Development of accelerators for ML and I(nference)aaS systems on FPGA

Giulio Bianchini [1] Diego Ciangottini [2] Mirko Mariotti [1,2] Loriano Storchi [3,2]
Giacomo Surace [2] Daniele Spiga [2]

[1]Dipartimento di Fisica e Geologia, Universitá degli Studi di Perugia

[2]INFN sezione di Perugia

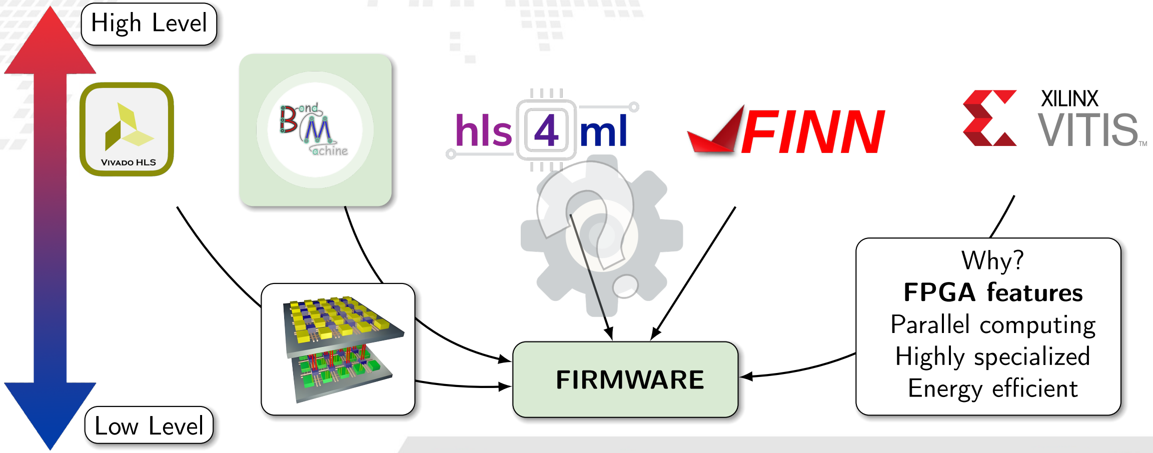
[3]Dipartimento di Farmacia, Universitá degli Studi G. D'Annunzio

Outline

Introduction

FPGA firmware generation

Many projects have the goal of abstracting the firmware generation process. One of them is **BondMachine**, an opensource software framework for the dynamical generation of computer architectures that can be synthesized on one or more FPGAs Workshop sul Calcolo nell'INFN Paestum 2022

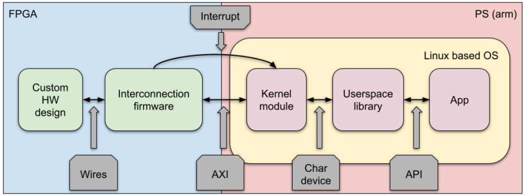


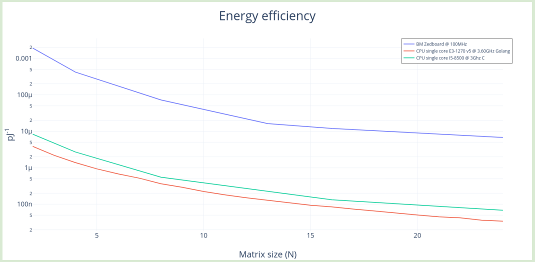
Why?
FPGA features
Parallel computing
Highly specialized
Energy efficient

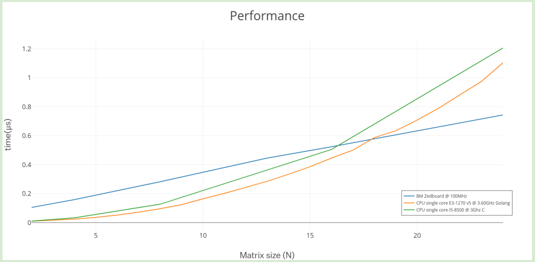
What we did see last year

An accelerated system from ground up

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \left[c_i\right]_{i=1}^{n} = \left[\sum_{k=1}^{n} a_{ik} b_k\right]_{i=1}^{n}$$







Where we left off

In the last slide of last year's talk...

Future directions

We plan to extend the benchmarks to:

- different data types
- different boards
- compare with GPUs
- include some real power consumption measures
- machine Learning Inference on FPGA

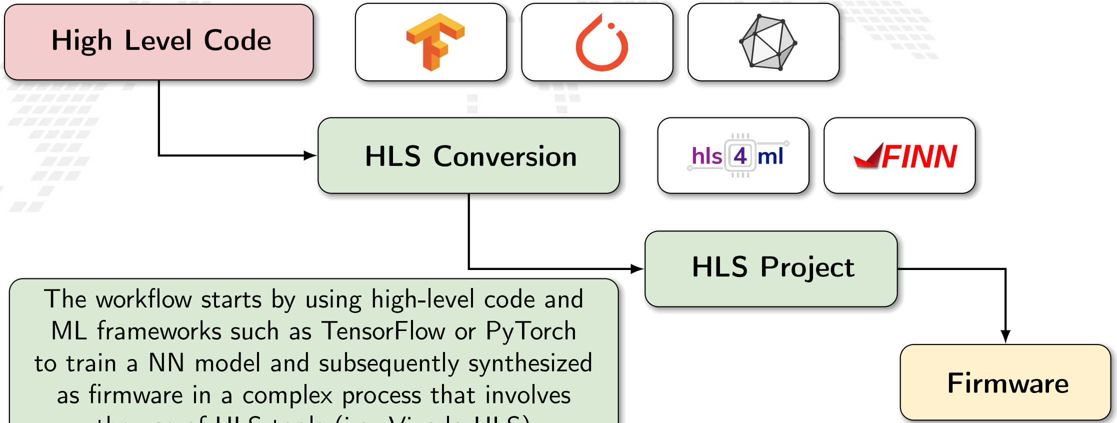
For the project:

- first DAQ use case
- complete the inclusion of Intel and Lattice FPGAs and try a more performant ZYNQ based board
- accelerator in a cloud workflow

Machine Learning Inference on FPGA

Converting NN to HDL

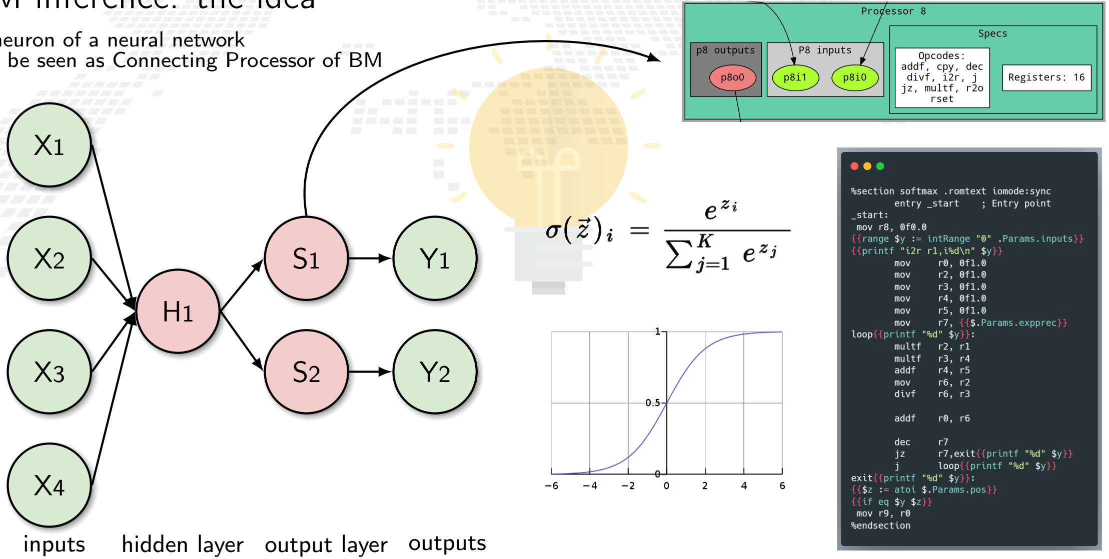
What is the typical process to reach FPGA inference?



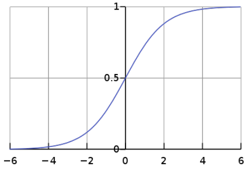
The workflow starts by using high-level code and ML frameworks such as TensorFlow or PyTorch to train a NN model and subsequently synthesized as firmware in a complex process that involves the use of HLS tools (i.e. Vivado HLS) to generate the HDL code

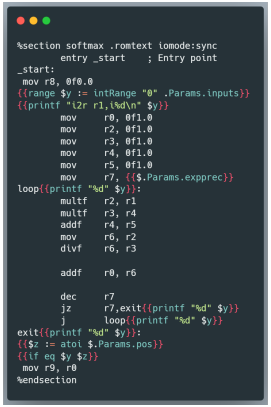
BM inference: the idea

A neuron of a neural network
can be seen as Connecting Processor of BM



$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

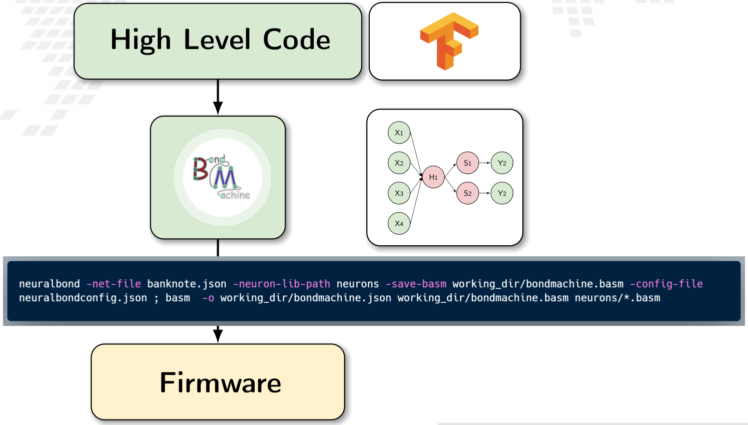




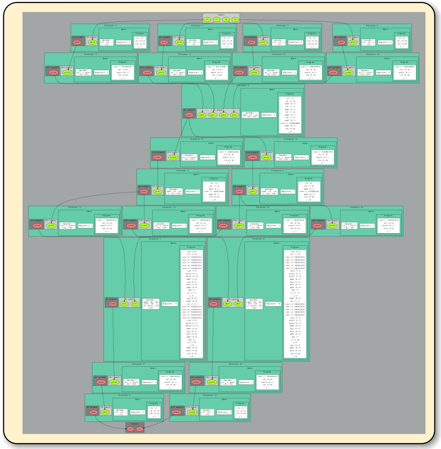
```
%section softmax .romtext iomode:sync
        entry _start    ; Entry point
_start:
 mov r8, 0f0.0
{{range $y := intRange "0" .Params.inputs}}
{{printf "i2r r1,i%d\n" $y}}
        mov     r0, 0f1.0
        mov     r2, 0f1.0
        mov     r3, 0f1.0
        mov     r4, 0f1.0
        mov     r5, 0f1.0
        mov     r7, {{$.Params.expprec}}
loop{{printf "%d" $y}}:
        multf   r2, r1
        multf   r3, r4
        addf    r4, r5
        mov     r6, r2
        divf    r6, r3

        addf    r0, r6

        dec     r7
        jz      r7,exit{{printf "%d" $y}}
        j       loop{{printf "%d" $y}}
exit{{printf "%d" $y}}:
{{$z := atoi $.Params.pos}}
{{if eq $y $z}}
 mov r9, r0
%endsection
```

From idea to implementation

Starting from High Level Code, a NN model trained with **TensorFlow** and exported in a standard interpreted by **neuralbond** that converts nodes and weights of the network into a set of heterogeneous processors.



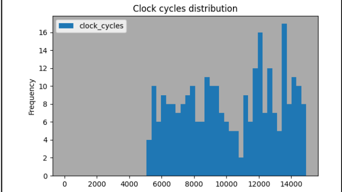
```
neuralbond -net-file banknote.json -neuron-lib-path neurons -save-basm working_dir/bondmachine.basm -config-file
neuralbondconfig.json ; basm  -o working_dir/bondmachine.json working_dir/bondmachine.basm neurons/*.basm
```



Inference evaluation

After checking the correctness of the probabilities of the predictions, we evaluated the implementation with a basic NN and dataset with the following metrics:

- **Inference speed**: time taken to predict a sample i.e. time between the arrival of the input and the change of the output measured with the **benchcore**;
- **Resource usage**: luts and registers in use;
- **Accuracy**: as the average percentage of error on probabilities.



- σ: 2875.94
- Mean: 10268.45 clock cycles
- Latency: 102.68 µs

Resource usage

resource	**value**	**occupancy**
regs	15122	28.42%
luts	11192	10.51%

Is it possibile to optimize this solution? (Yes)

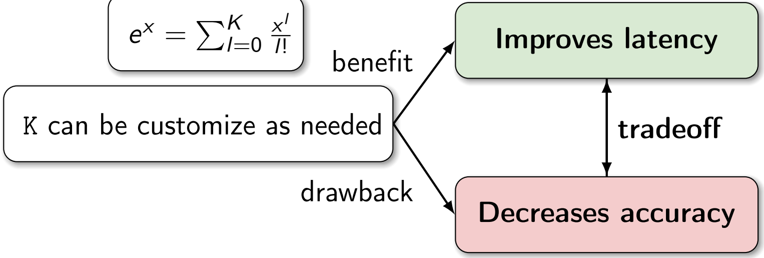
A first example of optimization

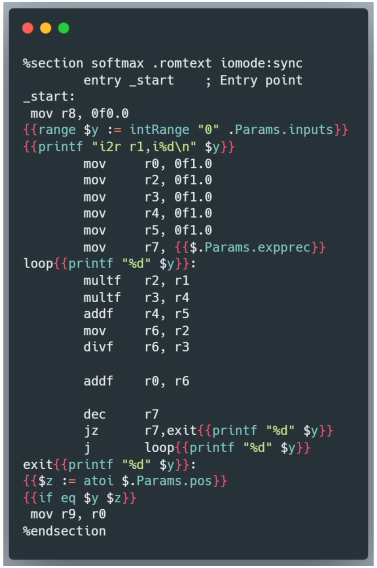
The key feature of our implementation is the high customization even in single neurons.

Remember the **softmax function**

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

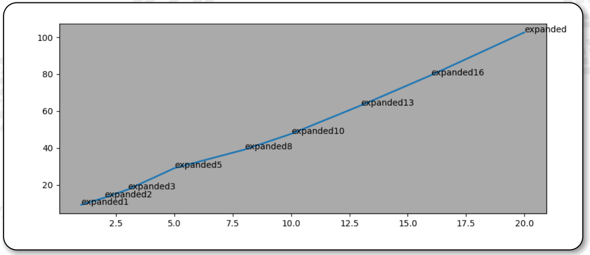
$$e^x = \sum_{l=0}^{K} \frac{x^l}{l!}$$





```
%section softmax .romtext iomode:sync
        entry _start    ; Entry point
_start:
 mov r8, 0f0.0
{{range $y := intRange "0" .Params.inputs}}
{{printf "i2r r1,i%d\n" $y}}
        mov     r0, 0f1.0
        mov     r2, 0f1.0
        mov     r3, 0f1.0
        mov     r4, 0f1.0
        mov     r5, 0f1.0
        mov     r7, {{$.Params.expprec}}
loop{{printf "%d" $y}}:
        multf   r2, r1
        multf   r3, r4
        addf    r4, r5
        mov     r6, r2
        divf    r6, r3

        addf    r0, r6

        dec     r7
        jz      r7,exit{{printf "%d" $y}}
        j       loop{{printf "%d" $y}}
exit{{printf "%d" $y}}:
{{$z := atoi $.Params.pos}}
{{if eq $y $z}}
 mov r9, r0
%endsection
```

Results of optimization

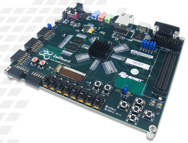


K	Latency	Err prob0	Err prob1	pred
1	9.23 μs	0.0990	0.0990	100%
2	13.11 μs	0.0193	0.0193	100%
3	17.50 μs	0.0053	0.0053	100%
5	29.11 μs	3.1070E-05	3.1071E-05	100%
8	39.13 μs	6.5562E-07	6.6098E-07	100%
10	47.66 μs	1.6162E-07	1.6525E-07	100%
13	63.12 μs	1.6470E-07	1.6652E-07	100%
16	79.46 μs	1.6470E-07	1.6652E-07	100%
20	102.68 μs	1.6470E-07	1.6652E-07	100%

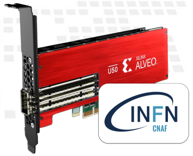
Reduced inference times by a factor of 10 only by decreasing the number of iterations.

Different boards

All tests were done using the **Zedboard** device, but BondMachine supports different boards also from different vendors (Intel lattice).



Xilinx Zynq-7000 SoC
85000 logic cells
53200 look-up tables (LUTs)



PCIe card
2800000 logic cells
1732000 Look-Up Tables (LUTs)
Special thanks to INFN - CNAF
Riccardo Travaglini, Daniele Bonaccorsi, Marco Lorusso, Diego Michelotto, Paolo Veronesi et al.

FPGA cluster ICSC
Xilinx and Intel FPGAs
(Mirko Mariotti)

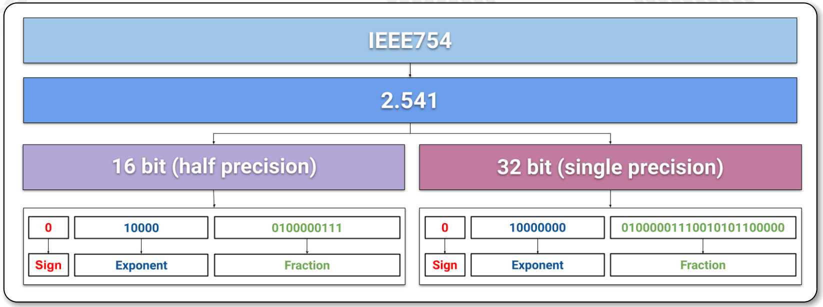
National supercomputing center (ICSC)

Resources are a key aspect and often a bottleneck ...

Model's compression

Why change numerical precision?

The same floating point number can be represented in different ways



Pro

- Reduced memory usage
- Increased computational speed
- Lower power consumption

Cons

- Reduced accuracy
- Increased rounding errors
- Limited range

Floating point FloPoCo

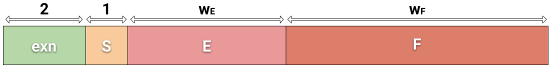
FloPoCo is an open source software project that provides a toolchain for automatically generating floating-point arithmetic operators implemented in hardware.

```
./flopoco pipeline=yes frequency=300 FPAdd wE=8 wF=23
```

```
Final report:
|---Entity RightShifter_24_by_max_26_F300_uid4
|       Pipeline depth = 1
|---Entity IntAdder_27_f300_uid8
|       Not pipelined
|---Entity LZCShifter_28_to_28_counting_32_F300_uid16
|       Pipeline depth = 2
|---Entity IntAdder_34_f300_uid20
|       Not pipelined
Entity FPAdd_8_23_F300_uid2
   Pipeline depth = 6
Output file: flopoco.vhdl
```

Features:

- exponent size and mantissa size can take arbitrary values
- 0, ∞ and `NaN` in explicit exception bits
  - not as special exponent values
  - two more exponent values available in FloPoCo
  - hardware efficient



Tests FloPoCo implementation

We've already seen the pros and cons of changing the numerical precision

Pro

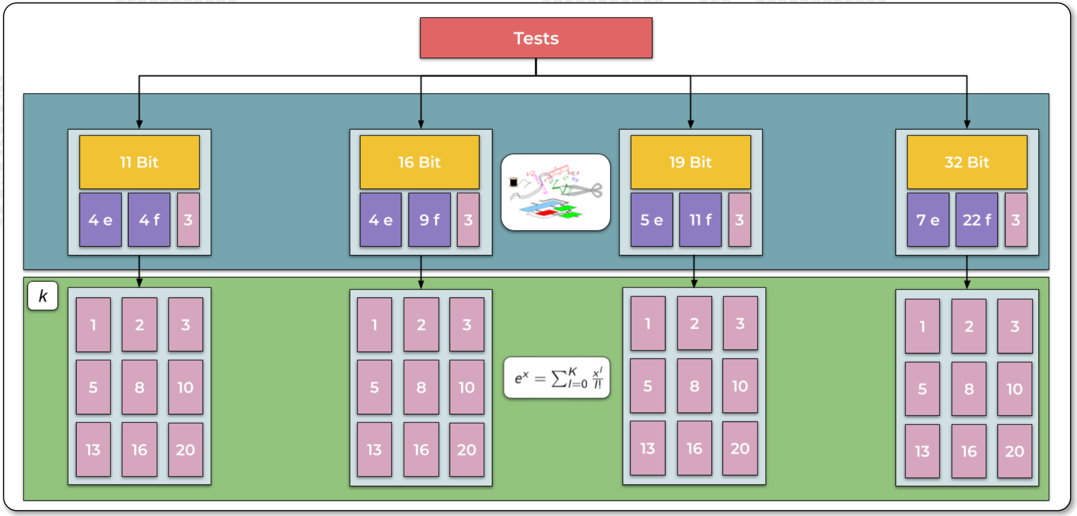
- Reduced memory usage
- **Increased computational speed**
- Lower power consumption

Cons

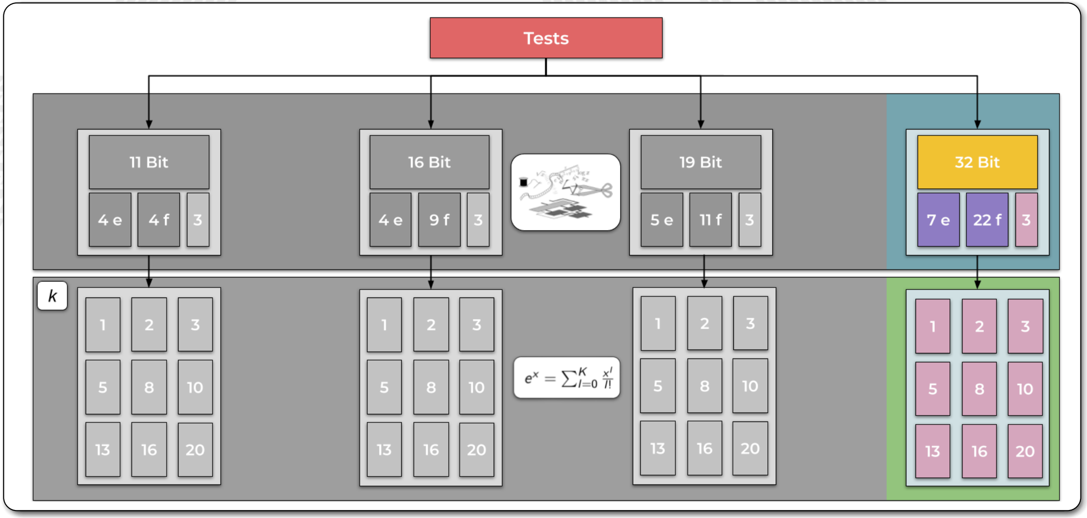
- **Reduced accuracy**
- Increased rounding errors
- Limited range

- How much computationally **faster** are the arithmetic operations implemented by FloPoCo?
- How do **latency**, **accuracy**, occupancy and power consumption vary by changing the numerical precision and the exponent of the exponential?

Tests and results with FloPoCo

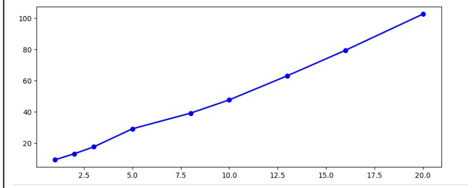


Tests and results with FloPoCo



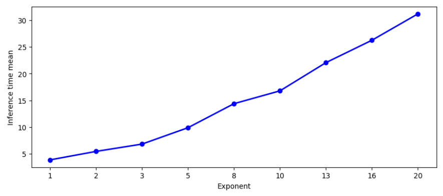
Tests and results with FloPoCo

32bit IEEE754



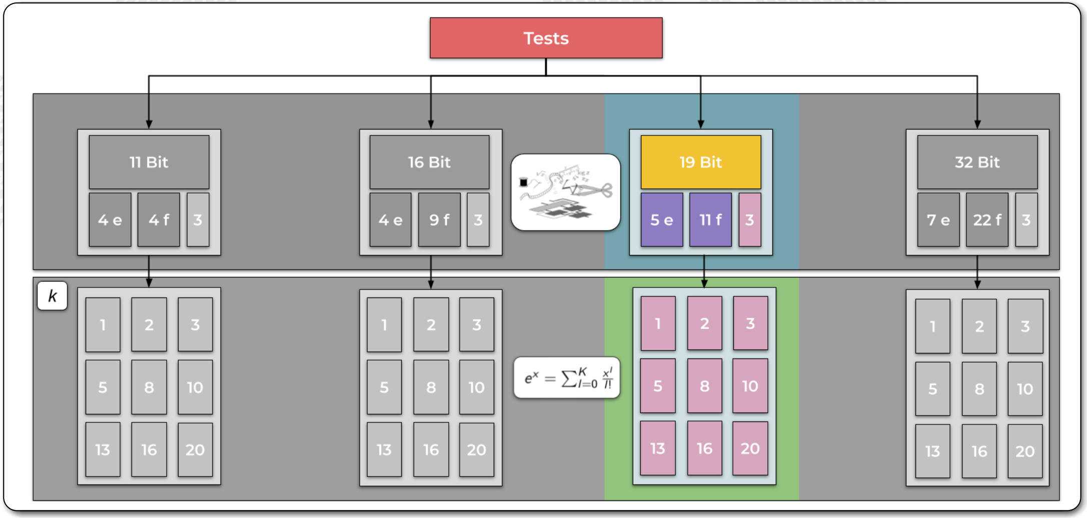
K	Latency	Err prob0	Err prob1
1	9.23 µs	0.0990	0.0990
2	13.11 µs	0.0193	0.0193
3	17.50 µs	0.0053	0.0053
5	29.11 µs	3.1070E-05	3.1071E-05
8	39.13 µs	6.5562E-07	6.6098E-07
10	47.66 µs	1.6162E-07	1.6525E-07
13	63.12 µs	1.6470E-07	1.6652E-07
16	79.46 µs	1.6470E-07	1.6652E-07
20	102.68 µs	1.6470E-07	1.6652E-07

32bit FloPoCo



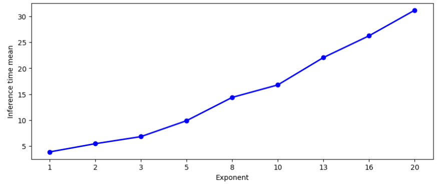
K	Latency	Err prob0	Err prob1
1	3.89 µs	0.0990	0.0990
2	5.47 µs	0.0193	0.0193
3	6.84 µs	0.0053	0.0053
5	9.90 µs	0.0001	0.0001
8	14.39 µs	6.5890E-07	6.5425E-07
10	16.79 µs	1.7316E-07	1.7770E-07
13	22.07 µs	1.7610E-07	1.8029E-07
16	26.25 µs	1.7610E-07	1.8029E-07
20	31.18 µs	1.7610E-07	1.8029E-07

Tests and results with FloPoCo



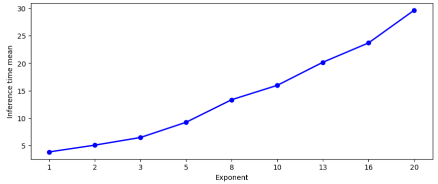
Tests and results with FloPoCo

32bit FloPoCo



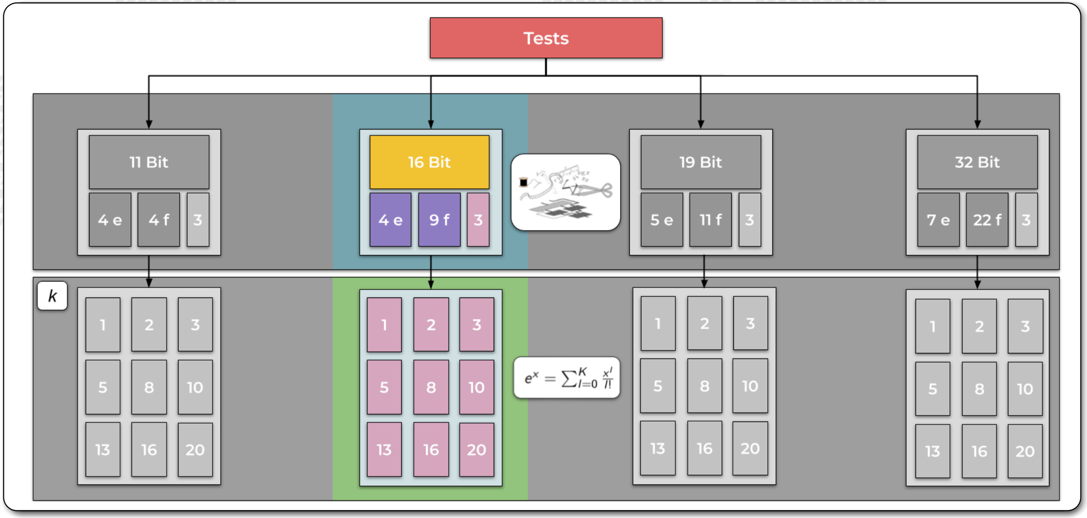
K	Latency	Err prob0	Err prob1
1	3.89 μs	0.0990	0.0990
2	5.47 μs	0.0193	0.0193
3	6.84 μs	0.0053	0.0053
5	9.90 μs	0.0001	0.0001
8	14.39 μs	6.5890E-07	6.5425E-07
10	16.79 μs	1.7316E-07	1.7770E-07
13	22.07 μs	1.7610E-07	1.8029E-07
16	26.25 μs	1.7610E-07	1.8029E-07
20	31.18 μs	1.7610E-07	1.8029E-07

19bit FloPoCo



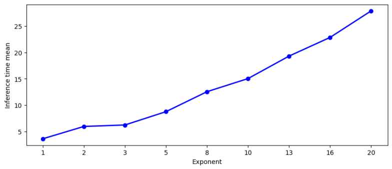
K	Latency	Err prob0	Err prob1
1	3.80 μs	0.1229	0.009
2	5.04 μs	0.0193	0.0193
3	6.44 μs	0.0054	0.0054
5	9.21 μs	0.00024	0.00025
8	13.33 μs	0.00010	9.9151E-05
10	15.95 μs	0.00010	9.9151E-05
13	20.17 μs	0.00010	9.9151E-05
16	23.70 μs	0.00010	9.9151E-05
20	29.67 μs	0.00010	9.9151E-05

Tests and results with FloPoCo



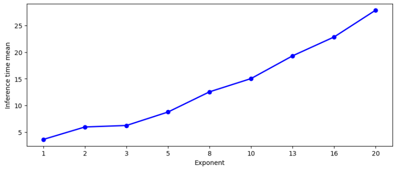
Tests and results with FloPoCo

19bit FloPoCo



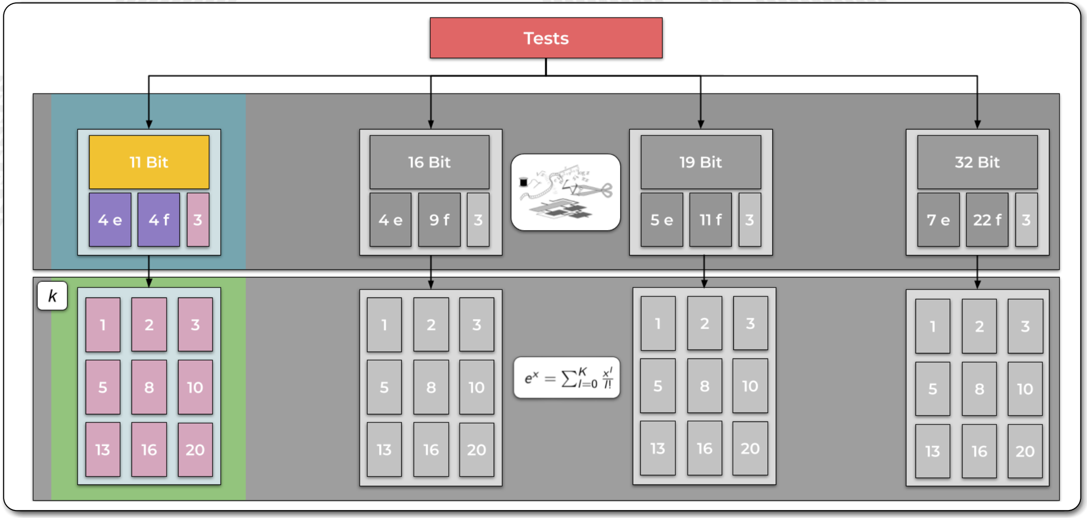
K	Latency	Err prob0	Err prob1
1	3.80 µs	0.1229	0.009
2	5.04 µs	0.0193	0.0193
3	6.44 µs	0.0054	0.0054
5	9.21 µs	0.00024	0.00025
8	13.33 µs	0.00010	9.9151E-05
10	15.95 µs	0.00010	9.9151E-05
13	20.17 µs	0.00010	9.9151E-05
16	23.70 µs	0.00010	9.9151E-05
20	29.67 µs	0.00010	9.9151E-05

16bit FloPoCo



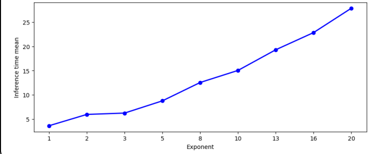
K	Latency	Err prob0	Err prob1	Pred
1	3.59 µs	1.3935	0.099	99.27%
2	5.93 µs	0.0192	0.0191	100%
3	6.21 µs	0.0057	0.0057	100%
5	8.74 µs	0.00125	0.0019	100%
8	12.54 µs	0.00125	0.0019	100%
10	15.04 µs	0.0012	0.0019	100%
13	19.32 µs	0.0026	0.0025	99.63%
16	22.87 µs	0.0037	1.8113	99.63%
20	27.91 µs	0.0060	4.1385	98.54%

Tests and results with FloPoCo



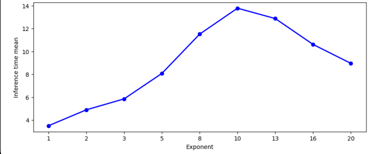
Tests and results with FloPoCo

16bit FloPoCo



K	Latency	Err prob0	Err prob1	Pred
1	3.59 µs	1.3935	0.099	99.27%
2	5.93 µs	0.0192	0.0191	100%
3	6.21 µs	0.0057	0.0057	100%
5	8.74 µs	0.00125	0.0019	100%
8	12.54 µs	0.00125	0.0019	100%
10	15.04 µs	0.0012	0.0019	100%
13	19.32 µs	0.0026	0.0025	99.63%
16	22.87 µs	0.0037	1.8113	99.63%
20	27.91 µs	0.0060	4.1385	98.54%

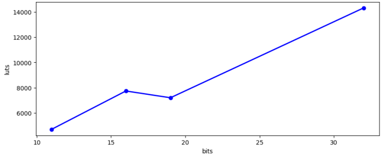
11bit FloPoCo

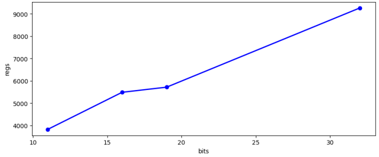


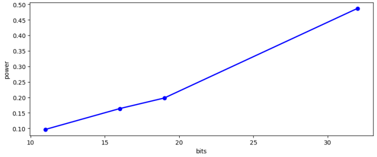
K	Latency	Err prob0	Err prob1	Pred
1	3.49 µs	0.2235	0.0992	97.45%
2	4.88 µs	0.0221	0.0168	98.54%
3	5.84 µs	0.0156	0.0126	98.54%
5	8.07 µs	0.0138	0.0110	98.54%
8	11.51 µs	0.0138	0.0110	98.54%
10	13.78 µs	0.0138	0.0110	98.54%
13	12.87 µs	0.0175	1.5069	97.09%
16	10.61 µs	0.0187	2.5789	96.72%
20	8.95 µs	0.0273	1.223	94.90%

Results with FloPoCo

How do latency, accuracy, **occupancy** and **power consumption** vary by changing the numerical precision ?







Bits	Luts	Usage
11	4704	8.84%
16	7738	14.54%
19	7202	13.54%
32	14306	26.89%

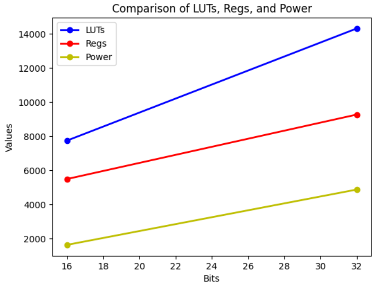
Bits	Regs	Usage
11	3828	3.59%
16	5487	5.15%
19	5717	5.37%
32	9264	8.70%

Bits	Power
11	0.096 W
16	0.163 W
19	0.198 W
32	0.487 W

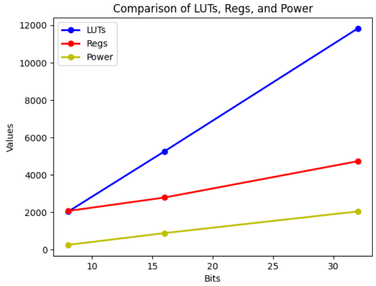
Quantization: tests, results and analysis

Linear quantization **reduces memory usage and computational complexity** by representing values with fewer bits, enabling efficient deployment on resource constrained devices (but it may introduce some loss of accuracy)

FloPoCo



Quantization



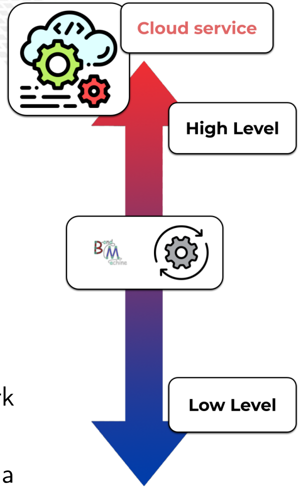
FloPoCo

Bits	Luts	Regs	Power	Latency	Pred
16	7738 (14%)	5487 (5%)	0.163W	6.21 μs	100%
32	14306 (26%)	9264 (8%)	0.487W	6.84 μs	100%

Bits	Luts	Regs	Power	Latency	Pred
8	2013 (3%)	2054 (2%)	0.024W	1.60 μs	91%
16	5259 (9%)	2774 (3%)	0.087W	1.60 μs	99%
32	11823 (22%)	4718 (5%)	0.203W	1.61 μs	99%

Wrap up

- Firmware generation for accelerated computing (**low level**)
  - Accelerated system from ground up and Machine Learning Inference on FPGA
    - highly customizable according to needs
    - finding the right balance between accuracy, resource utilization, and latencies
- Easy to use for the user (**high level**)
  - automations everything simple
    - Firmware generation
    - Jupyter Notebooks for testing the generated firmware and collecting data
    - Jupyter Notebooks to analyze the data
  - starting from high-level code with the most well-known framework
- We want to move even higher
  - To further abstract the complexity for the user by implementing a **cloud service**



Accelerator in a cloud workflow

Bring it to cloud level: why?

So we "know" how to build firmware for ML inference in a vendor agnostic way. Can we integrate it with cloud-native inference as-a-service solution to get any advantage?

- **Ease of usage and flexibility**
  - Being able to deploy an inference algorithm on FPGA without caring for "where" the resources are
  - Accessing ML predictions from a remote computing resource without having in place any specialized hardware or software piece
    - At the cost of increased latency $\rightarrow$ to be carefully evaluated case by case
  - Sharing the access to the same model predictions with other collaborators
- **Democratic access and management**
  - Leveraging cloud/k8s native tools, you can reuse a well established way to orchestrate the bookkeeping and distribution of the payloads
- **Easy Prototyping**
  - Automation of the build and load process -> the framework take care of vendor specific details

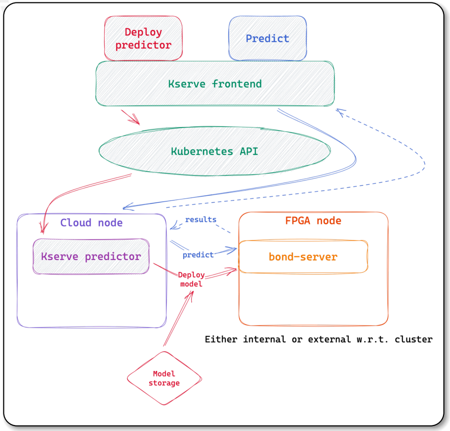
Implementing a KServe FPGA extension

The remote inference still an open field on many aspects, regardless we started from one of the main emerging ecosystems for ML: **Kubeflow**

KServe in particular is the component responsible for providing inference endpoint as-a-Service

Our simple workflow:

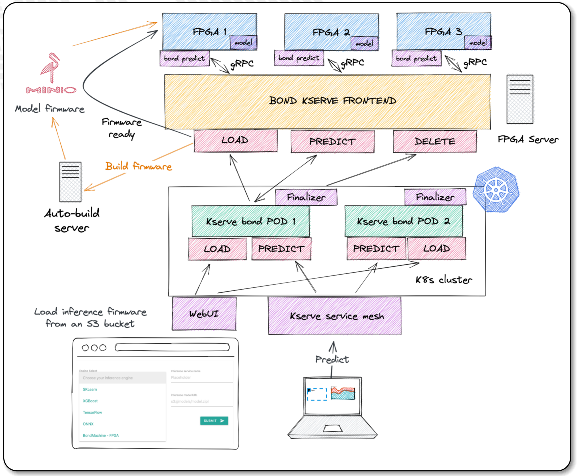
1. **Train your model** with your preferred framework (e.g. TF)
2. **Store the model** on a remote storage
   - S3 storage is the one used for our tests
3. Deploying the **same model on a remote FPGA via a user friendly UI**
4. Get back the **details of the endpoint to interact with**
   - Either via HTTP or grpc protocols



Kserve extension implementation

The main components that we developed are:

- **Custom WebUI** to hide complexity to the user
  - A Kubeflow managed solution exists, we are planning to integrate this work eventually
    - We need additional metadata to be passed (e.g. board model, provider, hls engine etc)
- Translate a **model load** request into conditional actions
  - Load the bitstream file from the remote location directly
    - Pre built by the user on its own
  - **building a firmware** "seamlessly" on an external building machine
- **Eventually load the firmware** on the FPGA board via the development of a grpc server installed on the machine that have access to the board



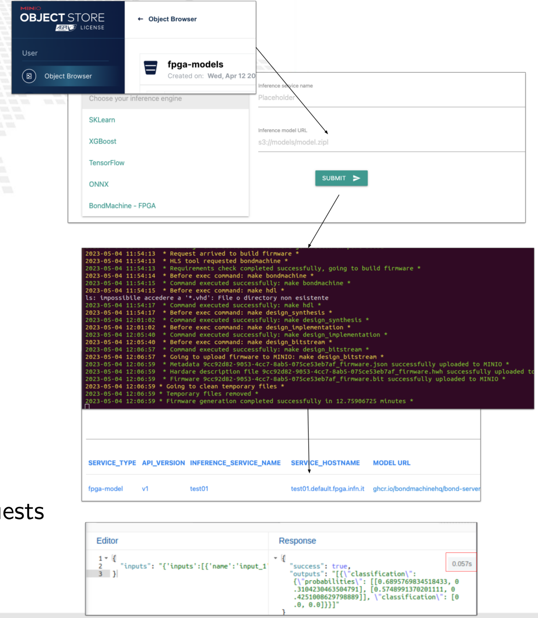
[CHEP 2023](#)

Where are we...

We have **validated an end to end workflow with a generic ML** algorithm.
With the following steps:

- Load the model description to an S3 bucket
- Report the model URL and name in the WebUI
  - Selecting HLS engine (BM in this case)
- Wait for the build server to build and store your firmware for the available FPGAs
  - Store back the firmware on S3 bucket for further reuse
  - Load the created firmware on a FPGA
- Publish the endpoint to send the prediction requests to and then do your prediction.

CHEP 2023



```
2023-05-04 11:54:13  * Request arrived to build firmware *
2023-05-04 11:54:13  * HLS tool requested bondmachine *
2023-05-04 11:54:13  * Requirements check completed successfully, going to build firmware *
2023-05-04 11:54:14  * Before exec command: make bondmachine *
2023-05-04 11:54:15  * Command executed successfully: make bondmachine *
2023-05-04 11:54:15  * Before exec command: make hdl *
ls: impossibile accedere a '*.vhd': File o directory non esistente
2023-05-04 11:54:17  * Command executed successfully: make hdl *
2023-05-04 11:54:17  * Before exec command: make design_synthesis *
2023-05-04 12:01:02  * Command executed successfully: make design_synthesis *
2023-05-04 12:01:02  * Before exec command: make design_implementation *
2023-05-04 12:05:40  * Command executed successfully: make design_implementation *
2023-05-04 12:05:40  * Before exec command: make design_bitstream *
2023-05-04 12:06:57  * Command executed successfully: make design_bitstream *
2023-05-04 12:06:57  * Going to upload firmware to MINIO: make design_bitstream *
2023-05-04 12:06:59  * Metadata 9cc92d82-9053-4cc7-8ab5-075ce53eb7af_firmware.json successfully uploaded to MINIO *
2023-05-04 12:06:59  * Hardare description file 9cc92d82-9053-4cc7-8ab5-075ce53eb7af_firmware.hwh successfully uploaded to
2023-05-04 12:06:59  * Firmware 9cc92d82-9053-4cc7-8ab5-075ce53eb7af_firmware.bit successfully uploaded to MINIO *
2023-05-04 12:06:59 * Going to clean temporary files *
2023-05-04 12:06:59 * Temporary files removed *
2023-05-04 12:06:59 * Firmware generation completed successfully in 12.75906725 minutes *
```

SERVICE_TYPE	API_VERSION	INFERENCE_SERVICE_NAME	SERVICE_HOSTNAME	MODEL URL
fpga-model	v1	test01	test01.default.fpga.infn.it	ghcr.io/bondmachinehq/bond-server

```
{
  "inputs": "{'inputs':[{'name':'input_1'
}
```

```
{
  "success": true,
  "outputs": "[{\"classification\":
   {\"probabilities\": [[0.6895769834518433, 0
   .3104230463504791], [0.5748991370201111, 0
   .4251008629798889]], \"classification\": [0
   .0, 0.0]}}]"
}
```

Conclusions and Future directions

Future directions

- More tests and work on numerical precision
  - add more numeric types and try more numerical precisions
  - try more quantization technique
- Consolidate the work done and improve portability
  - extend the automatisms and finalize the implementation on Alveo
  - make everything adaptable for FPGA clusters (BM is a multi-fpga system)
  - support more boards to spread our solution
  - test our solutions on ICSC (Spoke0) resources
- New estimates on energy consumption
  - Move from software energy estimates to real energy measurements
- For the cloud service implementation...
  - leveraging the kserve extension also for use cases beyond inference
  - FPGA bookkeeping
  - Systematic measurements of performances at the various stage of the chain

Thank you

Thank you

website: `http://bondmachine.fisica.unipg.it`
code: `https://github.com/BondMachineHQ`
parallel computing paper: link
contact email: giulio.bianchini@studenti.unipg.it, mirko.mariotti@unipg.it

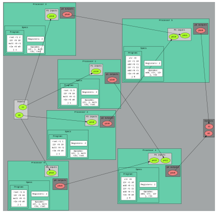
Backup

Backup

BondMachine

Open source software framework for the dynamical generation of computer architectures that can be synthesized on one or more FPGAs.

- High level programming language (Golang) for both the hardware and software
- Functional style programming
- Computational graph and Neural Networks
- Architecture generating compiler



- CCR
  - 2015 First ideas
  - 2016 Poster
  - 2017 Talk
  - 2022 Talk
  - 2023 Talk (Today)
- InnovateFPGA 2018 Iron Award, Grand Final at Intel Campus (CA) USA
- Invited lectures at FPGA workshops ICTP 2019 and 2022
- Golab 2018 talk and ISGC 2019 PoS
- Article published on Parallel Computing, Elsevier 2022 DOI:10.22323/1.351.0020

A simple example

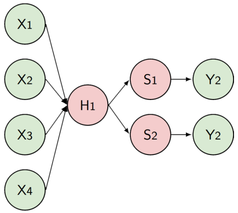
Dataset info:

- **Dataset name**: Banknote Authentication
- **Description**: Dataset on the distinction between genuine and counterfeit banknotes. The data was extracted from images taken from genuine and fake banknote-like samples.
- **N. features**: 4
- **Classification**: binary
- **Samples**: 1097

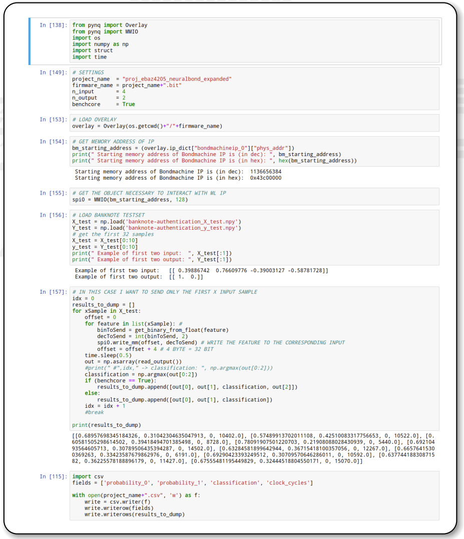
Neural network info:

- **Class**: Multilayer perceptron fully connected
- **Layers**:
  1. An hidden layer with 1 **linear** neuron
  2. One output layer with 2 **softmax** neurons

Graphic representation:



Make predictions and check correctness



```
In [138]: from pynq import Overlay
          from pynq import MMIO
          import os
          import numpy as np
          import struct
          import time

In [149]: # SETTINGS
          project_name  = "proj_ebaz4205_neuralbond_expanded"
          firmware_name = project_name+".bit"
          n_input       = 4
          n_output      = 2
          benchcore     = True

In [153]: # LOAD OVERLAY
          overlay = Overlay(os.getcwd()+"/"+firmware_name)

In [154]: # GET MEMORY ADDRESS OF IP
          bm_starting_address = (overlay.ip_dict["bondmachineip_0"]["phys_addr"])
          print(" Starting memory address of Bondmachine IP is (in dec): ", bm_starting_address)
          print(" Starting memory address of Bondmachine IP is (in hex): ", hex(bm_starting_address))

           Starting memory address of Bondmachine IP is (in dec):  1136656384
           Starting memory address of Bondmachine IP is (in hex):  0x43c00000

In [155]: # GET THE OBJECT NECESSARY TO INTERACT WITH ML IP
          spi0 = MMIO(bm_starting_address, 128)

In [156]: # LOAD BANKNOTE TESTSET
          X_test = np.load('banknote-authentication_X_test.npy')
          Y_test = np.load('banknote-authentication_y_test.npy')
          # get the first 32 samples
          X_test = X_test[0:10]
          y_test = Y_test[0:10]
          print(" Example of first two input:  ", X_test[:1])
          print(" Example of first two output: ", Y_test[:1])

           Example of first two input:   [[ 0.39886742  0.76609776 -0.39003127 -0.58781728]]
           Example of first two output:  [[ 1.  0.]]

In [157]: # IN THIS CASE I WANT TO SEND ONLY THE FIRST X INPUT SAMPLE
          idx = 0
          results_to_dump = []
          for xSample in X_test:
              offset = 0
              for feature in list(xSample): #
                  binToSend = get_binary_from_float(feature)
                  decToSend = int(binToSend, 2)
                  spi0.write_mm(offset, decToSend) # WRITE THE FEATURE TO THE CORRESPONDING INPUT
                  offset = offset + 4 # 4 BYTE = 32 BIT
              time.sleep(0.5)
              out = np.asarray(read_output())
              #print(" #",idx," -> classification: ", np.argmax(out[0:2]))
              classification = np.argmax(out[0:2])
              if (benchcore == True):
                  results_to_dump.append([out[0], out[1], classification, out[2]])
              else:
                  results_to_dump.append([out[0], out[1], classification])
              idx = idx + 1
              #break

          print(results_to_dump)

          [[0.68957698345184326, 0.31042304635047913, 0, 10402.0], [0.57489913702011108, 0.42510083317756653, 0, 10522.0], [0.
          60581505298614502, 0.39418494701385498, 0, 8728.0], [0.78091907501220703, 0.21908088028430939, 0, 5440.0], [0.692104
          93564605713, 0.30789506435394287, 0, 14502.0], [0.63284581899642944, 0.36715418100357056, 0, 12267.0], [0.6657641530
          0369263, 0.33423587679862976, 0, 6191.0], [0.69290423393249512, 0.30709570646286011, 0, 10592.0], [0.637744188308715
          82, 0.36225578188896179, 0, 11427.0], [0.67555481195449829, 0.32444518804550171, 0, 15070.0]]

In [115]: import csv
          fields = ['probability_0', 'probability_1', 'classification', 'clock_cycles']

          with open(project_name+".csv", 'w') as f:
              write = csv.writer(f)
              write.writerow(fields)
              write.writerows(results_to_dump)
```

🟥 Thanks to PYNQ we can easily load the bitstream and program the FPGA in real time.

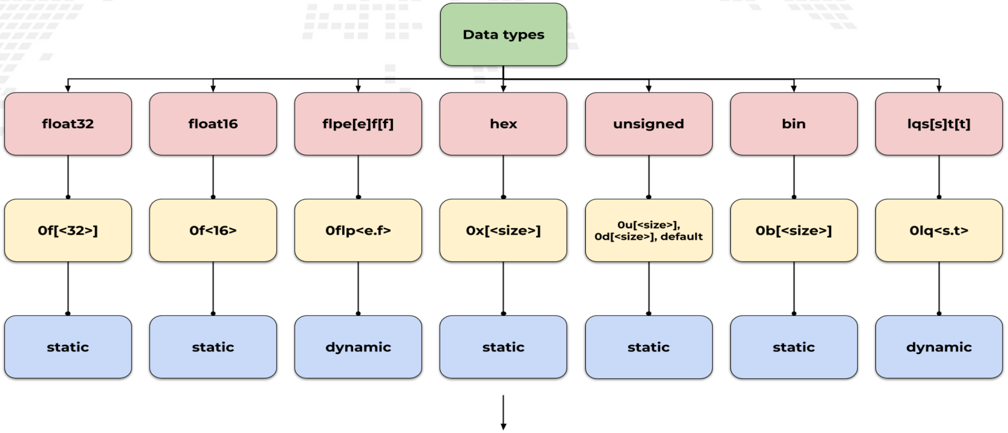
🟨 With their APIs we interact with the memory addresses of the BM IP to send data into the inputs and read the outputs

🟦 Dump output results for future analysis

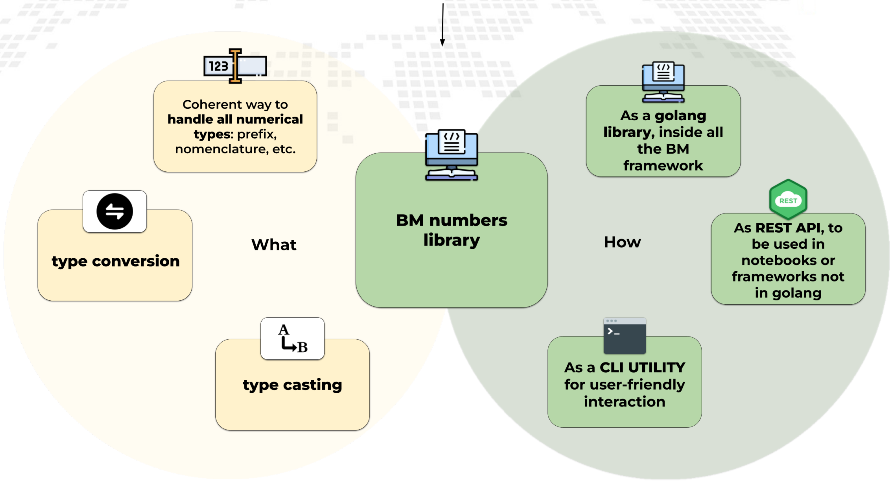
Software			BondMachine		
prob0	prob1	class	prob0	prob1	class
0.6895	0.3104	0	0.6895	0.3104	0
0.5748	0.4251	0	0.5748	0.4251	0
0.4009	0.5990	1	0.4009	0.5990	1

The output of the bm corresponds to the software output:
it works!

Data types in BondMachine: BMnumbers



Data types in BondMachine: BMnumbers



Coherent way to **handle all numerical types**: prefix, nomenclature, etc.

type conversion

What

type casting

BM numbers library

How

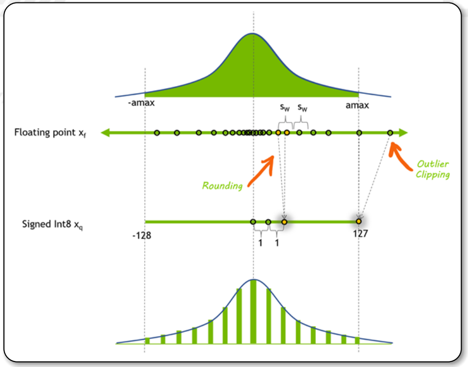
As a golang library, inside all the BM framework

As **REST API**, to be used in notebooks or frameworks not in golang

As a **CLI UTILITY** for user-friendly interaction

Linear quantization

Linear quantization is a widely used technique in signal processing, in particular in neural networks **reduces memory usage and computational complexity** by representing values with fewer bits, enabling **efficient deployment on resource-constrained devices** (but it may introduce some loss of accuracy).



BMnumbers translates the floating point number into the quantized equivalent using the data type `lqs[s]t[t]`

```
bmnumbers --show native -cast lqs16t1 -linear-data-range 1,ranges.txt "0b<16>010010110"

0lq<16.1>13.73291015625
```

Corrected signed integer instructions are used in hardware

Quantized networks can be **simulated** to check if the precision is acceptable.