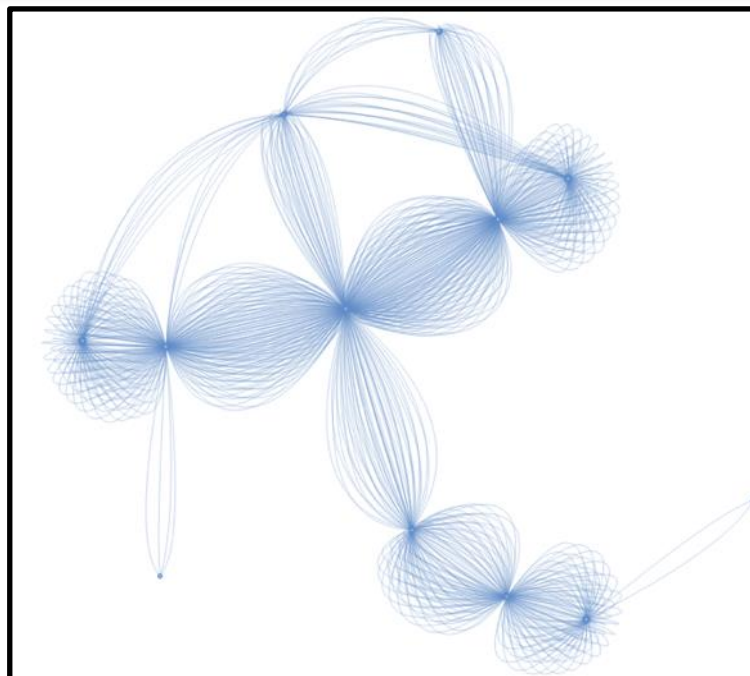




Q.David, G.Galbit, S.Viret

[IP2I Lyon](#)

s.viret@ip2i.in2p3.fr



Institut de Ciències del Cosmos
UNIVERSITAT DE BARCELONA



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH



2026

Topical Workshop
on Electronics
for Particle Physics

Hotel Rey Don Jaime · Castelldefels, Barcelona, Spain · **SEPTEMBER 28 – OCTOBER 2**

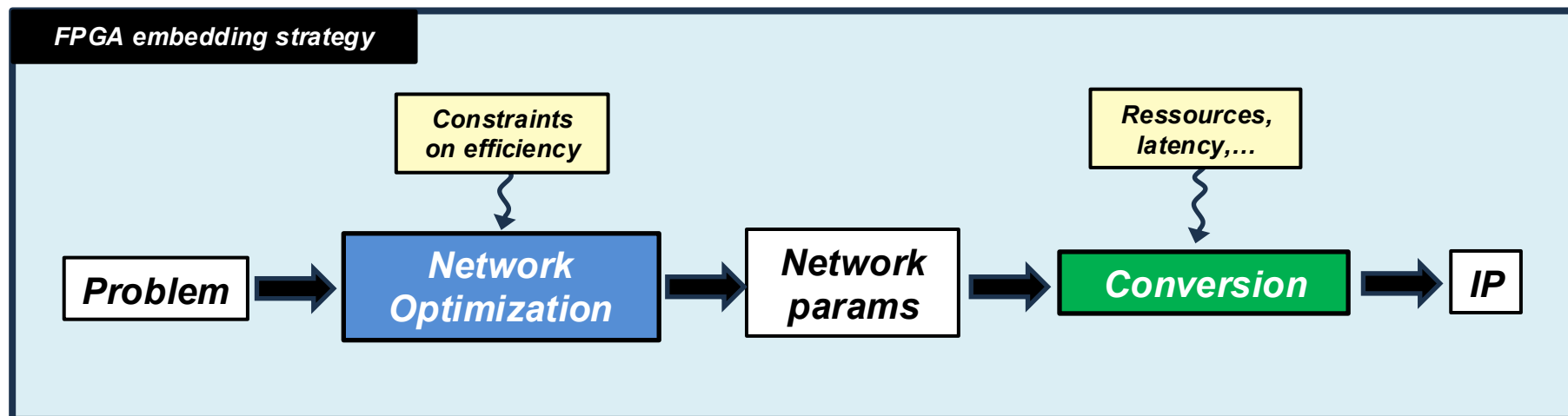
⇒ Porting ML on FPGAs

⇒ The pyml2fpga tool

⇒ Examples

- IA development exploded thanks to GPUs and were quickly ported in HEP.
- DAQ/Trigger constraints (high bandwidth, low latency) $\Rightarrow$ FPGA
- Lots of efforts in the past years (eg [HLS4ML](#)). Already used by some LHC experiments trigger scheme (see for example <https://arxiv.org/html/2503.09428v1>)

→ FPGA ML embedding as a 2-step process: **network optimization** and **conversion**



→ **Optimization** (not detailed in this talk) is actually also relevant if you use a GPU. **Compact networks are much easier to understand and control.** Quantization and pruning are widely used here, but simplification of the network complexity, for example with bayesian optimization, is also very important.

→ **Conversion** can start only after a proper network optimization. Here your constraints will drive the tool to use.

⇒ Network conversion is driven by:

- **Latency:** time available between data reception and output emission
- **Bandwidth:** I/O data rate to sustain
- **Target:** the FPGA where the network will be implemented

⇒ For most trigger applications, latency and bandwidth are driving requirements. Target is usually a high-end FPGA. HLS-based tools are very powerful in this corner

⇒ For front-end applications, **target becomes a constraint**, as power budget matters here. Frugal approach, with pure VHDL, is preferable here.

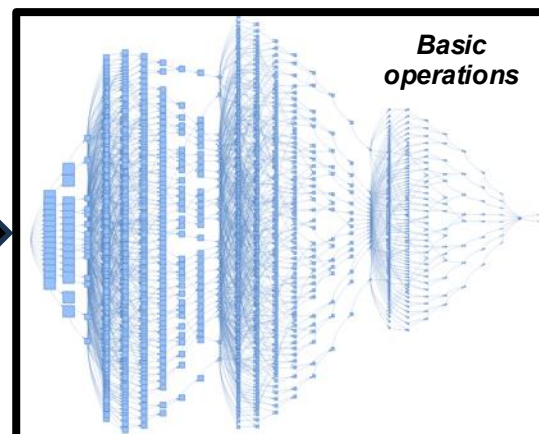
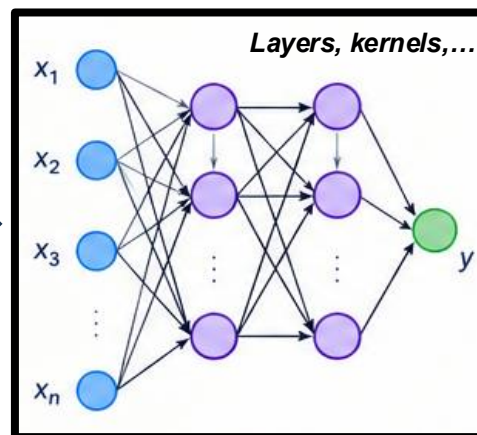
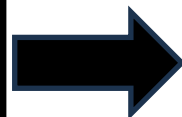
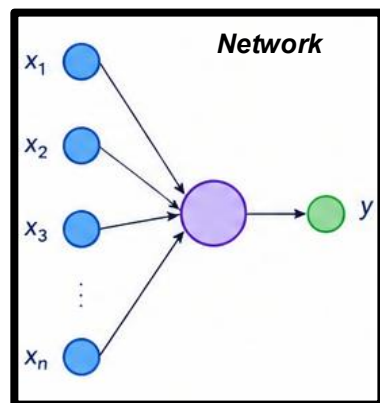
⇒ This is the problematic we tried to address with [pym12fpga](#), which provides:

- **Native, backend independent, VHDL code.**
- **An hardware network implementation adaptable to target ressources/latency**

⇒ We focus on experiments which needs to run small inferences (*~10k params / ~1M operations*) within a latency of few μ s/ms, on small/mid-size FPGAs (*cheap/low power FE boards*)

⇒ Model not converted into pseudo-code, but into a **directed acyclic graph**

Graph creation workflow



COARSER

Node granularity

FINER

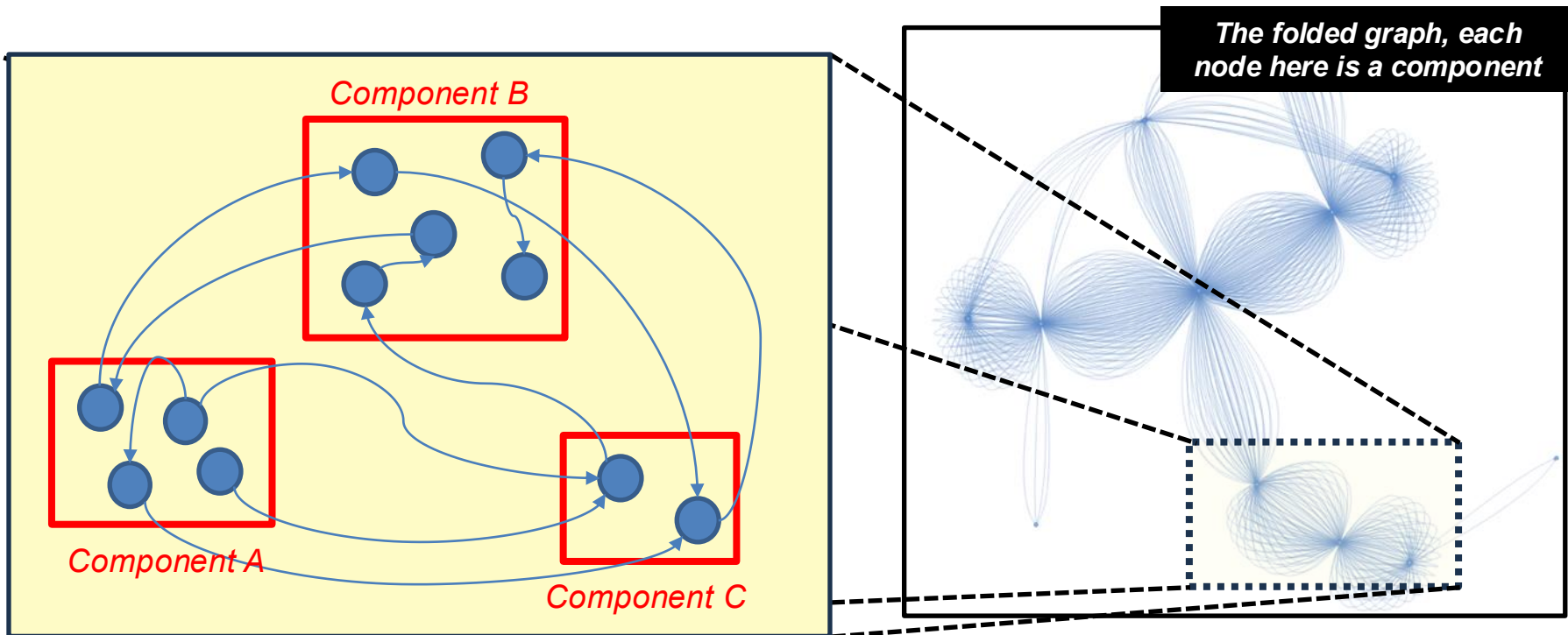
⇒ **Nodes granularity** (*neuron, layer, ...*) is user-defined and frozen for the rest of the procedure. **A node is defined by a fixed set of operations**

⇒ If **resource constrained** ⇒ finer granularity in order to get the best possible reuse factor

⇒ If **latency constrained** ⇒ coarser granularity as you will try to minimize the reuse factor

⇒ In both case, you will need optimize the graph before going to the implementation

⇒ **Optimizing the graph:** the folding



⇒ Similar nodes (*you have fixed it now*) are linked into a **dictionary**:

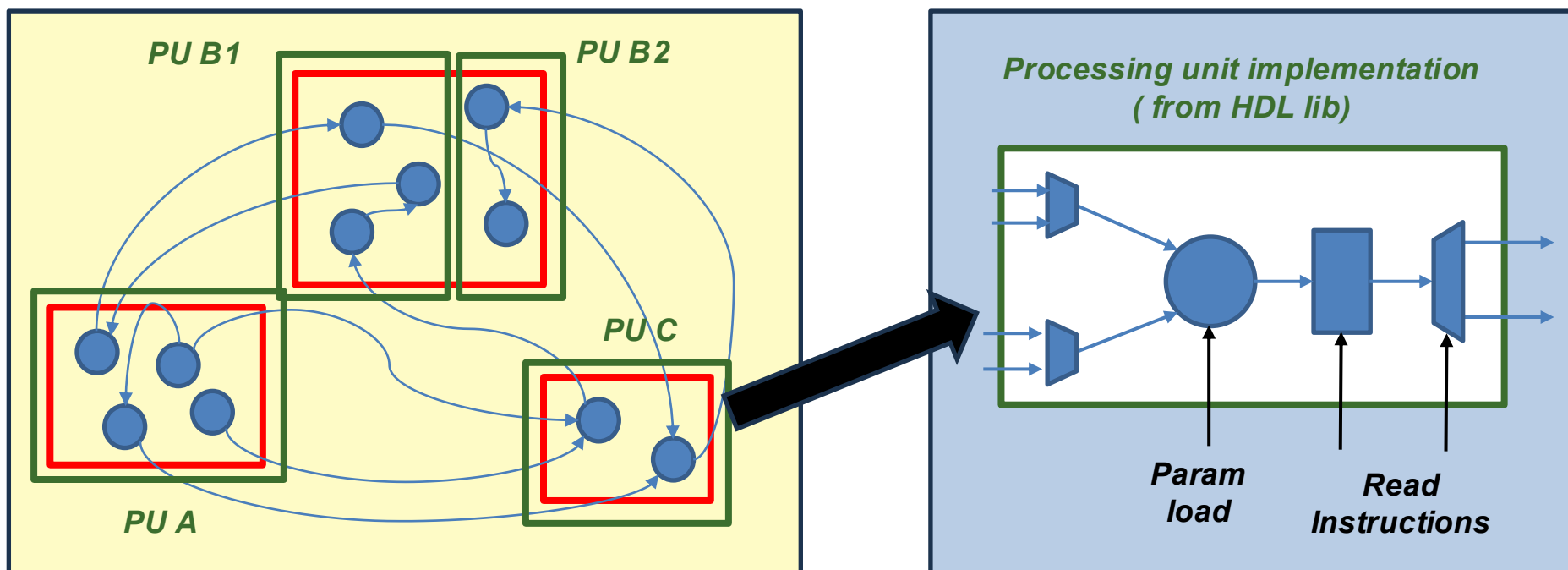
```
dict={Component A : (node 1, node 5, node 3); Component B : (node 2, node 6);...}
```

⇒ Component-based version of graph built (**folded graph**). Minimal resources configuration needed for the inference. **Maximum reuse factor for each component.**

→ **Optimizing the graph:** the HDL library

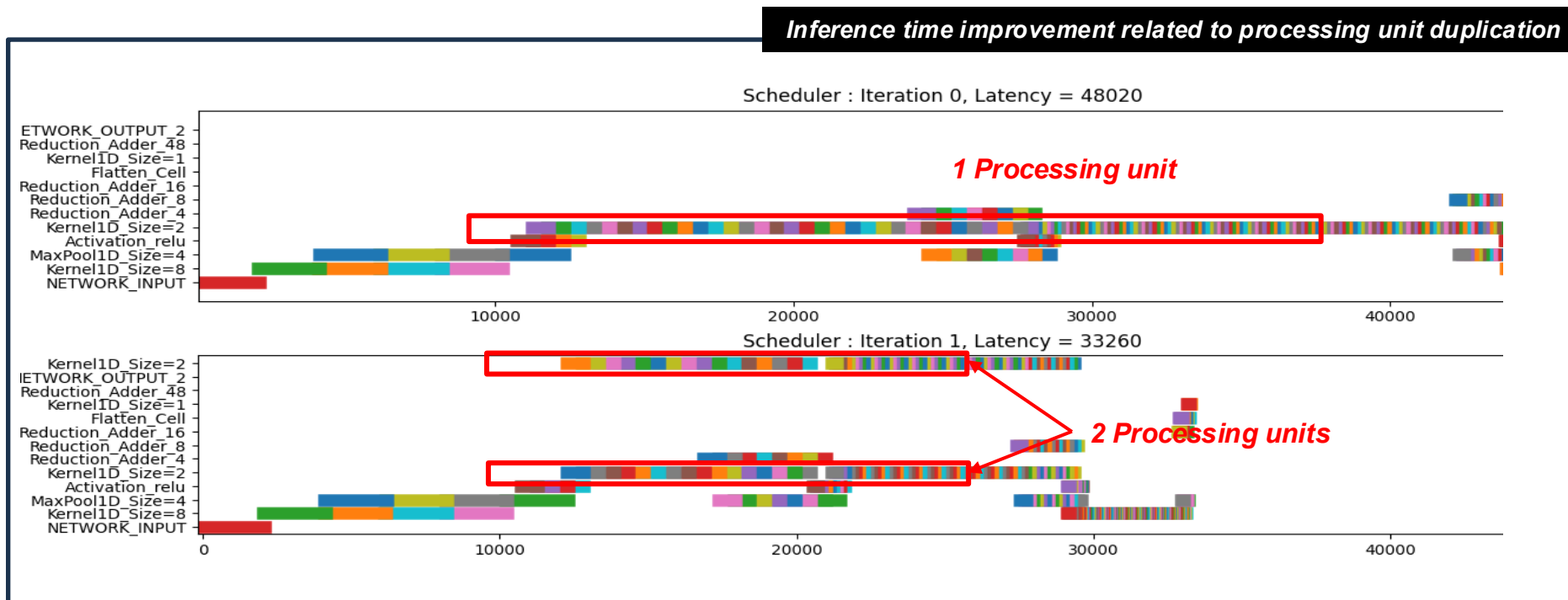
⇒ Each component is dealt with with one or more '**processing units**'. PU is pure VHDL code based on an **HDL lib** containing node description. **A given PU can be used by different layers (ressource optimization).**

⇒ **The graph can now be expressed in terms of FPGA components.** One can optimize the number of PU per components via **temporal scheduling**



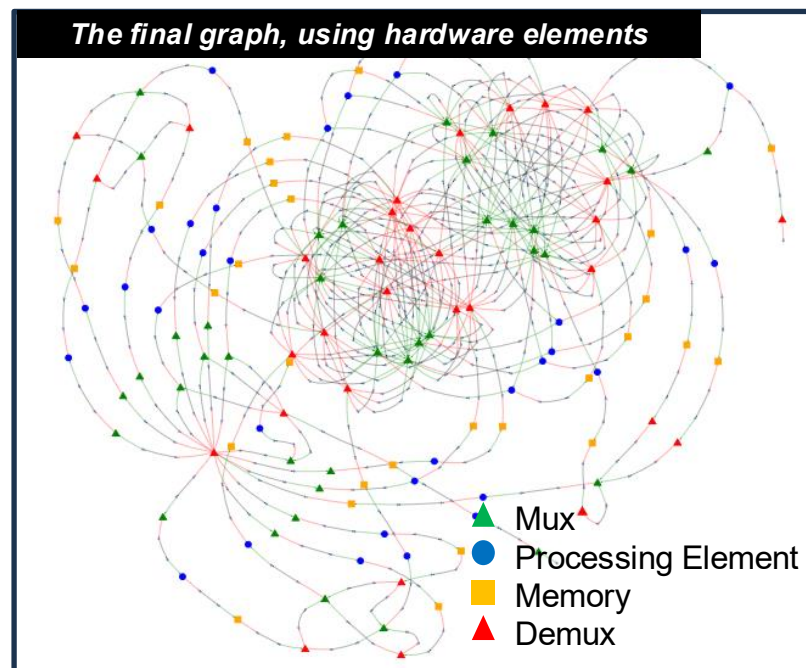
→ **Optimizing the graph:** temporal scheduling

⇒ Crucial stage to drive the component split strategy and identify bottlenecks. Done with a simple 'greedy' algorithm for the moment. Good results for simple networks but can be improved.

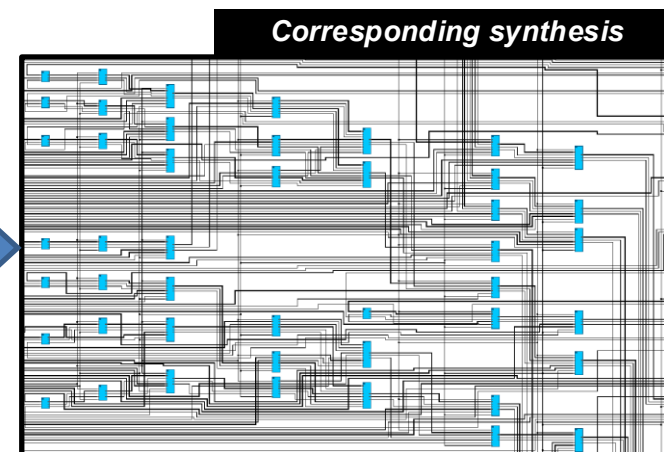


⇒ This step provides a sweetspot accounting for latency/ressource constraints of the user

→ From the optimal graph to the IP



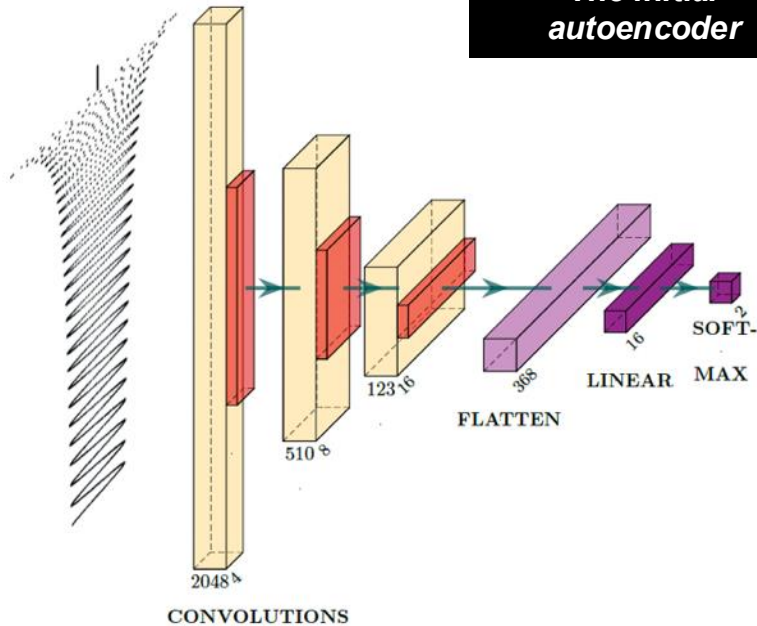
RTL code
generation



⇒ The final graph can be synthesized easily to an IP. **This is plain VHDL, no backend dependency.**

⇒ Dedicated tools to integrate the IP in a firmware (*in order to properly run the inference*) have also been developed.

The initial autoencoder



→ Legacy gravitational wave detection network ([1701.00008v3](#)). **Autoencoder (CNN+Dense)**.

→ Take 1s of input data (@2kHz), and provides 2 values at the output (p_{signal} , p_{noise})

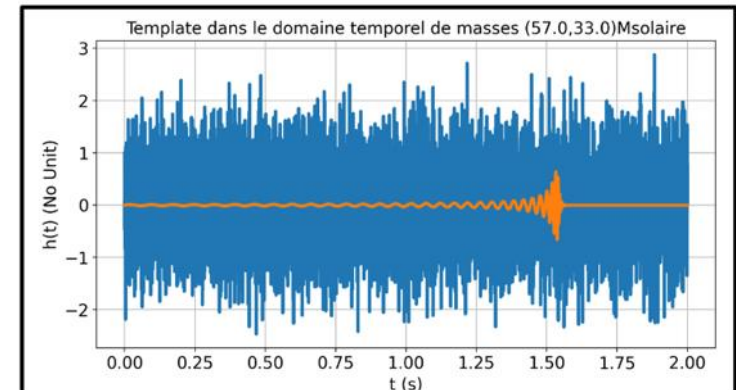
→ Already light, but still not negligible size for components like an FPGA

Number of parameters: 37094

Number of operations for 1 inference:

→ N additions : 1187600

→ N multiplications: 1209184



→ Optimisation:

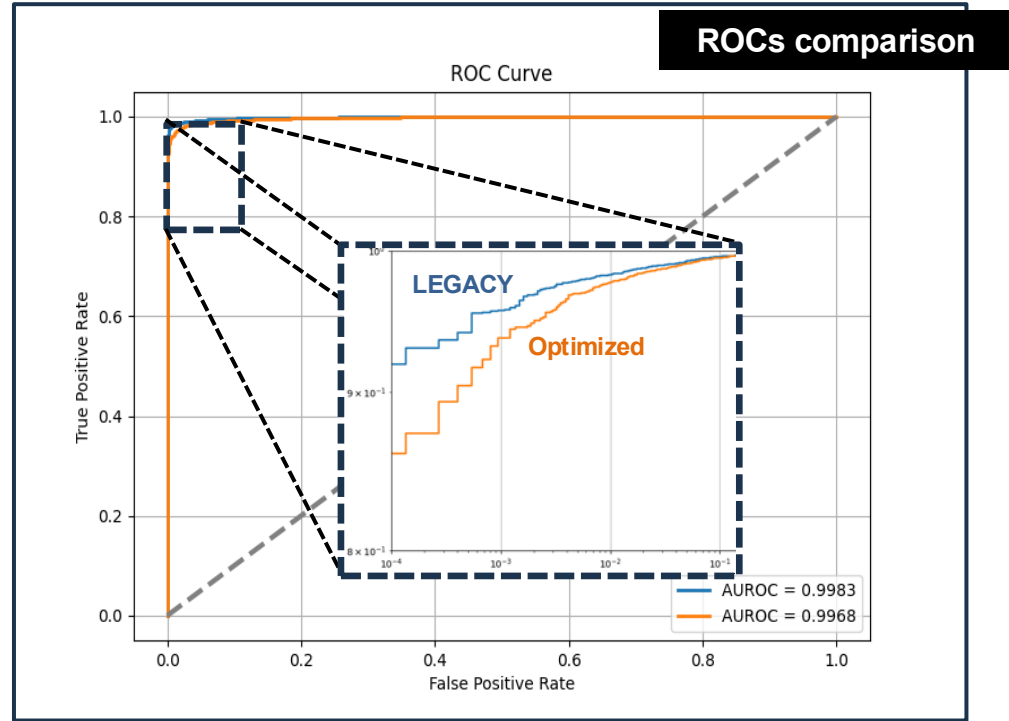
Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 2048, 1)	0
conv1d (Conv1D)	(None, 2041, 4)	36
max_pooling1d (MaxPooling1D)	(None, 510, 4)	0
activation (Activation)	(None, 510, 4)	0
conv1d_1 (Conv1D)	(None, 509, 8)	72
max_pooling1d_1 (MaxPooling1D)	(None, 127, 8)	0
activation_1 (Activation)	(None, 127, 8)	0
conv1d_2 (Conv1D)	(None, 126, 16)	272
max_pooling1d_2 (MaxPooling1D)	(None, 31, 16)	0
activation_2 (Activation)	(None, 31, 16)	0
conv1d_3 (Conv1D)	(None, 24, 8)	1,032
max_pooling1d_3 (MaxPooling1D)	(None, 6, 8)	0
activation_3 (Activation)	(None, 6, 8)	0
flatten (Flatten)	(None, 48)	0
dense (Dense)	(None, 2)	98

Total params: 1,510 (5.90 KB)
Trainable params: 1,510 (5.90 KB)
Non-trainable params: 0 (0.00 B)

Network structure

Number of parameters: 1510 (**37094**)

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

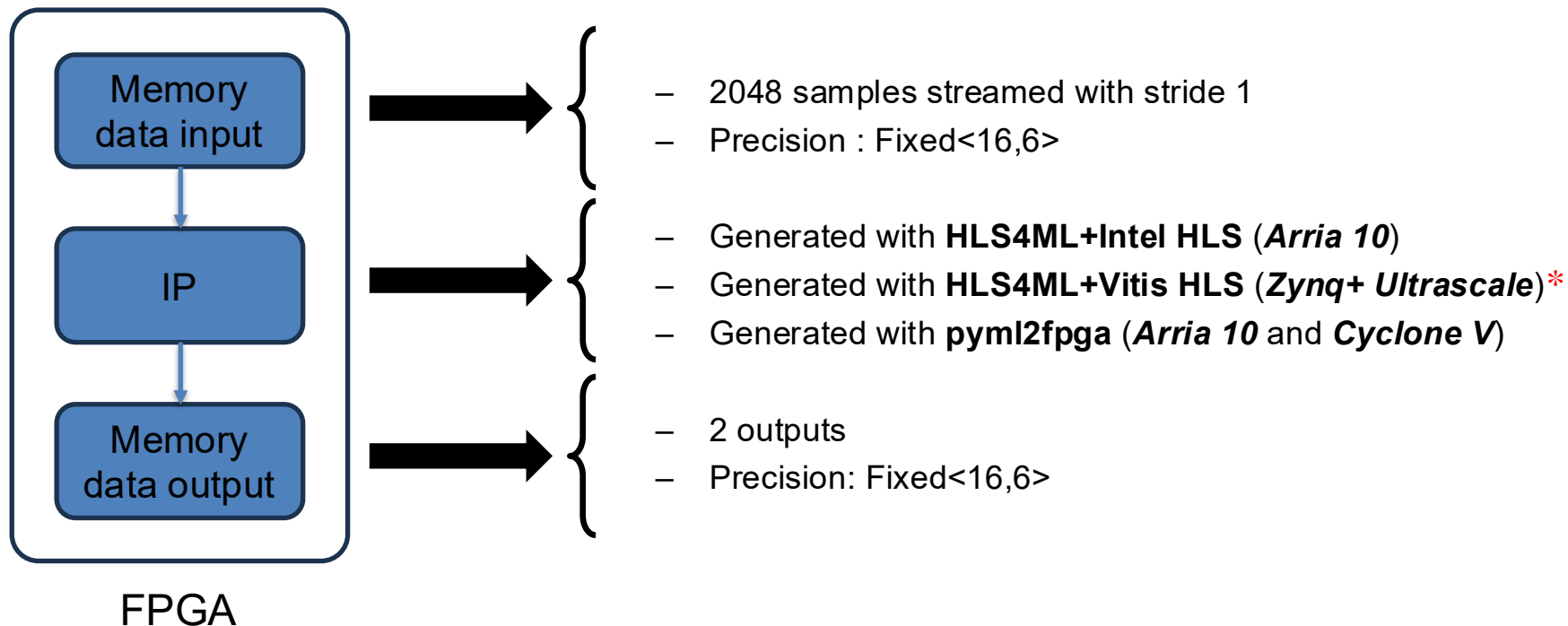


→ Before pruning and quantisation a lot can be done.

→ Optimisation could be a game-changer for FPGA-embedding.

→ Tests

- Optimized network IP obtained with **pyml2fpga** was ported to different targets. The node granularity level chosen for the graph is the neuron.
- In parallel we ran the **HLS4ML** workflow on the same network



* Work done by **G.Maraninchi** (CENBG/IN2P3)

→ Results

	HLS4ML				pyml2fpga	
	Intel HLS / Arria 10		Vitis HLS (Xilinx) / Zynq+ Ultrascale		Arria 10	Cyclone 5
Reuse Factor	1	4	2 (Latency)	32 (Latency)	-	-
Latency (Ck cycles)	71k	116k	32k	88k	17k	
Logic (ALMs)	65k (15%)	51k (12%)	67k (29%)	69k (30%)	9k (2%)	7k (38%)
Memory	2,5Mb (4%)	6,5Mb (12%)	5,0Mb (43%)	5,0Mb (43%)	5Mb (9%)	3Mb (85%)
DSP	231 (15%)	350 (23%)	736 (43%)	46 (3%)	44 (3%)	44 (66%)

→ **Latency definition:** time necessary to process a complete inference, from first data IN to result OUT

→ **pyml2fpga** provides slightly better performance in terms of latency, DSP, and logic modules usage. In a sense it is expected as HLS backend could add some overhead. The memory can be easily externalized, thus reducing the load on the FPGA.

→ **As there is no backend dependency, producing the IP does not rely on the type of FPGA you plan to use.**

- Embedding machine learning in FPGA requires dedicated optimization and conversion techniques.
- Most conversion tools are based on HLS backends which are not optimal for resource-limited components.
- We described a new conversion algorithm, **pym12fpga**, which combines a graph representation of the network to a versatile HDL library.
- A first test on a small network showed that this approach can achieve a very compact implementation while maintaining low latency.
- Try it yourself (*code still in development, but examples available*):

<https://gitlab.in2p3.fr/think2/pym12fpga/-/blob/master/README.md>



Part of this work (*pym12fpga*) is part of the IN2P3 THINK R&T initiative: <https://think.in2p3.fr/>

→ Integration of the IP in a project

