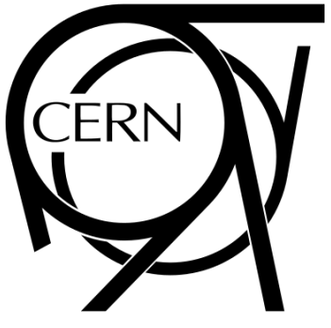
Deep learning on FPGAs for L1 trigger and Data Acquisition

UCL HEP Seminar, January 11, 2019

Javier Duarte, Sergo Jindariani, Ben Kreis, Ryan Rivera, Nhan Tran (Fermilab)
Jennifer Ngadiuba, Maurizio Pierini (CERN)
Edward Kreinar (Hawkeye 360)
Phil Harris, Song Han (MIT)
Zhenbin Wu (University of Illinois at Chicago)



Motivation

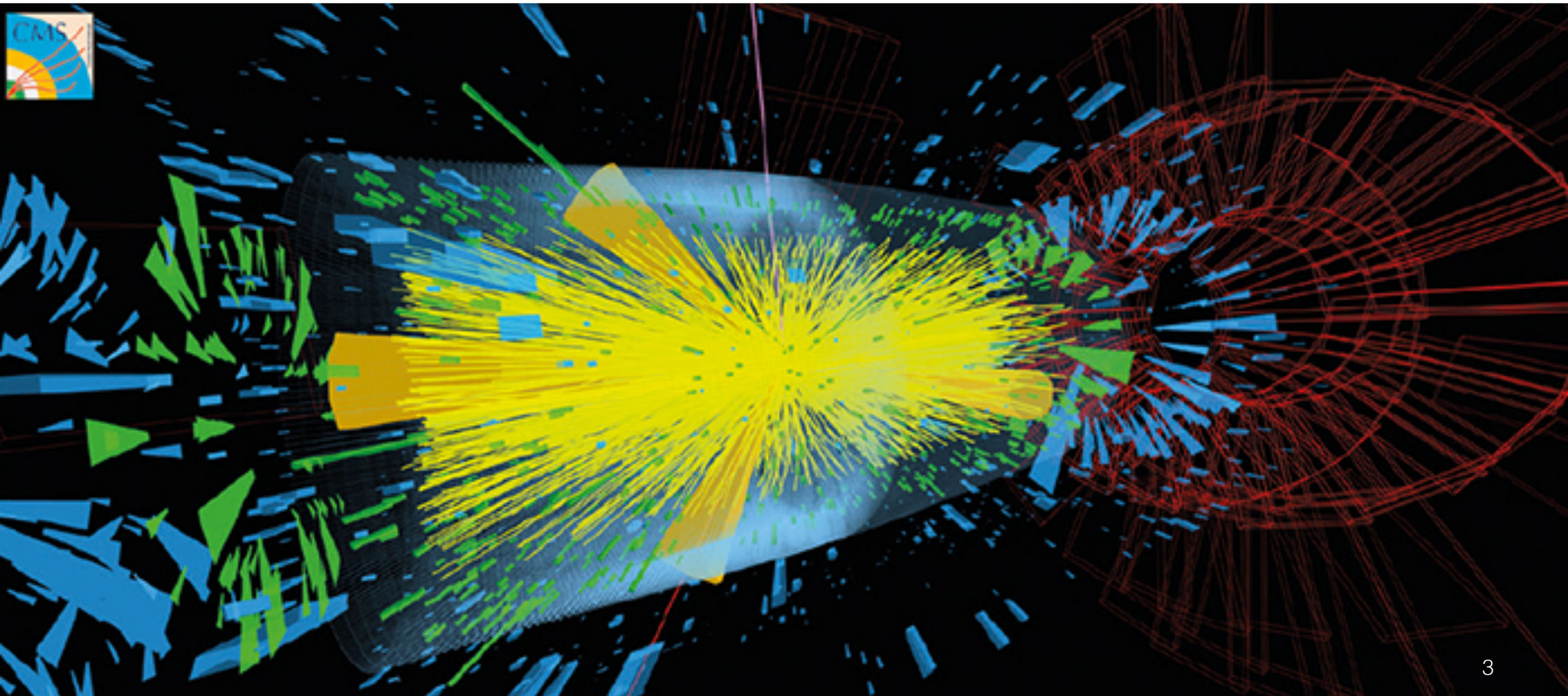
The challenge: triggering at (HL-)LHC

The challenge: triggering at LHC

The LHC big data problem

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 challenge: triggering at HL-LHC

The LHC big data problem

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

"Triggering" = filter events to reduce data rates to manageable levels

Squeeze the beams to increase data rates
→ multiple pp collisions per bunch crossing (pileup)

2016: <PU> ~ 20-50

2017 + Run 3: <PU> ~ 50-80

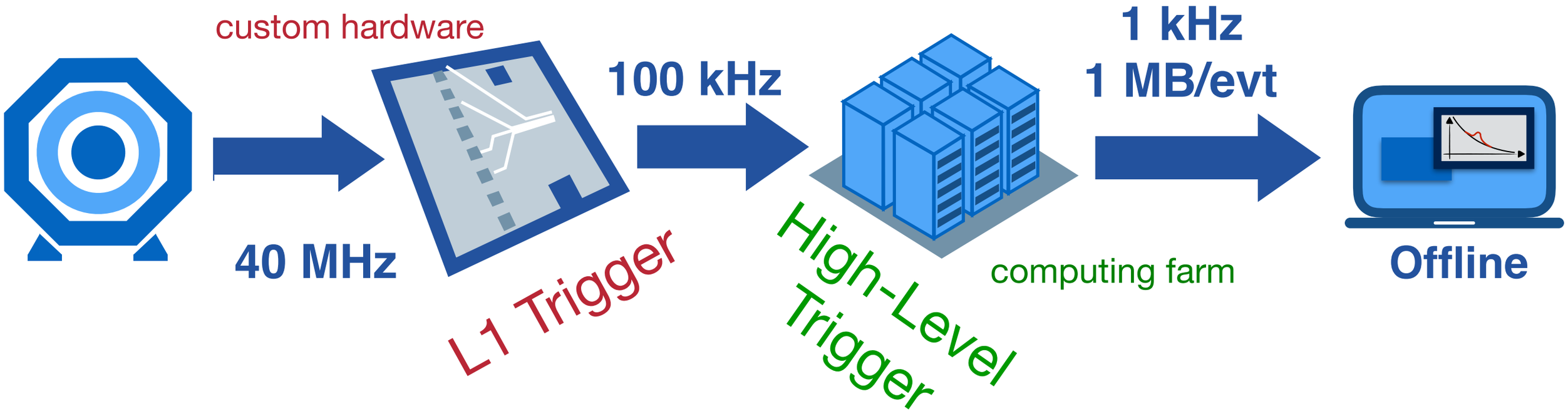
HL-LHC: 140-200

CHALLENGE: maintain physics in increasingly complex collision environment
→ untriggered events lost forever!

Sophisticated techniques needed to preserve the physics!

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

Computing farm for detailed analysis of the full event

Latency O(100 ms)

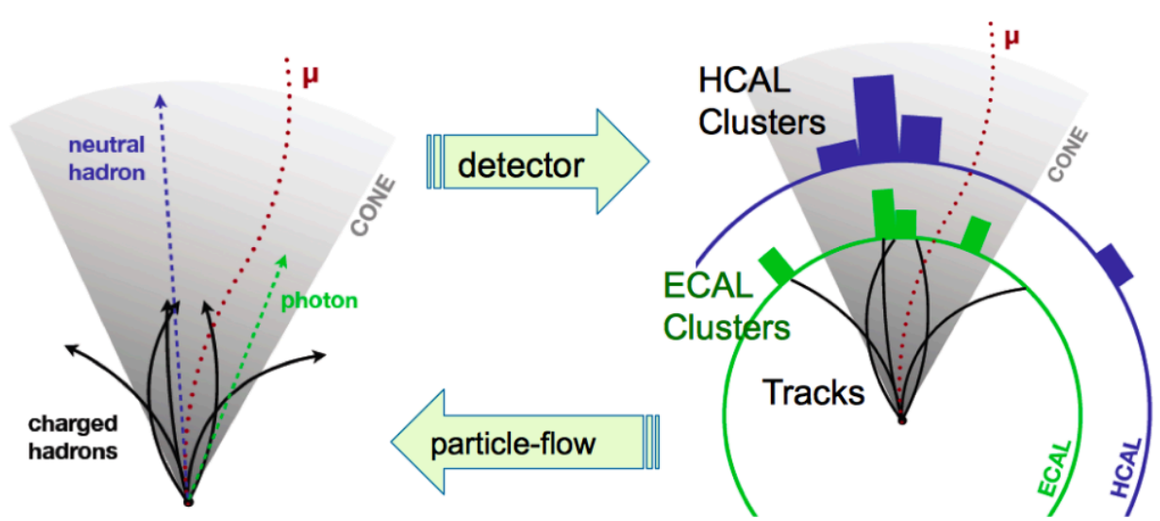
For HL-LHC upgrade: latency and output rates will increase by ~ 3 (ex: for CMS 3.8 → 12.5 μs @ L1)

New trigger algorithms

The detector and its trigger system

Output: trigger primitives
(calo energy clusters, muons, tracks)

Particle-flow algorithm @ L1



Output:
particle candidates
(prompt vs pileup particles)

Trigger decision

CMS-TDR-017, JINST 12 (2017) P10003

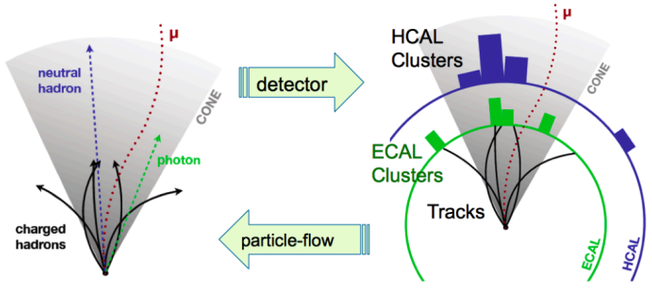
New trigger algorithms

The detector and its trigger system

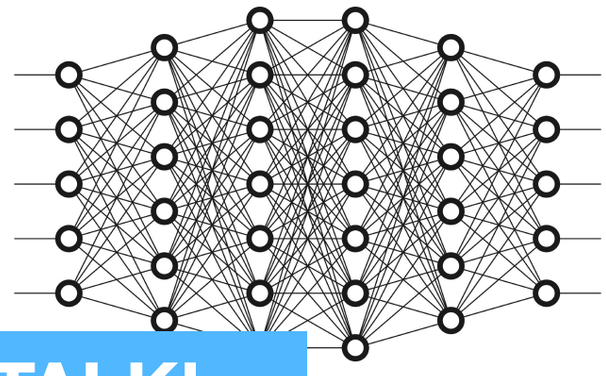
Output: trigger primitives
(calo energy clusters, muons, tracks)

Particle-flow algorithm @ L1

Output: particle candidates



Machine learning



THIS TALK!

Trigger decision

[CMS-TDR-017](), [JINST 12 (2017) P10003]()

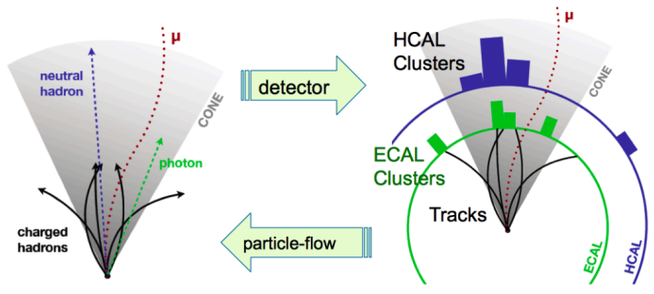
New trigger algorithms

The detector and its trigger system

Output: trigger primitives
(calo energy clusters, muons, tracks)

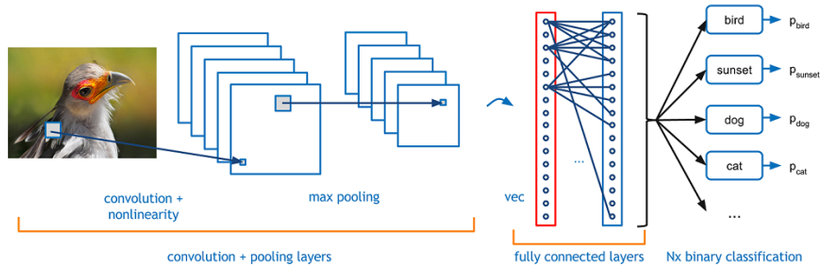
Particle-flow algorithm @ L1

Output: particle candidates



[CMS-TDR-017](), [JINST 12 (2017) P10003]()

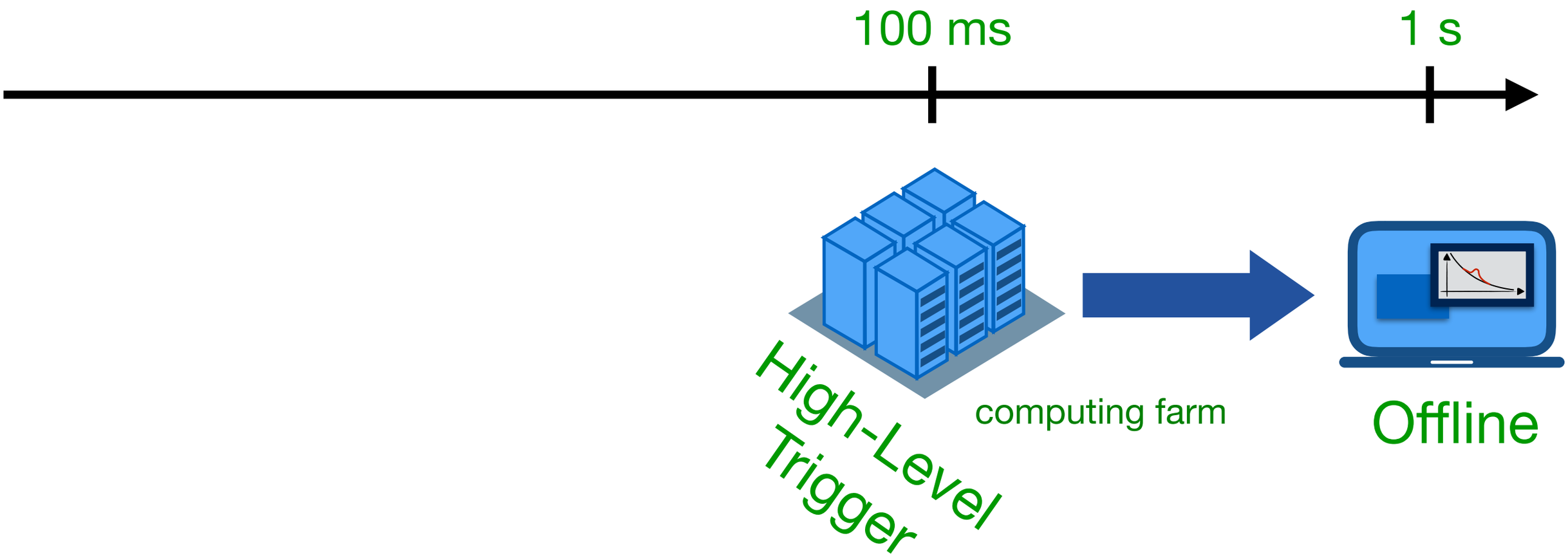
Machine learning



THIS TALK!

Trigger decision

The latency landscape @ LHC

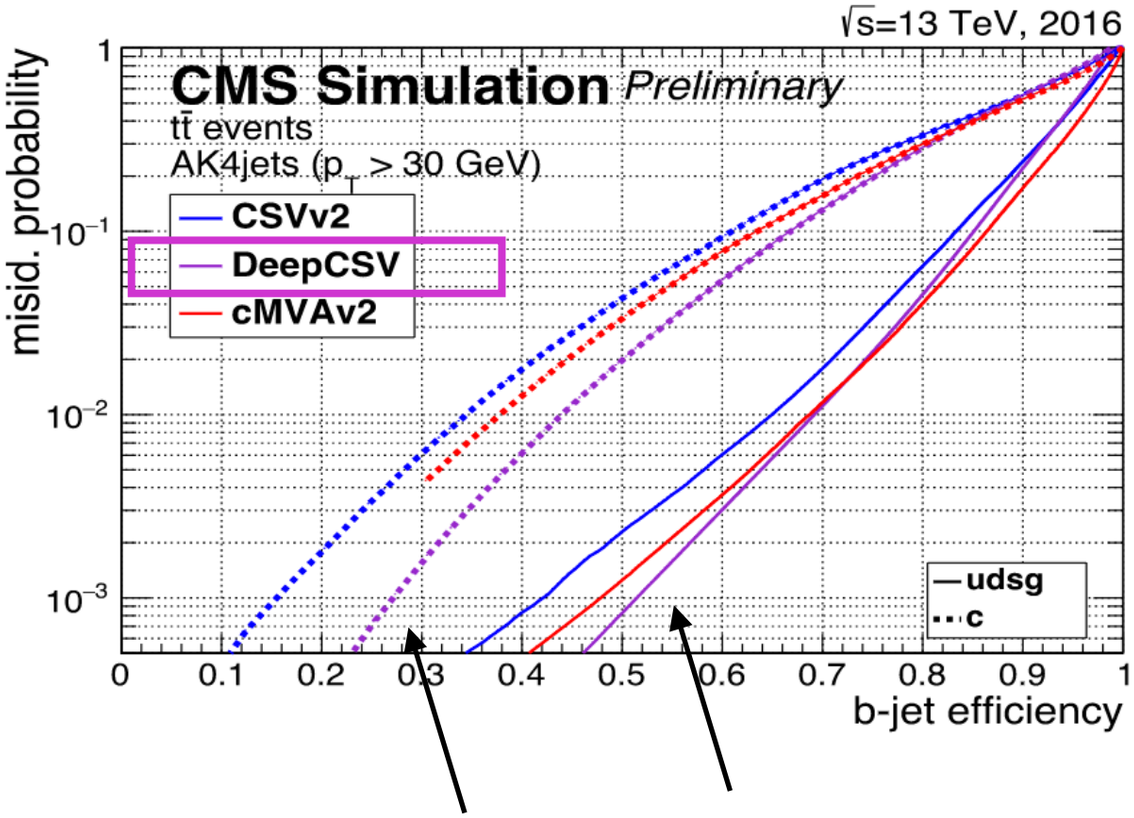


ML methods typically employed in offline analysis or longer latency trigger tasks

Many successes in HEP: identification of b-quark jets, Higgs candidates, particle energy regression, analysis selections, ….

The latency landscape @ LHC

ex, identification of b-quark jets

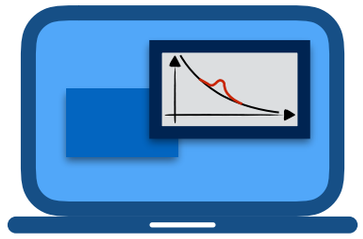


Deep neural network based on high-level features

both offline and @ HLT

High-Level Trigger

computing farm



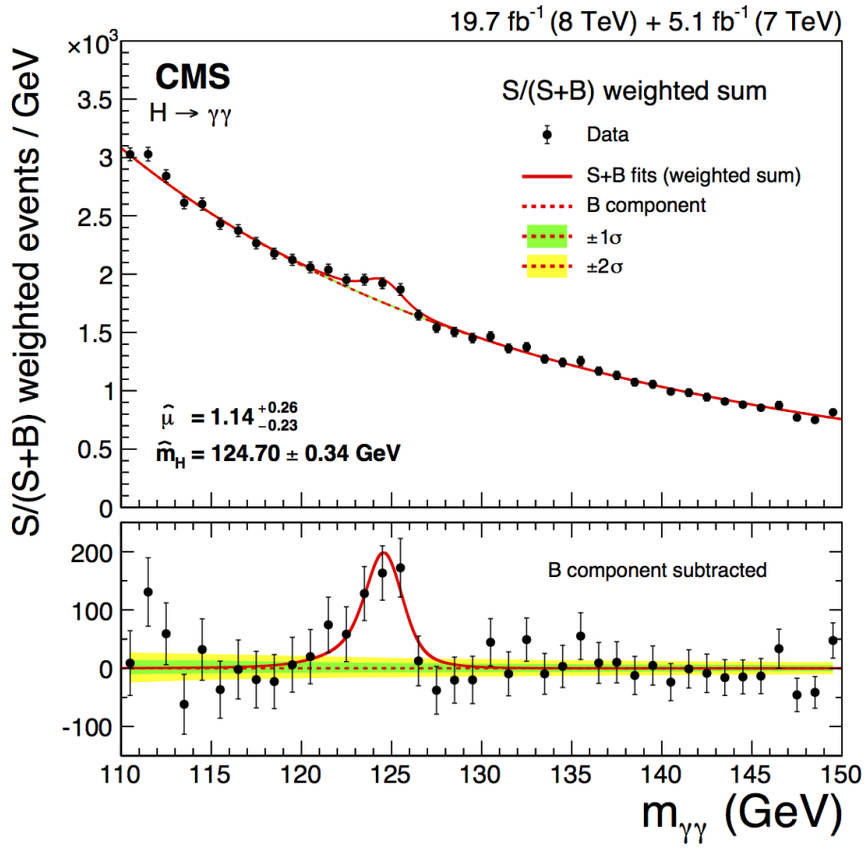
Offline

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

Many successes in HEP: identification of b-quark jets, Higgs candidates, particle energy regression, analysis selections, ….

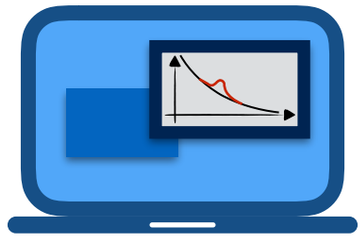
The latency landscape @ LHC

ex, Higgs discovery



High-Level Trigger

computing farm



Offline

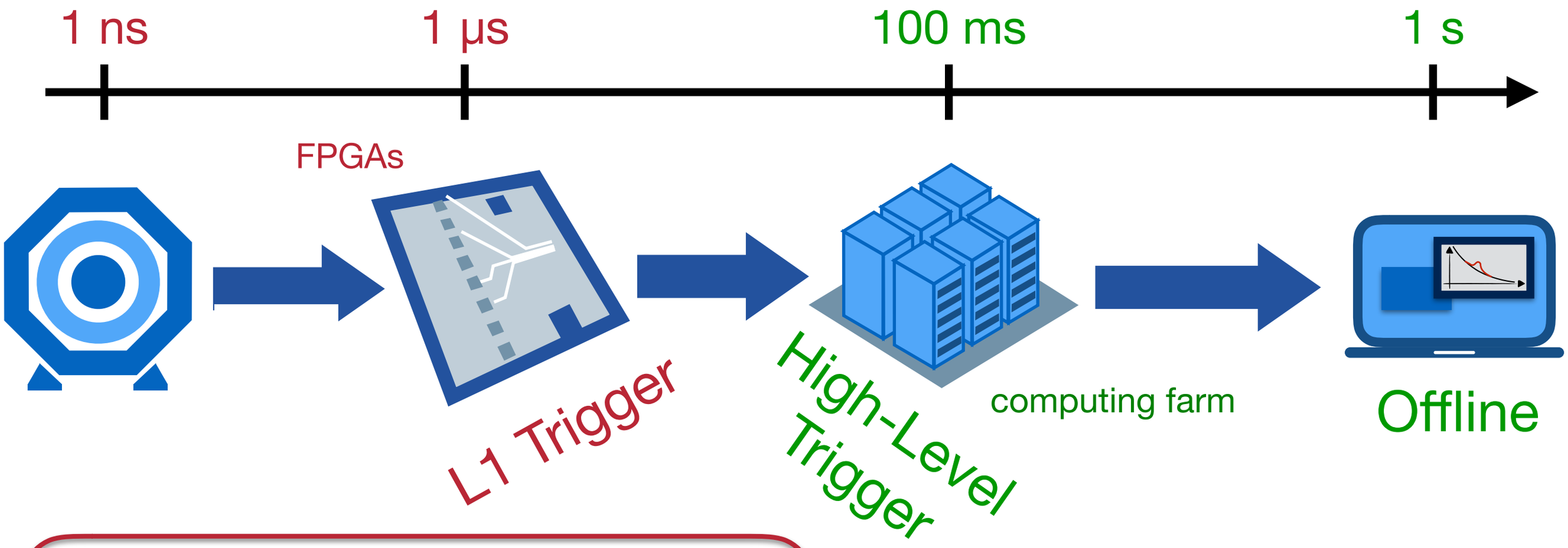
ML methods typically employed in offline analysis or longer latency trigger tasks

Many successes in HEP: identification of b-quark jets, Higgs candidates, particle energy regression, analysis selections, ….

ML algorithms used offline for

- improving Higgs mass resolution with particle energy regression
- enhancing signal/background discrimination

The latency landscape @ LHC



Exploration of ML algorithms in low-latency, real-time processing has just begun!

What can we do in < us on one FPGA?

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

Many successes in HEP: identification of b-quark jets, Higgs candidates, particle energy regression, analysis selections, ….

Muon reconstruction @ L1

First implementation of a ML algo for CMS L1 trigger on FPGAs [*]

A BDT is used to improve the momentum of muons in the forward region of the detector

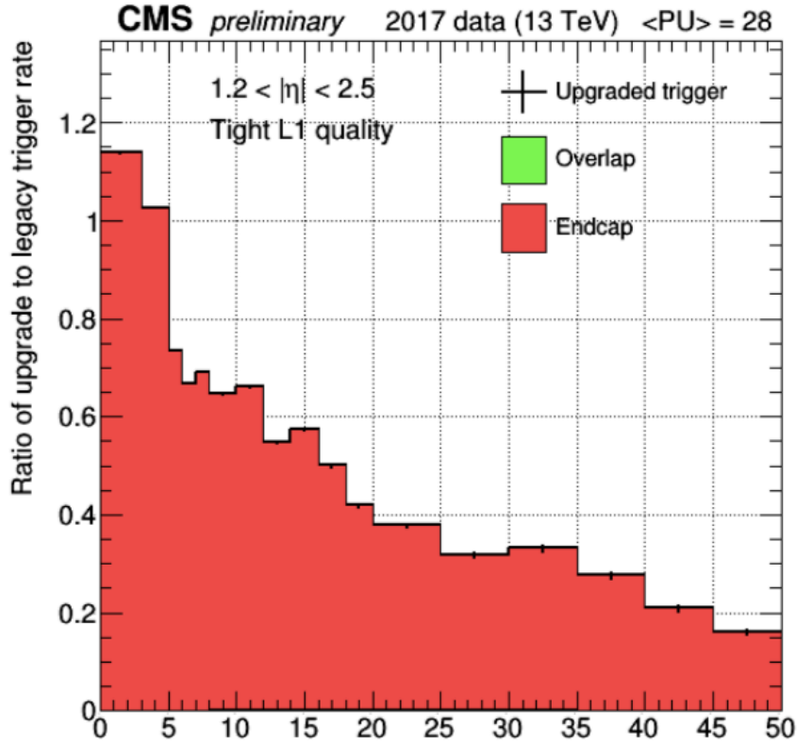
based on curvature angles in the magnetic fields ($\Delta\phi$,$\Delta\theta$) and few other variables

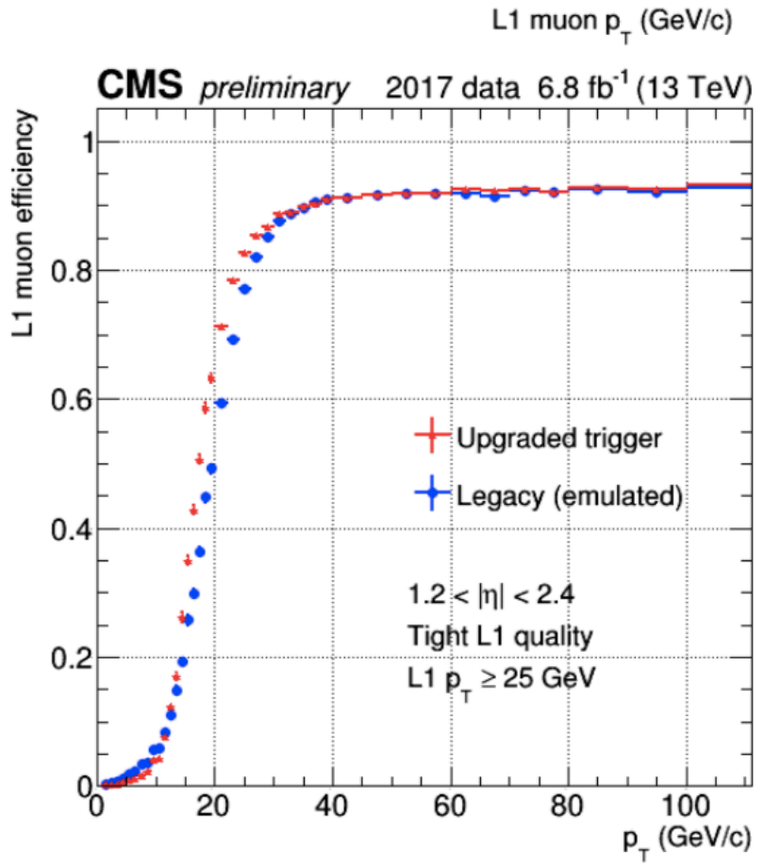
Prediction of BDT for every possible input stored into pre-computed 1.2 GB Look-Up Table (LUT) on FPGA

Achieved reduction of background rates by factor 3 w/o efficiency losses

Usage of LUTs does not scale nicely with ML algo complexity → quickly use all resources

Can we improve this approach?





[*] http://cds.cern.ch/record/2290188/files/CR2017_357.pdf?version=1

Machine Learning & FPGAs

The rise of specialized hardware for ML

GPUs excell at parallel processing

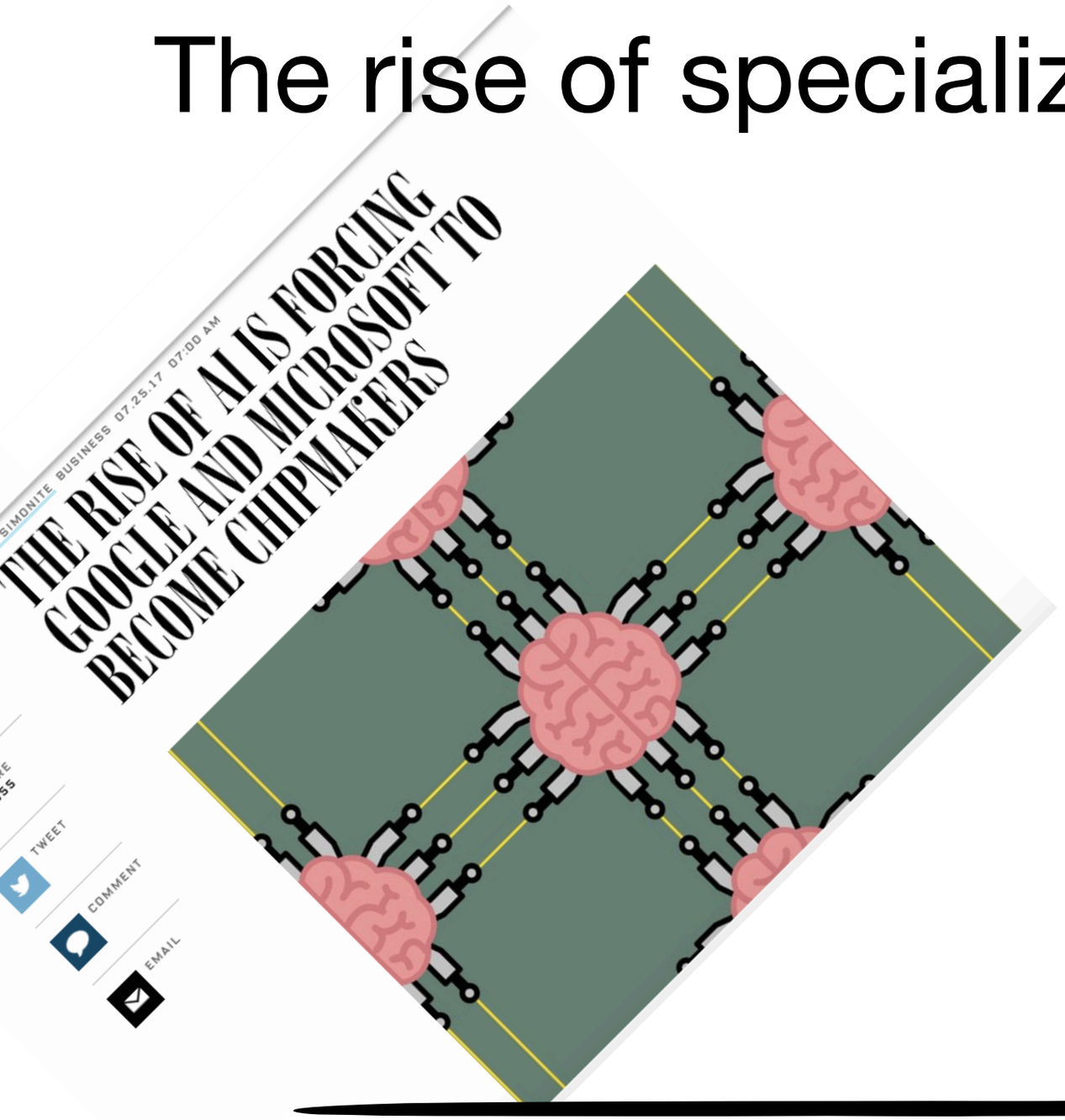
Good for complex NN training of huge amount of data!

Notoriously power-hungry

Sub-optimal for fast and simple NN inference

Optimize resources utilization for less intensive tasks

The rise of specialized hardware for ML



GPUs excell at parallel processing

Good for complex NN training of huge amount of data!

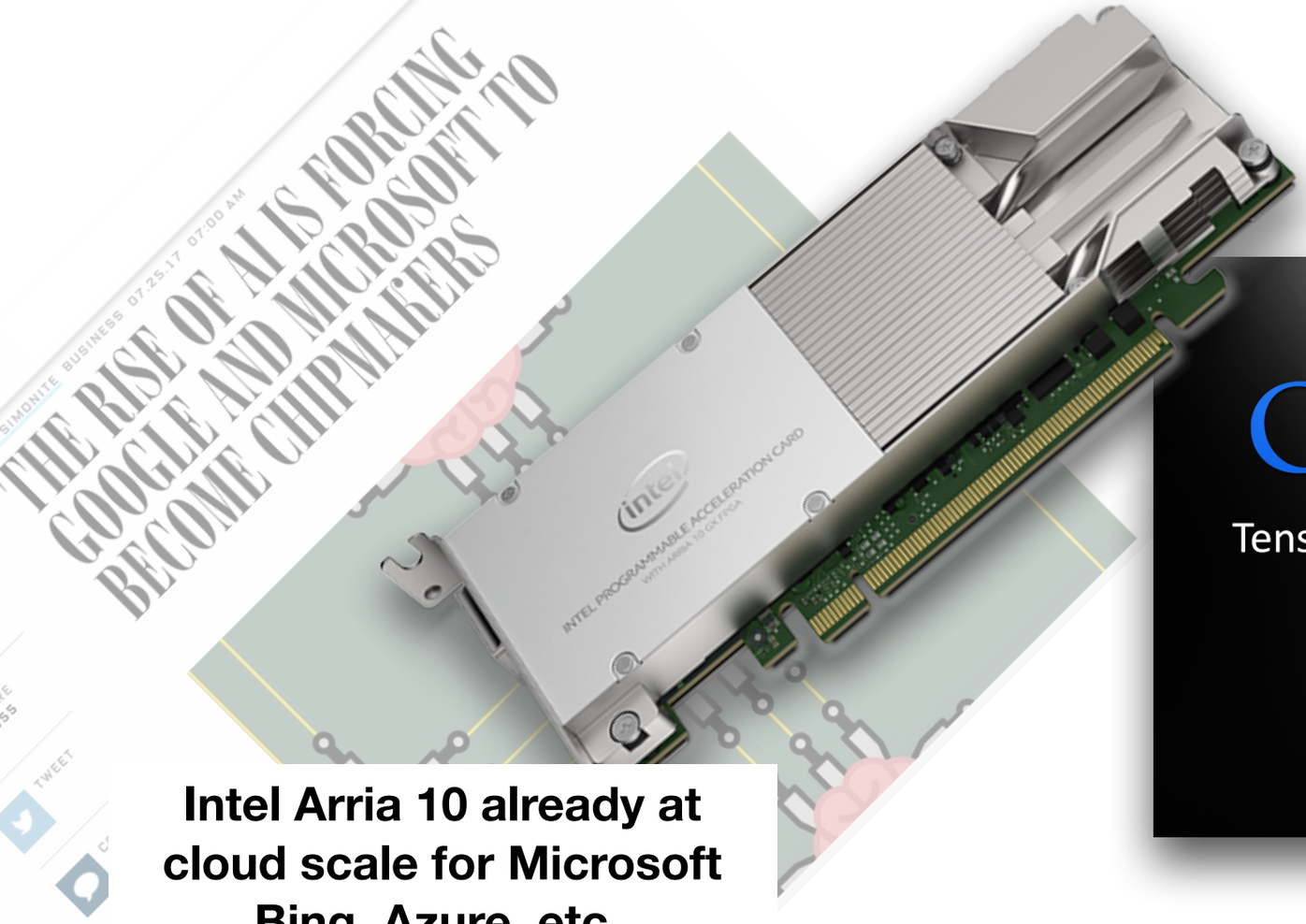
Notoriously power-hungry

Sub-optimal for fast and simple NN inference

Optimize resources utilization for less intensive tasks

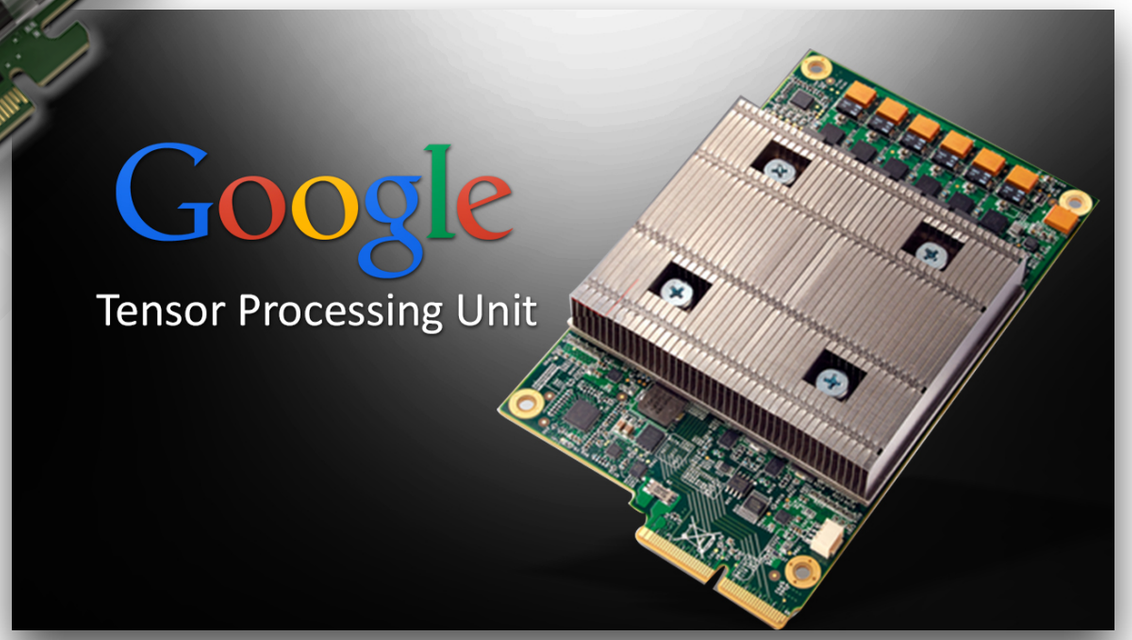
New developments in FPGAs and ASICs making #RealTimeAI possible!

The rise of specialized hardware for ML



Intel Arria 10 already at cloud scale for Microsoft Bing, Azure, etc..

Custom AI hardware for Google on the cloud

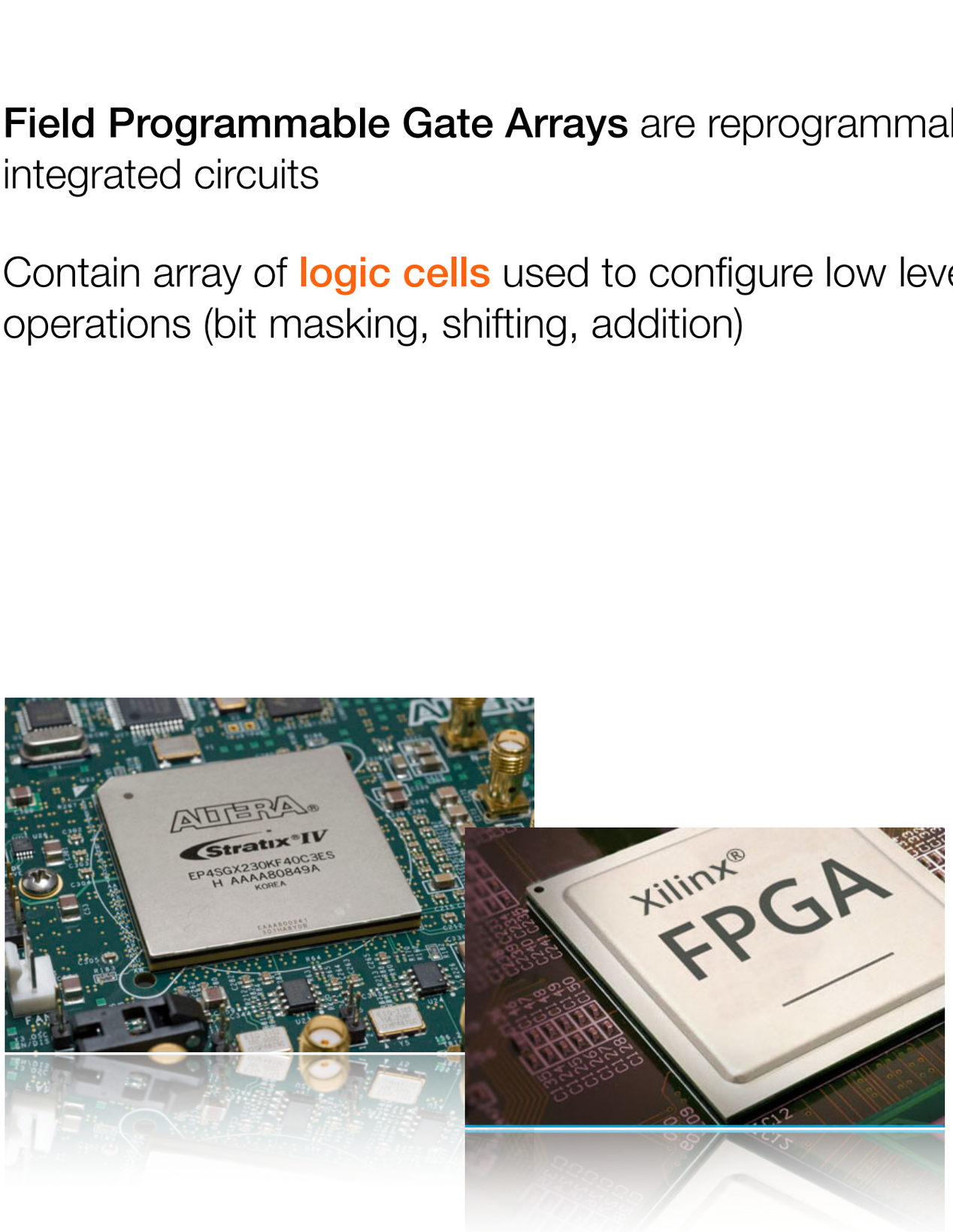


New developments in FPGAs and ASICs making #RealTimeAI possible!

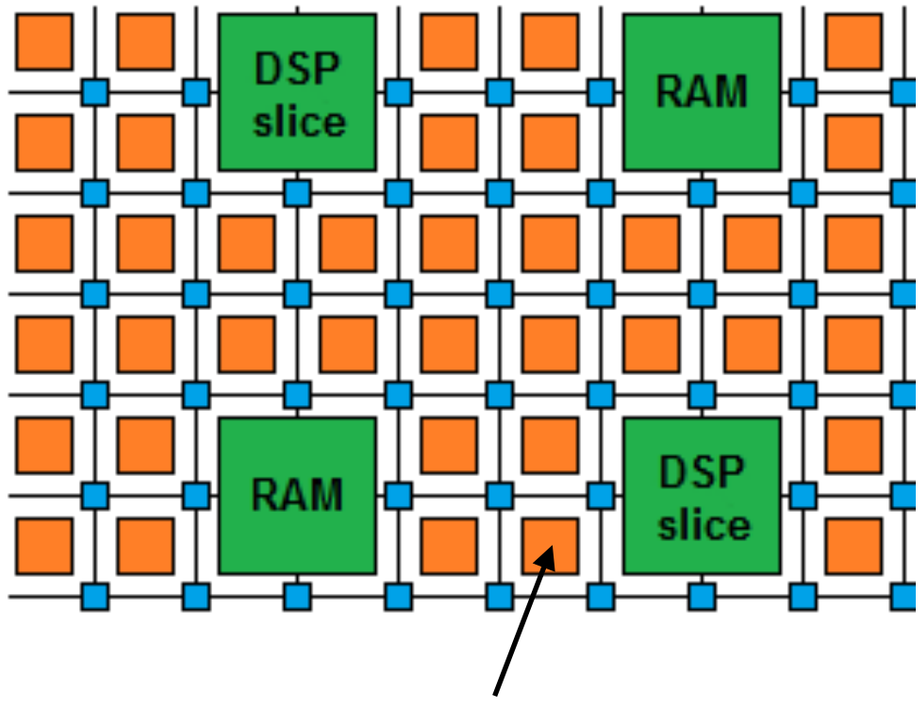
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

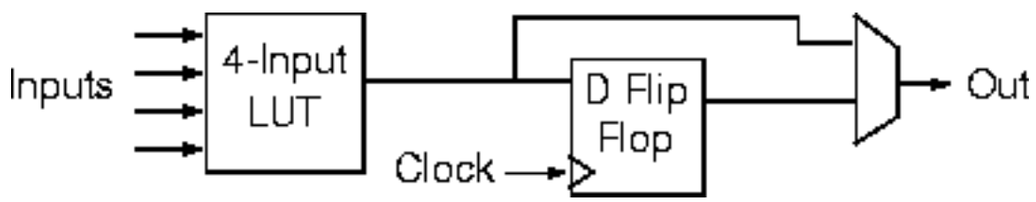
Contain array of **logic cells** used to configure low level operations (bit masking, shifting, addition)



FPGA diagram



Logic cell



Look-up table

(logic)

Flip-flop

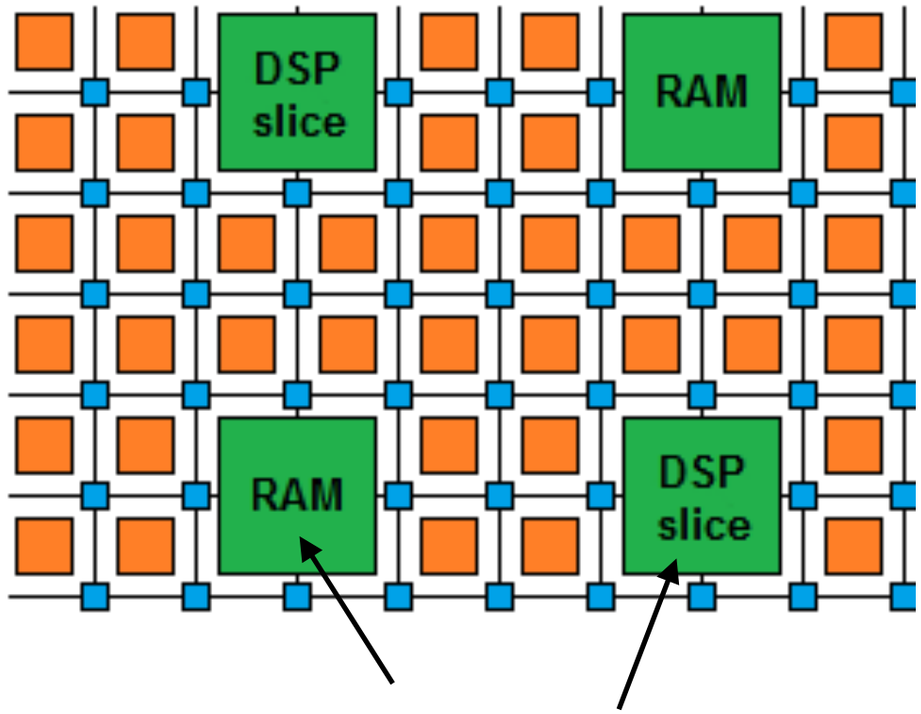
(registers)

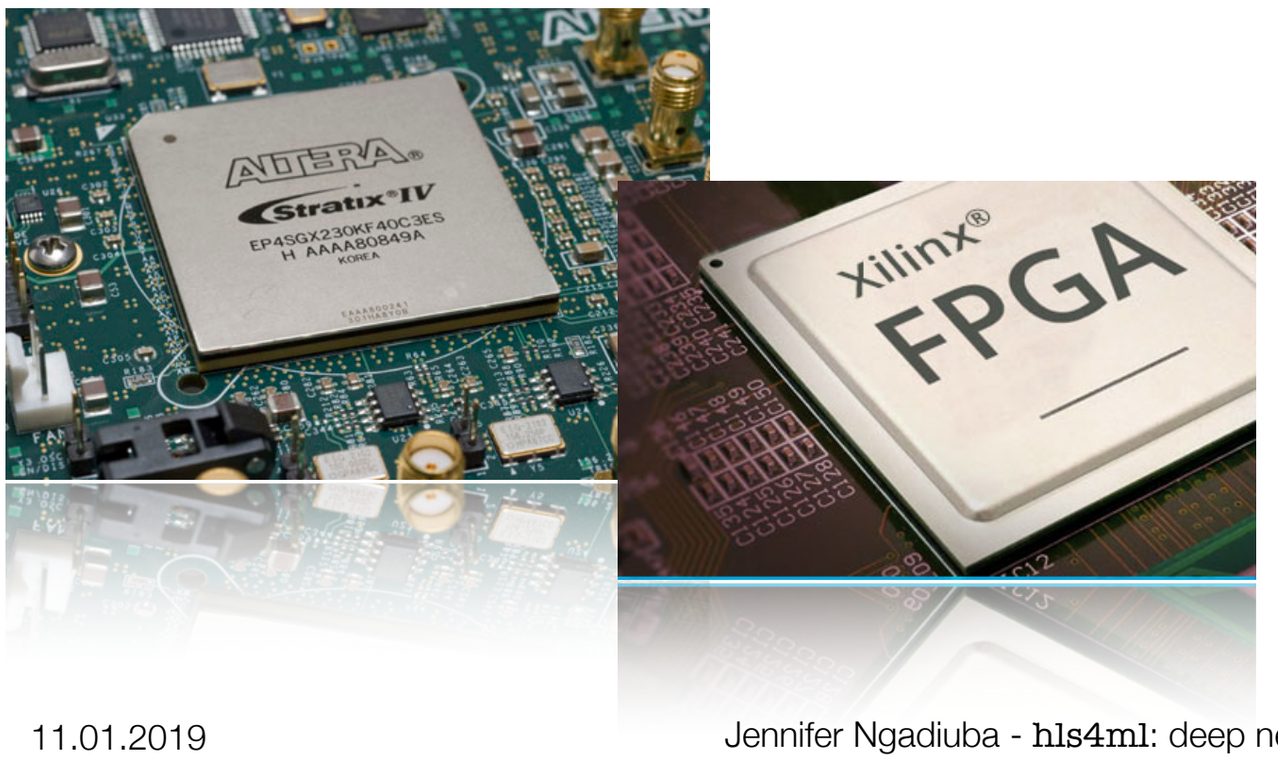
What are FPGAs?

Field Programmable Gate Arrays are reprogrammable integrated circuits

Contain array of **logic cells** used to configure low level operations (bit masking, shifting, addition)

FPGA diagram





Also contain embedded components:

Digital Signal Processors (DSPs):
logic units used for multiplications

Random-access memories (RAMs):
embedded memory elements

What are FPGAs?

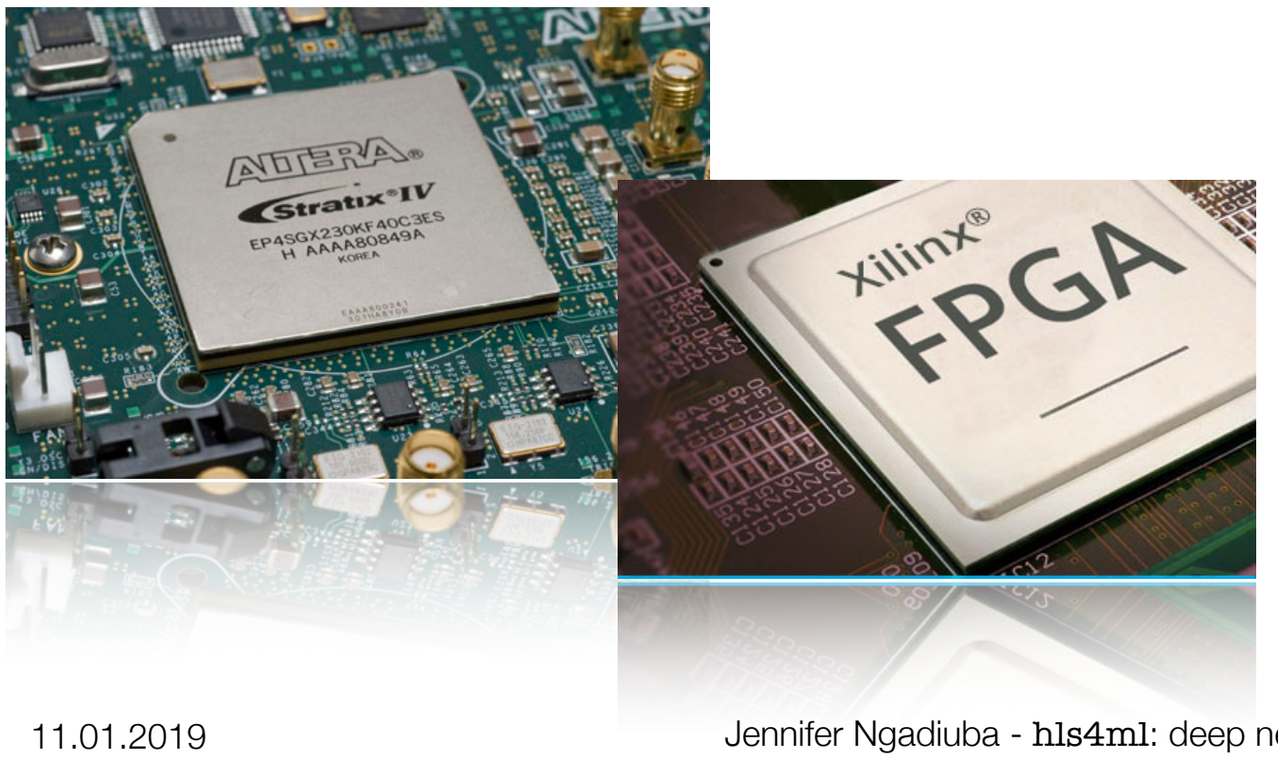
Field Programmable Gate Arrays are reprogrammable integrated circuits

Contain array of **logic cells** embedded with **DSPs**, **BRAMs**, etc.

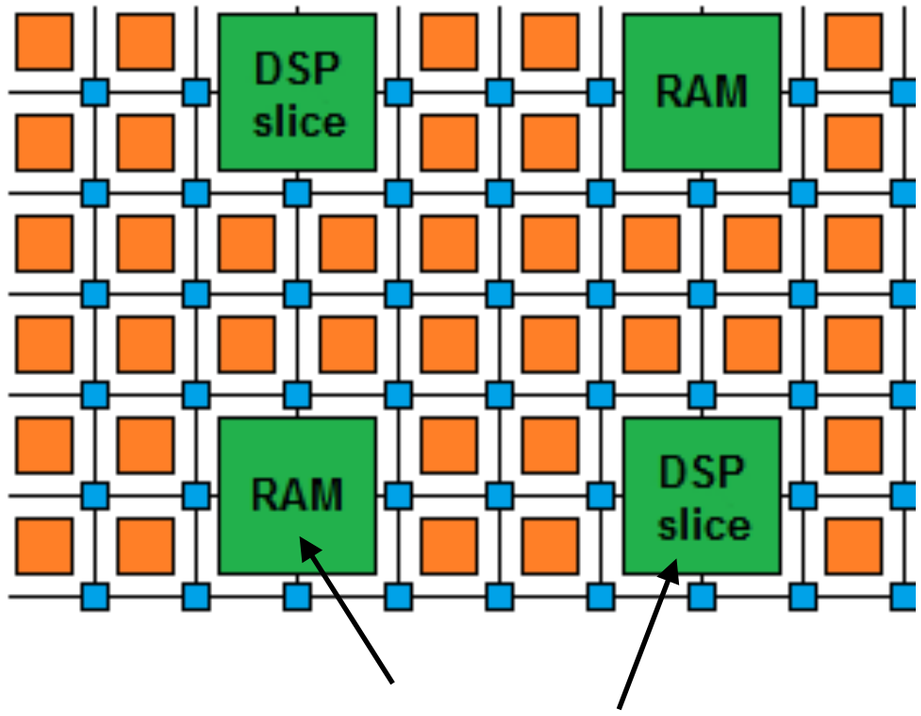
High speed input/output to handle the large bandwith

Support **highly parallel** algorithm implementations

Low power (relative to CPU/GPU)



FPGA diagram



Digital Signal Processors (DSPs): logic units used for multiplications

Random-access memories (RAMs): embedded memory elements

Flip-flops (FF) and look up tables (LUTs) for additions

How are FPGAs programmed?

Hardware Description Languages

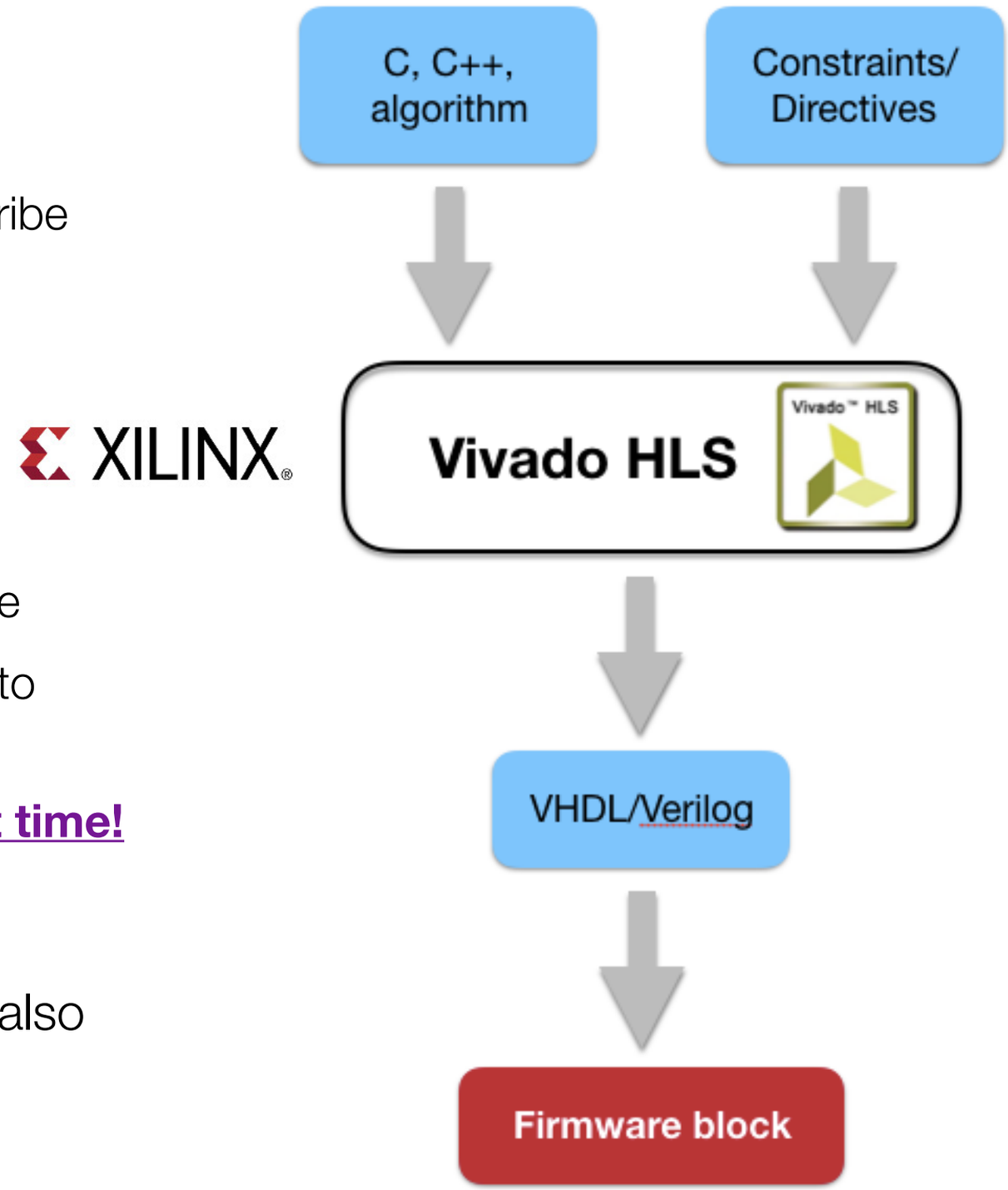
HDLs are programming languages which describe electronic circuits

High Level Synthesis

generate HDL from more common C/C++ code

pre-processor directives and constraints used to optimize the timing

drastic decrease in firmware development time!



We use here **Xilinx Vivado HLS [*]**, but plan also Intel/Altera Quartus HLS

[*] *https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf*

Machine learning & FPGAs

FPGAs used broadly across particle physics experiments for DAQ and trigger development

becoming more accessible thanks to use of HLS

the hardware structures maps nicely onto ML architectures

early adoption of ML algorithms for L1 trigger uses BDT

Extensive literature on deep learning in FPGAs

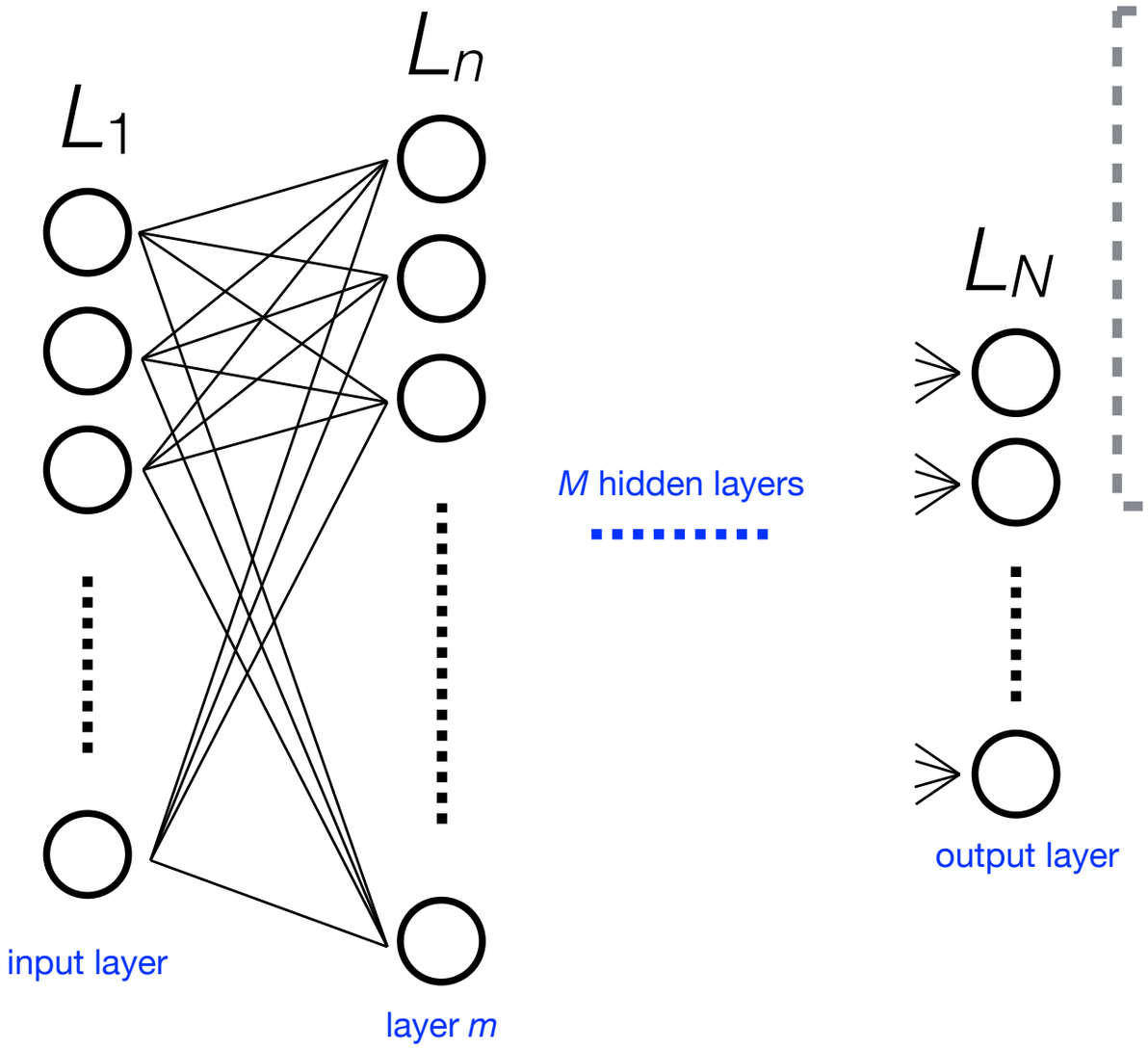
mainly targeting acceleration of large networks, relaxed latency constraints

support of ML architectures in `Keras/TensorFlow`, `Caffe`, `Torch`

This is the first dedicated study on inference of deep neural networks in FPGAs for low-latency application

arxiv.1804.06913

Neural network inference



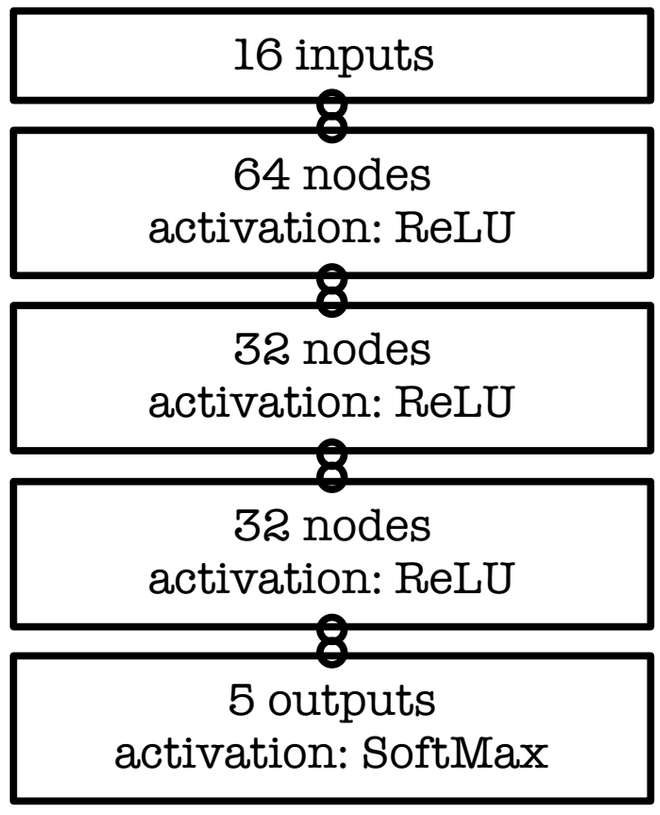
$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1} + \mathbf{b}_n)$$

activation function — precomputed and stored in BRAMs

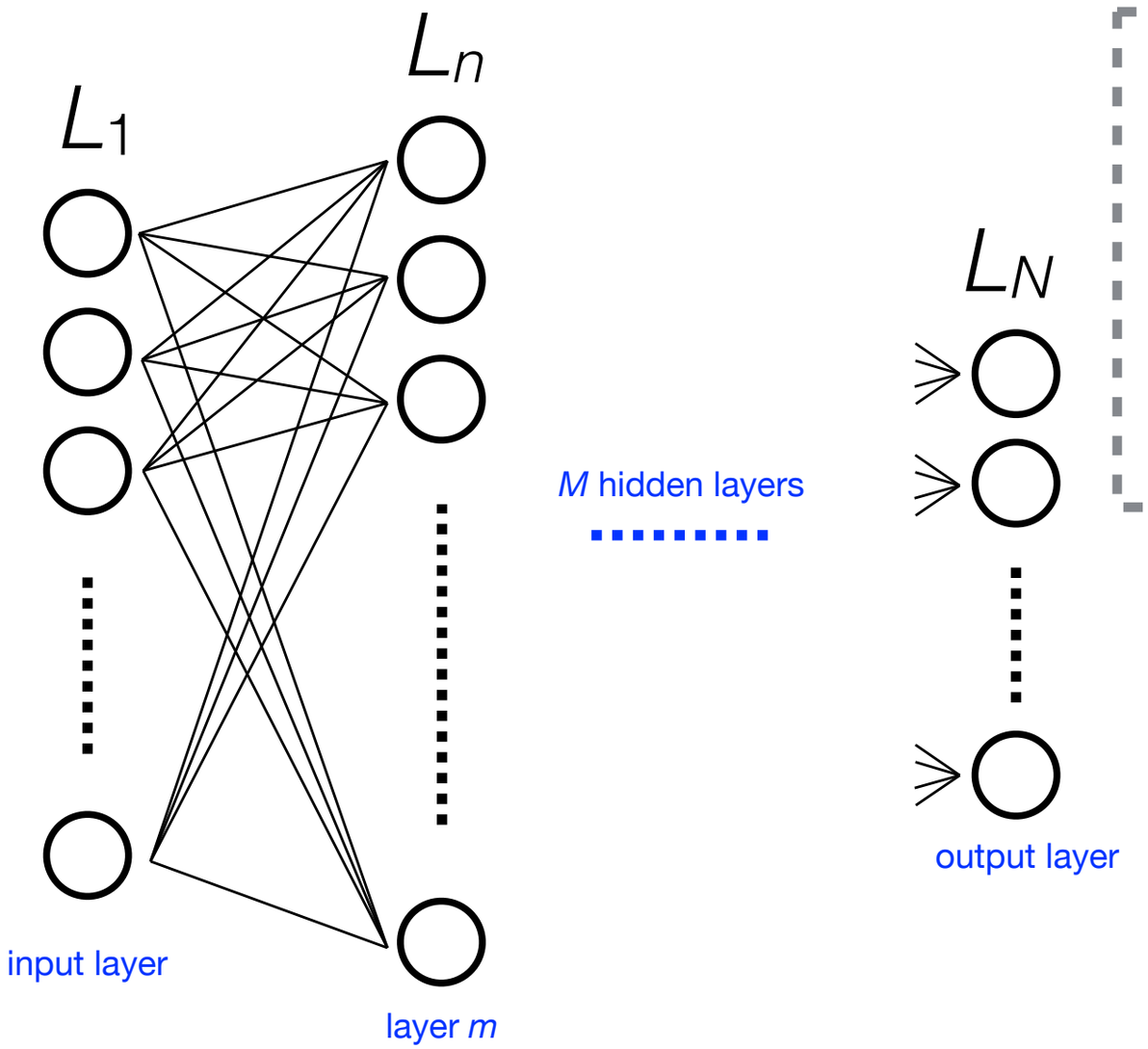
multiplication — DSPs

addition — logic cells

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



Neural network inference



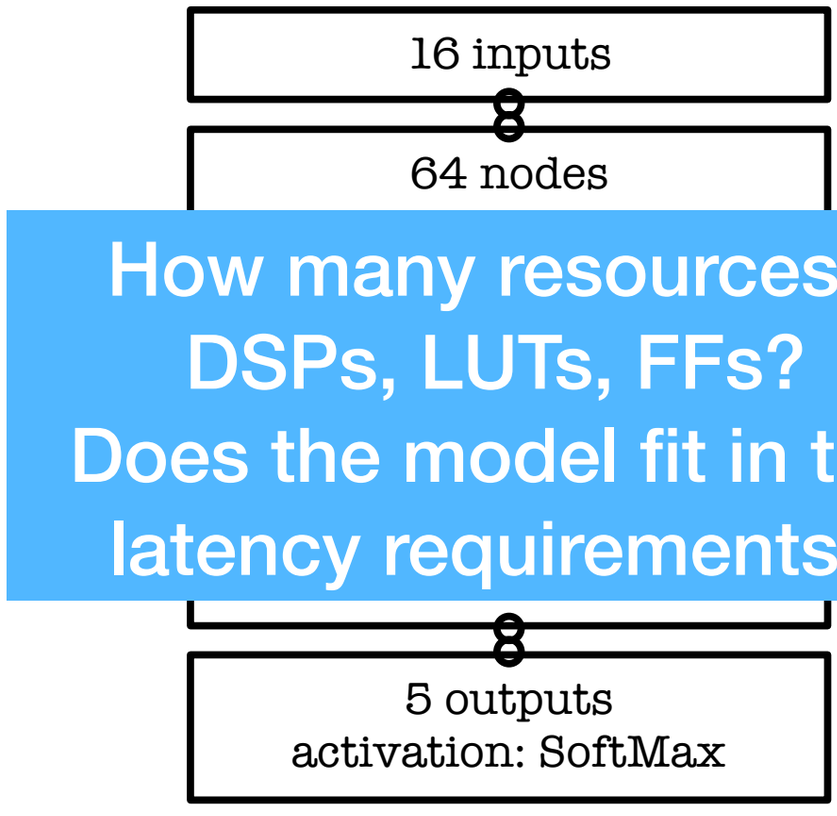
$$\mathbf{x}_n = g_n(\mathbf{W}_{n,n-1}\mathbf{x}_{n-1} + \mathbf{b}_n)$$

activation function — precomputed and stored in BRAMs

multiplication — DSPs

addition — logic cells

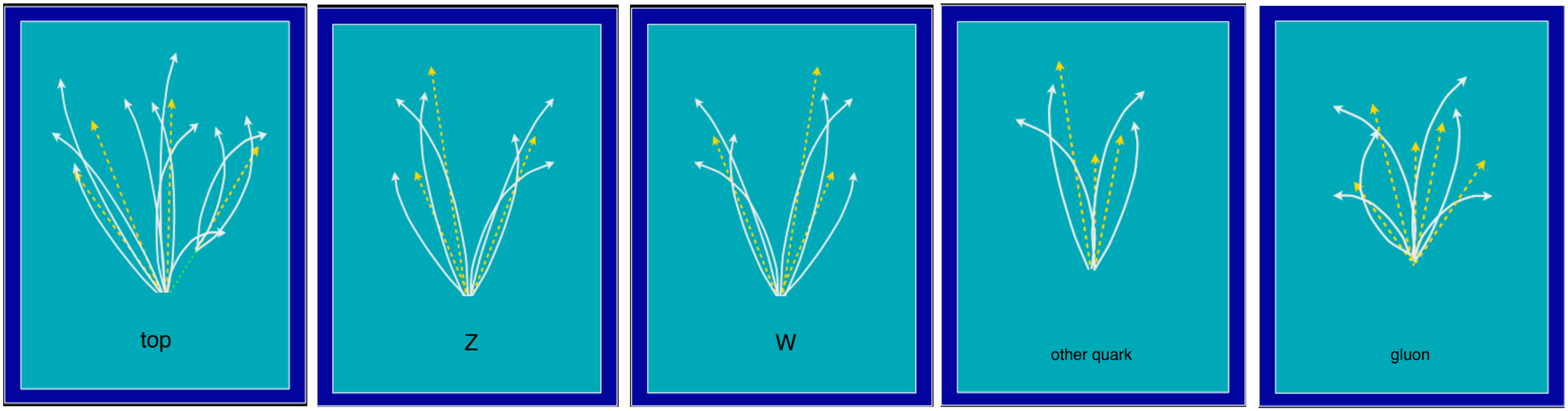
$$N_{\text{multiplications}} = \sum_{n=2}^{N} L_{n-1} \times L_n$$



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

Case study: jet tagging

Study a multi-classification task: 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

Signal: reconstructed as one massive jet with substructure

Jet substructure observables used to distinguish signal vs background [*]

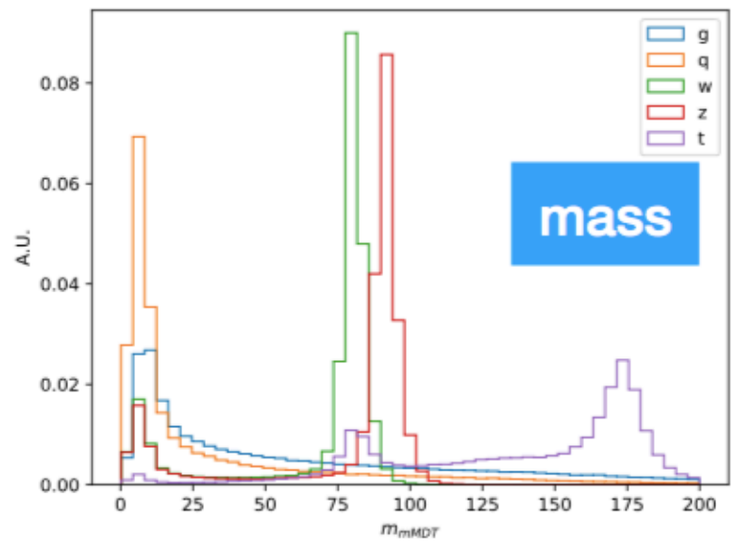
[*] D. Guest at al. PhysRevD.94.112002, G. Kasieczka et al. JHEP05(2017)006, J. M. Butterworth et al. PhysRevLett.100.242001, etc..

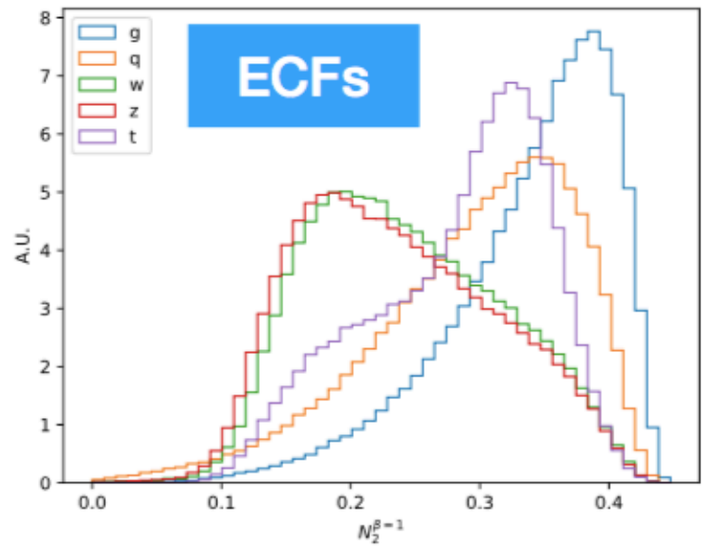
Jet substructure features

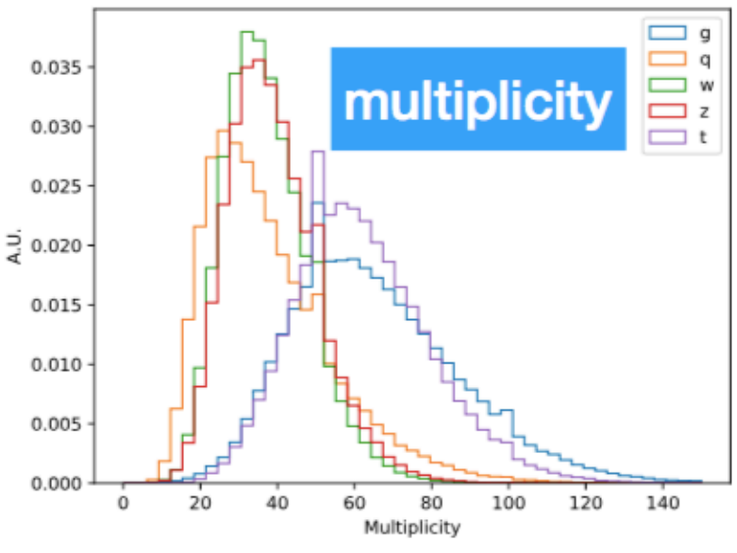
Jet substructure observables provide large discrimination power between these types of jets

mass, multipliticity, energy correlation functions, …
(computed with FastJet [*])

[*] E. Coleman et al. JINST13(2018) T01003,
M. Cacciari et al, Eur. Phys. J.C72(2012)1896







These are expert-level features

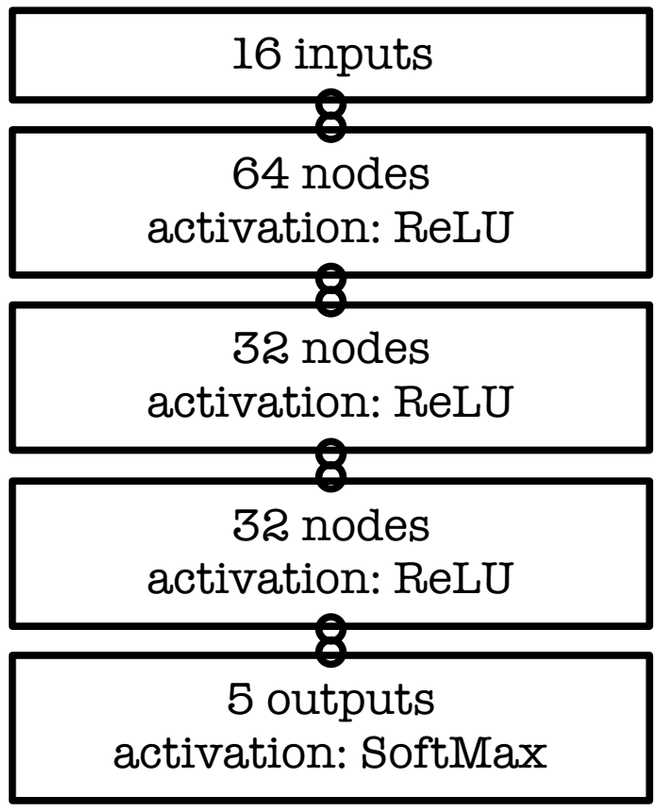
Not necessarily realistic for L1 trigger
"Raw" particle candidates more suitable *(to be studied next)*
But lessons here are generic

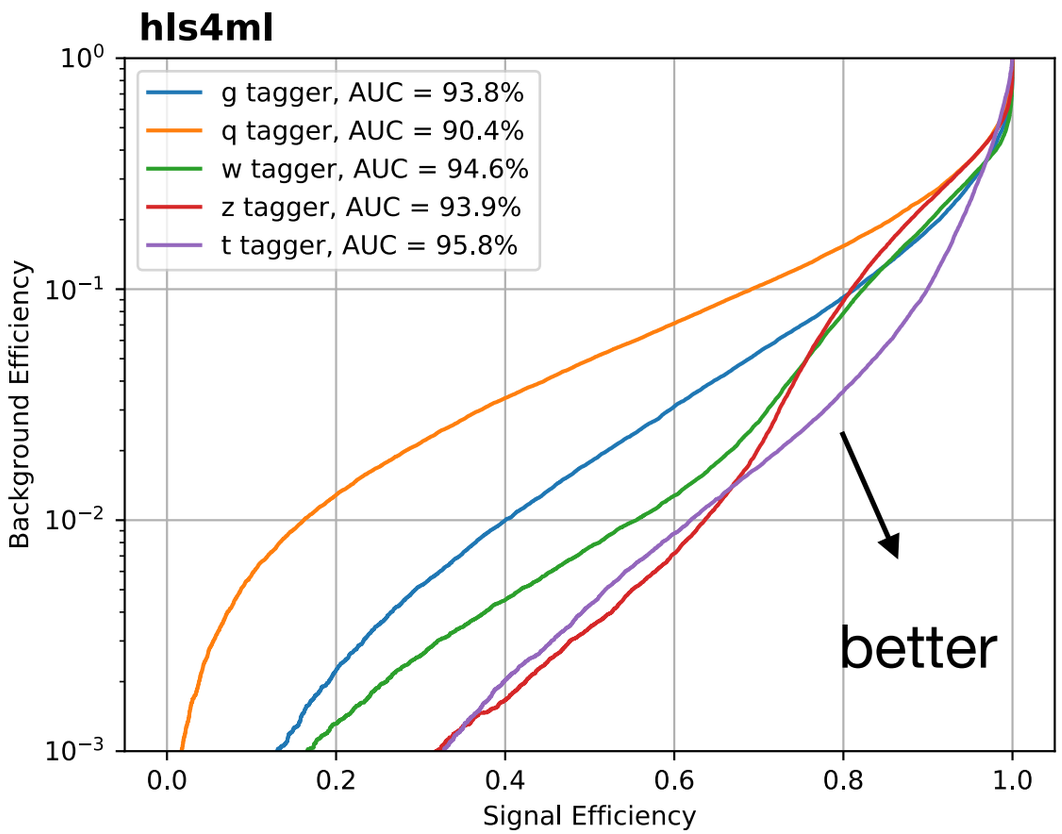
One more case: H→bb discrimination vs W/Z→qq requires more "raw" inputs for b-tagging information

Case study: jet tagging

- We train (on GPU) a **five output multi-classifier**: sample of events with two boosted WW/ZZ/tt/qq/gg anti-k_T jets, generated with Madgraph and showered with Pythia8

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





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

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

We have three handles:

- **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

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

We have three handles:

- **compression:** reduce number of synapses or neurons

- **quantization:** reduces the precision of the calculations (inputs, weights, biases)

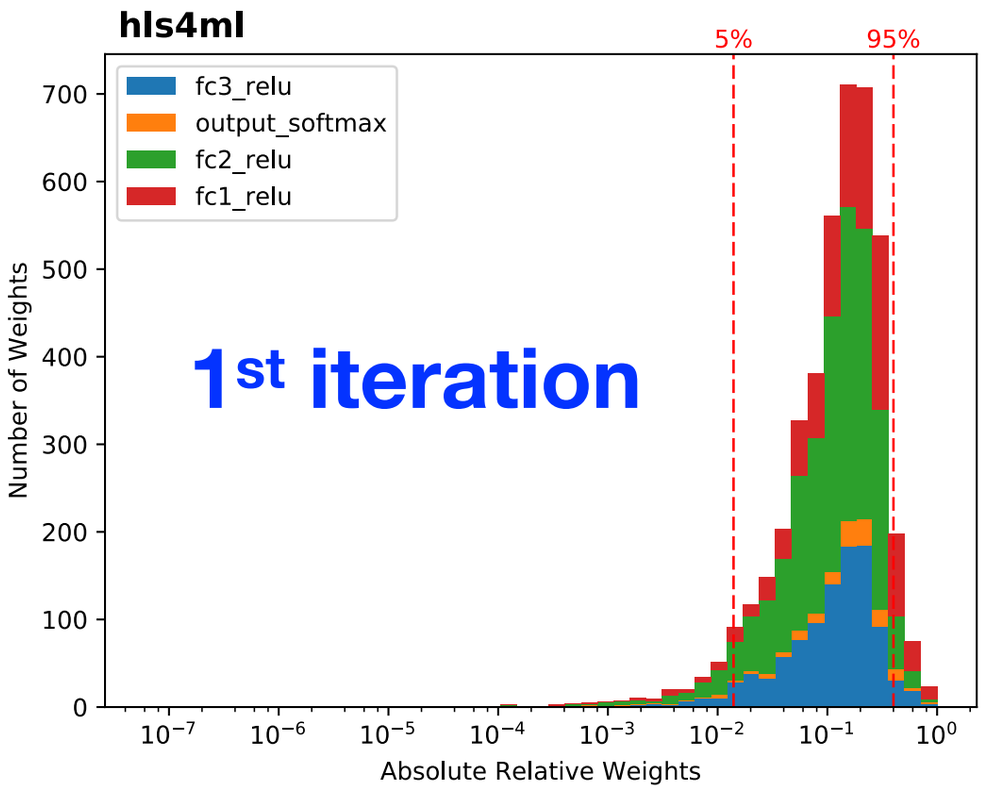
- **parallelization**: tune how much to parallelize to make the inference faster/slower versus FPGA resources

Efficient NN design: compression

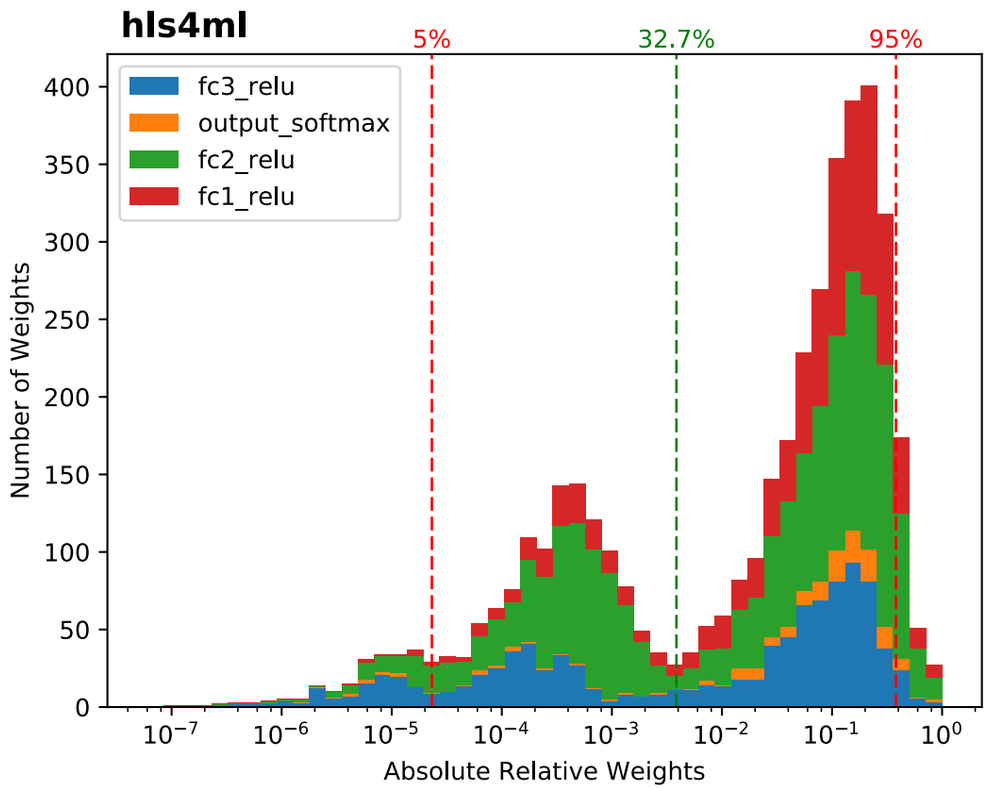
- Iterative approach:
  - train with **L1 regularization** (loss function augmented with penalty term):

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

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



Train with L_1



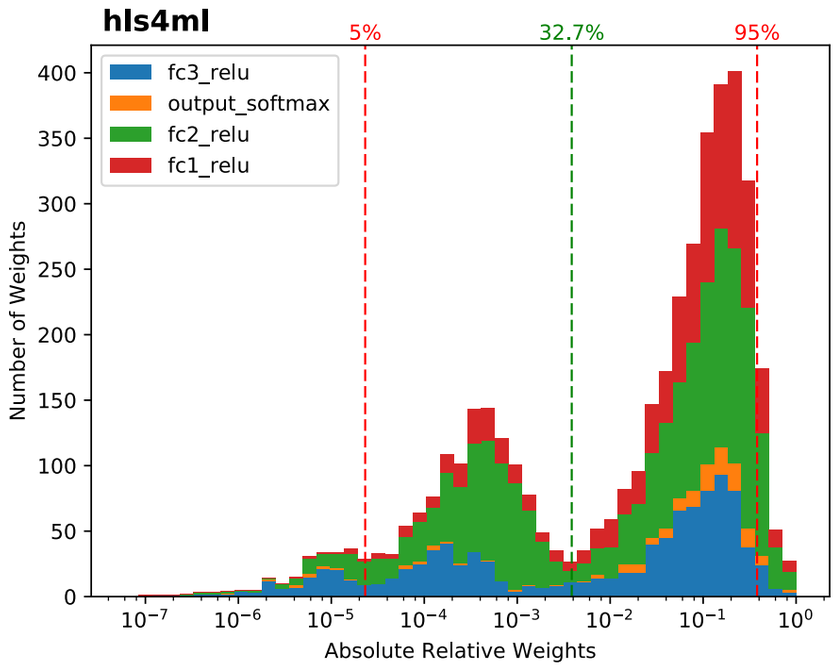
Efficient NN design: compression

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

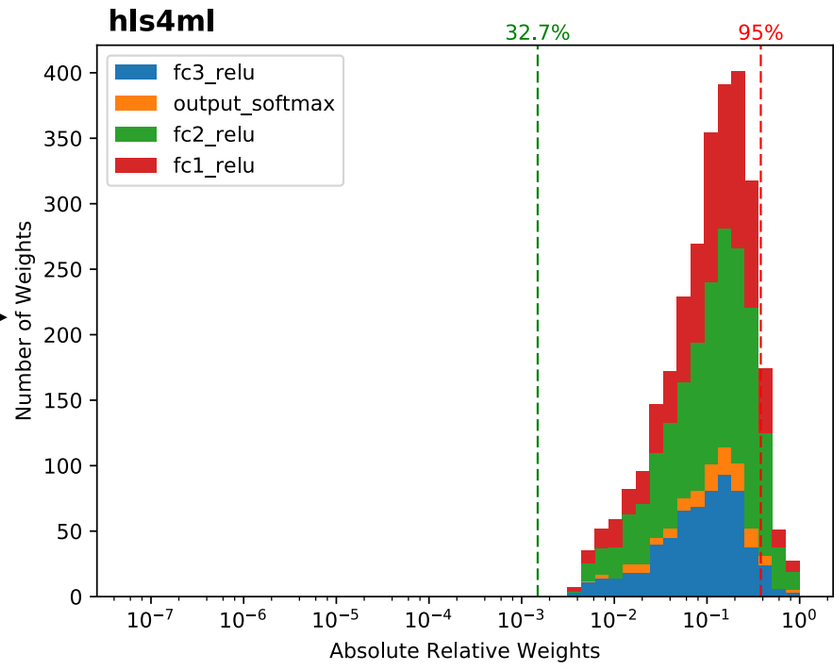
$$L_\lambda(\vec{w}) = L(\vec{w}) + \lambda||\vec{w}_1||$$

  - 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

Train with L_1

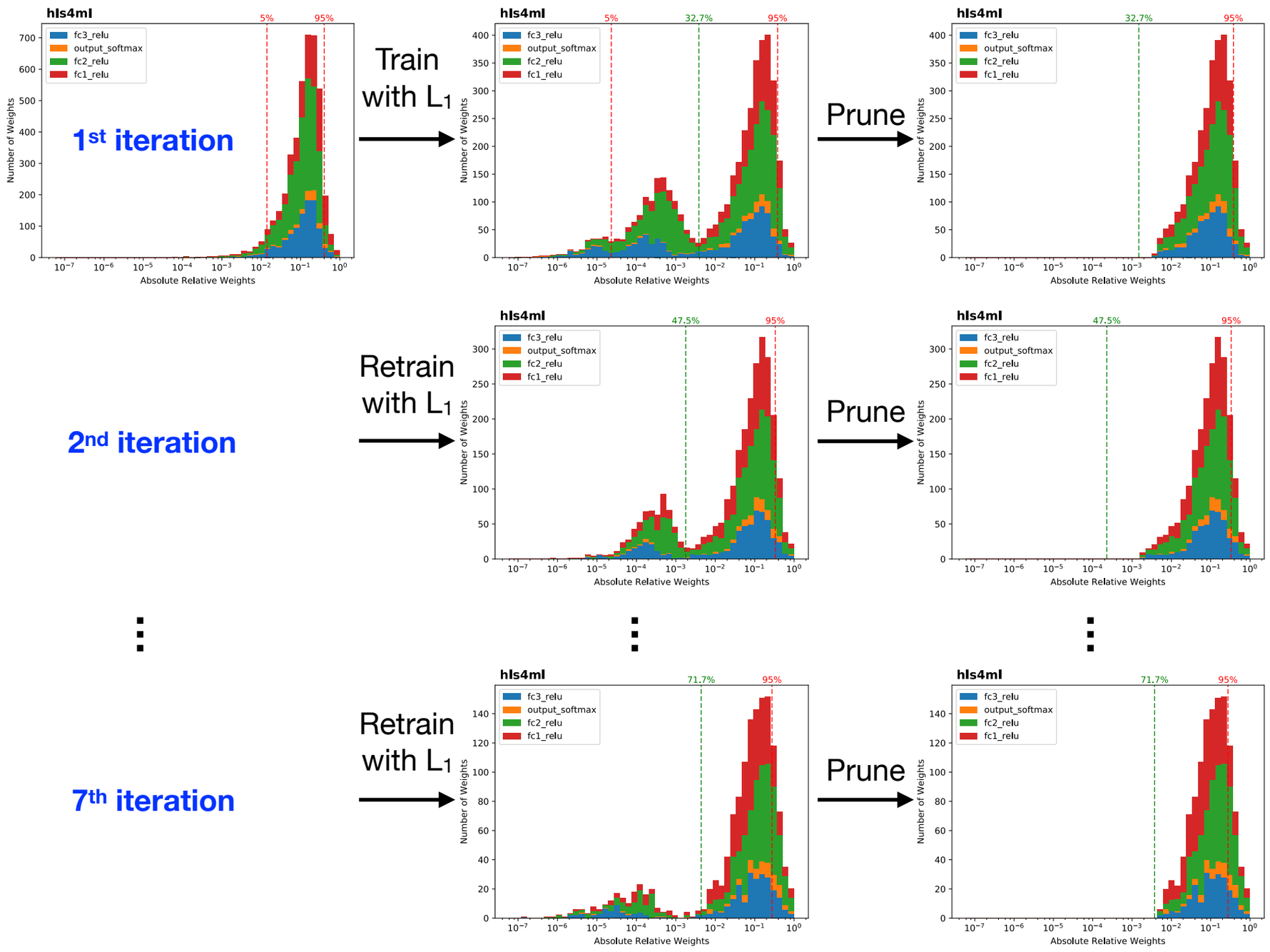


Prune



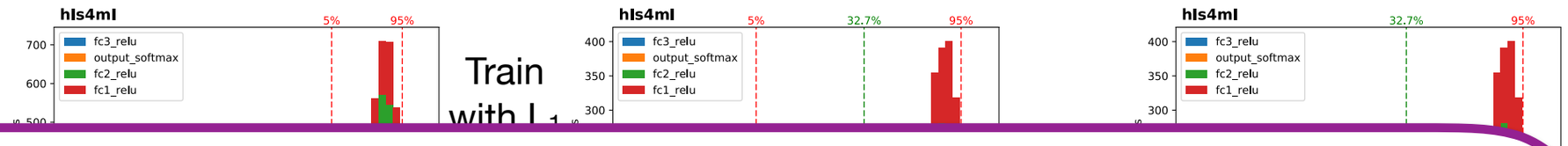
Efficient NN design: compression

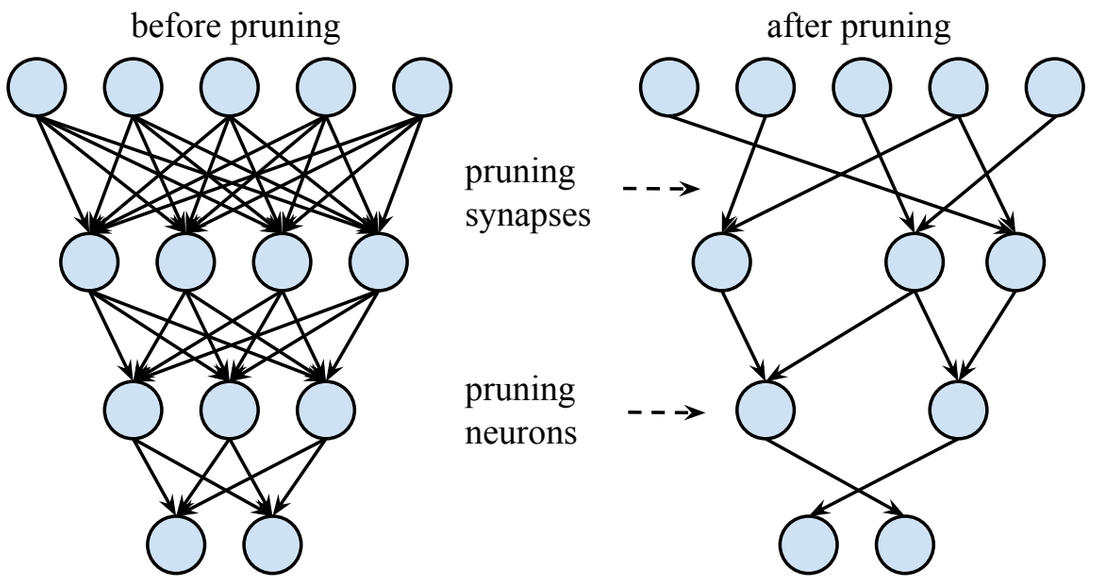
Prune and repeat the train for 7 iterations



Efficient NN design: compression

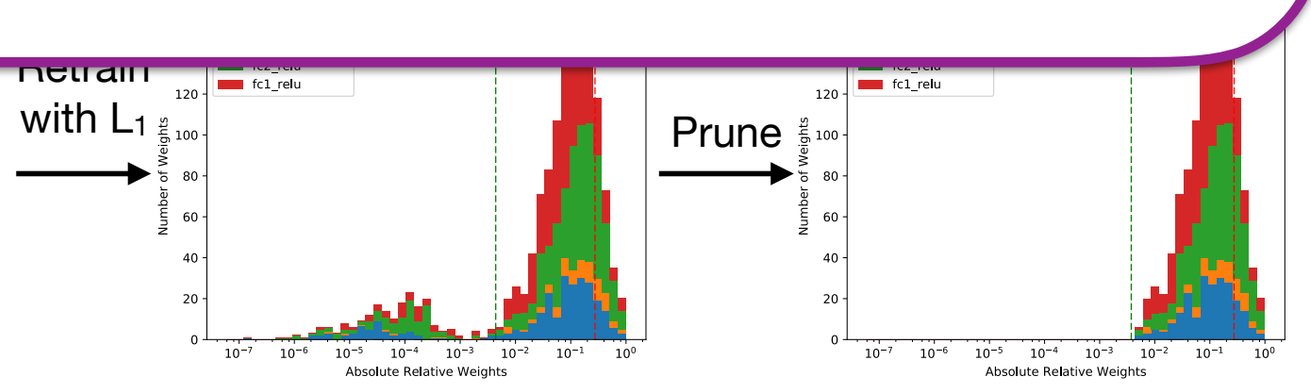
Prune and repeat the train for 7 iterations





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

7th iteration



Efficient NN design for FPGAs

FPGAs provide huge flexibility

Performance depends on how well you take advantage of this

Constraints:

Input bandwidth

FPGA resources

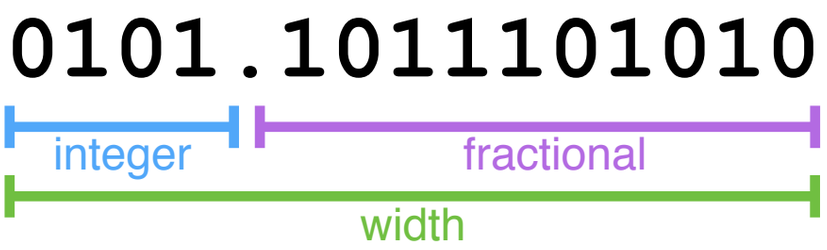
Latency

We have three handles:

- **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

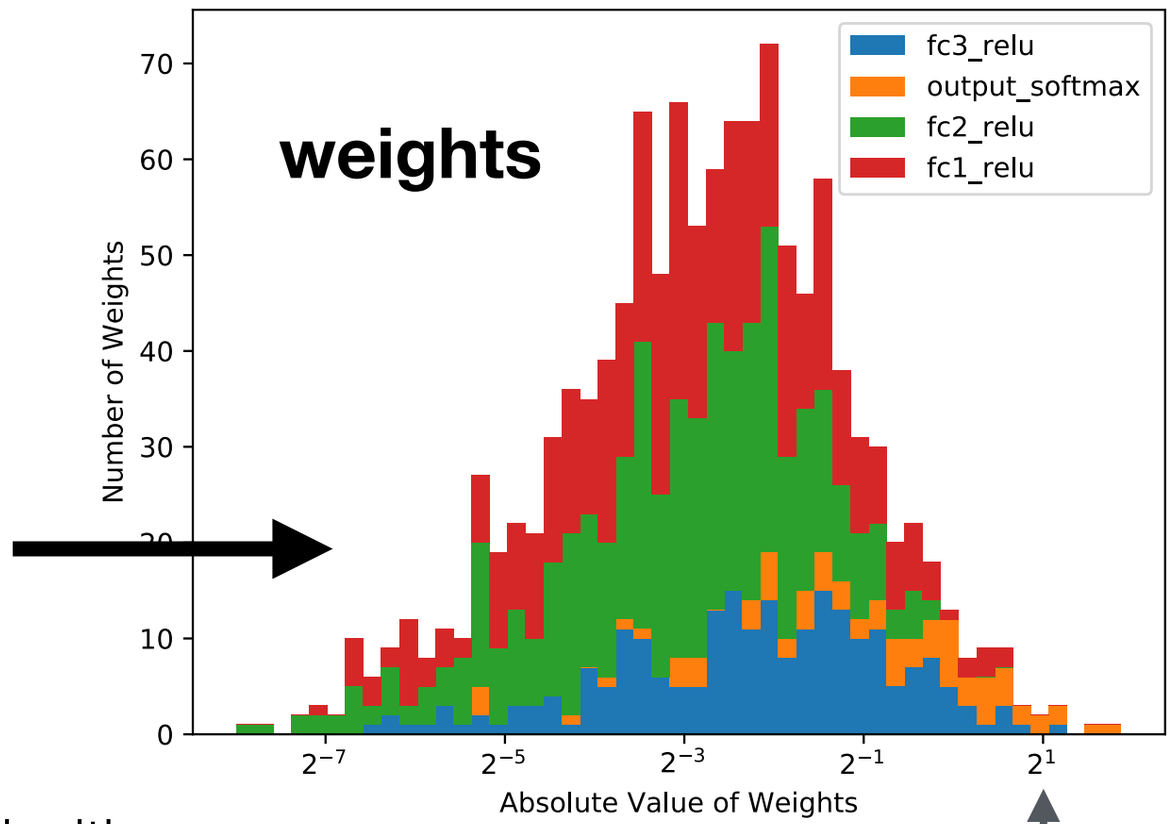
Efficient NN design: quantization

- In FPGAs use **fixed point data types** → less resources and latency than 32-bit floating point
- NN inputs, weights, biases, outputs represented as ap_fixed<width,integer>




```
ap_fixed<14,4>
```

- To avoid overflow/underflow of weights at least 3 bits needed



- But need more bits for neurons as computed with multiplications and sums → we perform a **scan of physics performance versus bit precision**

integer bits = 2 + 1 for sign
(need more for neurons)

Efficient NN design for FPGAs

FPGAs provide huge flexibility

Performance depends on how well you take advantage of this

Constraints:

Input bandwidth

FPGA resources

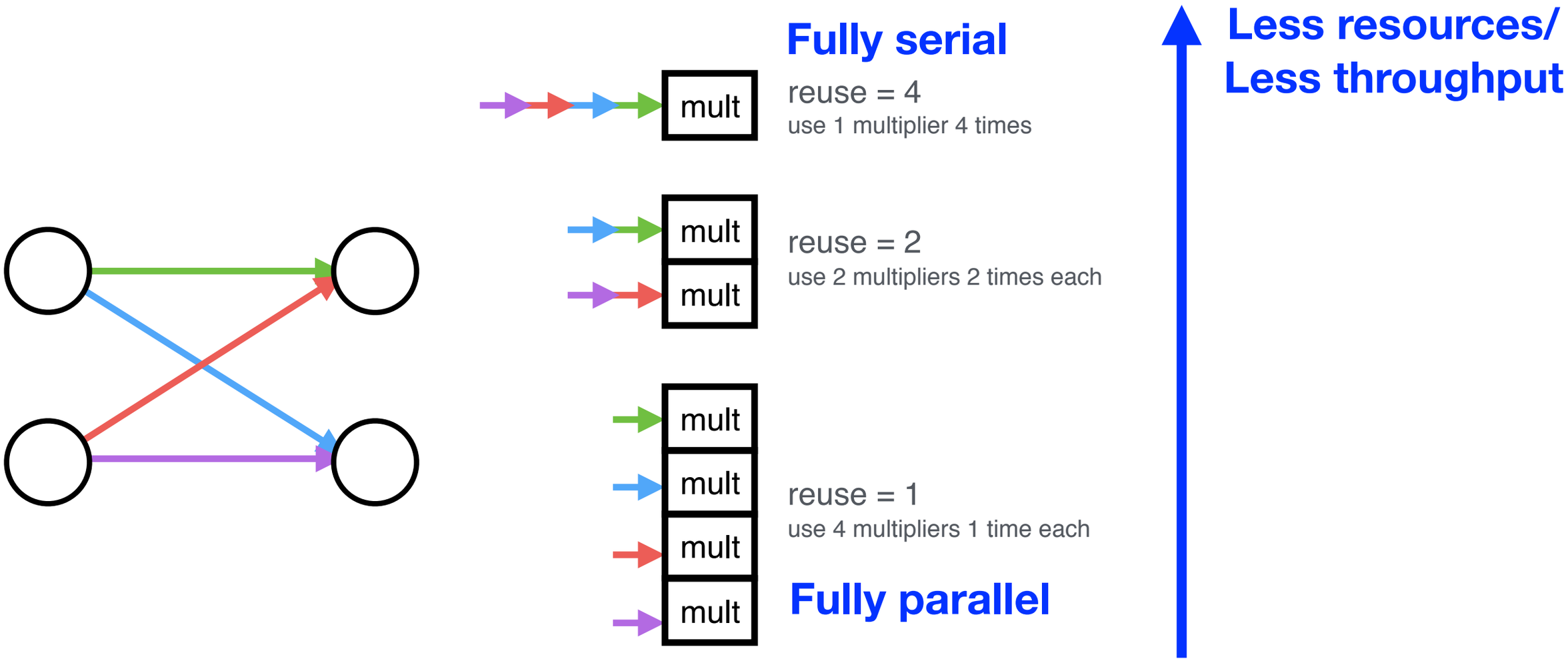
Latency

We have three handles:

- **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

Efficient NN design: parallelization

- Trade-off between latency and FPGA resource usage determined by the parallelization of the calculations in each layer
- Configure the **“reuse factor”** = number of times a multiplier is used to do a computation



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

Compression, Quantization, and Parallelization made easy in

high level synthesis for machine learning

PYTORCH

Keras
TensorFlow
PyTorch
...

model

compressed model

Usual ML software workflow

Vivado™ HLS

hls4ml

Co-processing kernel

HLS conversion

HLS project

Custom firmware design

tune configuration
precision
reuse/pipeline

K + TensorFlow

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

The hls4ml package

Translation →

```
python keras-to-hls.py -c keras-config.yml
```

Inputs →

Keras + HDF5

Config

```
KerasJson: example-keras-model-files/KERAS_1layer.json
KerasH5:   example-keras-model-files/KERAS_1layer_weights.h5
OutputDir: my-hls-test
ProjectName: myproject
XilinxPart: xc7vx690tffg1927-2
ClockPeriod: 5

IOType: io_parallel # options: io_serial/io_parallel
ReuseFactor: 1
DefaultPrecision: ap_fixed<18,8>
```

- `IOType`: parallelize or serialize
- `ReuseFactor`: how much to parallelize
- `DefaultPrecision`: inputs, weights, biases

```
my-hls-test/:
build_prj.tcl
firmware
myproject_test.cpp
```

The hls4ml package

Build HLS project → `vivado_hls -f build_prj.tcl`


```
#################
#    HLS4ML     #
#################
open_project -reset myproject_prj
set_top myproject
add_files firmware/myproject.cpp -cflags "-I[file normalize ../../nnet_utils]"
add_files -tb myproject_test.cpp -cflags "-I[file normalize ../../nnet_utils]"
add_files -tb firmware/weights
open_solution -reset "solution1"
set_part {xcku115-flvf1924-2-i}
create_clock -period 5 -name default
csim_design
csynth_design
cosim_design -trace_level all
export_design -format ip_catalog
exit
```

Produce a firmware block in ~ minutes!

Study details

GOAL

Map out FPGA performance, resource usage and latency versus compression, quantization, and parallelization hyperparameters

SETUP

Xilinx Vivado 2017.2

HLS target clock frequency: 200 MHz (5 clocks/BX)

Kintex Ultrascale, xcku115-flvb2104-2-i

- 1.4M logic cells, 5,520 DSPs, 1.3M FFs, 700k LUTs, 2200 BRAMs

RESULTS

First examine resource usage coming from HLS estimate

Then discuss the exact resources given by the final implementation

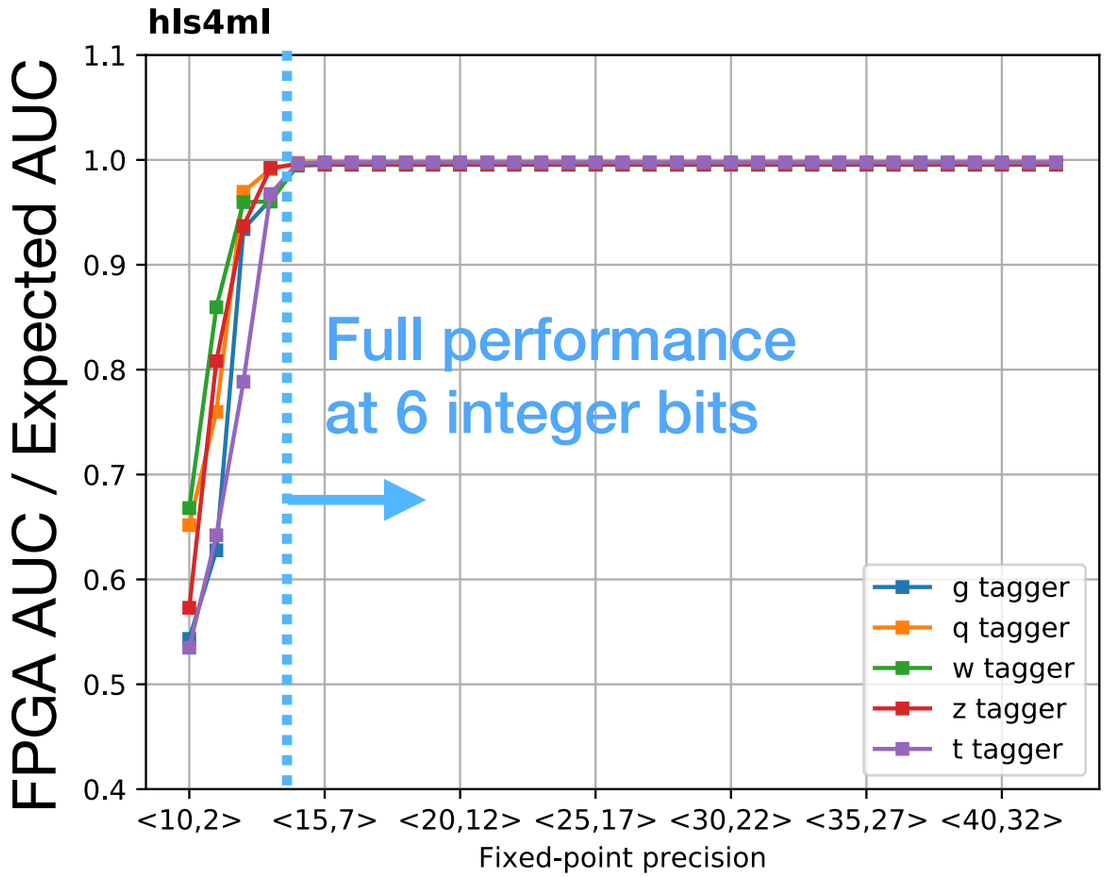
Efficient NN design: quantization

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

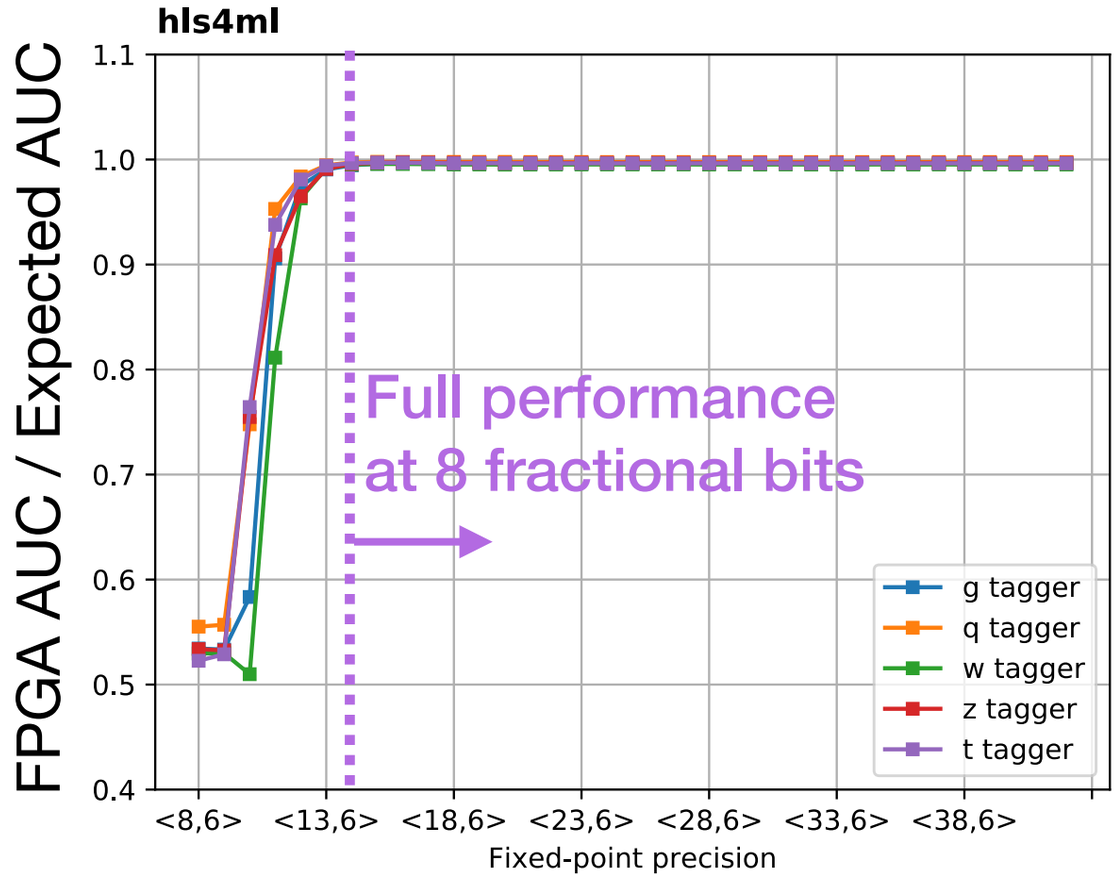
Scan integer bits

Fractional bits fixed to 8

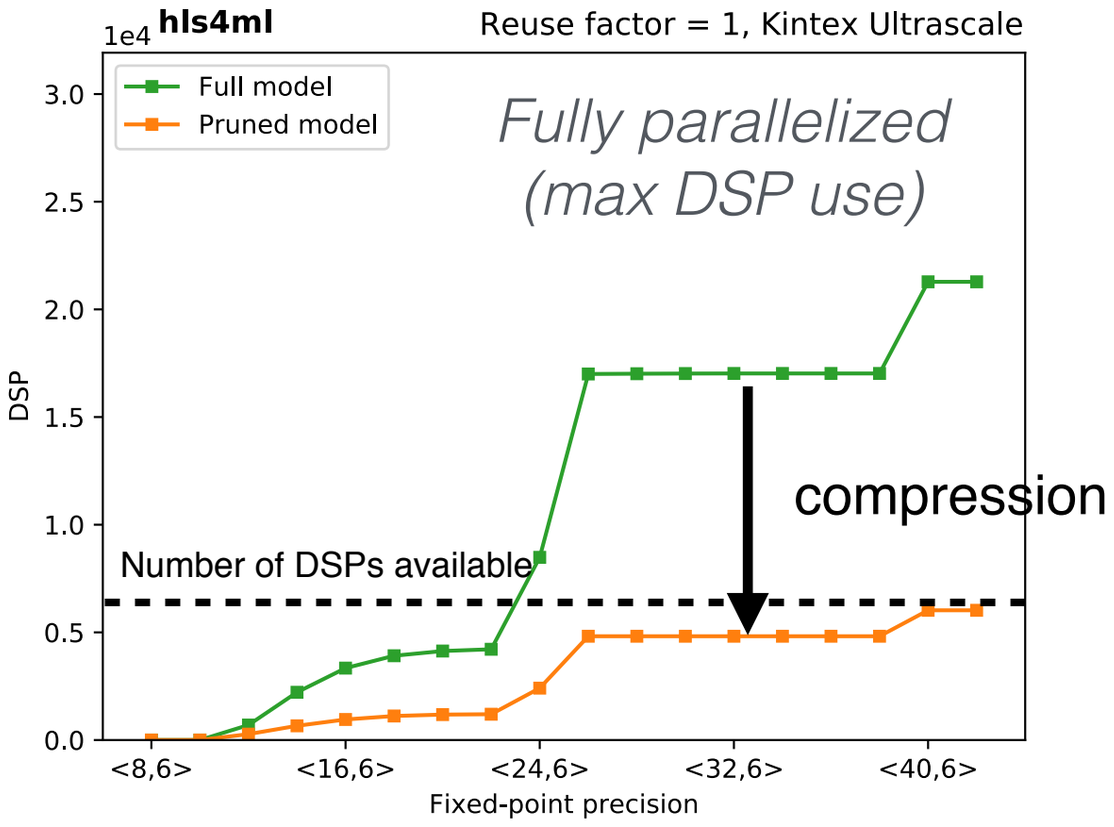


Scan fractional bits

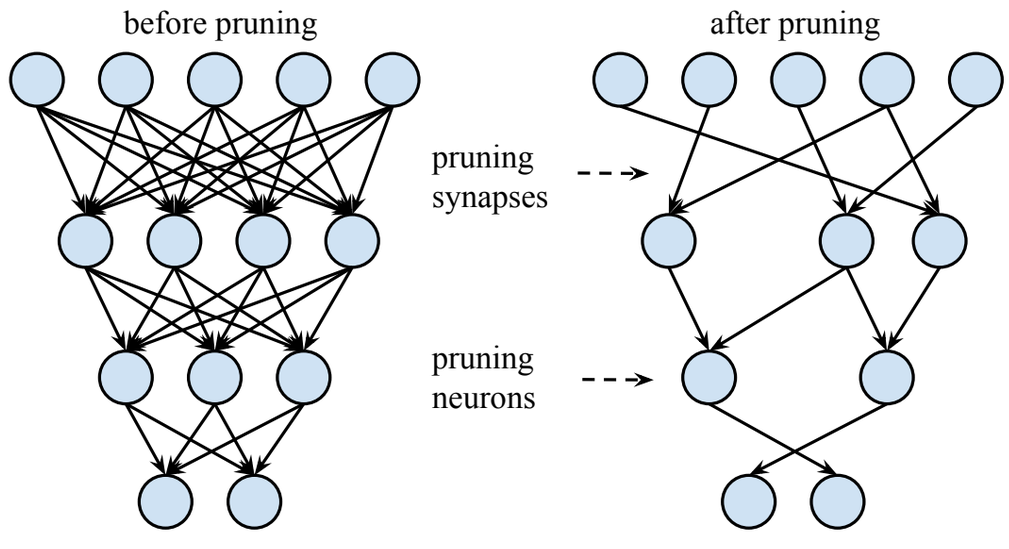
Integer bits fixed to 6



Efficient NN design: compression

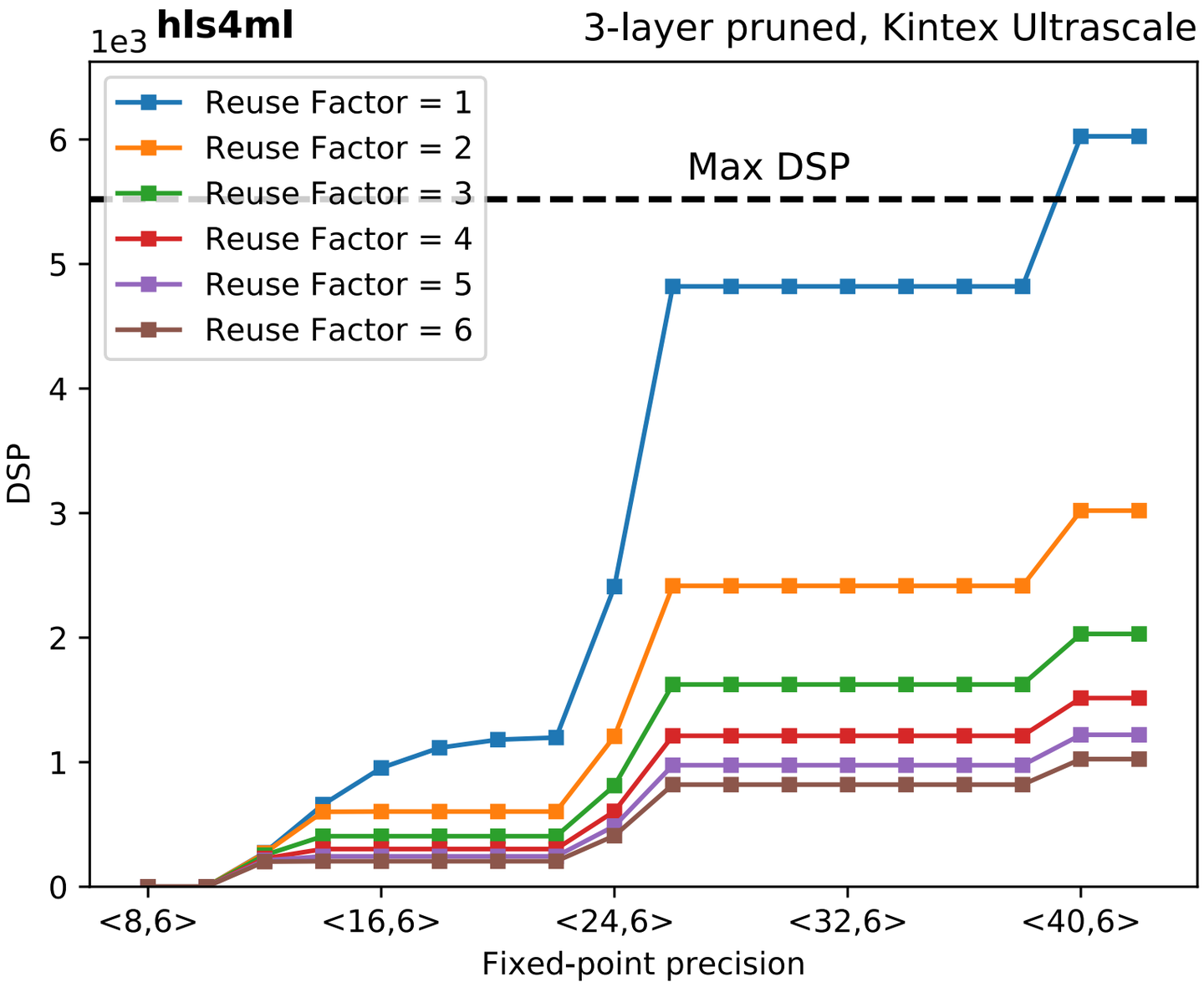


70% compression ~ 70% fewer DSPs



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

Parallelization: DSPs usage



Fully parallel
Each mult. used 1x

Each mult. used 2x

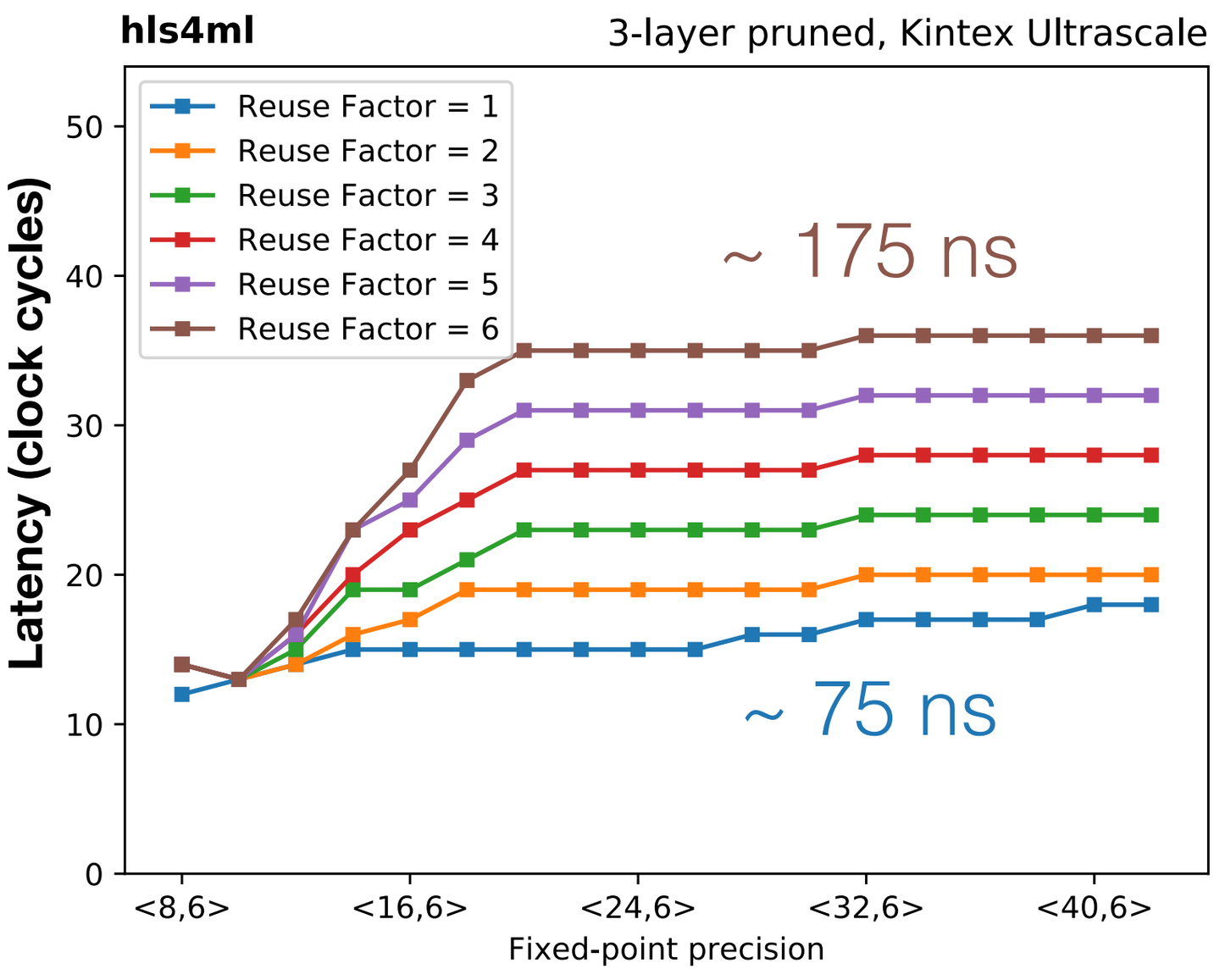
Each mult. used 3x

⋮

Parallelization: Timing

Latency of layer m

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



Longer latency

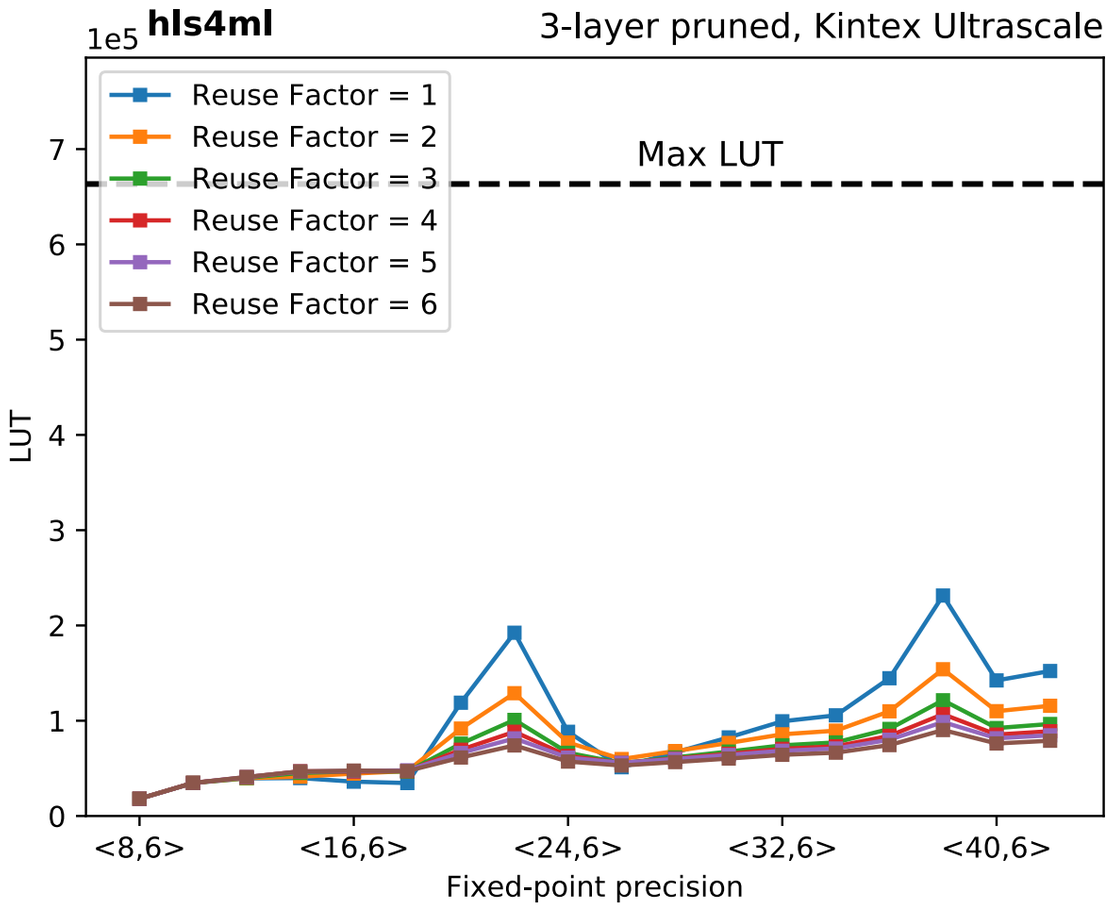
Each mult. used 6x

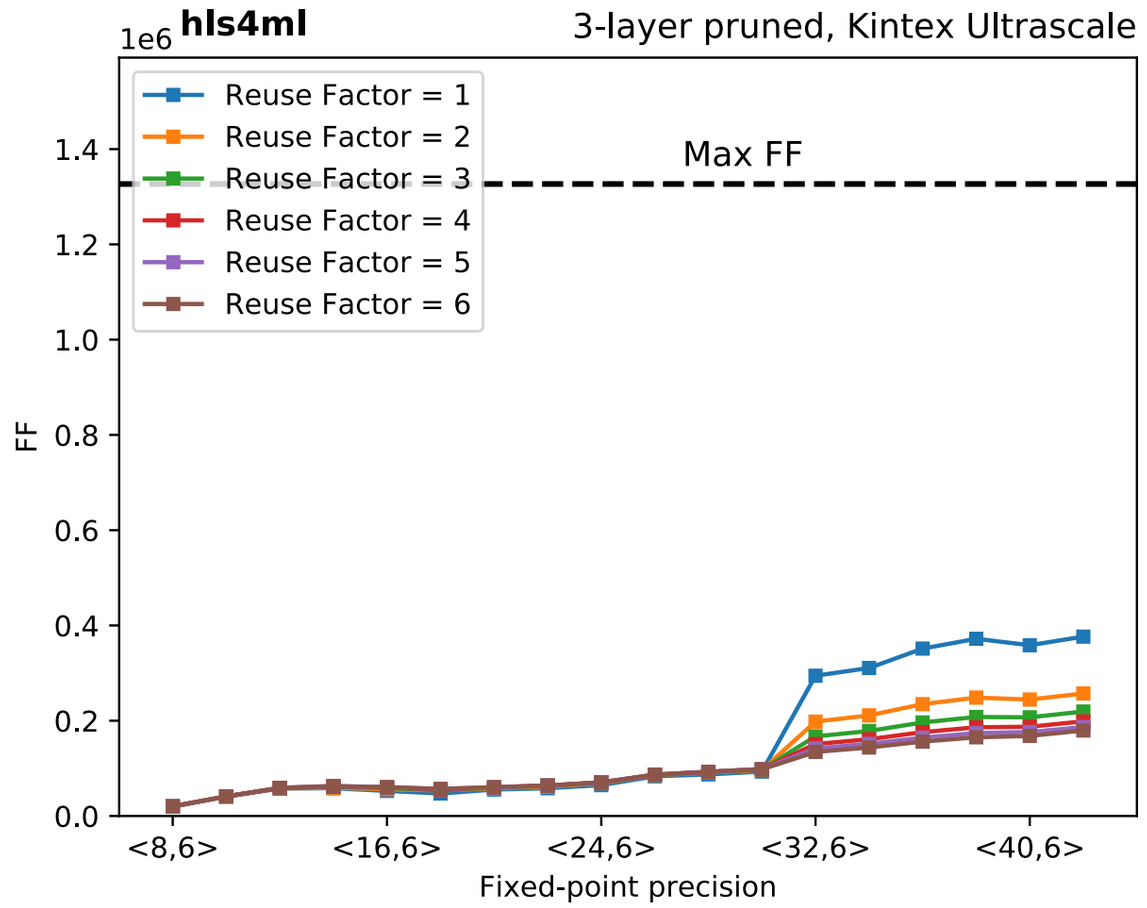
Each mult. used 3x

Fully parallel
Each mult. used 1x

More resources

Other resources: FFs and LUTs





Fairly linear increase with precision

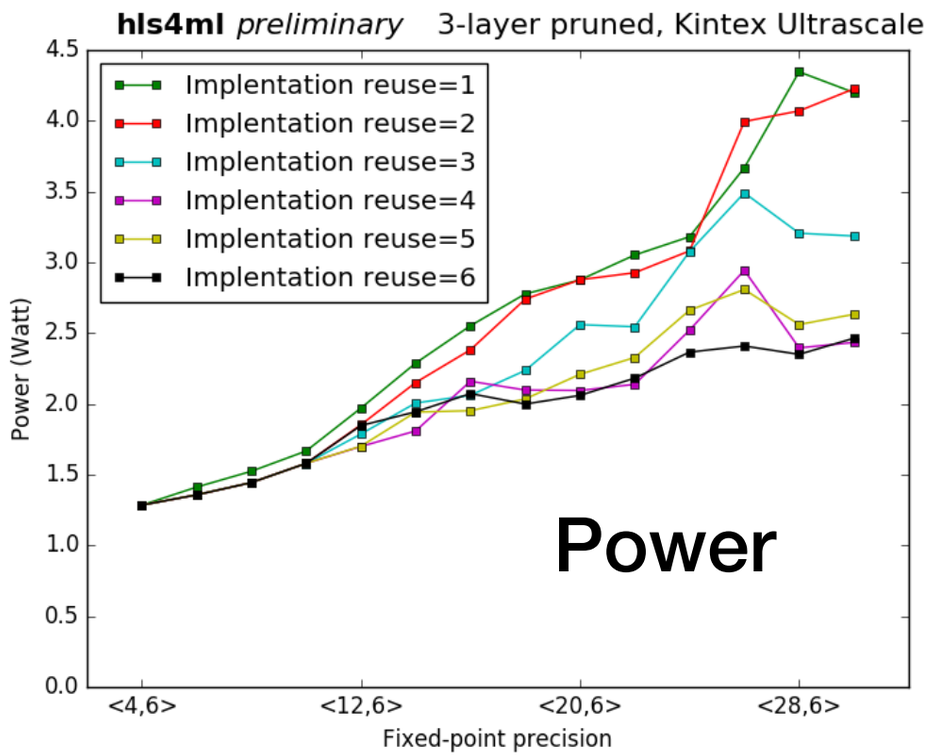
Small percentage of total available

Spikes present at steep transitions in LUTs usage as artifacts of HLS synthesis

Not observed in implementation
Found also dependence on Vivado HLS version

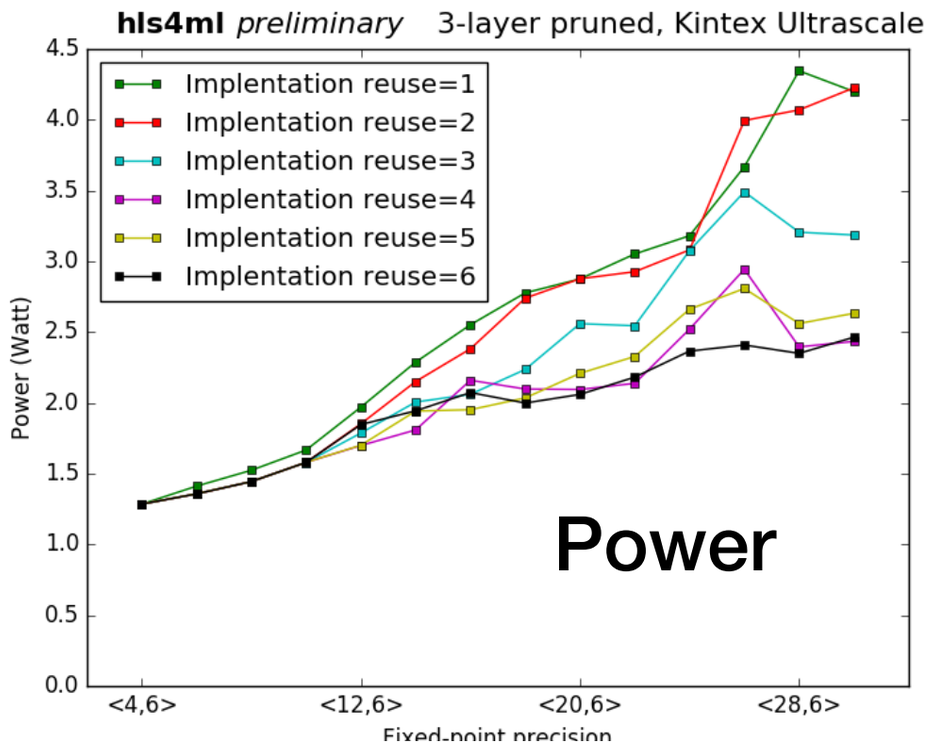
Firmware implementation

- Final implementation gives actual resource usage and timing estimate
  - **how optimal is the HLS design?**
- Power usage increases with precision, it goes down for less throughput (higher reuse factor)

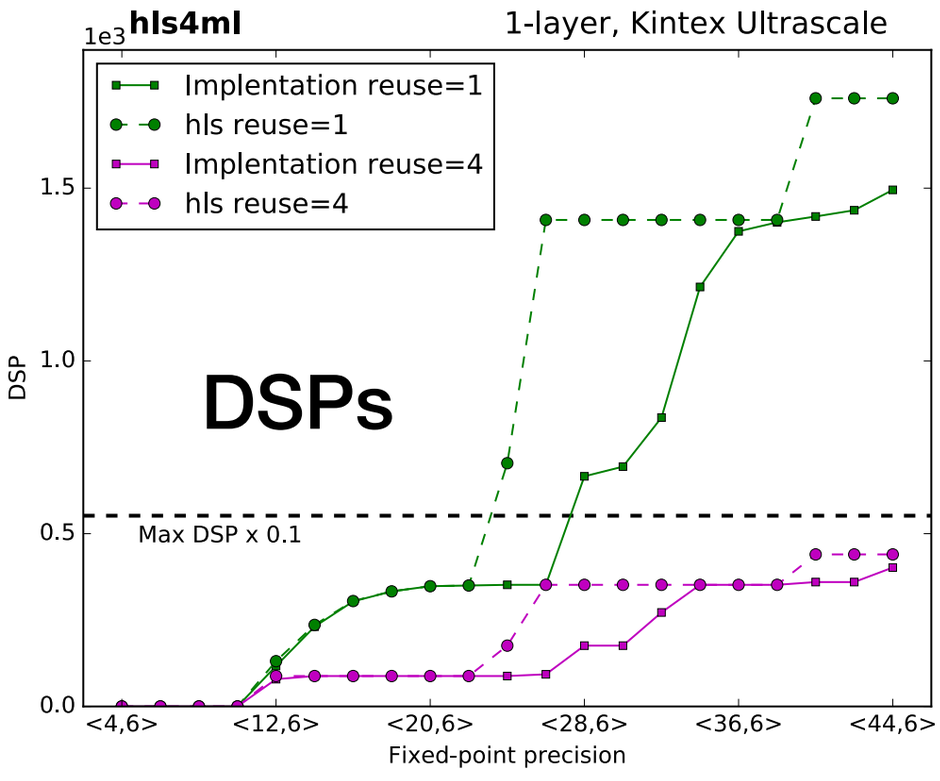


Firmware implementation

- Final implementation gives actual resource usage and timing estimate
  - **how optimal is the HLS design?**
- Power usage increases with precision, it goes down for less throughput (higher reuse factor)

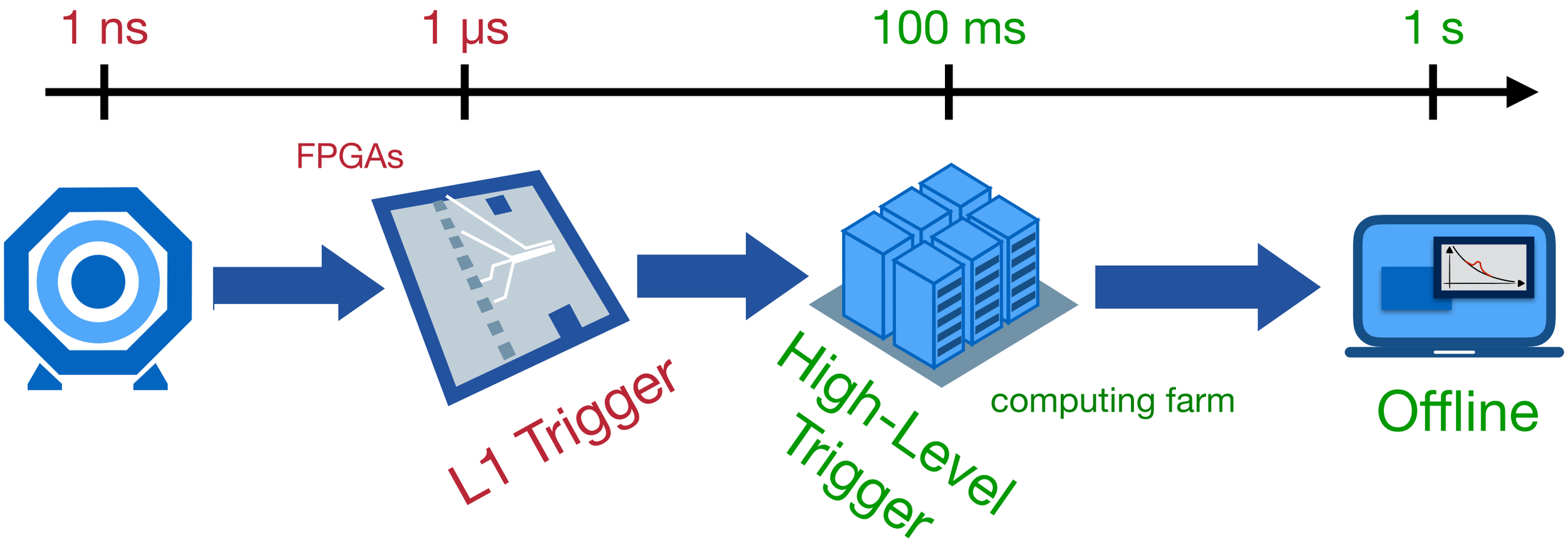


- Implement a 1-layer NN, simply routing all firmware block's inputs and outputs to FPGA available pins
- HLS estimate on resource usage are conservative
  - DSPs usage agree well below DSP precision transition (27 bit), implementation does further optimization
  - FFs and LUTs overestimated by a factor 2-4



Summary and outlook

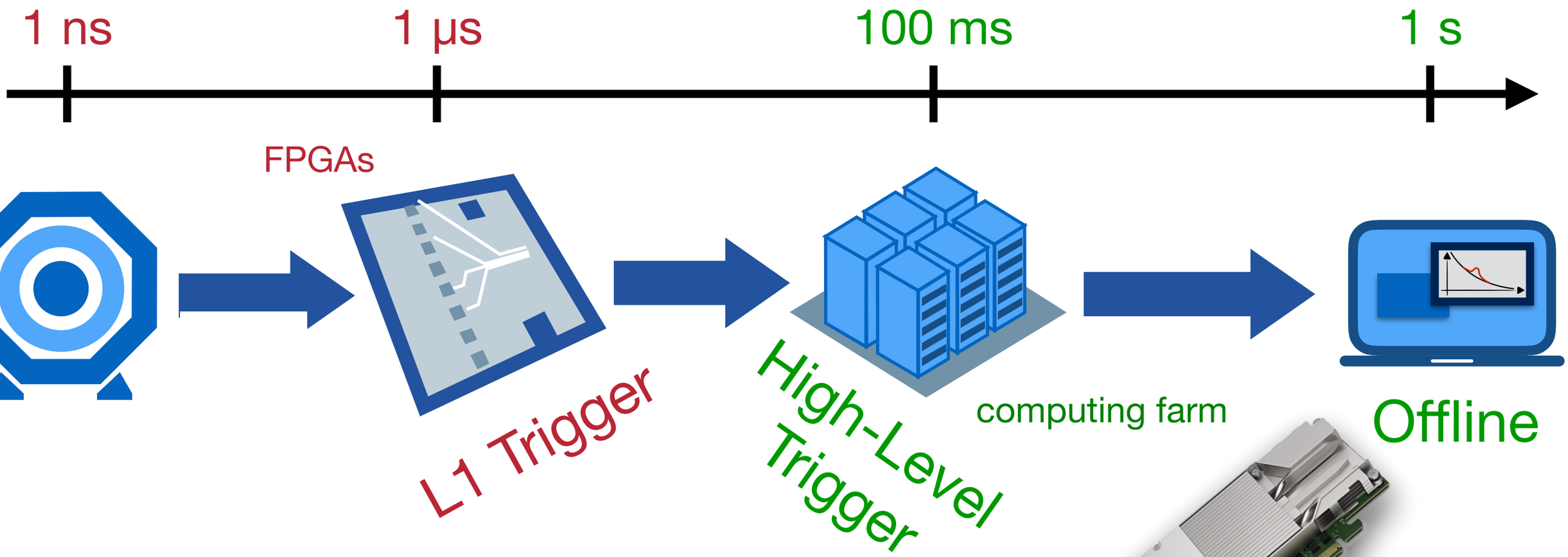
The latency landscape @ LHC



Focused on L1 trigger as first application
→ pure FPGAs

What can we do in < us on one FPGA?

The latency landscape @ LHC



Focused on L1 trigger as first application
→ pure FPGAs

What can we do in < us on one FPGA?

More time means
More resource reuse (x1000)
Bigger networks

→ acceleration with FPGAs

Heterogeneous computing with co-processors

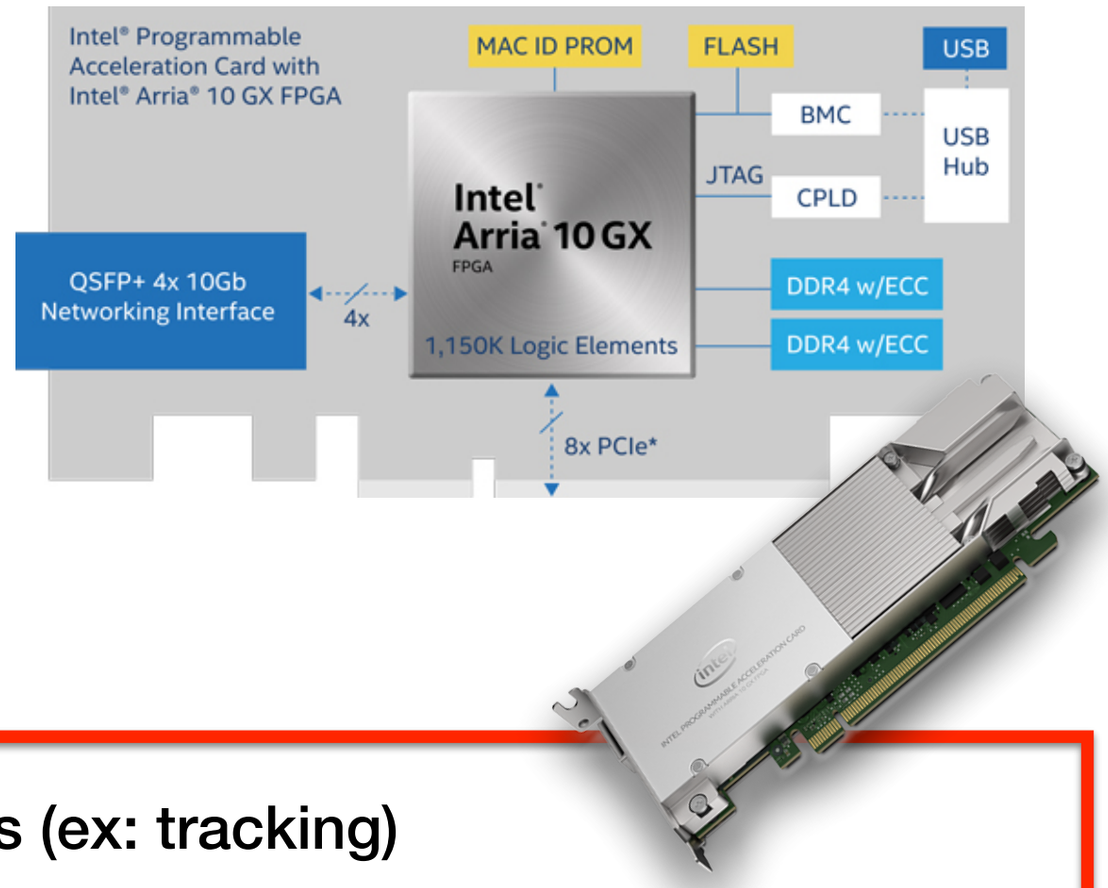
Offload a CPU from the computational heavy parts to a FPGA "accelerator"

Increased computational speed of 10x-100x
Reduced system size of 10x
Reduced power consumption of 10x-100x

Increasing popularity of co-processor systems

CPU+FPGA / CPU+GPU / CPU+TPU / …
Common setup for FPGA connects to CPU through PCI-express

Intel® Programmable Acceleration Card with Intel Arria® 10 GX FPGA



Use case @ LHC to accelerate slow algorithms (ex: tracking) and ML inference for HLT and offline analysis

Ongoing R&D on heterogeneous computing on-site (@CERN) and on commercial clouds (Microsoft Brainwave, Amazon Web Services, Google TPU cloud)

goes to the cloud

SDAccel™
Environment

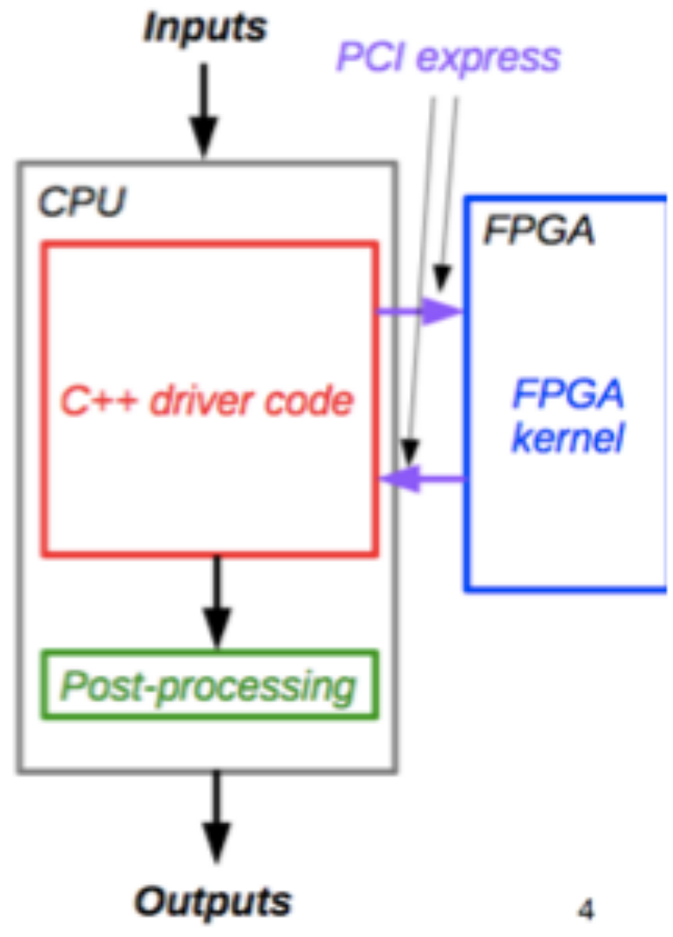
- **Amazon Web Service** provides co-processor CPU+FPGA systems with Xilinx Virtex Ultrascale+ VU9P

- Xilinx SDAccel development environment allows the development/running of connected FPGA kernels and CPU processes

  - any FPGA application defined in HLS, OpenCL, or VHDL/Verilog can be accelerated

- hls4ml project only needs to be wrapped to provide specific I/O ports configuration for SDAccel to interface properly

- **Succesfully accelerated 1D CNN example project on AWS F1:** 10 four-channel inputs , 3 convolutional layers, 2 dense layers, 5 outputs → *latency 116 ns*



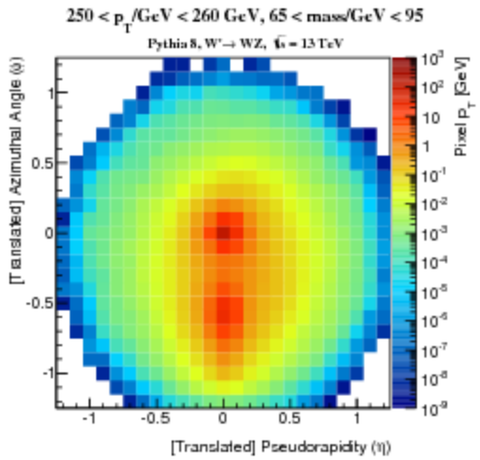
hls4ml: other NN architectures

- **Convolutional Neural Networks**
  - active implementation of small Conv1D and Conv2D with hls4ml
  - resources reuse and compression supported
  - work is ongoing to ensure large scale networks

- **Boosted Decision Tree** *(work in progress)*
  - each node in decision tree compares element against a threshold → boolean logic, thresholds in LUT, suitable for FPGA
  - each tree is independent → high parallelization

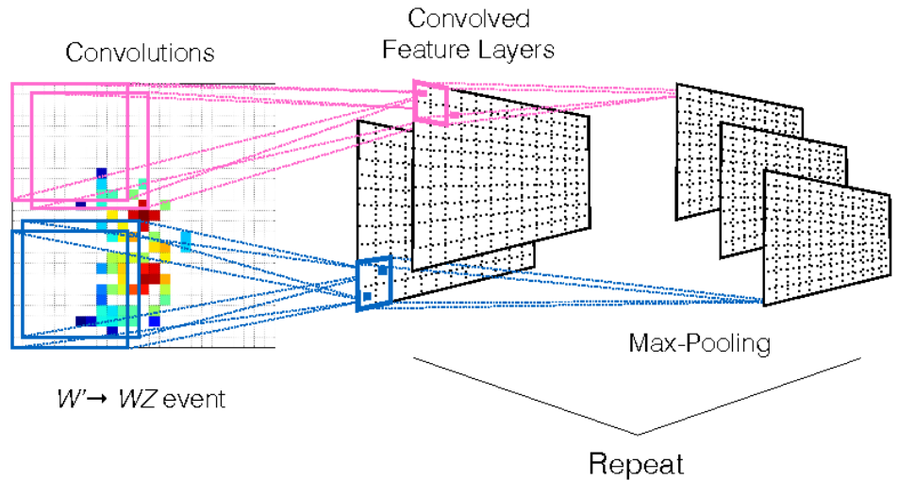
- **Binary/Ternary Neural Networks** *(work in progress)*
  - weights are binary/ternary in the inference = ±1,0
  - ternary NN does not need pruning/compression
  - similar performance and latency with 0% DSPs used

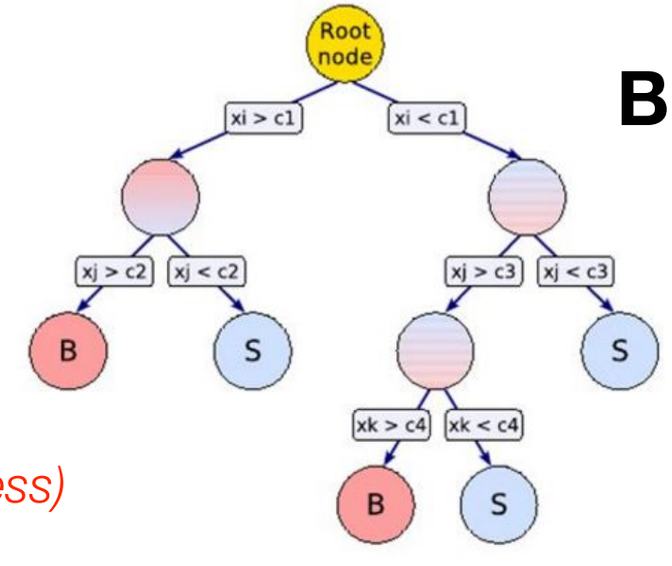
- **Recurrent NN and LSTM** under testing *(work in progress)*



Use case of CNN @ LHC for jet image

JHEP 07 (2016) 069





BDT

Summary

Introduced a new software/firmware package **hls4ml**

Automated translation of everyday machine learning inference into firmware in ~ minutes

Tunable configuration for optimization of your use case

First application is single FPGA, <1 us latency for L1 trigger or DAQ

Explore also applications for acceleration with CPU-FPGA co-processors for long latency trigger tasks

For more info

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

https://arxiv.org/abs/1804.06913

Universität Zürich[UZH]

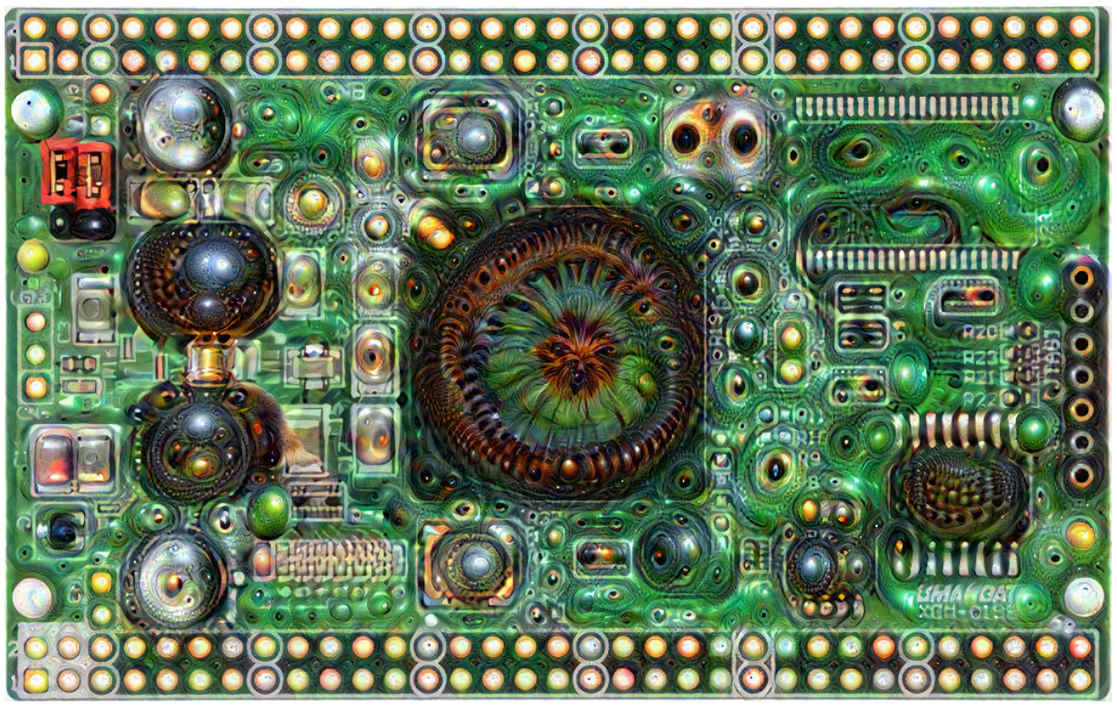
Physik-Institut

https://indico.cern.ch/event/769727/

How to do ultrafast Deep Neural Network inference on FPGAs

We accept few more registrations until January 18th!

6. February 2019
Physik Institut - Universität Zürich



A key component in autonomous vehicle and low-latency triggering systems, learn how FPGAs do real-time DNN inference in this hands-on course. Topics include:

- Model compression and quantization
- High-level synthesis
- Firmware implementation
- Model acceleration on cloud FPGAs

Lecturers:
Dr. Jennifer Ngadiuba (CERN)
Dr. Dylan Rankin (MIT)

Registration and further info at
indico.cern.ch/e/FPGA4HEP

Organizers:
Thea Årrestad (UZH)
Jennifer Ngadiuba (CERN)
Dylan Rankin (MIT)
Maurizio Pierini (CERN)
Ben Kilminster (UZH)