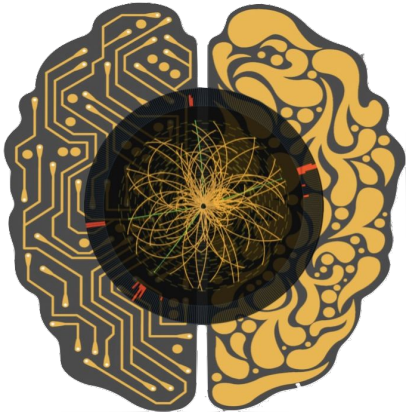


hls4ml

CERN

hls4ml tutorial

Sioni Summers (CERN)

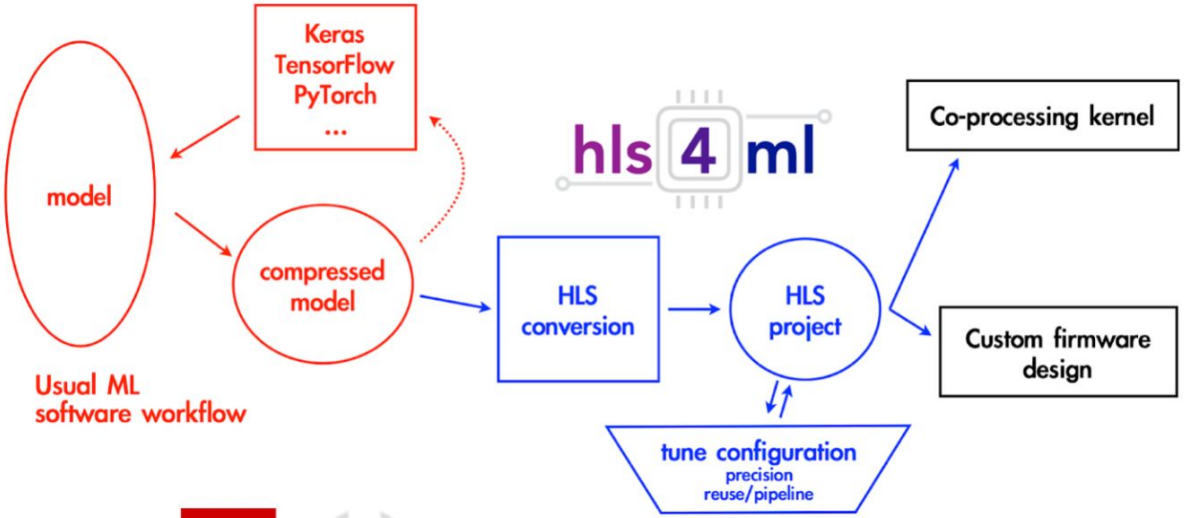
Introduction

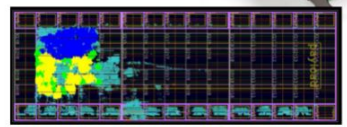
- **hls4ml** is a package for translating neural networks to FPGA firmware for inference with extremely low latency on FPGAs
    - https://github.com/fasthmachinelearning/hls4ml
    - https://fastmachinelearning.org/hls4ml/
    - `pip install hls4ml`

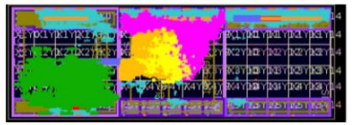
- In this session you will get hands on experience with the **hls4ml** package
- We'll learn how to:
    - Translate models into synthesizable FPGA code
    - Explore the different handles provided by the tool to optimize the inference
        - Latency, throughput, resource usage
    - Make our inference more computationally efficient with pruning and quantization

high level synthesis for machine learning

fastmachinelearning.org/hls4ml/







(Q) K + TensorFlow

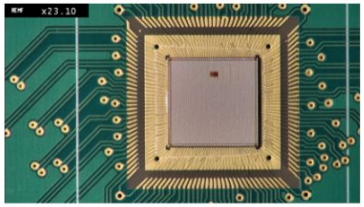
(Q) ONNX

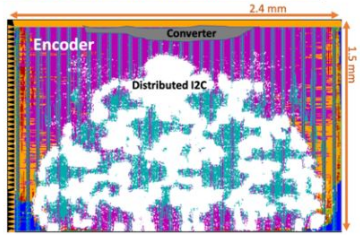
PYTORCH

Vivado™ HLS

HLS COMPILER

Mentor Catapult A Siemens Business
Coming soon

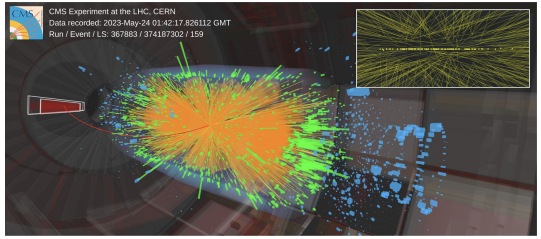


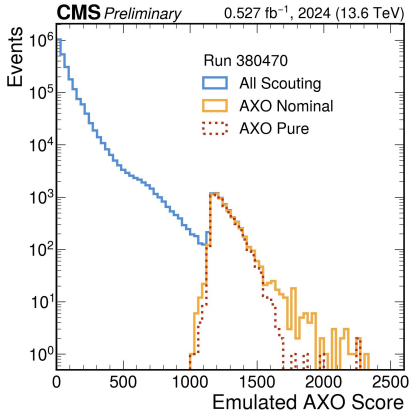


Some hls4ml Applications

Selecting anomalous LHC collisions at the CMS experiment

- Variational AutoEncoder
- 50 ns

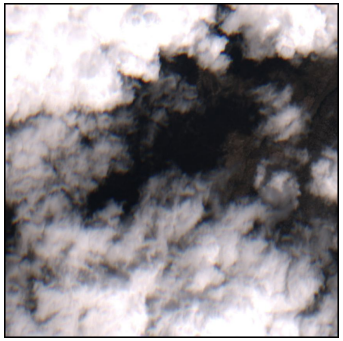




AXOL1TL

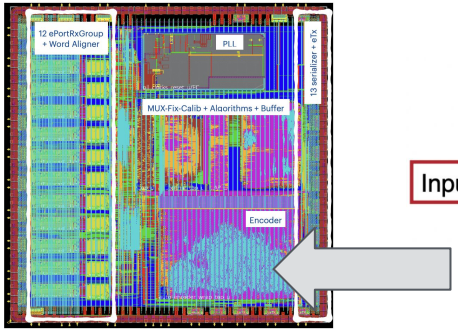
Filtering images onboard Earth Observation satellites

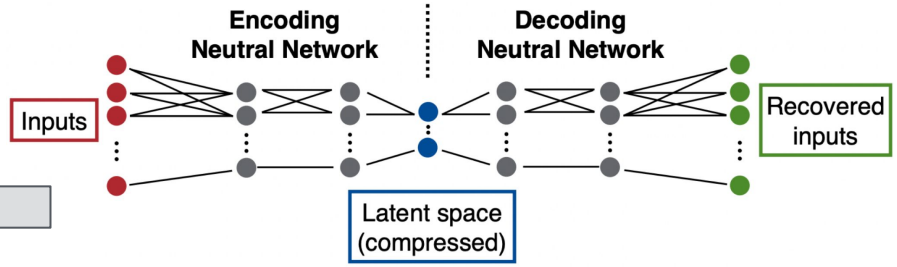
- UNet Image Segmentation
- Optimized for power efficiency and throughput



Edge SpAIce

Space AI monitoring marine plastic litter





CMS Detector frontend ASIC for data compression

- Radiation tolerant
- Low power, low latency, high throughput

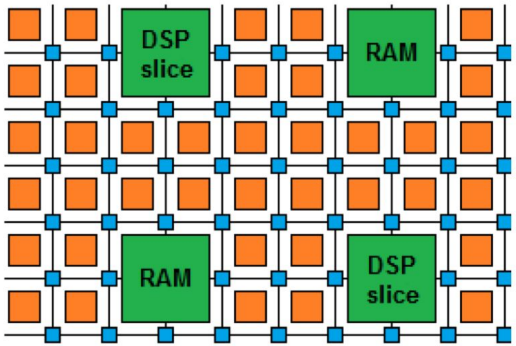
What are FPGAs?

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 and edge computing

FPGA diagram



What are FPGAs?

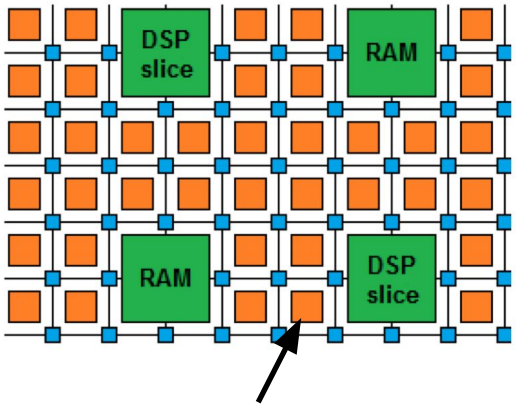
Field Programmable Gate Arrays are reprogrammable integrated circuits

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

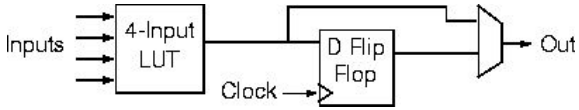
These can be used for boolean operations, arithmetic, small memories

Flip-Flops register data in time with the clock pulse

FPGA diagram



Logic cell



Look-up table (logic)

Flip-flop (registers)

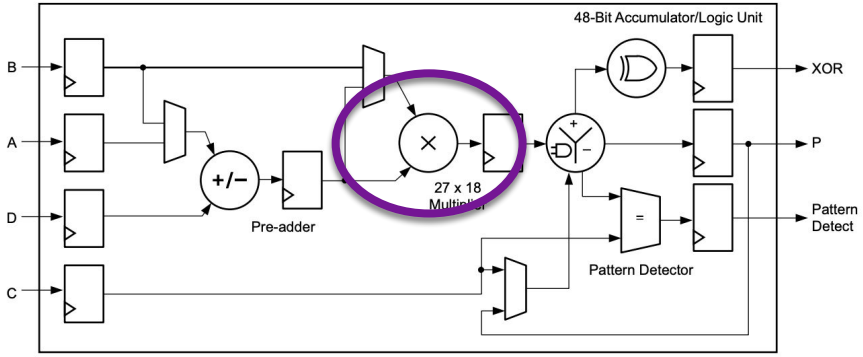
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

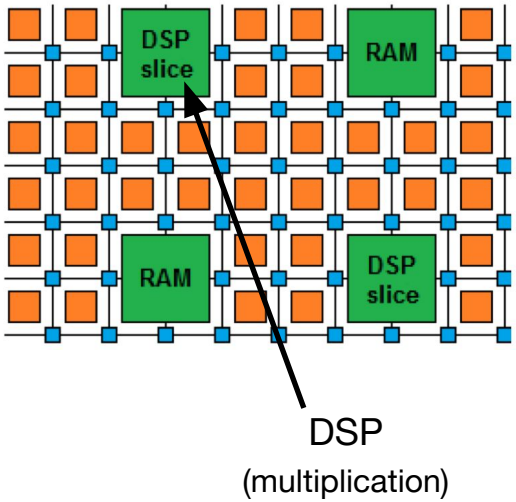
DSPs (Digital Signal Processor) are specialized units for multiplication and arithmetic

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

Neural Network inference involves lots of linear algebra



FPGA diagram



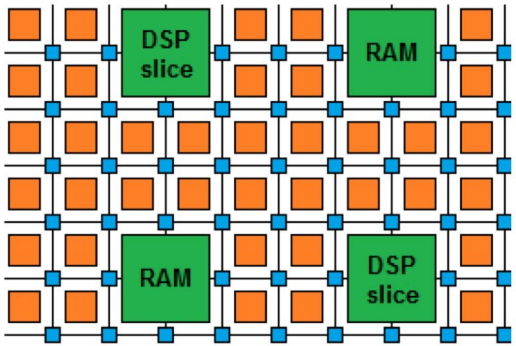
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

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

A big FPGA has nearly 100Mb of BRAM, chained together in blocks

FPGA diagram



What are FPGAs?

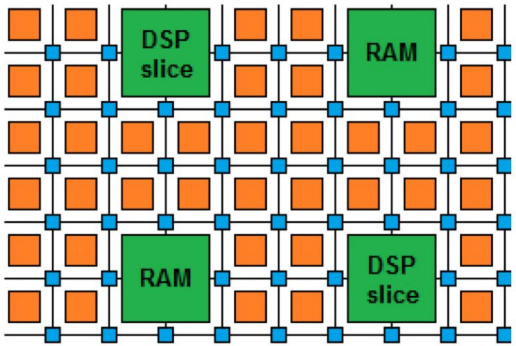
In addition, there are specialised blocks for I/O, making FPGAs popular in embedded systems and HEP triggers

High speed transceivers with Tb/s total bandwidth
PCIe, (Multi) Gigabit Ethernet, Infiniband

AND: Support highly parallel algorithm implementations

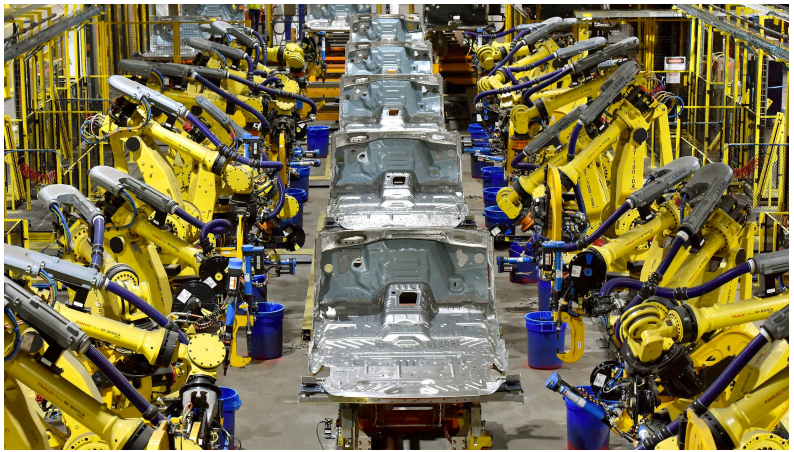
Low power per Op (relative to CPU/GPU)

FPGA diagram



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…

hls4ml dataflow

- **hls4ml** uses a “dataflow architecture” → each compute step is a discrete compute unit

Conv2D

ReLU

MaxPool2D

Conv2D

ReLU

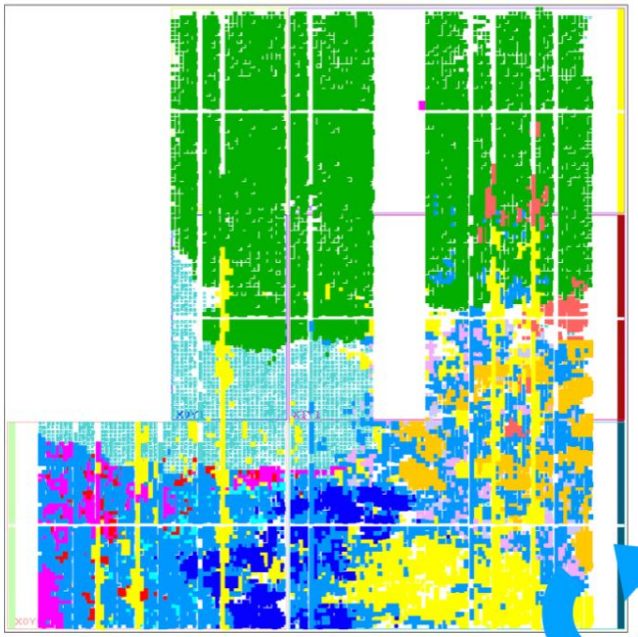
MaxPool2D

Flatten

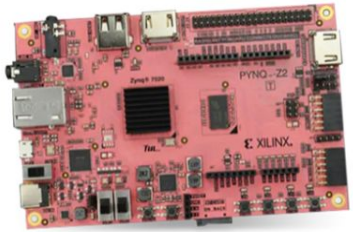
Dense

Softmax

Prediction



FIFOs



How are FPGAs programmed?

Hardware Description Languages

HDLs are programming languages which describe electronic circuits

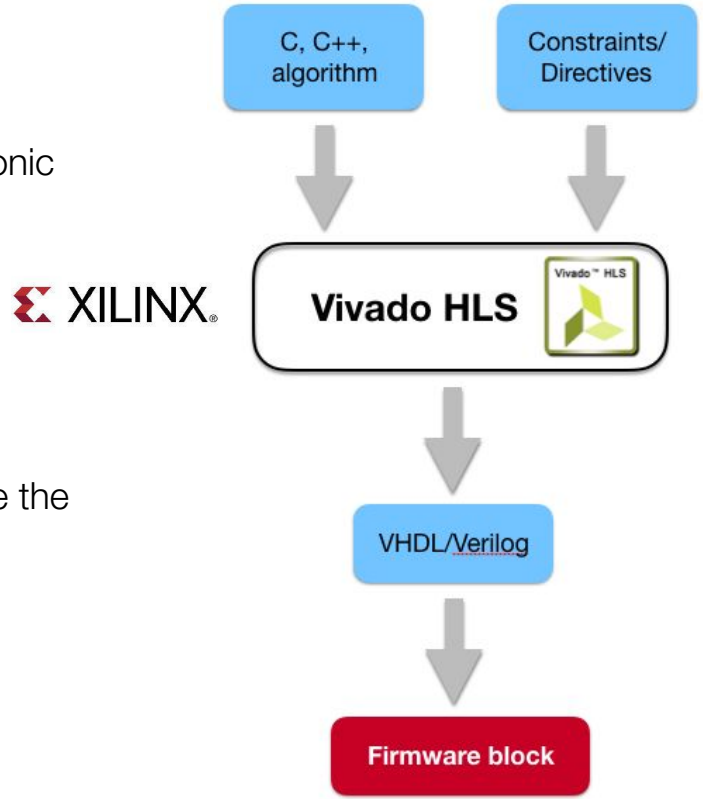
High Level Synthesis

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/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf

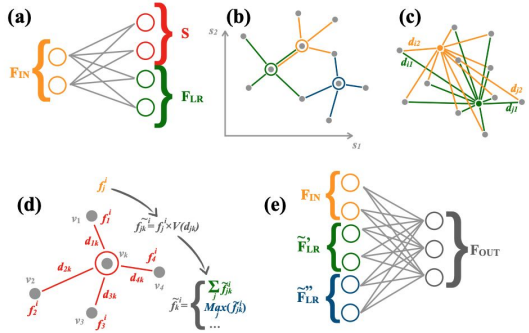
Jargon

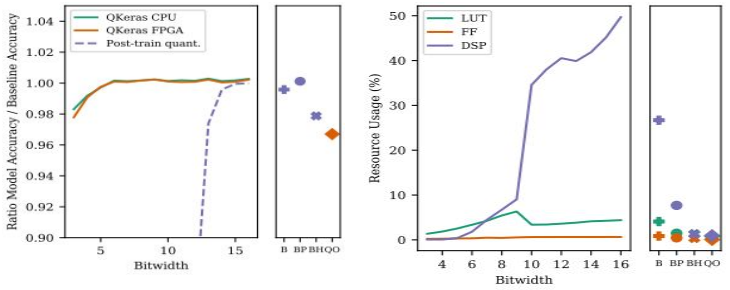
- **LUT - Look Up Table aka ‘logic’** - generic functions on small bitwidth inputs. Combine many to build the algorithm
- **FF - Flip Flops** - control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- **DSP - Digital Signal Processor** - performs multiplication 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
- **II - Initiation Interval** - time from accepting first input to accepting next input

Recent Developments

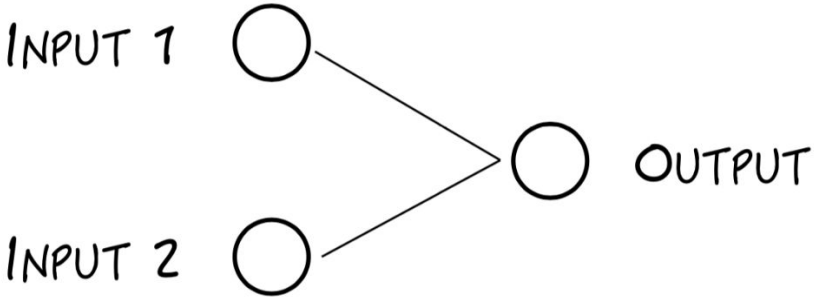
hls4ml community is very active!

- Quantization aware training QKeras + support in hls4ml: [arXiv: 2006.10159]
- Convolutional neural networks
  Mach. Learn.: Sci. Technol. 2 045015 (2021)
- Building accelerators (e.g. pynq, Alveo)
- Binary & Ternary neural networks:
  [2020 Mach. Learn.: Sci. Technol]
  - Compressed weights for low resource inference
- GarNet / GravNet: [arXiv: 2008.03601]
  - Distance weighted graph neural networks suitable for sparse/irregular point-cloud data





Neural network inference

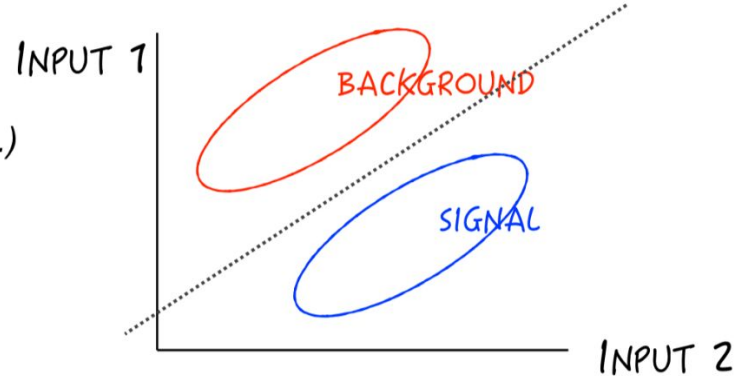


$$\vec{O}_j = \vec{I}_i \times \overleftrightarrow{W}_{ij} + \vec{b}_j$$

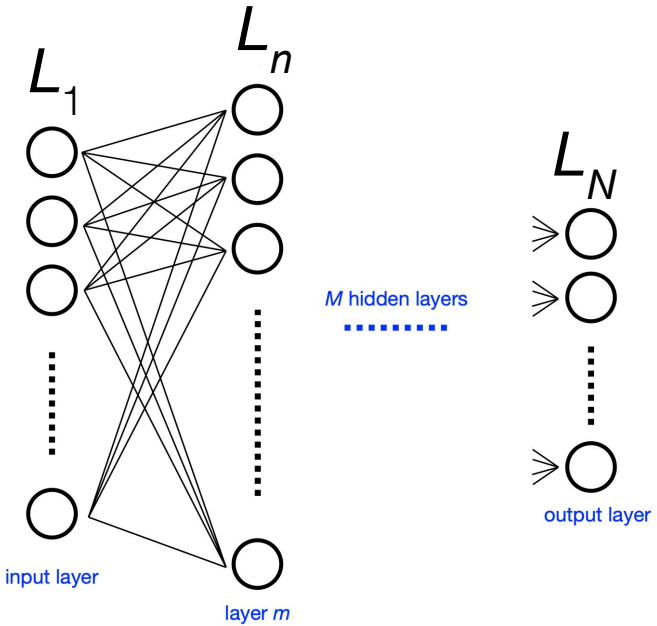
Simple 2 input example

(Fisher linear discriminant, linear support vector machine,…)

$$O_1 = I_1 \times W_{11} + I_2 \times W_{21} + b_1$$



Neural network inference



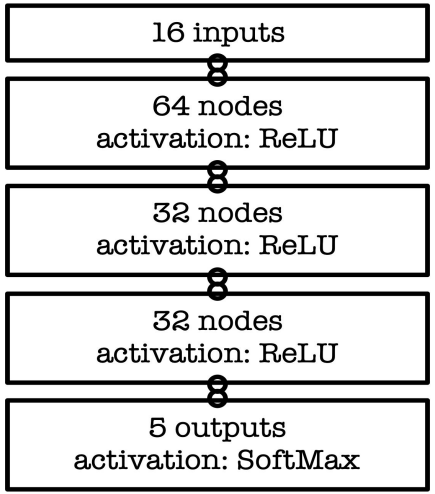
$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1} + \mathbf{b}_n)$$

precomputed and stored in BRAMs

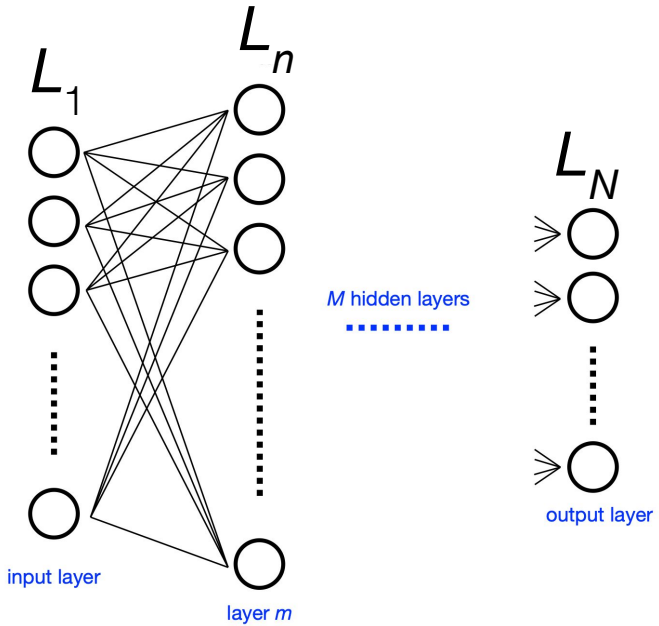
DSPs

logic cells

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



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

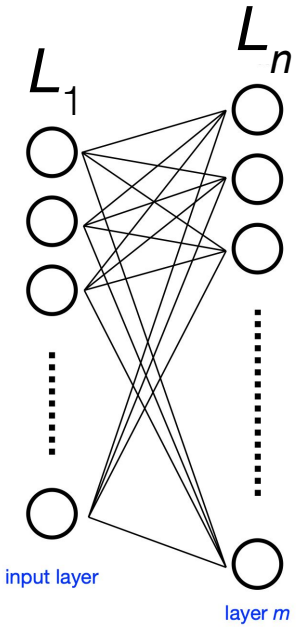
$$N_{\text{multiplications}} = \sum_{n=2}^{N} L_{n-1} \times L_n$$

16 inputs

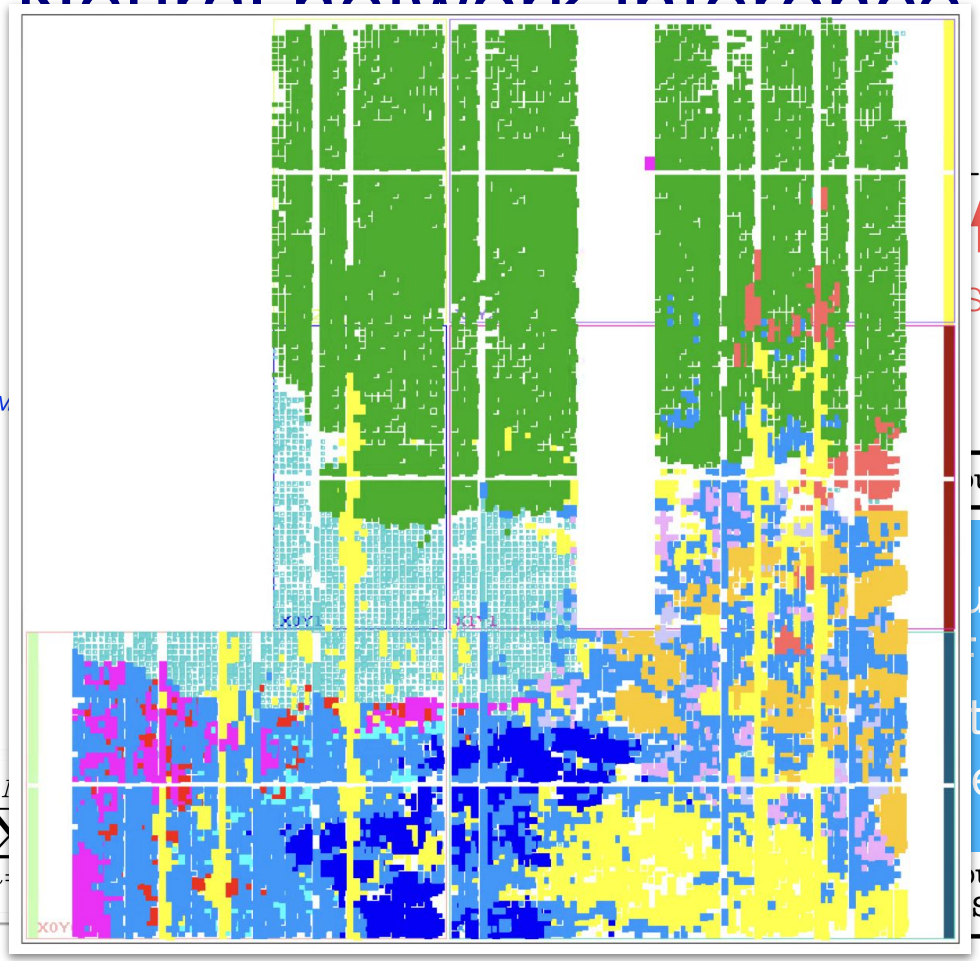
How many resources? DSPs, LUTs, FFs?
Does the model fit in the latency requirements?

5 outputs
activation: SoftMax

Neural network inference



$$N_{\text{multiplications}} = \sum_{n=}$$



$$\cdot 1\mathbf{x}_{n-1} + \mathbf{b}_n)$$

SPs

logic cells

uts

urces? DSPs,
FFs?
t in the latency
ents?

uts
SoftMax

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

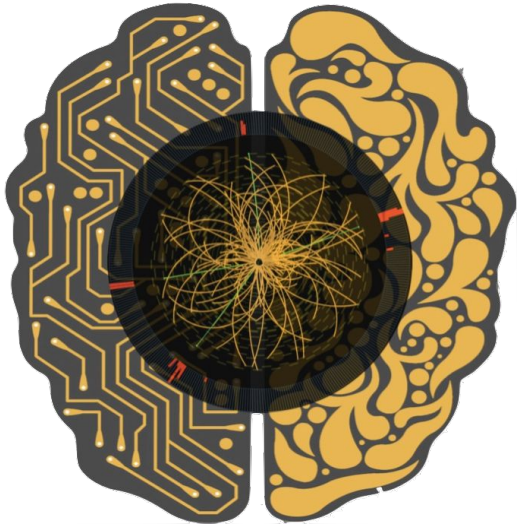
NN training

FPGA project designing

Today's **hls4ml** hands on

- Part 1:
  - Get started with hls4ml: train a basic model and run the conversion, simulation & synthesis steps

- Part 4:
  - Train using Quantization Aware Training using QKeras and study impact on FPGA metrics



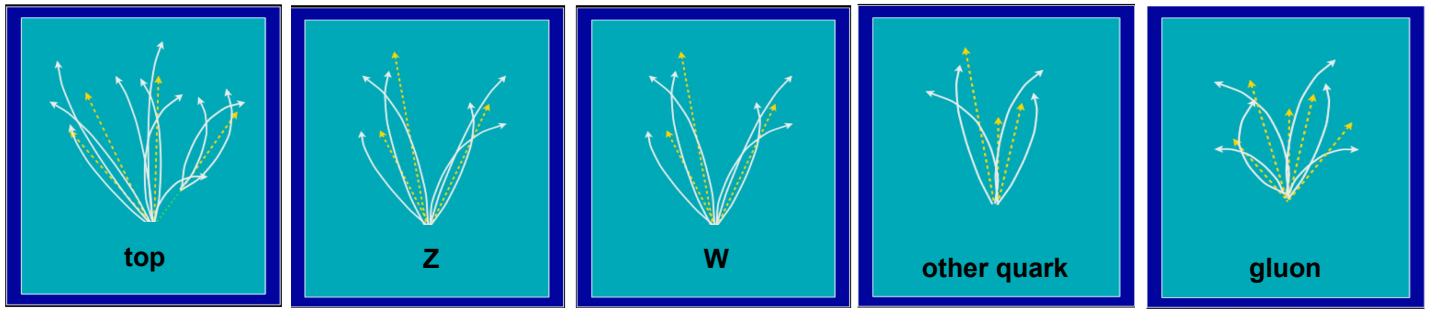
hls4ml tutorial

Part 1: Model Conversion

Dataset: 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 from hadronization of quarks and gluons



t→bW→bqq 3-prong jet

Z→qq 2-prong jet

W→qq 2-prong jet

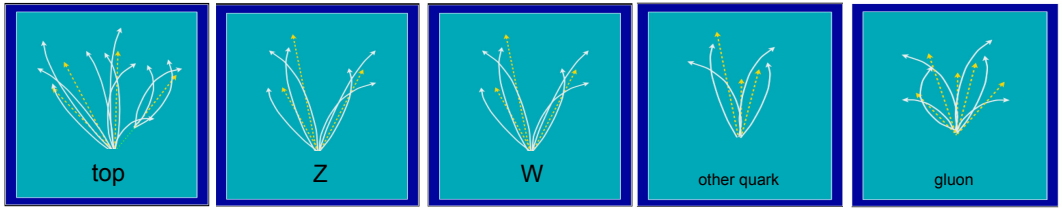
Reconstructed as one massive jet with substructure

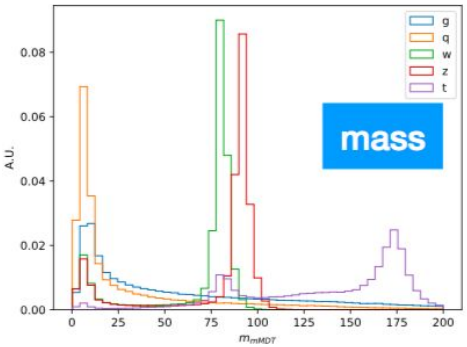
q/g background

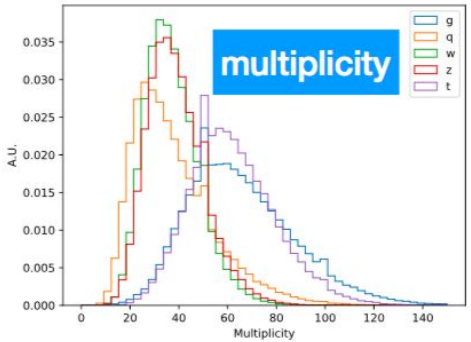
no substructure and/or mass ~ 0

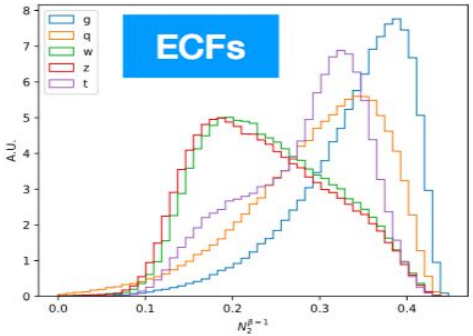
Dataset: jet tagging

Classes:









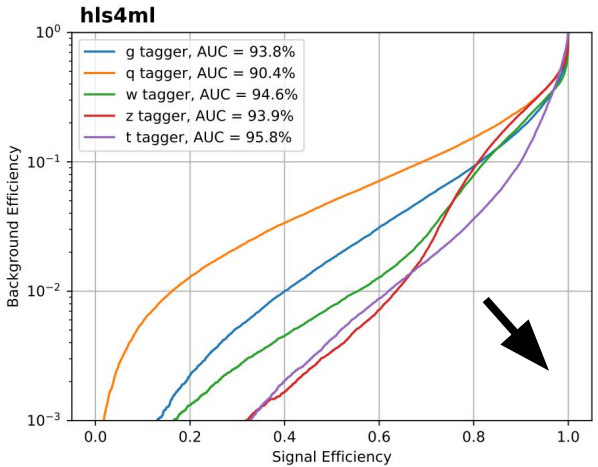
m_{mMDT}
$N_2^{\beta=1,2}$
$M_2^{\beta=1,2}$
$C_1^{\beta=0,1,2}$
$C_2^{\beta=1,2}$
$D_2^{\beta=1,2}$
$D_2^{(\alpha,\beta)=(1,1),(1,2)}$
$\sum z \log z$
Multiplicity

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

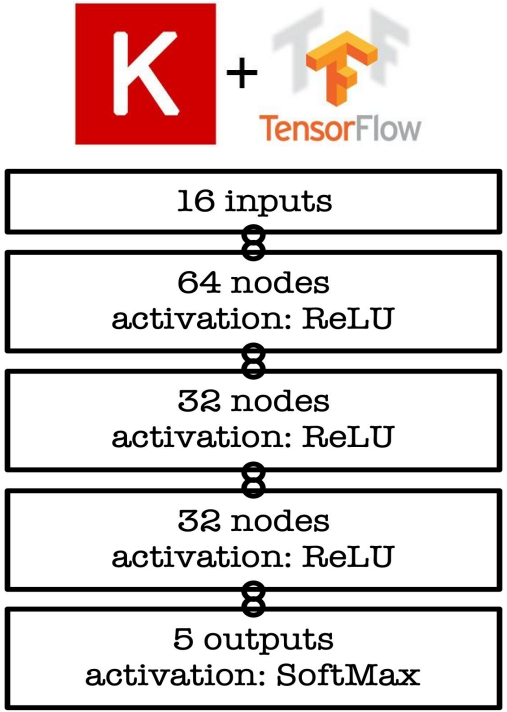
[*] E. Coleman *et al.* JINST.2018.T01003, D. Guest *et al.* PhysRevD.94.112002, G. Kasieczka *et al.* JHEP05(2017)006, J. M. Butterworth *et al.* PhysRevLett.100.242001

Dataset: 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

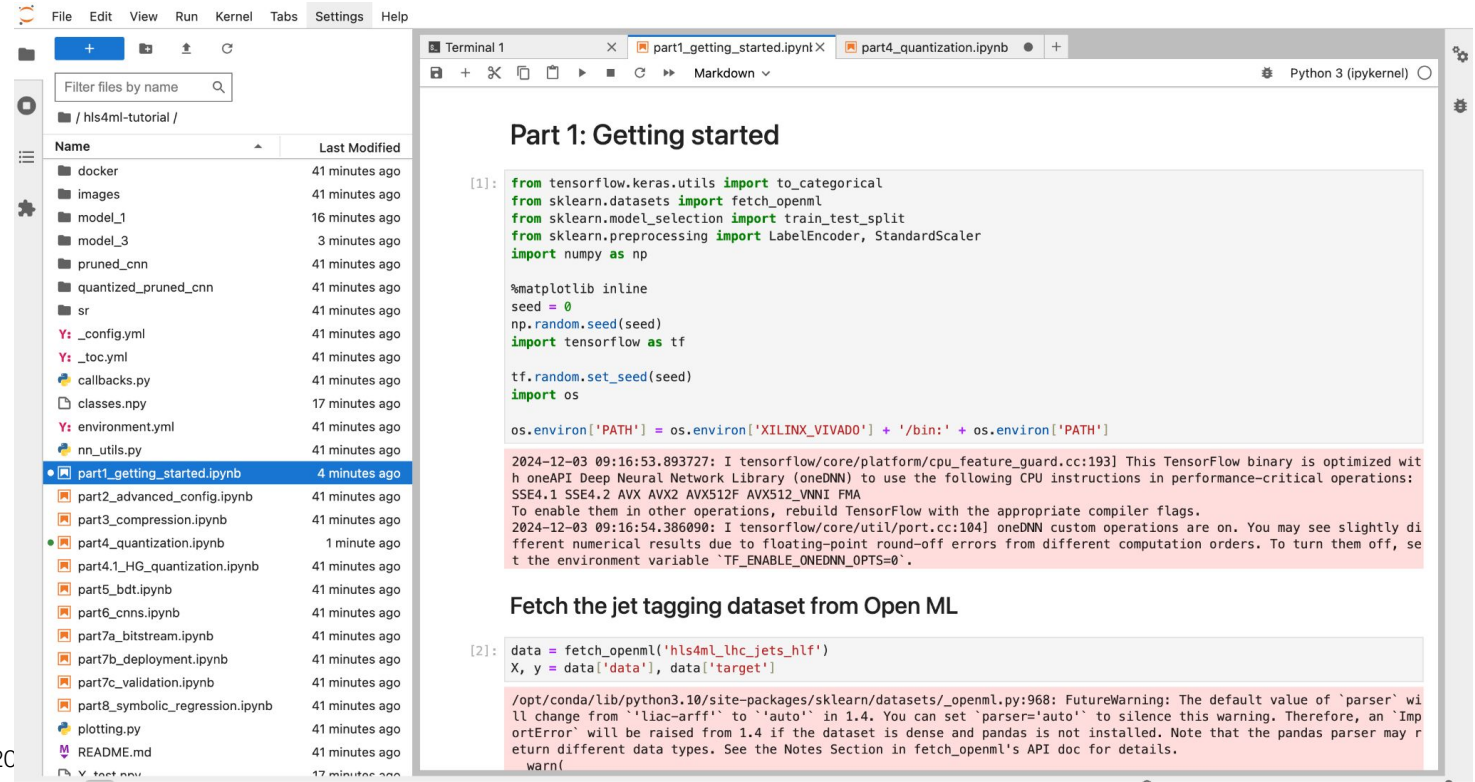


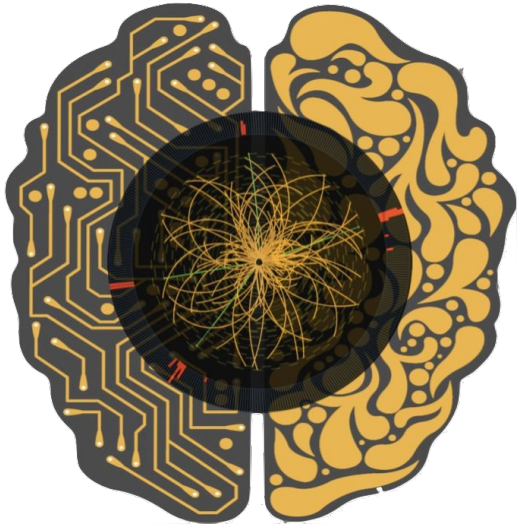
better



Hands On - Setup

- Open a web browser, go to https://ssummers.web.cern.ch/hls4ml-tutorial.php
- Use your GitHub account to authenticate



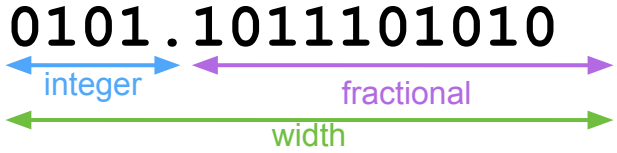


hls4ml Tutorial

Part 2: Advanced Configuration

Efficient NN design: quantization

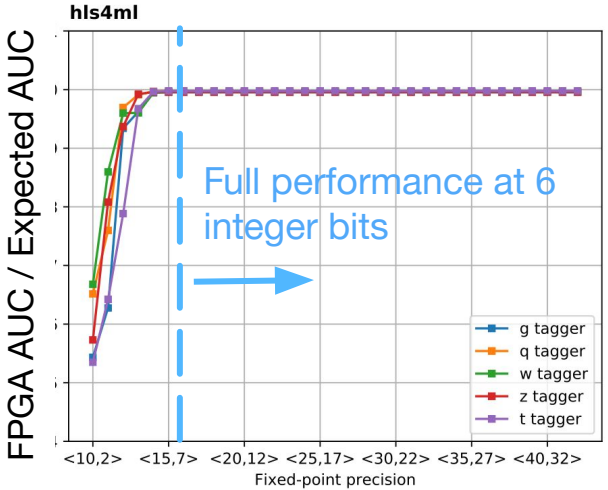
ap_fixed<width bits, integer bits>



- 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!

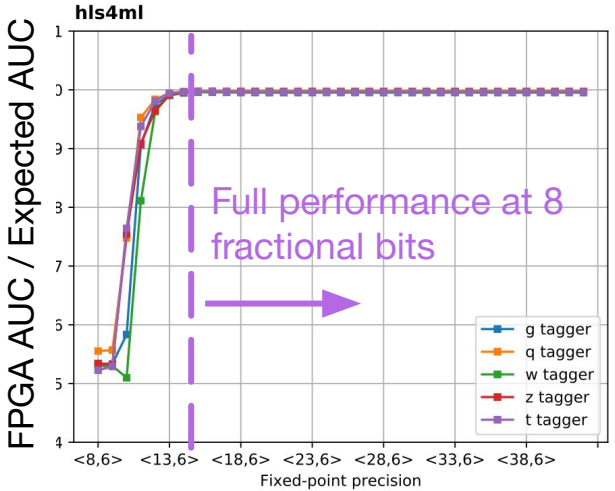
Scan integer bits

Fractional bits fixed to 8



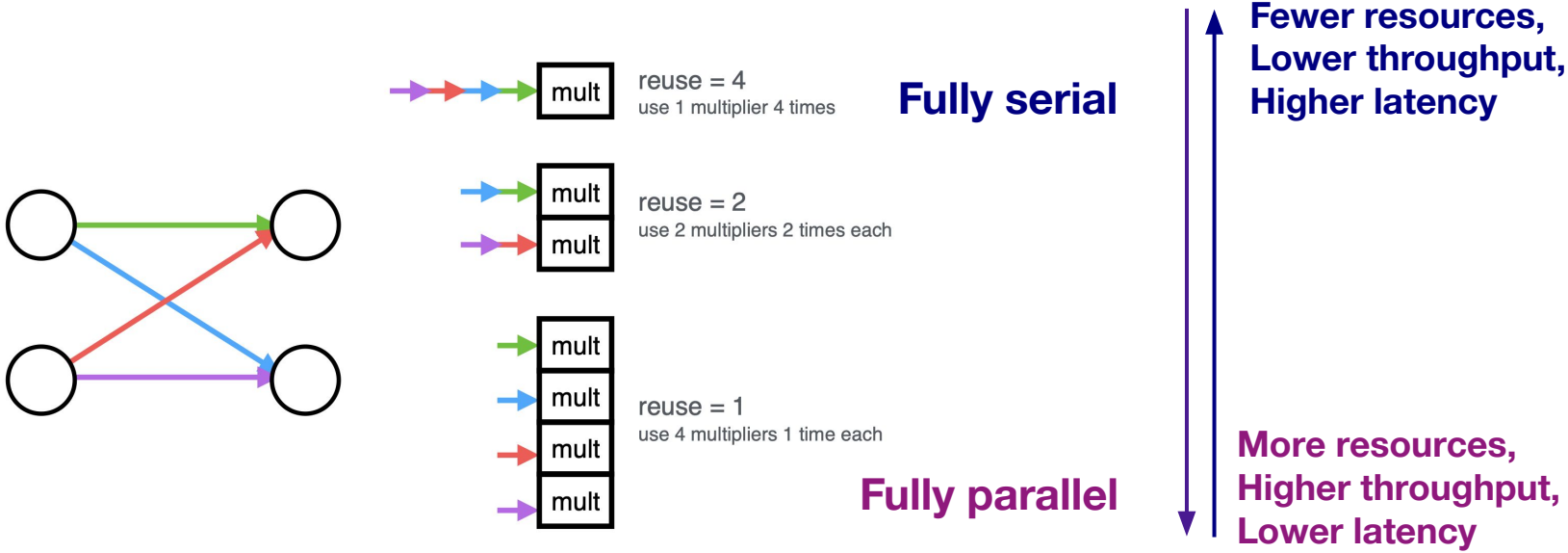
Scan fractional bits

Integer bits fixed to 6



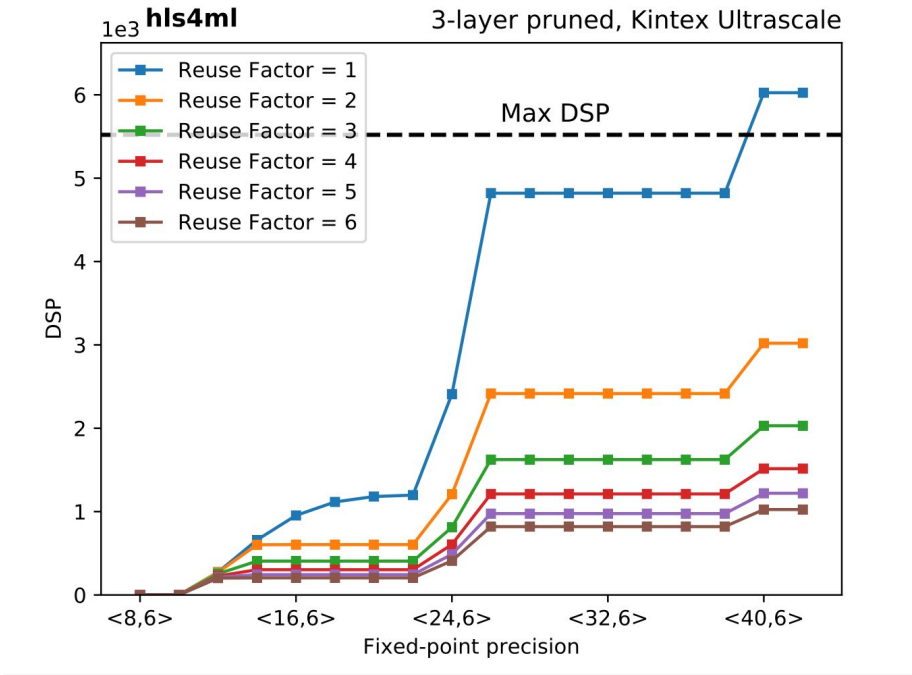
Efficient NN design: parallelization

- 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: resource usage



More resources

Fully parallel
Each mult. used 1x

Each mult. used 2x

Each mult. used 3x

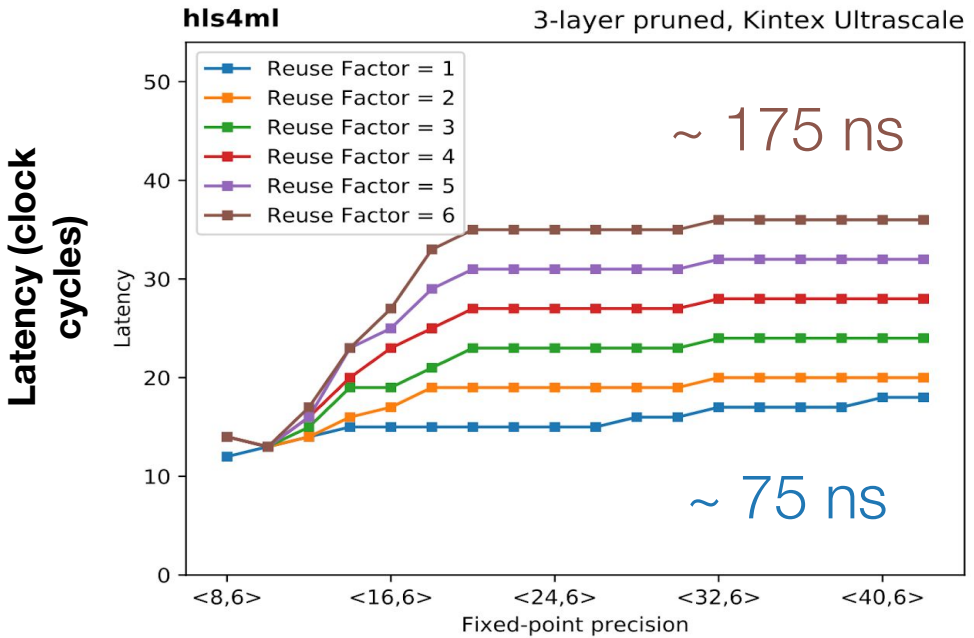
⋮

Longer latency

Parallelization → speed

Latency of layer m

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



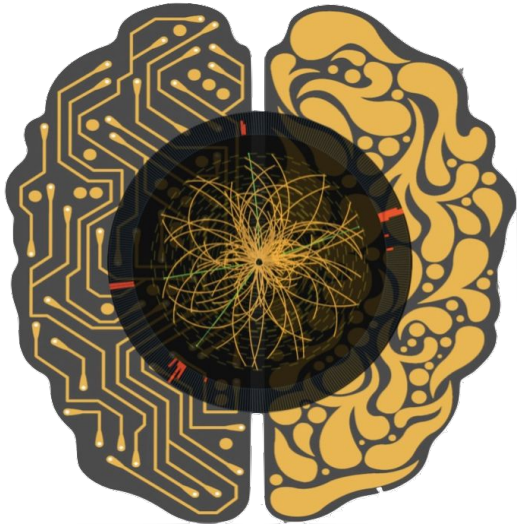
Each multiplier used 6x

Each multiplier used 3x

Each multiplier used 1x

Longer latency

More resources

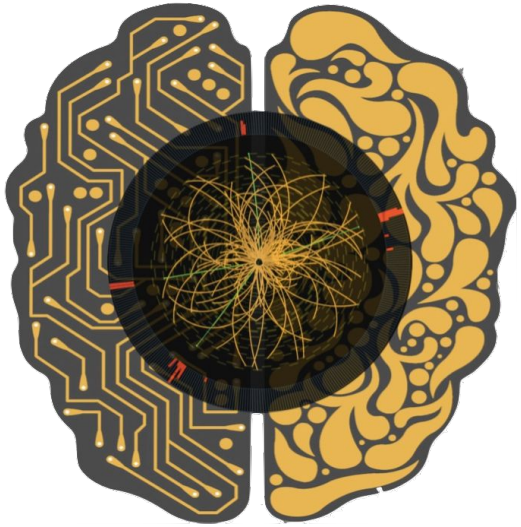


hls4ml Tutorial

Part 3: Compression

NN compression methods

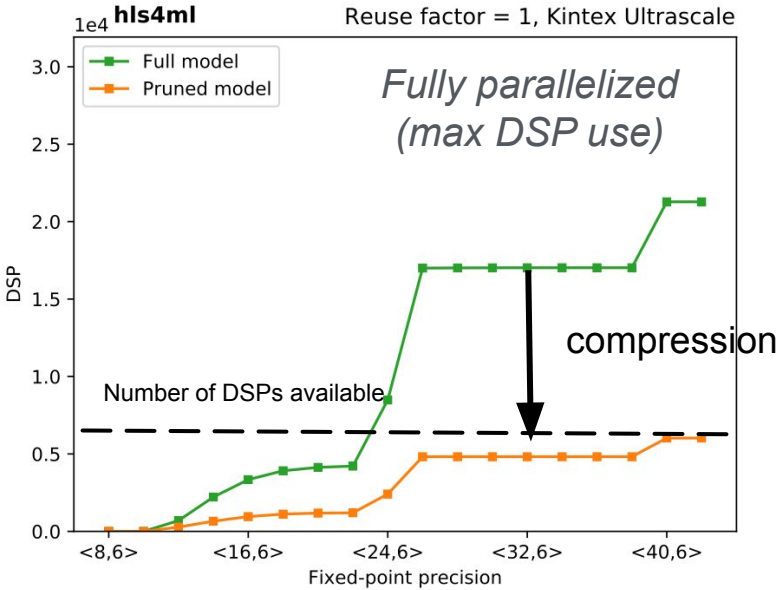
- Network compression is a widespread technique to reduce the size, energy consumption, and overtraining of deep neural networks
- Several approaches have been studied:
  - **parameter pruning:** selective removal of weights based on a particular ranking [arxiv.1510.00149, arxiv.1712.01312]
  - **low-rank 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!



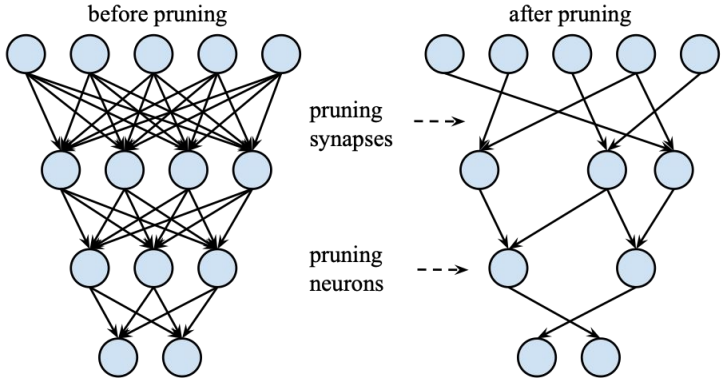
hls4ml Tutorial

Part 4: Quantization and Compression

Efficient NN design: compression



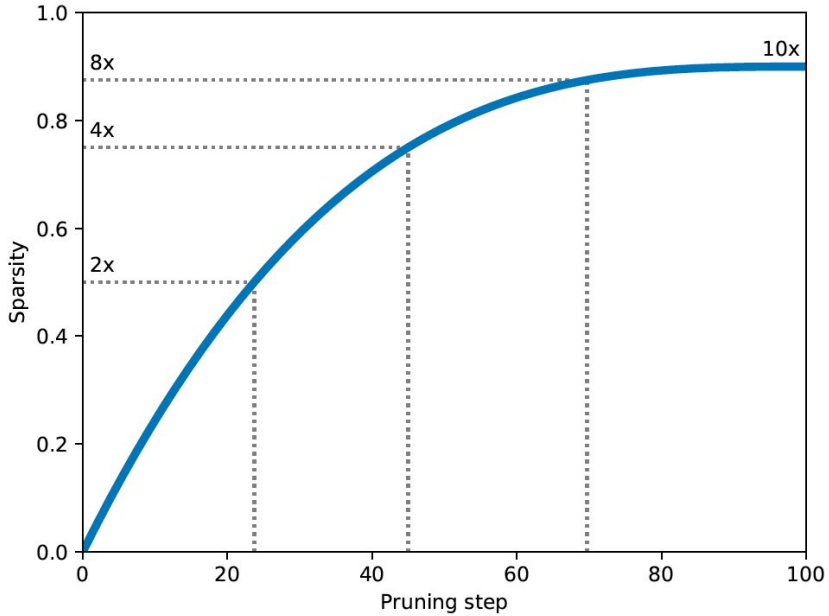
70% compression ~ 70% fewer DSPs



- 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

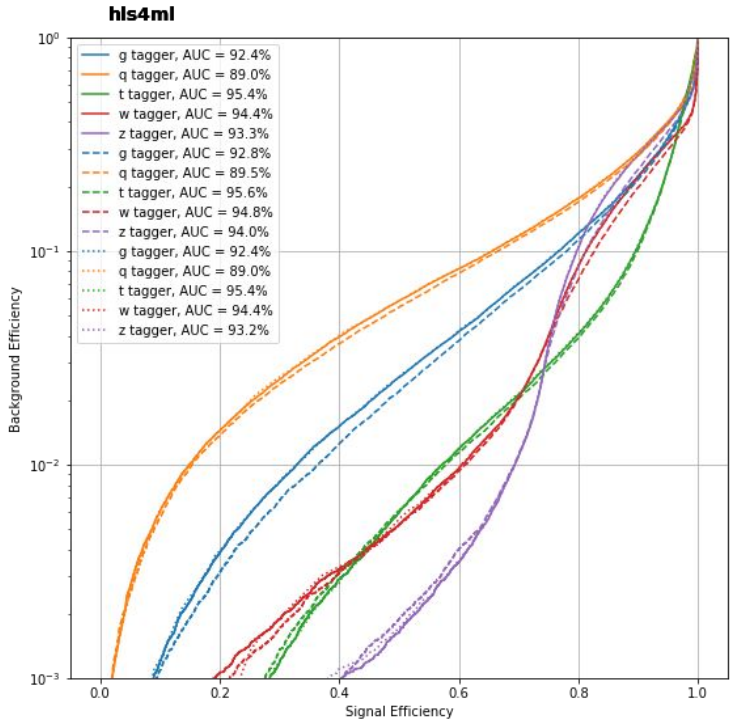
TF Sparsity

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



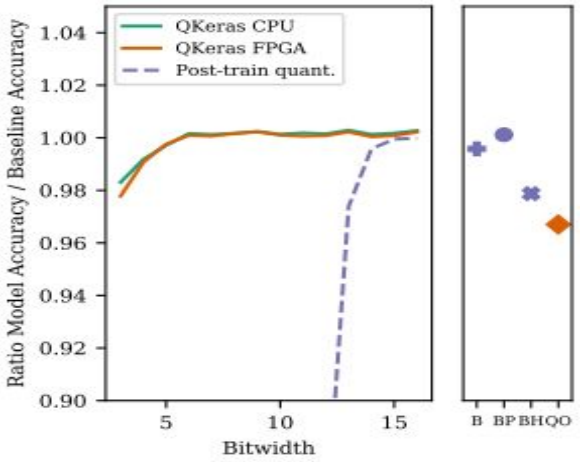
Efficient NN design: quantization

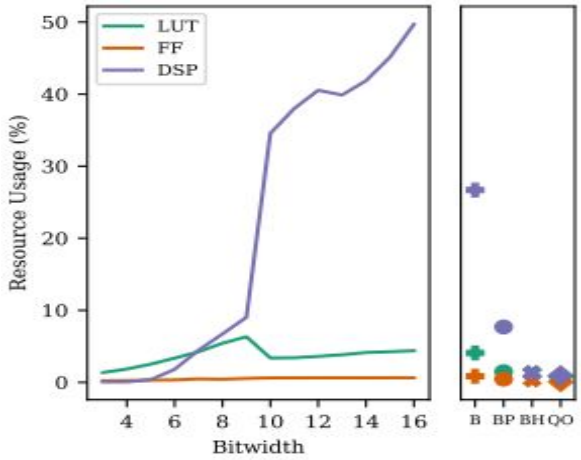
- hls4ml allows you to use different data types everywhere, we saw how to tune that in part 2
- We will also try quantization-aware training with QKeras (part 4)
- With quantization-aware we can even go down to just 1 or 2 bits
    - See our recent work: https://arxiv.org/abs/2003.06308



QKeras

- QKeras is a library to train models with quantization in the training
  - Developed & 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
  - And reduce resource usage easily!
- hls4ml supports QKeras and QONNX formats
  - The number of bits used in training is also used in inference





Summary

- After this session you've gained some hands on experience with **hls4ml**
  - Fast Inference of Neural Networks in FPGAs
  - Translated neural networks to FPGA firmware, run simulation and synthesis
  - Learned how to prune and quantize Neural Networks with with QKeras

- You can find these tutorial notebooks to run locally at: https://github.com/fastmachinelearning/hls4ml-tutorial

- Use **hls4ml** in your own environment: `pip install hls4ml`