Istituto Nazionale di Fisica Nucleare

Accelerating Machine Learning inference using FPGAs: a crush course

Dr. Marco Lorusso [1,2]

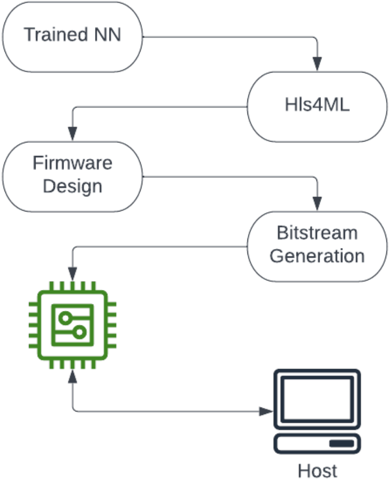
[1]University of Bologna - Department of Physics and Astronomy

[2]National Institute for Nuclear Physics - Bologna Division

4th November 2022

Implementing a Neural Network on an FPGA

- **NN Translation into HLS** (C++) using *hls4ml* (see next slide);
- **Firmware design** (I/O interfaces);
- **Synthesis and implementation** of the design;
- Production of the **bitstream and programming** of the FPGA;
- **Running** of the inference using an application on the **host** machine.



From Python to HLS Code

```python
import tensorflow as tf
from qkeras.qlayers import QDense, QActivation

netinputs = tf.keras.layers.Input(shape=(4,),dtype=X_train.dtype,name="input_1")
x = QActivation(activation=quantized_relu(16,6,relu_upper_bound=6.0),
                name='qrelu1')(inputs)
x = QDense(16, kernel_quantizer=quantized_bits(16,5,alpha=1),
            bias_quantizer=quantized_bits(16,5,alpha=1),
            kernel_initializer='random_normal',name='qdense_1')(x)
x = QActivation(activation=quantized_relu(16,6), name='qrelu2')(x)
#...# List of layers and activation functions
output = tf.keras.layers.Activation('softmax', name='soft1')(x)
model = tf.keras.Model(inputs=netinputs,outputs=netoutput,name="model")
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
history = model.fit(X_train, Y_train, epochs=num_epochs, validation_data=(X_test, Y_test))
```

↓

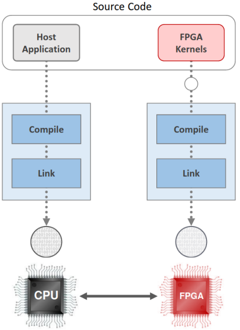
```python
import hls4ml

config = hls4ml.utils.config_from_keras_model(model, granularity='model')
hls_model = hls4ml.converters.convert_from_keras_model(model,
            hls_config=config, part='<id of FPGA model>')
hls_model.compile()
hls_model.build(csim=False,synth=False)
```

INFN
Istituto Nazionale di Fisica Nucleare

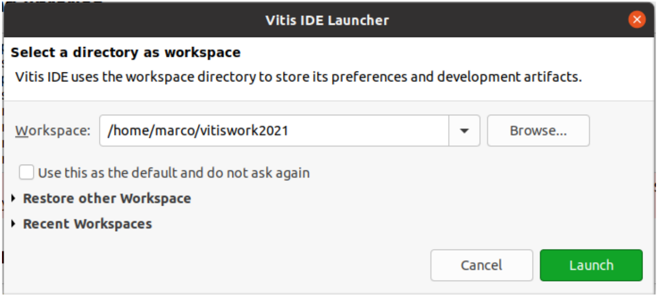
Producing the Bitstream with Vitis

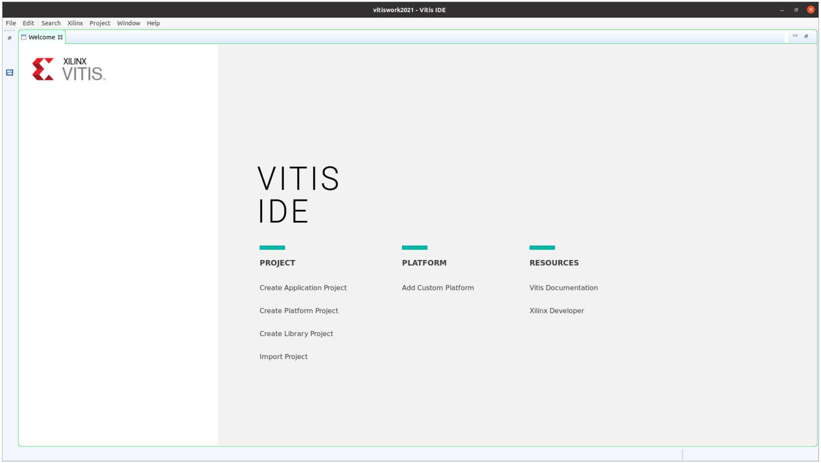
The *build* function creates the HLS code to import in the Vitis Software Platform developed by Xilinx.

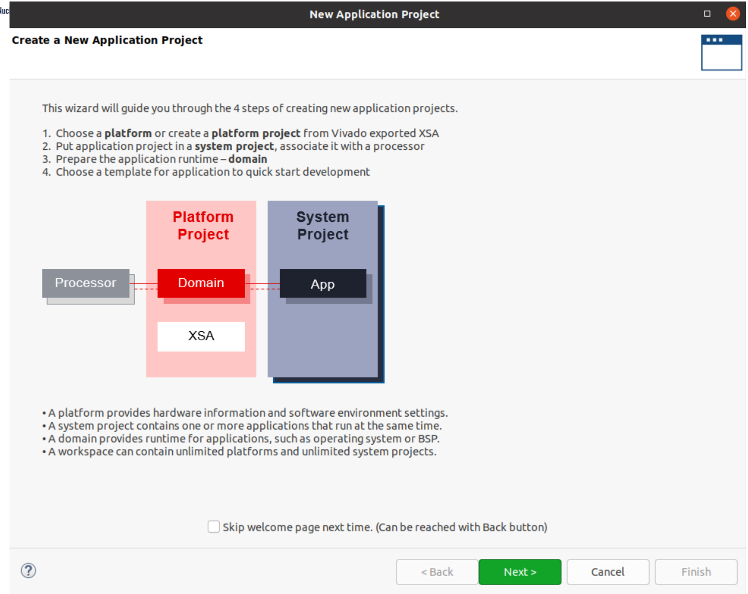
- An **application project** with the target platform is created;
- The HLS code from *hls4ml* is imported as **source** for the *kernel* of the application;
- A *Hardware function* is associated to the **main C++ function** in the code;
- The **host application** is usually written in OpenCL;
- The whole application is build for **hardware deployment** → **Bitstream**.

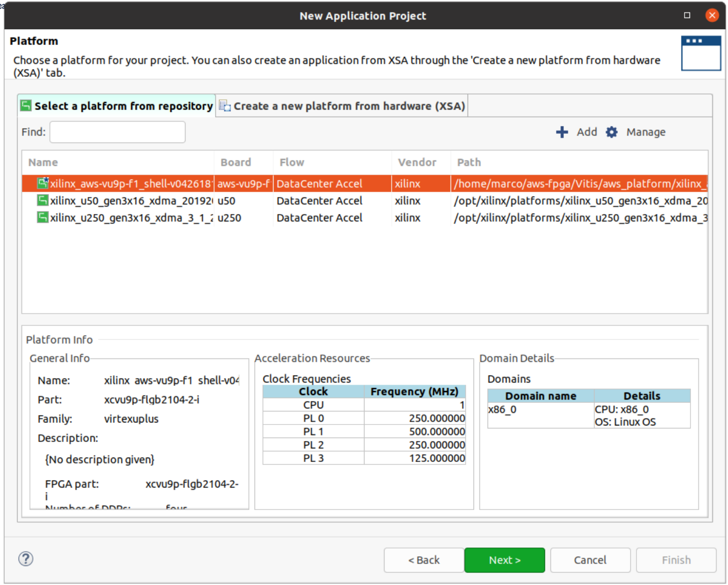


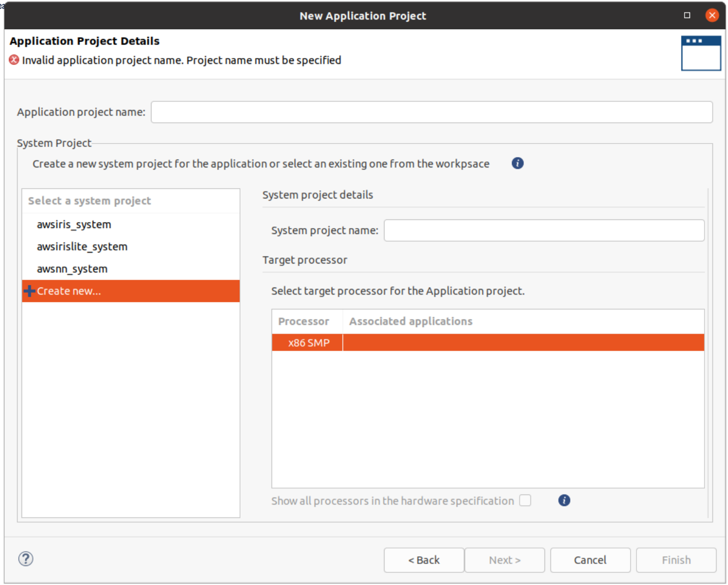
XILINX VITIS

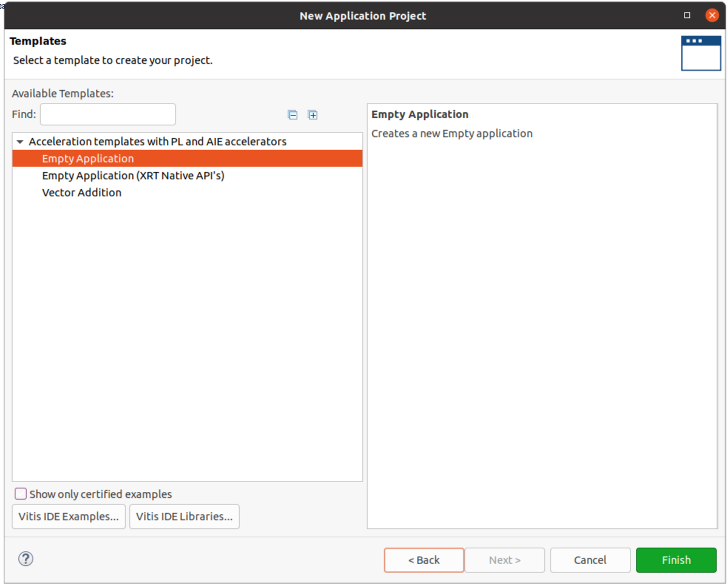

Vitis IDE Launcher
Select a directory as workspace
Vitis IDE uses the workspace directory to store its preferences and development artifacts.
Workspace:
/home/marco/vitiswork2021
Browse...
Use this as the default and do not ask again
Restore other Workspace
Recent Workspaces
Cancel
Launch

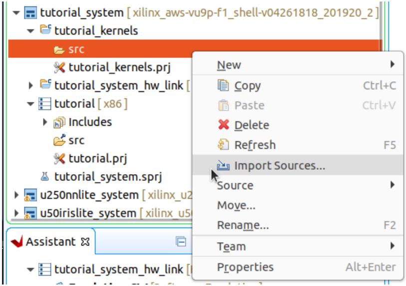

vitiswork2021 - Vitis IDE
File Edit Search Xilinx Project Window Help
Welcome
XILINX VITIS
VITIS IDE
PROJECT
Create Application Project
Create Platform Project
Create Library Project
Import Project
PLATFORM
Add Custom Platform
RESOURCES
Vitis Documentation
Xilinx Developer

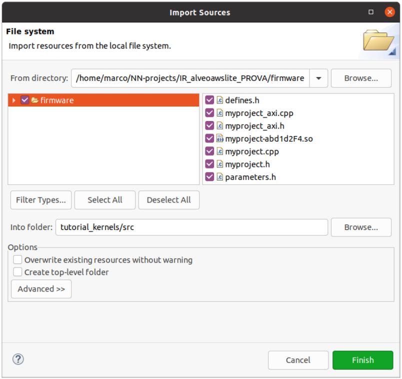

New Application Project
Create a New Application Project
This wizard will guide you through the 4 steps of creating new application projects.
1. Choose a **platform** or create a **platform project** from Vivado exported XSA
2. Put application project in a **system project**, associate it with a processor
3. Prepare the application runtime – **domain**
4. Choose a template for application to quick start development
Platform Project
System Project
Processor
Domain
App
XSA
• A platform provides hardware information and software environment settings.
• A system project contains one or more applications that run at the same time.
• A domain provides runtime for applications, such as operating system or BSP.
• A workspace can contain unlimited platforms and unlimited system projects.
Skip welcome page next time. (Can be reached with Back button)
< Back
Next >
Cancel
Finish


New Application Project
Platform
Choose a platform for your project. You can also create an application from XSA through the 'Create a new platform from hardware (XSA)' tab.
Select a platform from repository
Create a new platform from hardware (XSA)
Find:
Add
Manage
Name
Board
Flow
Vendor
Path
xilinx_aws-vu9p-f1_shell-v042618
aws-vu9p-f
DataCenter Accel
xilinx
/home/marco/aws-fpga/Vitis/aws_platform/xilinx_
xilinx_u50_gen3x16_xdma_20192
u50
DataCenter Accel
xilinx
/opt/xilinx/platforms/xilinx_u50_gen3x16_xdma_20
xilinx_u250_gen3x16_xdma_3_1_
u250
DataCenter Accel
xilinx
/opt/xilinx/platforms/xilinx_u250_gen3x16_xdma_3
Platform Info
General Info
Name: xilinx_aws-vu9p-f1_shell-v04
Part: xcvu9p-flgb2104-2-i
Family: virtexuplus
Description:
{No description given}
FPGA part: xcvu9p-flgb2104-2-i
Acceleration Resources
Clock Frequencies
Clock
Frequency (MHz)
CPU 1
PL 0 250.000000
PL 1 500.000000
PL 2 250.000000
PL 3 125.000000
Domain Details
Domains
Domain name
Details
x86_0
CPU: x86_0
OS: Linux OS
< Back
Next >
Cancel
Finish

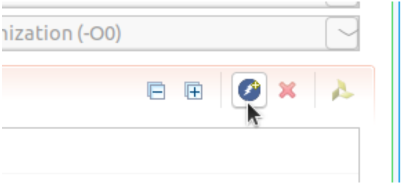

New Application Project
Application Project Details
Invalid application project name. Project name must be specified
Application project name:
System Project
Create a new system project for the application or select an existing one from the workspace
Select a system project
awsiris_system
awsirislite_system
awsnn_system
Create new...
System project details
System project name:
Target processor
Select target processor for the Application project.
Processor
Associated applications
x86 SMP
Show all processors in the hardware specification
< Back
Next >
Cancel
Finish


New Application Project
Templates
Select a template to create your project.
Available Templates:
Find:
Acceleration templates with PL and AIE accelerators
Empty Application
Empty Application (XRT Native API's)
Vector Addition
Show only certified examples
Vitis IDE Examples...
Vitis IDE Libraries...
Empty Application
Creates a new Empty application
< Back
Next >
Cancel
Finish

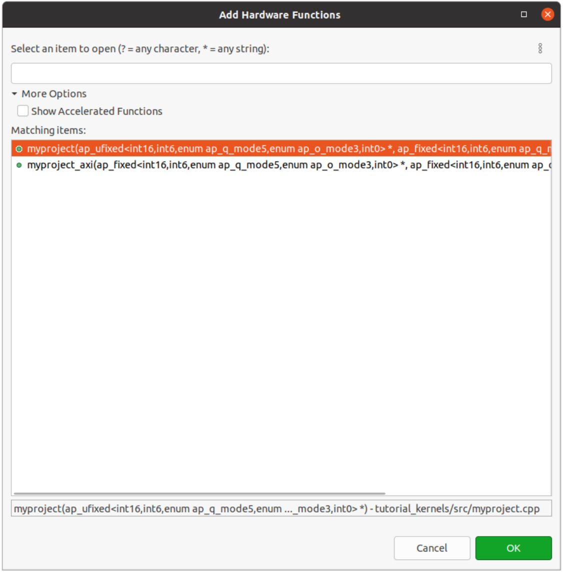

tutorial_system [xilinx_aws-vu9p-f1_shell-v04261818_201920_2]
tutorial_kernels
src
tutorial_kernels.prj
tutorial_system_hw_link
tutorial [x86]
Includes
src
tutorial.prj
tutorial_system.sprj
u250nnlite_system
u50irislite_system
Assistant
tutorial_system_hw_link
New
Copy Ctrl+C
Paste Ctrl+V
Delete
Refresh F5
Import Sources...
Source
Move...
Rename... F2
Team
Properties Alt+Enter

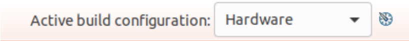

Import Sources
File system
Import resources from the local file system.
From directory: /home/marco/NN-projects/IR_alveoawslite_PROVA/firmware
Browse...
firmware
defines.h
myproject_axi.cpp
myproject_axi.h
myproject-abd1d2F4.so
myproject.cpp
myproject.h
parameters.h
Filter Types...
Select All
Deselect All
Into folder: tutorial_kernels/src
Browse...
Options
Overwrite existing resources without warning
Create top-level folder
Advanced >>
Cancel
Finish

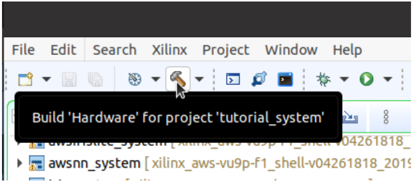
```
void myproject(
    input_t input_1[N_INPUT_1_1],
    result_t layer8_out[N_LAYER_6],
) {

    //hls-fpga-machine-learning insert IO
    //#pragma HLS ARRAY_RESHAPE variable=input_1 complete dim=0
    //#pragma HLS ARRAY_PARTITION variable=layer8_out complete dim=0
    //#pragma HLS INTERFACE ap_vld port=input_1,layer8_out
    #pragma HLS PIPELINE

    unsigned short const_size_in_1 = N_INPUT_1_1;
    unsigned short const_size_out_1 = N_LAYER_6;
```



Istituto Nazionale di Fisica Nucleare




Active build configuration:
Hardware


File Edit Search Xilinx Project Window Help
Build 'Hardware' for project 'tutorial_system'
awsnn_system

The testing ground: AWS F1 Instances

Cloud computing is used to test the capabilities of these tools in preparation for deployment of FPGA accelerator cards in a local server.

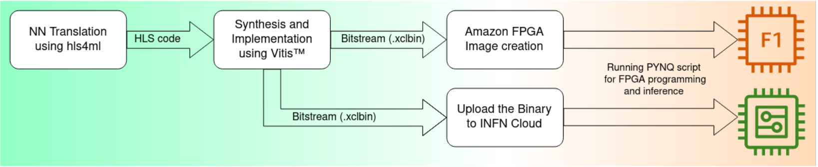
- Part of the **AWS Cloud Computing** catalogue;
- EC2 F1 instances use **FPGAs** to enable **delivery** of **custom hardware accelerations**;
- Packaged with **tools** to **develop**, simulate, debug, and **compile** a design.

Amazon EC2

Deploying on F1

- Follow the *Application Acceleration development flow*, offered by Vitis™, targeting data center acceleration cards;
- **Upload** the **bitstream** to a S3 bucket and request the **creation** of an *Amazon FPGA Image* (AFI) accessible from all F1 instances;
- Write a **Pyhton script** using PYNQ APIs.

A "more traditional" approach is to use **OpenCL** to write the host application: both ways follow the **same** list of **basic instructions**.



Istituto Nazionale di Fisica Nucleare

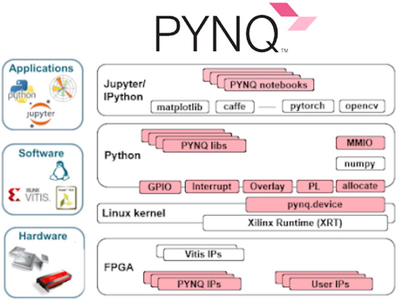
Amazon FPGA Image

- The *aws-fpga* repository contains all the tools needed for deploying (and developing) on a F1 instance;
- The awsxclbin (AFI) can be created by running the `create_vitis_afi.sh` script which is included in the `Vitis/tools/` directory;
- Before running the command, make sure that `aws-fpga/vitis_setup.sh` has been sourced;
- Remember to configure the AWS CLI and set up the bucket region, e.g. `aws configure set region us-east-1`;
- Create an AFI by running:

```
aws-fpga/Vitis/tools/create_vitis_afi.sh -xclbin=<filename>.xclbin
↪   -s3_bucket=<bucket-name> -s3_dcp_key=<dcp-folder-name>
↪   -s3_logs_key=<logs-folder-name>
```

The PYNQ project

- *PYNQ* is an **open-source** project from Xilinx®;
- It provides a **Jupyter-based framework** with Python APIs for using Xilinx platforms;
- The **Python language** opens up the **benefits of programmable logic** (PL) to people **without** in-depth knowledge of **low-level programming languages**.



https://pynq.readthedocs.io

An introduction to PYNQ

- The **overlay** class is the **core** of the library;
- An overlay object is built providing the **FPGA design** to run on the PL;
- FPGA is **programmed** and relevant **interface** is available through **PYNQ API function** calls;
- It is possible to **accelerate** a software **application**, or to customize the hardware platform for a particular application.

```python
from pynq import Overlay

overlay = Overlay("designbitstream.xclbin") # or .awsxclbin
result = overlay.<function described in FPGA design>
```

OpenCL vs PYNQ

The first thing to do in both cases, is to **program the device and initialize** the software context.

```cpp
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));

```

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

In OpenCL host and FPGA **buffers** need to be handled separately and linked after creation; with PYNQ, the user is only presented with a single interface for both:

```cpp
std::vector<int, aligned_allocator<int>> source_in1(DATA_SIZE);
OCL_CHECK(err, l::Buffer buffer_in1(context,
    CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, vector_size_bytes,
    source_in1.data(), &err))
```

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

OpenCL vs PYNQ (cont'd)

To **initiate data transfers** the direction as a function parameter must be specified in OpenCL, while in PYNQ the same is done with a specific function:

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

```
inp.sync_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()`; in PYNQ, the `.call()` function does everything in a single line.

```
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());
```

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

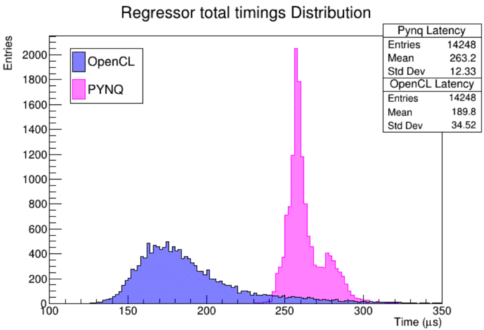
Finally, the **output is retrieved** in both cases similarly to the input transfer:

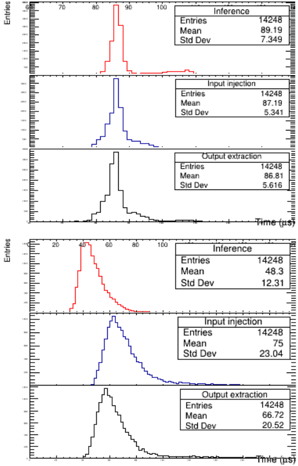
```
OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_output},
CL_MIGRATE_MEM_OBJECT_HOST));
```

```
out.synq_from_device()
```

Timing Comparison

A **difference** in **computation times** can be seen between the same algorithm deployed with PYNQ and OpenCL:





Thank you!

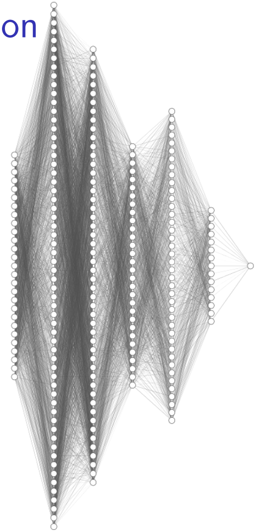
Backup

Neural Network for regression

A **Fully Connected MLP** was built using QKeras with:

- **Input layer**: 27 features;
- **6 hidden layers**: 35, 20, 25, 40, 20, 15 nodes;
- **Output layer**: returns the p_T value.
- **Activation function**: Rectified Linear Unit;
- **Weight pruned**.

The model was **tested using a consumer CPU** before the hardware implementation.



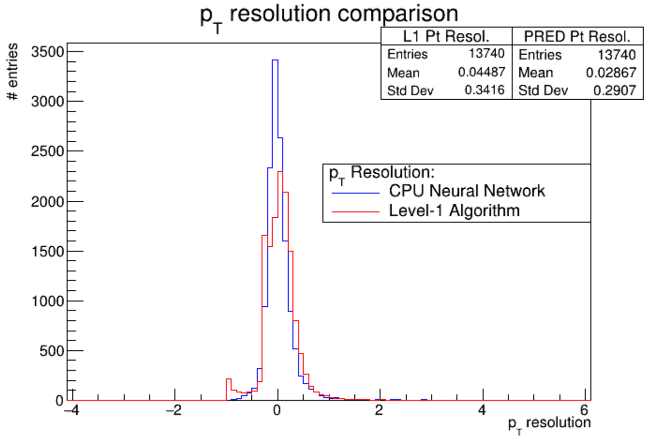
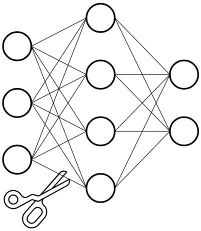


Figure: Transverse momentum resolution histograms computed for the machine learning model (blue) and Level-1 trigger (red) based momentum assignment.

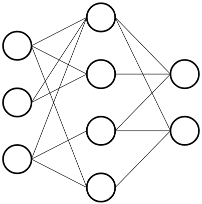
Optimization techniques

To produce an **optimized NN** for **implementation** on an FPGA:

- **Quantization**: the parameters were converted **from double precision floating-points to fixed points** to exploit the efficiency of DSPs;
- **Pruning**: **connections** between nodes with low influence were **cut** to **minimize** the number of **paramaters** and operations during inference and **reduce the resources** needed for implementation.



Before pruning

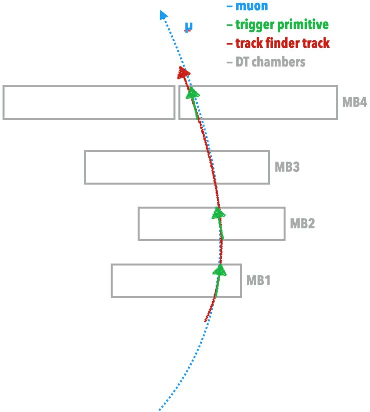


After pruning

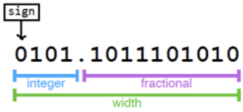
Dataset to train and test the NN

The entire **dataset** contains about **300000 simulated muons** with a range in p_T **from 3 to 200 GeV/c**. A set of **information** is included in order to **predict** the muon p_T:

- **Trigger segments' position** (wheel, sector, ϕ) for each station crossed by the particle;
- Their **direction** in CMS global coordinates (ϕ_b).
- Trigger primitives' **quality** (i.e. number of hits used to build a segment).



Quantization



ap_fixed<14,4>

In order to produce an **optimized NN** for **implementation** on an FPGA, the models were *quantized*:

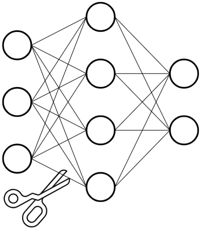
- *Quantization* is the conversion **from high-precision floating-points to normalized low-precision integers** (*fixed-point*) parameters;
- *QKeras* is a Python package developed as a collaboration between Google and HEP researchers to **build NN with quantized parameters**;
- It has an easy-to-use API: there are **drop-in replacements** for the most common layers used with Keras (e.g. Dense → QDense).

```
QDense(64, kernel_quantizer = quantized_bits(6,0),
           bias_quantizer = quantized_bits(6,0)(x))
QActivation('quantized_relu(6,0)')(x)
```

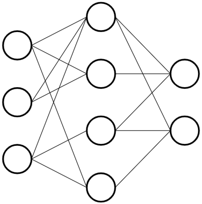
Slimming techniques - Weight Pruning

When building a NN model,
the final hardware platform where the inference computation will be run, has to be considered.

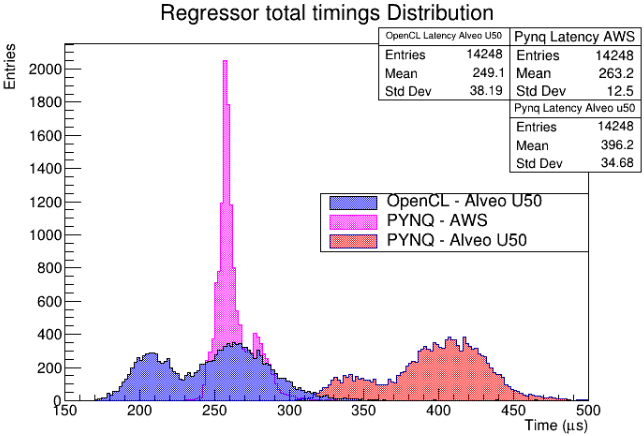
- *Weight Pruning* is the elimination of unnecessary values in the weight tensor;
- Connections between nodes with low influence are "cut" during the synthesis of the HLS design;
- This is aimed at minimizing the number of parameters and operations involved in the inference computation.



Before pruning



After pruning


Regressor total timings Distribution
OpenCL Latency Alveo U50
Entries 14248
Mean 249.1
Std Dev 38.19
Pynq Latency AWS
Entries 14248
Mean 263.2
Std Dev 12.5
Pynq Latency Alveo u50
Entries 14248
Mean 396.2
Std Dev 34.68
OpenCL - Alveo U50
PYNQ - AWS
PYNQ - Alveo U50
Entries
Time (μs)
0
200
400
600
800
1000
1200
1400
1600
1800
2000
150
200
250
300
350
400
450
500