Implementation analysis of convolutional neural networks on FPGAs

Ivanovs, I.

Award date:
2019

Technische Universiteit Eindhoven University of Technology

Department of Mathematics and Computer Science
Electronics Systems Research Group

Implementation Analysis of Convolutional Neural Networks on FPGAs

Master Thesis

Ilja Ivanovs

Supervisors:
K.G.W. Goossens (TU/e)
D. van den Heuvel (TOPIC Embedded Systems)

Abstract

Convolutional Neural Networks (CNNs) can achieve human-like results in Computer Vision applications. Unlike other algorithms CNNs are not hard-coded to detect features and instead learn them from a provided dataset. CNNs models are designed in special frameworks that most often utilize Graphics Processing Units for image processing.

In this master thesis, three tools to port CNN models for execution on FPGAs are investigated. Four CNN models for face recognition are trained and applied on the tools. The models have a different number of layers and parameters according to the limitations of the investigated tools. The model execution of BNN-PYNQ, CHaiDNN and ML Suite tools is compared to a GPU solution based on the accuracy, execution time, hardware utilization and power consumption. The tools utilize different strategies like model quantization and hardware architecture setup to achieve accuracy similar to a GPU with maximum of a 10% difference. NVIDIA K40m GPU has at average 2.64× better latency than CHaiDNN and ML Suite implementations but is 2.45× worse than BNN-PYNQ. BNN-PYNQ utilizes 24 DSPs and 124 BRAM blocks, CHaiDNN utilizes 1352 DSPs and 676 BRAM blocks and ML Suite utilizes 972 DSPs and 451 BRAM blocks. BNN-PYNQ and CHaiDNN utilize most of the hardware on their respective FPGAs, where ML Suite utilizes less as it is implemented in a relatively large FPGA. The GPU solution consumes 38.3×, 4.43× and 1.4× more power than BNN-PYNQ, CHaiDNN and ML Suite respectively.

In theory the FPGA solutions can have similar accuracy, better inference time and power consumption compared to the GPUs, however it comes at the cost of limited CNN model support and additional FPGA hardware design complexity.

Acknowledgements

I would like to thank my TU/e supervisor, prof. K.G.W. Goossens, for providing me with critical feedback and fruitful discussions. He offered me guidance and motivation to carry out the thesis.

I would like to thank members of Computer Vision group of TU/e for providing me with different points of view for the problems I have encountered. I also would like to thank my supervisor at Topic Embedded Systems Dirk van den Heuvel for critical feedback and support with various topics.

Finally, I would like to thank my friends and family for providing me with moral support.

Contents

Contents **vii**

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**
1.1 Context . . . 1
1.2 Problem statement . . . 1
1.3 Research question . . . 2
1.4 Approach . . . 2
1.5 Outline . . . 2

2 Background **3**
2.1 Artificial neural networks . . . 3
2.2 Convolutional neural networks . . . 4
2.3 Frameworks . . . 8
2.4 Hardware . . . 8
2.5 Quantization . . . 9
2.5.1 Binary quantization . . . 9
2.5.2 Fixed point quantization . . . 10

3 State of the art **13**
3.1 CNN hardware architectures on FPGAs . . . 13
3.1.1 Streaming accelerator . . . 13
3.1.2 Dedicated accelerator . . . 14
3.1.3 Programmable co-processor . . . 15
3.2 Design flow . . . 15

4 Experimental setup **17**
4.1 Evaluation metrics . . . 17
4.2 Face recognition application . . . 18
4.3 Hardware . . . 19
4.4 Applied CNN model configurations . . . 20
4.5 CNN model design flow . . . 22

5 Experiments and result analysis **27**
5.1 Performed experiments . . . 27
5.1.1 Training time and accuracy . . . 27
5.1.2 Inference accuracy . . . 27
5.1.3 Inference latency and throughput . . . 27
5.1.4 Power consumption . . . 28
5.2 Results . . . 28

5.2.1 Model training time . . . 28
5.2.2 Model accuracy . . . 28
5.2.3 Accuracy compared to training time . . . 29
5.2.4 Hardware-software ratio . . . 30
5.2.5 Tool FPGA utilization . . . 30
5.2.6 Tool power consumption . . . 31
5.2.7 Tool throughput and latency . . . 31
5.2.8 GOP performance . . . 32
5.3 Discussion . . . 34

6 Conclusion **37**

Bibliography **39**

Appendix **43**

A Training details **43**

List of Figures

2.1 Model of an artificial neuron k for a feedforward network described by [16]. 3
2.2 Example of a fully-connected feedforward network with two hidden layers and three outputs by [16]. 4
2.3 Example of a 3×3 kernel applied on a 5×5 hidden layer. 4
2.4 Example of a kernel filter is applied in a CNN Convolution layer by [12]. The example has an 4×3 input matrix, 2×2 kernel, stride 1 and valid padding. The result is a 3×2 output matrix. 5
2.5 Example of a Maxpool layer in a CNN. This Maxpool layer performs a max operation over a 2×2 matrix on 4×4 input features with a stride of 2. 6
2.6 Example of a FC layer in a CNN. A FC layer has 2 weights in the filter and the output is calculated for a 2×2 kernel. 6
2.7 Example of a CNN model used for object detection based on [31]. The model has two convolution and pooling, a FC and a Softmax layer. 7

3.1 CNN hardware architecture layer and buffer (grey) placement on FPGAs. 14

4.1 Pynq-Z2 board (small) 20
4.2 ZCU102 board (medium) 20
4.3 VCU1525 board (large) 20
4.4 Nvidia Tesla K40m GPU 20
4.5 CNN hardware architecture layer and buffer (grey) placement on the small FPGA by BNN-PYNQ tool. 23
4.6 CNN hardware architecture layer, buffer (grey) and controller (blue) placement on the medium FPGA by CHaiDNN tool. 24
4.7 CNN hardware architecture systolic array, buffer (grey), controller (blue) placement on the large FPGA by ML Suite tool. 25
4.8 Design flows performed on the three tools BNN-PYNQ, CHaiDNN and ML Suite. 26

5.1 Total time (hours) spent on training the models and time distribution over the frameworks. 29
5.2 Average time (seconds) spent on training for one epoch for all models in the three frameworks. 29
5.3 Top-1 accuracy of the Casia245 models after training on the GPU (float) and inference on the FPGAs. 29
5.4 Top-1 accuracy of Casia64 model after training on the GPU (float) and inference on the FPGAs. 29
5.5 Training efficiency of the Casia245 models that shows the average accuracy improvement of a model after an hour of training. 30
5.6 Training efficiency of Casia64 that shows the average accuracy improvement of a model after an hour of training. 30
5.7 Hardware utilization of the examined tools. 31
5.8 Power consumption for the evaluated FPGAs. 32

5.9 Latency in ms to execute an image for the Casia245 models. 32
5.10 Latency in ms to execute an image for Casia64. 32
5.11 Throughput in images per second for the Casia245 models. 33
5.12 Throughput in images per second for Casia64. 33
5.13 Number of GOP that can be performed in a second per consumed watt for the Casia245 models. 34
5.14 Number of GOP that can be performed in a second per consumed watt for Casia64. 34

A.1 Steps performed to create Casia245 model types. 46

List of Tables

2.1 CNN inference hardware comparison. TOP/s/W denote how many operations can be done per second for a consumed watt. 9

4.1 The investigated datasets for the face recognition application. 19
4.2 FPGA board hardware overview. 20
4.3 Composition of the Casia245 and Casia64 models. For each convolution and max-pool layer the parameters are denoted as [stride, pad, kernel, filters] 21
4.4 The number of parameters for the investigated CNN models. 22

5.1 The number of epochs spent on training in Keras, Theano and retraining in Caffe. 28
5.2 Hardware software ratio of the CNN models. 30
5.3 Number of images processed per consumed watt. 33
5.4 Number of operations performed in a second for the investigated CNN models. . . 33

Chapter 1

Introduction

1.1 Context

Computer vision has a wide range of applications that include surveillance, automotive, robotics and medical diagnostics. It requires a considerable amount of work to develop an algorithm capable to process and understand an image efficiently. Algorithms like Viola-Jones [50], Eigenfaces [22] or Kalman filter [25] make use of predefined features that can be present in an image. Defining key features and using them in an optimal way is the hardest part of these algorithms. Contrary to defining features, deep learning algorithms like Convolutional Neural Network (CNN) learn the features from a provided dataset of images.

CNNs have gained popularity thanks to their ability to perform object detection and recognition with human-competitive performance on certain tasks [8] and relative ease to construct compared to the other algorithms. To correctly detect and recognize an object, CNNs have to perform computationally intensive operations with floating point numbers. CNN models are usually trained and executed on Graphics Processing Units (GPUs) as GPUs can utilize massive parallelism for floating point number calculations. Alternatives to GPUs could be Central Processing Units (CPUs), Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs), namely Tensor Processing Units (TPUs) [24]. Each hardware platform has its own limitations and advantages. Computer vision applications on embedded platforms could benefit from accuracy of CNNs. GPUs would be consuming too much power for embedded applications, CPUs would not be able to reach real-time performance and ASICs would be costly to design and manufacture. Compared to other hardware platforms, FPGAs can offer flexible and energy efficient solutions for CNN applications on embedded platforms. CNNs can be designed in frameworks like Caffe, Theano, Tensorflow, etc. However, none of the frameworks offer solutions that could be directly implemented on an FPGA.

1.2 Problem statement

As there is a lack of tools that allow to port CNNs on FPGAs the aim of this thesis is to give an overview of tools available for designers and define if a CNN implementation on an FPGA is a feasible solution. Available tools will be evaluated on performance of different CNN implementations. The chosen tools implement different architectures on FPGAs but synthesize hardware for the same manufacturer of chips, namely Xilinx. Each tool has two essential parts of the design flow which are hardware synthesis and application preparation. To evaluate the design flow and performance of the tools a face recognition CNN will be implemented on FPGAs. Depending on the size of the model and the available resources on an FPGA an appropriate CNN architecture type will be synthesized by the tools.

1.3 Research question

The research question can be formulated as follows: *What are the trade-offs between different CNN inference architectures implementations on an FPGA and what design flow(s) should be followed to implement the architectures?* The following sub research questions can be derived from the main question:

- Which CNN *model types* can be synthesized on an FPGA?
- Which CNN hardware *architecture types* are available to perform inference of the CNN models?
- What are *FPGA* requirements to support the CNN model types?
- Which *actions* are required from the designer to implement the CNN models on an FPGA?
- Which *tools* can be used to implement the CNN models on an FPGA?

1.4 Approach

This work performs a survey of the existing tools to port a CNN to an FPGA. A comparison in hardware utilization, performance and the design flow between the tools will be drawn. The hardware utilization will include the hardware requirements (e.g. BRAM, LUT and DSP) and management (e.g. hardware-software ratio). Performance evaluation will include the power consumption, inference time and result accuracy for image classification. Additionally, the design flow will be analyzed for time and effort requirements to perform hardware synthesis, quantization and deployment of a CNN on FPGAs. The survey will give an overview of tools available to designers and will help to decide if a CNN implementation on an FPGA is a feasible solution.

1.5 Outline

Chapter 2 will present the necessary theory used in the thesis. In chapter 3 the publications regarding CNNs and tools will be discussed. Chapter 4 will describe the CNN models and experiments what were performed and used in the thesis. In chapter 5 the results of the experiments and tool evaluation will be presented.

Chapter 2

Background

In this chapter, information regarding the essential topics for the thesis is provided. First, artificial neural networks and the class of convolutional neural networks are discussed. Next the frameworks to develop neural networks are introduced. Then a description of hardware that is used by neural networks is provided and a technique to utilize CNN models on FPGAs is presented.

2.1 Artificial neural networks

Artificial neural networks (ANNs) were the earliest learning algorithms intended to be computation models of biological learning. ANNs are not realistic models of biological functions but rather models to replicate the way brains (of human or mammals) learn [12]. Generally ANNs consist of artificial neurons that form input, hidden and output layers of an ANN. The input layers receive input for the ANN, which could be any data. The hidden layers are responsible for feature extraction from the input data. Finally, the output layers present the output of the ANN, which could be classification results for the input data.

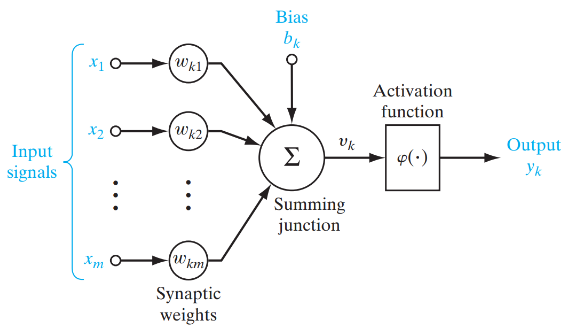


Figure 2.1: Model of an artificial neuron k for a feedforward network described by [16].

Figure 2.1 depicts the artificial neuron model used in feedforward ANNs. Artificial neurons calculate output $y_k = \varphi(\sum_{i=1}^{m}(x_{ki} \times w_{ki}) + b_k)$, where x_{ki} are the input signals to the neuron, w_{ki} are weights associated to the input signals, b_k is a bias term and φ is an activation function. The weights define which inputs are dominant to calculate the output, wheres the bias acts as a threshold to activate the output y_k. Activation functions ensure non-linearity of the neuron outputs thus enhancing the model's capability to learn non-linear dependencies. Since the information flows from the inputs x_{ki} to the evaluation function ε the network is called a feedforward network. Feedforward networks can be extended with feedback connections within neurons, these networks are called recurrent neural networks. The recurrent networks are usually used in machine translation, speech recognition, rhythm learning and music composition application. Feedforward networks are usually applied to computer vision (image classification, object detection) and pattern recognition applications like sales, chemical reaction prediction, etc. An example of a simple feedforward fully connected network is shown in Figure 2.2.

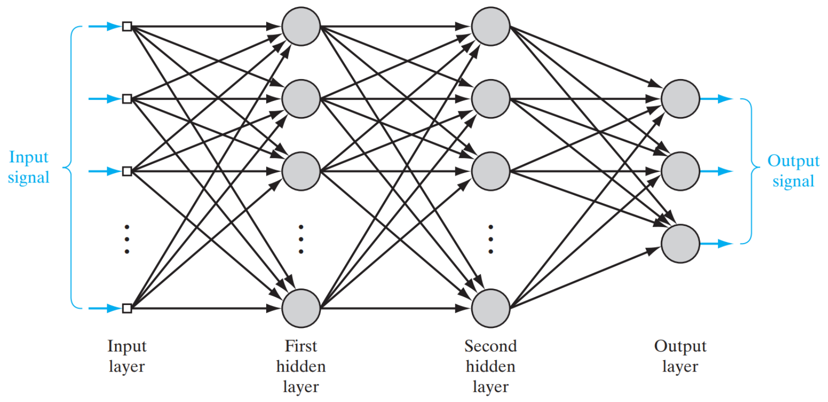


Figure 2.2: Example of a fully-connected feedforward network with two hidden layers and three outputs by [16].

2.2 Convolutional neural networks

Convolutional neural networks (CNNs) are a class of feedforward networks that utilizes convolution operations on a 2D data like images, and thus are mostly used in computer vision applications. Hidden layers of a simple artificial neural networks have all neurons fully connected to the previous layers (depicted in Figure 2.2). Due to this connectivity networks would not scale well with larger input images. For example if an input image is of size 32x32 RGB pixels (used in the CIFAR-10 dataset [29]) then a single neuron of the first hidden layer would have $32 \times 32 \times 3 = 3072$ weights. However, if the image size is larger like in the images of the ImageNet dataset [10] that have size 256x256x3 the first hidden layer would have 196608 weights. Considering that hidden layers consist of multiple neurons the number of weights and the calculation complexity drastically increases for additional hidden layers. CNNs assume that there is spacial locality of the features present in an input image, in other words on an image local pixels usually contribute to the same object. CNNs apply filters on a receptive field called a kernel (defined as a $k \times k$ matrix) for all hidden layers. Figure 2.3 shows an example of a kernel applied on a hidden layer in a CNN.

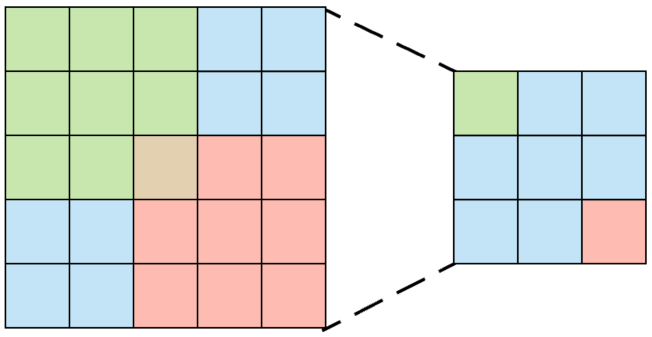


Figure 2.3: Example of a 3×3 kernel applied on a 5×5 hidden layer.

Several types of hidden layers are utilized to extract features from images. A brief description of the layers is shown next:

- **A convolution layer** is a hidden layer that performs convolution operations on the inputs with a defined kernel. Kernels have filter weights that are applied in steps to the inputs of the previous layer. The steps are described as a stride in a CNN model and define how many values have to be shifted in the input matrix to apply the kernel filter. Applying a kernel on an input matrix results in a smaller output matrix. The input matrix size can be preserved by applying padding values around the input matrix (filled with 0s). Padding is called "same" if the size of the output is equal to the size of the input matrix and "valid" otherwise. Figure 2.4 shows an example of a convolution layer.

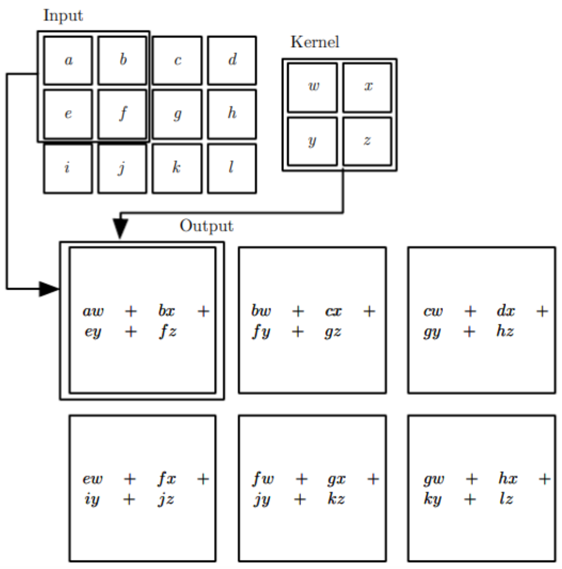


Figure 2.4: Example of a kernel filter is applied in a CNN Convolution layer by [12]. The example has an 4×3 input matrix, 2×2 kernel, stride 1 and valid padding. The result is a 3×2 output matrix.

- **A poling layer** combines input features over a kernel in a single output based on the layer function. There are two main functions used: maximum and average. A Maxpool layer takes a kernel of $k \times k$ of an input and returns the largest value, and an Avgpool returns the average of all values within the kernel. Figure 2.5 shows an example of a maxpool layer.

- **A fully-connected layer** (FC) calculates features of an input for all filter weights in the layer. The kernel for FC layers has the size of the outputs of the previous layer. FC layers are usually used at the end of a CNN to calculate class prediction scores. Although FC layers contribute more parameters than the other layers, they are less compute intensive. Figure 2.6 shows an example of a FC layer.

- **Non-linearity layers** are used after convolution and FC layers to apply non-linearity activation functions on the output values (depicted as φ in Figure 2.1). Sigmoid and Hyperbolic Tangent Function (tanh) activations were the first to be used in deep learning networks. Rectified Linear Unit (ReLU), its variants (leaky ReLU) and Exponential Linear Units (ELU) were proven to achieve better results [2] [37], and thus are more often used in modern neural networks.

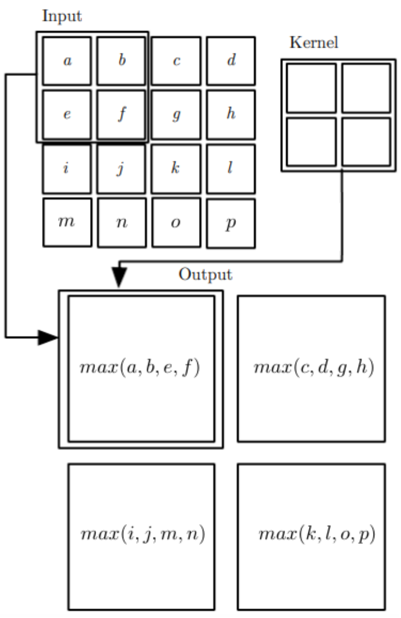


Figure 2.5: Example of a Maxpool layer in a CNN. This Maxpool layer performs a max operation over a 2×2 matrix on 4×4 input features with a stride of 2.

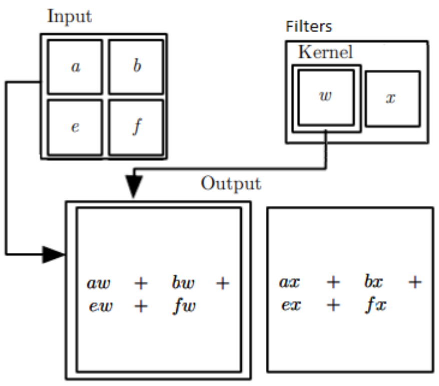


Figure 2.6: Example of a FC layer in a CNN. A FC layer has 2 weights in the filter and the output is calculated for a 2×2 kernel.

- **BatchNorm layers** are used to normalize inputs for hidden layers across several batches of training images. During network training implementation of BatchNorm layers can decrease the training time and improve model accuracy [21].

- **Dropout layers** can be implemented to improve accuracy and convergence of CNN models. The dropout layers randomly prevent some of the weights from activating during training, and thus prevent a model from overfitting [45].

- **A softmax layer** is used as the last layer for classification applications. The layer implements a softmax function that normalizes real values of the output classes in a vector whose components sum to 1.

An example of a complete CNN model is presented in Figure 2.7.

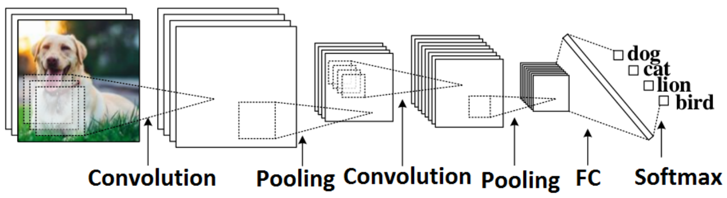


Figure 2.7: Example of a CNN model used for object detection based on [31]. The model has two convolution and pooling, a FC and a Softmax layer.

A designed CNN model initially has random parameters (weights, biases) that do not produce any meaningful results. The parameters have to be modified during CNN model training that is performed on a dataset. There are two possible approaches to train a CNN model: supervised and unsupervised learning. Supervised learning assumes that the training dataset has labels for all images that are used to correct error of a CNN model output. Unsupervised learning does not have labels, and thus model has to learn relationships between elements on its own. A CNN model is fed with training images in batches. Usually to speed-up training on a GPU batches of 128 images or more are used to utilize massive parallelism of the GPUs.

The duration of training a CNN model on whole set of training images is called an epoch. A CNN model can converge (learn features and produce meaningful output) after training for several epochs, which depends on the complexity of a task and the structure of a model. CNN models use a loss function that defines the difference between the output of the model and the input label (if present). The loss function is calculated for each image and is accumulated for the training batch. After a batch of images is processed the accumulated loss is used in optimization algorithms to update the parameters backwards in the model. The optimization algorithms are used to minimize the error calculated as the loss. By updating the parameters and minimizing the error CNN models learn features and dependencies to produce correct outputs. There are several optimization algorithms that are used, namely Adaptive Moment Estimation (ADAM) [27], Stochastic gradient descend (SGD) [26], Nesterov accelerated gradient (NAG) [4], etc.

During the training the performance of a model can be measured by the accuracy of the model to predict correct results and by the loss function. Usually the performance is done on a validation set, a set that contains images that were not used during training for the parameter update. The performance of a training set can indicate on how well the network can generalize. If the network has too many "free" parameters (not contributing for feature detection) it can start to overfit the training set by simply memorizing features instead of generalizing them. This behaviour can be detected if during training the training loss becomes smaller but the validation loss increases.

2.3 Frameworks

CNN models can be designed in frameworks like Caffe [23], Theano [48], Tensorflow [1], DarkNet [40], etc. The frameworks provide functionality to build layers for models, perform model training and run inference (execute a model without backpropogation).

Caffe designs model layers and configuration (kernel, padding, stride, etc.) in special files called .prototxt and stores the parameters in a .caffemodel file. The hyperparameters (base learning rate, momentum, optimization algorithm, learning rate decay) are used to define how fast, aggressive the model will modify the parameters of the hidden and output layers. The hyperparameters are used only for training are defined in a different configuration .prototxt file. The configuration files together with the model description can be passed as parameters to main Caffe scripts to start training or inference. The model design and hyperparameters have to be described in python scripts for Theano and Tensorflow by using framework specific low level functions. Keras or Lasagne APIs can be used to simplify usage of Tensorflow low level functions. Keras stores model description and weights in a .h5 format but Lasagne in .npz (numpy arrays).

The frameworks are built to utilize CPU or GPU for training and inference. Depending on CUDA configuration used by the frameworks, multiple GPUs could be used in parallel for training, for example Caffe supports multiply GPUs but Tensorflow and Theano can work with only one. Due to resource limitations on FPGAs and complexity of the backpropogation performing complete training process on a FPGA is a challenging task. There is an ongoing research on performing training on FPGAs and only some parts of the training can be accelerated [3]. Unfortunately, none of the frameworks offer solutions that could be directly implemented on an FPGA, and thus third party applications have to be used to run inference on FPGAs. A complete workflow is presented in Chapter 4.5.

2.4 Hardware

To correctly detect and recognize an object CNNs have to perform multiply-accumulate (MAC) operations with floating point numbers. The networks usually are trained and executed on Graphics Processing Units (GPUs) as GPUs can utilize massive parallelism for floating point number calculations. Running inference of neural networks requires much less computation. In addition to GPUs inference can be executed on CPUs, FPGAs and ASICs, namely TPUs. A brief description of the hardware platforms used for CNNs is shown next:

- **CPUs** can carry out instructions to perform necessary arithmetic calculation for the inference. Although CPUs can utilize SIMD instructions and multiple cores to execute several arithmetic computations in parallel, they still would lack the speed (compared to the alternatives) to perform overwhelming amount of computations required for CNNs.

- **GPUs** are processors with multiple cores designed to process blocks of data in parallel. CUDA and OpenGL are frameworks that can be utilized to distribute data over multiple processors and internal memory of a GPU. GPUs can store up to 12GB of data in a high bandwidth memory which allows fast access to floating point data. However, the immense processing power comes at cost of power consumption of GPUs which can reach up to 250W [36].

- **FPGAs** offer a flexible hardware architectures as different computation units can be utilized unlike CPU or GPU that are fixed to a single architecture. FPGAs can offer flexible solutions to match the pace of rapid developments done in the neural network research. Neural network layers can be placed on an FPGA and executed in parallel for an efficient inference.

Table 2.1: CNN inference hardware comparison. TOP/s/W denote how many operations can be done per second for a consumed watt.

	Architecture	Parallelism	Memory	TOP/s/W	Precision
CPU [20]	basic	low	12MB	x1	Float
GPU [36]	co-processor	high	12GB	x1.2-4	Float
FPGA [52]	streaming/dedicated/co-processor	medium	512MB-4GB	x90-244	1-16 bit
TPU [24]	co-processor	high	16GB	x127	8 bit

However, memory bandwidth and operating frequency of FPGAs would become a bottleneck for the inference time compared to a GPU solution. Nonetheless the bottleneck can be overcome by reducing data precision from a floating point to a lower fixed point representation or reusing data on an FPGA.

- **TPUs** are ASICs designed to perform inference of neural networks and improve cost-performance over GPUs [24]. TPUs are optimized to perform 8 bit calculations for the basic layers that are used in neural networks. As TPUs are optimized for certain layers rapid development of neural networks could potentially make the design obsolete. Moreover, the design and manufacturing of the TPUs is costly compared to the alternatives.

Each hardware platform has its own limitations and advantages. Table 2.1 shows relative comparison between the hardware platforms. GPUs have much more computing power than FPGAs or TPUs, and thus can outperform any alternative in terms of time of execution. To compete with GPUs FPGAs and TPUs quantize the data to a smaller representation than a 32 bit floating point.

2.5 Quantization

There are several techniques that can be utilized to reduce memory footprint of CNNs or speed up execution time. Techniques like Pruning [18], loop unrolling, reordering [56], tiling [49] and Winograd Convolution [11] are implemented as a part of a framework or a tool and thus are passively used. Quantization is a technique that has to be applied independently of a tool or a framework, although the tools and frameworks are usually optimized to support certain quantizations. In this work two data quantization techniques are explored that represent the data as a binary or fixed point numbers.

2.5.1 Binary quantization

In Binary quantization both activation and weights are represented as a single bit data. XOR-Net [39] proposed a strategy to binarize weights of a CNN. The strategy focuses on approximating float precision tensor of weights $\mathbf{W} \in \mathbb{R}^{\mathrm{c \times w \times h}}$ as a product of a binary matrix $\mathbf{B} \in \{+1, -1\}^{c \times w \times h}$ and a scaling factor $\alpha \in \mathbb{R}^+$ such that $\mathbf{W} \approx \alpha\mathbf{B}$. The parameters c, w and h represent the number of filter (or input) channels, width and height, respectively. A convolutional operation on the input tensor of activations $\mathbf{I}$ can be approximated by:

$$\mathbf{I} * \mathbf{W} \approx (\mathbf{I} \oplus \mathbf{W})\alpha \tag{2.1}$$

where, $\oplus$ indicates a convolution without any multiplication.

XOR-Net uses a deterministic binarization approach for the CNN layer parameters. The deterministic binarization defines the output depending on the sign of the CNN parameters:

$$x^b = \mathrm{Sign}(x) = \begin{cases} +1, & \text{if } x \geq 0, \\ -1, & \text{otherwise.} \end{cases} \tag{2.2}$$

BinaryConnect [9]is an alternative that uses a stochastic binarization, that is defined as:

$$x^b = \begin{cases} +1, & \text{with probability } p = \sigma(x), \\ -1, & \text{with probability } 1-p, \end{cases} \tag{2.3}$$

where σ is the *"hard sigmoid"* function:

$$\sigma(x) = \max(0, \min(1, \frac{x+1}{2})) \tag{2.4}$$

Parameter quantization is performed during training of a model on all parameters except the parameters of the input and the output layers. Parameters for the input and the output layers are saved as floating point numbers, to keep the information of the image and the rest are represented as binary values. The binary representation is used in accumulation and then converted to floating point for multiplication. For all parameters the Sign function is used as it is easy to implement in the hardware and fast to compute. The stochastic binarization is used in some activations of a model and requires the hardware to generate random bits.

Half-wave Gaussian Quantization [5] improves the quantization methodology and achieves a higher accuracy. The quantization is based on non-linearity activation (ReLU) parameters (represented as tensor **I**). The parameters are quantized in Forward approximation as:

$$Q(x) = \begin{cases} q_i, & \text{if } x \in (t_i, t_{i+1}], \\ 0, & x \leq 0, \end{cases} \tag{2.5}$$

where $q_i \in \mathbb{R}^+$ and $t_i \in \mathbb{R}^+$ ($t_1 = 0$ and $t_{i+1} = \infty$). During training the parameters are quantized in Backward approximation as a clipped ReLU:

$$\widetilde{Q}_c(x) = \begin{cases} q_m, & \text{if } x > q_m, \\ x, & \text{if } x \in (0, q_m], \\ 0, & \text{otherwise.} \end{cases} \tag{2.6}$$

The clipped ReLU helps to mitigate problems with inaccurate gradients during the backpropogation.

2.5.2 Fixed point quantization

The fixed point quantization described in [42] dynamically quantizes parameters across CNN layers. For each layer a threshold value γ is defined that can be expressed as:

$$\gamma = \alpha \times \beta \tag{2.7}$$

where, α is a scaling factor and $\beta = 2^{\text{exponent}} \times$ singed integer represents a reduced precision floating point value. The β fixed point representation is defined by quantization parameters n and p that represent the bit width and the number of significant bits, and is represented as

$$\beta = \begin{cases} (-1)^{b_{n-1}} \times \sum\limits_{i=0}^{p-1} (b_i \times 2^i), & \text{if } e = 0, \\ 2^{e-1} \times (-1)^{b_{n-1}} \times (2^p + \sum\limits_{i=0}^{p-1} (b_i \times 2^i)), & \text{if } e > 0, \end{cases} \tag{2.8}$$

where e is defined as

$$e = \begin{cases} 0, & \text{if } p = n-1, \\ \sum\limits_{i=p}^{n-2} (b_i \times 2^{i-p})), & \text{if } p < n-1, \end{cases} \tag{2.9}$$

During the quantization procedure γ values for all layer parameters (tensors) are estimated with the Kullback-Leibler-I from the previous tensor. For the estimation a calibration forwardpass of the network is executed on a calibration dataset. To prevent information loss by removing least significant bits the estimation of the threshold values is done after the activation layers for each convolution layer. Each floating point value in a tensor is clipped by $[-\gamma; \gamma]$ and then divided by its scale α. The α is calculated as the smallest positive value depending on n, p values that are the bit width and the number of significant bits in a signed integer. When inference is performed, the obtained thresholds and scaling factors during offline quantization are used to improve accuracy of the fixed point values, that are calculated from the floating-point values by Equation 2.8.

The thesis will focus on training CNNs on GPUs and executing inference on FPGAs. Several combinations and different number of CNN layers will be used to design CNN model types. FPGA resource restrictions will be taken into account during the CNN model design phase. Since the optimization techniques depend on a tool, parameter quantization to binary and fixed-point representation will be evaluated.

Chapter 3

State of the art

The popularity of CNNs and potential implementations on embedded devices has motivated researchers to investigate the possibilities to implement CNNs on FPGAs. FPGAs can offer real-time performance, high energy efficiency and flexible designs. Multiple studies have performed different approaches on improving performance optimization and power consumption.

3.1 CNN hardware architectures on FPGAs

Based on how CNN layers are handled on the hardware of an FPGA three architectures can be defined. For a Streaming accelerator all CNN layers are separately synthesized on an FPGA. As the output values of each layer can be streamed in the input buffers for the next layer. The parameters are loaded once in the layers from the global memory and reused locally. A Dedicated accelerator synthesizes layer classes on an FPGA and schedules layer execution with a controller. The controller defines which layers are executed and when the necessary parameters are loaded, usually from the global memory to local. A programmable co-processor architecture is based on the implementation of the TPU from Google. The MAC operations for the layers are executed in a systolic array and a controller is responsible for scheduling layer execution by managing the inputs and the outputs of the systolic array calculations. Some of the implementations of the presented classes also use a CPU to execute non-compute intensive layers (FC, Softmax) or as a scheduling controller. Figure 3.1 shows the CNN hardware architectures that will be evaluated in the thesis.

3.1.1 Streaming accelerator

Courbariaux et al. [9] have shown that performing binarization of the floating point parameters of a CNN can yield nearly state-of-the art results. Y. Umuroglu et al. [49] have presented FINN, a framework that optimizes CNNs and synthesizes a Streaming accelerator on an FPGA. FINN performs binarization of floating point numbers to keep all neural network parameters in the on-chip memory. Additionally, FINN optimizes hardware on the FPGA to utilize binary calculations, like using a boolean XOR operation for Maxpool layer and a pop-count for the accumulation. Li et al. [31] performed binarization of the CIFAR-10 dataset and managed to outperform GPU in the inference time and power consumption, however only for small batches of images. In addition to using binary gates for parameter computation, the model also utilized a double buffering scheme and pipelining to ensure massive computing parallelism on an FPGA.

Nakahara et al. [34] has implemented object detection and recognition on an FPGA that achieves real-time performance for the inference. The proposed model utilizes the FPGA to execute CNN layers in a streaming manner. Additionally, an ARM processor is used to define a multiscale sliding window, supply input images to the FPGA layers and compute Softmax function. Multiple

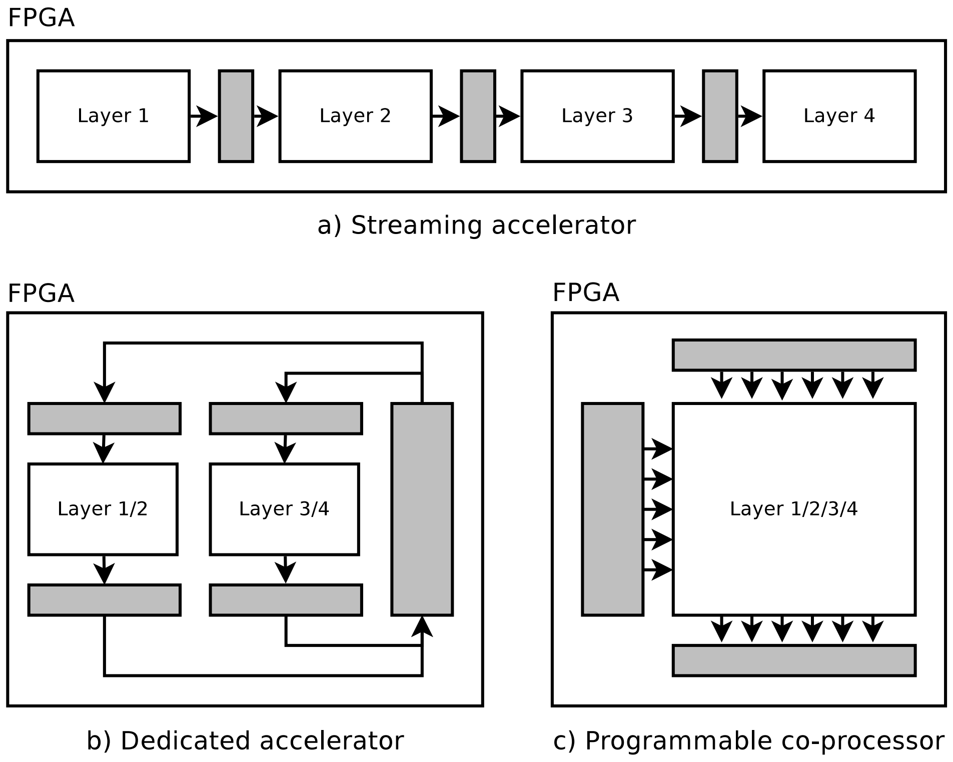


Figure 3.1: CNN hardware architecture layer and buffer (grey) placement on FPGAs.

sliding windows allow to precisely detect an object in the input image.

3.1.2 Dedicated accelerator

Chen et al. [56] have implemented a CNN with 32-bit floating point parameters on an FPGA containing Processing Engines (PEs) which accelerate layers called by a microprocessor. The PEs are configured with the theoretical roofline model [51], that relates system performance to off-chip memory traffic and performance of the hardware.

Preuer et al. [38] use an ARM CPU on a heterogeneous FPGA to process less computation intensive layers, while accelerating the other layers on the FPGA. Parallel test execution with the FPGA allows to increase the inference performance and resource utilization. DiCecco et al. [11] also use a CPU to reduce DSP utilization in a Winograd convolution algorithm. The FPGA is used to accelerate convolution layers. Huang et al. [16] use a CPU to perform pooling and fully connected layer computations, in addition to quantizing parameters of a CNN and Butterfly pipelining of the input data. As convolution layers are executed in pipelined manner, the delay of FC layer execution on a CPU is hidden.

Xilinx has released CHaiDNN a High Level Synthesis (HLS) library for Xilinx Ultrascale+ MPSoCs [53]. CHaiDNN can perform quantization of the given network parameters and map the network on an FPGA via HLS. At its core, CHaiDNN synthesizes default layers (Conv., Pool., ReLU, etc.) on FPGA and schedules the execution by a CPU. Additionally, CHaiDNN allows a configuration of initially unsupported layers, however those are executed on a CPU.

3.1.3 Programmable co-processor

Xilinx has also released ML Suite [54] a set of tools that can parse a CNN described in frameworks like Caffe and Tensorflow to an FPGA. ML Suite synthesizes a systolic array, instruction memory and execution controller on the FPGA that can perform acceleration of the convolution and pooling layers. FC and Softmax layers are computed on an ARM processor. Additionally, ARM processor schedules layer execution to reuse instructions for the systolic array.

Kiningham et al.[28] have implemented ConvAu a systolic array architecture that accelerates Convolutional, FC and BatchNorm layers. They have achieved 200× improved TOPs/W than inference on a K80 GPU and 1.9× improvement compared to the TPU.

3.2 Design flow

In previous works authors designed the FPGA hardware synthesis by themselves. ML Suite from Xilinx provides tools to compile CNNs for FPGA. xDNN IP tool synthesizes the Processing Engines of systolic arrays on FPGA and configures them for maximum throughput or lowest latency. xDNN Middleware is a software library that provides tools to optimize networks described in the supported frameworks and compile instructions for the synthesized systolic arrays. xDNN Quintizer is a tool that quantizes the parameters for all layers.

Sharma et al. [43] proposed a tool called DNNWeaver, which can synthesize hardware for an FPGA for a required network. The output of the synthesis is focused on the power consumption rather than the inference time. To achieve good results the tool utilizes handmade template accelerators to build the network.

The Streaming accelerator architecture can provide the fastest execution time as it has the least overhead for layer execution and communication to the global memory. However, it requires parameters of the CNNs to be binary to store them in local buffers. The parameter binarization can lead to worse accuracy compared to floating or fixed point parameters. The Dedicated accelerator architecture can support fixed point parameters of various sizes which can yield small accuracy drop compared to floating point parameters. A careful layer scheduling is required to run execution smoothly. The Programmable co-processor can be optimized well for a high throughput as it mainly accelerates the core operations (MAC) performed in convolution, pooling and FC layers. The co-processor can efficiently utilize hardware but comes with a larger overhead for operation executions compared to the other architectures. In this thesis a Streaming accelerator, Dedicated accelerator and Programmable co-processor architectures will be implemented with BNN-PYNQ, CHaiDNN and ML Suite tools respectively. Additionally, the design flows from training CNN models to running inference on FPGAs will be derived for the tools.

Chapter 4

Experimental setup

In this chapter, the setup on which experimental CNN models were implemented are presented. First, an application for the CNNs is chosen. Then the designed CNN models that use the application are presented. The models are the trained on a GPU to obtain useful parameters for successful application execution. Finally, the trained models are implemented on FPGAs and evaluated on inference performance. Additionally, the metrics used to evaluate the results and the design flow of the setup are given.

4.1 Evaluation metrics

The thesis evaluates three tools that port CNNs on FPGAs. To evaluate the performance of the tools and the implemented CNN models on FPGAs the following metrics are defined below.

(A) CNN model architecture

- **Learnable parameters**
  The total number of parameters represents how much data is required to store weights and biases of the CNN models. The parameters can be calculated from the layers that store them e.g. Convolution, FC. Based on the number of parameters a depth and size of a model can be approximated.
- **Number of operations**
  The total number of operations (MACC, compare, add, div) that are necessary to calculate outputs for all layers in a CNN model. The number can be calculated from the applied layer configurations (kernel, filters, padding, etc.) during the design phase. The number of operations can be used to determine the complexity of a CNN model and the load on hardware.

(B) CNN model training

- **Time**
  Time (hours) needed to train a CNN model on a GPU until the model starts to saturate on validation loss. The time will consist of the number of hours needed to train CNNs in Keras and re-train the parameters in Caffe. The hours are combined as Caffe failed to converge and model format created by Keras is not supported by the used tools. The comparison between CNN model training time and prediction accuracy can contribute to the trade-offs between the CNN models.
- **Accuracy (TOP-1)**
  Top-1 accuracy is the fraction of test images for which the correct label (class) has the highest score. The Top-1 score will be calculated based on the results of 1470 test images (six images per class) for the large CNN models and 364 test images for the

smaller model. If the highest probability is given to the correct class label, then the test image will be added to the fraction of the correct predictions for Top-1 accuracy. This metric can help to define how well CNNs perform in a face recognition task.

(C) CNN model mapping on FPGA

- **Hardware to software ratio**
  The metric represents the ratio of work performed on the hardware accelerated part of an FPGA and software executions on a CPU that is present on a board. The metric will be defined as the number of operations in the convolutional parts and in the fully-connected parts from the CNN models. It will show how well CNN parameter calculations are balanced between an FPGA and CPU.
- **FPGA hardware utilization**
  The metric will be defined as a number of hardware parameters (DSP/LUT/FF/BRAM) used in the inference architecture of the FPGAs over the total hardware parameters available on the FPGAs. The numbers of used hardware parameters will be obtained from the corresponding synthesis reports and the total available numbers from the FPGA datasheets. The utilization will show the requirements on FPGA hardware to perform CNN inference.

(D) CNN model inference

- **Accuracy (TOP-1)**
  Top-1 accuracy is measured in the same as for training. The measured accuracy during inference will be affected by the applied quantizations for the FPGA solutions.
- **Power consumption**
  The power consumption metric will represent the power (W) required to run inference of an image on an FPGA. To measure the power consumption of a BNN-PYNQ tool on the small FPGA an USB cable will be connected to a power measurement device that measures the voltage and current during the inference. To measure the power consumption of the medium FPGA used by the CHaiDNN tool, a multimeter will be used to define the input current and voltage for the medium FPGA. A software tool that measures on-board power consumption will be used for the GPU. The power consumption metric will show the trade-offs between the tools and will be combined with the number of operations per second.
- **Latency and throughput**
  Latency represents the time (ms) needed to execute a CNN model on an FPGA for one image without the network initialization. The time will be measured between the start and the end of the inference messages for all test images. Throughput (images/s) represents the number of images an FPGA can process in a second. The throughput will be measured as the number of batches of images that are performed over their latency. This metric will be used to evaluate the trade-offs between the CNN model sizes and FPGA architectures.
- **Number of operations per second (GOP/s)**
  A number of (Giga) operations per second a hardware type (FPGA, GPU) can perform. The metric will be defined as the number of operations (MAC, compare, add, divide) that have to be performed over the measure throughput for each tool. The metric will show compute possibilities of the hardware and combined with the power consumption the computation efficiency will be derived.

4.2 Face recognition application

During the application selection for the thesis an abstract problem that could be solved with CNNs on FPGAs was in mind. There are several problems that CNNs can solve, namely object classifica-

Table 4.1: The investigated datasets for the face recognition application.

	LFW [17]	CelebFaces [32]	MS-Celeb-1M [13]	CASIA [55]	VGGFace2 [6]
Total images	13K	202K	10M	500K	3.3M
Total classes	5K	10K	100K	10K	9K
Images/class	2	20	100	50	366

tion, object detection, picture segmentation, natural language processing. Since CNNs are mostly used in image processing, natural language processing was not considered. From the applications object classification was considered more interesting to explore as this type of CNNs would have to learn features to classify different classes, while in object detection CNNs would learn features of a single class to locate it in an image. It was decided to narrow down the application to a face recognition problem as it fits the object classification domain and potentially could be performed independently from a face detection.

The face recognition application is a CNN model that predicts actor labels based on the input images. An image would be loaded on a hardware platform, then CNN inference would be performed to define the face of the actor present in the image. Since it is an object classification problem, it is assumed that an existing class label (from learned actor features) is present for the input image.

The first step to create a face recognition application is choosing a dataset with faces. During the dataset selection only publicly available datasets were investigated. Ideally, the desired dataset should have 1000 images per class (suggested for best classification results). Unfortunately, not many publicly available datasets exceed more than 100 images per class. Additionally, some datasets are noisy with duplicate or irrelevant (no face present, faces of other people) images, which could influence the prediction accuracy of CNNs. Table 4.1 shows the datasets of faces that were investigated for the thesis.

It was decided to use 245 classes of the CASIA WebFace as these classes have at least 200 unique images per class and a few duplicate/noisy images. The dataset was divided in 70K images for training, 13K images for validation and 1,5K for testing. An alternative dataset that could be used, but was not considered is VGGFace2. It offers 9K classes with on average 362 images per class. As the dataset was found after the CNN models were trained on the CASIA WebFace it was not used.

4.3 Hardware

The hardware was chosen based on what the three tools BNN-PYNQ, CHaiDNN and ML Suite can support.

BNN-PYNQ is a tool that implements Binary Neural Networks (BNNs) from the FINN paper [49] on the small FPGAs (in terms of available hardware) Pynq-Z1, Pynq-Z2 and the medium ZCU104 FPGA boards. CHaiDNN is a tool that can perform synthesis of a CNN with quantized 8-bit and 6-bit parameters for the medium ZYNQ Ultrascale+ ZCU102 and ZCU104 boards. Xilinx ML Suite offers a set of tools to optimize and deploy a trained CNN on the large FPGAs like VCU1525 and Alveo U200/U250. When the thesis was conducted, ML Suite only supported VCU1525 and BNN-PYNQ supported Pynq-Z1 and Pynq-Z2. For the experiments Pynq-Z2, ZCU102 and VCU1525 were available, and will be referred as small, medium and large respectively. Figure 4.1 shows the small FPGA, Figure 4.2 shows the medium FPGA and Figure 4.3 shows the large FPGA.

The small and medium FPGAs were available in house, and thus the programming and experiments were executed directly on the FPGAs. The large FPGA was used via Nimbix cloud [35], and thus

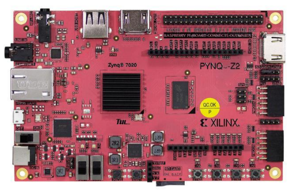


Figure 4.1: Pynq-Z2 board (small)

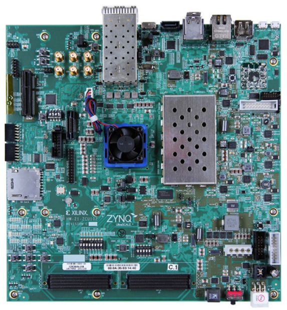


Figure 4.2: ZCU102 board (medium)

Figure 4.3: VCU1525 board (large)

Figure 4.4: Nvidia Tesla K40m GPU

Table 4.2: FPGA board hardware overview.

FPGA	small	medium	large
# LUTs (K)	53.2	274	1182
# FF (K)	106.4	548	2364
BRAM (KB)	630	4108.8	9715.2
DSP slices	220	2520	6840

not all desired experiments could be performed. For the power measurement, additional hardware was required to be connected to the large FPGA, to which there was no access. The CNN models and test images were implemented via Python scripts accessed through a shh connection to the Nimbix servers. Table 4.2 summarizes the total available FPGA hardware parameters that can be utilized by CNN models. The actual utilized hardware parameters are later compared in section 5.2. The Nvidia Tesla K40m GPU was used for the face recognition CNN model training. The GPU was part of the Nimbix cloud service and accessed through Nvidia DIGITS framework for Caffe training and through Python scripts executed via shh commands for training in Keras. Figure 4.4 shows the Nvidia K40 GPU used for CNN model training.

4.4 Applied CNN model configurations

There are different models publicly available for CNN training and inference. Some of the popular CNN models are AlexNet [30], VGG [44], ResNet [14], GoogleNet [47], MobileNet [15], SqueezeNet [19], Inception-v4 [46] and YOLO [41]. As BNN-PYNQ supports only VGG type models, to compare BNN-PYNQ to the other tools and for the sake of consistency a VGG-16 model was used as the base. The overview of all implemented CNN models is presented in Table 4.3. Due to CNN layer compatibility complications in ML Suite and BNN-PYNQ, two different VGG

Table 4.3: Composition of the Casia245 and Casia64 models. For each convolution and maxpool layer the parameters are denoted as [stride, pad, kernel, filters]

Model name	Casia64	Casia245-t1	Casia245-t2	Casia245-t3
input	32×32×3	128×128×3		
Conv1_x	[1, 0, 3, 64] [1, 0, 3, 64]	[1, 1, 3, 64]	[1, 1, 3, 64] [1, 1, 3, 64]	[1, 1, 3, 64] [1, 1, 3, 64] [1, 1, 3, 64] [1, 1, 3, 64]
Maxpool1	[2, -, 2, 64]			
Conv2_x	[1, 0, 3, 128] [1, 0, 3, 128]	[1, 1, 3, 128]	[1, 1, 3, 128] [1, 1, 3, 128]	[1, 1, 3, 128] [1, 1, 3, 128] [1, 1, 3, 128] [1, 1, 3, 128]
Maxpool2	[2, -, 2, 128]			
Conv3_x	[1, 0, 3, 256] [1, 0, 3, 256]	[1, 1, 3, 256]	[1, 1, 3, 256] [1, 1, 3, 256]	[1, 1, 3, 256] [1, 1, 3, 256] [1, 1, 3, 256] [1, 1, 3, 256]
Maxpool3	[2, -, 2, 256]			
Conv4_x	-	[1, 1, 3, 256]	[1, 1, 3, 256] [1, 1, 3, 256]	[1, 1, 3, 256] [1, 1, 3, 256] [1, 1, 3, 256] [1, 1, 3, 256]
Maxpool4	-	[2, -, 2, 256]		
Conv5_x	-	[1, 1, 3, 512]	[1, 1, 3, 512] [1, 1, 3, 512]	[1, 1, 3, 512] [1, 1, 3, 512] [1, 1, 3, 512] [1, 1, 3, 512]
Maxpool5	-	[2, -, 2, 512]		
FC1	512	245		
FC2	512	-		
FC3	64	-		

derivative models were used for the tools. Since the goal of this thesis was to investigate available tools, no modifications were made to ML Suite, CHaiDNN and BNN-PYNQ to support additional layers/settings.

The VGG model for BNN-PYNQ consists of six convolutional layers, three max pooling layers and three FC layers at the end. BNN-PYNQ can support maximum of 64 output classes, thus only 64 actors are used for the VGG model from the CASIA WebFace dataset. This model was implemented in BNN-PYNQ tool as a BNN and in CHaiDNN as a 6-bit and 8-bit CNN. The VGG model will be referred as Casia64.

Due to limitations on the number of FC layers in ML Suite (v1.1) another type of a VGG model had to be designed. The new model could only use a single FC layer as the ML Suite xDNN compiler tool can save parameters for one FC layer. To investigate the impact of the CNN size on accuracy and throughput three VGG model types were implemented for ML Suite and CHaiDNN. The first type model (Casia245-t1) implements one convolutional layer followed by a maxpool layer five times and ends with an FC layer. The second type model (Casia245-t2) doubles the number of Conv. layers between the maxpool layers and the third type (Casia245-t3) quadruples compared to the Casia245-t1 model. Rectification non-linearity (ReLU as in VGG-16) layers are

Table 4.4: The number of parameters for the investigated CNN models.

	Casia64	Casia245-t1	Casia245-t2	Casia245-t3
Parameters	1.57M	4.15M	7.87M	15.32M
GOPs	0.31	1.73	5.95	14.40

used after each Conv. layer in all models.

The original VGG-16 model uses 224×224 input images and a valid padding, but with these settings Casia245-t3 would not have any input parameters in the last FC layer. To make sure that the only difference between the Casia245 type models is in the number of Conv. layers (and thus learning parameters) the Conv. layer parameters were fixed to stride 1, kernel 3×3 and padding 1. Additionally, the image input size was set to 128×128×3 (RGB) to reduce the computational load of the models as FPGAs have less computational power compared to GPUs. The image input for Casia64 is fixed by BNN-PYNQ tool to 32×32×3 due to memory restrictions of the small FPGA.

The number of parameters in a model is defined by the total number of weights and biases. The higher the number of parameters the more time it takes for a model to learn, and thus more features can be extracted. The total number of operations for a CNN model is defined by the total number of MACs×2, compare, addition, division and exponent operations. Table 4.4 shows the number of parameters and (Giga) operations for the investigated models.

4.5 CNN model design flow

In this section, the preparation and setting up the Casia64 and Casia245 models on the FPGA are described. Additionally, design flows for the evaluated tools (BNN-PYNQ, CHaiDNN and ML Suite) are presented and summarized in Figure 4.8.

CHaiDNN and ML Suite tools are designed to work with Caffe files but unfortunately without an apparent reason the designed CNN models were not able to learn anything in Caffe framework. Thus the models were trained in Keras API [7] with Tensorflow backend and later converted and re-trained in Caffe. BNN-PYNQ is designed to work with Theano generated files and did not have problems like Caffe. More details regarding Caffe and Keras training process can be found in Appendix A.

To execute inference on FPGAs supported by the BNN-PYNQ, CHaiDNN and ML Suite tools some additional modifications had to be made to the trained models. For BNN-PYNQ tool hardware has to be synthesized in Vivado, a driver and application have to be rewritten. The hardware synthesis was run in Vivado to prepare a bitstream file. The bitstream file programs hardware on the FPGA that defines memory allocation, number of LUTs, FFs and DSPs. It synthesized the model on the FPGA fabric by setting up the connections between the layers. The software drivers have to be created to enable the communication between the application and the hardware. During the application setup, the location of the images can be configured to read and execute one image or run several images in a pipeline manner. Hardware synthesis, driver and application code has to be rewritten for new models as these depend on the model layer composition and settings. The existing hardware configuration and drivers were used for Casia64, and thus no Vivado synthesis and driver update was done. The parameters learned by Casia64 in Theano were already quantized during training to fit on the BRAM of the small FPGA. The layers for the Casia64 model are placed in a stream like fashion with input and output buffers to ensure efficient pipelining. The model parameters are stored within the layers and the layer calculation results are stored in the input/output buffers. Figure 4.5 shows how the layers for the Casia64 model are placed on the small FPGA.

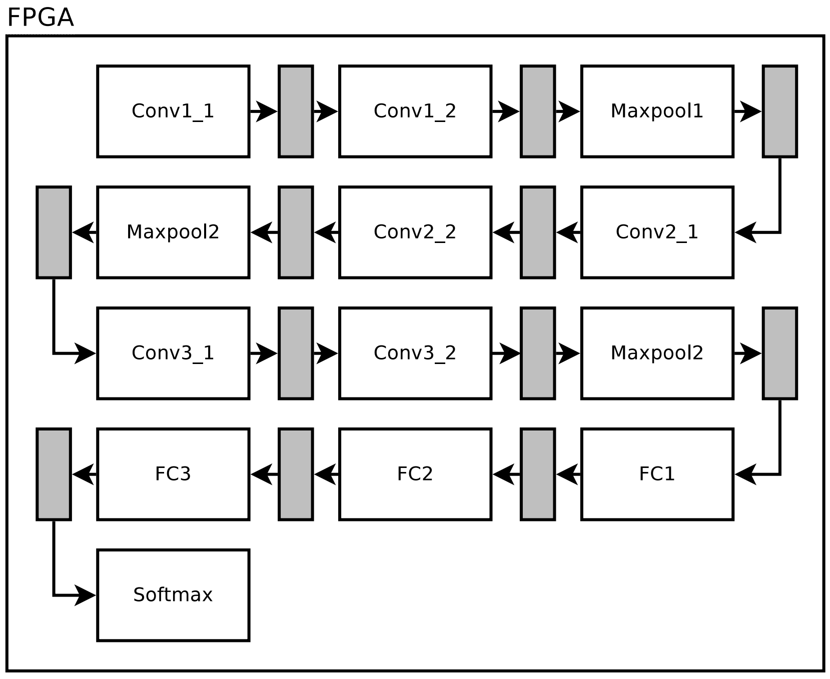


Figure 4.5: CNN hardware architecture layer and buffer (grey) placement on the small FPGA by BNN-PYNQ tool.

CHaiDNN utilizes a CPU to schedule layers of a CNN model and executes the most computation extensive layers sequentially on an FPGA tile. The user can build hardware configurations in Vivado and setup an application for the inference. There are four possible hardware configurations that can be used for the inference. The configuration that uses the most of the hardware was built in Vivado, as it promises the fastest inference and the best utilization of the available hardware. The user can define the start and the end layers of a caffemodel, input image dimensions, dataset pixel mean values, model and picture locations in the application. Once these parameters are defined, the application can be build as an executable .elf file. The CNN model quantization is done by calling a python script that implements the dynamic fixed point quantization described by S. O. Settle et al. [42]. The script creates a prototxt file with quantized weights, which is used by an elf executable. Convolution and pooling layer classes are synthesized on an FPGA. Additionally, the activation class (ReLU) is combined with the Convolution synthesized block on an FPGA. FC layers and Softmax are executed on a CPU. Figure 4.6 shows how the layers for the Casia64 and Casia245 models are placed on the medium FPGA.

ML Suite is composed of xDNN Intellectual Property (IP) cores, xfDNN and ML Framework. xDNN IP that is used for the experiments was included in the tool and designed for the large FPGA. No changes to the IP core could be made as the source files were not publicly available. The xfDNN middleware consists of two parts: a compiler and a quantizer. The compiler is a script used to optimize CNN model and divide it in instructions used by the xDNN. The quantizer is a script that implements the same quatization method as CHaiDNN but with additional optimization for the xDNN IP. The ML Framework provides Python APIs to configure the CNN model parameters (first, last layers, FC and Conv. layer inputs) and image location. Additionally, the framework calls the model files and instructions created by the xfDNN for the inference execution. The systolic array placed on the large FPGA is responsible for the convolution and pooling operations of the respective layers. The FC and Softmax layers are executed on a CPU. Additionally, the CPU is responsible for controlling layer execution and parameter loading on the

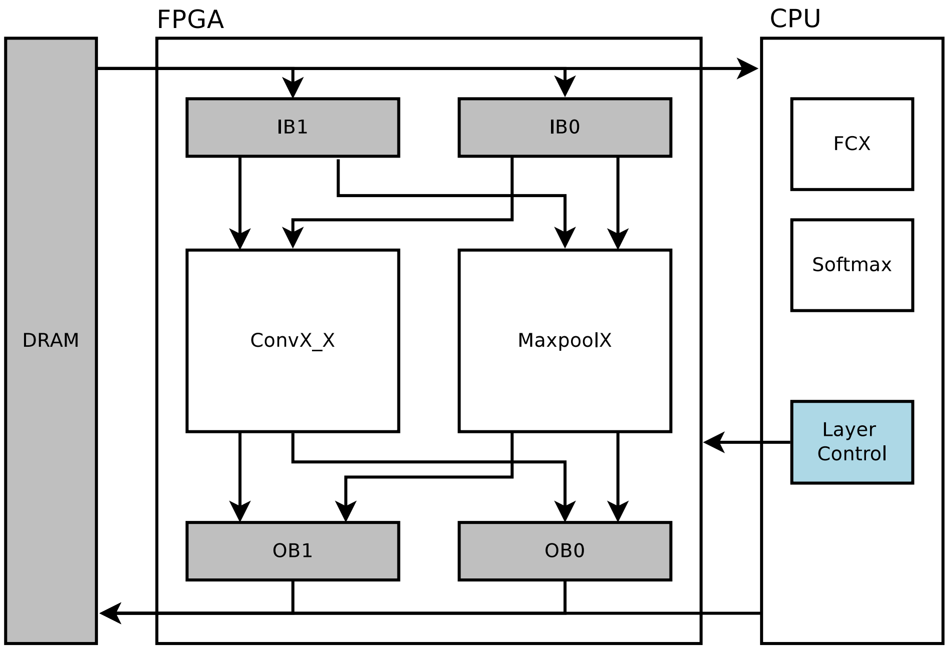


Figure 4.6: CNN hardware architecture layer, buffer (grey) and controller (blue) placement on the medium FPGA by CHaiDNN tool.

FPGA. Figure 4.7 shows how the layers for the Casia245 models are handled on the large FPGA.

Figure 4.8 summarizes the flows for BNN-PYNQ, CHaiDNN and ML Suite tools. The blue rectangles represent common steps that have to be done for all tools. The orange represents steps that are specific for each tool and red is the final goal to run the inference.

The tools to port CNN models on FPGAs target devices of various sizes. The tool restrictions on supported layers and CNN model configurations force to utilize different model layouts. A compromise had to be made for the applied CNN models to compare the tool performance. Model design flow is similar for all tools and mainly consists of designing a model, training and synthesizing the model on an FPGA. However, ML Suite and CHaiDNN tools are more generic and are easier to implement in the model synthesis step compared to BNN-PYNQ.

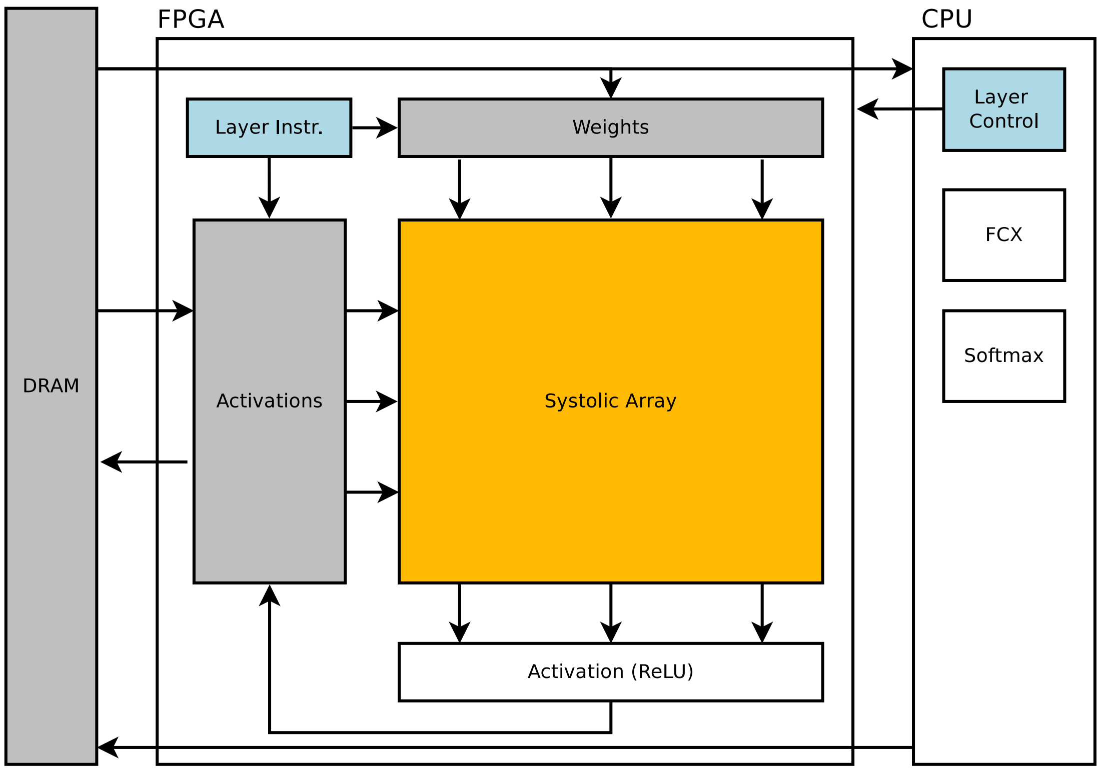


Figure 4.7: CNN hardware architecture systolic array, buffer (grey), controller (blue) placement on the large FPGA by ML Suite tool.

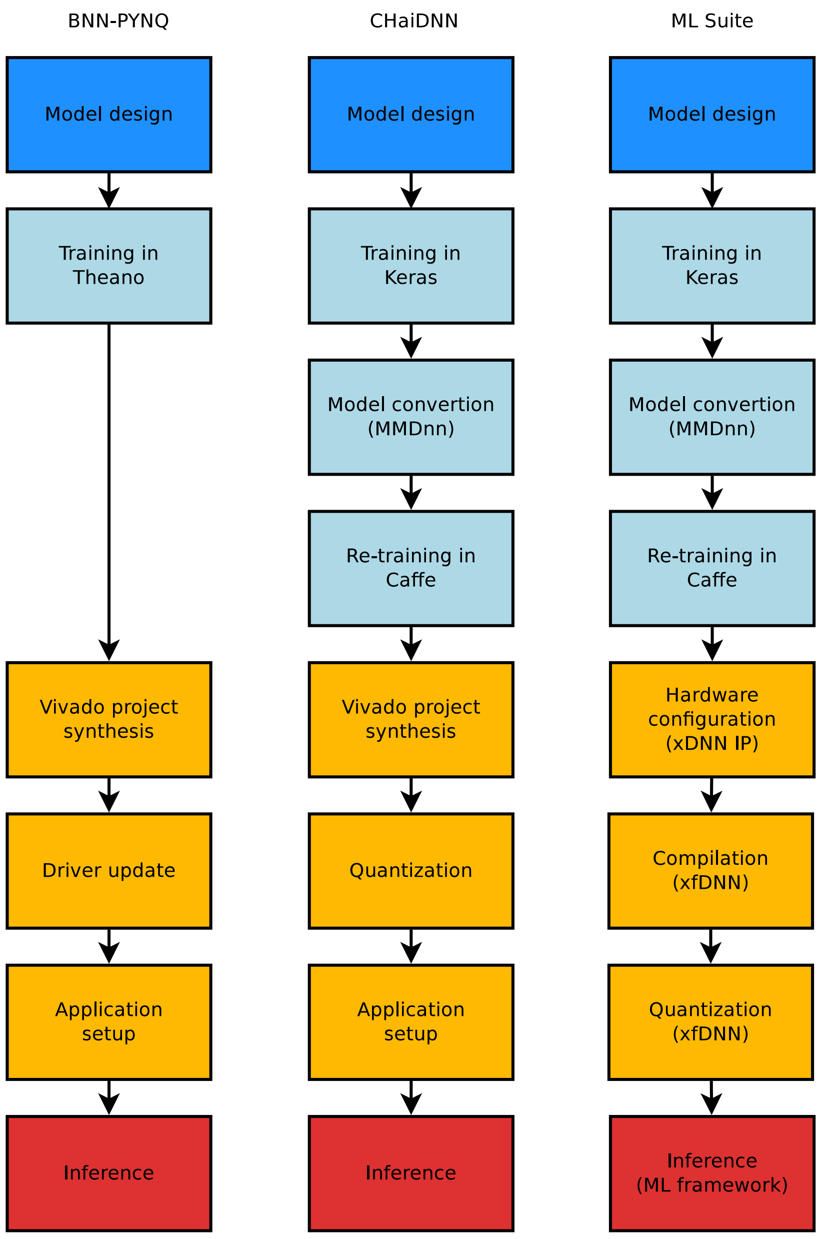


Figure 4.8: Design flows performed on the three tools BNN-PYNQ, CHaiDNN and ML Suite.

Chapter 5

Experiments and result analysis

In this chapter, the performed experiments on the FPGAs will be presented. Additionally, the results of the inference and training of the CNN models will be discussed and analyzed.

5.1 Performed experiments

The experiments were performed during training and inference on two datasets (one for Casia245 and one for Casia64). Training tests were performed on 13K validation images and 70K training images for the Casia245 models. During inference tests, a test set of 1430 images was used for the Casia245 models and a test set of 384 images for the Casia64 model. Both test sets use six images per class. The images for inference were taken from the Casia WebFace dataset and never used in training.

5.1.1 Training time and accuracy

The first conducted experiment involved measuring the time needed to train the models until the validation loss stopped improving. The measurement starts from the network initialization to the last epoch with the lowest validation loss. These measurements were performed for the Keras training phase. The Caffe recalibration was fixed to 50 epochs for Casia245 and to 200 epochs for Casia64. The time to recalibrate the models was added to time needed to train in Keras. During the CNN model training the accuracy of the test set was reported by the Nvidia DIGITS 5 tool after each epoch.

5.1.2 Inference accuracy

The Top-1 accuracy was measured for all models based on the predictions of the corresponding test sets. The accuracy of the models with the weights stored as floats was measured on retrained Caffe models on a GPU. Caffe *test* option was used to report Top-1 accuracy. The single bit model of Casia64 was run from Jupyther notebook on the small FPGA. The inference on CHaiDNN was executed as an elf file on the medium FPGA. The inference for ML Suite was run on the provided python script *batch_classify.py*.

5.1.3 Inference latency and throughput

The inference setup and actions for time measurements were almost the same as for accuracy. The time of inference on the GPU was obtained by running Caffe *time* command. The average forward pass time was taken to measure latency as it is the time needed to process an image from the input layer to the output layer. The time was measured only for the inference thus the time to load images or allocate memory was not measured. ML Suite inference script reported the time needed to write weights to an FPGA, execute main layers on the FPGA, read outputs and execute layers on the CPU. The total inference time was measured starting from layer execution on the FPGA and ending with FC layer execution on the CPU. The throughput was measured by setting batch number to 128 for BNN-PYQN and GPU, 2 for CHaiDNN and 4 for ML Suite. BNN-PYNQ and GPU implementations did not gain much for setting a larger batch size, CHiaDNN and ML Suite can support max batch size of 2 and 4 respectively.

5.1.4 Power consumption

The power consumption was measured during inference on the FPGAs and the GPU. The consumed power for small (BNN-PYNQ) and medium (CHaiDNN) FPGAs was calculated as $V_{in} \times I_{in}$. The small FPGA was powered with a USB cable. The input voltage and current was measured with a KCX-017 power bank capacity tester attached to the USB cable. The medium FPGA was powered with an TENMA 72-6905 power supply. The input voltage and current to the FPGA board were measured with a multimeter. The power consumption of the GPU was obtained from nvidia-smi tool. Unfortunately, no power measurements could be done for the large FPGA (ML Suite) as the FPGA was located in Nimbix data center and the additional hardware to measure the power consumption could not be connected. According to datasheet of the large FPGA it consumes 75W and can increase the power consumption by additional 150W if necessary.

5.2 Results

In this sections the results of the performed experiments are presented according to the metrics discussed in the previous chapter.

5.2.1 Model training time

The time spent on training the models is summarized in Figure 5.1 and the number of epochs are presented in Table 5.1. The bigger the model the more time has to be spent on training in terms of the number of epochs to reach optimal validation loss and time spent per epoch processing. With the increased number of learning parameters in the models the frameworks had to perform more computations to update the parameters. Since the computations were performed on a GPU data parallelism and computation optimization improved the time to compute larger networks. The time spent on re-calibration is significant compared to the training time due to Caffe requiring more time to process an epoch compared to Keras as can be seen in Figure 5.2. This could be due to inefficient data (images and weights) representation in Caffe. Similar issue can be observed with Theano training time compared to Keras and Caffe. However, Theano trained Casia64 model with special *BinaryConvolution* layers designed with HWGQ [5]. Theano spends on average 16 seconds on image cropping which could be removed as the training set images are already cropped. For image batch processing it spends 26 seconds but Caffe 17. Theano sequentially processes images in a loop and GPU CUDA does not run in parallel the loop itself, instead the calculations are done within the loop, which could be the reason for the difference with Caffe.

5.2.2 Model accuracy

Figure 5.3 shows Top-1 accuracy for the Casia245 models and Figure 5.4 shows Top-1 accuracy for the Casia64 model. The accuracy of Casia64 with floating point weights after training for both

Table 5.1: The number of epochs spent on training in Keras, Theano and retraining in Caffe.

	Casia64	Casia245-t1	Casia245-t2	Casia245-t3
Keras	188	48	89	143
Caffe	200	50	50	50
Theano	200	-	-	-

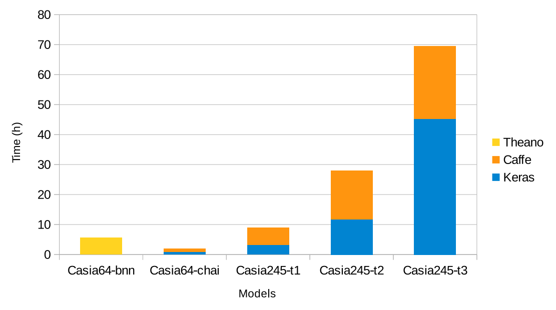


Figure 5.1: Total time (hours) spent on training the models and time distribution over the frameworks.

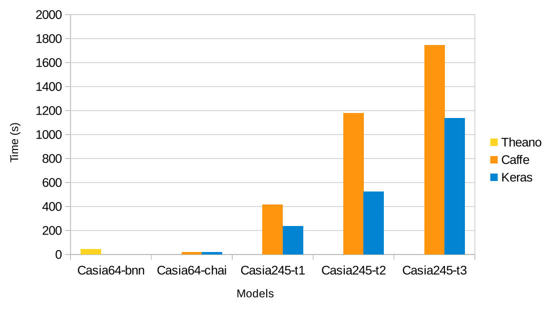


Figure 5.2: Average time (seconds) spent on training for one epoch for all models in the three frameworks.

Theano and Caffe retrained models (used for CHaiDNN) are 40.32% and 53.12% respectively. The development on Theano stopped on November 2017, however Caffe is still somewhat being updated, thus Caffe has a better weight calculation methodology. Moreover, Caffe retrained the models based on the weights trained in Tensorflow. The accuracy of the model was significantly reduced after quantization was applied. The poor accuracy of 1-bit weights could be explained by the lack of accuracy of the float model, although it only dropped by 5%. The drop in accuracy for the 6-bit and 8-bit models of Casia245-t3 for CHaiDNN is approximately 10%. However, the accuracy of the Casia245 models stayed the same as the base floating precision models except Casia245-t3 models for CHaiDNN. Xilinx quantization used for CHaiDNN and ML Suite mainly scaled the floating precision weights to a fixed point representation. Perhaps the increased number of parameters and round up of floating point parameters accumulated an error in quantization which lead to worse results for Casia245-t3.

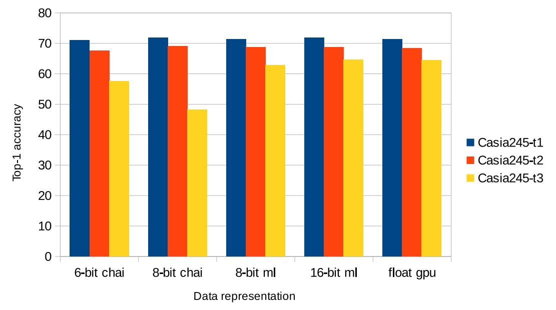


Figure 5.3: Top-1 accuracy of the Casia245 models after training on the GPU (float) and inference on the FPGAs.

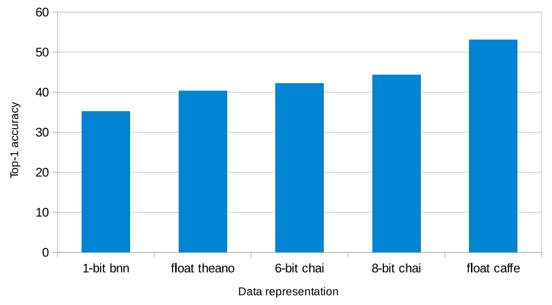


Figure 5.4: Top-1 accuracy of Casia64 model after training on the GPU (float) and inference on the FPGAs.

5.2.3 Accuracy compared to training time

Figure 5.5 shows the training efficiency of the examined Casia245 models. The efficiency was calculated as follows:

$$train\_efficiency = \frac{accuracy}{training\_time}$$

Table 5.2: Hardware software ratio of the CNN models.

	Casia64-BNN	Casia64-CHai	Casia245-t1	Casia245-t2	Casia245-t3
Total GOPs	0.3108	0.3108	1.7254	5.9458	14.4096
Software GOPs	0	0.0016	0.004	0.004	0.004
Software fraction	0	0.005	0.002	0.0006	0.0002
Hardware fraction	100	99.995	99.998	99.9994	99.9998

As it can be observed from the figure the Casia245-t1 model has the best time to accuracy ratio. The model was big enough to learn face features to execute the face recognition task and took less time to train than the other models. Increasing the model sizes did not improve the accuracy enough to justify the increased training time. Since the quantized models had similar accuracy there is almost no deviation in training efficiency compared to the models with the float parameters. The Casia64 model trained with Theano has worse accuracy on the test set than the model trained in Keras and retrained in Caffe. Due to long training in Theano the model has worse efficiency than Keras and Caffe combined.

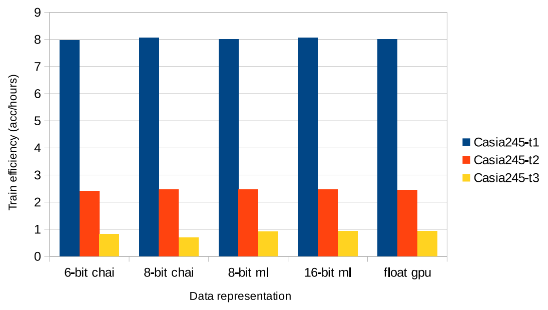


Figure 5.5: Training efficiency of the Casia245 models that shows the average accuracy improvement of a model after an hour of training.

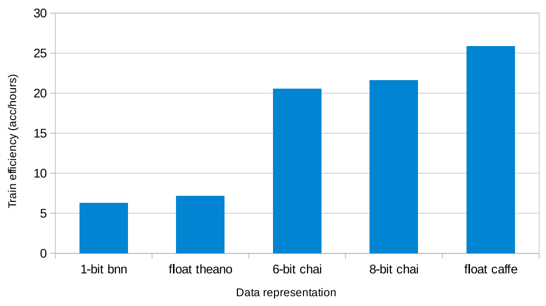


Figure 5.6: Training efficiency of Casia64 that shows the average accuracy improvement of a model after an hour of training.

5.2.4 Hardware-software ratio

Table 5.2 shows the hardware-software partitioning of the operations executed during inference of the CNN models based on which layer are executed on an FPGA or CPU. Both CHaiDNN and ML Suite execute on FPGAs Conv. and max-pool layers but FC and Softmax layers on CPUs. As only a small fraction of all operations has to be done for FC and Softmax layers (noted in Table 5.2 as software GOPs) most of the calculations are performed on the FPGAs. This calculations do not count instructions that are used to issue calculations for ML Suite and control scheduling on CHaiDNN. As all layers are synthesized on the FPGA fabric for BNN-PYNQ all calculations are done on hardware.

5.2.5 Tool FPGA utilization

The hardware utilization is presented as a number of LUTs, FFs, DSPs and BRAM used for inference as these are main components required for calculations on FPGAs. Figure 5.7 shows the hardware utilization of the examined tools. The highest hardware utilization is in BNN-PYNQ and CHaiDNN tools as these use much less hardware than ML Suite. BNN-PYNQ tool processes 1-bit weights thus it does not require additional DSPs or BRAM. Instead it relies on XOR and SHIFT operations that are executed by FFs and on LUTs to store the CNN parameters. CHaiDNN uses

6.2 times more LUTs, 3.6 times more FFs, 56 times more DSPs and 5.45 times more BRAM than BNN-PYNQ. CHaiDNN uses a substantial amount of DSPs on the medium FPGA for calculating 6-bit and 8-bit convolution operations. Although ML Suite hardware utilization of the large FPGA is relatively low, it utilizes 9 times more LUTs and FFs, 40 times more DSPs and 3.6 times more BRAM compared to BNN-PYNQ mapped on the small FPGA. ML Suite uses instructions to execute CNN layers on a systolic array, which consists of DSPs paired with LUTs. The BRAM is used to store all the weights of the CNN models on the FPGA and for executions the weights are saved in LUTs of the systolic array.

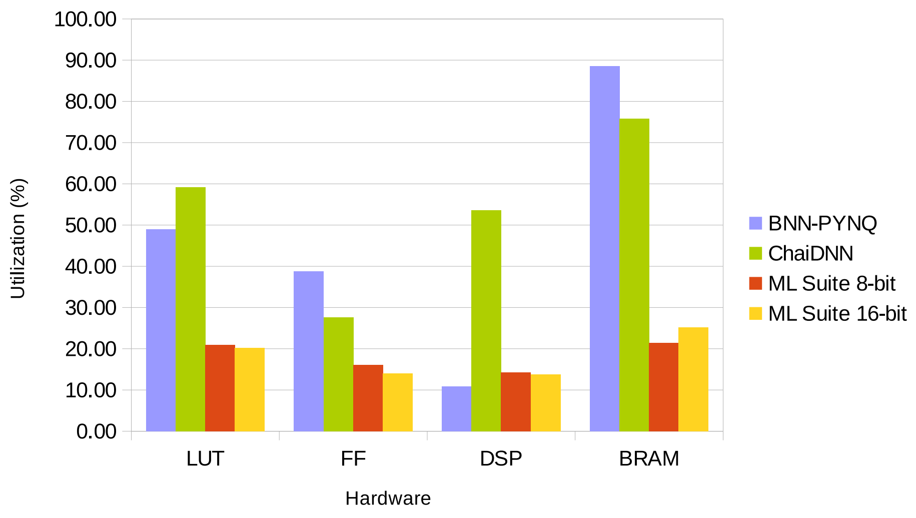


Figure 5.7: Hardware utilization of the examined tools.

5.2.6 Tool power consumption

The power consumption for the GPU inference is 4.5 times higher than for the medium FPGA (CHaiDNN) and 54 times higher than for the small FPGA (BNN-PYNQ). The GPU utilizes more hardware and has a more intensive cooling than FPGAs and thus the power consumption is larger. However, the GPU is used in a data center and can have passive cooling which reduces the load on cooling on the GPU, and thus reducing power required for it. The power consumption is less for Casia245-t1 as it utilizes slightly less hardware than Casia245-t2 and Casia245-t3. The power consumption of the medium FPGA for is the same across all models as CHaiDNN constantly utilizes the same hardware on the FPGA. Casia64 uses the least amount of power as it has less hardware to process the model and passive cooling system, while the medium FPGA and GPU utilize active cooling systems. Since power consumption for the large FPGA (ML Suite) could not be measured only optimistic theoretical value is used. Figure 5.8 shows the power consumed during CNN inference.

5.2.7 Tool throughput and latency

The execution latency of the Casia245 and Casia64 models are shown in Figures 5.9 and 5.10. The throughput is presented in Figure 5.11 and Figure 5.12. BNN-PYNQ 1-bit implementation on the small FPGA provides the lowest latency for Casia64 due to reduced operations performed for the inference. BNN-PYNQ greatly reduces the latency by implementing XOR and shift operations, instead of executing MAC on fixed point model weights. Inference on the GPU has the highest throughput and lower latency compared to other tools. The GPU implementation can exploit high computation parallelism and abundance of hardware to achieve the best results. Additionally,

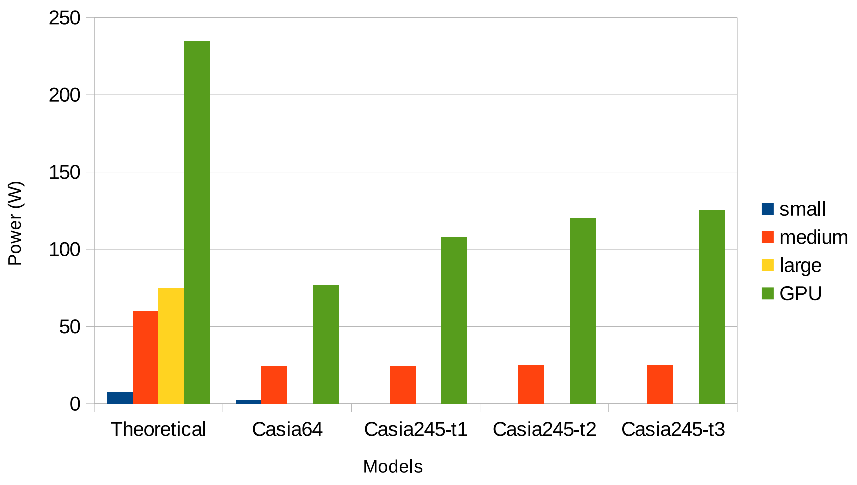
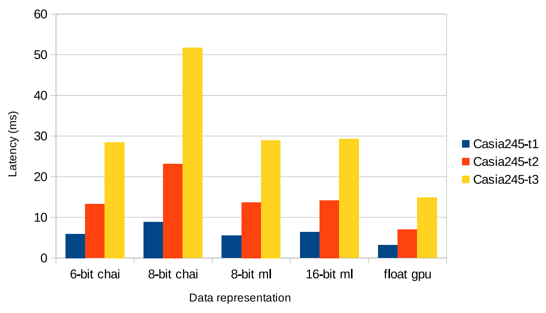


Figure 5.8: Power consumption for the evaluated FPGAs.

the GPU runs on 800MHz while ML Suite runs on 500MHz, CHaiDNN and BNN-PYNQ on 200MHz. CHaiDNN has the worst results as the application is executed at 200MHz and with fixed point numbers can process maximum two images in parallel. Although the GPU can have high throughput, it is not very power efficient. BNN-PYNQ tool, on the other hand, can deliver high throughput at lower power consumption for the small FPGA. CHaiDNN and ML Suite have similar efficiency compared to the GPU, since the tools have small throughput and power consumption. Table 5.3 shows how many images can be processed per consumed power (W).



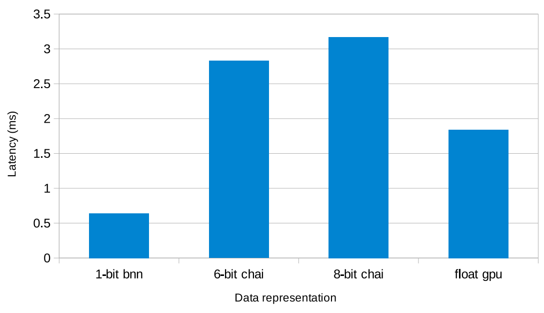


Figure 5.9: Latency in ms to execute an image for the Casia245 models.

Figure 5.10: Latency in ms to execute an image for Casia64.

5.2.8 GOP performance

Table 5.4 shows the number of GOPs that can be performed in a second for the investigated CNN models. The number is calculated as follows:

$$GOP/s = (MACC \times 2 + Comp + Div + Add + Exp) \times throughput$$

Figures 5.13 and 5.14 show how many GOPs per second (GOP/s) can be done for consuming one watt. GOP/s/W can show how efficiently the power is consumed for the inference execution. Casia64 BNN-PYNQ has the best GOP/s/W performance as it uses single bits and XOR operations. Moreover, it is executed on a low power device which improves the performance even

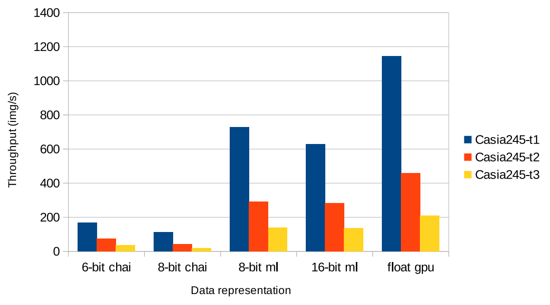


Figure 5.11: Throughput in images per second for the Casia245 models.

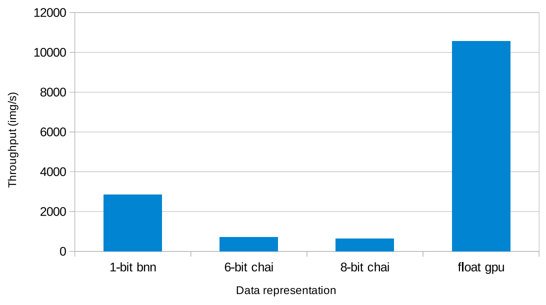


Figure 5.12: Throughput in images per second for Casia64.

Table 5.3: Number of images processed per consumed watt.

	Casia64	Casia245-t1	Casia245-t2	Casia245-t3
1-bit BNN	1431.06	-	-	-
6-bit CHai	29.11	7.02	3.13	1.46
8-bit CHai	25.95	4.67	1.79	0.81
8-bit ML	-	9.71	3.88	1.84
16-bit ML	-	8.39	3.76	1.82
float GPU	87.98	9.42	3.69	1.67

further. CHaiDNN falls behind the other solutions in GOP/s/W most likely because of the addtional overhead for the layer control needed for the inference. ML Suite performs reasonably well assuming the consumed power is capped at 75W. However, theoretically it could be consuming 235W and in that case it would have the worst performance. Although the GPU can perform complex calculations fast, it consumes a significant amount of power and thus has mostly lower performance per watt than the FPGA solutions.

Table 5.4: Number of operations performed in a second for the investigated CNN models.

	Casia64	Casia245-t1	Casia245-t2	Casia245-t3
1-bit BNN	889.81	-	-	-
6-bit CHai	219.91	290.7	447.77	505.92
8-bit CHai	196.07	193.66	256.73	278.82
8-bit ML	-	1257.15	1733.46	1990.28
16-bit ML	-	1086.89	1678.41	1968.53
float GPU	3282.42	2113.6	2853.98	3142.31

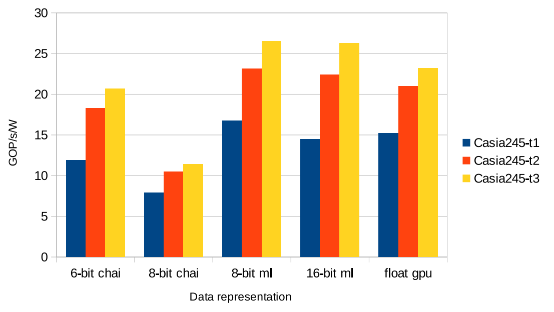


Figure 5.13: Number of GOP that can be performed in a second per consumed watt for the Casia245 models.

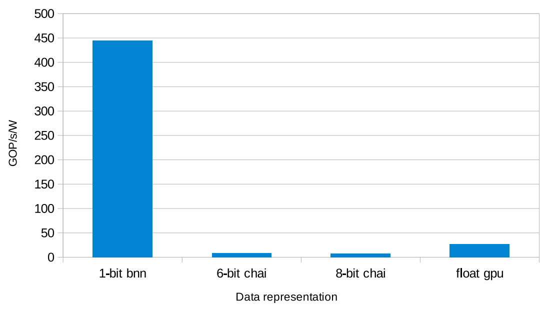


Figure 5.14: Number of GOP that can be performed in a second per consumed watt for Casia64.

5.3 Discussion

Creating a CNN model that can provide acceptable results (90% accuracy or more) is not a trivial task. A clean and large dataset is required as 287 pictures per class on average was not enough. The more there are pictures per classes the better a model can extract features from the classes and thus differentiate them. Fine-tuning layer composition in a CNN model and hyperparameter configurations can be a time consuming task.

Hyperparameters of the training setup can play a crucial role for model ability to learn and training time. Finding an optimal setting for the learning rate, batch size, number of epochs and initial parameters can take time.Learning rates of 0.002 and 0.001 worked best in pair with Stochastic Gradient Descend and no learning decay to start CNN model training. A learning rate of 2e-07 was used for fine-tuning Casia245 models in Caffe as higher learning rates were too unstable for the models to converge. Smaller batch sizes improve the models faster. The experimental models were trained and fine-tuned with the batch sizes set to 128, 16 and 2. The batch size of 16 was used for training the Casia245 and Casia64 models as it provided to be the optimal solution. A precise number of epochs for a model can not be estimated as it depends on the ability of a model to converge. The epochs trained in Keras in Table 5.1 show when each of the models stopped improving the validations loss. Since hyperparameters used for training were the same for all models, the difference in the model composition and the training set had an impact on the number of necessary epochs. 100 would be a good value for the number of epochs to start training new CNN models. Although initial parameters were not extensively tested, Glorot & Bengio initialization had a higher success rate in training than Gaussian initialization.

Increasing CNN complexity does not necessarily improve the accuracy. Casia245-t1 has a better accuracy than Casia245-t3 and was trained faster. Instead of increasing the number of Conv. layers, Dropout or BatchNorm layers can be introduced to improve feature generalization of a CNN model. Adding Dropout layers or BatchNorm for the models increased the training accuracy by 10% but using both did not improve inference results at all. Reducing FC layers for Casia245 from 3 to 1 decreased Top-1 accuracy by 20%.

The results of a model training also depends on a framework used for the training. Caffe failed to train Casia64, Casia245-t2 and Casia245-t3 from random weights and required pretrained models. Keras and Theano frameworks were able to train the models from randomized weights. Setting up models is easier and clearer in Keras than in Caffe. Model layers can be described in a single line in Keras thus all settings can be seen and accessed, while Caffe has a more cumbersome structure.

The limitations of the tools to implement CNNs on FPGAs have an impact on the CNN design as unsupported layers in the tools has to be taken into account. ML Suite could not support several FC layers thus two layers had to be removed form the VGG base model. The lack of documentation of BNN-PYNQ layer synthesis and CNN model design limits the number of models that could be executed by BNN-PYNQ. QNN-PYNQ can quantize weights of models to a fixed point representation and execute them on a Pynq FPGAs. Unfortunately, the tool to quantize custom models for QNN was not publicly available thus this tool was not evaluated.

Implementing a CNN model on FPGA can be simplified by adding compilation and quantization scripts. CHaiDNN and ML Suite had the scripts available which made implementation of the trained models straightforward. Model implementation for BNN-PYNQ requires extensive knowledge about the target FPGA and layer processing. Since the tool does not have a script to compile CNN models, the layers have to be programmed for the FPGA manually. Synthesized new models on FPGAs manually would not be a practical solution as the design could become obsolete due to extensive development in the CNN field. CHaiDNN and ML Suite generalize layer calculation approach at cost of additional hardware thus are more robust to possible changes in CNN model layers.

Chapter 6

Conclusion

In this thesis three implementations of CNN face recognition applications on FPGAs have been presented and analyzed. Although CNN models do not have to be hard-coded like other feature detection algorithms, training a CNN model with proper hyperparameters and dataset can be challenging. Based on the trained CNN models some hyperparameters were defined as a good start for models to learn.

Three tools to port CNNs on FPGAs were analyzed and compared to a GPU implementation. The tools come with limitations on supported CNN model layers, however some can be extended with unsupported layers to be executed on a CPU rather than on an FPGA. The tool setup and porting CNN models on FPGAs can be easy to use, for example synthesizing hardware and running application setup scripts like in CHaiDNN and ML Suite or hard as with BNN-PYNQ. The tools can support various FPGA designs and sizes, however come at the cost of hardware setup and flexibility as smaller FPGA come with stricter requirements. The tool restrictions can affect the performance of CNN models as happened with ML Suite and reduction of Fully-Connected layers. Additionally, weight file format support can affect the design time and results. Since CHaiDNN and ML Suite supported only Caffe models, which failed to learn features, Keras had to be used instead to get successful models. From the performed experiments, the FPGA can achieve accuracy with 10% difference as the GPU implementation. Although the power consumption for the computed operations (GOP/s/W) on CHaiDNN and ML Suite is 1.3× worse and 1.04× better that the GPU respectively, BNN-PYNQ has 16.26× better power efficiency than the GPU.

FPGA solutions are flexible and can be relatively small (hardware), and thus could be implemented as an end device. Quantization techniques (especially 1-bit) can greatly improve inference time with almost no loss in accuracy. However, in general the GPUs provide better execution time at cost of additional power consumption. Additionally, GPUs can perform inference immediately after training in a framework (Caffe, Tensorflow, etc.) but the FPGA porting tools require additional work that includes setting up hardware, software application and performing weight quantization. The GPU solution would be a better choice compared to the investigated FPGA solutions as without any additional porting tools, GPUs can provide a better execution time and accuracy.

Some improvements can be done for the tools to get better results and design flow compared to the GPU solution. CHaiDNN and ML Suite can greatly benefit from implementing binary parameter support. Although it comes with a slight drop in the accuracy, the throughput and latency of execution can be greatly improved. The support of binary values would require rework in Dedicated accelerator and Programmable co-processor architecture structures, as binary values require less memory and can be executed faster. Implementation of streaming architectures (FINN, BNN-PYNQ) is a challenging task as it requires knowledge about the target FPGA hardware. The implementation complexity and design time could be improved with scripts that could generate the architecture from predefined classes of the used layers. The script could receive a CNN model and layer configuration as the inputs to synthesize whole CNN hardware architecture.

Future work on this thesis includes the implementation of the suggested improvements for the tools and investigation of RTL scripts that could implement a streaming accelerator architecture on FPGAs.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, and et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 8

[2] Giovanni Alcantara. Empirical analysis of non-linear activation functions for deep neural networks in classification tasks. *CoRR*, abs/1710.11272, 2017. 5

[3] and, W. Luk, , , , and and. F-cnn: An fpga-based framework for training convolutional neural networks. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 107–114, July 2016. 8

[4] A. Botev, G. Lever, and D. Barber. Nesterov's accelerated gradient and momentum as approximations to regularised update descent. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1899–1903, May 2017. 7

[5] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *CVPR*, 2017. 10, 28

[6] Q. Cao, L. Shen, W. Xie, O. M. Parkhi, and A. Zisserman. Vggface2: A dataset for recognising faces across pose and age. In *International Conference on Automatic Face and Gesture Recognition*, 2018. 19

[7] François Chollet et al. Keras. `https://keras.io`, 2015. 22

[8] D. Ciregan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, June 2012. 1

[9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, abs/1511.00363, 2015. 10, 13

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009. 4

[11] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi. Caffeinated FPGAs: FPGA framework for convolutional neural networks. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 265–268, Dec 2016. 9, 14

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`. ix, 3, 5

[13] Yandong Guo, Lei Zhang, Yuxiao Hu, Xiaodong He, and Jianfeng Gao. MS-Celeb-1M: A dataset and benchmark for large scale face recognition. In *European Conference on Computer Vision*. Springer, 2016. 19

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. 20

[15] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. 20

[16] C. Huang, S. Ni, and G. Chen. A layer-based structured design of CNN on FPGA. In *2017 IEEE 12th International Conference on ASIC (ASICON)*, pages 1037–1040, Oct 2017. ix, 3, 4, 14

[17] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007. 19

[18] Qiangui Huang, Shaohua Kevin Zhou, Suya You, and Ulrich Neumann. Learning to prune filters in convolutional neural networks. *CoRR*, abs/1801.07365, 2018. 9

[19] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016. 20

[20] Intel. *Core i7-9700K*, 2 2018. 9

[21] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. 7

[22] Fares Jalled. Face recognition machine vision system using eigenfaces. *CoRR*, abs/1705.02782, 2017. 1

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. 8

[24] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, and et al. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017. 1, 9

[25] R. E. Kalman. A new approach to linear filtering and prediction problems. *ASME Journal of Basic Engineering*, 1960. 1

[26] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466, 09 1952. 7

[27] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. 7

[28] Kevin Kiningham. Design and analysis of a hardware cnn accelerator. 2017. 15

[29] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009. 4

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. 20

[31] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. A 7.663-tops 8.2-w energy-efficient FPGA accelerator for binary convolutional neural networks. *CoRR*, abs/1702.06392, 2017. ix, 7, 13

[32] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, 2015. 19

[33] Microsoft. Mmdnn. `https://github.com/Microsoft/MMdnn`, 2019. 43

[34] H. Nakahara, H. Yonekawa, and S. Sato. An object detector based on multiscale sliding window search using a fully pipelined binarized CNN on an FPGA. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 168–175, Dec 2017. 13

[35] Nimbix. Nimbix jarvice cloud service, 2018. 19

[36] NVIDIA. *Tesla K40M*, 11 2013. 8, 9

[37] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018. 5

[38] T. B. Preuer, G. Gambardella, N. Fraser, and M. Blott. Inference of quantized neural networks on heterogeneous all-programmable devices. In *2018 Design, Automation Test in Europe Conference Exhibition*, pages 833–838, March 2018. 14

[39] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016. 9

[40] Joseph Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016. 8

[41] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015. 20

[42] Sean O. Settle, Manasa Bollavaram, Paolo D'Alberto, Elliott Delaye, Oscar Fernandez, Nicholas Fraser, Aaron Ng, Ashish Sirasao, and Michael Wu. Quantizing convolutional neural networks for low-power high-throughput inference engines. *CoRR*, abs/1805.07941, 2018. 10, 23

[43] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to FPGAs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016. 15

[44] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. 20

[45] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. 7

[46] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016. 20

[47] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. 20

[48] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. 8

[49] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 65–74. ACM, 2017. 9, 13, 19

[50] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–I, Dec 2001. 1

[51] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. 14

[52] Xilinx. *ZCU102*, 11 2016. 9

[53] Xilinx. A HLS-based Deep Neural Network Accelerator library for Xilinx Ultrascale+ MPSoC devices, 2018. 14

[54] Xilinx. Xilinx ML Suite tool description, 2018. 15

[55] Dong Yi, Zhen Lei, Shengcai Liao, and Stan Z. Li. Learning face representation from scratch. *CoRR*, abs/1411.7923, 2014. 19

[56] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM. 9, 14

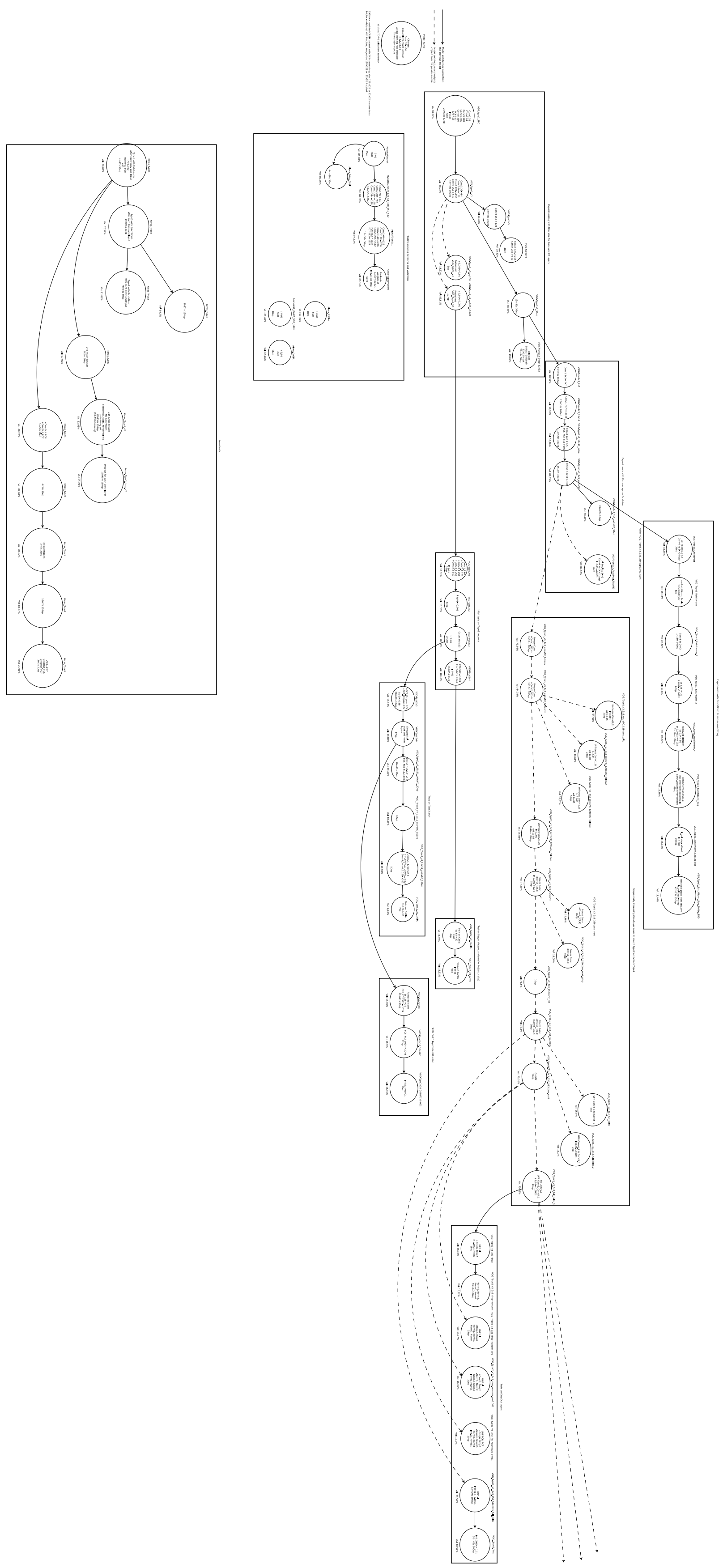
Appendix A

Training details

This appendix presents the details of CNN model design and training setup performed for the evaluated tools and CNN models.

Initially for the CNN model training Caffe and Theano frameworks were used. Tools CHaiDNN and ML Suite are designed to work with caffemodel files thus all models were trained in Caffe. However, because BNN-PYNQ works with Theano files, Casia64 had to be also trained in Theano. The training procedure for the models used in CHaiDNN and ML Suite was mostly based on the VGG paper with some modifications. The training starts with Casia245-t1 that used SGD optimizer with a weight decay, learning rate of 0.001, batch size of 128 and initial random variables based on Gaussian distribution. After 100 epochs the network had validation accuracy of 85% and started to overfit the training set. Then Casia245-t2 was trained with the same initial parameters and random variables, however after 100 epochs there was no progress in training. The learning rate was varied from 0.1 to 0.00001 but in both cases there were no meaningful results. SGD was replaced with Adam, however the model still was not able to converge after 100 epochs. Since Casia245-t1 model was able to extract features, it was taken as a base for Casia245-t2. The new Conv. layers were iteratively added one by one on top of Casia245-t1. The iterative Conv. layer addition was done as follows: first a new Conv. layer is added and the learning modifiers of the previous layers were set to 0 to ensure that only the new layer learns features. The model with the new layer was trained for 50 epochs. After that learning modifiers for all layers were enabled (set to 1) to calibrate the whole model. Then the process was repeated until all Conv. layers for Casia245-t2 model were added. At the end of the process Casia245-t2 had accuracy of 78% on the validation set. The same process was employed to make Casia245-t3 where Casia245-t2 was taken as the base. However, this time after six iterations there was no progress in learning and the model was not able to converge. Changing the model and its hyperparameters in Caffe did not give any meaningful results for Casia245-t3, thus another framework was used. As tools CHaiDNN and ML Suite require .caffemodel files the output file of the new framework should be compatible with or converted to caffemodel format. Unfortunately, Theano output files could no be converted to caffemodel format, so Tensorflow with Keras API was used instead as MMdnn [33] tool could convert Keras output files to caffemodel format. Keeping the same parameters and training from randomly initialized weights by Glorot, Keras proved to be successful with all models. MMdnn tool was able to convert weights of Conv. layers as it mixed the weights for FC layers. The converted caffemodels had to be recalibrated to ensure that the FC layers have proper weights. The recalibration was done in two steps. For the first step the Conv. layer learning modifiers were disabled, so that only the FC layer weights are adapted. The learning rate was set to 2-7e as with higher rates the models would not be able to learn. In the second step Conv. layer learning modifiers were enabled to recalibrated the models. The Casia64 model for BNN-PYNQ was trained in Theano with the same learning parameters as Casia245. As Casia64 is a much smaller model than the others it was recalibrated for 200 epochs.

Figure A.1 shows the steps performed for creating Casia245 models in Caffe and Keras. Each circle repsents a trained model and the difference to the previous models is written in the center of a circle.



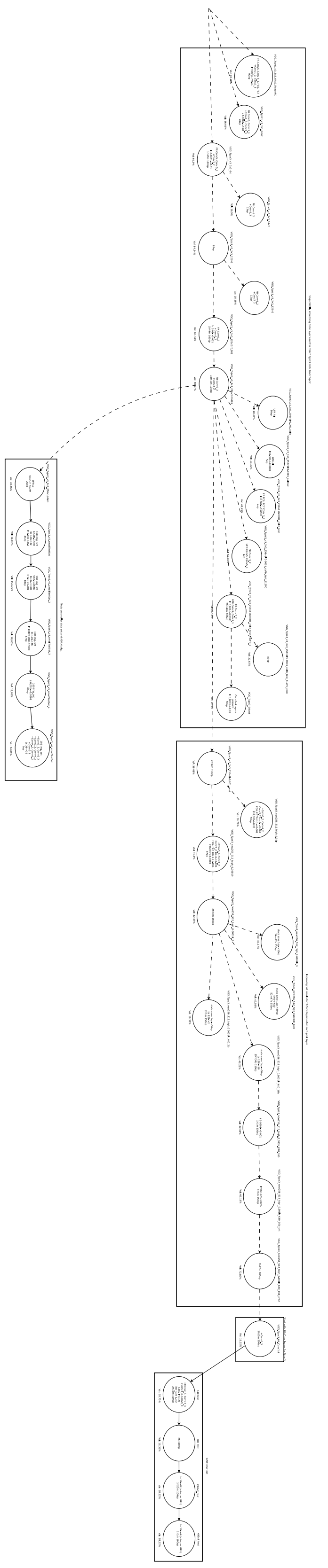


Figure A.1: Steps performed to create Casia245 model types.