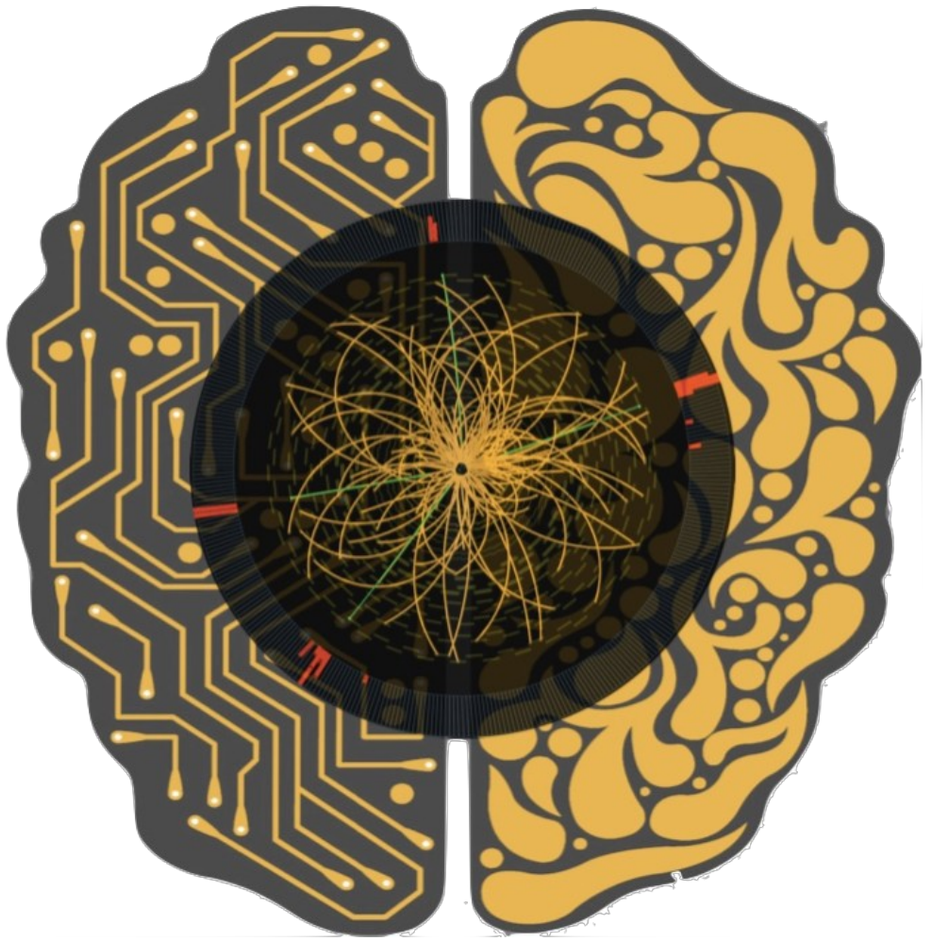


hls4ml tutorial

Thea Aarrestad, Vladimir Loncar,
Jennifer Ngadiuba, Maurizio Pierini
Sioni Summers

Introduction

- hls4ml is a package for executing neural network inference with extremely low latency on FPGAs
- In this session you will get hands on experience with the hls4ml package
- Translate pre-trained models into 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
- But first…

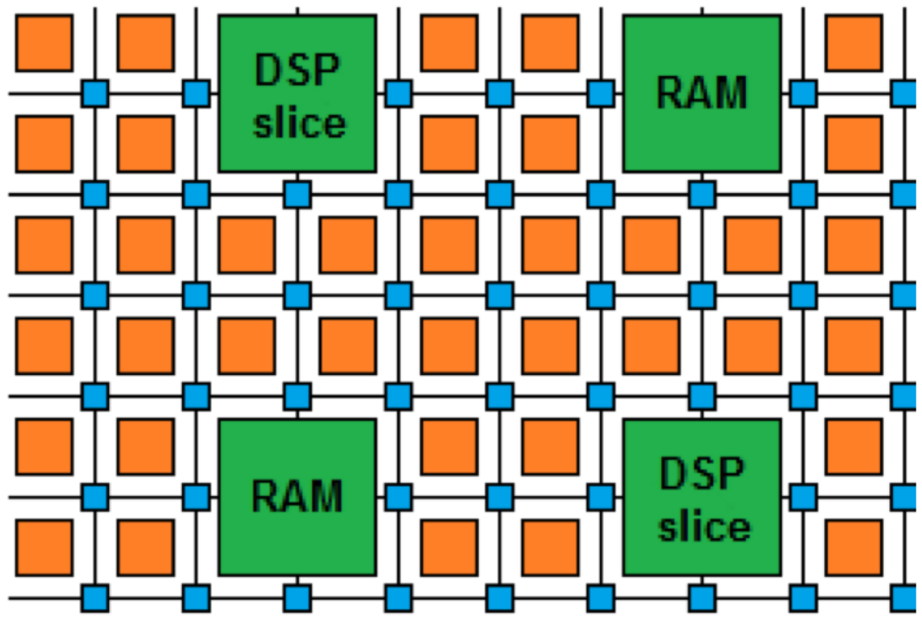
What are FPGAs?

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?

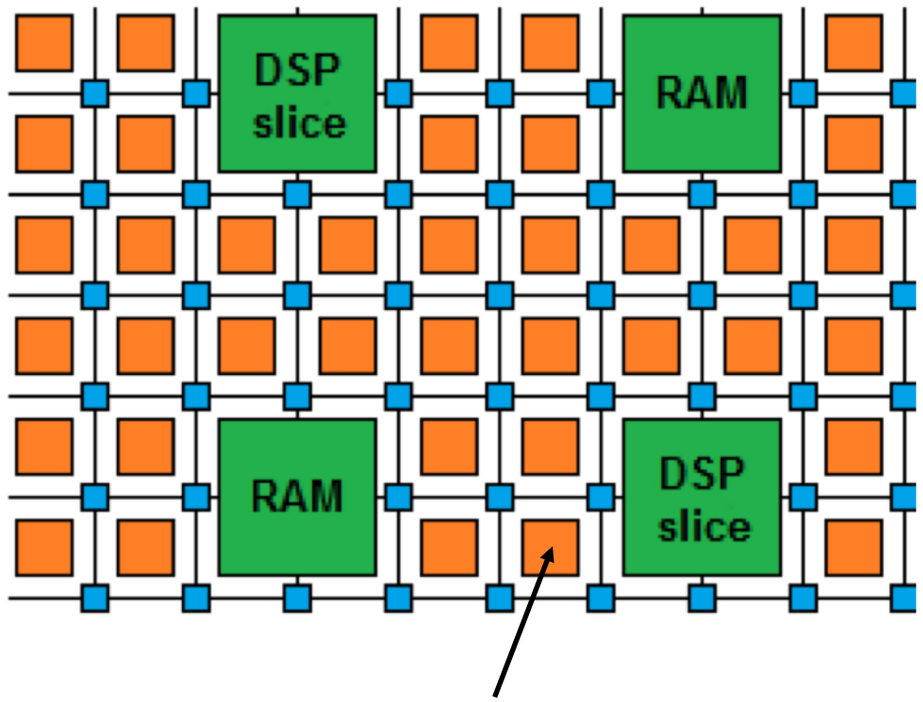
Field Programmable Gate Arrays are reprogrammable integrated circuits

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

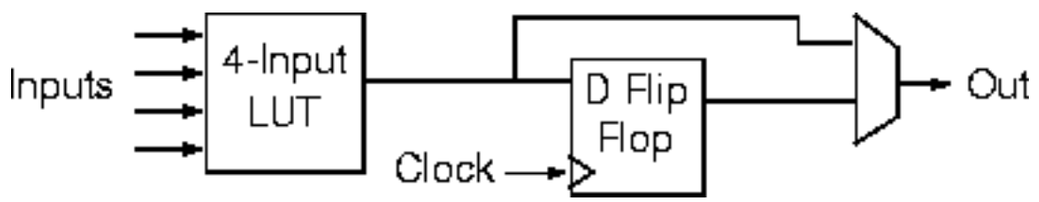
These can be used for boolean operations, arithmetic, small memories

Flip-Flops register data in time with the clock pulse

FPGA diagram



Logic cell



Look-up table

(logic)

Flip-flop

(registers)

What are FPGAs?

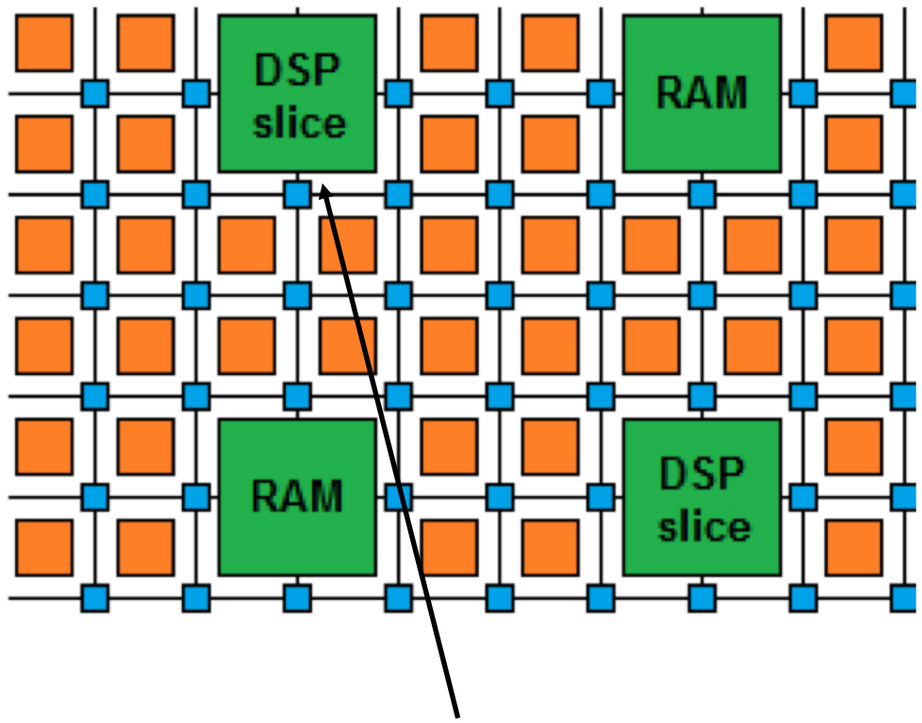
Field Programmable Gate Arrays are reprogrammable integrated circuits

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

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

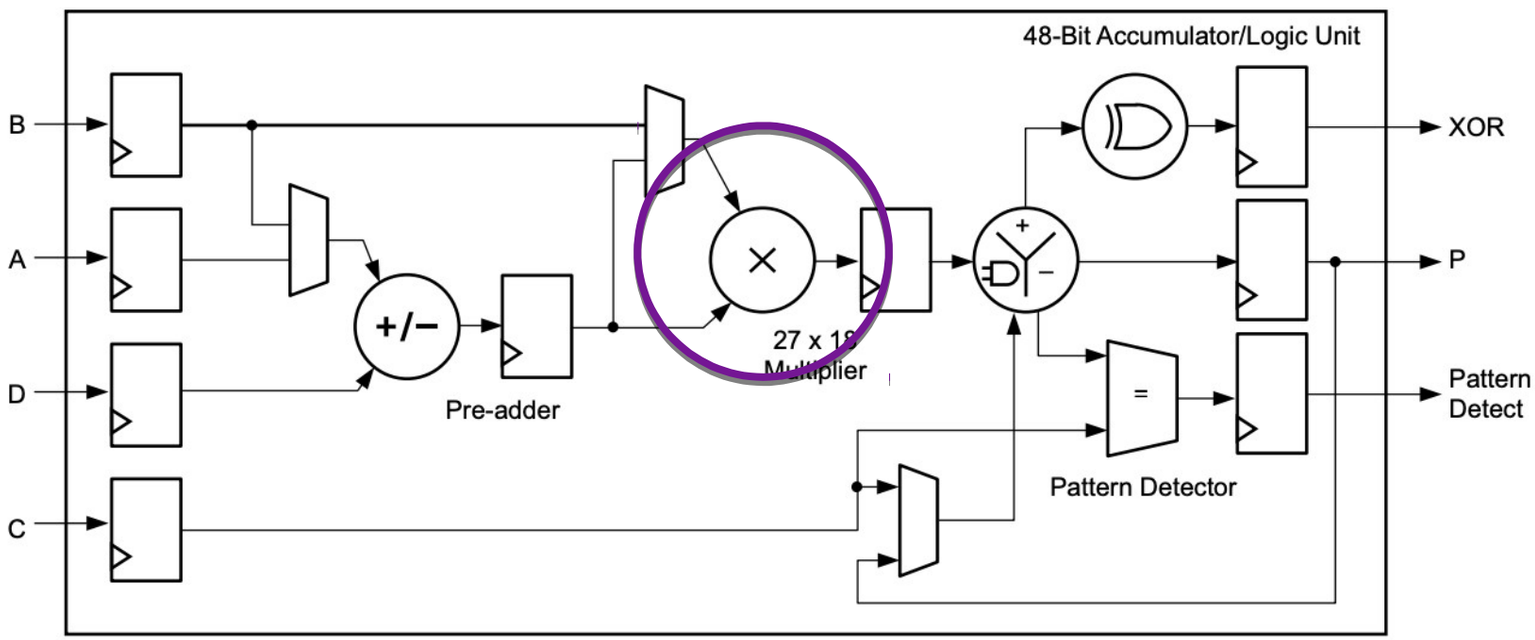
And for Neural Nets, DSPs are often the most scarce

FPGA diagram



DSP

(mutliplication)



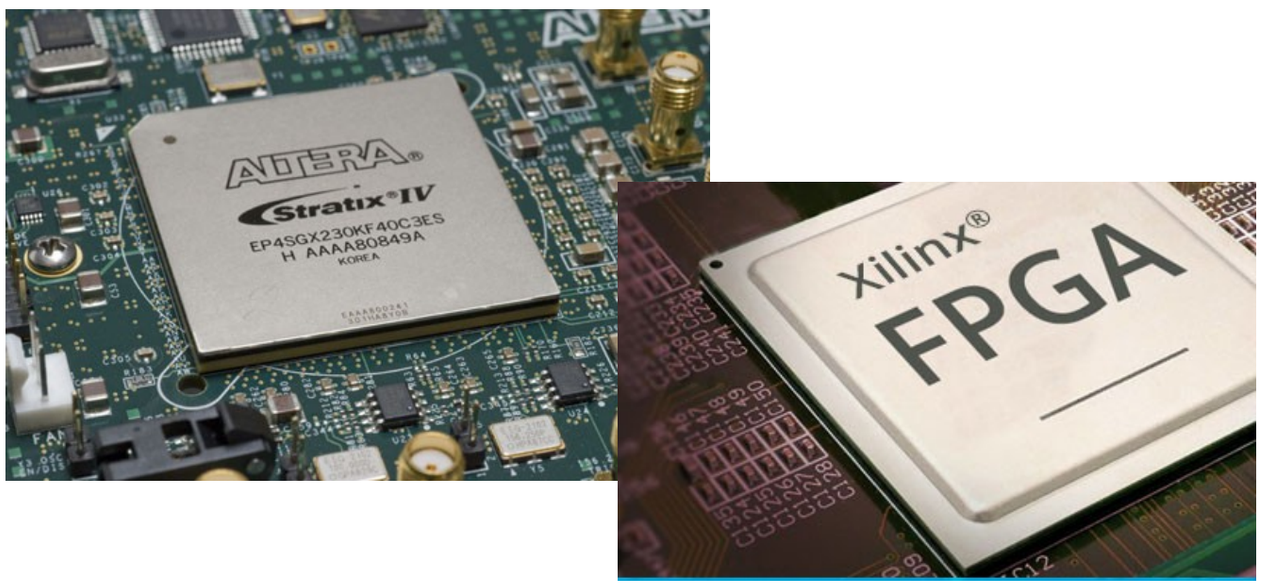
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

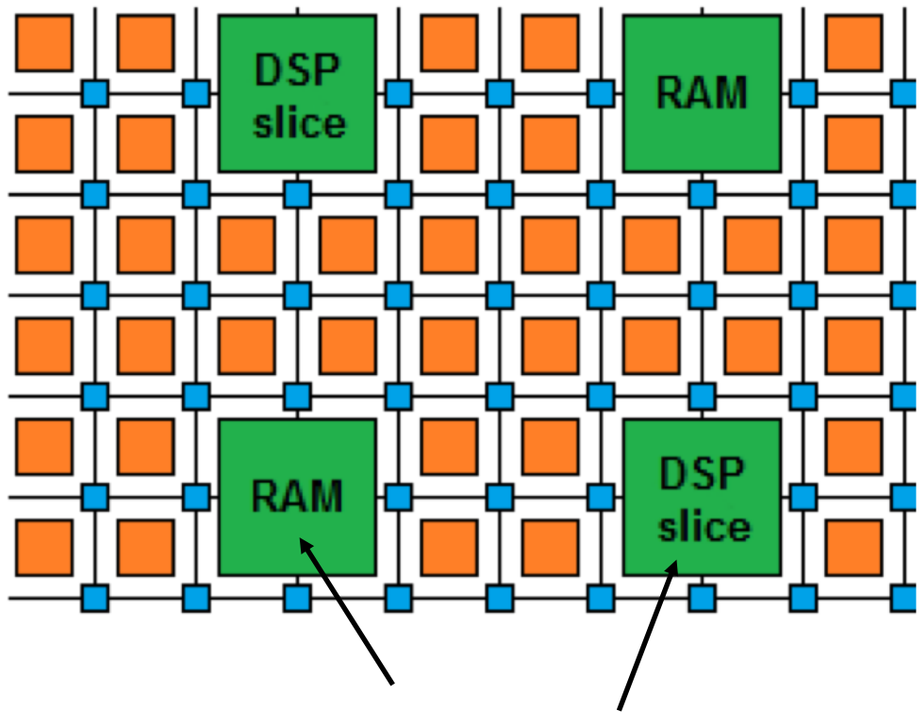
BRAMs are small, fast memories - RAMs, ROMs, FIFOs (18Kb each in Xilinx)

Again, memories using BRAMs are more efficient than using LUTs

A big FPGA has nearly 100Mb of BRAM, chained together as needed



FPGA diagram



Also contain embedded components:

Digital Signal Processors (DSPs):
logic units used for multiplications

Random-access memories (RAMs):
embedded memory elements

What are FPGAs?

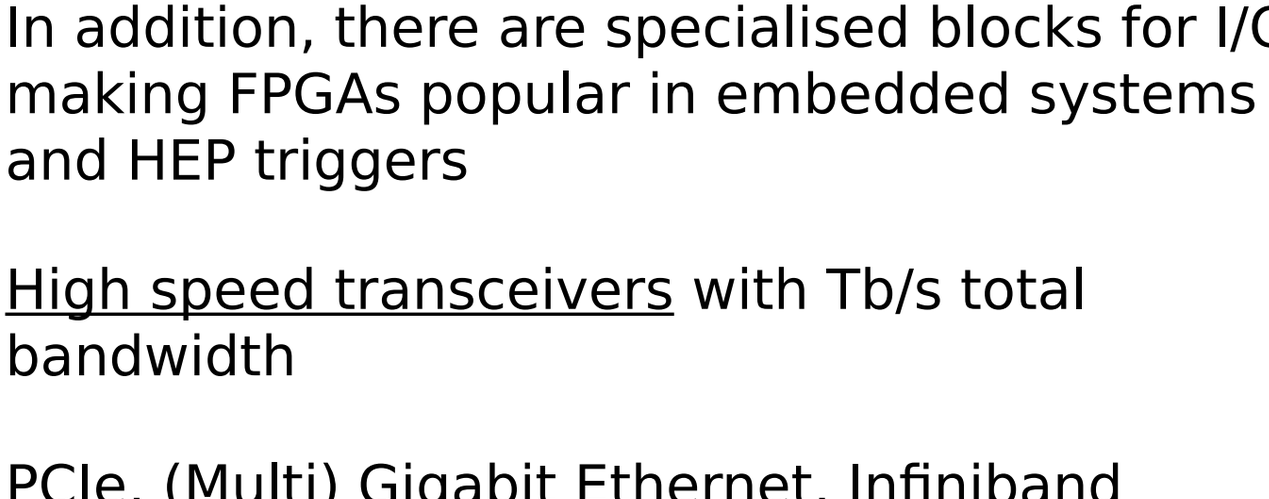
In addition, there are specialised blocks for I/O, making FPGAs popular in embedded systems and HEP triggers

<u>High speed transceivers</u> with Tb/s total bandwidth

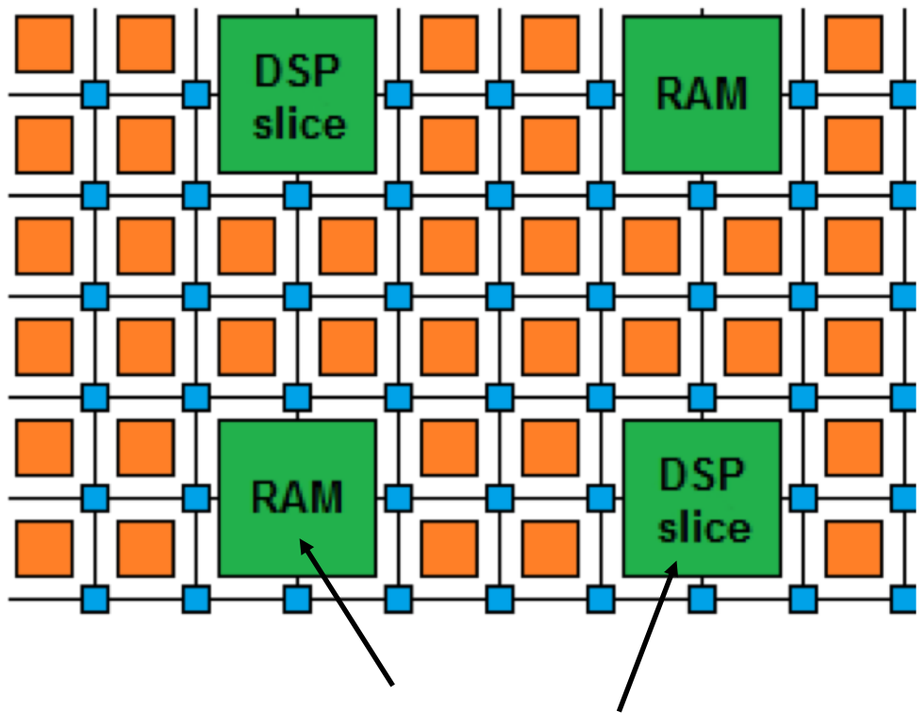
PCIe, (Multi) Gigabit Ethernet, Infiniband

<u>AND:</u> Support <u>highly parallel</u> algorithm implementations

<u>Low power per Op</u> (relative to CPU/GPU)



FPGA diagram



Digital Signal Processors (DSPs):
logic units used for multiplications

Random-access memories (RAMs):
embedded memory elements

Flip-flops (FF) and look up tables (LUTs) for additions

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…

How are FPGAs programmed?

Hardware Description Languages

HDLs are programming languages which describe electronic circuits

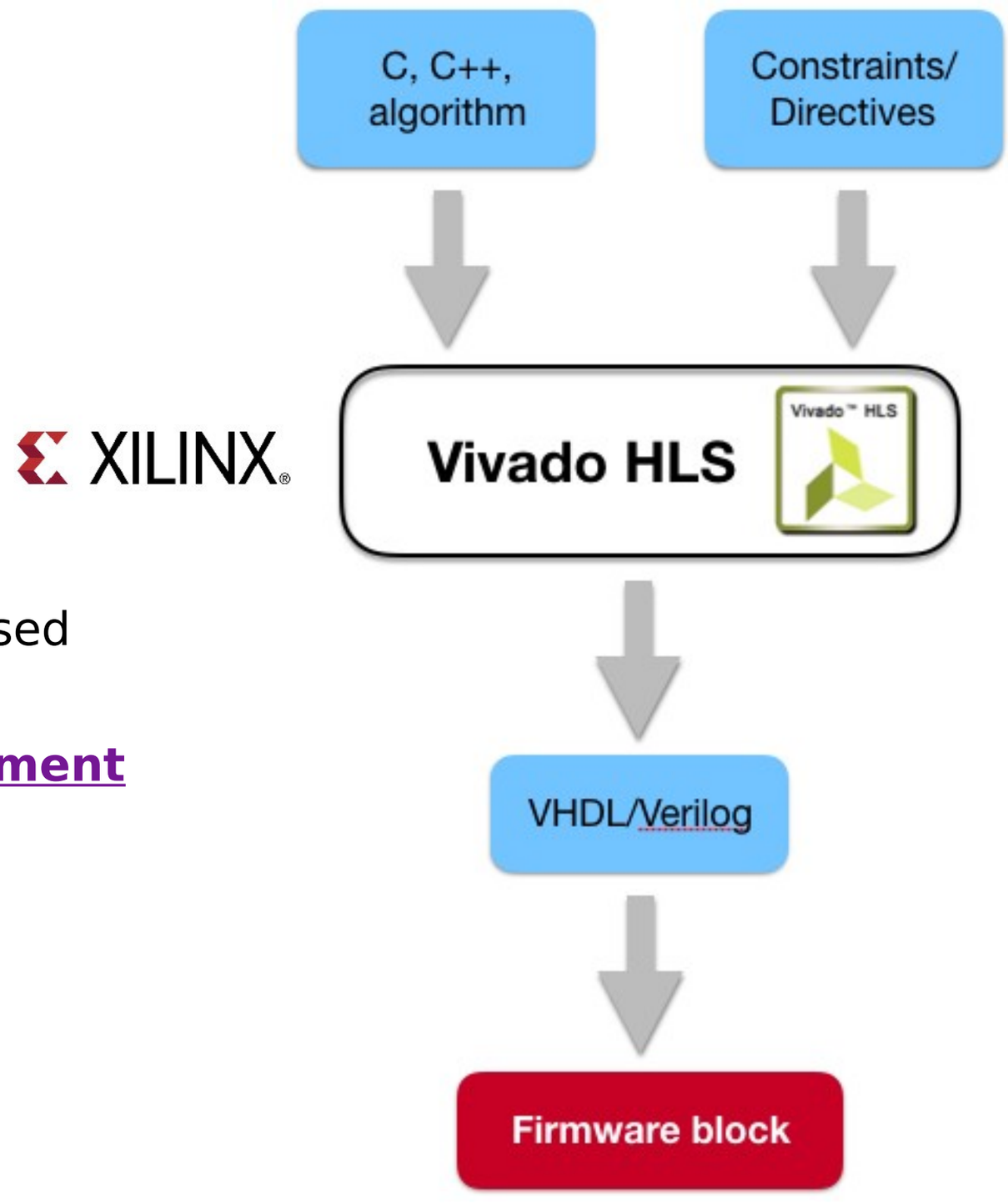
High Level Synthesis

Compile from C/C++ to VHDL

Pre-processor directives and constraints used to optimize the design

Drastic decrease in firmware development time!

Today we'll use Xilinx Vivado HLS [*

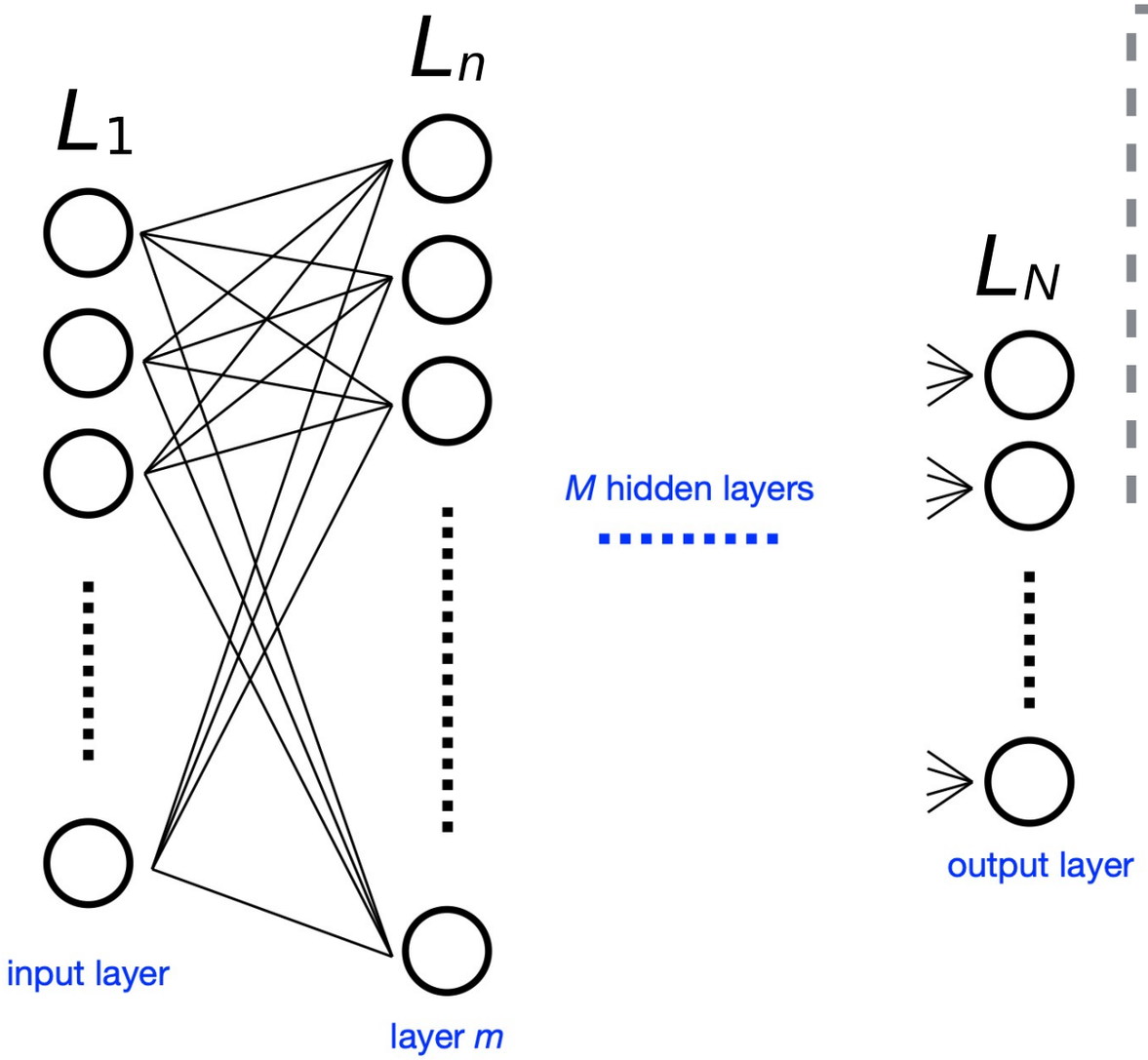


[*] *https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf*

Jargon

- LUT - Look Up Table aka ‘logic’ - generic functions on small bitwidth inputs. Combine many to build the algorithm
- FF - Flip Flops - control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- DSP - Digital Signal Processor - performs multiplication and other arithmetic in the FPGA
- BRAM - Block RAM - hardened RAM resource. More efficient memories than using LUTs for more than a few elements
- HLS - High Level Synthesis - compiler for C, C++, SystemC into FPGA IP cores
- HDL - Hardware Description Language - low level language for describing circuits
- RTL - Register Transfer Level - the very low level description of the function and connection of logic gates
- Latency - time between starting processing and receiving the result
  - Measured in clock cycles or seconds
- Initiation Interval - time from accepting first input to accepting next input

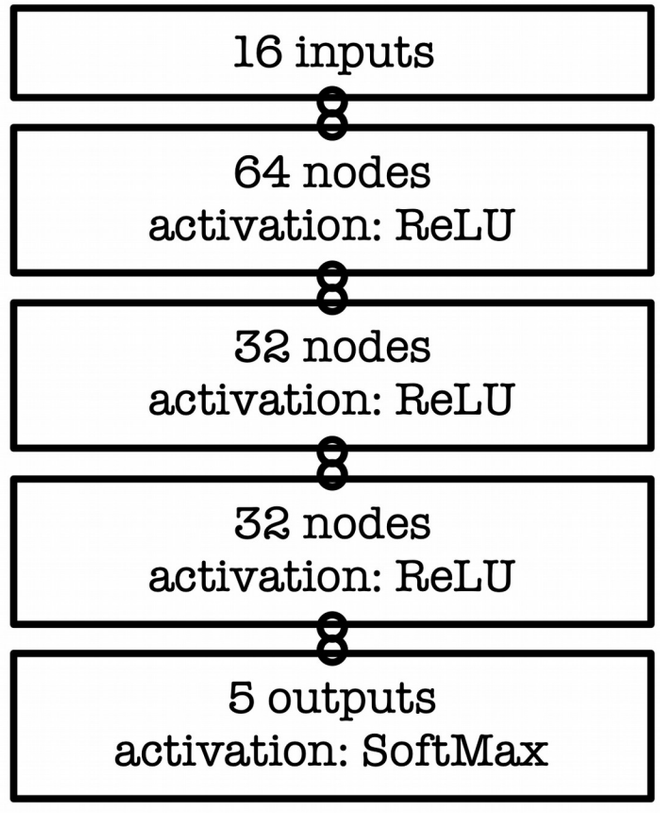
Neural network inference



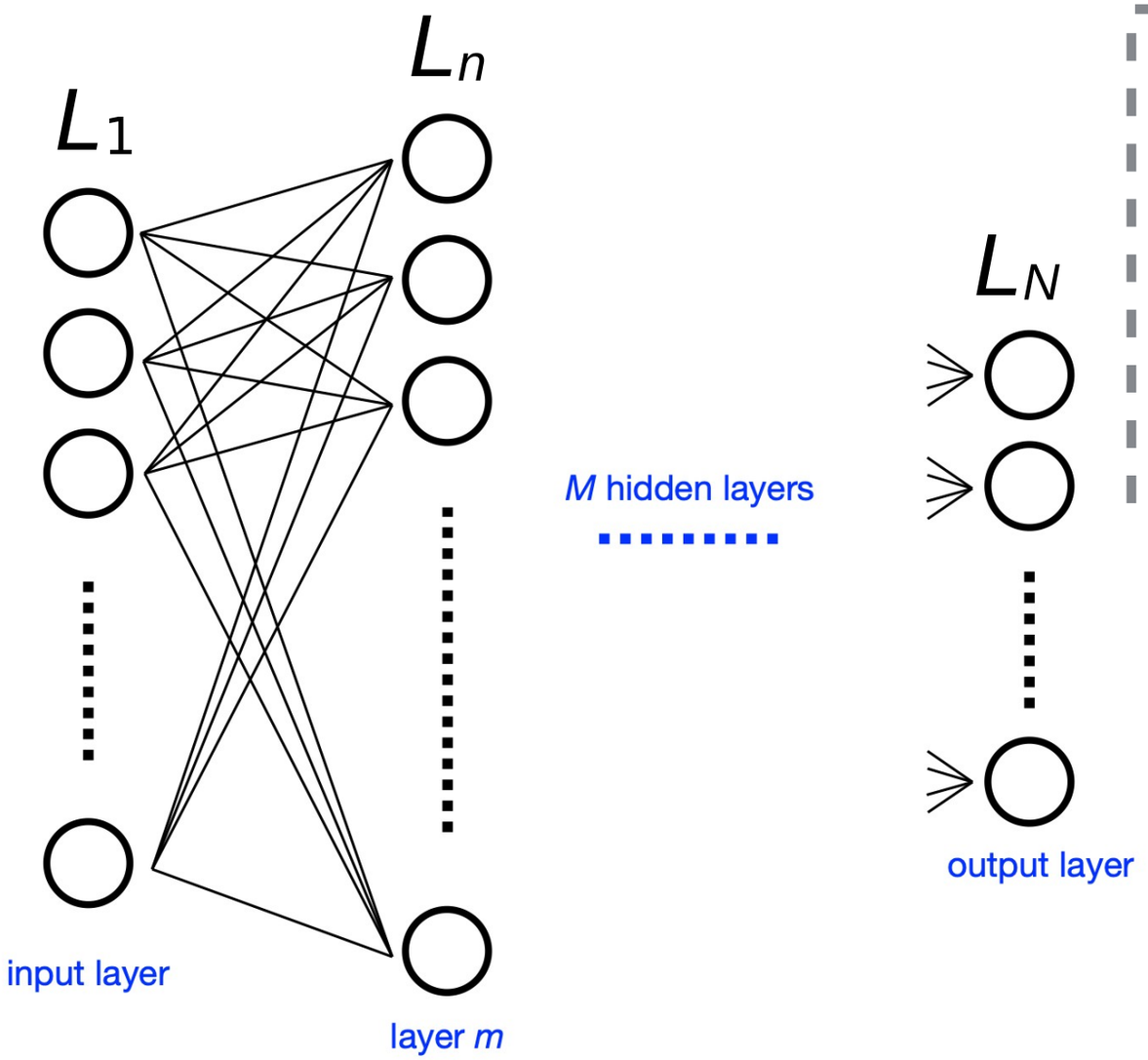
$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1} + \mathbf{b}_n)$$

activation function
precomputed and stored in BRAMs
multiplication
DSPs
addition
logic cells

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

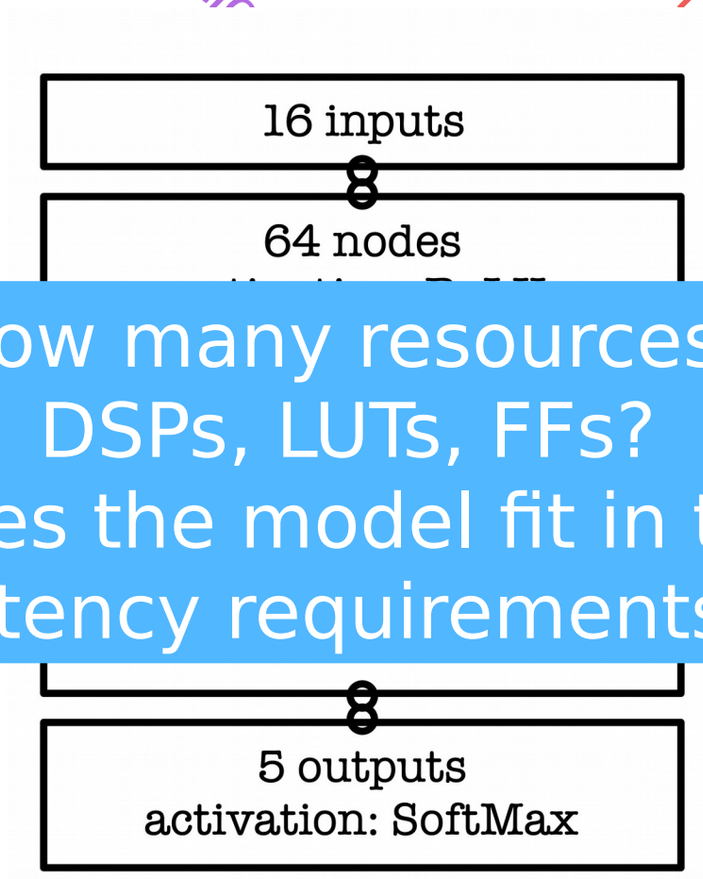


Neural network inference



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

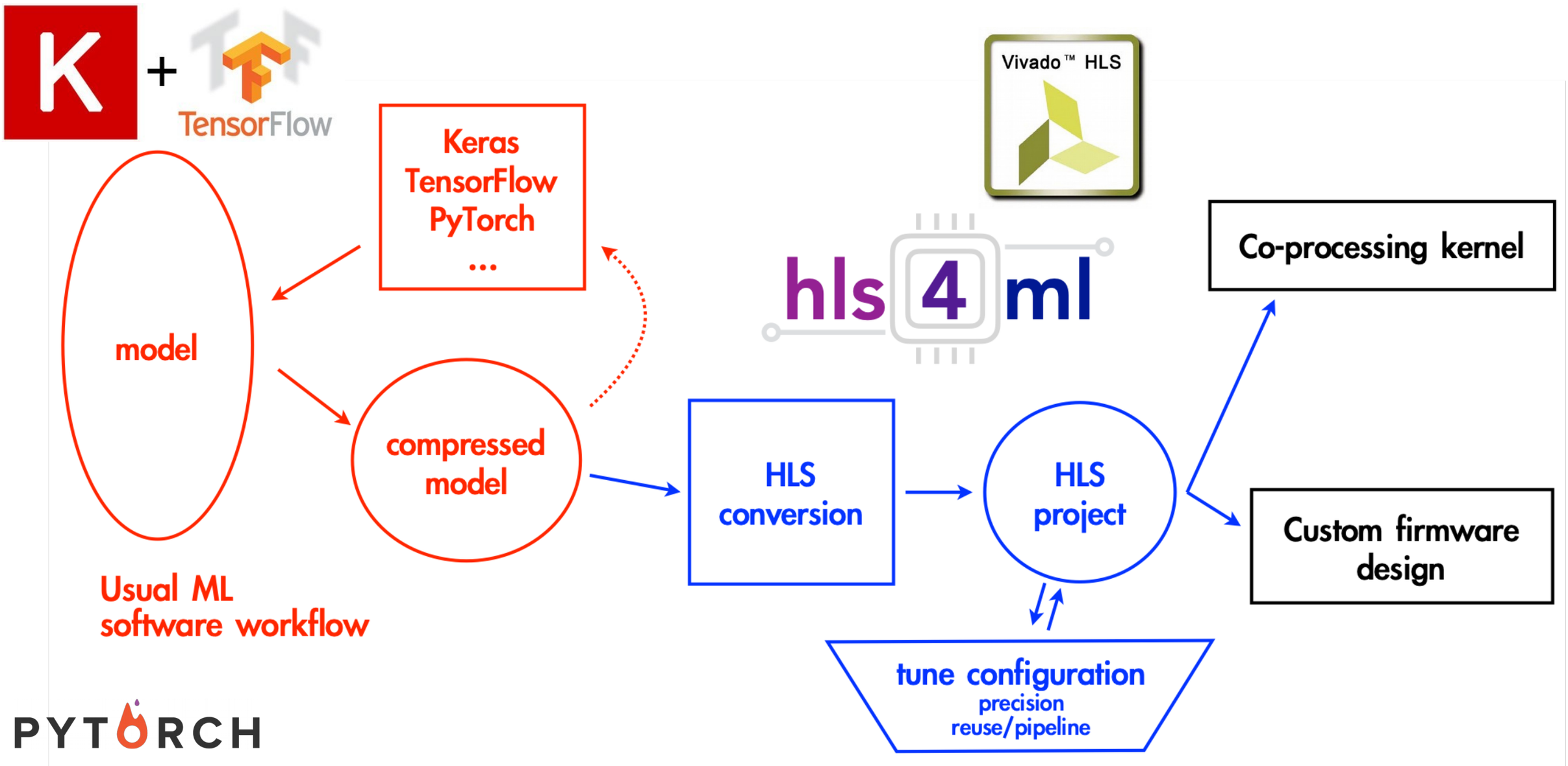
activation function
precomputed and stored in BRAMs
multiplication
DSPs
addition
logic cells



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

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

high level synthesis for machine learning



PYTORCH

ONNX

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

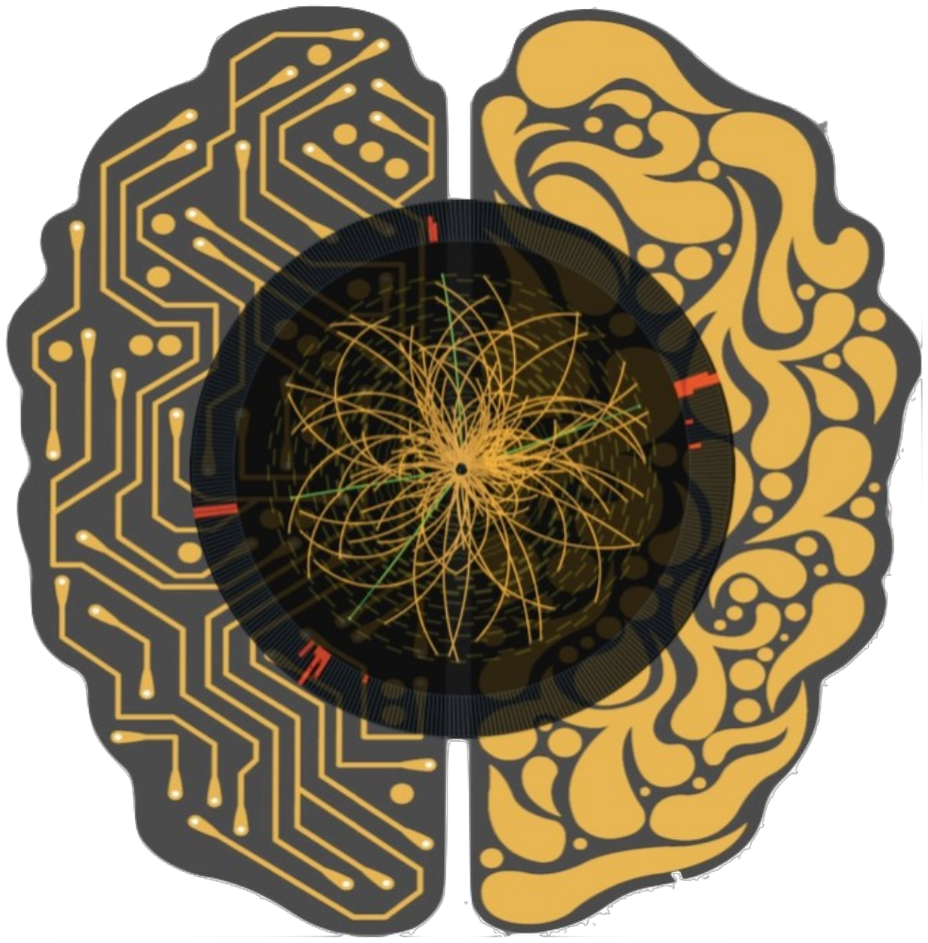
Constraints:
Input bandwidth
FPGA resources
Latency

<u>Today you will learn how to optimize your project through:</u>

- **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

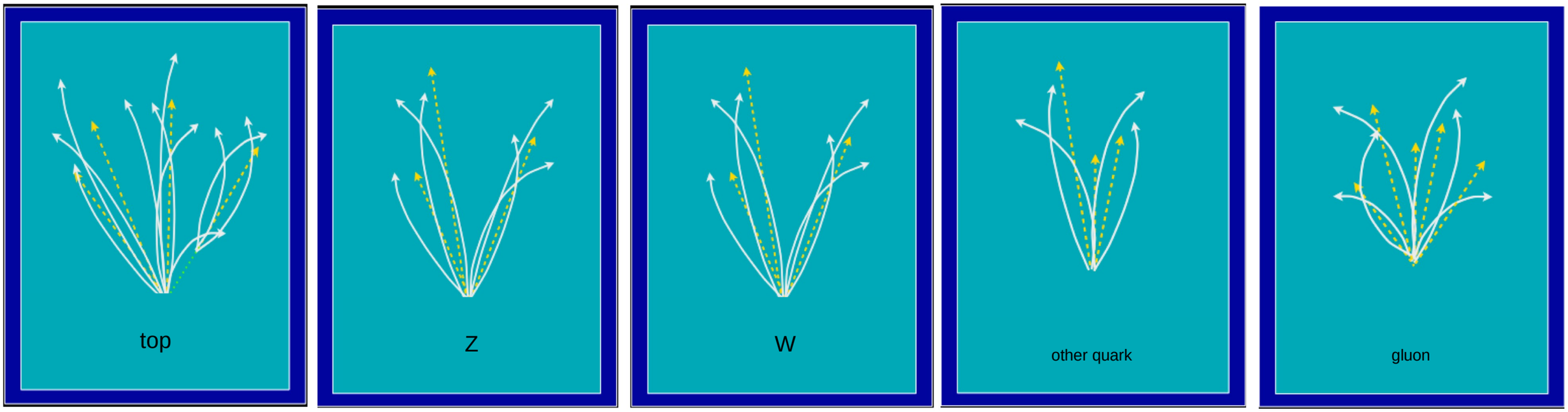


hls4ml tutorial

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



t→bW→bqq — 3-prong jet

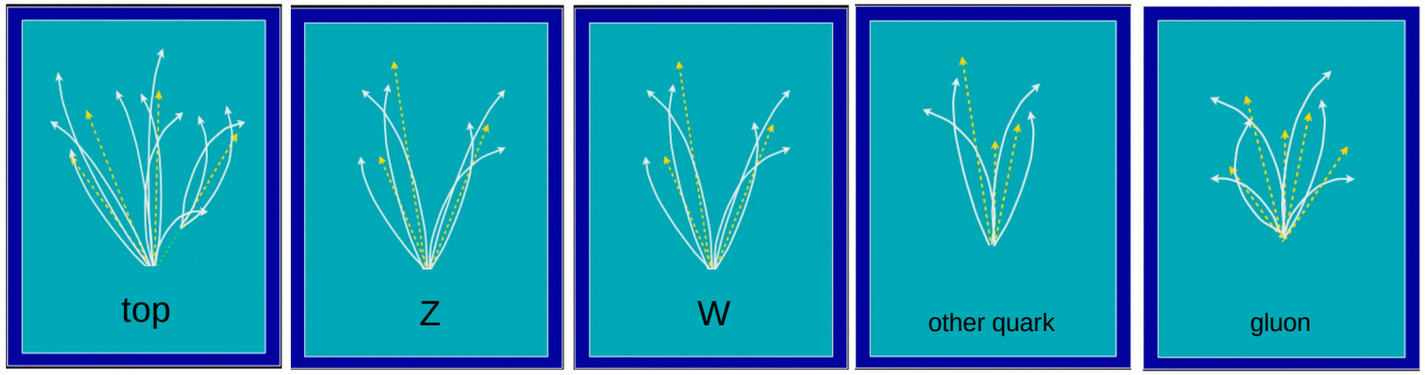
Z→qq — 2-prong jet

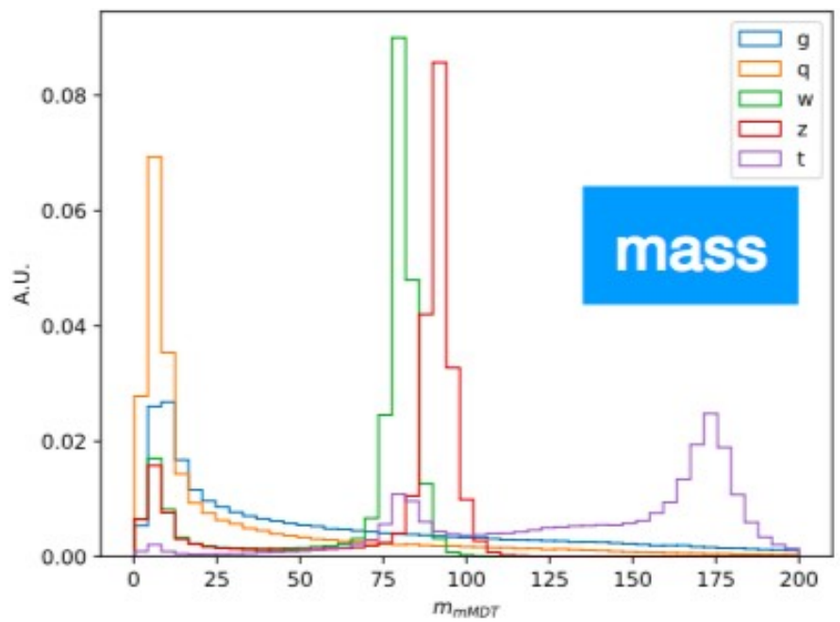
W→qq — 2-prong jet

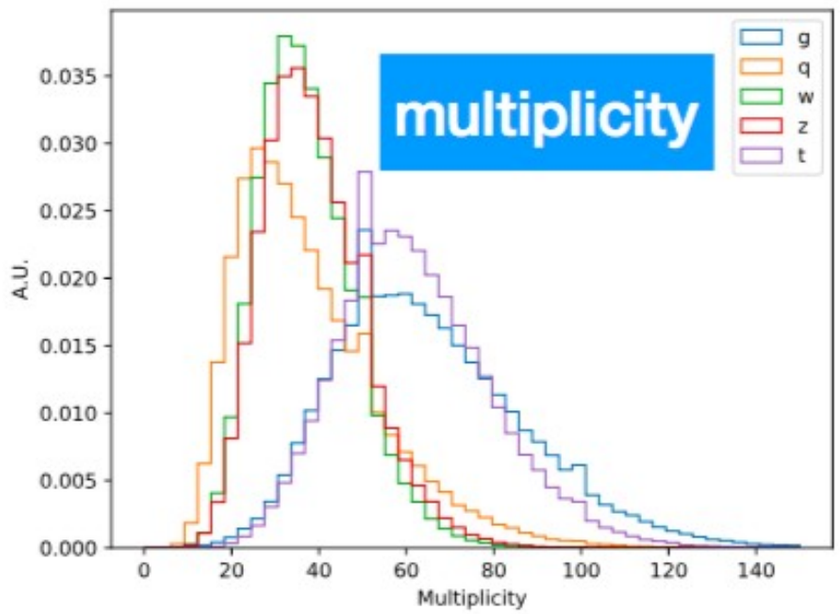
q/g background — no substructure and/or mass ~ 0

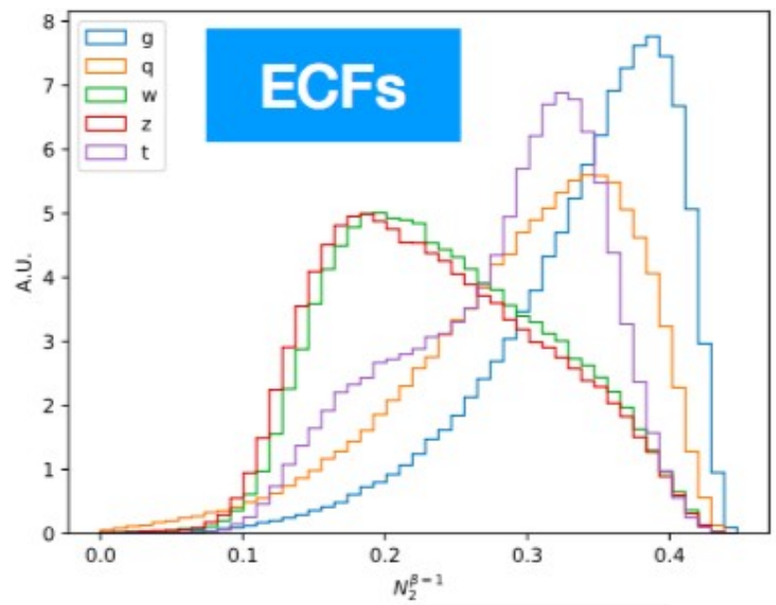
Reconstructed as one massive jet with substructure

Physics case: jet tagging









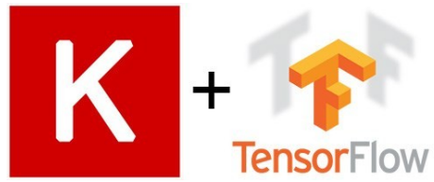
Input variables: several observables known to have high discrimination power from offline data analyses and published studies [*]

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

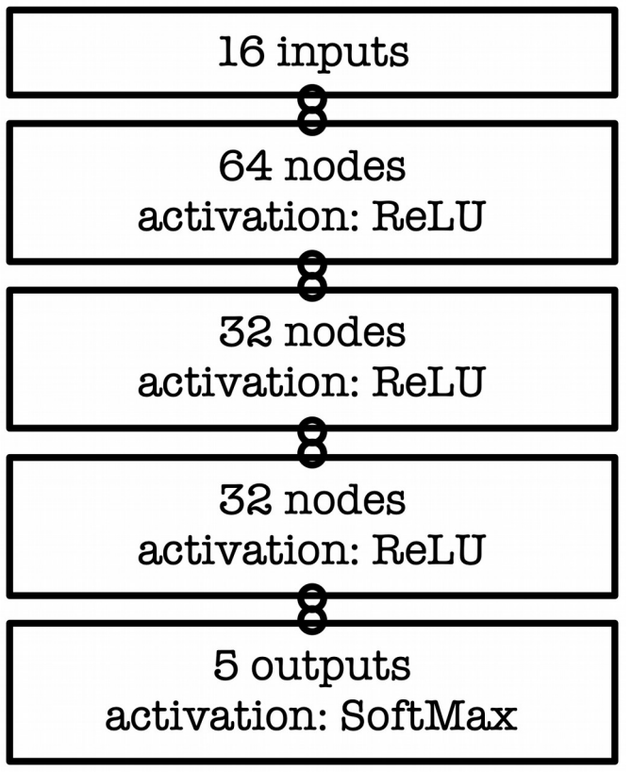
m_{mMDT}
$N_2^{\beta=1,2}$
$M_2^{\beta=1,2}$
$C_1^{\beta=0,1,2}$
$C_2^{\beta=1,2}$
$D_2^{\beta=1,2}$
$D_2^{(\alpha,\beta)=(1,1),(1,2)}$
$\sum z \log z$
Multiplicity

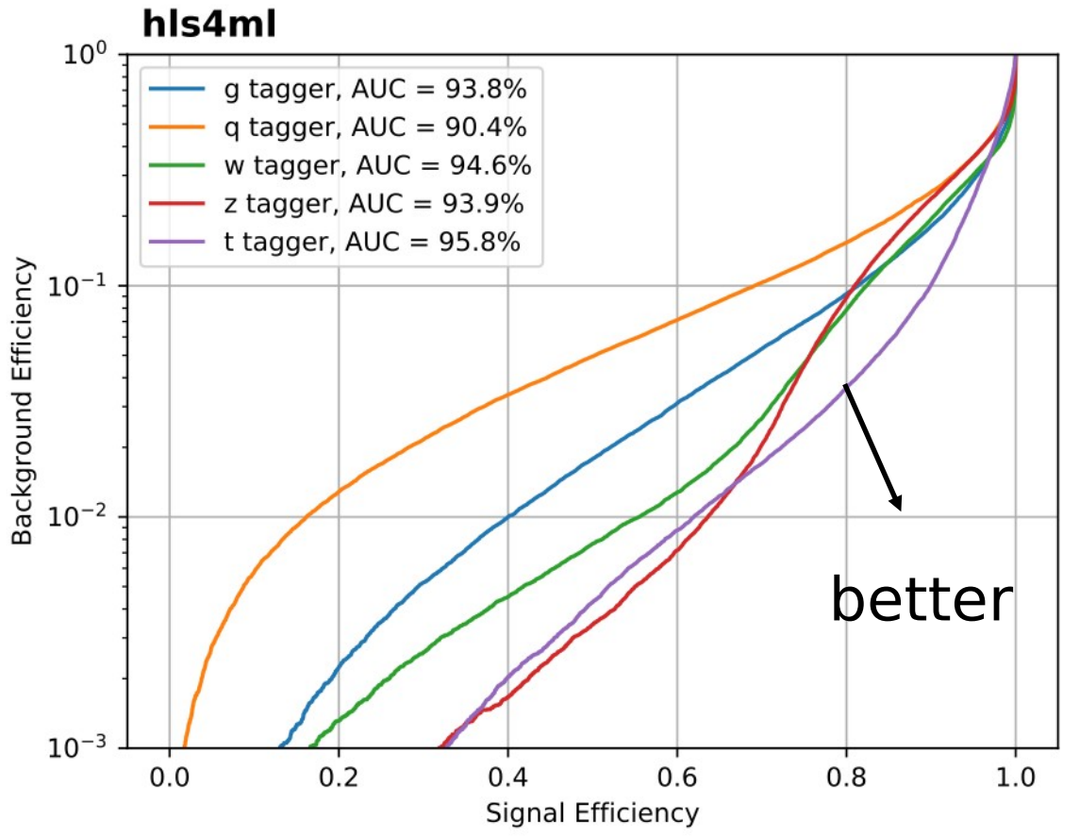
Physics case: jet tagging

- We train (on GPU) the five output 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

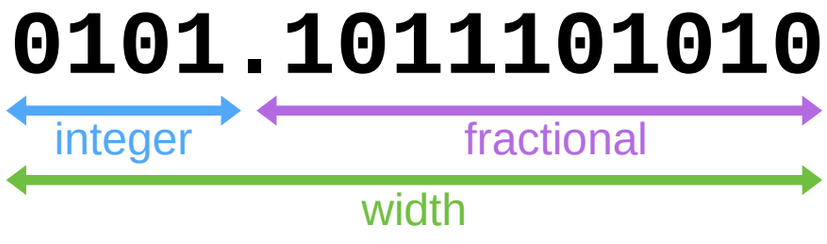




AUC = area under ROC curve (100% is perfect, 20% is random)

Efficient NN design: quantization

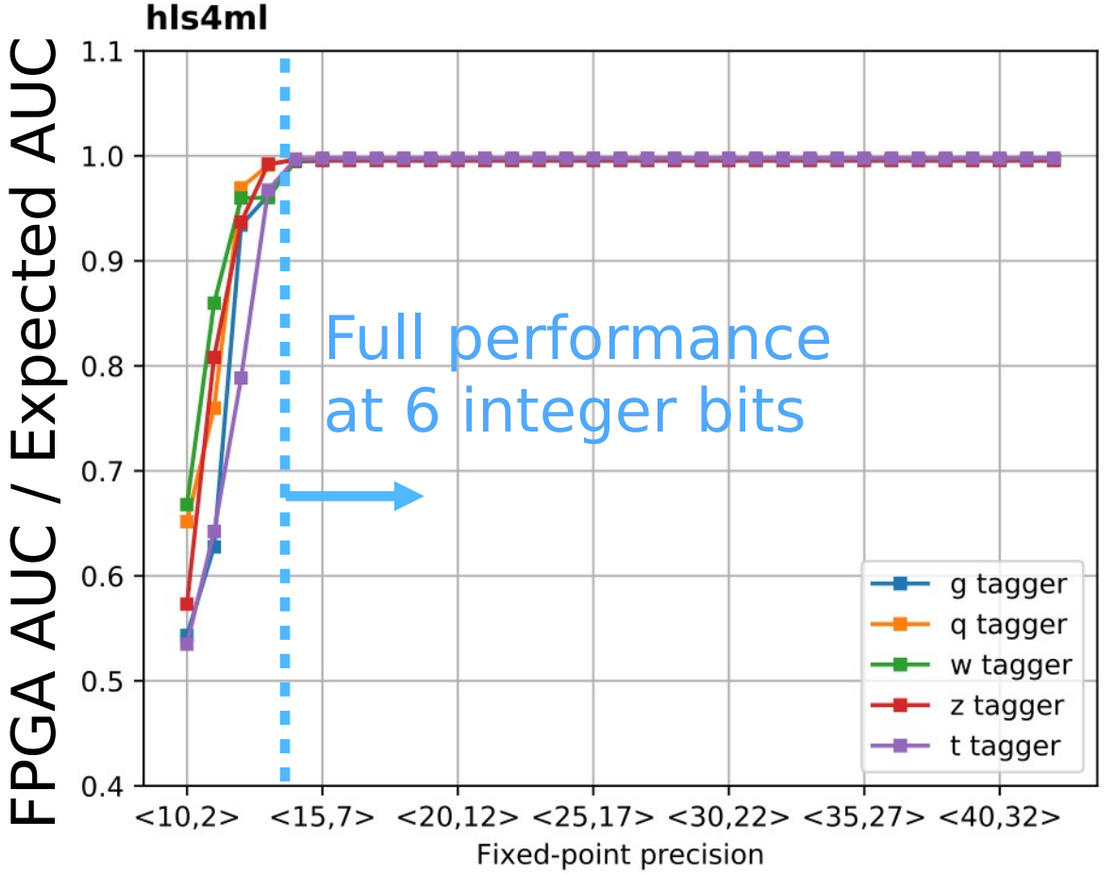
ap_fixed<width bits, integer bits>



- 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!

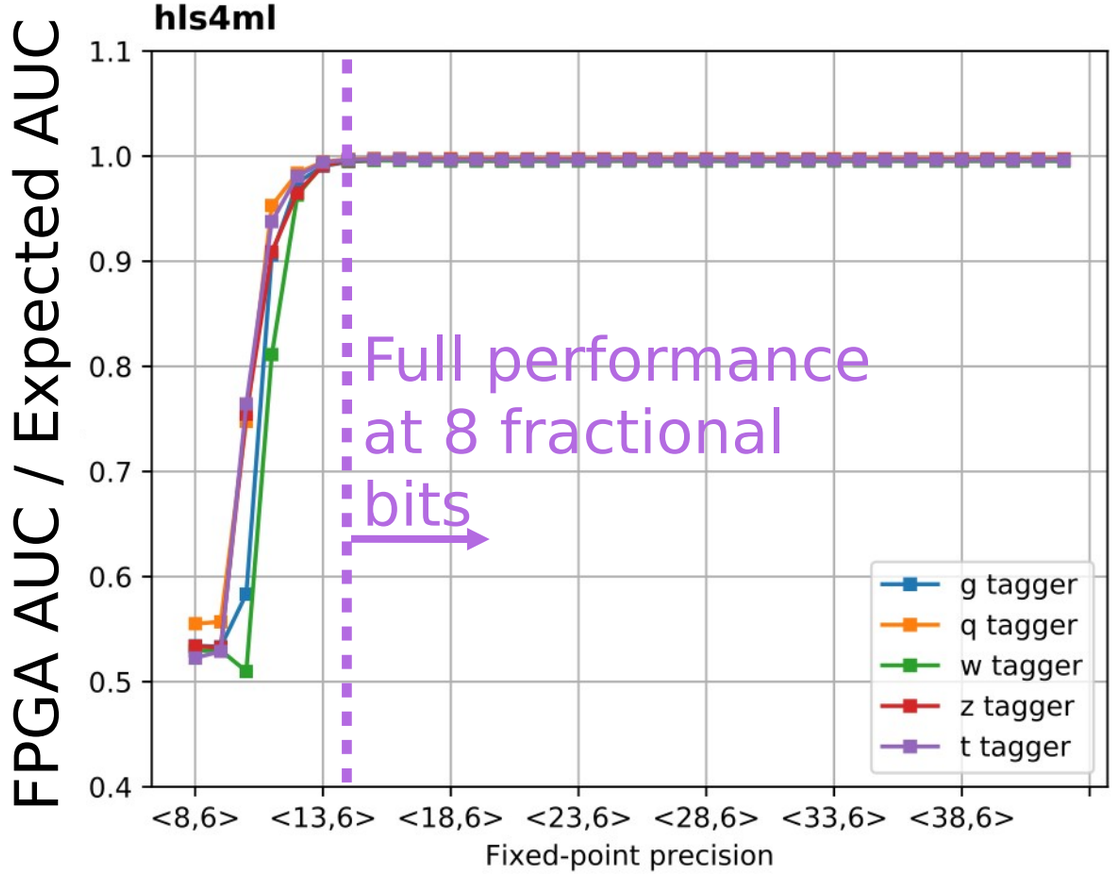
Scan integer bits

Fractional bits fixed to 8



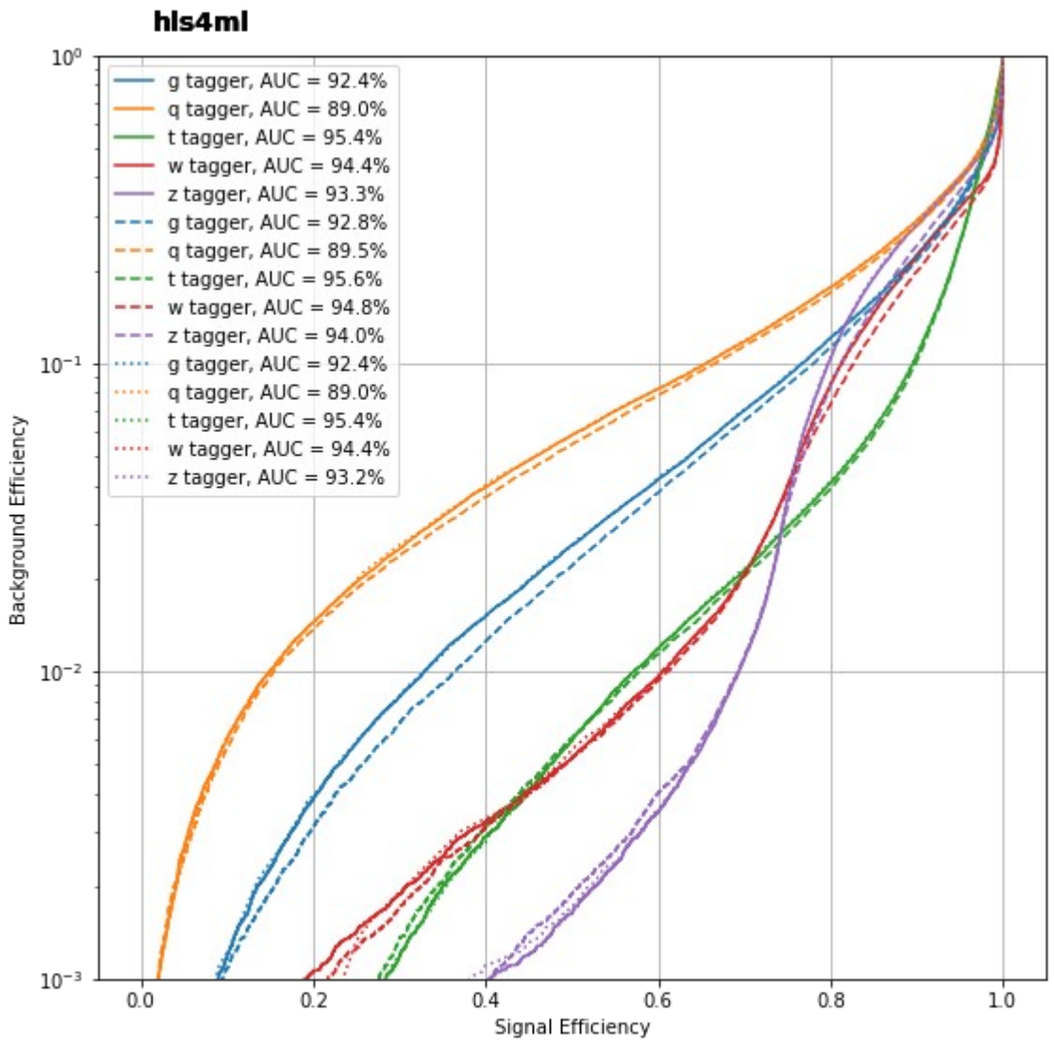
Scan fractional bits

Integer bits fixed to 6



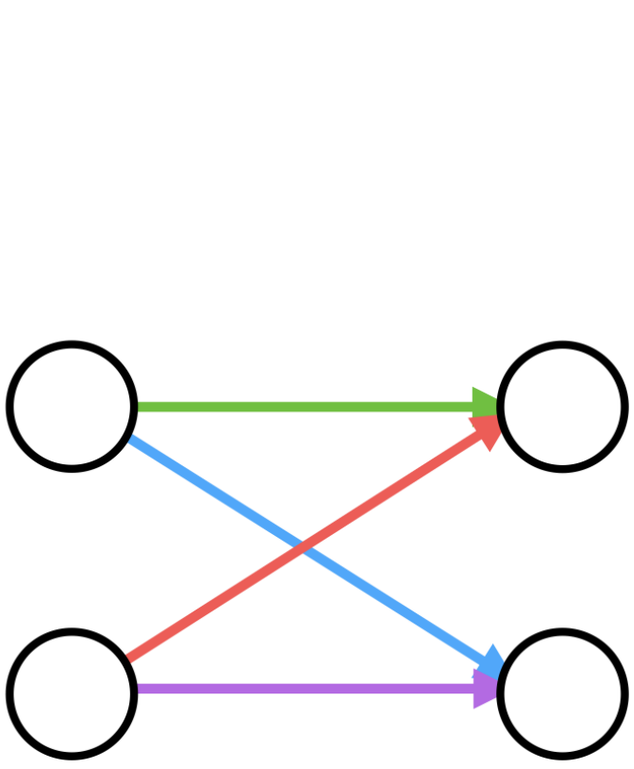
Efficient NN design: quantization

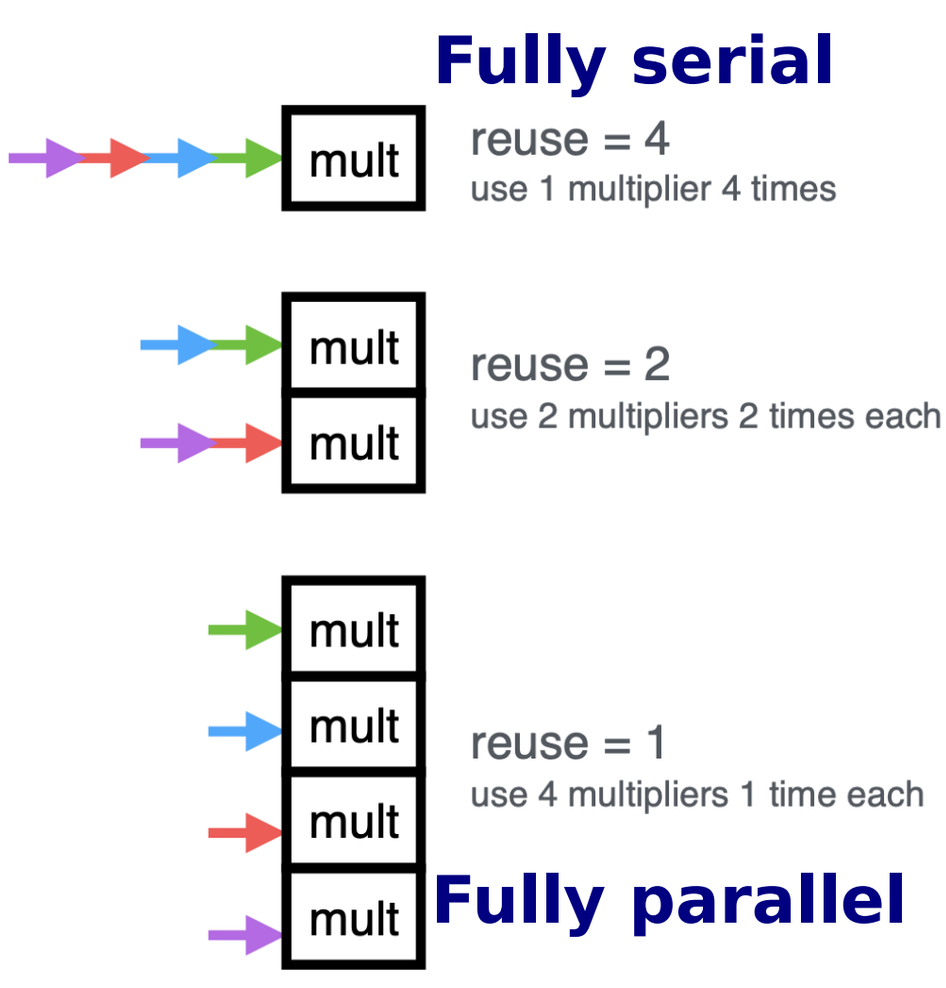
- hls4ml allows you to use different data types everywhere, we will learn how to use that
- We will also try quantization-aware training with Qkeras
- With quantization-aware we can even go down to just 1 or 2 buts
  - See our recent work: https://arxiv.org/abs/2003.06308

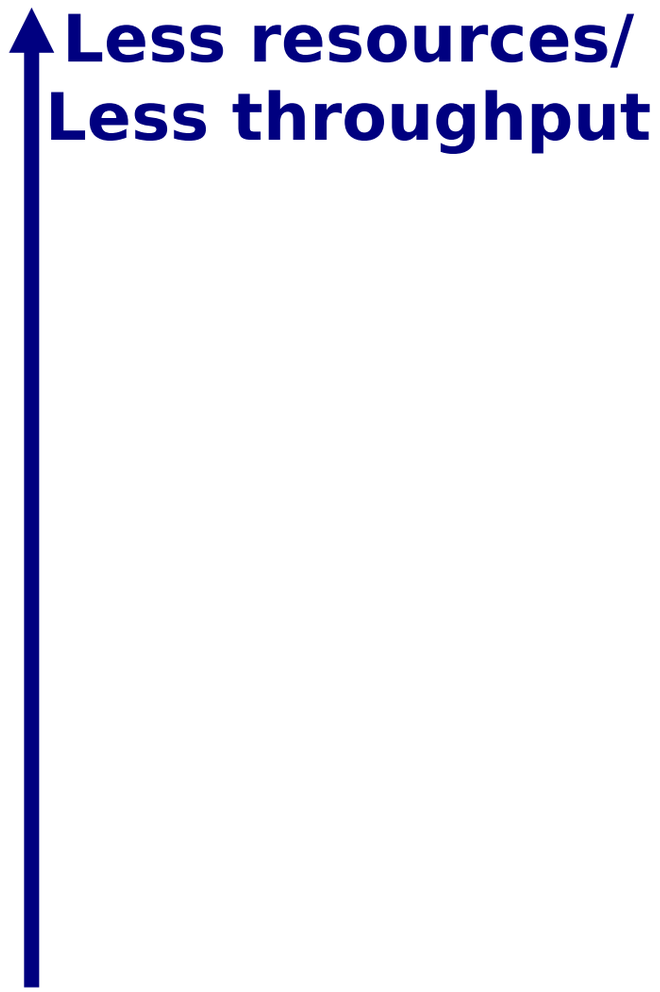


Efficient NN design: parallelization

- 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

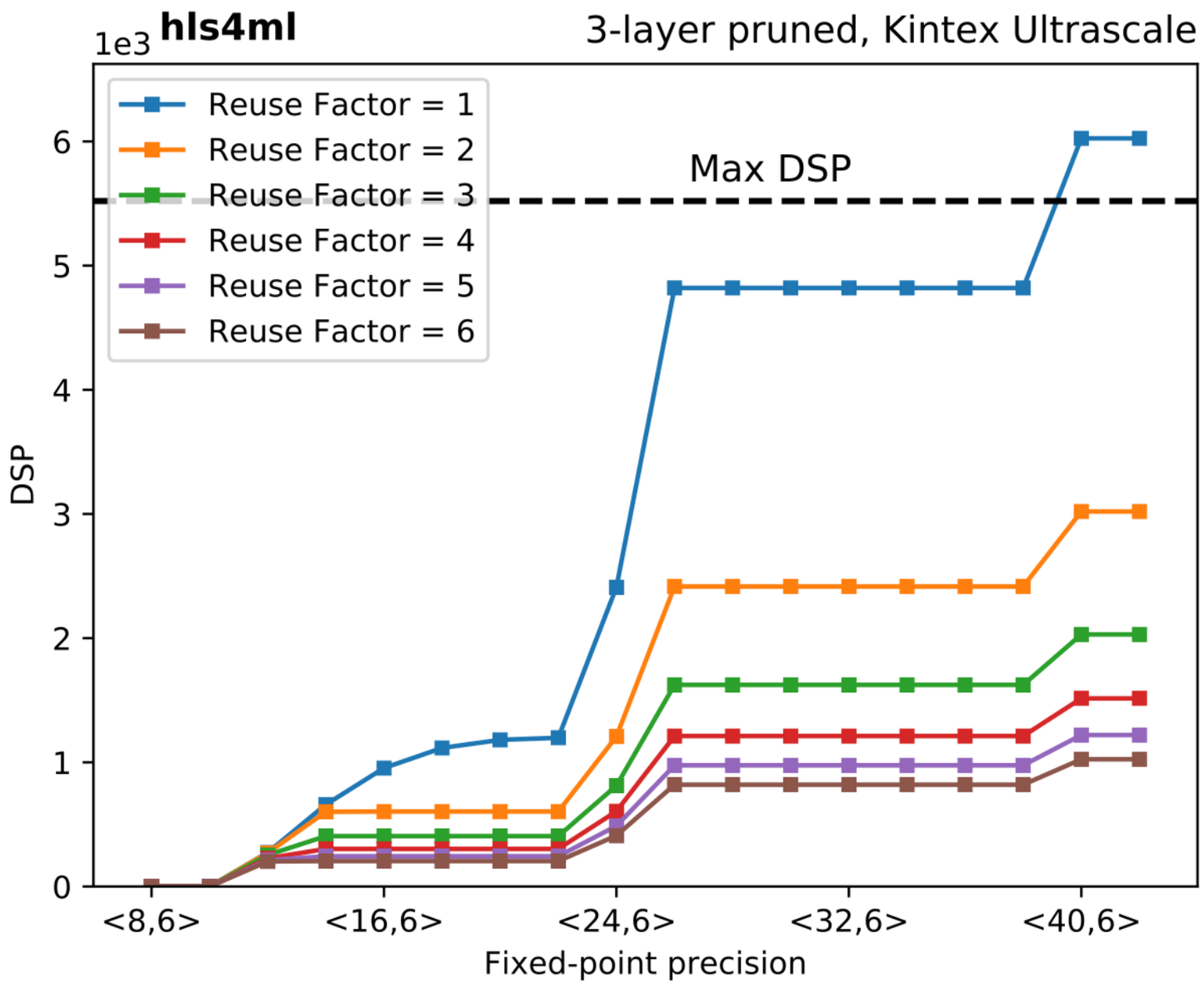






Reuse factor: how much to parallelize operations in a hidden layer

Parallelization: DSP usage



More resources

Fully parallel
Each mult. used 1x

Each mult. used 2x

Each mult. used 3x

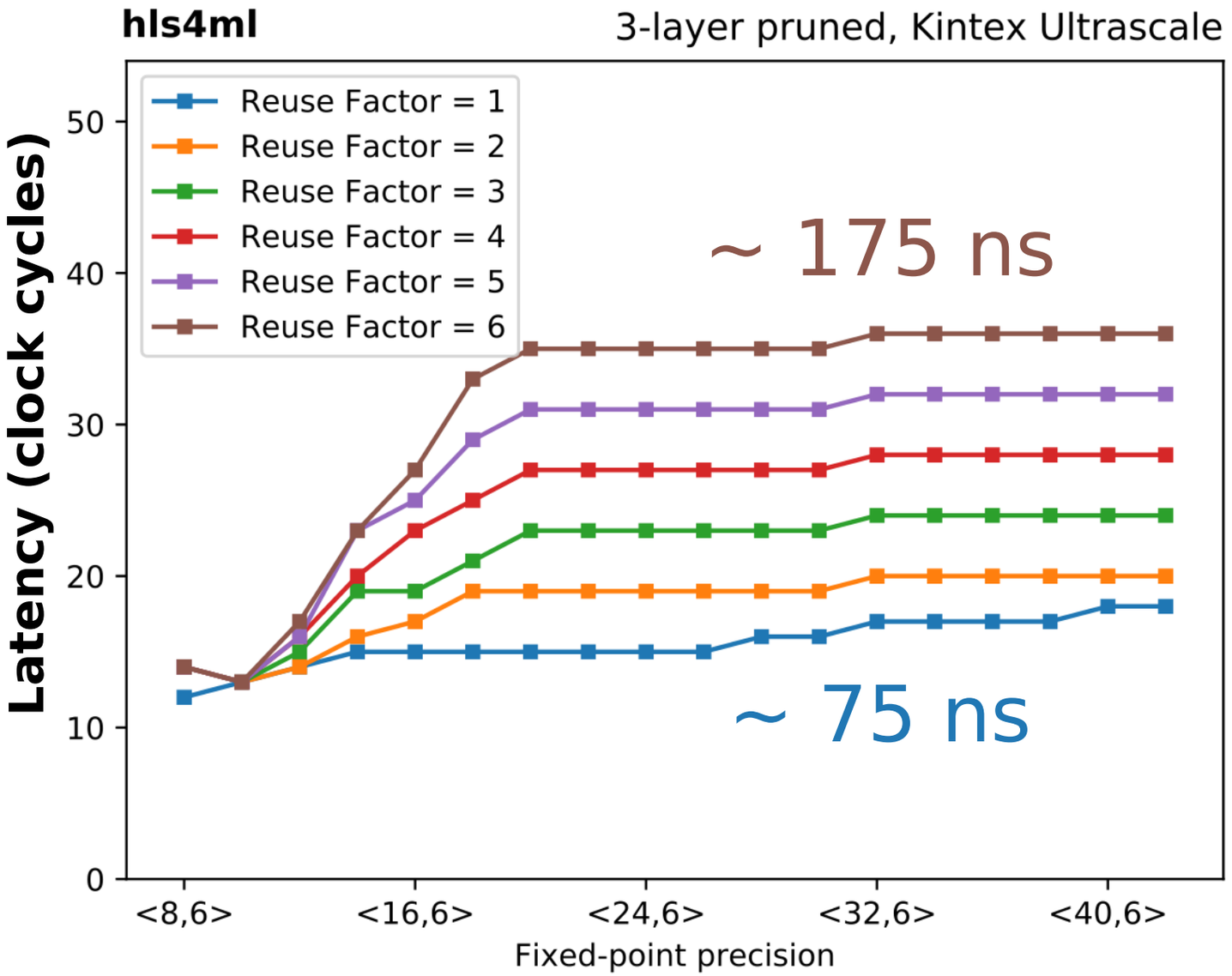
⋮

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

What we *won't* cover today

- Convolutional NNs : the convolutional layer implementation in hls4ml is in a state of flux, and whatever I show you today will soon be obsolete
  - You *could* try QKeras training of a CNN and evaluate model accuracy with hls4ml, but synthesizing the model doesn't make sense just yet
- Recurrent NNs: these have recently been added, but not yet ready for a tutorial here!
- What comes after hls4ml… you would need to integrate the 'IP core' into a larger design
  - For a custom board, you'd need to do this by hand (e.g. CMS L1 Trigger)
  - For boards integrated with Vitis, e.g. Alveo, the workflow is quite automatic

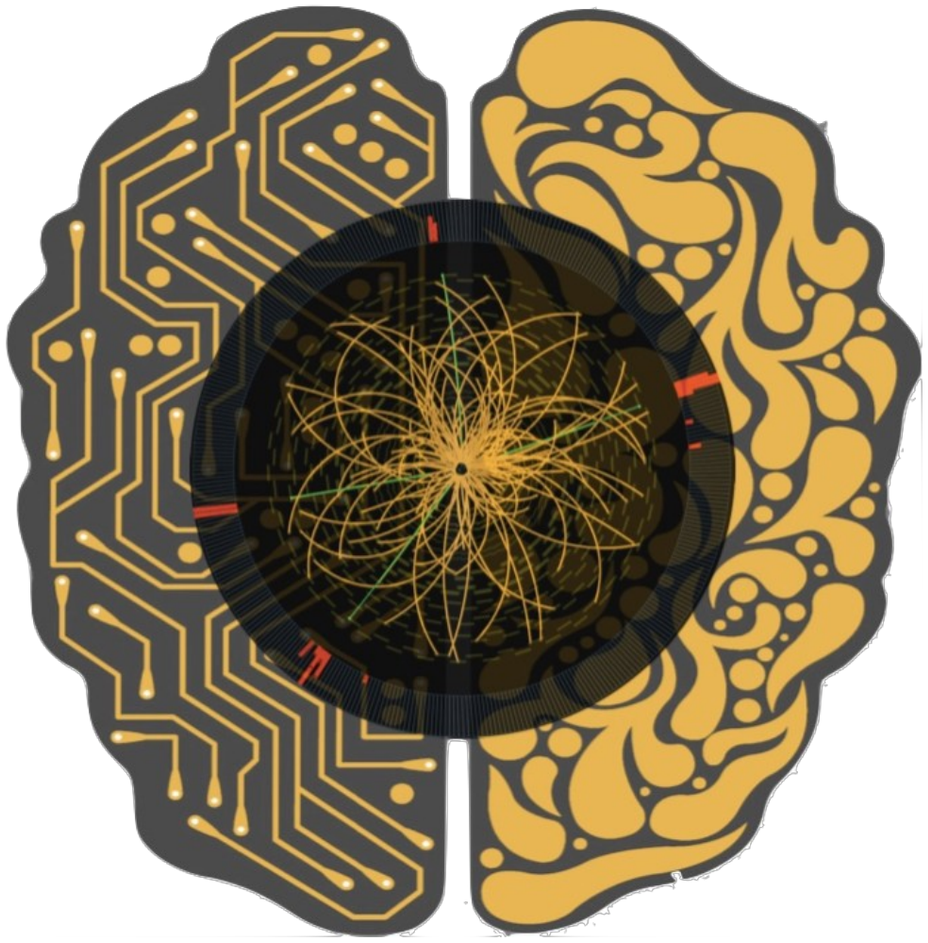
Hands On - Setup

- We have 8 PCs set up with temporary access for this tutorial
  - With installation of hls4ml Python env & Vivado HLS software
- The hands on is in the form of Jupyter notebooks served from these machines
- You'll need to connect a port on your local machine to the port we've opened for these notebooks, e.g.:
  - ssh -N -f -L 8888:localhost:ABCD <user>@lxplusEFG.cern.ch
  - Then open 'localhost:8888' in your browser
  - You will be asked for a token
  - The exact ABCD, EFG and token will be provided…
  - If port '8888' on your machine is taken (e.g. if you have a local Jupyter notebook server running), you can use another available port, e.g. 8889, 8890 etc
- List of notebook servers is at: https://docs.google.com/spreadsheets/d/1dWraLROwLo_Lg2Eg8lrexLQNfNEf-lvTGbjZggakfw4/edit#gid=0

Hands On - Setup

After successful connection you should see some notebooks, as here

Go ahead and open “part1”



hls4ml Tutorial

Part 2: Advanced Configuration

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
- We use a different partitioning on the first layer for the best partitioning of arrays

```
KerasJson: keras/MNIST_model.json
KerasH5:   keras/MNIST_model_weights.h5
OutputDir: my-hls-test
ProjectName: myproject
XilinxPart: xcku115-flvb2104-2-i
ClockPeriod: 5

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

This model was trained on the MNIST digits classification dataset
Architecture: 784 x 128 x 128 x 128 x 10
Model accuracy: ~97%
Can you calculate the number of DSPs it will use?
(Don’t cheat and look ahead)

Part 2: Large MLP

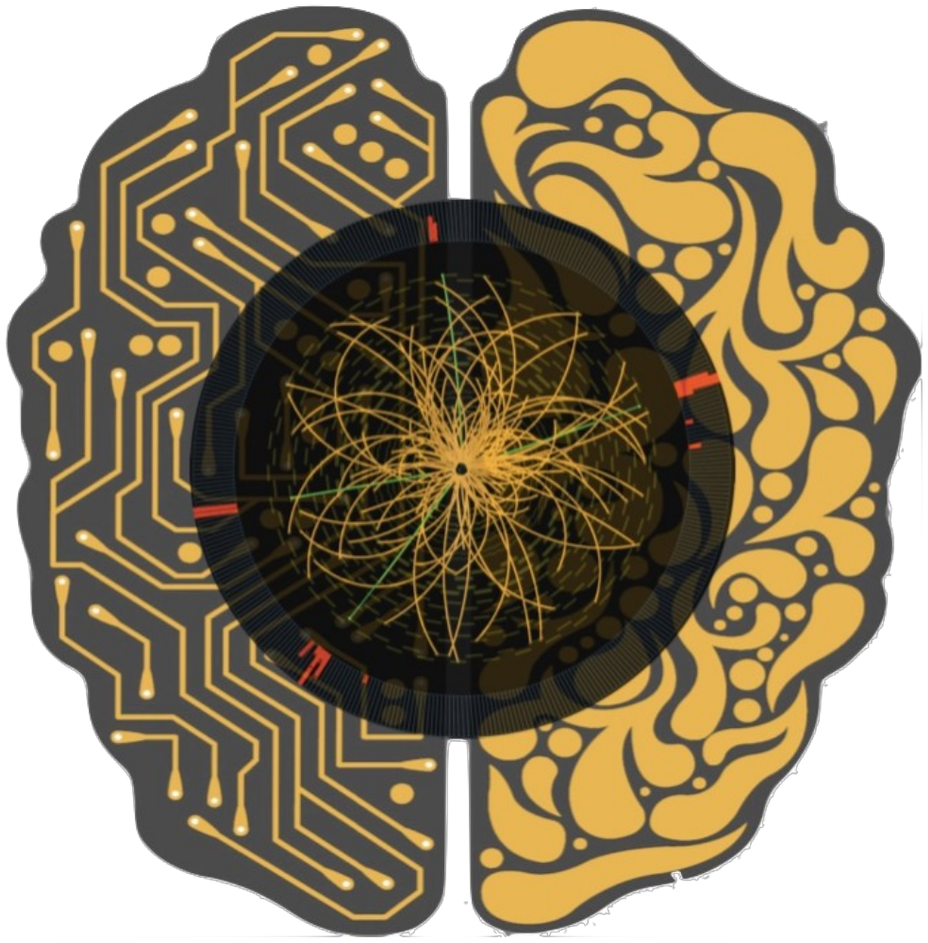
- It takes a while to synthesise, so here’s one I made earlier…
- The DSPs should be: (784 x 128) / 112 + (2 x 128 x 128 + 128 x 10) / 128 = 1162

```
================================
================================
+ 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   |
+---------------------+---------+-------+---------+--------+
|DSP                  |        -|      -|        -|       -|
|Expression           |        -|      -|        0|    3144|
|FIFO                 |     1394|      -|    28998|   46116|
|Instance             |      568|   1162|   140203|  166361|
|Memory               |        -|      -|        -|       -|
|Multiplexer          |        -|      -|        -|    7002|
|Register             |        -|      -|      778|       -|
+---------------------+---------+-------+---------+--------+
|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, arxiv.1712.01312]

  - **low-rank factorization:** using matrix/tensor decomposition to estimate informative parameters [arxiv.1405.3866]

  - **transferred/compact convolutional filters:** special structural convolutional filters to save parameters [arxiv.1602.07576]

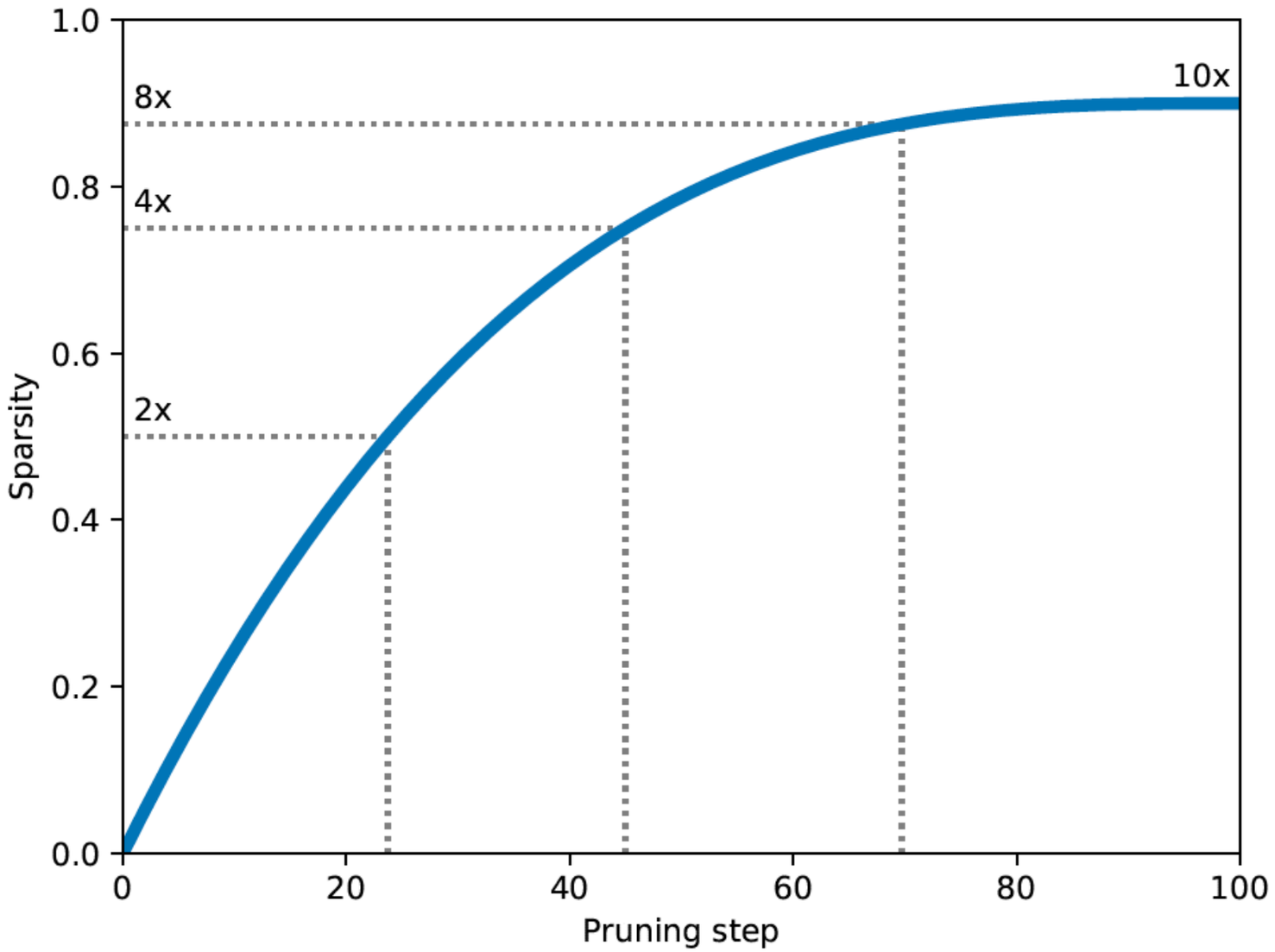
  - **knowledge distillation:** training a compact network with distilled knowledge of a large network [doi:10.1145/1150402.1150464]

- Today we'll use the tensorflow model sparsity toolkit
  - https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html

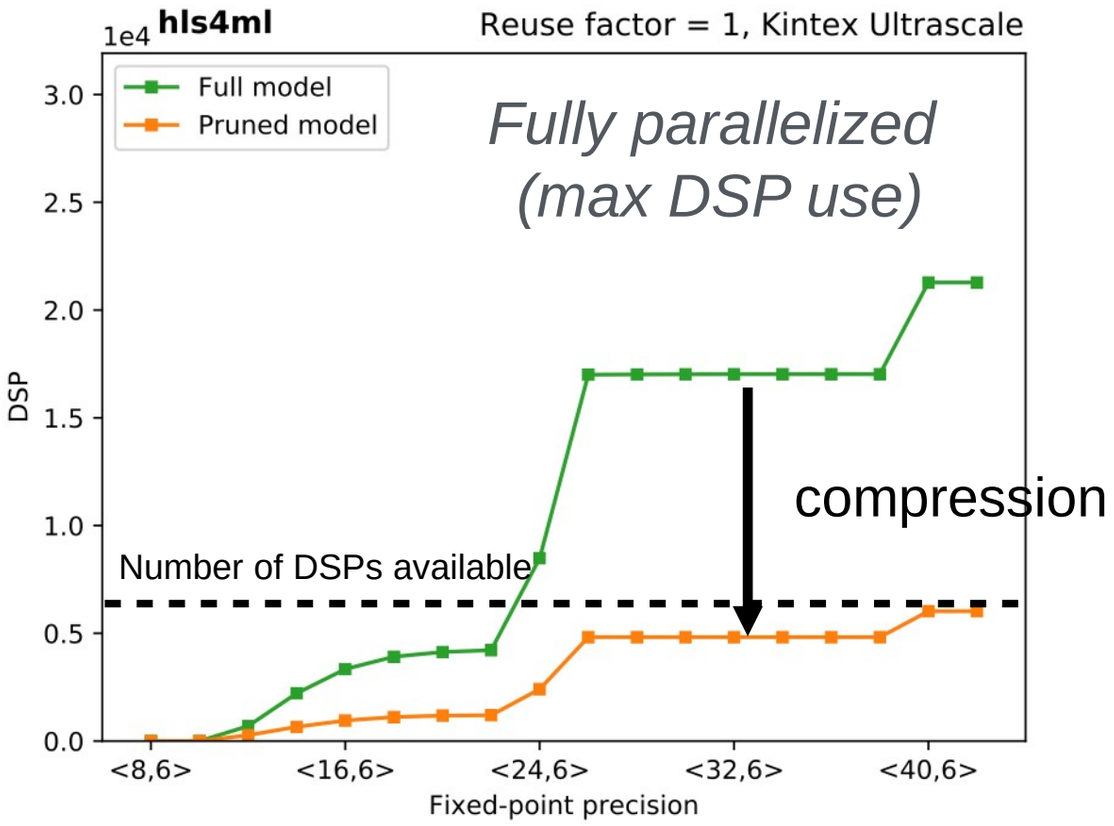
- But you can use other methods!

TF Sparsity

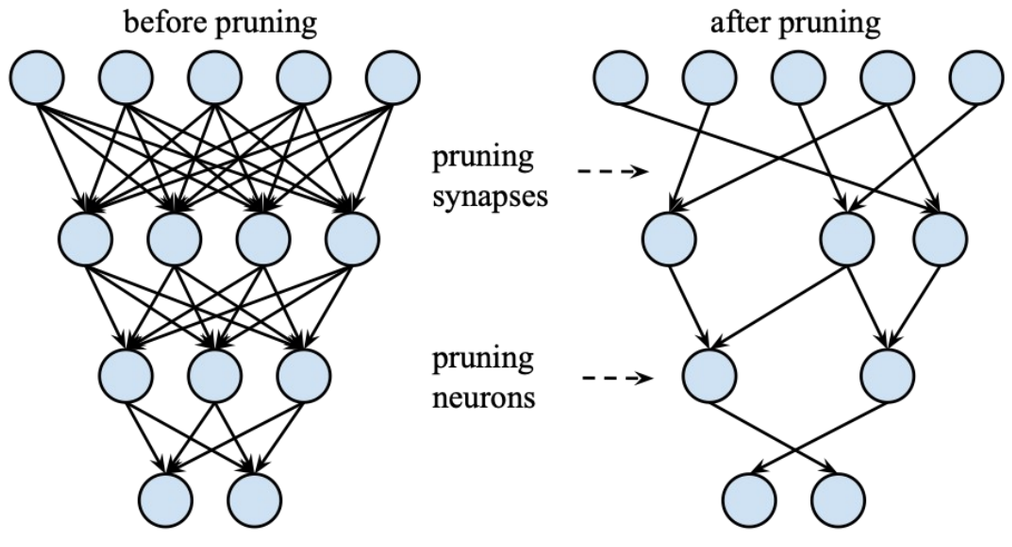
- Iteratively remove low magnitude weights, starting with 0 sparsity, smoothly increasing up to the set target as training proceeds



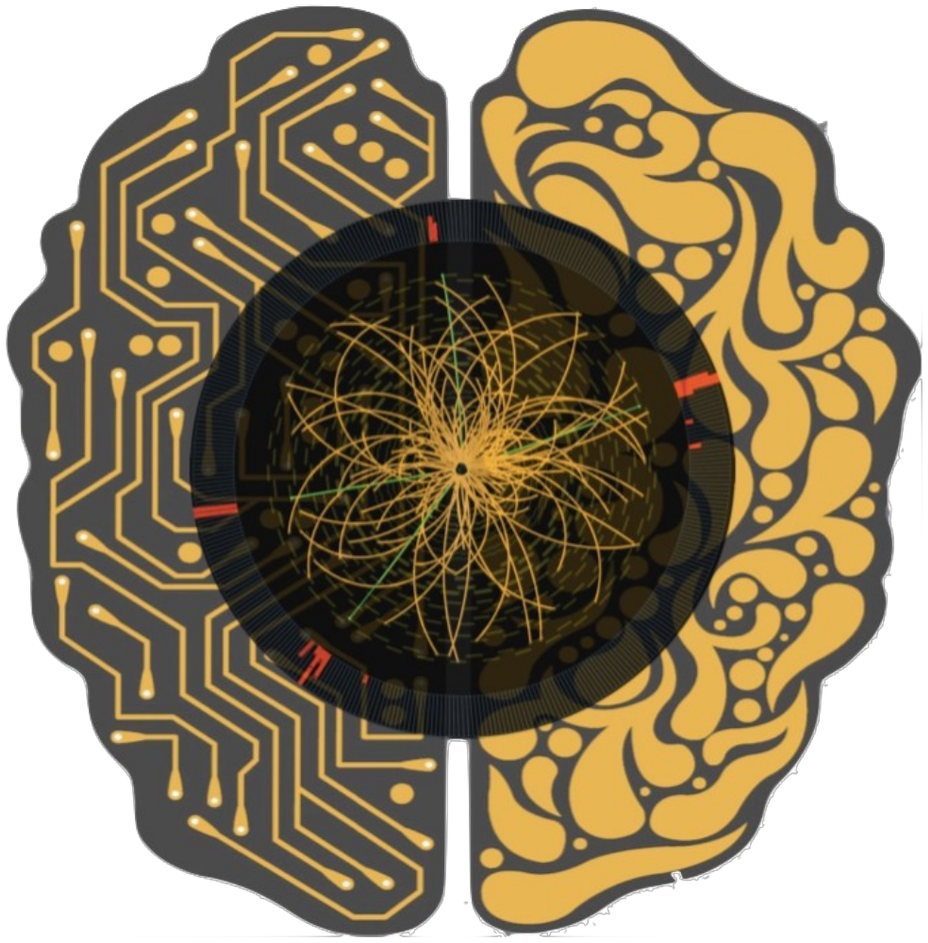
Efficient NN design: compression



70% compression ~ 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

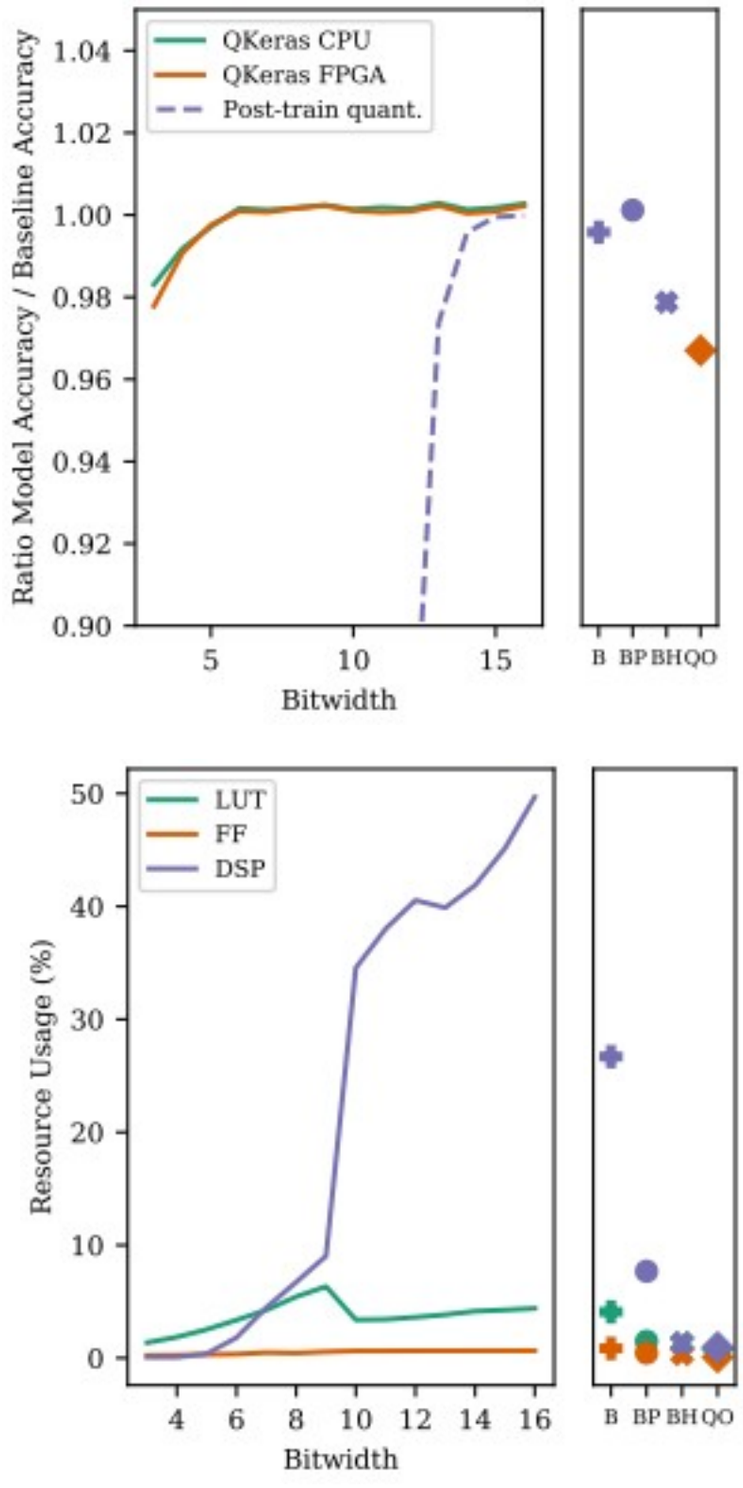


hls4ml Tutorial

Part 4: Quantization

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 had some hands on experience with hls4ml
  - Translated trained neural networks to FPGA firmware
  - Simulated the quantized firmware, checking performance
  - Synthesized the network, inspecting the resource and timing summary
- Seen how to simply prune a neural network and the impact on resources
- Trained a model with small number of bits using QKeras
- If you'd like to continue working with hls4ml and Vivado HLS, ask about adding your CERN user account to these PCs setup with the tools