A library-based tool to translate high level DNN models into hierarchical VHDL descriptions

Pablo Rubio-Ibañez, J.Javier Martínez-Álvarez, G. Doménech-Asensi
Departamento de Electrónica, Tecnología de Computadoras y Proyectos
Universidad Politécnica de Cartagena
Cartagena, Spain
gines.domenech@upct.es

Abstract. This work presents a tool to convert high level models of deep neural networks into register transfer level designs. In order to make it useful for different target technologies, the output designs are based on hierarchical VHDL descriptions, which are accepted as input files for a wide variety of FPGA, SoC and ASIC digital synthesis tools. The presented tool is aimed to speed up the design and synthesis cycle of such systems and provides the designer with certain capability to balance network latency and hardware resources. It also provides a clock domain crossing to interface the input layer of the synthesized neural networks with sensors running at different clock frequencies. The tool is tested with a neural network which combines convolutional and fully connected layers designed to perform traffic sign recognition tasks and synthesized under different hardware resource usage specifications on a Zynq Ultrascale+ MPSoC development board.

Keywords. High-Level Synthesis, VHDL, Deep Neural Network, Keras, Tensorflow

I. INTRODUCTION

Deep learning has become one of the most powerful methods within the paradigm of artificial intelligence to perform different computer vision tasks. From image recognition to object tracking, deep learning has reduced the use of well known feature-extraction based algorithms such FAST or SIFT; and currently there is a wide range of tools to develop, train, optimize and implement such deep neural networks (DNN). Among these tools, we can cite Tensorflow [1], PyTorch [2], Keras [3], Caffe [4] or even Matlab. These tools are focused on high performance computing (HPC) platforms, either using single or multi-core CPUs or GPUs to speed up both the training and the inference of the networks.

On the other hand, end devices such as Internet-of-Thing sensors or portable CMOS imagers, require real time processing capabilities while having a low power dissipation. So, it is usual to find applications where both deep learning and edge computing are combined to meet the requirements from both fields of study [5]. However, edge computing devices are usually based on cost effective microprocessors, SoCs or ASICs whose computing performance is far away from the aforementioned HPC.

Although currently it is usual to find lots of information regarding DNN training and inference even on personal computers, there is not so much when the target implementation is based on a specific hardware platform such an FPGA or an ASIC. So, when such edge computing devices for DNN based computer vision applications are designed, engineers find a gap between the high level tools aimed to develop DNNs on high performance computers and the resource limitations of the hardware in which the final application is being deployed. This means that once a DNN architecture has been trained on a computer, the designer must manually translate this network, from scratch, to the target hardware. This operation is not trivial and requires skilled designers with both knowledge of high level DNNs modeling tools and digital electronic design. Even if this is such a case, it is well known that the electronic system design cycle is composed by a set of sequential steps from higher to lower hierarchy levels, where it is usual to find iterative procedures between two consecutive steps until a satisfactory solution is found. This is the case, for instance, where a tradeoff between a high level specification such as accuracy or latency must meet low level performance such as silicon area / resources usage or power dissipation.

There are different proposals in the literature of tools able to transform high level descriptions of neural networks into synthesizable architectures. In [6] a tool able to build board level FPGA implementations from high level descriptions is presented. Also, in [7] an open source tool which translates Tensorflow designs into Cloud FPGAs is described. A tool to translate high level Keras descriptions of DNNs into low level C code is shown in [8]. Finally, in [9] a Python package to create DNN implementations on low-cost FPGAs is developed. It is clear then that tools to link high level descriptions created using common DNN development tools to low level hardware descriptions is a current research topic. However, most of these tools are focused on reconfigurable hardware platforms as target technology. The main advantage of such devices is that they allow to create prototypes very quickly, which are cost-effective when small volume productions are run. However, when a large amount of devices are manufactured or if the power dissipation specifications are really tight, ASICs become more competitive and thus, the use of a translation tool of the type described above becomes interesting.

In this work, a tool to create VHDL hierarchical descriptions using a set of parameterizable DNN building blocks from a high level DNN description is presented. The tool allows a tradeoff between hardware resource usage and network latency; and the VHDL code produced is able to be synthesized on different target technologies such as PFGAs, SoCs or ASICs. Its performance is tested performing the

synthesis of a traffic sign recognition (TSR) DNN. The rest of the paper is organized as follows. Section II describes the structure of the translation tool. Section III shows a comparison of the results obtained when synthesizing the TSR DNN under different design strategies. Finally, conclusions are drawn in section IV.

II. Tool Description

The general outline of the tool workflow is shown in Fig. 1. The tool reads an HDF5 format file [10] containing the high level description of a neural network. HDF5 is an open source file format which supports complex and large data; and its use is widely accepted by the majority of the high level deep learning tools such as Tensorflow, Caffe or Matlab. The tool parses then this file and extracts the network structure, as well as the weights, of each convolutional and fully connected layers. The aim is to build a structure of pipelined stages which maps the sequence of layers in the DNN model, as shown in Fig. 2. The translation process relies on a VHDL library which contains structural descriptions of different network components at different subsystem level, such as convolutional and fully connected layers; operators like comparators, adders and multipliers; auxiliary data path components like FIFO registers, counters, latches, multiplexers and others and interfacing ones such us clock domain crossing circuits.

The user preferences include the specification of data path width for the activations and for the network weights as well as the level of parallelism of the design, as it will be further detailed. As a result, the tool creates a refined hierarchical VHDL netlist which corresponds to the lowest level of the high level synthesis flow, since it is not based on any target technology cells. This VHDL description is then suitable to be processed by a low level synthesis tool to convert it to a proprietary technology cell based design, either for FPGA / SoC or ASIC, and later to be placed and routed to obtain the final prototype.

As shown in Fig. 2, for each layer read from the file, a new stage is added to the pipeline at the level 2 of the VHDL hierarchy. Level 3 of the VHDL hierarchy is composed by the parallel channels inside each layer. Finally, the fourth level of the hierarchy is a flat one which includes the two main types of resources involved in the synthesis process: computation units and memory. In order to balance the usage of computation resources and the network latency, the degree of hardware parallelism can be selected during the translation process. This parallelism is only applicable to the multiplier structures. In the case of a convolutional layer using an $k \times k$ convolution mask, the user can select to use a fully serial approach using a single multiplier, a full parallel one using n^2 multiplier or finally a mixed one using k multipliers (Fig. 3). In the case of a dense layer, the user can select whether to use a single multiplier or to use a multiplier per layer output.

Regarding the multiply operations, the tool does not select the architecture of the low level operator, i.e. DSP or carry-and-adder multiplier, being this selected by the low level synthesis software used afterwards, using the corresponding compiler directives or flags. From our experience, for DNNs with weights sizes up to 10 bits, multiplications can be synthesized using combinational logic without using in-chip DSPs, so it is easier to find a target technology to fit them in, like low-cost FPGAs.

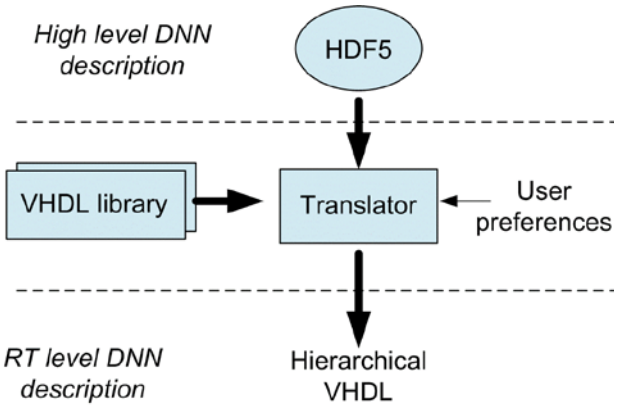

Fig. 1. Structure of the translation tool.

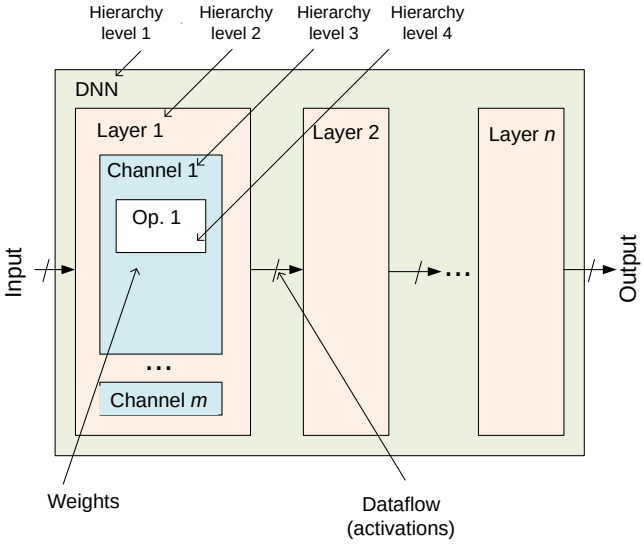

Fig. 2. VHDL pipelined structure of the synthesised DNN.

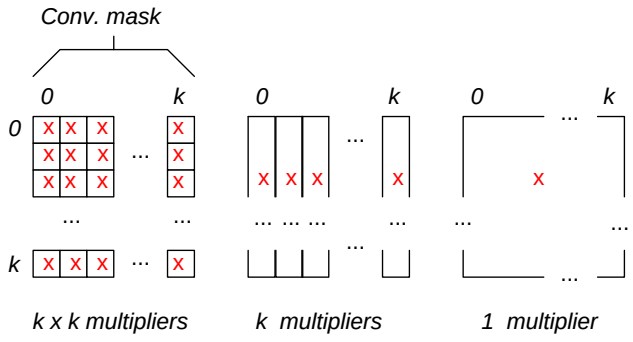

Fig. 3. Usage of multipliers in convolutional layers.

Regarding the memory resources, the current version of the tool only deals with in-chip memory to store the network weights. Again, it requires a tradeoff between the complexity of the network, and the size of the activations and the weights on the one side and the number of hardware resources on the other.

The VHDL library (Fig. 1) contains a set of parameterized VHDL circuits commonly used in DNNs. In order to offer flexibility for different types of networks, convolutional layers are fully parameterized and allow any mask size, being the smallest one a 3x3 pixel size. The stride and padding can also be parameterized. Fully connected layers also allow different combinations of input and output sizes. In both types of layers, the largest sized would be limited by the number of resources of the target device or the maximum silicon area available. With respect to the output activation functions, both types of layers can deal with linear or ReLU functions.

Other layers that are included in the library are the flatten one, needed to link convolutional layers with fully connected ones, and batch normalization layers. In this later case their implementation to perform synthesis is a linear function, given that, once the values of μ, σ, β and γ have been trained, their values are constant and the output activation y for a given input activation x, can be obtained as:

$$y = \gamma \frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}} + \beta \quad (1)$$

The tool provides some flexibility in order to support an easy interfacing between the input layer and an external sensor or device. First, it includes the possibility to use a clock domain crossing (CDC) circuit to enable the use of different clocks for the external sensor and for the synthesized DNN. Following the paradigm of the Globally Asynchronous Locally Synchronous (GALS) systems [11], these architectures can enhance the latency of a whole system while running subsystems at different clock rates. Second, in the case of a color image sensor, the input format can also be adapted to read different RGB sequence formats, depending on how the image bits are ordered at the RGB camera output.

Although the tool can be used for different types of DNNs, it has been thought to be used mainly for image processing. So, it can work with input tensors composed by grayscale or color images, and with different image sizes.

III. RESULTS

The described tool has been tested to model and synthesize a convolutional neural network aimed to perform TSR tasks to classify the 43 classes of traffic signs described in [12]. Fig. 4 shows the Keras description of the proposed neural network, which is composed by four convolutional layers and a dense layer. None of the layers use bias, in order to cut off on the number of weights to be stored. The input to the network are 32x32 RGB images and the output is a 43 classes vector.

The four convolutional layers use 3x3 masks and ReLU activations. The dense layer *softmax* activation is used only during the training process and it is replaced by a linear activation for the inference. Given that both functions are monotonic, the maximum output value will be the same. Although *softmax* has other advantages, such us helping to clarify the distance between two outputs, its synthesis is not trivial in an all-hardware implementation. In this example the synthesis has been performed on a Zynq Ultrascale+ MPSoC development board [13] using Xilinx Vivado Design Suite 2018 [14]. The absolute and relative usage of resources (logic LUTs, FFs or RAMB18) is detailed in Table I, where three different data resolution has been used (6, 7 and 8 bits) for three approaches of multiplier parallelism.

```
model = Sequential()

# Convolution 1
model.add(Conv2D(26, kernel_size=(3, 3),
   activation='relu', use_bias = False,
   input_shape=(32, 32, 3)))

# Convolution 2
model.add(Conv2D(20, kernel_size=(3, 3),
   activation='relu', use_bias = False))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Convolution 3
model.add(Conv2D(20, kernel_size=(3, 3),
   activation='relu', use_bias = False))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Convolution 4
model.add(Conv2D(12, kernel_size=(3, 3),
   activation='relu', use_bias = False))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Dense
model.add(Flatten())
model.add(Dense(43, activation='softmax',
   use_bias = False))
```

Fig. 4. Keras description of the proposed deep neural network.

TABLE I. USAGE OF RESOURCES

Resource	Bits	Number of parallel multipliers per convolution		
		1 Mult	***3 Mult***	***9 Mult***
Logic LUTs	6	237027 (86.48%)	338705 (123.58%)	261360 (95.36%)
	7	306830 (111.95%)	455894 (166.34%)	369093 (134.67%)
	8	339471 (123.86%)	516992 (188.63%)	508756 (185.62%)
FFs	6	47931 (8.74%)	93108 (16.99%)	47483 (8.66%)
	7	98210 (17.92%)	97682 (17.82%)	51192 (9.34%)
	8	100924 (18.41%)	101692 (18.55%)	55641 (10.15%)
RAMB18	6	311 (17.05%)	311 (17.05%)	311 (17.05%)
	7	311 (17.05%)	311 (17.05%)	311 (17.05%)
	8	311 (17.05%)	311 (17.05%)	311 (17.05%)

For a 3x3 mask size, there are three possible parallelism alternatives from the slowest to the fastest one: a single multiplier, 3 multipliers (one per mask column) or 9 multipliers. As it is shown in the table, the usage of resources is not proportional to each one of the described options. In the case of a single multiplier, the usage of logic LUTs is smaller, but, however, the multiplier implementation is the more expensive one in terms of hardware. If we analyze the FFs, the results are rather good for the 9 multipliers implementation. The results obtained with respect to the block RAMs do not vary with respect to the number of multipliers or the weight size. These results suggest that the low level synthesis process performed by Vivado performs some type of optimization which cannot be easily controlled by the user, and deeper analyses must be performed. Given that the main computational element of the convolution layers is the convolution kernel, an efficient design of this component is essential to minimize the use of logical resources of the DNN Thus, in the following paragraphs the contribution of this operator to the hardware resources usage is analyzed in deep.

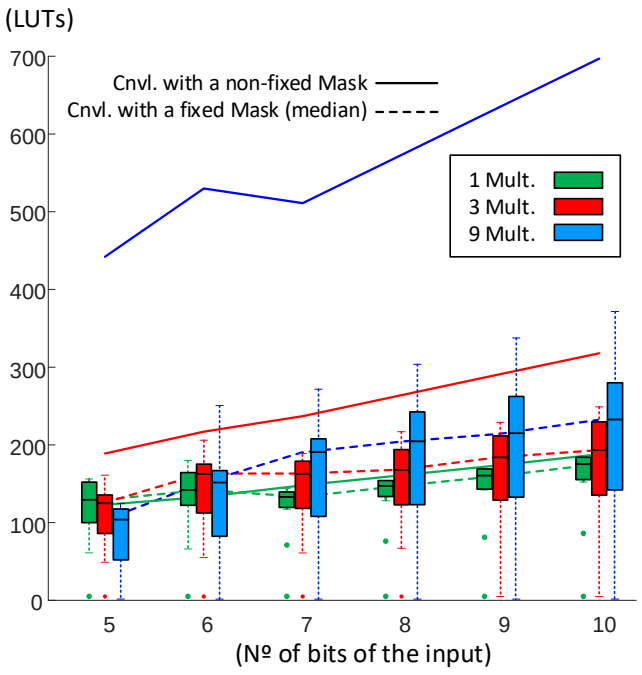

Fig. 5. Implementation results of convolution kernels for the three proposed parallelization alternatives and for the convolution kernels with non-fixed mask.

Fig. 5 shows the number of LUTs used to implement a 3x3 convolution kernel for the three proposed parallelization alternatives and different input data sizes. For reference, the figure also includes the implementation results of convolution kernels with non-fixed masks, i.e. when the synthesis tool cannot use the mask constants to simplify and optimize the arithmetic/logic circuits of the convolution kernel.

For the case of convolution kernels with a non-fixed mask (three solid lines), the graph shows that, in general, the trend in the use of LUTs for each implementations is proportional to both the number of bits in the input and the number of multipliers, with a slight increase observed for the case of 6-bit inputs. Experimental data show that this increase is related to the maximum number of inputs of the LUTs of the FGPA. It has been found that when the number of bits of the convolution input coincides with the maximum number of inputs of the LUTs (6), the possibility of using simultaneously the outputs O5 and O6 of the LUTs decreases, which results in a worse use of the LUTs.

For the case of convolution kernels with fixed masks, the number of LUTs needed for each implementation will depend on the value of the mask coefficients, i.e. the synthesis tool may optimize the convolution logic depending on the mask constants, e.g. a constant equal to 0 or 2 may eliminate the need to use a multiplier or may replace it by a shift operation. This optimization, however, affects the different parallelization alternatives unevenly. Since these results cannot be described by a single curve, a box-and-whisker plot created by implementing more than 200 different convolution masks has been used. The dashed lines show the trend in the number of LUTs (median) required to implement each of the parallelization alternatives. These central values, in general, follow a trend proportional to both the number of bits of the input and the number of multipliers used, although with a lower slope than in the case of convolutions with non-fixed masks.

TABLE II. NETWORK LATENCY

Clock frequency	**Number of parallel multipliers**	***Latency (µs)***
100 MHz	1	92.035
	3	37.855
	9	10.805
100 MHz / 300 MHz (CDC)	1	51.374
	3	51.294
	9	51.267

The figure shows that, for the case of 6-bit inputs, especially for convolutions with 1 and 3 multipliers, the number of LUTs required increases slightly again for the same reasons discussed above.

For the convolution kernel with a single multiplier, the trend of the central values remains slightly below the case with non-fixed masks. In this case, the interquartile range is the smallest and most stable of the three, appearing some outliers that correspond to masks whose elements are all zero, one, etc.

The figure shows that for the other two convolution kernels (with 3 and 9 multipliers) the interquartile range and the distance with respect to the version with non-fixed masks increases with the number of multipliers used. The figure shows that the option that best exploits the optimization performed by the synthesis tool is the version with 9 multipliers, while the option that least exploits it is the version with 1 multiplier. This result seems logical considering that the convolution kernel with 9 multipliers does not use multiplexers and therefore each multiplier can be optimized for a specific constant. The kernels with 3 and 9 multipliers making it more difficult for the synthesis tool to find a common factor that optimizes the multiplication. It is even possible that with a proper choice of convolution masks, the implementation with 9 multipliers will use fewer LUTs than an implementation with 1 or 3 multipliers.

Regarding the latency, Table II shows the figures achieved for different number of parallel multiplier used. The results show that for a fixed frequency (100 MHz) the latency is approximately inverse proportional to the number of multiplier, as it could be expected. However, when a CDC circuit is used to combine a 100 MHz input with a 300 MHz output, the results are different. Although a speed up is achieved for a single multiplier with respect to a single 100 MHz clock, the results achieved when using 3 or 9 multiplier are worse. This is due to the fact that the CDC requires additional clock cycles to convert data from one domain to the other and this extra latency is comparable to the own latency of fast circuits as this is the case when parallel multipliers are used.

IV. Conclusions

This paper has described a tool to transform high level descriptions of DNN models to hierarchical register transfer level descriptions based on VHDL. The tool allows some tradeoff between hardware resources usage and network latency, through the use of parallelism of the multiplier operator used in both convolutional and fully connected layers and allowing the specification of both the activation and the weights size. However, since the tool is aimed to be useful for different target technologies, such us FPGAs, SoCs or ASICs, it relies on low level synthesis tools, which take as input the created VHDL netlists and perform place and route operations. Thus it cannot perform a fully parameter space exploration which would lead to optimized designs and because of that the translation is based on a module library.

To solve this, each module in the library can be characterized in order to obtain its resource usage or silicon area for different target technologies, as well as its power dissipation besides its latency. This would provide a rough estimation of the whole power dissipation of the neural network and its silicon occupation. However, our tests have shown that this would be only an estimation, since the synthesis of some operators, like the multipliers, depends on the weights values. In this sense networks with a large amount of null weights or with a large number of null bits will occupy less silicon area that others with large number of '1's. Thus, it is difficult to find a priori a relationship between the number of layers or the number of channels of a DNN and its silicon area or number of resources in the case of an FPGA. In this latter case, the discrete distribution of resources within the CLBs yields to a more complex estimation of the overall usage of resources. An alternative is to include the low level synthesis tools within the translation loop, including an automatic evaluation of the final synthesized design in order to guide the optimization.

V. Acknowledgements

This work has been partially funded by Spanish Ministerio de Ciencia e Innovación (MCI), Agencia Estatal de Investigación (AEI) and European Region Development Fund (ERDF/FEDER) under grant RTI2018-097088-B-C33.

References

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

[2] https://pytorch.org/

[3] https://keras.io/

[4] https://caffe.berkeleyvision.org/

[5] J. Chen and X. Ran, "Deep Learning With Edge Computing: A Review," in Proceedings of the IEEE, vol. 107, no. 8, pp. 1655-1674, Aug. 2019, doi: 10.1109/JPROC.2019.2921977.

[6] X. Zhang et al., "DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs," 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2018, pp. 1-8, doi: 10.1145/3240765.3240801.

[7] S. Hadjis and K. Olukotun, "TensorFlow to Cloud FPGAs: Tradeoffs for Accelerating Deep Neural Networks," 2019 29th International Conference on Field Programmable Logic and Applications (FPL), 2019, pp. 360-366, doi: 10.1109/FPL.2019.00064.

[8] M. Riazati, M. Daneshtalab, M. Sjödin and B. Lisper, "DeepHLS: A complete toolchain for automatic synthesis of deep neural networks to FPGA," 2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2020, pp. 1-4, doi: 10.1109/ICECS49266.2020.9294881.

[9] K. Vipin, "ZyNet: Automating Deep Neural Network Implementation on Low-Cost Reconfigurable Edge Computing Platforms," 2019 International Conference on Field-Programmable Technology (ICFPT), 2019, pp. 323-326, doi: 10.1109/ICFPT47387.2019.00058.

[10] https://www.hdfgroup.org/

[11] D.M. Chapiro. Globally Asynchronous Locally Synchronous Systems. PhD thesis, Stanford University, 1984.

[12] S. Houben, J. Stallkamp, J. Salmen, M. Schlipsing and C. Igel, "Detection of Traffic Signs in Real-World Images: The German Traffic Sign Detection Benchmark," The 2013 International Joint Conference on Neural Networks (IJCNN), Dallas, TX, USA, 2013, pp. 1-8.

[13] https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[14] https://www.xilinx.com/products/design-tools/vivado.html