UCLouvain

École polytechnique de Louvain

Accelerating Ternary Quantized Convolutional Neural Networks using OpenCL for FPGA

Authors: **Victor Joos de ter Beerst, Antoine Vanderschueren**
Supervisors: **Christophe De Vleeschouwer, Jean-Didier Legat**
Reader: **David Bol**
Academic year 2018–2019
Master [120] in Computer Science and Engineering

Abstract

FPGAs balance the reprogammability of CPUs and the performance of ASICs. They seem the perfect solution to increase the throughput of neural networks.

However they must also prove to be highly competitive in a market dominated by GPUs. To achieve this, we focus on the strength of FPGAs that cannot be taken advantage of on GPUs. Extreme quantization and sparsity are both not suited for acceleration on GPU.

We use a quantized version of ResNet assembled from uniform building-blocks in order to achieve better area utilization on FPGA. We manage approximately 2.5% accuracy loss when compared to a floating-point model, while introducing highly quantized weights and activations. Our final network uses ternary weights and 4-bit activations, in addition to shift-based batch normalization.

With the use of OpenCL for high-level synthesis, we implement loop tiling, loop unrolling and loop interchange to speed up the convolution operation of our streamlined model on the Cyclone V FPGA. Using ternary weights, we are able to remove multiplications and replace them with simple branching, effectively getting rid of the need for DSPs. Our final FPGA implementation achieves a latency of 17ms per image using a quantized residual network.

Acknowledgments

We would like to start by thanking our two supervisors, Christophe De Vleeschouwer and Jean-Didier Legat, for their continued support during this master's thesis. The advice and guidance they provided, in addition to the interest for the topic they brought, was invaluable to us. We'd also like to thank them for their time and the constructive remarks that shaped this text, and changed the way our research was performed.

We'd also like to thank Martin Lefebvre, Simon Carbonnelle, and Charlotte Frenkel who provided helpful insights and helped us find a direction to take our work in.

We'd like to thank Marc Vanderschueren and Anne-Sophie Collin for proofreading our work, to ensure readability and coherency throughout.

Finally, we'd go amiss to mention our family and friends, and in particular Laura and Emmanuelle, for supporting us through thick and thin.

Contents

List of Figures and Tables

List of Abbreviations

ALM	Adaptive Logic Module
ALUT	Adaptive Look-Up Table
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BN	Batch Normalization
BNN	Binary Neural Network
BWN	Binary Weight Network
CNN	Convolutional Neural Networks
DA	Data Augmentation
DSP	Digital Signal Processing
FF	Flip-Flop
FFT	Fast Fourier Transform
fmap	feature map
FPGA	Field-Programmable Gate Array
GEMM	General Matrix Multiplication
GOPS	Giga-Operations Per Second
HLS	High-Level Synthesis
LE	Logic Element
LR	Learning Rate
MAC	Multiply-and-Accumulate
MSE	Mean-Square Error
PE	Processing Element
ReLU	Rectified Linear Unit
ResNet	Residual Network
RTL	Register-Transfer Level
SBN	Shift-based Batch Normalization
SGD	Stochastic Gradient Descent
SIMD	Single Instruction Multiple Data
SOTA	State-Of-The-Art
SWI	single-work-item
tanh	Hyperbolic Tangent
TNN	Ternary Neural Network
TWN	Ternary Weight Network

Chapter 1

Introduction

After a decade during which machine learning experienced a lack of innovation, the field is again in full expansion. "Deep Learning" and its related deep neural networks are taking over the field. The accuracy gained in various tasks is impressive. More than 95% top-5 accuracy was achieved on the ImageNet dataset for image classification using Convolutional Neural Networks (CNNs) in 2015 [1].

The need for large computational resources to access these accuracy gains constitutes the biggest drawback of such methods. This need means that the latency and the throughput of these models will suffer, unless a solution is found to reduce the computational costs.

One possible solution is the field-programmable gate array (FPGA). Balancing the reprogrammability of CPUs and the performance for specific designs of Application-Specific Integrated Circuits (ASICs), they seem to be the perfect solution to increase the throughput of neural networks.

However they must also prove to be highly competitive in a market dominated by Graphical Processing Units (GPUs). The current generation of GPUs is made for parallel floating-point computations, but they are limited to designs where every processing element (PE) achieves the same task.

The focus should therefore lie on the strengths of FPGAs that cannot be taken advantage of on GPUs. Reducing bit-width through extreme quantization is not efficiently exploitable on GPU because the hardware it contains is focused on extremely parallel floating-point operations. It also often doesn't make sense to rely on the sparsity of models on GPU, due to complex branching decreasing the performance of an implementation [2, 3].

The core of this work concentrates on the one hand on building models for image classification which take advantage of the specifics of FPGA devices, while being built from a generic CNN model, and on the other hand on the acceleration of convolutional layers. In the future, it should be possible to take a model definition and retrain the model to be optimal for our FPGA implementation. To show that this is achievable, a standard residual network (ResNet) [1] has been used, quantized and assembled from uniform building-blocks to allow better area utilization on the FPGA. The goal was to achieve no worse than 3% accuracy loss on the CIFAR-10 classification task, compared to a floating-point model, while introducing highly quantized weights and activations.

Initially, the focus was put on training state-of-the-art (SOTA) residual networks with different quantization approaches. Models with weight quantizations from 1 bit to 4 bits, and activation quantizations from 1 bit to 32 bits were developed. It was found by analyzing different quantization strategies that, in general, quantization is a viable approach for neural networks, and in particular ternary networks (constrained to values -1, 0, 1) can achieve near state-of-the-art performance on the CIFAR-10 dataset.

Secondly, the ResNet was modified to accommodate an easier design on FPGA. The network was restructured to only contain blocks with a convolution, a batch-normalization (BN), and an activation. All BN layers were changed to shift-based batch-normalization, to use fewer logic elements.

The quantized and modified model lost less than three percentage points, compared to a full floating-point model, while reducing the size of the weights by 16, and allowing faster computations. This result brings the conclusion that the implementation of a generic float-to-quantized framework is a worthwhile goal to pursue.

After building a quantized neural model, OpenCL [4] was used as a high-level synthesis (HLS) tool to create an implementation for FPGA. OpenCL allowed a faster development cycle, in addition to easier hardware-software co-design. The implementation of several promising solutions was possible due to OpenCL.

The first implementation on FPGA used floating-point operations, and served as a baseline implementation to test further optimizations, and to assess the advantage of quantization. The use of loop tiling, loop unrolling and loop interchange was necessary to speed up the convolution operation. The fully-connected layer was not accelerated due to the architecture of the ResNet only containing a small fully-connected layer, which required only 1280 operations, compared to more than 80M operations for the convolution.

After building a baseline implementation, the focus was on optimizations to the convolution operation. With the optimized convolution, we modified the kernel to use ternary weight, 4-bit activation networks, following the results of our quantized training research. Using ternary weights allowed removing multiply-and-accumulate (MAC) logic and replacing it by simple branching and accumulation.

After optimization, a thorough analysis of the benefits and drawbacks of every optimization technique is provided. The implementation on FPGA was mostly bottlenecked by the OpenCL host-to-kernel communication.

The final FPGA implementation achieves a latency of 17ms per image, using a quantized network. Further work optimizing the number of kernels and tasks could achieve a latency of 3.7ms (or 6.3ms if no accuracy drop is desired) with the current design.

Structure of the Thesis

This master's thesis is divided into three parts. Each part contains a high-level overview of its contents and then goes into further details the more the text progresses.

The first part details the background information necessary as context for this thesis. In Chapter 2 we first explain the theory behind convolutional neural networks (CNN) and different quantization approaches. Chapter 3 explains common FPGA designs, and

the OpenCL design goals and common patterns. This chapter also details the workflow of using OpenCL for FPGAs.

The second part puts forward related work, and positions our work in comparison to the state of the art. We divide this part into two chapters that focus on the one hand on training convolutional models and on the other hand on FPGA (and ASIC) implementations of neural networks. Chapter 4 expands on the concepts of quantization, and the different ways of training neural networks. Chapter 5 discusses both algorithmic optimizations and datapath optimizations that are possible when accelerating convolutional neural networks. In this latter chapter, we focus mainly on the convolutional layer but also present research which accelerates other parts of neural networks.

The third part explains our work in as much detail as possible, while we focus on the important differences with other work. As in the other two parts, we divide our presentation in training and inference, detailing every step of our approach and supporting our work with concise intermediate results. We finish each chapter with a discussion of the results relating to quantization or FPGA acceleration, while providing possible improvements according to our results.

Part I

Background

Contents

This part is divided into two chapters. The first chapter contains background knowledge on Convolutional Neural Networks (CNN) and quantization that will be necessary to understand the problems faced in designing a network suitable for FPGA.

The second chapter describes the OpenCL API and language as well as the design goals that make it a good fit for designing neural nets with it. This chapter also gives some background on the purpose of FPGAs, in particular when they are used to accelerate neural network designs. We also expand on design architectures that will be used in the other parts of this text. In particular, tiling and unrolling of the convolution operation is discussed.

Chapter 2

Convolutional Neural Networks

Neural networks and specifically convolutional neural networks are field-defining algorithms for image classification and have many other uses in the field of artificial intelligence. In this section we'll introduce the basic internal mechanics of these networks as well as the parts we'll be using in our experiments.

The first section discusses the basic building blocks needed to construct a neural network, starting from a historical perspective, and showing the multiple improvements that can be made by carefully designing layers that will interact with each other.

The second section focuses on finding the correct parameters for a model, by training the networks in a supervised setting. The theory behind back-propagation using gradient-descent and its alternatives will be expanded upon.

The third section will use the basic building blocks from section 2.1 to build complete networks. We focus on the three following: AlexNet, VGG, and residual networks (ResNets).

The last section of this chapter explains the theory of quantization, in particular regarding quantization in the domain of convolutional neural networks.

2.1 Layers

Layers are collections of weights and functions that need to be applied on an input. In this section, we detail the layers needed to build an effective convolutional neural network. To explain the theory behind the layers of a deep network, we start by explaining the basic building block of a neural network, the perceptron.

2.1.1 Perceptron

The perceptron was first proposed by F. Rosenbalt [5] more than half a century ago. This algorithmic building block was inspired by neuro-science. A neuron receives multiple electrical signals from neighboring neurons and integrates them. If this integration exceeds a certain threshold, the neuron will 'fire', sending an electrical signal that will be transmitted to connected neurons.
Transforming this into computer language we get a vector input ($\mathbf{x}$) whose elements will be multiplied by a certain value. These values are called weights ($\mathbf{w}$). The results of these operations are then summed up and if this sum is greater than zero the perceptron will be 'activated', meaning it will have a value of one instead of zero (fig 2.1). We can also add a constant term called bias which will act to lower/raise the bar for the

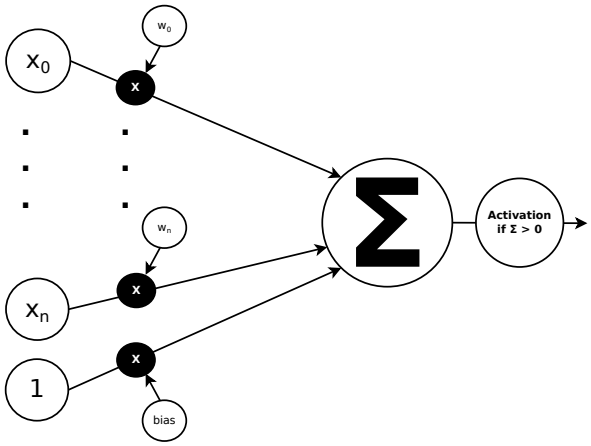


Figure 2.1: Perceptron architecture

activation of the perceptron. All this can be written in vector form:

$$h(x) = \begin{cases} \text{if } \mathbf{w^T} \cdot \mathbf{x} + b > 0 \text{ then} & 1 \\ \text{otherwise} & 0 \end{cases} \tag{2.1}$$

In the following sections we'll explore how we can make perceptron layers (sect. 2.1.2), computationally extend them (sect. 2.1.3) and modify the activation.

2.1.2 Fully Connected/Dense

We saw that a perceptron takes multiple inputs and generates one output. If we want multiple outputs we could just stack them until we obtain the wanted dimension. This is exactly what a fully connected layer is. Its name is derived from the fact that every input node is connected to every output node and vice-versa. Each perceptron has its own weights vector as can be seen in fig 2.2. We suppose $\mathbf{w}$ is a matrix containing all the weights with $\mathbf{w_{i,o}}$ the weight linking input neuron i with output neuron o. This leads us to:

$$\mathbf{y} = \mathbf{w^T} \cdot \mathbf{x}$$

$$\Leftrightarrow \forall o \in \{1, 2, .., N_{out}\} : y_o = \sum_{i=1}^{N_{in}} w_{i,o} \cdot x_i$$

2.1.3 2D Convolution

This layer is the one that greatly improved image recognition tasks. While the idea dates back to 1998 and the Lenet-5 architecture proposed by Lecun et al. [6] it was in 2012 that it came back to light when Alexnet [7] took home the prize for the most accurate image recognition and classification network at the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). It achieved accuracies of 37.5% (top-1) and 17.0% (top-5). Improving by nearly 10% over the previous results.

A convolution operation happens as follows. Suppose we have an image ($Nrow \times Ncol$) and a kernel ($Nk \times Nl$). A sliding window will now go over the complete image and take out chunks of the same size as the kernel. That chunk is then multiplied element-wise with that kernel, giving us $Nk \times Nl$ terms which are then summed. The

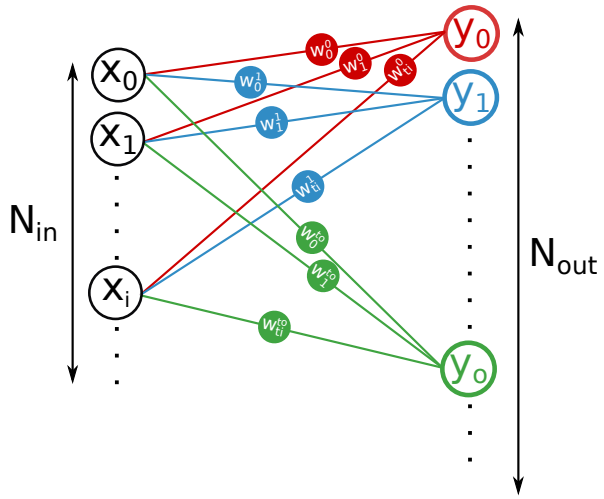


Figure 2.2: Fully connected layer (without bias)

new coordinates of this sum are based on the center coordinates of the sliding window. Unless the kernel is of size 1×1 there will be a dimensional reduction between the input image and the output. To counter this we can add padding (in our case we'll always add zeros) as you can see in fig 2.3.
The sliding window doesn't always have to go over every element of the input. Strides will define how many pixels will be skipped each time the window moves. Suppose we have strides of 2 in both x and y dimensions (row and column). If we use padding, we'll end up with an output that has exactly 4 times fewer pixels.

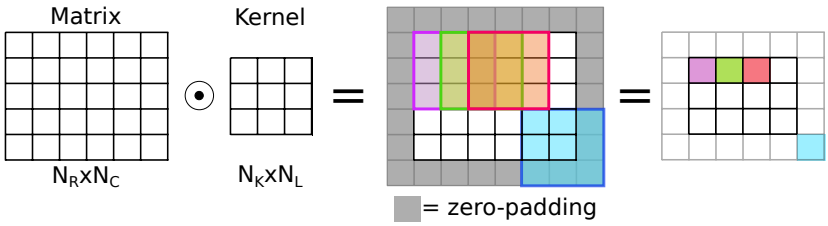


Figure 2.3: Simple convolution with padding (translucent and grey squares)

Instead of having Nin inputs and $Nout$ outputs like a dense layer, there are Nin input *channels* and $Nout$ output *channels*. Each input channel has the same size $(Nrow, in \times Ncol, in)$ and so does each output channel $(Nrow, out \times Ncol, out)$. In order to generate 1 output channel we need 1 unique kernel per input channel. The input channels are convolved with their respective kernel and the resulting Nin convolutions are then summed up to generate the output channel. Now to obtain $Nout$ output channels we'd need $Nin \times Nout$ kernels as can be seen in fig 2.4.

$$\forall o \in \{1, .., Nout\}, or \in \{1, .., Nr\}, oc \in \{1, .., Nc\} :$$
$$\mathbf{Out}[o, or, oc] = \sum_{i=1}^{Nin} \sum_{k=1}^{K} \sum_{l=1}^{L} \mathbf{W}_{i,o}[k, l] \cdot \mathbf{In}\left[i,\ or \cdot S - \left\lfloor \frac{K}{2} \right\rfloor + k,\ oc \cdot S - \left\lfloor \frac{L}{2} \right\rfloor + l\right]$$

The matrix that composes a channel is more commonly referred to as a *feature map* (fmap). The ensemble of kernels are called the weights of a convolutional layer and are of

size $Nin \times K \times L \times Nout$ (not counting the bias-terms which are of size $Nout$). Compared to a fully connected layer, the total number of weights is way smaller: for a similarly sized input and output we'd need $Nin \times Nrow, in \times Ncol, in \times Nrow, out \times Ncol, out \times Nout$ weights and would need to trust the neural net to figure out the spacial relationship between the inputs. The convolution does however come at a computational cost which is $K \times L$ times larger.

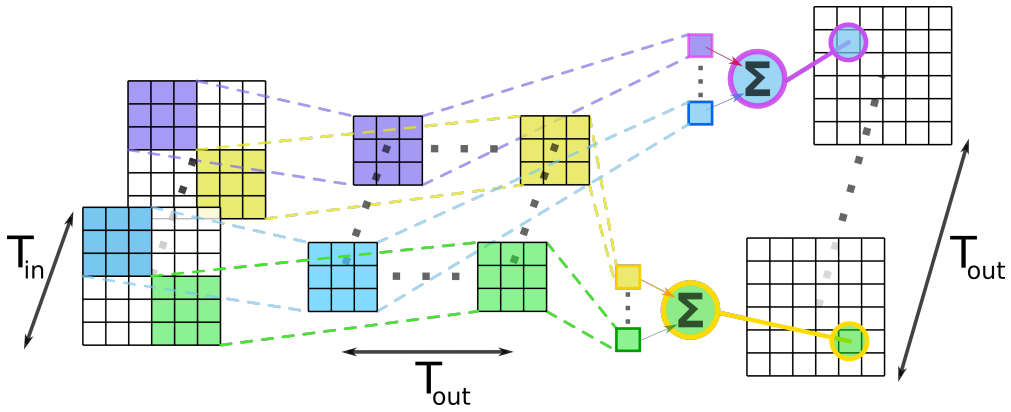


Figure 2.4: Complete 2D convolution

2.1.4 Activations

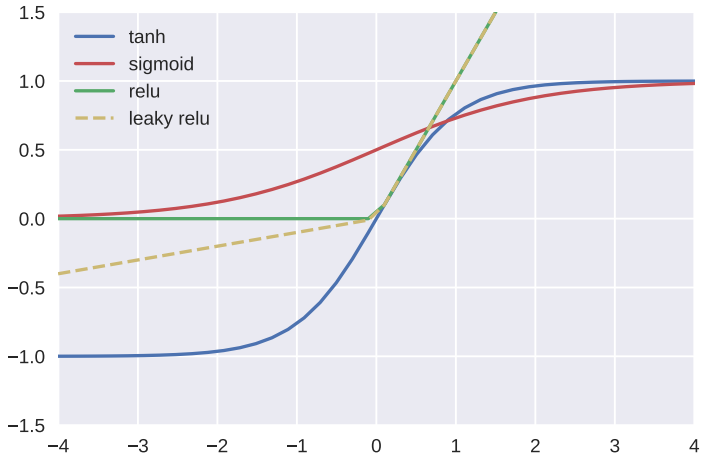


Figure 2.5: The different activation functions.

After each convolutional layer and fully-connected layer, an activation function is used to be able to solve nonlinear problems. Different activation functions have different properties, the most important one being easy to differentiate.

Hyperbolic Tangent (Tanh) and Sigmoid

The sigmoid and tanh functions are both smooth function which have the following relationship:

$$\tanh(x) = 2\sigma(2x) - 1 \tag{2.2}$$

The two functions can be seen in Figure 2.5. These activations are called saturating since they 'squeeze' the complete $\mathbb{R}$ range in a smaller one, respectively $[0, 1]$ (sigmoid) and $[-1, 1]$ (tanh).

Rectified Linear Unit (ReLU) and Derivatives

With the advent of deep networks, it was found by [7] that a better activation function was the ReLU, which can be seen in Figure 2.5. They also showed that training was much faster with this activation. This function is differentiable everywhere but at zero, which is enough for the case of gradient descent. Contrary to the 2 previous ones it is also non-saturating since only half of the initial domain is kept.

The problem with the ReLU function is that with a large gradient it could be possible that a weight never activates any neuron again. To solve this problem, we can use a leaky ReLU, which is again linear on the positive side, but instead of being zero on the negative side, it has a slow downward slope.

2.1.5 Pooling

To be able to reduce the size of the feature maps, it might be advantageous to apply a pooling layer after a convolutional layer. Different types of pooling layers exist, but the two main pooling operations are average pooling, and max pooling. The pooling layer reduces the size of the feature maps without reducing the number of channels.

Average pooling takes the average over a block from every channel, and aggregates the average into one value. Max pooling does the same for the maximum value. Figure 2.6 shows an example of both operations.

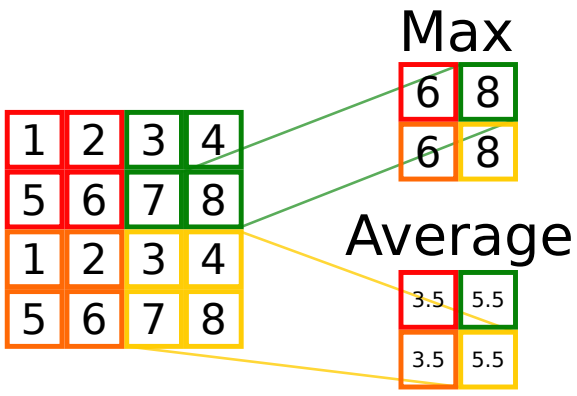


Figure 2.6: Pooling layer. The top-right image shows the max pooling, while the bottom-right shows the average pooling.

2.1.6 Batch Normalization

Recently, faster and more robust training was implemented by [8], through batch normalization. This approach consists of reducing internal covariate shift by applying

normalization on each mini-batch. The equations are as follows:

$$\mu_b = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.3}$$

$$\sigma_b^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_b)^2 \tag{2.4}$$

$$\hat{x}_i = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}} \tag{2.5}$$

$$y_i = \gamma\hat{x}_i + \beta \tag{2.6}$$

The parameters to be learned are γ and β. (2.3) is the mean of the mini-batch, (2.4) the variance, (2.5) the normalization, and finally (2.6) scales and shifts the normalization through the learned parameters.

2.2 Training

While the design of a model is important, the training methods used to find the weights is perhaps even more important. In this section, we detail the different steps of training that are required in order to get state-of-the-art accuracy on a convolutional neural network.

2.2.1 Initialization

When training a neural network, the first step is initializing the weights. Multiple methods exist but the most common is random initialization, where every weights is given a value chosen at random from a certain range and a certain distribution. In this thesis we tested two initializers: Glorot et al. [9] uniform initialization, which draws the weights from a uniform distributions around zero with limits at $\sqrt{6/(\text{fan-in} + \text{fan-out})}$ where fan-in (resp. fan-out) is the number of input (output) units in the weight tensor, and He et al. [10] normal initialization which uses a normal distribution with mean at zero and standard deviation equal to $\sqrt{2/\text{fan-in}}$.

2.2.2 Forward propagation

Once the weights are initialized the training data is used as input to the model and propagated through the different layers of the neural network, using the current weights. The forward propagation will result in an output vector which we can compare to the target vector (the label) using a loss function, in our case a categorical cross-entropy, which is defined as:

$$CCE(p, q) = -\sum_{x} p(x)\log q(x) \tag{2.7}$$

Another loss function that is often used is the mean-square error:

$$MSE(p, q) = \frac{1}{m}\sum_{x}(p(x) - q(x))^2 \tag{2.8}$$

This loss function must be minimized by changing the weights in the right way. A solution to the problem of minimizing the loss function will be described next, using gradient descent during back propagation.

2.2.3 Back-propagation

Multiple methods of back-propagation exist today in the literature, with an overview given by [11]. They all derive from the original idea of gradient descent which certainly dates back to the 1960s [12] if not sooner by Cauchy or Hadamard [13]. The original algorithm comes with multiple problems that are solved with subsequent improvements. We describe here the original algorithm and important improvements leading to Adam [14], the optimizer used in the training of our neural networks.

Gradient Descent

When describing gradient descent, we should be careful in separating three different algorithms that could claim the name, namely the original gradient descent, stochastic gradient descent, and (mini-)batch gradient descent [11].

Gradient descent (GD) uses all the data in one pass, and computes the gradient/Jacobian of the loss function (the partial derivative in all principal directions):

$$\mathbf{w} = \mathbf{w} - \alpha \frac{\partial L(\mathbf{p}, \mathbf{q})}{\partial \mathbf{w}}\bigg|_{\mathbf{w}(t)} \tag{2.9}$$

In practice, this means that for each weight, we check the direction which minimizes the loss and update that weight in that direction.

Gradient descent has the drawback of needing the complete dataset, which is both a problem for online learning and for memory usage, when the entire dataset is too big. Thus computation speed will be bottlenecked by disk speed.

Stochastic gradient descent (SGD) instead focuses on updating the weights using only one sample of the dataset at a time. This yields the following formula:

$$\mathbf{w} = \mathbf{w} - \alpha \frac{\partial L(p, q)}{\partial \mathbf{w}}\bigg|_{\mathbf{w}(t)} \tag{2.10}$$

SGD changes the weights with a much higher variance than GD. Due to this higher fluctuation, it takes more time to reach a local minimum than GD, but it has a higher probability of escaping a local minimum to find a potentially better one. When slowly decreasing the learning rate over time, SGD and GD have a similar convergence pattern.

Finally, batch gradient descent takes the best of both worlds by using batches of the dataset on which to perform gradient descent. In doing so, it reduces the variance, leading to a more stable convergence. This type of gradient descent is also often called SGD, as will be the case in the following text.

All these methods suffer from poor convergence. First a correct learning rate is hard to find, too small and convergence will be slow, too large and the loss function will fluctuate around the minimum. Second, having a single learning rate for all weight updates might not be optimal. Third, it is important to avoid getting trapped in a local minimum, in order to find the true optimal value.

Adaptive Learning

The previous challenges of training with SGD have pushed researchers to find new methods of gradient descent : SGD with Momentum, AdaGrad, Adadelta, RMSProp, and finally Adaptive Moment Estimation (Adam). Adam adds the following improvements to SGD (adding on top of RMSProp and Momentum): It computes an adaptive

learning rate for *each* parameter. It keeps a decaying average of both past gradients (m_t) and past squared gradients (v_t)

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.11}$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2.12}$$

Because both moments are biased towards zero, they compute $\hat{m}_t$ and $\hat{v}_t$ which are bias-corrected versions. Finally, the update rule is as follows :

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{2.13}$$

β_1, β_2 and ϵ are meta-parameters of the Adam method.

2.2.4 Regularization

To avoid having parameters which tend to infinity, it makes sense to regularize them by adding a term to the loss function [15]. Two common types of regularization are possible: L_2-norm and L_1-norm regularization. (2.14) shows the L_2 regularization, and (2.15) shows the same for L_1.

$$\frac{1}{2} \lambda w^2 \tag{2.14}$$
$$\lambda w \tag{2.15}$$

λ is a constant that is in the order of 10^{-4}. The L_1- and L_2-norms can be combined to form "Elastic Net Regularization" [16].

2.3 Deep Network Layouts

Different network layouts have both benefits and drawbacks. In recent years, deep architectures have made their appearance, due to systems with more computational power and more memory. In this section, we highlight a few deep neural networks, which have proven to be quite accurate on the ImageNet dataset.

2.3.1 AlexNet

Deep convolutional networks started with the paper by Krizhevsky et al. [7] with a network dubbed AlexNet. The network contains 8 learned layers, 5 convolutional and three fully-connected layers. The network layout can be seen Figure 2.7.

2.3.2 VGG

After the success of AlexNet's deep network, researchers tried optimizing parts of the network, to achieve better performance at a lower computational cost. The result of this work is the VGG network [17], a feed-forward (linear) network with one main difference to AlexNet.

VGG networks use multiple convolutional layers with kernels of size 3x3 to emulate layers with greater kernels. For example 3 layers with 3x3 kernels have the same receptive field as 1 layer with a 7x7 kernel. The advantage of this approach is that those

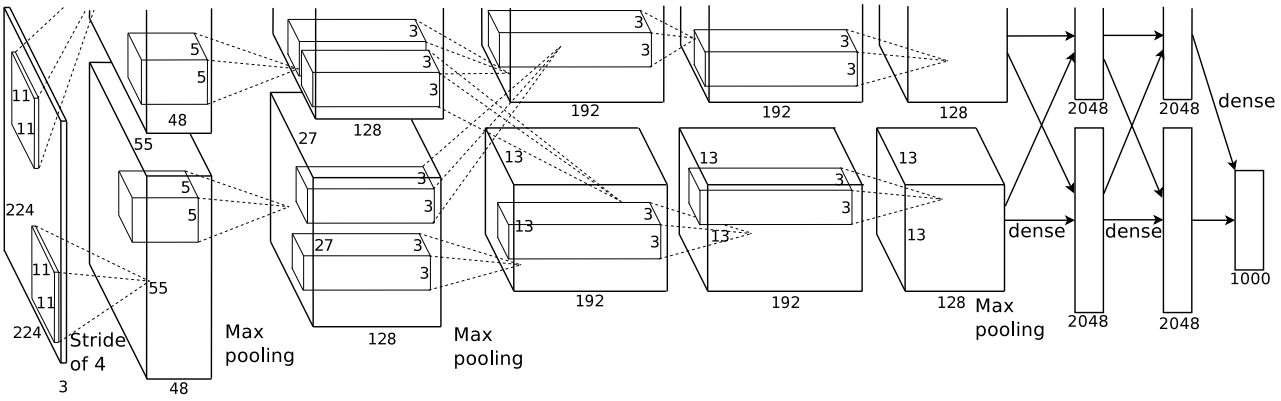
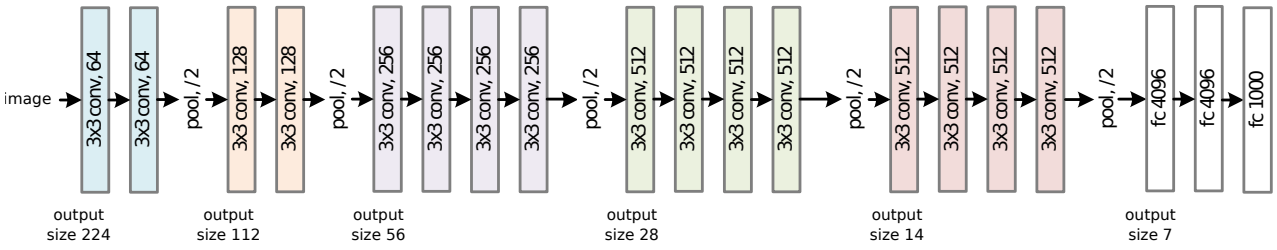


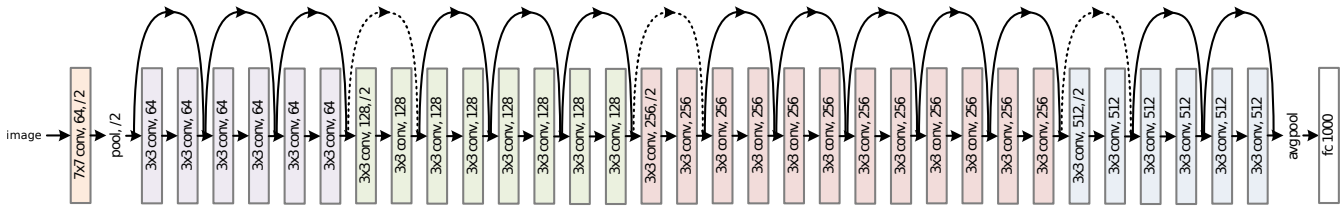
Figure 2.7: AlexNet network layout [7]

three layers have less parameters than the 7x7 kernel layer, having only $3 \cdot (3^2C^2) = 27C^2$ parameters compared to $7^2C^2 = 49C^2$ parameters. Another advantage is due to the non-linear activation function between every convolutional layer, which "makes the decision function more discriminative" [17].

The typical VGG-like network will have a succession of convolutional layers followed by a few fully-connected ones. The convolutional layers are followed by max-pooling layers in order to reduce the feature map dimensions. Those are often put right after convolutions that augmented the number of channels. A full VGG network is shown in Figure 2.8a.



(a)



(b)

Figure 2.8: Deep neural networks. (a) shows a VGG-19 architecture. (b) shows a residual network with 34 parameter layers. Both images adapted from [1]

2.3.3 Residual Networks

Several problems still plague deep feed-forward networks, one being that deeper network don't always display better performance, not even for the training set. Intuitively, a deeper network should always be able to have similar or better performance than a

smaller network, because it is always possible to map a deeper model to a smaller one by having some layers be the identity function. This is why ResNets [1] try to take advantage of this property. Instead of hoping to achieve this identity function by having the weights go to 1, they optimize the network for the difference between the optimal mapping H(x) and the obtained mapping :

$$F(x) = H(x) - x \tag{2.16}$$

Because of this residual mapping, when a network should be optimized to an identity function, weights must be pushed down to zero, which is a more natural state for weights.

The original mapping H(x) is therefore recast as $F(x)+x$. In contrast to the AlexNet and VGG networks, ResNet blocks contain two branches as can be seen in figure 2.9.

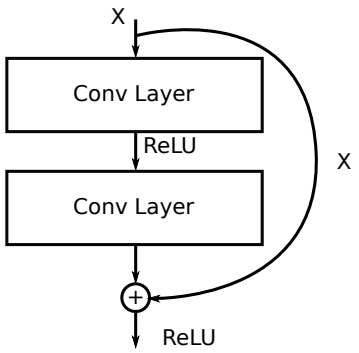


Figure 2.9: ResNet block [1]

The original paper suggests different blocks with a fixed set of parameters (width and kernel-size). One of these full networks, is shown in Figure 2.8b.

2.4 Quantization

According to Chen et al. in [18], data movement is often more energy-consuming than the actual computation, and it is therefore important to reduce the size of the models. To achieve this a simple method is quantization: reducing the bit-width. [19] and [20] discuss the fact that current CNN models contain too many parameters in comparison to the number of training data and that quantization can be used to reduce over-fitting. Another benefit is the fact that weights of smaller bit-size enable faster computation. For example, when quantizing to 8 bits, operations can be 4 times faster due to SIMD (single instruction multiple data) optimizations.

There are two possible forms of quantization, which both have their merits and their flaws.

The first approach uses fixed-point integers, where a number of bits is dedicated to the range between 0 and 1, an example:

0	0	1	0	0	0	0	1
2				1/16 = 0.0625			

This is easy to implement but when designing the fixed-point range, it is important to consider the number of bits before and after the comma. A common approach is

to do this after training, by examination. It is however not necessary to use the same range for every weight. While it would not be memory efficient to use a different range for every weight, one could choose a different range for every layer, as was done in [21] and [22]. This approach works well when not retraining models and was therefore commonly used when defining hardware accelerators.

The second approach to quantization is by redefining weights to be between $-\alpha$ and α. This would not work when reusing a trained network that used floating-point values, but when retraining the network (either by reloading floating weights or from scratch) it is possible to obtain good results with a very small number of bits. It is even possible to work with 2 bits or 1 bit.

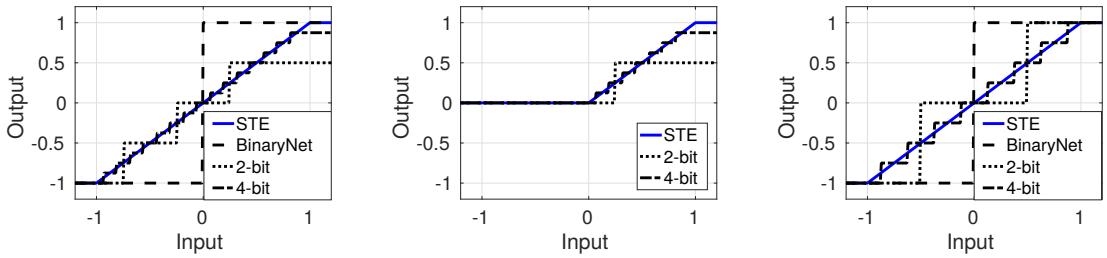


Figure 2.10: (a) Weight quantization. (b) Quantized ReLU activation function. (c) Quantized hard tanh activation function. [23]

When using 1 bit, every value is either -1 or $+1$, this is then called a binary network. It has very desirable properties regarding computation speed, because a convolution can be reduced to a XNOR operation followed by a popcount (counting the bits set to 1), this operation was first done by [24].

When using 2 bits, a number of possibilities arise. The most useful one is the ternary network. In a ternary network only 3 values have sense, -1, 0, and 1. Having a zero value has several benefits, the first one being that most networks make good use of zero values. The second one is that the XNOR operation, used on binary networks, can be used here too, by using the first bit to enable/disable the computation.

One important aspect of quantization is the method of rounding. In the literature the two different methods that are widely used are deterministic rounding and stochastic rounding. Both methods have benefits and drawbacks.

Deterministic rounding finds the closest number that can be represented in quantized form, and will always round a certain value in the same way. Thus deterministic rounding uses far less resources in hardware implementations. In concrete terms, the rounding is defined by:

$$\begin{cases} \lfloor x \rfloor & \text{if } \lfloor x \rfloor \leq x \leq \lfloor x \rfloor + \frac{\epsilon}{2} \\ \lfloor x \rfloor + \epsilon & \text{if } \lfloor x \rfloor + \frac{\epsilon}{2} < x \leq \lfloor x \rfloor + \epsilon \end{cases} \tag{2.17}$$

Where $\lfloor x \rfloor$ is the largest multiple of ϵ.

Stochastic rounding takes into account the intrinsic probability of rounding to the nearest integer. For example, when rounding 1.4 to integer, stochastic rounding will have 60% probability to round to 1, and 40% probability to round to 2. The equation

to round a number to its fixed-point representation is therefore:

$$\begin{cases} \lfloor x \rfloor & \text{with probability } 1 - \frac{\lfloor x \rfloor - x}{\epsilon} \\ \lfloor x \rfloor + \epsilon & \text{with probability } \frac{\lfloor x \rfloor - x}{\epsilon} \end{cases} \tag{2.18}$$

However this is way more resource intensive since the generation of truly random numbers is quite expensive.

2.4.1 Quantization approaches

The quantization of a neural network can be achieved through two different approaches.

The first approach to quantization, and the most widely used, is the quantization of the weight parameters. This can happen *after* training, where the weights are rounded to values for their compression, but the accuracy loss after rounding is often substantial. Quantization can also happen *during* training. When this technique is used, it is often possible to obtain the same accuracy as floating-point networks.

The second approach is the quantization through the activation layer. This approach requires a redefinition of the activation layer, in a way that the output of the activation function will be quantized, while also going through the original activation function.

Chapter 3

Background on FPGA and OpenCL

One of the most important tasks in accelerating deep learning is balancing speed and energy efficiency. To achieve speed, the consensus has since long been to use GPUs to accelerate the highly parallel computation of what is essentially matrix multiplication. GPUs have one flaw: they use a lot of energy. For our purpose, building an accelerator useful for embedded applications, GPUs were not an option. Enter the FPGA, an energy-friendly accelerator that allows for designing fast neural networks.

To supposedly make development of the final design faster, and to experiment on the validity of using a high-level language for FGPA programming, our design is programmed using OpenCL [4], an

> "Open standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms." [4]

This chapter starts with a primer on OpenCL, explaining the design behind it, and the communication channels possible between host and kernel[1]. The next section discusses the design of an OpenCL kernel, and the way multiple kernels can communicate between each other.

After explaining the general OpenCL API, we focus on using OpenCL on FPGA. We discuss the differences between a generic OpenCL implementation and one optimized for FPGA.

To conclude we discuss possible configurations to accelerate convolutions on FPGA.

3.1 OpenCL primer

OpenCL is defined by a standard written by CPU, GPU, and FPGA vendors under the umbrella of the Khronos Group [4]. It defines a parallel programming API (application programming interface), that can be used to run *kernels* written in a subset of the C language, from a host platform (a CPU) to another device (the accelerator). The kernels can be defined in two ways, NDRange or single-work-item.

NDRange kernels have access to functions that will define which part of memory they will work on. One kernel will be run multiple times with the same arguments, but different IDs (in a predefined *range*) to allow for parallel programming. Those kernels

[1] In contrast to the previous chapter, a kernel means a function that will be run on the device, and not a set of weights

will run simultaneously, allowing to use shared local memory over multiple concurrent threads.

On the other hand, single-work-items, which only make sense in the context of FPGA devices, are run only once, with some arguments, and will have to process all computations themselves. This may seem to have no advantage over NDRange kernels, and when programming for GPU or CPU indeed it doesn't, but it allows the FPGA compiler to create a pipeline, which will increase the bandwidth of the computation at the expense of latency (of the first computed value). Single-work-items can be defined to be copied on the FPGA into several compute units, allowing better resource occupation without code duplication.

3.2 Host

The host platform must go through a few API functions to be able to run kernels on a device (be it itself or another accelerator). The host can essentially be programmed in any language that can interact with C code, but in this thesis we chose homogeneity between the host code and the accelerator for simplicity of design.

The first step it must go through is finding the platform ID of the accelerator, either using the default or finding the preferred through string matching. Once the platform is known, one can find the device(s) on that platform, and create a context for every device found. It is possible, and even easy, to distribute computations to multiple accelerators, especially on the same platform (for example an array of FPGAs).

The design of the OpenCL language allows the host program to run commands (running kernels, but also memory allocations) by enqueueing them on a command queue. Commands are then run sequentially, waiting for the previous one to finish before running the next one. To run multiple commands concurrently, multiple command queues must be created.

The last step before running kernels is compiling or loading the OpenCL kernels to match the device code. whether kernels need to be compiled or loaded depends on the platform. For example CPU and GPU code can be compiled on the fly and run directly on the device, whereas FPGA devices need to have the code compiled to an RTL representation and cannot be compiled on the fly.

The following commands can be enqueued:

- `EnqueueNDRangeKernel()`: launches an NDRange kernel
- `EnqueueTask()`: launches a single work item kernel
- `EnqueueMapBuffer()` : creates a pointer to a location in host memory that will be accessible by the kernel.

While this is only a subset of the commands available to an OpenCL programmer, they show the direction of the OpenCL-design clearly.

3.3 Kernel

The host code must go through a lot of boilerplate code before being able to run kernels, but the kernels themselves can be written in almost pure C. While OpenCL defines pragmas and attributes that can help the compiler do a better job at optimizing the

code, the only additional functions it provides are to get the ID of the kernel. A kernel can have several different IDs depending on how it was called:

- `get_global_id()`: the current id, across all groups (different for each thread).
- `get_group_id()`: the current group id.
- `get_local_id()`: the current id, local to the group.
- `get_compute_id()`: the current compute unit (FPGA only)

In Figure 3.1 we can get a better understanding of the relationship between these different variables. To get the global id from all the other variables, the following equation is applied: $(G_x, G_y) = (w_x L_x + l_x, w_y L_y + l_y)$. A work-item is identified relatively to the other items in the same work-group (l_x, l_y). And each work-group is identified relatively to the other groups (w_x, w_y). Every work-group is defined by a local size (L_x and L_y) set programmatically when enqueuing a kernel. The number of work-groups is itself determined by the global size which is also set during the kernel's enqueing.
The compute id is not incorporated on here since it is something limited to FPGAs where hardware can be replicated. It is typically used to assign a replicated kernel element to a position in a systolic array , i.e. if a kernel has a compute id of 5 it has to send the data it contains to the kernel with compute id 6 at the next clock cycle.

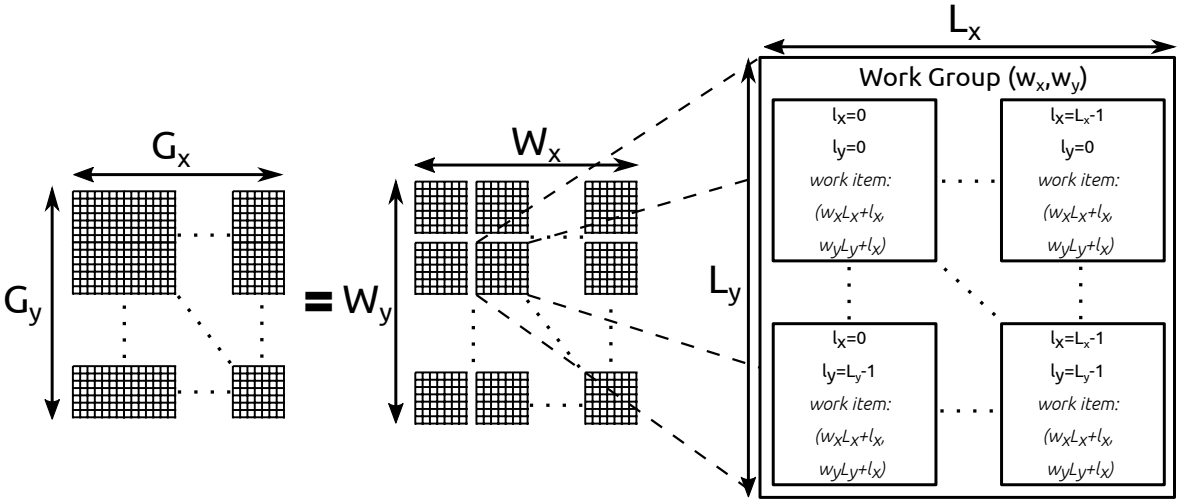


Figure 3.1: Representation of workgroups and id's in OpenCL kernels [25]

3.4 OpenCL and FPGAs

Intel FPGA and Xilinx both support compiling from OpenCL as a high-level design language. However, as it has already been hinted at, some necessary changes remain in order to support the differences between FPGA and parallel-programming-optimized accelerators. Here we will detail the design choices specific to Intel FPGA, but Xilinx has, in broad strokes, the same principles.

The biggest and most influential change is simply a recommendation in the documentation to only use single-work-items if at all possible. It has already been stated

that this allows for better pipe-lining, but it also decreases the amount of resources needed for ID resolution.

The second change is the addition of channels (pipes in official OpenCL), which allow direct communication between kernels using (un)buffered queues. This comes from the realization that memory transfer is often the most time-consuming operation when using the accelerator. It is therefore better to keep memory inside the FPGA for computations instead of having every communication pass through the host.

With the addition of kernel communication, another possible improvement involves removing the hardware needed for communication with the host, by allowing kernels to be "auto-run". They will thus only be able to communicate with other kernels and not the host (not even for arguments).

3.4.1 Compiling an OpenCL Design

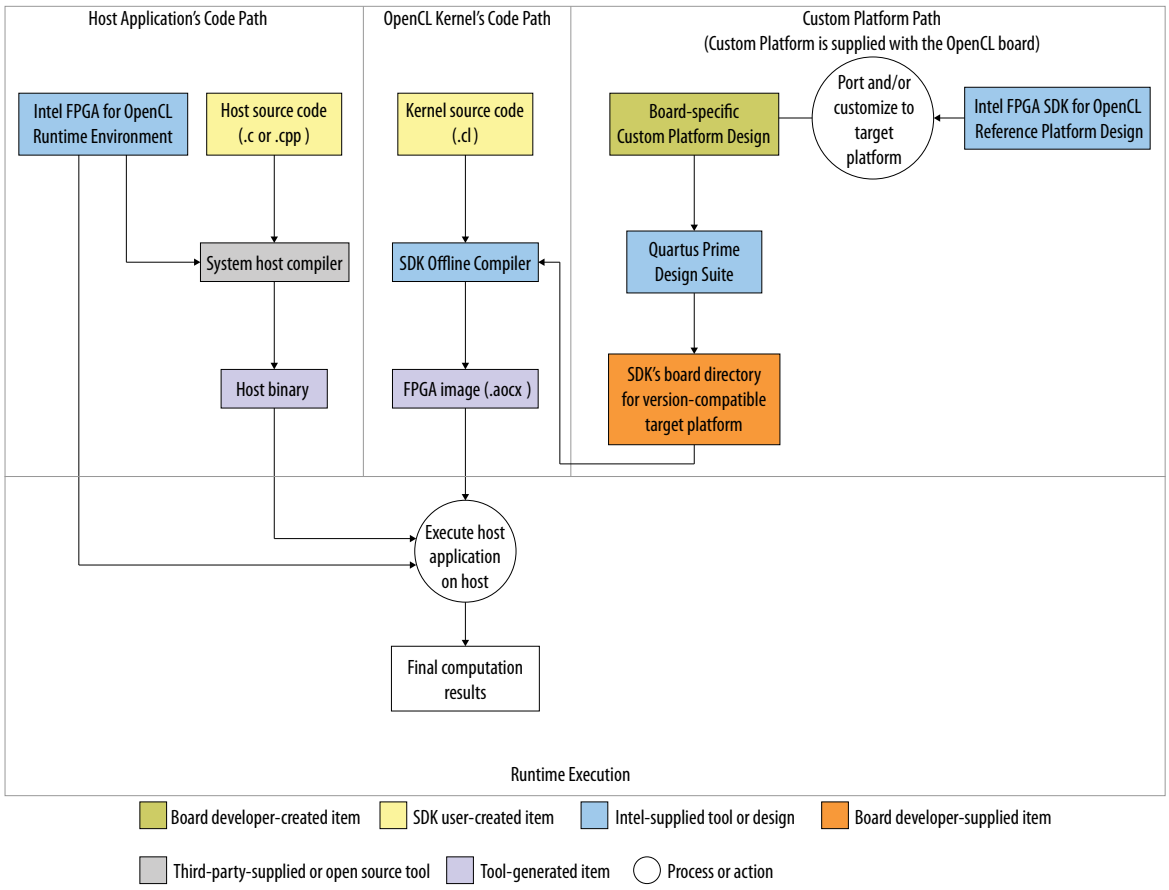


Figure 3.2: OpenCL FPGA flow [26]

When compiling your design for FPGA, it is important to follow steps to make sure the final design is optimal. Those steps are outlined in the Opencl FPGA Programming Guide [26], and can be found in Appendix C. The files needed for every part of the compilation are pictured in Figure 3.2.

Before compiling the full flow to FPGA, the first step is to make sure the design is working and sound. To achieve this the Intel FPGA Group provides a simulator for CPU. It simulates both the OpenCL API, and vendor-specific FPGA additions. The

compilation to simulation is almost immediate. The simulator is an invaluable tool for fast development on FPGA.

After the design has been adequately tested, the next step is to analyze the compilation report generated by the compiler. This report gives information about:

- Total area utilization: how many ALUTs, FFs, 10k RAM blocks and DSPs are used.
- Line-by-line analysis of area usage: which part of the source code is using up the most resources.
- Loop pipelining/unrolling: which loop is pipelined/unrolled. What blocks the pipelining from taking place (memory dependency, out of order inner loop...).
- Loop initiation interval: How many clock cycles between 2 pipeline intervals.
- Memory disposition: Visual representation of the memory access-points and memory-banks.
- System viewer: Block diagram representation of the code's memory accesses.

However, all this information is approximate. Having exact information requires the complete compilation and fitting by the compiler.

The step to compile the kernel source code to an FPGA image (`.aocx`) with the Offline Compiler takes the longest time. To compile a design which will be able to run on the FPGA takes between 1 and 2 hours. It also requires substantial available memory. After compiling, the final file must be tested on FPGA to ensure it works as intended.

3.5 Generic Configurations

The operation that is most often accelerated when inferring CNNs, is the convolutional layer. This is for two reasons: convolutions are the most expensive computations and don't require many weights (relative to a fully-connected layer), being therefore less memory-constrained.

Many designs therefore only focus on accelerating the convolution. While our thesis tries to go beyond this aspect, it seems important to detail the different designs that can be used to accelerate generic convolutions.

3.5.1 Systolic arrays

The first, most used and most researched design is the systolic array. The basic principle of a systolic array is shown in figure 3.3. This basic configuration can be extended to a 2D case.

In a systolic array, every processing element (PE) only does a small computation, often a multiply-accumulate operation (MAC) and only communicates with its neighbors, as in figure 3.4.

One of the advantages of a systolic array is that it is possible to have memory reuse between rows or columns of the array, but also having memory reuse in time, with a FIFO-queue.

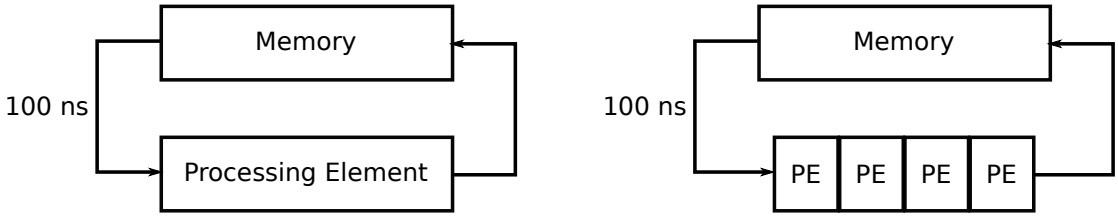


Figure 3.3: Basic principle of a systolic system [27]. (a) FPGA configuration without systolic array. Memory latency is bottleneck of application. (b) FPGA configuration with 1-d systolic array. Design can be made to be computationally bound.

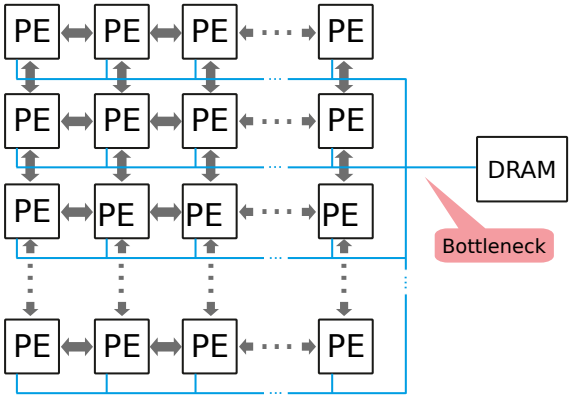


Figure 3.4: 2D systolic array

3.5.2 Tiling

For the convolution operation, the obvious solution is to loop over every variable, and multiply-accumulate all the results. This can cause a problem of not being able to keep all memory inside the FPGA, and having to reload parts of the feature maps or weights multiple times.

One way to solve this is by tiling, which means splitting the problem into several independent sub-problems which can be computed separately.

Figure 3.5 shows how to tile a convolutional layer. The pseudo-code for a tile is as follows :

```
for tr in 0..Tr..Nr:
  for tc in 0..Tc..Nc:
    for ti in 0..Tin..Nin:
      for to in 0..Tout..Nout:
        # Load weights and feature maps
        # Convolution for one tile
```

When processing a tile we can choose to unroll a loop over another. In an FPGA, it is almost always best to unroll as much as possible, as this will enable fully parallel computations. Figure 3.6 shows the four possible unroll configurations. The first one unrolls the loop over the kernel, the second unrolls the input feature maps, the third unrolls the spatial loops and the last unrolls the output feature maps. Each unroll has its benefits and drawbacks, which will lead to different bottlenecks.

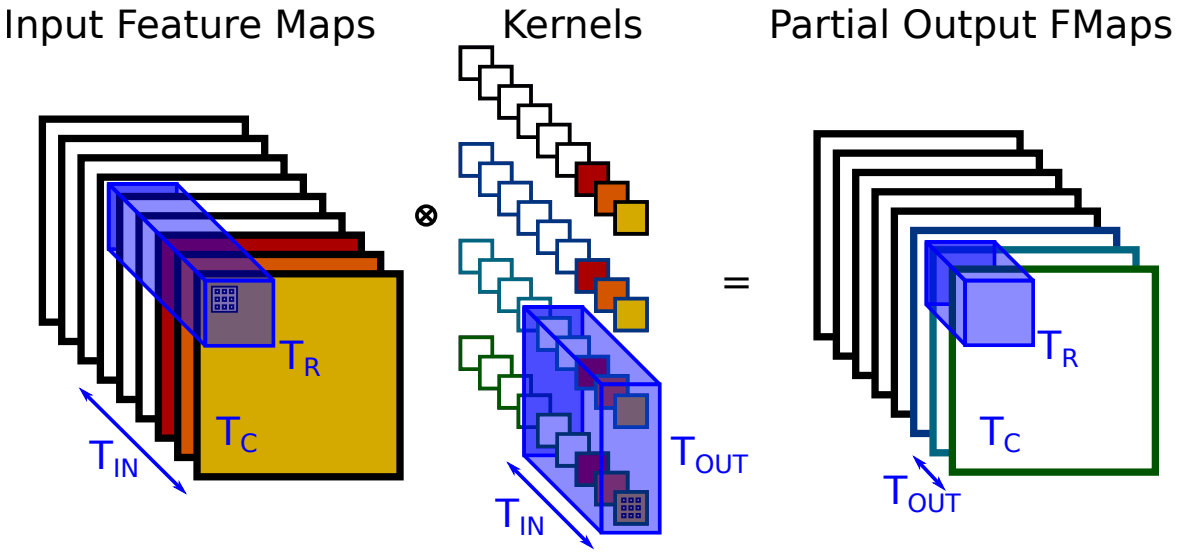


Figure 3.5: Tiling.

We have 3 different types of meta-parameters :

- Tiling: Tr, Tc, Tin, $Tout$, Tk, Tl
- Unrolling: Pr, Pc, Pin, $Pout$, Pk, Pl
- Loop order: *inter*-tile and *intra*-tile

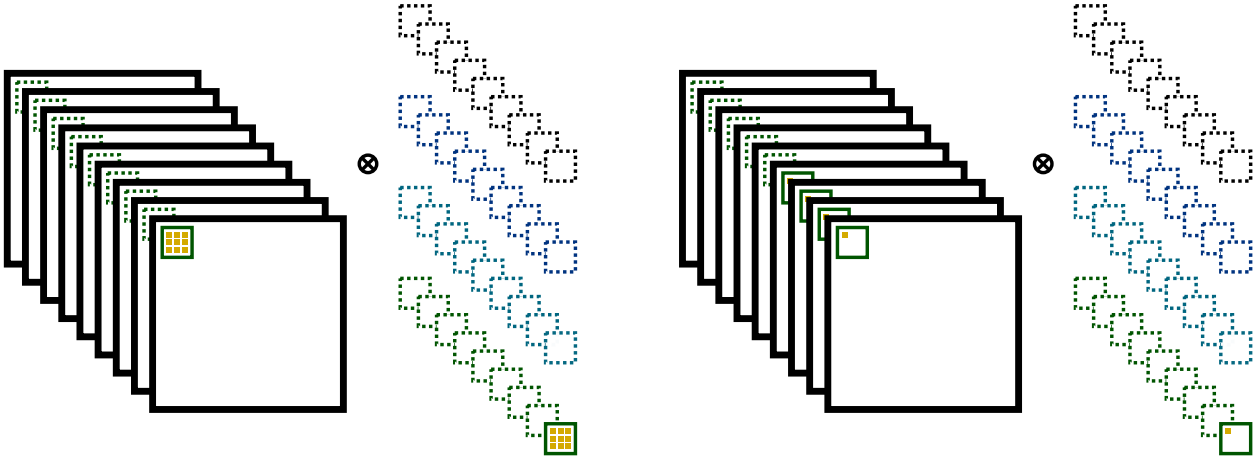
In addition to this, it is important to take into account the total (maximum) size of every parameter:

- Nr, Nc, Nin, $Nout$, Nk, Nl

Those values are often fixed by the model used. For example, a model could have the following values:

Nr	Nc	Nin	$Nout$	Nk	Nl
32	32	64	64	3	3

Nr and Nc are often the size of the image, when the size of fmaps decreases, Nin and $Nout$ the maximal number of input and output feature maps, and Nk and Nl are the maximal size of the kernel. An AlexNet network has $Nk = Nl$ equal to seven, a VGG-like network will have a maximal kernel size of 3.



(a) Unrolling Loop 1

(b) Unrolling Loop 2

(c) Unrolling Loop 3

(d) Unrolling Loop 4

Figure 3.6: Unrolling configurations. It is not always possible or necessary to fully unroll a loop.

Part II

Related Works

Contents

The chapters in this part explain related works, and how they are situated compared to our solution. As such it gives a good overview of the direction we will follow in later chapters. They contain the groundwork on which our thesis expands. On the other hand, this part also goes into the details of some domains which are only partially related to our thesis. These are of value for the subject of FPGA acceleration and compressed training.

Chapter 4

Models and Training for Compression and Hardware Optimization

One important issue when accelerating neural networks on embedded devices is model size and hardware utilization. In this chapter we will see three different methods to reduce the size of models, while trying to diminish accuracy loss.

The first section will go over quantization, where weights and activations which were previously continuous are quantized to a fixed number of bits. The second section will discuss network pruning and deep compression. The last section will talk about architectures which achieve small-sized models, by reducing the number of computations for each layer.

4.1 Quantization

One of the most studied forms of model compression is quantization. The general idea consists of compressing floating point parameters to fixed-point low-precision parameters. In chapter 2, the background for quantization was explained. In this section, we will highlight a few papers that have interesting methods of quantization. A few of those papers have been the basis for our own quantization methodology.

While simple quantization to fixed point was already widespread [28], it was only recently that work has been published on highly quantized neural networks. Using 16-bit and 8-bit quantization showed only a small loss in accuracy without retraining [29], but when fine-tuning a network it is possible to use only 1 or 2 bits. This result was shown in [30] for Binary-weight networks and in [31] for Ternary-weight networks. One important aspect of quantization is the transformation from the floating-point representation to an alternate representation. Binary and ternary weights differ only by the presence of a zero value for ternary weights.

Quantized neural networks [32] apply the same principles applied in binary and ternary networks, but with a formula applicable to any number of bits.

In this work, we study the impact of quantization on accuracy, and subsequently asses the performance of different quantization strategies. We start from the work of [23] to analyze the impact of combining quantized-weight networks with quantized activations. The analysis is done on extreme weight quantization, with weights not greater than 4 bits, while giving activations a greater range for the number of bits. This

approach stems from the work of [32] and [30] showing that weights did not need much precision compared to the feature maps.

In Table 4.1, we compare multiple training methods that use quantization. This table contains networks suitable for fast inference and/or fast training. The networks are not always directly comparable, as they are often only derivatives of the cited network, with some changes or improvements. The error percentage is therefore only added for approximate comparison. When multiple networks or datasets were used, only one result is shown (when possible CIFAR-10 with a VGG network).

Paper	**Training***	**Rounding**	**Weights**	**Acts**	**Network**	**Dataset**	**ΔAcc%**
BWN [24]	–	det.	1 bit	fl-32	AlexNet	ImageNet	0.8
TWN [31]		det.	2 bit	fl-32	ResNet-18	ImageNet	2.66
BinaryNet [30]	–	det. (stoch.†)	1 bit	1 bit	VGG	CIFAR-10	0.99
GXNOR-Net [33]	–	det.	2 bit	2 bit	VGG	CIFAR-10	0.38
QNN [32]	SBN, SBAdaMax	det. (stoch.†)	1 bit	2 bit	AlexNet	ImageNet	6.53
BinaryConnect	BN, Adam	det. (stoch.†)	1 bit	fl-32	VGG	CIFAR-10	-0.74
XNOR-Net [24]	BACP layer order	det.	1 bit	1 bit	AlexNet	ImageNet	11.0
DoReFa-Net [34]	Quant. of gradients	det.	1 bit‡	1 bit	AlexNet	ImageNet	12.3
			1 bit‡	2 bit			6.1
			1 bit‡	4 bit			2.9
Min.-Energy QNN [23]	–	det.	4 bit	4 bit	VGG	CIFAR-10	N/A+

* *Default training happens with a straight-through estimator, and update of the weights in fl-32.*
† *Only stochastic rounding during training, and/or in some tests.*
‡ *First and last layer have full precision weights.*
+ *[23] compares networks at equal accuracy, and subsequently show that 4-bit networks perform the best with minimal energy use.*

Table 4.1: Summary of the different methods of quantization.

4.2 Pruning

Another approach to compressing models is by reducing the number of weights. Having less weights reduces the number of computations, and reduces memory requirements.

Pruning networks is used by Han et al. [19] and Yang et al. [35]. Pruning is the act of setting a number of weights (randomly or with a threshold) to zero, such that less multiplications will have to be performed.

The two papers have different approaches to pruning. Deep Compression [19] uses pruning in a pipeline of compression techniques, including quantization and Huffman

coding. Their approach reduces their original models (AlexNet and VGG-16) 35 and 49 times respectively.

[35] focuses on pruning, and created a method called Energy-aware pruning, where they calculate the impact of every layer on energy usage (theoretically) and prune a layer more or less according to its impact. Their approach tries to achieve a good trade-off between model compression and accuracy. Compared to Deep Compression [19], they achieve better pruning with only a small loss in accuracy, as can be seen in Figure 4.1. This work does not focus on pruning in the same sense as [19] and [35], but when using extreme quantization, it is possible to achieve intrinsic pruning, with a particular thresholding approach. We experiment on the impact of pruning, both from a training perspective, and from a hardware perspective.

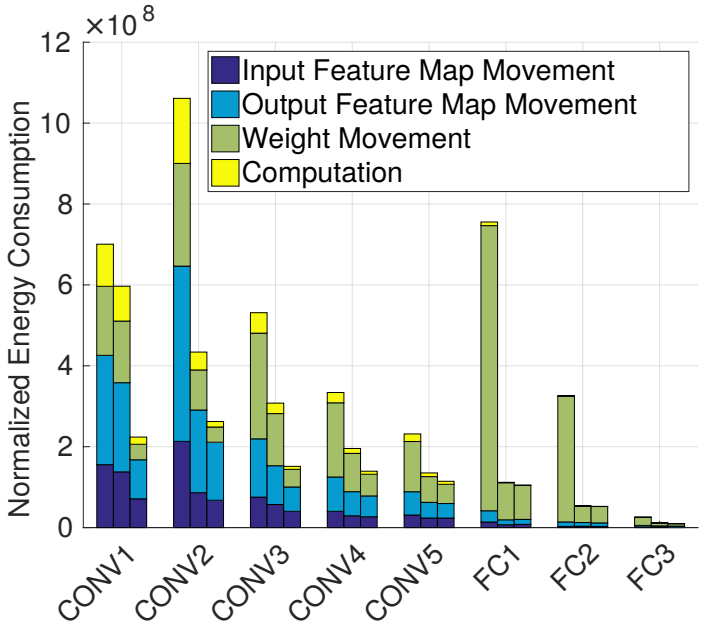


Figure 4.1: Comparing theoretical energy consumption of [19] and [35] in terms of the computation and the data movement of input feature maps, output feature maps and filter weights. From left to right: original AlexNet, AlexNet pruned by [19], AlexNet pruned by [35]. Source: [35]

4.3 Architectural Compression

The last approach to reducing model sizes is by changing the architecture of the models themselves to reduce the number of weights.

While many have offered ways of reducing weights, of which residual networks and inceptionNet are both examples, we focus our attention on two recent addition focusing on the embedded space: SqueezeNet [36] and MobileNet [37].

SqueezeNet starts from an architecture similar to AlexNet's and replaces all but the first and last convolutions by a Fire module. The Fire module is a block that contains a layer with only 1x1 convolutions which squeezes the network, and then a layer which subsequently expands the number of feature maps with a combination of 1x1 convolutions and 3x3 convolutions. This makes it possible to reduce the size of

parameters of AlexNet from 240MB to 4.8MB. When applying Deep Compression [19], they even get a model at 0.47MB, with no loss of accuracy from the baseline AlexNet method.

MobileNet separates the convolution operation in two distinct operations, with different filters: depthwise convolutional filters and pointwise convolutional filters. Pointwise filters are similar to 1x1 filters used in SqueezeNet and residual networks.

The subsequent text does not focus much on the same approaches of architectural compression, although the choice of using residual networks stemmed from the fact that ResNets contained less weights than a VGG network with the same accuracy. This work does however focus on architectural changes that have an impact on the performance of hardware implementations.

Chapter 5

FPGA frameworks for CNN

Deep learning is a relatively young field, even though it uses much of the same ideas developed in the previous century. The acceleration of deep learning with FPGA is younger still. While this is the case, a lot of very interesting and very thorough research has been done that we could use as basis for this work. This chapter showcases acceleration approaches particularly for the convolution operation but also more generally on frameworks which achieve high performance.

Well-summarized in [29], FPGA acceleration of convolutional neural networks can be categorized in three distinct groups:

- Algorithmic optimizations: convolutions are costly operations, but it is often possible to optimize them by doing a different computation. We research algorithmic optimizations by analyzing the methods of tiling convolutional operations. Other techniques include:
  - General Matrix Multiplication (GEMM) : convolution can be seen as a general matrix multiplication operation. It is therefore possible to parallelize it by changing the memory storage and access patterns.
  - Fourier transforms, using FFT or Winograd
- Datapath optimizations: when dealing with FPGAs, memory is often the bottleneck. It is therefore important to have cache-aware operations, and smart memory-management. Our work has an extensive section on datapath optimization, because of the importance of good memory management. The different techniques of datapath optimizations are:
  - Static data-flow
  - Datapath process networks
  - Design space exploration, the Roofline method
- CNN model optimizations: current models are designed to use fast GPUs with optimized floating-point operations. Changing the structure of the model to accommodate FPGA acceleration can also have important benefits. This category of optimizations was discussed in Sections 4.1 to 4.3. In this thesis, we focus our attention in the first place on this optimization.

In the following sections, we classify works that relate to our own by their domain of interest. The works showcased in the following sections either optimize one specific

operation (often the convolution) under specific circumstances, or try to provide a framework which can be used with some specific deep learning model types. Most frameworks use a mix of all approaches to accelerate neural networks.

5.1 Convolutional Layer Acceleration and Design Space Exploration

A convolution can be accelerated in many different ways. The loop-ordering can be changed and tiling sizes are completely up to the programmer to set. We found 3 works that tried to find an optimal way of setting those parameters:

- Zhang et al. [38] introduce a way to optimize a High Level Synthesis (HLS) implementation by the roof-line method (shown in Figure 5.1) under some fixed loop-order constraints.
- Ma et al. [39] use a flowchart to determine loop-ordering, tiling, and parallelism through partial sum and DRAM access analysis. Contrary to [38], their implementation is in Verilog.
- [40], [21], and [41] try to automatically explore the design space to generate an optimal solution.

```
for (i=0; i<K; i++)
  for (j=0; j<K; j++)
    for (trr=row ; trr < min(row+Tr,NR); trr++)
      for (tcc=col; tcc < min(col+Tc, NC ) ; tcc++)
        for (too=to; too < min(to+Tout,NOUT); too++)
        #pragma HLS UNROLL
          for (tii=ti; tii < min(ti+Tin,NIN); tii++)
          #pragma HLS UNROLL
            L : output_fm[too][trr][tcc] +=
                  weights[too][tii][i][j]*
                  input_fm[tii][S*trr+i][S*tcc+j];
```

Listing 5.1: Accelerator structure from [38]

As can be seen in Listing 5.1, [38] chose a loop order which enables an unroll factor of $Tout \times Tin$. Since their approach is in HLS, we don't talk about PEs strictly speaking, but we can see the unrolls as having the same effect (the Intel OpenCL guide even includes this as advice [42]). The effect of that unroll is a collision on the sum of Tin leading to the synthesizing of an adder tree to sum the Tin multiplications. The unroll on $Tout$ has the effect of leading to parallelization. This unroll enables sharing of the input pixels across $Tout$ unrolls. This way we don't have to load weights frequently: there are $Tr \times Tc$ iterations with the same weights loaded in for the multiplications. This method has the disadvantage that it needs a lot of local memory in order to store

the number of partial sums ($Tout \times Tr \times Tc$).

With this fixed loop ordering and unrolling, they used the roofline method in order to find the optimal parameters for their design. This design exploration method tries to find the best trade-off between memory bandwidth and computational speed. In Figure 5.1 we have 2 uncrossable limits:

- The Computational roof: determined by the number of DSP's on the FPGA
- The Bandwidth roof: determined by the memory bandwidth of the FPGA

The y axis is the attainable performance of the design or : $\frac{\#\text{ total operations}}{\#\text{ execution cycles}}$. The x-axis represents Computation to Communication ratio or: $\frac{\#\text{ total operations}}{\#\text{ total external memory accesses}}$. When looking at C and D on the figure, although they both should be as fast, it is better to select C's design since the bandwidth's margin is higher.

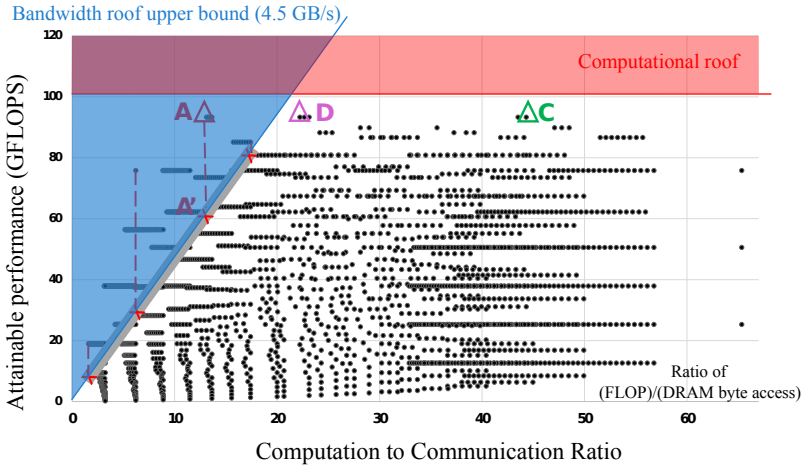


Figure 5.1: Roofline method from [38] for design selection

Ma et al. used a different approach. They wanted to minimize the latency by maximizing the DSP utilization. To this end, they decided to limit the **number of partial sums concurrently stored in local memory** as well as the total **number of DRAM memory transfers**. To make the design space exploration clearer they made 2 flowcharts (Figure 5.2). The number of partial sums is optimal if the unrolling happens on the number of parallel computations of output pixels. This implies that the partial sums should be completely computed first in order to be more rapidly evacuated.

5.2 FPGA Acceleration Frameworks

The last few years plenty of acceleration frameworks were proposed. This section will be more of a performance analysis between those different methods than a detailed analysis of their acceleration techniques since those were already detailed in the previous section. It should however be noted that only one implementation (PipeCNN [43]) was open-source and thus had reviewable code and performance metrics. For all the other papers we required them to have non-simulated, absolute metrics (not only internal comparisons with own work on other platforms).

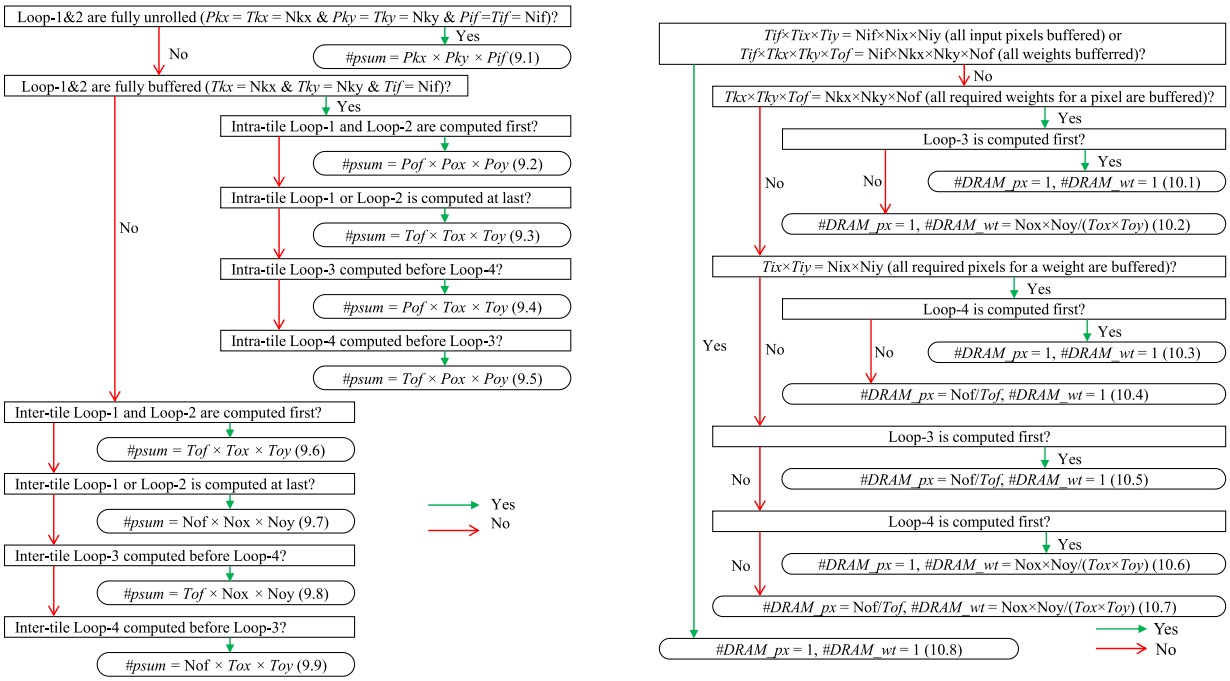


Figure 5.2: Design space exploration flow-charts (from [39]) for the # partial sums concurrently stored in local memory (a) and the # duplicated global memory accesses (b)

We are going to explore different implementations, ranging from ASIC designs to FPGA implementations in HLS. From multi-network frameworks to closed down solutions optimized for certain types of networks. We tried to enable a fair comparison between the different approaches by comparing their performance numbers in Table 5.1.

The number of giga-operations per second (GOPS) is more often than not approximated with being 2 times the number of MAC operations necessary to compute the convolution. The activation and batch-normalization operations are small compared to it, since the relation between both is quadratic in favor of the # MACs. When Winograd transforms or FFT are applied, it is stated as such in the 'transform' row. Since those transforms effectively *reduce* the number of operations, it is practice to use the value of the number of operations they *would* have performed in the original domain as metric during throughput computation.
In case of strong quantization, the 'accuracy drop'-row indicates the accuracy delta compared to the paper's floating point result. Most 16-bit fixed point implementations don't specify their delta, but we can assume it is lower than 2% as observed by ([21] and [39]). This reduction could probably be completely suppressed by finetuning the network with the 16-bit fixed point constraint, but it often isn't the main point of interest of these papers.

As can be seen in table 5.1, the hardware used is quite different. Even though the papers are within 2-3 years of each other, FPGAs have enjoyed tremendous energy savings thanks to the move to smaller processing nodes or simply better architectures. On account of this, we found that comparing energy efficiency didn't make a lot of sense. We prefer to turn our attention to the throughput speed. Thus, the FPGA-metrics that become important for comparison are the number of DSPs and the clock frequency

(which can either be design or hardware bound). But the reader should still be aware that some FPGAs have a faster memory bandwidth which can play an important role for memory-bound designs.

We tried to always select comparable networks, preferably AlexNet or VGG-16 on the ImageNet dataset, with one exception due to the FPGA used being of a lower tier.

The language row represents the language used to generate the logic. The host program's language (if applicable) is irrelevant (and most of the time C/C++).

Most implementations that want to maintain a high accuracy without needing to retrain a floating point network (or simply don't want to bother with it) will choose a 16-bit fixed point (fx-16) implementation. This bit-width reduction can effectively double the system's throughput by actually being able to perform 2 multiplications per DSP.

We can also note that the highly efficient implementations of Eyeriss [18] and Yin et al.'s hybrid NN [22] (200 and 300mW) tend to be slower throughput-wise.

When looking at the high-end implementations we can note that using pure Verilog implementations does not seem to be necessary. For example [44] uses an OpenCL framework for everything but the actual multiplication which is written in Verilog for efficient resource utilization on parallel computations.

Comparing [39] with [44], shows us the importance of frequency. They both run on the same model of the Arria X FPGA, but [44] is ∼2.77x faster. Most of the difference is accounted for by the frequency of the design which is ∼2.58x higher. This is a stark reminder that a bad operating frequency can severely handicap a good design.

Using Fourier (and by extension Winograd) transforms [45, 46, 47] seems to really increase the throughput. While those transforms will need more resources, they appear to easily beat high bit-width implementations in speed. Although the implementation of [43] got an upgrade quadrupling its throughput by moving to 8-bit quantization, it still doesn't come close to the throughput of [45]. Looking at higher tier FPGA models, only [44] remains competitive with the Fourier Transforms thanks to its high frequency. Sadly such algorithms can't be generalized to low bit-width models without removing their main advantage: compactness.

By looking at [41] we see the potential for lower bit-width implementations. While those will need more operations to be computed. Not needing to employ DSPs enables a greater parallelization since the whole design becomes strictly logic-bound. This is also shown by the binary implementations which achieve very high throughput on inferior hardware.

[21] goes into another direction, using Singular Value Decomposition to reduce the number of different possible weights. This approach seems less competitive than a better thought-out design in 16-bit fixed point (fx-16).

Binary network implementations [48, 49] show their strong side by attaining a very high throughput. The final throughput of [48] doesn't crack 2 TOPS, but it should be noted that it's held back by a relatively low frequency. This is in part due to the fabrication node of the Stratix-V being higher (and older) than that of the Arria-X.

	[43]	[45]	[50]	[41]	[48]	[49]	[21]
Hardware	Stratix V	Stratix V	Virtex 690t	Arria X	Stratix V	Zynq 7Z020	Zynq 7Z045
Language	OpenCL*	Verilog	Verilog	Verilog	Vivado HLS	Verilog	Verilog
Transform	None	(I)FFT	None	None	None	None	None
Bit-width	fl-32	fx-16	fx-16	2(t)/2	1^o	1^o	fx-16
Adaptable	yes	yes	yes	yes	yes	yes	yes
Frequency (MHz)	181	< 200	<150	∼275	<150	143	<150
# DSPs	162/256	256/256	2833/3600	UNK	384/1963	3/220	UNK/900
Network	Alexnet	Alexnet	VGG-16	Alexnet	Alex(Xnor)net	Convnet(Cifar10)	VGG-16 SVD
Acc. drop	None	UNK	UNK	∼6.5%	12.6%	∼3%	1.6%
Batch-size	1	UNK (> 1)	32	1	1	1	1
Latency (ms)	43 (c)	UNK	65.13	UNK	1.16	5.94	224.6
Throughput (GOPS)	33.9	780.6	354	4900	1963	207	137

	[18]	[22]	[46]	[39]		[44]	[47]	
Hardware	ASIC	ASIC	Arria X	Arria X		Arria X	Stratix V	
Language	-	-	OpenCL	Verilog		OpenCL/Verilog	Verilog	
Transform	None‡	None	Winograd	None		None	Winograd	
Bit-width	fx-16	adaptive(6-9)	fl-16	fx-16		fx-16	fx-16	
Adaptable	needs support	yes	not yet	yes		UNK	yes	
Frequency (MHz)	∼175	UNK	300	150		387	200	
# DSPs	N/A	N/A	UNK	1518/1518		1378/1518	1738/1963	1680/1963
Network	VGG-16 +	Alexnet	Alexnet	VGG-16	Resnet-50	VGG-16	VGG-16†	Resnet-50†
Acc. drop	UNK	1.8%	None	None	None	UNK	UNK	UNK
Batch-size	3	2	8	1	1	1	1	1
Latency (ms)	4309.5	UNK	UNK	47.97	∼13	17	14.5	7.95
Throughput (GOPS)	∼60	302	1382	645	∼643	1790	1928	973

* *Later implementation with a quadrupled throughput used a verilog module for the computation*

^{o} *First and/or last layer have less quantized weights*

† *The network only has 5 output classes instead of 1000, reducing the complexity of the last FC layer*

‡ *No transformation was applied, but this implementation makes use from the sparsity in FMs*

\+ *Values for Convolution only*

Table 5.1: Performance metrics comparison in recent works. fx-16 represents 16-bit fixed point and fl-16, 16-bit floats

Part III

FPGA Acceleration through Training Quantized Neural Networks

Contents

Building upon the background and the state of the art of the previous chapters, this final part discusses our methodology, experiments, and the results we obtained through quantization.

The first chapter details the optimizations and quantization choices we made to train an adequate convolutional neural network, which was to be adapted for FPGA acceleration.

The second chapter shows what we achieved in FPGA acceleration. In particular, it discusses the implementation choices and the results we obtained using OpenCL on the DE1-SoC, and compares it to existing solutions.

The final chapter brings the two parts together and concludes our work.

Chapter 6

Model Optimizations, Training and Quantization

In this chapter we'll discuss our approach to the quantization problem. The first section details our methodology. It starts with an explanation on quantization, followed by some implementation choices, search-space exploration limits and finally our framework to quantify the complexity of a network.

We follow this up in the second section with the tests we ran, and some preliminary observations. This includes hyper-parameter selection as well as some network structure changes favorable to an FPGA implementation.

The last and final section contains the discussion of the most interesting results, their limitations and possible improvements.

6.1 Methodology

6.1.1 Quantization choices

We wanted to study quantization on a broad spectrum, not limiting ourselves to purely low- or high-power implementations. But we had to select a fixed number of bit-width reductions in order to avoid large training times.

We decided to look at what had been done pertaining to reducing the size of weights and activations individually:

- Binarization : 1-bit
- Ternarization: 2-bit
- Quantization : 2-, 4-, 8-bit

Binarization

This approach is quite drastic, it aims at reducing the bit-width as much as possible. In this case, reducing everything to one bit is as low as possible. The rounding was done by following the approach of Courbariaux et al. [30]. This rounding limits the values to $-1/1$ based on their sign and clips the gradients to $[-1, +1]$ during back-propagation. They also proposed stochastic rounding, but this wasn't used since we didn't want to deal with random values during kernel programming. In their BinaryConnect

[20] implementation they recommend an ensemble of networks to obtain statistically significant results, but this would slow down inference by the size of the ensemble.
The binarization approach is identical when applied to either weights or activations. This means that during runtime, assuming both weights and activations are binarized in the same manner, every multiplication can be changed to a xnor-operation (keep in mind that 0b0 = −1 and 0b1 = 1 as encoded in memory):

a	b	$\sim(a \wedge b)$
0	0	1
0	1	0
1	1	1
0	1	0

$\equiv$

a	b	$a \cdot b$
-1	-1	1
-1	1	-1
1	1	1
-1	1	-1

This also allows for some interesting properties since we could compress the bits into a 32-bit array as proposed in [3] and [49], and then apply the xnor operation very easily between the weights and values. After the xnor operation, the values must be aggregated using a popcount (= counting the number of bits) on the result. This is very light on resources. When using the approach of counting all bits in a 32-bit number, an adder tree must still be used to add all results together.

Ternarization

Ternarization expands upon the concept of binarization in an interesting way. While it uses 2-bits, it can feature the same interesting properties as binarization by doubling the number of popcounts, as explained in [33]. It seemed quite interesting to us to compare ternary weights with binary ones and with 2-bit quantization. They are all 3 able to remove any multiplication, although the 2-bit implementation can't be reduced to a xnor+popcount operation. This means that an implementation could potentially not contain any multiplication operation between integers (or floating-point values). From a hardware perspective, this property is valuable.

The intuitive ternarization would be to round all weights in the $[-0.5, +0.5]$ range to zero and the others to $-1/+1$. While this is what we used for the activations, we implemented something more adaptive for the weights.
The ternarization problem can be generalized to:

$$\mathbf{W}_i^t = f_t(\mathbf{W}_i|\Delta) = \begin{cases} +1 & \text{if} \quad \mathbf{W}_i > \Delta \\ 0 & \text{if} \quad |\mathbf{W}_i| \leq \Delta \\ -1 & \text{if} \quad \mathbf{W}_i < -\Delta \end{cases} \tag{6.1}$$

In [31], Li et al. argue that this Δ has an optimal value for every layer, one that can be approximated in the following way:

$$\Delta^* \approx 0.7 \cdot \mathbb{E}(|W|) \tag{6.2}$$

This is the result of minimizing the euclidian distance between the ternary values and their floating point counterparts. The complete demonstration can be found in Appendix A.

Q-bits

We wanted to develop a general methodology which could be applied to every possible network. For this we looked at Moons et al.'s work in [23]. They examined quantized neural networks and tried to find the optimal quantization of weights and activations with regards to energy consumption. They adapted and generalized the binary quantization used by [32] (see *Binarization* in Sect. 6.1.1). [23] quantized them using two's complement fixed-point in the $[-1, +1]$ range on Q-bits, leading them to be rounded with the following formula:

$$q = clip\left(\frac{round\left(2^{Q-1} \cdot w\right)}{2^{Q-1}}, -1, 1 - 2^{-(Q-1)}\right)$$

With $Q > 1$. w and q respectively the value we want to round and the rounded result. During the back-propagation a straight-through estimator (STE) is used which is also clipped in the same interval. This is sufficient for us since we are not interested in backward-pass quantization.

Limiting the test-exploration space

We still needed to narrow down a lot of combinations in order to obtain a manageable amount of tests.

Based on early tests, we did not want to restrict ourselves to the conclusions of [23] since our results seemed to differ. This deviation can be explained by their performance metric which is energy efficiency. Since FPGAs have a more or less constant range of energy usage, it makes less sense to focus solely on this energy efficiency metric to measure the networks' complexity. Our approach tries to look at it more from a throughput and latency perspective. Moreover their conclusion were taken on VGG-like networks while we wanted to concentrate our attention towards ResNets. To this end we allow combinations with different quantizations on weights and activations.

Our pre-tests showed that sometimes moving from 2-bit to 4-bit activations would greatly improve the accuracy. Other pre-tests showed that scaling ResNets up using the way suggested in the original paper by [1] wasn't really that beneficial to the performance of the system. It was even detrimental at times on quantized weights, compared to the higher entailed model complexity.

We therefore opted to only slightly add depth to the ResNets and widen them by powers of 2 in our tests (more on this in Sect. 6.2.1). We opted to test the following quantizations for the weights: binary, ternary, 2-bit Q, 4-bit Q. And used for activation: binary, ternary[1], 2-bit, 4-bit, 8-bit Q.

6.1.2 Implementation

We decided to start off with the code from [23] which was open-source. The code enabled us to easily test out different network configurations. It was written for the Keras framework [51]. This API offers a functional interface to create models. We also

[1] binary and ternary activation only used when the weights are quantized as such

noticed that [23] took heavy inspiration from the third-party implementation of [30][2]. There we found an implementation for the ternary layers and modified it to reflect equation (6.2).

We modified the code in order to make it more general and improved it to make some parts more readable. This allowed us to test ResNet configurations. The code from [23] used a non-quantized fully-connected layer as well as non-quantized biases during training. We quantized the fully-connected layer and decided to remove the biases since they didn't have any effect on the accuracy of quantized networks. This result was also noted by [49].

Optimizer

During training we decided to use the Adam optimizer [14], described in section 2.2.3, combined with a categorical cross-entropy loss function, since Adam has the advantage of faster convergence over SGD with Momentum[3]. Cross-entropy has more or less become the standard for classification with neural networks and we didn't find a good reason not to stick with it.

Data augmentation

Data augmentation aims at extracting more information from one image by adding slightly altered versions of it to the dataset. We decided to keep it quite simple. Our goal wasn't to obtain the perfect accuracy but to be able to compare it to results in the literature. As we'll discuss in section 6.1.2, it can be observed that it also helps with the generalization especially for quantized networks. It is also important that in order to compare implementations, a similar data-augmentation strategy is used since it improves the average test accuracy by a significant margin.

For these reasons we settled on 2 augmentation techniques: randomly shifting the images up and down (by max 3 pixels), while filling the blank pixels with the nearest value; and randomly flipping images horizontally (around the vertical axis).

Before going any further we'll just define the following terms in order to avoid any confusion in the following text:

- (mini-)batch: A fixed number of images that are trained or inferred together
- step: A time(step) unit defining the training of one mini-batch
- epoch: A time unit defining that all input images were trained upon or inferred. In 1 epoch there are (# input images)/(mini-batch size) steps

We used the built-in Keras data-augmentation tool, `ImageDataGenerator`. This tool will randomly generate different images in real-time for each epoch. This guarantees

[2]The implementation can be found on Github at `https://github.com/DingKe/nn_playground/`.

[3]Even though [52] found that Adam and related methods don't perform as well as SGD on the metric of final accuracy, we observed a loss in accuracy when using SGD with Momentum.

the following: the training time for 1 epoch stays the same as the one without data-augmentation **and** every original image is used at every epoch, just not necessarily in its original state.

Dataset

Since we're only looking at ResNets we decided to only test the CIFAR-10 dataset [53]. There comes a point where datasets like MNIST become too easy for some network structures and we didn't find it useful to redo the works of previous researchers on smaller VGG-like networks, like that of [23].

CIFAR-10 is composed of 60000 images: 50000 for training (of which 5000 were used for validation) and 10000 for testing. The dataset only contains 10 classification tasks. Compared to the Imagenet dataset [54] which has 1000 classes and approximately 1 million images (in most cases resized to 224×224) it seems quite insignificant. However, despite its smaller size, it is still a valuable case study (unlike MNIST), with recent network structures scaling significantly on CIFAR-10 when scaling positively on Imagenet, i.e. ResNets perform better than equivalently sized VGG-like networks.

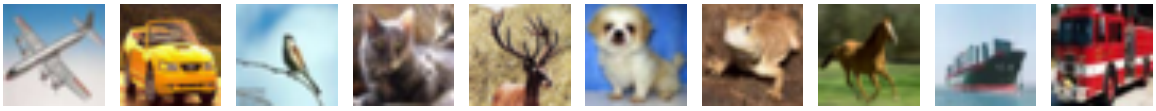


Figure 6.1: A sample of the CIFAR-10 dataset consisting of 32x32 images. The ten classes shown are (from left to right) airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck.

Repeatability of training

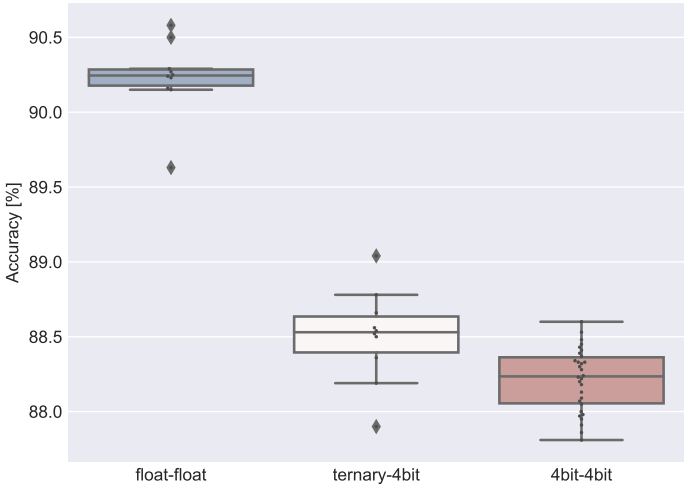


Figure 6.2: Boxplots of the variance across different validation folds (float-float and ternary-4bit) and across randomized data-augmentation (4bit-4bit).

The problem with deep neural networks is the repeatability of the training. Since an easier to train classifier could just be run multiple times and averaged out, while CNNs require prohibitively long training times. A confidence interval could then also

be established, making comparisons statistically significant. When comparing networks we use the same 10% of the training set as validation. The weights used to evaluate a trained model were the ones which scored best on this validation set, to overcome the problem of overfitting on the training set.

To estimate the variance of our training methodology and the influence of a changed validation set we took the following measurements on a ResNet-20:

- Used a 10-fold cross-validation to measure variance in float-float and ternary-4bit networks.

- Kept the same validation fold, repeat the training 30 times to assess the influence of the randomness of data augmentation

The results can be seen in Figure 6.2 and note that most values are within 0.5% of the median. Something to keep in mind when comparing different data-points later on.

6.1.3 Network complexity modeling

To describe the networks in a way that would enable relatively accurate comparisons, a model to evaluate the computational complexity was needed. We really liked the approach of [23] which consisted in comparing energy-efficiency on an iso-accuracy level. However we found the result of the energy formula strange: a ConvNet big enough to obtain >90% on the CIFAR-10 dataset consumes as much power as a ConvNet that obtains >99% on the MNIST dataset. This does not seem reasonable because of the size of networks for MNIST, for example LeNet5 [6], compared to networks used for CIFAR-10. The only other well-documented energy approach we found was the one from Eyeriss [18]. However, their model focuses heavily on *their* ASIC-implementation and is very network-dependent. They also look at the sparsity of the feature-maps at every stage. Thus their approach was too implementation-dependent, while the objective of this thesis was to generate a more general one.

We therefore decided to come up with a formula which would suit all types of networks as well as enable an iso-accuracy comparison. This would allow us to select our final model.

As a starting point, we looked at the number of memory accesses. Recent work in the implementation of the convolution operation [39] makes it so that we can suppose that all weights ($\#w$), input values ($\#v_{in}$) and output values ($\#v_{out}$) only need to be loaded once from or written once to global (=external) memory. If we define $abits$ and $wbits$ as the number of activation and weight bits respectively, this leaves us with the following term:

$$\#\text{global accesses} = \#w \cdot wbits + (\#v_{in} + \#v_{out}) \cdot abits \tag{6.3}$$

Only using this formula leads, we think, to a simplistic representation of real-world implementations (still for comparison's sake, the graph can be found in appendix B). We decided to add a term that weighs the accesses to local memory since those are a great proxy to represent the number of MACs (unless we possess unlimited logic elements and can do a nearly unlimited amount of multiplications concurrently). If we

had a naive implementation we would obtain the following additional equations:

$$\begin{aligned} \#\text{local accesses} &= (abits + penalty) \cdot \#MAC && \text{(load partial sums)} \\ &+ abits \cdot \#MAC && \text{(load input values)} \\ &+ wbits \cdot \#MAC && \text{(load weights)} \end{aligned} \tag{6.4}$$

The penalty added to the memory access for partial sums in Eq. 6.4 is there because we're looking at fixed-point implementations. Under those circumstances, partial sums have to use additional bits since the values before batch-normalization and activation have a bigger range, and therefore need a larger number of bits in memory. This penalty is not applied to floating point additions since those don't require additional bits to be stored. It is also calculated in function of the network's biggest feature map width (here $64 \times K$) and largest kernel size (here 9). In our case it took this form: $\lceil \log_2(64 \cdot K \cdot 9) \rceil$ with K being the relative width of the network. We're aware that some implementations like [39] and [46] use a shared exponent strategy to limit the influence of those additional bits, but we want to maintain a general approach and although the penalty could be lowered, its influence would still be partly present.

This would however imply that no weight or input value is ever shared between multiple PEs. It also eliminates possible synthesized adder-trees or shift-registers, which are often used in hardware implementations, for example in [38, 39]. Knowing that the # MAC is defined as the sum of the # MAC per layer:

$$\#MAC = \sum_{la=0}^{La} \#MAC_{la} = \#weights_{la} * \#row_{out,la} * \#col_{out,la}$$

It's easy to understand why the number of local accesses overwhelms that of global accesses. But, as we just stated, this isn't a realistic value. To adapt it, we decided to take registry accesses into account and add the following variables concerning loop optimizations:

- Tk and Tl: The tiling on a weight kernel
- Tr and Tc: The tiling on an output fmap
- Tin and $Tout$: The tiling on the input and output fmaps
- Pin and $Pout$: The unrolling on the input and the output fmaps
- Pk and Pl: The unrolling on a weight kernel

Most FPGA implementations use a PE-structure, as noted in section 3.5.1, which implies a degree of weight and pixel reuse to limit accesses to local memory. It also means that sums won't always need to transit through local memory at each addition.

The point of this formula is more or less to weigh two terms with respect to each other: the # MAC and the # Global Memory Accesses. Those values represent the computation and global memory transfer time. The number of partial sum loads is reduced by:

- *unrolling* on **weight kernels** and **input feature maps**: enables adder-tree synthesizing.

- *unrolling* on **output feature maps**: part of partial sums are temporarily stored in registers.

The number of feature map reads is reduced by:

- *tiling* on the **output feature maps**: values are recycled in neighboring PEs (horizontally)
- *tiling* on the **weight kernels**: values are recycled in neighboring PEs (vertically)

The number of weight reads is reduced by:

- *tiling* on a **output feature map**: weights can be used on a big patch (with padding) of the input to generate a (partial sum) patch of the output

$$\begin{aligned} \#\text{local accesses} \;=\; & \#macs \cdot \frac{(abits + penalty)}{Pk \cdot Pl \cdot Pout \cdot Pin} && \text{(load partial sums)} \\ +\; & \#macs \cdot \frac{abits}{Tout \cdot Tk \cdot Tl} && \text{(input values)} \\ +\; & \#macs \cdot \frac{wbits}{Tr \cdot Tc} && \text{(load weights)} \end{aligned} \tag{6.5}$$

Most implementations [38, 39] don't unroll the weight kernels leaving Pk and Pl to 1. These choices enable convolution optimizations which are kernel-size independent. On the other hand , the kernel is most often completely tiled in memory, meaning Tk and Tl are equal to the maximal kernel dimension. Since in our case, kernels are mostly 3×3 (only 2 convolutions use kernels of size 1×1) , those values are both set to 3.

The smallest hardware has normally enough local memory to tile Tin and $Tout$ up to the first convolution width, 16 and unroll on those same values as well (Pin, $Pout$). A bigger value would mean that the formula needs to be adapted to all the different layers and we should cap the tiling value at $\min(Tin, width(layer))$. For the unrolls it would also mean that during smaller convolutions, part of the hardware wouldn't be utilized. Extending the same reasoning to Tr and Tc, we set them to 8 to avoid making the formula too network specific.

6.1.4 Model Limitations

While we're confident enough in our approach to perform a model selection based on the results of our formula, its primary limitation is that the complexity of the multiplication is not taken into account since it is supposed to be pipelined. This implies that computations won't be the bottleneck of any implementation. This is at least partially true as floating point multiplications or fixed point ones are offset to DSP-units. But on the other hand, ternary and binary weights avoid multiplications altogether.

In practice, lower bit-widths go hand in hand with a better resource utilization. In turn, this leads to higher unroll factors, generating a higher throughput.

It is also important to note that our model is purely theoretical, and can only be used for model selection, and not for comparing the energy-efficiency of our networks with hardware implementations. We use this model only to be able to *estimate* the

performance of our relatively small networks on low-power hardware, and to motivate our choice of network.

This in no way replaces a thorough analysis of an application-specific formula. Such formula can be used to perform a precise roofline optimization (as in [38]). We eventually won't need such a precise roofline optimization since our design will end up not being bound by memory and throughput but by the available logic resources of the FPGA.

6.2 Experiments

This section first explores the difficulties and strange behaviors we encountered when training our networks. These difficulties are the reason we opt for our subsequently described implementation choices. Afterwards we focus on the experiments we conducted to select the right model to use for our hardware implementation.

6.2.1 Initial training

Initial network structure

We decided to go with a ResNet-v1 architecture from [1] as described in section 2.3.3. This architecture, while relatively recent, is well researched and well understood. The shortcut connection we opted for was a convolution. This would enable us to also test our future hardware implementation with convolutions of different kernel-sizes. The connection between different blocks is done through strided convolutions. Here you can find the structure of the network (with n=depth, K=width):

Block Name	Output Dimension	#Channels	Occurrence
Input	32×32	3	1
Conv 1	32×32	16K	1
Block 1	32×32	16K	2n
Block 2	32×32	32K	2n
Block 3	32×32	64K	2n
AvgPool 1	1×1	64K	1
Dense 1	32×32	10	1

While ResNets can easily be made deeper, this results in diminishing returns and if the hyper-parameter selection isn't perfect, the network could achieve worse results. Scaling in width however can make achieving higher accuracies less difficult. For this reason we decided to test the following depths with the following widths:

Name	Depth	Width	#MAC	#Weights
ResNet-14	2	1	26.6M	208k
ResNet-14×2	2	2	105.7M	831k
ResNet-20	3	1	40.8M	304k
ResNet-20×2	3	2	162.4M	1 218k
ResNet-20×4	3	4	647.7M	4 871k

Initial Hyper-Parameters

For this set we looked at what was normally done for training a residual network:

- Kernel regularization
- Step-down learning rate (LR) with Adam (or Stochastic gradient descent with Momentum)
- No data augmentation to compare raw performances
- Batch Normalization after every non-shortcut convolution

The step-down LR divides the learning rate by 10 at epochs 80 and 120 and by 2 at epoch 160 and finally by 5 at epoch 180. The step-down approach is widely used [1, 23, 7]. The reason for the smaller steps at the later epochs is simply due to a division by 10 being too high and no division being too low.

Problems

Our first attempts were riddled with failures and overfitting. We noticed that training residual networks with quantized weights wasn't as simple as training floating point weights. We also saw a lack of generalization between the training accuracy and validation accuracy.

We identified that the problems were threefold:

- Some training simply didn't converge,
- Some major overfitting was noticed,
- Keras had trouble reloading some quantized models' weights when we did not freeze the batch-normalization layer.

Solutions

The floating-point model never had any trouble training, so we knew we had to explore the problems stemming from quantization. First we noticed that there was a problem with the residual connection: the addition is followed by an activation. This is problematic for non-float activations since they're all constrained in the $[-1, +1]$ range. Knowing that after a BN the values are recentered around 0 and brought back to that same $[-1, +1]$ range, when adding them both together we get values that are distributed in the $[-2, +2]$ range. We resolved this by dividing the result of the addition by 2 before feeding it to the activation layer. While this technique was not documented before, we felt comfortable using it because it resulted in a major jump in accuracy and conclusive results.

We solved the problem of reloading the model by evaluating the CIFAR-10 test-set without starting up a new program. If we ever needed to reload the models, we used a frozen BN layer, which seemed to solve our issue.

The overfitting mitigation is explained in the next section.

6.2.2 Impact of Training Methods

In this section, we study the impact of several training methods. In particular, we study:

- Data augmentation
- Initial learning rate
- Kernel regularization
- Adaptive layer quantization
- Shift-based batch normalization
- Merging layers into uniform building-blocks

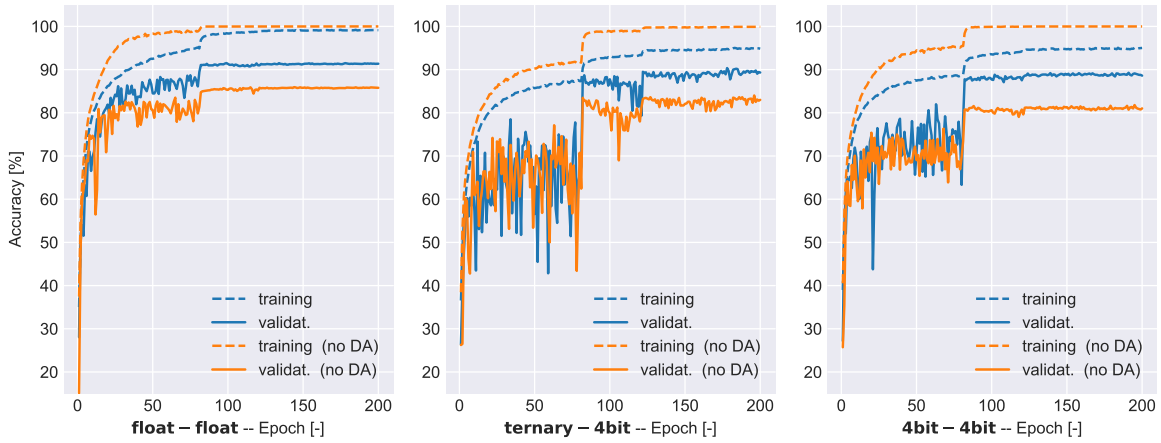


Figure 6.3: Comparison between data-augmentation and no data-augmentation

We included simple data augmentation, described in 6.1.2, during training in order to be in line with other works. As can be seen in Figure 6.3, data augmentation has a positive impact on accuracy, even when the augmentation is minor. When comparing training and validation accuracy, the figure shows that the omission of data augmentation leads to overfitting on the training set, and therefore lower accuracy on the validation set.

Because of its positive impact, we continued all training while using it, which subsequently allowed us to compare our results with related works. When using no data augmentation, the accuracy on the test set was below what we would have expected from the results of other papers.

The problem of choosing the best learning rate was circumvented by offering a fixed selection of 5 possibilities at the start: 0.001, 0.005, 0.01, 0.02, 0.1. This selection covers the most used learning rates for ResNets.

In Figure 6.4 the influence of kernel regularizers can be seen in its 3 different states: nearly imperceptible (float-float), completely detrimental (ternary-4bit)[4] and slightly

[4]the training was stopped after 20 epochs due to our non-convergence detector which stops the training if the validation accuracy doesn't increase fast enough

helpful (4bit-4bit). Due to this non-predictability we decided to exclude KRs from our training procedure. This could indeed have lowered the accuracy of 4bit-4bit networks in all configurations, giving the edge to ternary-4bit networks in final accuracy in the end. It should be noted however that 4-bit implementations still require some form of multiplications where ternary don't.

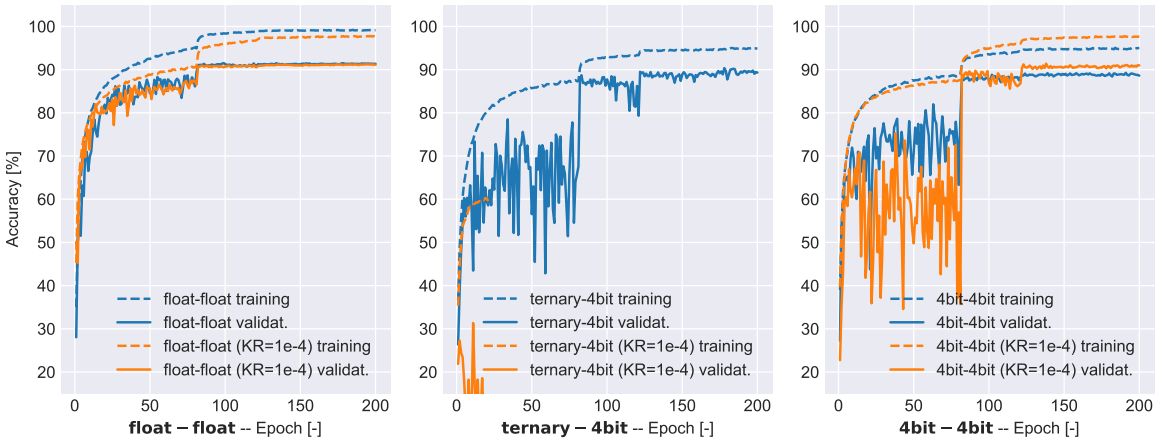


Figure 6.4: Comparison between kernel regularization and no kernel regularization

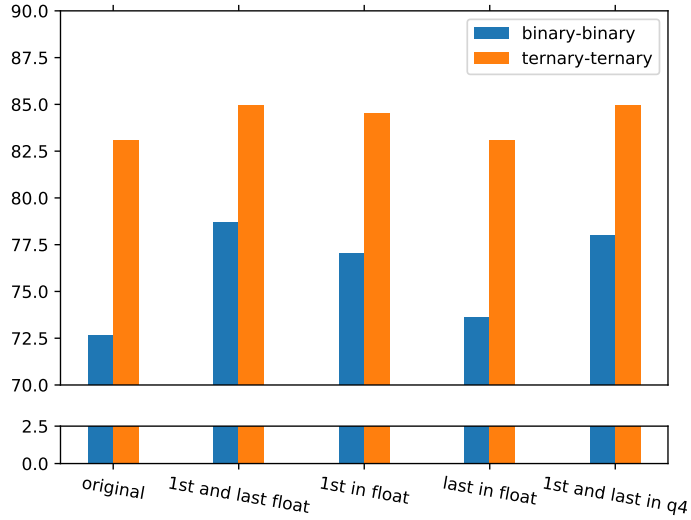


Figure 6.5: Comparison of different layer quantizations

Most papers on training highly quantized networks do not quantize the first and last layers, both activations and weights. In our initial design, we quantize all layers, except the input feature maps of the first layer (the original images).

It is however important to test the impact of quantifying different layers differently. In our experiment, we test different quantization strategies for the first and last layers. Figure 6.5 shows the result of the experiment. When we analyze the results for the binary case, we see that having the first layer in float has the highest impact on accuracy, while the last layer does not have as much impact. We also notice that when comparing float to quantizing with 4 bits, there is almost no difference for the ternary network, while there is only a slight drop-off for the binary one.

When comparing the results of the ternary network, we see that the impact of having all layers quantized is much smaller than for the binary one. This is one of the reasons we decided to go with a ternary-weight network for our final implementation. The first layer being only slightly prunable is something that was also observed on bigger networks (AlexNet, VGG-16) by [19].

To further modify the network for hardware compatibility, we switch the BN layer to shift-based batch-normalization. In short all previous multiplications are replaced by shift operations. This is achieved through approximation to the nearest power-of-two. The formula for batch normalization is:

$$y_i = \gamma \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}} + \beta \tag{6.6}$$

From (6.6), we can define a new γ and β, to reduce online computation:

$$\gamma' = \frac{\gamma}{\sqrt{\sigma_b^2 + \epsilon}} \tag{6.7}$$

$$\beta' = \beta - \gamma \frac{\mu_b}{\sqrt{\sigma_b^2 + \epsilon}} \tag{6.8}$$

And finally approximate γ' to the nearest power-of-two, and its sign, and round β':

$$\widehat{\gamma'} = round(log_2|\gamma'|) \tag{6.9}$$

$$\tilde{\gamma'} = sign(\gamma') \tag{6.10}$$

$$\widehat{\beta'} = round_{8bit}(\beta') \tag{6.11}$$

Using those definitions, we can define the SBN in equation (6.12). In this definition, the choice between left and right shift is determined by the sign of $\widehat{\gamma'}$.

$$y_i = sign(x_i) \cdot \tilde{\gamma'} \cdot \left(x_i \texttt{ << />> } |\widehat{\gamma'}|\right) + \widehat{\beta'} \tag{6.12}$$

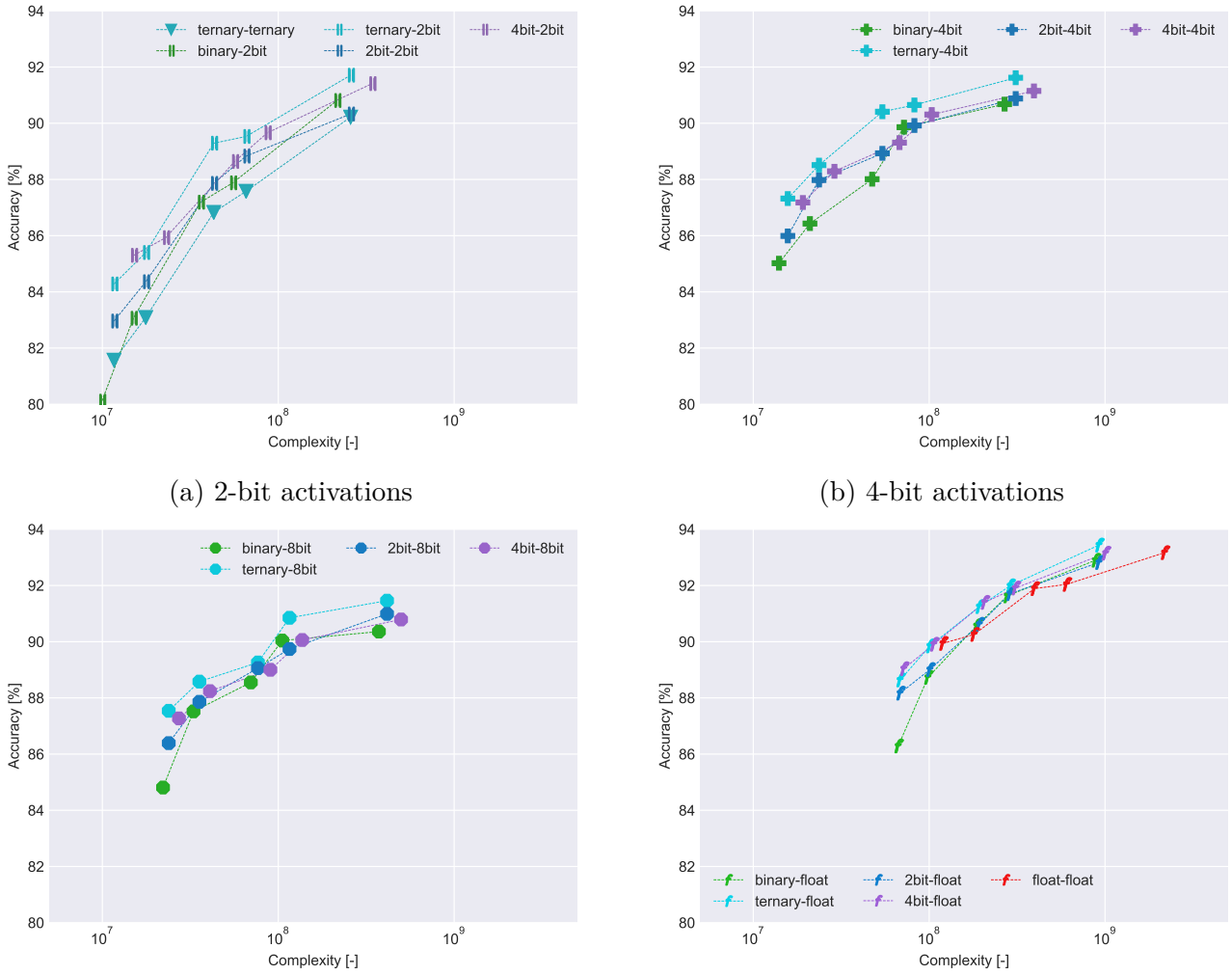
When modifying the BN layer to shift-based BN, without retraining, the accuracy drop is approximately 30% for a network with 4-bit activations (and $2\% - 10\%$ for 1-bit to 2-bit activations). After fine-tuning the network, by freezing the BN layer, we obtain an accuracy drop of less than 1%. The fine-tuning is done for only 20 epochs, making it worthwhile not to train the entire network from scratch.

During the design of our FPGA solution, we came upon the conclusion that the more branch-free we could make our implementation, the better. In order to do this, we decided to merge the convolutional layer, the BN layer and the activation layer. This would allow the implementation to have one generic design for every single layer, and not have to take care of corner-cases.

The other issue related with the hardware implementation is the number of bits needed after a convolution. While the input of a convolutional layer is the quantized activation, the output needs to have enough bits to fit the sum of the activations multiplied by the weights. This means that a full binary network, with a kernel of 9 weights and 64 input feature maps at the widest would need $\lceil \log_2(9 \cdot 64) \rceil = 10$ bits. In order to diminish global global memory transfers, it makes sense to put an activation layer after every convolution.

In order to achieve this change, we had to change the model, where even shortcut convolutions needed the BN and activation. This did not impact the accuracy of our models.

6.2.3 Impact of Weight Quantization



(a) 2-bit activations

(b) 4-bit activations

(c) 8-bit activations

(d) floating point activations

Figure 6.6: Comparison between same-activation networks. Each quantization option has 5 points. From left to right (in complexity order) those are: ResNet-14, ResNet-20, ResNet-14×2, ResNet-20×2, ResNet-20×4. The weights used to evaluate the test-set are the ones that achieved the highest validation accuracy.

The graphs in Figure 6.6 show the difference in accuracy for networks with the same quantization scheme for activations. They allow us to find the best weight configuration for a certain activation and thus remove combinations that don't have a good accuracy-complexity trade-off.

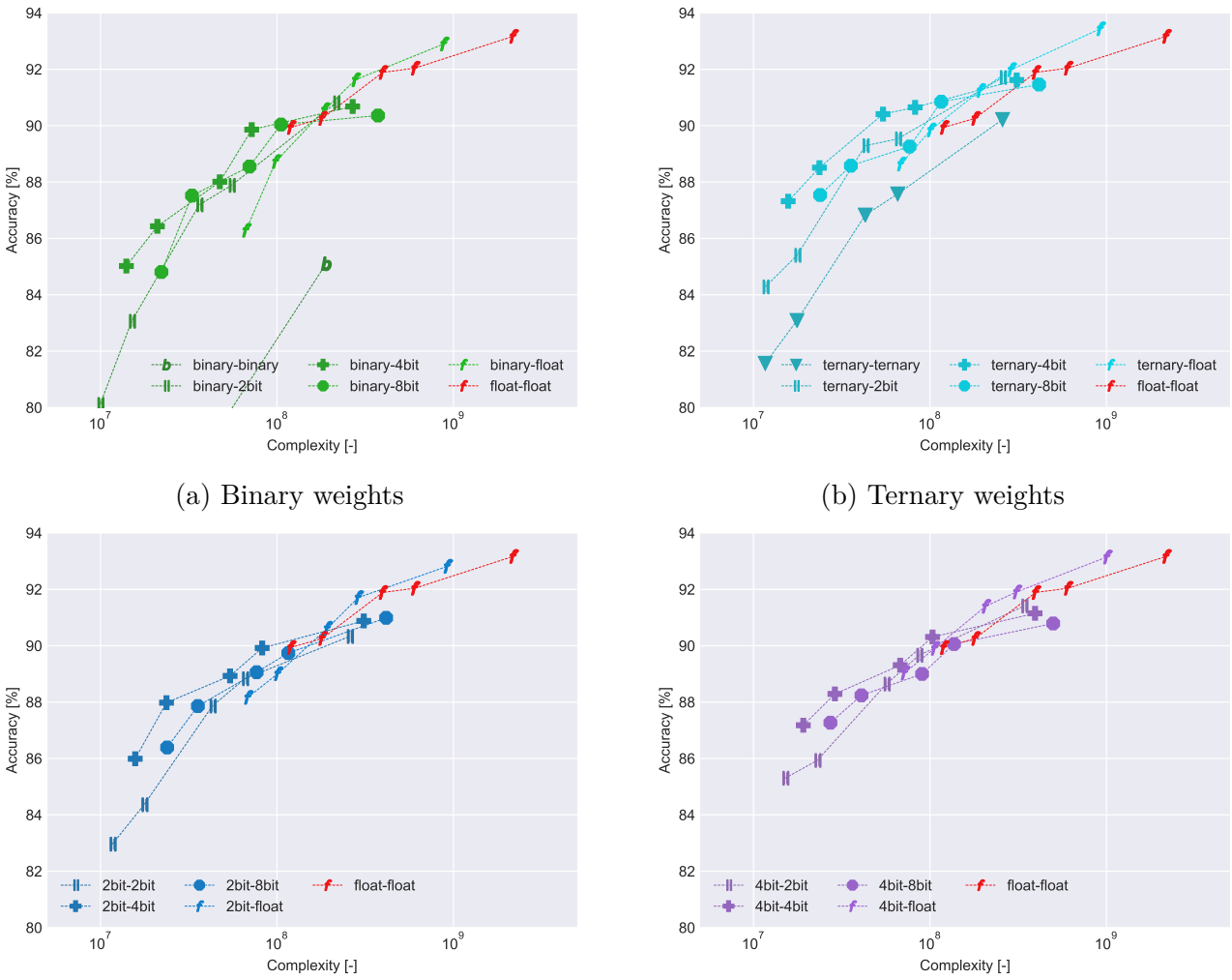
Figure 6.6a shows that when using 2-bit activations (either 2-bit Q or ternary), the best performing network uses ternary weights. The fact that higher bit usage does not necessarily mean higher accuracy is probably due to regularization from lower bit usage. The other figures show the same fact, ternary weights outperform all other weight quantization by a significant margin.

Ternary-weight networks even outperform the 4× widened full-float network. We believe this to be caused by the removal of the kernel regularizer, which had a negative impact on ternary weighted networks. But floating point weights do benefit from it

and ternary weights don't need them since our quantization scheme inherently tries to regularize the weights. Further experiments should be performed to test this hypothesis.

The difference between ternary weights and 2-bit weights is also quite noticeable. Both use two bits, and despite TWNs only using 3 out of 4 accessible values, they achieve better accuracy with same size networks.

6.2.4 Impact of Activation Quantization



(a) Binary weights

(b) Ternary weights

(c) 2-bit weights

(d) 4-bit weights

Figure 6.7: Comparison between same-weight networks. For the networks used, see Figure 6.6.

Figure 6.7 shows networks where the weights have been grouped by type. As can be seen from figure 6.7a, the full binary network has poor accuracy when compared to other networks at the same complexity level. Because we want to achieve near floating-point accuracy, the binary-binary network does not perform well enough for our use-case. Figures 6.7b to 6.7d show that quantized activations have a lower impact on accuracy than weights.

In Figure 6.7b, we see that in contrast to weights, where ternary weights outperformed 2-bit weights, it is the opposite for activations, where there is a significant difference

between ternary activations and 2-bit activations. In the same figure, when comparing 4-bit with 8-bit, the better network is the smaller one. The same is true in 6.7d. In Figure 6.7c, both 4-bit and 8-bit activations have approximately the same accuracy.

6.3 Results and Discussion

Due to limited resources it wasn't possible for us to test our results on a larger, more realistic dataset like ImageNet [54]. However, looking at the results presented in [41], it seems that widening a ternary-2bit ResNet-34 2 times is enough to nearly catch up to the full floating point implementation of said network. This corresponds with our observations. We see that widening our ternary, 2-bit and 4-bit weighted networks with quantized activations enables them to reach approximately the same accuracy as a non-widened float-float network. But once we widen that float-float network, the only weighted network that comes close with all its different quantized activations is the ternary one. We also observe that the 2-bit activation outperform the 4-bit activations on these 4× wider networks (with the exception of 2-bit weights).

The number of parameters scales quadratically with the width of a network. More parameters implies a need for stronger regularization. The lack of kernel regularizers could be felt by those wider networks. The 2-bit activation outputs the following possibilities: -1, -0.5, 0, 0.5. The ceiling of 0.5 could be acting as a regularizer during the back-propagation, clipping strong gradients to a lower value. 4-bit activations clip them to 0.992 or nearly 1.

The added (implied) regularization could also explain the nearly non-existing difference in accuracies between 4-bit and 8-bit activations. Meaning that the additional regularization of using only 4-bit activations offsets the added precision of 8-bit activations. It could also simply be that 4 bits are enough to store the information in the $[-1, +1]$ range and the drop-off compared to full floating point is due to the range being more limited.

The results of the binary networks were overall quite disappointing. The binary weighted ones lagged behind slightly more expensive quantizations in terms of accuracy and the full binary network had a lot of trouble peeking above the 80% accuracy barrier. While this could be alleviated by quantizing the first and last layer less, it is a compromise that will result in more hardware utilization in real-world implementations.

Even though our networks are quite small, [55] has proven that to obtain acceptable results on Imagenet, the full 224×224×3 resolution wasn't needed. Using a more conservative resolution of 64×64×3 (bicubicly downsampled from the original) combined with Wide Residual Networks [56] yielded results which had an error rate of 32.34%/12.64% (top-1/top-5). Moving down to 32×32×3 their results show a 5 point increase in the top-5 error rate. Compared to a full-fledged (non-widened) ResNet-34 with error rates of 26.77%/8.67%, which has an approximately 2× increased complexity, these results prove there are use cases for smaller networks even on real-world tasks.

In Figure 6.8 the selected models all have a relatively low complexity. We only selected those that could attain a relatively acceptable level of performance. For this reason the full binary networks were excluded. High-bit activations also significantly increase the model complexity (without much improvements in terms of accuracy) and were therefore excluded from this final round-up.

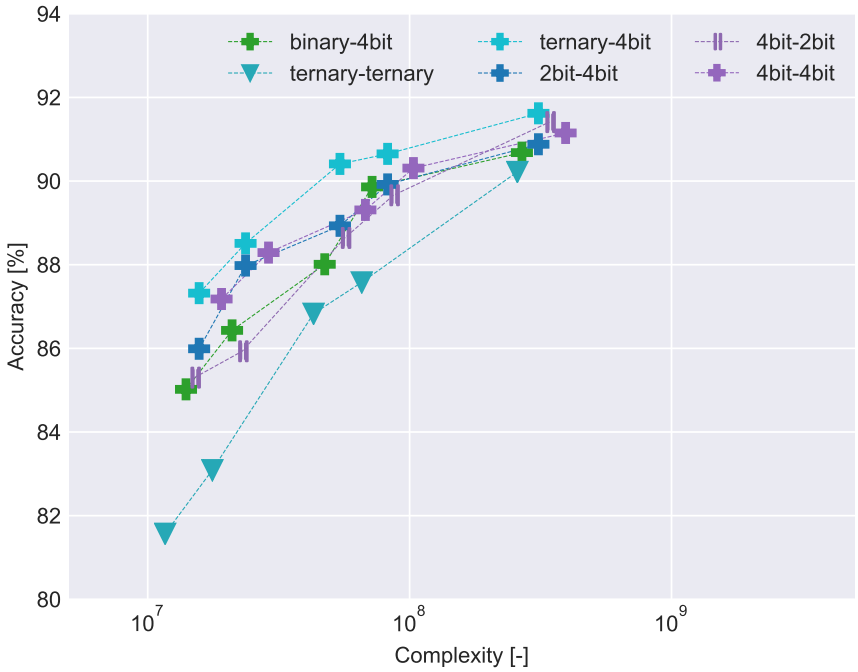


Figure 6.8: Models of interest. For the networks used, see Figure 6.6.

It should once again be emphasized that this selection will happen based on our complexity formula. We believe it to be a more accurate metric than to only look at the amount of MACs or the global memory transfer (both can still be found in Appendix B). This modeling does not in any way pretend to be better than a model tied to a specific implementation. We did the model selection beforehand and wanted to find an implementation which possessed a good accuracy-throughput trade-off.

We wanted to target an accuracy above 88% for our final design and select a model which had a reasonable drop-off compared to its floating point counterpart. When minimizing for model complexity, this leaves us with the 3 following candidates: ternary-4bit, 2bit-4bit, 4bit-4bit. In our experiments 4-bit activations strike a sweet spot in the complexity-accuracy trade-off. We note that using ternary weights is most profitable from a computational standpoint and also manages to achieve the best performance at slightly lower or equal complexity. For these reasons the selected model was the ternary weighted and 4-bit activated one.

Choosing the specific model from the curve requires to trade off either accuracy or model complexity. In our analysis, we prefer to lose a bit of accuracy in exchange for better latency. We therefore choose the second point from the left (a ResNet-20), which has a model complexity close to $5 \cdot 10^7$ with an accuracy of 88.5%. Another point to consider is the third point from the left, which has a big jump in accuracy. This ResNet-14×2 does possess 6 less convolutional layers but at the expense of 4 times more MAC operations at each layer (except at the first layer where the increase is only twofold).

6.3.1 Further Work

As we've already established, a possible improvement would be to use a different quantization scheme for the activations since those seemed to be the biggest bottleneck

when paired with quantized weights.

During our comparison between different network quantizations we used 32-bit floats. It would also be interesting to compare them with 16-bit float-float networks. We expect the incurred accuracy penalty to be minimal, but the complexity gains would be impactful and help reduce the difference in complexity between quantized and floating point implementations.

The inconsistent behavior of kernel regularizers merits a closer inspection. Since its effects range from slightly beneficial to outright detrimental. This might be explained for ternary networks, due to the regularization which is already included in such quantization. However its positive effect on 4-bit weighted networks, compared to floating point ones seems strange. This does merit further analysis.

While a substantial amount of research has been done on quantization, very little has been done with different activation styles for ternary weights on large networks. Thus, the next step would be to scale up our experiments to the ImageNet dataset with the bigger ResNet architectures. 4-bit quantization has also not been studied that much, often brushed aside as not being accurate enough. But we do show that on smaller networks both ternary and 4-bit weights are a great alternative.

Further work could also be done on quantifying the real model complexity of the different networks listed in this chapter. While our model complexity analysis allows comparing two networks theoretically, being able to compare networks from a latency or throughput perspective might be interesting.

While we widened our residual networks, we didn't do it based on the methods proposed in [56]. They propose a different re-ordering of the CONV-BN-ACT structure to BN-ACT-CONV for better training results and an additional drop-off layer between convolutions to regularize the widened weight channels. The re-ordering has also the unpleasant consequence of either augmenting global memory transfer or resource utilization in order to move the non-quantized values on the 2 branches of the ResNet-block. This would however remove the need for the division by 2 we added after the add-layer. Such wide networks are also slower to train and it wasn't realistic to train them on so many configurations.

Chapter 7

Inference On FPGA

Following the framework of the previous chapter, we first explain the methodology, and more precisely we discuss the implementation details of both our baseline implementation and the optimizations on top of it.

The second section will show the experiments we performed, including comparing different designs and several unrolling, tiling and loop order configurations.

The third section will wrap this part up by synthesizing the results and providing a thorough analysis of our results and possible improvements.

In this chapter we will also go over the constraints we worked under, be they self-imposed or due to the platform, or time limitations.

Our code is available at `https://github.com/victorjoos/fpga`

7.1 Implementation Details

7.1.1 Cyclone V design

Before detailing the implementation, it is important to show the architecture of the FPGA used, shown in Figure 7.1. Most of the features of the Cyclone V are the same as other FPGA devices, the only difference might be the ARM A9 cortex included in the design. An FPGA device mostly contains Adaptive Logic Module (ALM, in Intel FPGA terminology). Those modules contain elements which can be used for different functions. They can be used as register files, logic modules, and arithmetic modules. The register files are the first and closest type of memory that can be used for the FPGA.

In addition to the ALMs, the Cyclone V also contains embedded memory blocks (M10K, local memory), which are blocks of 10 kilobits of memory. These can be used as a second layer of memory. Because of their size, more information can be kept in them, but the latency to acquire information is greater than from register files.

The last blocks available to an FPGA programmer are Digital Signal Processing (DSP) blocks. These can be used for floating-point and fixed-point computations, addition, multiplication, or both (Multiply-accumulate, MAC operations). The advantage of this block type is that it was optimized for some types of computation, and will be fast and automatically pipelined.

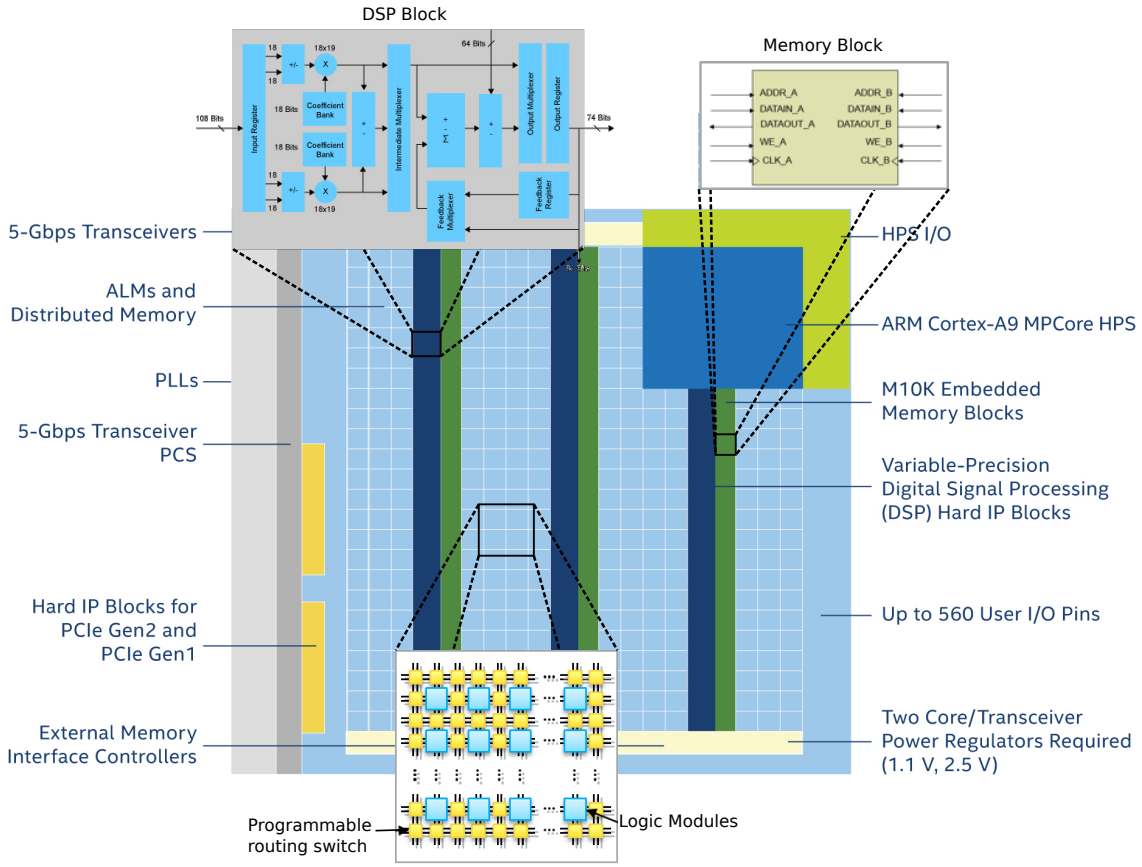


Figure 7.1: Cyclone V FPGA design [26]

7.1.2 Baseline implementation

The first step in the implementation of our FPGA accelerator was to be able to compare solutions with each other, but also to be able to compare them to a baseline implementation that was as simple as possible.

To achieve this, we implemented almost every layer in C to be run on the CPU, except the convolution which was multi-threaded through a naive OpenCL implementation. The main drawback of this approach is that the output of every layer must go back to the host memory. To overcome this problem, we make the assumption of using an FPGA with shared memory between the host and the FPGA. This is the case on the Cyclone V where a Cortex-A9 ARM processor is baked in to the FPGA, as can be seen from Figure 7.1. We use a version of the FPGA firmware that uses shared memory between the CPU and the FPGA for fast memory access.

If no processor were present, it would have been possible to program the hardware to have an additional processor core, at the expense of a lot of FPGA resources.

In the following subsections, each layer will be explained with, when necessary, some code or illustrations to follow along.

Host

The host, in our case the ARM processor, will contain most of the code of our baseline implementation. From previous work [39], it was clear that the convolution was the most resource-intensive part of CNNs. This is why the baseline implementation focuses on accelerating the convolution on FPGA, and no other layers.

The design of the host is quite straightforward, and tries to be as modular as possible for different deep learning architectures. Every layer (even the convolution, which will be run on FPGA) is called as a function. The input feature maps and other parameters (weight kernels) are passed as input and the function will return the output feature maps. The structure of the feature map is as follows:

```
typedef struct feature_map {
    cl_short * values;  // array on host
    cl_mem fpga_values; // same array as values but for FPGA
                        // (OpenCL shared pointer)
    int nchannels;
    int fdim;
    int fsize;            // = fdim*fdim
    int mem_buff_channel; // impl. det. : for memory recycling
} fm_t;
```

The structure of the convolution weight parameters is almost the same as the other structures (weights for the fully-connected layer, batch normalization):

```
typedef struct conv {
    cl_uchar * kernel;  // host
    cl_mem fpga_kernel; // FPGA (OpenCL shared pointer)
    int xsize;    // kernel size
    int size_in;  // number of channels
    int size_out; // number of weight maps
} conv_t;
```

With these data structures, it becomes very easy to program any network. Below, the program for a LeNet5 [6] network is shown as an example. Not shown is the instantiation of the OpenCL kernel :

```
int inferLeNet5(char* img, conv_t convs, bn_t bns, dense_t denses) {
        fm_t* fm = img_to_fm(img);
        fm = convolve(convs[0], fm, 1 /* strides */);
        fm = activate(fm, TANH);
        fm = max_pool(fm);
        fm = convolve(convs[1], fm, 1 /* strides */);
        fm = activate(fm, TANH);
        fm = max_pool(fm);
        fm = convolve(convs[2], fm, 1 /* strides */);
        fm = activate(fm, TANH);
        fm = max_pool(fm);
        fm = fully_connect(denses[0], fm);
        fm = fully_connect(denses[1], fm);

        /* Infer class */
        return max_index(fm);
}
```

FPGA device

For our baseline, we implement a simple convolution. The convolution iterates over four domains (and 6 loops), as can be seen in Figure 7.2. As explained in section 2.1.3 the order of the loops is as follows:

```
for outf in 0..Nout: # output loop (loop-4)
  for r in 0..Nrow:    # spatial loops (loop-3)
    for c in 0..Ncol:
      for inf in 0..Nin: # input loop (loop-2)
        for k in 0..Nk:    # kernel loops (loop-1)
          for l in 0..Nk:
            fmap_out[outf][r][c] += fmap_in[inf][r+k][c+l]*w[outf][inf][k][l]
```

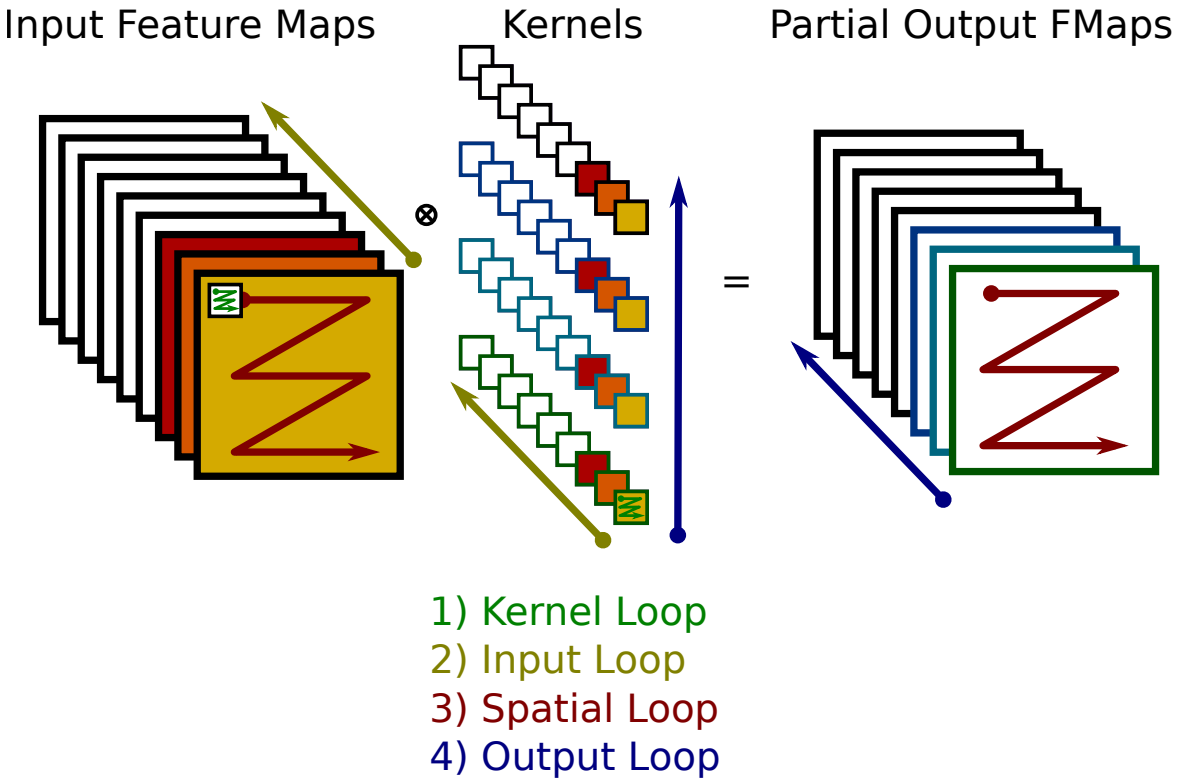


Figure 7.2: A convolution consists of 4 loops, detailed in the figure, the order of the loops is not fixed and can be optimized according to different criteria. The most important criterion is memory management

The first (inner-most) loop accumulates the multiplication between the weights and a kernel centered around the variable of the second loop. The third loop iterates over the input space, while accumulating the variable in the (x,y) dimension. The last loop iterates over different weight kernels, to output different feature maps. As explained in section 2.1.3, all input feature maps will be combined into only one output feature map. Using OpenCL, it is quite straightforward to implement such an algorithm.

For our baseline approach, we test both approaches of OpenCL kernel design. We first test an NDRange kernel with a global id on the output feature maps. We also test the same kernel in single-work-item mode. Our following implementations will either be

NDRange kernels or single-work-items, due to decisions made during implementation. As will become clear in the following sections, our final design will use a single-work-item, due to the ease of optimization and better resource utilization compared to an NDRange kernel.

The baseline implementation has a latency per image of 3 seconds.

7.1.3 Optimizations

On top of our baseline approach, it seemed immediately obvious that some layers should be merged together. To achieve this, it was important while training to have a model with as few macro-blocks as possible. As explained in section 6.2.2 we added batch normalization and activation layers after every convolution. This allowed us to have one single kernel structure, with only one critical path through the kernel.

This simple optimization reduced the latency per image by a factor of 3. Mainly because the BN and activation now took place on the FPGA which eliminated the need to iterate over the image after the convolution was done.

When comparing time spent for every layer, we can see in figure 7.3 that the convolution is by far the most computationally expensive operation. This is why optimization of this layer is especially important.

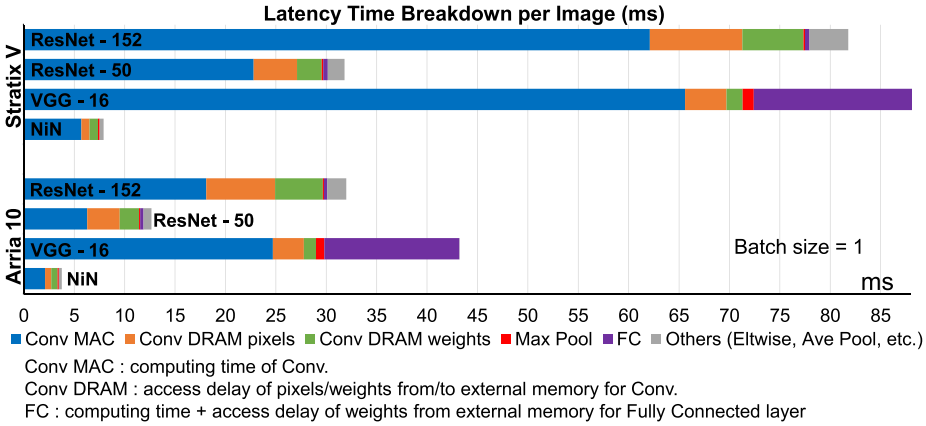


Figure 7.3: Graph from [39] showing the time spent accelerating each layer. We see clearly that the fully connected layers are almost negligible compared to the convolutions, especially for residual networks.

Memory transfer is often a bottleneck when designing FGPA applications [38]. However, by carefully designing a multi-layer cache, it is possible to minimize this bottleneck. Figure 7.4 shows the different types of memory available to us and their size.

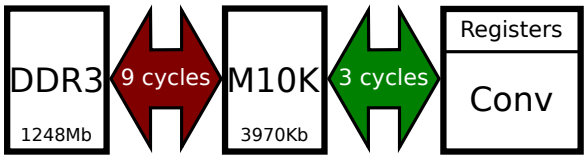


Figure 7.4: Memory hierarchy of the Cyclone V FPGA, DE1-SoC

To optimize the use of this memory hierarchy, multiple techniques have been used. In particular tiling, loop transformation, and local memory promotion. To be able to choose the best final structure, it is important to perform a thorough analysis of the different design objectives. Ma et al. [39] detail the different objectives that are important to minimize :

- Loop unrolling is the most important factor for computing latency
- Inefficient use of PEs
- External (and internal) memory transactions
- Partial sum storage requirement depends on the loop order
- Reduce on-chip buffer access, by loop unrolling
- Reduce external memory access, depending on tiling factor

While minimizing the previous objectives, it is also important to maximize area usage of the FPGA, since unused space is wasted doing nothing instead of accelerating computation.

Tiling

To enable such optimizations, we first make use of tiling, as explained in section 3.5.2. The pseudo-code for a tile is as follows :

```
for tr in 0..Tr..Nr:
  for tc in 0..Tc..Nc:
    for ti in 0..Tin..Nin:
      for to in 0..Tout..Nout:
        # Load weights and feature maps
        # Convolution for one tile
```

When processing a tile, we can choose to unroll a loop over another. In an FPGA, it is almost always better to unroll as much as possible, as this will enable fully parallel computations. Figure 3.6 shows the four possible unroll configurations. The first unrolls the loop over the kernel, the second unrolls the input feature maps, the third unrolls the spatial loops and the last unrolls the output feature maps. Each unroll has its benefits and drawbacks, which will lead to different bottlenecks.

The meta-parameters, which were described in section 3.5.2, are synthesized in table 7.1.

	Kernel	**Spatial**	**Input**	**Output**
Convolution Loops	loop-1	loop-3	loop-2	loop-4
Dimensions	Nk, Nl	Nr, Nc	Nin	$Nout$
Loop tiling	$Tk, Tl^{\dagger}$	Tr, Tc	Tin	$Tout$
Loop unrolling	Pk, Pl	Pr, Pc	Pin	$Pout$

† *In our implementation there is no tiling loop for the kernel ($Tk = Nk$ and $Tl = Nl$).*

Table 7.1: Tiling meta-parameters

According to these parameters, we can search an optimal (or convenient) solution through two flowcharts provided by [39], the results of which are provided in Figure 7.5. Figure 7.5a defines the size of the local memory for the partial sums, while Figure 7.5b defines the number of DRAM (global) accesses. The decision process for each choice is discussed in the next paragraphs.

For the choice of partial sums we keep in local memory, we first encounter a problem of logic resources. We cannot fully unroll both the kernel loop (3×3) and input loop (64-128). Neither can we fully buffer the input loop, by having the tile size equal to the maximal size of the input fmaps. This leaves us in the part where we compare the loop order of the *outer* loops (the inter-tiling loops). We do compute the input loop first (we do not loop over the kernel loop, and therefore it is fully buffered), the amount of partial sums we have to keep in local memory is therefore $Tout \times Tr \times Tc$. This value is acceptable to us, because it can be easily adapted by a larger or smaller tiling of the output loop and the spatial loop.

To find the amount of global accesses we need for our implementation, we proceed as we did for the partial sums. We cannot tile the spatial loop in addition to the input loop, nor can we tile the input, output and kernel loops simultaneously. As such, it is not possible to buffer all fmaps or all weights, and we cannot guarantee only having to access each input only once. In addition, we tile neither the output loop, nor the spatial loop fully, and therefore find ourselves in the bottom part of the flowchart. We do however compute the spatial loop first (loop-3), as it is the inner-most loop of the convolution of one tile. We therefore only need to access each weight in global memory once, but need to access the input pixels $Nout/Tout$ times. To make this value as small as possible, we try to make $Tout$ as big as possible, and for some designs are able to fully buffer all required weights for one pixel, which makes the number of accesses to global memory equal to the number of pixels.

Because of the use of ternary weights, our weight matrices are sparse as can be seen in Figure 7.6, and according to research by [19], it is possible to prune networks and retrain them to achieve even better sparsity without losing accuracy.

This sparsity allows us to try the following configuration: looping over the input image (row and column) as inner-most loop and (fully) unrolling this spatial loop. Because a lot of weights are zero, we then put a skip (an if block) around the spatial loop. We will see in the following section what the effects are of different sparsity coefficients.

Due to the fact that memory bandwidth is a bottleneck, we try parallelizing the reading of weights and input feature maps, and pipeline the reading with the convolution.

We also pipeline the writing of the output feature maps, but observe no real performance gain for it. It is however easier to add the activation and SBN layers when writing to global memory instead of during the convolution.

Creating three additional kernels adds some overhead to the area utilization of the FPGA. However, working with channels allows better control of the results of the synthesis.

The final architecture of our implementation can be found in Figure 7.7. It shows the interconnection between the loading of weights, the feature maps, and the batch normalization constants with the convolution, and the writing of the result back to memory. The bottleneck of our approach can be found in the implementation of the convolution operation.

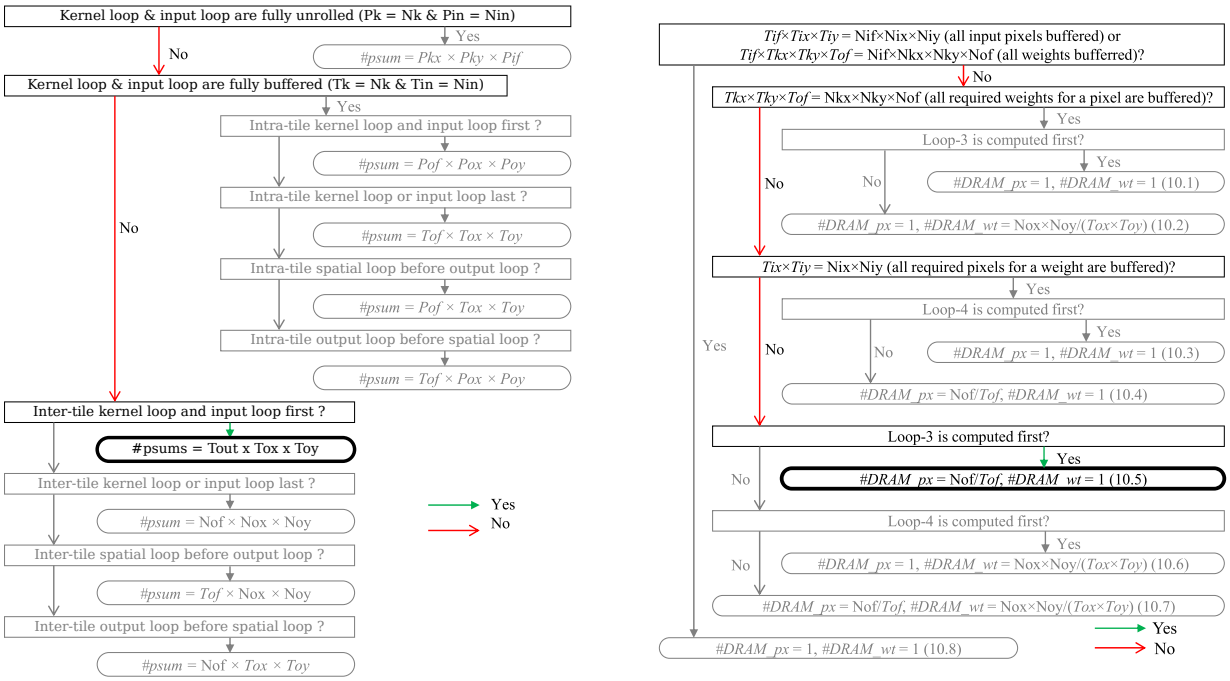


Figure 7.5: Flowcharts from [39] adapted to our case. (a) determines the number of partial sums that must be kept in local memory. (b) determines the number of DRAM accesses needed for our implementation. Tix, Tiy are equivalent to our Tr, Tc variables.

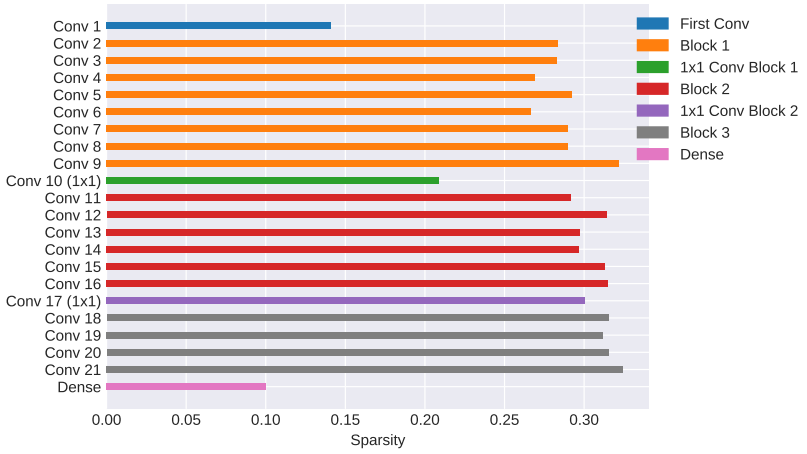


Figure 7.6: Sparsity by layer

7.1.4 Creating and Profiling Kernels using the OpenCL Offline Compiler

With a kernel design built, the next step involves running the design through the OpenCL compiler.

To be able to run our design on a DE1-SoC using the current version of OpenCL, we used firmware from [57]. This allowed us to use Intel FPGA SDK version 18.1, compared to the supported version by Terasic which was version 16.0.

One problem with the hardware we used, was that the report did not have the correct values for the resources available on our FPGA. However, this did not have an

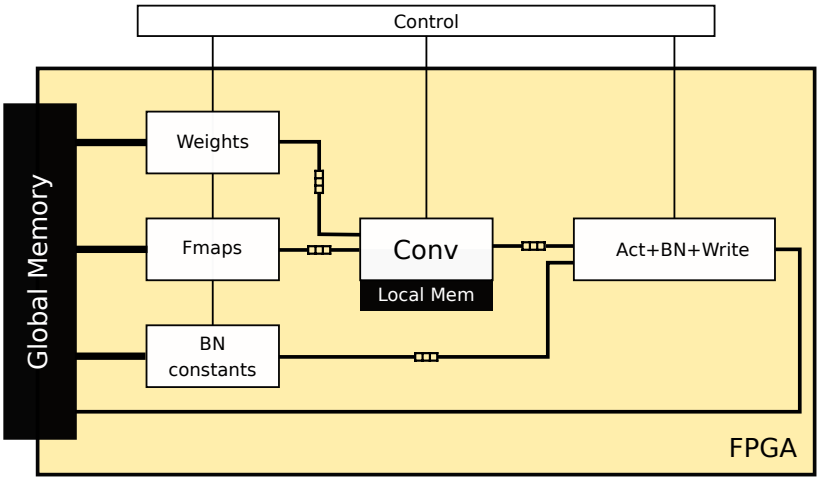


Figure 7.7: Architecture of the implementation. Every block inside the FPGA region represents one OpenCL kernel in our design. The kernel-to-kernel communication happens through buffered channels. In our case, the control is handled inside the CPU.

impact on the generated hardware.

For each design, we followed the steps outlined in [26], and the flowchart available in Appendix C. The report created by the compiler allowed us to check for area utilization and estimate what the design would look like on FPGA.

After compiling for 1 to 2 hours, we get the report of the true area utilization. The estimation from the previous report is not always correct and is highly dependent on the OpenCL features used.

Once a design has compiled from OpenCL to RTL, and has subsequently fitted, it can be tested on the device itself. Testing the design on the DE1-SoC was straightforward for the most part, since most testing was done through the simulator. One important difference between the simulator and the FPGA was that they did not have the same effect for a left-shift when using signed integers. The first one used an arithmetic left-shift, while the second one used a logical left-shift. Our solution was to replace shifting by a constant division, which would then be synthesized to an arithmetic left-shift (when dividing by a power of two).

7.2 Experiments

This section will describe the experiments that were done in order to ensure that an optimal solution was found. To confirm that the values obtained are repeatable enough, we run all experiments 100 times on 100 images, except when stated in the test. We can see from most images in this section that the variance is small enough not to need a larger amount of tests.

7.2.1 Reducing Logic Utilization

In this section, we analyze the impact of two methods that should reduce logic utilization on the FPGA. The first is moving from floating-point to quantized neural networks, and in particular a ternary weight network with 4-bit activations. The second method is modifying the batch-normalization to shift-based batch-normalization.

Transfering the floating-point representation to the quantized representation with a similar architecture reduced the DSP utilization by almost 70%, from 100% utilization to 31%. Only fixed computations needed for the loop control still used DSP blocks. The optimizations possible for the quantized network were more varied than for the floating-point network. The unrolling factor of the floating-point network was bound by the number of DSP blocks available, while this was not the case for the quantized network. The amount of local memory and logic utilization used is also reduced, due to the lower bit-width.

We didn't try to fit the classic BN operation on the FPGA, going straight to the SBN implementation. While it probably would have fitted, we still wanted to get rid of any multiply operation in computation. Later changes could suddenly benefit from a bigger unrolling factor only to be met by a DSP shortage due to the BN. This fact alone is enough for us to consider the usage of SBN as an improvement over BN, even if there is a small accuracy loss of less than one percent. This loss could probably even be made negligible by better training, as in [32]. As explained in chapter 6, we fine-tuned the network after initial training, while freezing the layers of the SBN. Using an SBN during the whole training period should be considered, and according to [32] SBN had no negative impact on accuracy.

7.2.2 NDRange Kernel versus Single-Work-Item Kernel

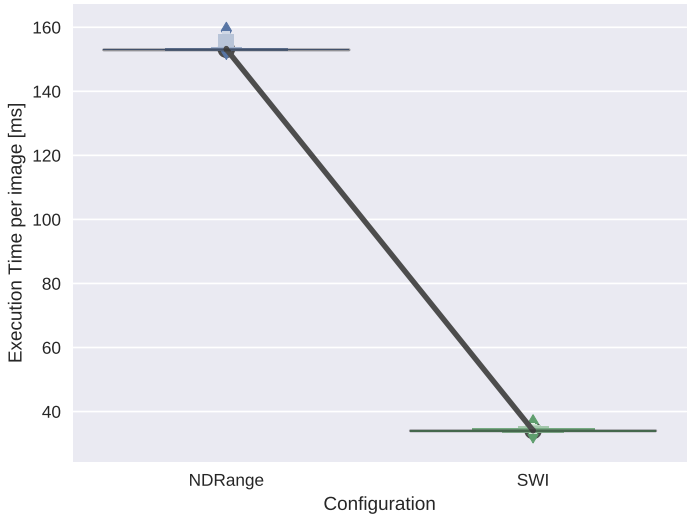


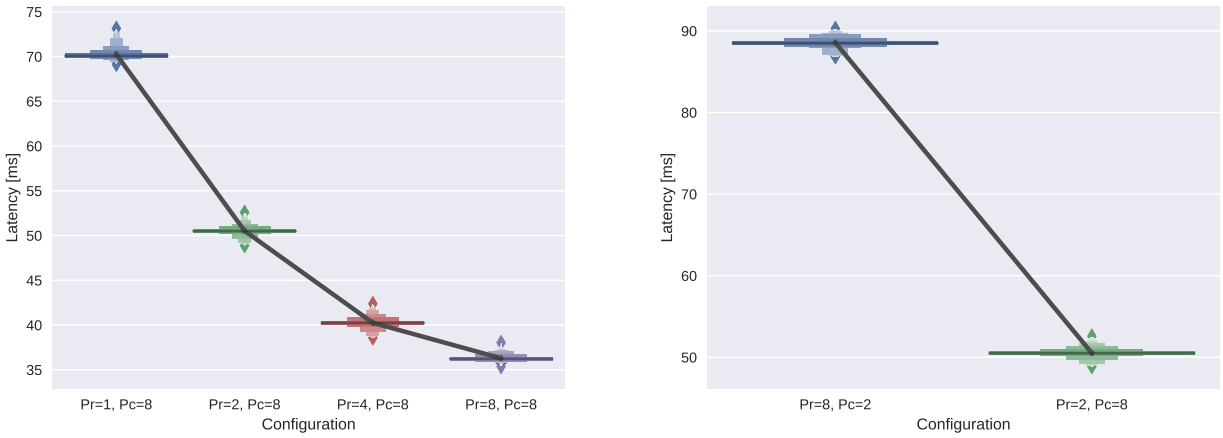
Figure 7.8: Comparison between NDRange kernel and Single-Work Item

According to [26, 42], it is recommended to write single-work-item (SWI) kernels, for most use-cases, unless "your kernel program benefits from explicitly describing multiple concurrent threads" [42]. Because this is the case for our application, we decided to compare the possible performance we could gain from either method. We found early on that the compiler was not able to optimize NDRange kernels to the extent it could optimize SWI kernels. Figure 7.8 shows that the single-work-item outperforms the NDRange kernel by a large margin.

We focus on a simple design, but one that utilizes tiling for both versions. While this test shows that SWI outperform NDRange kernels, it is important to note that this is not the case for naive versions of both kernels. Simple implementations are advantaged

on NDRange kernels, due to simple parallelism. From a designer's perspective, it is however easier to understand the possible improvements one can make to a SWI.

7.2.3 Impact of Unrolling and Tiling



(a) (b)

Figure 7.9: Impact of unrolling. Both graphs have a tiling of $Tr = 16$ and $Tc = 8$. (a) shows the trend of unrolling Pr with Pc constant. (b) shows the difference when unrolling more on Pr or Pc

Unrolling seemed one of the deciding factors for the performance of our implementation. This makes sense intuitively because unrolling improves parallelism, which is very important for the convolution operation.

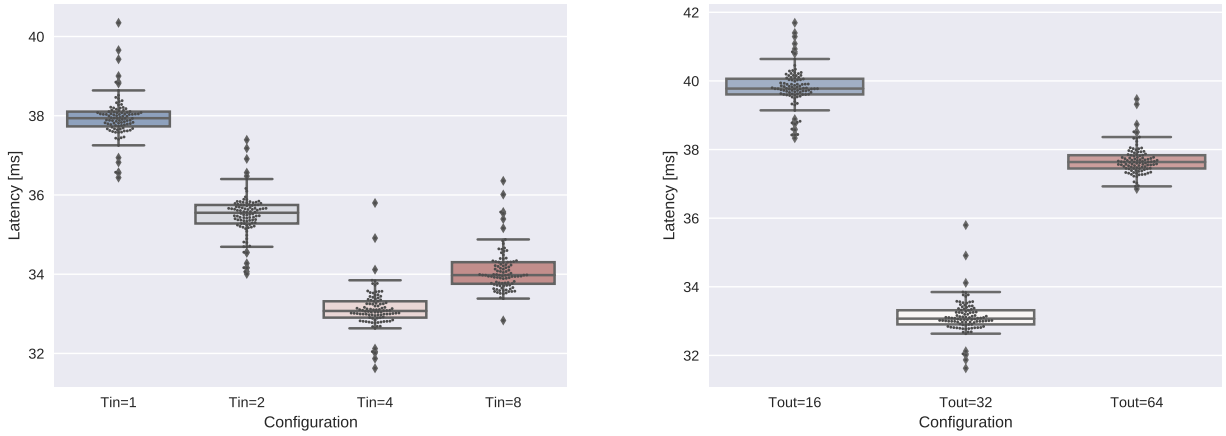
We therefore tested several unrolling strategies, which will be outlined here. Our first decision point was which loops to unroll. Due to the area utilization of our non-unrolled solution, it was quite hard to unroll anything other that the two inner loops. Due to this, our test is limited to varying Pr and Pc. We fix the tiling of all loops, and in particular fix Tr at 16 and Tc at 8.

In Figure 7.9, the impact of unrolling the two inner loops is shown. Figure 7.9a shows an inverse linear relationship between the latency and the unroll factor, which is expected. From this, we can analyze the overhead of the memory transfers and kernel enqueuing, which is roughly 30ms, by figuring out the asymptote of the function.

Figure 7.9b shows the unroll of Pc compared to the unroll of Pr, while keeping the total unroll factor constant. We would expect to see a constant latency, because the unroll factor is constant. That this is not the case means that the OpenCL compiler could not optimize the memory accesses of an inner loop where the unroll and tiling is not equal. From this experiment, we therefore conclude that we must always keep Tc equal to Pc.

We analyzed the impact of tiling on our implementation. In Figure 7.10 we plotted the impact of varying Tin in Figure 7.10a and varying $Tout$ in Figure 7.10b.

In Figure 7.10a, we can see that bigger tiling is not always better, but it is important to balance the number of memory accesses with the number of computations that need to be done. In our tests, we came upon the conclusion that $Tin = 4$ was a good value for our network.



(a) (b)

Figure 7.10: Impact of varying Tin and $Tout$. (a) shows all tiling fixed at $Tr = Tc = 16, Tout = 32$, and varying Tin from 1 to 8. (b) shows tiling fixed at $Tr = Tc = 16, Tin = 4$, and varying $Tout$ from 16 to 64.

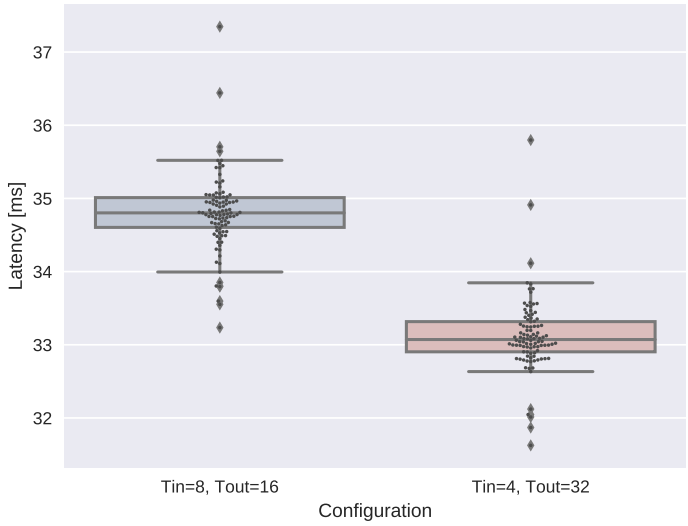


Figure 7.11: Comparing tiling configuration where the product $Tin \cdot Tout$ is constant

Figure 7.10b shows the same conclusion as for Tin. It is important to notice that we can tile the output loop to a bigger extent than the input loop without stalling on the trade-off between memory and computing. In our tests, we find that 32 is the recommended value for $Tout$. 64 would probably be even better, but the FPGA lacks the resources (with this architecture at least) to test it.

When comparing tiling for Tin and $Tout$ where we keep the product of both constant and equal to 128, which was the product of the best performing configuration of the previous test, we can see in Figure 7.11 that the difference is much smaller than when simply changing one tiling. In spite of this, the best configuration is still a tiling of $Tin = 4, Tout = 32$.

After finding values for Tin and $Tout$, we turn our attention to finding optimal values for Tr and Tc. Due to our loop configuration, and as explained in the previous paragraphs, we keep the unroll factor of the inner-most loop equal to its tiling factor, that is $Tc = Pc$. We do keep the total unroll factor equal to 64, this means that for

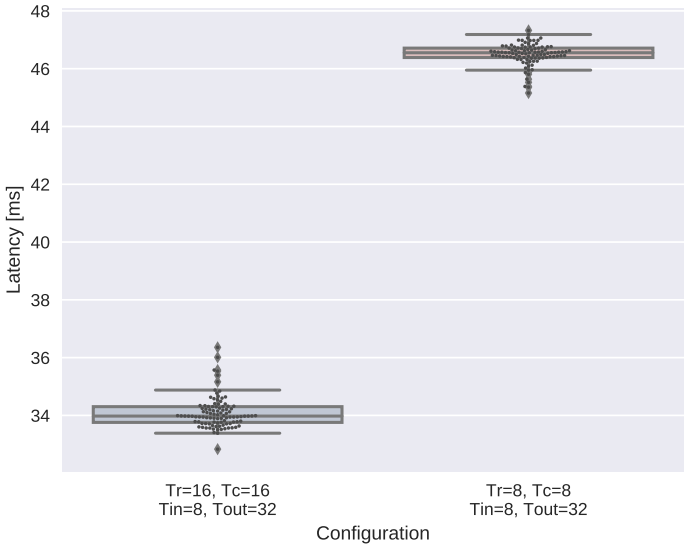


Figure 7.12: Impact of Tiling Tr and Tc

$Tr = Tc = 8$, we have $Pr = Pc = 8$, but for $Tr = Tc = 16$, we have $Pr = 4$ and $Pc = 16$. In our test, shown in Figure 7.12, tiling at 16 is better than 8. From this it might be interesting to wonder what would happen when tiling Tr and Tc at 32. However this design did not fit on the DE1-SoC device due to memory limitations on the partial sums stored in local memory.

7.2.4 Impact of Sparsity

To test the impact of sparsity on the performance of our framework, we modified the threshold of the ternary quantization, without retraining. By not retraining the network, it was possible to get different percentages of sparsity by one simple parameter change. Another solution to this problem would have been to randomly put weights to zero, but the thresholding will try to keep the locality of weights that are zero, which might be important. We run inference on our network for 1000 images, and run this experiment 100 times.

The results of this experiment can be seen in Figure 7.13. From this figure, we can see a speedup between no sparsity (th=0.0) and all zeroes (th=2.0). We notice however that the difference between all thresholds is small, and while statistically significant, sparsity doesn't give a high boost to performance, even when all weights are zero. The difference is in the range of 0.25 ms between no sparsity and regular sparsity (th=0.7, in white), which is a small benefit and the gain we can expect for our kinds of networks. This conclusion means that while our current implementation is relatively fast, compared to our previous efforts, we still don't gain much from not doing a big part of the convolution.

When comparing the version which takes advantage of sparsity (through an if statement around the spatial loop) and a version which doesn't take advantage of it (the if statement is moved inside the spatial loop), the first version is almost twice as fast as the second version. This result and the one before it, show that the first code is easier to synthesize for OpenCL and Quartus than the second.

We have found no plausible explanation to that fact in the report output by the OpenCL compiler. The slower code shows a better pipelining profile, and smaller

initiation interval, but is ultimately slower.

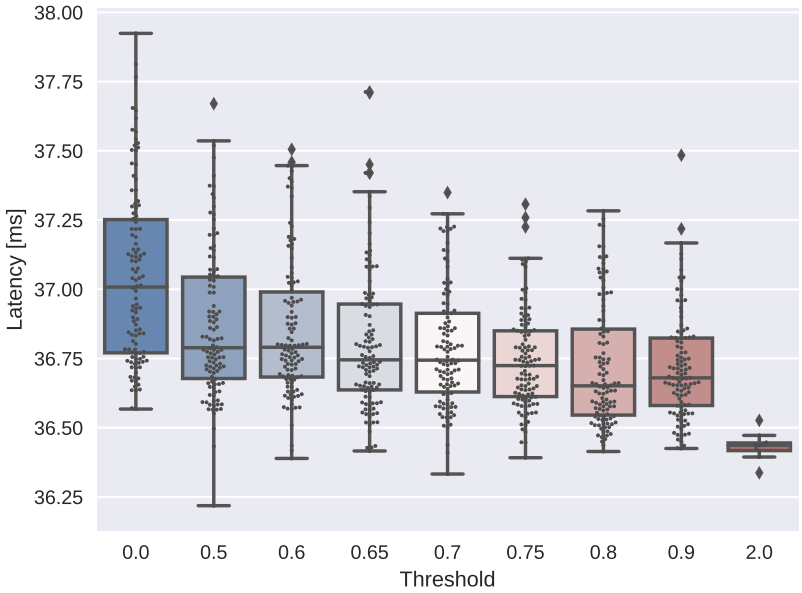


Figure 7.13: Execution time in function of sparsity. Execution time is counted for 1000 images. A treshold of zero means all weights are non-zero (binary), between 0.5 and 0.9, the percentage of weights at zero varies linearly between 0.15 and 0.4, at 2.0 all weights are zero.

7.2.5 Changing Loop Order

After the result from the previous section, we were brought to analyze other loop orders, since sparsity did not have a big impact on the performance of our implementation. The new computation loop order tested is (from outer-loop to inner-loop):

1. **kernel loop:** we keep the kernel loop as outer-most loop.

2. **spatial loop:** moving the spatial loop to the outside enables a faster kernel for more layers. while 32x32 is the maximum size of images on CIFAR-10, the number of input and output feature maps grows, especially for other datasets.

3. **output loop:** in this experiment, we always fully unroll this loop. It is possible to do this due to the efficient handling of the input loop.

4. **input loop:** also fully unrolled, we limit the tiling of this loop to 4, to enable the use of parallel registers in memory. This enables unrestricted memory-access and therefore better performance.

We perform some of the tiling and unrolling experiments as we did in the previous sections. Figures 7.14 and 7.15 show the results of these tests. From Figure 7.14a we see that the impact of the size of the tile on Tr and Tc is quite substantial. Figure 7.14b shows the impact of the tiling and unrolling of the output loop. While the output loop is completely unrolled, the tile size of the output loop has a smaller impact than the tile size of the spatial loop.

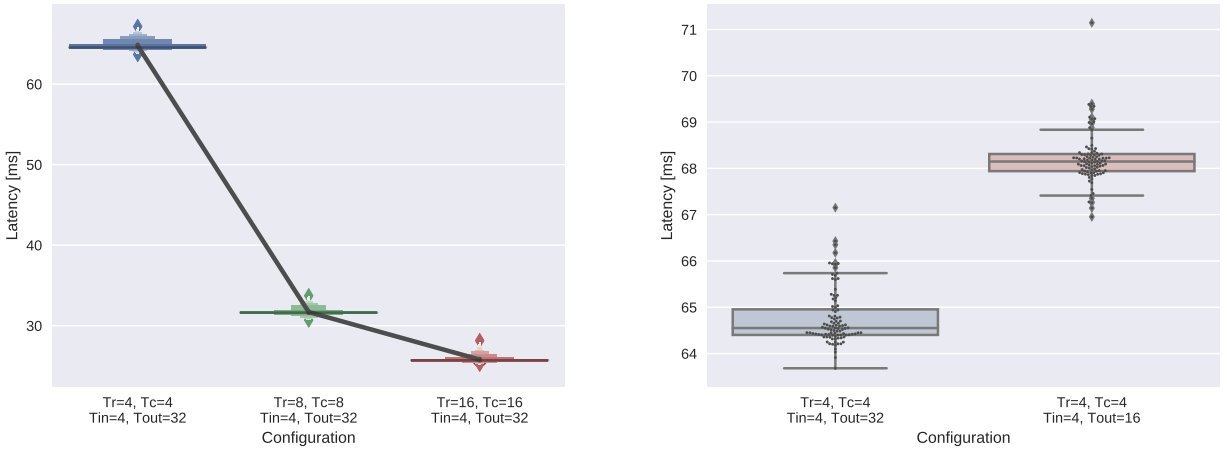


Figure 7.14: New loop order. (a) shows the variation of the tiling of the spatial loop. (b) shows the variation of the output loop.

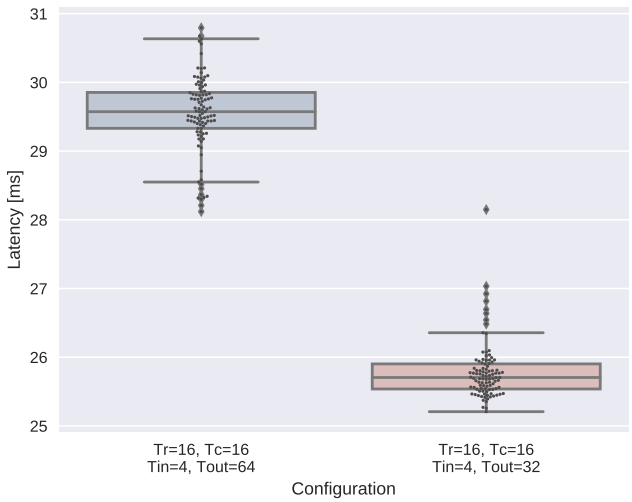
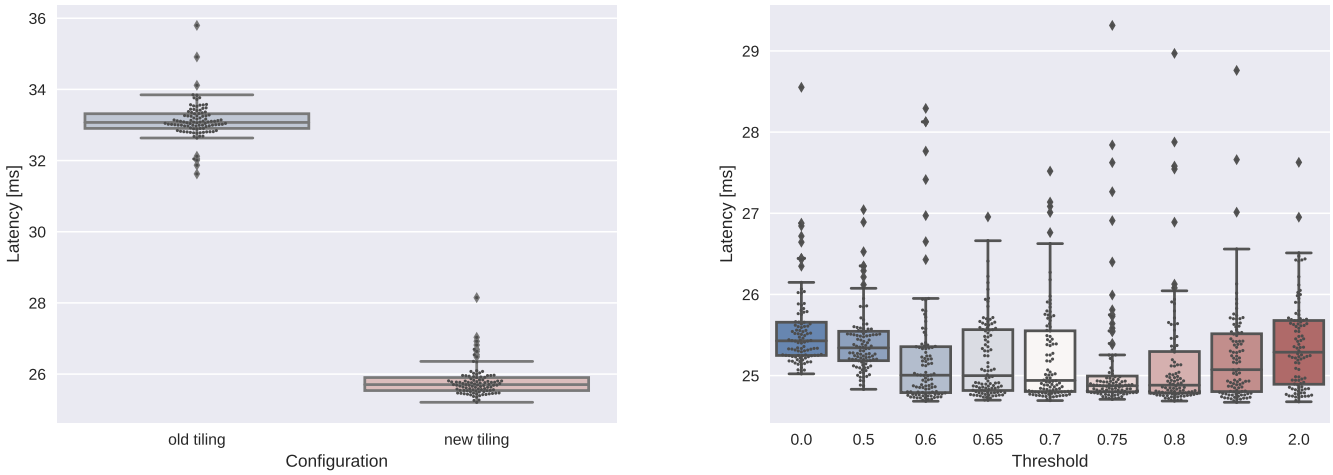


Figure 7.15: Impact of tiling with constant unroll factor. For both tiling configurations, the unroll factor is 32

Figure 7.15 shows the impact of tiling the output loop with fixed unroll factor. This experiment shows that higher tiling of the output loop does not give better performance. This result was already shown in Figure 7.10b, and only confirms the previous observations.

In order to make sure the results of the sparsity found in the previous section are indeed due to the sparsity, and not inherent to the network, we repeat the test with the new loop order. We expect to see no relationship between sparsity and execution time. Figure 7.16b shows that our expectations are met, sparsity has no impact on latency. This result shows that our previous implementation was impacted by sparsity, albeit hardly noticeably.

The optimal value gathered from the tests is a tiling of $Tr = Tc = 16$ for the spatial loop, $Tout = 32$ with a fully unrolled output loop, and $Tin = 4$ with a fully unrolled input loop. With these values, we obtain an average performance of 26ms per image or 38 FPS. Figure 7.16a shows the comparison of looping first over the spatial loop



(a) (b)

Figure 7.16: Comparing the new implementation to the old one. (a) shows the best performing configurations for both designs. (b) shows the latency in relationship to the sparsity.

compared to looping first over the input loop. This figure shows quite clearly that the new loop configuration is better for our design than the previous one.

We took this final design and iterated once more on it. The point was to try and reduce logic utilization in order to unroll more loops and enable more parallel computations. This last design therefore tries to get rid entirely of partial sum storage in local memory since this was the bottleneck in our previous implementation. To achieve this we looked at Ma et al.'s [39] design space exploration.

We kept the same data-path as before for the weights and inputs, with the slight change that we would now load all the channels in one tile. To enable a better hardware generation we load the channels in chunks of 16 ($= Pin$) and store the outputs with the same chunk size too ($= Pout$). Due to local memory limitations we had to reduce Tr and Tc to 8. Contrary to Ma et al., we decided to unroll computation on the input channels instead of rows and columns. This could enable our design to also be used to accelerate fully connected layer efficiently.

Figure 7.17 represents the generated PE structure. Each PE is not a multiplier but a selection between either the input value, the negative input value or zero. We can see that each fmap element is shared over $Pout$ PEs and that at every iteration $Pout \times Pin$ weights need to be loaded from the local memory buffer. Pin elements are then summed through an adder-tree and stored in a register that will accumulate the sum for one output.

This is the first design we achieved where the total number of parallel computations ($PinPoutPrPc = 16 \cdot 16 \cdot 1 \cdot 1 = 256$) exceeds 2 times the number of DSPs. Meaning we finally were able to take advantage of the DSP-less ternary computation. This is also the first time all our loops are pipelined. Chunk-loading is the biggest reason behind this optimization as well as the rewriting of some loop stopping conditions. For example, we now will compute at least Tr×Tc output pixels[1] per tile. This isn't a problem since we're only working with images that have a multiple of 8 in column and row size.

[1]When there are S strides it becomes $(Tr \times Tc)/S$ and FC layers would skip the Tr×Tc computations

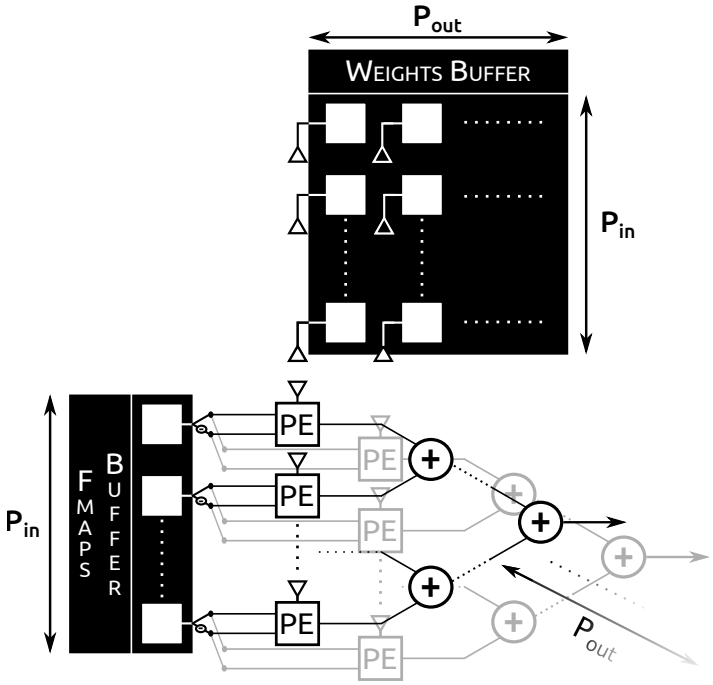


Figure 7.17: PE structure of our final design

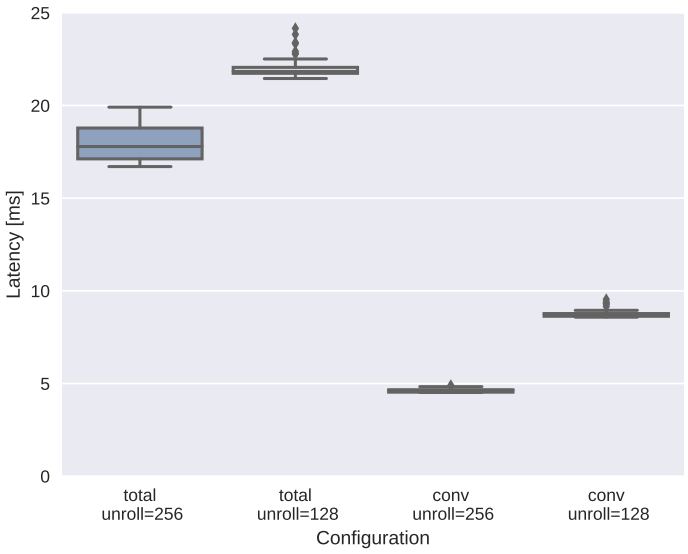


Figure 7.18: Comparing our final implementation with two different unroll factors. Displayed is both the total latency per image and the latency of only the convolutions.

After reducing the latency to 17.5ms, we had a new problem. The time spent starting up the new kernels, by sending the new parameters for each layer, took more time than the actual convolution. Figure 7.18 clearly shows that when comparing the total time of two completely equal implementations, with the first one unrolled twice as much as the second one, we don't reduce the latency by two. However, when we compare the latency of the convolution kernel only, with the profiling information from OpenCL, we do see a reduction by two. Furthermore, timing the `EnqueueTask` operations for the kernels, we see that we lose a lot of time for the enqueueing operation.

Trying to unroll *Pin* and *Pout* further didn't make that much sense, since 7 convolutions of the ResNet-20 have an input width smaller than or equal to 16. We wouldn't see the 2× increase we saw by going from 128 to 256. However since the FPGA still had some resources left, we set *Pin* to 32 (setting *Pout* to 32 would have been faster overall but required too many LEs). We saw no significant change in the total latency but the convolution latency dropped down do 3.7ms.

Since we now had an implementation where 30% of the layers were only able to use 50% of the PEs. We decided to also test a ResNet-14x2. It had the advantage that it only suffered an accuracy loss due to SBN, since the widening compensated the loss caused by quantization. This network also has the advantage of being less deep, thus needing less kernel enqueuing calls and being able to use 100% of the PEs on all layers[2]. We did see a reduction in total time down to 16.9ms, but the convolution time increased to 6.35ms. This is not surprising since the network has nearly 4 times as many MACs.

7.3 Results and Discussion

The final design has an utilization of the resources as shown in table 7.2, and in particular a logic utilization of 91%. While this utilization is almost optimal, the design only makes use of 31% of the DSPs. This is due to the use of ternary networks, which don't require fixed-point multiplications.

Logic Utilization	Registers	DSP	RAM	Frequency
29180	60209	27	397	119.61MHz
91%	47%	31%	100%	–

Table 7.2: System utilization on FPGA

	Kernel	**Spatial**	**Input**	**Output**	**Total**
Loop tiling	$Tk = Nk$ $Tl = Nl$	$Tr = 8$ $Tc = 8$	$Tin = Nin$	$Tout = 16$	589824×K[†]
Loop unrolling	$Pk = 1$ $Pl = 1$	$Pr = 1$ $Pc = 1$	$Pin = 32$	$Pout = 16$	512

[†] *The maximal #computations done for one tile w/ K the width of the network*

Table 7.3: Summary of final tiling and unroll parameters

The latency achieved with our design is 17ms per image. One major bottleneck of this design is the number of FPGA kernels we used and had to activate for every convolution. According to the Xilinx optimization guide [58], enqueueing a kernel can take between 30µs and 60µs. In our tests, the time to enqueue a kernel was sometimes even higher than this upper bound. The bottleneck is more apparent for small convolutions, where the time spent enqueuing is greater than the time spent computing the convolution. Because we have to enqueue 5 kernels for every convolution

[2]The first convolution not included since the input images only have 3 channels, but this is also the case for any other network

(a total of 105 enqueues for ResNet-20 and 75 for ResNet-14×2) which means that for every image, the minimal latency is 6ms (resp. 4.5). In our experiments, we believe the latency to be closer to 12ms (resp. 9).

When profiling the time spent in the convolutional kernel we obtain a latency of 3.7ms. This value seems a good lower bound on what our design could achieve, if the control was not achieved through the host, but rather in the FPGA itself.

It would therefore be beneficial to remove the overhead of the CPU by adding a kernel inside the FPGA which would take care of inter-layer communication through channels. One problem with this improvement is the additional area usage of an added kernel and managing parameter changes between different layers.

A more lightweight approach would be to move to a 3-way buffer of constant size. This way the kernels' memory spaces are given at set-up time and only one kernel enqueuing call will need to be executed. This call would go to a kernel that has communication channels with all other 5 to send them the necessary layer parameters.

	CPU			GPU		FPGA [49]	This Work			
Hardware	i7-4750HQ			Nvidia 960M		Zynq Z-7020	Cyclone V - SE-A5			
Bit-width (w/a)	fl-32			fl-32		b/b*	t/4			
Frequency (MHz)	2800			1176		143	∼120			
kLUT	N/A			N/A		46.9	34.5			
Network	ResNet-20			ResNet-20		ConvNet	ResNet-20		ResNet-14x2	
Acc. Drop	0%			0%		∼ 3%	∼ 2.5%		< 1%	
Batch-size	1	32	1†	1	128	1	1		1	
Latency (ms)	12.7	4.37	7.3†	5.3	0.47	5.94	17.5	3.70‡	16.9	6.35‡
Throughput (GOPS)+	6.4	18.7	11.2†	15.4	173.7	207	4.7	22.1‡	12.5	33.3‡

* *First layer in floating-point.*

† *Using our OpenCL implementation for CPU.*

‡ *Latency and therefore throughput computation for convolutions only.*

\+ *Computed from the latency and #MAC operations in the network*

Table 7.4: Comparison with previous FPGA design and with other platforms

To show the performance of our implementation, we compare it to standard floating-point implementations on a laptop CPU and GPU. In addition to this, we also compare our solution to the binary implementation of [49] on FPGA. On CPU, we compare both the tensorflow implementation and our OpenCL implementation compiled for CPU. On CPU and GPU, we compute results for a batch-size of 1 (which is similar to our case) and a batch-size that was optimized for each device.

The results of the comparison can be found in Table 7.4. We first compare the implementation of tensorflow with the OpenCL implementation on CPU. The OpenCL implementation is almost twice as fast for a batch-size of 1. This is probably due to the fact that tensorflow has optimized the use of a bigger batch-size, and also the use of GPU over CPU.

Our implementation first appears to offer no improvement on existing solutions, but by omitting the bottleneck and only taking the time of the convolutions into account (the dense, avg-pooling and add layers together have a latency smaller than 1ms), we are competitive with the CPU implementation for a batch-size of 1, with a speedup of 1.5×-2.7×. When comparing latency, our implementation is more or less equivalent to the GPU and [49]'s implementations, even providing a modest speedup. However,

due to the use of a binary network, the throughput of [49] is much higher than ours. This means that with a smaller network, their implementation would be much faster, however the accuracy drop would be higher too. Our implementation tries to strike a good balance between accuracy and latency.

We finally compare our implementation for different ResNet architectures. In the previous chapter, we chose a ResNet-20 according to an analysis of accuracy versus complexity. We can see from the last column from Table 7.4, that with the current overhead, it might be advantageous to adopt wider networks with fewer convolutional layers, since the accuracy gained is substantial, with the total latency even slightly dropping. The throughput of this wider network is another indication that the overhead of our solution needs further examination.

While comparing to other implementations is valuable, being able to compare our design to a lower bound might give indications about what is achievable for our design and what is not. Using the number of MACs and the frequency, in addition to the total unroll factor (in Table 7.3), we can compute the minimal latency achievable if not bound by memory transfers. The minimal latency is 1.7ms for the convolutions of ResNet-14×2, supposing 512 MACs / clock cycle. This is quite removed from the 6.35ms computation measured, the nearly 4× increase is quite removed from the ∼1.5× one we typically see for convolutions in related works (see Section 5.2). We think it could be caused by our very frequent accesses to local memory which could have effectively tripled the cost of a MAC. We ruled out global memory bottlenecks due to the latency decreasing by a factor of 2 between an unroll of 128 and 256. This is a big drawback of using OpenCL as discovering where such a bottleneck originates can become very tedious.

Getting to a baseline implementation in OpenCL is quite fast, even with no initial knowledge. It is however much harder to optimize a kernel to its full potential, even when following all best practices published by Intel FPGA [42]. A potential approach to using OpenCL would be to write the critical part of the convolution using an RTL language.

Chapter 8

Conclusion

Accelerating convolutional neural networks has been a critical problem in recent years. The advent of deeper and wider networks means that not only training will take a long time, inference will also be slowed down.

In this thesis, we analyzed different solutions which can enable acceleration on FPGA. We started with all the necessary background knowledge required in Part I. In Part II we reviewed the state of the art in CNN compression and different approaches to FPGA acceleration.

In Part III we expanded upon two complementary approaches to accelerating convolutional neural networks. The first approach consisted of building performant models from the ground up. Quantization, uniform building blocks, and using approximate layers for performance all helped in creating better models for acceleration, without losing much accuracy. The second approach dealt with accelerating the convolution operation in hardware, which has proven to be the main bottleneck of neural networks with convolutional layers.

We took advantage of very low bit-width weights and low bit-width activations to generate models which were less complex. To compare the complexity of several quantization approaches, we developed a formula to compute approximate model complexity, taking into account both global memory transfers and MACs. Looking at our results on the CIFAR-10 dataset, we observed a strong showing by ternary weighted networks which presented themselves as a lightweight alternative to floating point weights. 4-bit and 2-bit activations on top, only reduced the activation slightly. This accuracy loss can still be regained by widening the network 2 times. Reaching an estimated complexity which is still lower than that of a full floating point network.

This led us to create a ternary-weight, 4-bit activation network while losing less than 2% accuracy on CIFAR-10. We modified the batch-normalization layer to only use shifting and add operations. This resulted in a less than 1% accuracy drop compared to the quantized version by fine-tuning the network to use the shift-based batch normalization. Furthermore we moved from specific layers at specific places to one building-block layer which contains a convolution, an SBN and an activation function. Doing this led to no further loss of accuracy. Finally we also studied a double widened but shallower version of this network. Doing so eliminated the accuracy loss induced by the ternary and 4-bit quantization at the cost of a higher model and computational complexity. Although we showed that it only slightly increases latency, due to overhead and a slightly better PE utilization.

We turned to OpenCL for our FPGA implementation, and analyzed an architecture based on the acceleration of the convolutional layer. With tiling, unrolling and a judicious choice of loop ordering, we achieved a performant network. We found that a layout that tried to make use of the inherent sparsity in our network had almost no impact on latency, especially in the case of a residual network, which has often less weights than a similar-sized VGG network. In contrast, the impact of unrolling of the inner-most loops was found to be dramatic, no matter which loop. Due to this, we prioritized the fitting of a high unroll-coefficient design.

Finding the right tiling parameters required some design space exploration. For most of our initial designs, we noted that a high tiling factor for the *output* feature maps was more important than for the *input* feature maps. We therefore tried to use an output tiling that matched the dimensions of the output. We extended this reasoning to the tiling on the rows and columns, but were limited by the storage of the partial sums. Because of this constraint we implemented a final design which got rid of partial sum storage. With the unrolling factor being on the output and input feature maps, this design is also adequate for fully connected layer acceleration.

Our final design, which allows for stream-like classification, managed a latency of 17ms. This is achieved by having 512 parallel computations during the convolution. By using more than double the total amount of DSPs, we propose a software-hardware co-design that could fully take advantage of the removal of DSP computations like fixed point multiplication. Moreover, this is achieved without a significant loss in accuracy on the CIFAR-10 classification task.

Further Work

Although our quantization results are quite promising, the accuracy hit incurred by using quantized activations remains relatively high at 1.5-2% on the CIFAR-10 dataset. Researching more adaptive activation ranges or methods seems necessary.

Verifying our findings on larger networks (for larger datasets) is the next logical step. More closely studying the impact of kernel regularization on quantized networks seems important since 1 or 2 % gains could close the gap with full floating point networks.

Quantization is only one aspect of building a hardware-aware convolutional neural network. Further work can be done to improve every part of a network such that it will be faster and easier to port to an FPGA.

While our analysis of quantization has been quite thorough, a lot of work still needs to be done to accelerate inference on FPGA. Our initial design tried to take advantage of sparsity, but did not achieve it in any meaningful way. Using the same datapath, but moving the order of the loop, added a consequential performance boost. In further work, it might be interesting to analyze other loop orders, or try to optimize the last design presented in this work. Moving away from a fixed the tile design seems also necessary for bigger image sizes. A sliding window seems like a good alternative.

In addition to optimizing the convolutional layer, we recommend modifying the overall architecture of the implementation. Our hardware-software co-design architecture shows that the time spent on CPU is quite large compared to the time spent on computation on the FPGA. Transferring more logic to the FPGA might make the network faster without needing to optimize other parts of the network.

While in our case, the fully-connected layer did not need any optimization, due to its size, this is something that is specific to residual networks with small final layers. For a generic acceleration platform, it might be important to accelerate other layers than the convolutional layer. Our architecture should allow the acceleration of fully-connected layers by design, through the unroll of the input and output loops.

Bibliography

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[2] NVIDIA Corporation, "NVIDIA OpenCL Best Practices Guide," 2009.

[3] E. Nurvitadhi, S. Subhaschandra, G. Boudoukh, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, and D. Moss, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*. New York, New York, USA: ACM Press, 2017, pp. 5–14. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3020078.3021740

[4] Khronos OpenCL Working Group, "The OpenCL Specification," pp. 1–385, 2009.

[5] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in ...," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958. [Online]. Available: http://psycnet.apa.org/journals/rev/65/6/386.pdf{%}5Cnpapers://c53d1644-cd41-40df-912d-ee195b4a4c2b/Paper/p15420

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2323, 1998.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances In Neural Information Processing Systems*, pp. 1–9, 2012.

[8] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[9] X. Glorot and B. Yoshua, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, mar 2010, pp. 249–256. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *Proceedings of the IEEE international conference on computer vision*, no. 1, feb 2015, pp. 1026–1034. [Online]. Available: http://arxiv.org/abs/1502.01852

[11] S. Ruder, “An overview of gradient descent optimization algorithms,” pp. 1–14, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[12] R. Ramachandran, D. C. Rajeev, S. G. Krishnan, and P. Subathra, “Deep learning – An overview,” *International Journal of Applied Engineering Research*, vol. 10, no. 10, pp. 25 433–25 448, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.neunet.2014.09.003

[13] C. Lemaréchal, “Cauchy and the Gradient Method,” *Documenta Mathematica*, vol. ISMP, pp. 251–254, 2012. [Online]. Available: https://www.math.uni-bielefeld.de/documenta/vol-ismp/40_lemarechal-claude.pdf

[14] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *AIP Conference Proceedings*, vol. 1631, pp. 58–62, dec 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[15] A. Karpathy, “Stanford University CS231n: Convolutional Neural Networks for Visual Recognition,” 2016. [Online]. Available: http://cs231n.stanford.edu/syllabus.html

[16] H. Zou and T. Hastie, “Regression and variable selection via the elastic net,” *J R Stat Soc: Ser B*, vol. 67, no. 1, pp. 1–29, 2005. [Online]. Available: http://dx.doi.org/10.1111/j.1467-9868.2005.00503.x

[17] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” *CoRR*, 2014.

[18] Y.-H. Chen, S. Member, T. Krishna, J. S. Emer, V. Sze, and S. Member, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks,” *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, vol. 1, 2016.

[19] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” pp. 1–14, 2015. [Online]. Available: http://arxiv.org/abs/1510.00149

[20] M. Courbariaux, Y. Bengio, and J.-P. David, “BinaryConnect: Training Deep Neural Networks with binary weights during propagations,” pp. 1–9, 2015. [Online]. Available: http://arxiv.org/abs/1511.00363

[21] J. Qiu, S. Song, Y. Wang, H. Yang, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, and N. Xu, “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA ’16*. New York, New York, USA: ACM Press, 2016, pp. 26–35. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2847263.2847265

[22] S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, S. Zheng, T. Lu, J. Gu, L. Liu, and S. Wei, “A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications,” *IEEE Journal of Solid-State Circuits*, vol. 53, no. 4, pp. 968–982, apr 2018. [Online]. Available: http://ieeexplore.ieee.org/document/8207783/

[23] B. Moons, K. Goetschalckx, N. Van Berckelaer, and M. Verhelst, "Minimum energy quantized neural networks," *Conference Record of 51st Asilomar Conference on Signals, Systems and Computers, ACSSC 2017*, vol. 2017-Octob, pp. 1921–1925, 2018.

[24] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," pp. 1–17, 2016. [Online]. Available: http://arxiv.org/abs/1603.05279

[25] J. Rudolph, "OpenCL Work Item Ids: Global/Group/Local," 2012. [Online]. Available: https://jorudolph.wordpress.com/2012/02/03/opencl-work-item-ids-globalgrouplocal/

[26] Intel FPGA Group, "Intel ® FPGA SDK for OpenCL ™ Pro Edition: Programming guide," 2018.

[27] H. Kung, "Why Systolic Architectures?" 1982.

[28] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A Survey of FPGA Based Neural Network Accelerator," *arXiv preprint*, vol. 9, no. 4, pp. 1–22, 2017. [Online]. Available: http://arxiv.org/abs/1712.08934

[29] K. Abdelouahab, M. Pelcat, J. Serot, F. Berry, K. Abdelouahab, M. Pelcat, J. Serot, F. Berry, and A. Cnn, "Accelerating CNN inference on FPGAs : A Survey," 2018.

[30] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," *arXiv*, 2016.

[31] F. Li, B. Zhang, and B. Liu, "Ternary Weight Networks," no. Nips, 2016. [Online]. Available: http://arxiv.org/abs/1605.04711

[32] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," pp. 1–29, 2016. [Online]. Available: http://arxiv.org/abs/1609.07061

[33] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, "GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework," *Neural Networks*, vol. 100, pp. 49–58, 2018.

[34] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," vol. 1, no. 1, pp. 1–13, 2016. [Online]. Available: http://arxiv.org/abs/1606.06160

[35] T. J. Yang, Y. H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 6071–6079, 2017.

[36] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," pp. 1–13, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

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

[38] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. 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*. New York, New York, USA: ACM Press, 2015, pp. 161–170. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2684746.2689060

[39] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–14, 2018. [Online]. Available: http://ieeexplore.ieee.org/document/8330049/

[40] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC," *Proceedings of the 2016 International Conference on Field-Programmable Technology, FPT 2016*, pp. 77–84, 2017.

[41] P. Colangelo, N. Nasiri, E. Nurvitadhi, A. Mishra, M. Margala, and K. Nealis, "Exploration of Low Numeric Precision Deep Learning Inference Using Intel® FPGAs," *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '18*, pp. 294–294, 2018. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3174243.3174999

[42] Intel FPGA Group, "Intel ® FPGA SDK for OpenCL ™ Pro Edition: Best practices guide," 2018.

[43] D. Wang, J. An, and K. Xu, "PipeCNN: An OpenCL-Based FPGA Accelerator for Large-Scale Convolution Neuron Networks," 2016. [Online]. Available: http://arxiv.org/abs/1611.02450

[44] J. Zhang and J. Li, "Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network," *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pp. 25–34, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3020078.3021698

[45] H. Zeng, R. Chen, C. Zhang, and V. Prasanna, "A Framework for Generating High Throughput CNN Implementations on FPGAs," *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '18*, pp. 117–126, 2018. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3174243.3174265

[46] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL(TM) Deep Learning Accelerator on Arria 10," 2017. [Online]. Available: http://arxiv.org/abs/1701.03534

[47] R. Zhao, H.-c. Ng, W. Luk, and X. Niu, "Towards Efficient Convolutional Neural Network for Domain-Specific Applications on FPGA," 2018. [Online]. Available: http://arxiv.org/abs/1809.03318

[48] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "FP-BNN: Binarized neural network on FPGA," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018. [Online]. Available: https://doi.org/10.1016/j.neucom.2017.09.046

[49] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs," *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pp. 15–24, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3020078.3021741

[50] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," *35th International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2966986.2967011

[51] F. Chollet *et al.*, "Keras," 2015. [Online]. Available: https://keras.io

[52] N. S. Keskar and R. Socher, "Improving Generalization Performance by Switching from Adam to SGD," no. 1, 2017. [Online]. Available: http://arxiv.org/abs/1712.07628

[53] A. Krizhevsky, "(CIFAR10) Learning Multiple Layers of Features from Tiny Images," *... Science Department, University of Toronto, Tech. ...*, pp. 1–60, 2009.

[54] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[55] P. Chrabaszcz, I. Loshchilov, and F. Hutter, "A Downsampled Variant of ImageNet as an Alternative to the CIFAR datasets," pp. 1–9, 2017. [Online]. Available: http://arxiv.org/abs/1707.08819

[56] S. Zagoruyko and N. Komodakis, "Wide Residual Networks," 2016. [Online]. Available: http://arxiv.org/abs/1605.07146

[57] W. Tank, "IntelFPGA Cyclone V SoC OpenCL," 2018. [Online]. Available: https://github.com/thinkoco/c5soc_opencl

[58] Xilinx Inc., "SDAccel Environment Profiling and Optimization Guide," vol. UG1207, pp. 1–99, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug1207-sdaccel-optimization-guide.pdf

Appendix A

Adaptive Ternary Weights

The following is the development of the cut-off computation for ternary weighted networks of Li et al. [31]. They start by seeking to minimize the Euclidian distance between the full precision ($\mathbf{W}$) and ternary ($\mathbf{W}^t$) weights both of size n. Then define $\alpha > 0$ as a scaling factor. The optimization problem is formulated as:

$$\begin{cases} \alpha^*, \mathbf{W}^{t*} = & \arg\min_{\alpha, \mathbf{W}^t} J(\alpha, \mathbf{W}^t) = ||\mathbf{W} - \alpha\mathbf{W}^t||_2^2 \\ \text{s.t.} & \alpha \geq 0, W_i^t \in \{-1, 0, 1\}, i = 1, 2, .., n. \end{cases} \tag{A.1}$$

Suppose $\mathbf{X}$ is the input to the convolutional layer and g is the activation function, then the value after convolution ($\mathbf{X}^{\text{next}}$)

$$\begin{cases} \mathbf{Z} & = \mathbf{X} * \mathbf{W} \approx \mathbf{X} * (\alpha\mathbf{W}^t) = (\alpha\mathbf{X}) * \mathbf{W}^t \\ \mathbf{X}^{\text{next}} & = g(\mathbf{Z}) \end{cases} \tag{A.2}$$

It should be noted that $(\alpha\mathbf{X}) * \mathbf{W}^t$ is a multiplication-less convolution, since sign inversion can always be done by either bit change (float) or bit inversion combined with a $+1$ addition (2's complement).

There is no way to solve the $J(\alpha, \mathbf{W}^t)$ in a deterministic fashion: deriving the expression w.r.t. α and $\mathbf{W}^t$ would leave us with interdependant values. To overcome this, they approximated the solution through a threshold-based ternary function:

$$\mathbf{W}_i^t = f_t(\mathbf{W}_i|\Delta) = \begin{cases} +1 & \text{if} \quad \mathbf{W}_i > \Delta \\ 0 & \text{if} \quad |\mathbf{W}_i| \leq \Delta \\ -1 & \text{if} \quad \mathbf{W}_i < -\Delta \end{cases} \tag{A.3}$$

With $\Delta > 0$. The original problem (A.1) can be rewritten with (A.3) as:

$$\alpha^*, \Delta* = \underset{\alpha \geq 0, \Delta \geq 0}{\arg\min} \left(|\mathbf{I}_\Delta|\alpha^2 - 2\left(\sum_{i \in \mathbf{I}_\Delta} |W_i| \right) \alpha + c_\Delta \right) \tag{A.4}$$

with $\mathbf{I}_\Delta = \{i \text{ if } |W_i| > \Delta\}$ thus $|\mathbf{I}_\Delta| = \#\text{elements in } \mathbf{I}_\Delta$ and $c_\Delta = \sum_{i \in \mathbf{I}_\Delta^c} W_i^2$. Deriving equation (A.4) for α leads to the following solution:

$$\alpha_\Delta^* = \frac{1}{|\mathbf{I}_\Delta|} \sum_{i \in \mathbf{I}_\Delta} |W_i| \tag{A.5}$$

Substituting (A.5) in (A.4) gives us:

$$\Delta^* = \underset{\Delta > 0}{\arg\max} \frac{1}{|\mathbf{I}_\Delta|} \left(\sum_{i \in \mathbf{I}_\Delta^c} W_i \right)^2 \tag{A.6}$$

However (A.6) has no straightforward solution. To solve this they suppose that all weights are generated from a uniform or normal distribution. If we suppose they were uniformly distributed in $[-a, a]$ and $\Delta \in]0, a]$, $\Delta^* = \frac{1}{3}a = \frac{2}{3} \cdot E(|\mathbf{W}|)$. In the case of a normal distribution $\mathbb{N}(0, \sigma^2)$, then $\Delta = 0.6\sigma = 0.75 \cdot E(|\mathbf{W}|)$. Averaging both we can approximate Δ^* through fast computation with:

$$\Delta^* \approx 0.7 \cdot E(|\mathbf{W}|) \approx 0.7 \cdot \frac{\sum_{i=1}^{n} |W_i|}{n} \tag{A.7}$$

Appendix B

All Results with different complexity modelings

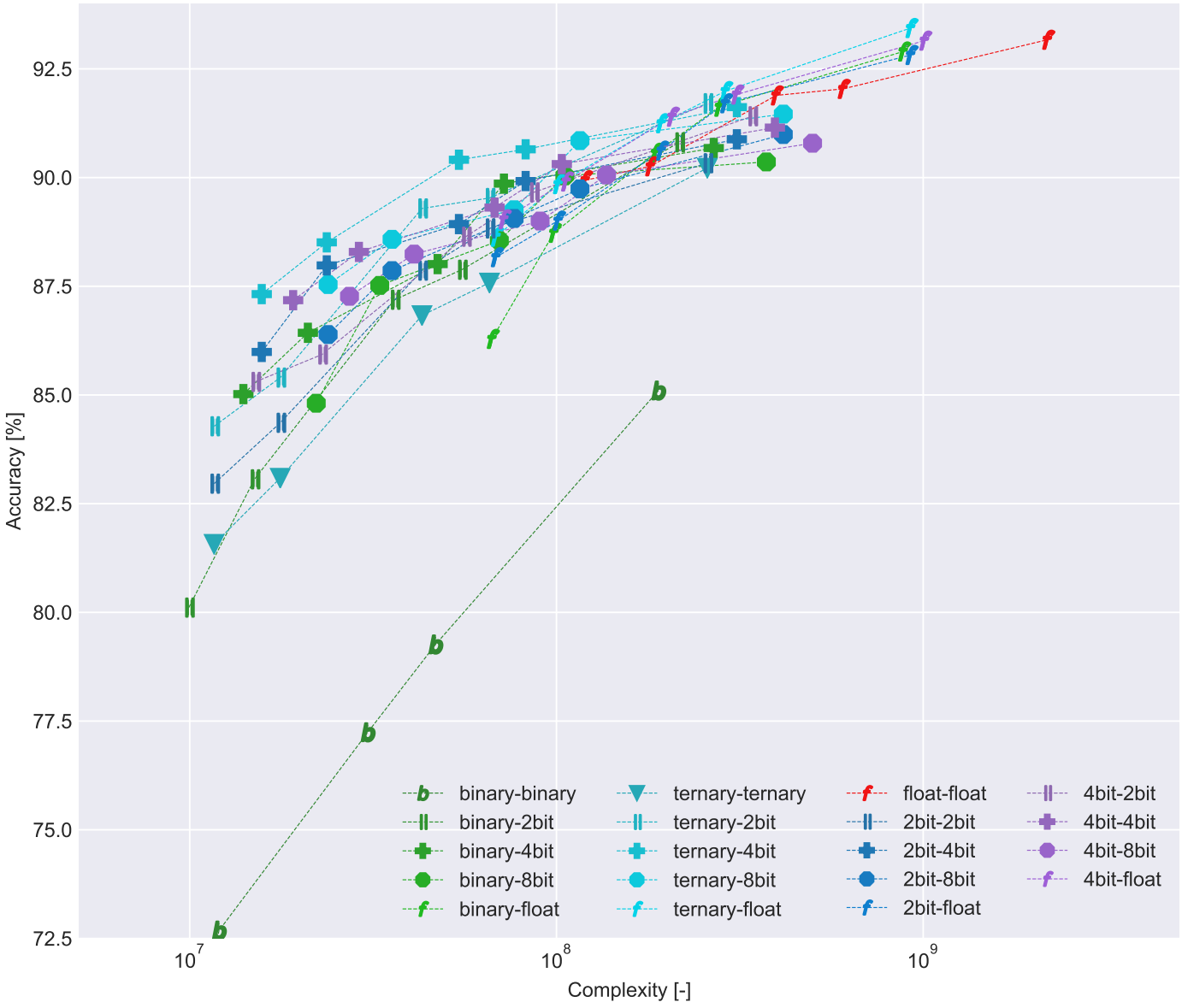


Figure B.1: All values with our custom complexity modeling

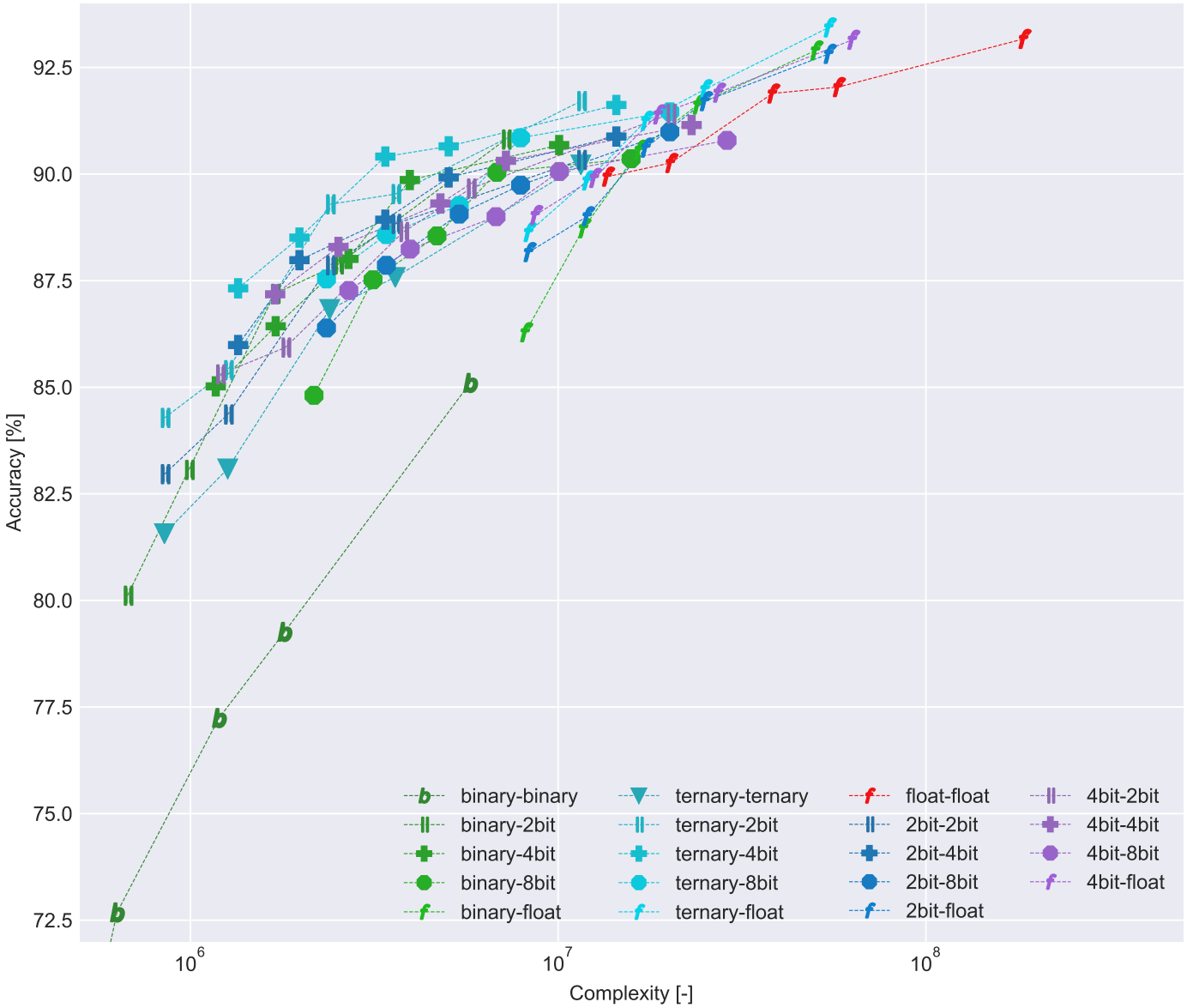


Figure B.2: Model complexity equal to # global memory accesses

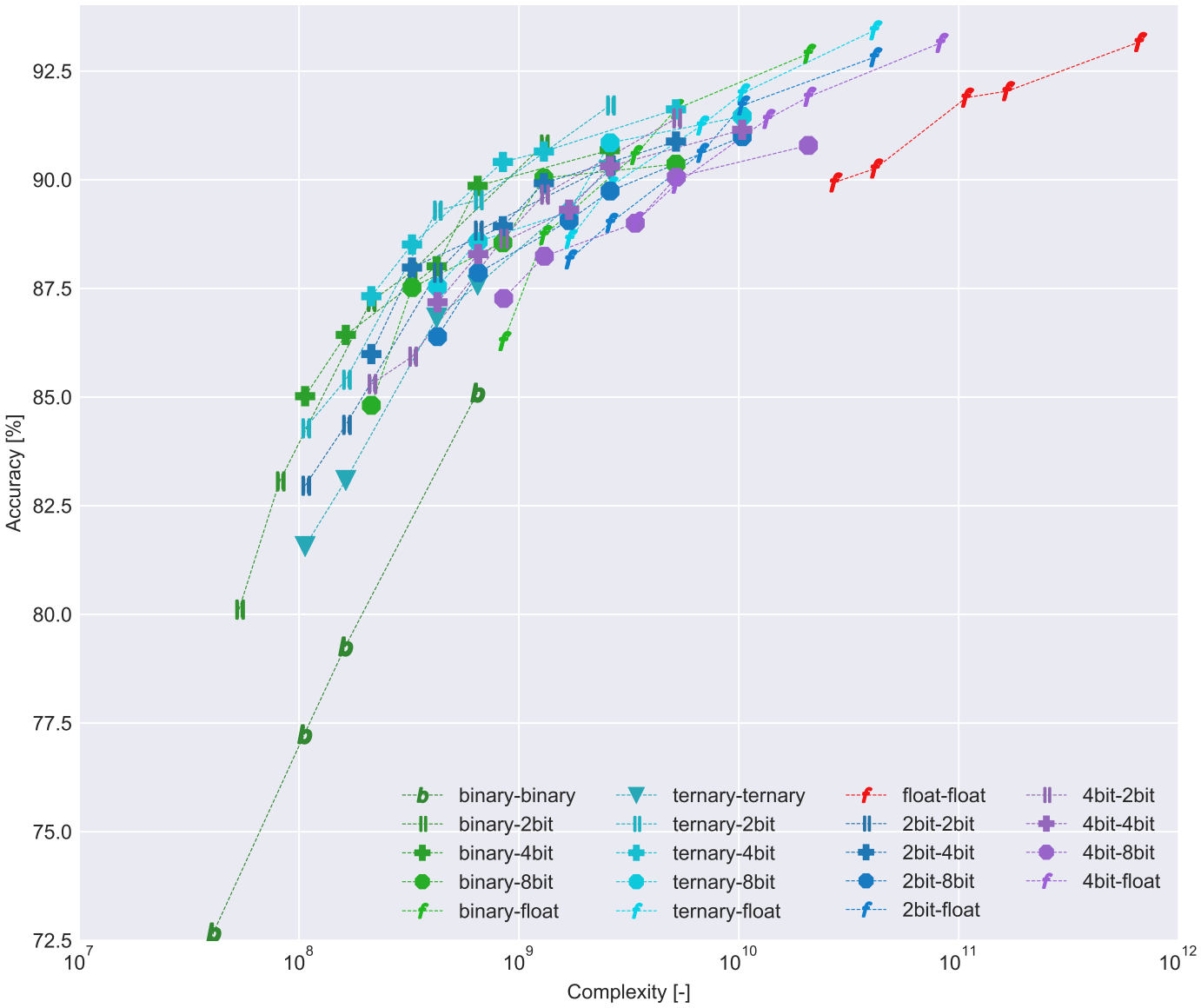
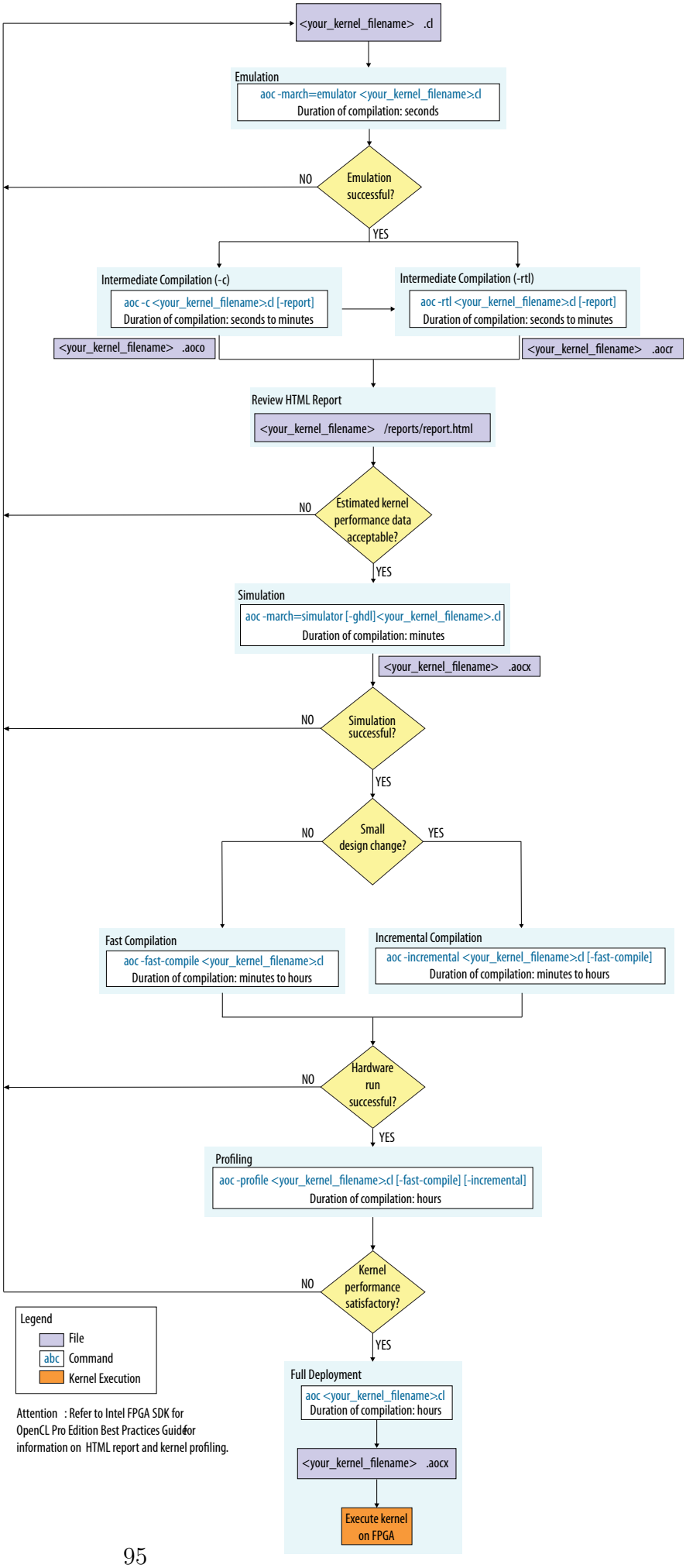


Figure B.3: Model complexity equal to # MAC×# activation bits ×# weight bits

Appendix C

OpenCL Dev. Flow

Figure from [26].



UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**