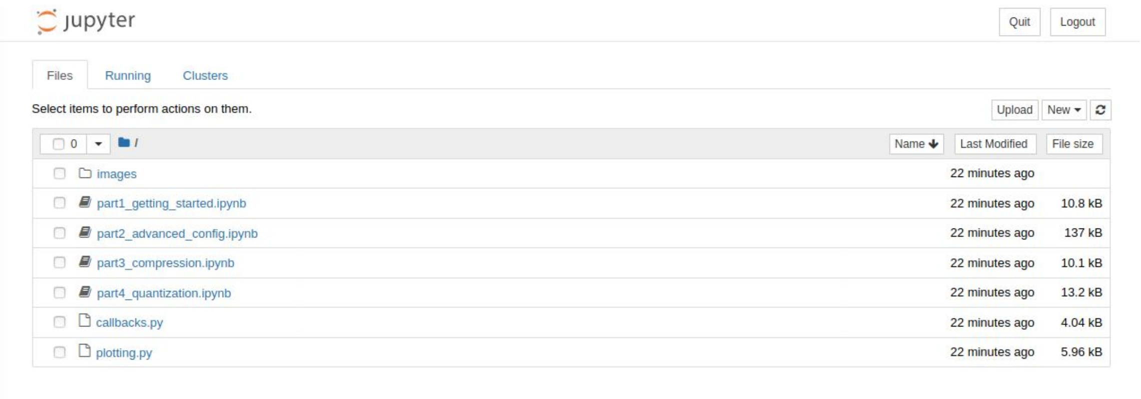
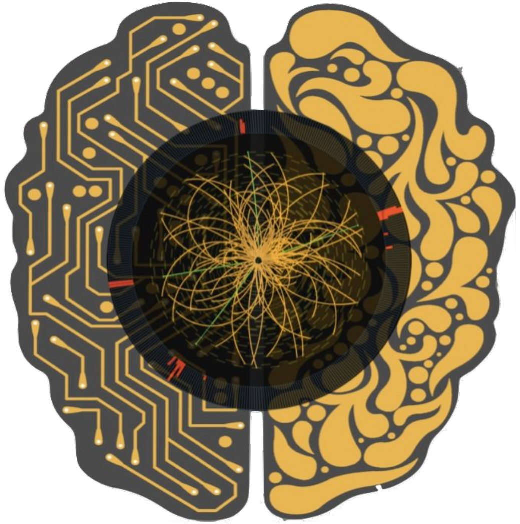
Hands On - Setup

- The interactive part is served with Python notebooks
- Open https://cern.ch/ssummers/hls4ml-tutorial in your web browser
- Authenticate with your Github account (login if necessary)
- If you're new to Jupyter notebooks, select a cell and hit "shift + enter" to execute the code
- If you have Vivado install yourself, you might prefer to work locally, see 'conda' section at: https://github.com/fastmachinelearning/hls4ml-tutorial



Select items to perform actions on them.

Name	Last Modified	File size
images	22 minutes ago	
part1_getting_started.ipynb	22 minutes ago	10.8 kB
part2_advanced_config.ipynb	22 minutes ago	137 kB
part3_compression.ipynb	22 minutes ago	10.1 kB
part4_quantization.ipynb	22 minutes ago	13.2 kB
callbacks.py	22 minutes ago	4.04 kB
plotting.py	22 minutes ago	5.96 kB



hls4ml tutorial

FastML Workshop 2020

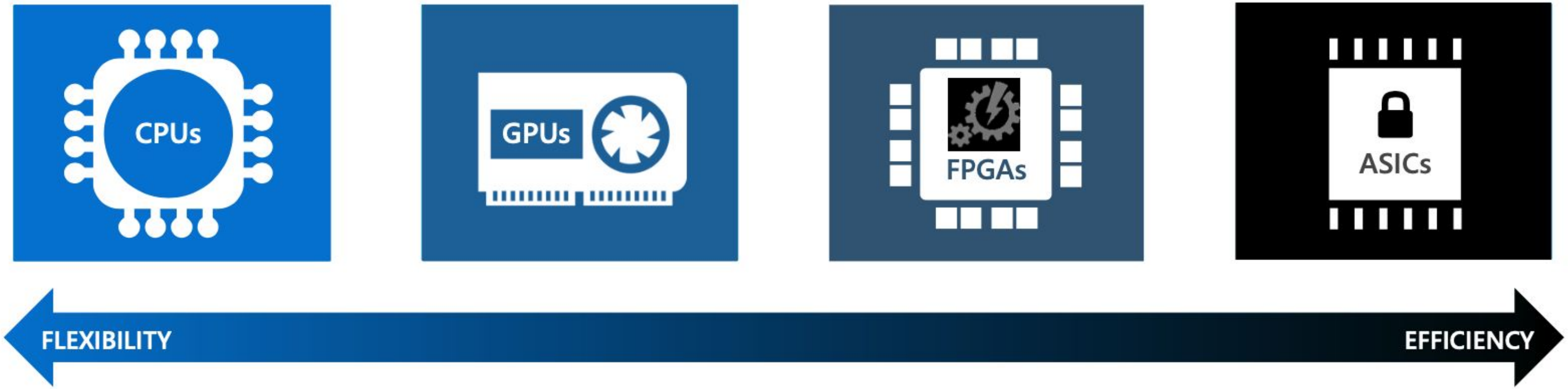
Sioni Summers et al. for the hls4ml team

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`

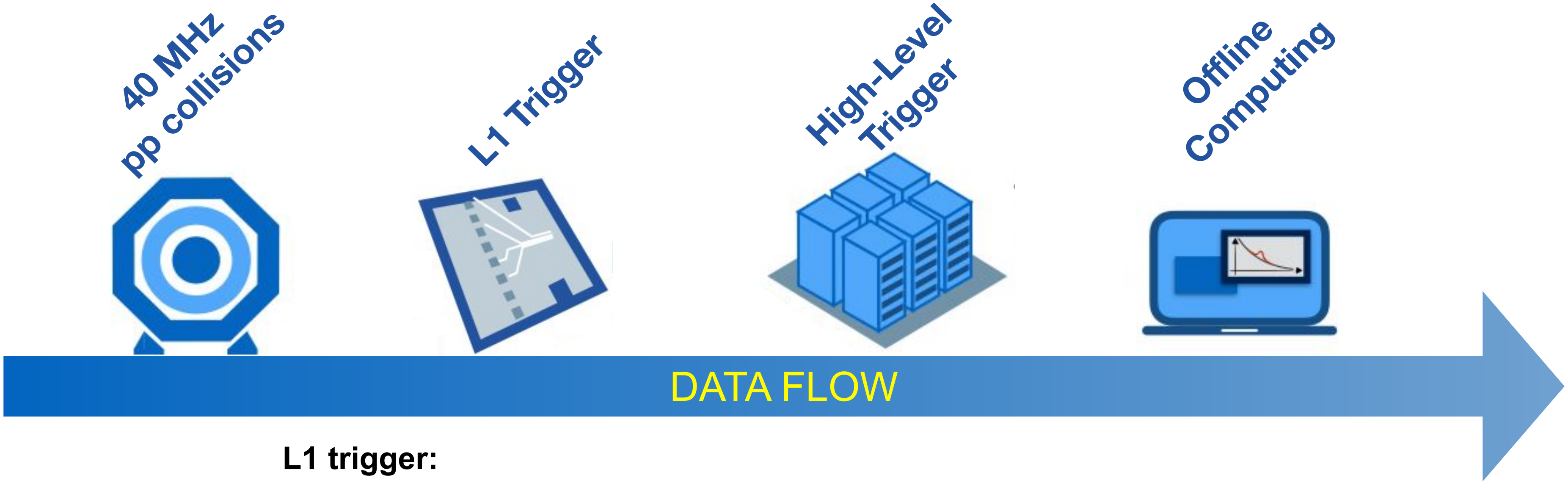
- In this session you will get hands on experience with the **hls4ml** package
- We'll learn how to:
    - Translate models into synthesizable FPGA code
    - Explore the different handles provided by the tool to optimize the inference
        - Latency, throughput, resource usage
    - Make our inference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

Why FPGAs?



Guidelines:
> 100 Gbps throughput
< 1ms computational latency
< 10W power budget

LHC Experiment Data Flow



40 MHz pp collisions

L1 Trigger

High-Level Trigger

Offline Computing

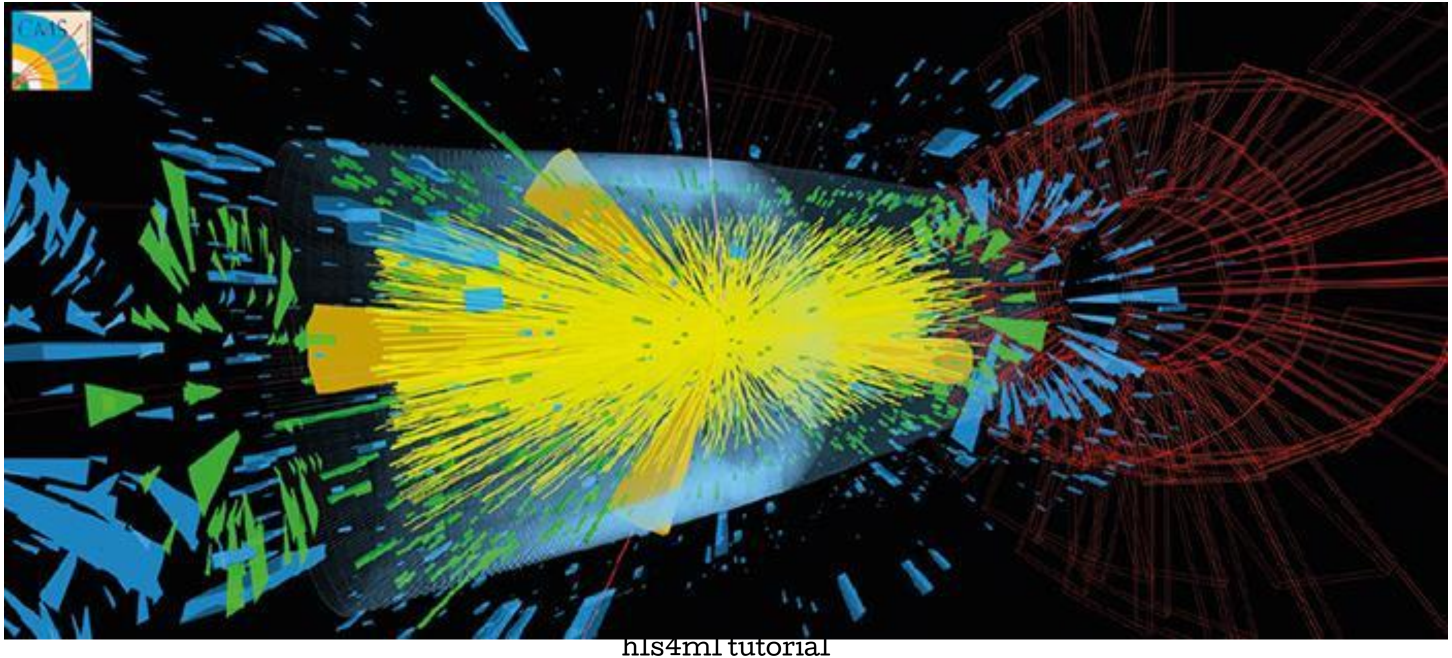
DATA FLOW

L1 trigger:

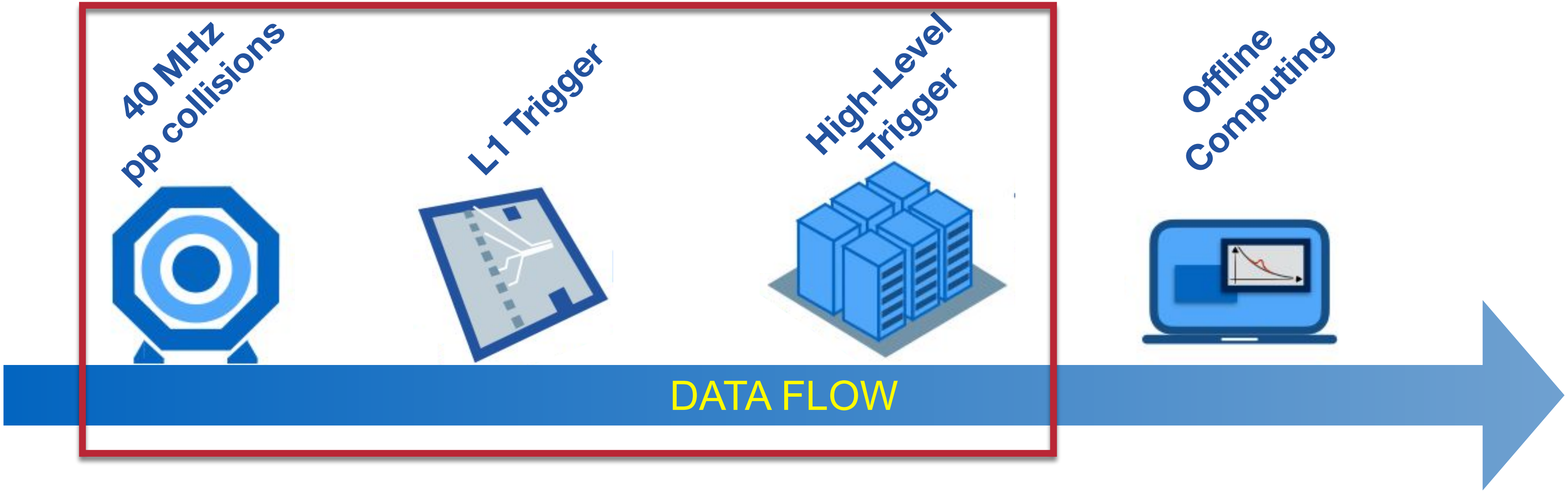
- 40 MHz in / 100 KHz out
- Process 100s TB/s
- Trigger decision to be made in ≈ **10 μs**
- Coarse local reconstruction
- FPGAs / Hardware implemented

hls4ml origins: triggering at (HL-)LHC

Extreme collision frequency of 40 MHz → extreme data rates O(100 TB/s)
Most collision “events” don’t produce interesting physics
“Triggering” = filter events to reduce data rates to manageable levels



LHC Experiment Data Flow



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

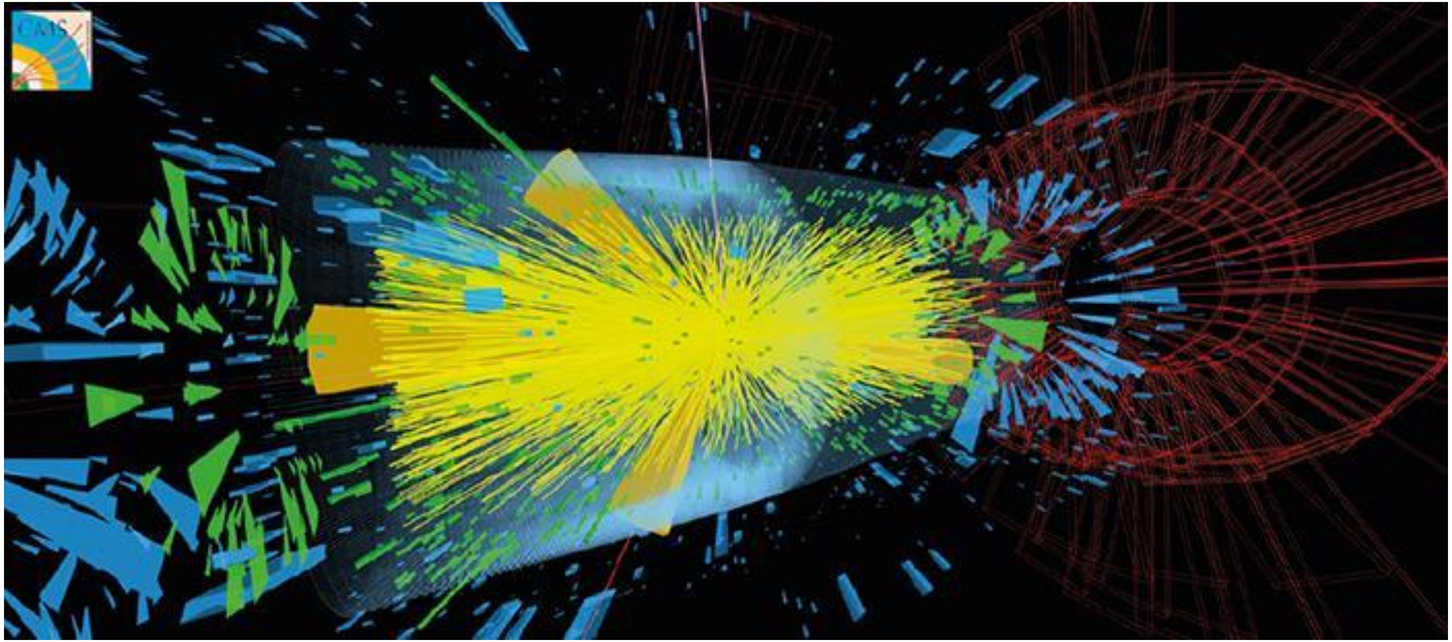
The challenge: triggering at (HL-)LHC

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

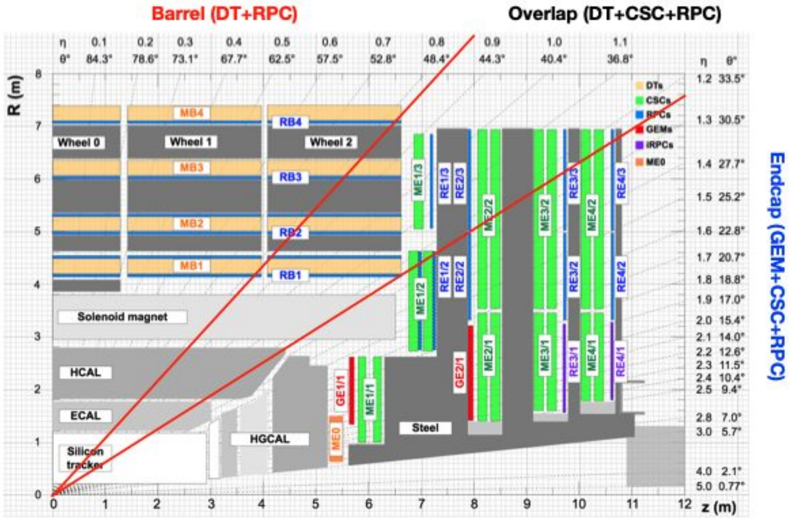
ML can improve sensitivity to rare physics

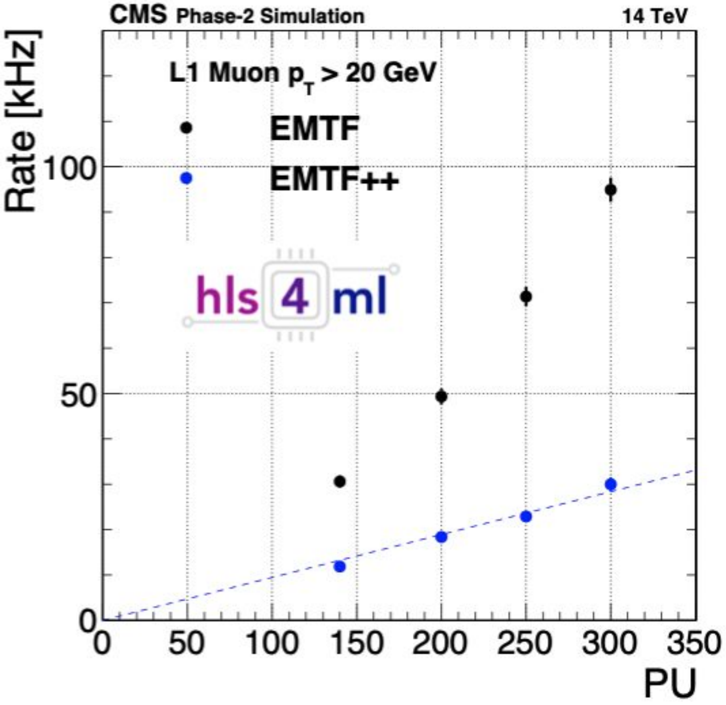
Needs to be ***fast!***

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



Muon trigger example





EMTF = BDT (external memory)

EMTF++ = NN

~3x reduction in the trigger rate for neural network!

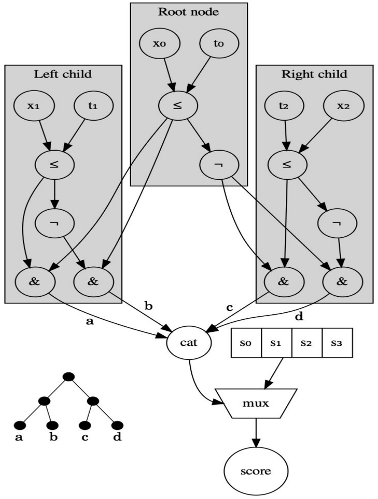
hls4ml: progression

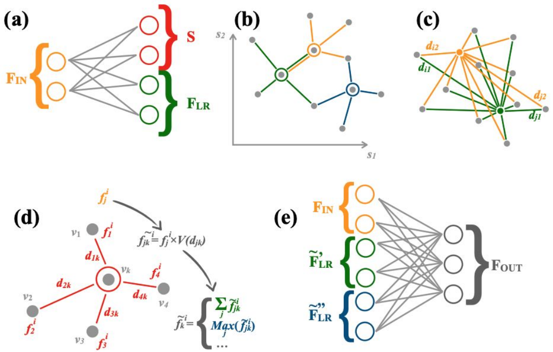
- Previous slides showed the original motivation for hls4ml
  - Extreme low latency, high throughput domain
- Since then, we have been expanding!
  - Longer latency domains, larger models, resource constrained
  - Different FPGA vendors
  - New applications, new architectures
- While maintaining core 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!

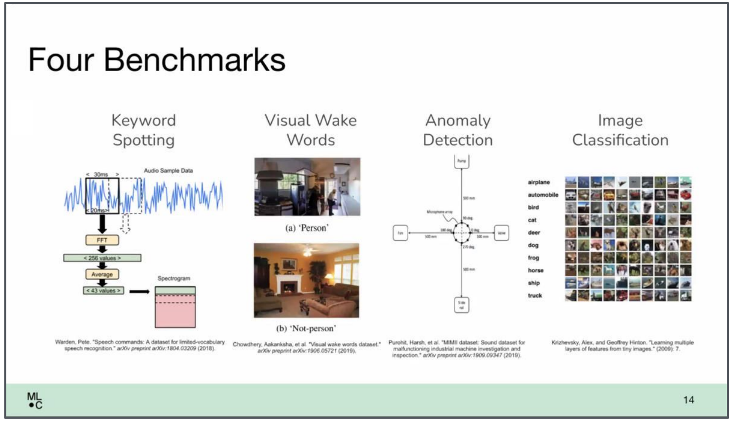
- Binary & Ternary neural networks:
  [2020 Mach. Learn.: Sci. Technol]
  - Compressed weights for low resource inference
- Boosted Decision Trees: [JINST 15 P05026 (2020)]
  - Low latency for Decision Tree ensembles
- GarNet / GravNet: [arXiv: 2008.03601]
  - Distance weighted graph neural networks suitable for sparse/irregular point-cloud data
- Quantization aware training QKeras + support in hls4ml: [arXiv: 2006.10159]
- Convolutional neural networks
  Mach. Learn.: Sci. Technol. 2 045015 (2021)





Coming Soon

- A few exciting new things are being developed and should become available soon:
  - Intel Quartus HLS, Mentor Catapult HLS, Intel One API 'Backends'
  - Recurrent Neural Networks
  - More integrated 'end-to-end' flow with bitfile generation and host bindings for platforms like Alveo, PYNQ
    - Bundled into MLCommons Tiny submission -- image classification and anomaly detection https://mlcommons.org/en/news/mlperf-tiny-v05/



Four Benchmarks

Warden, Pete. "Speech commands: A dataset for limited-vocabulary speech recognition." *arXiv preprint arXiv:1804.03209* (2018).

Chowdhery, Aakanksha, et al. "Visual wake words dataset." *arXiv preprint arXiv:1906.05721* (2019).

Purohit, Harsh, et al. "MIMII dataset: Sound dataset for malfunctioning industrial machine investigation and inspection." *arXiv preprint arXiv:1909.09347* (2019).

Krizhevsky, Alex, and Geoffrey Hinton. "Learning multiple layers of features from tiny images." (2009): 7.

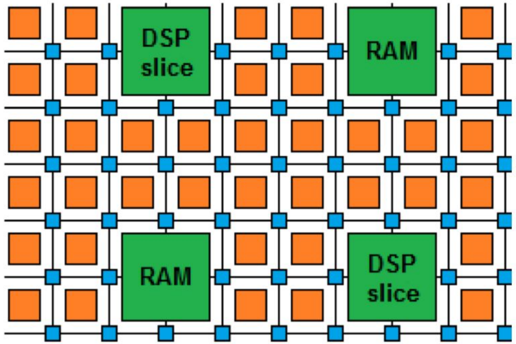
What are FPGAs?

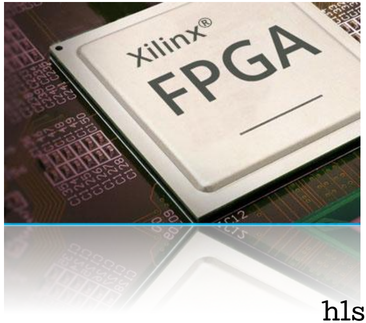
Field Programmable Gate Arrays are reprogrammable integrated circuits

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





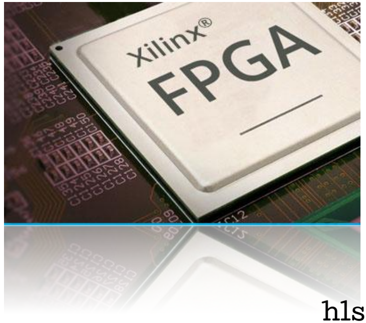
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

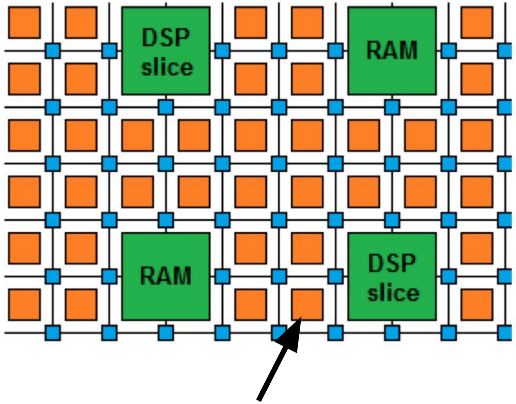
Logic cells / Look Up Tables perform arbitrary functions on small bitwidth inputs (2-6)

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

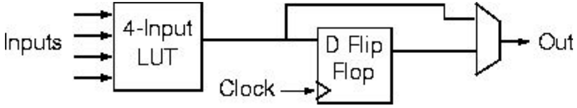
Flip-Flops register data in time with the clock pulse



FPGA diagram



Logic cell



Look-up table (logic)

Flip-flop (registers)

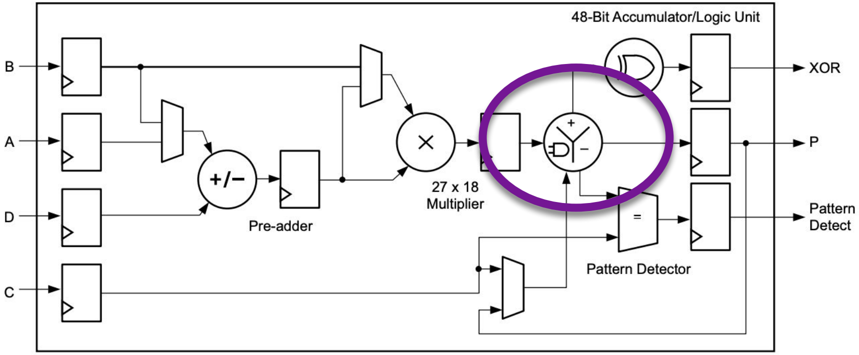
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

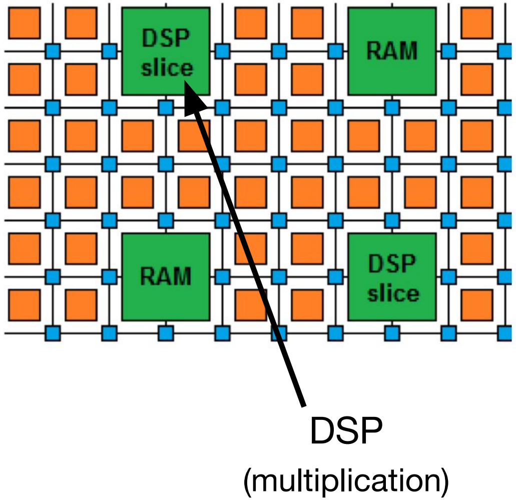
DSPs (Digital Signal Processor) are specialized units for multiplication and arithmetic

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

And for Neural Nets, DSPs are often the most scarce



FPGA diagram



What are FPGAs?

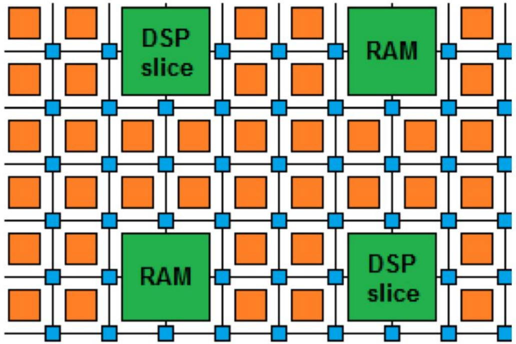
Field Programmable Gate Arrays are reprogrammable integrated circuits

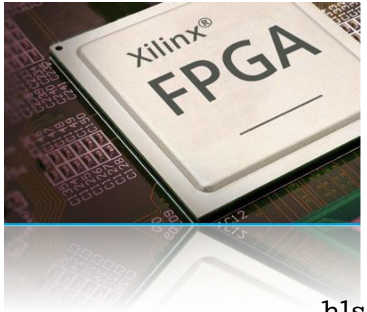
BRAMs are small, fast memories - RAMs, ROMs, FIFOs (18Kb each in Xilinx)

Memories using BRAMs more efficient than using LUTs

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

FPGA diagram





What are FPGAs?

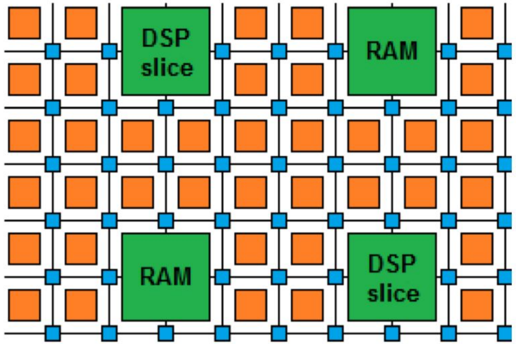
In addition, there are specialised blocks for I/O, making FPGAs popular in embedded systems and HEP triggers

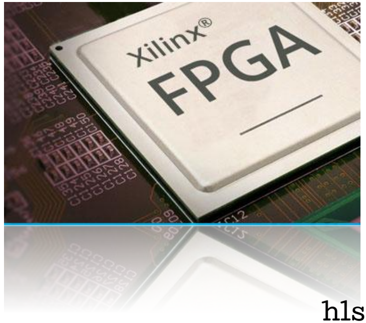
High speed transceivers with Tb/s total bandwidth
PCIe, (Multi) Gigabit Ethernet, Infiniband

AND: Support highly parallel algorithm implementations

Low power per Op (relative to CPU/GPU)

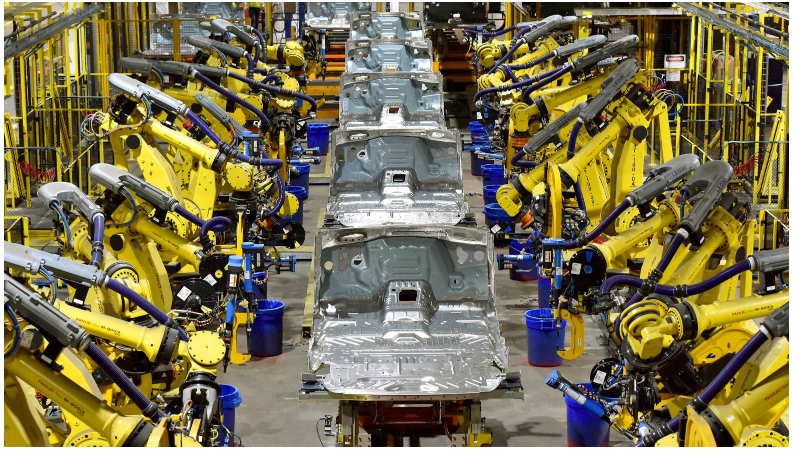
FPGA diagram





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…

How are FPGAs programmed?

Hardware Description Languages

HDLs are programming languages which describe electronic circuits

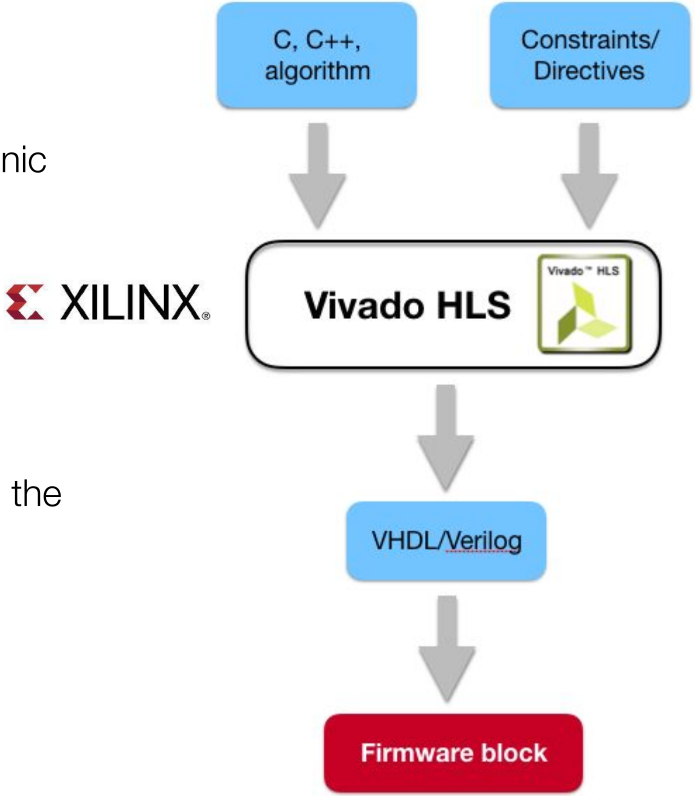
High Level Synthesis

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 [*]

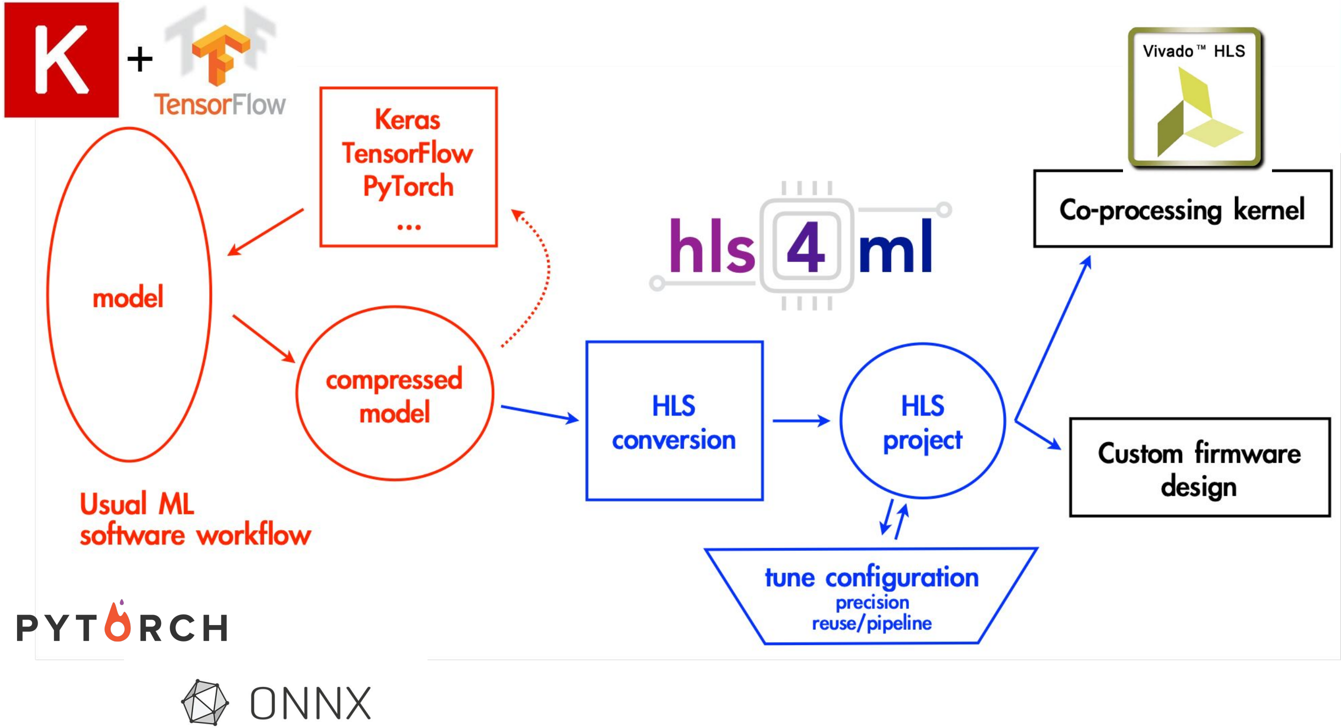


[*] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf

Jargon

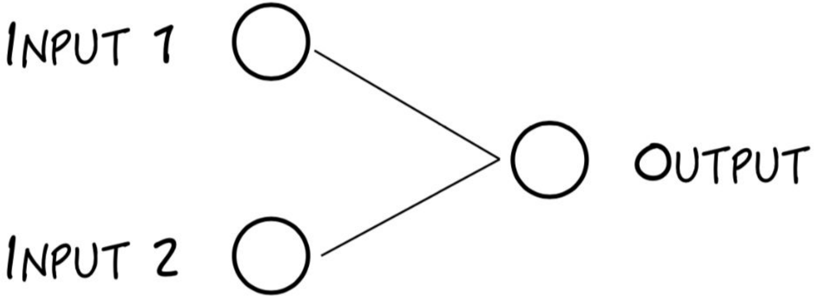
- **LUT - Look Up Table aka ‘logic’** - generic functions on small bitwidth inputs. Combine many to build the algorithm
- **FF - Flip Flops** - control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- **DSP - Digital Signal Processor** - performs multiplication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

high level synthesis for machine learning

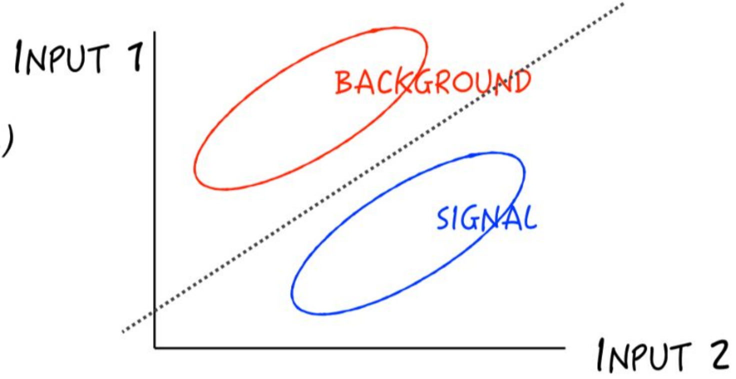


https://fastmachinelearning.org/hls4ml/

Neural network inference



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

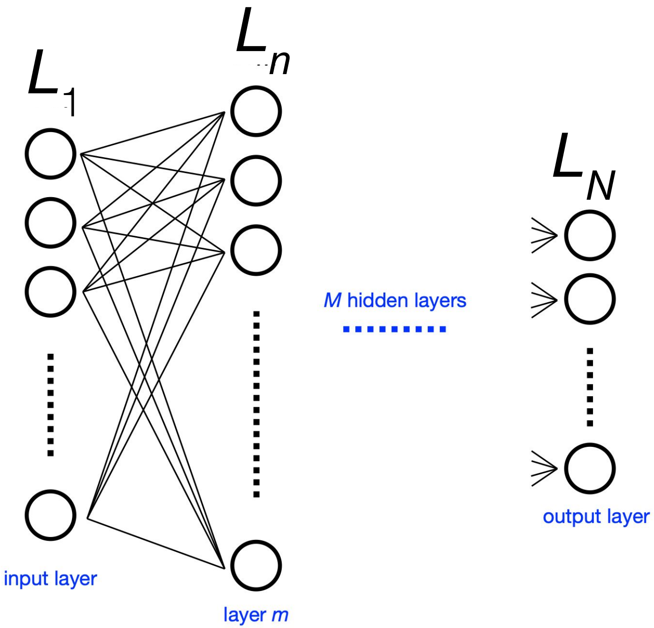


Simple 2 input example

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

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

Neural network inference



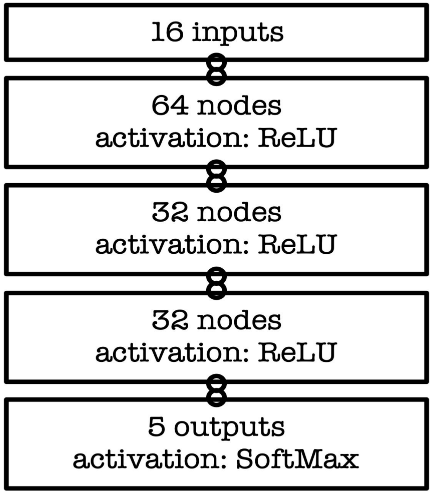
$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1} + \mathbf{b}_n)$$

precomputed and stored in BRAMs

DSPs

logic cells

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



16 inputs

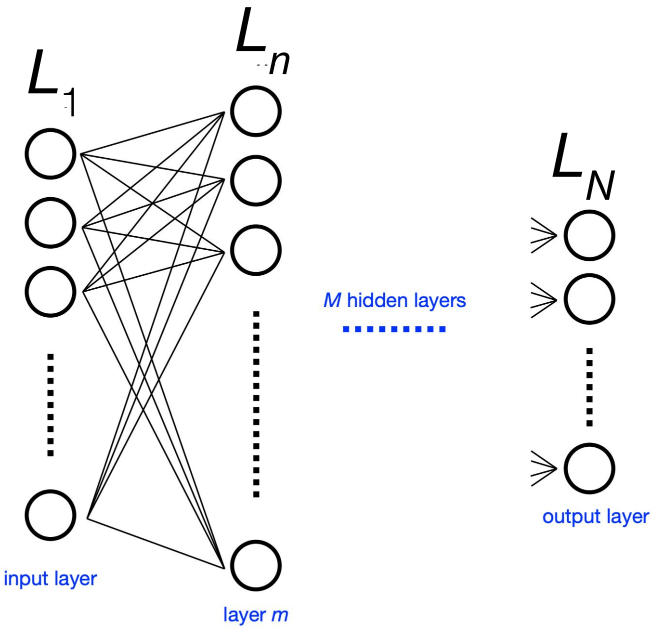
64 nodes
activation: ReLU

32 nodes
activation: ReLU

32 nodes
activation: ReLU

5 outputs
activation: SoftMax

Neural network inference



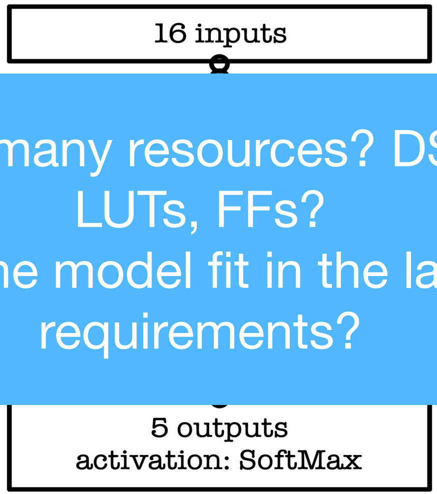
$$N_{\text{multiplications}} = \sum_{n=2}^{N} L_{n-1} \times L_n$$

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

precomputed and stored in BRAMs

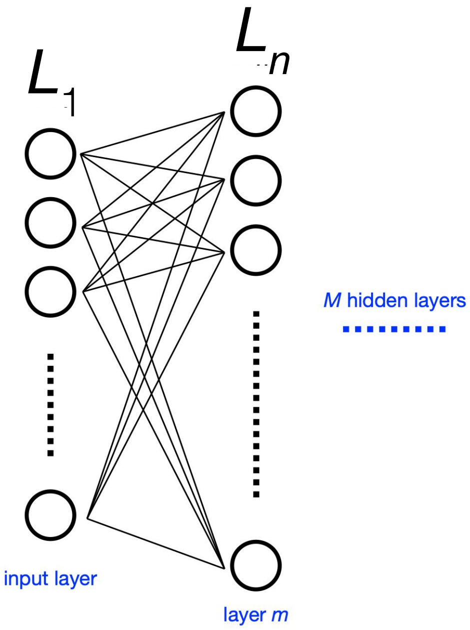
DSPs

logic cells

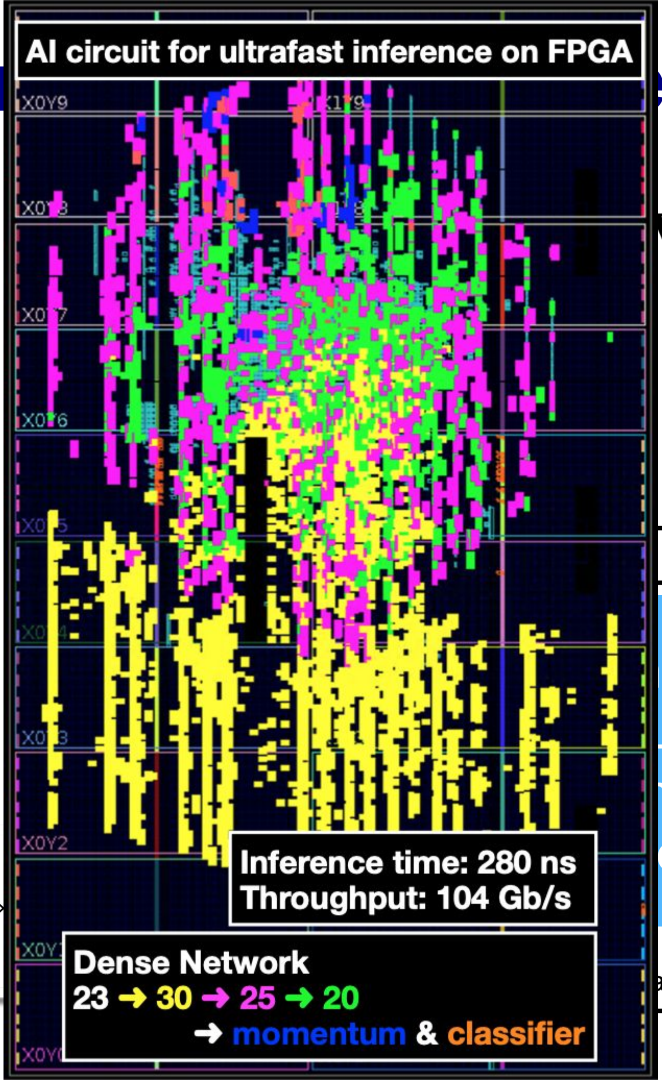


How many resources? DSPs, LUTs, FFs?
Does the model fit in the latency requirements?

Neu ence



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


AI circuit for ultrafast inference on FPGA

Inference time: 280 ns
Throughput: 104 Gb/s

Dense Network
23 → 30 → 25 → 20
→ momentum & classifier

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

DSPs

logic cells

16 inputs

y resources? DSPs, LUTs, FFs?
nodel fit in the latency quirements?

5 outputs
activation: 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

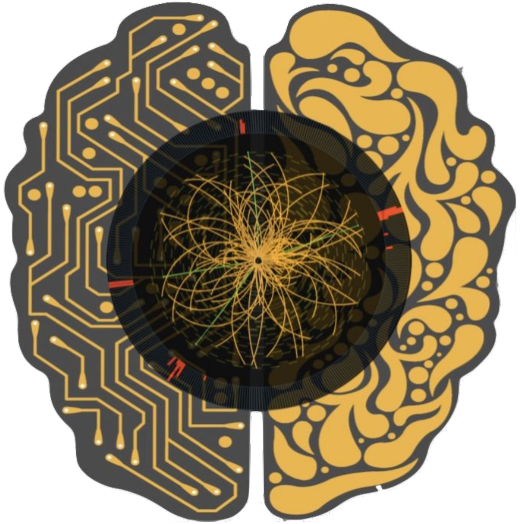
FPGA project designing

What we *won't* cover today

- Two new tutorial notebooks are not yet ready, but will be soon!
  - **Boosted decision trees**: implemented in a companion package to hls4ml
    - https://github.com/thesps/conifer
  - **Convolutional NNs**: convolutional layers can quickly increase in number of operations, recently available in hls4ml at larger scales
    - https://arxiv.org/abs/2101.05108
- 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, National Instruments DAQ framework)
  - For more off-the-shelf boards, integration with system-on-chip or host CPU can be more straightforward
    - https://github.com/mlcommons/tiny_results_v0.5/tree/main/open/hls4ml

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



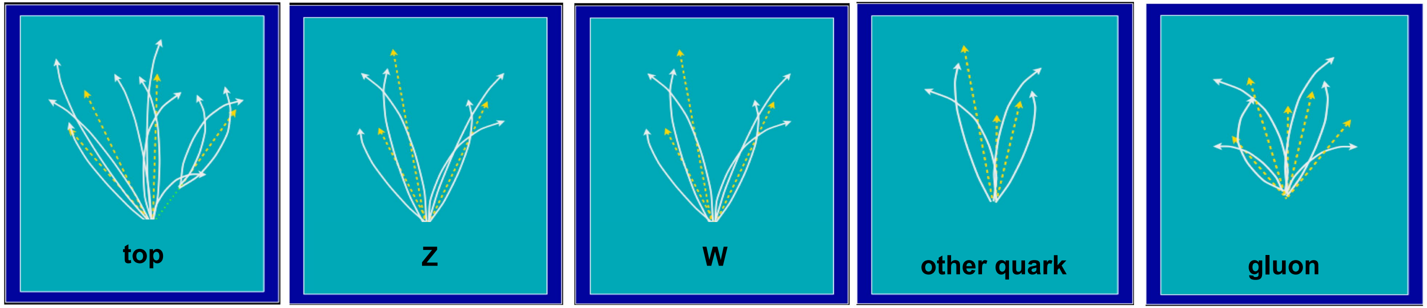
hls4ml tutorial

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→bW→bqq

3-prong jet

Z→qq

2-prong jet

W→qq

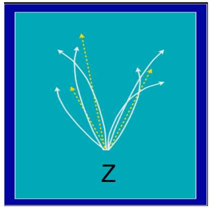
2-prong jet

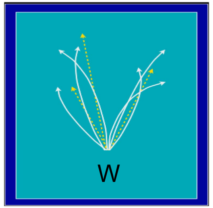
Reconstructed as one massive jet with substructure

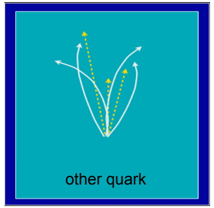
q/g background

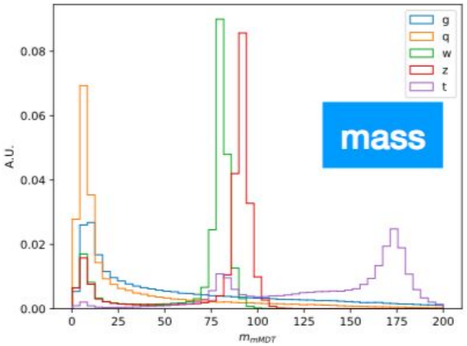
no substructure and/or mass ~ 0

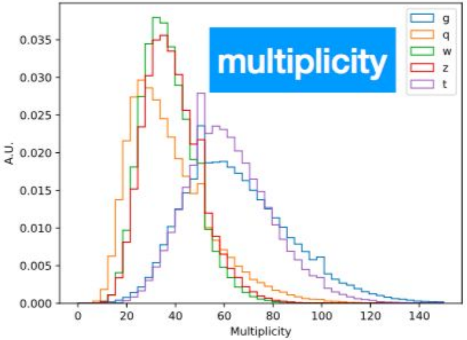
Physics case: jet tagging

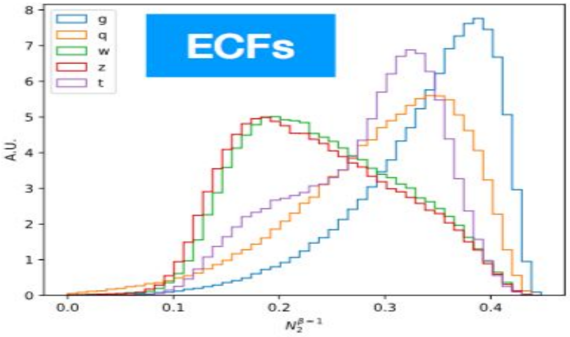












Input variables: several observables known to have high discrimination power from offline data analyses and published studies [*]

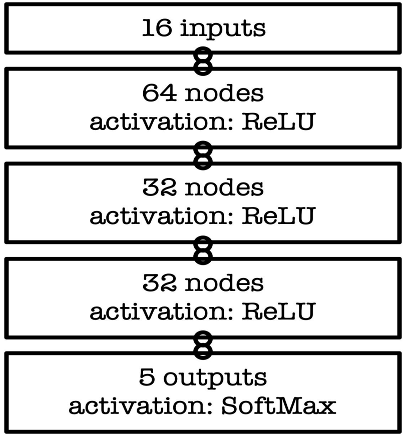
$m_{\rm mMDT}$
$N_2^{\beta=1,2}$
$M_2^{\beta=1,2}$
$C_1^{\beta=0,1,2}$
$C_2^{\beta=1,2}$
$D_2^{\beta=1,2}$
$D_2^{(\alpha,\beta)=(1,1),(1,2)}$
$\sum z \log z$
Multiplicity

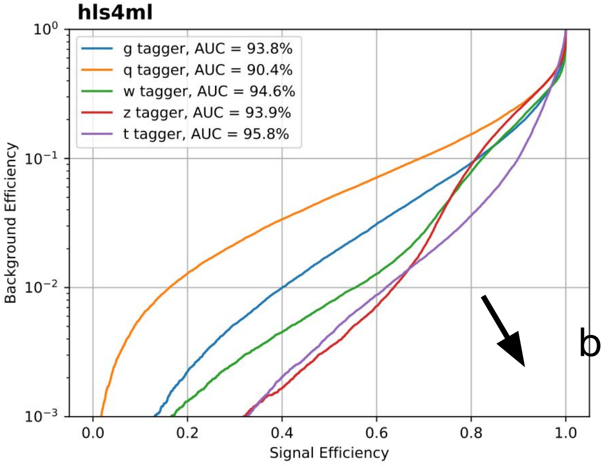
[*] D. Guest at al. PhysRevD.94.112002, G. Kasieczka et al. JHEP05(2017)006, J. M. Butterworth et al. PhysRevLett.100.242001, etc..

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



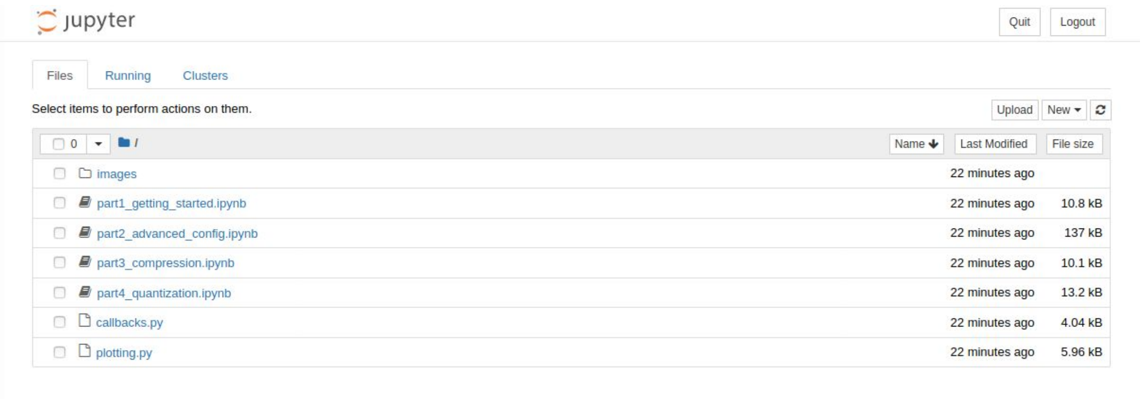


better

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

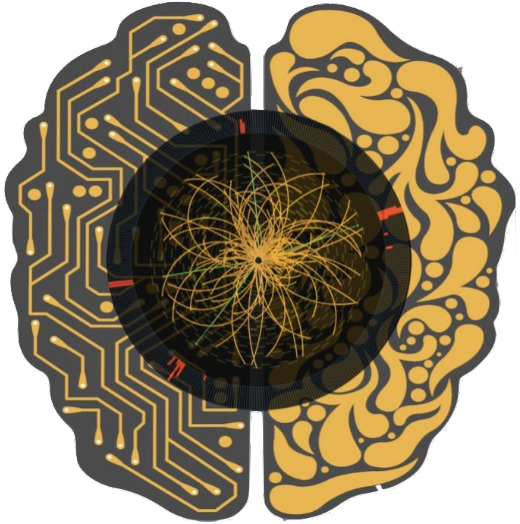
Hands On - Setup

- The interactive part is served with Python notebooks
- Open https://cern.ch/ssummers/hls4ml-tutorial in your web 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
- If you have Vivado install yourself, you might prefer to work locally, see ‘conda’ section at: https://github.com/fastmachinelearning/hls4ml-tutorial



Select items to perform actions on them.

	Name	Last Modified	File size
□	images	22 minutes ago	
□	part1_getting_started.ipynb	22 minutes ago	10.8 kB
□	part2_advanced_config.ipynb	22 minutes ago	137 kB
□	part3_compression.ipynb	22 minutes ago	10.1 kB
□	part4_quantization.ipynb	22 minutes ago	13.2 kB
□	callbacks.py	22 minutes ago	4.04 kB
□	plotting.py	22 minutes ago	5.96 kB

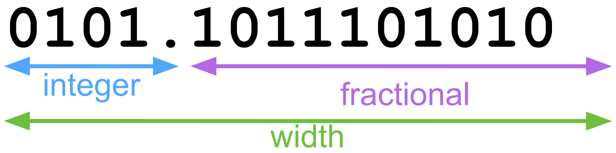


hls4ml Tutorial

Part 2: Advanced Configuration

Efficient NN design: quantization

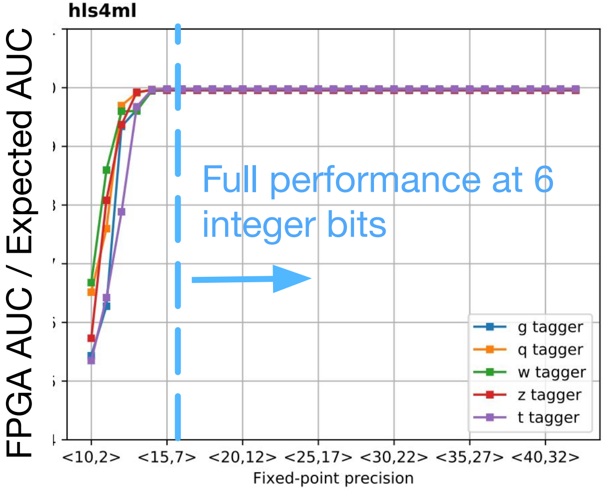
ap_fixed<width bits, integer bits>



- 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

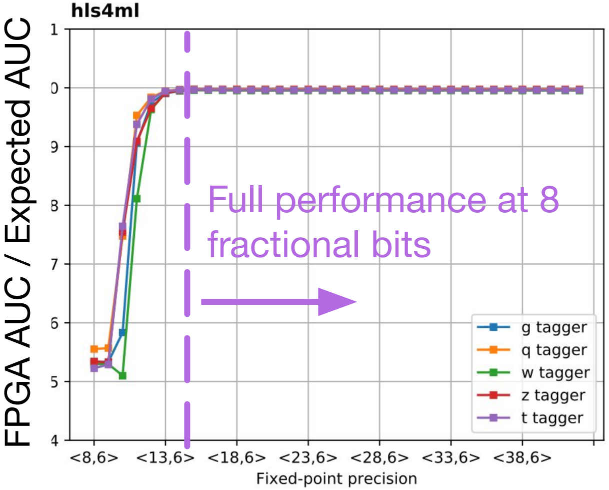
Fractional bits fixed to 8



Full performance at 6 integer bits

Scan fractional bits

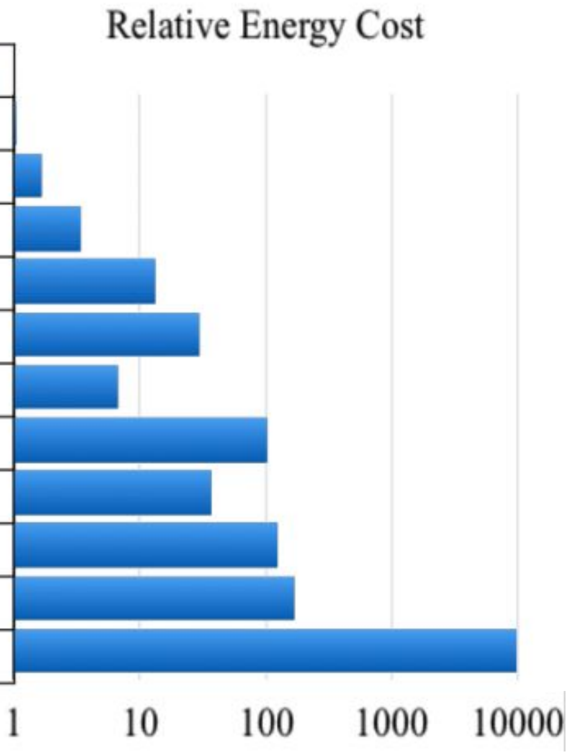
Integer bits fixed to 6



Full performance at 8 fractional bits

Efficient NN design: quantization

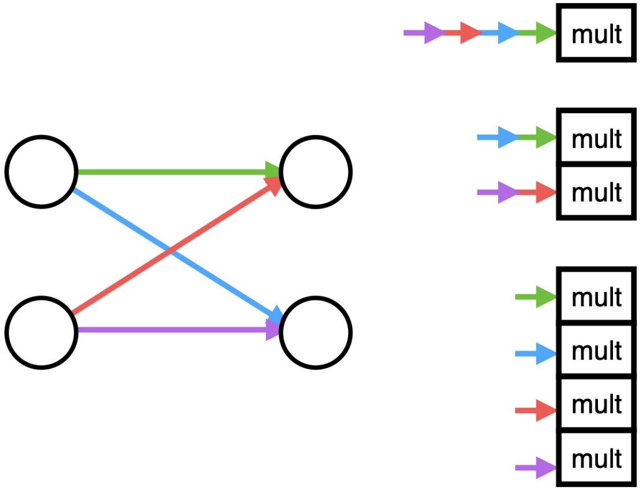
Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640



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 = 4
use 1 multiplier 4 times

Fully serial

reuse = 2
use 2 multipliers 2 times each

reuse = 1
use 4 multipliers 1 time each

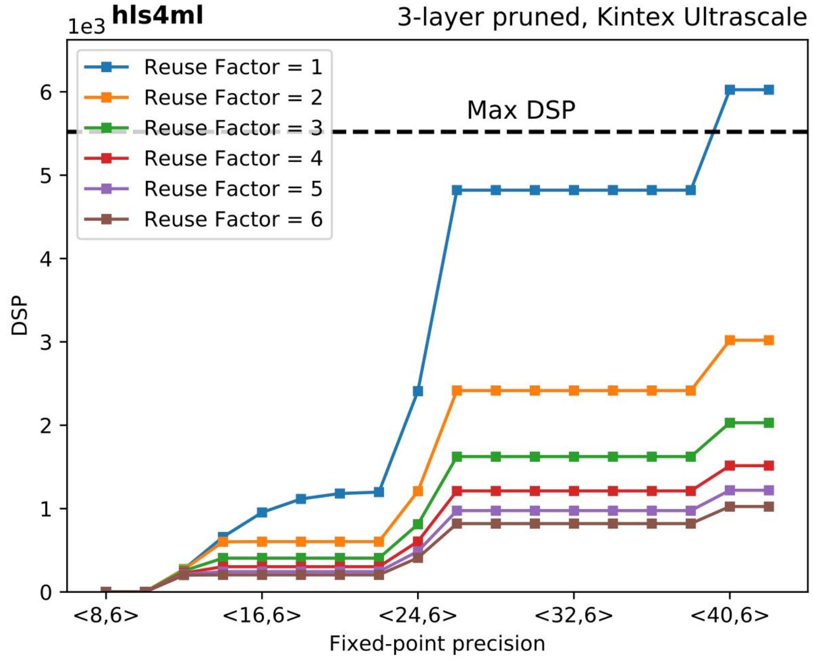
Fully parallel

Fewer resources,
Lower throughput,
Higher latency

More resources,
Higher throughput,
Lower latency

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

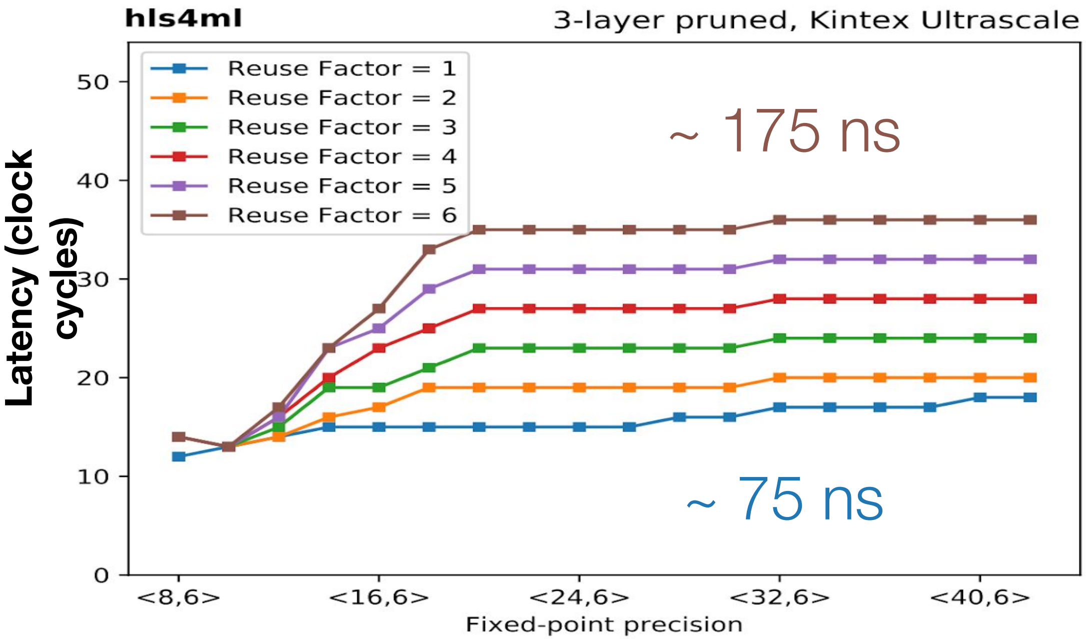
⋮

Longer latency

Parallelization: Timing

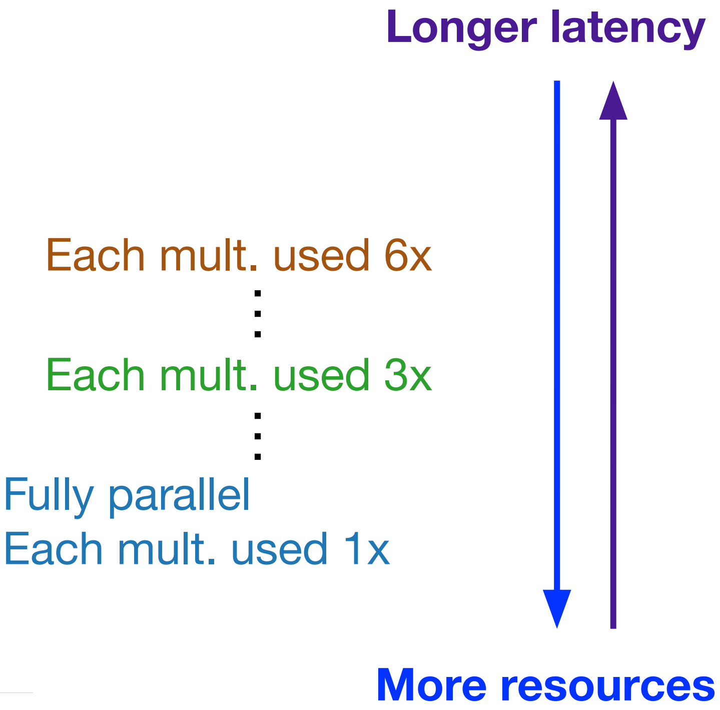
Latency of layer m

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



~ 175 ns

~ 75 ns



Longer latency

Each mult. used 6x

⋮

Each mult. used 3x

⋮

Fully parallel
Each mult. used 1x

More resources

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 → 128 → 128 → 128 → 10
Model accuracy: ~97%
We can work out how many DSPs this should use...

Large MLP

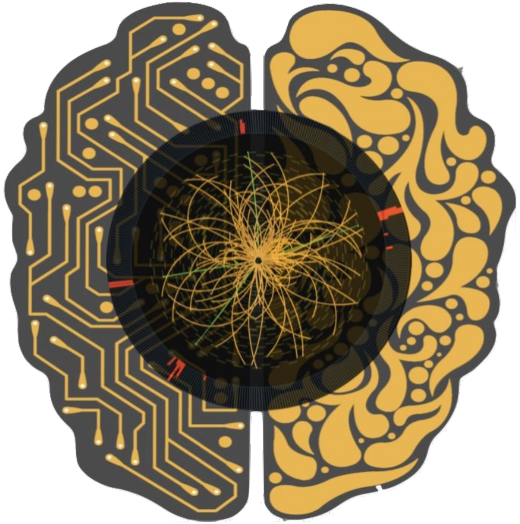
- It takes a while to synthesise, so here’s one I made earlier…
- The DSPs should be: (784 x 128) / 112 + (2 x 128 x 128 + 128 x 10) / 128 = 1162 🤞

```
============================
============================
+ 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  |
+---------------------+---------+-------+---------+--------+
...
+---------------------+---------+-------+---------+--------+
|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

hls4ml Tutorial

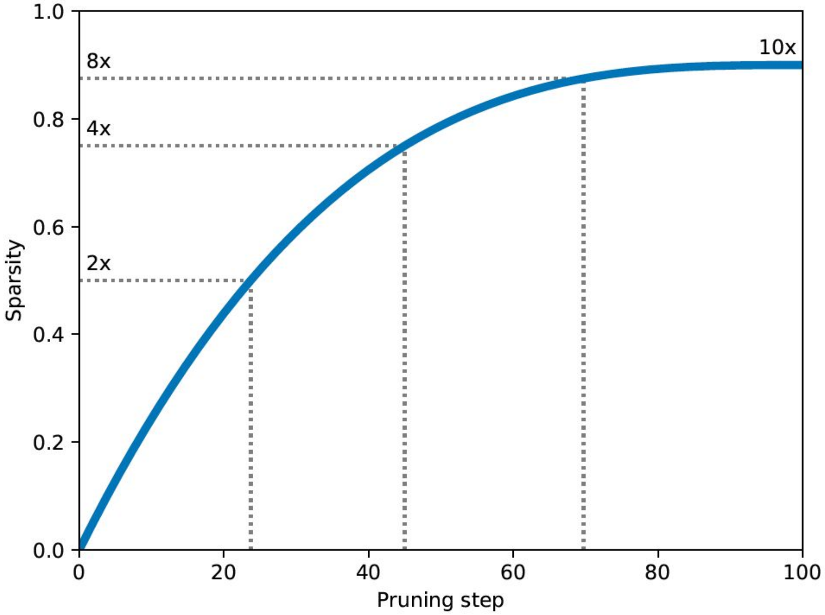
Part 3: Compression

NN compression methods

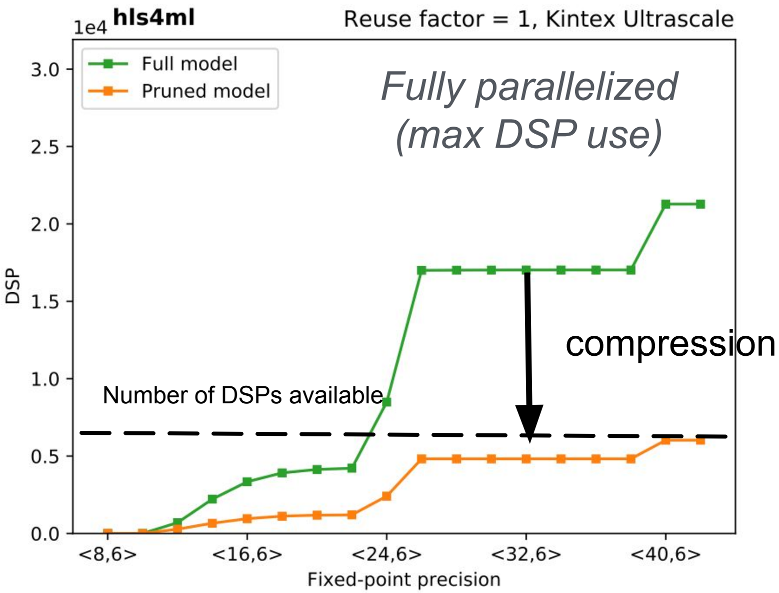
- Network compression is a widespread technique to reduce the size, energy consumption, and overtraining of deep neural networks
- Several approaches have been studied:
  - **parameter pruning:** selective removal of weights based on a particular ranking [arxiv.1510.00149, arxiv.1712.01312]
  - **low-rank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

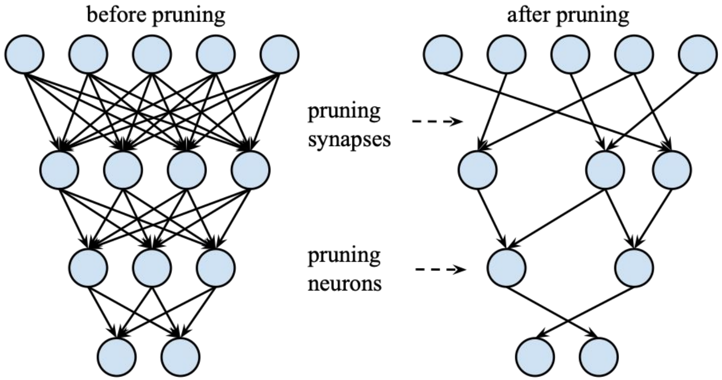
- Iteratively remove low magnitude weights, starting with 0 sparsity, smoothly increasing up to the set target as training proceeds



Efficient NN design: compression



70% compression ~ 70% fewer DSPs



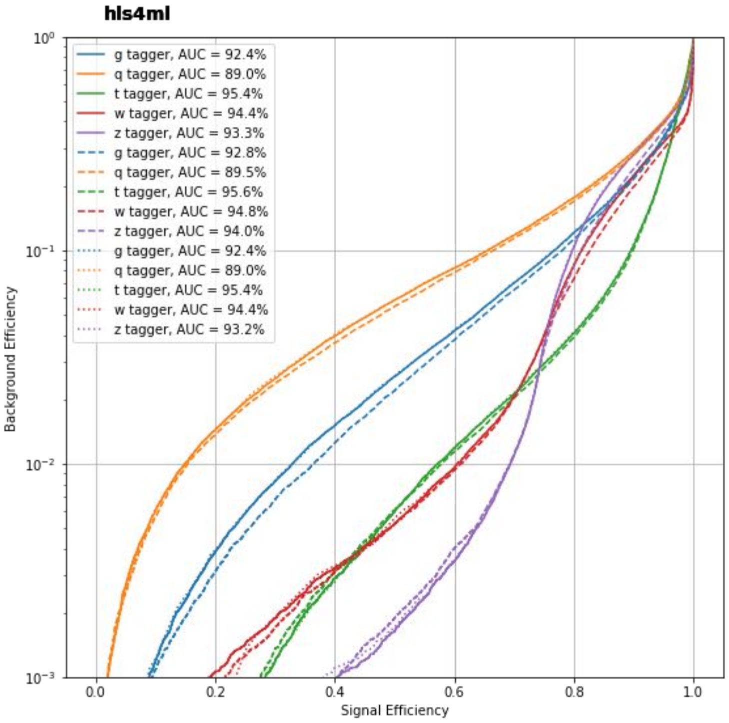
- DSPs (used for multiplication) are often limiting resource
  - maximum use when fully parallelized
  - DSPs have a max size for input (e.g. 27x18 bits), so number of DSPs per multiplication changes with precision

hls4ml Tutorial

Part 4: Quantization

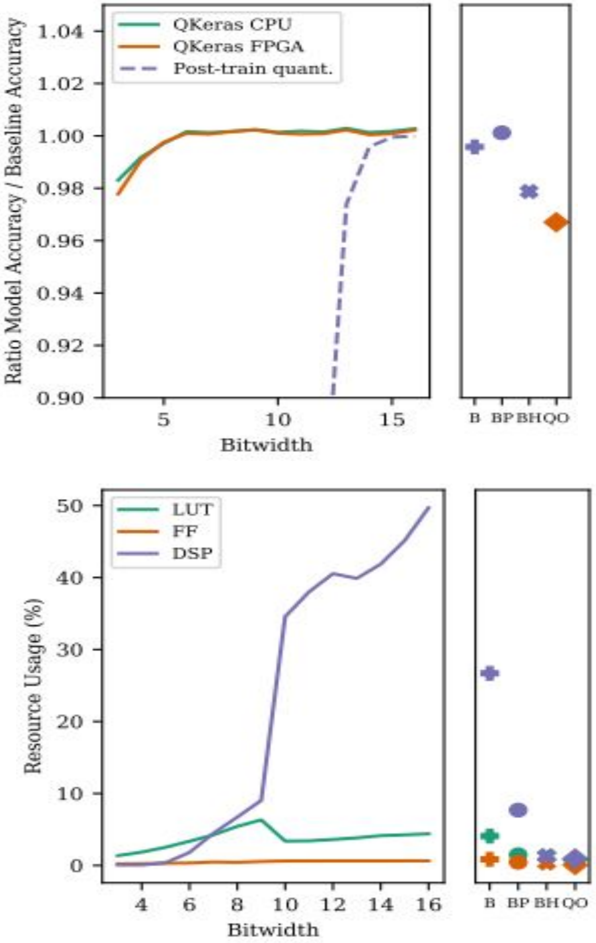
Efficient NN design: quantization

- 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
- See other talks on quantization at this workshop: Amir, Thea, Benjamin



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
- 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
- Learned 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**
- The tutorial server is always available at https://cern.ch/ssummers/hls4ml-tutorial
- You can find these tutorial notebooks to run locally at: https://github.com/fastmachinelearning/hls4ml-tutorial
- You can run the tutorial Docker image yourself like:
  - docker run -p 8888:8888 gitlab-registry.cern.ch/ssummers/hls4ml-tutorial:12.v
  - 15 GB download! Or remove '.v' for a much smaller image but without Xilinx tools (so no 'build')
- Use hls4ml in your own environment: `pip install hls4ml[profiling]`