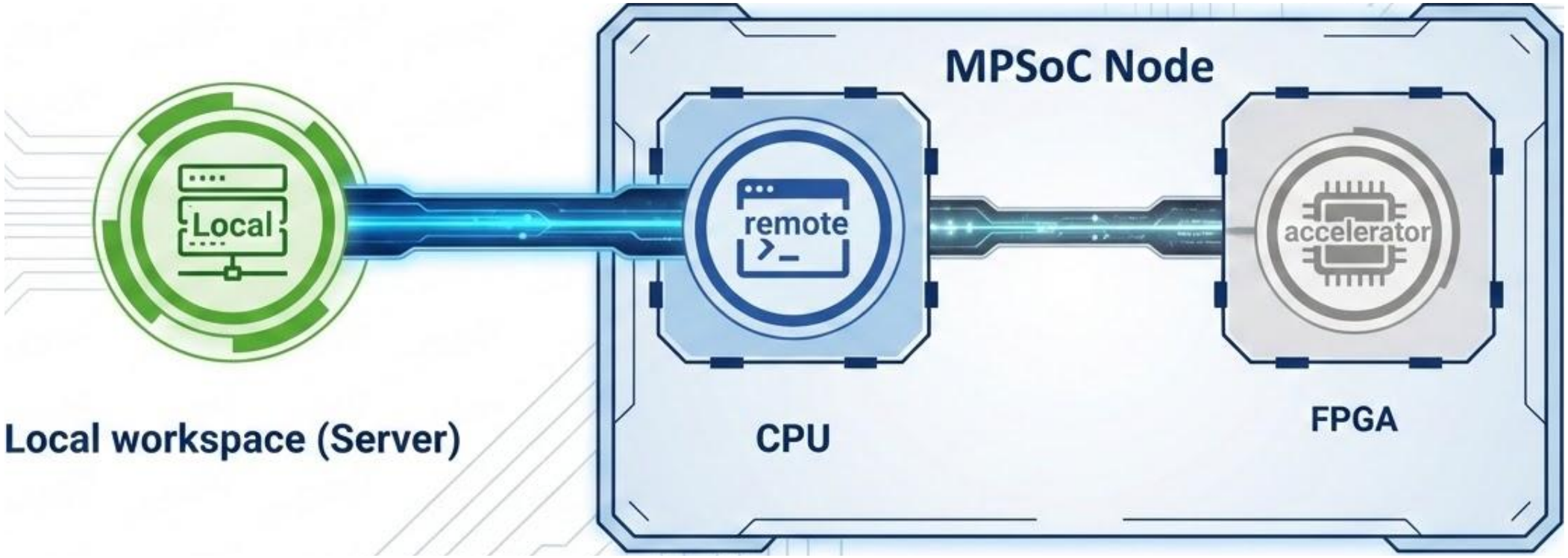
Deploying Machine Learning Models on Heterogeneous MPSoC-FPGA Devices

School on Applied AI for Sustainable Development
smr 4210, Trieste 25 March 2026

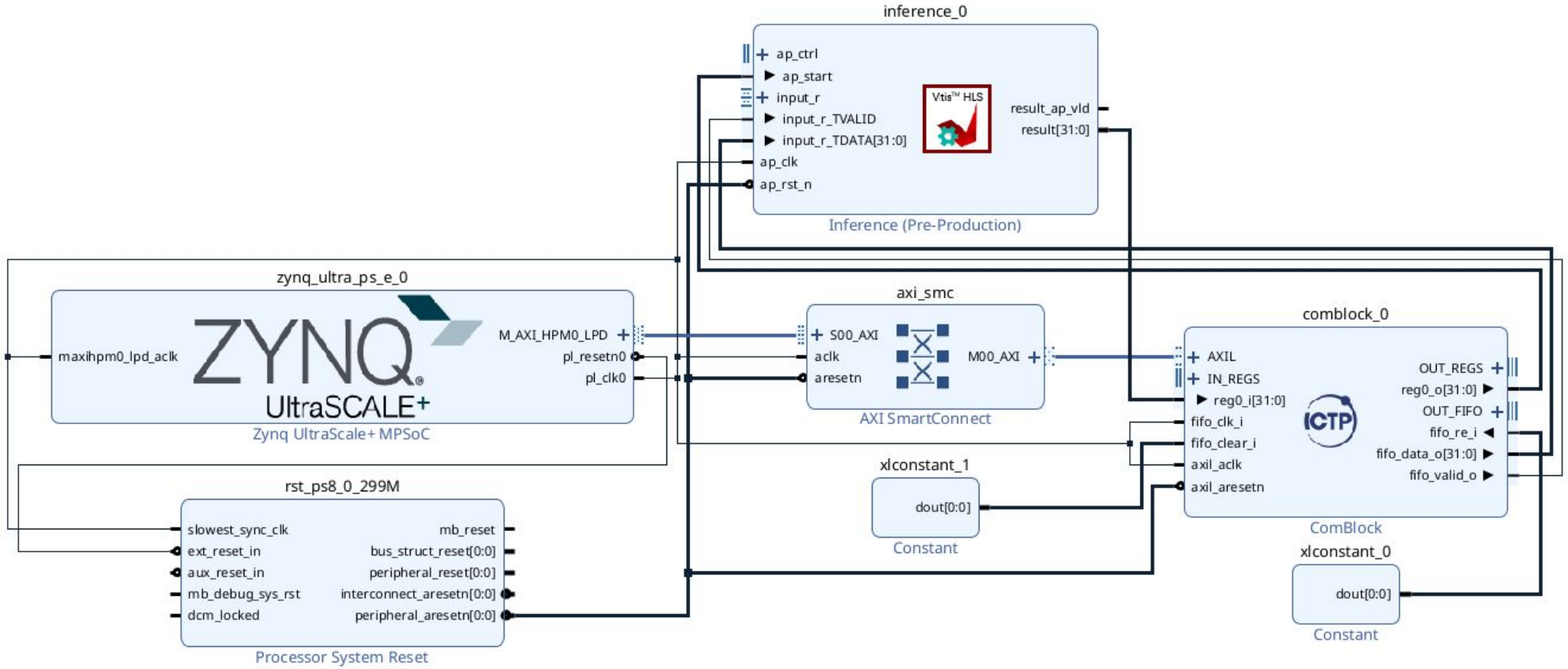
Maynor Ballina

The Abdus Salam International Centre for Theoretical Physics

Your Heterogeneous Ecosystem



Development Workflow - Vivado



Maynor Ballina

Development Workflow - Vivado

Default Part

Choose a default AMD part or board for your project.

Parts | **Boards**

To fetch the latest available boards from git repository, click on 'Refresh' button. Dismiss

Reset All Filters

Vendor: All Name: All Board Rev: Latest

Search:

Display Name	Preview	Status	Vendor	File Version	Part	I/O Pin Count	Board
HyperFPGA 3be11		Installed	ictp.it	1.0	xczu3eg-sfvc784-1-e	784	1.0
HyperFPGA 4ge21		Installed	ictp.it	1.0	xczu4eg-sfvc784-2-e	784	1.0

Export Hardware Platform

Files

Enter the name of your hardware platform file, and the directory where the XSA file will be stored.

-4ge21

XSA file name: mnist-4ge21

Export Hardware Platform

Files

Enter the name of your hardware platform file, and the directory where the XSA file will be stored.

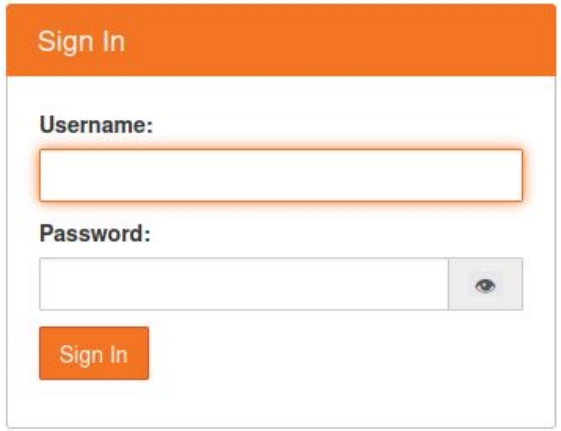
-3be11

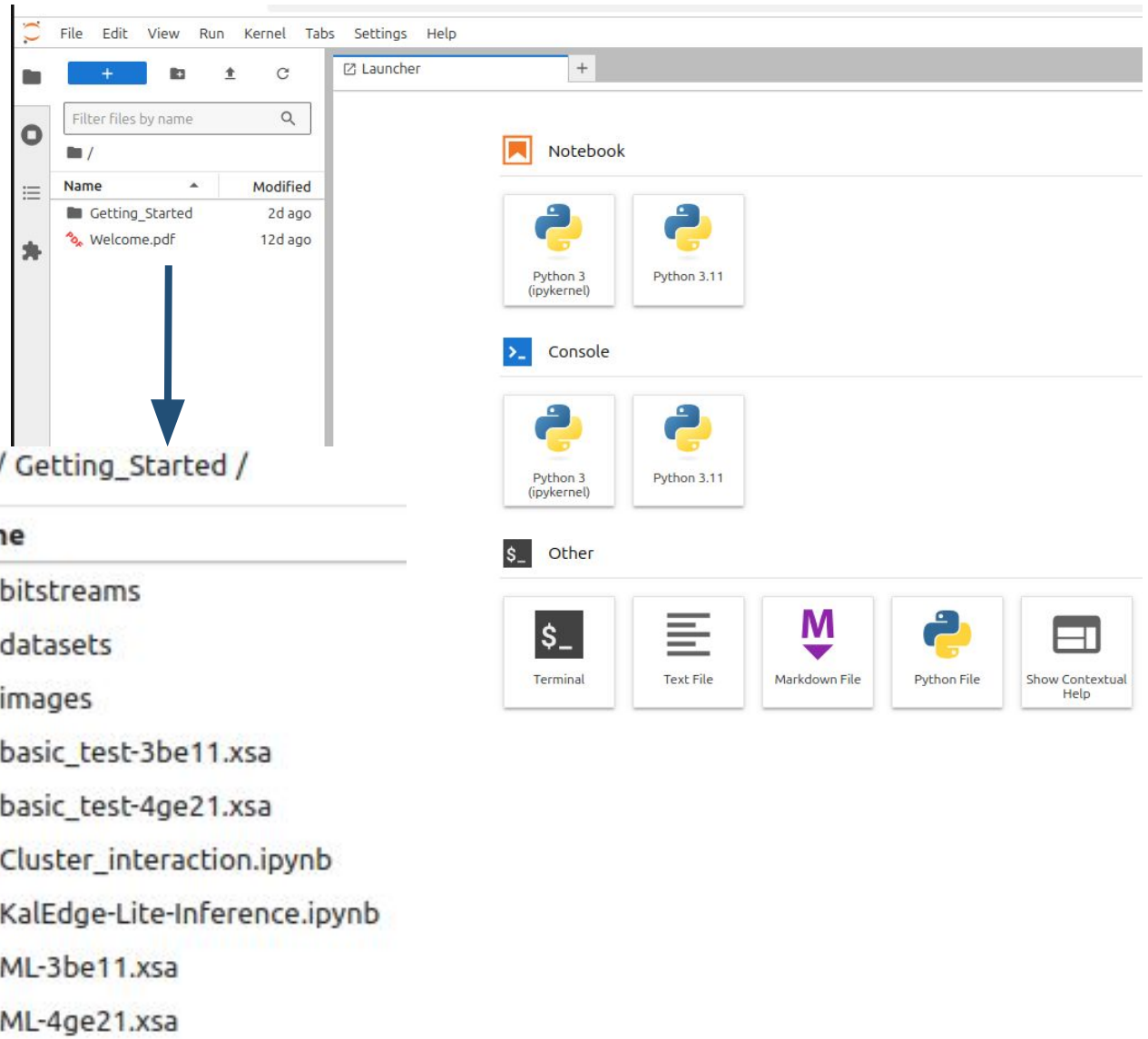
XSA file name: mnist-3be11

Maynor Ballina

Remote Access to the Cluster

https://hyperfpga.sti.ictp.it





Development Workflow - Python

Central server (Local): import hyperfpga_cluster as hfc

```
import hyperfpga_cluster as hfc
```

Node (Remoto): from comblock import Comblock - cb = Comblock()

```
dview.execute("""
from comblock import Comblock, NumericFormat, FixedPointPresets
cb = Comblock()""")
```

IPython Parallel is an extension of IPython for executing code in parallel across multiple cores or nodes.

Its goal is to improve the performance of computational tasks through parallelization. Its main features are concurrent execution, support for task distribution in clusters, and centralized control and monitoring.

Exercise 1: Cluster Interaction

Cluster Configuration Library

The user environment is set up to work with Python 3.11, so when running notebooks, make sure to select a Python 3.11 kernel.

This environment includes a library called **hyperfpga_cluster**, which provides the tools needed to configure your node and communicate with it. The library contains a class named **HyperFPGACluster**, which offers methods to configure and interact with the HyperFPGA cluster.

In addition, the library provides the functions `get_nodes` and `node_view`, which allow you to visualize the assigned node and its position within the cluster.

Below is an example of how to import this library:

```
import hyperfpga_cluster as hfc
```

where:

- import: A keyword used to include a module in your script.
- hyperfpga_cluster: The name of the module being imported.
- as: A keyword that assigns an alias to the module.
- hfc: The alias used to reference the module in your code.

Maynor Ballina

Exercise 1: Cluster Interaction

Create Your Cluster

To create an instance of the **HyperFPGACluster** class, you need to provide several parameters as shown below:

```
object_name = hfc.HyperFPGACluster(<nodes>, <firmware>, <update>, <n_engines>, <engines_per_node>)
```

Parameters:

- nodes: A variable containing the dictionary of assigned nodes.
- firmware: This can be either a single value or a list. You only need to specify the project name. For example, if the file is `test-4ge21.xsa`, the firmware value should be *'test'*.
- update: A flag that determines whether the configuration should be refreshed.
- n_engines: Total number of processing engines to be used by default is 1.
- engines_per_node: Number of engines allocated to each node by default is 1.

Initialization Process

When the class is initialized, the following steps are performed:

Property validation: The class defines properties for nodes, profile, engines per node, total number of engines (n_engines), and firmware. These properties include validation to ensure that the data types and values are correct.

Exercise 1: Cluster Interaction

Configure Your Cluster

Finally, you need to configure the cluster as described earlier. This process uses the required configuration files to set up the FPGA on each node, leveraging `asyncssh` for asynchronous communication.

Because these operations are asynchronous, it is important to use **await** when calling them. This ensures that each task is completed before moving on, as **await** pauses execution until the operation finishes.

If you include the parameter `view=True`, the configured node will be highlighted in blue within the cluster visualization.

> Important: This process may take a few minutes to complete. Please be patient.

ipyparallel

ipyparallel is a Python library that enables parallel computing, making it possible to use multiple CPU cores or even multiple machines simultaneously.

An **ipyparallel** cluster is composed of:

- A client, which manages and distributes tasks
- Multiple workers (also called engines), which execute those tasks

To run tasks in parallel, you first need to establish a connection between the client and the workers. Once connected, the client can send tasks to the workers and collect the results efficiently.

Maynor Ballina

Exercise 1: Cluster Interaction

start_and_connect_sync

The `start_and_connect_sync` function from *ipyparallel* is used to start a worker cluster and establish a synchronous connection to it.

- Starts the cluster: If the worker cluster is not already running, the function automatically starts it. This involves launching multiple worker processes (engines) that can execute tasks in parallel.
- Synchronous connection: Once the cluster is running, the function connects the client to it synchronously. This means that execution is blocked until the connection is fully established. In other words, when start_and_connect_sync is called, the program will wait at that line until the client is successfully connected to all workers.

Direct View (`DirectView`)

A Direct View in **ipyparallel** allows you to execute the same command on all workers simultaneously.

How does it work?

When you use `remote_client[:]` or `remote_client.direct_view()`, a `DirectView` object is created that represents all nodes in the cluster.
Any code executed through this object is sent to all workers at the same time, and each worker runs the same instruction.

Load-Balanced View (`LoadBalancedView`)

A Load-Balanced View distributes tasks automatically among available workers, ensuring an even workload.

How does it work?

When you use `rc.load_balanced_view()`, a `LoadBalancedView` object is created. Instead of sending the same task to all workers, tasks are distributed dynamically, so each worker processes a portion of the workload efficiently.

Maynor Ballina

Exercise 1: Cluster Interaction

Interacting with the Hardware

At this stage, you can start interacting with the hardware. Since Python runs on each node, any required Python libraries must be initialized on every node according to your needs. For this purpose, the sync_imports function should be used.

dview.execute()

dview is a DirectView object from ipyparallel that allows you to execute commands across multiple cores or nodes. The method `.execute("command")` runs a given Python command (as a string) on all connected workers.

Comblock library

Each node includes a Python library that enables interaction with the Comblock core. Before using it, the library must be initialized:

@dview.remote(block=True)

The decorator @dview.remote(block=True) specifies that the function defined below it will be executed on all nodes associated with dview. The block=True parameter ensures that execution waits until all nodes have completed the task before continuing.

For example, you can define a function that processes data (e.g., summing elements of a vector), and it will run on the CPU of each remote node.

Exercise 1: Cluster Interaction, solution

Exercises:

Based on the previous examples and the block design:

- Send a set of 20 random values to inputs A and B of the adders
- Retrieve the results and compare them
- Analyze the differences between the two adders based on numerical representation

```
import random
@dview.remote(block=True)
def exercise_register(A, B, format_type='unsigned'):
    if format_type == 'float':
        format_type = NumericFormat.FLOAT_32
    elif format_type == 'signed':
        format_type = NumericFormat.SIGNED_32
    else:
        format_type = NumericFormat.UNSIGNED_32

    cb.write_reg(1, A, format_type)
    cb.write_reg(2, B, format_type)
    return cb.read_reg(1, format_type), cb.read_reg(2, format_type)

for i in range(20):
    print(exercise_register(A=random.randint(-100, 100), B=random.randint(-100, 100), format_type='signed'))
```

Maynor Ballina

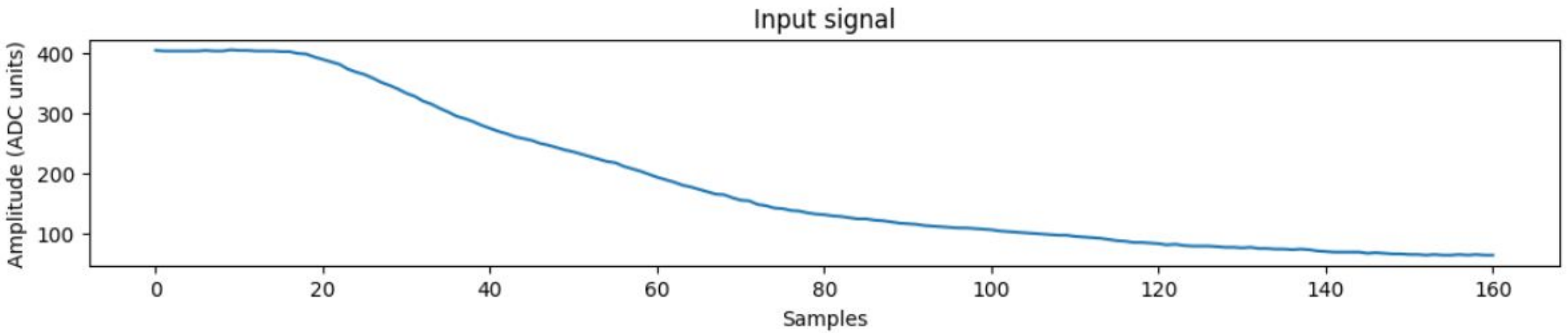
Exercise 2: ML y SoC-FPGA: Hardware verification

This exercise uses a machine learning model that has been optimized using various compression techniques to enable deployment on an FPGA.

In this activity, the student will:

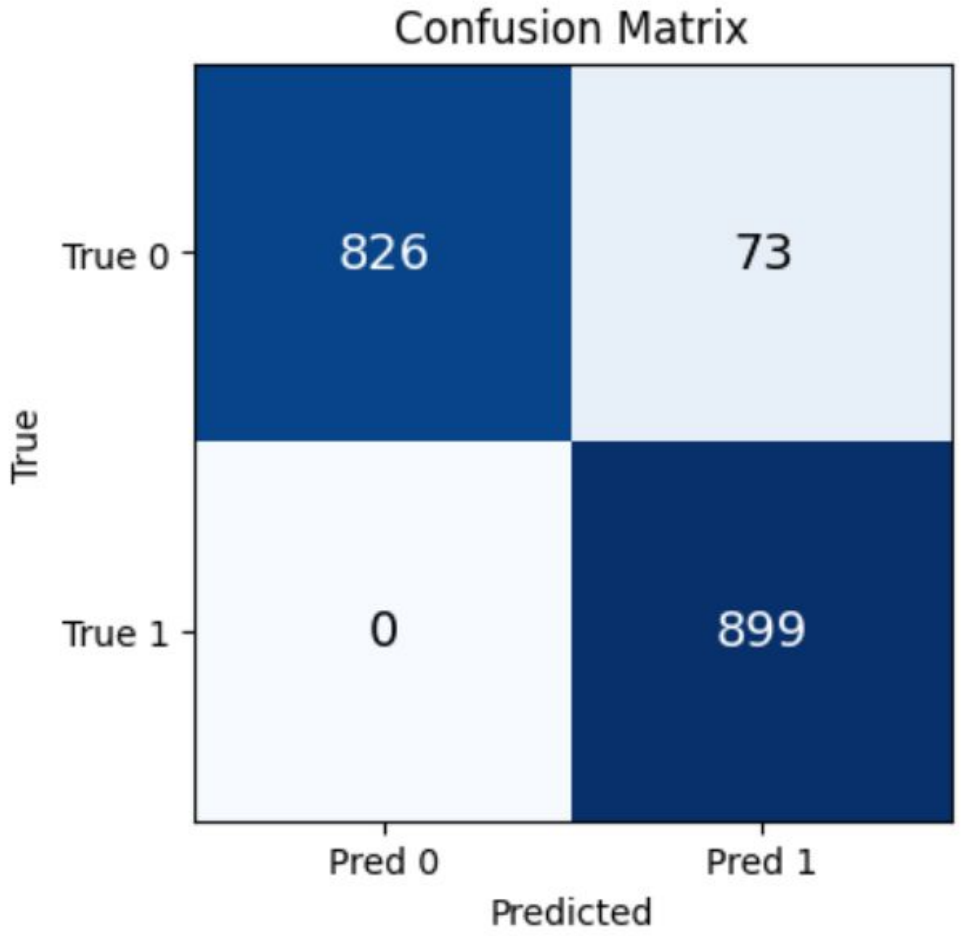
- Deploy the model onto the FPGA
- Manage data transmission between the host and the hardware
- Validate that the model implemented on the FPGA behaves the same as, or similarly to, the software model developed in the previous session

The goal is to ensure that the hardware implementation maintains the accuracy and functionality of the original machine learning model.



Exercise 2: Expected results

```
Accuracy : 0.9594
Precision: 0.9249
Recall   : 1.0000
F1-score : 0.9610
```



Thank you

Q&A

mballina@ictp.it