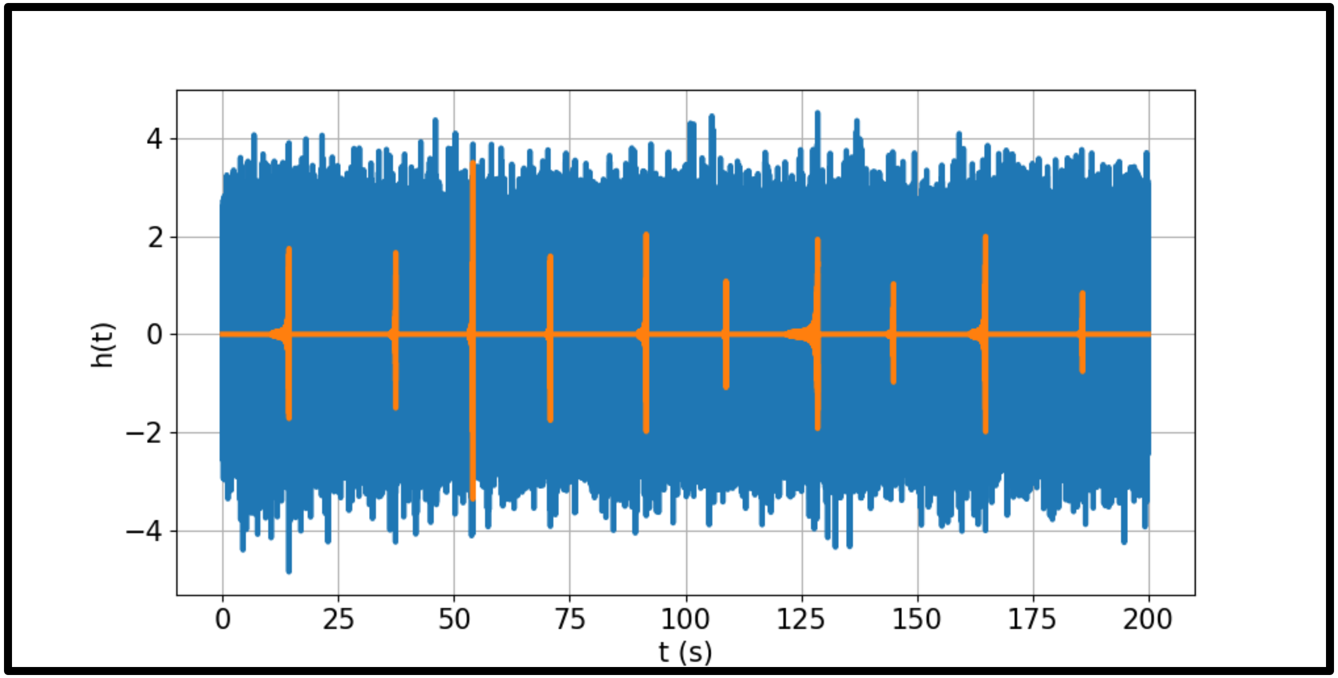
Neural network inference on FPGA
Application to gravitational wave detection



A.Boudon / Q.David / G.Galbit / G.Joubert / S.Viret
IP2I Lyon

→ AI and FPGAs: a long story:

→ Back in the 80s, first neural network inference implementations were done on FPGAs

→ GPUs washed away all that, but FPGAs recently reclaimed some popularity, in particular in HEP (*eg* ***HLS4ML***).

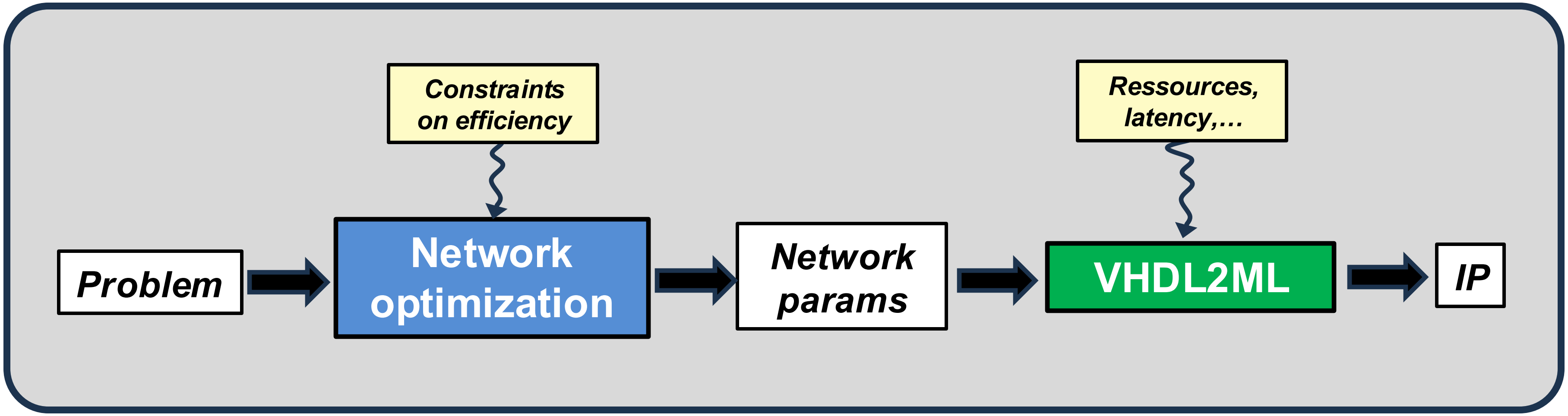
→ Indeed, when you have **low latency** and **large bandwidth**, FPGAs are far more adapted than GPUs.

→ But it comes to a price:

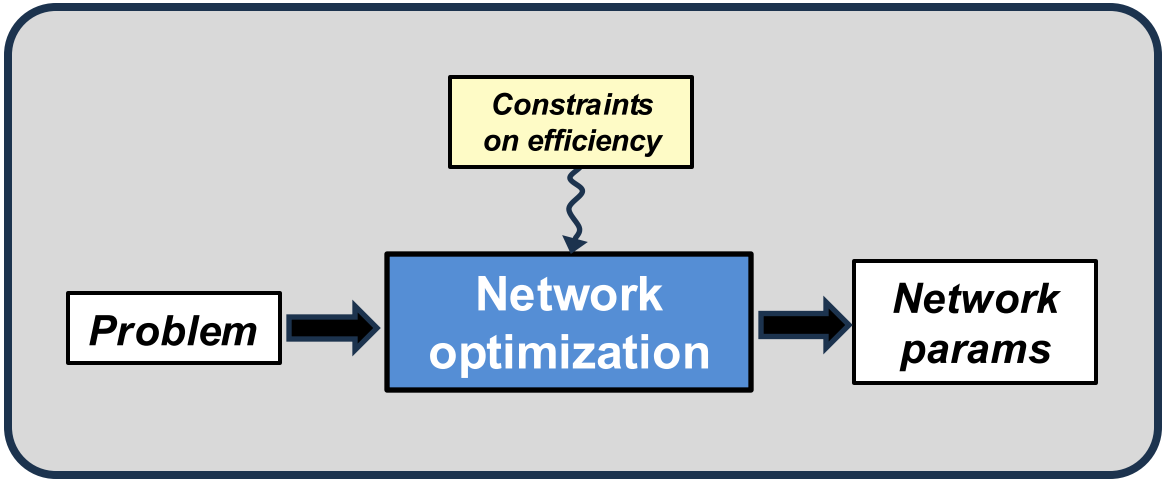
- Working on FPGA is rather more complex
- You can implement **only simple architectures** unless you have big (*and expensive*) FPGAs. Certainly not a big issue for LHC L1 trigger applications, but possibly a limitation for others

→ **Our approach:**

→ Try to develop a simple approach to implement complex neural network architecture on small FPGAs. Keep **low latency** and **good efficiency**, but also aim for **less power consumption**.



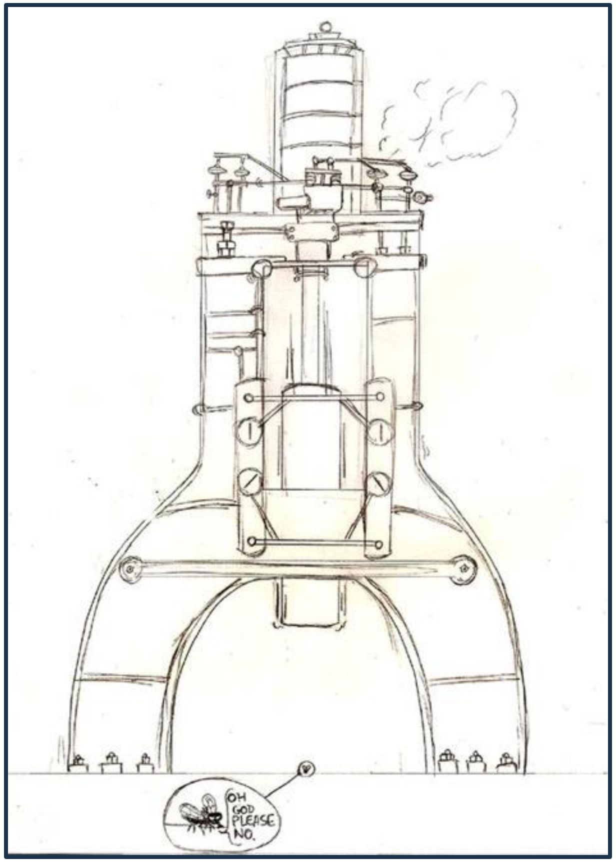
→ This work is done within the IN2P3 THINK2 R&T program (*see* [***https://think.in2p3.fr/***](https://think.in2p3.fr/) *and* [***https://gitlab.in2p3.fr/think2***](https://gitlab.in2p3.fr/think2) *for more info*)



A.Boudon, G.Joubert, SV

→ **Network optimization**

→ **A common mistake in machine learning is to start from a network far more powerful than what you need.**



→ Choosing the good model is a crucial step, often overlooked

→ The key there is to properly define what you have in hand, and what you are looking for.

→ From there you can define the most adapted architecture. There is no magic for this, you need to learn a bit, but there are some nice ressources (*eg:* ***https://d2l.ai/***)

→ **Network optimization**

→ OK you have the right network, but now how do you to dimension it?

→ **No magic here too, a**nd actually even less litterature. Heuristic approach is most common strategy (*the hammer, again…*), you test 10's of networks, and take the best one.

→ **How you define best is also a good question**. Are you just interested in efficiency, or also on fake rate minimisation,… For FPGA number of parameters, operations per inference, etc… are also important. How do you account for them?

→ For simple architecture, a bit of thinking can help a lot

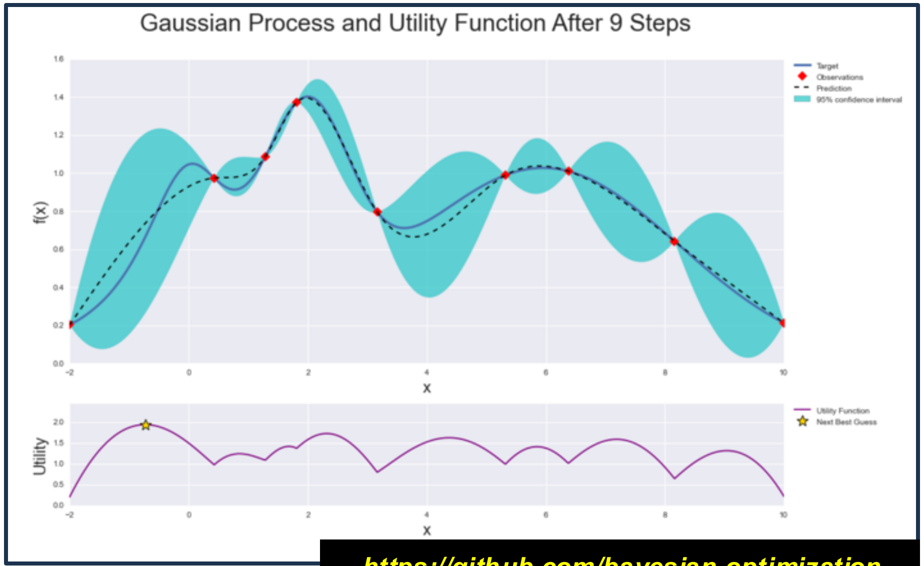
→ But when you start to play with complex neurons and/or large structure, this approach is quickly becoming cumbersome.

→ Simplifying network optimisation with bayesian approach

→ **Bayesian optimization** can help you to converge towards an optimal network topology

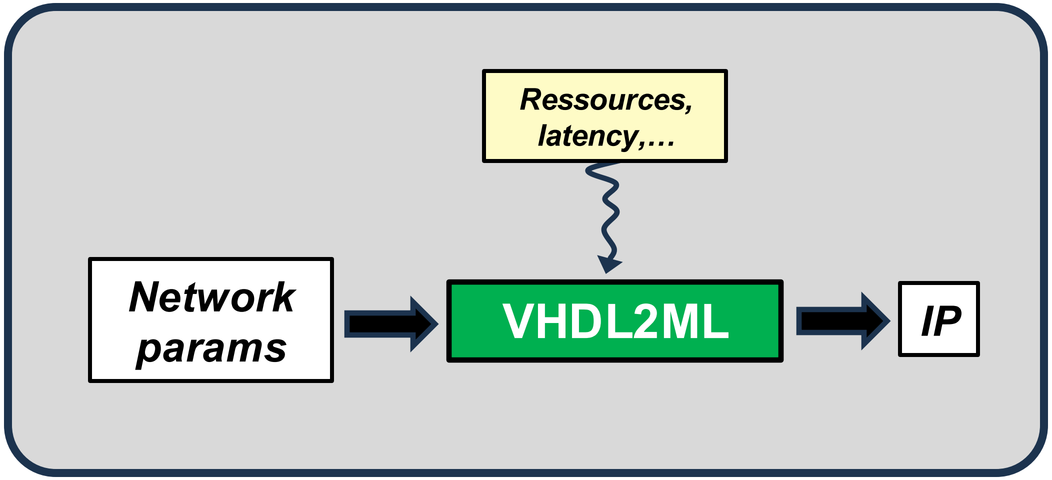
→ The optimization process tries to maximise a cost (*utility*) function. You can put the network efficiency there, but also sizing info.

→ You end up with a network combining **good efficiency**, but also optimized **inference complexity**



https://github.com/bayesian-optimization

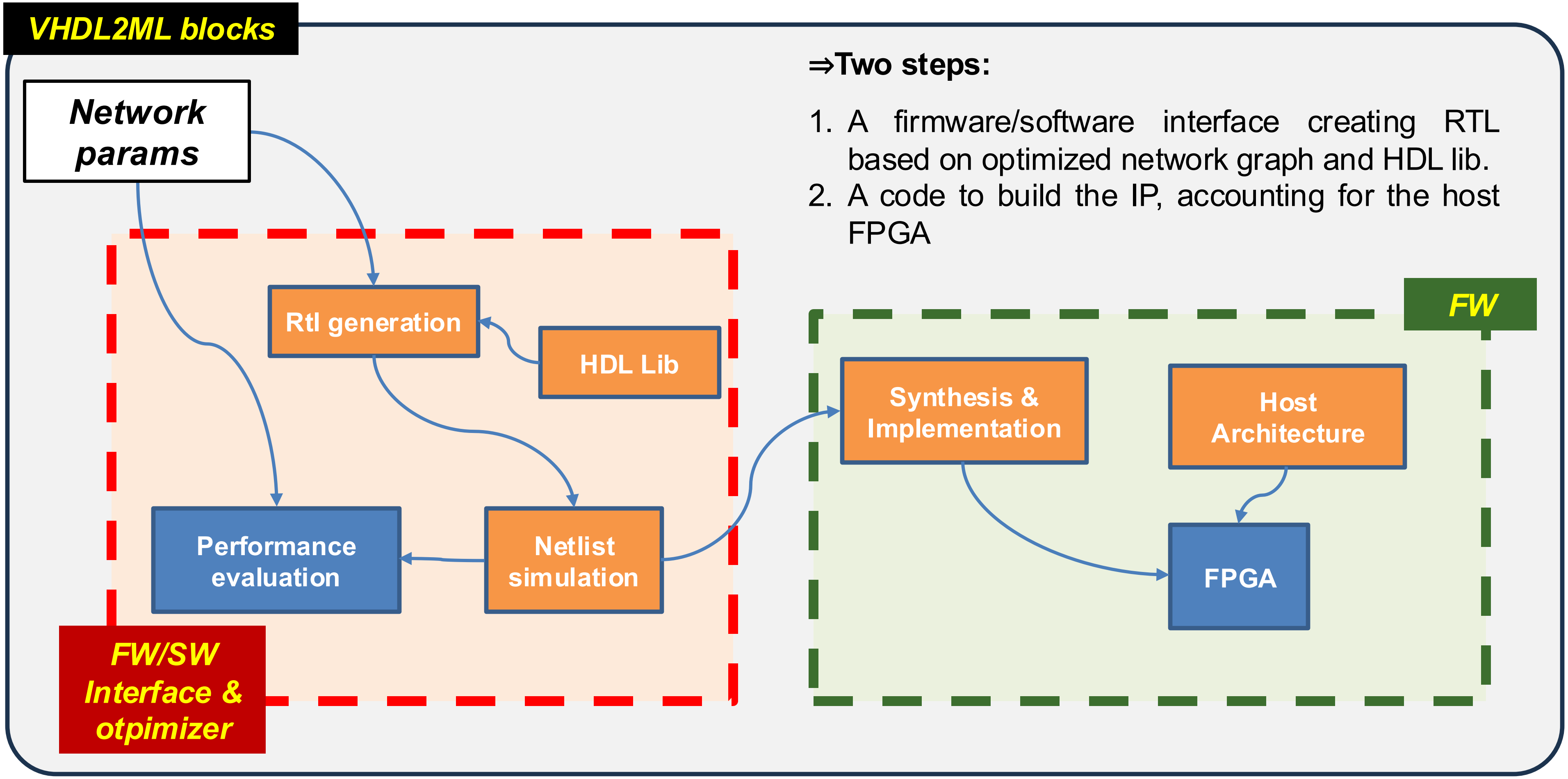
→ Applying this method to any type of network provides you with the **most FPGA-friendly topology.** It's mandatory to have this done before going to the next stage.



Q.David, G.Galbit

→ VHDL4ML implementation

→ The key elements here are the **HDL library** and the **graph optimizer system**.



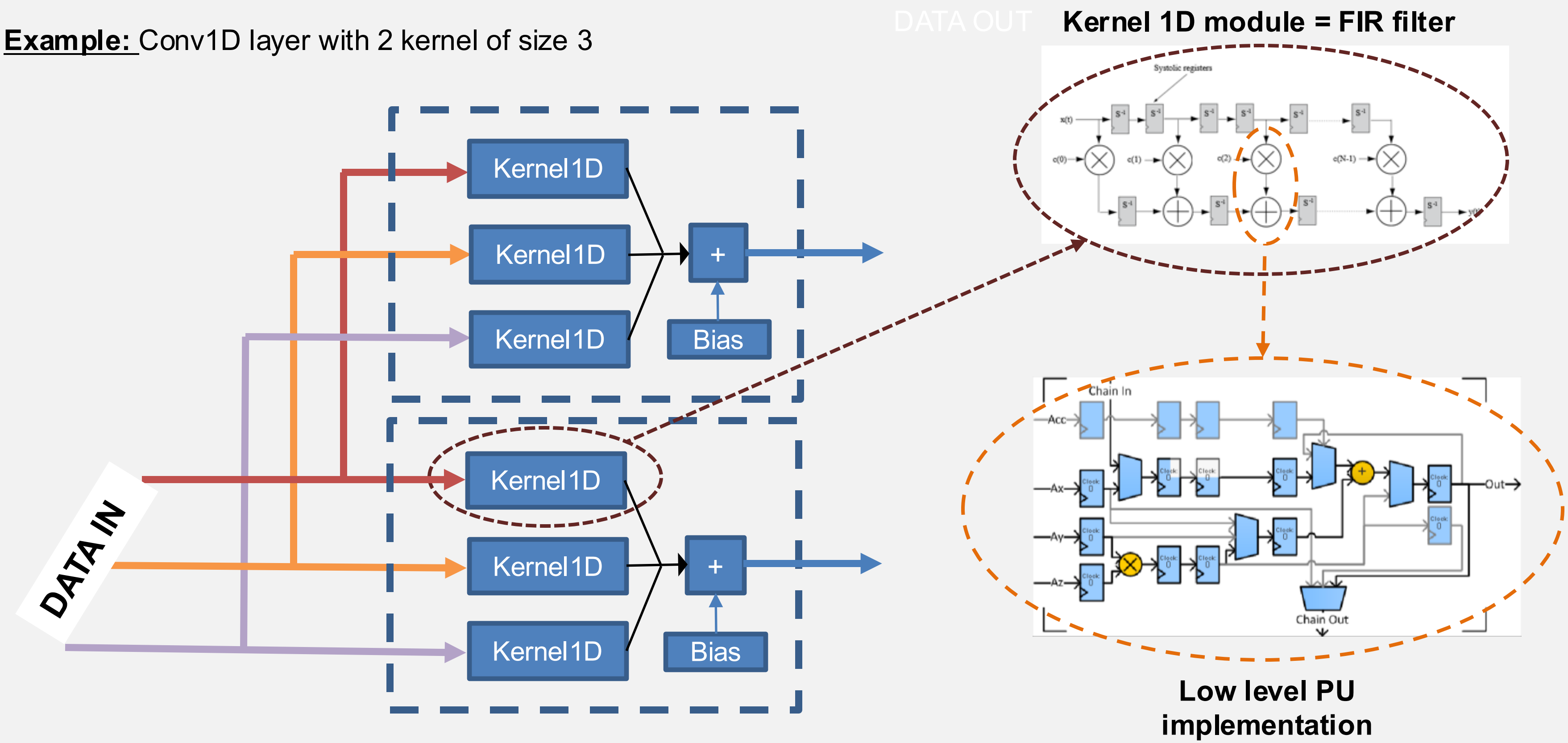
⇒**Two steps:**

1. A firmware/software interface creating RTL based on optimized network graph and HDL lib.
2. A code to build the IP, accounting for the host FPGA

→ HDL library

→ The library contains all the basic processing units (**PU**) necessary to build an **RTL version of neurons**

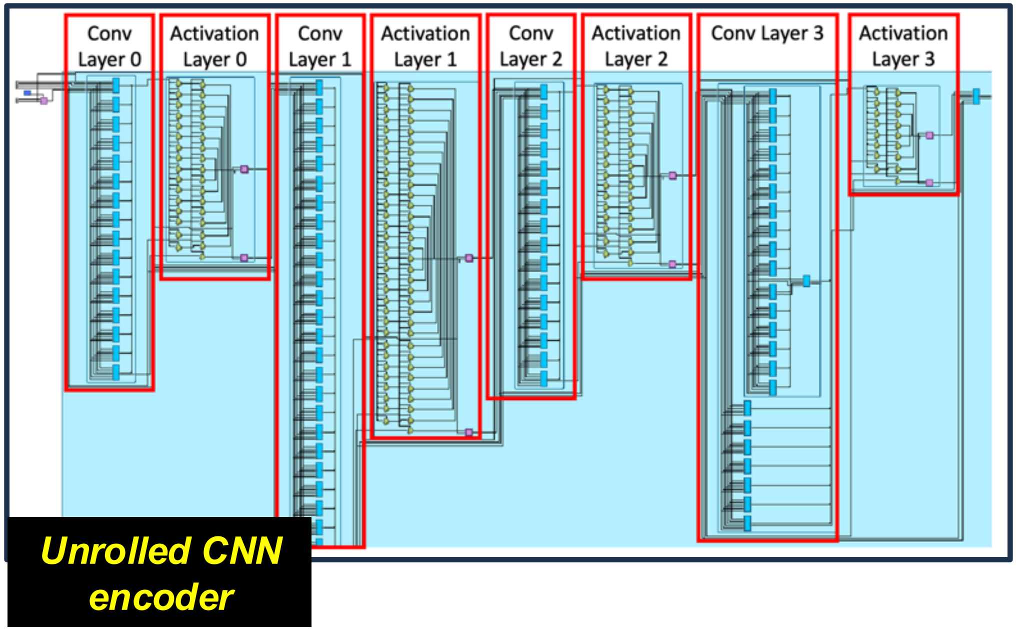
Example: Conv1D layer with 2 kernel of size 3



→ From the HDL lib one knows the latency/ressource budget of each PU

→ RTL generation

→ For each neuron type (*CNN, MLP, Activation,…*), HDL lib provides a basic RTL model. From there it's easy to create two RTL codes for the network.



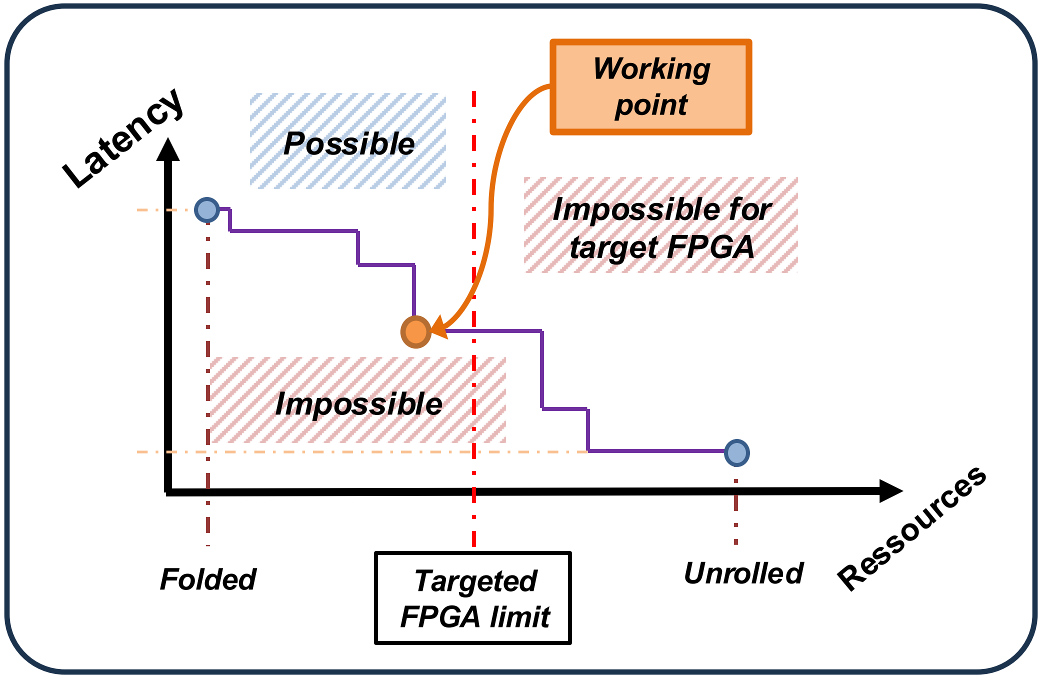
→ **Unrolled version** (*PU are used only once*). You get the best latency, but this is clearly suboptimal in terms of ressource usage.

→ **Folded version** (*only one PU used for all PU-related operations*). Best ressource usage, but poor latency.

→ One has to define an algorithm which, starting from one of those points and based on our latency/ressources constraints, finds an optimal working point

→ Optimizer principle

→ The optimization algorithm starts from one end.



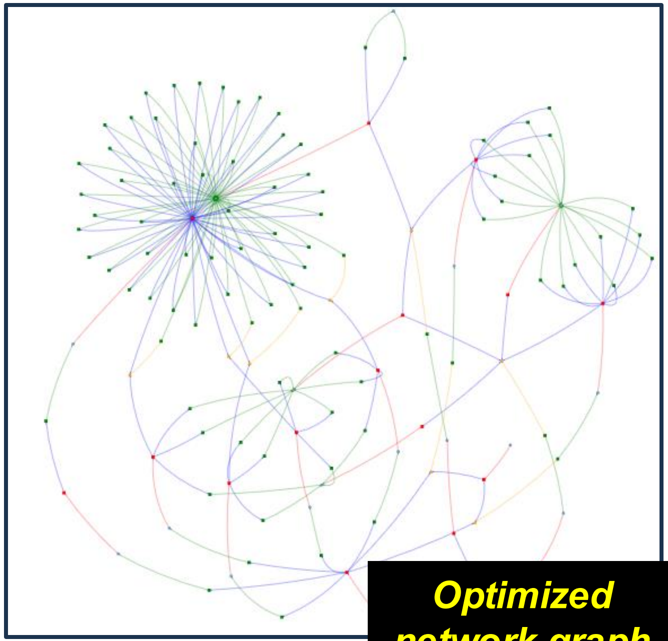
→ The best working point depends on latency/ressources constraints, and is found during the process.

→ The algorithm uses the latency/ressource parameters of each processing unit, there is no RTL involved here

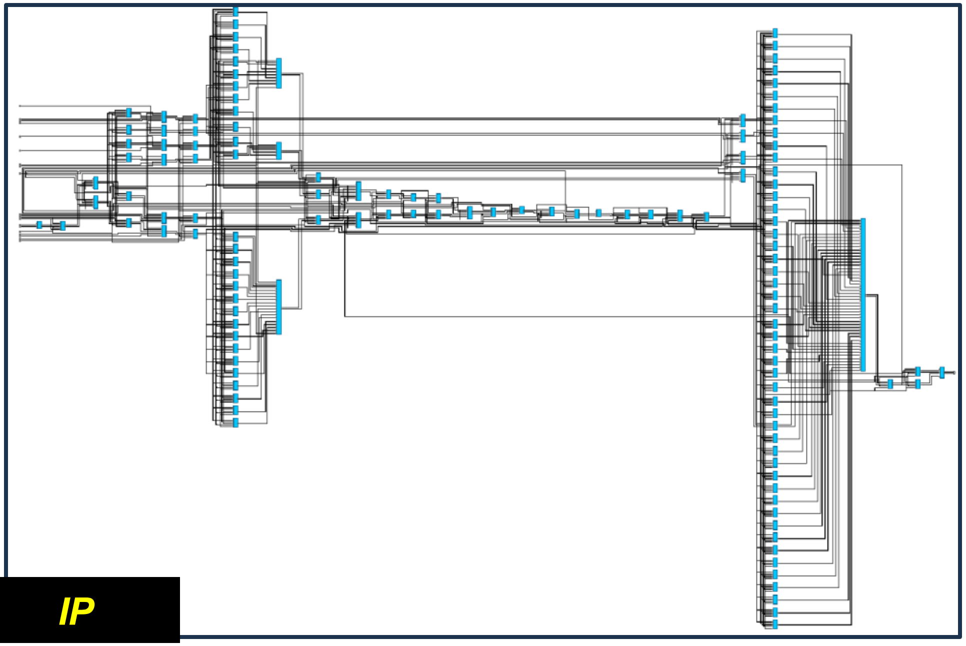
→ Folding/unfolding iteration consists in respectively removing/adding processing units to the design.

→ Increasing processing unit reusability (*during folding*) causes the insertion of **memory buffers** and **multiplexers.**

→ From optimized network to IP:



Optimized network graph

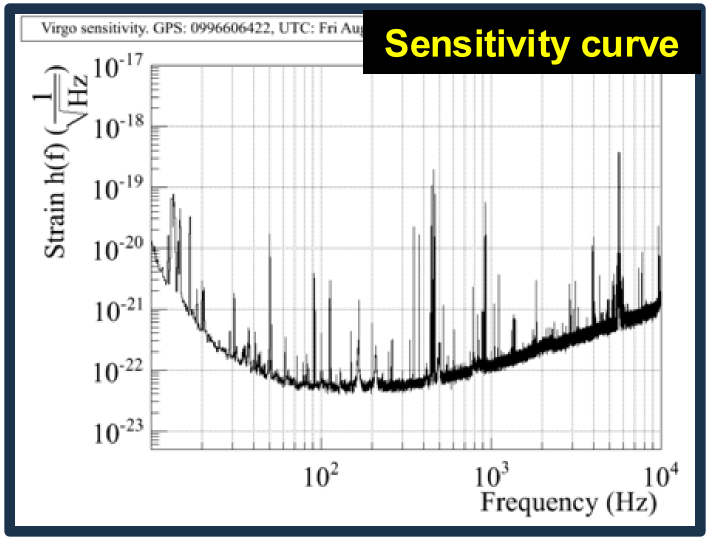


IP

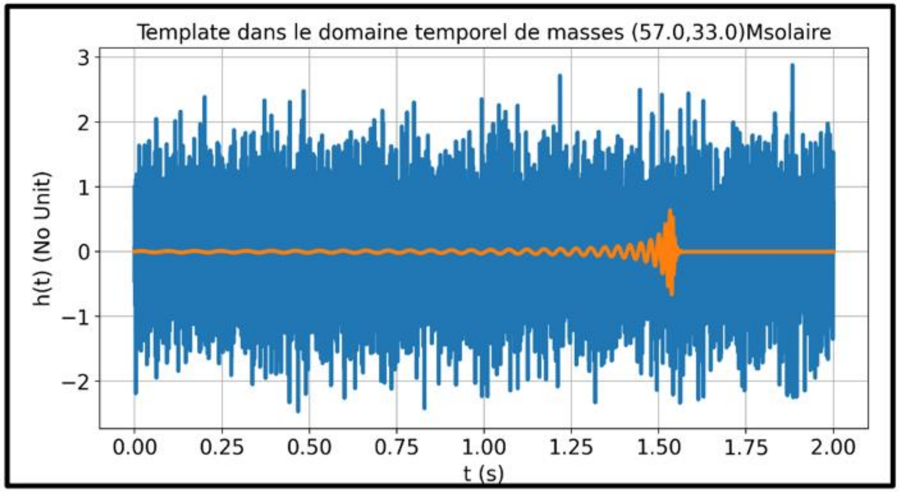
→ The best graph is synthetised and if FPGA constraints are fulfilled the network IP is created

→ The final scheduling (*given by the optimizer*) is handled by a **control unit** which is independent of the final network IP

→ Use case of GW:



→ GW interferometers have a very good sensitivity but there is still a lot of noise sources in the final data stream.



→ A proper noise id/removal currently requires dedicated and complex offline reprocessing. But **our detectors are triggerless**, so latency is not much of a problem a priori.

→ Except for multimessenger (**MM**) astronomy, where online detection is important

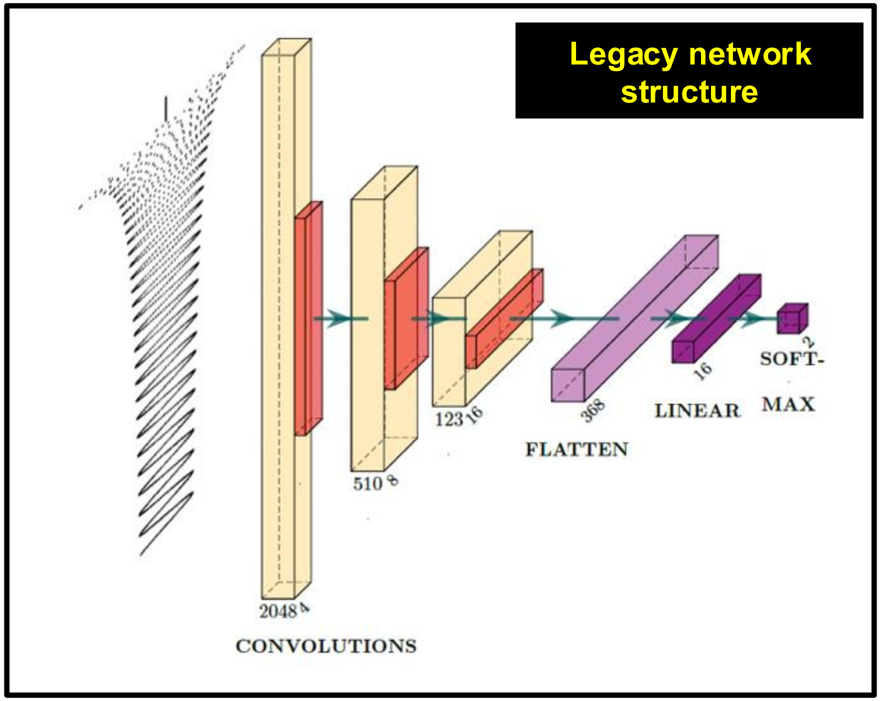
→ Current 'online' latency is **~O(20s)**.

→ Why machine learning?

→ The current detection process is based on signal processing techniques: **Fourier transforms, Q-transforms, match filtering,…**

→ **Some steps are particularly ML-friendly.** For example match filtering during which you compare the stream to a template bank. One can easily replace that by a simple network where the bank becomes the training sample. You can then replace the big comparison by a simple, much faster, inference, and put the CPU intensive part offline.

→ Network optimization



→ Started from a legacy network (1701.00008v3).

→ Very simple CNN encoder take 1s of the data strain as input (2048 points), a provides 2 values at the output:

- Compatibility of the data with signal
- Compatibility of the data with noise

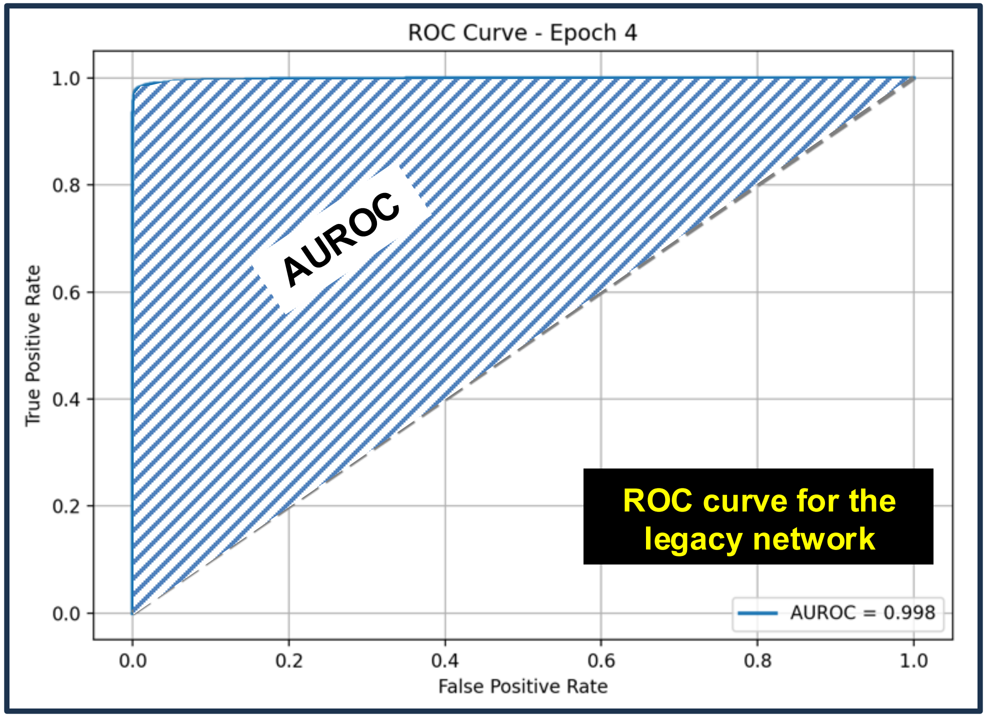
→ There are much more elaborated architectures today, but this one is simple, efficient, and particularly attractive for FPGA implementation

Number of parameters: 37094

Number of operations for 1 inference:
-> N additions : 1187600
-> N multiplications: 1209184

→ Make the network FPGA friendly:

→ Remove some **intensive steps**: batchnorm (*don't needed because data is whitened upstream*), use relu activations, remove the last softmax activation step (*signal cat over threshold is sufficient*)



→ Then try to adjust hyperparameters in order to maximise of the surface below ROC curve (**AUROC**), and minimise of the number of operations and parameters (*relatively easy with a simple network*).

→ Can be made less heuristic with bayesian optimisers with those params added to the cost function. **We made some first conclusive tests with this network.**

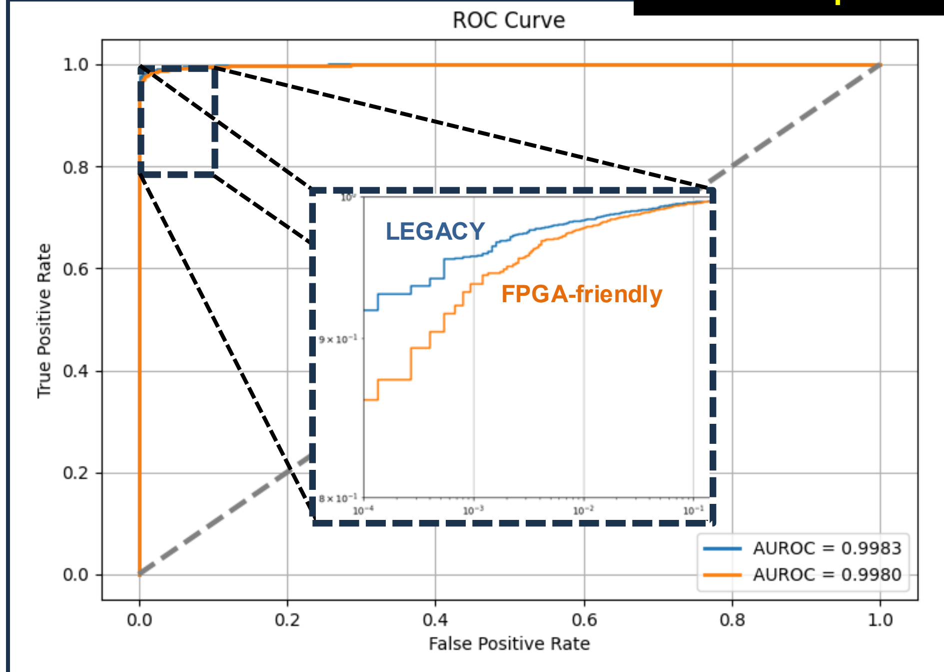
→ FPGA-friendly encoder, structure and perf:

→ Simple network, easy to get first interesting results:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 2048, 1)]         0

 batch_normalization (BatchN  (None, 2048, 1)          4
 ormalization)

 conv1d (Conv1D)             (None, 2041, 4)           36

 max_pooling1d (MaxPooling1D  (None, 510, 4)           0
 )

 activation (Activation)     (None, 510, 4)            0

 conv1d_1 (Conv1D)           (None, 509, 8)            72

 max_pooling1d_1 (MaxPooling  (None, 127, 8)           0
 1D)

 activation_1 (Activation)   (None, 127, 8)            0

 conv1d_2 (Conv1D)           (None, 126, 16)           272

 max_pooling1d_2 (MaxPooling  (None, 31, 16)           0
 1D)

 activation_2 (Activation)   (None, 31, 16)            0

 conv1d_3 (Conv1D)           (None, 24, 8)             1032

 max_pooling1d_3 (MaxPooling  (None, 6, 8)             0
 1D)

 activation_3 (Activation)   (None, 6, 8)              0

 flatten (Flatten)           (None, 48)                0

 dense (Dense)               (None, 2)                 98

=================================================================
Total params: 1,514
Trainable params: 1,512
Non-trainable params: 2
```

Network structure

ROCs comparison



Number of parameters: 1514 (37094)

Number of operations for 1 inference:
-> N additions : 154768 (1187600)
-> N multiplications: 154816 (1209184)

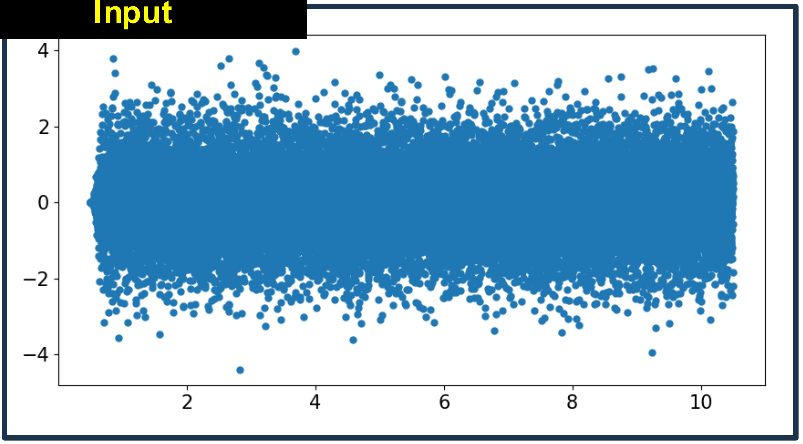
→ The performance loss is relatively negligible w.r.t. the **massive network size reduction**. This is a quick first look, there is room to improvement here.

→ **On the FPGA side this is clearly not the same story**

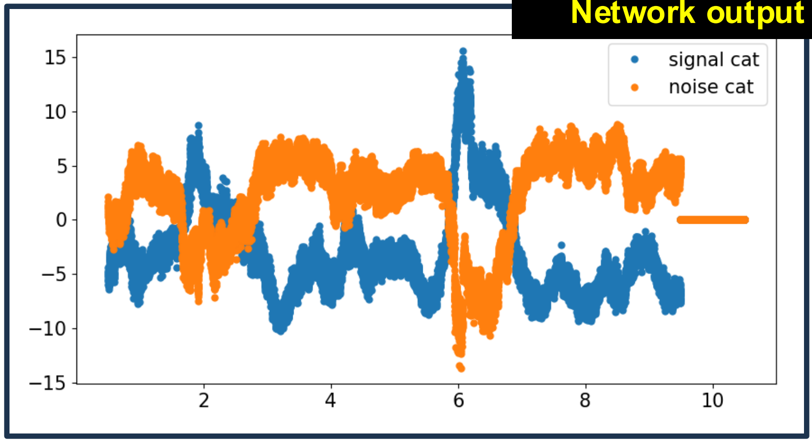
→ RTL implementation and simulation

→ 10 seconds of simulated data, compare the software network output (Tensorflow), with RTL simulation:

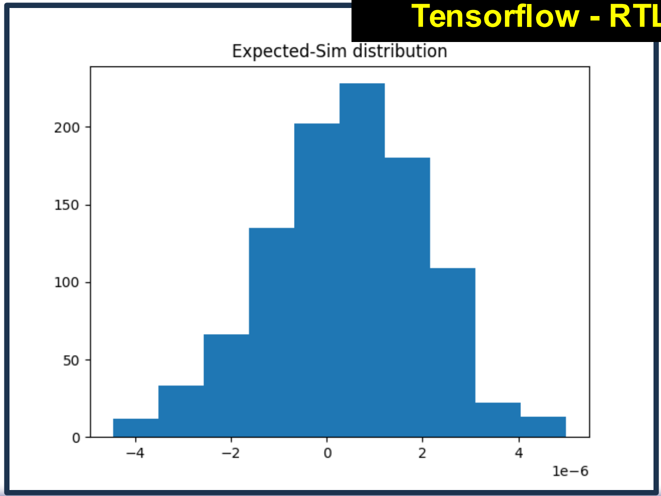
Input



Network output



Tensorflow - RTL



→ The RTL network is built with HDL lib processing units

→ **Output comparison validate the HDL lib.** This is a first important step.

→ Now start to test optimisation stage

→ **Conclusion**

→ Developed a procedure to implement neural network inference in VHDL on FPGA platform

→ **Take home message: a lot can/should be done to optimize the network upstream.** Bayesian optimization is an important ally there, but this will work only if you understand what you're doing (*eg to choose the right network*)

→ Once done, you can go to porting. For this part we started to develop a **low level modular HDL lib** which contains an RTL model of each neuron flavor. **First elements have been developed and successfully tested.**

→ Based on those bricks, a network graph is build and an optimizer finds an architecture based on customer requirements in terms of latency and FPGA ressources. A first basic version has been developed, still lot of improvements possible, work in progress

→ **Optimizer and HDL lib are independent**. You don't need to modify the optimizer when you upgrade the HDL lib with a new element.

→ **Next steps**

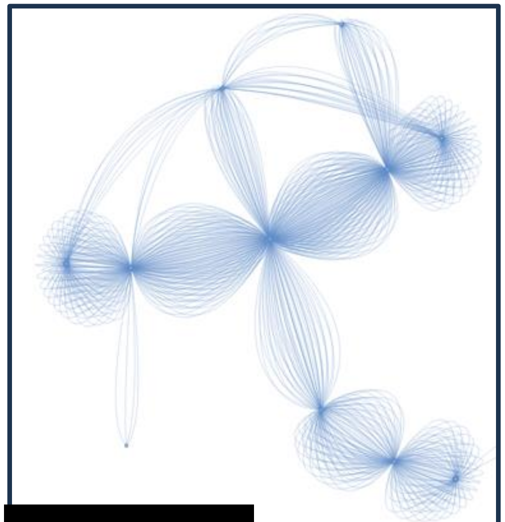
→ Add more complex cells to the HDL lib: currently looking at RNN cells and attention layers.

→ Continue to develop and improve the graph optimizer

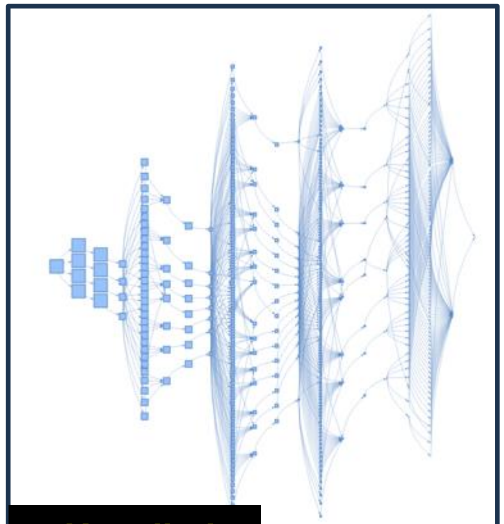
→ Start to port simple architectures (*like the GW encoder*) on low level FPGAs (*eg CycloneV*)

→ STEP 1: optimizer working principle

1. Construct the fully folded network graph: each node is a processing unit

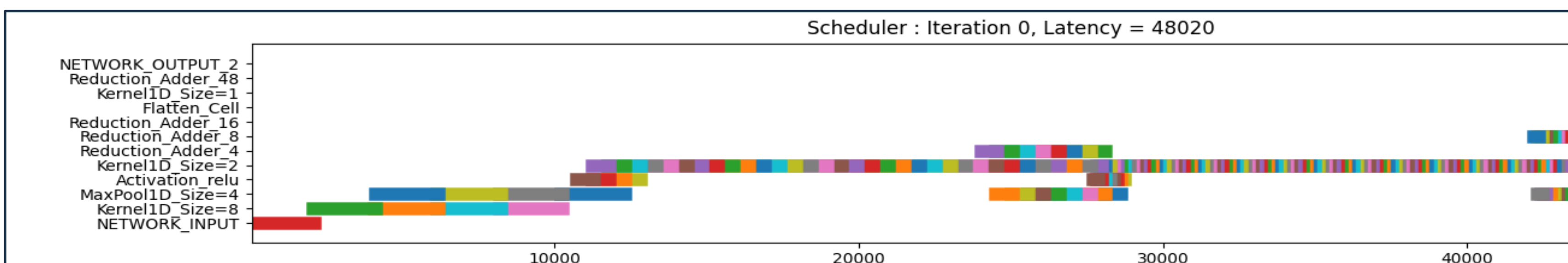


Folded



Unrolled

2. Estimate the total latency

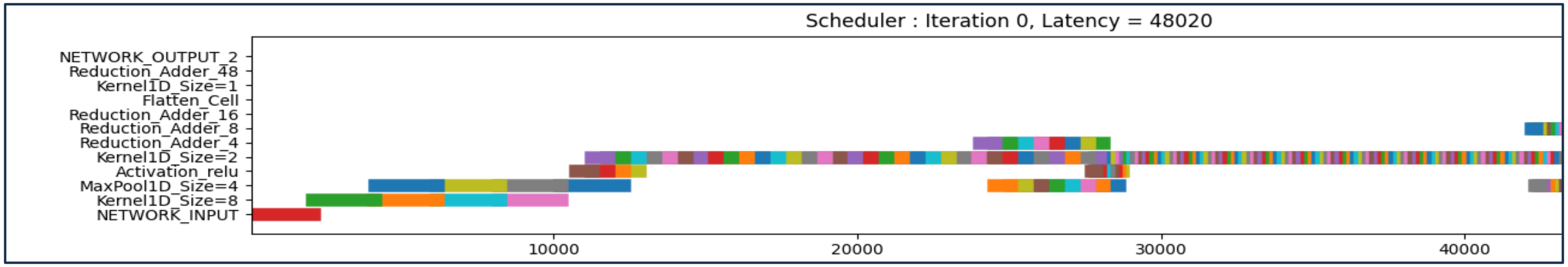


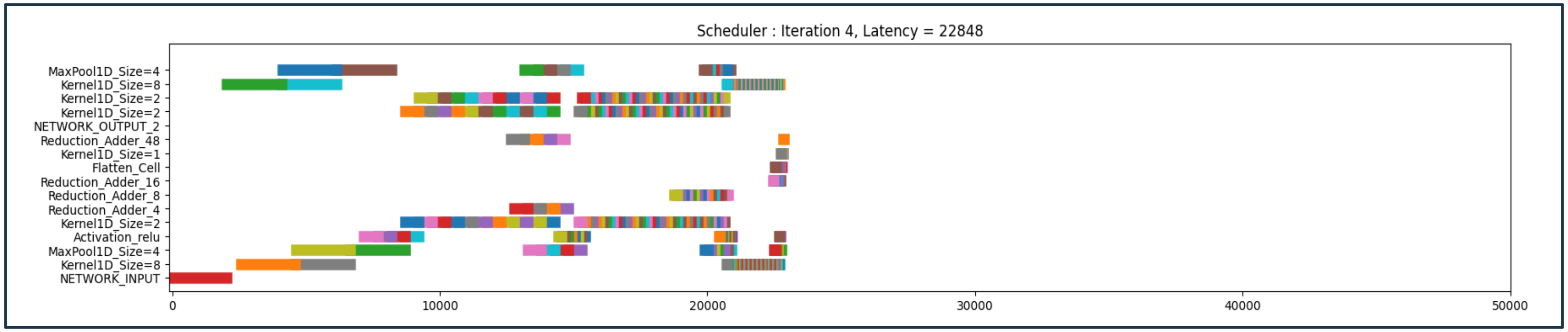
→ STEP 1: adding paralelization:

3. At each iteration, add some blocks (↗ ressources) to paralellize processing and reduce latency

4. Add memory blocks to handle data and network parameters

5. Add multiplexers to further simplify graph

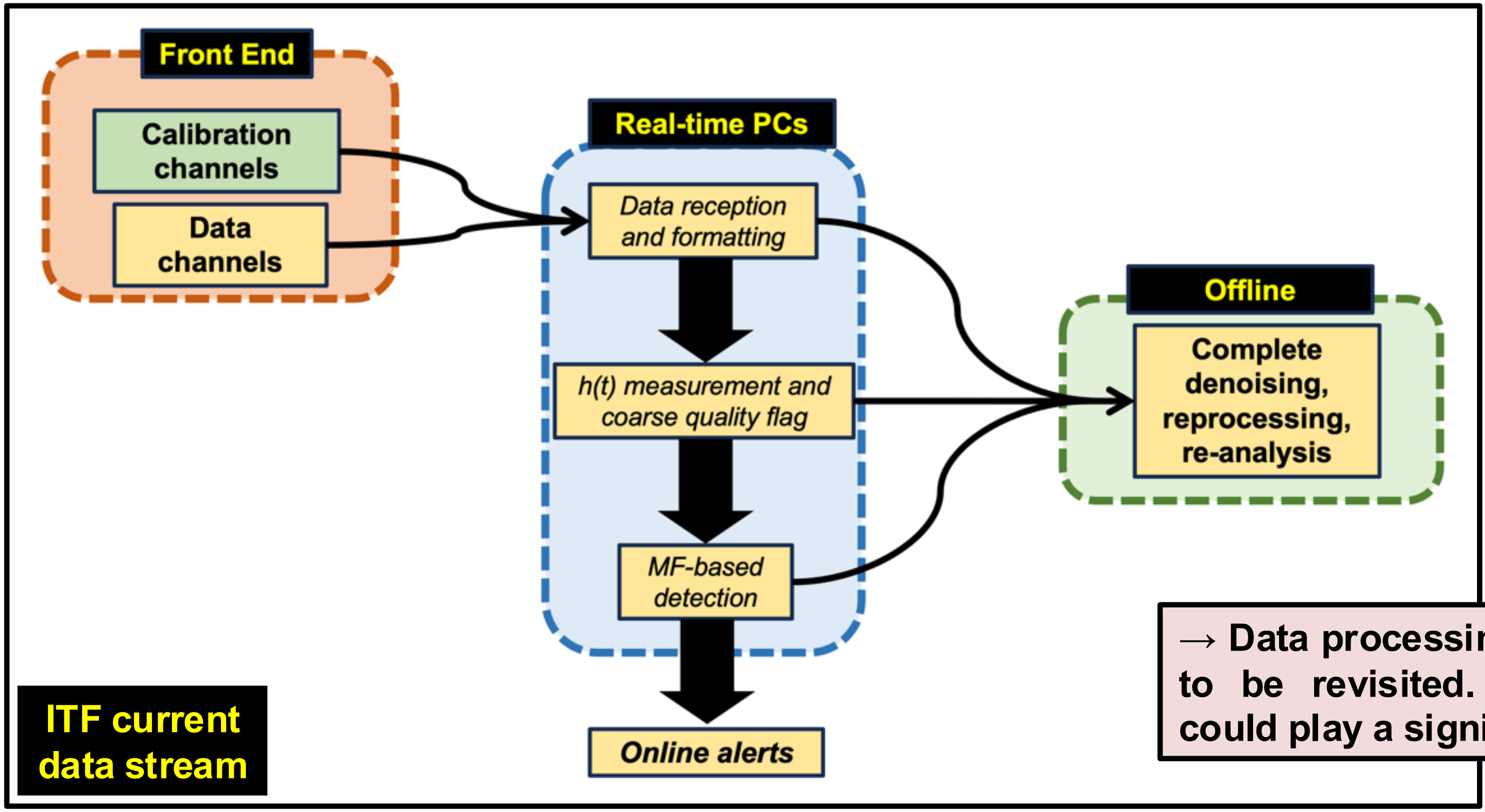




→ Context:

→ The next generation of GW ITFs will have 10x more sensitivity ⇒ 1000 times more events

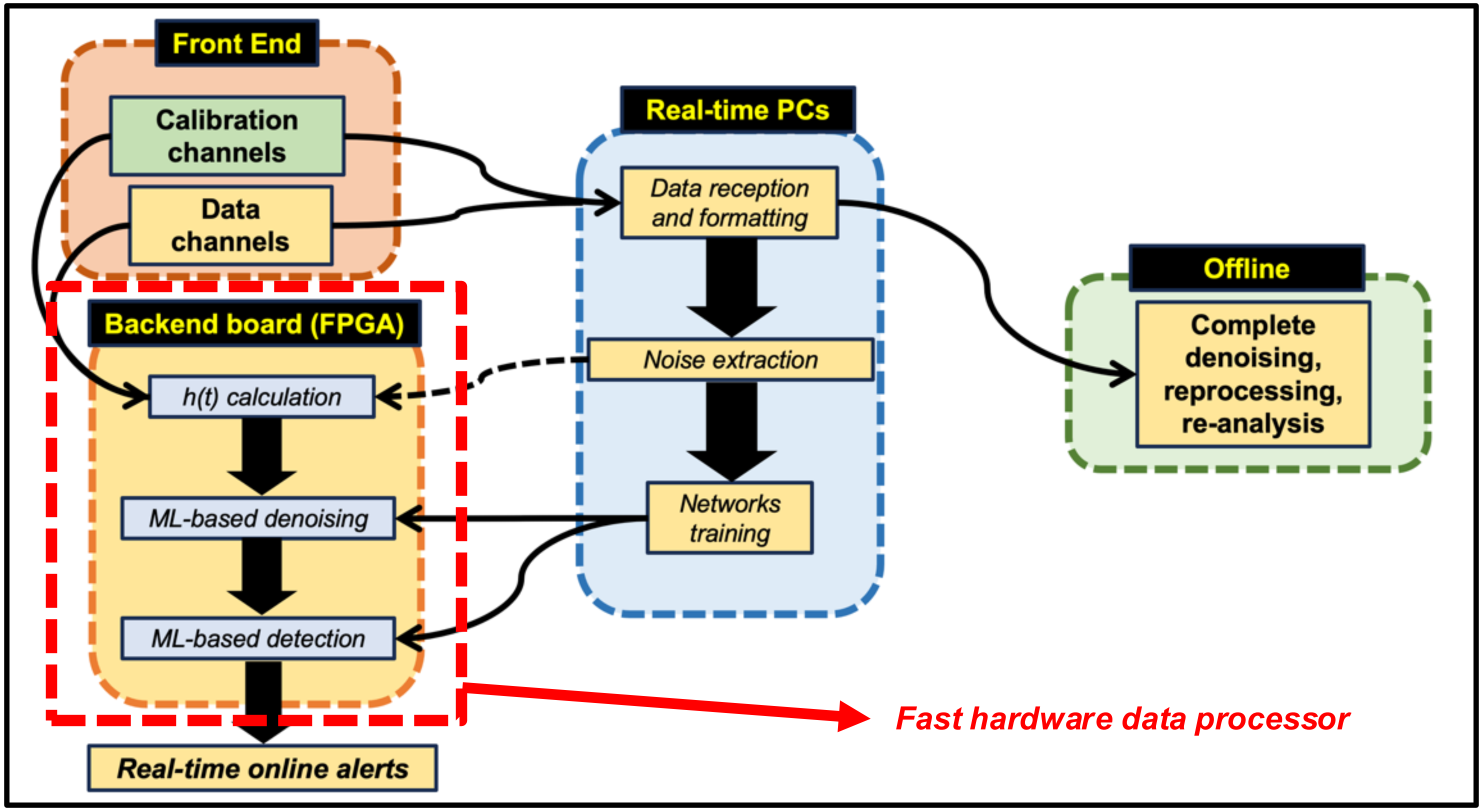
→ You will start to experience pileup 😀… You will see many more BNS events with EM counterpart (*MM golden events*). You will need a fast and efficient online detection, **a kind of proto GW trigger.**



→ Data processing stream will have to be revisited. Machine learning could play a significant role.

→ Why going hardware?

→ A fully ML-based DAQ for the next generation of detectors could look like that:



→ Denoising step (*hardware unfriendly*) can be skip in first approach.