UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Deepwise Separable Convolution Support in Neural Network Platform

DEGREE FINAL WORK

Degree in Computer Engineering

Author: Brian Miguel McMullen García

Tutors: José Flich Cardo
Carles Hernández Luz

Course 2020-2021

Resum

En els últims anys, la intel.ligència artificial (IA) s'està convertint en un element imprescindible en múltiples àmbits tecnològics. Al mateix temps que la IA s'està desenvolupant a nivell d'algorismes, també les arquitectures de processament s'estan adaptant per a un millor suport de la IA. Per millorar coneixement de les implicacions de les arquitectures amb els algoritmes de IA es fa imprescindible l'ús de noves eines que permetin una exploració adecuada i, per tant, un desenvolupament d'algorismes i arquitectures òptimes i adaptats a les necessitats particulars del problema a resoldre per la IA.

En aquest projecte es desenvoluparà el suport necessari per a permetre l'us de la convolució "DeepWise Separable"en una plataforma d'entrenament e inferència de procesos de IA. La convolució s'implementarà en un sistema basat en la FPGA ALVEO de Xilinx emprant High-level sintesis. Per a demostrar les ventajes que ens proporciona la nostra implementació de la convolució, compararem la nostra implementació amb una implementació de la convolució directa.

Paraules clau: Inteligència artificial, arquitectures, convolucions, infèrencia, entrenament

Resumen

En los últimos años la Inteligencia Artificial (IA) se está convirtiendo en un elemento imprescindible en múltiples ámbitos tecnológicos. Al mismo tiempo que la IA se está desarrollando a nivel de algoritmos, también las arquitecturas de procesamiento se están adaptando para un mejor soporte de la IA. Para un mejor conocimiento de las implicaciones de las arquitecturas con los algoritmos de IA se hace imprescindible el uso de nuevas herramientas que permitan una exploración adecuada y, por consiguiente, un desarrollo de algoritmos y arquitecturas óptimos y adaptados a las necesidades particulares del problema a resolver por la IA.

En este proyecto se desarrollará todo el soporte para habilitar el uso de la convolución DeepWise Separable en una plataforma de entrenamiento e inferencia de procesos de IA. La convolución se implementará en un sistema basado en una FPGA, la ALVEO de Xilinx utilizando High-Level Synthesis. Para demostrar las ventajas que nos proporciona nuestra implementación de la convolución, compararemos nuestra implementación con la implementación de una convolución directa.

Palabras clave: Inteligencia artificial, arquitecturas, convoluciones, inferencia, entrenamiento

Abstract

In the last few years, Artificial intelligence (AI), has become an essential element of many technological fields. While AI has been developing on the level of algorithms, processing architectures have also been developing to better support AI. To obtain a better understanding of the implications of these architectures on AI algorithms, it is indispensable to use the new tools that allow the appropriate exploration, and consequently, the development of optimum algorithms and architectures adapted to the particular needs of the given problem to be solved by AI.

This project will develop all the support necessary to enable the use of the Depthwise Separable (DWS) convolution on a training and inference platform for AI. The convolution will be implemented on a system based on an FPGA, an Alveo by Xilinx, using High-Level Synthesis. To demonstrate the advantages that our implementation of the

convolution provides us, we will compare our implementations with that of a direct convolution.

Key words: Artificial intelligence, architectures, convolution, inference, training

Contents

List of Figures

CHAPTER 1
Introduction

In the present day, neural networks are used for a wide variety of applications. Of particular interest in recent years have been deep convolutional neural networks of increasing depth and complexity, used for image processing in various fields. As these networks have become larger, the memory space that they occupy, as well as the time it takes to execute them has been steadily increasing. Thus it is of great interest to find methods and configurations by which we can decrease these costs and even study the possible trade-offs that can be made, and the relevance of these for different use cases.

The two main points of interest in convolutional neural networks (CNN) in particular are the use of heterogeneous computing platforms to accelerate the computations required by these deep networks, and the adoption of DWS layers for convolutional layers. The combination of these two technologies is the focus of my project, as the studying of the behaviour of DWS configurations will be much richer if one can also see how the memory usage is utilised for different sizes of the DWS layer. We will also be able to see more specifically how the increase in size for DWS layers varies from the increase in size from the standard convolution layers.

1.1 Motivation

With the advancing complexity of CNNs, as well as the scope of their uses, research is being done to optimize the use of machine learning by balancing resource usage with accuracy and responsiveness. As the architectures on which neural networks are used grows, the creation of development platforms is now more necessary than ever. One very useful technology that can be used to test implementations of architectures is the Field Programmable Gate Array (FPGA). (See Figure 1.1)

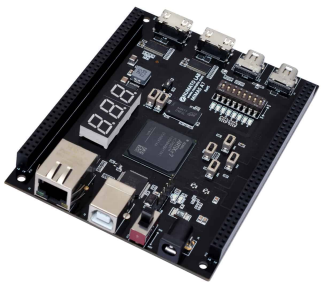

Figure 1.1: Mimas A7 Artix7 FPGA board

Figure 1.2: Example of autonomous driving car Tesla Model S

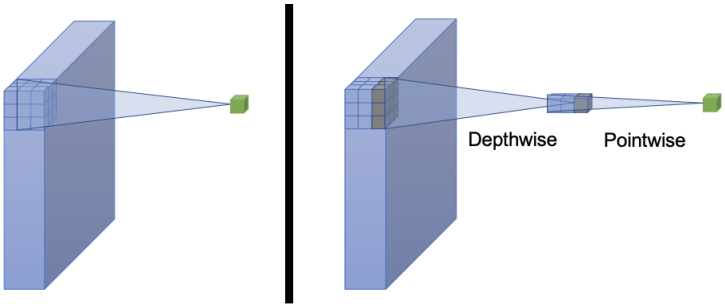

Figure 1.3: Left: Standard convolution, Right: Depthwise Separable configuration (composed of a Depthwise and a Pointwise layer)

Convolutional neural networks have revolutionised the image processing environment, such as in the case in self driving cars like Tesla, which allows cars to autonomously navigate complex environments by interpreting visual information and making decisions at a speed which humans are physically incapable of (see Figure 1.2). Image processing has also enabled the automation of processes that would have been unthinkable just 20 years ago. Some of these projects include sorting produce by quality (using only visual information) or recognising what nutrients plants are lacking. This can be revolutionary in the agricultural sector.

A particular configuration has become of interest in the realm of image processing and CNNs since the inception of the notable MobileNets [4]. This is the Depthwise Separable configuration that can be used in substitution of standard convolutional layers (see Figures 1.3 & 1.4).

The Universitat Politecnica de Valencia is developing a neural network simulation platform for research purposes in the academic field of machine learning, to study new algorithms and implementations for many devices. These devices include CPUs, many different GPUs and FPGAs.

It is in the interest of the academic community to develop the tools to test the increasingly sophisticated models and, for real world applications, FPGAs are a very useful technology to test low energy cost architectures, and depthwise separable layers have been shown to give great benefits to devices of low computational power, such as smartphones. The benefits of implementing the tools to deploy depthwise separable convolutions on FPGAs through neural network development platforms becomes apparent.

Enormous progress has been made over the past decade in AI research with deepening models that can be trained on the cloud for the benefit of all. Now the time has

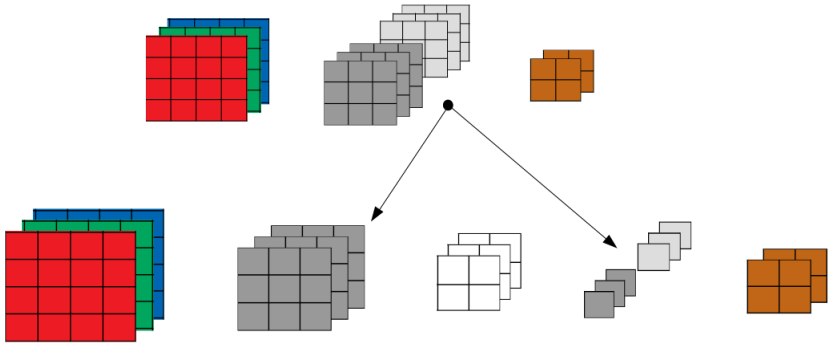

Figure 1.4: Illustration of a standard convolution kernel (top) of dimensions 3x3 with an input size of 4x4 with 3 channels and output size of 2x2 with 2 channels and its equivalent in the DWS configuration (bottom)

come to be smarter with implementations of the local inference process, which must be runnable in an offline environment, in order to make the benefits of machine learning more accessible to all. 86% of the world population owns a smartphone, meaning most of the world has access to computation, but not on the same scale as large corporations and entities.

The utility of researching the use of FPGAs for neural networks will help advance the understanding of neural networks and how they perform under different architectures and hopefully enable the modularisation of neural networks so that they can run inference as efficiently and effectively as possible, without the overhead of operating systems and other unnecessary complexities that are introduced when using standard computer architectures.

The end goal behind this project is to create support for the DWS layer and in particular to study the benefits that can be obtained by using this layer on an FPGA, compared with a standard convolutional layer, especially for use cases in which there are multiple input and output channels. Support for the DWS layer will be created within the HELENNA platform, a research tool created by the UPV, and will be subsequently used to carry out the various tests necessary to perform as extensive of a cost benefit analysis as possible. An FPGA has been selected as a benchmarking device in order to perform the necessary comparisons, this device is the Alveo U200, that is manufactured by Xilinx.

The expectation is that this project will help push the academic understanding and awareness of this new neural network architecture trick for image processing. This could inform and facilitate the design process for programmers of neural networks under various types of constraints, be their energy, response time or computational restrictions.

There may be a beneficial trade-off to be discovered in the mass adoption of DWS layers replacing standard convolutions under certain use cases. This could possibly lead to a new approach to making better inferences with the same amount of memory, if not only scaling back accuracy in order to fit larger neural networks on small devices.

The study of convolutional layers on FPGA's will most likely be the future of neural networks, as they facilitate prototyping circuits, which will help inform computer architecture engineers to make more informed design decisions when creating hardware for neural networks or even find shortcuts to boost efficiency and/or reduce complexity and size.

These new understandings of how hardware can interact with CNNs could also in turn inform machine learning engineers when creating neural networks so that they run

Figure 1.5: Xilinx Alveo U280

even more efficiently according to the state of the art hardware architectures that are being built for AI purposes.

This may also push our knowledge of how well neural networks can integrate into units made specifically for AI problem solving that could one day become completely autonomous, learning exclusively from the environment. If we want machine learning to transcend human capability, it is worth investing time and research into the logistics of making completely independent machines that require minimum computational overhead.

1.2 Objectives

The following project has the objective of developing the support for the DWS configuration on the neural network application HELENNA, for both CPUs and FPGAs. The final goal of this project is to be able to train a DWS layer and then deploy the trained values onto a FPGA to be used for inference. Once this has been accomplished, an analysis of the results given by using this configuration in comparison to a standard convolution will be made in terms of inference time, training time, memory occupation and accuracy/efficiency.

The FPGA implementation will be done with Xilinx HLS, and the training algorithm for CPU will follow the GEMM format to be easily converted to GPU implementations and for parallelisation in general. The specific objectives are as follows:

- Develop FPGA inference implementation for DWS configuration on HELENNA.
- Develop training and inference implementation of DWS configuration for CPU on HELENNA.
- Train a DWS layer and export the resulting model so that inference can be performed with an FPGA.
- Compare costs and benefits of a DWS convolution layer in comparison to a standard convolution layer.

1.3 Structure of report

The rest of this report is structured as follows:

- Chapter 2: This chapter will aim to explain the overall structure, parameters and general behaviour of convolutional layers. It will also explore the differences between a standard convolution and a depthwise-pointwise combination, going into detail for the functions involved and how they differ for each version.
- Chapter 3: Explanation of the HELENNA platform, going into detail about the structure that is used for FPGA kernels and tests. Rough explanation of Vitis HLS and how it is used. Also explaining the structure that is used for training complete networks.
- Chapter 4: Cover the process of implementation of the DWS inference process on the FPGA platform, as well as the CPU/GPU implementation in the format of matrix multiplications for both inference and weight training (forward pass and back propagation).
- Chapter 5: Analysis of results, comparing training times, accuracy loss/gain and inference times, advantages and disadvantages of using a DWS configuration compared to a standard convolution. This chapter will also cover the deployment process onto a FPGA platform and the results found with respect to memory occupation and inference times.
- Chapter 6: This final chapter draws conclusions from everything that has been explored in this project and how this information can be used moving forward.

CHAPTER 2

Structure, parameters and functions of convolution-type layers

One of the most commonly used components of neural networks today is the convolutional layer. It provides a way to process data and encapsulate the information that comes in the form of images, in other words, image processing. One of the ways the data can come in is the simple form of gray-scale style images, in which the information has one single channel, and two dimensions, height and depth. However, images will often come in more than one channel, such as is the case in RGB images, in which there are 3 values per pixel in the image. The ways in which image data can be processed can become more diverse once the channel output size becomes greater than one. An example of how this data can be treated can be shown with the use of DWS convolutions compared to standard convolutions.

2.1 State of the art

Neural networks have been developed since the 1950's with the inspiration of the behaviour of biological brains. The most notable starting point of this timeline was the perceptron, devised by F. Rosenblatt, as the first automated pattern recognition system[8]. The limitations of the perceptron were soon recognised once particular circuits were identified as being impossible with contemporary models. This, as well as the limited computing power available at the time, led to a drastic slowing of research in neural networks. Renewed interest arose in the 1970's and 1980's once runtimes became reasonable with more advanced computing technology and the backpropagation algorithm had been created [7].

Convolutional neural networks are a particular type of neural networks that approximate the behaviour of vision processing in biological organisms. The first iteration of these neural networks to lay the groundwork for future development was the neocognitron (See Figure 2.1) in 1979 by Japanese Computer Scientist Kunihiko Fukushima[3]. This artificial neural network was used for character recognition and other simple tasks. The development of more intricate CNNs was enabled by the use of GPUs in the 2000's. Today, CNNs have become one of the most interesting sectors of neural network research to be tackled. Competitions are held every year with the objective of AI researchers showing off their best architectures for image classification, a task which CNNs excel at.

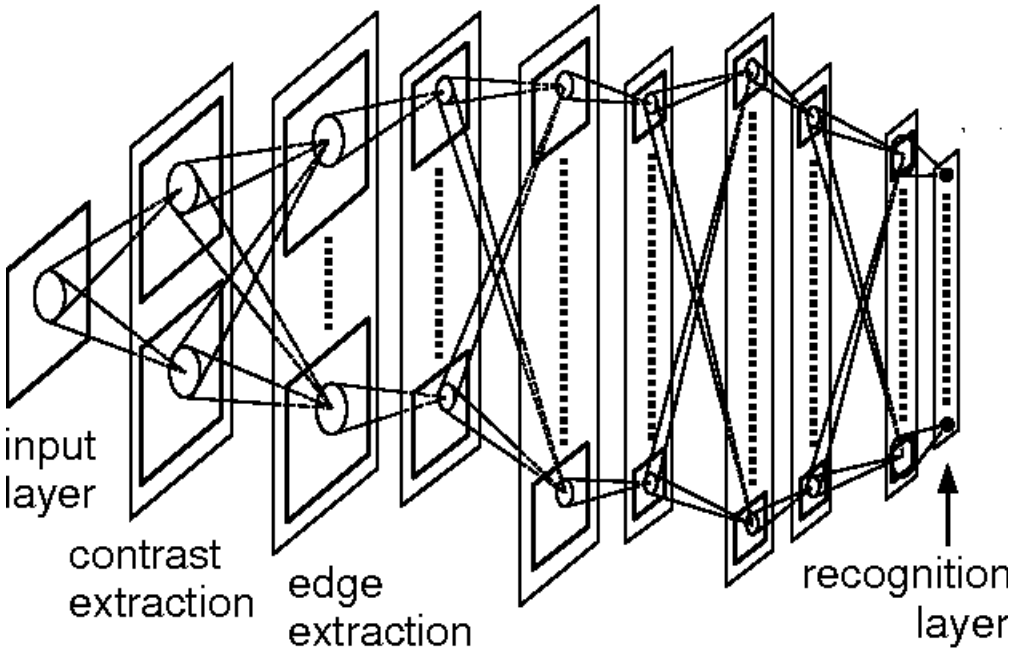

Figure 2.1: Neocognitron proposed by Kunihiko Fukushima from 1979

One of the latest developments in CNNs has been the inception of Depthwise Separable convolutions[9]. These configurations have become more significant with the widespread adoption of smartphones, as these devices have limited power consumption restraints as well as computational power constraints in comparison to desktop computers which can take advantage of greater processing power as well as constant power supply. Starting from the 2010s, DWS layers have been tested and shown to give a drastic decrease in inference times, ie. the time to produce a response, as well as training times compared to full convolutional layers, in exchange for minimal costs in accuracy.

The implications of developing more sophisticated neural networks have been predicted since the very inception of artificial neurons and the realisation that they are possible. It is known, that close to human level accuracy can be achieved in pattern recognition exercises, and in some applications, neural networks can even surpass the effectiveness of human level decision making, such as has been demonstrated with the famous AlphaGO in the realm of the ancient chinese board game known as Go[1].

Many costs can be cut with the use of artificial intelligence and even lives can be saved. We can see this in Google's use of AI to design microchips[5] or in Tesla's autopilot AI[11] which has a lower accident rate than that of human beings. Another point of interest in Tesla's use of AI is that Tesla's autopilot runs on its own architecture and chip, in order to increase inference efficiency drastically. (See Figure 2.2)

In order to advance the effectiveness of neural networks, a lot of research is being done into architecture design specifically for the execution of neural networks, such as has been done with Tesla's autopilot and Google's Cloud TPUs (Tensor Processing Unit).

Research into DWS layers is essential in the realm of CNNs as they provide a new realm to explore the efficiency of neural network architectures on different devices. With their reduced sizes and computational load, they can bring the scale of memory occupation as well as response times down by significant proportions, opening up new possibilities to use neural networks on smaller devices such as smartphones which have become a practically ubiquitous technology across the globe. This can enable much greater ac-

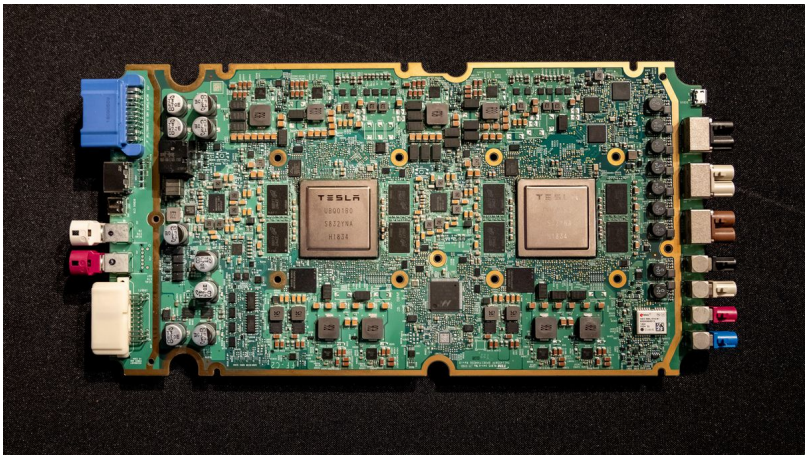

Figure 2.2: Tesla self driving computer

cessibility to the world of artificial intelligence by facilitating portable use cases and also improving the feasibility of using preexisting neural networks for those who do not have access to high end computing technology. This can be done by replacing standard convolutional layers in defined neural networks with DWS layers, lowering the bar for required computing power and memory occupation, and increasing the usability of larger neural network architectures.

2.2 The standard convolution

The standard convolutional layer is the simplest to understand. A typical convolutional layer is composed of 4 configuration variables, the kernel width and heights, K_w, K_h, the number of input channels (I_c or CPI) and the number of output channels (O_c or CPO), both of which depend on the input and the output layers.

The input channels (or channels per input) are the number of layers that an input has. For instance, if a layer were to take RGB images as inputs, the input would have 3 channels. An output channel follows the same concept. If a layer were to output a grayscale image, the number of channels in the output would be just one output channel.

There are 4 more variables that are to be considered, two are defined by the previous layer, input height (I_h) and input width (I_w), and two more, output width (O_w) and output height (O_h) which can be extrapolated from the input size and kernel size.

The value of each output pixel comes from overlaying the kernel with the input layer and multiplying all corresponding positions of both, across every input channel and adding them together. This process is repeated as the kernel is moved across every possible position on the input layer. The total number of positions in which the kernel can be applied corresponds to the total size of a single output channel (O_h and O_w).

The output layer is the set of data values that are produced by the layer and returned as an output or as the input values to a succeeding later. Also, the output layer is the interface from which the error gradients are obtained from the succeeding layer.

There are two more additional variables that are used to define a convolutional layer, the padding and the stride. Both the padding and the stride variables have two dimensions each, one for height and one for width.

What the padding does is extend the size of the images by adding a border of pixels across the top and bottom, and/or the left and right sides of the images as the padding

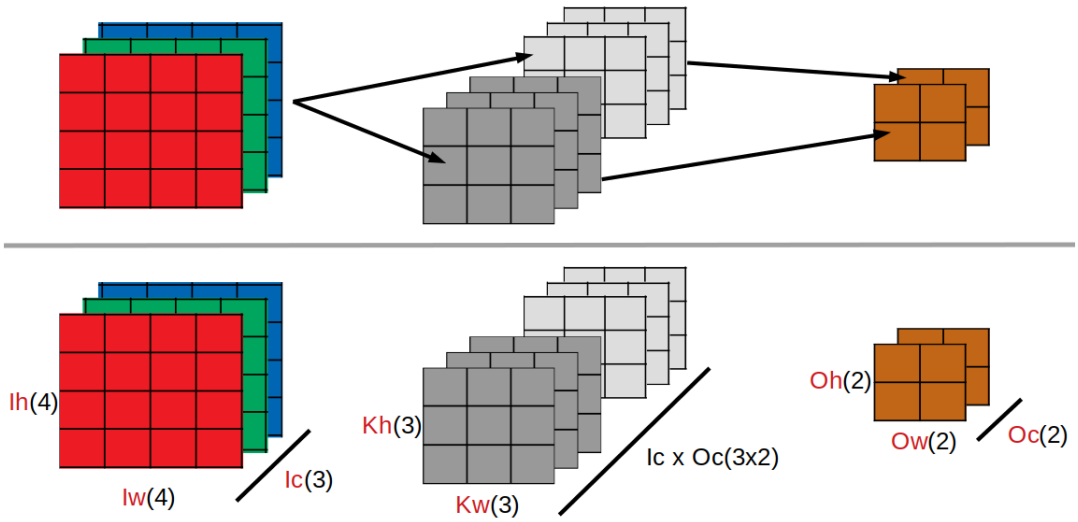

Figure 2.3: Standard convolution with kernel size (3 x 3), input size (4 x 4 x 3) and output size (2 x 2 x 2) From left to right: The input is represented by red, green and blue pixels, each color representing an input channel from an RGB image. The kernel is represented by two shades of grey pixels with each shade representing a kernel set. The output is represented by the brown pixels.

variables are increased respectively. The padding layer that is added to the image is filled with zero values. The effect that increasing the padding has is that it will change the size of the output images by increasing them in height and width.

The stride defines the size of the steps that are made for a convolution to be performed, both for horizontal passing steps, and also vertical steps once a horizontal edge has been reached by the kernel. For a stride of 1, the kernel will move by a single pixel for each iteration that composes a convolution pass. If the stride changes to 2, then the kernel will move by two pixels for each next iteration. The effects of increasing the stride is that it will decrease the size of the output image.

For the sake of maintaining the problem simple enough to navigate, from now on, it is to be assumed that padding is set to a size of 0, and the stride size is of 1.

As a convolutional layer can create an output of any number of channels, the convolutional layer will have a kernel set ($K_w \times K_h \times I_c$) for the number of output channels that has been selected, giving an output of size ($O_w \times O_h \times O_c$) and the corresponding total layer weight size of ($K_h \times K_w \times I_c \times O_c$) (see Figure 2.3).

2.3 Depthwise separable convolution

The DWS convolution is a different type of convolutional layer that has been used since 2014 [9]. It is composed of two parts, the deepwise (DW) convolution, and the pointwise convolution (PW) in succession (see Figure 2.4). In other words, the output of the deepwise convolution is the immediate input to the pointwise convolution. This configuration has been shown to be significantly smaller in size, and faster in execution than the standard convolutional layer. Once again the configuration variables are kernel width (K_w) and height (K_h), the input (I_c) and output channels (O_c).

The deepwise section is composed of a set of kernels, the set size being equivalent to the number of input channels (I_c), giving the deepwise convolution a weight size of ($K_w \times K_h \times I_c$). The output to this convolution is computed by overlaying each kernel in the set

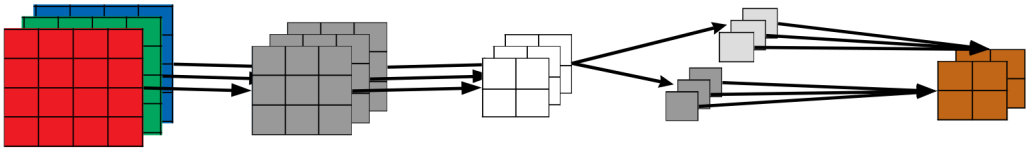

Figure 2.4: DWS convolution with deepwise kernel size (3 x 3), input size (4 x 4 x 3), intermediate size (2 x 2 x 3) represented by the white pixels, pointwise kernel size (1x1 x 3 x 2) and output size (2 x 2 x 2) From left to right: The input is represented by red, green and blue pixels, each color representing an input channel from an RGB image. The deepwise kernel is represented by grey pixels. The intermediate output is represented by white pixels. The pointwise kernel is represented by two shades of grey pixels with each shade representing a kernel set. The output is represented by the brown pixels.

with its corresponding input channel, and multiplying each corresponding position of the kernel and input and adding them together. The kernel is then moved across every input position and repeated. However, in this case, the sums for each channel are kept separate, such that an output of size $(O_h \times O_w \times I_c)$ is created, rather than $(O_h \times O_w \times 1)$ which would have been the output from a standard convolution. This convolution does not require an output channel size, or alternately, it can be said that the output channel size is that of the input channel size.

The pointwise section is much smaller and simpler, as it is in essence a standard convolution, with a kernel width and height of 1. The only configuration variables that concern it are the input and output channel sizes (I_c) and (O_c). In its entirety, it is composed of an O_c number of kernel sets $(1 \times 1 \times I_c)$, giving it a weight size of $(I_c \times O_c)$. The output of this layer is computed by multiplying and adding each set of $1 \times 1 \times I_c$ kernels across all input channels to obtain each output channel. The result is an output with the same width and height of its input, but with selected output channel size.

The total layer size of the DWS configuration ends up being $(K_h \times K_w \times I_c) + (I_c \times O_c)$ or conversely, $(K_h \times K_w + O_c) \times I_c$ compared to the standard convolution $(K_h \times K_w \times O_c \times I_c)$. As we can see here, the actual size of the number of weights to be stored for this layer scale much better for the DWS configuration, in particular when using larger output channel sizes.

2.4 The functions of convolution-type layers

The processes that are necessary to implement to be able to train and use a neural network are the inference and the training functions. In other words, the inference function is that which, when given layer weights and an input, should produce an output value for the next layer. The training process should be able to receive an error gradient from the succeeding layer, update the weight values accordingly and compute and pass on the corresponding error gradients to the previous layer.

The data that is given to update the values is the error with respect to the output dE/dO. The data that is already available is the input and the weights. Using the chain rule (see Figure 2.5), we can obtain the gradient we require. Firstly by multiplying the error with respect to the output by the weights, to obtain the error of the preceding layer. Secondly, by multiplying the same error with respect to the output by the input, to conversely obtain the weight gradient required to update the weights.

The simplest version of this process is for a single multiplication neuron(see Figure 2.6). A convolutional layer is essentially just a series of multiplication neurons.

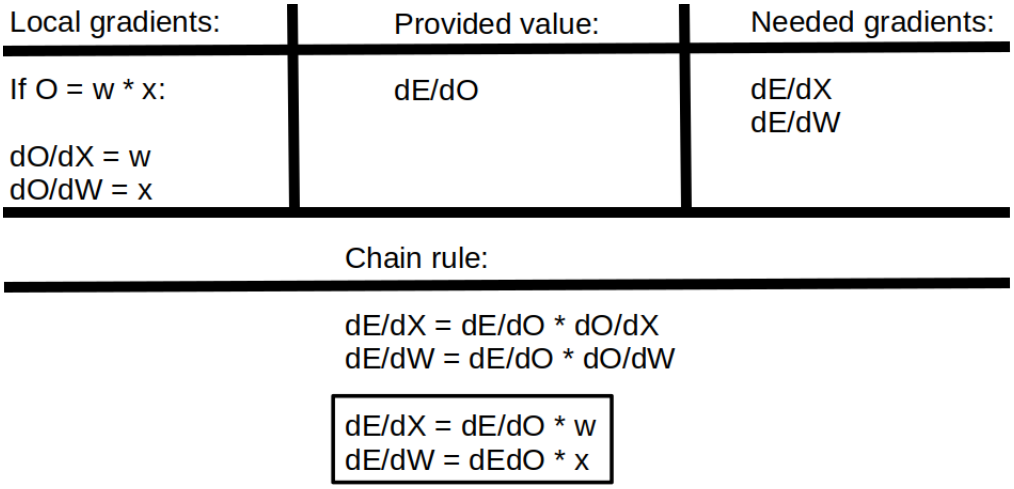

Figure 2.5: Derivation of the chain rule

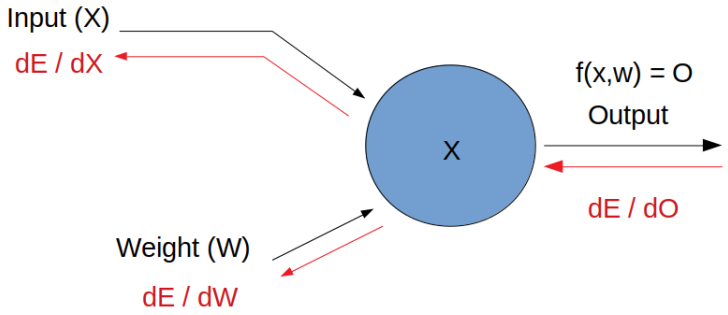

Figure 2.6: The functions of a multiplication neuron in a neural network

The function that is used for the inference process is known as the Forward pass, the function that is used to update the weight values of the layer is known as the Backward pass and the function for passing on the error gradient to the preceding layer is known as Error propagation. Next we will describe how these processes are carried out and how they interact with each other, both under a standard convolution context, and a DWS context.

2.4.1. Forward pass

The forward pass simply provides the output values for the layer, be they final classification guesses, or intermediate values to be taken as an input into another layer in the CNN.

For the standard convolution, the forward pass consists of the multiplication and addition of a kernel set as it slides across the input frame (see Figure 2.7), the kernel having a set of weights $(K_h \times K_w)$ for each input channel. The number of multiplications that are made per forward pass is computed as follows. There will be $O_h \times O_w$ iterations (positions in which the kernel fits on the input frame), each iteration will have one multiplication per weight in a kernel set $K_h \times K_w \times I_c$, and this will all be repeated for each output channel, O_c. This gives a total multiplication count of: $Kh \times K_w \times I_c \times O_h \times O_w \times O_c$. We can simplify this if we assume $K_h = K_w = K_d$ and $O_h = O_w = O_d$, giving a multiplication count of ${K_d}^2 \times {O_d}^2 \times I_c \times O_c$.

For the DWS convolution, we must break the forward pass process into its two composing parts, the deepwise and the pointwise. For the deepwise convolution, the process

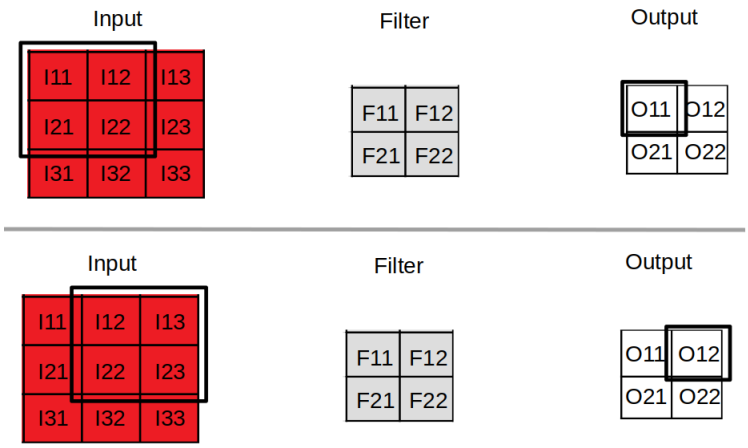

Figure 2.7: Demonstration of forward pass calculation of two out pixels (O_{11} and O_{12}) for a single input and output channel.

is the same as the standard convolution, in which the kernel set slides across the input frame, multiplying each kernel weight with the corresponding input value and summing them together to produce one of the output pixels $O_h \times O_w$, however, the sums are kept separate across each input channel. The result of the number of multiplications for this component is mostly the same as the standard convolution, but it is not repeated O_c times. There are $O_h \times O_w$ iterations, each iteration results in $K_h \times K_w \times I_c$ multiplications each, resulting in $K_h \times K_w \times I_c \times O_h \times O_w$ multiplications in total, simplified: ${K_d}^2 \times {O_d}^2 \times I_c$. For the pointwise convolution, we must remember that this part is in essence a standard convolution with kernel size 1 x 1. This means that we have a multiplication count of $1 \times 1 \times I_c \times O_h \times O_w \times O_c = I_c \times O_h \times O_w \times O_c$, simplified, ${O_d}^2 \times I_c \times O_c$. The sum of the multiplications from both of these sections gives us the total number of multiplications required for a forward pass in a DWS configuration being ${K_d}^2 \times {O_d}^2 \times I_c + {O_d}^2 \times I_c \times O_c = ({K_d}^2 + O_c) \times I_c \times {O_d}^2$.

This means that the execution time of the DWS configuration is:

$$(\frac{1}{O_c} + \frac{1}{{K_d}^2}) \times Std\_conv\_time$$

2.4.2. Error propagation

The error propagation function is used to compute the error values for preceeding layers. It will take the error gradients in from the "output" of the layer, and then calculate the gradient errors for each "input" pixel and pass them on to the preceeding layer to allow it to do the same. The error gradients are used to calculate by how much the weights of the network should be adjusted. However, the update process occurs in the backward pass function.

The error value to be propagated by a multiplication neuron to its preceding layer is obtained by the multiplication of the error with respect to its output, with its current weight.

$$\frac{dE}{dX} = \frac{dE}{dO} \times w$$

For a convolution, the process is the same, however, as a single kernel pixel can be treated as a multiplication neuron, it must be taken into consideration that the neuron will be applied multiple times in a single convolution pass. This multiplication must be summed

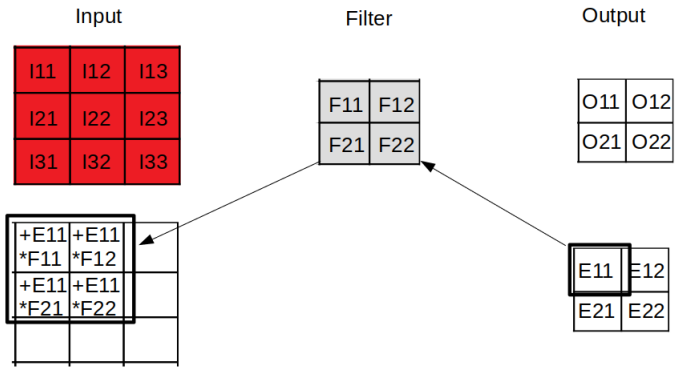

Figure 2.8: Demonstration of error propagation in a convolution (step 1)

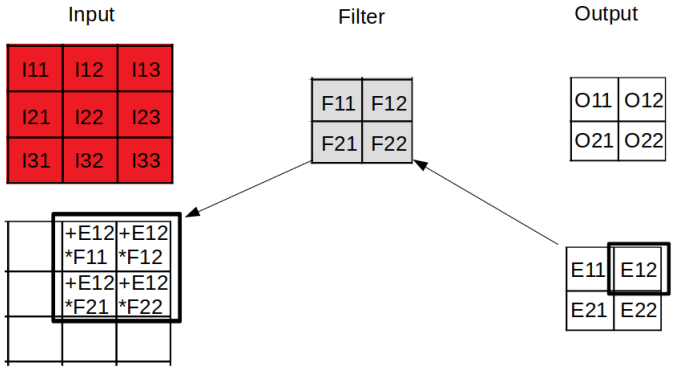

Figure 2.9: Demonstration of error propagation in a convolution (step 2)

up multiple times on the pixel to which the error is being propagated (see Figures 2.8 & 2.9).

In the case of the standard convolution, the process of error propagation is as follows. A series of values is received from the succeeding layer, for every output pixel of our layer, an error gradient is given. For each output pixel, the kernel frame should be multiplied by this pixel, and each resulting value from the kernel frame should be mapped to its corresponding input pixels positions. When values correspond multiple times to the same input pixel, the value should simply be summed to the previously mapped value. To clarify, in the case of there being multiple output channels, the multiplication of an output pixel from a given output channel with a kernel will correspond to the kernel set that is used to compute that output channel in the forward pass. Once all the multiplications have been made and summed up for each input pixel, the error propagation is ready to continue in the preceding layer of the neural network.

For the DWS convolution, the error propagation will occur in two steps, first from the output layer through the pointwise layer to produce an intermediate error gradient, and then this intermediate error gradient is used with the deepwise layer to produce the error gradient for the preceding layer to the DWS configuration. As previously stated, the process for the pointwise component is the same as a standard convolution of size 1 x 1. The input size will be the same as the output size, and due to the fact that the kernel size is 1 x 1, there will be no overlapping mapping values for each output channel separately, as the input and output pixels map 1 to 1. The only summing that will be necessary will come from the different output channels. Moving on to the deepwise section, there will be the opposite effect, as the only repeated mapping to an input pixel will come from the kernel moving across a given input pixel multiple times, but each output channel will

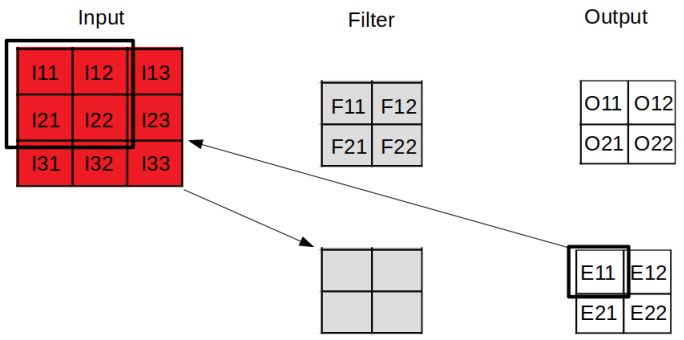

Figure 2.10: Demonstration of backward pass in a convolution (step 1)

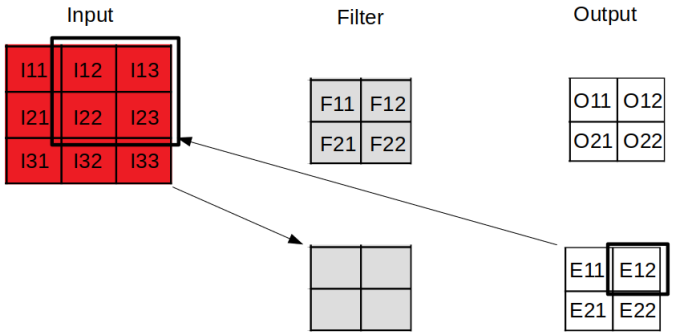

Figure 2.11: Demonstration of backward pass in a convolution (step 2)

only correspond with one input channel, as is the nature of the deepwise convolution in the forward pass process.

2.4.3. Backward pass

The backward pass function is used in the context of the training process by updating the weight values of the layer. Given the error gradient of the succeeding layer and the stored input values from the forward pass, the weight update adjustments can be computed.

The update value to be assigned to a weight in a multiplication neuron is calculated by the multiplication of the output error by the input value. The equation below shows a derivation of the chain rule (see Figure 2.5) from which the previously mentioned statement draws its behaviour.

$$\frac{dE}{dW} = \frac{dE}{dO} \times x$$

For a kernel pixel, this process is the same, however, as each kernel pixel will pass multiple inputs per convolution pass, this multiplication must be made for every position the kernel will take, with the corresponding input value, and summed to obtain the final value to update by (see Figures 2.10 & 2.11). Once the update gradient is obtained, the value of the kernel pixel weight is updated by the function F shown below:

$$F = F - learning\_rate * gradient$$

The backward pass for the standard convolution receives the error gradient from the succeeding layer after having performed and stored a forward pass input value, for each

mapping from output pixel to input pixel, each corresponding kernel weight must sum its corresponding multiplication of the output pixel by the alligned input value from the prior forward pass.

In the case of the DWS convolution, the order in which the weights are updated, deepwise or pointwise first, does not matter, so long as the error has been propagated through both. One could perform the backward pass immediately after having received its error gradient, in which case the pointwise layer must be dealt with first, or one could wait for the entire network to have received its error gradient, in which case the weights can be updated in any order. For the pointwise section, once again we are reminded that the input and output sizes are equivalent in height and width, therefore, the weight update values correspond to the sum of the multiplication of each output pixel by the input pixel in the same X,Y coordinate, each output channel pixel corresponding to the associated kernel set. Moving on to the deepwise section, we map the output pixels to their input pixels, and sum the multiplications to the corresponding kernel values, again we take note that there is no cross contamination across channels. Each output channel corresponds exactly to its input channel, and therfore, to its specific kernel channel.

CHAPTER 3
The HELENNA platform

HELENNA is a software platform for the development of training and inference processes for neural networks developed by the GAP research group from the UPV. HELENNA is the acronym for HEterogeneous LEarning Neural Networks Application. As such, it has the objective of using heterogeneous computation architectures.

Although multiple solutions exist for the training and inference of neural networks, HELENNA has been developed to explore new training methods and new neural networks in the context of the research projects carried out by the GAP research group. The use of this platform allows for greater control over training and inference processes, and consecuently, a more appropriate environment for the study and analysis of alternative processes that improve performance.

HELENNA is written in C and C++ (mainly in C) and has a code structure similar to that of object programming, although it does not use objects. Specifically, HELENNA allows the definition of layers in neural networks by isolating the code of each layer within a single file and implementing different basic functions to support the training and inference of the layer.

Another aspect of HELENNA is the management of memory and the abstraction of the location of the memory buffers used in the processes of training or inference. In fact, the tensors (implemented with buffers) are created dynamically with HELENNA and they are located in the selected target device (CPU, GPU, FPGA). The user is not aware in the moment of instantiation and use of the buffers, of where they are located. HELENNA internally performs the necessary transfers for the correct use of the buffers. Each buffer created in HELENNA is identified with a unique identifier and each operation on the tensor/buffer exclusively uses the identifier.

HELENNA allows for training on CPU and on GPU with different optimizations, mainly with the use of matrix operation libraries. Specifically, HELENNA allows for the use of the following devices and optimized configurations.

- CPU. With this device the CPU and all of its cores are used for training and/or inference. No matrix operations libraries are used. This device is the most basic one.

- MKL. With this device, the CPU is used alongside the MKL matrix operations library by Intel. This library works exclusively for Intel processors. Maximum performance is obtained for the CPU with this device.

- Cublas. With this device, NVidia GPUs are used. Specifically, the CuBLAS library by NVidia is used for matrix operations.

Layer	Parameters	Description
Fully_connected	Number of parameters	Dense layer, also known as MultiLayer Perceptron (MLP)
Convolution	Kernel size (KH, KW), stride size (SH, SW), padding size (PH, PW)	Convolutional layer
Batch Normalization	Gamma and Beta	Normalisation layer
ReLU	None	Activation function that returns the largest value as long as it is larger than 0
Softmax	None	Activation function that returns a probability distribution. All outputs are between 0 and 1.
Dropout	None	Component used to cut off unneeded neurons

Figure 3.1: Summary of all of the layers that have been implemented in HELENNA

- CuDNN. Recently, NVIDIA has offered CuDNN as a library for the training and inference of neural networks. This library works on NVIDIA GPUs. With this device, said library is used, significantly improving performance for machine learning-like workloads.
- OpenCL-FPGA. With this device, FPGA based systems are used for the inference process. Specifically, this device type allows for the use of modules generated specifically for FPGAs such as convolutions and activation functions.

Currently, HELENNA supports a variety of neural network layers. What follows are a list of supported layers, as well as some details on each one of them (see Figure 3.1):

HELENNA supports different database formats as well as databases that are often used in the context of neural networks. These are:

- MNIST: The Modified National Institute of Standards and Technology (MNIST) database is a collection of 28x28 pixel grayscale images of hand written numbers from 0 to 9. It is composed of 70,000 images, 10,000 of which are testing images, and the rest are training images.
- CIFAR-10: The Canadian Institute For Advanced Research (CIFAR) - 10 database is a collection of 60,000 32x32 RGB images, of 10 different classes (airplanes, cars, birds, cats, deer, dogs frogs, horses, ships and trucks). Each class has 6,000 samples in the database.
- CIFAR-100: The Canadian Institute of Standards and Technology (CIFAR) - 100 database is another collection of 60,000 32x32 RGB images, however, there are 100 total classes with 600 images per class.
- ImageNet: The ImageNet database is a crowd sourced database of 20,000 different classes of images in RGB. So far the database is composed of at least 14 million

annotated sample images. The ImageNet project runs an annual image recognition competition known as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) on its database.

Lastly, HELENNA enables distributed training on CPUs as well as on GPUs. Different distributed training strategies are supported by HELENNA.

3.1 Tools

The HELENNA platform uses a variety of tools in order to automate compilation, maintain version control as well as coordinate between contributors to the project. As previously mentioned, most of the code in HELENNA is written in C or C++, which requires the use of a compiler, and much of the compilation process is automated and easily made cross platform thanks to the use of the CMake tool.

With regards to version control, a git repository is available for contributors to submit changes and add new functionality. New branches are created for new contributors and these are used to try modifying parts of the code without breaking the overall master version. Communication among contributors is necessary when maintaining version control, which is why Microsoft Teams was used for weekly meetings to discuss progress and also ask professors any questions that may have arisen throughout the previous week. Slack was also used to communicate among other contributing members who are helping to build more support for HELENNA. Slack allows for a more forum style way of communicating between members, as different threads/groups can be made in order to easily find relevant discussions for the different components of HELENNA that are under construction.

Vitis analyzer is a tool for Xilinx FPGAs that allows designers to fine-tune their designs. In particular it is used to better probe the results of uploading a model to an FPGA, by tracking the application execution patterns to follow when user functions are activated, tracking the waveforms in order to follow and check that signals are being triggered in a timely and expected manner. The profile summary tab in Vitis Analyzer provides statistics of memory read and write speeds. The run guidance section will warn the user of possible detected inefficiencies, where potential improvements can be picked up from.

3.2 Training

Creating the support for a layer consists of the creation of a new file and the instantiation of the functions on the basis of the type of layer to be supported. Each supported layer in HELENNA must implement five basic functions:

- parse: This functions interprets the definition parameters of the layer and initialises the structure.
- allocate: With this function, the layer creates the appropriate buffers (tensors) to support the layer.
- initialize: In this function, the buffers (tensors) are initialized in the layer.
- backpropagation: With this function, the layer gradients are calculated for their subsequent application to the weights.

- propagate_error: With this function, the error for the input is calculated from the error at the output of the function.

The last three of the previously enumerated functions serve to perform the training process to the layer once inserted into a neural network. All of these functions are implemented in a single file, so that code can be reused when possible on implementations of different layers.

At the time of launching a training process, HELENNA allows for the selection of the learning rate which will be used to update weights according to the calculated error gradients as well as allow the selection of the batch size which will decrease the training time of a NN by compounding the accumulated errors from multiple inputs before updating weights[6]. HELENNA enables the option of storing or loading trained NN models with their weights on the platform. These models can later be used to transfer fully trained neural networks to other users or to be uploaded to FPGAs for the inference process.

3.3 FPGA support in HELENNA

HELENNA uses Xilinx's High Level Synthesis HLS tool in order to write code that will be transferable to an FPGA bitstream to be uploaded to an FPGA, from C++ code, which allows for the creation of circuits from well defined algorithms. This manner of writing the code for FPGAs also makes it easier to create unit tests for the different layers for which support is being created.

HELENNA only provides FPGA support for the inference process of supported layers. In order to obtain trained weights to upload with a neural network layer, the layer must first be trained on the platform by CPU or GPU and be stored as a model. Once the weights have been stored, they can be transferred to the model that will be uploaded to the FPGA in the bitstream.

To create FPGA support for a new layer, HELENNA requires both the source code for the FPGA using HLS, and corresponding test code in order to perform correctness tests. These tests will check the outputted value from an FPGA, both real and emulated, against the output that is expected, that is calculated directly on the CPU without use of additional libraries. This is in order to ensure that the FPGA behaves as is expected by comparing the outputs from its execution to its execution in a much simpler environment.

The test codes consist of two main functions, one which computes the expected outputs from a given series of inputs, and another which will check the results from both executions. The test code generates input values, either assigned deterministically, in order to facilitate debugging, or randomly, to test stability. The test will take as an input in the command line some given parameters in order to correctly construct the behaviour of the layer and also generate the .xclbin file. The test will then run the generated input series through both the CPU simulation function and through the FPGA emulator, and the results check function compares every corresponding output value from both. If there is a discrepancy between the two computed output values, then the position within the output array is given, and the actual two values will be printed alongside it, showing the difference between the two.

The testing code also allows for the analysis of the entire execution timeline and using Vitis analyzer tool allows for the study of the diagram of the produced circuit on the FPGA device. Figure 3.2 shows the FPGA setup used in this project. As show in the plot, the host (an x86 machine), is connected to an FPGA where a specific kernel function is implemented. The communication between the FPGA and host is performed through

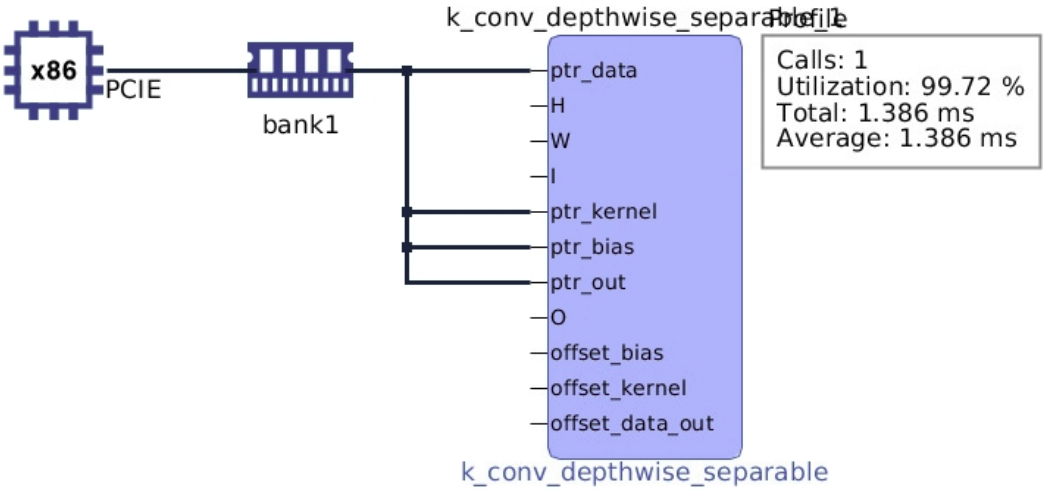

Figure 3.2: System diagram in Vitis analyzer

memory using a PCI express link. The data is moved from the host to the FPGAs memory by the host itself. The computations of the FPGA can be performed by accessing the data directly from the DDR memory of the FPGA card.

3.3.1. HLS, dataflow model and pragmas

When designing circuits for FPGAs, there are multiple options for programming them at different levels. The two low level languages that exist for FPGAs are VHDL and Verilog, the high level option is to use C and HLS. HELENNA uses HLS as it is much easier to implement highly complex algorithms with. A user can create C code that executes the behaviour desired and can run many types of software tests to ensure that the expected values are produced. During this stage, no device is required and compilation is simple and fast. Once the code is sufficiently functional, it can be easily and automatically be converted into implementable code for VHDL devices.

The dataflow model is a series of modules and streams that describe a circuit and are easy to follow visually due to the simplicity of their representation. A module in this context is a complete and contained circuit or function, the behaviour of this can be simply defined and therefore its relative position to other modules can be understood. A stream is in essence a First-In-First-Out queue that is used to connect modules together in order to create a larger circuit. This allows a pipelining process by which allows all modules can be running simultaneously. At any given time in the middle of an execution, all modules will be processing different pieces of data at different stages at the same moment. The analogy comes to mind of using a dryer and a washing machine at the same time for large quantities of clothes, one set of clothes can be in the dryer while the next collection of clothes is in the washing machine. The size of the pipeline can be designed by defining the objects that are to be passed through it. This requires defining the structure so that the pipeline can appropriately pass the correct chunks or units at a time.

This format can be implemented with ease using the HLS pragmas that are available thanks to Xilinx. Using HLS pragmas is very simple to use, as we can develop a dataflow structure in an algorithm and then simply add a few lines of code with the appropriate pragmas to convert the system into a dataflow model that the FPGA model can understand.

The HLS pragmas that will be used shall be the following[10]:

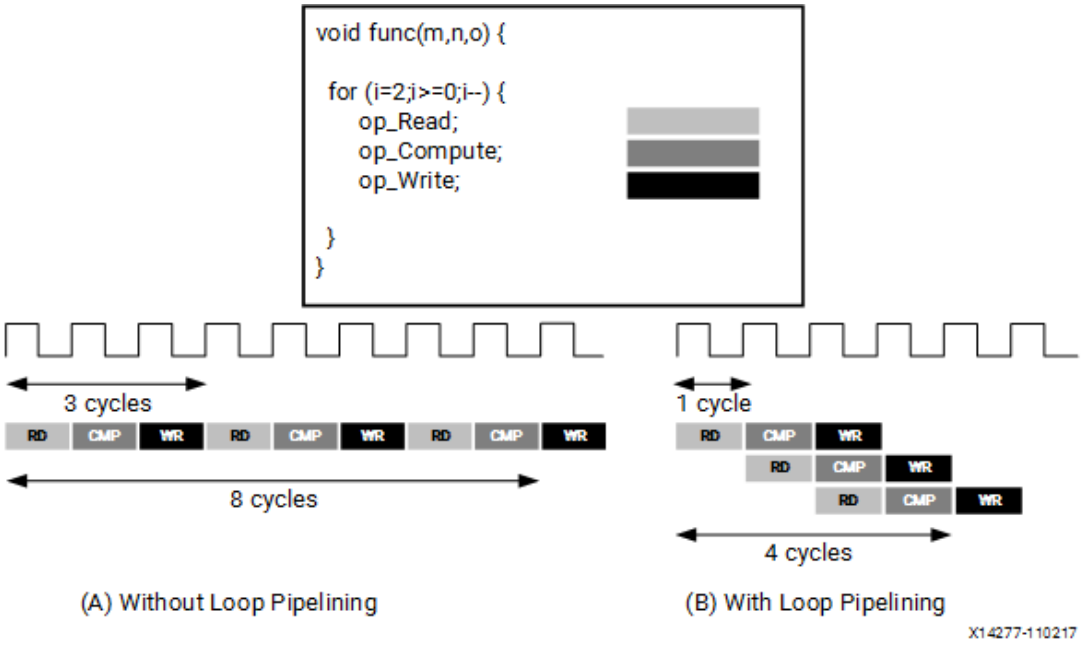

Figure 3.3: Illustration of (A) non-pipelined versus (B) pipelined loops

- HLS PIPELINE: The pipeline pragma allows for concurrent operations to be performed within a specified loop. (See Figure 3.3)
- HLS UNROLL: Pragma unroll creates multiple independent operations rather than a single set of operations which allows for some of the loop iterations to be run in parallel. This is done by creating multiple copies of the loop body in the RTL design.
- HLS LOOP_FLATTEN off: "LOOP_FLATTEN off" indicates that the RTL implementation should NOT flatten the indicated loop.
- HLS DATAFLOW: This pragma enables task level pipelining, which allows for the overlapping of several functions or loops, increasing concurrency and throughput.
- HLS INTERFACE: Pragma interface is used to indicate a variable as a port within the RTL. For the FPGA implementation of the DWS convolution, four HLS interfaces will be declared, and they are all marked as a AXI4 interface. These will be the data pointer, the kernel pointer, the bias pointer and the output pointer.
- HLS DATA_PACK: This pragma is only used twice in the FPGA implementation of the DWS convolution, once for the data/input pointer and one for the output pointer. These two variables will now store their data in a single wide vector in order to reduce the memory consumption of the pixel structure used by these variables.
- HLS ARRAY_PARTITION: The array_partition pragma is used for splitting arrays into smaller elements so that they can fit better in memory, rather than occupying a large localised block of memory. The number of arrays to be created from the main variable can be selected with the "dim=x" clause, but the array can be split entirely into its elements with the use of the "complete" option after the pragma array_partition.

CHAPTER 4
Implementation of the Depthwise Separable layer

When implementing a depthwise separable convolution, two components must be taken into account, the depthwise section and the pointwise section. As has been stated previously, the machine learning platform for which the DWS convolution will be created is HELENNA, for which we will create both an FPGA inference implementation as well as a CPU implementation of the inference and training process.

The training process will be created for CPU in order to have a functional starting point to ensure the correct behaviour of the DWS layer and to create trained models. The FPGA implementation will only implement the inference process of the DWS layer, as the training process will take up unnecessary space, when model training can be done separately by creating models with the HELENNA platform. These trained models will be easily uploaded to the FPGA once trained, as the investigation of the inference process on FPGAs is the final objective of this project.

As the most computation intensive part of creating support for the DWS is the training process, an implementation that will be useful for developing CPU support will be the GEMM (General Matrix Multiply)[2] approach. GEMM is an algebraic functionality that is available in the majority of math libraries.

The order in which these processes will be explained will first be the implementations required for the DWS training support, followed by the inference design process on FPGAs, as this will be the order in which the neural networks will be implemented for the inference process on the final FPGA.

For the FPGA section, both the depthwise and pointwise layers will be created in unison as there is no need to separate them. Creating them as distinct layers in this context would only increase overhead memory occupation and complexity.

However for the CPU section, the depthwise section will be created independently and the pointwise section will reuse the implementation of a standard convolution. In this section, the depthwise convolution must have both the inference and training functions defined in order to create models that will be used for future inference tasks.

4.1 DWS support in CPU

The base code that will be created for the required functions will be implementations for CPUs, and as this is the case, OpenMP will be used to increase the parallelism of

these processes wherever possible. OpenMP is a library implementing multithreaded processing for shared-memory multiprocessing programming in C, C++ and Fortran.

The most important pragmas that will be used are the "omp parallel for" and "private". This line signals to the CPU that the following "for" loop will be parallelizable, meaning that multiple cores can perform different iterations of the loop simultaneously. The "private" term within the same line signals that each individual thread will be holding its own privately modifiable version of the listed variables following the "private" mark in the omp pragma. This particular clause is only necessary for variables that are declared prior to entering the for loop. If the variables are declared inside an omp for-loop, then they are automatically assigned the "private" status for every thread.

The computations that are performed on convolutional layers of CNNs can most often be represented in the form of various matrix multiplications. This makes sense, as it is very common for image processing software and videogame engines to follow matrix multiplication style calculations. CNNs in effect are also image processing algorithms.

In order to create effective support for the DWS layer in CPU we must develop the kernel using GEMM. We will do so with the help of GEMM as it is a very effective way of implementing convolutions, whether for inference or training. This involves first rearranging the values in the input structure via the "im2col" function into their corresponding columns such that the new input structure to be used has as many columns as there are output pixels. The next step is to perform the matrix operations that correspond for the different functions required to train a neural network.

In order to explain the matrix operations that are to be performed, the following abbreviations shall be used:

K = kernel, **I** = input, **O** = output, **E** = error from current layer, E_{-1} = error from preceding layer.

The process of transposing the matrices will be handled by changing the order in which values are accessed, defined within the functions already so that the values can be passed to the functions in their non-transposed original formats.

- Forward pass: $K \times I = O$
- Error propagation: $K^T \times E = E_{-1}$
- Backward pass: $E \times I^T = K_{new}$

4.1.1. im2col

This is the function that enables the inference and training processes of convolutional layers to be performed in the format of matrix multiplications, and consequently, the GEMM functionality. It is used equally in both standard convolutions and in DWS convolutions.

The purpose of this function is to take an input image, and convert it into as many columns as there are output pixels $O_w \times O_h$. Each column will have a length equal to that of the size of the given kernel set $K_w \times K_h \times C_I$. The Figure 4.1) shows an example of a given input (left), the selected kernel size (middle) and the output (right) that would be produced by the im2col function.

This is done by mapping the input values into their corresponding positions in the column format such that each value will align with its corresponding kernel weight position at the time of performing the matrix multiplication.

The code below shows the implementation of the im2col function using C and OpenMP. The parameters of the function are the desired output size and the pointers to memory

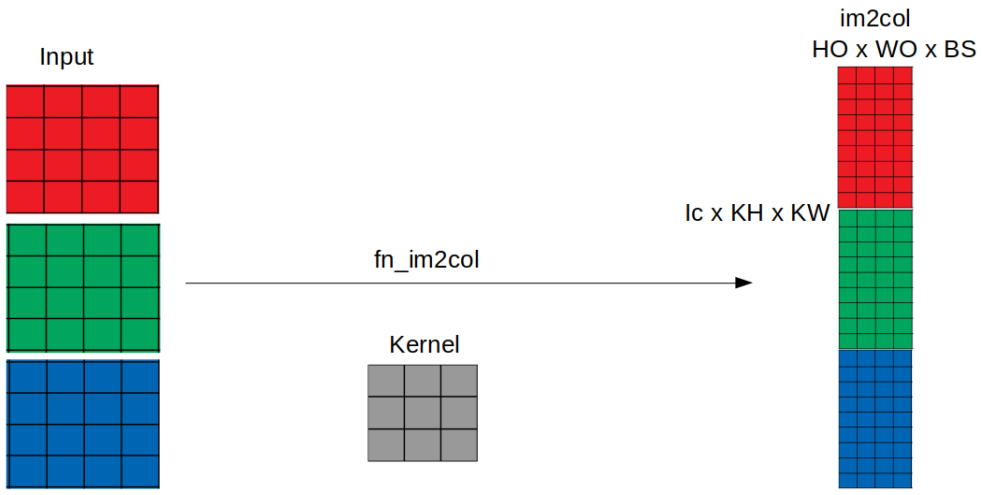

Figure 4.1: Illustration of im2col function on an input of size 4×4 with 3 channels using a kernel size of 3×3. The result is a structure of size $I_c \times K_h \times K_w$ by $O_h \times O_w \times BS$

to be used, as well as the sizes of the following: input, stride and padding. The output of the function is the converted structure of the input.

```
#pragma omp parallel for private (kh, kw, h, w, output_column, output_row)
for (i = 0; i < I; i++) {
  output_row = KW * KH * i;
  output_column = 0;
  size_t out_addr = num_cols * (size_t)KW * (size_t)KH * (size_t)i;
  out_addr -= first_row * num_cols;
  size_t in_addr1 = (size_t)i * (size_t)B * (size_t)WI * (size_t)HI;
  for (kh = 0; kh < KH; kh++) {
    for (kw = 0; kw < KW; kw++) {
      // row filter
      if ((output_row >= first_row) && (output_row <= last_row)) {
        for (h = 0; h < HO; h++) {
          int hi = h * SH - PH + kh;
          size_t in_addr2 = in_addr1 + ((size_t)hi * (size_t)B * (size_t)WI);
          for (w = 0; w < WO; w++) {
            // column filter
            if ((output_column >= first_col) && (output_column <= last_col)) {
              int wi = w * SW - PW + kw;
              int force_padding =
                (wi < 0) || (wi >= WI) || (hi < 0) || (hi >= HI);
              if (force_padding) {
                bzero(&out_ptr[out_addr], B*sizeof(type));
              } else {
                int in_addr = in_addr2 + (wi * B);
                memcpy(&out_ptr[out_addr], &in_ptr[in_addr], B*sizeof(type));
              }
              out_addr+=B;
            }
            output_column = (output_column + B) % (B * WO * HO);
          }
        }
      } else {
        out_addr+= B * WO * HO;
      }
      output_row++;
    }
```

```
  }
}
```

4.1.2. Forward matmul

The forward matmul (**mat**rix **mul**tiplication) function is necessary for DWS support, as it is the function that produces the outputs for the inference process. This function will also be required for the training process, as the outputs are logically needed in order to obtain error rates. The outputs are also necessary to calculate the gradients in order to update the weights of the layer.

This function will take as inputs the depthwise kernel **ptr_a** (rows: I_c, columns: $K_h \times K_w$) and the input in im2col format **ptr_b** (rows: $I_c \times K_h \times K_w$, columns: $O_h \times O_w \times Bs$).

The first operation is the depthwise matrix multiplication which is done by performing the matrix multiplications by their channel parts, such that for each corresponding input channel of the kernel and im2col, the output is calculated for that same channel output.

There will be I_c iterations of matmuls of size $(1 \times {K_d}^2) \times ({K_d}^2 \times O_h \times O_w \times Bs)$, each iteration providing the values for each output row (see Figure 4.2), giving the intermediate value **ptr_c** (rows: I_c, columns: $O_h \times O_w \times Bs$). The code below demonstrates an implementation of the modified matrix multiplication function for the depthwise component using C and OpenMP.

```
#pragma omp parallel for
for (int i = 0; i < rows_a; i++) {
  for (int j = 0; j < cols_b; j++) {
    type v = 0;
    for (int k = 0; k < cols_a; k++) {
        v += ptr_a[(cols_a * i) + k] *
             ptr_b[(cols_b * cols_a * i) + (cols_b * k) + j];
    }
    ptr_c[(i * cols_b) + j] = v;
  }
}
```

Once the intermediate value has been obtained from the depthwise section, a single matmul operation is performed by matrix multiplying the pointwise kernel **ptr_a** (rows: O_c, columns: I_c) by the intermediate value from the previous step **ptr_b** (rows: I_c, columns: $O_h \times O_w \times Bs$). (See Figure 4.3) This will provide the final output of a DWS forward pass **ptr_c** (rows: O_c, columns: $O_h \times O_w \times Bs$).

The code below shows the implementation of the standard convolution matrix multiplication using C and OpenMP, which serves for the pointwise component of the configuration.

```
#pragma omp parallel for
for (int i = 0; i < rows_a; i++) {
  for (int j = 0; j < cols_b; j++) {
    type v = 0;
    for (int k = 0; k < cols_a; k++) {
      v += ptr_a[(cols_a * i) + k] * ptr_b[(cols_b * k) + j];
    }
    ptr_c[(i * cols_b) + j] = v;
  }
}
```

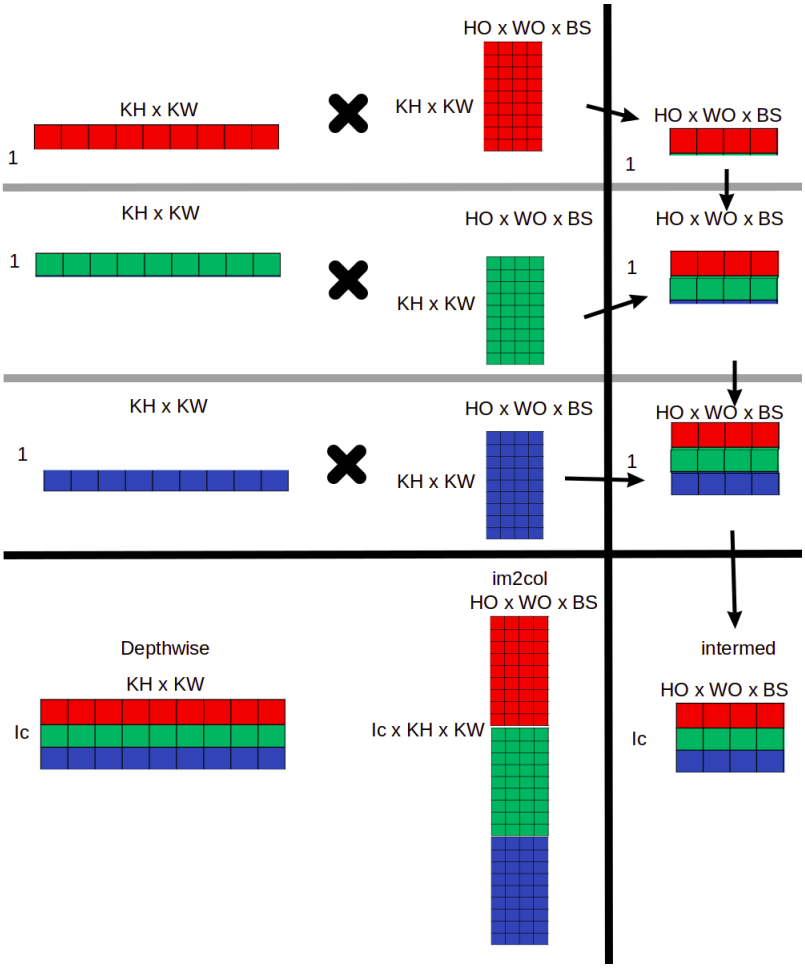

Figure 4.2: Demonstration of the 3 matrix multiplications for the forward pass of a depthwise layer, due to an I_c size of 3 (step 1 for the DWS configuration)

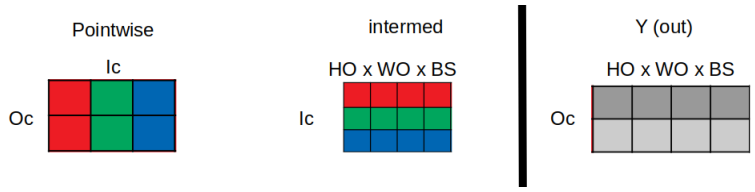

Figure 4.3: Forward demo step 2, a normal $(2 \times 3) \times (3 \times 4)$ matrix multiplication

4.1.3. Error propagation matmul

The error propagation matmul function is required in order to compute the error gradients that will be passed on to the preceding layer within the whole NN. Without the error propagation function, the depthwise component cannot be integrated with the pointwise component, and the DWS configuration as a whole could not be integrated into whole neural networks. The function multiplies the weights of the layer by the error values provided, and then sums them up on the inputs that correspond with the combination of each error value and kernel weight. These inputs then become the errors coming from the outputs of the adjacent layer.

The first step for the the error propagation function will be a simple matmul operation of the transposed pointwise kernel **ptr_a** (rows: I_c, columns: O_c) by the error gradient given from the succeeding/following/next layer **ptr_b** (rows: O_c, columns: $O_h \times O_w \times Bs$). (See Figure 4.4) The output to this step will be the intermediate error value **ptr_c**

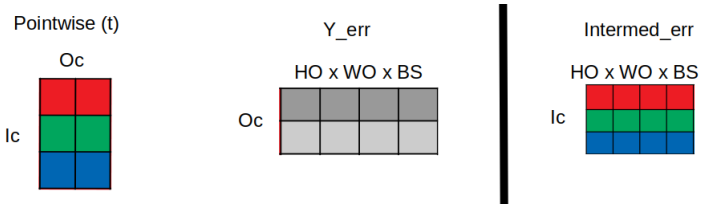

Figure 4.4: Error propagation demo step 1, normal $(3 \times 2) \times (2 \times 4)$ matrix multiplication

(rows: I_c, columns: $O_h \times O_w \times Bs$) that will be used to generate the error value for the depthwise step.

The code below illustrates the implementation of the standard matrix multiplication of $A_{transposed} \times B$, used for the error propagation of the pointwise component.

```
#pragma omp parallel for
for (int i = 0; i < cols_a; i++) {
    for (int j = 0; j < cols_b; j++) {
        type v = 0;
        for (int k = 0; k < rows_a; k++) {
            v += ptr_a[(cols_a * k) + i] * ptr_b[offset_b + (cols_b * k) + j];
        }
        ptr_c[(i * cols_b) + j] = v;
    }
}
```

The second step to pass the error gradient to the layer beyond the DWS configuration will require the transposed depthwise kernel **ptr_a** (rows: $K_h \times K_w$, columns: I_c) and the intermediate error gradients produced from the previous step **ptr_b** (rows: I_c, columns: $O_h \times O_w \times Bs$).

There will be a matmul operation performed for each channel (I_c), going by each column of the transposed DW kernel ($K_h \cdot K_w \times 1$) and the corresponding row of the intermediate errors ($1 \times O_h \cdot O_w \cdot Bs$). This will generate blocks of size ($K_h \cdot K_w \times O_h \cdot O_w \cdot Bs$). Each of these blocks will be appended beneath (See Figure 4.5), creating a final structure with the same dimensions as the im2col used in the forward pass **ptr_c** (rows: $I_c \times K_h \times K_w$, columns: $O_h \times O_w \times Bs$).

The code below shows the implementation of the depthwise derivative of the matrix multiplication of $A_{transposed} \times B$, used for error propagation.

```
#pragma omp parallel for
for(int i = 0; i < rows_a; i++) {
    for(int j = 0; j < cols_b; j++) {
        for(int k = 0; k < cols_a; k++) {
            ptr_c[(cols_b * cols_a * i) + (cols_b * k) + j] =
                ptr_a[(cols_a * i) + k] * ptr_b[(cols_b * i) + j];
        }
    }
}
```

4.1.4. Backward pass matmul

The backward pass matmul is used to implement the process of updating the weights of the layer, from the obtained error values of the succeeding layer and the input values

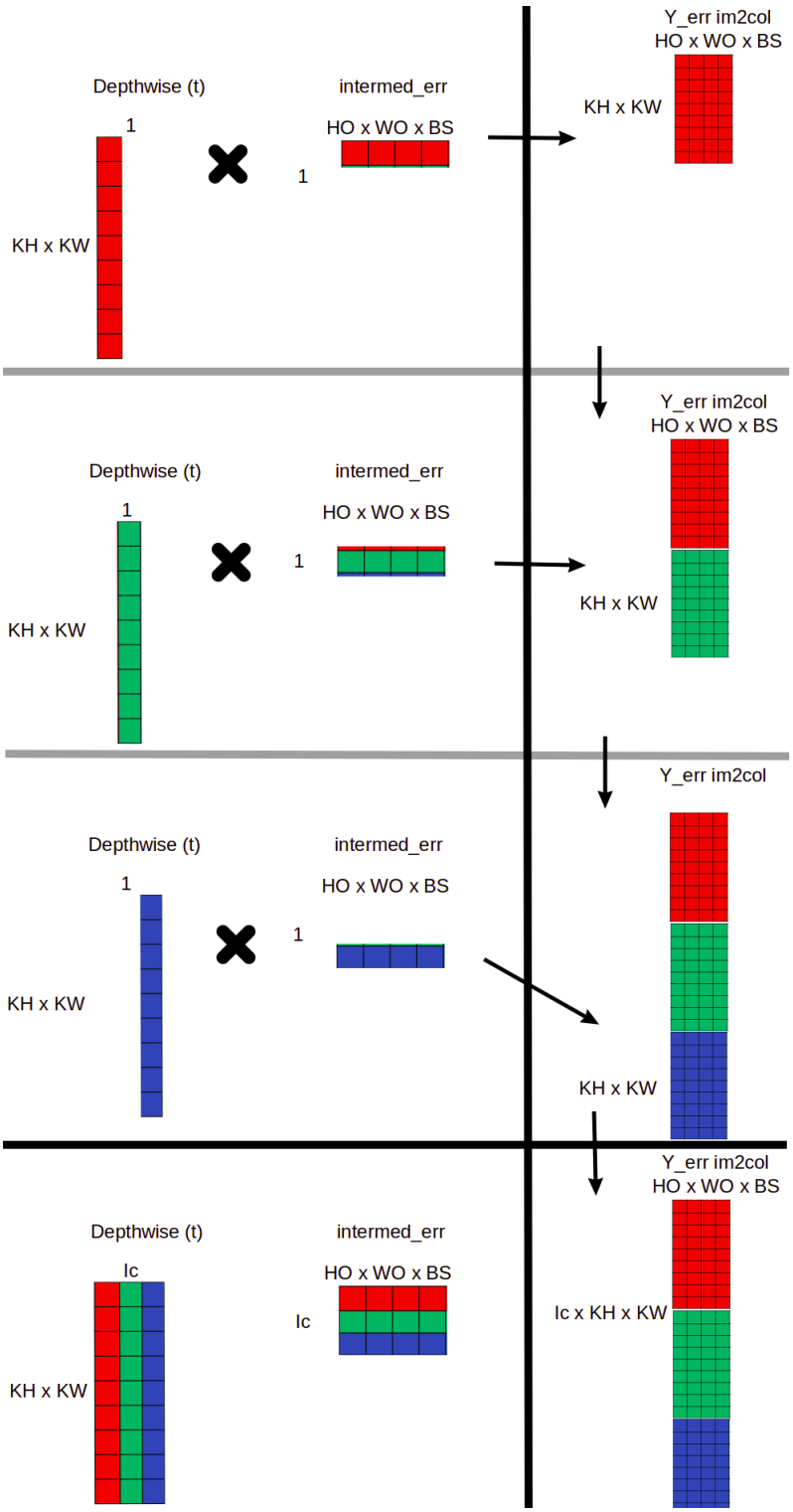

Figure 4.5: Illustration of the 3 matrix multiplications for the error propagation of a depthwise layer, due to an I_c size of 3 (step 2 of the DWS configuration)

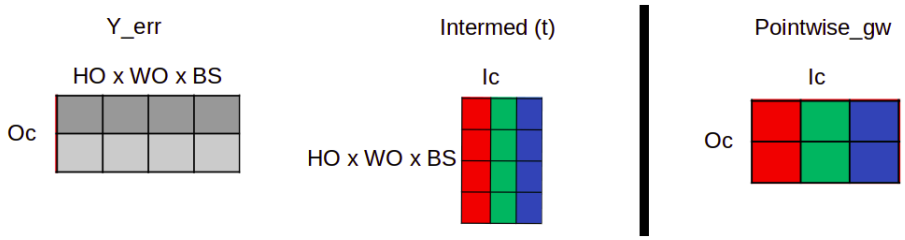

Figure 4.6: Backward demo step 1, normal $(2 \times 4) \times (4 \times 3)$ matrix multiplication

provided by the forward pass matmul. Put simply, this function multiplies the input values that correspond with the output values, and sums them up on the layer weights that connect them.

This function will require the error gradients to have been calculated beforehand via the execution of the error propagation. This is particularly true when considering the intermediate errors that are obtained after the first step of the error propagation for the DWS configuration. Either each step of the backward pass can be performed after the respective error propagation step, or both steps can be run in immediate succession after the full error propagation has been complete throughout the layers of the neural network. The values from the forward pass will also need to have been stored prior to running the backward pass.

The first step to the backward pass is a single matmul of the error gradient from the succeeding layer **ptr_a** (rows: O_c, columns: $O_h \times O_w \times Bs$) by the transposed intermediate output produced by the depthwise component **ptr_b** (rows: $O_h \times O_w \times Bs$, columns: I_c). This will produce the correction gradients for the pointwise component **ptr_c** (rows: O_c, columns: I_c) (See Figure 4.6).

The following code shows the implementation of the depthwise derivative of the matrix multiplication of $A \times B_{transposed}$, used for the depthwise backward pass process.

```
#pragma omp parallel for
for(int i = 0; i < rows_a; i++) {
    for(int j = 0; j < cols_c; j++) {
        for(int k = 0; k < cols_b; k++) {
            ptr_c[i * cols_c + j] +=
            ptr_a[i * cols_b + k] * ptr_b[(i * cols_b * cols_c)
            + (j * cols_b) + k];
        }
    }
}
```

The second step requires the error gradient that is passed from the pointwise component to the depthwise component **ptr_a** (rows: I_c, columns: $O_h \times O_w \times Bs$) and the transposed input to the depthwise layer **ptr_b** (rows: $O_h \times O_w \times Bs$, columns: $I_c \times K_h \times K_w$).

Once again, there will be I_c number of matmuls performed, each producing a single row of the output $(1) \times (K_h \cdot K_w)$. Each matmul operation will be of $(1) \times (O_h \cdot O_w \cdot Bs)$ by $(O_h \cdot O_w \cdot Bs) \times (K_h \cdot K_w)$. (See Figure 4.7) The result will be the correction gradients for the depthwise kernel **ptr_c** (rows: I_c, columns: $K_h \times K_w$).

The code below shows the implementation of the standard matrix multiplication of $A_{transposed} \times B$, used for the pointwise backward pass function.

```
#pragma omp parallel for
for(int i = 0; i < rows_a; i++) {
```

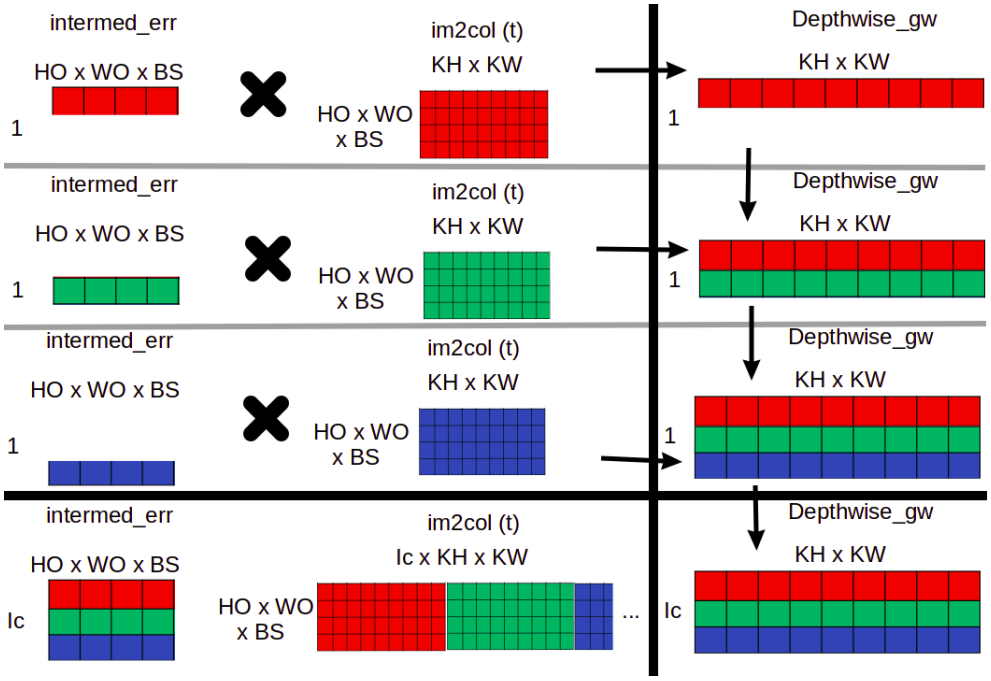

Figure 4.7: Illustration of the 3 matrix multiplications for the backward pass of a depthwise layer, due to an I_c size of 3 (step 2 of the DWS configuration backward pass)

```
    for(int j = 0; j < cols_c; j++) {
        for(int k = 0; k < cols_b; k++) {
            ptr_c[i * cols_c + j] +=
                ptr_a[i * cols_b + k] *
                ptr_b[(i * cols_b * cols_c) + (j * cols_b) + k];
        }
    }
}
```

4.2 Implementing inference support for DWS on FPGA

In order to create an efficient implementation of a DWS convolution with HLS, both the DW and the PW components will be implemented together and the only signals that will be outputs will be the output pixels of the entire configuration. This design decision is crucial to maximize the performance of the FPGA implementation of the inference process. To achieve this, the output of the DW component will be stored as an internal variable to be passed onto the pointwise part of the forwarding process. The output of the PW component will be treated as the output signal of the entire DWS layer. However, merging both implementatios requires being able to meet timing requirements of the FPGA implementation. Otherwise the maximum operating frequency of the entire accelerator would be affected. In the following sections we describe the general architecture of the implemented kernel, and how we have performed several optimizations to achieve the maximum throughput with the DWS convolution.

4.2.1. General schematic of the DWS convolution

The general structure of the DWS layer will consist of three input streams: the kernel weights, the bias values and the input data. There are three processing sections, the

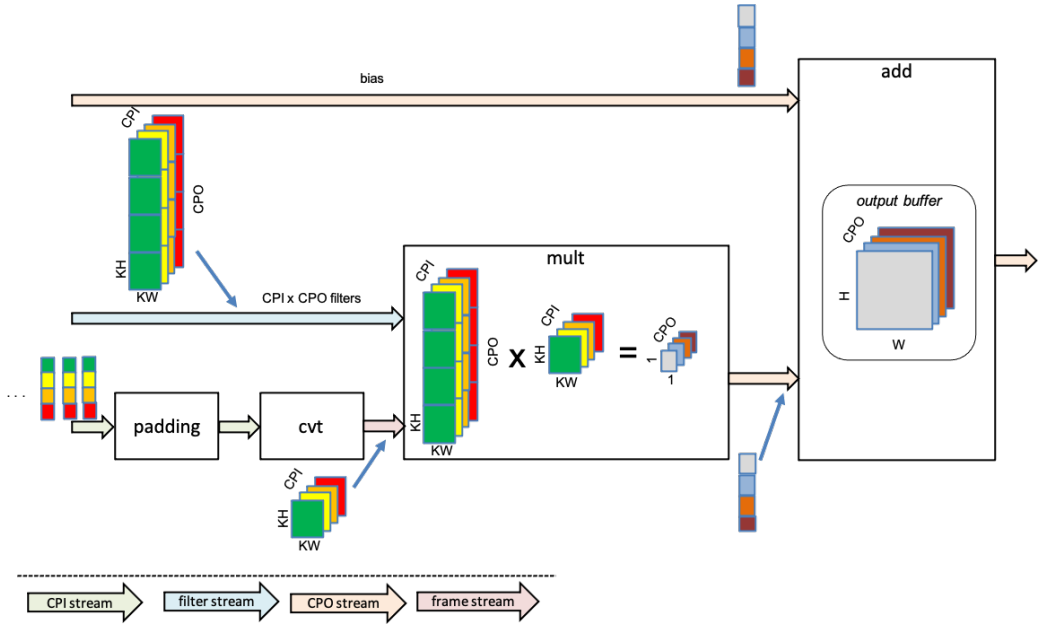

Figure 4.8: Schematic of the components of convolution in an FPGA

padding section, the convolution section, the multiplication section and the addition section. The padding section will add the necessary padding to the incoming input data. The convolution (cvt) section will convert the data into frame structures ($K_h \times K_w$) and output a frame stream. The multiplication (mult) section will perform the multiplications of the data with the DW kernel, and then the PW kernel, producing an output. The add section will simply add the bias values to the data values. The output of the add section will be the output of the DWS layer. For an illustration of this, see Figure 4.8.

The kernel weights will first be split into two streams, the depthwise weights and the pointwise weights. They will be loaded in the multiplication stage. The bias weights will only come in to the picture after the convolution is done, in the addition stage.

The input data will be passed through the padding stage, which will add on additional zero values in the corresponding positions. The padded values will then be joined into frame structures to be passed together in a frame stream in the multiplication stage.

The DW kernels will be multiplied with the data frames, and summed up according to their channels, producing a pixel with the same number of channels as the input. Each value of this pixel will be multiplied by the PW kernel, for each input channel there are as many values as there are output channels. This will produce a final output pixel with a value for each output channel.

Finally this output pixel has the corresponding bias added. There is one value for each output channel. This final output is then stored in memory.

The inputs that will be passed to the forwarding process *data_in* will be in the format of arrays of the kernel size, and each input will come in the format of a single step in the convolution by selecting a KW x KH sized square/rectangle of the input. We will call this a frame. Each step, will iterate across the width of the input image by the given stride size, and then at the end of each horizontal passing, will move the frame selection back to the left-most position and moving the frame selection down by the size of a stride. This will allow us to take advantage of the pipelining nature of the dataflow HLS implementations for FPGAs.

Note that while traditionally FPGA designs have been usually programmed with hardware description languanges, the advances in HLS tools in the recent years and the

algorithmic structure of the CNNs make HLS being a suitable tool for implementation of this designs.

For the depthwise section, we will store the computed intermediate value in the *aux_dw*[] buffer. In order to obtain the final values of the intermediate array, we must iterate through every input channel, for which we will use the iterator *cpi*, and for each channel, we will be iterating through the size of the kernel or frame $KW \times KH$ with iterator *j*. The size of the intermediate array is simply the number of input channels. For every corresponding value of *j* in both the input selection and the depthwise weights, the multiplication of the two is summed to their corresponding CPI position in the intermediate array.

```
loop_mul_cpi_dw:
for (int cpi=0; cpi<CPI; cpi++) {
  #pragma HLS PIPELINE II=1
  loop_mul_j_dw:
  for (int j=0; j<KW*KH; j++) {
    #pragma HLS PIPELINE II=1
    #pragma HLS UNROLL
    aux_dw[cpi] += data_in.pixel[j].pixel[cpi] * kernel_dw.pixel[j].pixel[cpi];
  }
}
```

For the pointwise section, the resulting values will be stored in the *sum* array, before being added to the *p_out* structure and being passed to the output pipeline.

For this part, we will be iterating across the output channels *cpo* and the input channels *cpi*. This time, the output is an array of size CPO, for which in each CPO position of the array, all corresponding CPI positions of the PW kernel and the intermediate value (from the output of the DW step) will be multiplied and summed.

Finally, the intermediate value will be cleaned up and set to zeroes in preparation for the next step of the convolution, and the values of the *sum* array are passed into the *p_out* structure.

This iterative process is shown in the code below. As before, we use UNROLL and PIPELINE pragmas to maximize the parallelisms and to avoid the introduction of time stalls in the computation pipeline of the alogrithm.

```
loop_mul_cpo_pw:
for(int cpo=0; cpo<CPO;cpo++){
  #pragma HLS PIPELINE II=1
  #pragma HLS UNROLL
  loop_mul_cpi_pw:
  for (int cpi=0; cpi<CPI; cpi++) {
    #pragma HLS PIPELINE II=1
    #pragma HLS UNROLL
    sum[cpo] += aux_dw[cpi] * kernel_pw[cpo].pixel[cpi];
  }
}
for(int cpi=0; cpi<CPI; cpi++) aux_dw[cpi] = 0.f;

pixel_out_t p_out;
for (int cpo=0; cpo<CPO; cpo++) {
  #pragma HLS PIPELINE II=1
  #pragma HLS unroll
  p_out.pixel[cpo] = sum[cpo];
  sum[cpo] = 0.f;
```

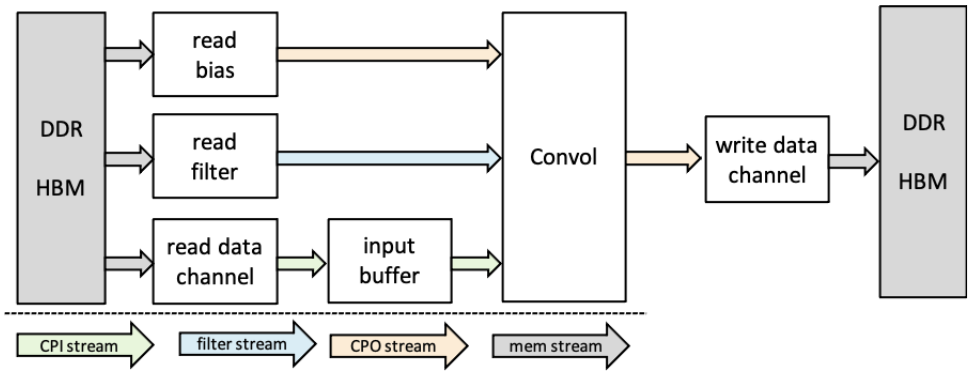

Figure 4.9: Dataflow of the DWS kernel on an FPGA

```
}
out << p_out;
for(int iterator = 0; iterator < CPO; iterator++) p_out.pixel[iterator] = 0.f;
```

After the output of the convolution process is obtained, the pipeline will pass the values onto the bias addition section which will simply sum a given bias value to all the output pixels. This is no longer part of the convolution process.

4.2.2. Data flow model

In summary, the dataflow of the DWS layer on the FPGA is as follows (for illustration see Figure 4.9):

The input data is read into FPGA memory in chunks (read_data_channels), and then, the chunks are serialized into pixels (ch_serialize_and_filter). This is the "read data channel" component. The following step consists of taking the individual pixels and grouping them into their corresponding RGB positions for each color-pixel (join). This is the "input buffer" component.

Now the bias and the kernel values are passed into the dataflow from memory (read_bias & read_filter). In the case of the kernel stream, two different streams are created, one for the deepwise weights and another for the pointwise weights.

Now all four of the streams, (input, 2 kernels & bias) are passed into the convolution section (conv) "Convol" in order to perform the actual convolutions required. This section is composed of 4 subsections, the padding addition stage, followed by the extraction of input values into the shapes of the frames that will be used to multiply against the kernels, then followed by the multiplication of the input data frames with the kernel frames.

After performing the convolution multiplications, all the outputs are added to their corresponding biases (one bias value per output channel). Lastly, the computed values are stored back into memory from the final pipeline "write data channel", stored into the original block format the inputs came in.

4.2.3. Expected gains of the DWS

It is expected that the memory occupation of the DWS configuration will be significantly smaller than that of a standard convolution of the same input, output and channel sizes. However this should only apply once above an output channel and kernel side size of 1.

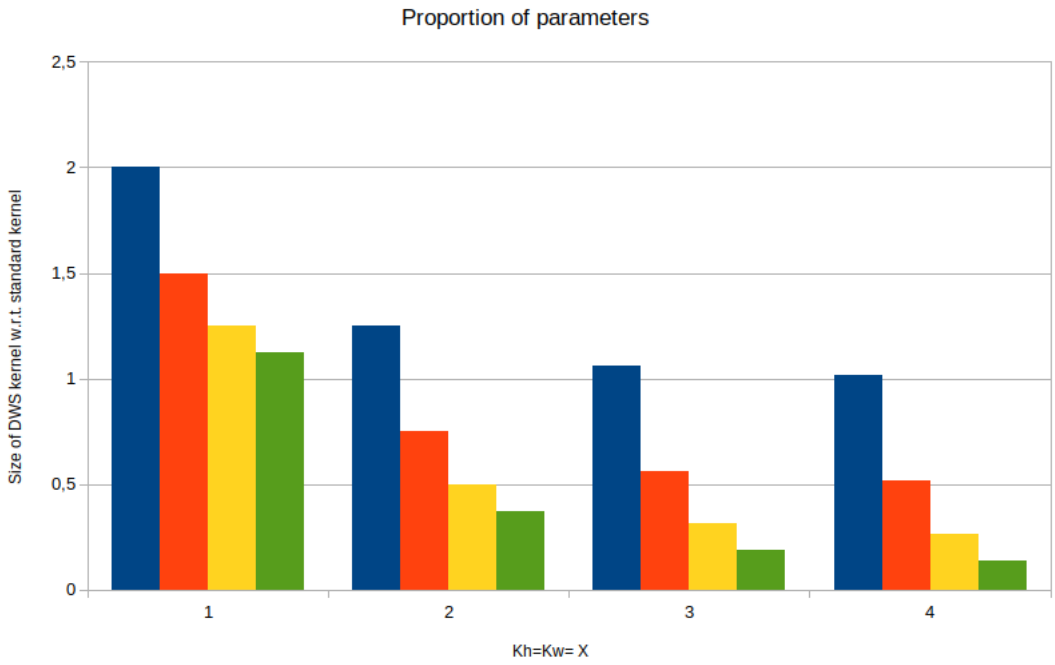

Figure 4.10: Chart showing size of DWS kernel with respect to a standard convolution kernel in terms of parameters. Blue: CPO=1, Orange: CPO=2, Yellow: CPO=4, Green: CPO=8

As seen in chapter 2, the number of multiplications required for a standard convolution is ${K_d}^2 \times {O_d}^2 \times I_c + {O_d}^2 \times I_c \times O_c = ({K_d}^2 + O_c) \times I_c \times {O_d}^2$ and the multiplications for the DWS configuration is ${K_d}^2 \times {O_d}^2 \times I_c \times O_c$. These exact formulas also represent the total number of parameters that are required for the convolution to function.

This means that we should expect the memory occupation of a DWS kernel to be around $(\frac{1}{O_c} + \frac{1}{{K_d}^2})$ times that of a standard convolution. So as the output channel size or the kernel size is increased, so should the difference in memory occupation. This is visually represented in Figure 4.10. In the evaluation section we will show how our overheads match with our expectations thus, confirming the efficiency of our implementation.

4.3 Integrating DWS in a neural network model

Neural networks are composed of many different types of layers to perform the training and inference process. Rarely will a single layer suffice or be appropriate for a classification problem that cannot be resolved in another way.

In order to study the behaviour of the DWS configuration in a realistic scenario, it must be integrated with other layers and even with multiple instances of the DWS, varying in channel sizes.

When defining a neural network, what we are essentially doing is connecting the dataflow between different modules. Each module in this instance, is a fully implemented NN layer. Some of the layers that we will use for our analysis shall be the following:

- Fully connected: This layer connects every input neuron with every activation function of the next layer.

- ReLu: This layer is an activation function that forwards the inputs it receives only when they are greater than zero. In all other cases, it outputs zero.
- Softmax: This is another activation function layer that takes all of its inputs, and operates on so that they sum to 1. This is to represent them as a distribution of probabilities.
- Maxpooling: This layer will simply group inputs into quadrants and then return the maximum value that is being received from each quadrant.

CHAPTER 5
Performance evaluation

In this chapter, we evalute the performance of the DWS. For that purpose, we deploy the DWS implementation in a FPGA and provide results of area overheads, execution time of the DWS in comparison with the regular convolution algorithm.

5.1 Deployment on FPGA

The deployment process required the training of a model on CPU within the HELENNA application. This can be done by training a model with the -sm_fpga option, which allows the user to save the final trained model for future use specifically in a FPGA.

Once the trained model has been saved, the trained weight values can be extracted from the model, to be used to create the right "xclbin" of the FPGA implementation model to be uploaded through the bitstream to the physical FPGA device. This is assuming that the device has been configured for the same kernel size and input & output channel sizes as the selected layer in the trained neural network.

The entire model is run locally on the CPU/GPU, except for the selected layer that runs the DWS configuration on the FPGA. This enables us to isolate and study the behaviour of the DWS layer, while also ensuring that the layer is behaving as is intended and expected inside a complete trained NN model.

This section compares the results obtained from training and performing inferences with neural networks that implement DWS convolutions in place of their original standard convolution counterpart. With respect to the training process, we will compare the time it taken to train multiple networks that use standard convolutions and then the same network again but replacing convolution layers with depthwise separable layers. We will then go on to compare the accuracy of both versions of the different networks as well as the inference times for each.

We will be using the following neural networks for comparing standard convolutions and DWS convolutions:

- A: conv-v3/conv-v3-dws: fully connected, convolution1, ReLu1, convolution2, ReLu2, convolution3, ReLu3, maxpooling, fully connected, softmax.
- B: conv/conv-dws: fully connected, convolution, ReLu, maxpooling, fully connected, softmax.
- C: vgg1/vgg1-dws: fully connected, convolution, ReLu, convolution, Relu, maxpooling, Relu, fully connected, softmax

- D: vgg16.1/vgg16.1-dws: fully connected, convolution, relu, batch normalisation, convolution, relu, batch normalisation, maxpooling, fully connected, relu, batch normalisation, fully connected, softmax.

After these comparisons, we will study the results that can be obtained from deploying a single DWS layer onto an FPGA. We will be looking at the resource consumption of the FPGA and also the inference times obtained with the FPGA. The FPGA we will be using to test the DWS layer is the `Alveo U200`.

The limitations of the capacity of the `Alveo U200` to be taken into account, are as follows (see Figure 1.5). This FPGA has at its disposition the following components:

- 1.18k look-up tables
- 6,840 DSP slices
- 4,320 Block RAMs
- 960 UltraRAMs

In this work, we have integrated the DWS layer into an existing FPGA implementation of a neural network model. The original model included the convolutional layer and we have replaced this model with our DWS implemtations. To allow a seamless integration we has used HLS streams to tranfer data between the different layers.

5.2 Accuracy

To obtain a notable difference between the standard convolution accuracy and the DWS accuracy, only a single epoch has been tested, as for higher numbers of training epochs, the difference between accuracies is almost indistinguishable. We can see that over all, there is a slight reduction in accuracy when replacing a standard convolutional layer with a DWS layer. This is due to the fact that less parameters are used to perform classifications with the DWS configuration, however, the differences could be seen as negligible when weighing the benefits for most use cases.

We can observe an average reduction of accuracy from standard convolutions to DWS convolutions of just 7%, the most severe case being that of Network C with an accuracy difference of 20%. It would appear that the drop in accuracy does not depend on the initial standard accuracy very much, as we we have two very similar accuracy values for networks C and D in the standard case, and yet the difference between the accuracy drops when applying a DWS is notable. Both Network B and Network D have a drop in accuracy of 4% with respect to their original standard convolution versions, and their original accuracies had a difference of 37 percentage points. Clearly the problem of predicting the accuracy loss is a complex problem with more factors to take into consideration that do not enter the scope of this project. Over all, this is more than satisfactory and is a very manageable accuracy loss (see Figure 5.1).

5.3 Resource occupation on FPGA

Hardware resource occupation on an FPGA is determined by how many components are used of the total available on the FPGA. The components to list are the FF, the Look-Up Tables, the DSP, the Block RAM and the UltraRAM.

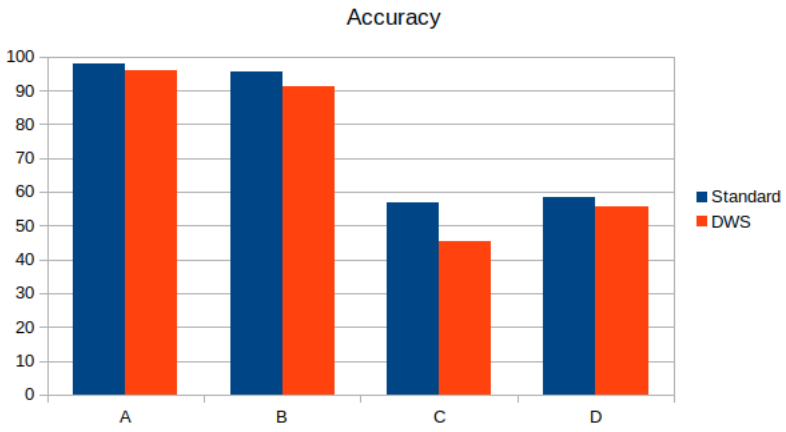

Figure 5.1: Bar chart of accuracies for standard vs DWS

	FF	LUT	DSP	BRAM
CPI/CPO=1	16883	16058	39	47
CPI/CPO=2	21483	19519	46	100
CPI/CPO=4	32462	23836	62	176
CPI/CPO=8	62036	36612	106	352
CPI/CPO=16	168913	67884	242	692
CPI/CPO=32	406259	151838	706	1372

Figure 5.2: Table of hardware resource consumption for different kernel sizes

Here we have the results from running a hardware simulation of the Alveo U200 for the implementation of multiple sizes of input and output channels. In particular the tested channels will both be tested for 1, 2, 4, 8, 16, 32 and 64 (See Figure 5.2). We maintain the kernel size at 3×3 for the sake of simplifying the analysis of the speedup provided.

The maximum quantity available of each of these components for the Alveo U200 has also been provided, in order to better see how the components hold up on the device. The values are included in the bar charts of the memory occupation results. (See Figures 5.3 & 5.4)

As can be seen in the charts (see Figure 5.4), the limiting channel size is CPI/CPO = 64. For this configurahe number of BRAM units that are consumed is superior to those that are available on the device itself.

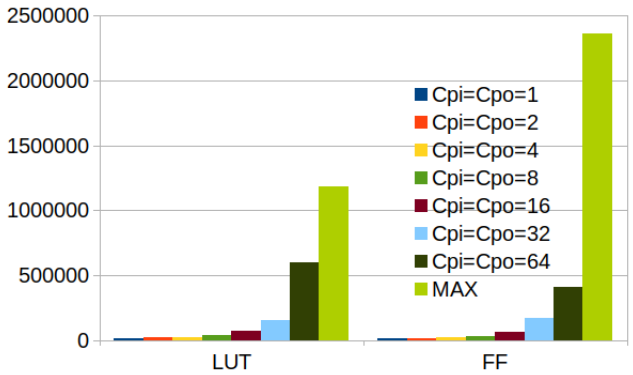

Figure 5.3: Bar chart for DWS convolution of utilised Look Up Tables (Left) and FF (Right) for cpi/cpo sizes 1, 2, 4, 8, 16, 32, 64 and the total amount available

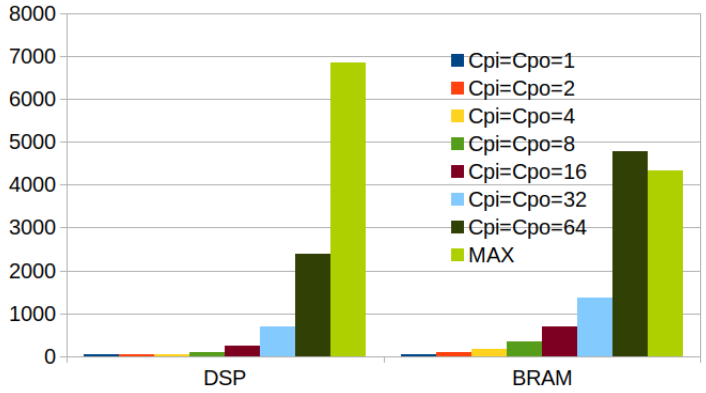

Figure 5.4: Bar chart for DWS convolution of utilised DSP (Left) and BRAM (Right) for cpi/cpo sizes 1, 2, 4, 8, 16, 32, 64 and the total amount available

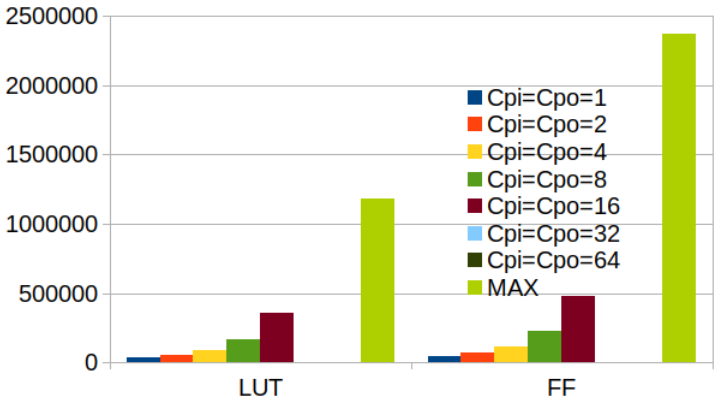

Figure 5.5: Bar chart for standard convolution of utilised Look Up Tables (Left) and FF (Right) for cpi/cpo sizes 1, 2, 4, 8, 16, 32, 64 and the total amount available

The over all trend of the memory costs is an exponential increase in utilisation across all the components. The most abundant remaining resource are the FFs. However, implementing memories with this resources would not be effective since and they are not sufficiently abundant and its use might also impact the maximum operating frequency that can be achieved.

Now we will look at the resource consumption of the same cpi/cpo channel sizes for the standard convolution that already has complete support implemented on the HELENNA platform (see Figures 5.5 and 5.6). However, it must be noted that due to the inefficacy of using standard convolutions for very large input and output channels, sizes 32 and 64 have been deemed unfeasible to emulate as the time it would take is much too long and would consume valuable computing power of the UPV's servers. In any case, the resulting implementation would not fit with the hardware resources of the ALVEO U200.

When comparing the DWS values with the corresponding sizes in a standard convolutional layer, it is evident that the DWS configuration fits much better on the FPGA for much larger channel input and output sizes.

To further explore the benefits of using the DWS over the standard convolution, we will now look at the theoretical speed up seen in chapter 4 compared to the measured speed up from real results. We will compare each type of component independently (LUT, FF, DSP & BRAM)with the theoretical speedup (see 5.7).

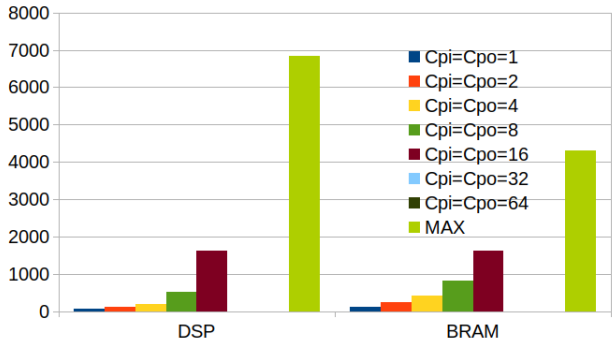

Figure 5.6: Bar chart for standard convolution of utilised DSP (Left) and BRAM (Right) for cpi/cpo sizes 1, 2, 4, 8, 16, 32, 64 and the total amount available

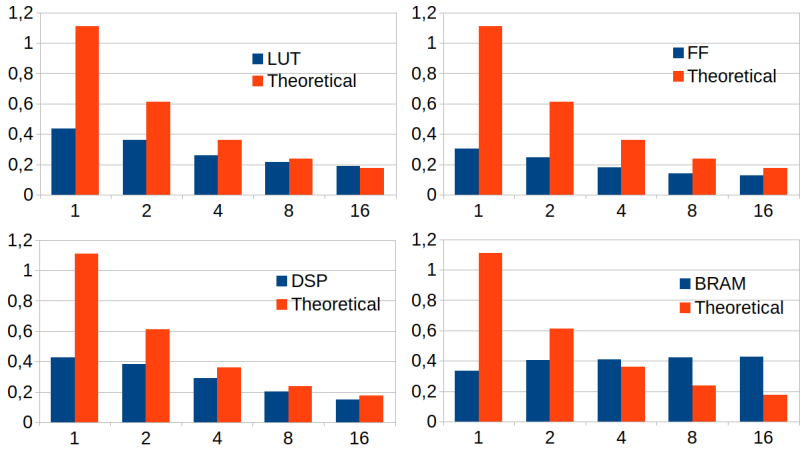

Figure 5.7: Bar chart comparing theoretical proportional memory occupation versus measured proportional occupation for each memory component (LUT, FF, DSP & BRAM). (Standard÷DWS)

The first observation to be made is the vastly lower proportions between the DWS and the standard convolution that are theoretically expected. This can be due to the superior design of the DWS convolution compared to the possibly outdated standard convolution. As channel sizes increase, the proportions begin to approach the theoretically obtained values, except for the BRAM units. The proportion of these would appear to slightly increase with size, but plateau at channel size 8, as we can see, the relative quantity between the DWS BRAMs and standard convolution BRAMs halts with a ratio of approximately 1:6 meaning for every 1 unit, the standard convolution would require 6 for the same size kernel.

Aside from the BRAM discrepancy, for all other memory units, we see a consistent reduction in memory occupation. In fact at the limit of the standard convolution (channel sizes = 16), if accounting for the greatest restriction in memory, this being the BRAMs, we could fit two entire DWS layers for every one standard convolutional layer in the same amount of space.

5.4 Results of execution on FPGA

The execution of the "conv-dws" network using the Alveo U200 was a success. The inference speed of the overall model was 11.95 images per second. We can observe that the model is running as expected on the FPGA, providing a final accuracy of 95% on the MNIST dataset. The time consumed per call for the DWS component in the model was

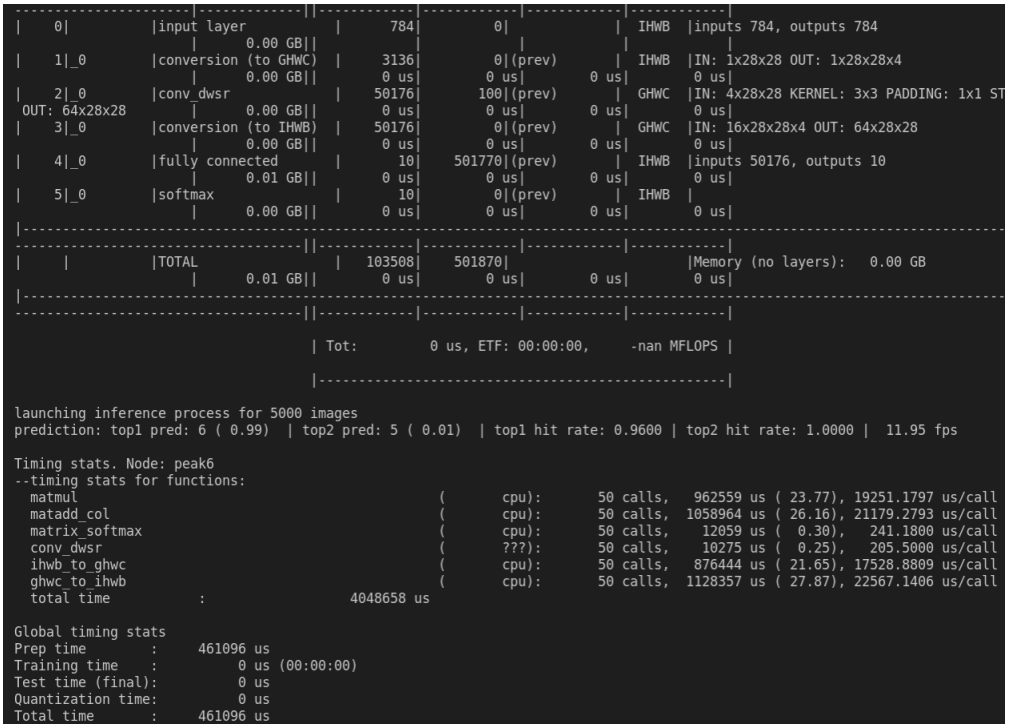

Figure 5.8: Demonstration of complete and correct execution of DWS layer within a model on the HELENNA platform

the shortest of all the components, with a speed of 205 microseconds per call. Over all, the FPGA succesfully executes the DWS structure with values trained on the HELENNA platform (see Figure 5.8).

As we can observe, the DWS component of the model "conv-dws" is the fastest layer to process within the entire NN, even though it is joined with an additional ReLu layer. This is a satisfactory demonstration of the viability of utilising an FPGA to accelerate DWS convolutions for NNs in general and CNNs in particular.

CHAPTER 6
Conclusions

From the various comparisons and tests done with the Depthwise Separable Configuration and the standard convolution, it is evident that the DWS configuration has a great potential of use in both environments with restricted amount of hardware resources and in environments requiring higher computational requirements. There are very few costs to using this configuration in replacement of a standard convolutional layer other than minor accuracy loss on the scale of up to 4% that has been observed in the tests done.

FPGAs prove to very effectively fit DWS layers with more room to spare than they do standard convolutions. The maximum size of channels for a 3x3 standard convolution kernel is 16, and the maximum channel size for DWS is 32, which is double the size.

The task of uploading a trained DWS layer to a FPGA is a very simple process and replacing old trained weights with new ones is also a straight forward process.

The emulation of the FPGA on software as well as hardware drastically facilitates the debugging process for figuring out where execution is failing and by how much. The use of Vitis analyzer further helps create a an efficient implementation beyond basic functionality and guides the user in how HLS can be used more effectively to eliminate internal bottlenecks.

The speedup provided from applying a DWS layer in place of a standard convolution comes from the drastic decrease in memory occupation with respect to the relatively insignificant loss of accuracy. More layers can be fit onto devices when using DWS convolutional layers. This room for more layers allows higher levels of parallelism and deeper neural networks to be produced which could actually allow for more accurate inference results, in particular for deep neural networks.

The major contributions of this work can be summarized as:

- Integration of the DWS functionality in HELENNA with optimized training process in CPUs using OpenMP.
- Development of the DWS layer with HLS for efficient inference in FPGAs and integration of this kernel with a CNN accelerator using a dataflow model
- Evaluation of the DWS implementation on FPGA and comparison with the conventional convolution layer in several configurations.
- Finally, it is notetworthy to mention that the DWS implemented in this work will be used to accelerate several use-cases being developed in the context of the H2020 selene project (www.selene-project.eu) that is coordinated by the UPV.

6.1 Future work

Many future projects can be done with the DWS configuration to explore its behaviour in depth. Studies can be done on the nature of the accuracy loss and under which contexts the accuracy loss will be more significant and possibly find limitations to the contexts in which DWS layers can be used.

In the HELENNA platform, support for GPUs for the DWS configuration could be created in order to decrease the training times and reduce response times on desktop computers or even smartphones that have GPUs. Other forms of future work that can be done with DWS layers and FPGAs is combining multiple convolutional layers together on the same FPGA device.

In the FPGA realm, research can be done in the area of training on FPGA devices. The logistics of implementing support for the training process of a layer on FPGAs have not yet been studied on HELENNA, and will surely bare useful information for creating neural network modules that could learn purely from the environment. Conversely, the training process could be used to gradually fine tune neural networks, once initial values are uploaded from a previously trained model.

Other work that should be done on top of this project is to study the effects of replacing all the layers of a large convolutional network, and then adding more layers to the network, filling the freed up memory. It would be interesting to investigate at what point increasing depth and using DWS layers would actually exceed the accuracy from using standard convolutions with less layers. It may be possible that DWS is not only useful for highly restricted memory spaces, but that it can, in some instances, provide a higher accuracy per unit of memory that is consumed by the neural network.

More research that could be done with FPGAs and DWS layers is to create an even more efficient implementation of the DWS layer. As seen in the results, the restricting aspect of the size of the DWS layer on the Alveo U200 was the BRAMs. However, these could be replaced with URAMs to alleviate the demand for BRAMs. It is to be noted that the standard convolution does not utilize URAMs either, so this may be of interest to improve the implementation of the standard convolution in future for the HELENNA platform.

Over all, the DWS and FPGA should be studied further in tandem, as the FPGA allows for the investigation into the computational structures that can be used for creating an entirely self contained NN. This can be used for studying the long term viability of self trained networks and how to make machine learning systems potentially entirely autonomous with minimal computational overhead. Another advantage of creating completely autonomous NNs is that they will be safer, as theoretically they would not require access to the internet. The future of AI, if we so choose, will be made up of self sufficient machines.

We must use FPGAs to look into the logistics of creating machines that learn from unguided training. Brains have not adapted to be purely logical systems, the realm of logic was created by humans to formally address problems. However brains pre-date human logic. Human brains run on intuition, as this is most likely the most efficient way to interact with a highly complex world. For ML systems, we must consider a similar approach for the most "human" problems. We must look into efficiency rather than pure computational power. Heuristics must be found on the cost of computation or memory against the effectiveness of the machines.

One future of AI depends on enormous computers running incredibly large neural networks. The other future will be entire NN circuits on a chip with robust bodies for maximum lifespan.

6.2 ODS

ODS-7. Access to safe, sustainable energy sources for all The use of FPGAs in combination with neural networks can be helpful in the process of providing power which requires predicting power demand with minimum error. Interestingly, despite the fact that the FPGA consumes power, it can be used to significantly reduce overall power consumption due to efficiency. It has also been shown that in scenarios where convolutions are used to analyse images in real time, DWS convolutions require much less space in memory and will also process more frames of data with the same amount of energy and time than a full standard convolutional kernel.

ODS-13. Climate action The adoption of FPGAs is much less power hungry than complete computers as they have much less overhead and unnecessary software running on them when they are designed efficiently. Also, prototyping is made much easier by promoting the use of FPGAs to test circuits rather than for example printing many PCB board circuits that will be tested before finding an error and being discarded to contaminate the environment, followed by ordering more PCBs and mining and consuming unnecessary natural resources, thereby contributing to a greater carbon footprint.

Bibliography

[1] David Silver et al Mastering the game of Go with deep neural networks and tree search Available at: https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf

[2] Marat Dukhan, "The Indirect Convolution Algorithm," Google Research, 2019 Available at: https://arxiv.org/abs/1907.02129.

[3] Kunihiko Fukushima, Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position., Biological Cybernetics, 1980 Available at: http://www.cs.princeton.edu/courses/archive/spr08/cos598B/Readings/Fukushima1980.pdf

[4] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," Google, Inc., 2017 Available at: https://arxiv.org/abs/1704.04861.

[5] Google's AI approach to microchips is welcome — but needs care, June, 2021.. Available at: https://www.nature.com/articles/d41586-021-01507-9.

[6] S. Ioffe, C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Google, Inc., 2015 Available at: https://arxiv.org/abs/1704.04861.

[7] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 541–551, December, 1989.

[8] Rosenblatt, F. *The perceptron: A probabilistic model for information storage and organization in the brain.* Psychological Review, October 1958 Available at: https://doi.apa.org/doiLanding?doi=10.1037%2Fh0042519.

[9] Laurent Sifre, "Rigid-Motion Scattering For Image Classification," Ecole Polytechnique, CMAP, October 2014 Available at: https://www.di.ens.fr/data/publications/papers/phd_sifre.pdf.

[10] Vivado Design Suite User Guide: High-Level Synthesis, December 2018. Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf.

[11] Fred Lambert. *Understanding Tesla's self-driving features: The Autopilot*, June 2015. Available at: https://electrek.co/2015/06/16/understanding-teslas-self-driving-features-the-autopilot/.

APPENDIX A
System configuration

The main necessary components for this project are the HELENNA repository, Vitis and Git. With these basic tools, the work environment can be set up to perform all of the steps that were made on this project. HELENNA requires around 400MB of space in order to be compiled with enough space on a computer.

In order to correctly analyze the fpga performance and memory occupation, the user must also install Xilinx Vitis/Vivado, in order to be able to utilize the Vitis analyzer tool. Installing this will also provide the variables needed in order to compile the FPGA kernels, which is necessary for this project, beyond the simple CPU/GPU training and inference support that is set up already.

A.1 Initialisation phase

The building of the HELENNA application requires first cloning the HELENNA git repository onto the chosen device. Before compiling, the user must edit the CMakeList.txt file by switching ON or OFF the settings for the different libraries that are supported by HELENNA, depending on whether the computer being used has any of said libraries installed on the device. Some of these supported libraries are:

- OCL
- MKL
- CUBLAS
- OPENCV
- OPENCL
- MANGO

Once all the source code is downloaded into a directory and the compilation options are set correctly, a new folder must be created inside the root folder, eg. "build". Switch to this directory and call "cmake ..". Once this has finished compiling, the user must call "make", and the application will be set up. Once this part is complete, the user can perform any training or inference with neural networks on CPU or GPU and save and load neural network models.

In order to select FPGA components to be compiled, the user will have to switch directory from the root to "helenna/fpga_kernels/tests/". Here, the user can select which

FPGA kernels to construct/simulate/emulate by adding or removing the desired layers from the Makefile variable "LIST" inside the "helenna/fpga_kernels/tests" folder. Afterwards, the user will have to switch to the helenna/fpga_kernels/tests/src and make another "make" call, in this case:

```
make -j12 all TARGET=sw_emu DEVICE=xilinx_u200_xdma_201830_2
```

A.2 Identification of devices

The devices used to perform this project were the UPV's servers, peak6 and peak8. And the target device for all of the FPGA testing was the Alveo U200 by Xilinx. My own personal computer was used to initially test the functionality of HELENNA before setting up the environment on the remote systems peak6 & peak8 for performing training, inference and emulation tasks with much more computational power, and thus a higher productivity. Peak 6 is used for the final FPGA testing, as the peak6 server has the Alveo U200 connected to it.