Michael Hirschmugl, BSc.

FPGA Implementation of a Convolutional Neural Network for Radar Interference Mitigation

Optimizing Automotive FMCW Radar Measurements with Machine Learning on Embedded Hardware

MASTER'S THESIS

submitted to

Graz University of Technology

Supervisors

Univ.-Prof. Dipl.-Ing. Dr.mont. Franz Pernkopf (TU Graz)
Dipl.-Ing. BSc. Johanna Rock (TU Graz)
Dr. Paul Meissner (Infineon Technologies Austria AG)

Signal Processing and Speech Communication Laboratory

in cooperation with Infineon Technologies AG

Graz, November, 2021

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

date	(signature)

Abstract

This thesis proposes a highly optimized convolutional neural network implementation on embedded hardware for radar interference mitigation as a proof-of-concept application in an automotive driver assistance system. The embedded platform is a *Xilinx Zynq-7000* system-on-chip, and for resource and performance comparison, the implementation is done in the programmable logic of the device and also as firmware for the integrated processing core.

Since interference mitigation based on a neural network, in general, is very expensive in terms of computational efforts and power consumption, different optimization and quantization techniques are investigated and compared. In order to keep implementation and computation as efficient as possible, all floating-point operations are converted to fixed-point and evaluated against the floating-point inference results.

In conclusion, all implementations and optimizations are compared in terms of resource utilization in order to be able to select an embedded platform with sufficient resources for a real-world application with specific performance requirements.

Contents

1 Introduction

1.1 Preface

Autonomous driving, and Advanced Driver-Assistance Systems (ADAS) in general, require fast, reliable, and precise perception of a vehicles' surroundings. Besides camera vision and ultrasound, radars are a well-established technology in the automotive sector. The Frequency-Modulated Continuous-Wave (FMCW)/Chirp Sequence (CS) radar enables accurate measurement of distance (range) and relative velocity of multiple surrounding objects.

In order to extract range and velocity features from radar measurements, range-Doppler processing is performed on the measurement data, which yields a Range-Doppler map (RD map) and enables the usage of optional, subsequent object detection and identification algorithms [1].

An increasing problem for radars in ADAS are mutual interferences from one car's radar sensors with another [1]. Interference prevention approaches are not reliable solutions, since secure communication between all vehicles involved must be assured, traditional approaches for interference mitigation suffer from other difficulties: Methods in the time domain, such as zeroing [2] or subtraction of interfered samples [3], all presume correct interference detection in a measured signal.

A novel approach is the usage of Neural Networks (NNs) in order to mitigate interference, and especially Convolutional Neural Networks (CNNs) benefit greatly from the 2D spectogram representation of input RD maps. In this approach, a 2D-Fast Fourier Transform (FFT) is performed on radar measurements for preprocessing and the CNN is applied to the RD maps in frequency domain. For simulated interference, the CNN offers better overall performance than all traditional methods, while even outperforming interference-free (clean) measurements with subsequent object detection in some cases [4]. An issue with deploying CNNs in the context of real-time systems is their computational complexity, which can require millions of Multiply-And-Accumulate operations (MACs) that need to be evaluated within tight time constraints. Typical measurement cycle durations for commercial radar systems in ADAS range from 40 ms to 80 ms, which already comprises of the actual measurement and all post-processing, and consequently does not leave much room for millions of MACs [5]. Timing limitations and high computational complexity are typically accompanied by increased power consumption and thermal restrictions.

This is where Field-Programmable Gate Arrays (FPGAs) with their inherent parallelism appear to be exceptionally advantageous. Even though modern Graphics Processing Units (GPUs) effortlessly reach the mentioned performance requirements, ADAS usually exhibit tight limitations regarding power consumption and space, and FPGAs exceed conventional GPUs in both measures [6]. Taking economic factors into account too, FPGAs are available in a wide range of resource configurations, speed-grades, optional features, and therefore cost, and provide the potential of choosing the most cost effective platform in order to reach a specific solution.

1.2 Goals and Motivation

The goal of this thesis is proof-of-concept implementation of a CNN-based interference mitigation system on embedded hardware, specifically a FPGA-based System-on-Chip (SoC). A schematic illustration of radar measurement and interference mitigation can be seen in Figure 1.1. The interference mitigation is expected to work as real-time processing system; it is applied to real-world measurements as obtained and preprocessed directly through the radar hardware [7]. Processed output data must be displayed as a continuous stream of images on a PC screen.

For radar data measurement, a 77 GHz FMCW radar demonstrator board is used. In order to process these measurements, a data link between the radar device and the processing system must be established.

The basis for the CNN has already been developed, trained, and evaluated, including a quantization-aware training approach. Training of the network is done offline, and the embedded system is used exclusively for the forward-path (inference) of the CNN [7]. Furthermore, range-Doppler processing must be performed on the measurements, ahead of the CNN inference.

The primary motivation for this work is to examine resource-efficient embedded implementations of CNN-based radar interference mitigation for real-world automotive scenarios, regarding resource utilization for various quantizations of a given CNN. A secondary objective is the examination of the correlation between achievable computational latency and offered SoC parameters for a given hardware platform (such as clock speeds and FPGA-area). Further details on the evaluations and comparisons can be found in Section 1.4.

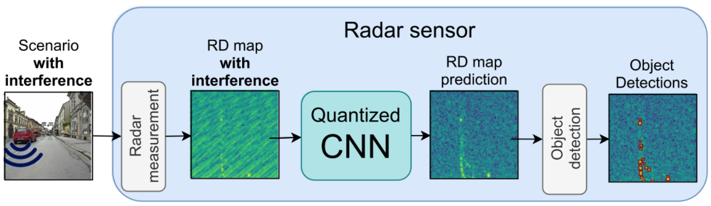


Figure 1.1: Schematic diagram of CNN-based interference mitigation and object detection. (Source: [7], "Interference mitigation of radar signals using a quantized CNN to remove interference patterns, retain object signals, and provide a high detection sensitivity.")

1.3 System Overview

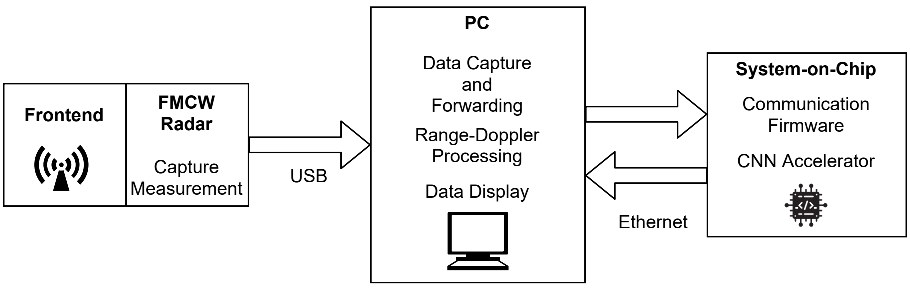


Figure 1.2: System Overview.

The block diagram in Figure 1.2 shows an overview of the resulting system. The radar device on the left is composed of a multi-channel 77GHz-frontend and the *RadarLog* radar data capturing environment, both by *INRAS* [8]. The radar demonstrator is connected to a PC via USB. This is required, because the USB communication protocol of the capture device is not open source and can only be accessed by a proprietary USB library provided by *INRAS*[1]. The PC receives the radar measurements, performs the range-Doppler processing[2] and transmits the data via ethernet to the CNN-accelerator on the SoC (which can be seen in the diagram on the right). Afterward, the interference-free data is transmitted back to the PC, where performance can be evaluated, and data can be visualized in custom Python tools.

In this thesis, three different CNN implementations for interference mitigation with increasing hardware support were realized, evaluated and compared:

- **Software Implementation:** Chapter 3 describes the basis CNN network, its Python implementation, the training procedure, full quantization including input and output layers, and the required modifications in order to deploy the network to an embedded platform.
- **Firmware Implementation:** This firmware CNN implementation is written in C and executed by the Central Processing Unit (CPU) of the SoC (= Processing System (PS)). The respective documentation is in Section 4.2.
- **Hardware Implementation:** Section 4.3 explains how the CNN was written as a hardware accelerator in a Hardware Description Language (HDL)[3] specifically for the FPGA area of the SoC (= Programmable Logic (PL)). Furthermore, it is shown how the accelerator works together with the PS and how parallelization of numerous MACs is accomplished.

Finally, Chapter 5 illustrates the custom Python tools for evaluation and visualization, and concludes with performance, timing, resource utilization, and power consumption evaluations of the individual implementations.

1.4 Contribution

This thesis builds on previous work, by proceeding present research in CNN-based radar interference mitigation. While CNN-based algorithms yield impressive results, the problems of high memory requirements and computational complexity were just recently investigated [4]. Quantization techniques for CNN-based interference mitigation were examined by analyzing quantization of weights and activations of different CNN-based model architectures in order to find a small model with decent resource requirements, while retaining high interference mitigation performance [7]. It was established that 8-bit quantization of all weights and activations yields a memory footprint reduction of approximately 75 % compared to a real-valued model, without significant performance degradation [7].

The contributions are:

Deployment on Embedded System: A previously developed CNN-model [7] is the basis for this thesis and illustrated in-depth in Chapter 3. This work aims at **deploying this model on actual hardware including real-time radar measurement** and documents the implementation process.

Additional Quantizations: Furthermore, **additional quantizations are investigated by varying the bit-width of inputs and inference outputs**, but without re-training of weights for reduced quantization depth. Since the network is trained for 32-bit floating-point input data, changing the quantization size of the inputs and outputs offers even more insights into the robustness and possible application constraints.

[1] In a real-world scenario, and using another radar device, the PC would not be needed, and all processing could potentially be done on the FPGA SoC. Optimally by a direct data link between radar and SoC.
[2] The implementation of the range-Doppler processing algorithm is not part of this work.
[3] In this thesis the Very high speed integrated circuit Hardware Description Language (VHDL) was used.

Fully Fixed-Point Implementation: Even though the network was trained for quantized weights, one floating-point multiplication is needed on all activations. This is because real-valued weights were mapped to integer values during training and need to be scaled correctly. On an embedded system, such a floating-point operation would dramatically decrease performance, and so **this thesis aims at creating an inference accelerator that works entirely with fixed-point operations**.

Implementation in Firmware and FPGA-Hardware: Today, SoCs combining PL and PS units are available. The CPU core in these systems is designed to provide fast and power efficient operations. Using only one such CPU for inference (= Firmware Implementation) is compared to the FPGA-based hardware accelerator with a high rate of parallel computations and thus potentially much higher throughput (= Hardware Implementation). Thereby, a summary of hardware requirements and related inference timing performance limits is established.

List of quantization and implementation combinations examined in this thesis:

- **Firmware Implementation:**
  - 32-bit floating-point inputs and inference outputs.
  - 32-bit fixed-point inputs and inference outputs.
  - Additionally, with, and without compiler optimization and usage of an auxiliary floating-point co-processor.
- **Hardware Implementation:**
  - 32-bit fixed-point inputs and inference outputs.
  - 16-bit fixed-point inputs and inference outputs.
  - 8-bit fixed-point inputs and 16-bit fixed-point inference outputs.[4]

Investigation of Power Consumption and Resource Utilization: For all these combinations, timing, power consumption, and hardware resource utilization is examined and illustrated in Chapter 5.

[4] 8-bit outputs yield considerably worse predictive performance and are thus not compared in this thesis. Presumably this results from an output dynamic that was not optimized for this range of values.

2 Background Knowledge

This chapter provides basic knowledge about the three key concepts relevant to this work, beginning with radar measurement in ADAS (Section 2.1). Section 2.2 quickly familiarizes the reader with well-established terminology and concepts used in CNN applications, and Section 2.3 will show why FPGA-based SoCs might be the optimal embedded platform in this usecase.

2.1 Automotive Radar Overview

Besides radar, there are different perception systems such as cameras, Light Detection And Ranging (LIDAR) and ultrasonics used in a typical ADAS (Figure 2.1). All these sensors have strengths and weaknesses as well reviewed in literature [1], [9]. For example, cameras offer excellent color perception and are thus beneficial to object classification but do not perform well at night or in adverse weather. LIDARs, on the other hand, have good angular resolution and range but are limited in Field of View (FoV) and again suffer in adverse weather. Generally, radars show "all-weather" capabilities and are the best sensors for range and radial velocity measurement[5] [1].

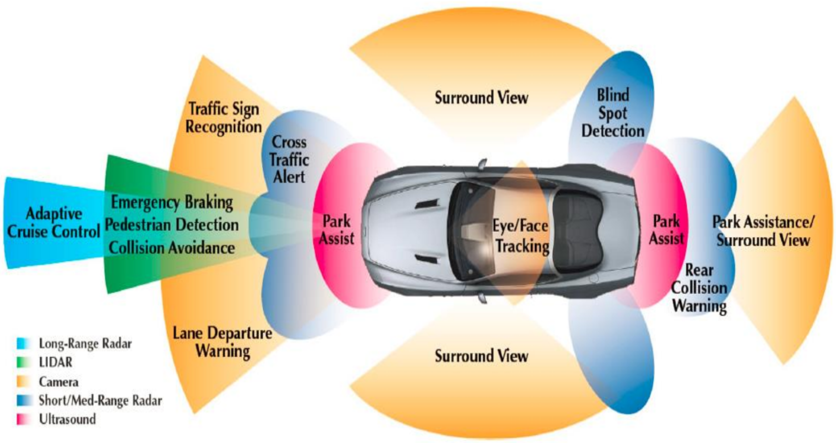


Figure 2.1: ADAS. (Source: [10], "Advanced Driver Assistance System")

Radar sensors can be broadly categorized in the two mainly used frequency bands, 24 GHz and 77 GHz, and the three configurations Short-Range Radar (SRR) with a maximum distance of about 30 m, Mid-Range Radar (MRR) with about 60 m, and Long-Range Radar (LRR) with about 250 m [11].

Applications for radar sensors in the automotive field include automatic cruise control, blind-spot detection, collision warning, as well as automatic emergency braking, which are vital features in most ADAS and self-driving cars. This leads to an increased number of installed radar sensors, which is followed by a much greater possibility of mutual interference [12], [13]. In the

[5] Radar sensors are weak in classification tasks and angular resolution [1].

timeframe from 2021 to 2023, up to six radar systems might be in use per car, which could dramatically decrease performance in these systems [1].

A prominent example of one such automotive radar sensor is the third generation of *Bosch*[6] LRR [5] (Figure 2.2). This device works in a frequency range of 76 to 77 GHz and can measure distances of 0.5 to 250 m at an accuracy of ±0.1 m. It can also measure relative speeds of −75 to 60 m/s at an accuracy of ±0.12 m/s. The sensor is a monostatic[7] FMCW radar with four fixed beams and uses silicon based technology by *Infineon*[8] for the Radio Frequency (RF) components [5] [14]. It is capable of detecting up to 32 surrounding objects [5].

Figure 2.2: Bosch LRR. (Source: Bosch)

2.1.1 FMCW Radar Basics

All radar technology is based on the principle that transmitted electromagnetic waves are reflected by objects in their path. The received reflections are measured and evaluated regarding time delay, phase, and frequency content. Radar devices differ in the generated signals, antenna technology, and construction. In the simplest form, a single antenna transmits a single wavelength signal, with the same antenna receiving the reflection. In the automotive area, the most common radar technology is the FMCW/CS radar (Figure 2.3). In this configuration, a linear frequency modulation (”chirp“) is applied to a high-frequency base signal and transmitted in a repeating sequence (”chirp sequence“). A receiving antenna then captures the reflected signals, and both the transmitted and received signals are fed into a mixer. This mixer builds the product of both signals, which is the Intermediate Frequency (IF) signal[9].

Modern FMCW radar hardware typically consists of five functional blocks [15]: Antenna section, RF section, Analog-to-Digital Converter (ADC) interface, signal processing section and power supply.

Microstrip, or patch antenna arrays, are commonly used in *mm*Wave[10] radars. This type of antenna is reasonably simple to produce on a Printed Circuit Board (PCB) and thus features a shallow profile, which is essential in mobile applications. An improved form of the patch array antenna is the series-fed tapered array design. In this configuration, the patch sizes are tapered in order to achieve better Voltage Standing Wave Ratio (VSWR) performance and greater sidelobe repression [16].

6 https://www.bosch.at/
7 Monostatic: A single device is serving as both transmitter and receiver.
8 https://www.infineon.com/
9 The IF signal can also be called the beat signal
10 Radar that uses frequencies in the *mm*-range, such as a 77 GHz-radar.

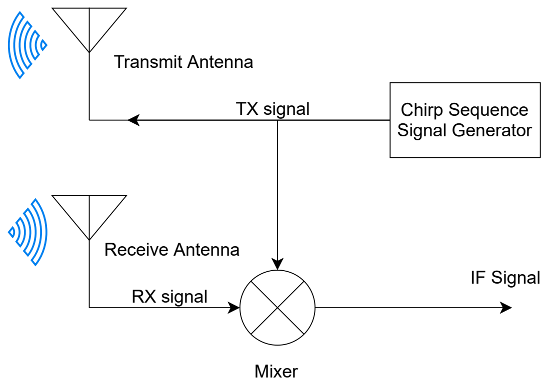


Figure 2.3: FMCW radar block diagram.

It is common practice to locate the whole RF section on its own PCB ("frontend“). The frontend contains the microstrip antenna array and RF processing components, such as a radar transceiver, Local Oscillator (LO), power amplifier, Low-Noise Amplifier (LNA) and the mixer. The LO is what generates the linear frequency-modulated continuous wave signal (chirp), which is then amplified by the power amplifier in order to drive the transmission antenna(s). An LNA amplifies the captured signals from the receive antenna(s). The mixer section multiplies both the transmitted and received signal, and the output IF signal is digitized by an ADC interface [15].

Integrated Circuits (ICs) for RF designs differ mainly in terms of the amount of integrated components. For instance, the *Infineon* BGT24MTR12 Silicon Germanium 24 GHz Transceiver Monolithic Microwave Integrated Circuit (MMIC) already combines all RF components and delivers IF output signals to subsequent sampling and signal processing [17]. Some transceivers might even include the ADC section, but especially in the 77 GHz area, it is more common to use separate ICs for signal synthesis, amplification and digitization [18].

In mathematical terms, the IF output of a FMCW radar is the product of a transmitted signal $u_t(t) = A_t \cdot cos(\omega_t t + \phi_t)$ and a received signal $u_r(t) = A_r \cdot cos(\omega_r t + \phi_r)$, which results in

$$u_{t,r} = \frac{1}{2}A_t A_r \Big[cos\Big((\omega_r - \omega_t)t + \phi_t - \phi_r\Big) + cos\Big((\omega_r + \omega_t)t + \phi_t + \phi_r\Big)\Big]. \tag{2.1}$$

A filter removes the higher frequency part (second *cos*-term), which leaves the IF signal as

$$u_{if} = \frac{1}{2}A_t A_r \cdot cos\Big((\omega_r - \omega_t)t + \phi_t - \phi_r\Big) = A_{if} \cdot cos\Big(\omega_{if} t + \phi_{if}\Big). \tag{2.2}$$

Equation (2.2) contains all information regarding range and relative radial velocity of reflecting objects in the argument of the *cos*, as described in the next section. After filtering, the mixer output can also be seen as a downconversion of the modulated RF carrier signal, since the IF signal only features frequencies corresponding to the difference of the transmitted frequency and the Doppler-shifted and time-delayed reflection signal [19]. This means that adjacent ADC components can operate at much lower frequencies than 77 GHz.

2.1.2 Range-Doppler Imaging

Range-Doppler imaging is an operation that produces a 2D-map, which represents both the range and the relative velocity of a reflecting object on the x- and y-axes (Figure 2.4).

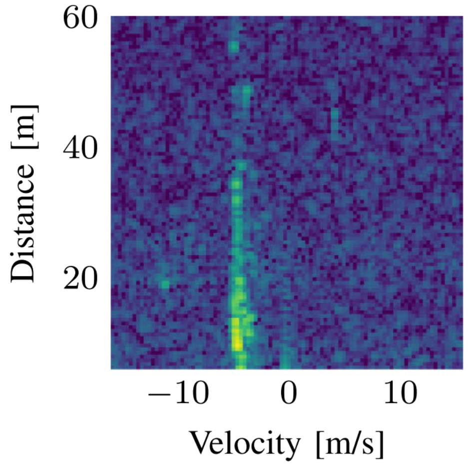


Figure 2.4: Range-Doppler map example. (Source: [7], "RD magnitude spectra in decibel [dB] without interference")

The range information is extracted from a single time-delayed chirp reflection as a frequency peak in the IF signal. Because of the Doppler-shift, relative velocity shows in a frequency and phase shift over multiple chirps.

Range Estimation: In the first step of range-Doppler-processing, the individual chirp signals are processed by a FFT, which results in frequency peaks at object ranges[11], as can be seen in Figure 2.5. An object at a constant distance will result in a constant frequency spectrum over multiple chirps [20].

Velocity Estimation: Measurement of relative velocity is based on evaluation of phase differences over multiple chirps because the phase of the IF signal is sensitive to small changes in the object distance. The phase may even distinguish between objects at equal distance from the transmitter, if the velocities differ. In order to extract the phase, a second FFT is performed over the rows of the range information matrix (Figure 2.5) [20].

[11] Also termed "range profiles".

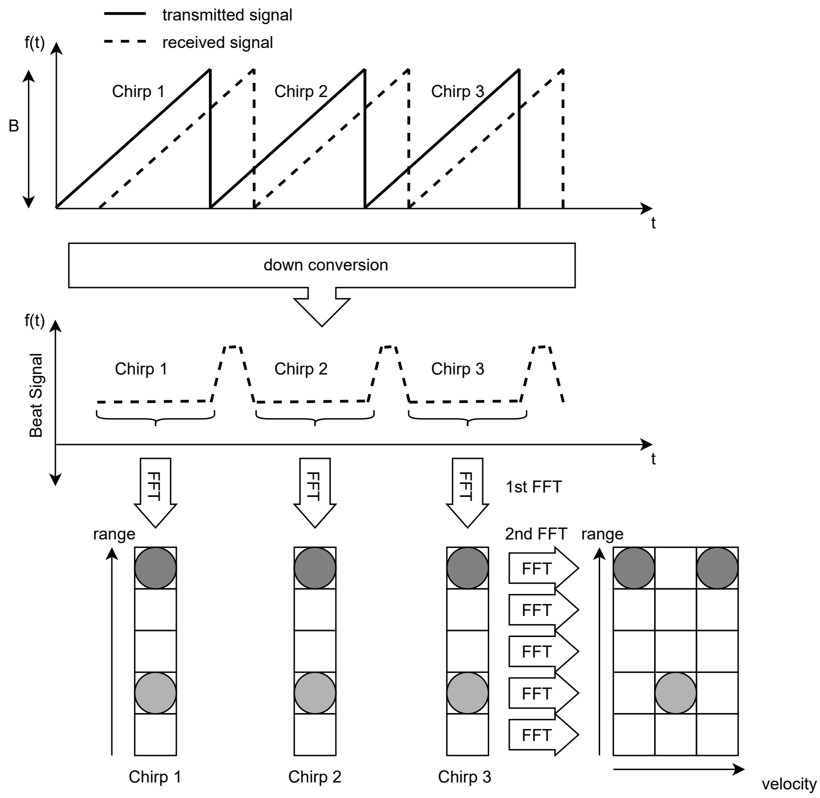
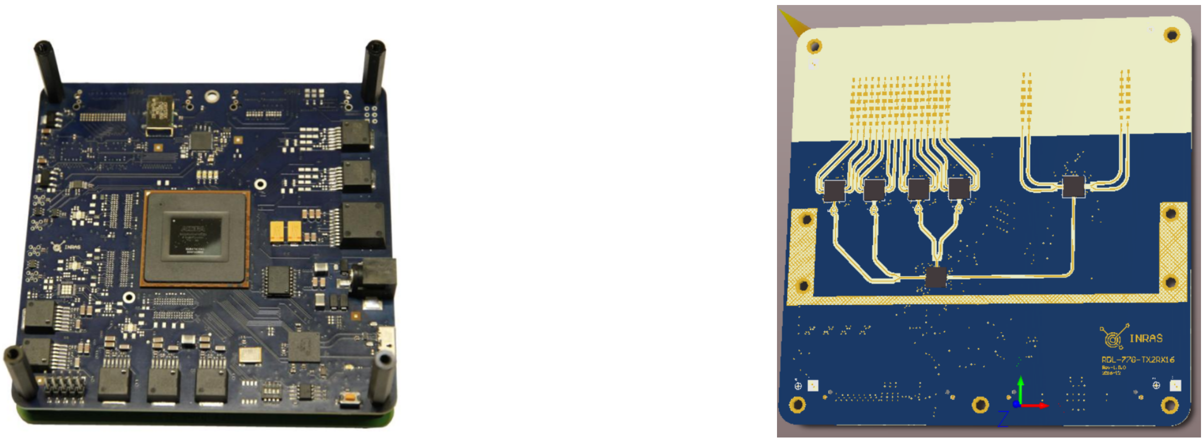


Figure 2.5: Range-Doppler processing.

2.1.3 *INRAS RadarLog* and Frontend

In this thesis, a radar device by the company *INRAS* is used. This device consists of a backend board called "RadarLog" (Figure 2.6(a)), and the RDL-77G-TX2RX16 77 GHz frontend with two transmit, and 16 receive antennas (Figure 2.6(b)).



(a) *INRAS RadarLog* (b) *INRAS* Radar Frontend

Figure 2.6: INRAS RadarLog and Frontend. (Source: INRAS)

Figure 2.7 is a block diagram of the frontend PCB. The design features multiple receive and transmit antennas and enables Multiple-In-Multiple-Out (MIMO) processing of FMCW signals. To generate the FMCW transmit signals, a RTN7735 77 GHz radar MMIC works in conjunction with a RCC1010 "radar companion IC"[12], both from *Infineon*. A RPN7720 dual power amplifier[13] drives the transmission antennas. It is possible to arbitrarily activate each transmission antenna in order to implement a virtual array for improved angular resolution. Four RRN7745 receivers complete the receiver frontend including sixteen serial-fed tapered patch array antennas with eight elements each[14] [21].

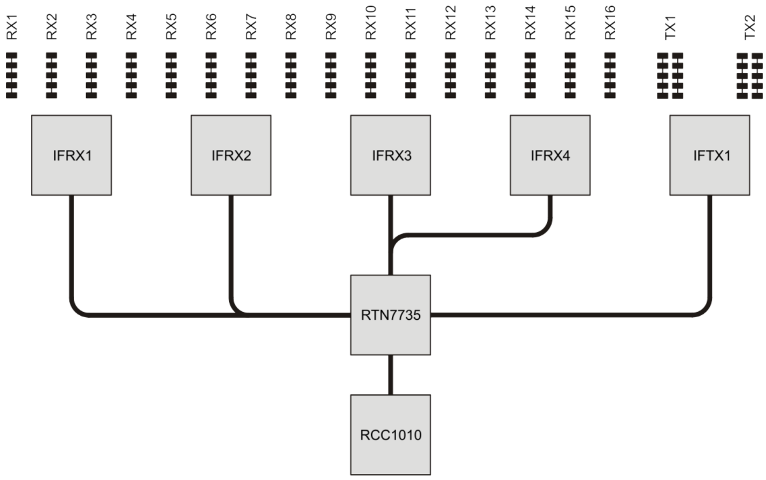


Figure 2.7: INRAS radar hardware overview. (Source: [21], "Antenna configuration and naming convention for MIMO frontend")

The *RadarLog* backend (Figure 2.8) is mainly used for sampling and communication with a PC. It is designed around an Arria V[15] FPGA from *Intel*[16] and two AFE5801 analog frontends[17] from *Texas Instruments*[18] for high-speed sampling of IF signals. The RF connector is the interface to variable radar frontends by *INRAS*, and connected to the RDL-77G-TX2RX16 frontend in this case. Double Data Rate (DDR) memory is used for buffering of samples and USB 3.0 (with USB 3.1 connector) for up to 3 Gbit/s data transfer to a connected PC [23]. It is possible to use a Matlab or Python library for data collection and processing.

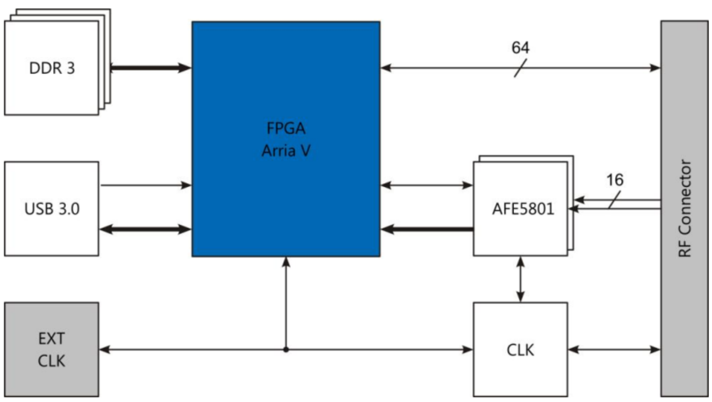


Figure 2.8: INRAS radar Radarlog (Block diagram). (Source: [23], "Radarlog (Block diagram)")

[12] https://octopart.com/rcc1010-infineon-34482848Specs
[13] https://octopart.com/rpn7720pl-infineon-75656071CadModels
[14] The exact funnctionalities of the RTN7735, RCC1010, RPN7720 and the RRN7745 ICs are not publicly disclosed.
[15] https://www.intel.com/content/www/us/en/products/details/fpga/arria/v.html
[16] https://www.intel.com/content/www/us/en/products/details/fpga.html
[17] AFE5801: 8-Channel Variable-Gain Amplifier (VGA) With Octal High-Speed ADC. Datasheet: [22]
[18] https://www.ti.com/

2.2 Convolutional Neural Networks

CNNs are a specific class of Artificial Neural Networks (ANNs) that exploit spatial information in network inputs and are typically used for image processing. This sections facilitates a short introduction of ANNs and CNNs, including common building blocks and optimization strategies.

2.2.1 Introduction to Artificial Neural Networks

Figure 2.9 illustrates a single artificial neuron, which are the basic building blocks of ANNs. In mathematical terms, the neuron performs the summation of several weighted inputs and a bias term, and passes the result through a nonlinear activation function[19] $f(\cdot)$. This yields an expression $y(\boldsymbol{x}_n, \boldsymbol{w}) = f(\boldsymbol{w}^T \cdot \boldsymbol{x}_n)$ as neuron output[20] [25].

If multiple neurons are stacked and cascaded in a layered fashion, the result is an ANN (Figure 2.10). This network multiplies an input vector $\boldsymbol{x}_n$ by a layer-specific weight matrix $\boldsymbol{w}^{(l)}$ and passes the results to the correlating neurons. Neuron outputs then act as inputs for the consecutive layer.

The example network in Figure 2.10 shows an ANN with two layers ($L = 2$): the hidden layer and the output layer. In this example, all neurons in one layer are connected to all other neurons in a successive layer. This architecture is also called a *fully-connected two-layer network*. The outputs y_k are given by

$$\begin{aligned} y_k(\boldsymbol{x}_n, \boldsymbol{w}) &= g\left(\sum_{m=0}^{M} w_{km}^{(2)} \cdot f\left(\sum_{d=0}^{D} w_{md}^{(1)} x_n^{(d)} \right) \right) \\ &= g\left(\sum_{m=0}^{M} w_{km}^{(2)} \cdot \phi_m(\boldsymbol{x}_n) \right), \end{aligned} \tag{2.3}$$

with f being the activation function of the hidden layer and g of the output layer. Since the functions $\phi_m(\boldsymbol{x}_n)$ are also dependent on the weights $w_{md}^{(1)}$, parametric basis functions[21] ϕ_m were attained in this minimal ANN configuration [25]. This means, that the basis functions of this network can be tuned by optimization of the weights $\boldsymbol{w}$. During a training phase, the network is able to adapt its weights and learn the required basis functions.

Deep Neural Networks (DNNs) typically use multiple hidden layers. In theory, a DNN is able to approximate any continuous function arbitrarily well[22] by optimization of the parameters $\boldsymbol{w}$ [25].

[19] Activation Function: Non-Linear transformation of an input value. In simple terms, the activation function "decides", whether a neuron is activated or not, and/or the amount thereof [24]. A higher output from an activation function can be interpreted as the presence of specific features.

[20] This expression is equivalent to logistic regression [25]. Logistic regression is a classification algorithm [26].

[21] Each parametric basis function being an assembly of a nonlinear function and a linear combination of inputs [27].

[22] On a compact domain (closed and bounded) $y(\boldsymbol{x}, \boldsymbol{w}) : \mathbb{R}^D \to \mathbb{R}$, given appropriate activation functions and enough data and hidden neurons [25].

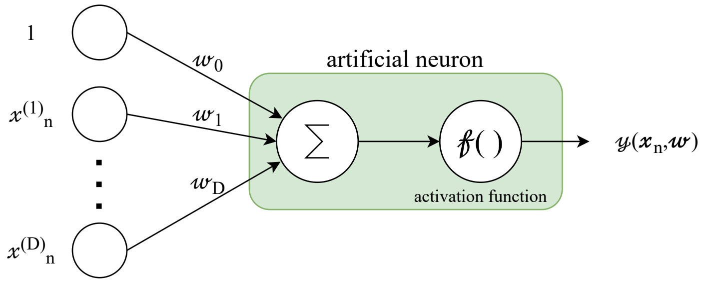


Figure 2.9: Single artificial neuron.

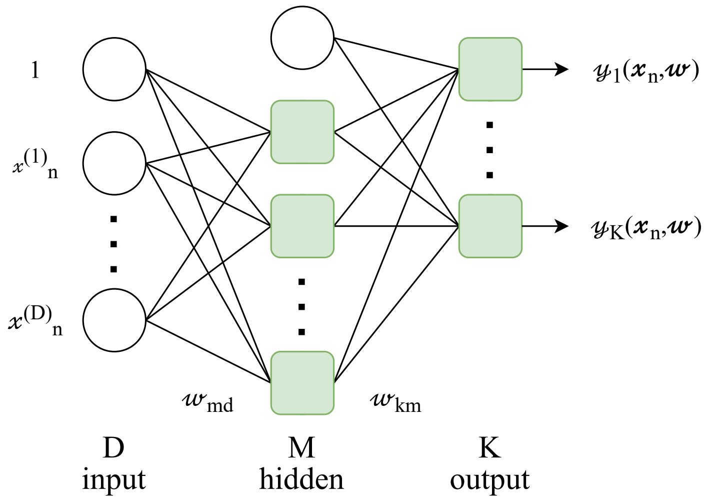


Figure 2.10: A simple artificial neural network.

2.2.2 CNN Basics

CNNs are a specialized type of neural networks that use convolutions instead of general matrix multiplications in at least one of their layers [28]. In a CNN, each layer is represented in two dimensions, in contrast to a traditional neural network layer, where all input data is stretched onto a one-dimensional vector. By replacing fully connected layers with convolutional layers, it is possible to exploit spatial information in image-like data[23] without the need of a myriad of weights and neurons. A convolution kernel can be seen as a 2D filter kernel that is shifted across input feature maps[24]. The weights of the kernel do not depend on the spatial location in the image. A single convolutional layer typically consists of an $C_i \times C_o$ array of convolution kernels, each of size $K \times K$ (usually, K being an odd integer in the range from 1 to 11) [29]. Figure 2.11 illustrates the typical nomenclature of CNNs.

- $\mathbf{X}_n$: input tensor. $shape(\mathbf{X}_n) = (C_i, W, H)$
  - c_i: input feature map (channel) index (C_i: amount of input feature maps (channels) in input tensor)
  - w: horizontal image index (W: image width)
  - h: vertical image index (H: image height)
  - n: input sample index (N: number of samples)
- $\mathbf{Y}_n$: output tensor. $shape(\mathbf{Y}_n) = (C_o, W, H)$

[23] Data, represented in at least one 2D array, comparable to (possibly) multi-dimensional image data from a visual sensor.
[24] Input Feature Map: Two-Dimensional array of input data.

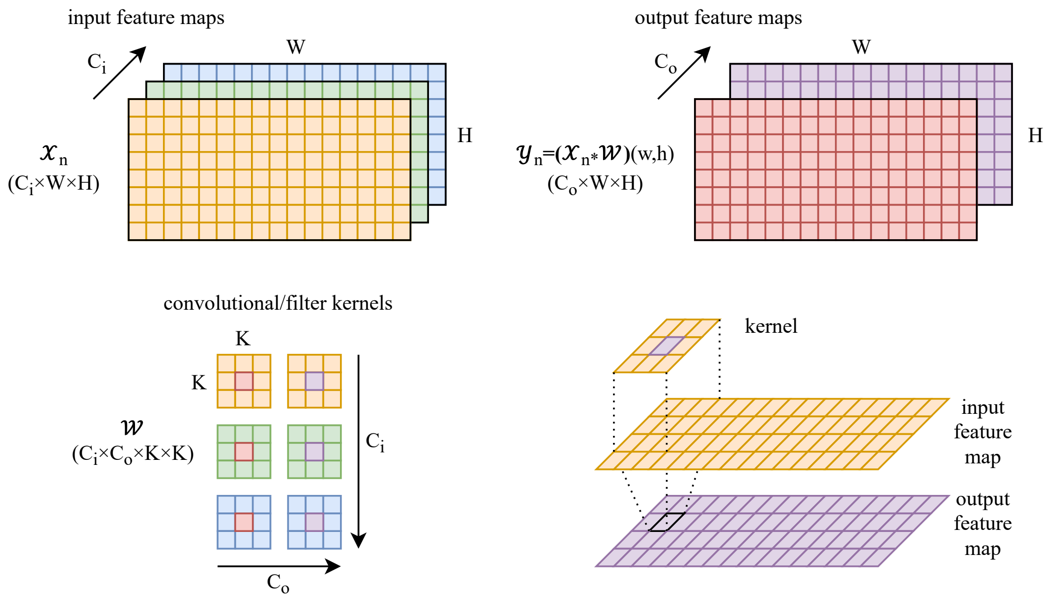


Figure 2.11: Illustration of CNN nomenclature. Top row shows input feature map $\boldsymbol{X}_n$ and the output feature map $\boldsymbol{Y}_n$, which consists of the convolutions of inputs and the shifted filter matrix $\boldsymbol{W}$. Spatial dimensions of the convolution are given by the indices h and v respectively. The number of maps (channels) is either C_i for the inputs, or C_o for the outputs. Bottom left represents the filter matrix $\boldsymbol{W}$ and bottom right the computation of a single point on a single output feature map.

  - c_o: output feature map (channel) index (C_o: amount of output feature maps (channels) in output tensor)

- $\mathbf{W}$: kernel filter weights matrix. $shape(\mathbf{W}) = (C_i, C_o, K, K)$
  - i: horizontal kernel weight map index (K: kernel width)
  - j: vertical kernel weight map index (K: kernel height)

The computation of a single point on the output tensor yields a vector $\mathbf{y}_n(w, h)$, i.e.

$$\begin{aligned} y_n(c_o, w, h) &= (\boldsymbol{X}_n * \boldsymbol{W})(w, h) \\ &= \sum_{c_i}^{C_i} \sum_{j}^{K} \sum_{i}^{K} x_n(c_i, w + i - 1, h + j - 1) \cdot w(c_i, c_o, i, j), \end{aligned} \tag{2.4}$$

where

$$x_n(c_i, w, h) = \begin{cases} x_n(c_i, h, v), & \textit{if } 0 \le c_i < C_i, 0 \le w < W, 0 \le h < H \\ 0, & \textit{otherwise}, \end{cases} \tag{2.5}$$

and

$$w(c_i, c_o, i, j) = \begin{cases} w(c_i, c_o, i, j), & \textit{if } 0 \le c_i < C_i, 0 \le c_o < C_o, 0 \le i < K, 0 \le j < K \\ 0, & \textit{otherwise}. \end{cases} \tag{2.6}$$

The complete output is collected in the tensor $\mathbf{Y}_n$. It is common to extend the borders of the input maps, in order to keep the same spatial dimensions on all layers[25]. Extending the input map with zeros is called *zero padding*, while other versions, such as *reflection padding*, are possible too [30].

2.2.3 CNN Building Blocks

Activation Function: Activation functions used in CNNs are mainly either *Rectified Linear Unit (ReLU)* ($f(x) = max(0, x)$), the *identity function* ($f(x) = x$), or different *sigmoid* functions. While *sigmoid* was used in earlier implementations, *ReLU* is by far the most commonly used activation function in modern networks. *ReLU* clips all negative values to zero and offers a cheap gradient in computational sense for positive values, while it is important to note that *ReLU* is not differentiable at 0.

Layer, Feature Map, Channel, Activation, and Filter Kernels: The hidden layers in a CNN are typically convolutional layers followed by activation layers. The output of a convolutional layer is called *feature map* and the outputs of an activation layer are *activations* or *activation maps*. *Channels* refer to the depth of an image, which must match the depth of the filter kernel. Typically a convolutional layer has more than one filter, and the result is one feature map per filter as output. The filter can also be called a *feature detector*. By applying multiple filters, a convolutional layer can detect and extract a variety of features. Furthermore, since the convolutional kernel is shifted across the whole image, one filter can detect features independant of the location withing an input (in contrast to a regular NN).

Batch Normalization: Batch normalization re-centers and re-scales the input distribution of activation functions, whereas the distribution parameters are learned during training. Normalized distributions lead to more robust and faster training [31]. In a mathematical sense, a batch normalization layer performs one MAC operation on all its input values.

See [30] for more information on additional building blocks, such as *pooling*, *dropout* or *softmax* layers, which are not used throughout this thesis.

2.2.4 CNN Representation, Training and Compression

Tensor Representation: Each input frame[26] can be represented as a 3D tensor of dimensions $(H \times W \times C_i)$. Weights of a convolutional layer can be seen as a 4D tensor of shape $(H \times W \times C_i \times C_o)$. The tensor representation is typically used in frameworks such as *TensorFlow*[27].

Training, Forward Pass, and Backpropagation: Passing inputs through the complete CNN and computing all outputs is called the forward path/pass. Before training, weights are initialized with small random values[28]. Optimal weights are not chosen, though; the network can learn optimum weights, for instance, with supervised learning, which requires a set of training data for the network to learn from. This training data needs to be labeled and must offer ground truth labels. The learning phase of the neural network is a vital part of every neural network. In this phase, a loss function measures the difference of the output of the CNN to the

[25] Also called *same padding*. There exist other padding algorithms that differ in the amount of padding. *Valid padding* (no padding) does not maintain input size. In general, padding is used to handle image borders.
[26] Subsequent radar measurements after range-Doppler processing depict a stream of complex-valued images. Thus, a single RD map is refered to as a "frame".
[27] https://www.tensorflow.org/
[28] There are a couple of different strategies for weight initialization, but drawing random weights from a specified range is the most common method.

expected results. The learning phase aims to minimize this error by optimizing the filter weight values. Now in the backward path, every neuron's gradient is calculated from the expected result deviation, and then the weights are adjusted in the opposite direction of the gradient. The *learning rate* determines the amount of adjustment of the weights.

Compression of Neural Networks: A great drawback of CNNs is the high memory requirement (while still lower than for other DNNs). In order to reduce these memory demands, it is feasible to either reduce the number of necessary weights, or the bit width of a single weight value. Limited numerical precision is a cheap compression method for CNNs. For particular use cases, it is even possible to reduce the bit width of weight values to 1, meaning that a number can only be one of two distinct values[29]. Moreover, while floating-point values show the highest computational accuracy (in most cases), converting such values to a fixed-point representation might also reduce memory requirements and computational complexity.

2.2.5 Quantization-Aware Training

Most commonly, training of a NN is performed using gradient-based algorithms. After evaluation of a loss-function, gradients are computed during backpropagation in many iterations. In order to gain quantized weights, a quantization function may be used to map weights to integer values. Quantization of activations requires a piecewise constant activation function. Quantizers and piecewise constant functions are not differentiable, and this lack of a gradient makes the usage of gradient-based algorithms impossible [32].

The Straight-Through Gradient Estimator (STE) can solve this issue simply by circumventing the zero gradients and instead approximating gradient computations using functions with similar shapes [32]. Figure 2.12 illustrates a computation graph for a simplified layer with a sign activation function. During the forward pass, the non-differentiable functions are applied, but avoided in the backward pass by an approximation of the gradient.

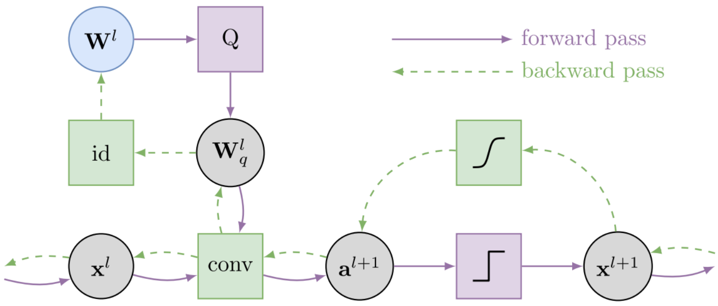


Figure 2.12: CNN forward and backward pass. (Source: [7], "Computation of forward (purple) and backward (green) pass through a simplified DNN building block using the STE")

2.3 Field Programmable Gate Arrays

This section gives a short introduction to FPGA and SoC hardware, and development. An understanding of how resources are distributed on FPGA-based SoC platforms and how they may be utilized in development is of great importance in multiple steps of the accelerator design process. Right from the start, when a suitable embedded platform has to be chosen, in

[29] Also termed "binary quantization"

the development process itself, and finally when an implementation is evaluated from reports generated from designated design tools.

It should be noted that in this thesis, a *Xilinx* SoC embedded platform was chosen. The following sections will intentionally be focused on *Xilinx*-specific hardware and nomenclature[30].

2.3.1 Introduction to Field Programmable Gate Arrays

An FPGA is a programmable IC, that consists of a matrix of freely Configurable Logic Blocks (CLBs). The developer uses these logic blocks in order to implement complex algorithms, basic arithmetic calculations, boolean logic or even memory [33]. In stark contrast to more general-purpose processors such as CPUs, Microcontroller Units (MCUs) or GPUs, a FPGA does not execute a set of instructions, while it is possible to implement multiple general-purpose processing cores on a single FPGA fabric in order to enable execution of custom instructions [34]. CLBs consist of the following parts:

- Look-Up Tables (LUTs): These elements implement logic behavior. A single LUT can be used as boolean logic compute engine or a data storage element. This is because, in hardware, a LUT can be seen as a collection of memory cells connected to a combination of multiplexer elements. A LUT is a truth table that combines different inputs in order to implement custom output functions [35].
- Flip-Flops (FFs): LUT outputs are buffered in FF register elements. A FF input is only latched and passed on to the output, if the clock signal and clock enable both are active [36]. The VHDL programming language is fundamentally built around this concept and uses specific syntax in order to infer FFs.
- Interconnections: Many wire traces form the electrical connections between CLBs.
- Input/Output (I/O) pads: These are the physical ports that either connect the FPGA to internal or external peripheral circuits and components, or to the PS in case of a SoC.

In order to increase the computational resources and data throughput, modern FPGA platforms integrate the following additional components:

- Digital Signal Processing (DSP)(48) blocks: Optimized hardware implementation of multiple MAC operations. In *Xilinx* FPGAs, the DSP blocks are called DSP48 and are the most complex functional blocks that are embedded in the FPGA fabric. A single DSP48 unit consists of an Arithmetic Logic Unit (ALU) and a chain of three different computational blocks that can perform functions of the form $P = B \cdot (A + D) + C$ [37].
- Embedded Memory: Besides the LUTs that can be used as memory, FPGAs also feature so-called "block memory" (or Random-Access-Memory Block (BRAM)) areas that are optimized data storage elements. These can be used to implement RAM, ROM, or shift registers. BRAMs are dual-port RAM modules for relatively large collections of data that are directly accessible from PL and therefore have fast access times, compared to off-chip memory. *Xilinx* FPGAs feature two types of BRAM: Modules that can hold 18k bits, or 36k bits at once [38]. The optimizer of the designated design tool automatically chooses to use BRAM whenever feasible, though it is possible to intentionally instantiate BRAM from HDL macros.

[30] While some of the nomenclatures in this section will specifically refer to Xilinx hardware, the concepts apply to other manufacturers FPGAs using corresponding terms.

- Off-Chip Memory Controllers and other peripherals: Since most FPGAs are configured by loading a logic description file from external Static Random-Access Memory (SRAM), a memory controller is needed as well. Some FPGAs already feature on-board configuration memory, but external memory is still needed if the FPGA should be able to process vast amounts of data.

2.3.2 Xilinx Zynq-7000 Platform Overview

The *Xilinx* Zynq-7000 device family is a cost-optimized range of SoCs. Figure 2.13 shows the block diagram of a typical Zynq-7000, which is also used in this thesis.

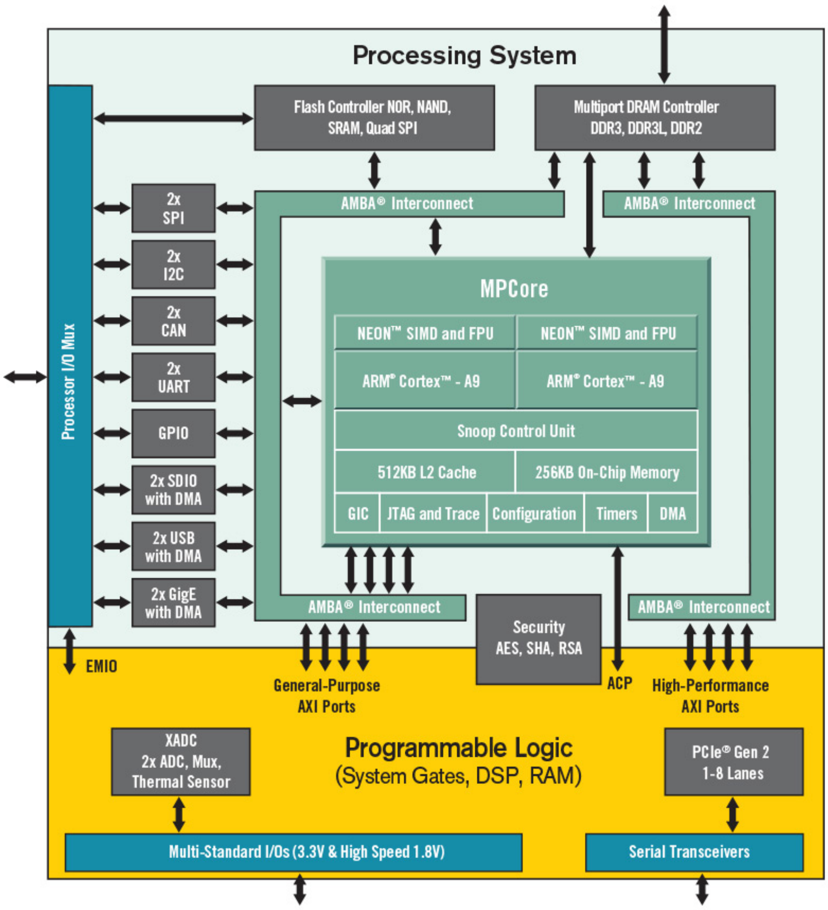


Figure 2.13: Zynq-7000 block diagram. (Source: Xilinx)

It consists of two tightly coupled domains: The PS (in the top of Figure 2.13) and the PL (bottom area of Figure 2.13). The PS contains two ARM Cortex A9[31] processing cores and additional NEON[32] coprocessors. These coprocessors are aimed at multimedia applications and support scalar double-precision floating-point computations, as well as 8, 16, 32, and 64-bit signed and unsigned integer Single Instruction Multiple Data (SIMD) computation. Together with the cache, timers, a General Interrupt Controller (GIC), a Direct Memory Access (DMA) controller, and JTAG interface, this makes up the Application Processing Unit (APU) of the

[31] https://developer.arm.com/ip-products/processors/cortex-a/cortex-a9
[32] https://www.arm.com/why-arm/technologies/neon

PS. Through an AMBA[33] Interconnect interface, the APU communicates with various I/O peripherals, such as SPI, CAN, UART, and GPIO controllers. Via the Extended Multiplexed Input/Output (EMIO) interface, some of these controllers are available to the PL as well [39].

As can be seen in Figure 2.14, there are several different interface options for communication between PS and PL. These can be divided into two groups:

- AXI Master interfaces that connect to slave interfaces in the PL. The CPU in the PS initiates communication through these interfaces. There are two 32-bit AXI master ports available in Zynq-7000 devices: General-Purpose (GP)0 and GP1 [40]. These are simple to implement but do not offer high throughput (<25 MB/s [41]). For the firmware developer, these interfaces can be seen as memory-mapped registers of the CPU, interfacing to hardware accelerators residing in the PL.
- AXI Slave interfaces that connect to master interfaces in the PL. There are four High Performance (HP) ports and an Accelerator Coherency Port (ACP) available. These offer full-duplex 64-bit transfers, meaning that at every clock cycle, 16 bytes of data can be transferred from master to slave and from slave to master concurrently. The two General-Purpose Slave (SGP) interfaces offer 32-bit connections [40]. All these ports are more complex to implement but offer higher throughput (>600 MB/s [41]).

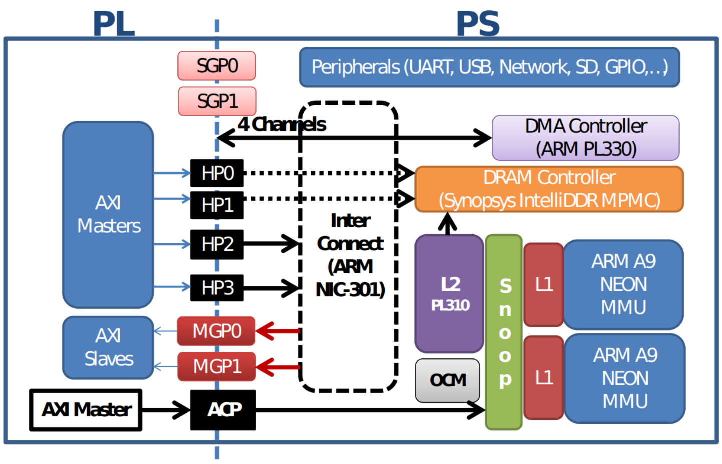


Figure 2.14: Zynq-7000 Signals, Interfaces, and Pins. (Source: [40], "A block diagram representing important elements of the Xilinx ZYNQ device.")

2.3.3 SoC Hardware used in this Thesis

In this thesis, the Arty Z7-20[34] (part number XC7Z020-1CLG400C) by *Digilent*[35] was used (Figure 2.15). This development board is targeted at "Hobbyists" and "Makers" and is related to a budget sector of SoCs platforms. The board is designed around a Zynq-7000 SoC with a dual-core Cortex-A9 processor clocked at 650 MHz. It features 512 MB of DDR3 off-chip memory and Gigabit Ethernet hardware. The Zynq-7000 SoC provides 53200 LUTs, 106400 FFs, and 630 kB of BRAM (divided into 140 blocks). Statements about resource utilization in this thesis will be done in percentages of these available resources.

[33] The AMBA bus was originally developed by ARM and adopted by Xilinx with AMBA Advanced eXtensible Interface (AXI)4, AXI4-Lite and AXI4-Stream as their main communication buses. https://developer.arm.com/architectures/system-architectures/amba

[34] https://digilent.com/reference/programmable-logic/arty-z7/start

[35] https://digilent.com/

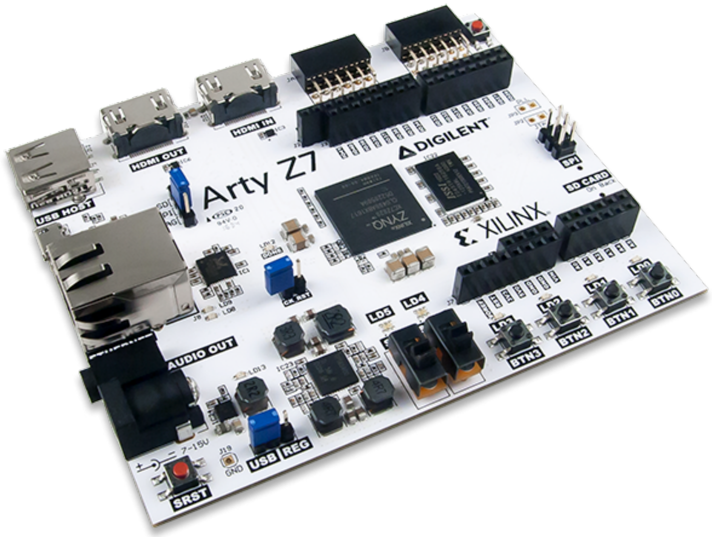


Figure 2.15: The Arty Z7 SoC development board. (Source: Digilent)

2.3.4 Xilinx Development Environment

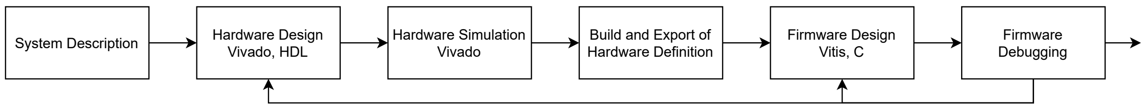


Figure 2.16: Typical SoC development flow.

Xilinx offers many tools, but *Vivado*[36] and *Vitis*[37] are the main parts of the SoC development environment. *Vivado* (successor to *Xilinx ISE*[38]) is used for PL design such as accelerators in HDL, configuration and instantiation of Zynq PS peripherals, or simply for everything in the development flow that does not concern firmware. It offers HDL language support for VHDL and Verilog, full Tool Command Language (Tcl)[39] script operation and extensive tools for simulation and resource reports.

Vitis offers the firmware development environment with compilers/debuggers for C and *C++*. The *Vitis* workflow starts with a hardware definition that is exported from *Vivado*. This hardware specification contains all PS peripherals that are in use, and the bitstream file needed to configure the PL.

[36] https://www.xilinx.com/products/design-tools/vivado.html
[37] https://www.xilinx.com/products/design-tools/vitis.html
[38] https://www.xilinx.com/products/design-tools/ise-design-suite.html
[39] https://en.wikipedia.org/wiki/Tcl

3 Analysis of the Convolutional Neural Network

This chapter analyzes the underlying CNN in detail. The examination starts with the original CNN [7] and its quantization-aware training approach, including a short overview of the software implementation of the CNN forward path, that was used as the basis for this thesis. An attempt is made to augment the previous examination by revealing the scope of necessary computations and data types involved.

Then, the network modifications to derive a new software implementation of the inference path are presented in Section 3.2. This software implementation breaks the CNN inference down into the most basic math operations and only uses fixed-point integer data types to compare various quantization combinations with the original CNN accuracy [7]. Results of the comparisons are discussed in Section 3.2.1.

3.1 Network Overview

The existing CNN performs interference mitigation in the frequency domain after the second Discrete Fourier Transform (DFT) of the range-Doppler processing chain (Figure 3.1) [7]. This operation can also be seen as a filtering of the RD map using trainable filter kernels.

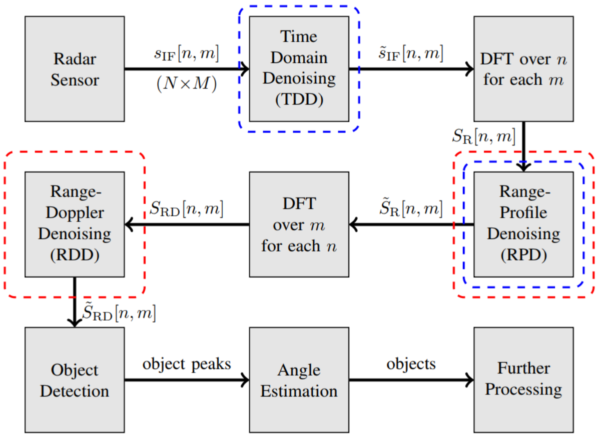


Figure 3.1: FMCW processing block diagram. The blue dashed squares show implementation points for conventional denoising algorithms and the red squares mark optional locations for NN-based denoising. This CNN performs the denoising after the second DFT of the RD map processing chain. (Source: [4], "Block diagram of a basic FMCW/CS radar processing chain")

The aim of previous investigations [7] was to "find small models with decent resource requirements that retain high interference mitigation performance", which was attempted with two different model architectures, only the more efficient resulting architecture is used in this thesis, which can be seen in Figure 3.2. Inputs are complex-valued RD maps with (simulated) interference and targets without interference (clean). Two channels are used for real and imaginary values. Square (3×3) kernels are used in the convolution, and zero-padding in order to gain outputs with the same spatial dimensions (96×96) as the inputs for every layer. During training, the network uses the Mean Squared Error (MSE) as loss function and the *Adam* algorithm[40].

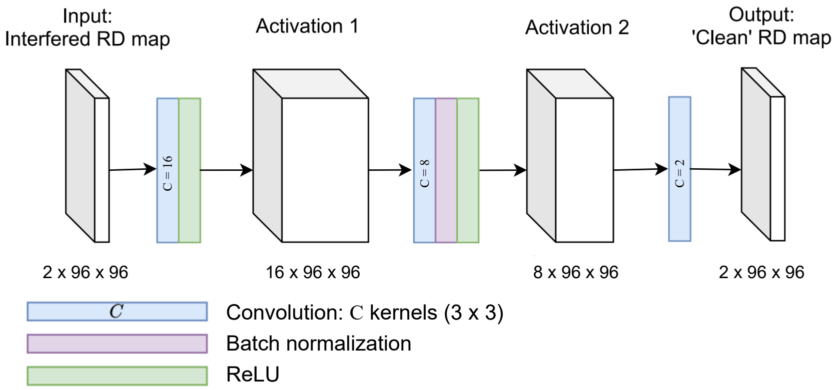


Figure 3.2: CNN architecture diagram. (Altered from source: [7], "Bottle-neck based architecture")

The model consists of three convolutional layers. Each layer performs a convolution operation and optionally the ReLU activation function and batch normalization. The purpose of the scaling operation in every layer is mentioned in Section 3.1.1. Rounding is performed after layers 1 and 2, in order to gain 8-bit (unsigned) activations. Detailed metrics can be seen in Table 3.1. A single input tensor (frame) consists of a complex-valued ($2 \times 96 \times 96$) 2D RD map, which is taken from an FMCW snapshot. The outputs of the CNN are of the same dimensions. After layers 1 and 2, the ReLU activation function is applied to the activations.

	IN Ch.	N	M	**OUT Ch.**	**Additional Processing**
Layer 1	2	96	96	16	ReLU, Scale, Round.
Layer 2	16	96	96	8	Scale, Batch Norm. Bias, ReLU, Round.
Layer 3	8	96	96	2	Scale, Weight Bias

Table 3.1: Model metrics overview.

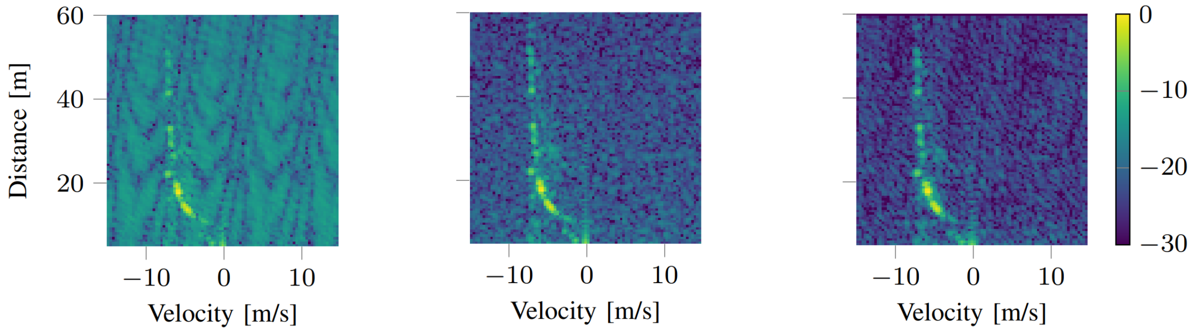
For training, test and validation, three sets of recorded measurements are available that contain a large number of RD maps (Table 3.2) [4]. Each measurement data sample consists of a RD map with simulated interference (as input), one without interference (as target), a manually labeled object mask and respective noise mask (for evaluation).

	Training Set	**Test Set**	**Validation Set**
Number data samples	2500	250	250

Table 3.2: Available data sets for training, test, and validation.

Figure 3.3 shows interference mitigation example performance of the CNN, visualized as magnitude images. As is clearly visible, the interfered signal (left) shows a significant noise floor over the measurements, compared to the clean RD map in the middle. The RD map on the right is the output of the CNN with a much lower noise floor that even undercuts the clean measurement [7].

[40] Adam is a variant of gradient descent [42].



(a) Interfered (b) Clean (c) After CNN Mitigation

Figure 3.3: Exemplary RD magnitude spectra. (Source: [4], "Exemplary RD magnitude spectra from the measurements test set.")

3.1.1 Training

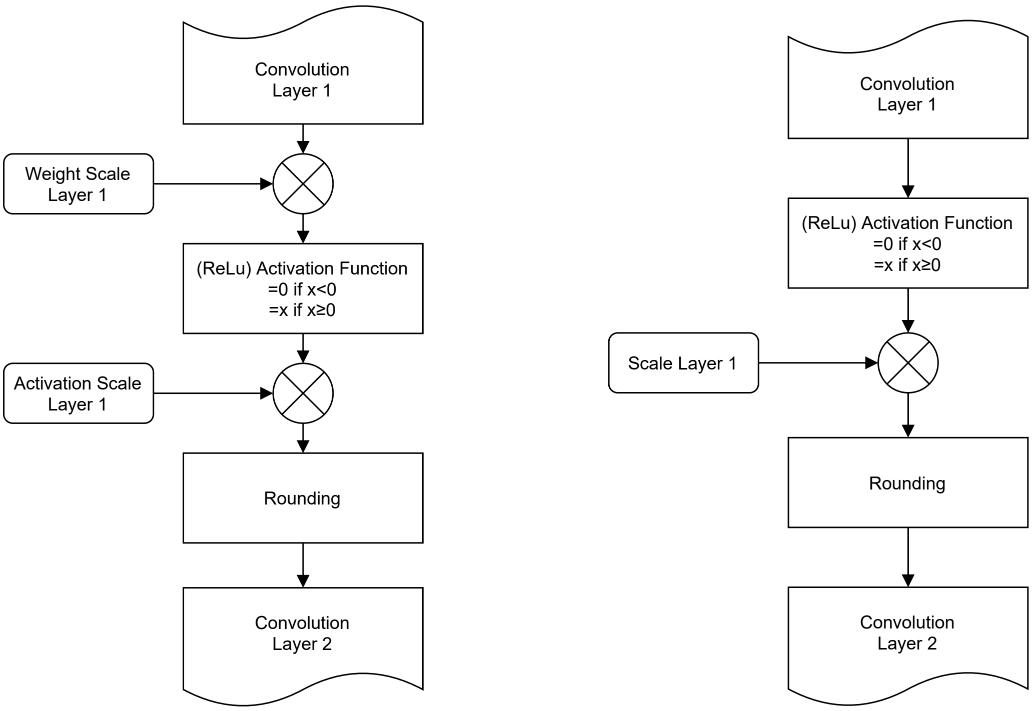
As already mentioned, the initial CNN [7] uses a quantization-aware training approach (Section 2.2.5) in order to conserve memory resources. It was shown that 8-bit-quantization of weights and activations yields the best results for the given model architecture[41] [7].

Integer quantization maps real-valued weights to the closest integer value [7]. In order to exploit the whole integer range of the given bit width, a dynamic range value is used, which leads to a scaling operation that increases model performance considerably [7]. This dynamic range value is a real-valued number that does not contribute to memory requirements, since only one such value is needed per layer. But, scaling with a real-valued number could decrease performance on an embedded system, if the number is stored and used in a floating-point format.

Since there is one scaling value stored for weights and another for activations, it is possible to combine these values and reduce the number of multiplication in each of the first two layers to one [7], in a process similar to *constant folding/propagation*[42]. Figure 3.4 illustrates the locations of the scaling operations for the first layer and how these values can be combined in order to reduce the amount of multiplications. Factor combination can be done during training, so the number of computations during inference is decreased. Still, even when combined, the scaling values are real-valued numbers, requiring more costly floating-point operations. In this thesis we show that integer quantization is sufficient in order to decrease the complexity of computations for embedded hardware, without significant loss in accuracy.

[41] Considering fixed bit widths for weights and activations. Learned bit widths for activations with an average of 6.5 bit could decrease memory requirements even further [7].

[42] https://en.wikipedia.org/wiki/Constant_folding



(a) Two scaling operations. (b) Combination of scaling operations.

Figure 3.4: The two scaling operations in 3.4(a) can be combined to a single multiplication as shown in 3.4(b).

3.1.2 Base Software Implementation

Figure 3.5 shows a flow diagram of the software implementation of the initial CNN [7]. In this thesis only the forward pass is considered, since the embedded CNN implementations focusses only in inference rather than on-device training.

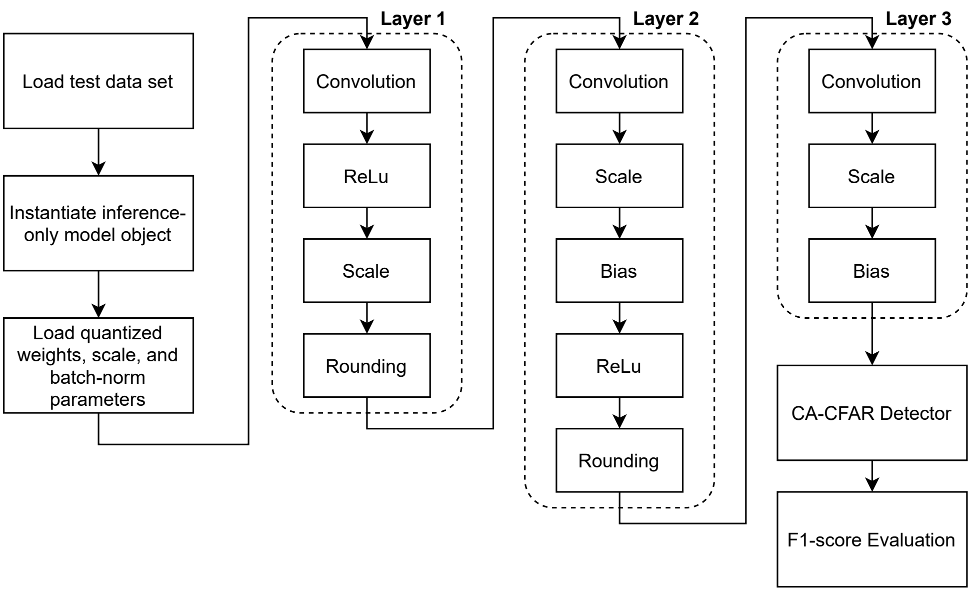


Figure 3.5: Short CNN software flow diagram.

After training, all weights and parameters for scaling and batch normalization are exported from the training framework and loaded into the ebedded inference implementation. The convolution operations are implemented as *TensorFlow*-functions `tf.keras.layers.Conv2D`[43], the ReLU activation is implemented using the *Numpy*[44] function `out = np.maximum(out, 0)`, which replaces all negative activation values with zero. The rounding operation is implemented as `out = np.round(np.clip(out, UINT8_MIN, UINT8_MAX))`, such that not only the fractional part of the value is removed, but also the maximum values are clipped to the range of 8-bit unsigned integer numbers.

After the CNN is executed for all input data sets, a Cell-Averaging Constant False Alarm Rate (CA-CFAR) object detection algorithm [43] is applied to the outputs of the CNN. The results of the object detection are then compared to the manually labeled reference data from the data sets, in order to compute the F1-score (Section 5.1) as performance measure of the CNN.

Data Types: It is important to note that the original CNN implementation [7] uses various data types. Firstly, the inputs (RD maps) are single-precision (32-bit) floating-point values, since this is the format that is produced by the preceding RD map processing. The inputs are convolved with 8-bit signed integer weights. This process again results in floating-point values, which are scaled with single-precision (32-bit) floating-point scale factors. After the rounding operation, the results of the first layer are 8-bit unsigned integers, which are passed on to layer 2. Convolution in layer 2 works solely with 8-bit integers (unsigned and signed), but the outputs are scaled by single-precision (32-bit) floating-point scale factors. After rounding, 8-bit unsigned integers are passed on to the next layer. In layer 3 the convolution works analogously to the last layer, while the layer outputs are at the same time the CNN model outputs and thus the filtered RD map predictions.

3.2 Fully Quantized Software Implementation

This section presents additional quantizations that were performed on the network weights to convert all MAC operations from floating-point to fixed-point operations. Specifically for FPGA hardware, this reduces resource requirements dramatically. In CPU firmware, fixed-point operations are faster than their floating-point equivalent despite additional floating-point coprocessing hardware that accelerates floating-point operations.

Input Conversion: Quantization of input values is done by searching through all 2500 input frames in the training set and looking for the maximum absolute value x_{max}. The upper limit of the desired k-bit signed integer range $r = 2^{k-1} - 1$ is divided by x_{max} in order to find a scale value $s = \frac{r}{x_{max}}$. Since scaling by powers of two by performing bit-shift operations is a cheap implementation on embedded hardware, an exponent m is needed, such that $2^m \leq s < 2^{m+1}$ is true. This exponent m donates the input bit-shift value throughout this thesis.

Scale Conversion: The scale factors f_1, f_2, f_3 for all three layers are real-valued numbers (Section 3.1.1). In the same manner as for the input conversion, every individual layer's scale factor is multiplied by a factor 2^{m_1}, 2^{m_2}, 2^{m_3}, such that the result satisfies $2^{k-2} - 1 \leq |f_l \cdot 2^{m_l}| \leq 2^{k-1} - 1$. This ensures the highest possible resolution of a real-valued number in a fixed-point integer format when using bit-shift operations for scaling.

Convolution Unrolling: In the original implementation [7], the convolution operation was performed in dedicated 2D-convolution functions that are provided in the *TensorFlow*-framework.

[43] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D
[44] https://numpy.org/

In order to break these convolutions down into atomic MAC operations, the whole convolution of a single layer was rewritten as loops. The pseudo-code in Figure 3.6 shows the implementation of one such convolution.

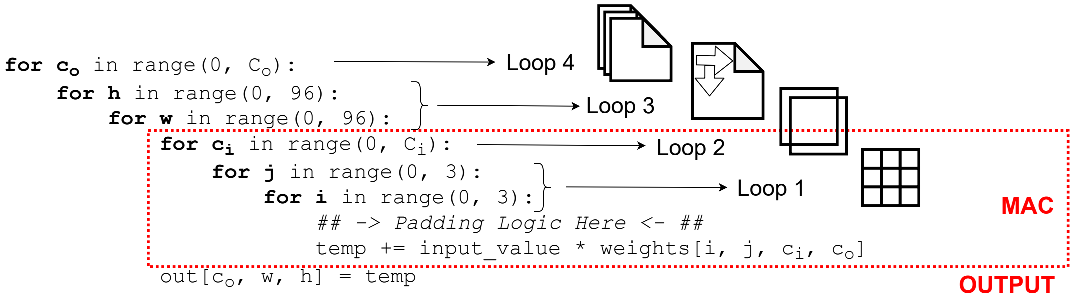


Figure 3.6: Convolution loop for a single CNN layer. Loop 1 and loop 3 each are a pair of loops, which could be linearized in a known memory layout.

Loop 1 iterates over both spatial dimension of the 2D-filter-kernel and consists of two actual loops to serve both dimensions. Loop 2 repeats these operations for all channels of the 2D-input-map. This comprises one MAC computation of the convolution, where the result is a single point on the output map for a single output channel. In loop 3, the convolution is slid over both spatial dimensions of the input map, deriving one complete 2D channel of the output feature map. Since there are multiple output channels (e.g. 16 for the first layer), loop 4 repeats the whole process for all output channels.

Zero-padding is implemented in the padding logic of the MAC operation in the convolution process. This padding logic ensures, that values of the input feature map are set to zero if a spatial index is negative or exceeds the maximum resolution (96×96 pixels width and height).

ReLU Activation Function: The CNN uses the ReLU activation function, which is done simply by setting negative values to zero and applying the identity function for values greater or equals zero (Listing 3.1).

```python
if temp < 0:
    temp = 0
```

Listing 3.1: ReLU activation function in Python.

Rounding: The rounding operation for fixed-point values becomes clear by looking at the fixed-point number representation of real-valued numbers. As an example, a real number such as 2.5 can be converted to a fixed-point representation by a multiplication with 10. The number $p = 10$ is the value that defines the position of the comma in a fixed-point representation, and the real number is represented as an integer value $i = 25$. In order to round the real number 2.5 in a fixed-point representation to the next integer equivalent, the fractional part 0.5 needs to be added, e.g.: $round(2.5) = 2.5 + 0.5 = 3$. Rounding integer values is done in multiple steps, as can be seen in Listing 3.2.

1. Assuming a real-valued number that was converted to a fixed-point representation by a scale of 2^{13}.

2. Divide fixed-point integer `temp` by scale 2^{13}. In the code example this is done by a bit-shift operation. Shifting `temp` 13 bits to the right exposes the integer component. Any bits right to the shift position are lost. [Listing 3.2, Line 1]

3. Shifting the result 13 bits to the left again, leaves the previous fixed-point integer, but without the fractional part. The result is stored in `rounding_temp`. [Listing 3.2, Line 2]

4. `rounding_temp` is subtracted from `temp`, which derives the fractional part in `temp_fractional`. [Listing 3.2, Lines 4-5]

5. The fractional part `temp_fractional` is added to `temp`, which results in a fixed-point number with an integer part, that is equal to the rounded floating-point number. [Listing 3.2, Lines 7-8]

6. By shifting the result 13 bits to the right again, the number is reduced to the rounded integer value. [Listing 3.2, Lines 10-11]

```
1 rounding_temp = np.array(temp >> 13)
2 rounding_temp = rounding_temp << 13
3
4 temp_fractional = temp - rounding_temp
5 temp_fractional = temp_fractional.astype(np.int32)
6
7 temp = temp + temp_fractional
8 temp = temp.astype(np.int32)
9
10 temp = temp >> 13
11 temp = temp.astype(np.uint8)
```

Listing 3.2: Rounding operation in integer math in Python.

3.2.1 Quantization Schemes and Network Performance

From the fully quantized software implementation, it is possible to further improve resource-efficiency by reducing:

- Bit width of inputs of the CNN
- Bit width of ouputs of the CNN

Inputs and outputs are values in fixed-point integer format and reducing the bit width of the integer part of these numbers yields two effects on the network:

- Less memory requirements for storing inputs and outputs.
- Potentially more quantization noise and therefore lower accuracy.

Figure 3.7 shows how the scale exponent for a given input bit width is dragged through the computation of the network. The resulting bit width of an integer multiplication is the sum of the bit widths of both factors. For an addition, a single bit has to be added. In a convolution, multiple products are added together, but not every addition increases the bit width by one. Instead, the resulting bit width B is calculated as $B = K + ceil(\frac{log(M)}{log(2)})$, with K being the product's bit width and M the amount of additions in the convolution [44]. In this example, the inputs were converted to fixed-point numbers by a multiplication with 2^{27}. After the first convolution, the resulting bit width is 45. In order to stay in a 32-bit range, the results have to be shifted to the right by 13 bits. This also decreases the scale exponent to 14, since a bit-shift to the right is equal to a division by the number of bits to the power of two. The results of layer 3 are computed in 32-bit accuracy in any case and optionally truncated to a desired bit width.

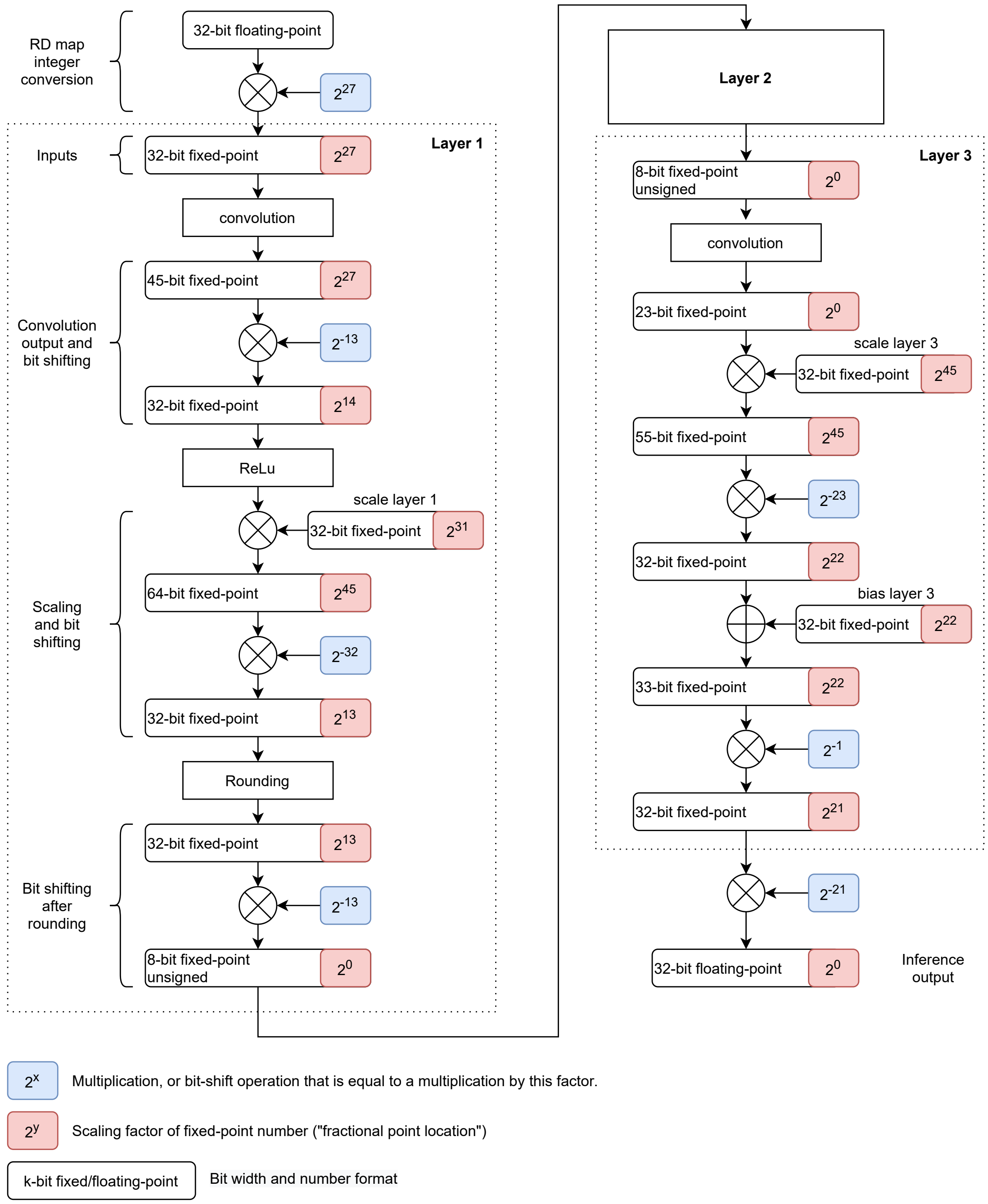


Figure 3.7: Locations of fixed-point integer scale values tracked throughout the whole CNN implementation.

Three combinations of input/output bit widths were examined and compared to the floating-point implementation as reference. In order to evaluate the change in memory requirements, the number of values in an input/output map is multiplied by the bit width of a single value. Effects of quantization noise on the network accuracy are investigated by evaluating the F1-score. For both measures, the results are presented in Figure 3.8 and Table 3.3. As can be seen, changing the number format from floating-point to fixed-point hardly influences the accuracy of the network. Reducing the bit with of inputs and outputs to 16-bit decreases accuracy slightly, while memory requirements of the corresponding layers is cut in half. Lowering input bit width to 8-bit and inferring 16-bit outputs cuts another quarter from memory requirements but also significantly decreases overall accuracy. 8-bit values for inputs and outputs results in bad performance, which is why 16-bit outputs are used for 8-bit inputs.

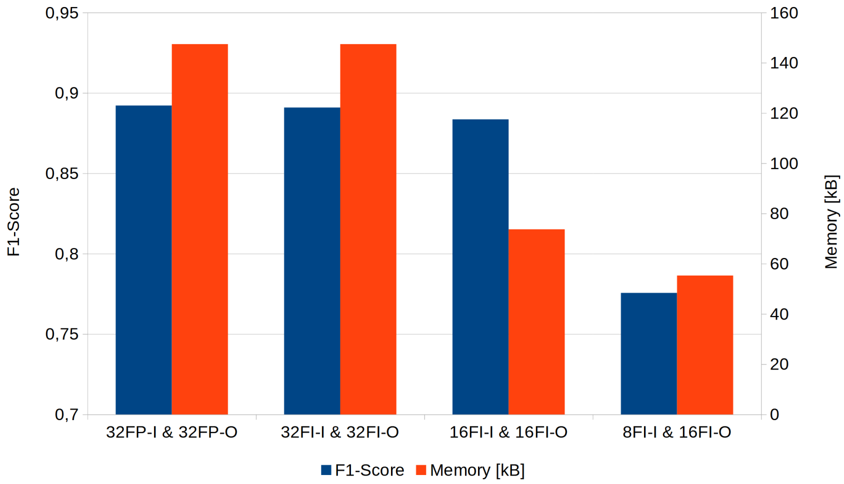


Figure 3.8: Memory requirements for storing a single input and output frame compared to F1-score for specific number formats. 32FP-I meaning 32-bit floating-point inputs and 32FP-O 32-bit floating-point outputs. FI stands for fixed-point.

Number Format	Abbreviation	F1-Score	Memory [kB]
32-bit float.-p. inputs/outputs	32FP-I & 32FP-O	0.8922	147.456
32-bit fixed-p. inputs/outputs	32FI-I & 32FI-O	0.8909	147.456
16-bit fixed-p. inputs/outputs	16FI-I & 16FI-O	0.8836	73.728
8-bit fixed-p. inputs & 16-bit outputs	8FI-I & 16FI-O	0.7756	55.296

Table 3.3: Memory requirements for storing a single input and output frame compared to F1-score for specific number formats.

3.2.2 Estimation of Computational Efforts

Table 3.4 shows the computations that are needed to infer a complete output frame of the CNN. Most of the network consists of MAC operations, but the scaling in layer 1 needs pure multiplication operations. As can be seen, a total of 1.4 million MACs are required with an additional 10 % of multiplications. Bit-shifts for scaling and rounding are not shown in this table.

	MACs	Multiplications
Layer 1 Convolution	$3 \times 3 \times 96 \times 96 \times 2 \times 16 =$ 2 654 208	
Layer 1 Scale		$96 \times 96 \times 16 =$ 147 456
Layer 2 Convolution	$3 \times 3 \times 96 \times 96 \times 16 \times 8 =$ 10 616 832	
Layer 2 Scale + Bias	$96 \times 96 \times 8 =$ 73 728	
Layer 3 Convolution	$3 \times 3 \times 96 \times 96 \times 8 \times 2 =$ 1 327 104	
Layer 3 Scale + Bias	$96 \times 96 \times 2 =$ 18 432	
Total MACs	**14 690 304**	
Total Multiplications		**147 456**

Table 3.4: Estimation of required MAC operations and multiplications for the inference of one complete frame.

4 Embedded CNN Design and Implementation

The following sections explain the implementation of the CNN algorithm on embedded hardware. Two distinctive variations of the same CNN were realized on the SoC in this thesis:

- A firmware solution that is executed entirely inside the PS
- A hardware implementation in PL that performs the computationally expensive tasks of the CNN and acts as an accelerator to the CPU in PS.

A block diagram of the overall SoC design is explained in the beginning of this chapter, followed by the analysis of the firmware implementation in Section 4.2. This section also shows how the hardware accelerator can be used from within the firmware.

Section 4.3 explains the hardware accelerator design and presents three different implementation methods that differ in the amount of parallel computations. The effects of different rates of parallelization are presented in Section 4.3.4.

4.1 Overview

Figure 4.2 shows a block diagram of the graphical programming environment in *Vivado*. This tool is used to configure the main components of SoC design. The two main components are:

- The ZYNQ-7000 Processing System: It hosts the firmware implementation of the CNN and performs data transmissions over an ethernet connection.
- The custom CNN hardware accelerator (*multiple_mac_seq_cnn_hw_acc*): This is the implementation of the CNN in VHDL in the PL of the SoC.

SoC Design Analysis: There are several blocks visible in Figure 4.2, and each block either represents an instance of an embedded peripheral, such as the *ZYNQ7 Processing System*, or a custom logic implementation in PL, such as the CNN hardware accelerator instance. The *AXI Interconnect* instance is used to control data traffic on the AXI bus between AXI masters and slaves, and exposes components as memory-mapped devices to the CPU. *Processor System Reset* is a component that is automatically initialized with every design that uses the *ZYNQ7 Processing System* and manages the start-up sequence of the PS after a reset. Finally, *AXI GPIO* is initialized, which is a custom logic block that is provided by *Xilinx* to control General-Purpose Input/Outputs (GPIOs) of the SoC.

SoC Configuration: The following configurations are set in the *ZYNQ7 Processing System*:

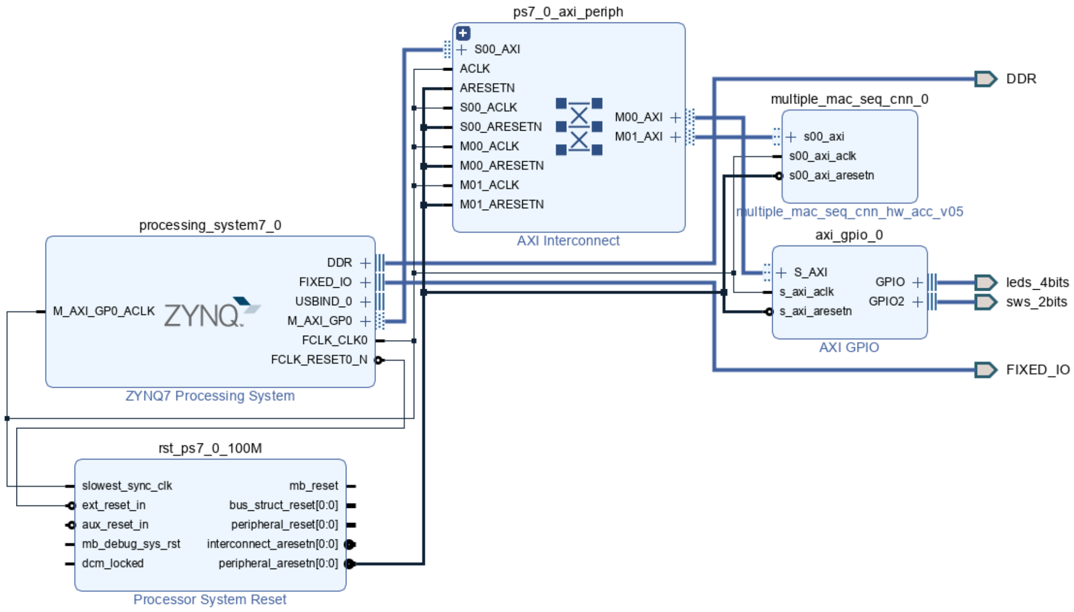


Figure 4.1: Overview of the hardware design, including the Zynq-7000 processing system and the accelerator in PL.

- *GP0 AXI Master Interface* enabled. This enables memory-mapped register-level data transfer to and from AXI slave peripherals in the PL.
- *I/O peripheral Ethernet 0* enabled. This initiates the embedded Gigabit Ethernet Controller (GEM), which implements a 10/100/1000 Mb/s Ethernet Media Access Control (EMAC)[45] interface. The *Ethernet 0* controller uses Reduced Gigabit Media-Independent Interface (RGMII) signals to connect to an external Physical Layer (PHY) chip. The RGMII signals are routed to external pins on the SoC through the Multiplexed Input/Output (MIO) interface.
- *I/O peripheral UART 0* enabled. This interface is needed for debug communication with the SoC.
- *I/O peripheral Quad SPI Flash* enabled. The QSPI controller connects to an external QSPI chip that contains the configuration bitstream of the SoC.
- CPU and DDR clock frequencies are set to the default values of the device (CPU: 650 MHz; DDR: 525 MHz). Higher values might not work with this particular device, but are possible with higher speed grade SoCs.
- *PL fabric clock 0* frequency is set to 100 MHz. The *Zynq 7000* features four different clock sources for PL. Only one is used in this thesis. This clock is the central timing constant for all logic in the PL of the SoC. While it is possible to set this clock to a maximum frequency of 250 MHz on this device, the higher frequency introduces signal integrity issues and might lead to less parallelization in the hardware accelerator[46].
- The DDR controller is enabled. The SoC board that is used in this thesis features a 512 MB DDR3 memory chip, which is connected to a dedicated interface on the SoC and used for ethernet data package buffering.

[45] The correct abbreviation for the Media Access Control component is MAC. Since this would lead to confusion with the Multiply-And-Accumulate operation, the abbreviation is changed to EMAC in this thesis.

[46] Signal integrity is violated, if multiple inputs of a FF cannot arrive simultaneously. This could happen if signal sources are physically scattered in distant locations of the PL or if FF slew rate is too slow.

CPU-Offloading: The CNN hardware accelerator cannot operate on RD maps without the help of a CPU. This is because it does not implement the logic that is needed to communicate with the PC that offers the radar measurement data. Hardware accelerators are supporting components to the CPU. The concept of lifting computationally intensive tasks onto supporting hardware accelerators is called "CPU-Offloading".

In this case, the CPU initiates the custom CNN hardware accelerator when it is done receiving one complete RD map frame from the PC. The CPU then proceeds to transmit this data from DDR memory to the accelerator through dedicated interfaces between PS and PL. Since the transmission of data from the PS to the PL introduces latency, the developer has to ensure that the latency reduction gained from lifting computations to the accelerator exceeds the latency introduced by the transfer between PS and PL [45].

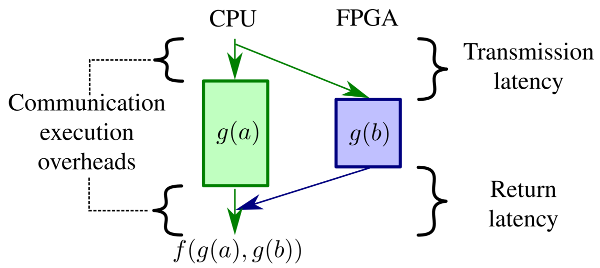


Figure 4.2: Illustration of transfer latency requirements when using hardware accelerators for computationally intensive tasks. (source: [45], "Cooperative system architecture and overheads")

4.2 Firmware Implementation

The firmware of the SoC actually serves two purposes: It is the central control and communication logic and features a complete implementation of the CNN.

Figure 4.3 shows a flow diagram of the firmware. In the first step, the SoC platform is initialized, and all memory areas are cleared. Then, the communication logic is initialized. This is essentially an implementation of a TCP/IP stack and enables data transfer over ethernet to and from a PC. The TCP/IP implementation uses the *LightWeight IP (lwIP)* library by Adam Dunkels [46], with the SoC acting as the server and the Python script on a PC being the client that connects to it. IP address and port number of the server are defined in the firmware header and set during compilation.

When a connection is established, the firmware waits for a RD map to be transmitted. In this implementation, the firmware will only transition to the next step, if a complete frame of $96 \times 96 \times 2 \times 4 = 73\,728$ bytes was received, because the transmission is always performed in regards to 32-bit signed integer numbers.

By design, *lwIP* buffers received data in DDR memory. So, in order to make the RD map data available to the processor, it is transferred to internal Random-Access Memory (RAM) (Section 2.3.1). Processed data that is returned to the PC must be buffered in DDR upfront as well.

When input data is ready for processing, either the firmware or hardware implementation of the CNN may be executed. It is also possible to bypass CNN processing alltogether. A define statement in the firmware header compiles the firmware either for hardware, firmware or bypassed operation. Two timing measurement locations (A and B in Figure 4.3) are embedded in the processing flow. During debugging, it is possible to measure

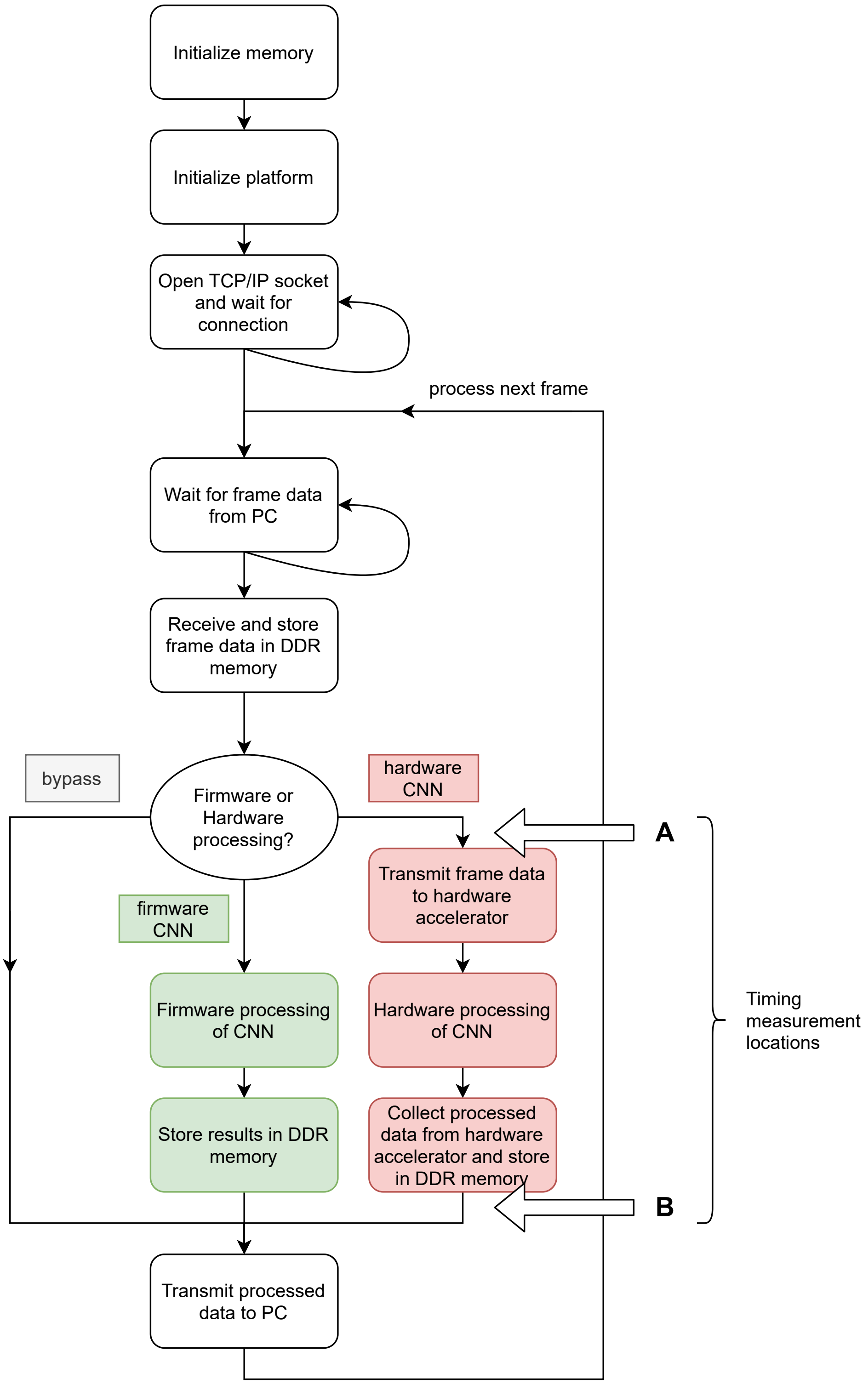


Figure 4.3: Firmware flow diagram.

the time span between these two points, in order to evaluate the latency of firmware and hardware processing.

4.2.1 CNN Implementation in Firmware

The firmware implementation of the CNN is realized in three functions, that each correspond to one layer of the network, and are executed sequentially. Data is passed between the layers by pointers to memory regions to minimize data copy operations. All convolution operations are rewritten as loops, as has been shown in Section 3.2. The MAC operation and padding logic are shown in Listing 4.1. If the index of a point on the RD map is negative or exceeds the width or height of the map, the value is set to zero and thus the multiplication with a weight value also results in zero[47].

```
a = weight_index_2 - weight_offset + input_index_2;
b = weight_index_1 - weight_offset + input_index_1;
if ((a < 0) || (a >= layer_size) || (b < 0) || (b >= layer_size))
{
  temp_input = (int64_t)0;
}
else
{
  temp_input = (int64_t) *(input_data+(input_layer_index + (a * 2) + ((2*96) * b)));
}
temp_result += temp_input * (int64_t)(weights[weight_index_2][weight_index_1][
        input_layer_index][output_layer_index]);
```

Listing 4.1: Padding logic in the firmware implementation of the CNN.

Weights and scale values are defined as constant integer arrays in a header file of the firmware. Weights are stored as 8-bit signed integers and scale values as 32-bit signed integers. The MAC operation works in a 64-bit range, bits exceeding the 32-bit range for post-convolution processing are eliminated by bit-shifting. It is also possible to operate the firmware CNN with single-precision floating-point values. In this case, the bit-shift operations are not required.

4.2.2 Resource Utilization of Firmware Implementation

Figure 4.4 and Table 4.1 show the resource requirements on the SoC for a pure firmware implementation. Resource requirements in % are in regards to the total amount of available resources (Section 2.3.3). Since there is no programmable logic in use, the requirements are very low. The processing system is embedded in the fabric of the SoC and does not use additional resources.

Whether the firmware CNN uses floating-point or fixed-point operations does not matter regarding resource utilization and will only be visible in the timing analysis.

[47] While it would be more efficient to skip the multiplication with the prospect of possibly more jitter, the optimizing compiler most likely performs this step anyway. Furthermore, jitter (meaning uncertain termination time of a processing cycle) is not of concern in this application.

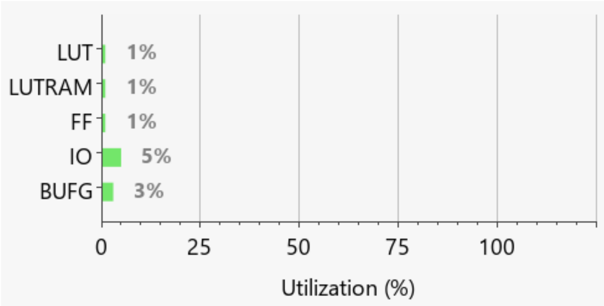


Figure 4.4: Resource utilization on SoC for firmware implementation. Diagram output as generated from Xilinx Vivado implementation report tool.

Resource	Utilization	Available	Utilization %
LUT	542	53200	1.01
LUTRAM	62	17400	0.35
FF	745	106400	0.70
BRAM	0	140	0.00
DSP	0	220	0.00
I/O	6	125	4.80
BUFG	1	32	3.12

Table 4.1: Resource utilization output of Xilinx Vivado IDE for firmware implementation. Available and utilized resources are given in total number of units on the specific Zynq platform.

4.3 Hardware Accelerator

This section describes the CNN hardware accelerator in PL of the SoC. It works in co-operation with the CPU firmware, as previously described (Section 4.1). The hardware accelerator performs complex computations that otherwise are done by the CPU. An accelerator in PL can be designed with a high rate of parallel computation, so it should be possible to decrease processing latency significantly, compared to firmware CNN processing. Since there can be a lot of overhead for data transfer to an accelerator, it is important to evaluate overall throughput with and without the accelerator. Processing latency is decreased only if both accelerator processing and data transfer latency are lower than pure CPU processing duration.

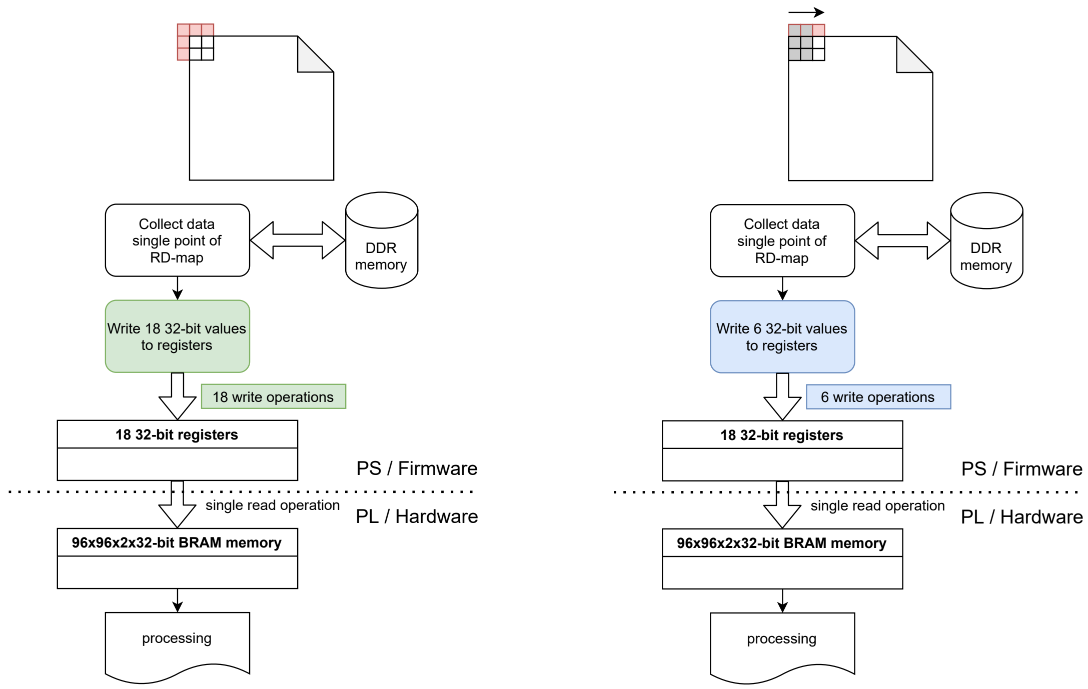
4.3.1 Overview

Since the accelerator is connected to the GP0 AXI master port (as an AXI slave), the CPU communicates with the accelerator through a set of 36 32-bit registers. Eighteen registers are dedicated to input data that consists of $3 \times 3 \times 2$ values needed by a single kernel convolution in the first layer. Sixteen registers for output data[48], and two additional registers for processing control and transmission hand-shaking. Weights are hard-coded into LUT memory[49], and scale values can be changed during synthesis of the accelerator in a graphical user interface (see Section 4.3.5) when the accelerator is connected to a PS.

Input data is transferred to the accelerator in kernel-sized blocks since one kernel of input data infers a single point on all output channels of the output map of layer 1. To initiate

[48] Outputs are transfered in blocks of sixteen values, instead of kernel-sized blocks.
[49] BRAM is only inferred by *Vivado* for larger memory requirements.

CNN processing, the CPU writes the first block of data into the input data registers and sets a flag in the control register, as shown in Figure 4.5(a). When the accelerator is done with the first point on the output map of layer 1, the CPU continues with the next point in the row. The accelerator already has two columns of this subsequent block stored from the last processing, so the CPU only needs to transfer the additional new column (six values) (Figure 4.5(b)). The hardware accelerator automatically shifts the former two columns to the left in its internal buffer and writes the input values into the following column. Again the CPU sets a ready flag in the control register and the accelerator processes the data.



(a) First transfer from PS to PL. (b) Second transfer from PS to PL.

Figure 4.5: Illustration of the transmission flow from PS to PL. In the first transfer of a row, the CPU has to perform eighteen write operations, because one kernel-sized input consists of $3x3x2 = 18$ *values. Subsequent columns require only six write operations, because only one novel column of* $1x3x2 = 6$ *values is transfered.*

This process continues until the end of a row is reached. In the next top-down row, the CPU again transmits a whole block of eighteen values to the accelerator buffer. Figure 4.6 illustrates the transfer protocol from start to finish of a complete RD map.

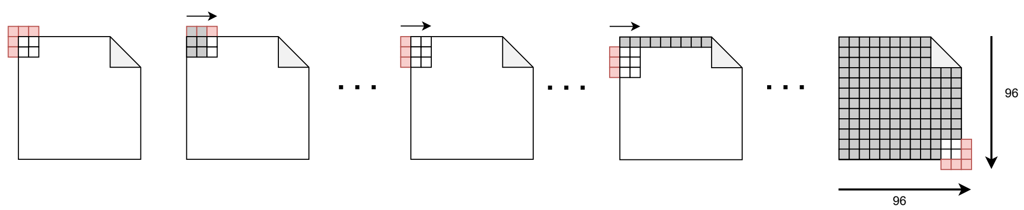


Figure 4.6: Illustration of the whole transmission of a complete (96×96)*-pixel RD map frame from PS to PL. Processing is done in kernel-sized chunks, so the transmission is also performed one kernel size of the RD map at a time. The first transmission starts at the first column of the first row of the RD map frame and continues along the row, while each consecutive transmission only writes the new column that has not been transmitted before.*

When a complete RD map frame is processed by the hardware CNN, the accelerator starts the data transfer back from PL to the PS. This is done in dedicated output data registers that can hold sixteen 32-bit values. The accelerator signals with a flag in the control register that the data is ready and the CPU collects the values and writes them to DDR memory. This is repeated until the complete inference output RD map of the CNN has been transferred back to the PS. Alternatively, significant portions of this process could be implemented using DMA for decreased CPU load, presuming perfect knowledge of memory locations.

The outputs of layer 1 are stored in the PL, either in BRAM or LUT memory. Layer 2 automatically fetches the needed data and generates the points on its output map, which again are stored in PL memory. When layer 2 is done, layer 3 fetches the required data and generates the inference output of the network, which is stored in PL memory. After this, the accelerator initiates the transfer back to the PS, which has to be accepted in firmware.

4.3.2 Single Layer in Hardware

Figure 4.7 shows a more detailed illustration of the computations in layer 1 of the accelerator. This illustration also displays the bit growth that happens to integer values during certain computations (Section 3.2.1).

The hardware accelerator performs each convolution in complete integer precision. After the convolution, numbers are clipped to 32-bit integers again. This way, additional quantization errors are minimized in the convolution, and further bit-growth is prevented. The omission of additional bits can be seen as a bit-shift operation, though, in hardware, this is done simply by omitting unused signals. A bit-shift operation on integer values still equals a division by a power of two. As can be seen in Figure 4.7, by omitting 13 bits after the convolution of layer 1, the initial scale factor of 2^{27} is reduced to 2^{14}.

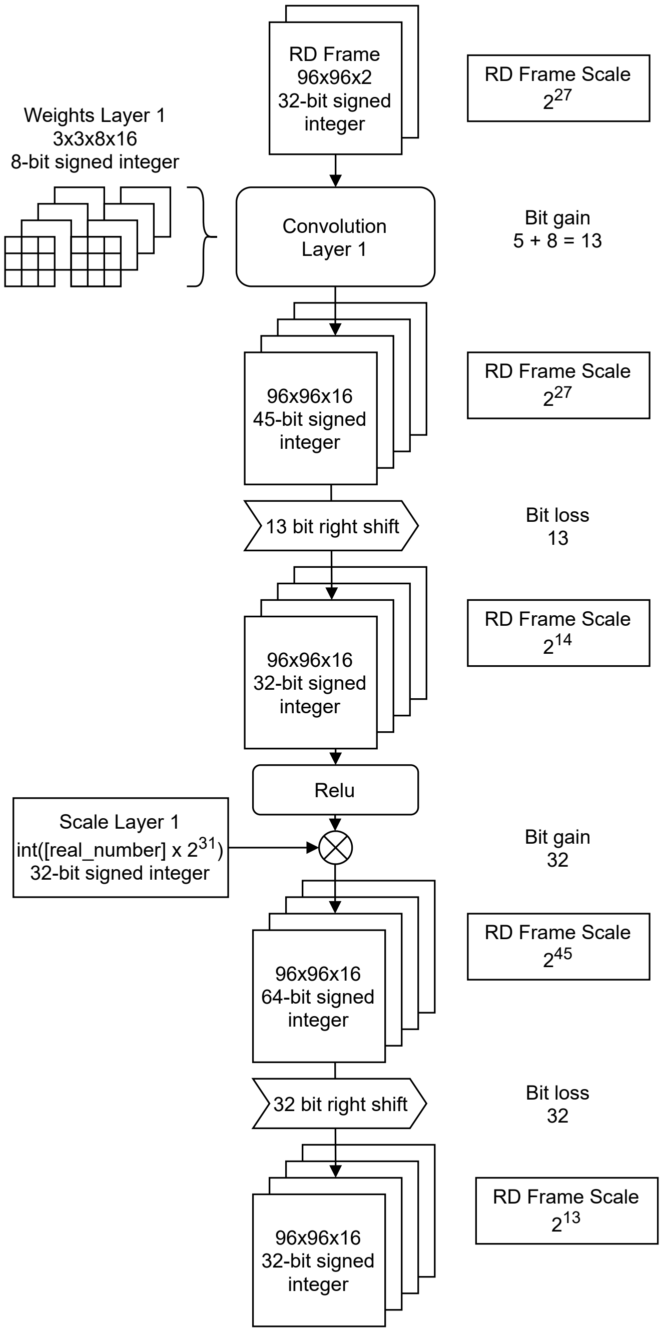


Figure 4.7: Part of the accelerator flow diagram that shows bit growth and tracks the input scale factor.

4.3.3 CNN Hardware Accelerator Implementation

The amount of parallel computations is a decisive factor in the latency of a hardware accelerator and a major paradigm shift from purely sequential code execution as in CPU software, but estimating the maximum amount of parallel computations in advance to accelerator design is a nearly impossible task. The CNN in this thesis requires more than 14 million MAC operations to infer a complete output map. By slicing the input map into kernel-sized blocks and processing these 96×96 blocks sequentially, each layer's amount of MAC operations is almost reduced to the size of the convolution while simultaneously increasing the inference latency to 96×96-times the duration it takes to process a single layer.

This thesis attempted to perform as many MACs in parallel as possible on the embedded platform. The limiting factors are the available amount of LUTs (and DSP blocks) and the PL speed grade. The speed grade determines the latency of signals between connected FFs. If too many FFs are connected in parallel to subsequent FFs, the VHDL synthesis tool cannot guarantee the simultaneous arrival of clocked signals. Thus, signal integrity and correct functionality of the implemented logic are not provided.

Three different (working) hardware designs, with differing amounts of parallelization, were implemented to compare latency reduction to resource utilization. Figure 4.8 shows a flow diagram of hardware design A, which features the least amount of parallelization. Still, this design already offers 32 parallel MAC operations in the first layer alone, which enables the evaluation of the complete first convolution kernel in a few clock cycles. Post-processing is performed in a pipelined sequence in very few clock cycles as well. It is impossible to tell the exact amount of cycles required in the first layer since this part of the accelerator still depends on data transmission from PS, which is not deterministic.

A single output on all sixteen output channels of layer one is written into its PL memory in a single clock cycle through a 128-bit wide bus. In the same fashion, layer 2 can collect a whole block of $(3 \times 3 \times 16)$ input values from this memory in a single clock cycle.

Since layer 2 is the most complex layer in terms of the amount of MAC operations, the different hardware designs feature varying amounts of parallel instances of layer 2. Each instance processes a subset of the eight output feature maps. With a single instance of layer 2, Hardware A (Figure 4.8) can process two output maps at once and thus has to iterate four times to infer a complete point on all eight output feature maps. Hardware B (Figure 4.9) decreases inference latency by offering two parallel instances of layer 2, and Hardware C (Figure 4.10) provides four parallel instances obtaining the lowest latency.

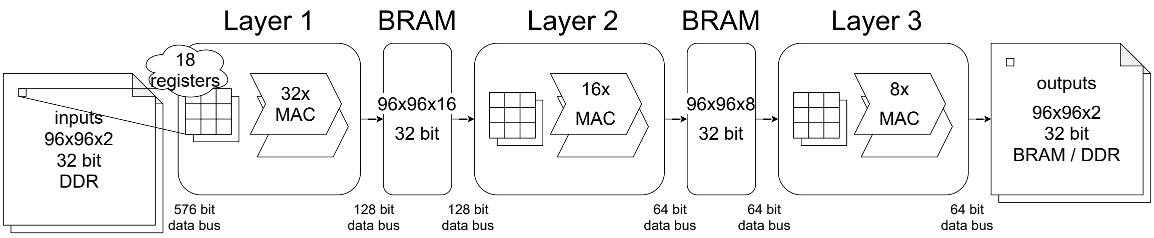


Figure 4.8: Flow diagram of the hardware accelerator. Single instance of layer 2: Design A.

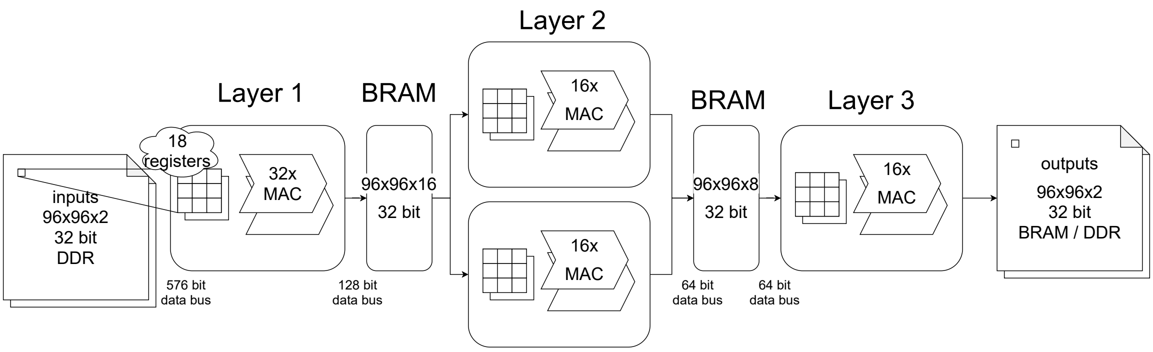


Figure 4.9: Flow diagram of the hardware accelerator with additional parallelization. Two instances of layer 2: Design B.

The number of required clock cycles in order to infer a complete RD map are provided in Table 4.2. These values were measured on the individual layers in simulations from the HDL simulator of *Vivado*. Total processing time means the minimum amount of

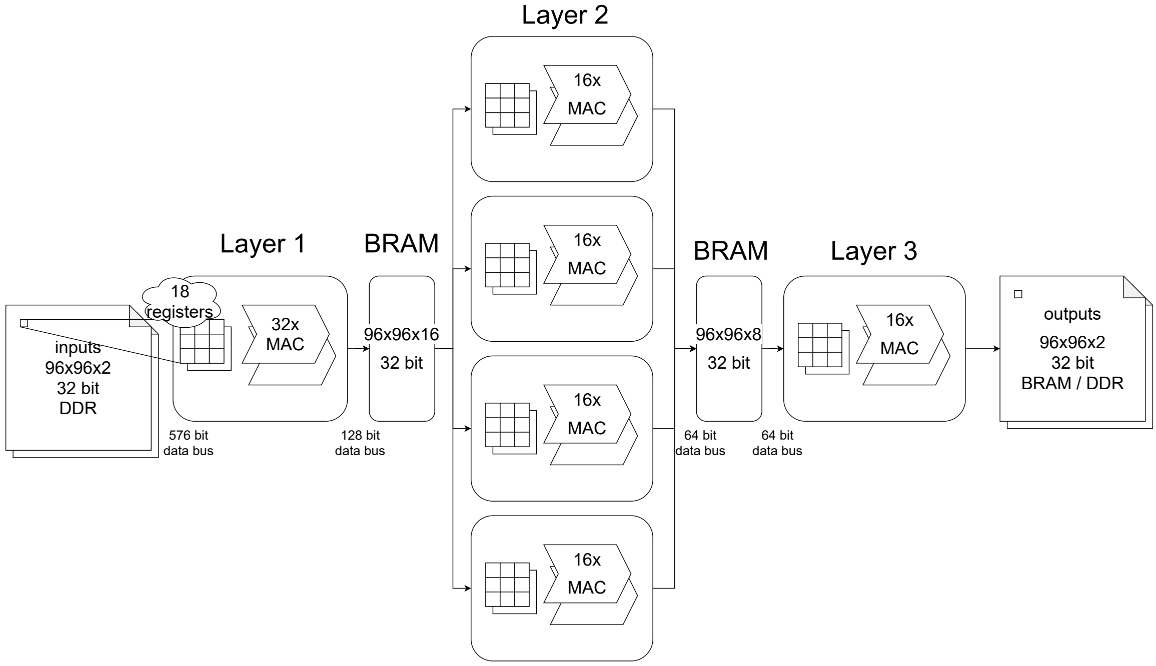


Figure 4.10: Flow diagram of the hardware accelerator with additional parallelization. Four instances of layer 2: Design C.

clock cycles with a 100 MHz PL clock. Note, that this is the theoretical processing time; the actual processing time is listed in Section 5.3. The table shows, that the number of clock cycles decreases significantly with the parallelization rate in layer 2. Not counting the transfer from PS to PL and back, the minimum latency for a complete RD map inference is 6.96 ms for hardware C. As expected, hardware A yields the highest latency with 23.63 ms.

	Layer 1	**Layer 2**	**Layer 3**	**PL to PS**	**Total Clocks**	**Total Time [ms]**
Hardware A	>267265	1723393	350212	>21832	**2362702**	**23.63**
Hardware B	>267265	949249	350212	>21832	**1588558**	**15.89**
Hardware C	>267265	56277	350212	>21832	**695586**	**6.96**

Table 4.2: Required clock cycles for single frame inference with different amounts of parallelization. Hardware A being the least amount of parallelization (1 instance of layer 2), Hardware B with two instances of layer 2 and Hardware C being the highest amount of parallelization with four instances of layer 2. Total time is calculated for PL clocked at 100 MHz

4.3.4 Resource Utilization of Hardware Accelerator

To compare the resource utilization of the three hardware designs, the *Vivado* HDL synthesis tool offers resource utilization reports with various SoC parameters. Furthermore, as shown in Section 3.2.1, a decreased bit width for input and output values might also result in reasonably accurate CNN performance, so the hardware designs A, B and C were expanded to three different input/output bit widths as well. Figure 4.11 is a direct comparison of all parameters as percentages of available resources on the specific platform. A detailed list can be found in the figures and corresponding tables in Appendix A.

Hardware C has the highest resource requirements. Decreasing bit widths reduce resource utilization. However, it has to be noted that the implementations with 8-bit inputs and 16-bit outputs make the least usage of embedded DSP blocks and rather requires a higher

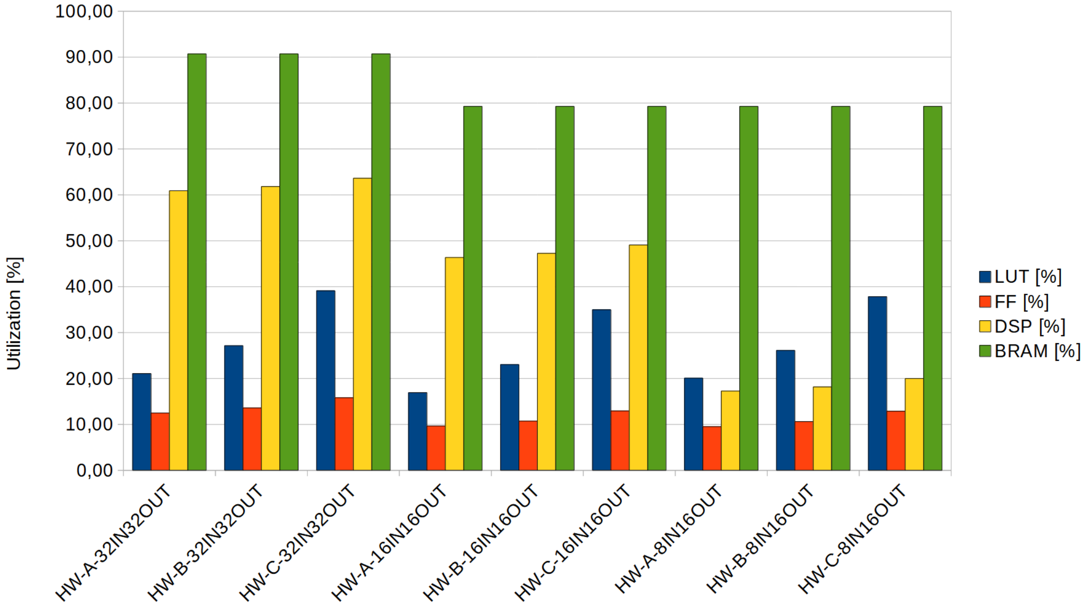


Figure 4.11: Resource utilization for all hardware accelerator implementations. HW-A, HW-B and HW-C stand for the specific hardware design (A: 1 instance layer 2, B: 2 instances layer 2, C: 4 instance layer 2). The resource utilization increases from A to C, since the amount of parallelization is increased as well. xINxOUT means the bit width of the input and output values in signed fixed-point format (32IN32OUT: 32-bit inputs and 32-bit outputs, 16IN16OUT: 16-bit inputs and 16-bit outputs, 8IN16OUT: 8-bit inputs and 16-bit outputs). Resource utilization increases with bit width.

degree of LUTs, still with the same requirements of BRAM instances. This can be explained by the design of DSP blocks, which are not optimized for 8-bit integers and thus not infered by *Vivado* in this configuration. Likewise, BRAMs can only be inferred in instances of a set of specified sizes, so smaller bit width numbers might not require less BRAM, depending on optimizer settings.

4.3.5 Graphical User Interface

The graphical user interface is a feature that the *Vivado* block designer offers. In the block designer, a developer can connect various computational blocks in a graphical environment. By offering configurable accelerator parameters in this Graphical User Interface (GUI), the developer can alter the behavior without VHDL code manipulation.

Figure 4.12 shows the first page of the GUI that was developed in this thesis to accompany the hardware accelerator. It is possible to change scale values, bit widths, and even the amount of parallelization besides many other parameters. All these values are hard-coded in the hardware configuration file in the *Xilinx Vivado* implementation run.

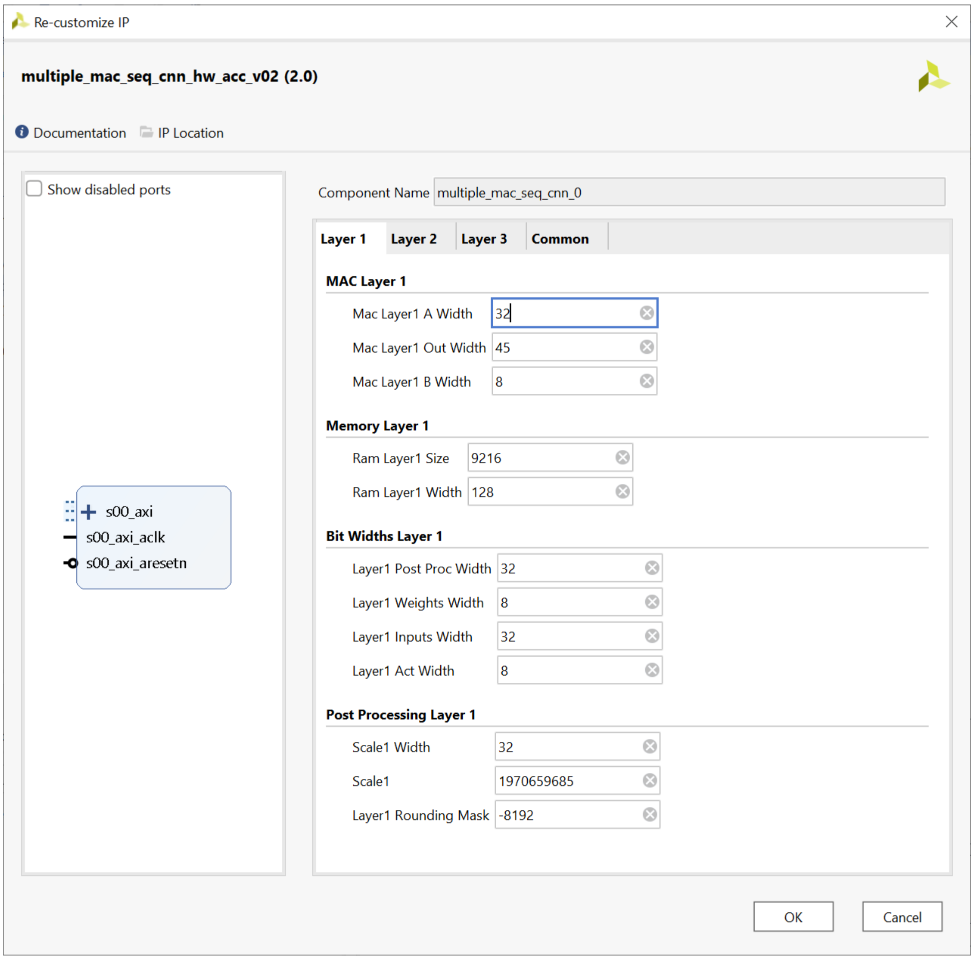


Figure 4.12: Graphical Interface for the block design integration in Xilinx Vivado.

5 Evaluation Tools and Results

This chapter will explain the evaluation tools used, how they are used, and the evaluation results for different implementations. Section 5.3 summarizes all results and shows the processing latencies for hardware and firmware implementations of the CNN. Furthermore, the power consumption of the embedded system is presented.

5.1 Performance Comparison

A custom Python script that uses the CA-CFAR object detection algorithm and subsequent F1-score evaluation compares the accuracy of various CNN implementations. In principle, this script performs the same steps for F1-score evaluation as in the original CNN implementation [7] but uses the fully quantized CNN software implementation from this thesis (see Section 3.2) as the CNN model. Since the fully quantized implementation works with atomic MAC operations instead of optimized convolution functions from a Python module, and emulates number formats by artificially truncating value ranges, it is extremely slow and the evaluation of the F1-score can take several hours, depending on the processing hardware.

F1-Score: The F1-Score is defined as the harmonic mean of precision p and recall r by $F_1 = 2\frac{p \cdot r}{p+r}$ (Figure 5.1). The precision p considers the number of false alarm object detections, and the recall r contains the number of correctly identified object peaks [7]. The ground truth target detections were manually labeled in clean measurement RD maps [7]. The CA-CFAR target detection algorithm [43] automatically yields peak locations from the interference mitigated RD maps.

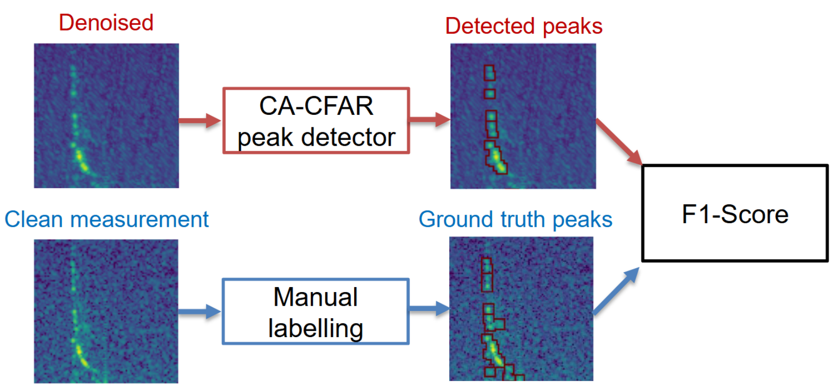


Figure 5.1: Evaluation of F1-Score. (Source: [47])

5.2 Result Visualization

An additional Python script was written to visualize radar samples before and after CNN processing. This script also works as a radar simulator, by loading radar measurements from the test data set and transmitting these to the SoC, essentially acting as radar device to the CNN processor. This was needed during development of the embedded network in order to enable fundamental debugging.

Simulation: The radar simulation script uses the Python *socket* library to act as a TCP/IP client and connect to the TCP/IP server on the SoC [48]. It then sends a configurable amount of RD maps to the embedded CNN processor and waits for the data to return after every sample. This way, the script can also perform timing measurements that include transfer speed as well as processing latency. By subtracting the actual processing latency that was measured in *Vitis*, the pure transfer duration can be extracted (as can be seen in Section 5.3).

When all processed samples have been received, the Python script starts to display all samples as magnitude images in consecutive order, thus enabling a first visual check of the CNN performance.

Real-Time Measurement: For real-time radar interference mitigation, an additional script is used for real-time radar data acquisition. *INRAS* provides the script *AN77_06.py* in the 6^{th} version of their Python framework. This script automatically reads radar measurement data from a connected *RadarLog* and computes the corresponding RD maps. Measurement data can then be sent to the embedded CNN for interference mitigation, using the mentioned radar board simulation script. Details on the *AN77_06.py* script can be found in the *INRAS* application note *IFX-TX2RX16-AN77-06 Range-Doppler Processing*[49].

5.3 Timing, Resource Utilization and Power Requirement Results for all Implementations

Figure 5.2 shows the four different CNN implementations compared in regards to frame duration and resource utilization. The duration time is the pure processing time measured in *Vitis* as mentioned in Section 4.2. Resource utilization is extracted from *Vivado* after an implementation run. The firmware implementation in this diagram is the fastest implementation, using fixed-point number format and maximum compiler optimization. The hardware designs A, B and C in this diagram are the hardware designs mentioned in Section 4.3.4 with different amounts of parallelization in layer 2. Altering the bit width of inputs and outputs does not decrease the number of clock cycles needed to infer outputs in a hardware implementation, so these variations are not considered in the diagram. As can be seen, the firmware implementation has by far the least resource requirements, while also displaying the longest processing duration of almost 350 ms. The resource requirements of the hardware designs increase with bit width, as expected, while outperforming the firmware implementation by almost one order of magnitude.

Power consumptions of the four different CNN implementations are shown in Table 5.1. The difference does not seem significant, but as expected, the hardware implementation with the highest ratio of parallelization also shows the highest power consumption.

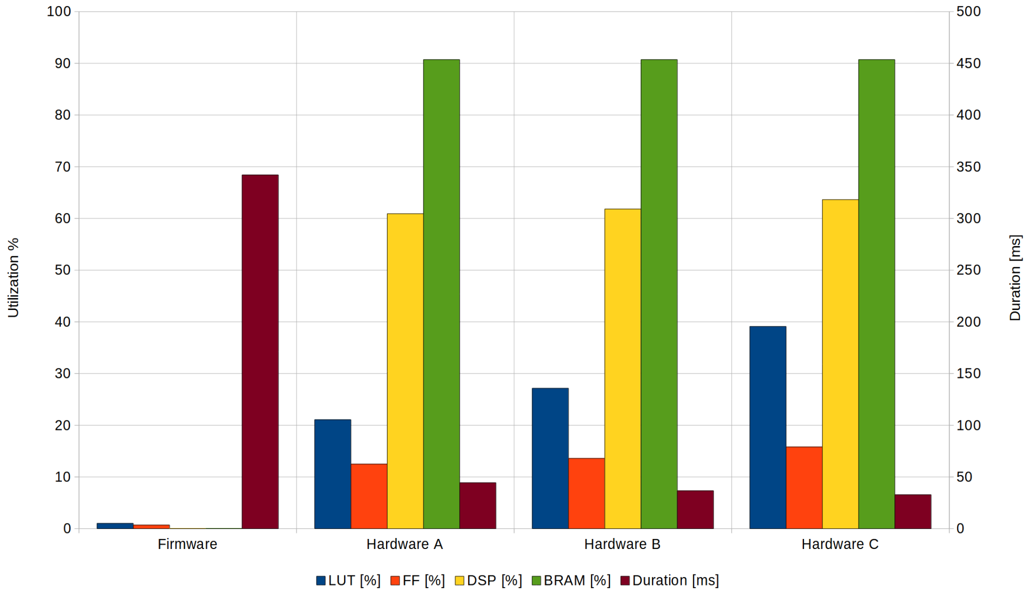


Figure 5.2: Comparison of resource utilization and timing for hardware and firmware implementations. Hardware A: Single instance of layer 2; Hardware B: Two instances of layer 2; Hardware C: Four instances of layer 2; Firmware: Firmware implementation with optimization and best timing performance. All HW and SW implementations work with 32-bit input/output values. FW is integer implementation.

	Design Dynamic Power Consumption
Firmware	1.262 W
Hardware A	1.536 W
Hardware B	1.570 W
Hardware C	1.632 W

Table 5.1: Power consumption for all implementations. The design dynamic power consumption means power that is dissipated by the specific design and not general dissipation of the device.

Timings for the complete list of CNN implementations can be found in Table 5.2. The first column labeled ”Frame“ is the timing measured in the Python script mentioned in Section 5.2, so it contains transfer and processing durations. The second column is the pure processing duration measured in *Vitis*, and column three is the difference of the first two columns. It was expected that this difference is the same for every implementation, since the data transfer rate should not change with the processing paradigm. A possible explanation is that ethernet transfer is not further accelerated by compiler optimization and die optimizer favors processing throughput rather than ethernet transfer. Processing optimization in the PS might also have an effect on the cache and instruction pipelines inside the ARM processing core, which could lead to decreased ethernet transfer rates. Table 5.2 contains a variety of optimization configurations for the C compiler in *Vitis*, such as *-o0* meaning no optimization, *-o1* the least, and *-o3* the highest degree of compiler optimization. It is quite interesting, that NEON coprocessor code optimization (*/w NEON*) does not show any signs of timing improvement. Note: While code optimization for the NEON coprocessor was enabled for these configurations, it was relied on the automatic optimization process. It could be possible that the firmware code was not suited for automatic NEON optimization. The floating-point firmware implementation has by far the slowest overall timing figures, with, and without compiler optimization enabled.

Figure 5.3 is a comparison of all hardware implementations regarding resource utiliza-

	Frame [ms]	Processing [ms]	Difference [ms]
No CNN	39	N/A	N/A
Firmware FI -o0	2000	1937.33	62.67
Firmware FI -o1	750	693.48	56.52
Firmware FI -o2	730	648.50	81.50
Firmware FI -o3	500	342.03	157.97
Firmware FI -o0 /w NEON	2000	1936.20	63.80
Firmware FI -o1 /w NEON	750	689.76	60.24
Firmware FI -o2 /w NEON	730	648.14	81.86
Firmware FI -o3 /w NEON	500	342.07	157.93
Firmware FP -o0	2250	2188.34	61.66
Firmware FP -o1	1140	1070.17	69.83
Firmware FP -o2	1100	1027.81	72.19
Firmware FP -o3	900	772.10	127.90
Firmware FP -o0 /w NEON	2250	2187.77	62.23
Firmware FP -o1 /w NEON	1140	1070.27	69.73
Firmware FP -o2 /w NEON	1100	1030.88	69.12
Firmware FP -o3 /w NEON	880	775.96	104.04
Hardware A	84	44.40	39.60
Hardware B	75	36.66	38.34
Hardware C	71	32.80	38.20

Table 5.2: Comparison of timings for all implementations. Frame Duration means the measurement in Python for the transmission and reception of a whole RD map frame. Processing Duration is the measurement of CNN processing in Xilinx Vitis. The difference is the time the CPU spends with data transmission. FI: fixed-point; FP: floating-point; -o0: no optimization; -o1: little optimization; -o2: medium optimization; -o3: high optimization; /w NEON: using the NEON coprocessor; Hardware A: hardware accelerator with least parallelization; Hardware B: hardware accelerator with medium parallelization; Hardware C: hardware accelerator with high parallelization

tion and achieved F1-score. In this diagram, bit widths for input and output values are mentioned, because the bit width has displayed a strong impact on the accuracy of the network in Section 3.2.1. Resource utilization is indicated as a mean value over all resource parameters, in order to provide an appealing visual presentation.

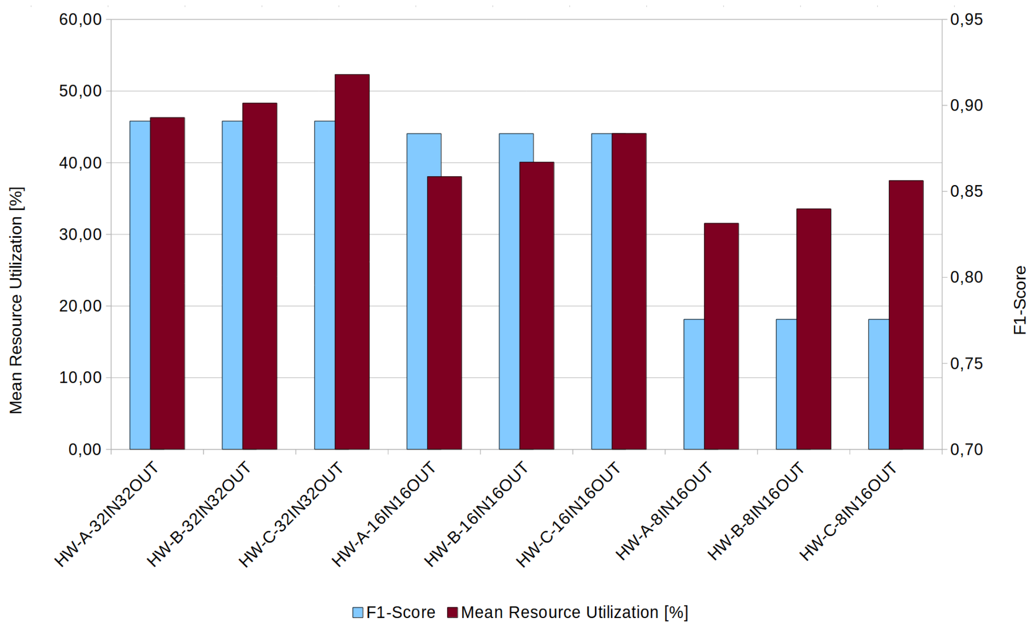


Figure 5.3: Comparison of resource utilization and F1-Score for all hardware implementations with different bit widths. Resource utilization is presented as a mean value of all resource parameters.

6 Conclusion

A resource-efficient implementation of a CNN-based system for radar interference mitigation was further optimized and deployed on embedded hardware for real-time application. Several different variations of the same CNN were implemented for this purpose and compared regarding resource utilization on the embedded platform, processing duration, accuracy, and power consumption. As embedded platform, a Zynq-7000 SoC from *Xilinx* with a dual-core Cortex-A9 processor was chosen. The *Digilent* Arty Z7-20 board was used as prototyping board, which hosts this device among other peripherals, such as an ethernet interface (see Section 2.3.3). The investigated CNN implementations were split into two distinct groups: The firmware implementation that is executed on the CPU of the processing system, and the hardware implementation, which consists of a hardware accelerator in the PL of the SoC and the firmware that uses this accelerator for faster CNN inference (see Section 4.1).

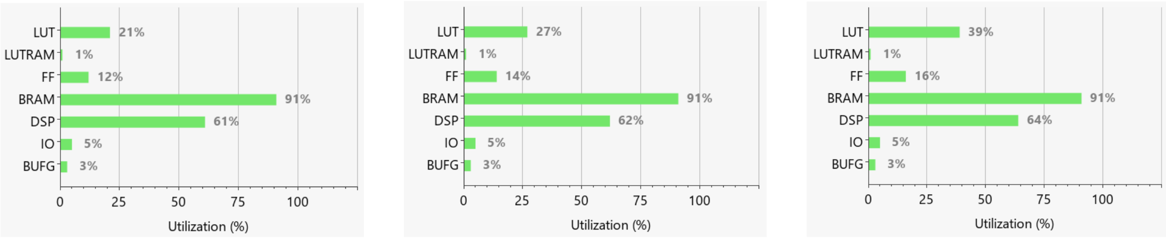
We showed in Section 5.3 that any firmware implementation cannot compete with any hardware implementation concerning timing. Even with high optimization settings and further coprocessor assistance, the processing duration is still magnitudes slower compared to the hardware accelerator assisted designs. The processing of one RD map might require up to 340 ms, which increases the cycle duration substantially for ADAS hardware. Therefore it is not advised to implement CNN based radar interference mitigation on a purely Advanced RISC Machines (ARM)-based embedded device.

Even if a great portion of CNN parameters is converted to a fixed-point representation, which introduces additional quantization noise on data and all computations, the fully quantized hardware implementation did not show a significant drop in accuracy for a 32-bit design. Compared to the software reference design with a F1-score of 0.8922, the 32-bit fixed-point hardware accelerator indicated only a slight drop to 0.8909 (see Figure 3.8), while reducing processing duration significantly to 34 ms per RD map in the case of a hardware design with a high rate of computational parallelization (see Figure 5.2). Naturally, a high parallelization ratio comes with proportionally high requirements on resource utilization on FPGA-based SoCs, but even an accelerator design with relatively low parallelization and resource utilization shows a drop in processing duration to at least 44 ms. Therefore it is recommended to use FPGA-based embedded hardware for CNN-based radar interference mitigation to achieve fast latency while retaining implementation flexibility.

Moreover, for severely resource constrained SoCs, a hardware design with little parallelization and 16-bit fixed-point inputs and outputs is a feasible solution. Compared to a 32-bit design with high parallelization that exhibits a resource utilization of more than 50 % on the designated platform, a 16-bit design with little parallelization decreases resource utilization to less than 40 % (see Figure 5.3) while retaining a relatively high F1-score of 0.8836 (see Figure 3.8).

Appendix A: Resource Utilization of all CNN Hardware Accelerator Implementations

Resource utilization for all hardware implementations are shown in the following diagrams and tables. For 32-bit quantized inputs and outputs, Figures 1 (a), (b), (c) show the visual resource utilization output of *Xilinx Vivado*. Tables 1, 2 and 3 show a detailed listing of the diagram values. Figures 2 (a), (b), (c), and Tables 4, 5 and 6 correspond to data for 16-bit inputs and outputs. Figures 3 (a), (b), (c), and Tables 7, 8 and 9 correspond to data for 8-bit inputs and 16-bit outputs.



(a) HW-A-32IN32OUT (b) HW-B-32IN32OUT (c) HW-C-32IN32OUT

Figure 1: Resource utilization of the three different implementations of the CNN accelerator in hardware using different amounts of parallelization in the computation. Hardware A (HW-A-32IN32OUT): single instance of layer 2; Hardware B (HW-B-32IN32OUT): two instances of layer 2; Hardware C (HW-C-32IN32OUT): four instance of layer 2. 32-bit inputs and 32-bit signed fixed-point outputs for all the above.

Resource	Utilization	Available	Utilization %
LUT	11211	53200	21.07
LUTRAM	74	17400	0.42
FF	13285	106400	12.48
BRAM	127	140	90.71
DSP	134	220	60.90
IO	6	125	4.80
BUFG	1	32	3.12

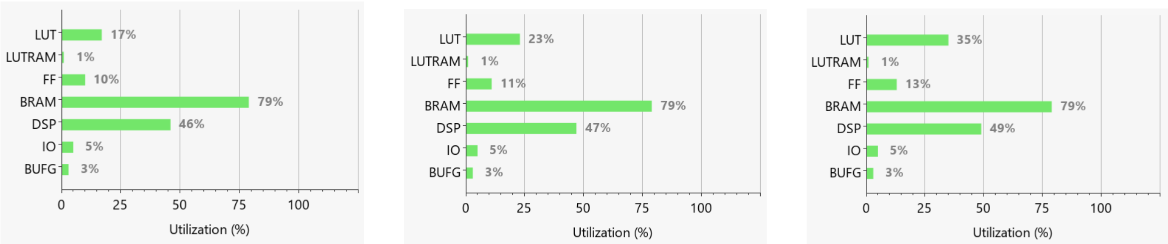
Table 1: Resource utilization output of Xilinx Vivado IDE for Hardware Design A, 32-bit inputs and 32-bit outputs (HW-A-32IN32OUT).

Resource	Utilization	Available	Utilization %
LUT	14444	53200	27.15
LUTRAM	74	17400	0.42
FF	14462	106400	13.59
BRAM	127	140	90.71
DSP	136	220	61.81
IO	6	125	4.80
BUFG	1	32	3.12

Table 2: Resource utilization output of Xilinx Vivado IDE for Hardware Design B, 32-bit inputs and 32-bit outputs (HW-B-32IN32OUT).

Resource	Utilization	Available	Utilization %
LUT	20809	53200	39.11
LUTRAM	74	17400	0.42
FF	16821	106400	15.80
BRAM	127	140	90.71
DSP	140	220	63.63
IO	6	125	4.80
BUFG	1	32	3.12

Table 3: Resource utilization output of Xilinx Vivado IDE for Hardware Design C, 32-bit inputs and 32-bit outputs (HW-C-32IN32OUT).



(a) HW-A-16IN16OUT (b) HW-B-16IN16OUT (c) HW-C-16IN16OUT

Figure 2: Resource utilization of the three different implementations of the CNN accelerator in hardware using different amounts of parallelization in the computation. Hardware A (HW-A-16IN16OUT): single instance of layer 2; Hardware B (HW-B-16IN16OUT): two instances of layer 2; Hardware C (HW-C-16IN16OUT): four instance of layer 2. 16-bit inputs and 16-bit signed fixed-point outputs for all the above.

Resource	Utilization	Available	Utilization %
LUT	9010	53200	16.93
LUTRAM	74	17400	0.42
FF	10278	106400	9.65
BRAM	111	140	79.28
DSP	102	220	46.36
IO	6	125	4.80
BUFG	1	32	3.12

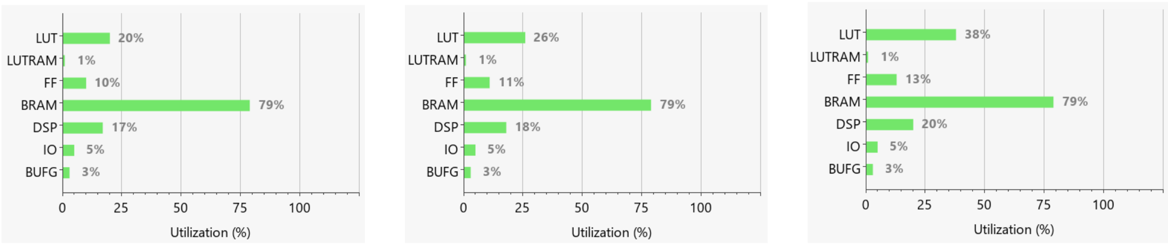
Table 4: Resource utilization output of Xilinx Vivado IDE for Hardware Design A, 16-bit inputs and 16-bit outputs (HW-A-16IN16OUT).

Resource	Utilization	Available	Utilization %
LUT	12257	53200	23.03
LUTRAM	74	17400	0.42
FF	11411	106400	10.72
BRAM	111	140	79.28
DSP	104	220	47.27
IO	6	125	4.80
BUFG	1	32	3.12

Table 5: Resource utilization output of Xilinx Vivado IDE for Hardware Design B, 16-bit inputs and 16-bit outputs (HW-B-16IN16OUT).

Resource	Utilization	Available	Utilization %
LUT	18610	53200	34.98
LUTRAM	74	17400	0.42
FF	13774	106400	12.94
BRAM	111	140	79.28
DSP	108	220	49.09
IO	6	125	4.80
BUFG	1	32	3.12

Table 6: Resource utilization output of Xilinx Vivado IDE for Hardware Design C, 16-bit inputs and 16-bit outputs (HW-C-16IN16OUT).



(a) HW-A-8IN16OUT (b) HW-B-8IN16OUT (c) HW-C-8IN16OUT

Figure 3: Resource utilization of the three different implementations of the CNN accelerator in hardware using different amounts of parallelization in the computation. Hardware A (HW-A-8IN16OUT): single instance of layer 2; Hardware B (HW-B-8IN16OUT): two instances of layer 2; Hardware C (HW-C-8IN16OUT): four instance of layer 2. 8-bit inputs and 16-bit signed fixed-point outputs for all the above.

Resource	Utilization	Available	Utilization %
LUT	10684	53200	20.08
LUTRAM	74	17400	0.42
FF	10122	106400	9.51
BRAM	111	140	79.28
DSP	38	220	17.27
IO	6	125	4.80
BUFG	1	32	3.12

Table 7: Resource utilization output of Xilinx Vivado IDE for Hardware Design A, 8-bit inputs and 16-bit outputs (HW-A-8IN16OUT).

Resource	Utilization	Available	Utilization %
LUT	13899	53200	26.12
LUTRAM	74	17400	0.42
FF	11300	106400	10.62
BRAM	111	140	79.28
DSP	40	220	18.18
IO	6	125	4.80
BUFG	1	32	3.12

Table 8: Resource utilization output of Xilinx Vivado IDE for Hardware Design B, 8-bit inputs and 16-bit outputs (HW-B-8IN16OUT).

Resource	**Utilization**	**Available**	**Utilization %**
LUT	20128	53200	37.83
LUTRAM	74	17400	0.42
FF	13695	106400	12.87
BRAM	111	140	79.28
DSP	44	220	20.00
IO	6	125	4.80
BUFG	1	32	3.12

Table 9: Resource utilization output of Xilinx Vivado IDE for Hardware Design C, 8-bit inputs and 16-bit outputs (HW-C-8IN16OUT).

Bibliography

[1] S. Alland, W. Stark, M. Ali, and M. Hegde, "Interference in automotive radar systems," *IEEE SIGNAL PROCESSING MAGAZINE*, Sep. 2019.

[2] C. Fischer, "Untersuchungen zum interferenzverhalten automobiler radarsensorik," Ph.D. dissertation, Universität Ulm, 2016.

[3] J. Bechter, K. D. Biswas, and C. Waldschmidt, "Estimation and cancellation of interferences in automotive radar signals," in *2017 18th International Radar Symposium (IRS)*, 2017, pp. 1–10. DOI: `10.23919/IRS.2017.8008126`.

[4] J. Rock, M. Toth, P. Meissner, and F. Pernkopf, "Deep interference mitigation and denoising of real-world fmcw radar signals," *IEEE*, Jan. 2020.

[5] *Long-range-radar lrr3: Radar sensor for railway applications*, Bosch Engineering GmbH, 2010.

[6] B. Betkaoui, D. Thomas, and W. Luk, "Comparing performance and energy efficiency of fpgas and gpus for high productivity computing," *IEEE*, 2010.

[7] J. Rock, W. Roth, M. Toth, P. Meissner, and F. Pernkopf, "Resource-efficient deep neural networks for automotive radar interference mitigation," *IEEE JOURNAL OF SELECTED TOPICS IN SIGNAL PROCESSING*, vol. 15, no. 4, Jun. 2021.

[8] INRAS. "RadarLog." (Aug. 1, 2011), [Online]. Available: `http://www.inras.at/en/products/radarlog.html` (visited on 08/01/2021).

[9] G. Rudolph and U. Voelzke. "Three Sensor Types Drive Autonomous Vehicles." (Nov. 10, 2017), [Online]. Available: `https://www.fierceelectronics.com/components/three-sensor-types-drive-autonomous-vehicles` (visited on 09/14/2021).

[10] V. Bhaskar and K. Joshi, "Basic radar system for automotive adas," *PathPartner Technology Pvt. Ltd.*, 2017.

[11] A. Kondepaddy. "FMCW CHIRP configurations for SRR, MRR and LRR." (Aug. 27, 2018), [Online]. Available: `https://www.pathpartnertech.com/fmcw-chirp-configurations-for-srr-mrr-and-lrr/` (visited on 09/14/2021).

[12] H. Meinel and J. Dickmann, "Automotive radar: From its origin to future directions," *Microwave Journal*, vol. vol.56, pp. 24–40, Sep. 2013.

[13] H. H. Meinel, "Evolving automotive radar — from the very beginnings into the future," in *The 8th European Conference on Antennas and Propagation (EuCAP 2014)*, 2014, pp. 3107–3114. DOI: `10.1109/EuCAP.2014.6902486`.

[14] Infineon. "Bosch to Use Radar Chip from Infineon." (Dec. 1, 2008), [Online]. Available: `https://www.infineon.com/cms/en/about-infineon/press/press-releases/2008/INFATV200812-015.html` (visited on 09/11/2021).

[15] L. Teschler. "The basics of automotive radar." (Nov. 6, 2019), [Online]. Available: `https://www.designworldonline.com/the-basics-of-automotive-radar/` (visited on 09/01/2021).

[16] T. Yuan, N. Yuan, and L.-W. Li, "A novel series-fed taper antenna array design," *IEEE Antennas and Wireless Propagation Letters*, vol. 7, pp. 362–365, 2008. DOI: `10.1109/LAWP.2008.928487`.

[17] *Bgt24mtr12 silicon germanium 24 ghz transceiver mmic data sheet*, Infineon, Jul. 2014.

[18] E. Kolmhofer, "From research to industry: Highly integrated mm-wave transceiver for automotive radar applications," DICE GmbH, THz-Workshop: Millimeter- and Sub-Millimeter-Wave circuit design and characterization, Sep. 2014.

[19] H. Winner, S. Hakuli, F. Lotz, and C. Singer, *Handbuch Fahrerassistenzsysteme - Grundlagen, Komponenten und Systeme für aktive Sicherheit und Komfort*, third, ser. ATZ/MTZ-Fachbuch. Springer Vieweg, 2015.

[20] C. Iovescu and S. Rao. "The fundamentals of millimeter wave radar sensors." (2021), [Online]. Available: https://www.ti.com/lit/wp/spyy005a/spyy005a.pdf?ts=1620999365997&ref_url=https%253A%252F%252Fwww.ti.com%252Fsensors%252Fmmwave-radar%252Findustrial%252Ftechnical-documents.html (visited on 09/17/2021).

[21] A. Haderer, *Rdl-77g-tx2rx16 frontend (user manual)*, INRAS GmbH., Altenbergerstraße 69, 4040 Linz, Austria, Dec. 2016.

[22] *Afe5801 - 8-channel variable-gain amplifier (vga) with octal high-speed adc*, Texas Instruments, May 2010.

[23] ——, *Radarlog and 77-ghz frontend (getting started)*, INRAS GmbH., Altenbergerstraße 69, 4040 Linz, Austria, Feb. 2018.

[24] T. Wood. "What is an Activation Function?" (Nov. 6, 2021), [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/activation-function (visited on 11/06/2021).

[25] F. Pernkopf and C. Knoll, *Computational intelligence course notes*, (unpublished) Technical University of Graz, Austria, Jul. 2020.

[26] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[27] E. Georga, D. Fotiadis, and S. Tigas, *Personalized Predictive Modeling in Type 1 Diabetes*, first. Academic Press, Nov. 2017.

[28] W. Zhang, K. Itoh, J. Tanida, and Y. Ichioka, "Parallel distributed processing model with local space-invariant interconnections and its optical architecture," *APPLIED OPTICS*, vol. 29, no. 32, Nov. 1990.

[29] A. Karpathy. "Stanford University CS231n: Convolutional Neural Networks for Visual Recognition." (Aug. 16, 2021), [Online]. Available: https://cs231n.github.io/ (visited on 08/16/2021).

[30] A. Géron, *Hands-on Machine Learning with Scikit-Learn, Keras TensorFlow*, second. O'Reilly Media, Inc., Sep. 2019.

[31] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015, pp. 448–456. [Online]. Available: http://jmlr.org/proceedings/papers/v37/ioffe15.pdf.

[32] H. Askary. "Intuitive Explanation of Straight-Through Estimators with PyTorch Implementation." (Sep. 19, 2020), [Online]. Available: https://www.hassanaskary.com/python/pytorch/deep%20learning/2020/09/19/intuitive-explanation-of-straight-through-estimators.html (visited on 09/24/2021).

[33] Xilinx. "What is an FPGA?" (Aug. 20, 2018), [Online]. Available: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/gac1504034293050.html (visited on 08/28/2021).

[34] ——, "Understanding FPGA Architecture." (Aug. 20, 2018), [Online]. Available: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/odz1504034293215.html (visited on 08/28/2021).

[35] ——, "LUT." (Aug. 20, 2018), [Online]. Available: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/yeo1504034293627.html (visited on 08/28/2021).

[36] ——, "Flip Flop." (Aug. 20, 2018), [Online]. Available: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/ksg1504034293914.html (visited on 08/28/2021).

[37] ——, "DSP48 Block." (Aug. 20, 2018), [Online]. Available: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/uwa1504034294196.html (visited on 08/28/2021).

[38] ——, "BRAM and Other Memories." (Aug. 20, 2018), [Online]. Available: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/jbt1504034294480.html (visited on 08/28/2021).

[39] T. Han, G. W. Liu, H. Cai, and B. Wang, "The face detection and location system based on zynq," in *2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2014, pp. 835–839. DOI: 10.1109/FSKD.2014.6980946.

[40] M. Sadri, C. Weis, N. Wehn, and L. Benini, "Energy and performance exploration of accelerator coherency port using xilinx zynq," *FPGAWorld '13*, Sep. 2013.

[41] R. Melo, B. Valinoti, M. B. Amador, L. García, A. Cicuttin, and M. L. Crespo, "Study of the data exchange between pl and ps of zynq-7000 devices," Instituto Nacional de Tecnologia Industrial, presentation, Apr. 2019.

[42] D. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," *ICLR*, 2015.

[43] L. Scharf and C. Demeure, *Statistical Signal Processing: Detection, Estimation, and Time Series Analysis*, ser. Addison-Wesley Series in Electrical and Computer Engineering. Addison-Wesley Publishing Company, 1991.

[44] G. Technology. "Bit growth in FPGA arithmetic." (Jul. 21, 2017), [Online]. Available: https://zipcpu.com/dsp/2017/07/21/bit-growth.html (visited on 10/31/2021).

[45] A. Kroh and O. Diessel, "A short-transfer model for tightly-coupled cpu-fpga platforms," *International Conference on Field-Programmable Technology (FPT)*, 2018.

[46] A. Dunkels. "Lightweight IP stack." (Oct. 21, 2021), [Online]. Available: https://www.nongnu.org/lwip/2_1_x/index.html (visited on 10/11/2021).

[47] A. Fuchs, J. Rock, M. Toth, P. Meissner, and F. Pernkopf, "Complex-valued convolutional neural networks for enhanced radar signal denoising and interference mitigation," Signal Processing and Speech Communication Laboratory, University of Technology Graz, Austria, presentation, Apr. 2021.

[48] "socket — Low-level networking interface." (Oct. 21, 2021), [Online]. Available: https://docs.python.org/3/library/socket.html (visited on 10/11/2021).

[49] A. Haderer, *Ifx-tx2rx16-an77-06 range-doppler processing*, INRAS GmbH., Altenbergerstraße 69, 4040 Linz, Austria, Apr. 2017.

List of Figures

List of Tables

Acronyms

ACP Accelerator Coherency Port. 26

ADAS Advanced Driver-Assistance System. 9, 13, 59

ADC Analog-to-Digital Converter. 14, 15

ALU Arithmetic Logic Unit. 25

ANN Artificial Neural Network. 19, 20

APU Application Processing Unit. 25

ARM Advanced RISC Machines. 59

AXI Advanced eXtensible Interface. 25, 26, 39, 44

BRAM Random-Access-Memory Block. 25, 27, 44, 46, 50

CA-CFAR Cell-Averaging Constant False Alarm Rate. 33, 53

CLB Configurable Logic Blocks. 24

CNN Convolutional Neural Network. LIX–LXI, 9–11, 13, 19–24, 29–37, 39, 41, 43–47, 49, 53–56, 59

CPU Central Processing Unit. 11, 12, 24–26, 33, 39–41, 44–47, 56, 59

CS Chirp Sequence. 9, 14

DDR Double Data Rate. 18, 27, 40, 41, 46

DFT Discrete Fourier Transform. 29

DMA Direct Memory Access. 25, 46

DNN Deep Neural Network. 20, 23

DSP Digital Signal Processing. 25, 44, 48–50

EMAC Ethernet Media Access Control. 40

EMIO Extended Multiplexed Input/Output. 25

FF Flip-Flop. 24, 27, 40, 44, 48

FFT Fast Fourier Transform. 9, 16

FMCW Frequency-Modulated Continuous-Wave. 9, 10, 14, 15, 17, 29, 30

FoV Field of View. 13

FPGA Field-Programmable Gate Array. 9–13, 18, 24, 25, 33, 59

GEM Gigabit Ethernet Controller. 40

GIC General Interrupt Controller. 25

GP General-Purpose. 25, 40, 44

GPIO General-Purpose Input/Output. 39

GPU Graphics Processing Unit. 9, 24

GUI Graphical User Interface. 50

HDL Hardware Description Language. 11, 25, 28, 48, 49
HP High Performance. 26

I/O Input/Output. 24, 44
IC Integrated Circuit. 15, 18, 24
IF Intermediate Frequency. 14–16, 18

LIDAR Light Detection And Ranging. 13
LNA Low-Noise Amplifier. 15
LO Local Oscillator. 15
LRR Long-Range Radar. 13, 14
LUT Look-Up Table. 24, 25, 27, 44, 46, 48, 50

MAC Multiply-And-Accumulate operation. 9, 11, 22, 25, 33, 34, 37, 38, 43, 47, 48, 53
MCU Microcontroller Unit. 24
MIMO Multiple-In-Multiple-Out. 17
MIO Multiplexed Input/Output. 40
MMIC Monolithic Microwave Integrated Circuit. 15, 17
MRR Mid-Range Radar. 13
MSE Mean Squared Error. 30

NN Neural Network. 9, 22, 23, 29

PCB Printed Circuit Board. 14, 15, 17
PHY Physical Layer. 40
PL Programmable Logic. 11, 12, 25, 26, 28, 39–41, 44–46, 48, 49, 59
PS Processing System. 11, 12, 24, 25, 28, 39, 41, 44–46, 48, 49, 55

RAM Random-Access Memory. 41, 44
RD map Range-Doppler map. 9, 22, 29, 30, 33, 41, 43, 45, 46, 48, 49, 53, 54, 56, 59
ReLU Rectified Linear Unit. 22, 30, 33, 34
RF Radio Frequency. 14, 15, 18
RGMII Reduced Gigabit Media-Independent Interface. 40

SGP General-Purpose Slave. 26
SIMD Single Instruction Multiple Data. 25
SoC System-on-Chip. 10–13, 24, 25, 27, 28, 39–41, 43, 44, 49, 54, 59
SRAM Static Random-Access Memory. 25
SRR Short-Range Radar. 13
STE Straight-Through Gradient Estimator. 23

Tcl Tool Command Language. 28

VHDL Very high speed integrated circuit Hardware Description Language. 11, 24, 28, 39, 48, 50
VSWR Voltage Standing Wave Ratio. 14