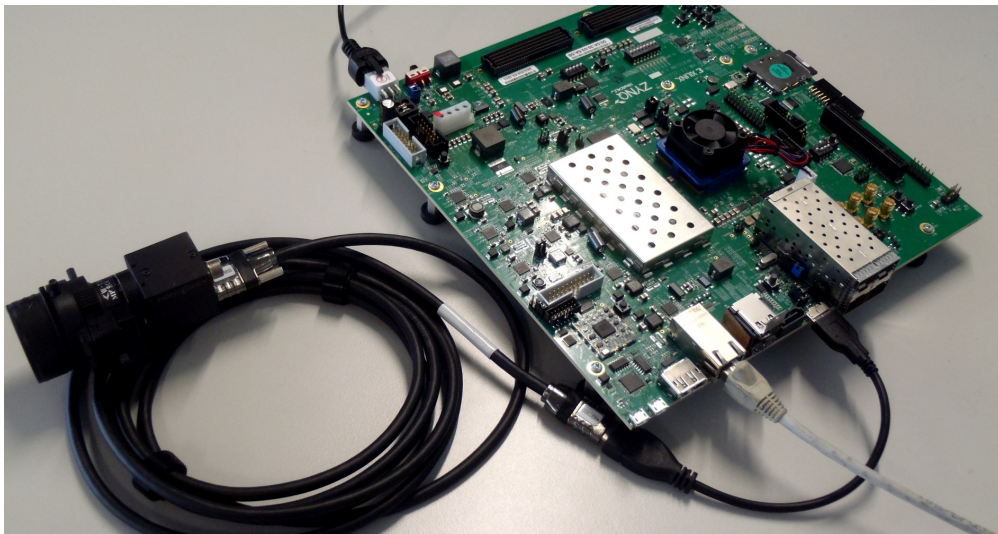


DEPARTMENT OF INFORMATION TECHNOLOGY AND  
ELECTRICAL ENGINEERING

Spring Semester 2017

# Development of an Embedded Object Detection System on the Zynq SoC Platform

Master Thesis



Philippe Degen  
degenp@student.ethz.ch

July 2017

Supervisors: Lukas Cavigelli, [cavigelli@iis.ee.ethz.ch](mailto:cavigelli@iis.ee.ethz.ch)  
Renzo Andri, [renzo.andri@iis.ee.ethz.ch](mailto:renzo.andri@iis.ee.ethz.ch)  
Professor: Prof. Dr. Luca Benini, [lbenini@iis.ethz.ch](mailto:lbenini@iis.ethz.ch)

# Acknowledgments

For the assistance on my master project I would especially like to thank my two supervisors Lukas Cavigelli and Renzo Andri, who were always able to help me in multiple situations and pointing out interesting ideas for the project. Further I would like to thank Alessandro Capotondi for his help with the OpenMP API in order to get it running, as well as Pirmin Vogel for the help with the power measurement setting. Also the friendly help by Hansjoerg Gisler with the power adapter and the help of the IT support group on my issues with the different programs should be mentioned. Finally I want to thank the *Integrated Systems Laboratory* and Prof. Benini for giving me the opportunity to pursue multiple very interesting and instructive projects in the group over the period of my Master's studies.

# Abstract

In the context of rising interest in object detection for embedded applications a proper implementation is developed on a Zynq Ultrascale+ board by Xilinx and later benchmarked.

After an algorithm evaluation SqueezeDet is selected as suited algorithm making it possible to realise a high performance, energy efficient implementation for object detection. Later a quantisation is applied to the network with a final setting working on 8-bit fixed point numbers for the weights and intermediate activations and 16-bit quantisation on the activations of the last convolution layer.

The final design features two hardware accelerators on the programmable logic of the SoC for 1x1 convolutions and 3x3 convolutions. With a parallelism of 1024 and 128 compute units they achieve a peak performance on the last fire layer of 193 GMAC/s and 32 GMAC/s respectively. The peak energy efficiency of the 3x3 convolution accelerator exceeds 30 GOp/s/W. The full network eventually runs in 190ms leading to 5 frames per second and consumes 10.9 W for the computation, keeping the accuracy of the detection at a level of over 79% mAP on the KITTI dataset.

Further comparisons on performance as well as on an energy efficiency measure are conducted with reference implementations on the chip including OpenMP multiprocessing and the ARM compute library. Additionally to that the overall performance is also compared to different platforms featuring other FPGA SoC and mobile GPU SoCs.

# Declaration of Originality

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. For a detailed version of the declaration of originality, please refer to Appendix B

Philippe Degen,  
Zurich, July 2017

# Contents

<b>Acronyms</b>	<b>x</b>
<b>1. Introduction</b>	<b>1</b>
<b>2. Background and Related Work</b>	<b>3</b>
2.1. Evolution of Object Detection Algorithms . . . . .	3
2.2. Optimisations for Embedded Applications and Hardware Implementations	5
2.2.1. Optimisation Schemes . . . . .	5
<b>3. System and Hardware Architecture</b>	<b>8</b>
3.1. Design Goal . . . . .	8
3.2. System Architecture . . . . .	8
3.2.1. Algorithm Selection . . . . .	8
3.2.2. SqueezeDet . . . . .	10
3.2.3. Network Quantisation . . . . .	10
3.3. Hardware Architecture . . . . .	14
3.3.1. Device Properties . . . . .	14
3.3.2. Algorithm Processing Load . . . . .	15
3.3.3. Data Transfer to Programmable Logic . . . . .	16
3.3.4. Single 3x3 Convolution Accelerator . . . . .	16
3.3.5. Full Expand Layer Accelerator with Single Convolution Accelerator	22
3.3.6. Full Expand Layer Accelerator with separate Convolution Accelerators . . . . .	22
3.3.7. Separate 1x1 and 3x3 Convolution Accelerators . . . . .	23
<b>4. Design Implementation - Embedded Object Detection Platform</b>	<b>26</b>
4.1. Xilinx ZCU 102 Zynq Board . . . . .	26
4.2. Vivado HLS & SDSoC . . . . .	27
4.3. Petalinux . . . . .	27
4.4. Point Grey Flea 3 Camera . . . . .	27

## Contents

4.5.	Output Transfer over Network . . . . .	28
4.6.	Algorithm Implementations . . . . .	28
4.6.1.	Data Arrangement . . . . .	29
4.6.2.	Basic C Implementation . . . . .	29
4.6.3.	OpenMP Implementation . . . . .	30
4.6.4.	Platform Implementation . . . . .	30
4.6.5.	Hardware Accelerator for 3x3 Convolution . . . . .	31
4.6.6.	Hardware Accelerator for Total Expand Layer . . . . .	31
4.6.7.	Two Separate Hardware Accelerators for 1x1 and 3x3 Convolution . . . . .	32
4.6.8.	General Algorithm Optimisations . . . . .	32
4.6.9.	Optimisation of the Last ConvDet Convolution . . . . .	37
4.6.10.	Reference Implementation with ARM Compute Library . . . . .	38
4.7.	Power Measurement Setting . . . . .	39
4.8.	Confinements of using a HLS Synthesis Tool combined with a High-Level System Design Platform like SDSoC . . . . .	40
<b>5.</b>	<b>Results</b>	<b>43</b>
5.1.	Network Quantisation . . . . .	43
5.1.1.	Training and Evaluation Environment . . . . .	43
5.1.2.	Quantisation Precision . . . . .	43
5.1.3.	Bit-Precision Choice . . . . .	44
5.1.4.	Fine-Tuning . . . . .	44
5.1.5.	Including Activation Quantisation in the Learning Process . . . . .	45
5.1.6.	Final Results used for the Platform Design . . . . .	46
5.2.	Algorithm Implementation . . . . .	48
5.2.1.	Pure Software Implementations . . . . .	48
5.2.2.	Hardware-Accelerated Implementations . . . . .	50
5.2.3.	Tracing . . . . .	51
5.2.4.	Power Measurements . . . . .	54
5.2.5.	Comparisons . . . . .	55
5.2.6.	Final Algorithm Performance . . . . .	57
<b>6.</b>	<b>Conclusion and Future Work</b>	<b>58</b>
6.1.	General Conclusion . . . . .	58
6.2.	Accelerator Utilisation . . . . .	58
6.3.	Memory Arrangement . . . . .	59
6.4.	Future HLS Releases . . . . .	60
6.5.	ARM Compute Library with Fixed Point Support . . . . .	60
6.6.	DSP Slices with Logic Shift Support . . . . .	60
<b>A.</b>	<b>Task Description</b>	<b>61</b>
<b>B.</b>	<b>Declaration of Originality</b>	<b>68</b>

# List of Figures

2.1. General Algorithm Structure of Faster R-CNN and comparable Algorithms	5
2.2. General Algorithm Structure of YOLO and comparable Algorithms . . . .	5
3.1. Fire Module . . . . .	10
3.2. SqueezeDet Network Structure . . . . .	11
3.3. Fixed Point Representation . . . . .	13
3.4. Block Diagram of DSP48E2 Slice in Xilinx Ultrascale Architecture . . . .	15
3.5. Block Diagram of 3x3 Convolution Hardware Accelerator . . . . .	20
3.6. Input Buffer Memory Layout and Processing Sequence . . . . .	21
3.7. Processing Pipeline of the 3x3 Convolution Accelerators Most Inner Loop	22
3.8. Block Diagram of 1x1 Convolution Hardware Accelerator . . . . .	25
4.1. Platform Setting with Board, Camera and Network Connection . . . . .	29
4.2. Power Plug Current Sense Setting . . . . .	40
5.1. Accuracy of SqueezeDet Network after Different Weight Quantisation on Training Set . . . . .	44
5.2. Accuracy Progress of Fine-Tuning 8-bit Quantised Weights . . . . .	45
5.3. Fine-Tuning with 8-bit Quantised Output Layer Activation . . . . .	47
5.4. Fine-Tuning with 11-bit Quantised Output Layer Activation . . . . .	47
5.5. Fine-Tuning with 16-bit Quantised Output Layer Activation . . . . .	47
5.6. Final Fine-Tuning with Quantised Weights and 16-Bit Quantised Last Activation . . . . .	48
5.7. Runtime vs. Computation Share of Floating Point Implementation . . . .	49
5.8. Performance of the Hardware Accelerators on the Different Layers . . . .	51
5.9. 1x1 Convolution Squeeze Layer Roofline Plot . . . . .	52
5.10. 3x3 Convolution Expand Layer Roofline Plot . . . . .	52
5.11. Example Trace of Partial Image Processing of L1 on 3x3 Convolution Accelerator . . . . .	53
5.12. Oscilloscope Chart for Total Power Measurement . . . . .	55

*List of Figures*

5.13. Processing Efficiency of different Calculation Schemes . . . . .	56
5.14. Computation Time Share of Final Implementation . . . . .	57



# List of Tables

3.1. Zynq UltraScale+ EG Programmable Logic Properties . . . . .	15
3.2. Computation Time Share Breakdown per Layer Type of SqueezeDet . . .	16
4.1. Main Available Power Domains on the PMBus . . . . .	39
5.1. Accuracy Impact of Activation Quantisation [mAP] . . . . .	45
5.2. Accuracy of Final Network compared with Initial Network and Untrained Network [mAP] . . . . .	47
5.3. Runtimes for Total Network and for Layer 7 . . . . .	49
5.4. Degrees of Parallelism in Convolution Accelerators . . . . .	50
5.5. Resource Use for Final Design . . . . .	50
5.6. Power Measurement over PMBus in W . . . . .	54
5.7. Comparison of Different Platform Performances . . . . .	56
5.8. Final Design Performance for the Full Network . . . . .	57

# List of Acronyms

ARM . . . . .ARM Ltd. (Advanced RISC Machines)

BRAM . . . . .Block RAM

CNN . . . . .Convolutional Neural Network

EDA . . . . .Electronic Design Automation

FPGA . . . . .Field-Programmable Gate Array

GPU . . . . .Graphics Processor Unit

HLS . . . . .High-Level Synthesis

IC . . . . .Integrated Circuit

IIS . . . . .Integrated Systems Laboratory

IO or I/O . . . .Input and Output

LUT . . . . .Look-up Table

MAC . . . . .Multiply and Accumulate

PL . . . . .Programmable Logic

PS . . . . .Processing System

ReLU . . . . .Rectifier Linear Unit

## *Acronyms*

RTL . . . . .Register Transfer Level

SIMD . . . . .Single-Instruction Multiple-Data

SoC . . . . .System on Chip

# Introduction

Looking at the general applications of image understanding and especially in object detection, most of the possible applications find themselves in an embedded surrounding. Given that embedded systems often come with a lot of design restrictions, often the power budget is a critical objective. We state three areas where embedded object has a big potential and an efficient implementation gives crucial benefits.

Firstly, in the area of surveillance and monitoring a big potential is around and to some extent already applied. Being it surveillance for security reasons on private or public ground or for general monitoring of animals, passenger flows, traffic or customer behaviour, the application happens always remotely in smart cameras that have tight design constraints. The second application is automotive scene understanding, that enables further security assistance to the driver or even up to autonomous driving that would lead to a new concept of personal transportation. Also in this case the implementation comes with various constraints and a special focus on energy consumption, since energy is rather scarce and temperature control is a challenging task as well. Eventually a third major application field is the aerial vision that is mostly installed on modern UAV systems such as drones. Here the energy budget is very tight since all the energy used for vision processing diminishes the motion range quite drastically.

Given the low energy consumption guideline the system is still expected to deliver highest algorithmic quality in order to give satisfying results in their respective products. Looking at recent trends in computer vision only convolutional neural networks (CNN) featuring very deep structures running thousands of convolutions on an image give state of the art results. However since the vast amount of convolutions implies a huge computation load still a processing system with a very high performance is needed in order to achieve the desired (near) real-time processing.

Next to embedded or mobile processors including GPUs, FPGAs have promising specifications enabling to run parts of algorithms on dedicated hardware and thereby reducing

## *1. Introduction*

drastically the power consumption. Modern implementations realise FPGAs as part of complete Systems-on-Chip (SoC) that incorporate multiple full processor cores and the necessary support for periphery. In this work the recent SoC by Xilinx is used in combination with a Xilinx development board in order to design a highly efficient object detection platform.

For the implementation the Xilinx tool suite containing SDSoC and Vivado HLS was used, enabling a fast implementation with the help of high-level synthesis.

In a first stage different algorithms for object detection are analysed and compared for their suitability for an embedded object detection platform including detection quality, processing load and memory requirements. In a second stage a quantisation scheme is introduced to the algorithm choice and the network is quantised in order to improve the processing performance. Eventually a hardware accelerator is designed for accelerating the algorithm on programmable logic before comparing the results to reference implementations of the algorithm without hardware accelerator but purely running on the processing system (PS) of the SoC.

# Chapter 2

## Background and Related Work

This chapter lays out the current progress and evolution of the object detection algorithms implemented in the appearance of convolutional neural networks (CNN). Prior to it, there were other implementations based on older computer vision techniques, that were not able to achieve similar performances on the same task. Further some optimisation schemes for embedded applications are presented.

### 2.1. Evolution of Object Detection Algorithms

With the massive growth of available computing performance in recent years the approach with ConvNets has had a powerful rise in the area of recognition. Under the name of *Deep Learning* large convolutional neural networks performed very well in image classification task, such that they were more and more also introduced in applications such as scene labelling or object detection.

The base of the ConvNets originates mostly from image classification tasks and gets tuned to also perform the additional tasks. In this section we will focus on the application of object detection since it is the application implemented later.

A first significant implementation of object detection with ConvNets was done by R. Girshik et al. called *R-CNN* [1]. It stands for region-based CNN. The chosen approach is to first compute a region proposal with known algorithms such as selective-search, objectness or CPMC, and later perform feature extraction in the proposed regions of the image with a classical ConvNet analog to the image classification task. This approach would already clearly outperform earlier algorithms. Later it was followed by a first optimisation called *Fast R-CNN* [2] by the same author. The main difference was that the region proposals would only be used after the image had passed an initial ConvNet, that is it is used on a feature map. A *Region-of-Interest (RoI)* pooling layer uses the

## 2. Background and Related Work

region proposals to later perform the final classification of the found object. Since the classification starts from a advanced feature map, the region-proposal specific computation is significantly smaller compared to a full classification task. For the region proposal however the still non-convolutional selective-search approach was mainly used. The main improvements were in the area of training and testing speed as well as in the achieved accuracy.

Again this implementation was further improved towards an algorithm called *Faster R-CNN* [3]. The main bottleneck that was optimised was the region-proposal based on the selective-search approach, that would need more time compared to the ConvNet evaluations. The approach was to also implement the region-proposal using ConvNets in a so called *Region Proposal Network* (RPN). Since this approach is fully convolutional, resources could be shared. This was achieved in the way that the region-proposal was no longer computed completely in parallel to the feature extraction task, but it would share a large part of the computation so that the specific RPN would compute its proposals starting from an intermediate feature map that would also later on be used as a starting point for the classification task. This approach was able to further lower the computational complexity while keeping the object detection accuracy or even improving it by a small amount.

Faster R-CNN was then improved by a new general approach of applying *Residual Networks* (ResNet) [4]. The chosen approach is to introduce shortcuts in the process flow in order to target only the value differences from an original feature map instead of the full values. These shortcuts showed to be useful in order to improve training speed as well as the overall accuracy of deep ConvNets. The approach is generally used for image classification but consequently also applied to the object detection task based on *Faster R-CNN* that had initially used a ConvNet based on the VGG-16 network [5].

Other improvements based on the state-of-the-art architecture of *Faster R-CNN* were also published last year, which concentrate on the optimisation of the region-proposal task. The main improvements were to increase the ability of detecting regions or objects of a large variety of sizes. This was mainly achieved by processing different sized feature maps along the ConvNet in the publications *MS-CNN* [6] and *PVANet* [7]. Also small adjustments in the Network design should enhance computational effort as described in the respective documentation. The *R-FCN* [8] approach would build its network fully convolutional to speed up the computation by reducing the classification network evaluations on the region of interest to a single run achieving considerable accuracy results.

Following the other goal of a lightweight implementation for object detection a handful of algorithms based on a single neural network were proposed in parallel to the evolution of the performance-driven algorithms. The particularity of the algorithms was that the full object detection is performed by a single network that can be trained and tested in an end-to-end manner. That is the bounding boxes and the class probabilities are computed directly from the image in one evaluation. The first approach proposed was

## 2. Background and Related Work

*You only look once (YOLO)* [9]. They achieved over real-time computation rates (45 fps) with a notable drawback in accuracy. But others followed improving the accuracy keeping the rates in real-time orders. *Single Shot Multibox Detector (SSD)* [10] increased the accuracy to Faster R-CNN level and later B. Wu et al. proposed the *SqueezeDet* [11] approach that originates on the *SqueezeNet* [12] architecture. They claim to achieve similar performance to SSD in terms of accuracy and computation rates with a very tiny ConvNet model. The SqueezeNet architecture stands out with its very small model size. The additional network in SqueezeDet in order to perform object detection is designed with a simple convolution layer that keeps the overall architecture very small.

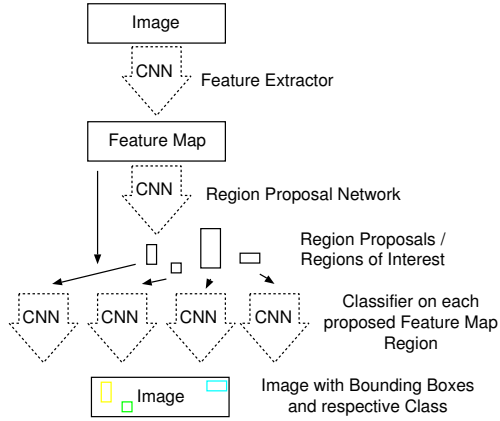


Figure 2.1.: General Algorithm Structure of Faster R-CNN and comparable Algorithms

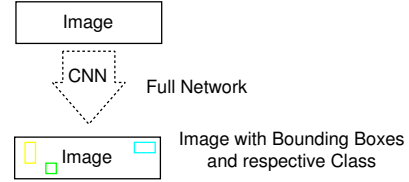


Figure 2.2.: General Algorithm Structure of YOLO and comparable Algorithms

## 2.2. Optimisations for Embedded Applications and Hardware Implementations

### 2.2.1. Optimisation Schemes

In the area of embedded applications for object detection an efficient computation as well as an energy optimised scheme is crucial to meet performance goals. At this point it is to state that these optimisations are thought to be conducted only on the inference task of ConvNets. This is mainly due to the application in embedded systems where the training task does not make any sense. On the other hand small imprecisions have a smaller effect than in the training task. The general goal is to have a small amount of data in order to minimise data movement. This improves compute performance as well as energy consumption.

A key publication is the paper by Song Han et al. that introduces the *Deep Compression*



## 2. Background and Related Work

scheme. [13] It essentially combines three efficiency improvement concepts: pruning, quantisation and Huffman coding.

Pruning is the concept of eliminating unimportant neurons, i.e. compute connections, which have a neglecting effect on the computation results. This is mainly done by eliminating the network weights that are sufficiently small. This leads to a sparse structure of the network, that reduces the size of the network as well as the amount of computation. Nevertheless a small overhead is introduced to cope with the sparse structure.

Secondly quantisation is introduced, that uses the fact that most of the network weights as well as the feature map values are all situated in a rather small interval. This suggests a computation on an inferior precision data type. In [13] and other publications a precision of 8 bits showed to be sufficient for a negligible impact on the accuracy of the network. This consequently diminishes the size of the data and the computation effort.

A further concept of weight sharing on the quantisations is investigated by Song Han et al. where the values of the stored variables are coded to some *shared weights*. This allows to adapt to non-uniform distributions of weights to gain higher precisions in areas of higher value occurrences. This nevertheless showed to be less effective in terms of computation and accuracy than uniformly distributed values. (standard number coding) This is assumed to be the case due to the importance of large values in the network rather than good distinction of similar values.

The last scheme investigated by Song Han et al. is to code the weights as well as the sparse data indexes from pruning. Since both showed to have a clear biased distribution, Huffman coding allowed to lower the memory access over the network storage and therefore lower energy consumption.

Other mentionable quantisation suggestions and investigations have been done by Bert Moons et al. [14], Philipp Gysel et al. [15],[16] and Pete Warden [17]. The first two investigate improvements of quantisation respective to the quantisation range selected. One main finding is that a per layer quantisation improves accuracy and enables diminishing further the bit width by a significant amount. Generally it is shown that up to a certain level it can be quantised without substantial loss in accuracy whereas after a certain threshold the accuracy is strongly affected.

In order to smaller the accuracy loss of the quantisation step Philipp Gysel and Song Han et al. propose to perform an additional *fine-tuning* learning step after quantisation of the trained network. This basically retrains the network with the quantised weights in the forward run. Gysel states that better results are achieved if the weights in the back propagation are kept at full precision because the update of the weights after an iteration can be too small for a quantised value. Also the feature map values are kept at full precision in order to have the best precision for training.

Finally Pete Warden describes the quantisation procedure chosen in the *TensorFlow* framework. It doesn't include fine-tuning and does range estimation live. This doesn't sound to be very efficient. Further this approach doesn't use even distributions but

## *2. Background and Related Work*

calculates the minimum and the maximum of the layers data range and maps the interval onto 8 bits.

# Chapter 3

## System and Hardware Architecture

In this chapter the design choices are presented and the details of the final architecture are described.

### 3.1. Design Goal

As stated in the introduction the general design goal is to realise an object detection system with high state-of-the-art algorithmic quality that runs on an embedded system realising near real-time processing with a minimal power consumption. The goal implies various trade offs that have to be considered and is restricted in the sense that the embedded system or platform is given. Otherwise the following design choices should explain the conducted thoughts and bring the design to the desired place.

### 3.2. System Architecture

#### 3.2.1. Algorithm Selection

In chapter 2 the various algorithm proposals were shortly described and the evolution in the design briefly laid out. The ideal algorithm for an implementation on an embedded platform should be small in model size and fast in computation as described before. Still a reasonable to high accuracy should be achieved in order to produce advanced results for such a system. With this goal in mind it was quickly clear that we would focus on single shot algorithms that incorporate both the bounding-box as well as the class prediction in one single network. This makes the network more compact and keeps the model smaller. Eventually *SSD* [10] and *SqueezeDet* [11] were the two algorithms to investigate in detail. They both achieve similar performances in terms of accuracy and

### 3. System and Hardware Architecture

computation speed. Their performance is difficult to compare since no direct comparison is available from their publication. *SSD* makes performance measurements with the VOC dataset while *SqueezeDet* does measurements with the KITTI image set. Another obstacle is that image sizes are not the same, which makes a comparison further difficult. Making some cross-comparisons over the performances compared to the state-of-the-art algorithm *Faster R-CNN*, it can be stated that the performance can be estimated to be comparable.

To determine the suited algorithm the structures and computation elements were compared. Both share a general structure of a first part with a deep ConvNet before processing the resulting feature map through a combined detection and classification network. Eventually the results of the detection and classification network are further processed to extract the bounding boxes with the highest class probabilities and perform non-maximum suppression.

The ConvNet at the start of the network differs mainly in size. SqueezeDet understandably uses SqueezeNet and SSD uses VGG-16. Both are straight forward ConvNets with a sequential flow. SqueezeNet is designed to have a small model size which is also because of the replacement of normal convolution modules by fire modules. These make partly use of smaller filters and a smaller amount of channels. The main difference of both designs is the method of computing the bounding box prediction and the corresponding class probabilities. SqueezeDet has one single convolution stage that they call ConvDet in order to process the desired information. On the other hand SSD uses a much more complicated approach of 4 further convolution layers that stepwise reduce the size of the feature map in order to then apply the final processing on different feature map sizes. This is done to better recognise objects with a large variety of sizes. All these 4 different sized feature map and an intermediate feature map from inside the first ConvNet is then further processed in 3 different modules. One is for localisation information, one is for the confidence score of an object and the last is to generate *priors* which are a kind of default bounding boxes. Finally the whole information is fed into the last stage as described above. The approach is called *Multibox* and originates from earlier work on the subject by Erhan et al. [18] [19].

The decision for the **SqueezeDet** algorithm was mainly based on the following three observations. First, the model size of the SqueezeDet network is significantly smaller. This is important for a desired embedded FPGA design, since it enables that a larger part of the parameters can be stored on-chip. This not only improves performance but also the energy consumption for the computation. Secondly, the SqueezeDet architecture is very straight forward, so that the computation can be performed in a very much streaming kind of way. In contrary to this, SSD has multiple splittings with additional shortcuts in the flow that impose additional store and load executions in order to save intermediate results and reload them at a given time. Lastly the SqueezeDet algorithm consists mainly of pure convolution modules whereas SSD has some different computation parts in the second-last stage that include memory reordering. This makes it hard to make

### 3. System and Hardware Architecture

custom hardware for all processing stages whereas with purely convolutional modules the convolution can be accelerated to it's full extent.

#### 3.2.2. SqueezeDet

This section illustrates the main structure of the above discussed *SqueezeDet* neural network. In general the network follows the design of the initial *SqueezeNet* network for image recognition. The main building block of the network is the *fire unit*. Figure 3.1 illustrates the two layers of the fire unit that characterises it. The first convolution module is a  $1 \times 1$  convolution that has a lot of input channels and relatively few output channels. Therefore the naming *squeeze layer*. As an example the squeeze layer of fire unit number 5 takes 256 input channels while outputting 32 output channels. The second layer is called *expand layer* since in contrast the number of channels is again increased. Furthermore two parallel convolution modules, one with a  $1 \times 1$  kernel, the other with a  $3 \times 3$  kernel, both contribute to it. This is the case since all output channels are concatenated.

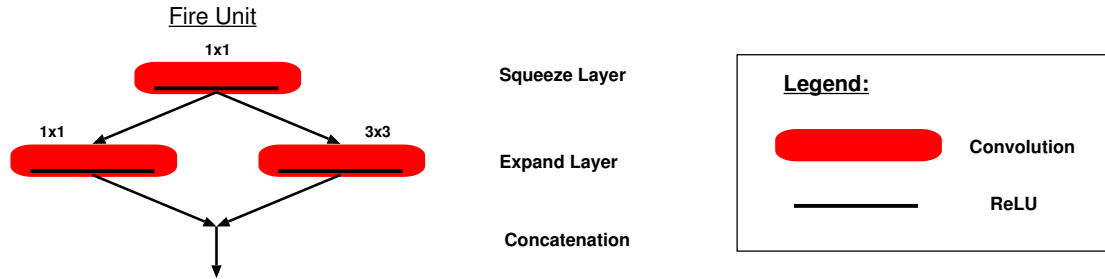


Figure 3.1.: Fire Module

*SqueezeDet* now strings multiple fire modules together, introducing 3 additional max-pooling layers in between. With this, the structure is very comparable to the *SqueezeNet* network. The final added feature to this structure is a further convolution module that is attached at the end. It extracts the final information used for object detection, that is, the bounding-box localisation, the class probability as well as a general objectivity rate. They called this layer *ConvDet*. After the convolutional network part, an interpretation graph is added, which performs a sigmoid, a softmax and a non-maximum suppression in order to extract the final bounding boxes with the highest probability. The total structure is summed up in figure 3.2.

#### 3.2.3. Network Quantisation

For our goal of a fast accelerated embedded object detection system quantisation showed to be ideal. On the one side smaller weight size would allow to better pack the full model into on-chip RAM which enables faster computation and much smaller energy

### 3. System and Hardware Architecture

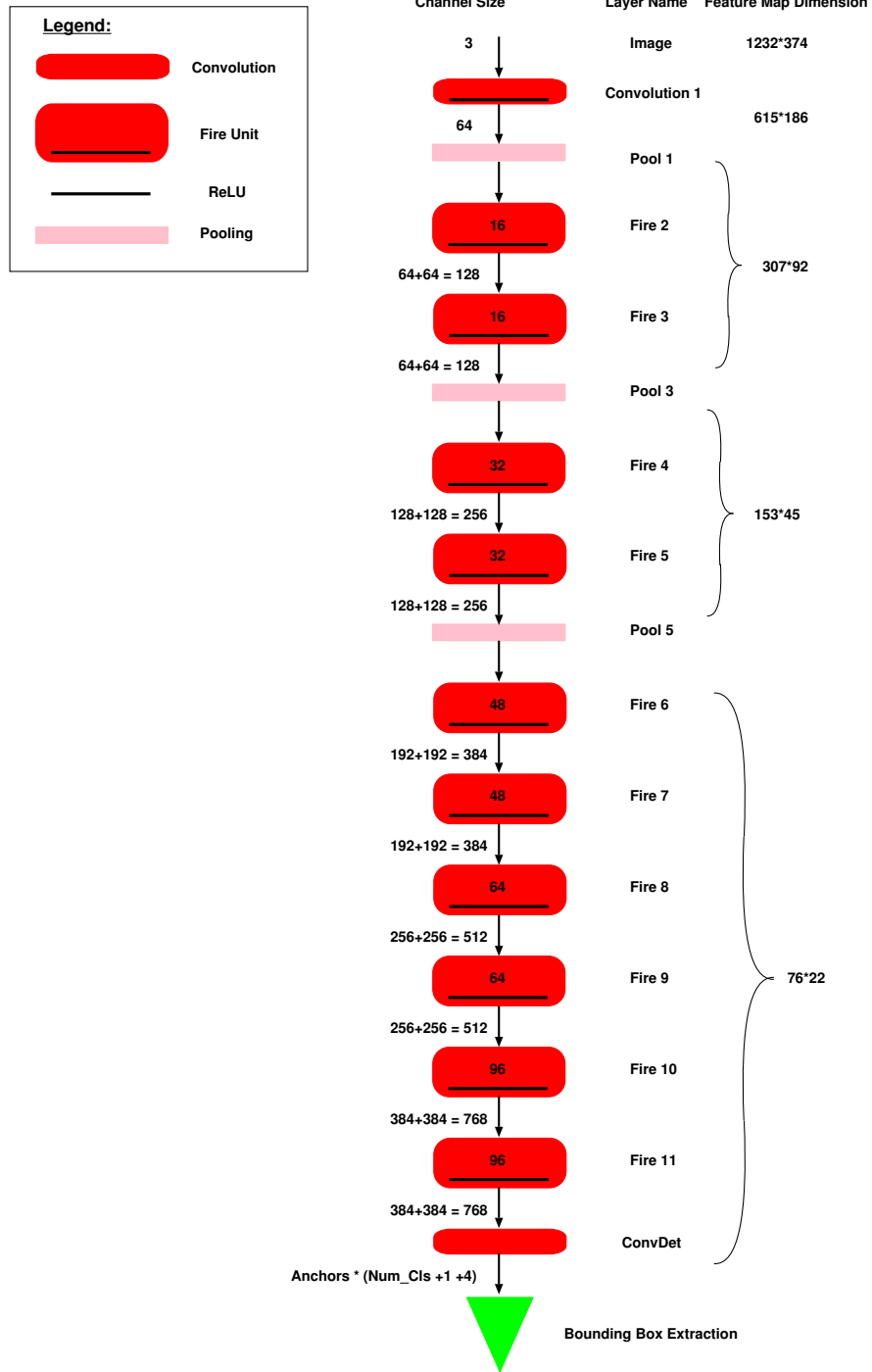


Figure 3.2.: SqueezeDet Network Structure

### 3. System and Hardware Architecture

consumption. Further the processing of small fixed-point values are much easier to handle for the FPGA’s DSP slices. This should also bring along a considerable performance gain.

Different approaches to quantisation have been discussed in chapter 2. For this application on an embedded SoC, the approach chosen by Ph. Gysel et al. [16] makes most sense. The live computation of fixed-point ranges like in [17] seems inefficient in an FPGA application. Since Ristretto by Ph. Gysel is a framework for *Coffee* we had to come up with our own implementation for a *Tensorflow* quantisation module.

For this purpose the existing implementation of the *SqueezeDet* network was modified to incorporate the new convolution and bias modules that feature quantisation. Both are designed in order to keep a set of full precision filter parameters but apply a quantised version in the forward path. In every forward run the weights are again quantized before they are applied in the convolution. In the backward propagation the full-precision weights are updated in order to also track small weights updates.

We chose to apply the principle of *dynamic fixed-point approximation* as described in [15]. That is, that each layer performs its processing on a different dynamic range. This allows the accuracy to stay high even for low precision data representations. The precision is the same for all layers, that is the bit-width for the fixed-point representation is kept the same. This makes sense in the outlook of using a hardware accelerator that is designed to run on a fixed bit-width. The difference between the layer is the number of integer bits or the position of the fractional point respectively.

The number of integer bits was determined using a pretrained set of weights. Following formula was used to determine the layer specific integer bit number.

$$\text{Integer bit size: } B_{int} = \lceil \log_2[\max(1.1 * |x|)] \rceil \quad (3.1)$$

This implementation keeps a ten percent margin for fine-tuning. According to [13] the largest values are very significant and should therefore not be truncated if possible.

The choice with *dynamic fixed-point* numbers assumes that the parameters are distributed around 0. That means that there is no significant bias. This is a fair assumption since it enables much easier computation compared to when a bias has to be added to each stored parameter.

In order to fine-tune the network after quantisation the learning rate is reduced by a factor of 10. This is described in [15] and enables more fine-grained parameter tuning.

#### Tensorflow Implementation

As mentioned earlier the used *SqueezeDet* network available in the *Tensorflow* framework was modified in order to perform quantisation on the forward path. Since the *SqueezeDet*

### 3. System and Hardware Architecture

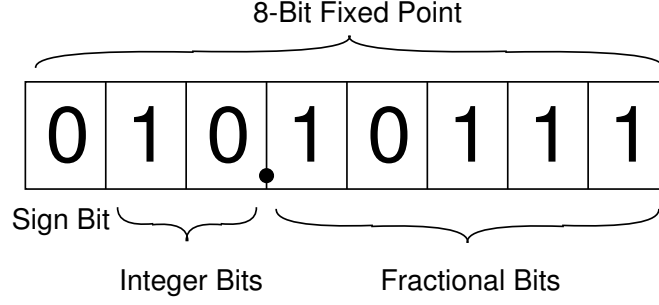


Figure 3.3.: Fixed Point Representation

network is purely convolutional, only a new convolution and a new bias module had to be designed when quantising the weights. Based on the implementation available in the open-source framework repository, new modules were implemented for CPU and GPU computation.

The adapted module copies the weight tensor and applies quantisation on the new tensor before feeding it to the convolution or bias computation. The whole process is kept in floating-point since GPUs work best with floating-point numbers and a framework support for fixed-point is not existent. This procedure is not a problem in this context since the operations in the network are only additions and multiplications.

The rounding in the quantisation step is done with *round nearest even*. The following equation shows the quantisation computation performed on the weights.

$$W_{quant} = \frac{round(W * 2^{B_{scale}})}{2^{B_{scale}}}, \text{ where } B_{scale} = B_{precision} - B_{integer} \quad (3.2)$$

Please note that the integer bit-width  $B_{integer}$  can also be a negative number in the case that the number is a low order fraction e.g.  $0.17 \rightarrow B_{integer} = -2$ . Further note that the amount of precision bits  $B_{precision}$  is one smaller than the fixed-point precision. This is due to the sign-bit as depicted in Figure 3.3.

In a further step quantisation was also introduced on the layer activations. The layer activations are the second multiplicand in the convolution's multiplication and therefore also very interesting to keep at a low bit-width. Additionally in a hardware perspective a lower bit-width on layer activations reduces the data transfer between compute unit and device memory. This has an impact especially on the power consumption.

In order to apply quantisation on the layer activation in the tensorflow model, we also introduced a module that simply quantises the input and outputs it. It was essential for us to perform the same computation as later on the platform and also to do training with quantised activations.



### 3. System and Hardware Architecture

The resulting findings on the impact of the quantisation on the network accuracy are summarised in chapter 5.

Concluding the results that we obtained we decided to go for a 8-bit quantisation scheme with an extended precision of the last layer (ConvDet) of 16 bits in order to achieve the best data precision and system accuracy trade off.

## 3.3. Hardware Architecture

The hardware architecture was designed with the help of the Xilinx design tools for high level system design on programmable logic. This is on the one side Vivado HLS compiler that generates a hardware design out of C code, and on the other side SDSoC that manages all hardware function calls and data transfers between the host processor and the hardware accelerator. In the following section the different approaches and their respective implications on the design and its performance are being laid out.

### 3.3.1. Device Properties

The employed programmable logic in the project is the programmable logic of a Zynq UltraScale+ EG SoC as described in the next chapter in more details. Its hardware resources are listed in table 3.1.

In general it is an FPGA of the Kintex family and a rather big FPGA for a Zynq SoC, with a large amount of on-chip memory and many DSP slices. These are both elements that help a lot in the computation of CNN since the main processing is a filter kind of processing that needs fast access to filter coefficients and performs DSP operations such as MAC's.

The FPGA incorporates DSP48E2 slices of the new Xilinx Ultrascale architecture that feature a 27 x 18 bit multiplier, a 48-bit accumulator and a logic unit as described in the block diagram in figure 3.4.

With these specifications this DSP architecture allows to compress computation for 8-bit calculations. For the quantised computing in CNNs as this project followed it, it is therefore possible to perform two 8-bit MAC's on a single DSP slice. The bottleneck in this scenario is then the accumulator that has to accumulate two separate values that need higher precision computation than the two factors of the multiplication. Xilinx suggests to make a trade-off of using 8 cascaded DSP slices to calculate 7·2 separate MAC's. This leads to a resource efficiency improvement of a factor 1.75. Further information can be gained from Xilinx's white paper on *INT8 Optimization for Deep Learning* [20].

### 3. System and Hardware Architecture

Type	Resources
System Logic Cells	600 K
LUT's	274 K
Flip-Flops	548 K
Block RAM	32.1 Mb
DSP Slices	2,520

Table 3.1.: Zynq UltraScale+ EG Programmable Logic Properties

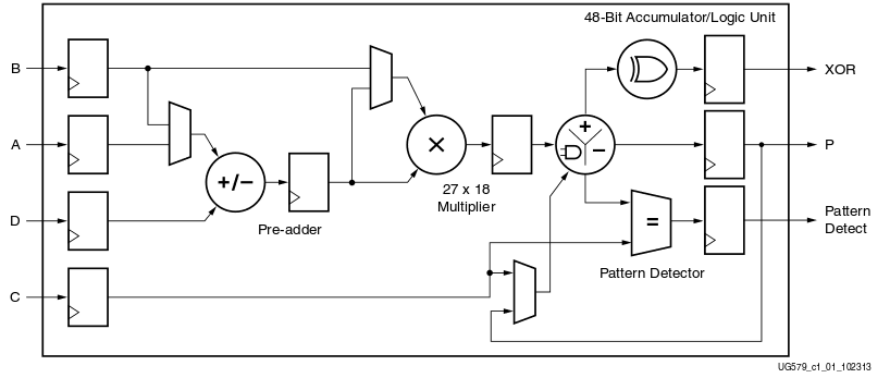


Figure 3.4.: Block Diagram of DSP48E2 Slice in Xilinx Ultrascale Architecture

#### 3.3.2. Algorithm Processing Load

The vast majority of the processing load of *SqueezeDet* are the convolutions. The main processing task of the convolution again, is the multiply - accumulate operation (MAC). In *SqueezeDet* the convolution load is mainly spread between 3x3 convolutions and 1x1 convolutions where most of them are inside the fire modules. Since the computation load rises quadratically with the kernel size the 3x3 convolutions in the expand layer of the fire modules make up nine times the computation amount compared to the parallel 1x1 module. With the additional first and last layer also being a 3x3 convolution module, the 3x3 convolution computation makes up a total of 82 % of the total computation load. Table 3.2 shows the more detailed computation breakdown. The categories in the table are chosen this way since our design choice as mentioned earlier was to compute the last *ConvDet* layer with higher 16-bit precision. The initial 3x3 convolution layer at the input is also noted separately since it performs a convolution that differs itself from the other as it has a stride of 2 and does not apply any padding. Only all the expand layer 3x3 convolutions perform the identical task sharing the stride, the padding, the activation function and the bit precision. This quickly led to the decision of accelerating the fire modules, particularly the 3x3 convolution at first.

### 3. System and Hardware Architecture

Function Type	Computation Share
3x3 Convolution Input Layer	4.1 %
1x1 Convolution Squeeze Layer	11.0 %
1x1 Convolution Expand Layer	6.8 %
3x3 Convolution Expand Layer	60.8 %
3x3 Convolution ConvDet	17.4 %

Table 3.2.: Computation Time Share Breakdown per Layer Type of SqueezeDet

#### 3.3.3. Data Transfer to Programmable Logic

One general design task was the data transfer from the processing system (PS) to the programmable logic (PL). It was implemented with the use of the Xilinx SDSoC tool that generates the full I/O logic on the PL and the respective I/O drivers for the PS. In SDSoC there are different possibilities to construct such an I/O channel on a Zynq. The Zynq incorporates different AXI interfaces from and to the PS (Master and Slave) that can be used for different applications. There is the possibility to access the DDR memory directly or via the L2 cache shared by the multiple ARM cores on the Zynq PS.

In our case we chose to use a streaming I/O solution with a high-performance AXI slave port, that would allow to stream data from the PS to the PL with highest throughput. With this solution a DMA is generated on the PL that acts as AXI master and accesses the memory in a sequential manner that allows to retrieve or write the data with no access latency.

Concerning the bandwidth of the I/O interface, SDSoC enables to enlarge the bus-width to a multiple of the word width in order to transfer several data instances simultaneously. Nevertheless this option is limited to 64-bit bus width in the current version of SDSoC (2016.3). In our case this means that a maximum of 8 data items can be transferred concurrently. The theoretical limit that is implied by the bit-width of the high-performance AXI ports is 128 bits. That is, a factor 2 of the the connections bandwidth is not utilised in this setting.

Further the bandwidth is proportional to the clocking of the data-motion network. This is a parameter that can be set according to the design.

#### 3.3.4. Single 3x3 Convolution Accelerator

In a first step of accelerating the *SqueezeDet* algorithm, a single hardware accelerator for 3x3 convolutions was designed. It already incorporated the quantized computation with the respective bit-shifters for the different precision ranges of the different layers.

### 3. System and Hardware Architecture

#### Parallel Computation

The main concept of a hardware accelerator for convolutions is to parallelise the computations as much as possible. Since the convolution over a feature map has that many dimensions basically four parallelisation dimensions are available:

**Output Channels** This is the most used dimension for parallelisation, since it comprises identical computations with different weights for the different output channels. In this case the feature map activation value is shared among the compute units and different weights have to be loaded.

**Feature Map Height** Parallelising over the feature map height incorporates that the convolution computation is performed concurrently for different output pixels. (on different feature map rows) In this case the computation weights are shared and the feature map activation value is different for each compute unit. The main drawback here is that for the case of having a 2-dimensional kernel, same activation values have to be loaded by different compute units at different times. Therefore they share the memory section, without having the same access pattern.

**Feature Map Width** In this case one basically finds the same setup as when parallelising over the feature map height.

**Input Channels** When parallelising over the input channels the great difference is that neither the weights nor the activation values can be shared. This means that more memory bandwidth is needed. A further drawback is that the different sums of the different units have to be summed up over different compute units in order to compute the final value. This is not resource efficient since a further unit is needed to do this specific calculation even though it is a simple accumulation that the compute units would be able to perform on their own.

Now on the way of achieving the highest possible parallelisation on the underlying system, quickly some design limitations came up. The first restriction came up when trying to parallelise the output channels as far as possible. Here the design of the algorithm came into our way with the limitation of 64 output channels in the first fire layer. Further parallelisation would mean that often not the whole processing system could be utilised which is a waste of resources.

The limiting factor when parallelising on the feature map height was that a larger feature map buffer was needed. Since the data storage pattern has the width dimension first and the access pattern of the hardware accelerator is sequential the full image width has to be loaded and stored in a feature map buffer up to the desired parallelising height. This implies that a large memory is used to buffer the activation values that also has to have the parallel structure in order to access the different rows used for computation in parallel. This results in a limiting design factor.

### 3. System and Hardware Architecture

Parallelisation on the feature map width showed not to work with SDSoC since it was not able to generate a memory access network in order to access the different columns concurrently. Even with separately instantiated block RAMs the generated logic would not achieve a fully parallel access. This issue is later further discussed at the end of the implementation chapter.

While parallelising the input channels the limitation is again on the design of the algorithm. The minimum of 16 input channels imply that when parallelising the input channels, not many accumulation cycles are available anymore. This leads, that the performance gain is not as large as desired since the only few accumulation cycles lie between the pipeline stall for finalising the output values, including the addition of the various intermediate results (of the different input channel compute units), the addition of the bias values, the quantisation, the shift and the ReLU evaluation. In other words the pipeline is not filled to a satisfying extent. Therefore an extensive parallelisation in this dimension is not desirable.

Considering above mentioned possibilities and circumstances the final design of the single 3x3 convolution accelerator was chosen to compute the convolutions in parallel with an order of parallelism of 64 in the output channel dimension, an order of parallelism of 8 in the feature map height dimension and an order of parallelism of 2 in the input channel dimension. This leads to a total parallelism of a factor of 1024.

This seems to be fairly low compared to the available DSP resources of 2520 DSP slices, but the aforementioned circumstances restricted an efficient parallelism and the fact that the circuit for the finalisation of the calculated value, that is the addition of the bias, the quantisation, the bit-shift and the ReLU evaluation, need a considerable amount of logic, made the DSP resources uncritical. In the end it was really the logic resources for the finalisation stage and the memory resources for the parallelism on the feature map height that would stand out as the parallelisation bottleneck.

This finding lead to the fact that the described fixed-point optimisation for 8-bit computation on fewer DSP slices was of no use since the bottleneck was not the amount of DSP slices available. I was able to see that the HLS compiler would undertake this optimisation for a very small accelerator but it would not consider the optimisation anymore when the circuit would enlarge to a critical size.

The generated hardware finally has following elements depicted in figure 3.5. On the one side the filter memory where the filter coefficients are stored at the beginning of the convolution computation and on the other side an image buffer, that stores the needed amount of rows of the feature map in order to compute the different rows in parallel. Additionally there is an output buffer that saves the computed values in order to write out the data in the needed sequential pattern. Once the the filter and the input is loaded to memory, the computation is performed in the parallel compute units that at the end output the respective output values to be stored in the output buffer.

### 3. System and Hardware Architecture

In the algorithm description 1 the general processing sequence is listed which is run on every 3x3 convolution layer.

---

**Algorithm 1:** Processing Sequence of 3x3 Convolution with Hardware Accelerator

---

**input** : Weights, Feature Map, Control Parameters

**output:** New Feature Map

```
1 load_weights_to_device();
2 for i ← 0 to height / parallel_rows do
3   load_input_buffer();
4   for j ← 0 to feature_map_width do
5     for j ← 0 to output_channels/parallel_output_channels do
6       run_parallel_3x3_convolution();
7     end
8   end
9   write_out_output_buffer();
10 end
```

---

#### Parallel Memory Access

In order to compute many convolutions in parallel, multiple data has also to be available at the same time. On the one hand this implies that a buffer is needed since the access from DRAM is sequential and also bandwidth limited. On the other hand the on-chip buffers have to present a parallel structure such that multiple data can be retrieved simultaneously from it. In this application on FPGA this is solved with a large amount of block RAM instances that can be joined together to form a larger buffer. Since every instance can perform a memory read or write at the same time, this allows to access multiple data from the buffer simultaneously.

In our case several different dimensions of data parallelism is needed in the buffers as described below. In a first instance the filter buffer and the input buffer have to feature parallelism in the input channel dimension since this is the first dimension on how the data is stored on the host memory (external DRAM) and therefore multiple words are read in parallel over the IO bus as described in section 3.3.3.

The filter buffer has to additionally feature data parallelism on the output channel dimension as multiple compute units perform processing on different output channels. Further the input buffer has to be able to deliver data from different rows simultaneously such that parallelism in this dimension is also necessary.

Finally also the output buffer needs parallelism in the dimension of the output channel and the height (rows) in order to manage the needed data processing.

### 3. System and Hardware Architecture

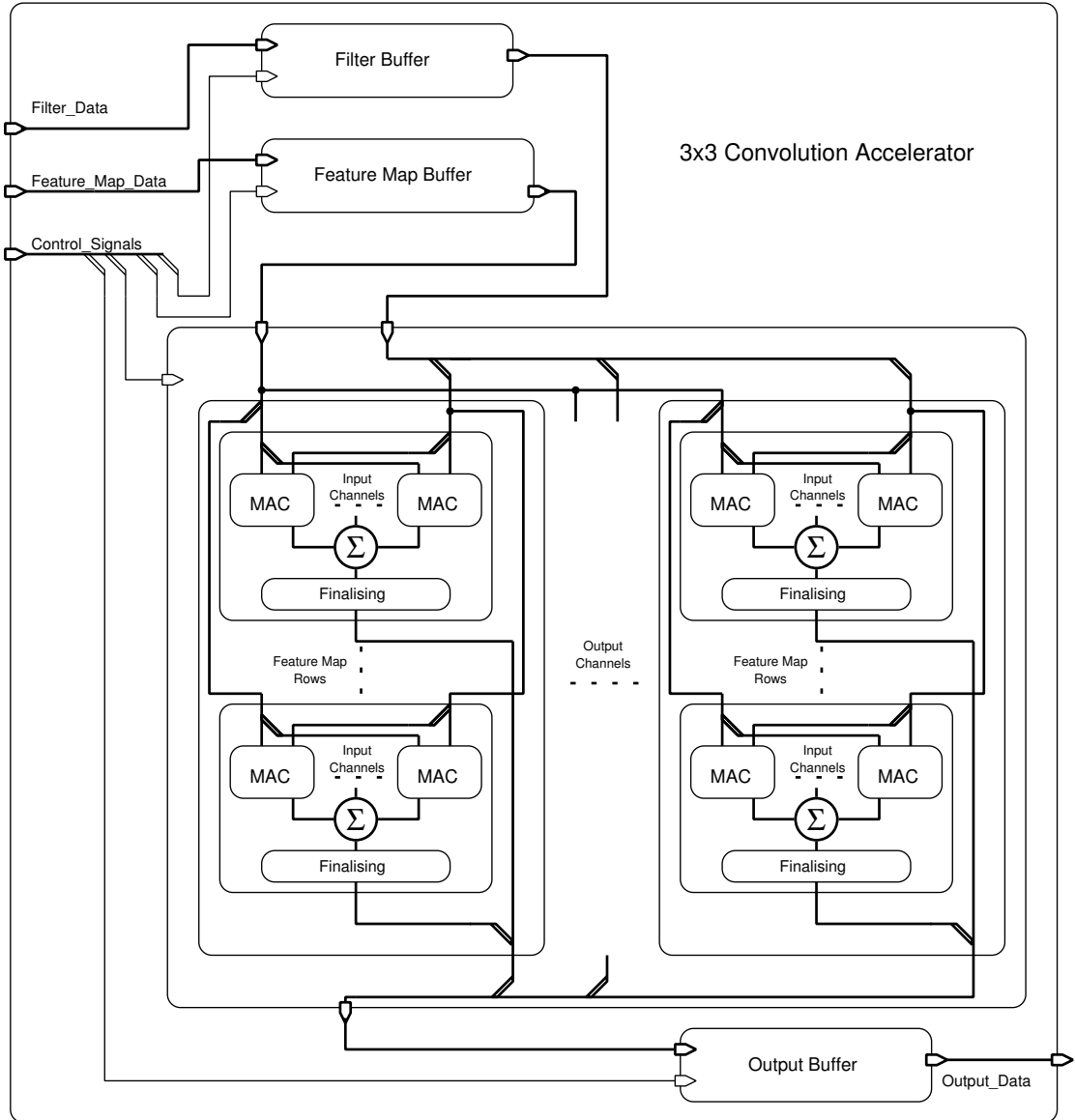


Figure 3.5.: Block Diagram of 3x3 Convolution Hardware Accelerator

### 3. System and Hardware Architecture

Figure 3.6 illustrates the parallelism of the input buffer and shows how the processing sequence on a feature map channel executes with an example of 6 compute units running on parallel rows.

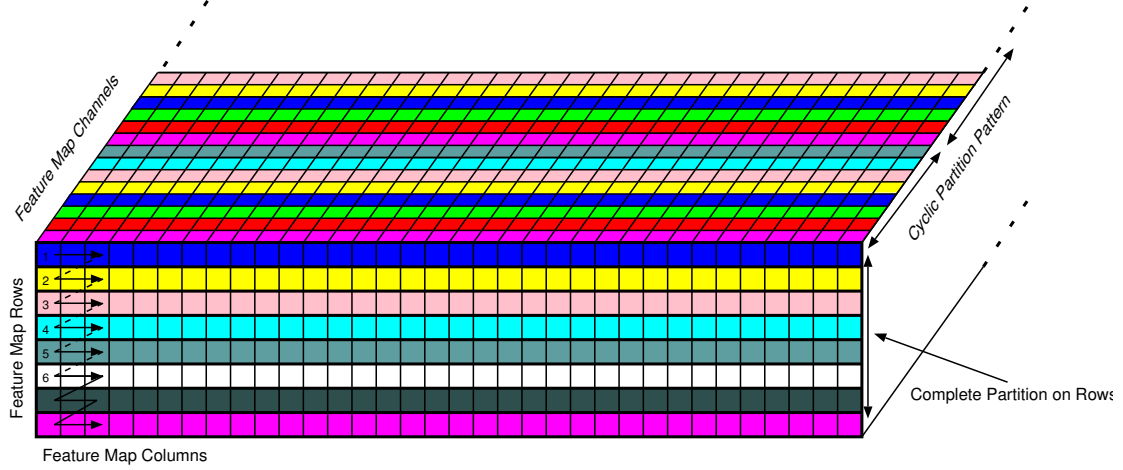


Figure 3.6.: Input Buffer Memory Layout and Processing Sequence

#### Processing Pipeline and Implications

The generated design of the 3x3 convolution accelerator features as described multiple outer loops with a pipelined most inner loop executing the actual convolution. This is the function *run\_parallel\_3x3\_convolution()* in the algorithm sequence 1 that runs a loop going through the 3x3 kernel pixels and through all respective input channels. Looking at the whole computation in the most inner loop, it shows a considerable delay, which leads to a pipeline that features 15 stages. The whole pipeline with all its stages is depicted in figure 3.7. However since the pipeline is set inside a loop, as soon as the most inner loop is finished, the pipeline is flushed instead of already computing intermediate results for the next iteration of the outer loop. This is the way Vivado HLS handles loop computation and performs quite well as long as you have significantly large loop boundaries for the most inner loop. It can be seen as a branch operation flushing the pipeline.

This observation points out the unsuitability of the parallelisation of the input channels. Increasing the input channel parallelism on the one hand increases the pipeline due to a longer pipeline to gather all the partial results and at the same time shortens the amount of runs of the most inner loop. That means that it decreases the efficiency of the underlying hardware which manifests itself in a too small speedup compared to the increase of the hardware resources.



### 3. System and Hardware Architecture

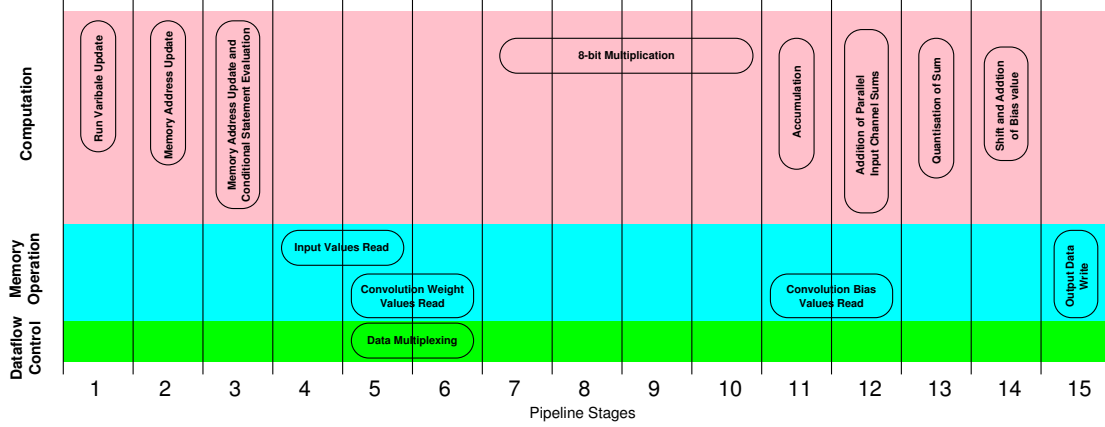


Figure 3.7.: Processing Pipeline of the 3x3 Convolution Accelerators Most Inner Loop

#### 3.3.5. Full Expand Layer Accelerator with Single Convolution Accelerator

In a second step of further optimising the algorithm acceleration, the goal is to include the parallel 1x1 convolution of the expand layer. Since it is also a convolution and the input data is identical it is obvious to perform the calculation directly on PL as well. What changed relative to the pure 3x3 convolution accelerator is that additional weights have to be loaded and stored in the beginning and the output buffer size increases by a factor 2 since the 1x1 convolution gives out the same amount of output data that has to be concatenated with the 3x3 convolution output data.

The more significant observation however is that when using the same convolution accelerator core, the modification for it being able to also perform 1x1 convolutions implies that the loop boundaries for the kernel loop is not 9 (3x3) anymore but it has to be able to tackle a loop end every cycle. This introduces a major drawback in the pipelining of the computation since the branch operation accrues much more often. This especially slows down the 3x3 convolution such that the gain in accelerating the 1x1 convolution is not as big as desired.

#### 3.3.6. Full Expand Layer Accelerator with separate Convolution Accelerators

In order to overcome the drawback described above, one idea was to pack the 1x1 convolution into a separate accelerator. This would enable the 3x3 convolution accelerator to optimise for a round trip of 9 (3x3) and let the other accelerator comprise an optimal 1x1 convolution accelerator.

### 3. System and Hardware Architecture

However this additional accelerator again uses a certain amount of resources such that the total resources are quickly filled. That is that the second accelerator cannot be designed to run as many computations in parallel because of missing resources. Even though the computation effort of the 1x1 convolution is a factor 9 smaller, the accelerator can't run as efficient since the pipeline is flushed more often due to the smaller amount of elements that are added up in the same sum. Then a factor 8 less parallelisation gets the 1x1 convolution to a similar computation time as the 3x3 convolution which in total results in a slower performance than the scenario described above with a shared convolution accelerator. In chapter 5 numbers are listed that illustrate this behaviour.

#### 3.3.7. Separate 1x1 and 3x3 Convolution Accelerators

In an other approach two completely separate accelerators are designed that act independently. Resource-wise it brings similar conditions as when having two separate accelerators inside an expand layer accelerator. Since the 1x1 convolution is meant not to be designed largely due to the resource limitations, it can be designed with no parallelism in the feature map height dimension. This implies that neither an input nor an output buffer is needed. Therefore a similar resource use results as in the section before.

The main gain of designing two separate accelerators is that the 1x1 convolution can also be used for the *Squeeze* layers. Additionally to that fact, the 3x3 convolution can be used for the layer 1 convolution as well, with only a few modifications, since that convolution has a stride of 2 with no padding, that has to be taken into account.

One implication that such a solution is implying is that the concatenation of both convolution outputs in the expand layer has to be accounted for. In our solution we designed the 1x1 convolution accelerator to take two separate input streams, in order to concatenate both feature maps if required.

#### Separate 1x1 Convolution Accelerator for Squeeze and Expand Layer Computation

As previously mentioned the design of the 1x1 convolution differs itself from the 3x3 convolution in the way that it has to be flexible due to its application on both squeeze and expand layers. The 1x1 convolution doesn't act on different input pixels for the computation of an output pixel and therefore makes a multiline input buffer superfluous. However it has to implement a channel concatenation in order to be efficiently applicable in the context of the *SqueezeDet* computations.

The flexibility of the accelerator implies that it can't parallelise extensively in the most desired output channel dimension. The smallest squeeze layer has only 16 output channels. Compared to the 3x3 expand convolution, where the minimal output channel size is

### 3. System and Hardware Architecture

64. Nevertheless the squeeze layer enables to make better use of input channel parallelisation due to its large input channel numbers. This however is again not well applicable to the 1x1 expand layer.

Looking at the memory implication we can state that the absent multiline input and output buffer increase the compute efficiency since no additional read/write function blocks have to be initiated. However the missing of multiple lines available in the buffer makes it impossible to perform a direct parallelisation of the computation in the feature map height in an analogue way to the 3x3 convolution accelerator. To counter this effect and still compute multiple pixels at a time, an approach of calculating two streams at a time was implemented. In this way, the hardware accelerator has double the amount of input and output streams and processes different pixels in parallel. This is doable since all the computations of the different pixels are completely independent. Compared to the 3x3 convolution accelerator the loaded weights can be applied to all the parallel streams.

Despite the non-use of the multiline buffer still a buffer is necessary. Since the input is read in a stream (identical to 3x3 convlution), all input values of a pixel have to be buffered for layers with a larger amount of output channels, that surpass the number of simultaneously computed channels. This is the case since the input values have to be reused for the computation with a different set of weights.

On the output side no buffer is needed since the computed values can directly be fed to the output stream. The values still have to be stored in a short term memory because many results are computed simultaneously, however the FIFO buffer that belongs to the streaming interface takes perfectly care of that. Thereby no additional buffer is needed. A block diagram of the architecture is depicted in figure 3.8

### 3. System and Hardware Architecture

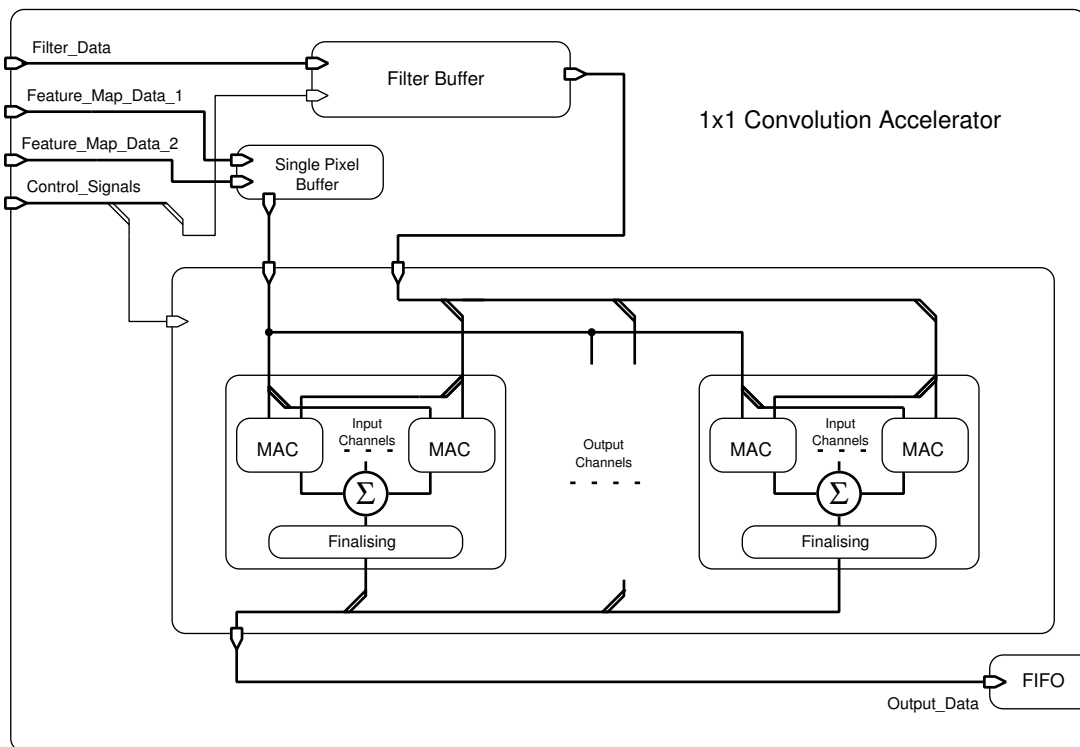


Figure 3.8.: Block Diagram of 1x1 Convolution Hardware Accelerator

# Chapter 4

## Design Implementation - Embedded Object Detection Platform

In this chapter the setting and tools used to create the embedded processing platform is described. In a second part the algorithm implementation is discussed.

### 4.1. Xilinx ZCU 102 Zynq Board

The project was performed on a Xilinx ZCU 102 evaluation board. The boards core component is a Xilinx Zynq Ultrascale+ MPSoC chip (XCZU9EG). It is an SoC chip with four ARM Cortex-A53 cores as main processing system and an FPGA equivalent to a Kintex Ultrascale+ device as programmable logic. The chip is produced in 16nm TSMC FinFET technology that is common for Xilinx devices with Ultrascale+ architecture. Some details over the resources of the programmable logic are listed in table 3.1 in chapter *System and Hardware Architecture* where details of the design possibilities with the chip are discussed. The core SoC on the board is equipped with a set of 4GB DDR4 memory for the processing system and a dedicated 512MB DDR4 memory for the programmable logic.

The board is equipped with numerous peripherals such as Gigabit Ethernet, USB 3.0, HDMI, Display Port or SATA. It also features four SFP+ cages, a PCIe interface and various other IO interfaces. The system on board can be programmed and accessed through an SD-card or over USB-JTAG and USB-UART.

In our setting we wanted to look at it comparable to an embedded and remote surveillance system and therefore only used a subset of all available resources. These were the SD-card to load the bitstream for the programmable logic and the operating system, the

#### 4. Design Implementation - Embedded Object Detection Platform

USB port to attach the camera and the Ethernet port to connect the platform remotely to a central system.

### 4.2. Vivado HLS & SDSoC

For the development of the system the Xilinx tool suite for high level system design including Vivado HLS and SDSoC was used. They form an automated design structure where top level C/C++ applications are written and part of the functions are sourced out to a dedicated hardware accelerator. The Vivado HLS tool compiles a hardware structure out of the underlying C code and SDSoC generates the needed driver and launch routines in order to directly launch the hardware accelerated functions in the standard function code.

Since hardware has a totally different process structure compared to a sequential program, the HLS compiler uses predefined code pragmas to enforce a specific hardware structure. These include instructions for parallelising a loop, perform pipelining on a loop or fragmentation of arrays in order to access multiple elements concurrently for parallel computing.

On the other hand SDSoC also includes a set of pragma instructions especially for the data exchange network between the processing system and the programmable logic. Various systems exist where different AXI ports act either as master or as slave and the port is either attached directly in the processing systems L2 cache or directly to the DDR RAM.

If not mentioned otherwise the Xilinx tools used are of the release 2016.3.

### 4.3. Petalinux

The operating system used is a Linux version generated with Xilinx's tool *Petalinux*. The standard OS image delivered with the board was also a Xilinx Linux image where different key parts were missing for a smooth usage. These included a specific MAC address, a standard user with password, an auto-login on the UART shell and various missing libraries. With the help of the *Petalinux* tool a custom Linux image was created and added to the system files of SDSoC that are needed for the generation of the SD-card.

### 4.4. Point Grey Flea 3 Camera

As a camera for the embedded object detection platform we used a Point Grey Flea 3. It is an advanced machine vision camera of the type FL3-U3-32S2C with 3.2 MP

#### 4. Design Implementation - Embedded Object Detection Platform

that achieves 60 frame per seconds with colors. It works with USB and performs some preprocessing on the camera before buffering images on a camera buffer of 32 MB size. The camera is very configurable and many different modes can be set varying color mode, frame rate, image size, image position or data transfer mode for example. The access to the camera and the configuration happens over a C/C++ API that is delivered and can be included in the SDSoc project. What is needed in addition are the respective includes and libraries.

For the platform setting we chose a image size of 1232·374 and BGR color mode with the image centred. There is no fixed frame rate since the images are polled by the application at its own computing rate. Older images in the buffer that are not fetched are dropped after FIFO.

### 4.5. Output Transfer over Network

Finishing off the object detection platform, only a setup for evaluating the detected objects had to be found. From the beginning on, the target was to compose a system in the smart surveillance camera domain. In this setting the camera is installed remotely and hands over the images and informations to a central office. The application of displaying the image from the board is rather unlikely and was therefore not considered. The chosen setup was instead to depict the found bounding boxes in the image and then transfer the image over an IP network to a host server that would be able to further process the information or the image respectively. The data transfer was implemented over TCP with a custom protocol that would only send the raw data with synchronisation tags in between and also at the end of the image transfer. The setup is fixed for a certain image dimension.

Basically in a fixed camera setting also the transfer of the bounding box information only would do its job. The bounding boxes could then clearly be assigned to an image part since the image is static. The information would include position and size of the bounding boxes and the class of the detected object. This would drastically reduce the needed data transfer size, which for an embedded application would be very desirable in order to minimise power consumption and network resources needed. However in our application it is desirable to have an image in order to check the quality of the algorithm and design as well as for demonstration purposes.

### 4.6. Algorithm Implementations

In this section the different algorithm architectures that were used are discussed and the optimisations on the basic implementation are explained with some rough numbers of improvement. It can be seen as chronological progress of the implementation. More detailed numbers and discussions are situation in the results chapter.

#### 4. Design Implementation - Embedded Object Detection Platform

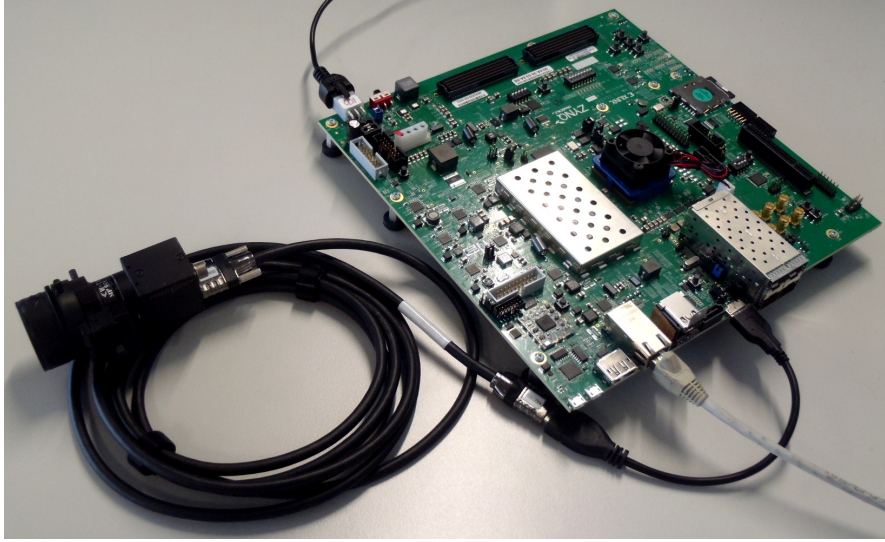


Figure 4.1.: Platform Setting with Board, Camera and Network Connection

##### 4.6.1. Data Arrangement

To start off a general design choice had to be made to build the system. Since the multidimensional feature maps and convolution weight tensors can be arranged in different ways, a specific had to be chosen. The choice was made with the arrangement of feature map featuring the channels in the first dimension, the width in the second dimension and the height in the third dimension. The arrangement is shortened with (HWC). The convolution weights are arranged in an analogue way, having the output channels in the fourth dimension. This leads to the short version of (FWHC). This arrangement is common since it keeps all channels of a pixels together, that are used for computation in the same run during the convolution. Changes in memory arrangement enables a vast set of different design considerations that are not followed in this project.

##### 4.6.2. Basic C Implementation

In a first instance a basic C implementation running the *SqueezeDet* on the ZCU 102 was created to get the whole system running, including importing the weights, importing the input image, performing all the essential parts of the algorithm, essentially the convolutions, the pooling, the softmax and sigmoid as well as the whole postprocessing that retrieves the essential bounding boxes that matter. As a reference input, an example image from the *Github* repository was used. In this first scenario the bounding boxes were inserted into the picture and then saved. The weights and the images were all set as pure bit files that only include the raw data.



#### 4. Design Implementation - Embedded Object Detection Platform

After the first implementation with floating-point numbers, equivalent to the reference design in *Tensorflow*, the whole setting was set up with quantised data and quantised computation. As described in section 3.2.3 the quantisation scheme chosen was with 8 bit fixed point numbers that have variable fractional precisions, depending on the processing layer and depending on the data being filter weights or layer activations. For the temporary values during the computation following scheme was used: 16 bit precision for the product of the multiplication and 32 bits for the accumulation.

The change to quantised computation also included the insertion of logic shifters in the computation sequence that were necessary in order to match the necessary precisions of the different layers. Also the bias values had to be shifted, since their fractional precision would not match the accumulators fractional precision.

Eventually the change to quantised computation already brought a significant speed up, leveraging the faster computation of fixed-point numbers and the increased locality of the data thanks to its reduced size.

Unfortunately not the whole calculation could be conducted in 8-bit quantised numbers as the results of the quantisation process in chapter 5 show. They depict that the quantisation of the last layers output activation lets the algorithm quality drop significantly. The conclusion was that the last (ConvDet) layer was calculated in a separate function that saved the output activations as float to the memory for further processing.

##### 4.6.3. OpenMP Implementation

In a second step the computation of the convolution was improved by distributing the computation load over all the four ARM cores of the PS with the help of OpenMP. The parallelisation however comes with a certain overhead and data sharing can imply some drawbacks when e.g. data locality in the cache is destroyed by an other thread. The implementation still showed that a good speed improvement with a factor of nearly four could be achieved on the computation-heavy convolutions and also on the softmax and sigmoid computation. The pooling layers that are more memory dominated however would not profit at all from the multi processing.

##### 4.6.4. Platform Implementation

Eventually the *SqueezeDet* algorithm should be running on an object detection platform, which is why the integration of a camera feeding real input images into the system and an output unit to output the result was due next. The integration of the camera involved loading various libraries, including the vendors API library, usb libraries and other Linux device utility libraries. The camera was then connected with the system over an API available in C and C++. Weird enough the camera setup and calibration would not work in C and is therefore performed with a separate C++ program.

#### 4. Design Implementation - Embedded Object Detection Platform

In order to write out the result of the CNN, a network application was instantiated that works on the standard Linux socket library implementing a TCP connection to a remote server and repetitively sends the final image with the respective bounding boxes inserted, to the server. The server then further processes the information, in our case, it displays the image.

##### 4.6.5. Hardware Accelerator for 3x3 Convolution

From the beginning on the goal of the project was to accelerate the algorithm with the help of a custom hardware accelerator on the PL of the Zynq SoC. After evaluating the computation load of the algorithms different parts, the obvious way to proceed was to accelerate the 3x3 convolutions. Around 80% of the computation of the *SqueezeDet* algorithm happens in those 3x3 convolutions. Again most of them are situated in the fire layers, that incorporate 3x3 convolutions with a stride of 1 and zero padding at the edge in order to keep the feature map measures.

Therefore a first implementation of the hardware accelerator incorporated a 3x3 convolution accelerator featuring a stride of one, border zero padding and a ReLU at the activation output. The structure implements multiple compute units, that perform the pixel wise multiply and accumulate, which are arranged in order to compute multiple output channels and multiple rows in parallel. The computation is supported by an input and an output buffer which enable efficient computation. Furthermore the whole weights are buffered in a separate buffer. The detailed architecture is described in chapter 3.

##### 4.6.6. Hardware Accelerator for Total Expand Layer

The next step of also running the parallel 1x1 convolution in the expand layer was not only desirable but necessary. This was especially the case since the used streaming IO interface, that stands out by its very efficient data transfer, would not be able to jump memory sections when outputting data in order to implement the concatenation of the two expand layers. The solution was to integrate the 1x1 convolution of the expand layer into the accelerator in order to have a complete expand layer accelerator. It would benefit from the fact that they share the input data for the computation. However the accelerator had to be reshaped since the memory requirements rose significantly because the output buffer had to be doubled and a second weights buffer had to be integrated.

For the computation two compute versions were considered and compared. One version would incorporate a single convolution accelerator core that is able to compute convolutions with variable kernel size. This however implied that the pipeline couldn't be optimised in order to have an initiation interval of 1 over the full pixel calculation. The other version would incorporate two separate cores that are each specialized for 1x1 convolution or 3x3 convolution respectively. This scenario has the drawback, that it uses considerably more resources such that the 1x1 convolution accelerator can't be designed

very large. The resulting designs show quite a similar performance and show a small trade-off. The results are treated in the following chapter.

##### 4.6.7. Two Separate Hardware Accelerators for 1x1 and 3x3 Convolution

It was clear that for a complete acceleration of the network, the design would also have to include an accelerator for the squeeze layers as well. In the structure that was given by the aforementioned expand layer accelerator the setting was laid out to firstly make use of the compute dimensions of the expand layer (few input channels and many output channels) and secondly to perform a concatenation of both convolutions at the output. This however is not applicable for the squeeze layers since they don't concatenate with any other convolution layer and since they have the opposite compute structure with many input channels and only a few output channels.

The solution was therefore to separate both accelerators and omit the concatenation in the accelerated computation of the expand layer. The independent 1x1 convolution accelerator is therefore designed to fit to both applications, which means that it has a moderate parallelism over the output channels such that it doesn't surpass the minimum output channel number of the first squeeze layer (16) and also over the input channels since the expand layers have only few input channels to compute in parallel. Anyways it showed to be undesirable to parallelise over the input channels since it forms the most inner loop that is pipelined and shortening its compute load makes the system less efficient and takes less performance out of the additional resource allocated.

Furthermore the new 1x1 convolution has to occasionally perform a concatenation, that is, it has to run the convolution on two different input streams. With this implementation a separate concatenation layer in software is omitted in order to save time.

The resource needs of the 1x1 convolution accelerator was especially for the memory requirements more modest as it would not need any input or output buffer. This is the case since the computation restrains itself on one single pixel, which makes a large buffer superfluous. More details are given in the Hardware architecture section.

This design eventually accelerated most of the feature extraction layers, which make up around 78% by accelerating all fire units by a significant amount.

##### 4.6.8. General Algorithm Optimisations

The following subsection tackles further optimisations that were conducted with the previously described accelerator setting. Those optimisations were performed on a system level as well as on the compute level (in the accelerator).

### 3x3 Accelerator on Layer 1

The biggest part missing of the feature extraction network running with 8bit quantisation was the first layers 3x3 convolution. The characteristics preventing to simply apply the designed accelerator for 3x3 convolution were the large feature map size, the stride of 2 and the inexistent padding. The large size of the feature map made it impossible to apply the accelerator to the layer as well as to enlarge the accelerators buffer to fit the necessary data in it. The solution to the different stride and the missing padding was to introduce a function picking the right values out of all the calculated values. Evidently in this way too many computations are performed, but with the gain of an accelerator, this can be made good by far.

The problem of the large feature map was resolved by splitting the image into 16 image parts that are computed separately. This brings up another overhead of splitting the image and calling the function multiple times. still the computation would run much faster. The striding and missing padding was only applied in the next layer (pooling) in order to save an additional memory reordering.

### Channel Padding for Larger Output Channel Parallelism

Looking at the performance of the system at that stage, the 1x1 convolutions took most of the time for processing. Since input channel parallelism showed not to be very efficient, the output channel parallelism was the target. However the limiting factor was the minimal output channel number of the second squeeze layer with a number of 16 output channels. In order to enlarge the parallelism to 32 the applied solution was to pad the layers 2,3,6 and 7 so that all channel sizes were a multiple of 32. This would on the one side include a padding of the output channel weights of the squeeze layers as well as padding of the input channel weights of the expand layers. Understandably the performance of the specific fire layers would decrease by a small amount but the more compute intensive layers would profit by a factor of close to 2.

### Bias Adding after Quantisation

With the extensive parallelisation of both accelerators the system quickly ran into resource bottlenecks. At that stage the bottleneck would be the LUT consumption. Some analysis of the LUT usage in different logic parts the shifters sticked out. A detailed analysis of the occurrences showed that many LUTs could be saved when performing the addition of the bias value after the quantisation of the accumulated intermediate result. In this way the shifter would only have to be 8 bits wide and not 32 bits which considerably reduced the LUT consumption. However it must be stated that the actual computation is not completely equivalent, since in this way two quantised numbers are added which brings along a very small error. This error would show in detailed numbers

#### 4. Design Implementation - Embedded Object Detection Platform

of the processing but not at the system level. A fine-tuning of the weights considering this computation would take this into account even though it doesn't seem necessary.

##### **Assert Range of Logic Shifter Input**

Next to the size of the shifter that could be modified, the system only resulted with a very limited range of shift numbers. While the HLS compiler had to assume that the input could be in the full range of an integer value, an assertion to the value range helped to adapt the hardware correspondingly. It was performed with the Vivado HLS assert statements. This optimisation would not help as much as the one before, reducing the shifters bit width.

##### **Resource Binding to DSP for Bias Adding**

Next, the resource occupation of the LUTs showed to add up for the generation of multiple adders. Those were among others the adders to add the bias values to the accumulated value. Since clearly the LUTs were the limiting resources and still a large amount of DSP slices were still available, the LUTs could be freed with the help of resource binding. With HLS assignments the compiler can be instructed to use DSP slices for the calculation instead of LUTs that he uses by default for adders.

##### **Double 1x1 Instances on separate Streams**

Since after the described optimisations some resources were freed they gave room to some further optimisations. At this point the 3x3 convolution would still use less compute time than the 1x1 convolutions and the absolute resource needs were much smaller, which in total suggested a further parallelisation of the 1x1 convolutions. As described before the 1x1 convolutions compute every single pixel separately such that basically the whole image can be parallelised. With this in mind the idea was set up to have multiple instances that work concurrently on a separate region of the feature map. This could easily be achieved with the SDSoc pragma "resource" that tells the compiler to use different instances of the underlying hardware accelerator. This brought a speedup of quite exactly two as the workload is effectively halved.

##### **Double Stream 1x1 Convolution Computation**

The drawback of the solution with two instances of the accelerator is that it consumes a severe amount of memory resources, because it loads the convolution weights twice. This is undesirable since the memory is also quite scarce resource due to the input and output buffers of the 3x3 convolution and affects the design of it when this method is applied. However when changing the design of the 1x1 convolution to concurrently process two

#### 4. Design Implementation - Embedded Object Detection Platform

streams of data with the same weights this drawback is eliminated. The only additional resources used for the design are more IO interfaces and the computation related DSPs and LUT's. Compared to a further parallelisation in the input channel dimension the speed up is actually near the factor two (in our case) and consumes a comparable amount of hardware resources.

##### **Contiguous Memory Allocation**

So far the feature map memory in the system was allocated by a normal *malloc()* system call that allocates common virtual memory by the Linux operating system. This way the memory is scattered in the memory distributed over multiple pages. This makes on the PL part a special DMA necessary which performs scatter-gather memory operations. On the one side this DMA is less efficient and on the other side it takes up more hardware resources than a normal DMA. The way to surpass this inconvenience is to use a system call by SDSoC called *sds\_alloc()* that allocates contiguous memory to the specified pointer. Like this the memory access is improved with the only drawback of the system having less flexibility to allocate new memory.

This improvement had already been looked at an earlier stage, however the improvement was only very small if not negligible compared to the runtime at this stage. The improvement manifests itself primarily when the data movement is a large part of the total processing. This implies that mainly the first layers and especially the 1x1 convolutions profit from it. In the setting with the 1x1 convolution integrated in an expand layer accelerator this would not help at all since the 1x1 convolution would read/write the data directly from/to the input/output buffer. Though in the newest setting a considerable speed gain was measurable.

##### **Concurrent Expand Layer computation**

Since at this stage the 1x1 and the 3x3 convolution accelerators were designed in separate instances, it was tempting to use the fact to accelerate the expand layer where both functions run in parallel. Like that both instances could execute at the same time. This is practicable because the output streams do not compete on the same memory sector and the read operations on the shared input are not jeopardising the data. However this fact brings in a certain slowdown for the memory reads but the overall time saving makes still a good impact.

##### **System 'Optimisations' with Minimal or Negative Impact**

In this subsection some design modifications are explained that were considered and rejected due to their little or inexistent performance gain.

#### 4. Design Implementation - Embedded Object Detection Platform

**Concurrent Input Read and Output Write of Accelerator** On the 3x3 convolution accelerator algorithm sequence level another parallelism is applicable. Namely the input read and the output write into and from both buffers happen totally independently, such that they could be run concurrently. When looking at the actual numbers the gain is quite moderate. This is firstly due to the imbalance of the input and output channels for the 3x3 convolution layers in *SqueezeDet*, which make a concurrent operation only save the shorter execution time, which in this case it is quite smaller than the bigger part. For example in the second layer per accelerator run only 4944 cycles are spent to load the input data whereas the output data takes 19'776 cycles to write to the DDR memory. The ratio doesn't vary a lot across the different layers. And additionally to small 20% savings of data transfer time, in most of the layers, that is layer 6-11, only 3 runs are performed by the accelerator, which makes that only two out of three times this optimisation is applied, since the first input read and the last output write can't be executed in parallel.

The implementation in HLS also didn't convince since the resource needs rose significantly. This impacted the possibility for earlier discussed parallelisation such that the idea was again discarded.

**General Concurrent Memory IO during Computation** In dependence on the method discussed before naturally the goal of hiding the latency of the memory transfers with the computation is very tempting as well as desirable. Compared to the computation time the memory IO time varies quite a bit. While in layer 2 the computation time makes up a bit more than 50% of the total processing time, in layer 11 the computation time is at 85%.

The problem arising with this setting is that with the sequential memory IO pattern it is impossible to read from different feature map rows simultaneously or right after another. Therefore in order to read or write from a certain row, all values of the previous row have to be already read or written. This makes it impossible to e.g. write out the simultaneously computed output values of different rows, since all the values of the respective first row have to be written first.

A desirable change to the architecture in order to change the compute parallelism from the height to the width was unfortunately not implementable. We tried it, but the HLS compiler would not be able to handle the complexity as it looked. The design possibility is further discussed in the hardware architecture section.

**Using Memory Allocated in a Non-Cacheable Way** Memory can also be allocated with the system call `sds_alloc_non_cacheable()` that allocates memory that will not be cached. This makes the access latency smaller for the hardware accelerator since the path, the memory has to run, is reduced since it doesn't have to go through the processors cache first. However in the moment that the data is used on the PS, it is not cached either.

#### 4. Design Implementation - Embedded Object Detection Platform

This introduces a very high latency since the access is just like having only memory misses. A trial implementation in our use case showed that the application of non-cacheable memory sped up the accelerated function in the single percentage area. While introducing a very high slowdown on functions executed in the PS. (Such as Pooling, Softmax, Sigmoid) The acceleration was probably only negligible because the streaming memory access used before, already hides the memory access latency and therefore it doesn't change considerably when accessing the data directly.

**Enlarging IO Bandwidth with Xilinx Suite 2017.1** So far with the Xilinx Suite of Release 2016.3 a maximum of 8 bytes (64 bits) could be used for packed structs, which are constructs to enlarge the bandwidth of an IO interface. With the new release the setting are updated such that the packed structs can have a size of up to 1024 bits. However the AXI high performance interfaces have a bit-width of 128 making a packed struct of more than 128 bits not improving the performance significantly.

When trying to implement the new feature, it showed to not easily realise in the expected way. The main problem is that the parallel transfer of data requires that the memory shows the same parallelism in order to have no latency in the operation. Unfortunately this increases the resource needs substantially even though not more actual memory is used. The assumption is, that the compiler is not able to put multiple variables on the same memory block. Therefore a parallelisation has the effect that just more memory blocks are needed without actually filling them. With this restriction an implementation of a wider IO bandwidth comes at the cost of dropping other memory occupation by the compute units such that the compute performance suffers from it. Therefore this optimisation is not a serious alternative.

##### 4.6.9. Optimisation of the Last ConvDet Convolution

In terms of the full algorithm performance the last *ConvDet* layer represents the handicap. Due to the severe quality loss the activation of the layer cannot be quantised to 8 bits. One interpretation is that different output layers act on different value ranges, which destroys the ability to quantise the layer. Namely the output layers are divided into class probability, objectness and bounding box scaling which are again arranged for all different bounding box anchors. The different channels have quite a different value range which is not desirable for the quantisation.

##### Weight Scaling for Precision Enhancement

The idea we came up with, was to scale the respective channel values in order to have a comparable value range for the different classes. This was done by weight scaling such that especially the small range values would be increased in order to have the same range as others and therefore increase its precision after quantisation. The problem that we



#### 4. Design Implementation - Embedded Object Detection Platform

encountered was that the weight values themselves are quantised and were already at a considerably high value in the applied range. This would impose that in order to scale them by a factor of e.g. four relative to the other values, all other channels had to be decreased by a respective factor. This implied that all the other channels that would be scaled down would be quantised with an even smaller precision as otherwise. An implementation of this setting showed even worse performance than the one with simple 8 bit quantisation. A direct hardware implementation would have brought along quite some additional resource needs, since all compute units would have to compute their own scaling factor and apply it to the activation.

##### Adaptation of the 3x3 Convolution Accelerator

Eventually the last convolution layer (ConvDet) was also accelerated with the 3x3 hardware accelerator. Some modifications were necessary which used some logic resources, but not too many. The main modifications included the choice of running a ReLU at the end of the computation and the switching to a 16-bit output mode. In that mode the sum is only quantised to 16-bits and respectively outputted in this precision. The other adaptation that had to be done was to de-concatenate the feature map of layer eleven, since the 3x3 convolution accelerator would only support up to 96 input channels. This implied that 8 partial computations are launched and then a summation of the respective intermediate sums has to be performed. This last step is quite costly and takes longer than the actual convolution computation, but the speedup is still significant and around a factor 5. (This is without accelerating the last step with multiprocessing)

##### 4.6.10. Reference Implementation with ARM Compute Library

Eventually also a reference implementation was done with the ARM Compute Library. This would help to compare our performance gained to an industry standard of simple performance. Since in the code structure the code was quite clumsy and elaborate only representative layers were implemented and compared.

The main drawback using the ARM CL was that it would only work for single-precision floating point numbers. This implies that it was not compatible with the rest of the algorithm running with 8 bit quantisation and could not be used to accelerate single elements like the pooling layer or the last *ConvDet* layer.

Since the performances were comparable to the OpenMP accelerated quantised computation the ARM CL was not an option to use in our design.

## 4.7. Power Measurement Setting

One main task of the project was also to gain data on the energy and accordingly on the power consumption of the designed system. In principle a detailed measurement of the consumption of every running part was the goal even though it was clear from the beginning that this would be challenging.

One available method was the measurement over the installed PMBus (Power Management Bus). This bus collects the running data from installed Maxim DC/DC converter controllers which incorporate voltage and current measurement. These data items are then read out over the PMBus to a PC over a specific USB interface. The drawback of this convenient power measuring setting is that the resolution is bad. The application would only be able to collect up to around 15 samples per second. For the application running at 300 MHz it is not a very detailed recording.

Table 4.1.: Main Available Power Domains on the PMBus

Section	Power Domain	Voltage
PS Full Power Domain	VCCPSINTFP	0.85 V
PS Low Power Domain	VCCPSINTLP	0.85 V
	VCCOPS	1.8 V
PL Domain	VCCINT	0.85 V
	VCCBRAM	0.85 V
	VCCAUX	1.8 V
Board Level Domain	DDR4_DIMM_VDD	1.2 V
	UTIL_3V3	3.3 V
	UTIL_5V0	5.0 V

To overcome this insufficiency we wanted to measure the power with an oscilloscope. However some jumpers for current probing or some shunt resistors for indirect current measurement were needed. The ZCU102 board effectively contains a variety of shunt resistors that are built in for a separate I2C based current measurement system. Those shunt resistors however have a resistance of only 2-5  $m\Omega$ . At a current level of a few Ampère this would result in only a few Millivolt. When sensing this voltage with the oscilloscope, the voltage would clearly be predominated by the high frequency noise of around 10 mV.

In order to enable high frequency measurements and also because of the fact that the total board power dissipation was not clear at that point, we built a small connector for

#### 4. Design Implementation - Embedded Object Detection Platform

the power plug with a current sense resistor of  $0.1\ \Omega$  in series. With this setting the voltage was much better visible and the total power measurable.

For the power measurement both techniques were eventually used and the detailed chip domain specific values had to be acquired with the average value of a specific unit running in a loop.

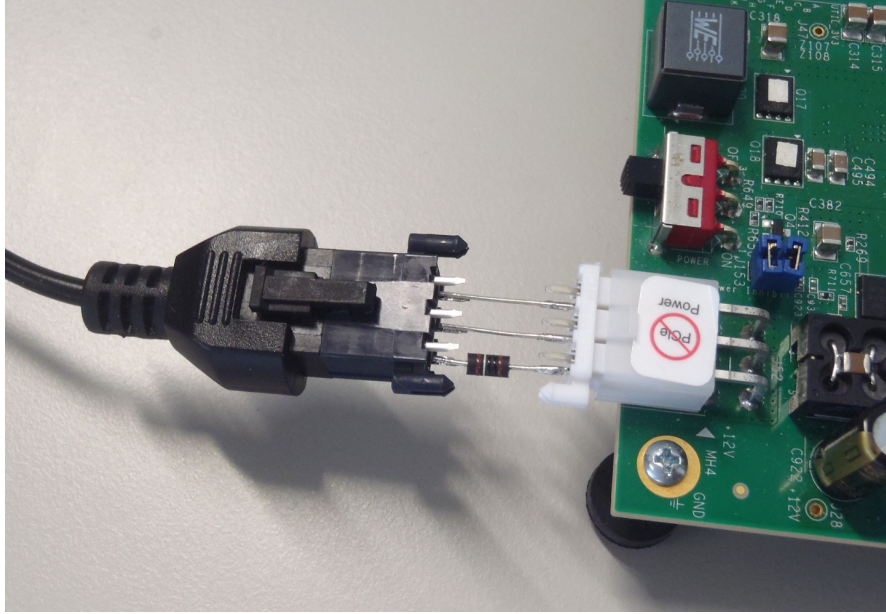


Figure 4.2.: Power Plug Current Sense Setting

#### 4.8. Confinements of using a HLS Synthesis Tool combined with a High-Level System Design Platform like SDSoC

The setting of EDA tools we used included as earlier described the system design platform SDSoC and the high-level RTL synthesiser by Xilinx. Both tools are relatively new so that they would not perform in full coverage as a completely mature software. Below are listed some drawback that we encountered during the project, which kept us busy or would be desirable if they would be implemented or improved respectively.

**No Section Jumping with Sequential Memory Access** The desired sequential memory access, that enables minimal data access latency, disabled a memory write process that would implement a part of two concatenated feature maps. The need of jumping on the memory section from time to time made it impossible to implement it in hardware. Another algorithmic solution had to be found to counter this

#### 4. Design Implementation - Embedded Object Detection Platform

problem. It would be desirable to have this option available even for a certain timing overhead when jumping. Often a purely sequential memory access comes along with other tradeoffs.

**No Feedback on Memory Stream Interface** Following the inconvenience above an annoyance was added in the design flow because the sequential access happens with a normal C array read statement. However because of the above problem the program doesn't run in an equivalent manner and in hardware the array index is just ignored. This is quite misleading especially because it renders a different result compared to the program run on the CPU.

**Not Usable Optimisation for 8 Bit Fixed Point MAC Processing** As mentioned in the architecture chapter Xilinx claimed to be able to run two 8 bit MAC on a single DSP [20]. We were able to see this optimisation with a design with a very small amount of parallel MAC computations. Further as described multiple times the limiting factor when implementing this quantised scheme was not the number of DSPs but the LUTs, which makes it somewhat irrelevant whether it is possible to compute more units per DSP or not. It might be possible that the compiler is not able to perform some optimisations when the design has a certain size, because we were able to observe that even though the LUTs were the limiting factor, the accumulator of the MAC was still made in LUTS outside the DSP.

**Parallelism over Width** In a first attempt we tried to parallelise the computation over the width. This is especially desirable because it enables a sequential memory read and write as the accelerator advances in the buffer. However the width of the buffer would have needed a parallel structure in the size of the parallelism. Basically this structure can be instantiated with Vivado HLS using the *cyclic* property for the array partition. However because the width has a certain size the compiler would not be able to achieve a parallel data access in order to compute multiple items concurrently. This is quite unsatisfying especially since the compiler annotation exists, but the compiler is apparently not able to use this structure to build up a system on it.

**Block RAM Not Shareable for Different Arrays when Partitioned** When partitioning a certain array basically the array is split over multiple BRAM instances in order to provide enough parallelism. In one design iteration the parallelism was increased and with it the whole memory occupation rose. Unfortunately this made the whole design fail to compile when merging the specific accelerator with the second accelerator with Vivado. This is a pity since the resources would actually suffice. The problem as it looks is that the compiler does not aggregate two arrays on the same BRAM instance when they are used for another arrays parallelism even though it has still some space left. As mentioned before the bottleneck is in the Vivado compilation, however it is still a pity.

**Tracing the Hardware Accelerator** SDSoC provides a tool for tracing an application, which inserts hardware modules and software modules into the application in order

#### 4. Design Implementation - Embedded Object Detection Platform

to trace the application. However when using it with the our design some inconveniences appeared. Firstly the trace buffer would not suffice and it would overflow at the second layer, including some corruption of some data. The first layer is fine but there, no precise information is available on the data transfers during execution. The streamed IO data transfer is displayed as one large block, without giving information on the latency of the specific read requests/operation by the accelerator. One gets an idea of the time breakdown of the function call but we were not able to compare the different layers because of the buffer overflow. Eventually not too much could be read from the application tracing.

**Parallel Unaccelerated Version of Program for Development** One problem that arose during the development of the application was that when choosing the design mode *Release* in SDSoC and pick the accelerated functions, one would not be able to run the code without HW accelerator in a way that the design (with HW accelerator) would be stored for later use. The only option available is to run it in the *Debug* design mode, which however compiles the program in debug mode and makes it very slowly. In my case even more desirable would have been to easily build the project for execution on the host PC. This, because I had the problem that the board would sometime not boot up for quite a long time after being switched on for a certain time.

# Chapter 5

## Results

### 5.1. Network Quantisation

#### 5.1.1. Training and Evaluation Environment

The setting used for the fine-tuned training and the accuracy evaluation is the environment provided by Bitchen Wu’s SqueezeDet repository on Github. The underlying image set is the KITTI set that was randomly divided into two sets for training and validation by the supplied scripts. As a starting point for fine-tuning the provided checkpoint was used. Note that the checkpoint was not learned on the same partition and therefore the relative performances on the training and the validation set are not typical. This is not a problem, it just manifests itself in the accuracy levels.

#### 5.1.2. Quantisation Precision

In a first step the impact of different precision choices on the quantisation process were examined. The underlying quantisation procedure is explained in section 3.2.3. As described in chapter 2 different other literatures found that in the network a small accuracy loss could still be achieved with a fixed-point precision of down to 8-bit. Further lowering precision would then show a sudden and considerable loss in accuracy that significantly impairs the quality of the algorithm.

The experiments showed a very similar result. In figure 5.1 we see that the different precisions imply different accuracies, which again can be retrained by a significant amount if the precision drops by too much. We also see that down to 8 bits the accuracy is only affected very slightly compared to the floating-point benchmark. It must be stated that a considerable accuracy loss can be recovered with fine-tuning of low precision

## 5. Results

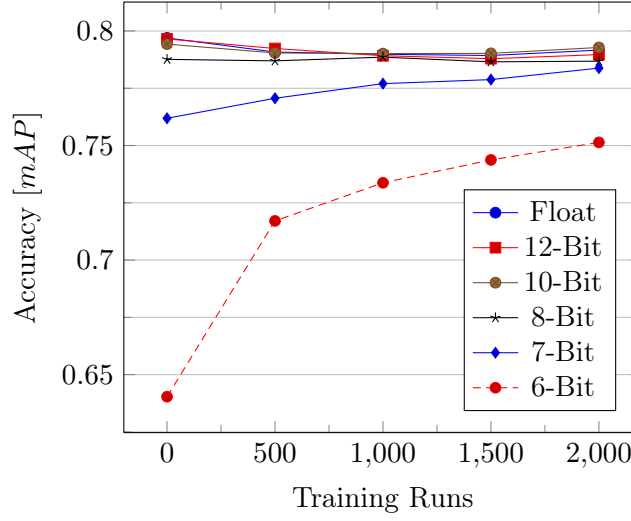


Figure 5.1.: Accuracy of SqueezeDet Network after Different Weight Quantisation on Training Set

weights. Nevertheless the reachable accuracy is still notably worse than higher-precision quantisation. Here only network weights were quantised.

### 5.1.3. Bit-Precision Choice

The findings lead to the choice of an 8-bit quantisation. Next to it being the lowest precision quantisation showing no significant accuracy drop on the larger scale (Figure 5.1) the choice was endorsed by the fact that 8 bit is the word length of the majority of todays computation systems. This facilitates the data handling significantly.

### 5.1.4. Fine-Tuning

Following this choice further evaluations were examined. First the fine-tuning of quantised weights was investigated. The detailed procedure is described in more details in section 3.2.3. The general effect and effectiveness of fine-tuning is already visible in figure 5.1 at low precision quantisations. Nevertheless also with the chosen 8-bit quantisation a notable improvement could be seen. Clearly the benefit is small since the potential is also smaller because of the accuracy being closer to the full precision level anyway. Figure 5.2 shows the steady improvement over a learning session of 10000 learning steps.

A total in retrained accuracy of 0.5% and 0.8% respectively is achieved at fine-tuning. This is about 50% of the accuracy loss that is imposed by the weight quantisation.

## 5. Results

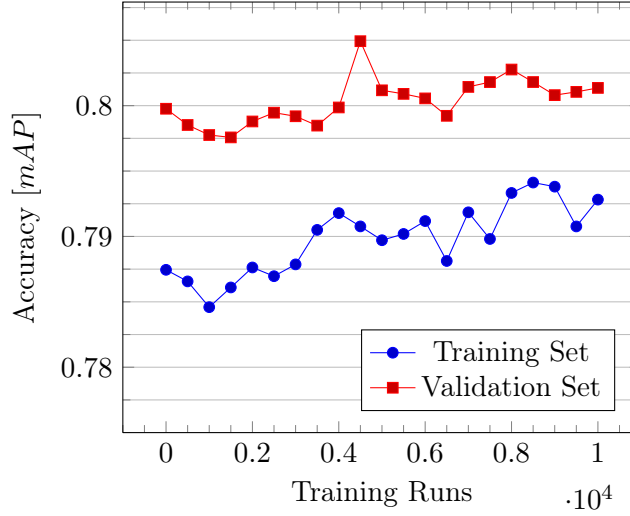


Figure 5.2.: Accuracy Progress of Fine-Tuning 8-bit Quantised Weights

Table 5.1.: Accuracy Impact of Activation Quantisation [mAP]

Quantised Activations	Training Set	Validation Set
Intermediate Activations only	79.0	80.0
Intermediate Activations and Input	78.32	78.83
Intermediate Activations and Output	74.56	75.46
All Activations	74.2	74.9

### 5.1.5. Including Activation Quantisation in the Learning Process

In a further step quantisation was also introduced on the layer activations. This was on the one hand to have a reference to the quantised processing on the platform and the idea was to also compensate for the activation quantisation by including these in the fine-tuning process. The quantisation design was essentially adapted from the *Ristretto* implementation for *Caffee* framework. In its implementation they keep activations at full precision in order to enable analytical computation of the error gradient with respect to each other gradient. Only the last layer activation, on which the scoring is applied is quantised. We investigated different ways of learning with quantisation on the activations starting with an overview of the accuracy affection induced. The setting was chosen to perform 8-bit quantisation as well. This choice is motivated by results in literature and the fact that with 8-bit quantised weights and an 8-bit coded input most of the crucial data elements share the precision.

Table 5.1 shows the rough impact on the network accuracy. The numbers show a clear precision sink for including the quantisation of the output activations to the network. The first idea was that a training or fine-tuning could compensate the loss. So a learning



## 5. Results

setting was applied with only one intermediate layer activation quantisation in order to gradually improve the quantisation effects. It showed that when training the network with quantised layer activations the learning algorithm would quickly drift away into a non-convergent mode. This would already manifest in the first checkpoint after 500 runs where the accuracy would drop to a few percentage points.

The exact origin of the training divergence was not investigated in further details. The main assumption that is made, is that the training algorithm can not cope with the imprecise gradient calculation of the activation quantisation module. Since the quantisation function is not continuous, no analytical gradient can be computed and the gradient was simply handed over unchanged. This fact lets the learning algorithm misbehave and it was observed that on many layers it would fall into the truncation interval where it would not get out again.

In a second step the approach chosen by P.Gysel for *Ristretto* was used to only use quantisation on the output layer activation. This resulted in the behaviour showed in figure 5.3. An even poorer performance with no learning effect showed to be true. Apparently the network would not work properly with this choice of activation quantisation. This note and the findings in table 5.1 show that the weak point is the output layer activation. It showed that the precision of 8 bit chosen for the activation quantisation would not suffice in that case. Therefore two further evaluations were performed with extended quantisation precisions on the last layer. Figure 5.4 and 5.5 show that the extension of the last layer's bitwidth brings along a large improvement on the accuracy in general and an improvement of the training capability. Whereas a quantisation of 11 bits improves the accuracy of the system by a large amount, it still does not improve over the training cycles. Compared to the scenario in figure 5.3 at least it does not loose accuracy. In the computation with 16 bit quantisation of the last layer we then can observe that the network is again able to improve over the learning cycles. Also the accuracy has again improved by a very small amount.

Satisfyingly the accuracy level achieved in figure 5.5 with an output layer quantisation of 16 bits is in the same accuracy range compared to the fine tuning of the system with full precision on this activation. Therefore we can state that with a 16-bit quantisation no significant loss has to be expected.

### 5.1.6. Final Results used for the Platform Design

In a final fine-tuning run with the lastly stated configuration of 8-bit quantisation of the weights and 16-bit quantisation of the output layers activation the following final accuracy levels were achieved. (Table 5.2 or Figure 5.6)

The result shows that with the introduction of the 16-bit activation the accuracy loss of switching the network to a 8-bit quantisation on weights and activations can be kept to half a percent with the help of fine-tuning. On the training set the loss is 0.2% and on the validation set it is 0.5%. This is a satisfying result for the reduction of memory

## 5. Results

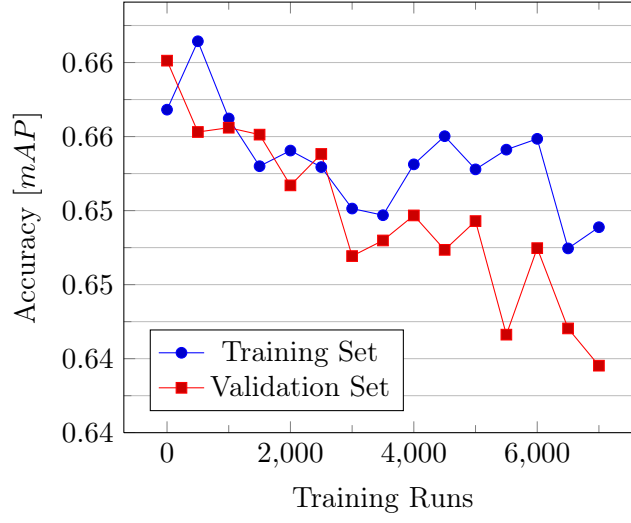


Figure 5.3.: Fine-Tuning with 8-bit Quantised Output Layer Activation

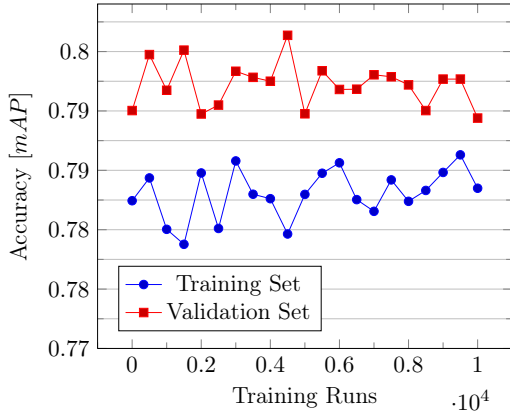


Figure 5.4.: Fine-Tuning with 11-bit Quantised Output Layer Activation

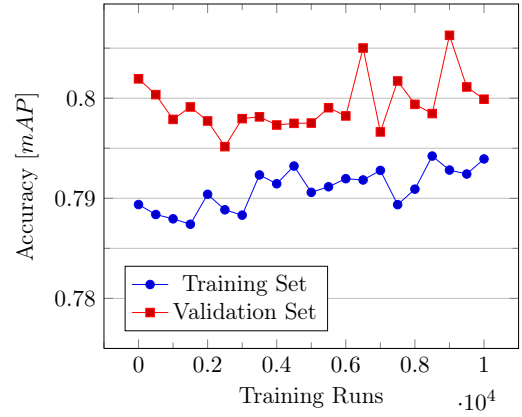


Figure 5.5.: Fine-Tuning with 16-bit Quantised Output Layer Activation

Table 5.2.: Accuracy of Final Network compared with Initial Network and Untrained Network [mAP]

Setting	Training Set	Validation Set
Full Precision Network (No Quantisation)	79.7	80.6
Quantised Network, no fine-tuning	79.0	80.1
Quantised Network, with fine-tuning	79.5	80.1

## 5. Results

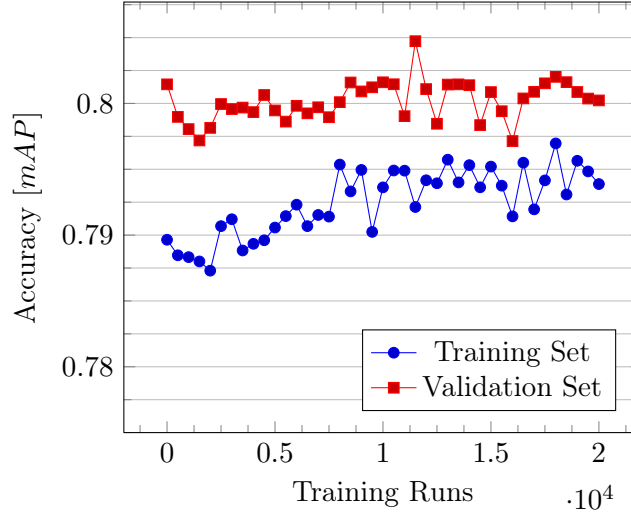


Figure 5.6.: Final Fine-Tuning with Quantised Weights and 16-Bit Quantised Last Activation

used for on-chip weight storage and the memory transactions for the layer activations of a factor 4.

## 5.2. Algorithm Implementation

### 5.2.1. Pure Software Implementations

The first software implementations included a pure single-precision floating point implementation, an 8-bit quantised (with 16-bit activation on the last layer) implementation and each time an OpenMP accelerated version. Additionally part of the algorithm were also implemented with the ARM compute library. Figure 5.7 shows the general computation share of the different layers and also the respective computation times as percentage of the total. The values show that the flat layers (early layers) run less efficiently than the deep layers that come at the end. It is visible since the share of the computation time is less than the actual computation share.

The total network runtimes are listed in table 5.3 with the runtime values of the layer 7. Also the relative speedup to the floating point implementation measured at the 3x3 convolution are given.

## 5. Results

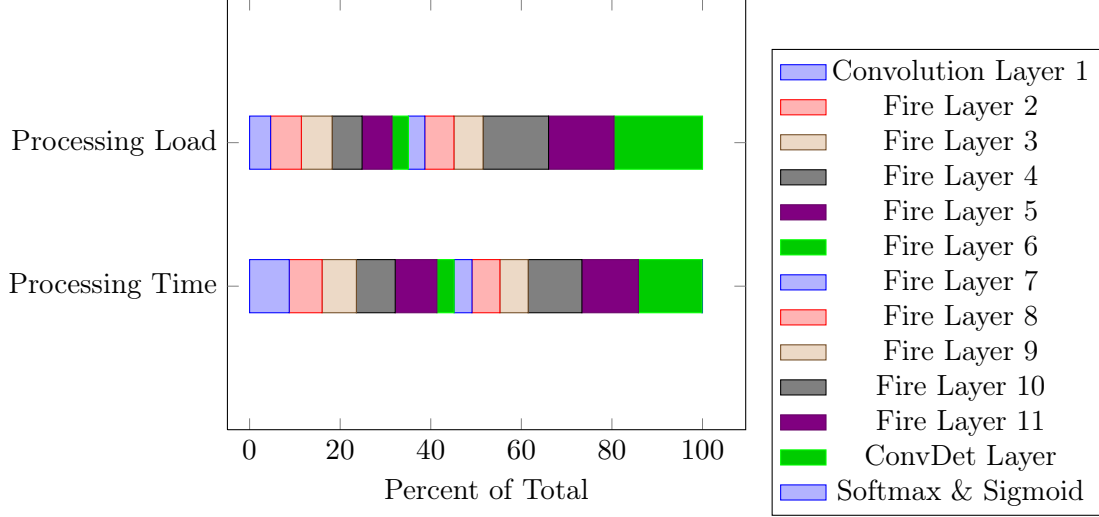


Figure 5.7.: Runtime vs. Computation Share of Floating Point Implementation

Table 5.3.: Runtimes for Total Network and for Layer 7

Implementation	Full Network	Squeeze 1x1 Layer	Expand 1x1 Layer	Expand 3x3 Layer	Speedup
Float	23.5 s	134 ms	72 ms	629 ms	x 1.0
Float OpenMP	6.1 s	34 ms	18 ms	157 ms	x 4.0
Quant (8-Bit)	6.8 s	37 ms	27 ms	203 ms	x 3.1
Quant OpenMP	2.1 s	9 ms	7 ms	51 ms	x 12.3

## 5. Results

Table 5.4.: Degrees of Parallelism in Convolution Accelerators

Dimension	1x1 Convolution Accelerator	3x3 Convolution Accelerator
Ouput Channels	32	64
Height	2	8
Width	1	1
Input Channels	2	2

Table 5.5.: Resource Use for Final Design

Design, Level	BRAM	DSP	FF	LUT	CLB
1x1 Accelerator, HLS	14 %	5 %	3 %	11 %	n/a
3x3 Accelerator, HLS	64 %	61 %	27 %	83 %	n/a
Total, HLS	78 %	66 %	30 %	94 %	n/a
Total, Vivado	90 %	68 %	29 %	70 %	91 %

### 5.2.2. Hardware-Accelerated Implementations

#### Accelerators Dimensions

Eventually the quantised implementation was complemented with two convolution hardware accelerators. One 1x1 convolution accelerator and one 3x3 convolution accelerator. The 3x3 convolution accelerator was parallelised by a larger factor because the workload is also considerably larger. Table 5.4 shows what parallelisation was chosen for the final design. The choice bases on a trade-off between resources, and general algorithm runtime improvement. The parallelisation on the height is different on the two accelerator since the 3x3 convolution accelerator has feature map buffers and the 1x1 convolution accelerator doesn't. More details on the architecture can be found in the respective chapter.

The total resource usage can be found in table 5.5. The distinctive difference in usage of BRAM modules in the HLS report and the final Vivado report is most probably due to the extensive need of BRAM modules in order to parallelise the memory access and the respective module are then not actually completely filled.

#### Accelerators Compute Performance

The compute performance of the implemented hardware accelerators is illustrated in figure 5.8. The significantly larger performance of the 3x3 convolution accelerator is clearly visible and also the difference between the squeeze and expand 1x1 convolution.

## 5. Results

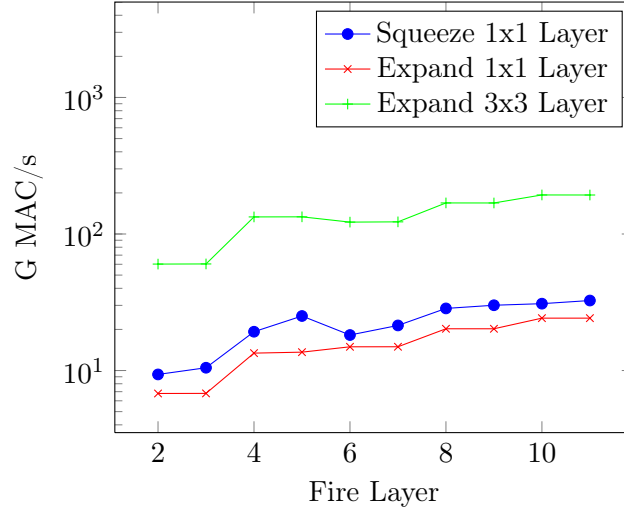


Figure 5.8.: Performance of the Hardware Accelerators on the Different Layers

The difference here originates in the different compute dimensions of the convolution, which in the case of the squeeze layer has many input channels that can fill the most inner pipeline very well.

In order to estimate the compute-resource efficiency of the algorithm the following two roofline plots were generated. The red line shows the theoretical maximum performance of the algorithm for perfect conditions. The top of the roofline is calculated by multiplying the number of compute units of the design with the used clock, in our case 300MHz. The ramping area of the roofline is the area where the design is memory-bound, that is it can't perform more calculations because of the respective need for new data. The different datapoints stand for the different layers, where each has a different compute effort relative to its data exchange needs.

### 5.2.3. Tracing

SDSoC enables to trace the application with the help of software and hardware tracing modules that monitor system call timings and data movements. One problem that arose was that the trace buffer seemed to be too small to trace the full application such that only layer one could be traced. The trace shows the small time consumption for the weights transfer (as expected) and the shifted input read and output write. However the timings are not too clear. The 'output wait' system call varies in length on different function calls. It seems that it is the time of the processor performing a context change in order to return to the executing function.

## 5. Results

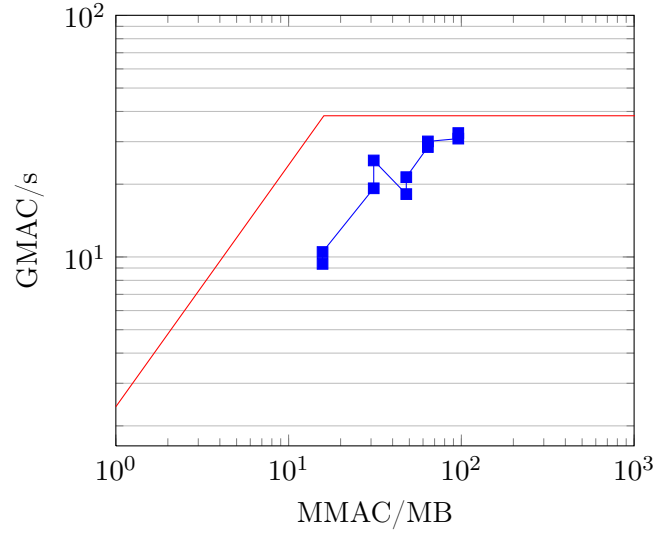


Figure 5.9.: 1x1 Convolution Squeeze Layer Roofline Plot

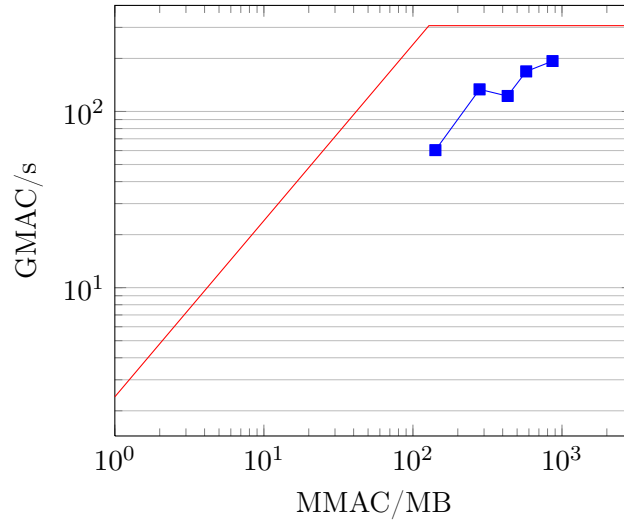


Figure 5.10.: 3x3 Convolution Expand Layer Roofline Plot

## 5. Results

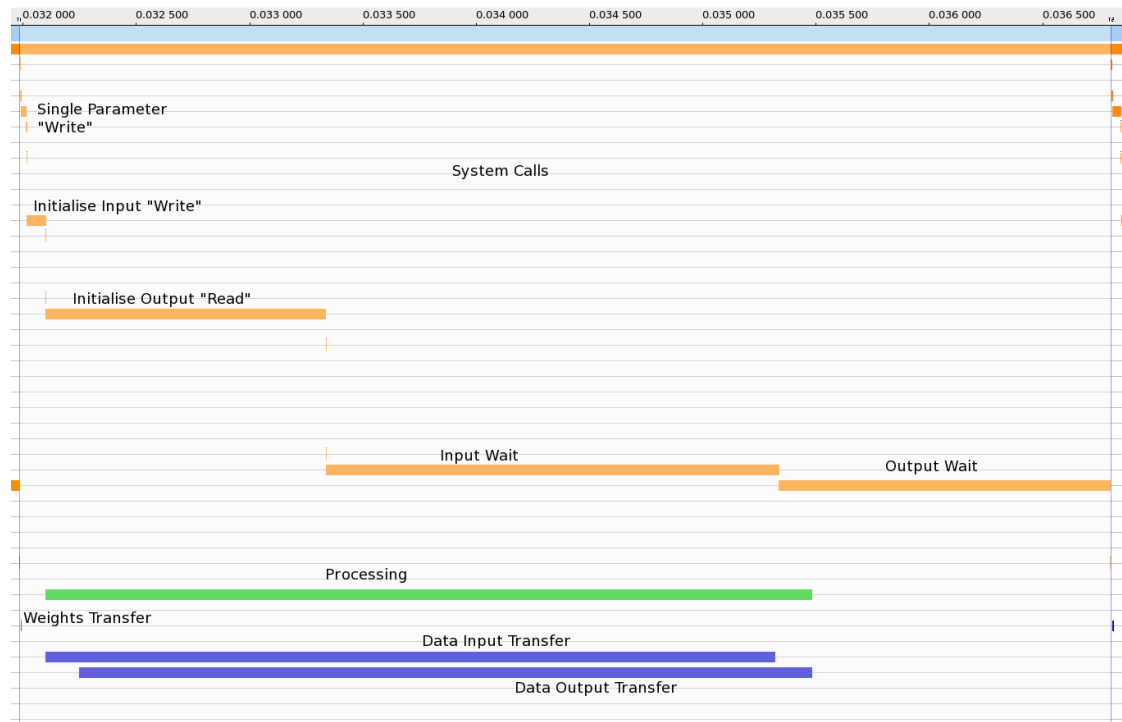


Figure 5.11.: Example Trace of Partial Image Processing of L1 on 3x3 Convolution Accelerator



## 5. Results

Table 5.6.: Power Measurement over PMBus in W

<b>Power-Domain</b>	<b>Idle, no Bitstream</b>	<b>Idle, w/ Bitstream</b>	<b>Full Network Computation</b>	<b>Only 3x3 Accelerator</b>	<b>Idle, /w USB Cam</b>
PS Full Power	1.2	1.2	1.3	1.3	1.2
PS Low Power	0.3	0.3	0.3	0.3	0.3
PL Main	0.8	3.5	4.3	4.9	0.8
PL AUX	0.3	0.3	0.3	0.3	0.3
DDR RAM	0.6	0.6	0.7	0.7	0.6
Utils 3.3 V	4.2	3.2	4.0	3.9	3.4
Utils 5.0 V	0.1	0.1	0.1	0.1	1.8
Total	7.4	9.1	10.9	11.3	9.1

### 5.2.4. Power Measurements

The systems power was measured in two ways, one over the 12 Volt power plug and one with the help of the PMBus as described in the implementation chapter.

#### PMBus Measurements

The measurements of the different intermediate power domains on board are listed in table 5.6. Measurements were performed in different settings including average power usage during full network computation, during iterative accelerator execution and in the idle state. The numbers were in some cases rather difficult to fix since the the numbers (the average computed by the the Maxim tool) varied by a considerable amount. This was especially the case for the PL main domain since the accelerator is launched and stopped regularly as well as the 3.3 Volt utils domain where the Ethernet controller is attached. Further must be stated that we were not able to measure the PL BRAM domain since probably the PMBus had a failure.

#### Total Board Power

Since the PMBus only gives information on the intermediate power levels, the total board power had to be measured separately. In order to do that, a small current sense adaptor was constructed that could be attached between the power supply and the power connector of the board to measure the current. Figure 5.12 shows a total compute cycle of the network measured on an oscilloscope. The executions of the different layers with the corresponding accelerator launch are clearly visible and also marked in the image. At the edge reference power coordinates are inserted that perform the conversion of the

## 5. Results

measured voltage on the current sense resistor to the actual power that is fed through the plug.

The offset in power that is visible compared to the values in the table above is due to the idle board power consumption. An idle power measurement at the power connector showed a power dissipation of 16.8 W. This measurement was made with no OS booted (no SD-card inserted) compared to the idle measurements in the table. This means that the values of the above table can be added to the board-bias value to get the numbers measured with the oscilloscope.

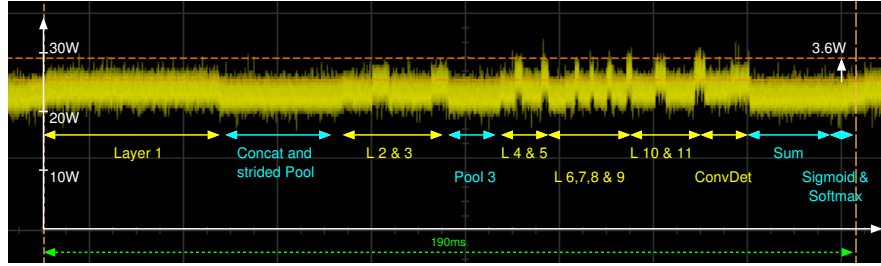


Figure 5.12.: Oscilloscope Chart for Total Power Measurement

### 5.2.5. Comparisons

In order to put the resulting design into perspective some other implementations were considered. Those include a complete single precision floating point implementation, an extension of it using OpenMP for multiprocessing on the PS, the final 8-bit quantised implementation without hardware accelerator and a accelerated version with OpenMP and finally an implementation using the ARM compute library with its optimised convolution accelerator making use of multiprocessing and the NEON SIMD accelerators.

Their respective compute performance and their power consumption are illustrated in a graph in figure 5.13 measured on the 3x3 convolution accelerator at the expand layer 7. That expand layer has a feature map size of 76\*22 pixels, 48 input channels and 192 output channels.

The graph shows that still a significant performance gain of a factor 39 can be achieved compared to the ARM CL accelerated computation, that is already a factor 23 faster than the simple float implementation. It must be stated that however the ARM CL works with floating point and not on quantised fixed-point numbers. Also the improvement in computation-per-power efficiency is visible with the help of the blue lines that indicate constant MAC/s/W levels. Here also the accelerator outperforms other implementations by far.

Another comparison that we want to make is to compare it to other platforms. Therefore five implementations/platforms are checked against each other on terms of compute performance and performance-per-Watt efficiency in table 5.7.

## 5. Results

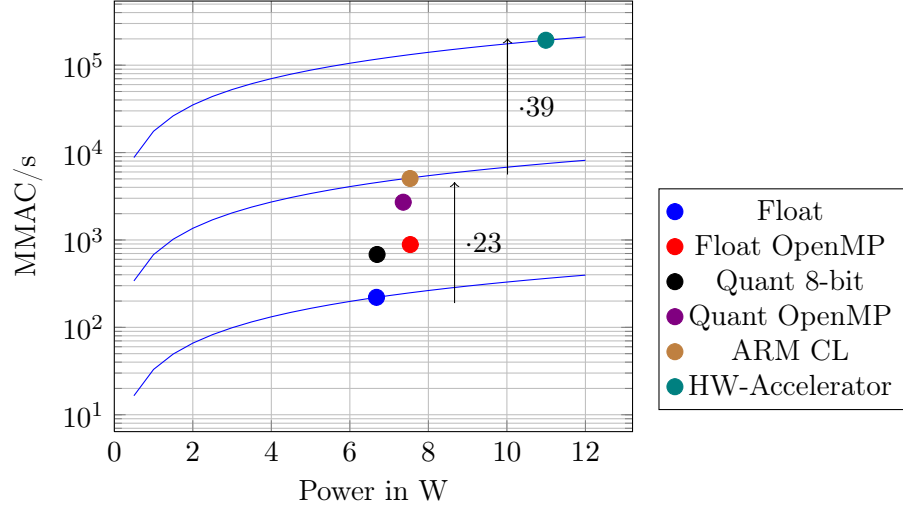


Figure 5.13.: Processing Efficiency of different Calculation Schemes

Table 5.7.: Comparison of Different Platform Performances

Platform	Theo. Gop/s	Peak Gop/s	Gop/s/W	Datatype
ARM CL, own		9	1.2	float32
Zynq MPSoC, own	614	386	33.2	fixed8
Zynq 7000 [21]	345	255	19.5	fixed16
Tegra K1 [22]	365	95	8.6	float32
Tegra X1	512	423	23.4	float32

## 5. Results

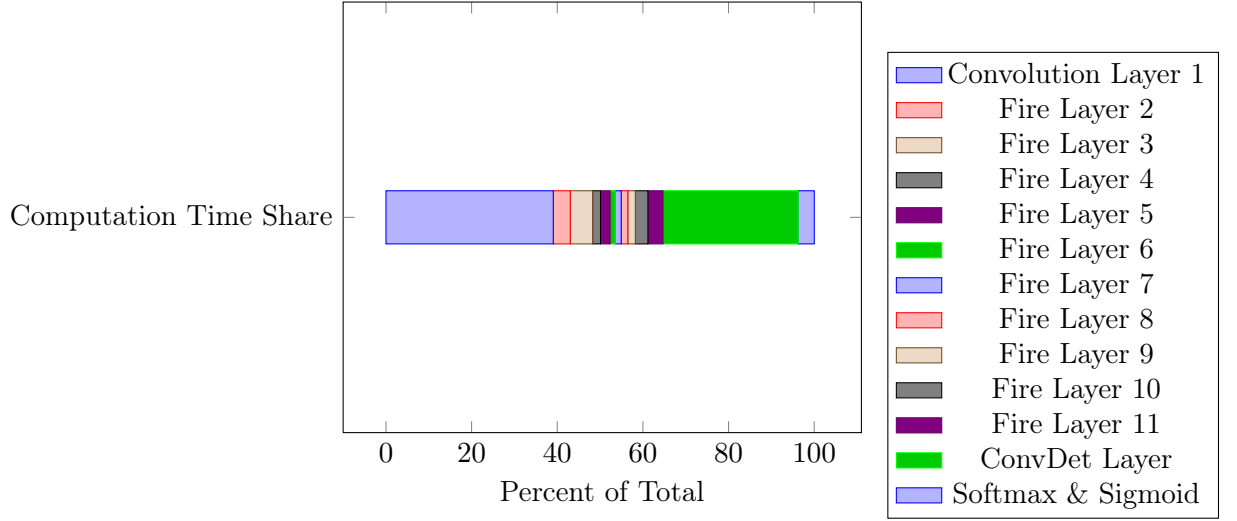


Figure 5.14.: Computation Time Share of Final Implementation

Table 5.8.: Final Design Performance for the Full Network

Runtime	Frames per Second	Power	Accuracy
190 ms	5.2 fps	10.9	79.7 % mAP

The comparison shows that this hardware accelerated version clearly outperforms designs of earlier implementation on the old Zynq 7000 chip by means of both performance and energy efficiency. Also the values are fairly better than the Tegra K1 mobile GPU that is even worse than the old Zynq implementations. Compared to the more recent mobile GPU Tegra X1 by NVidia the performance is comparable. The X1 is slightly better in compute performance but also worse in energy efficiency.

### 5.2.6. Final Algorithm Performance

Looking again at the complete algorithm, the algorithm improved in performance by a factor of around 33 compared to the floating point implementation of the whole algorithm. Eventually the first and last layer clearly stand out, because their dimensions exceed the accelerators maximum dimensions, such that in both cases some additional computation has to be done in order to get the intermediate results from the partial computation on the accelerator in order to compute the correct values. This however takes a lot of time since it is costly PS computation. The final performance values are listed in table 5.8

# Chapter 6

## Conclusion and Future Work

### 6.1. General Conclusion

With the final implementation of the object detection algorithm we were able to accelerate the two main building blocks, the 1x1 convolution and the 3x3 convolution, on the programmable logic of the SoC in order to achieve 5 frames per second on a 11 W budget. In architectural purposes we used the vast amount of block RAM on the programmable logic to save the full layer weights on chip and buffer multiple feature map rows in order to achieve a parallelisation of 1024 for the 3x3 convolution accelerator. The whole algorithm would run on 8-bit fixed-point numbers without giving up on the accuracy on more than 0.5 % mAP of accuracy. With this setting, a full working platform could be set up with a camera and a network connector such that the platform would be able to remotely perform object detection on a low power budget and forward the information to a remote server. Sending whole images with bounding boxes over the network would introduce some serious latency because of unknown inefficiency in the networking subprogram.

In terms of general compute performance we were able to reach a new performance level with the new Xilinx SoC that would clearly outperform solutions with earlier chip generations and reach comparable performance to modern mobile GPU platforms. However there are still some open points that could optimise the design a little further. Some of them are described below with other interesting suggestions for improvement.

### 6.2. Accelerator Utilisation

In our implementation we have on the one side the hardware accelerator that runs the most efficient if it has many input channels in order to fill the MAC pipeline, on the other side we have the algorithm structure that is designed to have a small network and

## 6. Conclusion and Future Work

therefore also reducing respective feature map dimensions. In our case especially the expand layer that makes up most of the computation is always launched after a squeeze layer that reduces the feature map channels. This therefore reduces the input channels of the following expand layer and makes the hardware accelerator run less efficiently.

It is difficult to make a statement that it would be desirable to have more feature map channels in order to run the accelerator more efficiently, especially since more compute effort is still more compute effort, even though it is computed more effectively. However this awareness could help to structure algorithms to gain depth more quickly. Or another way would be to shift the depth from layers at the end to the ones in the beginning in order to improve their performance and flatten the layers relative depth. This however has to happen on an algorithm development side which is difficult to do since algorithms are rarely designed for a specific hardware platform. However since the MAC is a general convolution element, a large accumulation size should have a positive impact on the performance for various designs.

### 6.3. Memory Arrangement

One downside of the implemented design is that the activation I/O to and from the buffers happen in a sequential manner to the processing block. A more desirable way would have been to read and write from and to the buffers during the computation. This would be possible for example with a line buffer that continuously reads new data making sure that enough data is around for the 3x3 convolution. However with the used design framework and the current HLS version a parallelisation over the feature map columns was not possible as mentioned earlier. The resulting parallelisation on the feature map rows however necessitated that data from multiple rows are available for computation simultaneously. However with the sequential memory access that we use to keep the memory latency down, pixels from multiple rows cannot be accessed without reading the full rows in between. This makes it inevitable to perform the IO separate to the computation.

Of course a call for a different memory arrangement comes quickly to mind. The used setting of feature map channels in a first dimension followed by the width, the height and the output channels imply that the full memory section has to be read before performing the computation on that memory section. This is also valid for outputting the calculated values. In our setting the ideal case would be to read first the rows, and then the feature map channels and the columns. However when using the current design one still has to jump on the memory since only 8 rows are read to the buffer at the same time. Changing that we would lead again into the same situation as the current. The needed jumping however is not possible with the sequential memory access.

A solution that is not yet available would be to enable 'jumping sequential access', where the DMA is told what jumping pattern exists such that a quasi sequential access can be

## 6. Conclusion and Future Work

scheduled to minimise access latency. Another idea would be to have a sort of memory reorder unit in hardware that would take care of rearranging the data after the data is read out just in the way it is processed, in order that it is again available for reading in the desired order later on.

### 6.4. Future HLS Releases

During the design of the implementation we encountered multiple small inefficiencies, where Vivado HLS would not make or be able to make the desired design that we had in mind. One example is the aforementioned parallel memory access on the buffers columns. Further, for example, the addition of the MAC is performed in separate logic even though the DSP has the needed resources to run it on the same DSP slice.

The expectations are that the following versions of the framework, that are going to be released, will make up for some of the missed features. Already the new release 2017.1 features as described in the *Implementation* chapter the possibility to have larger packed structs in order to increase the bandwidth of an array. Such improvements can be expected since the program suite is not yet matured and it is still developing.

### 6.5. ARM Compute Library with Fixed Point Support

One observation that could be made is that the ARM compute library already improves the performance by a considerable amount, compared to the standard float implementation. This, even though the computation is still performed with single precision float numbers. With that in mind a fixed point implementation would be desirable since this would probably again boost the performance in a comparable way to the standard quantised implementation (factor 3) because the computation is still memory-bound. With 8-bit quantisation 4 times more data elements can be stored in the same cache and boost up the performance.

### 6.6. DSP Slices with Logic Shift Support

One observation that we were able to see was that the logic shift operations on the accumulated values introduced a significant logic resource utilisation. Even though the DSP slices have an arithmetic unit, it only features bitwise logic operations and not logic shifts. The savings on the 3x3 Accelerator would be in the range of one forth of the total LUT utilisation when the logic shifts could be transferred to the DSP slices. It is especially desirable since on the PL a lot of DSPs are still unused and the freed logic resources could theoretically be used to further parallelise the compute structure.

Appendix	<b>A</b>
----------	----------

Task Description





Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

Integrated Systems Laboratory

Master Thesis at the  
Department of Information Technology and  
Electrical Engineering

for

**Philippe Degen**

**Development of an Embedded Object Detection  
System on the Zynq SoC Platform**

**Advisors:** Lukas Cavigelli  
Renzo Andri

**Professor:** Prof. Dr. Luca Benini

**Handout Date:** 03.01.2017 (week 02)

**Due Date:** 03.07.2017 (week 28)

## 1 Introduction

Imaging sensor networks, UAVs, smartphones, and other embedded computer vision systems require power-efficient, low-cost and high-speed implementations of synthetic vision systems capable of recognizing and classifying objects in a scene. Many popular algorithms in this area require the evaluations of multiple layers of filter banks. Almost all state-of-the-art synthetic vision systems are based on features extracted using multi-layer convolutional networks (ConvNets).

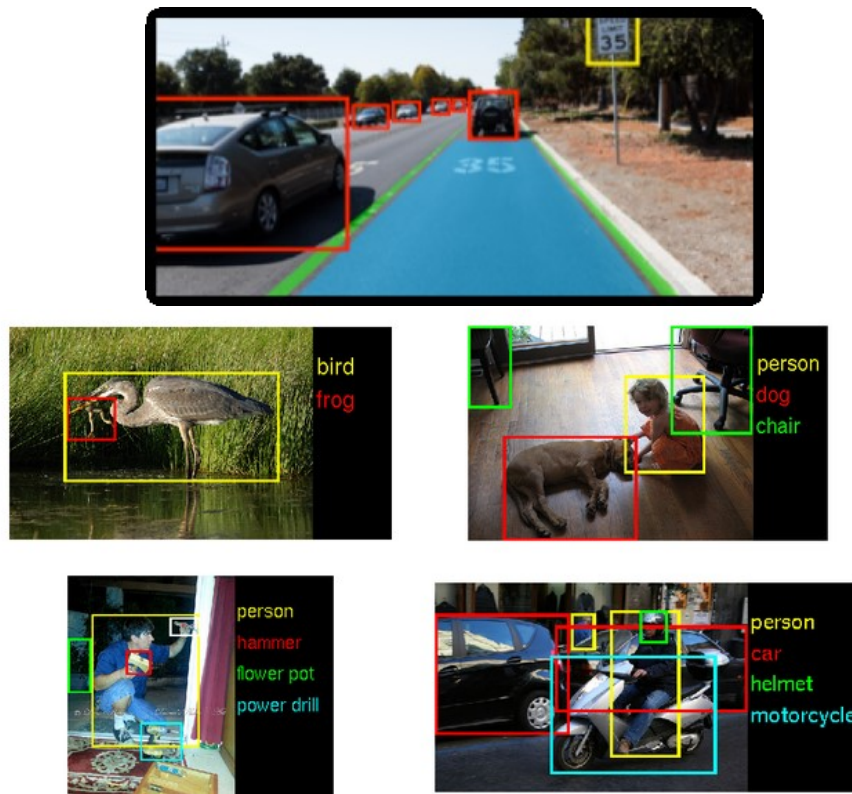


Figure 1. Object detection: Finding and classifying objects in images.

Running such object detection algorithms is computationally very demanding and yet would provide exciting new options when performed on an embedded smart camera. Such workloads can be computed [1] on (embedded) GPU platforms such as the Tegra K1 or X1 platform, but this comes at a relatively high energy cost and its fixed architecture is non-optimal because it cannot be adapted to exploit the numerical precision requirements present in CNNs and is very unsuitable to perform data compression to reduce the power-demanding high I/O bandwidth.

**Prerequisites:** Knowledge of C/C++; Previous experience in VLSI design; Interest in computer vision and system engineering; Motivation to learn high-level synthesis and software-based design concepts to join them with hardware architecture know-how.

**Character:** 10% Literature Research, 30% Software Development, 45% Hardware Development (HLS), 15% Theoretical Evaluations.

## 2 Project Goal

The goal of this project is to develop a complete FPGA+ARM-based system to perform object detection in images obtained from a camera connected to the development board. First a simple C-based implementation is developed as a baseline, which will run on the ARM cores only. The most critical functions are then accelerated by implementing accelerators for them on the FPGA using high-level synthesis (HLS).

This setup is then extended with a camera to perform this object detection in real-world scenery. The performance and limitations of this system are then assessed and improved by evaluating novel approaches to filter weight compression, quantization, or change-based evaluation.

General note applicable to tasks 1 and 2: You might be able to reuse some of the code developed in HS16 by Martin Zihlmann who has done a simpler image classification implementation using SDSoC as a semester project. Furthermore, we learn more about the limitations of such systems and might thus adapt the tasks later in the project during its course. It might also be worth to take a look at ZynqNet, a FPGA-based image classification system recently developed at Supercomputing Systems (SCS) as part of a Master thesis using HLS – also for quantization and export of trained weights, etc. [2].

## 3 Tasks

The project will be split up into four phases, as described below:

### Task 1 – Building an Object Detection Pipeline (5-6 Weeks)

1. Familiarize yourself with the methods used to perform object detection, in particular the key paper “Faster R-CNN” [3].
2. Write C code to perform all these operations. Export of the weights, biases, etc. from an existing trained object detection network and verify functional correctness. Keep it simple here: don’t use too complex C/C++ constructs and don’t over-engineer your code – otherwise it will be very painful later on with the HLS tools. Don’t forget to implement any preprocessing steps.
3. Go through the Xilinx SDSoC tutorial; take 2 days for this and do it thoroughly. Then create a SDSoC project to run your code on the processing system side of the Zynq. This is the baseline, from here we start optimizing. Obtain measurement results: throughput and energy efficiency are our primary concerns.
4. Spend a limited amount of effort on optimizing and parallelizing the software with OpenMP (basic vectorization and parallelization). Make sure you also keep a copy of the simple implementation from before.

## **Task 2 – Building Basic Hardware Acceleration (6 Weeks)**

1. We can confidently expect that the most processing intensive part of the software implementation will fall to the convolution layers. Check related work on how this layer is best accelerated [2], [4]–[7].
2. Build an architecture keeping the weights for convolutional layers on-chip (at least for one layer), and streaming in and out the images/feature maps. Make sure the operations are suitable for FPGAs (e.g. avoid adder trees used every cycle). Go through the Vivado HLS tutorial before starting with this. Use half- or single-precision floating-point types for this exploration.
3. Evaluate the performance and resource usage to check the code gets mapped to hardware as you expect it to – and fix any issues to minimize the difference. Benchmark the final combined SW/HW system. Evaluate the performance and energy efficiency.

## **Task 3 – Building a Demonstrator (1-3 Weeks)**

1. An important aspect of this project is to get to a working demonstrator. At this point you should make this step, connect a camera to the system, and adapt everything to work at the speed measured before.

## **Task 4 – Improving the System (9-11 Weeks)**

The following items in this task are options which can be kept very short or investigated very deeply. The idea is to get new and good results one or two of them, and evaluate it/them thoroughly. You will pick one when getting to this task together with the advisors and based on the most relevant issues seen in the system.

1. Using lower-precision number formats reduces the energy and resources spent for computation and buffering as well as I/O to the external RAM. Perform a thorough literature research on state-of-the-art approaches and possibly existing frameworks [8]–[10]. Make a good comparison of them and choose the best for the task at hand considering all the constraints given and implement it. We might also propose a novel quantization scheme for you to evaluate (and you are very welcome to propose your own).
2. If the decision falls on a relatively simple quantization scheme, the investigations should be extended to compressing the weights and/or intermediate results [11].

3. Implement an optimized hardware accelerator using the results of the previous project on change-based evaluation of CNNs (for the feature extraction).
4. Extend the functionality to allow to switch between object detection and scene labeling/semantic segmentation.

### **Task 5 – Gather and Present Final Results (3 Weeks)**

1. Prepare presentation (20 min. + 5 min. discussion).
2. Write final report. Include all major decisions taken during the design process and argue your choice. Include everything that deviates from the very standard case -- show off everything that took time to figure out and all your ideas that have influenced the project.

## **4 Project Organization**

### **4.1 Weekly Report**

There will be a weekly report (WR) sent by the student every Friday evening. The main purpose of this report is to document the project's progress and should be used by the student as a way to communicate any problems that arise during the week. The report, along with all other relevant documents (source code, slides, papers, etc.), should be uploaded regularly to the assigned GIT repository. Another aspect of the weekly report is to reflect about what has been done and thinking about how to proceed.

### **4.2 Weekly Meetings**

Preferably on Monday afternoon, the student shall meet with the advisors in order to discuss the weekly report, along with any issues/problems that may have persisted during the previous week. These meetings are meant to provide a guaranteed timeslot for mutual exchange of information on how to proceed, clear out any questions from either side and to ensure the student's progress.

### **4.3 Final Report**

Two hard-copies of the report are to be turned in. All copies remain the property of the Integrated Systems Laboratory. A copy of the developed software, build script/project files, drawings/illustrations, acquired data, etc. needs to be handed in at the end of the project.

### **4.4 Final Presentation**

At the end of the project, the outcome of the thesis will be presented in a 20-minute talk and 5 minutes of discussion in front of interested people of the Integrated Systems Laboratory.

## 4.5 Working Place

We expect the student to perform most of the work at ETH and be present in the student room on most workdays. Personal experience has shown that people working from home have a significantly reduced progress, which is dissatisfying for the student and the advisors. This shall not prevent the student from taking a vacation or individual days off.

## References

- [1] L. Cavigelli, M. Magno, and L. Benini, "Accelerating Real-Time Embedded Scene Labeling with Convolutional Networks," in *Proc. ACM/IEEE Design Automation Conference*, 2015.
- [2] D. Gschwend, "ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network," ETH Zurich/Supercomputing Systems AG, 2016.
- [3] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *arXiv:1506.01497*, 2015.
- [4] C. Zhang, P. Zhou, J. Cong, and C. Zhang, "Caffeine : Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks Caffeine : Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," no. August, 2016.
- [5] P. Meloni, G. Deriu, F. Conti, I. Loi, L. Raffo, and L. Benini, "Curbing the Roofline: a Scalable and Flexible Architecture for CNNs on FPGA Paolo," *Proc. ACM Int. Conf. Comput. Front. - CF '16*, pp. 376–383, 2016.
- [6] L. Cavigelli and L. Benini, "Origami: A 803 GOP/s/W Convolutional Network Accelerator," *IEEE Trans. Circuits Syst. Video Technol.*, 2016.
- [7] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights," *arXiv:1606.05487*, 2016.
- [8] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," vol. 16, 2016.
- [9] B. Moons, B. De Brabandere, L. Van Gool, and M. Verhelst, "Energy-Efficient ConvNets Through Approximate Computing," 2016.
- [10] S. Han, J. Pool, J. Tran, W. J. Dally, m john Tran, and W. J. Dally, "Learning Both Weights and Connections for Efficient Neural Networks," *NIPS*, May 2015.
- [11] S. Han, H. Mao, and W. J. Dally, "Deep Compression - Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *Iclr*, pp. 1–13, 2016.

Appendix	<b>B</b>
----------	----------

## Declaration of Originality

## B. Declaration of Originality



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

### Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

**Titel der Arbeit** (in Druckschrift):

Development of an Embedded Object Detection System on the Zynq SOC Platform

**Verfasst von** (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

**Name(n):**

Degen

**Vorname(n):**

Philippe

Ich bestätige mit meiner Unterschrift:


- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

**Ort, Datum**

Zurich, 21.07.2017

**Unterschrift(en)**

  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.



# Bibliography

- [1] T. D. J. M. Ross Girshick, Jeff Donahue, “Rich feature hierarchies for accurate object detection and semantic segmentation,” 2013. [Online]. Available: <https://arxiv.org/pdf/1311.2524v5.pdf>
- [2] R. Girshick, “Fast r-cnn,” 2015. [Online]. Available: <https://arxiv.org/pdf/1504.08083.pdf>
- [3] R. G. J. S. Shaoqing Ren, Kaiming He, “Faster r-cnn: Towards real-time object detection with region proposal networks,” 2016. [Online]. Available: <https://arxiv.org/pdf/1506.01497v3.pdf>
- [4] S. R. J. S. Kaiming He, Xiangyu Zhang, “Deep residual learning for image recognition,” 2016. [Online]. Available: <https://arxiv.org/pdf/1512.03385v1.pdf>
- [5] K. Simonyan and A. Zisserman. (2014, aug) Very Deep Convolutional Networks for Large-Scale Visual Recognition. VGG 16-19. [Online]. Available: [http://www.robots.ox.ac.uk/~vgg/research/very\\_deep/](http://www.robots.ox.ac.uk/~vgg/research/very_deep/)
- [6] R. S. F. N. V. Zhaowei Cai, Quanfu Fan, “A unified multi-scale deep convolutional neural network for fast object detection,” 2016. [Online]. Available: <https://arxiv.org/pdf/1607.07155v1.pdf>
- [7] K.-H. K. Y. C. M. P. Sanghoon Hong, Byungseok Roh, “Pvanet: Lightweight deep neural networks for real-time object detection,” 2016. [Online]. Available: <https://arxiv.org/pdf/1611.08588v2.pdf>
- [8] K. H. J. S. Jifeng Dai, Yi Li, “R-fcn: Object detection via region-based fully convolutional networks,” 2016. [Online]. Available: <https://arxiv.org/pdf/1605.06409v2.pdf>
- [9] R. G. A. F. Joseph Redmon, Santosh Divvala, “You only look once: Unified, real-time object detection,” 2015. [Online]. Available: <https://arxiv.org/pdf/1506.02640v5.pdf>

## Bibliography

- [10] D. E. C. S. S. R. C.-Y. F. A. C. B. Wei Liu, Dragomir Anguelov, "Ssd: Single shot multibox detector," 2015. [Online]. Available: <https://arxiv.org/pdf/1512.02325v4.pdf>
- [11] P. H. J. K. K. Bichen Wu, Forrest Iandola, "Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving," 2016. [Online]. Available: <https://arxiv.org/pdf/1612.01051v1.pdf>
- [12] M. W. M. K. A. W. J. D. K. K. Forrest N. Iandola, Song Han, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," 2016. [Online]. Available: <https://arxiv.org/pdf/1602.07360v4.pdf>
- [13] W. J. D. Song Han, Huizi Mao, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2015. [Online]. Available: <https://arxiv.org/pdf/1510.00149v5.pdf>
- [14] L. V. G. M. V. Bert Moons, Bert De Brabandere, "Energy-efficient convnets through approximate computing," 2016. [Online]. Available: <https://arxiv.org/pdf/1603.06777v1.pdf>
- [15] P. Gysel, "Ristretto: Hardware-oriented approximation of convolutional neural networks," 2016. [Online]. Available: <https://arxiv.org/pdf/1605.06402v1.pdf>
- [16] S. G. Philipp Gysel, Mohammad Motamedi, "Hardware-oriented approximation of convolutional neural networks," 2016. [Online]. Available: <https://arxiv.org/pdf/1604.03168.pdf>
- [17] P. Warden, "How to quantize neural networks with tensorflow," 2016. [Online]. Available: <https://petewarden.com/2016/05/03/how-to-quantize-neural-networks-with-tensorflow/>
- [18] A. T. D. A. Dumitru Erhan, Christian Szegedy, "Scalable object detection using deep neural networks," 2013. [Online]. Available: <https://arxiv.org/pdf/1312.2249v1.pdf>
- [19] S. I. S. R. D. A. Dumitru Erhan, Christian Szegedy, "Scalable high quality object detection," 2014. [Online]. Available: <https://arxiv.org/pdf/1412.1441v3.pdf>
- [20] A. S. S. A. K. K. Yao Fu, Ephrem Wu and R. Wittig, "Deep learning with int8 optimization on xilinx devices," 2017. [Online]. Available: [https://www.xilinx.com/support/documentation/white\\_papers/wp486-deep-learning-int8.pdf](https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf)
- [21] S. Y. K. G. B. L. E. Z. J. Y. T. T. N. X. S. S. Y. W. H. Y. Jiantao Qiu, Jie Wang, "Going deeper with embedded fpga platform for convolutional neural network," 2016. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2993452.2995268>
- [22] L. B. L. Cavigelli, Michele Magno, "Accelerating real-time embedded scene labeling with convolutional networks," 2015. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2744769.2744788>

## *Bibliography*

- [23] K. G. J. Y. L. S. S. Y. S. H. Y. W. Junbin Wang, KeYan, “Real-time pedestrian detection and tracking on customized hardware,” 2016. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2993452.2995268>
- [24] F. E. A. G. David Gschwend, Emanuel Schmid, “Zynqnet: An fpga-accelerated embedded convolutional neural network,” 2016. [Online]. Available: [https://github.com/dgschwend/zynqnet/blob/master/zynqnet\\_report.pdf](https://github.com/dgschwend/zynqnet/blob/master/zynqnet_report.pdf)