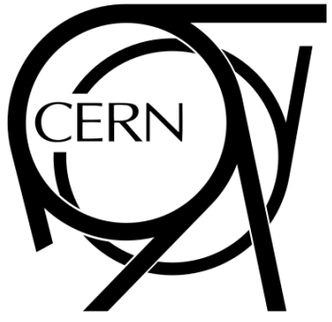
Fast Machine Learning Inference on FPGAs for Trigger and DAQ

4th ATLAS Machine Learning Workshop - CERN - 11th November 2019

Javier Duarte, Sergo Jindariani, Ben Kreis, Ryan Rivera, Nhan Tran (Fermilab)
Jennifer Ngadiuba, Maurizio Pierini, Vladimir Loncar, **Sioni Summers** (CERN)
Edward Kreinar (Hawkeye 360)
Phil Harris, Song Han, Dylan Rankin (MIT)
Zhenbin Wu (University of Illinois at Chicago)
Giuseppe di Guglielmo (Columbia University)

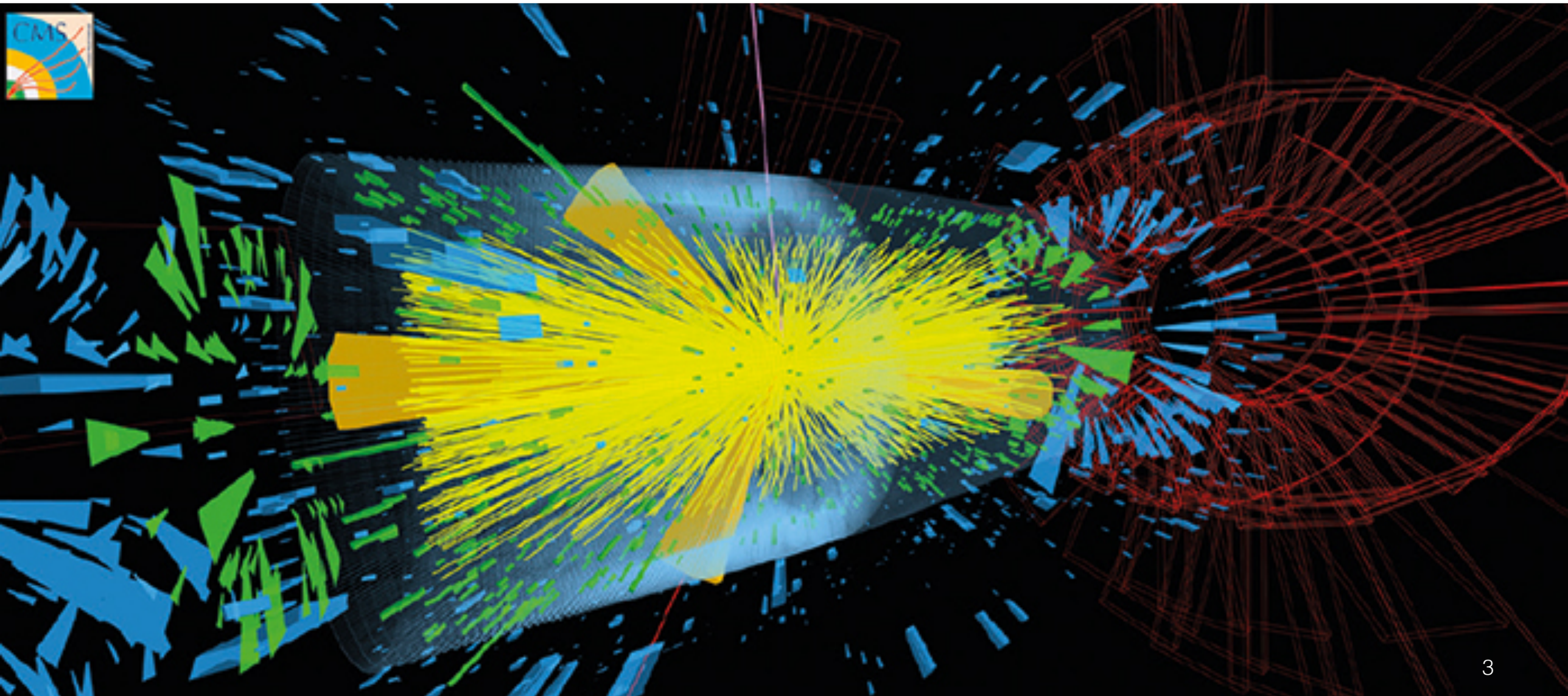


Contents

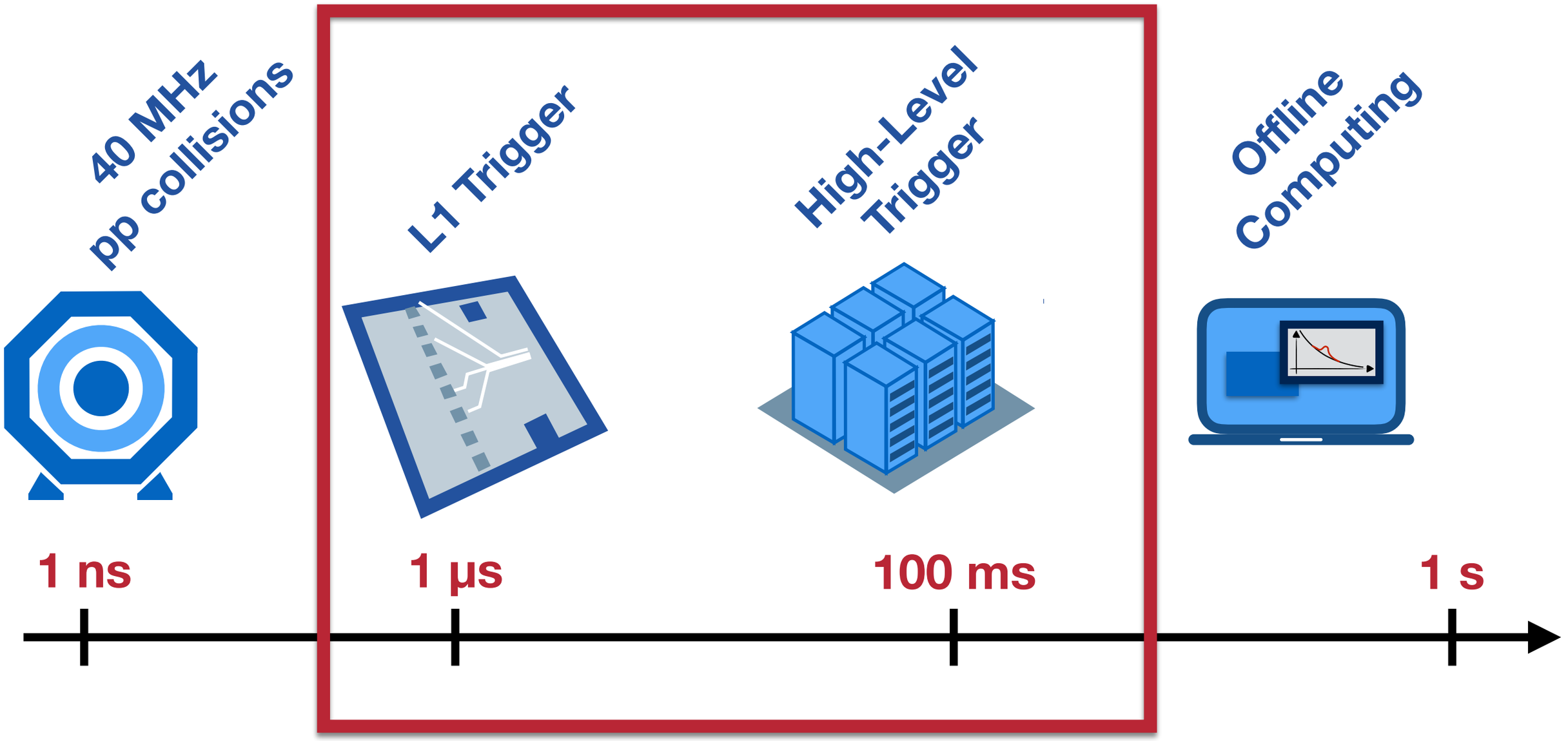
The challenge: triggering at (HL-)LHC

Extreme bunch crossing frequency of 40 MHz → extreme data rates O(100 TB/s)

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



The LHC big data problem



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

What are FPGAs?

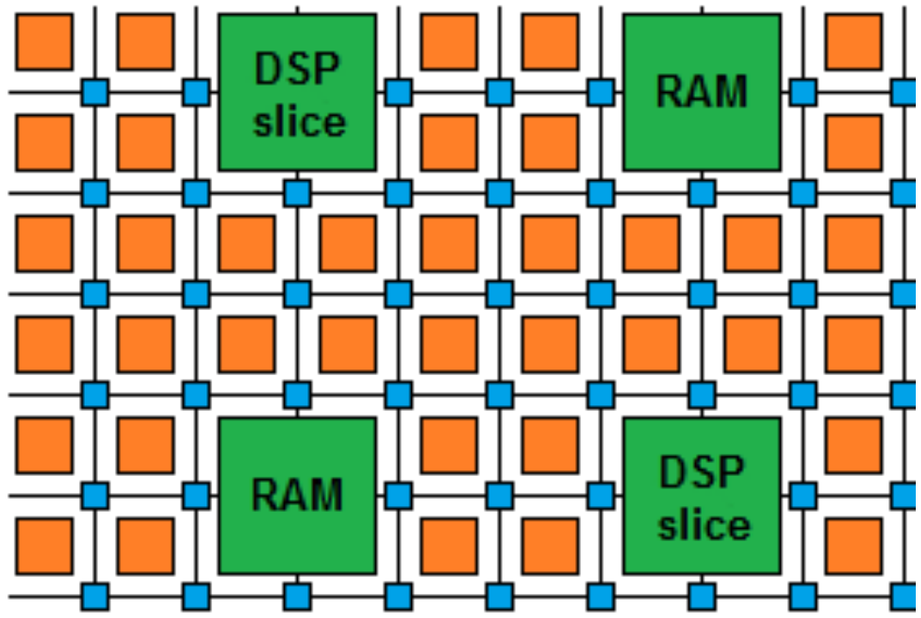
Field Programmable Gate Arrays are reprogrammable integrated circuits

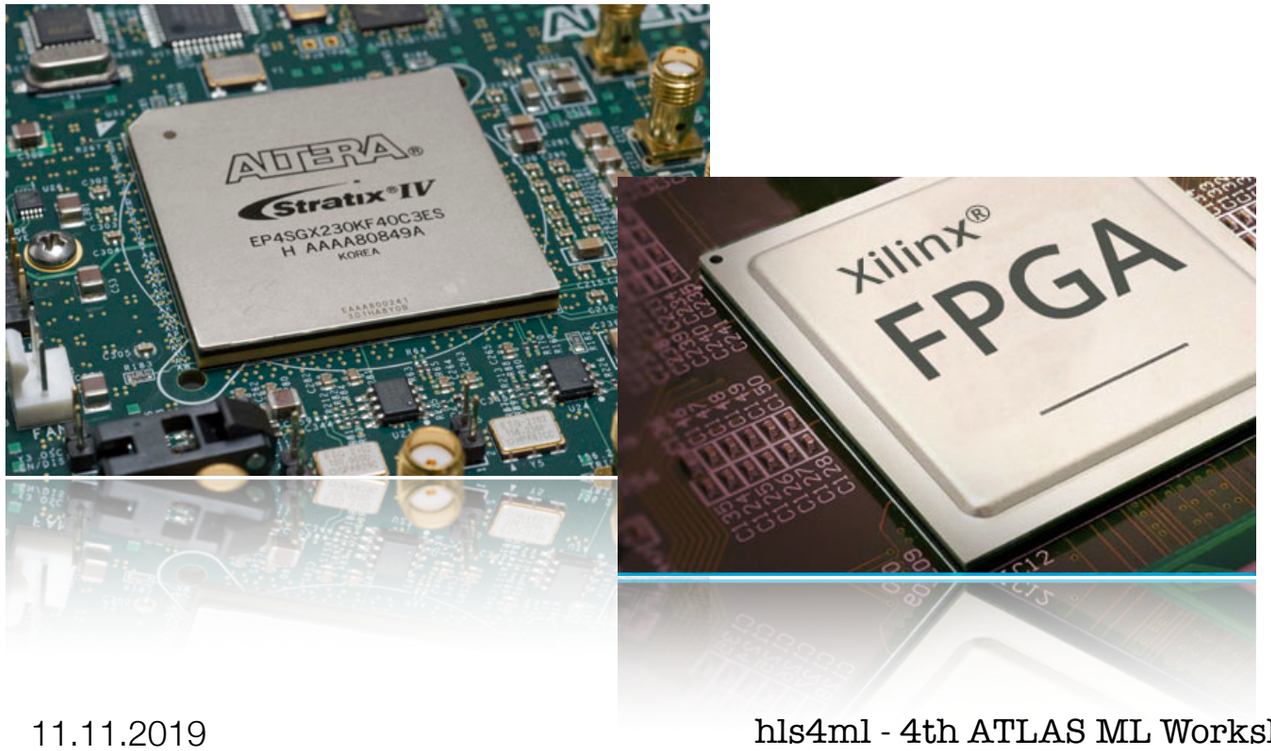
Logic cells / Look Up Tables perform arbitrary functions on small bitwidth inputs (2-6)

These can be used for boolean operations, arithmetic, memory

Flip-Flops register data in time with the clock pulse

FPGA diagram



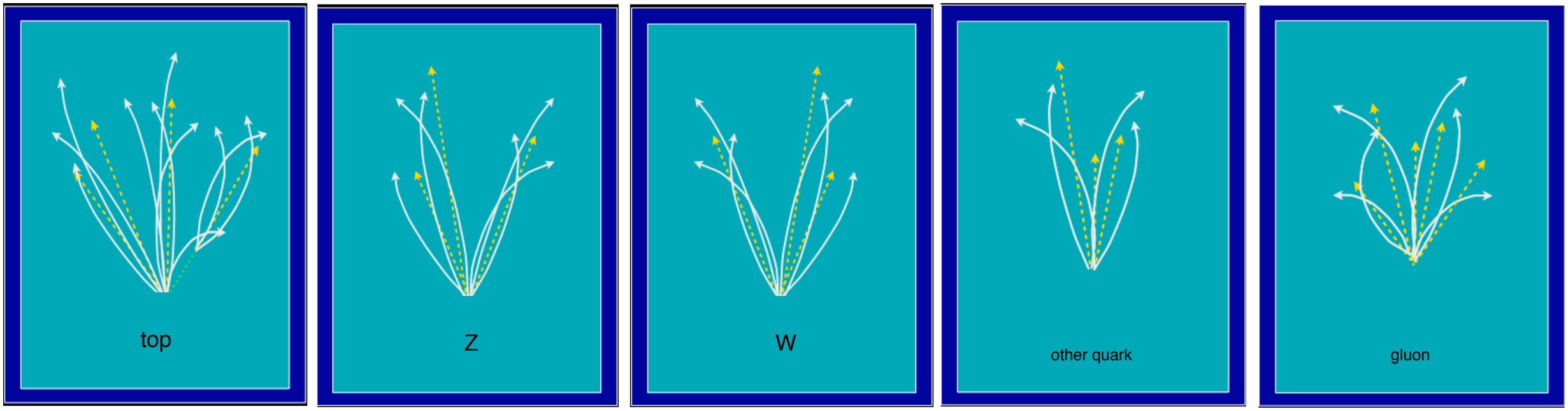


DSPs are specialized units for multiplication and arithmetic

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

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	**Z→qq**	**W→qq**	**q/g background**
3-prong jet	2-prong jet	2-prong jet	no substructure and/or mass ~ 0

Reconstructed as one massive jet with substructure

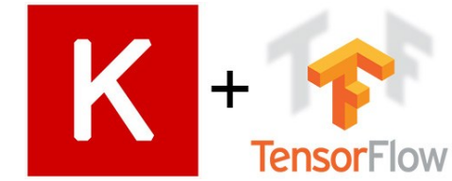
Physics case: jet tagging

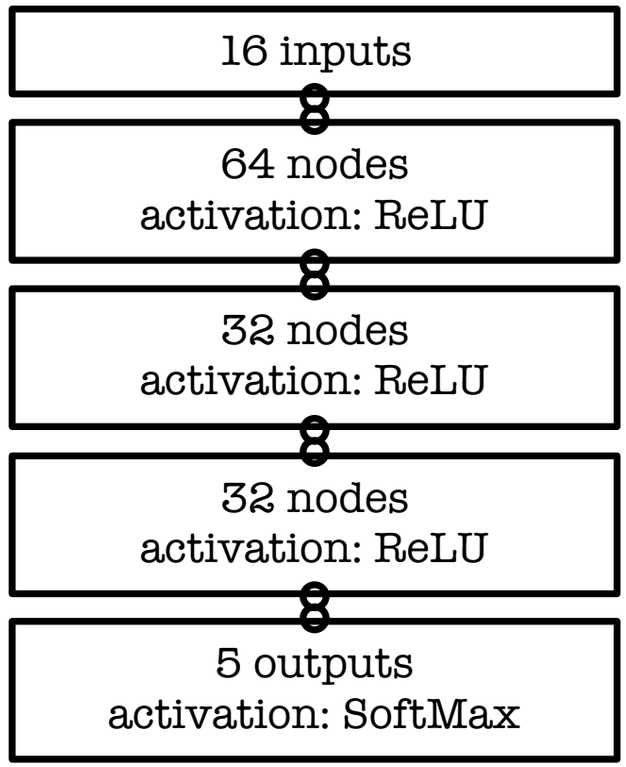
- We train (on GPU) the **five output multi-classifier** on a sample of ~ 1M events with two boosted WW/ZZ/tt/qq/gg anti-k_T jets

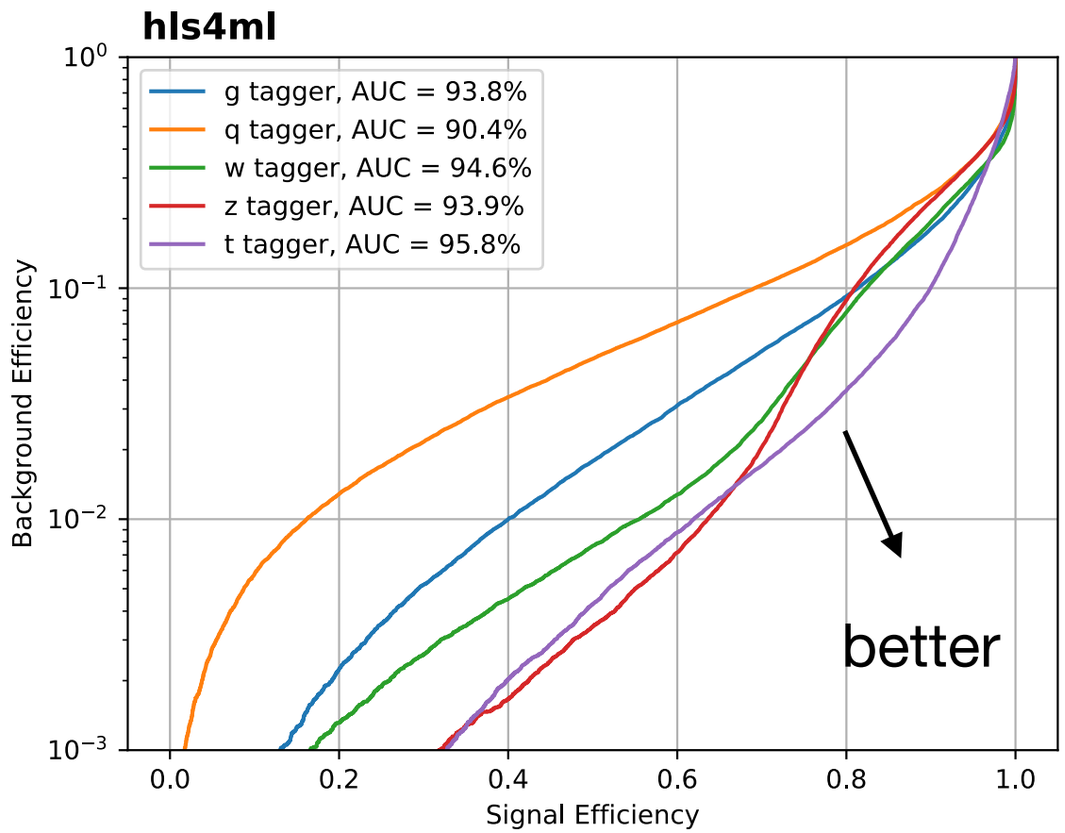
- Fully connected neural network with **16 expert-level inputs**:

  - Relu activation function for intermediate layers

  - Softmax activation function for output layer







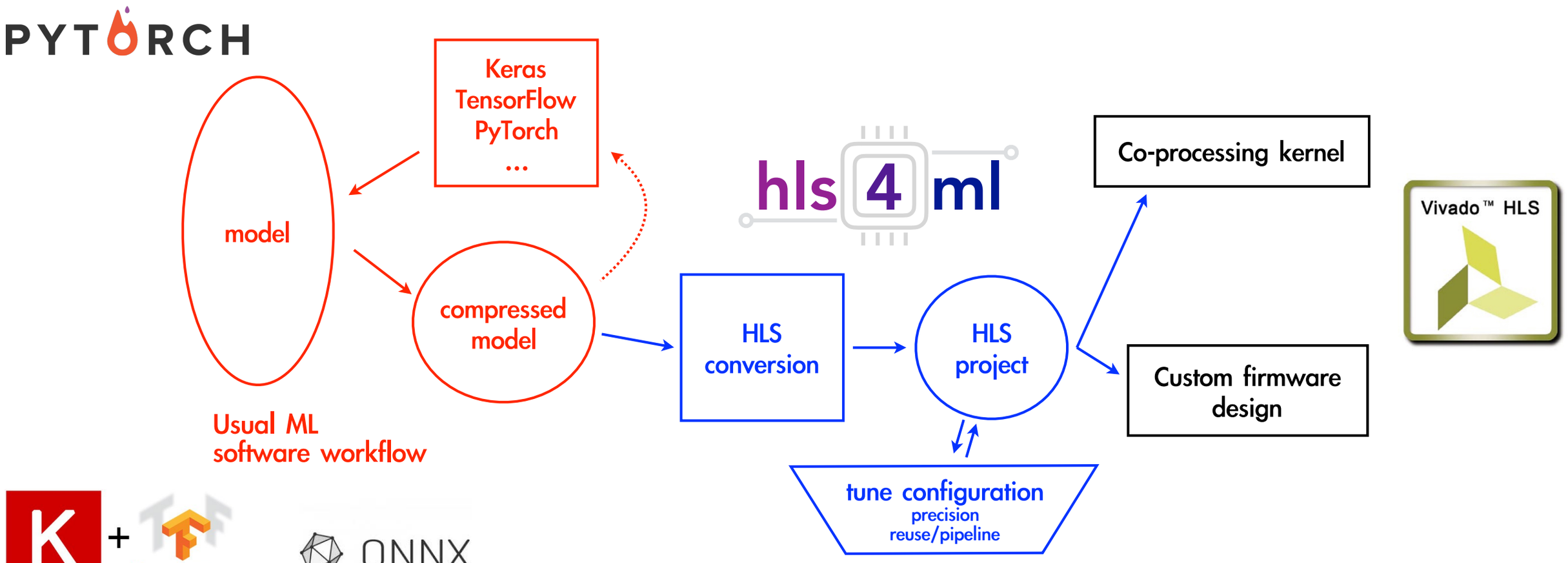
AUC = area under ROC curve (100% is perfect, 20% is random)

Neural Network to FPGA translation with hls4ml

high level synthesis for machine learning

User friendly and optimised translation for ML to FPGA

- Input model trained with standard DL libraries
- Xilinx HLS software: accessible to non-FPGA-expert, a resource not common in HEP
- Comes with implementation of common ingredients: layers, activation functions
- ‘Exotic things’: binary/ternary networks
- And optimisations: layer merging
- https://fastmachinelearning.org/



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

https://arxiv.org/abs/1804.06913

Efficient NN design for FPGAs

FPGAs provide huge flexibility

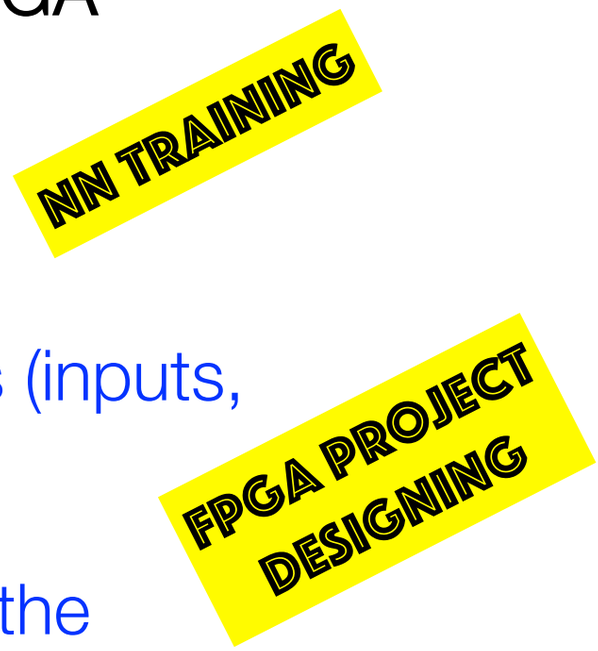
Performance depends on how well you take advantage of this

Constraints:

- Input bandwidth
- FPGA resources
- Latency

With `hls4ml` package we have studied/optimized the FPGA design 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

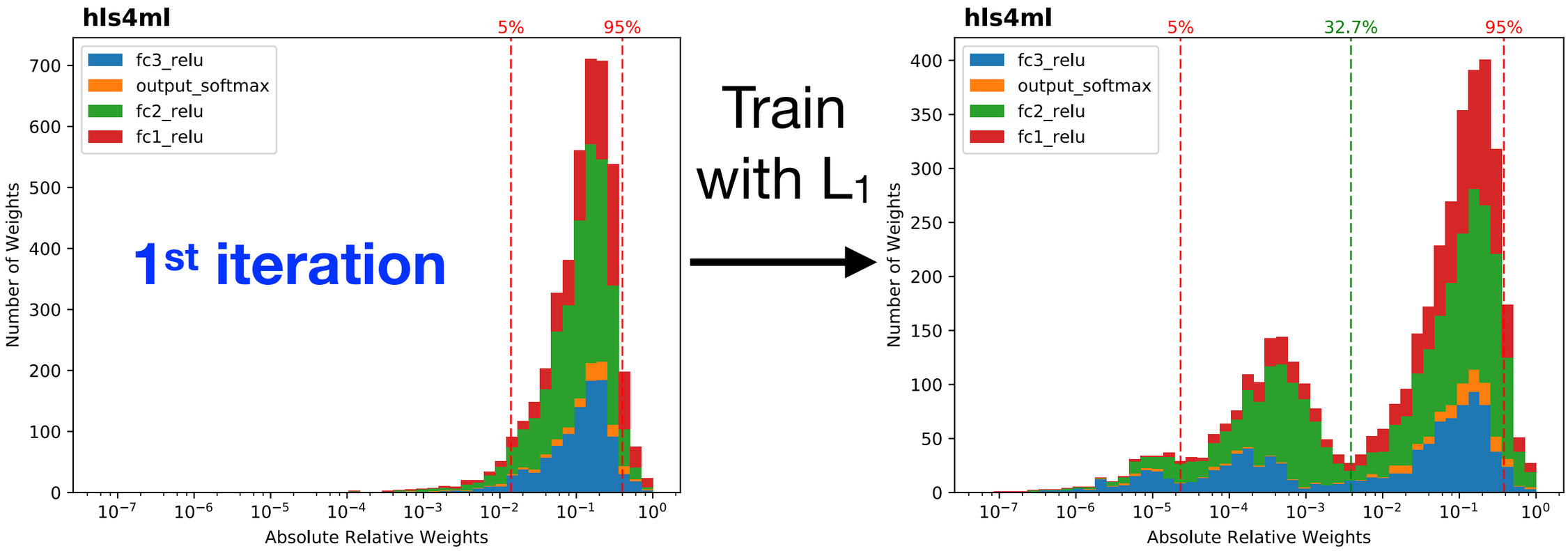


Compression with parameter pruning

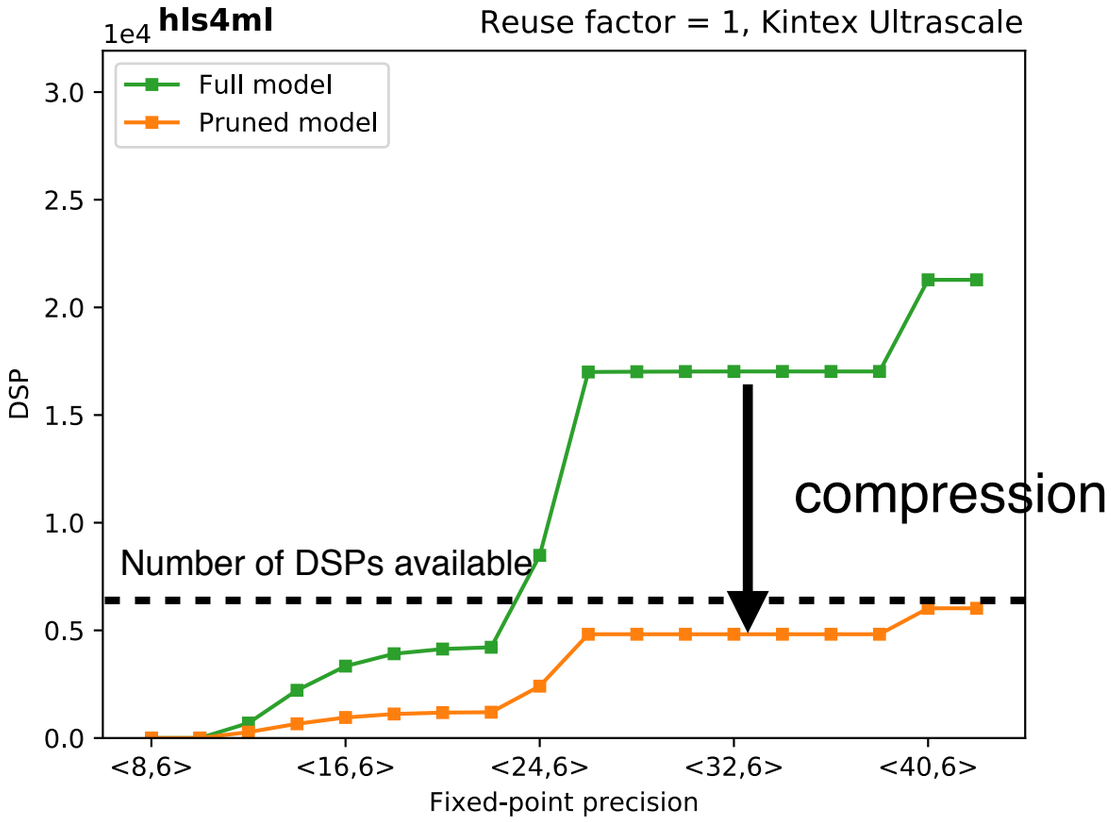
- Iterative approach:
  - train with **L1 regularization** (loss function augmented with penalty term):

$$L_\lambda(\vec{w}) = L(\vec{w}) + \lambda||\vec{w}_1||$$

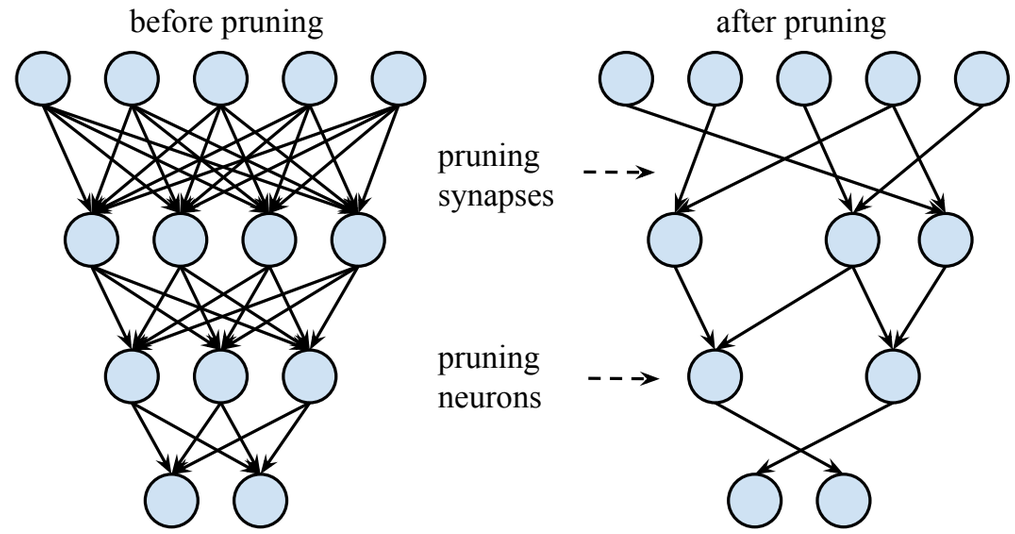
  - sort the weights based on the value relative to the max value of the weights in that layer



Efficient NN design: compression



70% compression ~ 70% fewer DSPs



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

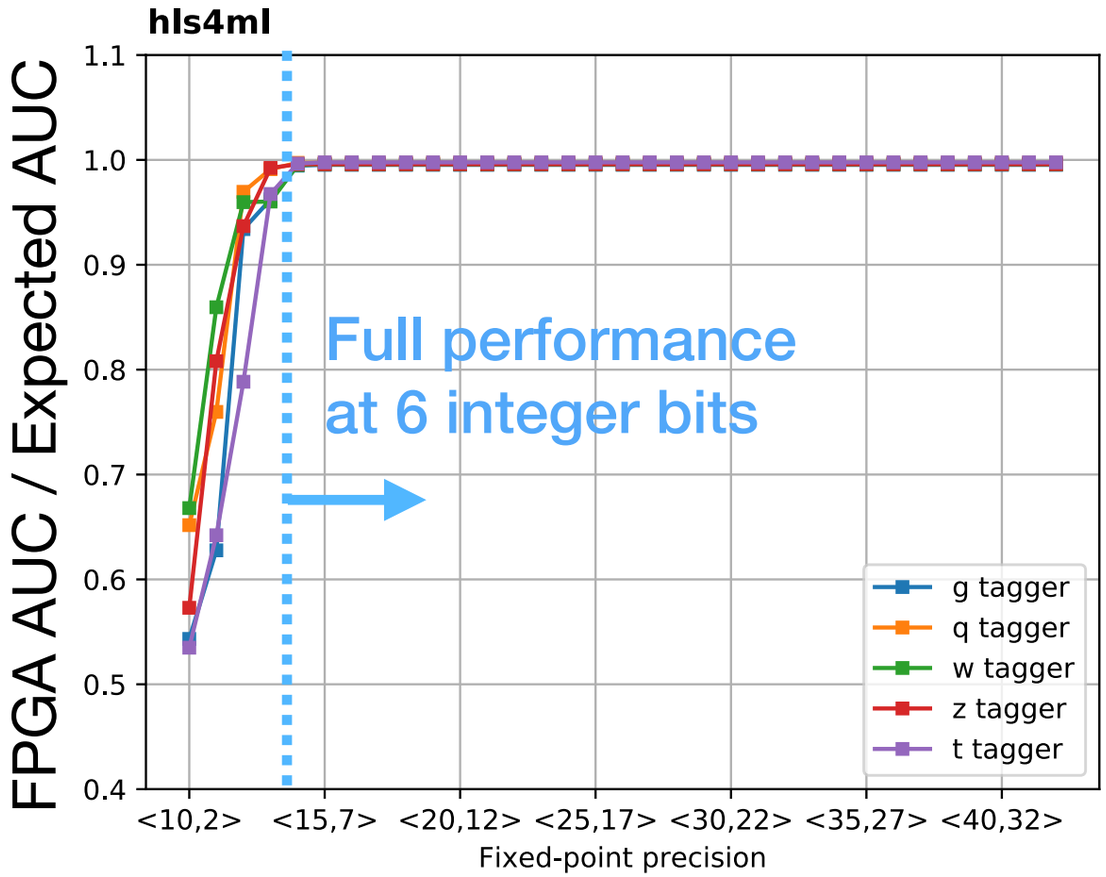
Efficient NN design: quantization

ap_fixed<width,integer>

- Quantify the performance of the classifier with the AUC
- Expected AUC = AUC achieved by 32-bit floating point inference of the neural network

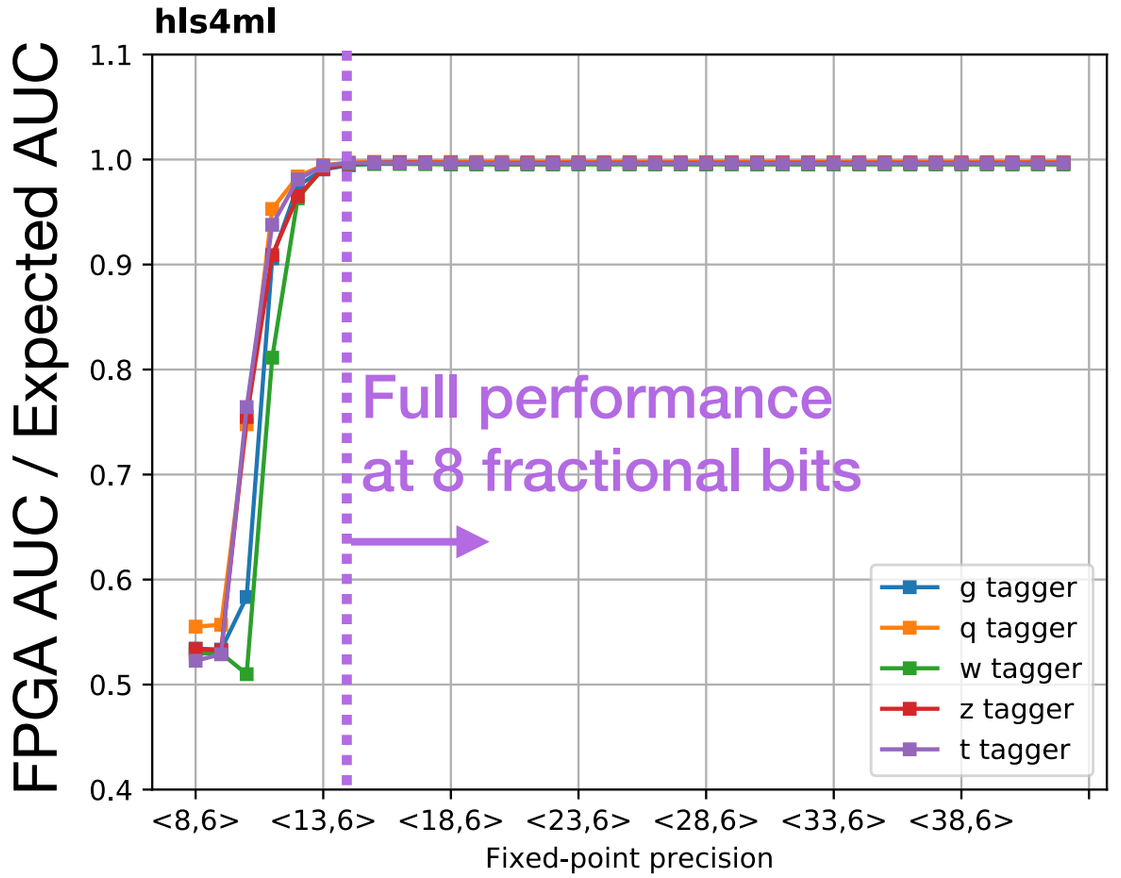
Scan integer bits

Fractional bits fixed to 8

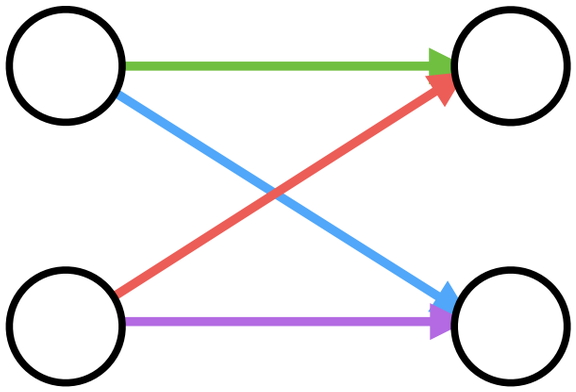


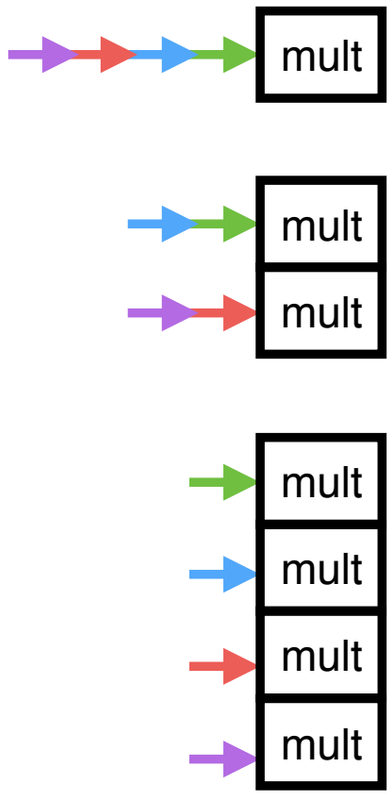
Scan fractional bits

Integer bits fixed to 6



Efficient NN design: reuse

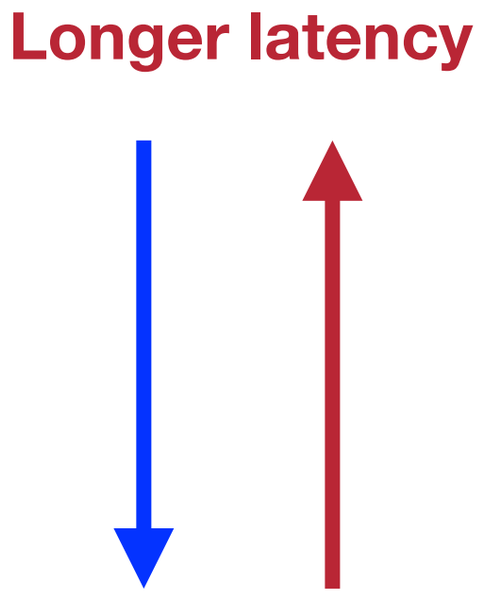




reuse = 4
use 1 multiplier 4 times

reuse = 2
use 2 multipliers 2 times each

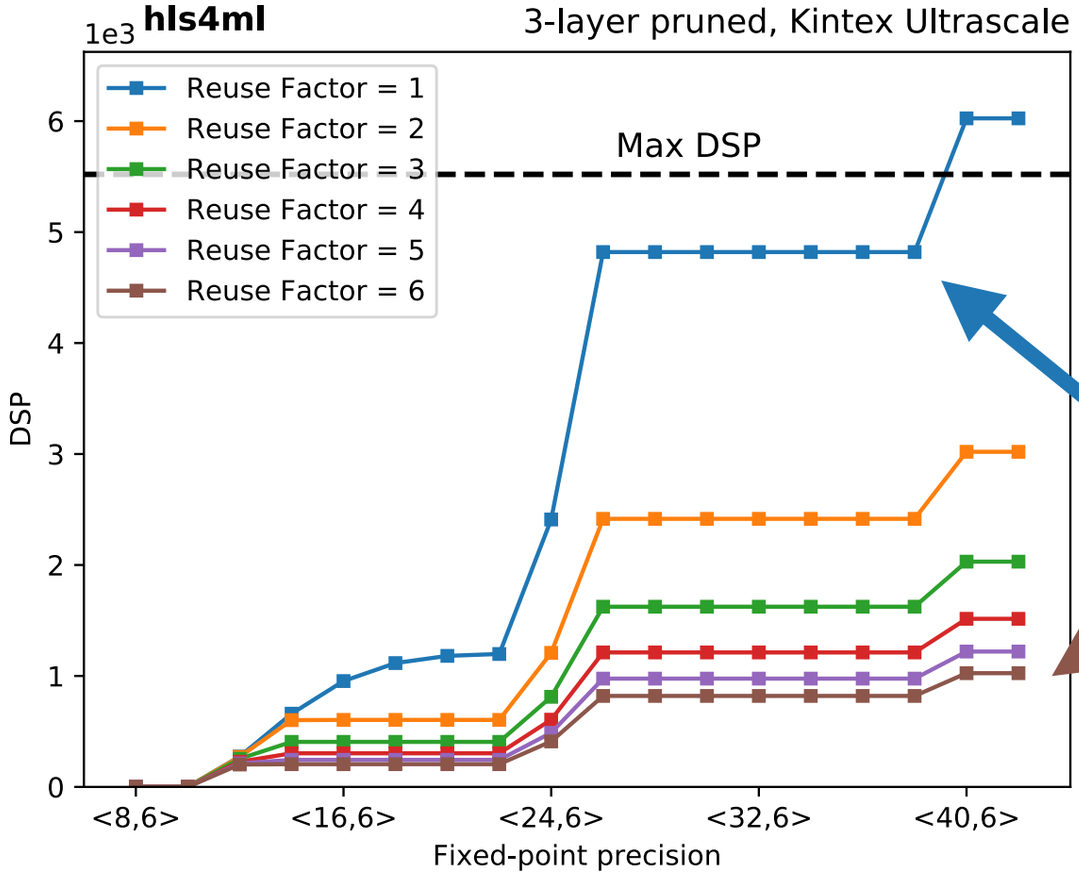
reuse = 1
use 4 multipliers 1 time each

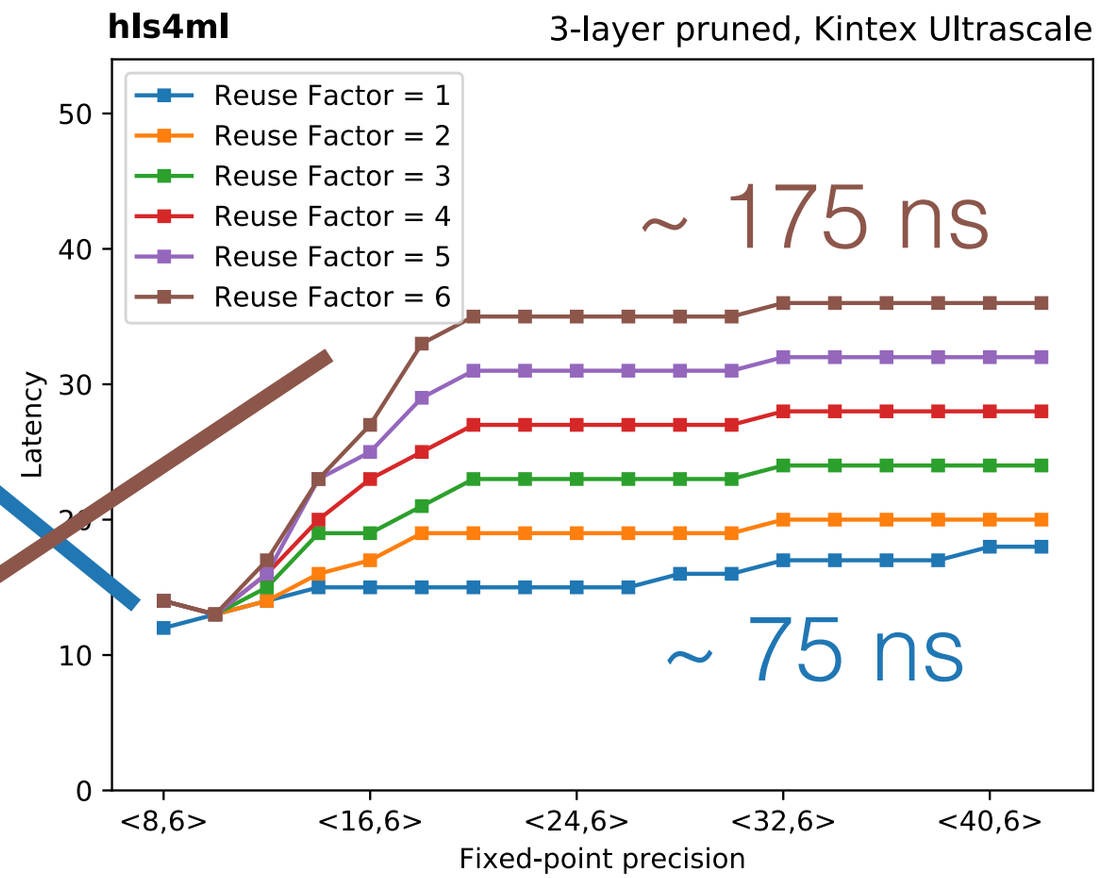


- Key feature of **hls4ml**: a handle to trade resource usage and latency/throughput
- Reuse = 1: fully unroll everything onto different resources
  - Fastest, most resource intensive
- Reuse > 1: one resource used sequentially for several operations
  - Slower, but save resources

Parallelisation

- Low reuse gives lowest latency, most resource usage
- High reuse gives longer latency, lower resource usage
- *Throughput* decreases with increasing reuse
- A large enough model will use all of the resources with reuse=1, so sometimes must increase it

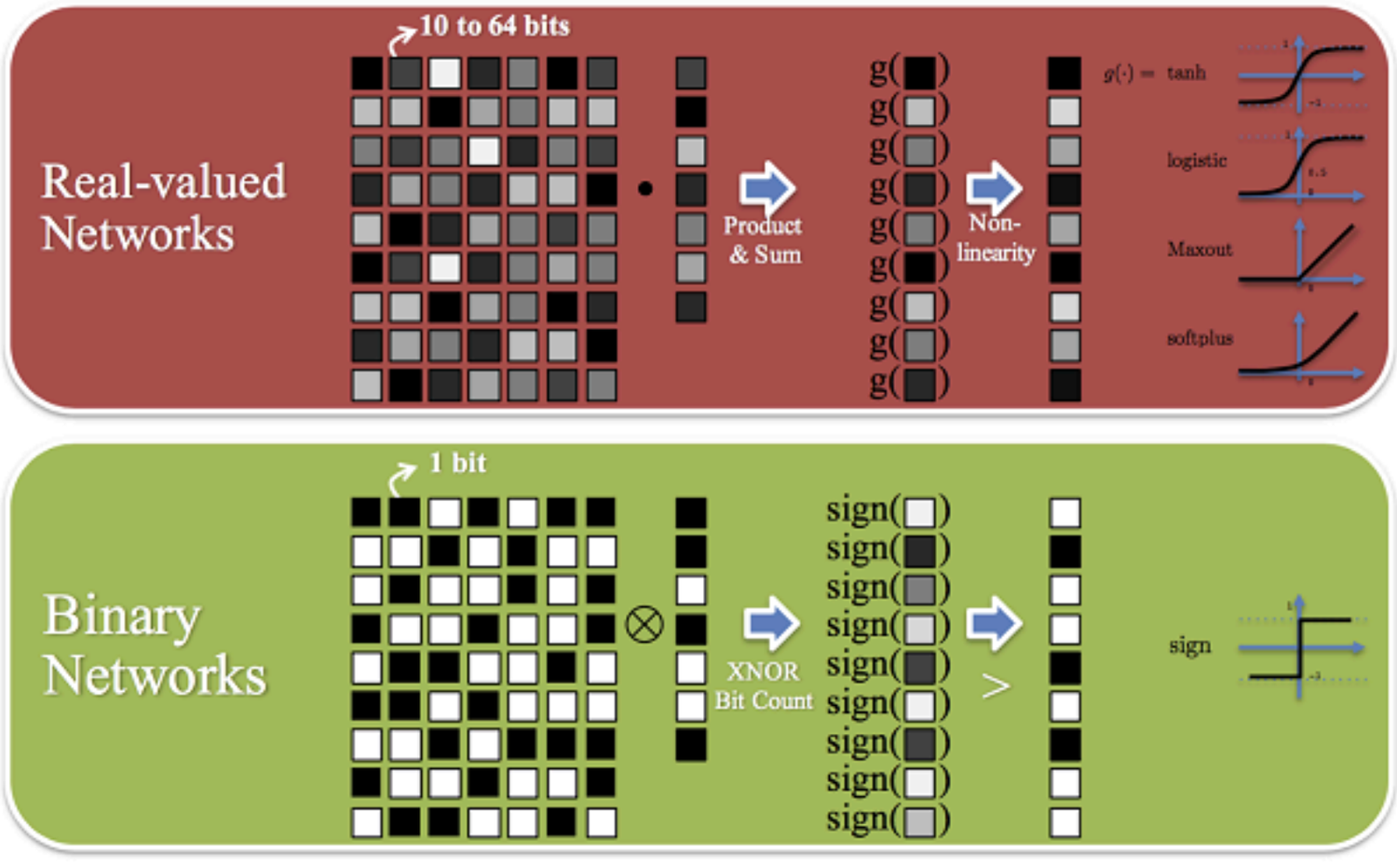




Low precision Neural Networks in hls4ml

Binary / Ternary neural networks

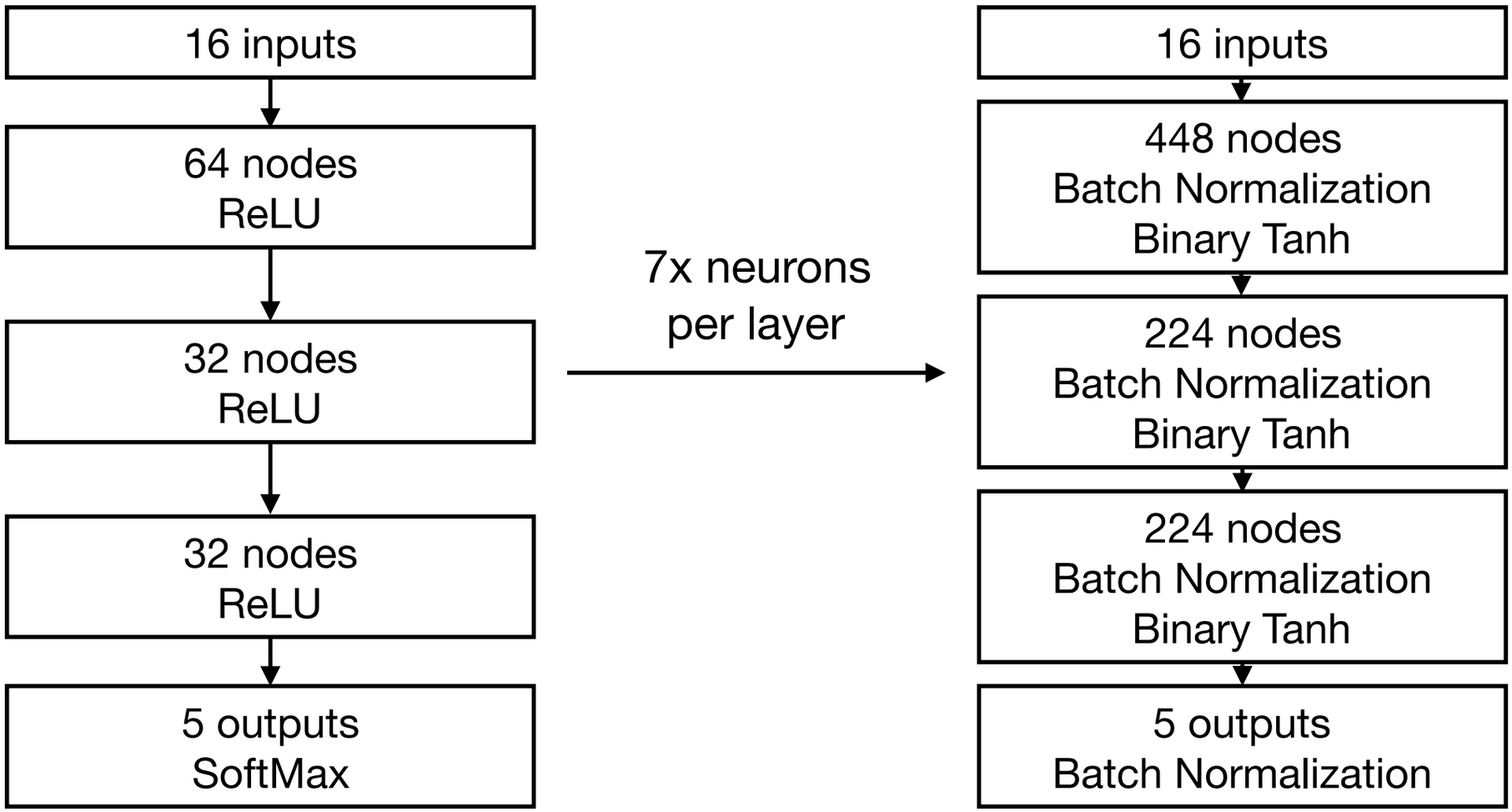
- DSPs (multipliers) usually the limiting resource for our NN inference
- Instead, use 1- or 2-bit weights with limited performance loss
- Can have very efficient computation in the FPGA
- Binarize weights but not gradients during backpropagation
- Use Binary Tanh, Ternary Tanh or ReLu activation
- Batch Normalization
- BNN: arxiv.1602.02830
- TNN: arxiv.1605.04711



https://software.intel.com/en-us/articles/accelerating-neural-networks-with-binary-arithmetic

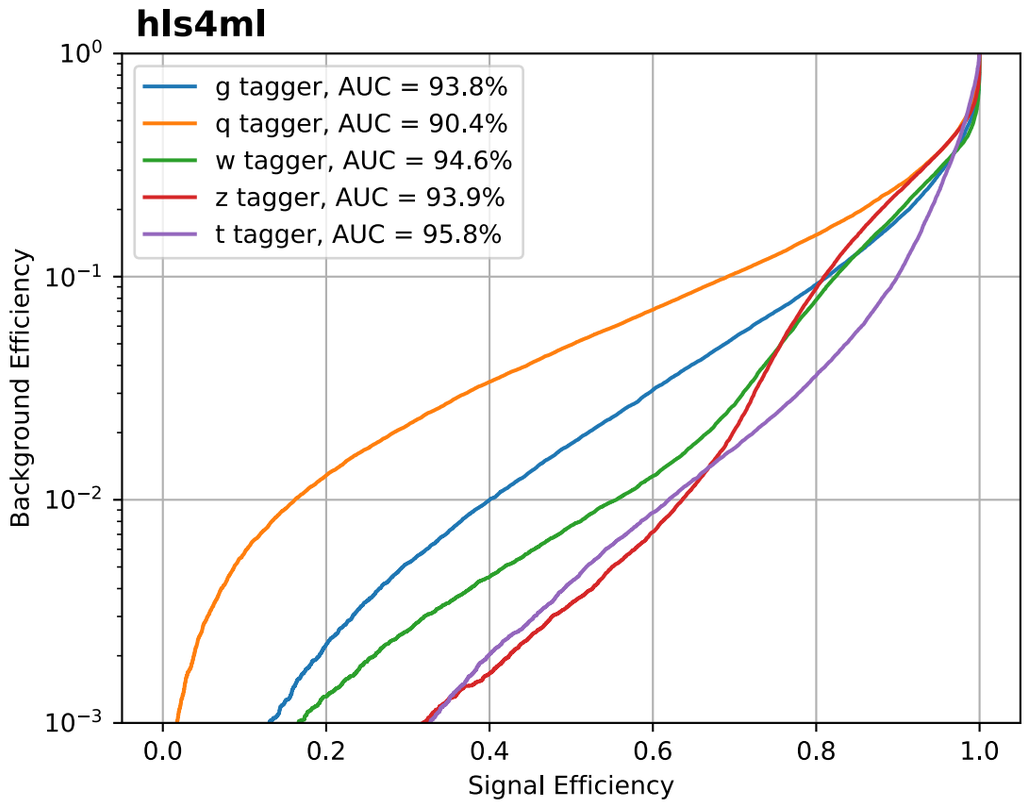
BNN - Jet Classification

- Design an architecture to perform the same jet classification task but now with binary weights and activations
- Performed hyperparameter optimization to find most performant model within some constraints

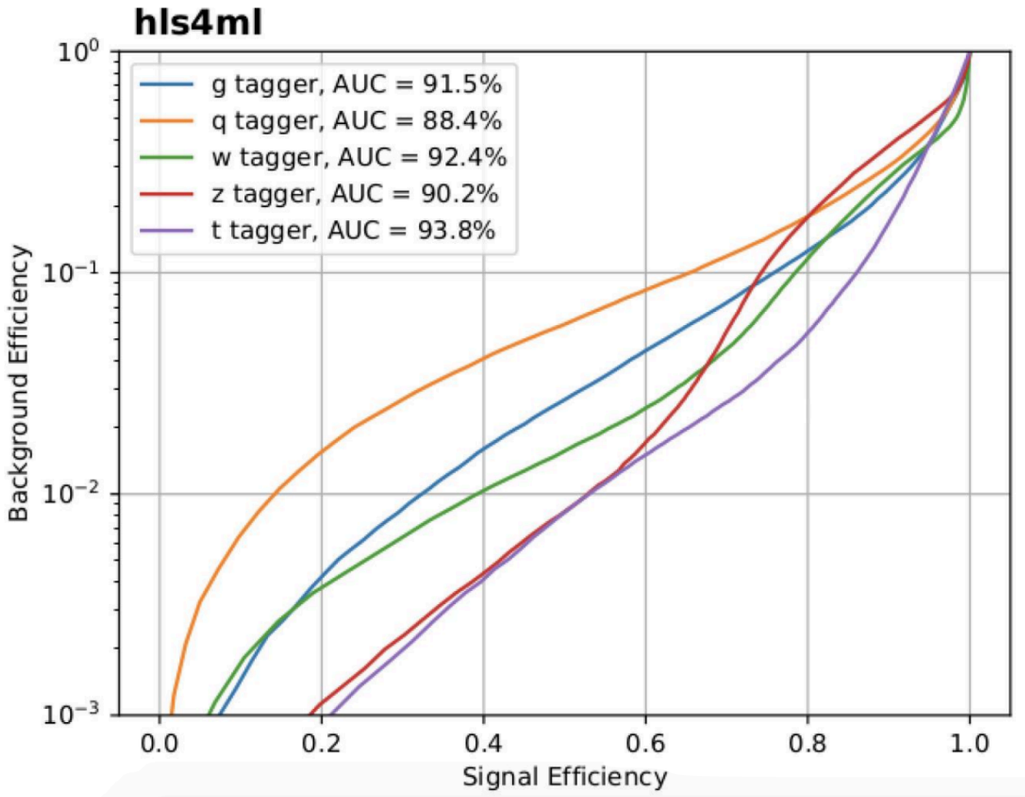


BNN - Jet Classification

- Design an architecture to perform the same jet classification task but now with binary weights and activations
- Performed hyperparameter optimization to find most performant model within some constraints
- Performance is a little bit worse, but not a lot



Original: 16-bit weights
Average accuracy: 0.75



Binarized: 1-bit weights
Average accuracy: 0.72

BNN - Jet Classification

- Results targeting Xilinx VU9P FPGA at 200 MHz

Model	Accuracy	Latency (μs)	DSP (%)	LUT (%)	FF (%)
Original model	0.75	0.06	60	7	1
Original model (70% compressed)	0.75	0.09	15	1.7	0.7
Small BNN (16x64x32x32x5)	0.62	0.04	-	0.8	0.1
Optimized BNN (16x448x224x224x5)	0.72	0.21	-	15	7
BNN w/ReLu (16x128x64x64x5)	0.70	0.140	4	6	1
Optimized TNN (16x128x64x64x64x5)	0.72	0.11	-	6	1
TNN w/ReLu (16x64x32x32x5)	0.68	0.06	2	2	0.2

BNN - Jet Classification

- Model compression saves a lot of resources with no impact on classification performance

Model	Accuracy	Latency (μs)	DSP (%)	LUT (%)	FF (%)
Original model (16x64x32x32x5)	0.75	0.06	60	7	1
Original model (75% compressed)	0.75	0.06	15	1.7	0.7
Small BNN (16x64x32x32x5)	0.62	0.04	-	0.8	0.1
Optimized BNN (16x448x224x224x5)	0.72	0.21	-	15	7
BNN w/ReLu (16x128x64x64x5)	0.70	0.140	4	6	1
Optimized TNN (16x128x64x64x64x5)	0.72	0.11	-	6	1
TNN w/ReLu (16x64x32x32x5)	0.68	0.06	2	2	0.2

BNN - Jet Classification

- Same-architecture Binary NN is *much* smaller, but accuracy is much worse

Model	Accuracy	Latency (μs)	DSP (%)	LUT (%)	FF (%)
Original model (16x64x32x32x5)	0.75	0.06	60	7	1
Original model (75% compressed)	0.75	0.06	15	1.7	0.7
Small BNN (16x64x32x32x5)	0.62	0.04	-	0.8	0.1
Optimized BNN (16x448x224x224x5)	0.72	0.21	-	15	7
BNN w/ReLu (16x128x64x64x5)	0.70	0.140	4	6	1
Optimized TNN (16x128x64x64x64x5)	0.72	0.11	-	6	1
TNN w/ReLu (16x64x32x32x5)	0.68	0.06	2	2	0.2

BNN - Jet Classification

- 7x larger Binary NN uses no DSPs, slightly more LUTs and FFs (max % utilisation lower)
- Accuracy drops from 0.75 to 0.72

Model	Accuracy	Latency (μs)	DSP (%)	LUT (%)	FF (%)
Original model (16x64x32x32x5)	0.75	0.06	60	7	1
Original model (75% compressed)	0.75	0.06	15	1.7	0.7
Small BNN (16x64x32x32x5)	0.62	0.04	-	0.8	0.1
Optimized BNN (16x448x224x224x5)	0.72	0.21	-	15	7
BNN w/ReLu (16x128x64x64x5)	0.70	0.140	4	6	1
Optimized TNN (16x128x64x64x64x5)	0.72	0.11	-	6	1
TNN w/ReLu (16x64x32x32x5)	0.68	0.06	2	2	0.2

BNN - Jet Classification

- Can use a smaller model with 1-bit weights, ReLu activations. Still a bit worse than original
- Using ReLu activation reintroduces some DSPs (for Batch Norm)

Model	Accuracy	Latency (μs)	DSP (%)	LUT (%)	FF (%)
Original model (16x64x32x32x5)	0.75	0.06	60	7	1
Original model (75% compressed)	0.75	0.06	15	1.7	0.7
Small BNN (16x64x32x32x5)	0.62	0.04	-	0.8	0.1
Optimized BNN (16x448x224x224x5)	0.72	0.21	-	15	7
BNN w/ReLu (16x128x64x64x5)	0.70	0.14	4	6	1
Optimized TNN (16x128x64x64x64x5)	0.72	0.11	-	6	1
TNN w/ReLu (16x64x32x32x5)	0.68	0.06	2	2	0.2

BNN - Jet Classification

- Ternary NN can be smaller than Binary NN for same performance (smaller architecture, smaller resources, lower latency)

Model	Accuracy	Latency (μs)	DSP (%)	LUT (%)	FF (%)
Original model (16x64x32x32x5)	0.75	0.06	60	7	1
Original model (75% compressed)	0.75	0.06	15	1.7	0.7
Small BNN (16x64x32x32x5)	0.62	0.04	-	0.8	0.1
Optimized BNN (16x448x224x224x5)	0.72	0.21	-	15	7
BNN w/ReLu (16x128x64x64x5)	0.70	0.14	4	6	1
Optimized TNN (16x128x64x64x64x5)	0.72	0.11	-	6	1
TNN w/ReLu (16x64x32x32x5)	0.68	0.06	2	2	0.2

BNN - Jet Classification

- Ternary NN with ReLu again smaller than Binary equivalent

Model	Accuracy	Latency (μs)	DSP (%)	LUT (%)	FF (%)
Original model (16x64x32x32x5)	0.75	0.06	60	7	1
Original model (75% compressed)	0.75	0.06	15	1.7	0.7
Small BNN (16x64x32x32x5)	0.62	0.04	-	0.8	0.1
Optimized BNN (16x448x224x224x5)	0.72	0.21	-	15	7
BNN w/ReLu (16x128x64x64x5)	0.70	0.14	4	6	1
Optimized TNN (16x128x64x64x64x5)	0.72	0.11	-	6	1
TNN w/ReLu (16x64x32x32x5)	0.68	0.06	2	2	0.2

Other activities

- Many features under development coming soon:
- CNNs - already have support, but working to scale up to larger models
- Recurrent NNs, Graph NNs, Autoencoders
- Multi-FPGA inference - for low latency inference of large models, split model across devices
- BDTs - not NNs, but can be fast and lightweight
- Other vendors / tools: Intel FPGAs with Quartus HLS, and Mentor Catapult HLS

Conclusion

- hls4ml software package translates trained neural networks into synthesizable FPGA firmware
- User can tune resource usage vs. latency/throughput with *reuse factor*
- Initially targeting Level 1 Trigger - big FPGAs, O(1 μs) latency
- Compression techniques can greatly improve the resource usage in the FPGA
  - L1 Regularization to zero weights
  - Binary / Ternary NNs with low precision in each weight
- fastmachinelearning.org
- arxiv.org/abs/1804.06913

BNN - Dense Layer

- DSPs often limiting FPGA resource
- Encode ‘-1’ as ‘0’
- Multiplication become XNOR, sum becomes bitcount

A	B	A*B
-1	-1	1
-1	1	-1
1	-1	-1
1	1	1

A	A’
-1	0
1	1

A	B	A==B
0	0	1
0	1	0
1	0	0
1	1	1

Original: 16-bit weights

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

activation function — precomputed and stored in BRAMs

multiplication — DSPs

addition — logic cells

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

activation function — simple binary tanh / sign function

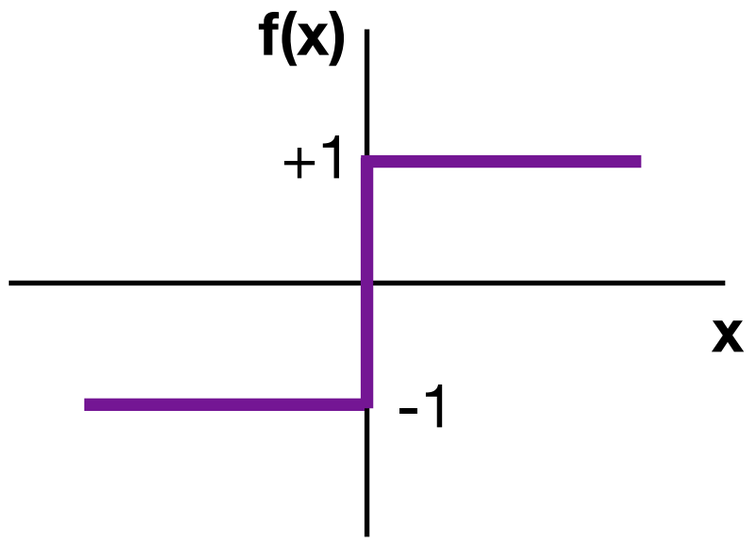
xnor — logic cells

no bias

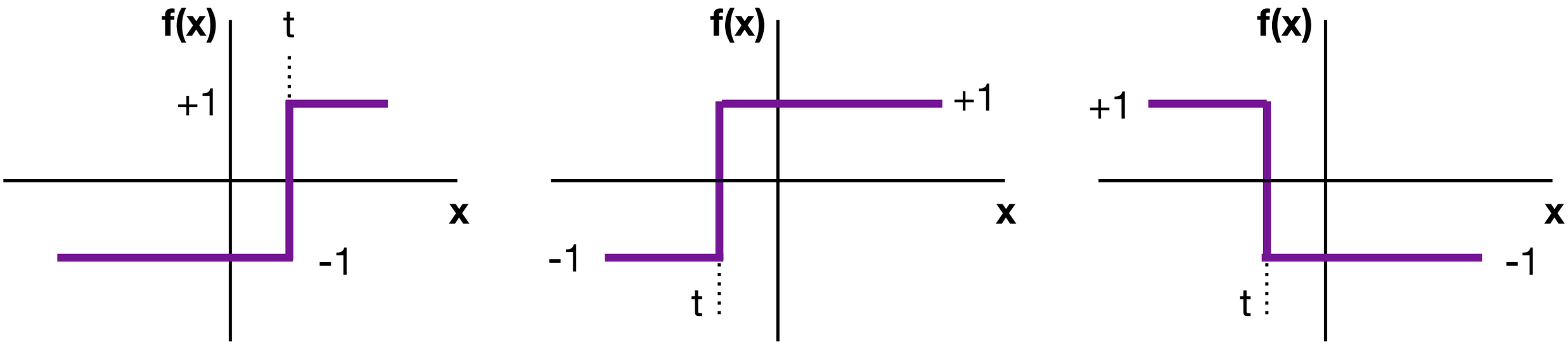
Binarized: 1-bit weights

BNN - Activation

- Using '*binary tanh*' activation function: f(x) = sign(x) : +1 if input is +ve, -1 if input is -ve



- Can be merged with *Batch Normalization* layer $y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta$
- After *binary tanh*, becomes a shift or inversion: +1 if input is > threshold, -1 if input is < threshold



Using **hls4ml**

- The model to translate
- Some test vectors for simulation (check precision)
- Output directory / name
- Target FPGA, clock speed

```
KerasJson: keras/KERAS_3layer.json
KerasH5:   keras/KERAS_3layer_weights.h5
#InputData: keras/KERAS_3layer_input_features.dat
#OutputPredictions: keras/KERAS_3layer_predictions.dat
OutputDir: my-hls-test
ProjectName: myproject
XilinxPart: xcku115-flvb2104-2-i
ClockPeriod: 5
```

- Model data precision and parallelisation
- More fine grained data precision and parallelisation
  - Per-layer, or per-layer type

```
IOType: io_parallel # options: io_serial/io_parallel
HLSConfig:
  Model:
    Precision: ap_fixed<16,6>
    ReuseFactor: 1
#  LayerType:
#    Dense:
#      ReuseFactor: 2
#      Strategy: Resource
#      Compression: True
```

- Then:

```
hls4ml convert -c my_model.yml
hls4ml build -p my-hls-test
```