly parallel problem presented by convolutional neural networks.

Digital Signal Processors (DSPs) are highly specialized microprocessors. They are optimized for processing floating-point signals fast and efficiently (especially multiply-accumulate operations) and they typically include *Very Long Instruction Word* (VLIW) instructions to increase parallelism. Modern DSPs such as the Texas Instrument C6678 include eight cores, run at 1.25 GHz and compute up to 160 GFLOP/s at less than 15 W. Specialized vision processors such as Cadence Tensilica Vision DSP [68], [69] or the Movidius Myriad 2 [70] even promise teraflops of performance at just 1 W. However, DSPs are still primarily "few-core" processors which are optimized for fast sequential operation and thus cannot fully exploit the parallelism present in CNNs.

Graphics Processing Units (GPUs) are many-core processors which were originally designed for highly parallel graphical workloads. GPUs have recently been discovered for general-purpose computing tasks, referred to as *General-Purpose Computing on GPUs* (GPGPU), which is supported by the OpenCL and CUDA programming frameworks. High-end GPUs such as the NVidia GeForce GTX Titan X [71] contain more than 3000 floating-point processing cores running at 1 GHz, and offer more than 330 GB/s memory bandwidth. They compute up to 6600 GFLOP/s, but also consume up to 250 W. Mobile GPUs such as the NVidia Tegra X1 [72] (which is also used in the NVidia Jetson TX1 modules and the NVidia Drive PX platform) include up to 256 processing cores running at 1 GHz and a memory bandwidth of roughly 25 GB/s. They compute up to

[10] High-end CPUs can include multiple cores and SIMD instructions to attain a certain level of parallelization, but they are still primarily destined for sequential computation.

512 GFLOP/s while consuming less than 10 watts [73]. GPUs are well suited for the parallel workloads presented by CNNs and are fully supported by most deep learning frameworks. They constitute the primary platform for research in the area of CNNs.

Field-Programmable Gate Arrays (FGPAs) have been introduced in section 2.2. The largest devices, such as the Xilinx Virtex UltraScale+ XCVU13P, include more than 3 million logic cells, 12 thousand DSP slices and 56 MB of on-chip SRAM [74]. Estimating the floating-point performance of FPGAs is not straight forward [75], but a conservative estimate for the XCVU13P with 3 DSP slices per multiplication and $f = 300\,\mathrm{MHz}$ results in more than 1000 GFLOP/s at a few tens of watts [76], [77], [78]. FPGA designs work best for very regular calculations which can be heavily parallelized by building custom processing engines using the programmable logic blocks. Algorithms that require data-dependent branching and decisions are less suited for this type of parallelization and result in a poor utilization of the computational power. The performance of FPGA designs can be further increased by utilizing fixed-point or half-precision floating-point data formats.

Application-Specific Integrated Circuits (ASICs) are the ideal solution when it comes to maximum performance and maximum energy efficiency. However, ASICs are even less suited for irregular computation than FPGAs, and they further require much of the algorithm to be freezed at design time. For this reason, ASICs are typically only built to accelerate a certain aspect of CNNs, such as the partial calculation of a convolutional or fully-connected layer, but seldomly to calculate entire neural networks. A prominent exception are neuromorphic integrated circuits, which use analog electronic circuits to mimic neurons and neural networks on custom-designed ICs [79].

Besides these options for *local evaluation* of the CNN, a popular approach is to delegate the energy and resource intensive computation to remote datacenters. However, this method requires a permanent high-bandwidth network connection and introduces additional latency which might not be acceptable, e.g. in mobile, safety-relevant or real-time systems.

2.3.2 Existing CNN Implementations on Embedded Platforms

This section introduces some of the most important milestones in the field of non-GPU-powered CNN implementations, with a special focus on FPGA-based solutions.

The Design Space of Neural Network Accelerators In his mid-2015 research proposal [80], M. Drumond from EPFL Lausanne provides a survey of the design space of neural network accelerators on the platforms GPU, ASIC and FPGA. He focuses on the tradeoffs involved (in terms of energy-efficiency, flexibility and scalability) and the performance achievable. The paper provides an excellent overview of implementation options (albeit with a focus towards data center applications), and concludes that FPGAs can be much more energy efficient and scalable compared to GPUs, while maintaining a reasonable level of flexibility.

Deep Learning on FPGA: Past, Present and Future Lacey et al. also investigate the suitability of FPGAs for accelerating CNNs in their 2016 paper [81]. Besides presenting an overview of prior FPGA-based neural network accelerators, they propose to explore model-level optimizations on Convolutional Neural Networks to fully leverage the advantages of FPGAs. The paper identifies OpenCL and High-Level Synthesis as important steps towards the

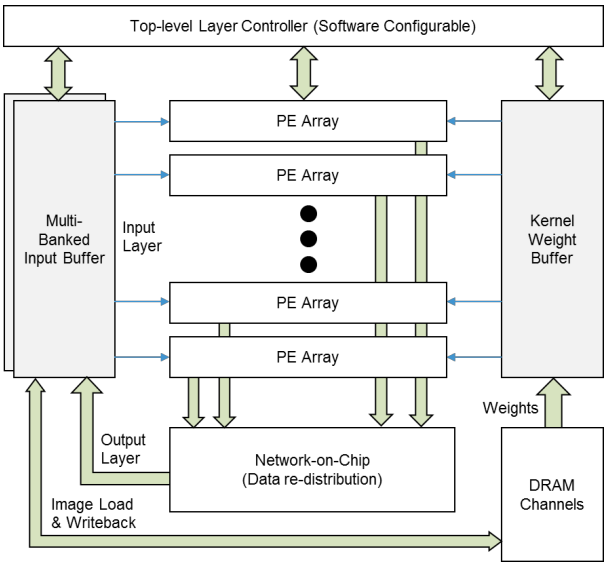


Figure 2.6.: Top-Level Overview of the FPGA-based CNN Accelerator developed by Microsoft. The architecture contains a number of generic Processing Elements as well as a Network-on-Chip, which feeds computation results back into the Input Buffer for reuse in the calculation of the next layer. [83]

widespread acceptance of FPGAs as deep learning accelerators, and suggests that datacenters would especially profit from this platform's attractive scalability and performance per watt.

Accelerating Datacenter Workloads using FPGAs Both Microsoft and Baidu seem to have come to the same conclusion, and have built FPGA-based accelerators for their datacenters. Microsoft's Catapult platform [82] (2014) was originally conceived to double the speed of the Bing ranking algorithm. It has been utilized to implement a record-breaking AlexNet accelerator in 2015 [83], achieving $^1/_2$ of the throughput of a modern GPU at $^1/_{10}$ of the power budget (fig. 2.7 depicts a top-level overview of the accelerator architecture). Chinese search giant Baidu has announced similar plans and a strategic partnership with FPGA manufacturer Altera [84]. Google also considered an FPGA-based accelerator for deep learning, but recently decided to go one step further and developed a custom ASIC solution [85].

ASIC Implementations *DaDianNao* (2014) is a multi-chip accelerator system consisting of 64 ASIC nodes with large on-chip memories to save off-chip memory accesses and thereby optimize energy efficiency. Based on their synthesis results, the authors claim up to 450× higher performance and 150× lower energy consumption with respect to a GPU implementation [86]. *Origami* (2015) is an accelerator ASIC co-developed by the author. The IC has been designed as a co-processor to speed up the computationally intensive 2D convolutions in CNNs, with a focus on minimizing external memory bandwidth and maximizing energy efficiency. The accelerator has been manufactured in 65nm technology and achieved new records in terms of area, bandwidth and power efficiency [17], [87]. An FPGA-based implementation is in progress as of 2016 [88]. Finally, *EyeRiss* (2016) is another accelerator ASIC for energy efficient evaluation of CNNs. The IC has been developed at the Massachusetts Institute of Technology and provides maximum flexibility regarding the network dimensions by using an array of 168 generic processing elements and a flexible

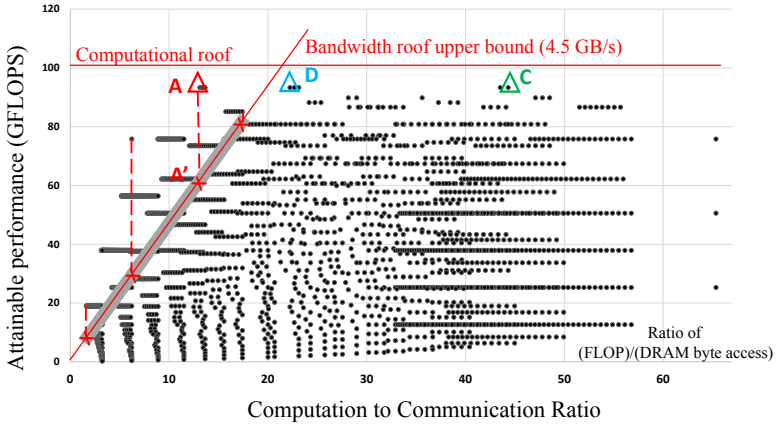


Figure 2.7.: Illustration of the Design Space Exploration and Roofline Method developed by Zhang et al. Their algorithm calculates the communication and computation requirements for a large number of implementation variants (shown as dots), draws the roof lines dictated by the platform (computational and memory bandwidth limits, red lines) and then selects the implementation with the highest throughput, yet lowest communication requirements, which still fits the platform's capacity (in this case, implementation C) [90].

network-on-chip interconnect. Additionally, this IC features run-length compression of the off-chip memory and automatic zero skipping to conserve energy [89]

Optimizing FPGA-based CNN accelerators through automated Design Space Exploration In their early-2015 paper, Zhang et al. observe that most previous FPGA-based CNN accelerators do not achieve best performance due to underutilization of either logic resources or memory bandwidth. The researchers use a polyhedral-based optimization framework to identify all legal permutations and tilings of the nested loops which form the algorithmic basis of a convolutional layer. All these potential schedules are then analyzed with respect to their memory bandwidth and computational throughput requirements. Using a roofline model (FLOPS vs. Computation-to-Communication Ratio), the accelerator with best performance and lowest memory bandwidth requirement is then selected (see fig. 2.7 for an illustration). Zhang et al. successfully implement a proof-of-concept AlexNet accelerator with Vivado High-Level Synthesis on a Xilinx Virtex-7 485T FPGA [90].[11] A very similar approach has been taken by Motamedi et al. in 2016 [91]. They identify four sources of parallelism in convolutional layers: inter-layer (independence of layers for different input images), inter-output (independence of output feature maps), inter-kernel (independence of convolutions at different image positions) and intra-kernel (independence of multiplications in convolution kernels). The authors determine the ideal combination of these sources of parallelism by enumerating the design space of possible accelerators analytically. By additionally utilizing the opportunity for tiling at kernel level, they achieve a speedup of almost 2× compared to the accelerator proposed by Zhang et al.

2.4 Project Goals and Specifications

After consideration of the prior work introduced above and the evaluation of several alternatives (e.g. the design of a binary- or ternary-valued CNN), the project goal for this

[11] At the time of publication, Zhang et al. set a new record by running inference on AlexNet at 46 FPS drawing only 18.6 W. However, Microsoft's accelerator [83] soon broke the record, reaching almost 3× the performance.

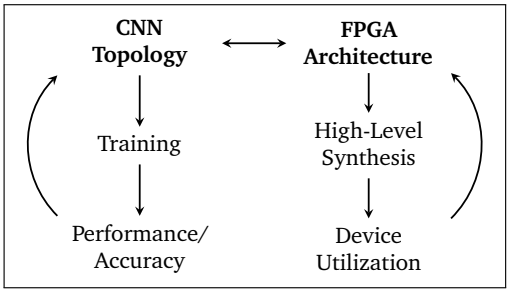


Figure 2.8.: Illustration of the System-Level Design Approach for the Project, involving Optimization of both the CNN Topology and the FPGA Accelerator Architecture.

master thesis has been defined as "[... to] build a demonstrator device that shows a convolutional neural network in operation", with focus on the optimized co-operation of the neural network and the underlying hardware platform. The hardware platform has been fixed to the *SCS Zynqbox*, an embedded systems platform based on the Xilinx Zynq XC-7Z045 System-on-Chip.[12]

Design Approach We decided to take a system-level design approach as illustrated in fig. 2.8. The emphasis has been put equally on the *design and optimization of a Convolutional Neural Network* and the *design and optimization of an FPGA-based accelerator*, with the common purpose of reaching the best possible system-level performance. This approach is different from most previous FPGA-based CNN implementations, which typically rely on a maximally flexible accelerator to run a standard CNN from research.

Project Specification The following requirements and constraints were the guidelines during the work on this project:

Primary Goal: Design and Implementation of a real-time CNN demonstrator

1. Implementation of best practices from prior work and recent research
2. Optimization of a CNN for demonstration purposes
    a) Image classification on ImageNet (realistic problem, impressive visuals)
    b) Selection and Training of a suitable CNN topology (existing or custom-built)
    c) Optimization of CNN for implementation on FPGA (resource efficiency)
    d) Optimization of CNN for accuracy
3. Elaboration of an FPGA-based architecture for the chosen CNN
    a) Based on existing Zynqbox platform (Zynq XC-7Z045 + 1GB DDR3 Memory [92])
    b) Algorithm design and block-level organization (focus on energy efficiency)
    c) Implementation using High-Level Synthesis
    d) Optimization regarding efficiency, performance and device utilization
4. Verification and evaluation of the CNN demonstrator system

The remainder of this report details the implementation of these specifications.

[12] See appendix B for the original task description.

3 Convolutional Neural Network Analysis, Training and Optimization

> *Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.*
>
> **— Antoine de Saint-Exupéry**
> (Inspiring Writer and Pioneering Aviator)

3.1 Introduction

The previous chapter introduced the two central goals of this project: The *optimization of an Image Classification CNN for ImageNet*, and the *design of a corresponding FPGA-based accelerator*. This chapter is concerned with the first of the two aspects: the analysis of existing CNN Topologies, the setup of a training platform and finally the optimization of our custom CNN architecture.

First, the CNN topologies from prior work (presented in section 2.1.3) are thoroughly examined with regard to their resource efficiency. The corresponding *Network Analysis Tools, Methods and Results* are presented in section 3.2. The following Section 3.3 then introduces the *CNN Training Hardware and Software Setup* used for the training of more than 70 different CNN variants during this project, and shares some of the lessons learned during the countless hours of CNN training. The last section 3.4 finally discusses the *Network Optimizations* that have been applied to shape our own custom-tailored Convolutional Neural Network architecture, *ZynqNet CNN*.

3.2 Convolutional Neural Network Analysis

Although research in the area of Convolutional Neural Network topologies is very active and new architectures emerge almost monthly, much of the attention seems to be currently focused on accuracy improvements, and much less on resource efficiency. During our search for an optimized CNN, the lack of tools for visualizing, analyzing and comparing CNN topologies became a serious problem. Therefore, we decided to develop the *Netscope CNN Analyzer Tool for Visualizing, Analyzing and Modifying CNN Topologies*, which is introduced in the first section 3.2.1. In section 3.2.2, we define a wish-list of desired *Characteristics of a Resource Efficient CNN Architecture*. Finally, section 3.2.3 employs the Netscope tool to analyze a number of different CNN topologies, and presents the findings from this *CNN Topology Efficiency Analysis*.

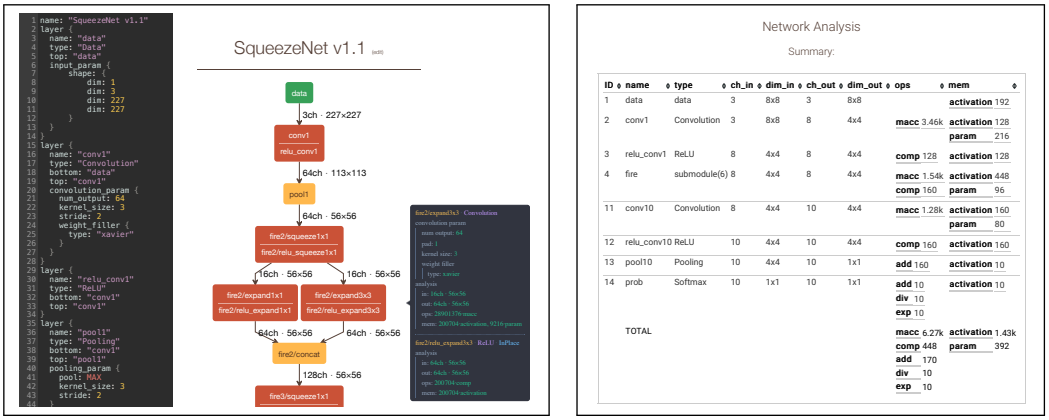


Figure 3.1.: Screenshots from the Netscope CNN Analyzer: Visualization of the layer-level Network Graph (left), Analysis Summary Table (right)

3.2.1 Netscope CNN Analyzer

The structure of Convolutional Neural Networks is inherently multi-dimensional, which makes them difficult to grasp, analyze and optimize intuitively (see fig. C.1 in the appendix for a 3D illustration of the convolutional layers in a simple 2-layer CNN module). Because no proper tools for the analysis of CNNs were available,[1] we decided to implement our own custom solution for the *Visualization, Analysis and Modification of CNN topologies*, based on an existing tool for the visualization of CNNs [93].

The *Netscope CNN Analyzer* is a web-based tool written in CoffeScript, HTML and CSS for analyzing data flows and memory requirements in CNNs, and currently has these features:

- In-browser editor for CNN descriptions with syntax highlighting
- Load Caffe `.prototxt` files from GitHub Gists, from built-in presets or by copy-paste[2]
- Visualization of the layer-level CNN structure as network graph
- Visualization of layer type, settings and dimensions[3]
- Analysis of computational complexity and resource requirements in each layer:
  - Number of operations: multiply-accumulate[4] (macc), comparison (comp), addition/subtraction (add), division (div), exponentiation (exp)
  - Memory requirements: size of output feature maps (activation), number of weight parameters (param)
- Report of analysis results in summarized and detailed table
- Report of layer characteristics in Excel-compatible format for further analysis

Figure 3.1 shows two screenshots of the user interface and appendix C.3 lists advanced usage tips as well as current restrictions. Netscope is accessible online [94] and includes presets for all the CNN topologies introduced in section 2.1.3. The full source code for Netscope is available on Github [95].

[1] Caffe includes a python script "draw_net.py", which draws the network structure but doesn't do any analysis and Excel tables tend to either explode or disintegrate after a short time.
[2] All visualizations and analyses are calculated locally, the network description never leaves the computer.
[3] Supported layer types (visualization + analysis): `DATA`, `CONVOLUTION`, `INNER_PRODUCT`, `POOLING`, `BATCHNORM`, `LRN`, `CONCAT`, `RELU`, `DROPOUT`, `SOFTMAX`, `SOFTMAX_LOSS`, `FLATTEN`, `ELTWISE`, `DECONVOLUTION`, `CROP`, `SCALE`, `IMPLICIT`.
[4] One multiplication and one addition are counted as a single MACC operation.

3.2.2 Characteristics of Resource Efficient CNNs

Convolutional Neural Networks are very demanding with respect to their computational complexity. In order to reach acceptable real-time performance with the resources available on the chosen embedded platform, a highly optimized Convolutional Neural Network architecture is mandatory. While a number of factors influence the resource efficiency of neural network topologies, the following characteristics are especially desirable:

Low Computational Complexity The Zynqbox platform constitutes a relatively small and low-power target (the Zynq XC-7Z045 has an upper bound of 468 GFLOP/s [76], which is however not realistically reachable [75]). In addition, real-time inference is one of the project objectives, and while no required frame-rate is specified in the goals, around 10 FPS might be a realistic target for many applications. This sets a hard upper bound of 23 billion MACCs per forward pass assuming a perfect accelerator, and makes especially small CNNs with low computational complexity attractive.

Regularity FPGAs are very good at processing highly parallel and regular workloads. Those can be distributed and concurrently computed on many parallel yet simple processing elements, ideally in a dataflow manner. Conditional execution of operations, data-dependent decisions and complex control sequences are better suited for other platforms. Therefore, highly regular CNNs are preferred. Problematic structures include Batch Normalization and LRN layers (where different output maps influence each other), convolutional layers with many different kernel sizes (which may need to be accelerated in different ways) and overly complex network graphs.

All-Convolutional Networks A network that consists only of convolutional layers and does not contain any fully-connected layers is called *all-convolutional*. All-convolutional networks need less memory bandwidth: while weights in convolutional layers are reused multiple times, fully-connected layers need to load a new weight for every single multiply-accumulate operation. Because the memory bandwidth in the Zynq XC-7Z045 FPGA is limited,[5] the higher computation-to-communication ratio found in all-convolutional CNNs is very welcome. Furthermore, all-convolutional networks can eliminate the need for pooling layers as shown by Springenberg et al. [96], which increases their regularity.

Accuracy Despite all these constraints, the Image Classification CNN should still deliver top quality results, and should be optimized with respect to its classification accuracy.

With this wish-list of CNN characteristics in mind, the following section looks at different CNN topologies from prior work, and judges their suitability for our embedded implementation.

3.2.3 Efficiency Analysis of Prior CNN Topologies

In search of an efficient CNN architecture, the CNN topologies from prior work introduced in section 2.1.3 have been analyzed using Netscope. Figure C.2 in the appendix shows the network graph visualizations for *AlexNet*, *Network-in-Network*, *VGG-16*, *GoogLeNet*, *ResNet-50*, *Inception v3*, *Inception-ResNet-v2* and *SqueezeNet*. A comparison of the architectures in terms of computational complexity, memory requirements and classification accuracy has

[5] The Zynqbox can access its 1 GB of shared 32-bit DDR3-1066 memory at approximately 8 GB/s [92]. Transferring the 470 MB of weights in the last three fully-connected VGG-16 layers then already requires more than 50 ms.

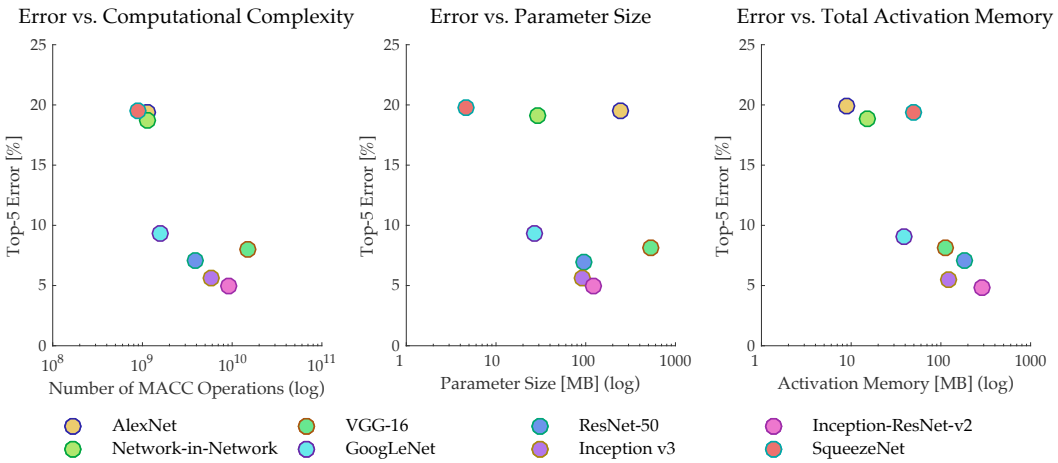


Figure 3.2.: Design Space Exploration Charts, comparing the Top-5 Error Rate to the Number of MACCs, Parameter Size and Total Activation Memory for each of the CNN Topologies from Prior Work. Parameter and Activation Memory are calculated for 32-bit weights and activations. The x-Axes are in logarithmic scale.

.

been shown in table 2.1 in section 2.1.3. The most relevant of these characteristics are visualized in the Design Space Exploration charts in fig. 3.2.

The Design Space Exploration (fig. 3.2 and table 2.1) shows that AlexNet, NiN, and SqueezeNet have the lowest computational complexities (approximately 1 billion MACCs), closely followed by GoogleNet (1.6 billion MACCs). The same four CNNs also use the least amount of Activation Memory (measured as the aggregate size of all Output Feature Maps). When looking at the number of Parameters, the clear winner is SqueezeNet, which is almost 6× smaller than GoogLeNet and 50× smaller than AlexNet. However, GoogLeNet achieves 9.2 % top-5 error while SqueezeNet has 19.7 % top-5 error.

Most state-of-the-art CNNs reach their high performance at the price of exponentially higher computational complexity and exponentially increased memory requirements (as seen by the quasi-linear distributions in the semi-log comparison graphs). VGG-16 is always in a pareto suboptimal position. AlexNet can almost always be replaced by the smaller SqueezeNet. In the Netscope topology visualization (fig. C.2 in the appendix), GoogLeNet, ResNet and the Inception variants stand out with their architectural complexity, while AlexNet, NiN, VGG-16 and SqueezeNet look relatively compact and regular. Furthermore, NiN, VGG-16 and SqueezeNet are the only networks without Batch Normalization and LRN layers.

The final choice was made for SqueezeNet as the basis for our own CNN topology, due to its good fit for an FPGA-based implementation. The tiny parameter set could even be fit into the on-chip SRAM of a medium-sized FPGA, and optimizations are relatively easy to try thanks to the fast training cycles and the clear network structure.

3.3 Convolutional Neural Network Training

With the network topology fixed to SqueezeNet, we decided to set up a training environment for this CNN. The following sections describe the *hardware and software* used (section 3.3.1), gives an introduction on *how to prepare a dataset and a CNN* for a successful training run (appendix D.5), and finishes with some *tips and tricks* learned during more than 2200 GPU-hours of CNN training (appendix D.5).

3.3.1 Construction of a GPU-based Training System

First Steps The first experiments were conducted with the *Torch* framework on a workstation with an NVidia Tesla C2075 graphics card. Torch proved surprisingly tough to install, yet very flexible and powerful to use. However, only few network descriptions were readily available online. The next Deep Learning framework installed was Caffe. Using the Caffe binaries directly proved tedious, because the training progress needs to be tracked from verbose log files and each training run has to be prepared by creating scripts, editing multiple `.prototxt` files, and starting tasks manually once the GPU becomes available.

DIGITS Training Software Many of these difficulties can be resolved by using NVidia's Deep Learning GPU Training System (DIGITS) [34], an open-source software package, which includes Caffe and Torch and can be controlled from a web-based interface. DIGITS allows the creation of datasets, the definition of CNN models, and the launch of multi-GPU training runs with extensive visual progress reports and an automatic scheduler for pending jobs. Many concurrent training runs and models can be created, compared and managed, and the weights and activations in trained CNNs can be visualized (see fig. D.2 in the appendix for screenshots of these interfaces). By accepting `.prototxt` model descriptions and internally using Caffe for the training, DIGITS retains much of the flexibility and performance while significantly simplifying the handling of multiple CNN architectures and GPUs.

Workstations for CNN Training The training performance with the NVidia Tesla C2075 graphics card soon proved to be unsatisfying, mostly because of its incompatibility with NVidia's optimized CNN libraries.[6] NVidia offers a preconfigured quad-GPU Deep Learning workstation [97] at a price of 15 000 $, but we decided to build our own dedicated workstations for CNN Training. Because the system performance during CNN training is mostly determined by the number of CUDA cores and the amount of memory available in the graphics card, we decided to use a dual-GPU setup based on the NVidia GeForce GTX Titan X, the most powerful workstation graphics card at the time. In order to exploit a dual-GPU setup, a motherboard with at least two PCIe 3.0 x16 slots running both at least in x8 mode is required. The chosen Gigabyte Z170XP-SLI motherboard would support up to 4 GPUs in parallel. Caffe fully utilizes one CPU core per training process, so at least a dual-core processor is needed, even though its performance is not critical. The Titan X GPUs both have 12 GB of graphics memory, which should at least be matched by the system's main memory.[7] A fast solid-state disk is necessary to avoid a bottleneck when loading training samples, so a 500 GB S-ATA III SSD has been chosen for the storage of models and training

[6] The *cuDNN* library contains optimized algorithms for the calculation of Pooling, ReLU, LRN, Batch Normalization and Convolutional Layers and provides substantial speedups, but requires CUDA compute capability 3.0.

[7] Memory accesses during training are mostly linear, therefore SDRAM latency and CPU Cache size are not especially important for the system performance.

data. Additionally, at least a 700 W power supply and a case provisioning reliable cooling is necessary [98], [99]. In the appendix, table D.1 lists the hardware components used in our setup, and fig. D.1 shows a photograph of the assembled workstation. Two of these dual-GPU workstations have been assembled and named *Rhea* and *Kronos* after the Titans in Greek Mythology. The setup has proven its high performance and reliability during thousands of GPU-hours already.

3.3.2 CNN Training with Caffe and DIGITS

Successfully training CNNs requires experience and is even considered "more art than science" (Matthew Zeiler, winner ILSVRC 2013 [100]). The Caffe and DIGITS installations on the CNN Training Workstations have been used to train more than 70 Convolutional Neural Network variants during this project.

Before training, a suitable dataset with training samples has to be prepared. Optionally, this dataset can be artificially enlarged using *data augmentation*, which increases the variance in the data samples and can help to improve the network performance. Next, the CNN model itself needs to be defined, and the *solver*, which is responsible for the model optimization, needs to be configured. Especially this solver configuration requires the choice of several *hyperparameters* that heavily influence the learning process. There are no unique valid settings, and intuition as well as experience are necessary to find a good combination. Section D.5 in the appendix gives an overview of the training process with DIGITS, and a number of tips and tricks for the successful training of Convolutional Neural Networks.

3.4 Network Optimization

Three types of optimizations have been applied while transforming the original SqueezeNet into ZynqNet CNN: *efficiency-related optimizations* (detailed in section 3.4.1), *FPGA-related optimizations* (introduced in section 3.4.2), and *accuracy-related* optimizations (presented in section 3.4.3).

3.4.1 Optimizations for Efficiency

The original SqueezeNet v1.0 architecture has been published in February 2016 [44] and can already be considered a highly optimized topology. Nonetheless, we have discovered some general opportunities for improvement during our experiments. Beneficial modifications have also been discovered by the authors of SqueezeNet and have led to the publication of SqueezeNet v1.1 on April 25, 2016 [101]. These two networks form the basis of ZynqNet.

Structural Analysis of SqueezeNet As shown in the Netscope visualization in fig. C.3, SqueezeNet has a relatively regular structure. The topology consists of an initial convolutional layer, eight stacked *fire modules*, and a last convolutional layer, with three max-pooling layers and a dropout layer interposed. All convolutional layers are complemented by ReLU nonlinearities, and the topology is finished with a global average-pooling layer. Each fire module consists of a *squeeze layer* (1×1 convolutions plus ReLU) and two parallel *expand layers* (1×1 and 3×3 convolutions plus ReLU). The squeeze layer has relatively few output channels and is responsible for compressing the internal representation. The expand layers evaluate both 1×1 and 3×3 kernels on this compressed feature map, and their outputs are

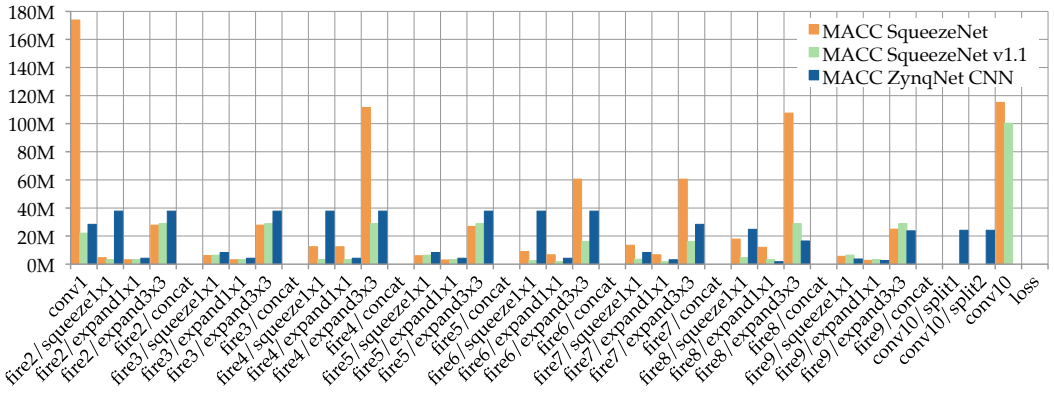


Figure 3.3.: Per-Layer Computational Complexity (Number of Multiply-Accumulate Operations) for SqueezeNet, SqueezeNet v1.1 and ZynqNet CNN.

concatenated along the channel dimension (fig. C.1 in the appendix shows a 3D illustration of a single fire module with 16 squeeze channels and 64+64 expand channels).

SqueezeNet Complexity Analysis The computational complexity of each individual layer in SqueezeNet and ZynqNet CNN has been analyzed with Netscope, and the results are visualized in fig. 3.3. The most expensive layer in SqueezeNet is *conv1*, the initial 7×7 convolutional layer (20 % of all MACC operations). The final 1×1 convolutions in *conv10* and the 3×3 convolutions in *fire4* and *fire8* (each approximately 13 %) are also disproportionately expensive.

Out-of-Sync Dimension Adjustments Figure 3.4 takes a closer look at the layer capacities $w_{\text{out}} \times h_{\text{out}} \times ch_{\text{out}}$, the layer widths w_{out} and the number of output channels ch_{out} in each stage of the network. Here we can see how the spatial output dimensions are periodically stepped down (using stride 2 in layers *conv1*, *pool1*, *pool4*, *pool8*, and using global pooling in *pool10*). The number of output channels is periodically increased, while the internal ratio of output channels in squeeze and expand layers is kept constant at $1 : 4 : 4$. However, the spatial shrinking and the channel-wise expansion in the original SqueezeNet are not ideally synchronized, and *fire4* as well as *fire8* increase the number of output channels before decreasing the pixel count, leading to a surge in computational complexity. By decreasing the spatial dimensions earlier, both SqueezeNet v1.1 and ZynqNet CNN solve this problem. The modification saves up to 40 % in activation memory and reduces the computational complexity in *fire4* and *fire8* by a factor of 3.7 and 3.9 respectively.

7×7 Convolutional Input Layer Using a convolutional input layer with a large kernel size and a large stride is typical for many CNNs (e.g. GoogLeNet, AlexNet and ResNet) and gives the network a large receptive field in the first layer. However, the large filter dimensions are computationally expensive: A 7×7 filter requires $5.4\times$ more MACC operations than a 3×3 filter. As long as the learned filters are well-behaved, a 7×7 kernel can be approximated by three stacked 3×3 kernels, which have the same receptive field, but only need $^{27}/_{49}$ of the computations. At the same time, the stacked 3×3 filters are more expressive thanks to the additional nonlinearities [102]. Interestingly, the accuracy in SqueezeNet dropped by less than 1 % when we simply replaced the 7×7 kernels in *conv1* with 3×3 kernels. Further tests were made with 5×5 and 11×11 filters, as well as combinations of multiple 3×3 conv layers such as (3×3)/16×(3×3)/16×(3×3)/96 (which increased both accuracy and training time).

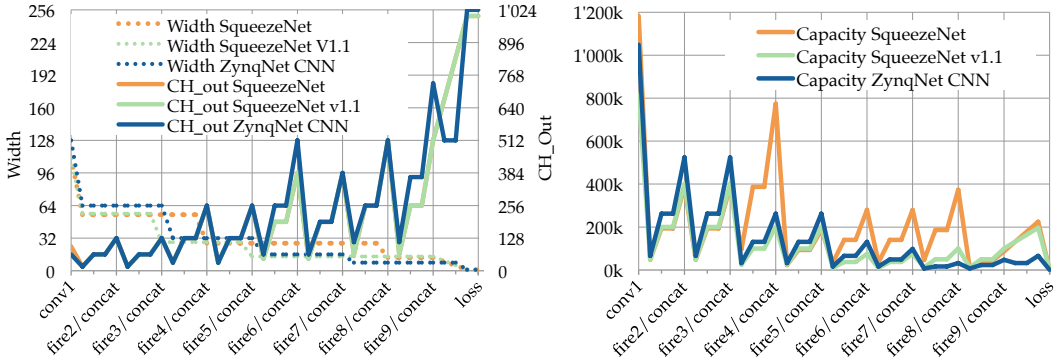


Figure 3.4.: Per-Layer Dimension Analysis of SqueezeNet, SqueezeNet v1.1 and ZynqNet CNN. Left: Layer Widths w_{out} (primary axis) and Output Channels ch_{out} (secondary axis). Because the number of output channels in SqueezeNet and SqueezeNet v1.1 is mostly equivalent, their curves overlap. Right: Layer Capacities $w_{\text{out}} \times h_{\text{out}} \times ch_{\text{out}}$.

See appendix D.3 for an overview of all experiments conducted. The final decision was made for a single 3×3 convolutional input layer with 64 output channels, similar to the one used in SqueezeNet v1.1.

Unnecessary Padding The original SqueezeNet used `pad=1` in the 1×1 conv layer *conv10*. Padding makes no sense for 1×1 kernels, and setting `pad=0` gives exactly the same results while saving 33 % of the MACC cycles in *conv10*.

3.4.2 Optimizations for FPGA Implementation

The CNN architecture has been adapted to the FPGA requirements concurrently with the work on the FPGA-based accelerator. Most changes aim to simplify the network architecture, or make sure that all layers fit into the accelerator memory.

Power-of-2 Layer Dimensions Most CNNs trained on ImageNet expect either 227×227 or 224×224 pixel images as input.[8] ZynqNet CNN however has been designed with spatial layer dimensions w and h which are a power of 2, such as 8, 16, 32, 64, 128 and 256. On the FPGA, multiplications and divisions by a power of 2 can be calculated with inexpensive shift operations, which enables optimizations in the addressing of image caches in the accelerator. The number of channels ch was initially rounded to powers of 2 as well, but the final ZynqNet CNN uses multiples of 16 instead to make better use of the available resources. The CNN architecture with all-power-of-2 dimensions required 14 % more parameters and 9 % more MACC operations, but also reached 1 % higher accuracy. The adapted CNN expects 256×256 pixel input images and is trained on a 300×300 pixel ImageNet dataset.

All-Convolutional Network In 2014, Springenberg et al. published a paper that is well-known for the introduction of the "guided backpropagation" method for visualizing CNN filters [96]. The same paper also introduced the idea of *all-convolutional networks*, which are CNNs consisting exclusively of convolutional layers and nonlinearities.[9] The authors tested

[8] The idea is to enable repeated stride-2 downscaling to integer dimensions (e.g. $224/2/2/2/2/2 = 7$). AlexNet produces non-integer intermediate dimensions no matter if input images are 227×227 or 224×224 pixels, possibly due to a missing `pad=5` in the first conv layer — reasonable padding settings help to avoid confusion.

[9] Note the distinction between *fully-connected layers* (layers where all inputs are connected to all neurons) and *all-convolutional networks* (CNNs which do *not* contain fully-connected, but only convolutional and nonlinearity layers).

networks where all max-pooling layers had been replaced by convolutional layers with stride 2, and reached state-of-the-art accuracy. Based on this idea, we removed all max-pooling layers in our CNN, and used *stride 2 in the subsequent convolutional layer*. However, all max-pooling layers in SqueezeNet are followed by 1×1 *squeeze* convolutions and stride 2 would thus waste a lot of information. We decided to increase the kernel size in these layers to 3×3 to allow for overlapping convolutions with stride 2. The resulting all-convolutional CNN has 12 % more parameters and requires 18 % more MACC operations, but also reaches 1.5 % higher accuracy. In addition, the CNN architecture is strongly unified, leaving the global average pooling as the only other layer type besides 1×1 and 3×3 convolutional layers and their ReLU nonlinearities.[10]

Layer Splitting One of the most limited resources on the FPGA is on-chip memory, which is used to hold the current layer parameters. The FPGA fabric in the Zynq XC-7Z045 contains a total of 2180 kB Block RAM memory [103], which is enough to hold approximately 560 000 32-bit floating-point parameters. However, the *conv10* layer in ZynqNet CNN has been designed with $ch_{\text{in}} = 736$ input channels and $ch_{\text{out}} = 1024$ output channels, and would therefore require $n = ch_{\text{in}} \cdot ch_{\text{out}} = 753\,664$ kernels of size 1×1. To make the layer fit onto the FPGA, it has been split into two parallel convolutional layers *conv10/split1* and *conv10/split2* with $ch_{\text{out}} = 512$, which are then concatenated along the channel dimension.[11]

3.4.3 Optimizations for Accuracy

The final type of optimizations in ZynqNet targets the classification accuracy. Multiple previous optimizations already resulted in accuracy improvements, such as replacing the max-pooling layers with 3×3 stride 2 convolutions (+1.9 %) and the power-of-2 layer dimensions (+1 %). Three additional measures are introduced in this section.

Linear Learning Rate Policy As already mentioned in appendix D.5, experiments by Mishkin et al. [104] have shown that a linear learning rate policy works best for AlexNet. They found the same to be true for SqueezeNet, which initially used a square-root learning rate policy [105]. The accuracy improvement is approximately 2 %.

Equalization of Layer Capacities Intuitively, a CNN can be understood to transform a vast amount of pixels with low individual information density into very few outputs of high abstraction level. The layer capacity $w_{\text{out}} \times h_{\text{out}} \times ch_{\text{out}}$ can be seen as a measure for this concentration of information. As shown in fig. 3.4, the layer capacities of SqueezeNet, SqueezeNet v1.1 and ZynqNet all converge from more than one million data points to just 1000 class probabilites. However, both SqueezeNet variants have intermediate capacity peaks which do not follow a smooth decline (besides the typical zig-zag pattern caused by the compression-expansion architecture that can be seen for all three CNNs). SqueezeNet v1.0 has pronounced outliers in the *fire4* and *fire8* modules, which have already been discussed as *Out-of-Sync Dimension Adjustments* in section 3.4.1. Further, both SqueezeNet versions have a strong peak in *conv10*. ZynqNet CNN follows a much smoother and more regular capacity reduction, which saves resources, but also increases accuracy by almost 2.3 %.

[10] The dropout layer only needs to be considered during training.
[11] The required facilities for the concatenation are already present from the parallel *expand* layers in each fire module.

Augmented Dataset and Extended Training Runs Data augmentation, previously mentioned in appendix D.5, has been patched into DIGITS and used for the final training runs of ZynqNet. The ImageNet dataset has been prepared as follows:

- 300×300 pixel images, 256×256 pixel crops
- 6 copies per input image (total 7.7 million training examples)
- hue modulation ($\pm 60°$, $p = 0.75$)
- contrast modulation ($0.5\times$ to $1.5\times$, $p = 0.75$)

These settings add a substantial amount of variation to the images, and were chosen to approximately emulate the reduced quality of webcam images, preparing the network for actual input images during demonstrations. In addition to the increased amount of images in the augmented dataset, the final trainings were run for 60 epochs instead of 30 epochs, effectively showing the network each image 60 times in 6 variations. This resulted in another gain of 3.1 % accuracy.

Fine-Tuning Final experiments were conducted with fine-tuning the trained model. By re-training the finalized network for a few epochs with a very low learning rate (and possibly with data augmentation turned off), sometimes a slightly better optimum can be reached. However, our best try resulted in just 0.2 % accuracy gain.

3.4.4 Final Results

Overall, the top-1 validation accuracy of our ZynqNet CNN has been increased by more than 7 % versus the initial SqueezeNet v1.0 and by more than 8 % versus the SqueezeNet v1.1 architecture.[12] The final version of ZynqNet CNN uses 2.5 million parameters, roughly twice as many as the SqueezeNet variants, but still roughly an order of magnitude less than most other CNNs. The total number of activations has been reduced by 40 %, and the number of MACC operations by 38 % with regard to the original SqueezeNet, to 530 million activations. Additionally, neither max-pooling, Batch Normalization nor LRN layers are required. The fully-trained CNN reaches a top-1 accuracy of 63.0 % and a top-5 accuracy of 84.6 % on the ImageNet validation dataset.

[12] The fact that the total accuracy improvement is less than the sum of the individual improvements indicates that some optimizations were not orthogonal and had similar effects.

4 FPGA Accelerator Design and Implementation

> *Good [HLS coding] style not only requires an understanding of the underlying hardware architecture of an algorithm, so that it is reflected in the C++ design, but also an understanding of how HLS works.*
>
> **— Mike Fingeroff**
> (High Level Synthesis Expert, Mentor Graphics)

4.1 Introduction

This chapter introduces the *ZynqNet FPGA Accelerator*, which is a purpose-built FPGA-based accelerator for the ZynqNet CNN introduced in the last chapter. After a quick overview of the *Zynqbox Platform*, a consideration of different *Data Types* for the accelerator and a formulation of the *Design Goals* in this section, the following section 4.2 introduces the *Algorithm* used for the evaluation of ZynqNet CNN, and details its *Parallelization* and the *Caching Strategy*. Section 4.3 then presents the *Hardware Architecture and Schedule*, before section 4.4 describes the *Implementation* of the FPGA accelerator and our experiences with *High-Level Synthesis*.

4.1.1 Zynqbox Platform Overview

The Zynqbox has been designed by Supercomputing Systems AG for the evaluation of high-performance image processing algorithms, especially in automotive settings. The embedded platform is based on the Xilinx Zynq-7000 All Programmable System-on-Chip (SoC), which combines a dual-core ARM Cortex-A9 processor with programmable FPGA fabric in a single device. The Zynqbox includes a Xilinx Zynq XC-7Z045 SoC, 1 GB DDR3 memory for the ARM processor, 768 MB independent DDR3 memory for the programmable logic, and plenty of connection options (Serial Camera Interfaces, USB, CAN, Gigabit Ethernet). The Kintex-7 FPGA fabric of the SoC features 350k logic cells, 218k LUTs, 2180 kB Block RAM and 900 DSP slices. The CPU runs at up to 1 GHz, boots a standard Linux operating system and is connected to the programmable logic via high-performance AXI4 ports for data exchange and control. Figure 4.1 shows a schematic overview of the Zynqbox platform [92].

4.1.2 Data Type Considerations

The choice of suitable data types is an important step when implementing designs on FPGA and ASIC platforms. This section gives a quick introduction to the possible options and justifies our choice.

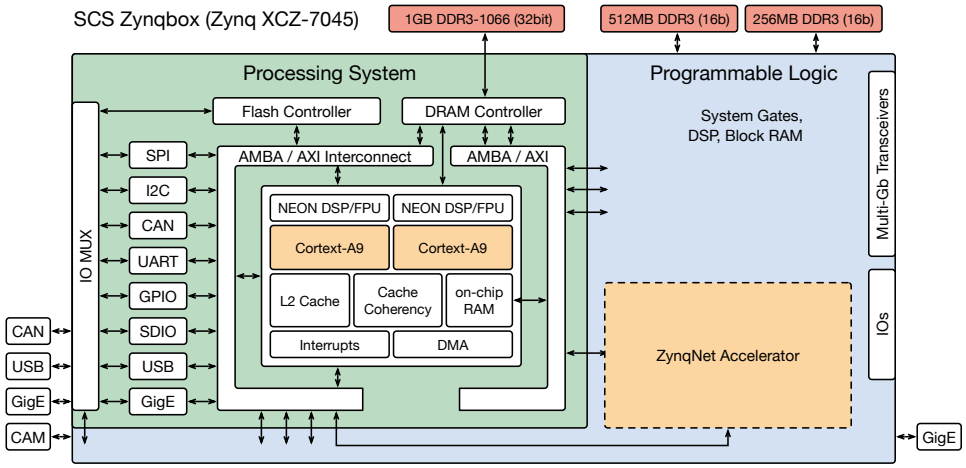


Figure 4.1.: Schematic of the SCS Zynqbox Embedded Platform based on the Xilinx Zynq XC-7Z045.

Floating-Point Data Format When real-valued numbers have to be represented in a computer, a floating-point data format is typically used. The single-precision floating-point format uses 32 bits, and stores numbers as a combination of *sign* (1 bit), *significand* (23 bit) and *exponent* (8 bit). The data format can represent all numbers from approximately -3.4×10^{38} to $+3.4 \times 10^{38}$ with at least 6 significant decimal digits, which makes it very versatile [106]. However, the computational complexity of arithmetic operations on floating-point data is high, and specialized hardware (e.g. floating-point units) are usually needed.

Fixed-Point Data Format An alternative to the floating-point format is given by the fixed-point format. A fixed-point number format can be described by the Q-format specification $Qm.f$, where m denotes the number of *integer bits* and f denotes the number of *fractional* bits. The actual fixed-point number is stored as a normal signed integer in 2's complement format, with bit-width $1 + m + f$. For example, the value 3.375 can be stored in $Q2.5$ format within 8 bits as 011.01100_2, and interpreted as $01101100_2/2^f = 108/32 = 3.375$. The format specification is required for the interpretation of the stored bits, however it is usually implicit and not stored with the value. The standard arithmetic units for integers can be used to calculate fast and efficiently with fixed-point numbers, and their range and precision can be adapted exactly according to the application's requirements.

Data Type Requirements for Convolutional Neural Networks Most CNN implementations use single-precision floating-point numbers for their weights and activations, arguably mostly because it is the standard data type on modern GPUs. As already shown in section 2.1.4 on network compression, CNNs are inherently very robust against the effects of limited numerical precision. Neither an enormous dynamic range nor very high precision are needed, and in the most extreme case even binary weights and activations can be sufficient to train and run a Convolutional Neural Network as shown by Courbariaux et al. [56], [57]

Fixed-Point versus Floating-Point on the Zynqbox Platform Today's FPGAs typically do not contain specialized floating-point hardware.[1] In the Zynq's programmable logic, each floating-point multiplication *or* addition occupies two to three DSP slices as well as hundreds of look-up tables and flipflops, and limits the clock speed to a maximum of 460 MHz [77].

[1] A notable exception are the higher-end Arria-10 and Stratix-10 FPGA series by Altera which include hardened floating-point support in each DSP block [107].

On the other hand, a fixed-point multiplication *and* addition (MACC) with 18-bit or smaller operands can be carried out by a single DSP slice at up to 750 MHz [103], [108]. As a result, the Zynq XC-7Z045 can reach up to 1500 GMACC/s in fixed-point, but only 468 GFLOP/s in floating-point [76]. An additional advantage of fixed-point numbers is their reduced memory requirement. Using 16-bit values doubles the number of weights and activations which can be stored on-chip, and even 8-bit values should be precise enough for many CNNs.

Fixed-Point Quantization with Ristretto Together with their paper from April 2016, Gysel et al. published a CNN approximation tool called *Ristretto* [58]. The application converts the weights and activations in Caffe-based Neural Networks to fixed-point format, and automatically determines the number of integer and fractional bits which are necessary to avoid serious degradation of the resulting classification accuracy (the maximum allowed accuracy loss can be configured). The authors are able to quantize SqueezeNet to 8-bit weights and activations with an accuracy drop well below 1 %.

Choice of Data Type Even though the fixed-point numbers have many advantages for an FPGA-based accelerator and can improve the quality of results significantly, we decided to use *single-precision floating-point numbers* in this project. The deciding factor was the wish to retain compatibility with the GPU-based Caffe version of ZynqNet CNN. Incorporating the fixed-point quantization would have resulted in a higher project risk, more potential points of failure and increased debugging complexity. Unfortunately, *Ristretto* had not yet been published by the time of this decision, as it might well have changed this choice by heavily reducing the risk involved in a fixed-point implementation.[2]

4.1.3 Design Goals

This project focuses on the proof-of-concept implementation of an FPGA-accelerated embedded CNN. First and foremost, the challenge in this chapter consists of

> *fitting a complete CNN for image classification on ImageNet*
> *onto the low-power Zynq XC-7Z045 with decent performance.*

Performance refers to *throughput* in this context, measured as the number of images classified per second (FPS). In order to maximize the throughput, the CNN needs to be computed as fast as possible, which implies the following design goals for the algorithm and accelerator:

- minimum number of operations and clock cycles
- minimum number of data relocations per classified image
- maximum possible clock rate
- maximum amount of parallelization and resource utilization (especially DSP Slices)

In addition to throughput, *power efficiency* is a key characteristic, because both heat dissipation and input power are typically limited in an embedded system. Power efficiency can be measured as the number of images classified per energy consumed (Images/J = FPS/W).

[2] A conversion of the current ZynqNet FPGA Accelerator implementation from floating-point format to fixed-point would not be trivial because the memory and computational resource requirements have influenced architectural decisions. Nonetheless, a conversion should be feasible and remains a very important optimization of the current architecture. Approximate results for a potential 16-bit fixed-point version are also reported in chapter 5.

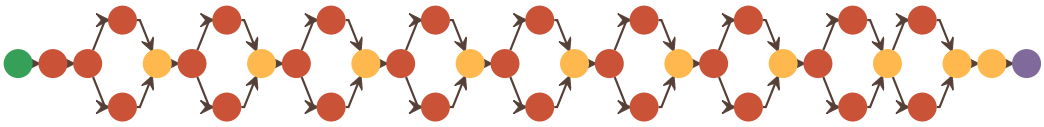


Figure 4.2.: High-Level Visualization of the ZynqNet Topology. Red dots symbolize Convolutional Layers with ReLU Nonlinearities, yellow dots Concatenations or the final Average Pooling.

4.2 Algorithm Design

4.2.1 Requirements Analysis

ZynqNet CNN, visualized in fig. 4.2 and fig. C.3, and detailed in table D.3 in the appendix, is a stripped-down version of SqueezeNet and consists exclusively of convolutional layers, ReLU nonlinearities and a global average pooling. The network is highly regular, with most layers arranged in *fire modules*. Each fire module combines three convolutional layers: a *squeeze* layer, followed by two parallel *expand* layers. The output channels of both *expand* layers are concatenated to form a single feature map with twice as many output channels. This ability to concatenate two layers is reused in convolutional layer *conv10*, which is calculated in two separate splits *conv10/split1* and *conv10/split2* to reduce the memory requirements. The dropout layer *drop9* is only relevant during training, and can be completely ignored during inference. *Pool10* reduces the spatial dimensions from 8×8 pixels to 1×1 pixel by computing the mean, while leaving the channel dimension intact. Finally, a *softmax* classifier is used to calculate the individual class probabilities.

The computational complexity in ZynqNet comes almost entirely from the 1×1 and 3×3 convolutions, which add up to 530 million MACC operations. The ReLU nonlinearities amount to 3 million comparisons. The average pooling requires 66 000 additions and one division, and the final softmax executes 1024 exponentiations, additions and divisions.

Because the exponentiations and divisions are rare, the softmax layer can be readily handled by the ARM processor. This leaves the FPGA-based accelerator with the following layer types:

- convolutional layers
  - kernel size 1×1, padding 0
  - kernel size 3×3, padding 1
  - stride 1 or stride 2
- ReLU nonlinearities
- concatenation
- global average pooling

These layer types need to be efficiently accelerated in order to successfully run the ZynqNet Embedded CNN on the FPGA.

4.2.2 Algorithmic Options

The Mathematics behind Convolutional Layers

The central operation to be accelerated is the 2D convolution of multiple input feature maps with a number of small filter kernels. The two-dimensional convolution of an input image and a filter can be intuitively understood as the result from sliding the filter over the input image, and taking the dot product between the filter and the pixels underneath at each possible filter position. For a filter of size $k \times k$, each dot product $\langle \mathbf{A}, \mathbf{B} \rangle = \sum_{i=0}^{n-1} A_i \cdot B_i = A_0 \cdot B_0 + A_1 \cdot B_1 + \cdots + A_{n-1} \cdot B_{n-1}$ requires k^2 multiplications and additions. The 2D convolution between $k \times k$ filter $\mathbf{F}$ and $H \times W$ input image $\mathbf{I}$ yields output image $\mathbf{O}$ with

$$\mathbf{O}_{(y,x)} = \sum_{j=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \sum_{i=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \mathbf{I}_{(y-j,x-i)} \cdot \mathbf{F}_{(j,i)} \tag{4.1}$$

under the assumptions that k is an odd integer and the input image is appropriately zero-padded, i.e. $\mathbf{I}_{(y,x)} = 0$ for all pixels outside of the valid image area $W \times H$. In convolutional layers there is not a single input image, but a three-dimensional stack of ch_{in} input images called *input feature maps* $\mathbf{I}^{(ci)}_{(y,x)}$. The convolutions then produce a stack of ch_{out} output images, called the *output feature maps* $\mathbf{O}^{(co)}_{(y,x)}$ by applying a bank of filters $\mathbf{F}^{(ci,co)}$. Under the above assumptions, a convolutional layer computes

$$\mathbf{O}^{(co)}_{(y,x)} = \sum_{ci=0}^{ch_{\text{in}}-1} \left(\sum_{j=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \sum_{i=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \mathbf{I}^{(ci)}_{(y-j,x-i)} \cdot \mathbf{F}^{(ci,co)}_{(j,i)} \right) = \sum_{ci=0}^{ch_{\text{in}}-1} \langle \mathbf{I}^{(ci)}_{\substack{y+\lfloor k/2 \rfloor \,\dots\, y-\lfloor k/2 \rfloor \\ x+\lfloor k/2 \rfloor \,\dots\, x-\lfloor k/2 \rfloor}}, \mathbf{F}^{(ci,co)} \rangle \tag{4.2}$$

for every output pixel (y, x) and every output channel co, which amounts to a total of $n_{\text{MACC}} = H \times W \times ch_{\text{in}} \times ch_{\text{out}} \times k^2$ multiplications and accumulations. Despite requiring a high computational effort, the mathematical operations behind convolutional layers are not complex at all, and offer a lot of opportunities for data reuse and parallelization, which will be explored in the next section.[3]

Different Approaches to Calculating 2D Convolutions

When it comes to the calculation of the convolutional layers, there are two other approaches besides the direct "sliding-filter" method described above.

Matrix Multiplication The first approach transforms the 2D convolution into one large *matrix multiplication*. For this, each local input region (the image region underneath each possible filter location) is stretched out into a column vector, and all the column vectors are concatenated to form a matrix $\mathbf{C}$. Since the filter's receptive fields usually overlap, every image pixel is replicated into multiple columns of $\mathbf{C}$. The filter weights are similarly unrolled into rows, forming the matrix $\mathbf{R}$. The 2D convolution is then equivalent to a matrix product $\mathbf{RC}$, which can be calculated very efficiently using highly optimized linear algebra (BLAS) routines which are available for CPUs, GPUs and DSPs. The disadvantage of this approach is the exploding memory consumption of the column matrix [109]. For a small 3×3 filter,

[3] Padding and strides larger than 1 add some complications, but the overall operation stays the same.

matrix $\mathbf{C}$ is already blown up by a factor of 9 compared to the original input image. This makes it necessary to split the problem into a number of overlapping tiles, and later stitch the results back together, which artificially increases the complexity of the problem. On FPGAs and in ASICs, matrix multiplications can be efficiently implemented with a *systolic architecture*. A suitable systolic array consists of a regular grid of simple, locally-connected processing units. Each of them performs one multiplication and one addition, before pushing the operands on to their neighbors. Thanks to the locality of computation, communication and memory, these architectures are very hardware-friendly [110].

Fast Fourier Transformation The second approach to 2D convolutions makes use of the fact that a convolution in the Spatial Domain corresponds to a simple element-wise multiplication in the Fourier Domain. This approach can be implemented using the *Fast Fourier Transformation* (FFT) and is especially suited for large kernels and large batch sizes, where it can provide speedups of more than to 20×compared to the matrix multiplication method [111].

Advantages of the Sliding-Filter 2D Convolution Approach Both the Matrix Multiplication and the FFT approach are well suited for general-purpose architectures such as GPUs. They are especially efficient for large problem sizes and batched computation. However, their additional memory consumption and the resulting need for tiling and re-stitching introduce artificial memory and computation requirements, which reduce the resource efficiency of the architecture. Our focus on the regular, well-optimized ZynqNet CNN further eliminates the need to support all kinds of different parameter combinations. Therefore we believe the direct 2D convolution approach as formulated in eq. (4.2) to be the most efficient way to implement an FPGA-based accelerator, regarding both memory and computational requirements. The nested summations clearly expose parallelism and translate well into nested loops in a high-level programming language, which makes the approach a good fit for High-Level Synthesis.

Algorithm Description

Based on the above considerations, a straightforward, nested-loop based formulation of 2D convolution was chosen as the foundation for this CNN accelerator. The loops are arranged in the order *layer* > *height* > *width* > *input channels* > *output channels* > *kernel elements*. For each layer, the outermost loops traverse all pixels left-to-right, top-to-bottom. At each pixel position, one input channel after the other is focused, and all corresponding output channels are calculated and accumulated.[4] Algorithm 1 gives an algorithmic formulation for the complete ZynqNet CNN in pseudo-code, including *stride* and *concatenation* facilities.

4.2.3 Parallelization

To reach a good throughput, the ZynqNet FPGA Accelerator needs to use all the computational resources available on the FPGA platform, especially the DSP slices. The computation algorithm introduced in the last section therefore needs to be parallelized.

[4] Note that the algorithm actually calculates the 2D *cross-correlation* between the filter and the input image, which corresponds to a 2D convolution with the filter mirrored at the origin. The Caffe implementation also uses this variation [112].

Algorithm 1 Nested-Loop based Computation of the All-Convolutional ZynqNet CNN.

procedure ZYNQNET(input image $\mathbf{I}$, trained weights $\mathbf{W}$, layer config)
  $in[0,\ldots] \leftarrow \mathbf{I}$
  for $L \leftarrow 0$ **to** $layers - 1$ **do** ▷ loop over all layers
    block ▷ per-layer setup: configuration and initialization
      load layer config $w_{\text{out}}, h_{\text{out}}$ ▷ output width and height
      load layer config $ch_{\text{in}}, ch_{\text{out}}$ ▷ input and output channels
      load layer config k, s ▷ kernel size ($k \times k$) and stride length
      load layer config is_1st_split ▷ flag for *expand1x1* and *split1* layers
      load layer config is_2nd_split ▷ flag for *expand3x3* and *split2* layers
      if not is_2nd_split **then**
        $out[L,\ldots] \leftarrow 0$ ▷ initialize output feature maps $out[L]$
      end if
    end block
    for $y \leftarrow 0$ **to** $h_{\text{out}} - 1$ **do** ▷ loop over y dimension
      for $x \leftarrow 0$ **to** $w_{\text{out}} - 1$ **do** ▷ loop over x dimension
        for $ci \leftarrow 0$ **to** $ch_{\text{in}} - 1$ **do** ▷ loop over input channels
          for $co \leftarrow 0$ **to** $ch_{\text{out}} - 1$ **do** ▷ loop over output channels
            block ▷ dot-product at pos. (y, x, ci) for output channel (co)
              $dotprod \leftarrow 0$
              for $j \leftarrow -\lfloor k/2 \rfloor$ **to** $\lfloor k/2 \rfloor$ **do**
                for $i \leftarrow -\lfloor k/2 \rfloor$ **to** $\lfloor k/2 \rfloor$ **do**
                  $image\_pixel = in[L, s \cdot y + j, s \cdot x + i, ci]$
                  $filter\_pixel = \mathbf{W}[L, ci, co, j, i]$
                  $dotprod = dotprod + image\_pixel \cdot filter\_pixel$
                end for
              end for
            end block
            block ▷ accumulate contributions from different input channels
              if is_2nd_split **then** ▷ concatenate to existing output channels
                $out[L, y, x, co + ch_{\text{out}}] \leftarrow out[L, y, x, co + ch_{\text{out}}] + dotprod$
              else
                $out[L, y, x, co] \leftarrow out[L, y, x, co] + dotprod$
              end if
            end block
          end for
        end for ▷ one pixel done
        for $co \leftarrow 0$ **to** $ch_{\text{out}} - 1$ **do** ▷ apply bias and ReLU to pixel (y, x, co)
          $out[L, y, x, co] \leftarrow \text{ReLU}(out[L, y, x, co] + W[L, bias, co])$
        end for
      end for
    end for ▷ one layer done
    if is_1st_split **then** ▷ second split layer will have same in and out
      $in[L+1,\ldots] = in[L,\ldots]$
      $out[L+1,\ldots] = out[L,\ldots]$
    else
      $in[L+1,\ldots] = out[L,\ldots]$
    end if
  end for ▷ all layers done
  for $co \leftarrow 0$ **to** $ch_{\text{out}} - 1$ **do** ▷ global average pooling
    $out[layers, 0, 0, co] = \sum_{y,x} in[layers, y, x, co] \cdot {}^{1}/_{(h_{\text{out}} \cdot w_{\text{out}})}$
  end for
  $\mathbf{P} = \text{softmax}(out[layers,\ldots])$ ▷ final softmax classifier
end procedure

Parallelization Opportunities The nested loops can be a source of *loop-level parallelism*: independent loop iterations can be partially or fully unrolled and executed in parallel on different processing elements. The following sources of loop-level parallelism can be exploited in the ZynqNet CNN:

- independence of *layers* when applied to different image frames
- independence of dot-products at different *pixel positions* (y, x)
- independence of *input channels* ci
- independence of *output channels* co
- independence of *intra-kernel multiplications*

The inter-layer parallelism is interesting for batched, pipelined or dataflow implementations, but not for low-latency real-time inference where we would like to finish processing the current frame before starting with the next. A second source of parallelism lies in the spatial domain. In principle, filter applications at different locations (y, x) can be computed concurrently without interdependencies.[5] When parallelizing over the input channels, multiple results are generated which stem from different input channels ci, but target the same output channel co. They must therefore be summed up, according to the summation over ci in eq. (4.2). This is very similar for parallelization over the output channels, where multiple filters $\mathbf{F}^{(ci,co)}$ can be applied to the same input channel ci to generate results for different output channels co. The results for the individual output channels need to be accumulated as well, however with the difference that this accumulation concerns each output channel separately and can happen in a distributed way. The final and most straight-forward opportunity for concurrency lies in the dot-product operation, where all multiplications can be executed in parallel. In the example of a 3×3 kernel, a speedup factor of 9 can be reached.

Parallelization Strategy We exploit intra-kernel parallelism by fully unrolling all 3×3 kernels into 9 parallel multiplications combined with an adder tree. Additionally, we make use of the independence of the output channels and partially unroll the co loop by a parametrizable factor of N_{PE}. Although many more opportunities exist, these transformations are enough to fully utilize the available computational capacity of the given FPGA platform. Furthermore, no unnecessary multiplications are executed, which makes the chosen algorithm and parallelization strategy ideal with respect to the design goals.

4.2.4 Data Reuse

Need for on-chip Caching Looking at the pseudo-code in algorithm 1, it can be seen that multiple memory locations are read and written more than once. Accesses into main memory are expensive, both in terms of latency and energy. They cannot be completely avoided because the on-chip memory is not big enough to hold all CNN parameters as well as the intermediate feature maps. However, the goal is to minimize the number of reads and writes to the external memory by maximizing on-chip data reuse. Furthermore, all unavoidable memory operations should be linear in order to facilitate *burst mode transfers*. *Caches* allow both the linearization of memory accesses, as well as the temporary storage of values that will be reused shortly after. The arrays which can profit from caching in algorithm 1 are:

[5] Note, however, that the memory access patterns for loading local receptive fields are unfavorable, unless these are stretched out beforehand, such as in the Matrix Multiplication method.

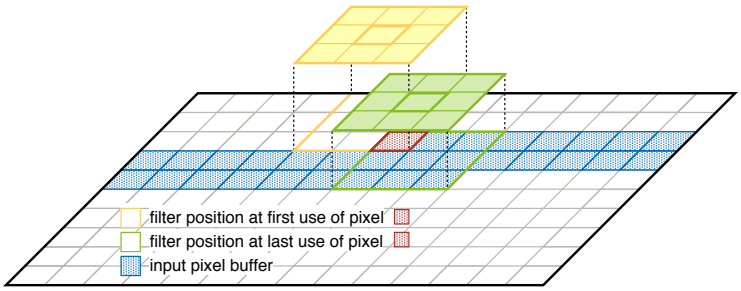


Figure 4.3.: Illustration of the Input Line Buffer, which needs to span slightly more than 2 full image lines of each input feature map to cover all reuse opportunities.

- the *input feature maps* $in[L, y, x, ci]$ (line 22)
- the *output feature maps* $out[L, y, x, co]$ (lines 30 and 32)
- the *weights memory* $\mathbf{W}[L, ci, co]$ (line 23)

The ZynqNet CNN has a maximum filter size of 3×3, therefore each input pixel (y, x, ci) is part of up to 9 local receptive fields and is used in up to 9 dot-products. Additionally, there are co output filters which are applied to the same input feature map which further increases the reuse factor by co. In the formulation of algorithm 1, each input pixel (y, x, ci) must be cached for a little more than 2 full image lines y to cover all reuse opportunities, as illustrated in fig. 4.3. The output feature maps are *read-modify-write* accessed when the output channels are accumulated in lines 30 and 32. This is a particularly inefficient memory access pattern, because latency is involved in both the read and the write transfer. However, a buffer that holds all output channels of the current pixel is enough to keep these memory accesses on-chip. A similar opportunity for caching exists during global average pooling, where the system needs to hold the accumulated mean values calculated in line 50. Finally, for each pixel (y, x), all of the current layer's weights $\mathbf{W}[L, ci, co]$ are required. These $ci \times co \times (k \times k)$ parameters should be kept local and not be fetched from main memory for each single pixel.

Caching Strategy To optimize the memory accesses, we introduce four on-chip caches.

Image Cache (ICache) is a line buffer which holds a few lines of each input feature map. The largest input image has a width of 256 pixels, and the deepest input feature maps count 736 channels. However, the ZynqNet CNN trades image width against channel depth in each layer, with the result that an image line never contains more than 8192 pixels. A little more than 2 lines need to be cached, but for simplicity and speed, a capacity of 4 lines is used in the current accelerator.[6] The ICache therefore holds 32 768 elements.

Output Cache (OCache) is a small cache used to buffer the output channels of a single pixel. Different input channels generate contributions to the same output channel, which are accumulated on the 512-element OCache. This buffer can be written back in a burst transfer when all input and output channels of a pixel have been calculated.

Global Pooling Cache (GPoolCache) is similar to the OCache, and holds the intermediate accumulation results during the global average pooling.

[6] The necessary division and modulo by 4 in the address calculation are essentially free in hardware.

Weights Cache (WCache) is the final and biggest cache. It holds all the $ci \times co$ filters of the current layer. The accelerator benefits massively from the low parameter count in ZynqNet CNN, which allows all weights to be kept on-chip. The maximum number of $384 \times 112 \times (3 \times 3) = 387\,072$ weights plus 112 bias values is required in layer *fire8/squeeze3x3*. Layer *conv10* requires a comparable number of $736 \times 512 \times (1 \times 1) = 376\,832$ parameters plus 512 bias values. Due to implementation details, the cache is implemented with a capacity of $16 \times 3 \times 1024 \times 9 = 442\,368$ elements.

These caches are sufficient to completely avoid any unnecessary accesses to the main memory. Each input pixel $in[L, y, x, ci]$ is loaded exactly once, each output pixel $out[L, y, x, co]$ is written exactly once and each weight $\mathbf{W}[L, co, ci]$ is fetched only once. In terms of main memory accesses, the chosen algorithm and caching strategy are therefore ideal.

4.3 Hardware Architecture and Schedule

Introduction The nested-loop algorithm presented in the last section 4.2 provides the basis for the computation of the ZynqNet CNN in the FPGA-based accelerator. The parallelization strategy introduced in section 4.2.3 defines which operations can be efficiently executed in parallel. And finally, the caching strategy from section 4.2.4 describes the necessary buffers to avoid unnecessary main memory accesses. With all this groundwork laid, the architecture is basically ready to be implemented in software and compiled with High-Level Synthesis. However, HLS expert Mike Fingeroff warns very early on in the introduction of his *High Level Synthesis Blue Book*, that good quality of results "requires an understanding of the underlying hardware architecture of an algorithm, so that it is reflected in the C++ design" [67]. We learned the essentiality of this statement the hard way, during multiple complete redesigns of the C++ software description. This section introduces both a hardware block diagram and a detailed schedule, which try to capture the previously described algorithm and the proposed optimizations. Although time-consuming and not strictly necessary, it is highly recommended to prepare such documents before starting with the HLS implementation to avoid even more expensive redesigns.

Algorithmic Schedule Figure 4.4 captures the nested-loop algorithm defined in the previous sections, and turns it into a detailed schedule, which also highlights the most important opportunities for *task-level parallelism* (blocks drawn in parallel and sections marked with "dataflow"), *loop-level parallelism* (loops marked with "unroll") and *pipelining* (sections marked with "pipelining").

High-Level Block Diagram Even though not strictly necessary, drawing a block diagram can help essentially to

- optimize the HLS code for the given hardware platform
- get early estimates on resource utilization and memory requirements
- get a feeling for possible bottlenecks in the design, such as high-fanout nets, large muxes or possible routing congestions
- efficiently and appropriately structure the software representation

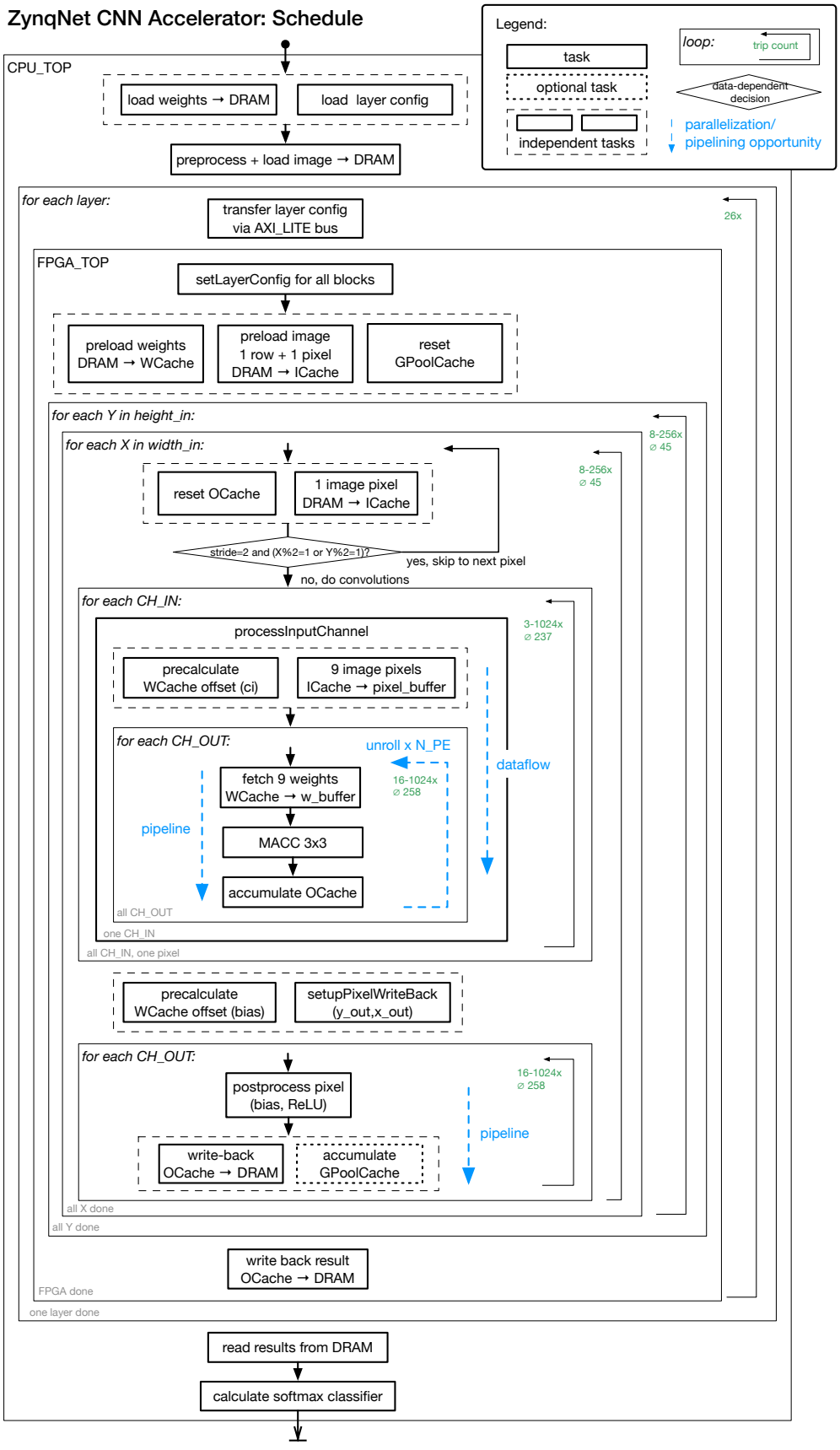


Figure 4.4.: Detailed Algorithmic Schedule for the ZynqNet FPGA Accelerator, highlighting the most important Parallelization and Pipelining Opportunities.

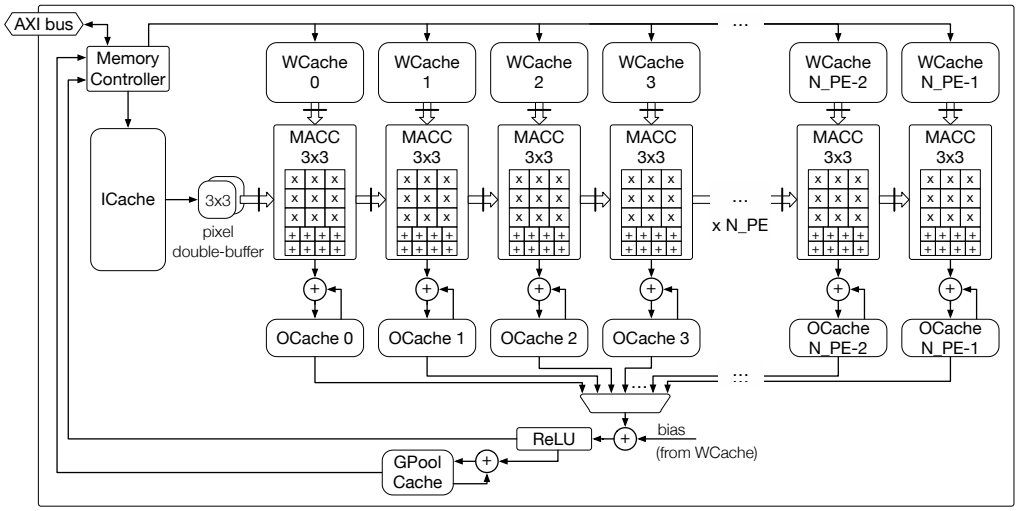


Figure 4.5.: High-Level Block Diagram of the FPGA-based Accelerator for the ZynqNet CNN.

Figure 4.5 gives an overview of the hardware organization in the final version of the FPGA-based accelerator. Another block diagram which includes the actual cache sizes and references to the C++ software implementation can be found in fig. E.1 in the appendix. Note, however, that both diagrams were manually created, and are therefore not necessarily representative for the hardware generated by Vivado HLS.

4.4 High-Level Synthesis Design Flow

ASICs and FPGA-based systems are typically planned and designed down to the finest details before implementation. Debugging those designs is very inconvenient, requires long simulation runs, and hidden bugs cannot easily be fixed later. Designers therefore start with a high-level block diagram, and then carefully refine the architecture down to the last register and state-machine before writing the first line of RTL code. High-Level Synthesis promises an alternative approach. By abstracting away many implementation details (such as the design of finite state machines, insertion of pipeline registers, definition of handshake interfaces, setup of test-benches, etc.) and handling them in the HLS compiler instead, designers can start experimenting and optimizing immediately after they have a working software implementation in C, C++ or SystemC. High-Level Synthesis is supported for the Zynq FPGAs by *Vivado HLS*, and Xilinx talks of "4× speed-up in development time" for designs using their High-Level Productivity Design Methodology [113]. Faster development time has been especially attractive considering the short time frame of this master thesis. Therefore, even though neither the author nor any of the co-workers at Supercomputing Systems AG had previous experience with *Vivado HLS*, we decided to give this promising design methodology a try.

The following sections introduce some of the experiences gained through working with Vivado HLS 2016.2, ranging from different C++ coding styles (section 4.4.2) to different ways of constraining and shaping the synthesis (section 4.4.3). We also report some of the difficulties and limitations encountered in the current generation of VHLS in section 4.4.4.

4.4.1 Introduction to Vivado HLS

Vivado HLS, abbreviated VHLS and formerly known as AutoESL, is a High-Level Synthesis tool for Xilinx FPGAs. It is a standalone application based on the Eclipse development environment. VHLS can be used to write and debug C, C++ and SystemC specifications of FPGA designs. Its most important component is the *HLS compiler*, which analyzes and translates the high-level code into an intermediate low-level representation of all the necessary instructions to run the program. It then optimizes and parallelizes these instructions into a *schedule* and a *resource allocation* scheme, and generates suitable RTL code in either Verilog or VHDL.

A very important capability of HLS software is the automated *verification* of the generated RTL code, which enables designers to use the original high-level software specification as a *test-bench*. If the software model includes enough test cases, and the automated verification passes, the generated RTL code can be assumed to be correct. This feature works by means of *co-simulation* in Vivado HLS: In a first step, the software model is executed and all input and output data consumed and generated by the function-to-be-synthesized are recorded. Then the RTL code is simulated with the recorded data as input stimuli, and the output from both the software model and the RTL simulation are compared. All data values consumed and emitted must match in order for the co-simulation to be successful.

A unique characteristic of high-level synthesis with C and C++ is the complete absence of the concepts of timing and clock cycles in the software specification (which can be both a curse and a blessing, as further explained in section 4.4.4 on the limitations of the HLS approach). The HLS design is constrained, shaped and optimized using a number of *compiler directives* (either as in-code *pragmas* or using a separate TCL-based script). The directives can be used to specify the implementation and partitioning of memories, the unrolling of loops, function-level pipelining, etc. More details on compiler directives follow in section 4.4.3.

Vivado HLS also has an astonishingly good support for object-oriented C++. There is full support for C++ classes, private and public member variables, and even (compile-time resolvable) inheritance.[7] Pointers and even double-pointers are also supported, albeit with some limitations: Pointers can only be casted between native C types, arrays of pointers may only point to scalars or arrays of scalars, and all functions which use a double-pointer are inlined together [63].

After each synthesis run, Vivado HLS estimates the device utilization and the maximum achievable clock frequency of the design. The tool also provides a number of different analysis views that visualize the resources allocated for each code section as well as the exact schedules for each loop and function.

Handing the tedious process of writing register transfer level code off to the compiler can heavily speed up the development of FPGA designs. Xilinx talks about average 4× speed gains in the development of new components, and speed gains of up to 10× when adapting previous designs, while reaching between 70 % to 120 % of the quality of results with respect to hand-coded RTL [113]. This speedup, combined with a more agile development style and increased flexibility are especially important with regard to the ever-growing design complexities and the increasing capacities of newer FPGA generations.

[7] Note, however, that the top-level function has to be a plain global function.

4.4.2 Coding Style

The *Vivado Design Suite User Guide for High-Level Synthesis* (UG902, [63]) is the most important document when working with Vivado HLS. It states that HLS enables designers to "work at a level that is abstract from the implementation details, which consume development time" and "create specific high-performance hardware implementations" by "controlling the C synthesis process through optimization directives", which sounds almost too good to be true.

The *High-Level Synthesis Blue Book* [67] by Mike Fingeroff, HLS expert at Mentor Graphics, is another major resource for guidelines regarding the design with High-Level Synthesis. While also highlighting the benefits of the higher level of abstraction and the dramatic improvements made in the last years, the author also notes that "there is still the potential for ending up with poor quality RTL when the C++ is not well written" and that "good style not only requires an understanding of the underlying hardware architecture of an algorithm, so that it is reflected in the C++ design, but also an understanding of how HLS works." In his book, he advocates an astonishingly "low-level" design style, which tries to directly mimick individual registers, muxes and arithmetic operations down to bit level in the C code – something which is arguably not very "abstract from the implementation details" of the algorithm.

Of course, HLS compilers get better at understanding different algorithms and coding styles with every new version, and their coming can be compared to the rise of logic synthesizers which required very specific description styles initially, and are capable of generating relatively efficient logic from almost any specification today. One key aspect is to accept a design that may not be a *perfect* solution, but which does the job *well enough* [67]. With RTL designs, the goal was often to optimize an architecture down to the last bit. This may no longer be feasible with HLS – but maybe also no longer necessary, thanks to the abundance of resources available in modern FPGAs.

Initially, we found it very difficult to find a satisfactory coding style, given the short code examples and the different styles used in the two HLS guidelines. This section describes the experiments made and reports their successes and failures.[8]

Unstructured, Monolithic

The first software implementation of the ZynqNet FPGA Accelerator algorithm was designed to be as simple as possible:

- straightforward implementation of algorithm 1
- 7 nested loops (layers, height, width, input channels, output channels, kernel y and x)
- all arrays and most variables declared as global

This seemed like an adequate representation of the FPGA hardware (where all resources necessarily exist "globally") with minimum complexity of the control flow (no function calls, no handshakes, just a number of interconnected counters for the loop indices). The C++ software model eventually compiled and worked well. However, the HLS synthesis got stuck in the middle of the *design analysis* phase without any error or indication of what was

[8] More code examples can also be found in the Vivado High-Level Synthesis Tutorials (UG871) [114].

currently being analyzed. Many variants and changes to the code were tried without success, and overall this coding style proved to be:

- hard to constrain using pragmas (scopes of application are not well defined)
- hard to read and maintain due to the unstructured "spaghetti code" nature
- very hard to debug due to the lack of indicative error messages

Object-Oriented

Having learned from the previous approach, the software model was rewritten from scratch. The new version included extensive testing, logging, debugging and verification facilities. Additionally, a conversion tool for `.prototxt` network description files and `.caffemodel` weight files as well as support for strided convolution and on-the-fly padding were added. This enables a very easy adaptation and reconfiguration of the architecture for different network topologies. This time, the accelerator core was written in an object-oriented manner:

- hardware blocks modeled as class instances (MemoryController, ImageCache, WeightsCache, OutputCache, ProcessingElement)
- arrays and variables encapsulated as private class members
- data movement via high-level member functions (`data_t ICache::getPixel(y,x,ci)`, `void PE::macc2D(data_t pixels[9], data_t weights[9], data_t& result)...`)
- control flow still via nested loops in top-level function (layer, height, width, input channels) and inside `class ProcessingElement` (output channel, kernel y and x)

This coding style gave better results and was more pleasant to work with. By splitting the code into separate functional units, problems during synthesis became easier to trace and isolate. The usage of compiler directives was simplified, because the lexical scopes to which the directives apply now coincide with functional blocks (e.g. pipelining can be explicitly applied to the postprocessing function and to the pixel writeback loop).

However, it was still not possible to complete synthesis. This time we experienced multiple fatal crashes of the HLS compiler process during the *RTL generation* phase. Closer inspection suggested that the compiler automatically inlined multiple hierarchical levels of function calls in order to avoid double or even triple pointers, and tripped somewhere in that process. Double and triple pointers are very easily created in object-oriented code. For example, assume a class `ProcessingElement`, which includes a reference to an instance of another class `WeightsCache *ProcessingElement::WCache`, which *itself* contains an array `data_t WeightsCache::BRAM[]` (the variable `BRAM` may be hidden behind an interface `data_t WeightsCache::read(int addr)`, but due to various reasons, this type of functions tend to be inlined by Vivado HLS). `BRAM` is then accessed as `this->WCache->BRAM` from inside class `ProcessingElement` (double pointer), and is itself a pointer to elements of type `data_t` (triple pointer). The HLS compiler tries to avoid these double and triple pointers, and may for example try to inline the whole instance `WCache` into `ProcessingElement`, but this quickly gets messy (imagine for example that a member function of another class *also* accesses the `BRAM` array). Therefore, the object-oriented coding style had mixed success:

- much easier to read, modify and debug

- much easier to apply compiler directives
- still no successful synthesis (triple pointers due to class instances)

Block-Structured

The third approach was finally designed to be a compromise between the flat spaghetti code approach and the fully hierarchical object-oriented approach. This coding style uses *namespaces* to structure the code conceptually, while avoiding the need for references and pointers. With namespaces, modular and object-centric code can be written (such as `data_t px = ImageCache::getPixel(y,x,ci)` or `OutputCache::reset()`), but the actual hierarchy stays flat (when `OutputCache` is simply a namespace and not an object, no references or pointers to it are needed to access `data_t OutputCache::BRAM[]`). The software model for the ZynqNet FPGA Accelerator has been partially rewritten to fit this *namespace-based* or *block-structured* coding style:

- use namespaces to structure code into modules
- arrays and variables are encapsulated in namespace-scopes
- data movement is done via high-level namespace-scoped functions
- control flow still via nested loops in top-level function (layer, height, width, input channel) and inside `namespace ProcessingElement` (output channel, kernel y and x)

This approach worked very well, except for one flaw: Global pointers are not supported for synthesis in Vivado HLS, and variables defined within namespaces are also considered global. Therefore, the declaration of a pointer into main memory via `data_t* MemoryController::SHARED_DRAM` is not synthesizable, and accesses into main memory can not be properly hidden behind an interface (such as `data_t MemoryController::loadNextWeight()`). Instead the pointer into main memory (which comes in as an argument to the top-level function) has to be dragged through all affected functions as an additional argument (such as `data_t MemoryController::loadNextWeight(data_t* SHARED_DRAM)`, and therefore also `WeightsCache::loadFromDRAM(data_t *SHARED_DRAM)`). While this solution is not very elegant, it works and this last coding style finally resulted in a synthesizeable design. The *namespace-based* coding style combines the advantages of both previous attempts, and we would describe our next HLS design in this coding style again:

- straightforward, close to hardware description
- easy to read, modify and debug code
- easy to apply compiler directives

4.4.3 Compiler Directives

The high-level languages C and C++ by themselves do not allow the designer to specify concurrency in the code. Frameworks which enable the explicit parallelization of C and C++ programs typically use either the concept of *kernels* or *threads* which are launched in parallel (e.g. CUDA, OpenCL or Pthreads), or they allow designers to annotate the source code with *compiler directives* that specify the desired type of parallelization (e.g. OpenMP).

As already indicated earlier, Vivado HLS uses this second approach and supports the annotation of the high-level source code using *#pragma* directives.[9] The compiler directives affect all code in the lexical scope in which they have been placed (such as a function, loop, or the branch of an if-clause), and can influence e.g. the synthesis of FPGA memories from arrays, the derivation of control and data flows, and the parallelization and pipelining of individual code sections. However, in comparison to directly writing RTL code where the structure and timing of the design can be exactly controlled, shaping an architecture using compiler directives can feel more like trying to thread a needle while wearing fireproof gloves.

In this section we introduce the most important *#pragma HLS* compiler directives which have been used for the ZynqNet FPGA Accelerator.

Interfaces

Vivado HLS usually synthesizes C/C++ functions into different functional entities. All blocks automatically receive clock and reset ports (`ap_clk`, `ap_rst`). The function arguments are turned into RTL ports of different types. The compiler directive *#pragma HLS INTERFACE <mode> [register] [depth=<D>] port=<P>* allows the specification of the *function-level interface protocol* and the *port-level interface protocol* for each argument.

Function-Level Interface Protocols The function-level interface protocol is set by applying the *#pragma* to the `return` port. The choices are `ap_none`, `ap_ctrl_hs` and `ap_ctrl_chain`, where the *handshake* protocol `ap_ctrl_hs` is the default and creates `ap_start`, `ap_done`, `ap_ready` and `ap_idle` signals which let the blocks negotiate data transfers.

Port-Level Interface Protocols When the *#pragma* is applied to individual arguments to set the port-level interface protocol, there are many modes available depending on the type of argument, and both inputs and outputs can be automatically registered. `ap_none` is the default mode for scalar pass-by-value and pass-by-reference inputs and corresponds to a simple wire. The `ap_vld`, `ap_ack`, `ap_ovld` and `ap_hs` modes add increasingly complex handshake protocols to the individual signals, with the *output-valid* protocol `ap_ovld` being standard for pass-by-reference outputs. Arrays on the function interface are normally synthesized into `ap_memory` ports, which creates *data, address* and *RAM control* ports. Alternatively, the port can be turned into an `ap_fifo` interface if the access patterns correspond to a first-in-first-out buffer behavior.

AXI4 Interfaces On the top-level, Vivado HLS also supports the `axis` (AXI4-Stream), `m_axi` (AXI4-Master) and `s_axilite` (AXI4-Lite) interfaces which strongly simplify the connection of the design into a larger system architecture. This project uses the AXI4-Master interface to connect to the main memory via the AXI4 bus in the Zynq XC-7Z045. An AXI4-Lite interface is used for configuring, starting and stopping the accelerator. Vivado automatically generates C/C++ driver files for accessing the AXI4-Lite ports from software running either on the Zynq's ARM cores or on Soft Processor Cores in the FPGA fabric.

[9] Alternatively, TCL-based scripts can be used, which allows a separation of the optimization directives and the code. The scripts support the same compiler directives, but have not been used in this project.

AXI4 Depth Settings When proceeding to the co-simulation, it is crucial to set the `depth` of the AXI4-Master ports correctly (i.e. to the exact number of elements in the array connected to this port on the test-bench side). Setting the depth too small results in the simulation getting stuck. Setting the depth too large results in ambiguous error messages or even segmentation faults in Vivado HLS 2016.2. The depth can also be passed to the *#pragma* using a `const int` variable in C++.

Data and Control Flow

There are a number of *#pragma* directives that affect the control flow in hardware, and are therefore very important for parallelizing algorithms.

Loop Unrolling *#pragma HLS UNROLL [factor=<N>]* instructs the compiler to unroll the loop in which the *#pragma* is placed, either completely or partially by a factor of `N`. Because Vivado HLS by default schedules all operations as soon as they are ready for execution, these unrolled iterations are then executed in parallel. Of course, unrolling only works if there are no dependencies between the loop iterations, and complete unrolling requires known loop bounds at compile time. Besides the opportunity for parallel execution of the loop body, unrolling also removes the loop entry and exit overhead, which otherwise adds two clock cycles to every iteration.

Dependencies *#pragma HLS DEPENDENCE variable=<var> <intra|inter> [false]* allows to override the automatic (and relatively conservative) dependency analysis. This directive needs to be applied when loops cannot be unrolled because a (false) inter-iteration dependency is detected by the compiler. For example, a loop which executes a read-modify-write operation on every individual element of an array cannot be unrolled by default, because the compiler sees read-after-write operations on the same array variable. However, the designer knows that the operations target different elements in the array in every loop cycle, and can therefore assert a *false dependency* to re-enable loop unrolling.

Loop and Function Pipelining *#pragma HLS PIPELINE [II=<N>]* is a very important optimization directive for loops as well as for functions. This *#pragma* enables pipelining for the context in which it is placed, and for all entities in the hierarchy below. Vivado tries to build a pipelined design with an initiation interval `II=<N>` (default: `N=1`), which means that a new data element is accepted into the pipeline every `N` clock cycles. The necessary depth of the pipeline (and the corresponding latency) are automatically determined by the compiler. An important caveat is the fact that pipelining *forces all loops in the hierarchy below to be fully unrolled*. Full unrolling requires fixed loop bounds, and therefore this requirement can often prevent the pipelining of higher-level loops and functions, even if the lower-level loops are themselves pipelined and would be fully compatible with e.g. `II=1`.

Resource Specification and Pipelining of Arithmetic Operations *#pragma HLS RESOURCE variable=<var> core=<string> [latency=<N>]* specifies the resource (core) that should be used to implement variable `var` in the RTL. This can be useful to select a certain type of memory for an array (e.g. dual-ported block RAM `RAM_2P_BRAM` or single-ported distributed ROM `ROM_1P_LUTRAM`), but it is also very useful to pipeline arithmetic operations:

```
int sum = a + b;
int product = a * b;
#pragma HLS RESOURCE variable=sum core=AddSubnS latency=2
#pragma HLS RESOURCE variable=product core=MulnS latency=4
```

This code instructs Vivado HLS to use a pipelined `AddSub` block with 2 register stages for the addition, and a multiplier with 4 pipeline stages for the multiplication. Pipelining arithmetic operations like this can be very useful to resolve problems with slow paths, and complements *#pragma HLS PIPELINE*.

Function Inlining *#pragma HLS INLINE* forces a function to be inlined into all its callers, which effectively creates copies and additional hardware, and thereby avoids the overhead of the function-level handshake (which is typically around 2 to 3 clock cycles). Vivado HLS often inlines functions automatically, e.g. to increase throughput. This can be prohibited by specifying *#pragma HLS INLINE off*.

Function Instantiation *#pragma HLS FUNCTION_INSTANTIATE variable=<arg>* also creates multiple copies of the function in which it is placed, one for each value that the function argument `<arg>` takes on during execution. In contrast to *inlining*, this *#pragma* keeps the function hierarchy intact. It allows the specialization of each function instance for a fixed value of `arg`. This is an important *#pragma* in combination with the parallelization of array accesses. Consider the following code example:

```
int readArray(int block, int idx) {
  return array[block][idx];
}
for (int i = 0; i < N; i++) {
  readArray(i%4, i/4);        // -> sequential array access
}                             //    [0][0],[1][0],[2][0],[3][0],[0][1],[1][1],...
```

Assuming that `array` has enough read ports, the loop could be partially unrolled to allow parallel read accesses. However, this is prevented because the function `readArray` can only be called sequentially. Adding the function instantiation directive creates four copies of `readArray`, and unrolling by a factor of 4 becomes possible:

```
int readArray(int block, int idx) { // instances: readArray_{0,1,2,3}
  #pragma HLS FUNCTION_INSTANTIATE variable=block
  return array[block][idx];
}
for (int i = 0; i < N; i++) {
  #pragma HLS UNROLL factor=4
  readArray(i%4, i/4);        // -> parallel array access
}                             //    [0,1,2,3][0],[0,1,2,3][1], ...
```

Dataflow *#pragma HLS DATAFLOW* activates the dataflow optimization used for task-level parallelism in Vivado HLS. By default, the compiler always tries to minimize latency and improve concurrency by scheduling operations as soon as possible. Data dependencies limit

Listing 1 Example Code using the *Dataflow* Compiler Directive for Task-Level Parallelism.

```
void ProcessingElement::processInputChannel(const coordinate_t y,
                                            const coordinate_t x,
                                            const channel_t ci,
                                            const channel_t num_ch_out) {
#pragma HLS INLINE off

  // Dataflow Channels:
  weightaddr_t ci_offset;  // precalculated offset into WCache
  data_t px_buffer[9];     // double-buffer for preloaded pixels (reg.
   ↪  file)
#pragma HLS ARRAY_PARTITION variable=pixel_buffer complete dim=0

#pragma HLS DATAFLOW

  // Task 1: Preload Image Pixel Buffer (fetch pixels around (y,x,ci))
  //         and precalculate ci-dependent offset into Weights Cache
  preloadPixelsAndPrecalcCIoffset(y, x, ci, num_ch_out, ci_offset,
   ↪  px_buffer);

  // Task 2: MACC All Output Channels on Preloaded Pixels
  processAllCHout(num_ch_out, ci, ci_offset, px_buffer);
}
```

this type of parallelism: By default, a process A must finish all write accesses to an array before it is considered finished and a second process B can start consuming the data.

By adding the *dataflow* directive, Vivado HLS analyzes which data elements are produced and consumed in the individual processes within the directive's scope, and tries to create channels (double-buffer/pingpong RAMs or FIFOs) between producer and consumer loops or functions. These allow data elements to be exchanged as soon as they are ready. However, there are multiple restrictions: Only single-producer single-consumer schemes are allowed, blocks cannot be bypassed or conditionally executed, and feedback between tasks is not supported. Further, *dataflows* cannot be created within loops with variable bounds or with multiple exit conditions.

The dataflow *#pragma* is used in this project to allow simultaneous prefetching of a new image patch, while the filters are applied to the previously fetched image patch. Example code for this scenario can be seen in listing 1. All dependencies between the two tasks should be made explicit via function arguments (i.e. exchanging data between the two blocks via class member variables does not work reliably).

Latency *#pragma HLS LATENCY [min=<int>] [max=<int>]* specifies a minimum and/or maximum latency for a certain code segment (such as a function, loop iteration, etc.) Specifying a high minimum latency for uncritical code sections can reduce the resource consumption and increase sharing. Specifying a low maximum latency causes Vivado HLS to increase its scheduling effort to achieve the target. This directive can be especially useful to relax the latency constraints in short blocks and increase the scheduling effort in larger tasks of a dataflow pipeline.

Array Synthesis

Memory Type and Style Memories in the FPGA hardware are described as `arrays` in the high-level source code. Only statically declared arrays with a fixed size are supported for synthesis. The mapping between the C/C++ arrays and the underlying hardware can be influenced with a number of compiler directives. The memory type (RAM, ROM, FIFO) and implementation style (Block RAM, Distributed RAM, Shift Register) can be chosen by using the previously introduced *#pragma HLS RESOURCE* directive.

Array Partitioning *#pragma HLS ARRAY_PARTITION variable=<var> <block, cyclic, complete> [factor=<int>] [dim=<int>]* is the most important compiler directive with regard to memory synthesis. By default, arrays are mapped linearly to Block or Distributed RAM (depending on their size) and receive either one or two access ports (two if it helps to reduce latency). The *array partitioning* directive allows an array `<var>` to be split into sub-arrays along different dimensions, which results in additional read and write ports. These enable concurrent access to array elements of the different sub-arrays, and thereby increase the overall parallelizability of the design. *#pragma HLS ARRAY_PARTITION* is especially important when *load* or *store conflicts* prevent the optimization of the design. Figure 4.6 illustrates the partitioning of a one-dimensional array by `factor=2` in the *block* and *cyclic* modes, as well as *complete partitioning*. For multi-dimensional arrays, `dim` specifies the dimension to be partitioned. If an array is partitioned with `dim=0` and mode `complete`, it is fully disassembled in all dimensions and a register field is inferred. Unfortunately, the *#pragma* cannot be applied repeatedly to the same dimension to create more complex array partitions. It is therefore recommended to already structure the arrays in C/C++ code along multiple dimensions, and then split the required dimensions into separate memories using *complete partitioning*. The following code example partitions the weights cache memory `WBRAM` in multiple dimensions:

```
// Weights BRAM Config for ZynqNet CNN:
//          [ 16 ][       3        ][    1024  ][9] x 32bit
data_t WBRAM[N_PE][NUM_BRAMS_PER_PE][BLOCK_SIZE][9];

// Array Partitioning (dimensions indexed from 1)
#pragma HLS ARRAY_PARTITION variable=WBRAM complete dim=1 // PE ID
#pragma HLS ARRAY_PARTITION variable=WBRAM complete dim=2 // block ID
#pragma HLS ARRAY_PARTITION variable=WBRAM complete dim=4 // weight ID
#pragma HLS RESOURCE variable=WBRAM core=RAM_S2P_BRAM latency=3
```

4.4.4 Limitations and Problems

Considering the enormous complexity of the transformation of high-level sequential code into optimized FPGA hardware, Vivado HLS does an impressively good job. However, the road is still full of bumps – designers should not expect a smooth ride, especially for larger and more complicated designs. This section highlights a few of the most relevant problems and limitations that we encountered while working with High-Level Synthesis.

Global Pointer and Multi-Pointer Support As explained earlier, the fact that global pointers are not supported in Vivado HLS prevented the abstraction of the main memory

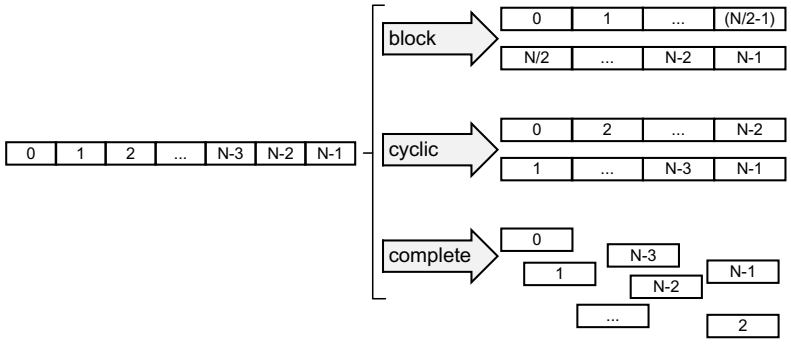


Figure 4.6.: Example of different Array Partitioning Modes in Vivado HLS (Illustration from [63]).

interface. While the problem can be circumvented by including a pointer to the shared memory in all of the concerned function's arguments, the solution is cumbersome and inelegant. Very similar problems occur when the pointer to the shared memory is stored in a class, this time due to multi-pointers which result when accessing these classes.

Unsupported Pointer Casting to Custom Data Types Because pointer casting is only allowed between native data types (`char`, `int`, `float`, ...) and not from and to custom data types (including `typedef` aliases and `struct`), loading such variables and especially structures over a memory bus is very complicated. Interpreting a structure `struct { char c; float f; } S;` as an array of bytes or integers is straightforward in C and C++: `char *B = (char*) &S; uint32_t *U = (uint32_t*) &S;`. In RTL code, it is just as simple to reconstruct the original data by bit-slicing and reassembling the incoming words. This combination would make it very easy to transfer arbitrary structures over a 32-bit AXI4-Master bus. Unfortunately, the necessary pointer reinterpretation (`(uint32_t*) &S`) is unsupported in the current version of VHLS, and tedious, error-prone manual transformations between the custom types or structures and the bus data type are necessary, for example using *unions*:[10]

```
union { custom_t custom; uint32_t bus; } U;
U.custom = ...; bus_transfer(U.bus);      // custom-to-bus
U.bus = bus_receive(); ... = U.custom;    // bus-to-custom
```

Imperfect Code Analysis While the HLS compiler mostly does a good job at interpreting the high-level source code, it sometimes misses very important code optimizations. Consider the following example which describes the wrapping logic of a counter:

```
a++;
if (a == MAX) a = 0;
```

```
if (a==MAX-1) a = 0;
else          a = a + 1;
```

The left version is an intuitive description in C++. The right version is a more verbose description of the same logic. While both pieces of code have the exact same functionality,[11] the left version takes two clock cycles to execute when synthesized, while the right version takes only one clock cycle. Manual low-level optimizations can therefore still be necessary in unexpected places.

[10] This strategy has been used to transfer custom integer data types via a `float` AXI4-Master bus. Note, however, that this only works for custom types which are smaller or equal to the width of the bus data type.
[11] Assuming that `a` is not declared `volatile`.

Difficulties with Compiler Directives Constraining the design with compiler directives alone can be difficult. The `#pragma` directives are not binding, and are sometimes interpreted by the HLS compiler rather as suggestions than imperatives. Furthermore, when multiple directives are combined, the result depends on the order of the `#pragma` commands as well as the order in which the directives are encountered by the compiler.

Pipelining Requires Loop Unrolling As already mentioned above in section 4.4.3, the compiler directive `#pragma HLS PIPELINE` requires all loops in the function hierarchy below to be fully unrolled. The rationale is probably that VHLS can only build one giant pipeline with a single initiation interval in the current version. However, it would often be convenient to build pipelines while retaining the loop hierarchy: For example, two outer loops could be responsible for iterating over pixels, precalculating some values, and then feeding an additional *inner loop* which does the heavyweight calculations. It is currently possible to pipeline this inner loop. However, the outer loops and therewith the whole outer loop entry and exit logic, the precalculations, etc. cannot be pipelined without fully unrolling the inner loop. In a more ideal scenario, it would be possible to create an outer pipeline which feeds the inner pipeline, where both have independent initiation intervals and both can be specified with a simple `#pragma` directive.

Pipeline Flushing Issue for Pipelines Nested in Dataflow Similar to the previous issue, Vivado HLS currently has a serious limitation when it comes to the combination of `dataflow` and `pipeline` compiler directives: An inner pipeline (`L_INNER`) that is part of a `dataflow` scheme, which itself is placed inside an outer loop (`L_OUTER`), is unnecessarily *flushed* in every iteration of `L_OUTER`. Listing 2 illustrates this configuration using a minimal example. This issue is present in the current implementation of the elerator and heavily degrades its performance. Xilinx has acknowledged the problem and currently cannot offer a workaround or a solution. The only recommended workaround is to avoid High-Level Synthesis altogether and rewrite the architecture as RTL code [115].

Slow Simulation Runs due to Unnecessary Warnings The current version Vivado HLS 2016.2 seems to introduce a bug into the RTL code for floating-point multipliers and adders, which causes the `OPMODE` input port oft the DSP slices involved to contain undefined values. While the functionality of the simulation model is not impaired, the undefined values cause the simulator to issue hundreds of warnings per clock cycle. This slows the co-simulation so much that even the smallest designs take hours to simulate. The full ZynqNet FPGA Accelerator architecture, simulated with a small five-layer CNN, has been run for four days without reaching an end. Suppressing the warning message is not possible and thus we had to live without the assurance of a working co-simulation.

4.5 Post-HLS Design Flow

Despite the significant performance loss due to the Pipeline Flushing Issue explained above, we decided to finish the design and try to estimate its performance. After a successful synthesis run, Vivado HLS creates a register transfer level description of the design in either VHDL or Verilog format. This RTL code can be exported in the so-called “IP Catalog” format, which is directly compatible with the Post-HLS *Vivado Design Suite* design flow.

Listing 2 Example Code combining Dataflow and Pipelining Compiler Directives. Loop `L_OUTER` cannot be pipelined, because loop `L_INNER` in the hierarchy below cannot be unrolled due to its variable bounds. Even worse, the Pipeline in loop `L_INNER` is unnecessarily flushed in every iteration of loop `L_OUTER`.

```
void task1_precalculate(int &channel) {
#pragma HLS INLINE off                    // needed to enable dataflow
  channel = precalculate();
}

void task2_hardwork(int o, int &channel) {
#pragma HLS INLINE off                    // needed to enable dataflow
  L_INNER: for (int i = 0; i < o; i++) {  // (variable loop bounds)
#pragma HLS PIPELINE II=1
    work_with(channel, i, o);             // do calculations (inlined)
  }
}

void f_dataflow(int o) {
#pragma HLS INLINE off                    // needed to enable dataflow
#pragma HLS DATAFLOW
  int channel;
  task1_precalculate(channel);
  task2_hardwork(o, channel);
}

L_OUTER: for (int o = 0; o < o_max; o++) {
#pragma HLS PIPELINE      /* DOES NOT WORK
                               because L_INNER can't be unrolled */
  f_dataflow(o);          /* FLUSHES the innermost pipeline
                               on every call */
}
```

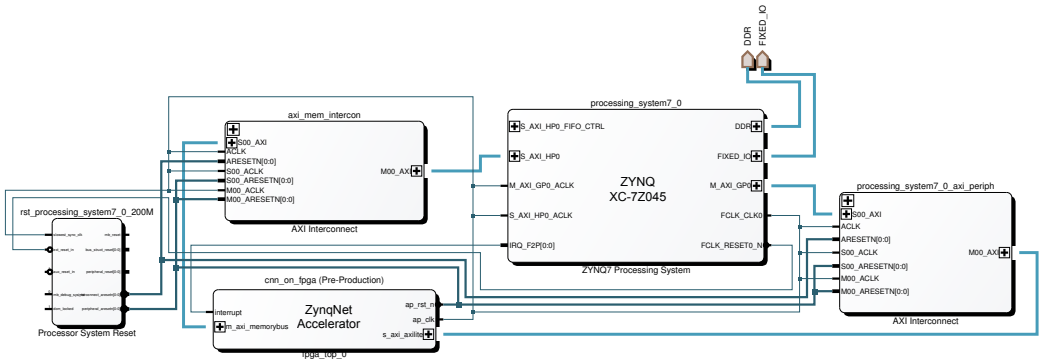


Figure 4.7.: Block Diagram of the ZynqNet FPGA Accelerator and Zynq Processing System as generated by the Vivado Design Suite Block Design tool.

4.5.1 Vivado Design Suite

Importing the VHLS Design The RTL code generated by Vivado HLS still hast to be compiled into a bit-stream for the FPGA. Luckily, this has been made very easy with the Vivado Design Suite. All required steps are described and illustrated in the High-Level Synthesis Tutorial (UG871) [114]. After importing the previously exported "IP Catalog" file in the Vivado Design Suite IP Catalog, a new *Block Design* can be created, and the VHLS design can be added as a new component. Further, the "Zynq7 Processing System" block has to be added and configured, before the components can be automatically connected using the *Run Block Automation* tool. Figure 4.7 shows a diagram of the fully connected ZynqNet FPGA Accelerator and Zynq XC-7Z045 blocks in the Vivado Design Suite Block Design tool. The Vivado Design Suite then automatically generates the necessary VHDL or Verilog wrapper files and instantiates the VHLS design. At this point, the design is ready for synthesis.

Synthesis With the schedule and resource allocation already fixed and all timing constraints properly set by Vivado HLS, there is not much left to be configured in Vivado Design Suite itself. The synthesis and implementation can be influenced slightly by choosing from a number of different preset strategies, and should run through smoothly. However, the timing results reported by the Vivado Design Suite can be quite different from the estimates reported by Vivado HLS.[12] The Design Suite results respect the load and fanout of each signal, and include all actual wire delays. The ZynqNet FPGA Accelerator synthesized to a slightly slower design than estimated, due to highly congested areas where more than 85 % of the routing capacity was utilized. The synthesis reports also highlight the longest paths, which gives vital hints for the optimization of the VHLS design. The routing delays can be significant, and therefore early synthesis runs, even with incomplete designs, are highly recommended. When the RTL synthesis is finished, the *bitstream* containing the binary FPGA configuration can be exported as a `.bit` file and loaded onto the Zynqbox.

4.5.2 Zynq Driver Development

With all these steps done, the FPGA side of the CNN accelerator is complete. The CPU-side software is also ready and tested as part of the test suite in Vivado HLS. However, there is still

[12] In our (very limited) experience, small designs resulted in faster implementations than estimated by Vivado HLS, while larger designs sometimes ran into routing problems and resulted in significantly slower implementations.

a missing key component: The low-level driver which lets the CPU software communicate with the FPGA block.

Xilinx Software Development Kit The Vivado Design Suite exports a `.hdf` *Hardware Design File* which contains a description of the Zynq setup configured in the *Block Design* step. Additionally, C-based driver files for the AXI4-Lite port are created. The hardware design file is then normally opened in the *Xilinx Software Development Kit* (SDK) application. The Xilinx SDK supports the creation of both bare-metal applications which do not rely on an operating system, and Linux-based applications. It includes all the tools needed to create a completely new, custom-tailored Linux environment including a custom *First Stage Boot Loader* (FSBL) for the chosen Zynq configuration and a custom *device tree* which has to be passed to the Linux kernel at boot time. However, the Zynqbox already runs a fully working and properly tested Linux installation and includes tools to load new bitstreams into the programmable logic. Due to lack of time, we strongly favored reusing this existing installation.

Memory-Mapped Input and Output The C-based driver files for the AXI4-Lite port, which can be exported from Vivado Design Suite, include functions for:

- starting and stopping the top-level FPGA entity
- checking the status of the accelerator (idle, ready, done)
- setting and getting every top-level function argument bundled into the AXI4-Lite interface

The files also contain all the relative address offsets of the corresponding memory-mapped registers. However, the driver relies on the *Userspace I/O* (UIO) kernel module, which in turn relies on the correct device tree being loaded into the kernel at boot time. Neiter of these requirements is fulfilled in the default SCS Zynqbox installation, and the advanced project time did not allow to fix this. Therefore, we had to patch the low-level driver functions to directly access the Zynq's memory bus to talk to the FPGA-based block, instead of using the elegant UIO module.

In Linux, the root user can directly access the physical memory bus without going through the virtual-to-physical address translation by reading and writing the `/dev/mem` character file. The physical address range which is assigned to the accelerator's AXI4-Lite interface can be found in the Address Editor in Vivado Design Suite's Block Design tool. The corresponding section of the `/dev/mem` file can then be *memory-mapped* into the application's own memory space using `int fd = open("/dev/mem", O_RDWR); volatile uint32_t* axilite = (uint32_t*)mmap(NULL, AXILITE_LENGTH, PROT_READ|PROT_WRITE, MAP_SHARED, fd, AXILITE_BASEADDR);` All subsequent reads and writes of `*(axilite + byte_offset)` are mapped into the `/dev/mem` file, and from there directly onto the memory bus. This method has been successfully implemented, and the communication between the FPGA accelerator and the CPU-based software is fully functional. The only drawback is the requirement for root privileges when running the ZynqNet Embedded CNN.

Remarks on the current ZynqNet Driver

- The current First Stage Boot Loader (FSBL) in the Zynqbox configures the `FCLK_CLK0` clock source for the programmable logic to 100 MHz. This setting cannot easily be changed, and therefore the ZynqNet FPGA Accelerator is currently only running at one half of the full 200 MHz clock speed which it was synthesized for.

- Before launching the driver, the *High Performance AXI Slave* port `S_AXI HP0` needs to be configured for 32 bit bus width. This can be done by calling `axi_hp_config 0 32` on the Zynqbox.
- The bitstream for the ZynqNet FPGA Accelerator can then be loaded by calling `loadbit zynqnet_200MHz.bit` in the firmware directory.

Evaluation and Results

5

> *You can't always get what you want.*
> *But if you try, sometimes*
> *You just might find*
> *You get what you need.*
>
> **— The Rolling Stones**

The last two chapters have given a detailed introduction the both the *ZynqNet CNN* and the *ZynqNet FPGA Accelerator*. Both of these components have been completed successfully, and together constitute the fully operable *ZynqNet Embedded CNN*. This chapter is concerned with an in-depth evaluation of this system regarding different aspects. First, we assess the performance of the *ZynqNet CNN* and compare it to prior work (section 5.1). Next, the performance of the *ZynqNet FPGA Accelerator* is estimated, both in its current version and with a number of potential improvements applied (section 5.2). The final section brings both components together and investigates the overall system performance of the *ZynqNet Embedded CNN* (section 5.3).

5.1 ZynqNet CNN Performance

In section 3.4 on the optimization of ZynqNet CNN, many aspects of the convolutional neural network's performance have already been discussed. Therefore, we confine ourselves to a summary of the most important characteristics in this section. To start with, table 5.1 repeats the comparison of the different CNN topologies, and this time includes the ZynqNet CNN and its key parameters. Figure 5.1 shows the updated design space exploration charts including the ZynqNet CNN.[1]

[1] The table as well as the design space exploration charts also report the parameters for SqueezeNet v1.1, which has been published during the development of ZynqNet CNN.

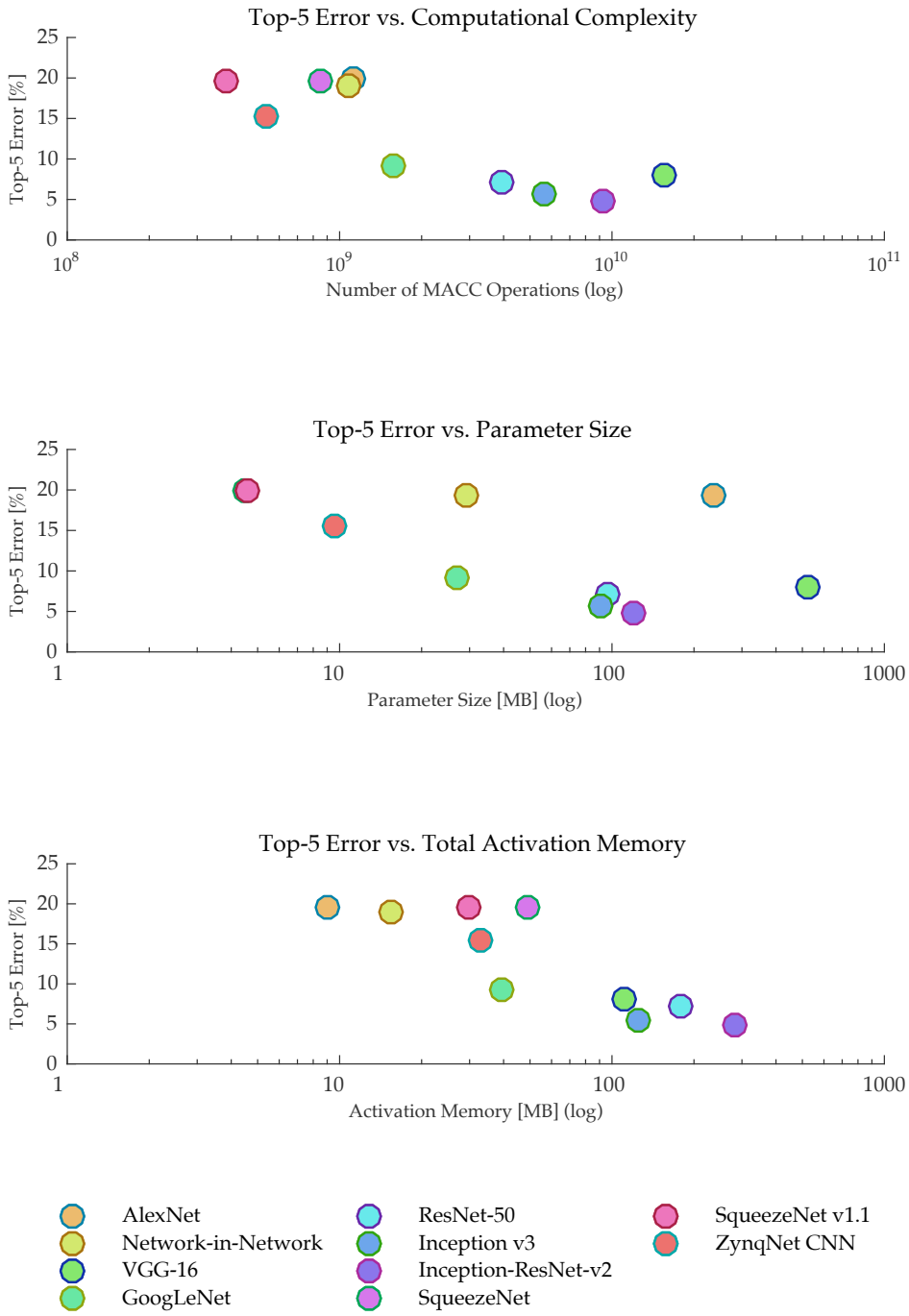


Figure 5.1.: Comparison of the ZynqNet CNN to CNN Architectures from Prior Work. Note the Logarithmic Scale on the x-Axes.

Table 5.1.: Comparison of ZynqNet CNN to CNN Architectures from Prior Work.[2]

	#conv. layers	#MACCs [millions]	#params [millions]	#activations [millions]	ImageNet top-5 error
ZynqNet CNN	**18**	**530**	**2.5**	**8.8**	**15.4%**
AlexNet	5	1 140	62.4	2.4	19.7%
Network-in-Network	12	1 100	7.6	4.0	~19.0%
VGG-16	16	15 470	138.3	29.0	8.1%
GoogLeNet	22	1 600	7.0	10.4	9.2%
ResNet-50	50	3 870	25.6	46.9	7.0%
Inception v3	48	5 710	23.8	32.6	5.6%
Inception-ResNet-v2	96	9 210	31.6	74.5	4.9%
SqueezeNet	18	860	1.2	12.7	19.7%
SqueezeNet v1.1	18	390	1.2	7.8	19.7%

5.1.1 Accuracy

The DSE charts in fig. 5.1 as well as the parameter summary in table 5.1 both show the increased accuracy of ZynqNet CNN with respect to both SqueezeNet and AlexNet. The top-5 error rate has been improved by more than 20 % relative to the starting point, or by a total of 4.3 percentage points. ZynqNet labels 84.6 % of all ImageNet examples correctly in the top-5 validation test. The top-1 accuracy has been improved from 55.9 % to 63.0 %. These results place ZynqNet approximately in the midfield of the tested CNN topologies. Note, however, that most of the other topologies have been optimized for maximum possible accuracy at the cost of heavily increased computational and memory requirements.

5.1.2 Computational Complexity

The computational complexity of ZynqNet has been lowered by almost 40 % in comparison to the original SqueezeNet and by more than 50 % compared to AlexNet. The network requires only 530 million multiplications and accumulations for one forward pass, making it one of the least expensive CNNs for image classification on ImageNet. The lightness of ZynqNet CNN is complemented by its highly regular architecture. The CNN consists only of convolutional layers, ReLU nonlinearities and one global average pooling layer.

5.1.3 Memory Requirements

In terms of its memory requirements, ZynqNet CNN differs from its ancestor SqueezeNet. SqueezeNet is mostly concerned with the minimization of the number of weight parameters. On the other hand, ZynqNet CNN tries to strike a balance between the number of parameters, the computational complexity, the size of each intermediate feature map and the overall accuracy of the convolutional neural network. Therefore, ZynqNet CNN uses 2.5 million weight parameters, which is twice as many as SqueezeNet, yet still roughly one order of magnitude less than most other CNNs for image classification on ImageNet. The total number of activations as well as the size of the largest output feature maps are approximately equal in the two networks.

[2]The ImageNet top-5 error rate is reported for single-net single-crop evaluation. #MACCs refers to the number of multiply-accumulate operations in one forward pass. #activations is the total pixel count in all output feature maps of all layers.

5.1.4 Resource Efficiency

MACC Operations From the *Top-5 Error vs. Computational Complexity* graph in fig. 5.1, it can be seen that ZynqNet CNN belongs to the Pareto-optimal designs, and in particular outperforms AlexNet, NiN and the original SqueezeNet in both accuracy and computational effort. Reaching a higher accuracy generally seems to be very expensive: In order to improve the top-5 accuracy by 6 percent-points compared to ZynqNet, GoogLeNet requires 3× more MACC operations. The state-of-the-art Inception-ResNet-v2 requires more than 17× more operations to improve the top-5 error by 11 percent-points. Of course, these last few percent-points towards 100 % accuracy contain the hardest images in ImageNet and thus require an overproportional effort. However, this implies that every actual application should precisely assess whether these last few percents of accuracy are actually required, or if orders of magnitude of computational effort can be saved in compromise.

Parameter Memory Another measure of resource efficiency can be seen in the *Top-5 Error vs. Parameter Size* graph in fig. 5.1. Here, a seemingly log-linear relationship between the number of parameters and the top-5 error of each model shows up: reducing the top-5 error by 5 percent-points requires approximately twice the number of weight parameters in all of the Pareto-optimal designs. ZynqNet CNN is again one of the Pareto-optimal designs, which highlights its good efficiency with regard to the number of weight parameters used.

5.2 ZynqNet FPGA Accelerator Performance

First and foremost, the ZynqNet FPGA Accelerator is meant to be a proof-of-concept for the implementation of CNNs on the basis of an FPGA. The secondary goal targets a maximum throughput on the given small and low-power platform, and in consequence a good power efficiency. The design and implementation details have been thoroughly discussed in the previous chapter 4, and the chosen architecture was found to be optimal with regard to the number of arithmetic and memory operations required. This section evaluates the finished design with regard to the factors *resource utilization*, *achieved clock frequency* and *operation schedule*, which determine the throughput of the accelerator. Finally, a number of potential architectural optimizations are highlighted.

5.2.1 Resource Utilization

The final ZynqNet FPGA Accelerator contains $N_{\mathrm{PE}} = 16$ processing units, which concurrently operate on the calculation of different output feature maps. Each processing unit contains a fully pipelined 3×3 multiply-accumulate unit with 9 separate floating-point multipliers and a subsequent adder tree for the summation of their products. This results in a total of 144 floating-point multipliers and 128 floating-point adders, which constitute the computational core of the accelerator. The processing units are fed from on-chip caches. In total, up to 1.7 MB parameters (442 000 single-precision floating-point weights) and 133 kB image data are buffered in the on-chip Block RAM. When synthesized for the Zynq XC-7Z045 FPGA, this configuration results in the resource requirements and device utilization figures shown in table 5.2. The fact that more than 90 % of all Block RAM resources and more than 80 % of the DSP slices are utilized highlights the good fit of the architecture to the given FPGA and is a result from the co-optimization of both the FPGA architecture and the ZynqNet CNN.

Table 5.2.: Resource Requirements and FPGA Utilization of the ZynqNet FPGA Accelerator when synthesized for the Zynq XC-7Z045.

resource	Block RAM	DSP Slices	FF	LUT
used	996	739	137 k	154 k
available	1090	900	437 k	218 k
utilization	91 %	82 %	31 %	70 %

5.2.2 Maximum Clock Frequency

Despite the high resource utilization and the resulting long paths in the interconnect, the ZynqNet FPGA Accelerator can still be synthesized for an adequate clock frequency of $f_{\max} = 200\,\mathrm{MHz}$. This is possible because the architecture fully distributes the computation as well as all the required data onto the different computational units. There are no dependencies between the individual computational units, even their results are accumulated separately. This leads to mostly local routing and few global interconnections, all of which can be sufficiently pipelined.

5.2.3 Operation Schedule

The last factor that determines the ZynqNet FPGA Accelerator's throughput is the efficiency of the operation schedule. The nested loops that form the system's algorithmic basis principally allow a fully pipelined operation, where new inputs are fetched and processed in every clock cycle. There are no data dependencies or feedback loops in the architecture that could prevent pipelining within a single convolutional layer.

Pipeline Flushing Issue in Vivado HLS 2016.2 An ideal processing pipeline also requires correspondingly efficient control logic and scheduling. When using High-Level Synthesis, the state machine that determines the operation schedule is automatically derived from the software model during synthesis. Unfortunately, as described in section 4.4.4, Vivado HLS 2016.2 has an issue with the derivation of an efficient operation schedule when pipelined regions are nested within dataflow sections, which are themselves part of an outer loop. In such situations, the scheduler flushes the complete inner pipeline in each iteration of the outer loop — something which is diametrically opposed to the idea of a pipelined core. This HLS-related deficiency strikes a weak spot in the ZynqNet FPGA Accelerator architecture, and results in a total slow-down of a factor of 6.2× across all ZynqNet CNN layers (see table E.1 in the appendix for the calculation). Therefore, the FPGA currently spends more than 80 % of its time flushing the innermost pipeline rather than performing any useful operations. The situation is worst for layers with a small number of output channels, where all channels can be calculated in one or two clock cycles using the 16 parallel processing units. The computation-to-flushing ratio is then as bad as $1 : 63$ or $2 : 64$. If the pipelining would function correctly, the computation of these layers would be limited by the time it takes to prefetch a new image patch (currently 9 clock cycles, with room for optimizations).

5.2.4 Potential Improvements

1. The *Pipeline Flushing Issue* is by far the most pressing problem. Most other optimizations only make sense when the pipelining functions correctly. Besides waiting for a fix in a future version of Vivado HLS, the only workaround is the implementation of the architecture in RTL code. Correct pipelining should improve the FPGA accelerator performance by a factor of 6.2.
2. The incorporation of *fixed-point arithmetic* is the second most important issue. A quick test synthesis with Vivado HLS indicates the potential to save 50 % of the Block RAMs and 80 % of the DSP slices by using a 16-bit fixed-point data format. One DSP slice suffices to calculate a multiply-accumulate operation in 16-bit fixed-point format, which allows 5× more processing units on the same FPGA fabric. However, the potential for parallelization in the output channels is mostly used up and moderate architectural changes would be necessary to tap the potential for parallelization in the input channels.[3]
3. An architectural bottleneck can be seen in the *prefetching of image pixels from the image cache*. Although this task is executed in parallel to the actual output channel calculation to hide the prefetch latency, the current delay of 9 clock cycles is relatively long. The architecture of the image cache might need to be improved to allow for more parallel read accesses, or a register-field might be used to cache the active image patch. This should be viable as the image cache occupies less than 8 % of the Block RAMs, and a total of 300 k flipflops are still unused. An ideal image cache would have a latency of less than 5 clock cycles, which would result in a speedup factor of 1.4.
4. A further architectural optimization concerns the *removal of the Global Pooling Cache*. As the latest CNN training experiments have shown, the ReLU nonlinearity in the last convolutional layers does not influence the overall classification accuracy (see table D.2 in the appendix). It is therefore possible to use the existing Output Cache for the pixel-wise accumulation during global average pooling. The Global Pooling Cache can be omitted, freeing approximately 16 Block RAMs and 5 DSP slices.
5. Finally, *1×1 convolutions* are currently not implemented efficiently: a full 3×3 MACC unit is used for the single necessary multiplication. The potential overall speedup from utilizing all 9 multipliers in the MACC units for individual 1×1 convolutions is approximately 1.2 with the current prefetch latency of 9 clock cycles. With an ideal Image Cache, a speedup factor of nearly 1.5 could be achieved.

In the ideal case, the incorporation of all these improvements would increase the ZynqNet FPGA Accelerator throughput by a factor of almost 64.

[3] Bit-widths smaller than 16 bits might be feasible from the CNN side, but would not allow further parallelization due to the lack of further DSP resources in the FPGA fabric. The DSP48E1 slices in the Zynq-7000 family do not support single-instruction-multiple-data (SIMD) for the multiplication of smaller data types.

5.3 System Performance

The ZynqNet Embedded CNN has been completely assembled and successfully taken into operation on a SCS Zynqbox. The full test system consists of

- SCS Zynqbox (Zynq XC-7Z045 with 1 GB DDR3 memory), running under Linux[4]
- ZynqNet CNN network description and trained weights, copied to the Zynqbox
- ZynqNet FPGA Accelerator bitstream, loaded into the FPGA fabric
- ZynqNet `/dev/mem` driver, connected to the AXI4-Lite configuration bus and the shared main memory
- ZynqNet CPU-side application, feeding the input images, launching the FPGA accelerator, measuring the timing and checking the classification results.

Using the above system configuration, the ZynqNet Embedded CNN has been evaluated in a realistic embedded scenario. The following final sections take a look at the overall system performance in terms of throughput and power efficiency.

5.3.1 Throughput

The embedded CNN's throughput is measured in terms of *images per second*. In a typical scenario, the CNN accelerator is configured with the network description and the trained weights beforehand, and is then utilized to classify an incoming stream of images. Therefore, the run-time per frame is measured from the moment when the FPGA accelerator is started, to the moment when the calculation of the Softmax Classification layer is finished. The ARM CPUs take $t_{CPU} = 45\,\mathrm{s}$ to calculate the ZynqNet CNN using the software model, with all optimizations and hardware floating-point support enabled. The current version of the ZynqNet FPGA Accelerator requires $t_F = 1955\,\mathrm{ms}$ per frame, which corresponds to a frame rate of $r_F = 0.51\,\mathrm{FPS}$. There are two important limiting factors at play in this result:

1. the FPGA clock rate `FCLK_CLK0` has been configured to 100 MHz instead of 200 MHz
2. the Pipeline Flushing Issue slows the design down by a factor of $s \approx 6.2$.

It is safe to assume that with these two issues corrected, the design would reach $t'_F \approx 158\,\mathrm{ms}$ per frame, and a reasonable real-time frame rate of $r'_F = 6.3\,\mathrm{FPS}$. Additionally, switching to a 16-bit fixed-point data format could potentially boost the frame rate to 30 FPS. With all improvements from section 5.2.4 implemented, the frame rate could ideally reach 65 FPS.

5.3.2 Power Efficiency

The energy consumption of the complete Zynqbox platform running the ZynqNet Embedded CNN has been evaluated using a Fluke 177 Multimeter and a TTi EX1810R laboratory power supply. The power measurements include all conversion losses, peripheral devices, as well as the system fan and can be found in table 5.3. The system has not been optimized for low-power operation due to the advanced project time, and a significant amount of energy is already consumed in the idle state.

[4] A custom Debian-based distribution with Linux Kernel version 3.12.0.

Table 5.3.: Power Measurement Results for the Zynqbox Platform running ZynqNet Embedded CNN. The total System Power includes all Conversion Losses, Peripherals and the System Fan.

system state	current draw @12 V	power dissipation
system idle	486 mA	5.83 W
CPU cores under full load	502 mA	6.02 W
FPGA accelerator idle	622 mA	7.46 W
FPGA accelerator running	650 mA	7.80 W

All measurements regarding the ZynqNet FPGA Accelerator's power dissipation have to be considered with caution due to the presence of the Pipeline Flushing Issue. The issue might substantially reduce the amount of switching activity in the FPGA fabric by causing zeros to be flushed through the computation pipeline, and might thereby distort the energy consumption. It is therefore currently not possible to make any precise statements regarding the system's power efficiency.

It is however relatively safe to assume a power consumption well below 20 W even under maximum load.[5] With a moderate assumption of $P = 12\,\mathrm{W}$ system power[6] and $r'_F = 6.3\,\mathrm{FPS}$, the ZynqNet FPGA Accelerator's power efficiency would be at

$$\eta_{\text{ZynqNet}} = \frac{r'_F}{P} = \frac{6.3\,\text{frames}}{12\,\mathrm{W\,s}} \approx 0.53\,\text{images/J} \tag{5.1}$$

The NVidia Jetson TX1 Whitepaper [52] provides some context for this number: AlexNet computed on a Intel Core i7 CPU reaches an efficiency $\eta_{\text{Corei7}} = 1.3\,\text{images/J}$, the same CNN on a NVidia Titan X $\eta_{\text{TitanX}} = 2.5\,\text{images/J}$ and on a NVidia Tegra X1 $\eta_{\text{TegraX1}} = 8.6\,\text{images/J}$. Although these figures probably do not consider conversion losses and the total system power, they show that further improvements of the FPGA accelerator, such as those presented in section 5.2.4, are unavoidable if the embedded system requires best-in-class power efficiency. With all known improvements (corrected pipeline flushing, 16-bit fixed-point arithmetic, improved image caching and ideal 1x1 convolutions) applied, the power efficiency could possibly be boosted to a respectable $\eta_{\text{improved}} = {}^{65\,\text{frames}}/_{12\,\mathrm{W\,s}} \approx 5.4\,\text{images/J}$.

[5] For example, consider that the Texas Instruments PMP8251 Power Management Reference Design for the Zynq-7000 Platform is dimensioned for a maximum power consumption of 23 W [116]. The ZynqNet FPGA Accelerator utilizes neither transceivers, I/Os nor the additional memory interfaces, and almost no peripherals, which would all cost considerable amounts of energy.

[6] Based on estimations using the Xilinx Power Estimator (XCE) tool [117].

6 Conclusion

In this master thesis, I designed and implemented a proof-of-concept FPGA-accelerated embedded Convolutional Neural Network. The *ZynqNet Embedded CNN* is designed for image classification on the ImageNet dataset and consists of two main components: *ZynqNet CNN*, a highly optimized and customized CNN topology, and the *ZynqNet FPGA Accelerator*, a FPGA-based architecture for the evaluation of ZynqNet CNN.

1. *ZynqNet CNN* is derived from the small and efficient SqueezeNet CNN topology. A detailed network analysis and optimization using the custom-designed *Netscope CNN Analyzer* tool has enabled a reduction of the classification error by 20 % relative to SqueezeNet. At the same time, the ZynqNet CNN requires 38 % less multiply-accumulate operations. Its topology is highly regular and consists of just three layer types: convolutional layers, ReLU nonlinearities and a global average pooling layer. Further, all layer dimensions have been converted to powers of two, which enables optimizations in the cache and memory addressing on the FPGA. Finally, the individual layers have been shaped to fit ideally onto the on-chip caches in the FPGA architecture.
2. The *ZynqNet FPGA Accelerator* is an FPGA-based architecture which allows the efficient evaluation of ZynqNet CNN and similar networks. It accelerates the convolutional layers, which encompass 99.3 % of all required operations, as well as the ReLU non-linearities and the global average pooling. The FPGA architecture benefits from the optimized CNN topology and is conceptually simple. Nevertheless, it supports a nested-loop algorithm which minimizes the number of arithmetic operations and memory accesses necessary for the evaluation of ZynqNet CNN, and can therefore be considered ideal. The FPGA accelerator has been synthesized using Vivado High-Level Synthesis for the Xilinx Zynq XC-7Z045 System-on-Chip, and reaches a clock frequency of 200 MHz with a device utilization of 80 % to 90 %.[1]

The ZynqNet Embedded CNN has been assembled into a fully working proof-of-concept system on the Xilinx Zynq-7000 All Programmable platform. This project clearly demonstrates the feasibility of FPGA-based embedded CNN implementations. The current solution already exhibits a reasonable performance, and a number of opportunities for further gains in throughput and power efficiency have been pointed out.

The tough requirements of embedded CNNs regarding the size, efficiency and computational power of the underlying computing platform are very hard to meet with the systems available today. A number of different platforms can be considered for future implementations, and by now it is not clear which one will conquer this market. Even though the presented ZynqNet Embedded CNN does not yet provide the massive amounts of computational power required for future applications of embedded image understanding, it may still serve as a stepping stone and a guide for further explorations of the FPGA as a platform for embedded CNNs. The biggest advantage of these FPGA-based systems can be seen in their scalability. Using a

[1] The application of High-Level Synthesis has been an interesting and instructive, yet also adventurous journey, which has been extensively detailed in this report for the benefit of later users.

larger device, much higher performance can be attained at comparable efficiency figures, while most other platforms are inherently limited by the amount of computational power available on a given chip. FPGAs therefore provide a promising path towards the vision of powerful embedded CNNs and the abundance of fascinating applications which could profit from on-board image understanding — and the ZynqNet Embedded CNN may be a first small step on this path.

I am looking forward to the exciting times ahead in this fast-paced field of research.

Declaration of Originality

A

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Gschwend	David

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 10.08.2016	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Task Description

B

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Prof. Dr. A. Gunzinger

Spring Semester 2016

MASTER THESIS

For
David Gschwend

FPGA accelerator for convolutional neural networks

Supervisor: Emanuel Schmid, Supercomputing Systems AG, Zürich

Start date: 10. February 2016
Submission date: 10. August 2016

Introduction

Convolutional neural networks (CNN) are currently a promising approach for image and object recognition, in both science and industry.
Many different network topologies have been published in the last years, trying to solve various image recognition tasks. The best performing topologies require enormous amounts of computational power and memory bandwidth. Even for simpler networks a specialized accelerator might be necessary to achieve useful framerates.
The project at hand shall examine the feasibility of programmable logic for the acceleration of convolutional neural networks.

Goal

The candidate shall build a demonstrator device that shows a convolutional neural network in operation. The accelerator shall run on a commercially available hardware platform, preferably a PCIe extension card.

Scope of work

1. Acquire a solid knowledge base about the topic.
2. Elaborate an overall concept for the demonstrator. Ideally, it can be included in the already existing platform for stereo images (e.g, with a PCIe card). The type of classification (objects, body parts, people ...) must be chosen such that the demonstration works well in the scope of a company or fair.
3. Elaborate a project plan. In particular, it shall be prioritized how much time is spent on the research of network topologies and their training.

Theory

4. Acquire an overview of the published work. Choose a suitable topology or build your own, if necessary. Do consider the available hardware platforms (see point 5).
5. Verify the chosen topology with an existing CNN framework. If required, train the network first.

Hardware / Firmware

6. The focus of this work shall not be put on frame rate or recognition performance. Instead, define an efficiency measure that you will optimize. The power efficiency (frames/Joule) should be taken into account, possibly extended by other resources (memory, dsp slices, ...)
7. Select and acquire (if necessary) a fitting hardware platform. Make sure that it integrates in the planned demonstrator platform.
8. Elaborate an architecture for the acceleration of the required layers.
9. Implement your architecture on the selected platform with a high-level synthesis language. Verify the code as you go.
10. Bring the accelerator into service and verify the correct function on the hardware.
11. Optimize the efficiency measure defined above for the setup by tuning both the network and the implementation.

Software

12. The demonstrator shall visualize the original image as well as the classification. If possible, the existing framework for the Stereo camera demo shall be used.

Procedure

General

- You will have a desk and a PC in the SCS premises at your disposal.
- Discuss your progress and procedure weekly with your supervisors.
- The work shall be continuously documented in a written report. Finalize your report towards the end of your thesis.
- The progress of the thesis shall be frequently compared against the project plan. Unforeseen issues with the intended procedure might require adjustments to the project plan and shall be documented.
- You will present your work at the end of the thesis in the scope of a Tuesday talk at SCS, including a live demonstration.

Deliveries

- Two signed copies of the written report must be turned in no later than six months after the start of the work. This task description shall be included in the report.
- Clean up your computer account: Keep only relevant files such as source code, schematics, layouts, configurations, special executables, documentation... A possible follow-up work must be able to start from these files.

Zürich, 5.2.2016

Prof. Dr. A Gunzinger

C Convolutional Neural Network Visualizations

C.1 3D Illustration of Convolutional Layers

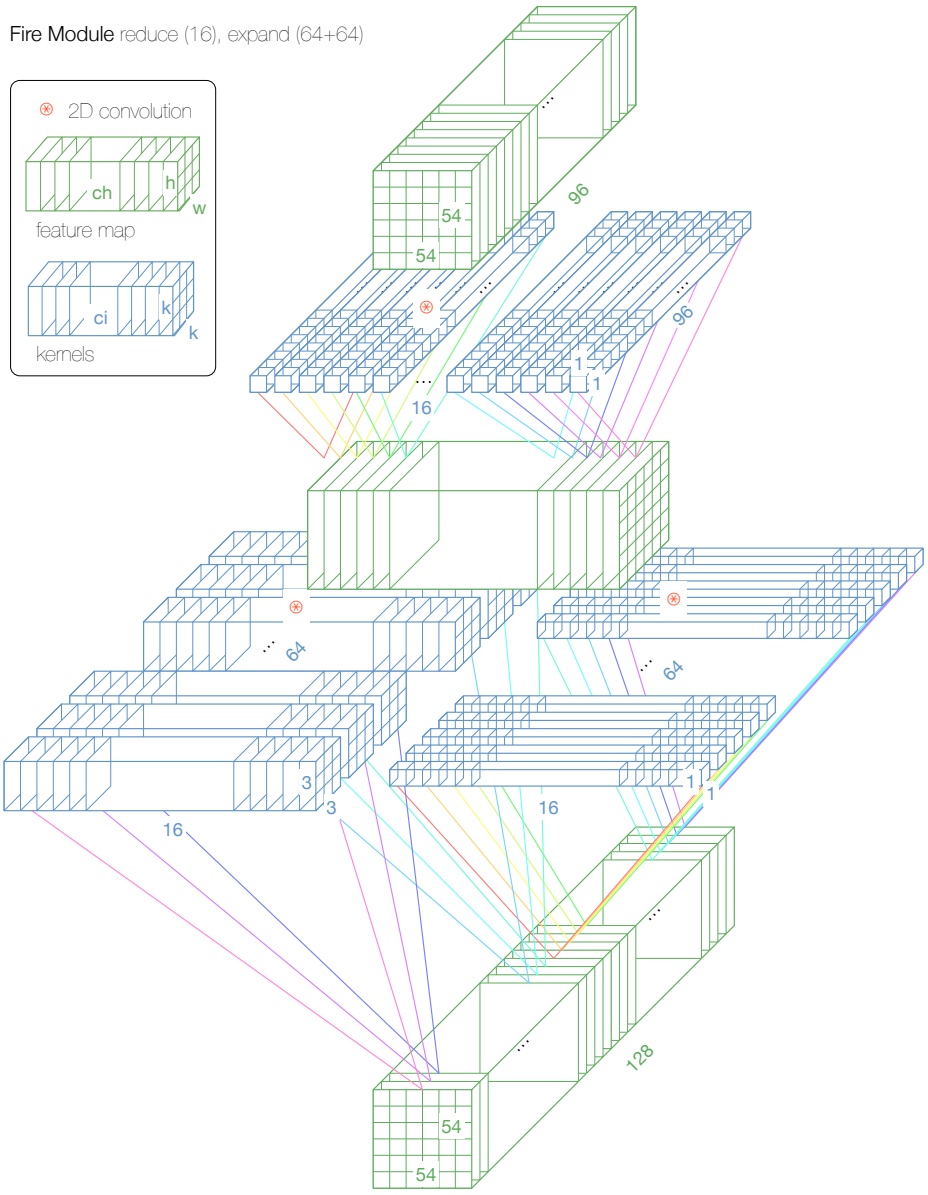


Figure C.1.: 3D Illustration of the Convolutional Layers in a SqueezeNet or ZynqNet Fire Module. Convolutional Layers can be seen as Transformations on 3D Volumes.

C.2 Netscope Visualizations of Different CNN Topologies

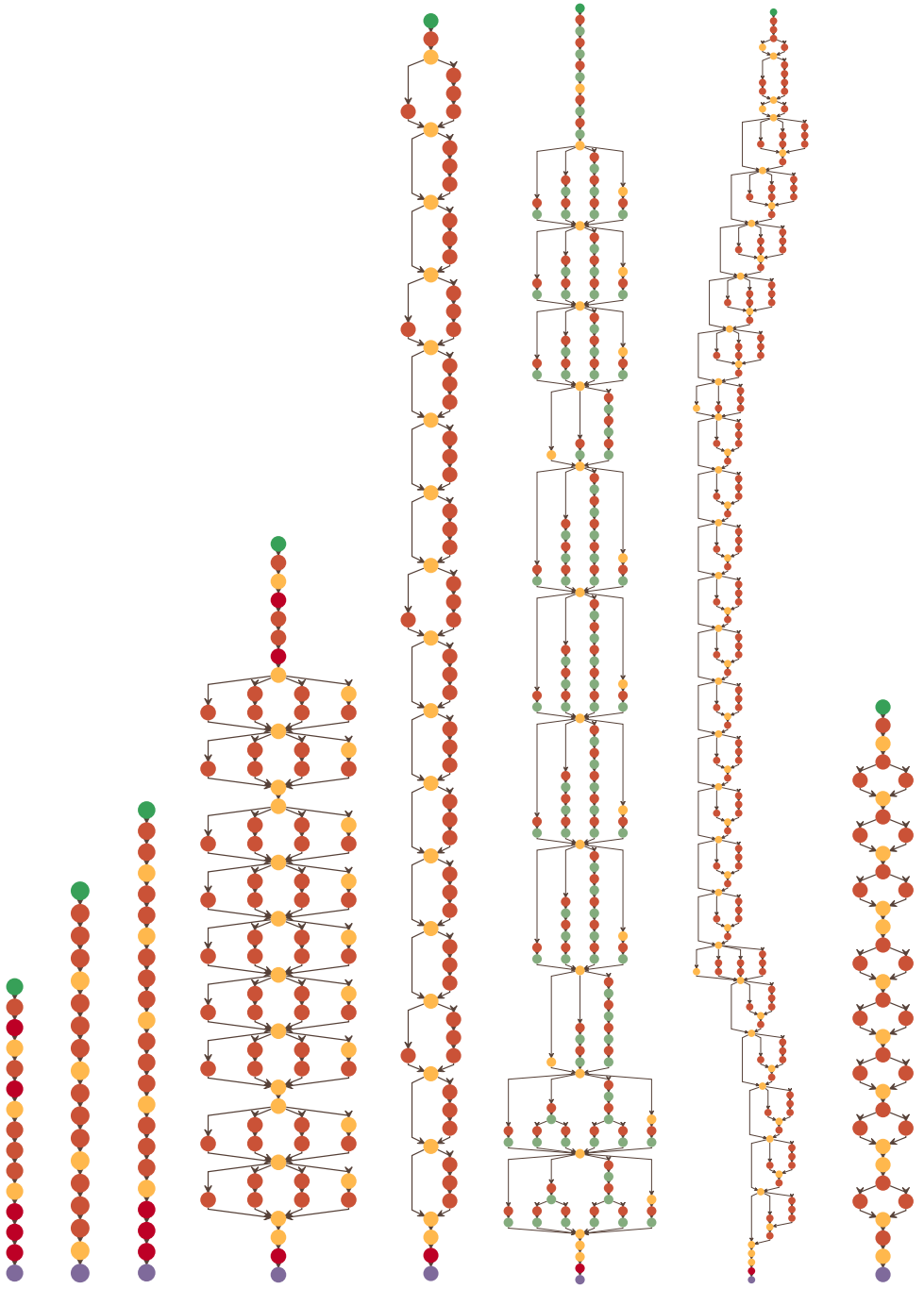


Figure C.2.: Netscope Visualizations of CNN Topologies from Prior Work. Left to right: AlexNet, Network-in-Network, VGG-16, GoogLeNet, ResNet-50, Inception v3, Inception-ResNet-v2, SqueezeNet.

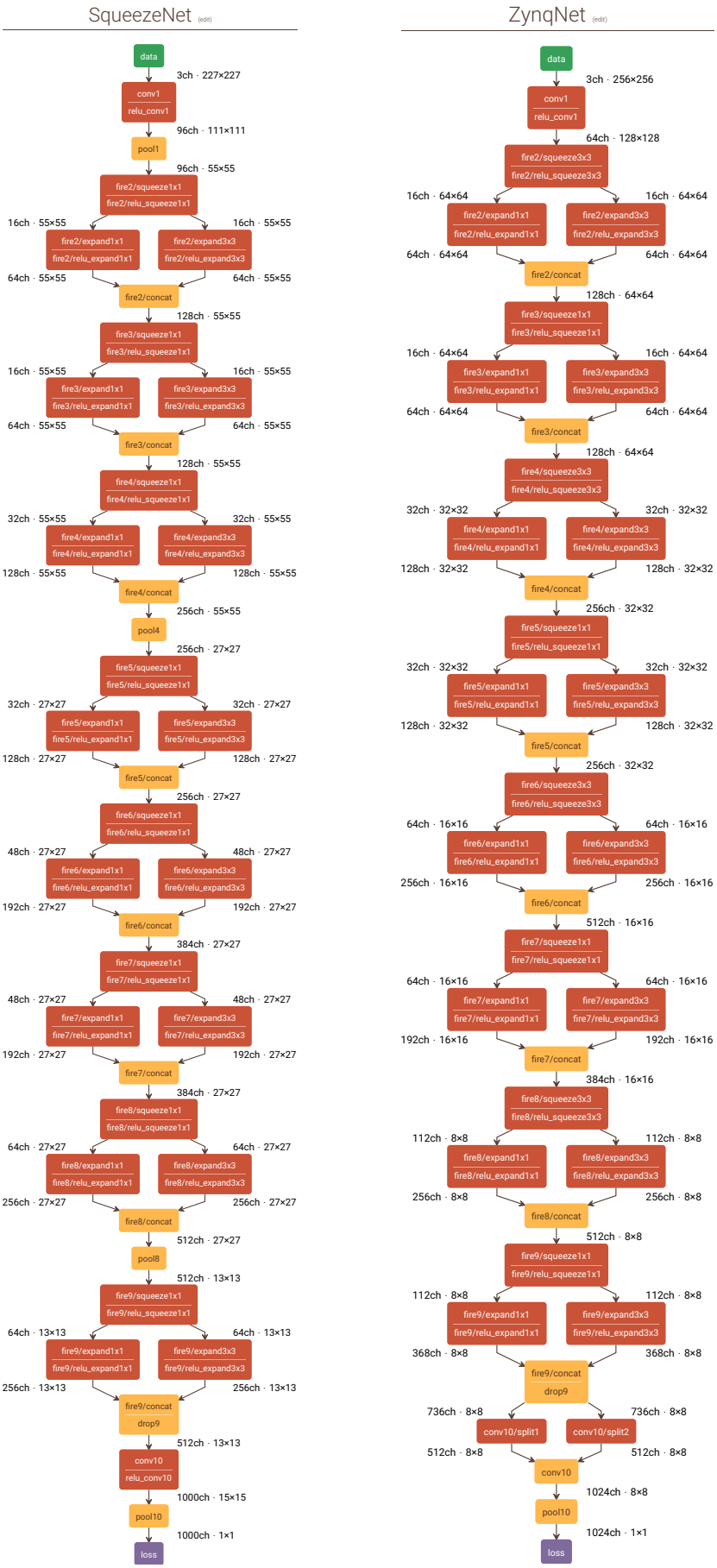


Figure C.3.: Detailed Netscope Visualizations of the SqueezeNet and the ZynqNet CNN Topologies.

C.3 Advanced Usage Tips and Restrictions for Netscope

Advanced Usage Tips

- Clicking a layer in the network graph directly scrolls to its entry in the summary table and vice-versa.
- The *edit* link next to the network title opens the `.prototxt` source code for the current CNN for editing.
- Shift-Enter in the Editor updates the graph and all tables.
- Naming layers according to the scheme "module/layer" groups these layers as one module in the summary table.
- Clicking “Excel-Compatible Results” at the very bottom opens a list with the most relevant layer characteristics, suited for further analysis in e.g. Excel or Matlab.

Current Restrictions

- In each layer, the field `top` needs to match the field `name`, except for `InPlace` layers where `top` matches `bottom`.
- `Data` and `Input` Layers are not accepted in all possible `.prototxt` syntaxes, refer to the built-in presets for valid examples.

CNN Training Details and Results

D

D.1 Hardware Components of the CNN Training Workstations

Table D.1.: Hardware Components used in the GPU-based CNN Training Workstations.

Count	Component	Type Name	Price (CHF)
2×	Graphics Card	Gigabyte GTX Titan X XTREME (12GB)	2300.00
1×	ATX Motherboard	Gigabyte Z170XP-SLI	150.00
1×	Processor	Intel Core i5 6400 Quad Core (2.70 GHz)	200.00
4×	DRAM Memory	Corsair Vengeance LPX 8GB DDR4-2400	150.00
1×	Solid-State Disk	Kingston SSDNow V300 (120GB, System)	50.00
1×	Solid-State Disk	Samsung 850 EVO Basic (500GB, Data)	160.00
1×	Hard Disk Drive	Western Digital Caviar Black (1TB, Archive)	70.00
1×	Power Supply	Cougar GX 800 V3 80 Plus Gold (800W)	120.00
1×	PC Case	Corsair Carbide 100R (Midi Tower)	60.00
	Total		3070.00

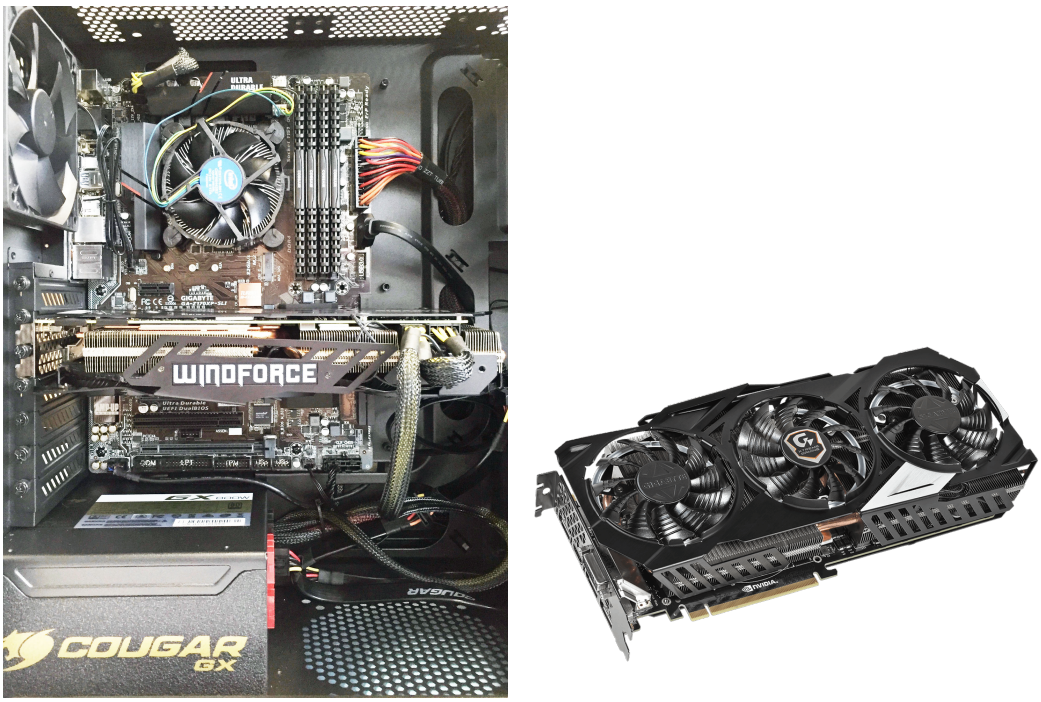


Figure D.1.: Photograph of the custom-built CNN Training Workstation (with one of two NVidia GeForce GTX Titan X installed) and of a Titan X Graphics Card (GPU photo from [118]).

D.2 Screenshots from the DIGITS CNN Training Software

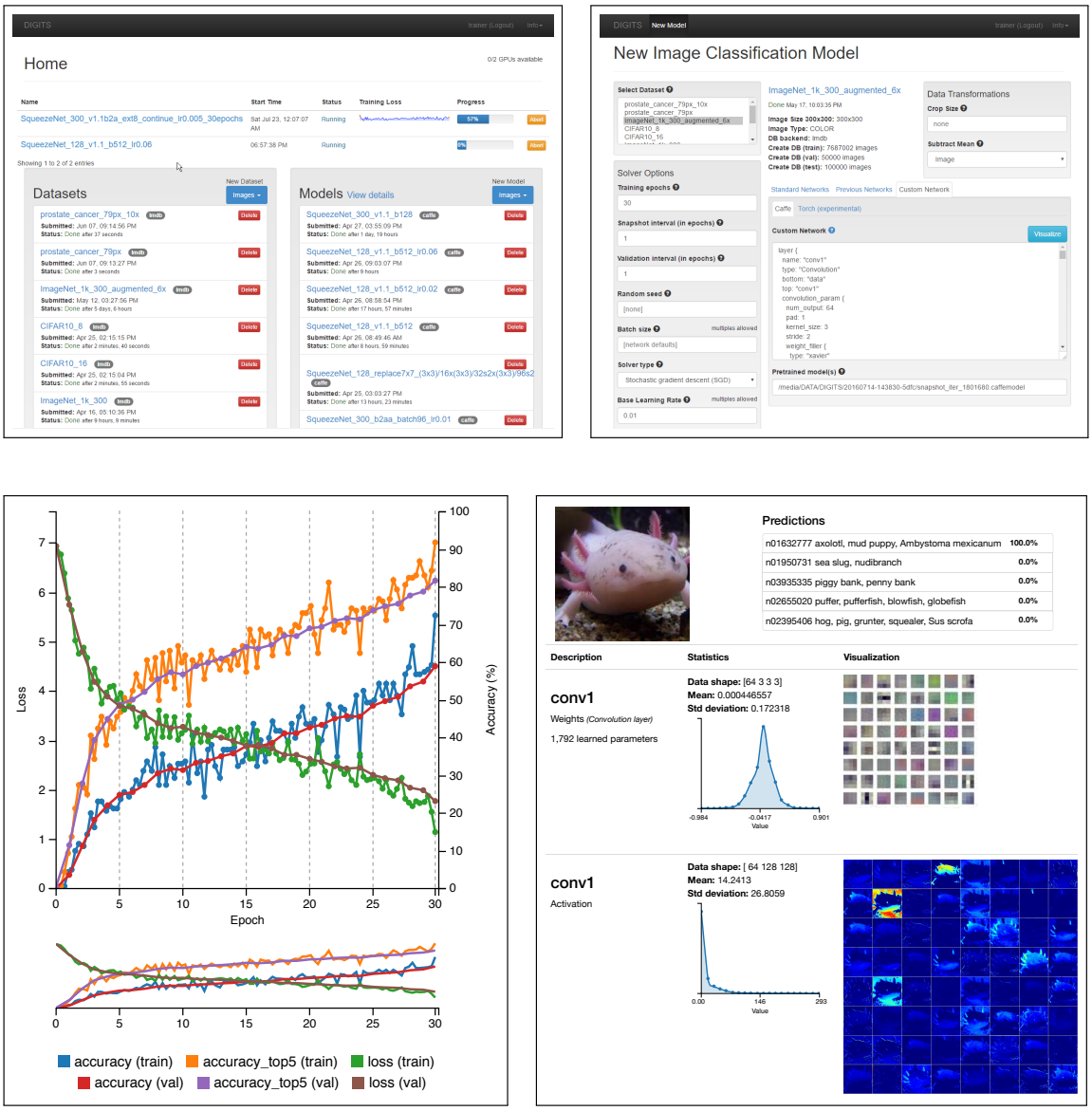


Figure D.2.: Screenshots from the DIGITS v3.4 CNN Training Software, showing the job schedule and dataset/model management on the home page (top left), the model definition interface (top right), a training progress chart with decreasing loss and increasing accuracy figures (bottom left), as well as the visualization of weights and activations in a trained network (bottom right).

D.3 Overview of all CNN Training Experiments

Table D.2.: Overview of all Experiments conducted during CNN Training. ZynqNet CNN is listed as *SqueezeNet_300_v1.1_b2a_ext8*.

SqueezeNet v1.0 Experiments (112x112 crops)	**Training Duration**	**#GPUs**	**Duration norm.***	**Accuracy Top-1**	**#MACC [M]**	**#params [M]**	**#activations [M]**	* dual-GPU training: x 1.7
SqueezeNet_128_base	5.0 h	2	9 h	47.86%	202	1.24	2.92	
SqueezeNet_128_replace_7x7_(3x3)/16x(3x3)/16x(3x3)/96	12.5 h	1	13 h	49.16%	243	1.25	3.86	
SqueezeNet_128_replace_7x7_11x11	9.0 h	1	9 h	47.30%	240	1.27	2.66	
SqueezeNet_128_replace_7x7_3x3	9.0 h	1	9 h	46.30%	230	1.23	4.03	
SqueezeNet_128_conv10_nopadding	8.8 h	1	9 h	48.36%	188	1.24	2.86	
SqueezeNet_128_conv1_pad3_conv10_pad0_lr0.02	8.8 h	1	9 h	49.34%	221	1.24	3.32	
SqueezeNet_128_2xBatchNorm_pool48_drop0.2_lr0.08	9.0 h	1	9 h	45.70%	203	1.25	2.98	
SqueezeNet_128_2xBatchNorm_pool48_drop0.2_lr0.01	9.0 h	1	9 h	43.80%	203	1.25	2.98	
SqueezeNet_128_replace7x7_(3x3)/16x(3x3)/16s2x(3x3)/96_withpool	9.0 h	1	9 h	38.96%	65	1.25	1.28	
SqueezeNet_128_replace7x7_(3x3)/16x(3x3)/16s2x(3x3)/96_nopool	9.5 h	1	10 h	49.27%	190	1.25	3.04	
SqueezeNet_128_replace7x7_(3x3)/16x(3x3)/32s2x(3x3)/96	9.8 h	1	10 h	47.60%	207	1.26	3.14	
SqueezeNet_128_v1.1 (squeezenet v1.1 on 128x128 crops)	9.0 h	1	9 h	43.47%	78	1.23	1.73	lr0.02: 42.8 lr0.04: 42.9 lr0.06: 43.5

SqueezeNet v1.0 Modifications (227x227 crops)	**Training Duration**	**#GPUs**	**Duration norm.***	**Accuracy Top-1**	**#MACC [M]**	**#params [M]**	**#activations [M]**	
SqueezeNet_256_base	32.0 h	1	32 h	55.85%	860	1.24	12.7	
SqueezeNet_256_replace_7x7_3x3	32.0 h	1	32 h	55.04%	774	1.23	13.4	
SqueezeNet_256_replace_7x7_(3x3)/16x(3x3)/16x(3x3)	57.0 h	1	57 h	55.97%	1060	1.25	16.7	
SqueezeNet_256_replace_7x7_(3x3)/16x(3x3)/16x(3x3)_MSRA	doesn't converge				1060	1.25	16.6	MSRA Init doesn't converge
SqueezeNet_256_replace_7x7_5x5	19.0 h	2	32 h	55.26%	830	1.24	13.4	
SqueezeNet_256_replace_7x7_11x11	19.0 h	2	32 h	55.60%	1090	1.27	12.5	
SqueezeNet_256_replace_7x7_(3x3)/8x(3x3)/16x(3x3)_pad0	16.0 h	2	27 h	54.10%	926	1.25	15.13	
SqueezeNet_256_replace_7x7_(3x3)/16x(3x3)/16S2x(3x3)					966	1.25	15.4	
SqueezeNet_256_replace_7x7_(3x3)/16x(3x3)/32x(3x3)	(not finished)				1350	1.26	18.4	
SqueezeNet_256_replace_7x7_(3x3)/16x(3x3)/96					950	1.24	15.3	
SqueezeNet_256_replace_7x7_(3x3)/16x(3x3)/96_pad0					936	1.24	15	
SqueezeNet_256_replace_7x7_(3x3)/8x(3x3)/96	19.0 h	2	32 h	55.19%	840	1.24	14.2	
SqueezeNet_256_crop227_base2_batch64_lr0.01	59.0 h	1	59 h	55.83%	1230	1.42	18.44	base2 works: 2x longer, same accuracy
SqueezeNet_256_crop256_base2_batch96_lr0.01	33.0 h	2	56 h	53.17%	1160	1.42	17.62	single-center-crop training = bad idea
SqueezeNet_300_crop256_base2_batch64_lr0.01	68.0 h	1	68 h	54.74%	1590	1.42	23.55	
SqueezeNet_300_crop256_base2allconv_batch112_lr0.01	43.0 h	1	43 h	55.47%	994	1.42	14.37	
SqueezeNet_300_crop256_b2allconv_b2aa_batch96_lr0.01	45.0 h	1	45 h	54.37%	1090	1.67	15.55	

SqueezeNet v1.1 Modifications (256x256 crops)	**Training Duration**	**#GPUs**	**Duration norm.***	**Accuracy Top-1**	**#MACC [M]**	**#params [M]**	**#activations [M]**	
SqueezeNet_300_v1.1	43.0 h	1	43 h	54.93%	506	1.23	10.2	super small network
SqueezeNet_300_v1.1_base2	43.0 h	1	43 h	55.90%	550	1.4	10.4	SqueezeNet v1.1 nice *and base2 helps!*
SqueezeNet_300_v1.1_base2_onlypool1	43.0 h	1	43 h	55.33%	486	1.4	8.44	
SqueezeNet_300_v1.1_base2_allconv_squeeze3x3S2	43.0 h	1	43 h	56.16%	520	1.41	8.18	
SqueezeNet_300_v1.1_base2_AllPoolToSq3x3S2	43.0 h	1	43 h	57.38%	650	1.57	10.07	-> +25% MACC, +22% Activations,
SqueezeNet_300_v1.1_b2a_ext1 [**b2a**: base2 + AllPoolToSqueez]	43.0 h	1	43 h	59.54%	574	2.51	9.4	+11% Weights vs. v11_b2a (above)
SqueezeNet_300_v1.1_b2a_ext1_ADAM_MSRA_lr.0.0003	42.0 h	1	42 h	49.40%	574	2.51	9.4	
SqueezeNet_300_v1.1_b2a_ext2	23.0 h	2	39 h	59.89%	683	2.94	9.83	
SqueezeNet_300_v1.1_b2a_ext3	40.0 h	1	40 h	60.25%	614	3.72	9.19	BEST RESULT
SqueezeNet_300_v1.1_b2a_ext4	40.0 h	1	40 h	56.67%	480	1.93	8.69	
SqueezeNet_300_v1.1_b2a_ext5	40.0 h	1	40 h	58.25%	535	2.49	9.07	
SqueezeNet_300_v1.1_b2a_ext8 (batch 160)	40.0 h	1	40 h	59.66%	530	2.52	8.8	lr0.01: 59.2 lr0.02: 59.7 lr0.03: 59.6 lr0.04: 59.0

ReLU yes/no at End?	**Training Duration**	**#GPUs**	**Duration norm.***	**Accuracy Top-1**	**#MACC [M]**	**#params [M]**	**#activations [M]**	
SqueezeNet_300_v1.1b2a_ext3_noFinalReLU	40.0 h	1	40 h	60.25%	614	3.72	9.19	
SqueezeNet_300_v1.1b2a_ext3_ReLUAfterPool	40.0 h	1	40 h	59.12%	614	3.72	9.19	
SqueezeNet_300_v1.1b2a_ext3_ReLUAfterConv10	40.0 h	1	40 h	60.17%	614	3.72	9.19	BEST RESULT
SqueezeNet_300_v1.1b2a_ext2_noFinalReLU	40.0 h	1	40 h	56.41%	683	2.94	9.83	

Augmented and Many-Epoch Trainings	**Training Duration**	**#GPUs**	**Duration norm.***	**Accuracy Top-1**	**#MACC [M]**	**#params [M]**	**#activations [M]**	
SqueezeNet_300_v1.1_base2_allconv_squeeze3x3S2_80epochs	110.0 h	1	110 h	59.08%	520	1.41	8.18	
SqueezeNet_300_v1.1_base2_AllPoolToSq3x3S2_80epochs	60.0 h	2	102 h	61.01%	650	1.57	9.95	
SqueezeNet_300_v1.1_b2a_ext1_b160_augmented6x	140.0 h	1	140 h	62.96%	574	2.51	9.4	BEST RESULT
Sq300_v11b2a_ext8_lr0.05_batch256_augment6x_60epochs_2gpu	149.0 h	2	253 h	62.73%	530	2.52	8.8	
SqueezeNet_300_v1.1b2a_ext8_continue_lr0.005_10epochs	26.0 h	2	44 h	62.96%	530	2.52	8.8	BEST RESULT (fine-tuning)
SqueezeNet_300_v1.1b2a_ext8_continue_lr0.005_30epochs	75.0 h	2	128 h	62.90%	530	2.52	8.8	BEST RESULT (fine-tuning)
SqueezeNet_256_v1.1b2a_ext8_anneal_lr0.003_b160_10epochs	7.5 h	1	8 h	61.73%	530	2.52	8.8	(fine-tuning)

Total GPU hours: 2205 h

D.4 Layer Description Table for ZynqNet CNN

Table D.3.: Detailed Description of all ZynqNet CNN Layers and their Parameters.

ID	Name	Type	Kernel	Stride	Pad	CH in	W×H in	CH out	W×H out	Notes
1	data	Data						3	256×256	
2	conv1	Convolution	3×3	2	1	3	256×256	64	128×128	
3	relu_conv1	ReLU				64	128×128	64	128×128	
4	fire2/squeeze3x3	Convolution	3×3	2	1	64	128×128	16	64×64	
5	fire2/relu_squeeze3x3	ReLU				16	64×64	16	64×64	
6	fire2/expand1x1	Convolution	1×1	1	0	16	64×64	64	64×64	
7	fire2/relu_expand1x1	ReLU				64	64×64	64	64×64	
8	fire2/expand3x3	Convolution	3×3	1	1	16	64×64	64	64×64	
9	fire2/relu_expand3x3	ReLU				64	64×64	64	64×64	
10	fire2/concat	Concat				128	64×64	128	64×64	
11	fire3/squeeze1x1	Convolution	1×1	1	0	128	64×64	16	64×64	
12	fire3/relu_squeeze1x1	ReLU				16	64×64	16	64×64	
13	fire3/expand1x1	Convolution	1×1	1	0	16	64×64	64	64×64	
14	fire3/relu_expand1x1	ReLU				64	64×64	64	64×64	
15	fire3/expand3x3	Convolution	3×3	1	1	16	64×64	64	64×64	
16	fire3/relu_expand3x3	ReLU				64	64×64	64	64×64	
17	fire3/concat	Concat				128	64×64	128	64×64	
18	fire4/squeeze3x3	Convolution	3×3	2	1	128	64×64	32	32×32	
19	fire4/relu_squeeze3x3	ReLU				32	32×32	32	32×32	
20	fire4/expand1x1	Convolution	1×1	1	0	32	32×32	128	32×32	
21	fire4/relu_expand1x1	ReLU				128	32×32	128	32×32	
22	fire4/expand3x3	Convolution	3×3	1	1	32	32×32	128	32×32	
23	fire4/relu_expand3x3	ReLU				128	32×32	128	32×32	
24	fire4/concat	Concat				256	32×32	256	32×32	
25	fire5/squeeze1x1	Convolution	1×1	1	0	256	32×32	32	32×32	
26	fire5/relu_squeeze1x1	ReLU				32	32×32	32	32×32	
27	fire5/expand1x1	Convolution	1×1	1	0	32	32×32	128	32×32	
28	fire5/relu_expand1x1	ReLU				128	32×32	128	32×32	
29	fire5/expand3x3	Convolution	3×3	1	1	32	32×32	128	32×32	
30	fire5/relu_expand3x3	ReLU				128	32×32	128	32×32	
31	fire5/concat	Concat				256	32×32	256	32×32	
32	fire6/squeeze3x3	Convolution	3×3	2	1	256	32×32	64	16×16	
33	fire6/relu_squeeze3x3	ReLU				64	16×16	64	16×16	
34	fire6/expand1x1	Convolution	1×1	1	0	64	16×16	256	16×16	
35	fire6/relu_expand1x1	ReLU				256	16×16	256	16×16	
36	fire6/expand3x3	Convolution	3×3	1	1	64	16×16	256	16×16	
37	fire6/relu_expand3x3	ReLU				256	16×16	256	16×16	
38	fire6/concat	Concat				512	16×16	512	16×16	
39	fire7/squeeze1x1	Convolution	1×1	1	0	512	16×16	64	16×16	
40	fire7/relu_squeeze1x1	ReLU				64	16×16	64	16×16	
41	fire7/expand1x1	Convolution	1×1	1	0	64	16×16	192	16×16	
42	fire7/relu_expand1x1	ReLU				192	16×16	192	16×16	
43	fire7/expand3x3	Convolution	3×3	1	1	64	16×16	192	16×16	
44	fire7/relu_expand3x3	ReLU				192	16×16	192	16×16	
45	fire7/concat	Concat				384	16×16	384	16×16	
46	fire8/squeeze3x3	Convolution	3×3	2	1	384	16×16	112	8×8	
47	fire8/relu_squeeze3x3	ReLU				112	8×8	112	8×8	
48	fire8/expand1x1	Convolution	1×1	1	0	112	8×8	256	8×8	
49	fire8/relu_expand1x1	ReLU				256	8×8	256	8×8	
50	fire8/expand3x3	Convolution	3×3	1	1	112	8×8	256	8×8	
51	fire8/relu_expand3x3	ReLU				256	8×8	256	8×8	
52	fire8/concat	Concat				512	8×8	512	8×8	
53	fire9/squeeze1x1	Convolution	1×1	1	0	512	8×8	112	8×8	
54	fire9/relu_squeeze1x1	ReLU				112	8×8	112	8×8	
55	fire9/expand1x1	Convolution	1×1	1	0	112	8×8	368	8×8	
56	fire9/relu_expand1x1	ReLU				368	8×8	368	8×8	
57	fire9/expand3x3	Convolution	3×3	1	1	112	8×8	368	8×8	
58	fire9/relu_expand3x3	ReLU				368	8×8	368	8×8	
59	fire9/concat	Concat				736	8×8	736	8×8	
60	drop9	Dropout				736	8×8	736	8×8	p = 0.5
61	conv10/split1	Convolution	1×1	1	0	736	8×8	512	8×8	
62	conv10/split2	Convolution	1×1	1	0	736	8×8	512	8×8	
63	conv10	Concat				1024	8×8	1024	8×8	
64	pool10	Pooling	8×8			1024	8×8	1024	1×1	global avg. pooling
65	loss	Softmax				1024	1×1	1024	1×1	

D.5 Tips and Trick for the Training of CNNs

The following section gives an overview of the training process with DIGITS v3.4 and NVidia's Caffe fork v0.14 [34]. The subsequent section lists a number of tips and tricks for the successful training of Convolutional Neural Networks.

Training with Caffe and DIGITS

Dataset Preparation The first step before training is the creation of a *dataset*. For the standard datasets *MNIST* (28×28 grayscale hand-written digits) and *CIFAR* (32×32 color images in 10 or 100 classes), a download script is provided with DIGITS. Other datasets have to be provided by the user. For datasets that are structured using separate subfolders for each class, DIGITS automatically recognizes the classes. In the case of ImageNet, the training and validation data has to be downloaded from the ILSVRC 2012 website [119]. The dataset has not changed since 2012 and consists of 138 GB training and 6 GB validation images. Additionally, the files `train.txt` and `val.txt` containing the mapping from image names to class numbers, as well as `synset_words.txt` with the mapping from class numbers to class names, are needed and can be downloaded using a tool supplied with Caffe [120].

Dataset Creation In DIGITS, a new dataset is created by choosing `Datasets > New Dataset Images > Classification`. Subfolder-structured datasets can be created by pointing to the root folder and choosing the percentage of images which should be used as validation images (the default of 25 % is reasonable in many cases). For other datasets, the paths to the image folders are set individually, and the `train.txt` and `val.txt` text files are uploaded. It is important to set the option `Shuffle Lines` to `Yes` to randomize the training set, and to upload `synset_words.txt` under `Labels`. The image size can be chosen freely,[1] the transformation type should be set to `half crop, half fill` for best results. LMDB is the default backend and allows fast image fetching during training. JPEG image compression slightly increases the runtime and completely loads the CPU during training, but significantly reduces the database size on disk.[2]

Data Augmentation Especially for small datasets (where CNNs have a high risk of overfitting), data augmentation can improve the quality of results by creating additional artificial training samples either on-the-fly or during dataset creation. DIGITS and Caffe do not natively support on-the-fly augmentation yet [121], but for this project, we added basic data augmentation support during dataset creation to DIGITS based on patch [122]. The user can create multiple copies of each training image and apply random rotations, hue modulations, contrast modulations and translations with a chosen probability and in a chosen modulation range.

Model Definition In DIGITS, a new CNN model is created by choosing `Models > New Model Images > Classification`. There is a choice of three preset networks (LeNet, AlexNet and GoogleNet), which can be adapted, as well as the option to enter a custom network definition in Caffe `.prototxt` or Torch format.[3] DIGITS uses a custom fork of Caffe for training,

[1] Most CNNs use 256×256 pixel images, with random 224 or 227 pixel crops during training. Inception networks use 299 pixel crops and therefore need larger training images.
[2] An ImageNet dataset with size 256×256 pixel images and JPEG compression occupies 43 GB disk space, the same dataset with 128×128 pixel images and lossless PNG compression 51 GB.
[3] Many additional CNN models can be found in the Caffe Model Zoo [123] and on Github [124], [125], [126].

which supports all layer types defined in `caffe.proto` [127] and is mostly compatible with the official Caffe.[4] However, most models require slight adaptations of their *data*, *softmax* and *accuracy* layers to match the DIGITS style (refer to the given networks for examples). Networks using custom layer types, such as *Highway Networks* [128], even require a recompilation of the underlying Caffe binaries. After entering the network description and selecting the previously created dataset, a crop size C may be specified. This causes a random $C \times C$ pixel crop of each example image to be used during training, which helps the model to develop translational invariance. During testing and inference, the $C \times C$ center-crop is used. Typically the mean of all pixels in the training set (the so-called "mean pixel") is subtracted from input images to help with training convergence.[5] In addition to these settings, DIGITS allows a pre-trained model (`.caffemodel` file) to be specified for fine-tuning instead of training the CNN model from scratch.

Solver Configuration With the dataset and the CNN model fully specified, only the *Solver* is left to be configured. Unfortunately, there exist no unique valid settings, and both model performance and training convergence are highly dependent on these *hyperparameters*. Besides the choice of the *Base Learning Rate* and the *Batch Size*, the *Solver Algorithm*, the *Learning Rate Schedule*, and the *Number of Training Epochs* can be changed. The duration of a training run is directly proportional to the *Number of Training Epochs*. Shorter trainings are usually welcome and allow more experiments to be made, but longer training runs usually converge to slightly more ideal solutions. The *Learning Rate* also strongly influences how well trainings converge, by scaling the weight updates made in each training step. If the learning rate is chosen too low, the training converges quickly, but to a suboptimal solution. And if the learning rate is set too high, the training may diverge. The learning rate starts at the *Base Learning Rate* and is then annealed over time according to the *Learning Rate Policy*. The default *Solver Algorithm* is *Stochastic Gradient Descent* (SGD), but other solver types are also available. The *Batch Size* determines how many training examples are pushed to the GPU at once. Larger Batch Sizes results in faster training, but may outgrow the available graphics memory. The final settings are the optional *Random Seed*, which enables reproducible weight initializations, as well as the *Snapshot and Validation Intervals*.

Training Launch DIGITS makes multi-GPU training as simple as selecting the desired number of GPUs to be used for the job. The scheduler then queues the task and waits until enough GPUs become available. Once the job transitions from *waiting* to *running*, an estimate for the remaining time is calculated. The training can be monitored in the progress chart, which tracks the CNN's loss and accuracy for both the training and the validation set. For a training from scratch, the loss curve should start decreasing within the first epoch, otherwise the learning rate was probably set too high. If the validation accuracy starts significantly drifting away from the training accuracy, the CNN model is overfitting and data augmentation or increased model regularization should be considered.

Training Tips and Tricks

The successful training of CNNs requires persistence, good intuition and experience. The following rules of thumb worked for most of our experiments, which mainly consisted of the

[4] One known incompatibility concerns Batch Normalization layers, which use a different syntax and rely on different libraries underneath.

[5] The helpfulness of mean subtraction is being debated in [121]. The researchers come to the conclusion that mean image subtraction is seldomly useful, and often one can even omit mean pixel subtraction.

training of SqueezeNet variants on ImageNet with 128×128 or 256×256 pixel crops using one or two NVidia GeForce GTX Titan X GPUs:

Batch Size Choose the maximum Batch Size that still fits onto the GPU to speed up the training significantly. Batch Sizes are often chosen as powers of 2 to fit well onto the GPU's CUDA cores, but multiples of 32 seem to work just as well. The Batch Size further influences training convergence, because weight updates are deferred and averaged over each batch. When multiplying the Batch Size by a factor of k, the Base Learning Rate should also be changed by a factor of $\sqrt{k}$ (although a factor of k usually works just as well) [129].

Learning Rate and Batch Size Sweeps The ideal Learning Rate depends on the Batch Size, the dataset, as well as the CNN model. Base Learning Rates typically lie between 0.0001 and 0.01 when training from scratch, and even smaller learning rates may be used for finetuning. The `solver.prototxt` file supplied with most pre-trained networks can give a hint, otherwise a trial and error approach with a geometric series works best. DIGITS accepts lists in the format `[0.001, 0.002, 0.004]` for Batch Size and Base Learning Rate and automatically generates jobs for each permutation.

Learning Rate Policies As mentioned, the learning rate is usually lowered over time during the training, which helps the optimization to settle into an optimum. The classic approach steps down the learning rate every few epochs by a fixed factor k. Other approaches include exponential decay, sigmoidal decay and polynomial decay. Mishkin et al. thoroughly explored many optimizations and found a *Linear Learning Rate Policy* (polynomial decay with power 1) to work best for AlexNet-like CNNs [104].

Solver Types DIGITS supports a number of different optimization algorithms, including Nesterov's Accelerated Gradient (NAG), Adaptive Gradient (AdaGrad), RMSprop, AdaDelta, and Adam which should in theory all lead to a faster training. These algorithms improve convergence for example by adding momentum to the optimization steps, or by adaptively tuning the learning rate for each individual weight (see [102] for details and a beautiful illustration). Despite their appeal, these solvers require a new trial-and-error hyperparameter search and quick tests led to inferior optimization results in our case. Therefore, we used the basic Stochastic (Mini-Batch) Gradient Descent algorithm in this project.

Batch Normalization ResNet and all Inception variants use Batch Normalization Layers. Unfortunately, the implementation and syntax of BN layers differs between the original Caffe and the NVidia fork, as well as between CPU and GPU-based computation. See [130], [131], [132] for hints if you want to experiment with BN layers.

Cloning existing Jobs The fastest way to create a new design iteration is to clone a previous job, which copies the network description and all previous settings into a new job, ready for customization. The list of "previous networks" then optionally allows the pre-trained network weights to be loaded as initialization or for finetuning.

Running Time There are two peculiarities with regard to the running time. First, when starting a new job, the estimated time remaing is initially very high and completely wrong. The value takes a few minutes to settle to a realistic estimate. Second, when looking for the runtime of an active or completed job, the correct value is found in the job page under `Job Status > Train Caffe Model > Running`. All other values (including the `runtime` field in table `Details`) wrongly add the job's *waiting* time.

E

FPGA Accelerator Details

E.1 Analysis of the Pipeline Flushing Issue

Table E.1.: Calculation of the Slow-Down Factor caused by the Pipeline Flushing Issue in Vivado HLS 2016.2.

Pipeline Flushing Issue Slow-Down

N_PE	inner L w/flush	L_preloadPixels
16	63	9

layer	width	height	ch_in	ch_out	stride	outer loops	inner loops	inner L w/flush	total L w/flush	inner L pipelined	total L pipelined	speed-up pipelined
c1	256	256	3	64	2	49152	4	66	3244032	9	442368	7.3
f2/s3	128	128	64	16	2	262144	1	63	16515072	9	2359296	7.0
f2/e1	64	64	16	64	1	65536	4	66	4325376	9	589824	7.3
f2/e3	64	64	16	64	1	65536	4	66	4325376	9	589824	7.3
f3/s1	64	64	128	16	1	524288	1	63	33030144	9	4718592	7.0
f3/e1	64	64	16	64	1	65536	4	66	4325376	9	589824	7.3
f3/e3	64	64	16	64	1	65536	4	66	4325376	9	589824	7.3
f4/s3	64	64	128	32	2	131072	2	64	8388608	9	1179648	7.1
f4/e1	32	32	32	128	1	32768	8	70	2293760	11	360448	6.4
f4/e3	32	32	32	128	1	32768	8	70	2293760	11	360448	6.4
f5/s1	32	32	256	32	1	262144	2	64	16777216	9	2359296	7.1
f5/e1	32	32	32	128	1	32768	8	70	2293760	11	360448	6.4
f5/e3	32	32	32	128	1	32768	8	70	2293760	11	360448	6.4
f6/s3	32	32	256	64	2	65536	4	66	4325376	9	589824	7.3
f6/e1	16	16	64	256	1	16384	16	78	1277952	19	311296	4.1
f6/e3	16	16	64	256	1	16384	16	78	1277952	19	311296	4.1
f7/s1	16	16	512	64	1	131072	4	66	8650752	9	1179648	7.3
f7/e1	16	16	64	192	1	16384	12	74	1212416	15	245760	4.9
f7/e3	16	16	64	192	1	16384	12	74	1212416	15	245760	4.9
f8/s3	16	16	384	112	2	24576	7	69	1695744	10	245760	6.9
f8/e1	8	8	112	256	1	7168	16	78	559104	19	136192	4.1
f8/e3	8	8	112	256	1	7168	16	78	559104	19	136192	4.1
f9/s1	8	8	512	112	1	32768	7	69	2260992	10	327680	6.9
f9/e1	8	8	112	368	1	7168	23	85	609280	26	186368	3.3
f9/e3	8	8	112	368	1	7168	23	85	609280	26	186368	3.3
c10/p1	8	8	736	512	1	47104	32	94	4427776	35	1648640	2.7
c10/p2	8	8	736	512	1	47104	32	94	4427776	35	1648640	2.7
								Total FPGA Cycles for MACC	137537536		22259712	6.2
								Inference Time @200MHz [ms]	688		111	
								Speedup Factor			**6.18**	

E.2 Detailed Block Diagram for the ZynqNet FPGA Accelerator

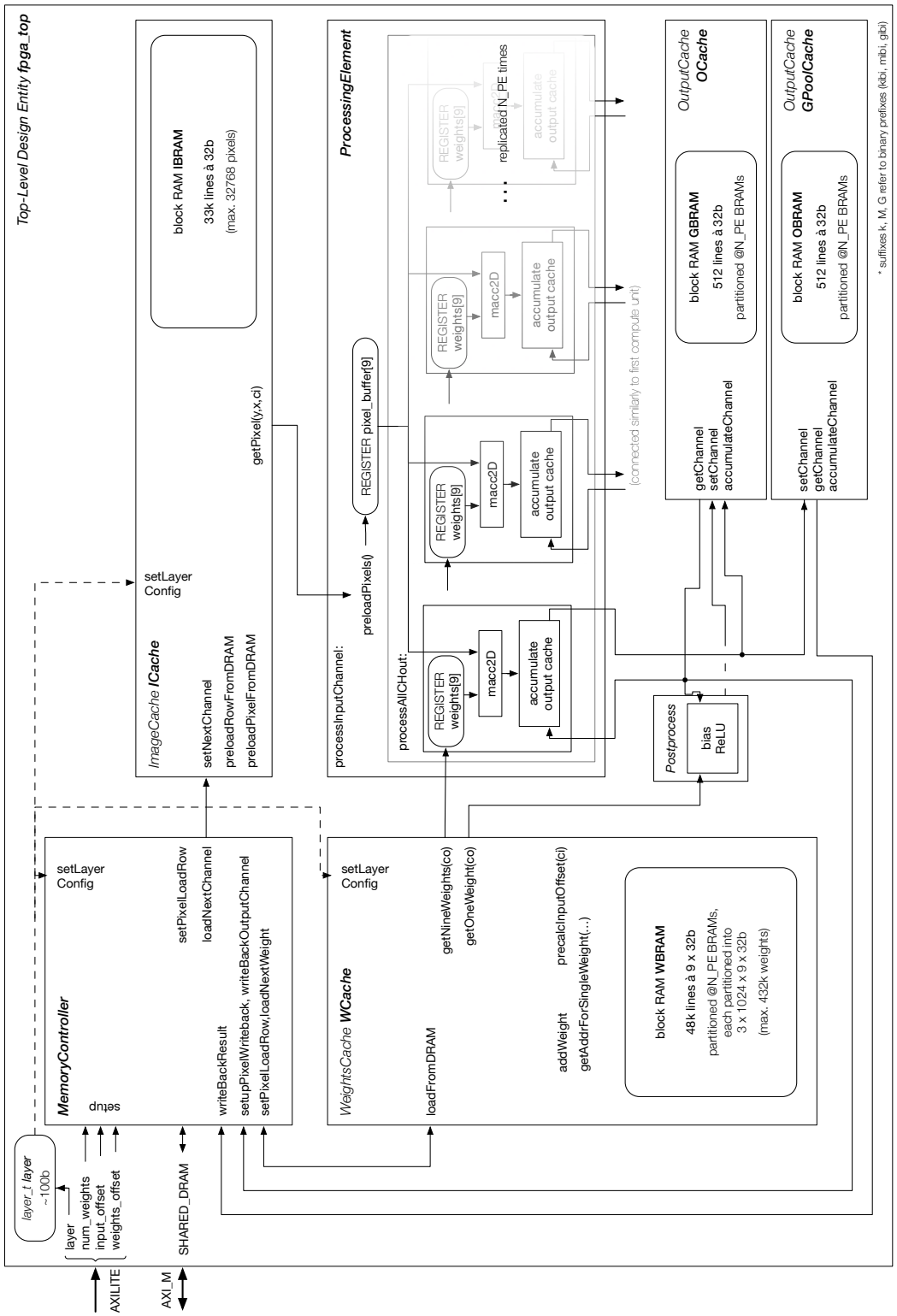


Figure E.1.: Detailed Block Diagram for the ZynqNet FPGA Accelerator, including actual Cache Sizes and References to the C++ Software Implementation.

.

Bibliography

[1]F. Conti and L. Benini, „A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters“, in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2015, pp. 683–688 (cit. on p. 1).

[2]W. S. McCulloch and W. Pitts, „A logical calculus of the ideas immanent in nervous activity“, *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943 (cit. on p. 1).

[6]N. Gupta, „Machine learning in the cloud: Deep neural networks on FPGAs“, *Xcell Journal*, vol. 92, pp. 46–51, Sep. 2015 (cit. on p. 2).

[12]I. Goodfellow, Y. Bengio, and A. Courville, „Deep learning“, Book in preparation for MIT Press, 2016 (cit. on pp. 3, 5).

[15]S. Herculano-Houzel, „The human brain in numbers: A linearly scaled-up primate brain“, *Frontiers in Human Neuroscience*, vol. 3, no. 31, 2009 (cit. on p. 3).

[16]V. G. Maltarollo, K. M. Honório, and A. B. F. da Silva, „Applications of artificial neural networks in chemical problems“, 2013 (cit. on p. 4).

[17]D. Gschwend, C. Mayer, and S. Willi, „Origami: Design and implementation of a convolutional neural network accelerator ASIC“, Semester Thesis, ETH Zürich, 2015 (cit. on pp. 4, 6, 17).

[18]S. Hochreiter and J. Schmidhuber, „Long short-term memory“, *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997 (cit. on p. 4).

[19]H. Sak, A. W. Senior, and F. Beaufays, „Long short-term memory recurrent neural network architectures for large scale acoustic modeling.“, in *INTERSPEECH*, 2014, pp. 338–342 (cit. on p. 4).

[20]Y. LeCun, Y. Bengio, and G. Hinton, „Deep learning“, *Nature*, vol. 521, no. 7553, pp. 436–444, 2015 (cit. on p. 5).

[21]C. Farabet, B. Martini, P. Akselrod, *et al.*, „Hardware accelerated convolutional neural networks for synthetic vision systems“, in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, IEEE, 2010, pp. 257–260 (cit. on p. 6).

[22]K. He, X. Zhang, S. Ren, and J. Sun, „Delving deep into rectifiers: Surpassing human-level performance on imagenet classification“, in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1026–1034 (cit. on p. 7).

[23]I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, „Maxout Networks“, *ArXiv e-prints*, Feb. 2013. arXiv: `1302.4389 [stat.ML]` (cit. on p. 7).

[24]D. Clevert, T. Unterthiner, and S. Hochreiter, „Fast and accurate deep network learning by exponential linear units (elus)“, *CoRR*, vol. abs/1511.07289, 2015 (cit. on p. 7).

[25]A. Krizhevsky, I. Sutskever, and G. E. Hinton, „ImageNet classification with deep convolutional neural networks“, in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105 (cit. on pp. 7, 9).

[26]S. Ioffe and C. Szegedy, „Batch normalization: Accelerating deep network training by reducing internal covariate shift“, *CoRR*, vol. abs/1502.03167, 2015 (cit. on p. 7).

[28]R. Al-Rfou, G. Alain, A. Almahairi, *et al.*, „Theano: A python framework for fast computation of mathematical expressions“, *CoRR*, vol. abs/1605.02688, 2016 (cit. on p. 8).

[29]E. Battenberg, S. Dieleman, D. Nouri, *et al.*, *Lasagne*, `https://github.com/Lasagne/Lasagne`, 2014 (cit. on p. 8).

[30]F. Chollet, *Keras*, `https://github.com/fchollet/keras`, 2015 (cit. on p. 8).

[31]R. Collobert, K. Kavukcuoglu, and C. Farabet, „Torch7: A Matlab-like environment for machine learning“, in *BigLearn, NIPS Workshop*, 2011 (cit. on p. 8).

[32]Martin Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015 (cit. on p. 8).

[33]Y. Jia, E. Shelhamer, J. Donahue, *et al.*, „Caffe: Convolutional architecture for fast feature embedding“, *ArXiv preprint arXiv:1408.5093*, 2014 (cit. on p. 8).

[34]NVIDIA Corporation, *NVIDIA Deep Learning GPU Training System (DIGITS)*, `https://developer.nvidia.com/digits`, 2016 (cit. on pp. 8, 25, 80).

[35]O. Russakovsky, J. Deng, H. Su, *et al.*, „ImageNet Large Scale Visual Recognition Challenge“, *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015 (cit. on p. 9).

[36]F. N. Iandola, K. Ashraf, M. W. Moskewicz, and K. Keutzer, „Firecaffe: Near-linear acceleration of deep neural network training on compute clusters“, *CoRR*, vol. abs/1511.00175, 2015 (cit. on p. 10).

[37]M. Lin, Q. Chen, and S. Yan, „Network in network“, *CoRR*, vol. abs/1312.4400, 2013 (cit. on p. 10).

[38]K. Simonyan and A. Zisserman, „Very deep convolutional networks for large-scale image recognition“, *ArXiv preprint arXiv:1409.1556*, 2014 (cit. on p. 10).

[40]C. Szegedy, W. Liu, Y. Jia, *et al.*, „Going deeper with convolutions“, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9 (cit. on p. 10).

[41]K. He, X. Zhang, S. Ren, and J. Sun, „Deep residual learning for image recognition“, *ArXiv preprint arXiv:1512.03385*, 2015 (cit. on p. 11).

[42]C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, „Rethinking the inception architecture for computer vision“, *CoRR*, vol. abs/1512.00567, 2015 (cit. on p. 11).

[43]C. Szegedy, S. Ioffe, and V. Vanhoucke, „Inception-v4, Inception-ResNet and the impact of residual connections on learning“, *CoRR*, vol. abs/1602.07261, 2016 (cit. on p. 11).

[44]F. N. Iandola, M. W. Moskewicz, K. Ashraf, *et al.*, „SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1mb model size“, *ArXiv:1602.07360*, 2016 (cit. on pp. 11, 26).

[45]M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, „Predicting parameters in deep learning“, *CoRR*, vol. abs/1306.0543, 2013 (cit. on p. 11).

[46]E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, „Exploiting linear structure within convolutional networks for efficient evaluation“, in *Advances in Neural Information Processing Systems*, 2014, pp. 1269–1277 (cit. on p. 11).

[47]J. Jin, A. Dundar, and E. Culurciello, „Flattened convolutional neural networks for feedforward acceleration“, *CoRR*, vol. abs/1412.5474, 2014 (cit. on p. 11).

[48]Y. Kim, E. Park, S. Yoo, *et al.*, „Compression of deep convolutional neural networks for fast and low power mobile applications“, *CoRR*, vol. abs/1511.06530, 2015 (cit. on p. 11).

[49]G. Thimm and E. Fiesler, „Pruning of neural networks“, IDIAP, Tech. Rep., 1997 (cit. on p. 11).

[50]S. Anwar, K. Hwang, and W. Sung, „Structured pruning of deep convolutional neural networks“, *ArXiv preprint arXiv:1512.08571*, 2015 (cit. on p. 11).

[53]S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, „Deep learning with limited numerical precision“, (cit. on p. 11).

[54]S. Anwar, K. Hwang, and W. Sung, „Fixed point optimization of deep convolutional neural networks for object recognition“, in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2015, pp. 1131–1135 (cit. on p. 11).

[55]W. Sung, S. Shin, and K. Hwang, „Resiliency of deep neural networks under quantization“, *ArXiv preprint arXiv:1511.06488*, 2015 (cit. on p. 12).

[56]M. Courbariaux and Y. Bengio, „BinaryNet: Training deep neural networks with weights and activations constrained to +1 or -1“, *CoRR*, vol. abs/1602.02830, 2016 (cit. on pp. 12, 32).

[57]M. Courbariaux, Y. Bengio, and J. David, „Binaryconnect: Training deep neural networks with binary weights during propagations“, *CoRR*, vol. abs/1511.00363, 2015 (cit. on pp. 12, 32).

[58]P. Gysel, M. Motamedi, and S. Ghiasi, „Hardware-oriented approximation of convolutional neural networks“, *ArXiv preprint arXiv:1604.03168*, 2016 (cit. on pp. 12, 33).

[59]S. Han, H. Mao, and W. J. Dally, „A deep neural network compression pipeline: Pruning, quantization, huffman encoding“, *ArXiv preprint arXiv:1510.00149*, 2015 (cit. on p. 12).

[60]H. Kaeslin, *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*, 1st ed. Cambridge University Press, 2008 (cit. on pp. 13, 14).

[67]M. Fingeroff, *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010 (cit. on pp. 14, 40, 44).

[79]S. Liu, *Analog VLSI: Circuits and principles*, ser. Bradford Book. MIT Press, 2002 (cit. on p. 16).

[81]G. Lacey, G. W. Taylor, and S. Areibi, „Deep learning on FPGAs: Past, present, and future“, *ArXiv preprint arXiv:1602.04283*, 2016 (cit. on p. 16).

[82]A. Putnam, A. Caulfield, E. Chung, *et al.*, *A reconfigurable fabric for accelerating large-scale datacenter services*, Jun. 2014 (cit. on p. 17).

[83]K. Ovtcharov, O. Ruwase, J.-Y. Kim, *et al.*, *Accelerating deep convolutional neural networks using specialized hardware*, Feb. 2015 (cit. on pp. 17, 18).

[86]Y. Chen, T. Luo, S. Liu, *et al.*, „DaDianNao: A machine-learning supercomputer“, in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2014, pp. 609–622 (cit. on p. 17).

[87]L. Cavigelli, D. Gschwend, C. Mayer, *et al.*, „Origami: A convolutional network accelerator“, in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15, Pittsburgh, Pennsylvania, USA: ACM, 2015, pp. 199–204 (cit. on p. 17).

[89]Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, „Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks“, in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, IEEE Computer Society, 2016, pp. 262–263 (cit. on p. 18).

[90]C. Zhang, P. Li, G. Sun, *et al.*, „Optimizing FPGA-based accelerator design for deep convolutional neural networks“, in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2015, pp. 161–170 (cit. on p. 18).

[91]M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, „Design space exploration of FPGA-based deep convolutional neural networks“, in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2016, pp. 575–580 (cit. on p. 18).

[96]J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, „Striving for simplicity: The all convolutional net“, *ArXiv preprint arXiv:1412.6806*, 2014 (cit. on pp. 23, 28).

[104]D. Mishkin, N. Sergievskiy, and J. Matas, „Systematic evaluation of CNN advances on the ImageNet“, *ArXiv e-prints*, Jun. 2016. arXiv: 1606.02228 (cit. on pp. 29, 82).

[110]C. R. Wan and D. J. Evans, „Nineteen ways of systolic matrix multiplication“, *International Journal of Computer Mathematics*, vol. 68, no. 1-2, pp. 39–69, 1998 (cit. on p. 36).

[111]N. Vasilache, J. Johnson, M. Mathieu, *et al.*, „Fast convolutional nets with fbfft: A GPU performance evaluation“, *CoRR*, vol. abs/1412.7580, 2014 (cit. on p. 36).

[115]R. Wittig (Distinguished Engineer at Xilinx), E-Mail Conversation, Subject “CNN on FPGA: HLS Obstacles”, 2016 (cit. on p. 53).

[129]A. Krizhevsky, „One weird trick for parallelizing convolutional neural networks“, *CoRR*, vol. abs/1404.5997, 2014 (cit. on p. 82).

Websites

[3]A. Karpathy. (2014). What I learned from competing against a ConvNet on ImageNet, [Online]. Available: http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/ (visited on Jul. 18, 2016) (cit. on pp. 1, 9).

[4]Auviz Systems. (2015). AuvizDNN: Convolutional neural networks | data center performance, [Online]. Available: http://auvizsystems.com/products/auvizdnn/ (visited on Jul. 19, 2016) (cit. on p. 2).

[5]——, (2015). AuvizDNN: Accelerating machine learning in the cloud: Deep neural networks on FPGAs, [Online]. Available: http://auvizsystems.com/applications/accelerating-machine-learning-in-the-cloud-deep-neural-networks-on-fpgas/ (visited on Jul. 19, 2016) (cit. on p. 2).

[7]Falcon Computing Solutions. (2016). Machine learning library: Accelerating machine learning algorithms using FPGAs, [Online]. Available: http://support.falcon-computing.com/demos/falcon_ml_datasheet.pdf (visited on Jul. 19, 2016) (cit. on p. 2).

[8]MulticoreWare, Inc. (2015). Multicoreware joins Xilinx alliance program, [Online]. Available: http://www.prweb.com/releases/2015/05/prweb12748587.htm (visited on Jul. 19, 2016) (cit. on p. 2).

[9]Altera Corporation. (2015). Altera OpenCL SDK: Machine learning - overview, [Online]. Available: https://www.altera.com/solutions/technology/machine-learning/overview.html (visited on Jul. 19, 2016) (cit. on p. 2).

[10]Xilinx, Inc. (2014). The Xilinx SDAccel development environment: Bringing the best performance/watt to the data center, [Online]. Available: http://www.xilinx.com/support/documentation/backgrounders/sdaccel-backgrounder.pdf (visited on Jul. 19, 2016) (cit. on p. 2).

[11]A. Karpathy. (2016). Stanford University CS231n: Convolutional Neural Networks for Visual Recognition, [Online]. Available: http://cs231n.github.io (visited on Jul. 14, 2016) (cit. on pp. 3–7).

[13]M. Nielsen. (2015). Online Book: Neural Networks and Deep Learning, [Online]. Available: http://neuralnetworksanddeeplearning.com/ (visited on Jul. 14, 2016) (cit. on pp. 3, 4, 6).

[14]Y. Jia, E. Shelhamer, J. Donahue, *et al.* (2016). Caffe tutorial, [Online]. Available: http://caffe.berkeleyvision.org/tutorial/ (visited on Jul. 14, 2016) (cit. on pp. 3, 7).

[27]The MathWorks, Inc. (2016). Neural Network Toolbox - MATLAB, [Online]. Available: http://mathworks.com/products/neural-network (visited on Jul. 18, 2016) (cit. on p. 8).

[39]J. Lim. (2016). Github caffe-googlenet-bn, Caffe implementation of GoogleNet, [Online]. Available: https://github.com/lim0606/caffe-googlenet-bn (visited on Jul. 18, 2016) (cit. on p. 10).

[51]J. Dean. (2014). Techniques and systems for training large neural networks quickly, [Online]. Available: http://stanford.edu/~rezab/nips2014workshop/slides/jeff.pdf (visited on Jul. 19, 2016) (cit. on p. 11).

[52]NVIDIA Corporation. (2015). GPU-based deep learning inference - Nvidia, [Online]. Available: https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf (visited on Jul. 19, 2016) (cit. on pp. 11, 66).

[61]Xilinx Inc. (2016). Xilinx: What is an FPGA? Field Programmable Gate Array, [Online]. Available: http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm (visited on Jul. 19, 2016) (cit. on p. 13).

[62]——, (2013). Xilinx UG998: Introduction to FPGA Design with Vivado High-Level Synthesis, [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf (visited on Jul. 19, 2016) (cit. on pp. 13, 14).

[63]——, (2016). Xilinx UG902: Vivado Design Suite User Guide, High-Level Synthesis, [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf (visited on Jul. 19, 2016) (cit. on pp. 14, 43, 44, 52).

[64]——, (2016). Xilinx buys high-level synthesis EDA vendor. Press release, [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1258504 (visited on Jul. 19, 2016) (cit. on p. 14).

[65]J. Cooley. (2014). Cadence to acquire Forte Cynthesizer at a rumored fire sale price. Press release, [Online]. Available: http://www.deepchip.com/items/0537-03.html (visited on Jul. 19, 2016) (cit. on p. 14).

[66]Synopsys, Inc. (2014). Synopsys acquires high-level synthesis technology from synfora, inc. Press release, [Online]. Available: http://news.synopsys.com/index.php?item=123168 (visited on Jul. 19, 2016) (cit. on p. 14).

[68]Cadence Design Systems, Inc. (2015). Using convolutional neural networks for image recognition, [Online]. Available: http://ip.cadence.com/uploads/901/cnn_wp-pdf (visited on Jul. 20, 2016) (cit. on p. 15).

[69]——, (2016). Cadence announces new tensilica vision p6 dsp targeting embedded neural network applications. Press release, [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/company/newsroom/press-releases/pr/2016/cadenceannouncesnewtensilicavisionp6dsptargetingembeddedneuralnetworkapplications.html (visited on Jul. 20, 2016) (cit. on p. 15).

[70]Movidius. (2016). Myriad 2 MA2x5x Vision Processor Product Brief, [Online]. Available: http://uploads.movidius.com/1463156689-2016-04-29_VPU_ProductBrief.pdf (visited on Jul. 20, 2016) (cit. on p. 15).

[71]NVIDIA Corporation. (2016). NVIDIA GeForce GTX Titan X Secifications, [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications (visited on Jul. 20, 2016) (cit. on p. 15).

[72]——, (2016). NVIDIA Tegra X1 Whitepaper, [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf (visited on Jul. 20, 2016) (cit. on p. 15).

[73]——, (2015). NVIDIA Launches Tegra X1 Mobile Super Chip. Press release, [Online]. Available: http://nvidianews.nvidia.com/news/nvidia-launches-tegra-x1-mobile-super-chip (visited on Jul. 20, 2016) (cit. on p. 16).

[74]Xilinx Inc. (2016). Ultrascale+ FPGAs Product Tables and Product Selection Guide, [Online]. Available: http://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf (visited on Jul. 20, 2016) (cit. on p. 16).

[75]M. Parker, Altera Corporation. (2014). Understanding Peak Floating-Point Performance Claims, [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf (visited on Jul. 20, 2016) (cit. on pp. 16, 23).

[76]Xilinx Inc. (2016). DSP Solution | Maximum DSP Capabilities, [Online]. Available: http://www.xilinx.com/products/technology/dsp.html (visited on Jul. 20, 2016) (cit. on pp. 16, 23, 33).

[77]——, (2016). Performance and resource utilization for floating-point v7.1, [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html (visited on Jul. 29, 2016) (cit. on pp. 16, 32).

[78]——, (2016). Xilinx Power Estimator (XPE) 2016.2 Virtex UltraScale+, [Online]. Available: http://www.xilinx.com/publications/technology/power-tools/UltraScale_Plus_XPE_2016_2.xlsm (visited on Jul. 20, 2016) (cit. on p. 16).

[80]M. P. Drumond. (2016). Navigating the design space of reconfigurable neural networks accelerators, [Online]. Available: http://wiki.epfl.ch/edicpublic/documents/Candidacy%20exam/PR15drumond.pdf (visited on Jul. 20, 2016) (cit. on p. 16).

[84]Altera Corporation. (2014). Altera and Baidu collaborate on FPGA-based acceleration for cloud data centers. Press release, [Online]. Available: http://newsroom.altera.com/press-releases/altera-baidu-fpga-cloud-data-centers.htm (visited on Jul. 20, 2016) (cit. on p. 17).

[85]S. Higginbotham, The Next Platform. (2016). Google takes unconventional route with homegrown machine learning chips, [Online]. Available: http://www.nextplatform.com/2016/05/19/google-takes-unconventional-route-homegrown-machine-learning-chips/ (visited on Jul. 20, 2016) (cit. on p. 17).

[88]Integrated Systems Laboratory, D-ITET, ETH Zürich. (2016). IIS Projects Proposal: FPGA System Design for Computer Vision with Convolutional Neural Networks, [Online]. Available: http://iis-projects.ee.ethz.ch/index.php/FPGA_System_Design_for_Computer_Vision_with_Convolutional_Neural_Networks (visited on Jul. 20, 2016) (cit. on p. 17).

[92]Supercomputing Systems AG. (2014). SCS ZYNQ-BOX Quick Start Guide, [Online]. Available: http://www.scs.ch/fpgabox (visited on Jul. 21, 2016) (cit. on pp. 19, 23, 31).

[93]S. Dasgupta. (2016). Netscope neural network visualizer, [Online]. Available: https://github.com/ethereon/netscope (visited on Jul. 21, 2016) (cit. on p. 22).

[94]D. Gschwend. (2016). Netscope CNN Analyzer, [Online]. Available: http://dgschwend.github.io/netscope/ (visited on Jul. 21, 2016) (cit. on p. 22).

[95]——, (2016). dgschwend/netscope: Neural network visualizer and analyzer, [Online]. Available: https://github.com/dgschwend/netscope (visited on Jul. 22, 2016) (cit. on p. 22).

[97]NVIDIA Corporation. (2016). NVIDIA DIGITS DevBox, [Online]. Available: https://developer.nvidia.com/devbox (visited on Jul. 25, 2016) (cit. on p. 25).

[98]Quora. (2016). I want to build a workstation for deep learning practice. what are some suggestions of what to buy?, [Online]. Available: https://www.quora.com/I-want-to-build-a-workstation-for-deep-learning-practice-What-are-some-suggestions-of-what-to-buy (visited on Jul. 25, 2016) (cit. on p. 26).

[99]R. Pieters. (2015). Building a deep learning (dream) machine, [Online]. Available: http://graphific.github.io/posts/building-a-deep-learning-dream-machine/ (visited on Jul. 25, 2016) (cit. on p. 26).

[100]Wired. (2014). The AI Startup Google Should Probably Snatch Up Fast, [Online]. Available: http://www.wired.com/2014/07/clarifai/ (visited on Jul. 25, 2016) (cit. on p. 26).

[101]F. Iandola. (2016). Github DeepScale/SqueezeNet, squeezenet_v1.1, [Online]. Available: https://github.com/DeepScale/SqueezeNet/tree/master/SqueezeNet_v1.1 (visited on Jul. 26, 2016) (cit. on p. 26).

[102]A. Karpathy. (2016). CS231n neural networks part 3: Learning and evaluation, [Online]. Available: http://cs231n.github.io/neural-networks-3/ (visited on Jul. 25, 2016) (cit. on pp. 27, 82).

[103]Xilinx Inc. (2016). Zynq-7000 All Programmable SoC Overview, [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (visited on Jul. 27, 2016) (cit. on pp. 29, 33).

[105]D. Mishkin. (2016). Github DeepScale/SqueezeNet, issue #3 squeezenet benchmark, [Online]. Available: https://github.com/DeepScale/SqueezeNet/issues/3 (visited on Jul. 27, 2016) (cit. on p. 29).

[106]Wikipedia. (2016). Single-precision floating-point format, [Online]. Available: https://en.wikipedia.org/wiki/Single-precision_floating-point_format (visited on Jul. 31, 2016) (cit. on p. 32).

[107]Altera Corp. (2016). Arria 10 hard floating point DSP block, [Online]. Available: https://www.altera.com/products/fpga/features/dsp/arria10-dsp-block.html (visited on Jul. 31, 2016) (cit. on p. 32).

[108]Xilinx Inc. (2016). 7 series DSP48E1 slice, [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf (visited on Jul. 31, 2016) (cit. on p. 33).

[109]A. Karpathy. (2016). CS231n convolutional neural networks: Architectures, convolution / pooling layers, [Online]. Available: http://cs231n.github.io/convolutional-networks/ (visited on Jul. 30, 2016) (cit. on p. 35).

[112]Berkeley Vision and Learning Center (BVLC). (2016). Github BVLC/caffe, issue #2513 Caffe CPU convolution, [Online]. Available: https://github.com/BVLC/caffe/issues/2513 (visited on Aug. 9, 2016) (cit. on p. 36).

[113]Xilinx Inc. (2016). UG1197: Ultrafast high-level productivity design methodology guide, [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf (visited on Jul. 29, 2016) (cit. on pp. 42, 43).

[114]——, (2016). Vivado design suite tutorial high-level synthesis, [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug871-vivado-high-level-synthesis-tutorial.pdf (visited on Aug. 2, 2016) (cit. on pp. 44, 55).

[116]Texas Instruments Incorporated. (2016). PMP8251: Power Solution for Xilinx FPGA Zynq 7, [Online]. Available: http://www.ti.com/tool/pmp8251 (visited on Aug. 9, 2016) (cit. on p. 66).

[117]Xilinx Inc. (2016). Xilinx Power Estimator (XPE) 2016.1 Zynq-7000, [Online]. Available: http://www.xilinx.com/publications/technology/power-tools/7_Series_XPE_2016_1.xlsm (visited on Aug. 10, 2016) (cit. on p. 66).

[118]Gigabyte Technology Co., Ltd. (2015). Gv-ntitanxxtreme-12gd-b xtreme gaming graphics card, [Online]. Available: http://www.gigabyte.de/products/product-page.aspx?pid=5684 (visited on Jul. 25, 2016) (cit. on p. 76).

[119]Stanford Vision Lab. (2012). ImageNet Large Scale Visual Recognition Challenge 2012, [Online]. Available: http://image-net.org/challenges/LSVRC/2012/ (visited on Jul. 25, 2016) (cit. on p. 80).

[120]Berkeley Vision and Learning Center (BVLC). (2016). Github BVLC/caffe: get_ilsvrc_aux.sh, [Online]. Available: https://github.com/BVLC/caffe/blob/master/data/ilsvrc12/get_ilsvrc_aux.sh (visited on Jul. 25, 2016) (cit. on p. 80).

[121]NVIDIA Corporation. (2016). Github NVIDIA/DIGITS: Pull Request #777 Torch Data Augmentation, [Online]. Available: https://github.com/NVIDIA/DIGITS/pull/777 (visited on Jul. 25, 2016) (cit. on pp. 80, 81).

[122]groar. (2015). Github NVIDIA/DIGITS: Pull Request #330 Support for train data augmentation, [Online]. Available: https://github.com/NVIDIA/DIGITS/pull/330 (visited on Jul. 25, 2016) (cit. on p. 80).

[123]Berkeley Vision and Learning Center (BVLC). (2016). Model zoo, [Online]. Available: https://github.com/BVLC/caffe/wiki/Model-Zoo (visited on Jul. 25, 2016) (cit. on p. 80).

[124]K. He. (2016). Deep Residual Networks for Caffe, [Online]. Available: https://github.com/KaimingHe/deep-residual-networks (visited on Jul. 25, 2016) (cit. on p. 80).

[125]I. M. Haloi. (2016). Google Inception V3 for Caffe, [Online]. Available: https://github.com/n3011/Inception_v3_GoogLeNet (visited on Jul. 25, 2016) (cit. on p. 80).

[126]S. Michalowski. (2016). Google Inception V3, Inception V4, Inception-ResNet for Caffe, [Online]. Available: https://github.com/soeaver/caffe-model (visited on Jul. 25, 2016) (cit. on p. 80).

[127]NVIDIA Corporation. (2016). Github NVIDIA/caffe: caffe.proto, [Online]. Available: https://github.com/NVIDIA/caffe/blob/caffe-0.14/src/caffe/proto/caffe.proto (visited on Jul. 25, 2016) (cit. on p. 81).

[128]R. K. Srivastava. (2016). Github flukeskywalker/highway-networks: Highway Networks for Caffe, [Online]. Available: https://github.com/flukeskywalker/highway-networks (visited on Jul. 25, 2016) (cit. on p. 81).

[130]NVIDIA Corporation. (2016). Github NVIDIA/DIGITS, issue #694 error running inference on cpu for network with batchnorm, [Online]. Available: https://github.com/NVIDIA/DIGITS/issues/694 (visited on Jul. 25, 2016) (cit. on p. 82).

[131]——, (2016). Github NVIDIA/DIGITS, issue #629 batchnorm does not converge in digits, [Online]. Available: https://github.com/NVIDIA/DIGITS/issues/629 (visited on Jul. 25, 2016) (cit. on p. 82).

[132]——, (2016). Github NVIDIA/caffe, issue #194 resnet-50 testing, [Online]. Available: https://github.com/NVIDIA/caffe/issues/194 (visited on Jul. 25, 2016) (cit. on p. 82).