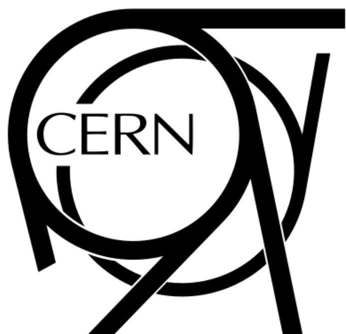
Fast Inference with FPGAs

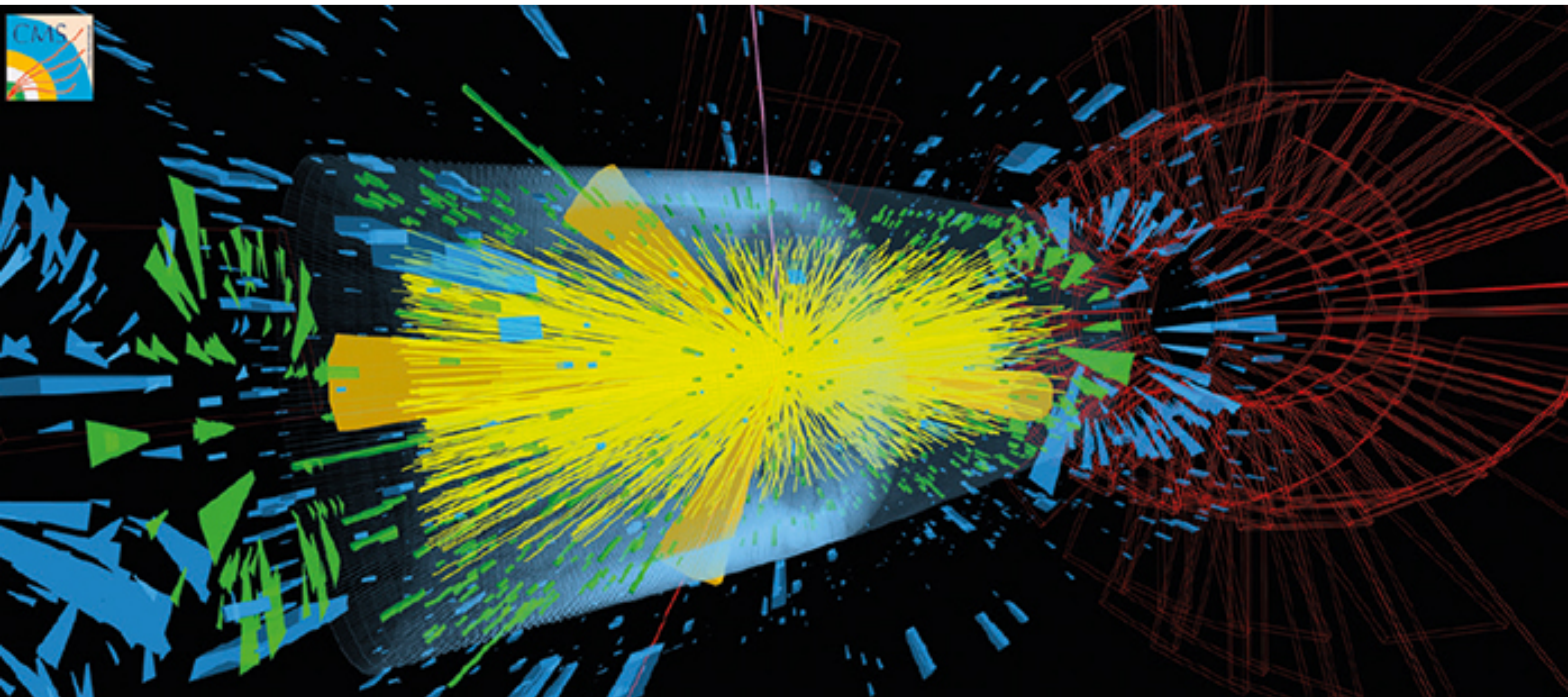
Game of Flavours - IUC Dubrovnik - 3/5/19

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)



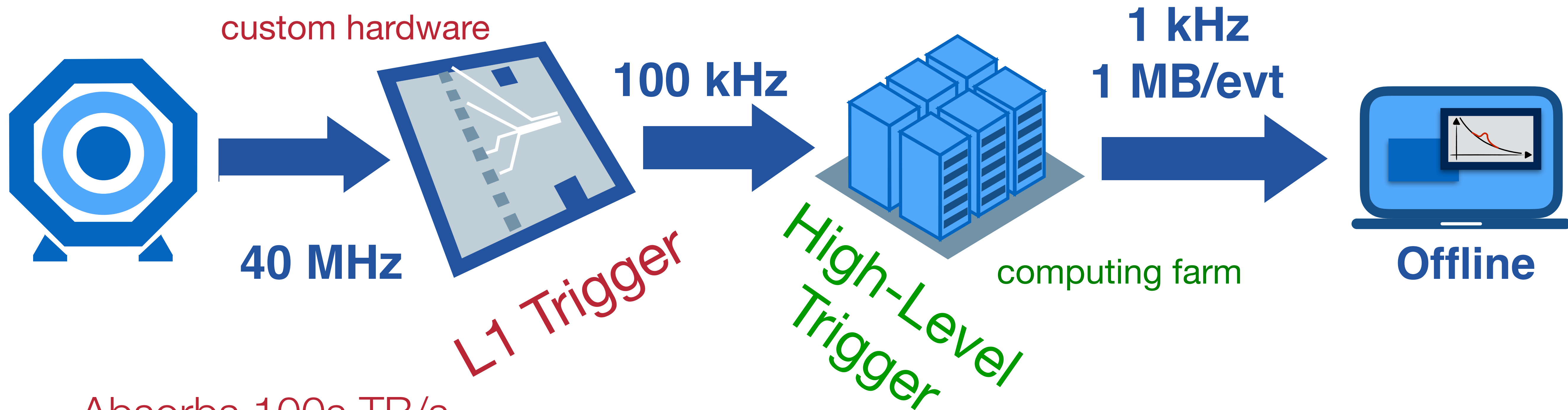
Introduction & Motivation

- Machine Learning very popular in LHC experiments - used in reconstruction and analysis, offline and in HLT
- We trigger because we cannot store all events (100s TB/s @ HL-LHC) and not all events are interesting
- Extreme pileup at HL-LHC (~200) needs sophisticated techniques to maintain the performance of the trigger - maintaining efficiency without exceeding allowed rate - can we exploit Machine Learning?



A typical trigger system

Triggering typically performed in multiple stages @ ATLAS and CMS



Absorbs 100s TB/s

Trigger decision to be made in O(μs)

Latencies require all-FPGA design (and/or custom ASICs)

Computing farm for detailed analysis of the full event

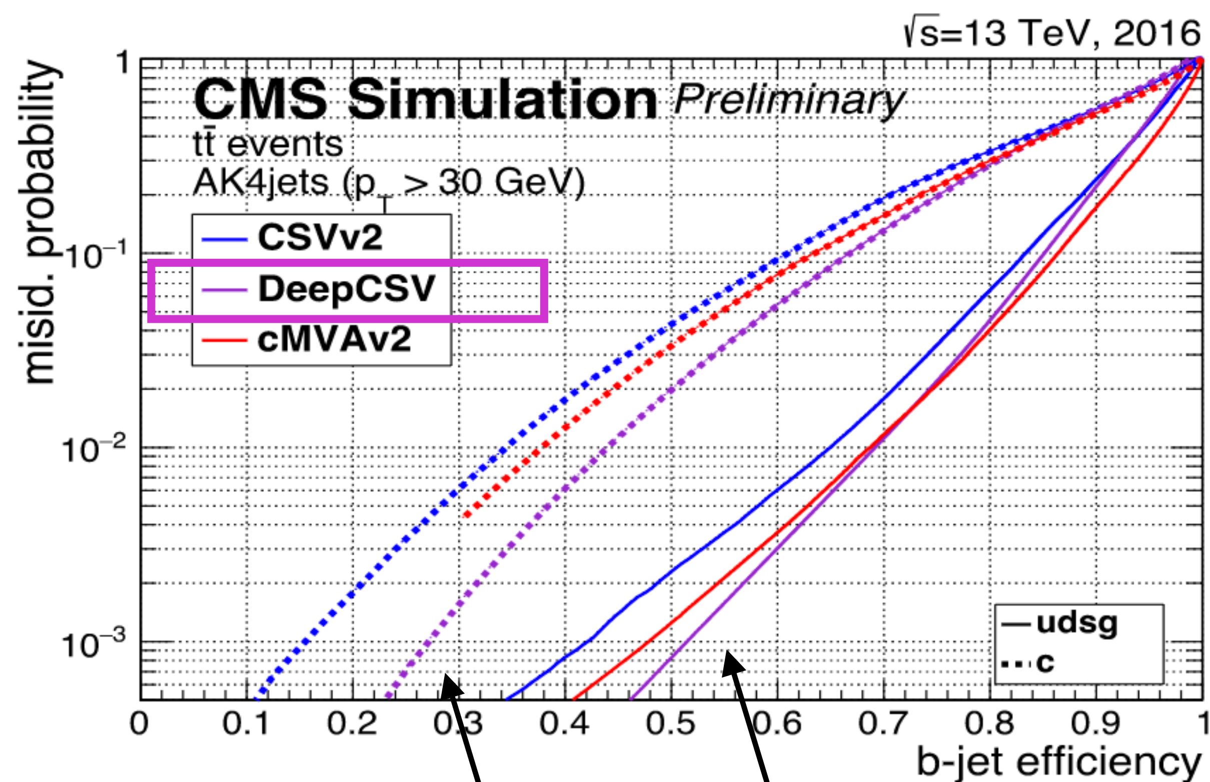
Latency O(100 ms)

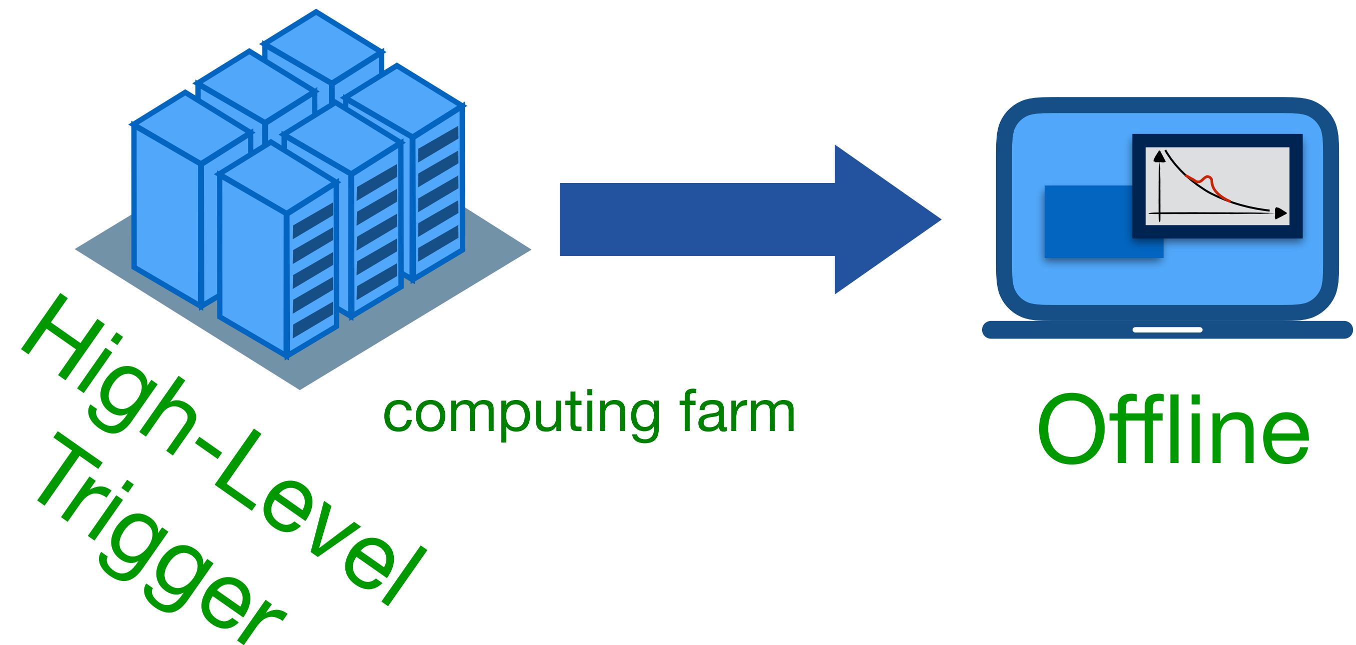
For HL-LHC upgrade: latency and output rates will increase by ~ 3 (ex: for CMS 3.8 → 12.5 μs @ L1)

The latency landscape @ LHC

100 ms | 1 s

ex, identification of b-quark jets





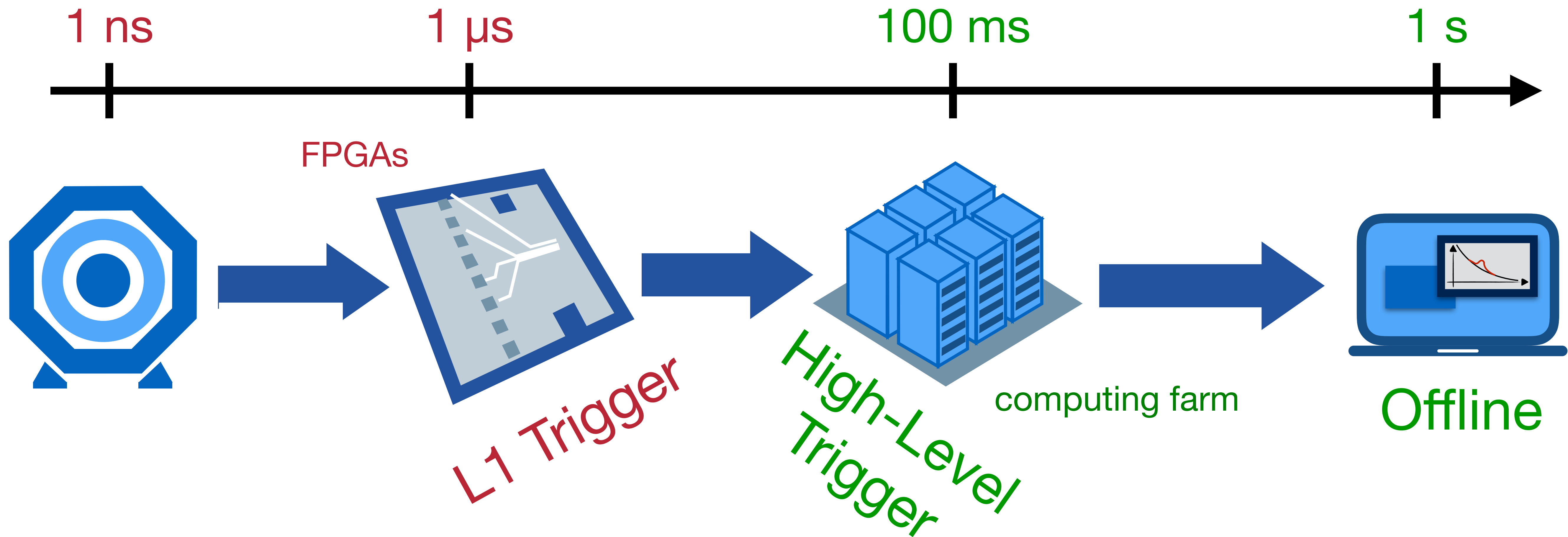
Deep neural network based on high-level features

both offline and @ HLT

ML methods typically employed in offline analysis or longer latency trigger tasks

This workshop: many different ML based flavour taggers in use

The latency landscape @ LHC



Goal of hls4ml: explore what ML can we do in ~1 μs in FPGAs, and enable physicists to deploy ML models in the trigger

Also investigate acceleration of ML models with FPGAs for the offline and HLT

https://arxiv.org/abs/1904.08986

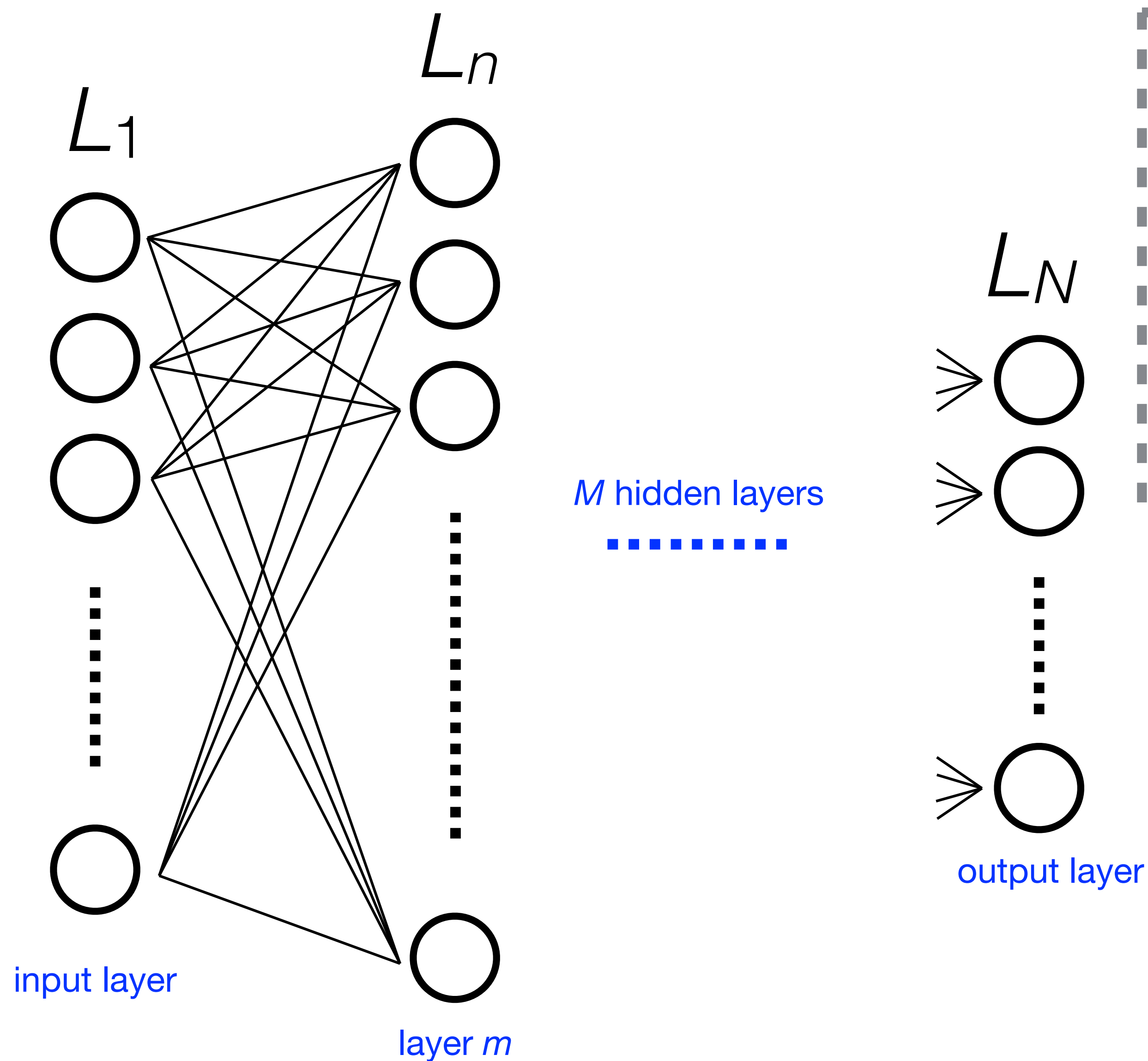
Neural network inference

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

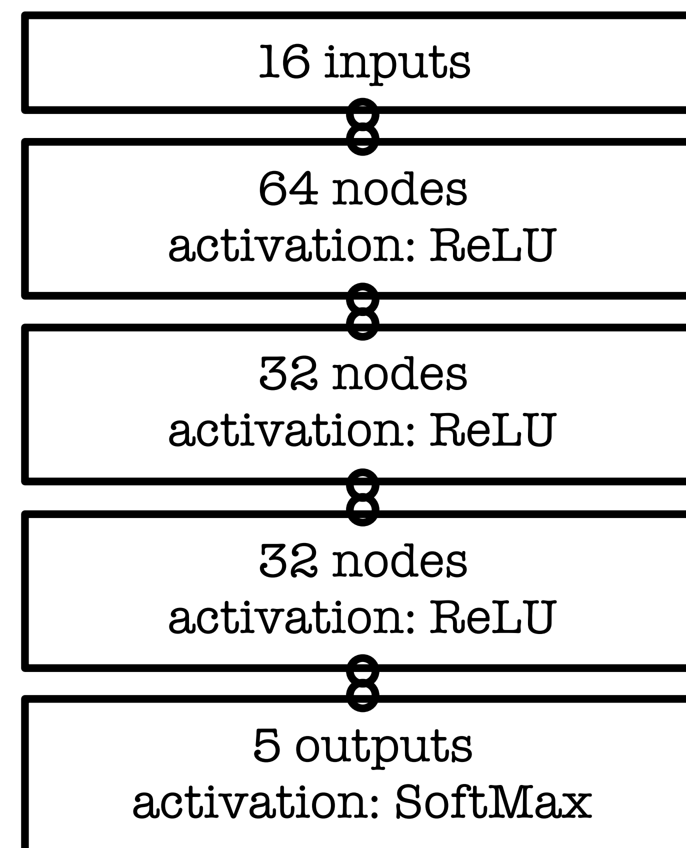
activation function — precomputed and stored in BRAMs

multiplication — DSPs

addition — logic cells



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



What are FPGAs?

- *'Field Programmable Gate Array'*
- Like a programmable, reconfigurable digital circuit
- Computation performed using *'resources'*:
  - **LUTs** (look up tables) - general purpose programmable logic (arithmetic, boolean functions)
  - **DSPs** (digital signal processors) - specialised cores for multiplication & accumulation
  - **BRAMs** (block RAMs) - small memories
  - **Transceivers** - Multi Tb/s streaming in
- Massive fine-grained *parallelism* from many computational resources in one chip
- *Pipeline* processing is like a production line for cars - many units (resources) work on different data

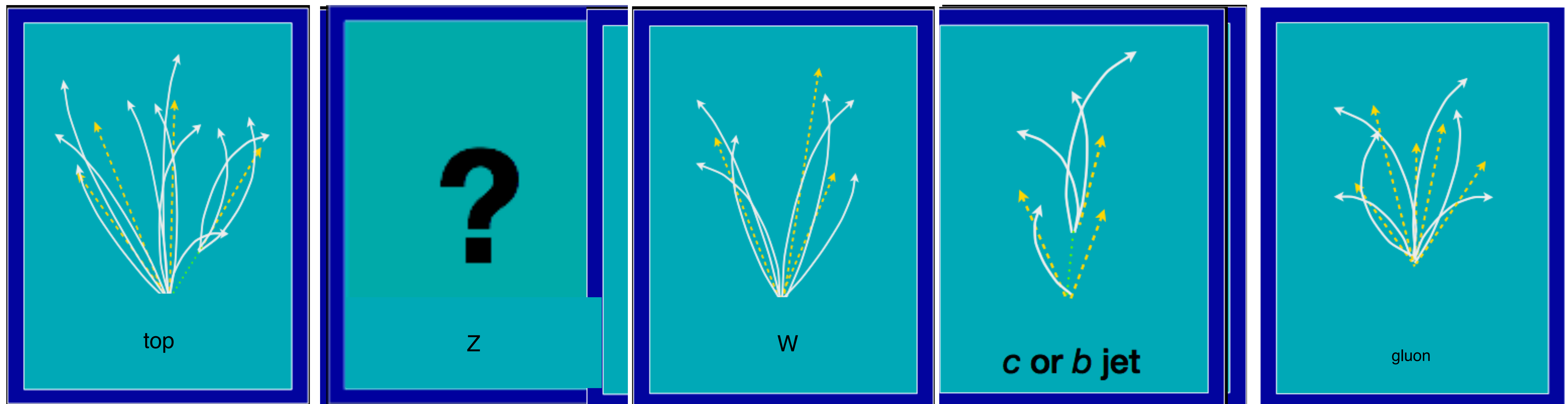
gluon jet

pileup jet

case: jet tagging

k to be implemented on FPGA: discrimination

l) ***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

W or *Z* jet

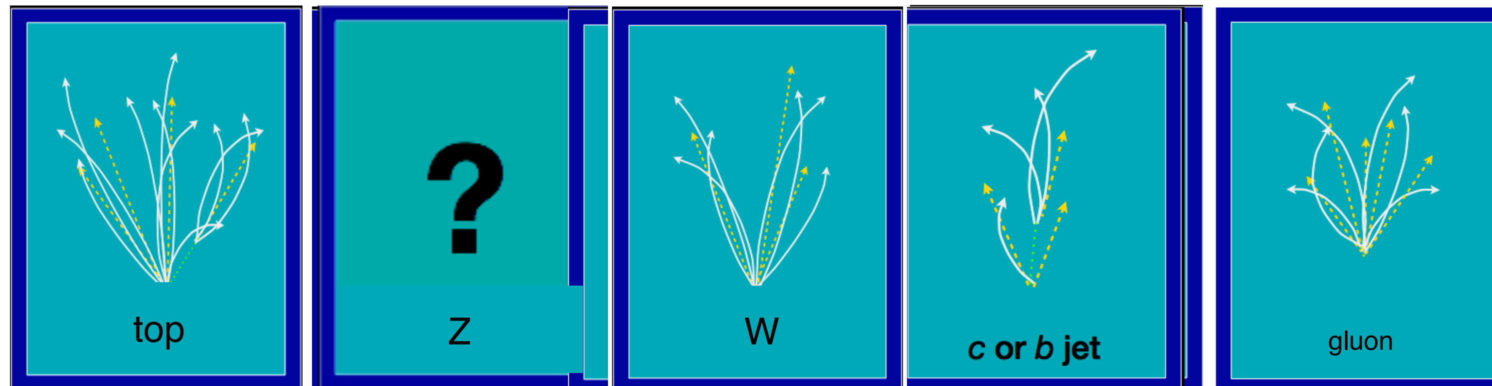
Higgs jet

top jet

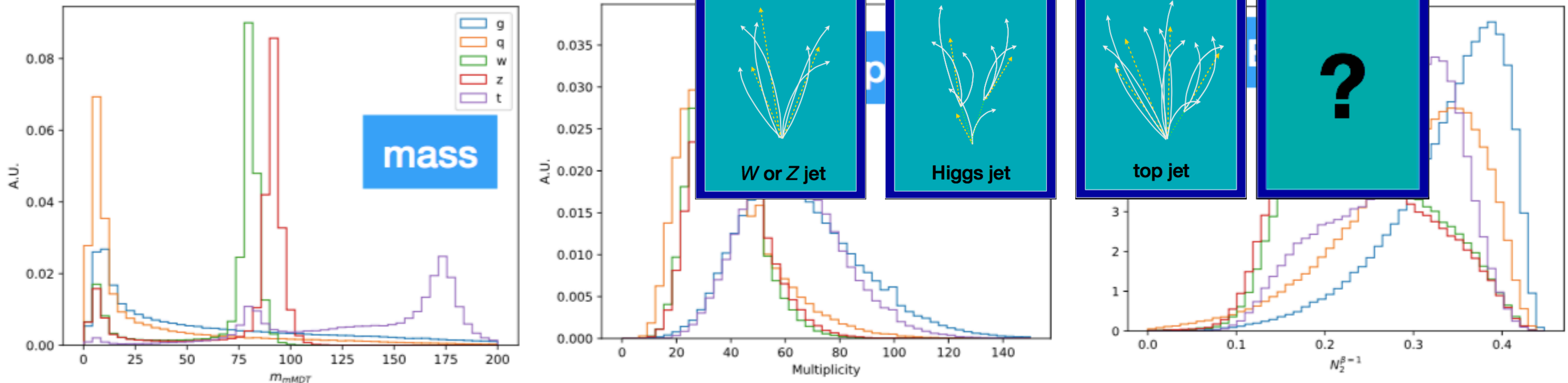
u,d or *s* jet
c or *b* jet
gluon jet
pileup jet

ase: jet tagging

W or *Z* jet
Higgs jet



pileup jet

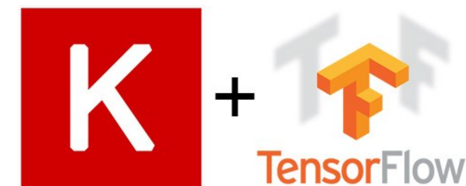


Input variables: several observables known to have high discrimination power from offline data analyses and published studies [*]

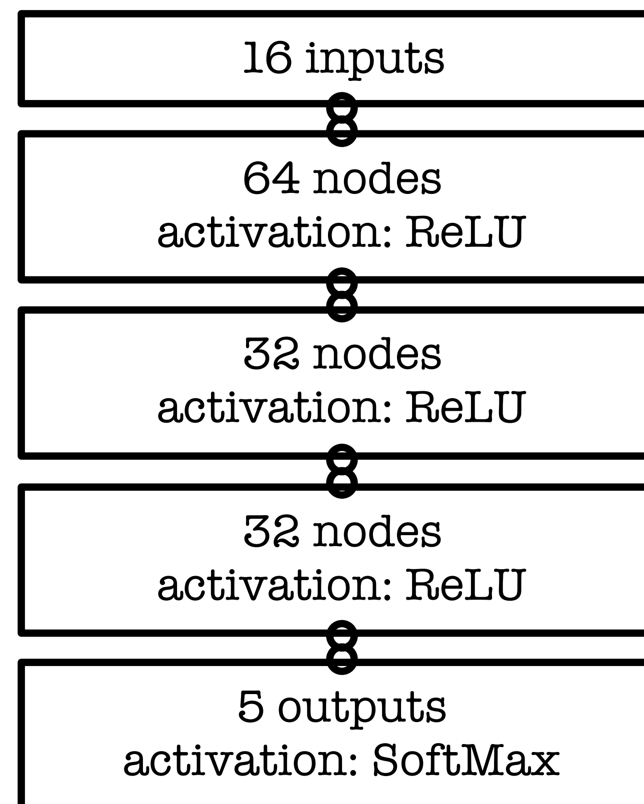
[*] D. Guest at al. PhysRevD.94.112002, G. Kasieczka et al. JHEP05(2017)006, J. M. Butterworth et al. PhysRevLett.100.242001, etc..

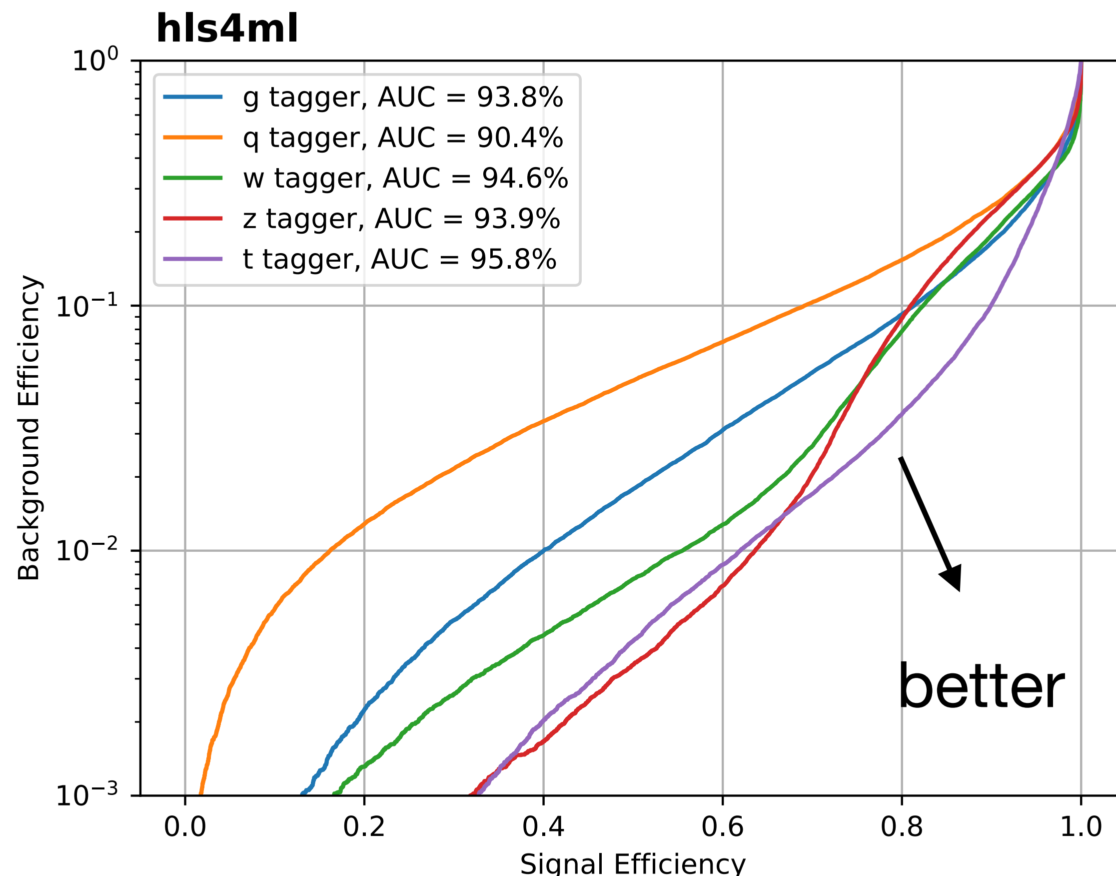
Physics case: jet tagging

- 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



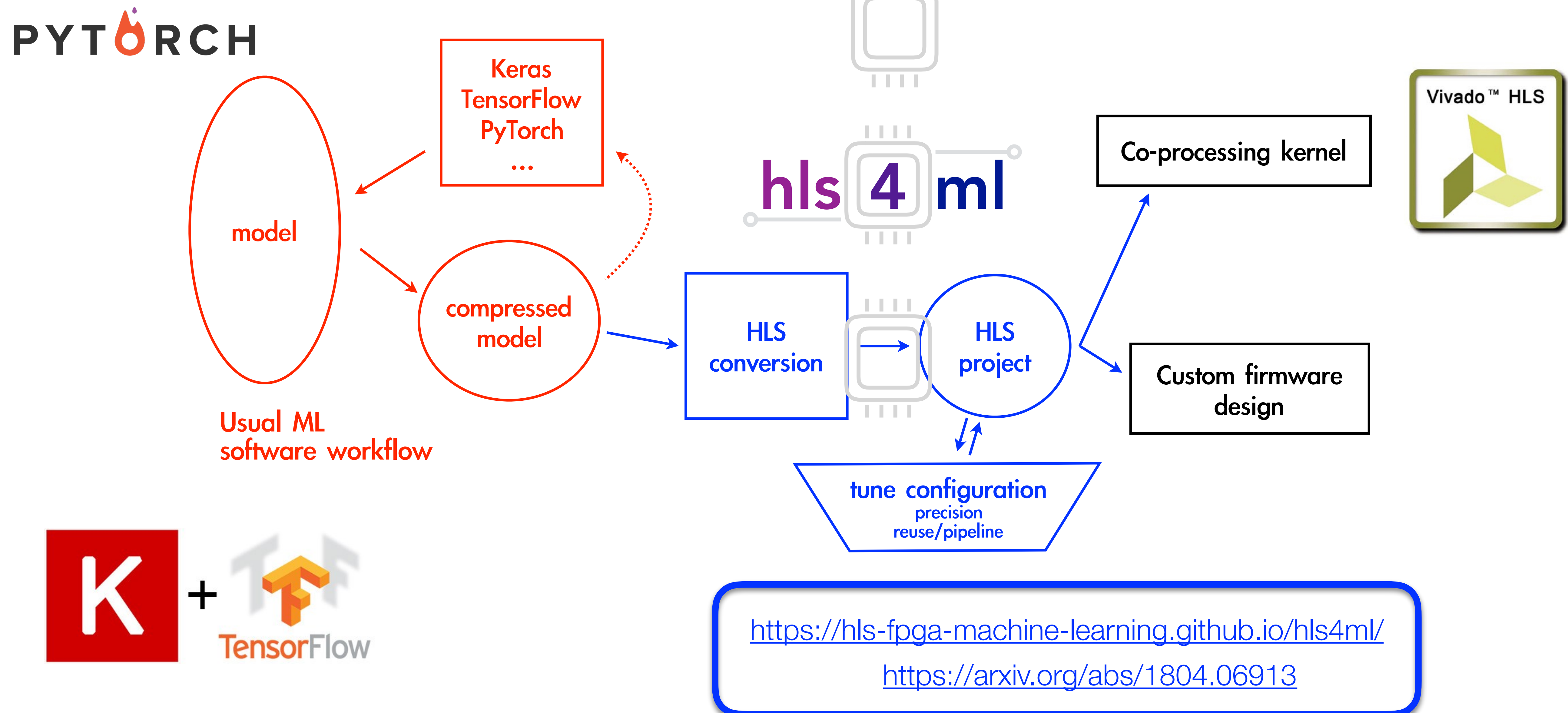


AUC = area under ROC curve (100% is perfect, 20% is random)

high level synthesis for machine learning

Implemented a user-friendly and automatic tool to develop and optimize FPGA firmware design for DL inference:

- Train ML models with your favourite ML library, run `hls4ml` to make FPGA code
- We implement and support as many architecture types, activation functions, specialities as possible
- Uses Xilinx HLS software - C++ code accessible to non-experts, and provides lots of flexibility



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:

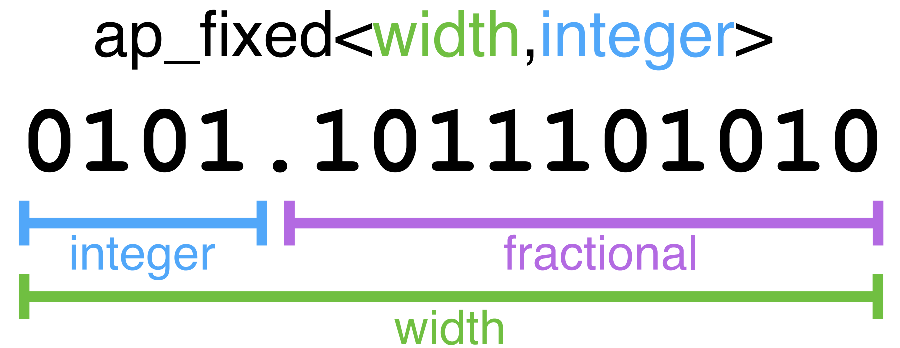
- **compression:** reduce number of synapses or neurons
- **quantization:** reduces the precision of the calculations (inputs, weights, biases)
- **parallelization**: tune how much to parallelize to make the inference faster/slower versus FPGA resources

NN TRAINING

FPGA PROJECT DESIGNING

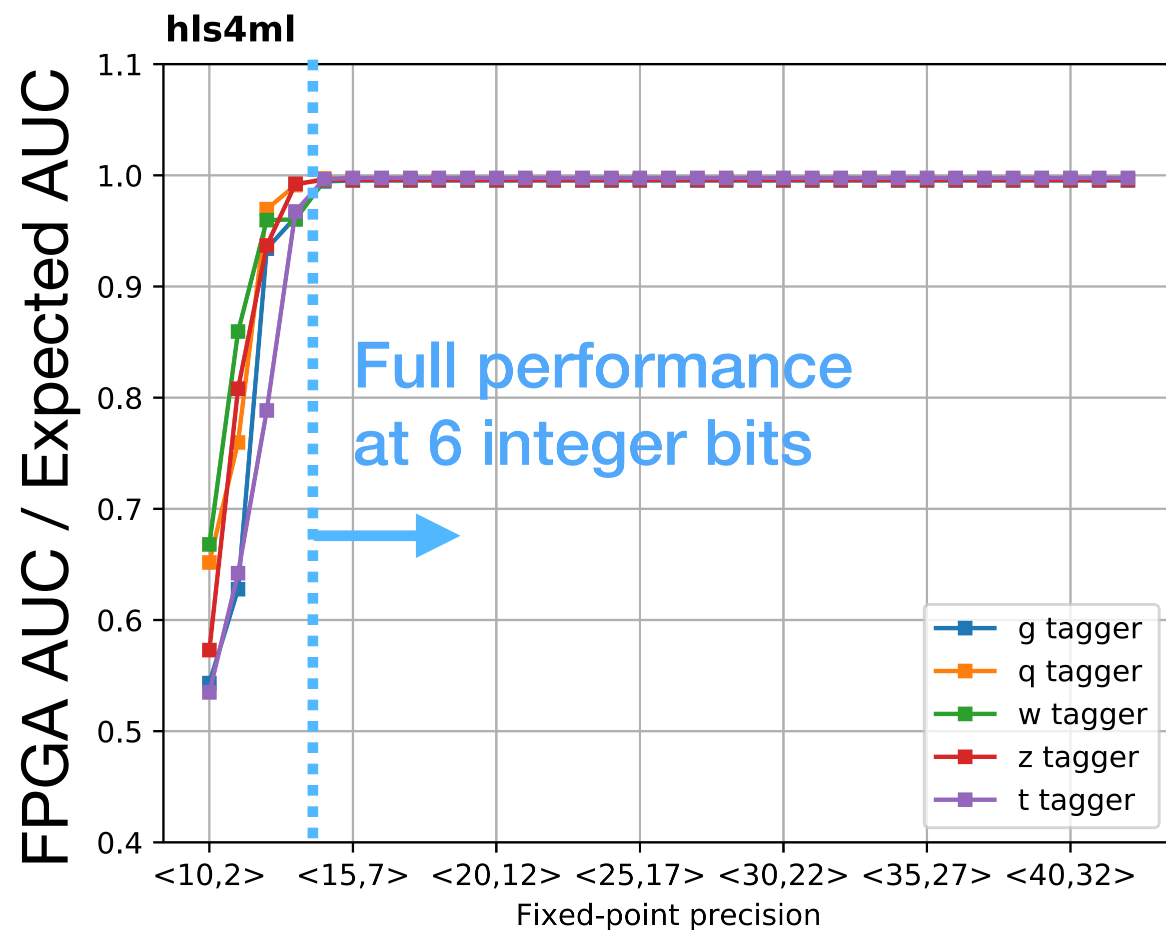
Efficient NN design: quantization

- FPGAs not well suited to floating-point operations: prefer *fixed-point* representation
- FPGA flexibility offers choice over representation: tune to the task
- Quantify the performance of the classifier with the AUC
- Expected AUC = AUC achieved by 32-bit floating point inference of the neural network



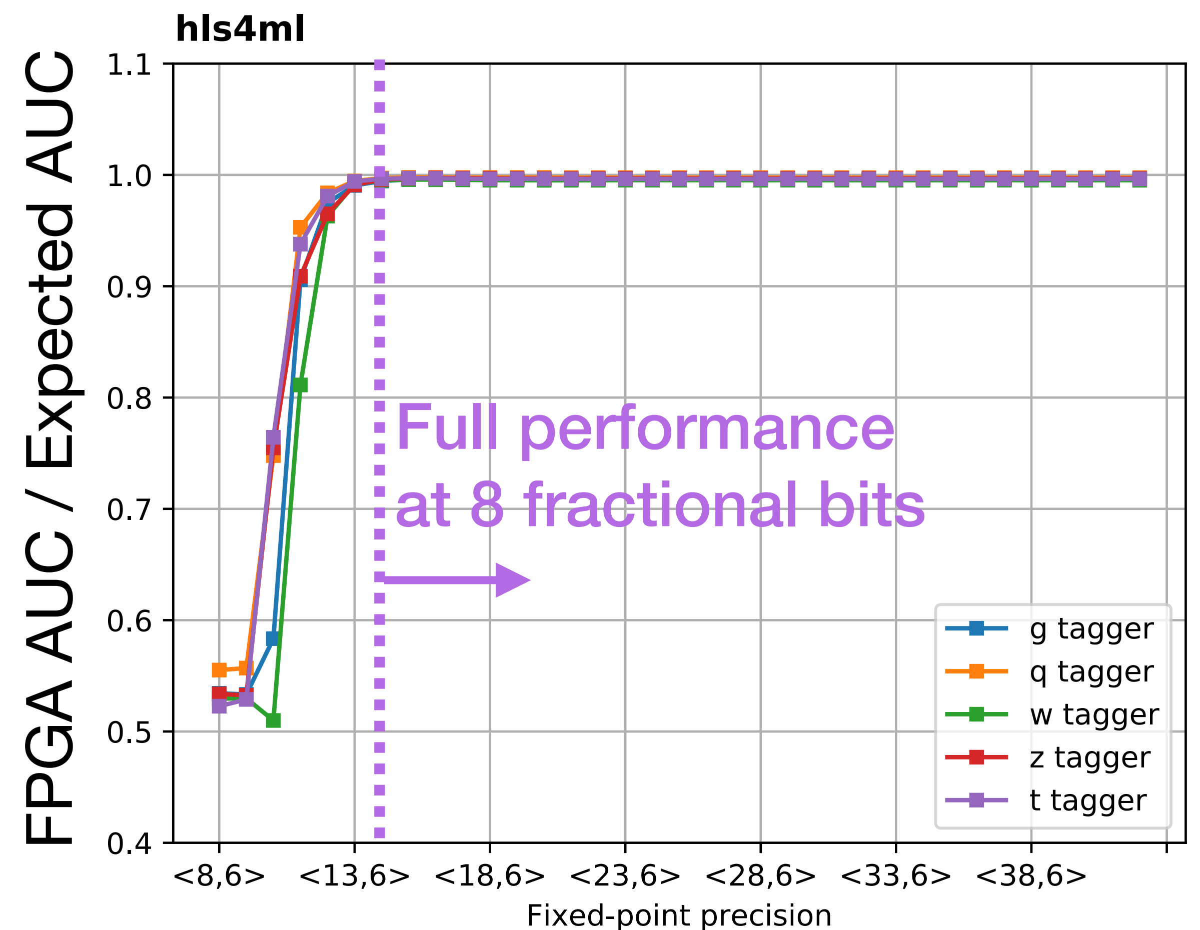
Scan integer bits

Fractional bits fixed to 8

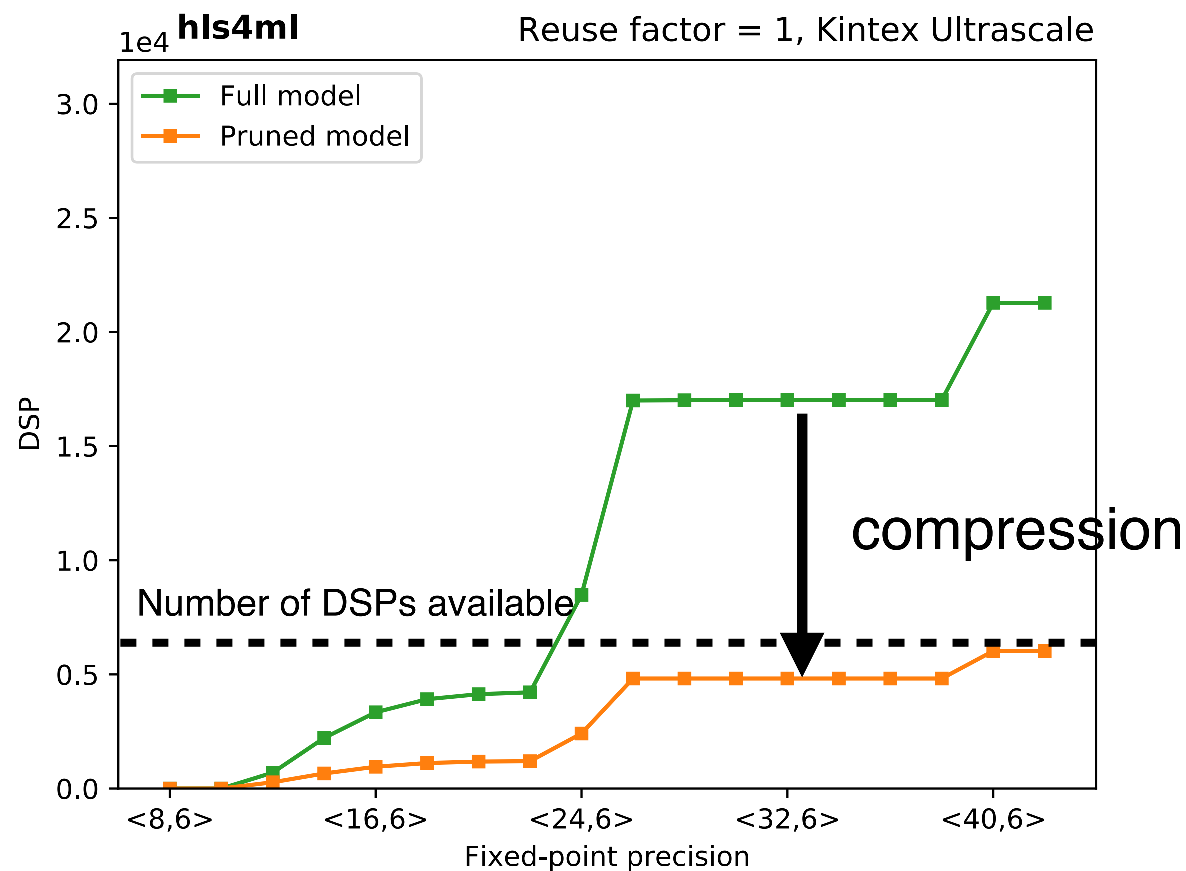


Scan fractional bits

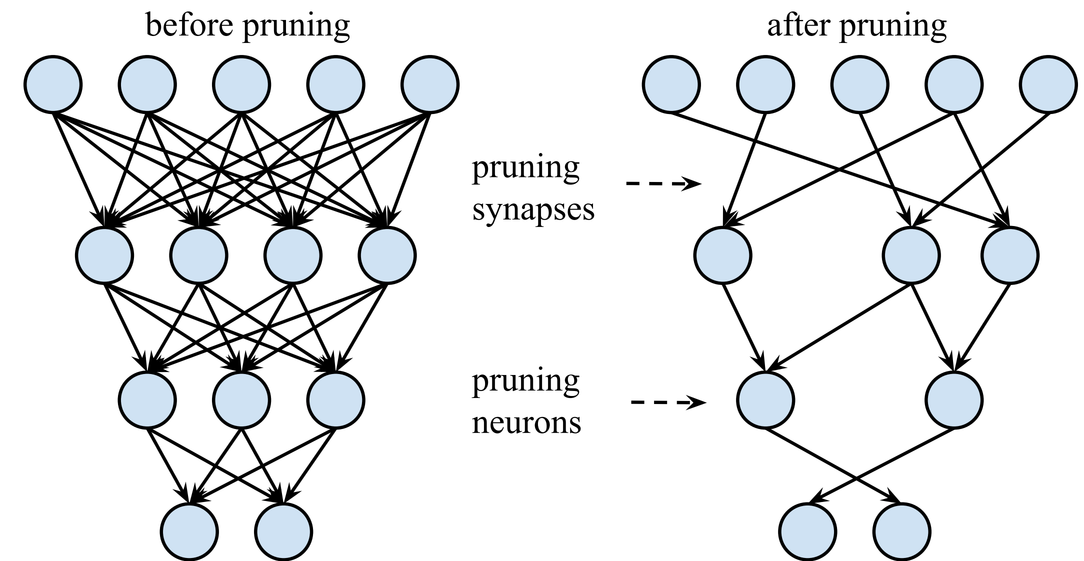
Integer bits fixed to 6



Efficient NN design: compression



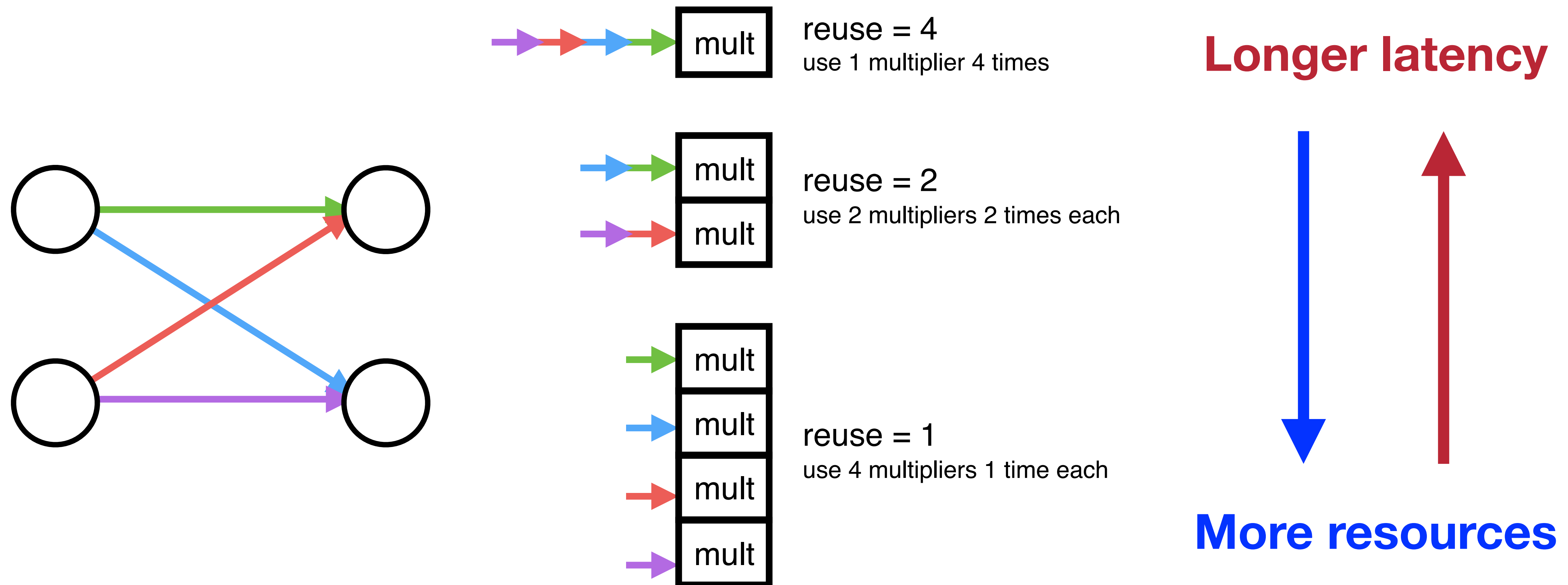
70% compression ~ 70% fewer DSPs



- 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

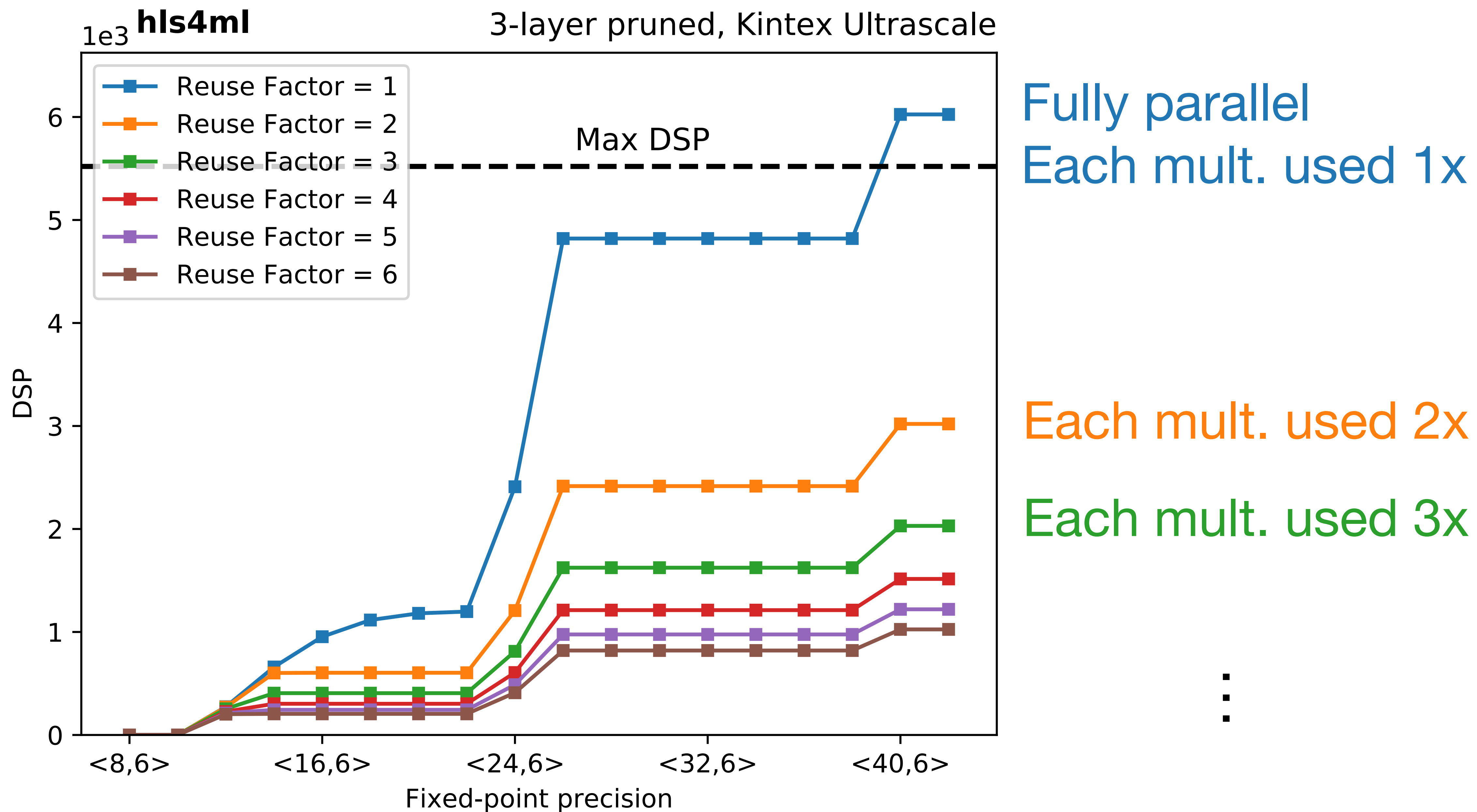
- We perform L1 regularisation to remove small weights which don't contribute much to the final outcome

Efficient NN design: reuse



- 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 saves resources

Parallelization: DSPs usage

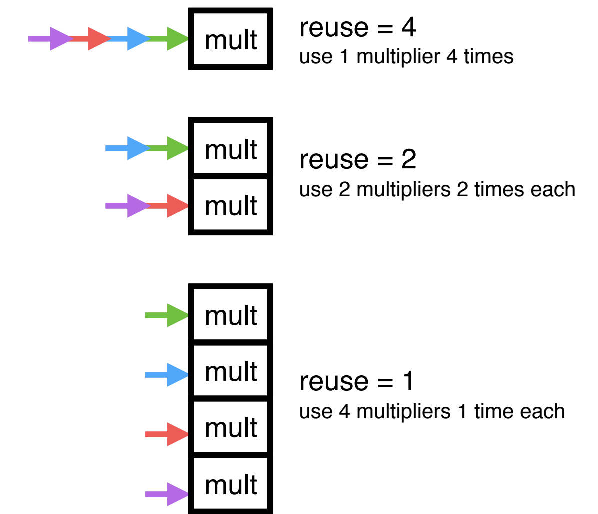


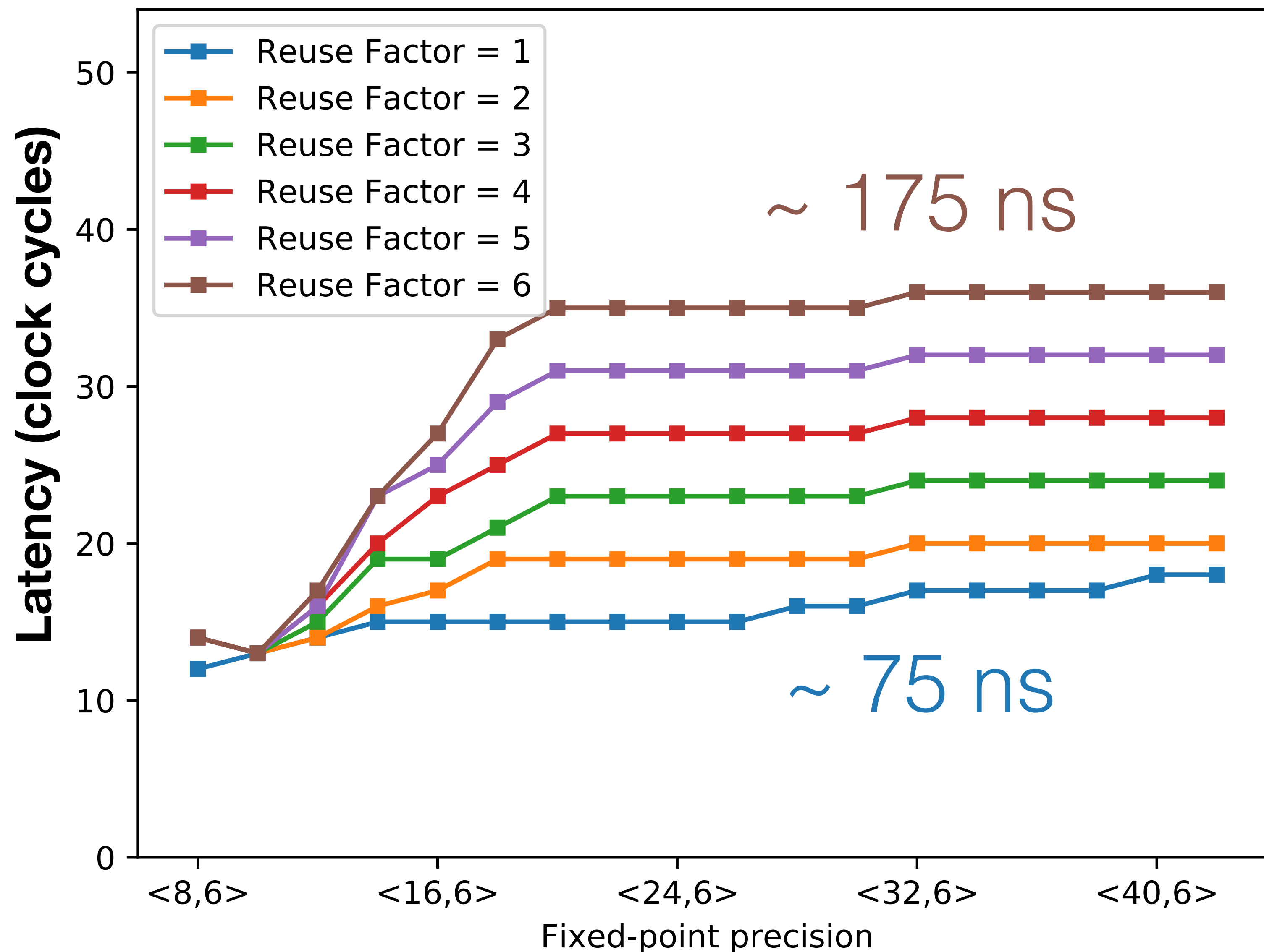
Reuse factor: how much to parallelize operations in a hidden layer

Parallelization: Timing

Latency of layer m

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





Longer latency

Each mult. used 6x

⋮

Each mult. used 3x

⋮

Fully parallel
Each mult. used 1x

More resources

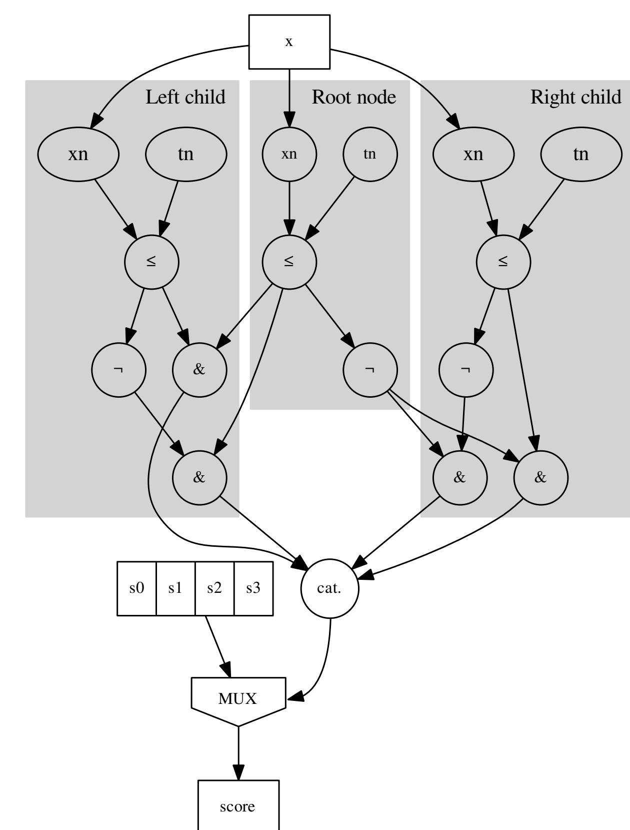
New features under development

- Support for larger networks - with a longer latency budget, can afford much bigger networks with a high *resource reuse*
- Other architectures: convolutional, recurrent
  - Already in package, but working to support larger ones here too
  - Jet images, particle lists
- Binary / Ternary neural networks - very low precision networks to reach very low resource usage
  - 1- or 2-bit activations and weights
- Boosted Decision Trees - (not neural networks), can be very lightweight on resources and latency

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



Conclusions

- Neural network inference on FPGAs can be put directly into the lowest level triggers
- Techniques like compression and quantisation help reduce the footprint without performance loss
- The `hls4ml` package translates trained NN models to FPGA-firmware
  - Inference latencies of 10s - 100s of nanoseconds
  - Powerful, easy-to-control tradeoff between resources and latency/throughput

https://arxiv.org/pdf/1804.06913.pdf https://hls-fpga-machine-learning.github.io/hls4ml/

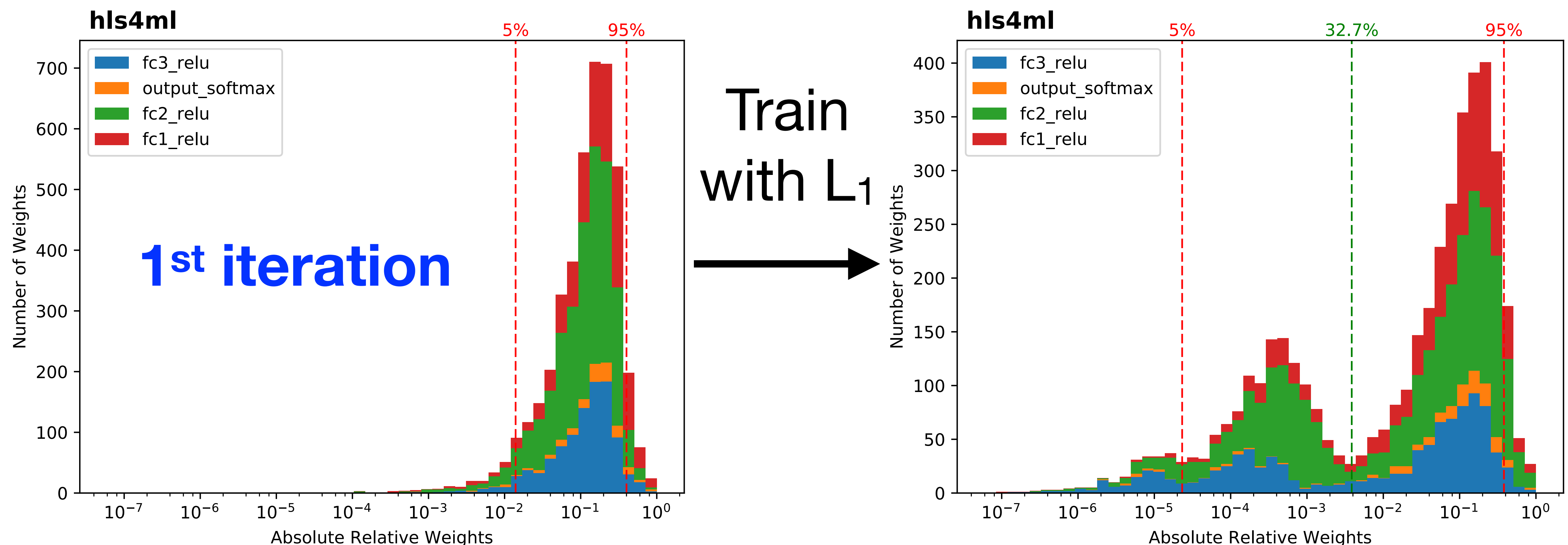
BACKUP

Efficient NN design: compression

- 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

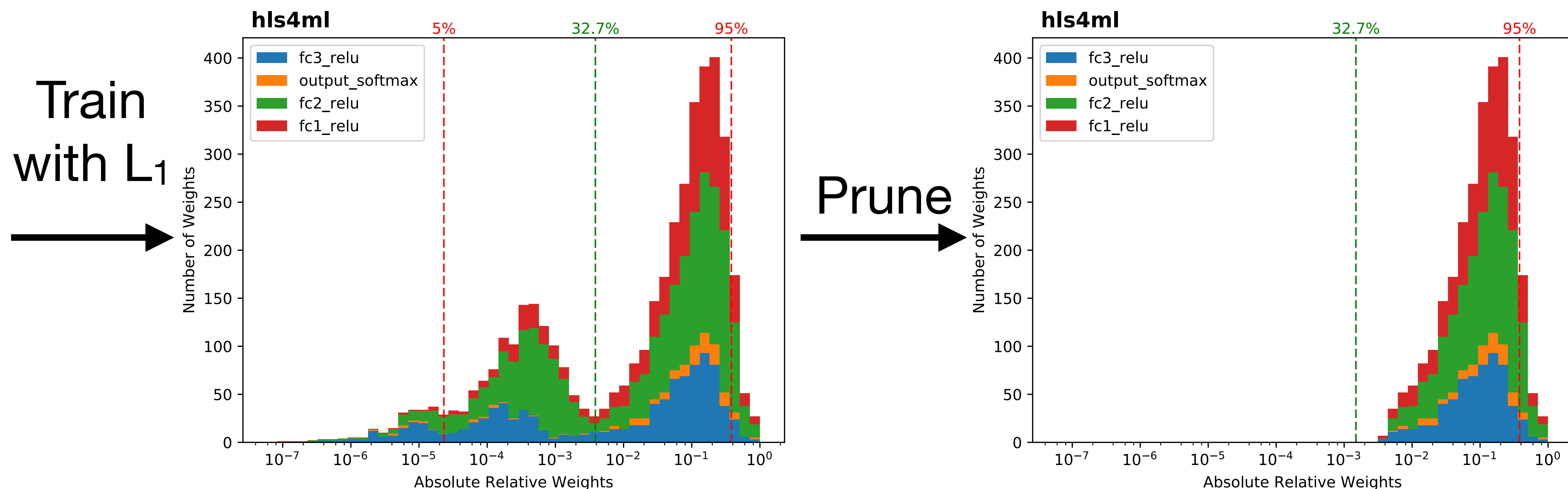


Efficient NN design: compression

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

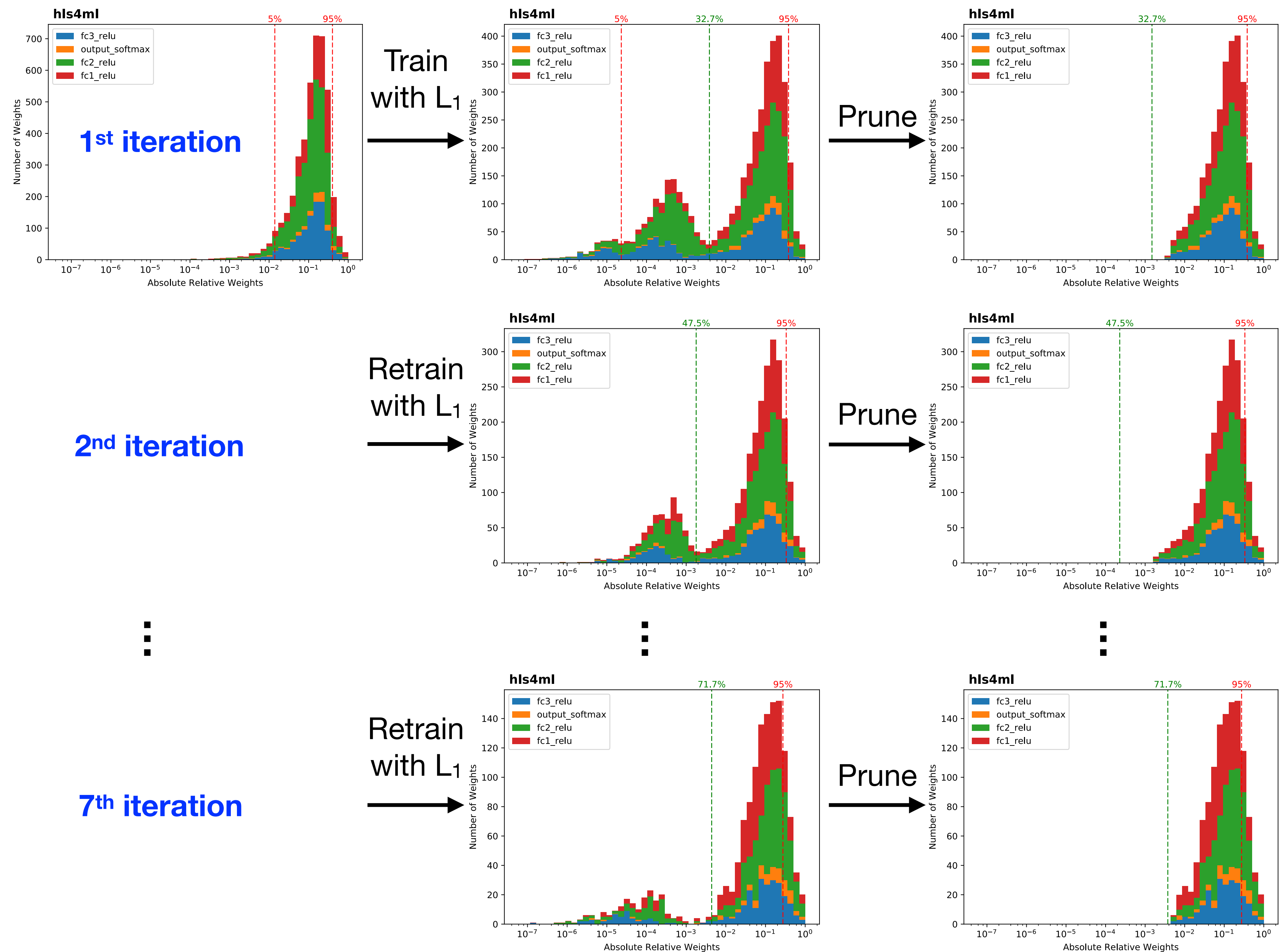
$$L_\lambda(\vec{w}) = L(\vec{w}) + \lambda||\vec{w}_1||$$

  - sort the weights based on the value relative to the max value of the weights in that layer
  - prune weights falling below a certain percentile and retrain



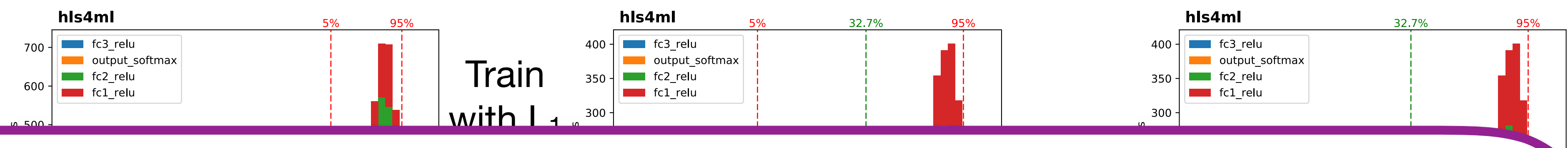
Efficient NN design: compression

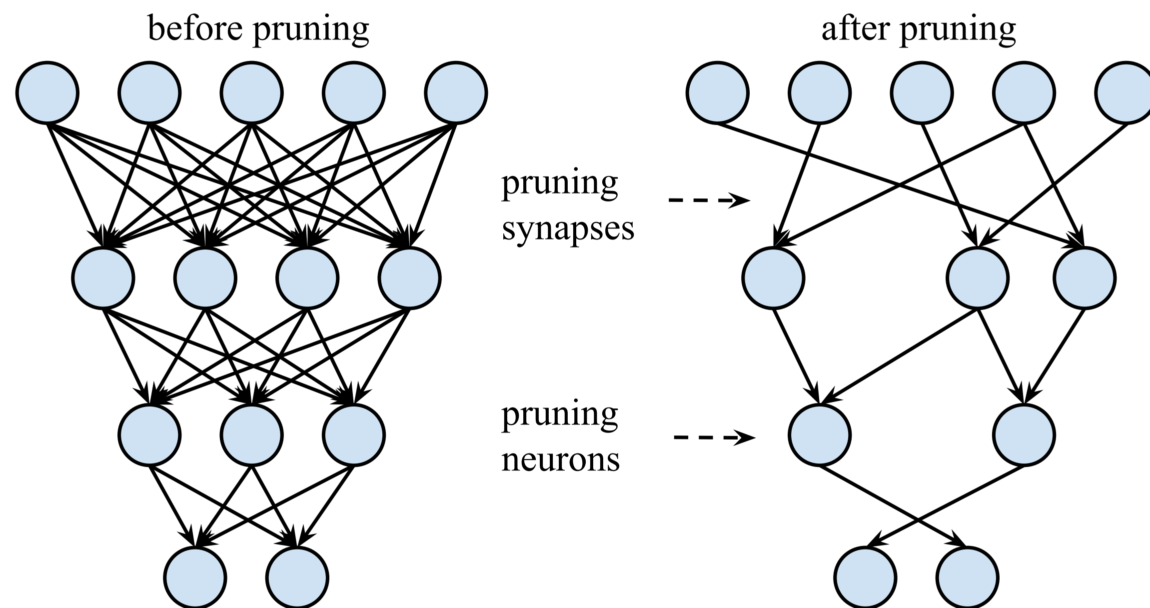
Prune and repeat the train for 7 iterations



Efficient NN design: compression

Prune and repeat the train for 7 iterations

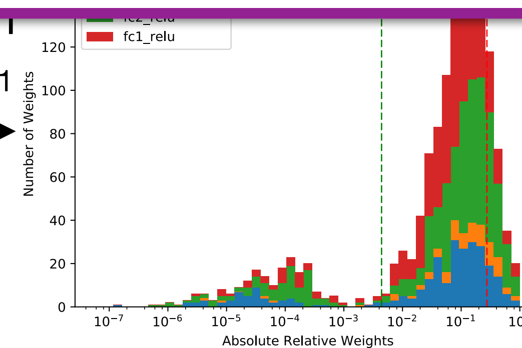




→ 70% reduction of weights and multiplications w/o performance loss

Retrain
with L_1

7th iteration



Prune