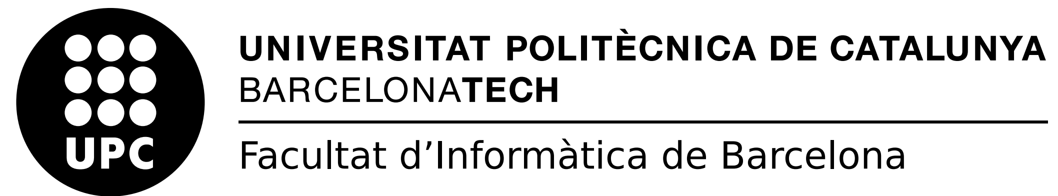

Master in Innovation and Research in Informatics
High Performance Computing

Master Thesis

An Enegy-Efficient FPGA Accelerator for Convolutional Neural Networks

Author: Berta Delgado Ventosa

Advisor: Jose Maria Arnau

Co-Advisor: Antonio González

Date: 25th October 2018

// Abstract

Deep Neural networks are mathematical algorithms inspired in the biological connection of the human brain. This powerful technique currently provides the best solutions for image recognition, natural language processing and other cognitive related tasks. Nevertheless, DNNs are computationally expensive, very time exhausting and power demanding. To cope with this problems different accelerators solutions have been proposed.

This master thesis focuses on a state-of-the-art DNN specifically build for image classification. We develop a new architecture design that will run on an experimental Intel accelerator platform called HARP. HARP combines an Intel Xeon processor connected to an FPGA. During this project we will also asses the possibilities and requirements of this innovative low power solution.

Keywords

Deep Neural Networks, FPGA, HARP, Residual Networks.

Contents

1 Introduction **13**

1.1 Motivation . . . 13

1.2 Objectives . . . 14

1.3 Structure . . . 15

2 Background **17**

2.1 Artificial Neural Networks . . . 17

2.2 Convolutional Neural Networks . . . 18

2.2.1 Residual Networks . . . 23

2.3 Field Programmable Gate Arrays . . . 24

2.3.1 The HARP Platform . . . 25

3 Evaluation Methodology **29**

3.1 Darknet . . . 29

3.2 OpenCL on FPGA programming model . . . 29

4 CNN Accelerator on Hybrid CPU-FPGA Platform **35**

4.1 Original Resnet152 . . . 35

4.2 CNN Accelerator . . . 36

4.3 Optimizations . . . 42

4.3.1 Stream Buffer 42
4.3.2 Vectorization 43
4.3.3 Reducing resource usage 47

5 Experimental Results 49

6 Conclusions 55

6.1 Conclusions 55
6.2 Future Work 56

List of Tables

2.1 Intel Arria 10 resources 25

4.1 Original ResNet152 execution times 36

4.2 Number of channels occurrences per input maps. 44

5.1 Prediction example in Resnet152 50

5.2 FPGA memory usage by number of PEs. 50

5.3 Execution times depending the number of PEs 51

5.4 FPGA resource usage by number of PEs. 51

5.5 FPGA resource usage for different vector sizes. 52

5.6 Bandwidth with 32 PEs 53

List of Figures

2.1 Single Neuron of an ANN Source:[17] 18

2.2 Convolution Operation example. Source:[17] 19

2.3 Filters example. Source:[17] 19

2.4 Max Pooling example Source:[15] 21

2.5 All basic building blocks of CNN. Source:[17] 22

2.6 Left: Plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers.(Thin curves denote training error, and bold curves denote validation error) Source:[8] 23

2.7 Residual block Source:[8] 24

2.8 Cache Protocol in HARP platform Source: [14] 26

2.9 Harp platform overview Source:[7] 27

3.1 Programming Interfaces in OpenCL Source:[8] 30

3.2 Intel SDK channel implementation Source: [6] 31

4.1 PE filter distribution 38

4.2 ResNet152 DLA on FPGA 41

4.3 Naive vectorization 44

4.4 Channel Vectorization 45

4.5 Weight Vectorization 46

4.6 Output Kernel Vectorization 46

5.1 Speedup evolution depending on DSP usage. Baseline configuration is the software version running on Intel Xeon CPU 52

5.2 Intel SDK kernels profiling 54

List of Algorithms

1 Weigh distribution on the PE 39
2 Input distribution on the PE 43

1 Introduction

1.1 Motivation

Deep Neural Networks (DNNs) are sophisticated mathematical models built to simulate the activity of the human brain. This biologically-inspired programming pattern learns from observational data. They have generated a lot of expectations and excitement thanks to its amazing results on speech recognition, text processing and other cognitive related tasks [17].

DNNs have many layers of neurons and a large number of synaptic connections. There are no real restrictions to determine how many layers can be added to the network, but going beyond two layers was unfeasible. The limitation were overcome and Deep Learning appeared thanks to advanced algorithms, the availability of huge data sets for training and the use of accelerated computing platforms.

Each layer of a DNN is used for a different pattern recognition, meaning that each one is in charge of a distinct set of features. The further you advance into the neural net, the more complex the features your nodes can recognize [15]. Nonetheless, adding more layers to the DNN architecture is rather easy but determining the patterns and features that will be better for the network become extremely challenging.

On a quick and simplified overview, deep learning methods are a two-stage process. This first stage consist of training the deep neural network. Its parameters are trained on a labelled set of inputs and compared with its desired output. If the errors are low the

network is already trained, otherwise more training must be performed until is reached the acceptable accuracy. As DNNs are intelligent machines it is awfully critical that they must be as accurate as possible. DNNs can be trained to recognize images, diseases or even to tell which trousers you are more likely to buy next [11].

The second step of the DNN is called inference. The goal of inference step is very different from the training phase. Inference in DNNs consist on classifying the input images in categories (dogs, cars, people...), i.e. making prediction from unseen data, and recognize new unknown inputs. Nevertheless both stages are absolutely interrelated: as the number of images that the network has been trained increases, the classification of new inputs is better predicted.

DNNs training phase has a substantial computational cost and it is very time exhausting. Because of this, different solutions have been proposed in order to achieve better performance rates. The two more common solutions have been to use accelerators such as Graphic Processing Units (GPU) and Field Programmable Arrays (FPGA). These proposals are widely adopted to improve the training and the inference phase of different DNNs models.

Graphics Processing Units (GPUs) improve memory bandwidth in its devices and have adopted GPU-optimized CUDA-based libraries for deep learning. GPUs are inherently good with matrix operations and parallel computations making deep learning accessible. Even they are obtaining good performance rates that are not enough for modern DNNs.

FPGA solutions consist of using this programmable devices due its advantages of fast development, high performance and low power algorithms [5]. FPGA platforms have better performance over generic processors in computational throughput, however they still have its limitations. FPGAs still have not achieved the desired performance due to their limited external memory bandwidth and they are not capable of moving big amount of data between the compute units and the CPU. Furthermore, some designs normally do not take full advantage of all FPGA resources and misuse the vast majority of them [21, 20, 16].

1.2 Objectives

In this Master thesis we develop an FPGA implementation of ResNet152 [8], a modern deep neural network that won the first place in the ImageNet [9] Large Scale Visual Recognition Challenge (ILSVRC) 2015 with a error rate of 3.57%. In this contest, software programs compete to precisely classify and detect objects with the highest accuracy and the best performance.

Resnet152 is a very deep network, having 152 layers. This characteristic makes ResNet a high computation cost network, very time consuming and extremely power demanding. When developing this project we have taken into account the article *"Deep Learning Accel-*

erator on Arria 10" [4] since now it is the most excellent accelerator in an FPGA to the best of our knowledge. In this article they work with a much less deeper DNN of just 8 layers. Even some of their optimization techniques may be applied to Resnet152, not all of them will be suitable, since our working network uses larger data sets. This big inputs will have high memory and logic requirements.

There are some previous proposals to implement DNNs on FPGAs/GPUs accelerators for better performance. Nevertheless, this Master Thesis proposes a solution using an Intel experimental platform named HARP [2]. This platform consist of a combination of an Intel Xeon and an Arria 10 FPGA, with a coherent low-latency interconnections between them. This makes the communication between CPU and the FPGA more lightweight and enhance a new paradigm for workload balance between both devices.

There are three main objectives in this project. The first of all consists of understanding the behaviour of Resnet152 and find out which parts of the neural network are more suitable to develop in the HAR platform. Once this objective is fulfilled, we optimize the performance of the ResNet implementation, improving the power consumption in comparison to a multiprocessor architecture, and analyze the behaviour of the new hybrid platform.

The third objective is to understand and test the performance of the new experimental platform, i.e. HARP. The goal is to study its strengths and evaluate its drawbacks in comparison with other traditional accelerators. This master thesis not only focuses on achieving better performance for ResNet152, but also analyzing how this specific platform behaves when working with very deep neural networks.

1.3 Structure

This Master Thesis is divided in five more chapters. The following section explains the basic fundamentals to understand deep neural networks. It provides an overview of convolutional neural networks and emphasizes what is different on Residual networks since it is the type of network we are working on. Afterwards, the main characteristics on HARP platform and its possibilities are described. Later on, we take a look on how this platform has been programmed and the programming model followed. Next, we will present the developed code and its optimization. Finally, the results are analyzed and a final chapter of future work and conclusions wraps up this work.

2

Background

This chapter presents the basics fundamentals and key concepts required to understand the project done. It does not pretend to demonstrate mathematically how a neural networks operates or which specific training algorithm exists, however it provides a solid theoretical base.

2.1 Artificial Neural Networks

In Artificial Neural Networks (ANN) the basic unit of computation are the neurons, also called unit or node. They are brain-inspired nodes designed to replicate the way people learn. This models are programmed to perform tasks without any specific rule. This means they learn to recognize images by looking in other images previously *labelled*. It can mimic a human baby behaviour, when its born it can not recognize a *cat*, even though when he learns and see a lot of cats, he learns to categorize it as an animal.

Lots of connected neurons build an ANN. Each node connection spreads information from one neuron to the other. Neurons can receive input from external source or other neuron, these connections are called edges. Units and connections have an associated *weight* that indicates how important the input is; this weights are adapted during the learning process. Neurons gather this information and calculate a linear function to the sum of its inputs and, next, they output its result to the next neuron [17]. Neurons are aggregated and assembled into layers, different layers can perform different operations and interact between them.

In Figure 2.1 the node takes input X1 and X2 with its associated weights (w1, w2) plus a b variable. This variable is called Bias and its needed to provide a trainable constant value to the network. The purpose of function f is to provide non-linearity to the output. This kind of functions are called activation functions. Its commitment is to provide non-linearity to the data sets, since in real world data representation is not linear.

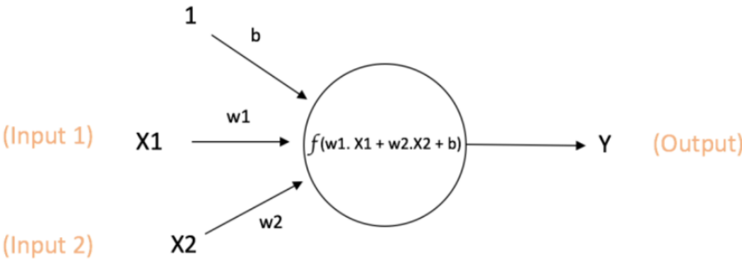

Figure 2.1: Single Neuron of an ANN Source:[17]

2.2 Convolutional Neural Networks

In this project we will be focusing on a particular case of ANN. Convolutional Neural Networks (ConvNets or CNNs) are a category of deep neural networks (DNNs) used in image processing and recognition. The term deep in DNN refers to the number of layers that a network has. It is considered to be a *deep* network if it has more than two layers. As much layers as we aggregate, the complexity of the network increases. On the other side, more layers implies better image classification.

CNNs are DNNs specific to treat and process images. Input images are transformed to tensors, understanding tensors as matrices of numbers with additional dimensions. Therefore, every image can be represented as a matrix of pixel values. To clarify the concept with an example: a normal picture of a camera has three channels (RGB), each channel will be a 2D matrix stacked over the other with values from 0 to 255. To operate with the described image we will need to transform it to a 3D matrix.

These matrices, as explained, need filters to detect certain features in the pictures. These filters are also a matrix of values trained for a specific task, e.g. edge detection. The filters move towards the entire input matrix doing a convolution operation to detect if the feature is present or not. Convolution operation consist on element-wise product and sum between these two matrices. If the feature is detected, the convolution ends with a real number with a high value, otherwise the value is lower meaning the feature is not found. The filter must have the same channels as the input image, so the element-wise operation can take place. We can observe a convolution operation in Figure 2.2. The input being a 5x5 matrix with a

3x3 filter. Figure 2.2 only shows the dot product operation on a single channel of the tensor. This operation needs also to be performed to the remaining channels to obtains a full feature map.

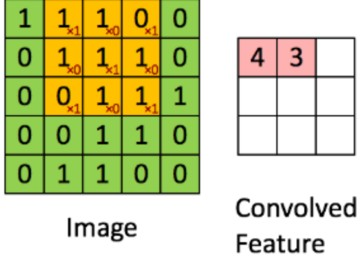

Figure 2.2: Convolution Operation example. Source:[17]

As it can be intuitively deduced, different weight values for the same input map affects directly to the output (features maps) obtained. In practice, the value of the filters is learned during the training phase of the neural network. The more filters the CNN has, the more features it can detect. Therefore, the more filters it has the better it can recognize new images. In Figure 2.3 we can see how distinct filter values affect to the convoluted image.

Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Figure 2.3: Filters example. Source:[17]

The convoluted feature is controlled by three parameters:

- Depth: The number of filters to use in the convolution operation. Each filter will output a different output feature map. The number of filters may vary, but it must always have the same channels as the input feature map to be able to perform the convolution.
- Stride: Is the number of positions that the filter will move on after each convolution operation. For example, when the filter has a stride of two it means that we slide two positions over the input map every time we compute a dot product. If we would have a stride of value two in Figure 2.2, the filter matrix would slide two positions to the right fitting over the two last columns and the three first rows. Having a small stride will produce bigger feature maps.
- Padding: In some matrix multiply paradigms padding is employed to simplify the computation on the border pixels.

Non Linearity

Other important operation in the CNN is the execution of the activation function. As explained before, DNNs need to introduce non-linearity to mimic real world data. To do so, activation functions are typically performed such as [12]:

- Rectified Linear Unit (ReLU): It is a pixel wise operation that converts all negative values of the feature maps into 0 value and leaves the positive values unmodified: $Output = Max(0, input)$
- Sigmoid: It takes the real value of the feature maps and shrink it to the a range between 0 and 1: $output = 1/(1 + exp(-x))$
- Tanh: It is similar to sigmoid activation function but it generates a featured map in the ranges between -1 and 1, corresponding to the hyperbolic tangent function.

Pooling

One of the problems with CNNs is the cost of time and computing power to achieve output features maps for big inputs. For that reason a pooling (also called subsampling or downsampling) step is performed to reduce the dimensionality to each feature map. It must be taken into account that even this pooling step is performed, it still needs to retain the most important information of the feature maps. Pooling also reduces the number of parameters and makes the network invariant to small distortions; a smaller output layer would not change the output of the overall pooling stages. Typical pooling steps are:

- MaxPool: Finding the the maximum of the spatial neighbourhood and creating the new matrix with the maximums found.
- MinPool: Finding the the minimum of the spatial neighbourhood and creating the new cluster with them.
- AvgPool: Computing the average of the spatial neighbourhood and creating the new cluster with the results obtained.
- SoftMax: It finds the softmax function of the neighbourhood. This functions squashes the outputs of each spatial neighbourhood to be between 0 and 1 (like sigmoid function); but it also divides each output such as the total sum of the new cluster is equal to 1. At the end, this functions indicates the probability for a feature to be true.

In Figure 2.4 we can see a graphical example of a Max Pooling operation in a 2D feature map. It uses a 2x2 window over a 4x4 map. This window has a stride of 2, meaning that will slide 2 pixels per operation taking the maximum per each segment.

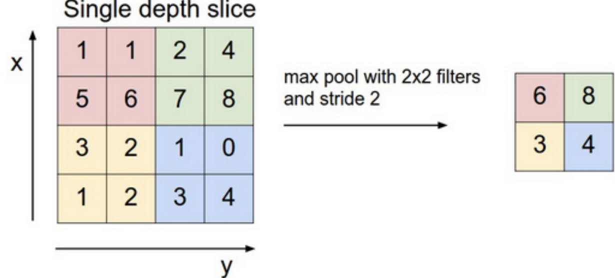

Figure 2.4: Max Pooling example Source:[15]

Fully-Connected Layer

Other important operation for image classification is the fully-connected layer. Fully-connected layer is used in several CNNs. On this particular layer each neuron is connected to the next adjacent layer neuron, using a softmax function in the output layer. This classifies in which final category the image belongs to. At that point the convolutions end and the output prediction is done.

Figure 2.5 shows all basic building blocks of a typical CNN. In a classic CNN, first of all a convolution operation is applied to the input image and, next, a ReLu to introduce non-linearity to the results. Afterwards, to minimize the output feature map (that it will the next layer input feature map) it computes a pooling stage. We can observe the feature map is reduced at this point. This process is followed by a second convolution operation and a pooling step to make the fully-connected layer even smaller. This is done to reduce

memory usage, because fully-connected layers require a lot of memory since it includes all possible neuron to neuron connections between consecutive layers. Eventually, after the fully-connected layer we will obtain the output predictions. This forecast shows the categories that has greater probability to fit in. The better the network the more accurate prediction it obtains.

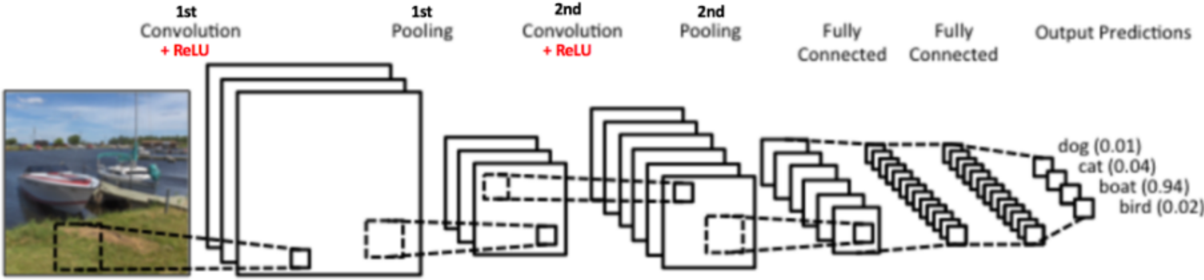

Figure 2.5: All basic building blocks of CNN. Source:[17]

Training

Training the network is an important phase for obtaining an accurate model. It must be done before the neural network starts to make predictions for new images. Nowadays, there are various algorithms to do so, but basically all of them teach the DNN by telling it that the image processed is from a specific category (e.g a cat) and telling the answer again and again until it is enough intelligent to say, *Look! That is a cat!*. This process is called supervised learning.

The Backpropagation algorithm is the most common method used to train the DNNs. It obtains better accuracy on image classification. Before backpropagation algorithm, it was terribly difficult to optimize the networks. What this algorithm does is to train the weights of the connections between the nodes. It will adjust its value depending on how good the network is behaving. Prior to backpropagation, other more elementary algorithms were used. One of the most exploited techniques consisted on adjusting the weights in a random uniform direction (increase or decrease) and observe if the accuracy of the ANN was improved.

Backpropagation algorithm consist on selecting the optimal weight value to estimate the best functions for modelling the training data. To select the optimal values, this algorithm divides the training in two main stages:

- Forward propagation: It evaluates the neural network in the forward direction for the new input image. When the last layer outputs its values, the results are compared with the expected output and the error obtained computed by using a cost function.

- Backward Propagation: It consist on propagating backwards the error obtained. Normally, error is propagated backwards to find the derivative of the error, i.e. cost function, with respect to each weight and, afterwards, subtract this value from the weight value. This mathematical model has been proved to reduce errors obtained in the network.

This process of going forward and backward on the network can go over and over until some threshold of accuracy is reached. Training the network is really time and computing demanding. Fortunately Resnet152, the DNN that this master thesis is studying, is already trained and therefore we do not need to train it again.

2.2.1 Residual Networks

Deep neural network achieve good results and performance in visual recognition jobs due to its benefits from the "deep models". The number of features can be enriched by the number of layers [19]. Therefore, it has been a trend on adding more layers in DNNs in recent years. Deeper networks, theoretically, can solve more complex tasks and improve classification accuracy.

When the network is very deep (above 25 layers), it becomes much more difficult to train an a degradation problem occurs. Accuracy gets saturated and ends degrading the overall results. Residual Networks, a.k.a. ResNets, represents a recent successful approach to address the degradation problem [8]. It consist on introducing *residual connections*, which means connecting the output of previous layers to the output of the current layers.

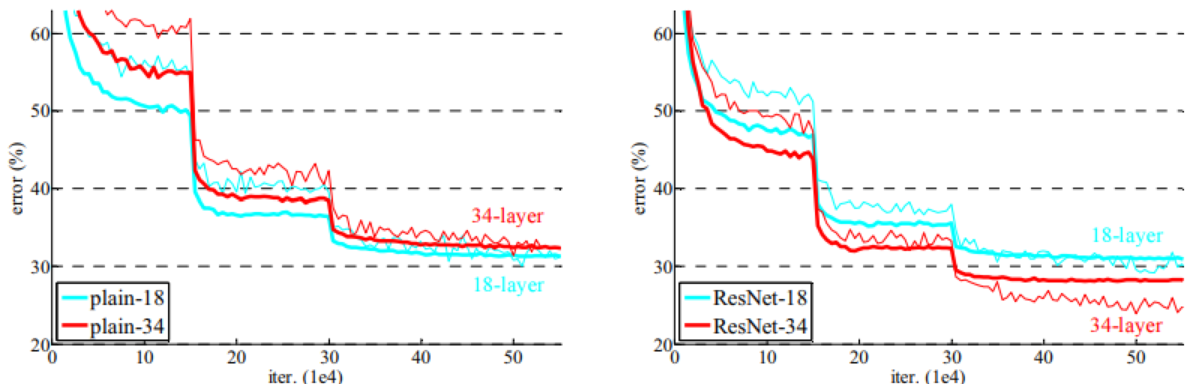

Figure 2.6: Left: Plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers.(Thin curves denote training error, and bold curves denote validation error) Source:[8]

In Figure 2.6 we can see a graphic extracted from [8] where they analyze the performance of the DNN on the ImageNet data set. ImageNet project is a huge visual database designed

to be used in object recognition software research. Its database currently has 14.197.122 images from 21841 different categories [9]. In the left graphic we can see that 18 layers plain networks achieve an error rate of 27.94%, pretty similar to Resnet-18 that gets very close but worse results with a 27.88% error. This difference is reverted when we increase the number of layers. When the number of layers is increased in 34 layers plain network has a higher error rates reaching a 28.54% in comparison of the 25% in the ResNet-34.

For example, given an 8-layer residual network, output from layer 1 will not only go to input layer 2, but also add up the outputs from layer 1 to the outputs of layer 2:

- Denotation of a layer: $f(x)$
- In a plain (standard deep) network: $y = f(x)$
- In a ResNet network: $y = f(x) + x$

This bypass of the output from a previous layer to a subsequent layer is called shortcut connection. This new "layer" does a bit-wise addition of its elements. We can see a graphical example in Figure 2.7. ResNets can be deeper having up to 152 layers (the one we will optimize) and achieving an accuracy of 93.8%, even though they can be slower and more complex than their plain counterparts.

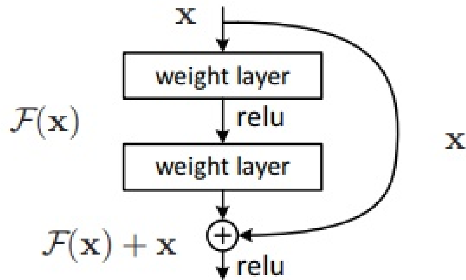

Figure 2.7: Residual block Source:[8]

2.3 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are integrated circuits that are based on matrix configurable blocks and interconnections circuits. They can be programmed with different requirements after manufacturing. This flexibility allows to adjust its design to the neural network requirements, contributing to an increase in performance and a reduction of the power consumption with respect to conventional general purpose processors.

2.3.1 The HARP Platform

Heterogeneous Architecture Research Platform (HARP) program, is an Intel project that works on a platform based on a single chip, which integrates an FPGA and an Intel Xeon processor. This is such a revolutionary design that allows to rethink the usual architectures used for image classification. HARP platform enables a disruptive hardware solution that adapts perfectly to the software being execute on it. This architecture is targeting big workloads on data centers and application-customized hardware.

We need to move to a more heterogeneous computing architectures like HARP, to go beyond homogeneous parallelism and incorporate the strong demand for silicon specially tuned for machine-learning techniques. Is it true that shrinking transistors have powered decades of advance in computing. However, we are achieving the physical limit of transistor density per chip and we can no longer rely on Moore's law to achieve better processing power.

Intel Arria 10

As explained, the HARP prototype used for this master thesis consist of an $IntelXeonE5-2600v2$ product family with an FPGA module $Arria10$ [3] connected via QPI bus. There exists also more oldish models that uses $AlteraStratixV$, other FPGA device with less computing power than the actual Arria 10. Intel Arria 10 is a high performance FPGA, it can provide 20GB/s bandwidth with the processor and has the following specifications:

LUTS (LookUp Tables)	427,200
REGISTERS	1,708,800
BRAMS (Block DRAM)	2,713
DSP (Digital Signal Processors)	1,518

Table 2.1: Intel Arria 10 resources

System Overview

HARP FPGAs have a coherent low-latency interconnection and full access to system memory. Intel Xeon is connected though a Quick Path Interconnect (QPI) bus to the FPGA. This QPI bus has a speed of 6.4 GT/s [7].

The processor and the FPGA communicate using a cache protocol, as shown in Figure 2.8. In HARP, the FPGA hardware module for communication is composed by the Intel QPI channel and the AFUs (Accelerated Function Unit). QPI connects the FPGA to the CPU last level cache. The cache allows zero cost in copy data buffers between the multiprocessor and the FPGA. In the best case, we would obtain a read hit and we would

not need to access main system memory. AFUs can access to the last level CPU cache (which is coherent) but they can not implement a second level cache since they are outside of the coherence domain of the CPU. In this organization the FPGA and the CPU share the same address space, this simplifies the data sharing and it largely reduces the overheads of offloading computation to the FPGA, avoiding transfers between CPU memory and FPGA memory.

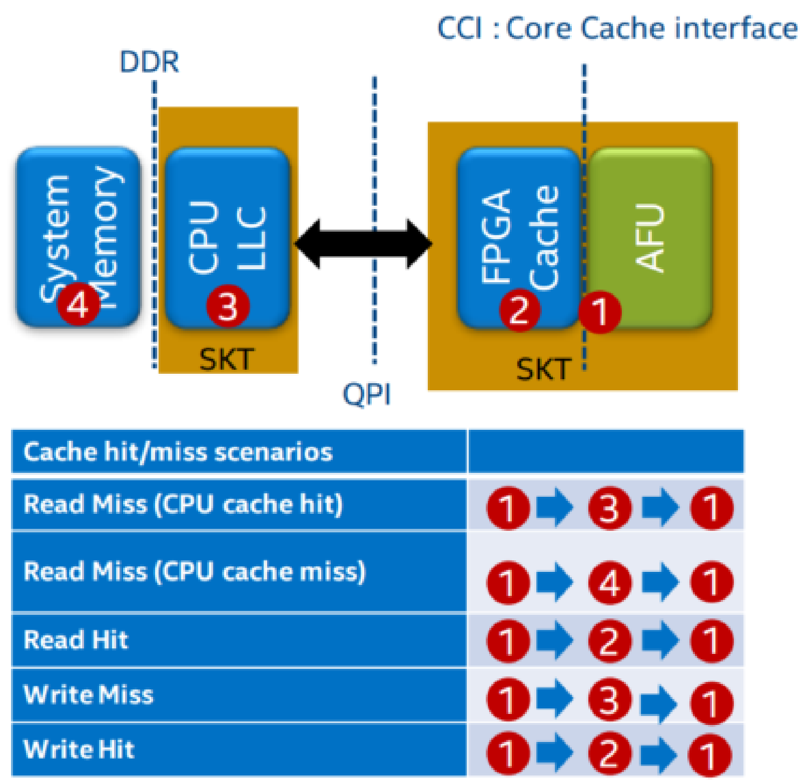

Figure 2.8: Cache Protocol in HARP platform Source: [14]

In Figure 2.9 we can see the overall architecture of the platform. Apart from the QPI bus, this FPGA also has an expansion connector PCIe 3.0 for extra I/O connections (e.g Ethernet). This platform can exploit a good workload balance distributed between the host and the device obtaining hybrid algorithms.

Working with HARP platform requires part of the code running in the processor and the other part running on the FPGA side. Normally, FPGAs need to be programmed in a low level programming language such as Verilog or VHDL. Early versions of HARP machines only enabled this programming languages. It was a high cost programming task since there were too many options of configuration being decided by the programmer and backward and forward compatibility with different FPGA generations was very difficult to achieve.

Nowadays, HARP allows to have unified application code that is abstracted from the

hardware environment in order to port algorithms across different generations and families of CPUs and FPGAs. Intel provides a specific SDK for OpenCL for developing heterogeneous applications [6]. Intel also provides different libraries and drivers to simplify the programming procedure to communicate host/device.

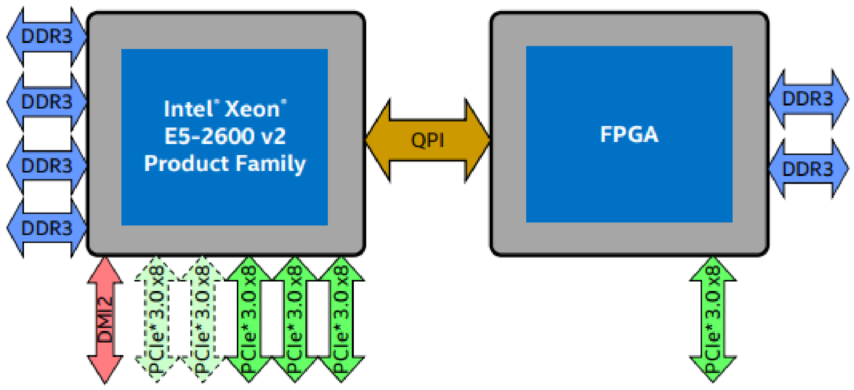

Figure 2.9: Harp platform overview Source:[7]

A normal program pipeline executed in HARP would consist on a program running in the Intel Xeon part, then at certain point it triggers a call to the FPGA. The FPGA starts its computations and return its results to the multiprocessor (host). Once the host has received its results it ends the connections with the FPGA (device). It means that in the CPU side, the user only needs to prepare the attributes for the services implemented and does not need to worry on how the FPGA implementation is done, just needs to rely on its results. On the other hand, the FPGA does not need to care about the implementation in the host.

Calls between host and device need to prepare a pointer system that allows to access the region of shared memory for read and write access, to be able to pass the parameters between them. The CPU also needs to take care of the activation signals of the FPGA and control when it needs to be stopped; due to the Intel's provided libraries these processes are rather easy and does not degrade performance. Finally, FPGA needs to receive the activation signals and execute its functions that as said can be programmed in Verilog, VHDL or OpenCL. Once it has finished it will send an end signal to the processor. Host will recover its state and carry on with the program execution.

3 Evaluation Methodology

3.1 Darknet

To do this research we have employed the ResNet152 implementation on Darknet open source neural network framework [13]. Darknet is a pull of neural networks written in C, providing a high-performance CUDA version for each network for GPUs, and an efficient implementation for multicore CPUs by using OpenBLAS library.

Each DNN found in Darknet has its own configuration, weights and it is already pre-trained. It is up to the user to choose which type of DNN needs to be run. Darknet is an interesting framework because its networks are trained using the Imagenet dataset for image recognition. On Darknet website all the pre-trained models can be found with the accuracy and performance they have obtained on the Imagenet ILSVRC challenge.

3.2 OpenCL on FPGA programming model

The OpenCL (Open Computing Language) is a standard language for parallel programming of heterogeneous systems. It is an open, royalty-free standard for cross-platform devices used in personal computers, mobile phones and embedded platforms [10]. It differentiates itself from other languages since it is not a proprietary programming model; it is based on ANSI C (C99) with extensions to extract parallelism. OpenCL also includes an application

programming interface (API) for the host, i.e. CPU, to communicate with the device, i.e. FPGA, traditionally over the PCIexpress. Nonetheless, in this thesis we use it over the Intel proprietary channel QPI explained in the previous chapter.

We have decided to implement this master thesis problem in OpenCL for a more quickly development environment. The program developed will be portable to future FPGA platforms, in this way we abstract FPGA details. OpenCL implementations still deliver good performance in comparison to Verilog or VHDL implementations [4].

Kernels

In Figure 3.1 we can observe how the programming model fits over the HARP architecture. Kernels (OpenCL functions) can communicate with the host over the cache. There exist OpenCL functions to coordinate the calling of the kernels with the execution of the code in the host. Normally, the host calls the kernel (or kernels) and blocks itself until FPGA signals it to return to program execution.

This kind of OpenCl architecture also has its drawbacks. The limitations of the OpenCL design are mainly that some resources (I/O pins, registers..) are used for basic host/device communication and, therefore, the programmer can do nothing to reduce the resource usage for this part. Another constraint to take into account of this programming model is that OpenCL APIs are bounded to the thread that creates the OpenCL context, meaning that the user can not create multiple kernel queues or wait with a secondary thread the kernels completion.

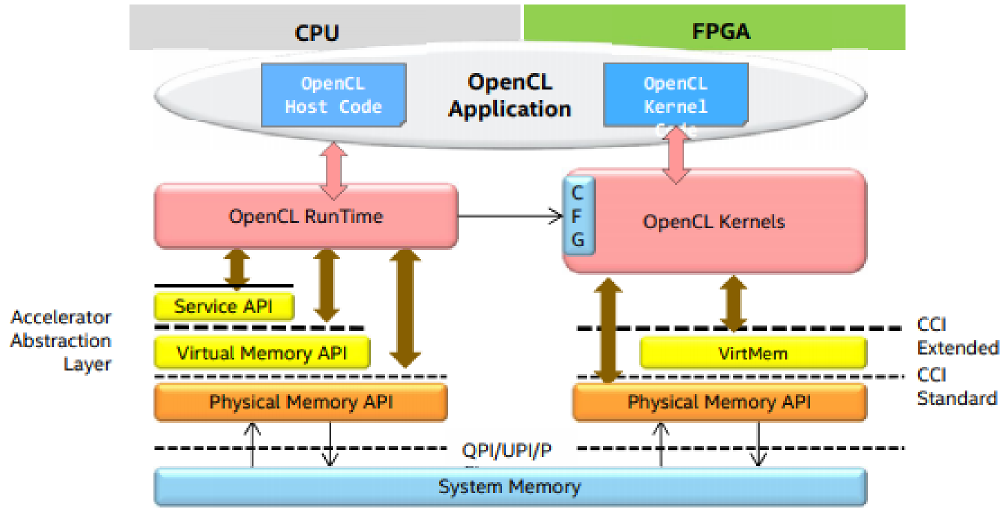

Figure 3.1: Programming Interfaces in OpenCL Source:[8]

Autorun Kernels

Furthermore, Intel FPGA SDK for programming allows to have *autorun* kernels. This specific kind of kernels omit the logic to communicate with the host and start to execute automatically with any explicit call. They restart automatically when they finish its execution. Even though, we had to be carefully with our implementation since they do not support either I/O channels nor any arguments.

Autorun kernels can be replicated in order to have several compute units doing the same task without the need to replicate the code. One important feature for the programmer is that they can be distinguished by its id. The function *get_compute_id()* return a valid ID in a range from 0 to N-1, being N the number of autorun kernels created by the programmer.

Channels

One of the key concepts in OpenCL are the channels. They are a special extension originally designed by Altera corporation. They are a special structure that handles in/out data. However, they can not read and write concurrently. They can be blocking calls or not blocking calls, depending on the design requirements. These channels are used to pass data between kernels, no matter if kernels are autorun or not or even between a normal kernel and an autorun kernel.

Channels do not use global memory. If the data is too big, they are translated as a BRAM inside the FPGA. The Intel FPGA SDK for OpenCL channels extension allows kernels to communicate directly with each other through FIFO buffers [6]. Channels are important because they decouple data movements between the FPGA and the host.

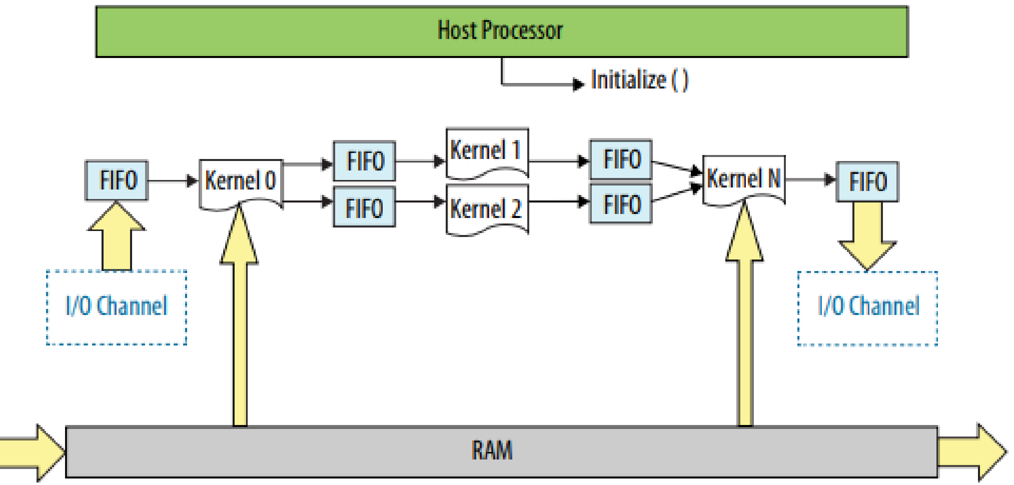

Figure 3.2: Intel SDK channel implementation Source: [6]

Compiler

In order to be able to execute a C/OpenCL program in the Harp platform, we need to compile it using the Intel SDK compiler. It is a source to source compiler that generates intermediate Verilog code for the FPGA from the OpenCL code provided by the programmer. At the end it creates a bitstream from the OpenCL code. The bitstream is a binary file that describes how the low-level FPGA resources have to be configured to implement the required function. Currently it is almost transparent to the user to upload code to the HARP platform. The steps to do it are:

1. Generate the bitstream using the Intel SDK compiler for OpenCL.
2. Connect to a Harp machine and create a project directory.
3. C compiled objects need to be copied in a directory explicitly called **host**.
4. Store the bitstream files (.aocx and .aoco) in the project directory.
5. Enqueue the job in a HARP node that supports OpenCL, or connect a session to a HARP node and run the jobs there.

This workflow for compilation to execution is very time consuming. To generate a complex design bitstream, it can take an average time of 6 hours. The bitstream generation for this project takes up to 7 hours to be generated (circa 8/9h if you are not lucky, and assuming the design is feasible to place in the FPGA). This has supposed an obstacle since a little change on the design represents a whole day of compilation to obtain the results. Furthermore having access to the remote cluster of HARP machines has been sometimes difficult due to overcrowded nodes.

In this master project, compilation step and bitstream generation has been done in an old version of HARP Machine v1, owned by the UPC's investigation group ARCO. This represents an advantage since there are not many people using this resource. Nonetheless, when running the design it has been used the newest HARP platform, owned by the University of Paderborn, in Germany.

Simulator

To overcome the bitstream generation problem and speed up the development process a simulator has been used. Intel SDK provides the HARP user with the Quartus II emulator. The compiler can generate project files from the OpenCL source code and emulates it with the Quartus software. This tool provides functional emulation of the OpenCL code, allowing for quick functional verification of the OpenCL. It checks initially the syntax errors and creates the intermediate files in order to simulate the correct hardware.

This is very suitable and profitable for HARP platform users. We have tested and simulated several architectures in the simulator. Although this is a clear improvement on the workflow for testing the designs, the emulator also has its drawbacks. For a single image classification, the emulator can take more than an hour, just to verify the prediction results are correct. This very same image on a real HARP platform cost only 6 seconds. Iterative development stages take very long even if the emulator is used. Additionally, the simulator does not consider any resource limitations (e.g too many channels or not enough memory) meaning that not all the architectures that work well on the emulator will work well on hardware. This has also been a milestone to overpass in this project, due to some design failing during the bitstream generation process (after 5h) even thinking they would fit in the FPGA.

4

CNN Accelerator on Hybrid CPU-FPGA Platform

One of our main objective is to build a DLA (Deep Learning Accelerator) for the Resnet152 network on the HARP platform. On this chapter we will describe the architecture of the CNN accelerator we have built in order to improve Resnet152 performance. We will also reason and explain the implementation details and the steps followed to achieve it.

4.1 Original Resnet152

First of all we performed an analysis of how this particular DNN was behaving. A main distinctive characteristic is that we will need to operate with a deepness of 152 layers. These layers have different memory requirements. Inputs feature maps size can differ a lot, from just a 0.015625MB up to 9MB, and the same happens to the weights (filters), their sizes range from 0.125MB to 4MB.

It was relatively easy to determinate which part of the network we were going to accelerate. Doing some profiling we found out that the 93% of the execution time of the network corresponds only to the convolution operations. ResNet152 has a total of 152 convolutional layers spending the vast majority of computation resources needed for this DNN. To summarize, we will work with a total amount of 218.47MB needed for filters and 101,625MB

needed for input feature maps. Other operations performing such as average pooling, maximum pooling, shortcuts and the activation functions associated to the convolution layers were only representing the 5% of the total time execution in the image classification procedure.

Studying the behaviour of that network we realized that the convolution layer was already highly optimized for CPUs. As stated before, convolution consists on doing a dot product between the input feature map and the associated weight. The original Resnet152 code implements that operation using a high performance library for matrix multiplication called OpenBLAS [18]. OpenBLAS is a BLAS (Basic Linear Algebra Subprograms) library that provide parallel routines for matrix and vector operations. We discovered that this matrix multiply optimization was having a huge positive impact on performance.

Computing the CNN in the FPGA we cannot use OpenBLAS library, so instead of it, we will use a standard matrix multiply algorithm. To be able to perform the convolution, a certain padding has been added to the input matrix in order to make it easier to multiply input feature maps with its associated weights (and avoid extra control signals). Table 4.1 shows the big impact that high performance library OpenBlas has. The results obtained were executed on an Intel Xeon (1 socket with 2 threads per core and 14 cores per socket, making a total of 28 thread contexts). It shows a 16.60x Speed-Up when using OpenBlas compared to the straightforward dot-product algorithm. In the standard matrix multiply execution the percentage of the time doing a convolution is even bigger and it represents the 99% of total execution time.

Resnet152	Total Time (ms)	Convolution Time (ms)	Convolution %
OpenBlas matMul	5.59	5.19	93%
standard matMul	74.28	73.82	99%

Table 4.1: Original ResNet152 execution times

To clarify, even our DLA will be implemented using the standard matrix multiplication procedure, all the metrics and results provided in the next chapters will be in comparison on the OpenBLAS improved version to provide a fair comparison, i.e. we will compare our accelerator with the best implementation available for CPUs of Resnet152.

4.2 CNN Accelerator

In this section we will define the actual architecture of the CNN accelerator. As a big sketch, we will have three kernels that have a communication host-device, and as much as possible autorun kernels that do the hard computational work. In this chapter we will define how the work between this autorun kernels called processing elements (PE) is distributed and how they are interconnected.

We are building an DLA just for the convolutional layer since convolutional layers computation suppose the 93% of total time execution in Resnet152. It is not worth using more resources (adders, multipliers, registers...) in other layer, given their low computational cost compared with the convolutional ones. Additionally, HARP allows you to easily and efficiently calculate some layers in the CPU and others in the FPGA.

Convolutional Layers

To improve the throughput and performance we need to exploit the parallelism from the dimensions of the matrices (inputs + filters). Since we have retrieved big input and output features map, enough parallelism can be extracted from breaking the convolution operation; the computation is broken in dot-product operation processed in each PE; this will also help us to use as much as possible the available resources in the FPGA.

It can be applied as many filters per layer as required, but the filters always need to have as many channels as the input features map does (seen as a 3D matrix, they need to have the same third dimension value). Based on that convolution operation assertion, each PE will receive the input feature map and just one of the filters (e.g. one 3D matrix) to compute with them the dot-product operation. Therefore each PE at the end of its execution will have computed a channel of that layer output feature map.

To explain better the weight and input distribution we take a look at Figure 4.1. For example, first PE needs to receive the whole input feature map illustrated in grey, and one filter illustrated in a green matrix $(size * size * l.c)$. This particular PE generates the output feature map 0. The ideal case would be to have as many PEs as filters $(l.n)$, this would guarantee that at the end of the PEs computation a whole output feature map would have been generated. The optimal fitting case is unfeasible because in the worst case the number of filters per layer is too high $(l.n = 2048)$. In order to solve this problem, we call for the CNN accelerator multiple times with the remaining filters to apply, therefore to solve one convolution layer we may have several calls to the FPGA.

Input kernel and Stream buffer

Based on the OpenCL on FPGA programming model, the program has a device kernel named Input Kernel that communicates with the host. Its main objective consists of receiving and storing the input data. This entry point to the CNN accelerator is connected to just the first PE (autorun kernel), showed in Figure 4.2. This will feed the adjacent PE; PE_i will pass it to PE_{i+1} and son on.

PEs broadcast data every cycle in a linear daisy-chain fashion. This is more efficient placing than a point to point connection with the input kernel, since the FPGA is a 2D

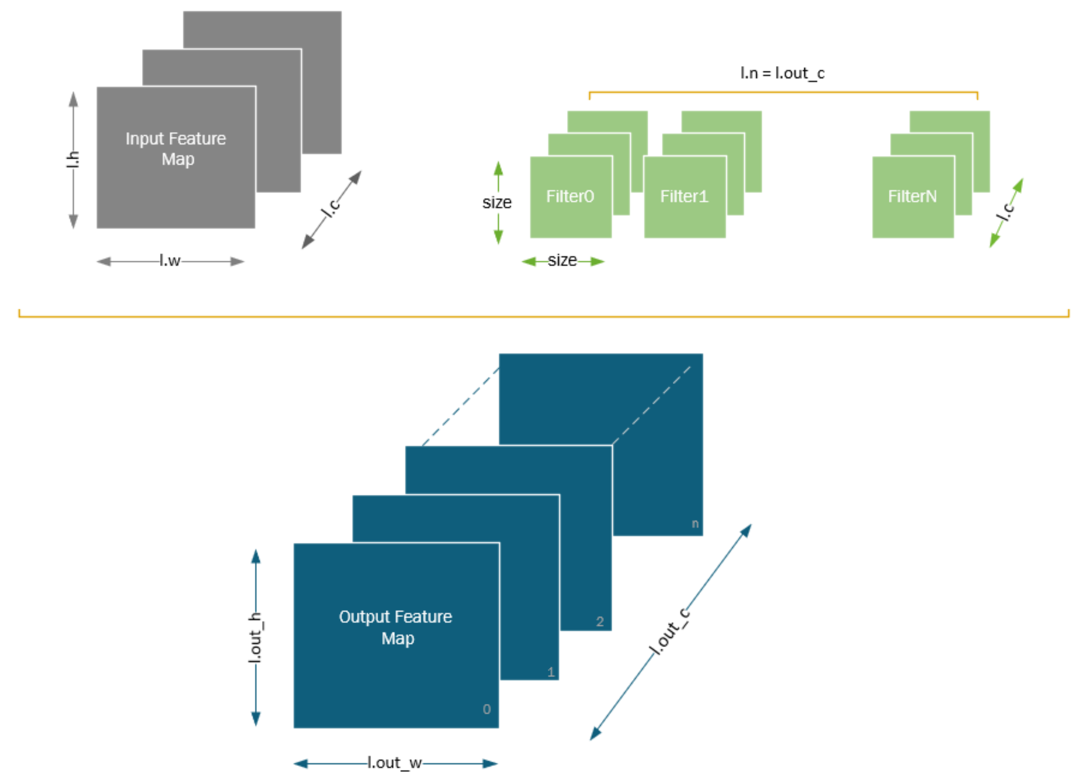

Figure 4.1: PE filter distribution

grid of logic. At that moment, it seems an acceptable idea to, once the input is received, store the whole input for each PE so they can operate with it. Nevertheless, this option is unfeasible. A full input map can demand 9MB of memory, assuming 32 PEs we would need 288MB used only for input storage.

To solve the storage problem, the Stream Buffer was created. This control block introduces some kind of intelligence to the input kernel. It supplies only the needed data for each dot product step. Stream buffer allows to not store the whole input feature map and receive at every dot product cycle just the chunk of the input map needed for the specific operation. For example, if PE_0 operates with a filter size of $2x2x3$, Stream Buffer will only need to provide in the fist iteration to PE_0 $input[0][0][0]$, $input[0][1][0]$, $input[1][0][0]$ and $input[1][1][0]$ to compute the first dot product in the channel 0 of the feature map.

Stream buffer complexity increases when we take into account the stride. If the stride is 1, implies sending the same value over the channels more than once. Even it can have performance issues, it is better than storing the whole input maps over all the PEs. In all the cases, channel connections between input kernel and PE, and PE with PE are blocking. PE_{i-1} is not able to start a new computation until PE_i has read all the sent data.

Weight Fetcher

Weight fetcher is similar to input kernel, it is also a connection between the host and the device. Essentially, its fundamental objective is to feed the first PE with the weights needed to compute the convolution. For doing so, we encounter a similar problem as in the input distribution; we can not store all weight on each PE or we would be out of memory pretty soon. Fortunately, the amount of a single filter is not that high, in the worst case a single PE should store just 2304 bytes. Following a linear daisy-chain fashion the filters are distributed among the first PE; PE_0 takes its weights, and it distributes to PE_1 the remaining filters.

As an example, if we have 64 filters with $7x7x3$ size, all of them $(7x7x3x64)$ are distributed to PE_0. PE_0 then keeps the first $7x7x3$ weights and forwards the remaining $(7x7x3x63)$ weights to PE_1 and so on. The last PE does not need to send any information. A snippet code of that procedure can be seen in algorithm 1.

In Figure 4.2 Weight Fetcher can be depicted as *Weight Fetcher*. As we can see, it has a connection to the first PE, and is this PE the one in charge of following the filter distribution to the chain. However, it is not in charge to decide if all filters fit in a single call to the FPGA. In the host side it exists an offset value, when setting the kernel arguments, it can be decided which chunks o total filter will be transferred to the FPGA.

Algorithm 1 Weigh distribution on the PE

```
Pe(i): Dot Product
while w < total number of weights do
    dataWeight = read Channel(i)
    if w < single filter then
        dataWeight[w] = dataWeight

    else if getComputeId() < NUM_PROCESSING_ELEMENTS - 1 then
        write channel(i+1) = dataWeight
    ++w;
end
```

Control Unit

As it names suggest it serves as a control unit for all the designed elements. It contains the width, height and number of channels of the input feature map. It also contains the the number of filters and the size of each layer. Input, output and weight kernels receive information from the CU (Control Unit). Every kernel uses the information for different purpose but all of them need to receive it to compute the same number of iterations. The control unit also must take into account the stride applied for the convolution. As it happen with the filters and the input features maps, CU feeds PE_0 and is this kernel the one in

charge to distribute to its contiguous PE.

It may happen that all the filters do not fit in the same FPGA call. When that happens we should store the data in the host and prepare another call to the FPGA with the new filters. In this case, CU is updated and it contains the information with the remaining filters to be applied. At the end, the output feature maps obtained must be merged with the already computed output feature maps from past calls to the FPGA.

Output Writer

Output writer is a kernel also connected with the host. Its main task is to populate the results of the PEs to the main memory. Even its task is theoretically easy it is quite complex to implement, due to the daisy-chain topology. First, the row PE computes a dot product with an input feature map and a filter. When it has the results it start to send it to the next neighboring PE. In the same way PE_1 needs to send the received results from PE_0, calculates its own and send it to PE_2. This process will carry on until it get to the last PE in the row. PE_n will send to the output writer all received results, compute its dot product and then send the last output map result to the output writer.

The output writer does not care if the output maps are all computed in just one call to the FPGA, or if it needs more calls because the number of filters is larger than the number of available PEs. It is the host which gathers the output maps and appends to the previous output maps of the same convolution layer.

Processing Element

PE are autorun kernels, meaning that they are always automatically executing, i.e. they are not invoked from the host CPU. The information control they receive is very important since they have been programmed to do a dot product just when they receive a valid control information. By reason of being an autorun kernels, PEs kernels can not receive any information as a parameter so we need to communicate between kernels and other PEs using blocking channels.

PEs do not have intelligence, they will be blocked by the channels until they receive the information control. At that point they will know which weight size are waiting. They read the exact amount of filters they need, remove from the filter channel and forward it to its contiguous PE. Once PEs have this information they can perform the dot-product operation. While computing, each PE will be sending/receiving the appropriate chunk of input map. As all PEs use the same input map and the filter sizes are the same, they can communicate to share inputs inside the loops. In 4.2 the overall architecture design is showed.

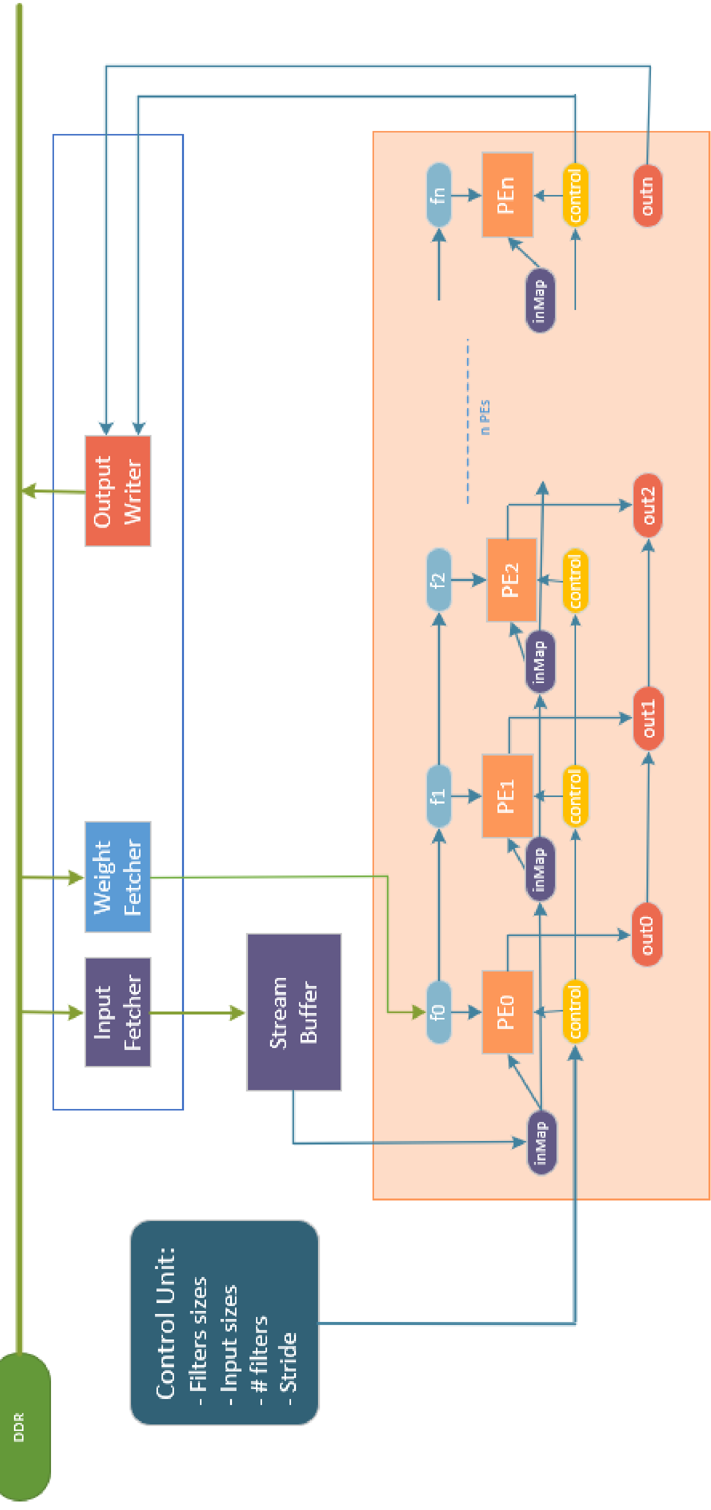

Figure 4.2: ResNet152 DLA on FPGA

4.3 Optimizations

At that point we have explained the basic architecture needed to build a CNN accelerator on the HARP platform. Nevertheless the design is already fairly complex and has been a rough task to implement, its performance is still not comparable to the original version using OpenBLAS libraries. In the results chapter we will discuss this topic. Aiming to obtain better performance results, this chapter explains the optimization done during this project.

4.3.1 Stream Buffer

Stream buffer optimization was one of the most necessary implementations in the early design stages. We realize very soon that if we stored all the input feature map in each PE we could not fit the program in it. We had to be extra carefully because some implementations work properly in the simulator but they are not feasible to map into the FPGA due to lack of resources.

A first attempt to operate with smaller inputs consisted on receiving a chunk of input feature map. As the PE stores its associated weights it can apply the dot product operation over that input block. Afterwards, before operating with the next input map block we could send to the next PE the used input feature map chunk. We execute this approach not without realizing it was not efficient enough. The drawback was that the input feature block maps were still too big. They did not reduce as much memory use as we wanted and it was not as fast as we wish. That was due to the bottleneck we were causing when calculating the dot-product. PE_i was not able to start operating with the first chunk of input until $PE_{0...i-1}$ was finished. Furthermore, PE_i was not able to following its execution until all data was transmitted over its channel.

To solve all the encountered problems, input feature maps will no longer be send by chunks. They are sent one by one every time the PE needs them. This means reading and writing the channels as many times as dot product operations need to be performed. This has not supposed any performance degradation since the data movement between Intel SDK channel extensions are low latency and highly efficient.

We will have as much channels used for input scattering as PEs. We will refer as the total number of PEs as *NUM_PROCESSING_ELEMENTS*. Each PE_i will read from the $channel_{i-1}$ and write to the $channel_{i+1}$. We need to take into account that the PE_0 will be reading the input from a channel connected directly to the Input Fetcher Kernel. On the other hand PE_n must not write to any channel. In the following snipped pseudo-code 2 this phenomena is showed.

It can be observed that the PE do not have control over the input it is receiving. It only reads its corresponding channels and assumes that the obtained value is the correct for that dotProduct iteration. This assertion is true since all the logical needed to determine which input to send is controlled by the stream buffer (placed on the Input Fetcher kernel).

Algorithm 2 Input distribution on the PE

```
Pe(i): Dot Product
while output feature map do
    dataInput = read Channel(i)
    if getComputeId() < NUM_PROCESSING_ELEMENTS - 1 then
        write channel(i+1) = read Channel(i)
    end
    compute();
end
```

4.3.2 Vectorization

In this context vectorization consist on exploiting the OpenCL vector data types that allow to create vector from a given list of scalars. Intel SDK is also prepared to create vector data types for its channels, being able to support communication between kernels sending packs of 4, 8 or even 16 floats. We introduce this optimization to reduce the call to read/write channel without losing the benefits of our design.

Vectorization has been applied to the input and weight feature maps. They can be sent in a pack and computed in the PE. Nonetheless, additional logic utilization will be needed. Before, a PE was receiving a single input and with the corresponding weights it output every time a single value of the output, sending it then to the output Kernel. Now, it will be executing four parallel dot-product operations, assuming a vector size of four elements. Note that OpenCL supports addition and product with vector data types.

Data disposition for vectorization is very important since a bad addressing would introduce overhead. A naive attempt to vectorize the dot product is to just extend the channels to operate with *float4*, receive four inputs per iteration and operate with the corresponding weights. This approach requires an extra reduction at the end of each dot product operation to maintain the coherence. Hence, the filter would be applied in the first channel in the feature map sequentially over the rows and, later, applied the filter to the second channel of the input and so on.

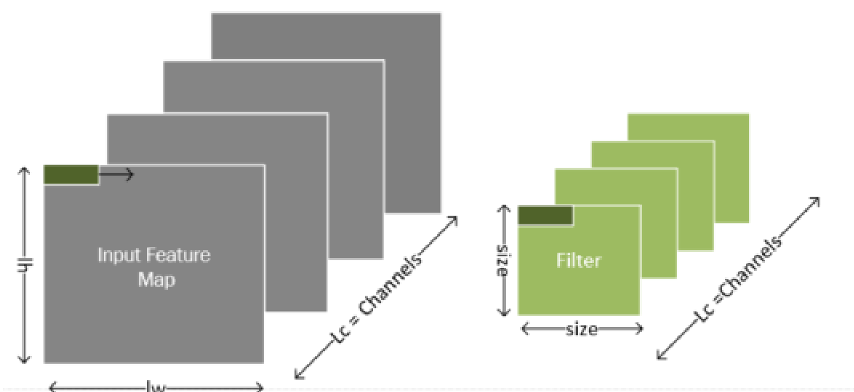

Figure 4.3: Naive vectorization

This approach would work if the size of the filter would be greater than 4x4 filter independently of its third dimension. If that would be true, as seen in Figure 4.3, it would be as easy as receiving four inputs and multiply by its four weights, do the same with the next row and when a block ends slide right the filter and repeat the same operation. Having a look on Resnet152 filter sizes we found that they are extremely small; all of them are 1x1 or 3x3 size and just one layer has 7x7 matrix filter. On the other hand, Resnet152 has up to 2018 channels. In Table 4.2 we can see the number of channels an input map has and how many times these sizes appear over the layers.

Input Channels Size	3	64	128	256	512	1024	2048
Occurrences	1	7	16	75	14	36	3

Table 4.2: Number of channels occurrences per input maps.

A more suitable architecture is needed to fulfill the network requirements. To do so, vectorization along multiple channels has been implemented. It consist on, instead of sending the vector size (e.g four) first elements in the first row of the first channel to the PE, sending the first elements of the different channels in the input map. Doing that, we can exploit the parallelism and compute vector size channels at the same time, we can see it in Figure 4.4. This optimization requires multiple changes in the kernels.

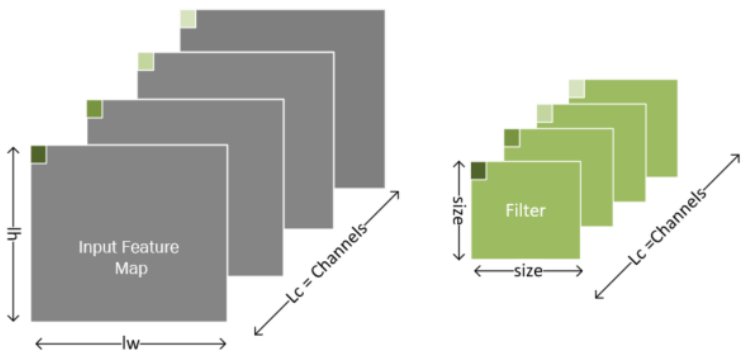

Figure 4.4: Channel Vectorization

Changes to Input Fetcher Kernel and Stream Buffer

Stream Buffer, rather than feeding the PE with always the first channel of the input feature (e.g.: $input[0][0][0], input[0][1][0], input[0][2][0]...$) and increasing the row sequentially, now it has to interleave the accesses through the channel. Following the example in Figure 4.4 with a vectorization size of four; stream buffer first write_channel will contain $input[0][0][0], input[1][0][0], input[2][0][0], input[3][0][0]$. On a bigger matrix, the next iteration would send $input[4][0][0], input[5][0][0], input[6][0][0], input[7][0][0]$.

Stream buffer acting in this interleaved mode demands extra control actions. When the number of channels in the input matrix does not divide the vectorization size exactly, it happens that the vector is not correctly set. To solve this problem, extra padding is inserted to initialize a valid vector. The used padding ends creating a *fake* last channel of input feature map which values are 0.

Changes to Weight Fetcher

Weight Fetcher will act as usual when retrieving the data from the host. However as it will send the filter using vectorization it would need to add extra padding to have valid vectors. If it is needed, it creates a fake last filter channel which will be send to the PE. This may increase the memory usage since in the worst case it will create an additional fake layer to each PE. In Figure 4.5 we can see how a weight fetcher would act. In this example we would have a certain number of filters of size*size*3 channels. Weight fetcher will add a fake last channel to all the filters to be able to create the fake last channel and send them to the PEs.

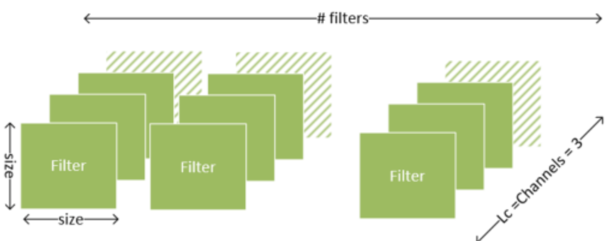

Figure 4.5: Weight Vectorization

Since weights will be sent also interleaved each PE will receive and store its weights ready for the dot product operation. This will allow a better locality for memory access. Each PE will receive the same amount of $size * size * lc$ filter but with different addressing and will store its filters with the new disposition adding the padding. PEs does not have a problem to cope with the padding due to the added channels are set to 0. They will do a dot product but will not alter the final reduction over the vectors.

Changes to Output Writer

Output Kernel does not add extra padding but needs to ignore the possible received extra layer added for the vectorization. Moreover, adding this optimization makes the output kernel to receive the results disordered. In Figure 4.6 we can see an example of how the results are retrieved. This kernel needs to properly rearrange the output feature maps and send them back to the host in the way they are expected. If it would maintain the FPGA new order, shortcuts and pooling stages would fail because they are executed on the host and they are not aware of this FPGA implementation changes.

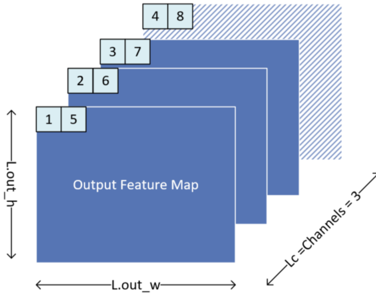

Figure 4.6: Output Kernel Vectorization

4.3.3 Reducing resource usage

The biggest limitation encountered was the lack of memory in the device. To reduce the memory usage we have reduced the CU information struct. Previously we store the data as integers. Now, the data is stored in shorts data type. Also some control information that can be computed was erased from the control unit information. We could reduce from 36 bytes per struct to 10 bytes. Since all this control data was distributed through all the system, assuming 32 PEs and one instance per input, output and weight fetcher kernels we are saving up to 910 bytes (1260 bytes vs 350 bytes). This do not suppose a big difference but it will help to generate the bitstream for the FPGA.

To use bigger vectorization sizes to improve the performance, we decided to reduce even more the memory usage. We were operating with float data types of 32 bits (FP32), we converted them to a half precision floating point (FP16). This drastically reduces the resource requirements of each PE. The conversion from FP32 to FP16 is performed on the data before entering to the PEs and, thus, it only needs to be applied once and it will be common for all the PEs. After all dot-product computation, output kernel need to transform again from FP16 to FP32 since host operates with FP32. To operate with FP16, an especial extension on Aria 10 needs to be enabled (#pragma OPENCL extension cl_khr_fp16). Enabling this extension is compulsory because the FPGA does not support FP16 natively, this leads to additional use of logic. However, at this point, we still have enough logic resources to cope with this problem.

5 Experimental Results

In this chapter we are going to present the results obtained during the development of this master thesis on the different version of the architecture design. As stated, all speedups will be compared with the best Resnet152 implementation found in Darknet repository (which used high performance libraries for matrix multiplication). We discuss the performance obtained and the resource usage of the FPGA.

To test the timing execution we have performed 5 image inferences to the network, is really important to choose in which machine the timings are obtained because they can differ a lot. OpenBLAS library optimizes the matrix multiply operation according to the number of threads available in the processor, it attempts to use all the thread contexts and the vector units of the CPU. We have used an Intel® Xeon® E5-2628L v4 [1]. Just by switching from an Intel® i5-6200U to the Intel Xeon we already gain 14x speedup.

In Table 5.1 we can observe the different images used to infer to the DNN. For these images, Intel Xeon takes an average time of 5.9 img/sec. In the third row, we can see the predictions that the DNN has done. Predictions are very accurate foretelling even the dog breed, however it also fails. It recognizes the horses in the fourth pictures as a bighorn with a confidence of just a 30.13%

						Total Average
darknetOriginal on Intel Xeon	5.62	5.63s	5.52s	5.67s	5.49s	5.59s
Prediction	91.01%malamute 6.8% Eskimo Dog 0.92% Siberian Husky 0.59%bicycle-built-for-two 0.10% mountain bike	86.93% bald eagle 11.23% kite 1.46% ruddy turnstone 0.05% ptarmigan 0.03% goose	74.66% zebra 4.58% impala 2.96% ostrich 1.87% gazelle 1.02% leopard	30.13% bighorn 17.20 water buffalo 16.17% plow 13.94% ram 4.41% ox	76.10% kelpie 4.34% collie 3.24% German Shepard 3.02% llama 2.36 great Pyrenees	

Table 5.1: Prediction example in Resnet152

The results obtained without vectorization optimization were worse than the ones obtained with the Intel Xeon multiprocessor. With 32 PEs the DNN needed an average of 8.23 img/sec to predict the outputs. This had supposed a huge milestone to overcome solved by using vectorization. It significantly improved the performance of the algorithm, we will see it later in this chapter.

The maximum number of PEs we have been able to generate for all the possible versions of the design has been 32 due to limitations of the FPGA. We have also tried to generate a bitstream with more PEs. However, we could not generate the bitstream due to memory limitations. In Table 5.2 we can see how the memory usage grows dramatically when increasing the number of PEs.

# PEs	**4**	**8**	**16**	**32**	**48**
RAM blocks	21 %	23 %	27 %	35 %	64 %

Table 5.2: FPGA memory usage by number of PEs.

Vectorization

An important factor in order to optimize the code, as said, has been to take into account the number of PEs we were setting. In the optimal case we would have, in each layer of the DNN, the same number of PEs as channels in the input map. This may not happen since the number of autorun kernels and channels must be set during compilation time and the number of input channels is variable during the execution. Furthermore, the design must not exceed the FPGA resources. When the number of PEs is increased by two, the execution time is reduced by a factor of two. In Table 5.3 we can see the different execution times obtained with the same design (vectorization size of 4) just by changing the number of PEs.

vec 4	4PE	8PE	16PE	32PE
Convolution Time	38.98	18.95	9.72	5.36
Total Time	39.12	19.39	10.17	5.45

Table 5.3: Execution times depending the number of PEs

Resource usage also increases with the number of PEs. In Table 5.4 we can see the evolution. As the same way as the previous table, we have used the same design with a vector size of 4 increasing the number of PEs. We can appreciate that I/O pins are independent from the number of PEs. This is because this program does not require any I/O pins further from the ones needed to establish communication between host/device. Logic utilization is one of the most demanded resources and it can also restrict the number of PEs. Nonetheless, with 32 PEs there is still enough ALUTs (programmable logic element) , registers and DSP blocks available on the FPGA. In Arria10, DSP blocks consists of basically one adders and one multipliers. With 32 PEs we are just using 23% of the total DSPs. It is expected that a growing use of DSP is reflected with a performance improvements.

# PEs	**4**	**8**	**16**	**32**
ALUTs	130504	142363	1667705	214728
Registers	204052	229283	283442	381184
Logic Utilization	30 %	33%	38 %	49 %
I/O Pins	19 %	19 %	19 %	19 %
DSP blocks	5 %	8 %	21 %	24 %
RAM blocks	21 %	23 %	27 %	35 %

Table 5.4: FPGA resource usage by number of PEs.

Increase vectorization size

We would not like to stuck with the minimum vectorization size (4) and not using all the DSPs available. To increase the FPGA area in use, we have incremented the vectorization size up to 8. This change implied wasting too much logic and memory resources. Again, we were not able to generate the bitstream due to hardware limitations. To decrease the memory usage we have changed the float precision to FP16 to fit the new vectorization size (explained in previous chapter).

Resource usage in this version were heavily increased as can be observed in Table 5.5. The amount of registers needed is more than doubled and the RAM blocks are above the 60% in use. As stated before, I/O pins remains constant. On the other hand, DSP blocks are slightly increased.

# PEs	**32 V4**	**32 V8**
ALUTs	214728	333251
Registers	381184	536389
Logic Utilization	49 %	66 %
I/O Pins	19 %	19 %
DSP blocks	24 %	28 %
RAM blocks	35 %	65 %

Table 5.5: FPGA resource usage for different vector sizes.

This optimization is not improving the performance drastically. This is due that Intel Arria 10 does not support natively the use of *halfs*. To support this feature, it needs to add extra logic for computing the conversions and, thus, reducing the possible performance growth. In Figure 5.1, we see the DSP usage corresponding to different design versions (4, 8, 16 ,32 vector size 4, and 32 PEs vector size 8) and the speedup obtained. The baseline configuration is the software implementation running on Intel Xeon CPU.

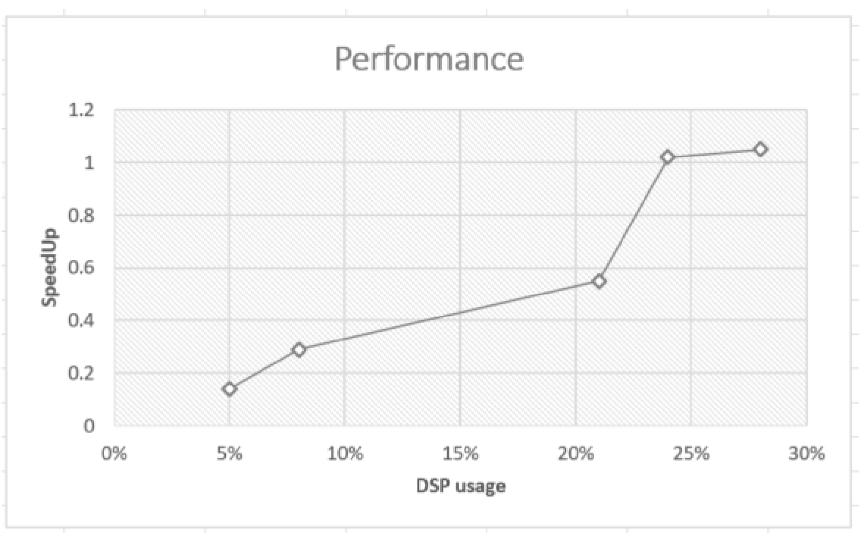

Figure 5.1: Speedup evolution depending on DSP usage. Baseline configuration is the software version running on Intel Xeon CPU

We also check that the results were not altered when using half-precision instead of single precision. Other designs (as AlexNet Intel implementation) also appeal to this techniques. We found that even reducing the float precision the output predictions were practically the same. With a vector size of 8 we have obtained a slightly better version.

Bandwidth

Bandwidth host/device is an interesting point to have a look in the HARP platform. Normally FPGA implementations have bottlenecks when they need to communicate with the host. Not just FPGA, also DNN algorithms running on GPU platforms need memory interchange and lose performance doing it. In HARP this overhead is almost negligible, all the iterations of copying to the device and back to the host is tiny because of the memory sharing architecture.

The Table 5.6 shows the total amount of megabytes read/write to the device. A total of 207744 MB is transferred host/device in just the 0,08% of the total execution time. This allows to have the shortcut and pooling layers in the host. It is not needed to overload the FPGA since it the cost to connect it with the host is negligible. This is one of the main advantage of this platform; the work can be balanced and transferred from the CPU to the FPGA with negligible overhead.

32 PEs	Total Bytes host/device	Total Bytes device/host
Data	102.625 MB	105.119 MB
Time in seconds	0.26	0.29
TOTAL	∼0.08% of total time code	

Table 5.6: Bandwidth with 32 PEs

Profiling

We tried to do a profiling of the latest code version (32 PEs with a vector size of 8). Unfortunately, we were not able to generate the associated bitstream. When instrumenting the code using Intel SDK directives, it automatically enables performance counters, increasing the FPGA area usage which is normally translated to a small decrease in performance. In this case the growth of logic and memory caused the compiler to fail. It outputs an error saying that the design does not fit on Arria 10. To avoid that problem, we have focused on the vector 4 size version.

In Figure 5.2 we can see the time intervals where kernels are executed. Input Fetcher kernel is always executing. This is what we should expect because it is always feeding PE_0 with the needed input values. Output writer executing all of the time means the results are being obtained at the first cycles of the FPGA computation. It starts almost at the same time as the input kernel, retrieving the first dot-product result in just a few milliseconds. On the other hand, Weight fetcher execution intervals are really small. At the beginning we can see the that the kernel is distributing the filters to PE_0. Once it ends writing the channel it stalls until the end of the program. At that moment, the synchronization primitives with the host are applied in order to finish the FPGA call.

Autorun Kernels (PEs) by definition are automatically executed before the host explicitly launches them and they restart automatically on completion. Since autorun kernels never totally complete in a single FPGA call, the profiler does not capture any data. In this case, is the programmer who must instruct the host code with the SDK library call *clGetProfileDataDeviceIntelFPGA* . When adding this call the user can retrieve autorun kernels data. In our case, adding this call also causes the bitstream to fail due to resource limitation. This program is calling the FPGA several times and adding Intel SDK counters infers too much overhead in memory usage.

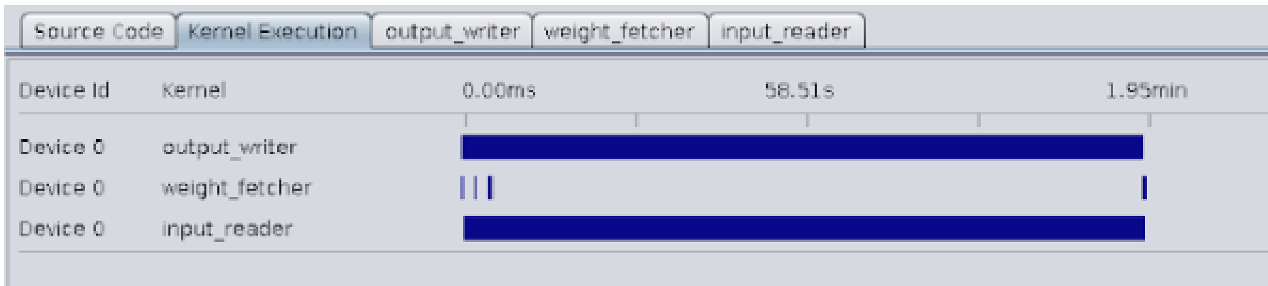

Figure 5.2: Intel SDK kernels profiling

Power Consumption

To summarize, we have obtained a 1.06x total of speedup in comparison to a high performance multiprocessor using OpenBLAS library. Nevertheless, this proposal reduces the power consumption of the design. CPUs are multipurpose architectures that, as a rule, consumes more power than a FPGA. The Intel Xeon used for this experiment has a TDP (Thermal Design Power) of 75 W. When running the image classification algorithms it uses all the CPU available resources, for that reason, we can assume a 75 W TDP. On the other hand, FPGA Arria 10 has a TDP of 35W. In the worst case, this architecture is using only 66% of the total resources of the FPGA thus, achieving a power consumption of 23W. We are enabling low-powered solution for the image recognition problem that does not downgrade the performance of the CNN.

To wrap up, our DLA accelerator achieves 6% speedup, 69.3% reduction in power and 70.9% reduction in energy consumption over a highly optimized software implementation running on a modern Intel Xeon CPU.

6 Conclusions

6.1 Conclusions

It has been a tough task to develop this project. I have learned a new programming language, i.e. OpenCL, to operate with the FPGA. Furthermore, HARP platform is rather new and there are not much studies on how it behaves or how to configure it. One problem that we encountered was that, at the beginning of each compilation stage, it generates an approximation of the resources it will need. At first, it seemed that none of our designs fitted on the FPGA and it was delaying the project. Theoretically it was impossible that this happened, it take time and effort to discover that the approximation that it was showing to the user was a very pessimistic approximation and had nothing to do with the actual results. Afterwards (and after 9hours of compilation) when the bitstream was generated we were able to assert that the resource usage was much lower than the approximated values.

It has been challenging to work with this new environment, finding were the correct logs where and discover a different way to work. We had to cope sometimes with the crowded servers and retry some compilations. When several users were trying to generate a bitstream, it usually fails or lasted even longer.

Other essential point was to understand how Resnet152 was behaving and which were the possibilities to optimize its performance. Studying Resnet152 has been very different from other proposals, even if you are trying to do as much as possible a customized DLA, each layer in ResNet152 has very different dimensions and requirements. Part of this project

has also consisted of learning the state-of-the-art solutions that existed and try to apply some of the best techniques to this project.

We have realized that the inputs are very big, causing to exploit the memory and under-utilized other FPGA resources. If we had not the memory limitation, we could have achieved better design performance, either augmenting the number of PEs or storing at each PE more values to reduce the usage of channels. Nonetheless, with a vector size of 8 we achieve almost a 70% of logic utilization, meaning that this network is very computing intensive. We achieve that 92% of the DNN is executed on the FPGA reducing drastically the CPU resource usage that can perform other tasks.

To summarize, we have obtained a suitable DLA for Resnet152 6% faster than in a high performance CPU counterpart. Moreover this implementation is low power solution. Our DLA accelerator 69.3% reduction in power and 70.9% reduction in energy consumption over a highly optimized software implementation running on a modern Intel Xeon CPU. This DLA could be suitable for other DNNs that apply convolutional layers.

We have also fulfilled the second objective of this thesis that consisted of exploring and studying the behaviour of HARP. Once you get to know the environment is easy to use the HARP platform and work with your designs, but it is rather confusing to getting started with it. In my opinion, HARP platform is a disruptive architecture with lots of potential that can provide good results in a near future. Other DLAs can be designed using HARP because of its advantages in memory bandwidth and low power solution. However, it would be nice to have more documentation about it.

Overall it has been enjoyable experience. I have worked and discovered a new Intel experimental platform while getting to know state-of-the-art neural networks. Image recognition is a trending topic nowadays that still have a lot of opportunities to grow. It has lots of practical purposes with emerging possibilities.

6.2 Future Work

In this master thesis we have obtained a good DLA but there is still future work that can be done. First of all, we could distribute the same data in different ways. We could distribute the input channels on different PEs and apply its corresponding filters. That would suppose that, instead of each PE computing an output feature map, a single PE would compute only a chunk of output feature map. This approach would require less memory usage since the FPGA would receive less filters and just a portion of the input map, but would increase the logic utilization for reordering the results of the feature map when the results need to be send to the host.

Other interesting solution would be to maintain the same functionality in the FPGA

but having non-blocking calls to it. Now, the host blocks and waits until the FPGA send the results. There are ways to configure the host to have non-blocking calls. Doing that, we would obtain an algorithm that balances the convolution work between the host and the device. This approach would require additional synchronization tasks to ensure a correct output map for the next layer iteration is obtained. Different architectures can be proposed and tested in HARP platform to obtain low power and fast solutions.

Acknowledgement

I would like to express my sincere gratitude to my advisers Antonio González and Jose Maria Arnau to bring me the opportunity to work on this topic and provide me with helpful guidance. Moreover, I am thankful that they actively answer all my questions and shares their knowledge. They have been comprehensive and patient through all the adversities that arose during the project.

Last but not least I would like to thanks my parents who support me during all these years. Finally, I appreciate the learning experience I gained from studying at FIB, and especial mention to my fellow classmates, for making my stay here a cheerful and unforgettable one.

Bibliography

[1] Especificaciones de producto de Intel® Xeon® Processor E7-8870 (30M Cache, 2.40 GHz, 6.40 GT/s Intel® QPI).

[2] Hardware Accelerator Research Program.

[3] Intel® Arria® 10 SoC FPGAs Overview - Arria SoC FPGAs Software.

[4] Utku Aydonat, Shane O'connell, Davor Capalija, Andrew C Ling, and Gordon R Chiu. An OpenCL TM Deep Learning Accelerator on Arria 10.

[5] Clément Farabet, Cyril Poulet, Jefferson Y. Han, and Yann LeCun. CNP: An FPGA-based processor for Convolutional Networks. In *FPL 09: 19th International Conference on Field Programmable Logic and Applications*, number June 2014, pages 32–37, 2009.

[6] Optimization Guide. Intel FPGA SDK for OpenCL Programming Guide. 2013.

[7] Prabhat K Gupta. Xeon+ FPGA Platform for the Data Center. *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 119, 2015.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2016.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[10] Group Inc. Khronos. OpenCL Overview - The Khronos Group Inc, 2018.

[11] Michael Copeland. What's the Difference Between Deep Learning Training and Inference? — The Official NVIDIA Blog, 2016.

[12] Hyperparameter Optimization and Model Ensembles. CS231n Convolutional Neural Networks for Visual Recognition. pages 1–18, 2017.

[13] Joseph Redmon. Darknet: Open source neural networks in c. *h ttp://pjreddie. com/-darknet*, 2016, 2013.

[14] David Sheffield. IvyTown Xeon + FPGA : The HARP Program. In *International Symposium on Computer Architecture (ISCA): Tutorial*, 2016.

[15] Skymind. A Beginner's Guide to Neural Networks and Deep Learning — Skymind, 2018.

[16] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-Sun Seo, and Yu Cao. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '16*, pages 16–25, 2016.

[17] Ujjwalkarn. A Quick Introduction to Neural Networks – The data science blog, 2016.

[18] Zhang Xianyi. OpenBLAS: An optimized BLAS library.

[19] Junfeng Yao, Yao Yu, and Xiaoling Xue. Sentiment prediction in scene images via convolutional neural networks. In *Proceedings - 2016 31st Youth Academic Annual Conference of Chinese Association of Automation, YAC 2016*, pages 196–200, 2017.

[20] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2016.

[21] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '15*, pages 161–170, 2015.