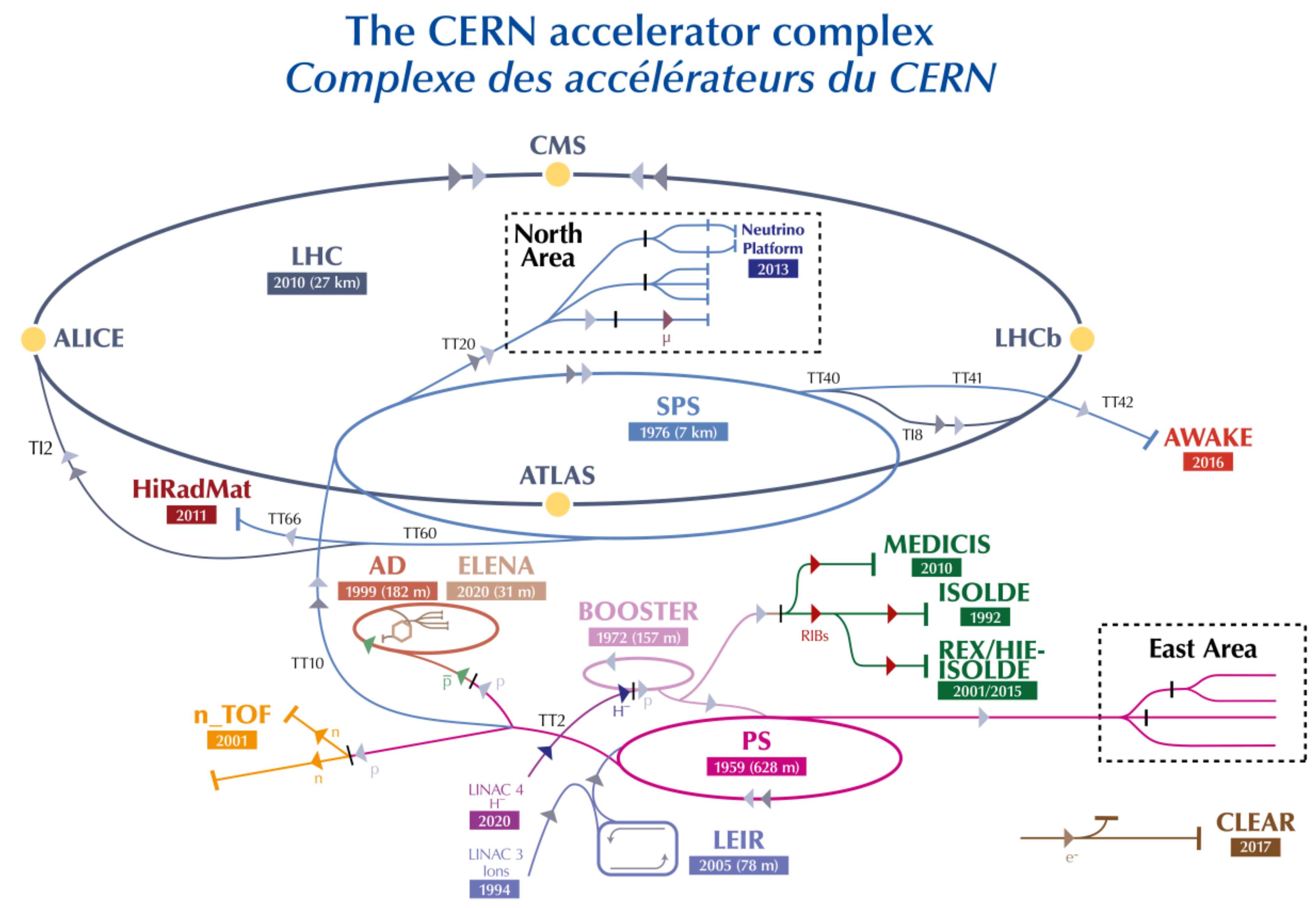
ML algorithms on FPGAs: Recent developments in hls4ml

Jovan Mitrevski for the hls4ml group
TWEPP 2022
Sept 21, 2022

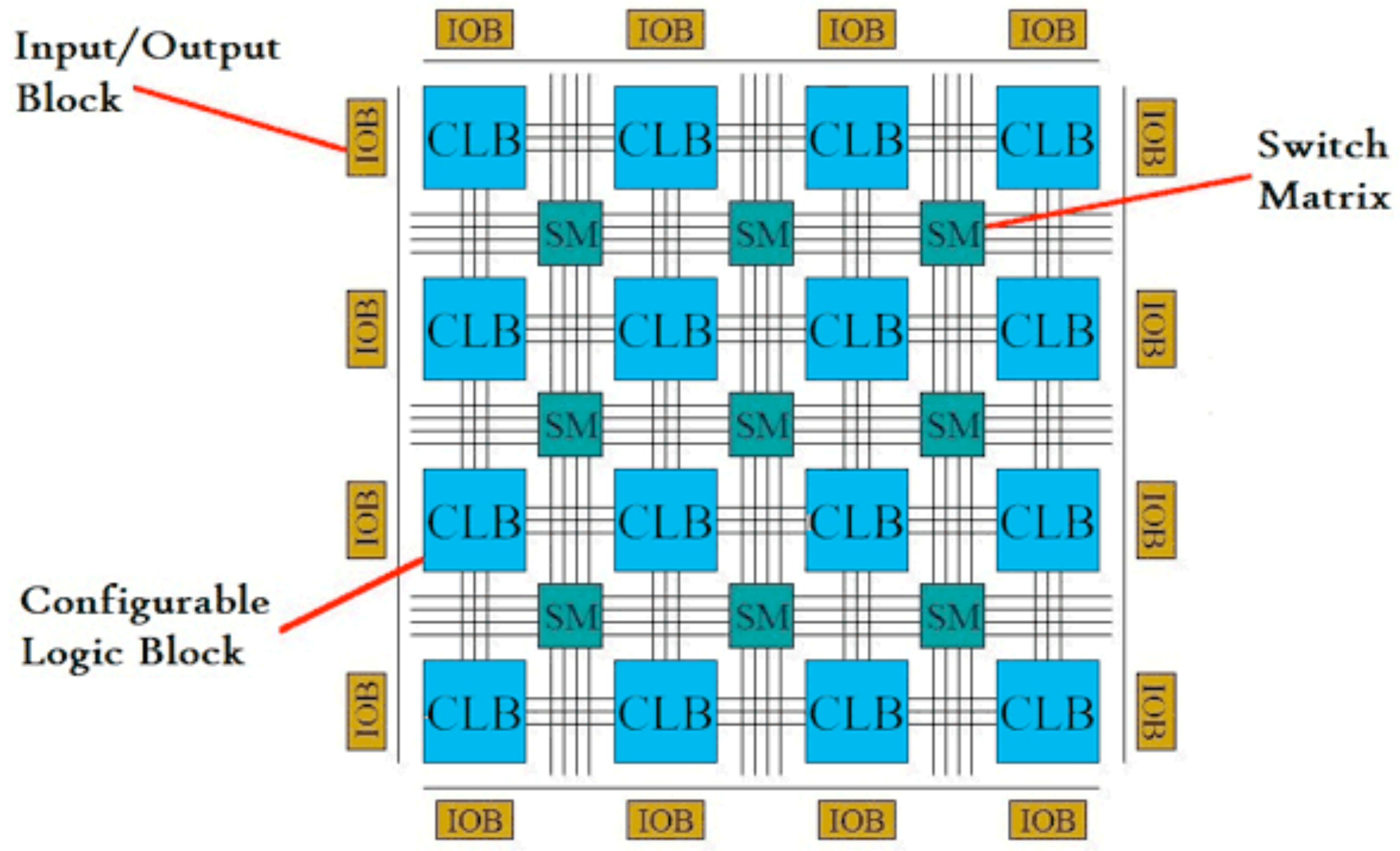
Motivation for hls4ml

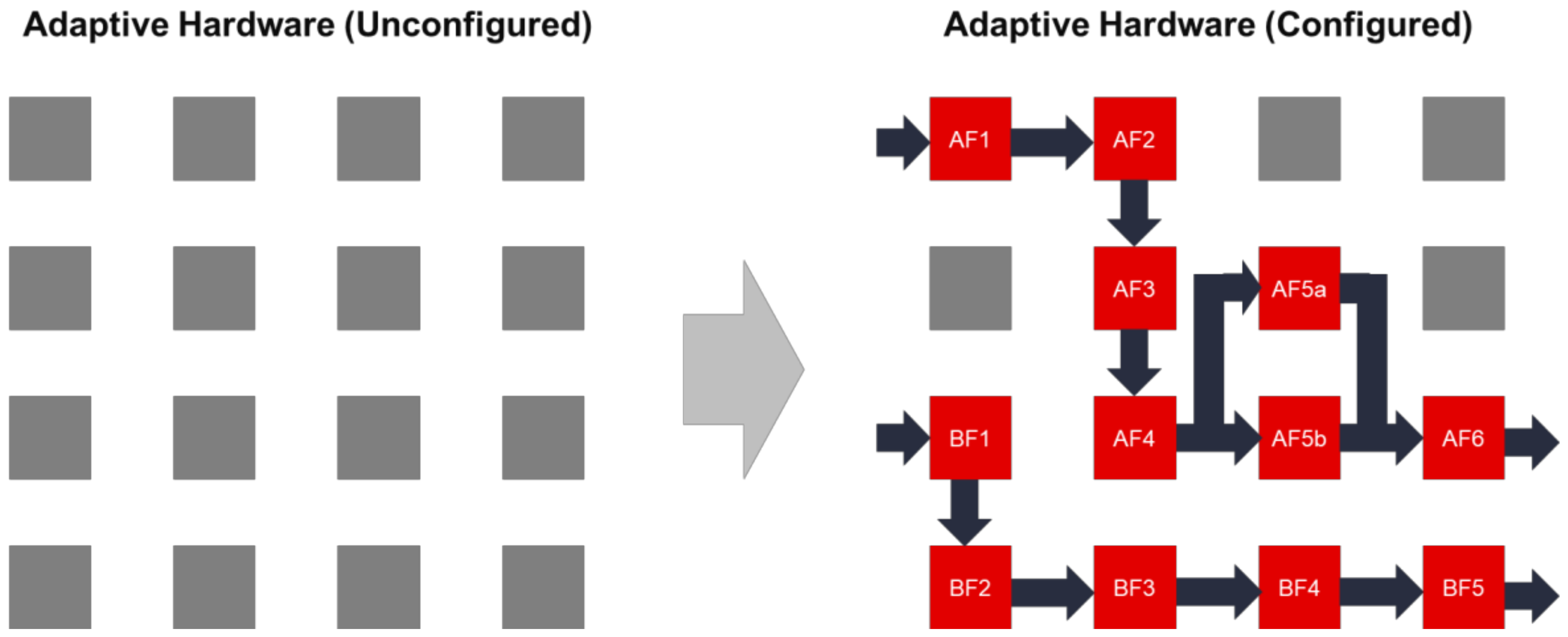
- hls4ml was originally created for use in the first level trigger of the LHC
- Collisions occur at 40 MHz, and trigger decisions need to be made in the order of a few μs.
- Need to reject most events, but efficiently accept interesting events: machine learning
- Original focus of hls4ml: implement relatively small NNs in FPGAs to execute very fast
  - Weights stored in the fabric, parallel execution
- Focus has subsequently broadened



Why use FPGAs to run ML inference?

- FPGAs exploit the parallelism of the problem for low latencies
- FPGAs exhibit predictable real-time latencies
- FPGAs tend to use less power than GPUs or CPUs for solving similar problems
- FPGAs can be reprogrammed as algorithms evolve





From Xilinx Adaptive Computing Technology Overview

How does one program FPGAs?

- Hardware description languages (HDLs) like VHDL or Verilog
  - Closely tied to the hardware implementation: can be complicated

HDL

- High Level Synthesis (HLS)
  - Use (restricted) C++ code with pragmas
    - Main restriction is that dynamic memory is not allowed
  - Can be both easier and more flexible to write algorithms without having to explicitly deal with time: pipeline stages can change based on requirements.
  - Can be easier to debug: the C++ code can be compiled and run to check for correctness much more quickly than HDL can be simulated.

HLS

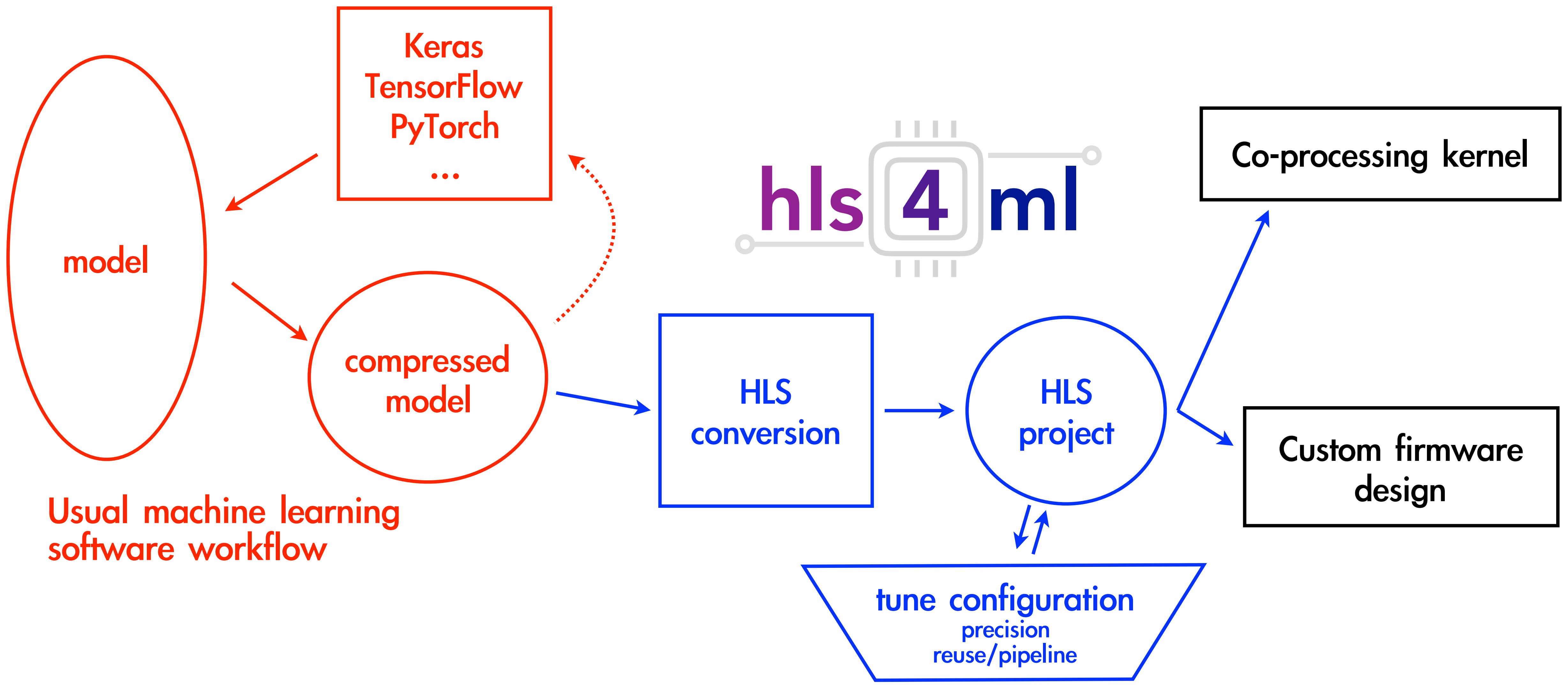
Converting NNs to HLS: hls4ml

- hls4ml is a compiler taking Keras, pytorch, or ONNX as input and usually producing HLS.
- The “backend” can be changed. Although non-HLS backends exist, hls4ml generally produces HLS for Vivado HLS, Intel HLS, or Catapult. Vitis HLS backend in development.
- Produces spatial dataflow code specific to the program at hand (not systolic array)

Keras

ONNX

PyTorch



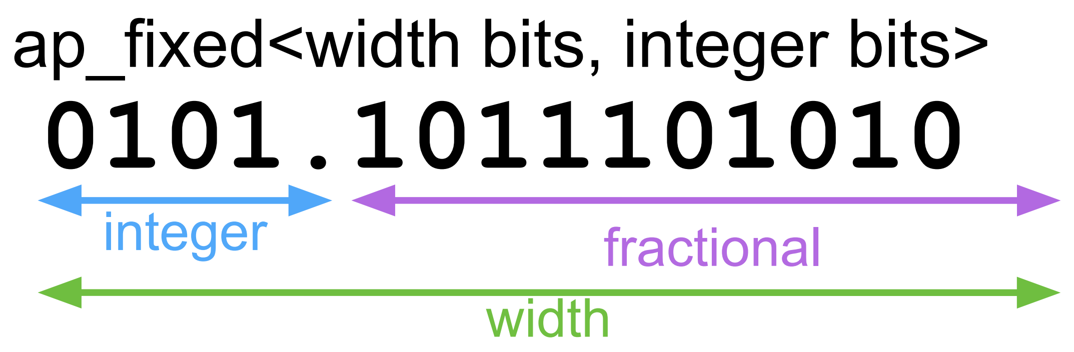
Vivado™ HLS

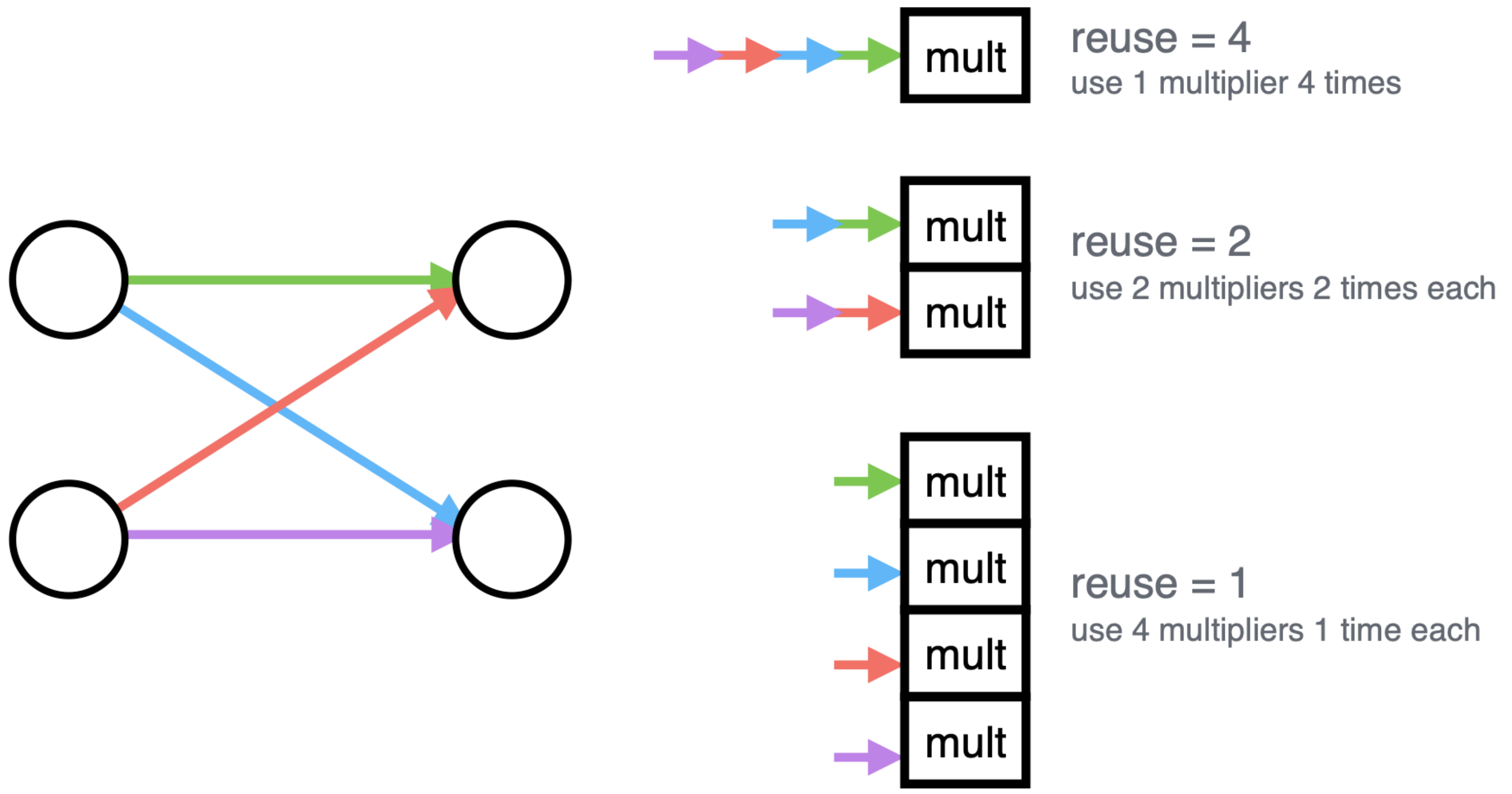
HLS COMPILER

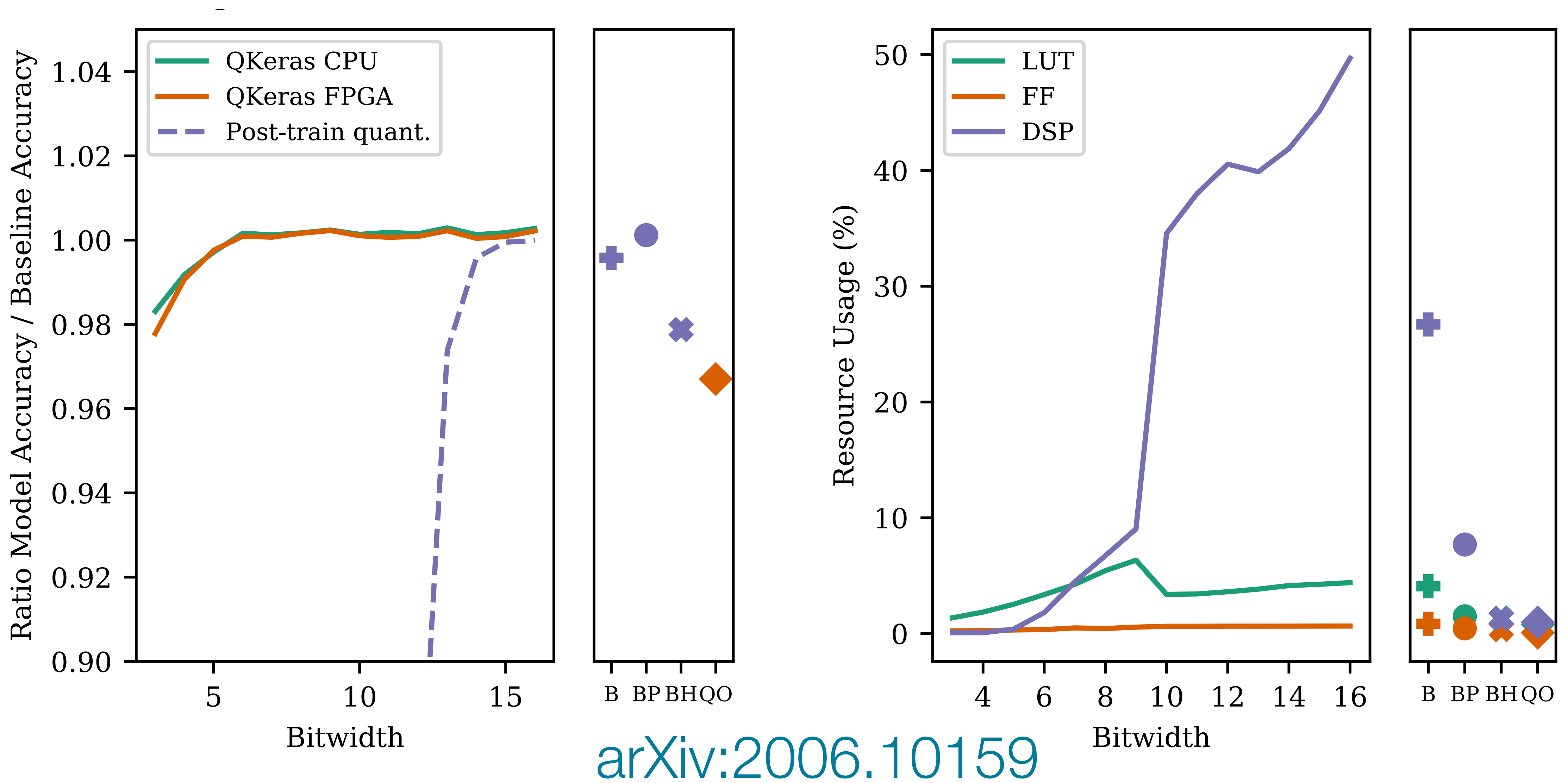
Catapult

Optimizing for FPGAs

- Fixed-point arithmetic is preferred for efficiency.
- Quantization-aware training (QKeras, Brevitas) performs better than post-training quantization.
- Also have a number of options in tweak the implementation, including “reuse factor”







arXiv:2006.10159

Types of layers supported

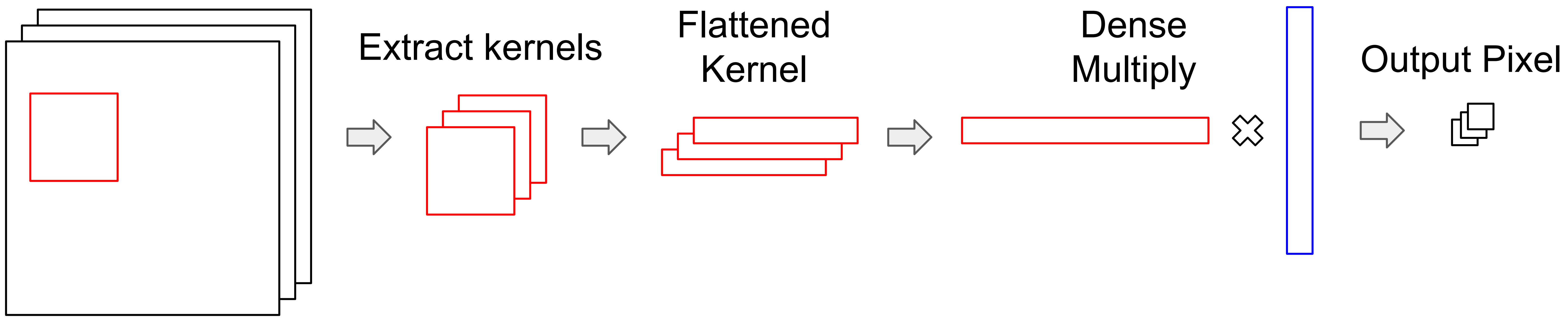
- MLP: Dense matrix/vector multiplies map well into FPGA calculations
  - Some support for sparse matrices, more in development
- 1D and 2D CNNs
- Batch Normalization
- Max/AveragePooling
- Various activations
- GRU, LSTM, and Simple RNN
- Embedding
- Special support for binary and ternary networks

CNN developments: streaming

- Parallel CNN implementations quickly run into limitations for large CNNs
- Streaming implementations support large CNNs.
  - Instead of getting input in parallel, inputs are sent one data point at a time.
    - use hls::stream (Vivado) or ihc::stream (Intel) of an array of channels associated with a data point.
    - A streaming implementation using ac_channels is being developed for Catapult
  - FIFOs are used between the layers
    - Can allow for more flexible network structure
- Also introduced the option to store weights externally for large models

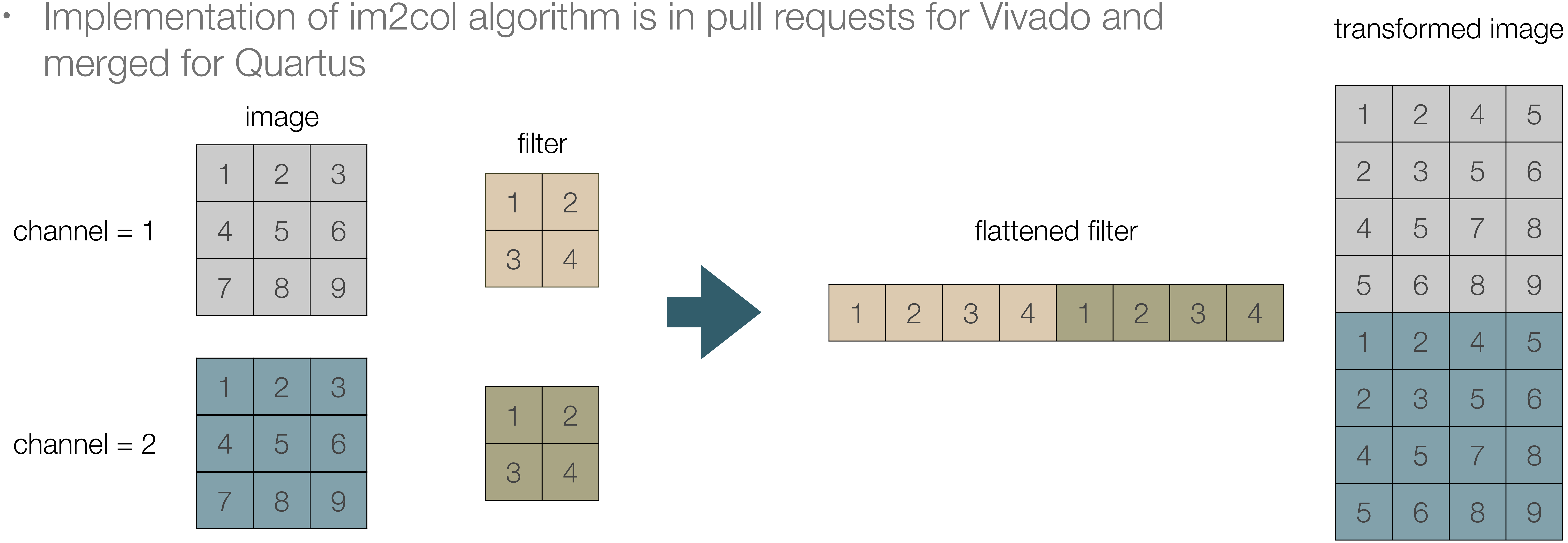
CNN developments

- We have two streaming CNN implementations for the Vivado backend: line buffer (default) and encoded
  - A streaming CNN implementation is in a pull request for the Quartus backend.
- A tutorial with CNNs is available in the hls4ml-tutorial.



Parallel CNNs

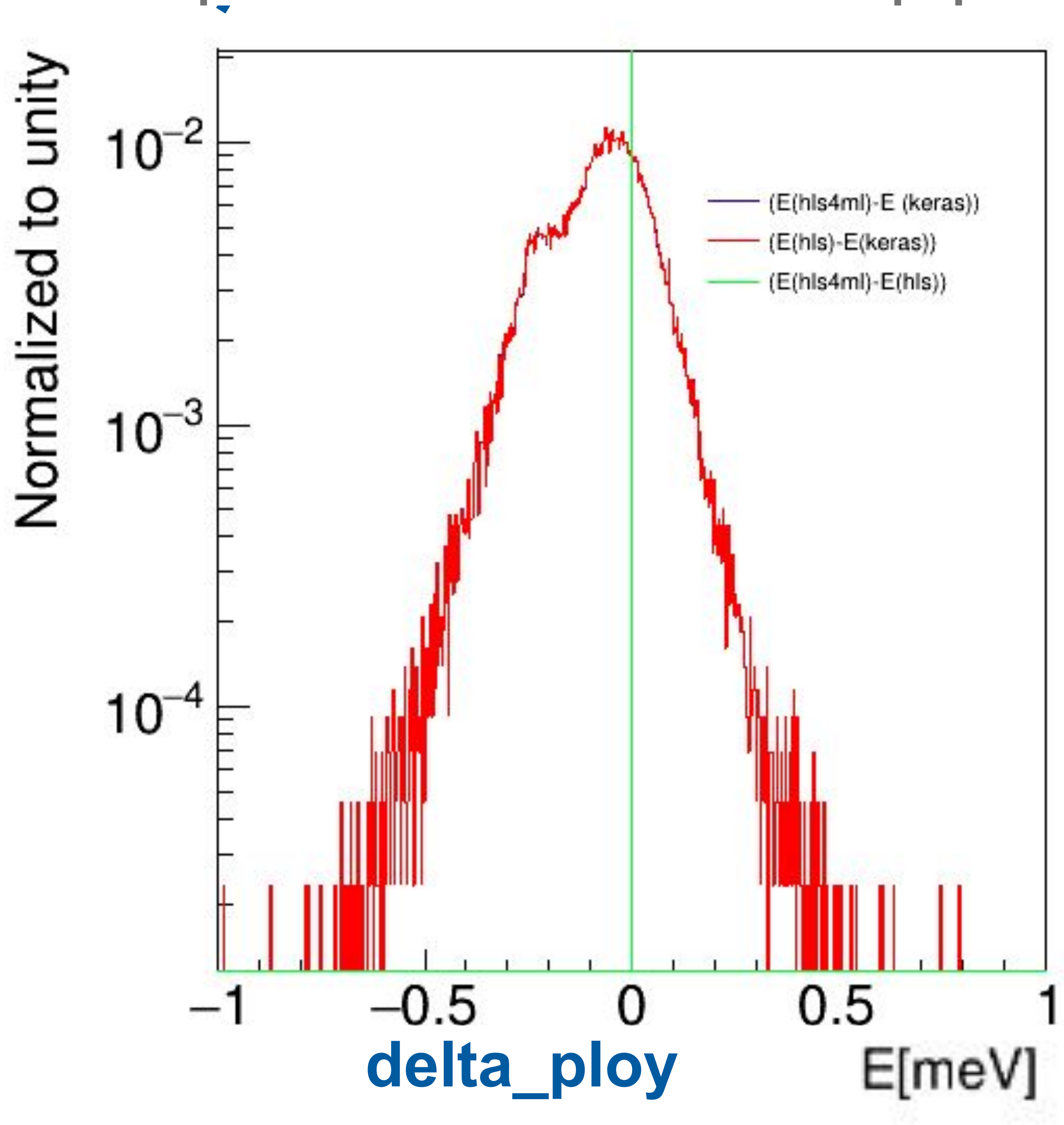
- Parallel CNNs remain useful for smaller networks.
  - Implementation of im2col algorithm is in pull requests for Vivado and merged for Quartus



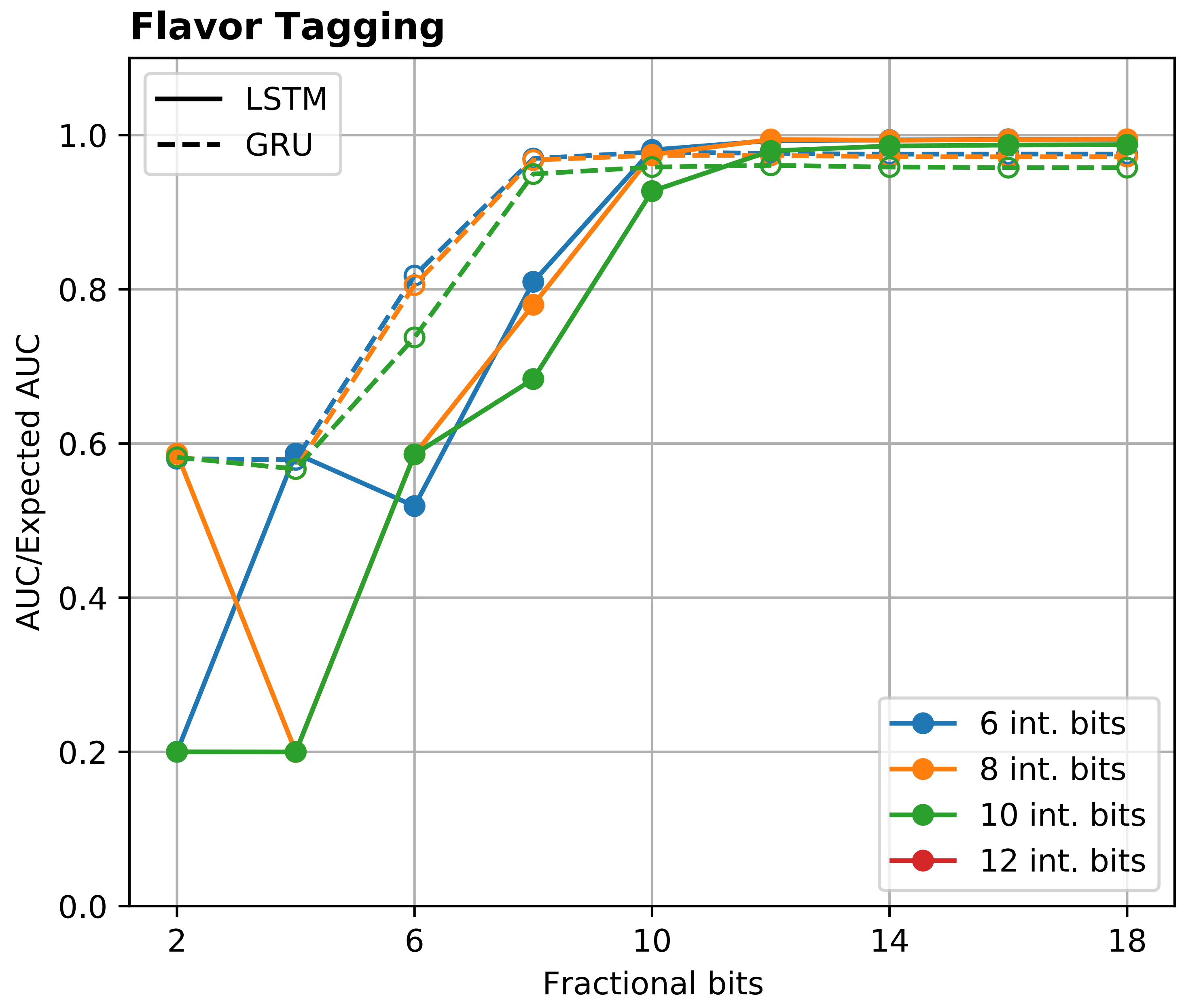
  - Implemented Winograd's minimal filtering algorithm for special cases (arXiv:1509.09308 [cs.NE])

Recurrent NNs

- Two RNN implementations were made independently, one for the Quartus backend (10.1007/s41781-021-00066-y), one for Vivado (arXiv:2207.00559)
- The implementations have been made uniform in style and merged. LSTM, GRU, and simple RNN are supported.



Quartus version is for ATLAS calorimeter readout

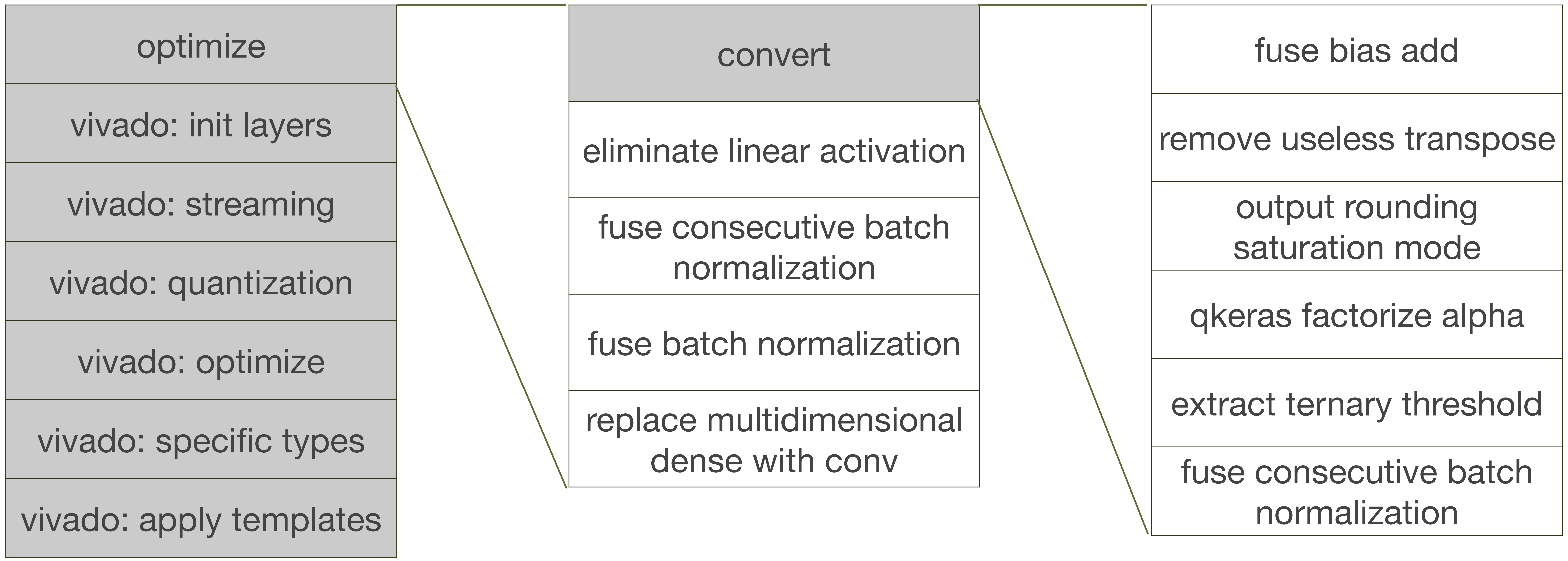


Vivado b-tagging example

Internal hls4ml evolution

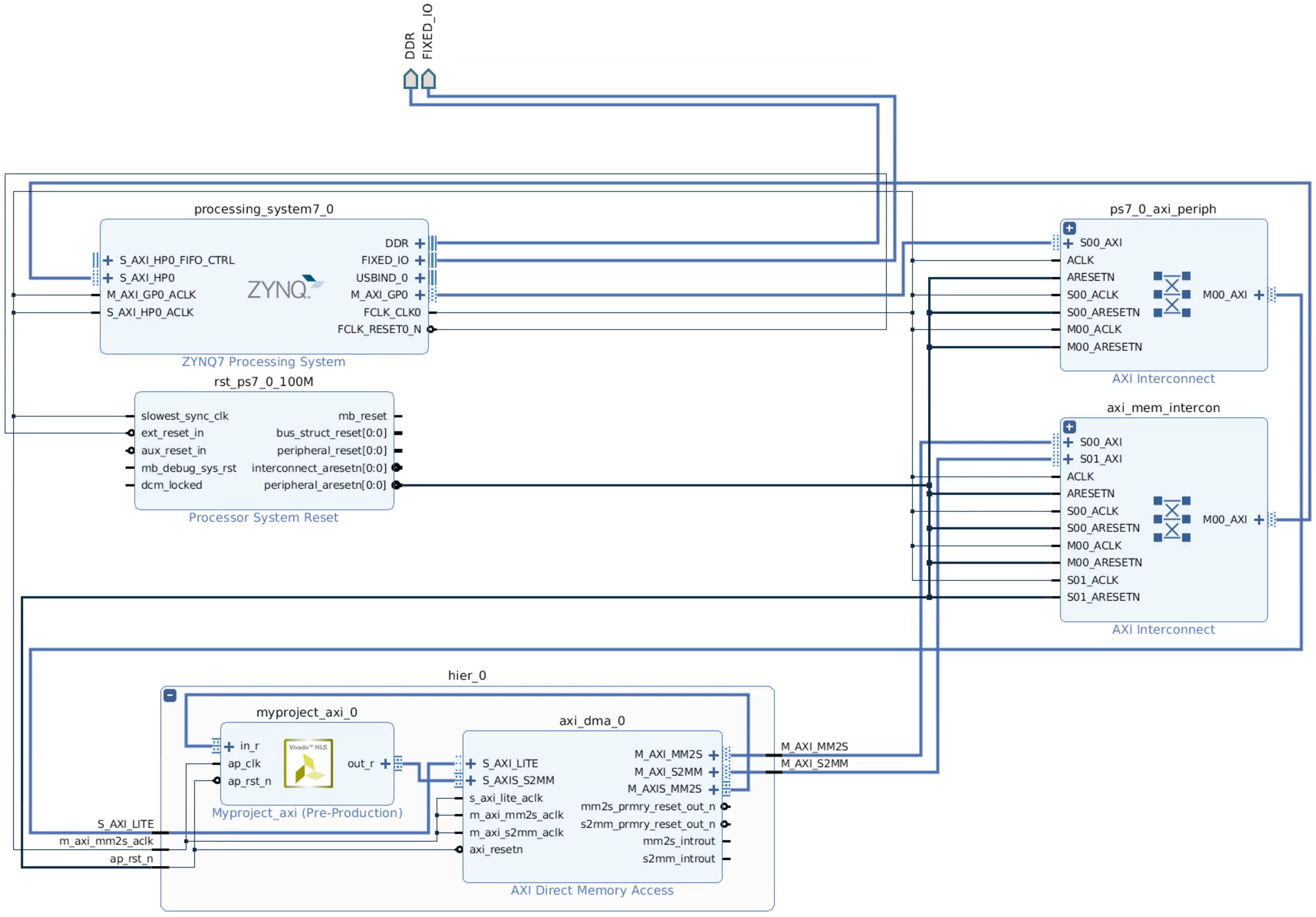
- In order to better support different backends, and also to better support optimizations, hls4ml's internal representation and processing were overhauled
  - Processing consists of flows of optimizers
  - Backend-specific optimizers produce the code

Vivado IP flow



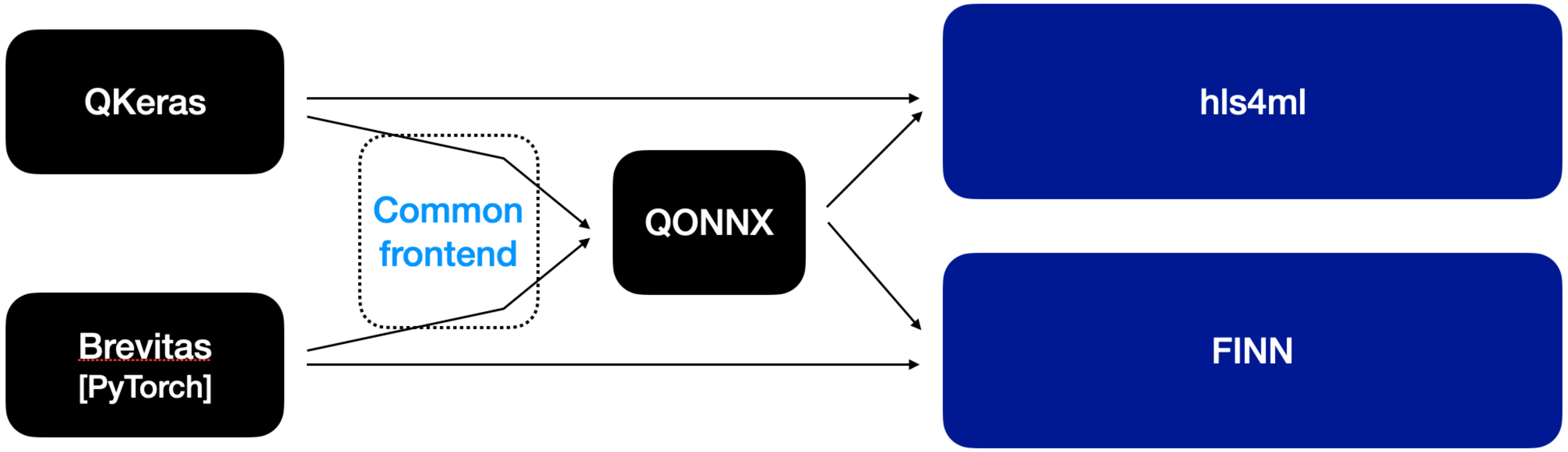
VivadoAccelerator backend

- A Block Design is created containing the NN IP, as well as the other necessary IPs to create a complete system.
- More information is available in the hls4ml-tutorial.
- Work is being done towards supporting Alveo cards.



Collaboration with FINN group

- AMD/Xilinx's FINN project has similar goals, with emphasis on smaller bit widths.
- We recently started cooperating, with the first step being a common frontend.
  - Brevitas (PyTorch) and QKeras can export QONNX, with HAWQ export in development: then hls4m and FINN can import QONNX
  - The frontend has common cleaning and QONNX manipulation utilities
- We have a QONNX model zoo for example models



QONNX arXiv:2206.07527 [cs.LG]

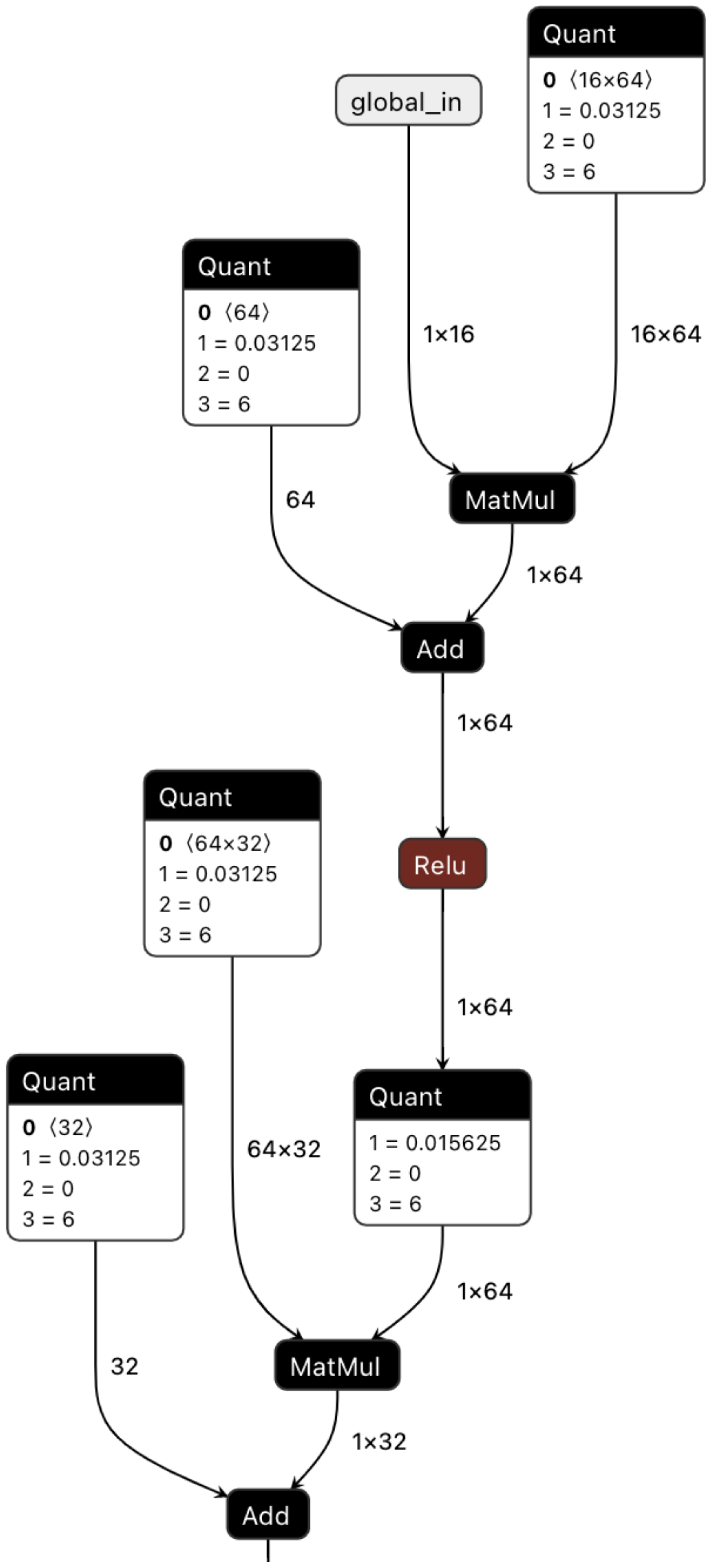
- QONNX is a simple but flexible method to represent uniform quantization
  - lightweight: only 3 operators (Quant, BipolarQuant, Trunc)
  - abstract: not tied to any implementation

- Fused quantize-dequantize (QDQ) format

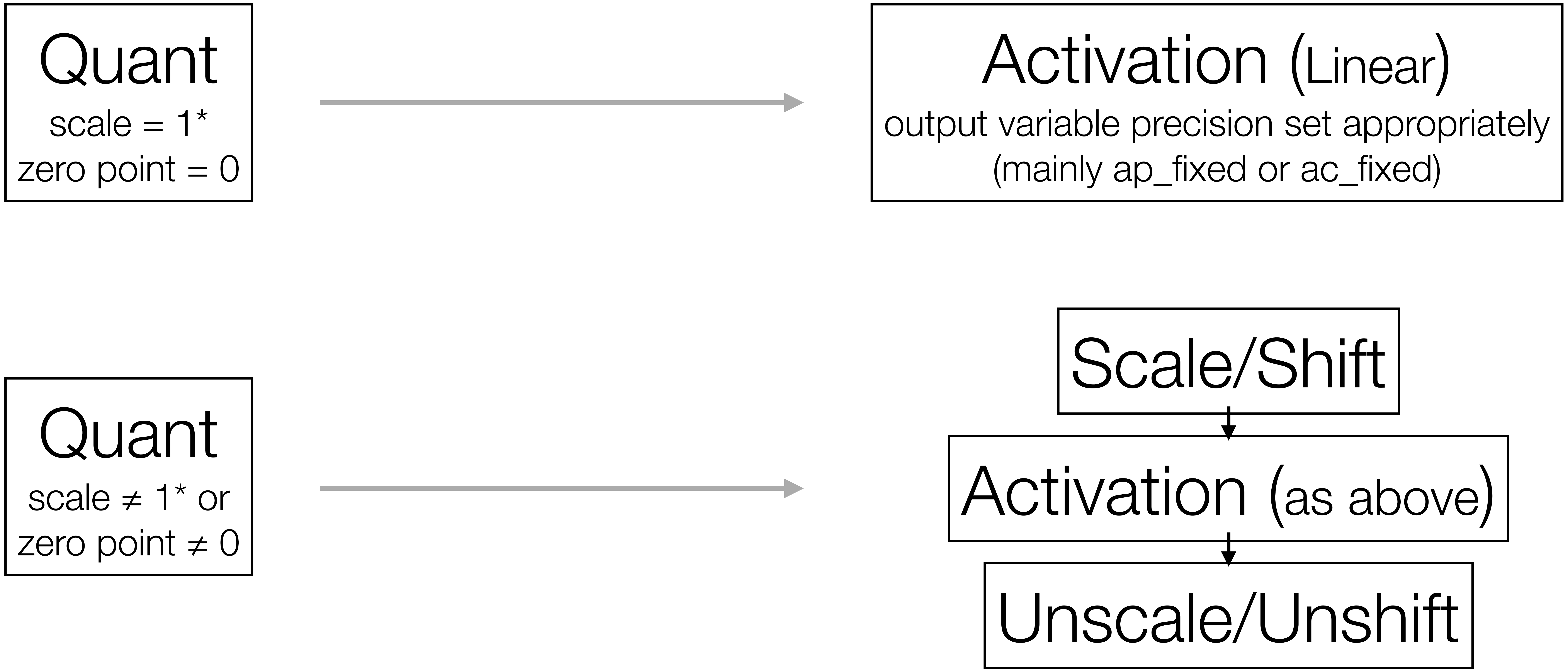
$$\text{quantize}(x) = \text{clamp}\left(\text{round}\left(\frac{x}{s} + z\right), y_{\min}, y_{\max}\right)$$

$$\text{dequantize}(y) = s(y - z)$$

where s is scale and z is zero offset.



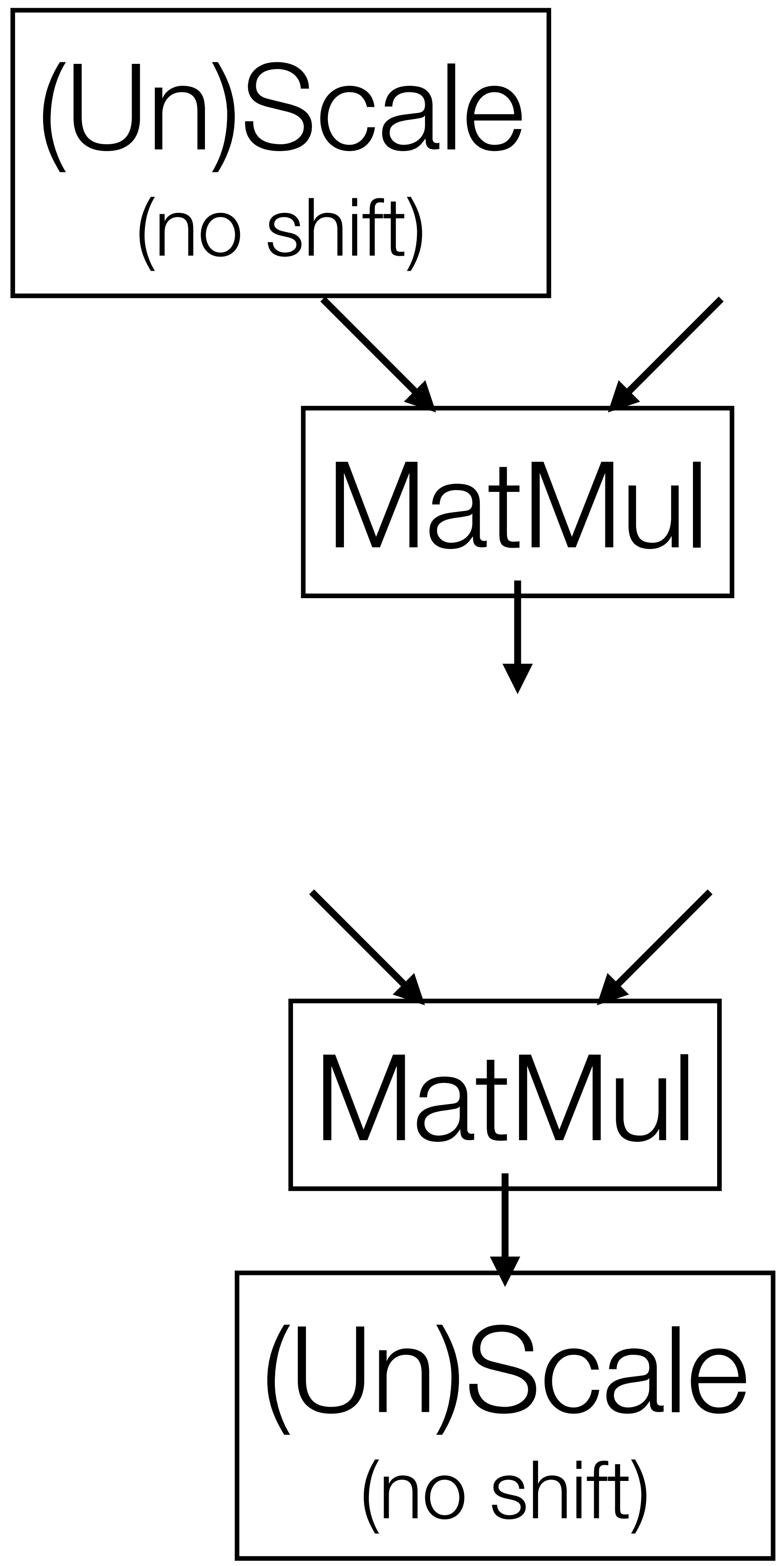
Logical Quant Node Handling



*as an optimization, powers of 2 can be handled the same as when scale = 1

Propagating scales

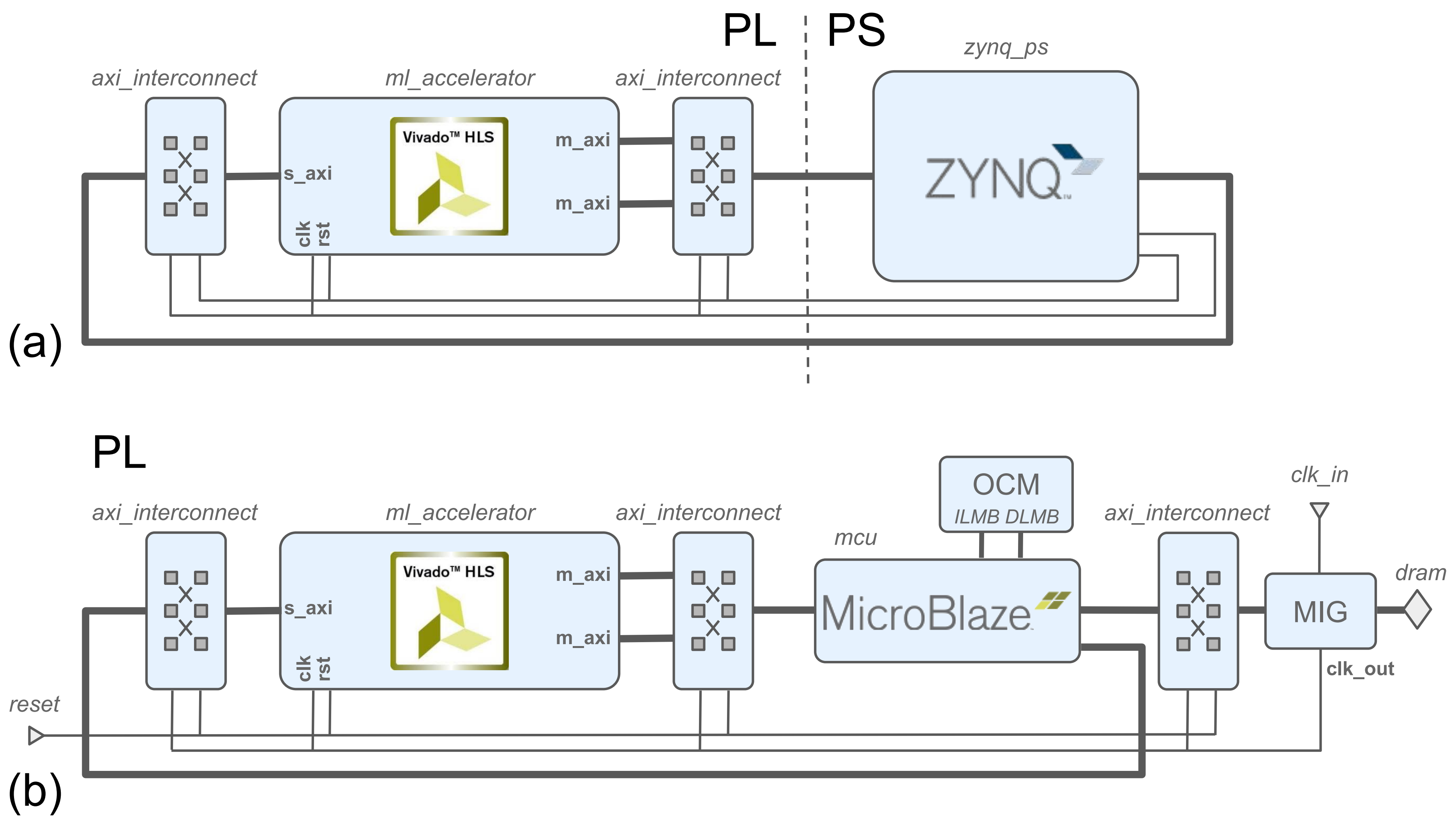
- QDQ is not meant to be implemented directly
- Can propagate scales/shifts and across linear operators if certain conditions are met
- Often make use of the power of 2 optimization to offload the scale propagation to the HLS compiler.



TinyML arXiv:2206.11791 [cs.LG]

- One of the advantages of FPGAs is low power vs performance
- Together with the FINN group we competed in MLPerf Tiny Inference Benchmark v0.7 open division
  - hls4ml was used for image classification (IC) and anomaly detection (AD)
  - Used a SoC (ZYNQ) and an FPGA-only design (Arty)

Benchmark	Flow	Prec. [bits]	Params.	Accuracy
IC	hls4ml	8–12	58 115	83.5%
IC	FINN	1	1 542 848	84.5%
AD	hls4ml	6–12	22 285	0.83 AUC
KWS	FINN	3	259 584	82.5%



TinyML

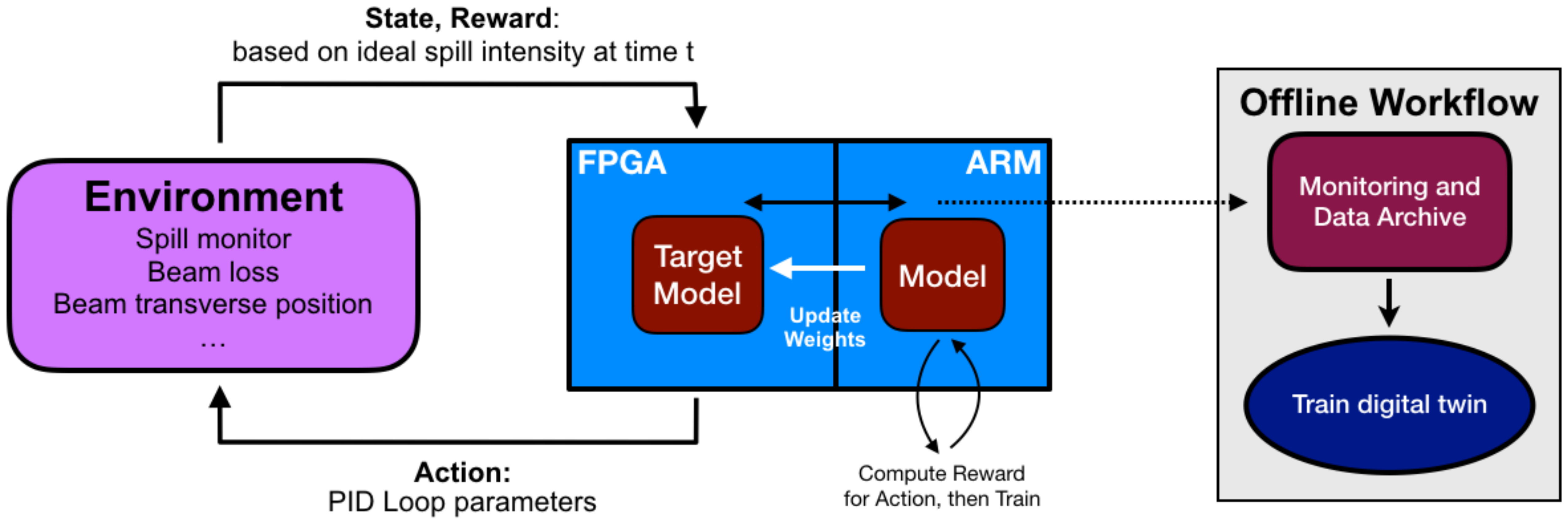
- Developing the models for the competition discovered useful optimizations:
  - Buffer depth optimization: FIFOs are used between the layers in streaming implementations. One can reduce resources by tuning the size.
  - Dense + ReLU merging: can avoid FIFO altogether in this common case

	BRAM [18 kb]		FF		LUT	
Available	280		106 400		53 200	
Without opt.	477	170.4%	79 177	74.4%	66 838	125.6%
With FIFO opt.	278	99.3%	72 686	68.3%	58 515	110.0%
With ReLU opt.	345	123.2%	72 921	68.5%	55 292	103.9%
With all opt.	146	52.1%	66 430	62.4%	46 969	88.3%

  - Quantized Dense + BatchNormalization merging: new layer avoids FIFO. (New layer also added to QKeras.)
- There are pull requests to the main branch of hls4ml from these developments

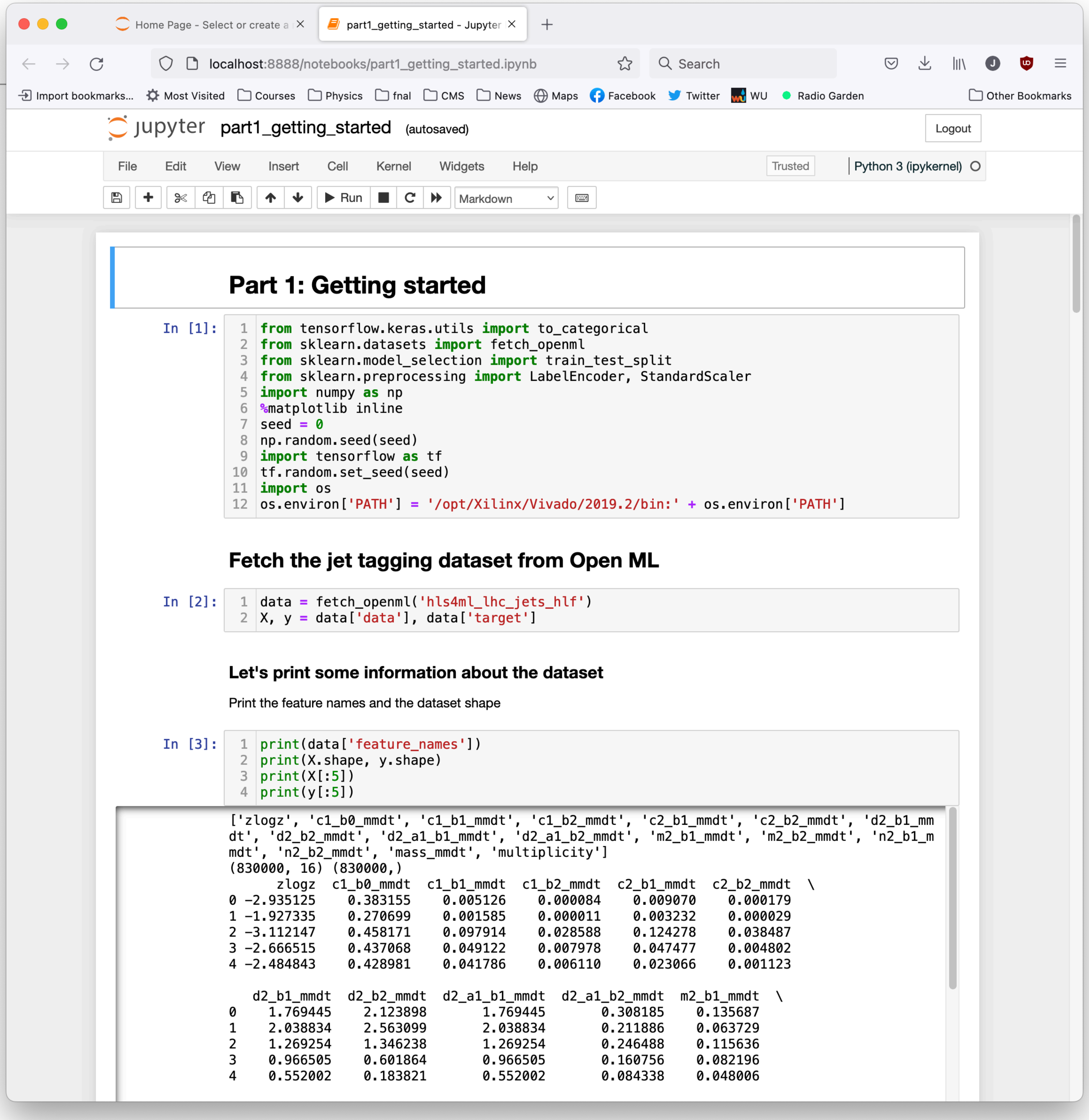
ML methods on the edge for accelerators

- Study using reinforcement learning to regulate the gradient magnet power supply of the Fermilab Booster (arXiv:2011.07371)
- Improve beam performance for the Mu2e experiment by integrating ML into accelerator operations (arXiv:2103.03928)
- Employing Intel Arria 10 SoC systems with distributed controls, in cooperation with Crossfield Technology LLC.



For more information

- Main repository: https://github.com/fastmachinelearning/hls4ml
- Good starting point for those interested: https://github.com/fastmachinelearning/hls4ml-tutorial
- Documentation: https://fastmachinelearning.org/hls4ml/
- Help available at https://github.com/fastmachinelearning/hls4ml/discussions
- Open-source project, so welcome to contribute



Backup

Quant and BipolarQuant nodes

Supported

`Quant`: calculate the quantized values of one input tensor and produces one output data tensor.

Attributes:

- `signed` (boolean): defines whether the target quantization interval is signed or not.
- `narrow` (boolean): defines whether the target quantization interval should be narrowed by 1. For example, at 8 bits if `signed` is true and `narrow` is false, the target is $[-128, 127]$ while if `narrow` is true, the target is $[-127, 127]$.
- `rounding_mode` (string): defines how rounding should be computed during quantization. Currently available modes are: `ROUND`, `ROUND_TO_ZERO`, `CEIL`, `FLOOR`, with `ROUND` implying a round-to-even operation.

Inputs:

- `x` (float32): input tensor to be quantized.
- `scale` (float32): positive scale factor with which to compute the quantization. The shape is required to broadcast with `x`.
- `zero_point` (float32): zero-point value with which to compute the quantization. The shape is required to broadcast with `x`.
- `bit_width` (int, float32): the bit width for quantization, which is restricted to be ≥ 2. The shape is required to broadcast with `x`.

Outputs:

- `y` (float32): quantized then dequantized output tensor

Not yet supported

`BipolarQuant`: calculate the binary quantized values of one input tensor and produces one output data tensor.

Attributes: None

Inputs:

- `x` (float32): input tensor to be quantized.
- `scale` (float32): positive scale factor with which to compute the quantization. The shape is required to broadcast with `x`.

Outputs:

- `y` (float32): quantized then dequantized output tensor

Trunc nodes

Not yet supported

`Trunc`: truncate the least significant bits (LSBs) of a quantized value, with the input's `scale` and `zero_point` preserved.

Attributes:

- `rounding_mode` (string): defines how rounding should be computed during truncation. Currently available modes are: `ROUND`, `CEIL`, and `FLOOR`, with `FLOOR` being the default.

Inputs:

- `x` (float32): input tensor to quantize.
- `scale` (float32): positive scale factor with which to compute the quantization. The shape is required to be broadcast with `x`.
- `zero_point` (float32): zero-point value with which to compute the quantization. The shape is required to be broadcast with `x`.
- `in_bit_width` (int, float32): bit-width of the input, which is restricted to be ≥ 2. The shape is required to broadcast with `x`.
- `out_bit_width` (int, float32): bit width of the output, which is restricted to be ≥ 2. The shape is required to broadcast with `x`.

Outputs:

- `y` (float32): dequantized output tensor.