Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Elettronica
(Electronic Engineering)

Tesi di Laurea Magistrale

Architectural exploration and efficient FPGA implementation of convolutional neural networks

Relatore:
Luciano Lavagno

Candidata:
Rachele Setto

Aprile 2021

To my beloved grandmothers Liliosa and Maria

"Any sufficiently advanced technology is indistinguishable from magic"

Arthur C. Clarke

Abstract

Nowadays image recognition algorithms are used in various fields, which go from simple mobile phone face recognition, to detect object from drones but also to land rovers on Mars.
Among these algorithms, the Convolution Neural Networks (CNN) are the most used one. Even if their construction and structure is very simple and easy to be understood, their computational cost and memory requirements are nowadays challenging, especially when the network has to be inferred on FPGAs, which are the most suitable devices for embedded systems and data-centers, due to the low energy requirements.

In this thesis work an architecturally optimized CNN is considered as starting point for further data precision optimization. This network is called *SkyNet* and is the winner of the *System Design Contest for low power object detection in the 56th IEEE/ACM Design Automation Conference (DAC-SDC)*. Given an image, this network is able to detect objects which are present in there.

In order to optimize this network, a *quantization aware training* QAT technique, which consists in reducing the amount of bits on which the network parameters are stored, is adopted. The goal of quantization aware training is to find the best trade-off among memory saving and accuracy reduction: *Brevitas*, from Xilinx Research Lab, turned out to be a very good library for this purpose. This thesis describes how to use *Brevitas* to quantize networks (by quantizing SkyNet) and how the quantization is implemented in the library.

After the QAT, the model is optimized, synthesized and implemented using the *FINN* compiler which, as *Brevitas*, has been developed by the Xilinx Research Lab. This thesis deeply describes the steps to be followed in FINN to implement the network on a target FPGA, starting from the export of the model from *Brevitas*, then optimizing the model using *Transformations* functions, and finally inferring the network on a target device, using Vivado HLS and Vivado Design Suite. Furthermore, the mains FINN problems encountered during the development of the quantized network are listed and analyzed, giving partial solutions on how to fix them.

In conclusion, a comparison among the initial SkyNet network and its quantized version is reported, highlighting the memory reduction required to store the network parameters.

Contents

List of Figures

List of Tables

Acronyms

BRAM Block Random Access Memory

CNN Convolutional Neural Network

DNN Deep Neural Network

DSP Digital Signal Processing

FM Feature Map

FPGA Field Programmable Gate Array

NCHW N (batch size), Channel, Height, Width

NHWC N (batch size), Height, Width, Channel

PSO Particle Swarm Optimization

QNN Quantized Neural Network

Chapter 1

Introduction to CNNs

Convolutional Neural Networks (CNNs or ConvNet) are a set of neural networks used in the object detection and tracking field.

Given an image as input, the CNN recognizes the elements in the image and classifies them, by giving as output the probability that that image belongs to a particular *class* (as person, bike, cat, dog, car...). Some CNNs, such as *SkyNet* (see Section 2) are able also to detect the position of the detected object in the figure.

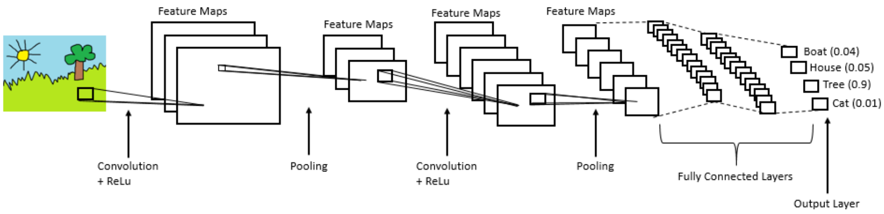


Figure 1.1: A basic CNN schematic.

CNNs are made of different *layers*, each one with a specific function, that are repeated several times, depending on the CNN implementation. An example of schematic of CNN is reported in Figure 1.1.

In order to classify the image, the CNN takes as input the related matrix[1], called *Feature Map* (FM), and makes it flows into these layers, where the FM is *convoluted* and the learnable parameters, called *weights*, are updated.

The most common type of layers are:

- *Convolutional Layer*: in CNN it is the most important one. Given a *filter*, this layer is able to detect particular shapes inside the figure.

- *Batch Normalization Layer*: this optional layer is used in order to allow a better and faster training of the network.

[1] CNN and, more in general, computer see images as matrices of pixels: if the image is coded in RGB, then CNN will decode it as a H×W×3 matrix, where H and W represent height and width respectively, while 3 is the number of channels, where every of them stores the value of the Red, Green, Blue color, which goes from 0 to 255.

- *Activation Layer*: this layer is usually added right after the convolutional layer to add a non-linearity factor in the network.
- *Pooling Layer*: it is usually placed after the activation layer; it is used to reduce the size of its output.
- *Fully Connected Layers*

1.1 Convolutional Layers

Convolutional Layers are used to detect any kind of shapes in an image. In order to do that, the image, represented by a $n_A \times n_A \times 3$ matrix (RGB coding), is *convoluted* with specific filter matrices, also called *kernels*, who store the learnable parameters of the network, i.e. the *weights*.
The kernels are used to detect specific shapes inside an image, such as horizontal and vertical lines and, as the image, they are represented by square matrices, with different weights, depending on which shapes they have to detect. In order to understand what *convolution* is and how it works, an example is here reported.
Considering two matrices (called *tensor* in Pytorch), A for the image and K for the kernel, both with dimensions 3×3 the convolution is given by:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix} = \sum_{i=1}^{3}\sum_{j=1}^{3} a_{i,j}k_{i,j} \tag{1.1}$$

Thus, in convolution each element of one tensor is *dot multiplied* with the corresponding element of the second tensor and then all the values are summed together to obtain the output value [5].

Typically, the kernel tensors are small, 3×3 or 5×5, with respect to the image tensors, that can be big as 1024×1024×3, depending on the image resolution, therefore in order to apply convolution, the filter *slides* over the image matrix.

Considering the case of a gray scale image (i.e. an matrix with just one channel), with tensor $n_A \times n_A \times 1$, with n_A=4 represented as:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \tag{1.2}$$

and the filter tensor on $n_K \times n_K$ with n_K=3

$$K = \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix} \tag{1.3}$$

the convolution is performed by sliding the kernel on the image tensor, from top-left corner to the top-right corner (i.e. the right-end of the matrix), with a step given by a parameter called *stride*, which is the number of pixels shifts over the input matrix (in this example, *stride*=1):

$$FM_{11} = \begin{pmatrix} \mathbf{a_{11}} & \mathbf{a_{12}} & \mathbf{a_{13}} & a_{14} \\ \mathbf{a_{21}} & \mathbf{a_{22}} & \mathbf{a_{23}} & a_{24} \\ \mathbf{a_{31}} & \mathbf{a_{32}} & \mathbf{a_{33}} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \cdot \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix}$$
$$= a_{11} \cdot k_{11} + a_{12} \cdot k_{12} + a_{13} \cdot k_{13} + a_{21} \cdot k_{21} + a_{22} \cdot k_{22} + a_{23} \cdot k_{23} +$$
$$+ a_{31} \cdot k_{31} + a_{32} \cdot k_{32} + a_{33} \cdot k_{33}$$

FM_{11} represents the first element of the feature map. Moving the window to right, the second element of the feature map is computed by:

$$FM_{12} = \begin{pmatrix} a_{11} & \mathbf{a_{12}} & \mathbf{a_{13}} & \mathbf{a_{14}} \\ a_{21} & \mathbf{a_{22}} & \mathbf{a_{23}} & \mathbf{a_{24}} \\ a_{31} & \mathbf{a_{32}} & \mathbf{a_{33}} & \mathbf{a_{34}} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \cdot \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix}$$
$$= a_{12} \cdot k_{11} + a_{13} \cdot k_{12} + a_{14} \cdot k_{13} + a_{22} \cdot k_{21} + a_{23} \cdot k_{22} + a_{24} \cdot k_{23} +$$
$$+ a_{32} \cdot k_{31} + a_{33} \cdot k_{32} + a_{34} \cdot k_{33}$$

Since the kernel window has reached the right-end of the image, it is moved back to the left-end and shifted down with the same step given by the stride parameter. Thus the convolution proceeds by computing the third element of the feature map, FM_{21}.

$$FM_{21} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \mathbf{a_{21}} & \mathbf{a_{22}} & \mathbf{a_{23}} & a_{24} \\ \mathbf{a_{31}} & \mathbf{a_{32}} & \mathbf{a_{33}} & a_{34} \\ \mathbf{a_{41}} & \mathbf{a_{42}} & \mathbf{a_{43}} & a_{44} \end{pmatrix} \cdot \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix}$$
$$= a_{21} \cdot k_{11} + a_{22} \cdot k_{12} + a_{23} \cdot k_{13} + a_{31} \cdot k_{21} + a_{32} \cdot k_{22} + a_{33} \cdot k_{23} +$$
$$+ a_{41} \cdot k_{31} + a_{42} \cdot k_{32} + a_{43} \cdot k_{33}$$

Then, moving the kernel window to right, the last element of the feature map is computed:

$$FM_{22} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & \mathbf{a_{22}} & \mathbf{a_{23}} & \mathbf{a_{24}} \\ a_{31} & \mathbf{a_{32}} & \mathbf{a_{33}} & \mathbf{a_{34}} \\ a_{11} & \mathbf{a_{42}} & \mathbf{a_{43}} & \mathbf{a_{44}} \end{pmatrix} \cdot \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix}$$
$$= a_{22} \cdot k_{11} + a_{23} \cdot k_{12} + a_{24} \cdot k_{13} + a_{32} \cdot k_{21} + a_{33} \cdot k_{22} + a_{34} \cdot k_{23} +$$
$$+ a_{42} \cdot k_{31} + a_{43} \cdot k_{32} + a_{44} \cdot k_{33}$$

Thus, the output tensor, which is also called *feature map*, is obtained:

$$FM = \begin{pmatrix} FM_{11} & FM_{12} \\ FM_{21} & FM_{22} \end{pmatrix} \tag{1.4}$$

Its dimensions are given by:

$$n = \left\lfloor \frac{n_A - n_K}{s} + 1 \right\rfloor \tag{1.5}$$

where s stands for *stride*.
In this example, the image is represented by a square tensor. More in general, when the image tensor dimensions are $n_H \times n_W$, and the kernel tensor dimensions are $n_K \times n_K$, the feature map dimensions $n_h \times n_w$ are given by:

$$n_h = \left\lfloor \frac{n_H - n_K}{s} + 1 \right\rfloor \tag{1.6}$$

$$n_w = \left\lfloor \frac{n_W - n_K}{s} + 1 \right\rfloor \tag{1.7}$$

In Figure 1.2, an example of convolution over an input image is reported.

In this case the filter has been chosen in order to detect the edges and it has been directly assigned by the user to the layer, without any training. As a results the output image highlights the structure of the bridge.

(a) Input image before convolution.

(b) Input image after convolution.

Figure 1.2: Application of convolution on a grayscale image in order to detect edges.

1.2 Normalization Layers

In order to make the gradient descent reaching the global minimum faster, batch normalization is usually applied in CNN. This technique consists in normalizing the input data in order to have a restricted range of values, in such a way that when the training is performed the possibility to overshoot the minimum is reduced.

The *batch normalization* layer is usually placed before the *activation* layer (see section 1.3) and it simply zero-centers and normalizes the inputs, then scales and shifts the result using two new parameters per layer (one for scaling, the other for shifting). This operation allows the model to learn the optimal scale and mean of

the inputs for each layer. [2]
In order to zero-center and normalize the inputs, the algorithm needs to estimate the mean and the standard deviation of the input. It does so by evaluating the mean and standard deviation of the inputs over the current *mini-batch*, i.e. over a number, a *batch*, of images belonging to the image dataset.

1.3 Activation Layers

After convolutional layer, an *activation* layer is usually present. The purpose of activation layer is to introduce a non-linearity factor in the network, in order to make the network learn correctly. The most used activation function is the ReLU:

$$ReLU(x) = max(0, x) \tag{1.8}$$

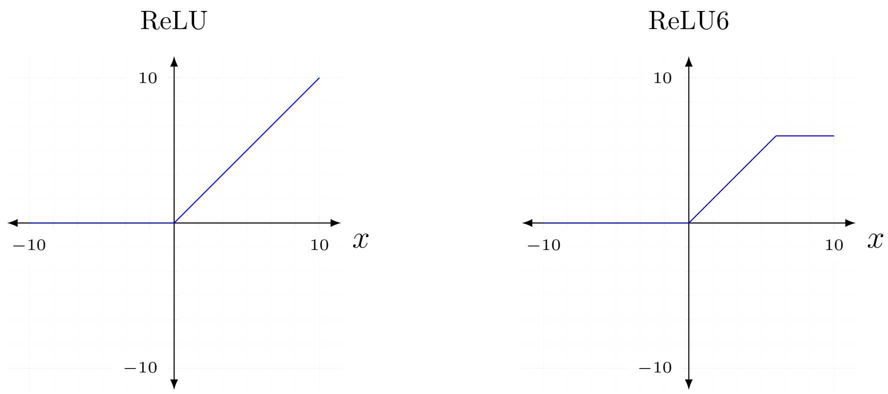


Figure 1.3: On the left, the ReLU(x) graph, on the right the ReLU6(x) graph.

As displayed in the Figure 1.3, the ReLU function simply filers the input values: if the input is negative, the ouput is set to 0, while if it is positive, it is left as it is.
In SkyNet (see Section 2) a particular type of ReLU is adopted, that is the ReLU6: this function simply behaves like the standard ReLU with the exception that in case of positive values the maximum output is set to 6.

1.4 Pooling Layers

When input images are particularly big, the output of convolution layer, namely the feature map, has a consistent $n_H \times n_W$ size. In order to reduce the feature map size, after the convolution layer, a *pooling* layer is usually instantiated.
As in the convolutional layer, the pooling layer is characterized by a kernel, with defined kernel size n_K and stride s, which is made flown over the input feature map. In this case the filter is empty and it is used like a window to highlight a region of the feature map.

Given an highlighted region (the red one in Figure 1.4), the output of the pooling layer depends on the type of pooling is applied. Actually, there are two main types of pooling: the *max pooling*, where the output is given by the maximum value of the window, and the *average pooling*, where the output is given by the average among the values of the window. In Figure 1.4, an example of Max Pooling application is reported.

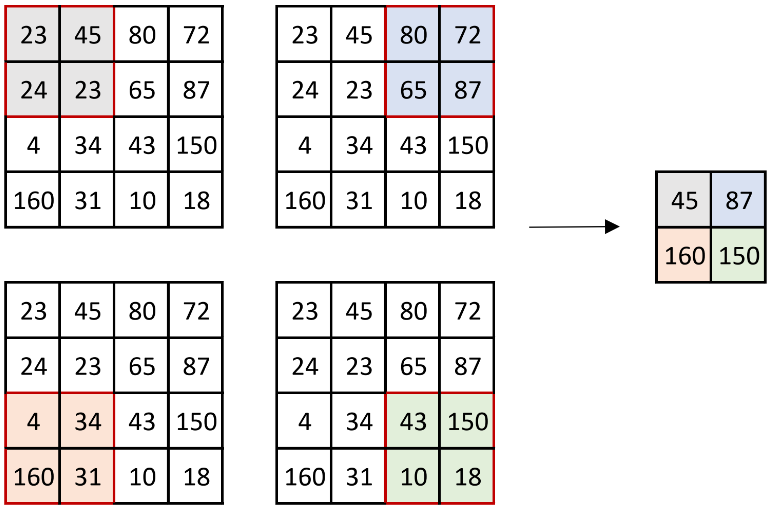


Figure 1.4: Example on Pooling Layer application on a tensor of 4x4

1.5 Fully Connected Layers

Typically, the last layer of the CNNs is a fully connected layer, which first flattens the matrix into a number vectors, as many as the number of classes of the network and then gives those vectors to a neural network.

Chapter 2

SkyNet

SkyNet is a powerfull convolutional neural network developed by [8], winner of the *System Design Contest for low power object detection in the 56th IEEE/ACM Design Automation Conference (DAC-SDC)*. Its aim is to detect object inside images.

2.1 SkyNet Design Workflow

SkyNet has been designed with a *bottom-up* approach, considering the hardware constraints at the very beginning. This approach has made SkyNet extremely efficient and different from others CNNs, which have not been implemented to be hardware optimal.
Actually, in the standard *top-down* design process, an efficient DNN is selected as target, then, since it is typically expensive in term of resource usage, it is compressed using software and hardware optimization techniques such as quantization, pruning and layer fusion, so that it can be inferred on common FPGAs.
Skynet developers have found out why this kind of *top-down* approach is actually not the best one [8]: even if the DNN selected has great accuracy, the final accuracy when the network is inferred will depends strictly on the compression technology adopted. As example, in case of quantization, the accuracy may variate significantly in case the quantization is applied on parameters (i.e. weights) or on feature maps. As example consider the AlexNet network: as shown in Figure 2.1.(a) if the intermediate FMs are quantized the accuracy decreases more with respect to the case in which the parameters are quantized.
In addition, architectures with almost the same accuracy may have different resource usage depending on their implementation. As example Figure 2.1.(b) shows some implementations of the same network but with different FMs quantization and input size: it can be noticed how, by simply resizing the input image of a 0.9 factor, the BRAM (the on-chip memory in FPGA) utilization is almost halved. Similarly, 2.1.(c) shows how DPS utilization can vary a lot by using a different type of quantization for the weights and the FMs.

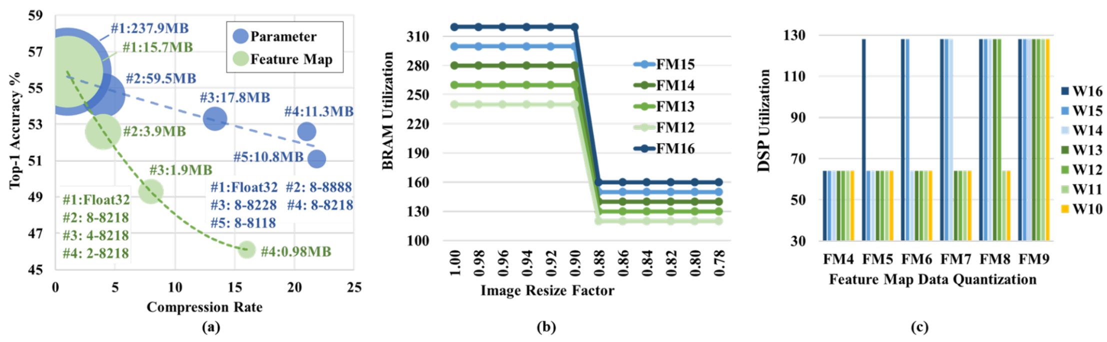


Figure 2.1: (a) Accuracy results for different quantization on FM and on parameters on the AlexNet network. (b) BRAM utilization for the same architecture for different resized input image and FM quantization. (c) DSP utilization for different quantization combinations on weights and FMs. [8]

2.1.1 The Skynet Bottom-Up Approach

The *bottom-up* approach followed by SkyNet developers is made of three stages:

1. Bundle selection and evaluation
2. Hardware-aware DNN search
3. Feature Addition

Bundle Selection and Evaluation

The first step is to search for the best *bundle* implementation. A *Bundle* is a set of sequential DNN layers (such as Convolution, BatchNormalization, Activation): repeated bundles forms a network. From an hardware point of view, a bundle is a set of IPs that need to be implemented in hardware.
In order to select the best bundle implementation, different bundles are proposed first, each of them containing a different order and different type of DNN layers. To search for the best one, the front-end and the back-end of the architecture are fixed based on the given task, while the in the middle a single type of bundle is repeated n times: this limit to one single type of bundle has been set in order to guarantee the best hardware efficiency.
To find out the best bundle for the SkyNet network, the front-end has been made of a input resizing unit, while the back-end has been made of a bounding box regression unit. Then, all the possible sketches have been trained with targeted dataset, to compute the latency and the accuracy of each bundle selection and to find out the pareto points, and thus the best bundle implementations have been selected.

Hardware-Aware DNN Search

In order to select the best network among the ones laying in the pareto curve of the previous step, a group-based *Particle Swarm Optimization* (PSO) algortihm

is adopted. In the PSO each DNNs proposed is seen as a particle in the design space, but since in this particular case every DNNs is made of the same repeated bundle, they are considered as *particle group*.
The pareto point of the group, i.e. the best position of the group in the design space, is labeled as P^i_{group}. Each P^i_{group} is composed of different particle n^i_j, where j is the particle in group i, characterized by a pair of vector $(fv1, fv2)$, where $fv1$ are the number of channels in each bundle replication, while $fv2$ is the Pooling layer position between bundles. Both the two vectors have a dimension equal to the number of bundle in n^i_j and impacts on the accuracy and hardware performance of the DNN.
The PSO algorithm adopted is here reported:

P $\leftarrow$ **InitialPopulation**(M, N)
while $itr <$ I **do**
 FastTraining(P, e_{itr})
 $Fit^j_i \leftarrow$ **GetFitnessVal**(P) #evaluate all candidates
 for *each group* i **do**
  GroupRank(i) #rank candidates in group i
  $N^i_{group} \leftarrow$ **GroupBest**(i) #select the best one in group i
  #get the group best position
  $P^i_{group}(fv1,\ fv2) \leftarrow$ **GetPosition**(N^i_{group})
  for *each candidate* $n^i_j(itr)$ *in group* i **do**
   #rank n^i_j across all passing iterations
   LocalRank(i, j)
   $N^{ij}_{local} \leftarrow$ **LocalBest**(i, j)
   #get the local best position
   $P^{ij}_{local}(fv1,\ fv2) \leftarrow$ **GetPosition** (N^{ij}_{local})
   #get the current position
   $P^i_j(fv1,\ fv2) \leftarrow$ **GetPosition**($n^i_j(itr)$)
   #get the velocity toward the local and the group best
   $V_{local} \leftarrow$ **GetV**(P^i_j, P^{ij}_{local})
   $V_{group} \leftarrow$ **GetV**(P^i_j, P^{ij}_{group})
   $n^i_j(itr+1) \leftarrow$ **Evolve**($n^j_i(itr)$, V_{local}, V_{group})
   end
 end
end

- P - Population: Initially a set of possible DNNs is generated through the function **InitialPopulation** with M groups and N networks for each group. The process is iterated I times and in the itr-th iteration, all networks are fast trained for e_{itr} epochs (**FastTraining**(P, e_{itr})), where e_{itr} increases with itr.

- Latency is estimated: in case of GPUs, it is directly computed on the one which has been used for the training, then its value is scaled to the target one. In case of FPGAs, a predefined IP-based DNN accelerator template [1] for hardware performance evaluation is followed and, to get the best performance, IPs are configured to fully consume the available resources.

- Then the *fitness value* Fit_i^j for each network n_j^i is computed. This value is given by:

$$Fit_i^j = Acc_j^i + \alpha \cdot (Est(n_j^i) - Tar)$$

where Acc_j^i is the validation accuracy of n_j^i, while $Est(n_j^i)$ represents the latency estimation on hardware and Tar is the targeted latency. The parameter α (where $\alpha < 0$) is used to balance between network accuracy and hardware performance.

- In standard PSO, the velocity $\overrightarrow{V_i^{itr+1}}$, namely the vector used in order to calculate the position in the design space for the particle in the next iteration, is computed considering the current velocity $\overrightarrow{V_i^{itr}}$, the personal best solution $\overrightarrow{P_i^d}$ and the global best solution $\overrightarrow{G_i^d}$.

$$\overrightarrow{V_i^{itr+1}} = w\overrightarrow{V_i^{itr}} + c_1 r_1(\overrightarrow{P_i^d} - \overrightarrow{X_i^d}) + c_2 r_2(\overrightarrow{G_i^d} - \overrightarrow{X_i^d})$$

In this case, DNNs in the same group update their positions based on the current design, the local best design (the best one across all passing iterations), and the group best design. Then to compute the velocity towards the local best V_{local} and the group best V_{group}, the differences between positions of current and the local/group best designs are computed. Since each position is represented by ($fv1$, $fv2$), position differences are evaluated by the mismatch of layer expansion factors $fv1$ and pooling spots $fv2$, respectively. Then, with the velocities V_{local} and V_{group}, the current network is evolved (line 22) by updating its position toward the local and the group best by a random percentage.

Feature Addition

It is possible to insert more features to the resulting DNNs in order to further improve the design. A possibility could be to substitute the ReLU layer with the ReLU6 layer, which has the advantage of representing FMs in a range restricted to [0, 6], meaning the use of less bits for representation, instead of using ReLU which operates in the [0, +∞] range (see Section 1.3).

2.1.2 Skynet Architecture

Therefore, the SkyNet architecture has been implemented following the reported *bottom-up* approach. The best configuration found has been identified in a bundle composed of *Depth-Wise Convolution*, *Batch Normalization*, *ReLU6*, *Pont-Wise Convolution*, *Batch Normalization* and *ReLU6* (see Figure 2.2).
This bundle is repeated three times followed by a *Max Pooling* layer, then it is repeated again three times. As reported in the Figure 2.2, after the last pooling

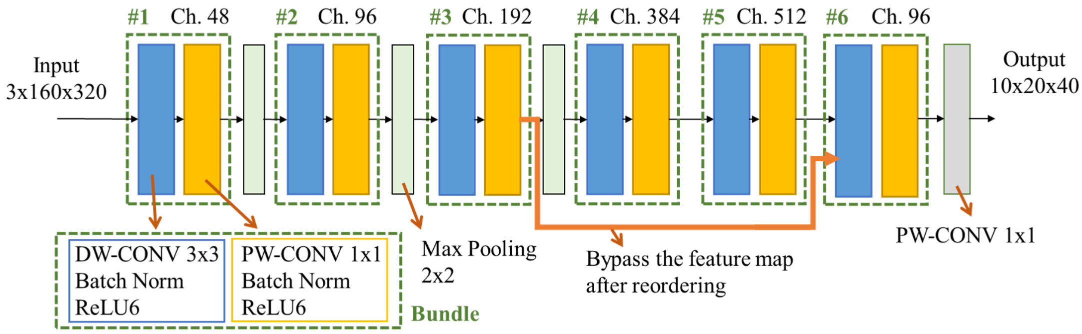


Figure 2.2: The SkyNet Architecture representation. [8]

layer a *feature map bypass and reordering* is performed: this feature has been added in order to make the network able to detect more easily objects that are very small, which have a very small bounding boxes. Actually, thanks to the *bypass* the feature map keeps an higher resolution, since no more calculus are performed on it; then *reordering* is used to align the size of the two FMs without losing information.
Furthermore, to reach the best accuracy and performance, SkyNet has been implemented with:

- *Depth-Wise/Point-Wise Convolution*
- *Layer Fusion*

Depth-Wise/Point-Wise Convolution

In order to reduce the computational cost, in each Bundle the standard convolution has been replaced by a *Depth-Wise/Point-Wise convolution*. Actually, as reported in [3], a standard convolutional layer has a computational cost of:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_{Fx} \cdot D_{Fy} \tag{2.1}$$

where D_K is the kernel size, M is the number of channels, N is the number of filters applied to the feature map and D_{Fx} and D_{Fy} are respectively the width and the height of the feature map (see Figure 2.3).

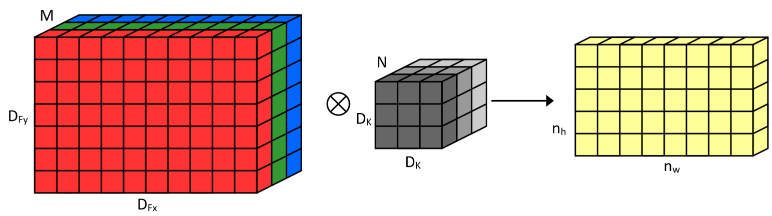


Figure 2.3: The picture describes the standard convolution of a 3 channels feature map with a 3×3 kernel filter. As it can be notice, the output size of the convolution respects the n_h and n_w formulas (1.6, 1.7) described in Section 1.1.

In order to reduce this cost, the convolution can be split in two phases: a *depth-wise* convolution, where single channels of the feature map are convoluted with single channel of the filter (see Figure 2.4), and a *point-wise* convolution, where a 1×1 kernel (see Figure 2.5) is used to combine the outputs of the depthwise convolution.
Thus, the *depth-wise convolution* has a computational cost of:

$$D_K \cdot D_K \cdot M \cdot D_{Fx} \cdot D_{Fy} \tag{2.2}$$

and the *point-wise convolution* has a cost of:

$$N \cdot M \cdot D_{Fx} \cdot D_{Fy} \tag{2.3}$$

Thus, the total cost of the depthwise separable convolutions is:

$$D_K \cdot D_K \cdot M \cdot D_{Fx} \cdot D_{Fy} + N \cdot M \cdot D_{Fx} \cdot D_{Fy} \tag{2.4}$$

Thus, comparing the standard convolution to the depth-wise convolution, the computation is reduced by a factor of:

$$\frac{D_K \cdot D_K \cdot M \cdot D_{Fx} \cdot D_{Fy} + N \cdot M \cdot D_{Fx} \cdot D_{Fy}}{D_K \cdot D_K \cdot M \cdot N \cdot D_{Fx} \cdot D_{Fy}} = \frac{1}{N} + \frac{1}{D_K^2} \tag{2.5}$$

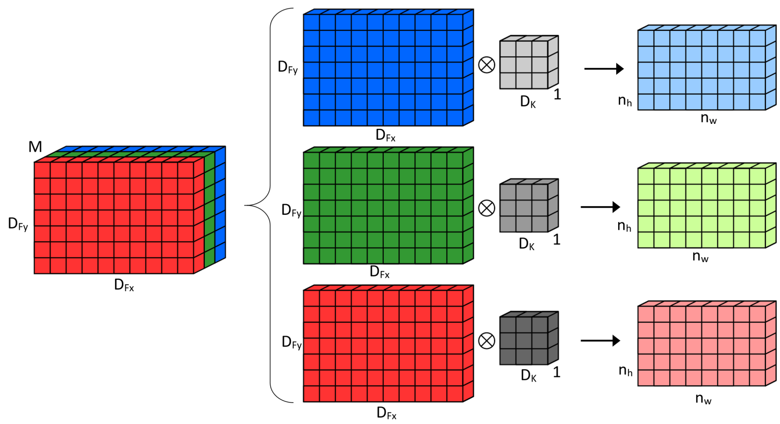


Figure 2.4: The picture describes the depthwise convolution, where the 3 input channels of the image are separated and convoluted with 3 different kernels.

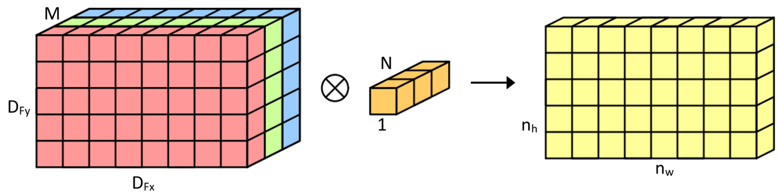


Figure 2.5: The picture describes the point-wise convolution, where the 3 feature maps of the depthwise convolution are convoluted with N 1×1 kernel in order to obtain the final feature map.

In order to better understand the power of this methodology consider the case in which the bundle of SkyNet is not composed of a *Detpth-Wise/Point-Wise convolution* but by a standard convolution.
In this case the computation cost required is given by:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_{Fx} \cdot D_{Fy} = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 320 \cdot 160 = 4147200$$

While, thanks to the *Depth-Wise/Point-Wise convolution* it is actually:

$$\begin{aligned} &D_K \cdot D_K \cdot M \cdot D_{Fx} \cdot D_{Fy} + N \cdot M \cdot D_{Fx} \cdot D_{Fy} = \\ &= 3 \cdot 3 \cdot 3 \cdot 320 \cdot 160 + 3 \cdot 3 \cdot 320 \cdot 160 = 1843200 \end{aligned}$$

Thus 4147200-1843200=2304000 operations do not need to be performed, meaning a great saving in term of computational cost (more or less the 44%).

Layer Fusion

The traditional linear structure of CNNs, where each layer is evaluated after the previous one, generates a large amount of intermediate data.
Consider, as example, two subsequent convolutional layers: in order to get the output feature map, the first layer is computed first, generating an intermediate feature map which is then used as input to the second layer. This intermediate feature map, which is needed only as input to the second layer, is in general extremely consistent and does not fit in the on-chip memory of common FPGAs. Thus, it has to be saved in the off-chip memory and reloaded when the second convolution layer is executed.

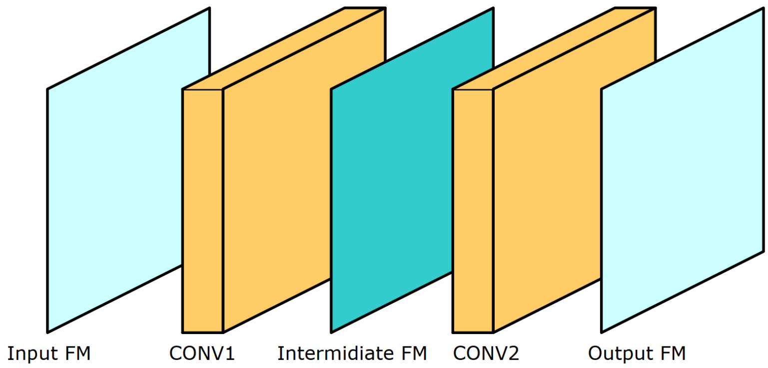


Figure 2.6: Structure of two subsequent convolution, highlighting the presence of the intermediate FM.

In order to avoid this transfer from on-chip to off-chip and then again to on-chip memory, a possibility is to *fuse* the two convolutional layer together.

Consider the example in Figure 2.7. The input feature map is a 7×7 matrix, which is convoluted by the first convolutional layer CONV1 by a 3×3 kernel; the

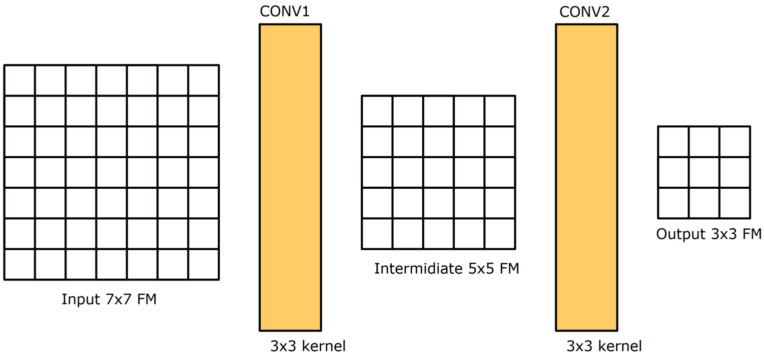


Figure 2.7: An example of two sequential CNN layers: the intermediate output is saved in the off-chip memory while it is computed. For simplicity, only one channel is displayed.

intermediate feature map is a 5×5 matrix, which is convoluted by the second convolutional layer CONV2 with a 3×3 kernel to give the final output feature map of 3×3. Following the standard flow, the CONV1 layer is executed entirely and its output, i.e. the intermediate feature map, is saved in the off-chip memory. Then the intermediate feature map is loaded back in the on-chip memory to perform the second convolution by layer CONV2.

The idea of the *layer fusion* technique is to exploit the locality in the convolution's dataflow: actually, each output value of the feature map computed by a convolutional layer depends only on a small window of the input feature map.
As reported from the example in Figure 2.8, the computation of one of the element of the output feature map depends only on a 3×3 window of the intermediate feature map, and this 3×3 window itself depends only on a 5×5 window of the input feature map. The required input feature map sizes are simply obtained reversing the formulas 1.6 and 1.7:

$$n_{FMintermidiate} = s \cdot (n_{FMoutput} - 1) + n_K \tag{2.6}$$

Considering this facts, there is no need to upload the entire 7×7 input feature map to obtain one of the elements of the output, and also no transfer of the intermediate feature map from on-chip to off-chip and vice-versa is required, since the output can be directly computed.

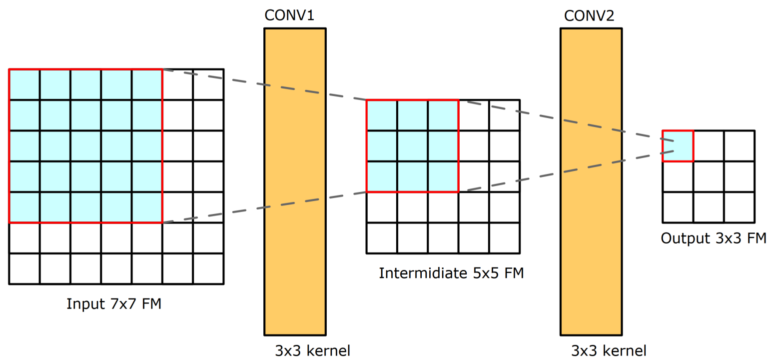


Figure 2.8: Example of layer fusion technique, highlighting the dependency among the output, the intermediate FM and the input.

After the computation of one element of the output feature map, in order to compute the second one, the input feature map has to be shifted to right by one position (assuming the case in which the *stride* parameter is equal to 1): in this case just a line of data has to be loaded in the memory for the input feature map (the pink one in the Figure 2.9), while the others are still present from the previous convolution.

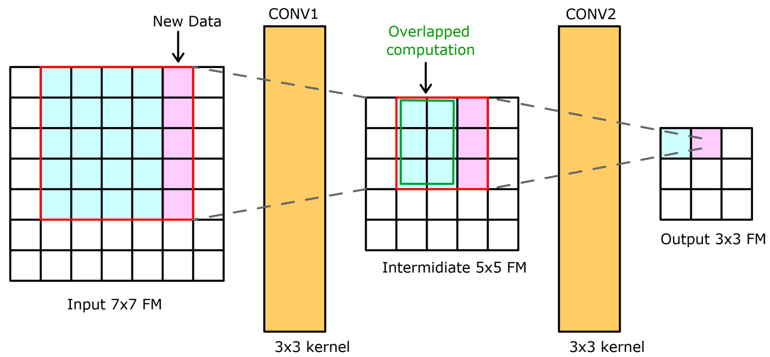


Figure 2.9: The sketch highlights how the input data has to be loaded from off-chip memory.

Concerning the intermediate feature map, as it can be noticed, some of the values have already been computed by the previous convolution, thus can be reused to compute the second value of the output feature map: this implies a saving in terms of computations, but requires on-chip buffering.

2.2 SkyNet Results on GPU and FPGA

SkyNet network is trained on DAC-SDC dataset, using data augmentation to distort, jitter, crop and resize input image to 160x320. The optimizer adopted to update the weights parameter is the *Stochastic Gradient Descent* (SDG), with an initial learning rate of $1e^{-4}$, which is decreased at every epoch reaching the value of $1e^{-7}$.

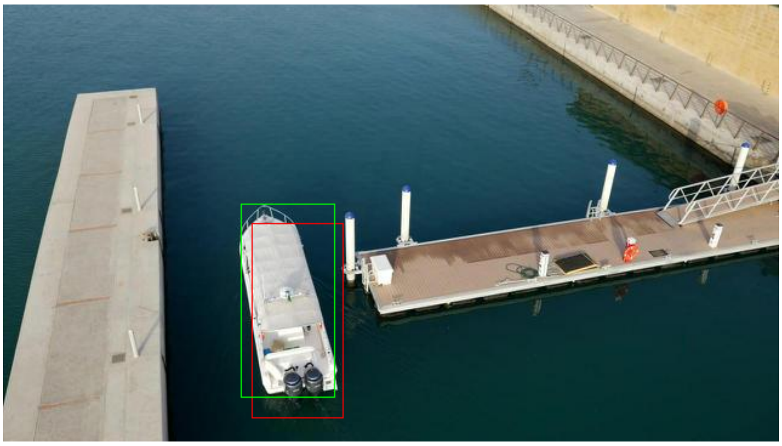


Figure 2.10: The predicted bounding box in red and the true bounding box in green. The IOU is given by the ratio among the area of intersection of the two boxes and the area of the true bounding box.

Since SkyNet is a CNN used for object detection, the metric used in order to evaluate the result is the *Intersection Over Union* (IOU), which represent the ratio among the overlap of the true and the predicted bounding box with the true bounding box, as:

$$\mathrm{IOU} = \frac{\mathrm{Overlap}}{\mathrm{True\,BB}} \tag{2.7}$$

Actually, the dataset of the DAC-SDC contains with the images also the position of the object inside those images: in this way the optimizer can evaluate how much the predicted bounding box differs from the true one. In Figure 2.10 an example of true and predicted bonding box is reported. The Figure has been taken from the DAC dataset images, which has been used to train the SkyNetQuant network described in Section 3.2. The best IOU result of SkyNet on GPU is **0.741**.

2.2.1 Implementation on TX2 GPU

The model has then been optimized for TX2 GPU implementation, by dividing the SkyNet execution in four main steps:

1. Image fetching from memory;
2. Image resizing and preprocessing;
3. SkyNet inference;
4. Bounding Boxes computation and store of the result in DDR memory.

and applying pipelining (see Figure 2.11) by fusing togheter the first two steps. With respect to the original sequential design, the pipelined one has increased its speed of a factor 3.35X and the throughput of 67.33 FPS.

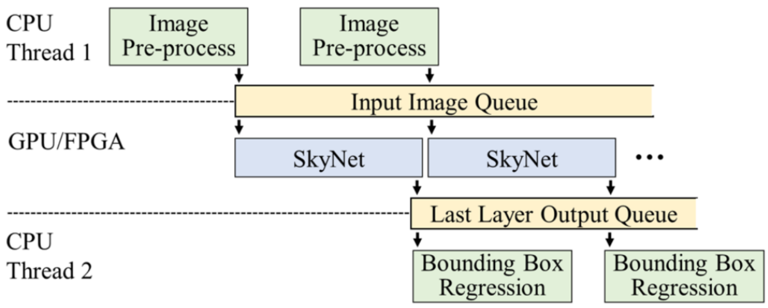


Figure 2.11: SkyNet pipeline on TX2 GPU.

2.2.2 Impelementation on FPGA

The target FPGA is the Ultra96, Xilinx Zynq UltraScale+ MPSoC board. Due to the FPGA resource limits, the SkyNet FMs and weights have been converted from *float32* to *fixed point* representation: 9 bits for the FMs and 11 bits for the weights, dropping the accuracy from 0.741 to **0.727**.

Chapter 3

Quantization Aware-Training Using Brevitas

Starting from the SkyNet network reported above, the aim of this thesis work has been to develop a better implementation by maintaining as possible the SkyNet's original structure.
In order to do that, the idea taken in consideration has been to implement a *quantized* version of SkyNet, by using ***Brevitas*** as quantization tool.

Brevitas is an extremely new *PyTorch* library for quantization-aware training (actually, no documentation has been provided yet) developed by the Xilinx Research Lab.
This library provides several quantized version of the standard PyTorch layers and it is extremely easy to use: given a model made of PyTorch layers, the user simply has to replace them in the code with their Brevitas implementation.
At the moment, Brevitas provides only the layers reported in Table 3.1: as it can be noticed the implementation of normalization layers is still missing. However this is not a problem, since Brevitas allows the user to mix together Brevitas and PyTorch layers, meaning that the user can really decide which layer to quantize in the model. The quantized version of SkyNet, namely `SkyNetQuant` has been actually developed with the standard `BatchNorm2d` from PyTorch, as reported in Section 3.2.

3.1 Quantization in Brevitas

Brevitas library is built upon the PyTorch library, implementing the quantization on the standard PyTorch layers by giving to them quantized parameters.
Actually, considering the sketch of `QuantConv2d` in Figure 3.1, the layer is build inheriting the standard `Conv2d` PyTorch layer and by instantiating a quantization class, called `QuantWBIOL` (which stands for `QuantWeightBiasInputOutputLayer`) which receives the input, the bias and the weights of the `Conv2d` layer and returns back their quantization version, thus the convolution performed by `Conv2d` is done among quantized parameters.

PyTorch Layer	Brevitas Layer
Convolutional Layers	
nn.Conv1d	QuantConv1d
nn.Conv2d	QuantConv2d
nn.ConvTranspose1d	QuantConvTranspose1d
nn.ConvTranspose2d	QuantConvTranspose2d
Pooling Layers	
nn.MaxPool1d	QuantMaxPool1d
nn.MaxPool2d	QuantMaxPool2d
nn.AvgPool2d	QuantAvgPool2d
nn.AdaptiveAvgPool2d	QuantAdaptiveAvgPool2d
Non-linear Activations	
nn.Hardtanh	QuantHardTanh
nn.ReLU	QuantRelu
nn.Sigmoid	QuantSigmoid
nn.Tanh	QuantTanh
Dropout Layers	
nn.Dropout	QuantDropout

Table 3.1: The table reports the PyTorch layers which have already a correspondent layer in the Brevitas library. Notice that there is no Brevitas version of `nn.BatchNorm2d`: actually this layer still has to be implemented.

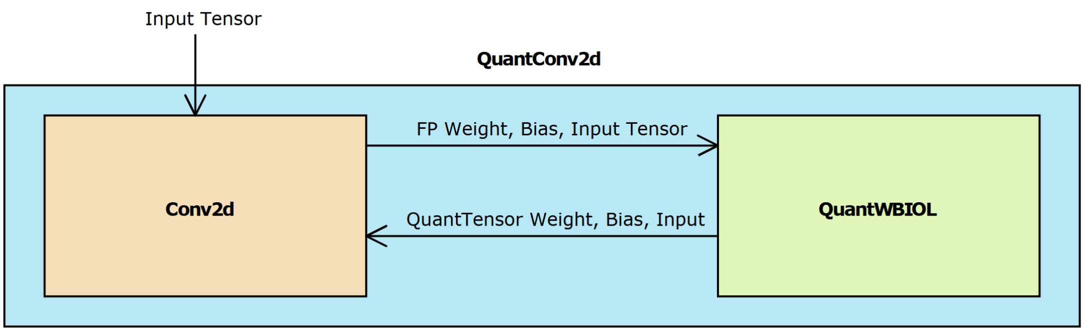


Figure 3.1: The Figure describes the implementation of the `QuantConv2d` layer in Brevitas, made inheriting the standard PyTorch `Conv2d` and instantiating as quantizer the `QuantWeightBiasInputOutputLayer`.

In order to explain how this mechanism is implemented, consider the `QuantConv2d` layer implementation code:

```
from typing import Union, Tuple, Type, Optional
import math

import torch
from torch import Tensor
from torch.nn import Conv1d, Conv2d
from torch.nn import functional as F
```

```python
from torch.nn.functional import conv2d

from brevitas.inject import BaseInjector as Injector
from brevitas.function.ops import max_int
from brevitas.function.ops_ste import ceil_ste
from brevitas.proxy.parameter_quant import WeightQuantProxyProtocol, BiasQuantProxyProtocol
from brevitas.proxy.runtime_quant import ActQuantProxyProtocol
from brevitas.quant_tensor import QuantTensor
from brevitas.inject.defaults import Int8WeightPerTensorFloat
from .quant_layer import QuantWeightBiasInputOutputLayer as QuantWBIOL


__all__ = ['QuantConv1d', 'QuantConv2d']

    ...


class QuantConv2d(QuantWBIOL, Conv2d):

    def __init__(
    self,
    in_channels: int,
    out_channels: int,
    kernel_size: Union[int, Tuple[int, int]],
    stride: Union[int, Tuple[int, int]] = 1,
    padding: Union[int, Tuple[int, int]] = 0,
    dilation: Union[int, Tuple[int, int]] = 1,
    groups: int = 1,
    bias: bool = True,
    padding_type: str = 'standard',
    weight_quant: Union[WeightQuantProxyProtocol, Type[Injector]] = Int8WeightPerTensorFloat,
    bias_quant: Union[BiasQuantProxyProtocol, Type[Injector]] = None,
    input_quant: Union[ActQuantProxyProtocol, Type[Injector]] = None,
    output_quant: Union[ActQuantProxyProtocol, Type[Injector]] = None,
    return_quant_tensor: bool = False,
    **kwargs) -> None:
    Conv2d.__init__(
    self,
    in_channels=in_channels,
    out_channels=out_channels,
    kernel_size=kernel_size,
    stride=stride,
    padding=padding,
    dilation=dilation,
    groups=groups,
    bias=bias)
    QuantWBIOL.__init__(
    self,
    weight=self.weight,
    bias=self.bias,
    weight_quant=weight_quant,
    bias_quant=bias_quant,
    input_quant=input_quant,
```

```
191         output_quant=output_quant,
192         return_quant_tensor=return_quant_tensor,
193         **kwargs)
```

As described from the Figure 3.1, the `QuantConv2d` layer is implemented inheriting two classes: `Conv2d` (line 174), the class that implements the convolution in PyTorch and that instantiates the weight and bias parameters, and `QuantWBIOL` (line 184) which receives the weight and bias of `Conv2d` (see line 186-187) and compute its quantized version, so that the convolution is performed using quantized parameters.
As for the standard `Conv2d`, to instantiate `QuantConv2d`, the user has to specify the dimension of the input and output channels, the dimension of the filter size and other parameters such as stride, padding, dilation, group and bias. The main difference is that in this case, the user can select a *quantizer* for the weights and the biases (but also for the input and the output): in this case, the standard `QuantConv2d` applies quantization only on the weights parameters.

Brevitas already provides several quantizers (they can be found in folder `brevitas.quant` at [6]) and each of them is fully customizable by the user according to its own requirements.
Each quantizer is characterized by different parameters whose values define how the quantizer should work; the mains ones are:

- `quant_type`: the kind of quantization that the library implements for the parameter. The available most used ones are:
  - `QuantType.INT`: **integer quantization** implemented by the module `IntQuant()`. Giving an input Tensor, `IntQuant()` implements scale, shifted, uniform integer quantization according to the parameters scale, zero-point and bit-width, which are given as argument. It returns the quantized tensor in a de-quantized format (see section B.1 for code implementation).
  - `QuantType.BINARY`: **binary quantization** implemented by the module `BinaryQuant()`. It returns the quantized output in the de-quantized format, the scale, the zero-point and the bith_width, which in this case is equal to 1 (see section B.2 for code implementation).
  - `QuantType.TERNARY`: **ternary quantization** implemented by the module `TernaryQuant()`. Given an input tensor, it returns its quantized output in de-quantized format, scale, zero-point and bit_width, which in this case is always equal to 2 (see section B.3 for code implementation).
- `bit_width`: the amount of bit on which the original parameter is quantized.
- `narrow_range`: boolean parameter that if it is `True` implements the value in a range from $(-2^{N-1}+1)$ to (2^N-1), instead of -2^{N-1} to (2^N-1), where N correspond to `bit_width`. As example, in case N=8, if `narrow_range=True`

the quantized value will go from-127 to 127 and not from -128 to 127; this will make the hardware inference more efficient.

- `signed`: if it is `True` the quantized value can be both positive and negative.

In this case, the layer `QuantConv2d` uses as default the quantizer `Int8WeightPerTensorFloat` (see line 168) for the weights parameter which, as reported in [6], is "8-bit narrow per-tensor signed int weight quantizer with floating-point scale factor computed from backpropagated statistics of the weight tensor", i.e. the weight of the convolution kernel are quantized on 8 bit in a range which goes from -127 to 127, with a floating point scale factor.
The formula used by `Int8WeightPerTensorFloat` to compute the scale is given by:

$$scale = \frac{th}{int\_th} \tag{3.1}$$

where th is the threshold and it is defined as the maximum absolute value in an input tensor X:

$$th = \max_{i,j=1,\dots,dim(X)} \{|x_{i,j}|\} \tag{3.2}$$

while int_th is the integer threshold given by:

$$int\_th = \begin{cases} 2^{N-1} - 1 & \text{if } \texttt{signed}{=}\texttt{True} \\ 2^{N} - 1 & \text{if } \texttt{signed}{=}\texttt{False} \end{cases}$$

Then, the quantization is performed doing the ratio among the floating point value and the scale factor:

$$\mathrm{IntW} = \frac{\mathrm{FPW}}{scale} \tag{3.3}$$

Thus, considering the following numerical example, in which the quantization is performed on 4 bits, with signed True, the quantization will be computed with these steps:

$$\mathrm{FPW} = \begin{pmatrix} 0.678 & 0.231 & 0.912 \\ -0.234 & 0.654 & 0.342 \\ -0.123 & 0.825 & -0.702 \end{pmatrix}$$

$$\begin{aligned} th &= \max|FPW_{ij}| = 0.912 \\ int\_th &= 2^{N-1} - 1 = 2^{4-1} - 1 = 7 \\ scale &= \frac{th}{int\_th} = \frac{0.912}{7} = 0.130 \end{aligned}$$

Then to compute the quantized weight:

$$\mathrm{IntW} = \frac{\mathrm{FPW}}{scale} \approx \begin{pmatrix} 5 & 2 & 7 \\ -2 & 5 & 3 \\ -1 & 6 & -5 \end{pmatrix}$$

where $\approx$ approximate the result to the nearest integer.

Thus, coming back to `QuantConv2d` implementation, only the weights parameters of `Conv2d` are quantized by `QuantWBIOL`.
It is important to notice that during the training of the network the quantized parameters (and the scale) are recomputed each time the *optmizer* updates the original non-quantized parameters (FPW in the example). Also it is important to highlight that the convolution is not performed among the input and integer representation of the weight, but with the quantized weight in the *de-quantized format*. Actually, as seen from code B.1 at line 89, the quantizer, giving the scale, the zero_point, the bit_width and the input tensor X, computes its integer representation `y_int`, but then it returns the quantized parameter in the de-quantized float representation. Thus, during training the convolution operations are performed among floating point values (see Figure 3.2).
The weights' de-quantized format is given by:

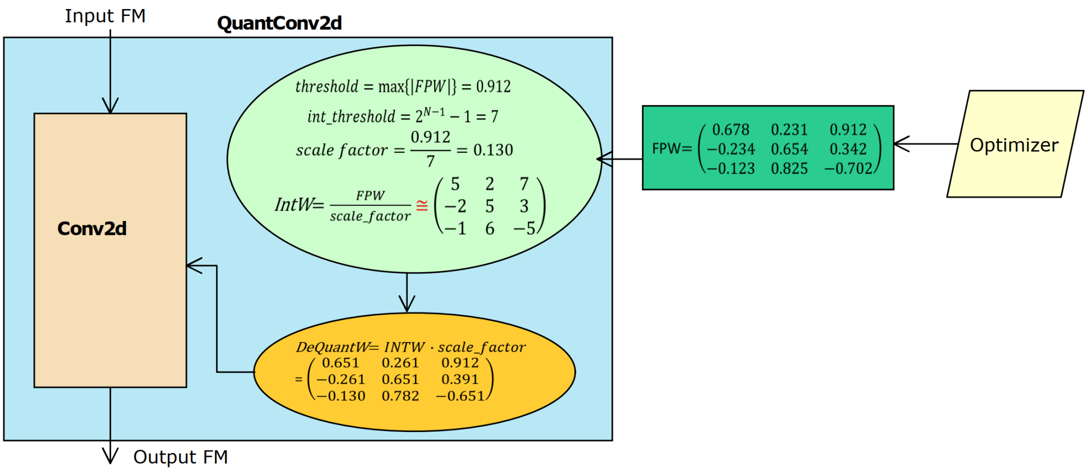


Figure 3.2: Numerical example of quantization in Brevitas.

$$\text{DeQuantW} = \text{IntW} \cdot scale \tag{3.4}$$

which in this specific case is:

$$\text{DeQuantW} = \text{IntW} \cdot scale = \begin{pmatrix} 5 & 2 & 7 \\ -2 & 5 & 3 \\ -1 & 6 & -5 \end{pmatrix} \cdot 0.130 = \begin{pmatrix} 0.651 & 0.261 & 0.912 \\ -0.261 & 0.651 & 0.391 \\ -0.130 & 0.782 & -0.651 \end{pmatrix}$$

Of course, when inferring the network on FPGA, the weights are exported and stored in the integer quantized format and, in order to keep the result correct as the one during training, the output FM will be multiplied times the scale factor, since it is true that:

$$\text{InputFM} * \text{DeQuantW} = \text{InputFM} * \text{IntW} \cdot \text{scale} \tag{3.5}$$

A similar layer construction is adopted also for the other Brevitas layer.

3.1.1 Quantization of Activation and Pooling Layers

In Brevitas, also activation and pooling layers are quantized. Actually, even if these layers do not learn any parameters, their output can be quantized. Considering as example the common `ReLU` layer described in section 1.3, it is implemented by Brevitas in the following way:

```
class QuantReLU(QuantNLAL):

    def __init__(
            self,
            input_quant: Union[ActQuantProxyProtocol, Type[Injector]] = None,
            act_quant: Type[Injector] = Uint8ActPerTensorFloat,
            return_quant_tensor: bool = False,
            **kwargs):
        QuantNLAL.__init__(
            self,
            act_impl=nn.ReLU,
            passthrough_act=True,
            input_quant=input_quant,
            act_quant=act_quant,
            return_quant_tensor=return_quant_tensor,
            **kwargs)
```

Again, as for the convolutional layer, `QuantReLU` is composed of two classes: the standard `nn.ReLU` imported from PyTorch and `QuantNLAL` (`QuantNonLinearActivationLayer`) defined in Brevitas, which is simply used in order to quantize the `nn.ReLU` output. In this case, the default quantization is performed on unsigned values (due to the ReLU behavior) on 8 bits, but again it is fully customizable by the user.
In this case, when `QuantRelu` receives the input, it first executes the `nn.ReLU`

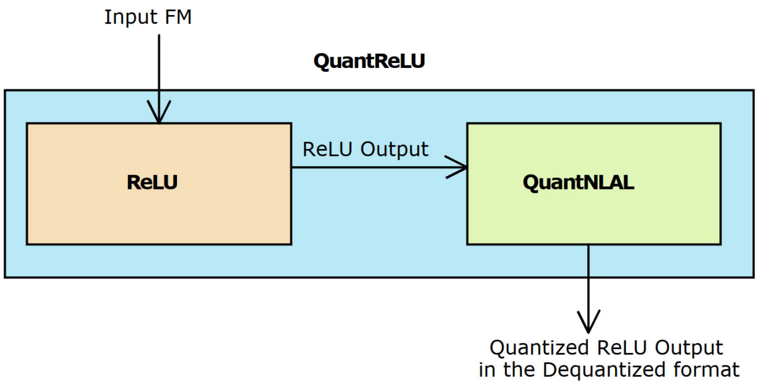


Figure 3.3: The Figure describes the implementation of the `QuantReLU` layer in Brevitas, made inheriting the standard PyTorch `ReLU` and instantiating as quantizer the `QuantNonLinearActivationLayer`.

function, filtering positive values, then its output is quantized by `QuantNLAL`: again the scale factor is computed as described in equation 3.5 and the integer and the de-quantized output values are computed. As in the previous cases, the formal output of the `QuantReLU` is the de-quantized one.

3.1.2 How to define custom quantizers in Brevitas

In Brevitas the user can also define its own custom quantizer. As example, consider the following code:

```
1 from brevitas.inject import BaseInjector as Injector
2 from brevitas.inject.enum import QuantType, BitWidthImplType, ScalingImplType
3 from brevitas.inject.enum import RestrictValueType, StatsOp
4 from brevitas.core.zero_point import ZeroZeroPoint
5 from brevitas.nn import QuantConv2d
6
7 class MyLearnedWeightQuant(Injector):
8     quant_type = QuantType.INT
9     bit_width_impl_type = BitWidthImplType.PARAMETER
10    narrow_range = True
11    signed = True
12    zero_point_impl = ZeroZeroPoint
13    scaling_impl_type = ScalingImplType.PARAMETER_FROM_STATS
14    scaling_stats_op = StatsOp.MAX
15    scaling_per_output_channel = False
16    restrict_scaling_type = RestrictValueType.LOG_FP
17    bit_width = 4
18
19 conv = QuantConv2d(..., weight_quant=MyLearnedWeightQuant)
```

The user firstly defines the quantizer `MyLearnedWeightQuant` (line 7) and then replaces the standard `Int8WeightPerTensorFloat` in `QuantConv2d` with the new quantizer (line 19).
As `Int8WeightPerTensorFloat`, to define `MyLearnedWeightQuant`, some already built-in parameters are used, such as the quantization of integer type (line 8) on 4 bits (line 17) and the zero point in the half of the quantization interval. Notice that in this case the parameter `bit_width_implementation_type` is not constant, but variable (line 9): this means that it is a learnable parameter whose value will be determined during the training.

3.2 SkyNet Quantization using Brevitas

The qunatized model of SkyNet, named `SkyNetQuant`, has been developed with the following code:

```
from collections import OrderedDict
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init as init
from region_loss_cuda import RegionLoss
from utils import *
from collections import OrderedDict

```

```python
#BREVITAS LIBRARY
import brevitas.nn as qnn
from brevitas.core.quant import QuantType

class PrintLayer(nn.Module):
    def __init__(self):
        super(PrintLayer,self).__init__()

    def forward(self,x):
        print('Printing a layer:')
        print(x)
        return x

class ReorgLayer(nn.Module):
    def __init__(self, stride=2):
        super(ReorgLayer, self).__init__()
        self.stride = stride
    def forward(self, x):
        stride = self.stride
        assert(x.data.dim() == 4)
        B = x.data.size(0)
        C = x.data.size(1)
        H = x.data.size(2)
        W = x.data.size(3)
        assert(H % stride == 0)
        assert(W % stride == 0)
        ws = stride
        hs = stride
        x = x.view([B, C, H//hs, hs, W//ws, ws]).transpose(3, 4).contiguous()
        x = x.view([B, C, H//hs*W//ws, hs*ws]).transpose(2, 3).contiguous()
        x = x.view([B, C, hs*ws, H//hs, W//ws]).transpose(1, 2).contiguous()
        x = x.view([B, hs*ws*C, H//hs, W//ws])
        return x


class SkyNetQuant(nn.Module):
    def __init__(self, weight_bit_width=4, act_bit_width=4, in_bit_width=4):
        super(SkyNetQuant, self).__init__()
        self.width = int(320)
        self.height = int(320)
        self.header = torch.FloatTensor([0,0,0,0])
        self.seen = 0
        self.reorg = ReorgLayer(stride=2)

        def conv_dw_Brevitas(inp, oup, stride):
            return nn.Sequential(
                qnn.QuantConv2d(in_channels=inp, out_channels=inp, kernel_size=3,
                    stride=1, padding=1, groups=inp, bias=False,
                    weight_bit_width=weight_bit_width),
                nn.BatchNorm2d(inp),
                qnn.QuantReLU(bit_width=act_bit_width, max_val=6),
                qnn.QuantConv2d(in_channels=inp, out_channels=oup, kernel_size=1,
                    stride=1, padding=0, groups=1, bias=False,
                    weight_bit_width=weight_bit_width),
                nn.BatchNorm2d(oup),
```

```
                qnn.QuantReLU(bit_width=act_bit_width, max_val=6),
            )

        self.model_p1 = nn.Sequential(
            conv_dw_Brevitas( 3, 48, 1), #dw1
            qnn.QuantMaxPool2d(kernel_size=2, stride=2),
            conv_dw_Brevitas( 48, 96, 1), #dw2
            qnn.QuantMaxPool2d(kernel_size=2, stride=2),
            conv_dw_Brevitas( 96, 192, 1), #dw3
        )
        self.model_p2 = nn.Sequential(
            qnn.QuantMaxPool2d(kernel_size=2, stride=2),
            conv_dw_Brevitas(192, 384, 1), #dw4
            conv_dw_Brevitas(384, 512, 1), #dw5
        )
        self.model_p3 = nn.Sequential( #cat dw3(ch:192 -> 768) and dw5(ch:512)
            conv_dw_Brevitas(1280, 96, 1),
            qnn.QuantConv2d(in_channels=96, out_channels=10, kernel_size=1,
                weight_bit_width=weight_bit_width, bias=False),
        )


        self.loss = RegionLoss([1.4940052559648322, 2.3598481287086823,
            4.0113013115312155, 5.760873975661669],2)
        self.anchors = self.loss.anchors
        self.num_anchors = self.loss.num_anchors
        self.anchor_step = self.loss.anchor_step
        self._initialize_weights()

    def forward(self, x):

        x_p1=self.model_p1(x)
        x_p1_reorg = self.reorg(x_p1)
        x_p2 = self.model_p2(x_p1)
        x_p3_in = torch.cat([x_p1_reorg, x_p2], 1)
        x = self.model_p3(x_p3_in)

        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, qnn.QuantConv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def quantize_weight_extractor(self):
        for m in self.modules():
            if isinstance(m, qnn.QuantConv2d):
                print(m.weight_quant(m.weight))
            elif isinstance(m, nn.BatchNorm2d):
                print(m.weight)
```

As it can be notice from the code, the PyTorch layer `Conv2d`, `ReLU` and `MaxPool2d` have been replaced by their Brevitas implementation, while the standard PyTorch `BatchNorm2d` layer has been left, for the reasons explained in Section 3. Concerning the quantization of the weights, the default quantizer `Int8WeightPerTensorFloat` is adopted with bit_width set to 4, meaning that the integer weight value will be in a range from -7 to 7.
Also the activation function is quantized on 4 bit: in this case the `QuantRelu` behaves like a standard ReLu layer, with the only difference that its output is quantized on 4 bit.
Notice that beside network quantization also the network input size has been modified from $3 \times 160 \times 320$ to $3 \times 320 \times 320$. This variation is due to the fact that the current release of FINN, i.e. the tool used to optimize the inference on FPGA of the network, support only squared feature maps and not rectangular ones.

3.3 SkyNetQuant Accuracy Results

The training of SkyNetQuant has been performed using the *Adam optimizer*, with a starting learning rate of 0.001 and the dataset of the 2020 DAC-SDC. As for the original Skynet, the images have been preprocessed using dataset augmentation technique.
The highest IOU reached by SkyNetQuant is **0.7248**, which is more or less equal to the IOU of the original SkyNet, which is 0.741. Given these results the new SkyNet implementation seems to be efficient, since the IOU is almost the same of the original, while the amount of memory requested for the weights is smaller. Unfortunately this efficiency cannot be demonstrated since it has not been possible to inference the SkyNetQuant model on FPGA, due to the reasons explained in Section 4.

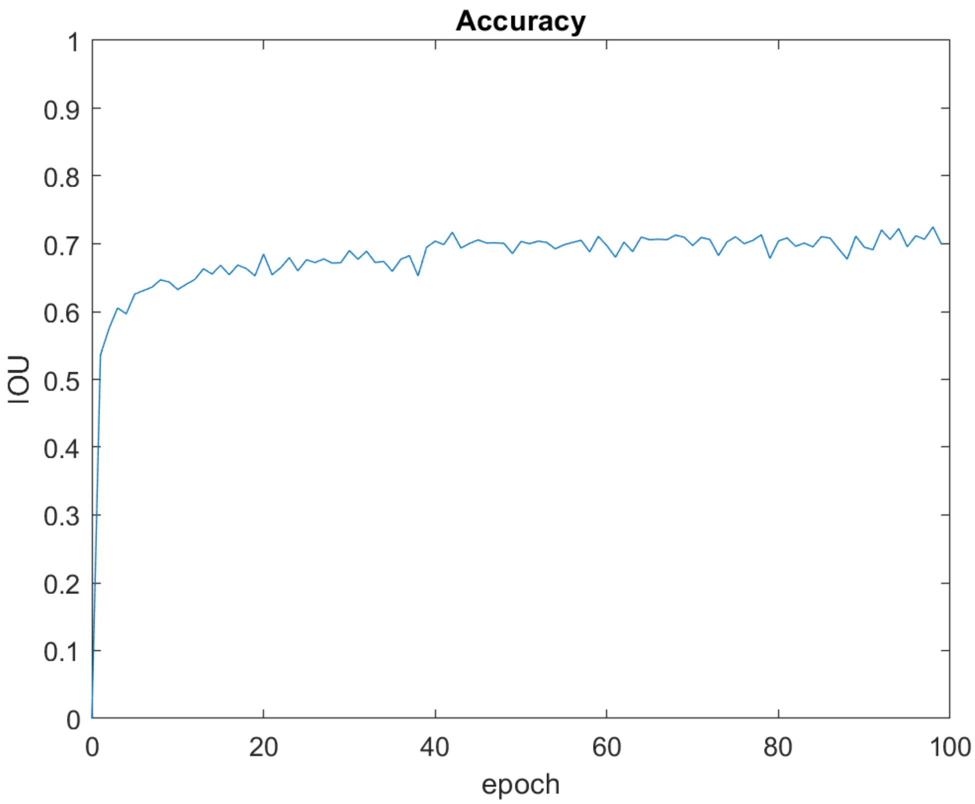


Figure 3.4: Accuracy IOU results of SkyNetQuant at every epochs during the training.

Chapter 4

FINN

FINN is a new powerful tool developed by the Xilinx Research Lab that can be used to synthesize and implement quantized network on FPGA.
To work with FINN, the user should have:

- Ubuntu 18.04 with bash installed;
- Vivado 2019.1 or 2020.1 installed;
- Docker, a virtual container for applications;

It is also possible to avoid the use of Docker, by installing FINN from the command line: in this case the user has to modify several files to make FINN work (if possible, it is better to use Docker).

FINN is a *compiler infrastructure* [4], namely a collection of scripts that can be used to convert a QNN into a custom FPGA accelerator that performs high-performance inference. Indeed, to use FINN the user has to prepare the script to transform and inference the model on FPGA.
Furthermore there is a function, which is still under development, called `built_dataflow`, which executes all the transformation steps by itself, so that the user has just to give the trained QNN model as input. However, this function, as FINN itself, is extremely new and works only with very small and standard structure QNNs, thus is not suitable for SkyNetQuant.

The FINN design flow is reported in Figure 4.1 and can be summarized in three main steps:

1. **ONNX export**: after the training, the network has to be exported in the ONNX format in order to be imported in FINN. At the moment, Brevitas is the only tool that supports the export to FINN.
2. **Network Transformation and Streamlining**: the ONNX model is transformed with several FINN transformations in such a way that each layer (represented by one ore more ONNX *nodes*) is suitable for the *finn-hls library.*

3. **Hardware Generation**: giving a target FPGA and clock frequency, the network is inferred on hardware.

4.1 ONNX export: Brevitas export to FINN

After the training, the Brevitas model is exported as ONNX model, so that it can be used in FINN.
The export is performed by loading the best `state_dict`[1] on the model, as reported in the following code:

```
import onnx
import os
import brevitas.onnx as bo
from model4bit import *

#The SkyNetQuant model is loaded with the parameters that
#have reached the best accuracy results.
checkpoint_path= os.getcwd()+"/checkpoint/best.tar"
model = SkyNetQuant()
checkpoint = torch.load(checkpoint_path)
model.load_state_dict(checkpoint['state_dict'])

#SkyNetQuant is exported to ONNX
quantskynet=model.eval()
dir=os.getcwd()+"/finn_model/"
export_onnx_path = "quantskynet_brevitas_export.onnx"
input_shape = (1, 3, 320, 320)
bo.export_finn_onnx(quantskynet, input_shape, dir+export_onnx_path)
```

Adopting the `load_state_dict` function by PyTorch, the `SkyNetQuant()` is loaded with the quantized parameters that had made the model reach its best accuracy, i.e. its highest IOU of 0.7248.
Then, when executing the `export_finn_onnx`, each weight that during training was given to the convolutional layer in its de-quantized format is converted to its integer representation, which is given by the equation 3.3 here reported:

$$\mathrm{IntW} = \frac{\mathrm{FPW}}{scale}$$

The exported model can be visualized by the user adopting *Netron*[2], which is a tool used to display ONNX networks (see Figure 4.4).

As it can be noticed the exported ONNX model is characterized by different types of nodes, each one representing a layer of the Brevitas model. In addition, it is possible to notice the multiplication among the output of the `Conv` layer with the scale factor, as explained in equation 3.5 reported in Section 3.1.

[1] A `state_dict` is simply a Python dictionary object that maps each layer to its parameter tensor (`https://pytorch.org/tutorials/beginner/saving_loading_models.html`).

[2] Netron can be found here: `https://github.com/lutzroeder/netron`

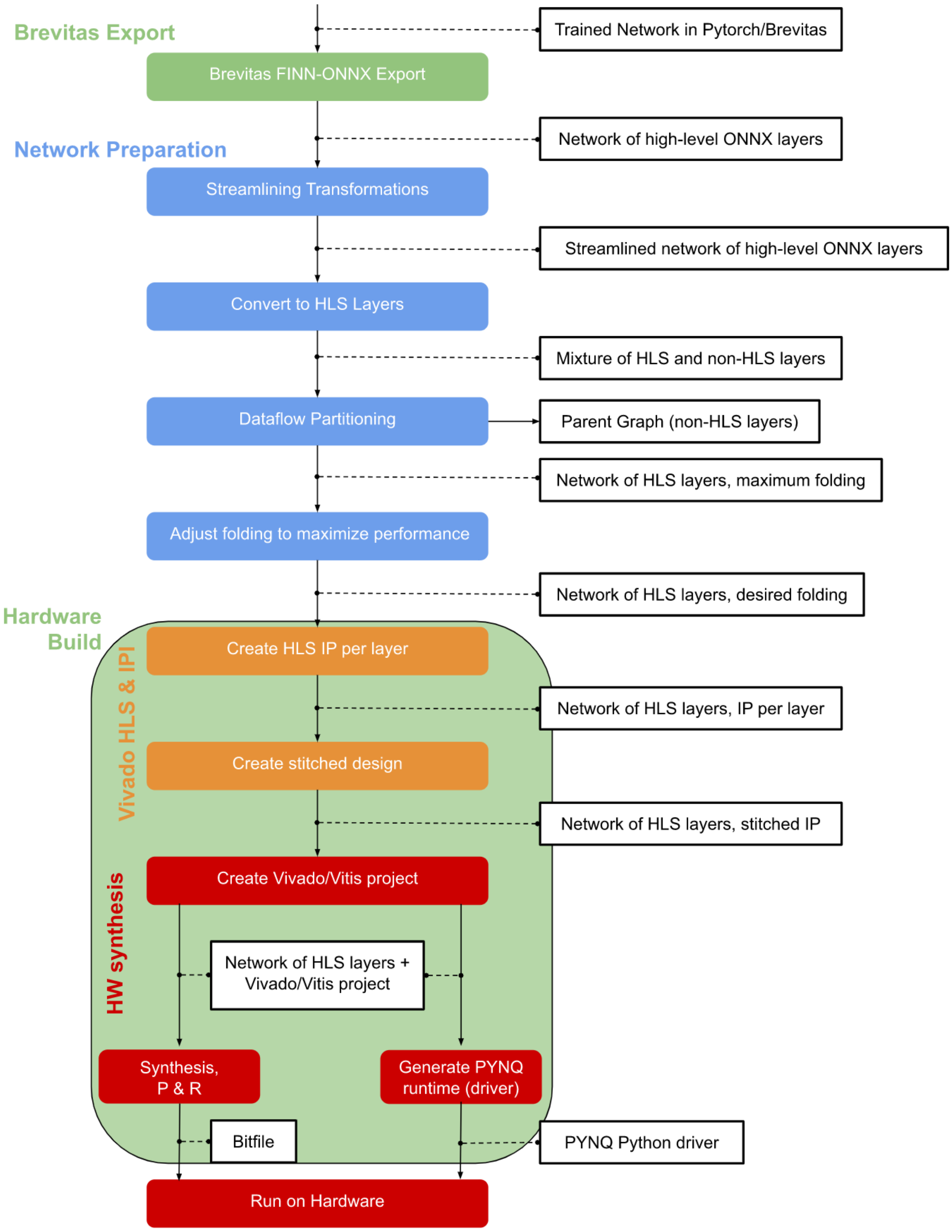
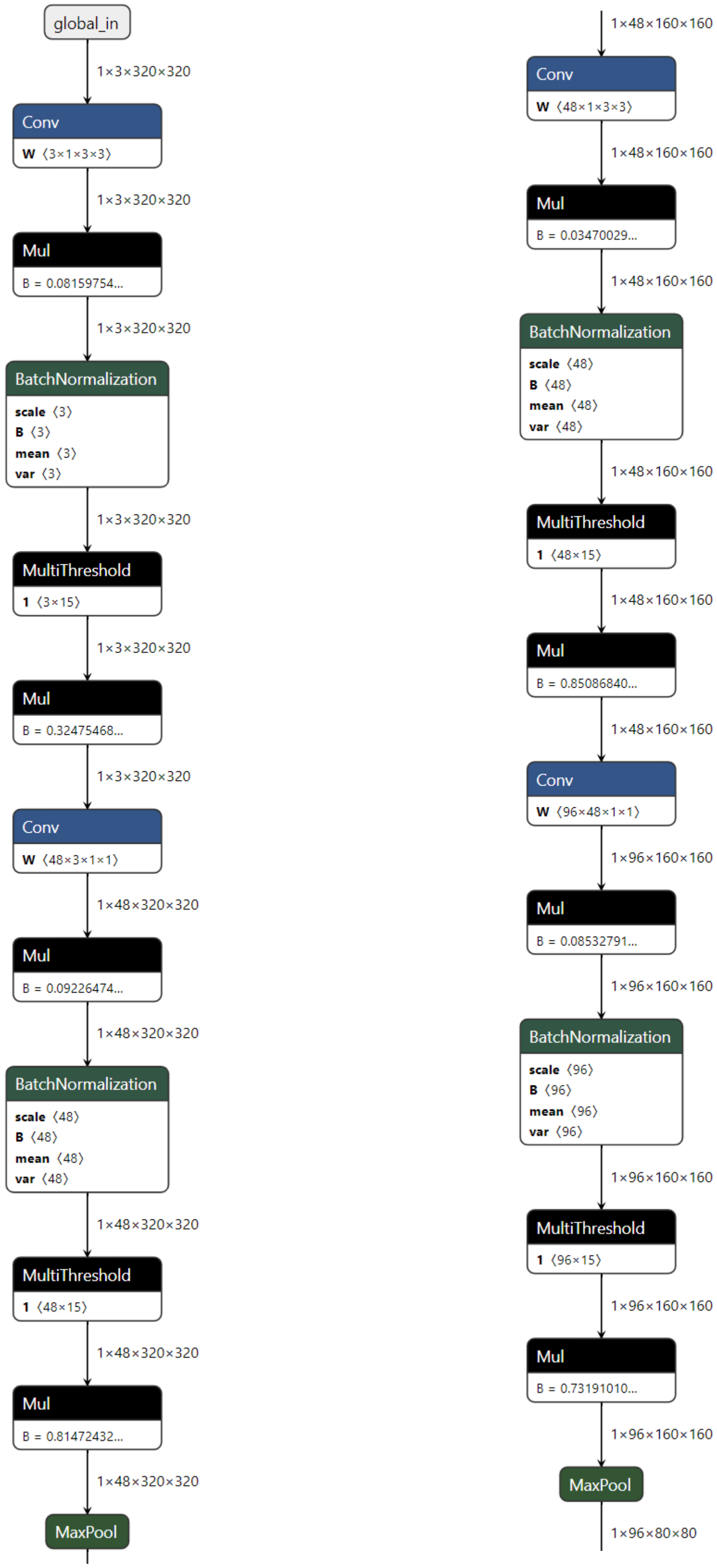
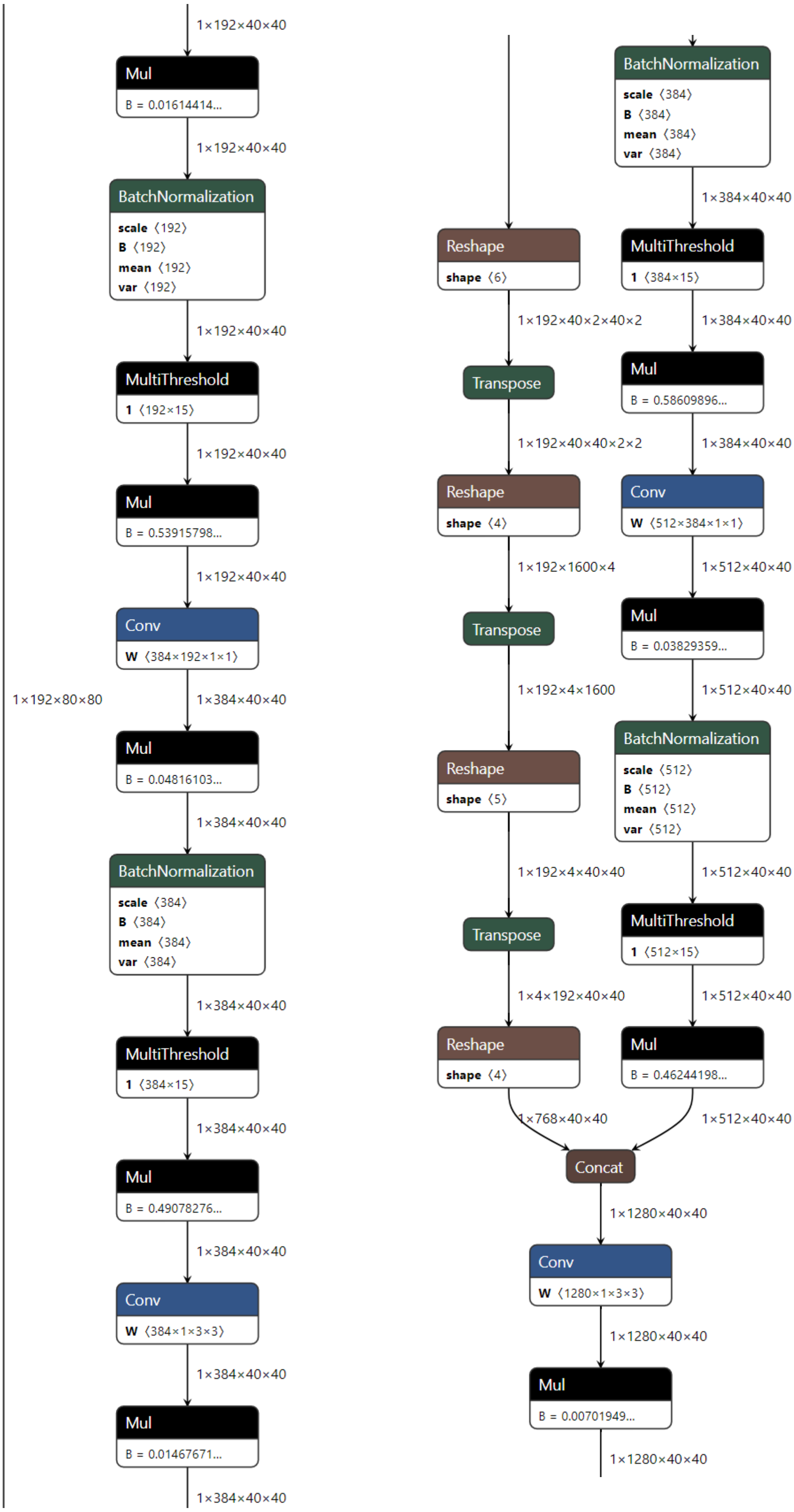


Figure 4.1: FINN standard design flow.


global_in
1×3×320×320
Conv
W ⟨3×1×3×3⟩
1×3×320×320
Mul
B = 0.08159754...
1×3×320×320
BatchNormalization
scale ⟨3⟩
B ⟨3⟩
mean ⟨3⟩
var ⟨3⟩
1×3×320×320
MultiThreshold
1 ⟨3×15⟩
1×3×320×320
Mul
B = 0.32475468...
1×3×320×320
Conv
W ⟨48×3×1×1⟩
1×48×320×320
Mul
B = 0.09226474...
1×48×320×320
BatchNormalization
scale ⟨48⟩
B ⟨48⟩
mean ⟨48⟩
var ⟨48⟩
1×48×320×320
MultiThreshold
1 ⟨48×15⟩
1×48×320×320
Mul
B = 0.81472432...
1×48×320×320
MaxPool
1×48×160×160
Conv
W ⟨48×1×3×3⟩
1×48×160×160
Mul
B = 0.03470029...
1×48×160×160
BatchNormalization
scale ⟨48⟩
B ⟨48⟩
mean ⟨48⟩
var ⟨48⟩
1×48×160×160
MultiThreshold
1 ⟨48×15⟩
1×48×160×160
Mul
B = 0.85086840...
1×48×160×160
Conv
W ⟨96×48×1×1⟩
1×96×160×160
Mul
B = 0.08532791...
1×96×160×160
BatchNormalization
scale ⟨96⟩
B ⟨96⟩
mean ⟨96⟩
var ⟨96⟩
1×96×160×160
MultiThreshold
1 ⟨96×15⟩
1×96×160×160
Mul
B = 0.73191010...
1×96×160×160
MaxPool
1×96×80×80


1×192×40×40
Mul
B = 0.01614414...
1×192×40×40
BatchNormalization
scale ⟨192⟩
B ⟨192⟩
mean ⟨192⟩
var ⟨192⟩
1×192×40×40
MultiThreshold
1 ⟨192×15⟩
1×192×40×40
Mul
B = 0.53915798...
1×192×40×40
Conv
W ⟨384×192×1×1⟩
1×192×80×80
1×384×40×40
Mul
B = 0.04816103...
1×384×40×40
BatchNormalization
scale ⟨384⟩
B ⟨384⟩
mean ⟨384⟩
var ⟨384⟩
1×384×40×40
MultiThreshold
1 ⟨384×15⟩
1×384×40×40
Mul
B = 0.49078276...
1×384×40×40
Conv
W ⟨384×1×3×3⟩
1×384×40×40
Mul
B = 0.01467671...
1×384×40×40
BatchNormalization
scale ⟨384⟩
B ⟨384⟩
mean ⟨384⟩
var ⟨384⟩
1×384×40×40
Reshape
shape ⟨6⟩
MultiThreshold
1 ⟨384×15⟩
1×192×40×2×40×2
1×384×40×40
Transpose
Mul
B = 0.58609896...
1×192×40×40×2×2
1×384×40×40
Reshape
shape ⟨4⟩
Conv
W ⟨512×384×1×1⟩
1×192×1600×4
1×512×40×40
Transpose
Mul
B = 0.03829359...
1×192×4×1600
1×512×40×40
Reshape
shape ⟨5⟩
BatchNormalization
scale ⟨512⟩
B ⟨512⟩
mean ⟨512⟩
var ⟨512⟩
1×192×4×40×40
1×512×40×40
Transpose
MultiThreshold
1 ⟨512×15⟩
1×4×192×40×40
1×512×40×40
Reshape
shape ⟨4⟩
Mul
B = 0.46244198...
1×768×40×40
1×512×40×40
Concat
1×1280×40×40
Conv
W ⟨1280×1×3×3⟩
1×1280×40×40
Mul
B = 0.00701949...
1×1280×40×40

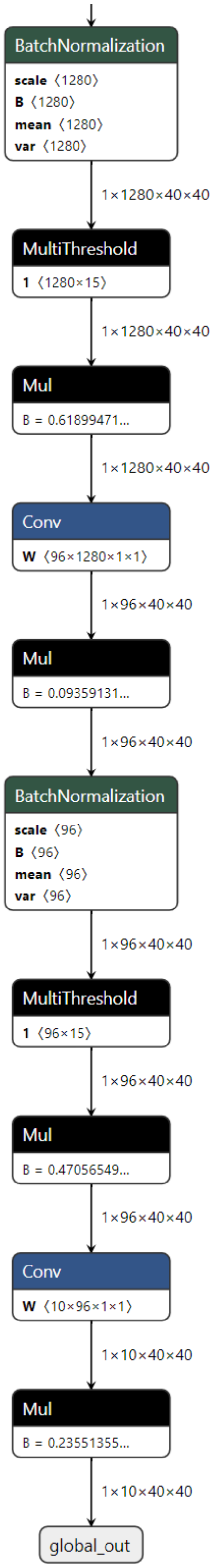


Figure 4.4: SkyNet quantized ONNX model displayed using *Netron*. The model has been split into 6 parts due to its huge dimension, it has to be read from top to bottom starting from left and going to right.

The `Multihreshold` node followed by the `Mul` node represent the `QuantRelu` layer. Actually, FINN goal is to reduce floating point values as much as possible thus, the `QuantRelu` layer is converted into a `Multithreshold` layer, in such a way that the input is no more simply filtered (as described in 1.3), but depending on its value it is converted to a given threshold [7]. When the model is exported running `export_finn_onnx` the scale factor of `QuantReLU` and its bit_width N are used in order to compute the thresholds:

$$step = scale\_factor \tag{4.1}$$

$$min\_th = \frac{step}{2} \tag{4.2}$$

$$num\_th = 2^N - 1 \tag{4.3}$$

where *step* is the threshold size, and it is constant for each thresholds, *min_th* is the first value of the threshold, namely the minimum value, and *num_th* is the number of thresholds.
In the SkyNetQuant model, since the bit_width has been set to 4 for the `QuantReLU` layer, the number of threshold computed is 15. In Figure 4.5, the thresholds of the first MultiThreshold node are displayed.

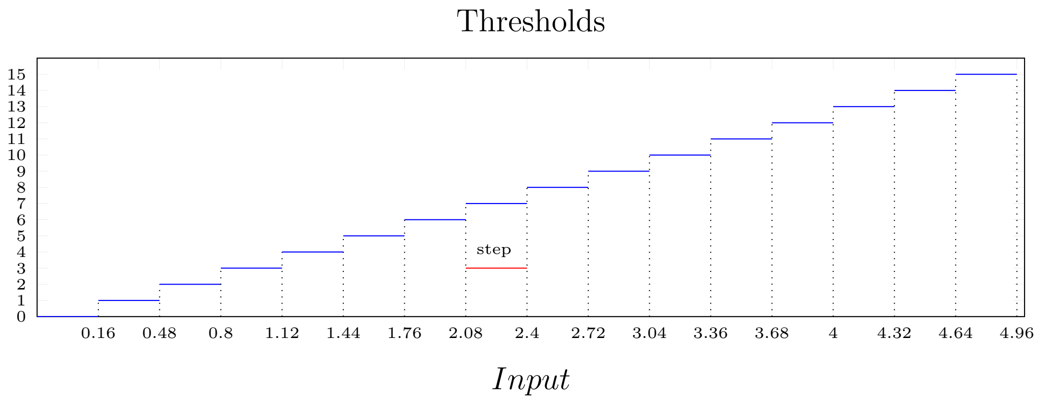


Figure 4.5: The fifteen thresholds adopted by the first MultiThreshold node in the SkyNetQuant model.

4.2 Network Transformation and Streamlining

After the export, the ONNX model has to be optimized in order to be synthesizable by FINN framework. In order to do that the ONNX model is transformed by executing function that are called *Transformation* and that can be classified in three main categories:

- **General**: are transformations used to assign names to nodes or to infer shapes to nodes' input.

- **Streamlined**: are the ones that impact more on the graph. They are used to collapse nodes together and to reorganize the graph's structure. In particular, the `Streamline()` transformation is a collection of several streamline transformations that the user can use to optimize the graph without the need of searching for the right transformation.
- **HLS**: given a ready to be converted graph, they are used to convert nodes of the ONNX model into HLS nodes, that can be mapped to the `finn-hls` library, considering some constraints.

4.2.1 General Transformation

The first transformations after the network export are the ones that simply *tidy-up* the ONNX model. Actually, those kind of transformation are called *Tidy Up Transformation*: they give unique node names to the graph, assign input tensor dimension to the nodes and readable tensor names to every node parameters. The following code is the one that has been used in order to get the graph of Figure 4.4.

```
20 #Importing General Transformation classes
21 from finn.core.modelwrapper import ModelWrapper
22 from finn.transformation.infer_shapes import InferShapes
23 from finn.transformation.fold_constants import FoldConstants
24 from finn.transformation.general import GiveReadableTensorNames,
       GiveUniqueNodeNames, RemoveStaticGraphInputs
25 from finn.transformation.infer_datatypes import InferDataTypes
26
27 #Loading the exported ONNX model
28 model=ModelWrapper(dir+"quantskynet_brevitas_export.onnx")
29
30 #Simple tranformations on the network
31 model = model.transform(InferShapes())
32 model = model.transform(InferDataTypes())
33 model = model.transform(FoldConstants())
34 model = model.transform(GiveUniqueNodeNames())
35 model = model.transform(GiveReadableTensorNames())
36 model = model.transform(RemoveStaticGraphInputs())
37 model.save(dir+"quantskynet_tidy.onnx")
```

At line 9, the class `ModelWrapper` is used to load the just extracted ONNX model: it is implemented by FINN, and, beside being used to load and save model (line 18), it allows the ONNX model to be transformed, plus it has some useful function that allow to rename, modify, delete nodes and much more.

4.2.2 Streamlining Transformation

Then, the *Streamline transformations* are applied in order to reduce the model as much as possible and make every nodes suitable for HLS node conversion.
Actually, FINN HLS conversion function supports only these type of nodes:

- Add

- Mul
- MultiThreshold
- MatMul
- Im2Col
- MaxPoolNWHC

thus the user has to apply transformations on the graph to obtain a ONNX model where only these kind of nodes are present, otherwise no conversion will be performed. Beside that, each input of these node *must be integer*, which is the only datatype that FINN HLS node conversion supports.

Replacing Convolutional Layers: the LowerConvsToMatMul Transformation

First of all, if the ONNX model presents Conv nodes, they have to be replaced using the `LowerConvsToMatMul` transformation. This transformation is one of the most relevant from the hardware point of view, since it is strictly related on how finn-hls library performs the convolution.

When executing `LowerConvsToMatMul`, FINN searches in the model for Conv nodes and replace them with a pair of Im2Col→MatMul nodes, in case of *depthwise* convolution (which can be asserted checking that the number of tensor's input channels is equal to the number of tensor's output channels), or a single MatMul node, in case of *point-wise* convolution (which can be asserted checking that the number of tensor's input channels is *not* equal to the number of tensor's output channels). As explained in Section 1.1, when performing the convolution a sliding window of size K×K (where K is the kernel dimension) highlights a K×K section of the feature map a time and performs the convolution: in hardware this procedure is lowered to a matrix by matrix multiplication.

In case of *depthwise* convolution, the input tensor is reshaped in a matrix of dimension $K^2 \cdot C \times N$, as showed in Figure 4.6. This reshaping is performed in FINN by the Im2Col node which, given a feature map of size $H \times W \times C$, returns a matrix whose structure is given by different columns which are made of the $K^2 \cdot C$ parameters highlighted by the sliding window. The number output columns N is given by:

$$N = nH \times nW \tag{4.4}$$

where

$$nH = \frac{H - 2 \times P - K}{S} + 1 \tag{4.5}$$

$$nW = \frac{W - 2 \times P - K}{S} + 1 \tag{4.6}$$

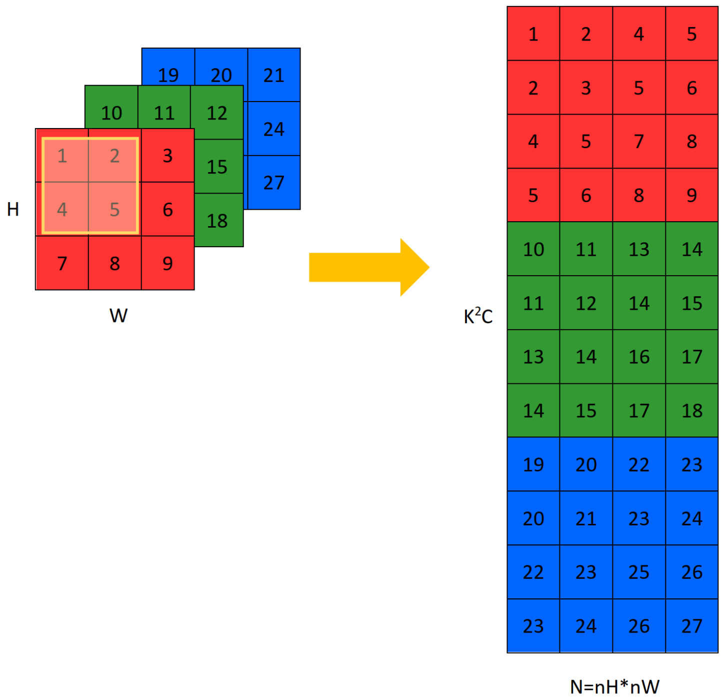


Figure 4.6: The picture describes how Im2Col creates the global feature map matrix that will be convoluted with the filter matrix.

and S, P are respectively the stride and the padding.
Then the convolution is performed by the MatMul node wich multiplies the output of Im2Col by the filter. The two nodes are displyed in Figure 4.7.

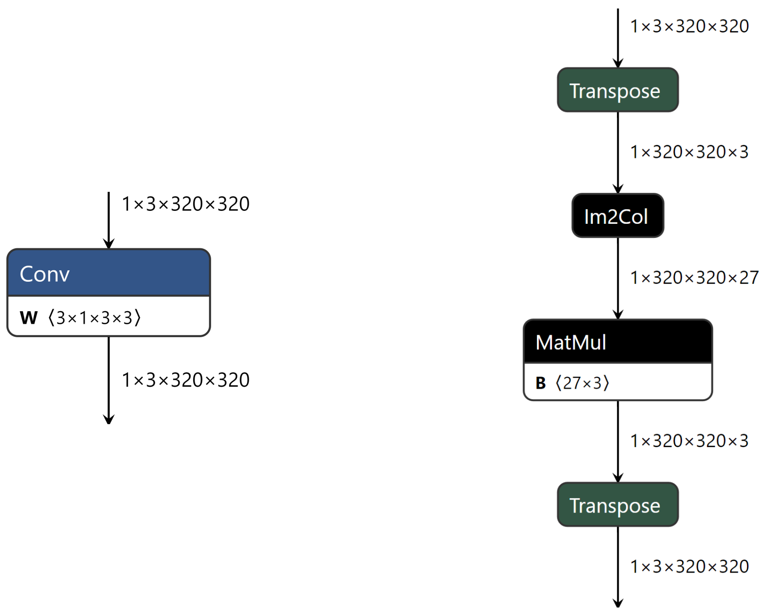


Figure 4.7: On the left, the Conv node, on the right its replacement performed when running `LowerConvsToMatMul`

Notice that two Transpose nodes have been inserted at the input and at the output: this is due to the fact that both Im2Col and MatMul operates on NHWC format, while in this case the input tensor of the Conv is on NCHW format. Thus, the input and the output are transposed to maintain the original shapes. Also notice that the weight matrix, whose dimension in Conv is $3 \times 1 \times 3 \times 3$ is reshaped to 27×3: this is done by `LowerConvsToMatMul` when inferring the MatMul node in order to make the matrix multiplication with the $1 \times 320 \times 320 \times 27$ output tensor of Im2Col possible, due to the fact that in matrix multiplication the number of columns of the first matrix must be equal to the number of rows of the second matrix.

In case of *pointwise* convolution, no Im2Col layer is needed, as showed in Figure 4.8. In this case `LowerConvsToMatMul()`, simply replaces the Conv node with the MatMul node, reshaping the $48 \times 3 \times 1 \times 1$ weight matrix to 3×48, and adding two Transpose nodes at the input and at the output. Again this operation is done in order to allow the matrix multiplication.

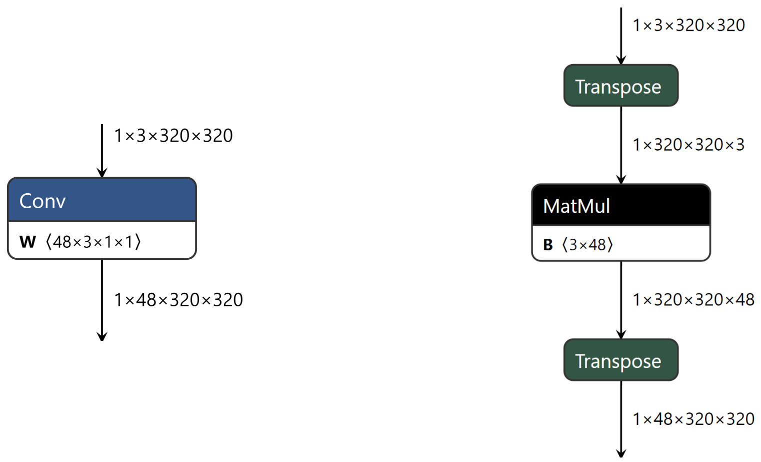


Figure 4.8: On the left, the Conv node, on the right its replacement performed when running `LowerConvsToMatMul()`

4.2.3 Optimizing the model: the Streamline Transformation

FINN has a already a built-in class called `Streamline()` that can be used to optimize the model and to remove the non-convertible to HLS nodes. Its code is here reported:

```
class Streamline(Transformation):
    """Apply the streamlining transform, see arXiv:1709.04060."""

    def apply(self, model):
        streamline_transformations = [
            ConvertSubToAdd(),
            ConvertDivToMul(),
```

```
            BatchNormToAffine(),
            ConvertSignToThres(),
            AbsorbSignBiasIntoMultiThreshold(),
            MoveAddPastMul(),
            MoveScalarAddPastMatMul(),
            MoveAddPastConv(),
            MoveScalarMulPastMatMul(),
            MoveScalarMulPastConv(),
            MoveAddPastMul(),
            CollapseRepeatedAdd(),
            CollapseRepeatedMul(),
            AbsorbAddIntoMultiThreshold(),
            FactorOutMulSignMagnitude(),
            AbsorbMulIntoMultiThreshold(),
            Absorb1BitMulIntoMatMul(),
            Absorb1BitMulIntoConv(),
            RoundAndClipThresholds(),
        ]
        for trn in streamline_transformations:
            model = model.transform(trn)
            model = model.transform(RemoveIdentityOps())
            model = model.transform(GiveUniqueNodeNames())
            model = model.transform(GiveReadableTensorNames())
            model = model.transform(InferDataTypes())
        return (model, False)
```

As it can be notice, `Streamline()` is made of different transformations:

- `ConvertSubToAdd()`: this transformation detects Sub node in the graph and converts them to Add node, since it is true that $A - B = A + (-B)$. This conversion is made in order to have only Add node in the model, so that they can be collapsed together (executing `CollapseRepeatedAdd()`) or absorbed into MultiThreshold nodes (executing `AbsorbAddIntoMultiThreshold()`).

- `ConvertDivToMul()`: this transformation detects Div nodes in the graph and converts them to Mul nodes, since it is true that $\frac{A}{B} = A \cdot (\frac{1}{B})$. As in the previous case, this transformation allows to have only Mul nodes, so that they can be collapsed together (executing `CollapseRepeatedMul()`) or absorbed into MultiThreshold nodes (executing `AbsorbMulIntoMultiThreshold()`).

- `BatchNormToAffine()`: this transformation detects BatchNormalization nodes and converts them in and Add→Sub nodes, as shown in Figure 4.9: Actually, PyTorch BatchNormalization layer output is given by:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} \star \gamma + \beta \tag{4.7}$$

where x is the input tensor, $\mathrm{E}[x]$ is its mean and $\mathrm{Var}[x]$ is its standard deviation; γ and β are respectively the scale and the bias, learnable parameters

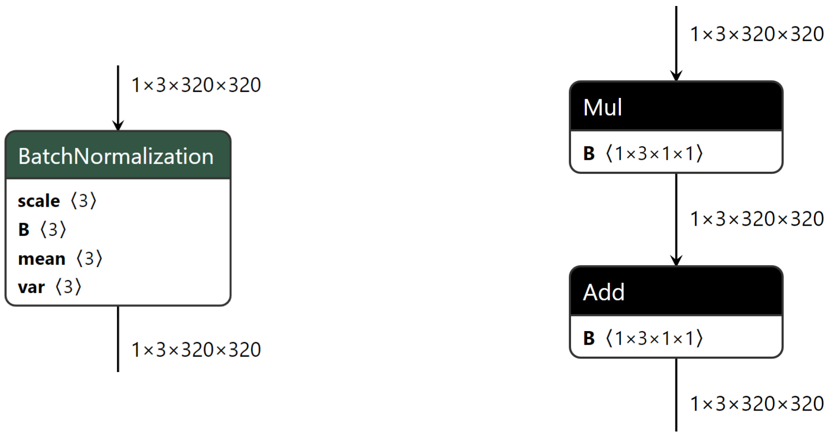


Figure 4.9: On the left, the BatchNormalization node, on the right its replacement performed when running `BatchNormToAffine()`

updated during training. Thus assuming:

$$A = \frac{\gamma}{\sqrt{\mathrm{Var}[x] + \epsilon}} \tag{4.8}$$

$$B = \beta - A \cdot \mathrm{E}[x] \tag{4.9}$$

the BatchNormalization node can be replaced by a Mul node, which multiplies the input tensor x by A, and an Add node, which sums the Mul node output (xA) to B.

- `ConvertSignToThres()`: Convert Sign node instances to MultiThreshold with threshold at 0.

- `AbsorbSignBiasIntoMultiThreshold()`: this transformation searches in the model for two subsequent MultiThreshold→Add nodes and if the Add node performs a scalar addition, the scalar factor is summed to the thresholds of the MultiThreshold node, then the Add node is removed from the graph.

- `MoveAddPastMul()`, `MoveScalarAddPastMatMul()`, `MoveAddPastConv()`, `MoveScalarMulPastMatMul()`, `MoveScalarMulPastConv()`: these transformations search in the graph pair of subsequent Add→Mul, Add→MatMul, Add→Conv, Mul→MatMul, Mul→Conv respectively and swap them, thanks to the commutative property.

- `CollapseRepeatedAdd()`, `CollapseRepeatedMul()`: these transformations search in the graph for two subsequent Add→Add, Mul→Mul respectively and collapse them together, so that only one single Add node, or one single Mul node, is maintained in the graph.

- `FactorOutMulSignMagnitude()`: Splits multiply-by-constant nodes into two multiply-by-constant nodes, where the first node is a bipolar vector (of signs) and the second is a vector of magnitudes.

- `AbsorbMulIntoMultiThreshold()`: this transformation searches in the model for two subsequent Mul→MultiThreshold nodes and if Mul is a scalar positive value, it is absorbed into the MultiThreshold node, by updating the threshold values. Thus the Mul node is removed from the graph.

- `Absorb1BitMulIntoMatMul()`, `Absorb1BitMulIntoConv()`: these transformations search in the model for two subsequent MatMul→Mul, Conv→Mul nodes and if Mul is a 1 bit value, it is absorbed into the preceding matrix multiply or convolution node. Then, the Mul node is removed from the graph.

- `RoundAndClipThreshold()`: this transformation searches for MultiThreshold nodes in the graph and if their input datatype is integer, its thresholds values are rounded to the nearest integer. Then, if the input is unsigned, negative thresholds are set to zero.

Usually, applying `Streamline()` transformation is already enough for reducing network size and preparing it for the HLS convertion. In the case of SkyNetQuant, these transformations have not been enough, for reasons that are explained in the following.

SkyNetQuant Streamlining problems Even if `SkyNetQuant` has been transformed by using *Streamline tranformations*, it has not been possible to reach a model where every node is suitable for the finn-hls nodes library. In the following, a list of all the problems encountered during SkyNetQuant development is reported:

1. **Tensor's shape not supported**: Some nodes of the SkyNetQuant graph have an input or an output tensor shape which is not supported by the FINN library. Actually, FINN supports only tensor shapes of 4 dimensions, while as it can be noticed from Figure 4.10, some nodes of SkyNetQuant have a dimension of 5 or 6.
   This is a real problem in FINN: with these dimensions the compiler is not going to synthesize and implement the model. In order to solve this issue, a custom transformation, called `CollapseReshape()`, has been created and added to the FINN library (the code is reported in Appendix C.1). Basically, *CollapseReshape* searches in the graph the chain Reshape→Transpose→ Reshape (line 17-21) and gives the input edge of the first Reshape (`n.input[0]` in the code) to the ReorderBypass node (whose code is reported in Appendix C.3), plus the input size of the Transpose node and its output size (`first_reshape` and `second_reshape` in the code), which will be used by the new node to perform exactly the same operations. Thus, from a functionality point of view, the behavior is the same, but in this way the tensor lengths are hidden from the FINN compiler and no error messages occur.
   However, this issue has been partially solved, due to the fact that it is not possible to add the new ReorderBypass node to the finn-hls library and thus this node is not synthesized by FINN.

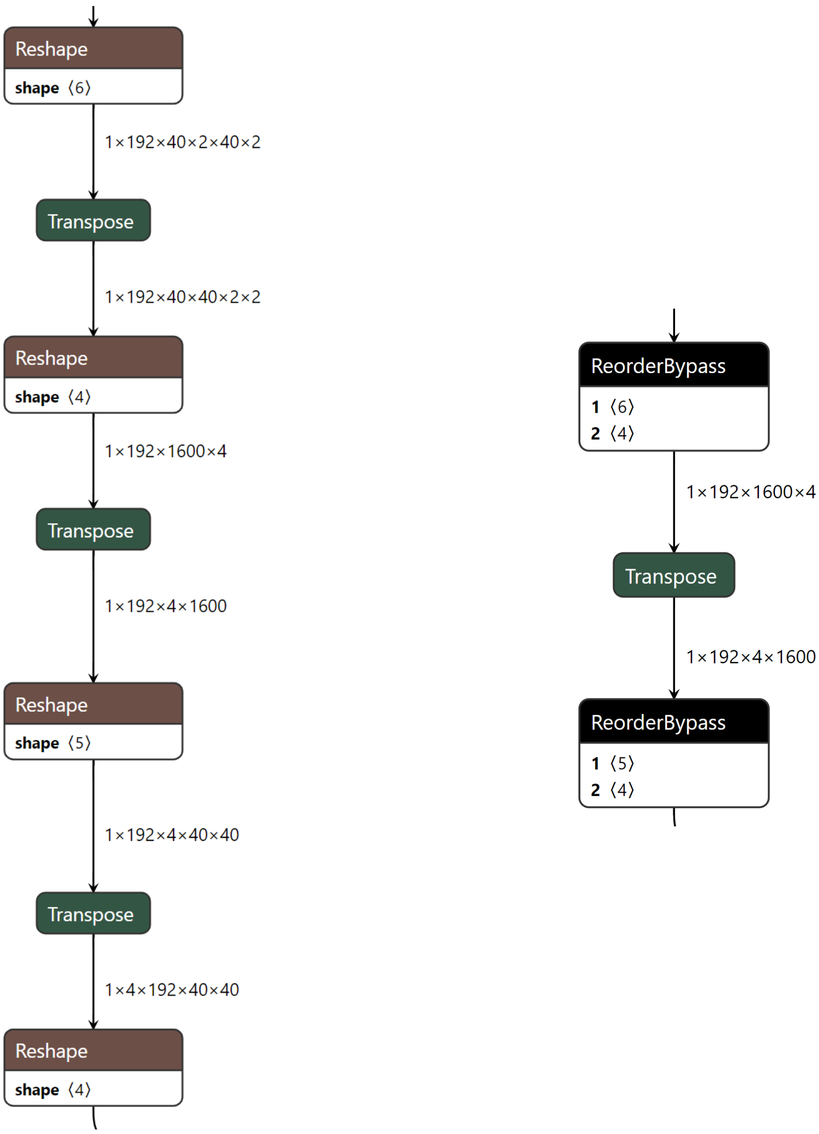


Figure 4.10: On the left, the model before the `CollapseReshape()` transformation, on the right the new model, with Reshape→Transpose→Reshape been substituted by ReorderBypass node by the transformation.

2. **Non-Integer input for finn-hls node**: In some cases nodes ready to be converted to hls-node cannot be converted, due to the fact that their input is not integer, but is floating point. This problem can be solved by going back to the Brevitas model and by adding some quantization layers (`QuantIdentity`), whose only purpose is to quantize the feature map among layers. In this case, `QuantIdentity` layers are added before every `QuantConv2d` layers since when exporting the model to FINN and after doing every kind of transformations, the Im2Col nodes have a non integer input which make them non-convertible to HLS nodes. The code of this new SkyNetQuant model is reported in Appendix C.4. In this case the quantization of the feature map is done on 8 bits.

Of course, the quantization performed on the FMs has made the accuracy drops, as stated in Section 2.1: after training, the highest IOU reached by this version of SkyNetQuant is 0.5563, which is more or less the 23.25% lower with respect to the previous version of SkyNetQuant, whose IOU is 0.7248.

3. **Presence of Transpose layers**: Due to the FINN transformations, in particular the *LowerConvsToMatMul*, plenty of Transpose nodes are inserted in the ONNX model and as it is noticed, they do not have a HLS implementation in the finn-hls library.
   In some cases, these Transpose nodes have been absorbed creating a custom transformation. Actually, as seen from Figure 4.11 using the custom `AddTranspose()` transformation, whose code is reported in Appendix C.2, it is possible to add two subsequent Transpose nodes, which do not affect the model behavior, in such a way that when running the *AbsorbTransposeIntoMultiThreshold* the structure Transpose→MultiThreshold→Transpose is detected and the Transpose nodes are collapsed into the MultiThreshold node.

Figure 4.11: Going from left to right, the model before the *AddTranspose* tranformation, then the model after the *AddTranspose* tranformation and finally the model after the *AbsorbTransposeIntoMultiThreshold* transformation.

The code used in order to prepare the model to synthesis is here reported:

```
#Importing classes for Streamlining Transformations
import finn.transformation.streamline.absorb as absorb
from finn.core.modelwrapper import ModelWrapper
from finn.transformation.infer_shapes import InferShapes
from finn.transformation.fold_constants import FoldConstants
from finn.transformation.general import GiveReadableTensorNames,
    GiveUniqueNodeNames, RemoveStaticGraphInputs
from finn.transformation.infer_data_layouts import InferDataLayouts
```

```
from finn.transformation.streamline import Streamline
from finn.transformation.lower_convs_to_matmul import LowerConvsToMatMul
from finn.transformation.general import RemoveUnusedTensors
from finn.transformation.streamline.reorder import MoveMaxPoolPastMultiThreshold,
    MakeMaxPoolNHWC, MoveScalarLinearPastInvariants, MoveScalarMulPastConv
from finn.transformation.addtranspose import AddTranspose
from finn.transformation.newreshape import NewReshape

#Loading the just tidy-up model..
model = ModelWrapper(dir+"quantskynet_tidy.onnx")


print("Running Streamline Tranformation...")
model = model.transform(MoveScalarLinearPastInvariants())
model = model.transform(Streamline())
model.save(dir+"quantskynet_streamlined.onnx")

print("Running LowerConvsToMatMul Tranformation...")
model = model.transform(LowerConvsToMatMul())
model.save(dir+"quantskynet_lower_convs.onnx")

#Further Tranformation are executed to optimize and make the model
#convertible to finn-hls library
model = model.transform(MoveMaxPoolPastMultiThreshold())
model = model.transform(MakeMaxPoolNHWC())
model = model.transform(absorb.AbsorbTransposeIntoMultiThreshold())
model = model.transform(absorb.AbsorbMulIntoMultiThreshold())
model = model.transform(AddTranspose())
model = model.transform(absorb.AbsorbTransposeIntoMultiThreshold())
model = model.transform(absorb.AbsorbScalarMulAddIntoTopK())
model = model.transform(GiveUniqueNodeNames())
model = model.transform(GiveReadableTensorNames())
model = model.transform(InferShapes())
model = model.transform(FoldConstants())
model = model.transform(GiveUniqueNodeNames())
model = model.transform(GiveReadableTensorNames())
model = model.transform(RemoveStaticGraphInputs())
model = model.transform(NewReshape())
model = model.transform(GiveUniqueNodeNames())
model = model.transform(GiveReadableTensorNames())
model = model.transform(InferDataLayouts())
model = model.transform(RemoveUnusedTensors())

model.save(dir+"quantskynet_ready_to_hls_conv.onnx")
```

4.2.4 HLS Transformations

After the network optimization, the node of the ONNX model are converted to the HLS node of the finn-hls library. Each node topology is converted by a specific transformation, the main ones are listed in Table 4.1.
In order to convert the ONNX nodes, the user has to go and look for the FINN file named “`convert_to_hls.py`” and to search for the transformation which better fits

its model, namely depending on the kind of nodes present in the model. Actually there is no a single global transformation that can be used to transform the model completely.
In the SkyNetQuant case the following HLS transformations have been executed:

```
#Importing classes for HLS conversions
from finn.transformation.move_reshape import RemoveCNVtoFCFlatten
from finn.custom_op.registry import getCustomOp
from finn.transformation.infer_data_layouts import InferDataLayouts
from finn.core.modelwrapper import ModelWrapper
from finn.core.datatype import DataType
from finn.transformation.streamline.reorder import MoveMaxPoolPastMultiThreshold,
    MakeMaxPoolNHWC, MoveScalarLinearPastInvariants, MoveScalarMulPastConv
import finn.transformation.fpgadataflow.convert_to_hls_layers as to_hls
from finn.transformation.streamline.round_thresholds import RoundAndClipThresholds

#Loading the streamlined model..
model=ModelWrapper(dir+"quantskynet_ready_to_hls_conv.onnx")

#Running HLS conversions..
model = model.transform(to_hls.InferQuantizedStreamingFCLayer())
model = model.transform(to_hls.InferConvInpGen())
model = model.transform(to_hls.InferStreamingMaxPool())
model = model.transform(to_hls.InferVVAU())
model = model.transform(to_hls.InferChannelwiseLinearLayer())
model = model.transform(RoundAndClipThresholds())
model = model.transform(to_hls.InferThresholdingLayer())
model = model.transform(absorb.AbsorbConsecutiveTransposes())
model.save(dir+"quantskynet_pre_dataflow_partition.onnx")
```

ONNX NODE	TRANSFORMATION	OUTPUT HLS NODE
Im2Col	InferConvInpGen()	ConvolutionInputGenerator
MaxPoolNHWC	InferStreamingMaxPool()	StreamingMaxPool
XnorPopcountMatMul ↓ MultiThreshold	InferBinaryStreamingFCLayer()	StreamingFCLayer_Batch
MatMul ↓ MultiThreshold	InferQuantizedStreamingFCLayer()	StreamingFCLayer_Batch
MatMul ↓ MultiThreshold	InferVVAU()	Vector_Vector_Activate_Batch
MultiThreshold	InferThresholdingLayer()	Thresholding_Batch
Add	InferAddStreamsLayer()	AddStreams

Table 4.1: The Table reports the HLS transformations that have to be applied to convert the ONNX nodes to HLS nodes.

4.3 Synthesis and Implementation on FPGA

After converting the nodes to finn-hls nodes, the model can be finally synthesized and implemented on a target FPGA.
In order to make sure that every node of the graph is synthesizable, namely that every node belongs to the HLS node class, the `CreateDataFlowPartiton()` transformation should be executed on the final model: this transformation is used in order to separate the HLS nodes from the NON-HLS nodes.
Actually, `CreateDataFlowPartiton()` searches in the graph for chains of fpga-dataflow nodes, namely HLS nodes, and NON-HLS nodes and returns two different ONNX models: one made of only HLS nodes, called *Child* model, the other made of NON-HLS nodes, called *Parent* model.
As it can be noticed from Figure 4.12, the ONNX model is cut by the `CreateDataFlowPartiton()` transformation and the connection among the two graphs is given by a new node called StreamingDataflowPartion, which contains the path to the *Child* model that is called by the *Parent* model when executing the network.

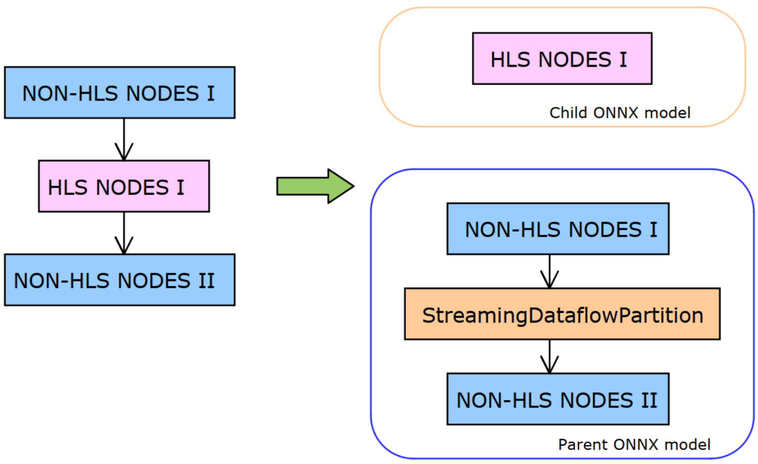


Figure 4.12: The creation of the parent model and child model done by the `CreateDataFlowPartition()` transformation.

In order to make the transformation succeed, every HLS node of the graph should be connected together and should not be interleaved by NON-HLS nodes. Actually, as seen from Figure 4.12, the best case is the one where there are two NON-HLS nodes chains interleaved by a single HLS-NODE chain. In this case the *Child* model is made of the only HLS-NODE chain, while the *Parent* model is made of the two NON-HLS node chains connected by the StreamingDataflowPartition node.
It is important to notice that when running the synthesis and implementation FINN will focus only on the *Child* model and the *Parent* model will be left unsynthetized: thus the real best case is the one where an unique chains of HLS nodes is present in the model, so that the user will have the complete model synthesis as output product.

If the final graph has got a structure as the one represented in Figure 4.13, the `CreateDataflowPartition()` will not succeed. Actually, since there are more than

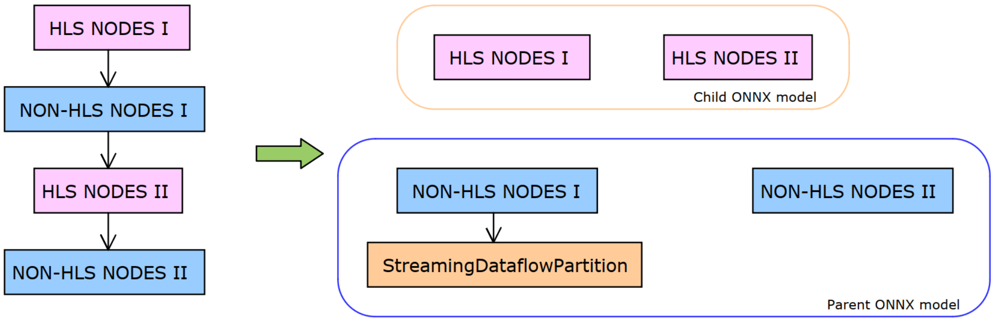


Figure 4.13: The broken models returned by the `CreateDataflowPartition()` when the initial model is made of multiple HLS-NODE chains.

one single chain of HLS nodes, the transformation will return both the chains in the *Child* model and the *Parent* model will be broken, due to the fact that only one single StreamingDataflowPartition node is instantiated by FINN.
In this case, if the user tries to synthesize the *Child* model, the synthesis will fail, because FINN has not been developed to synthetize multiple chains yet.
In case of SkyNetQuant, `CreateDataflowPartition` returned a *Child* model made of three chains (see Figure 4.14). In this case, if this model is synthesize, FINN will start creating and running Vivado HLS bash files that will never return.

In order to solve this issue, the only possibility is to go back to the model after the streamlining transformation and to limit the number of ONNX node converted to HLS node in such a way that when running the `CreateDataflowPartition` transformation the *Child* model will have only one single chain of HLS nodes. In the particular case of SkyNetQuant only the first chain (the bigger one) has been kept in order to be returned in the *Child* model.

Before going deeply on the synthesis and implementation part, it is important to highlight again that FINN should be used only if the user manage to convert all the node to HLS, because it is the only possibility to synthesize the entire network.

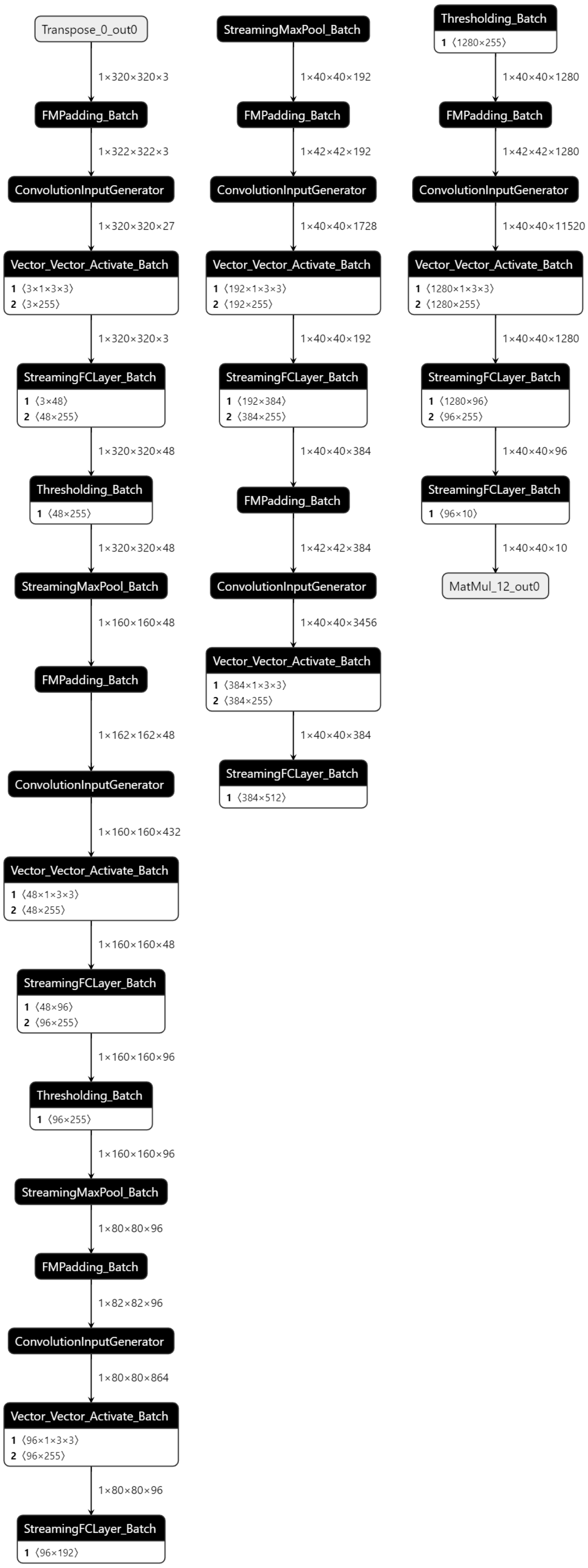


Figure 4.14: Child model returned for SkyNetQuant.

4.3.1 Synthesis and Implementation: the `ZynqBuild` Transformation

Again, to synthesize and implement the *Child* model, the user has to execute a transformation, in this case the `ZynqBuild` transformation. In the case of SkyNetQuant the following script has been adopted:

```
1  from finn.core.modelwrapper import ModelWrapper
2  from finn.transformation.fpgadataflow.make_zynq_proj import ZynqBuild
3  from finn.util.basic import pynq_part_map
4  
5  #Selecting the target board
6  pynq_board="ZCU104"
7  fpga_part=pynq_part_map[pynq_board]
8  #Setting the desired clock period
9  target_clk_ns=10
10 
11 model = ModelWrapper("skynet_child_model.onnx")
12 model = model.transform(ZynqBuild(platform = pynq_board, period_ns =
       target_clk_ns))
```

As it can be notice from the reported code, the user simply has to declare the target board (line 6) and the target clock period (line 9). In this case, SkyNetQuant has been inferred on the Zynq ZCU104 board with a target clock period of 10 *ns*.

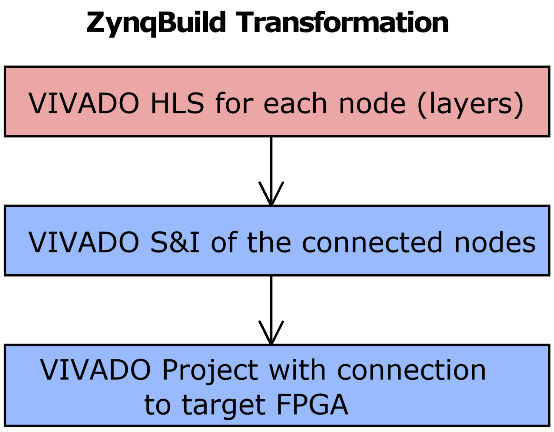


Figure 4.15: Synthesis and implementation steps of FINN.

The inference on FPGA is performed by FINN in three main steps:

- **Synthesis of every node**: FINN synthesizes separately every HLS nodes using *Vivado HLS* and storing the HLS results into different folders, one for every node. The synthesis is done by running a script which is created by FINN.
- **Synthesis of the full network**: FINN synthesizes the complete network by connecting every HLS node synthesis together.
- **Inference on FPGA**: FINN creates a *Vivado Design Suite* project were

the IP of the synthesized network is connected to the target FPGA; then the *bitstream* file is generated.

The results of the implemented child model of SkyNetQuant are reported in Table 4.2, where the quantized model is compared with the implementation of original SkyNet model, both with a target clock frequency $f_{CLK} = 100.00$ MHz.

Resource Type	**SkyNet**	**SkyNetQuant**
CLB	52266	178081
BRAM	209	40
DSP	360	6

Table 4.2: Comparison among the resource usage of the original SkyNet architecture with the SkyNetQuant architecture synthesized using FINN.

As expected the BRAM resource usage is decreased with respect to the original SkyNet: this is due to the fact that SkyNet has been sinthetized with fixed point weight on 9 bits, while SkyNetQuant's weights require only 4 bits each. However, it has to be considered that the results of SkyNetQuant are related to just the half of the entire architecture, thus they are expected to be doubled in case of complete synthesis.
On the contrary, the CLB is three time bigger with respect to the original implementation: this could be related on how FINN implements convolutions and activation function.
Another point that has to be highlighted is that SkyNetQuant besides storing the weights, also needs to store the thresholds related to the `MultiThreshold` nodes, that are automatically inferred in place of the `QuantReLU` and the `QuantIdentity` layers. Thus, the 40 BRAM are used to store both weights and thresholds.

Chapter 5

Conclusions

Due to the fact that both Brevitas and FINN are extremely new and still under development, it was impossible to complete the SkyNetQuant implementation.

Regarding Brevitas, the results in term of accuracy are extremely good. Also, once understood how it works, it is really easy to use on already existing PyTorch models and it is fully customizable by the users. The possibility to define a specific kind of quantizer and to mix quantized layer with standard one, allows users to explore any kind of model and to select the best one depending on their needs.

Concerning FINN, at the moment it could be used only with very small network with standard structure: the presence of the *bypass and reordering branch*, used to increase the ability to detect small objects, has made the SkyNet and SkyNetQuant models' structure not standard. In particular, the `Streamline()` transformation function works perfectly for one single chain model, namely without fork nodes as SkyNetQuant, since it manages to collapse and reorder nodes in such a way that every node has got integer input and can be converted to finn-hls library. In this case, the model structure did not allow the `Streamline()` to reach this scope. Also, even if the model has changed by adding quantization layers for intermediate FMs, the model structure still be too particular to be synthesized with FINN.
Another problem has been the adding of the `Transpose` node when executing the `LowerConvsToMatMul()` transformation: this node is created automatically by FINN even if not present in the original CNN and, since it is not present into finn-hls library, it results in a non-implementable network if it cannot be absorbed back into some other layer.
Finally, the fact that the *Parent* model is left unsynthesized is a real problem, since the user cannot reach the complete network implementation. Then, last but not least, the documentation related to FINN and Brevitas is extremely poor.

Since it has been impossible to complete the entire model synthesis, an *hypothetical* synthesis of the SkyNetQuant model has been carried out with *Vivado HLS* using the original C++ files of SkyNet and by setting the weights variable on 4 bits. Unfortunately, this is an *hypothetical* version of SkyNetQuant, since

the original HLS SkyNet implementation was too specific to be modified on time, thus no simulation has been carried out. The results, both for Ultra96v2 board, are displayed on Table 5.1.
Firstly, the original SkyNet has been synthesized with three different clock frequency (f_{CLK} = 115.39 MHz, f_{CLK} = 125.00 MHz, f_{CLK} = 136.37 MHz), by tuning the PLL of the Ultra96 board; then, in order to compare the results, SkyNetQuant has been synthesized with the same clock period. From Table 5.1, it could be notice that the maximum clock frequency reachable by SkyNet without negative slack is f_{CLK} = 125.00 MHz, while SkyNetQuant still have positive slack also with f_{CLK} = 136.37 MHz.

	SkyNet		SkyNetQuant	
f_{CLK} = 115.39 MHz				
WNS	0.470 ns		0.722 ns	
TNS	0		0	
Resources				
Type	Units	%	Units	%
CLB	52266	74.07 %	43814	62.09 %
BRAM	209	96.76 %	193	89.35 %
DSP	360	100.00 %	359	99.72%
Power				
Total Power	4027 W		3602 W	

	SkyNet		SkyNetQuant	
f_{CLK} = 125.00 MHz				
WNS	0.003ns		0.190ns	
TNS	0		0	
Resources				
Type	Units	%	Units	%
CLB	52303	74.13 %	43815	62.10 %
BRAM	209	96.76 %	193	89.35 %
DSP	360	100.00 %	359	99.72 %
Power				
Total Power	4216 W		3748 W	

	SkyNet		SkyNetQuant	
f_{CLK} = 136.37 MHz				
WNS	-0.108 ns		0.020 ns	
TNS	-8.616 ns		0	
Resources				
Type	Units	%	Units	%
CLB	52321	74.15 %	43859	62.16 %
BRAM	209	96.76 %	193	89.35 %
DSP	360	100.00 %	359	99.72 %
Power				
Total Power	4413 W		3900 W	

Table 5.1: Comparison among three different implementations results for SkyNet and SkyNetQuant. (WNS=Worst Negative Slack; TNS=Total Negative Slack).

Notice how the BRAM resource usage is reduced of a 7.41% factor, going from the

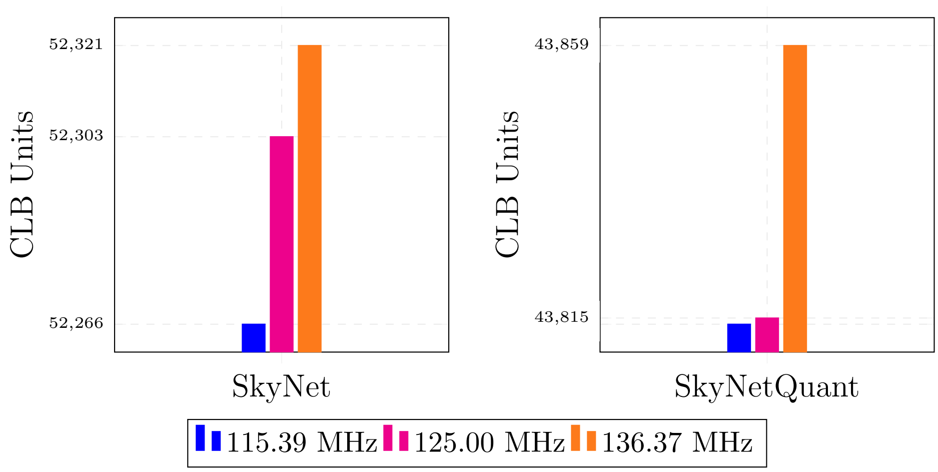


Figure 5.1: On the left, the SkyNet CLB resource usage, on the right, the SkyNetQuant CLB resource usage. Notice how both the architecture requires more CLB units as the frequency increases.

96.76% requested by SkyNet to 89.35% requested by SkyNetQuant. Of course, these values are constants for all the implementations, due to the fact that the amount of memory requested by SkyNet and SkyNetQuant is the same for every implementation.
On the contrary, the CLB usage increases as frequency increases for both the architecture (see the graphs of Figure 5.1). Notice that SkyNetQuant requires almost the 10% less of CLB units than SkyNet.
As the frequency increases, also the total power of the two architecture increases. Again SkyNetQuant requires less power than SkyNet.

In conclusion, FINN has to be further improved to be used for every kind of quantized network; at the moment it can be used just for restricted type of networks. On the contrary, Brevitas is already extremely powerful and easy to use. Thus, if FINN problems are fixed, these tools used together could be very useful for future developers.

Appendix A

Skynet Model PyTorch Code

```
from collections import OrderedDict
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init as init

from region_loss import RegionLoss
from utils import *
from collections import OrderedDict


class ReorgLayer(nn.Module):
    def __init__(self, stride=2):
        super(ReorgLayer, self).__init__()
        self.stride = stride
    def forward(self, x):
        stride = self.stride
        assert(x.data.dim() == 4)
        B = x.data.size(0)
        C = x.data.size(1)
        H = x.data.size(2)
        W = x.data.size(3)
        assert(H % stride == 0)
        assert(W % stride == 0)
        ws = stride
        hs = stride
        x = x.view([B, C, H//hs, hs, W//ws, ws]).transpose(3, 4).contiguous()
        x = x.view([B, C, H//hs*W//ws, hs*ws]).transpose(2, 3).contiguous()
        x = x.view([B, C, hs*ws, H//hs, W//ws]).transpose(1, 2).contiguous()
        x = x.view([B, hs*ws*C, H//hs, W//ws])
        return x


class SkyNet(nn.Module):
    def __init__(self):
        super(SkyNet, self).__init__()
        self.width = int(320)
        self.height = int(160)
        self.header = torch.IntTensor([0,0,0,0])
```

```
        self.seen = 0
        self.reorg = ReorgLayer(stride=2)

        def conv_bn(inp, oup, stride):
            return nn.Sequential(
                nn.Conv2d(inp, oup, 3, stride, 1, bias=False),
                nn.BatchNorm2d(oup),
                nn.ReLU(inplace=True)
                )

        def conv_dw(inp, oup, stride):
            return nn.Sequential(
                nn.Conv2d(inp, inp, 3, stride, 1, groups=inp, bias=False),
                nn.BatchNorm2d(inp),
                nn.ReLU6(inplace=True),

                nn.Conv2d(inp, oup, 1, 1, 0, bias=False),
                nn.BatchNorm2d(oup),
                nn.ReLU6(inplace=True),
                )

        self.model_p1 = nn.Sequential(
            conv_dw( 3, 48, 1), #dw1
            nn.MaxPool2d(kernel_size=2, stride=2),
            conv_dw( 48, 96, 1), #dw2
            nn.MaxPool2d(kernel_size=2, stride=2),
            conv_dw( 96, 192, 1), #dw3
            )

        self.model_p2 = nn.Sequential(
            nn.MaxPool2d(kernel_size=2, stride=2),
            conv_dw(192, 384, 1), #dw4
            conv_dw(384, 512, 1), #dw5
            )

        self.model_p3 = nn.Sequential( #cat dw3(ch:192 -> 768) and dw5(ch:512)
            conv_dw(1280, 96, 1),
            nn.Conv2d(96, 10, 1, 1,bias=False),
            )

        self.loss = RegionLoss([1.4940052559648322,
            2.3598481287086823,4.0113013115312155,5.760873975661669],2)
        self.anchors = self.loss.anchors
        self.num_anchors = self.loss.num_anchors
        self.anchor_step = self.loss.anchor_step
        self._initialize_weights()

    def forward(self, x):
        x_p1 = self.model_p1(x)
        x_p1_reorg = self.reorg(x_p1)
        x_p2 = self.model_p2(x_p1)
        x_p3_in = torch.cat([x_p1_reorg, x_p2], 1)
        x = self.model_p3(x_p3_in)
        return x
```

```
    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
             nn.init.kaiming_normal_(m.weight, mode='fan_out')
             if m.bias is not None:
             nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
             nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
             nn.init.normal_(m.weight, 0, 0.01)
             nn.init.constant_(m.bias, 0)

    from finn.core.onnx_exec import execute_onnx
    output_dict = execute_onnx(onnxmodel, input_dict, True)
```

Appendix B

Brevitas Library

B.1 Integer Quantizer Code Implementation

```
import torch
from torch import Tensor
from torch.nn import Module

import brevitas
from brevitas.function.ops import max_int, min_int
from brevitas.core.function_wrapper import RoundSte, TensorClamp
from brevitas.core.quant.delay import DelayWrapper


class IntQuant(brevitas.jit.ScriptModule):
    """
    ScriptModule that implements scale, shifted, uniform integer quantization of
        an input tensor,
    according to an input scale, zero-point and bit-width.

    Args:
        narrow_range (bool): Flag that determines whether restrict quantization to
            a narrow range or not.
        signed (bool): Flag that determines whether to quantize to a signed range
            or not.
        float_to_int_impl (Module): Module that performs the conversion from
            floating point to
            integer representation. Default: RoundSte()
        tensor_clamp_impl (Module): Module that performs clamping. Default:
            TensorClamp()
        quant_delay_steps (int): Number of training steps to delay quantization
            for. Default: 0

    Returns:
        Tensor: Quantized output in de-quantized format.

    Examples:
        >>> from brevitas.core.scaling import ConstScaling
        >>> int_quant = IntQuant(narrow_range=True, signed=True)
```

```
        >>> scale, zero_point, bit_width = torch.tensor(0.01), torch.tensor(0.),
            torch.tensor(4.)
        >>> inp = torch.Tensor([0.042, -0.053, 0.31, -0.44])
        >>> out = int_quant(scale, zero_point, bit_width, inp)
        >>> out
        tensor([ 0.0400, -0.0500, 0.0700, -0.0700])

    Note:
        Maps to quant_type == QuantType.INT == 'INT' == 'int' in higher-level APIs.

    Note:
        Set env variable BREVITAS_JIT=1 to enable TorchScript compilation of this
            module.
    """

    __constants__ = ['signed', 'narrow_range']

    def __init__(
            self,
            narrow_range: bool,
            signed: bool,
            float_to_int_impl: Module = RoundSte(),
            tensor_clamp_impl: Module = TensorClamp(),
            quant_delay_steps: int = 0):
        super(IntQuant, self).__init__()
        self.float_to_int_impl = float_to_int_impl
        self.tensor_clamp_impl = tensor_clamp_impl
        self.signed = signed
        self.narrow_range = narrow_range
        self.delay_wrapper = DelayWrapper(quant_delay_steps)

    @brevitas.jit.script_method_110_disabled
    def to_int(
            self,
            scale: Tensor,
            zero_point: Tensor,
            bit_width: Tensor,
            x: Tensor) -> Tensor:
        y = x / scale
        y = y + zero_point
        min_int_val = self.min_int(bit_width)
        max_int_val = self.max_int(bit_width)
        y = self.tensor_clamp_impl(y, min_val=min_int_val, max_val=max_int_val)
        y = self.float_to_int_impl(y)
        return y

    @brevitas.jit.script_method
    def min_int(self, bit_width):
        return min_int(self.signed, self.narrow_range, bit_width)

    @brevitas.jit.script_method
    def max_int(self, bit_width):
        return max_int(self.signed, self.narrow_range, bit_width)

    @brevitas.jit.script_method
```

```
    def forward(
            self,
            scale: Tensor,
            zero_point: Tensor,
            bit_width: Tensor,
            x: Tensor) -> Tensor:
        y_int = self.to_int(scale, zero_point, bit_width, x)
        y = y_int - zero_point
        y = y * scale
        y = self.delay_wrapper(x, y)
        return y
```

B.2 Binary Quantizer Code Implementation

```
from typing import Tuple

import torch
from torch import Tensor
from torch.nn import Module

import brevitas
from brevitas.function.ops import tensor_clamp
from brevitas.function.ops_ste import binary_sign_ste
from brevitas.core.bit_width import BitWidthConst
from brevitas.core.utils import StatelessBuffer
from brevitas.core.quant.delay import DelayWrapper


class BinaryQuant(brevitas.jit.ScriptModule):
    """
    ScriptModule that implements scaled uniform binary quantization of an input tensor.
    Quantization is performed with :func:`~brevitas.function.ops_ste.binary_sign_ste`.

    Args:
        scaling_impl (Module): Module that returns a scale factor.
        quant_delay_steps (int): Number of training steps to delay quantization for. Default: 0

    Returns:
        Tuple[Tensor, Tensor, Tensor, Tensor]: Quantized output in de-quantized format, scale,
            zero-point, bit_width.

    Examples:
        >>> from brevitas.core.scaling import ConstScaling
        >>> binary_quant = BinaryQuant(ConstScaling(0.1))
        >>> inp = torch.Tensor([0.04, -0.6, 3.3])
        >>> out, scale, zero_point, bit_width = binary_quant(inp)
        >>> out
        tensor([ 0.1000, -0.1000, 0.1000])
```

```
        >>> scale
        tensor(0.1000)
        >>> zero_point
        tensor(0.)
        >>> bit_width
        tensor(1.)

    Note:
        Maps to quant_type == QuantType.BINARY == 'BINARY' == 'binary' when applied
            to weights
         in higher-level APIs.

    Note:
        Set env variable BREVITAS_JIT=1 to enable TorchScript compilation of this
            module.
    """

    def __init__(self, scaling_impl: Module, quant_delay_steps: int = 0):
        super(BinaryQuant, self).__init__()
        self.scaling_impl = scaling_impl
        self.bit_width = BitWidthConst(1)
        self.zero_point = StatelessBuffer(torch.tensor(0.0))
        self.delay_wrapper = DelayWrapper(quant_delay_steps)

    @brevitas.jit.script_method
    def forward(self, x: Tensor) -> Tuple[Tensor, Tensor, Tensor, Tensor]:
        scale = self.scaling_impl(x)
        y = binary_sign_ste(x) * scale
        y = self.delay_wrapper(x, y)
        return y, scale, self.zero_point(), self.bit_width()
```

B.3 Ternary Quantizer Code Implementation

```
from typing import Tuple

import torch
from torch import Tensor
from torch.nn import Module

import brevitas
from brevitas.function.ops_ste import ternary_sign_ste
from brevitas.core.bit_width import BitWidthConst
from brevitas.core.utils import StatelessBuffer
from brevitas.core.quant.delay import DelayWrapper


class TernaryQuant(brevitas.jit.ScriptModule):
    """
    ScriptModule that implements scaled uniform ternary quantization of an input
        tensor.
    Quantization is performed with
        :func:`~brevitas.function.ops_ste.ternary_sign_ste`.
```

```

    Args:
        scaling_impl (Module): Module that returns a scale factor.
        threshold (float): Ternarization threshold w.r.t. to the scale factor.
        quant_delay_steps (int): Number of training steps to delay quantization for. Default: 0

    Returns:
        Tuple[Tensor, Tensor, Tensor, Tensor]: Quantized output in de-quantized format, scale,
            zero-point, bit_width.

    Examples:
        >>> from brevitas.core.scaling import ConstScaling
        >>> ternary_quant = TernaryQuant(ConstScaling(1.0), 0.5)
        >>> inp = torch.Tensor([0.04, -0.6, 3.3])
        >>> out, scale, zero_point, bit_width = ternary_quant(inp)
        >>> out
        tensor([ 0., -1., 1.])
        >>> scale
        tensor(1.)
        >>> zero_point
        tensor(0.)
        >>> bit_width
        tensor(2.)

    Note:
        Maps to quant_type == QuantType.TERNARY == 'TERNARY' == 'ternary' in higher-level APIs.

    Note:
        Set env variable BREVITAS_JIT=1 to enable TorchScript compilation of this module.
    """

    __constants__ = ['threshold']

    def __init__(self, scaling_impl: Module, threshold: float, quant_delay_steps: int = None):
        super(TernaryQuant, self).__init__()
        self.scaling_impl = scaling_impl
        self.threshold = threshold
        self.bit_width = BitWidthConst(2)
        self.zero_point = StatelessBuffer(torch.tensor(0.0))
        self.delay_wrapper = DelayWrapper(quant_delay_steps)

    @brevitas.jit.script_method
    def forward(self, x: Tensor) -> Tuple[Tensor, Tensor, Tensor, Tensor]:
        scale = self.scaling_impl(x)
        mask = x.abs().gt(self.threshold * scale)
        y = mask.float() * ternary_sign_ste(x)
        y = y * scale
        y = self.delay_wrapper(x, y)
        return y, scale, self.zero_point(), self.bit_width()
```

B.4 QuantTensor Code Implementation

```python
from abc import ABC
from typing import Optional, NamedTuple

import torch
from torch import Tensor

from brevitas.function.ops_ste import ceil_ste, round_ste
from brevitas.function.ops import max_int


class QuantTensor(NamedTuple):
    value: Tensor
    scale: Optional[Tensor] = None
    zero_point: Optional[Tensor] = None
    bit_width: Optional[Tensor] = None
    signed: Optional[bool] = None
    training: Optional[bool] = None

    @property
    def tensor(self):
        return self.value

    @property
    def is_valid(self):
        return self.value is not None \
               and self.scale is not None \
               and self.zero_point is not None \
               and self.bit_width is not None \
               and self.signed is not None

    def set(self, **kwargs):
        return self._replace(**kwargs)

    def detach_(self):
        self.value.detach_()
        self.scale.detach_()
        self.zero_point.detach_()
        self.bit_width.detach_()

    def detach(self):
        return QuantTensor(
            self.value.detach() if self.value is not None else None,
            self.scale.detach() if self.scale is not None else None,
            self.zero_point.detach() if self.zero_point is not None else None,
            self.bit_width.detach() if self.bit_width is not None else None,
            self.signed)

    def int(self, float_datatype=False):
        if self.is_valid:
            int_value = self.value / self.scale
            int_value = int_value + self.zero_point
            int_value = round_ste(int_value)
```

```
            if float_datatype:
                return int_value
            else:
                return int_value.int()
        else:
            raise RuntimeError(f"QuantTensor not well formed, all fields must be
                set: {self}")

    @staticmethod
    def check_input_type(other):
        if not isinstance(other, QuantTensor):
            raise RuntimeError("Other tensor is not a QuantTensor")

    def check_scaling_factors_same(self, other):
        if self.training is not None and self.training:
            return True
        if not torch.allclose(self.scale, other.scale):
            raise RuntimeError("Scaling factors are different")

    def check_zero_points_same(self, other):
        if self.training is not None and self.training:
            return True
        if not torch.allclose(self.zero_point, other.zero_point):
            raise RuntimeError("Zero points are different")

    def check_bit_width_same(self, other):
        if not torch.allclose(self.bit_width, other.bit_width):
            raise RuntimeError("Bit widths are different")

    def check_sign_same(self, other):
        if not self.signed == other.signed:
            raise RuntimeError("Signs are different")

    def view(self, *args, **kwargs):
        return self.set(value= self.value.view(*args, **kwargs))

    def reshape(self, *args, **kwargs):
        return self.set(value=self.value.reshape(*args, **kwargs))

    def flatten(self, *args, **kwargs):
        return self.set(value=self.value.flatten(*args, **kwargs))

    def size(self, *args, **kwargs):
        return self.value.size(*args, **kwargs)

    @property
    def shape(self):
        return self.value.shape

    def add(self, other):
        return self + other

    @staticmethod
    def cat(tensor_list, dim):
```

```python
        assert len(tensor_list) >= 2, 'Two or more tensors required for concatenation'
        first_qt = tensor_list[0]
        if all([qt.is_valid for qt in tensor_list]):
            for qt in tensor_list[1:]:
                QuantTensor.check_input_type(qt)
                first_qt.check_scaling_factors_same(qt)
                first_qt.check_scaling_factors_same(qt)
                first_qt.check_bit_width_same(qt)
                first_qt.check_sign_same(qt)
            output_value = torch.cat([qt.value for qt in tensor_list], dim=dim)
            output_scale = sum([qt.scale for qt in tensor_list]) / len(tensor_list)
            output_zero_point = sum([qt.zero_point for qt in tensor_list]) / len(tensor_list)
            output_bit_width = sum([qt.bit_width for qt in tensor_list]) / len(tensor_list)
            output_signed = first_qt.signed # they are the same
            return QuantTensor(
                output_value, output_scale, output_zero_point, output_bit_width, output_signed)
        else:
            output_value = torch.cat([qt.value for qt in tensor_list], dim=dim)
            return QuantTensor(output_value)


    # Reference: https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types

    def __neg__(self):
        if self.signed:
            return QuantTensor(
                - self.value, self.scale, self.zero_point, self.bit_width, self.signed)
        else:
            return QuantTensor(
                - self.value, self.scale, self.bit_width + 1, signed=True)

    def __add__(self, other):
        QuantTensor.check_input_type(other)
        if self.is_valid and other.is_valid:
            self.check_scaling_factors_same(other)
            self.check_zero_points_same(other)
            output_value = self.value + other.value
            output_scale = (self.scale + other.scale) / 2
            output_zero_point = (self.zero_point + other.zero_point) / 2
            max_uint_val = max_int(signed=False, narrow_range=False, bit_width=self.bit_width)
            max_uint_val += max_int(signed=False, narrow_range=False, bit_width=other.bit_width)
            output_bit_width = ceil_ste(torch.log2(max_uint_val))
            output_signed = self.signed or other.signed
            output = QuantTensor(
                output_value, output_scale, output_zero_point, output_bit_width, output_signed)
        else:
```

```
            output_value = self.value + other.value
            output = QuantTensor(output_value)
        return output

    def __mul__(self, other): # todo zero point
        QuantTensor.check_input_type(other)
        if self.is_valid and other.is_valid:
            output_value = self.value * other.value
            output_scale = self.scale * other.scale
            output_bit_width = self.bit_width + other.bit_width
            output_signed = self.signed or other.signed
            output = QuantTensor(output_value, output_scale, output_bit_width, output_signed)
        else:
            output_value = self.value * other.value
            output = QuantTensor(output_value)
        return output

    def __sub__(self, other):
        return self.__add__(- other)

    def __truediv__(self, other): # todo zero point
        QuantTensor.check_input_type(other)
        if self.is_valid and other.is_valid:
            output_tensor = self.value / other.tensor
            output_scale = self.scale / other.scale
            output_bit_width = self.bit_width - other.bit_width
            output_signed = self.signed or other.signed
            output = QuantTensor(output_tensor, output_scale, output_bit_width, output_signed)
        else:
            output_value = self.value / other.value
            output = QuantTensor(output_value)
        return output

    def __abs__(self):
        if self.signed:
            return QuantTensor(
                torch.abs(self.tensor), self.zero_point, self.scale, self.bit_width - 1, False)
        else:
            return QuantTensor(
                torch.abs(self.tensor), self.zero_point, self.scale, self.bit_width, False)

    def __pos__(self):
        return self
```

Appendix C

FINN Custom Transformations and Node

C.1 *CollapseReshape* Transformation

```
import finn.custom_op.registry as registry
import finn.core.data_layout as DataLayout
from finn.transformation.base import Transformation
import warnings
import numpy as np
import onnx.helper as helper
from onnx import TensorProto

class CollapseReshape(Transformation):

    def apply(self, model):
        graph = model.graph
        node_ind = 0
        graph_modified = False
        for n in graph.node:
            node_ind += 1
            if n.op_type == "Reshape":
              consumer_one=model.find_consumer(n.output[0])
              if consumer_one.op_type=="Transpose":
                consumer_two=model.find_consumer(consumer_one.output[0])
                if consumer_two.op_type=="Reshape":
                  graph_modified = True
                  first_reshape=model.get_initializer(n.input[1])
                  second_reshape=model.get_initializer(consumer_two.input[1])
                  first_edge = helper.make_tensor_value_info(
                    model.make_new_valueinfo_name(), TensorProto.FLOAT,
                        first_reshape.shape
                    )
                  graph.value_info.append(first_edge)
                  model.set_initializer(first_edge.name, first_reshape)

                  last_edge = helper.make_tensor_value_info(
                    model.make_new_valueinfo_name(), TensorProto.FLOAT,
                        second_reshape.shape
```

```
                  )
                graph.value_info.append(last_edge)
                model.set_initializer(last_edge.name, second_reshape)

                new_node = helper.make_node(
                  "ReorderBypass", [n.input[0], first_edge.name, last_edge.name],
                      [consumer_two.output[0]], domain="finn",
                      first_shape=first_reshape, second_shape=second_reshape)
                graph.node.insert(node_ind, new_node)
                graph.node.remove(n)
                graph.node.remove(consumer_one)
                graph.node.remove(consumer_two)
        return (model, graph_modified)
```

C.2 *AddTranspose* Transformation

```
import finn.custom_op.registry as registry
import finn.core.data_layout as DataLayout
from finn.transformation.base import Transformation
import warnings
import numpy as np
import onnx.helper as helper
from onnx import TensorProto

class AddTranspose(Transformation):

    def apply(self, model):
        graph = model.graph
        node_ind = 0
        j=0
        graph_modified = False
        for n in graph.node:
              node_ind += 1
              if n.op_type=="MultiThreshold":
                consumer=model.find_consumer(n.output[0])
                if consumer.op_type=="MultiThreshold":
                  ifm_ch = model.get_tensor_shape(n.output[0])[1] #48
                  ifm_dim = model.get_tensor_shape(n.output[0])[-2] #320
                  idt=model.get_tensor_datatype(n.output[0])
                  inp_trans_out = helper.make_tensor_value_info(
                      model.make_new_valueinfo_name(),
                      TensorProto.FLOAT,
                      (1, ifm_dim, ifm_dim, ifm_ch), # NHWC
                  )
                  graph.value_info.append(inp_trans_out)
                  inp_trans_out = inp_trans_out.name
                  model.set_tensor_datatype(inp_trans_out, idt)

                  graph_modified = True
                  transpose_layer_one = helper.make_node(
                  "Transpose", [n.output[0]], [inp_trans_out], perm=[0, 2, 3, 1]
                  )
```

```
                graph.node.insert(node_ind+1, transpose_layer_one)

                inp_trans_out_2 = helper.make_tensor_value_info(
                    model.make_new_valueinfo_name(),
                    TensorProto.FLOAT,
                    (1, ifm_ch, ifm_dim, ifm_dim), # NCHW
                )
                graph.value_info.append(inp_trans_out_2)
                inp_trans_out_2 = inp_trans_out_2.name
                model.set_tensor_datatype(inp_trans_out_2, idt)
                transpose_layer_two = helper.make_node(
                "Transpose", [inp_trans_out], [inp_trans_out_2], perm=[0, 3, 1, 2]
                )
                graph.node.insert(node_ind+2, transpose_layer_two)
                consumer.input[0]=inp_trans_out_2


        return (model, graph_modified)
```

C.3 *ReorderByPass* Custom Node

```
import finn.core.data_layout as DataLayout
from finn.transformation.base import Transformation
import warnings
from finn.custom_op.base import CustomOp
from finn.util.basic import get_by_name
import numpy as np
import onnx.helper as helper
from onnx import TensorProto


class ReorderBypass(CustomOp):
    def get_nodeattr_types(self):
        return {
            "first_shape": ("i", True, 1),
            "second_shape": ("i", True, 1),
        }

    def infer_node_datatype(self, model):
        node = self.onnx_node
        dtype = model.get_tensor_datatype(node.input[0])
        model.set_tensor_datatype(node.output[0], dtype)

    def get_normal_output_shape(self, model):
        node = self.onnx_node
        if (node.op_type=="ReorderBypass"):
           oshape = model.get_initializer(node.input[2])
        return oshape

    def make_shape_compatible_op(self, model):
        oshape = self.get_normal_output_shape(model)
        values = np.random.randn(*oshape).astype(np.float32)
```

```
        return helper.make_node(
            "Constant",
            inputs=[],
            outputs=[self.onnx_node.output[0]],
            value=helper.make_tensor(
                name="const_tensor",
                data_type=TensorProto.FLOAT,
                dims=values.shape,
                vals=values.flatten().astype(float),
            ),
        )

    def verify_node(self):
        pass


    def execute_node(self, context, graph):
        node = self.onnx_node
        iname = node.input[0]
        first_input= node.input[1]
        second_input= node.input[2]
        x = context[iname]
        first_shape=context[first_input]
        second_shape=context[second_input]
        reshaped_one=np.reshape(x, first_shape)
        if len(first_shape)==6:
          transposed=reshaped_one.transpose((0, 1, 2, 4, 3, 5))
        elif len(first_shape)==5:
          transposed=reshaped_one.transpose((0, 2, 1, 3, 4))

        reshaped_two=np.reshape(transposed, second_shape)
        context[node.output[0]] = reshaped_two
```

C.4 SkyNetQuant model for FINN

```
from collections import OrderedDict
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init as init
from region_loss_cuda import RegionLoss
from utils import *
from collections import OrderedDict

#BREVITAS LIBRARY
import brevitas.nn as qnn
from brevitas.core.quant import QuantType

class PrintLayer(nn.Module):
    def __init__(self):
        super(PrintLayer,self).__init__()

```

```
    def forward(self,x):
        print('Printing a layer:')
        print(x)
        return x

class ReorgLayer(nn.Module):
    def __init__(self, stride=2):
        super(ReorgLayer, self).__init__()
        self.stride = stride
    def forward(self, x):
        stride = self.stride
        assert(x.data.dim() == 4)
        B = x.data.size(0)
        C = x.data.size(1)
        H = x.data.size(2)
        W = x.data.size(3)
        assert(H % stride == 0)
        assert(W % stride == 0)
        ws = stride
        hs = stride
        x = x.view([B, C, H//hs, hs, W//ws, ws]).transpose(3, 4).contiguous()
        x = x.view([B, C, H//hs*W//ws, hs*ws]).transpose(2, 3).contiguous()
        x = x.view([B, C, hs*ws, H//hs, W//ws]).transpose(1, 2).contiguous()
        x = x.view([B, hs*ws*C, H//hs, W//ws])
        return x


class SkyNetQuant(nn.Module):
    def __init__(self, weight_bit_width=4, act_bit_width=4, in_bit_width=4):
        super(SkyNet, self).__init__()
        self.width = int(320)
        self.height = int(320)
        self.header = torch.FloatTensor([0,0,0,0])
        self.seen = 0
        self.reorg = ReorgLayer(stride=2)

        def conv_dw_Brevitas(inp, oup, stride):
            return nn.Sequential(
                qnn.QuantConv2d(in_channels=inp, out_channels=inp, kernel_size=3,
                    stride=1, padding=1, groups=inp, bias=False,
                    weight_bit_width=weight_bit_width),
                nn.BatchNorm2d(inp),
                qnn.QuantReLU(bit_width=act_bit_width, max_val=6),
                qnn.QuantConv2d(in_channels=inp, out_channels=oup, kernel_size=1,
                    stride=1, padding=0, groups=1, bias=False,
                    weight_bit_width=weight_bit_width),
                nn.BatchNorm2d(oup),
                qnn.QuantReLU(bit_width=act_bit_width, max_val=6),
            )


        self.model_p1 = nn.Sequential(
            qnn.QuantIdentity(bit_width=8),
            conv_dw_Brevitas(3, 48, 1), #dw1
            qnn.QuantMaxPool2d(kernel_size=2, stride=2),
```

```python
            qnn.QuantIdentity(bit_width=8),
            conv_dw_Brevitas(48, 96, 1), #dw2
            qnn.QuantMaxPool2d(kernel_size=2, stride=2),
            qnn.QuantIdentity(bit_width=8),
            conv_dw_Brevitas(96, 192, 1), #dw3
        )
        self.model_p2 = nn.Sequential(
            qnn.QuantMaxPool2d(kernel_size=2, stride=2),
            qnn.QuantIdentity(bit_width=8),
            conv_dw_Brevitas(192, 384, 1), #dw4
            conv_dw_Brevitas(384, 512, 1), #dw5
        )
        self.model_p3 = nn.Sequential( #cat dw3(ch:192 -> 768) and dw5(ch:512)
            conv_dw_Brevitas(1280, 96, 1),
            qnn.QuantConv2d(in_channels=96, out_channels=10, kernel_size=1,
                weight_bit_width=weight_bit_width, bias=False),
        )
        self.identity=qnn.QuantIdentity(bit_width=8)
        self.loss = RegionLoss([1.4940052559648322, 2.3598481287086823,
            4.0113013115312155, 5.760873975661669],2)
        self.anchors = self.loss.anchors
        self.num_anchors = self.loss.num_anchors
        self.anchor_step = self.loss.anchor_step
        self._initialize_weights()

    def forward(self, x):
        x_p1=self.model_p1(x)
        x_p1_reorg = self.reorg(x_p1)
        x_p2 = self.model_p2(x_p1)
        x_p3_in = torch.cat([x_p1_reorg, x_p2], 1)
        x_p3_in=self.identity(x_p3_in)
        x = self.model_p3(x_p3_in)
        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, qnn.QuantConv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, qnn.QuantLinear): #NOT PRESENT IN THE NETWORK
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)
```

Bibliography

[1] He J. Zhang X. Hao C. Chen, Y. and D. Cloud Chen. *DNN: An open framework for mapping DNN models to cloud FPGAs*. 2019.

[2] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 1 edition, 2017.

[3] Zhu M. Chen B. Kalenichenko D. Wang W. Weyand T. Andreetto M. Howard, A. G. and H. Adam. Efficient convolutional neural networks for mobile vision applications. In *arXiv preprint arXiv:1704.04861*, 2017.

[4] Xilinx Research Lab. Finn, 2020.

[5] Umberto Michelucci. *Applied Deep Learning: A Case-Based Approach to Understanding Deep Neural Networks*. Apress, 2018.

[6] Alessandro Pappalardo. Xilinx/brevitas.

[7] Magnus Jahre Yaman Umuroglu. *Streamlined Deployment for Quantized Neural Networks*. arXiv preprint arXiv:1709.04060, 2017.

[8] Xiaofan Zhang, Haoming Lu, Cong Hao, Jiachen Li, Bowen Cheng, Yuhong Li, Kyle Rupnow, Jinjun Xiong, Thomas Huang, Honghui Shi, Wen-mei Hwu, and Deming Chen. SkyNet: a hardware-efficient method for object detection and tracking on embedded systems. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

Ringraziamenti

Ringrazio il mio relatore, il professor Luciano Lavagno, per i suoi consigli sul lavoro di tesi e per la sua immensa disponibilità in un anno così difficile. Ringrazio inoltre tutti i colleghi del gruppo *HLS Weekly Meeting*, con i quali ogni settimana ho condiviso i progressi del mio lavoro.

Ringrazio gli sviluppatori di *Brevitas* e *FINN*, rispettivamente Alessandro Pappalardo e Yaman Umuroglu di Xilinx Research Lab Ireland, per esser stati disponibili e per avermi dato la possibilità di capire fino in fondo il funzionamento di questi programmi.

Ringrazio i miei colleghi e migliori amici Mario e Pietro, perché questi 5 anni di Politecnico non sarebbero stati la stessa cosa senza di loro. Grazie per essere stati i migliori compagni di banco in assoluto e per aver condiviso con me gioie e dolori di questo percorso. Grazie per le risate, le sessioni studio, le corse al treno e le passeggiate notturne a Torre Pellice, non vedo l'ora di tornare a festeggiare con voi.

Ringrazio Monica, per essere mia amica, collega, confidente e anche compagna di nuoto. Grazie per essermi sempre stata accanto durante questi anni e per avermi capita e sostenuta.

Ringrazio Michela, amica di una vita e compagna di banco di sempre, perché questo percorso è cominciato ben prima di 5 anni fa e senza il suo sostegno e la sua amicizia molto probabilmente non sarei arrivata fin qui ora.

Ringrazio mia madrina Ornella, per avermi aiutata e per essermi stata accanto durante ogni passo di questo lungo percorso, ancor prima che iniziasse.

Ringrazio mio papà, che mi ha spronato a mettermi in gioco e che mi ha detto di credere sempre in me stessa e nelle mie idee.

Ringrazio mia mamma, che mi ha sempre ricordato le cose che contano e che ha lottato affinché il mio futuro fosse il migliore possibile.

Ringrazio mia sorella Luisa, che mi ha incoraggiata a non mollare mai e che mi ha insegnato a vivere la vita con più spensieratezza.

Infine, il ringraziamento più importante è per Simone, il mio compagno di avventure e di vita, perché crede in me ancor più di quanto ci creda io e così facendo mi ha dato l'incoraggiamento necessario a resistere. Grazie per avermi sostenuta anche nei momenti bui di questo percorso, grazie per avermi ascoltata e consolata, grazie per avermi fatto ridere e grazie per esserci sempre stato.