

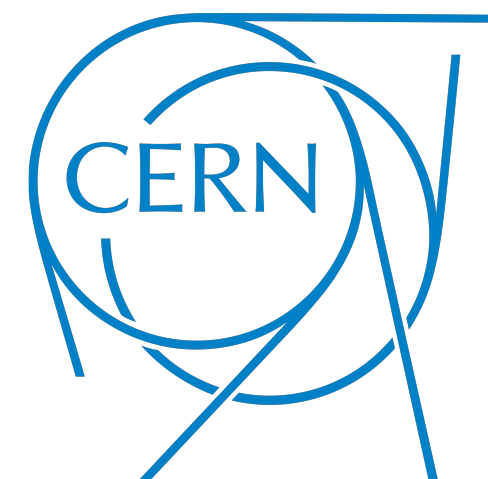
# Machine Learning for Trigger and DAQ

OPENLAB SUMMER LECTURES 2024

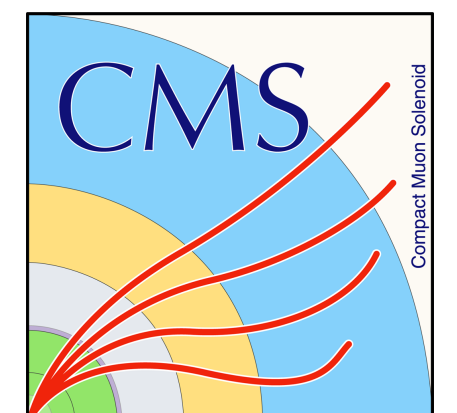
**Thomas James, CERN**

CERN openlab, CTO for AI and Edge Devices

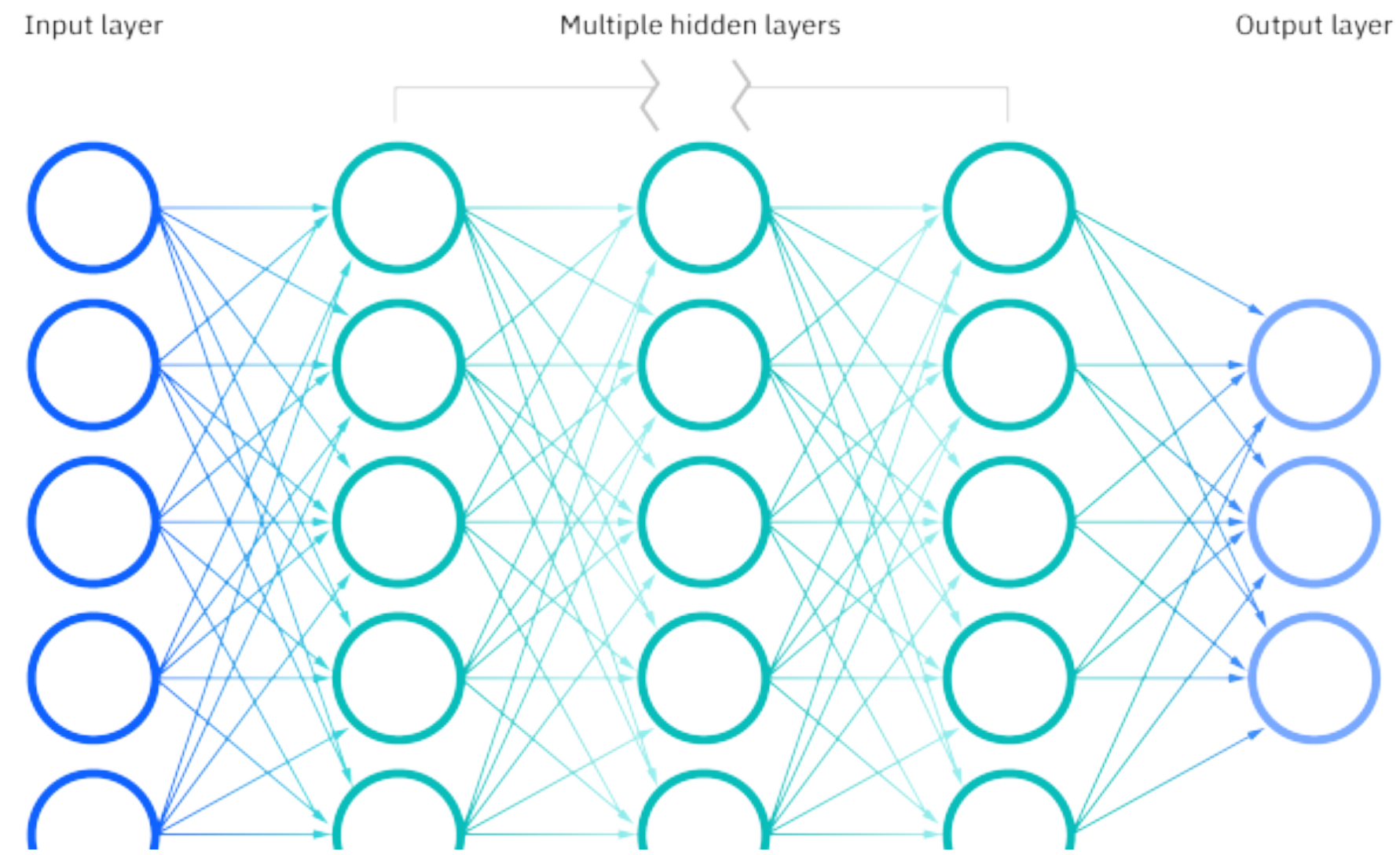
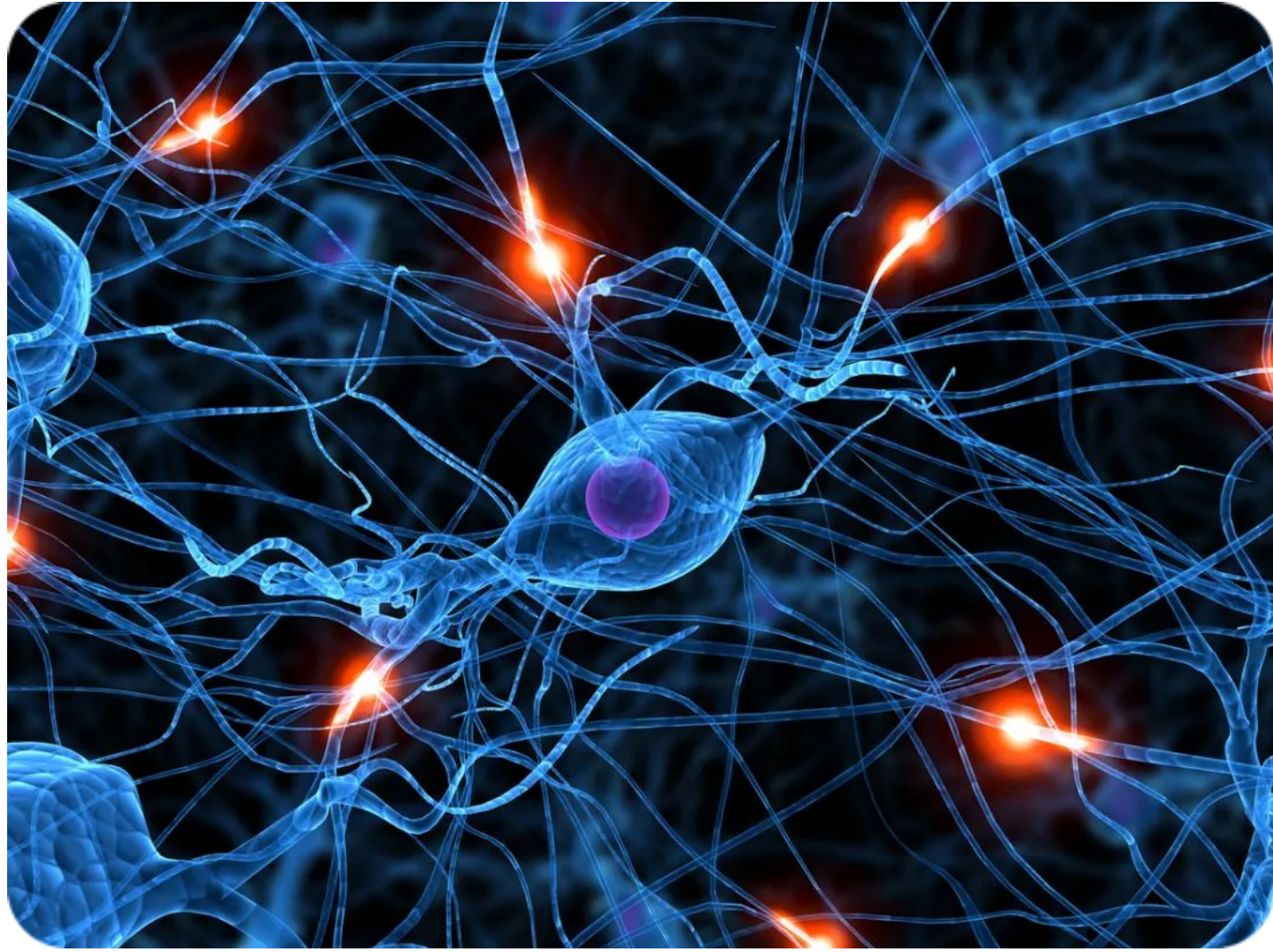
Applied Physicist, CMS



**CERN  
openlab**



Many thanks to Sioni Summers (CERN) for contributions from previous talks



---

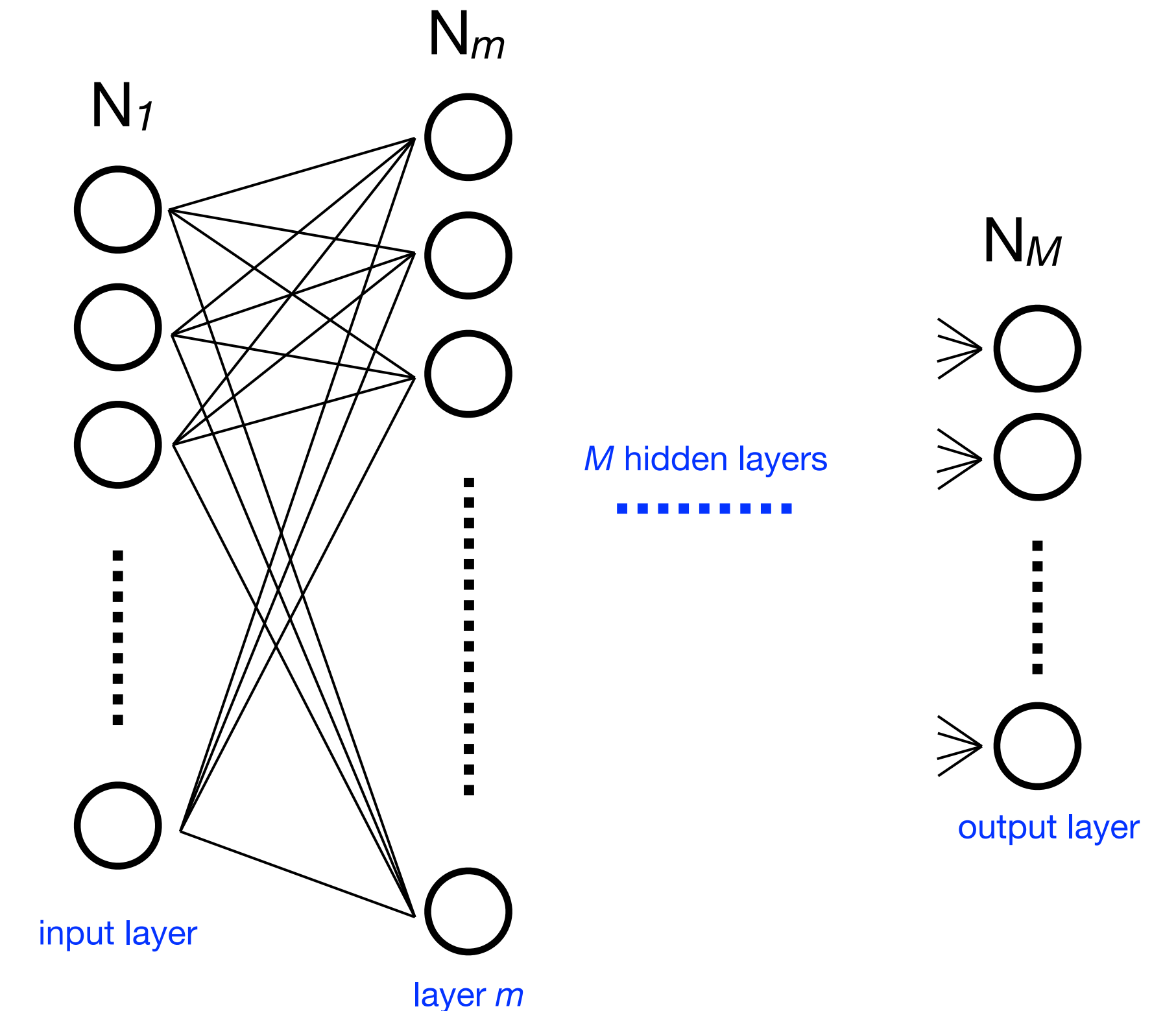
# Introduction/Recap of ML & NNs

---

# Neural Networks

- Loosely inspired by brain structure with neurons and synapses
  - Neurons are real valued representations of ‘something’
  - Synapses connect neurons (in one direction) with a *weight*
- Input neurons are your data variables
- Output neuron(s) are your predictions
  - class probabilities,
  - or continuous variables if performing a regression
- Hidden layers bring the performance of deep neural networks
  - Intermediate layers of neurons learn a more abstract representation of the data
  - More capable than ‘shallow’ networks on raw data
- Many topologies exist for different types of problems

Fully Connected or Dense Neural Networks



# Neural Networks

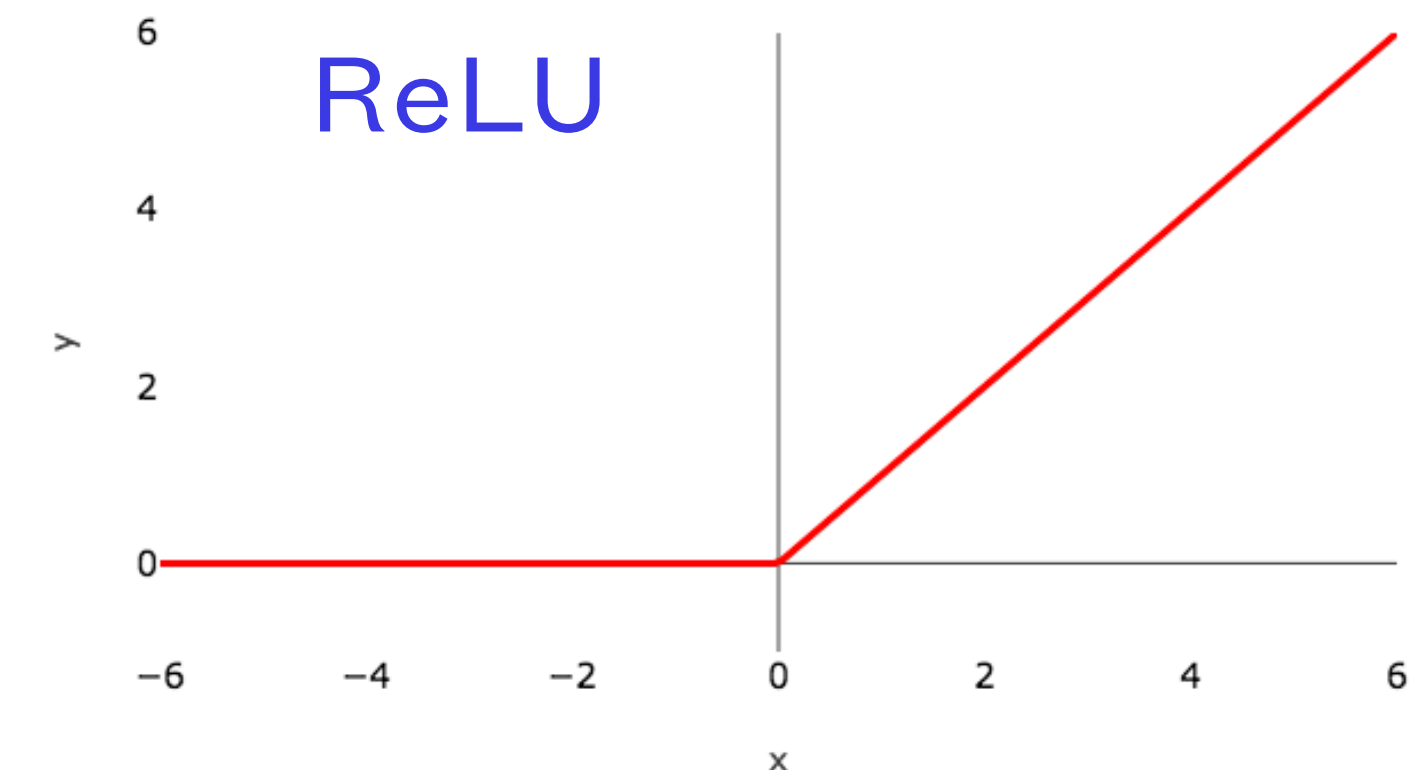
- The values of neurons in a layer is given by the product of the neuron values of the previous layer and the matrix of weights, with an added 'bias', and a non-linear 'activation function' applied

$$X_n = g_n(W_{n,n-1}x_{n-1} + b_n)$$

Non-linear activation function

Matrix of weights

Bias vector addition



- Without the activation function, we're just doing linear transformations of our variables
- Values of weights and biases learned from data during training

# Training with Gradient Descent

- *Supervised learning* - start with a NN of randomised weights and a collection of *training data*
- Evaluate performance network with a loss function, e.g. mean squared error:

$$L(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

Loss                      Truth      Prediction

$$w_j = w_j - l_r \frac{\partial L}{\partial w_j}$$

Weights                      Learning rate

The diagram illustrates the components of the loss function and the weight update equation. On the left, the loss function  $L(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$  is shown. Blue arrows point from the labels 'Loss', 'Truth', and 'Prediction' below to the corresponding parts of the equation: 'Loss' points to  $L(y, \hat{y})$ , 'Truth' points to  $y_i$ , and 'Prediction' points to  $\hat{y}_i$ . On the right, the weight update equation  $w_j = w_j - l_r \frac{\partial L}{\partial w_j}$  is shown. Blue arrows point from the labels 'Weights' and 'Learning rate' below to the corresponding parts of the equation: 'Weights' points to  $w_j$  and 'Learning rate' points to  $l_r$ .

- Minimise loss function to get the best performing network
  - Predictions as close to true labels as possible
- Update the (initially not very good) network parameters by evaluating the derivative of the loss function w.r.t those parameters, and iterate!

---

# Tools / Frameworks

- Many excellent software tools and frameworks are out there for building ML models, training and deploying them
- There are particularly good sets of tools in Python



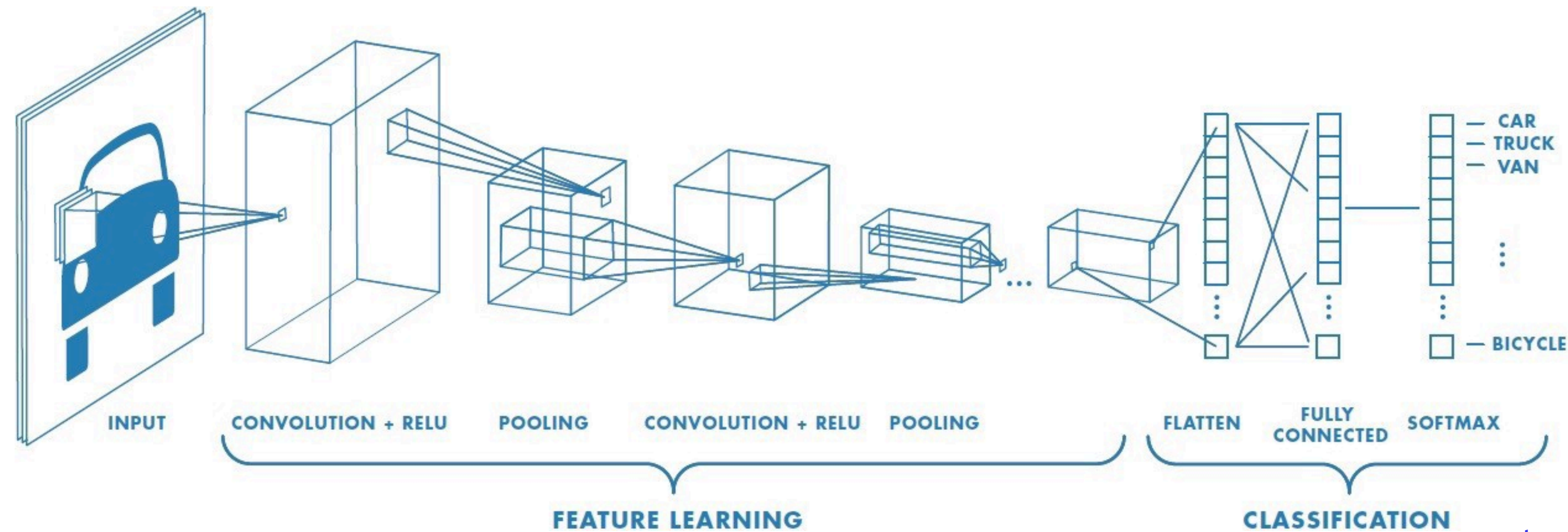
# Minimal example - *tensorflow* NN

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from sklearn.model_selection import train_test_split
import uproot

X, y, = uproot.open('data.root').arrays([...])
X_train, X_test, y_train, y_test = train_test_split(X, y)
inputs = Input(shape=(3,))
hidden = Dense(64, activation='relu', input_shape=2, name='hidden')(inputs)
output = Dense(1, activation='sigmoid', name='output')(hidden)
nn = Model(inputs=inputs, outputs=output)
nn.compile(optimizer="Adam", loss="binary_crossentropy", metrics=["accuracy"])
nn.fit(X_train, y_train, batch_size=100, epochs=10)
nn.save('nn.h5')
```

# Convolutional Neural Networks

- Convolutional Neural Networks for images: apply *convolutional filters* - small neural networks - scanning over the pixels
  - Reduces the number of parameters compared to feeding the pixels into a Fully Connected NN
  - Adds translational invariance: the object in the image could be anywhere, and is filtered down by the convolutions

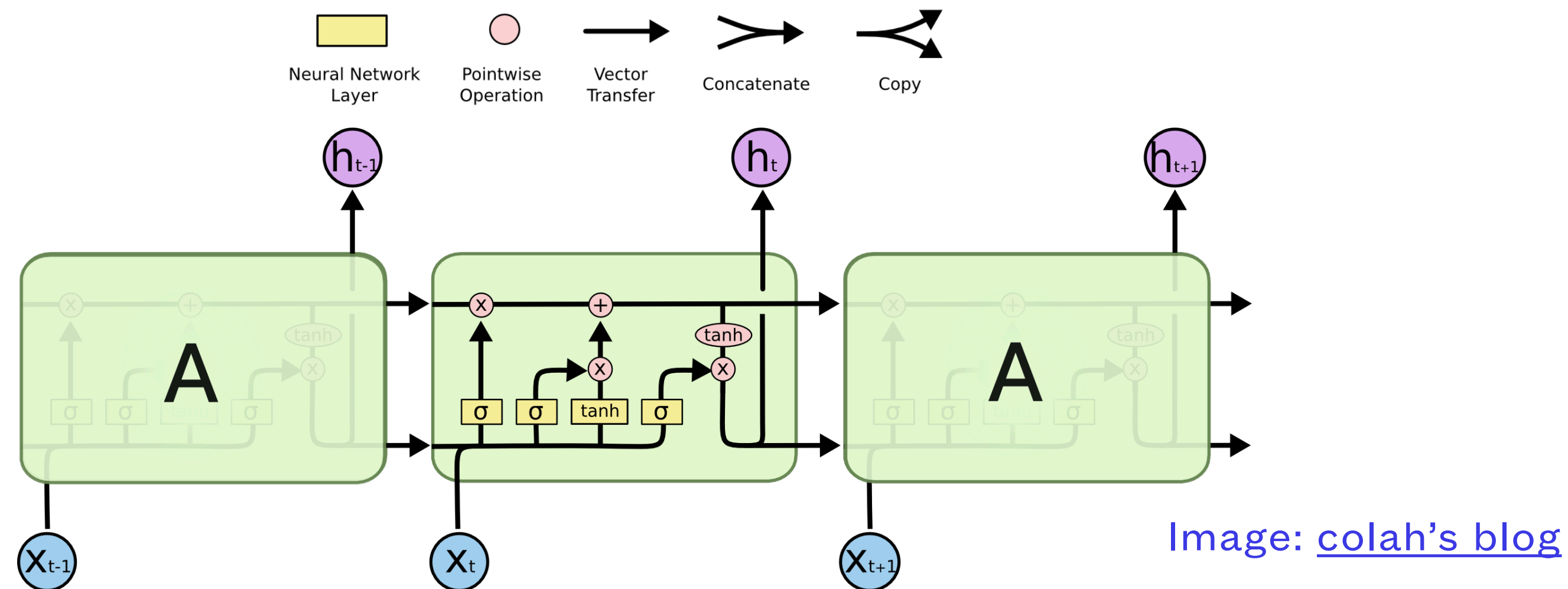


[towardsdatascience.com](https://towardsdatascience.com) [8]



# Recurrent Neural Networks

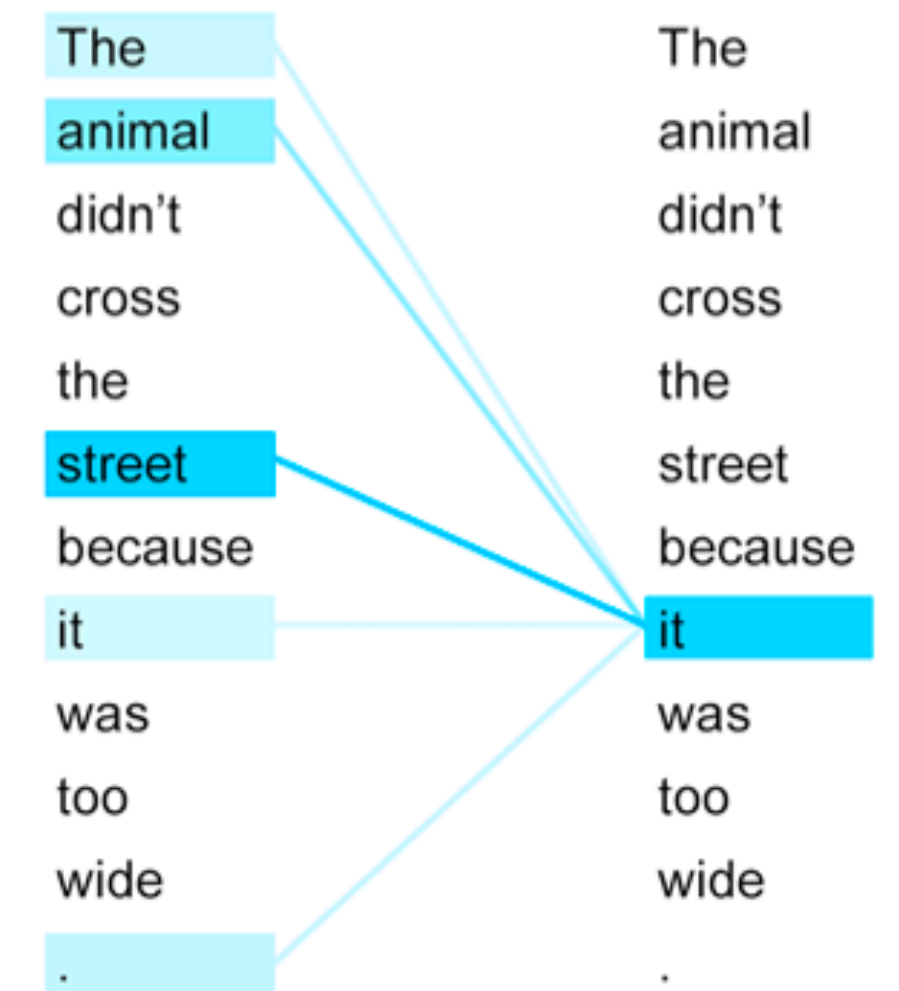
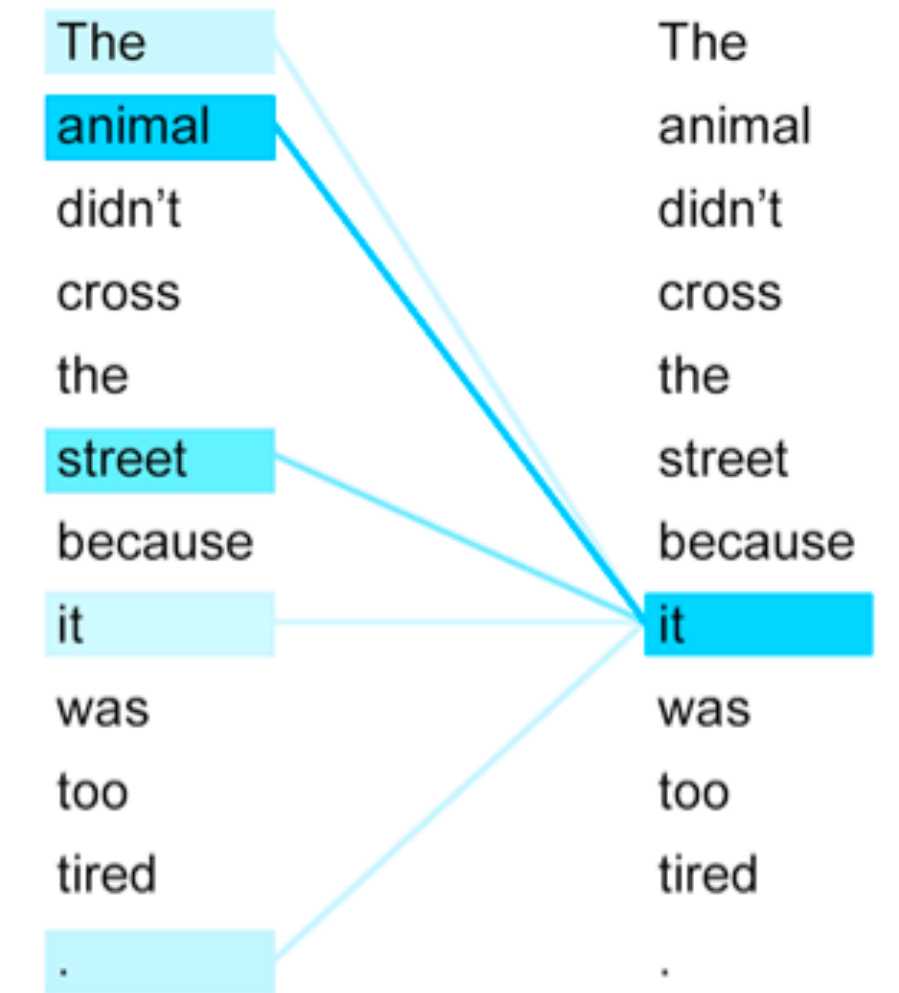
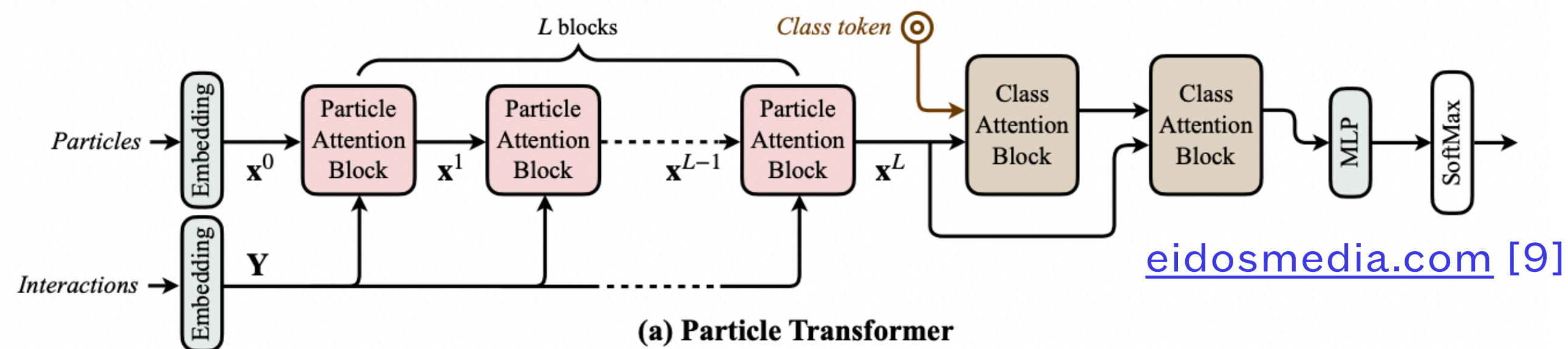
- Built in *memory*
- Used for ordered data, e.g. time series, natural language processing
- Few different flavours: Long Short Term Memory (LSTM), Gate Recurrent Unit (GRU)



- The LSTM cell has an internal state, and fully connected neural networks update this at each iteration
- Could be used, e.g. to predict the next word in a sentence

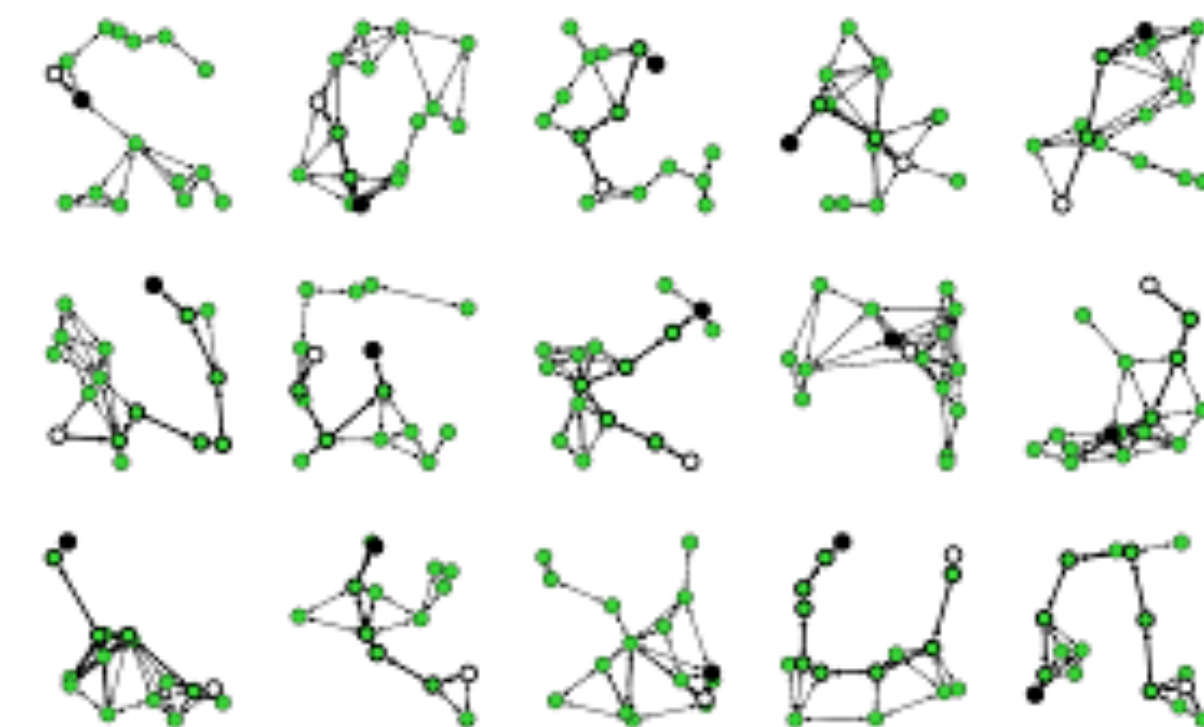
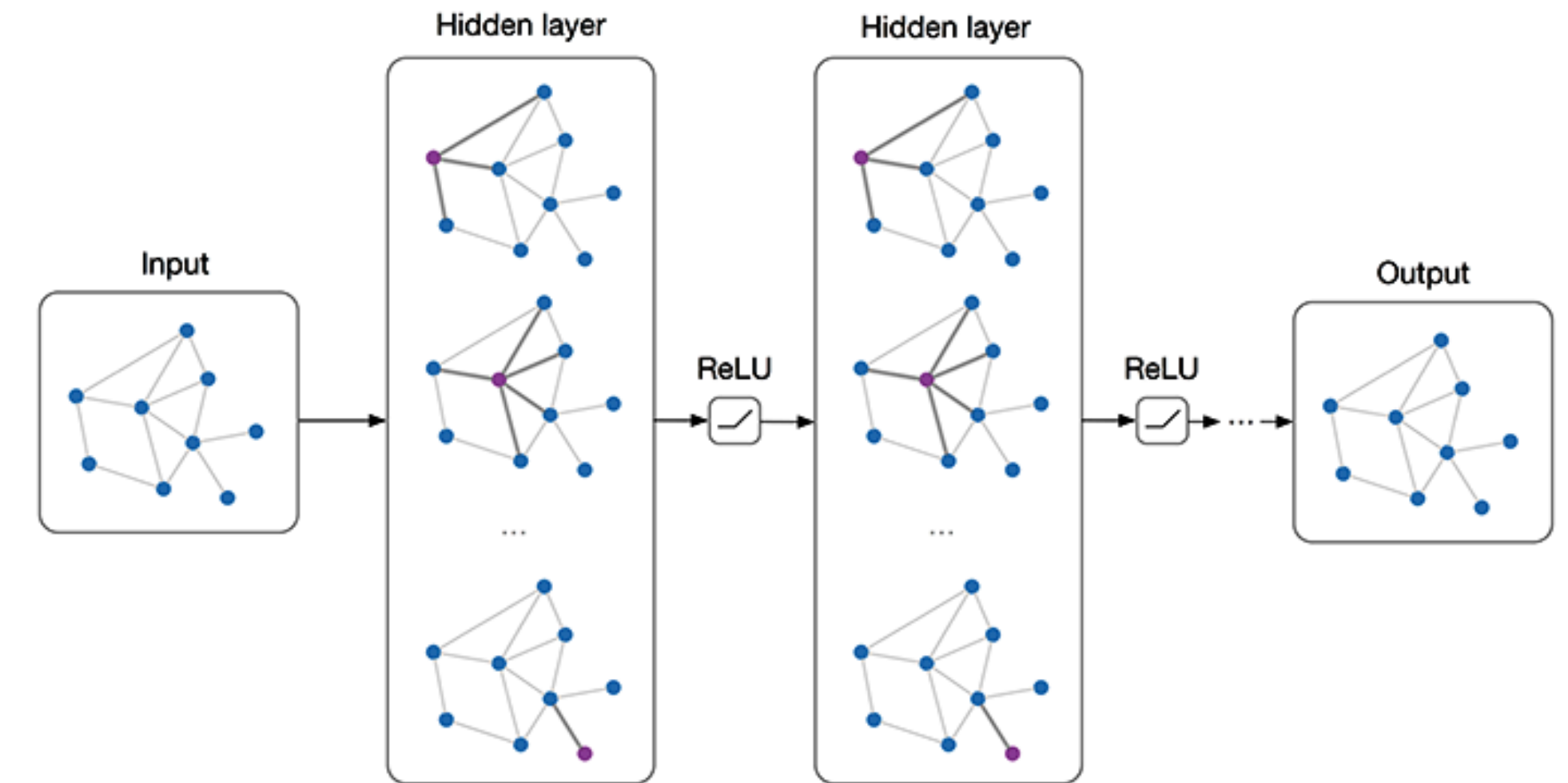
# Transformers

- Sequence-to-sequence type problems
  - The big Natural Language Processing (NLP) models like GPT
  - Billions of parameters
  - Unlike RNNs the full sequence enters at once - more parallelizable
- Attention mechanism - learning relationships / context
- Also relevant in HEP - Particle Transformer (ParT) (jet tagging)
- Better than CNN & RNN at long-distance connections



# Graph Neural Networks

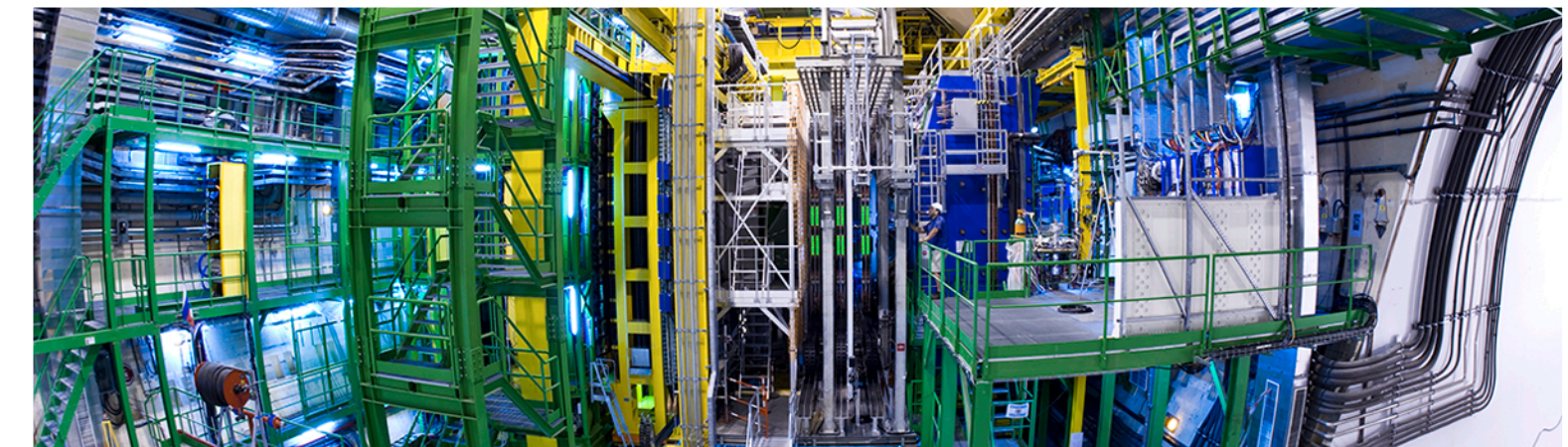
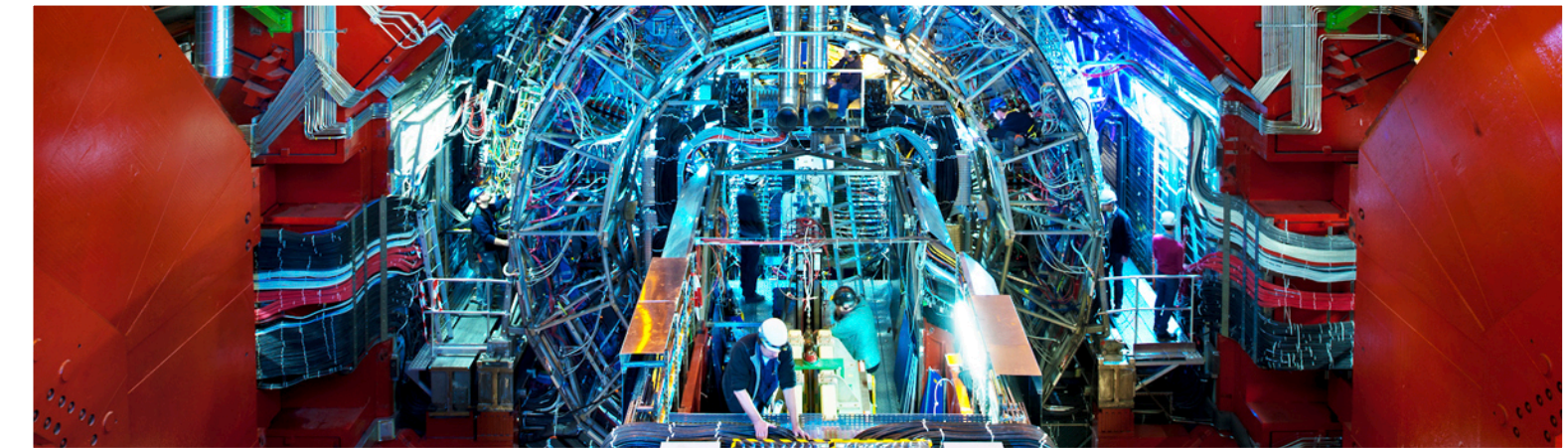
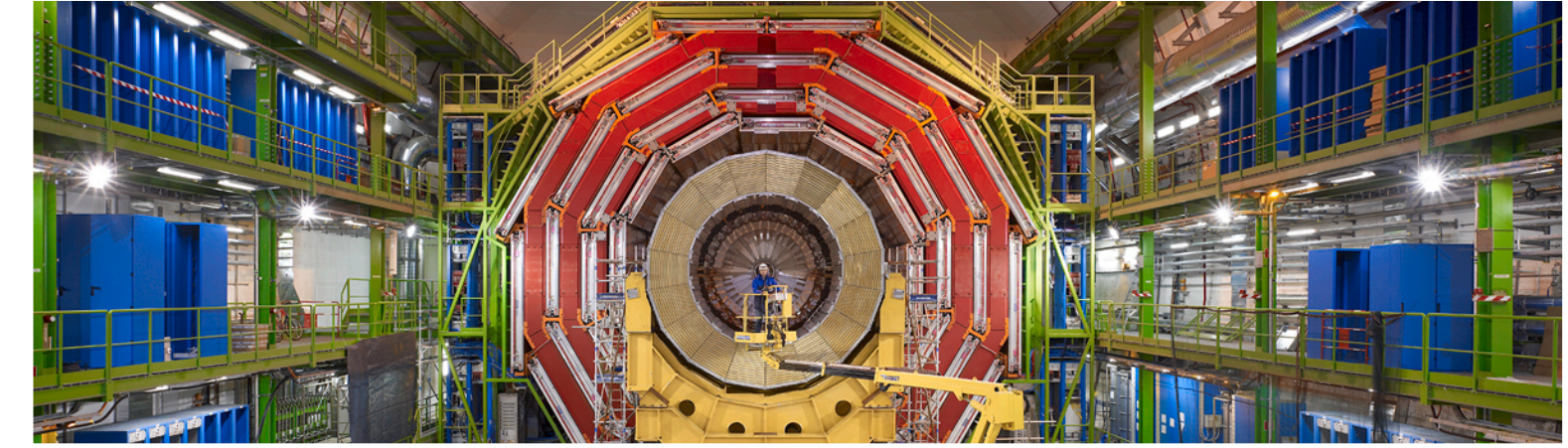
- Well suited to problems described by graphs of vertices and edges
- Cluster / classify data not only according to its coordinates, but its neighbourhood
- Iteratively update (strengthen/weaken) connections with fully connected or convolutional networks
- Used in, e.g., molecule synthesis for drug discovery
- Promising in HEP for multi-clusters in 'point cloud' like detectors (sparse images)
  - tracking, calorimetry in high pileup
  - hierarchical type problems, e.g. tracking, jets



---

# ML for TDAQ Overview

- ML algorithms **highly parallelisable**
  - NN forward pass just matrix-vector products and non-linear functions on vectors
- Can be **accelerated with appropriate hardware:**
  - CPUs with vector/SIMD units (e.g. AVX - get packages from Intel, for example)
  - GPU, FPGA, TPU (T = Tensor), IPU (I = Intelligence)
  - Need also good software and compilers to utilise hardware effectively
- Need to **(re)optimize ML models for online performance**
  - Tuning the learning rate, optimizer, loss function, activation function
  - Tune network architecture
    - Type of network, N layers, N neurons / layer
  - Hyperparameter scan / optimization - e.g. Keras Tuner, Ray Tune



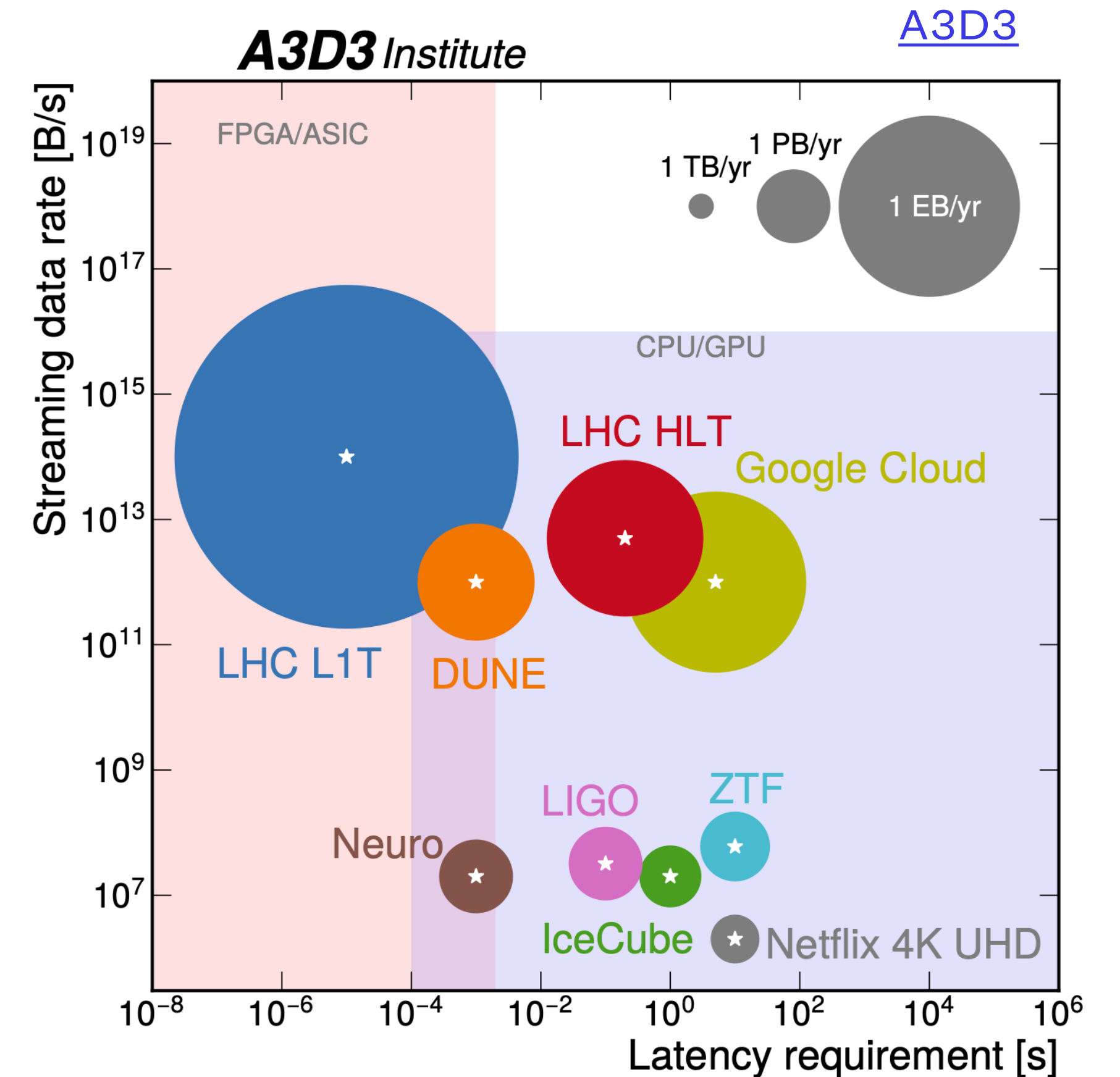
---

# Fast ML at LHC

---

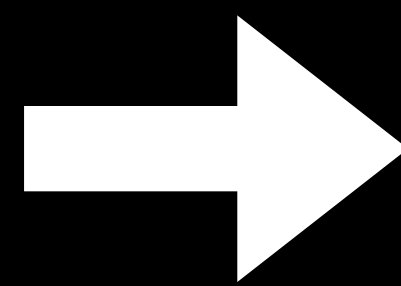
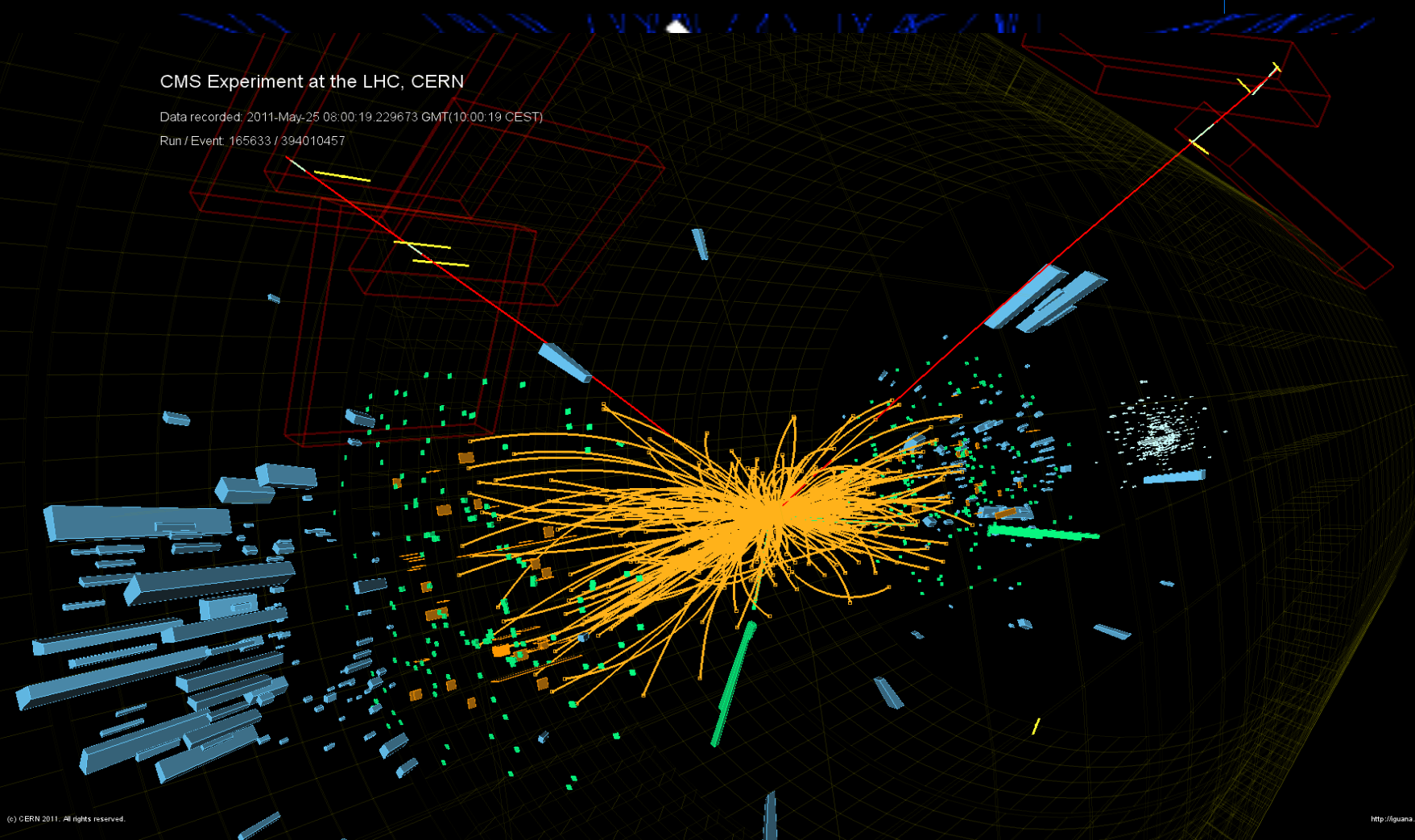
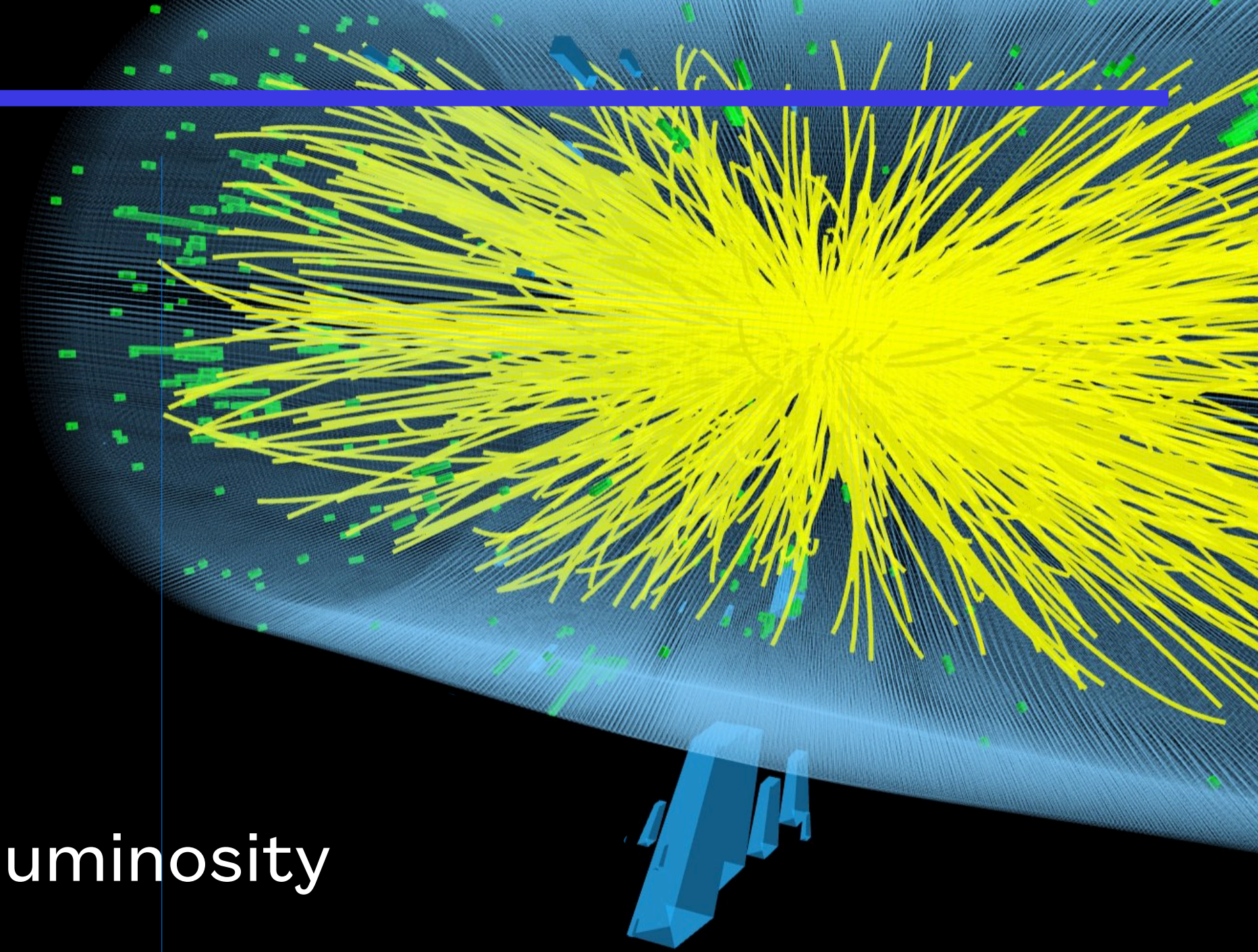
# Big data at the LHC

- LHC produces vast amounts of data, **billions of collisions per second** during operation
  - Without selection would generate **~ Pb/s** raw data for CMS & ATLAS
  - Impossible to readout/process/store all data
  - Need fast **trigger** to select interesting collisions for analysis with high efficiency, low fake rate
    - **Particles of interest rare** among background
    - e.g Higgs produced **~1** in a billion collisions
- Two layered selection:
  - Hardware-level: Fixed latency of  $\sim \mu\text{s}$   $\rightarrow$  FPGAs required
  - Software-level: Flexible latency  $\sim 100$  ms compute / event  $\rightarrow$  CPUs/GPUs

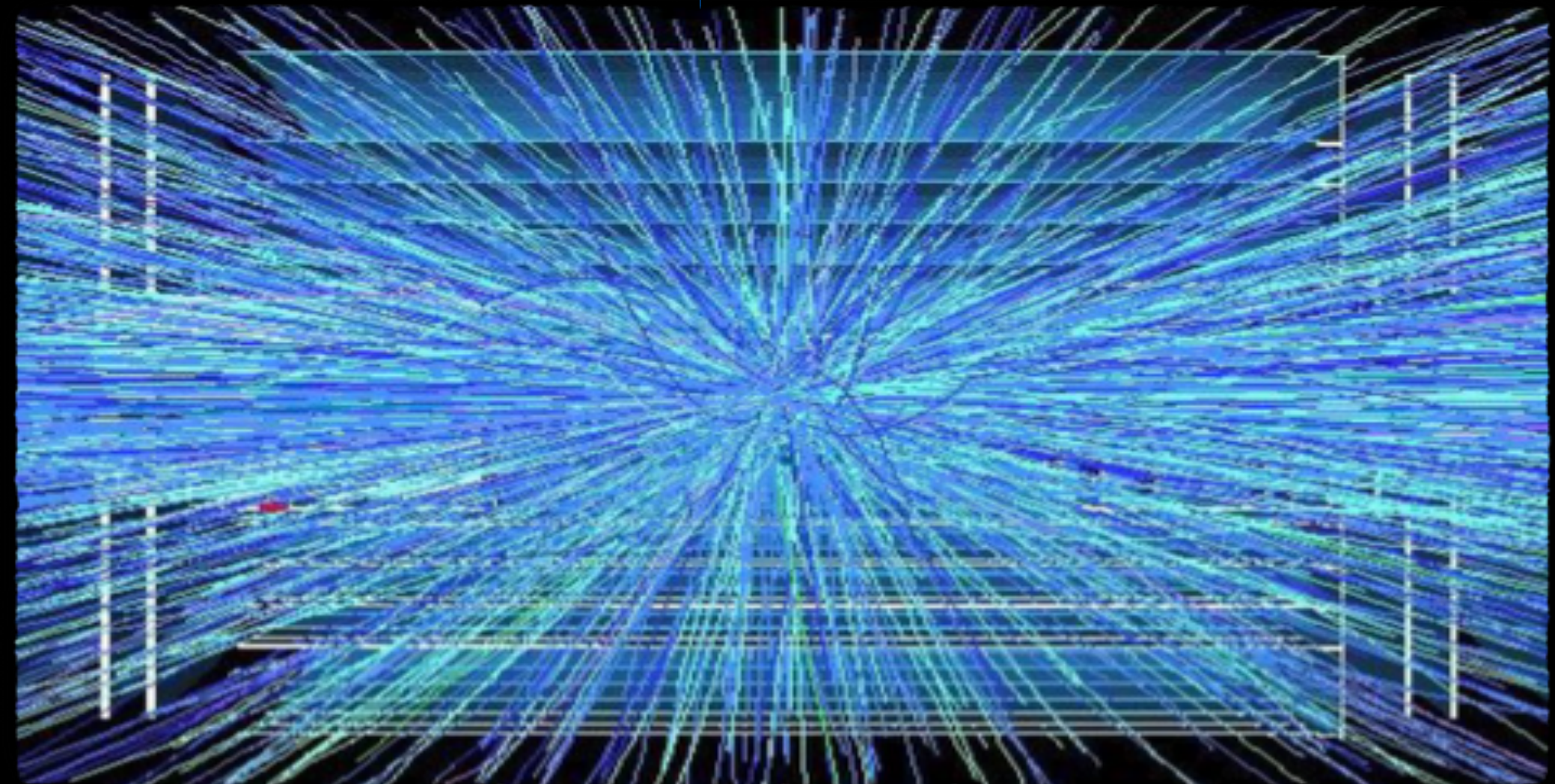


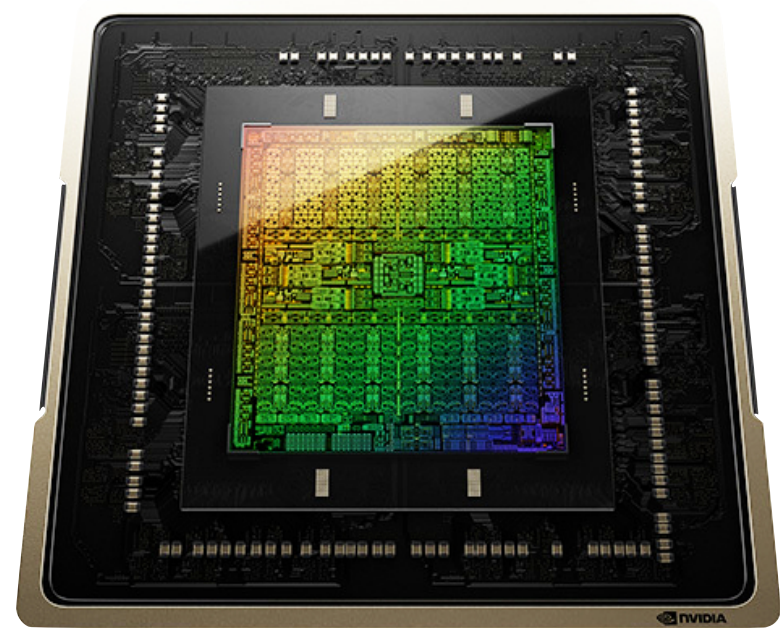
# AI inference at the edge

- Machine learning being exploited across particle physics
- ML allows us to *speed up* data processing by training networks on much more complex algorithms than implementable within latency constraints
- High-Luminosity LHC upgrade from ~2029 -> ~5x increase in luminosity
- Fast ML *at the edge* needed for reducing and filtering data in real-time; *train offline, predict online*



High  
Luminosity  
upgrade





---

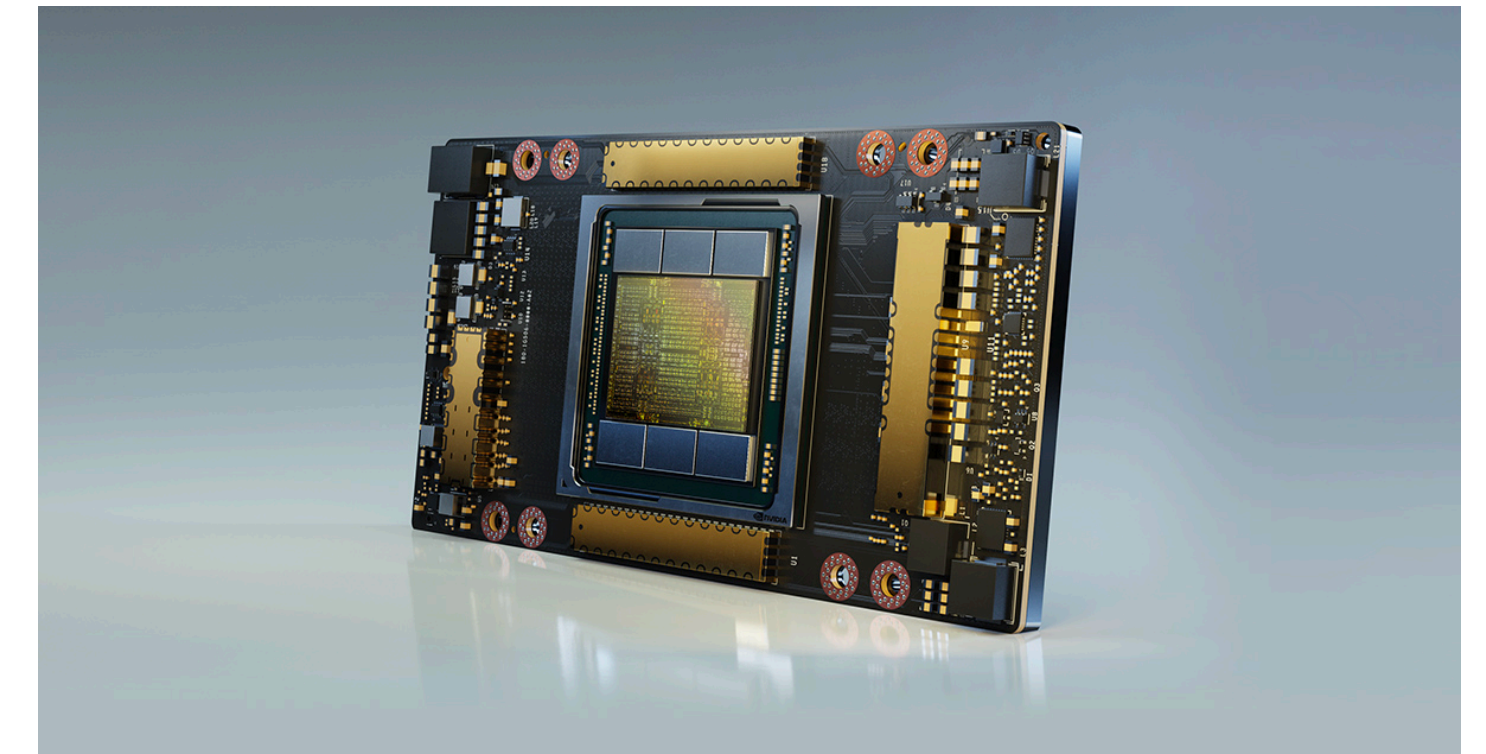
# ML with GPUs

---



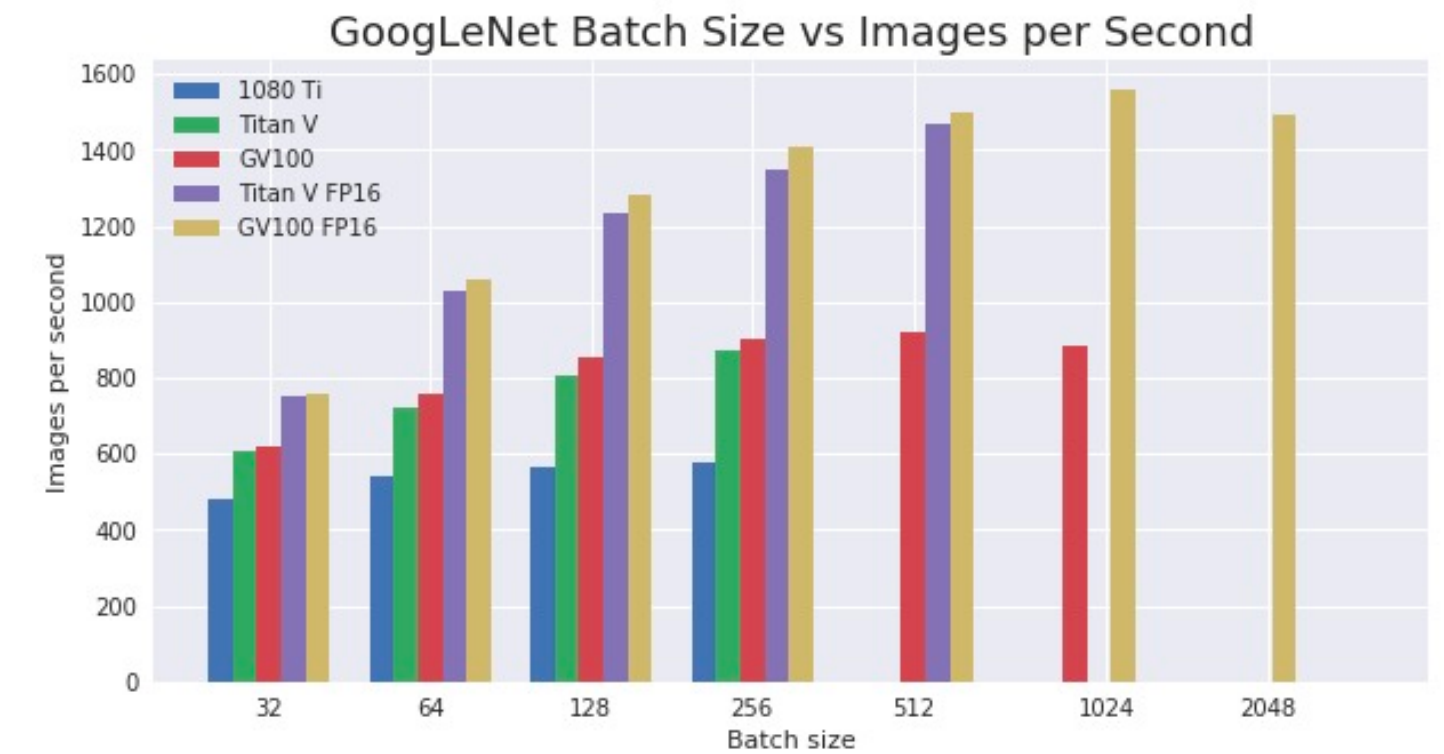
# GPUs for ML

- GPUs are very powerful for machine learning
  - Many more parallel arithmetic ops than a CPU
  - Very high memory bandwidth
  - Training / predicting ML models on large datasets doesn't involve much branching/control
  - Plus the GPU can be useful for other things
- Usually, using GPUs for ML, you don't write CUDA code yourself but use a higher level framework like Tensorflow (or higher still with Keras, PyTorch)
  - Extremely easy to execute on a GPU with these environments
  - Exception might be when doing something extremely custom
- Biggest gains in training, but also beat CPUs in inference
  - But remember you have to get the data to the device

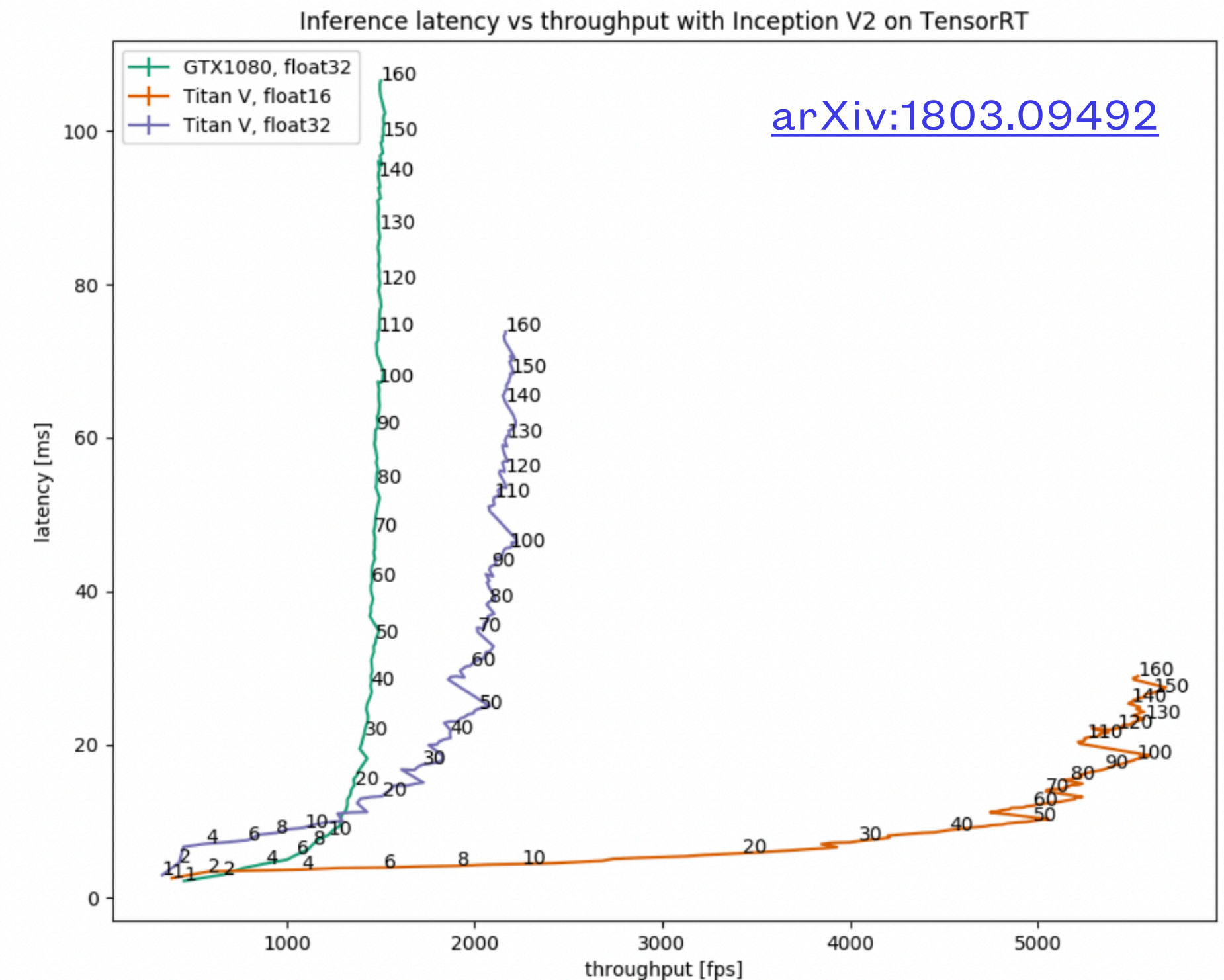


# GPUs for ML - batching

Puget Systems

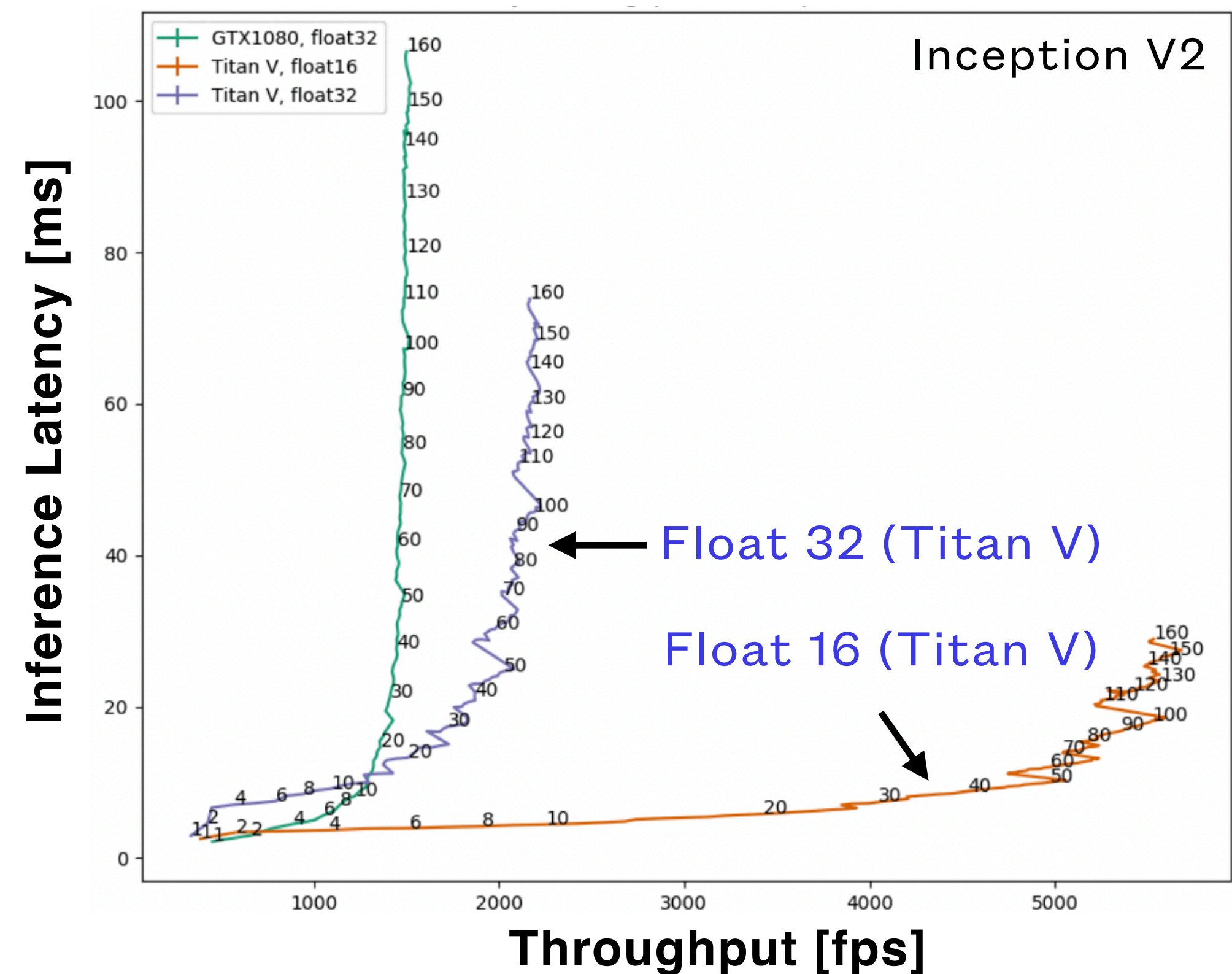
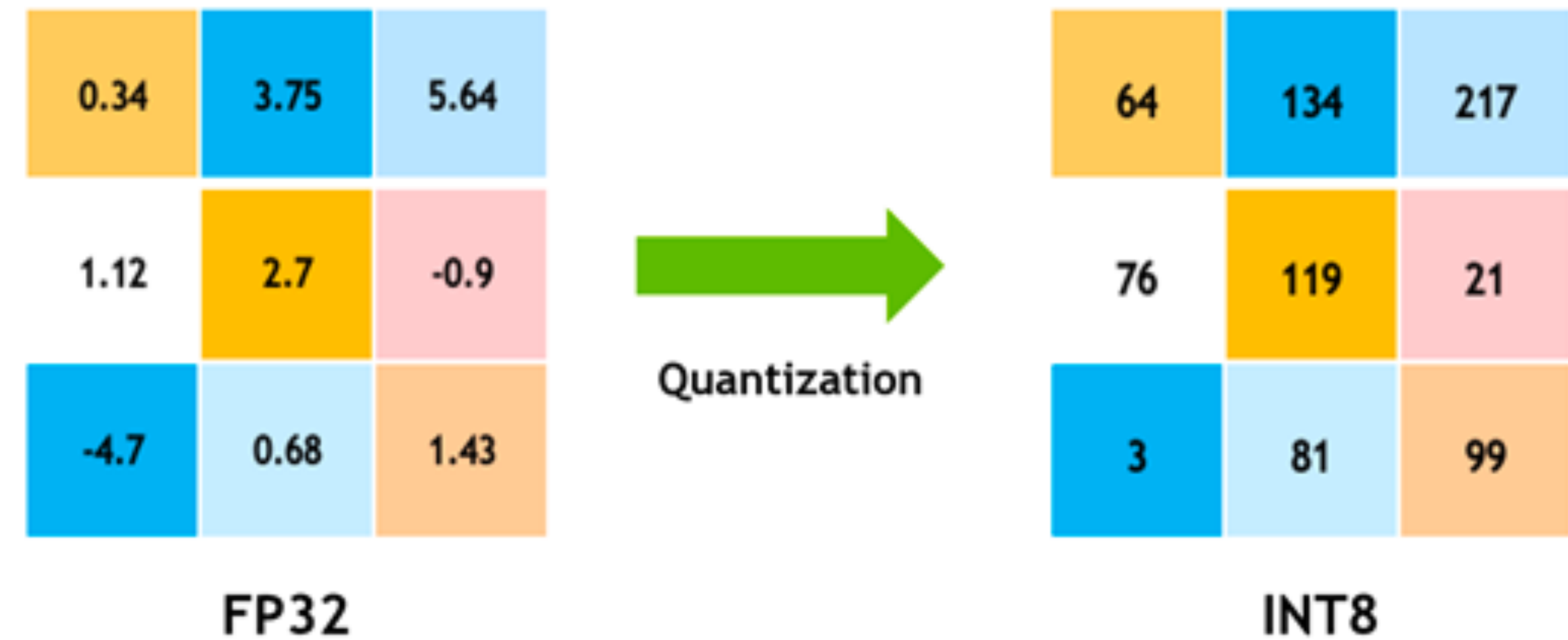


- **Batching**: a common technique for better hardware utilisation
  - Relevant both at training and inference time
  - Send several data samples to the GPU in one *batch*
    - Maximise use of memory bandwidth and compute
  - Is the constraint latency or throughput?
    - If strictly latency: low batch size
    - If throughput: high batch size
    - Both: batch size where throughput saturates
- [mlperf.org](http://mlperf.org) has nice benchmarking of different hardware (not only GPUs) running on different models



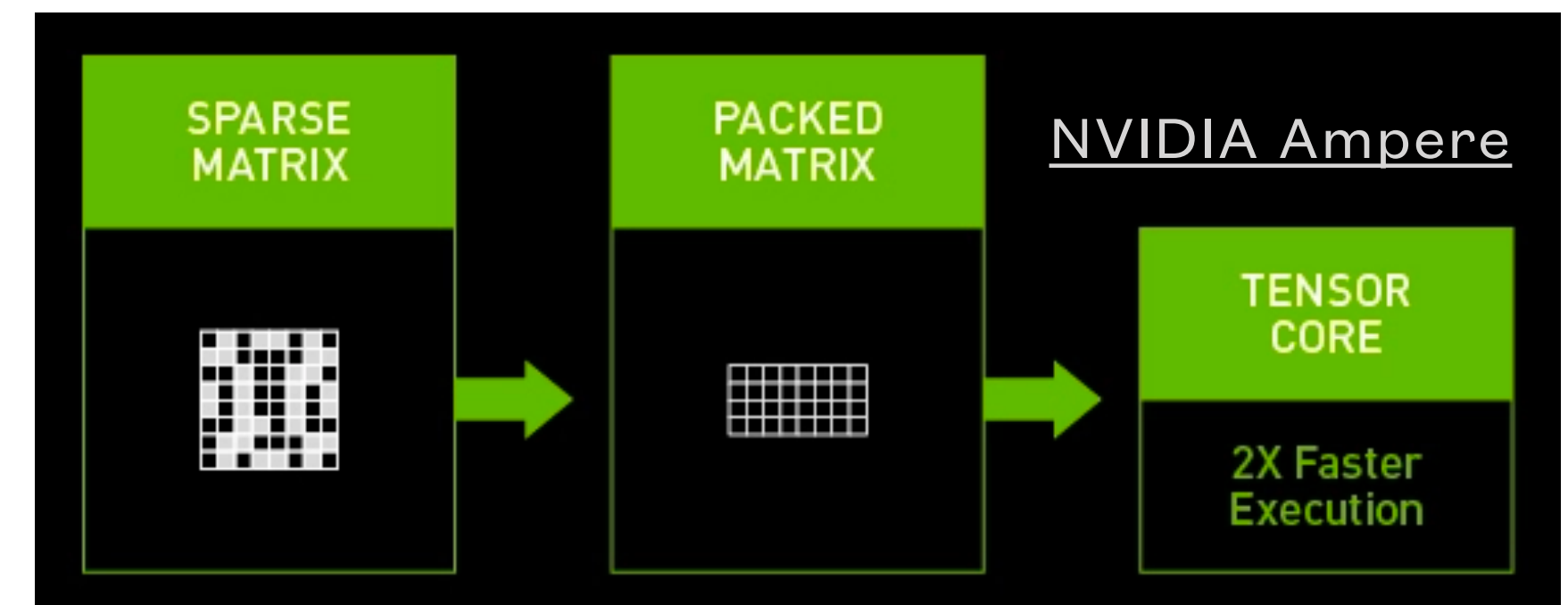
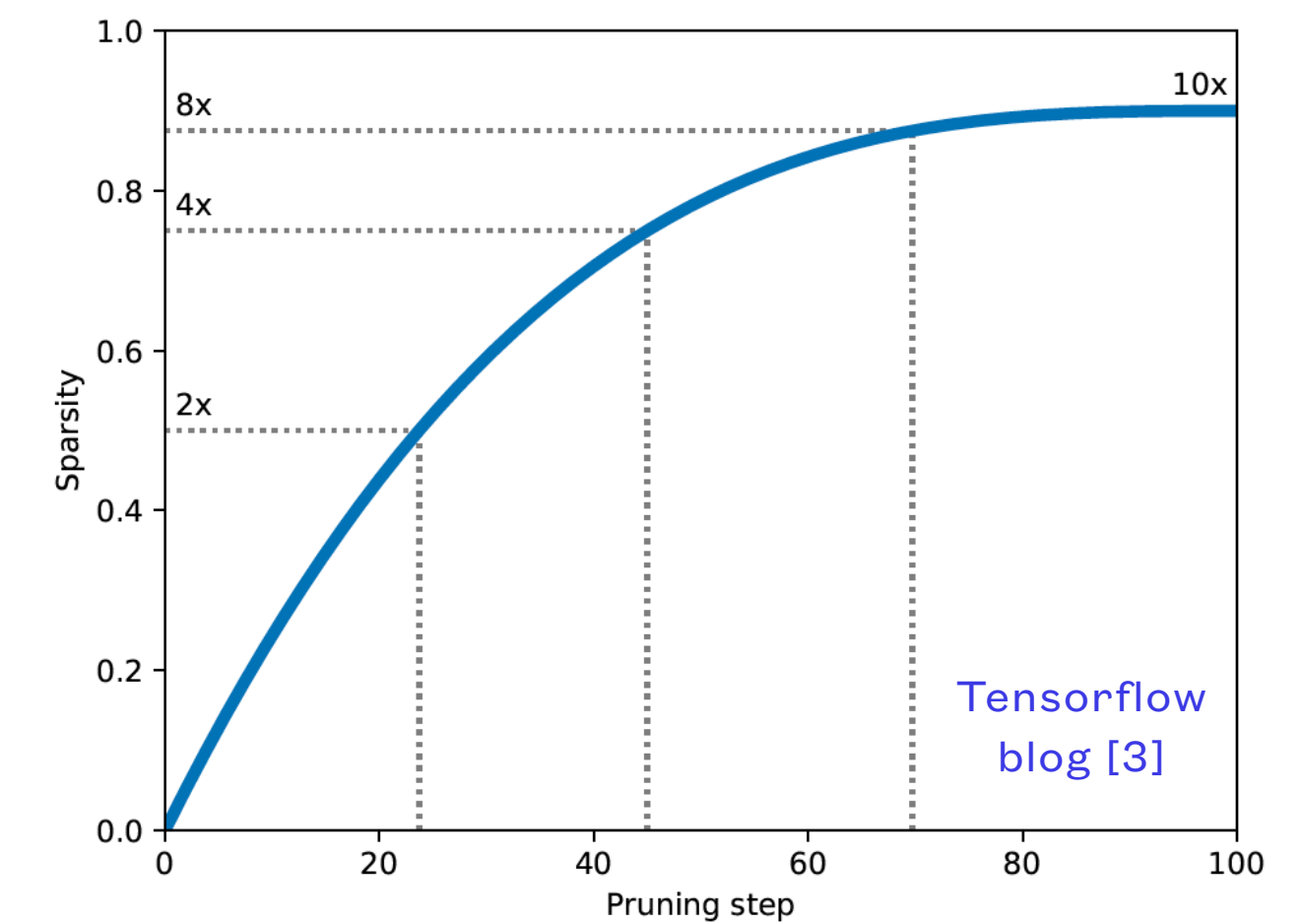
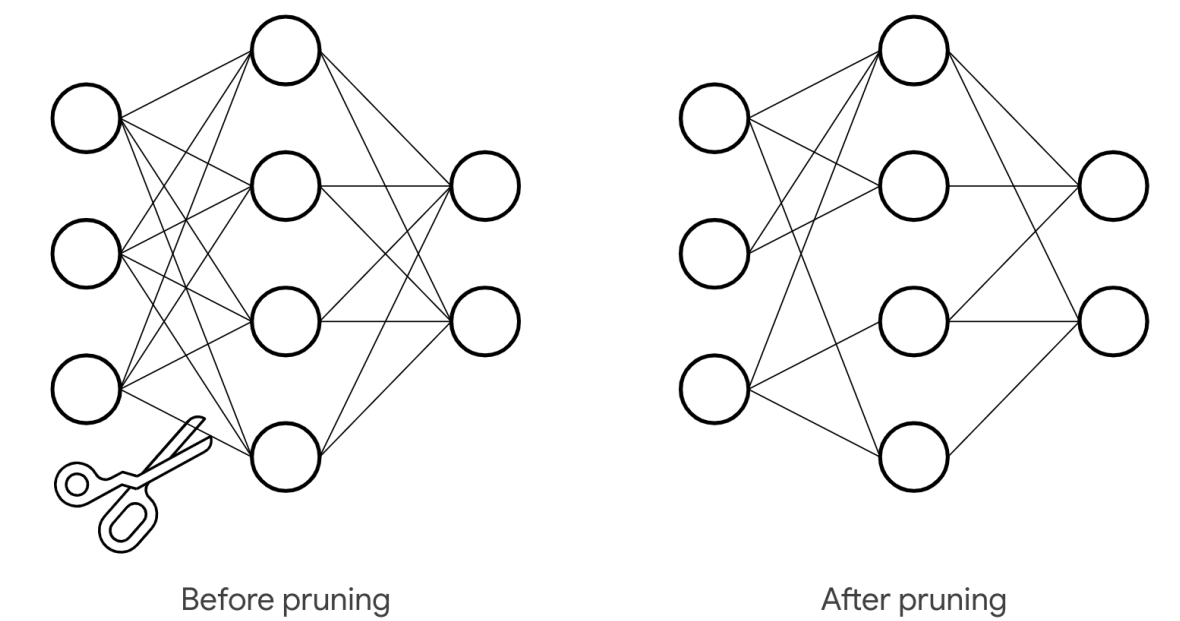
# Quantization

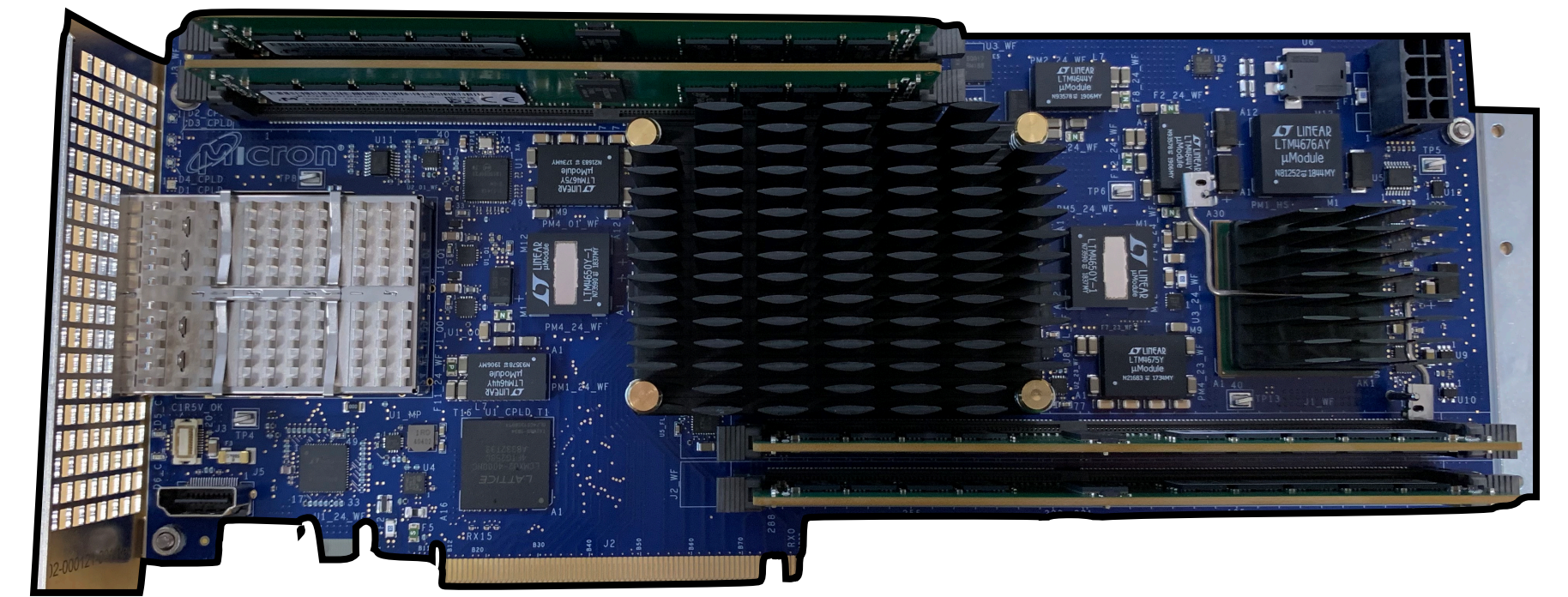
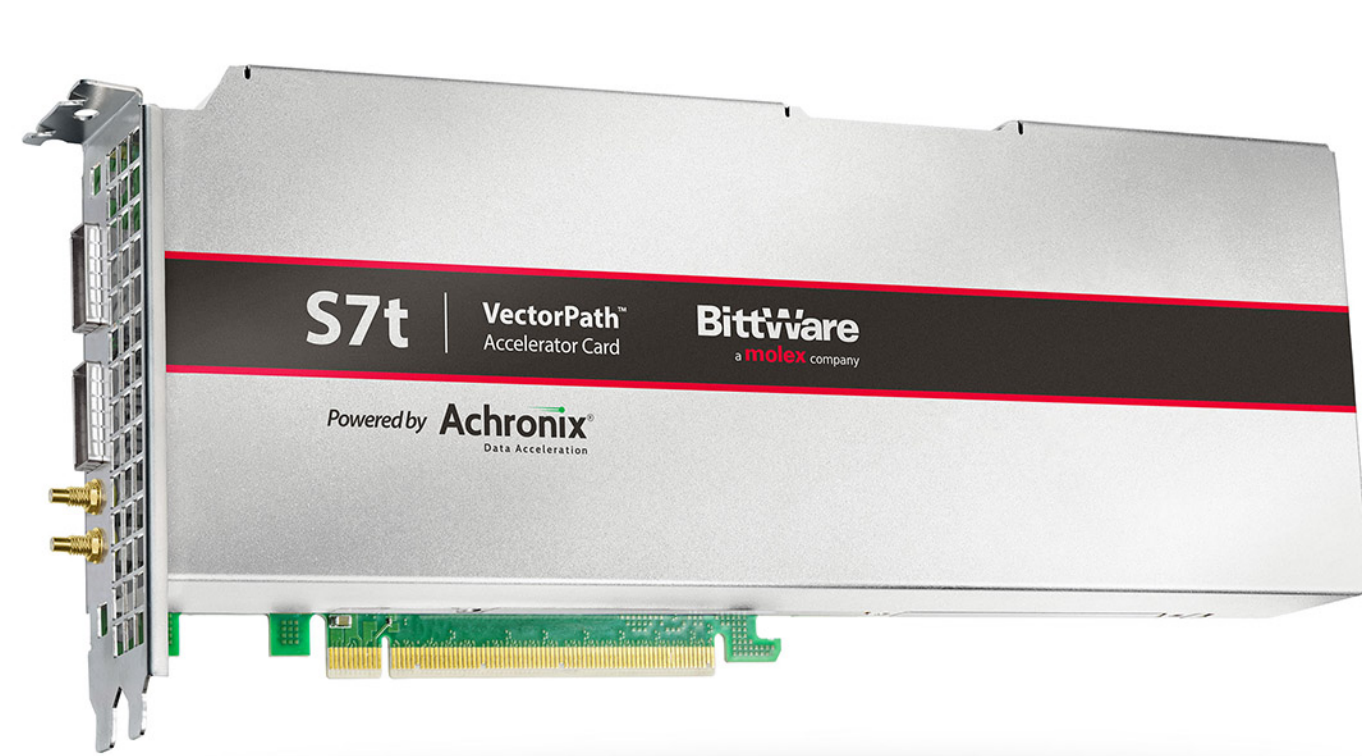
- Many GPUs support Int8, float16, bfloat16 precision with many more OPS than float32
- **Post Training Quantization (PTQ)** - train with FP32 then scale & round to lower precision
- **Quantization Aware Training (QAT)** - train with lower precision
  - TensorRT (NVIDIA GPU),
  - TensorFlow Lite (Google),
  - torch.quantization (PyTorch)
- Choice of precision depends on target hardware and requirements



# Pruning / Sparsity

- NN often contains many redundant connections
- Pruning: remove some connections from final model
- Can reduce model size (memory footprint)
- Some processors can skip/accelerate multiply by zero calc.
- Methods:
  - Regularisation (penalise low value weights, then make them 0)
  - Target sparsity,
  - Structured pruning - remove continuous blocks of weights
  - Filter pruning - entire filters of CNN
- Applies also to BDTs ( $\lambda$ ,  $\alpha$  in xgboost)
- Can be coupled with Quantisation Aware Training





---

# ML inference with FPGAs

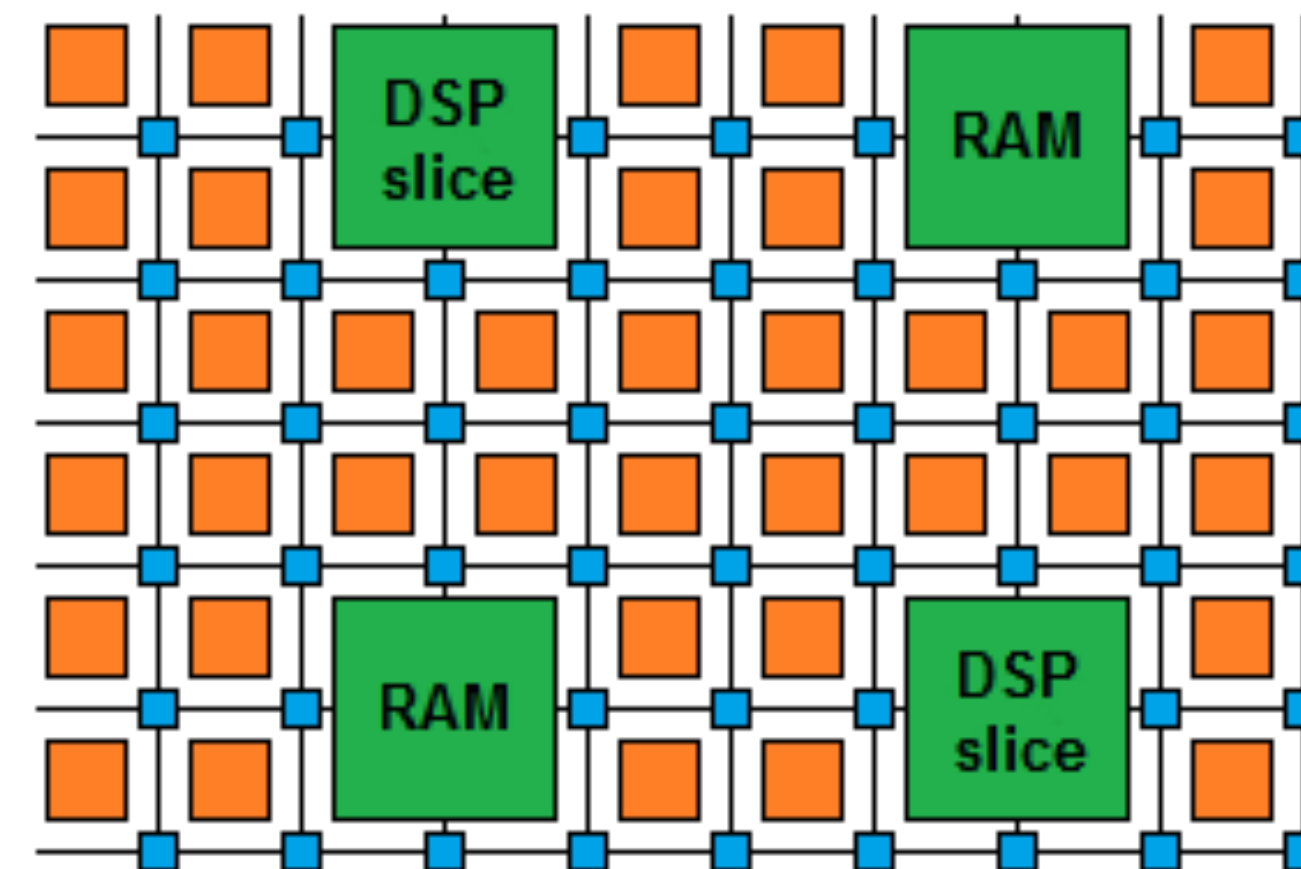
---

# What are FPGAs?

- Field Programmable Gate Arrays = reprogrammable integrated circuits
- Contain many different building blocks (*resources*) which are connected together as desired
- Extremely parallel processors
- *Computing in space as well as time*
- Utilised by most low level HEP triggers



## FPGA diagram



LUTs - generic logic

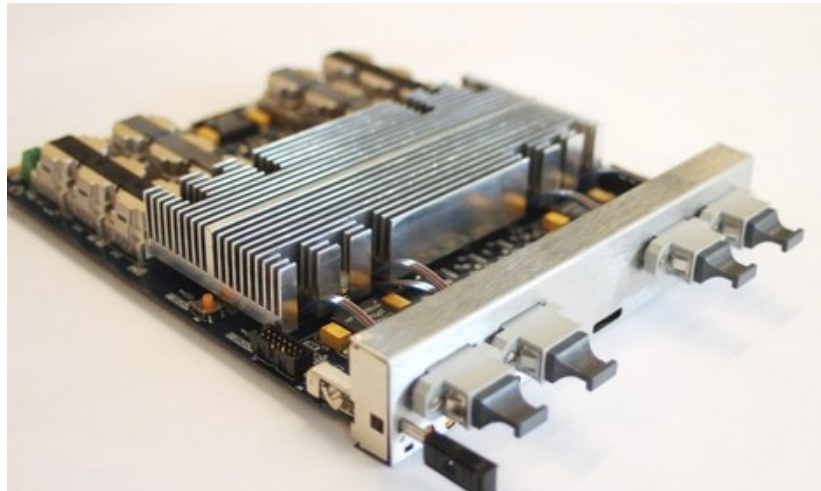
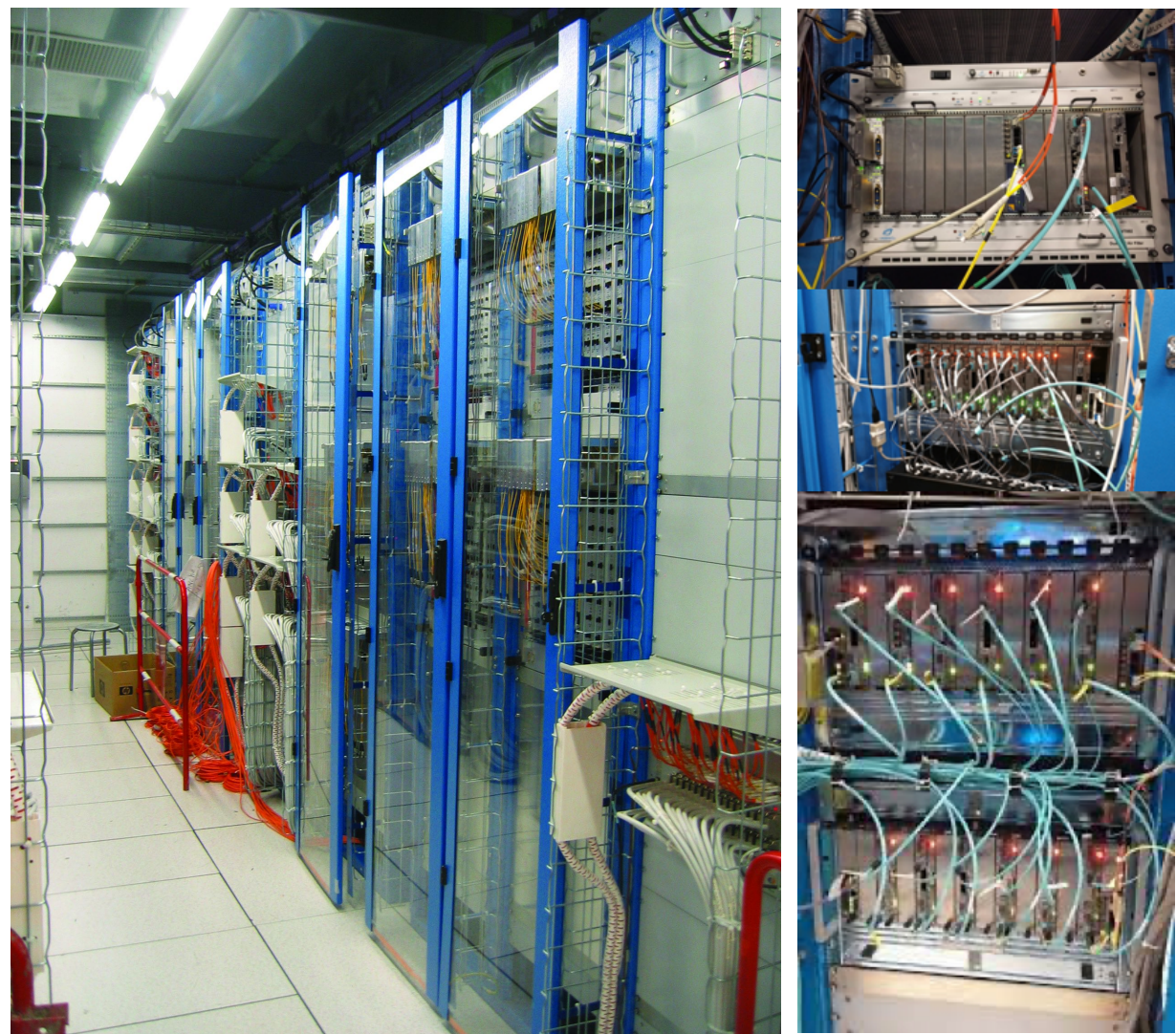
DSPs - for multiplication

BRAM - for local, high-throughput storage

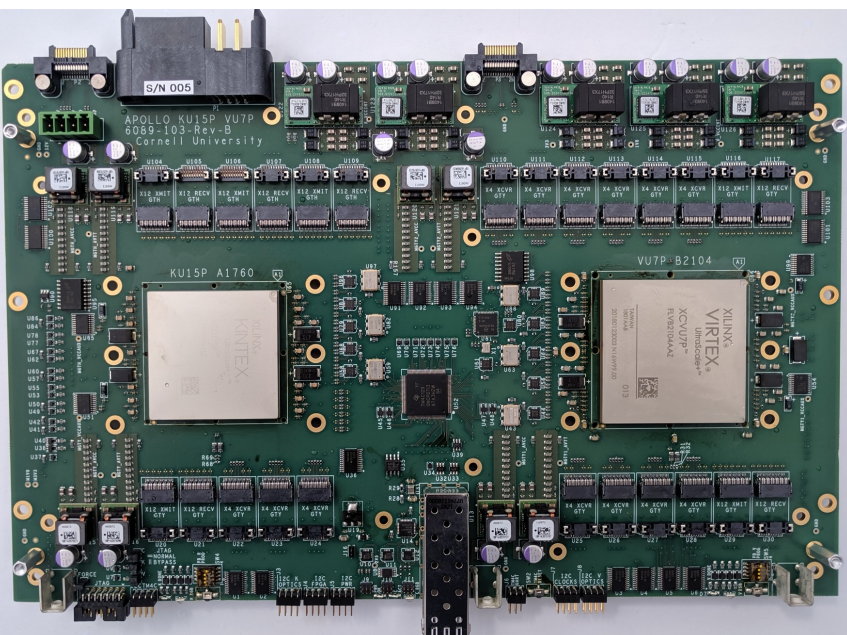
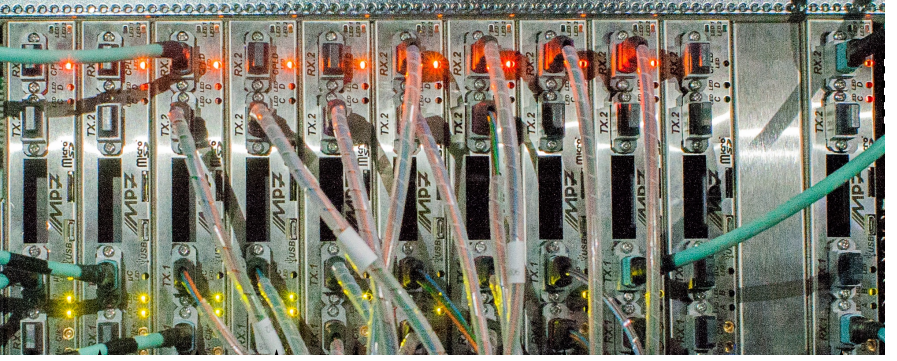
# FPGAs at the LHC

Most commonly: Stream processor / real-time : data acquisition, trigger, control

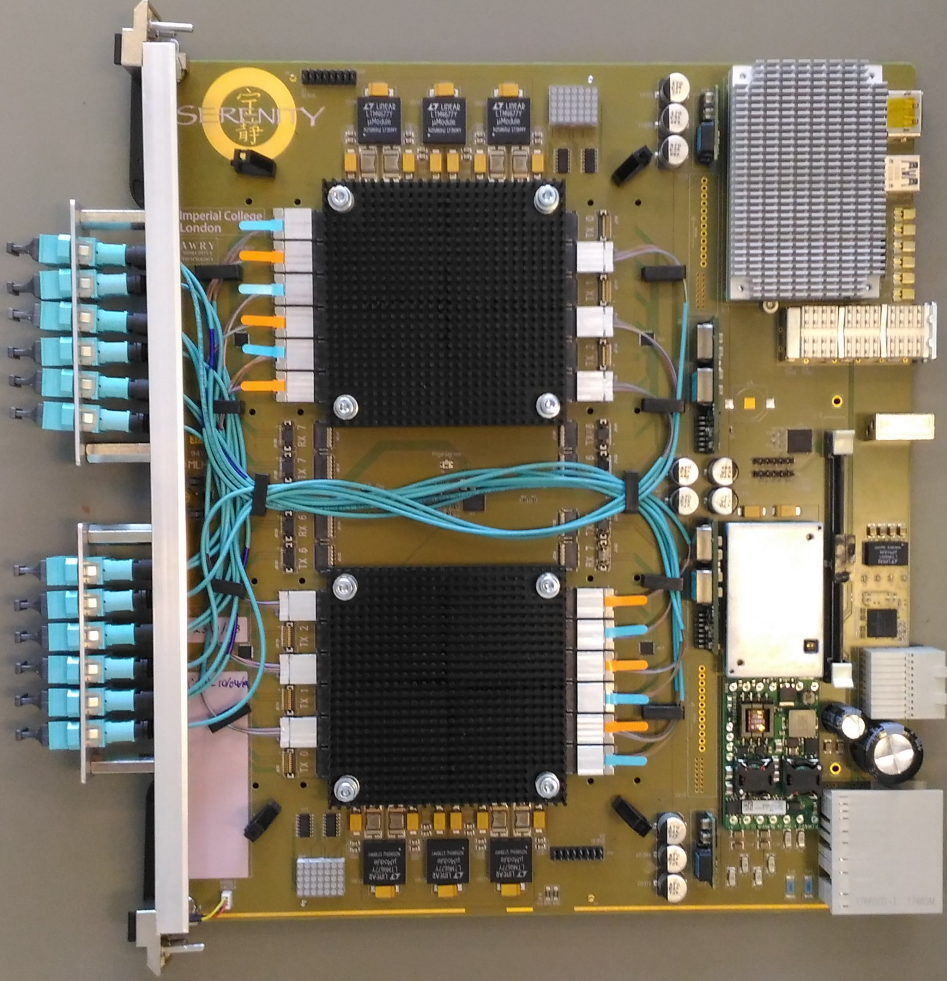
- Primarily use **custom hardware**
- Very high IO bandwidth / optical inputs
- Often custom board for each task



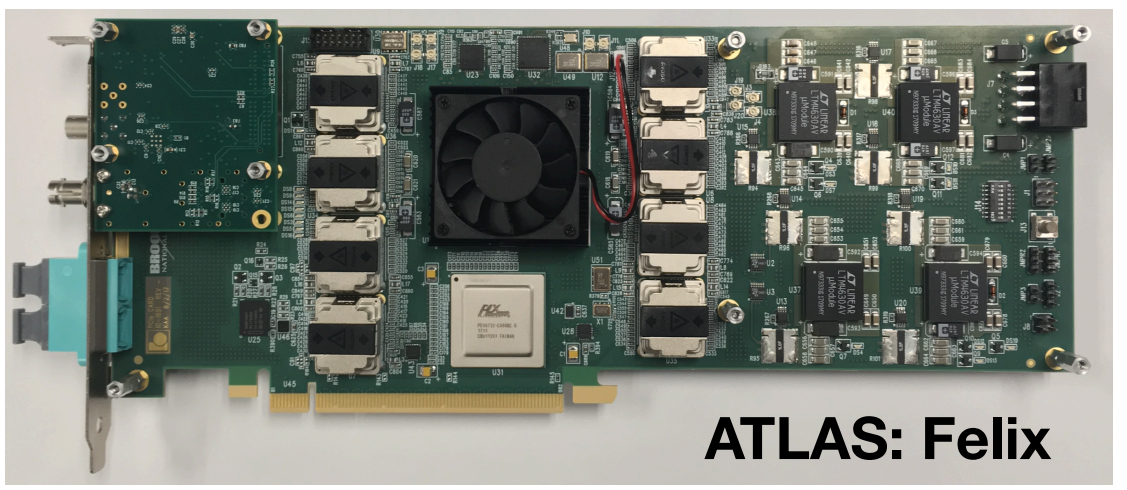
CMS/UK: MP7



CMS/US: APOLLO



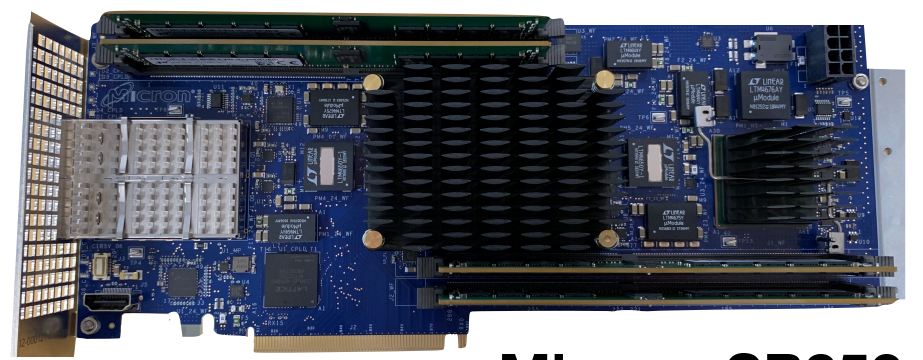
CMS/UK: Serenity



ATLAS: Felix

## Accelerators

- More often commercial hardware
- Mostly PCIe form-factor



Micron: SB852

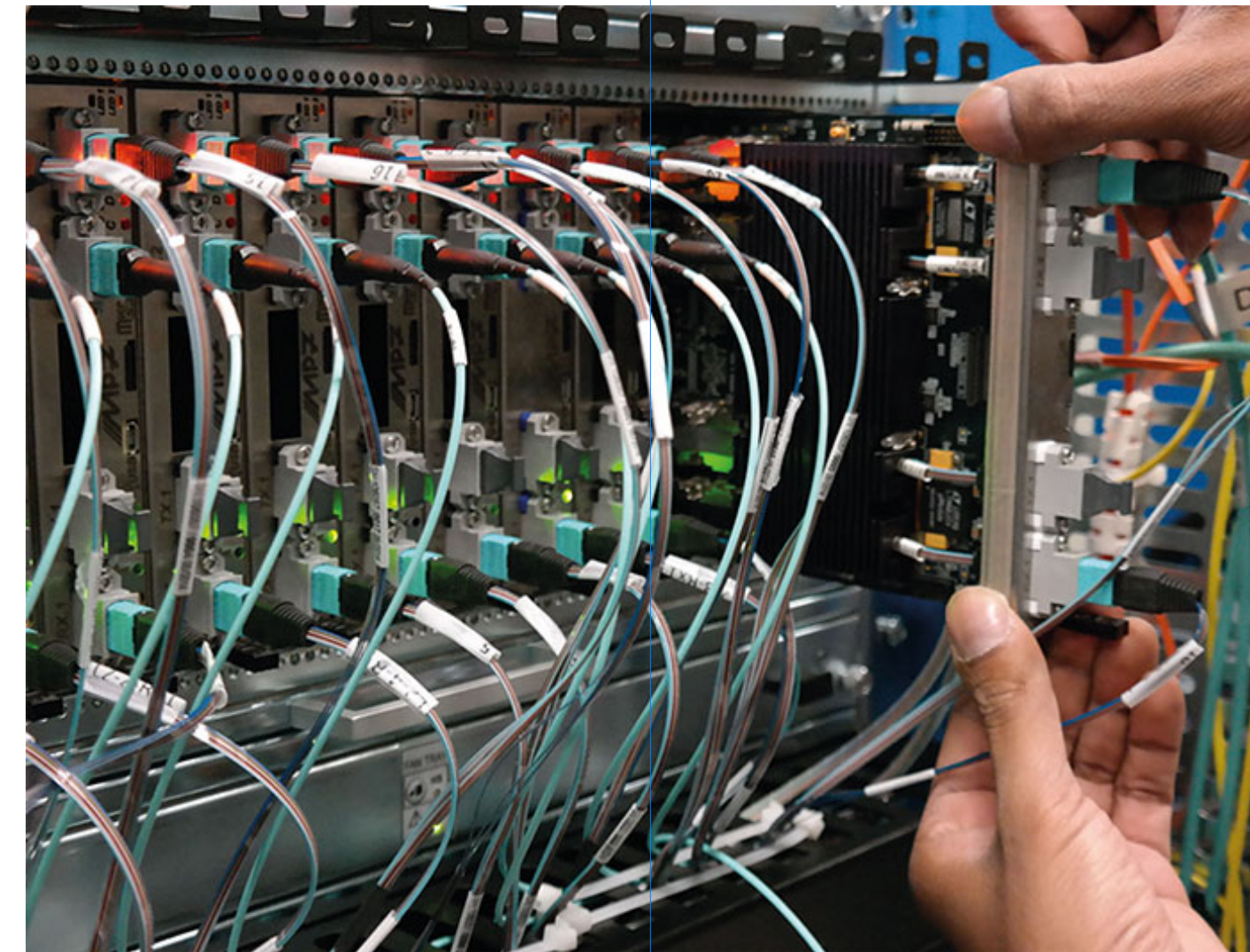


AMD: Alveo U250

Form factor: VME -> MTCA -> ATCA / PCIx -> PCIe

# Algorithms running on FPGAs

- LHC Run 2 (2015-2018)
  - Clustering
  - Pattern Recognition
  - Energy Sums
  - Zero Suppression
  - Boosted Decision Trees
- LHC Run 3 (2022-2025)
  - Multi Layer Perceptrons: DNNs
  - Kalman Filters

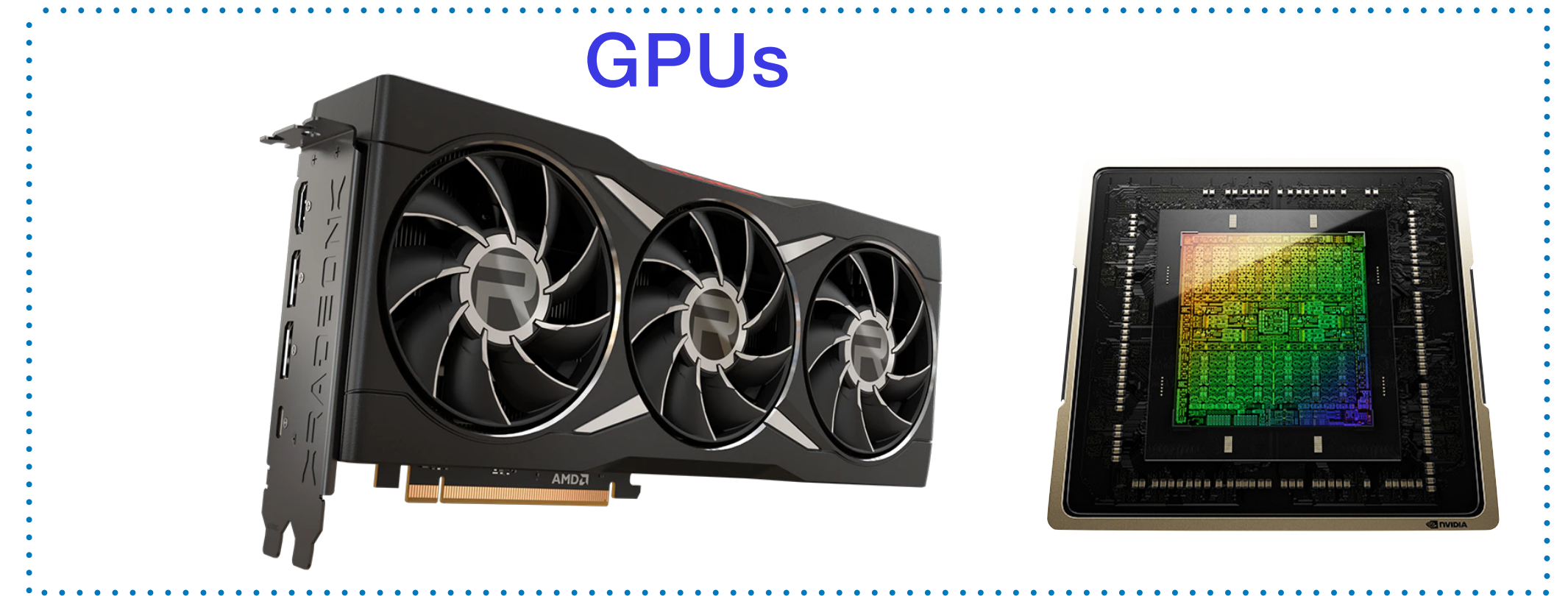


- LHC Run 4 dev. 2029-
  - Hough Transform
  - Convolutional Neural Networks
  - ??????

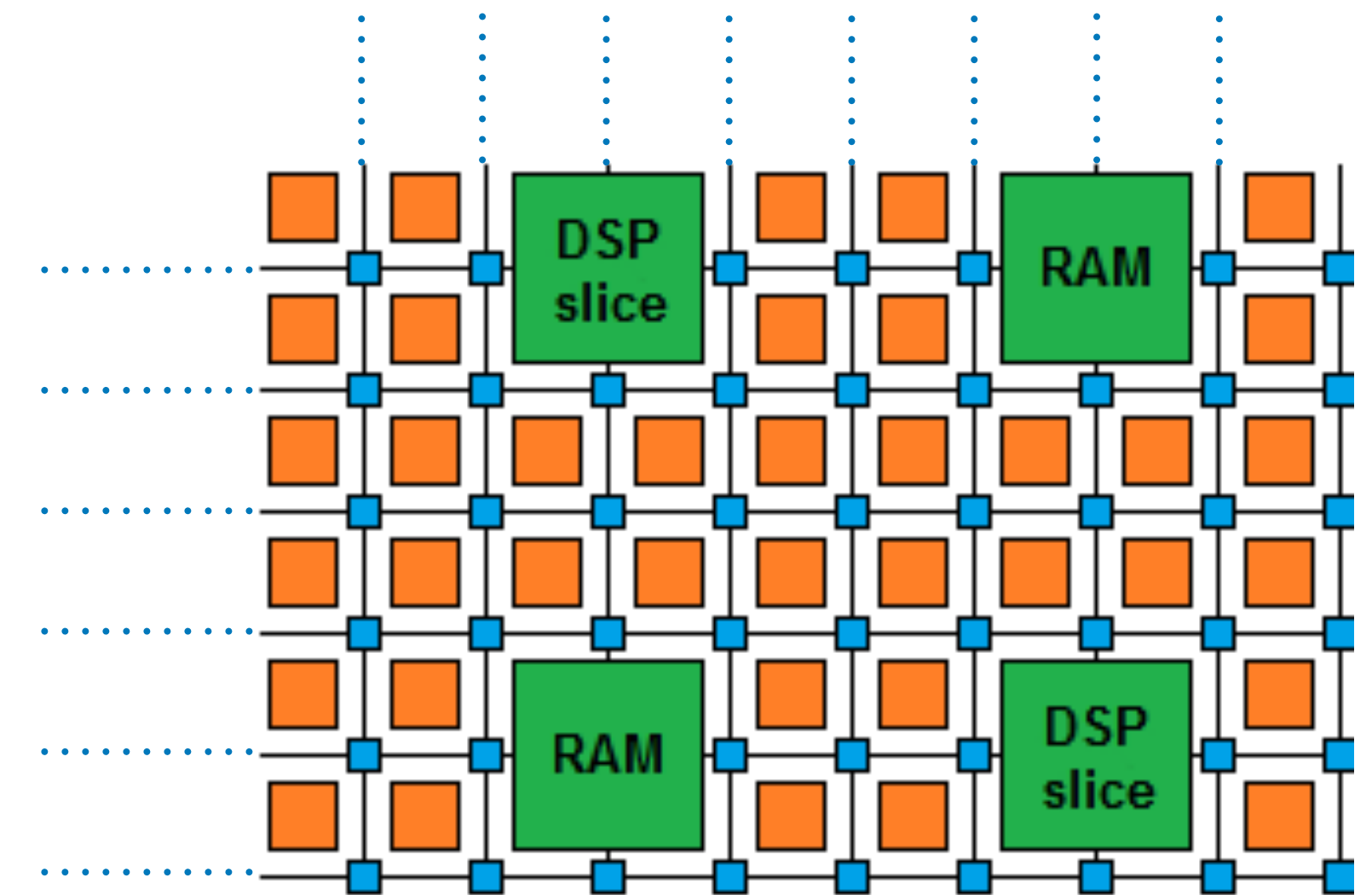


# FPGAs for ML

- **GPUs** are very powerful for machine learning
  - Many more parallel arithmetic ops than a CPU
  - Very high memory bandwidth
- However **FPGAs are also highly suited to ML tasks**
  - massive **parallelism**, high **memory bandwidth**
  - Outperform GPUs at maintaining high-throughput & **low latency** with (often) best 'performance per Watt'
  - **Deterministic** latency - requirement for low-level trigger
    - Not possible with GPUs/CPUs



FPGA diagram



LUTs - generic logic  
DSPs - for multiplication  
BRAM - for local, high-throughput storage



# High Level Synthesis

- FPGA programming requires expert engineering knowledge, long development cycles - you are describing a circuit
- Newer design tools from the FPGA companies - **HLS**
- You describe algorithm, compiler decides circuit
  - Enables more physicists to contribute & accelerates development timelines
  - Allows us to bring more of the offline algorithms into the L1 trigger
  - e.g Kalman Filter, Particle Flow, etc
  - **Machine Learning...**

## VHDL

```
entity add is
port(
  clk : in  std_logic;
  a   : in  signed(31 downto 0);
  b   : in  signed(31 downto 0);
  c   : out signed(31 downto 0)
)
end add;

architecture rtl of add is
  if rising_edge(clk) then
    c <= a + b;
  end if;
end rtl;
```

## HLS

```
int add (int a, int b){
  return a + b;
}
```

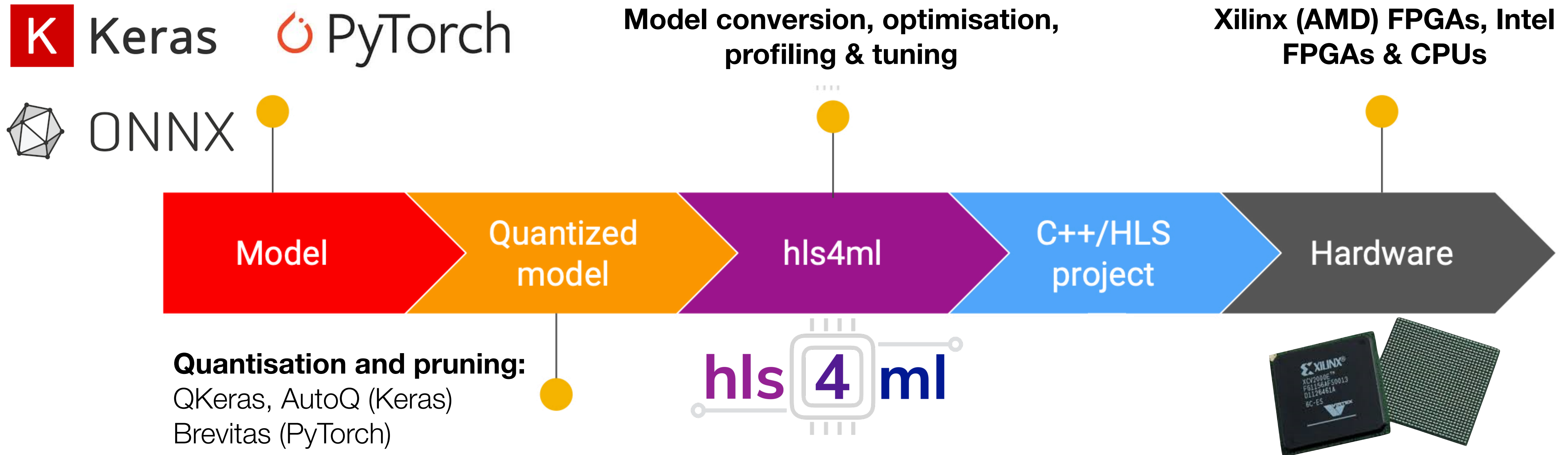
*Still need to be mindful of design & use #pragma*

```
//Use registers
#pragma HLS array_partition variable=a,b,c complete
//Execute loop iterations in parallel
#pragma HLS unroll
```

# HLS4ML

- Open-source Python API & command line tool that translates trained NNs to synthesizable FPGA firmware

<https://fastmachinelearning.org/hls4ml>

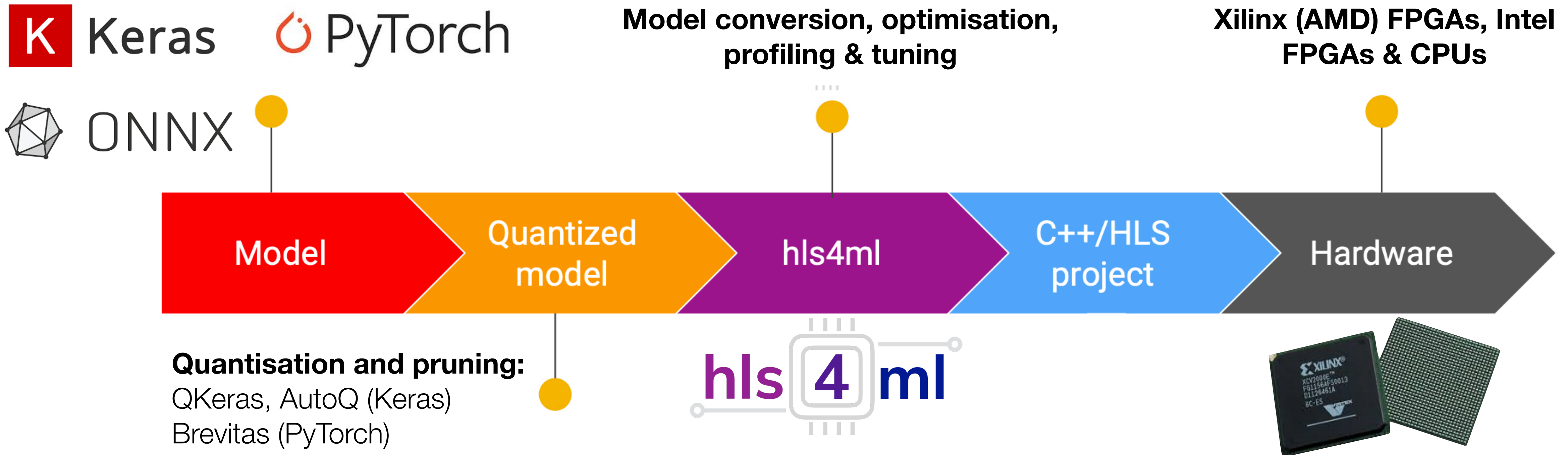


- Can tune latency vs resource utilisation with per-layer ‘reuse factor’
- Weights stored on-chip -> very fast access times, limited capacity
- Excels at very low latency applications: planned to be widely used at High-Lumi LHC

# HLS4ML

- Open-source Python API & command line tool that translates trained NNs to synthesizable FPGA firmware

<https://fastmachinelearning.org/hls4ml>



- Implementations of common ingredients - layer types, activation functions
- Novel ingredients for fast, efficient inference - binary/ternary NNs, heterogeneous quantisation

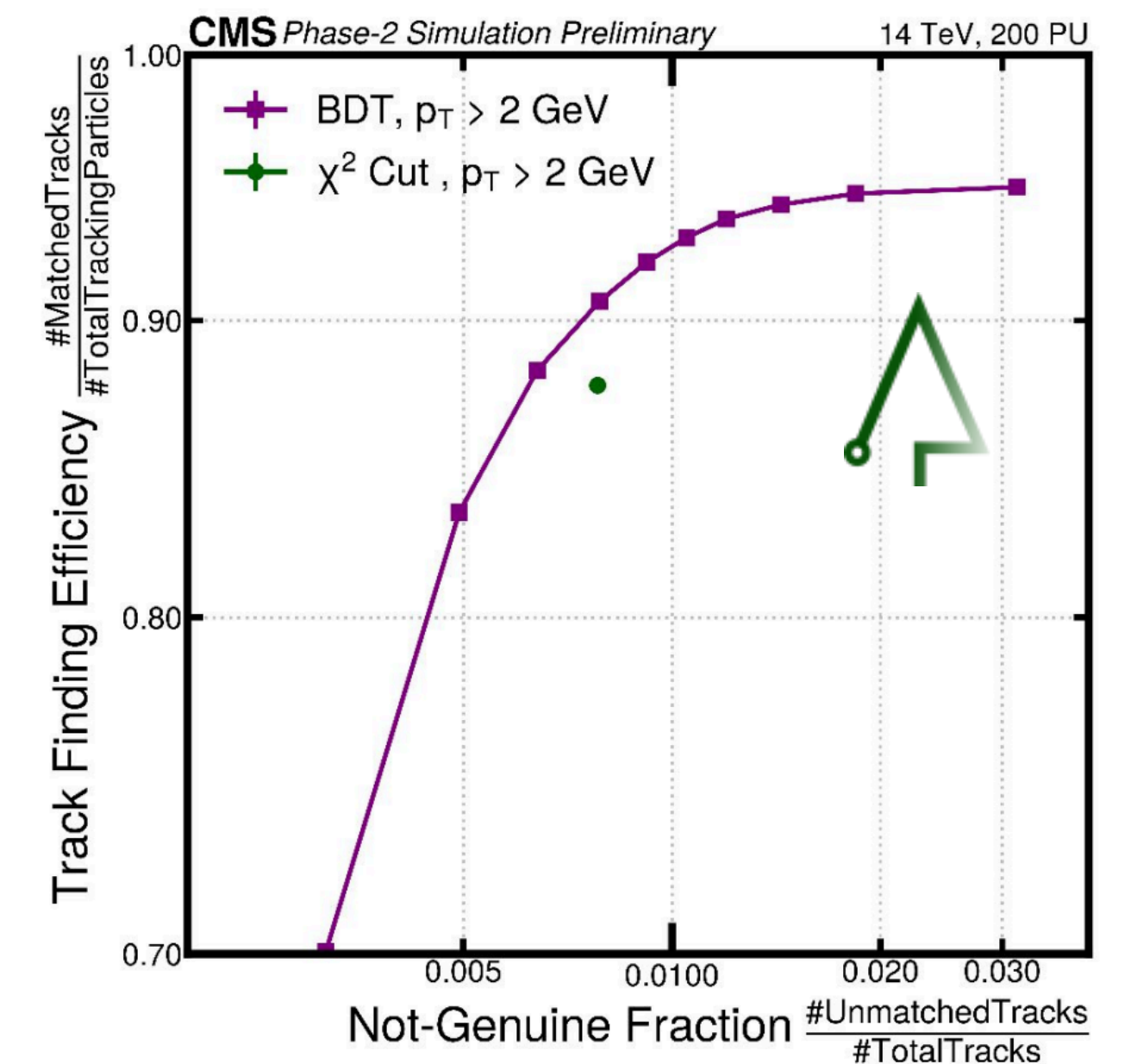
# HLS4ML

<https://fastmachinelearning.org/hls4ml>

- Caveats:
  - Relies on Xilinx HLS (tool that produces FPGA code from C++), blackbox that can produce non-optimal results
  - Requires a bit more knowledge of FPGA design than some other solutions, but still accessible to non-Verilog/VHDL experts
  - Work on support for new backends & off-chip weights ongoing
- Ideal for L1-trigger applications: expected to be widely used for CMS Phase II trigger
- Many algorithms in development for CMS at the HL-LHC
  - Improving object reconstruction
  - Improving event selection of difficult signatures

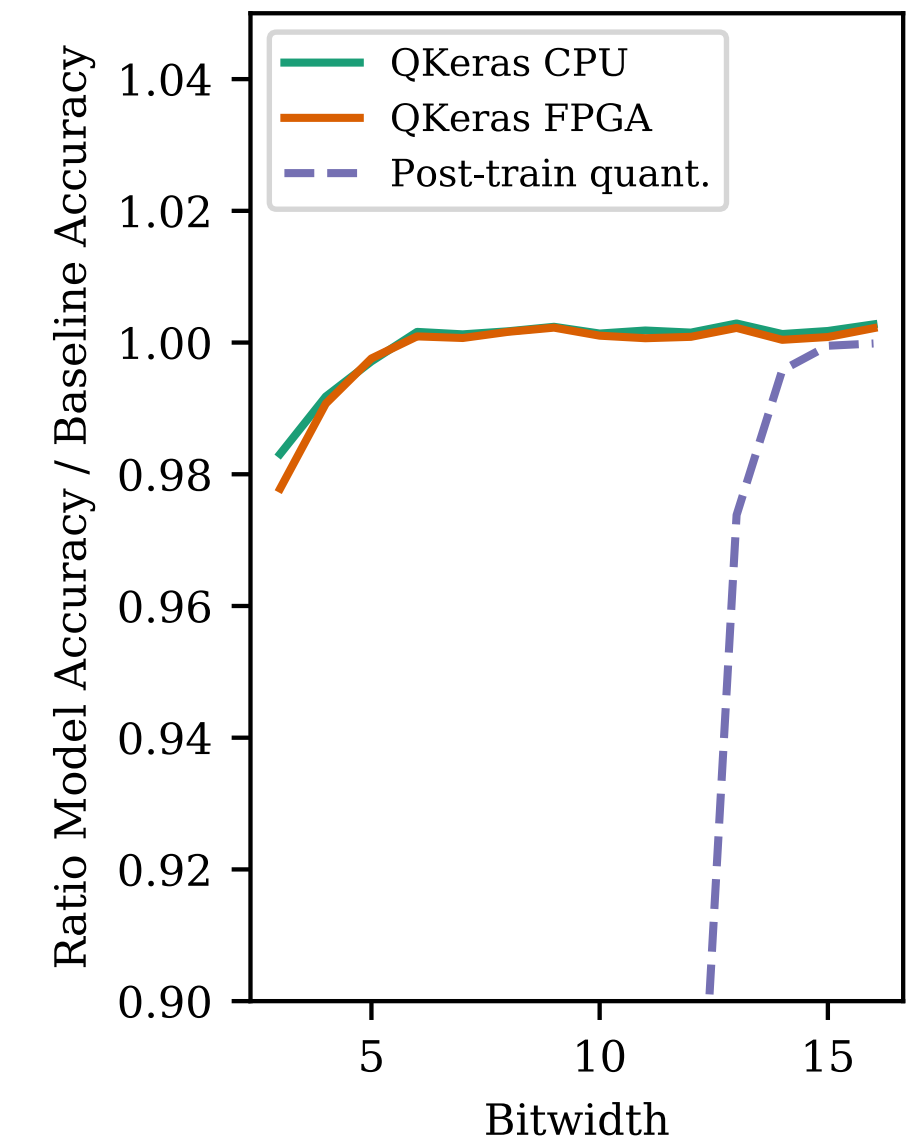
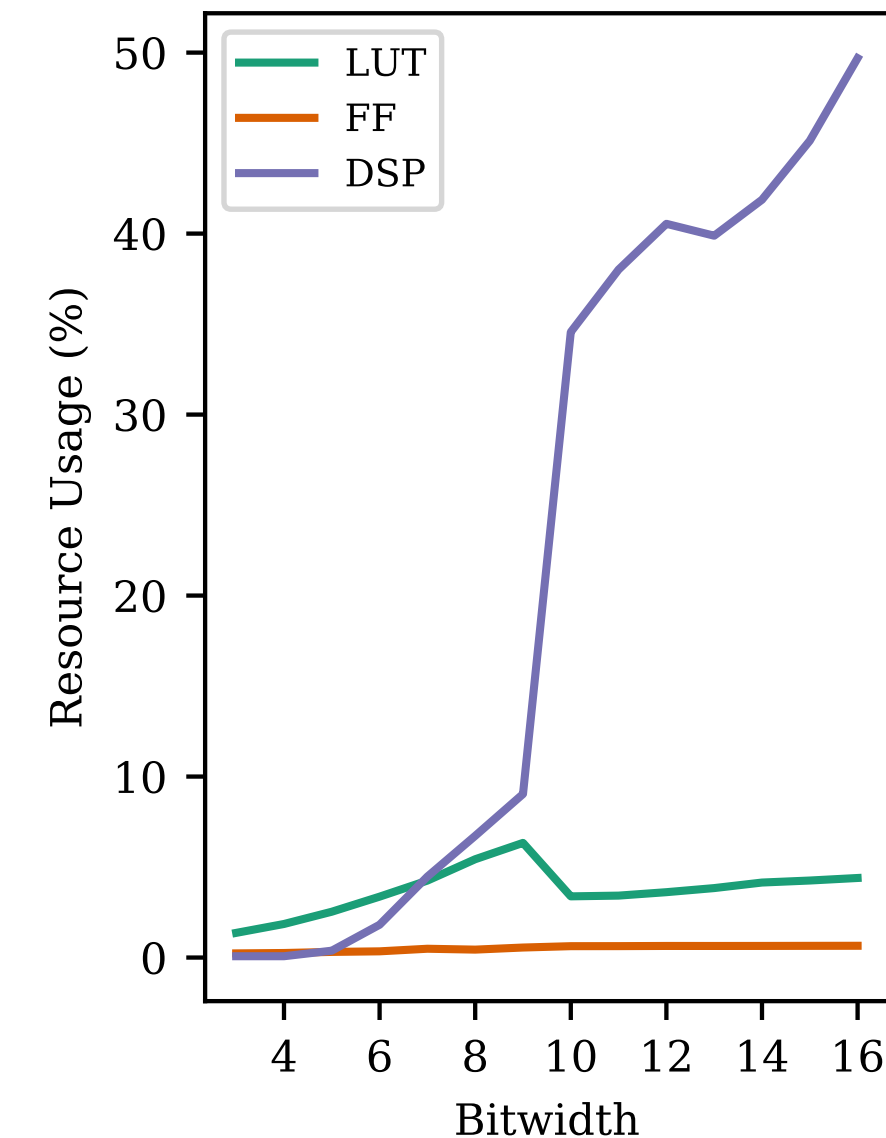
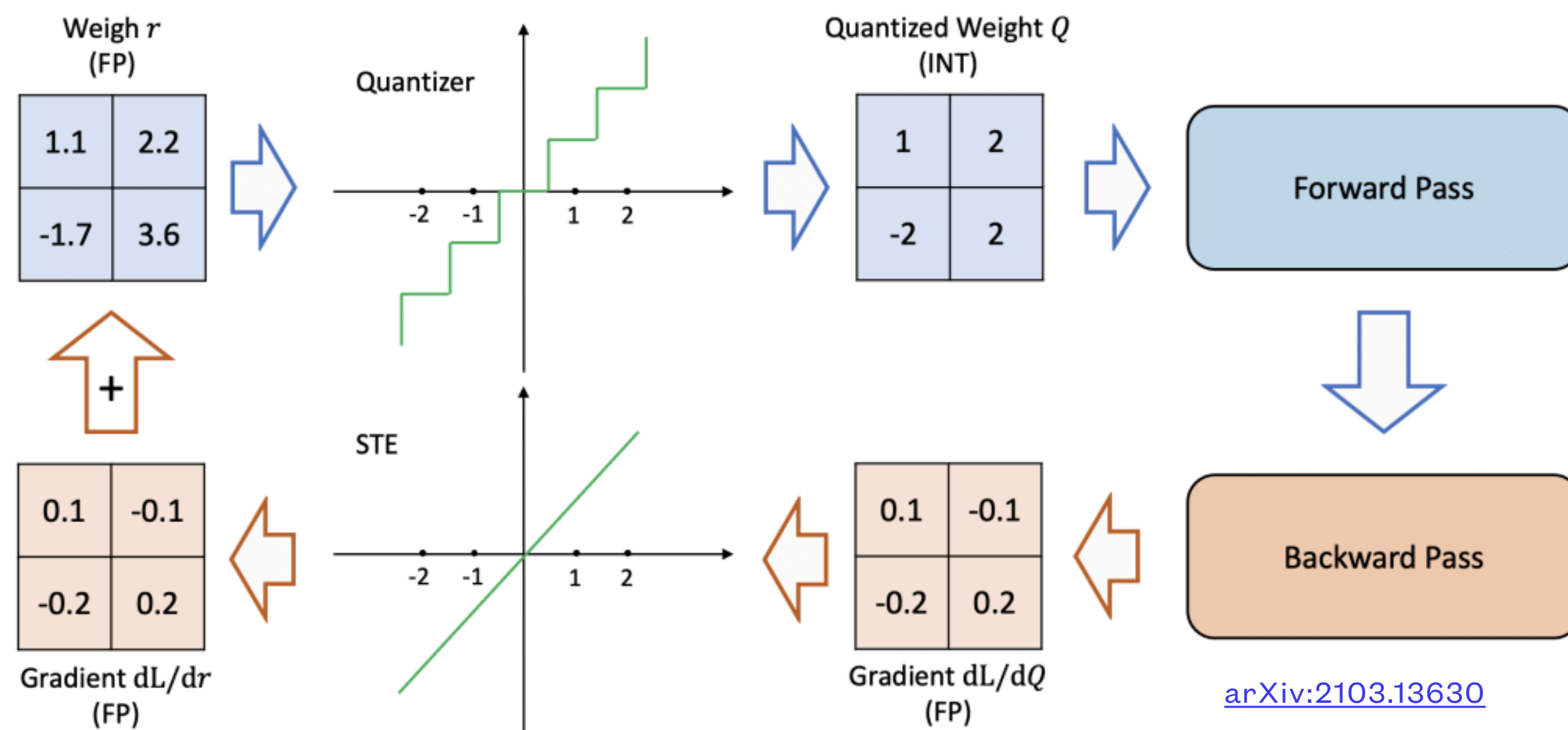
# Conifer

- Tools like hls4ml and conifer bring ML into FPGAs with sub-microsecond latency
- Conifer library maps BDT onto FPGA logic
- Example: identifying fake tracks from CMS Level 1 Track Finder (Phase 2 Upgrade)
- Fake tracks are identified in simulation as those not associated to a simulated particle
  - Often from combinatorics (200 pileup scenario), they harm trigger performance later
- A BDT with 60 trees and depth of 3 finds fakes better than simple cuts
- In this case 33 ns latency and < 1% resources of a VU9P



# Quantization

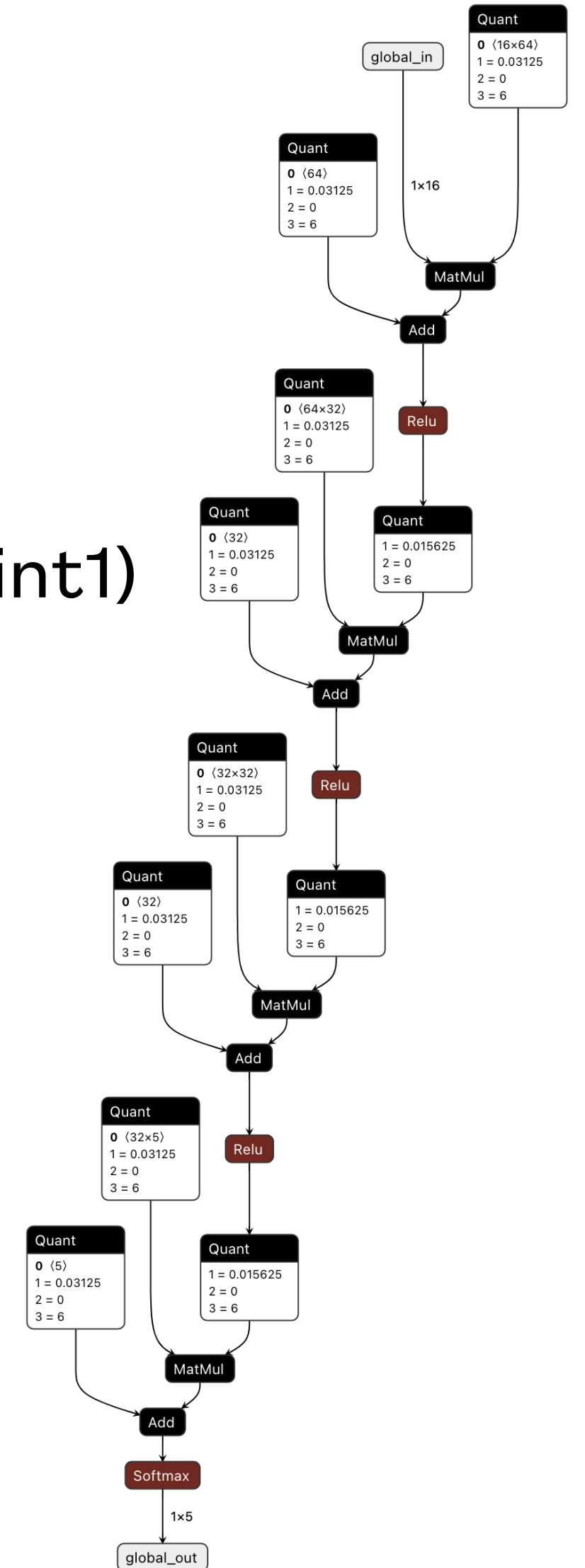
- Like to avoid floating point in FPGAs - much more resources & latency than fixed point
- *Post-training quantisation* - represent the float values with some fixed point
- Better to use **quantisation aware training (QAT)**



- **Heterogeneous quantisation** useful:
    - Regression -> higher precision output layers
    - Auto encoder -> higher precision bottleneck layers
  - When DSPs limiting resource can try binary / ternary quantisation
- AutoQ tool for training NNs with hardware-cost constraints [6]

# Representing Quantized NNs

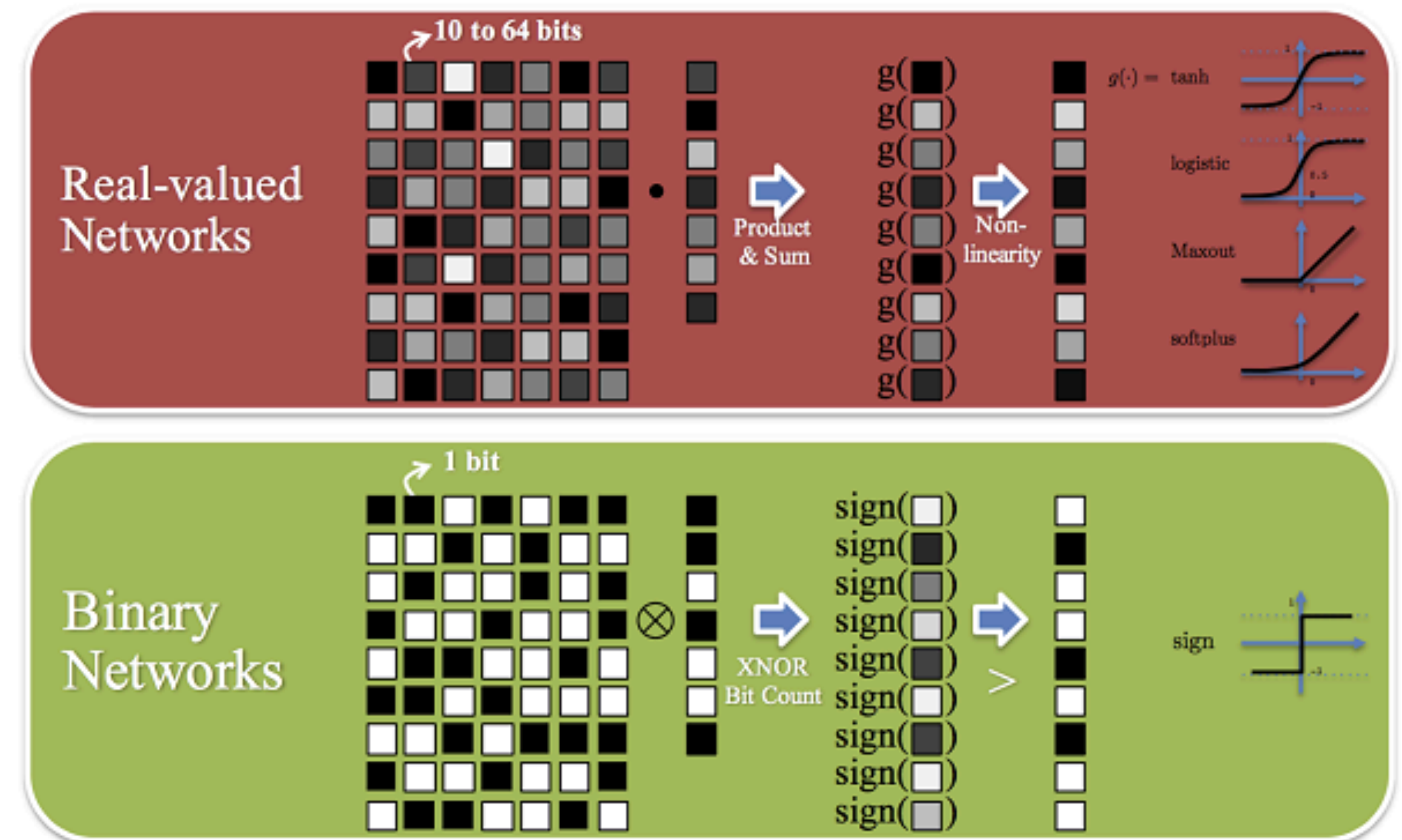
- Lots of tools like Tensorflow, PyTorch, TensorRT have support for low precision (including QAT)
- But they are typically restricted to common CPU/GPU types (float16, int8, int4, int1)
  - For dataflow (layer unrolled) FPGA inference, we would like more flexibility
- Collab w/ Xilinx Research Labs: HLS4ML team develop QONNX [7]
- Extend QONNX with `Quant` node
  - Flexible number of bits, zero-point, and per-channel scale factors
  - onnxruntime execution thanks to FINN (Xilinx RL NNs)
  - QONNX is exported by Brevitas, others are working on it, and we develop a QKeras to QONNX conversion
- [github.com/fastmachinelearning/qonnx](https://github.com/fastmachinelearning/qonnx)





# Binary / Ternary neural networks

- DSP multipliers often limiting resource
- Can often go down to 1- or 2-bit weights with limited performance loss
- Can have very efficient computation in the FPGA (and CPU/GPU/smartphone)
- Binarize weights **but not gradients** during backpropagation
- Use Binary Tanh, Ternary Tanh or ReLU activation
- BNN: [arxiv.1602.02830](https://arxiv.org/abs/1602.02830)
- TNN: [arxiv.1605.04711](https://arxiv.org/abs/1605.04711)

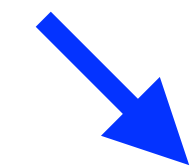


[intel.com](https://intel.com) [4]

# BNN - Dense Layer

- DSPs often limiting FPGA resource for NNs
- Encode '-1' as '0'
- Multiplication become XNOR, sum becomes bitcount

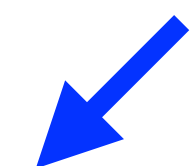
A	B	A*B
-1	-1	1
-1	1	-1
1	-1	-1
1	1	1



A	A'
-1	0

-1 0

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



1	1
---	---

Original: 16-bit weights

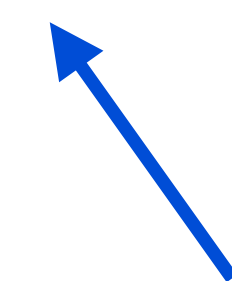
$$X_n = g_n(W_{n,n-1}x_{n-1} + b_n)$$



Activation function:  
precomputed,  
stored in BRAMs



Multiplication:  
DSPs



Bias addition:  
LUTs/FFs

Binarized: 1-bit weights

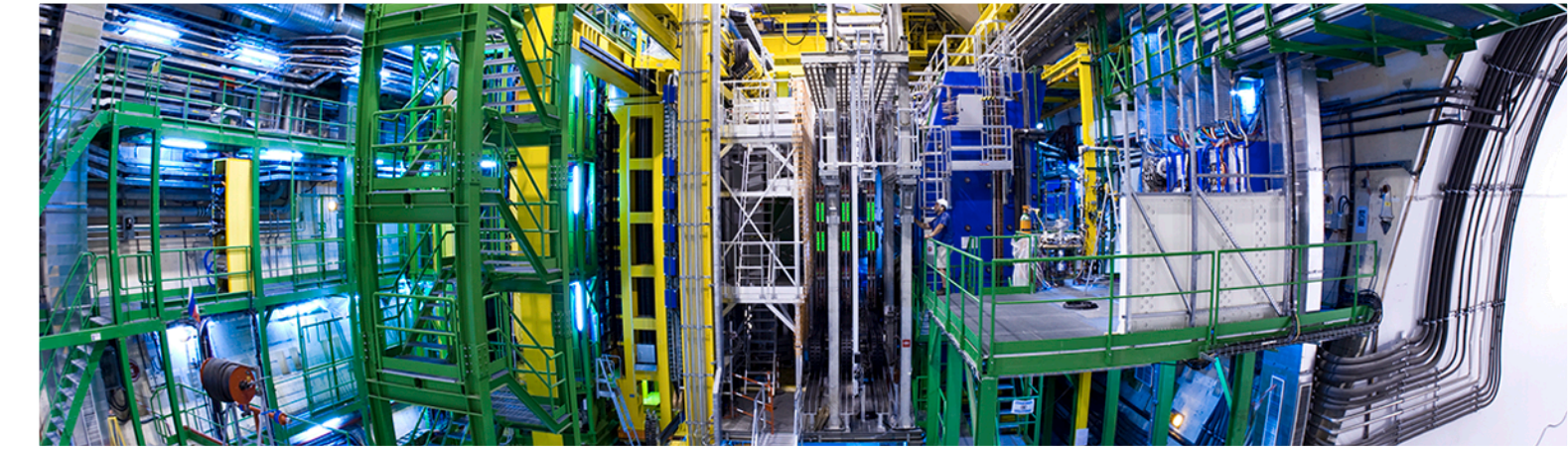
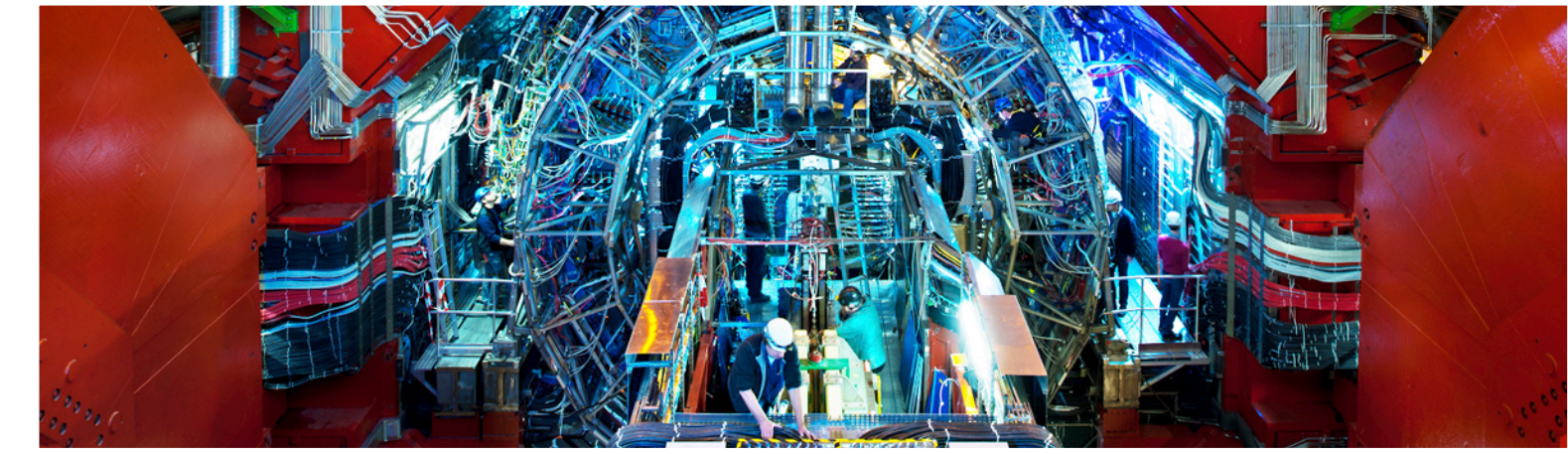
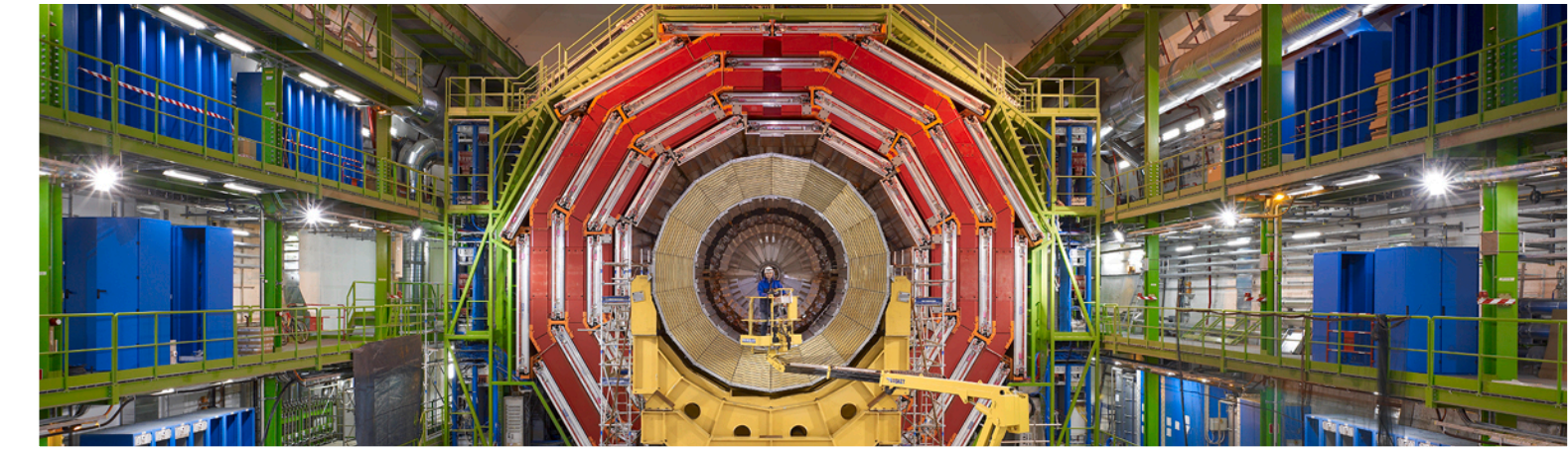
$$X_n = g_n(W_{n,n-1}x_{n-1})$$



Activation function:  
simple binary  
function



XNOR:  
LUTs/FFs



---

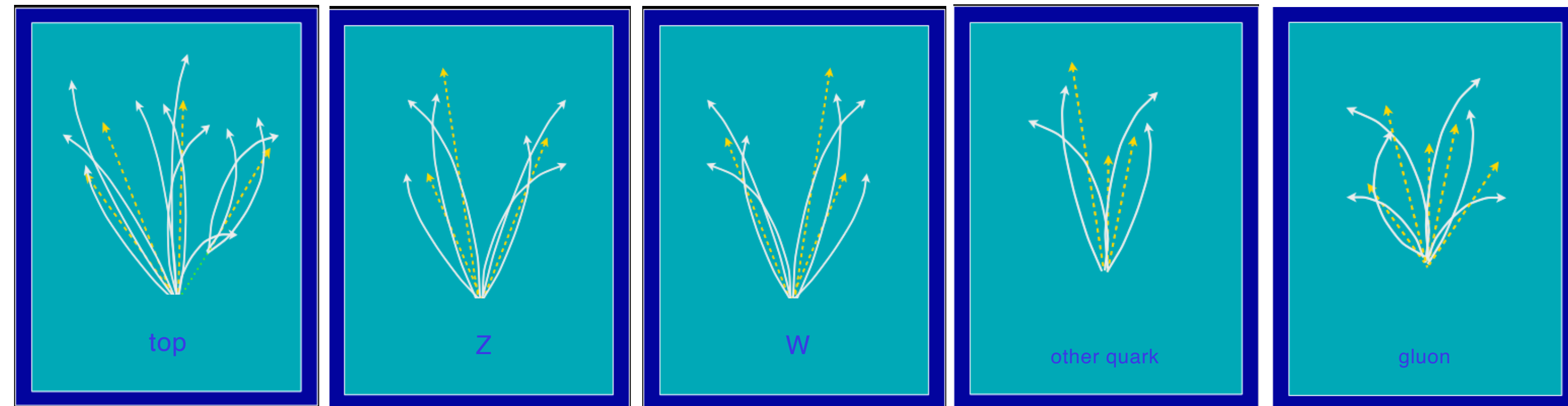
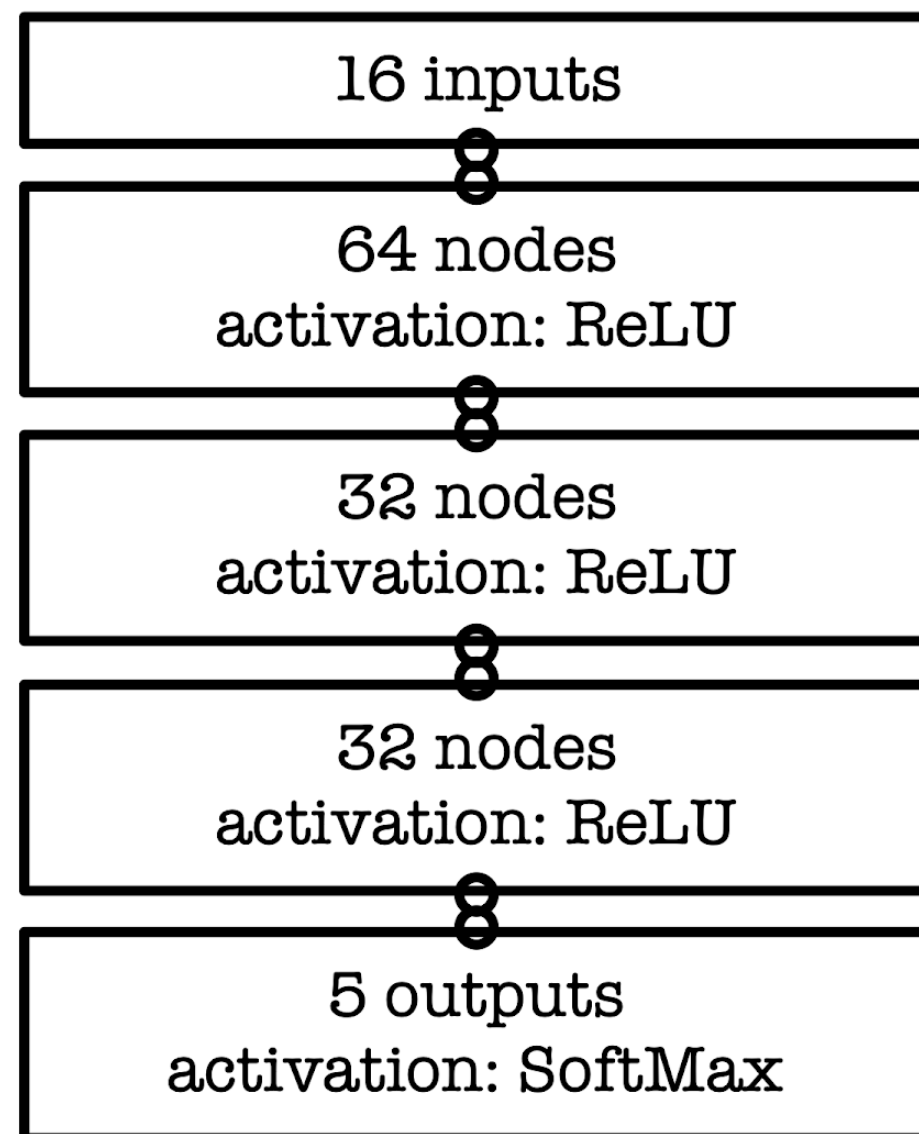
# ***Fast ML in action (i.e examples<sup>†</sup>)***

---

<sup>†</sup> absolutely non-exhaustive list

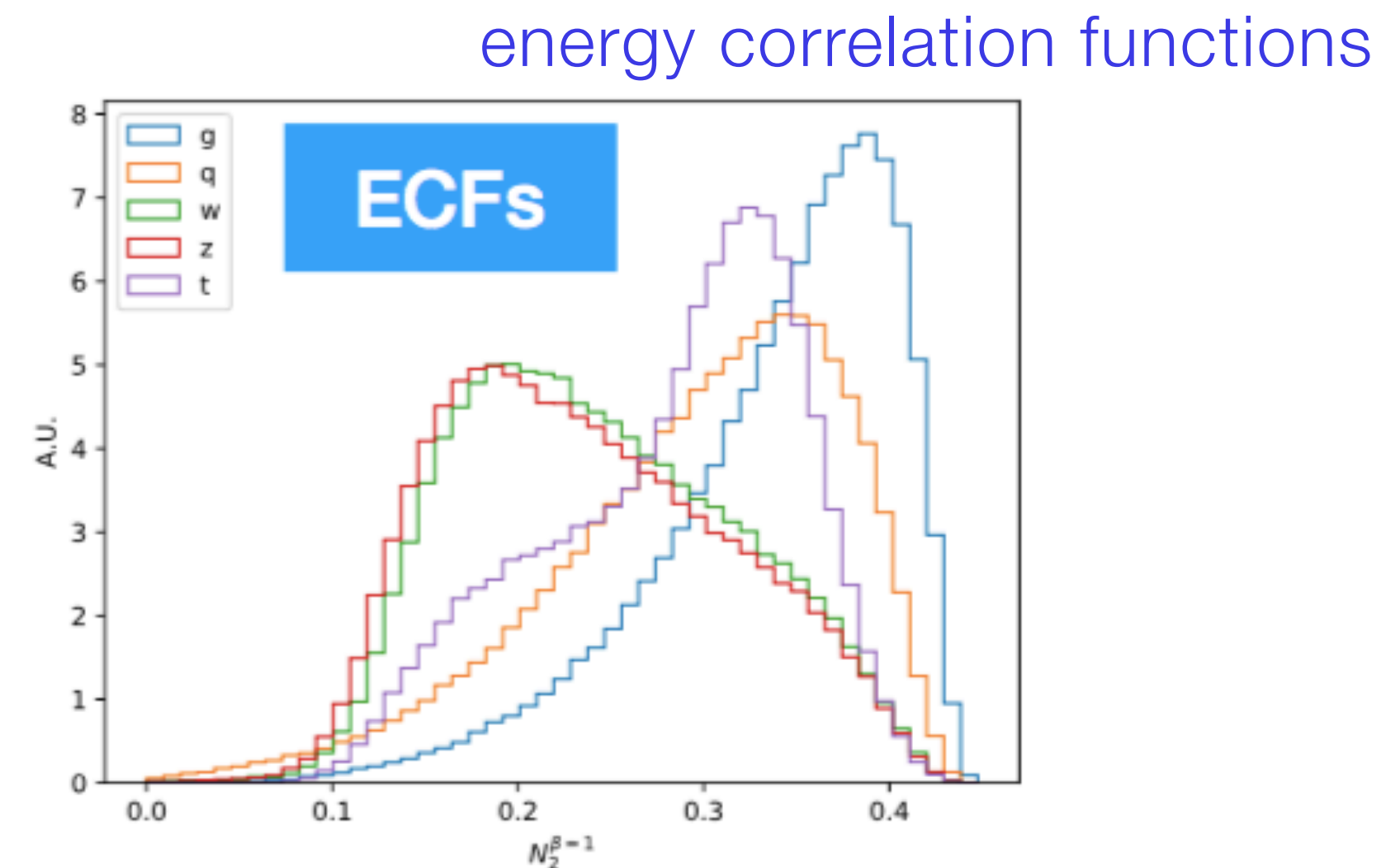
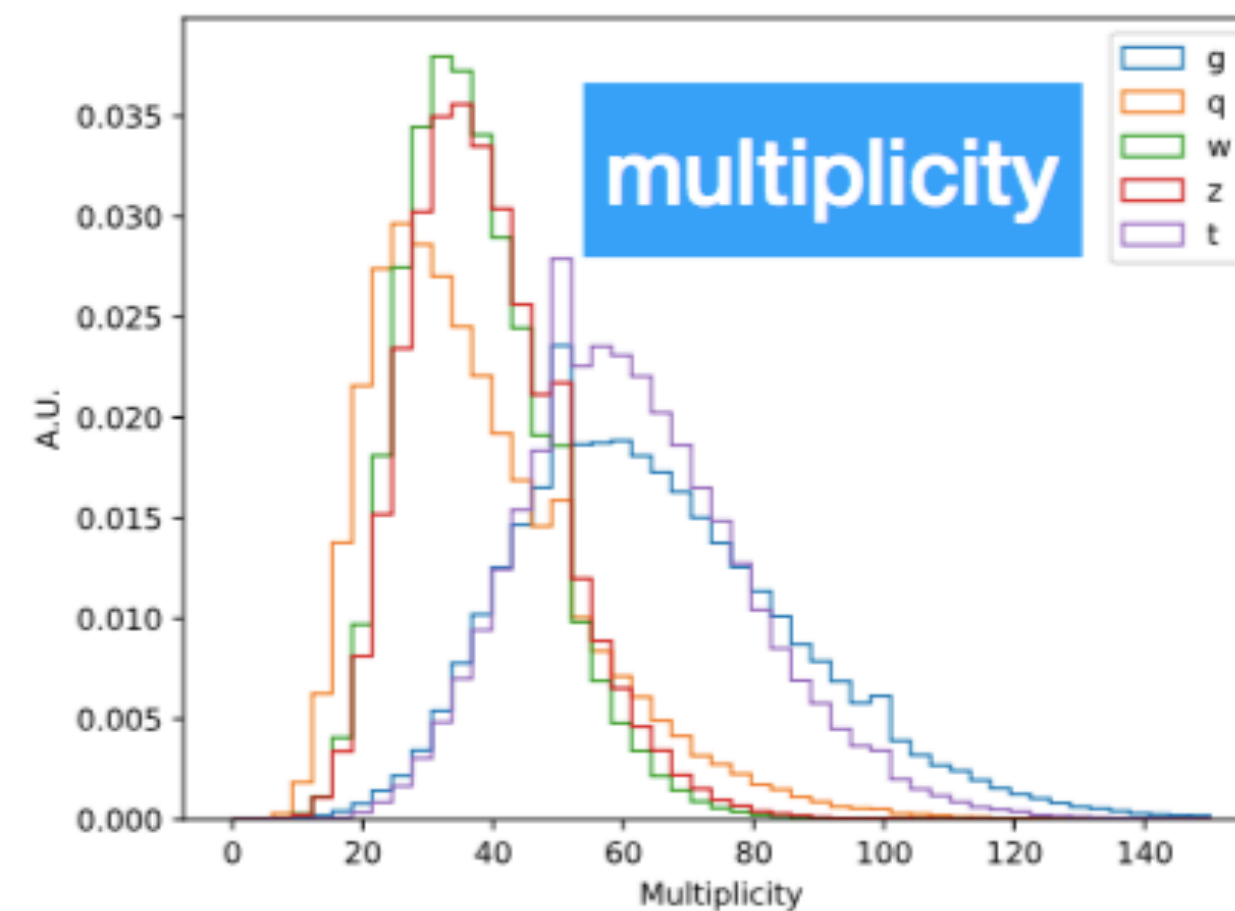
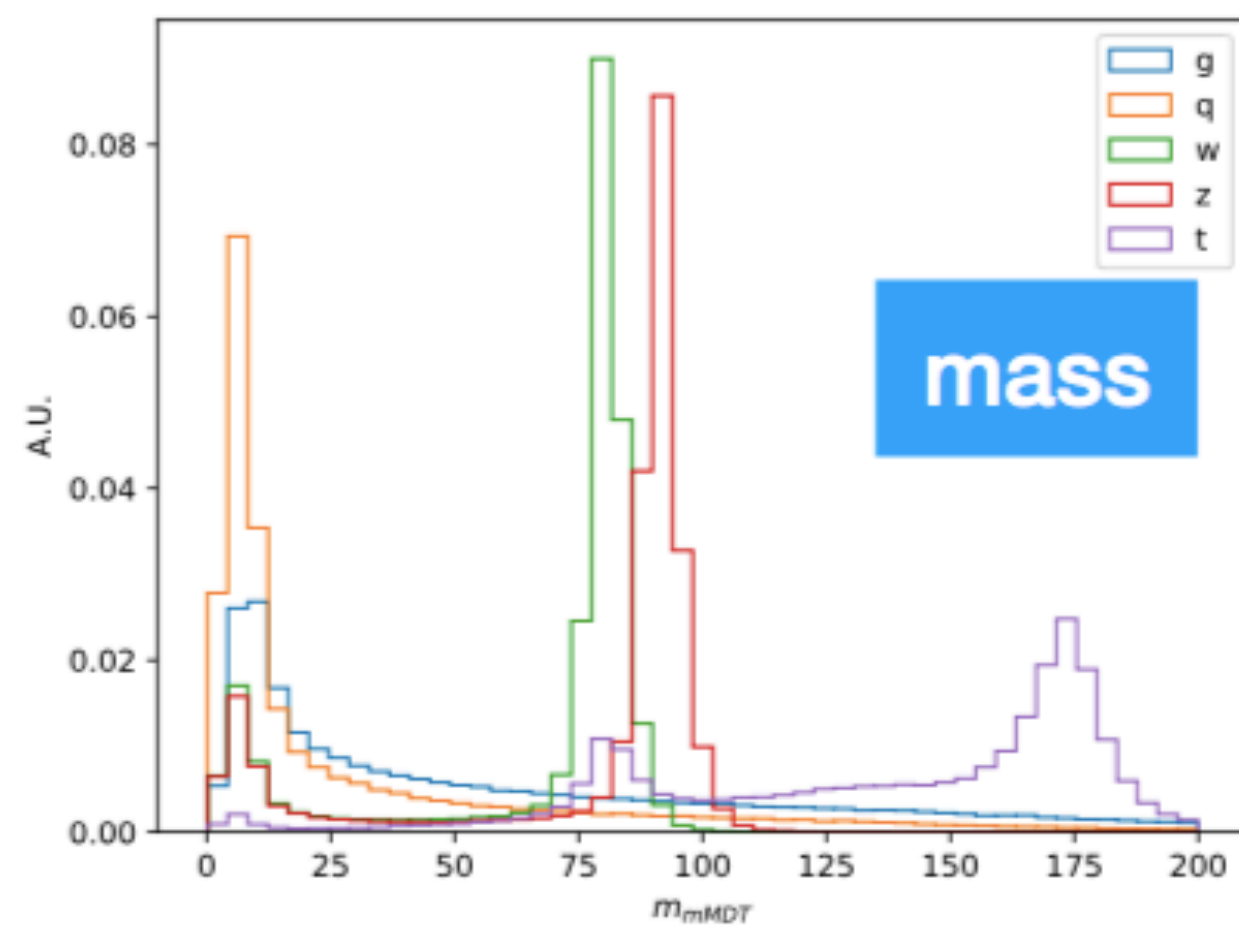
# Jet tagging

- HLS4ml tutorial example [2]
  - Tagging jets (5 classes, 16 input variables)
- 3 fully connected layers

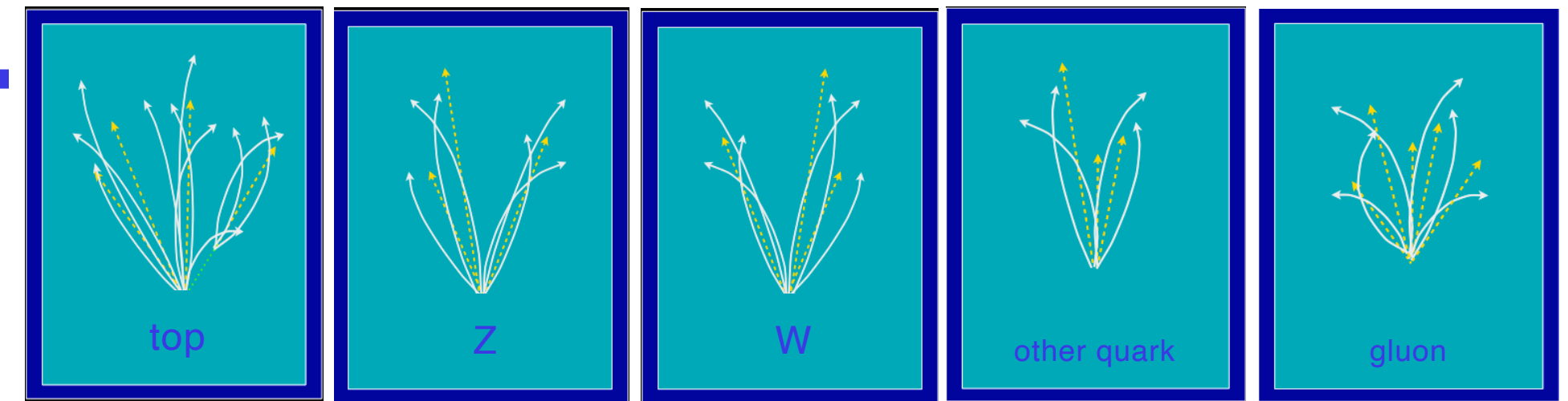


# Jet tagging

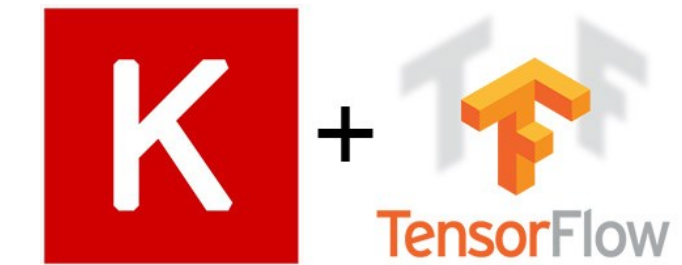
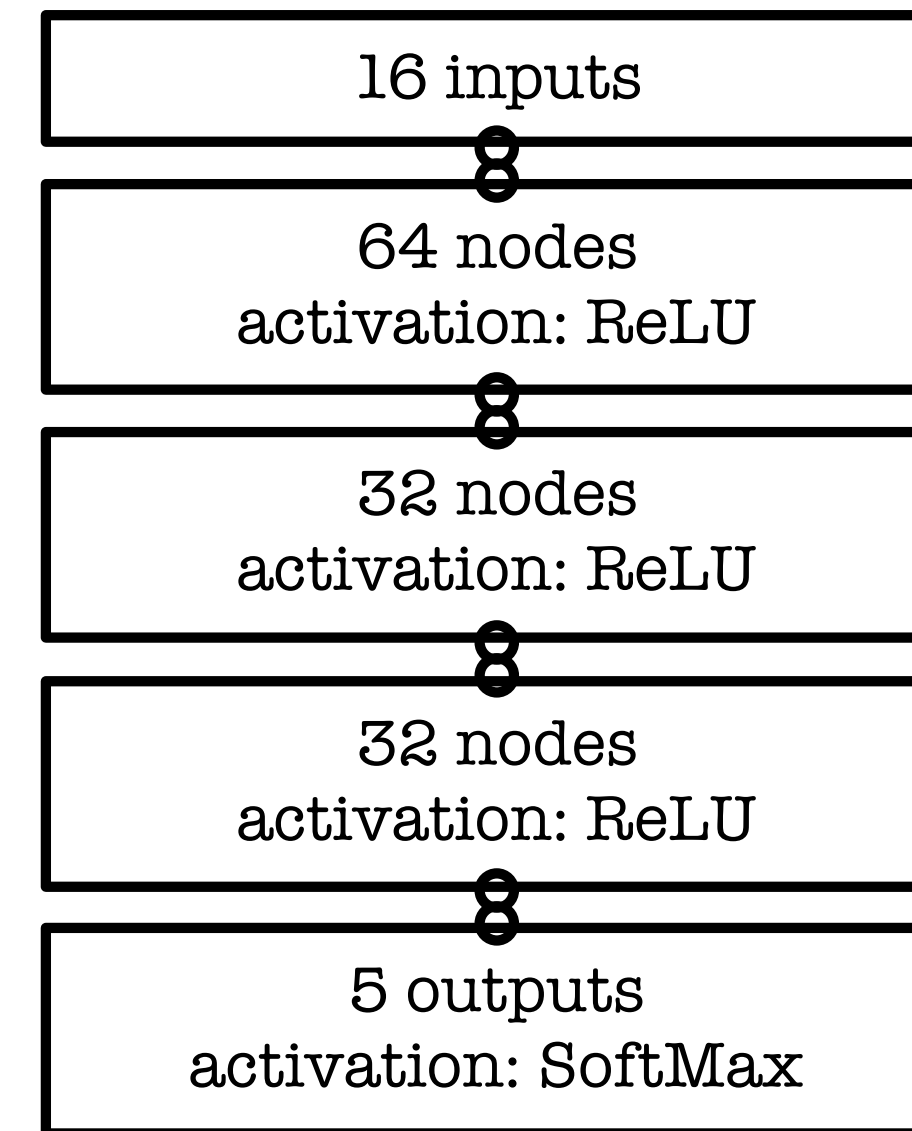
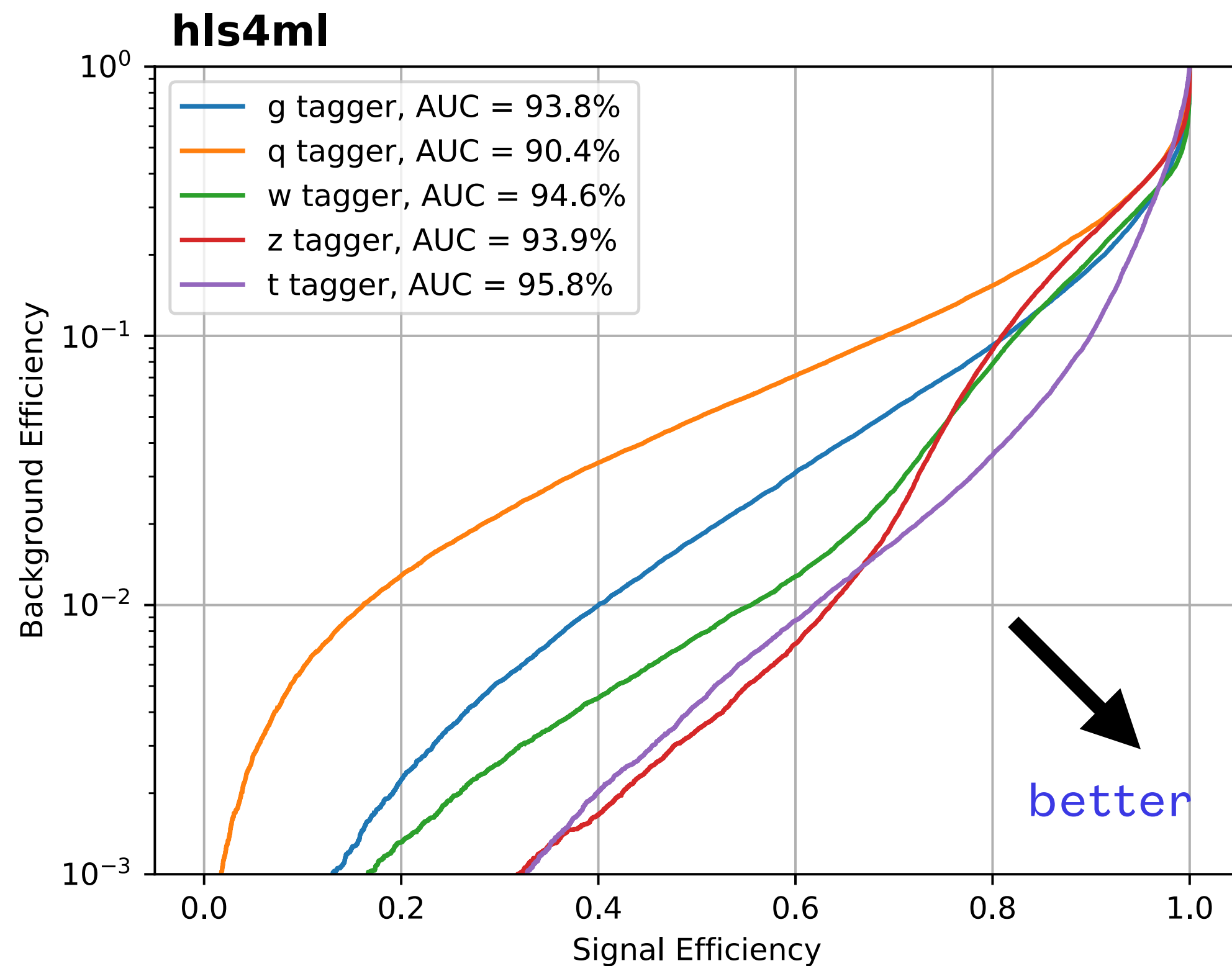
- Trained (on GPU) the five output multi-classifier on a sample of  $\sim 1\text{M}$  events with two boosted  $WW/ZZ/tt/qq/gg$  anti-kT jets
- 16 expert-level input variables, computed with FastJet:
  - known to have high discrimination power from offline data analyses and published studies



# Jet tagging



- Fully connected neural network with **16 expert-level inputs**:
  - Relu activation function for intermediate layers
  - Softmax activation function for output layer



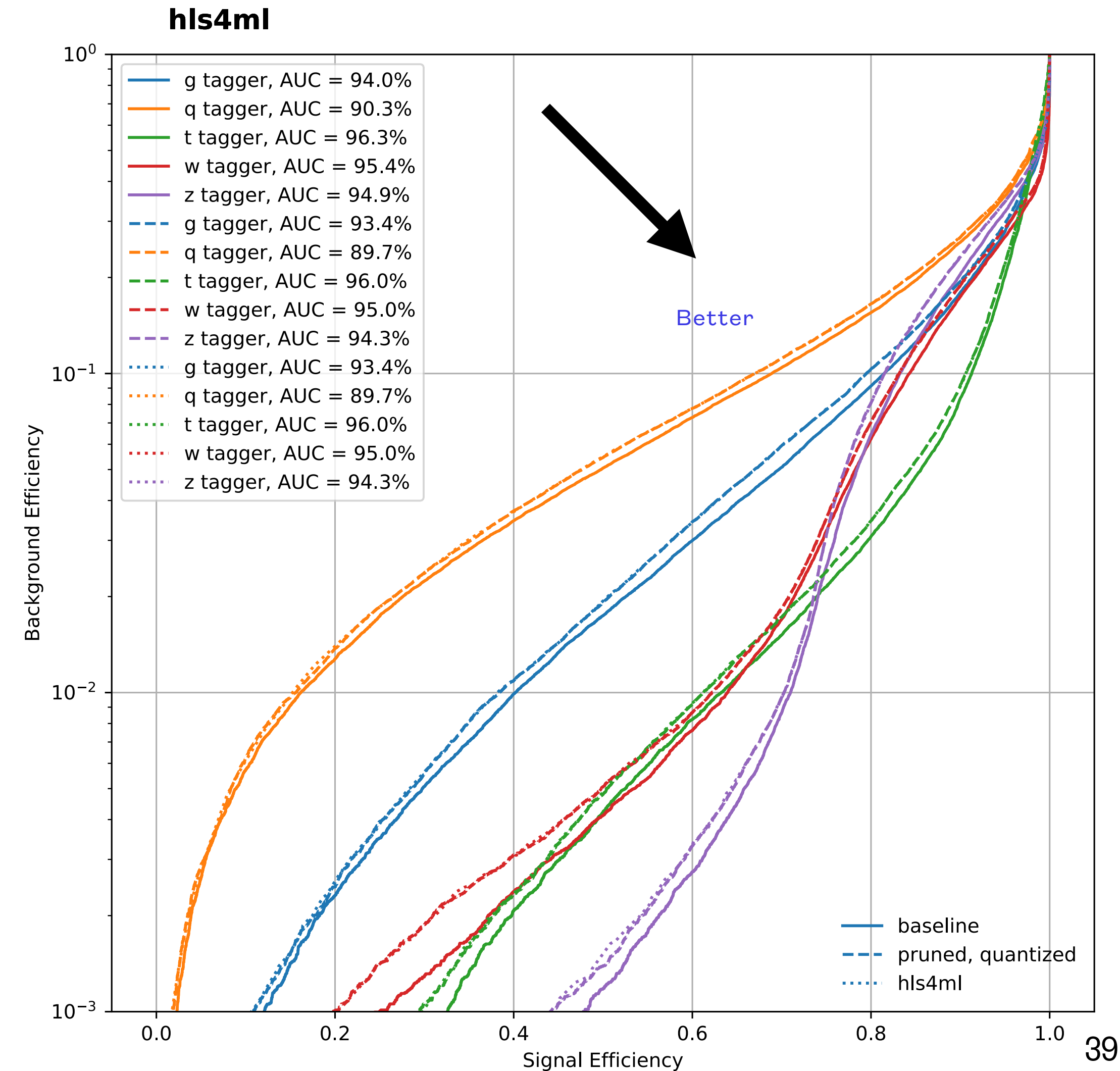
**AUC = AREA UNDER ROC CURVE  
(100% IS PERFECT, 20% IS RANDOM)**

# Jet tagging w/ QAT & Pruning

- A. **Keras** floating point training, 16b inference
- B. **QKeras** with 6 bits for weights, biases, activations & 75% sparsity target with TFMOT

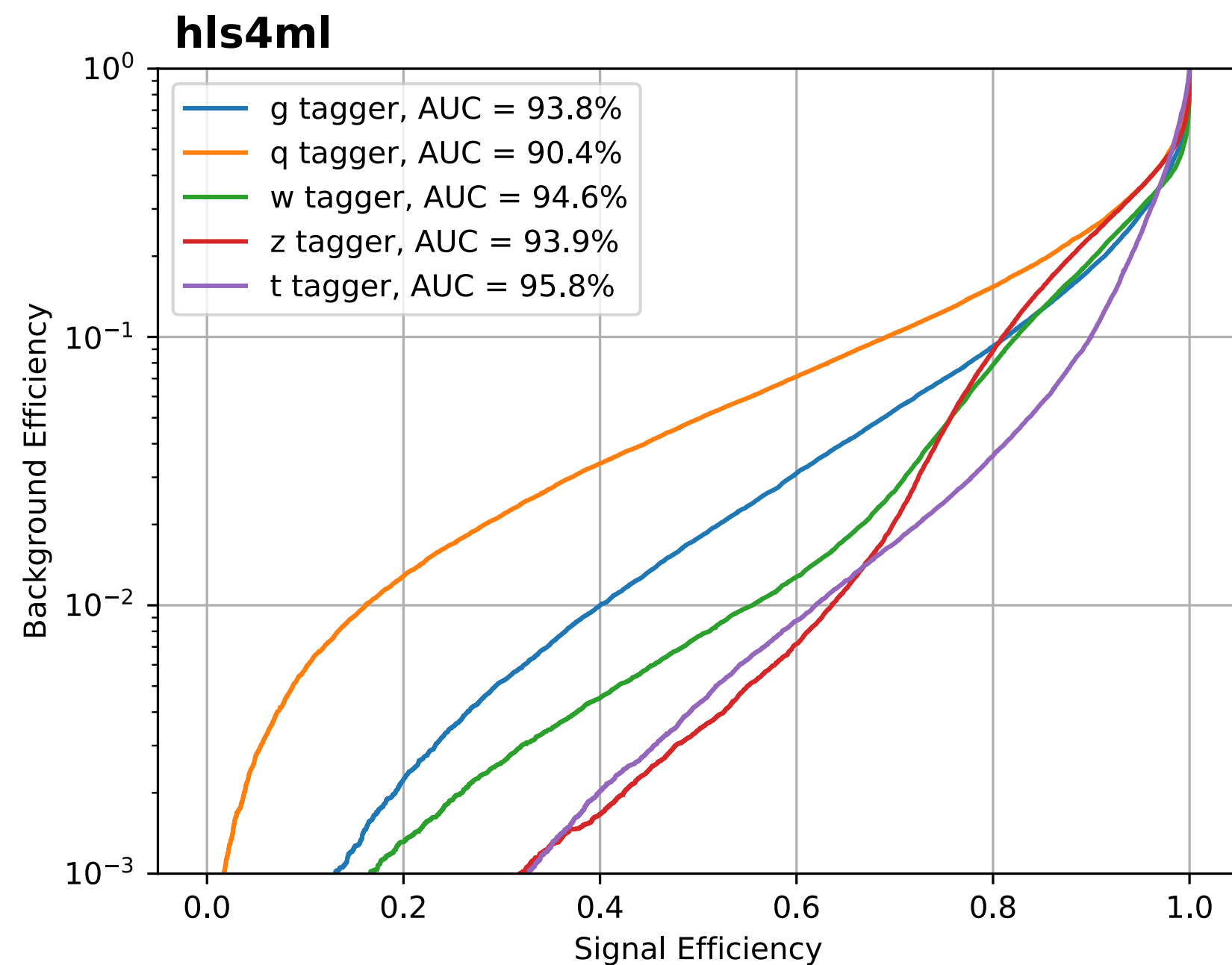
Minimal code changes to go A to B

Xilinx VU9P	Latency	DSP	LUT
Keras 16b	50 ns	1890 (15%)	5%
QKeras 6b	40 ns	22 (~0%)	1%



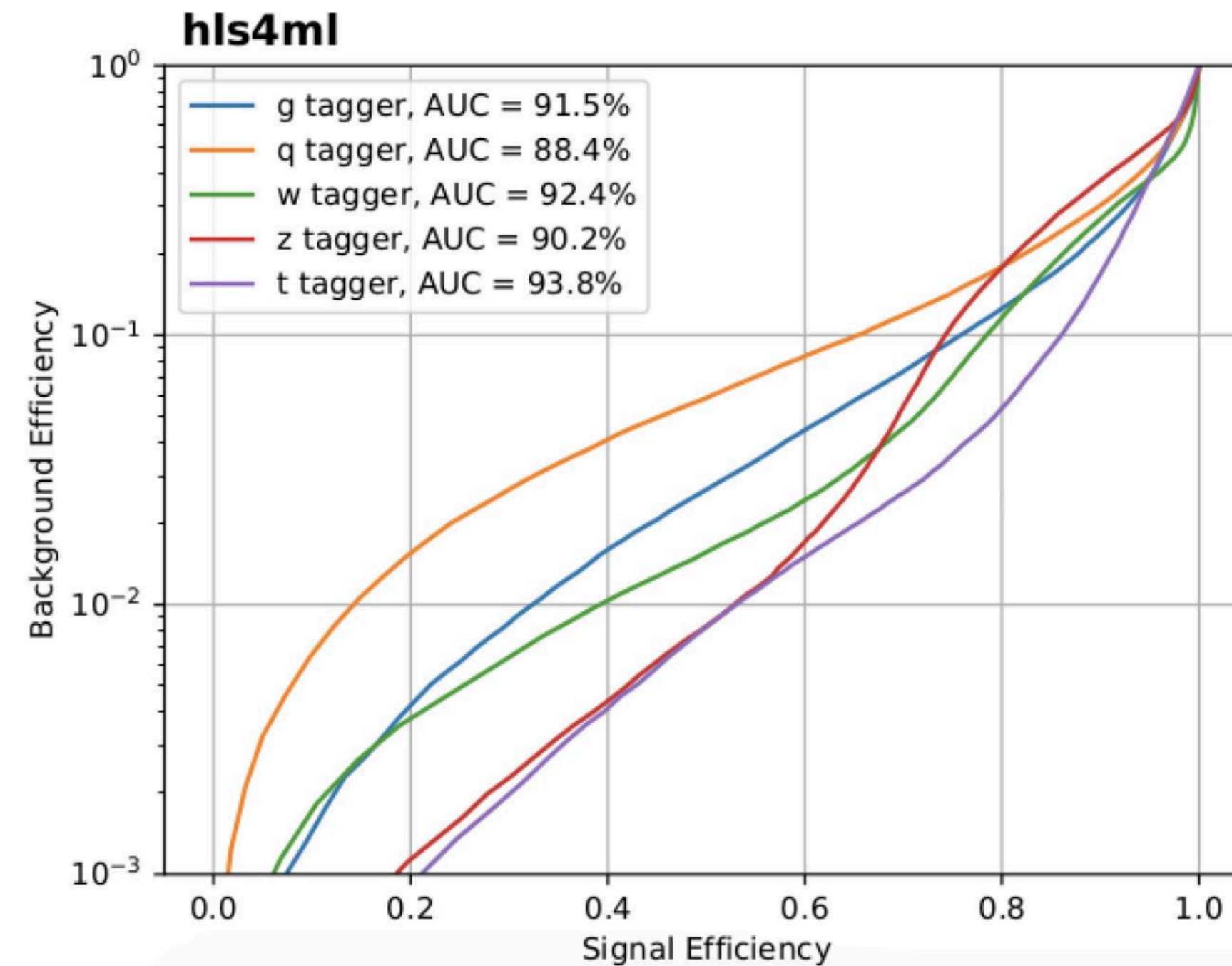
# BNN - Jet Classification

- Design an architecture to perform the same jet classification task but now with binary weights and activations - n neurons 7x per layer
- Performed hyperparameter optimization to find most performant model within some constraints



**Original: 16-bit weights**

**Average accuracy: 0.75**



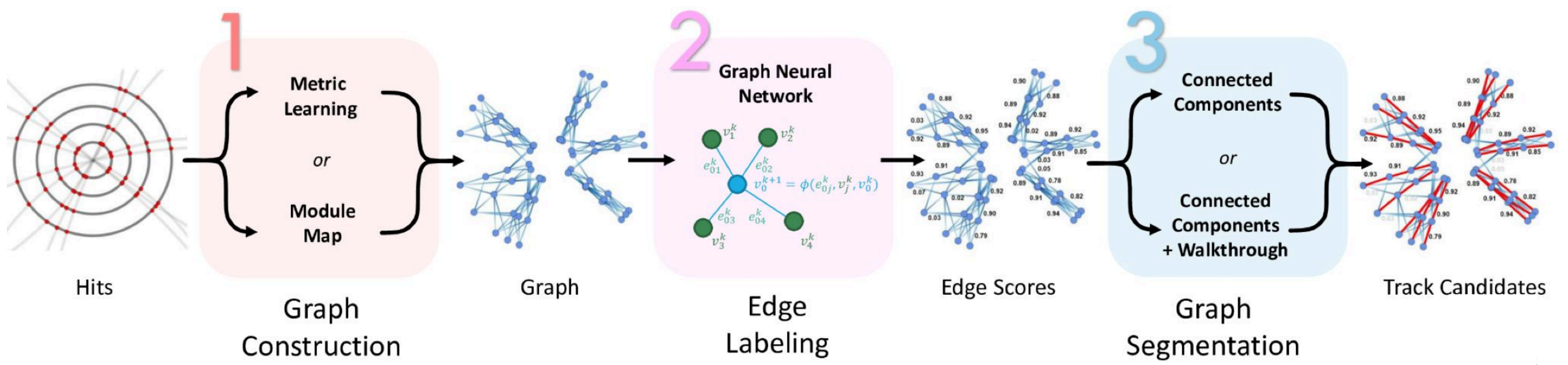
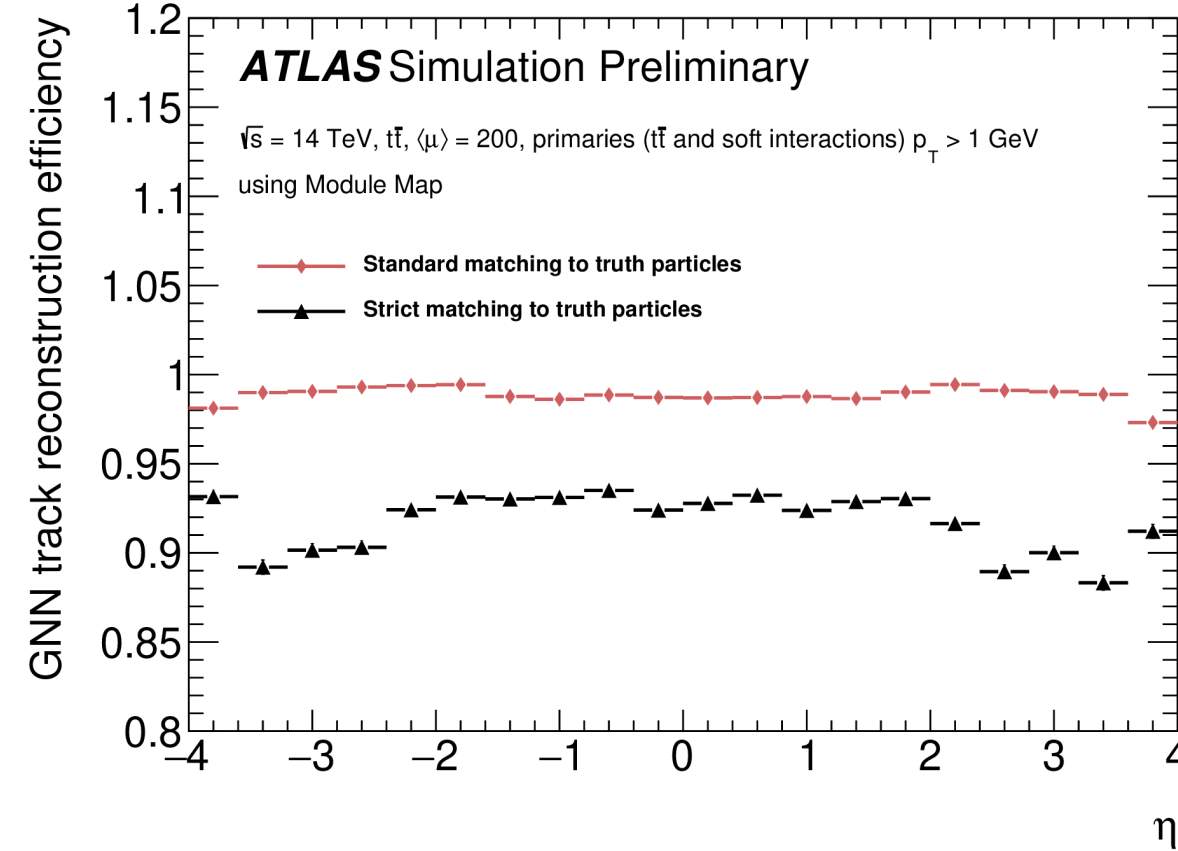
**Binarized: 1-bit weights**

**Average accuracy: 0.72**



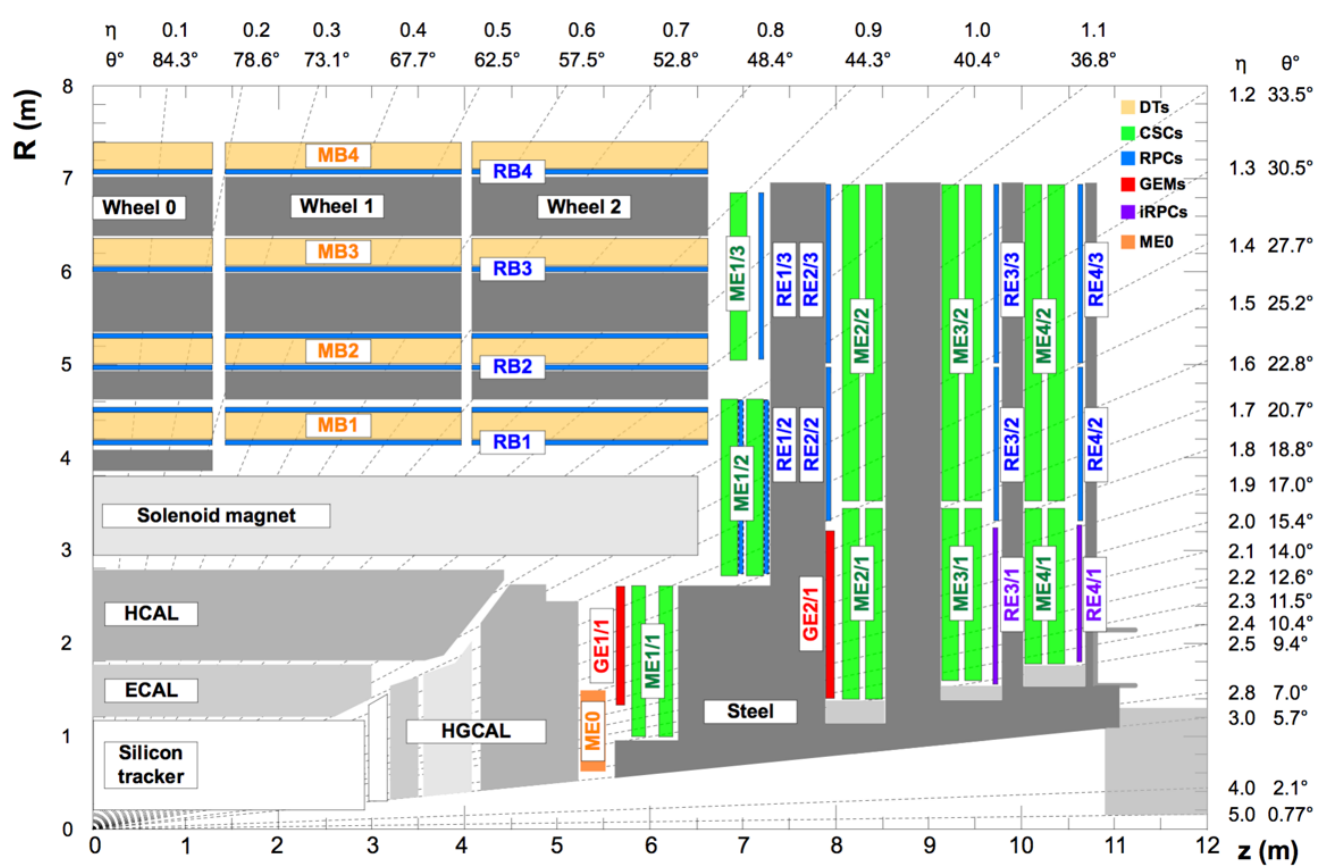
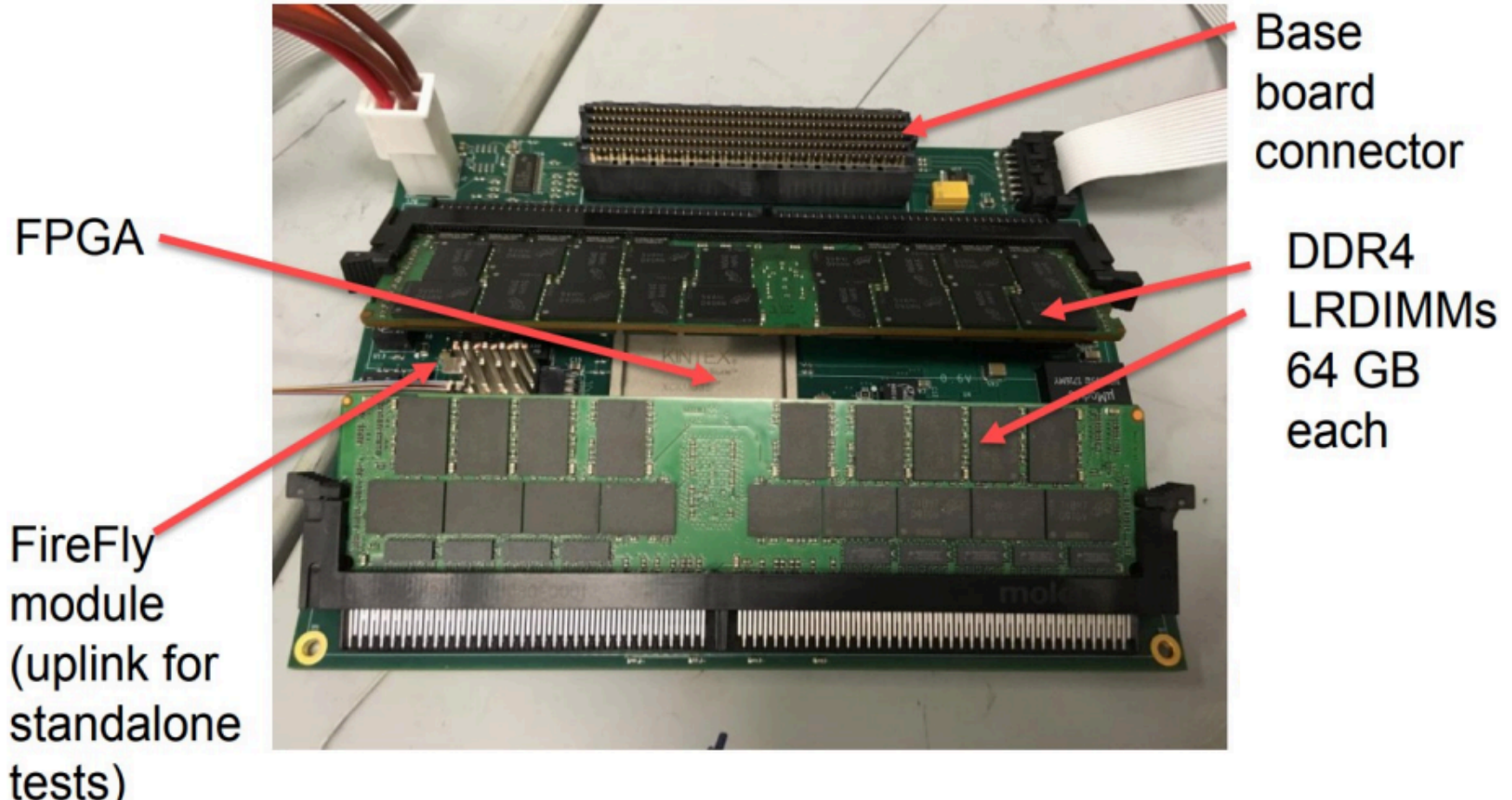
# ATLAS event filter tracking with GNNs

- ATLAS upgraded tracker & trigger for HL-LHC : O(10k) particles per 25ns bunch crossing
- High particle density -> track reconstruction computationally intensive -> scalability challenges
  - Exploring ML/GNN solution on heterogeneous architecture CPU + GPU + FPGA
  - MLP converts hits in detector -> graphs where edges are possible track segments; edges classified by learned geometry
- GNN -> HLS4ML -> Intel S10 GX FPGA



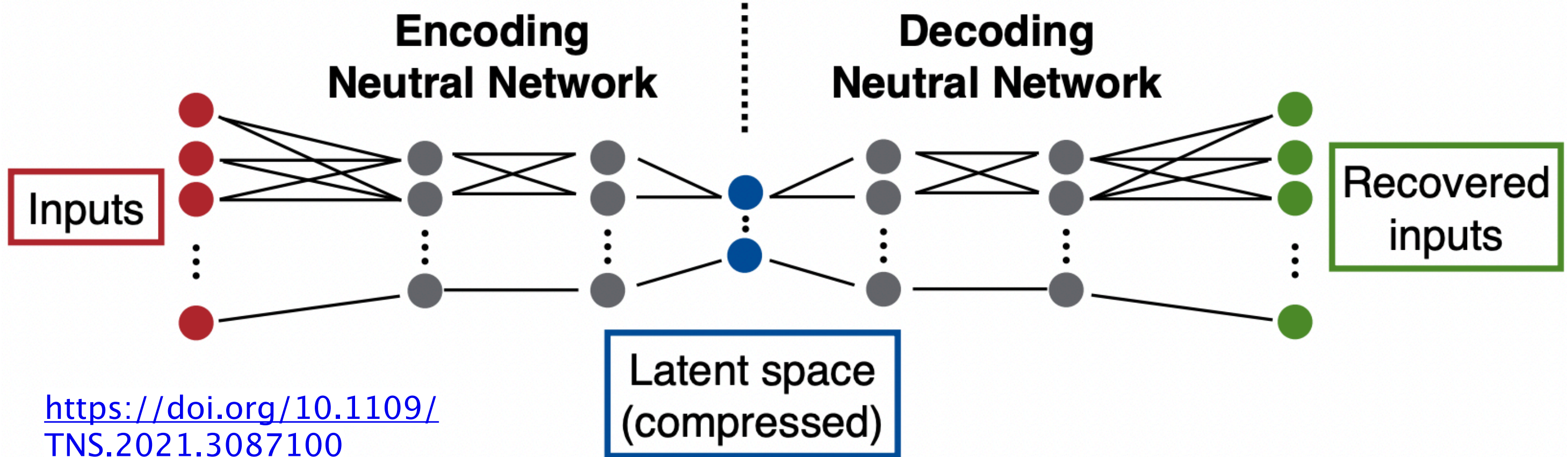
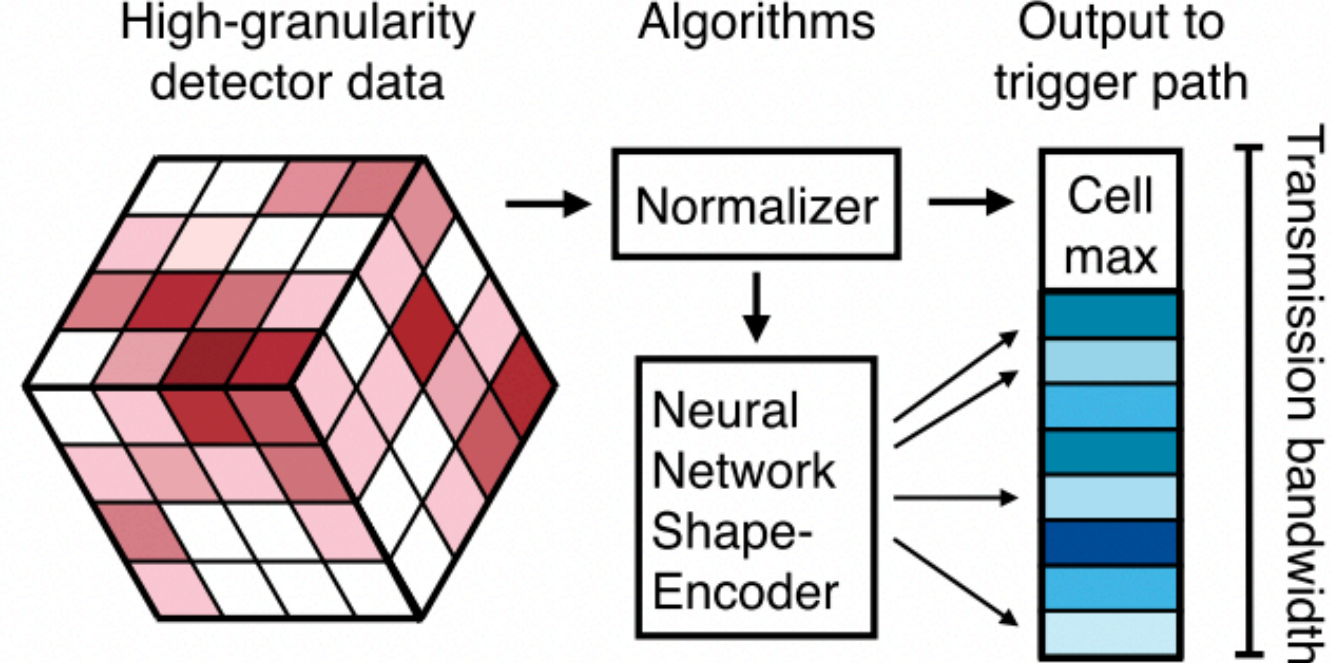
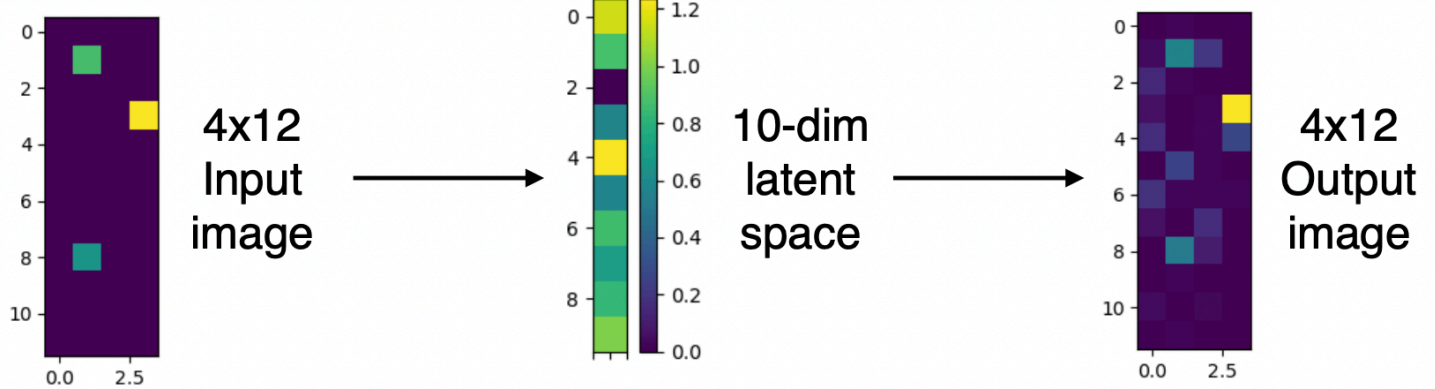
# CMS Level 1 Trigger Endcap Muon Track Finder

- BDT to fit the muon momentum from hits in the muon stations
- Complicated geometry and magnetic field makes an ML solution useful
- Deployed using a 'large LUT' implemented in DDR on a mezzanine card to the FPGA
- BDT is evaluated for every possible input, with the output written at that position in the LUT



# Autoencoder ASIC for CMS upgrade

- New *high granularity calorimeter*, 6.5M readout channels
- **ECON-T**: compress data **on detector** with AutoEncoder, decode off detector
- Inference on chip matches software implementation; costs 75-100mW
- Radiation tolerant (triplication), cooled to -30 °C, 1.5µs latency
- NN architecture fixed, weights & biases re-programmable



<https://doi.org/10.1109/TNS.2021.3087100>

**Active Elements:**

- Hexagonal modules based on Si sensors in CE-E and high-radiation regions of CE-H
- "Cassettes": multiple modules mounted on cooling plates with electronics and absorbers
- Scintillating tiles with SiPM readout in low-radiation regions of CE-H

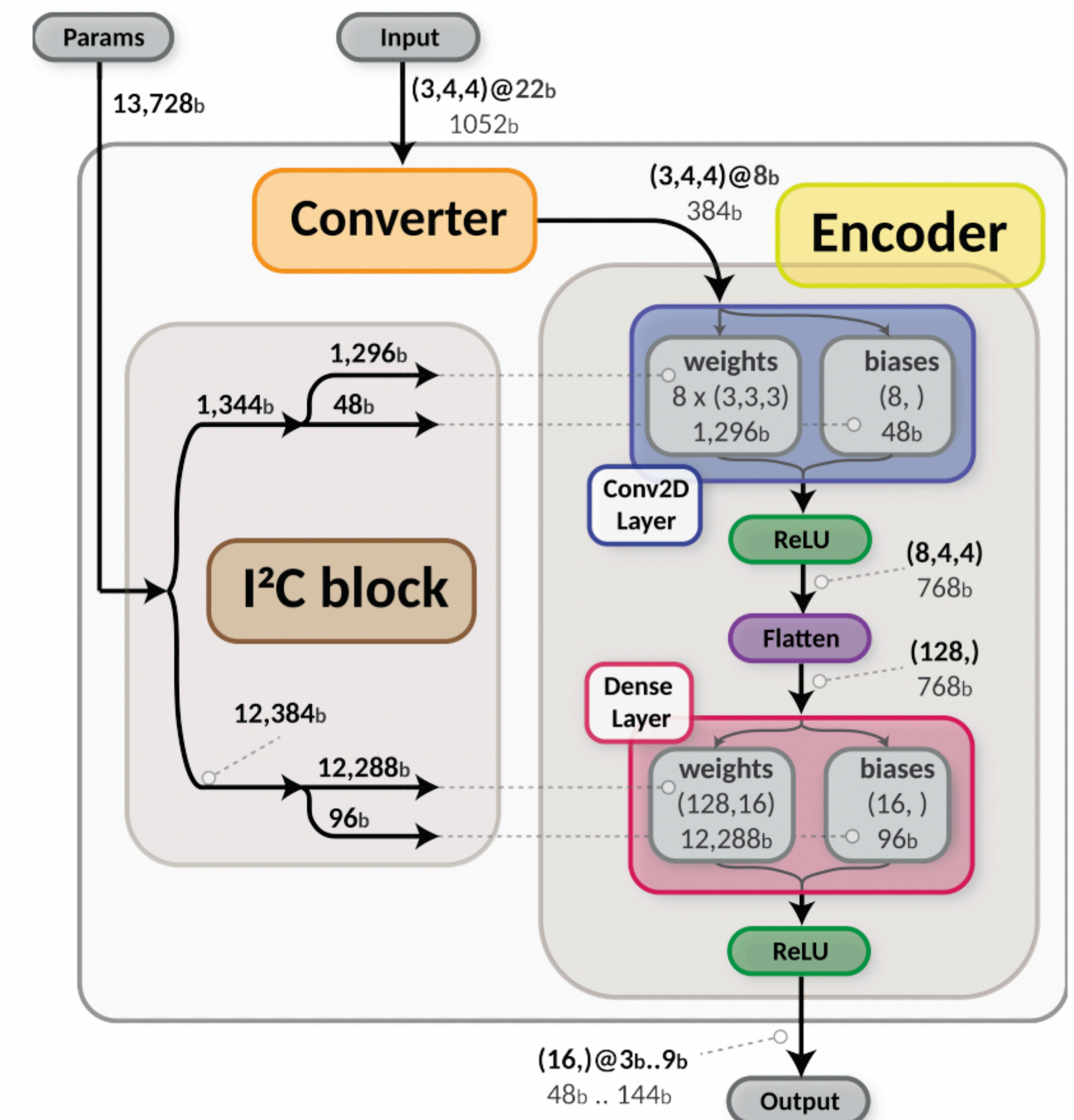
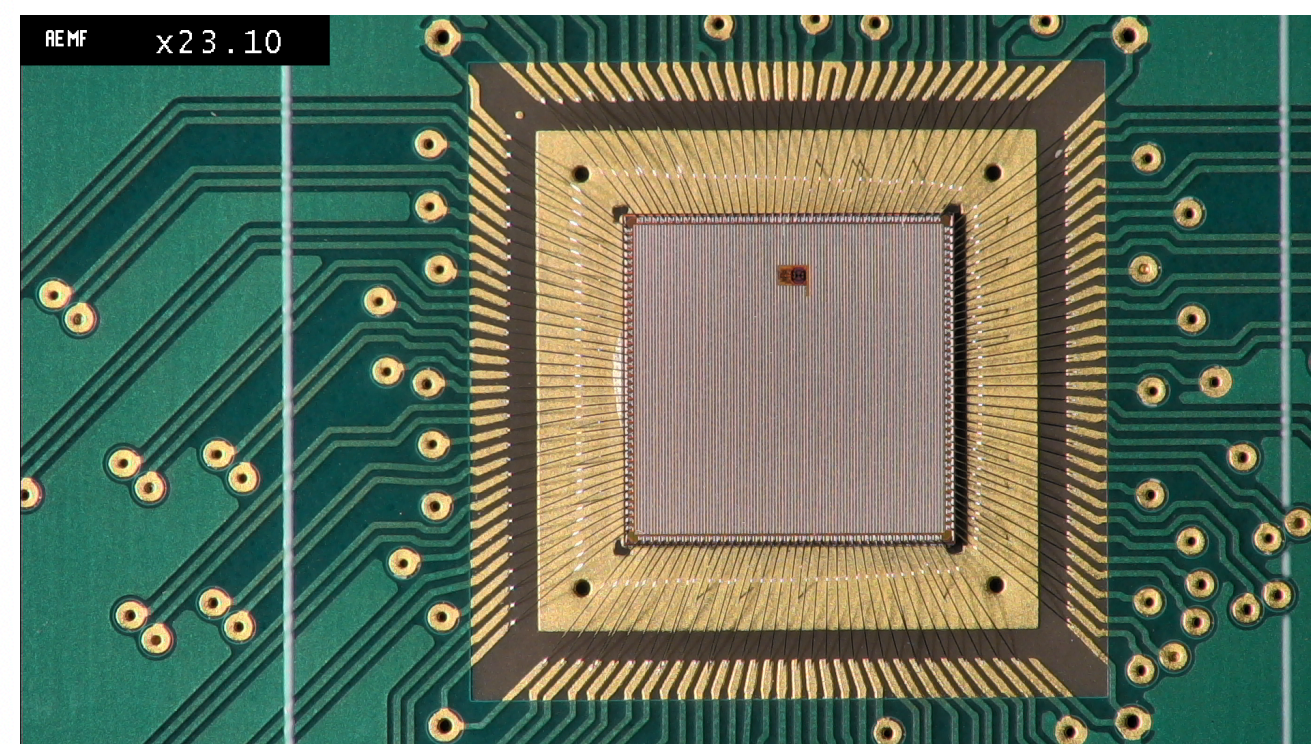
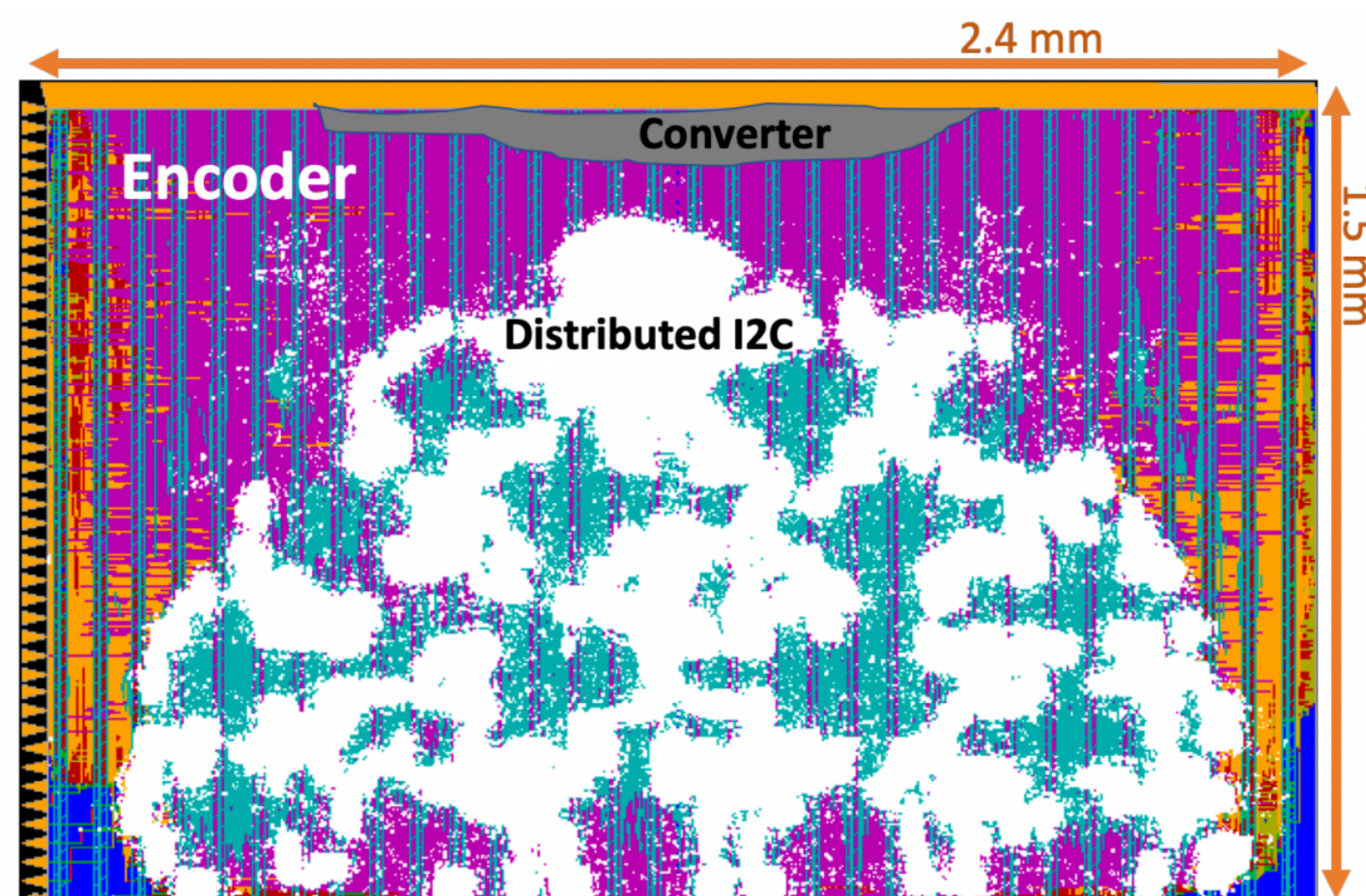
**Key Parameters:**

- Coverage:  $1.5 < |\eta| < 3.0$
- ~215 tonnes per endcap
- Full system maintained at -30°C
- ~600m<sup>2</sup> of silicon sensors
- ~500m<sup>2</sup> of scintillators
- 6M Si channels, 0.5 or 1 cm<sup>2</sup> cell size
- ~27000 Si modules
- Power at end of HL-LHC: ~110 kW per endcap

Electromagnetic calorimeter (CE-E): Si, Cu & CuW & Pb absorbers, 28 layers, 25 X<sub>0</sub> & ~1.3λ. Hadronic calorimeter (CE-H): Si & scintillator, steel absorbers, 24 layers, ~8.5λ.

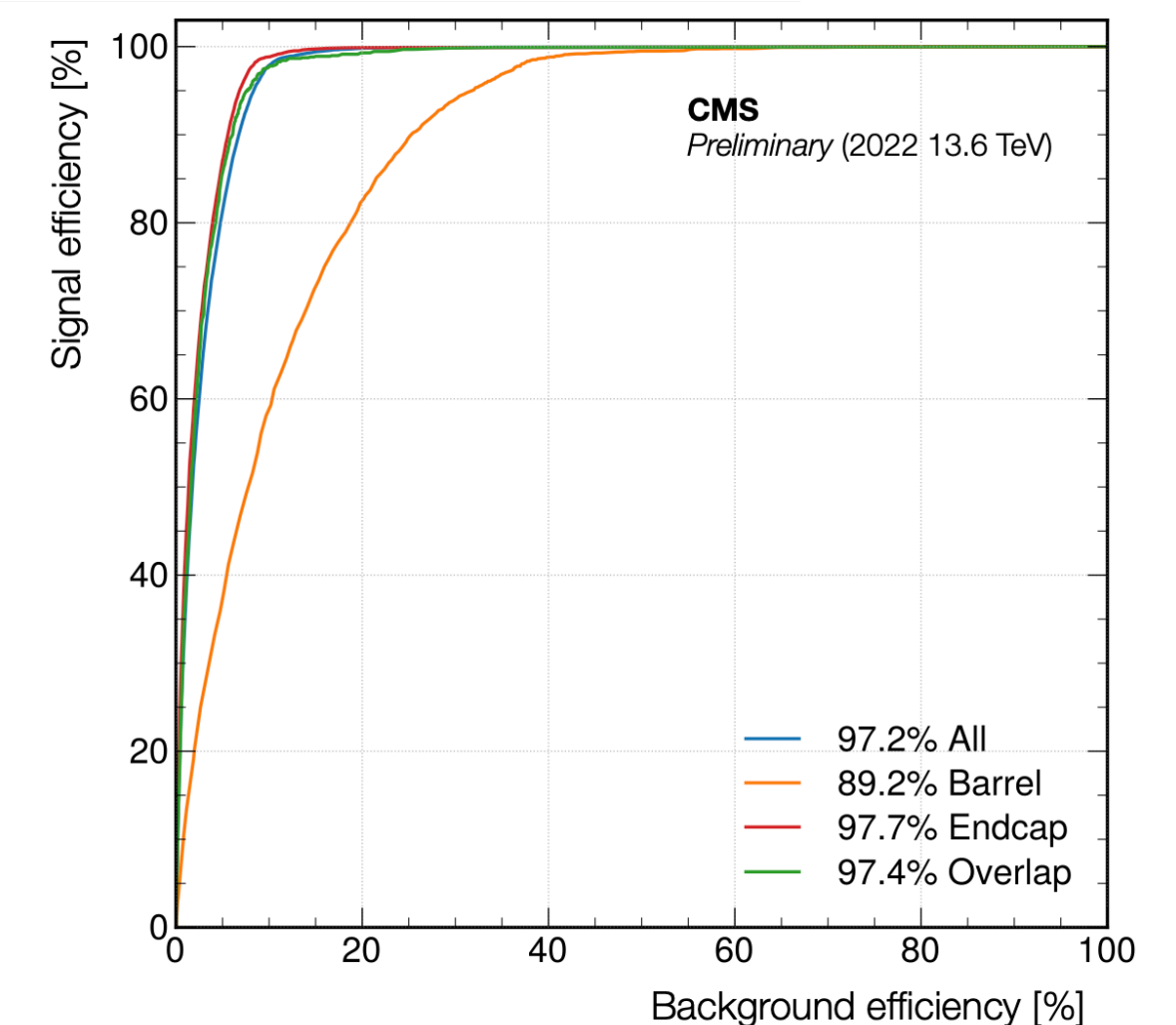
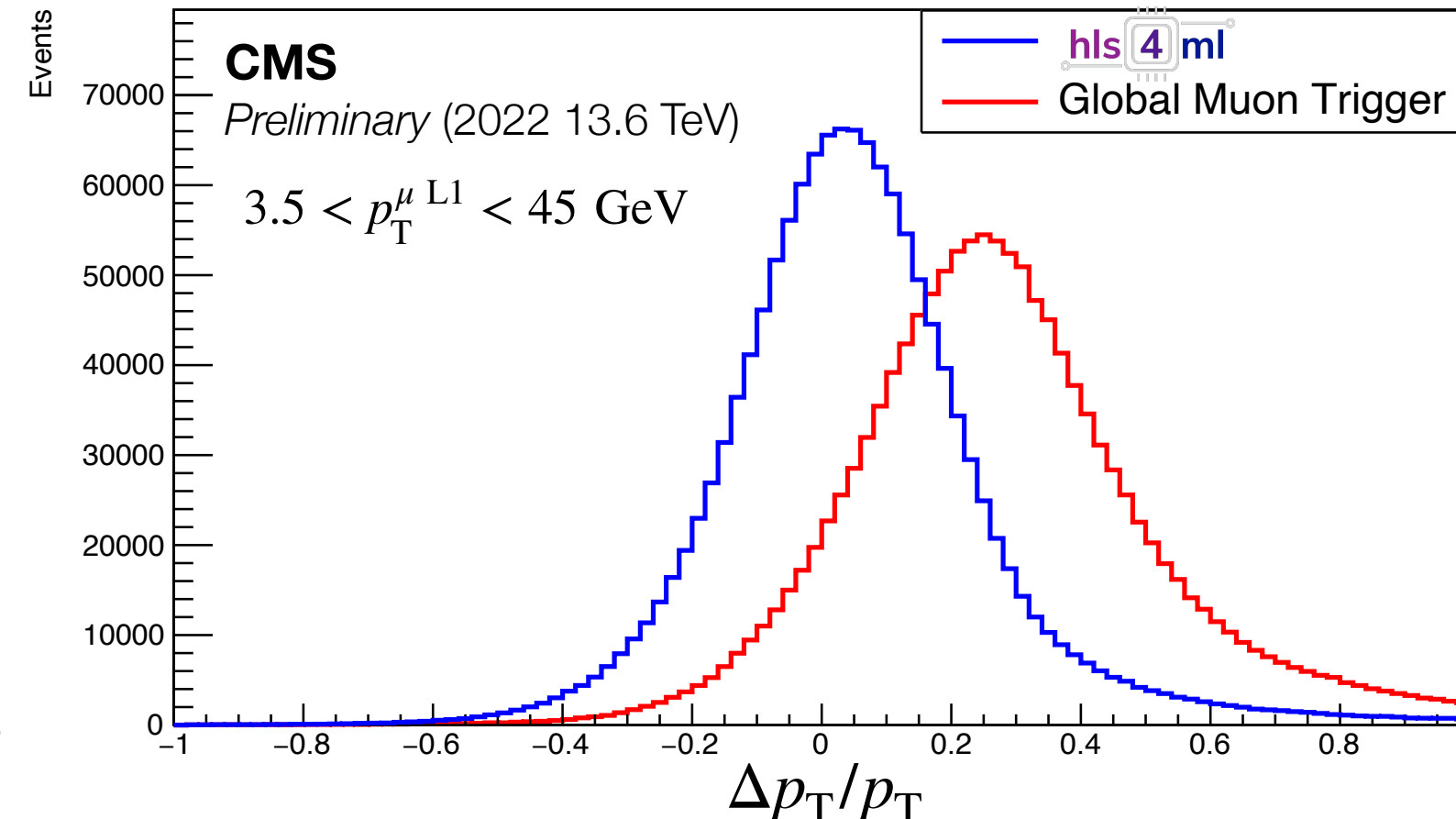
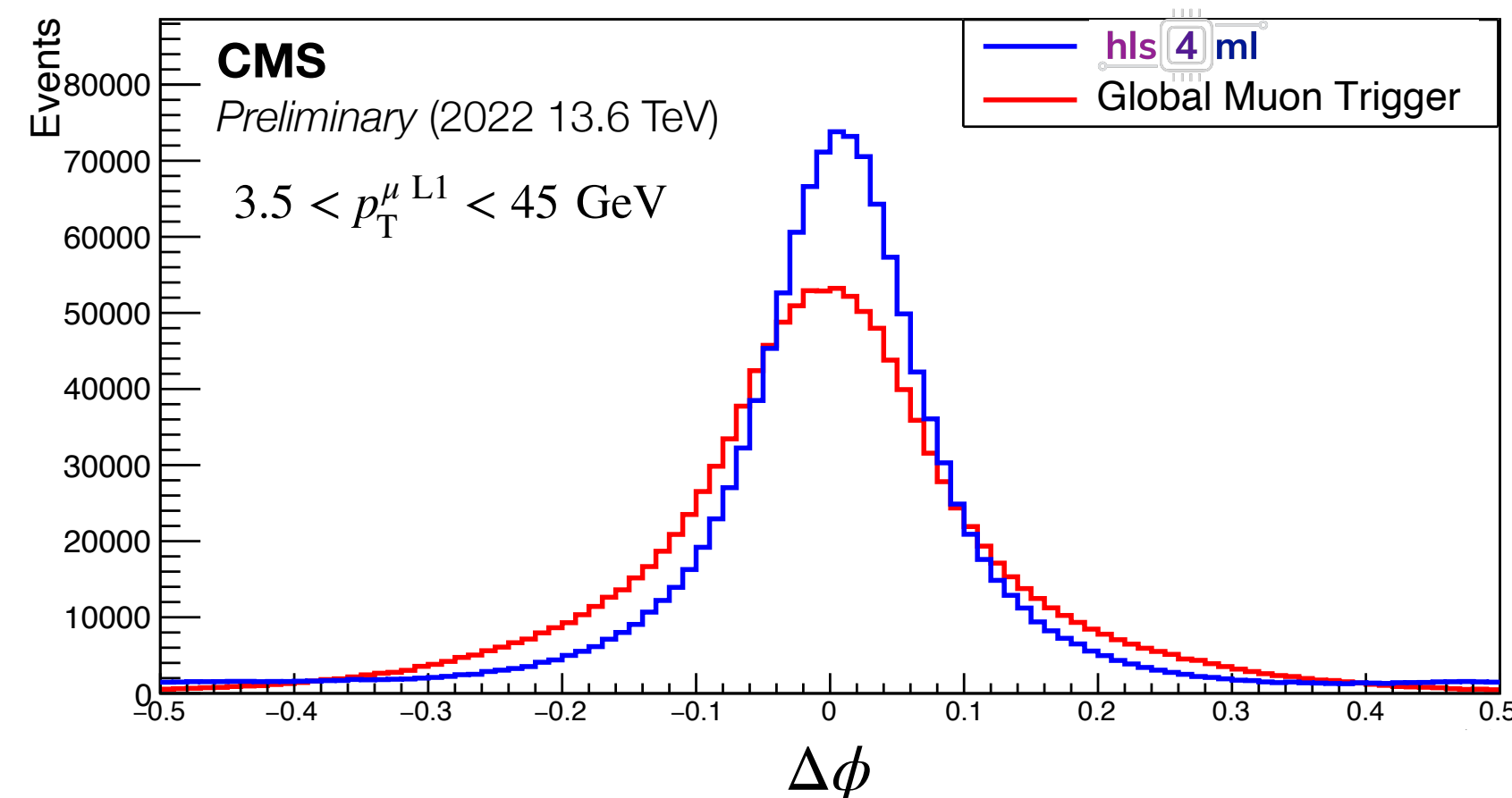
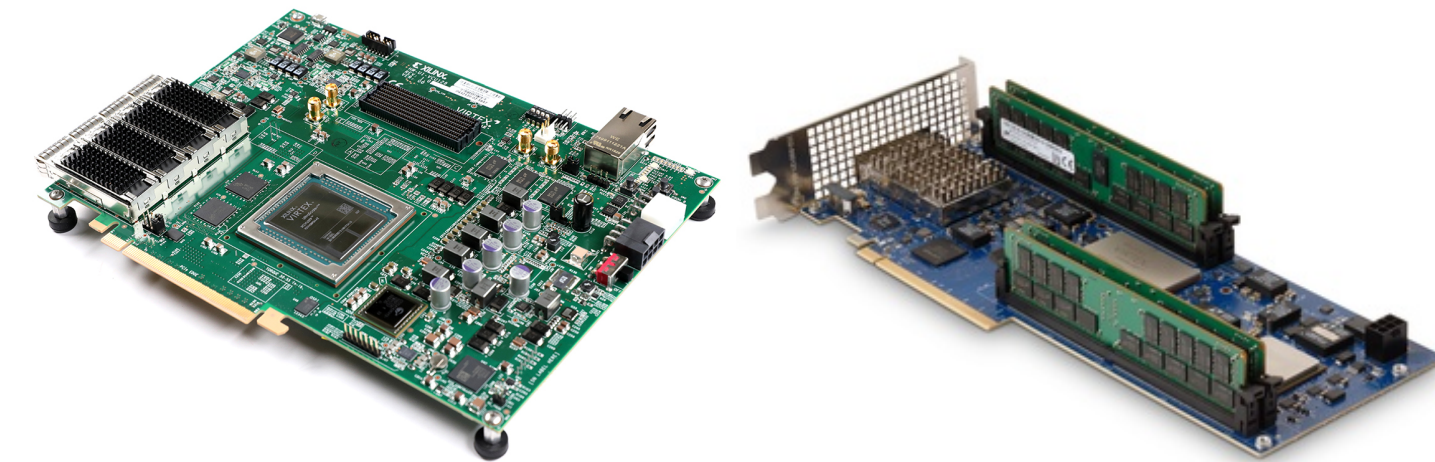
# Autoencoder ASIC for CMS upgrade

- New *high granularity calorimeter*, 6.5M readout channels
- **ECON-T**: compress data **on detector** with AutoEncoder, decode off detector
- Inference on chip matches software implementation; costs 75-100mW
- Radiation tolerant (triplication), cooled to  $-30\text{ }^{\circ}\text{C}$ ,  $1.5\mu\text{s}$  latency
- NN architecture fixed, weights & biases re-programmable



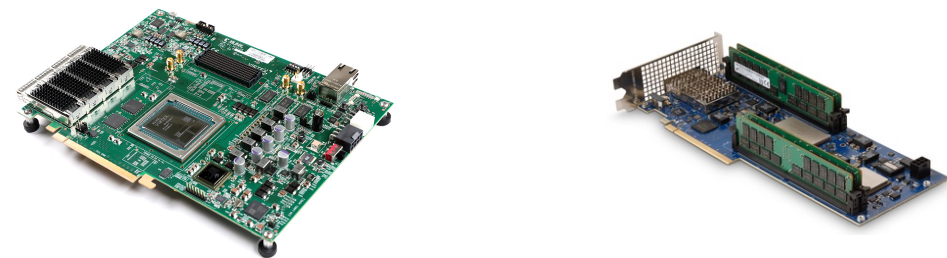
# CMS L1 scouting: muon re-calibration

- L1 scouting at CMS acquires **subset of granular data at full bunch crossing rate**
  - Use ML to recover resolution / accuracy in comparison with full reconstruction
  - Fully-connected DNN for muon recalibration generated w/ HLS4ML; 4 layers with 8-256 nodes
  - **Knowledge distillation** used to obtain compression factor of ~200 with almost no performance loss: no pruning required
  - Runs on Micron SB852 (VU9P) & Xilinx VCU128 (VU37P) boards
- Improvement for all variables in comparison with raw trigger values



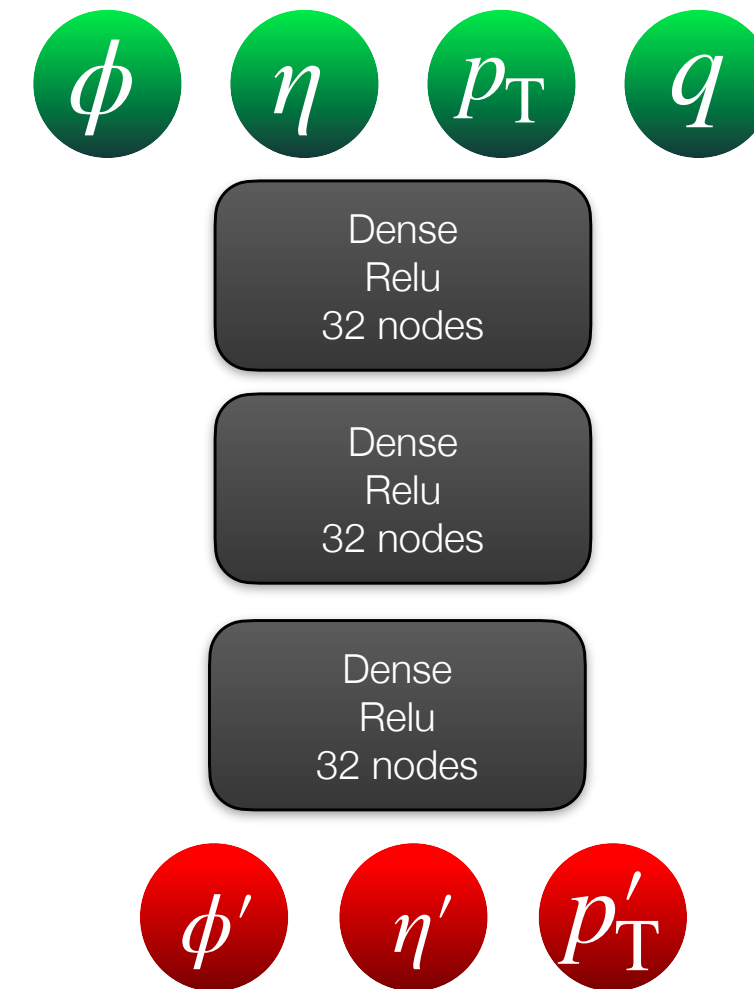
# CMS L1 scouting

- Q6.12 precision, pruning factor 0.5
- Idea: 2 NN each process 4 muons / BX
- Latency  $\lesssim 100$  ns FIFO latency, can accept 2 muons / clock

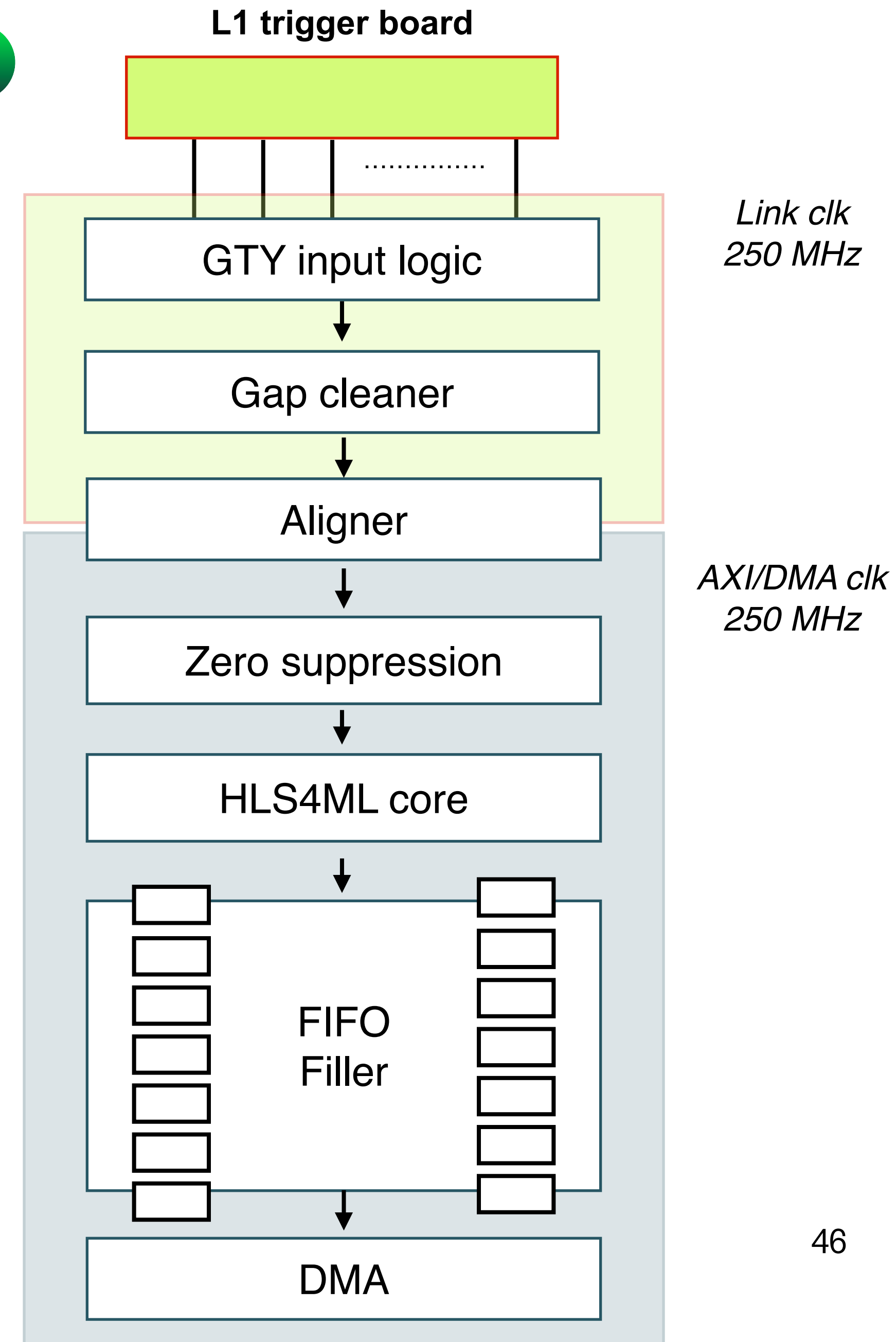
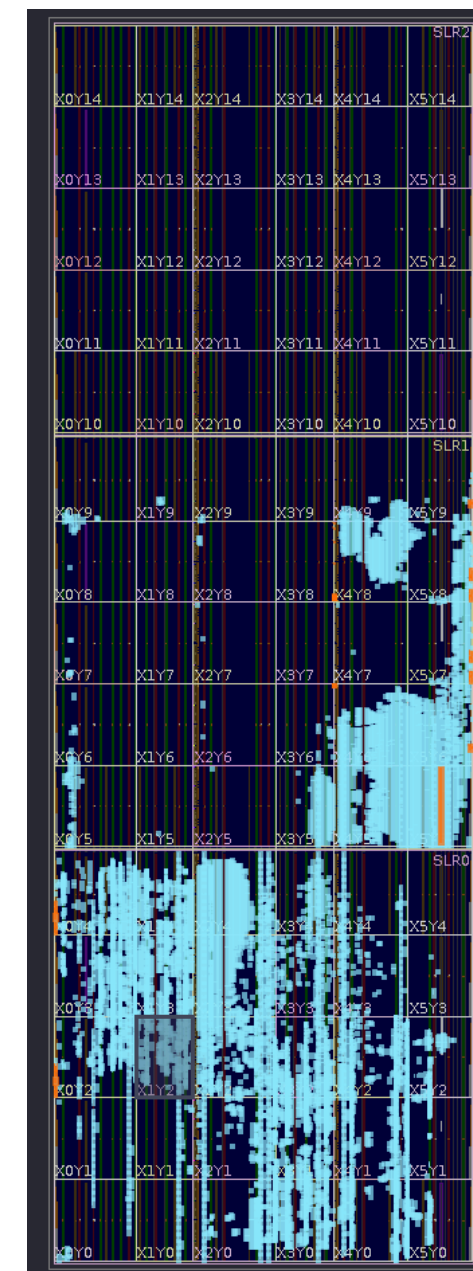


	VU37P	VU9P
LUTs	57k / 4%	52k / 5%
BRAM (all infra)	584 / 29%	610 / 28%
DSP (all NN)	992 / 11%	1016 / 15%

post-implementation of full design w/ scouting fw & infrastructure



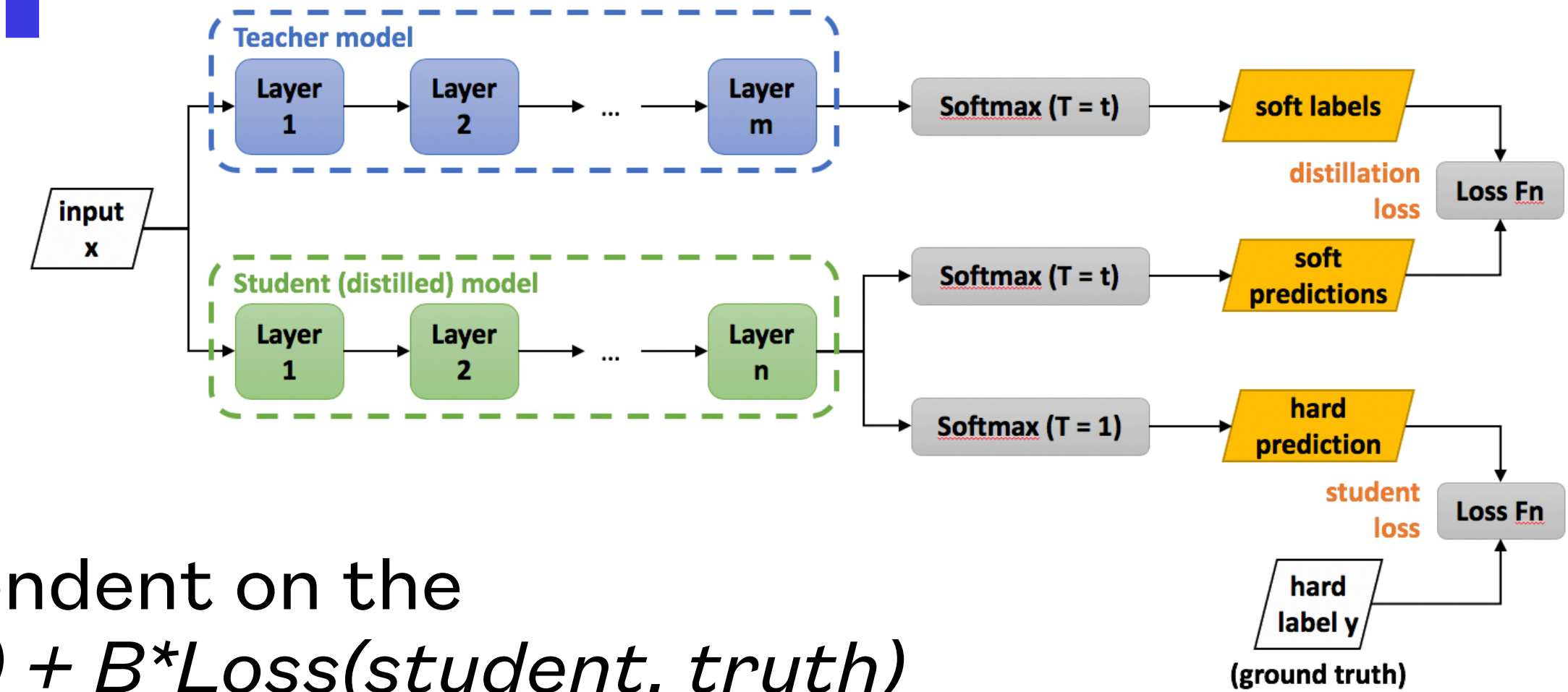
VU9P (SB852) floorplan



# Knowledge distillation

[https://intellabs.github.io/distiller/knowledge\\_distillation.html](https://intellabs.github.io/distiller/knowledge_distillation.html)

- Process of transferring knowledge from large model to smaller model i.e reducing computations while maintaining performance
- Larger Teacher model pre-trained
- Smaller Student model trained with loss function dependent on the output of the teacher model:  $A * \text{Loss}(\text{student}, \text{teacher}) + B * \text{Loss}(\text{student}, \text{truth})$
- In CMS L1 Scouting: Applied to both re-calibration & classification: both models 4 layers; precision AP (18,5)

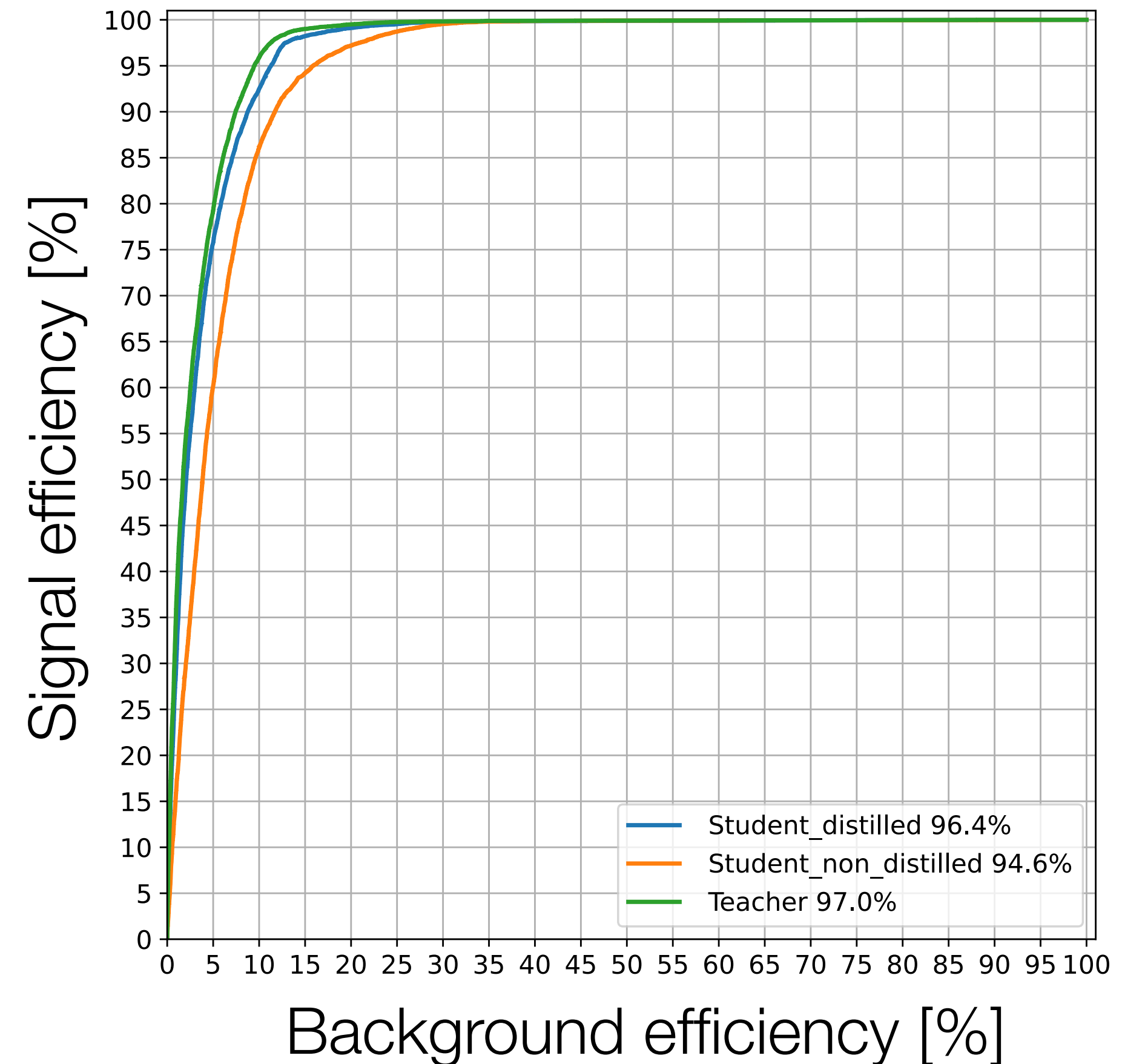


Performance comparison	N Neurons / layer	N parameters total	FWHM			Classification: area under ROC curve
			$\Delta\phi$ [rad]	$\Delta\eta$	$\Delta p_T$ [GeV]	
Teacher	256	204 000	0.12	0.061	0.41	0.97
Student	8	419	0.12	0.063	0.42	0.96
Original baseline	16	1219	0.14	0.066	0.43	0.97

- Compression factor of ~200 with almost no performance loss: no pruning required

# Knowledge distillation

- Applied to both re-calibration & classification: both models 4 layers; precision AP (18,5)
- Same model architecture for re-calibration & classification
- 250 MHz clock
- Can easily fit multiple copies in FPGA

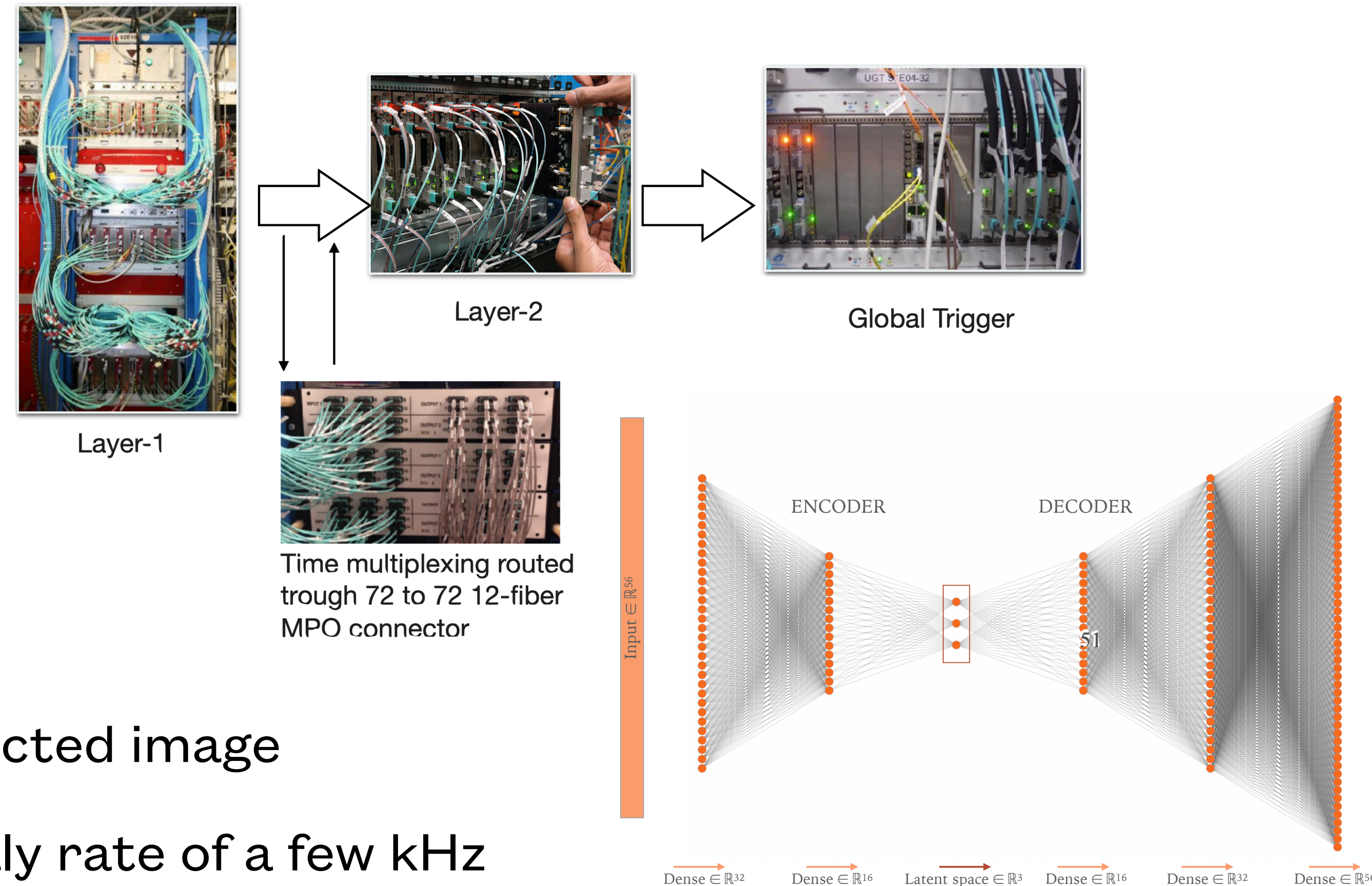


VU9P FPGA	DSP	Flip Flops	Look Up Table	BRAMS
Available	9024	2.6 M	1.3 M	2160
Used	72 (0.79%)	5677 (0.21%)	11.3 K (0.87%)	0

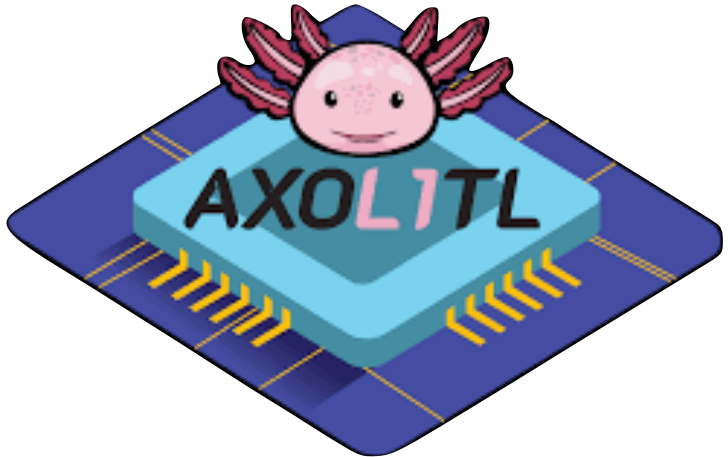


# Anomaly detection in CMS trigger

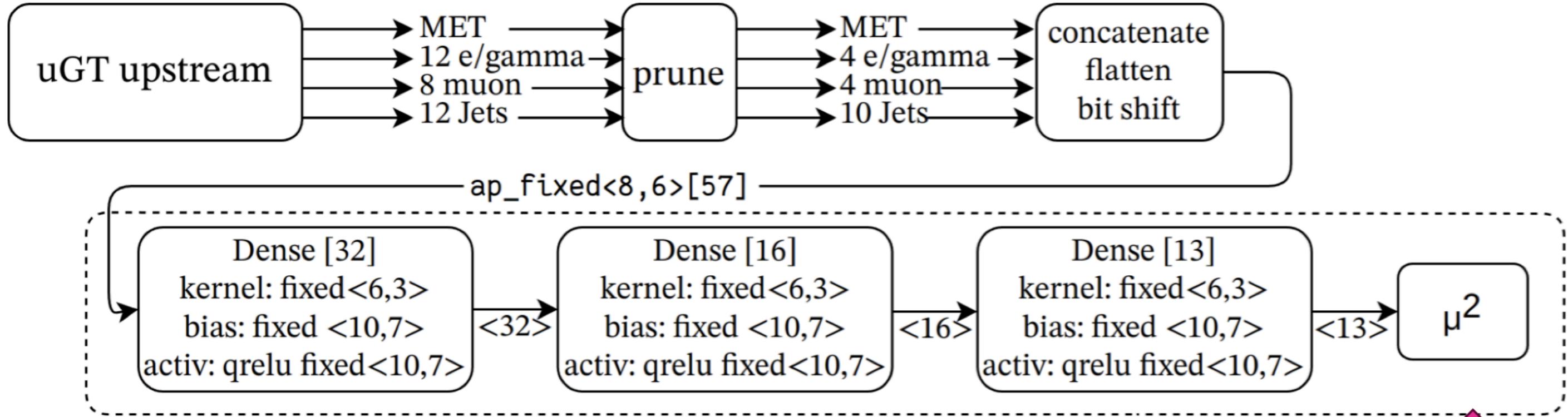
- Unknown signature for new physics
- New signals may be difficult to trigger on with standard cuts-based selection
- **CICADA** [1]: (Calorimeter Image Convolutional Anomaly Detection Algorithm)
  - 2D CNN autoencoder runs on the calorimeter region energy deposit topologies
  - 18x14 pixel input; MSE loss from re-constructed image
  - Implemented in calorimeter trigger: anomaly rate of a few kHz



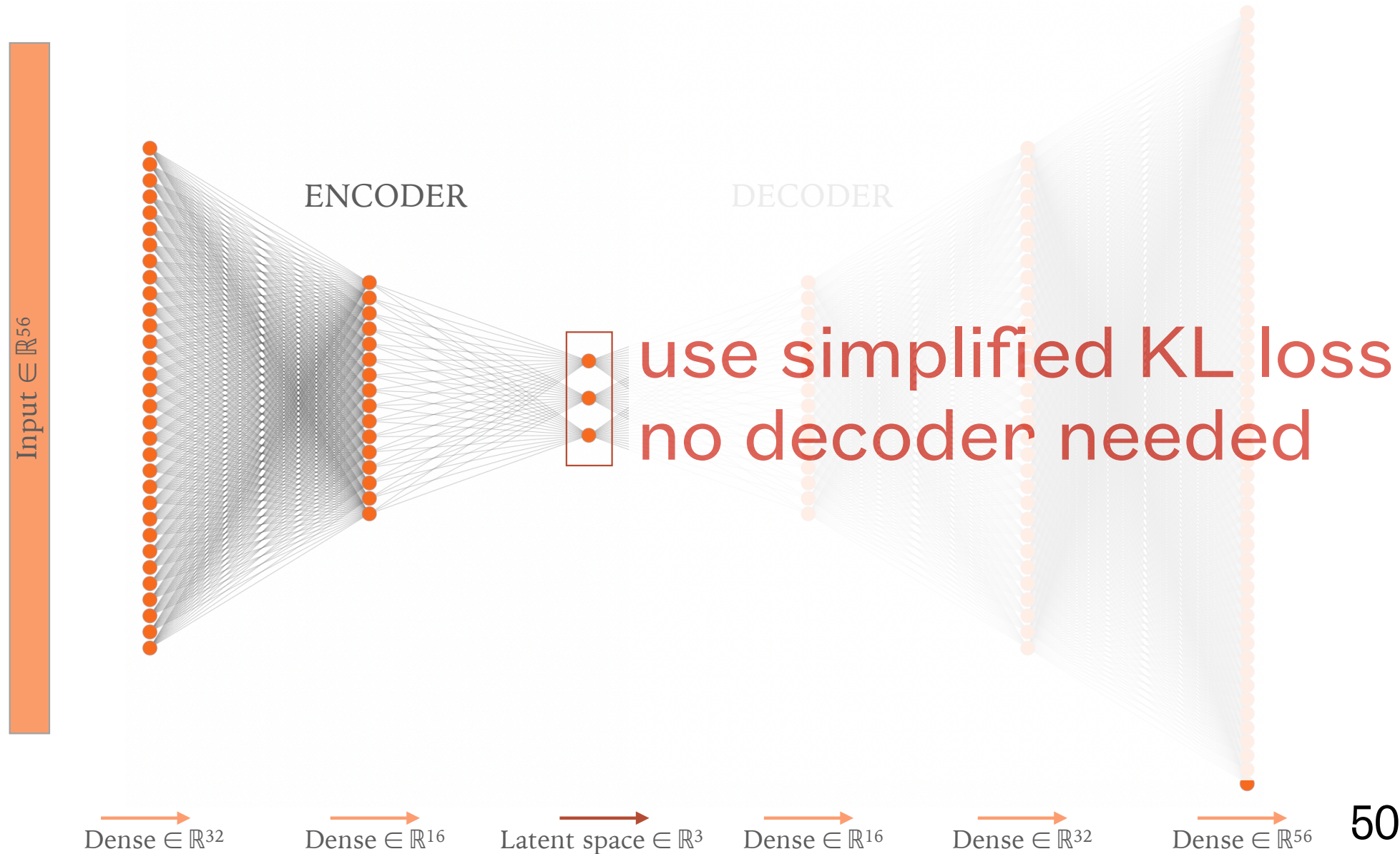
# Anomaly detection in CMS trigger



- **AXOLITL** [2]: variational auto encoder to select anomalous events in real time
  - Trained on minimally biased, unfiltered data from detector, simplified KL loss
  - Firmware developed with HLS4ML on FPGA-based custom global trigger board (Xilinx Virtex-7), meets strict timing and resource limitations; 50 ns latency / inference
  - Physics performance improvements in (*beyond*) standard model signals - could we find something we have been missing?



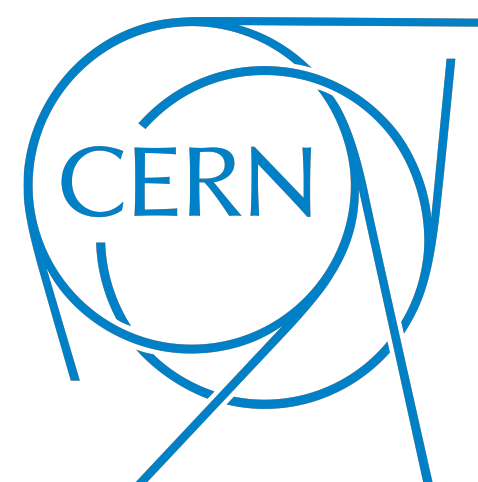
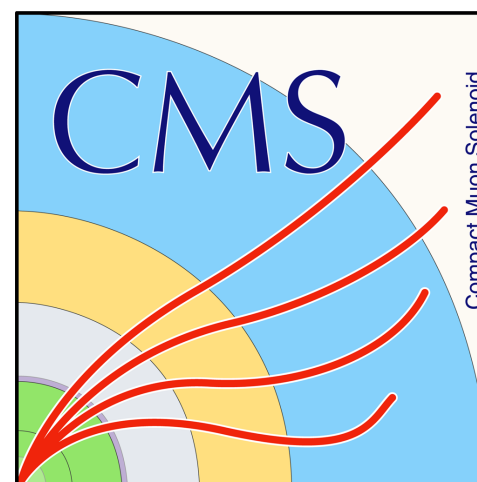
The anomaly score



[2] <https://doi.org/10.1038/s42256-022-00441-3>.

# Summary

- Deploying ML into the realtime processing for Trigger and DAQ is becoming increasingly possible and relevant
- FPGAs *and* ML crucial to processing huge throughput of the HL-LHC
- New tricks of the trade for *fast* ML inference with low latency: quantisation, knowledge distillation
- Many examples at LHC experiments - just a sub-set shown
- Full exploitation of ML inference at the edge is necessary to continue advancing our understanding of the universe
- Many more ideas for what can be done up to and including at the HL-LHC



**Thomas James, [tom.james@cern.ch](mailto:tom.james@cern.ch)**

CERN openlab, CTO for AI and Edge Devices

Applied Physicist, CMS detector

---

# Links and additional reading

[1] <https://fastmachinelearning.org/hls4ml>

[2] <https://github.com/fastmachinelearning/hls4ml-tutorial>

[3] <https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html>

[4] <https://software.intel.com/en-us/articles/accelerating-neural-networks-with-binary-arithmetic>

[5] <https://arxiv.org/abs/1804.06913>

[6] <https://www.nature.com/articles/s42256-021-00356-5>

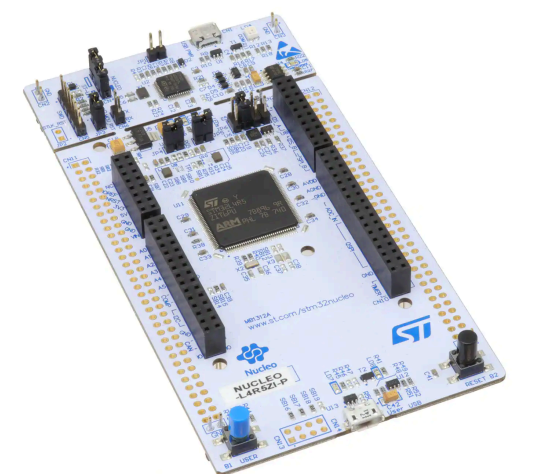
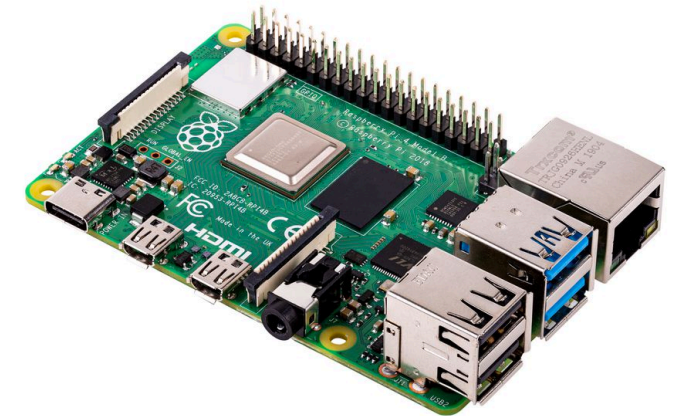
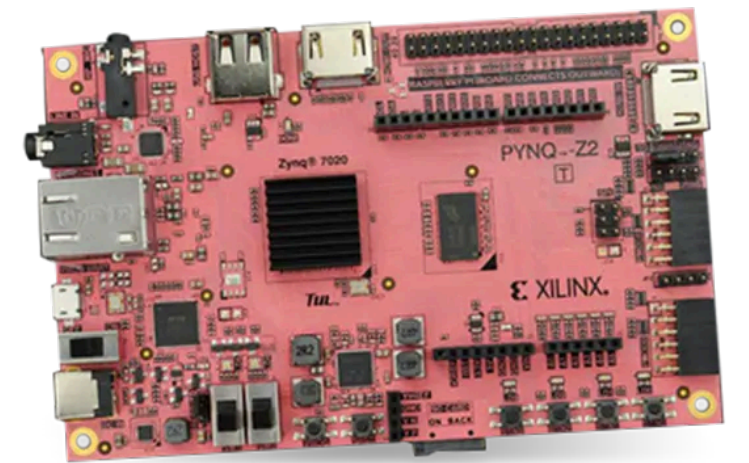
[7] [github.com/fastmachinelearning/qonnx](https://github.com/fastmachinelearning/qonnx)

[8] <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>

[9] <https://www.eidosmedia.com/blog/technology/machine-learning-size-isn-t-everything>

# BACKUP: Fast ML benchmarking

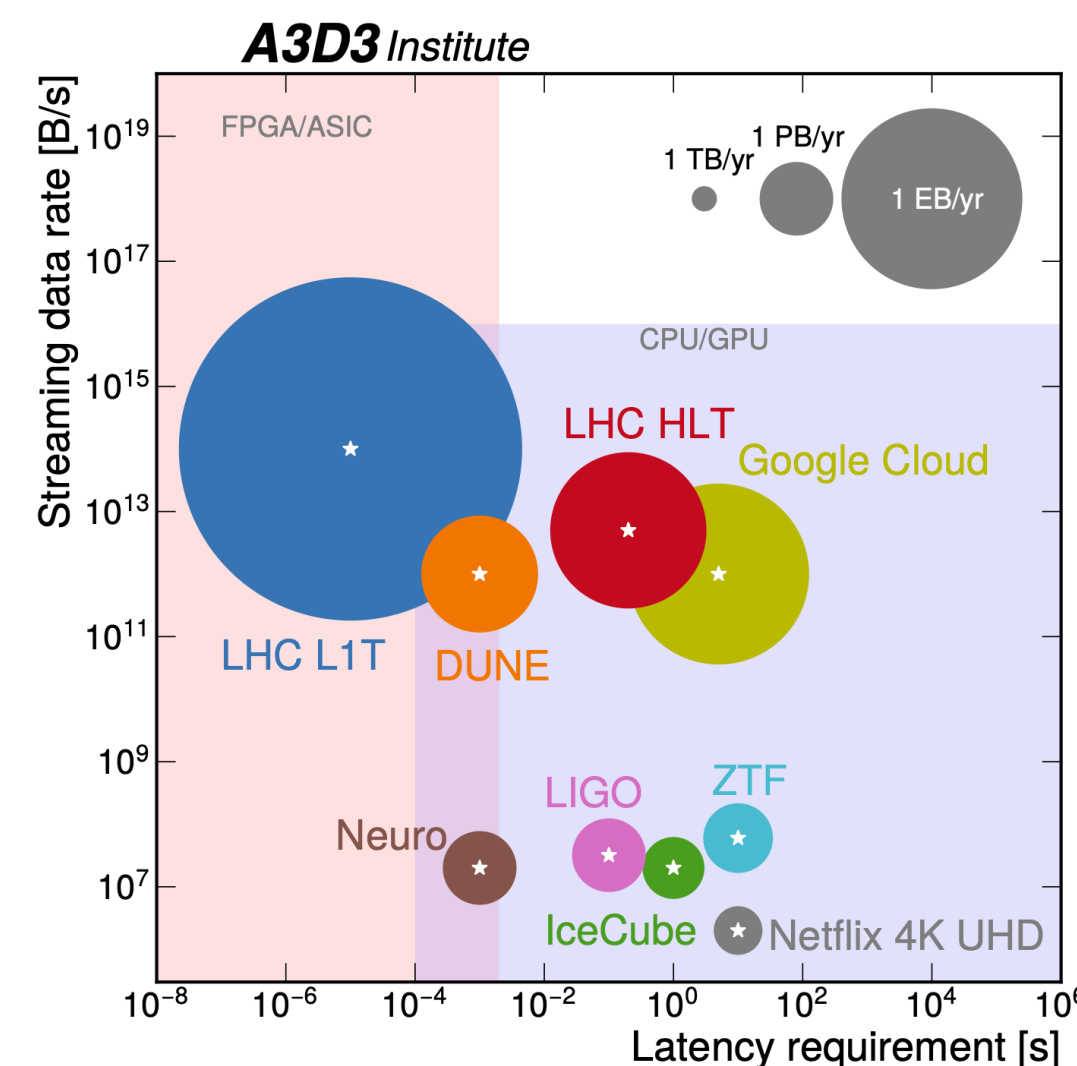
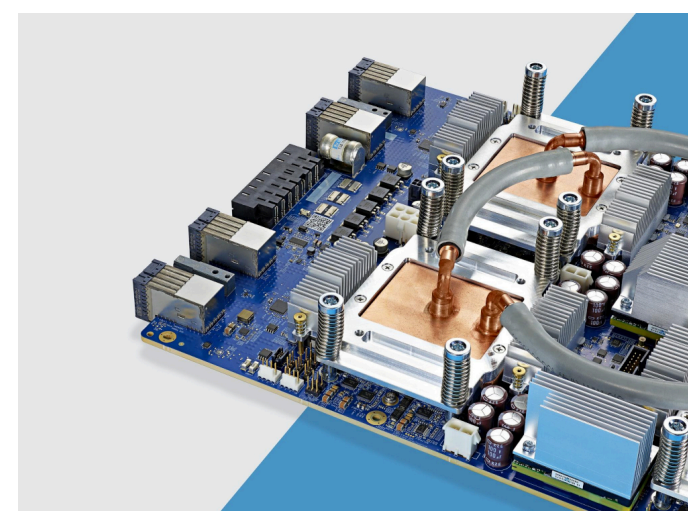
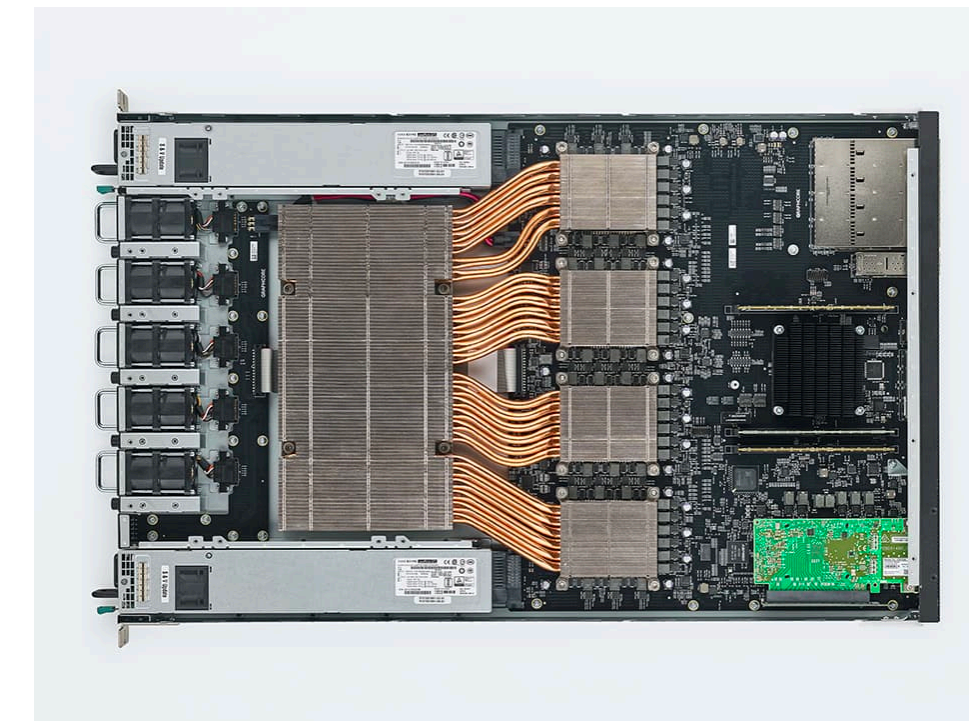
- Common FastML
- MLCommons recently added 'Tiny' category to MLPerf benchmark ([link](#))
- hls4ml submission targeted pynq-z2
- Fully on-chip hls4ml implementation is efficient for low power inference



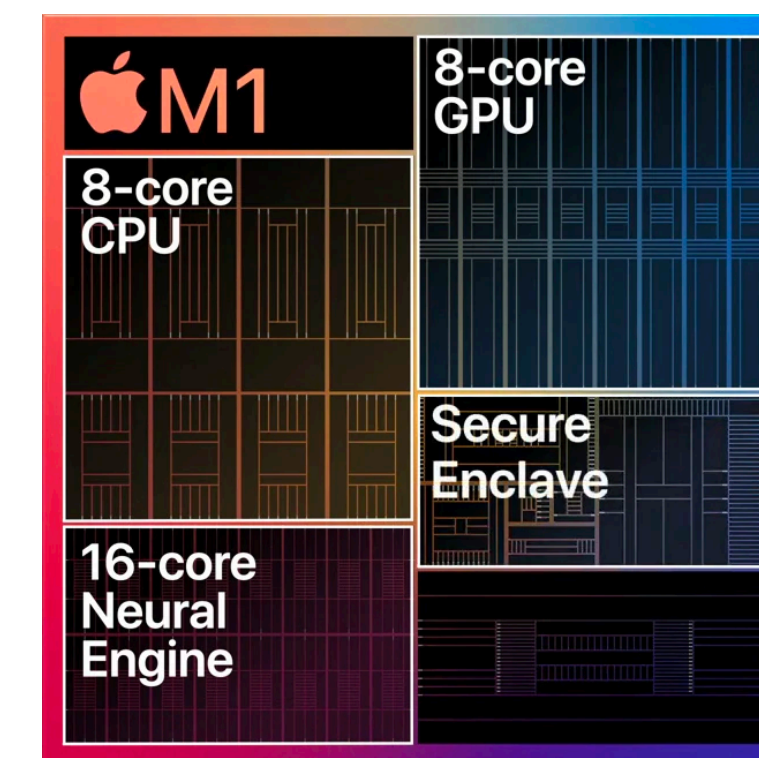
Benchmark		CIFAR-10			ToyADMOS		
Team	Device	Accuracy	Latency (ms)	Power (W)*	AUC	Latency (ms)	Power (W)*
<b>hls4ml</b>	Pynq-z2	77%	7.9	~ 1.5	0.82	0.096	~ 1.5
<b>Latent AI</b>	Raspberry Pi 4	85%	1.07	~ 4 - 5	0.85	0.17	~ 4 - 5
<b>Harvard</b>	Nucleo-L4R5ZI	85%	704		0.85	10.4	
<b>Peng Cheng Lab</b>	PCL Scepu02	85%	1239.16		0.85	13.65	

# BACKUP: ML Specific Processors

- There are some processors out there specifically designed for Machine Learning / AI
- e.g. Tensor Processing Unit (TPU) from Google, Intelligence Processing Unit (IPU) from Graphcore
- Devices aiming at low power embedded
  - Internet of Things, Smartphones
- Xilinx Versal ACAP for FPGAs with embedded Vector units, Vector/NN units in CPUs
- Many different things out there, each targeting a specific optimisation:
  - Best overall throughput
  - Lowest latency
  - Lowest power / smallest footprint
- Choose appropriate device for your task

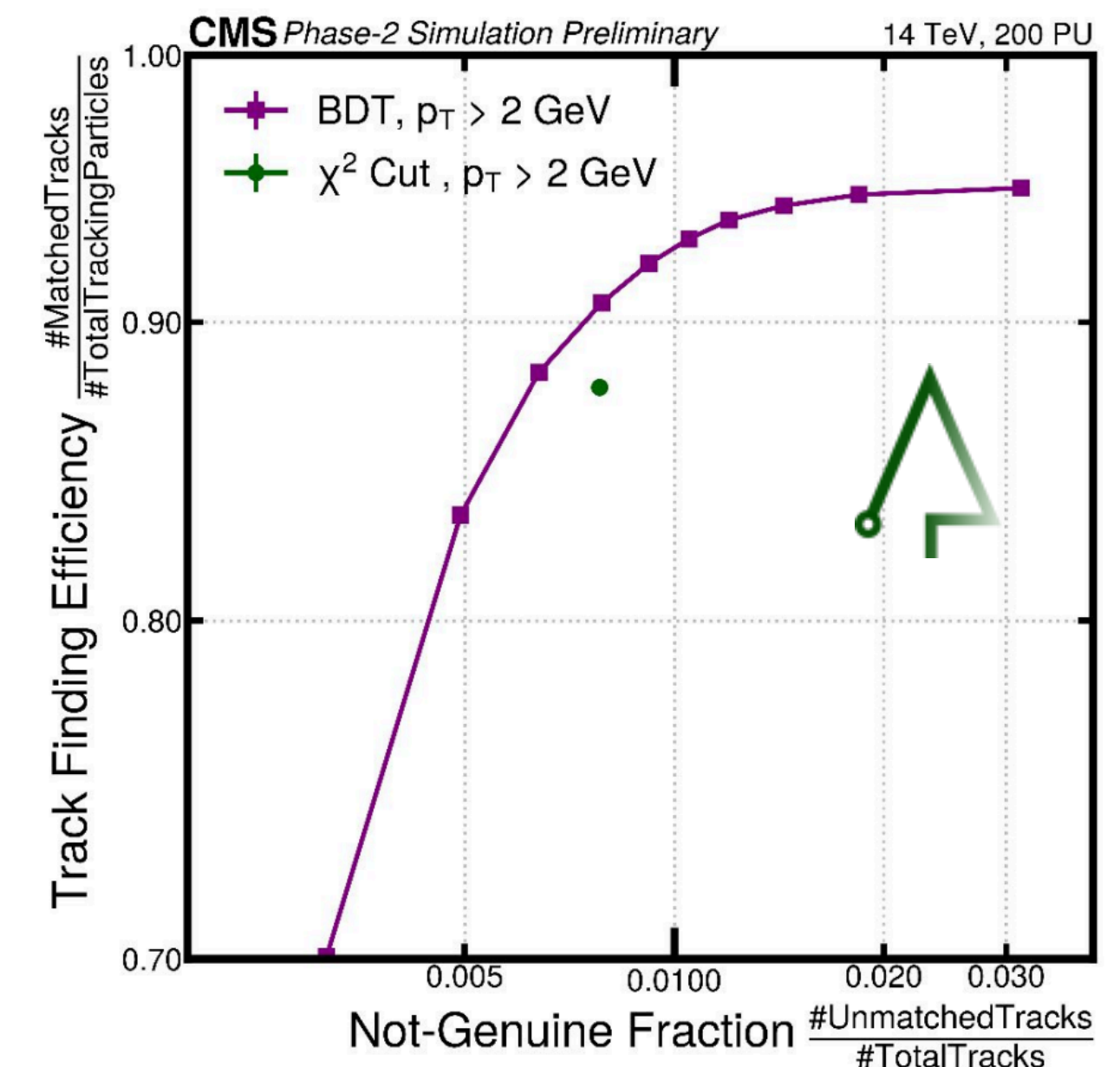


[A3D3](#)



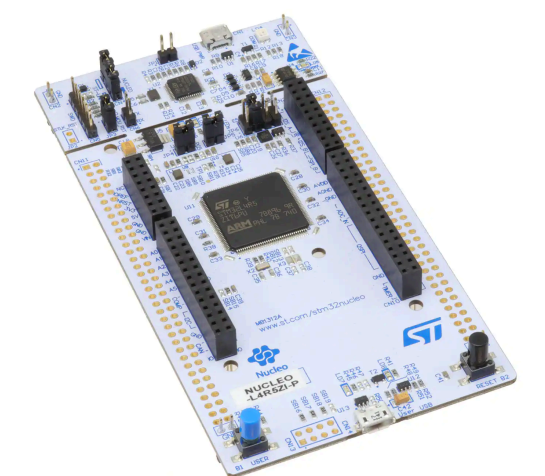
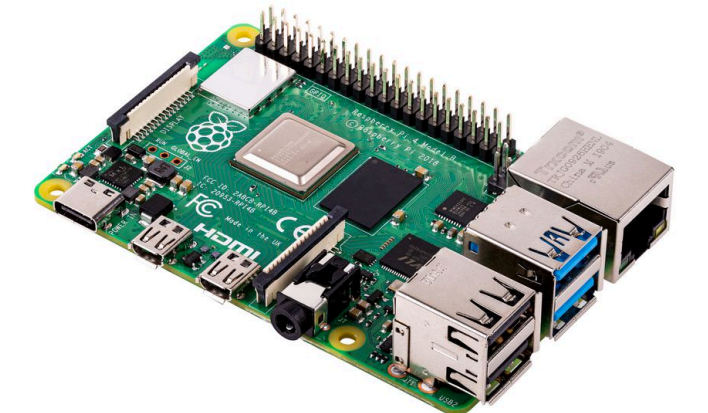
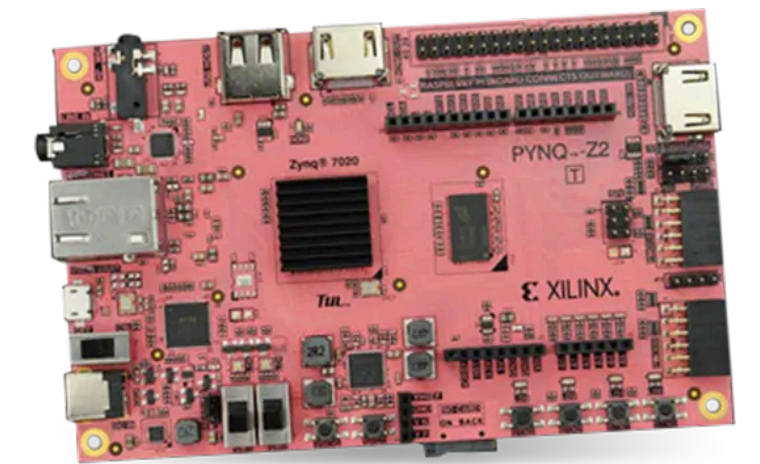
# ML in L1T FPGAs

- Tools like hls4ml (more later) and conifer bring ML into FPGAs with sub-microsecond latency
- Example: identifying fake tracks from CMS Level 1 Track Finder (Phase 2 Upgrade)
- Fake tracks are identified in simulation as those not associated to a simulated particle
  - Often from combinatorics (200 pileup scenario), they harm trigger performance later
- A BDT with 60 trees and depth of 3 finds fakes better than simple cuts
- conifer library maps BDT onto FPGA logic
  - In this case 33 ns latency and < 1% resources (VU9P)
- Many algorithms in development for Phase 2
  - Improving object reconstruction (as here)
  - Improving event selection of difficult signatures



# Fast ML benchmarking

- Common FastML
- MLCommons recently added 'Tiny' category to MLPerf benchmark ([link](#))
- hls4ml submission targeted pynq-z2
- Fully on-chip hls4ml implementation is efficient for low power inference

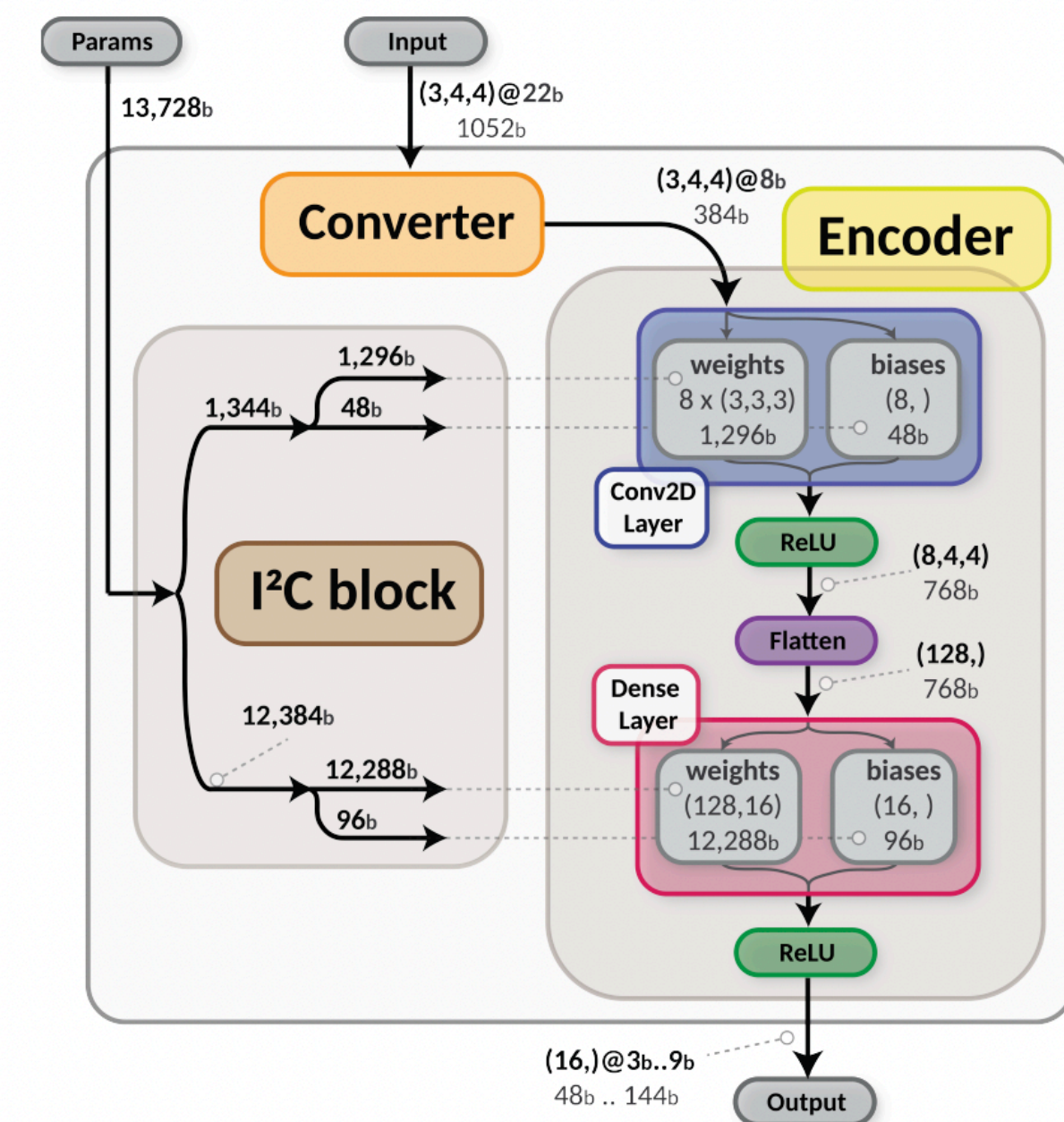
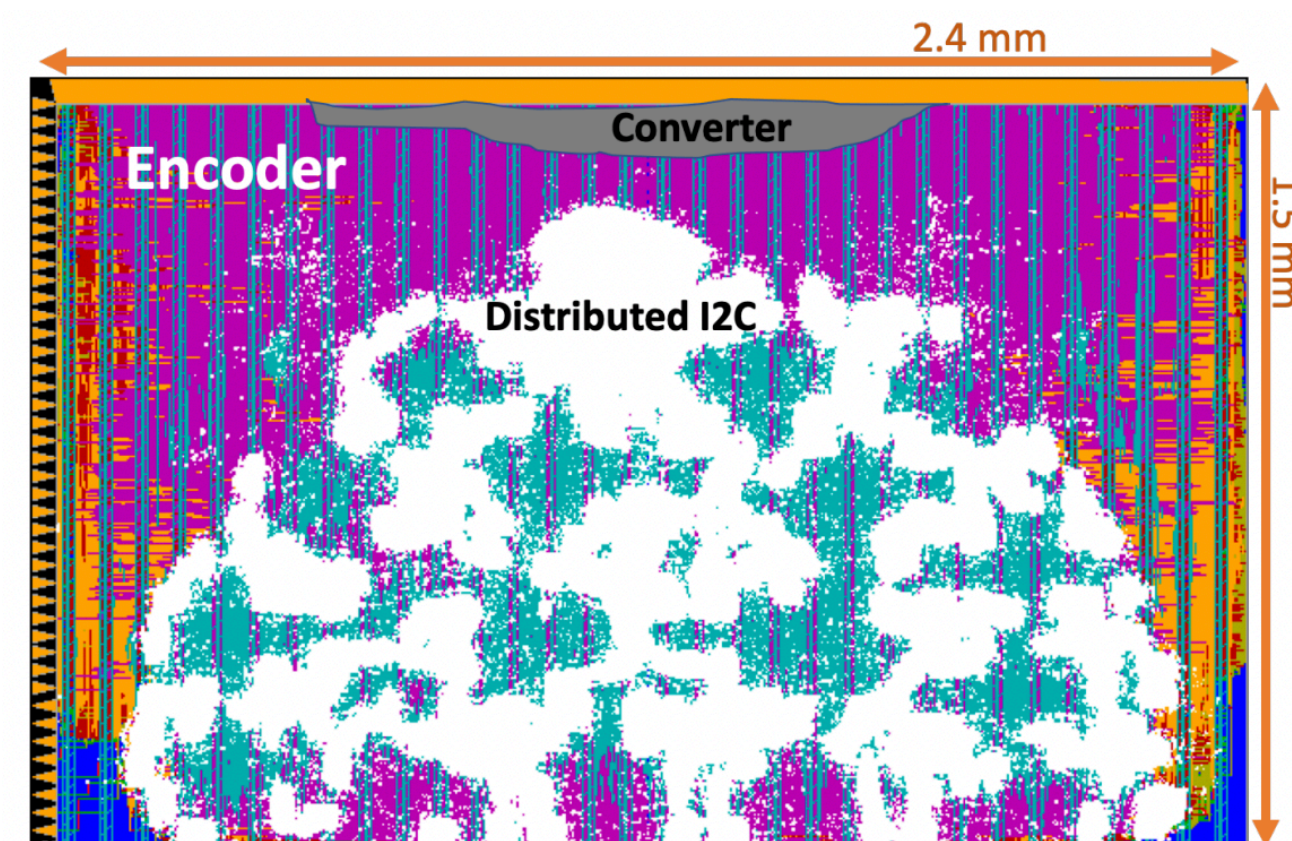
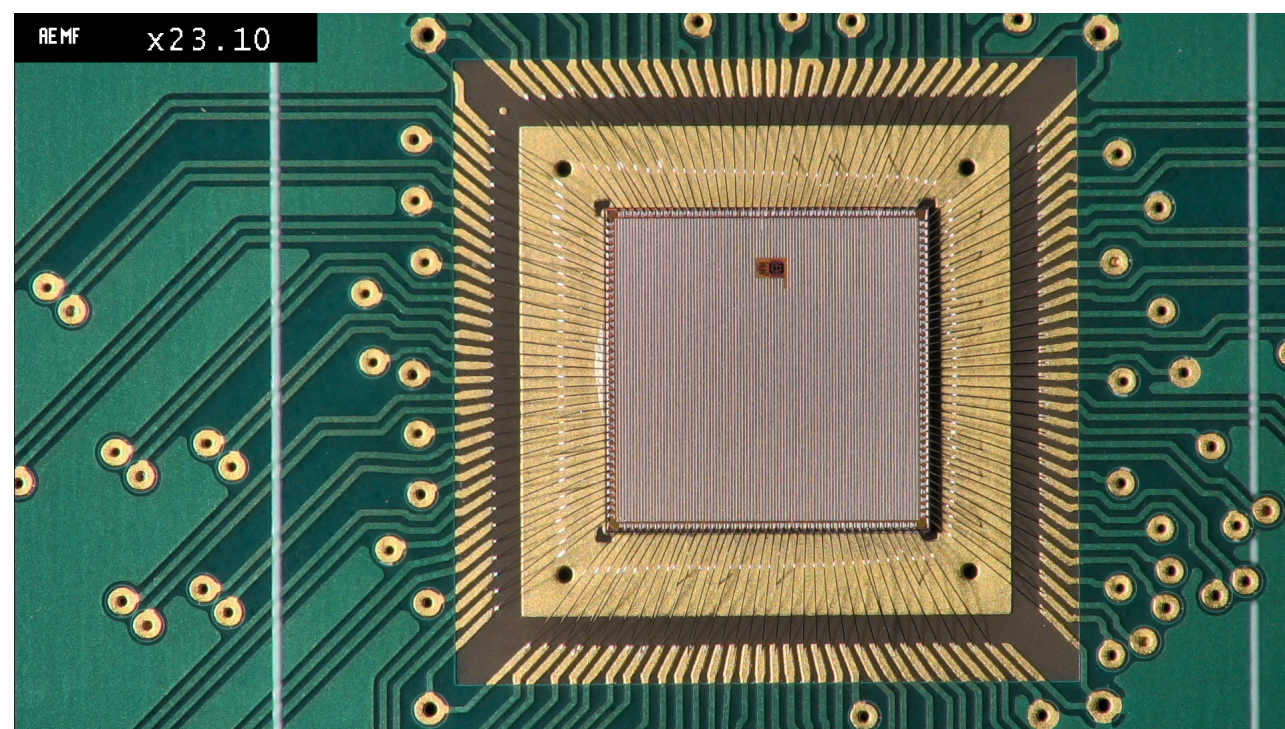


Benchmark		CIFAR-10			ToyADMOS		
Team	Device	Accuracy	Latency (ms)	Power (W)*	AUC	Latency (ms)	Power (W)*
<b>hls4ml</b>	Pynq-z2	77%	7.9	~ 1.5	0.82	0.096	~ 1.5
<b>Latent AI</b>	Raspberry Pi 4	85%	1.07	~ 4 - 5	0.85	0.17	~ 4 - 5
<b>Harvard</b>	Nucleo-L4R5ZI	85%	704		0.85	10.4	
<b>Peng Cheng Lab</b>	PCL Scep02	85%	1239.16		0.85	13.65	



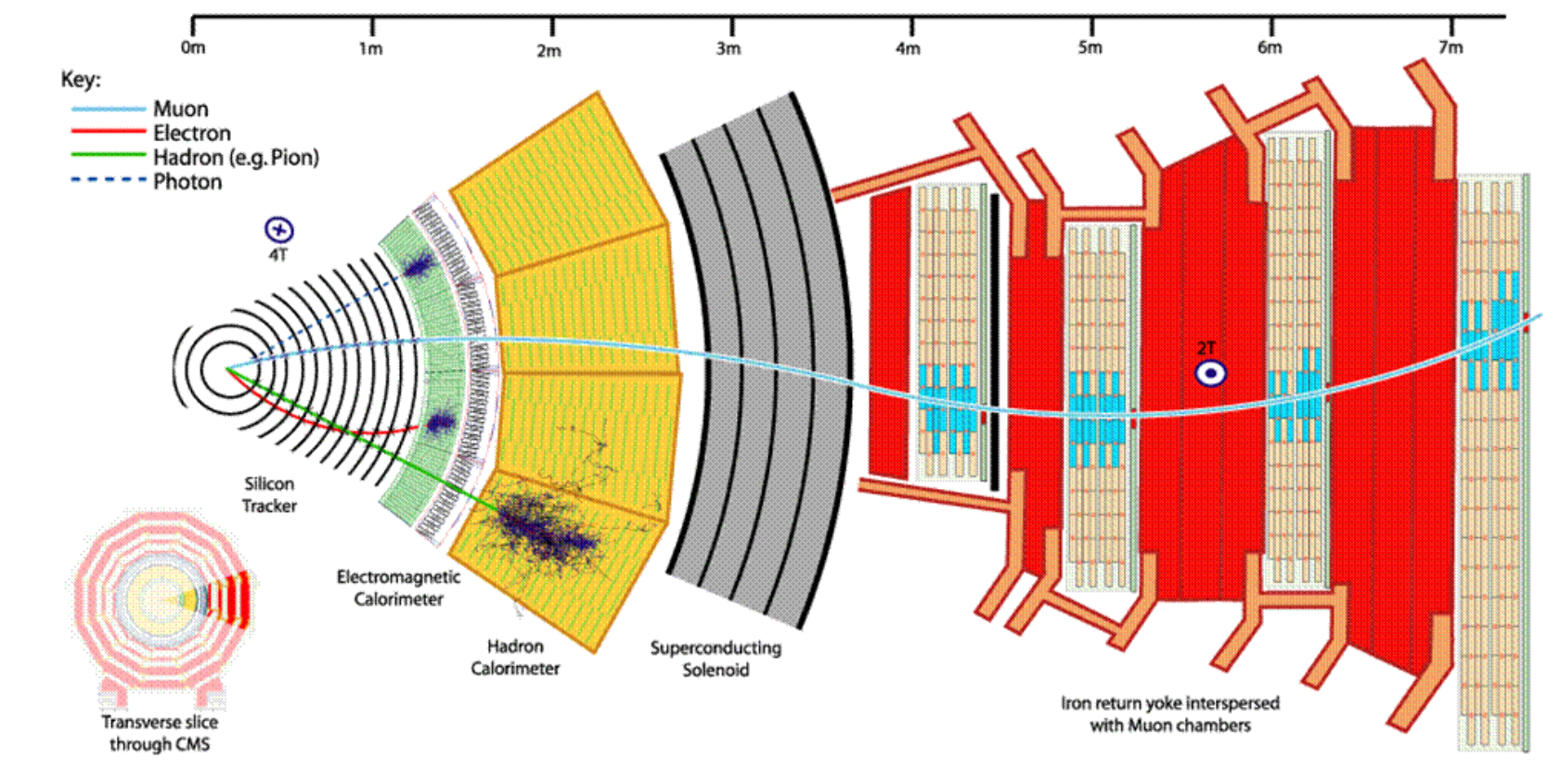
# ECON-T ASIC for CMS HGCal

- Neural Net encoder IP block created for ECON-T ASIC with Catapult HLS (Mentor/Siemens) and hls4ml (more later)
  - NN architecture is fixed, weights can be reprogrammed (e.g. after NN retraining)
  - ECON-T also includes non-ML baseline compression algorithms
- Decoder block would run in trigger FPGAs
- Device manufactured and undergoing testing



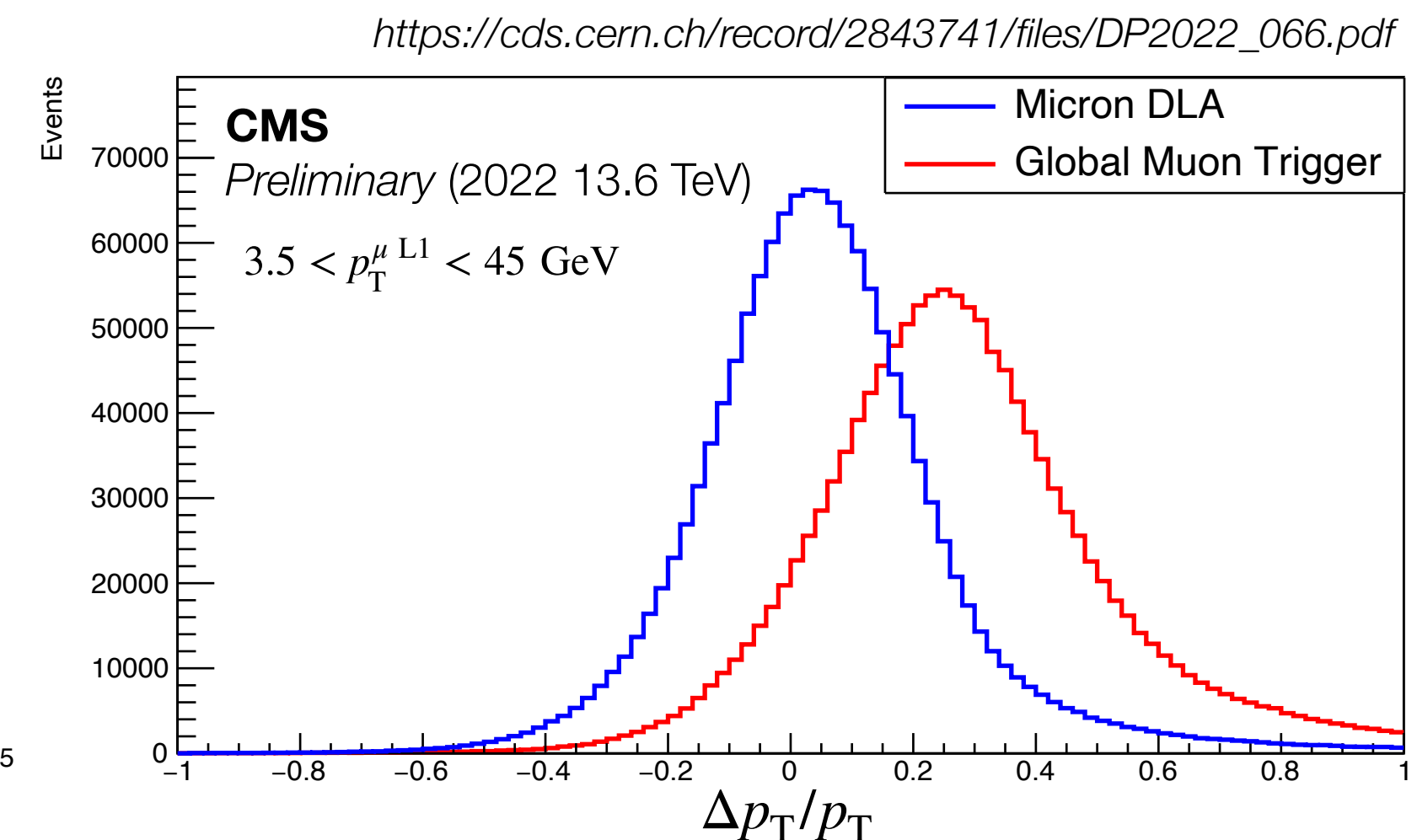
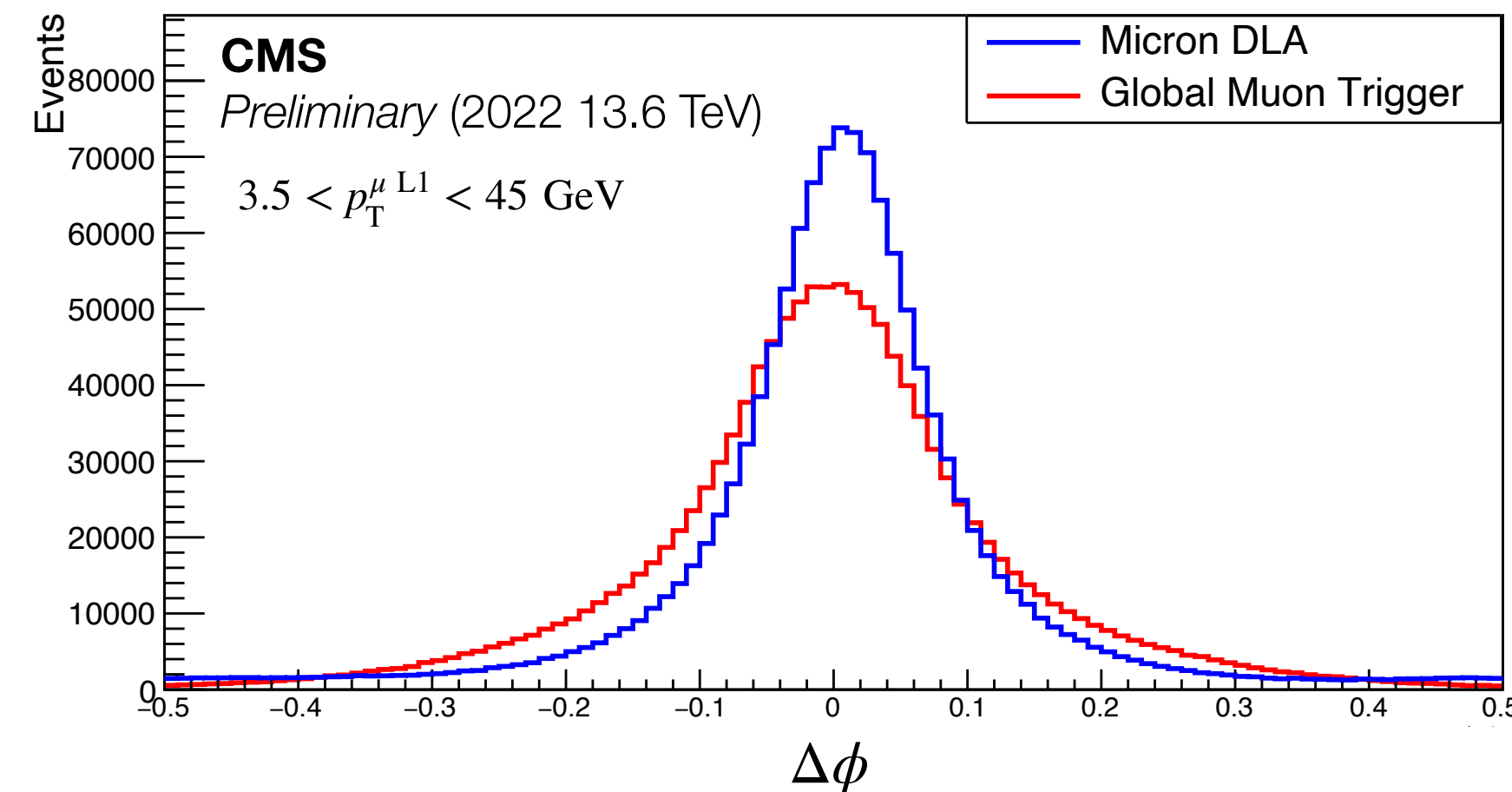
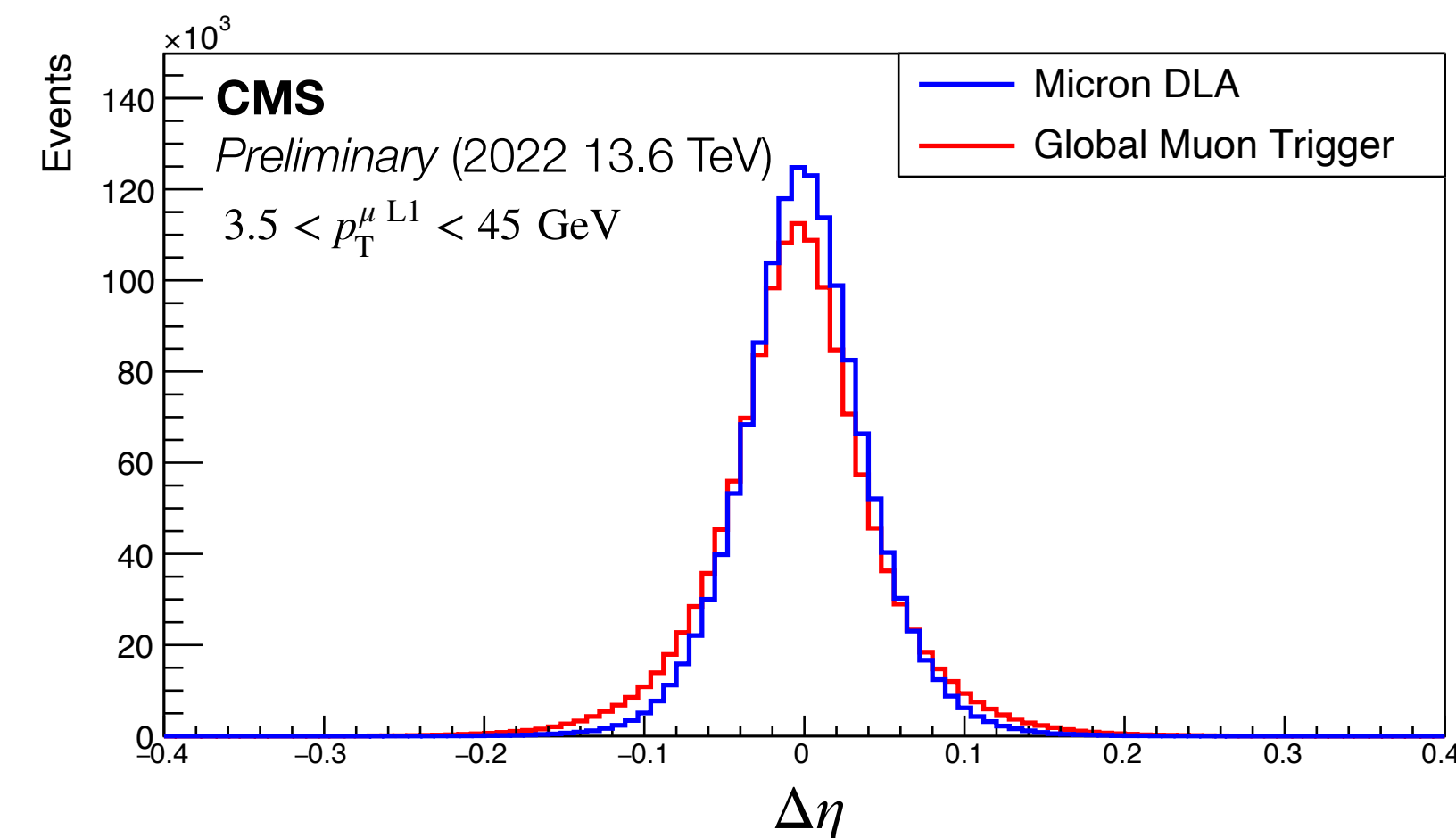
# L1 Scouting

- Stream processor / accelerator hybrid
- What does L1 accept miss?
- Can we acquire L1 trigger data at full bunch crossing rate
  - subset of detector information, limited resolution
- Allows for analysis of certain topologies at full rate
  - semi real-time analysis and/or
  - storing of tiny event record
- Demonstrated for first time at end of 2018
- Upgraded w/ new boards in 2021 - validated with LHC test beams
- For LHC Run 3 (2022) - prompt & displaced muons, jets, electrons/photons, taus and global trigger outputs included



# CMS L1 scouting: muon re-calibration

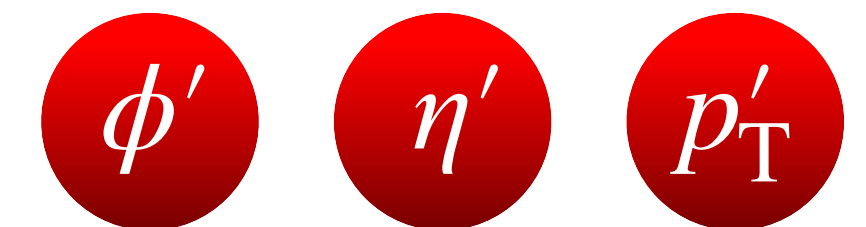
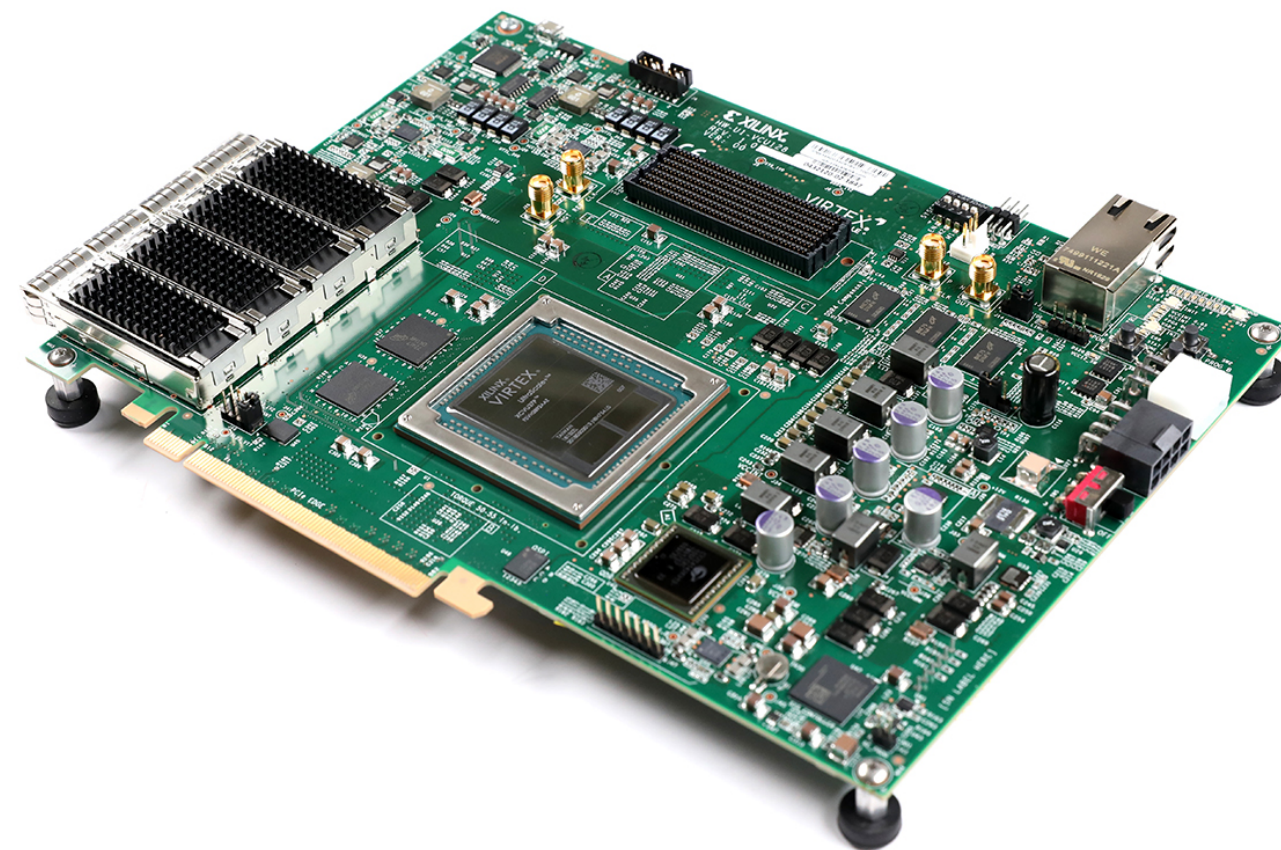
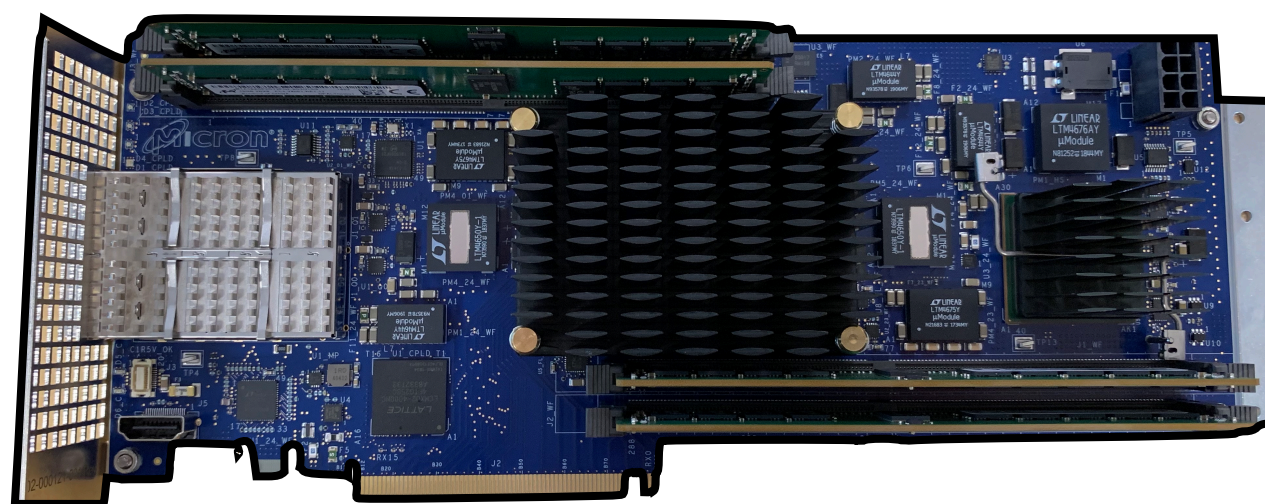
- › L1 scouting at CMS acquires subset of granular data at full bunch crossing rate - what does L1 accept miss?
- › Use ML to recover resolution / accuracy in comparison with full reconstruction
- ›  $\Delta\eta$ ,  $\Delta\phi$ ,  $\Delta p_T$  is the difference between the prediction (or GMT) values, and the offline reconstructed global muon tracks for matched muons ( $\Delta R < 0.1$  at 2nd muon station)



- › Target: centred on zero & higher peak
- › NN shows improvement for all variables in comparison to raw trigger values

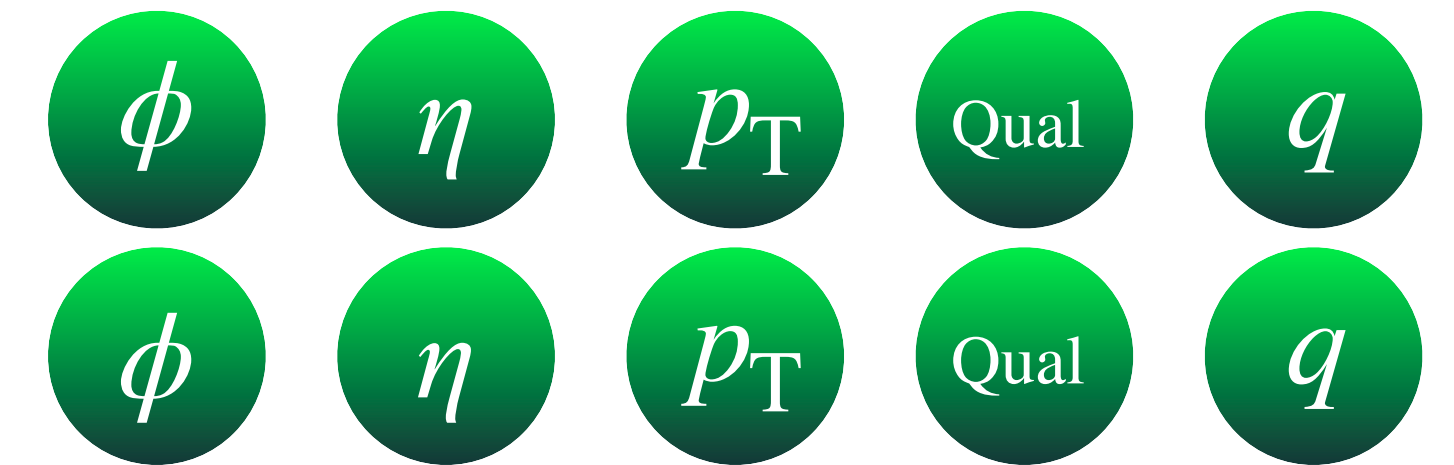
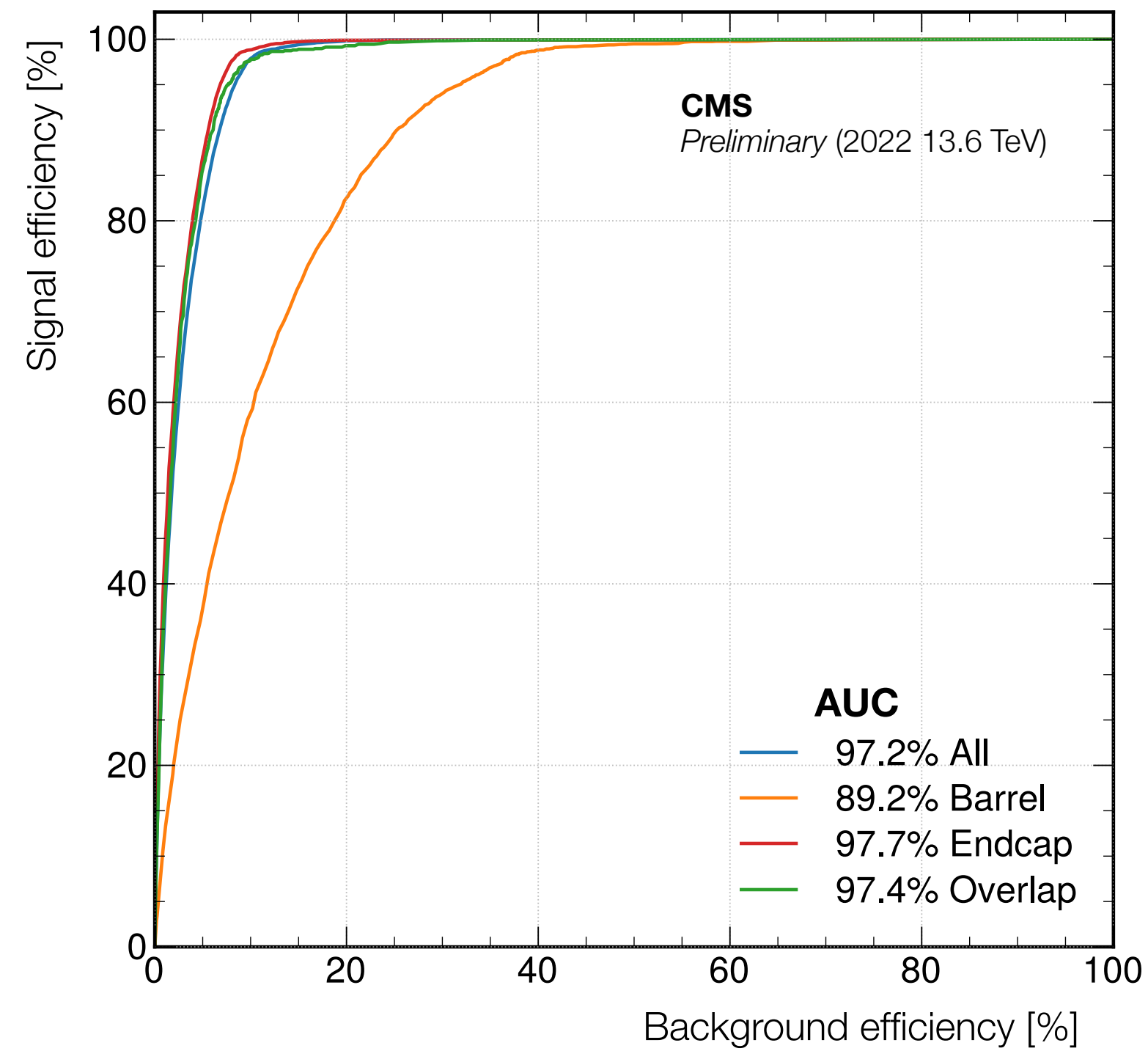
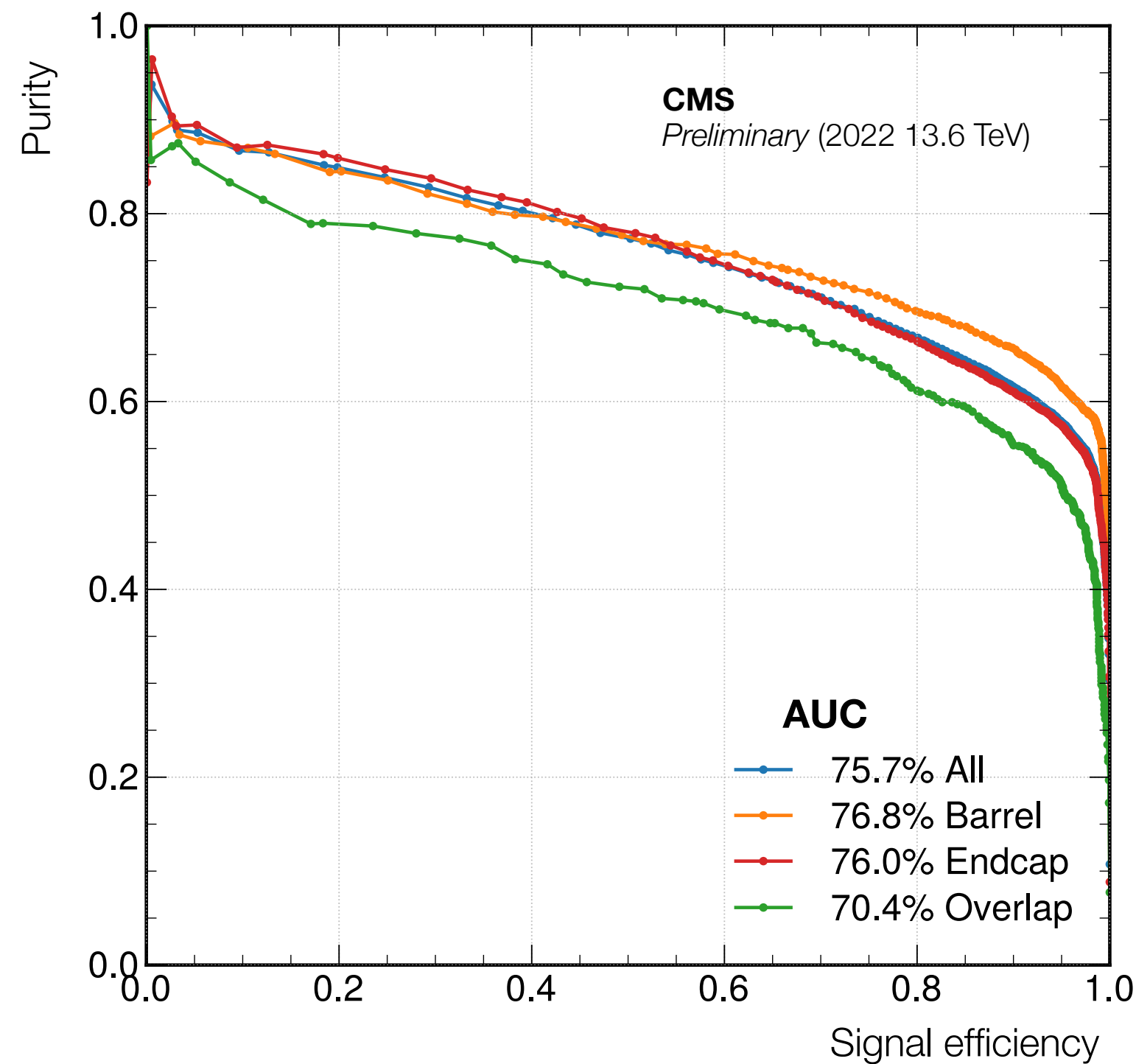
# CMS L1 scouting: muon re-calibration

- Integrated NN for muon recalibration generated w/ HLS4ML
- Runs on Micron SB852 & Xilinx VCU128 boards
- Q6.12 precision, pruning factor 0.5
- 2 NN each process 4 muons / BX
- Latency  $\lesssim$  100 ns FIFO latency, can accept 2 muons / clock



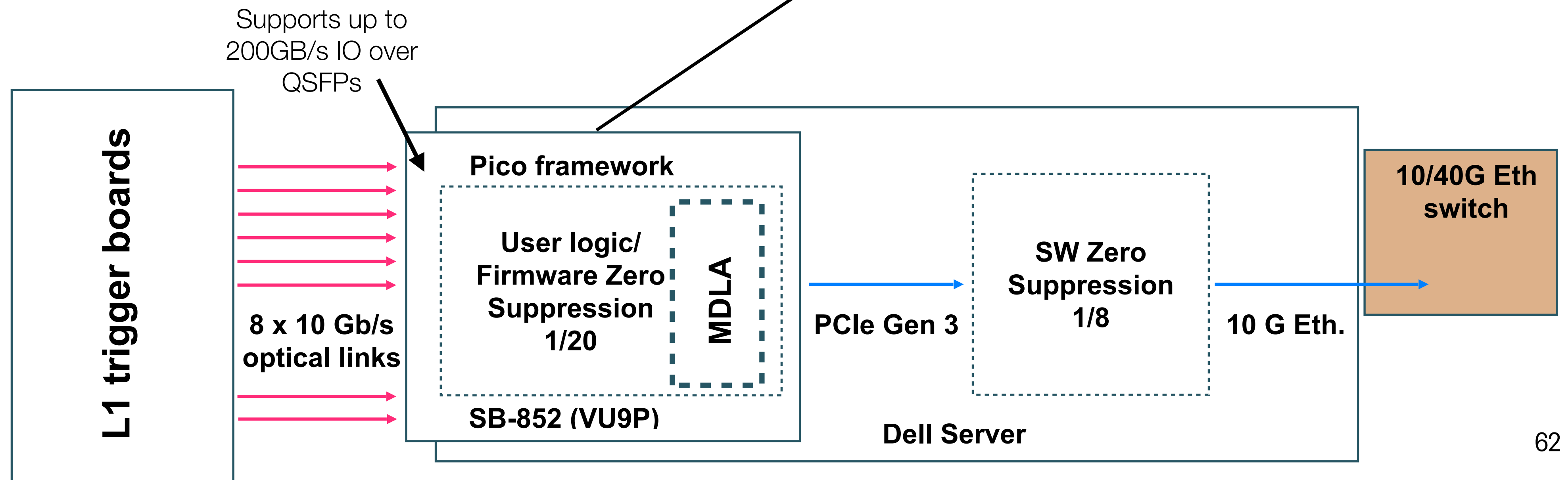
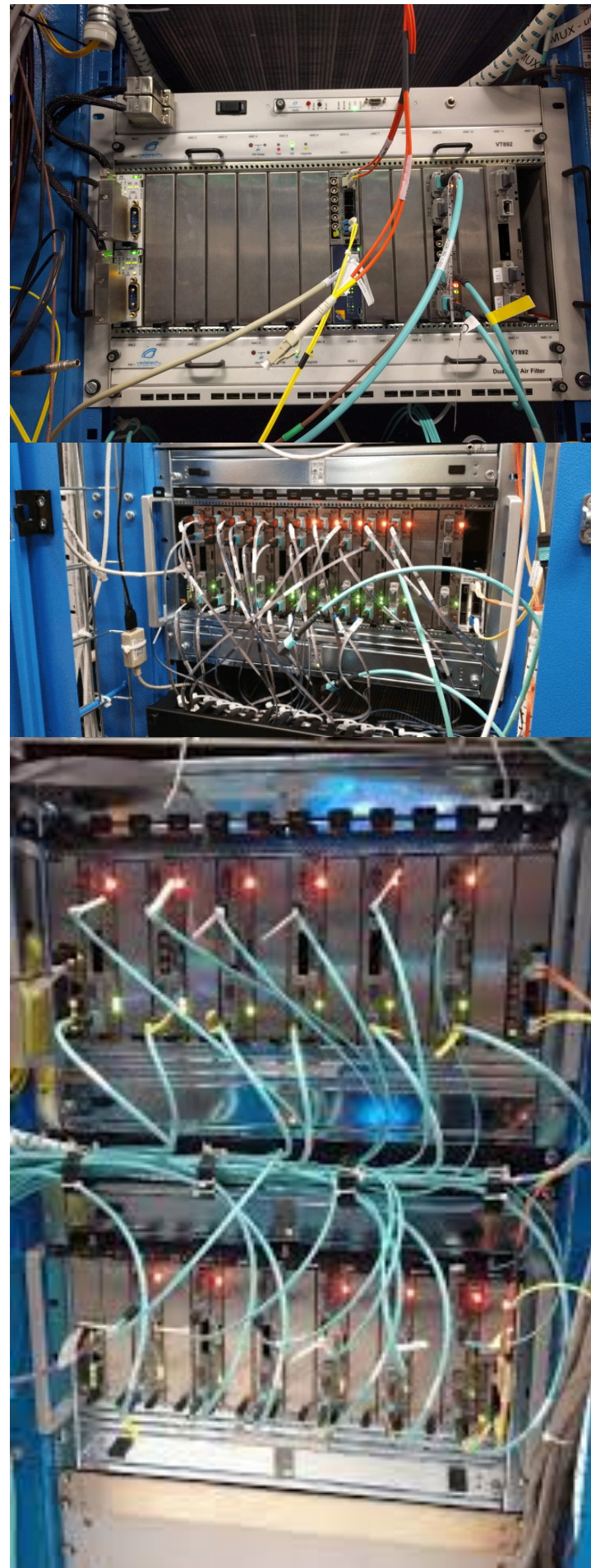
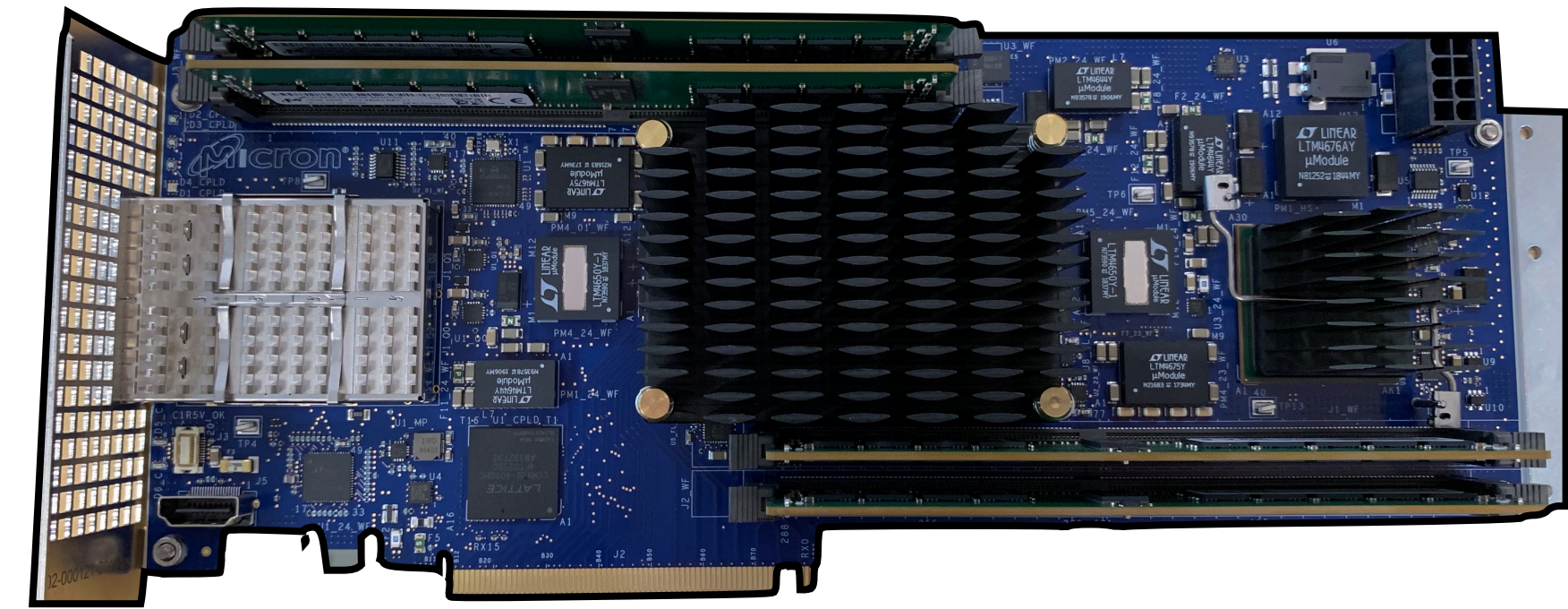
# L1 dimuon classifier

- › Acts on muon pairs: rejects misconstructured pairs of L1 muons not found in offline reco i.e L1 duplicates
- › Trained/tested with 2022 data - 305k pairs (identical pre-removed)
- › Huge gains in purity for small efficiency cost



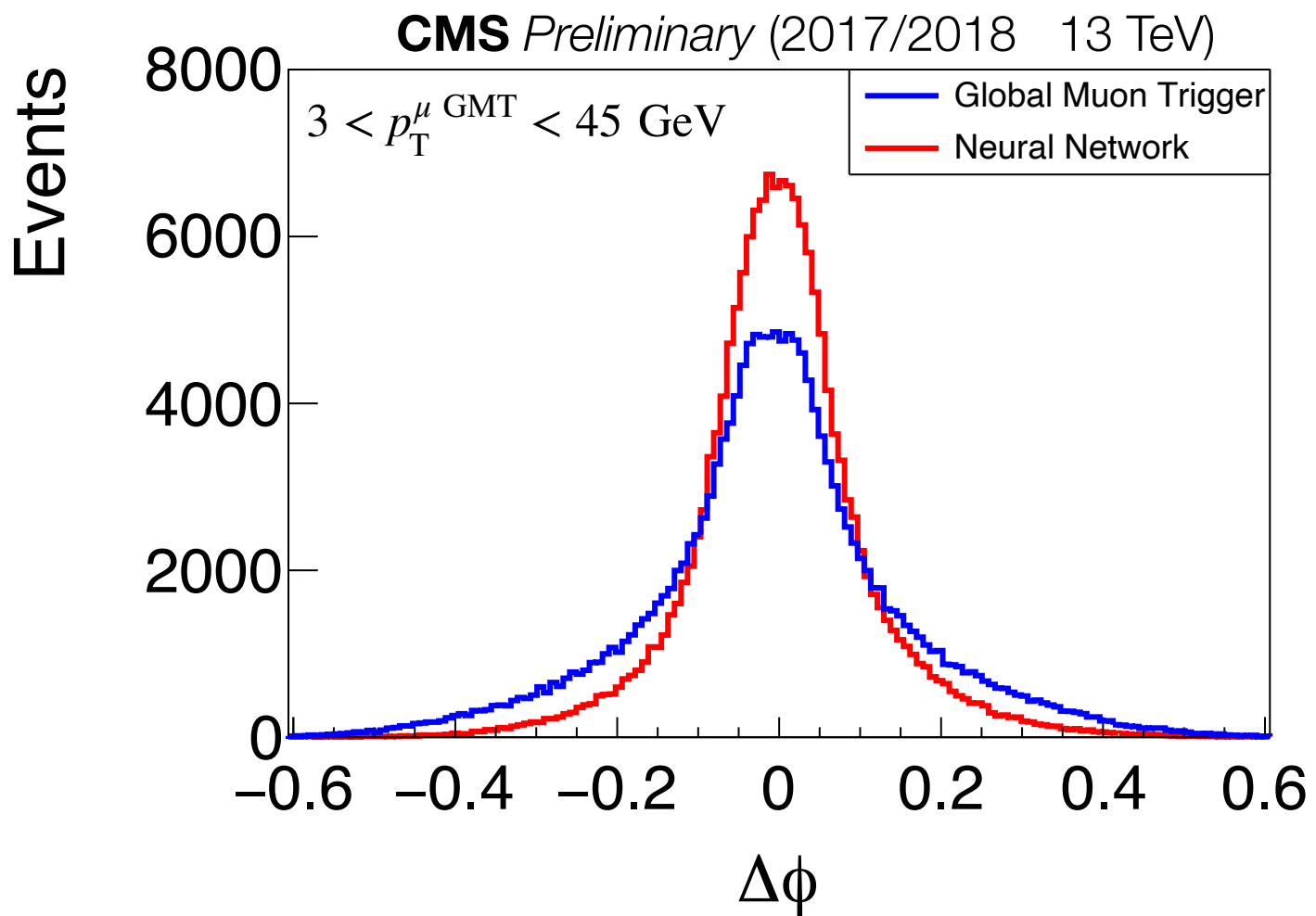
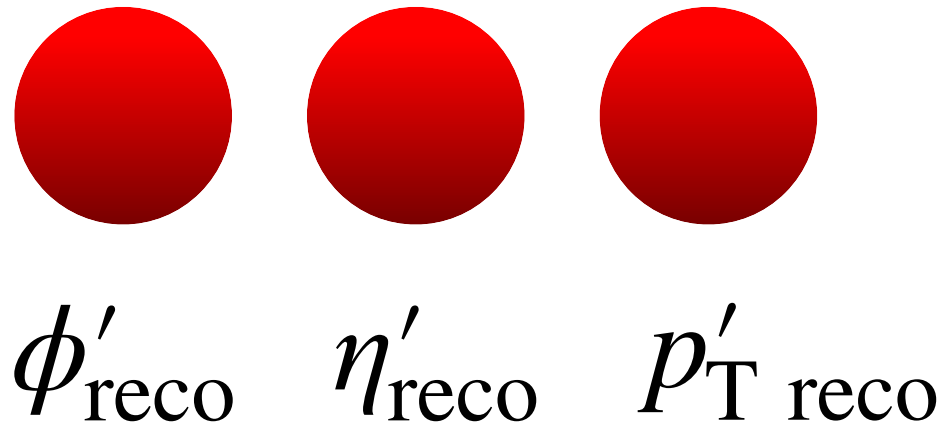
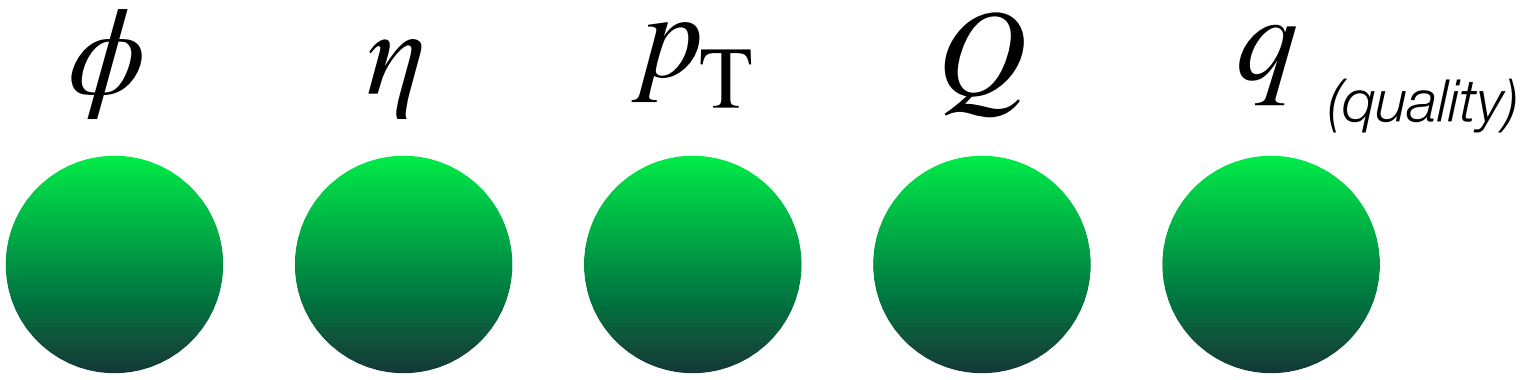
# L1 Scouting with SB-852

- Micron SB-852 for optical input -> DMA to PC
- Perform NN inference with Micron DLA after firmware ZS

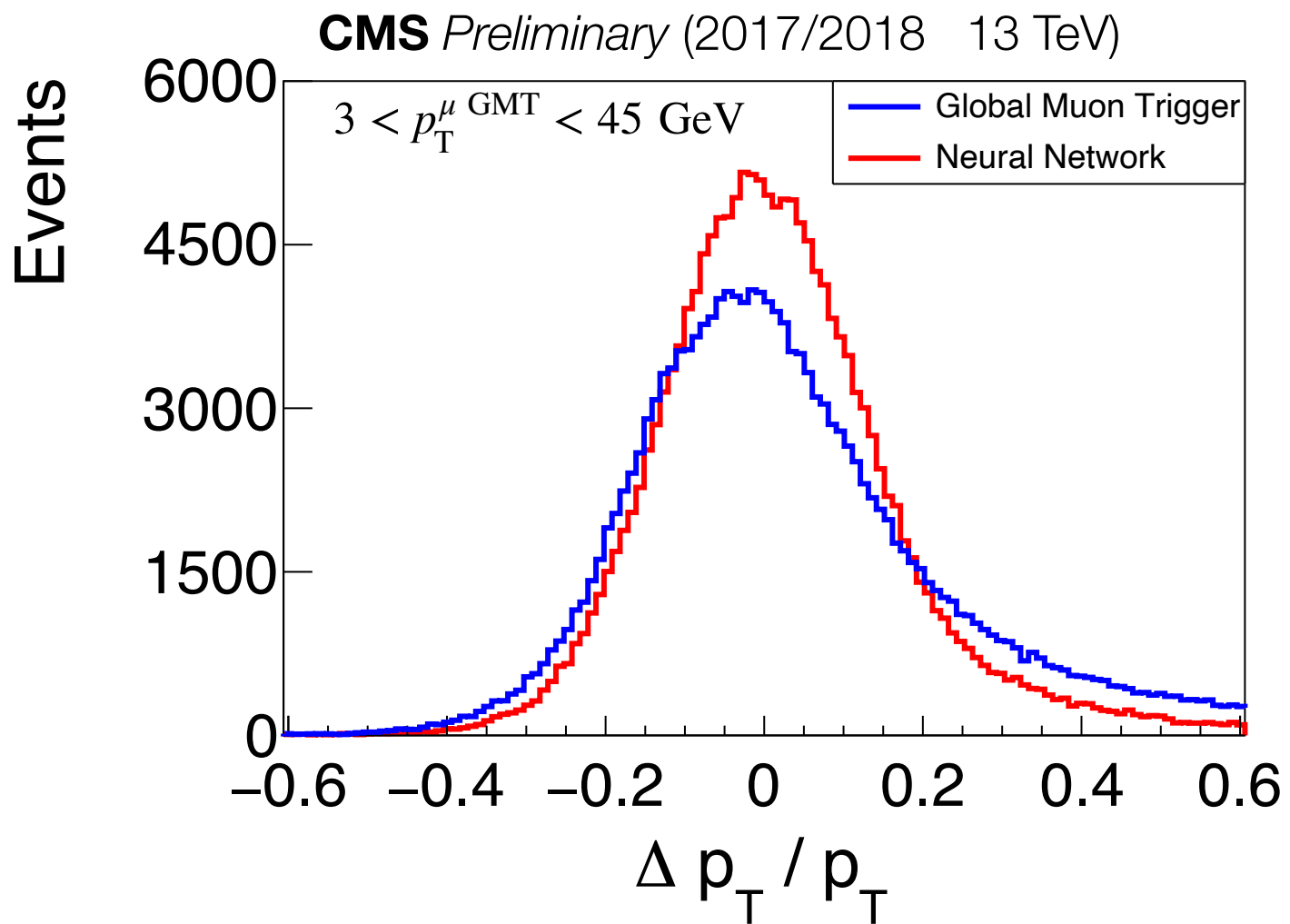


# Why ML for L1 scouting?

- Use of classical (FF-DNN) neural networks to ‘recalibrate’ L1 information to improve their utility for an online analysis
- Inputs - L1 objects e.g GMT muons:
- Target - Offline fully reconstructed objects



Particle angular position perpendicular to beam



Particle momentum in direction transverse to beam

# Muon recalibration on SB-852

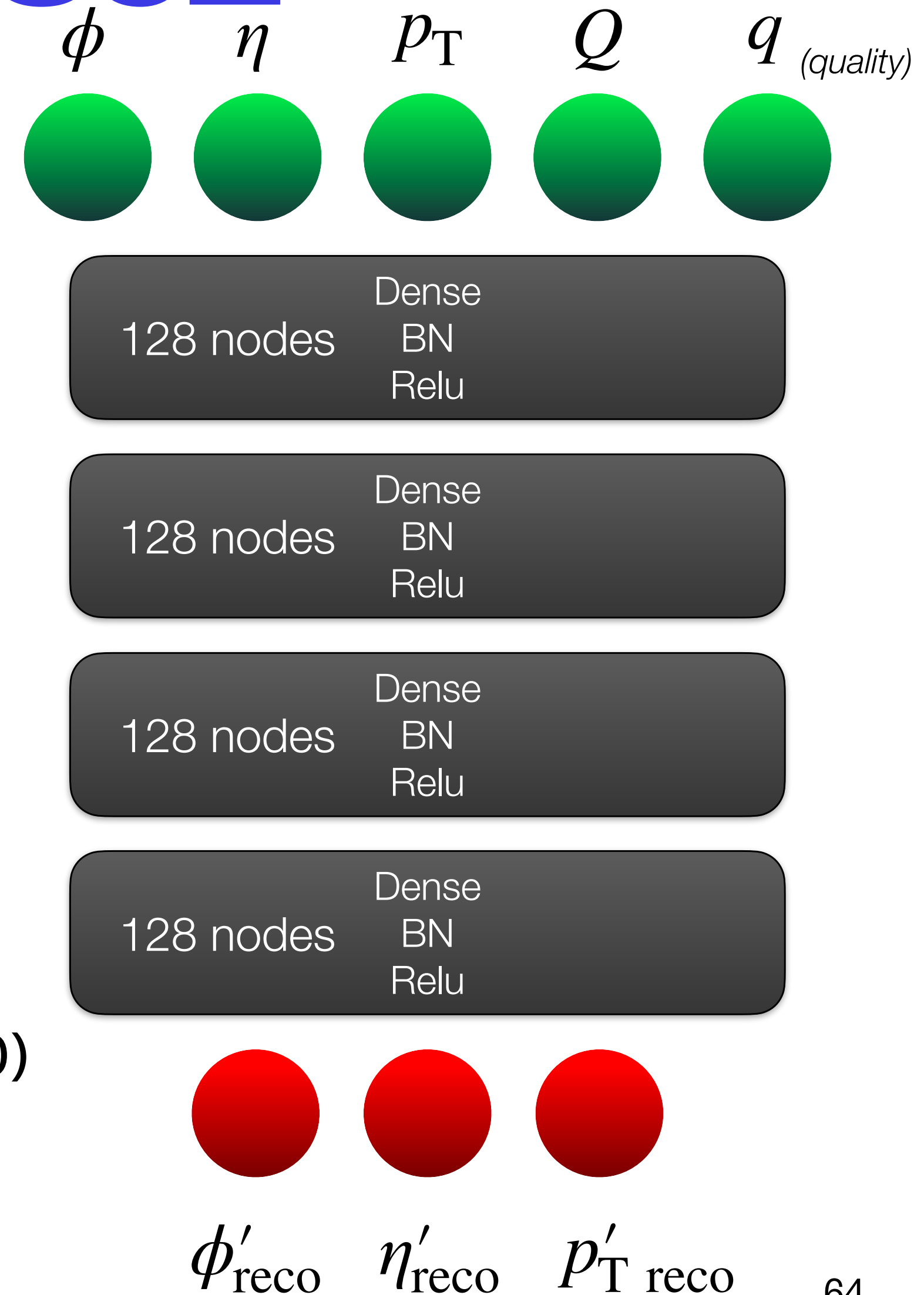
N DLA clusters	Inference rate	Average latency / muon inference
4 cluster	5.6 MHz	171 ns
2 cluster	2.8 MHz	342 ns
1 cluster	1.4 MHz	683 ns

- 4 clusters maximum in VU9P FPGA
- Majority of latency from data/weights transfer RAM/FPGA, batching implemented to remove this bottleneck (batch size 1280)

Precision |hw - Keras sw|      Frac. < 1% diff

Model w/ integer inputs, no batch norm

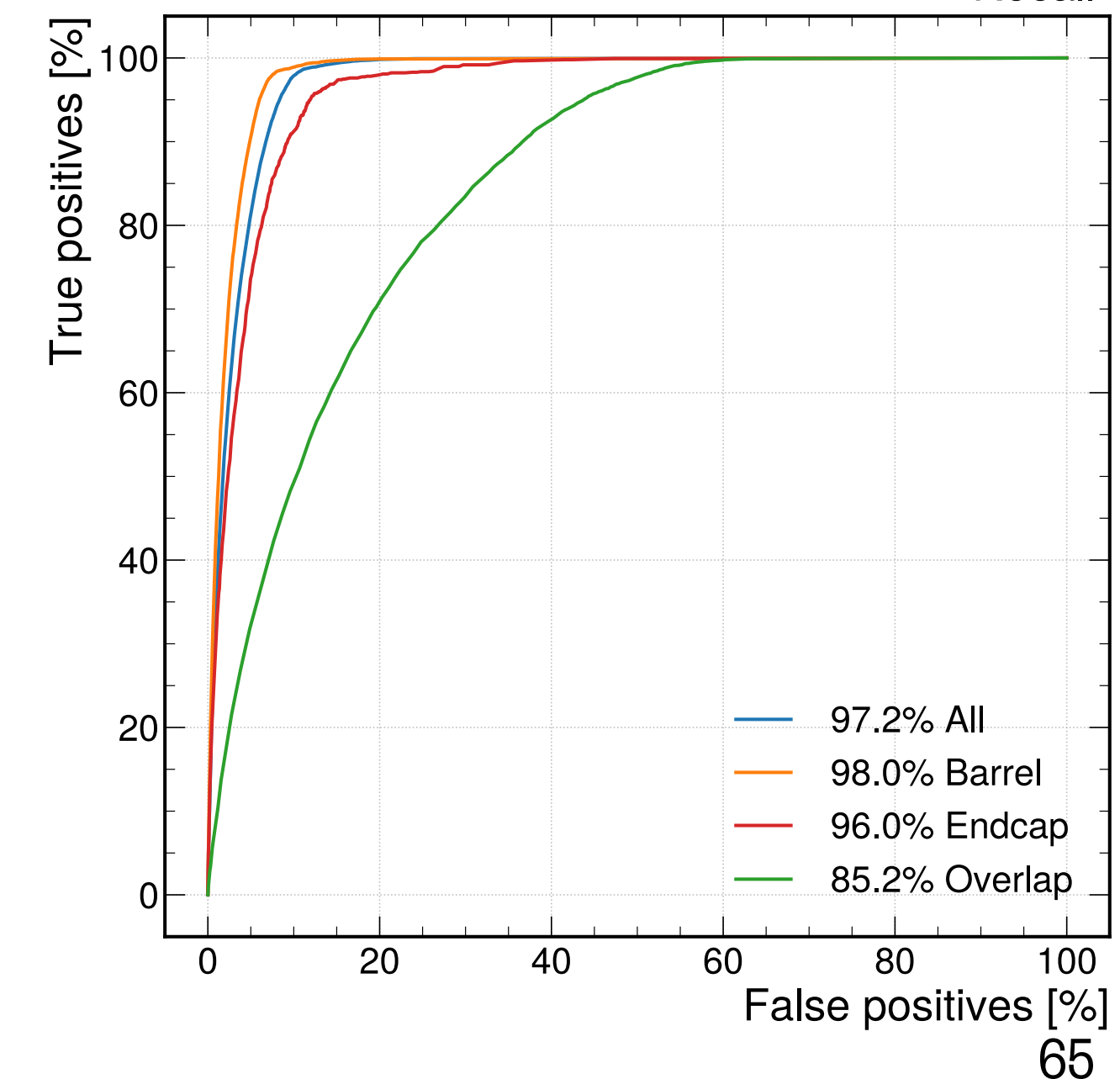
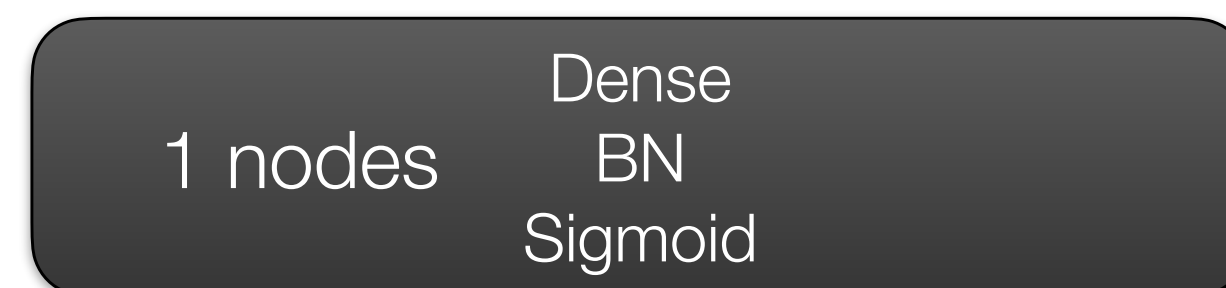
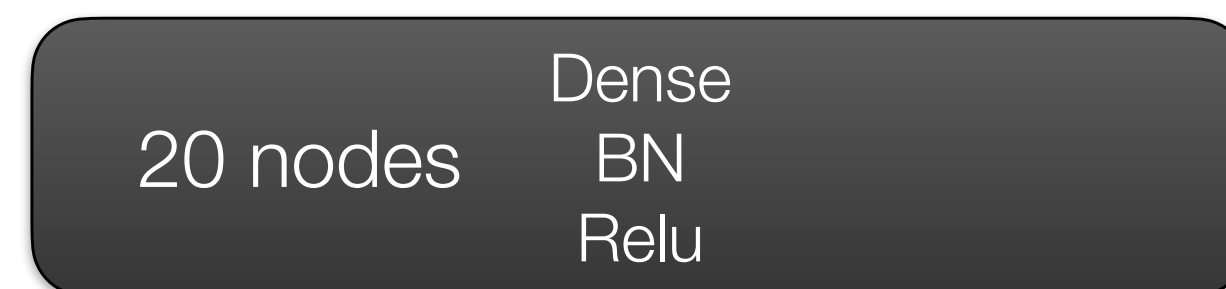
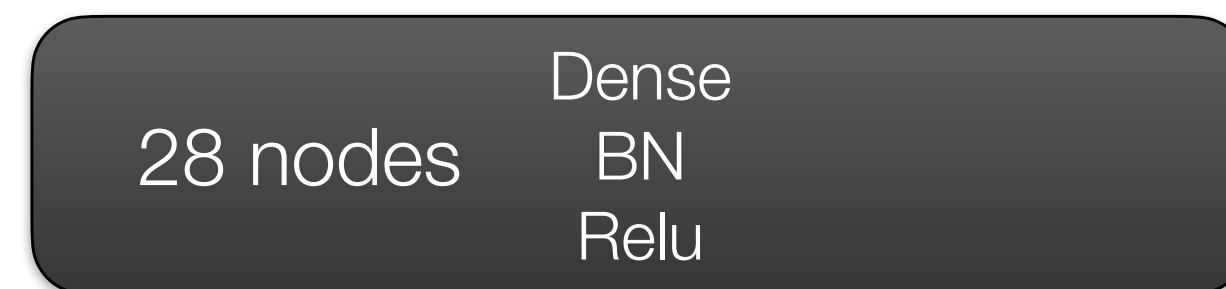
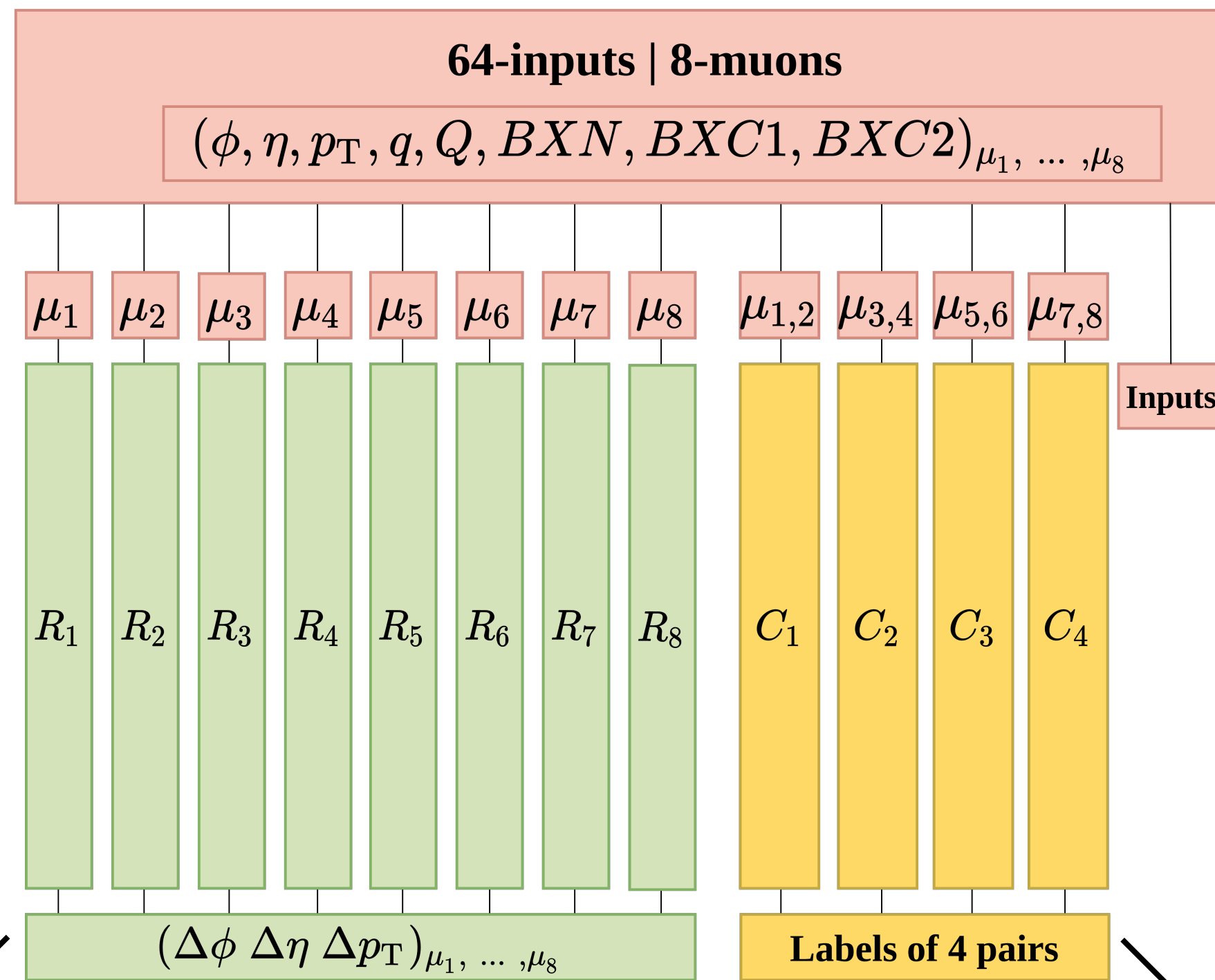
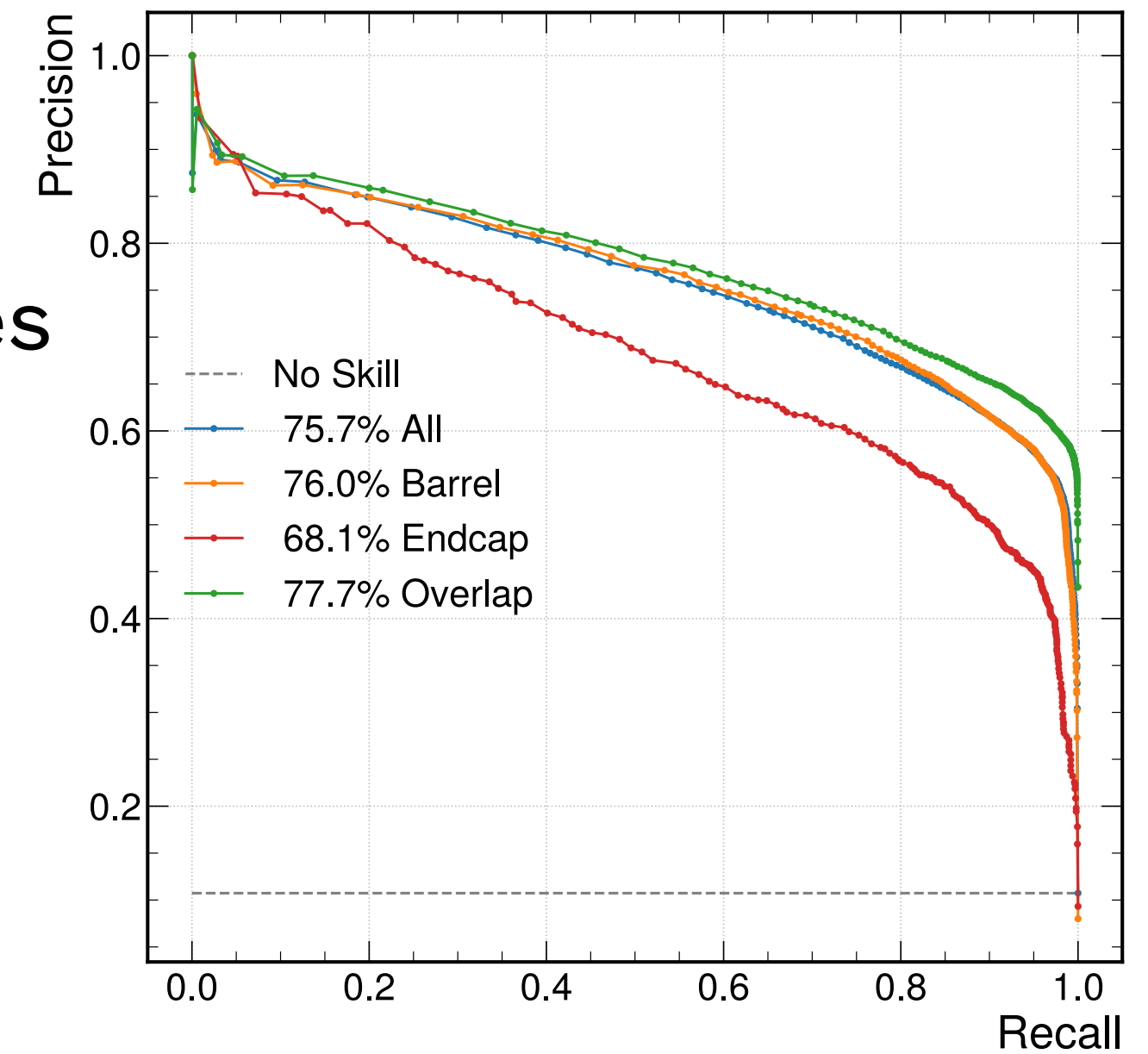
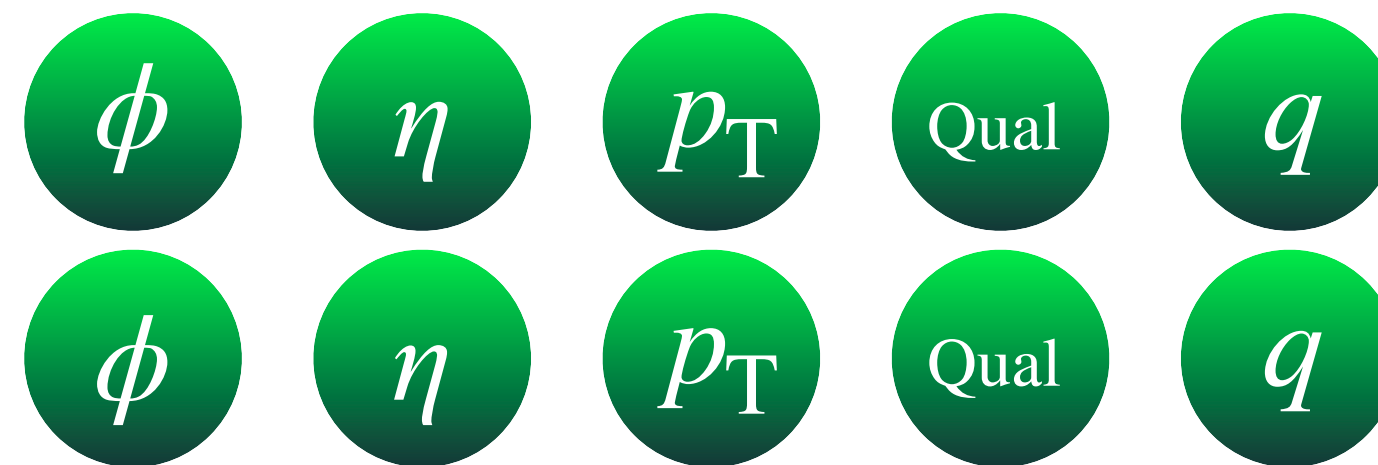
99%





# Fake muon pair classifier

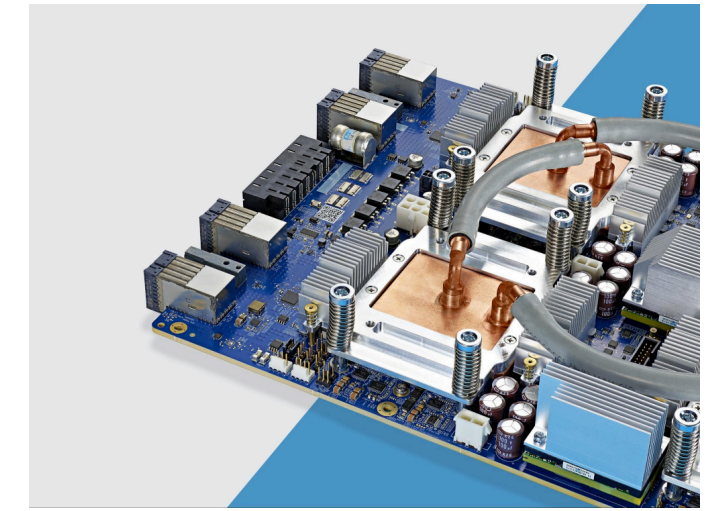
- Network consists of 8 recalibration branches & 4 classification branches
- Trained/tested with Run 3 Zero-bias data



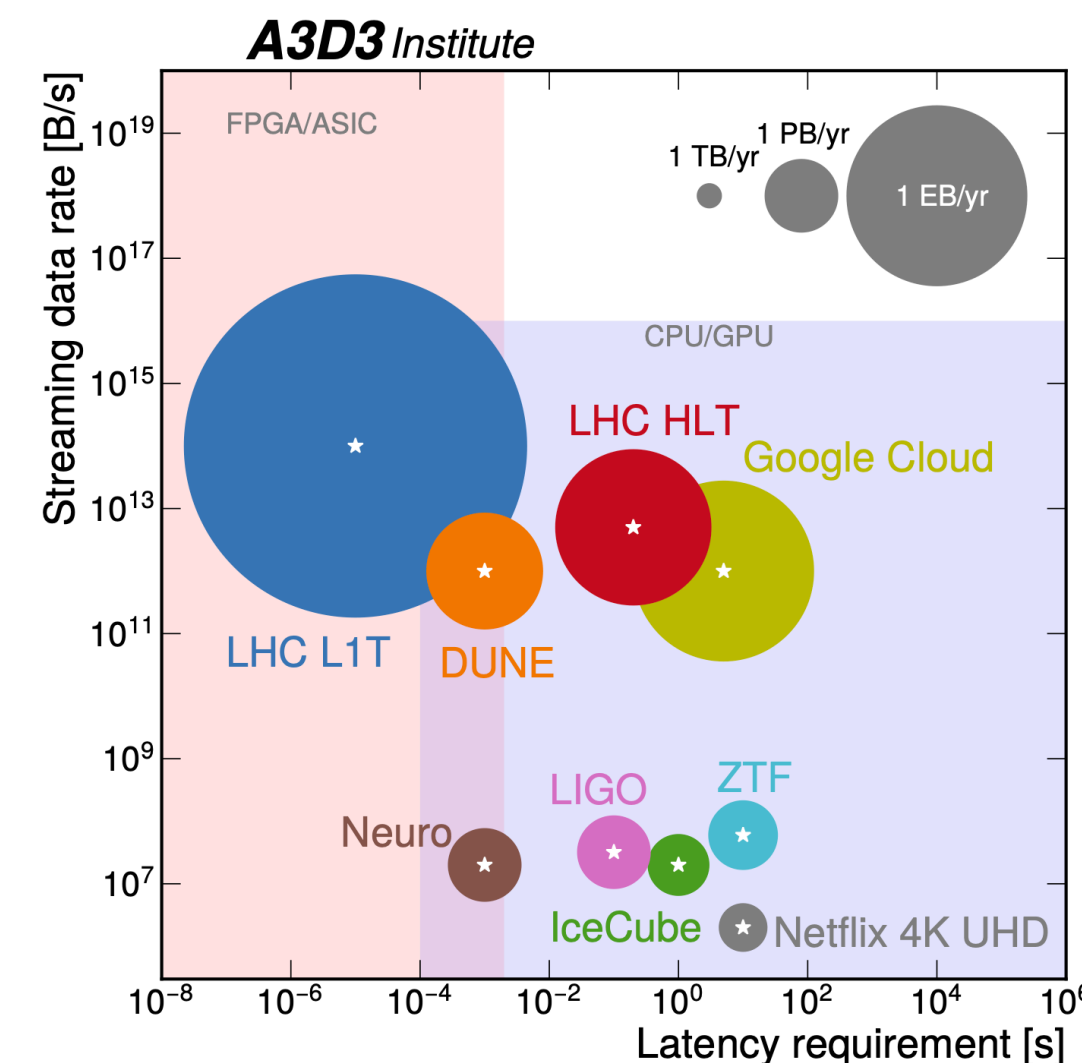
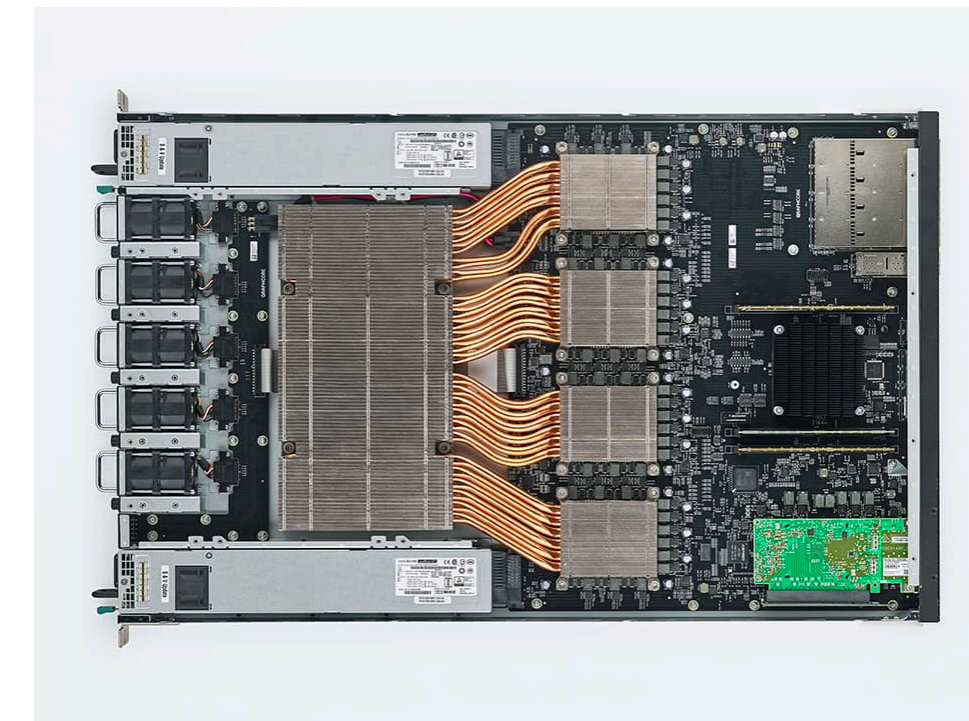
Recalibration

Classification

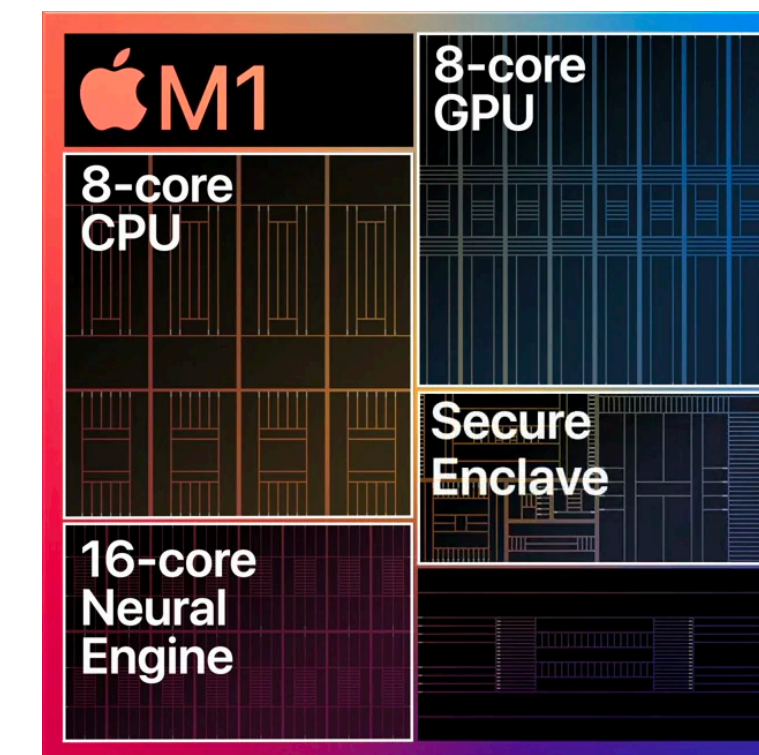
# ML Specific Processors



- There are some processors out there specifically designed for Machine Learning / AI
- e.g. Tensor Processing Unit (TPU) from Google, Intelligence Processing Unit (IPU) from Graphcore
- Devices aiming at low power embedded
  - Internet of Things, Smartphones
- Xilinx Versal ACAP for FPGAs with embedded Vector units, Vector/NN units in CPUs
- Many different things out there, each targeting a specific optimisation:
  - Best overall throughput
  - Lowest latency
  - Lowest power / smallest footprint
- Choose appropriate device for your task

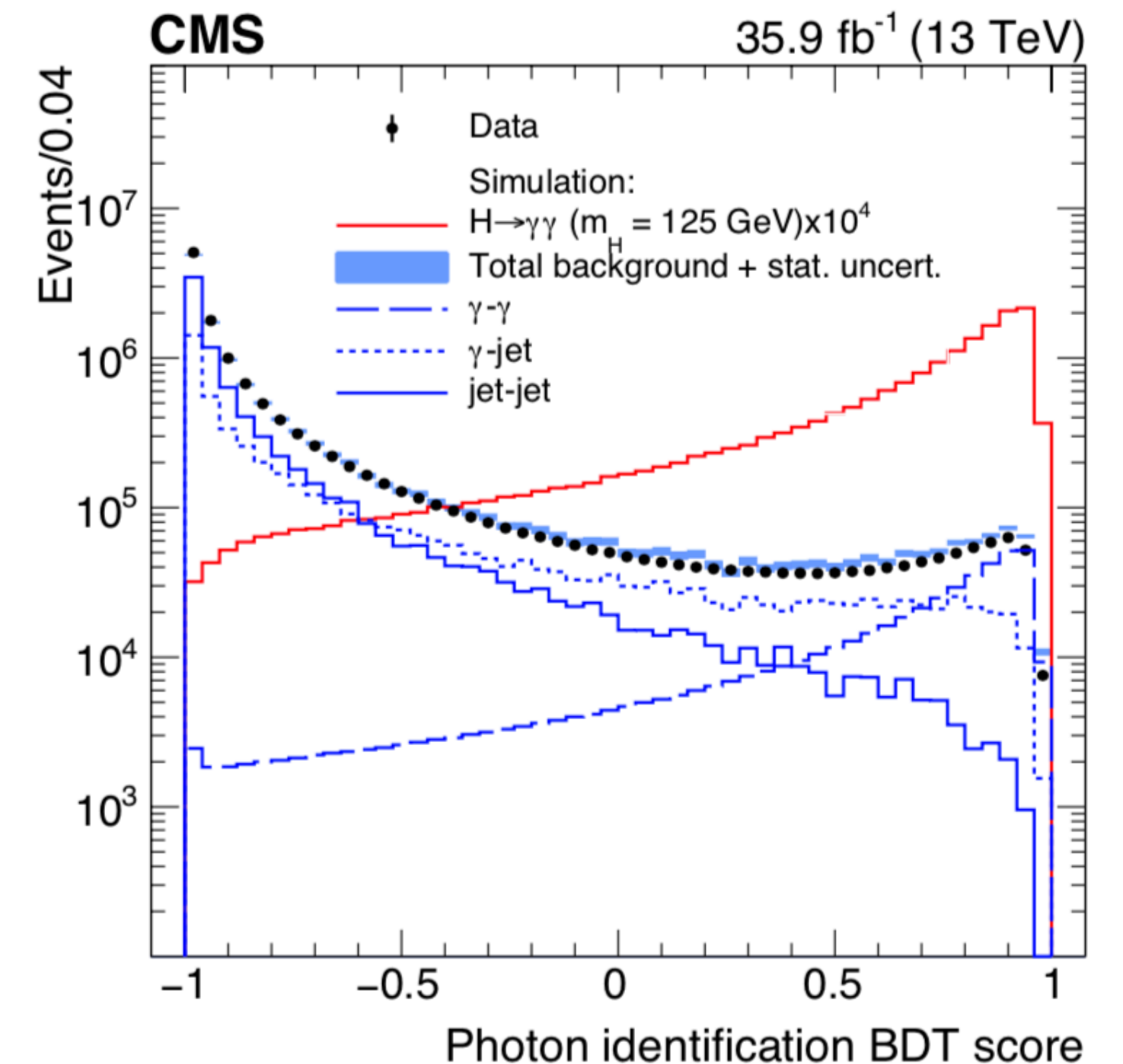


[A3D3](#)



# BDTs for Higgs

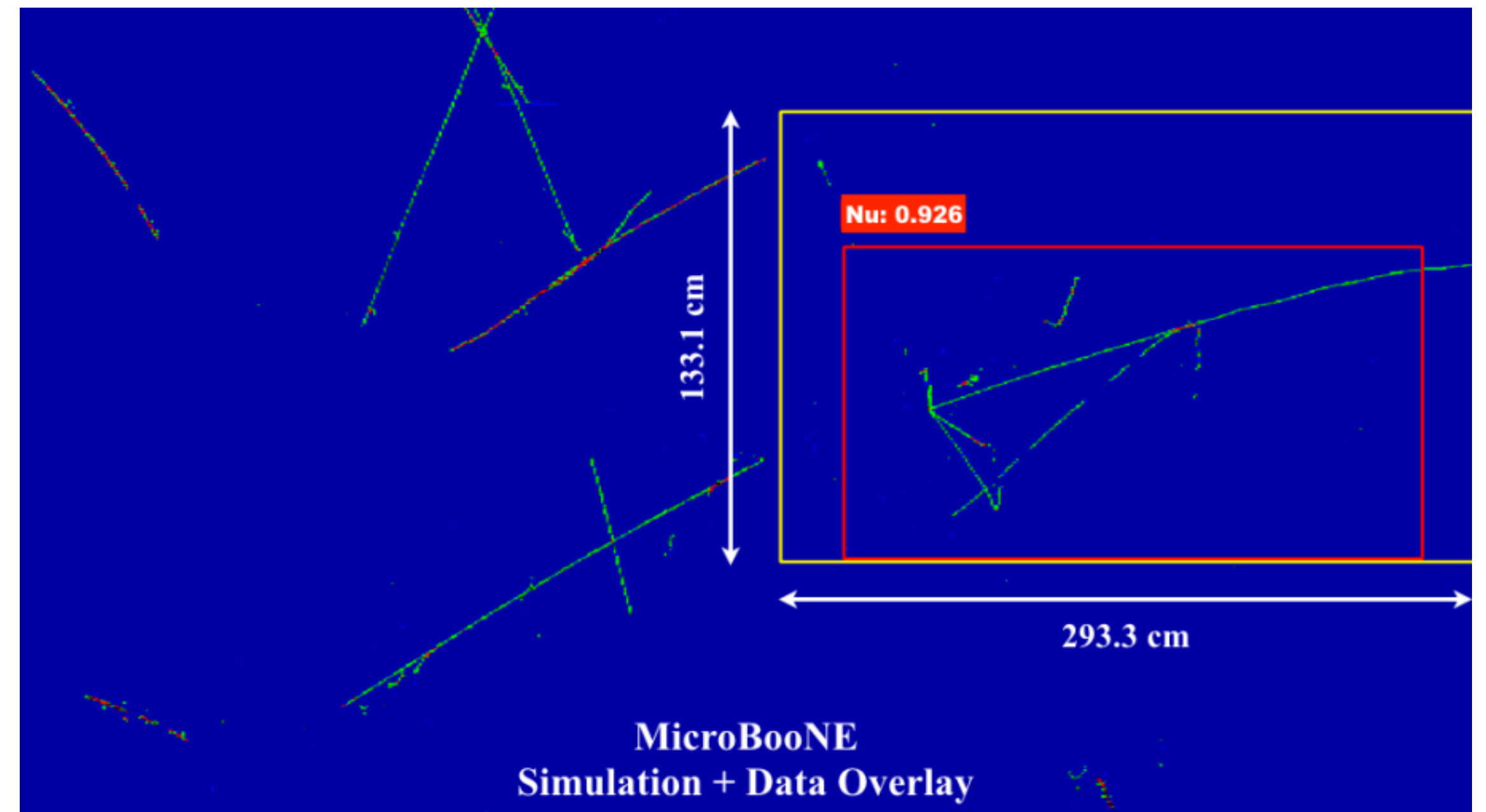
- Several BDTs involved in the analysis of Higgs boson decay to two photons using high-level variables
  - e.g. particle mass,  $\eta$ , isolation
- To separate signal photons from background (photons from jets)
- Choosing the most likely vertex for the photons (they are neutral, so no tracking)
- A diphoton quality BDT (separating signal like  $\gamma\gamma$  events from background)
- Used to increase the purity of the selected diphoton dataset
- Increase in sensitivity due to ML equivalent to having 50% more data (and no ML)



[arXiv:1804.02716v2](https://arxiv.org/abs/1804.02716v2)

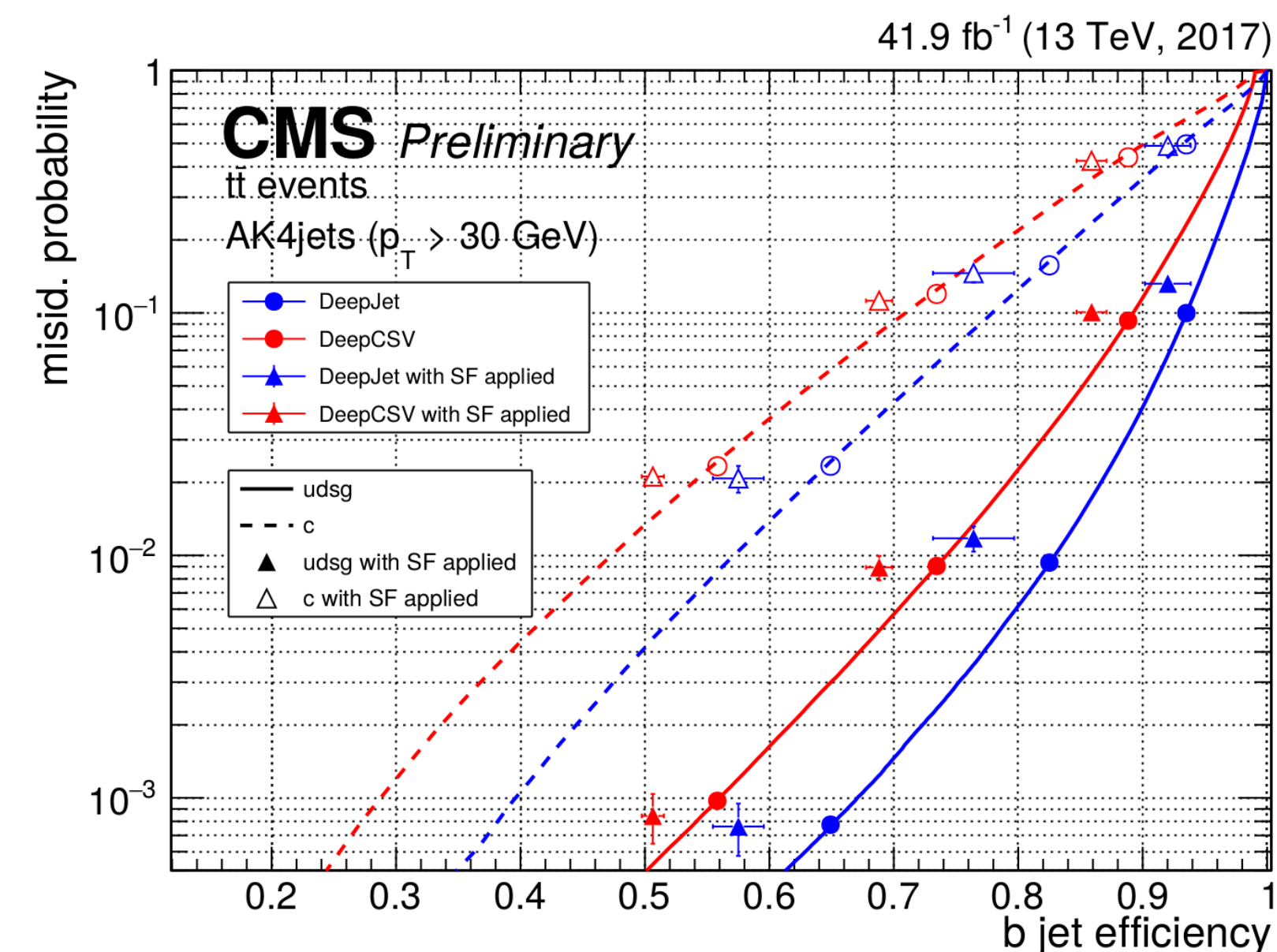
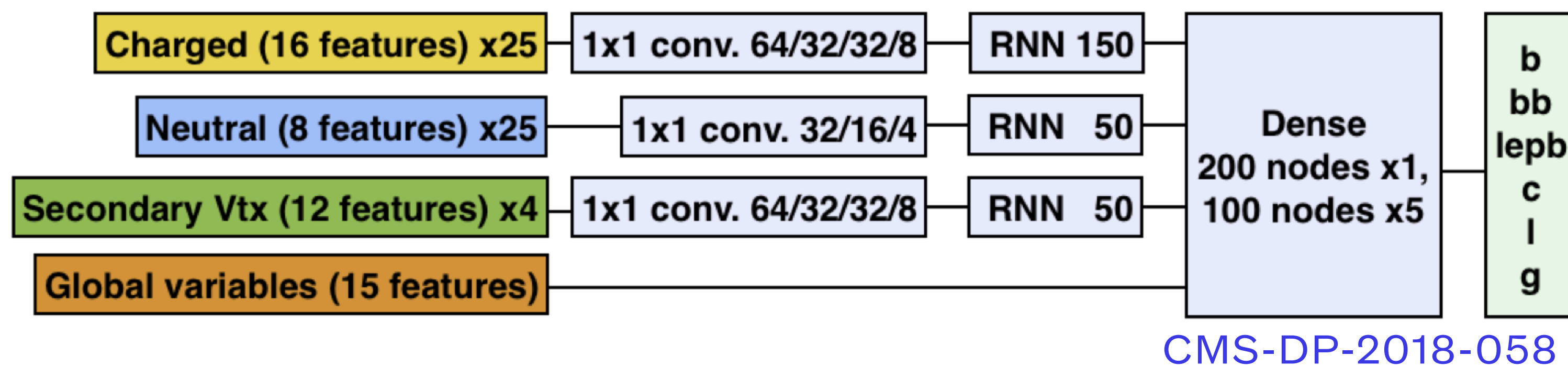
# Neutrino Detector Reconstruction

- From MicroBooNE, Liquid Argon time-projection chamber (LArTPC) neutrino experiment
- Using a CNN to identify neutrino interactions using a CNN
- e.g. simulated neutrino interaction yielding 1  $\mu$ , 3  $p$ , 2  $\pi$ . Background from cosmic data
- Yellow box is 'truth' box containing all charge deposits from simulated interactions
- Red is bounding box predicted by CNN



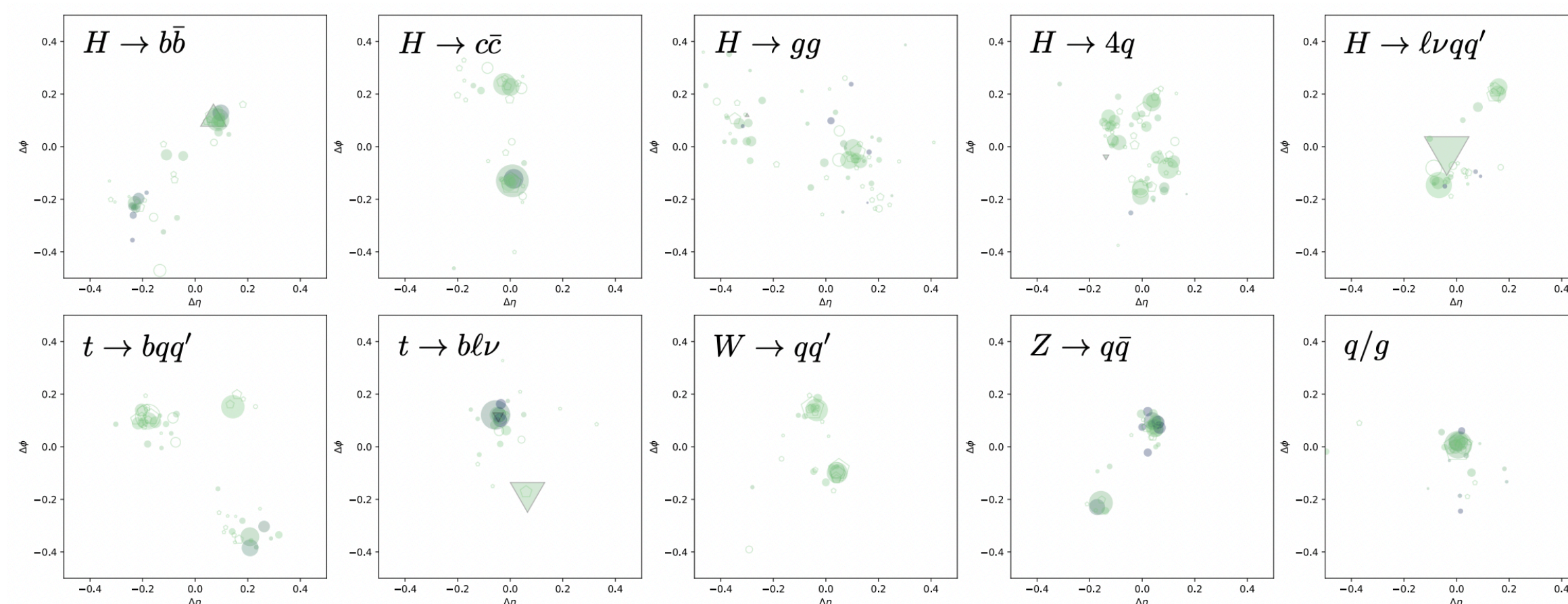
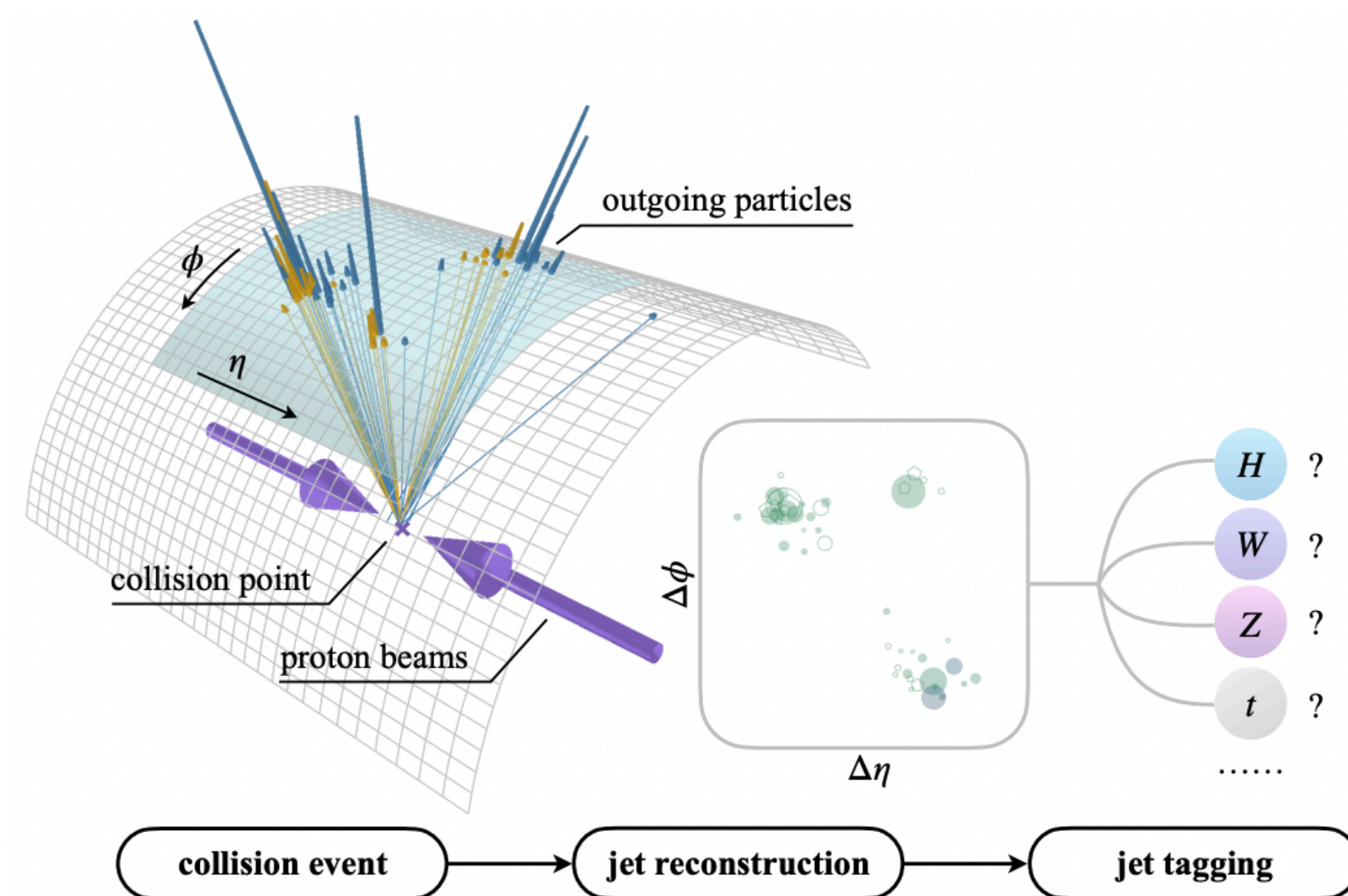
# Jet Tagging

- Big successes in HEP from ML for jet ID, example: DeepJet from CMS
- 1x1 CNN layers for 'feature engineering' (combining variables of single particles)
- LSTM recurrent networks iterate over particles sequentially
- Finally Dense layers combine features learned from the previous steps and the global variables



# Jet Tagging

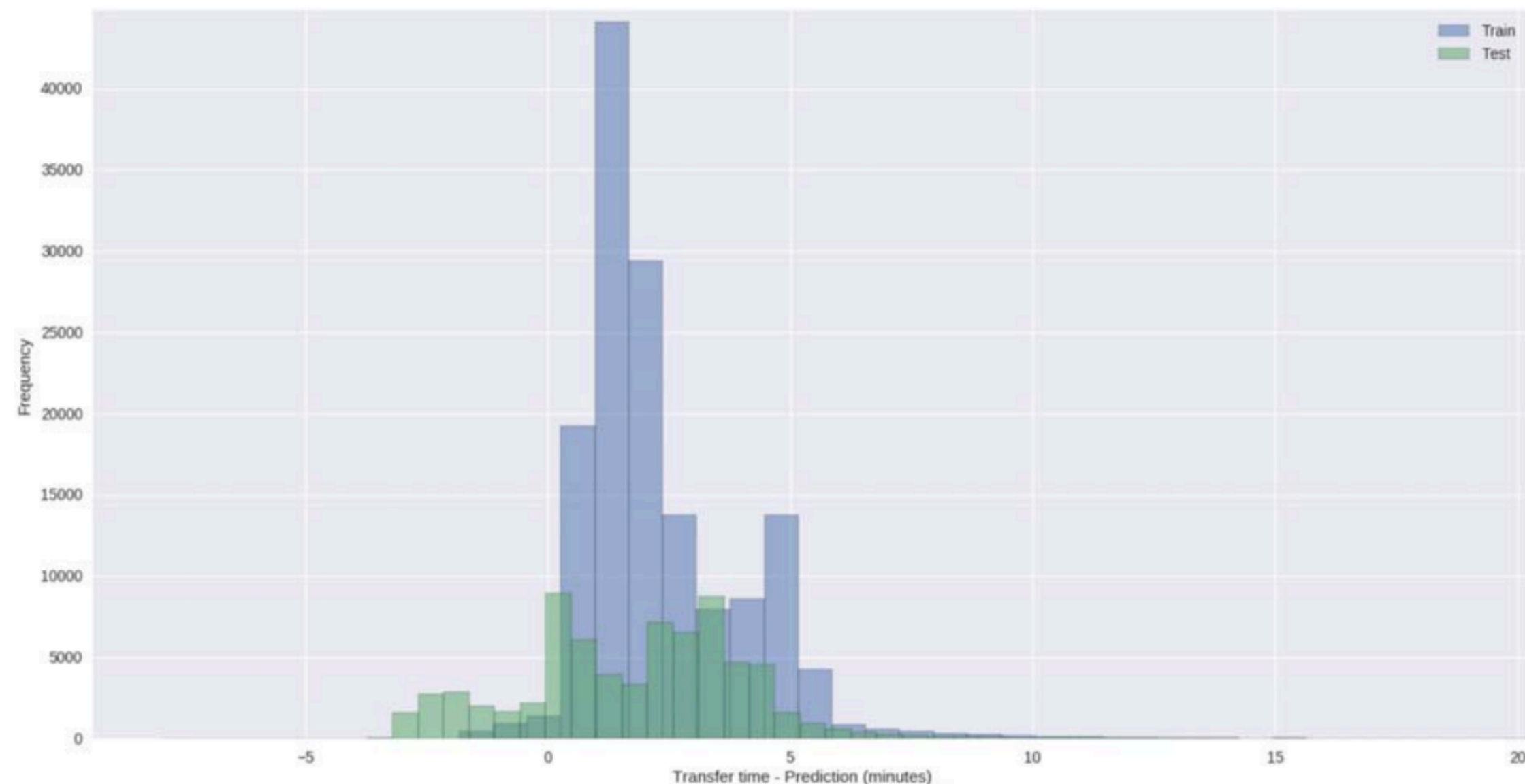
- Jet tagging is an area of HEP rich in ML: given the final state observables, what type of particle initiated the jet?
- How to represent the jet? Lots of approaches have been tried, relating to the different NN architectures
  - High-level observables reconstructed with *classical* means -> fed into MLP
  - Make images from individual particles by applying a grid -> Convolutional NN
  - Make lists of particles (often  $p_T$  ordered) -> Recurrent NN or Transformer
  - Represent particles as a graph (point cloud with connections) -> Graph NN



[arXiv:2202.03772](https://arxiv.org/abs/2202.03772)

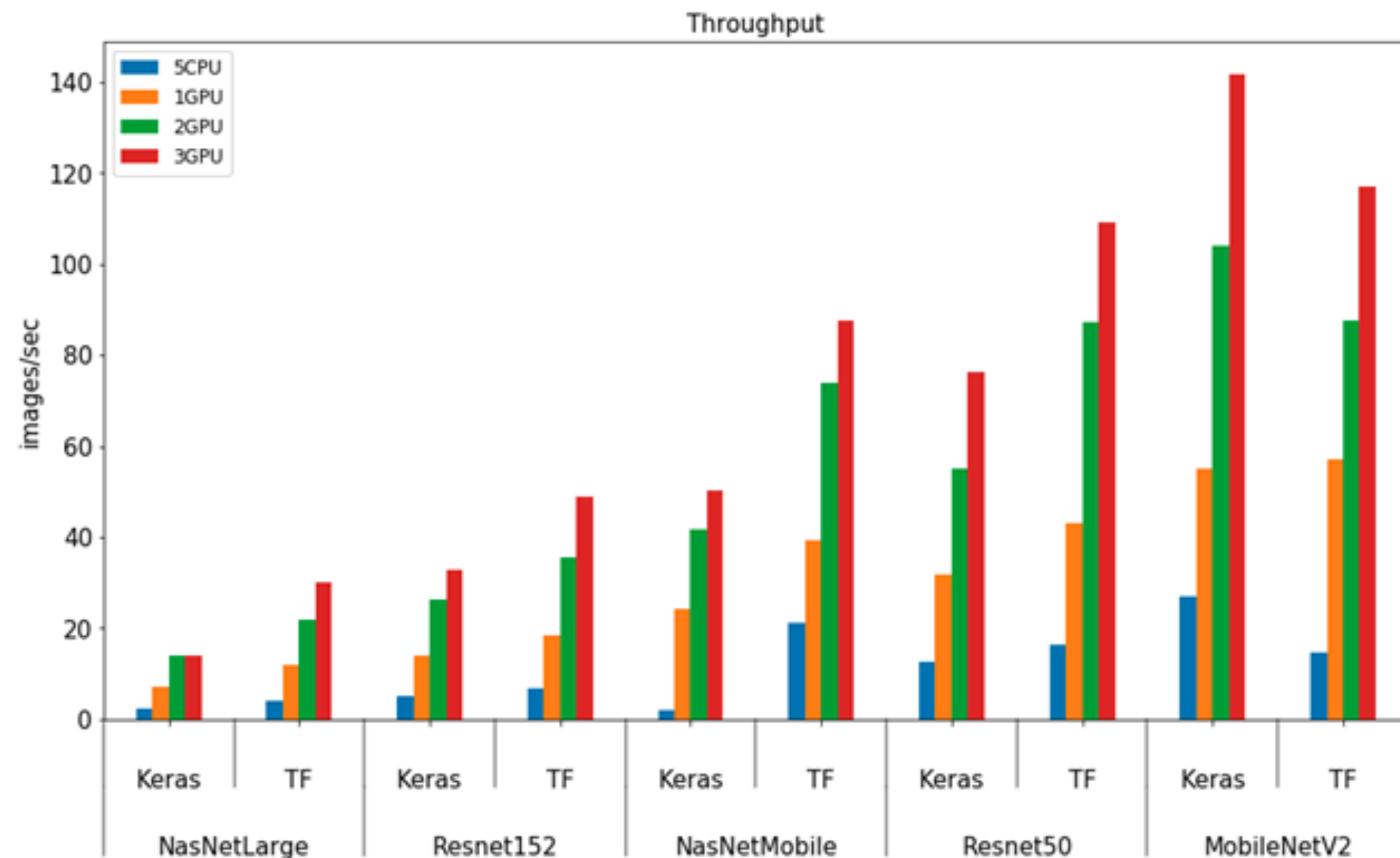
# ML For Networking

- From ATLAS, predicting the transfer time of files between sites
- One metric in determining the network-aware scheduling of GRID jobs and file storage
- Uses a Long Short Term Memory (LSTM)
- Inputs: source, destination, activity, bytes, start timestamp, and end timestamp



# GPUs for ML

- Biggest gains for GPUs are seen in training, but they also outcompute CPUs in inference
  - But remember you have to get the data to the device
- Here, running inference on K80 GPUs, measuring images / second (throughput)



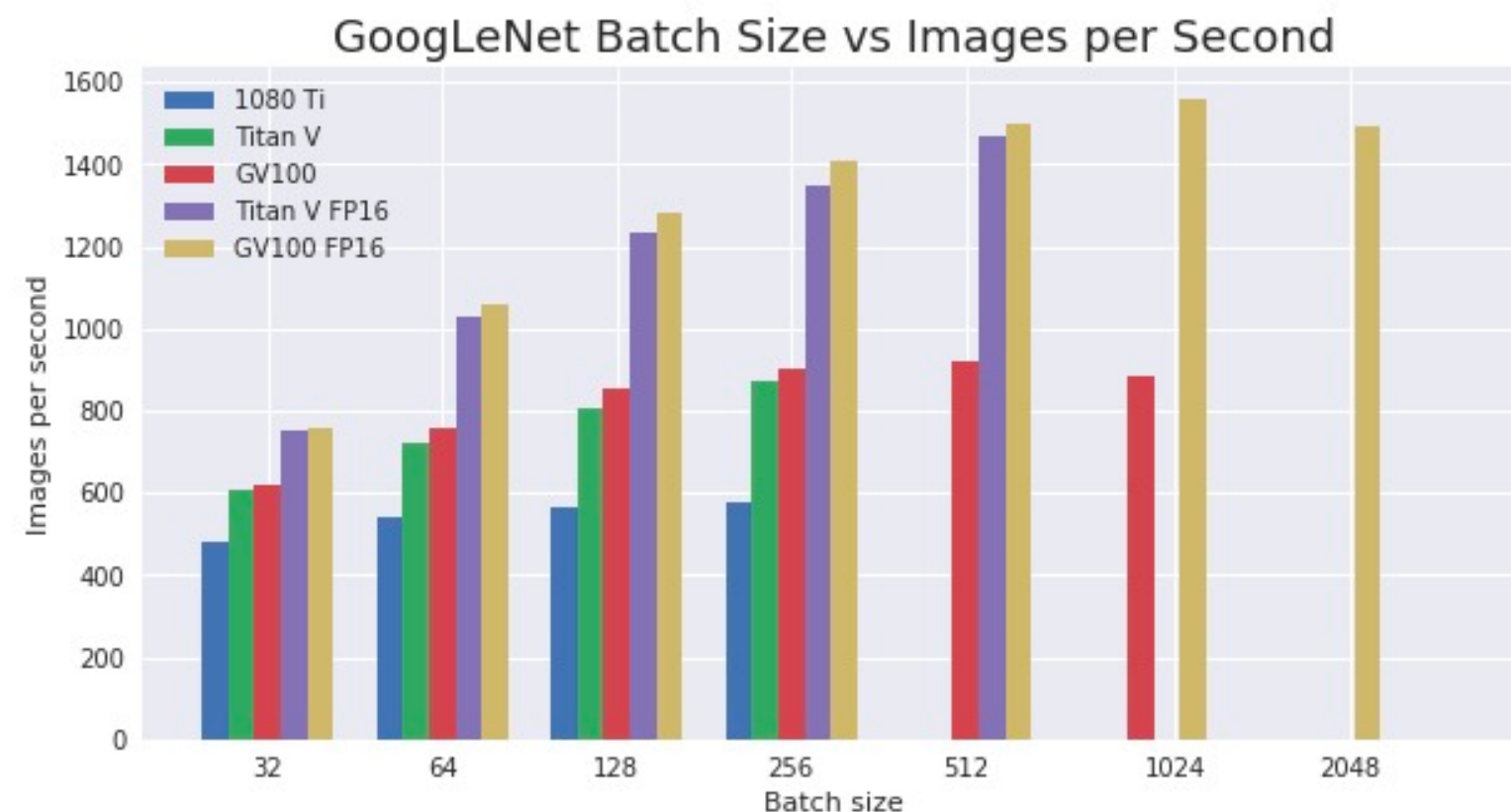
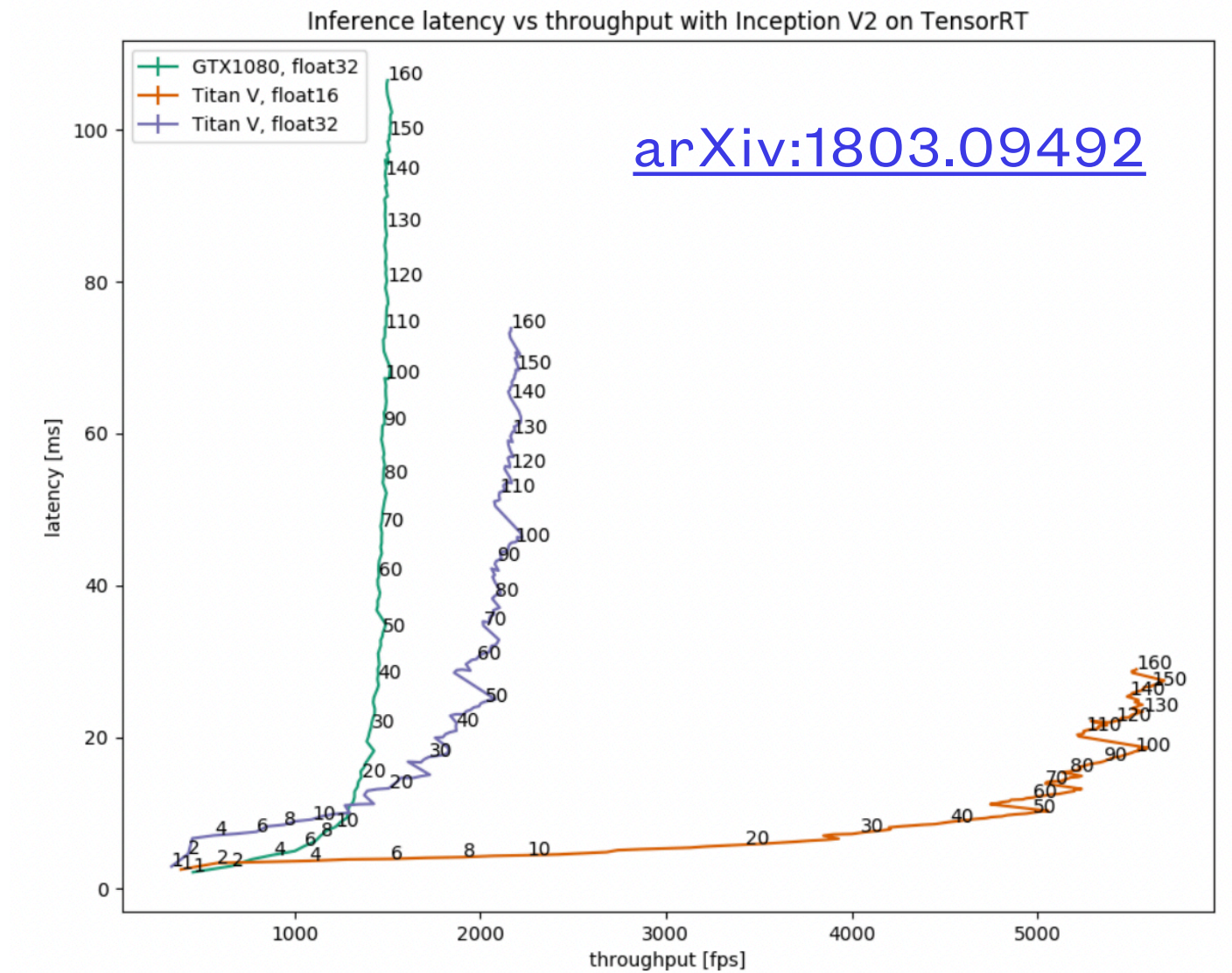
From [Microsoft Azure](#)

- [mlperf.org](#) has nice benchmarking of different hardware (not only GPUs) running on different models



# GPUs for ML - batching

- “Batching” is a common technique for better hardware utilisation
  - Relevant both at training and inference time
- Send several data samples to the GPU in one batch to maximise use of memory bandwidth and compute
- Is the constraint latency or throughput?
  - If strictly latency: low batch size
  - If throughput: high batch size
  - Both: batch size where throughput saturates

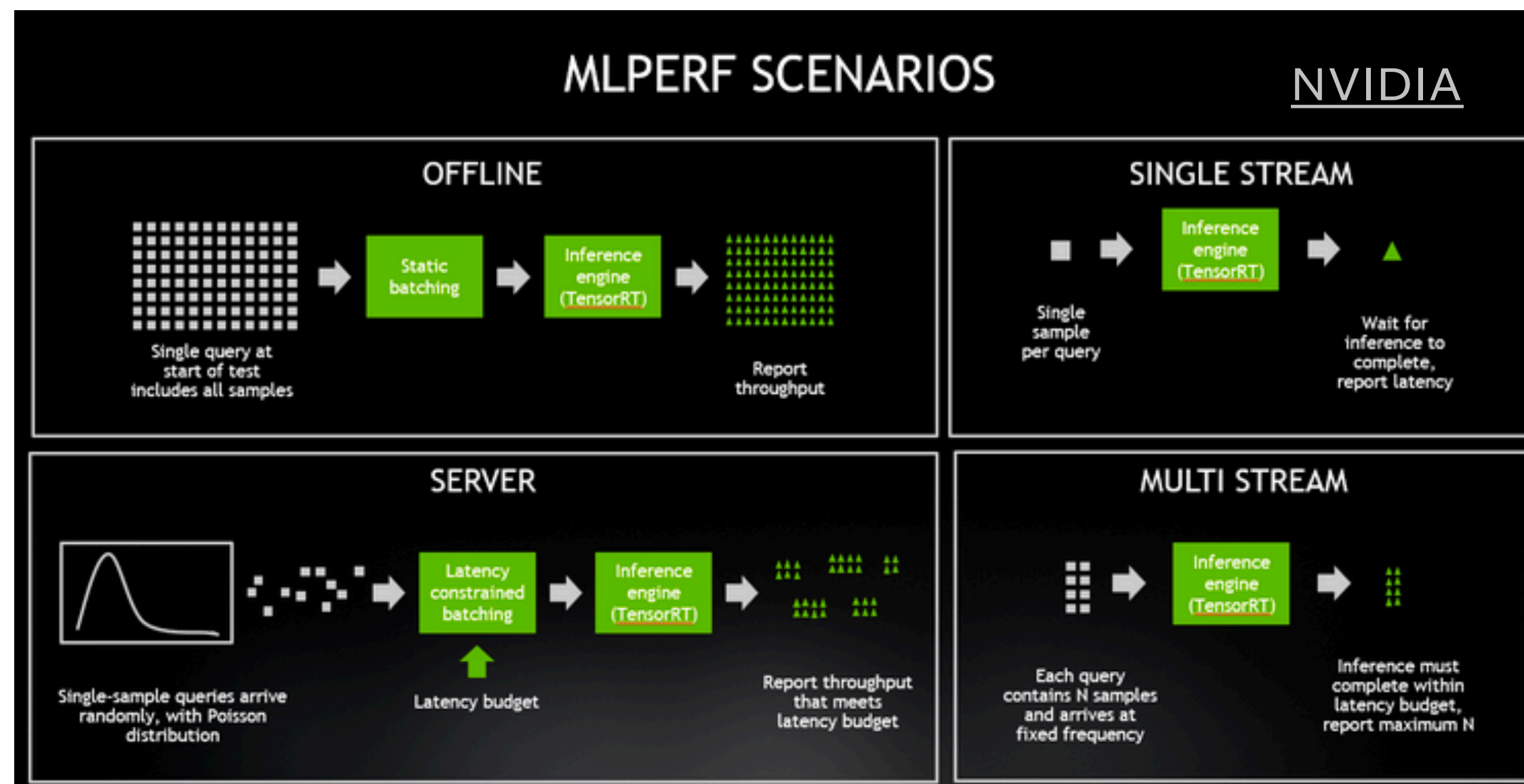


[Puget Systems](#)

- Plot: throughput vs latency at different batch sizes for Inception V2 (large computer vision CNN)
  - On different GPUs and different precisions

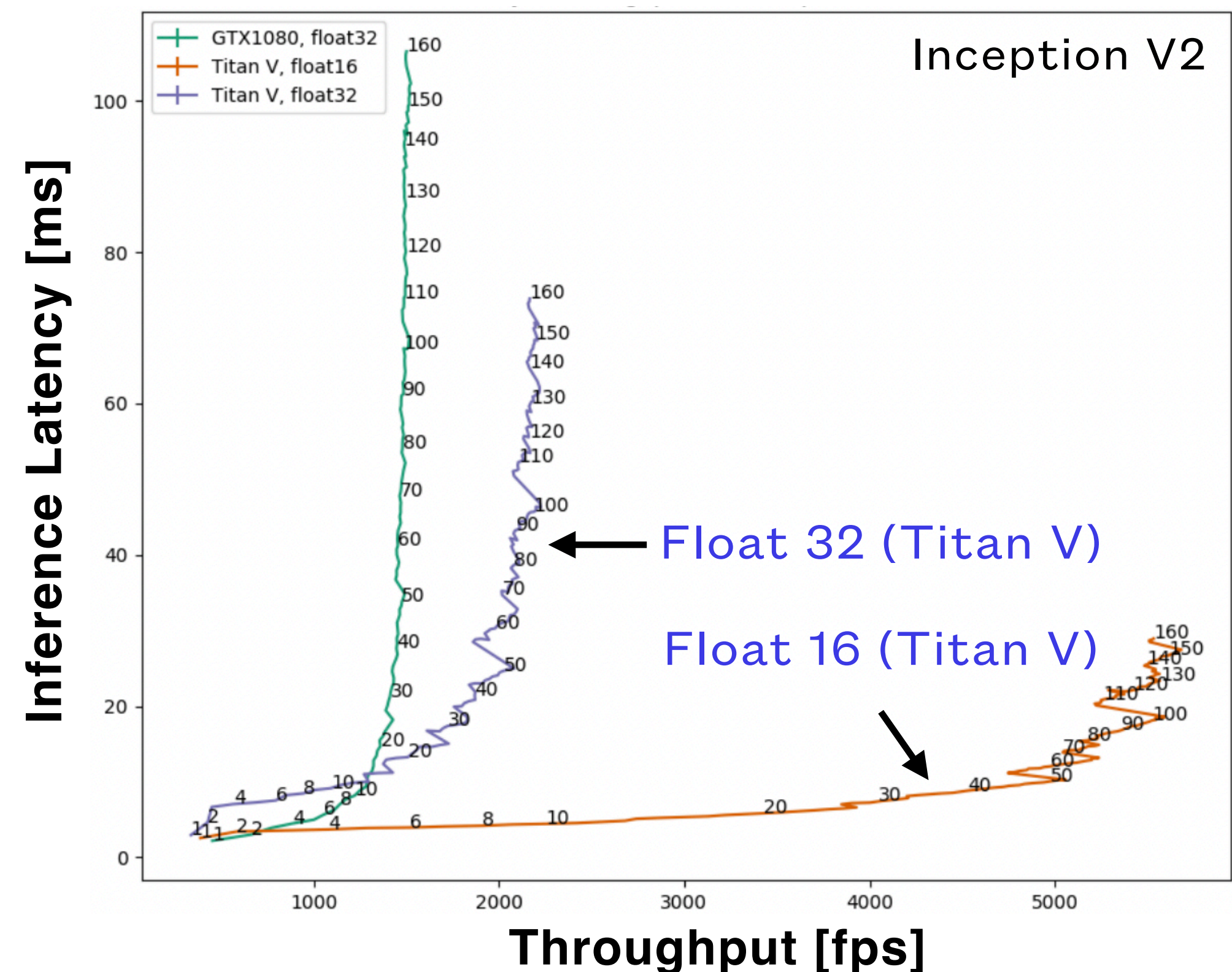
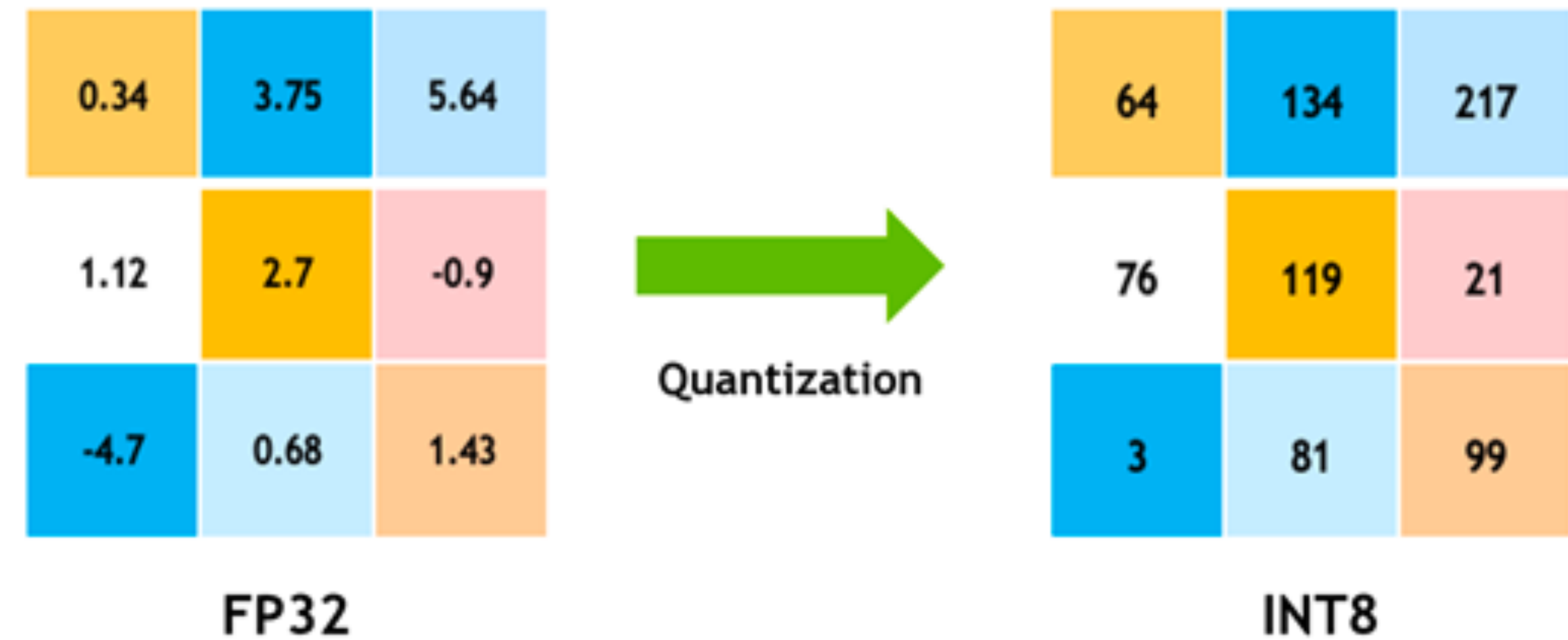
# GPUs for ML - batching

- Whether or not you can profit from batching depends also on:
  - Is the main constraint on throughput or latency? (Or both?)
  - The data source: do data arrive at fixed intervals (bottom right image), or stochastically (bottom left)?
  - Can you afford to wait to accumulate several samples before sending them to the GPU?



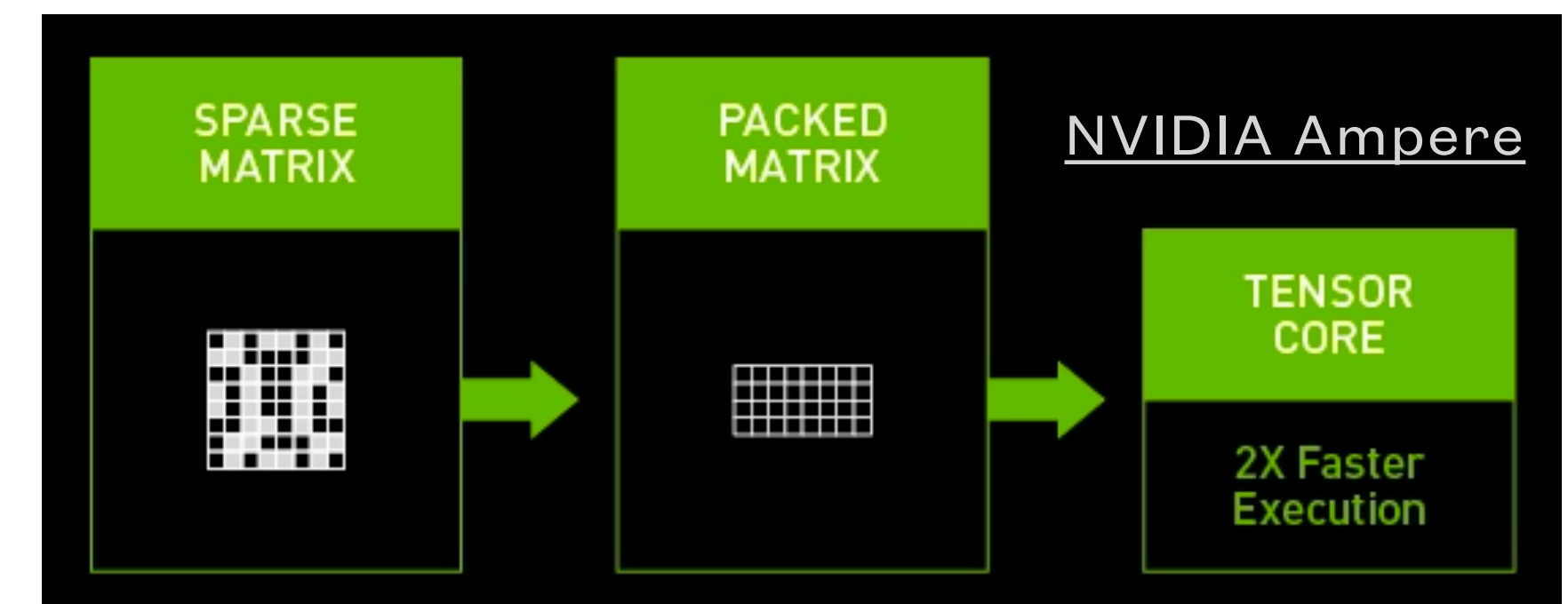
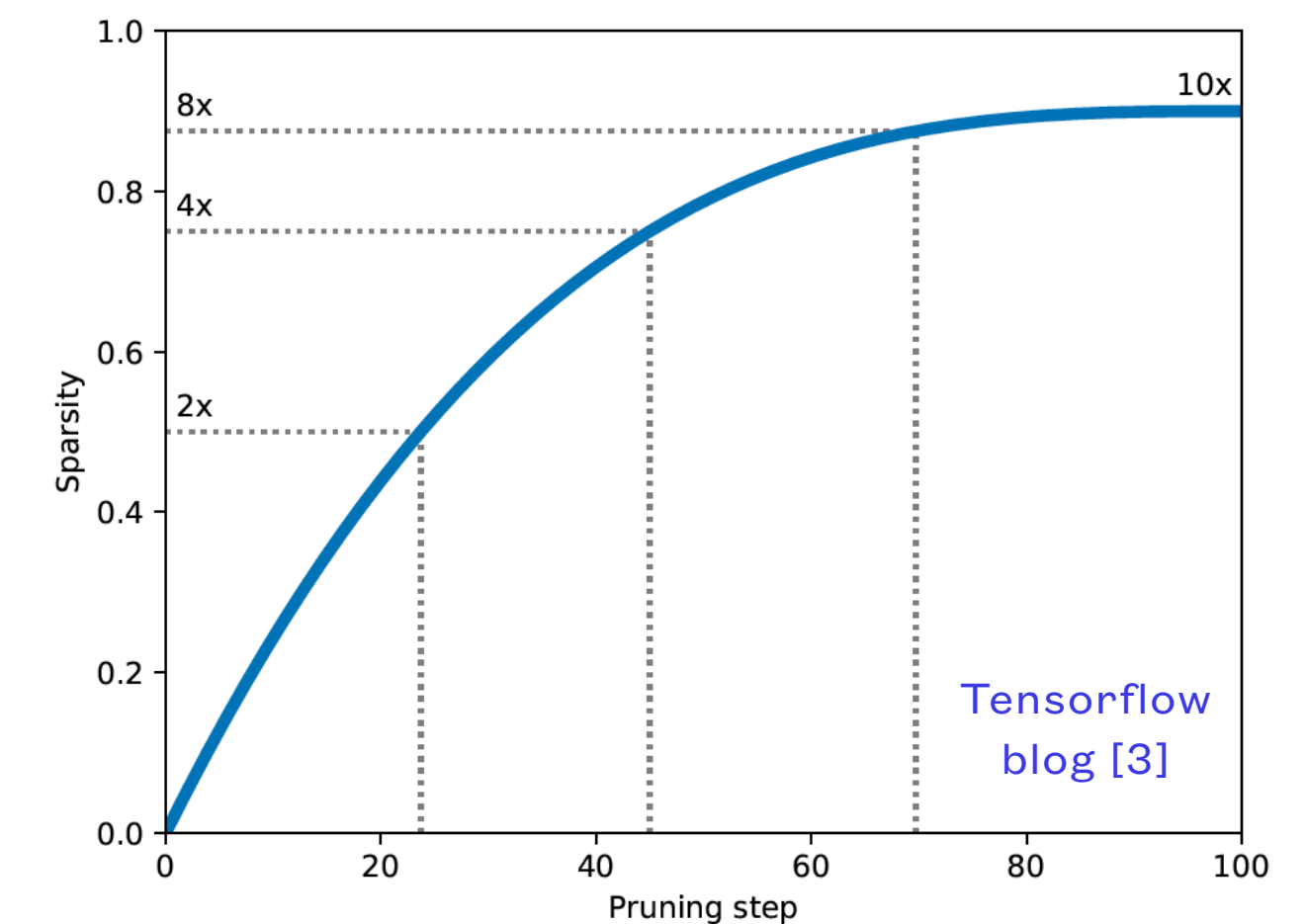
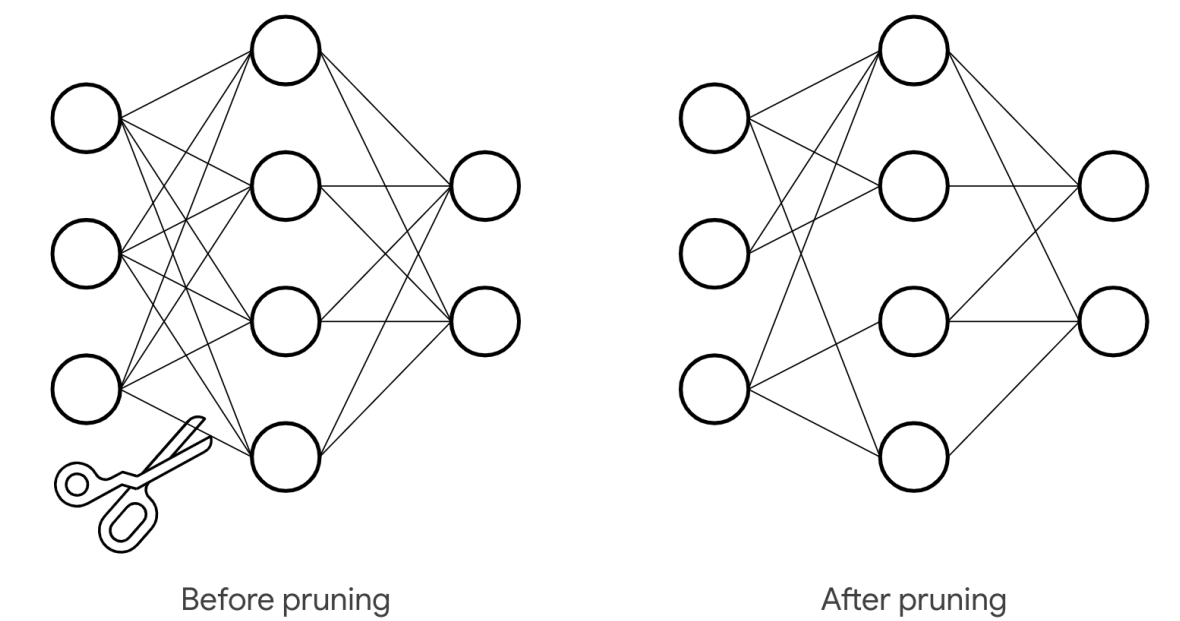
# Quantization

- Many GPUs support Int8, float16, bfloat16 precision with many more OPS than float32
- **Post Training Quantization (PTQ)** - train with FP32 then scale & round to lower precision
- **Quantization Aware Training (QAT)** - train with lower precision
  - TensorRT (NVIDIA GPU),
  - TensorFlow Lite (Google),
  - torch.quantization (PyTorch)
- Choice of precision depends on target hardware and requirements



# Pruning / Sparsity

- A Neural Network often contains many redundant connections
- Pruning = remove some connections from final model
- Can reduce the model size (memory footprint)
- Some processors can accelerate sparse networks
  - Basically - don't do the *multiply by 0* computations
- Different methods:
  - Regularisation (penalise low value weights, then make them 0)
  - Target sparsity, e.g. sparsity ramp up with TFMOT
  - Structured pruning - remove continuous blocks of weights;
  - Filter pruning - entire filters of CNN
- Applies also to BDTs ( $\lambda$ ,  $\alpha$  in xgboost)
- Can be coupled with Quantisation Aware Training



# LHCb, Bonsai BDT

- In LHCb, Bonsai BDT has been used since the beginning of LHC data taking in their online software event selection
- Bonsai BDT is a technique to compress BDTs into a binned parameter space for faster execution
- Was used in the main selection path for most LHCb analyses



# Data Quality Monitoring

- Using an Autoencoder for anomaly detection
  - Network has a 'bottleneck' that learns an abstract representation of the data
  - After bottleneck, decoder network tries to reproduce the input image
  - For anomalous input, the recreated image is not similar to the original input, and flagged
- Applied to CMS muon drift tube system, able to identify failures not spotted by previous, rule based system

