

hls4ml tutorial

4th IML Workshop

19th October 2020

Sioni Summers

for the **hls4ml** team

Introduction

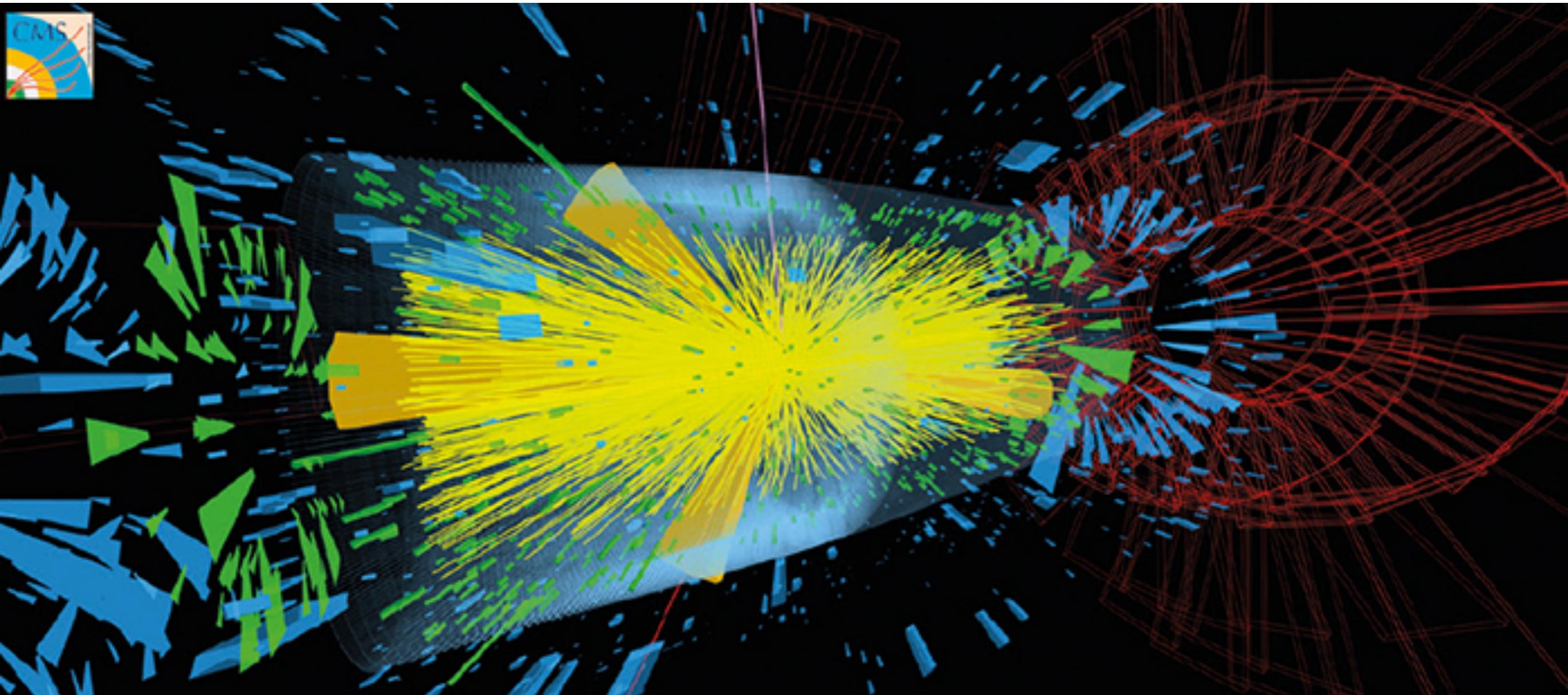
- **hls4ml** is a package for translating neural networks to FPGA firmware for inference with extremely low latency on FPGAs
 - <https://github.com/hls-fpga-machine-learning/hls4ml>
 - <https://fastmachinelearning.org/hls4ml/>
 - pip install hls4ml
- In this session you will get hands on experience with the **hls4ml** package
- We'll learn how to:
 - Translate models into synthesizable FPGA code
 - Explore the different handles provided by the tool to optimize the inference
 - Latency, throughput, resource usage
 - Make our inference more computationally efficient with pruning and quantization

hls4ml origins: triggering at (HL-)LHC

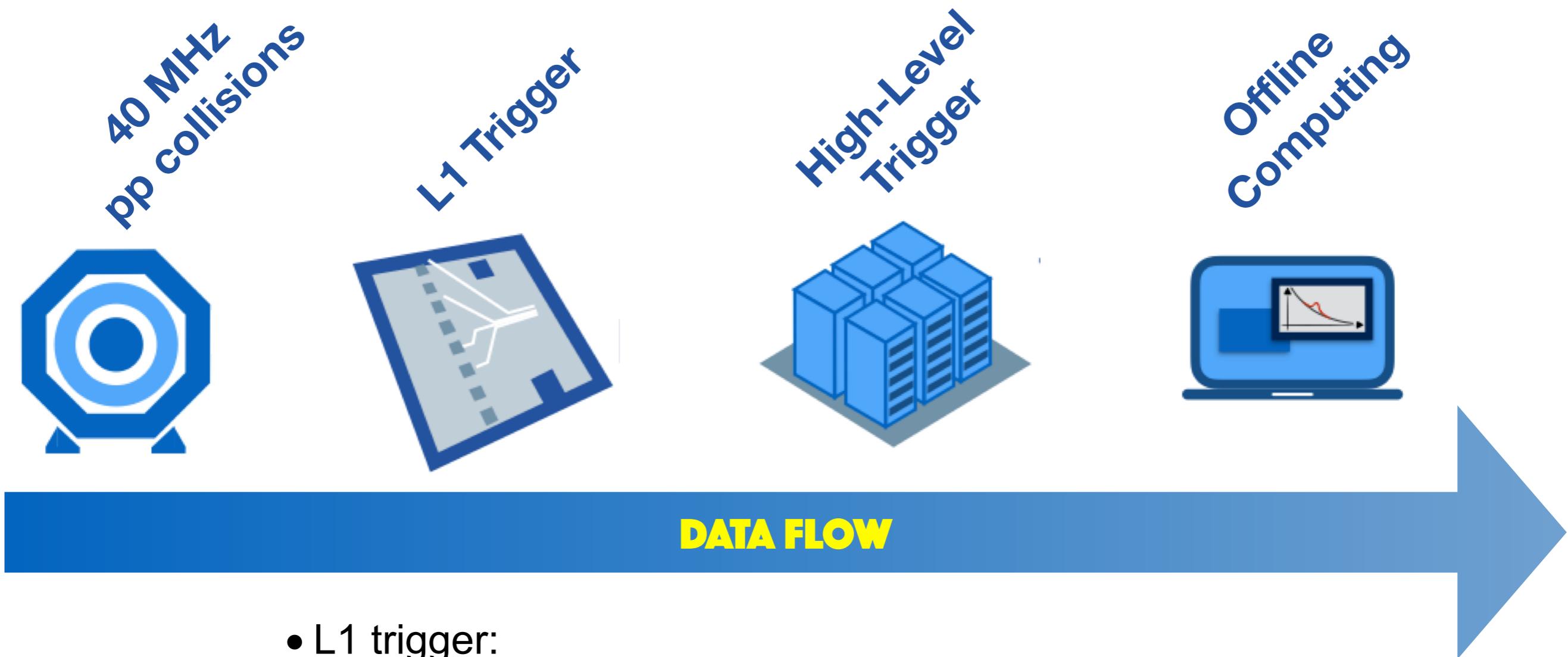
Extreme collision frequency of 40 MHz → extreme data rates $O(100 \text{ TB/s})$

Most collision “events” don’t produce interesting physics

“Triggering” = filter events to reduce data rates to manageable levels

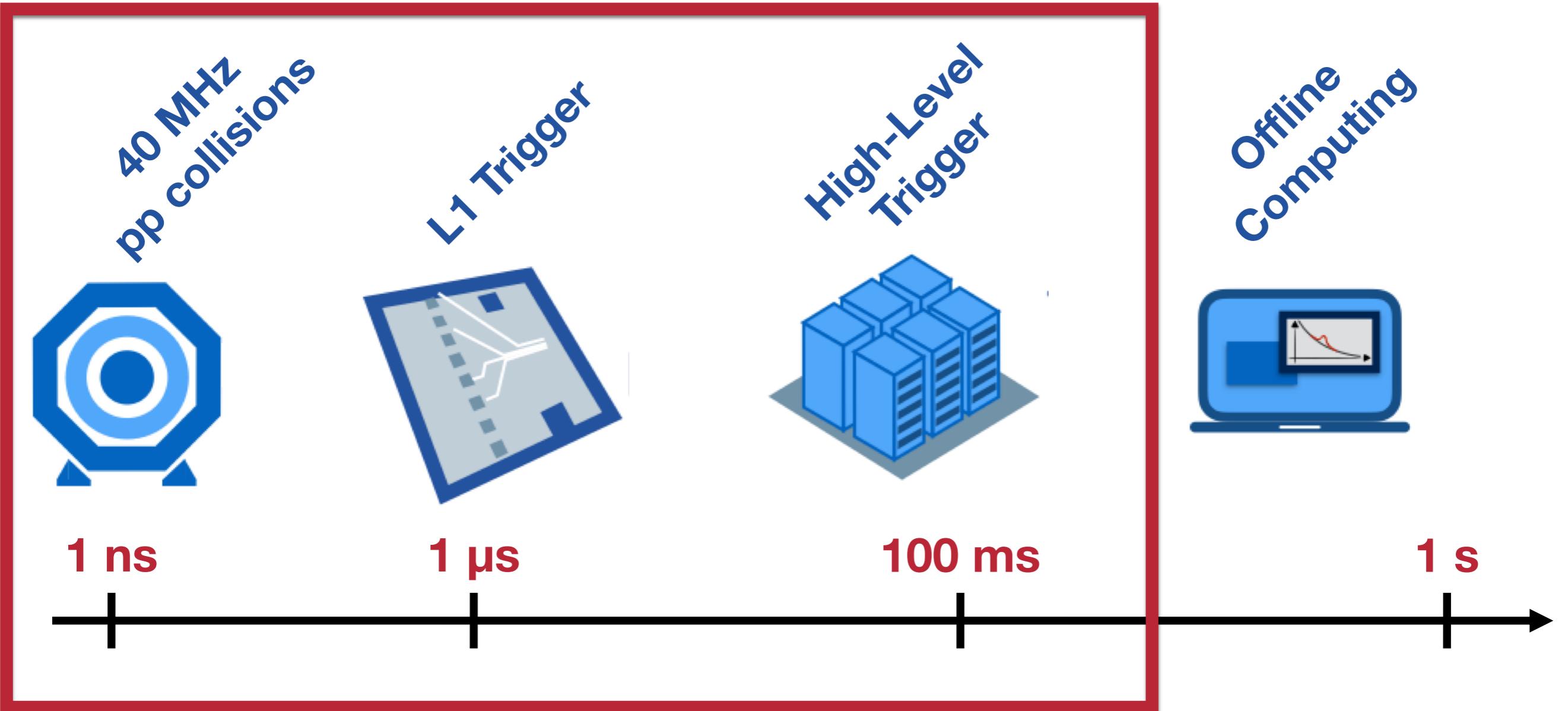


LHC Experiment Data Flow



- L1 trigger:
- 40 MHz in / 100 kHz out
- Process 100s TB/s
- Trigger decision to be made in $\approx 10 \mu\text{s}$
- Coarse local reconstruction
- FPGAs / Hardware implemented

LHC Experiment Data Flow



Deploy ML algorithms very early in the game
Challenge: strict latency constraints!

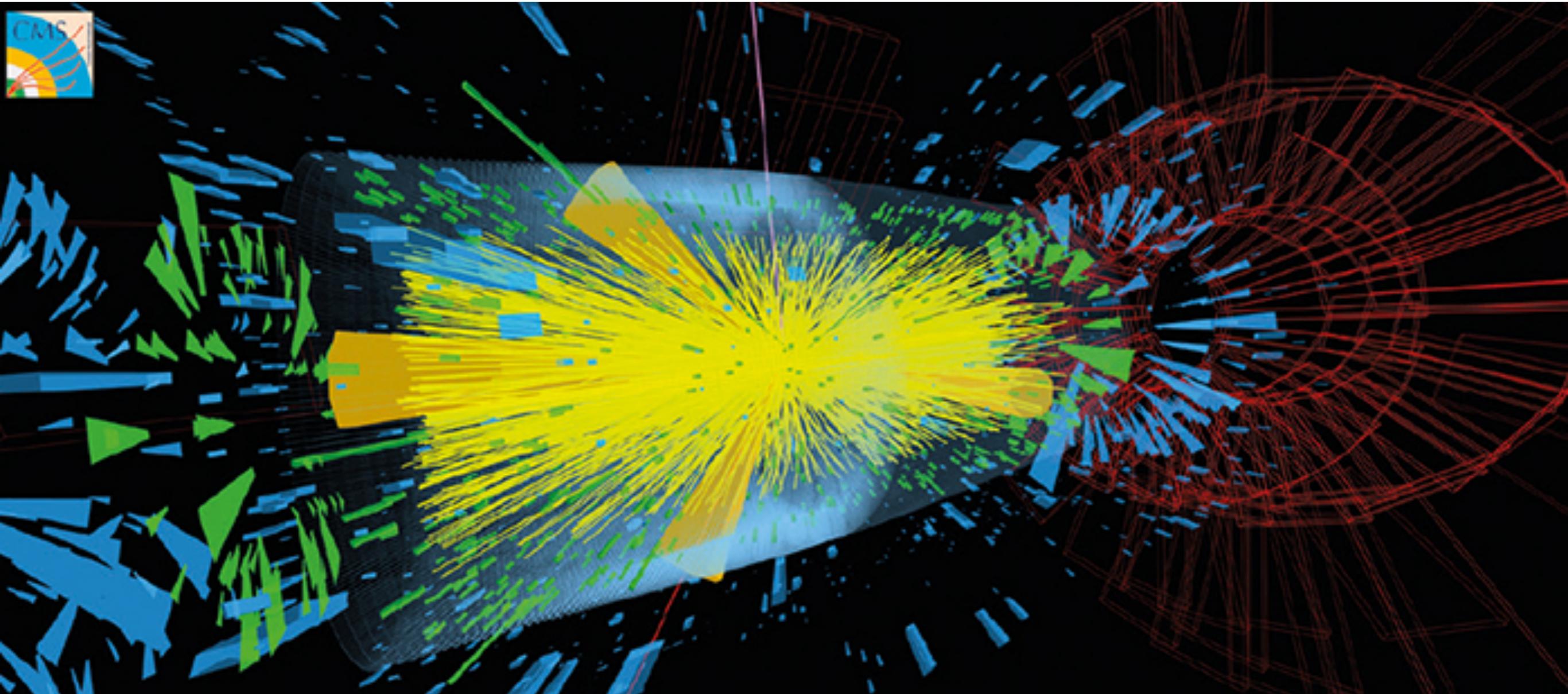
The challenge: triggering at (HL-)LHC

The trigger discards events **forever**, so selection must be very precise

ML can improve sensitivity to rare physics

Needs to be **fast!**

Enter: **hls4ml** (high level synthesis for machine learning)



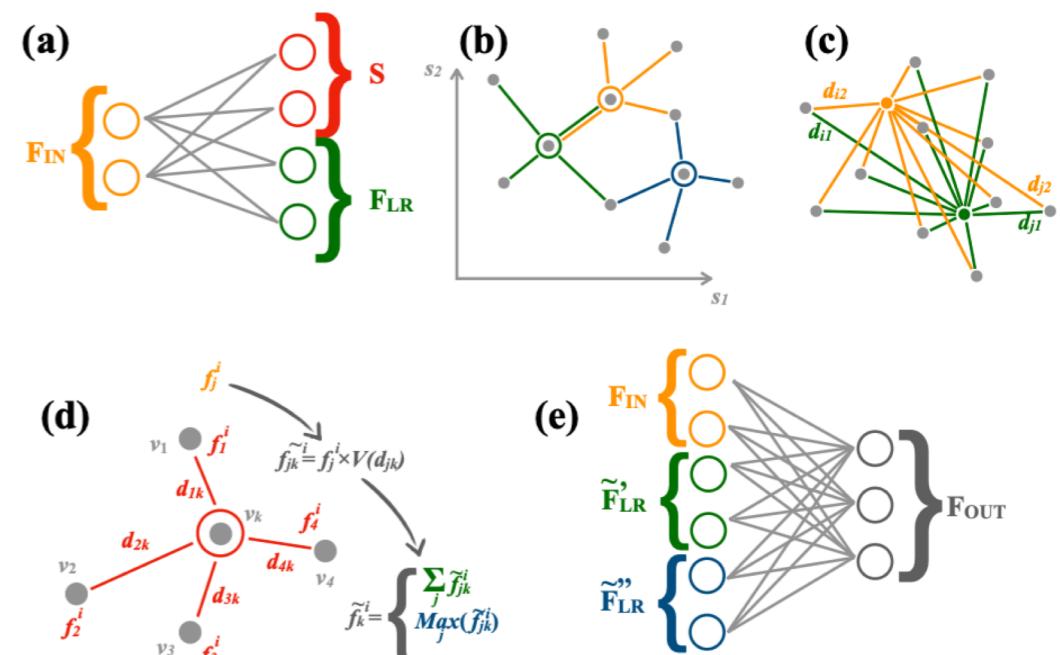
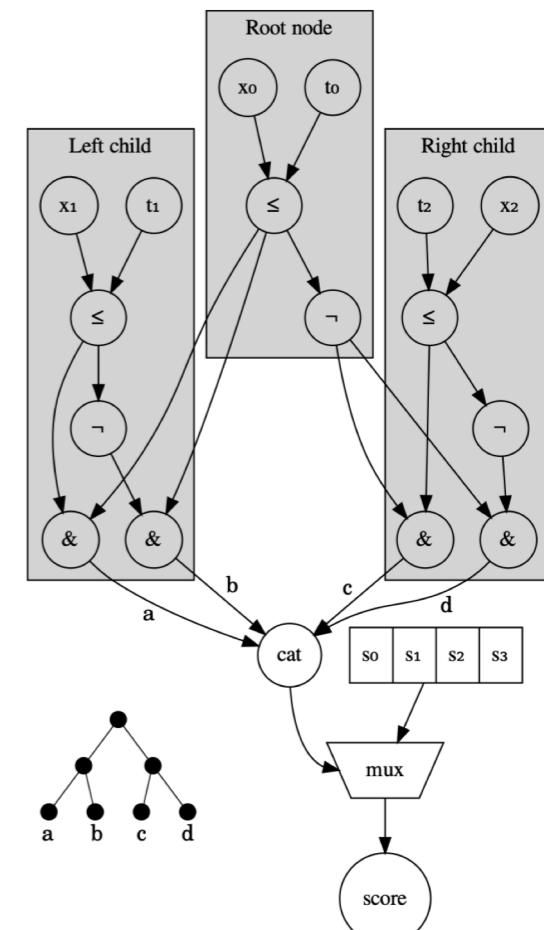
hls4ml: progression

- Previous slides showed the original motivation for hls4ml
 - . Extreme low latency, high throughput domain
- Since then, we have been expanding!
 - . Longer latency domains, larger models, resource constrained
 - . Different FPGA vendors
 - . New applications, new architectures
- While maintaining core characteristics:
 - . “Layer-unrolled” HLS library → not another DPU
 - . Extremely configurable: precision, resource vs latency/throughput tradeoff
 - . Research project, application- and user-driven
 - . Accessible, easy to use

Recent Developments

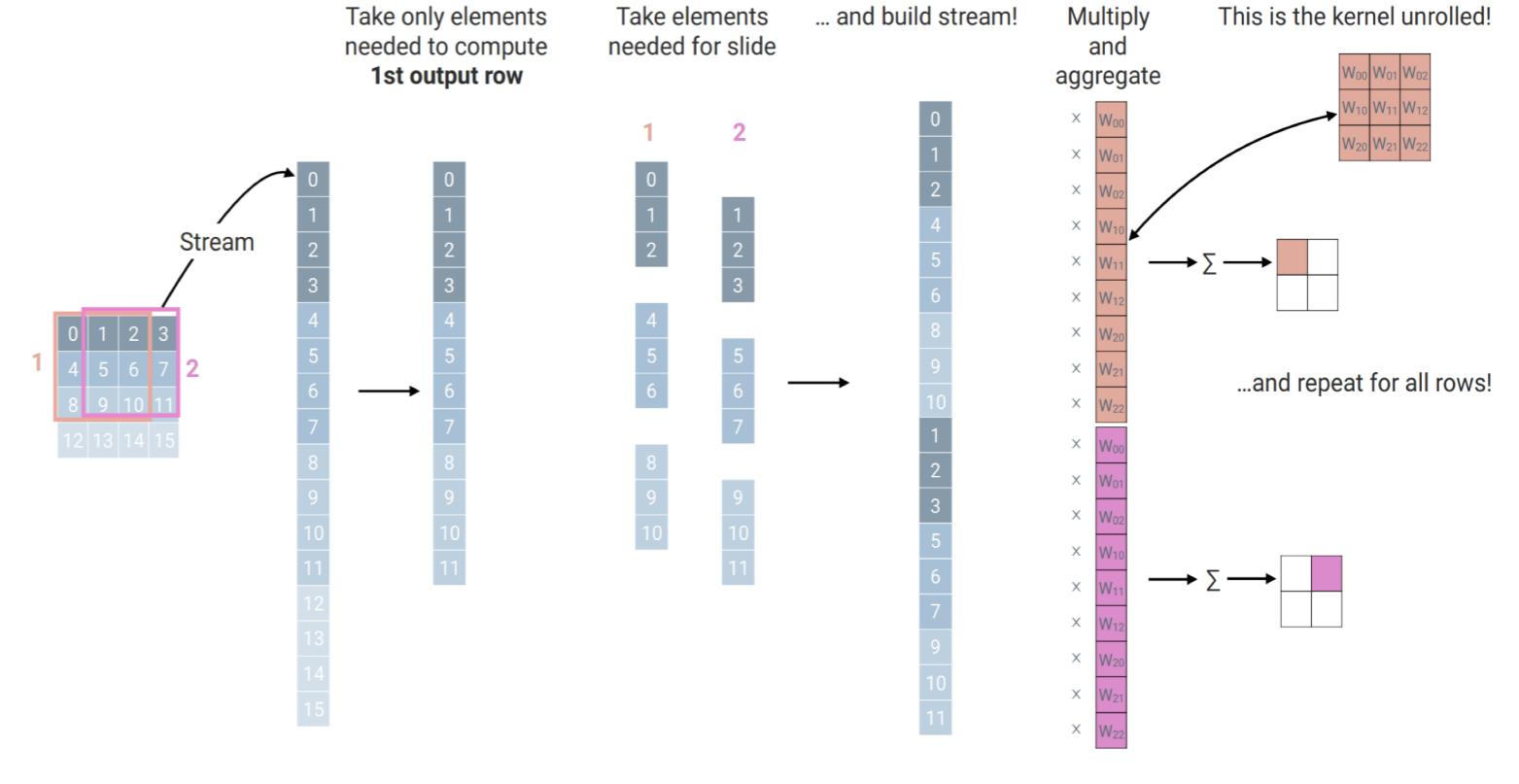
hls4ml community is very active!

- Binary & Ternary neural networks: [\[2020 Mach. Learn.: Sci. Technol\]](#)
 - Compressing network weights for low resource inference
- Boosted Decision Trees: [\[JINST 15 P05026 \(2020\)\]](#)
 - Low latency inference of Decision Tree ensembles
- GarNet / GravNet: [\[arXiv: 2008.03601\]](#)
 - Distance weighted graph neural networks suitable for sparse and irregular point-cloud data, such as from LHC detectors
 - Implemented with low latency for FPGAs in hls4ml
- Quantization aware training QKeras + support in hls4ml: [\[arXiv: 2006.10159\]](#)



Coming Soon

- A few exciting new things should become available soon (this year):
 - Intel Quartus HLS & Mentor Catapult HLS ‘Backends’
 - Convolutional Neural Networks
 - Much larger models than we’ve supported before
 - Recurrent Neural Networks
 - More integrated ‘end-to-end’ flow with bitfile generation and host bindings for platforms like Alveo, PYNQ



= Data pack of channels{c1, c2, c3} values

14

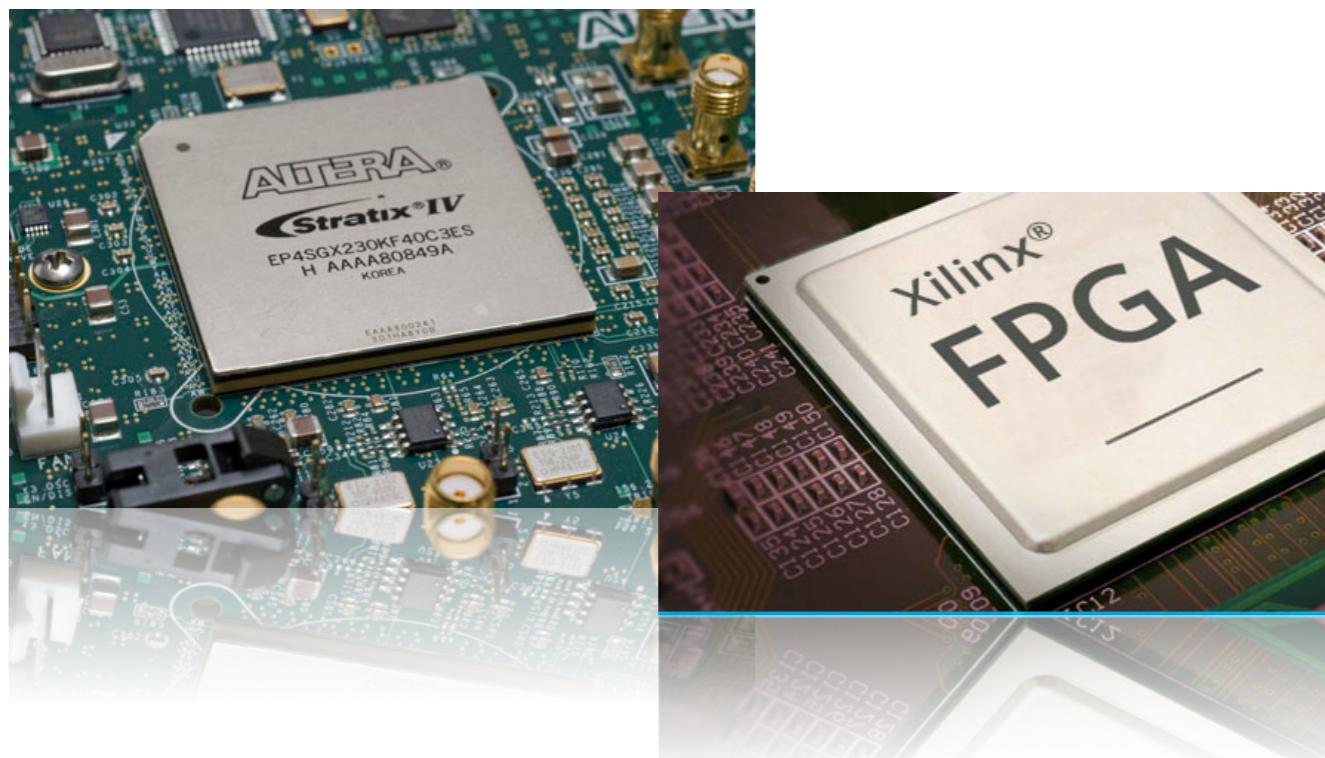
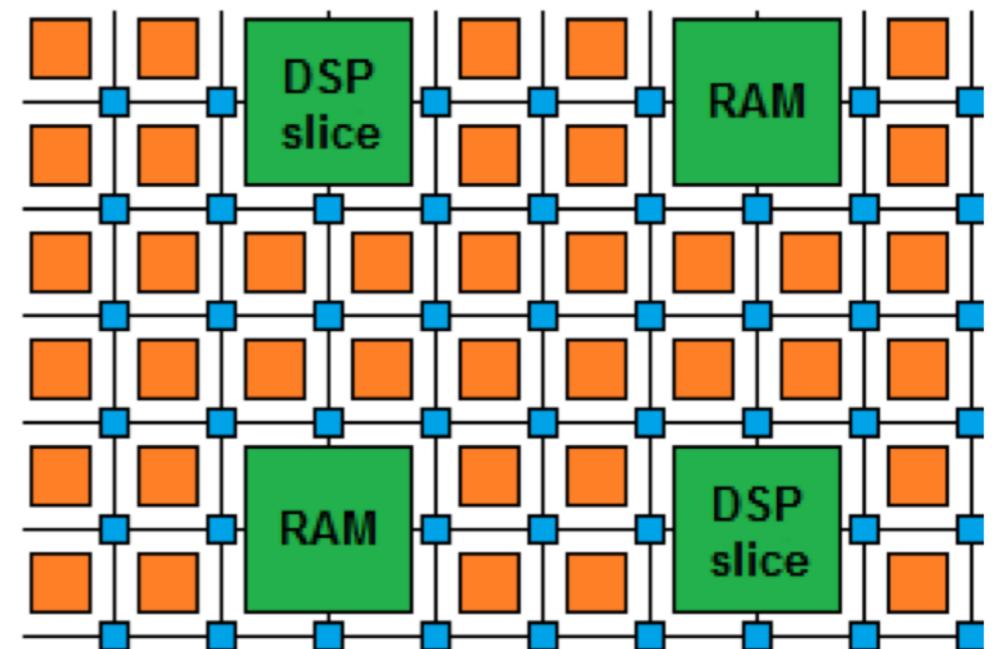
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

Contain many different building blocks ('resources') which are connected together as you desire

Originally popular for prototyping ASICs, but now also for high performance computing

FPGA diagram



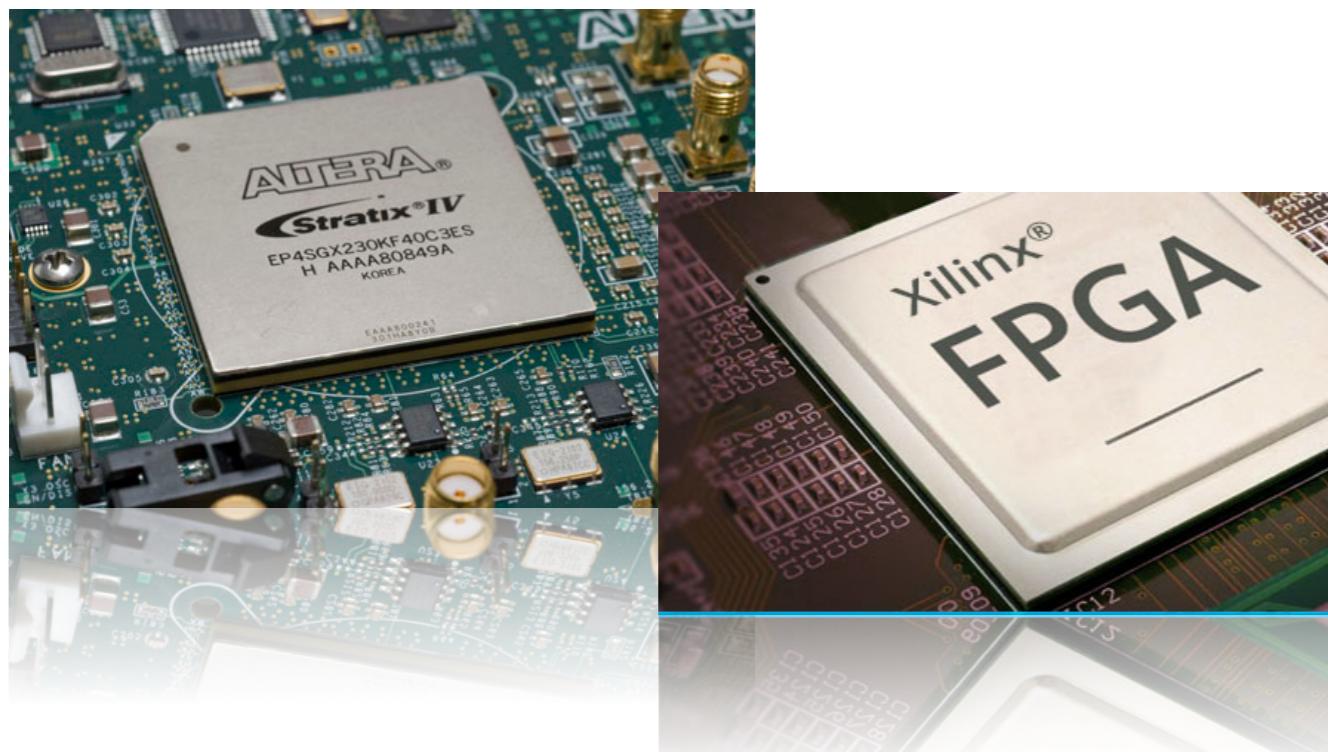
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

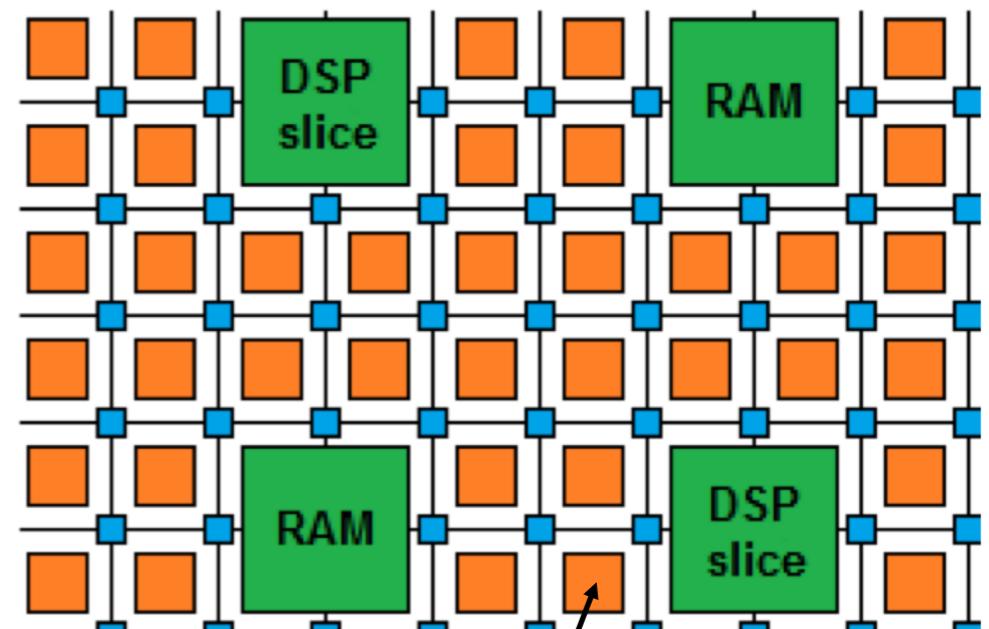
Logic cells / Look Up Tables perform arbitrary functions on small bitwidth inputs (2-6)

These can be used for boolean operations, arithmetic, small memories

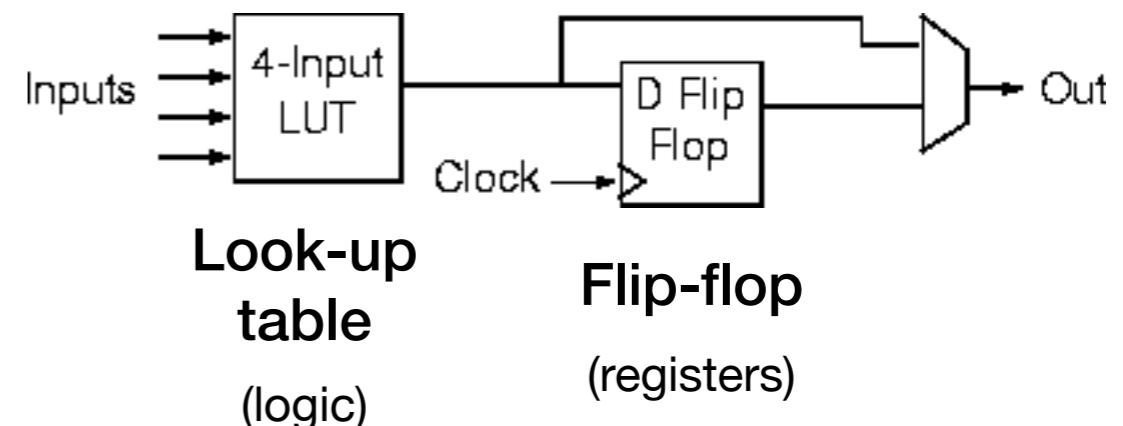
Flip-Flops register data in time with the clock pulse



FPGA diagram



Logic cell



Look-up
table
(logic)

Flip-flop
(registers)

What are FPGAs?

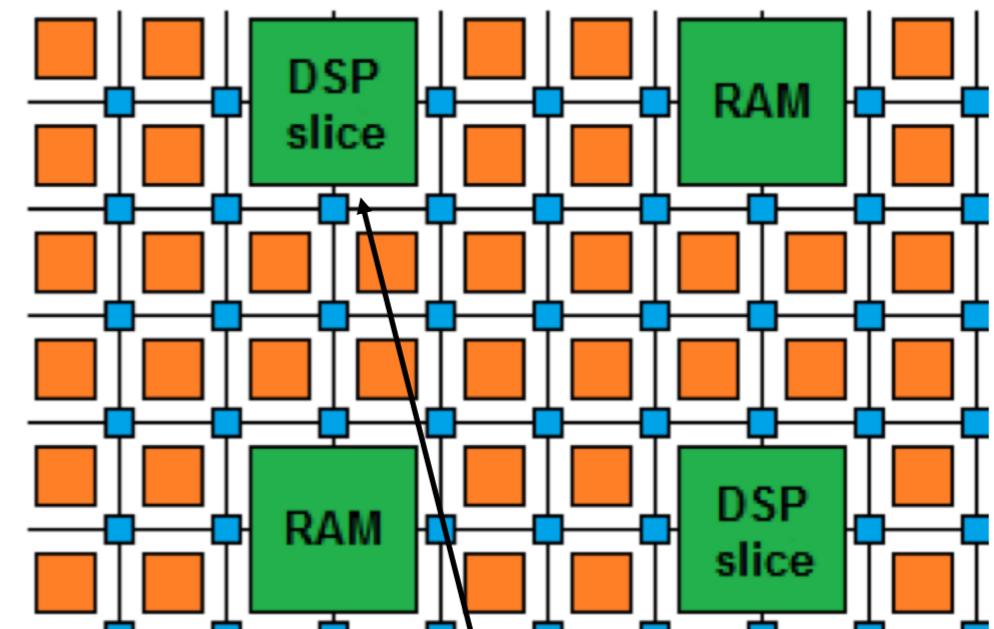
Field Programmable Gate Arrays are reprogrammable integrated circuits

DSPs (Digital Signal Processor) are specialized units for multiplication and arithmetic

Faster and more efficient than using LUTs for these types of operations

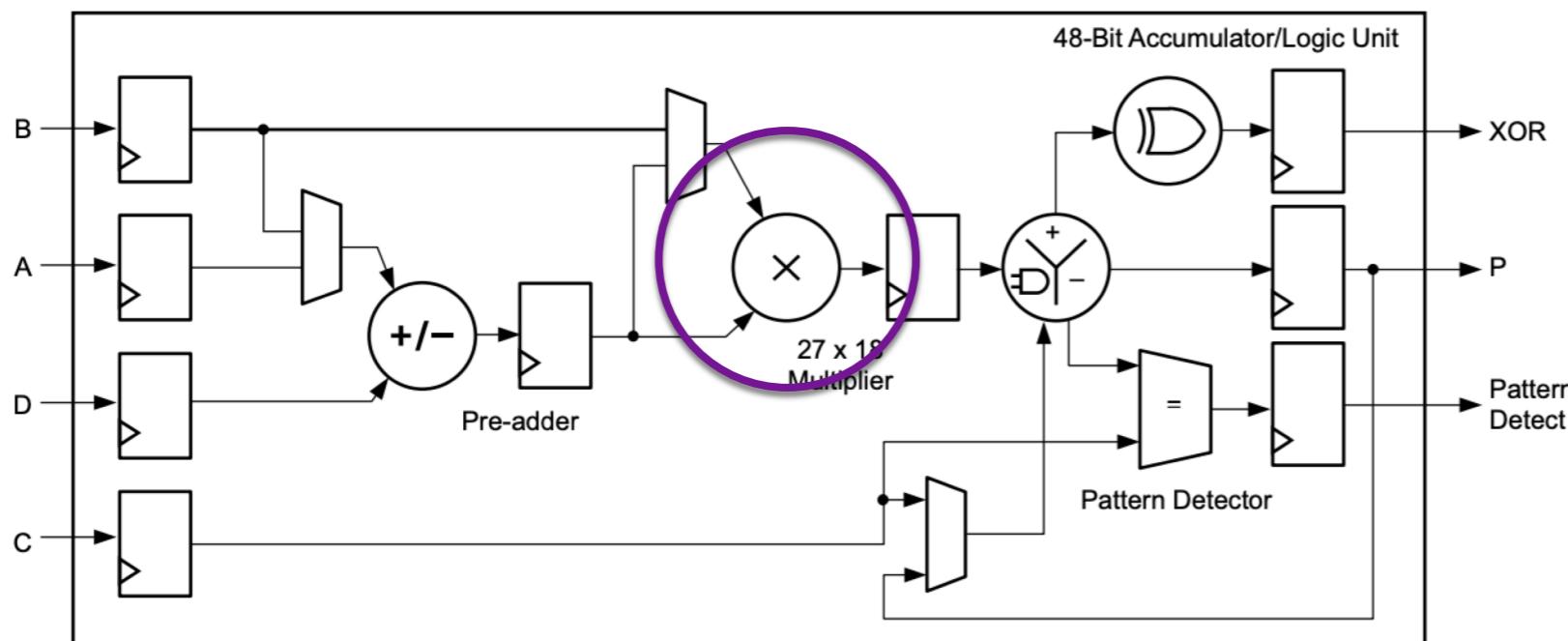
And for Neural Nets, DSPs are often the most scarce

FPGA diagram



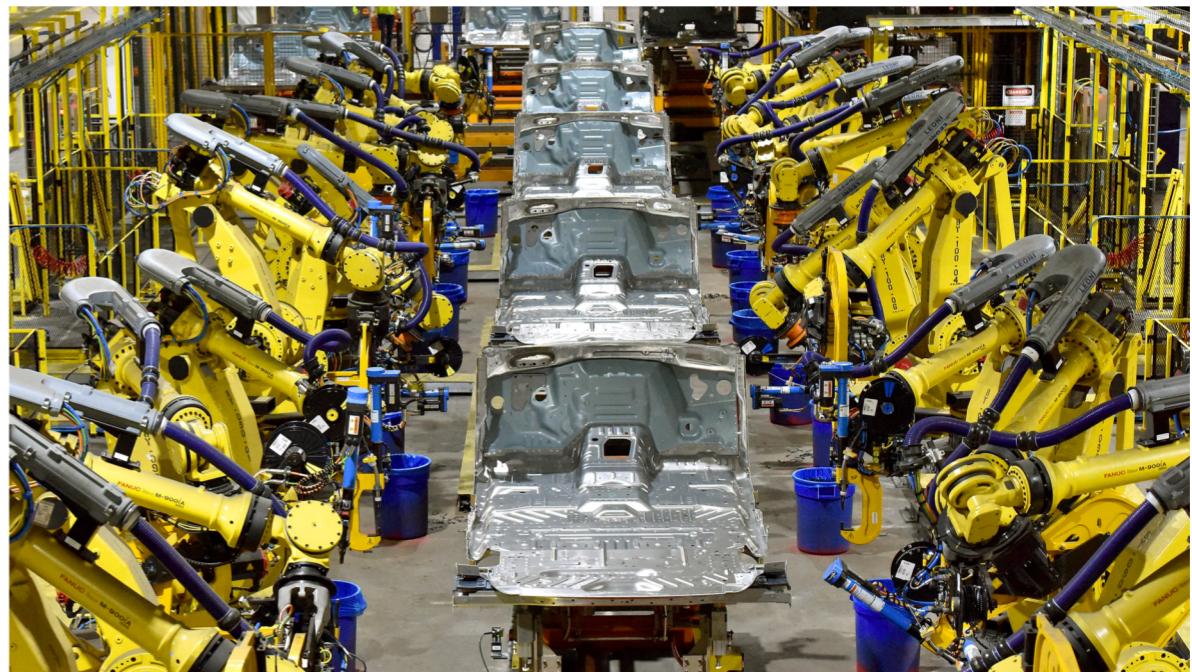
DSP

(multiplication)



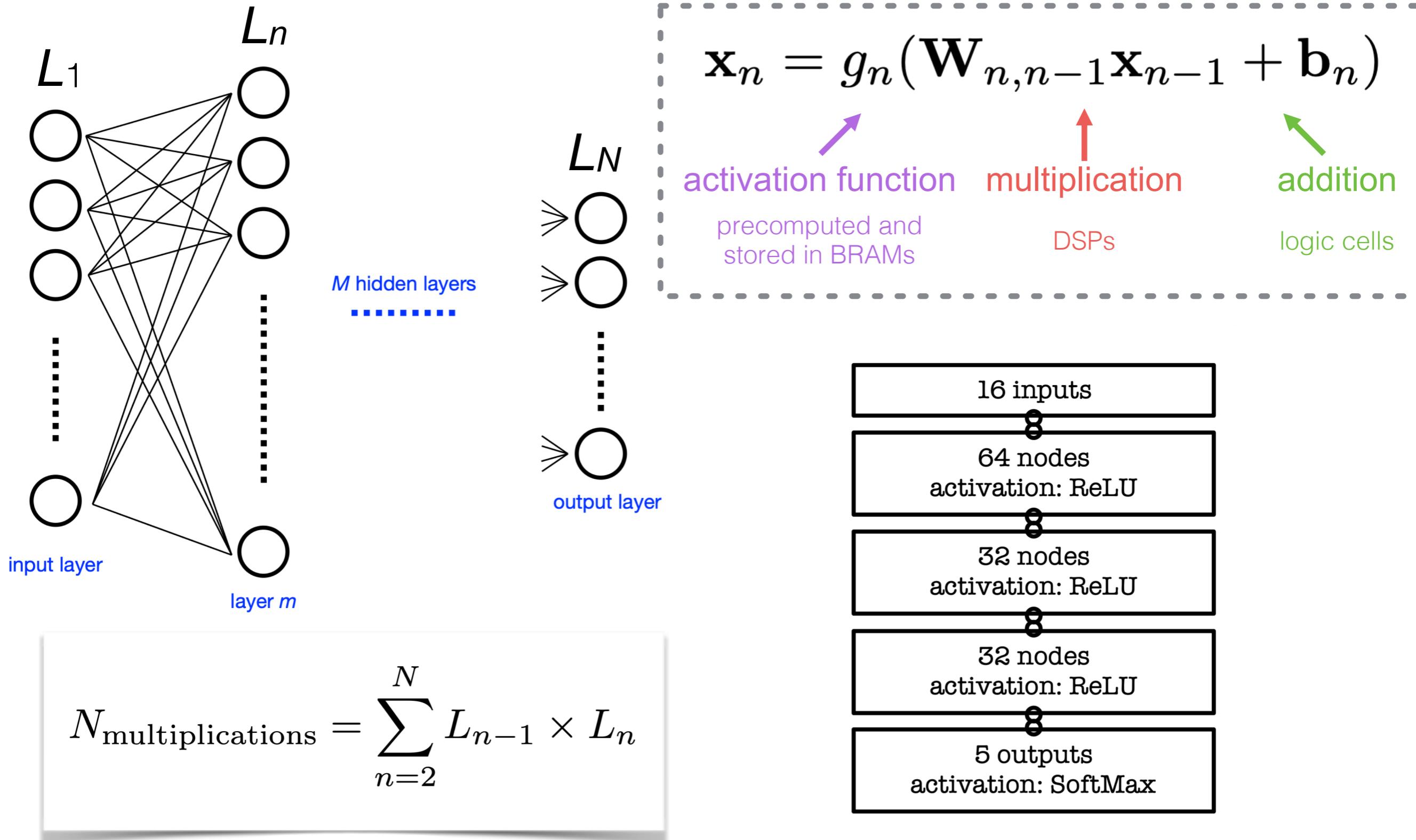
Why are FPGAs *Fast*?

- Fine-grained / resource parallelism
 - Use the many resources to 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**

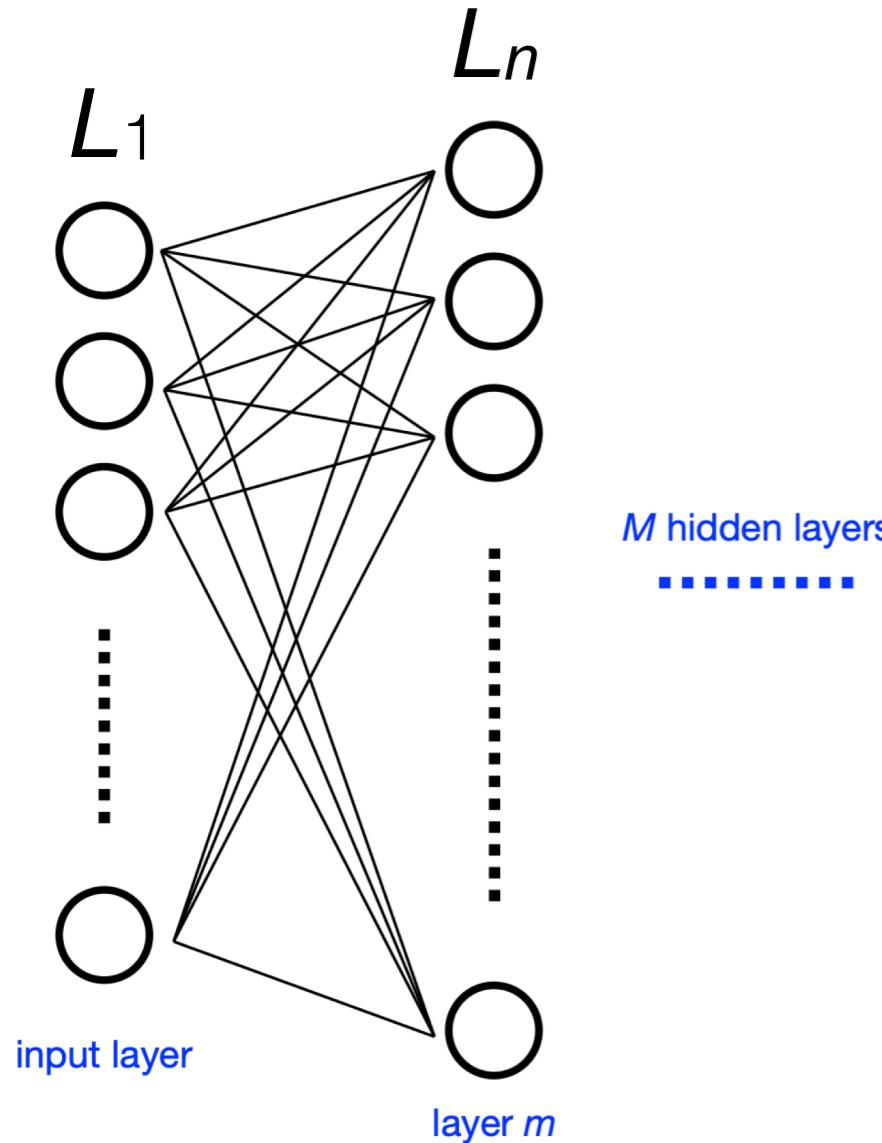


Like a production line for data...

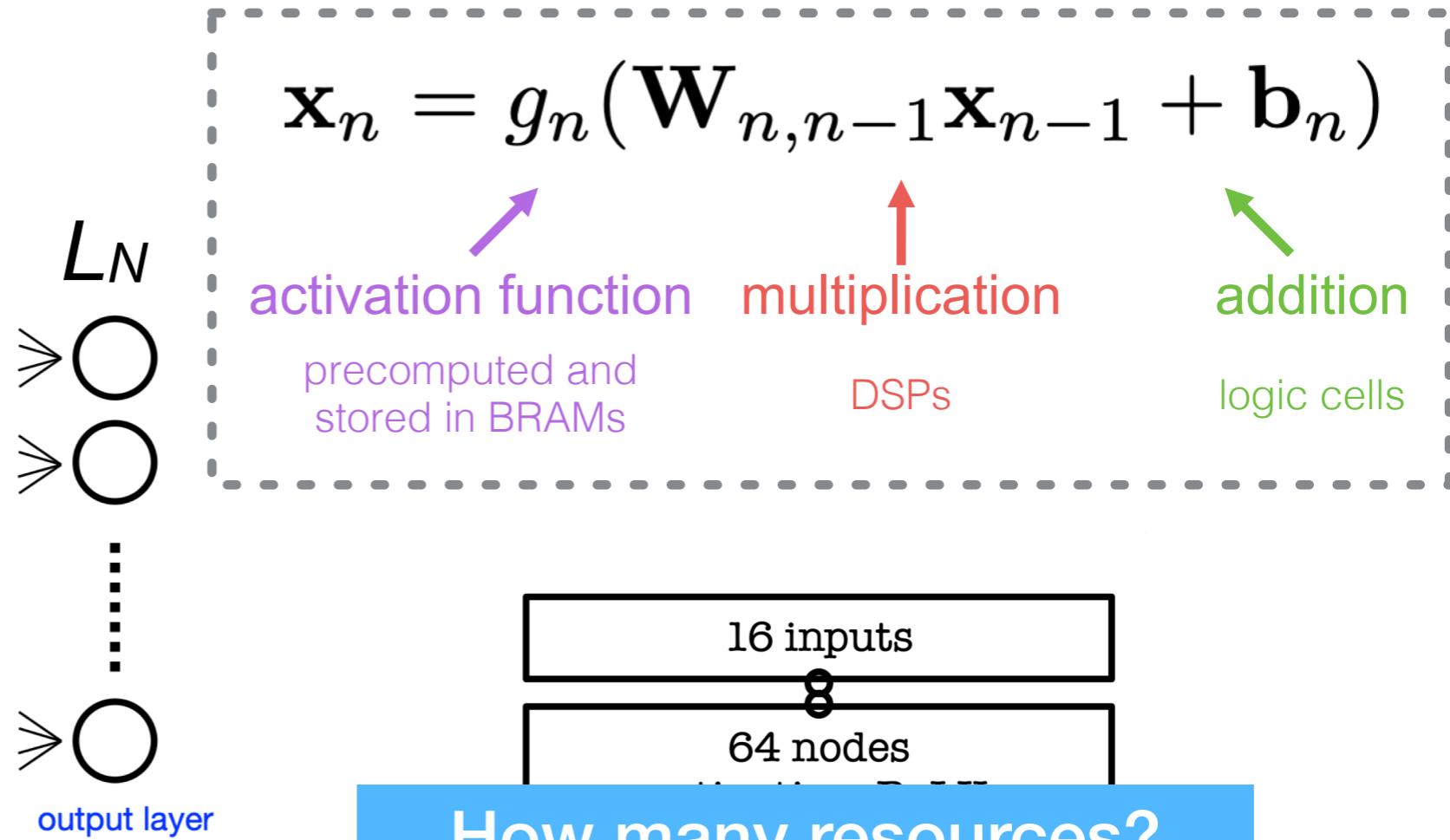
Neural network inference



Neural network inference



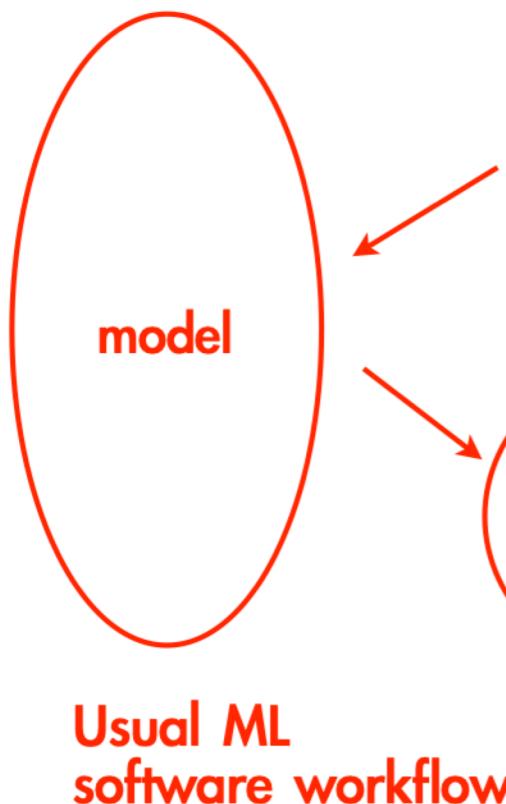
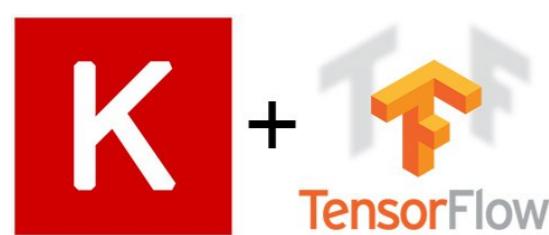
$$N_{\text{multiplications}} = \sum_{n=2}^N L_{n-1} \times L_n$$



How many resources?
DSPs, LUTs, FFs?
Does the model fit in the
latency requirements?

8
5 outputs
activation: SoftMax

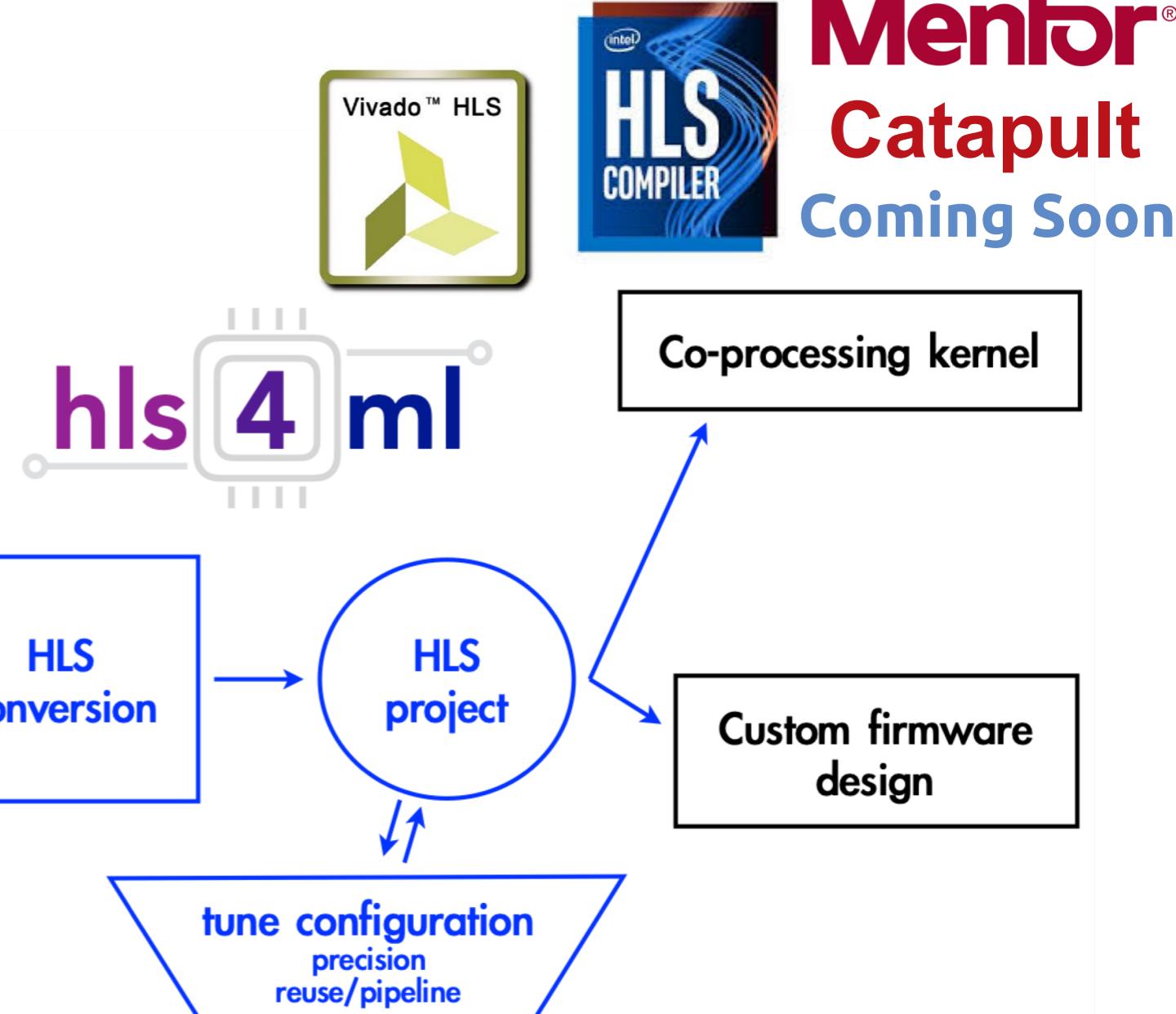
high level synthesis for machine learning



PYTORCH

 ONNX

Keras
TensorFlow
PyTorch
...



<https://hls-fpga-machine-learning.github.io/hls4ml/>

Efficient NN design for FPGAs

FPGAs provide huge flexibility

Performance depends on how well you take advantage of this

Constraints:

Input bandwidth
FPGA resources
Latency

Today you will learn how to optimize your project through:

- **compression:** reduce number of synapses or neurons
- **quantization:** reduces the precision of the calculations (inputs, weights, biases)
- **parallelization:** tune how much to parallelize to make the inference faster/slower versus FPGA resources

NN TRAINING

FPGA PROJECT
DESIGNING

Today's hls4ml hands on

- Part 1:

- Get started with hls4ml: train a basic model and run the conversion, simulation & c-synthesis steps

- Part 2:

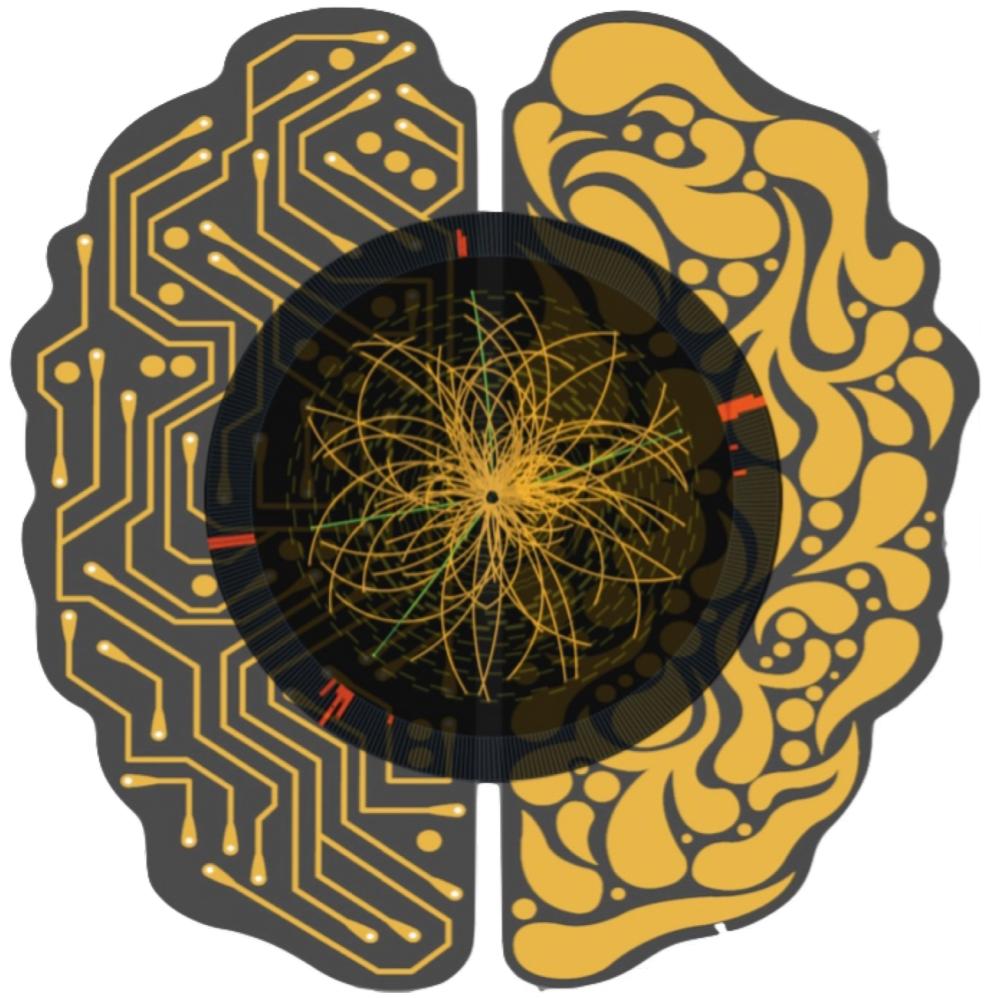
- Learn how to tune inference performance with quantization & ReuseFactor

- 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



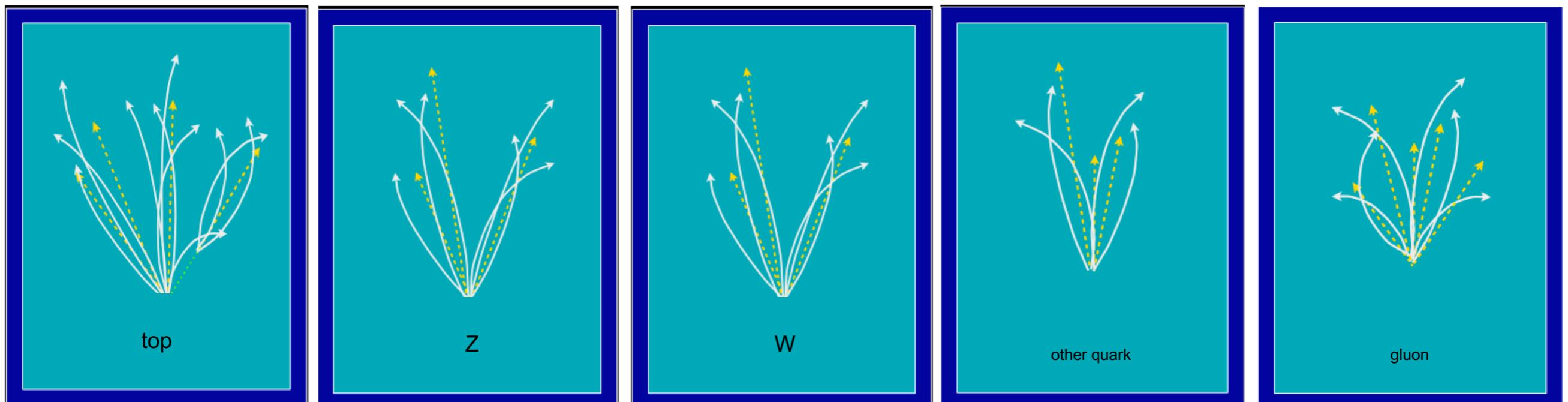
hls4ml tutorial

Part 1: Model Conversion

Physics case: 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



$t \rightarrow bW \rightarrow bqq$

3-prong jet

$Z \rightarrow qq$

2-prong jet

$W \rightarrow qq$

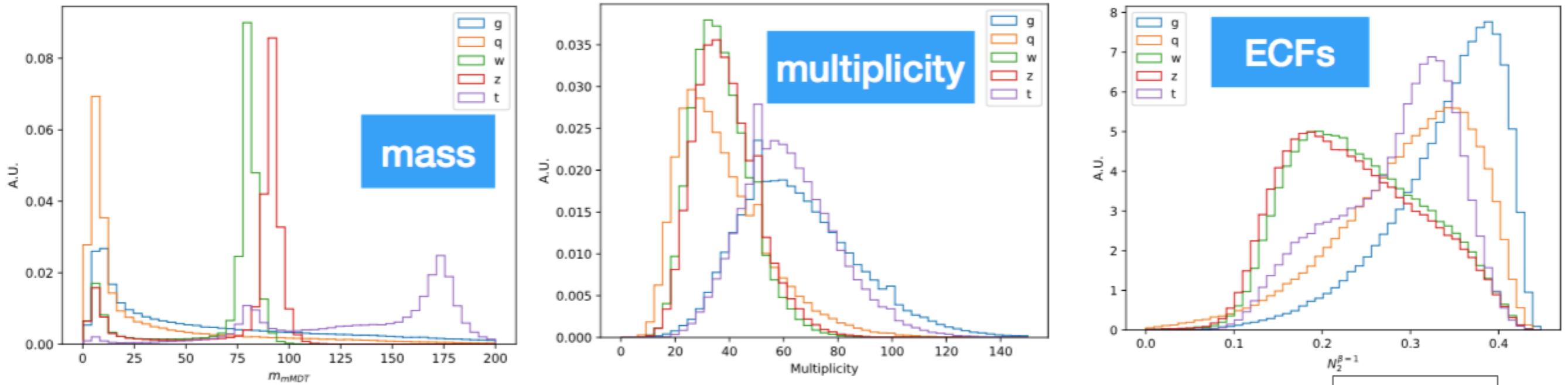
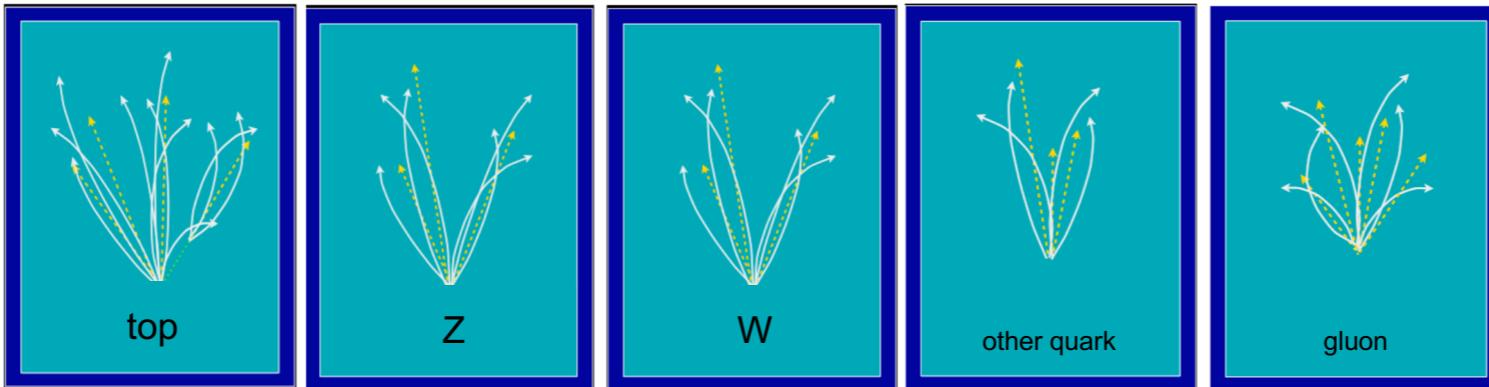
2-prong jet

q/g background

no substructure
and/or mass ~ 0

Reconstructed as one massive jet with substructure

Physics case: jet tagging



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

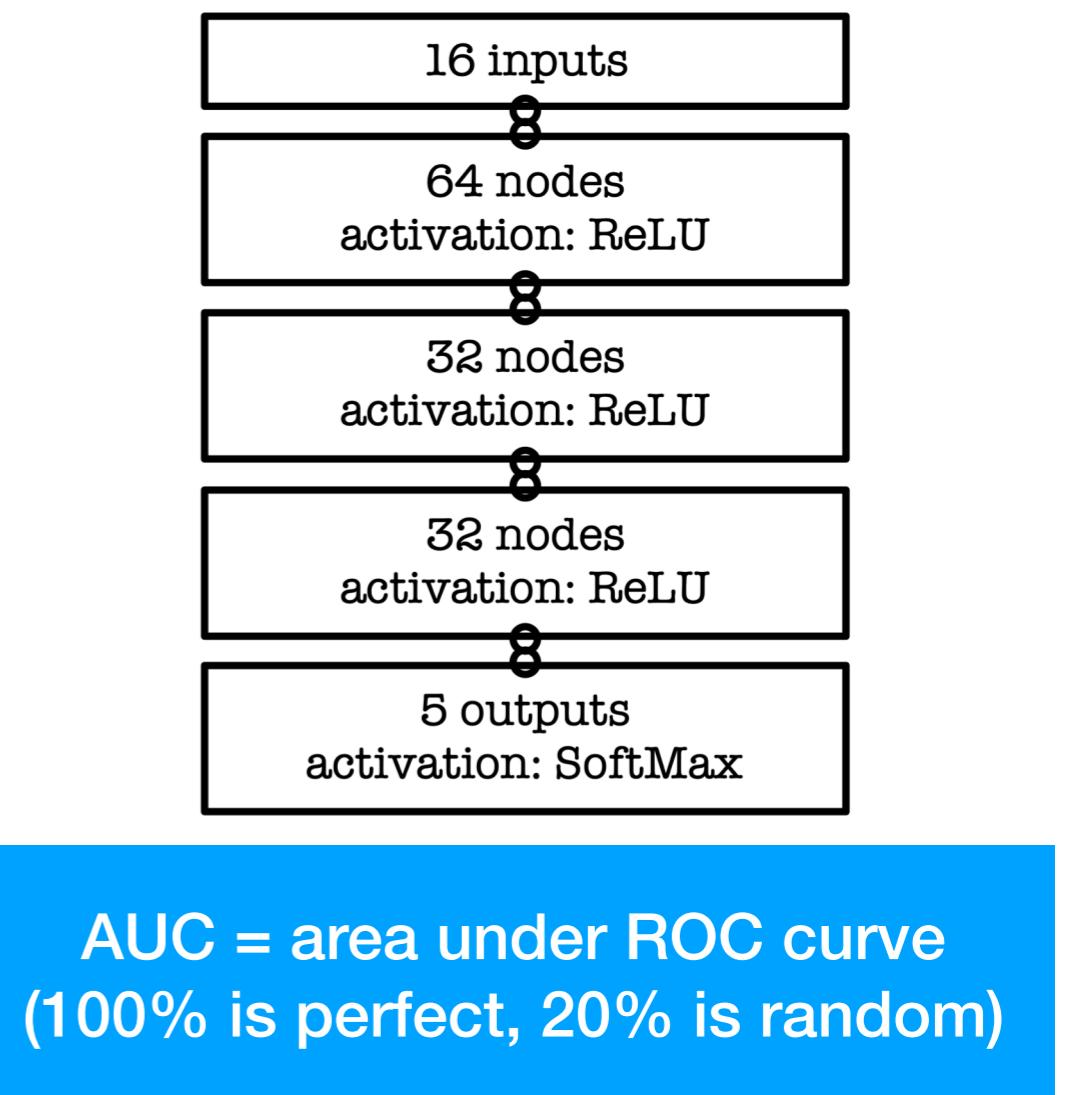
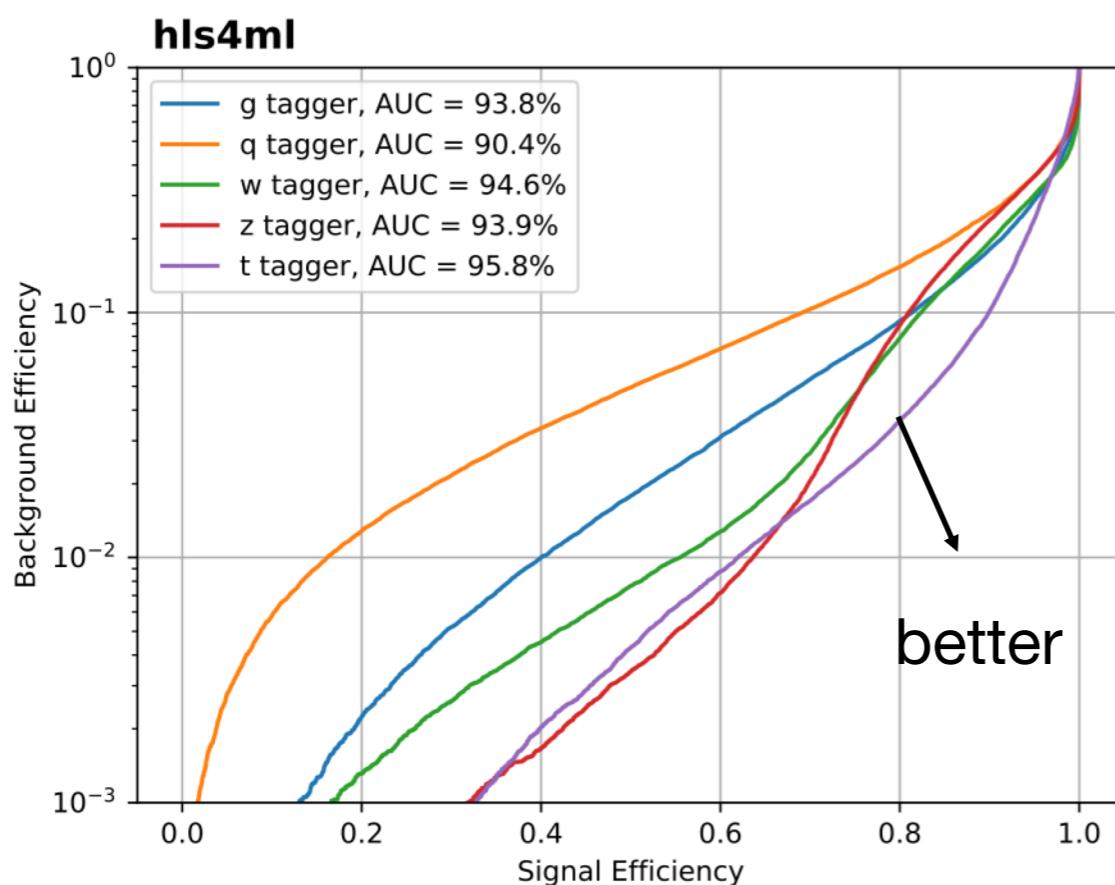
[*] D. Guest et al. [PhysRevD.94.112002](#), G. Kasieczka et al. [JHEP05\(2017\)006](#),
J. M. Butterworth et al. [PhysRevLett.100.242001](#), etc..

m_{mMDT}
 $N_2^{\beta=1,2}$
 $M_2^{\beta=1,2}$
 $C_1^{\beta=0,1,2}$
 $C_2^{\beta=1,2}$
 $D_2^{\beta=1,2}$
 $D_2^{(\alpha,\beta)=(1,1),(1,2)}$
 $\sum z \log z$
Multiplicity

Physics case: jet tagging

- 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 expert-level inputs**:

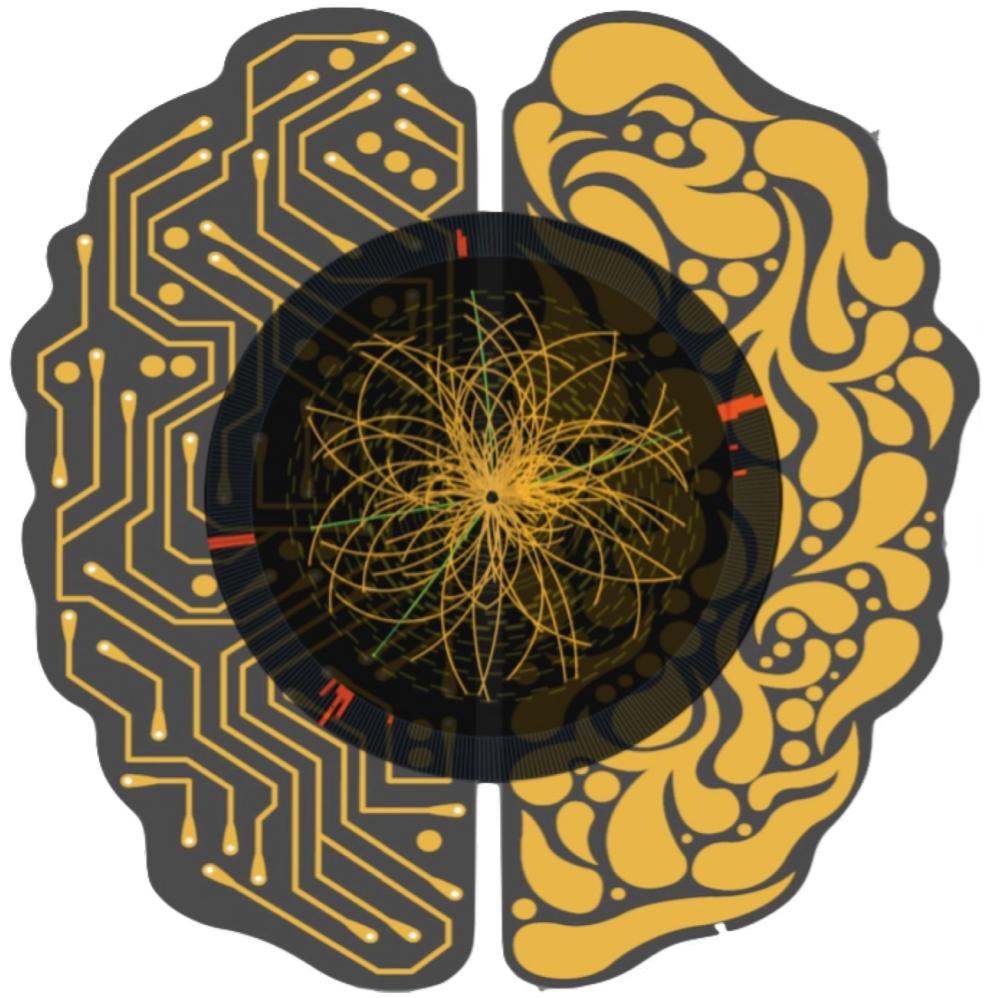
- Relu activation function for intermediate layers
- Softmax activation function for output layer



Hands On - Setup

- The interactive part is served with Python notebooks
 - Open <http://cern.ch/ssummers/iml> in your browser
 - Authenticate with your Github account (login if necessary)
 - Open and start running through “part1_getting_started” !
 - If you’re new to Jupyter notebooks, select a cell and hit “shift + enter” to execute the code





hls4ml Tutorial

Part 2: Advanced Configuration

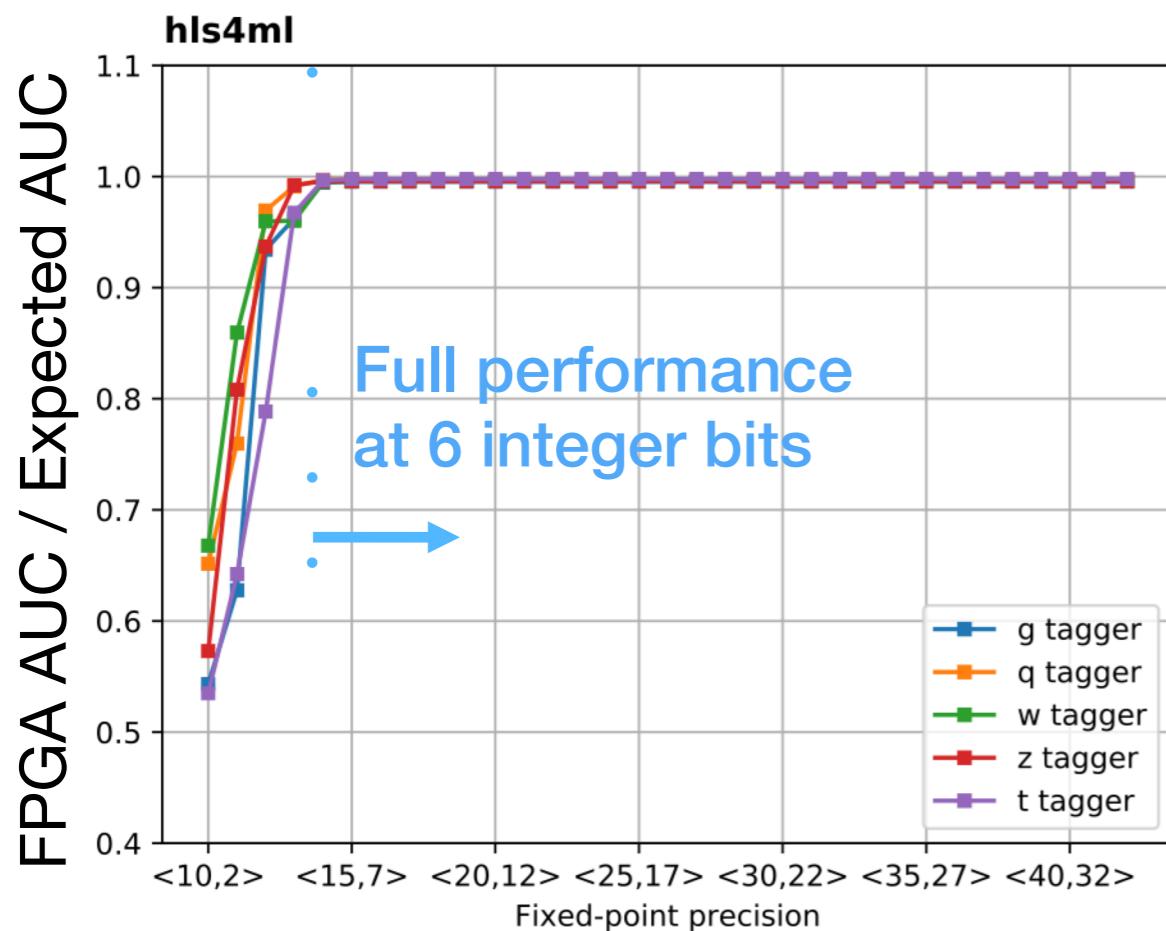
Efficient NN design: quantization

`ap_fixed<width bits, integer bits>`

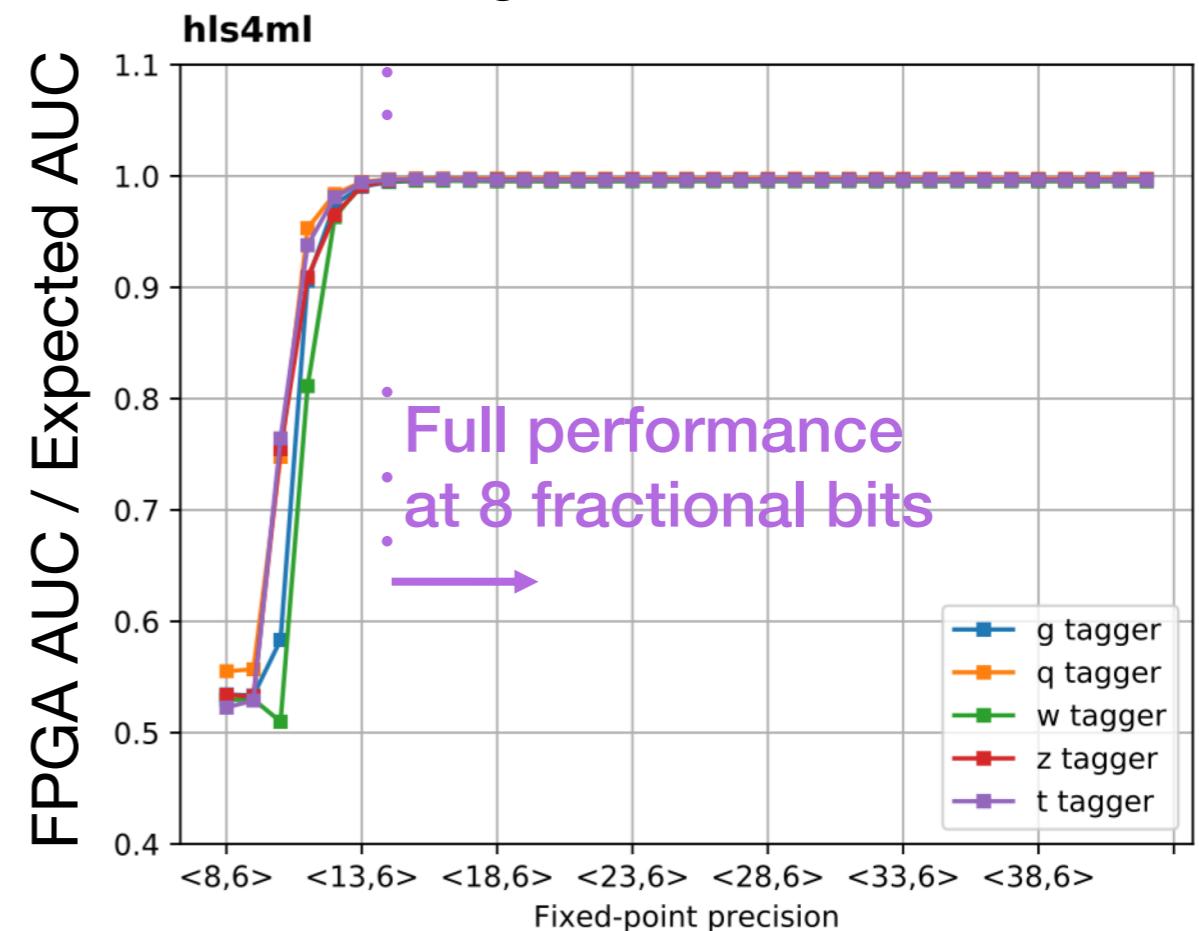
0101.1011101010
integer fractional
width

- 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

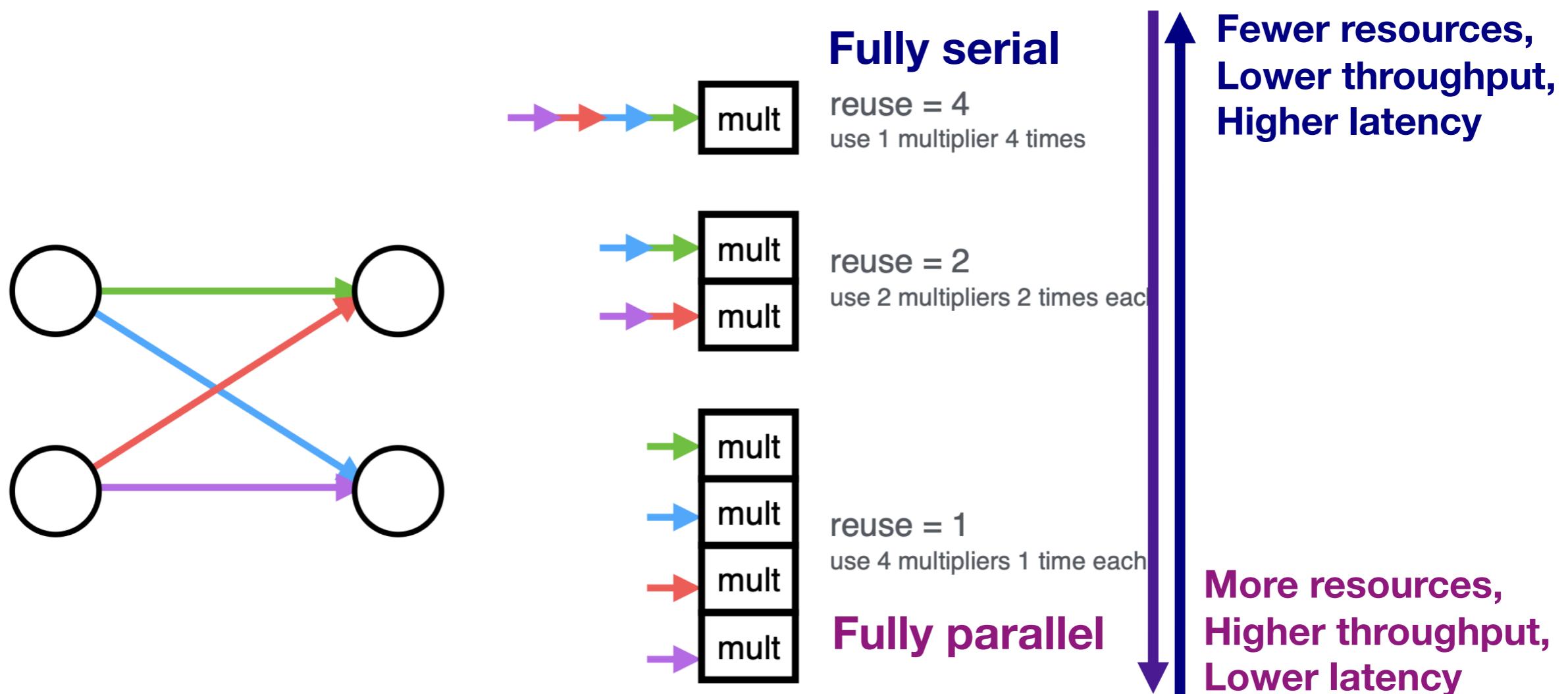


Scan fractional bits
Integer bits fixed to 6



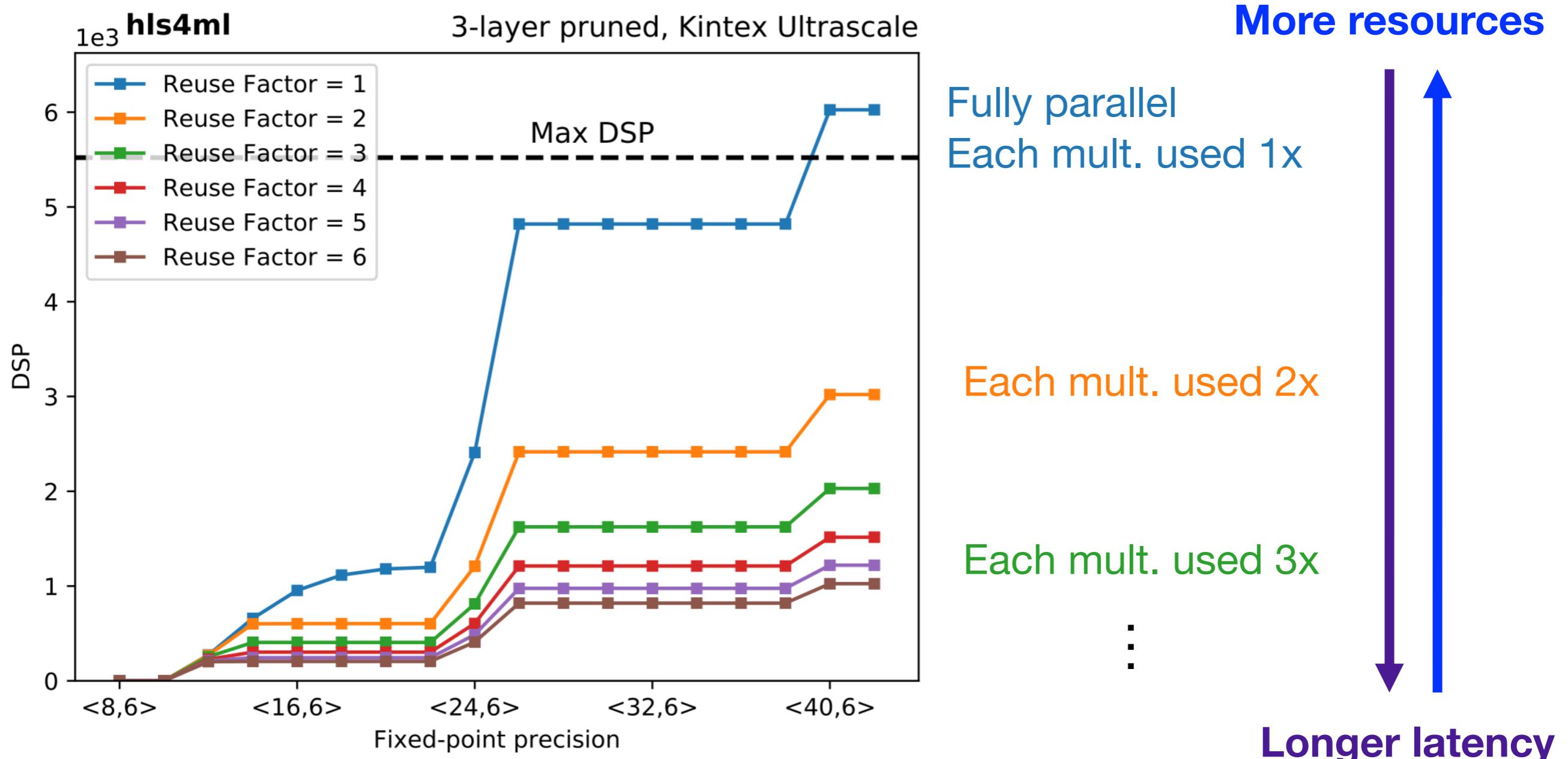
Efficient NN design: parallelization

- 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



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

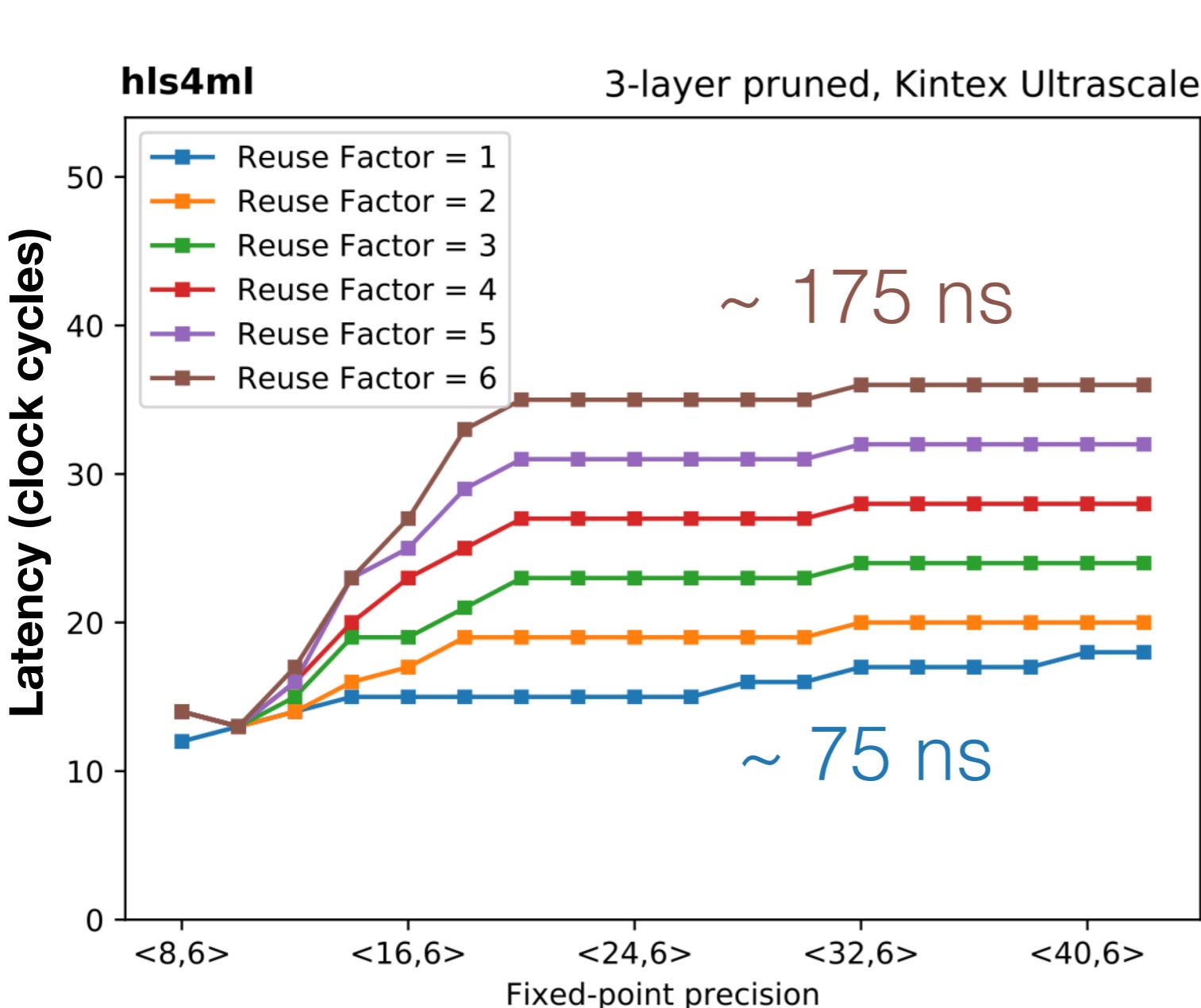
Parallelization: DSP usage



Parallelization: Timing

Latency of layer m

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



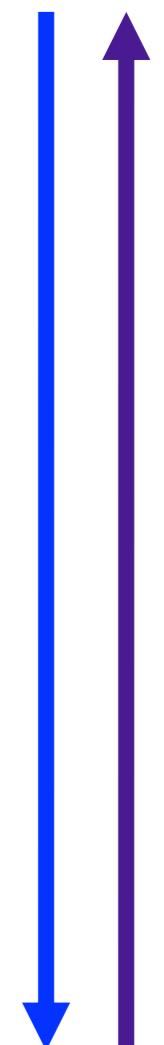
Longer latency

Each mult. used 6x

Each mult. used 3x

Fully parallel

Each mult. used 1x



More resources

Part 2: Large MLP

- ‘Strategy: Resource’ for larger networks and higher reuse factor
- Uses a slightly different HLS implementation of the dense layer to compile faster and better for large layers
- Here, we use a different partitioning on the first layer for the best partitioning of arrays

```
IOTYPE: io_parallel # options: io_serial/io_parallel  
HLSConfig:  
    Model:  
        Precision: ap_fixed<16,6>  
        ReuseFactor: 128  
    Strategy: Resource  
    LayerName:  
        dense1:  
            ReuseFactor: 112
```

This config is for a model trained on the MNIST digits classification dataset

Architecture (fully connected): $784 \rightarrow 128 \rightarrow 128 \rightarrow 128 \rightarrow 10$

Model accuracy: ~97%

We can work out how many DSPs this should use...

Part 2: Large MLP

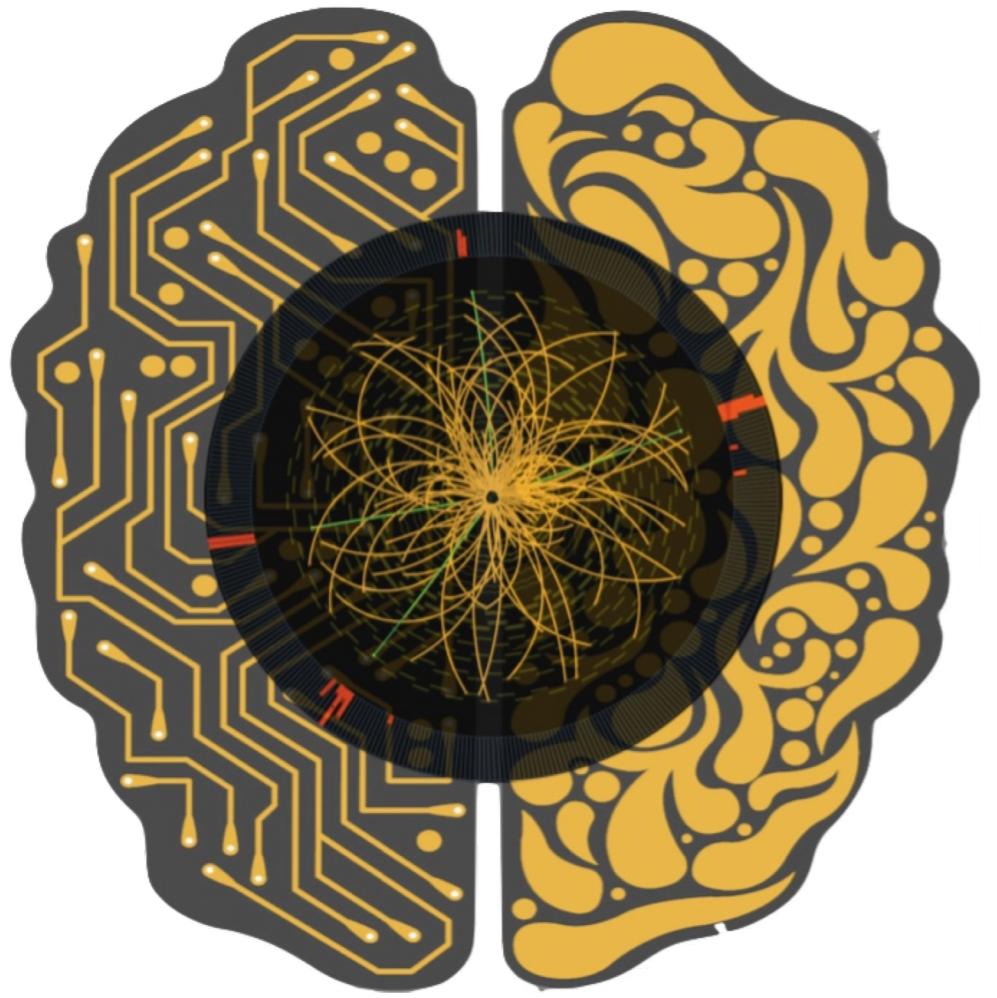
- It takes a while to synthesise, so here's one I made earlier...
- The DSPs should be: $(784 \times 128) / 112 + (2 \times 128 \times 128 + 128 \times 10) / 128 = 1162$ 🤓

```
=====
=====
+ Timing (ns):
  * Summary:
  +-----+-----+-----+
  | Clock | Target| Estimated| Uncertainty|
  +-----+-----+-----+
  | ap_clk | 5.00 | 4.375 | 0.62 |
  +-----+-----+-----+
+ Latency (clock cycles):
  * Summary:
  +-----+-----+-----+
  | Latency | Interval | Pipeline |
  | min | max | min | max | Type   |
  +-----+-----+-----+
  | 518 | 522 | 128 | 128 | dataflow |
  +-----+-----+-----+
```



```
=====
===== Utilization Estimates =====
=====
+-----+-----+-----+-----+
|       Name       | BRAM_18K | DSP48E | FF    | LUT   |
+-----+-----+-----+-----+
| ...           |          |          |        |        |
+-----+-----+-----+-----+
| Total         |          | 1962 | 1162 | 169979 | 222623 |
+-----+-----+-----+-----+
| Available SLR |          | 2160 | 2760 | 663360 | 331680 |
+-----+-----+-----+-----+
| Utilization SLR (%) | 90 | 42 | 25 | 67 |
+-----+-----+-----+-----+
| Available     |          | 4320 | 5520 | 1326720 | 663360 |
+-----+-----+-----+-----+
| Utilization (%) | 45 | 21 | 12 | 33 |
+-----+-----+-----+-----+
```

II determined by the largest reuse factor



hls4ml Tutorial

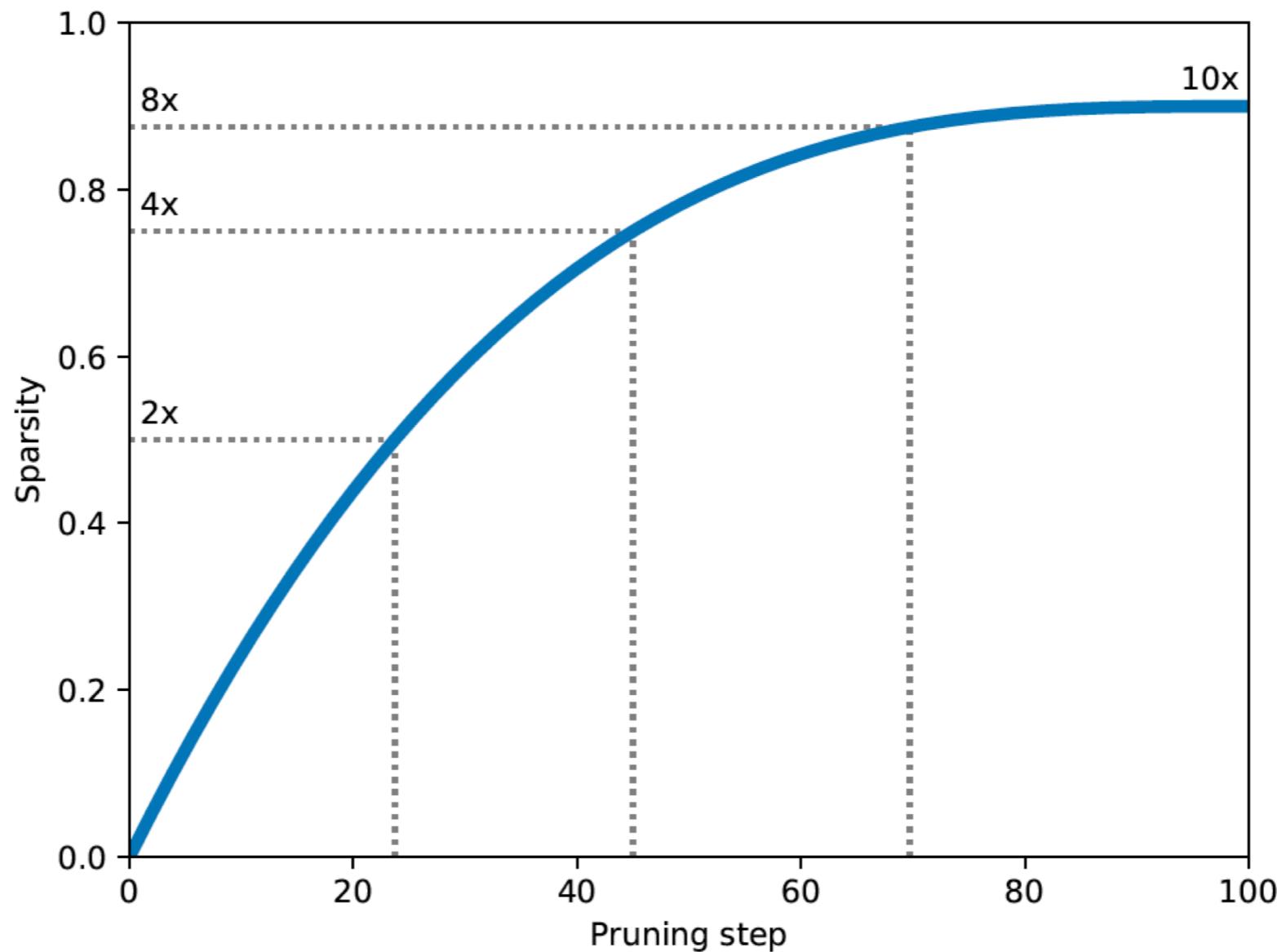
Part 3: Compression

NN compression methods

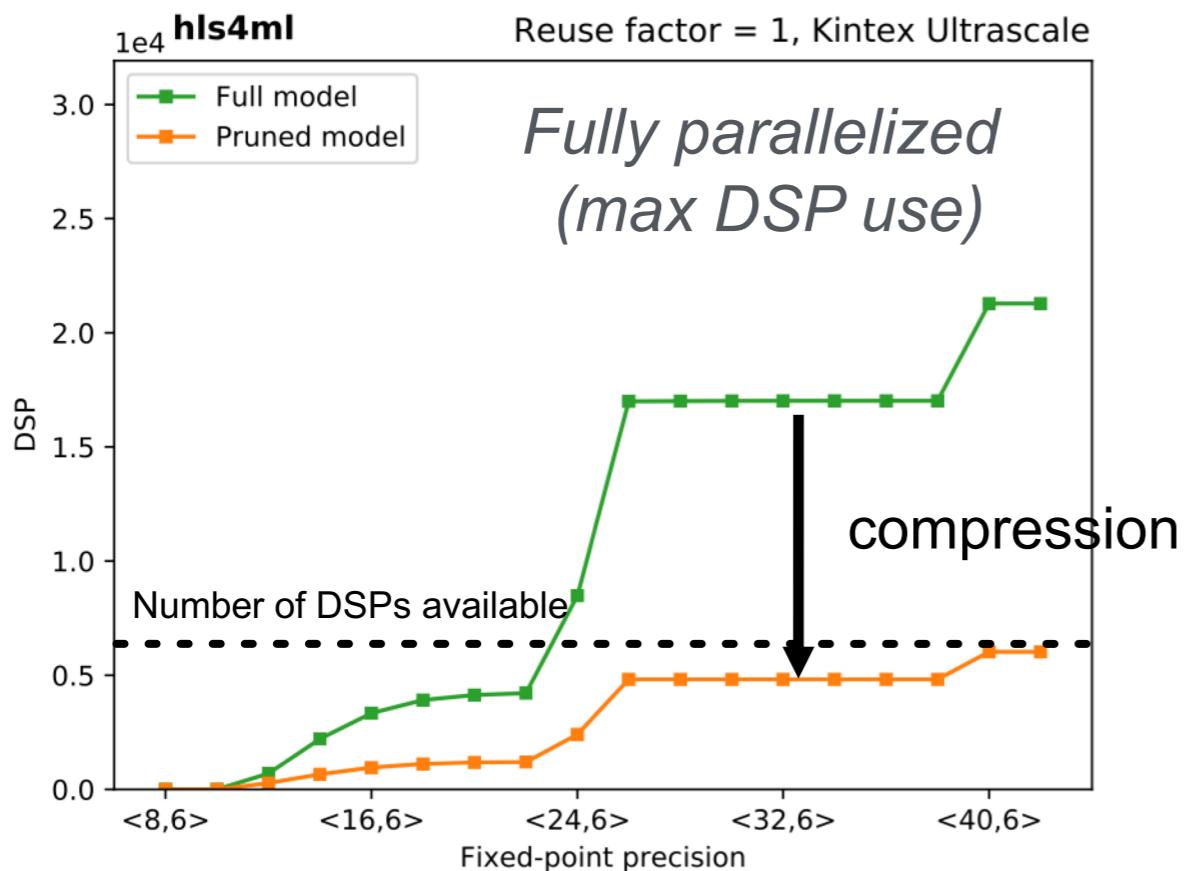
- Network compression is a widespread technique to reduce the size, energy consumption, and overtraining of deep neural networks
- Several approaches have been studied:
 - **parameter pruning:** selective removal of weights based on a particular ranking
[\[arxiv.1510.00149\]](https://arxiv.org/abs/1510.00149), [\[arxiv.1712.01312\]](https://arxiv.org/abs/1712.01312)
 - **low-rank factorization:** using matrix/tensor decomposition to estimate informative parameters [\[arxiv.1405.3866\]](https://arxiv.org/abs/1405.3866)
 - **transferred/compact convolutional filters:** special structural convolutional filters to save parameters [\[arxiv.1602.07576\]](https://arxiv.org/abs/1602.07576)
 - **knowledge distillation:** training a compact network with distilled knowledge of a large network [\[doi:10.1145/1150402.1150464\]](https://doi.org/10.1145/1150402.1150464)
- Today we'll use the tensorflow model sparsity toolkit
 - <https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html>
- But you can use other methods!

TF Sparsity

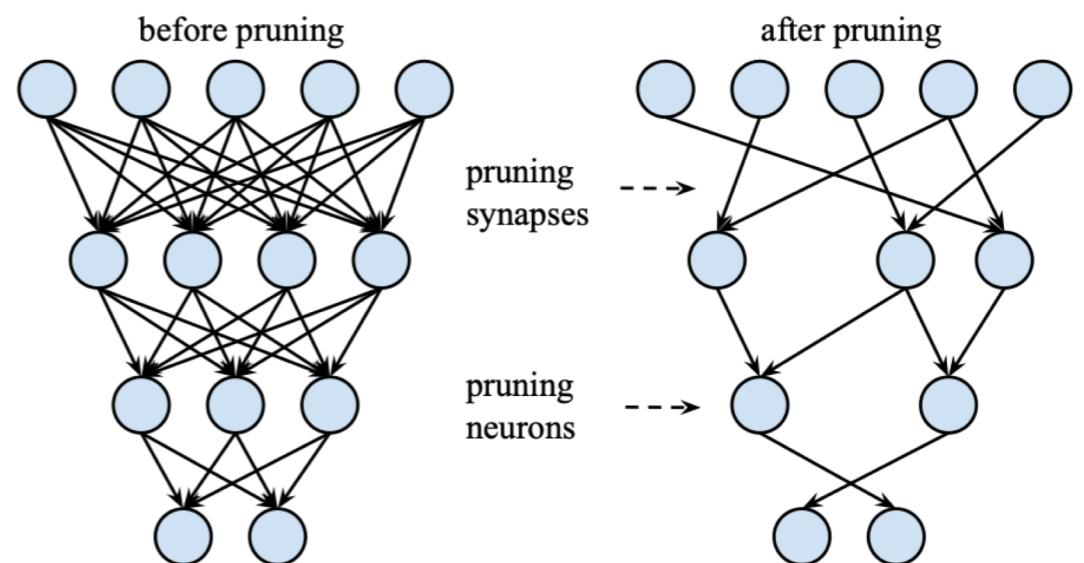
- Iteratively remove low magnitude weights, starting with 0 sparsity, smoothly increasing up to the set target as training proceeds



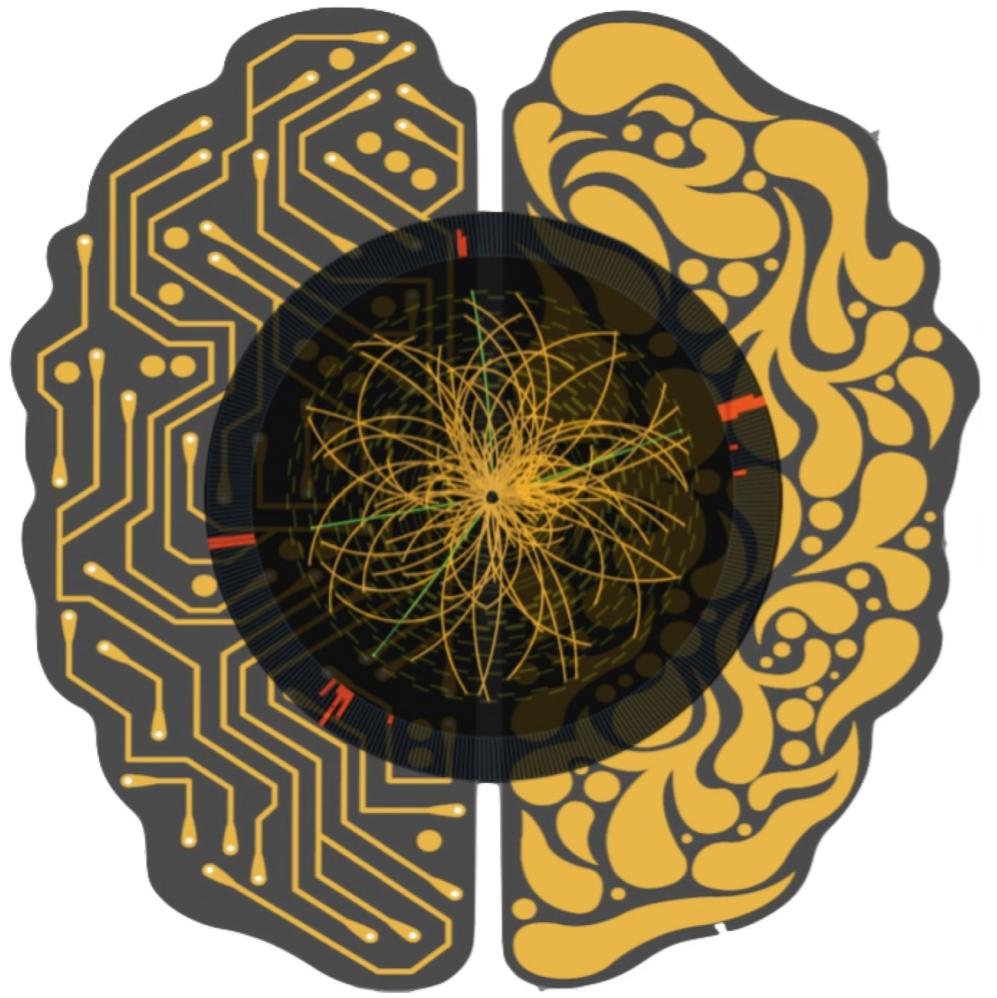
Efficient NN design: compression



70% compression \sim 70% fewer DSPs



- DSPs (used for multiplication) are often 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

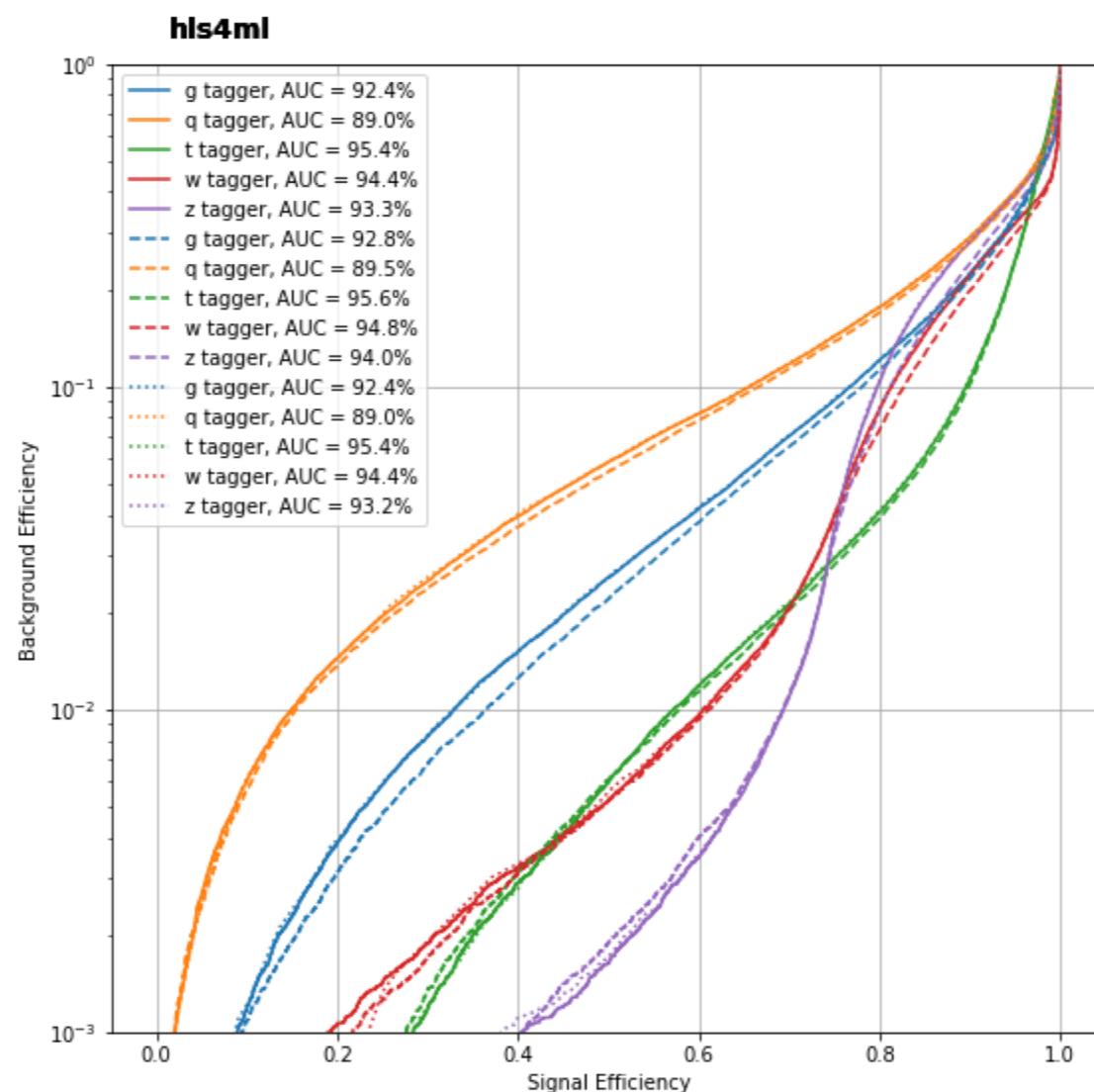


hls4ml Tutorial

Part 4: Quantization

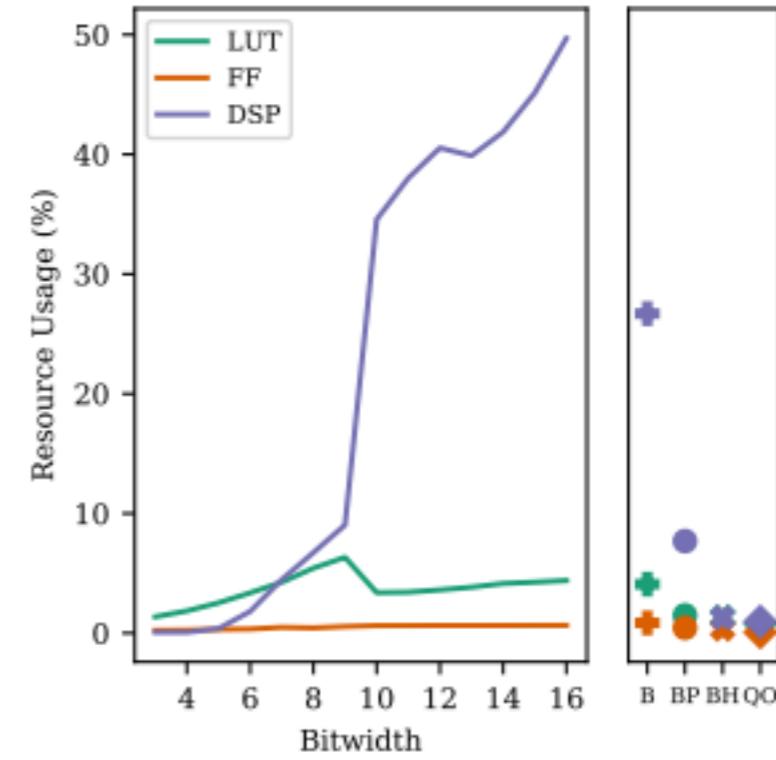
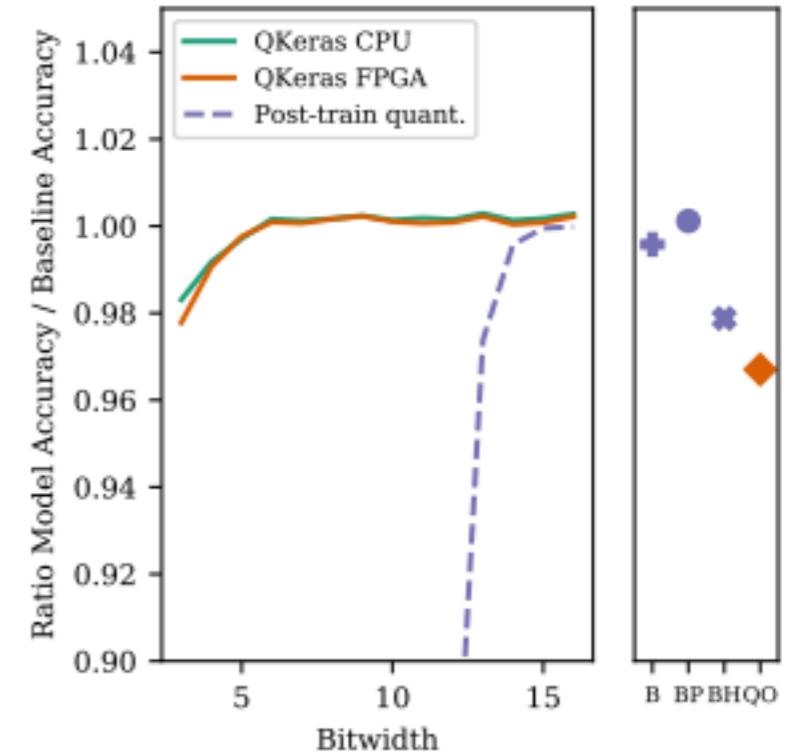
Efficient NN design: quantization

- hls4ml allows you to use different data types everywhere, we will learn how to use that
- We will also try quantization-aware training with QKeras (part 4)
- With quantization-aware we can even go down to just 1 or 2 bits
 - See our recent work: <https://arxiv.org/abs/2003.06308>



QKeras

- QKeras is a library to train models with quantization in the training
 - Maintained by Google
- Easy to use, drop-in replacements for Keras layers
 - e.g. Dense → QDense
 - e.g. Conv2D → QConv2D
 - Use ‘quantizers’ to specify how many bits to use where
 - Same kind of granularity as hls4ml
- Can achieve good performance with very few bits
- We’ve recently added support for QKeras-trained models to hls4ml
 - The number of bits used in training is also used in inference
 - The intermediate model is adjusted to capture all optimizations possible with QKeras



Summary

- After this session you've gained some hands on experience with **hls4ml**
 - Translated neural networks to FPGA firmware, run simulation and synthesis
- Tuned network inference performance with precision and ReuseFactor
 - Used profiling and trace tools to guide tuning
- Seen how to simply prune a neural network and the impact on resources
- Trained a model with small number of bits using QKeras, and use the same spec in inference easily with **hls4ml**
- You can find these tutorial notebooks to run yourself: <https://github.com/hls-fpga-machine-learning/hls4ml-tutorial>
 - (FPL2020 branch)
- You can run the tutorial Docker image yourself like:
 - docker run -p 8888:8888 gitlab-registry.cern.ch/ssummers/hls4ml-tutorial:9
 - No FPGA tools on this one!
- Use hls4ml in your own environment: `pip install hls4ml [profiling]`