Expanding the testing infrastructure of High-Level Synthesis for Machine Learning (HLS4ML)

Author:
Gulnar Najafova
Baku State University
gulnarnajafovagn@gmail.com

Supervisor:
Vladimir Loncar
CERN
vloncar@ipb.ac.rs
vladimir.loncar@cern.ch

Abstract

HLS4ML - an open-source package based on High-Level Synthesis (HLS) is used for converting machine learning algorithms to utilize them in particle physics. It is designed to deploy neural network architectures on FPGA chips, targeting extremely low-latency requirements of the Large Hadron Collider (LHC). The main aim of the project is implementing an infrastructure for testing the conversion of Keras library with High-Level Synthesis for Machine Learning (HLS4ML). For building this infrastructure Pytest testing framework and Jenkins continuous integration platform was utilized. The designed testing infrastructure serves to maintain and to validate the consistency and the functionality of the software when new features are added by means of Keras library. The result of the work makes the project a more robust and mature framework, allowing the deployment of more complex neural network architectures. The testing infrastructure has to check the accuracy of model conversion, internal representation, HLS synthesis, and resource estimation.

Keywords *– hls, hls4ml, fpga, keras, machine learning testing, big data*

1. Introduction

The Compact Muon Solenoid (CMS) experiment [1] is a big, multipurpose particle detector, designed to explore an extensive range of physics at the Large Hadron Collider (LHC) [2]. Physicists smash particles together at high energies and record what happens with the CMS experiment. The CMS simulates particle collisions with a bunch of crossing rates of 40 MHz. The LHC accelerates protons to nearly the velocity of light before colliding them inside the CMS [2]. The energy of the colliding protons transforms into matter spraying particles in every direction. The CMS detector acts as a high-speed camera recording collisions for further analysis. It means that enormous amounts of data are generated during the process.

LHC sensors detect more than a billion collisions between particles. Some of the particles are so accelerated in the collider that they even reach 99.9999991% of the speed of light. It is clear that a large amount of data is generated during the collisions. The LHC alone generates about 80-90 petabytes of data in a year. This data is analyzed by algorithms that

serve to identify particles that CERN is looking for and difficult to capture. Since these particles appear and then disappear, they are identified based on the traces of energy they leave behind.

Particles are accelerated within 27 kilometers of the LHC ring and the four giant experiments (Atlas, Alice, CMS, LHCb) happen in it. In every second thousands of turns complete and up to a billion collisions occur. But only one out of a billion collisions is interesting to researchers. The fast electronic preselection passes one out of a billion events and stores them on computer memory and tens of thousands processor cores select one of the remaining events. In data centers, close to one hundred thousand processor cores are running.

To collect data the LHC uses light sensors - it records protons that are accelerated in the collider. The light energy is emitted during collisions by the colliders and converted into data and analyzed by computer algorithms. Most of this data is unstructured data and in the form of photos. The LHC's sensors deliver data millions of times per second and can be stored on thousands of physical disks. Normally, CERN sensors generate hundreds of terabytes of data per second but it is decreased to megabytes of data per second.

The capacity of data keeps growing day by day. Research continues and requires more resources. Initially, to reduce the number of events, to select interesting ones, and to get rid of noise from the data set physicists have been writing algorithms. Analysis process concentrates on the most significant events. The composition of storage capability, access patterns, unpredictable analysis workloads are the biggest challenges. To manage this scale of data effectively, CERN has used distributed data processing and revolutionized the data storage attitude.

CMS operates a Level-1 (L1) trigger to dismiss unnecessary events [3]. “Triggering” is filtering events to reduce data rates to manageable levels. This enacts a rate decline on the order of a factor of 400. At the time of construction, it would have been impossible to read-out the tracker data at 40 MHz, mainly because of the considerably great size of data and rate. To solve this issue Level-1 decision making comes to help. The trigger decision is made approximately in 10 μs in the second stage. Hardware - FPGAs are implemented in this stage too. To get strict latency constraints machine learning algorithms are deployed in very early stages - L1 Trigger and High-Level Trigger (Figure 1):

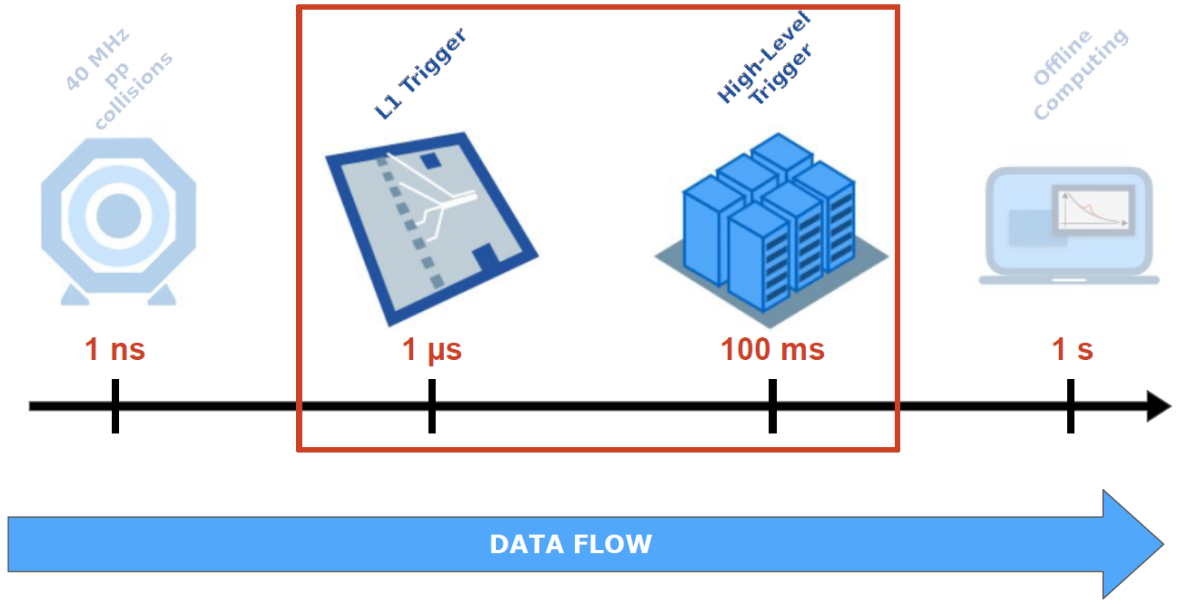

Figure 1. Data flow at the LHC and deployment of machine learning algorithms in early stages.

2. What is the Field-Programmable Gate Array and why is it used?

While machine learning has already been proven to be extremely useful in the analysis of data that is generated from detectors at the LHC, it is typically performed in an “offline” environment after the data is taken and agglomerated. However, one of the largest problems at detectors on the LHC is that collisions, or “events”, generate too much data to be saved. As such, filters called “triggers” are used to determine whether a given event should be kept or not. Using FPGAs allows for significantly lower latency, so machine learning algorithms can essentially be run “live” at the detector level for event selection. As a result, more events with potential signs of new physics can be preserved for analysis.

FPGAs have been used at CERN for several years in many applications [4]. FPGA is a set of integrated circuits on a chip. These integrated circuit strings can also be configured after the designer's manufacturing process is complete. A processor does its task by following the instructions sequentially. This means that the processor's operations are inherently limited: the desired functionality must be adapted to existing instructions, and in most cases it is not possible to perform multiple processing tasks at the same time. Dissimilar to the computer central processing unit FPGA chips operate basic instructions and process a wide range of tasks at the same time. In other words, it completes these tasks in parallel, and this is its main and unique feature. Small processing units allow working on a lot of parts of a task at the same time, so low latency and high efficiency can be achieved. Reprogrammability and parallel processing features make them suitable for machine learning-based applications.

The function of the FPGA can change every time the device is powered on. Therefore, when a design engineer wants to make a change, they can download a new configuration file to the device and try the change. It has an expensive and time-consuming hardware "functionality" of ASSPs and ASICs. Due to FPGA flexibility, original equipment manufacturers have the ability to ship systems as soon as the design is run and tested. FPGAs

effectively accelerate overall system performance by giving CPUs off-load and acceleration functions.

FPGAs can be programmed again and again in the field, without going to the manufacturing place. To solve the specific problems millions of logic gates in the chip (they are programmable) can be configured. Because of FPGAs' extensive usage in high-performance computing and the advantages to accelerate crucial processing in data centers, these chips are a leader among tech companies. Task-specific customization due to the computer architecture can be done on FPGA chips without changing their functionality.

FPGAs are regarded as more effective than other high-performance computing chips because of their parallel processing capability, and they use considerably low amounts of power per operation. FPGAs can also support high bandwidth inputs and outputs of up to about 100 dedicated high-speed serial links, making them ideal workhorses to process the deluge of data that streams out of particle detectors [5].

It is possible to produce an FPGA code with low latency for trigger applications with high-level tools. Its resource consumption emulates low-level code that is developed advanced [7]. Unlike the preceding projects that were implemented entirely in standard Hardware Description Languages (HDLs) the ‘hybrid’ solution is being developed with a High-Level Synthesis (HLS) language. It is developed by manufacturer Xilinx [6]. Recent collaborations specialize in the C++-based Vivado HLS tool of the FPGA. The study has the potential for innovative new tools to be used in future triggers, without notably increasing resource utilization and latency. This modification should allow for easier and faster algorithm development, quicker understanding of non-experts, and advanced long-dated maintainability.

The goal has been to fit complex Deep Learning algorithms in FPGA chips with limited capacity. To reach this goal a CERN-based project was developed and called "HLS4ML". This technology reduces algorithms and enables them to work on FPGA chips without any loss of performance or accuracy. It also enables decision-making algorithms to take place on chips within microseconds. With the use of such tools, non-FPGA experts can trade-off latency and resource usage – two critical metrics of performance, which would require significant extra effort to balance in collaboration with engineers writing low-level code.

3. What is High-Level Synthesis for Machine Learning?

High-Level Synthesis is a process that is designed to interpret an algorithmic description of a desired behavior automatically and create digital hardware that implements this behavior [9]. Synthesis starts from the high-level specification of the problem, where behavior is usually separated from low-level circuit mechanics. The code is analyzed, architecturally constrained, and scheduled to transpile into a register-transfer level (RTL) design in a hardware description language (HDL), which is in turn commonly synthesized to the gate level by the use of a logic synthesis tool. To allow hardware designers to build and confirm hardware efficiently is the main aim of HLS. For this it gives them effective control on the optimization of its architectural design and lets them describe the design at a higher level of abstraction during the implementation of RTL [8].

HLS4ML is an open-source package developed in Python. It is a firmware implementation of machine learning algorithms using HLS. In other words, traditional machine learning models were translated to HLS to use them on FPGAs. HLS4ML interprets neural networks written in frameworks like Keras, Tensorflow, QKeras, PyTorch, Onnx and converts them to a HLS representation. The HLS4ML package serves to reduce time that is taken to get results and give an opportunity to the user to design a suitable machine learning algorithm for the application perfectly [9]. Simultaneously, the user can balance performance, resource usage and latency requirements. The workflow of HLS4ML is illustrated below (Figure 2):

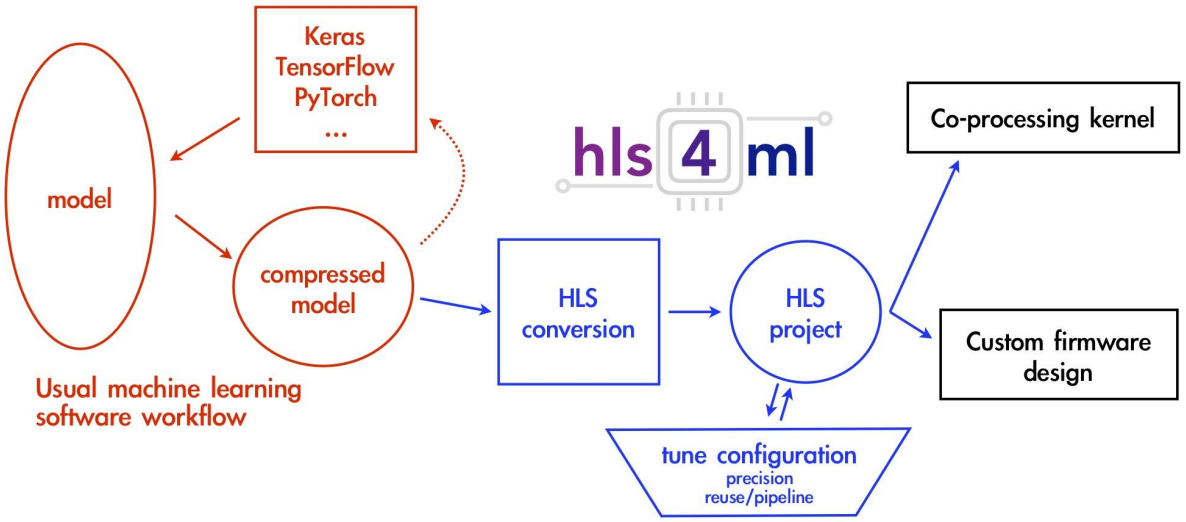

Figure 2. The workflow of HLS4ML project.

FPGAs also often come at a comparatively low power cost with respect to CPUs and GPUs. This allows HLS4ML to build HLS code from compressed neural networks that result in predictions on the microsecond scale for latency. Important configurations are the reuse factor and the model precision as well as the use of optimization techniques. The produced IP core can extend more complex designs or be used as a kernel for the CPU co-processing.

The user can define many of the parameters of their algorithm to find the best suit for their needs. The user can control characteristics of their model such as:

- **Size/compression:** This is an important optimization to use the FPGA resources efficiently like dropping unnecessary weights (zero or close to zero) to reduce the number of digital signal processing (DSP) used;
- **Precision:** Define the precision of calculations in model;
- **Dataflow/resource reuse:** Control the parallel or serial model implementations with varying levels of pipelining and the inference latency versus utilization of FPGA resources;
- **Quantization aware training:** Achieve the best performance at low precision with tools like QKeras, and benefit automatically during inference with HLS4ML parsing of QKeras models.

4. HLS4ML testing

It is very important to keep the functionality and consistency of the software when new features are added to the layers. Unit tests are needed to guarantee the correct execution of the conversion process of deep neural networks. Testing is the process of executing a program over a previously developed program with the aim of finding errors and bugs in it. The main aim of this work is expanding the existing testing infrastructure of the HLS4ML project. Starting from the existing testing infrastructure, inclusion of unit tests were worked on. These tests cover the whole HLS4ML usage pipeline.

Within the scope of expanding testing infrastructure of the HLS4ML project, tests have been implemented for the conversion of the Keras models to HLS models, resource estimation after the conversion process and internal representation to HLS synthesis (Figure 3).

```
======================== test session starts ========================
platform linux -- Python 3.6.9, pytest-6.0.1, py-1.9.0, pluggy-0.13.
1
rootdir: /home/gulnar/Documents/TestFolder/hls4ml-master, configfile
: pytest.ini
collected 45 items

test_keras.py .............................................  [100%]

======================== 45 passed in 30.20s ========================
```

Figure 3. Test session of the Keras API of the project.

The total of 55 tests (including the 10 tests for synthesizing the model with Vivado HLS) have been successfully executed over the part of the project that is developed with Keras.

During the testing process, the following attributes of the both Keras and HLS models are compared: accuracy, the number of layers in the whole model, input shape, output shape, sizes of attributes of one-dimensional, two-dimensional convolution layers and pooling layers, etc.

5. Testing of the HLS4ML developed with Keras library

Keras is a minimalist Python library for deep learning that runs on TensorFlow. Keras enables to accelerate experimentation cycles with its high-level simplicity. It is possible to implement any research ideas through its low-level flexibility.

Throughout the testing of HLS4ML both, the Sequential model and the Functional API of Keras were considered. The layers, like Core (Reshape, BatchNormalization, BinaryDense, TernaryDense) Convolution (Conv1D, Conv2D), Pooling (MaxPooling1D, MaxPooling2D, AveragePooling1D, AveragePooling2D), Activation (ELU, LeakyReLU, ThresholdedReLU, PReLU) have been implemented in the Sequential model. Merging layers like Add, Substract, Multiply, Average, Maximum, Minimum, Concatenate have been implemented on the Functional API, and Dense and Input layers were implemented on both APIs. The Functional API allows us to create more flexible models than the Sequential ones.

The Pytest framework provides the possibility to test the functions of the application easily. In all sets the builtin `@pytest.mark.parametrize` decorator is used and it allows one to define multiple sets of arguments and fixtures at the test function or class [10]. This decorator provides a way to use all the arguments that are given it. In the following example LeakyReLU and ELU activation functions are used during the run of the test (Snippet 1):

```
import pytest
from tensorflow.keras import LeakyReLU, ELU,
keras _activation_functions  = [LeakyReLU, ELU]
@pytest.mark.parametrize("activation_functions",
keras_activation_functions)
```

Snippet 1. Defining a set of the activation functions with the help of `@pytest.mark.parametrize` decorator.

All tests in the testing infrastructure are implemented in the form of the following set. All sets consist of the four functions:

- ➢ Creating the Keras model;
- ➢ Converting all functionalities of the Keras model to the HLS one;
- ➢ Testing the correctness of the conversion process;
- ➢ Predicting and comparing results of both models.

Creating Keras model: The Keras models are created from the Sequential() class or the Functional API, layers like Dense and activation are added, is compiled by choosing an optimizer and loss function and become ready to convert to the HLS model (Snippet 2):

```
import tensorflow.keras.models as models
def make_model():
      model = models.Sequential()
      model.add(Dense(2, input_shape=(1,), name='Dense'))
      model.add(Activation(activation='elu', name='Activation'))
      model.compile(optimizer='adam', loss='mse')
      return model
```

Snippet 2. Creating a Sequential Keras model.

Converting all functionalities of the Keras model to the HLS model: The Keras model is translated to HLS one with the help of the HLS4ML configuration that is developed for conversion process of Keras – HLS (Snippet 3):

```
def convert_model():
      model = make_dense_model()
      config = hls4ml.utils.config_from_keras_model(model)
      hls_model = hls4ml.converters.convert_from_keras_model(model,
      hls_config=config)
      return hls_model
```

Snippet 3. Conversion of the Keras model to the HLS model.

Testing the correctness of the conversion process: Through the `assert` statement of Pytest, attributes of the Keras model like input_shape, output_shape, number of layers, class name and etc. are compared with the HLS model's (Snippet 4):

```
def test_conversion():
      model = make_dense_model()
      hls_model = convert_dense_model()
      assert len(model.layers) + 1 == len(hls_model.get_layers())
      assert list(hls_model.get_layers())[1].attributes["class_name"]
      == model.layers[0]._name
      assert list(hls_model.get_layers())[0].attributes['input_shape']
      == list(model.layers[0].input_shape[1:])
      assert list(hls_model.get_layers())[1].attributes['n_out'] ==
      model.layers[0].output_shape[1:][0]
```

Snippet 4. Testing of the converted attributes.

Predicting and comparing results of both models: Both models are trained and predict results by using input data that is generated from Numpy randomly. Then both prediction results are compared with each other and this comparison can be done with Numpy's function, like `assert_allclose` (Snippet 5):

```
from numpy.testing import assert_allclose
def test_dense_prediction():
      model = make_dense_model()
      hls_model = convert_dense_model()
      X_input = np.random.rand(1,)
      keras_prediction = model.predict(X_input)
      hls_model.compile()
      hls_prediction = hls_model.predict(X_input)
      assert_allclose(keras_prediction, hls_prediction, rtol=1e-4)
```

Snippet 5. Comparing the prediction results of the both models.

Unlike the layers of the Sequential model, the layers of the Functional API have not been created and converted in the set of quaternary functions, but the testing process remained the same (Snippet 6):

```
merge_layers = [Add, Subtract, Multiply, Average, Maximum, Minimum,
Concatenate]
@pytest.mark.parametrize('merges', merge_layers)
def test_merge(merges):
      input1 = tf.keras.layers.Input(shape=(16,))
      x1 = tf.keras.layers.Dense(8, activation='relu')(input1)
      input2 = tf.keras.layers.Input(shape=(32,))
      x2 = tf.keras.layers.Dense(8, activation='relu')(input2)
      added = merges()([x1, x2])
      out = tf.keras.layers.Dense(4)(added)
      model = tf.keras.models.Model(inputs=[input1, input2], outputs=out)
      hls_model = hls4ml.converters.convert_from_keras_model(model)
```

Snippet 6. Creating and converting the Keras models with the Functional API layers.

The accuracy error is accepted as error = 15.

While the prediction results of the Keras and HLS models do not distinguish much from each other, some bugs appeared during the testing of the conversion process. For example, when we convert the Keras model with Batch Normalization layer to the HLS model, it is found out that the Batch Normalization layer is absent in the HLS model and disappears within the conversion process. As known, Batch normalization layer maintains the mean output close to 0 and the output standard deviation close to 1. It is a significant layer for machine learning applications and absence of it can be catastrophic. There is another error in the conversion of the Keras model when padding is “same” and the data format is “channel_first” form in MaxPooling2D and AveragePooling2D layers. Through the conversion of two-dimensional pooling layers (MaxPooling2D and AveragePooling2D) the column of the input in the layers of Keras model is assigned to the input height of the HLS model instead of the width of it. In consequence of this bug, input width and input height of the HLS models assigned to wrong data.

Such bugs are small and, although unnoticed, they can cause major problems in the overall project. However, it can be easily detected and eliminated with the help of tests.

Furthermore, the accuracy of the models does not change significantly within the conversion process and it is considered as acceptable.

After all, the models are synthesized with the help of Vivado HLS and the prediction results are compared with each other.

6. Jenkins pipeline for testing of HLS4ML

Continuous integration is a requirement of software projects where active and continuous development is provided [11]. Each development process needs to go through a testing process before it reaches the user so that the product is stabilized. Jenkins pipeline is set up for automating the project whenever new changes are done in Keras testing infrastructure.

There are also models that were implemented with QKeras and converted to HLS models. QKeras is a quantization extension to Keras which uses the implementation of some of Keras layers, for example QDense, QConv1D and QActivation accordingly Dense, Conv1D and Activation in Keras. This version gives an opportunity to create neural networks with lower precision bits and lower inference time with minimal accuracy loss [12]. Jenkins pipeline has been set up for QKeras conversion process. This process consists of two stages: the conversion from Python API test and Vivado synthesis test [12]. It is useful to use two docker images to prevent collision of two stages. Nevertheless, one docker image can be used for the entire pipeline too.

With reference to Jenkins pipeline for QKeras framework conversion, the same thing can be done for Keras to HLS conversion process and it becomes as follows (Snippet 7) [12]:

```
pipeline {
    agent {
        docker {
            image 'hls4ml-with-vivado:latest'
        }
    }
```

```
          options {
        timeout (time: 6, unit: 'HOURS')
    }
    stages {
        parallel {
            stage('keras-api-test'){
        steps {
            sh './run-test test_keras.py'
        }
   }
  stage('keras-vivado-test') {
        steps {
            sh './run-test test_keras_vivado.py'
        }
  }
          }
    }
}
```

Snippet 7. Jenkins pipeline for testing of Keras-HLS conversion.

7. Conclusion

The project works effectively on FPGA chips by minimizing the loss of accuracy when converting machine learning algorithms to the HLS4ML framework targeting extremely low-latency requirements of the LHC detector. When testing the conversion of Keras models, it revealed that there were a few bugs in the conversion of BatchNormalization, MaxPooling2D and AveragePooling2D. These are very important layers and any small mistake in converting them can result in serious issues. Consubstantiation of the testing infrastructure to this project makes it more strong and mature. The test will endorse the implementation of new features in the future and allow it to run smoothly while the network architecture becomes increasingly complex.

As Jenkins pipeline has already been set up for the whole testing infrastructure, it will encompass all stages of new unit tests inclusion, maintain consistency and functionality of the project. For future developments, a three-dimensional Convolution layer and Pooling layers can be added. Moreover, other frameworks like Onnx and Pytorch, other Neural Network architectures like Recurrent Neural Networks and Graph Networks can be developed.

Acknowledgements

I thank my supervisor Vladimir Loncar who took the time for the useful discussions and supported me to complete this project successfully.

References

[1] CMS Collaboration, “The CMS Experiment at the CERN LHC,” JINST 3, S08003 (2008). [doi: 10.1088/1748-0221/3/08/S08004].

[2] L.Evans et al., “LHC Machine,” JINST 3, (2008). [doi:10.1099/1748/3/08/S08001].

[3] CMS Collaboration, “The CMS Trigger System,” JINST 12, P01020 (2017) [arXiv:1609.02366v2].

[4] CERN articles from home page, “From capturing collisions to avoiding them”, (2019).

[5] CERN Courier, “FPGAs that speak your language”, (September 2019 p21).

[6] CMS Collaboration, “Level-1 Track Finding with an all-FPGA System at CMS for the HL-LHC”, Connecting the Dots and Workshop on Intelligent Trackers (CTD/WIT 2019) [arXiv:1910.12668].

[7] Claudionor N. Coelho Jr., Aki Kuusela, Shan Li, Hao Zhuang, Thea Aarrestad, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, Adrian Alan Pol, Sioni Summers, “Automatic deep heterogeneous quantization of Deep Neural Networks for ultra low-area, low-latency inference on the edge at particle colliders”, (2020, June 15) [arXiv:2006.10159].

[8] Wikipedia, “High-level synthesis”, (n.d.).

[9] HLS4ML Documentation, (n.d), https://fastmachinelearning.org/hls4ml/.

[10] Pytest framework documentation (n.d.), https://docs.pytest.org/en/stable/.

[11] Jenkins Documentation (n.d.), https://www.jenkins.io/.

[12] S. Nuntaviriyakul, “HLS4ML - Testing Infrastructure”, CERN Summer Student Program, (2020, October), https://cutt.ly/gchp4fT.