UPPSALA
UNIVERSITET

UPTEC E 19006
Examensarbete 30 hp
2019-06-10

Acceleration of deep convolutional neural networks on multiprocessor system-on-chip

Martin Reiche Myrgård

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Acceleration of deep convolutional neural networks on multiprocessor system-on-chip

Martin Reiche Myrgård

In this master thesis some of the most promising existing frameworks and implementations of deep convolutional neural networks on multiprocessor system-on-chips (MPSoCs) are researched and evaluated. The thesis' starting point was a previousthesis which evaluated possible deep learning models and frameworks for object detection on infra-red images conducted in the spring of 2018. In order to fit an existing deep convolutional neural network (DCNN) on a Multiple-Processor-System on Chip it needs modifications. Most DCNNs are trained on Graphic processing units (GPUs) with a bit width of 32 bit. This is not optimal for a platform with hard memory constraints such as the MPSoC which means it needs to be shortened. The optimal bit width depends on the network structure and requirements in terms of throughput and accuracy although most of the currently available object detection networks drop significantly when reduced below 6 bits width. After reducing the bit width, the network needs to be quantized and pruned for better memory usage. After quantization it can be implemented using one of many existing frameworks. This thesis focuses on Xilinx CHaiDNN and DNNWeaver V2 though it touches a little on revision, HLS4ML and DNNWeaver V1 as well. In conclusion the implementation of two network models on Xilinx Zynq UltraScale+ ZCU102 using CHaiDNN were evaluated. Conversion of existing network were done and quantization tested though not fully working. The results were a two to six times more power efficient implementation in comparison to GPU inference.

Handledare: Markus Jangblad
Ämnesgranskare: Carlos Pérez Penichet
Examinator: Tomas Nyberg
ISSN: 1654-7616, UPTEC E 19006

Populärvetenskaplig sammanfatting

MPSoCs är ett samlingsbegrepp för flerprocessoriga System-on-chips och de bygger oftast på en eller flera processorer kombinerat med programmerbar logik och andra hårdvaruelement. Dessa utgör en utmärkt plattform för att exekvera beräkningstunga algoritmer som de som används i maskinlärning och artificiell intelligens. Anledningen till att denna rapport undersöker möjligheten till en mobil plattform för denna typen av beräkningar är att det finns ett intresse i dagens industri att kunna använda Ai för bild analys i mindre och ofta strömbegränsade miljöer. Den specifika miljön som är av intresse i detta fall är ett flygplan.

Utgångspunkten för detta examensarbete grundades i avslutningen av ett annat examensarbete där uppgiften var att ta fram en modell för att med hjälp av Ai analysera och detektera landningsbanor och landningsljus automatiskt på bilder från en IR-kamera och sedan rita ut rektanglar kring objekten. Examensarbetet hade som mål att få en så säker och exakt träffbild som möjligt med hjälp av ett AI nätverk tränat på ett dataset med IR-bilder. Kameran kunde både ta in långvågs- och kortvågsinfraröda bilder vilket gav indata på två separata kanaler. Det finns en uppsjö av deep convolutional neural network (DCNN) nätverk som kan utföra objektdetektering men ett specifikt vid namn RetinaNet valdes. De bygger på en rad olika lager som faltningslager som genom faltning med vikter projiceras till nästa lager som analyserar dess indata vidare. Att träna ett sådant nätverk innebär att det tillåts analysera givna bilder med specificerade klasser och ramar som visar var på bilden dessa klasser hittas och sedan uppdatera detta själv med hjälp av hyperparametrar och vikter. Vikterna uppdateras itterativt i en träningsprocess beroende av hur nära nätverkets prediktion stämmer överens med verkligheten.

Detta examensarbete började med att analysera möjligheterna till modifiering av det befintliga RetinaNet nätverket och vilka alternativa nätverk som eventuellt skulle passa bättre i denna implementation. Slutsatsen av undersökningen var att det fanns ett par verktyg för att konvertera nätverket från ramverket Keras till Caffe vilket är ett ramverk som ofta används som indata till en implementation på en MPSoC. Att hitta ett passande ramverk för att implementera ett DCNN på en MPSoC var huvuduppgiften för detta examensarbete. Vid litteraturstudien konstaterades att det finns en del olika ramverk men att de flesta är byggda av en mindre forskningsgrupp, ofta för att utföra en väldigt specifik uppgift. Det finns dock en del "open source" ramverk, alltså ramverk med öppen källkod för att implementera DCNN på MPSoCs samt två bibliotek från Xilinx vilket är samma företag som tillverkar kretskorten som fanns att tillgå. Detta gjorde det ganska självklart att välja Xilinx ramverk initialt med ett eller två "open source" ramverk att testas mer för bredd i undersökningen.

Två av de vanligare metoderna för att modifiera ett redan existeande nätverk är att kvantificera vikterna och att beskära nätverket. Vid kvantificering avrundas vikterna till närmaste kvantificeringstal vilket minskar antalet bitar som krävs för att beskriva vikten. Man kan även beskära de vikter som påverkar nätverkets beslut minst.

Slutsatsen är att det finns en del tillgängliga ramverk för implementering men många av dem är ofullständiga eller stödjer inte nätverkstypen som var tilldelad. Att konvertera nätverket tränat på IR-bilderna gick inte. Däremot fungerade det att konvertera ett mindre närverk som var skapat i det tidigare examensarbetet. Att sedan kvantifiera det mindre nätverket och anpassa det för en implementation i CHaiDNN med hjälp av Xilinx Quantizer gick inte med det svårtolkade felmeddelandet: "KeyError: 'data'". Istället implementerades två andra nätverk på MPSoCen. Ett var samma typ som grunden till RetinaNet nämnligen ResNet50. Detta gick att köra på MPSoCen med resultat på mellan 2 till 6 gånger mer effektivt i jämförelse med en Graphics processing unit (GPU) vid jämförelse av bilder/sekund/watt. Detta var dock endast ett bild-klassificeringsnätverk. Därför testades även en annan typ av objektdetekteringsnätverk på MPSoCen. Detta presterade sämre än både teoretiska värden och evaluering på en GPU i termer av bilder/sekund. En intressant fortsättning av arbetet kan bygga på en analys av de mindre nätveken som finns tillgängliga vilka är anpassade för att köras på mobila plattformar och har visat sig prestera i paritet med datorexekveringar.

Acknowledgements

There are a few people who i would like to acknowledge for helping me getting this job done with all the challenges and complications it gave. Firstly i would like to thank my supervisor at Saab
Markus Jangblad for his introduction and guidance to everything between the majestically big and confusing directory composition of his master thesis to the coffee machine as well as his inspiring drive and motivation in his every day work.

I would also like to thank Joakim Lindén at Saab for his expertise and knowledge regarding hardware and Ai. He has been supportive and helped me understand most of the obstructions interfering with this work. He's also been the second reader on this report giving valuable feedback on this thesis.

Then i would like to thank Carlos Pérez Penichet, my subject reviewer at Uppsala University for his help and his opinion on the thesis and subject. An honourable mention to Erasmus Cedernaes at Saab for his help with all the struggles with desktop, Python, SDSoC and other software issues encountered during the thesis.

Acronyms	
Artificial intelligence	AI
Block Random access memory	BRAM
Convolutional Neural Networks	CNN
Deep convolutional Neural Networks	DCNN
Data Motion Clock	DMC
Deep neural network development kit	DNNDK
Digital signal processing	DSP
Enhanced flight vision system	EFVS
Fully connected	FC
Fully convolutional network	FCN
Flip-flops	FF
Fast fourier transform	FFT
Field programmable gate array	FPGA
Giga operations per second	GOPS
Graphic processing units	GPU
High level synthesis	HLS
Head up display	HUD
Integrated circuit	IC
Integrated development environments	IDEs
Instrument landing system	ILS
ImageNet large scale visual recognition challenge	ILSVRC
Intersection over union	IoU
Intellectual property	IP
Infra red	IR
Joint test action group	JTAG
Long short-term memory	LSTM
Lock-up table	LUT
Long wave infra red	LWIR
Multiply Accumulate	MAC
Mean average precision	mAP
China's Ministry of Industry and Information Technology	MIIT
Multiple inputs multiple outputs	MIMO
Machine learning	ML
Model Management deep neural network	MMdnn
Non maximum suppression	NMS
Operating system	OS
Precision approach path indicator	PAPI
Peripheral Component Interconnect Express	PCIe
Programmable logic	PL
residual network	RESNET
Recurrent neural networks	RNN
Read only memory	ROM
Software-defined system on chip	SDSoC
Single shot detection	SSD
Short wave infra red	SWIR
Terra floating point operations per seconds	TFLOPS
Very High Speed Integrated Circuit Hardware Description Language	VHDL
You only look once	YOLO

Contents

1 Introduction

Ever since the aeroplane was invented by the Wright brothers in 1903 the problem of take-off and landings has been one of the issues concerning airborn vehicles. To get a motor driven machine in the air and back on the ground safely was and still is a rather delicate task with the majority of accidents occurring during takeoff and landing[1]. Over the decades runways and aeroplanes have evolved to ease the procedure for the pilots in a variety of ways. Initially a patch of grass was enough for the aeroplanes to take off and land on and since they were not particularly heavy or fast they therefore did not need much guidance. During the world wars, aeroplanes started to play a role in warfare which resulted in heavier and more sophisticated aircrafts that needed better guidance through instrumentation. This has resulted in a lot of research and development of aeroplanes and airstrips.

Modern aircrafts are guided by a set of computers and micro controllers both on the ground and in the air. These computers process a lot of information from all kinds of sensors and other sources. Most of the tasks they face are predictable or manageable through human machine interaction or by human expertise. There are sensors indicating speed, height, attitude (yaw, pitch and roll angle)[2] and more. Modern aircrafts also have a variety of radars on board as well as GPS receivers to indicate position of themselves and other aircrafts. Airstrips are equipped with instrument landing system (ILS) transmitters and receivers to guide inbound aircrafts to the right flight course and glide angle. The analogue airstrip guidance are landing lights and precision approach path indicator (PAPI) lights that show the extended vector of the runway and the correct angle for approach. The PAPI lights indicates the flight altitude and angle by a set of four lights[3].

One thing concerning take-off and landing that can not be controlled or worked around is the weather. Therefore, pilots use modern instruments to land in non perfect weather conditions. There are several different ways the instrumentation uses the sensor readings to guide the plane to its runway. Heat emitted by approaching lights, PAPI lights and heat generated on the runway by friction can be detected in poor sight conditions with infra red (IR) sensors which can be mounted on the aeroplane with its readings processed and displayed for the pilots. One way to help the pilots identify landing guidance objects is object detection on IR images through machine learning by deep convolutional neural networks (DCNN).

1.1 Background

In march 2018 the standards regarding the use of enhanced flight vision system (EFVS) changed for aeroplanes which allowed them to land in harsher weather conditions with low or no visibility[6] using EFVS systems. The standards made it possible to descend from 100ft to the ground using EFVS in lieu of direct vision of the runway.

The technology for more advanced visual assist systems exists and some of it is already in use but modern artificial intelligence (AI) based systems has not yet reached the avionic world. The object detection of DCNN have a low error rate and is well suited for acceleration on power efficient hardware like MPSoCs. This means it can be useful in assisting in the landing procedure by visualising set objects for the pilot on a head up display (HUD).

There exists a few trained AI networks for IR image detection but they mostly rely on graphic processing units (GPU) to execute. To get them operational on an aircraft however they need to operate in real-time and be more energy efficient then a GPU with as low latency as possible. This masters thesis aims to research different ways of modifying DCNN networks to run on a field programmable gate array (FPGA). FPGAs are known for quick and power efficient execution of complex tasks such as multiple matrix multiplications and are therefore a suitable platform to implement the real-time DCNN on.

1.2 Objectives

The master thesis was conducted at Saab Avionics Systems, Engineering Department for Video & Graphics Hardware section with the following objectives

- Literature study of Deep Convolutional Neural Networks
- Literature study of Xilinx UltraScale+ MPSoC
- Literature study of modification techniques of DCNN to be implemented on a SoC (Pruning and quantization e.g.)
- Study performance of an existing DCNN on GPU
- Adjusting an existing DCNN trained for object detection
- Implementation of DCNN on UltraScale+ MPSoC by different methods
- Comparison of performance and energy efficiency of different FPGA implementations and GPU performance

2 Theory

2.1 Deep convolutional Neural Network

The complexity of the human body has enraptured humans throughout history. Ever since the industrial revolution machines has been designed in order to assist or complete the human input in almost every thinkable application. Everything from heavy lifting to solving advanced mathematical problems with transistor-based computers[4].

More trivial problems for the human, like identifying what an image shows or pointing it out on the image has proven to be really complex to recreate with a computer. This area has been heavily researched the latest decade with a major breakthrough in 2012 when researcher Alex Krizhevsky won a competition[5] with the first DCNN called AlexNet and 2015 when the computer got a lower error rate than a human classifying the same images. Deep convolutional Neural Networks can often be dissected into three main components. The major component as the name suggests is the convolutional layers. Secondly there are layers for managing the size and resolution for the next layer. Last there is often a couple of fully connected layers to conclude all the features from the image and identify the class or object. The general structure of a DCNN can be described as in figure 1.

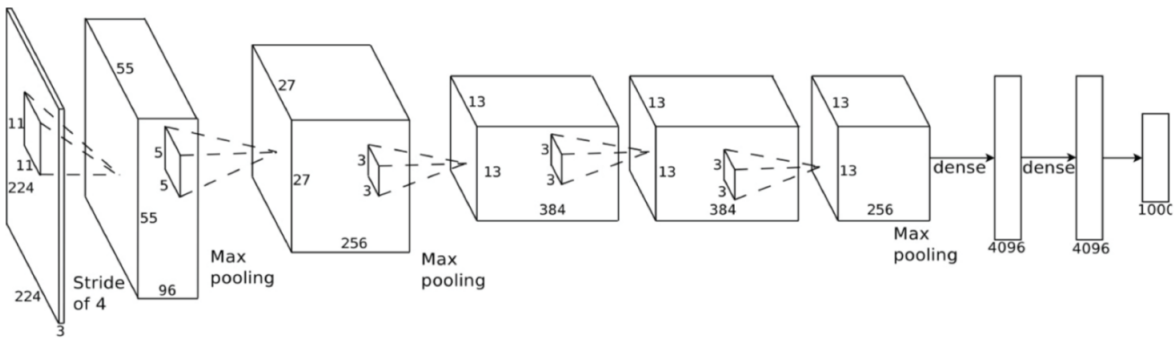

Figure 1: DCNN where each block consist of multiple layers

The structure is based on convolutional neural networks (CNN) with a higher solving capability for more complex cases. This is done by stacking more layers than a normal CNN and therefore gives DCNN its "deep" form. The convolutional part describes the weighting matrices of the different layers convoluted with the prior layer output. The whole system is inspired by the neurons in human or animal brains and how they seem to perform one simple task each but together manage to process and detect millions of details in an image in real-time. To replicate the process of neurons activating other neurons an image is fed to a DCNN and then processed through many layers activating the next layer in a DCNN. The network neurons all have different assignments in multiple layers to analyse and weigh different characteristics of the input image. The most common layers are convolutional layers consisting of weights that projects the input to create a weighted output. Figure 2 shows an example of a convolution.

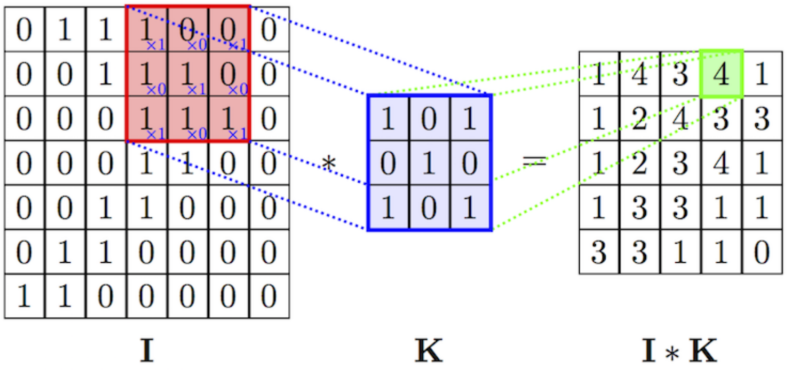

Figure 2: One input layer [I] with weighting matrix [K] convoluted to I*K

The pooling layer and flattening layers serve to optimise and control the filter and to adjust it to be compatible to the fully connected layers. The most commonly used pooling layer uses the max value in a certain area (usually 2x2 pixels) and scales down the picture. Therefore it's not a trainable parameter but rather controlled by the designer. This saves a lot of real estate in the following layers and eases the computational load. Figure 3 displays a model of a pooling layer.

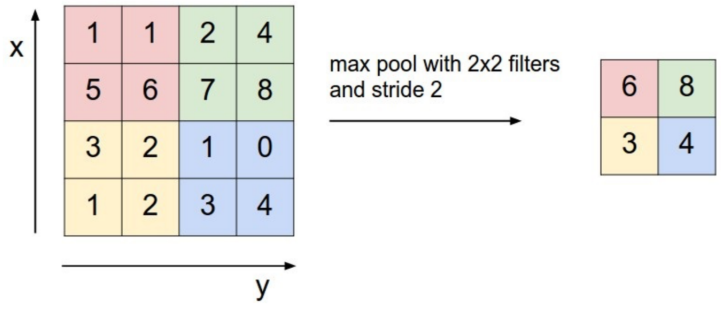

Figure 3: A max pooling layer reducing a 4x4 matrix to a 2x2

The input layers differ depending on the input but most common is a RGB input where the input layer separates the red, blue and green parts of the image for the neural network to process. Input layers can also be divided to long wave infra red (LWIR) images and short wave infra red (SWIR) images.

2.1.1 Object detection

There are different ways a DCNN can classify an image. Some are more complex than others and offers higher precision while others are less precise but executes quicker with lower latency. A DCNN can be trained both to detect and localise objects in images. In order to accelerate the process of object detection the bit width of the weights and image resolution can be adjusted as well as the number of convolutional layers and hardware utilisation amongst other optimisations. The most common ways for a DCNN to operate is image classification, object detection and image segmentation. Image classification classifies the image with a label based on the content of the image. The image segmentation implementation for DCNN classifies each pixel in an image to an object class and displays it by an overlay of colour for each object class found. This is a more precise way of detecting objects and their position in an image but requires heavier computations and the resolution is not a necessity for highlighting key runway and land assist objects. The midway solution to this is to use DCNN for object detection. An object of interest such as landing lights, PAPI lights and runway is boxed in to show its location without altering the visibility for the pilot. Figure 4 shows an example of a possible object detection network output.

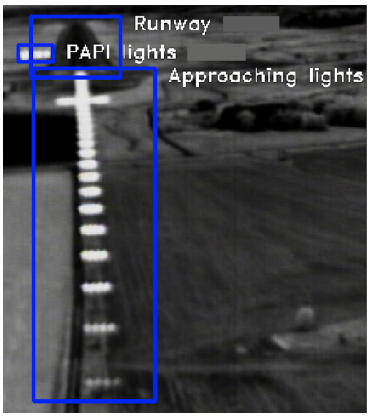

Figure 4: Example of an object detection network output using IR image as input

The DCNN supplied for this thesis to implement on an MPSoC is a result of a master thesis[7] from spring 2018 conducted for SAAB. The network is a RetinaNet which consists of a RESNET-50 (residual network)[8] as backbone trained on ImageNet [9] and a feature pyramid net in order to extract feature maps from different levels of the RetinaNet to classify the objects of interest. Figure 5 shows the structure of the RetinaNet, with the backbone classifying the image and how the feature pyramid net extracts mathich boxes.

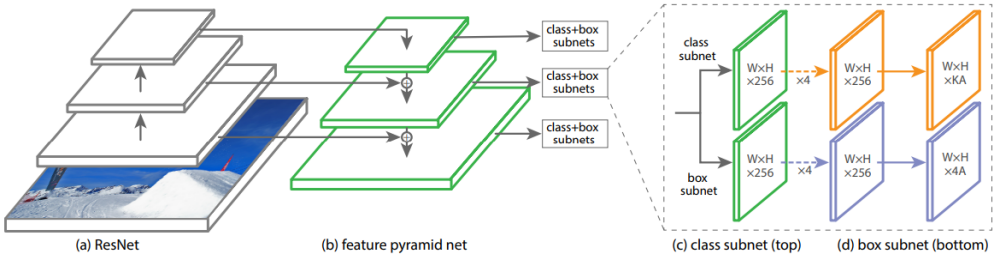

Figure 5: RetinaNet with a ResNet backbone and feature pyramid net for object detection

The network was created for GPU implementation and constraints of the FPGA was not considered while designing it. There is a variety of methods to measure an object detection network accuracy. One, which is most common in official papers, is the mean average precision (mAP)[10]. The mAP takes the mean of the

average precision score of each of the classes on a test set of images. The network used in the antecedent thesis had a mAP of 0.5071 on the COCO data set. In comparison, a published paper had a mAP of 0.5320 on COCO with images size 830x600 and IoU at 0.5 [11]. By using Bayesian hyper parameter optimisation on the data set of IR pictures from SAAB the object detection network got a mAP of 0.9060 [7] however this score is not to be confused with the COCO mAP which is a different evaluation on a different data set. The high mAP on the SAAB object detection network is due to the test data being vary similar to the training data, also known as over fitted data. Table 1 shows the performance of the trained object detection network on a nVidia GeForce Titan Xp GPU.

Image size	mAP	Prediction time (ms)	AP runway	AP approaching lights	AP PAPI lights
224 x 224	0.2446	18	0.1501	0.5419	0.0417
830 x 600	0.8463	42	0.7541	0.9439	0.8410
1107 x 800	0.8646	61	0.7894	0.9429	0.8614
1384 x 1000	0.8142	80	0.6804	0.8897	0.8725

Table 1: DCNN with RetinaNet architecture trained and evaluated on infra red images tested on a Titan Xp GPU

2.1.2 Modifying DCNN

The practical implementation of this network would be on a flying aeroplane with both size and power constraints concerning heat development making the choice of implementation platform crucial. The latency need to be as low as possible for real-time execution which means no batching and therefore a GPU is ill suited for this purpose. In order to fulfil the mobility and speed requirements for the sought application modifications need to be done to the GPU-trained network. Figure 6 shows a tree describing different modification possibilities. The goal for the MPSoC inference is to run in real-time for a low latency video stream. This means that some modifications need to be made to the network in order for it to fit on the MPSoC and perform as desired.

There are multiple ways of fitting a trained network to a desired hardware such as the Xilinx UltraScale+ MPSoC. Some of the biggest constraint on an MPSoC is the block random access memory (BRAM) size and memory buffer management. In order to overcome these challenges some standard methods can be used like pruning and weight quantification. There are more ways to modify the network in the implementation stage such as modifying the convolutional layers, ReLU layers and fully connected layers however the theory will mostly cover the first two methods (pruning and weight quantification) since those parameters can be changed without having to change architectural structures of the network. If said modifications are necessary chances are that there are other networks with better suited layer-design for the desired implementation.

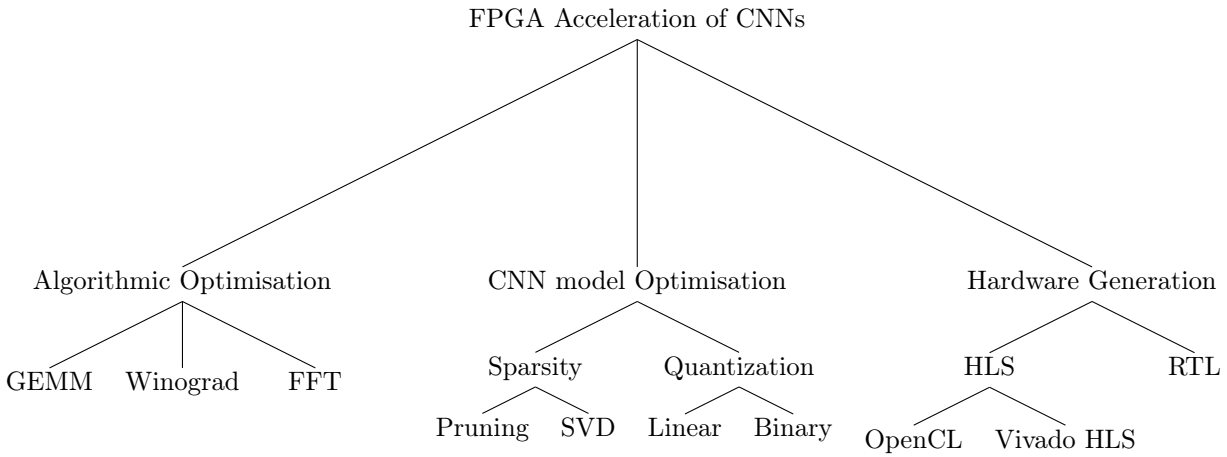

Figure 6: Different modification processes for FPGA Acceleration of CNNs

The algorithmic optimisation variants all focus on trimming the execution of convolutional layers and fully connected layers by reducing the number of arithmetic operations through the feature maps and kernels. They are more commonly used on CPUs and GPUs through a variety of frameworks for specific network designs. The most commonly known algorithmic optimisation is the fast fourier transform (FFT) which reduces the arithmetic complexity[15] in two dimensional convolutions.

2.1.3 Quantization

In order to reduce the load and size of a DCNN implementation on an FPGA one can change the arithmetic of the data. In general DCNN networks that are trained on GPUs and CPUs use single-precision floating point representation. Normally this is 32-bit floating points arranged according to IEEE754 standard. To reduce the computational load from these data points conversions to either lower resolution such as 16 bit floating point representation can be done or even better convert them to fixed points. To maximise the execution efficiency the fixed points can be sorted to predetermined discrete numbers. This means that data points are rounded to set data levels depending on what resolution the network needs as figure 7 shows. This is called quantization and the operation is typically applied on feature maps and convolutional weights.

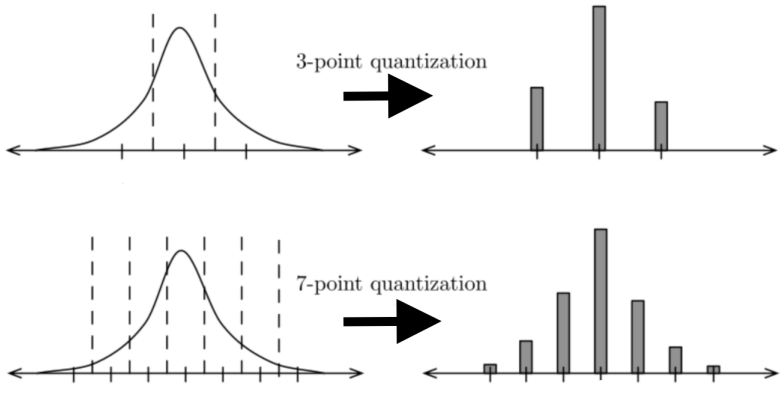

Figure 7: Quantization of two different resolutions

By quantizing the weights, less memory needs to be used for representing the numbers. Furthermore a 32-bit floating point representation requires broader bandwidth for transmission which is not desirable in FPGA implementations. One can change a trained network data representation from 32-bit floating point representation to another, more desirable, data representation through a variety of scripts in equally many different programming languages. However doing this conversion might cause overflow or underflow in the weight layers because a fixed data representation have a much lower dynamic range. Therefore it isn't advantageous to use the same bit width through out the whole network. Additionally deeper layers tend to need larger numerical range in order to compute.

To overcome said issue the usage of a dynamic fixed point is proved to be efficient. It works by having different scaling factors for different parts of the network. The best known way to acquire the optimal dynamic fixed points is by brute force and compare the accuracy and execution time of the network. This way we save real estate on the board without compromising the resolution in deep layers. These are called linear quantizations. Another way to quantify a DCNN is by binary quantization[16], this has however not been proven to uphold a high enough accuracy for this thesis application.

Representation	Range	Accurarcy
Floating Point 32-bit	$10^{-38} - 10^{38}$	$\frac{6}{10^6}$
Floating Point 16 bit	$6{\cdot}10^{-5} - 6{\cdot}10^{4}$	$\frac{5}{100}$
Integer 32-bit	0 - $2{\cdot}10^{9}$	$\frac{1}{2}$
Integer 16 bit	0 - $6{\cdot}10^{4}$	$\frac{1}{2}$
Integer 8 bit	0 - 127	$\frac{1}{2}$

Table 2: Unsigned Number representation, numerical range and accuracy

2.1.4 Pruning

Pruning is another way to ease the computational load and accelerate the network execution. A simple implementation of weight pruning cuts weights under a specified threshold effectively setting it to 0. This way the weights with higher impact on a convolutional layer stays unaltered while minor weights that most probably don't affect the over all performance and accuracy of the network gets reduced. In a DCNN this means a lot of weights can be pruned away, especially on a network with a broad backbone and narrow objective such as the RetinaNet architecture.

The RetinaNet architecture backbone consisting of a ResNet-50 can be pruned using this technique to around 3/10[17] of its original size. To optimise the pruning even further one can retrain the network with the pruned weights as input and iterate it over time. This way the unaffected weights get optimised for the "thinner" network and the mAP won't decrease. This is of course relative, too much pruning will decrease the performance of the network and will cause problems in the object detection. The most efficient way of using pruning to cut the size of a network and still keeping its accuracy is by pruning smaller pieces and retrain the network with the pruned weights iteratively.

2.2 Hardware platforms

In recent years FPGAs have grown on the computer processing market by proving to be a well suited solution for complex algorithm acceleration in demanding industrial and telecom applications. They have a superiority to CPUs and GPUs in terms of speed with regard to energy efficiency but falls short in comparison to the GPUs and CPUs when it comes to processing larger data and higher bit width.[13][14] What sets the FPGA apart from other processors is the ability to execute digital computations in parallel and asynchronous. Some FPGAs support multiple processors on one board where the flexibility when combining programmable logic (PL) with a soft processing cores like an ARM processor makes for quick data management and high speed signal processing.

Most FPGA boards today come with on board RAM memory and flash memory to store data for quick access and execution as well as SD-card sockets or similar peripherals for larger data storage. When using a real-time application such as an object detecting DCNN the transfer time from off-chip memory is not enough to suffice the need of the network. Therefore an FPGA is constrained by the on chip memory in object detection DCNN applications. FPGA are generally programmed by hardware descriptive languages such as Very High Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog. The FPGA integrated circuit (IC) block consists of multiple programmable logic cells containing digital signal processing (DSP) slices, digital flip-flops (FF) and lock-up table (LUT) which are simple 4 to 6 input 1 output multiplexers which in their turn can create simple logic gates such as AND/OR gates. They are designed similarly to a read only memory (ROM) as displayed in table 3 and table 4 which shows an example of two AND gates in gray code [12] a bit more intuitively. The FPGA assembles these logic elements by programmable connections between the logic blocks called routing channels which can be reprogrammed using software. The general FPGA structure is illustrated in figure 8.

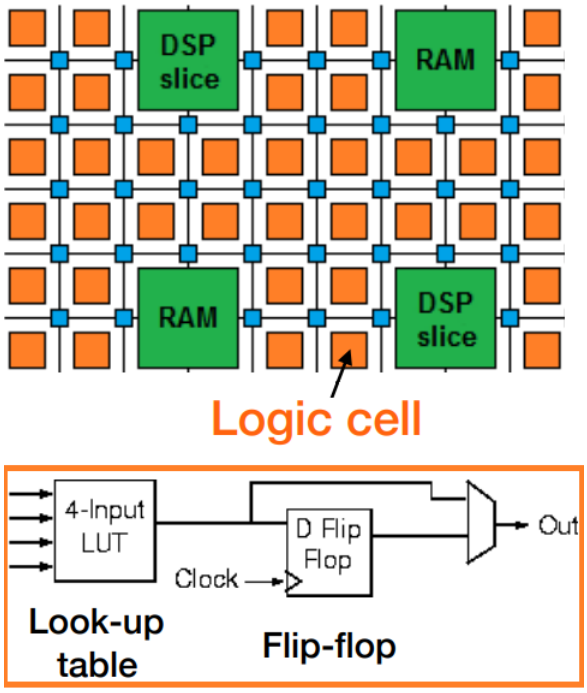

Figure 8: Diagram illustrating FPGA structure

I3	I2	I1	I0	Output State
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Table 3: LUT for an OR gate with inputs (I3 AND I2) OR (I1 AND I0) and output state

I3 I2 \ I1 I0	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	1	1	1	1
10	0	0	1	0

Table 4: Karnaugh diagram of an OR gate with inputs (I3 AND I2) OR (I1 AND I0) and output state shown in the lower right corner

For the object detection market and its constantly expanding community with more diverse and incompatible interfaces a re-programmable device is desirable. The computational workload of a DCNN inference is the result of an intensive use of the Multiply Accumulate (MAC) operation. A network designed on a GPU without size or complexity constraints can easily demand up to 60 Terra floating point operations per seconds (TFLOP/s) for real-time image processing which no single FPGA can handle. Xilinx claim that depending on data type the computational power of their products range from 7 to 22 TFLOP/s[14].

Most of these MACs occur in the convolutional layers of the network. As a consequence the convolutional layers are responsible, in a typical implementation, for more than 90 percent of the execution time during the inference. Therefore a network needs to be designed with constraints[15] determined by the FPGA hardware restrictions. There exist frameworks that help with design and mapping of the DCNN to make use of both the FPGA and the generic processor on the board as efficiently as a pre-programmed solution can be. Many claim their solution is the most efficient and optimal however the reality of FPGA design is that most boards and designs differ in specification therefore, in order to create an optimal solution, each implementation should have its own specification.

2.3 Hardware implementation frameworks

2.3.1 CHaiDNN

CHaiDNN[18] is a Xilinx Deep Neural Network library for acceleration of deep neural networks on MpSoCs. It serves the purpose of structuring the implementation of an existing neural network for a specific Xilinx FPGA through Xilinx Software-defined system on chip (SDSoC) IDE in c++ code. This way CHaiDNN bridges the gap between hardware and software developers by transposing software specified c++ code through SDSoC and autonomously creates a Vivado project and designs a complete logic placement, optimisation, routing and mapping giving an elf-file output for the MpSoC to read from a SD-card.

Xilinx provide a modifying script for a user to adjust an existing Caffe [19] based network prototxt and Caffemodel files to fit the input requirements of CHaiDNN. This way CHaiDNN can structure the network layout in their own way based on the Caffe models blueprint. In order for the network to align with the FPGA constraints Xilinx provide a quantizer which cuts the 32-bit float weights to 6bit fixed point integers. If supplied with well-crafted precision parameters the performance loss is minimal[18] for the network. CHaiDNN is also compatible with 8 bit integer representation although the provided example performance shows little improvement over the 6 bit. CHaiDNN supports both Xilinx Quantization as well as Dynamic fixed point quantization. The accuracy of CHaiDNN inference with Xilinx Quantizer on an MPSoC compared to Caffe inference on a GPU with Xilinx Quantizer is shown in table 5 with Top-1 being the percentage of correct class having highest score and top-5 being the percentage that the correct class being one of the 5 highest class scores.

	Accuracy			
	Caffe inference		ChaiDNN inference	
Network	6bit (Top-1/Top-5)	8bit (Top-1/Top-5)	6bit (Top-1/Top-5)	8bit (Top-1/Top-5)
GoogleNet	67.27 / 87.99	67.09 / 87.97	66.76 / 87.35	66.15 / 87.69
AlexNet	55.24 / 78.49	55.18 / 78.60	55.22 / 78.34	54.27 / 78.08
ResNet-50	68.80 / 89.04	73.40 / 91.47	69.89 / 89.61	72.97 / 91.11
VGG-SSD	77.28 (mAP)	78.06 (mAP)	76.62 (mAP)	69.12 (mAP)

Table 5: Caffe GPU inference accuracy compared to ChaiDNN inference accuracy

To process the heavier layers such as convolutional layers, ReLU layers and max pool layers CHaiDNN accelerates them on to the MPSoC's programmable logic segments. For this to work CHaiDNN has a set of predefined supported layers. The supported layers are listed in Table 6.

Supported Layers			
Convolution	BatchNorm	Power	Scale
Deconvolution	ReLU	Pooling(Max, Avg)	InnerProduct
Dropout	Softmax	Crop	Concat
Permute	Normalize(L2 Norm)	Argmax	Flatten
PriorBox	Reshape	NMS	Eltwise
CReLU	Depthwise Separable Convolution	Software Layer Plugin	Input/ Data
Dilated Convolution			

Table 6: Supported layers in CHaiDNN v2

To allow optimisation of a pre-defined network, CHaiDNN is compatible with users specifying hardware or software execution and whether to do this parallel or in sequence. These options requires a deeper understanding of how the network is structured and what layers demand which inputs at a given time. This is

due to CHaiDNN not supporting multiple inputs multiple outputs (MIMO) for networks and sub-graphs inside networks. By specifying which layers to accelerate on hardware and software an increase in inference of up towards 40 percent can be achieved. ChaiDNN v2 has a lot to offer with capability of handling used defined custom layers which makes it compatible with most network types. In comparison to version 1 Xilinx managed to up the giga operations per second (GOPS) by a factor four in version 2. They also added support for double-pumping the DSP slices which increase inference. To do so one must create a custom platform in vivado design suite and change the clock frequencies for the MPSoC of choice. This has a threshold as to where the increase in frequency gets greater than the FPGA can handle and a trade-off with the number of DSP slices has to be made.

Xilinx provide a range of example networks to run on their MPSoCs with a promise of performance increases around 2 to 5 times compared with a nVidia Jetson TC2. Xilinx provides the results for inference on MPSoC UltraScale + Zu9, some of which are shown in table 7 with performance measured in FPS and only the results evaluated on networks with fully connected (FC) layers. A complete comprehensive performance evaluation can be found at the Git-hub repository[20].

Device	Zu9							
Compute DSPs	128		245		512		1024	
Fmax (MHz)	300/600		300/600		300/600		250/500	
DSP Utilisation	442		574		838		1352	
LUT Utilisation	105811		114317		127949		161983	
FF Utilisation	111789		121363		136481		151616	
BRAM Utilisation	618		626		642		676	
Networks	Performance(FPS)							
	6-bit	8-bit	6-bit	8-bit	6-bit	8-bit	6-bit	8-bit
ResNet-50	17.28	9.31	29.89	17.15	46.25	29.41	60.57	41.99
VGG-SSD	1.69	1.01	2.47	1.63	3.21	2.33	3.76	2.92

Table 7: Performance of CHaiDNN v2 from Xilinx

Considering the FPGA constraints when executing on an MPSoC shows smaller networks are preferable for real-time low latency use. Evaluating some of the more popular small object detection networks on the market right now such as You only look once (YOLO) v2 and v3 and their respective tiny versions is possible through CHaiDNN API but because some layers in those networks are not directly supported in CHaiDNN but rather through customer defined software layers a quicker performance estimation is desirable. Xilinx have created a Quick performance evaluation script for their API which neglect certain layers specified by user in a network and calculates an estimation of network latency. This gives a network, which mostly consist of CHaiDNNs supported layers, a good ball park estimation to whether or not it's worth investing time in specifying the custom layers for the network or search for an alternative solution. Their results for evaluating YOLO networks is shown in table 8.

Networks	Input dimensions	Total latency(ms)	Total layers	Layers not supported
YOLO-v3	416x416	141.363556	107	YOLO(3),Route(4),Upsampling(2)
YOLO-v3-tiny	416x416	17.483529	24	YOLO(1),Upsampling(1)
YOLO-v2	416x416	57.257304	32	Reorg(1),Route(2),Region(1)
YOLO-v2-tiny	416x416	16.960856	16	Region(1)

Table 8: Quick performance evaluation on YOLO object detection networks

To conclude the CHaiDNN library it supplies the user with a complete package to take a pre-defined neural network and implement it on an MPSoC with multiple ways of accelerating the network and increase the inference efficiency of the system. It does require some user modifications but has next to complete documentation for any thinkable situation.

2.3.2 reVISION

reVISION[21] is a Xilinx product stack that includes a broad range of development resources for algorithm, platform and application development. It targets computer vision implementations and machine learning (ML) algorithms and just like CHaiDNN it operates in c++ for algorithm and application design in SDSoC. The stack supports the most common DCNN such as AlexNet, GoogLeNet, single shot detection (SSD) and fully convolutional network (FCN). Along side this it provides pre-defined and optimised implementations of the most common layers allowing users to structure custom neural networks with the stack. reVISION also include OpenCV [22] which consist of multiple acceleration-ready computer vision functions. Just like CHaiDNN it aims to bridge the gap between hardware and software developers by utilising c++ code in SDSoC. The workflow of a reVISION design is shown in figure 9.

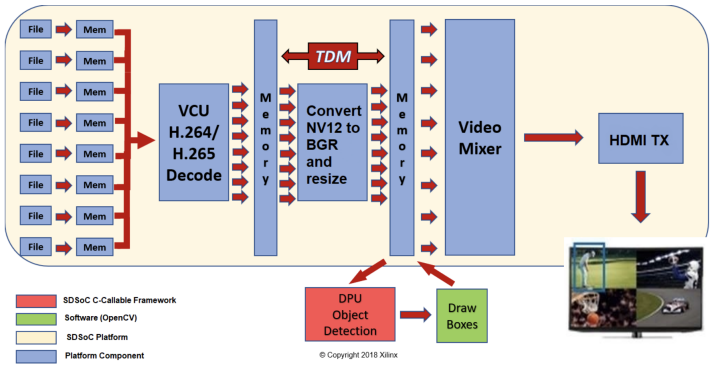

Figure 9: Block diagram of the reVISION 8-stream VCU + CNN design

Similarly to CHaiDNN, reVISION takes a pre-defined Caffe output file as input though reVISION does not include Xilinx quantizer which modified the network to better suit an MPSoC implementation. Instead it supports deep neural network development kit (DNNDK) tool from DeePhi which helps design the flow from Caffe model through quantization, compilation and deployment to Zynq MPSoCs as well as a few older platforms. According to Xilinx, reVISION achieves 6x better images/second/watt [21] for machine learning inference relative to embedded GPUs and typical SoCs and promises an order of magnitude higher frames/second/watt for computer vision processing. Additionally reVISION supports a variety of I/O for camera plugins and video output such as USB2/3 and HDMI. reVISION comes with a fully compatible stack to manage sensor input all the way to image output with its many potential workflows for a single sensor implementation shown in figure 10.
The SDSoC environment does not support performance estimation for OpenCV library nor c++ templates which makes reVISION compatible networks harder to evaluate pre-deployment. reVISION does not have any alternative to this like quick performance estimator in CHaiDNN. Trying to use the high level synthesis (HLS) performance estimator of hardware resources results in a indefinite communication process between the board and SDSoC IDE without any error messages causing the board to stall.

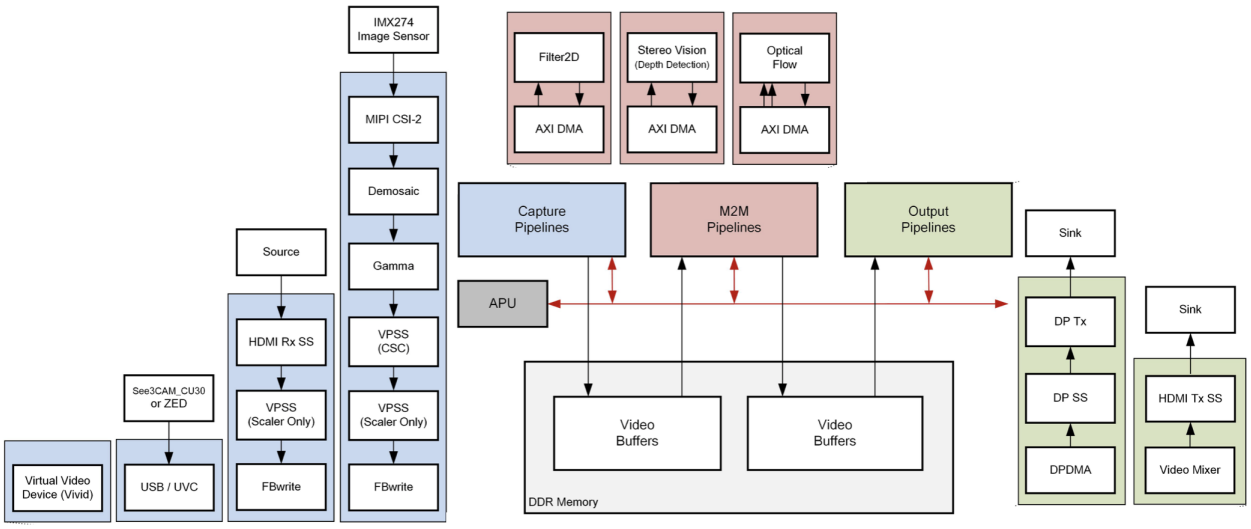

Figure 10: Block diagram of reVISION single sensor design

In conclusion reVISION promises a full stack for ML and computer vision compatible with start to end implementation of a c++ defined model of a pre-defined network or computer vision algorithm to MPSoC configuration with optimal platform usage. It provides a broad solution for accelerating computational heavy algorithms without demanding users to have any prior knowledge of hardware programming.

2.3.3 DNNWeaver

DNNWeaver[23] is an open-source specialised computing stack for accelerating deep neural networks. The first version of DnnWeaver was one of the pioneers in the market of open source neural network acceleration software for FPGA implementations. The DnnWeaver developers aim to lower the need for FPGA hardware high-level specification skills regularly used by FPGA developers in order to be more accessible for a broader crowd. Therefore it's a python based program with some Vivado dependencies. DnnWeaver is compatible with a wide range of convolutional layers which makes it applicable for many different convolutional network types and frameworks. DnnWeaver uses Verilog templates to structure the network for the FPGA. DnnWeaver requires a hardware setup in Vivado for the desired platform. They do provide a comprehensive guide as to how this hardware is to be set up however the users specific platform constraints is not available from DnnWeaver. This creates a wrapper which configures the MPSoC to be programmed with Verilog code sampled through python specifications by the user through Peripheral Component Interconnect Express (PCIe). This way the FPGA can be altered dynamically without rebooting the chip and without other complications which follow with booting from SD-Card or similar. The DnnWeaver workflow is shown in figure 11.

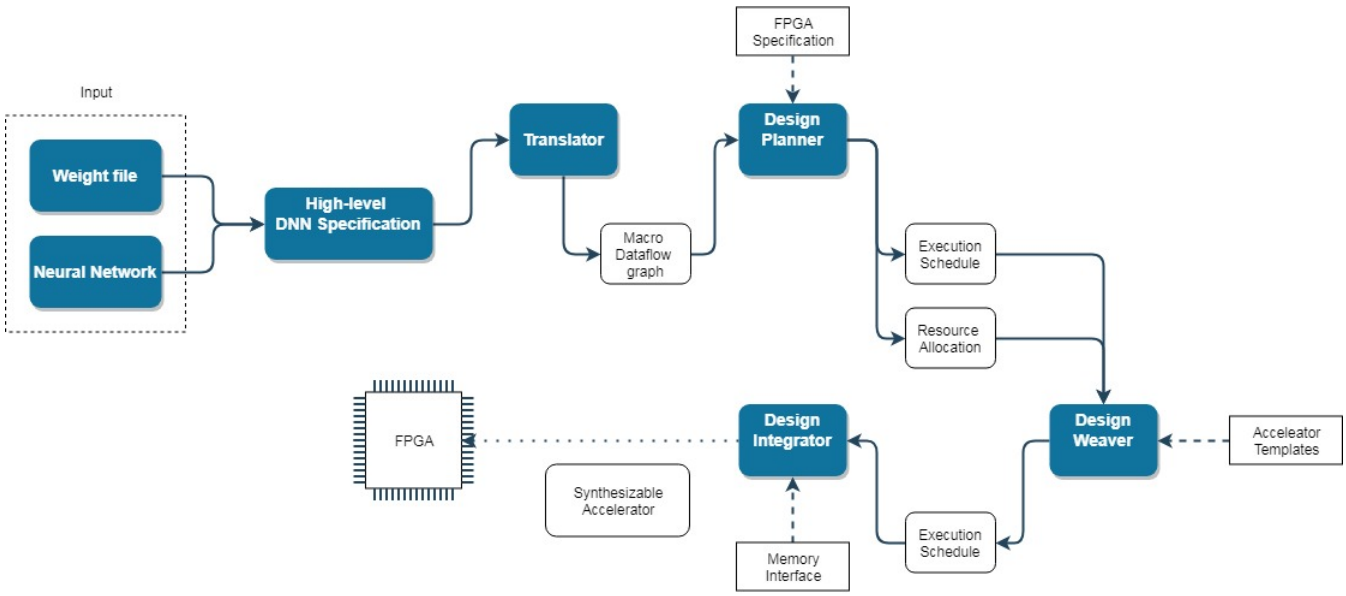

Figure 11: Workflow description of DnnWeaver

In the DnnWeaver workflow the translator serves the purpose of creating a macro-dataflow graph of the deep neural network using DnnWeaver's instruction set architecture where each node in the macro-dataflow graph represent a layer in the deep neural network. From this DnnWeaver generates a static execution schedule for the accelerator. Post scheduling the design planner optimises the organisation of the accelerator to maximise the performance. It also manages the data by slicing it to minimise the need of off-chip memory access. It takes in to consideration the constraints of the chip and its amount of LUTs, DSP and BRAM. Next in line is the design weaver which generates the actual acceleration core utilising the preset Verilog templates DnnWeaver supplies in accordance with the design planner. The final step is the design integrator. This component appends the memory interface code to the FPGA accelerator and then it's all sent to the FPGA through PCIe interface.

2.3.4 HLS4ML

HLS4ML is a framework developed by a team of researchers from CERN, Fermilab, Hawkeye360, MIT and University of Illinois at Chicago. It's a package defined in HLS for acceleration of ML algorithms on FPGAs designed for managing large bulks of data $\mathcal{O}(100$ TB/s) [24] coming from the CERN particle accelerator and analysing the data with high power efficiency and low latency. Their challenge is to maintain physics in increasingly complex collision environment. They've translated open-source ML package models to HLS that can be modified by a user for the desired ML acceleration. The package supports ML frameworks such as Keras/Tensorflow, PyTorch and scikit-learn. It supports Fully connected neural networks (FCNNs) with multi-layer perceptrons, standard convolutional neural networks (CNN), however the CNN still in beta testing. HLS4ML also has prototype support for recurrent neural networks (RNN), long short-term memory (LSTM) which is an artificial RNN and boosted decision trees.

HLS4ML requires the user to format and normalise the input network and does not support any pre-processing scripts or frameworks like the Xilinx frameworks. Nor does it supply a direct way to implement the system on an MPSoC. According to HLS4ML this gives the user more control over different aspects of the model. When defining a network the user has to set the precision parameters which in its turn is directly connected to the size and accuracy of the network. This makes for a flexible solution when it comes to net-

work size vs accuracy trade-offs and the user need to take the MPSoC constraints in to consideration when defining and training the network on a GPU. HLS4ML supports various levels of pipe-lining for re-usage of resources and parallel or serial model implementations. HLS4ML designs an intellectual property (IP) for the user to implement in a more complex design in Vivado or to be used to create a kernel for CPU co-processing. The workflow for HLS4ML is illustrated in figure 12.

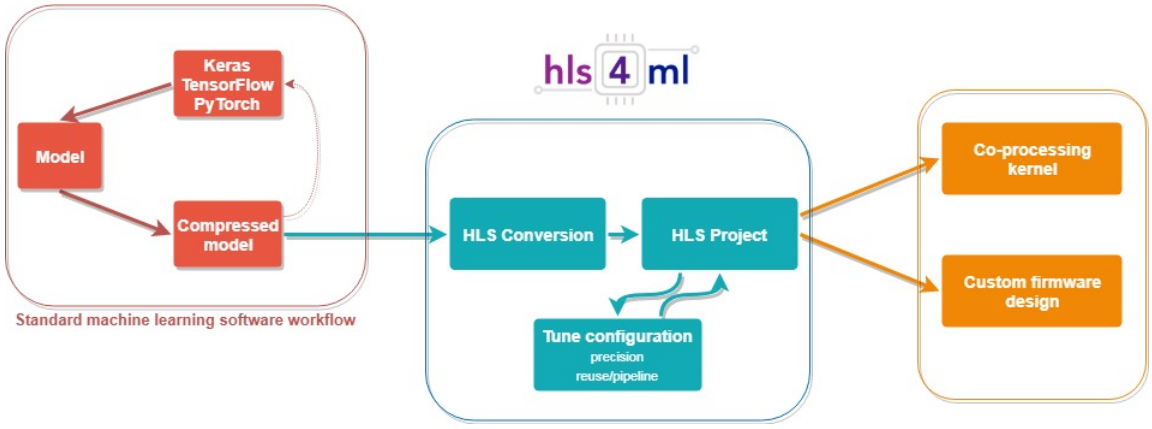

Figure 12: Workflow description of HLS4ML

By parallelisation of DPS slice usage one of the biggest bottlenecks in throughput is reduced, especially in bit widths $<$ 12bit width. HLS4ML reports in a seminar about L1 (level one triggering in ATLAS [A Toroidal LHC ApparatuS] detector) and data acquisition that the number of DSP slices required for executing a test network is reduced from around $6 \cdot 10^3$ to around $2 \cdot 10^3$ when reusing the DPS slices. This comes with the cost of an increase in latency with around 30 ns, which is almost 45 percent more than with no DPS re-usage and the system requires three times the number of clock cycles for a new inference input can be accepted by the network [24]. They also conclude that the power consumed by the FPGA increases with higher precision e.g. increased weight bit width but is lowered using DSP slices multiple times. In this case the power usage went down from around 4.25 watt to around 3.2 watt.

In conclusion HLS4ML delivers a package to create a quick prototype for algorithm acceleration on FPGA and MPSoC. The package is flexible but slim with quite high user input demands for a efficient implementation. No pre-processing such as quantizer for desired input network or hardware implementation included. Designed to handle large data for academic and research purposes demanding low latency in data management make HLS4Ml not well suited for image classification or object detection networks such as this thesis needs.

2.4 Deep learning model converters

The different MPSoC and FPGA implementation frameworks all support a few specific neural network frameworks. Because of this, the chosen implementation framework must be compatible with the chosen neural network and all its layers (except for CHaiDNNs case in which a user specified custom-layer is supported). Even in the CHaiDNN case it's most likely better in terms of accuracy and compatibility to choose a supported framework. If the neural network is not compatible with the framework it's plausible to assume it would be quicker and more efficient to train a new model in a supported framework with the intended data set and implement it in the implementation framework even though this might vary from case to case. If the network of choice for some reason can't be changed it needs to be converted or translated

to a supported framework. The demand for these type of conversion grows as the community for neural network grows. The number of frameworks expands increasingly fast making it harder for the hardware implementation frameworks to keep up. Table 9 shows the best performing networks of the ImageNet large scale visual recognition challenge (ILSVRC) from 2010 to 2015 and their top-5 performance on image classification trained and evaluated on ImageNet[9]. In 2012 AlexNet[5] came as the first DCNN and since then DCNNs have dominated the contest.

Year	Network	Top-5 accuracy (%)
2010	NEC-UIUC	71.8
2011	XRCE	74.3
2012	AlexNet	83.6
2013	ZF	88.3
2014	VGG	92.7
2015	GoogLeNet	93.3
	Human performance	95
2016	ResNet	96.4
2017	GoogLeNet-v4	96.9

Table 9: Winning networks on ILSVRC the last 9 years

To overcome said troubles deep learning model converters have been created by the open-source community. There exists quite a few converters that together can handle most of the popular network frameworks but since they're open-source community created, the reliability varies a lot from converter to converter. No framework works the same way as another, the only thing most of them have in common is the choice of python as their programming language. This way they can cover TensorFlow, Caffe and PyTorch which all can be written in or handled by python.

2.4.1 Keras2Caffe

Keras2Caffe seems to be the obvious choice for a network modelled in Keras to be converted to Caffe. This Git-Hub repository based converter is compatible and tested with version 1.0 of Caffe which is the latest update of Caffe 1, Keras version 2.1.5 (newest version may 2019 is 2.2.4) and TensorFlow version 1.4.1 (latest version is 1.13). This makes it useful for a few neural network frameworks. Table 10 lists the model compatibility of Keras2Caffe.

Supported Network Models		
Inception V3	Inception V4	Xception V1
SqueezeNet	VGG16	MobileNet(Still not full support)

Table 10: Supported network models in Keras2Caffe

2.4.2 MMdnn

Model Management deep neural network (MMdnn) is a comprehensive cross-framework converter with additional capability to visualise and diagnose deep neural network models. It's one of the most comprehensive converters today with support for most existing network models and layers. Operating in python with a pip package to ease installation and enhance user experience makes the converter useful in both industrial implementations as well as academic. MMdnn converts a model from one framework to another in multiple steps to be compatible with converting between multiple frameworks with one structure. Most frameworks have their own network structure definition and file format which means decoding and interpreting must be adaptive to fit the model. Therefore MMdnn converter uses an intermediate representation for all the networks to convert to and unpack from. In order to interpret the output of some networks to get a desired file type a second conversion is some times necessary and included in the MMdnn package. The frameworks MMdnn support is listed in figure 13.

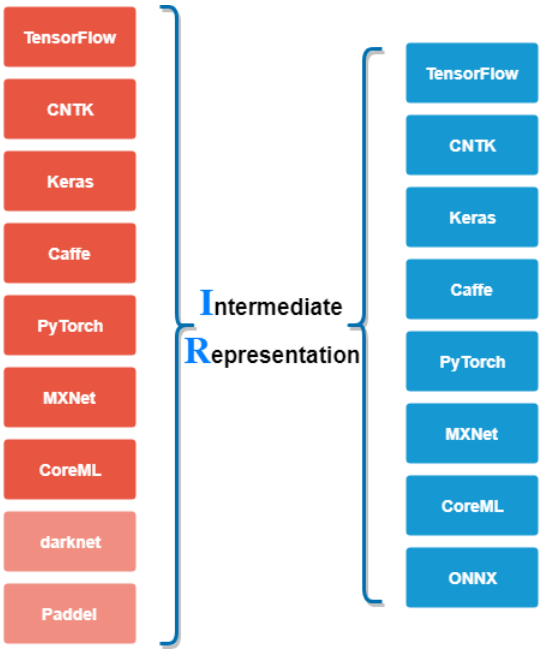

Figure 13: MMdnn supported frameworks

With such a broad framework support a lot of network models compatibility is required. MMdnn mostly supports image classification network models and is currently developing support for face detection, semantic segmentation, image style transfer, object detection and RNN. Even though the support is limited to image classification MMdnn fully supports 13 model conversions between the listed frameworks in figure 13 and additionally supports voc FCN conversions between TensorFlow and CNTK and YOLO 3 conversions between Keras and CNTK. Table 11 lists the models supported by MMdnn.

Supported Models				
VGG 19	Inception V1	Inception V3	Inception V4	NasNext
ResNet V1	ResNet V2	ResNext	MobileNet V1	MobileNet V2
Xception		SqueezeNet	DenseNet	

Table 11: Models supported by MMDnn

3 Implementation

Implementing and accelerating an existing deep convolutional neural network, trained and evaluated on a GPU, on a state of the art MPSoC demands a lot of work with quite some debugging involved. Even though most of the frameworks and Integrated development environments (IDEs) available for implementation of ML algorithms and networks such as the one this thesis intends to cover claim to have a complete and easy to use end-to-end solution this is rarely the case. The network needs pre-processing before being a suitable fit for hardware implementation and acceleration. Here one must quantize and prune the network so that it fits the intended hardware design constraints without compromising the networks desired performance. There exists quantizers which converts 32-bit floating point representation to any desired bit width. Most research[25] marks a 6 bit fixed point representation as the start of the saturation point of a curve where bit width and accuracy are plotted against each other. If there is room for a larger bit width and higher accuracy is desired most quantizers supports larger bit widths as well all though the increase in accuracy almost always comes with throughput frequency decrease.

If the framework for implementing the network on the MPSoC is not compatible with the network model one must convert the model. Here it has to be considered what structure the network has and layer specifications since compatibility vary a lot between different frameworks and converters. Having a neural network that's compatible with the implementation framework from the beginning eases the hardware implementation step quite a bit. Depending on what framework is chosen for designing the hardware the user has to do more or less of the actual fitting and routing. Some implementations are therefore more well suited than others. For instance, working on a Xilinx UltraScale+ MPSoC one would prefer to use one of the available Xilinx frameworks. This due to them only supporting their own SoCs giving well suited solution with documentation regarding the constraints as well as the optimal acceleration possibilities. If one were to implement on smaller FPGA or SoC from another manufacturer another framework like DNNWeaver v1 might be a better solution. If communication through PCIe is desired DNNWeaver V2 is a good start.

In this thesis the implementation had a predetermined board (ZCC102 evaluation board) and a Keras RetinaNet object detection network which made CHaiDNN or reVISION seem like the best framework considering the hardware and HLS4ML considering software. Having the main objective finding frameworks for implementation of neural networks on Xilinx MPSoCs main focus has been on CHaiDNN. This based on CHaiDNNs complete and thorough documentation and well described workflow which clears the fog around efficient implementation and its many examples for guidance and evaluation. Nevertheless the whole chain is to be tested. The implementation of a neural network on an MPSoC can be divided in to six parts with an overview illustrated in figure 14 and detailed steps in the following subsections.

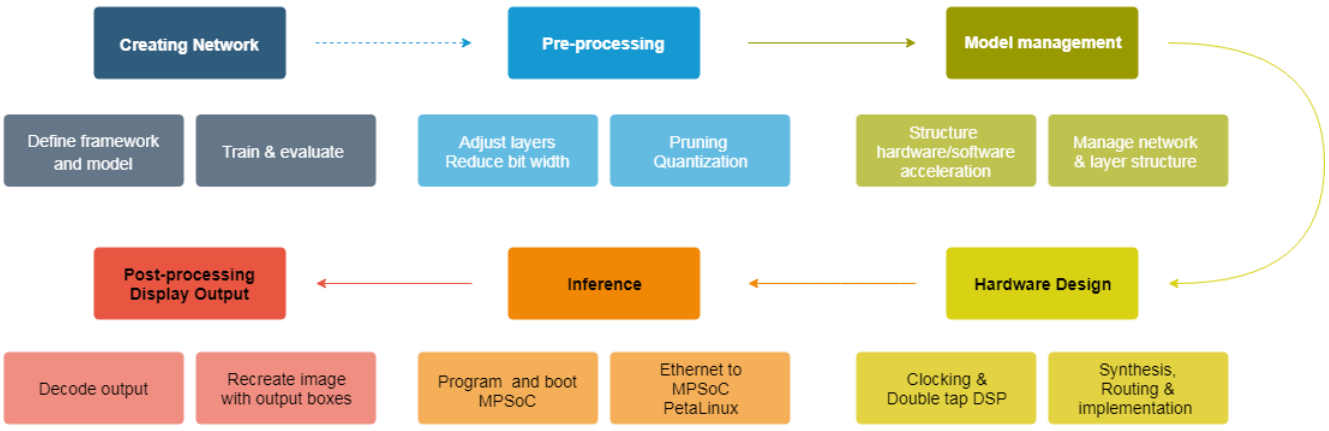

Figure 14: Universal workflow of implementation

The steps this thesis took to accomplish the successfully implemented DCNN is illustrated in Figure 15.

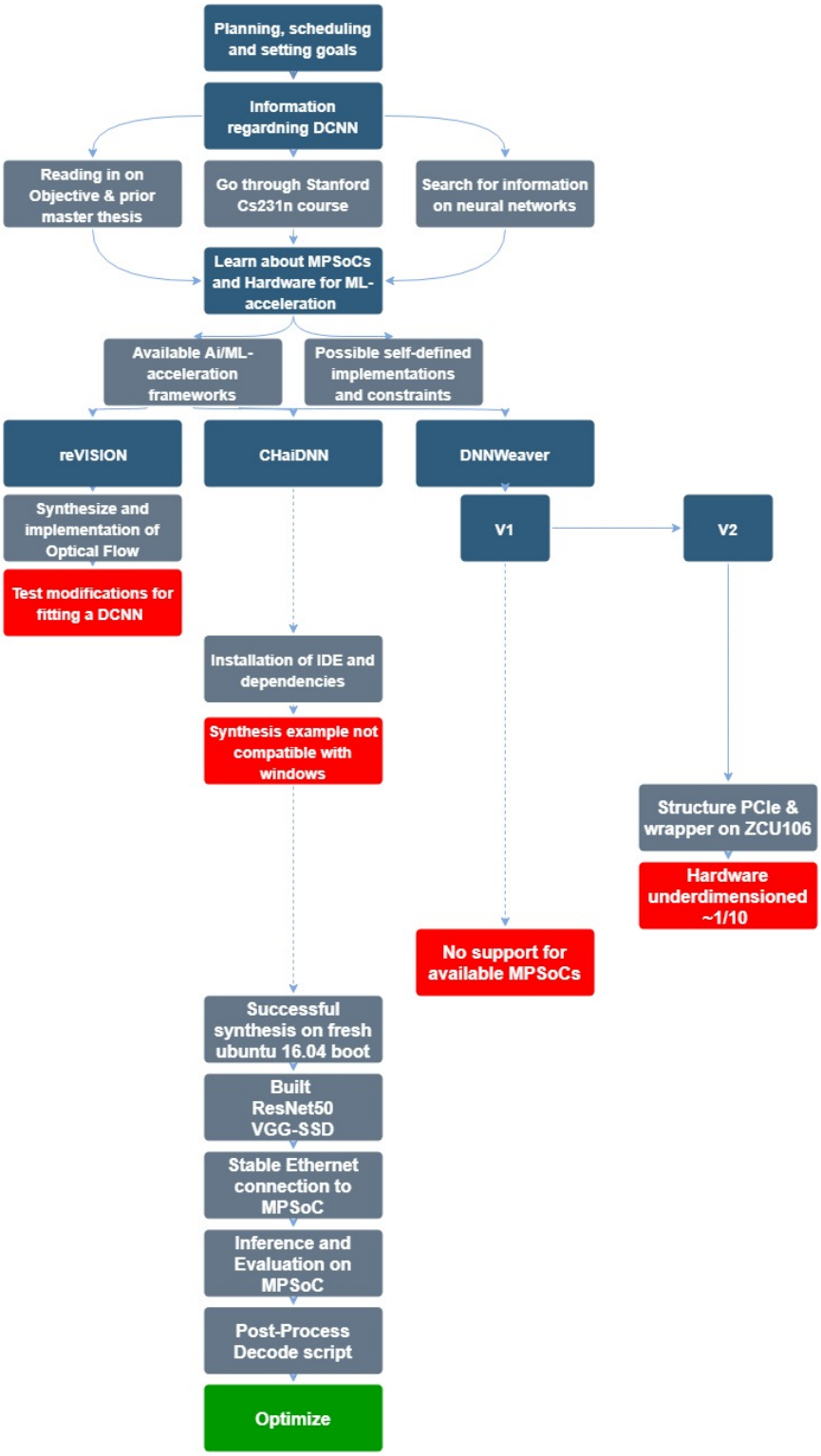

Figure 15: Workflow of successful MPSoC implementation

3.1 From Keras model to Caffe model

Having a set network model to work with (RetinaNet) conversion to Caffe was inevitable since CHaiDNN was the prime implementation framework of choice. There exists three converters for this purpose however two of those are practically the same. Initially conversion using Keras2Caffe was tried but the script got stuck on continuous error messages and failed to convert the network due to lack of support for RetinaNet and more specifically a couple of layers like ReLU, regress boxes, non maximum suppression (NMS) which is used in their detection filter and more. The lack of documentation from Keras2Caffe made this a dead end for this thesis.

Instead of Keras2Caffe, MMdnn looks more promising with its many supported frameworks and models. Even though it doesn't claim any explicate support for RetinaNet in the list of supported frameworks/models most of the layers used in RetinaNet occurs in other networks that MMdnn supports. Being a python pip package, the converter is easy to install and set up however the virtual environment installed supported only one Keras site package which caused a couple of errors with input types. This could have been solved by re-installing the virtual environment but in order to not loose its content it was instead solved by modifying the save script for python in Keras engine. It required .h5 file for the weight and .json file for the architecture but the network supplied only had the .h5 file with both the architecture and weights included. The .h5 file is a binary file which requires decoding before extraction. Instead of trying to generate a new .json file, conversion of a smaller RetinaNet with separate files for architecture and weights was used as input for the converter. Again trouble with layer compatibility and several error messages with no direct solution hindered the conversion. The conclusion is that MMdnn is at this moment not compatible with converting RetinaNet to Caffe and quick debug search for others issues[26] confirms this.

In order to compare a network trained on local GPU with the same network implemented on the MPSoC a minor testing net detecting cats and dogs in pictures was chosen. This was based on ResNet50 which is supported by MMdnn and convertible from Keras to Caffe. MMdnn first converts the Keras model to a intermediate representation giving a three file output (.pb , .npy and .py). The output is now compatible to be converted with MMdnn Caffe model converter giving a output of two files (.py and .npy). In order to pass this to Xilinx quantizer one last conversion needs to be done. MMdnn converts the two output files to a .caffemodel and .prototxt file with architecture and weights defined.

3.2 Caffe to SDSoC

Taking a Caffe model straight from GPU evaluation won't be a suitable network for implementation on an MPSoC. In order for a model to fit the MPSoC quantization and conversion from 32-bit floating point representation to 6 bit or 8 bit is necessary. This can be done in various ways like Python and MatLab however since CHaiDNN supplies a quantizer that not only quantizes the input but also modifies it to fit CHaiDNN implementation better which makes it a more suitable quantizer. In order to accelerate a layer on hardware the multiplication matrices in the layer need to be optimised both in size and throughput to the next layer. In a general deep convolutional neural network the convolutional layers can account for more than 90 percent of the computations which is why these layers are crucial to accelerate with hardware.

In CHaiDNN quantizer, the output network is designed in a way to take advantage of the matrix multiplying capabilities of the FPGA to its top potential. Using the converted network detecting cats and dogs as input the expected output would be quantized and slimmed to fit an implementation based on an example net with modifications according to Xilinx specification. At this stage the quantizer was set to quantize the network to 6-bit fixed point precision and with a total size of around 10 Mb this would result in an output just below 2 Mb. Unfortunately in the quantization process an error occurs in a binary bit-file with single output message "KeyError: 'data' ". The quantizer can read the input file and its 15 layers including Conv2D, Activation, MaxPooling2D, Flatten, Dense and Dropout layers. It updates the layers as expected and processes the weights from what can be read through terminal messages.

In order to get results that are comparable between GPU execution and MPSoC a ResNet50 model trained in caffe was used instead. It was quantized in Xilinx quantizer with no problem to both 6 and 8 bits width in separate iterations. The last key to generate a bit file is to get the now trained and quantized network through SDSoC. This is done by defining a c++ wrapper for integration and defining of processors and logic usage of the MPSoC. In this wrapper the resizing of images is set as well as the directory paths to weight files, caffe model files and input images. In this implementation any input image will be resized to 224x224 pixels. The wrapper needs no specification of the full network however it needs to know the initial layer and end layers of the network the user wants batched into graphs. These sub-graphs is how SDSoC want to represent a combination of layers in order to structure the network and accelerate critical layers. It proceeds to specify the normalisation parameters if the network is to be normalised. This wrapper creates input and output buffers, allocates extra output buffers for ping-pong for graphs and calculates memory size for output buffers. Last the wrapper proceeds to specify the arguments to pass to thread routines before starting specifying the execution loop.

After looping through the threads the total time is collected and an average FPS is printed. Now the only thing remaining is for the wrapper to specify the unpacking of the output and saving it to a user specified directory on the SD-card mounted on the MPSoC and releasing the memory. It writes the output in a txt file specifying number of boxes found and Intersection over union (IoU). Then it specifies each box with label, estimate precision, x-min coordinate, y-min coordinate, x-max coordinate and y-max coordinate. The coordinates are a scaled value between 0 and 1 for a user to be able to apply it on any desired resolution of the input image. This way the output image resolution can be specified after inference.

3.3 SDSoC to MPSoC with CHaiDNN

SDSoC can now, with fully defined instruction files and board specification, compile the c++ code and create a Vivado project for software to hardware translation with HLS. Then it creates all necessary IP-blocks, synthesises them and runs Vivado implementation with all that comes in a regular Vivado project build. The output files include a boot image for PetaLinux and elf file for mapping and running the network on the MPSoC. Now the user must load the output files to a SD-Card together with maps containing OpenCV, cblas, protobuf, lib, test images and the input model files (caffemodel and prototxt) and mount the SD-Card on the MPSoC. This user input is only necessary once if the model is complete. Booting up the MPSoC will load a slimmed PetaLinux to the board RAM for communication between a desktop and the MPSoC and to control, modify and deploy the network.

3.3.1 PetaLinux and Inference

In order for the MPSoC's many different components to interact with each other they must be programmed to do so. This can be done multiple ways but most common is to use either USB Joint test action group (JTAG), PCIe, from SD-Card or in some cases like this booting an operating system (OS) from a SD-card to enable communication via Ethernet and control the inference via a terminal command on a desktop.

PetaLinux[27] is an embedded Linux development solution designed for Xilinx Zynq Ulstrascale + MPSoCs and a few other pure FPGA chips. It offers a complete package including command line interface, device driver library generators, bootable system image builder, a variety of tools and support for Xilinx system debugger. These properties give the user ability to customise interface for the boot loader, Linux kernel and general Linux applications. PetaLinux enables developers to monitor deployment and inference of applications in real-time and modifications can be made without having to reboot the system or device. The OS also include configuration tools which are compatible with Xilinx hardware development tools which gives build instructions to autonomously build and deploy drivers for Xilinx embedded IP cores according to user specified addresses. This implementations tactic is quite practical for implementation such as this for sending files to and from the MPSoC without the hassle of having to restart the whole process each iteration.

In order to open a connection between a desktop and an MPSoC some protocol needs to be set. In this case secure shell (SSH) was suitable. This protocol requires the host desktop to establish the connection which places the MPSoC ip address in ssh-key list of known hosts. SSH saves approved connections in a list "known hosts" to ease the connection process next time a request is sent. This complicates things for an MPSoC with a boot image on the RAM because each time it's restarted the image will have to boot from scratch giving it a new "identity" (MAC-address and more) which in its turn are not compatible with the list of saved hosts on the desktop resulting in a requirement to remove the saved identity and requesting a new connection to be established. Therefore usage of this system is eased if the board has its UART connected to the same host desktop. When booting PetaLinux startup routine commands will be sent through the MPSoC's UART channel 0. Amongst the information passing through the UART at boot up Ethernet connectivity and assigned ip address will show along with other useful information to confirm successful boot.

After PetaLinux has booted to the on board RAM library paths for OpenCV, Protobuf and Cblas need to be specified. This can be done through terminal commands or by defining a initial bash script to run after the boot up. In this thesis a simple bash script with library paths and deployment commands for easy setup and inference of the two networks was defined. Modifying the bash script is possible through vi editor directly on the MPSoC thorough SSH or simply on the desktop before the SD-Card is mounted on the MPSoC. Post inference the output is saved in a directory on the SD-Card. To process it and display it must be transferred to the desktop, once again either by SSH or manually changing the SD-Cards location.

3.3.2 Custom Platform and Double-pumped DSP

Having a goal to make object detection network execute in real-time all possibilities to increase throughput needs to be used. One way to increase throughput for network inference implemented with CHaiDNN is to "double-pump" the DSP slices. As the name suggest, double-pumping the DSPs is done by doubling the frequency the DPSs operate on in comparison to the rest of the logic. The MPSoCs used in this thesis provide the clocks needed however to get higher frequencies synchronous the clock wizard IP must be modified. By changing the output clocks and wiring it up according to figure 16 the throughput should increase.

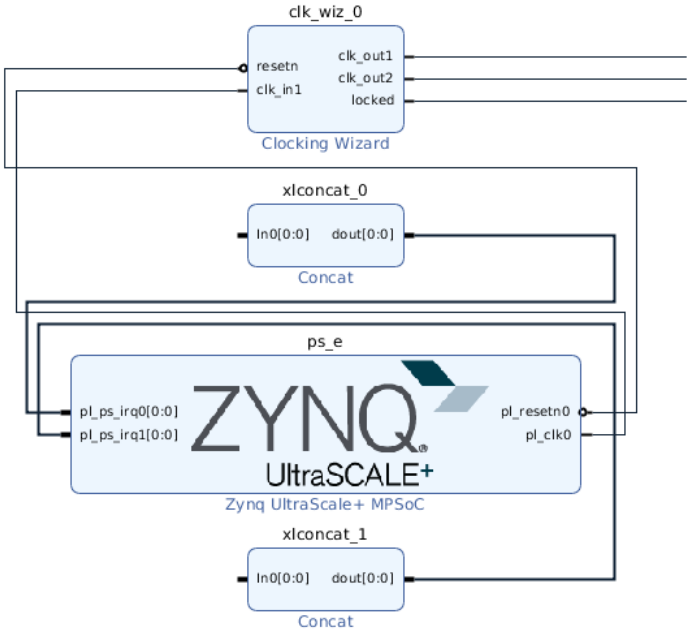

Figure 16: Custom platform for double tap DPS slices

3.4 Implementing DnnWeaver

Implementing DnnWeaver seemed promising for the object detecting neural network provided and the many papers citing it when searching for efficient DCNN implementations. It seemed as a lot of easy tests for benchmark and quick deployments of various networks had been done with DnnWeaver with competitive results and inference time. DnnWeaver is python based and in order to use it user must set up a Vivado project defining PCIe communication and other critical communication busses DnnWeaver needs. DnnWeaver has released two version and both were set up for evaluation, though V2 more thoroughly.

3.4.1 DnnWeaver V1

DnnWeaver V1 is python based and requires a stable PCIe communication with the MPSoC prior to network deployment and inference. Therefore setting up the hardware with Vivado tools is the first step in the implementation of DnnWeaver V1. It requires Vivado 16.2 to operate and the two MPSoCs available (Xilinx ZCU102 and ZCU106) are not supported in Vivado version 16.2. Vivado design suite can update designs and Vivado-compatible IPs from one version to a newer though not all functions are the same in two versions which can cause compilation errors. In the case of DnnWeaver v1, Vivado 16.2 does not fully support ZCU102 or ZCU106 which made the transition to 18.2 inevitable. Having the issue of card compatibility with frameworks and updating the IPs in Vivado block design made DnnWeaver V1 not a suitable framework to focus on initially.

3.4.2 DnnWeaver V2

DnnWeaver V2 is an update to DnnWeaver V1 compatible with Vivado 18.2 which supports both Xilinx ZCU102 and ZCU106 evaluation boards which are the MPSoCs available for this thesis. Having an implementation framework supporting the hardware is quite essential if modifications to the existing implementation framework is not of interest or might in some cases not be possible. DnnWeaver V2 requires only software that's either open source or Xilinx software and is thereby suitable for evaluation on this thesis. It only supplies examples of YOLO neural network implementation which is not the same as the DCNN architecture this thesis has as input however the deployment and benchmarks of DnnWeaver V2 with YOLO2 seemed promising and would be an interesting subject to review for a future EFVS solution, more about this in the discussion chapter 5.

DnnWeaver V2 works with python defined scripts to structure and deploy neural network inference on the MPSoC. In order for python scrips to program the device and its components, a communication bus is needed. DnnWeaver V2 uses PCIe just like DnnWeaver V1 which is not a compatible bus for programming the ZCU102 board and trying to set it up with hardware PCIe interconnects did not work. Therefore a Xilinx Zynq UltraScale+ MPSoC 106 (ZCU106) was used. For a desktop to program the MPSoC through PCIe with python scripts it must find it amongst known devices, which in ubuntu (used in this thesis) should be under "/dev/xdma_0". For this PCIe connection to be established a Vivado project must be designed, including PCIe IP, AXI interconnects, DDR4 and a wrapper handling DnnWeaver commands and manages the hardware accelerated algorithms.

DnnWeaver V2 supplies a design description to create a synthesizable Vivado project for enabling PCIe communication and DnnWeaver V2 inference compatibility. This guide supplied by DnnWeaver V2 is quite profound and easy to follow. The images illustrating the block design are simplified and the block automation in Vivado creates several extra IPs which are not defined in the guide. Running the whole guide through yields a block design illustrated in figure 17.

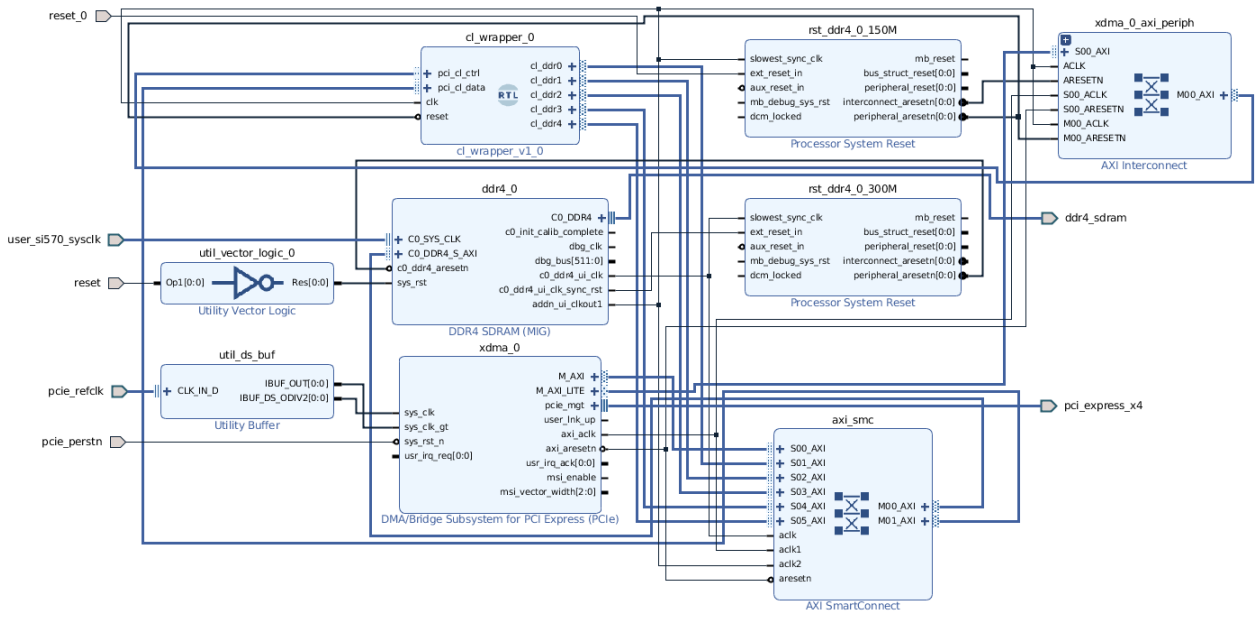

Figure 17: Block design of DnnWeaver V2

This design looks promising and synthesising it output a few warnings but nothing unusual for a Vivado project. Implementing it is another story though. The cl_wrapper IP block gives quite a few variables to change however the documentation of these variables are non existent and they lack descriptive names. Table 12 lists the variables configurable by a user.

Customisable variables			
Acc Width	Array M	Array N	Axi Addr Width
Axi Burst Width	Axi Id Width	Bbuf Addr Width	Bbuf Axi Data Width
Bbuf Capacity Bits	Bbuf Wstrb W	Bias Width	Buf Type W
Ctrl Addr Width	Ctrl Data Width	Ctrl Wstrb Width	Data Width
Ibuf Addr Width	Ibuf Axi Data Width	Ibuf Capacity Bits	Ibuf Wstrb W
Ififo Addr W	Inst Addr W	Inst Addr Width	Inst Burst Width
Inst Data Width	Inst W	Inst Wstrb Width	Loop Id W
Num Tags	Obuf Addr Width	Obuf Axi Data Width	Obuf Capacity Bits
Obuf Wstrb W	Op Code W	Op Spec W	Pu Axi Data Width
Pu Wstrb W	Wbuf Addr Width	Wbuf Axi Data Width	Wbuf Capacity Bits
		Wbuf Wstrb W	

Table 12: Customisable Variables

With no modifications done to Cl_wrapper the implementation in Vivado gets stuck with over dimensioned hardware usage. DnnWeaver V2 is demonstrated and evaluated on Xilinx Kintex UltraScale+ (KCU1500) which comes with one of the largest FPGAs from Xilinx with 2.4 times as many system logic cells, 2.2 times more DSP slices, 2.36 times on chip block ram and more than 2.5 times more input/output pins in the FPGA in comparison to ZCU102. The ZCU106 which was the only PCIe compatible board available has even less DSP slices and System logic cells than ZCU102 and though it has a few more input/output pins and around 6 MB of BRAM more it's under dimensioned for a program designed for the KCU1500. Modifying the parameters to fit the ZCU106 in not that straight forward and wouldn't complete Vivado Implementation which led to DnnWeaver V2 being put on ice while CHaiDNN was further evaluated.

3.5 Implement reVISION

reVISION operates similarly to CHaiDNN through Xilinx IDE SDSoC. The main difference between reVISION and CHaiDNN is that the reVISION stack mainly focuses on computer vision whilst CHaiDNN is purely for accelerating deep neural networks. reVISION supports many different solutions for implementing computer vision and incorporating them with a few peripherals like three cameras, a few sensors and video stream outputs however reVISION does not supply any examples or guides on how to handle a deep convolutional neural network or the many different layers and complex architecture of DCNNs. Implementing the optical flow example in reVISION is no issue with hardly any complications though not really close to what this thesis is to achieve. Therefore using Deephi and its ML algorithm accelerators is necessary.

One way to create an evaluation example of a network like ResNet50 one could use the integrated DeePhi machine learning IPs and libraries which should support most of the standard networks. DeePhi has a couple of reference models to make network deployment setup easier. Being fully dependant on DeePhi is an issue since most of the documentation is flawed from faulty translations and the web page for DeePhi quite often blocks access with the following message:

"A Kindly Reminder
The website is unable to access for the moment
Sorry, the website is unable to access for the moment. According to the filing requirements of China's Ministry of Industry and Information Technology (MIIT), the website is accessible only if the ICP information is accurate and the ICP license is filed.Please contact the person in charge of the website for assisstance."

Having issues with documentation and implementation examples there were really no reason to keep pursuit the implementation of a DCNN with reVISION stack and the work followed the workflow described in figure 15.

3.6 Python decode/post-process

After successful inference the output of a neural network can be processed differently and illustrated in a variety of ways. CHaiDNN gives the output from inference in a short version directly through a terminal emulator such as minicom through UART as well as a txt file with the complete output. In the case of ResNet50 that was tested the output is a text file with all the 1000 classes and the score of the network for each class (between 0 and 1). Most networks supply their own decoding scripts and runs it with the rest of the inference however since CHaiDNN modifies the input network to fit its own deployment a script for decoding CHaiDNNs output has to be written, or modified from a pre defined code, by the user. In this thesis the code for handling the output from VGG-SSD network inference was modified from Intel's original plot detection tool found in their github repository[19].

3.6.1 Visualising output from Image Classification network

One way of displaying the output is by saving all the ImageNet classes in an array or similar then search the output for the highest score and set the label for that match saving it to a new list with the top 5 matches. This can be done in most languages like python and MatLab and similar with rather few lines of code and with no advanced coding knowledge required. Usually the top 5 results is sufficient and in this thesis the command line prints from the network was enough to evaluate the network output.

3.6.2 Visualising output from Object Detection network

Being an object detection network the inference will result in an output file consisting of boxes framing the detected objects. In CHaiDNN these boxes are independent of the input image size and thereby just scaled numbers between 0 and 1. The output text file starts with two rows specifying total number of boxes and, if supplied with a data set specification, the average IoU of the inference followed by information about the boxes. The box information comes in the following order: label (0-20 from the VOC2012 dataset), score, x-min of box, y-min of box, x-max of box and y-max of the box. Then an additional IoU line for the specific box which in the case of testing images without proper specifications however in this implementation it got tossed. Table 13 shows the first 16 rows of an output text file running VGG-SSD on an image of two SAAB gripen aeroplanes shown in figure 20.

The top 2 highest scored boxes output from VGG-SSD300
182.000000
0.000000
1.000000
0.995613
0.274735
0.015819
0.760868
0.359273
0.000000
1.000000
0.991003
0.100976
0.375606
0.934958
0.994799
0.000000

Table 13: Output from 6 bit VGG-SSD inference

In order to parse the output and generate a image a python script from Caffe could be used. Though it required some modifications the code for connecting the label-map to output class and method to draw the boxes in a plot could be used straight of. The main difference in Caffe based network output and CHaiDNN is the the layout of the box descriptions. Using standard libraries like numpy and matplotlib, multiple plots of the different boxes is created. It's then scaled to the input image size read by the script using PIL library. In the end a new image is created with the input image as background and all boxes with a score higher than user specified threshold placed as a layer on top. For visual reference, images processed with this plot detection script are located under section 4 performance.

In order to create a real-time image flow some method to incorporate the python script with a output image display via HDMI or similar is necessary. This can be done either by looping back the image to the FPGA or utilising another interface and PetaLinux on the MPSoC.

4 Performance

Evaluating the performance of a DCNN inference on an MPSoC can be done in many different ways. Researching the options to evaluate a network inference and the most common comparison numbers indicates the exponential development of neural networks. Just a few years ago the most common key evaluation point was the top 5 accuracy of a image classification network. This is the number of times the correct class was in the top 5 score after inference. Later that number got so close to 1 for not only the state of the art networks but even the smaller and less known networks that it didn't really suffice as comparison. This is when the top 1 accuracy got more relevant which, as its name suggests, is the percentage of the network classifying the highest score to the correct class.

After a network is trained and evaluated on a proper data set the average precision score doesn't change except if the network is modified. Therefore a network evaluated on a GPU and then pruned and quantized won't perform identically as the original network. As this report has shown the accuracy of a object detection network is not expected to drop significantly with a bit width above 6 bits if quantized correctly although some decrease in accuracy can be expected. Even though a hardware accelerator might have better suited components to compute MACs it's hard to challenge a GPU in inference since the resources are not as limited on a GPU.

In order to get a comprehensive comparison between GPU inference and MPSoC inference the same neural network model and data set need to be used. To get a fair comparison power consumption of the hardware needs to be taken in consideration. Measuring the exact power consumption during inference isn't that straight forward on neither the FPGA nor the GPU. Instead this thesis will present worst to best case scenario when comparing images/second/watt and other power related comparisons. The average power consumption for nVidia Titan Xp is shown in figure 18 and for the Xilinx Zynq UltraScale+ (ZCU102) in figure 19. [31] [32]

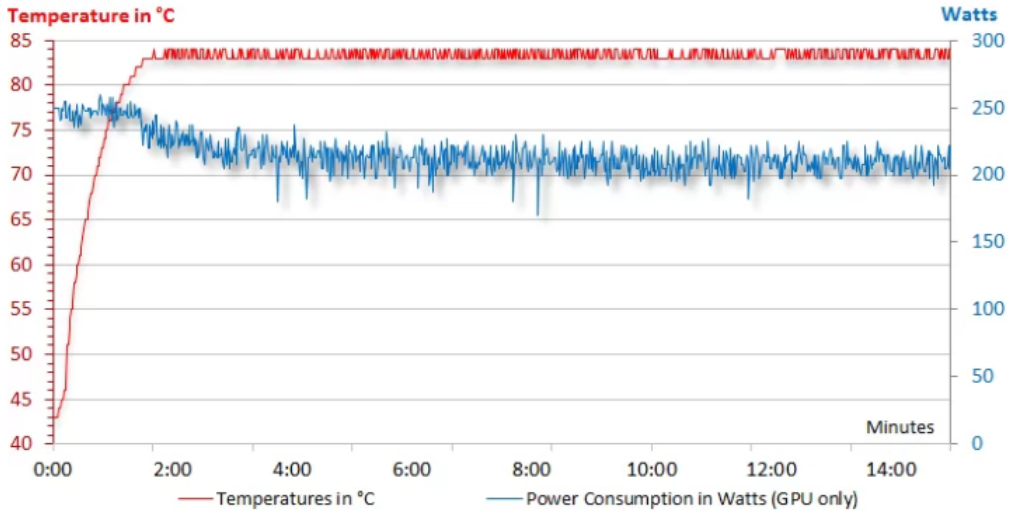

Figure 18: Power consumption compared to temperature on GPU

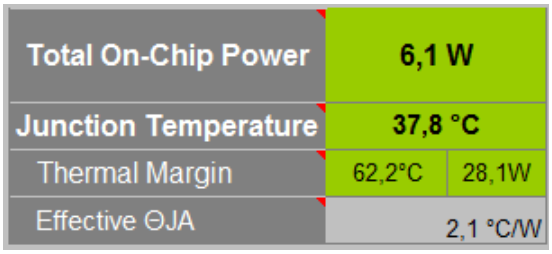

Figure 19: Power consumption compared to temperature on MPSoC

Based on the plots in figure 18 and figure 19 the power consuption can be estimated. From the GPU plot we can see it starts at around 250W power and as soon as the GPU reaches the max temperature of 85°C the power goes to threshold around 210W. The max power for the Titan Xp caps at 275W which would be considered the peak in terms of throughout for the GPU however it's quite unlikely the GPU reaches this power for longer executions. The GPU heats up to max threshold in about 2 minutes in the plot however this is directly connected how heavy computations it's executing. Having a DCNN inference deployed and maxing out the GPU with image size 224x224 pixels takes less than half a second for a single image which makes the assumption of the GPU using close to peak power while executing the DCNN reasonable. The theoretical best scenario in terms of images/second/watt is set to 210W similarly to the run in figure 18. That would be considered the best case scenario in terms of throughput with the worst case scenario being the GPU using 275W.

The FPGA can operate in higher temperatures however the plot supplied by Xilinx caps at 100°C. Having an ambient temperature of 25°C in the test environment and no active cooling for the FPGA except for air cooling the junction temperature independent of the workload will not go under 25°C. Therefore the measurement at 10 degrees can't be confirmed and considering the computational heavy assignment the MPSoC is executing it won't really change any outcome. Therefore the best scenario is set to 6,1W as the calculation in figure 19. The worst case Xilinx Vivado suite estimates the power consumption of the ZCU102 wrapper that CHaiDNN creates for inference on MPSoC to 19,3W which is what we set the worst case to. The wrapper is designed to handle a lot of different network frameworks and models which means some of the logic is planned and counted for in terms of power but not used in the DCNN model inference. The effects of these power consumption calculations in reference to inference are under section 4.1 Results and its subsections.

Another way to increase throughput is to increase the batch size and make the program smaller. This was evaluated in the only way possible with CHaiDNN with the batch size set to 1 instead of 2 and the loops set to 50 instead of 100. Since CHaiDNN only support batch size of 2. Lowering the batch size somehow increased the throughput however it totally diminished the accuracy of the program to levels way below 0,5 which can not be considered valid.

4.1 Results

The following results were achived by accelerating a ResNet50 Image classification network and a VGG-SSD Object detection network on a Xilinx Zynq UltraScale+ MPSoC using CHaiDNN for inference implementation framework. As comparison for the ResNet50 inference on the MPSoC a ResNet50 execution on nVidia Titan Xp GPU is shown. Both the implementations are also compared to the benchmark values from Xilinx CHaiDNN. The VGG-SSD network has a threshold of 200 detected boxes. Data Motion Clock (DMC) is set in SoC environment for the implementation framework together with DSP slice frequency (DSPsF). Supported batch size in CHaiDNN is 2 with 1 tested aswell. Number of execution loop iterations is defined in the code computing the average time disregarding the first loops to avoid the effect of warm-up run. The GPU performance[33] is compared to different frequency settings, batch size and execution loops for the deployed network on the MPSoC. The score is an average of ResNet50 correctly identifying ImageNet class #895 'warplane, military plane' tested on figure 20 and is no the average precision of the network which is Top1/Top5 69,89/89,61 for 6 bit quantized ResNet50 and Top1/Top5 72,97/91,11 for 8 bit quantization. [5]

Figure 20: Test scoring image of two SAAB gripen aeroplanes

4.1.1 ResNet50

ResNet50			
DMC=100MHz DPSsF=100MHz Batch=2 Loops=100			
	MPSoC		GPU
Bit width	6-Bit	8-Bit	32-Bit
FPS	29,77	20,58	348,43
Score	0,894	0,915	-
	FPS/Watt		
Best case	4,88	3,37	1,66
Worst case	1,54	1,06	1,39

Table 14: Results of first ResNet50 implemented with CHaiDNN

ResNet50 DMC=250MHz DPSsF=500MHz Batch=2 Loops=100			
	MPSoC		GPU
Bit width	6-Bit	8-Bit	32-Bit
FPS	59,39	41,59	348,43
Score	0,884	0,915	-
	FPS/Watt		
Best case	9,74	6,82	1,66
Worst case	3,08	2,15	1,39

Table 15: Results of optimized ResNet50 implemented with CHaiDNN

After this test the network inference dropped significantly in score and the last two results is a test of maximum throughput and lowest latency without considering the accuracy of the DCNN.

ResNet50 DMC=250MHz DPSsF=500MHz Batch=1 Loops=50			
	MPSoC		GPU
Bit width	6-Bit	8-Bit	32-Bit
FPS	59,83	47,75	196,1
Score	0,562	0,281	-
	FPS/Watt		
Best case	9,8	7,83	0,93
Worst case	3,1	2,47	0,78

Table 16: Results of over optimized ResNet50 implemented with CHaiDNN

In the last test the inference only completed on 6 bit and crashed trying to compute 8 bit, therefore there is no results for 8 bit. In this last test most of the categories got a low score indicating that the network had no idea what the image showed.

ResNet50 DMC=500MHz DPSsF=500MHz Batch=1 Loops=50			
	MPSoC		GPU
Bit width	6-Bit	8-Bit	32-Bit
FPS	87,71	-	196,1
Score	0,02	-	-
	FPS/Watt		
Best case	14,38	-	0,93
Worst case	4,54	-	0,78

Table 17: Results of broken ResNet50 implemented with CHaiDNN

From table 15 which is the best optimised solution for ResNet50 this thesis achived the images/second/watt in range somewhere from 2 to 6 times more efficient on the MPSoC than Titan Xp. Though the implementation for the Titan Xp can't be considered optimal with batch size 1. The optimal batch size in reference paper[33] is 64 with a FPS of 746,3. This would give the titan a best images/second/watt of 3.55 which still just above the worst case scenario of the optimal MPSoC implementation.

4.1.2 VGG-SSD

The VGG-SSD network has a mAP of 0.7728 for 6 Bit quantization and 0.7806 for 8 Bit.

VGG-SSD			
DMC=100MHz DPSsF=100MHz Batch=2 Loops=100			
	MPSoC		GPU
Bit width	6-Bit	8-Bit	32-Bit
FPS	1,92	1,60	123
Top Score	0,995	0,979	-
#Detected Boxes	182	67	-
FPS/Watt			
Best case	0,31	0,26	0,586
Worst case	0,1	0,08	0,33

Table 18: Results of first VGG-SSD implemented with CHaiDNN

Figure 21: Object detection using 6 bit quantization

Figure 22: Object detection using 8 bit quantization

Reported mAP for this network is 77,28 for 6 bit quantization and 78,06 for 8 bit quantization. For reference the 6 bit image without threshold is shown in figure 23.

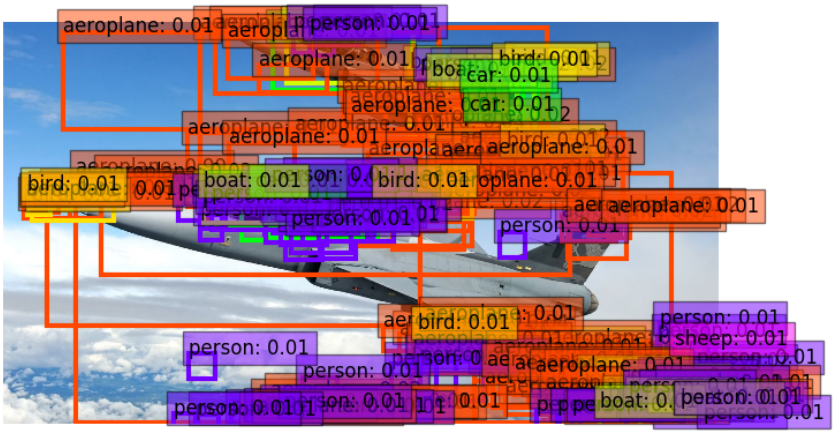

Figure 23: Object detection using 6 bit quantization and no threshold

VGG-SSD			
DMC=100MHz DPSsF=200MHz Batch=2 Loops=100			
	MPSoC		GPU
Bit width	6-Bit	8-Bit	32-Bit
FPS	2,14	1,91	123
Top Score	0,996	0,979	-
#Detected Boxes	182	67	-
	FPS/Watt		
Best case	0,35	0,31	0,586
Worst case	0,11	0,1	0,33

Table 19: Results of double pumped DSPs with frequency 200Mhz and a data motion clock of 100 MHZ on VGG-SSD implemented with CHaiDNN

Figure 24: VGG-SSD 6 bit quantization 100/200 MHz

Figure 25: VGG-SSD 8 bit quantization 100/200 MHz

VGG-SSD			
DMC=250MHz DPSsF=500MHz Batch=2 Loops=100			
	MPSoC		GPU
Bit width	6-Bit	8-Bit	32-Bit
FPS	2,17	1,96	123
Top Score	0,996	0,979	-
#Detected Boxes	182	67	-
	FPS/Watt		
Best case	0,36	0,32	0,586
Worst case	0,112	0,101	0,33

Table 20: Results of double pumped DSPs with frequency 500MHz and a data motion clock of 250MHz on VGG-SSD implemented with CHaiDNN

Figure 26: VGG-SSD 6 bit quantization using 250/500 MHz clock frequencies

Figure 27: VGG-SSD 8 bit quantization using 250/500 MHz clock frequencies

VGG-SSD			
DMC=250MHz DPSsF=500MHz Batch=1 Loops=50			
	MPSoC		GPU
Bit width	6-Bit	8-Bit	32-Bit
FPS	2,26	2,022	123
Top Score	0,996	0,979	-
#Detected Boxes	182	67	-
FPS/Watt			
Best case	0,37	0,33	0,586
Worst case	0,12	0,105	0,33

Table 21: Results of the best VGG-SSD implement using CHaiDNN and frequencies 250/500 MHz

Figure 28: VGG-SSD 6 bit quantization 250/500MHz batch size 1

Figure 29: VGG-SSD 8 bit quantization 250/500MHz batch size 1

4.1.3 Power estimation

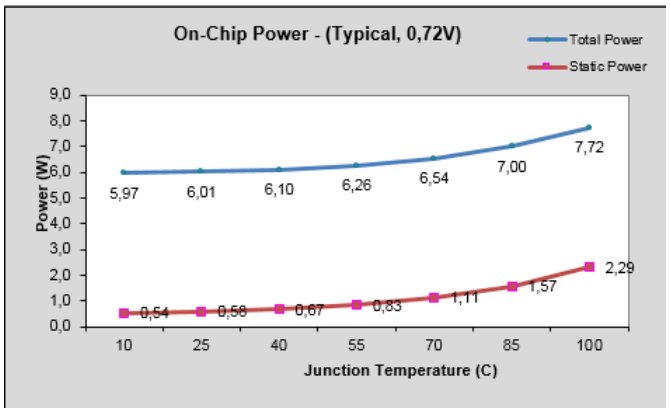

Figure 30: Power estimation according to Xilinx Power Estimator

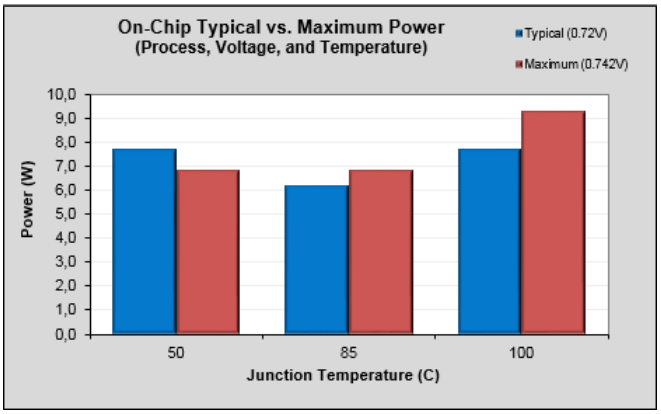

Figure 31: Maximum power estimation for MPSoC ZCU102 using Xilinx Power estimator

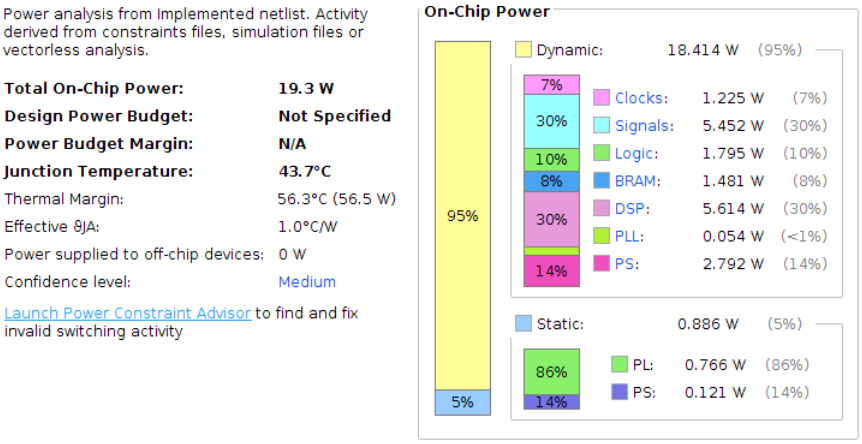

Figure 32: Power estimation for ZCU102 wrapper according to Vivado

5 Discussion

The objectives for the thesis was to research the possibilities of accelerating DCNNs on MPSoCs. The DCNNs were to be modified to fit an implementation where the resources are limited such as for an MPSoC. This was done by trying to convert two existing Keras networks. There were two more reputed converter scripts however non of which claimed direct support for RetinaNet. Initially there were struggles of installing Caffe with two versions not compiling the test runs supplied with the framework. There were also problems getting other tools to work.

5.1 Compatibility

There were compatibility issues on the initial two desktops. Most of it were due to one of them running Windows and the other, even though it was running Ubuntu 16.04 (which most of the frameworks and IDEs used supports), had a lot of packages and other compatibility issues which initially were delt with but in the end were too significant and severe to be handled without taking way to much time. An easy solution to said issue was to use a fresh installation of Ubuntu 16.04 on another computer. After doing this no further compatibility issues regarding desktop OS or packages were encountered.

Coming to this point seems quick in the report however it took some time. Initially the Windows Desktop got the Xilinx SDSoC license and reVISION was tested. The optical flow example gave a representative demonstration of how the workflow of reVISION is meant to be. Optical flow however is a computer vision implementation and not really built as a neural network implementation. Trying to get DeePhi example was not manageable considering the website providing DeePhi and its examples was not accessible. After searching for alternative frameworks a conclusion of Windows compatibility, or more non-compatibility, was drawn with hardly any frameworks being designed for Windows. Therefore a Ubuntu workstation was a necessity. There was one available fully set up and running a lot of other applications locally. It proved to be an issue when trying to install Caffe which was needed for converting the DCNNs provided from Keras to Caffe. The conversion was chosen since both CHaiDNN and DNNWeaver was based on prototxt and caffemodel input files and the frameworks seemed most promising for a working implementation.

5.2 Modifications

Adjusting and modifying a network seems like an excellent idea in theory and a necessary in reality however some DCNNs are designed with MPSoC or similar constrained implementations taken in consideration and others are plainly focused on throughput and accuracy. If the DCNNs performance is at focus naturally the winner of one of the most well known competitions for image classification (ILSVRC) is of major interest. This was the case where the DCNN provided for this thesis was a Keras RetinaNet with ResNet50 as backbone. Although initially it didn't seem to be a big issue with the framework and network model, after studying the subject a worrying few mentions of RetinaNet MPSoC solutions were found. YOLO in all its versions seems to be one of the most popular object detection DCNNs when it comes to networks suited for MPSoCs, and SqueezNet was often used as reference network. The conversion of RetinaNet didn't work and therefore a smaller image classification network that was also provided was tested. The network, called CatsAndDogs, did convert to Caffe with some question marks as to whether or not the correct ".json" files actually were used. These questions rose as there were about five version of the CatsAndDogs network with non descriptive hexadecimal names for their output such as "09589ab176154b70926aac0038895880.json" all in the same directory. After some brute force trial and error of the network a correct output was converted.

5.3 Relevant comparisons

Having a network converted gave some hope upon having one self-trained network evaluated and compared on both local desktop and MPSoC. The hopes were quite quickly destroyed after Xilinx Quantizer refused to handle the network with a cryptic error message occurring in a bit-file. Therefore a standard version of ResNet50 and the only object detection network supplied with CHaiDNN V2 were used to get a ball park estimation of the MPSoC's capabilities and implementations was used. ResNet50 had been evaluated on GPU and multiple sources gave similar results which was good for comparison of the two hardware deployment inferences. The SSD network had also been evaluated on GPU however not with identical prerequisites as the network deployment on the MPSoC. The difference was amongst others the batch size on the GPU. Therefore the throughput on the MPSoC was expected to be lower than the one on GPU. The evaluation done in the previous thesis[7] was almost exclusively on Keras RetinaNet with custom data set which, after the said complications, was not possible to directly compare to inference on the MPSoC. Evaluating ResNet50 and SSD on Titan Xp has been done in multiple papers with reasonable results in comparison to other GPUs and were therefore not tested locally in this thesis.

5.4 Power consumption

Having the objective of comparing inference on a GPU and MPSoC means the largest difference between them both should be taken in to consideration. Deploying a DCNN on a mobile platform such as an aeroplane has a lot of constraints but one of the biggest would be power. The power is defined as the product of the voltage over the device times the current coming through it. Measuring the power is by all means possible but using inference numbers from various sources means the power consumption varies between the sources as well. The same goes for the MPSoC though Xilinx Vivado suite supplies an estimation for the power. Xilinx also have a separate Power Estimator[36] for calculating the power needed for a specified project. According to Vivado suite the deployment of ResNet50 with CHaiDNN requires an estimated 19,3W. Though the power estimator plainly calculates what the block design with the ZCU102 wrapper for CHaiDNN uses and this wrapper is the same for all the different supported frameworks. This means that the Vivado project designed for deploying a DCNN of choice will be designed to handle all the other DCNNs specific model architectures as well and therefore might not utilize all the resources that Vivado calculates with. The usage is illustrated in figure 32.
The other tool Xilinx provides tells a completely different story with an estimated power consumption of 6.1W usage. The calculated power usage from Xilinx Power Estimator is illustrated in figure 30. The plot shows a max power of 7.72W however the program also supports a max power plot which is illustrated in figure 31. This plot show a maximum power usage of 9.3W which is 10W less than Vivado shows. This tool takes the information about resource utilisation as input together with BRAM information and clock frequencies. Therefore it should be considered a decent estimation of power requirements. It might not have the signals power included however adding those from the Vivado estimation results in a total power usage of 11.8W which still is significantly lower than Vivado estimation.

5.5 Results

A successful evaluation of performance on an MPSoC has been concluded. The main issue of building and testing many different deployments with smaller differences in clock speeds and minor changes in code has been the build time. The quickest build took about two and a half hours and the longest took over six and a half hours. While building I researched and set up the next tests which demanded a structured and well organised test plan. A lot of tests resulted in errors or IDE crashes which made more than 3/4 of the tests unusable for evaluation. The frameworks that were tested but not fully evaluated due to the complications covered in this report took quite some time to test as well.

The only framework dropped fairly quickly after struggling was DNNWeaver V1 since there was an updated version available and there was really no use of debugging the implementation of V1 before V2. Since DNNWeaver V2 was way over dimensioned for the ZCU102 evaluation board some research was put in to finding a solution to slim the program down without compromising its deployment. This proved harder than expected. Some online forums have open threads about these issues but no solution. Considering the many different FPGAs on the market it's hard to see why they would choose one of the largest FPGAs available to design the framework.

Normally when deploying a DCNN batching is used to increase the throughput. By batching up a set number of images the time it normally takes to load them of another memory one by one is cut. This is easier to do on a desktop where RAM space normally isn't an issue however it's not as easy on an MPSoC and not applicable in a real time implementation since batching comes with increased latency. To find comparable results batching needs to be accounted for however no deployment of a VGG-SSD net with batchsize lower than 32 were found. Therefore the comparison in the results is unfairly better for the GPU than the MPSoC.

It's not optimal to test accuracy on a single image however to set up a proper Average Precision test for the MPSoC is easier said than done. The network accuracy should not change from its specifications no matter the deployment as long as the whole network is implemented fully. Therefore testing change in accuracy between one frequency and another on the same network can be done with the same image. As the VGG-SSD network shows there were no differences there. However ResNet50 showed a change in accuracy shen changing the same frequencies which is a strange behaviour. Therefore continuous testing was done until the network didn't score anything on the correct class. Optimal implementation was thereby set to the deployment where the accuracy didn't drop significantly and the network still worked as expected. The image might be a bit over fitted to the network considering the high accuracy.

The VGG-SSD network didn't perform as good as expected and no reason for this was found. Theoretically it should be able to execute and analyse an image in just about 27ms whilst my quickest implementation took around 44ms.

6 Conclusion

In conclusion, modifications for a deep convolutional neural network is required for an efficient deployment regardless of MPSoC and framework. There exists a couple of end to end solutions for modifying and converting deep convolutional networks however non of which is fully complete and most of them require a lot of user input in order to give a usable output. The framework I managed to implement was CHaiDNN though using example networks and not SAABs IR-image network. Both an image classification network and an object detection network were evaluated on MPSoC using CHaiDNN. It's preferable and probably way more efficient to initially set up a sketch of the implementation goal with regard to assignment, performance and constraints and choose the best suited network framework from there instead of focusing on the network first and then having to modify it heavily in order for it to fit.

The choice of platform can be hard depending on power and space constraints but an MPSoC provides a good hardware platform for efficient deployment and execution of most networks. The MPSoC will not perform as good as a GPU if not supplied with perfected input architecture and weights. This though is changing rapidly with many interesting new frameworks and network models such as binary networks being developed and perfected. I recon that in the near future the issues of implementing a DCNN on an MPSoC will be completely different and probably not as significant as the ones encountered in this thesis. I will follow the development of this area with great interest. Future investigation of smaller object detection network and their implementations on MPSoCs using CHaiDNN is of great interest and have a better chance for a successful object detection inference on the IR-image data-set than the RetinaNet architecture.

7 References

[1] Commercial aviation accidents statistics - Risks versus flight phases
1001 Crash
https://www.1001crash.com/index-page-statistique-lg-2-numpage-3.html

[2] Roll, Pitch, and Yaw
How things fly
https://howthingsfly.si.edu/flight-dynamics/roll-pitch-and-yaw

[3] Eurocontrol
Visual References, Skybrary, 2018
https://www.skybrary.aero/index.php/Visual$_R$*eferences*

[4] Ian Goodfellow, Yoshua Bengio, Aaron Courville
Deep Learning, MIT Press, 2016
http://www.deeplearningbook.org

[5] Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)
ImageNet
http://www.image-net.org/challenges/LSVRC/2012/results.html

[6] Federal Aviation Administration (FAA), DOT.
Revisions to Operational Requirements for the Use of Enhanced Flight Vision Systems (EFVS) and to Pilot Compartment View Requirements for Vision Systems; Correcting Amendment.
https://www.federalregister.gov/documents/2018/03/12/2018-04888/revisions-to-operational-requirements-for-the-use-of-enhanced-flight-vision-systems-efvs-and-to

[7] Markus Jangblad
Object Detection in Infrared Images using Deep convolutional Neural Networks
http://uu.diva-portal.org/smash/get/diva2:1228617/FULLTEXT01.pdf

[8] Kartik Podugu
What is the deep neural network known as "ResNet-50"?
https://www.quora.com/What-is-the-deep-neural-network-known-as-"ResNet-50"

[9] Olga Russakovsky*, Jia Deng*, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei.
ImageNet
http://www.image-net.org

[10] Jonathan Hui
mAP (mean Average Precision) for Object Detection
https://medium.com/@jonathan$_h$*ui/map* − *mean* − *average* − *precision* − *for* − *object* − *detection* − 45*c*121*a*31173

[11] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollár
Focal Loss for Dense Object Detection
https://arxiv.org/abs/1708.02002

[12] Frank Gray
Gray code
https://en.wikipedia.org/wiki/Gray$_c$*ode*

[13] Eriko Nurvitadhi, Suchit Subhaschandra, Guy Boudoukh, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, and Duncan Moss.
Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?
Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17, pages 5–14, 2017.

[14] Xilinx
DSP Solutions
https://www.xilinx.com/products/technology/dsp.html

[15] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Sérot, and François Berry
Accelerating CNN inference on FPGAs: A Survey
https://arxiv.org/abs/1806.01683

[16] KAIYUAN GUO, SHULIN ZENG, JINCHENG YU, YU WANG AND HUAZHONG YANG
A Survey of FPGA-Based Neural Network Inference Accelerator

[17] Hyeong-Ju Kang
Accelerator-Aware Pruning for convolutional Neural Networks
https://arxiv.org/pdf/1804.09862.pdf

[18] Xilinx
CHaiDNN v2 is now live!
https://forums.xilinx.com/t5/SDSoC-Environment-and-reVISION/CHaiDNN-v2-is-now-live/td-p/875107

[19] Intel
Caffe
https://github.com/intel/caffe

[20] Xilinx
CHaiDNN
https://github.com/Xilinx/CHaiDNN

[21] Xilinx
reVISION overview
https://www.xilinx.com/support/documentation/backgrounders/revision-overview.pdf

[22] openCV
Open source computer vision library
https://opencv.org/

[23] Alternative computing techonolgies lab
DnnWeaver
http://dnnweaver.org/

[24] Javier Duarte, Sergo Jindariani. Ben Kreis, RyanRivera, Nhan Tran (Fermilab)
Jennifer Ngadiuba, Maurizio Pierini (CERN)
Edward Keinar (Hawkeye 360)
Phil Harris, Song Han (MIT)
Zhenbin Wu (University of Illinois at Chicago)
Deep learning on FPGAs for L1 trigger and Data Acquisition
https://instrumentationseminar.desy.de/sites2009/
site$_i$*nstrumentationseminar/content/e*70397*/e*257439*/e*280095*/Ngadiuba_DesySeminar_HLS4ML.pdf*

[25] FxpNet: Training deep convolutional neural network in fixed-point representation
Xi Chen, Xiaolin Hu, Ningyi Xu, Hucheng Zhou
https://www.microsoft.com/en-us/research/wp-content/uploads/2017/04/FxpNet-submitted.pdf

[26] MMdnn Issues
MMdnn
hrefhttps://github.com/Microsoft/MMdnn/issues/https://github.com/Microsoft/MMdnn/issues/

[27] PetaLinux Tools
Xilinx
https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

[28] "From High-Level Deep Neural Models to FPGAs"
H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, H. Esmaeilzadeh
in the Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016

[29] Kintex UltraScale FPGAs
Xilinx
https://www.xilinx.com/products/silicon-devices/fpga/kintex-UltraScale.html

[30] Xilinx Zynq UltraScale+
Xilinx
https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

[31] nVidia Titan X Pascal 12GB Review
Chris Angelini
https://www.tomshardware.com/reviews/nVidia-titan-x-12gb,4700-7.html

[32] Xilinx Power Estimator (XPE)
Xilinx
https://www.xilinx.com/products/technology/power/xpe.html

[33] Benchmark Analysis of Representative Deep Neural Network Architectures
SIMONE BIANCO, REMI CADENE, LUIGI CELONA, AND PAOLO NAPOLETANO
https://arxiv.org/pdf/1810.00736.pdf

[34] SAAB Gripen
saab
https://saab.com/air/air-c4i-solutions/communication-systems/tacticall-vcs-air-operations/

[35] Titan RTX Deep Learning Benchmarks
Michael Balaban
https://lambdalabs.com/blog/titan-rtx-tensorflow-benchmarks/

[36] Power Efficiency
Xilinx
https://www.xilinx.com/products/technology/power.html