Therefore, each circle plotted on the grid part of Figure 3.1 does two task. Fist implement instantaneus corrologram of CAR's channels using a xnor gate; and secondly accumulate the output of xnor gate over a preset time window.

3.4 Quantize and sort output image

The HDL design on the SSL system is slightly different from the simulation system. In the simulation as shown is Figure **??** the instantaneous corrologram and onset corrologrames operates as separate blocks.

In the HDL system design we actually mixed these operations with the quantization calculations. We need to implement instantaneous corrologram, onset corrologram (which consists of accumulation of instantaneous corrologram and divide the result by $\Delta t \times f_s$ factor) and quantization respectively (see sections 2.1.4 and 3.5.2). In this project we mixed this three operations and implement them in 2 separate blocks.

The operations of calculating instantaneous corrologrames and their accumulations happen in the Grid shape, as explained in section3.3; and the operations of dividing the result over $\Delta t \times f_s$ and quantizations are merged in the "Quantize and sort output" block as shown in equatio 3.5. So this block requires two receive two parameters: 1- pre-calculated value for : $\frac{1}{s_a \times \Delta t \times f_s}$ and 2- z_a.

$$q_{onset\_corr} = \frac{1}{s_a}(\frac{Acc_{instant\_corr}}{\Delta t \times f_s}) + z_a \tag{3.5}$$

This block receives all the input values in parallel, complete the onset corrologram and quantization, then send out the extracted image pixels serially.

3.5 CNN hardware design

This section introduces "tf4fpga" FPGA accelerator which is a purpose-built FPGA-based accelerator for the CNN introduce in section2.1.5. There are a few available approaches for transferring a trained CNN to the edge computing systems such as High-Level Synthesis (HLS), Xilinx Vitis AI, tfLite and so on. Among the all possible approaches tfLite model is our interest in this project since we need to have a good control over the low level hdl design. TensorFlow Lite uses TensorFlow models converted into a smaller, more efficient machine learning (ML) model format. You can use pre-trained machine

learning models with TensorFlow Lite, modify existing models, or build your own TensorFlow models and then convert them to TensorFlow Lite format. TensorFlow Lite models can perform almost any task a regular TensorFlow model can do: object detection, natural language processing, pattern recognition, and more using a wide range of input data including images, video, audio, and text [1]. An important development in CNN research relevant to hardware accelerators is the development of methods for training CNNs that use low precision weights, activations. In this project we use an standard approach that tensorflow is provided named quantization aware training. Since for using this concept on FPGAs we need to have a good understanding of the tfLite model and it's training mechanism, it's explained in section 3.5.2. After getting familiar with the quantization mechanism, in section 3.5.3 our proposed block diagram for the FPGA-based CNN architecture is described. Section 3.6 reports an employed cnn in a sound source localizing task with a comparison to the golden model.

3.5.1 CNN HDL design methodology

Figure3.8 illustrates our methodology for designing tf4fpga. We first train the convolution neural network in TensorFlow using our pre-cochlea processed data. Then quantize the trained CNN .tf model. In the next step, the quantized CNN model is converted to the TensorFlow Lite (TFLite) model. TFLite is a collection of tools to convert and optimize TensorFlow models to run on mobile and edge devices. It was developed by Google for internal use and later open-sourced. Today, TFLite is running on more than 4 billion devices. It's possible to visualize the .tflite model using different tools. Here we use Netron for visualizing the weight, biases, and all the scaling factors of the CNN .tflite model. Then using the extracted factors, weights, and biases, we build a golden model of the CNN. After designing and implementing the HDL model of the CNN, the result of the HDL design is confirmed by comparison with the golden model result.

3.5.2 TensorFlow Lite 8-bit quantization specification

The following outlines the specification for TensorFlow Lite's 8-bit quantization scheme. This is intended to assist hardware developers in providing hardware support for inference with quantized TensorFlow Lite models [2].

[1] https://www.tensorflow.org/lite/models

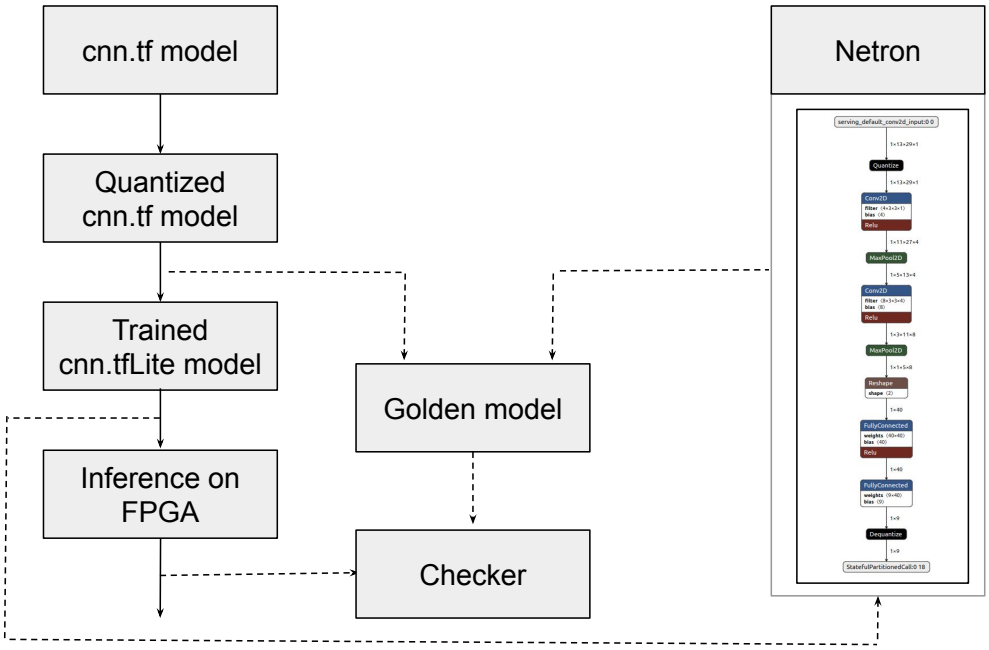

Figure 3.8: Our methodology for designing an HDL model of CNN.

Quantization-aware training

In this section, we describe the general quantization scheme, that is, the correspondence between the bit representation of values (denoted q below, for "quantized value") and their interpretation as mathematical real numbers (denoted r below, for "real value"). This quantization scheme is implemented using integer-only arithmetic during inference and floating-point arithmetic during training, with both implementations maintaining a high degree of correspondence with each other. A basic requirement of this quantization scheme is that it permits efficient implementation of all arithmetic using only integer arithmetic operations on the quantized values. Equation 3.6 is the quantization scheme and the constants S and Z are the quantization parameters. Here S and Z represent scale and zero-point respectively [5].

$$r = S(q - Z) \tag{3.6}$$

Quantize operation Its main role is to bring the float values of a tensor to low precision integer values. It is done based on the above-discussed quantization scheme. Scale and zero-point are calculated in the following way:

The main role of scale is to map the lowest and highest value in the floating range to the highest and lowest value in the quantized range. In the case of 8-bit quantization, the quantized range would be [-128,127].

$$S = \frac{f_{max} - f_{min}}{q_{max} - q_{min}} \tag{3.7}$$

here f_{max} and f_{min} represent the maximum and minimum value in floating-point precision, q_{max} and q_{min} represent the maximum value and minimum value in the quantized range. Similarly, we can find the zero-point by establishing a linear relationship between the extreme floating-point values and the quantized values. Considering we have coordinates of two points of a straight line (q_{min}, f_{min}) and (q_{max}, f_{max}), we can obtain its equation in the form of y = mx +c, x being the quantized values and y being the real values. So to get the mapping of 0 in the quantized domain, we just find the value of x with y=0. On solving this we would get:

$$Z = q_{min} - \frac{f_{min}}{scale} \tag{3.8}$$

But what if 0 doesn't lie between f_{max} and f_{min}, our Zero point would then go out of the quantization range. To overcome this, we can set Z to q_{max} or q_{min} depending on which side it lies on.

De-quantize operation To obtain back the real values we put the quantized value in Equation 1, so that becomes:

$$r_{new} = S([\frac{r}{S} + Z] - Z) \tag{3.9}$$

Specification summary

8-bit quantization approximates floating point values using the following formula.

$$real\_value = (int8\_value - zero\_point) \times scale \tag{3.10}$$

Per-axis (aka per-channel in Conv ops) or per-tensor weights are represented by int8 two's complement values in the range [-127, 127] with zero-point equal to 0. Per-tensor activations/inputs are represented by int8 two's complement values in the range [-128, 127], with a zero-point in range [-128, 127].

Per-axis vs per-tensor Per-tensor quantization means that there will be one scale and/or zero-point per entire tensor. Per-axis quantization

means that there will be one scale and/or zero_point per slice in the quantized_dimension. TFLite has per-axis support for a growing number of operations.

Symmetric vs asymmetric: Activations are asymmetric: they can have their zero-point anywhere within the signed int8 range [-128, 127]. Many activations are asymmetric in nature and a zero-point is an relatively inexpensive way to effectively get up to an extra binary bit of precision. Since activations are only multiplied by constant weights, the constant zero-point value can be optimized pretty heavily.

Weights are symmetric: forced to have zero-point equal to 0. Weight values are multiplied by dynamic input and activation values. This means that there is an unavoidable runtime cost of multiplying the zero-point of the weight with the activation value. By enforcing that zero-point is 0 we can avoid this cost. Explanation of the math which is similar to section 2.3 of this reference [5] is as follow:
A is a matrix of $m \times n$ quantized activations.
B is a matrix of $n \times p$ quantized weights.

Consider multiplying the jth row of A, a_j, by the kth column of B, b_k, both of length n. The quantized integer values, zero-points values and scale values are q_a, z_a, s_a , and q_b, z_b, s_b , respectively (see Figure 3.9 as an example).

$$a_j.b_k = \sum_{i=0}^{n} a_j^{(i)} b_k^{(i)} = \sum_{i=0}^{n} (q_a^{(i)} - z_a) s_a (q_b^{(i)} - z_b) s_b \tag{3.11}$$

$$= s_a s_b (\sum_{i=0}^{n} q_a^{(i)} q_b^{(i)} - \sum_{i=0}^{n} q_a^{(i)} z_b - \sum_{i=0}^{n} q_b^{(i)} z_a + \sum_{i=0}^{n} z_a z_b) \tag{3.12}$$

The term $s_a s_b(\sum_{i=0}^{n} q_a^{(i)} q_b^{(i)})$ is unavoidable since it's performing the dot product of the input value and the weight value.
The term $s_a s_b(\sum_{i=0}^{n} q_a^{(i)} z_b)$ needs to be computed every inference since the activation changes every inference. By enforcing weights to be symmetric ($z_b = 0$) we can remove the cost of this term.
The $s_a s_b(\sum_{i=0}^{n} z_a z_b)$ and $s_a s_b(\sum_{i=0}^{n} q_b^{(i)} z_a)$ terms are made up of constants that remain the same per inference invocation, and thus can be pre-calculated (for $z_b = 0$ we only need to take care of $s_a s_b(\sum_{i=0}^{n} q_b^{(i)} z_a)$ term). With this assumptions 3.12 can be simplified as follow:

$$(q_c - z_c) s_c = s_a s_b (\sum_{i=0}^{n} q_a^{(i)} q_b^{(i)} + \sum_{i=0}^{n} q_b^{(i)} z_a) \tag{3.13}$$

$$q_c = M (\sum_{i=0}^{n} q_a^{(i)} q_b^{(i)} + \sum_{i=0}^{n} q_b^{(i)} z_a) + z_c \tag{3.14}$$

where the multiplier M is defined as:

$$M = \frac{s_a s_b}{s_c} \tag{3.15}$$

In Equation 3.13, the only non-integer is the multiplier M . As a constant depending only on the quantization scales s_a, s_b, s_c, it can be computed offline. M values always in the interval (0, 1), and can therefore express it in the normalized form:

$$M = 2^{-n} M_0 \tag{3.16}$$

where M0 is in the interval [0.5, 1) and n is a non-negative integer. The normalized multiplier M0 now lends itself well to being expressed as a fixed-point multiplier. Meanwhile, multiplication by 2^{-n} can be implemented with an efficient bit-shift [2].
In summary the down-scaling corresponds to multiplication by the multiplier M in equation3.14. As explained in this section, it is implemented as a fixed-point multiplication by a normalized multiplier M0 and a rounding bit-shift. Afterwards, we perform a saturating cast to uint8 and to the range [0, 255]. We focus on activation functions that are mere clamps, e.g. ReLU, ReLU6. Mathematical functions are discussed in [5], Appendix A.1 and we do not currently fuse them into such layers. Thus, the only thing that our fused activation functions need to do is to further clamp the uint8 value to some sub-interval of [0, 255] before storing the final uint8 output activation. In practice, the quantized training process tends to learn to make use of the whole output uint8 [0, 255] interval so that the activation function no longer does anything, its effect being subsumed in the clamping to [0, 255] implied in the saturating cast to uint8. In simple words to practically use this technique, first we need to visualize the tfLite model using some packages for example Netron [3] is a very powerful tool for visualizing a tfLite model. Then extract the down-scaling parameters such as scales and z-points (s_a and z_a for input, s_b and z_b for weights, and s_c and z_c for output). Pre-calculate M_0 and n and also $s_a s_b(\sum_{i=0}^{n} q_b^{(i)} z_a)$. Now with having all the pre-calculated values, 8bit can be extracted from 32bit using equation 3.14. Adding z_c to the product shifts the output range [0, 255] to a signed range with the center of z_c.

3.5.3 High level block diagram of the proposed FPGA-based CNN

Fig 3.10 illustrates a top level block diagram of the FPGA-based CNN architecture. Our custom CNN composed of two convolution (Conv2D) layers.

[2] https://www.tensorflow.org/lite/performance/quantization_spec
[3] https://netron.app/

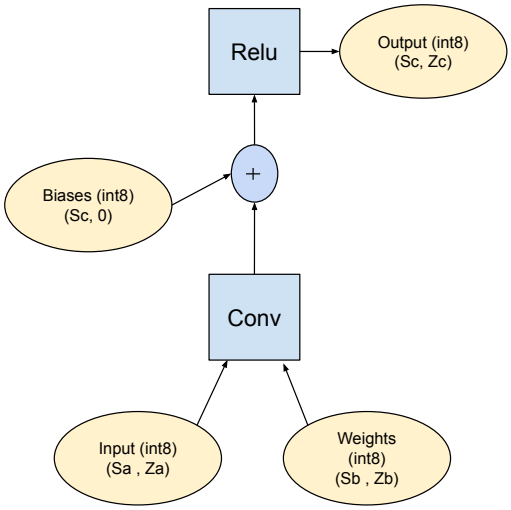

Figure 3.9: Representation of Quant-Inference Graph.

Conv2D has flexible input and output feature maps dimension, and accordingly flexible number of 3x3 kernels. For this specific application we don't have Padding. The ReLU function is the non-linearity added to the system. The non-linear transformation of the system follows by fixed-size 2x2 Max-pooling operation. We don't implement batch normalization in inference of the CNN. Flatten outputs of the two Conv2D layers are fed to one Fully-Connected (or dense) layer. At the end of the CNN we have our last full-connected layer with 9 output neuron each specifies information about one spatial division of the space (based on the application).

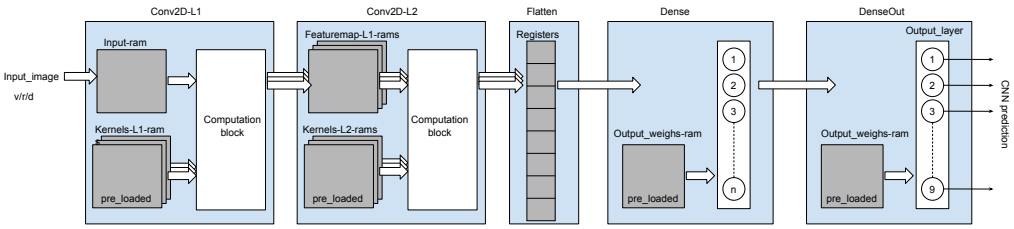

Figure 3.10: High level block diagram of our custom FPGA-based CNN.

Conv2D: Fig 3.12 shows the high level block diagram of Conv2D layer with F^{in} and F^{out} as input and output feature maps, respectively. Each Conv2D layer has $F^{in}_{depth} \times F^{out}_{depth}$ kernels of a fixed size 3x3 which should be preloaded in separate BRAMs. Before starting the whole convolution operation

we need to save the whole feature ins in the memories as well. After saving the complete F^{in}s in the memories, convolution operations start with doing multiply-adds (MACs). The output of MACs are fed to the accumulator blocks (ACCs) to calculate a summation of the MACs results as well as the bias for each F^{out} channel. Therefore, there are F^{in}_{depth} MACs and one ACC for each F^{out} channel. tfLite model selects 8 bits for weights and 32 bits for activation and biases. It means the summation result has 32 bits and needs to be down-scaled to 8 bits before applying to the output. For doing so we need to have a good understanding of the quantization aware training explained in section 3.5.2. Block "Int32_t_int8" in Fig 3.12 implements this mechanism and it's shown in more details in Fig 3.13. According to the tfLite quantization mechanism this block needs to receive 4 pre-calculated parameters including M0, n base on equation 3.16, z_c which is the output z-point and can be extracted from tfLite model using visualization tools such as Netron, and finally term $\frac{s_a s_b}{s_c}(\sum_{i=0}^{n} q_b^{(i)} z_a)$ which we call it b_{extra}. (s_a, z_a), (s_b, z_b) and (s_c, z_c) are scale and z-point factors of input, weights and output, respectively. An example of CNN network visualization is shown in figure **??**. It is possible to extract all the CNN parameters such as each layers' hyper parameters, quantization factors, weights and biases by visualizing the tfLite model uzing Netron. We can use this parameters to calculate the preset values we need in "Int32_t_int8" block. As shown in Fig 3.13, the first step after receiving Int32_i as input is the summation with the b_extra value and then a cast operation. Casting works similar the ReLU function and the output is either positive (Uint) or zero. Then the casting output should be multiply by the M_0 factor and shift-right n times to implement the multiplication by M. Output if this stage should be clamped in the range [0,255], at the end of the pipeline to make the data ready for the next stage, the z-point value should be added to the result.

The final processing step of the computational pipeline is Max-pooling. When transferring activation from the Int32_to_Int8 buffer to the output buffer, a max operation is performed so that the value stored in the output buffer is the maximum of the incoming activation and the original stored value. When pooling is enabled, the maximum of the three values of the activation pair and the current content of the output buffer is stored back into the output buffer.

Flatten: The Flatten implements an finite state machine to fictionalize a really simple task; This block receive the output of the Conv2D layer sequentially and flats and stores them in a 1D array.

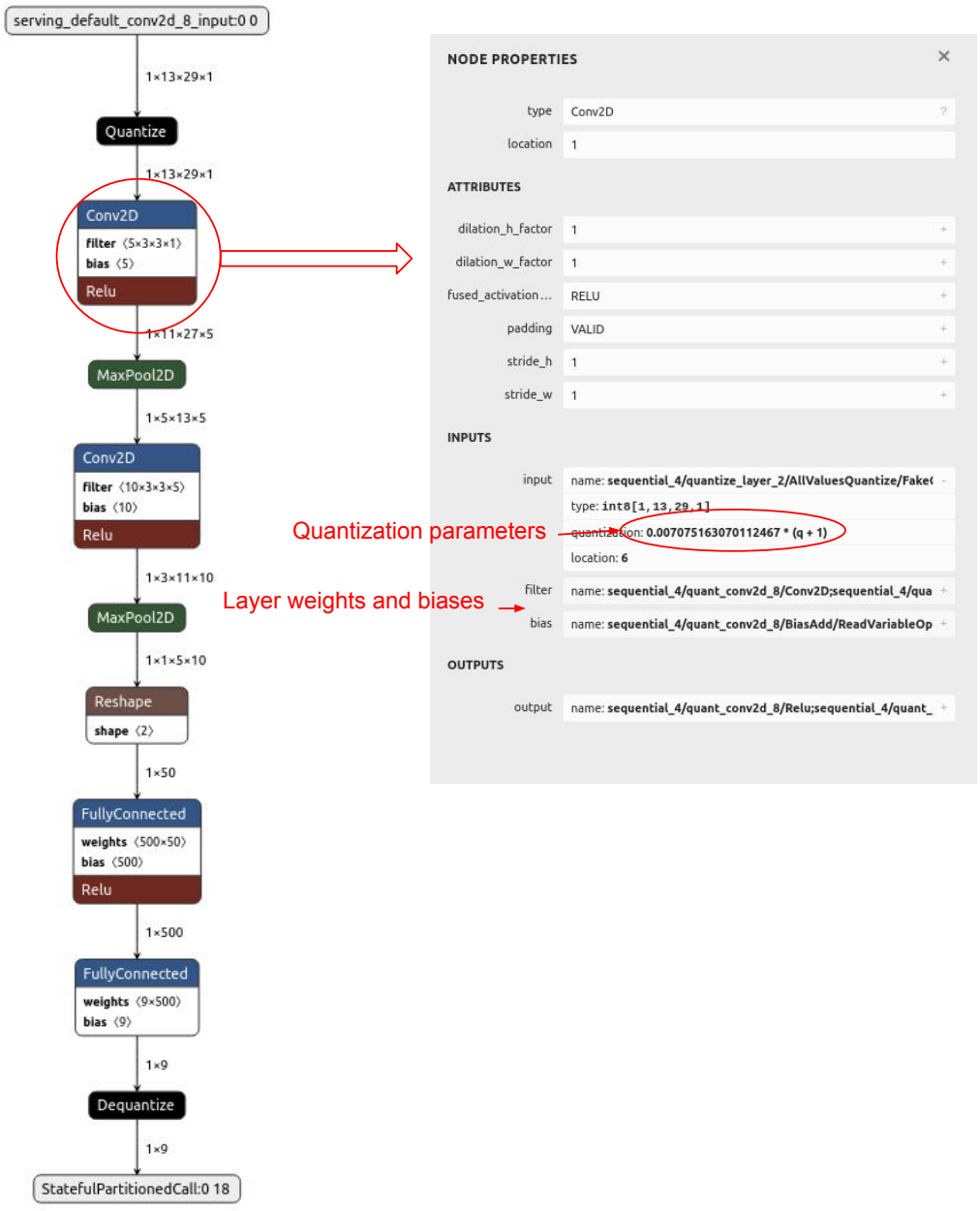

Figure 3.11: Visualization of a trained and quantized CNN model so-called tfLite using Netron.

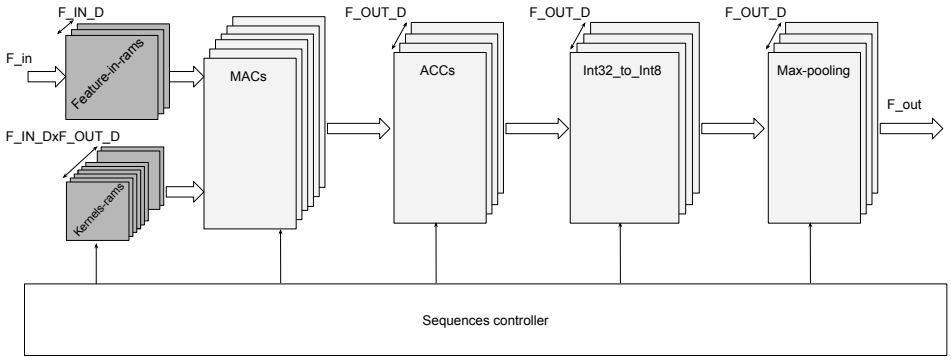

Figure 3.12: High level block diagram of FPGA-based Conv2d.

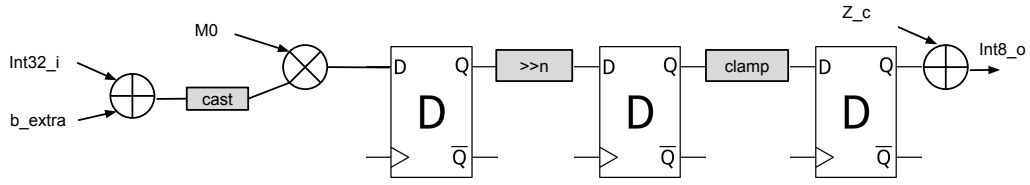

Figure 3.13: Down scaling block from 32bit to 8bit

Dense: As shown in Fig 3.14, this block operates exactly similar to FlattenDense with difference in accessing the input. Dense layer receives input directly from the previus layer (which should be FlattenDense layer in this specific application) and then does MAC and down-scaling operations for each output neuron.

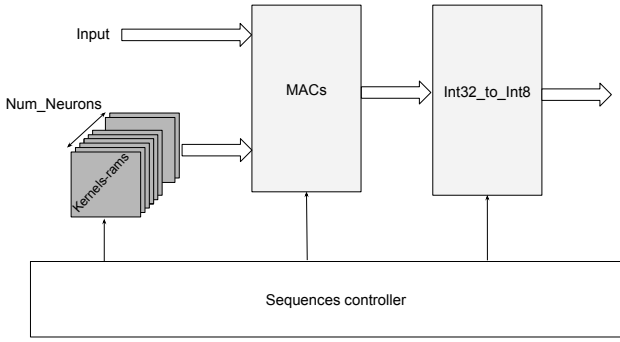

Figure 3.14: High level block diagram of Dense layer block.

3.6 SSL HDL implementation result

In the previous sections of this chapter, we have described the block diagram of all the individual parts of the sound source localizar system. In this section we put all the pieces together and employ the SSL system to localize an input sound. We first describe the datasets, then show the cochlea processing results, visualize the conv2D feature maps and finally analyze the output.

3.6.1 Resource Utilization

The proposed SSL system implements a digital electronic sound source localizer as shown in Figure 3.1. It composes of a digital cochlea processing and a CNN (tf4fpga). All parts of the design are described using the standard top-bottom digital design flow. As a proof of concept, we trained the CNN off-line using pre-recorded datasets (as described in section **??**) and implemented the SSL system on a Xilinx Artix-7 xc7a100tcsg324-1 device. Based on the synthesis results from Table 3.2, the CAR uses only less than 1 percent of the available Slice FFs and LUTs and 3 percent of available BRAMs and DSPs. 2.31 percent of available Slice LUTs and 38 percent of available BRAMs. The total resource utilization of the SSL system is 60 % of LUTs, 18% of FFs, 41% of BRAMs, and 37% of DSPs.