PROCEEDINGS OF SCIENCE

Machine Learning inference using PYNQ environment in a AWS EC2 F1 Instance

Marco Lorusso,[a,b,*] **Daniele Bonacorsi,**[a,b] **Davide Salomoni**[a,b,c] **and Riccardo Travaglini**[a,b]

[a] *INFN Bologna,*
viale Berti Pichat 6/2, Bologna, Italy
[b] *Department of Physics and Astronomy, University of Bologna,*
viale Berti Pichat 6/2, Bologna, Italy
[c] *INFN CNAF,*
viale Berti Pichat 6/2, Bologna, Italy
E-mail: marco.lorusso11@unibo.it, daniele.bonacorsi@unibo.it, d.salomoni@unibo.it, riccardo.travaglini@bo.infn.it

In the past few years, using Machine and Deep Learning techniques has become more and more viable, thanks to the availability of tools which make the need of specific knowledge in the realm of data science and complex networks less vital to achieve a satisfactory final result in a variety of research fields. This process has caused an explosion in the adoption of such techniques, e.g. in the context of High Energy Physics. The range of applications for ML becomes even larger if we consider the implementation of these algorithms on low-latency hardware like FPGAs which promise smaller latency with respect to traditional inference algorithms running on general purpose CPUs.
This paper presents and discusses the activity running at the University of Bologna and INFN-Bologna where a new open-source project from Xilinx called PYNQ is being tested. Its purpose is to grant designers the possibility to exploit the benefits of programmable logic and microprocessors using the Python language and libraries. This new software environment can be deployed on a variety of Xilinx platforms, from the simplest ones like ZYNQ boards, to more advanced and high performance ones, like Alveo accelerator cards and AWS EC2 F1 instances. The use of cloud computing in this work lets us test the capabilities of this new workflow, from the creation and training of a Neural Network and the creation of a HLS project using HLS4ML, to testing the predictions of the NN using PYNQ APIs and functions written in Python.

International Symposium on Grids & Clouds 2022 (ISGC 2022)
21 - 25 March, 2022
*Online, Academia Sinica Computing Centre (ASGC), Taipei, Taiwan****

*Speaker

PoS(ISGC2022)001

1. Introduction

Machine Learning (ML) has become in recent years one of the pillars of computer and data science and it has been introduced in almost every aspect of everyday life and research fields alike. Currently, the spread of learning algorithms in many sectors finds its roots mainly in an increased quantity of data available, combined with a technological progress in storage and computational power, which can nowadays be delivered with lower maintenance and building costs.

In order to reach the full potential of ML algorithms, new computing solutions are being developed and tested like never before since the rise of the x86 architecture as the *de facto* standard for general purpose computing. This is done to find the perfect combination of fast prediction times and low energy consumption needed to deploy ML efficiently in a variety of use cases, from IoT devices to data centers applications and scientific research.

This work focuses on a specific type of hardware called Field Programmable Gate Array (described in Section 2) which promises low latencies and unprecedented power efficiency. 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, which allows the translation of Neural Networks built using tools like TensorFlow to a High-Level Synthesis description (e.g. C++) in order to implement this kind of ML algorithms on FPGAs. More details can be found in Section 4.

The analysis described in this paper concentrate on a new way to interact and retrieve results from FPGAs: PYNQ (Section 5). This Python package allows to use a simple Python script to program the FPGA and use the function included in its design in a similar way to usual function calls.

Finally, performance tests on a regressor model used as benchmark, will be presented in Section 6, where the consistency in the predictions of the NN with respect to using an OpenCL application, will be verified.

Summing up, this paper describes the work done to produce a complete and as simple as possible workflow to implement algorithms of interest to the HEP field, namely Neural Networks, on FPGAs. A case study from the CMS experiment at CERN was used as an example to test the different tools employed and as a benchmark to take some preliminary measurements regarding latency and accuracy of the algorithm.

2. Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) [1] are devices that blend the benefits of both hardware and software [2]. They implement circuits just like hardware, providing huge power, area and performance benefits over software, yet they can be reprogrammed cheaply and easily to implement a wide range of tasks. FPGAs implement computations spatially, simultaneously computing millions of operations in resources distributed across a silicon chip. Such systems can be hundreds of times faster than microprocessor based designs. However, unlike in ASICs, these computations are programmed into the chip, not permanently frozen by the manufacturing process. This means that an FPGA-based system can be programmed and reprogrammed many times.

The internal layout of an FPGA, shown in Figure 1, is made up of replicated units of digital electronic circuits, called logic blocks, embedded in a general routing structure, hence the gate and array in the name of this type of devices. The various kinds of logic blocks perform different functions, e.g. :

LookUp Tables (LUT) for simple combinational logic;

Flip-Flops (FF) for implementing sequential logic;

Digital Signal Processors (DSP) for efficient multiplication of fixed-point numbers.

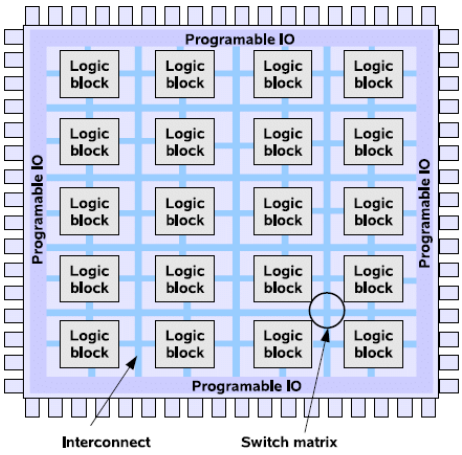


Figure 1: An abstract view of an FPGA. Logic cells are embedded in a general routing structure. [4]

With these predefined, fixed-logic units, which are fabricated into the silicon, FPGAs are capable of implementing complete systems in a single programmable device, by configuring the connections between these units to perform complex algorithms. Because customizing an FPGA involves storing values to the memory bits that control every routing choice, the creation of an FPGA based circuit is a process of creating a bitstream to load into the device. This is usually done starting with an application written in a hardware description language (HDL), such as VHDL or Verilog, however in this work a "higher-level" approach is followed, using tools and libraries that make it possible to finalize a FPGA design starting from a *behavioural description* written in C++ or, in the case of Neural Networks, in Python.

2.1 AWS EC2 F1 Instance

In order to test the capabilities of the implementation workflow presented in this work, cloud computing resources, more specifically Amazon Web Services' EC2 F1 instances [5], equipped with Xilinx FPGA acceleration cards, have been used. F1 instances are equipped with tools to develop, simulate, debug, and compile a hardware acceleration code, including an FPGA Developer Amazon Machine Image (AMI) and supporting hardware level development on the cloud.

Using F1 instances to deploy hardware accelerations can be useful in many applications to solve complex science, engineering, and business problems that require high bandwidth, enhanced networking, and very high compute capabilities. Examples of target applications that can benefit

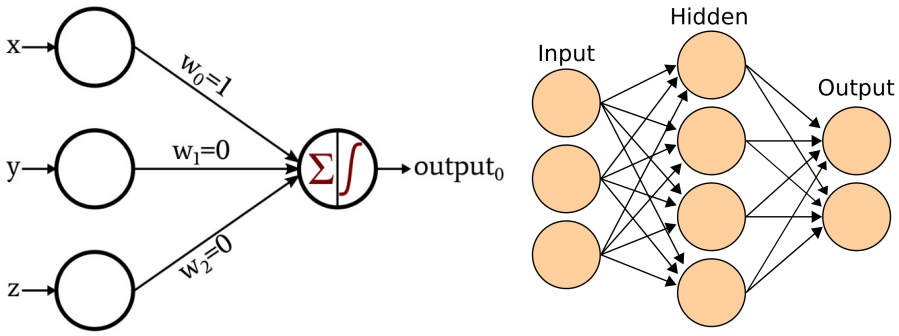
from F1 instance acceleration are genomics, search/analytics, image and video processing, network security, electronic design automation (EDA), image and file compression and big data analytics.

F1 instances provide diverse development environments: from low-level hardware developers to software developers who are more comfortable with C/C++ and OpenCL environments. Once an FPGA design is complete, it can be registered as an Amazon FPGA Image (AFI), and deployed to every F1 instance needed.

3. Artificial Neural Network

An Artificial Neural Network (ANN) is a learning algorithm vastly used in Machine and Deep Learning, inspired by the biological neural connections that constitute the human brain, specifically designed to tackle non-linear learning problems [6].

The network architecture chosen for this work is the Fully Connected Multilayer Perceptron [7], due to its relative simple design. MLPs, as the name suggests, are made up of single units called *Perceptrons*. Inside these units, as shown in Figure 2a, the input values $\mathbf{x} = (x, y, z)$ are multiplied by their weights $(w_i x_i)$. All of these are then added together to create the net or weighted sum $\sum_i w_i x_i$, which is given to the *activation function* $f(\sum_i w_i x_i)$ resulting in the perceptron's outputs, adding a non-linearity compared to the simple linear combination alone.



(a) Graphical representation of a Perceptron. **(b)** Neural Network basic layout.

Figure 2: Simple diagrams representing the layout and functioning principle of Neural Networks.

Perceptrons can be stacked together to make a layer of *neurons*, each producing its own outputs. These layers can then be put together to build arbitrarily deep custom networks, by feeding the outputs of a layer to the neurons of the next layer, which will be "hidden" to the user, and resulting in a structure like the one in Figure 2b.

3.1 Neural Network models

In this work two different NN models have been taken in consideration. The first consists in a pattern recognition classifier trained and tested using the Iris dataset from the UCI Machine Learning Repository [8]. Its structure sees an input layer, ingesting the 4 features included in the data for each entry, linked to a hidden layer made up of 16 nodes, then connected to the output layer with 3 nodes due to the 3 different classes the model can choose from. Thanks to the ready to use

dataset and relatively low training time required, this model was mainly used to quickly test the workflow for the implementation on the FPGA.

The second model built for this research is the next iteration of the 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 Compact Muon Solenoid 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. The results listed in this paper will focus mainly on this second, more complex and more realistic application.

Developing a neural network in a specific hardware is usually accompanied by a certain level of optimisation in terms of compression, to reduce the storage and computation costs for deep models. In this case this task is accomplished by performing *pre-training quantization* and *weight pruning*. The former consists in the conversion of the arithmetic used within the NN from high-precision floating-points to normalized low-precision integers (fixed-point) [9]. The latter is the elimination of unnecessary low values in the weight tensor, by practically setting the NN parameters' values to zero, which will be translated into "cut" connections between nodes of the NN, reducing the number of parameters and operations involved in the computation of each output.

4. Implementing a NN on FPGA

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 [10] 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. A schematic workflow of *hls4ml* is illustrated in Figure 3. The parts of the workflow illustrated on the left in red indicates the usual software steps required to design a neural network for a specific task. The blue section in the middle describes the task done by *hls4ml*, resulting in a HLS project that can be synthesized and implemented to run on an FPGA.

As shown in Figure 4, 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* [11], an application 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 *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).

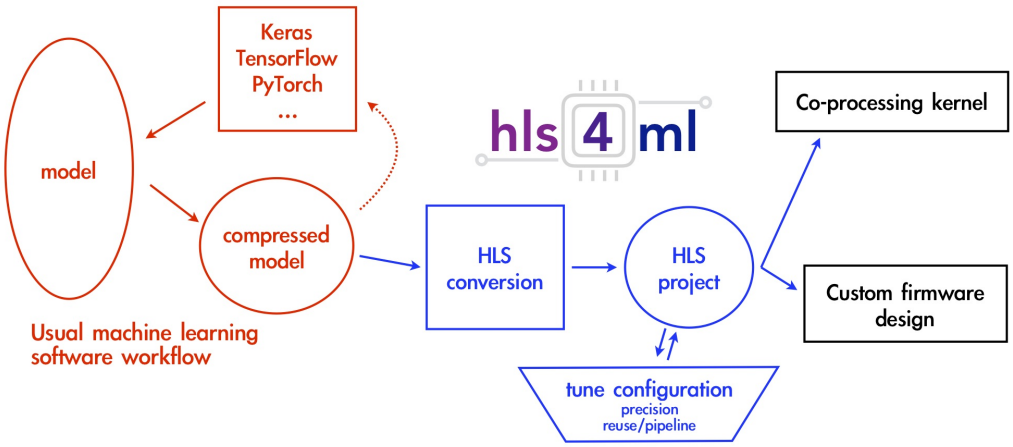


Figure 3: A typical workflow to translate a model into an implementable FPGA design using *hls4ml*.

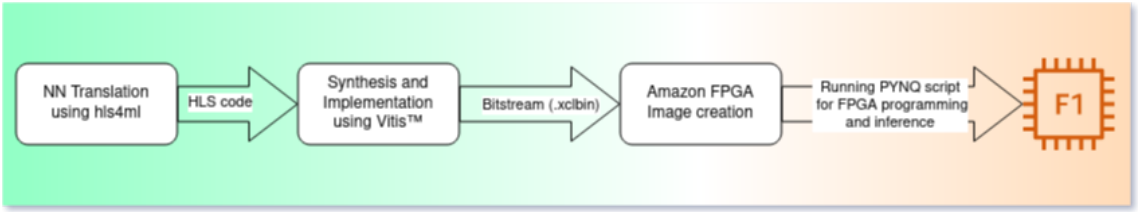


Figure 4: Deploying workflow targeting AWS EC2 F1 instances.

Moreover, to deploy a design on Amazon's F1 instances, the bitstream must be uploaded to an S3 Bucket [12] 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 [13]. This will produce a `awsxclbin` file that can be used to program Amazon's FPGAs.

5. The PYNQ project

PYNQ [14] is an open-source project from Xilinx®, a prominent FPGA producer. It provides a Jupyter-based framework with Python APIs for using Xilinx platforms and AWS-F1 instances.

FPGA designs are presented as Python objects called *overlays* that can be accessed through a Python API. Creating a new overlay still requires developers with expertise in designing programmable logic circuits. Overlays, like software libraries, are designed to be configurable and re-used as often as possible in many different applications.

To date, C or C++ are the most common embedded programming languages. In contrast, Python raises the level of programming abstraction and programmer productivity. These are not mutually exclusive choices, however. PYNQ uses CPython which is written in C, and integrates thousands of C libraries and can be extended with optimized code written in C. Wherever practical, the more productive Python environment should be used, and whenever efficiency dictates, lower-level C code can be used.

PYNQ aims to work on any computing platform and operating system. This goal is achieved by adopting a web-based architecture, which is also browser agnostic. It incorporates the open-

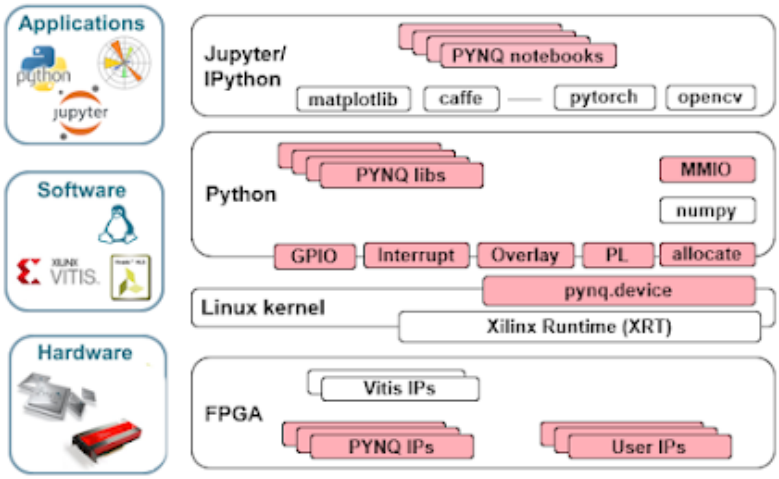


Figure 5: PYNQ's components in the different level of abstraction needed for running applications on FPGAs.

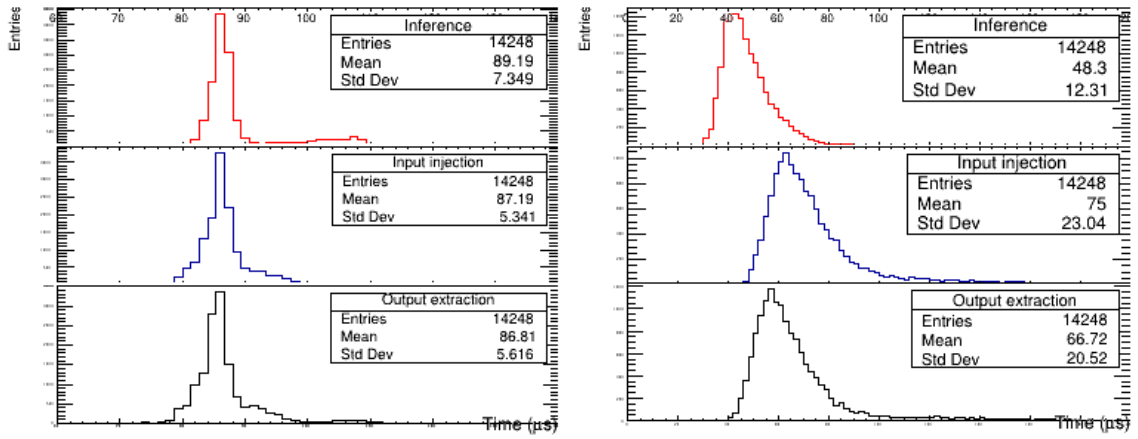
source Jupyter notebook infrastructure to run an Interactive Python (IPython) kernel and a web server directly on the ARM processor of a MPSoC or host's CPU of an acceleration card. The web server brokers access to the kernel via a suite of browser-based tools that provide a dashboard, bash terminal, code editors and Jupyter notebooks. The browser tools are implemented with a combination of JavaScript, HTML and CSS and run on any modern browser. PYNQ's main components are summed up in Figure 5.

A description on how to use PYNQ and a comparison with writing an OpenCL application can be found in Appendix A. By looking at both approaches, it is evident how writing Python code including the PYNQ package is less complicated than the alternative.

6. 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. As briefly explained in Section 3.1, the following analysis will focus only on the more advanced regressor model, due to the fact that the Iris model was used only to test the principle of operation and verify that this workflow could also be applied to classification algorithms, albeit this example being a very simple one.

For the first metric, the *wall* time has been measured for the three main tasks that are executed by the host-FPGA pair for each inference that is requested. In Figure 6 the time distribution for the input injection on the FPGA card (blue), the actual inference (red) and output extraction (black) is shown for the entire validation dataset using PYNQ on the left and the OpenCL application on the right. In the PYNQ case, a degree of consistency can be seen between the different tasks. This can be explained by a common overhead caused by Python's nature as an interpreted language, which can also be considered as the main cause for the overall larger total processing time, shown in Figure 7, with respect to the application compiled in C++.



(a) PYNQ **(b)** OpenCL

Figure 6: Distribution of the times needed to inject data in the FPGA, perform NN inference and extract the output using the PYNQ package in Python (left) and an OpenCL application (right).

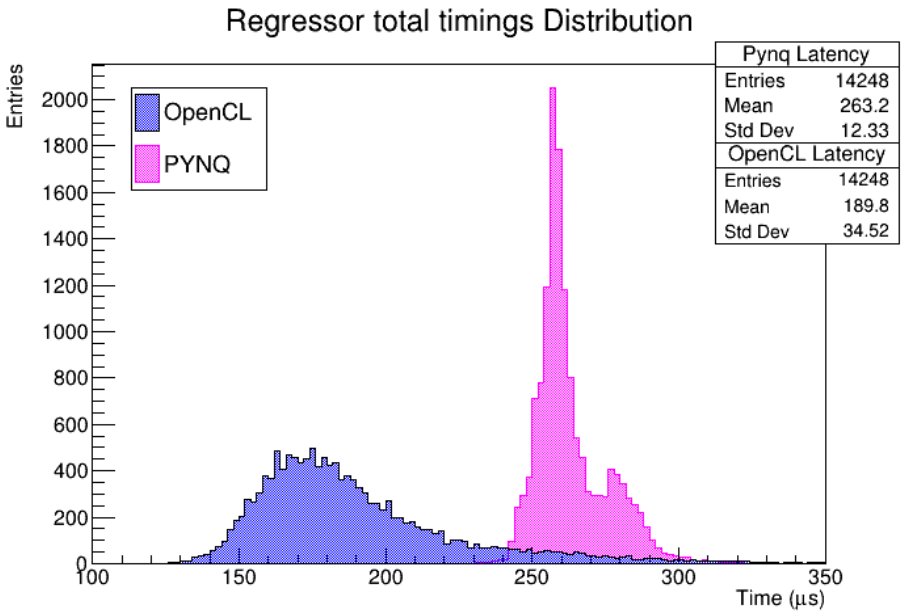


Figure 7: Total inference time distribution (input injection + inference + output extraction) using PYNQ (pink) and an OpenCL application (blue).

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.

6.1 p_T resolution histogram

To study the accuracy of the NN model implemented on the F1 instance p_T resolution histograms were used. For each entry of the dataset, the histograms were built using the following

relation:

$$\frac{\Delta p_T}{p_T} = \frac{p_{T_{est}} - p_{T_{sim}}}{p_{T_{sim}}} \tag{1}$$

where $p_{T_{est}}$ is the estimation of the transverse momentum, given by the model prediction or the actual algorithm used in the Level-1 trigger at CMS to perform this task, and $p_{T_{sim}}$ is the "true" transverse momentum associated to each entry of the validation set. Even though this metric makes a quick and easy to understand comparison possible, it is important to keep in mind that this resolution is asymmetric, i.e. its range can go from -1 to infinite. This means that, for a constant actual spread, the standard deviation associated to its distribution is affected by the value of its mean: the smaller it is, the smaller the standard deviation gets.

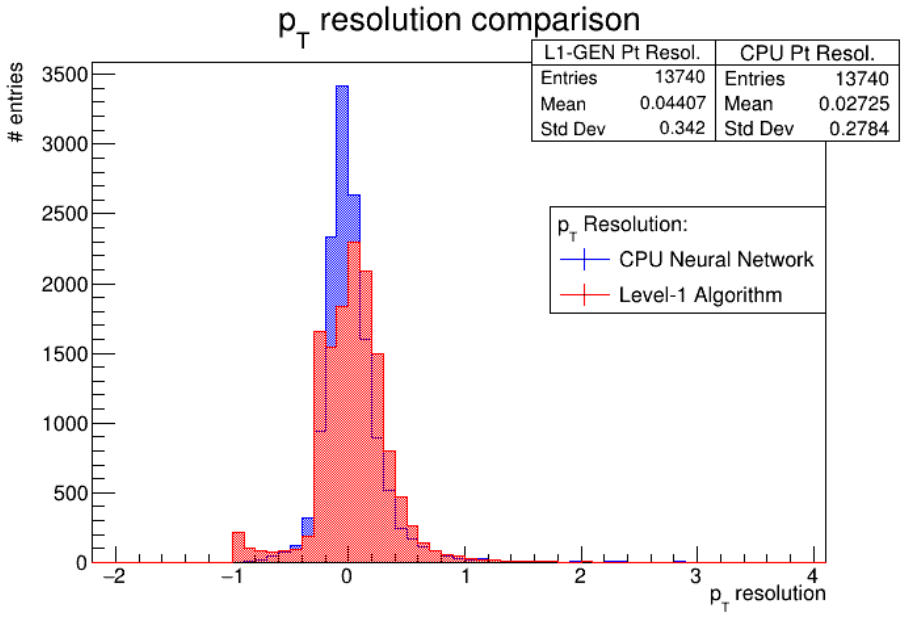


Figure 8: Transverse momentum resolution histograms computed for the machine learning model (blue) and Level-1 trigger (red) based momentum assignment.

Firstly, the resolution of the model before the implementation on the FPGA must be checked (Figure 8). The red histogram describes the resolution distribution of the Level-1 trigger system while the blue one shows the resolution of the predictions made by the network model running on a consumer CPU.

In particular, it is possible to notice a less broad distribution for the ML resolution, resulting in an overall improvement, yet small, with respect to the Level-1 trigger system. Another noticeable detail is the small peak corresponding to the value -1: this happens when the p_T assigned by the trigger is significantly underestimated with respect to the true p_T. The Machine Learning based momentum assignment is therefore less prone to large p_T underestimation.

Having verified the accuracy of the NN model, its implementation on the FPGA available in the F1 instance can be analyzed. In Figure 9 the p_T resolution histogram obtained by performing the inference using the PYNQ environment is shown over the model resolution described before. It is clear that the model infer momenta with a resolution distribution which is narrower when the

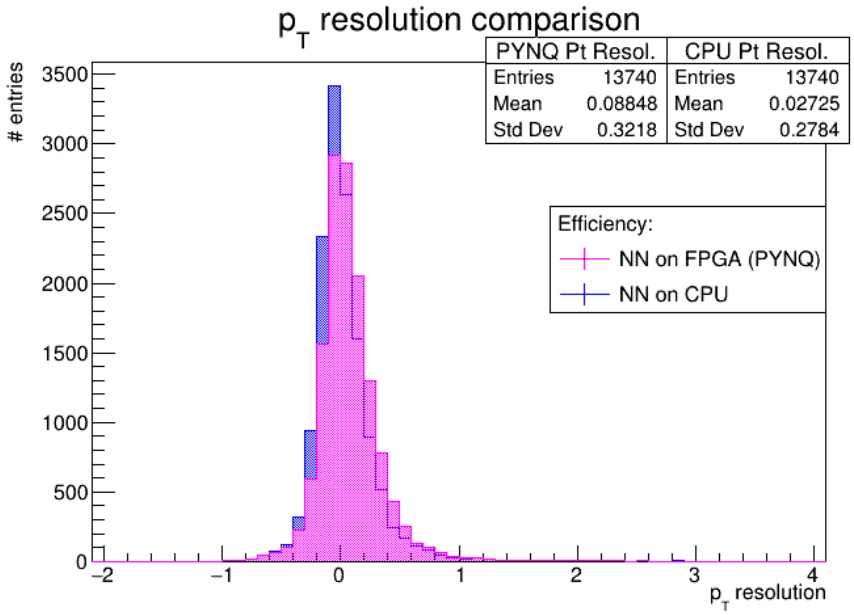


Figure 9: Transverse momentum resolution histograms computed for the machine learning model (blue) and Level-1 trigger (red) based momentum assignment.

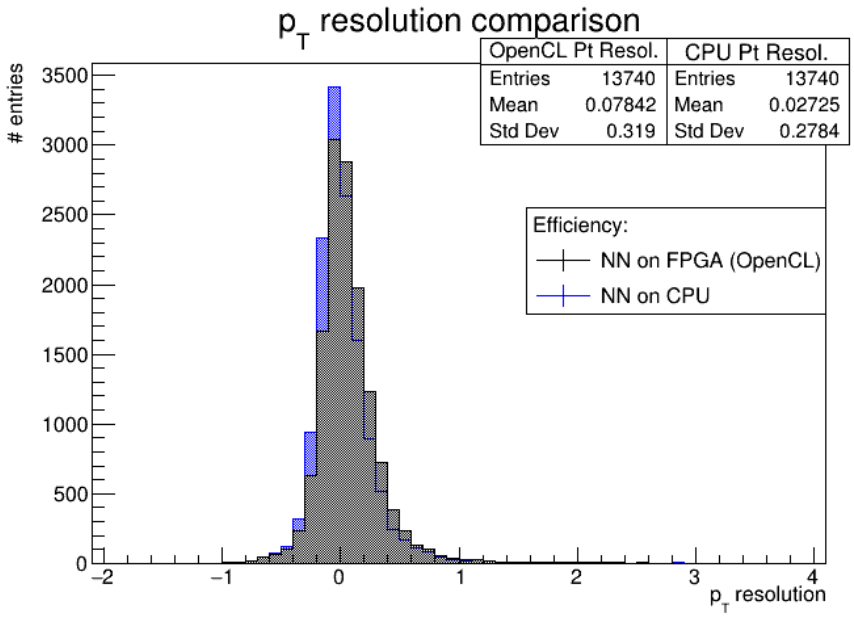


Figure 10: Transverse momentum resolution histograms computed for the machine learning model (blue) and Level-1 trigger (red) based momentum assignment.

computation is carried out on a CPU. When the assignment is performed on an FPGA, slightly worse results are produced, 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 [2, 3]. Nevertheless, the

hardware approach still appears compatible, or in case of higher momenta, even better than the Level-1 trigger based momentum assignment.

For a final comparison, in Figure 10 there is the resolution histogram obtained by performing the inference on the FPGA using an OpenCL application. As expected the result is very similar to the PYNQ one, however there is a small difference which can be explained by a different implementation of floating point numbers to fixed point precision conversion.

7. Conclusions

An open-source project from Xilinx (a major FPGA producer) called PYNQ has been tested, combined with the HLS4ML toolkit, in order to program a Neural Network on an FPGA and use it to perform inference. The PYNQ purpose is to grant designers the possibility to exploit the benefits of programmable logic and microprocessors using the Python language. Cloud computing was used in this work to test the capabilities of this workflow, from the creation and training of a Neural Network and the creation of a HLS project using HLS4ML, to testing the predictions of the NN using PYNQ APIs and functions written in Python.

Hardware and software set-up, together with performance, were tested. An increase in latency of the algorithm was discovered when using PYNQ with respect to a more traditional way of interacting with an FPGA via an application written in OpenCL. This can be explained by an overhead caused by Python's nature as an interpreted language. Consistency between the predictions of a NN before and after its implementation on the FPGA was verified. The inference results obtained using PYNQ were very similar to the OpenCL application ones, however there is a small difference which can be explained by a different implementation of floating point numbers to fixed point precision conversion.

As a next step for this study, Alveo accelerator cards are expected to be tested with the presented workflow, and a local server devoted to test NN in a fast, reliable and easy-to-use way will be assembled as part of the INFN Cloud catalogue.

References

[1] André DeHon Scott Hauck, *Reconfigurable computing: the theory and practice of FPGA-based computation*, Systems on Silicon, Morgan Kaufmann, 2007.

[2] M. Lorusso, *FPGA implementation of Muon Momentum assignment with Machine Learning at the CMS Level-1 Trigger*, University of Bologna master thesis (unpublished).

[3] T. Diotalevi, M. Lorusso, R. Travaglini, C. Battilana and D. Bonacorsi, *Deep Learning fast inference on FPGA for CMS Muon Level-1 Trigger studies*, PoS ISGC2021, 2021.

[4] A. Shawahna, S. Sait and A. El-Maleh, *FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review*, IEEE Access. PP. 1-1, 2018.

[5] https://aws.amazon.com/ec2/instance-types/f1

[6] S. Sandhya, *Neural networks for applied sciences and engineering. From fundamentals to complex pattern recognition* (2007).

[7] M. C. Popescu, V. E. Balas, L. Perescu-Popescu and N. Mastorakis, *Multilayer perceptron and neural networks*, WSEAS Trans. Cir. and Sys. 8, 7, July 2009, 579-588.

[8] D. Dua and C. Graff, *UCI Machine Learning Repository*, University of California, School of Information and Computer Science, 2019.

[9] A. Taylor, *The basics of FPGA mathematics*, Xilinx Xcell Journal 80, 2012.

[10] J. Duarte et al. *Fast inference of deep neural networks in FPGAs for particle physics*, In Journal of instrumentation 13.07, July 2008.

[11] https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

[12] https://aws.amazon.com/s3

[13] https://github.com/aws/aws-fpga

[14] http://pynq.readthedocs.io/

[15] https://github.com/Xilinx/Alveo-PYNQ/blob/master/pynq_alveo_examples/notebooks/1_introduction/4-opencl-comparison.ipynb

A. A PYNQ Crush Course

After this introduction, we can describe how to use the PYNQ package and compare this new approach against a "more traditional" way to deploy accelerated functions on FPGAs [15], namely an application written in OpenCL.

The first thing to do in both cases, is to program the device and initialize the software context. In the OpenCL version, this is achieved with the following code:

```
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, NNkernel = cl::Kernel(program, "myproject", &err));
```

In particular, the `get_xil_devices()` function finds the available Xilinx devices and return them as a list. Then, `read_binary_file()` loads the binary file (the `.xclbin` provided to the application) and returns a pointer to the loaded file, that is then consumed to initialize the `bins` object. A new OpenCL `context` is then created, that will be passed along the different functions as a handle. After thath, a command queue `q` is created, in order to send commands to the device. Then, the detected `device` is programmed, and finally the NN kernel include in the design is assigned to the `NNkernel` variable.

With PYNQ the same set of operations is achieved by instantiating a `pynq.Overlay()` object (the device is programmed at this stage), and then assigning the vector addition kernel to the `nn` variable, accessing directly the overlay.

```
import pynq
ov = pynq.Overlay("designbitstream.xclbin") #or .awsxclbin for F1 instances
nn = ov.myproject
```

In OpenCL host and FPGA buffers need to be handled separately. Therefore, we first have to create the host buffer, and only after that is done, we can instantiate the FPGA buffer, linking it to the corresponding host buffer:

```
std::vector<input_t, aligned_allocator<input_t>> sample(N_IN);
std::vector<result_t, aligned_allocator<result_t>> hw_results(N_OUT);

OCL_CHECK(err, cl::Buffer buffer_input(context,
            CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
            size_bytes_in, sample.data(), &err));
OCL_CHECK(err, cl::Buffer buffer_output(context,
            CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
            size_bytes_out, hw_results.data(), &err));
```

with `N_IN` and `N_OUT` the number of features in input for each sample and the number of outputs of the NN, respectively.

On the other hand, with PYNQ buffers allocation is carried out by `pynq.allocate`, which provides buffer object with the same interface as a `numpy.ndarray`. Host and FPGA buffers are managed under the hood, and the user is only presented with a single interface for both:

```
inp = pynq.allocate((27,1),'u2')
out = pynq.allocate((1,1),'u2')
```

The `enqueueMigrateMemObjects()` is used in OpenCL to initiate data transfers. The developer must specify the direction of the transfer as a function parameter. In this case, we are sending data from the host to the FPGA memory, therefore we need to pass `0` as direction for the input:

```
OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_input}, 0, NULL));
```

The same behavior is achieved in PYNQ by invoking the `.synq_to_device()` method of the input buffer:

```
inp.synq_to_device()
```

To run the kernel in OpenCL each kernel argument need to be set explicitly using the `setArgs()` function, before starting the execution with `enqueueTask()`:

```
OCL_CHECK(err, err = NNkernel.setArg(0, buffer_input));
OCL_CHECK(err, err = NNkernel.setArg(1, buffer_output));
// send data here
OCL_CHECK(err, err = q.enqueueTask(NNkernel,NULL));
OCL_CHECK(err, err = q.finish());
//retrieve data here
```

The `.call()` function is used instead in PYNQ to do everything in a single line. This function will take care of correctly setting the `register_map` of the IP and send the start signal:

```
nn.call(inp,out)
```

To retrieve data from the FPGA the `enqueueMigrateMemObjects()` is used again in OpenCL to initiate data transfer. In this case, the host code uses the `CL_MIGRATE_MEM_OBJECT_HOST` constant to specify the direction of the transfer:

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

Finally, the same is achieved with PYNQ by calling `.synq_from_device()` from the output buffer:

```
out.synq_from_device()
```