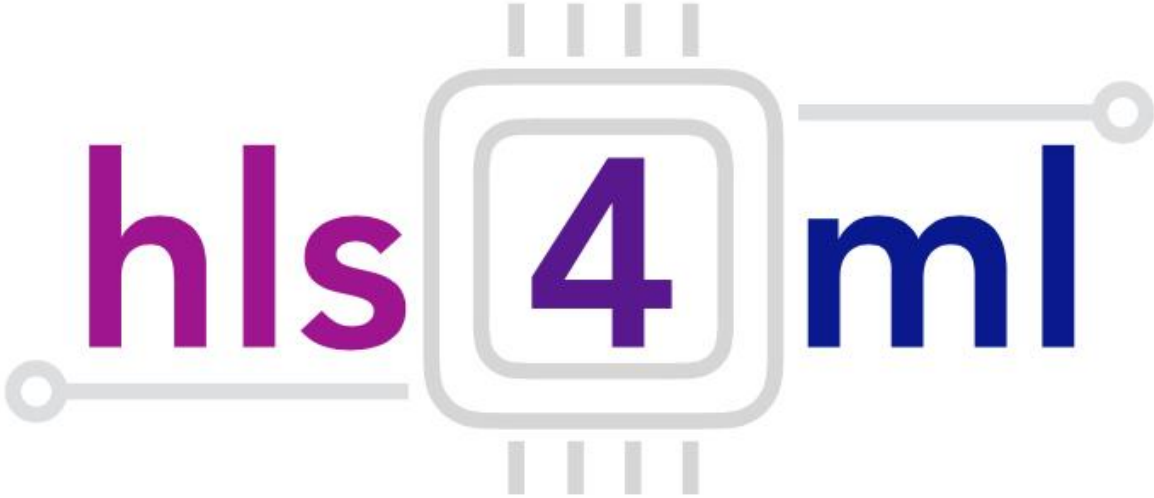
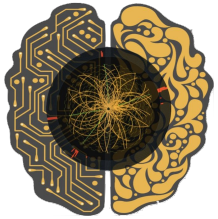


Overview of **hls4ml** project

Vladimir Lončar
For the FastML team
fastmachinelearning.org

CERN

mPP



The Large Hadron Collider

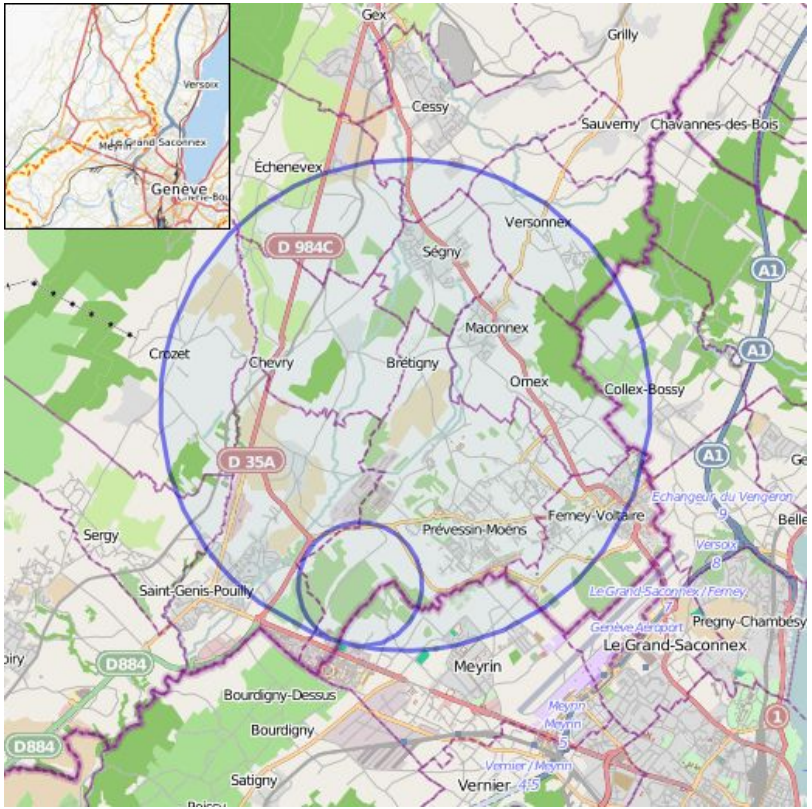
27 km circumference accelerator at CERN on the border of France and Switzerland near Geneva

Accelerates protons close to the speed of light, and collides them at 14 TeV centre of mass energy

Searching for new fundamental physics of the universe!

Collisions happen at 4 points where there are detectors

- We work on one of these: the **CMS** experiment

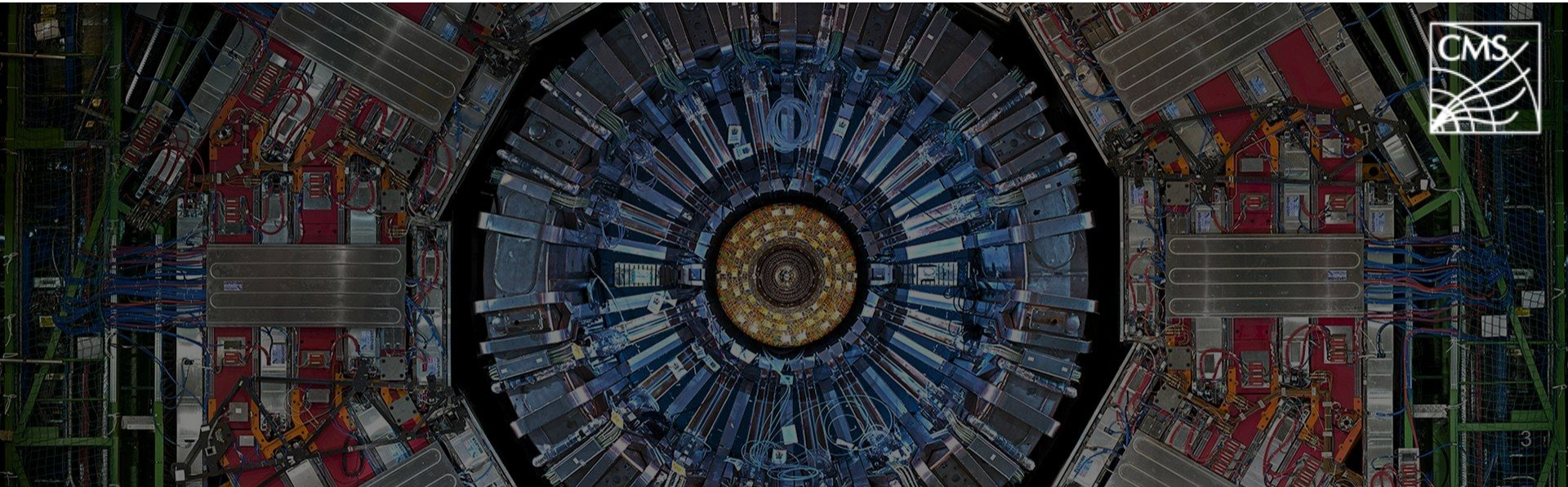


Challenges in LHC

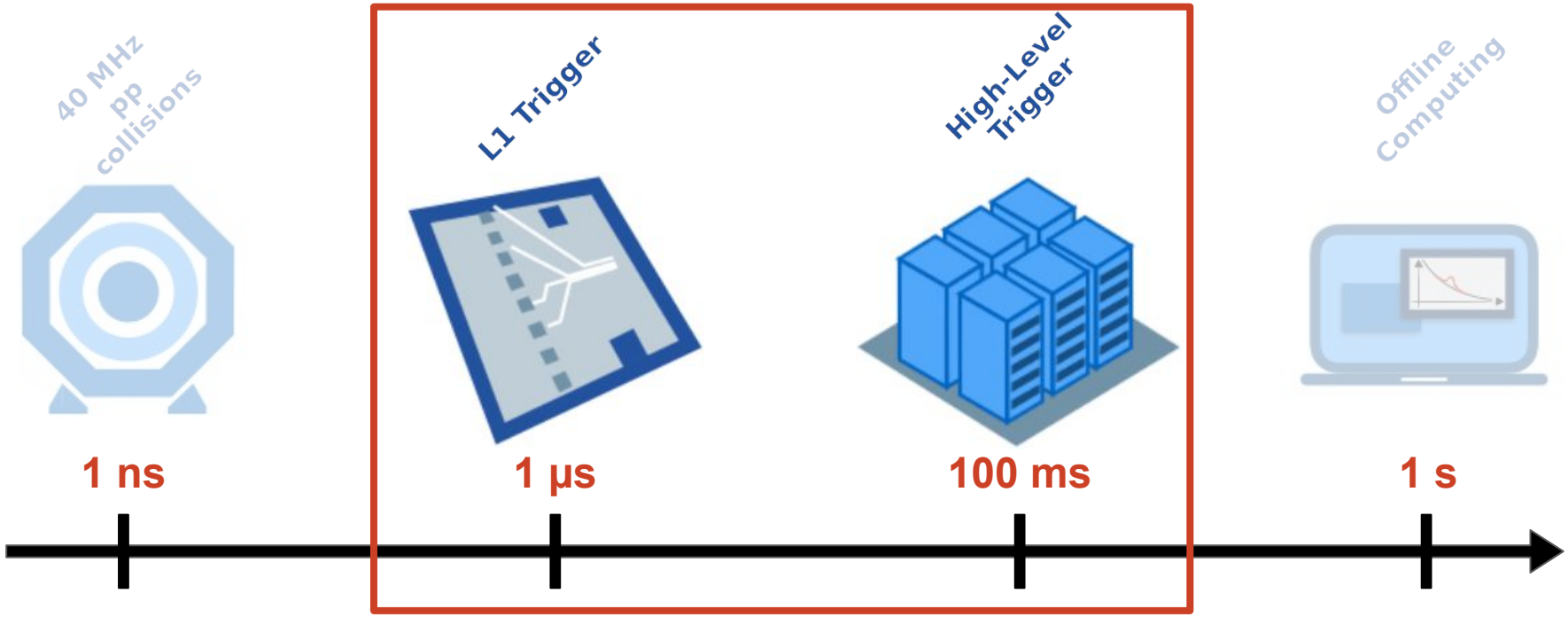
At the LHC proton beams collide at a frequency of 40 MHz

Extreme data rates of O(100 TB/s)

“Triggering” - Filter events to reduce data rates to manageable levels



The LHC big data problem



Deploy ML algorithms very early

Challenge: strict latency constraints!

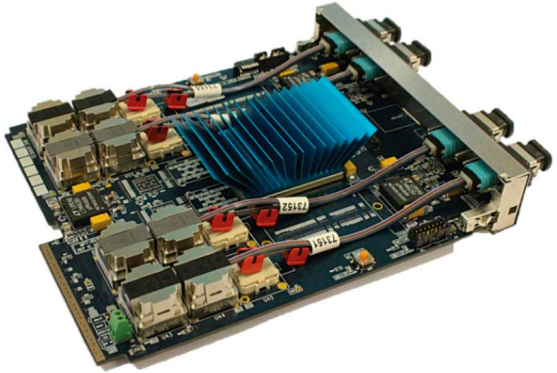
L1 trigger hardware

We need fast processing of raw data *O*(µs)

- Not possible to use common hardware, such as Intel CPUs, nor common operating systems

Must be flexible and modular to support reconfiguration and upgrade/maintenance of modules

➔ Field-programmable gate arrays (FPGAs)



Field-Programmable Gate Array

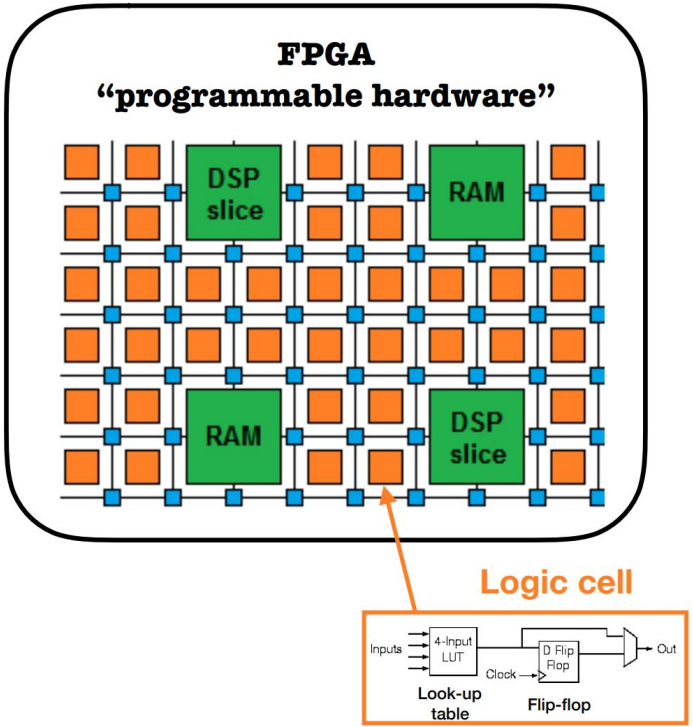
Reprogrammable integrated circuits

Configurable logic blocks and embedded components

- Flip-Flops (registers)
- LUTs (logic)
- DSPs (arithmetic)
- Block RAMs (memory)

Massively parallel

Low power



Why are FPGAs *Fast*?

Fine-grained / resource parallelism

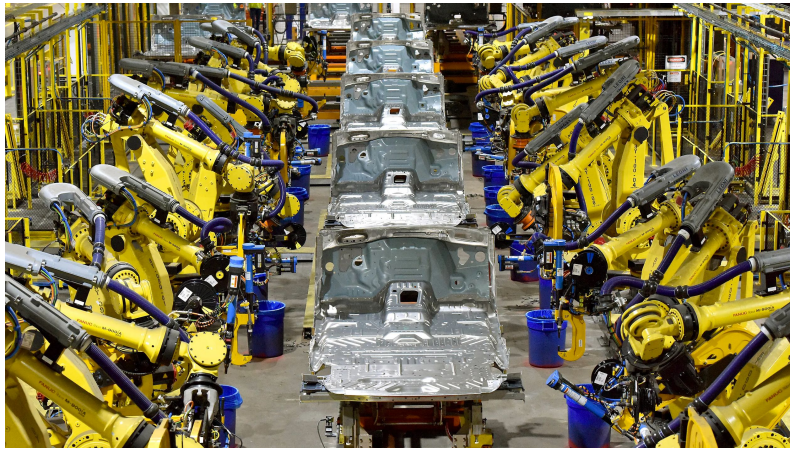
- Work on different parts of the problem simultaneously
- ➔ Allows us to achieve **low latency**

Most problems have at least some sequential aspect, limiting how low latency we can go

- But we can still take advantage of it with…

Pipeline parallelism

- Use the register pipeline to work on different data simultaneously
- ➔ Allows us to achieve **high throughput**



Like a production line for data…

How are FPGAs programmed?

Hardware Description Languages

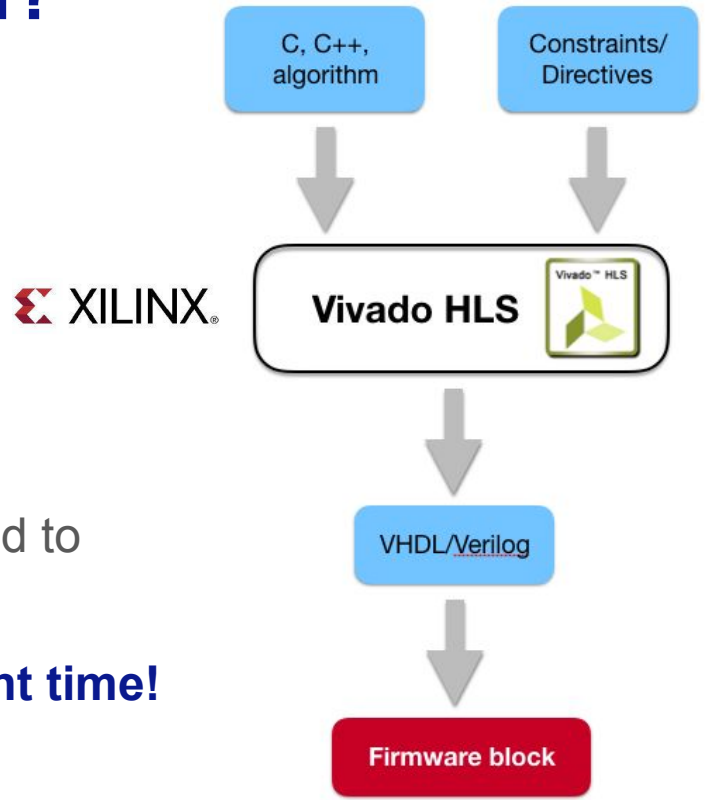
- Languages that describe electronic circuits
- VHDL, Verilog

High Level Synthesis

- Compile from C/C++ to VHDL/Verilog
- Preprocessor directives and constraints used to optimize the design
- **Drastic decrease in firmware development time!**

Many different HLS implementations exist

- Today we'll use Xilinx Vivado HLS



hls4ml pipeline

Supported DL frameworks:

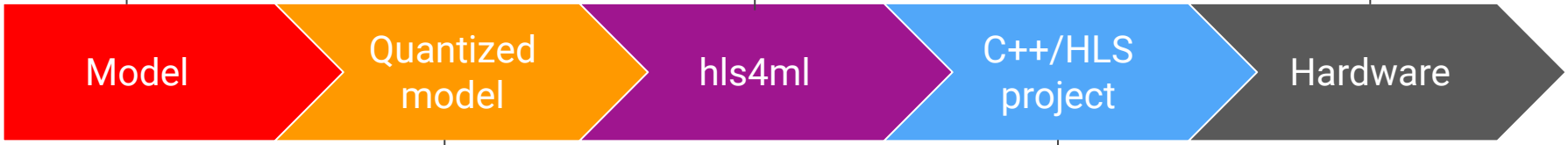
Keras
PyTorch
ONNX

Model conversion, optimization, profiling & tuning

hls4ml

Xilinx FPGAs, Intel/Altera FPGAs, Intel x86 CPUs

XILINX
intel



Quantization and pruning techniques:

- QKeras + AutoQ (Keras)
- Brevitas (PyTorch)

VIVADO
Mentor A Siemens Business
oneAPI
Quartus Prime

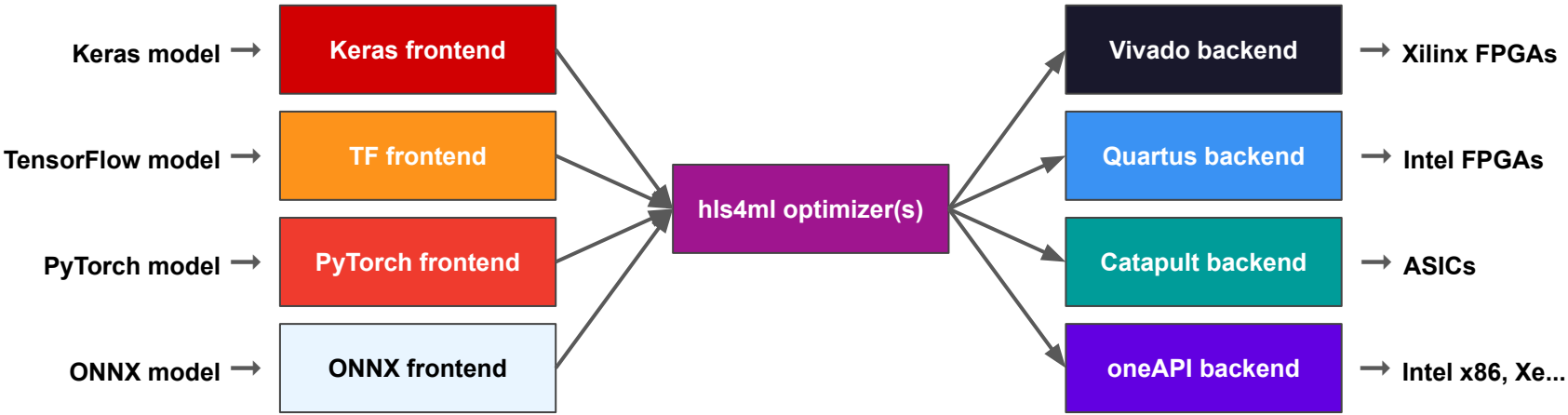
Architecture of hls4ml

Modular design inspired by traditional compilers

- Fully extensible by introducing new frontends/optimizers/backends

Hardware-agnostic internal representation (IR) model

- Abstraction of NN layers, tensors, variables, types, precision
- Evolving



Features of Vivado backend

On-chip weights

- Much faster access times → lower latency
- Weights can be stored in registers or block RAM

User controllable trade-off between resource usage and latency/throughput

- Tuned via “reuse factor”

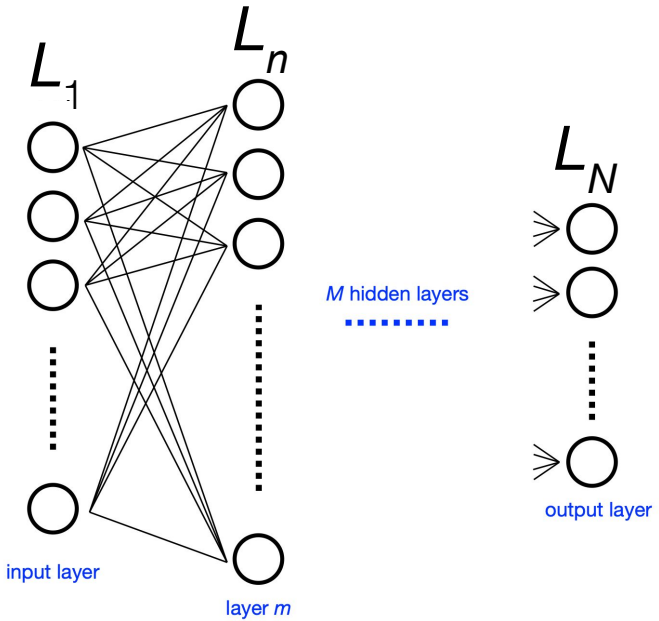
QKeras integration - arxiv:2006.10159

- Binary/Ternary layers (computation without using DSPs) - arxiv:2003.06308

Supported architectures:

- **DNNs**
- **CNNs** NEW
- **RNNs** WIP
- **Graph NNs -** GarNet architecture - arxiv:2008.03601

Neural network inference



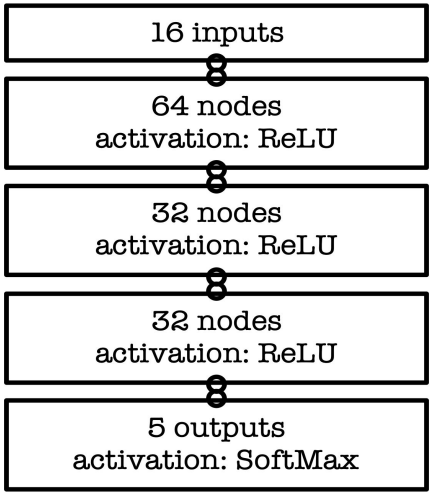
$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1} + \mathbf{b}_n)$$

precomputed and stored in BRAMs

DSPs

logic cells

$$N_{\text{multiplications}} = \sum_{n=2}^{N} L_{n-1} \times L_n$$



How to use hls4ml

Installation:

```
pip install hls4ml
```

Usage via Python API:

```
import hls4ml
my_model = … # build the model in Keras
my_config = hls4ml.utils.config_from_keras_model(my_model)
hls_model = hls4ml.converters.convert_from_keras_model(my_model,
    fpga_part='xcvu9p-flgb2104-2-e', output_dir='my_hls_prj', hls_config=my_config)
report = hls_model.build()
```

Usage via CLI:

```
hls4ml config  --model my_model.onnx --fpga xcvu9p-flgb2104-2-e \
               --dir my_hls_prj --output my_config.yml
hls4ml convert --config my_config.yml
hls4ml build   --project my_hls_prj --all
```

Much, much more is available in the docs

Live demo

Interactive notebooks (GitHub login required): https://cern.ch/ssummers/hls4ml-tutorial

GitHub link for the tutorial: https://github.com/fastmachinelearning/hls4ml-tutorial

Today:

- **Part 1:** Get started with hls4ml: train a basic model and run the conversion, simulation & C-synthesis steps
- **Part 2:** Learn how to tune inference performance with quantization & `ReuseFactor`

On your own:

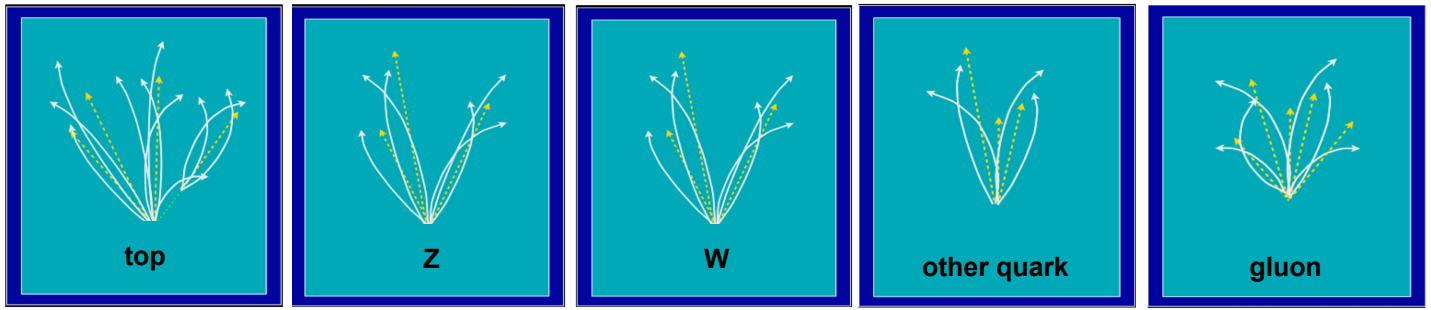
- **Part 3:** Perform model compression and observe its effect on the FPGA resources/latency
- **Part 4:** Train using QKeras “quantization aware training” and study impact on FPGA metrics

Part 1: Model Conversion

Example model - *jet tagging*

Study a **multi-classification task to be implemented on FPGA:** discrimination between highly energetic (boosted) ***q, g, W, Z, t*** initiated *jets*

Jet = collimated 'spray' of particles



t→bW→bqq 3-prong jet

Z→qq 2-prong jet

W→qq 2-prong jet

Reconstructed as one massive jet with substructure

q/g background

no substructure
and/or mass ~ 0

Example model - *jet tagging*

Input variables: several observables known to have high discrimination power from offline data analyses and published studies

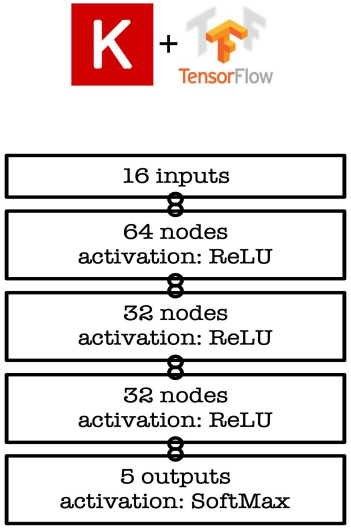
- D. Guest at al. PhysRevD.94.112002, G. Kasieczka et al. JHEP05(2017)006, J. M. Butterworth et al. PhysRevLett.100.242001, etc..

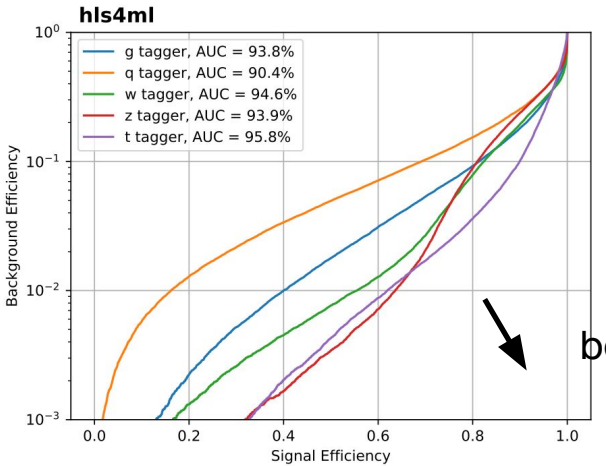
We'll train the **five class multi-classifier** on a sample of ~ 1M events with two boosted WW/ZZ/tt/qq/gg anti-k_T jets

- Dataset DOI: 10.5281/zenodo.3602254
- OpenML: https://www.openml.org/d/42468

Fully connected neural network with 16 inputs:

- Relu activation function for intermediate layers
- Softmax activation function for output layer

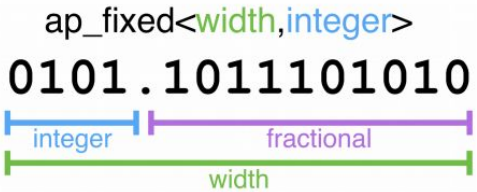




AUC = area under ROC curve
(100% is perfect, 20% is random)

Part 2: Controlling the conversion process

Controlling model conversion: quantization



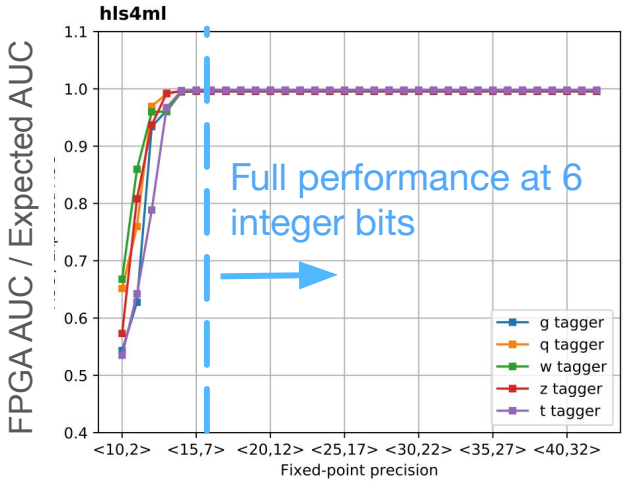
In the FPGA we use fixed point representation

- Operations are integer ops, but we can represent fractional values

But we have to make sure we've used the correct data types!

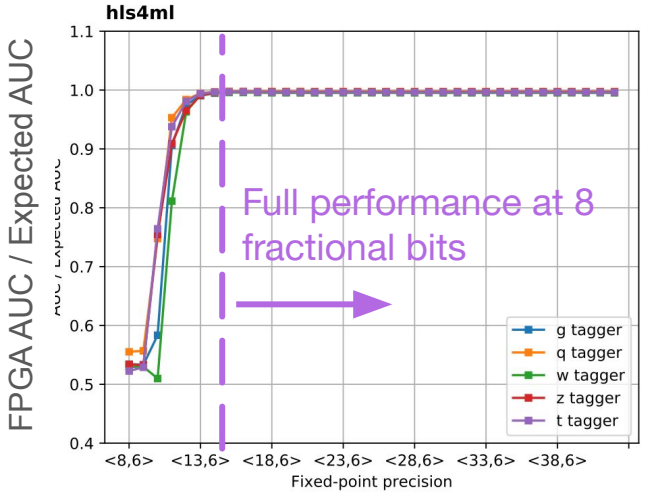
Scan integer bits

Fractional bits fixed to 8



Scan fractional bits

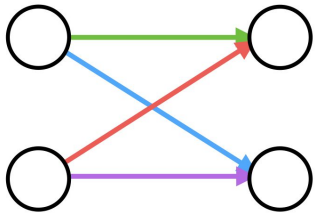
Integer bits fixed to 6

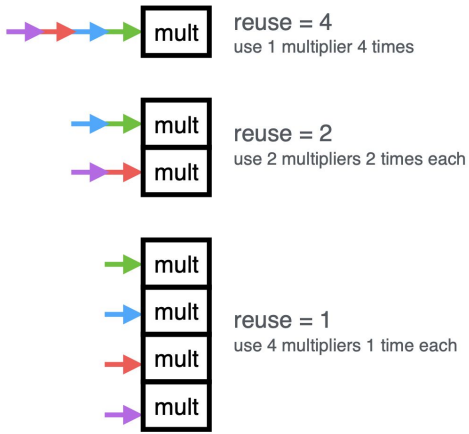


Controlling model conversion: parallelization

Trade-off between latency and FPGA resource usage determined by the parallelization of the calculations in each layer

Configure the **“reuse factor”** = number of times a multiplier is used to do a computation





Fully serial

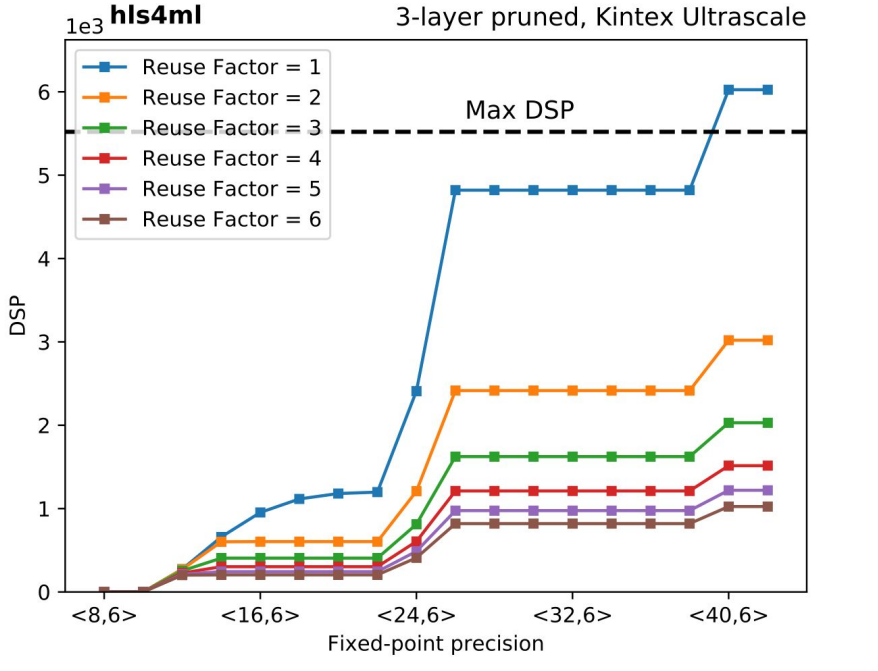
Fewer resources, Lower throughput, Higher latency

More resources, Higher throughput, Lower latency

Fully parallel

Reuse factor: how much to parallelize operations in a hidden layer

Parallelization: DSP usage



Fully parallel
Each mult. used 1x

⋮

Each mult. used 2x

⋮

Each mult. used 3x

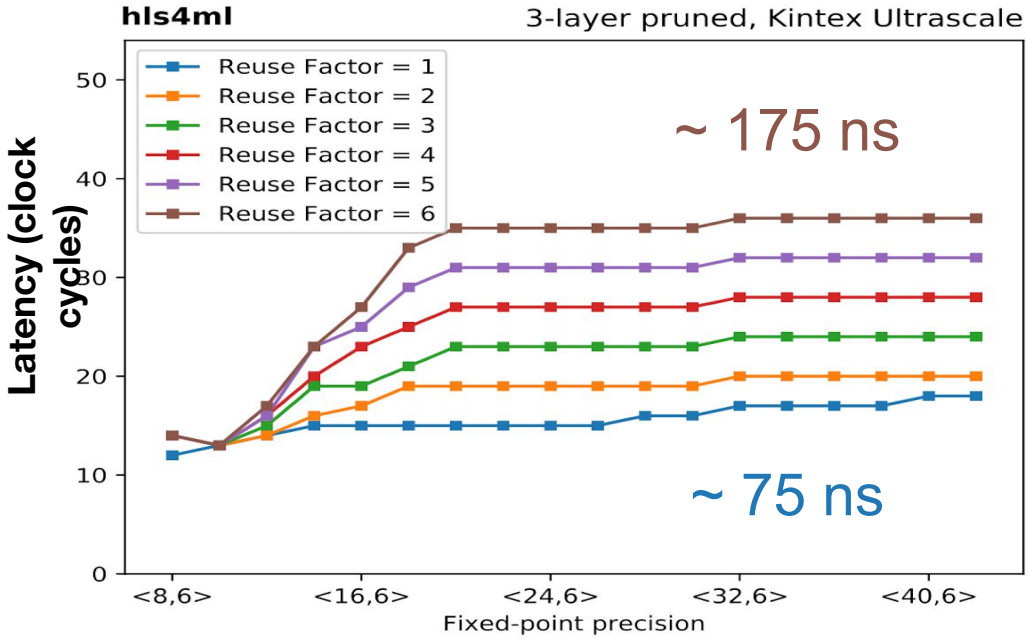
More resources

Longer latency

Parallelization: Timing

Latency of layer m

$$L_m = L_{\text{mult}} + (R-1) \times II_{\text{mult}} + L_{\text{activ}}$$



Longer latency

Each mult. used 6x

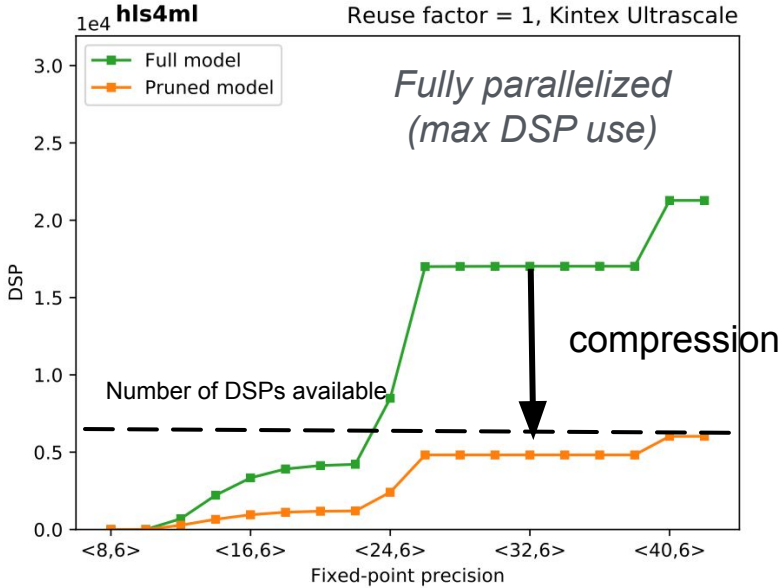
Each mult. used 3x

Fully parallel
Each mult. used 1x

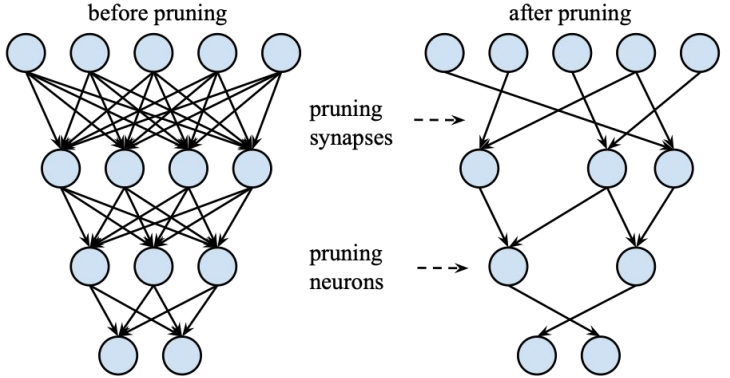
More resources

Going further: Efficient NN design

Efficient NN design: compression



70% compression ~ 70% fewer DSPs



DSPs (used for multiplication) are often the limiting resource

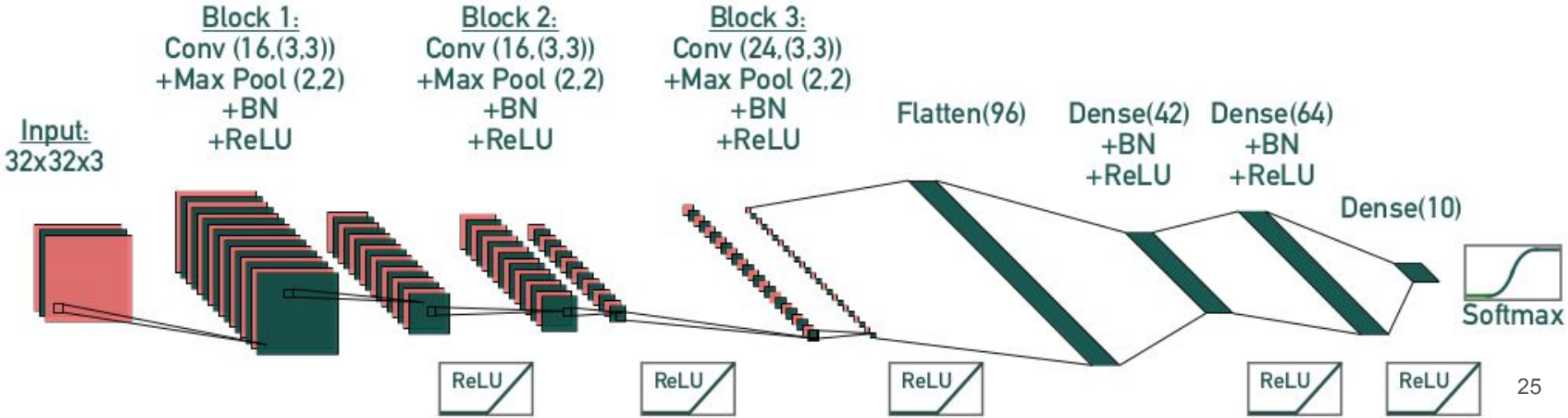
- Maximum use when fully parallelized
- DSPs have a max size for input (e.g. 27x18 bits), so number of DSPs per multiplication changes with precision

Advanced example: *digit classification with CNNs*

Street-view house numbers dataset (SVHN)

- 32x32x3 images
- A tougher MNIST

Model architecture:



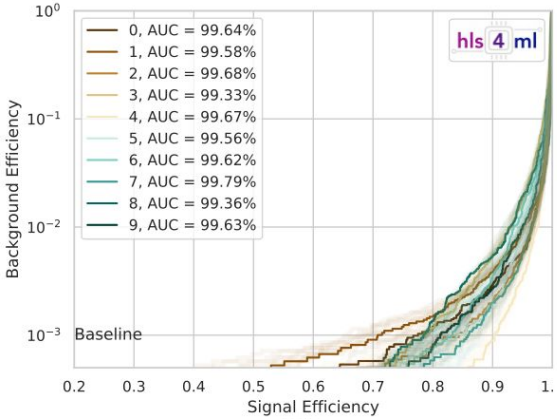
CNN model performance

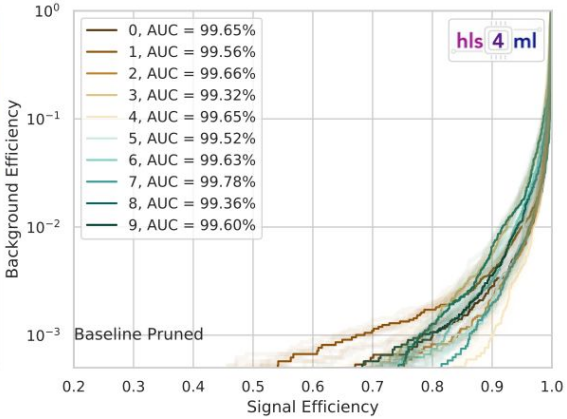
Baseline models

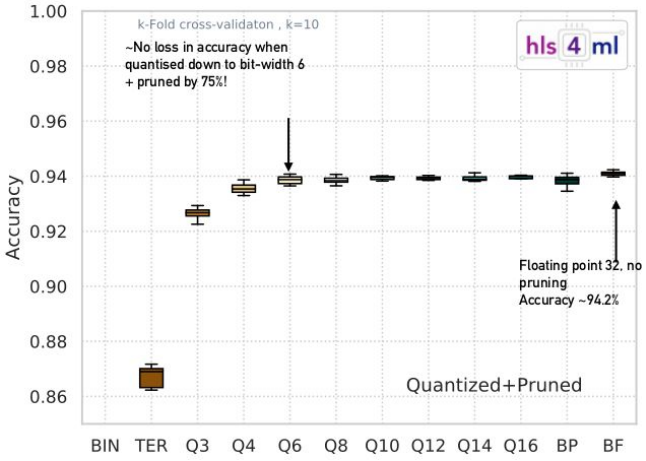
- Full 32-bit precision (BF)
- Full 32-bit precision, pruned (BP)
  - 75% sparsity
  - Polynomial decay

QKeras models

- Quantized (Q)
  - Binary (1-bit)
  - Ternary (2-bit)
  - Quantized to 3-16 bits
- Pruned (QP)
  - 75% sparsity







CNN model performance on an FPGA

Targeting a Xilinx Virtex UltraScale+ VU9P series FPGA

Vivado HLS 2020.1

200MHz clock

Table 3: Accuracy, resource consumption and latency for the Baseline Full (BF) and Baseline Pruned (BP) models quantized to a bit width of 14, the QKERAS (Q) and QKERAS Pruned (QP) models quantized to a bit width of 7 and the heterogeneously quantized AutoQ (AQ) and AutoQ Pruned (AQP) models. The numbers in parentheses correspond to the total amount of resources used.

Model	Accuracy	DSP [%]	LUT [%]	FF [%]	BRAM [%]	Latency [cc]	II [cc]
BF 14-bit	0.87	93.23 (6377)	19.36 (228823)	3.40 (80278)	3.08 (66.5)	1035	1030
BP 14-bit	0.92	48.85 (3341)	12.27 (145089)	2.77 (65482)	3.08 (66.5)	1035	1030
Q 7-bit	0.93	2.56 (175)	12.77 (150981)	1.51 (35628)	3.10 (67.0)	1034	1029
QP 7-bit	0.93	2.54 (174)	9.40 (111152)	1.38 (32554)	3.10 (67.0)	1035	1030
AQ	0.85	1.05 (72)	4.06 (48027)	0.64 (15242)	1.5 (32.5)	1059	1029
AQP	0.88	1.02 (70)	3.28 (38795)	0.63 (14802)	1.4 (30.5)	1059	1029

More details in our paper!

Conclusions

hls4ml - software package for translation of trained neural networks into synthesizable FPGA firmware

- Tunable resource usage latency/throughput
- Fast inference times, O(1μs) latency

Currently being extended to multiple hardware architectures

- FPGAs, CPUs, GPUs etc

More information:

- Website: https://hls-fpga-machine-learning.github.io/hls4ml/
- Code: https://github.com/hls-fpga-machine-learning/hls4ml
- Tutorial: http://cern.ch/ssummers/hls4ml-tutorial