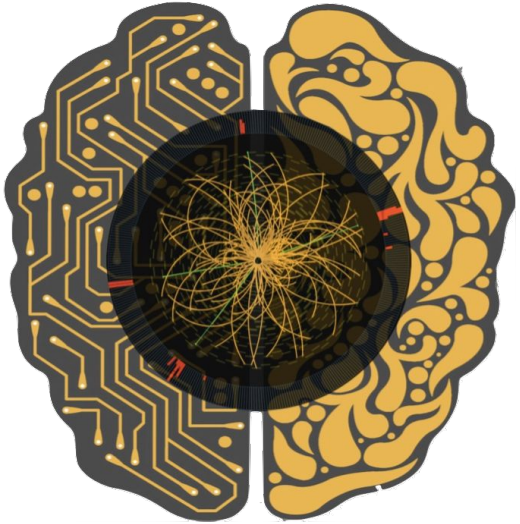


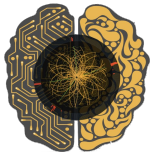
# hls4ml tutorial

## School on Machine Learning and AI Methods at LHC

## 16 July 2026

Enrico Lupi et al. for the hls4ml team

# Introduction

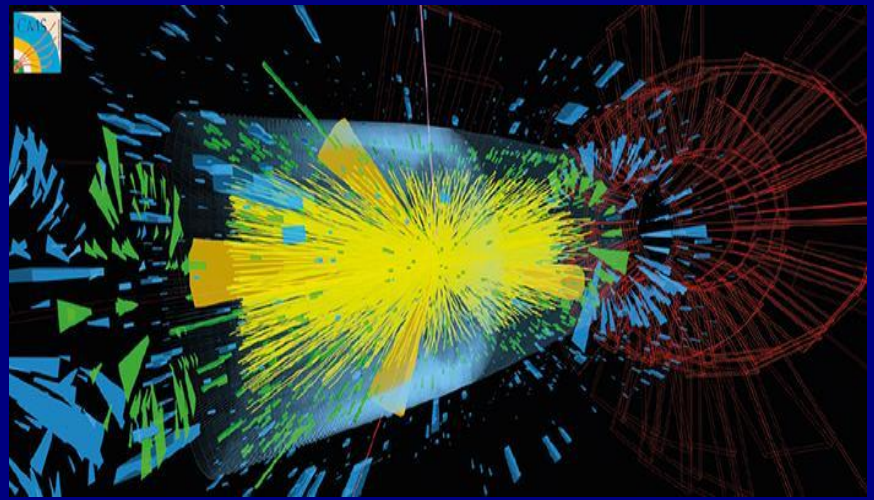



- **hls4ml** is a package for translating neural networks to FPGA firmware for inference with extremely low latency on FPGAs
  - https://github.com/hls-fpga-machine-learning/hls4ml
  - https://fastmachinelearning.org/hls4ml/
  - `pip install hls4ml`

- In this session you will get a brief introduction and hands on experience with the **hls4ml** package

- We'll learn how to:
  - Translate models into synthesizable FPGA code
  - Explore the different handles provided by the tool to optimize the inference
    - Latency, throughput, resource usage
  - Make our inference more computationally efficient with pruning and quantization

# hls4ml origins: Triggering at LHC

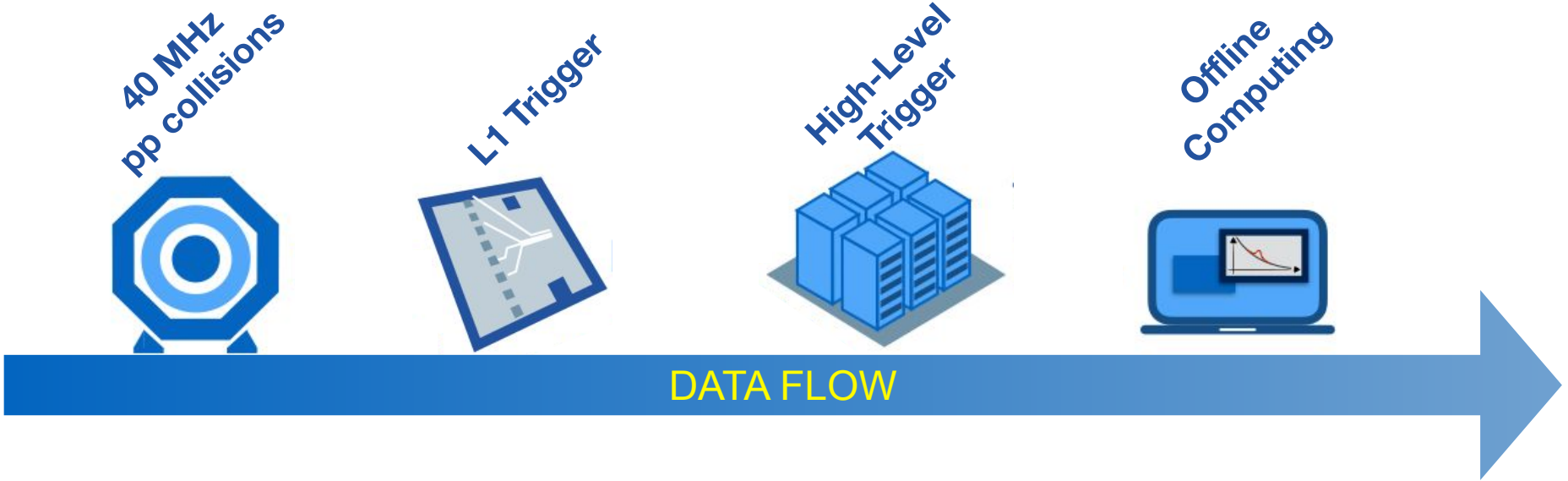


- Extreme collision frequency of **40 MHz** ⟹ extreme data rates **~100 TB/s**
- Most collision “events” don’t produce interesting physics

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

- ML can improve sensitivity to rare physics, but needs to be ***fast****!*

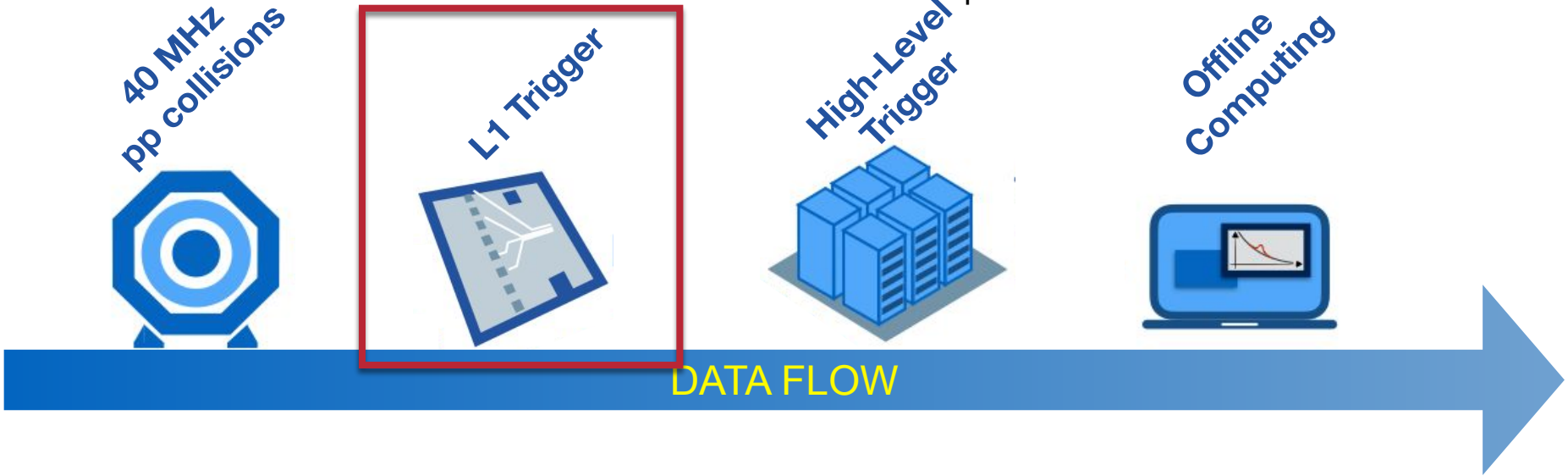
# LHC Experiment Data Flow



# LHC Experiment Data Flow

Deploy ML algorithms very early, avoiding off-line computation and storage

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

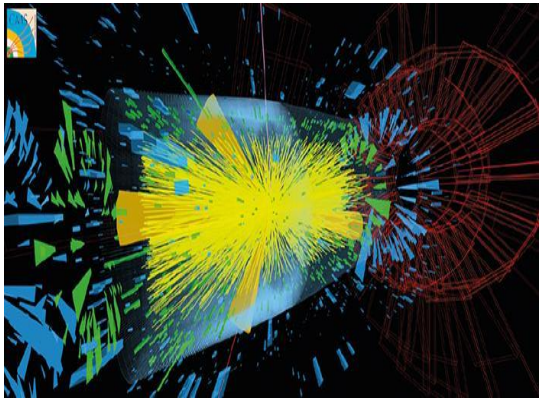


# Latency - Visualized

~1-3 seconds

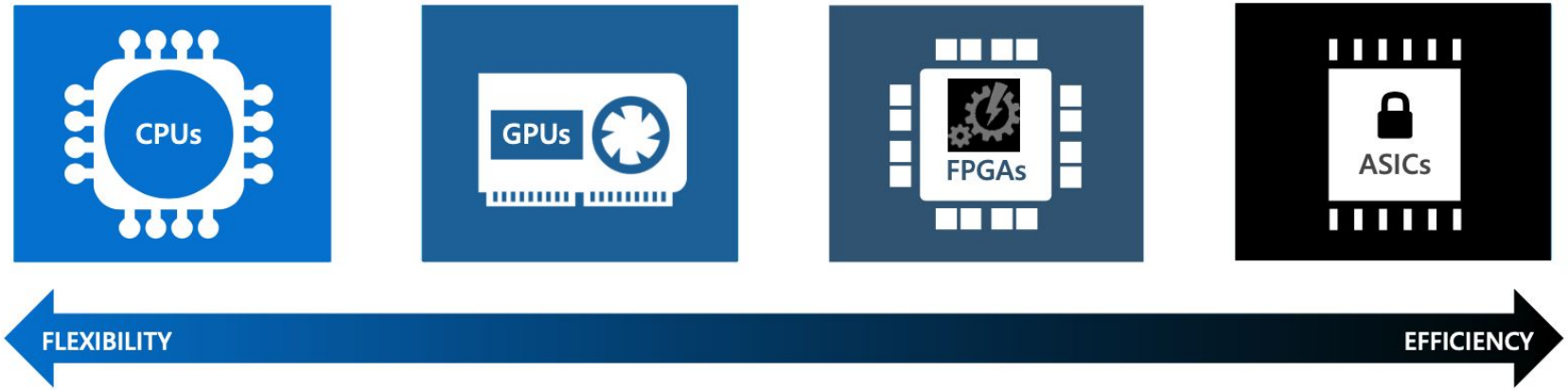

~50ms

~500 ns

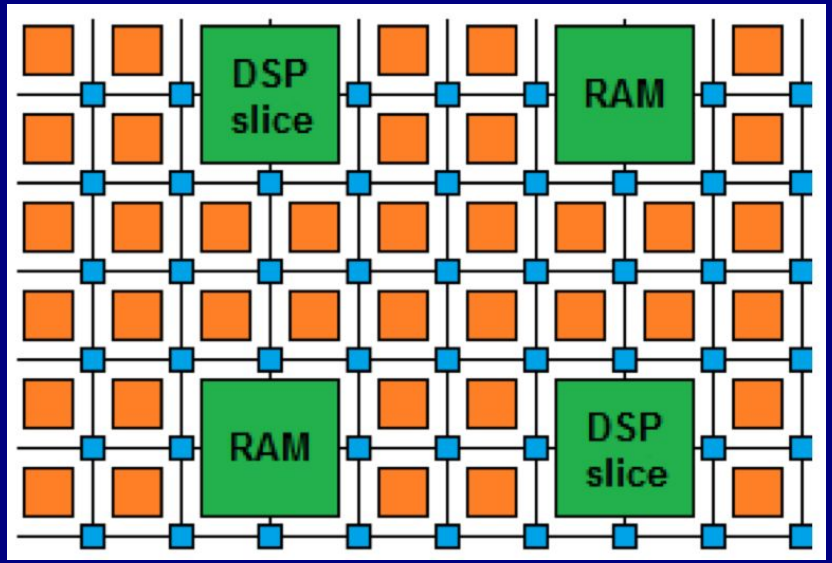


# Why FPGAs?

- Custom hardware acceleration, precisions and memory management:
  - E.g., GPUs offer very little support for arbitrary (e.g., 2-bit) precisions
- Data-flow architecture with no scheduling or control overheads:
  - E.g., A CPU has an operating system, a GPU has a workload scheduler etc.

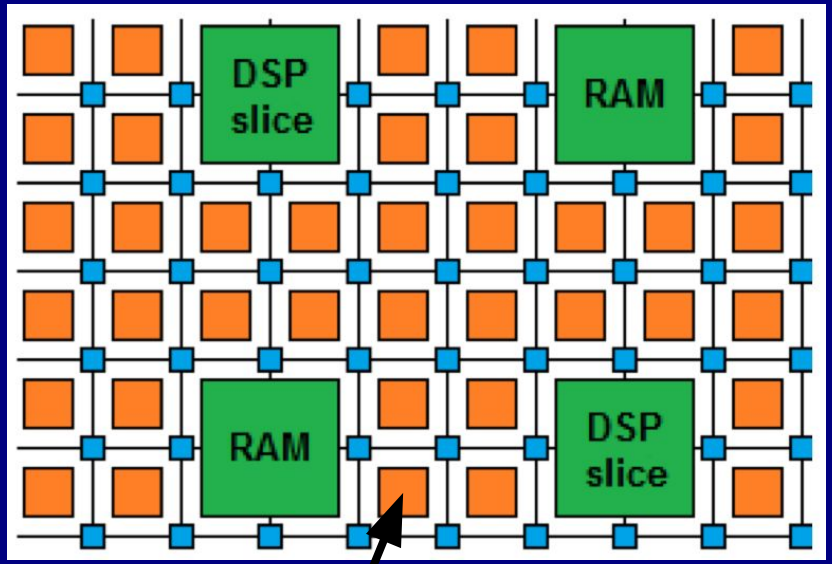


# What are FPGAs?

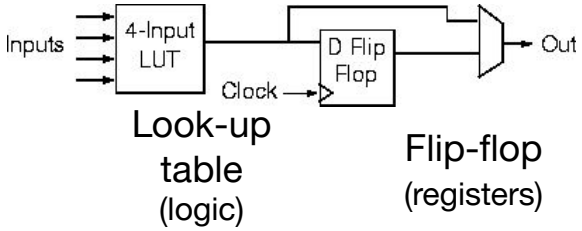



- **Field Programmable Gate Arrays** are reprogrammable integrated circuits
- Contain many different building blocks (‘resources’) which are connected together as you desire
- Originally popular for prototyping ASICs, but now also for high performance computing

# What are FPGAs?

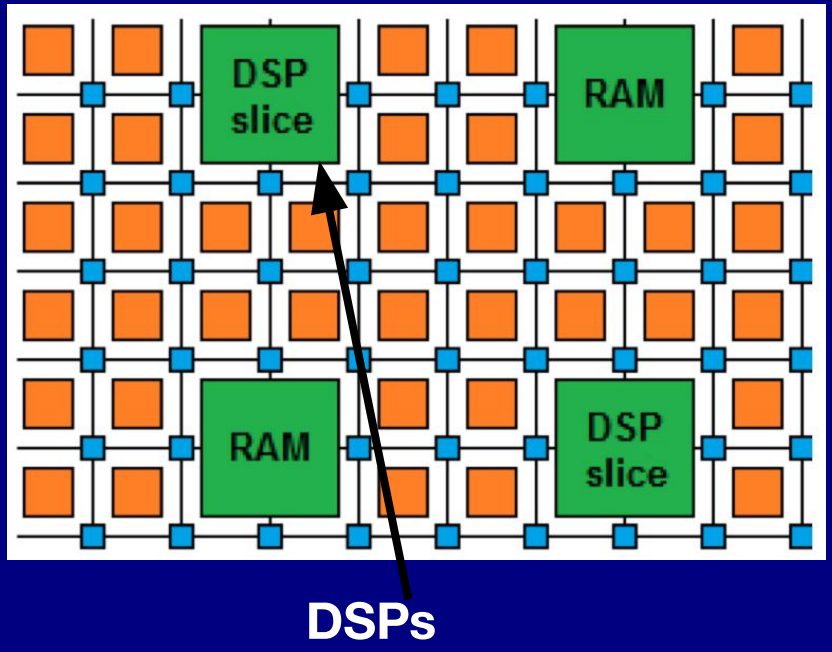



- **Logic cells / Look Up Tables** perform arbitrary functions on small bitwidth inputs (2-6)
- These can be used for boolean operations, arithmetic, small memories



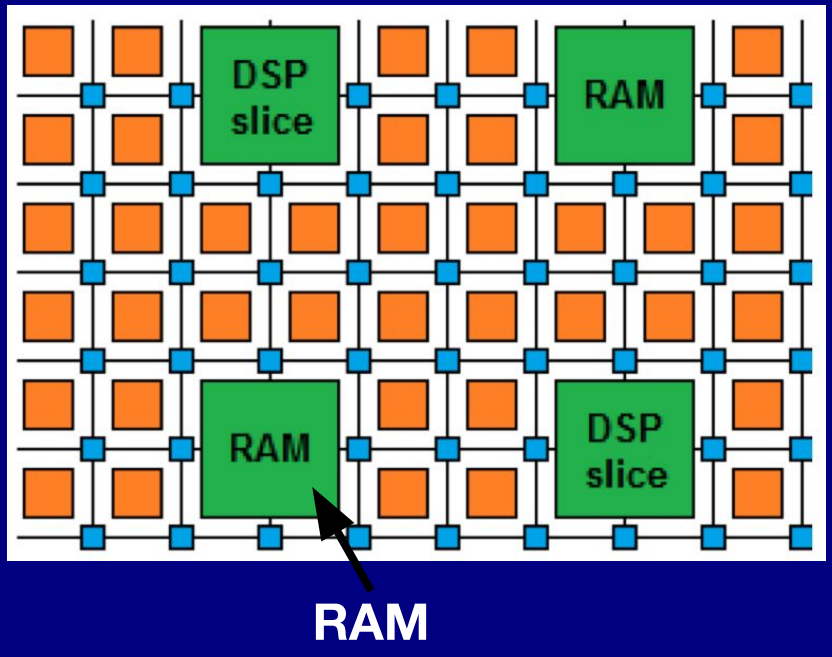

- **Flip-Flops** register data in time with the clock pulse

# What are FPGAs?



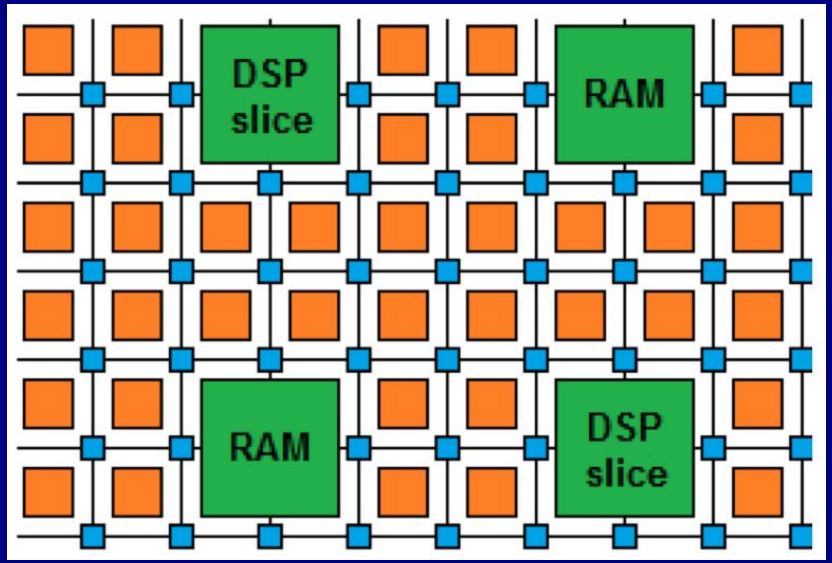

- **DSPs (Digital Signal Processor)** are specialized units for multiplication and arithmetic
- Faster and more efficient than using LUTs for these types of operations
- And for Neural Nets, DSPs are often the most scarce:
  - Even high-end FPGAs will have ”only” around 10,000 DSPs, corresponding to 10,000 multiplications with DSPs at any given moment

# What are FPGAs?




- **BRAMs** are small, fast memories - RAMs, ROMs, FIFOs (18Kb each in Xilinx)
  - Can access data in one clock cycle
  - Memories using BRAMs is more efficient than using LUTs
- A big FPGA has nearly 40Mb of BRAM, chained together as needed
- Recent accelerators cards also come equipped with off-chip HBM memory (up to 800 GBps)

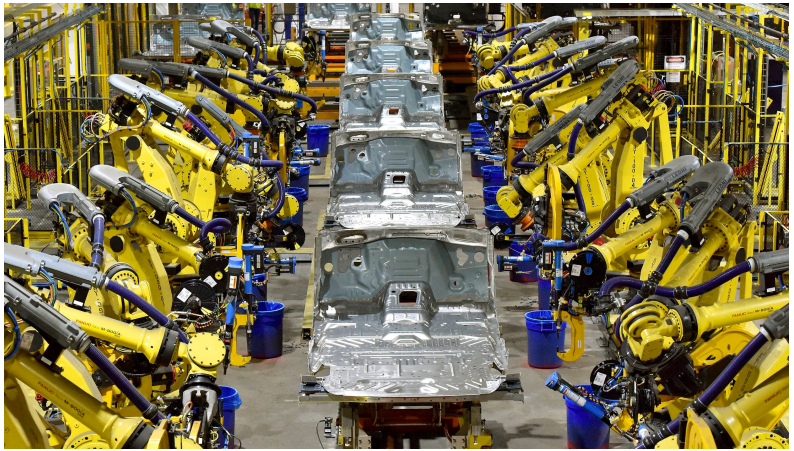
# What are FPGAs?




- In addition, there are specialised blocks for I/O, making FPGAs popular in embedded systems and HEP triggers
- **High speed transceivers** with Tb/s total bandwidth
  - PCIe, (Multi) Gigabit Ethernet, Infiniband
- Support **highly parallel** algorithm implementations
- **Low power per Op** (relative to CPU/GPU)

# Why are FPGAs *Fast*?

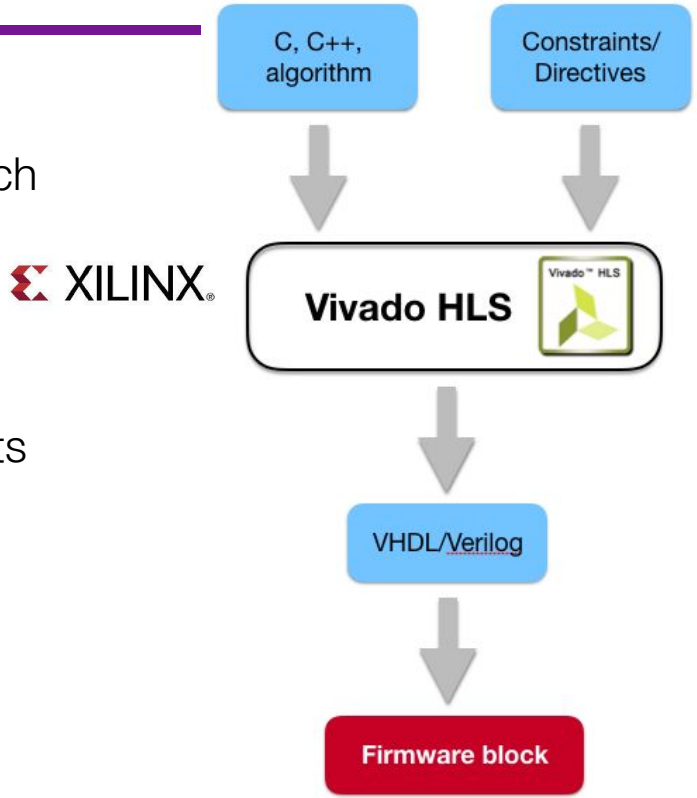
- Fine-grained / resource parallelism
  - Use the many resources to work on different parts of the problem simultaneously
  - Allows us to achieve low latency
- Most problems have at least some sequential aspect, limiting how low latency we can go
  - But we can still take advantage of it with…
- Pipeline parallelism
  - Use the register pipeline to work on different data simultaneously
  - Allows us to achieve high throughput



*Like a production line for data…*

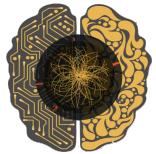
# How are FPGAs programmed?

- **Hardware Description Languages**
  - HDLs are programming languages which describe electronic circuits
- **High Level Synthesis**
  - Compile from C/C++ to VHDL
  - Pre-processor directives and constraints used to optimize the design
  - Drastic decrease in firmware development time!
- Today we'll use **AMD/Xilinx Vitis HLS** [*]




[*] https://docs.amd.com/r/2023.2-English/ug1399-vitis-hls/Introduction

# How are FPGAs programmed?



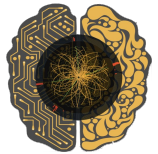

```
for(int i = 0; i < 16; i++) {
    #pragma HLS UNROLL 4
    c[i] = a[i] + b[i];
}
```

- Loop parallelised 4 times in each clock cycle
- Total block execution: 16 / 4 = 4cc

```
float weights[16]
#pragma HLS ARRAY_RESHAPE variable=weights block_factor=block_factor
#pragma HLS RESOURCE variable=weights core=ROM_2P_BRAM
```

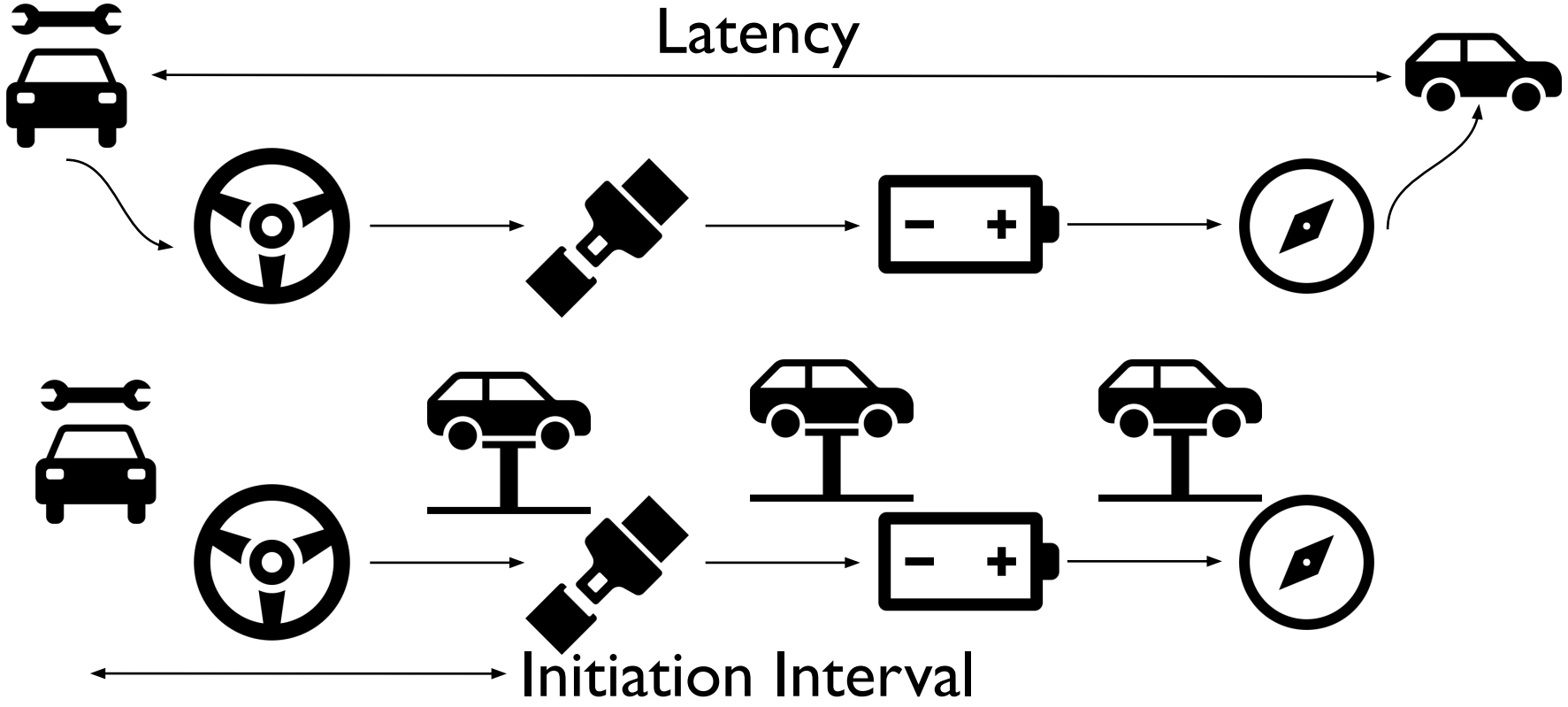
- Reshapes the weight matrix according to a specific block factor
- Stores the resulting array in on-chip ROM

# Jargon

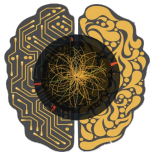



- **LUT - Look Up Table** (aka ‘logic’) - generic functions on small bitwidth inputs. Combine many to build the algorithm
- **FF - Flip Flops** - control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- **DSP - Digital Signal Processor** - performs multiplication and other arithmetic in the FPGA
- **BRAM - Block RAM** - hardened RAM resource. More efficient memories than using LUTs for more than a few elements
- **HLS** - High Level Synthesis - compiler for C, C++, SystemC into FPGA IP cores
- **HDL** - Hardware Description Language - low level language for describing circuits
- **RTL** - Register Transfer Level - the very low level description of the function and connection of logic gates
- **Latency** - time between starting processing and receiving the result
  - Measured in clock cycles or seconds
- **II - Initiation Interval** - time from accepting first input to accepting next input

# Latency vs Initialization Interval



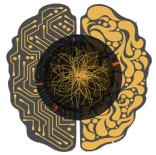
# What is **hls4ml** today?




A generic framework for FPGA acceleration of neural networks:

- **Front-end agnostic**: Keras, PyTorch, (Q)ONNX
- **Back-end agnostic**:  Vivado HLS, Vitis HLS, Intel HLS, oneAPI etc.
- **Many supported layers**: Dense, Conv, Recurrent, Graph etc.
- **High configurability**: Tune precision, reuse factor, custom layers etc.
- **An active, open-source community**: Many collaborators from many different fields and institutions

# What is **hls4ml** today?




**About**

Machine learning on FPGAs using HLS

fastmachinelearning.org/hls4ml

python machine-learning fpga neural-network hls keras pytorch vivado vivado-hls onnx intel-hls

Readme
Apache-2.0 license
Cite this repository
Activity
Custom properties
**1.3k** stars
**58** watching
**409** forks
Report repository

Fast inference of deep neural networks in FPGAs for particle physics [PDF] iop.org

J Duarte, S Han, P Harris, S Jindariani, E Kreinar, B Kreis, J Ngadiuba, M Pierini, R Rivera…

Journal of instrumentation, 2018 • iopscience.iop.org

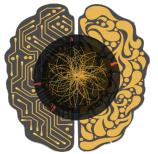
**Abstract**

Recent results at the Large Hadron Collider (LHC) have pointed to enhanced physics capabilities through the improvement of the real-time event processing techniques. Machine learning methods are ubiquitous and have proven to be very powerful in LHC physics, and particle physics as a whole. However, exploration of the use of such techniques in low-latency, low-power FPGA (Field Programmable Gate Array) hardware has only just begun. FPGA-based trigger and data acquisition systems have extremely

MEHR ANZEIGEN

☆ Speichern Zitiere Zitiert von: 546 Ähnliche Artikel Alle 12 Versionen Web of Science: 249

# hls4ml Use Cases




REAL-TIME SEMANTIC SEGMENTATION ON FPGAS FOR AUTONOMOUS VEHICLES WITH HLS4ML

…and many more in science, quantum, biomedical etc!

Nicolò Ghielmetti, Vladimir Loncar, Maurizio Pierini, Marcel R
European Organization for Nuclear Research (CEF
CH-1211 Geneva 23, Switzerland

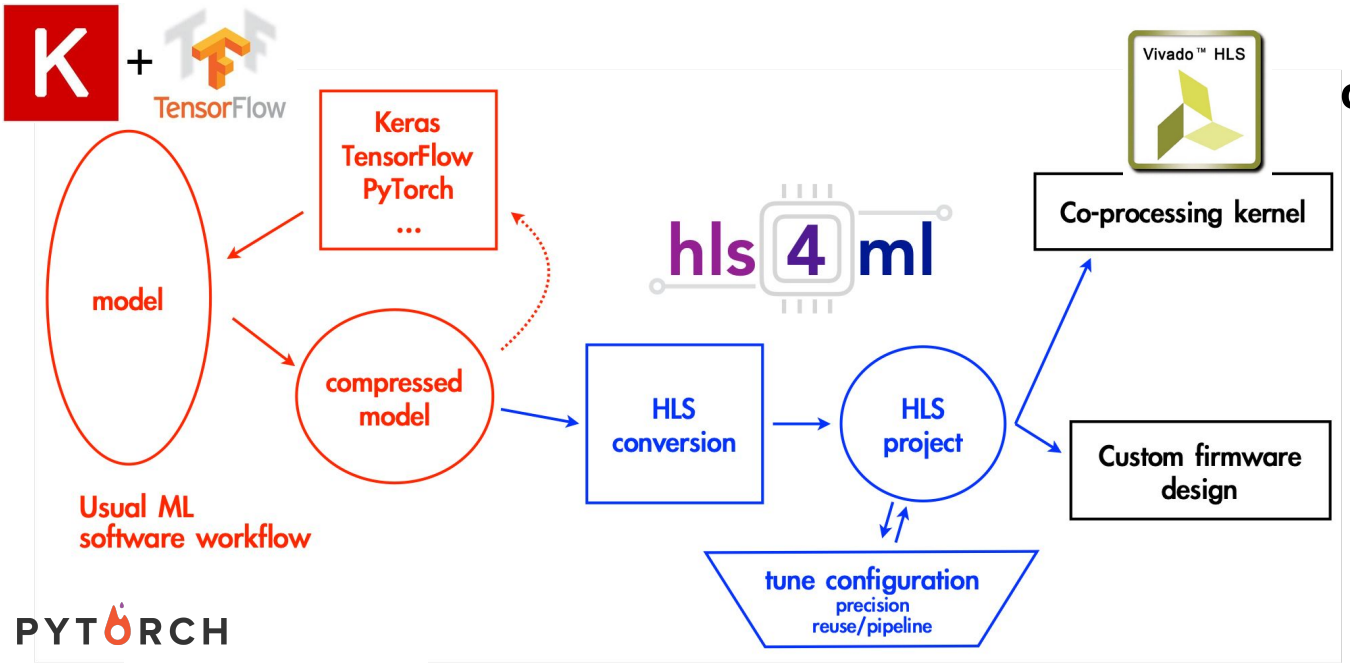
SoC-based implementation of 1D Convolutional Neural Network for 3-Channel ECG Arrhythmia Classification via HLS4ML

Feroz Ahmad, Saima Zafar, *Senior Member, IEEE*

RDMA Deep Packet Inspection at Line Rate with FPGAs

Maximilian I. Heer, Benjamin Ramhorst, Gustavo Alonso

# high level synthesis for machine learning




oneAPI

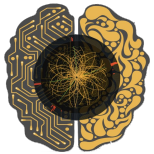
Mentor Catapult

PYTORCH

ONNX

https://fastmachinelearning.org/hls4ml/

# Today's Tutorial




**Part 1**:

- Get started with **hls4ml**: train a basic model and run the conversion, simulation & C-synthesis steps

**Part 2**:

- Train using QKeras / Brevitas "quantization-aware training" to study impact on FPGA metrics

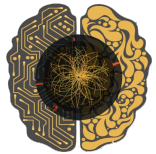
**Part 3** :

- Learn how to tune inference performance with ReuseFactor and profiling

**Part 4**:

- Test on CNN

# What's not covered Today?

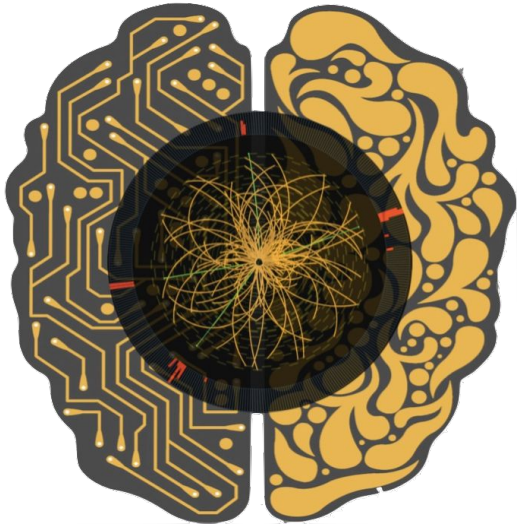



- **Boosted decision trees**: implemented in a companion package to hls4ml
  - https://github.com/thesps/conifer

- What comes after **hls4ml**… you would need to integrate the 'IP core' into a larger design
  - For a custom board, you'd need to do this by hand (e.g. CMS L1 Trigger)
  - For more off-the-shelf boards, integration with system-on-chip or host CPU can be more straightforward, using tools such as XRT

# Hands On - Setup

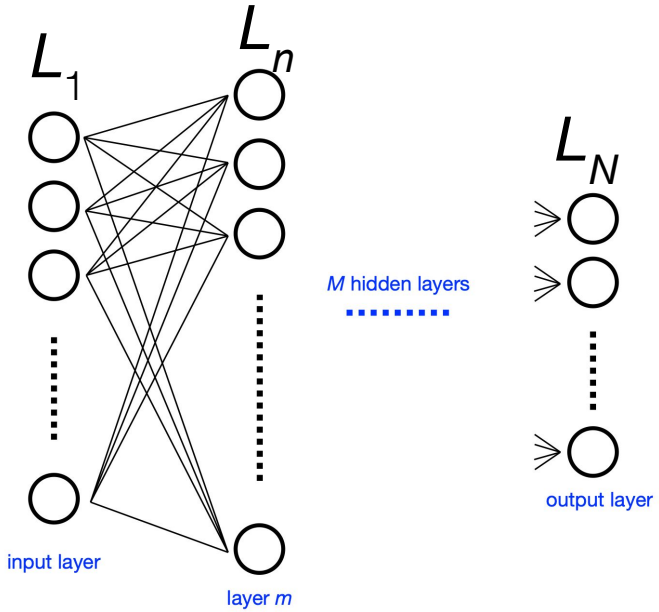
- The interactive part is served with Python notebooks
- Launch and play online using Binder:
  https://mybinder.org/v2/gh/enlupi/hls4ml-tutorial/SchoolAIandMLatLHC-Jul2026?urlpath=lab
- It is also possible to run locally by creating the appropriate env
  `conda env create -f environment.yml`
- Vitis synthesis cannot be run online.
  I will have some pre-synthesised designs to check the reports in case



# **hls4ml** tutorial

## *Part 1: Model Conversion*

# Neural network inference




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

precomputed and stored in BRAMs

DSPs

logic cells

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

16 inputs

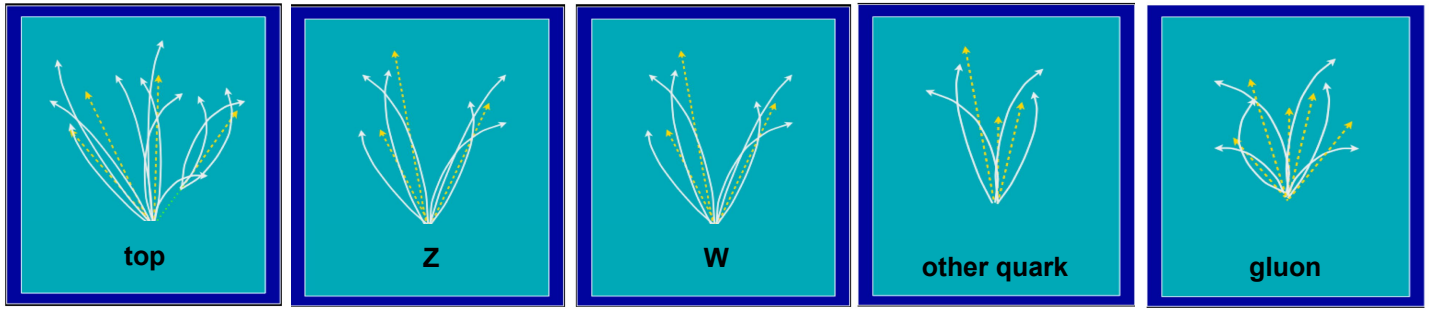
How many resources? DSPs, LUTs, FFs?
Does the model fit in the latency requirements?

5 outputs
activation: SoftMax

# Physics case: jet tagging

Study a **multi-classification task to be implemented on FPGA:** discrimination between highly energetic (boosted) ***q, g, W, Z, t*** initiated *jets*

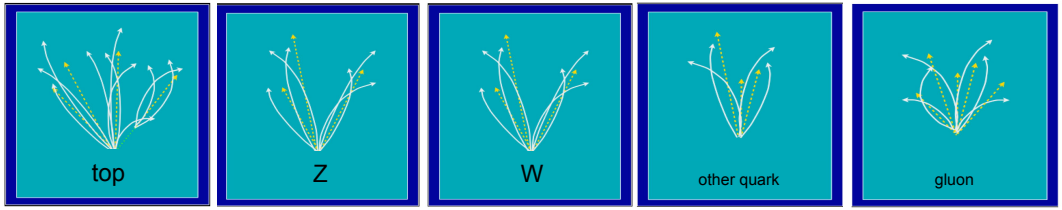
*Jet* = collimated ‘spray’ of particles

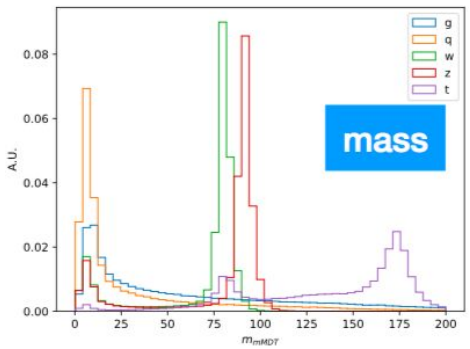


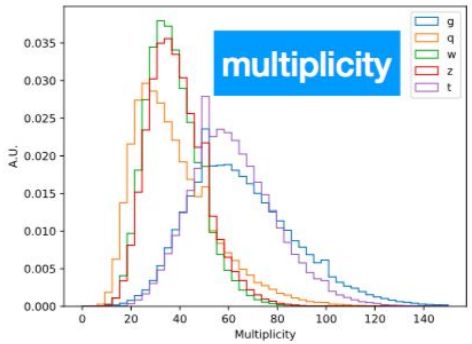

| **t→bW→bqq** | **Z→qq** | **W→qq** | **q/g background** |
|---|---|---|---|
| 3-prong jet | 2-prong jet | 2-prong jet | no substructure and/or mass ~ 0 |

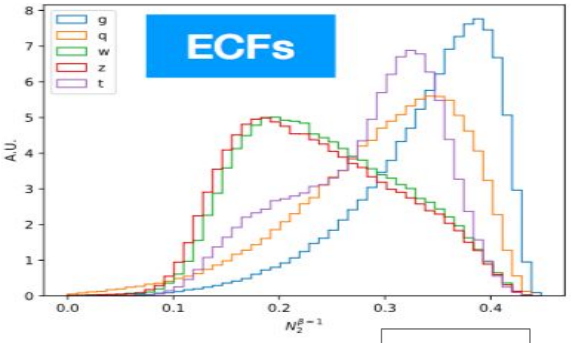
Reconstructed as one massive jet with substructure

# Physics case: jet tagging



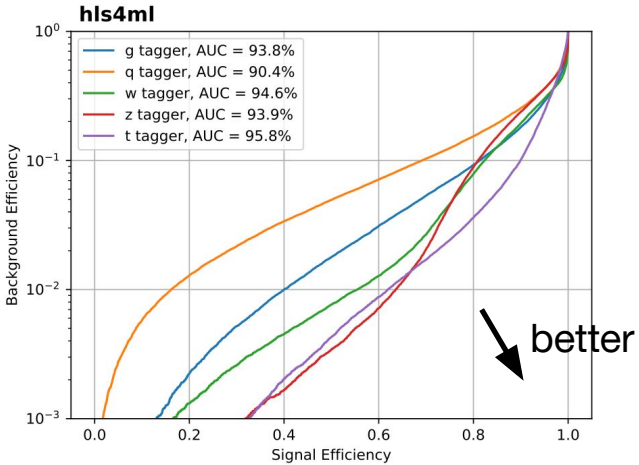

**Input variables: several observables known to have high discrimination power from offline data analyses and published studies [*]**

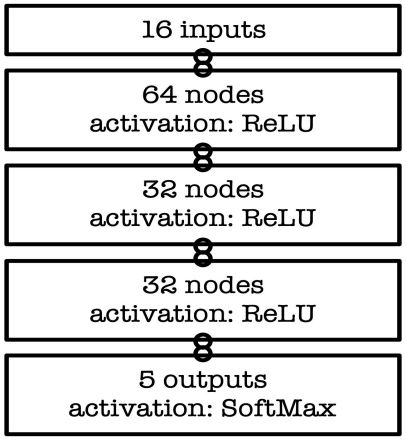
[*] D. Guest at al. PhysRevD.94.112002, G. Kasieczka et al. JHEP05(2017)006, J. M. Butterworth et al. PhysRevLett.100.242001, etc..

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

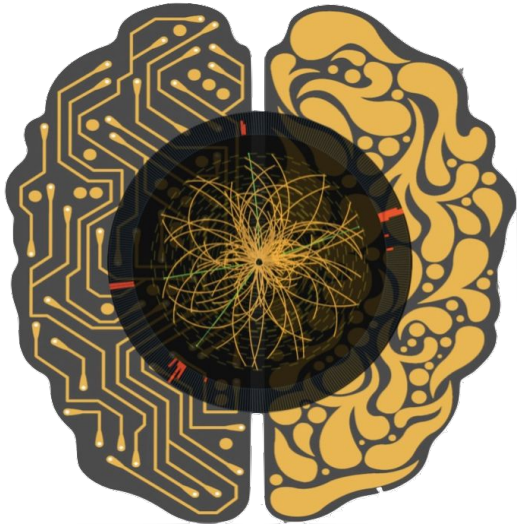
# Physics case: jet tagging

- We'll train the **five class multi-classifier** on a sample of ~ 1M events with two boosted WW/ZZ/tt/qq/gg anti-$k_T$ jets
  - Dataset DOI: 10.5281/zenodo.3602254
  - OpenML: https://www.openml.org/d/42468
- Fully connected neural network with **16 expert-level inputs**:
  - Relu activation function for intermediate layers
  - Softmax activation function for output layer



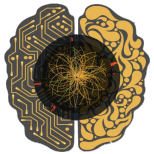

AUC = area under ROC curve
(100% is perfect, 20% is random)



# **hls4ml** Tutorial

## *Part 2: Compression & Quantization*

# NN Compression Methods

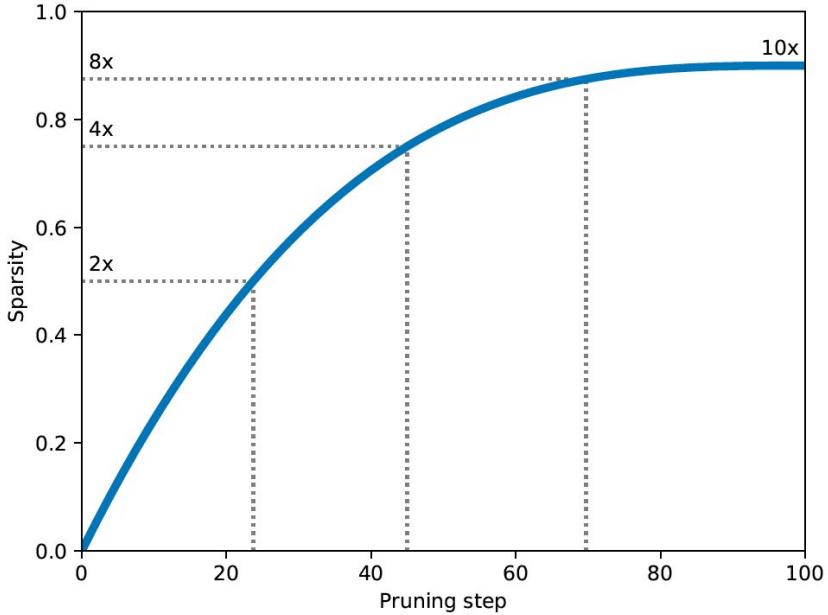



- Network compression is a widespread technique to reduce the size, energy consumption, and overtraining of deep neural networks

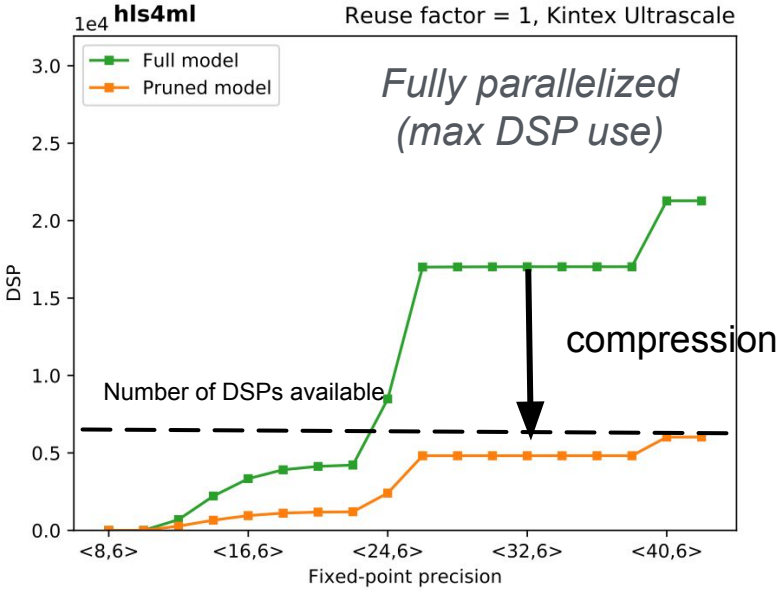
- Several approaches have been studied:
  - **parameter pruning:** selective removal of weights based on a particular ranking [arxiv.1510.00149, arxiv.1712.01312]
  - **low-rank factorization:** using matrix/tensor decomposition to estimate informative parameters [arxiv.1405.3866]
  - **transferred/compact convolutional filters:** special structural convolutional filters to save parameters [arxiv.1602.07576]
  - **knowledge distillation:** training a compact network with distilled knowledge of a large network [doi:10.1145/1150402.1150464]

# TF Sparsity

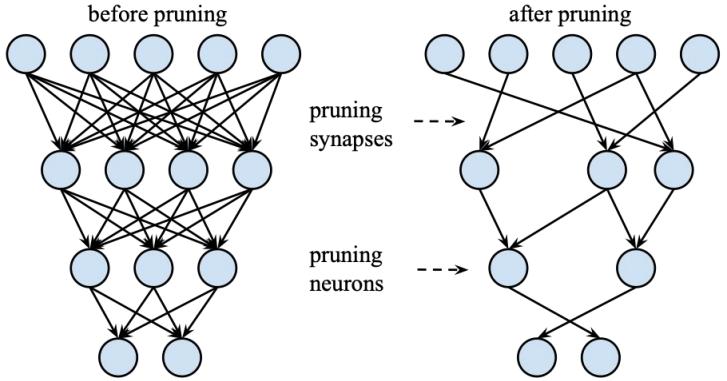
- Iteratively remove low magnitude weights, starting with 0 sparsity, smoothly increasing up to the set target as training proceeds



# Efficient NN Design: Compression



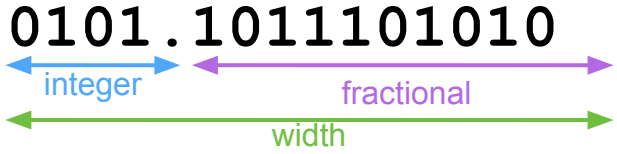

*70% compression ~ 70% fewer DSPs*




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

# Efficient NN Design: Quantization

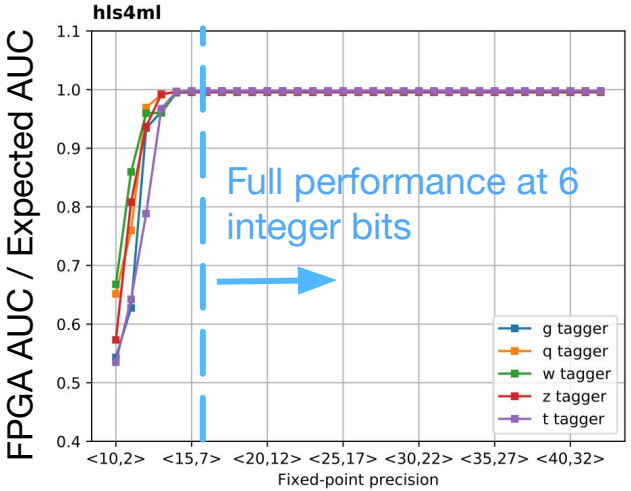
ap_fixed<width bits, integer bits>




- In the FPGA we use fixed point representation
  - Operations are integer ops (very fast!), but we can represent fractional values
- But we have to make sure we've used the correct data types!

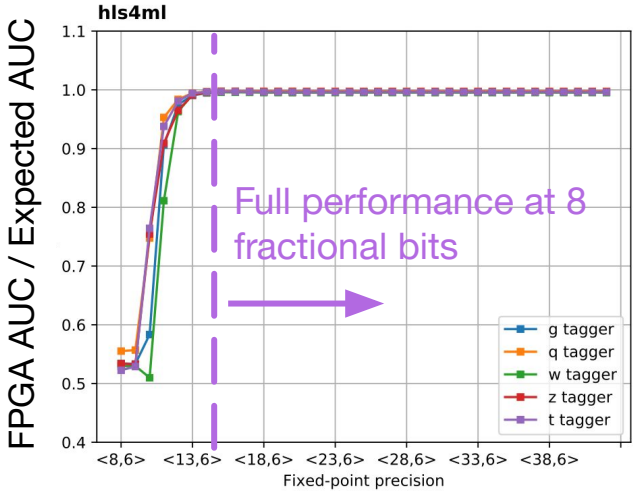
## Scan integer bits

Fractional bits fixed to 8



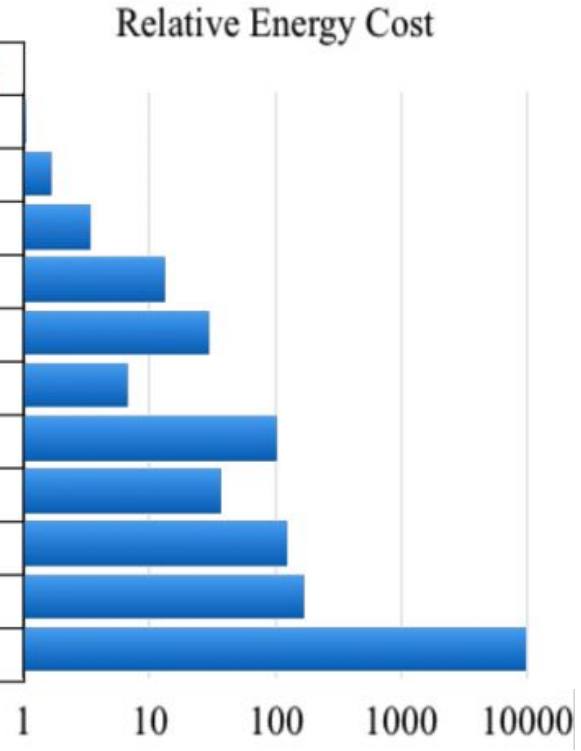

## Scan fractional bits

Integer bits fixed to 6



# Efficient NN Design: Quantization

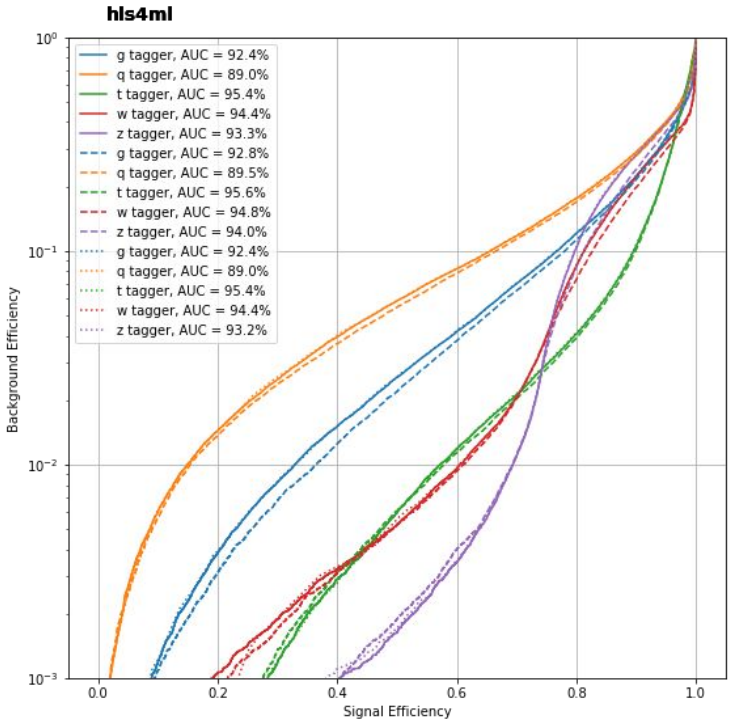
| Operation: | Energy (pJ) |
|---|---|
| 8b Add | 0.03 |
| 16b Add | 0.05 |
| 32b Add | 0.1 |
| 16b FP Add | 0.4 |
| 32b FP Add | 0.9 |
| 8b Mult | 0.2 |
| 32b Mult | 3.1 |
| 16b FP Mult | 1.1 |
| 32b FP Mult | 3.7 |
| 32b SRAM Read (8KB) | 5 |
| 32b DRAM Read | 640 |




[Horowitz @ ISSCC'14]

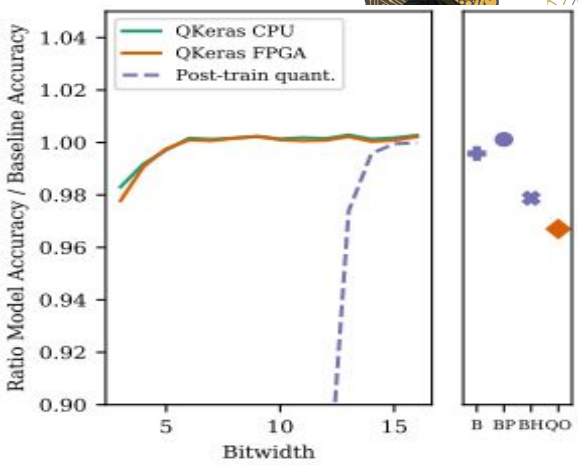
# Efficient NN Design: Quantization

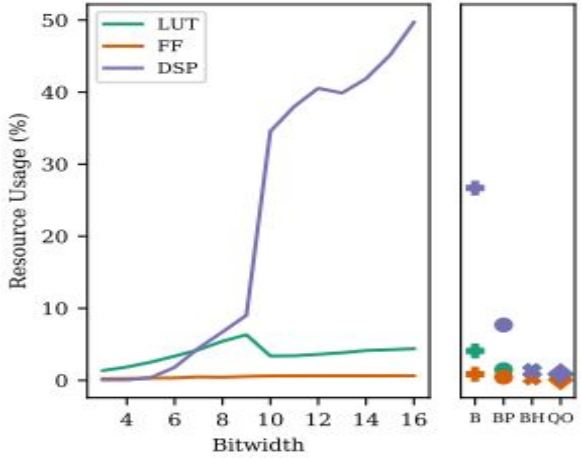
- **hls4ml** allows you to use different data types everywhere
- We will also try quantization-aware training with QKeras (but many more tools are available!)
- Quantisation-aware training enables training models with very low precision:
  - Out-performs post-training quantisation significantly
  - At a high level, it performs the forward pass with reduced precision and the backward pass in floating point precision
  - Possible to achieve very precisions (binary and ternary models)



# QKeras

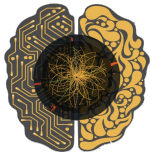
- QKeras is a library to train models with quantization in the training
  - Developed by Google[*]
- Easy to use, drop-in replacements for Keras layers
  - e.g. Dense → QDense
  - e.g. Conv2D → QConv2D
  - Use ‘quantizers’ to specify how many bits to use where
  - Same kind of granularity as hls4ml
- Can achieve good performance with very few bits
- Stable support for QKeras-trained models to hls4ml
  - The number of bits used in training is automatically parsed for conversion & inference
  - The intermediate model is adjusted to capture all optimizations possible with QKeras




*[*]* https://github.com/google/qkeras and doi:10.1038/s42256-021-00356-5

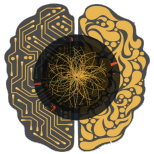
# Other tools: HGQ




- HGQ is QAT library built on top of Keras that offers:
  - **High Granularity:** supports per-weight and per-activation bitwidth optimization, or any other lower granularity
  - **Automatic Quantization:** HGQ can automatically optimize the bitwidth of all parameters during training (pruning is included as 0 bitwidth reduction)
  - **Bit-accurate conversion to hls4ml**
  - **Accurate Resource Estimation:** through **EBOPs** (effective bit operations) metric.
    Can be used as a regularization term in the loss function, controlled by a hyperparameter $\beta$ (trade-off between accuracy and resource usage)
- HGQ 2 (based on Keras 3) is already available

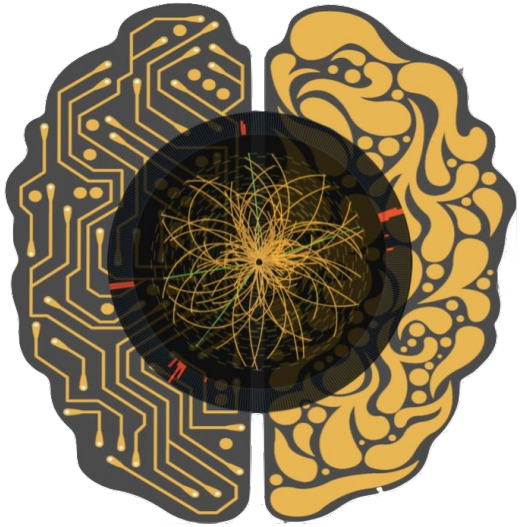
*[*]* https://github.com/calad0i/HGQ and doi:10.7907/HQ8JD-RHG30

# Other tools: PQuantML🌶️




- PQuantML🌶️ is a tool to train and optimize compressed neural networks for hardware:
  - **Pruning:** Includes various pruning algorithms (unstructured, N:M, structured)
  - **Quantization:** Supports fixed-point quantization and HGQ
- Automatic model wrapping, training and cleanup: no detailed knowledge of compression techniques needed
- Provides **hyperparameter optimization** to find best compression and training config
- Seamless and bit-accurate **hls4ml** integration
  - compressed model is fed into hls4ml normally
  - the tool automatically deduces the correct precision
- Compatible with PyTorch and Keras

*[*]* https://github.com/cern-nextgen/PQuantML and *arXiv:2603.26595*



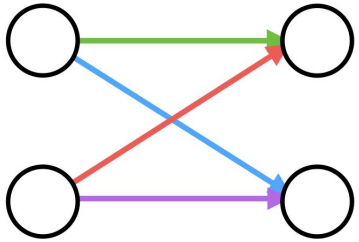
# **hls4ml** Tutorial

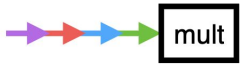
## *Part 3: Advanced Configuration*

# Efficient NN Design: Parallelization

- Trade-off between latency and FPGA resource usage determined by the parallelization of the calculations in each layer

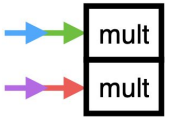
- Configure the **"reuse factor"** = number of times a multiplier is used to do a computation



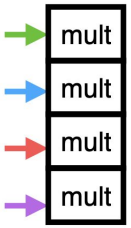



reuse = 4
use 1 multiplier 4 times

**Fully serial**




reuse = 2
use 2 multipliers 2 times each




reuse = 1
use 4 multipliers 1 time each

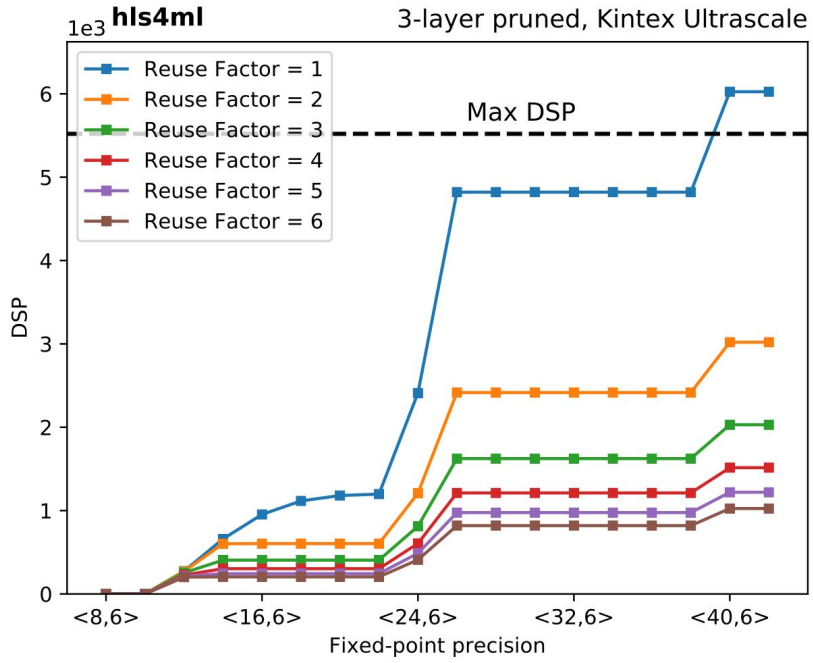
**Fully parallel**

**Fewer resources,**
**Lower throughput,**
**Higher latency**

**More resources,**
**Higher throughput,**
**Lower latency**

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

# Parallelization: DSP Usage




**More resources**

Fully parallel
Each mult. used 1x

Each mult. used 2x

Each mult. used 3x

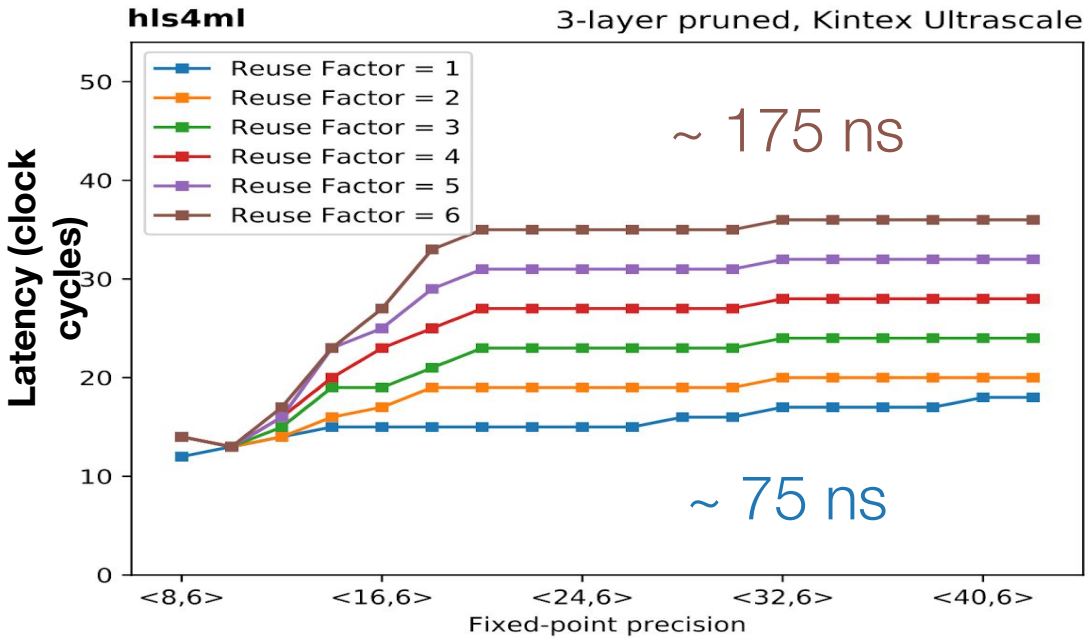
⋮

**Lower throughput**

# 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

# Large MLP

- '`Strategy: Resource`' for larger networks and higher reuse factor
- Here, we use a different partitioning on the first layer for the best partitioning of arrays

```
IOType: io_parallel # options: io_serial/io_parallel
HLSConfig:
  Model:
    Precision: ap_fixed<16,6>
    ReuseFactor: 128
  Strategy: Resource
  LayerName:
    dense1:
      ReuseFactor: 112
```

This config is for a model trained on the MNIST digits classification dataset
Architecture (fully connected): 784 → 128 → 128 → 128 → 10
Model accuracy: ~97%
**We can work out how many DSPs this should use...**

# Large MLP

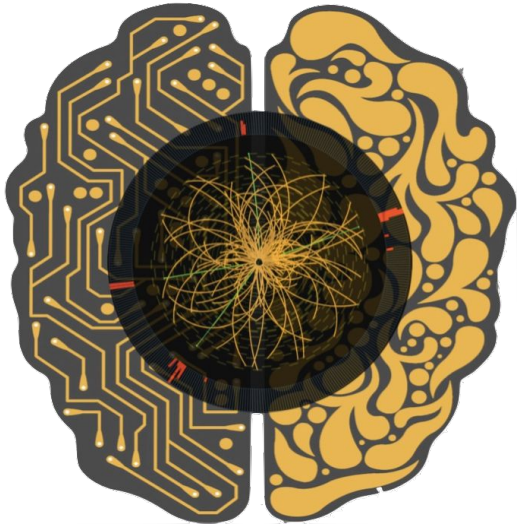
- It takes a while to synthesise, so here's one I made earlier…
- The DSPs should be: (784 x 128) / 112 + (2 x 128 x 128 + 128 x 10) / 128 = 1162 🤞

```
============================
============================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  |   5.00|     4.375|        0.62|
    +--------+-------+----------+------------+

+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+----------+
    |  Latency  |  Interval | Pipeline |
    | min | max | min | max |   Type   |
    +-----+-----+-----+-----+----------+
    |  518|  522|  128|  128| dataflow |
    +-----+-----+-----+-----+----------+
```

```
====================================
== Utilization Estimates
====================================
+---------------------+---------+-------+---------+--------+
|         Name        | BRAM_18K| DSP48E|    FF   |   LUT  |
+---------------------+---------+-------+---------+--------+
...
+---------------------+---------+-------+---------+--------+
|Total                |     1962|   1162|   169979|  222623|
+---------------------+---------+-------+---------+--------+
|Available SLR        |     2160|   2760|   663360|  331680|
+---------------------+---------+-------+---------+--------+
|Utilization SLR (%)  |       90|     42|       25|      67|
+---------------------+---------+-------+---------+--------+
|Available            |     4320|   5520|  1326720|  663360|
+---------------------+---------+-------+---------+--------+
|Utilization (%)      |       45|     21|       12|      33|
+---------------------+---------+-------+---------+--------+
```

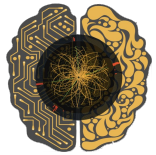
II determined by the largest reuse factor



# **hls4ml** Tutorial

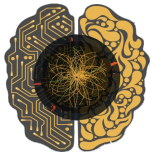
## *Part 4: Advanced Models*

# Hardware Implementation: *io_type*



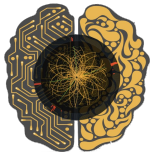

- Regulates **data transfer** between layers
- Selected at *model conversion* time
- Two modes supported:
  - **io_parallel:**
    - **Default data transfer type** for **hls4ml**
    - Directly wires the output of one layer to the input of the next layer
    - Suitable for small models with moderate resource usage
  - **io_stream:**
    - Uses **FIFO** buffers between layers
    - Used with larger models where some layer requires a significant II
    - Special **optimizer** is implemented **for Vivado/Vitis backend** to optimize FIFOs depth by performing RTL co-simulations

# Hardware Implementation: Strategy



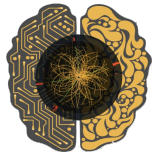

- Regulates how **CMVM** (Constant Matrix-Vector Multiplication) are implemented in hardware
  ⟹ Impact on overall latency and resource consumption of each layer
- Three possible solutions:
  - **Latency:**
    - **Default strategy** for **hls4ml**
    - Suitable for **small models**, targeting very low inference latencies
    - CMVM is unrolled and back end compiler automatically determines the exact hardware primitive mapping based on data type, precision, etc.
    - If MAC operations are implemented using fabric (e.g. LUTs) a higher RF will not necessarily result in resource reduction, but will still increase the II

# Hardware Implementation: Strategy




- Regulates how **CMVM** (Constant Matrix-Vector Multiplication) are implemented in hardware
  ⟹ Impact on overall latency and resource consumption of each layer
- Three possible solutions:
  - **Latency:**
  - **Resources:**
    - Increasing RF directly reduces the number of MAC units used: given a linear layer with ***N*** inputs, ***M*** outputs and reuse factor ***RF***, there will be $P = M \cdot N / RF$ multipliers operating in parallel.
    - Weights are stored in **BRAM**, partitioned for simultaneous read
    - Only certain numbers of RF are allowed such that the whole CMVM problem can be completely partitioned without remainder
    - **Latency is higher** than for Latency Strategy

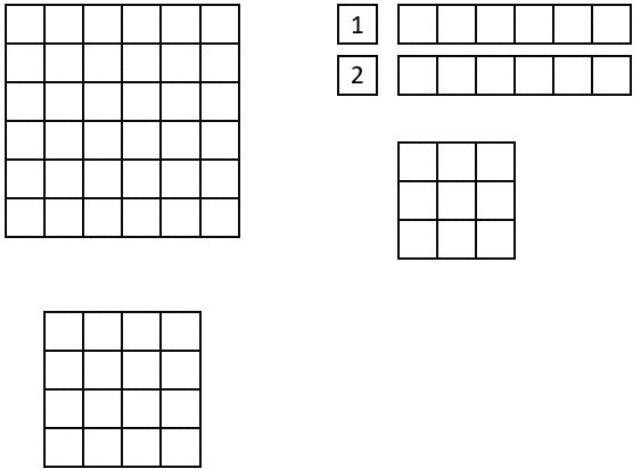
# Hardware Implementation: Strategy

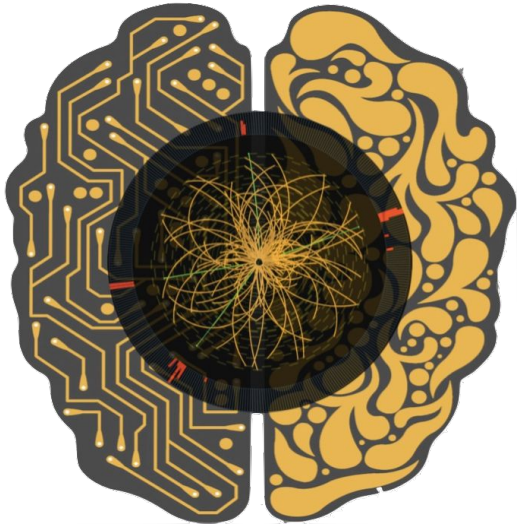



- Regulates how **CMVM** (Constant Matrix-Vector Multiplication) are implemented in hardware
  ⟹ Impact on overall latency and resource consumption of each layer
- Three possible solutions:
  - **Latency:**
  - **Resource:**
  - **Distributed Arithmetic (DA):**
    - Relies on external ***da4ml*** library:
      https://github.com/calad0i/da4ml and doi:10.5281/zenodo.14194660
    - Implements the CMVM operation by decomposing it into an **adder graph** with only shift-and-add (or subtract) operations
    - DA usually has **lower fabric resource utilization** and latency than Latency strategy. DSPs or other hardened multipliers will not be used due to the lack of explicit multiplications
    - In general does not support RFs greater than one

# Hardware Implementation: PF

- **PF** (Parallelization Factor) controls the number of CMVM operations that are performed in parallel
- Used in layers that perform identical CMVM operations multiple times on different inputs in one forward pass (e.g. im2col Conv layers)
- Independent of Strategy and RF used for the CMVM operation
- Higher PF reduces the II of one layer at the expense of higher resource usage
- PF must fully divide number of CMVM operations to avoid remainder
- Set with keyword `ParallelizationFactor` in layer config

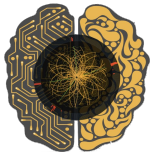




# **hls4ml** Tutorial

## *Conclusions*

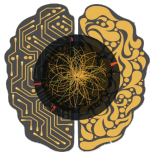
# Summary




- After this session you've gained some hands on experience with **hls4ml**
  - Translated neural networks to FPGA firmware, run simulation and synthesis
- Tuned network inference performance with precision and ReuseFactor
  - Used profiling and trace tools to guide tuning
- Trained a model with small number of bits using QKeras/Brevitas, and use the same spec in inference easily with **hls4ml**
- Converted a CNN model

- You can find these tutorial notebooks to run locally at: https://github.com/fastmachinelearning/hls4ml-tutorial
- Use hls4ml in your own environment: `pip install hls4ml[profiling]`

# Special Thanks




Thanks to Sioni Summers and Benjamin Ramhorst for providing the slides which the current presentation is based on.

Special thanks to the Fast Machine Learning Community for the on-going efforts in hls4ml development and its accompanying tutorials. The materials used today are based on: https://github.com/fastmachinelearning/hls4ml-tutorial and were used and edited with permission of the authors.