Rodrigo Pedrosa Rodrigues

Estimating Optical Flow using Convolutional Neural Networks in Reconfigurable Logic

Dissertation supervised by Professor Doctor Jorge Lobo and submitted to the Electrical and Computer Engineering Department of the Faculty of Science and Technology of the University of Coimbra, in partial fulfillment of the requirements for the Degree of Master in Electrical and Computer Engineering, branch of Computation.

Supervisor:
Professor Doutor Jorge Nuno de Almeida e Sousa Almada Lobo

Jury:
Professor Doutor Jorge Manuel Miranda Dias
Professor Doutor Gabriel Falcão Paiva Fernandes
Professor Doutor Jorge Nuno de Almeida e Sousa Almada Lobo

Coimbra, February of 2020

Acknowledgments

Firstly I would like to thank my supervisor, Professor Doctor Jorge Lobo for giving me the opportunity to learn and work in an interesting area such as the CNN and FPGA field. For the guidance and advice provided.

Would also like to appreciate everyone who I've met and worked with in these years of university.

My close friends and family made it possible for me to conclude my masters degree. My most sincere 'thank you' goes to every single one of them.

Abstract

This dissertation explores the mapping of a convolutional neural network (CNN) architecture onto reconfigurable logic. The end goal is to enable the computation of optical flow using CNNs on an FPGA. This platform is not constrained to a specific CNN, it is open-source and can be extended for extra functionality and different CNN types.

Optical flow is the mapping of movement between two images of the same scene and can be very important for several applications in the Computer Vision field. Giving the ability of ”seeing” movement to a machine can be very useful. The results accuracy is important, but given the real-time requirements of many applications, computation time is a key factor. The goal is to explore trade-offs between hardware complexity, computation time, power, energy consumption and precision using an FPGA.

Recently CNNs have surpassed traditional computer vision methods. FPGAs have been emerging as a rising platform and in this work, we're exploring if there's advantages to a platform like this instead of the more conventional Graphical Power Unit (GPU). By providing a framework that allows flexibility between different CNN architectures we are allowing an easier way for people to work in this field using this type of platform. With the end goal of computing optical flow with CNNs, our work focuses on a specific base architecture, the FlowNet-S, that provides a simple but robust architecture for its size. Using a previously developed open-sourced project called PipeCNN, the expansion of some of its functionalities made it possible to use in optical flow estimation. PipeCNN is an FPGA OpenCL based framework that allows the deployment of classification CNNs in Intel or Xilinx supported boards. It uses Caffe as its base to extract weight and layer deployment information. A CNN that can estimate optical flow requires different types of layers which were not contemplated in the original framework. The added Transposed Convolution layer and the Concatenation layer open new possibilities for the use of PipeCNN.

We have learned that optical flow can be computed with an FPGA and that there are advantages to this system. Although it still doesn't meet the computation timings of GPUs, it can have similar accuracy while spending less power. There are still optimizations possible which could reduce the existing gap between both platforms.

Keywords : Optical Flow (OF), Convolutional Neural Network (CNN), Field-programmable Gate Array (FPGA), PipeCNN

Resumo

Esta dissertação explora o mapeamento da arquitetura de uma rede neuronal convolucional (RNC) em lógica reconfigurável. O objetivo final é existir computação de Fluxo Ótico (FO) através de RNCs numa FPGA. Esta plataforma não está limitada a esta RCN específica, é *open-source* e pode ser expandida para funcionalidades extra e diferentes tipos de RCN.

Fluxo ótico é o mapeamento de movimento entre duas imagens da mesma cena e pode ser muito importante para várias aplicações no campo de Visão por Computador. Dar a capacidade the ”ver” movimento a uma máquina pode ser muito útil. A exatidão de resultados é importante, mas dado os requisitos de tempo-real de muitas aplicações, o tempo de computação é um fator chave. O objetivo é explorar diferentes compromissos entre complexidade de *hardware*, tempo de computaão, consumo de potência e precisão ao usar uma FPGA. Recentemente RCNs têm superado métodos tradicionais. FPGAs têm vindo a emergir como uma plataforma crescente e neste trabalho, vamos explorar se existe vantagem numa plataforma destas em vez da mais convencional Placa Gráfica (GPU). Ao providenciar um *framework* que permite flexibilidade entre diferentes arquiteturas de RCNs estamos a permitir uma maneira mais fácil de trabalhar neste campo ao usarem este tipo de plataforma. Como objetivo final de computar flúxo ótico com RCNs, o nosso trabalho foca-se numa arquitetura específica base, a FlowNet-S, que fornece uma arquitetura simples mas robusta para o seu tamanho. Ao usar um projeto *open-source* previamente desenvolvido chamado PipeCNN, a expansão de algumas das suas funcionalidades fez que fosse possível usar em estimação de fluxo ótico. O PipeCNN é um *framework* baseado em *OpenCL* para FPGAs que permite a implantação de RCNs de classificação em placas suportadas pela Intel ou Xilinx. Usa o Caffe como base para extraír os pesos e informação relativa à sua estrutura. Uma RCN que consiga estimar fluxo ótico necessita de diferentes tipos de camadas que não estavam contempladas no *framework* original. As novas camadas, Transposta de Convolução e Concatenação abrem novas possibilidades no uso do PipeCNN.

Aprendemos que o flúxo ótico pode ser computado numa FPGA and que é uma plataforma promissória. Apesar de ainda não conseguir atingir os tempos de computação de Placas Gráficas, consegue ter precisão similar e consumir menos potência. Ainda existem otimizações possíveis que podem reduzir a lacuna entre ambas as plataformas.

Keywords : Fluxo ótico (FO), Redes neuronais convolucionais (RNC), Lógica Reconfigurável (FPGA), PipeCNN

List of Figures

List of Tables

Contents

1

Introduction

This chapter presents an introduction to this dissertation. Some insights concerning the context and motivation of the developed work are presented, as well as the main goals and key contributions.

1.1 Context and motivation

Sight is one of Man's main senses and it is the most developed and complex. It is a physical experience, how our eyes focus on light that is reflected from objects and shapes and then with that sensory information we transform it into signals that are sent to the brain that then form the images we see. Allied to this there is the concept of vision, which is very different. Which is how our brain interprets everything in the images that are sent from our eyes. It is used for almost everything, for most people, and it leads our everyday lives. It is how we retain most information, how we judge most situations. It guides our way and helps us have a sense of the rest of the world, *what is* everything and *where is* everything in relation to ourselves. The way we know how to differentiate between a dog or a cat, but also how we know if something is moving away from us or moving toward us.

The latter is the topic of this study, it is what is called Optical Flow (OF), the apparent motion of shapes and objects, the relative motion between the observer and the scene. It is perceiving, understanding and predicting the movements of all the bodies in our environment, it allows us to position ourselves in relation to the rest of the world. Having this capability is invaluable to humans and is necessary in a lot of artificial systems as well. Automation of processes that require object tracking is where it finds its most usefulness, most recently self-driving vehicles have used this concept in some form or another, as it allows the cars to, like ourselves, position themselves in the roads the best way possible.

The classical way of estimating OF is by calculating pixel brightness and assume that it stays the same between frames [4], this method has a lot of flaws but it can work in some examples.

There are two variants of OF, sparse and dense. The first one detects the existence of features such as edges and corners, and will create the flow vectors that come from the motion of those particular pixels. One of the most famous algorithms that uses this type of OF is the Lucas-Kanade method

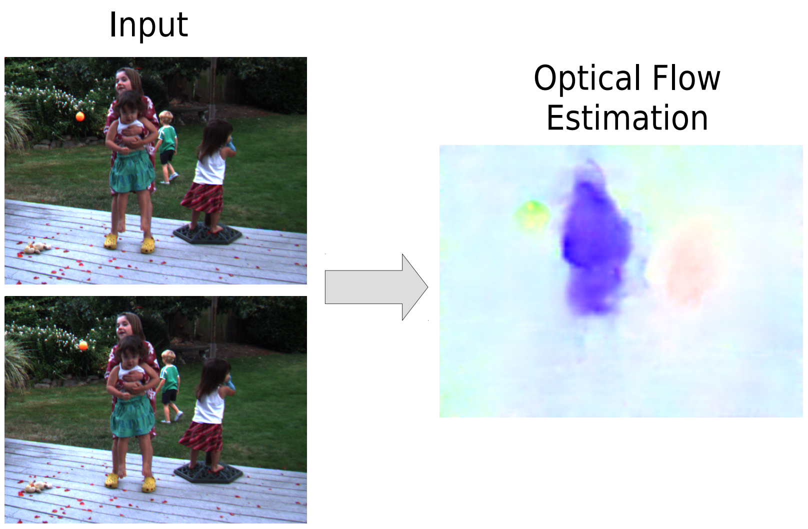

Figure 1.1: An example of what dense optical flow might look like, in this picture we have 2 images that have elements that move, these are fed into our CNN that estimates its optical flow. The optical flow is represented by one vector in each pixel, which are then colored based on their values so a more visual representation is produced

[5], which actually takes the concept of pixel brightness not changing between frames, and through a series of iterations will output the flow vectors of the features detected. The dense OF, however gives a flow vector for each pixel and this, in the context of the full image, will provide us with a more complete mapping of movement but it comes at the cost of some computation power, as there are a lot more calculations being done.

Instead of having complicated algorithms, research has focused on using machine learning which is inspired by biology and how it has the capability of learning. There is a system which uses variable parameters as its drive to correctly predict patterns. It is fed a large database of ground truth examples and through mechanisms that try to minimize the error between the results and the examples that were previously fed, these varying parameters are adjusted through several iterations, after a large enough number of iterations. The parameters are not the same as they started and they turn this system into a machine that can recognize patterns.

Performance is very important in a lot of computer vision fields, but it becomes one of the cornerstones of optical flow estimation because the vast majority of applications require results in real time, as video feed is the primary source of information in these types of systems. Because of this, there is always a trade-off between accuracy and speed, as we trade better results at a slow speed with reasonable accuracy with a boost in computation speed.

This is where the use of a Field-Programmable Gate Array (FPGA) comes in. FPGA can be programmed to have different logic circuits, they can certainly be faster than a Central Processing Unit (CPU) and in some cases faster than a Graphical Processing Unit (GPU). The FPGA can work alongside other processing units as a kind of accelerator where it computes a logic circuit that was previously synthesized and outputs a result that is needed in the pipeline of the system. Because of its very limited resources, the FPGA is a difficult platform to design with but can produce great results without requiring a lot of electrical power.

This is where this dissertation is working towards, having a non conventional platform providing computation power and enabling the acceleration of the process and comparing it with a GPU approach. Some compromises need to be done to accommodate to the FPGA because it is a different type of processing unit, but the main objective is having a working framework where it is possible to use different optical flow convolutional neural networks rather easily instead of having an extremely efficient system that needs heavy modifications if some changes are to be made. This flexibility comes from the use of OpenCL which helps with the programming side of the work but also enables an easier way to test different architectures.

1.2 Objectives

For this dissertation our end goal is to have a working framework which allows optical flow estimation done in an FPGA. Our work focuses on a known CNN called FlowNet-S [7] which provides a good starting point as it has a simple layout and architecture.

Using an FPGA open-source framework which has explored classification CNNs, our work is towards developing extending features which will ultimately allow for the use of FlowNet-S. Although precision is important in this field, due to the real-time nature of most applications, computation time is very important as well. We want to compare how PipeCNN [19] (the framework) can fare against the most broadly used option in the optical flow estimation field, the Graphical Power Unit. By exploring the trade-offs between precision, computation time and power consumption, we want to see if there is any potential for reconfigurable logic in this field. The work done here will not be only for optical flow estimation as some features are used in other types of CNN, namely image segmentation where the transposed convolution layer is used fairly often.

By providing an extension to an already developed framework, we hope to open the door to more exploration in this field.

1.3 Our work and key contributions

There are several aspects that need to be taken into consideration before the start of the development of an FPGA based system, especially the platform which will be used. Different FPGA development boards have very different type of resources. Having a specially powerful and expensive board is not really the priority as we do not want this system to work only with top-shelf equipment, we need to also take into consideration what we have available in the laboratory. Also, we need to decide if we want to build an application from the ground-up or try to use some work that has been done.

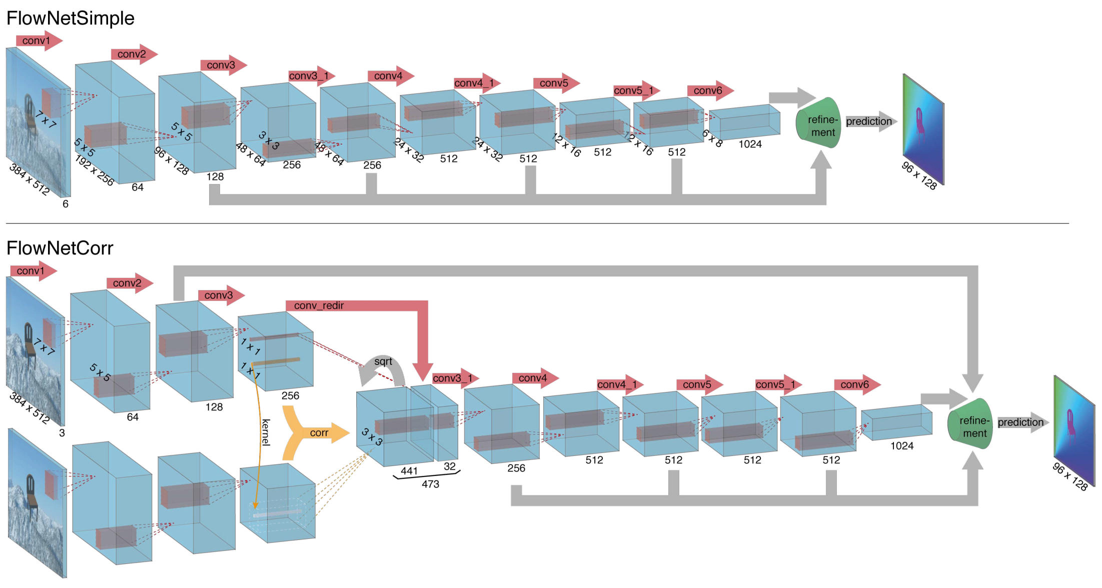

Figure 1.2: Visual representation of both FlowNet architectures, taken from [7]

If we are not building our system from the ground up we should be looking at work done in the field and mainly choosing a working architecture. This way there is no need to worry about training and having a functioning process, we just need to implement the working layers that comprise the architecture that we choose. Not only that but depending on the CNN, there may be different needs especially in terms of memory usage. More complex CNN usually require some information to be saved and used later in the chain of process, which means we need a board that can hold this amount of information.

One of the most prolific projects in the optical flow estimation machine learning paradigm is the FlowNet [7], which paved the way for other research that has been done until the present. Up until that point, most CNN in the computer vision field were used classically for classification, but because of the advancements that were done in semantic segmentation and depth estimation, FlowNet was born. The way CNN work is that they have a chain of layers, and in each layer there is a operation that transforms the data. This cascading of layers then, ultimately, computes the output. These layers are, essentially, operations that manipulate data in some way. The architecture we choose will end up using determines the different layers that we will need to implement. Figure 1.2 illustrates the architectures developed in [7].

The base of all CNN are, obviously, convolutions, but some other operations are also used. In the realm of optical flow, another important layer is the transpose convolution. Another name for this operation and actually more commonly used is *deconvolution*, but since it is not actually the mathematical deconvolution process it can lead to some confusion, so in this dissertation it will be addressed as transpose convolution. The basic idea in these kinds of systems is that, there is feature extraction in the first phases of the architecture and after that these are extrapolated and up-scaled, resulting in a mapping of movement. Contrary to classification networks where the result is a list of values that are compared with a reference file, the output here is the size of the input image with vector values instead of color for each pixel. Both the regular convolution and transpose convolution are the pillars of these architectures and are very computationally intensive. This is where the biggest speedups are achieved, in regards to performance.

In the end, the decision was to use an open-source framework called PipeCNN [19]. This framework was originally developed to use with classification CNN so there are important changes needed to accommodate to working with another type of CNN. Working with OpenCL makes designing for FPGA much easier but it comes with less efficient kernels because of the overhead that exists in a lot of fundamental operations.

The following implementations and contributions have been achieved:

- Expansion of standalone framework that implements the deployment of CNN with the purpose of estimating optical flow. Through the use of OpenCL there's communication between the host computer and the FPGA kernel where the computation happens.

- Alteration of a framework that enables the comparison of different CNN architectures (focused on optical flow estimation). Because of the way it works, all the layers are fully costumizable. It is also easily expandable and can still be optimized.

- Performance comparisons with mainstream's platform, the Graphical Processing Unit. Several parameters are compared, such as: Clock speed, data precision, performance speed, power and energy spent.

- Open-sourced work allows for easier development of new ideas in the field.

- Showed that less energy can be spent to compute the same amount of data.

1.4 Overview of the thesis

The course flow of this dissertation and content of each chapter are the following:

- In Chapter 2 an explanation of some concepts regarding Convolutional Neural Networks, its parameters and general way of working. Also some background information regarding convolutions, transposed convolutions and fixed point data arithmetic.
- Chapter 3 describes the entire PipeCNN development environment, the functional elements performed by the system, and the methods studied for our approach.
- In Chapter 4 a validation and discussion of the paradigms developed in this work are presented.
- Some final conclusions are made in Chapter 5.

2

Background and state-of-the-art

This chapter is divided into three parts, the first section explains some fundamental CNN concepts to facilitate the comprehension of some words and concepts covered in this dissertation. The second section describes some of the fundamentals of important arithmetic operations that appear in CNN. The third section regards some fixed point basics and arithmetics that are important to understand its basic functionality and rules.

2.1 Background on CNN

To understand the main focus, the key points, and the functionality of the layers needed for this dissertation, there was a lot of consultation of [6]. Convolutional Neural Networks are a branch of the Machine Learning field, and they are most commonly applied to the analysis of visual problems due to the nature of how discrete convolutions work. Convolutions have the distinct feature of utilizing the spatial structure of the image itself to convey valuable information and since images are a very structural and spatial type of data, this type of processing works especially well. CNN are, essentially, a cascade of different layers where each layer represents a mathematical function where the input of one is, usually, the output of the layer before it. Through the organization of different types of layers there's feature extraction from the data and these features are then used and transformed into information that is useful, whether it be for classification, or in our case, the estimation of optical flow.

Classification CNN are the foundation of this field and its understanding is very important because the knowledge can be then transposed to a different type of problem. The architecture of CNN are made entirely by a number of different layers that the designer chooses to have. If we look at it from a black box abstraction, our input is an image and the output is a list of which is then compared with a reference file. A coloured image is comprised by 3 intensity values for each pixel, these values are the Red, Green and Blue (RGB) of each colour. We can imagine 3 different images or planes being combined instead of the usual 2D representation. The data inside the CNN is transformed into the list-shape in the output by shrinking in height and width but expanding in depth, so in the beginning we had a 512x512x3 but in the output we end up having a 1x1x1024 (for example), a good analogy is thinking that just like each plane in a regular image gives information about each colour each depth value gives information about different features of the image. See

Fig.2.1 as an example.

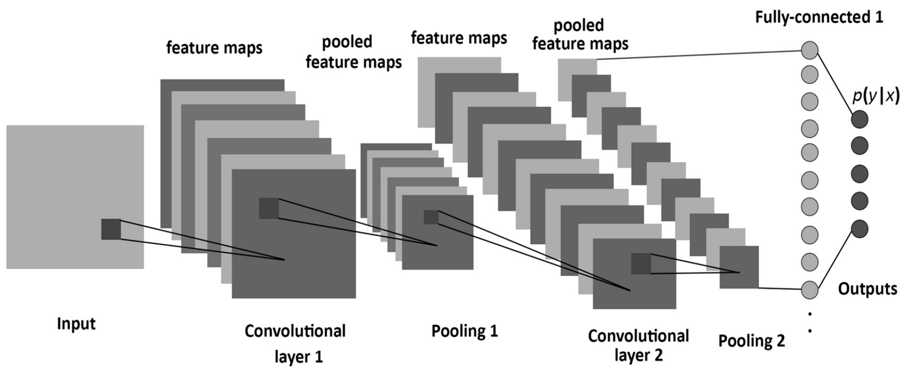

Figure 2.1: Example of a general classification CNN architecture [2]

A typical classification CNN uses three types of layers, the convolution layers, pooling layers and fully-connected layers, there are also activation functions which serve the purpose of deciding between allowing an input to affect the output of that layer and are usually placed after a couple of convolution and pooling layers. The main pipeline is divided into two phases, first there is extraction of features using several layers, usually a convolution followed by a pooling layer, sometimes also with an activation function after the pooling layer, this setup of these three types of layers in succession is repeated for several times depending on what the designer wants to accomplish. The main idea here is that the convolution layers find distinctive aspects in the image such as edges, corners or just blobs of interest points, these form the shapes of whatever is in the picture which in turn enable these to be labeled and classified. The detection of these features is the whole objective and process that enables the classification and for these types of systems to work.

Pooling layers are used for down-sampling, which in other words is the reduction of the size of information, making computation easier since there is less data to process and less parameters to be learned but also because it summarizes the features in a region which means further data manipulation won't be as precise. This *dilution* of information it actually helps the model to be more robust since the parameters aren't as focused and slight variations of the same information are treated the same or almost the same. As in everything in engineering, this is a trade-off, accuracy for more flexibility and speed, so the designer needs to be careful with the amount of pooling layers used, as too many will make the model much less precise but too few and the computation will be a lot slower.

After each layer of computation there's transformation of data from the input into the output, the feature maps are resized, down-sampled, up-sampled, etc, there's some sort of operation happening which changes data. Just before the output can be fed as input to the next layer, it usually needs to get past a last function, the activation function. Activation functions have the crucial job of deciding how a computed output will be passed onward as input to the next layer. There are several types of activation functions in the Machine Learning field such as sigmoid, tanh and the most common in CNN, Rectified Linear Unit (ReLU). The classical ReLU takes negative values and turns them into zero, deactivating any influence they might have had on the next layer. See Fig.2.2

for a visual representation.

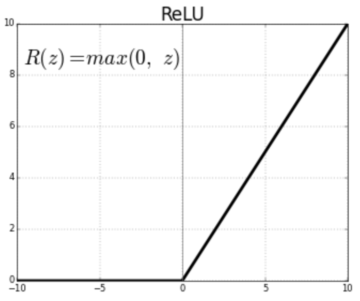

Figure 2.2: Classical ReLU function.

There are other types of ReLU, where instead of zero the negative values are divided by a factor of 10 or 100, and in other types, the value is learned in the training process. The activation function guides the data into more practical values and decreases the influence of certain parts in the feature maps.

The second phase in classification CNN is where the actual classification happens, the home of the fully-connected layer. The objective of a fully connected layer is to take the results of the convolution/pooling process and use them to classify the image into a label. After the first phase, the data has been transformed into a single vector of values. Each value in this list represents a probability that a certain feature belongs to a label. The fully-connected layer also has a learnable aspect to it, as the weights that determine the label need to be determined by the training process.

2.2 Background on training and back-propagation

Even though the matter of study isn't training or designing a CNN architecture, it is important to have some notions of how a CNN actually functions and at its core, training is the reason CNN even exist, it's the *Learning* in Machine Learning. Back-propagation is how the networks adjust values and weights and the motor of the training mechanism [14] [18]. Before we delve into back-propagation we first must understand how the forward propagation, the function that represents the computing of the network, works. A convolution is the main and most important mathematical operation that appears in CNN. Given an input map I, a filter (kernel) K of dimensions $k_1 \times k_2$, C as channels, and biases b its definition is as follows:[12]

$$(I * K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \sum_{1}^{C} K_{m,n,c} . I_{i+m,j+n,c} + b \tag{2.1}$$

The kernel is slid across the input map, and the sum of all the multiplications inside the kernel (there's also the addition of a constant called bias which helps guide the model in the right direction) is then put in position, in the output map. This operation is repeated along the input as many times as it was setup, beforehand. After it, we are left with an output map which contains the information we want, but we want to maximize its usefulness by adjusting the kernel weights. The mean squared error formula is given as:

$$E = \frac{1}{2}\sum_{p}(t_p - y_p)^2 \tag{2.2}$$

Where t_p is our target value and y_p is our output. We want the minimize the error value. Since training is not the focus of this dissertation there's no reason to explore all of the details of this subject. Backpropagation can be expressed as the cross-correlation with a flipped-kernel. It is given by the following expression:

$$\frac{\partial E}{\partial x^l_{i',j'}} = \delta^{l+1}_{i',j'} * rot_{180} w^{l+1}_{m,n} f'(x^l_{i',j'}) \tag{2.3}$$

where:

$$\delta^l_{i,j} = \frac{\partial E}{\partial x^l_{i,j}} \tag{2.4}$$

2.3 Background on the convolution operation

2.3.1 Convolution

The most important operation in Convolutional Neural Networks is in the name, it is the convolution. Convolution is a mathematical operation that is applied on two functions and, simply, results in a third function that expresses how one of the functions shapes the other one. The integral of the product of both of the functions, while one of them is reversed and shifted:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{2.5}$$

$$(f * g)[n] = \sum_{m=-\infty}^{+\infty} f[m]g[n-m] \tag{2.6}$$

Equation 2.2 is the discrete convolution which is more of our interest since we are looking into using it in the discrete domain, obviously. This operation can also be executed within n-dimensions, which is why it is so valuable in the computer vision field. 2D convolutions are used for a wide range of applications, but they are especially good to filter images because they make use of the neighborhood of the pixel, thus taking advantage of its spatial features. They can smooth or sharpen images, work as high-pass or low-pass filters, detect edges and other shapes, etc.

The convolution process in most applications is done by having a kernel that is overlaid into a 2D representation of the image and then each value of the kernel is multiplied by the value it is laid upon in the input image, after it, all of these values are summed and that's the result of one convolution. This operation is repeated several times, depending on the size of the kernel and other parameters such as padding and striding.

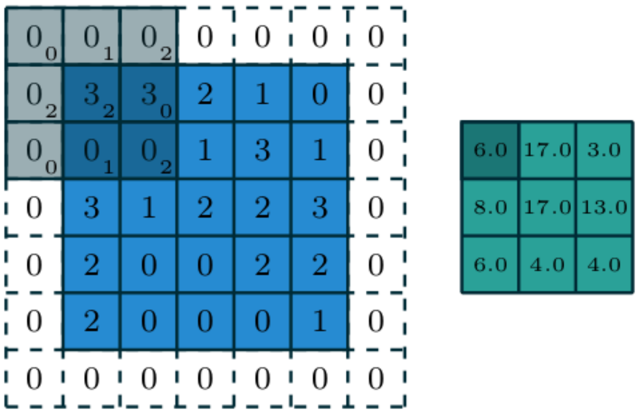

Figure 2.3: 2D-convolution example where I is the input data, K is the kernel and O is the output. [3]

In Fig.2.3 you can see an example of convolution. Kernels can have varying sizes but they're usually odd numbers because the center of the kernel is aligned with one of the pixels in the input image. In this example there is only a single matrix, but in most CNN there are stacked matrices meaning that the kernel is actually applied to all the different matrices in the stack and for one output value there are $K \times K \times Z$ multiplications happening where Z is the number of matrices stacked and K is the one of the sides of the kernel (kernels are generally square), after that we need to perform a sum of all these values which will add more operations. Having a good algorithm is key in having good computational performance.

Apart from the weights (the values in the kernel) which the designer of the architecture doesn't control directly as they come through training of the CNN, there are several parameters need to be set that direct how and where the convolution will be applied. Besides the kernel size, stride and padding have great impact on the output. The designer chooses these values to mainly control its size.

Stride controls where the center of each convolution will be. Stride is the size of the increment that happens whenever the center changes, horizontally and vertically, having a value higher than 1 drastically reduces the size of the output. If we take a look at Fig.2.3, after the first convolution with center in "3", with the value of stride at 1, the kernel would move horizontally one square at a time, after the first "3" it would go to the second "3", then "2", etc. With stride at 2, it would go from "3" to "2" and then to "0". This "jump" only happens in the 2D, every matrix in the stack has a convolution applied to it. Padding are the 0s around the matrix in blue. They virtually augment the size of the input matrix reducing the impact in the output size that a convolution usually has. Padding size regulates how many "layers" of 0s are put around the actual input. Padding comes at a cost, computationally, since we are adding 0s which take up space in memory but also because when we are calculating every output value the 0s are not having an impact on the output itself.

These parameters control the size of the output in the following fashion:

$$O = \frac{I - K + 2 \times P}{S} + 1 \tag{2.7}$$

Where O is the output size, I is the input size, K is the kernel width (or height), P is the padding size and S is the stride value.

2.3.2 Transpose convolution

Besides the regular convolution, optical flow estimation requires the opposite of down-sampling, instead of trying to reduce the size of the feature maps we want to extract the features but then extrapolate them into the size of the original images. There are several up-sampling techniques but the most used in these type of architectures is an operation called transpose convolution. The transpose convolution does the reverse of the regular convolution, while the regular convolution reduces the input, a transpose convolution augments it, it's an operation that helps the data to regain its shape. We can look at how the transpose convolution is used in FlowNet2-S. After the feature maps were reduced we "upconvolve" our inputs meaning they keep getting wider but thinner until they're the size of our original input image. Looking at Figure 2.4 illustrates how this is applied in FlowNet2-S.

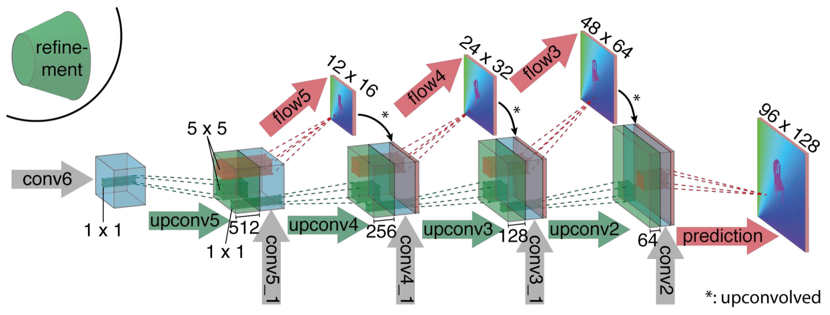

Figure 2.4: Up-sampling features, here the "flow" layers are up-sampling the input. They serve to extract features that will eventually lead to the actual optical flow estimation [7]

The analysis of [6] shows how a convolution can be displayed as a matrix operation and how that helps understanding the transpose convolution operation. If we represent each output of a convolution as a vector, we end up with a matrix like the following if our input is a 4×4 matrix, our kernel is a 3×3 matrix, and our output is a 2×2 matrix:

$$C = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{bmatrix} \tag{2.8}$$

The forward pass of C takes as input a vector with 16 elements, which is the flattened input matrix. The variables are the weights that come from the kernel. It's generally easy to obtain the backward pass of this operation as we can just transpose it. We end up having a flattened input matrix of 4 elements and producing 16 outputs. Transposing C:

$$C^T = \begin{bmatrix} w_{0,0} & 0 & 0 & 0 \\ w_{0,1} & w_{0,0} & 0 & 0 \\ w_{0,2} & w_{0,1} & 0 & 0 \\ 0 & w_{0,2} & 0 & 0 \\ w_{1,0} & 0 & w_{0,0} & 0 \\ w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\ w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\ 0 & w_{1,2} & 0 & w_{0,2} \\ w_{2,0} & 0 & w_{1,0} & 0 \\ w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\ w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\ 0 & w_{2,2} & 0 & w_{1,2} \\ 0 & 0 & w_{2,0} & 0 \\ 0 & 0 & w_{2,1} & w_{2,0} \\ 0 & 0 & w_{2,2} & w_{2,1} \\ 0 & 0 & 0 & w_{2,2} \end{bmatrix} \tag{2.9}$$

As we can see, there are the same amount of mathematical operations but each output vector is much smaller which actually is bad for our implementation since this way we need to spend more resources and time allocating memory and transferring data between buffers. Our function to calculate the output size comes from equation 2.3 but instead we reverse the input and output variables. The output size of the transpose convolution becomes:

$$O = S(I-1) + K - 2 \times P \tag{2.10}$$

There are several ways to implement this operation. The most straight-forward and direct one is by applying a direct convolution with padding with the weight order reversed. The direct implementation of this method is very inefficient due to the amount of 0s that are needed which don't have any impact on the final result of each convolution. But it really helps visualizing the problem, what the result for each iteration should be and how an algorithm can come from it. As we can see in Fig.2.5, we have a 2×2 input kernel in blue, with padding value of 2. As we can see, it creates an output of a 4×4 matrix.

Stride and padding behave opposite of regular convolution. Because it's the reverse operation stride doesn't reduce the size of output but increase it. This can be done and more easily visualized by adding empty space between input values, proportionally to the stride value, and then

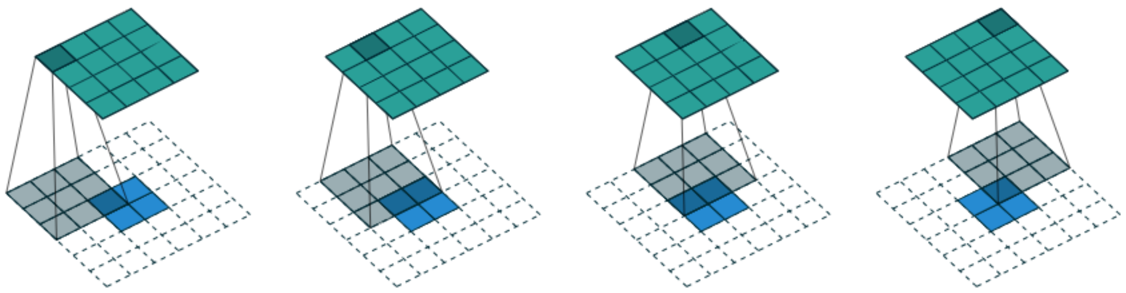

Figure 2.5: Up-sampling features, here the ”flow” layers are up-sampling the input. They serve to extract features that will eventually lead to the actual optical flow estimation[6]

using unitary stride like a normal convolution. Padding is also the opposite of a regular convolution operation, instead of adding a wall of 0s, we are erasing them. Reducing the size of input proportionally to the value of padding.

2.4 Background on fixed-point numbers

We are going to be using a framework that uses a non-conventional way of computing and organizing numbers. Instead of using a common **float** it uses **short** as its main computational measure. This is done because it can increase the performance as it’s much less expensive memory wise. Since the numbers are smaller computation is much much faster but comes at the cost of less precision because the range of possible numbers is smaller (32bits vs 16bits). The use of fixed point numbers is a deliberate strategy that needs careful planning from the designer because it can be very detrimental to the accuracy of calculations. It’s a trade-off between accuracy and speed which can be instrumental in the practicality of certain real time applications.

A **char** only uses 8 bits which can only be used to represent 256 variations. This means that the range of values goes from 0 to 255 for unsigned numbers, and -127 to 128 in signed data. Imagine we have a system that uses **unsigned char** as its base and the problem it’s used for only deals with values that go from 0 to 50. If we use regular notation, only the integer data is used, which means around 20% of the values available are being used but the rest are wasted. All of the decimal values are not regarded which takes a toll on the precision of the measurements. To change this we can use a different notation where more bits are used which in turn will make the system more accurate. We can do this by creating a virtual decimal point that determines which part of the information is integer and which is fractional. The designer decides where this virtual point is put and determines how accurate the calculations will be. Depending on the application and the platforms constraints it may be beneficial to use fixed-point data.

It is important to understand that this notation does not affect in any way how the machine looks at the numbers, it does its calculations as it was using regular notation but the programmer does an adjustment after all the calculations that instead take the virtual decimal separator into account. Figure 2.6 shows how for the computer the actual (binary) values for each number don’t change.

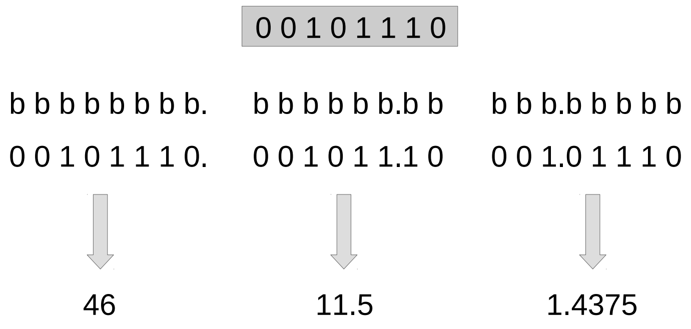

Figure 2.6: An example of how fixed point can dramatically change the perceived value of numbers. From the same set of 8 bits we derive a different value depending of where our virtual decimal point is placed, all the while the computer reads it and computes it as its natural value of 46.

The bits on the right side of the decimal number still use the common 2^m formula, but are of course negative values for m, meaning they are the inverse of the powers of two of the numbers on the left (e.g. $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, and so on). For addition and subtraction the process is quite easy and does not need too much preparation because the result of these operations will most not need carry over bits. When we multiply the number of bits needed increases greatly and can lead to loss of information or just wrong calculation if not taken care of.

There are several rules regarding the mathematical operations just stated before, they can be very useful to understand how data manipulation needs to be handled which were considered from [20].

2.4.1 Fixed-point arithmetic

2.4.1.1 Fixed-point numbers

The value of a particular N-bit binary number x in fixed point form, where a are the integer bits and b are the decimal bits, is given by the expression:

$$x = (1/2^b) \sum_{n=0}^{N-1} 2^n x_n \tag{2.11}$$

Where x_n represents the n bit of x. This is only true for unsigned data, for signed data we use a different approach. Two's complement is how signed data is handled. The most significant bit is used for sign and the rest of the bits represent the data value. For two's complement we can quickly get the inverse of a number number by simply inverting all the bits and adding one to the result. To invert all the bits arithmetically we can subtract the original value x from $2^N - 1$, but since for two's complement we add one after inversion, we get:

$$\tilde{x} = 2^N - 1 - x + 1 \tag{2.12}$$

$$\tilde{x} = 2^N - x \tag{2.13}$$

Where $\tilde{x}$ is the two's complement of x. If we want a specific number in this form, we can arithmetically define it with:

$$x = (1/2^b)\left[-2^{N-1}x_{N-1} + \sum_{n=0}^{N-2} 2^n x_n\right] \tag{2.14}$$

As it is noted by the $N-2$, two's complement uses one less bit of magnitude in relation to the unsigned value. When using signed values our range is transposed to the right by one bit, for example in 8 bits, instead of 0 to 255, our range becomes -128 to 127. The notation of data is important, but we also need to understand how fixed point data is affected by arithmetic calculations and how should we prepare it before or handle it after calculation. The following arithmetic rules will only be valid for signed data, as it is what is used in this dissertation.

2.4.1.2 Addition

In order to have correct addition, both of the numbers need to be in the same scale. The value for a and b needs to be the same if we want to have accurate results. We can do this by simply shifting the bits of one of the numbers to match the other before we add them together.

In the end, the number of bits needed for accurate representation of the result is $a+b+1$. Essentially the number of bits needed are the same as the input, with the addition of another one in case of overflow. The maximum overflow possible is the magnitude of one extra bit. Subtraction won't need any extra bits, and the result can be presented in $a+b$.

In a more practical approach, if we use 8 bits as an example, the results of calculations will need to be held in at least 9 bits, after that there's rounding of the result, which will have an absolute error the size of the least significant bit. The rounding is done by shifting the results to the right.

2.4.1.3 Multiplication

For multiplication, there's more to be taken into consideration. If we take x and y as two N-bit numbers, having (a_1, b_1) and (a_2, b_2) bits, respectively. For accurate representation of the results of a multiplication of both of these numbers, we need at least $(a_1 + a_2 + 1 + b_1 + b_2)$ bits. If both of the numbers are of the same type, we need double the amount of bits plus one.

Practically we want a buffer that can hold these values and then shift them maintaining the order of magnitude that we had before. Since we are dealing with convolutions it is very important to have control over overflows because these will affect the next results deeply and can corrupt the final results easily.

2.5 Background on Intel® FPGA SDK for OpenCL

Intel® FPGA SDK for OpenCL is a programming environment which enables developers to target FPGA as an heterogeneous platform. It creates an abstraction of the board supported by a compiler which helps the developer to concretely work on a system like an FPGA without having to worry about a lot of the hardware details and problems attached to a system like it. The developer still has to understand the system but the compiler alleviates part of the programming burden. The compiler decodes the information in C and translates it into various logic blocks, Digital Signal Processing (DSP) blocks, memory registers, etc, which will then be synthesized in the FPGA. It also allows the direct utilization of Register Transfer Level (RTL) modules in code, with some restrictions.

The developer can instruct the compiler with some pragmas. These pragmas essentially tell the compiler specific characteristics about the code which can lead to improvements and optimization. The most common is probably pragma unroll which tells the compiler that a certain loop can be parallelized and there aren't any data dependencies.

There are two parts in a system like this, the host and the kernel. The host is controlled by a regular computer, it calls the functions to be issued in the FPGA and basically manages the whole process. The kernels are the functions called by the host to be implemented in the actual board. The SDK optimizes the code to the hardware being used.

2.5.1 NDRange kernel vs single work-item

There are different types of kernels, and how they're initialized creates different environment and behaviour. Single work-item kernels allow using computation loops in the kernel, but also to generate custom pipelines. These kernels allow the use of the pragmas. They're easier to implement as they don't use work-groups, and the global size is set to 1. They allow the compiler to parallelize by itself, unrolling loops and optimizing pipelines as well. They also allow the use of 'channels' to move data between the single work-item kernels, relieving some memory bandwidth usage. These channels allow the transfer of data between kernels without having to wait for reception (in some cases), essentially creating an uninterrupted work-flow between kernels.

OpenCL produces a report after compilation which states how it unrolled certain loops, or how it wasn't capable of unrolling these loops. It helps the programmer understand data dependencies and helps optimization in case there are changes in the algorithm possible. In most cases, single-work item kernels are used, unless the problem to be solved has the features of a classic GPU partition problem where it's easy parallelizable, which is what the NDRange kernels are for.

NDRange kernels are designed for Single Instruction, Multiple Data (SIMD) operations like the GPU. They need partitioning of data into work-items and work-groups. This is also the type of kernel that's more common in environments which allow parallelism in GPU platforms. It is usually for kernels that don't have memory dependencies between themselves as NDRange kernels need to access global memory to share data between work-groups. NDRange kernels are almost only used in case there are a lot of efficiency problems when using single-work item kernels or the problem is easily partitioned.

2.5.2 Pipelining and unrolling loops

The OpenCL compiler tries to optimize the code written by the developer by optimizing loops, most of the time. Logic circuits work by using clock cycles as their input, the system only computes when the clock ticks. It is easy to imagine that the best scenario possible is whenever there is a tick of the clock, new information is processed, thus not wasting any clock cycles. But this is not always possible.

Logic blocks only process information if there are valid inputs ready. If the information has not reached the input of a certain block, nothing happens. Taking as an example [8], if the circuit to be designed has three multiplications followed by an addition: $(a_i \times b_i \times c_i + d_i)$.

The most common way to solve this problem is to create a circuit like Figure 2.7.

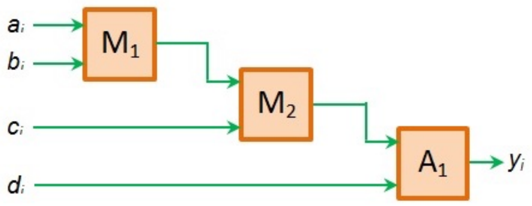

Figure 2.7: Simple circuit which illustrates $(a_i \times b_i \times c_i + d_i)$[8]

The problem is that information is constantly flowing. On the first clock cycle only M1 has valid inputs (a_1, b_1), which has output $(a_1 \times b_2)$. The second clock cycle comes and M2 needs to activate to produce $(a_1 \times b_1 \times c_1)$, the problem is that M1 cannot be activated because the output would change to $(a_2 \times b_2)$. When it is time for the third clock, A1 has a valid input of $(a_1 \times b_1 \times c_1, d_1)$ and produces y_1. This means that y_i can only be produced every third clock tick, M1 stops doing work after the first clock tick, M2 after the second and A1 only processes information on the third. It is not very efficient. The solution is to pipeline the information and have registers that store the values needed.

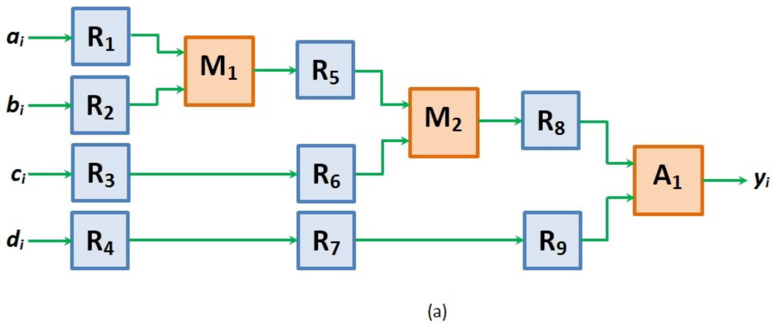

Figure 2.8: Simple circuit which illustrates $(a_i \times b_i \times c_i + d_i)$[8]

In Figure 2.8 there is a system that would be a pipelined version of the circuit in Figure 2.7. After the first clock tick M1 stores the value of $(a_1 \times b_1)$ in R5, R3 stores the value of c_1 and R4 stores d_1. On the second clock tick M1 can be activated and multiply the next numbers (a_2, b_2). R5 has $a_1 \times b_1$ and R6 has c_1 which means M2 can be activated and produce $(a_1 \times b_1 \times c_1)$. And so on, this makes the system produce y_i every clock cycle instead of every other third. This speeds up the system but it ois obviously memory expensive as there is an addition of a register on every arithmetic operation, at least.

Unrolling loops essentially means that the compiler takes part of the code and does more than one operation in every loop. For example, if we take this excerpt of code:

```
for(i=0; i<100; i++){
      a[i]=a[i]+s;
}
```

a[] is an array and s is a constant value. In a normal compiler, this loops would need to happen 100 times, one for each value of i. Loop unrolling tries to cut this by doing more than one operation per iteration. The compiler needs to know that a[i] doesn't have any memory dependencies on itself and can use a[i] liberally. The compiler can instead do something like this:

```
for(i=0; i<100; i=i+2){
      a[i]=a[i]+s;
      a[i+1]=a[i+1]+s;
}
```

In every loop called, there are 2 additions happening, and this can be done however many times are possible to try and optimize the number of loops calls. This just minimizes the amount of times there's a call to check if the loop has reached its limit.

2.5.3 Channels

One of the big problems with systems that use these heterogeneous types of devices is the communication between them. Imagining a typical scenario where there is a host, several kernels and a global memory (usually an external device to the kernels) which is accessed, usually, by a PCIE bus. A memory cycle can look like the following:

Host->Global Memory->Kernel#1->Global Memory->Kernel#2->Global Memory->Host

Communicating directly with the global memory is usually very costly and can become one of the bottlenecks using FPGAs. To accelerate this process there are communication channels between kernels which allow for direct communication without having to go through the global memory. This not only accelerates the process of accessing through the global memory but it also enables ongoing communication. Using the standard procedure, Kernel#1 needs to be called, after it ends, Kernel#2 is called and reads from global memory. Channels allow for both kernels to work at the same time, Kernel#1 keeps sending information to Kernel#2, which receives it and processes it at the same time as Kernel#1 processes its own data. Channels not only allow for faster communication but also parallelization of work.

2.6 State of the art

In the past few years, the number of publications related to optical flow estimation using CNN has been increasing substantially. The use of FPGA as an accelerator has also been a hot topic. In this chapter, some of the most representative optical flow paradigms currently available will be presented and analyzed, as well as FPGA implementation of some CNN architectures.

2.6.1 Optical flow estimation paradigms

Optical flow refers to the correspondence points between pairs of images that belong to the same scene. There was a lot of research done on the topic, work done by Horn and Schunck [9] was especially important. It introduced the energy minimizing approach which would revolutionize future work as all the top-performing methods would adopt this strategy. When CNN were starting to appear as a resourceful tool in Computer Vision, researchers also worked on a solution that involved this type of framework. FlowNetS and FlowNetC were the first models to be developed by Fischer and Dosovitskiy [7] and earned themselves a lot of praise because of its extraordinary performance in real-time. Because of this, there was a paradigm shift, it was proved that a scalable and end-to-end trainable CNN could produce results faster and perform at state-of-the-art level.

The jump came with FlowNet2, created by Ilg *et al* [11] which stacked several types of FlowNetS and FlowNetC networks, adopted a new operation called Warping, and focused on the training datasets. One of the problems with FlowNetS and FlowNetC was that it couldn't rigorously estimate optical flow in small displacements, which was something traditional methods had no trouble with, and the results were also a bit blurry as well. FlowNet2 solved this problem by stacking differently trained CNN, some appropriate for larger displacements and one that was trained to accurately estimate small ones, this resulted in an architecture that had very good results and competed directly with state-of-the-art methods, while being really fast. Figure 2.9 illustrates the architecture of FlowNet2.

One problematic factor is that the whole architecture is very large (650MB) and has over 160M parameters which makes it not suited for any type of mobile or embedded devices. Solving this problem came with structuring data differently. LiteFlowNet [10] took inspiration in [17] which used spatial pyramids instead of the brightness constancy optimization that is commonly used to estimate optical flow. Spatial pyramids are constructed by taking the original two images as input and have convolution filters that represent them at the lth layer, where L represents the number of times the image was downsampled. SPyNet had proven that this method helps with the faulty estimation of small displacements that happened in FlowNet, and the likes. It also reduced the size of weight parameters needed.

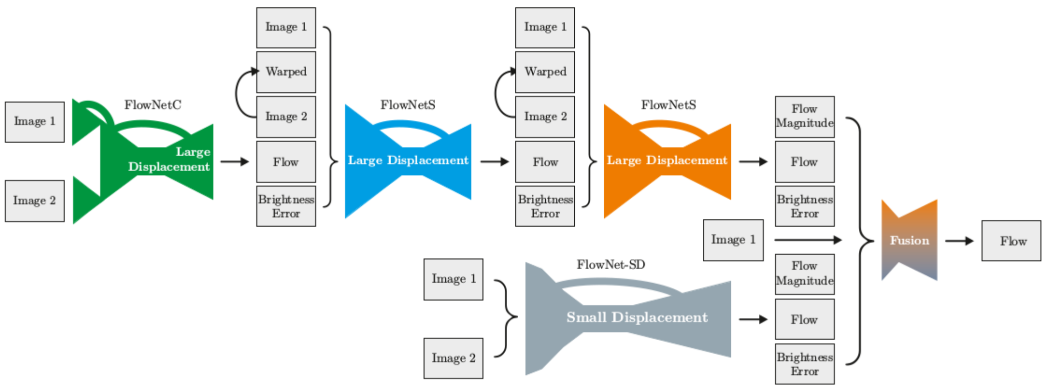

Figure 2.9: The whole architecture of FlowNet2, it stacks several CNN and because of this it is very large, as we can visually understand. Each of the CNN are meant to accurately predict optical flow in different conditions and in the end, the results are fused and produce a map of the whole scene [11].

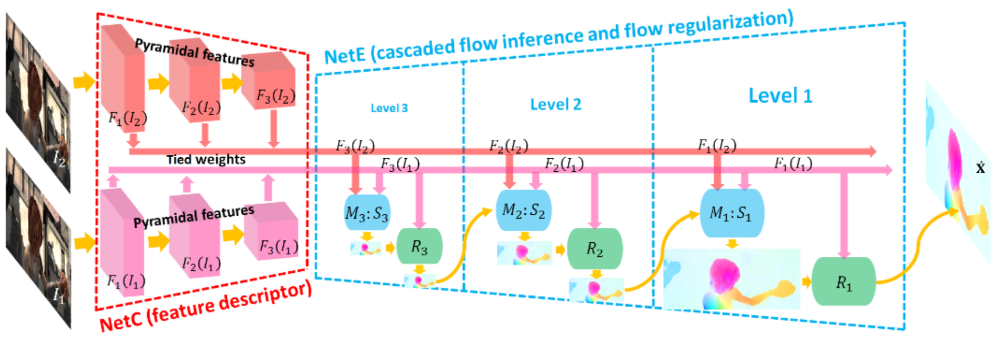

Figure 2.10: Architecture of one of the levels in LiteFlowNet [10].

LiteFlowNet has two compact sub-networks, one extracts pyramidal features and the other estimates optical flow. Part of its architecture is shown in Figure 2.10. By using these different techniques LiteFlowNet has produced incredible results. The team that worked on it went further and by analyzing their own architecture produced a new version with better results, LiteFlowNet2 [10]. It is arguably the best performing CNN in the optical flow estimation field. It performs at the same level or even better than FlowNet2 at a fraction of the size.

2.6.2 Transposed Convolution in FPGA

Transposed convolution is a very important operation in different CNN architectures. It is a fundamental upsampling layer that is usually used in optical flow estimation, but also in image segmentation CNN. It is an important part of the pipeline and can be quite an expensive operation. Because of this, there have been several ways that investigators have tried to optimize them. As stated before, transposed convolution can be implemented as a regular convolution with padding around the input matrix and around data in case of stride larger than 1. This causes massive memory usage while being quite wasteful since the multiplications with zero are absorbed. The most efficient way of doing this operation seems to rely on a different perspective, multiplying each value in the input matrix by the kernel, add some overlapping values and crop the final matrix, as in [15], Figure 2.11 aids visually.

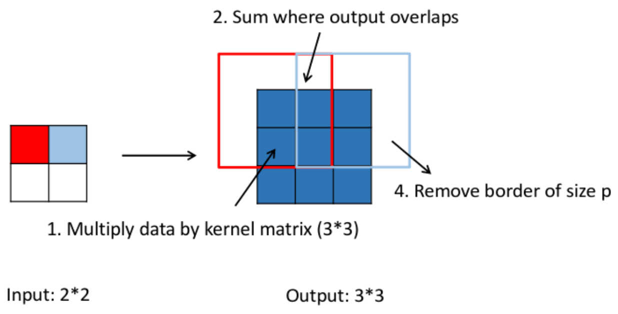
2. Sum where output overlaps
4. Remove border of size p
1. Multiply data by kernel matrix (3*3)
Input: 2*2
Output: 3*3

Figure 2.11: Representation of what the operation looks like. Depending on the parameters of the layer we may have no overlap of data to be summed (if stride is equal to kernel width/height), and if padding is equal to zero, there's no cropping to be made. In this specific case, stride is equal to two, that's why there's overlap only on that small part that is intercepted by the red and cyan box, and padding is equal to one, that's why there's cropping of one square around the border [15].

2.6.2.1 Algorithm Comparison

If we want to compare the performance of both algorithms we need to obviously understand how both of them work.

Parameter	**Description**
H_I	Height of the input feature map
H_O	Height of output feature map
N_I	Number of channels in the input feature map
N_O	Number of channels in the output feature map
k	Height of the kernel
s	Stride
p	Padding

Table 2.1: Summary of parameter nomenclature. Since we will only be using square matrices as input there's no need for both height and width representation [15]

Using the Table 2.1 as reference for nomenclature. There are several ways to compare efficiency, but number of operations per algorithm is probably the one with the biggest impact. Looking into the algorithm that treats it as a direct convolution, the size of the input image is $H_I * H_I * N_I$ (note that the image is square and both sides are equal, that's why we have $H_I * H_I$), the kernel weights can be thought of as a coefficient matrix with size $k*k*N_I*N_O$ (in equation 2.8 we have an example of what this matrix can look like). The output of this layer will be a matrix with size $H_O * H_O * N_O$. The computation can be achieved by just having two nested loops. We will call this algorithm "DC" (Direct Convolution), we are comparing transposed convolution methods but this essentially is a direct convolution.

```
//DIRECT CONVOLUTION ALGORITHM
for o=1 to NO
    for i=1 to NI
        O[o] += transposed_conv(I[i], K[o,i])
    end
end
```

Figure 2.12: Direct convolution algorithm pseudo-code [15]

”transposed_conv” multiplies and calculates the sum of all the parts involved in each iteration. This algorithm is obviously computationally intensive, and even more than the regular convolution operation if we take into consideration same input size. Essentially it expands the input size to then decrease it with the convolution operation. This algorithm will be called DC (direct convolution) onwards. We can calculate the number of operations needed and then compare them to the algorithm discussed in [15].

$$OP_{DC} = 2 * k * k * H_O * H_O \tag{2.15}$$

Important to understand that H_O will be larger than H_I which is the opposite of what happens in a regular convolution. For the algorithm proposed in [15], which we will call SP (summed parts), the number of operations is given as:

$$OP_{SP} = k^2 H_I^2 + k(k-s)(H_I - 1) + k(k-s)(W_I - 1) + (k^2 - s^2)(W_I - 2)(H_I - 2) \tag{2.16}$$

Although apparently more complex than Equation 2.15, the paper states that it has several advantages and that it has its biggest efficiency gains when a large kernel size is used. Table 2.2 exposes data from the paper as well, but we’ll only report the values for the U-Net CNN they used for testing:

Layer	Parameters	OP_{DC}			OP_{SP}			**Efficiency Speedup**
	$(k, s, H_I, N_I, N_O, H_O)$	Mult	Add	Total	Mult	Add	Total	
deconv1	(2,2,28,1024,512,56)	6.58G	6.58G	13.15G	1.64G	1.64G	3.28G	4x
deconv2	(2,2,52,512,256,104)	5.67G	5.67G	11.34G	1.42G	1.42G	2.83G	4x
deconv3	(2,2,100,256,128,200)	5.24G	5.24G	10.5G	1.31G	1.30G	2.60G	4.01x
deconv4	(2,2,196,128,64,392)	5.04G	5.04G	10.07G	1.26G	1.24G	2.51G	4.02x

Table 2.2: Example of how the SP algorithm can reduce the number of operations significantly. [15]

Besides the significant reduction in operations, it is obvious that having to add so much padding will have a negative effect on the efficiency of the operation. The resources will be underutilized as all the multiplications with zero will be wasted. The article does not compare directly with the regular convolution approach of adding zeros. But it is easily understandable that if an implementation of

	FPGA	**GPU**	**CPU**
Platform	ZC706	Nvidia Titan X (Pascal)	Intel Core i7-950
Precision	32-bit fixed-point	32-bit floating-point	32-bit floating-point
Deconv time	125ms	37.11ms	517ms
Power	9.60W	168W	106W
Energy consumption	1.20J	6.24J	54.80J

Table 2.3: Results obtained in [15] regarding the transposed convolution layer (called Deconv in this table). Each column specifies the type of platform. The first row specifies the platform used. The second row represents the precision at which the data was handled. The third row describes the kernel time in the transposed convolution layer. The last row describes the power consumption.

this is done correctly and efficiently it will save resources. This article states that its implementation achieves a speedup of 4.14x in processing speed and is much more energy efficient in relation to a CPU implementation. It is slower than the GPU by quite a large margin, but it is important to note that the ZC706 only has 900 DSP units which can be limiting in regards to the amount of operations done in parallel. Table 2.3 shows us other results achieved

3

Estimating optical flow with PipeCNN

In this chapter, details of each element in the pipeline are described. Namely the optical flow estimation CNN, all the software used and terminology. The chosen CNN architecture was FlowNet-S [7], for its simplicity. The pipeline of the setup is shown in Figure 3.1 MATLAB [Mathworks 2015b] was used for pre-processing of the images and, using the 'Deep Learning Toolbox', to read the weight values from the Caffemodel. A tool called 'flow2image' was used to transform the flow maps to pictures. Each colour represents a different direction. This dissertation implemented new kernels and adapted ones that were originally in PipeCNN.

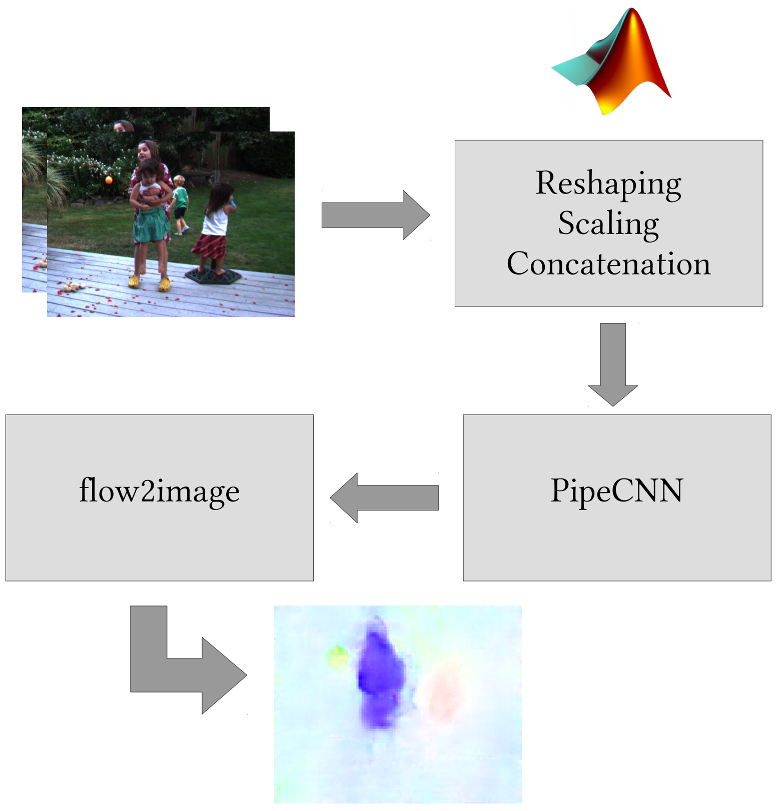

Figure 3.1: Graphical representation of the pipeline.

Before using a pair of images in PipeCNN, normalization and concatenation is needed. For the sake of simplicity, all of this pre-processing was done in MATLAB instead of using the layers of the CNN. In FlowNet-S the input is two concatenated images, after they've been processed and scaled. This is different than most state-of-the-art technology. Usually there are two parallel processes happening to the two different images, and at some point they're joined again. The focus was on having a working system and so, all of these additions weren't contemplated, LiteFlowNet and FlowNet2 cannot run on this dissertation's system mainly because there are some important layers that weren't developed. Warp and correlation layers are the most important which are missing, and would need implementation in the future. The system does however write the flow map, this operation occurs on the host machine, after getting the results from the kernel. Some file writing operations are executed, following the '.flo' standard [16].

The FPGA used in this dissertation is Gidel's ProceV which is depicted in Figure 3.2. Some notes: the only Board Support Package available for the ProceV boards is the Intel® FPGA SDK for OpenCL 14.1, while PipeCNN was designed to use at least version 16.0. This means some optimization features such as being able to compile and use RTL components directly in the code weren't available. The overall use of DSP blocks in these boards becomes very low because of this, and there's no direct way to correct this or take more advantage of the board's features.

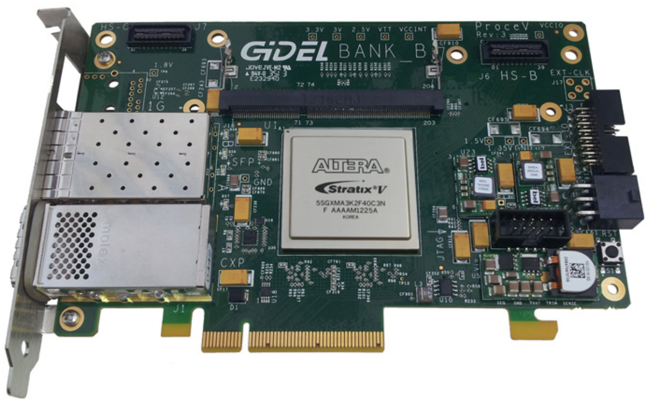

Figure 3.2: Picture of Gidel's ProceV.

Description of the board:

- Statrix V family
- Four levels of memory:
  - DDR3 (ECC): Up to 16GB SODIMMs at up to 19.2 GB/s sustain throughput
  - SRAM: Up to 2 × 144Mb
  - FPGA's M20K (20K-bit) SRAM blocks: Up to 52 Mb at 8,000 GB/s (300 MHz)
  - FPGA's MLAB (640-bit) SRAM blocks: Up to 8 Mb
- PCIe x8 (Gen 2 and Gen3)

- Typical system frequencies: 150-450MHz

Before compilation of the kernels for hardware, extensive debugging was done using the Intel® FPGA SDK for OpenCL emulator. This emulator doesn't emulate the FPGA, it only runs the code in the CPU as it would normally run C code. This means no real parallelism is achieved, but apart from some known issues it is enough to use as a debugger. Hardware compilation for this board takes about 5 to 6 hours. The emulator becomes the only option to correct the code.

3.1 Caffe and MATLAB

Another part of the whole process is the utilization of a common CNN framework called Caffe [13]. Caffe was created in Berkeley UC, by Yangqing Jia and continues to be developed by Berkeley AI Research (BAIR) and by community contributors.It supports popular interfaces such as Python and MATLAB. Caffe developed a standardized model in which CNN layers are easily deployed, and trained. Since the focus of this dissertation is only implementing existing CNN architectures, the use of matcaffe (Caffe's MATLAB interface) allowed quick reading of the caffe models. After extracting the weight and bias values, these were transformed into fixed point form using MATLAB's Fixed Point Toolbox and then converted into '.dat' files which are then used by PipeCNN directly. For the layout of the CNN and its parameters, Netscope [1] was useful for visualizing how layers were connected but also to check the output size of each layer, since Caffe's CNN layout files don't provide this type of information.

Some code was developed in MATLAB to prepare the images. Apart from the concatenation of both images into one; scaling and transposition of the matrix was also a part of this pre-processing. PipeCNN functions with BGR channels instead of the conventional RGB. Some pseudo-code is given in Figure 3.3 as an example.

```
function [ crop[2] preprocessed_image ] = preprocessingFlow_scaled( ...
    filename[i], CROPPED_DIM )
    im[i]=read_images(filename[i]);

    %values taken from FlowNet s.prototxt
    input_scales[i] = scales;
    channel_constants[i][3] = constant_values;
    %each channel has unique constant
    im[i] = im[i].*input_scales[i] + channel_constants[i][c];
    % permute channels from RGB to BGR and flip width and height
    im[i] = permute(im[i](:, :, [3, 2, 1]), [2,1,3]);
    % resize im[i]
    crop[i] = imresize(im[i], [CROPPED_DIM CROPPED_DIM], 'nearest');
    %concatenate into final matrix
    preprocessed_image(:,:,:)=cat(3,crop[1],crop[2]);
end
```

Figure 3.3: Pseudo-code representing the pre-processing that is done to the pair of images.

Having the images pre-processed was chosen over using layers inside the CNN because it alleviates the processing work needed from the FPGA and all the resources attached to it, such as memory usage. Another reason was the necessity of having to allow more than one image as input which the framework doesn't allow as of now.

Using matcaffe allowed debugging layer by layer and was very important in the development of the kernels as well as serving as ground truth when comparing them to the FPGA results.

3.2 PipeCNN pipeline

PipeCNN uses Intel® FPGA SDK for OpenCL technology. Because the FPGA is a hard platform to program and manage, this tool allows the designer to abstract from a lot of the problems encountered when doing work at this level. It substantially cuts the development time needed. There's a reverse of the medal, of course, because the compiler does a lot of the optimization, working directly on the hardware/software would probably have a more efficient result, but in much more time and probably with a bigger team of developers.

3.2.1 Host

In PipeCNN, the host computer controls the CNN architecture. Through a file called 'layer_config.h' the user needs to configure the layers of the architecture. All of the input, output and kernel sizes. The number of bias values, use of activation layer or pooling layer, use of concatenation layer and its parameters, it's all introduced by the user. Setting the image and weight files paths, as well as the number of existing layers in the CNN. This allows for flexibility in the CNN architecture used.

The host reads all of this information and calls the kernel functions accordingly. Results from each layer are held in several global memory buffers. The host sends the kernel which memory pointer to access. There aren't any direct exchanges of large amounts of data between the beginning of the process and the last layer. The results are finally transmitted and read by the host to be then written into a '.flo' file [16].

3.2.2 Kernel

The kernel is the reason there's work being done with FPGAs. It receives information from the host, and executes the functions needed. OpenCL transforms code written in C into logic circuits with DSP blocks and other components which can accelerate performance. The compiler also tries to pipeline all the circuits as best as it can, creating a higher throughput system.

The most important functions in PipeCNN are 'memRead', 'coreConv' and 'memWrite'. 'concat' is not part of the core layers, but it was added as a new module.

- 'memRead' reads data from the image and weight buffers and sends it to 'coreConv'.
- 'coreConv' gets the data and processes the data by applying a convolution. Sends the results directly to 'memWrite'.

- ’memWrite’ receives the data processed and writes it to a buffer. ’memWrite’ is the only function that is a NDRange kernel.
- ’concat’ reads data from the global buffer. Concatenates it appropriately.

3.2.2.1 memRead

’memRead’ is a single work-item kernel where the input is read from the global memory. It divides the input into several windows and sends it to ’coreConv’ via channel communication. There are three channels open between both kernels ’data_ch’, ’weight_ch’ and ’bias_ch’, which send input data, weight data and bias data respectively. These channels send a vector of input data, vector of weight data and a scalar for bias data.

Each window of data sent pairs the input value with the respective weight value to be multiplied. All the padding in the input data is done in this kernel as well, from regular convolution padding to the padding necessary for the transposed convolution when using stride > 1.

The data is sent according to the number of pipelines in ’coreConv’. There’s a variable that controls the number of different pipelines that incur in the convolution processing.

3.2.2.2 coreConv

’coreConv’ is a single work-item kernel that instantiates a user defined number of parallel pipelines that process all of the information coming from ’memRead’. It reads data from the channels, appropriately transfers it to buffers and processes the data. Data is processed in the ’mac’ function, where it’s multiplied and added, all of the values are then accumulated to then be adjusted because of the usage of fixed-point.

After it, data is sent to ’memWrite’ to be written into global data, through ’conv_ch’, the convolution channel.

3.2.2.3 memWrite

’memWrite’ is a NDRange kernel which means it follows an approach common in GPU platforms. Each work-group is given an ID which then calculate a coordinate, that coordinate indicates the output position of the results, and it then stores these values which are received from ’coreConv’. No pooling is used in FlowNet-S, but if a CNN needs a maxpooling operation, the data is sent through a channel to the function responsible for the pooling. All of these operations are occurring at the same time thus making the pipeline fairly efficient.

3.2.2.4 concat

’concat’ is a single work-item kernel, and it serves the purposed of concatenating results in the z axis. It takes pointers to global memory as inputs and concatenates them accordingly. At the time, 3 results can be concatenated together. The operation is done vector by vector, cycling through each matrix of results, which makes it quite a costly operation in regards to computation time.

This kernel is called after 'memWrite'. It could be optimized by the use of channels but for ease of development a more straight-forward implementation was the choice.

3.2.3 FlowNet-S architecture

FlowNet-S was the choice because of its simplicity in the different layers needed. It has 23 layers, all of them with either a convolution or a transposed convolution operation. Some layers only use a reduced size in the z direction, the ones in which the optical flow is being mapped. Because of this, there was a necessity to pad the weight buffer with zeros accordingly. This way, the result of the convolution will be zero and it will not have any impact in the final result. No other option was found to fix this other than padding, it is due to the nature of how the vectors in PipeCNN are stored. Table 3.1 provides details of the architecture's layers.

Name	Operation	Weight size	Stride&Padding	Input&Output Buffer
conv1	conv	(7, 7, 6, 24)	(2,3)	0&1
conv2	conv	(5, 5, 24, 48)	(2,2)	1&10
conv3	conv	(5, 5, 48, 96)	(2,2)	10&1
conv3_1	conv	(3, 3, 96, 96)	(1,1)	1&11
conv4	conv	(3, 3, 96, 192)	(2,1)	11&1
conv4_1	conv	(3, 3, 192, 192)	(1,1)	1&12
conv5	conv	(3, 3, 192, 192)	(2,1)	12&1
conv5_1	conv	(3, 3, 192, 192)	(1,1)	1&13
conv6	conv	(3, 3, 192, 384)	(2,1)	13&1
conv6_1	conv	(3, 3, 384, 384)	(1,1)	1&13
predict_conv6	conv	(3, 3, 384, 2)	(1,1)	0&7
deconv5	transp. conv	(4, 4, 384, 192)	(2,1)	0&6
upsample_flow6to5	transp. conv	(4, 4, 2, 2)	(2,1)	7&1
concat2	concat			(13,6,1)&0
predict_conv5	conv	(3, 3, 386,2)	(1,1)	0&7
deconv4	transp. conv	(4, 4, 386, 96)	(2,1)	0&6
upsample_flow5to4	transp. conv	(4, 4, 2, 2)	(2,1)	7&1
concat3	concat			(12,6,1)&0
predict_conv4	conv	(3, 3, 290, 2)	(1,1)	0&7
deconv3	transp. conv	(4, 4, 290, 96)	(2,1)	0&6
upsample_flow4to3	transp. conv	(4, 4, 2, 2)	(2,1)	7&1
concat4	concat			(11,6,1)&0
predict_conv3	conv	(3, 3, 146, 2)	(1,1)	0&7
deconv2	transp. conv	(4, 4, 146, 24)	(2,1)	0&6
upsample_flow3to2	transp. conv	(4, 4, 2, 2)	(2,1)	7&1
concat5	concat			(10,6,1)&0
predict_conv2	conv	(3, 3, 74, 2)	(1,1)	0&1

Table 3.1: Description of every layer in FlowNet-S, as well as the buffers necessary for it to work in PipeCNN [7].

This architecture needs some auxiliary buffers to be created overall. Apart from the regular input and output buffers, 6 more were added. Three of the results in the first half of the CNN need to be stored so they can later be used in a concatenation layer. Results from conv2, conv3_1, conv4_1 and conv5_1. But because of the nature of concatenation layer, there was the use of 2 more buffers to store the values and not have any memory corruption.

In case there was a different architecture being used the developer would need to create these by hand, on the host file.

3.2.4 Scalability and parallel operations

PipeCNN organizes data using arrays of VEC_SIZE size. For each (x, y) position in the input there's a vector of adjacent feature maps. The whole set of vectors together form a matrix which is decomposed into a 1D array. The number of pipelines instantiated in 'coreConv' is controlled by the variable LANE_NUM. Each pipeline processes its part of the output. Essentially, the total number of outputs are divided by the number of lanes. Figure 3.4 illustrates the concept.

Because each pipeline processes the information in parallel, this can speed-up the performance of PipeCNN dramatically. It also means that this framework is scalable, optimizable and tweakable to the needs of the user's board. With a bigger board more lanes and bigger vectors are allowed and faster the process will be.

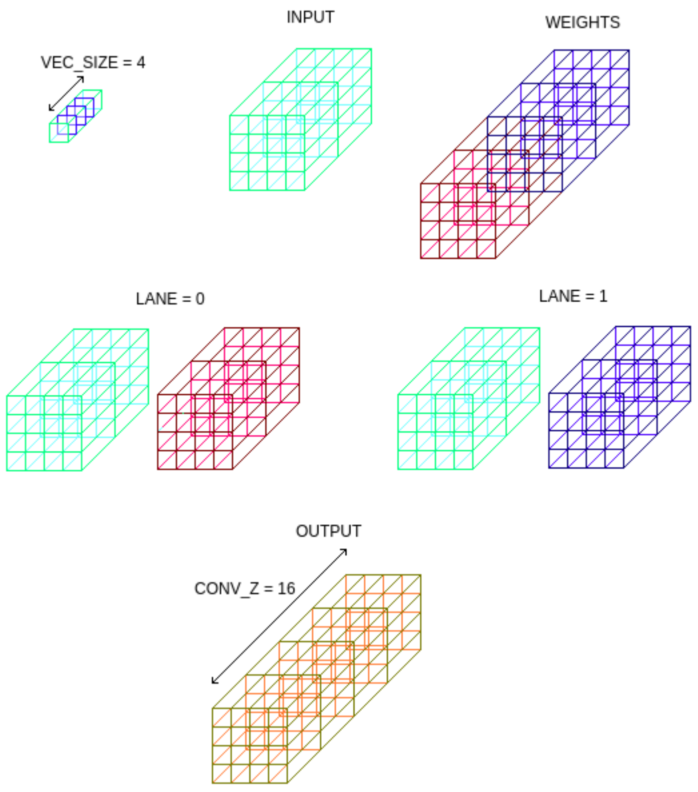

Figure 3.4: In this example, there's a 4x4x8 block of feature maps as input and a block of 4x4x8x2 of weight values. Each lane takes one of the outputs, in this case. After the convolution process is done LANE=0 will have its results set to z=[0,8] and LANE=1 will have its results written on z=[9,15].

As it was stated before, these are just conceptual and illustrations of how data is organized. All the

```
for(m = 0; m<(dim4/LANE_NUM); m++){
    for(n = 0; n<(dim3/VEC_SIZE); n++){
        for(i = 0; i<dim2; i++){
            for(j = 0; j<dim1; j++){
                for(ll = 0; ll<LANE_NUM; ll++){
                    for( k = 0; k<VEC_SIZE; k++){
                        array[m*dim1*dim2*dim3*LANE_NUM + ...
                            n*dim1*dim2*VEC_SIZE*LANE_NUM + ...
                            i*dim1*VEC_SIZE*LANE_NUM + ...
                            j*VEC_SIZE*LANE_NUM + ll*VEC_SIZE + k]
                    }
                }
            }
        }
    }
}
```

Figure 3.5: Algorithm describing how data is organized in PipeCNN.

data is stored inside a 1D array. The decomposition of the matrix is exemplified in Figure 3.5.

3.2.5 Fixed-point adjustment

Using fixed-point requires adjustment and scaling of the results after the convolution process. The number of fractional bits are decided in the host and can be adjusted easily. There are three different values to be adjusted, weight_frac, input_frac and output_frac.

Since there is a need to have the weight values stored in a file, the weight_frac is essentially already chosen. When creating the file that will serve as the weight input in MATLAB, the user needs to choose the amount of bits used for fractional values and this value will be used as the reference. For input_frac and output_frac there is more flexibility and it's up to the user to choose accordingly of the best results. There is room for optimization but also, if not chosen carefully, both of these can be detrimental to the overall results.

Instead of adjusting the input to the same scale as the weights, the convolution operation happens before it. This means, there are multiplications and additions happening with numbers in different scales, but it does not affect the final results. Storing these values is done using buffers with 32 bits (double the amount of **short** data type) after which there is a rescaling operation by simply shifting the results by (weight_frac+input_frac-output_frac-1) bits and adding '1' as a rounding bit. After this shift, if the data is bigger or lower than the determined threshold (32767 and -32767 for **short** values), it is stored as this limit value. After determining the value of the result in the new data type, it's stored as this data type and then sent to the MemWr function.

Although there is only rescaling after the arithmetic operations, the use of rounding techniques allows for precise results in the end.

3.2.6 Bias and ReLU

Original PipeCNN uses the activation layer only to discard values lower than 0. This is the operation that the creators of the framework needed, as the CNN architectures used only needed the classic ReLU. FlowNet-S uses a different ReLU function, it multiplies the results by 0.1. Because of this there were changes necessary to the way this function works. ReLU was only called in the end of the rescaling process, after it had written the values in the functioning data type, **short**.

To correctly divide the resulting output of the convolution by a factor of 10 in case it was negative, Bias had to be added before the rescaling. So, Bias values are now rescaled by (weight_frac + input_frac) bits, which scales the value to be on the same magnitude as the input. Some pseudo-code in Figure 3.6 illustrates the ReLU function as well as the rescaling of the resulting values

```
conv_sum_bias[ll] = (conv_out[ll] + (bias[ll]<<(frac_w+frac_din))) + 0x01;
    if(conv_sum_bias[ll]≥0){
        conv_sign_exten[ll] = 0x0000;
        conv_with_rnd_bit[ll] = (conv_sign_exten[ll] | ...
            (conv_sum_bias[ll]>>(frac_w+frac_din frac_dout 1))) + 0x01;
    }else{
        if((relu&0x01)==0x01){ //relu activation
            conv_sign_exten[ll] = ¬...
                (0xFFFFFFFF>>(frac_w+frac_din frac_dout 1));
            conv_with_rnd_bit[ll] = (conv_sign_exten[ll] | ...
                ((conv_sum_bias[ll]/reluc)>>(frac_w+frac_din frac_dout 1))) ...
                + 0x01;
        }else{
            conv_sign_exten[ll] = ¬...
                (0xFFFFFFFF>>(frac_w+frac_din frac_dout 1));
            conv_with_rnd_bit[ll] = (conv_sign_exten[ll] | ...
                (conv_sum_bias[ll]>>(frac_w+frac_din frac_dout 1))) + 0x01;
        }
    }

    if(conv_with_rnd_bit[ll]≥32767)
        conv_adjust[ll] = MASK17B & 0xFFFF;
    else if(conv_with_rnd_bit[ll]< 32767)
        conv_adjust[ll] = MASK17B & 0x10000;

    else
        conv_adjust[ll] = (MASK17B & conv_with_rnd_bit[ll]);

    conv_final[ll] = MASK16B & ((conv_adjust[ll])>>0x01);

    conv_ch_in.lane[ll] = conv_final[ll];
```

Figure 3.6: Fixed-point adjustment, bias addition and ReLU operation pseudo-code.

4

Results and analysis

This chapter presents all the tests, results, and analyses that have been done to verify and validate the developed work. The first section concerns the overall performance of the system, comparing results with other platforms. The second part of this chapter concerns the added modules and its performances.

4.1 PipeCNN performance

Several input matrix sizes were experimentally tested as well as different pairs of images. Each test consisted of cropping, scaling and concatenation of some image pairs and a subsequent PipeCNN run. The tests on the GPU were made using the Caffe/Flownet2 platform and serve only as a general comparison of what is out there at the moment. Data regarding power usage in the FPGA is an estimation using PowerPlay Power Analyzer Tool in Quartus II.

4.1.1 FPGA results

There are several aspects that are adjustable and tweakable in this framework. The two biggest factors that affect a system like this is the size of the input images, as well as the data range used. PipeCNN originally uses 1 byte for data. This was proven not to be enough as results can easily tell. Another factor that was decided was the size of the input, testing was made using 384x384 images. Initially this was chosen because memory usage was a big factor impacting the compilation of the kernel. FPGAs synthesis process is complex and sometimes, even when estimations of resource usage was at around 50% on most parameters, the compiler would not finish compilation and wouldn't be able to fit the kernel in the device. Other times it would finish successfully with kernels that used larger kernels and more resources. For this reason, the choice for the size was conservative but it is not the maximum the system allows.

384x384 is probably the only size that is small, but big enough to guarantee reasonable results, for the particular CNN architecture (FlowNet-S) used. Other sizes, such as 256x256 were tested initially, but even using MATLAB as the engine, produced bad results. The same pair of images was used in all of the comparisons. These are depicted in Figure 4.1. Other pairs of images were tested to understand if the framework had similar quality of results.

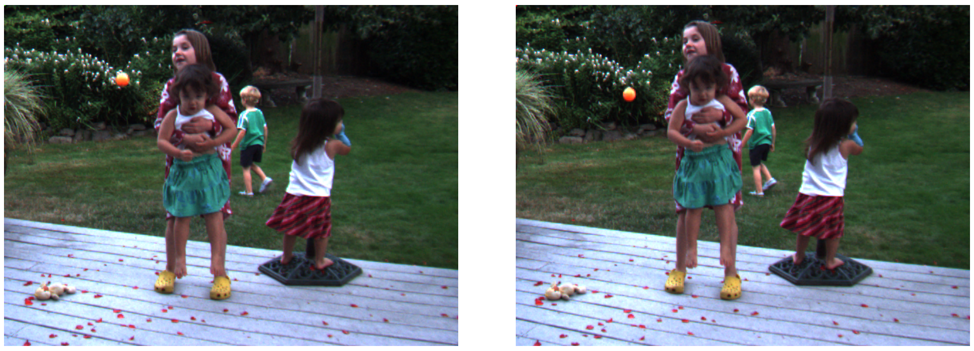

Figure 4.1: Backyard scene, the biggest movement happens in the middle, with both girls facing the camera going up and down, but also the orange ball going downwards

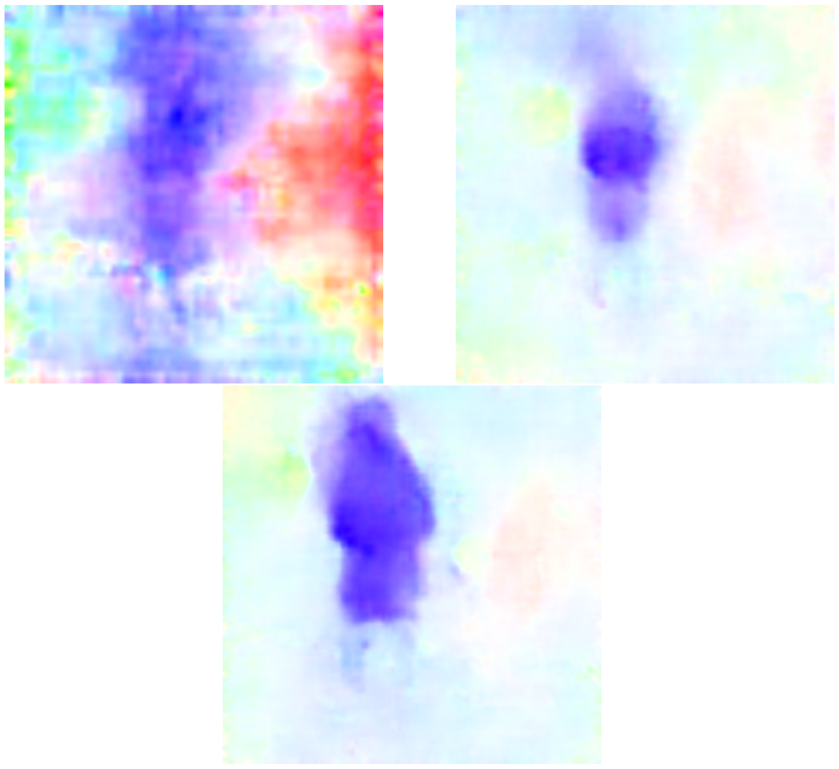

Figure 4.2: Example of three optical flow estimations depending on the size of the input image, from left to right: 256x256, 384x384 and 448x448

The differences between using different resolutions is apparent in the final optical flow estimation. In Figure 4.2. there are three different optical flow estimations based on three different resolutions for the same pair of images.

This scene was used primarily because it has reasonable results even using small resolution images. As discussed in [7], the optical flow estimation map is smaller by a factor 4 than the input images. The output map was resized using bilinear interpolation to match the input size.

Because of how PipeCNN works, all the fixed-point are decided in the host on a layer by layer basis. This means there's some flexibility in results but also that, if it's not setup correctly the

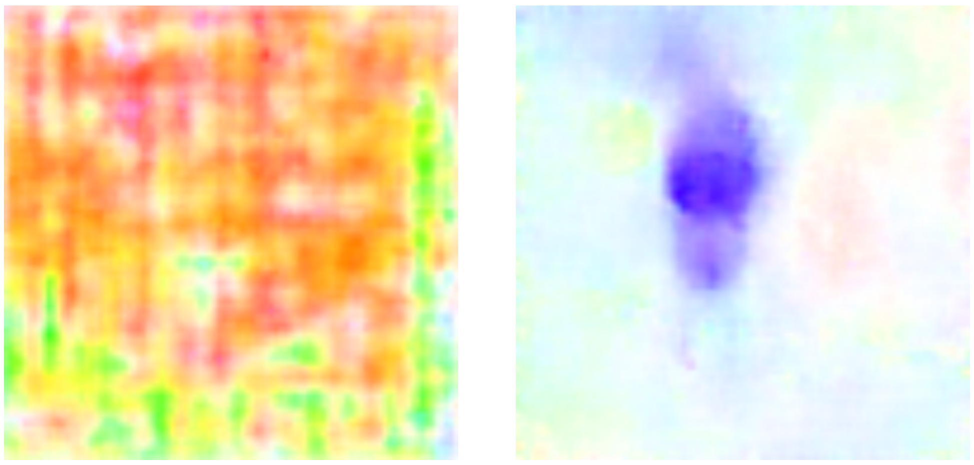

Figure 4.3: The 8-bit example on the left and the 16-bit output on the right

system might output bad optical flow maps even though it works correctly.

This was also true when the system used 8 bits only, it did not work correctly. Experimentation was done to try and mitigate this small data range by using adequate fixed-point values for each layer. Each layer has a designated fixed-point scale. When normalizing the outputs, some results are lost if they fall above the stipulated data range. To combat this, there was a change in the type of data used. Instead of **char**, **short** was used, which is a 16-bit type of variable. This made a very big difference and helped have correct results.

Due to the nature of how convolutions work, hitting the maximum value early on increases the subsequent layers. This is why the 8-bit result in Figure 4.3 looks so different from the 16-bit.

Using 16-bit data allowed to have results very similar to ground-truth, almost unnoticeable to the naked eye in some cases. Figure 4.4 presents some examples of the differences between the output of the FPGA and the output of the ground truth. Optical flow estimation has a bit of a subjective aspect to it, since it's only the conveyance of motion happening. It's not easy to easily discern if there's motion happening in some cases. FlowNet-S has a very big problem of being quite blurry, there aren't very defined edges and for more accurate results, more precise CNN architectures should be used. There isn't a way to measure accuracy by comparing the results obtained in PipeCNN with the ground truth. Even if there was a comparison with median values, it is easier to understand results just by looking at the differences between pictures.

Ground truth in Figure 4.4 was achieved by running FlowNet-S in MATLAB, because we are comparing PipeCNN's performance and results with a stable engine.

4.1.2 Experimental results

The system's performance can essentially be measured by the speed and time consumed doing each operation and by each layer. Not only that, but the resources used by the synthesis of the kernel.

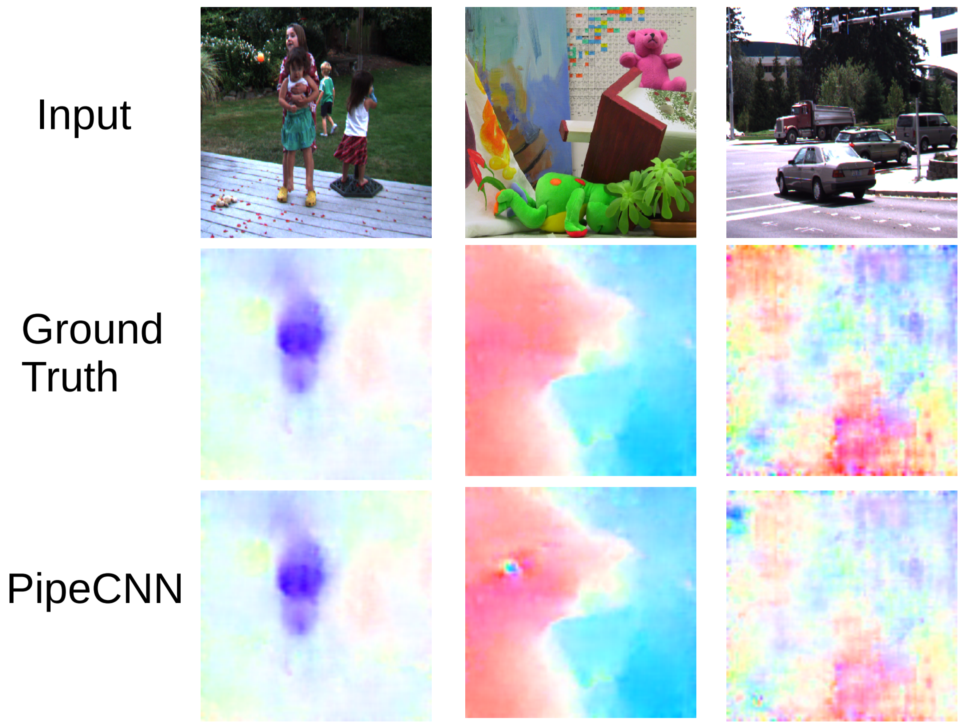

Figure 4.4: Example of results using PipeCNN comparing to ground truth. These three pictures are a good example that there's some images that reveal no difference to the naked eye, others are similar but there are examples where there's clear disconnection. It's clear that FlowNet-S is only good in certain scenes as well.

The size and resources used in each compilation of the kernel lie in the PipeCNN configuration file called 'hw_param.cl'. VEC_SIZE and LANE_NUM parameters decide the size of the pipeline as it was discussed before.

The number used in LANE_NUM needs to be a multiple of the output sizes, using a value which is not will result in a higher number of logic utilization and it will not be efficient as it will waste logic units.

Compilation of the kernel was only successful using LANE_NUM=2,4 and 8. The compiler could not fit a larger kernel, but as it was stated before, there are some values which the compiler cannot resourcefully contain in the board used. For example, LANE_NUM=6 was tried as well, and although the estimations of resources used in the pre-compilation seemed achievable, it was never successful. Reducing the variable VEC_SIZE and increasing LANE_NUM also did not have a successful compilation.

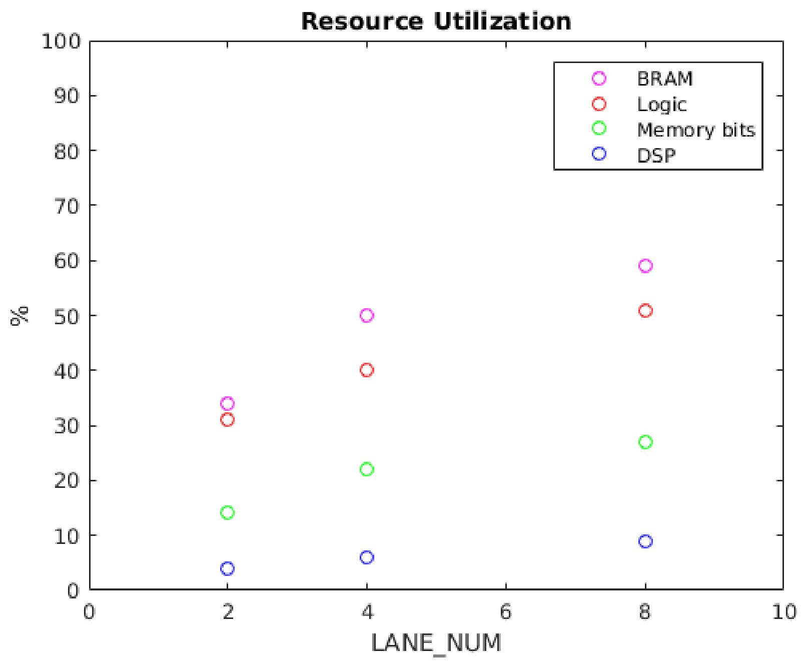

Figure 4.5: Resources used by the different LANE_NUM values that successfully compiled.

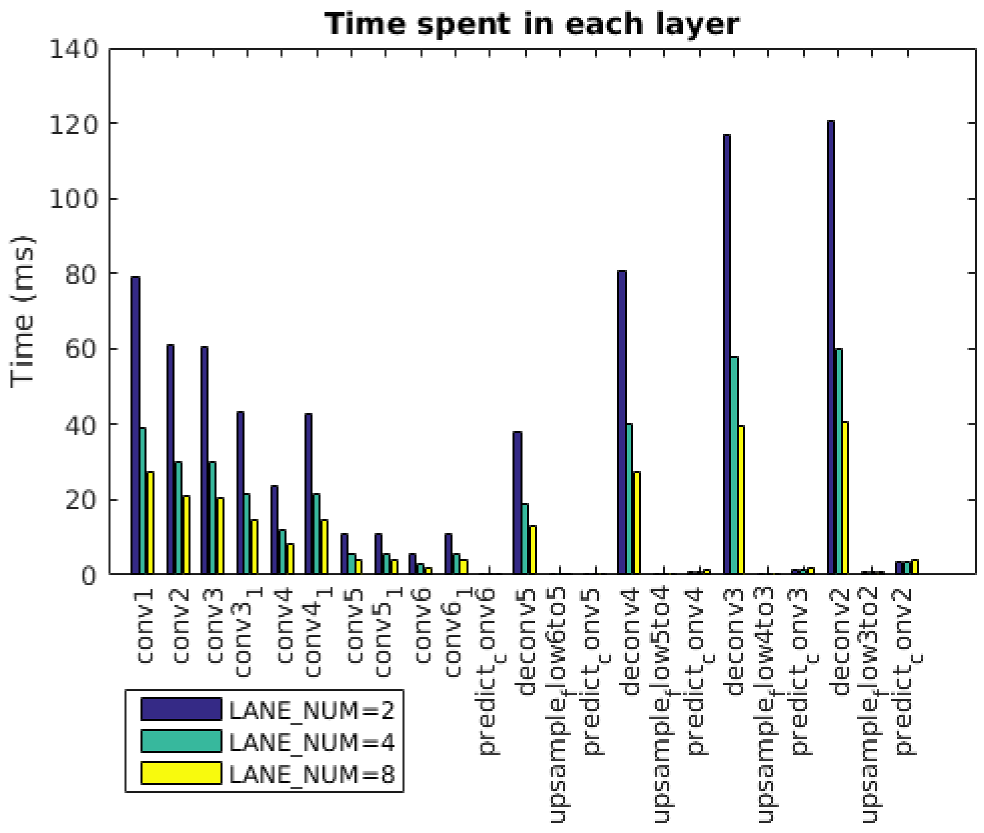

Figure 4.6: Time spent by each layer in the different successful compilations of the kernel.

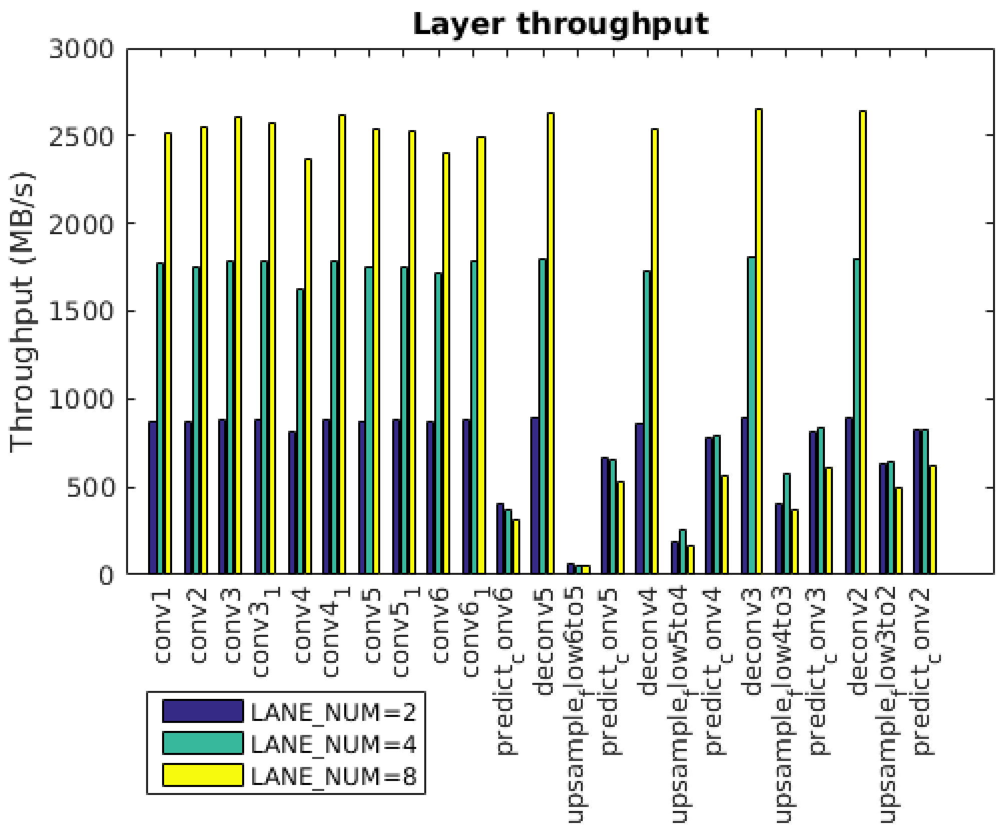

Figure 4.7: Throughput of FlowNet-S on PipeCNN in MB/s

LANE_NUM	Total time spent
2	710 ms
4	355 ms
8	248 ms

Table 4.1: Total time spent by each kernel size

4.1.3 Discussion

PipeCNN has proven that it can compute optical flow estimations. Although the use of floating-point data is substituted by fixed-point, the end result is satisfactory. This type of system can always be fine-tuned as well. Other variations of the decimal point disposition can still be explored and optimized.

The use of FlowNet-S serves its purpose of being a simple but robust CNN, which can estimate OF.

Resource utilization was taken from the files that are generated after compilation. All of the resources referred are part of the kernel and belong to the FPGA board. Only a small part of the resources available on this board are currently being used as we can see in Figure 4.5. The biggest bottleneck is the block RAM memory, one of the reasons for this is that this architecture needs to store results of several layers. One way to reduce the usage of BRAM would be to transfer these buffers with results to the host, and then have the host communicate them back to the kernel when they are needed. This would reduce the use of global memory (BRAM) but would increase the latency in the layers in which they are needed.

There's also a deep lack of DSP block usage. By issuing the use of the RTL blocks directly on the code, to perform the multiplication and addition of the data, it would be interesting to see if there was any relevant difference in performance. This could be achieved either by rewriting the Board Support Package of the ProceV board or by just using a different board altogether.

Even though it seems the board isn't being used to its full potential, the compiler wasn't able to fit larger kernels than LANE_NUM=8. Using an updated version of the SDK could solve this problem, but it's still unknown.

Looking at Figure 4.6. it is clear the transpose convolution layers are computationally intensive. The amount of data that needs to be processed is very large compared to the first few convolutions which need a much lower input size. Using the state-of-the-art method to compute them could increase the performance of this framework. Right now, PipeCNN achieves 0.71,0.35 and 0.2 results per second (Table 4.1.), its biggest bottleneck being the transposed convolution. If it was possible to obtain a 4x speedup, by using the algorithm described in [15], in every transposed convolution the time spent for the whole process would be reduced to around 443 ms, 223 ms and 158 ms, which would be a reduction of about 38%, 37% and 36%, for each LANE_NUM size respectively.

Looking at Figure 4.7 it may seem easy to misjudge the performance on the transpose convolution layers. PipeCNN performs as well on transpose convolution as in convolution, the problem is that the current algorithm needs to be changed to be much more data efficient. The current implementation is not efficient in data usage as a lot of the multiplications happening are redundant.

Because of how PipeCNN works, computation of the 'predict_conv' and 'upsample_flow' layers will always have bad throughput. Not only because the amount of data processed is small but also because adding more parallelization via additional LANES won't result in any better performance. Each LANE processes part of the output, and since the output is always '2' for those layers, most LANES are being wasted computing zero as inputs.

It is obvious that there are several optimizations possible in a system like this. Algorithm improvements especially, but also a more extensive use of the board's resources. PipeCNN could be sped up significantly.

4.2 Comparison between FPGA and GPU

This section analyzes whether using an FPGA is a good option in regards to results' precision, speed and electrical efficiency. Comparisons with a GPU setup are essential to understand the value an FPGA can bring.

4.2.1 Direct Comparisons

The comparisons happening here need to take into account the different setups. The GPU used Caffe as its engine, using a very capable and powerful unit, that is the Nvidia's Geforce GTX TITAN X. Caffe uses CUDA as the engine to calculate results, but the comparison is needed because the GPU is the more broadly used platform in this field. PipeCNN can still be thoroughly optimized in several aspects while Caffe is a very stable framework that doesn't have these problems with optimizations.

Regarding software usage, the GTX TITAN X uses the latest drivers available, while the ProceV has an outdated BSP. This could be hindering its performance, not only because the compiler used is not the most recent, but also because some of the features of more recent versions of the SDK are not available as well.

Power consumption is important to understand how both machines use energy and their efficiency. The power estimations done are a result of how this kernel behaves when it's running and they're achieved using the PowerPlay Power Analyzer tool of Quartus II. Thermal power dissipation is important because most of the power that is fed to the board is dissipated as heat from the device. Getting the actual power usage during runtime is not possible because special equipment is needed, for this specific board, to physically get the power spent, which is currently not available in our lab.

For the power measurements of the GPU, the command 'nvidia-smi' was used. Although the measurements are made in real-time, the following comparisons were done using the highest value recorded.

4.2.2 Experimental results

In this section we'll be comparing both systems. As it was discussed before, the engine used for the GPU processing was Caffe, the image sizes are the same (384x384) but the GPU does some pre-processing while the FPGA doesn't. The use of the different precision data is also important. The FPGA used 16-bit fixed-point while the GPU used 32-bit floating point data. Keeping that in mind, we can compare both systems:

4.2.3 Discussion

As we can see in Figure 4.8 the GPU computes the same amount of data much faster than all the kernels in the FPGA, while doing the additional pre-processing work. CNNs are very computation-

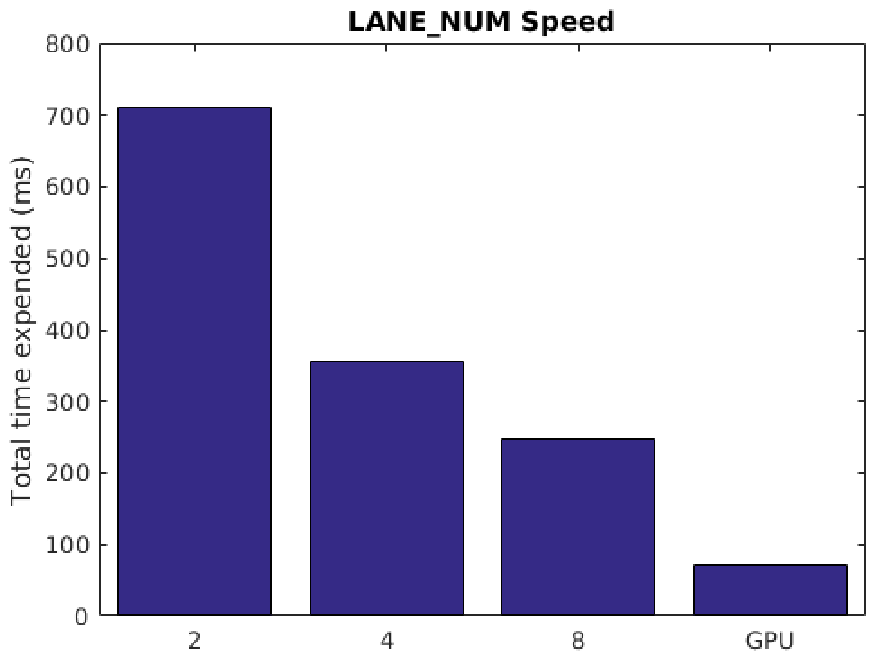

Figure 4.8: Total time needed to compute a frame of dense optical flow given two input image frames. Comparing the performance of different sizes of hardware, defined by LANE_NUM, with the GPU.

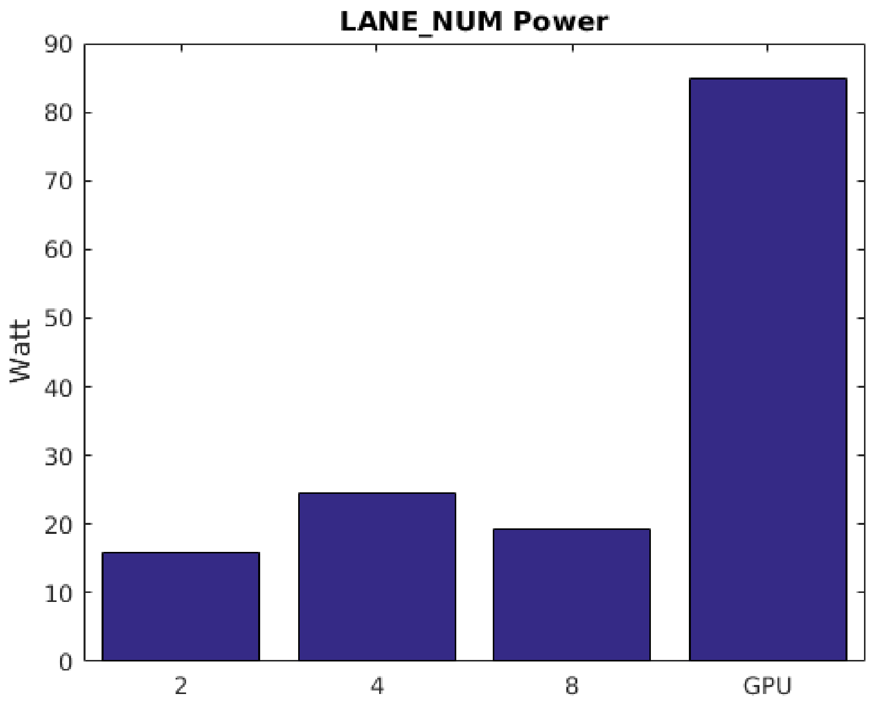

Figure 4.9: Comparing the different power needs between different sizes of hardware, defined by LANE_NUM, with the GPU.

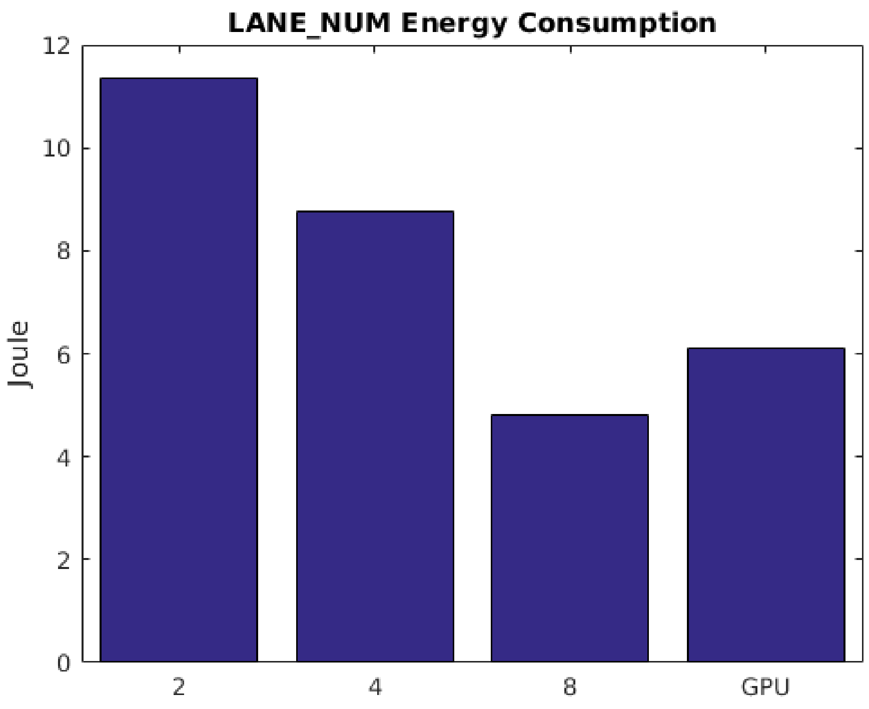

Figure 4.10: Energy consumed to compute a frame of dense optical flow given two input image frames. Comparing the performance of different sizes of hardware, defined by LANE_NUM, with the GPU.

ally intensive systems and in this state the FPGA cannot compete directly. Several improvements are needed until the system runs on the same level as the GPU, comparing both boards (FPGA and GPU) used.

Although using high-end FPGAs, the specific board company does not provide an updated BSP. The lack of recent BSP updates are probably very devastating as newer compilers will probably optimize the circuits more efficiently.

The parallelization aspect of having several ’sequences’ of layers running exists in the Caffe models, but isn’t taken advantage of in PipeCNN. In the case of FlowNet2-s, the ’deconv’ layers can be run in parallel with the ’predict_conv’ and ’upsample_flow’ layers. Also, most recent CNN architectures try to have several layers running in parallel. To prepare PipeCNN for the future, work could be done to improve it in this area.

Reconfigurable logic wins by a lot when the comparison is made in the power usage subject. This system consumes much less power, per second, than the GPU. As it is shown in Figure 4.9.

The increase in power consumption when using LANE_NUM=4 and its decrease in LANE_NUM=8 can be explained by the lower frequency clock used in both systems. Because of the larger parallelization in LANE_NUM=8, the clock is running slower thus having decreased power usage.

In Figure 4.10, it is shown that the FPGA can surpass the GPU in electrical efficiency. This is what we were working towards. The FPGA achieves better results than the GPU considering. The difference is not very large, but with some optimizations, it could be improved.

Depending on the applications, electrical consumption is a very important measure as the world is turning into more handheld devices. These devices are mostly run on a battery and being energetically efficient is a very important aspect of technologies like this.

Because CNNs are very calculation intensive systems, which is exactly the type of systems GPUs strive in, FPGAs cannot competing directly with them. Further optimizations can still be obtained addressing direct RTL instead of using OpenCL.

5

Conclusions and further work

The goal of this dissertation was to develop new modules in an open-source framework to allow it to estimate optical flow. The work that was done can be utilized in several types of applications and it is not restricted to optical flow estimation. Creation of new modules in this open-source framework allow for new work to be achieved easier in this field. Development of with the function of computing transposed convolution, concatenation layer as well as changes in specific architectural nuances allowed for the use of an optical flow CNN to be used in a reconfigurable logic environment. Which as far as we are aware, there are not any. The performance of this framework was reasonable but it still cannot compete with the frameworks that use a GPU. The end goal was achieved but there are improvements to be made to this framework. Power usage superiority still is not enough for a system that is, most likely, wanted for real-time applications.

Testing the modules developed on a higher-end FPGA board might lead to more conclusions. Since the compiler is not able to successfully synthesize the hardware necessary for it to run on a larger configuration even though its pre-compilation estimations don't use all of the resources on the board. The more extensive use of the board space is still a bit unexplored. It would improve the system's efficiency most likely, but at the same time some core changes are needed to achieve better results. The development of a different algorithm for the transposed convolution as well as optimization of the 'concat' layer are essential.

Hopefully this work can open the door for other developments in the field. By working on this open-sourced framework, the window for access is very large and there's hope for more advancements to be made. New work should be developed to allow for different optical flow CNNs to be deployed using this framework. New layers such as 'correlation' and 'warp' are very important in today's CNN architectures and it would allow for the expansion of this FPGA framework into new territory. This territory brings much more precise results than does possible to achieve with FlowNet-S.

To completely discard reconfigurable logic's usefulness in this field at least some of the improvements stated before need to be accomplished. The trade-offs between hardware complexity, computation time, power consumption and precision achieved using an FPGA are certainly very important in this field and can be useful in a wide range of applications. Maybe with some more developments in this field, the FPGA can become a more fierce competitor.

Bibliography

[1] Netscope. `https://ethereon.github.io/netscope/quickstart.html`.

[2] Saleh Albelwi and A. Mahmood. A framework for designing the architectures of deep convolutional neural networks. *Entropy*, 19, 06 2017.

[3] Louis Andrianaivo, Roberto D'Autilia, and Valerio Palma. Architecture recognition by means of convolutional neural networks, 07 2019.

[4] M. J. Black and P. Anandan. A framework for the robust estimation of optical flow, 1993.

[5] Andrés Bruhn, Joachim Weickert, and Christoph Schnörr. Lucas/kanade meets horn/schunck: Combining local and global optic flow methods. *International Journal of Computer Vision*, 61(3):211–231, Feb 2005.

[6] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. 03 2016.

[7] Philipp Fischer, Alexey Dosovitskiy, Eddy Ilg, Philip Häusser, Caner Hazirbas, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. *CoRR*, abs/1504.06852, 2015.

[8] Sneha H.L. The why and how of pipelining in fpgas. `https://www.allaboutcircuits.com/technical-articles/why-how-pipelining-in-fpga/`, 2018.

[9] Berthold K.P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1):185 – 203, 1981.

[10] Tak-Wai Hui, Xiaoou Tang, and Chen Change Loy. Liteflownet: A lightweight convolutional neural network for optical flow estimation. *CoRR*, abs/1805.07036, 2018.

[11] Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy, and Thomas Brox. Flownet 2.0: Evolution of optical flow estimation with deep networks. *CoRR*, abs/1612.01925, 2016.

[12] Jefkine. https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/. `https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/`, 2016.

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, page 675–678, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Yann Lecun. A theoretical framework for back-propagation. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School, CMU, Pittsburg, PA*, pages 21–28. Morgan Kaufmann, 1988.

[15] Shuanglong Liu, Hongxiang Fan, Xinyu Niu, Ho-cheung Ng, Yang Chu, and Wayne LUK. Optimizing cnn-based segmentation with deeply customized convolutional and deconvolutional architectures on fpga. *ACM Trans. Reconfigurable Technol. Syst.*, 11(3), December 2018.

[16] middlebury.edu. flowio.cpp. `http://vision.middlebury.edu/flow/code/flow-code/`.

[17] Anurag Ranjan and Michael J. Black. Optical flow estimation using a spatial pyramid network. *CoRR*, abs/1611.00850, 2016.

[18] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988.

[19] D. Wang, K. Xu, and D. Jiang. Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 279–282, Dec 2017.

[20] Randy Yates. Fixed-point arithmetic: An introduction. `https://courses.cs.washington.edu/courses/cse467/08au/labs/l5/fp.pdf`, 2007.