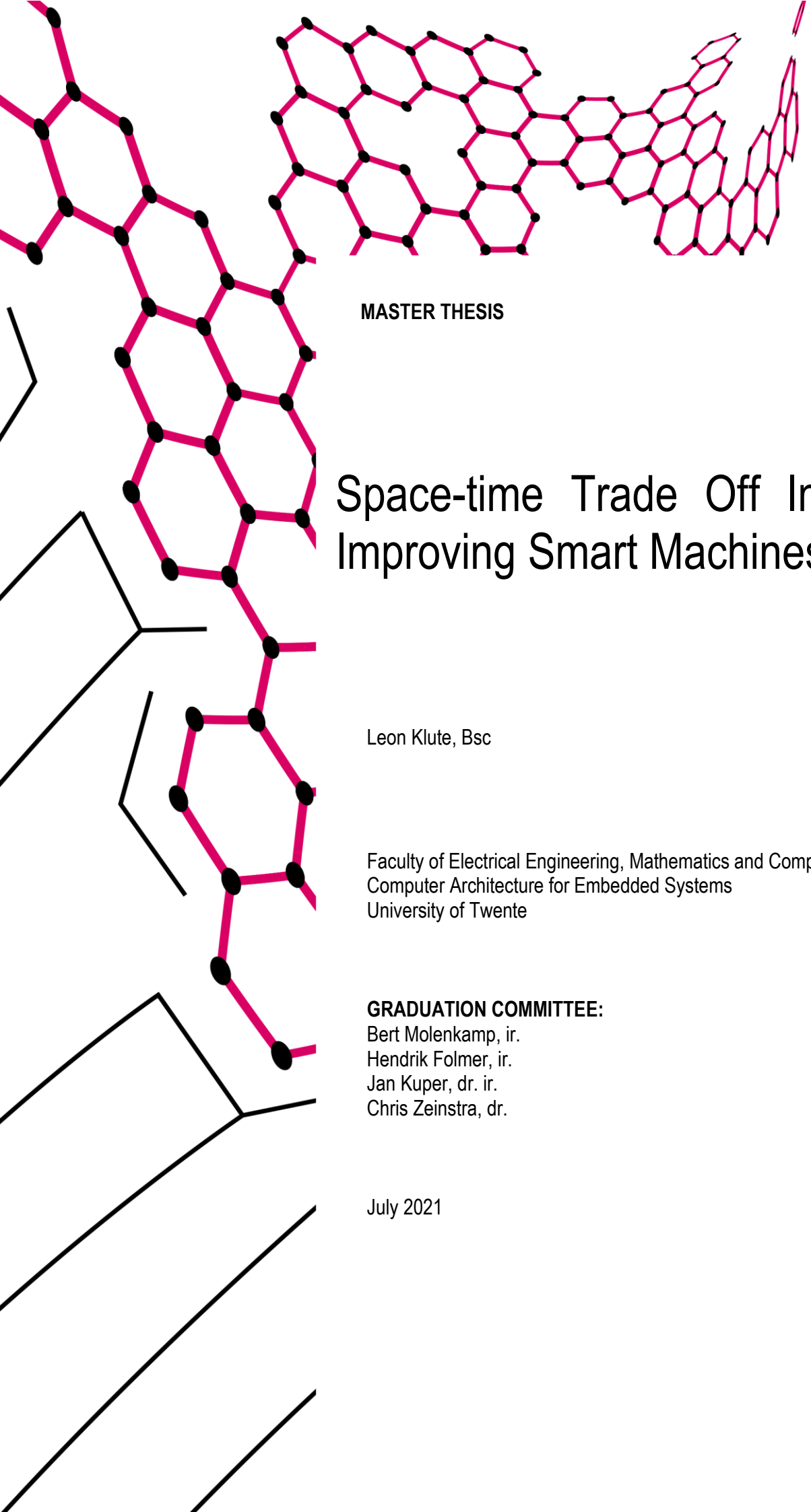

MASTER THESIS

Space-time Trade Off In Clash: Improving Smart Machines

Leon Klute, Bsc

Faculty of Electrical Engineering, Mathematics and Computer Science
Computer Architecture for Embedded Systems
University of Twente

GRADUATION COMMITTEE:
Bert Molenkamp, ir.
Hendrik Folmer, ir.
Jan Kuper, dr. ir.
Chris Zeinstra, dr.

July 2021

ABSTRACT

To implement artificial neural networks on embedded systems, it is desirable to compute them using specifically designed hardware. Making this hardware can currently be done with high-level synthesis tools, but these often do not offer a developer enough transparency and options. A new design flow is presented that incorporates the modern functional hardware description language Clash. This design flow allows the developer to scale the implementation to their needs.

CONTENTS

LIST OF ABBREVIATIONS

Abbreviation	Full phrase
[1234]D	[1234] Dimensional
ANN	Artificial Neural Network
ASIC	Application Specific Integrated Circuit
AST	abstract syntax tree
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HLS	High Level Synthesis
IR	Intermediate Representation
ML	Machine Learning
ONNX	Open Neural Network Exchange
ReLU	Rectified Linear Unit
RTL	Register Transfer Level
YOLO	You Only Look Once

LIST OF FIGURES

1 INTRODUCTION

Machine learning has shown to be a capable tool in tackling many tasks within computer engineering. For many of these tasks, it is also beneficial to implement them on embedded systems, such as small robots [1]. Embedded systems have limitations that are not as prevalent as in general computing. There are often strict timing constraints, such as in real-time systems, or little power is available. Because some machine learning algorithms, like artificial neural networks (ANN), generally use a lot of computing power, they are not easily implemented within these constraints. A solution could be to transfer the computations to an FPGA or an ASIC. Because they can perform many of the calculations in parallel. FPGAs and ASICs will often use less power than a general-purpose processor would use for the same computation.

Translating an ANN to an FPGA is currently not accessible to the computer engineers building the machine learning applications. It requires a different mindset and proficiency in a different field to develop an application in a hardware description language (HDL), compared to the data science knowledge needed for machine learning applications. A computer engineer attempting to offload work to an FPGA or an ASIC could use a general ANN accelerator. General ANN accelerators often support a wide variety of networks. For these general-purpose accelerators to support such a broad set of architectures, they will introduce more overhead than desired.

Much research has been conducted into the effort of translating software implementations to hardware implementations. High-level synthesis (HLS) tools are developed for this purpose, which can translate C-like software to HDL. However, these tools offer little to no transparency in the compilation process. This can result in unforeseen consequences from small changes in the software implementation. Thus, limiting the control of the developer.

A better intermediate language could be a functional language like Haskell, as it does not describe the steps to be taken by a processor but the relationship between input and output.

In the software development community, flexible platforms for data scientists already exist, e.g., TensorFlow [2], Caffe [3], and Theano [4]. These offer a unified interface to build and test networks on various computing platforms, like CPU, GPU, TPU, and cloud computing facilities. The same would be useful for the development process from high-level software implementations to custom hardware accelerators.

In this report we discuss the current status of using a different design flow to translate the artificial neural networks to an FPGA, namely using Clash.

1.1 Problem statement

The current HLS based systems do not offer enough transparency. The trade-off between resource usage and execution time is hard to make within these current tools.

To build a flexible platform we need some intermediary steps, and in this research, we investigate whether Clash is useful in this process, thus we come to the following research question:

How can Clash be used in a design flow from a software artificial neural network implementation to a hardware accelerator?

To answer this main question, we will first investigate the following sub-questions:

1. Can a design flow including Clash offer a developer an interface for making a time-area trade-off?
2. Can a design flow including Clash offer the developer transparency in their design choices?
3. How much flexibility does a design flow including Clash offer?

1.2 Overview of the report

In chapter 2, the background knowledge, needed for this report, is discussed. Such as the machine learning terms, their meaning, and the tools used while creating the design flow.

In chapter 3, related works, relevant papers researching aspects important for this research are summarized. Afterwards, we summarize the importance of the findings from the related works.

In chapter 4, design space exploration, the broadest scope of developing any design flow is narrowed. We see that it is best to start from an existing framework, construct a compiler that will

translate from this framework to Clash. The framework is a library that eases the network creation, while the compiler eases the translation to FPGA.

In chapter 5, we discuss how to implement the design flow chosen in chapter 4. Which languages to use for which purpose and the predefined building blocks used by the created compiler are discussed.

In chapter 6, the resulting design flow is discussed. Firstly we discuss how it works, then we show an example of it being used and we show the characteristics of the resulting implementation.

In chapter 7, we examine the resulting design flow and its performance to answer the questions from the problem statement.

Finally, in chapter 8, the future improvements and possibilities are discussed.

2 BACKGROUND

2.1 Artificial neural networks

Artificial neural networks are a part of the study of machine learning (ML). Machine learning is a device used in computer science when problems that need to be solved, become too abstract to write a direct algorithm to calculate solutions. Instead, the algorithms will be trained to produce the correct behaviour. This behaviour is not based on logic predefined by the developer but based on relations the system learns itself.

Examples of machine learning algorithms are decision trees, support-vector machines, Bayesian networks, genetic algorithms, artificial neural networks, and Q-learning. These approaches differ in what challenges they excel at and are thus used for different purposes.

In this project, the developed design flow is focused on the artificial neural network. Other machine learning algorithms will not be discussed in similar detail. Neural networks are most computationally demanding and will thus benefit the most from acceleration by an FPGA or ASIC. In the following sections, we will discuss from the basis of the artificial neural network, the perceptron, to the extension, the convolutional neural network, which was used in the project.

2.1.1 Machine learning frameworks

Machine learning frameworks are frameworks in which it is easier to develop, train and test machine learning algorithms, compared to building the algorithm from the ground up. They offer access to training algorithms and activations functions, without the developer having to implement them. Usually, all this functionality will be accessed by including a library in the project. Furthermore, these libraries have a backend that speeds up the computations executed for the algorithms.

Four machine learning frameworks are commonly used as tools in developing hardware, namely, TensorFlow, Keras, Caffe and Theano.

TensorFlow allows developers to easily leverage their hardware when training ANNs, as it provides a general interface to many hardware platforms. This way a developer can design a network without thinking about the performance on specific hardware [2]. Together with a user-friendly development environment like Python and Keras, the development and testing of ANN can become very trivial. Keras is a deep learning API written in Python running on top of TensorFlow. It enables even more user-friendly and faster prototyping of ANNs [5].

Caffe(Convolutional Architecture for Fast Feature Embedding) is a framework developed and maintained by *Berkeley Vision and Learning Center*. It is written in C++ and has Python and MATLAB bindings [3].

Theano is an open-source Python library for abstracting machine learning [4].

2.1.2 Perceptron

The basis of the artificial neural network is the perceptron. It multiplies inputs by internal weights. The results are summed and fed through an activation function to give the activation of the perceptron. This is mathematically described by Equation 1. The perceptron is also shown schematically in Figure

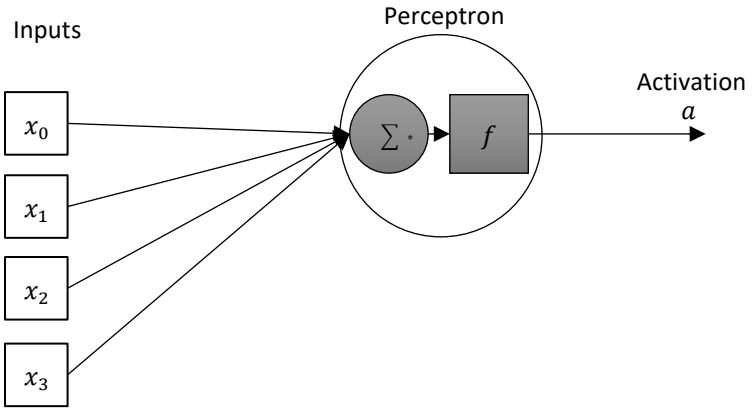

FIGURE 1 SCHEMATIC REPRESENTATION OF A PERCEPTRON

1. In the schematic, the internal weights are not shown to keep the schematic uncluttered. But each input ($x_0 \dots x_3$) to the multiply and accumulate operator (the grey circle), has a corresponding weight ($w_0 \dots w_3$) within this operator.

The perceptron can be used to make one prediction about a set of measurements. If the perceptron is used for a prediction, the activation (a), the output of the perceptron, will be used as the prediction.

The input ($\boldsymbol{x}$) is a vector consisting of n values, in the schematic shown as $x_0 \dots x_{n-1}$. The perception has a vector of weights ($\boldsymbol{w}$) of the same size n. The weights and inputs get multiplied and summed, shown by the grey circle in the schematic and $\sum_{n=0}^{N} w_n \cdot x_n$ in Equation 1. The result is a scalar value, which will be translated by the activation function f, in the schematic shown in the grey square.

2.1.2.1 Example of using a perceptron

FIGURE 2 AN IRIS FLOWER [26]

As an example, the perceptron will be used for predicting flower species. More specifically, predicting the type of iris from several leaf measurements, in Figure 2 such an iris can be seen. The petals and sepals can be measured. These measurements of an iris can be used for predicting the likeliness of these measurements belonging to the Setosa species. In this case, the prediction will be taken as the class of the measurements: 0.0, **not a setosa iris**, or 1.0 **a setosa iris**. The network will receive four measurements of an iris and predict which class it belongs to.

Four measurements of an iris are taken $[5.1\ cm, 3.5\ cm, 1.4\ cm, 0.2\ cm]^T$ from the data set [6], which are sepal length, sepal width, petal length, and petal width respectively. A pre-trained perceptron with weights $[-0.06205392, 0.90441537, -1.3889375, -2.893819]$ and bias 3.0697248, predicts whether these Setosa measurements do indeed match the setosa species, see Equation 2. The prediction is 0.97, which is close to the Setosa target of 1.0, this means the perceptron predicts these measurements are very likely to belong to a Setosa iris. If measurements of a Versicolor iris are taken $[7.0\ cm, 3.2\ cm, 4.7\ cm, 1.4\ cm]^T$, the prediction is 0.0064, close to the minimum of 0, thus the perceptron predicts these measurements do likely not correspond to a Setosa iris. See Equation 3 for the calculations.

EQUATION 1 PERCEPTRON EQUATION

$$\boldsymbol{a} = \boldsymbol{f}\left(\sum_{\boldsymbol{n=0}}^{\boldsymbol{N}} \boldsymbol{w_n} \cdot \boldsymbol{x_n}\right)$$

Where $\boldsymbol{a}$ is the activation, $\boldsymbol{f}$ is the activation function, $\boldsymbol{x}$ is the vector of inputs and $\boldsymbol{w}$ is the vector of internal weights.

EQUATION 2 EXAMPLE IRIS SETOSA PERCEPTRON CALCULATION WITH SETOSA MEASUREMENTS

$$\boldsymbol{a} = \boldsymbol{f}\left(\sum_{\boldsymbol{n=0}}^{\boldsymbol{N}} \boldsymbol{w_n} \cdot \boldsymbol{x_n}\right)$$

$$\boldsymbol{a} = \boldsymbol{\sigma}\begin{pmatrix}(\boldsymbol{-0.06205392 \cdot 5.1}) + (\boldsymbol{0.90441537 \cdot 3.5}) \\ +(\boldsymbol{-1.3889375 \cdot 1.4}) + (\boldsymbol{-2.893819 \cdot 0.2}) + \boldsymbol{3.0697248}\end{pmatrix}$$

$$\boldsymbol{a} = \boldsymbol{\sigma}(\boldsymbol{3.395427303}) = \frac{\boldsymbol{1}}{\boldsymbol{1 + e^{-3.395427303}}} = \boldsymbol{0.97}$$

The activation function $\boldsymbol{f}$ is the logistic function $\boldsymbol{\sigma}$ for this perceptron. The last activation of a network (in this case the network consists of only 1 neuron and is thus not a network) is the prediction, in this case, $\boldsymbol{a}$=$\boldsymbol{0.97}$. High likelihood of being measurements of the Setosa.

EQUATION 3 EXAMPLE IRIS SETOSA PERCEPTRON CALCULATION WITH VERSICOLOR MEASUREMENTS

$$\boldsymbol{a} = \boldsymbol{f}\left(\sum_{\boldsymbol{n=0}}^{\boldsymbol{N}} \boldsymbol{w_n} \cdot \boldsymbol{x_n}\right)$$

$$\boldsymbol{a} = \boldsymbol{\sigma}\begin{pmatrix}(\boldsymbol{-0.06205392 \cdot 7.0}) + (\boldsymbol{0.90441537 \cdot 3.2}) \\ +(\boldsymbol{-1.3889375 \cdot 4.7}) + (\boldsymbol{-2.893819 \cdot 1.4}) + \boldsymbol{3.0697248}\end{pmatrix}$$

$$\boldsymbol{a} = \boldsymbol{\sigma}(\boldsymbol{-5.049876306}) = \frac{\boldsymbol{1}}{\boldsymbol{1 + e^{-(-5.049876306)}}} = \boldsymbol{0.0064}$$

The activation function f is the logistic function σ for this perceptron. The last activation of the network is the prediction, in this case, a=0.0064. Not likely to be a Setosa.

2.1.3 Dense Layers

Dense layers are combinations of perceptrons, where the network can make multiple predictions at the same time. They are called dense layers because of the large number of connections with the previous layer, as every perceptron receives every input. They are also called fully connected layers for the same reason.

In one layer all the perceptrons get the same measurements but have different internal weights. The schematic can be seen in Figure 3. A layer consists of multiple parallel perceptrons, in this example 3. The operation of a layer can mathematically be represented as in Equation 4. There is now a vector of activation functions ($\boldsymbol{f}$) and a vector of weight vectors ($\boldsymbol{W}$).

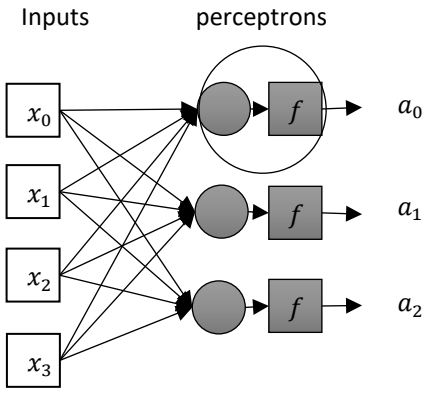

FIGURE 3 DENSE LAYER

EQUATION 4 DENSE/ FULLY CONNECTED LAYER

$$\boldsymbol{a} = \boldsymbol{f}(\boldsymbol{W} \cdot \boldsymbol{x})$$

Where $\boldsymbol{a}$ is the vector of activations, $\boldsymbol{f}$ is the vectorized activation function, $\boldsymbol{W}$ is the 2D matrix, containing one vector of weights per perceptron. And $\boldsymbol{x}$ is the vector of inputs

As an example, we can use such a layer to predict for given measurements what is the most likely type of iris. We can use pre-trained weights: $[[-0.06205392, -0.13310145, -0.14622506], [\,0.90441537, 0.28964716,\ 0.1499178\,], [-1.3889375\,, -0.33376053, -0.08010176], [-2.893819\ ,\ 0.4568877\ , 1.6784256]]$ and biases: $[\,3.0697248\,,\ 0.80369616, -2.2667842\,]$.

These weights and biases, together with the logistic activation function, define 3 perceptrons in one layer as in Figure 3. The calculation can be seen in Equation 5. Where the weights and the measurements from the perceptron example are used in this layer.

EQUATION 5 EXAMPLE OF A DENSE/ FULLY CONNECTED LAYER PREDICTING IRIS SPECIES

$$\boldsymbol{a} = \boldsymbol{f}(\boldsymbol{W} \cdot \boldsymbol{x})$$

The vectorized logistic function, $\boldsymbol{\sigma}$, will be used as the activations $\boldsymbol{f}$. $\boldsymbol{x}$ is extended with ones to add the biases within the matrix multiplication to $\hat{\boldsymbol{x}}$.

$$\boldsymbol{a} = \boldsymbol{\sigma}(\boldsymbol{W} \cdot \hat{\boldsymbol{x}})$$

$$\boldsymbol{a} = \boldsymbol{\sigma}\left(\begin{bmatrix} \boldsymbol{3.0697248} & \boldsymbol{0.80369616} & \boldsymbol{-2.2667842} \\ \boldsymbol{-0.06205392} & \boldsymbol{-0.13310145} & \boldsymbol{-0.14622506} \\ \boldsymbol{0.90441537} & \boldsymbol{0.28964716} & \boldsymbol{0.1499178} \\ \boldsymbol{-1.3889375} & \boldsymbol{-0.33376053} & \boldsymbol{-0.08010176} \\ \boldsymbol{-2.893819} & \boldsymbol{0.4568877} & \boldsymbol{0.4568877} \end{bmatrix}^{\boldsymbol{T}} \cdot \hat{\boldsymbol{x}}\right)$$

First, the Setosa iris measurements are input

$$\boldsymbol{a} = \boldsymbol{\sigma}\left(\begin{bmatrix} \boldsymbol{3.0697248} & \boldsymbol{0.80369616} & \boldsymbol{-2.2667842} \\ \boldsymbol{-0.06205392} & \boldsymbol{-0.13310145} & \boldsymbol{-0.14622506} \\ \boldsymbol{0.90441537} & \boldsymbol{0.28964716} & \boldsymbol{0.1499178} \\ \boldsymbol{-1.3889375} & \boldsymbol{-0.33376053} & \boldsymbol{-0.08010176} \\ \boldsymbol{-2.893819} & \boldsymbol{0.4568877} & \boldsymbol{1.6784256} \end{bmatrix}^{\boldsymbol{T}} \cdot \begin{bmatrix} \boldsymbol{5.1} \\ \boldsymbol{3.5} \\ \boldsymbol{1.4} \\ \boldsymbol{0.2} \\ \boldsymbol{1} \end{bmatrix}\right)$$

$$= \boldsymbol{\sigma}\left(\begin{bmatrix} 3.3954273 \\ 0.76275662 \\ -2.26427705 \end{bmatrix}\right) = \begin{bmatrix} 0.96756132 \\ 0.68195193 \\ 0.09412505 \end{bmatrix}$$

The network predicts Setosa very likely, 0.97, Versicolor probable 0.68, and Virginica unlikely, 0.094. For the measurements of the Versicolor, the result is:

$$\boldsymbol{a} = \boldsymbol{\sigma}\left(\begin{bmatrix} \boldsymbol{3.0697248} & \boldsymbol{0.80369616} & \boldsymbol{-2.2667842} \\ \boldsymbol{-0.06205392} & \boldsymbol{-0.13310145} & \boldsymbol{-0.14622506} \\ \boldsymbol{0.90441537} & \boldsymbol{0.28964716} & \boldsymbol{0.1499178} \\ \boldsymbol{-1.3889375} & \boldsymbol{-0.33376053} & \boldsymbol{-0.08010176} \\ \boldsymbol{-2.893819} & \boldsymbol{0.4568877} & \boldsymbol{1.6784256} \end{bmatrix}^{\boldsymbol{T}} \cdot \begin{bmatrix} \boldsymbol{5.1} \\ \boldsymbol{3.5} \\ \boldsymbol{1.4} \\ \boldsymbol{0.2} \\ \boldsymbol{1} \end{bmatrix}\right) = \begin{bmatrix} \boldsymbol{0.0063693} \\ \boldsymbol{0.46750218} \\ \boldsymbol{0.30210267} \end{bmatrix}$$

Meaning Setosa unlikely, 0.0064, Versicolor probable, 0.47 and Virginica less probable, 0.30. Even though the layer is not that sure, the highest prediction is still correct.

2.1.4 Artificial Neural Networks

The complexity between the input and output of one layer is very limited, as the output is a linear combination of the inputs, with one possibly nonlinear activation function. If the network needs to find more complicated relationships in the data, multiple layers can be combined to allow for any relationship to be possible to be learned. An example to show that layers with only linear activations cannot predict nonlinear behaviour is shown in Figure 6. For both predictions, a network with architecture (4 hidden neurons, 1 output neuron) was used to predict the blue target. The network with hyperbolic tangent as an activation function (in orange) was able to predict nonlinear output, to match the target more closely than the network (in green) with linear activations could.

Sequential layers can mathematically be described by Equation 6, where three layers 0, 1, and 2 are used. Which schematically looks like Figure 4.

The activations of the first layer are the inputs of the next layer. The first layer will extract information that is not directly obvious related to the input or output. In Equation 6 we can see that $f_0(W_0 \cdot x)$ is the description of the single-layer predictor from 2.1.3 Dense Layers, in this case, the output of this layer is multiplied by the weights of the following layer, W_1, and activated by its activation function, f_1, and so on till the output layer is reached.

EQUATION 6 MATHEMATICAL DESCRIPTION OF AN ANN

$$\boldsymbol{y = f_2\left(W_2 \cdot f_1\left(W_1 \cdot f_0(W_0 \cdot x)\right)\right)}$$

For a network with 3 layers, where y is the vector of predictions $\boldsymbol{f_m}$ is the activation of layer m, $\boldsymbol{W_m}$ is the weight matrix of layer m and x is the input vector.

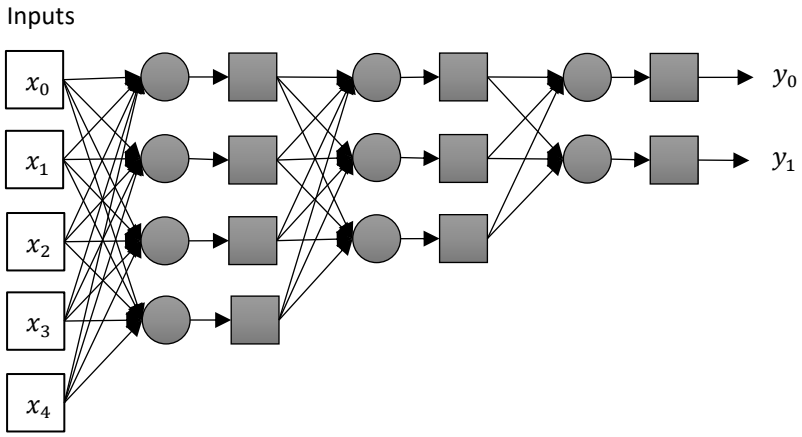

FIGURE 4 MULTI-LAYER NETWORK

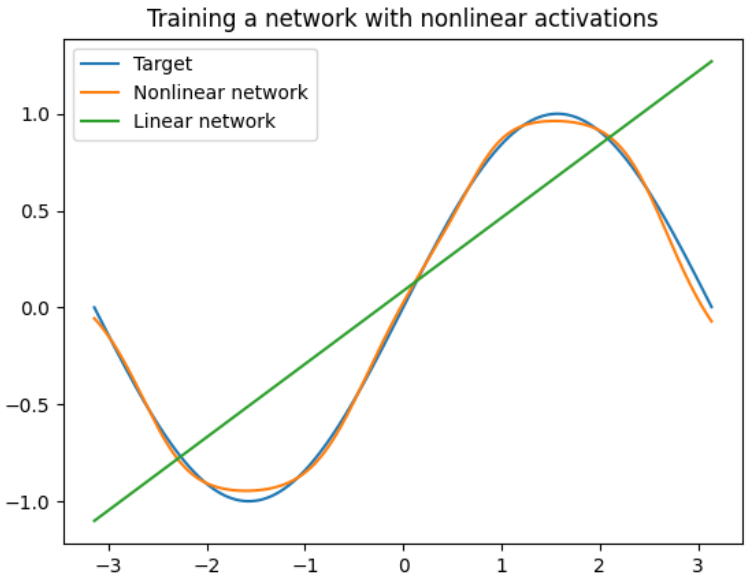

FIGURE 6 A MODEL WITH NONLINEAR ACTIVATIONS MODELLING A SINE WAVE

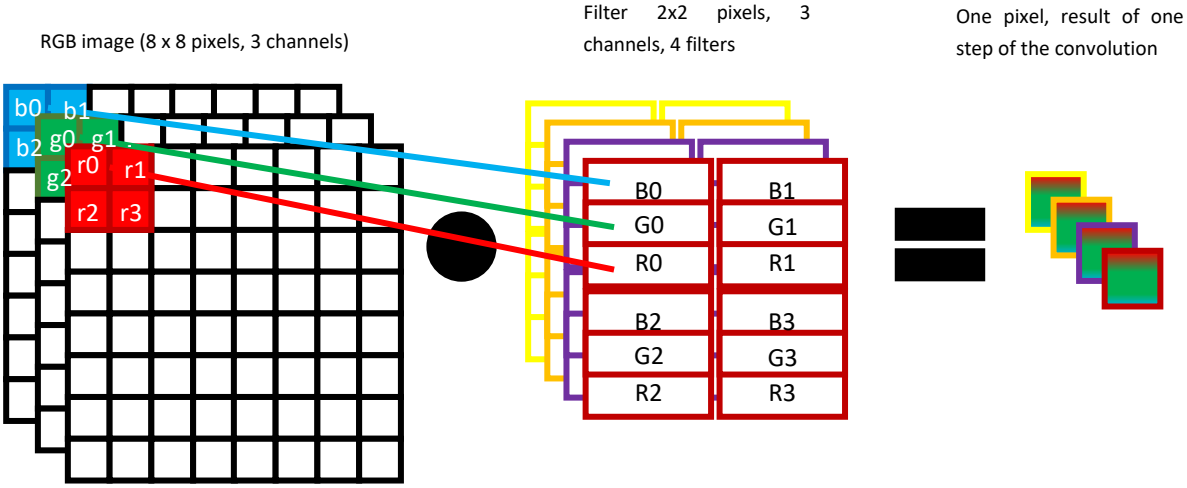

FIGURE 5 CONVOLUTION OPERATION ON AN RGB IMAGE USING 4 FILTERS AND WINDOW SIZE 2X2

2.1.5 Convolutional layers

The networks we have seen up to now work with a small set of measurements. The networks could also be used on images, to predict what items can be seen in the image for example. For an image, each measurement or input is one pixel value. For an RGB image of 6 by 6, that is $6 \cdot 6 \cdot 3 = 108$ inputs. Thus, for larger images, the number of weights becomes unmanageably large. This can be limited by using some knowledge about the inputs. The input image is a 3D matrix of values, but the information about the location within the matrix of each value is lost in a dense network.

Pixels close to each other are likely to have a relationship, and this relationship can be taken advantage of, to assist in making predictions about images.

We can try to find these relations in early layers. Small networks that look for the information on a small part of the image can be used as early layers. These small networks will be used on each part of the image as a filter. Applying such a filter to each part of the image is called convolution. Hence the name convolutional layer. In the following sections, the workings of these convolutional layers are discussed.

2.1.5.1 Convolution

Firstly, convolution can be explained in one dimension. The 1D discrete convolution is given by Equation 7. One signal is multiplied value by value by a filter, and the result of these multiplications is summed. This results in a new 1D sequence, where each output is a weighted average of the input

sequence. For convolutional networks, the sequences and filters are finite and 2D, which means the output is also finite and 2D.

EQUATION 7 1D DISCRETE CONVOLUTION EQUATION

$$(\boldsymbol{f} * \boldsymbol{g})[\boldsymbol{n}] = \sum_{\boldsymbol{m}=-\infty}^{\infty} \boldsymbol{f}[\boldsymbol{m}]\boldsymbol{g}[\boldsymbol{n}-\boldsymbol{m}]$$

Where $\boldsymbol{f}$ is the 1D input and $\boldsymbol{g}$ the filter.

2.1.5.2 Relation of dense networks and convolutional networks

An artificial neural network with convolutional layers is called a convolutional neural network (CNN). The convolution operation is visually represented in Figure 5. The input is an input RGB image and a set of 4 filters is applied to the input image. In this case, the filters have the size of 2-by-2 pixels, and each pixel has 3 channels, while the input sequence is the image, which is a 3D matrix of 6-by-6-by-3. Multiple filter sequences are shown as brown, purple, orange and yellow. Each of these filter sequences has weights for each of the channels B0, B1, B2 and B3 for the blue channel, G0-G3 for the green channel and R0-R3 for the red channel. Each of these filters is convolved with its respective channel, resulting in 3 values per filter ($\sum_{n=0}^{3} r_n \cdot R_n, \sum_{n=0}^{3} g_n \cdot G_n, \sum_{n=0}^{3} b_n \cdot B_n$), but these three are summed resulting in one value per filter. Shown as a value with a brown edge, a value with a purple edge, a value with an orange edge and a value with a yellow edge. The resulting pixel will be placed at the same index from the input image (x=0, y=0). In this case, because the filters have size 2-by-2, a window of size 2-by-2 was also taken from the input image at index (x=0, y=0).

After each filter has been applied to one index of the input sequence, (x=0, y=0), the filters will be applied to the next index in the input sequence (x=1, y=0). This new step will result in a different pixel at location (x=1, y=0) in the output image. When the filters have been applied to all indices of the input image, a new “image”, called a tensor is created. This tensor can be fed into a different convolutional layer. The following layer will have 4 channels instead of 3. The channels of the output tensor are equal to the number of filters in the convolutional layer. Channels are also often called features, as in later stages of a convolutional network there can be thousands of features/channels, which have no relation to colour channels. In the example of Figure 5, there are 4 filters, creating 4 features. These 4 filters applied to one window of the input image produce one “pixel” with 4 features.

A network with convolutional layers has a similar equation as a standard artificial neural network, but some of the layers will have a convolution operation, instead of the matrix multiplication, as can be seen in Equation 8.

A schematic representation of such a complete network with 2 convolutional layers and a fully connected layer can be seen in Figure 7. The network shown has two convolutional layers with ReLU activation. Each convolution has a pooling layer. The final classification is done by a dense layer. The pooling and activations will be discussed in 2.1.9 and 2.1.10 respectively. This example network predicts which vehicle is shown in the input image.

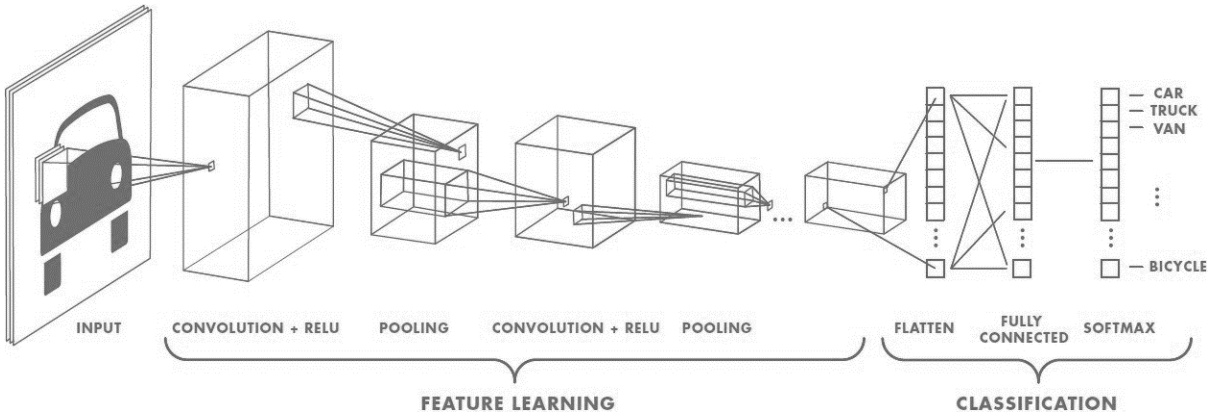

FIGURE 7 SCHEMATIC REPRESENTATION OF THE INNER WORKINGS OF A CNN FROM [25]

EQUATION 8 MATHEMATICAL DESCRIPTION OF A CONVOLUTIONAL NEURAL NETWORK

$$\boldsymbol{y} = \boldsymbol{f_2}\left(\boldsymbol{W_2} \cdot \boldsymbol{f_1}(\boldsymbol{W_1} * \boldsymbol{f_0}(\boldsymbol{W_0} * \boldsymbol{x}))\right)$$

For a network with 3 layers, where y is the vector of predictions $\boldsymbol{f_m}$ is the activation of layer m, $\boldsymbol{W_m}$ is the weight matrix of layer m and x is the input vector. The first two layers are convolutional layers,

which means that the matrix multiplication ($\cdot$) is replaced by the convolution operation ($*$). The weights of a convolutional layer are often called filters. Where the size of a fully connected layer needs to match the size of the input, the filters are smaller than the input.

2.1.6 Sparse neural networks

For very large networks many weights can become zero or close to zero, making the weight matrices sparse. These networks can be classified as sparse networks. If these zero weights are removed from the network architecture, fewer computations are needed. Because the weights close to zero added very little to the output, the accuracy of the network will stay the same while performing fewer computations.

2.1.7 Recurrent neural networks

Recurrent neural networks store a state, such that they can work well with time series when making predictions. Because the recurrent and sparse networks are not supported in the resulting design flow, we will not elaborate further on them.

2.1.8 Deep Neural Networks

Most currently researched ANNs fall in the category Deep Neural Networks (DNN) or Deep Convolutional Neural Networks, which means they have many layers, thus a lot of depth.

2.1.9 Pooling

To decrease the size of convolutional layers, pooling layers are often implemented. Pooling layers take the maximum or average value out of windows of their input and produce a smaller feature map. Instead of stride (1,1) as seen in the convolution example, the stride is often equal to the window size.

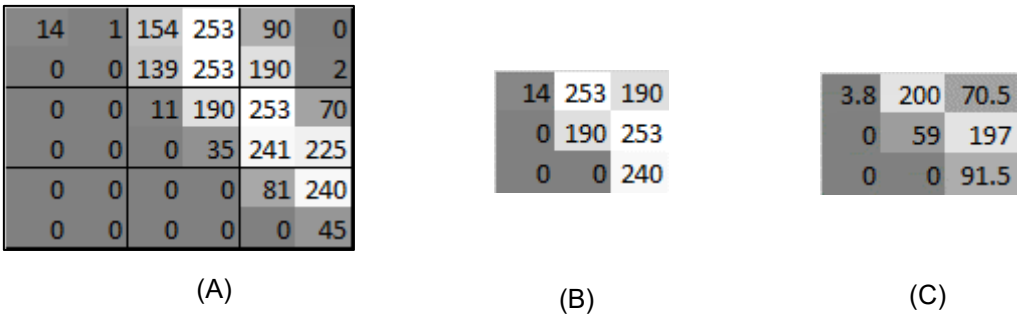

FIGURE 8 EXAMPLES OF POOLING WITH WINDOW SIZE 2X2 AND STRIDE 2X2; (A) ORIGINAL SAMPLE, (B) MAX POOLING, (C) AVERAGE POOLING

An example of pooling a sample of one feature can be seen in Figure 8. In this example, both the stride and windows are (2,2). The windows are shown with black lines around them. Each of those windows gets scaled down to one pixel in B and C, either by taking the average or by selecting the maximum.

Pooling with window size and stride (2,2) results in the input feature being cut in half in both x and y-direction. With both pooling methods, information about the input can be retained while removing 3 quarters of the data.

2.1.10 Activation layers

Activation layers apply an activation function to their inputs. The input to an activation layer is the output of a convolutional or a dense layer.

These activation functions are often nonlinear functions, like the Rectified Linear Unit or a sigmoidal function. These nonlinear functions make it possible for an ANN to make nonlinear predictions, as the dense and convolutional layers can only perform linear transformations as seen in Figure 6. Multiple linear transformations would only result in one equivalent linear transformation.

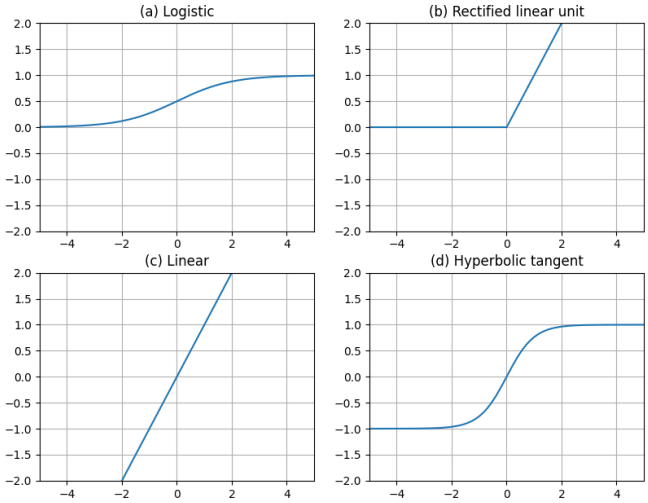

FIGURE 9 EXAMPLES OF ACTIVATION FUNCTIONS

2.1.10.1 Activation functions

Any function can be an activation function, but the most common are logistic, rectified linear unit and normalized logistic (softmax). The softmax is a normalized sigmoid, such that the sum of the outputs equal one. Other possible activation functions are the linear (no activation), exponential linear unit, softplus, and tanh. The exact workings of each of these activations are not important for the general working of the artificial neural networks. Some of the activation functions are shown in Figure 9.

2.1.11 Normalization

The output of the layers can be normalized, to keep them within a certain range. This is usually done through learning the variance and average of the features during training and using this to apply a transformation that centres the features around zero with a standard deviation of one.

2.1.12 Tensors

The multidimensional arrays containing the data between the layers are often called tensors. A tensor could be a single scalar value or a 1D vector, but when working with image processing they are often 3D. The output and input of a convolution layer are usually 3D tensors. The input and output of fully connected layers are usually 1D tensors also called vectors.

2.1.13 Training Artificial Neural Networks

To make correct predictions, correct weights are required. These weights are found through some training process.

For ANNs backwards propagation is normally used. Because of backwards propagation, activation functions are differentiable.

To perform backwards propagation, a data set, with inputs and outputs, is collected. The network will start blindly making predictions based on the inputs. These predictions will be compared to the correct output. The difference between the prediction and the correct output is measured through a loss function (E). If we differentiate the network with respect to the loss function, we can find a direction to move the weights in to get a smaller error. This process is performed iteratively until a local minimum is reached.

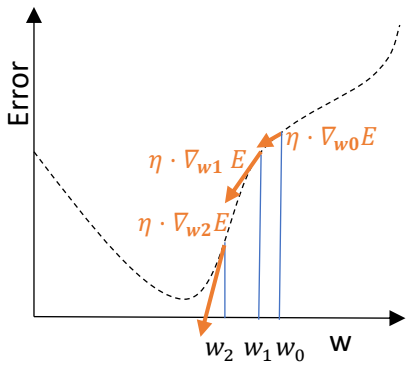

FIGURE 10 REGRESSION OF ONE WEIGHT

A visual representation of the process of numerically stepping toward a local minimum can be seen in Figure 10, where a theoretical system with only one weight would get the error given by the dotted line. The derivative at the point w_0 will point up the “hill” thus going the other way using a negative learning factor η, the weight will move towards the local minimum.

2.2 Field Programmable Gate Array (FPGA)

An FPGA is a device that contains programmable logical components. This allows a developer to design a system made of logical functions. Most computing tasks are done with a central processing unit (CPU). Where a CPU is an unchangeable block of logic elements that can execute instructions to perform a computation, an FPGA is more flexible. The elements to build a similar system are present but can also be used to build a different system. If an algorithm needs to be computed where two arrays need to be added, a CPU might have to do all additions one after another. The FPGA system could be implemented to perform this vector addition in one step.

Executing a logical function on an FPGA instead of on a CPU can increase the speed and reduce the power required. The speed can increase due to the potential for parallelizing the operations needed to perform the function. The power consumption can decrease, as less control hardware might be necessary, and parts of the CPU which are not required for the computation will not be present.

2.3 Compilation

Compilation in computer science is the process of translating a program from one language to another. The translation is often from a higher-level language to a lower-level language. For example, GCC compiles programs from C to machine code. But MATLAB coder is a program that compiles a MATLAB script to C.

2.4 Languages

Some programming languages that are important to this research explained concerning how they are used.

2.4.1 Python [7]

Python is an interpreted language, which makes it easy to quickly build a prototype. However, it is not very performant because of the interpreter. It is an imperative language where the developer describes the steps to be taken by the processor, opposed to declarative languages where the developer describes the desired result instead.

2.4.2 C(++)

C and C++ are programming languages that provide the developer control over the processer executions. They are developed to be compiled into a sequential program to execute on a versatile processor. They are inherently imperative, giving the programmer the task of deciding how to come to the desired results.

2.4.3 VHDL

VHDL stands for (Very High Speed Integrated Circuit) Hardware Description Language, which is, together with Verilog, the most commonly used Hardware Description Language (HDL). They are very low level as they require the developer to have a lot of understanding of how the hardware works.

2.4.4 Clash [8]

Designing hardware for specific tasks will become more important as general-purpose processors seem to reach the limit of their size and speed. The design workflow of hardware may include multiple languages like is currently the case for many software projects.

Clash is a functional hardware description language that can be used to design synchronous and asynchronous logic, thus also Mealy and Moore finite state machines. Clash is the name of both the language and the compiler. The language is an extension of a subset of Haskell, the Clash compiler can translate this language to VHDL and Verilog. Haskell is extended with time series in the form of signals. The functional language paradigm of Haskell is well suited for describing combinatorial operations. The signal allows for these combinatorial descriptions to be used on time series. Because Clash is based on the Haskell it features many modern abstraction mechanisms such as higher order

functions and type inference, while the paradigm on which Haskell is based, functional programming, is especially well suited for describing the combinatorial behaviour of a system.

3 RELATED WORK

Because accelerating ANNs and specifically CNNs could provide such a great benefit, numerous ways of building the accelerators have been developed. In this chapter, we will explore how these design flows operate and which insights they offer.

3.1 Frameworks

3.1.1 WiderFrame: An Automatic Customization Framework for Building CNN Accelerators on FPGAs: Work-in-Progress

The authors of [9] propose a framework for a systematic design space exploration methodology, for designing a convolutional network accelerator. This promises to make the right choices when starting with a CNN specification and an FPGA specification. Their schematic architecture can be seen in Figure 11.

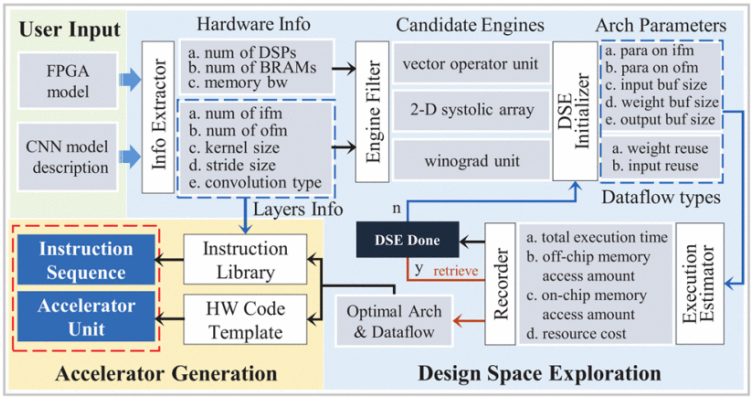

FIGURE 11 ARCHITECTURE OF WIDERFRAME FROM [8]

This framework has been made because the existing frameworks only support one neural computing engine, while they have identified three architectures. These architectures have different parallelization characteristics. The engines are the vector operator unit, the 2D systolic array and the Winograd unit. These engines can also be seen in the other accelerator implementations in 3.2 Accelerator designs.

The system can easily be extended with extra instructions to support emerging new CNN architectures.

The hardware code template is written in high-level synthesis-based C++ language. In the hardware code template, a description is written of the engines and the other predefined blocks, that could be needed to develop the design that the DSE method proposes.

From this paper, we can see how code templates can be used to build a custom accelerator for a network.

3.1.2 ONNC: A Compilation Framework Connecting ONNX to Proprietary Deep Learning Accelerators

The authors of [10] aim to translate *Open Neural Network Exchange* (ONNX) network specifications to deep learning accelerators. Where the intermediate representation (IR) within the compiler has a one-to-one mapping with the ONNX IRs, making it easy to add operators that are not in the standard environment. This should make it easier to make an accelerator with an instruction set, and then use this compiler framework to build the program to run a specified network. The toolchain is open source so anybody can easily add a backend for their accelerator.

An important part of the compilation process is pass management, in which each pass can perform certain translations/optimizations. Pass management is inherited from LLVM.

A big advantage of this Compiler compared to Glow and TVM, other compilation frameworks, is that no LLVM IR is used in between, which often has too fine-grained operations compared to the instructions going into Deep Learning Accelerators.

From this paper, we can learn that translating a network specification, can best start from a higher-level, coarser-grained description.

3.2 Accelerator designs

3.2.1 Scalable and Modularized RTL Compilation of Convolutional Neural Networks onto FPGA

The authors of [11] translate Caffe and Theano based CNN models directly to an unspecified RTL code (likely VHDL or Verilog).

The Convolutional layer is seen as 4 for loops, as can be seen in Code Block 1.

1. Across the output feature maps of N_{of}	**Loop-4**
2. Across the input feature maps of N_{if}	**Loop-3**
3. Scan within one input feature map with $X \cdot Y$	**Loop-2**
4. MAC within a kernel window of $K \cdot K$	**Loop-1**

CODE BLOCK 1 FOUR FOR LOOPS DEFINING CONVOLUTION OPERATION FROM [11]

Where N_{of} = Number of output features, N_{if} = number of input features, X, Y are the dimensions of the input features and K is the window size.

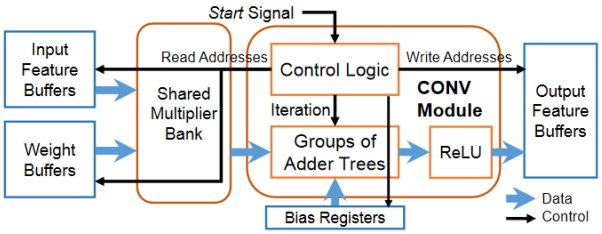

FIGURE 13 CONVOLUTION ACCELERATION MODULE BLOCK DIAGRAM FROM [10]

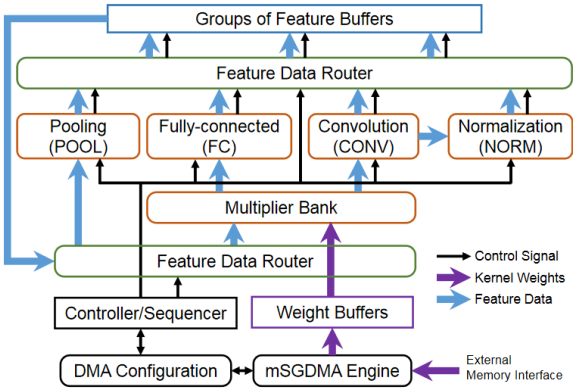

FIGURE 13 INTEGRATED SYSTEM FROM [10]

These loops will be unrolled according to their analysis:

unroll loop-3 such that pixels from different inputs can be multiplicated with their filters in parallel.

How much the layer can be unrolled depends on the number of multipliers (N_{mult}) implemented, if $N_{mult} \geq N_{if}$, loop-3 can be fully unrolled. N_{mult} can be defined by the user.

They can then unroll loop 4 to calculate if there are enough multipliers. They state that if they reorder their data, they could in the future also unroll loop 1.

Their scalable convolution acceleration module can be seen in Figure 13

Furthermore, they have modules for the other layers, but because the other layers use relatively less computation less time is put into optimizing these.

The other modules are:

- Pooling module, which has two variants, average and max pooling
- Normalization module, which computes the local response normalization operation. All activations of a sample get normalized to have a standard deviation of 1 and a mean of 0.
- Inner product module, which calculates the fully connected layers
- DMA configuration module, which controls the Direct Memory Access (DMA) to communicate between the on and off-chip memory

The block diagram for the integrated system can be seen in Figure 13.

In the actual implementation, a softcore is used to coordinate the memory transfers, together with the DMA module. There is one shared *multiplier bank* which is the part of which the size can be configured to increase or decrease the area while increasing or decreasing the calculation time.

They mostly limit memory usage based on the finding that multicore processors use most of their power due to their cache [12].

From this paper, we can see how to unroll the convolution operation, and which building blocks could be built for the design flow. Furthermore, we can learn how the fully connected and convolutional layers can reuse hardware. The hardware that is implemented is a kind of vector unit, using multipliers with adder trees.

3.2.2 Utilizing cloud FPGAs towards the open neural network standard

Translating from *open neural network exchange format* (ONNX) to FPGAs has been shown to work [13]. They built an application to run ONNX models on cloud FPGAs.

It is built as a streaming application, such that the intermediate results do not have to be stored in off-chip memory. The intermediate results are stored in block RAM, close to the arithmetic operations.

The researchers tried the HLS4ML (an HLS tool) which didn't work directly as it caused memory issues. So, they needed to make several modifications to the HLS4ML tool to make it work.

8-bit integers are used in most of the network, multiple of these values are passed together to the DSP to improve performance, because the DSPs have an input of size 27, they can fit 2 8-bit integers at once. But different precisions are possible between the layers. The precision of the weight and activations can also be changed per layer. To achieve this, each layer has its own accelerator.

Processing can be performed in batches, thus increasing the theoretical throughput. As more parallelism can be utilized as the weights can be used for multiple samples at the same time.

During training, the weights were regularized to be positive and within a small range [0,2.5].

Their hardware optimized network performs significantly less accurate than a default model, they stated about 4% accuracy loss. The overview of how the system works can be seen in Figure 15.

This paper shows how building an accelerator as a streaming operation with intermediate storage close to the operations will benefit a design. It also shows that scaling down the bit-width can be performed to decrease the resources needed at the cost of accuracy.

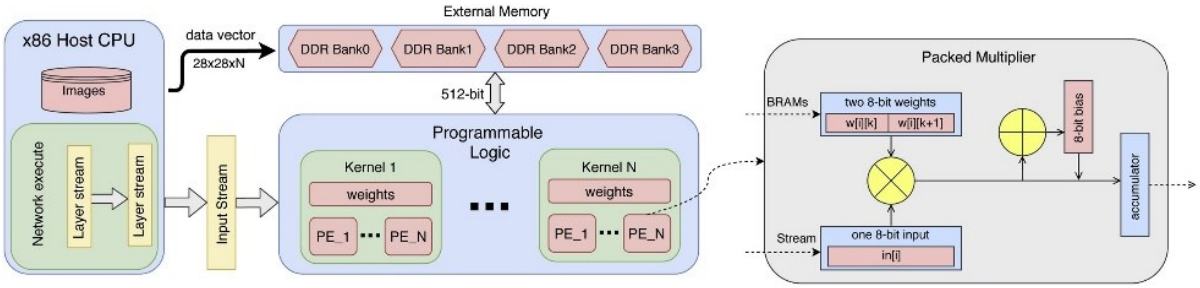

FIGURE 15 SYSTEM OVERVIEW FROM [12]

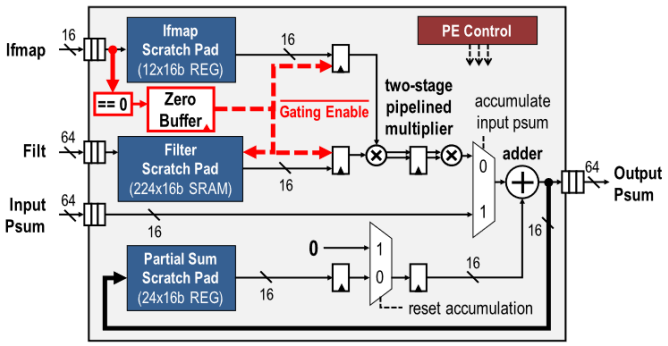

FIGURE 14 BLOCK DIAGRAM OF A PE FROM [13]

3.2.3 Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks [14]

Eyeriss is a CNN accelerator, which has been developed with a hardware design that minimizes data transfers, as this can be a major factor in energy cost. An analysis framework is created that compares energy cost under area and processing parallelism constraints. It is implemented as an *Application Specific Integrated Circuit* (ASIC). It aims to support as many convolutional layers as possible, but because it is implemented beforehand it has limited supported CNN layer shapes:

- Filter height: [1...12]
- Filter width: [1...32]
- Num of filters: [1...1024]
- Num of channels: [1...1024]
- Vertical stride: {1,2,4}
- Horizontal stride: [1...12]

These limitations will still cover the most common CNNs.

The arithmetic precision is also decided beforehand at 16 bit. The accelerator consists of a 2D array of 168 Processing Elements. It is not a systolic array as some data is transferred globally to use data in parallel, thus the PEs are not necessarily in the same operating state.

The block diagram of the PE can be seen in Figure 14. And how these PEs are used within the system can be seen in Figure 16.

The multi dimensional convolution operation is first divided into 1D convolution operations. The 1D convolutions calculate partial sums (Psums), that are summed to create the output feature map (Ofmap). To limit the data transfer from off-Chip DRAM a run-length decoder is used, that encodes the leading number of zeros, which is often very large in a convolutional network.

The design shows how a systolic array could be used as the computation engine. In this case, the systolic array is extended with some global data casts, resulting in an engine that is not a systolic array. It also provides some strategies for limiting data transfers between the FPGA and any off-chip memory.

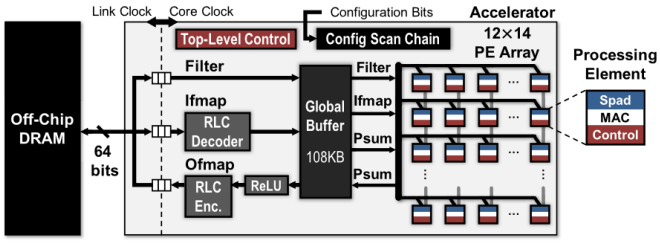

FIGURE 16 SYSTEM ARCHITECTURE FROM [13]

3.2.4 Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA

The authors of [15] design a flexible accelerator, together with a compiler for the Caffe network descriptions. The model is compressed before synthesis, by decreasing the bit-width in each layer. This compression can go down to 8-bit if it shows to be accurate enough. They analyse the statistics of the weights and outputs of each layer to see how many bits they need.

The accelerator has three kinds of instructions: LOAD, SAVE and CALC.

They use PEs that can compute convolution already using parallelism, but multiple PEs can be implemented to work side by side. Their accelerator only supports 3x3 kernel size, but they can pad smaller kernel sizes and split larger kernel sizes into 3x3 windows.

This project shows an actual systolic array architecture as a computation engine. In this case, the design flow includes software development which drives the accelerator. The benefits of differentiating the bit-width per layer are also shown.

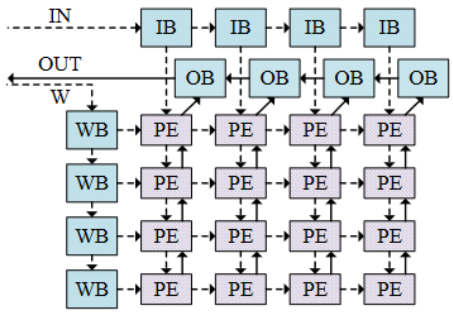

FIGURE 17 SYSTOLIC ARRAY ARCHITECTURE FROM [15]

3.2.5 Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs

Making a scalable CNN implementation on FPGAs has been achieved by making a systolic array [16], where they can transfer a high-level C implementation to a flexible 2D systolic array for computing CNNs while scaling up to the size of the FPGA. The 2D systolic array is chosen because it maps nicely to the architecture of FPGAs, resulting in low routing complexity. The structure of the systolic array can be seen in Figure 17. While the structure of the elements of the systolic array can be seen in Figure 18.

The layers are defined as pseudo-C loops, which give some difficulty in mapping to the systolic array because of the unclear data dependencies.

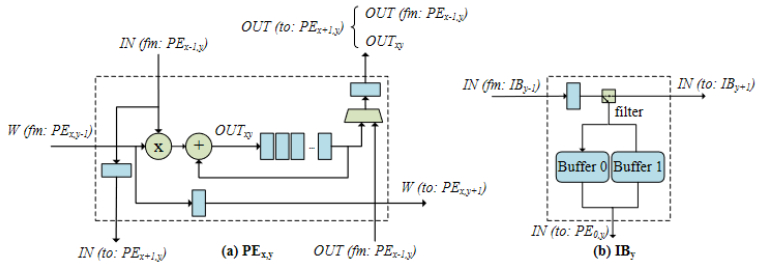

FIGURE 18 BLOCK DIAGRAM OF PE AND BUFFERS FROM [15]

The systolic array only has local communication which allows for high clock frequencies.

The convolution is mapped to the 2D systolic array using an analytical model that can be optimized for maximum throughput within the feasible design space. The analytical model is called the loop tiling [17] representation, which defines a link between the architecture and high-level program code. This intermediate representation (IR) is sequential, such that they can use some standard tools for the analysis and modelling.

This paper shows the advantages of the systolic array, like the high clock frequency. The unrolling of the convolution operation is also discussed.

3.2.6 Embedded Neural Network Design on the ZYBO FPGA for Vision-Based Object Localization

The author of [1] has built a CNN implementation on the ZYBO FPGA platform using VHDL, to test whether this was feasible. This is tested because the latency when performing inference off-site is unpredictable. Performing CNN interference on the processor in an embedded system is infeasible, because of the power and resource limitations.

To test the feasibility a robot was made that uses an FPGA to perform object tracking from a camera in a power-constrained and real-time environment. The network must tell from the camera whether another robot is centre, left, right or not in sight.

An accelerator for this CNN is developed, where the network was created and trained using Keras-TensorFlow and trained on a workstation. The FPGA is used to accelerate the CNN layers, while ARM cores are used for the final fully connected layers. Several choices are made to fit the network on the FPGA, the activation function is chosen to be ReLU, as ReLU is a very computationally effective activation function, as it can simply use the sign bit as mux input. Furthermore, the kernel size is kept to 3 by 3.

Training of the network became hard with a versatile dataset; sigmoid functions were needed on the fully connected layers. Multiple convolutional layers without activation function seem to perform more similar to a convolutional layer with a larger kernel size (than the 3x3 used). This might indicate that a larger kernel size is desirable

The generated data did not seem adequate for training this small CNN, but it also overfits. Thus, a different network architecture was chosen, namely a one-shot detection similar to YOLO.

To reach a clock frequency of 100 MHz a pipeline is created with 4 stages.

A python Keras object is automatically translated into instructions for the accelerator. This thus works for the convolution layers. 90% of calculation time for an object detection network was spent in the convolution layers. In the end, the object detection did not work due to an inadequate training set.

From this paper, we see a possible application, where a network is trained on a workstation using a machine learning library. Afterwards, this network is implemented on an FPGA in a robot. The robot can perform inference using the accelerator.

3.2.7 CaFPGA: An automatic generation model for CNN accelerator

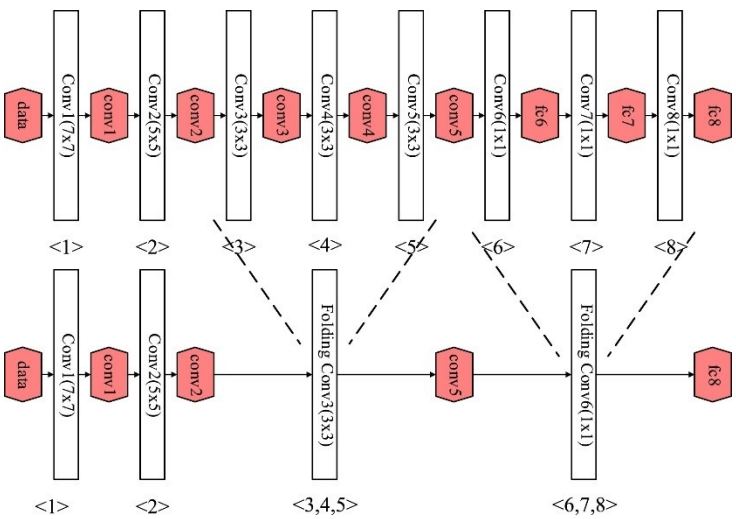

FIGURE 20A POSSIBLE LAYER FOLDING FROM [17]

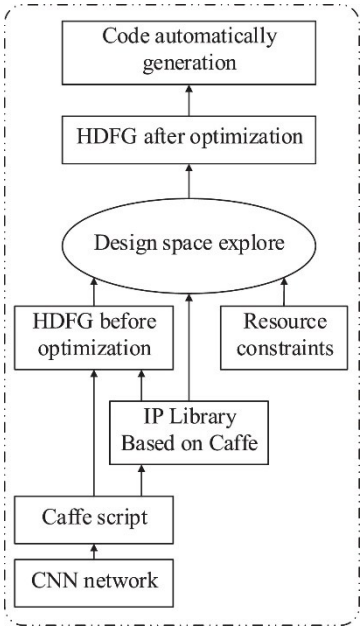

FIGURE 20B WORKFLOW FRAMEWORK FROM [17]

The authors of [18] have made an algorithm to translate Caffe network descriptions to Verilog. The model takes in a Caffe script, translates this to an IR. This IR is a hard data-flow graph (HDFG). The design space exploration algorithm will modify the HDFG for optimal performance. The workflow can be seen in Figure 20B.

The accelerator they build uses an array of Processing Elements (PEs), in which they use a 2D convolver structure. Each PE calculates one layer from the network but can also calculate multiple layers if they are similar enough. The Design space exploration algorithm decides how many layers each PE can calculate. Between the PE there is a ReBuffer IP (their predefined blocks are called IP), which either stores the intermediate data in a Cache IP or stores it off-chip if there is not enough space available.

The layer combinations can be seen visually in Figure 20A. The layers with similar window sizes can be calculated with only one PE, to be reused in time.

The parallelism can be divided into temporal and spatial parallelism, where temporal parallelism means a pipeline structure. The convolutional layer parallelism is divided into three layers: feature-map-level, window-level and operator-level. These parallelisms will be exploited as spatial parallelism, while the pipelining allows different images to be processed at the same time.

This approach shows that when layers closely match, a PE can process multiple layers, while it does not have to be able to process all layers. We can also see a proof of concept of a design flow from Caffe, with custom predefined blocks.

3.2.8 Other nameworthy accelerators

ALAMO [19] uses adder trees and a shared multiplier bank, which is similar to the approach of [11].

3.3 Future of hardware description languages

The authors of [20] discuss the challenges and needs for future hardware descriptions languages. Verilog is the current standard, but the future might be a multi-language environment with space for functional HDLs, and Virtual Machine approaches (commonly called HLS). In which higher-level languages can have their place next to the lower-level approaches. Languages like Clash offer more reusability and abstraction than Verilog, but the lower level can offer a more direct interface with the hardware for the developer. Although almost everything is expressible in Clash, it might in some cases be more convenient to do in VHDL or Verilog. For example, how to enter two 8-bit integers into one DSP, is more transparent when done in a lower-level language.

The RTL Codes, Verilog and VHDL, are likely less readable, when produced by a compiler, compared to human written RTL Code.

It might thus be beneficial to use multiple hardware description languages in a design flow when it is found which languages perform best for specific tasks.

3.4 Conclusion

The research on the topic of generating hardware architectures for ANNs can generally be split into the following approaches:

1. Using an accelerator, this accelerator can be utilized via a specific software toolchain.
2. A high-level synthesis (HLS) tool is used to compile a software implementation to an HDL implementation.
3. A hybrid variant, where a combination of a soft-core processor is used together with a synthesized accelerator.

The papers on design space exploration frameworks in 3.1 develop analysis tools for their architectures. This analysis is used to build an optimal system for a given architecture-platform combination.

We can see that most existing tools are either based on classic RTL codes, VHDL and Verilog, or on a C++ implementation that will be translated to these RTL codes. Using a modern RTL has not been tested.

4 DESIGN SPACE EXPLORATION

In this chapter, the possible choices while designing the system are discussed. The possible advantages and disadvantages of the options are weighed, and we elaborate on the choices made.

4.1 Overview

To include an ANN in an FPGA project we first have to develop, train and test a network architecture. We can build ANNs from scratch, but because many useful machine learning platforms have already been developed, it is not desirable to develop a new platform to translate networks to hardware. Therefore, an existing platform can be used as the basis to build, train and test architectures.

Because the existing platforms do not support Clash or Haskell, the system needs to translate/compile from one of these high-level machine learning frameworks to an FPGA. In this system, Clash needs to be used as an intermediary to investigate whether it benefits the developer.

To translate from the high-level platform to Clash, a compiler will be developed. It will take some representation in the high-level platform infrastructure and translate it to Clash.

After translating the high-level implementation to Clash, we need to translate from Clash to RTL codes. This is done by the Clash compiler. The Clash compiler can compile to both VHDL and Verilog, allowing for some flexibility for the output. Both of these RTL Codes (VHDL and Verilog) are supported by the software coming with FPGA platforms, such as Quartus [21].

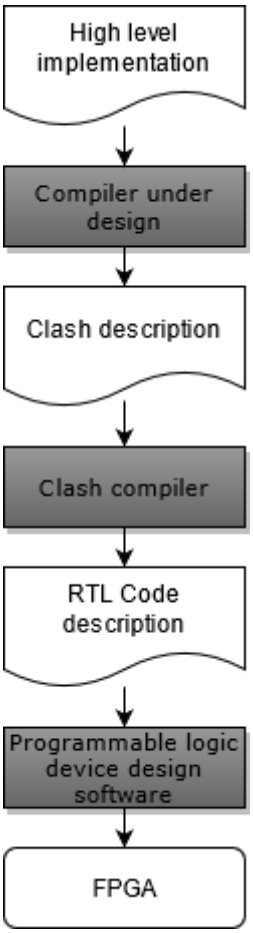

FIGURE 21 OVERVIEW OF THE SYSTEM UNDER DESIGN

4.1.1 Starting framework

To build the compiler, a platform needs to be developed or chosen. Because no new framework will be designed for this design flow, an existing platform will be chosen. There are several possible platforms from which we can start the compilation: *TensorFlow*, *TensorFlow-Keras*, *Theano*, *Caffe* or the *Open Neural Network eXchange* (ONNX) [22]. TensorFlow is currently one of the most used frameworks. It comes packaged with the high-level API Keras, which makes prototyping artificial neural networks on TensorFlow even more user friendly. Theano and Caffe are developed by universities, *Montreal Institute for Learning Algorithms* [4] and *Berkeley Vision and Learning Center* [3] respectively. They offer mostly the same capabilities, but have fewer contributors and contributions, making them slightly less extensive.

ONNX is a general way of representing Neural networks, and not meant for developing, training and testing the networks. But for sharing networks between different frameworks. Using ONNX as the basis would make most frameworks accessible to the system, albeit with an extra layer in between.

Some frameworks exist for making quantized networks, where the arithmetic weights and are set to be of a type smaller than the floating-point numbers used in standard networks. Using ONNX would allow access to these quantized networks, which could be very beneficial for an FPGA based accelerator.

Keras is chosen as the basis because Keras is currently very commonly used and user friendly. The resulting compiler will be a Keras-to-Clash compiler. Which representation we will use from the framework is discussed in the following section.

4.1.2 The entry point to the framework

The high-level ANN description needs to be compiled to some hardware implementation. Normally, the ML platforms translate the network into instructions for supported hardware, such as a CPU or a GPU. In this case, the Network needs to be translated to a different platform, the FPGA. Thus, somewhere in the standard process from building the network to running it on a CPU or GPU, the description needs to be branched off and translated to a different set of instructions, and in this case a hardware description. There are several options from where to start the translation:

- A compiled implementation of the network

- An intermediary representation within the compiler framework
- Python implementation for the platform
- The high-level description within the platform

These points are shown within the typical flow of a high-level implementation to an embedded implementation in Figure 22.

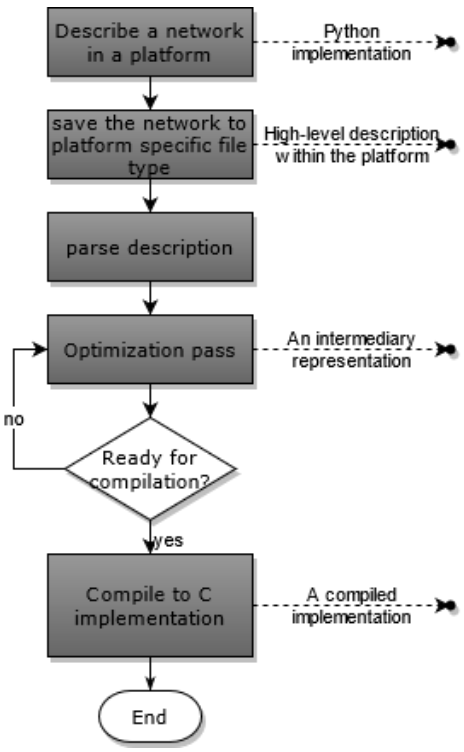

FIGURE 22 POSSIBLE ENTRY POINTS

The compiled implementation of the network is reached by first compiling the network into a lower-level language, that describes all the steps that need to be taken to compute the result. The existing compilers are aiming at CPU instructions or GPU instructions. This lower-level representation could be translated to Clash. However, lower level means the instructions are finer-grained and describe the steps to be taken relating to a state instead of describing an architecture. Therefore, loops will often be used to describe the input to output translation, with calculations like multiplications and additions within the loops. These loops can be translated to higher-level functions like maps and folds, but the data dependency would need to be analysed. Doing this analysis has been proven to be very hard, and is thus not feasible [23].

When using an intermediary representation from the compiler, the representation will have moved towards the finer-grained, giving up some control, as the finer-grained instructions allow less room for making choices later in the translation process.

It would thus be more beneficial to use more coarse-grained instructions for translating to Haskell. Coarse-grained instruction can include, convolve, normalize, apply an activation function to a tensor. Depending on the level of abstraction different building blocks are required. If the fine-grained instructions are taken, hardware needs to be designed that works similar to a processor, but specifically made to compute the specific instructions needed for the network. On the other hand, when the coarse-grained instructions are taken as the basis, the hardware can be specialized to perform the few tasks it can execute.

The Python implementation is the script, that builds the network in the ML learning framework. The framework may support multiple ways of describing the network, which could make this route more difficult.

The high-level description within the framework is a file type specific to each platform. The platforms often support some form of saving and exporting the implemented networks, such that they can be stored and used in another place or time. This representation will be consistent and contain all the parameters needed to build the network. Therefore, it is the best choice. Thus, some exporting function within the platform should be used to get a consistent description of the network.

4.2 Constraints of the design flow

To show the possibility of using Clash in a design flow, we develop a compiler. This compiler will translate from the high-level internal Keras description to a Clash description.

This compiler is built for certain specifications, but will not be capable of translating any machine learning application. The space of all possible networks is too broad, thus targets will be set for the compiler and design flow.

4.2.1 Size limitation

Larger networks are shown to always be better at certain tasks, such as autoregressive language models [24]. However, larger networks come at the cost of more resource consumption and computation time. Therefore, we limit the network sizes for the design flow will support.

While the system should be able to handle most network sizes in the end, it might be more feasible to start with a smaller ANN, like an ANN for the MNIST data set, and slowly extend the capabilities to support increasingly large networks. To support a larger network, more flexibility should be offered in

reusing computing resources and which memory can be used. The system will first support the smaller MNIST recognition network as a proof of concept and should be tested on larger networks such as AlexNet.

4.2.2 Supported architectures

Commonly used architectures are the following:

- Deep fully connected networks
- Convolutional neural networks
- Sparse neural networks
- Recurrent neural networks
- Long short-term memory recurrent network

From these, the most common ANN architectures are currently the CNN and the standard DNN. Thus, it would be most desirable to support at least one of these networks. CNN networks often also use fully connected layers from the DNN. Thus, to support CNN would likely also mean supporting DNN.

A fully connected layer can be calculated as a special case of a convolutional layer. It would be a convolutional layer with a window size equal to the input. Other networks that are interesting to support are the sparse neural network, which could greatly benefit from being accelerated in hardware designed specifically for accelerating sparse networks. Recurrent neural networks could also be built, although the state they hold would require large additions to the hardware, as this could require much more memory and data following a different path.

Convolutional networks and fully connected networks will be supported.

4.3 Implementation choices

Now that we have decided on the goals of the design flow, the development needs to be divided into smaller steps to build the design flow.

4.3.1 System input

To build an implementation that can process data on the FPGA, the input format needs to be defined. The input vector could be loaded on on-chip memory if the vector is expected to be small enough. The inputs are not likely to be small enough, thus the inputs need to enter the FPGA sequentially.

More parallelism can be achieved if multiple input images are calculated at the same time, as the weights can be reused for both inputs. For cases where there is a stream of images coming in, it might be undesirable to wait for multiple images to come into storage before starting processing.

Because for most machine learning applications it is infeasible to process a whole input vector at once, some way of splitting the input must be devised. Images can be split into windows that fit the convolutional window size.

The final choice on the input is what type of data can be used as an input.

- Integer
- Unsigned integer
- Floating point
- Fixed point

The fixed-point representation is chosen, because it balances the size and range flexibility, while not needing the hardware complexity that is required when doing floating-point arithmetic.

4.3.2 How to implement the layers in the design flow

The design flow will use predefined blocks to build an FPGA architecture that can compute a given ANN architecture. These blocks will be based on the layers present in an ANN. The Keras library has functions, which also match the theoretical building blocks of artificial neural networks. The layers can be based on these building blocks based on theory and the Keras library.

4.3.2.1 Convolutional layer

To perform the convolution operation several “neural network engines” have been developed and used in the past as discussed in chapter 3 Related Work:

- The systolic array
- The vector unit
- The Winograd unit

Each of these can be implemented in the design flow, or multiple can be optional. Some automatic design space explorations can even compare the benefits of the engines to choose the right one for a given network. Thus, it might be useful to implement multiple.

The systolic array has the advantage of very high clock speeds by limiting the combinational and data paths.

The advantage of the vector unit is the natural way of describing the relationship from input to output. The scalar values from the matrices are mapped directly as would be expected from the defining convolution equation.

The Winograd unit uses the Coppersmith-Winograd algorithm [25] to reduce the number of multiplications needed to perform matrix multiplications. In this case, used for convolution.

Because this research also focuses on readability and is a proof of concept the vector unit is used.

4.3.2.2 Fully connected layers

The fully connected layers, also called dense layers, are usually much less computationally intensive in CNNs. They are less computationally intensive because they perform the final steps of the classification process where less intermediary data needs to be processed. Because they need fewer computations, they could be computed on a processor.

Otherwise, a special predefined accelerator block can also be built for the fully connected layers, lastly, it is possible to make the convolutional layers flexible enough to handle the special case of a fully connected layer, as it can be seen as a special case of the convolutional layer, where the filter size is the image size, and the image is (usually) 1 dimensional. Using the convolutional processing element could become infeasible for fully connected layers if the fully connected layers are large. Large fully connected layers would request a large number of weights at the same time. For most convolutional networks the fully connected layers have a relatively small number of weights, thus the convolutional layer element can be used for the fully connected layers.

4.3.2.3 Activation functions

To support all possible activations the processor could be used, but this will require many data transfers. Otherwise, several commonly used activations can be implemented in one activation unit. Lastly, multiple activation units can be made, for the different activations that need to be supported.

Many of the activation functions use divisions and exponents, which are expensive operations in an FPGA, thus, lookup tables and approximations should be considered. The ReLU and Sigmoid are the minimum that should be implemented. More activation functions can easily be added.

4.3.2.4 Memory type and usage

Because many ANNs use large amounts of data, this data needs to be stored. Often there is not enough space within an FPGA to store the input, intermediary results and output, hence external memory is often a necessity.

External memory incurs more delays; therefore, it is desirable to manage the amount of external memory required and how often it needs to be accessed. The registers are the fastest memory type on the FPGA; however, the number of registers is limited. Because of the aforementioned, FPGAs also have internal block RAM, often available in larger amounts. Block RAM needs one clock cycle to retrieve the data and is thus marginally slower, than registers, but faster than external memory. It might be necessary to check for each storage element, what the size is and then select the appropriate storage type. If this analysis is not done, the system either always uses the larger option or the supported network sizes will be limited.

In this case, because it is a proof of concept, only the block RAM will be implemented. The block RAM balances the available size and difficulty of communicating with off-chip memory. No analysis will be needed.

4.3.3 How to build a compiler

To use the workflow, a compiler is required that takes the Keras specifications and turns it into a usable Clash project. The compiler can be implemented in any software language, but the most obvious options to consider are:

- Python
- Haskell
- C++

These are considered because many of the machine learning environments are based in Python and Clash is based in Haskell. Any other language could also be used. For example, many compilers are implemented in C++, because of its potential for low execution time. Haskell can also be compiled and could thus offer similar performance to a C++ implementation. Furthermore, Haskell is especially well suited for handling tree structures, which are often present when talking about compilers, as the abstract syntax tree (AST) is a common intermediate structure to represent an abstraction of a program.

In this case, Python was chosen because of the ease of prototyping. Its flexibility in how the language can be used makes prototyping easier.

4.4 Space-time trade-off interface

The goal of the system is to offer the developer flexibility and ease of implementing the time-area trade-off. Therefore, some user-friendly way of inputting some variables, which decide the area/resource usage should be implemented. A natural place to add these variables is at the top-level description of the network. This will be the point at which the developer adds the network to their project and can then immediately decide the amount of parallelism desired.

The network can be made scalable in multiple ways:

1. Multiple processing elements can be implemented per layer.
2. The vector processing units can be implemented over multiple clock cycles.
3. One layer processing element can work on multiple layers.

For each type of parallelism accessible to the developer, extra variables need to be added to the interface. For the proof of concept, it is important to choose the most effective strategy. The number of variables that must be chosen will also increase with more options, which could clutter the interface.

Multiple processing elements per layer offer large scale customizability. This is a viable option because it is an obvious place to implement the to start.

The parallelism offered by spreading the vector processing unit operations over time could equally be used to change the execution time of each layer. Making the parallelism of the vector processing units available to be changed by the developer is a useful addition, as it allows for more fine-tuning.

Allowing the developer to choose to let processing elements work on multiple layers can be useful if layers are similar enough. Because layers often do not behave the same using one element for multiple layers quickly becomes infeasible.

Only the multiplying of processing units for each layer is present in this proof of concept.

4.5 Automatic architecture analysis

The parallelism in the layers should be balanced, such that they have similar execution times. Some papers also include some analysis of architecture, which model the system and can then optimize for some cost function, an example in 3.1.1. This often includes balancing bandwidth and parallelism. Furthermore, the resource usage (especially the DSP blocks) and their efficiency (how much of the operational time they are active) are used to measure the effectiveness of an implementation. To implement this automatic design space exploration some model of the network is required, which can then be measured and show some optimal or desired parallelism. This was not within the scope of this research.

4.6 Design space exploration overview

An overview of the choices made in the design space exploration can be seen in. The path taken can be seen by being made bold.

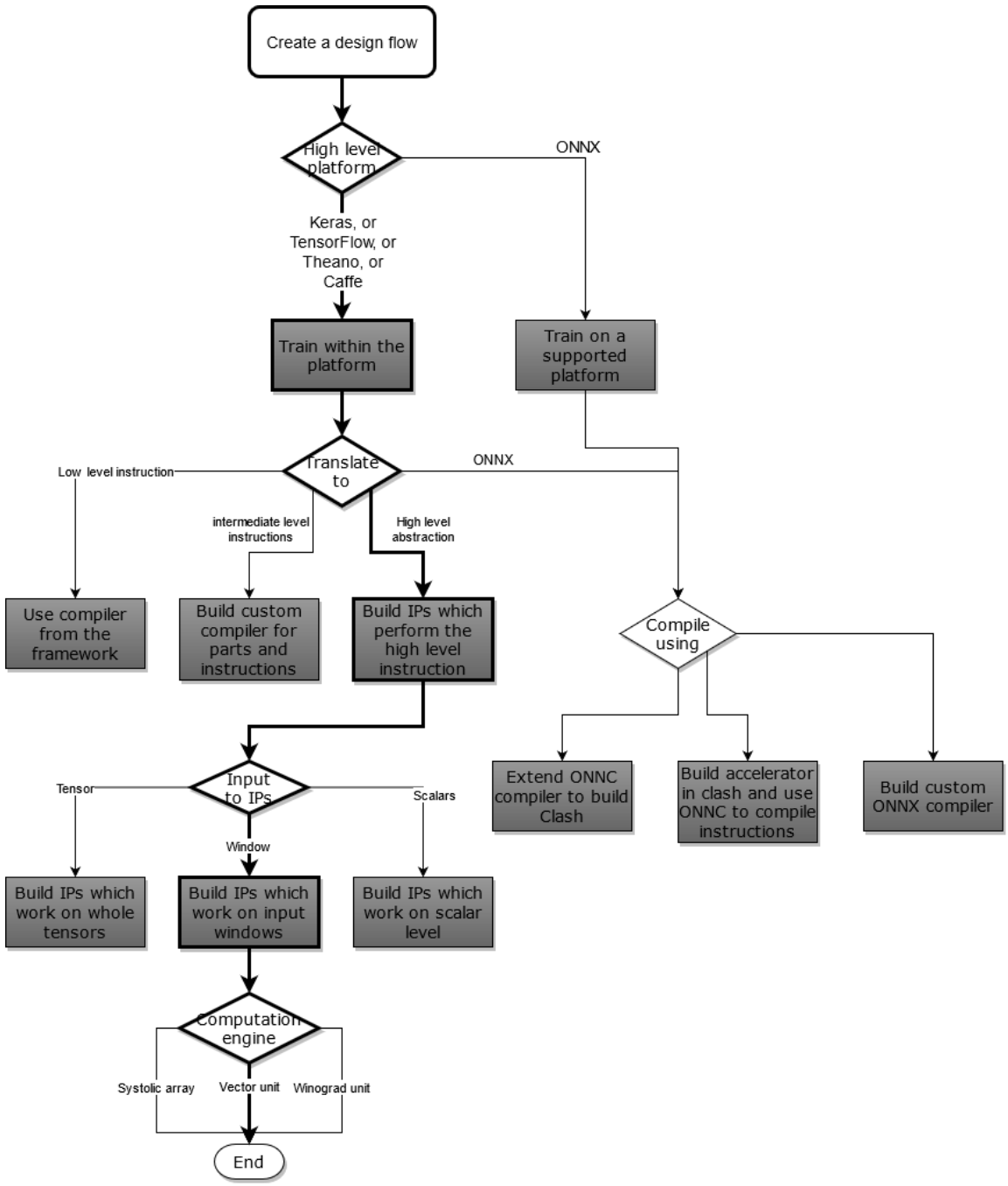

FIGURE 23 OVERVIEW OF THE DESIGN SPACE EXPLORATION OF THE CLASH IMPLEMENTATION

5 IMPLEMENTATION OF THE DESIGN FLOW

To test whether the Clash paradigm offers a better way of translating machine learning algorithms to FPGAs and other hardware implementations, a design flow has been developed with a compiler that translates networks from a commonly used machine learning platform to a general hardware implementation in Clash.

This will consist of a system in which the TensorFlow-Keras library can be used within python, to quickly build and train a network on available hardware. This trained network can then be translated to a general Clash implementation. The general Clash implementation can be included in a Clash project and the resource utilization can easily be changed by the developer.

This chapter will show the important implementation choices for this design flow and the Keras-to-Clash compiler.

5.1 Design flow overview

The proposed design flow for the user/developer is as follows:

1. The developer builds a network within the python Keras environment, which makes it easy to build, train and test network architectures.
2. This network can be translated to a Clash implementation.
3. The developer can optimize this Clash implementation to their system, which can be tested within this environment. To see if the output is correct, and how many clock cycles one inference takes.
4. The working system can be compiled to a lower-level hardware description language like VHDL, using the Clash compiler.
5. This implementation can be implemented on an FPGA or used for ASIC design.

5.2 Keras-to-Clash Compiler

To translate the high-level Keras implementation to a usable Clash implementation a compiler is required. The schematic overview of the steps taken by such a compiler can be seen in Figure 24. It starts with the Keras specification written in Python. This gets saved in two files: the configuration and the weights. These two files are read by the parser. The parser produces a list of layer descriptions. This list goes to the emitter, together with a fixed-point format. The emitter produces three files. These files can be included in a Clash project, from where they can be compiled by the Clash compiler to VHDL or Verilog. An example of these files and representations is shown in chapter 6.2 Case study of the new design flow.

5.2.1 Intermediate Representation within the Keras-to-Clash compiler

The Keras-to-Clash compiler will have some internal representation, an intermediate representation when translating from the high level to the Clash description. The translation could also work together with the compiler frameworks made for ONNX [10].

For many high-level specifications, a compiler exists for embedded systems. These implementations are often optimized for microcontrollers, which gives different optimizations than needed for an FPGA, and translating this C(++) implementation often involves solving data dependencies within loops, which can take an intractable amount of time.

Therefore, a custom IR is defined, this results in fewer dependencies. The custom IR only stores information important to this system, in a format that translates well to the implementation. The internal representation is a list of layer objects, implemented as Python dictionaries. Each layer can be agnostic of the other layers. This custom data format can also save development time in further stages, as the translation can be more direct. This custom representation will only hold the exact information needed for building the Clash implementation, the rest of the information, like the training method, can be removed. The dictionaries have different fields depending on the layer they describe, but they always have a “type” and an “input_shape”. The dense and convolutional layers also include their activation function, which is not implemented as a separate layer, following the standard from Keras.

5.2.2 Saving the Keras model

After successfully training a model, the Keras library is used to store the network configuration as a JSON file. While the python pickle library is used to serialize and store the array of weights corresponding to this network.

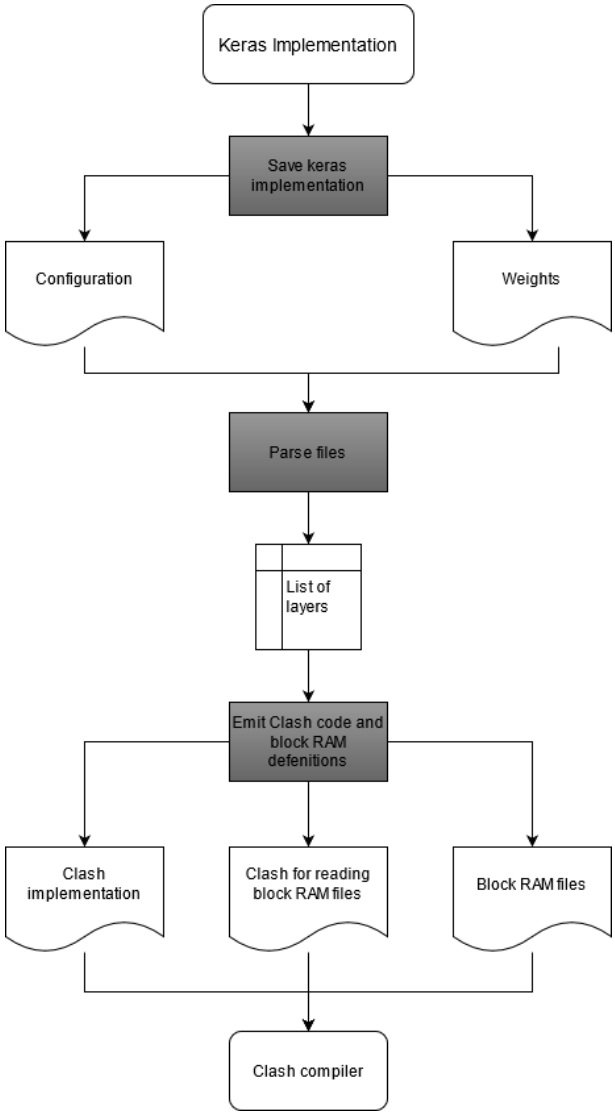

FIGURE 24 FLOW CHART OF THE COMPILER

5.2.3 Parser

The parser can read the contents of the Network configuration and weights file and produce a list object containing all the layers that are needed for the Clash implementation. This list contains the sequential layers, such as Convolutional, activation, and fully connected. These layers also contain the corresponding weight data.

The format of the layers is tuned to the needs of the emitter. But it does not contain any data on what will be implemented, if the emitter decides that certain layers from the list will be multiple components in the Clash implementation, this parser does not need to be aware.

5.2.4 Emitter

The emitter processes the list of layer objects and combines it with the correct predefined blocks, set with the correct values, such as window size. It also produces the block RAM binary files, which contain the weights in a format that can be loaded onto an FPGA. To store the weights, the data format must be decided at this point, up till this point, the weights have been represented following the standard of Keras, which is float16, float32, or float64. But in the FPGA, this will need to be represented as some fixed-point representation. This format is presented to the emitter as a tuple of (sign bit {0,1}, integer bits $\mathbb{N}$, fractional bits $\mathbb{N}$).

The emitter produces 3 types of files:

1. One Clash network description
2. Several binary weights files
3. One Clash file to read the weight files

The network description gets linked to the predefined blocks by the emitter. It also sets the correct parameters for each predefined block and combines all these blocks into one network predictor function.

The network (prediction) function is the interface for the developer to the machine-learned network. It is where the data is entered into the network, and the output predictions will be presented. At the same time, it presents the interface to the parallelism variables.

This network function can be included in any Clash project, where the network could make useful predictions.

The network function passes Natural numbers from the call to the predefined filter blocks, which instruct how many predefined Filter blocks should be instantiated in each layer. These Clash descriptions use Haskell abstractions like polymorphism and partial application, to define the network without fully specifying the parallelism. This can be done through its partial application. The “network” function takes the parallelism variables and then the input data, thus the parallelism variables can be passed resulting in a network function with defined parallelism.

An example of using the network function can be seen in Code Block 2. The network gets the parallelism of its three layers set to 1, 2 and 5, given by natural numbers: *d1*, *d2* and *d5* (line 5). The network gets included in the *topEntity* function.

```
1 topEntity :: (KnownDomain System)
2   => Clock System -> Reset System -> Enable System
3   -> Signal System (Maybe ((Pntr, Pntr), Vec 3 (Vec 3 (Vec 1 Repr))))
4   -> Signal System (Maybe ((Pntr, Pntr), Vec 1 (Vec 1 (Vec 10 Repr))))
5 topEntity clk rst ena input = exposeClockResetEnable (network d1 d2 d5) clk rst ena input
```

CODE BLOCK 2 NETWORK FUNCTION IMPLEMENTED AS THE TOP-LEVEL ENTITY

5.2.5 Compiling the network

The emitted files can be included in a project, the parallelism must be defined before it can be compiled. The width is given as a Natural number, that must be known at compile time.

5.3 Transparency in the output

The developer can access the building blocks of the network, the layers, separately instead of the network function. This would allow the developer to perform actions on intermediate results if he so desires.

To ensure that the generated code is usable by a developer, it should be readable. The developer is not shown too much of the internal structure. However, if they need to look, the structure should still be understandable. Thus, the naming of functions, parameters, and constants is important. The functions should not be too long or should be cut up into blocks that have clear objectives. A function should not get too many input variables. The project should be made of files named after their part of the system. Where possible the data transformations are expressed in higher-order functions, to provide clarity to the developer.

Therefore, the layers are named after their counterparts in the Keras library, “dense”, “conv2d”, “flatten” and “maxpooling2d”, with a layer identifier, an integer, separated by an underscore. This way the developer could choose to spread the layers over multiple FPGAs.

5.4 The Clash general implementations blocks

To build the Haskell implementation, several predefined blocks are created which form the building blocks for the ANN accelerator. The Keras-to-Clash compiler will use these building blocks to sequentially build the network in the same order as the network was defined. It will set the correct parameters construed from the IR.

The blocks used are discussed in this chapter. Each block uses internal registers to pipeline the data processed through the system, meaning all blocks can each work on one sample at the same time. To synchronize the blocks the *Maybe* typeclass from Haskell is leveraged. When no data is available *Nothing* is produced, and the following blocks will receive *Nothing*. When *Nothing* is received, the block does not need to start processing new data.

5.4.1 Filter

The filter predefined block performs the multiplications and additions required for both convolutional and dense layers, in Keras these layers are built using *keras.layers.Conv2D*, for a 2D convolution. The implementation in Clash can be seen in Appendix A. Looking at line 86 from this implementation shown simplified in Code Block 3 we see how higher-order functions can be used to create multiple processing elements. The number of processing elements can be set when implementing the block by making the elements map over the input while splitting the weights into equal parts. In this example, the natural number passed through *splits* conveys the number of processing elements present, by presenting that many copies of the input.

The processing elements each have predefined states, defined using Haskell datatype, which will be implemented efficiently by the Clash compiler. The Moore machine can be used to implement this state machine, while describing the combinatorics, that performs the data processing, without state.

```
1 outputs = imapA filterUnit (replicate splits input)
```

CODE BLOCK 3 USING TEH HIGHER-ORDER FUNCTION IMAPA TO USE A VARIABLE NUMBER OF PROCESSING ELEMENTS

The polymorphism of Clash allows the filter to be defined without setting the window input size. This definition of the filter unit is independent of the window size for the specific implementation.

A schematic representation can be seen in Figure 25. In the shown example 3 filter Processing elements are instantiated, which all take 1/3 of the filters from memory. Because they have 1/3 of the filters, they will produce 1/3 of the output features. These partial output features are combined into the output “pixel” (a 1-by-1 window of the output tensor), which will be transferred to the following block of the system.

The amount of filter processing elements is related to the depth of the layer. If the layers do not process an equal amount of data, control must be built in to handle the behaviour and data that do not match in time and space. For the proof of concept, limiting the number of processing elements that can be set to a divisor of the number of filters is adequate. Limiting the possible parallelism factors means no hardware will be wasted on processing dummy data and it limits the amount of control logic necessary.

5.4.2 Pooling

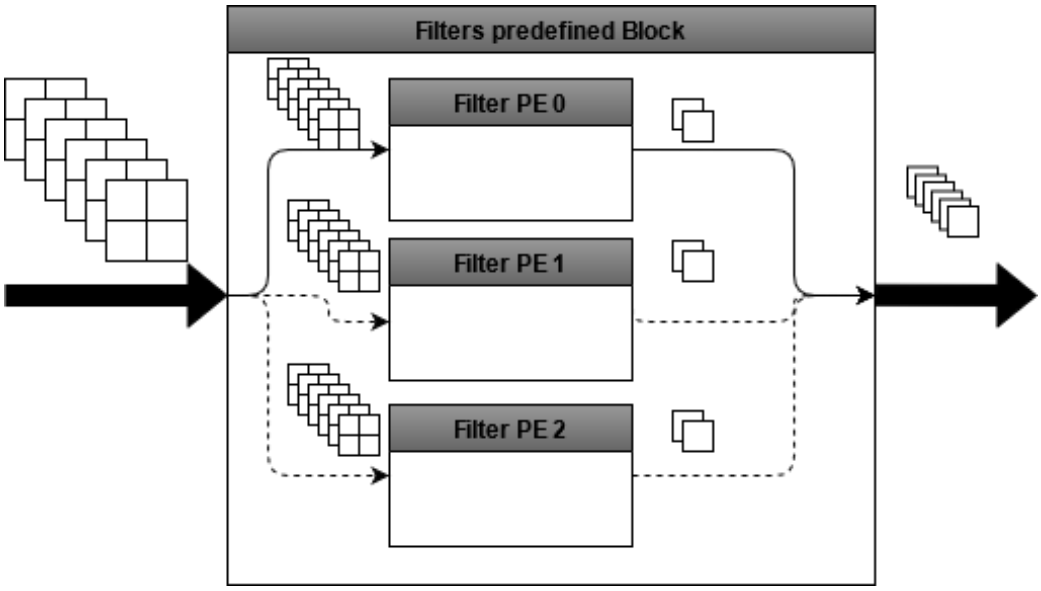

FIGURE 25 SCHEMATIC OF THE FILTERS PREDEFINED BLOCK WITH 3 FILTER PROCESSING ELEMENTS IMPLEMENTED

The Pooler predefined block can be one of two implementations, the average pooler, *avgPooler*, or the maximum pooler, *maxPooler*. They both calculate the maximum or average per feature, but this can be done very concisely as can be seen in Code Block 4. In lines 2-11 the avgPooler can be seen, the signature is similar to the maxPooler, lines 14-21, except, it requires the window size to calculate the average. In this case, the triple fmap (<<<$>>>) is used, because the average and max pooler apply to 2D matrices. The result is buffered in a register, to avoid timing issues, the average and max functions are defined within their appropriate predefined block, line 6-11 and line 18-21. The schematic overview

can be seen in Figure 26. The polymorphism of Clash makes sure that the pooling can be performed on any size window that comes into the processing blocks. Where in the schematic, the representation a number to draw it, this is not required in the code implementation. The input is treated as a vector of a vector, which can be concatenated and mapped over, regardless of size and depth. Such a pooling layer corresponds to the Keras function *keras.layers.MaxPooling2D* or *keras.layers.AveragePooling2D*.

```
1  -- the average Pooler predefined block
2  avgPooler windowSize i = o
3    where
4      o = register def $ avgWindow windowSize <<<$>>> i
5      -- function to find the average
6      avgWindow windowSize@(width, height) wind = avgs
7        where
8          size = snatToNum (width `mulSNat` height)
9          avgs = map (/size) sums
10         sums = map (foldl (+) 0 . concat) transposed
11         transposed = transpose $ map transpose wind
12
13 -- the maximum Pooler predefined block
14 maxPooler i = o
15   where
16     o = register def $ maxWindow <<<$>>> i
17     -- function to find the maxima
18     maxWindow wind = maxs
19       where
20         maxs = map (fold max . concat) transposed
21           transposed = transpose $ map transpose wind
```

CODE BLOCK 4 CLASH IMPLEMENTATION OF THE POOLER PREDEFINED BLOCK

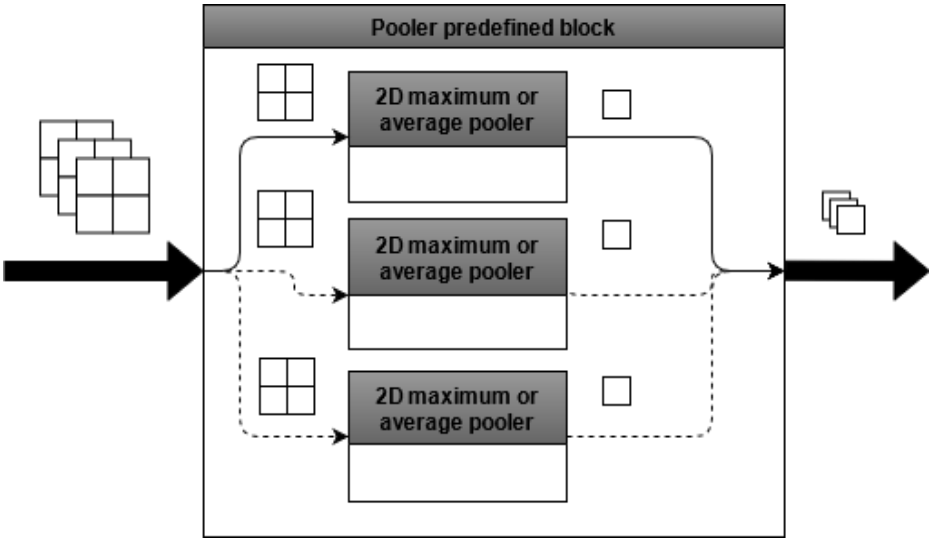

FIGURE 26 SCHEMATIC OVERVIEW OF THE POOLER PREDEFINED BLOCKS

5.4.3 Activation

The activation predefined block is a block that takes a function and applies it to the data from the input. The current implementation does not support the softmax function, thus if it is to be implemented, a different predefined block needs to be made. It supports the following activation functions: linear, ReLU, a logistic approximation from the authors of [26].

The implementation of the activation predefined block can be seen in Code Block 5. Where the window coming in gets concatenated to a 1D vector. After that, the *singleton* function is used twice two give it the same dimensionality as the window. As an example, an input of a matrix with dimensions

3x3x16 gets transformed to a matrix with dimensions 1x1x144. All of these steps can be taken by Clash's folds, which can work on any size vector, therefore not needing size to be set before it is implemented and gets a defined size.

In lines 6-7 the linear activation is implemented; the output x is the same as the input. In lines 9-10 the ReLU is implemented when the input is smaller than 0 the output is zero, if the input is larger than zero the output is the same as the linear activation.

An approximation is made of the logistic function and this is used as the sigmoidal activation function. This can be seen in 16-28 and 30-31 respectively. The softmax is approximated by the sigmoidal function, as can be seen in 34-35. The body of the activation function line 44-46 shows how the activation function can be applied to all the scalars in the tensor. A quadruple fmap (<<<<$>>>>) is used. This is because the scalars the activation need to be applied to are in four functor wrappers: *Signal dom*, *tuple*, first vector dimension and second vector dimension. Haskell functor typeclass allows the operations to be done on the data without unwrapping all the layers of abstraction.

The result is stored in a register, to prevent timing issues with the following layers.

The softmax function is not yet fully supported. In inference, it can be replaced by a sigmoidal function. The softmax helps in training as it strengthens the difference between the outputs, making clearer to the training algorithm whether the network is making correct predictions. But the largest result can still be taken as the prediction when using a not normalized output.

5.4.4 Memory

Between the predefined blocks, the tensors sometimes need to be stored to collect the windows for the next layer. The output of the blocks often needs to be combined before the next block can work on them.

The memory currently only supports using block RAM. When instantiating the memory predefined block, the compiler knows the size of the tensor it must store. The tensor is stored in a linear block ram where each address stores one "pixel". Because the system transfers the 2d addresses of the pixels, these 2D coordinates can be consistently translated to the memory index.

The schematic overview of the Memory controller predefined block can be seen in Figure 27. The data input is a tuple of an address ((x_i, y_i)) and a pixel. The address gets translated to a linear index of the memory and the pixel is stored at that location. The valid memory address vector is updated. The valid memory address vector contains all the addresses of the block RAM that currently hold valid data.

The window builder requests the pixel values from the Block RAM if they are valid according to the

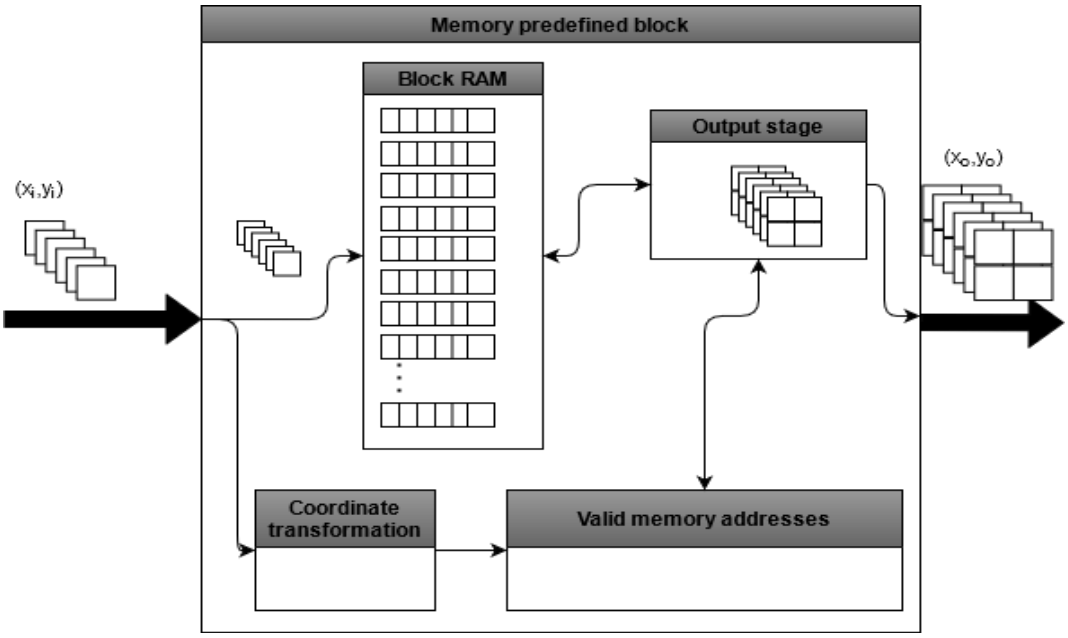

FIGURE 27 SCHEMATIC OVERVIEW OF THE MEMORY PREDEFINED BLOCK

valid memory address vector. The window gets build in the *output stage*, which will output a complete window when it is filled. It builds the windows in sequence from top left to bottom right, idling when the pixels are not yet available.

```haskell
module Convolutional.Activation where
import Clash.Prelude

import Tools.Tools

lin :: (Num a) => a -> a
lin x = x

relu :: (Ord a, Num a) => a -> a
relu x = max 0 x

-
- from https://www.researchgate.net/publication/228618304_Digital_Implementation_of_The_Sigmoid_Function_for_FPGA_Circuits
-- and https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=646812
planLogistic :: (Ord a, Bounded a, Fractional a, Bits a) => a -> a
planLogistic x | (abs' x) < 1     = pos $ 0.5     + shiftR (abs x) 2
               | (abs' x) < 2.375 = pos $ 0.625   + shiftR (abs x) 3
               | (abs' x) < 5     = pos $ 0.84375 + shiftR (abs x) 5
               | otherwise        = pos $ 1
               where
                 pos = if x < 0 then (1-) else (0+)
                 -- abs of minBound is undefined,
                 -- but in this case, it is okay to take the next value,
                 -- as the difference of the result at the end of the range
                 -- is minimal/zero
                 abs' y | y == minBound = maxBound
                        | otherwise = abs y

sigmoid :: (Ord a, Bounded a, Fractional a, Bits a) => a -> a
sigmoid = planLogistic

-- because softmax is actually quite expensive, replace it with sigmoid
softmax :: (Ord a, Bounded a, Fractional a, Bits a) => a -> a
softmax = sigmoid

--------------------------------------------------------------------------------
activationUnit :: (HiddenClockResetEnable dom,

  NFDataX (f2 (f3 (f4 b))), Default (f2 (f3 (f4 b))), Functor f2,
  Functor f3, Functor f4)
  => (a -> b)
  -> Signal dom (f2 (f3 (f4 a))) -> Signal dom (f2 (f3 (f4 b)))
activationUnit act_f i = o
  where
      o = register def (act_f <<<<$>>>> i)
```

CODE BLOCK 5 CLASH IMPLEMENTATION OF THE ACTIVATION PREDEFINED BLOCK

5.4.5 Flattening

To “flatten” the output of a convolutional layer to prepare it for a fully connected layer, means to change the multidimensional matrix to a 1D representation. This is done with the *flatten* predefined block as can be seen in Code Block 7. Only the description of the data changes, but in hardware, no transformations happen. The transformation happens in line 11, where it is also stored in a register.

5.4.6 Indexing

Because the windows are streamed through most of the predefined blocks, the structure of the tensors is lost. Hence, alongside each streamed window, coordinates that describe the position in the tensor are passed along. All predefined blocks receive a tuple of the address and the data and can manipulate either or both or pass both indices on, depending on their task. An example of the type description of a packet can be seen in Code Block 6.

```
Signal dom (Maybe (address, (Vec x (Vec y (Vec n (SFixed 8 8))))))
```

CODE BLOCK 6 EXAMPLE TYPE DESCRIPTION OF A PACKET BETWEEN PREDEFINED BLOCKS

5.4.7 No input

When there is no data is available, the predefined block should not start producing nonsensical data, or get into undefined states. Therefore, the streamed data is always wrapped in a *Maybe* construct. The Maybe comes from Haskell and is in the functor and applicative typeclass. These typeclasses assist in handling this type of data and can in this case be used to instruct whether data should be processed. When a predefined block receives valid data, it will operate on it and produce valid output. Valid data in the Maybe construct is represented as *Just data*, invalid data is represented as *Nothing*.

```
1  -- The flatten units receive a:
2  -- Signal Maybe (address, vec X (vec Y Vec N Repr))
3  flattenUnit :: (HiddenClockResetEnable dom,
4    NFDataX (f2 (f3 (Vec 1 (Vec 1 (Vec ((x * y) * n) a))))),
5    Default (f2 (f3 (Vec 1 (Vec 1 (Vec ((x * y) * n) a))))),
6    Functor f2, Functor f3)
7    => Signal dom (f2 (f3 (Vec x (Vec y (Vec n a)))))
8    -> Signal dom (f2 (f3 (Vec 1 (Vec 1 (Vec ((x * y) * n) a)))))
9  flattenUnit i = o
10   where
11       o = register def ( (singleton.singleton.concat.concat) <<<$>>> i)
```

CODE BLOCK 7 CLASH IMPLEMENTATION OF THE FLATTEN PREDEFINED BLOCK

6 RESULTS

A proof-of-concept design flow was made, based on the choices presented in Chapters 4 and 5. Our Keras-to-Clash compiler translates a Keras network description, such as shown in Code Block 8, to a scalable Clash description.

6.1 Resulting design flow

The new design flow including the Keras-to-Clash compiler, allows the developer to build and train a model in the Python & Keras environment. This model can be saved and loaded into the parse function of the parse module. The parser will take the *keras_model* and produce the IR (intermediate representation). The emitter processes the IR into Clash and block RAM files. These files can be included in a Clash project and compiled. These steps are shown in a flow chart in Figure 28

The modules implemented in Python for the compiler can be seen in the class diagram in Figure 29.

6.1.1 Modules of the design flow

The model trainer is an abstract class that gives a name to a network and has a train model function. The Keras implementation

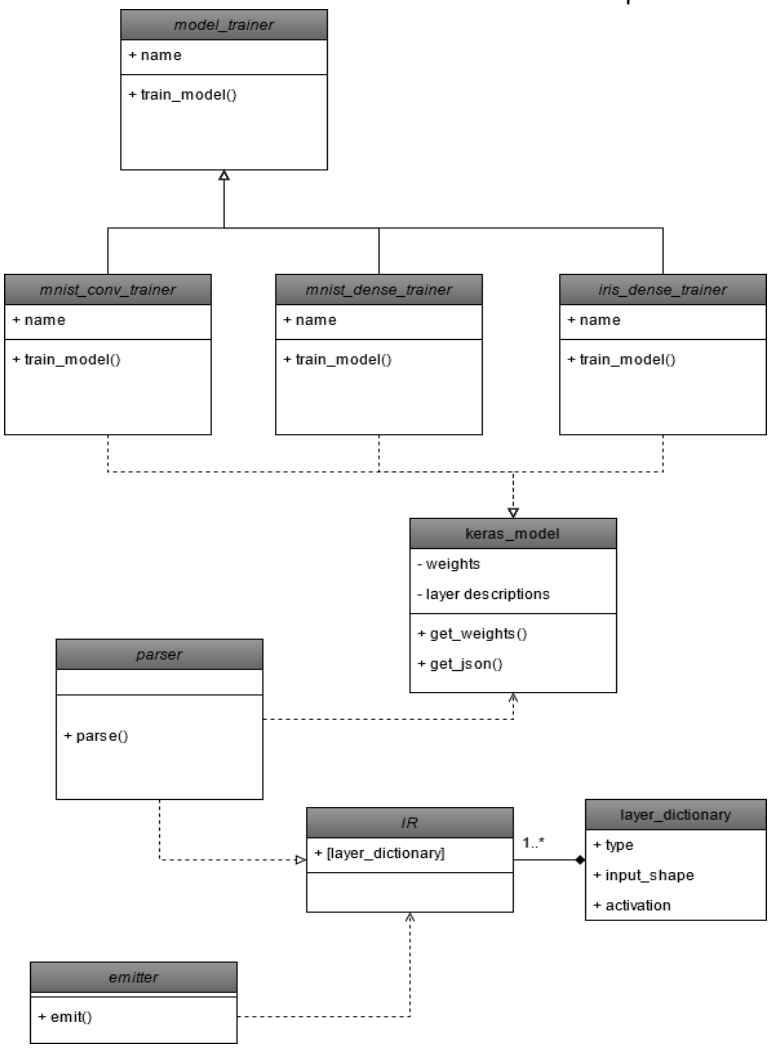

FIGURE 29 CLASS DIAGRAM FOR THE KERAS TO CLASH DESIGN FLOW

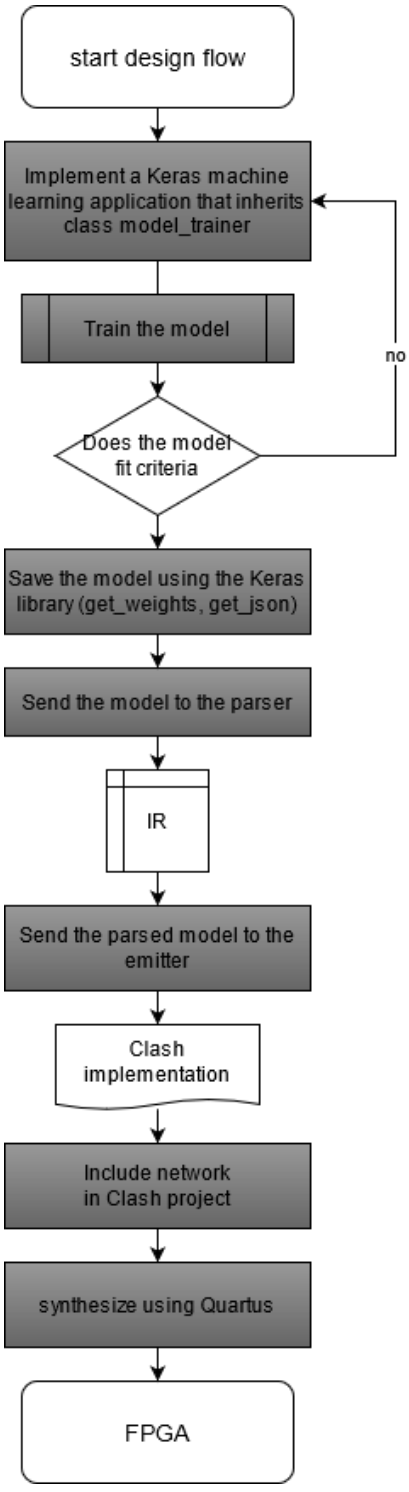

FIGURE 28 RESULTING DESIGN FLOW CHART

of a network can implement the train_model function, such that it can be accessed by the compiler. The train_model function should return a Keras model. Some example implementations in Figure 29 include the MNIST CNN, MNIST Dense and a dense iris recognition network.

6.2 Case study of the new design flow

6.2.1 Building a network

The first step of using the new design flow is building a machine learning application using Keras in Python. The application can be wrapped in a *train_model* function, to be accessible from outside the module.

In Code Block 8, an example machine learning application is shown. In line 2, the training data is loaded together with the test data, input shape, and the number of prediction classes. The loading function will handle any necessary pre-processing of the data. In this case, the MNIST set of handwritten digits is loaded. An example is shown in Figure 30. The input shape of this dataset is (9,9,1) because the images are scaled down from the original (28,28,1). The number of classes is 10, because the digits are encoded as a vector of 10 values, where 0 is represented by [1,0,0,0,0,0,0,0,0,0], 1 is represented by [0,1,0,0,0,0,0,0,0,0], 2 by [0,0,1,0,0,0,0,0,0,0], and so on.

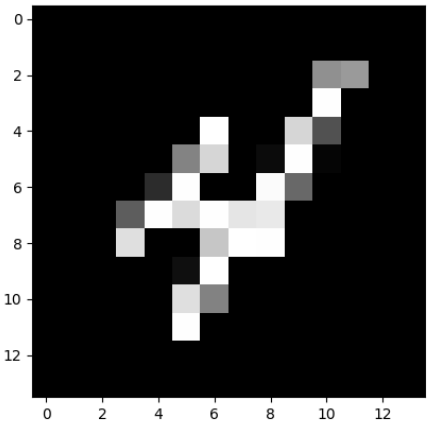

FIGURE 30 EXAMPLE OF THE DOWNSCALED MNIST DATASET OF WRITTEN DIGIT [29]

The sequential model is instantiated (from line 3) and the layers are defined (lines 4-13). The dropout layer, line 11, is used during training only. Dropout layers set certain activations to zero randomly, to increase the robustness of the trained network. These layers can be ignored in the compilation step.

```
1  def train_model():
2    (x_train, y_train), (x_test, y_test), (input_shape, num_classes) = load_data()
3
4    model = keras.Sequential(    [
5      keras.Input(shape=input_shape),
6      layers.Conv2D(16, kernel_size=(3, 3), activation="relu"),
7      layers.MaxPooling2D(pool_size=(2, 2)),
8      layers.Conv2D(16, kernel_size=(3, 3), activation="relu"),
9      layers.MaxPooling2D(pool_size=(2, 2)),
10     layers.Flatten(),
11     layers.Dropout(0.5),
12     layers.Dense(num_classes, activation="sigmoid"),
13     ]
14   )
15
16   model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
17   model.fit(x_train, y_train, batch_size=128, epochs=200, validation_split=0.1)
18   return model
```

CODE BLOCK 8 KERAS DNN FOR RECOGNIZING THE MNIST DATASET

6.2.2 Training the network

This network was trained using TensorFlow-Keras on a workstation. The Keras library is used to define which loss function is used and which optimizer to use (line 12). Then the *fit* method from Keras is used to train the model on the training data. The network training does not keep the weights within a range. Constraints could be given, such as non-negativity or a minimum and maximum value. If the format used later on is chosen beforehand, the weights can be limited to be within the range supported by the format. A weights distribution for a dense implementation is shown in Figure 32. This weight distribution can be used to estimate a useful format.

```
1   Name = "MNISTConvNetwork"
2   model = train_model(name=name)
3
4   ksave.save_network(model, name=name, blockramfile=True)
5
6   with open("config.json", "r") as jsonfile:
7       contents = jsonfile.read()
8       network_dict = json.loads(contents)
9   with open("config.obj", "rb") as weightsfile:
10      network_weights = pickle.load(weightsfile)
11
12   network = parse(network_dict, network_weights)
13  (network_text, weights_text, blockram_texts) = emit(network, name, format=(1,7,8))
```

CODE BLOCK 9 SAVING A NETWORK, LOADING IT FROM FILES AND PARSING IT

6.2.3 Compilation

In Code Block 9, the *train_model* function is used to create a Keras model (line 2). This model gets saved with the *save_network* function (line 4). By saving the network, it can be reloaded and retrained, or compiled with different settings. In this case, the files are immediately read in lines 6-10. The internal representation from Keras is shown in Code Block 11. It is a JSON file, which holds name-value pairs. Besides the JSON file, a binary file containing the weight values is also stored in floating-point format.

These files get read for parsing. The loaded network was passed to the Keras-to-Clash compiler. The files first get parsed, by the *parse* function from the parser module. This result in the IR object: *network*, line 12. This network is passed to the emitter, together with a name for the network and the format to save the weights in (line 13). The format in this example is 1 sign bit, 7 integer bits, and 8 decimal bits. The emitter produces two Haskell file texts and a block RAM file per layer.

The *network_text* is a Haskell file, which is shown in Appendix B. This file defines the building blocks for the network, using the predefined blocks. Multiple predefined blocks are used for some of the layers while others layers are made up of one predefined block. The predefined blocks get stacked using function composition (signified by a dot) as can be seen in Code Block 10. When using function composition the first layer is on the right. We can see from right to left how the network processes an image. The image gets flattened by *flatten_0*, then *dense_1* will process the flattened image. The parallelism of *dense_1* can be set later. After the first dense layer, two more dense layers follow, for each the parallelism can be set by the developer.

```
1   network width1 width2 width3 = (dense_3 width3).(dense_2 width2).(dense_1 width1).flatten_0
```

CODE BLOCK 10 USING FUNCTION COMPOSITION TO STACK THE LAYERS OF A NETWORK


```
1   {"class_name": "Sequential", "config": {"name": "sequential", "layers": [{"class_name": "In
2   putLayer", "config": {"batch_input_shape": [null, 14, 14, 1], "dtype": "float32", "sparse":
3    false, "ragged": false, "name": "input_1"}}, {"class_name": "Conv2D", "config": {"name": "
4   conv2d", "trainable": true, "dtype": "float32", "filters": 16, "kernel_size": [3, 3], "stri
5   des": [1, 1], "padding": "valid", "data_format": "channels_last", "dilation_rate": [1, 1],
6   "groups": 1, "activation": "relu", "use_bias": true, "kernel_initializer": {"class_name": "
7   GlorotUniform", "config": {"seed": null, "dtype": "float32"}}, "bias_initializer": {"class_
8   name": "Zeros", "config": {"dtype": "float32"}}, "kernel_regularizer": null, "bias_regulari
9   zer": null, "activity_regularizer": null, "kernel_constraint": null, "bias_constraint": nul
10  l}}, {"class_name": "MaxPooling2D", "config": {"name": "max_pooling2d", "trainable": true,
11  "dtype": "float32", "pool_size": [2, 2], "padding": "valid", "strides": [2, 2], "data_forma
12  t": "channels_last"}}, {"class_name": "Conv2D", "config": {"name": "conv2d_1", "trainable":
13   true, "dtype": "float32", "filters": 16, "kernel_size": [3, 3], "strides": [1, 1], "paddin
14  g": "valid", "data_format": "channels_last", "dilation_rate": [1, 1], "groups": 1, "activat
15  ion": "relu", "use_bias": true, "kernel_initializer": {"class_name": "GlorotUniform", "conf
16  ig": {"seed": null, "dtype": "float32"}}, "bias_initializer": {"class_name": "Zeros", "conf
```

```
17 ig": {"dtype": "float32"}}, "kernel_regularizer": null, "bias_regularizer": null, "activity
18 _regularizer": null, "kernel_constraint": null, "bias_constraint": null}}, {"class_name": "
19 MaxPooling2D", "config": {"name": "max_pooling2d_1", "trainable": true, "dtype": "float32",
20  "pool_size": [2, 2], "padding": "valid", "strides": [2, 2], "data_format": "channels_last"
21 }}, {"class_name": "Flatten", "config": {"name": "flatten", "trainable": true, "dtype": "fl
22 oat32", "data_format": "channels_last"}}, {"class_name": "Dropout", "config": {"name": "dro
23 pout", "trainable": true, "dtype": "float32", "rate": 0.5, "noise_shape": null, "seed": nul
24 l}}, {"class_name": "Dense", "config": {"name": "dense", "trainable": true, "dtype": "float
25 32", "units": 10, "activation": "sigmoid", "use_bias": true, "kernel_initializer": {"class_
26 name": "GlorotUniform", "config": {"seed": null, "dtype": "float32"}}, "bias_initializer":
27 {"class_name": "Zeros", "config": {"dtype": "float32"}}, "kernel_regularizer": null, "bias_
28 regularizer": null, "activity_regularizer": null, "kernel_constraint": null, "bias_constrai
29 nt": null}}]}, "keras_version": "2.4.0", "backend": "tensorflow"}
```

CODE BLOCK 11 INTERNAL HIGH-LEVEL REPRESENTATION OF KERAS

6.2.4 Synthesis

The network function created in Code Block 10 can be implemented in a Clash project such as in Code Block 12. This is an otherwise empty project where the prediction of the network is not used. This project can be synthesised by Intel Quartus Prime to produce the network shown in Figure 31. The RTL netlist corresponds nicely to the network implementation from Code Block 10, adding to the readability of the hardware implementation. Each of the layers is visible in the RTL netlist as well as in the network implementation.

In Code Block 12, line 5 imports the output file of the Keras-to-Clash compiler containing the network function. This file also carries the data formats for the values in the block RAM (*Repr*) and the format of the coordinates (*Pntr*).

In this project, the parallelism for the network is set to 1 for the first layer, 2 for the second layer and 2 for the third layer (line 10). The input to this system is a window of a downscaled version of an image from MNIST. The windows have size 3-by-3 and have 1 channel (line 10). The network outputs a vector with 10 values which predicts which of the digits (0-9) is shown in the input image (line 11).

```
1  module Test where
2  import Clash.Prelude
3  import qualified Data.List as L
4
5  import MNISTNetwork (network, Repr, Pntr)
6
7  topEntity :: (KnownDomain System)
8    => Clock System -> Reset System -> Enable System
9    -> Signal System (Maybe ((Pntr, Pntr), Vec 3 (Vec 3 (Vec 1 Repr))))
10   -> Signal System (Maybe ((Pntr, Pntr), Vec 1 (Vec 1 (Vec 10 Repr))))
11 topEntity clk rst ena i = exposeClockResetEnable (network d1 d2 d2) clk rst ena i
```

CODE BLOCK 12 INCORPORATING THE PRODUCED NETWORK IN A CLASH PROJECT

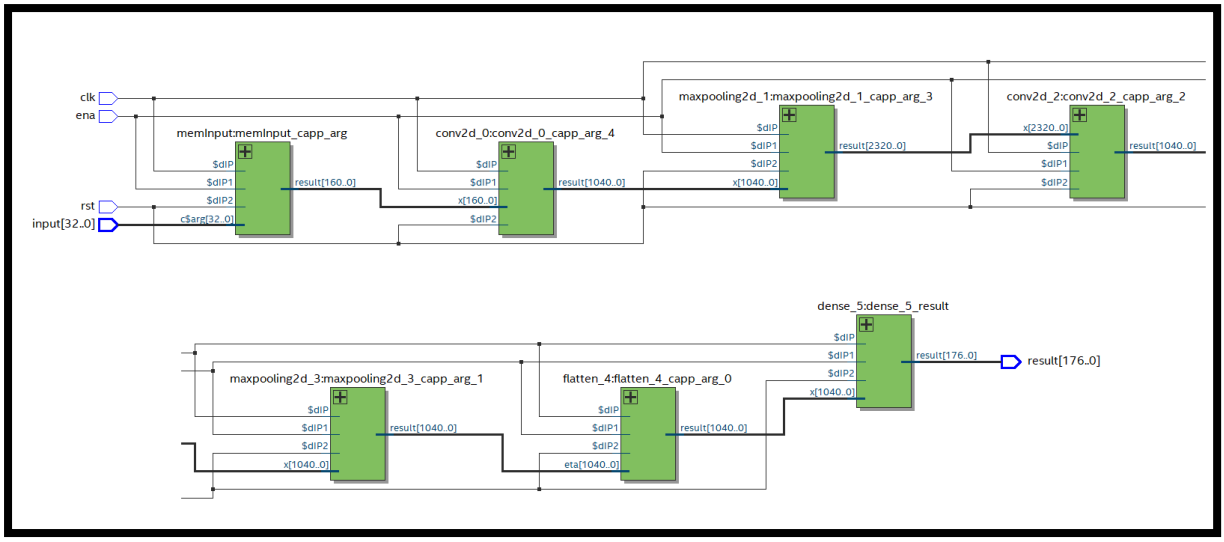

FIGURE 31 QUARTUS RTL NETLIST OF THE MNIST TEST NETWORK

6.3 Simulation results

The network produced by the Keras-to-Clash compiler can make predictions within the Clash interactive environment, to test the accuracy of the compiled design. The simulation can also be used to measure the number of clock cycles used to calculate one result.

6.3.1 Resource utilization

The DSP usage was tested by synthesizing the architecture with Intel Quartus Prime. When the fixed-point representation is set to (1,7,8), each filter unit uses one DSP block per input. Because the parallelism can be controlled per layer with an integer that is a divisor of the number of neurons, this results in the following DSP block usage, presented in Table 2, for the convolutional network. The parallelism available depends on the number of neurons/filters in the layer. For the current design flow, only divisors of this number can be used for parallelism. These constraints produce the table of possible options per network.

6.3.2 Latency and throughput

The results of the simulations for the convolutional network can be seen in Table 1, a subset of the possible configurations is simulated. The throughput of the system will be slightly higher than the figures in this table suggest, as the network can work on multiple samples at the same time due to the pipelining. The effects of this are very limited though, as for one output many samples need to be processed. Thus, only when all samples of one image have been processed by the first layer, can processing of a second image be started.

The networks are simulated in the Clash interactive environment until an output that is not *Nothing* is presented. This can be seen as the latency, from the start of processing an image till an output is computed. Different configurations can take in samples at different frequencies. For each network, the maximum sample frequency is used to perform the simulation. The period between the samples is given in clock cycles between the samples.

TABLE 1 CLOCK CYCLES FROM START TO FINISH WHEN PROCESSING ONE IMAGE

Parallelism in layer 1	Parallelism in layer 2	Parallelism in layer 3	Time between samples	Total clock cycles	DSP usage
1	1	1	18	2804	217
2	2	2	10	1631	434
4	4	2	6	1047	740
8	8	5	4	755	1544

TABLE 2 AVAILABLE DSP BLOCK USAGE FOR THE CONVOLUTIONAL MNIST RECOGNITION NETWORK

layer 1 with 16 filters chosen parallelism	DSP blocks for layer 1	layer 2 with 16 filters chosen parallelism	DSP blocks for layer 2	layer 5 with 10 neurons chosen parallelism	DSP blocks for layer 5	**total (DSP blocks)**
1	9	1	144	1	64	**217**
1	9	1	144	2	128	**281**
1	9	1	144	5	320	**473**
1	9	2	288	1	64	**361**
1	9	2	288	2	128	**425**
1	9	2	288	5	320	**617**
1	9	4	576	1	64	**649**
1	9	4	576	2	128	**713**
1	9	4	576	5	320	**905**
1	9	8	1152	1	64	**1225**
1	9	8	1152	2	128	**1289**
1	9	8	1152	5	320	**1481**
2	18	1	144	1	64	**226**
2	18	1	144	2	128	**290**
2	18	1	144	5	320	**482**
2	18	2	288	1	64	**370**
2	18	2	288	2	128	**434**
2	18	2	288	5	320	**626**
2	18	4	576	1	64	**658**
2	18	4	576	2	128	**722**
2	18	4	576	5	320	**914**
2	18	8	1152	1	64	**1234**
2	18	8	1152	2	128	**1298**
2	18	8	1152	5	320	**1490**
4	36	1	144	1	64	**244**
4	36	1	144	2	128	**308**
4	36	1	144	5	320	**500**
4	36	2	288	1	64	**388**
4	36	2	288	2	128	**452**
4	36	2	288	5	320	**644**
4	36	4	576	1	64	**676**
4	36	4	576	2	128	**740**
4	36	4	576	5	320	**932**
4	36	8	1152	1	64	**1252**
4	36	8	1152	2	128	**1316**
4	36	8	1152	5	320	**1508**
8	72	1	144	1	64	**280**
8	72	1	144	2	128	**344**
8	72	1	144	5	320	**536**
8	72	2	288	1	64	**424**

8	72	2	288	2	128	**488**
8	72	2	288	5	320	**680**
8	72	4	576	1	64	**712**
8	72	4	576	2	128	**776**
8	72	4	576	5	320	**968**
8	72	8	1152	1	64	**1288**
8	72	8	1152	2	128	**1352**
8	72	8	1152	5	320	**1544**

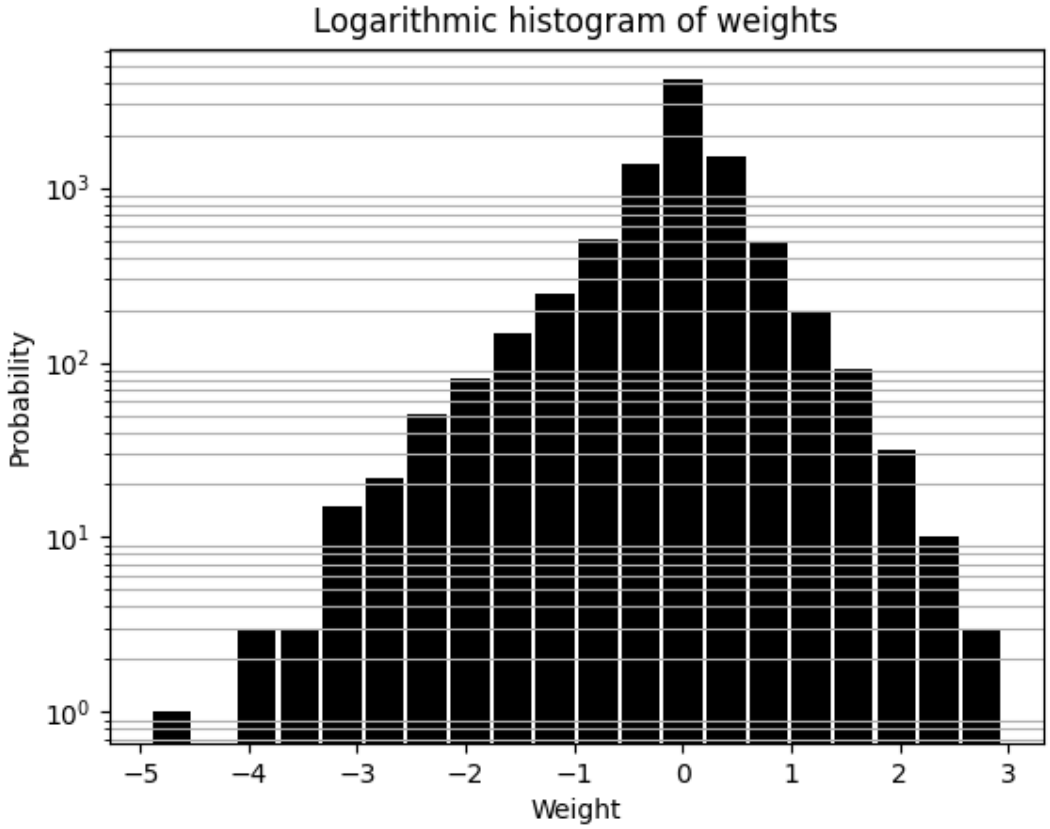

FIGURE 32 HISTOGRAM OF WEIGHTS IN AN MNIST ANN

6.4 Bit-width compared to accuracy

6.4.1 Experiment

The compiler can use different fixed-width weight representations. The bit-width will affect the accuracy of the resulting network. To get an idea of the effect of using different bit widths in a design, the effect will be measured.

A dense network for predicting MNIST digits was trained using Keras. A histogram of the weights present in the trained network is shown in Figure 32. As the format used gets smaller, the range of representable weights will decrease. From the histogram, we can see that most weights are close to zero, meaning that few weights will be clipped when the format gets smaller.

This effect is measured by compiling a dense network with various bit-widths, namely

6.4.2 Results

The resulting accuracies can be seen in Table 3 and Figure 33. Bit widths from 6 to 34 were measured, while the Keras library used floating-point numbers with 32 bits.

Because rounding the predictions can result in accuracy loss, we compare the accuracy of the network to the accuracy of the original network, but with the predictions rounded to the fixed point

format. If the difference between two values is smaller than can be represented by the format, just rounding can result in an incorrect prediction. The accuracy of the rounded predictions is shown next to the accuracy of the network.

TABLE 3 ACCURACY DEPENDING ON BIT-WIDTH

Format	Total bits	Accuracy after rounding	Accuracy
Keras	32	-	0.966
1,16,17	34	0.967	0.911
1,15,16	32	0.967	0.911
1,14,15	30	0.967	0.912
1,13,14	28	0.966	0.909
1,12,13	26	0.964	0.909
1,11,12	24	0.963	0.908
1,10,11	22	0.963	0.912
1,9,10	20	0.96	0.901
1,8,9	18	0.956	0.899
1,7,8	16	0.95	0.888
1,6,7	14	0.936	0.793
1,5,6	12	0.917	0.487
1,4,5	10	0.901	0.243
1,3,4	8	0.862	0.101
1,2,3	6	0.824	0.084

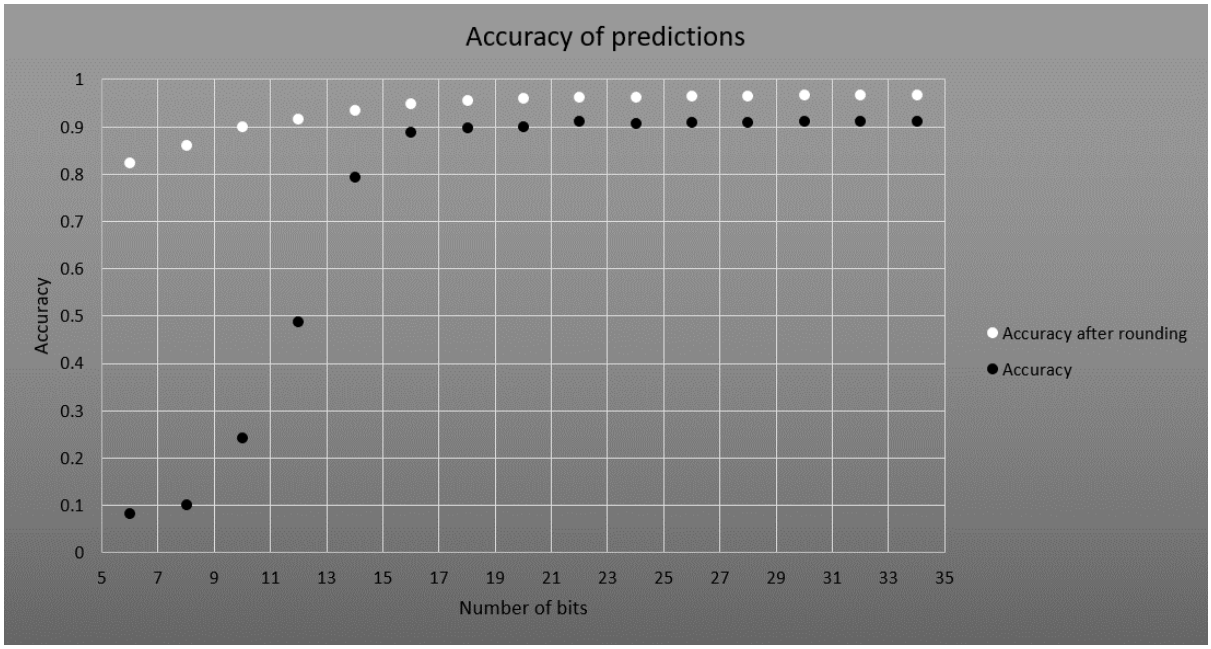

FIGURE 33 GRAPH OF ACCURACY DEPENDING ON BIT-WIDTH

7 CONCLUSION

To definitively answer the main question, we will first answer the sub-questions.

7.1.1 Can a design flow including Clash offer a developer an interface for making a time-area trade-off?

As seen in the case study, the developer can set the parallelism in each layer in their network. The features of Clash, such as the strong type system and the higher-order functions, could be used to make a time-area trade-off. The strong type system can be used to set the number representation throughout the whole system at compile time. The higher-order functions can be used to make variable arrays of processing elements available.

7.1.2 Can a design flow including Clash offer the developer transparency in their design choices?

By using Clash in the resulting implementation, some of the steps in building an ANN become more transparent, such as connecting the layers using function composition. The higher-order functions let the parallelism be set in the implementation with one parameter, which decides the length of the array of processing elements.

Using function composition from Haskell to build the network results in a very readable stacking of layers, which is not even achieved in the python-based platforms.

The higher-order functions used to implement the operations can offer the developer greater inside into the operations happening within the system than for-loops could offer.

7.1.3 How much flexibility does a design flow including Clash offer?

The design flow offers the developer to change the parallelism in each layer and offers the developer to change the bit-width. This offers a large number of options to consider. For the small network from the case study, there were already 48 parallelism configurations. And each parallelism configuration can have any of the fixed-point representations, which have no upper bound. These options result in a small network already having a range of magnitudes in latency and resource utilization as seen in chapter 6.3 Simulation results.

7.2 How can Clash be used in a design flow from a software artificial neural network implementation to a hardware accelerator?

Clash can be used to build abstractions of the building blocks for a compiler, thus it could be used for the implemented design flow. Because of the abstraction, it offers the developer input in the parallelism used within the implementation. This allows the developer to make a time-area trade-off such that a network could be implemented on different FPGA platforms.

8 DISCUSSION

8.1 Resulting design flow

The relation between the set parallelism and the sample throughput is not directly visible but needs to be tested. This lessens the ease of building a project around the *network* function. Because it is not known how often the samples can be fed into the network function, and when a prediction is expected.

Building the network as a function composition means that the layers cannot communicate information to previous layers like would be desirable for implementing backpressure. When using backpressure, layers will hold their output until the next layer is ready to receive data. This absence of communication to previous layers relates to the behaviour of ANNs not depending on future layers, while for control mechanisms this might be useful.

A downside of Clash is that the memory access remains more abstract, while it can cost a lot of energy and execution time. Fine-tuning the memory access's would allow for additional performance improvements. The system for building block RAM files in Clash is not flexible enough to allow the data type to be set after Keras-to-Clash compilation. The weights first need to be translated into fixed-point values and loaded into a block RAM file. Therefore limiting the flexibility within the Clash implementation.

Furthermore, the block RAM files are not easily split up, meaning that instantiating multiple filter processing units will multiply the block RAM, instead of cutting the block RAM into multiple pieces.

8.2 Case study

Although the range of possible parallelism configurations is large, most of the configurations will not make efficient use of the DSP blocks. They could be idle a large part of the time, meaning their utilization will be inefficient. To provide the developer with more information on the consequences of their parallelism choices in terms of latency and throughput, a temporal model is required. This temporal model is necessary to find the optimal set of parallelism parameters for a given design.

The case study does not show a useful project, but a nearly empty project. This nearly empty project assists in showing the resource utilization of the network, but showing a project in which the predictions of the network are used to perform some action, can convey the possible use of the network better.

8.3 Simulation results

The throughput of the system is probably even more important than the simulation output of a single sample. However, the system was not capable of processing multiple samples in sequence without a reset. This is a shortcoming of the current implementation.

8.4 Bit-width compared to accuracy

The bit-width of the network can be changed to as low as 1 bit. However, many of the lower bit widths will not have useful accuracies, as the network cannot be trained to work with such low precision numbers. From the simulation results, we can see that an 8-bit implementation works as well as random guesses while a 6-bit implementation works even worse.

It is not clear why the network cannot reach the same accuracy as the Keras implementation of the network. This could indicate that there is some error somewhere in the implementation.

9 FUTURE WORK

9.1 Process multiple inputs

The resulting implementation should be able to process multiple inputs in sequence. To achieve this goal, the layers should be able to reset their state after the last sample of an input image has been processed.

9.2 Memory improvements

9.2.1 Splitting the memory blocks

For a system that could be used to build machine learning into Clash projects, it would be necessary to allow the developer to decide which memory resource is used by the memory predefined blocks. Thus, they need some interface to an external

9.2.2 Changeable block RAM representation

Currently, the number representation is set when compiling from Keras to Clash. This means that when a different representation should be tested the network needs to be recompiled. It would be beneficial if the developer could choose the representation within the Clash environment. How this could be achieved is not directly obvious, but would make the design flow more user friendly.

9.3 Other intermediate representations

Currently, Keras is used as the basis and on top of Keras, a custom abstraction is created. The broadly supported ONNX (Open Neural Network Exchange) format could be used instead. This would mean that not only models defined in the Keras library can be translated, but models from any library, for which a translation to ONNX format exists, could be used to build a hardware model.

For the design flow, it would mean developing in any Platform supported by ONNX, translating to an ONNX format. And this ONNX format can then be compiled into a Clash implementation.

9.4 Quantized network training

Frameworks for training quantized networks have been made, while this is not possible within Keras (yet). When the training step already includes the knowledge that the weights are quantized to some degree, this will more accurately train for the quantized weights. Often these platforms use a floating-point representation in the background but do inference on the quantized version of this float value.

Quantized networks are supported in ONNX, thus that could be used instead of the custom description as input. ONNX allows for different quantization per layer, which can be very useful. Thus, the design flow should be extended to enable different quantization per layer.

9.4.1 Different quantization per layer

It could be useful to have different bit-width per layer. Thus, an extension of the design flow should also allow for changing the quantization for each layer.

9.5 Other architectures

Next to convolutional neural networks, sparse neural networks are also very well suited to be sped up using an FPGA. They are designed to be applied to very large networks where many of the weights can be zero. In a normal network, the zeros do not account for a large part of the network, thus they are multiplied with their input and summed. This operation is a waste as $w_0 \cdot x_0 + 0 \cdot x_1$ is not dependent on the result of the multiplication with zero. Hardware or software that can skip these operations can shorten the computation time of large networks where many weights are zero.

Recurrent networks are another interesting architecture. They can hold on to information of previous inputs and can thus more effectively work with time series. Their implementation would be quite different as they need extra storage to keep a record of previous states.

9.6 Design Space Exploration framework

An analysis of the emitted network should be done to give the developer indications on what relationship the parallelism in each layer has. If a layer must wait for a previous layer's output, it is not beneficial to add more parallelization to the layer. These relations can be found analytically during the compilation. If we can report these relations to the developer, they can choose a more appropriate parallelization per layer.

Currently, the design flow gives a lot of options for parallelism configurations, while most of these will not use the resources effectively. This analysis would greatly assist the developer.

9.7 Window accessing

Efficiently reading the data from memory can be done with the rolling window reading as can be seen in Figure 34. When going to the next operation only three new values need to be read from memory while the rest is shifted either left or up. This could increase the throughput of the resulting network implementation.

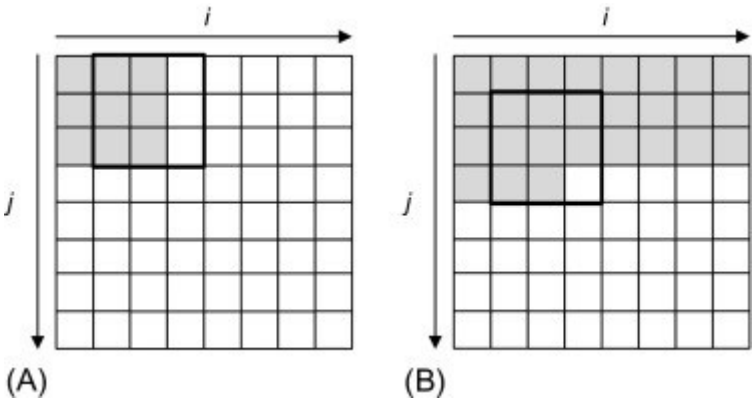

FIGURE 34 EFFICIENT WINDOW ACCESSING FROM [16]

9.8 More efficient convolution

Newer faster methods for computing matrix multiplications have been developed, which could also be used. An example is the method by [27]. But the Winograd method seen before might already be a significant improvement.

9.9 Backpressure

Because the system follows the linear architecture of an ANN, layers cannot tell previous layers whether they are ready for new input. A control system that encompasses all layers could be implemented to keep track of the samples and the state of each layer. This would allow making sure no samples are lost when layers are busy. This control could take up resources to store the samples, but if the layers could tell previous layers that they are busy, this would not be an issue. The samples are simply held in the layers until the next layer is ready to process a new sample.

REFERENCES

[1] K. Fatseas, "Embedded Neural Network Design on the Zybo FPGA for Vision Based Object Locatlization," Enschede, NL, 2018.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz and Y. Jia, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 23 04 2015. [Online]. Available: https://www.tensorflow.org/about/bib.

[3] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, New York, NY, USA, 2014.

[4] The Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," The Theano Development Team, 2016.

[5] Keras, "About Keras," [Online]. Available: https://keras.io/about/. [Accessed 20 May 2021].

[6] D. Dua and C. Graff, "UCI Mahcine Learning Repository: Iris Data Set," University of California, Irvine, School of Information and Computer Sciences, 2017. [Online]. Available: https://archive.ics.uci.edu/ml/citation_policy.html. [Accessed 29 06 2021].

[7] T. E. Oliphant, "Python for Scientific Computing," *Computing in Science Engineering,* pp. 10-20, 2007.

[8] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink and M. Gerards, "CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell," in *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, Lille, France, 2010.

[9] L. Gong, C. Wang, X. Li and X. Zhou, "WiderFrame: An Automatic Customization Framework for Building CNN Accelerators on FPGAs: Work-in-Progress," in *2020 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2020.

[10] W.-F. Lin, D.-Y. Tsai, L. Tang, C.-T. Hsieh, C.-Y. Chou, P.-H. Chang and L. Hsu, "ONNC: A Compilation Framework Connecting ONNX to Proprietary Deep Learning Accelerators," in *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, Hsinchu, Taiwan, 2019.

[11] Y. Ma, N. Suda, Y. Cao, J.-s. Seo and S. Vrudhula, "Scalable and Modularized RTL Compilation of Convolutional Neural Networks onto FPGA," in *IEEE 2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Lausanne, Switzerland, 2016.

[12] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it),," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014.

[13] C. K. D. S. Dimitrios Danopoulos, "Utilizing cloud FPGAs towards the open neural network standard," *Sustainable Computing: Informatics and Systems,* 2021.

[14] Y.-H. Chen, J. Emer and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco, CA, USA, 2016.

[15] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang and H. Yang, "Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* pp. 35-47, 2018.

[16] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang and J. Cong, "Automated Systolic Array Architecture Synthesis forHigh Throughput CNN Inference on FPGAs," in *DAC '17: Proceedings of the 54th Annual Design Automation Conference 2017*, Austin, TX, USA, 2017.

[17] J. M. Cardoso, J. G. F. Coutinho and P. C. Diniz, "Chapter 5 - Source code transformations and optimizations," in *Embedded Computing for High Performance*, Boston, MA, USA, Morgan Kaufmann, 2017, pp. 137-183.

[18] J. Xu, Z. Liu, J. Jiang, Y. Dou and S. Li, "CaFPGA: An automatic generation model for CNN accelerator," *Microprocessors and Microsystems,* pp. 196-206, 2018.

[19] Y. Ma, N. Suda, Y. Cao, S. Vrudhula and J.-s. seo, "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler," *Integration,* pp. 14-23, 2018.

[20] L. Truong and P. Hanrahan, in *A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity*, B. S. Lerner, R. Bodik and S. Krishnamurthi, Eds., Dagstuhl, Germany, Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik, pp. 7:1--7:21.

[21] Intel, "FPGA Design Software - Intel Quartus Prime," Intel, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html. [Accessed 29 06 2021].

[22] onnx, "onnx/onnx: Open standard for machine learning interoperability," 26 April 2021. [Online]. Available: https://github.com/onnx/onnx.

[23] A. Delcher and S. Kasif, "Some results on the complexity of exploiting data dependency in parallel logic programs," *The Journal of Logic Programming,* vol. 6, no. 3, pp. 229-241, 1989.

[24] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss and G. Krueger, *Language Models are Few-Shot Learners,* 2020.

[25] D. C. a. S. Winograd, "On the asymptotic complexity of matrix multiplication," in *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, 1981.

[26] I. Tsmots, O. Skorokhoda and V. Rabyk, "Hardware Implementation of Sigmoid Activation Functions using FPGA," in *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*, 2019.

[27] o. Alman and V. V. Williams, "A Refined Laser Method and Faster Matrix Multiplication," 2020.

[28] S. Saha, "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way | by Sumit Saha | Towards Data Science," Towards Data Science, 15 December 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. [Accessed 15 July 2021].

[29] Y. LeCun, C. Cortes and C. J. Burges, "MNIST handwritten digit database," [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed 08 07 2021].

APPENDIX A

```
module Convolutional.Filter where
import Clash.Prelude
import Data.Maybe (isJust)
import qualified Data.List as L

import Tools.Tools

import Convolutional.DotProduct (dotProduct')


data State = StartUp | Running | Idle | Placement | Finished deriving (Generic, NFDataX, Sh
ow, Eq)

{-# NOINLINE ft #-}

ft s i = s'
  where
    (minIndex, maxIndex, filt, bias, inAddrWind) = i
    (index, prev_index, vecIn, state, addr, wind) = s
    filtLin = concat filt
    windLin = concat wind
    dotPs = zipWith (dotProduct' 0) filtLin windLin
    index'' = if state == StartUp then minIndex else prev_index
    indexWithin = satSub SatZero index'' minIndex
    psum = (foldl (+) bias dotPs)
    vecOut  | state == Running = replace indexWithin (psum) vecIn
            | state == Placement = replace indexWithin (psum) vecIn
            | otherwise = vecIn

    state' | isJust inAddrWind && state == Idle = Running
           | state == Running && index == maxIndex = Placement
           | state == Placement = Finished
           | state == StartUp = Idle
           | state == Finished = Idle
           | otherwise = state
    index' = case state of
      Running -> satSucc SatWrap index
      Placement -> index
      Finished -> minIndex
      Idle -> minIndex
      StartUp -> minIndex
    (addr', wind') | state == Idle = case inAddrWind of
                                       Just (inAddr, inW) -> (inAddr, inW)
                                       Nothing -> (addr, wind)
```

```
                  | otherwise = (addr, wind)
    s' = (index', index, vecOut, state', addr', wind')


fo s = (index, o)
  where
    (index, _, vecIn, state, addr, _) = s
    o = if state == Finished then Just (addr, vecIn) else Nothing

{-# NOINLINE filterUnit #-}

filterUnit splits nFilters filtRom biasRom filterIndex inp = o
  where
    filterIndex' = (snatToIndex nFilters . resize) <$> filterIndex
    unitLength = nFilters `divSNat` splits

    unitLength' = snatToNum unitLength
    startIndex = satMul SatWrap unitLength' <$> filterIndex'
    filterIndexSucc' = satSucc SatWrap <$> filterIndex'
    endIndex = satPred SatWrap <$> satMul SatWrap unitLength' <$> filterIndexSucc'
    startIndex' = if (snatToNum splits) == 1 then (pure 0) else startIndex
    endIndex' = if (snatToNum splits) == 1 then (pure (snatToNum (nFilters `subSNat` d1)))
else endIndex
    ------------------------------------------------------------------------
    wind0 = def -- replicate d3 (replicate d3 (replicate d1 0))
    addr0 = def -- (0,0)
    vec0 = def  -- replicate unitLength 0
    ------------------------------------------------------------------------
    s0 = (0, 0, vec0, StartUp, addr0, wind0)

    filt = filtRom index
    bias = biasRom index

    filterUnitInput = bundle (startIndex', endIndex', filt, bias, inp)
    filterUnitOutput = moore ft fo s0 filterUnitInput

    (index, out) = unbundle filterUnitOutput

    o = out


filtersUnit splits nFilters filtRom biasRom inp = out
  where
    filterUnit' = filterUnit splits nFilters filtRom biasRom
    os = imapA filterUnit' (replicate splits inp)
    out = concatOutput <$> bundle os
    concatOutput ins = o
      where
      concatMaybe vecMaybes = vec'
```

```
      where
      vec' = if all isJust vecMaybes then Just concatCleanVec else Nothing
      cleanVec = getJustOrDef <$> vecMaybes
      concatCleanVec = concat cleanVec
    addr = (fmap fst . last) ins
    vecs = snd <<$>> ins
    vec = concatMaybe vecs
    o = (,) <$> addr <*> vec

```

CODE BLOCK 13 IMPLEMENTATION OF THE FILTER PREDEFINED BLOCK

```haskell
module Convolutional.MemoryManager (memManager) where
import Clash.Prelude
import Data.Maybe (isJust)

import Tools.Tools


ramAddrTranslate :: Num a => a -> (a, a) -> a
ramAddrTranslate maxX (x, y) = (x + y * maxX)

getWriteAddr :: Maybe (a, b) -> a
getWriteAddr (Just (addr, _)) = addr

divideImgCntByStride :: (Integral b, Integral a)
  => (SNat n1, SNat n2) -> (a, b) -> (a, b)
divideImgCntByStride stride@(strX,strY) imgCnt@(x,y) = (x',y')
  where
    x' = x `div` (snatToNum strX)
    y' = y `div` (snatToNum strY)

memManager :: (HiddenClockResetEnable dom,
  Default a1, Default b1, Integral b1, KnownNat n1, KnownNat b2, KnownNat a2,
  NFDataX b1, NFDataX a1)
  => Vec n1 Bool
  -> (SNat a3, SNat b3)
  -> (SNat a2, SNat b2)
  -> (SNat n2, SNat n3)
  -> (Signal dom b1 -> Signal dom (Maybe (b1, b4)) -> Signal dom a1)
  -> Signal dom (Maybe ((b1, b1), b4))
  -> Signal dom (Maybe ((b1, b1), Vec a2 (Vec b2 a1)))
memManager
  valids0
  imgSize@(imgWidth, imgHeight)
  windSize@(windWidth, windHeight)
  stride@(xStride, yStride)
  ram
  i = out
  where
    -----constants-------------------------------
    windWidthI = snatToNum windWidth
    windHeightI = snatToNum windHeight
    imgWidthI = snatToNum imgWidth
    imgHeightI = snatToNum imgHeight
    maxImgX = imgWidthI - windWidthI + 1
    maxImgY = imgHeightI - windHeightI + 1
    -------counters-----------------------------
    wndCounter = count2d windWidthI windHeightI 1 1
    wndCnt' = wndCounter <$> wndCnt --create next window counter value
```

```
    wndCnt = register def $ isValidRead <?> wndCnt' <:> wndCnt --
only update the counter if a valid value was read to put in the window
    isWindDone =
      (((windWidthI - 1, windHeightI - 1) ==) <$> wndCnt) .&&. (isValidRead)
    imgCounter = count2d maxImgX maxImgY (snatToNum xStride) (snatToNum yStride)
    imgCnt'= imgCounter <$> imgCnt
    imgCnt = register def $ (isWindDone .&&. isValidRead) <?> imgCnt' <:> imgCnt
    -------counter to addresses------------------------------
    imgRamAddr = ramAddrTranslate imgWidthI <$> imgCnt
    wndRamAddr = ramAddrTranslate imgWidthI <$> wndCnt
    rdAddr = imgRamAddr + wndRamAddr
    isValidRead = (!!) <$> valids <*> rdAddr -- check if the value is already written
    isValidReadDelayed = toSignal $ delayN d1 def (fromSignal isValidRead)
    isWriting = isJust <$> wr
    wrAddr = getWriteAddr <$> wr
    valids' = replace <$> wrAddr <*> (pure True) <*> valids
    valids = register valids0 $ isWriting <?> valids' <:> valids
    ------memory--------------------
    wr = (\(addr, v) -> ((ramAddrTranslate imgWidthI addr, v))) <<$>> i
    memOut = ram rdAddr wr
    -------bufs---------------
    wind0 = replicate windWidth (replicate windHeight def)
    wind = register wind0 wind'
    wndCntDelayed = toSignal $ delayN d1 def (fromSignal wndCnt)
    wind' = isValidReadDelayed <?>
      (uncurry <$> (replace2d <$> wind) <*> wndCntDelayed <*> memOut) <:>
      wind
    -------------
    isWindDoneDelayed = toSignal $ delayN d2 False (fromSignal isWindDone)
    divdImgCnt = divideImgCntByStride stride <$> imgCnt
    divdImgCntDelayed = toSignal $ delayN d2 def (fromSignal divdImgCnt)
    --
    out = isWindDoneDelayed <?> (Just <$> bundle (divdImgCntDelayed, wind)) <:> (pure Nothi
ng)

```

CODE BLOCK 14 CLASH IMPLEMENTATION OF THE MEMORY PREDEFINED BLOCK

APPENDIX B

```
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# OPTIONS_GHC -Wno-missing-signatures #-}
module MNISTNetwork where
import Clash.Prelude
import Tools.Tools

import qualified Convolutional.MemoryManager as MemoryManager
import qualified Convolutional.Filter as Filter
import qualified Convolutional.Pooler as Pooler
import qualified Convolutional.Activation as Activation
import qualified Convolutional.Flatten as Flatten

import qualified MNISTNetworkWeights as Weights
type Repr = Weights.Repr
type Pntr = Unsigned 8
allValid n = replicate n True
allInvalid n = replicate n False

--------------------------------------------------------------------------------
layer0InputShape :: (SNat 9, SNat 9)
layer0InputShape = (d9, d9)
{-# NOINLINE flatten_0 #-}
flatten_0 :: (HiddenClockResetEnable dom,
    NFDataX (f2 (f3 (Vec 1 (Vec 1 (Vec ((9 * 9) * n) a))))),
    Default (f2 (f3 (Vec 1 (Vec 1 (Vec ((9 * 9) * n) a))))),
    Functor f2, Functor f3)
    => Signal dom (f2 (f3 (Vec 9 (Vec 9 (Vec n a)))))
    -> Signal dom (f2 (f3 (Vec 1 (Vec 1 (Vec ((9 * 9) * n) a)))))
flatten_0 = Flatten.flattenUnit
--------------------------------------------------------------------------------
layer1InputShape :: (SNat 1, SNat 1)
layer1InputShape = ((SNat :: SNat 1), (SNat :: SNat 1))
{-# NOINLINE filters1 #-}
filters1 :: (HiddenClockResetEnable dom,NFDataX a2, Default a2,
  KnownNat m1, KnownNat n1)
  => SNat (n1 + 1)
  -> Signal dom (Maybe (a2, Vec 1 (Vec 1 (Vec 81 Repr))))
  -> Signal dom (Maybe (a2, Vec ((n1 + 1) * m1) Repr))
filters1 width = Filter.filtersUnit width (SNat :: SNat 39) Weights.wss1Rom Weights.bss1Rom
filters1Ram :: (HiddenClockResetEnable dom, Enum addr)
  => Signal dom addr
  -> Signal dom (Maybe (addr, Vec 39 Repr))
  -> Signal dom (Vec 39 Repr)
```

```
filters1Ram = blockRam (def :: Vec (1 * 1) (Vec 39 Repr))

{-# NOINLINE memLayer1 #-}
memLayer1 = MemoryManager.memManager
  (allInvalid d1) (d1, d1) (d1,d1) (d1,d1) filters1Ram

{-# NOINLINE activationLayer1 #-}
activationLayer1 :: (HiddenClockResetEnable dom,
  NFDataX (f2 (f3 (f4 b))), Default (f2 (f3 (f4 b))),
  Functor f2, Functor f3, Functor f4, Ord b, Bounded b, Fractional b, Bits b)
  => Signal dom (f2 (f3 (f4 b)))
  -> Signal dom (f2 (f3 (f4 b)))
activationLayer1 = Activation.activationUnit Activation.relu

{-# NOINLINE dense_1 #-}
dense_1 width = memLayer1 . activationLayer1 . filters1 width
--------------------------------------------------------------------------------
layer2InputShape :: (SNat 1, SNat 1)
layer2InputShape = ((SNat :: SNat 1), (SNat :: SNat 1))
{-# NOINLINE filters2 #-}
filters2 :: (HiddenClockResetEnable dom,NFDataX a2, Default a2,
  KnownNat m1, KnownNat n1)
  => SNat (n1 + 1)
  -> Signal dom (Maybe (a2, Vec 1 (Vec 1 (Vec 39 Repr))))
  -> Signal dom (Maybe (a2, Vec ((n1 + 1) * m1) Repr))
filters2 width = Filter.filtersUnit width (SNat :: SNat 24) Weights.wss2Rom Weights.bss2Rom
filters2Ram :: (HiddenClockResetEnable dom, Enum addr)
  => Signal dom addr
  -> Signal dom (Maybe (addr, Vec 24 Repr))
  -> Signal dom (Vec 24 Repr)
filters2Ram = blockRam (def :: Vec (1 * 1) (Vec 24 Repr))

{-# NOINLINE memLayer2 #-}
memLayer2 = MemoryManager.memManager
  (allInvalid d1) (d1, d1) (d1,d1) (d1,d1) filters2Ram

{-# NOINLINE activationLayer2 #-}
activationLayer2 :: (HiddenClockResetEnable dom,
  NFDataX (f2 (f3 (f4 b))), Default (f2 (f3 (f4 b))),
  Functor f2, Functor f3, Functor f4, Ord b, Bounded b, Fractional b, Bits b)
  => Signal dom (f2 (f3 (f4 b)))
  -> Signal dom (f2 (f3 (f4 b)))
activationLayer2 = Activation.activationUnit Activation.relu

{-# NOINLINE dense_2 #-}
dense_2 width = memLayer2 . activationLayer2 . filters2 width
--------------------------------------------------------------------------------
layer3InputShape :: (SNat 1, SNat 1)
```

```
layer3InputShape = ((SNat :: SNat 1), (SNat :: SNat 1))
{-# NOINLINE filters3 #-}
filters3 :: (HiddenClockResetEnable dom,NFDataX a2, Default a2,
  KnownNat m1, KnownNat n1)
  => SNat (n1 + 1)
  -> Signal dom (Maybe (a2, Vec 1 (Vec 1 (Vec 24 Repr))))
  -> Signal dom (Maybe (a2, Vec ((n1 + 1) * m1) Repr))
filters3 width = Filter.filtersUnit width (SNat :: SNat 10) Weights.wss3Rom Weights.bss3Rom
filters3Ram :: (HiddenClockResetEnable dom, Enum addr)
  => Signal dom addr
  -> Signal dom (Maybe (addr, Vec 10 Repr))
  -> Signal dom (Vec 10 Repr)
filters3Ram = blockRam (def :: Vec (1 * 1) (Vec 10 Repr))

{-# NOINLINE memLayer3 #-}
memLayer3 = MemoryManager.memManager
  (allInvalid d1) (d1, d1) (d1,d1) (d1,d1) filters3Ram

{-# NOINLINE activationLayer3 #-}
activationLayer3 :: (HiddenClockResetEnable dom,
  NFDataX (f2 (f3 (f4 b))), Default (f2 (f3 (f4 b))),
  Functor f2, Functor f3, Functor f4, Ord b, Bounded b, Fractional b, Bits b)
  => Signal dom (f2 (f3 (f4 b)))
  -> Signal dom (f2 (f3 (f4 b)))
activationLayer3 = Activation.activationUnit Activation.sigmoid

{-# NOINLINE dense_3 #-}
dense_3 width = memLayer3 . activationLayer3 . filters3 width
--------------------------------------------------------------------------------
network width1 width2 width3  = (dense_3 width3).(dense_2 width2).(dense_1 width1).flatten_
0
```

CODE BLOCK 15 OUTPUT OF THE KERAS TO CLASH COMPILER FOR THE FULLY CONNECTED NETWORK

```haskell
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# OPTIONS_GHC -Wno-missing-signatures #-}
module MNISTConvNetwork where
import Clash.Prelude
import Tools.Tools

import qualified Convolutional.MemoryManager as MemoryManager
import qualified Convolutional.Filter as Filter
import qualified Convolutional.Pooler as Pooler
import qualified Convolutional.Activation as Activation
import qualified Convolutional.Flatten as Flatten

import qualified MNISTConvNetworkWeights as Weights
type Repr = Weights.Repr
type Pntr = Unsigned 8
allValid n = replicate n True
allInvalid n = replicate n False

--------------------------------------------------------------------------------
layer0InputShape :: (SNat 3, SNat 3)
layer0InputShape = ((SNat :: SNat 3), (SNat :: SNat 3))
{-# NOINLINE filters0 #-}
filters0 :: (HiddenClockResetEnable dom, NFDataX a2, Default a2,
  KnownNat n1, KnownNat m1)
  => SNat (n1 + 1)
  -> Signal dom (Maybe (a2, Vec 3 (Vec 3 (Vec 1 Repr))))
  -> Signal dom (Maybe (a2, Vec ((n1 + 1) * m1) Repr))
filters0 width = Filter.filtersUnit width (SNat :: SNat 16) Weights.wss0Rom Weights.bss0Rom
activationLayer0 :: (HiddenClockResetEnable dom,
  NFDataX (f2 (f3 (f4 b))), Default (f2 (f3 (f4 b))),
  Functor f2, Functor f3, Functor f4, Ord b, Bounded b, Fractional b, Bits b)
  => Signal dom (f2 (f3 (f4 b)))
  -> Signal dom (f2 (f3 (f4 b)))
activationLayer0 = Activation.activationUnit Activation.relu

filters0Ram :: (HiddenClockResetEnable dom, Enum addr)
  => Signal dom addr
  -> Signal dom (Maybe (addr, Vec 16 Repr))
  -> Signal dom (Vec 16 Repr)
filters0Ram = blockRam (def :: Vec (12 * 12) (Vec 16 Repr))

{-# NOINLINE memLayer0 #-}
memLayer0 = MemoryManager.memManager
  (allInvalid d144) ((SNat :: SNat 12), (SNat :: SNat 12)) layer1InputShape (d2,d2) filters
0Ram

{-# NOINLINE conv2d_0 #-}
```

```
conv2d_0 width = memLayer0 . activationLayer0 . filters0 width
--------------------------------------------------------------------------------
layer1InputShape :: ( SNat 2, SNat 2)
layer1InputShape = ((SNat :: SNat 2), (SNat :: SNat 2))
{-# NOINLINE poolerLayer1 #-}
poolerLayer1 = Pooler.maxPooler

pool1Ram = blockRam (def :: Vec (6 * 6) (Vec 16 Repr))
{-# NOINLINE memPoolerLayer1 #-}
memPoolerLayer1 = MemoryManager.memManager
  (allInvalid d36) ((SNat :: SNat 6), (SNat :: SNat 6)) layer2InputShape (d1,d1) pool1Ram

{-# NOINLINE maxpooling2d_1 #-}
maxpooling2d_1 = memPoolerLayer1 . poolerLayer1
--------------------------------------------------------------------------------
layer2InputShape :: (SNat 3, SNat 3)
layer2InputShape = ((SNat :: SNat 3), (SNat :: SNat 3))
{-# NOINLINE filters2 #-}
filters2 :: (HiddenClockResetEnable dom, NFDataX a2, Default a2,
  KnownNat n1, KnownNat m1)
  => SNat (n1 + 1)
  -> Signal dom (Maybe (a2, Vec 3 (Vec 3 (Vec 16 Repr))))
  -> Signal dom (Maybe (a2, Vec ((n1 + 1) * m1) Repr))
filters2 width = Filter.filtersUnit width (SNat :: SNat 16) Weights.wss2Rom Weights.bss2Rom
activationLayer2 :: (HiddenClockResetEnable dom,
  NFDataX (f2 (f3 (f4 b))), Default (f2 (f3 (f4 b))),
  Functor f2, Functor f3, Functor f4, Ord b, Bounded b, Fractional b, Bits b)
  => Signal dom (f2 (f3 (f4 b)))
  -> Signal dom (f2 (f3 (f4 b)))
activationLayer2 = Activation.activationUnit Activation.relu

filters2Ram :: (HiddenClockResetEnable dom, Enum addr)
  => Signal dom addr
  -> Signal dom (Maybe (addr, Vec 16 Repr))
  -> Signal dom (Vec 16 Repr)
filters2Ram = blockRam (def :: Vec (4 * 4) (Vec 16 Repr))

{-# NOINLINE memLayer2 #-}
memLayer2 = MemoryManager.memManager
  (allInvalid d16) ((SNat :: SNat 4), (SNat :: SNat 4)) layer3InputShape (d2,d2) filters2Ra
m

{-# NOINLINE conv2d_2 #-}
conv2d_2 width = memLayer2 . activationLayer2 . filters2 width
--------------------------------------------------------------------------------
layer3InputShape :: ( SNat 2, SNat 2)
layer3InputShape = ((SNat :: SNat 2), (SNat :: SNat 2))
{-# NOINLINE poolerLayer3 #-}
```

```
poolerLayer3 = Pooler.maxPooler

pool3Ram = blockRam (def :: Vec (2 * 2) (Vec 16 Repr))
{-# NOINLINE memPoolerLayer3 #-}
memPoolerLayer3 = MemoryManager.memManager
  (allInvalid d4) ((SNat :: SNat 2), (SNat :: SNat 2)) layer4InputShape (d1,d1) pool3Ram

{-# NOINLINE maxpooling2d_3 #-}
maxpooling2d_3 = memPoolerLayer3 . poolerLayer3
--------------------------------------------------------------------------------
layer4InputShape :: (SNat 2, SNat 2)
layer4InputShape = (d2, d2)
{-# NOINLINE flatten_4 #-}
flatten_4 :: (HiddenClockResetEnable dom,
    NFDataX (f2 (f3 (Vec 1 (Vec 1 (Vec ((2 * 2) * n) a))))),
    Default (f2 (f3 (Vec 1 (Vec 1 (Vec ((2 * 2) * n) a))))),
    Functor f2, Functor f3)
    => Signal dom (f2 (f3 (Vec 2 (Vec 2 (Vec n a)))))
    -> Signal dom (f2 (f3 (Vec 1 (Vec 1 (Vec ((2 * 2) * n) a)))))
flatten_4 = Flatten.flattenUnit
--------------------------------------------------------------------------------
layer5InputShape :: (SNat 1, SNat 1)
layer5InputShape = ((SNat :: SNat 1), (SNat :: SNat 1))
{-# NOINLINE filters5 #-}
filters5 :: (HiddenClockResetEnable dom,NFDataX a2, Default a2,
  KnownNat m1, KnownNat n1)
  => SNat (n1 + 1)
  -> Signal dom (Maybe (a2, Vec 1 (Vec 1 (Vec 64 Repr))))
  -> Signal dom (Maybe (a2, Vec ((n1 + 1) * m1) Repr))
filters5 width = Filter.filtersUnit width (SNat :: SNat 10) Weights.wss5Rom Weights.bss5Rom
filters5Ram :: (HiddenClockResetEnable dom, Enum addr)
  => Signal dom addr
  -> Signal dom (Maybe (addr, Vec 10 Repr))
  -> Signal dom (Vec 10 Repr)
filters5Ram = blockRam (def :: Vec (1 * 1) (Vec 10 Repr))

{-# NOINLINE memLayer5 #-}
memLayer5 = MemoryManager.memManager
  (allInvalid d1) (d1, d1) (d1,d1) (d1,d1) filters5Ram

{-# NOINLINE activationLayer5 #-}
activationLayer5 :: (HiddenClockResetEnable dom,
  NFDataX (f2 (f3 (f4 b))), Default (f2 (f3 (f4 b))),
  Functor f2, Functor f3, Functor f4, Ord b, Bounded b, Fractional b, Bits b)
  => Signal dom (f2 (f3 (f4 b)))
  -> Signal dom (f2 (f3 (f4 b)))
activationLayer5 = Activation.activationUnit Activation.sigmoid

```

```
{-# NOINLINE dense_5 #-}
dense_5 width = memLayer5 . activationLayer5 . filters5 width
--------------------------------------------------------------------------------
network width0 width2 width5  = (dense_5 width5).flatten_4.maxpooling2d_3.(conv2d_2 width2)
.maxpooling2d_1.(conv2d_0 width0)

```

CODE BLOCK 16 OUTPUT OF THE KERAS TO CLASH COMPILER FOR THE CONVOLUTIONAL NETWORK

```
module MNISTNetworkWeights where
import Clash.Prelude

type Repr = Fixed Signed 32 32

--------------------------------------------------------------------------------
wss1Path = "bins/wss1.bin"
wss1Rom :: (Enum addr, HiddenClockResetEnable dom) =>
     Signal dom addr -> Signal dom              (Vec 1 (Vec 1 (Vec 196 Repr)))
wss1Rom rd = unpack <$> blockRamFile (SNat :: SNat 39) wss1Path rd (pure Nothing)

bss1Path = "bins/bss1.bin"
bss1Rom :: (Enum addr, HiddenClockResetEnable dom) =>
     Signal dom addr -> Signal dom Repr
bss1Rom rd = unpack <$> blockRamFile (SNat :: SNat 39) bss1Path rd (pure Nothing)
--------------------------------------------------------------------------------
wss2Path = "bins/wss2.bin"
wss2Rom :: (Enum addr, HiddenClockResetEnable dom) =>
     Signal dom addr -> Signal dom              (Vec 1 (Vec 1 (Vec 39 Repr)))
wss2Rom rd = unpack <$> blockRamFile (SNat :: SNat 24) wss2Path rd (pure Nothing)

bss2Path = "bins/bss2.bin"
bss2Rom :: (Enum addr, HiddenClockResetEnable dom) =>
     Signal dom addr -> Signal dom Repr
bss2Rom rd = unpack <$> blockRamFile (SNat :: SNat 24) bss2Path rd (pure Nothing)
--------------------------------------------------------------------------------
wss3Path = "bins/wss3.bin"
wss3Rom :: (Enum addr, HiddenClockResetEnable dom) =>
     Signal dom addr -> Signal dom              (Vec 1 (Vec 1 (Vec 24 Repr)))
wss3Rom rd = unpack <$> blockRamFile (SNat :: SNat 10) wss3Path rd (pure Nothing)

bss3Path = "bins/bss3.bin"
bss3Rom :: (Enum addr, HiddenClockResetEnable dom) =>
     Signal dom addr -> Signal dom Repr
bss3Rom rd = unpack <$> blockRamFile (SNat :: SNat 10) bss3Path rd (pure Nothing)
```

CODE BLOCK 17 CLASH WEIGHTS FILE, OUTPUT BY THE KERAS-TO-CLASH COMPILER FOR THE DENSE MNIST NETWORK