Accelerating Machine Learning inference using FPGAs: the PYNQ framework tested on an AWS EC2 F1 Instance

M Lorusso[1,2], D Bonacorsi[1,2] and R Travaglini[2]

[1]Alma Mater Studiorum - University of Bologna, Viale Berti-Pichat 6/2, Bologna, Italy
[2]INFN - Bologna division, Viale Berti-Pichat 6/2, Bologna, Italy

E-mail: marco.lorusso11@unibo.it

Abstract. In the past few years, using Machine and Deep Learning techniques has become more and more viable, thanks to the availability of tools which allow people without specific knowledge in the realm of data science and complex networks to build AIs for a variety of research fields: in the context of High Energy Physics, new algorithms based on ML are being tested for event selection in trigger operations, end-user physics analysis, and more. Time critical applications can benefit from implementing algorithms on low-latency hardware like specifically designed ASICs and programmable micro-electronics devices known as FPGAs.

In order to facilitate the translation of ML models to fit in the usual workflow for programming FPGAs, a variety of tools have been developed. One example is the HLS4ML toolkit, developed by the HEP community.

This paper presents and discusses the activity started at the Physics and Astronomy department of University of Bologna and INFN-Bologna devoted to preliminary studies for the trigger systems of the CMS experiment at the CERN LHC accelerator. A broader-purpose open-source project from Xilinx (a major FPGA producer) called PYNQ is being tested combined with the HLS4ML toolkit.

Even though a rich documentation can be found on how to use hls4ml, a comprehensive description of the entire workflow from Python to FPGA is still hard to find. This work tries to fill this gap, presenting hardware and software set-up, together with performance tests.

1. Field Programmable Gate Array

Field Programmable Gate Arrays (FPGAs) [1, 2] are specialized devices that that implement circuits just like hardware, delivering significant power, area, and performance advantages over software. Furthermore, these devices can be easily and inexpensively reprogrammed to handle a broad range of tasks. Creating an FPGA-based circuit involves configuring the memory bits that control each routing decision with the necessary values, which is accomplished by generating a bitstream to load into the device. This typically involves starting with an application written in a hardware description language (HDL), such as VHDL or Verilog. However, in this study, a "higher-level" approach was employed using tools and libraries that enable FPGA design to be completed from a *behavioral description* written in C++ or, in the case of neural networks, in Python.

1.1. AWS EC2 F1 Instance

To evaluate the effectiveness of the implementation workflow proposed in this study, Amazon Web Services' EC2 F1 instances with Xilinx FPGA acceleration cards were employed. These F1 instances come with various tools to facilitate the development, simulation, debugging, and compilation of hardware acceleration code, such as an FPGA Developer Amazon Machine Image (AMI) that supports various development environments suited for both low-level hardware developers and software developers who are more comfortable with C/C++ and OpenCL environments. When the FPGA design is finished, it can be registered as an Amazon FPGA Image (AFI) and deployed to any F1 instances required.

2. The Implementation of a NN on FPGA

Improving the transverse momentum measurement performed by the Compact Muon Solenoid (CMS) muon Level-1 trigger, namely the momentum resolution, is very important to achieve a reduction of trigger rates. In fact, due to the rapidly decreasing shape of the inclusive muon p_T spectrum, even a relatively small reduction of the resolution can provide a significant decrease of the trigger rate at a given p_T threshold, by reducing the number of low momentum muon candidate misidentified as high momentum ones. This becomes particularly pressing in the context of future upgrades of CMS, in view of the High Luminosity LHC upgrade, to avoid using higher momentum thresholds as luminosity increases, with the consequence of losing physics acceptance.

By implementing an Artificial Neural Network (NN), a class of Machine Learning (ML) algorithms, on an FPGA, this work places itself in the search for ways to make the p_T prediction faster, other than more accurate.

```
1  import tensorflow as tf
2  from qkeras.qlayers import QDense, QActivation
3
4  netinputs = tf.keras.layers.Input(shape=(27,),dtype=X_train.dtype)
5  x = QActivation(activation=quantized_relu(16,6,relu_upper_bound=6.0),
6                  name='qrelu1')(inputs)
7
8  x = QDense(35, kernel_quantizer=quantized_bits(16,5,alpha=1),
9              bias_quantizer=quantized_bits(16,5,alpha=1),
10             kernel_initializer='random_normal',name='qdense_1')(x)
11 x = QActivation(activation=quantized_relu(16,6), name='qrelu2')(x)
12
13 #...# List of layers and activation functions
14 output = tf.keras.layers.Activation('softmax', name='soft1')(x)
15
16 model = tf.keras.Model(inputs=netinputs,outputs=netoutput,name="model")
17 model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
18 history = model.fit(X_train, Y_train, epochs=num_epochs, validation_data=(X_test
       , Y_test))
```

Listing 1: Building a Fully Connected Neural Network using QKeras.

2.1. The Model

The model built for this research is the next iteration of the Fully Connected Multilayer Perceptron regressor designed for my previous work in [2, 3]. Its purpose was to find an alternative algorithm to perform transverse momentum (p_T) assignment to muons in the context of the Level-1 trigger at the CMS experiment at CERN. This NN has been implemented with the following structure: the first hidden layer has 35 neurons and receives the information directly from the input layer of 27 different features with the ReLU (Rectified Linear Unit) selected as activation function. The second layer is identical to the first one but contains 20

neurons and this is repeated for other 4 additional hidden layers with 25, 40, 20 and 15 neurons, respectively. In the end, the output layer (with only one node) closes the network. A snippet of code describing how to create such network is in Listing 1. The model has been optimized for hardware implementation with *pre-training quantization* [4] and *weight pruning*.

2.2. The Implementation

The first step required for the implementation of a Neural Network on an FPGA is the conversion of the high-level code used for the creation of the model (Python + Tensorflow & QKeras) into High Level Synthesis (HLS) code. HLS describes the process of automatic generation of HDL code from *behavioural description* contained in a C/C++ script. To accomplish this task, the *hls4ml* package [5] has been used. This tool has been developed by members of the High Energy Physics (HEP) community to translate ML algorithm, built using frameworks like TensorFlow2, into HLS code. The process of setting up the conversion step using *hls4ml* is demonstrated in Listing 2. First, a configuration dictionary is created, allowing for the customization of per-layer settings. Next, the target hardware for the firmware is specified to set up the conversion. Finally, the project can be compiled, which involves the creation of relevant folders and scripts, and can be built if necessary by performing synthesis using Vivado HLS.

```
1 import hls4ml
2
3 config = hls4ml.utils.config_from_keras_model(model, granularity='model')
4 hls_model = hls4ml.converters.convert_from_keras_model(model,
5             hls_config=config, part='<id of FPGA model>')
6 hls_model.compile()
7 hls_model.build(csim=False,synth=False)
```

Listing 2: Example of simple configuration and use of the hls4ml library.

Once the target hardware has been defined, and the trained model converted into HLS code using *hls4ml* (more details are available in [2, 3]), the project has to be imported in *Vitis*, a tool part of the Xilinx Design Suite, dedicated to developing applications for data center acceleration cards. Here the C++ code must be tweaked in order to expose the interface of the Neural Network and make it compatible with the *Application Acceleration development flow*, offered by Vitis.

Then, we can instruct Vitis to build the entire project targeting the desired hardware. This will produce a bitstream file used to flash our design onto the FPGA. Together with the firmware design, an OpenCL application can be written that can be launched on the machine that houses the FPGA to program it, start the inference and retrieve the results (as shown in the next section).

Moreover, to deploy a design on Amazon's F1 instances, the bitstream must be uploaded to an Amazon S3 Bucket and request the creation of an *Amazon FPGA Image* (AMI) using a script included in the official github repository of the AWS EC2 FPGA Hardware Development Kit [6]. This will produce a `awsxclbin` file that can be used to program Amazon's FPGAs.

2.3. The PYNQ Project

PYNQ [7] is a project developed by Xilinx®, a leading FPGA manufacturer, which offers a Python API-based framework for utilizing Xilinx platforms and AWS-F1 instances via a Jupyter-based interface.

FPGA designs are represented as Python objects referred to as *overlays*, which can be accessed via a Python API. Although creating a new overlay still requires skilled developers with experience in designing programmable logic circuits, overlays are designed to be configurable and re-used as much as possible in various applications, much like software libraries.

Traditionally, C or C++ have been the most common embedded programming languages. Python, on the other hand, raises the level of programming abstraction and increases

programmer productivity. These options are not mutually exclusive, however. PYNQ employs CPython, which is written in C and can be extended with optimized C code while also integrating thousands of C libraries. Whenever possible, the more productive Python environment should be employed, and lower-level C code can be utilized whenever efficiency demands it.

```
import pynq
ov = pynq.Overlay("model_binary.awsxclbin")
nn = ov.myproject
```

Listing 3: Programming and calling kernel function using PYNQ.

PYNQ strives to work on any computing platform and operating system, which it accomplishes by utilizing a web-based architecture that is also browser-agnostic. The open-source Jupyter notebook infrastructure is used to execute an Interactive Python (IPython) kernel and a web server directly on the ARM processor of an MPSoC or the host CPU of an acceleration card.

```
auto devices = xcl::get_xil_devices();
auto fileBuf = xcl::read_binary_file(binaryFile);
cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
OCL_CHECK(err, context = cl::Context({device}, NULL, NULL, NULL, &err));
OCL_CHECK(err, q = cl::CommandQueue(context, {device}, CL_QUEUE_PROFILING_ENABLE
    , &err));
OCL_CHECK(err, cl::Program program(context, {device}, bins, NULL, &err));
OCL_CHECK(err, krnl_vector_add = cl::Kernel(program, "vadd", &err));
```

Listing 4: OpenCL code to programme a FPGA.

In Listing 3 how simple it is to load a firmware on an FPGA and retrieve the kernel function inside the design is shown. This can be considered equivalent to the code in Listing 4, which is much less straightforward. Furthermore, to actually send and receive data from the FPGA and run the algorithm, the code in Listing 5 is needed. On the other hand, using PYNQ the creation of input and output buffer is done by calling the `allocate` function, which returns objects that behave like `numpy` arrays and can be moved to and from the device with `sync_to_device()` and `sync_from_device()`. Finally, the kernel function is callable providing the buffers, for example: `nn.call(input,output)`.

```
OCL_CHECK(err, l::Buffer buffer_in1(context,
    CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, vector_size_bytes,
    source_in1.data(), &err))

OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_input}, 0 /*0 means from
     host*/,NULL,&eventinp));

OCL_CHECK(err, err = myproject.setArg(0, buffer_input));
OCL_CHECK(err, err = myproject.setArg(1, buffer_output));
//[...]
OCL_CHECK(err, err = q.enqueueTask(myproject,NULL,&eventker));
// wait for all kernels to finish their operations
OCL_CHECK(err, err = q.finish());

OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_output},
CL_MIGRATE_MEM_OBJECT_HOST));
```

Listing 5: OpenCL code to create I/O buffers and call the kernel function of the FPGA firmware.

3. Neural Network model performance on FPGA

Two main aspects have been considered to study the performance of using the PYNQ package to carry out Neural Network inference on an FPGA: latency and inference accuracy.

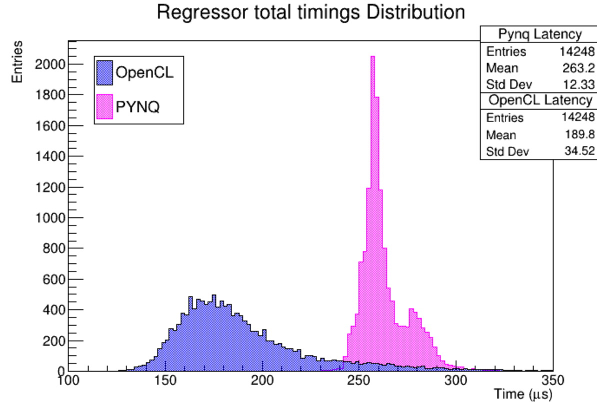


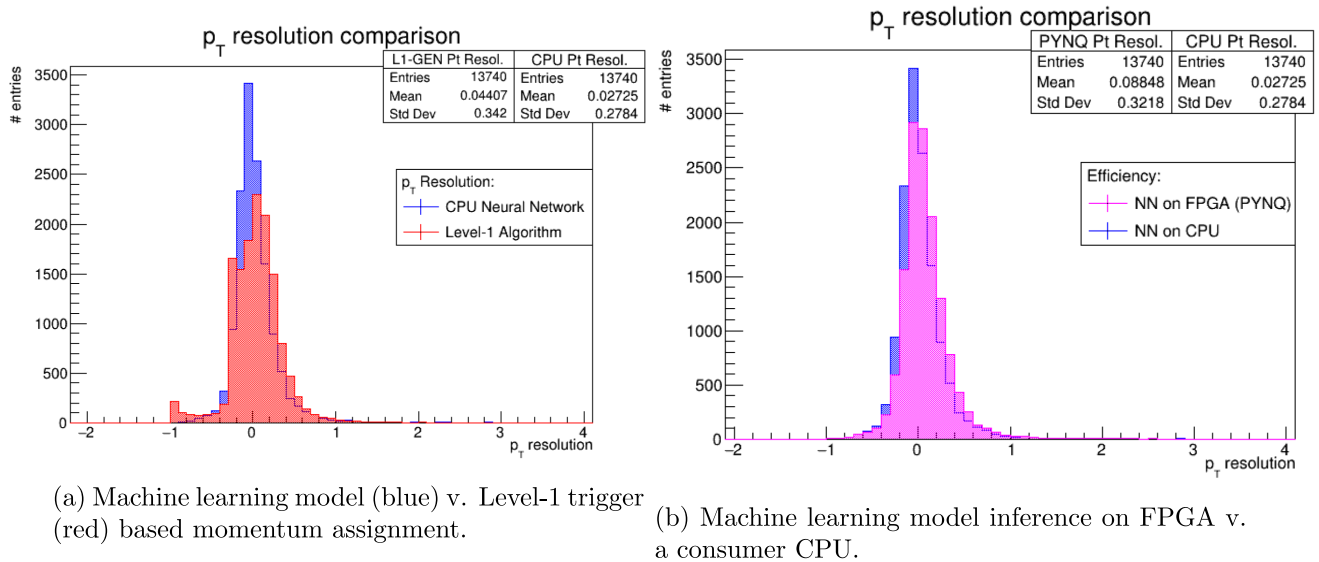
Figure 1: Total inference time distribution (input injection + inference + output extraction) using PYNQ (pink) and an OpenCL application (blue).

For the first metric, the *wall* time was measured for three main tasks that the host-FPGA pair performs for each requested inference. The three tasks are input injection on the FPGA card, actual inference, and output extraction. In the PYNQ case, a certain degree of consistency can be observed between the execution times of these tasks. This consistency can be attributed to a common overhead caused by Python's interpreted nature. This overhead is also the main reason for the overall longer total processing time of the PYNQ implementation compared to the C++ application. The total inference time distribution is shown in Figure 1, where the overall lower times using OpenCL can clearly be seen.

Nonetheless, the main objective of using PYNQ is offering an easier interface and less steep learning curve in dealing with accelerating algorithms using FPGAs. This means that, to achieve the full potential of this type of hardware, the traditional approach using C/C++ application is still the way to follow.

3.1. p_T resolution histogram

The accuracy of the NN model implemented on the F1 instance was studied using p_T resolution histograms. For each dataset entry, the histograms were built using $\frac{\Delta p_T}{p_T} = \frac{p_{T_{est}} - p_{T_{sim}}}{p_{T_{sim}}}$, where $p_{T_{est}}$ is the estimation of the transverse momentum and $p_{T_{sim}}$ is the "true" transverse momentum associated with each validation set entry. Firstly, the resolution of the model before implementation on the FPGA was checked, with the red histogram representing the resolution distribution of the Level-1 trigger system and the blue histogram showing the resolution of the predictions made by the network model running on a consumer CPU (Figure 2a). The ML resolution had a less broad distribution, resulting in an overall improvement with respect to the Level-1 trigger system. The Machine Learning based momentum assignment is also less prone to large p_T underestimation. After verifying the accuracy of the NN model, its implementation on the FPGA available in the F1 instance was analyzed. The p_T resolution histogram obtained by performing the inference using the PYNQ environment is shown over the model resolution described before in Figure 2b. Slightly worse results were produced when the assignment was performed on an FPGA, with a small bias towards higher values of $\Delta p_T/p_T$. This could be the effect of the loss in precision the input features have to go through due to the conversion to fixed-point representation needed to perform computations efficiently in an FPGA. Nevertheless, the hardware approach still appears compatible or, in case of higher momenta, even better than the Level-1 trigger based momentum assignment.



(a) Machine learning model (blue) v. Level-1 trigger (red) based momentum assignment.

(b) Machine learning model inference on FPGA v. a consumer CPU.

Figure 2: Transverse momentum resolution histograms.

4. Conclusions

The combination of the HLS4ML toolkit and the PYNQ open-source project from Xilinx, a leading FPGA producer, was utilized in this work to program a Neural Network on an FPGA and perform inference. PYNQ enables designers to take advantage of the benefits of programmable logic and microprocessors using the Python language. Cloud computing resources were utilized to evaluate the capabilities of this workflow, from creating and training a Neural Network to generating an HLS project using HLS4ML, testing predictions of the NN using PYNQ APIs and functions written in Python.

The hardware and software set-up were evaluated along with performance. The results showed an increase in algorithm latency when using PYNQ compared to a more conventional approach of interacting with an FPGA via an OpenCL application. This latency increase can be attributed to Python's interpreted nature, resulting in additional overhead. Consistency between the Neural Network's predictions before and after implementation on the FPGA was validated.

References

[1] Hauck S and DeHon A 2007 *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation* (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.) ISBN 9780080556017

[2] Lorusso M *FPGA implementation of muon momentum assignment with machine learning at the CMS level-1 trigger* Master's thesis URL `http://amslaurea.unibo.it/23211/`

[3] Diotalevi T, Lorusso M, Travaglini R, Battilana C and Bonacorsi D 2021 Deep Learning fast inference on FPGA for CMS Muon Level-1 Trigger studies *Proceedings of International Symposium on Grids & Clouds 2021 — PoS(ISGC2021)* vol 378 p 005 URL `https://doi.org/10.22323/1.378.0005`

[4] Coelho C N, Kuusela A, Li S, Zhuang H, Ngadiuba J, Aarrestad T K, Loncar V, Pierini M, Pol A A and Summers S 2021 *Nature Machine Intelligence* **3** 675–686 ISSN 2522-5839 URL `https://doi.org/10.1038/s42256-021-00356-5`

[5] Duarte J, Han S, Harris P, Jindariani S, Kreinar E, Kreis B, Ngadiuba J, Pierini M, Rivera R, Tran N and Wu Z 2018 *Journal of Instrumentation* **13** P07027 URL `https://dx.doi.org/10.1088/1748-0221/13/07/P07027`

[6] Aws-fpga: Official repository of the aws ec2 fpga hardware and software development kit URL `https://github.com/aws/aws-fpga`

[7] Pynq introduction - python productivity for zynq (pynq) URL `http://pynq.readthedocs.io/`