Neural Nets on FPGA a Machine Vision Algorithm Applied On MNIST Dataset Using Hls4ml Library

Fabrizio Alfonsi[1], Alessandro Gabrielli[1], and Elisabetta Ronchieri[2(✉)]

[1] Department of Physics, University of Bologna, Bologna, Italy
[2] INFN CNAF, Bologna, Italy
elisabetta.ronchieri@cnaf.infn.it

Abstract. In this paper we describe a machine vision Neural Net al.gorithm implemented in a FPGA. The algorithm is trained on a hand written digit MNIST dataset. For Neural Net Intellectual Property generation it is used the *hls4ml* library, which is a really powerful tool for fast implementation of Neural Net on FPGA.

Keywords: FPGA · Neural network · Machine learning

1 Introduction

The *hls4ml* [1,2] is a library that acts as a bridge between machine learning based on CPU/GPU (such as Keras, Tensorflow and PyTorch) [3], and VHSIC Hardware Description Language (VHDL) for designing FPGA's Neural Net Intellectual Property (NN IP) cores for fast inference. This library exploits High Level Synthesis (HLS), a way of synthesizing hardware from a pseudo-C++ code. *hls4ml* in fact automatically writes the HLS code that corresponds to the specified NN: it needs a json file for the architecture and a hdf5 file for weights.

Figure 1 shows the *hls4ml* workflow that is characterized by three phases. In the first phase, our workflow chooses a model and trains it on CPU by using Keras; it interactively, or just at the end, compresses the model by pruning weights; it exports and saves model's architecture in a json format and it weights in hdf5 format. In the second phase, our workflow generates a HLS project using *hls4ml*, it tunes the configuration of data type used by the model and it generates an IP core by using *hls4ml*. This task is carried out using the Vivado High-Level Synthesis (HLS) [4]. In the third phase, our workflow constructs a Vivado project [5] on the firmware design by importing the NN IP core.

In order to generate a HLS project, *hls4ml* needs to complete a yaml configuration file. The following *kindex_keras_config.yml* file is an example: XilinxPart contains FPGA's name [6], ClockPeriod sets the default clock period in ns, PrecisionL sets the default type and precision of numbers in NN processing and ReuseFactor represents the maximum number of times of reusing resources that increases the latency.

This work was supported by the University of Bologna and INFN.

© Springer Nature Switzerland AG 2020
O. Gervasi et al. (Eds.): ICCSA 2020, LNCS 12253, pp. 597–605, 2020.
https://doi.org/10.1007/978-3-030-58814-4_46

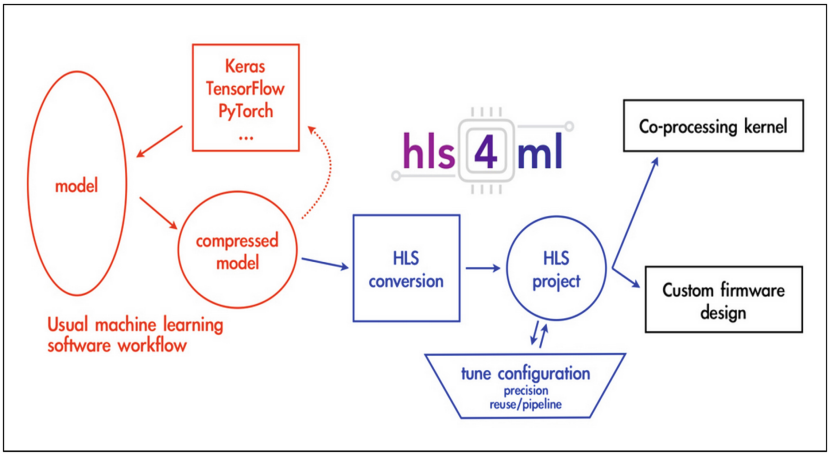

Fig. 1. hls4ml three-phase workflow

```
# Filename: kindex_keras_config.yml
KerasJson: My_NN/model_MNIST_flat8_5bit_pruned.json
KerasH5: My_NN/model_weights_MNIST_flat8_5bit_pruned.h5

OutputDir: Generated_NN/KINTEX/MNISTflat8_5bit_MinPrecision_v2
ProjectName: MNISTflat8_5bit_MinPrecision_v2

#kintex kc705
XilinxPart: xc7k325tffg900-2
ClockPeriod: 5

IOType: io_parallel #options: io_serial/io_parallel

HLSConfig:
  Model:
    PrecisionL ap_fixed<24,1>
    ReuseFactor: 1
```

2 Selected Model

The *hls4ml* library is an ongoing work in progress. Since updates with new capabilities and bug resolution are under development as require a long and detailed verification. This is why 2D convolutional NN is not supported. In order to work with images, one workaround is to flatten images into arrays and implement classical feed forward NN with dense layers.

By experiencing with capabilities of Vivado HLS to generate *hls4ml* cores, we have realized that nets with more than 100 nodes per layer cause a crash, because of several Vivado's critical warnings and excessive memory usage. We observed that the crash sometimes originated from a complete RAM saturation. In our experiments we used an 8 GB RAM PC and we used images of size 8×8 or 64 input nodes. MNIST dataset is a benchmark for machine vision tasks by ensembling single handwritten digits images in 8-bit gray scale of size 28×28 pixels. MNIST dataset is composed of 60'000 images for training and 10'000 for test/validation.

In order to satisfy the *hls4ml* limitations we decided to preprocess images by cropping the central portion of size 22 × 22 pixels and after that down sampling (e.g.. *scipy.ndimage.zoom()* in Python) to decrease the resolution to 8 × 8 pixels.

The choice of transforming the color depth of pixels from 8-bit to 5-bit (see Fig. 2) has been made to reduce the PC resource consumption: on one hand for keeping the NN IP core as small as possible and on the other hand because these images were shown on a screen using the VGA port on the board featuring the FPGA implemented for the NN. This board used a 6-bit green and 5-bit red and blue colour coding. In this way we have reached a reasonable accuracy of image reconstruction. In other words, increasing in color depth at constant model causes on increase in test accuracy but with a 5-bit color depth we can reasonably say that we have almost reached the maximum possible accuracy.

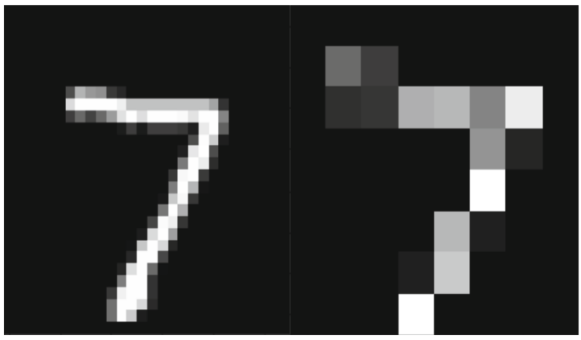

Fig. 2. Comparison between original images and processed images used to train the model: on the left there is the original image taken from the MNIST dataset (28 × 28 pixels, 8-bit color depth); on the right there is the processed image (8 × 8 pixels, 5-bit color depth).

After some experimentation with different architectures we realized that the two hidden layers in Fig. 3 show an excellent accuracy on both training set and test set as shown in Fig. 4, proving its capabilities of generalizing well on new data. Figure 4 shows almost the same accuracy on both sets, which means that the net has learned the maximum amount of information without overfitting training data.

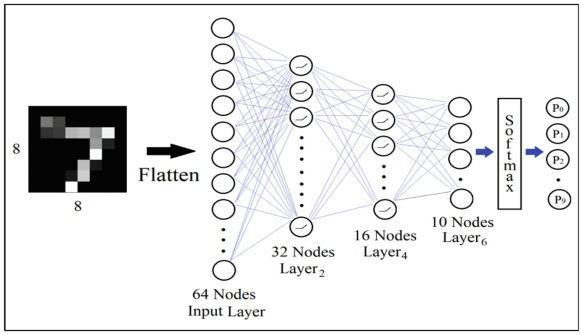

Fig. 3. Representation of model's architecture: symbols inside nodes are shown representing activation function (i.e. relu).

3 Training and Compression

After a training phase of 15 epochs the accuracy of the model respect to training data has already reached the maximum of almost 94%. One epoch occurs when

all example of training set are presented at least one time to the net for training purpose.

When the model is trained we implement the compression. The *hls4ml* library is optimized for avoiding the implementation of unnecessary operations while writing the output HLS code, like multiplications per zero, with net effect of reducing model's hardware resource request. To take advantage of this feature we have done the so called pruning, i.e. artificially setting to zero weights smaller than, in absolute sense, a certain cutoff; of course this has been done after studying the resulting accuracy to avoid wrong outputs.

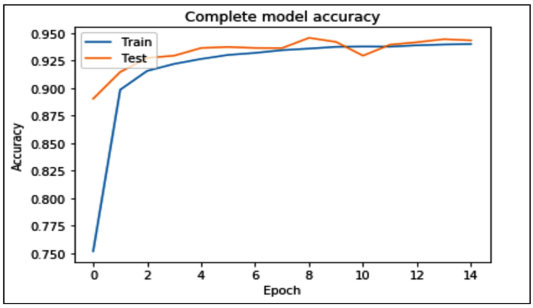

Fig. 4. Model accuracy obtained for the training and test samples.

Figure 5 shows that the cut-off used is equal to 2^{-5}. Furthermore we can notice that the shape of this histogram resembles an half Gaussian distribution and this is compatible with the fact that weights of the pruned model are almost a half with respect to the weights of the complete one (see Table 1). In terms of FPGA resources usage we obtained a reduction of almost 50% of the necessary multiplications.

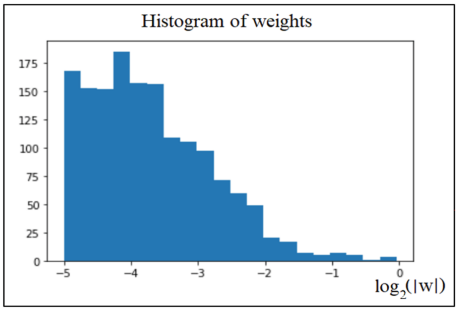

Fig. 5. Histogram of absolute values of non-null weights of the trained model after pruning.

Table 1. Overview table of compression. Complete list of parameter for every layer (weights plus biases) pre and post pruning.

	Parameters	
	Complete model	Pruned model
Layer_2	2'048 + 32	979 + 30
Layer_4	512 + 16	363 + 15
Layer_6	170 + 10	130 + 10
TOTAL	2'778	1'527

The accuracy is not greatly affected by the pruning and, in fact, with the training dataset the accuracy was 0.9389 in the complete model, while it was 0.9392 in the pruned model. For the test dataset, accuracy was 0.9471 in the complete model, while it was 0.9423 in the pruned model. Eventually the model is saved and exported the model in order to generate the HLS project through *hls4ml*. The PrecisionL parameter in the configuration file (see the *kindex_keras_config.yml* file in Sect. 1) is tuned as described in Sect. 4.

4 Implementation on FPGA

Once we have our NN HLS project, the fine tuning process on the data set takes place in order to fit our goal, consume less resource as possible and avoid overflow. The most widely used data type in HLS is an arbitrary precision fixed point numbers **ap_fixed<width, integer_size>** as shown in Fig. 6.

First of all, we focus our attention on weights (generically called w) and biases: $0.03125 = 2^{-5} \leq | \ w \ | \leq 0.97244$. Therefore, we need just 2 bits as integer size I_s (one for sign and one for 0) and at least 5 bits as fractional size $F_s = -log_2(min| \ w \ |max\imath Delta_{rel}) = -log_2(2^{-5}2^{-3}) = 8$. If we choose 5 bits as fractional size we badly quantize our parameter's space, indeed we would than have a maximum relative error of 1.

In order to better represent our space of parameters we choose a maximum relative error $\max\{\imath Delta_{ref}\}$ of $2^{-3} = 0.125$ or in other words 8 bits as fractional size. So for weights and biases should be enough **ap_fixed<10, 2>**.

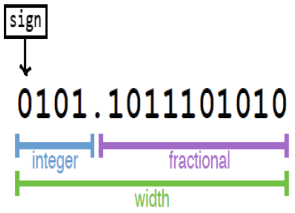

Fig. 6. **ap_fixed<14, 4>** with 14 as total data bits and 4 as the integer part of the number.

Our NN returns probabilities due to the final Softmax operation (see Fig. 7). Predicted classes probabilities are, as always, less than or equal to 1. Therefore, 2 bits as I_s are enough and we choose for the last layer **ap_fixed<16, 2>**.

As input we choose the data type **ap_int<6>** and consider the sign. In order to fine tuning data types for hidden layers we have to calculate the maximum value that each node can take; this is necessary to prevent overflow. For this purpose we introduce a new quantity for the i-th layer A_i which is the maximum acceptable node numbers per layer. A_i is simply defined as the product of the maximum weight and the $A_j \cdot \#\text{node}_j$ of the incoming layer: because we are using relu as activation function, A_i for relu layers has the same value as the incoming layer.

– Layer_2

$$A_2 = max| \ w \ | \cdot A_{Inputs} \cdot \#node_{Inputs} < 1 \cdot 2^5 \cdot 2^6 = 2^{11} \tag{1}$$

$$\Rightarrow I_{s,2} = 1 + log_2(A_2) = 12 \tag{2}$$

– Relu_3

$$A_3 = A2 \Rightarrow I_{s,3} = I_{s,2} \tag{3}$$

– Layer_4

$$A_4 = max| \ w \ | \cdot A_3 \cdot \#node_3 < 1 \cdot 2^{11} \cdot 2^5 = 2^{16} \tag{4}$$

$$\Rightarrow I_{s,4} = 1 + log_2(A_4) = 17 \tag{5}$$

– Relu_5

$$A_5 = A_4 \Rightarrow I_{s,5} = I_{s,4} \tag{6}$$

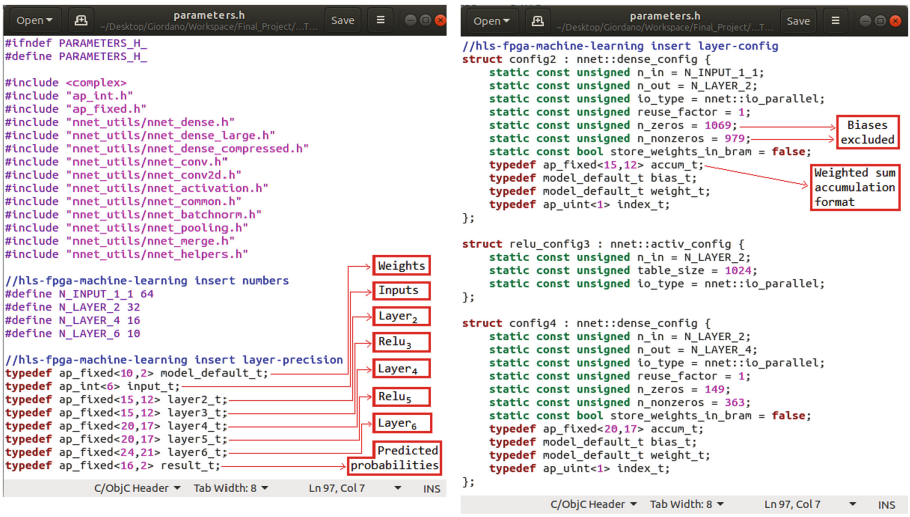

Fig. 7. In order to fine tuning the NN we have to actively modify the file *OutputDir//rmware//parameters.h*.

– Layer_6

$$A_6 = max|\ w\ | \cdot A_5 \cdot \#node_5 < 1 \cdot 2^{16} \cdot 2^4 = 2^{20} \tag{7}$$

$$\Rightarrow I_{s,4} = 1 + log_2(A_5) = 21 \tag{8}$$

We choose to assign as fractional size the value of 3 for all hidden layers: $F_{s,i} = 3$ for $i = 2, 3, 4, 5, 6$.

In the next Table 2 we show how data type fine tuning change demand for resources.

Table 2. Data type fine tuning.

NN IP core version	LUTs	FFs	BRAMs	DSPs
Not-tuned	58'109	58'628	25	512
Tuned	34'766	19'163	25	513

The not-tuned version is simply obtained setting, in the configuration file, the maximum required precision for weights and hidden layers: **ap_fixed<29, 21>**, which corresponds to $I_s = 21$ and $F_s = 8$. Data type of inputs and predicted probabilities are tuned as usual.

4.1 Lab Implementation

We use two FPGA boards (see Fig. 8): a *Kintex KC705* for NN IP core processing and a *Zybo Z701* to display input images on a screen through VGA port.

The second board is used to monitor the system. Indeed this board mounts a XILINX ZYNQ-7000 FPGA, a 28 nm device with available 17600 Look-Up-Tables and 36000 Flip Flops, which are not enough to apply high density computational systems. The ARM processor features with two cores at 800 MHz,

32 kB of Level-1 cache memory and 512 MB of RAM memory. The process can implement a custom Linux system to control all the mounted components on the board and any logic system implemented in the Programmable Logic FPGA, using only high level software. The XILINX Kintex-7 (always a 28 nm technology) works as computational station of the setup.

The 203800 Look-Up-Tables, the 407600 Flip Flops and the 840 Digital Signal Processors allow a complex algorithmic structure to be implemented, even at high speed due to the dense connections of all the logic components. The average frequency implementable in this kind of chips is up to 300 MHz. These two FPGAs are in the entry level of the 7 generation XILINX FPGAs. Today FPGAs have approximately 10 times higher numbers in terms of logic components, and many projects, including CERN ones, are planning to use them in the next years of High Energy Physics as real-time fast computational systems.

We designed the firmware of both FPGAs and 100 images were used from MNIST test set. The first 10 pictures correspond to digits 0,1,..., 9; while the others were randomly chosen. In this case the accuracy of our model, calculated with Keras and applied to this set of pixel numbers was as good as 97%. This result is very challenging for human capabilities, because it's really difficult to recognize these digits on some of these low-resolution images.

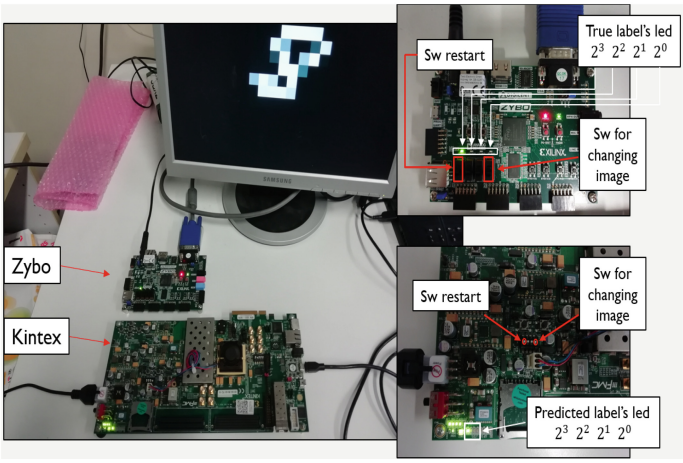

Fig. 8. Implementation settings in laboratory. Switches control the changing in input images. Leds on Zybo prints out true label and leds on Kintex the predicted ones.

4.2 Computation Time

We have evaluated the computational time needed by the NN to process an image. The inference time on FPGA, i.e. 125 ns, is relative to a Kintex kc705 FPGA, which has a 5 ns clock: in fact, this NN's elaboration takes 25 clock cycles. The inference time required on CPU, i.e. 1'982'269 ns is relative to a Intel(R) Core(TM) i7-8550U CPU @ 1.80 GHz 1.99 GHz, running Keras routines.

Figure 9 shows the Vivado tool interface which allows monitoring the behavior of the internal signals of the applied logic. The interface, called "waveforms table", has on the left column the name of the signal and on the right their behavior during the run time, with time going from left to right. In this implementation version (see Fig. 9) switches sw[:] controls the input image to the NN. sw = 0 gives an image of a zero, sw = 1 gives an image of a one and so on. The NN IP core is initially in idle mode. The inference process is triggered by **ap_start** signal in concurrence of the **input1_V_ap_vld** signal which is the signal that validates the input (**image_flat**). After 25 clock cycles the NN IP core returns a valid output in concurrence with the **ap_done** signal. The predicted class probabilities are than processed and the class label predicted is visible through led signal.

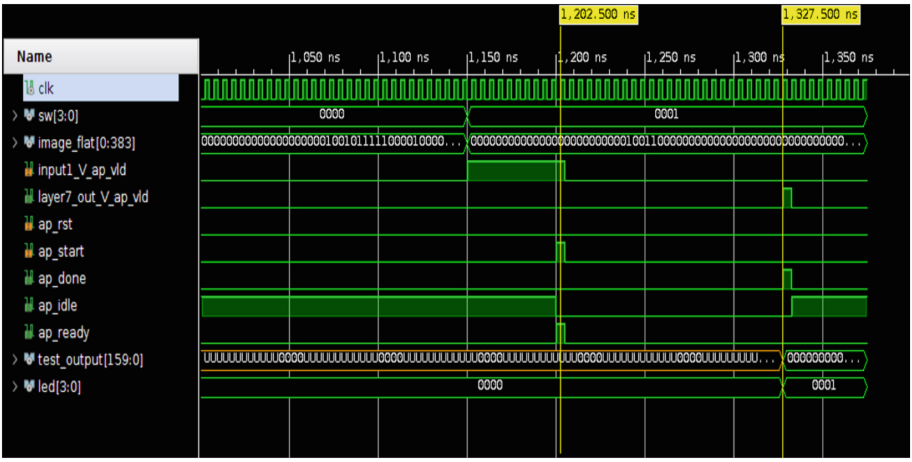

Fig. 9. Vivado simulation for measuring the inference time or latency of the NN IP core.

The Vivado interface proves the NN IP core capability of pipelining inference. The NN is fed with two valid input images with a time delay smaller than latency and the NN IP core is still able to process both. So, latency of the NN IP core is not a limiting speed factor and therefore this is a really interesting feature in order to apply this kind of hardware for triggering event mechanism such as pattern recognition in particle physics. We finally want to point out that the ReuseFactor parameter in the *hls4ml* configuration file has no effect, in term of resource usage, when applied to data type fine tuned NN IP cores. Benefits from this parameter instead can be seen when applied to not-tuned NN IP cores, but this will necessary cause an increase in latency.

5 Conclusion

The future challenges in terms of High-speed Data and Big Data are directing in the so called "hardware heterogeneity", which means the concurrent usage of several type of device genres as CPU, GPU, FPGA and custom ASIC(Application Specific Integrated Circuit), to solve data computing tasks. The implementation of "math" algorithms has almost always been implemented by using not-real-time devices as CPUs and GPUs, hardware architectures with great computational capability but with very low control in terms of latency. Future tasks as trigger structures of the biggest physics experiments at CERN [7] will require a latency control at ns level, not achievable by CPU or GPU. In the last years many groups and laboratories started to study FPGA and ASIC computational algorithm implementation [8], and the work shown in this paper wanted to demonstrate the general performance achieved with the implementation of a machine vision neural net on an FPGA. Other than the accuracy, the latency compared to the one reached by the used CPU demonstrates the capability of hardware implemented computational algorithm. The Bologna group is now working on exploiting our FPGA developing experience, acquired even with the type of task showed above, to study tracking algorithms for the ATLAS trigger, FPGA implemented.

Acknowledgment. The authors thank Giordano Calvanese for his collaboration in this work.

References

1. hls4ml: Software Open Access hls-fpga-machine-learning/hls4ml: v0.2.0 (2020). https://doi.org/10.5281/zenodo.3734261
2. hls4ml: hls4ml - GitBook. https://fastmachinelearning.org/hls4ml/
3. Duarte, J., Han, S., Harris, P., et al.: Fast inference of deep neural networks in FPGAs for particle physics. JINST 13 P07027 (2018). arXiv:1804.06913
4. HLS, V.: Vivado Design Suite User Guide - High-Level Synthesis. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf (2018)
5. Vivado.: Vivado Design Suite User Guide - Getting Started. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug910-vivado-getting-started.pdf (2018)
6. FPGA: KC705 Evaluation Board for the Kintex-7 FPGA - User Guide. https://www.xilinx.com/support/documentation/boards_and_kits/kc705/ug810_KC705_Eval_Bd.pdf (2019)
7. ATLAS collaboration. https://cds.cern.ch/record/2285584?ln=it (2017)
8. Shojaii, R.S.: An associative memory chip for the trigger system of the ATLAS experiment (2016). https://doi.org/10.22323/1.287.0058