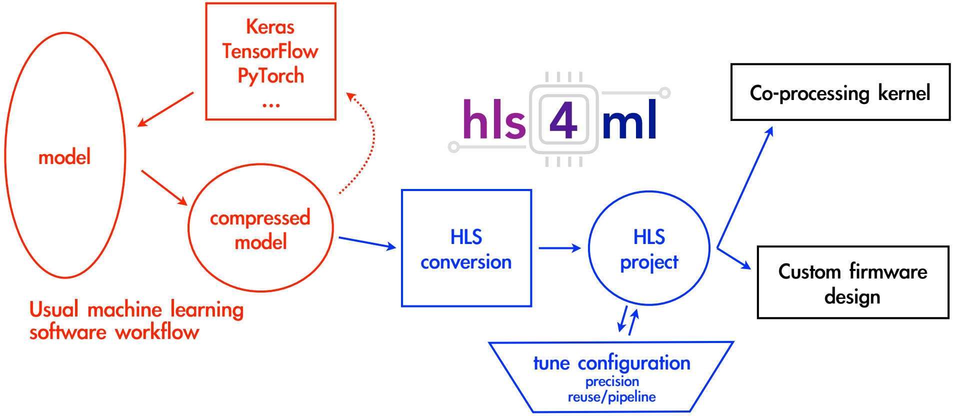
Summary of hls4ml release 1.0

Jovan Mitrevski for the hls4ml team
CSAID AI Jamboree
January 30, 2025

hls4ml introduction

- hls4ml is a compiler, converting Keras, PyTorch, or ONNX to HLS
- The “backend” can be changed. Although non-HLS backends exist, hls4ml generally produces HLS for Vitis HLS, oneAPI, or Catapult.
  - (Vivado HLS and Intel HLS also supported, but no longer a focus of development)
- Produces spatial dataflow code specific to the program at hand (not systolic array)

Keras
ONNX
PyTorch



AMD Vitis
Catapult
oneAPI

Reminder: two styles of sending data between layers

- io_parallel: all data from one event is transferred in parallel between the layers
  - good for smaller models without skip connections
- io_stream: data is transferred one pixel at a time (sending all channels in parallel)
  - generally used for CNN models. (For 1D MLPs, all inputs are still sent in parallel)
  - FIFOs are used between the layers
  - useful for larger models and for skip connections

Updated backend, Vitis HLS

- Although experimental Vitis HLS existed before, its performance was not up to Vivado HLS level.
- Starting with Vitis HLS 2022.2, the performance issues have been fixed
- We now support Vitis HLS 2022.2 or later (focusing on 2023.2 or later)
- Fully supported is the Vivado IP flow; the accelerator flow exists in a pull request
- The FIFO depth optimization algorithm that we supported for Vitis HLS exists in a pull request, not in the release
  - As an aside, Vitis provides more tools for setting the FIFO sizes

New backend: oneAPI

- The HLS code from the former “Quartus”, really Intel HLS, backend, served as the basis for a new oneAPI backend.
  - oneAPI release 2025.0 is the supported version (2024.x versions may work)
  - (Due to the Altera spinoff, 2025.0 will be the version to use for a while. OneAPI FPGA compiler responsibility will transition from Intel to Altera.)
- The HLS (IP) flow is currently the only flow supported. In the future, we would like to support the accelerator flow, too.
- oneAPI is SYCL-based, so the HLS is treated as a SYCL kernel
  - The “host code” becomes the testbench

oneAPI backend differences (vs Quartus—Intel HLS)

- The software had to change because of the SYCL interface
  - Explicit pipe is used for input and output for both io_stream **and io_parallel**
  - Each pipe is synthesized to a conduit with it’s own handshaking (not just the component’s)
- There is a preference for std::array over C-style arrays per suggestion by Intel.
- Nevertheless, io_parallel should produce roughly the same results as it would have had with Intel HLS.
- io_stream now explicitly implements the “dataflow” style we use in the Vitis and Vivado backends.
  - Before, we never used “dataflow” with Intel.

io_stream “dataflow” style

```
void Myproject::operator()() const {

    // hls-fpga-machine-learning declare task sequences
    task_sequence<nnet::dense_resource_stream<Fc1InputPipe, Layer2OutPipe, config2>> fc1;
    task_sequence<nnet::relu_stream<Layer2OutPipe, Layer4OutPipe, relu_config4>> relu1;
    task_sequence<nnet::dense_resource_stream<Layer4OutPipe, Layer5OutPipe, config5>> fc2;
    task_sequence<nnet::relu_stream<Layer5OutPipe, Layer7OutPipe, relu_config7>> relu2;
    task_sequence<nnet::dense_resource_stream<Layer7OutPipe, Layer8OutPipe, config8>> fc3;
    task_sequence<nnet::relu_stream<Layer8OutPipe, Layer10OutPipe, relu_config10>> relu3;
    task_sequence<nnet::dense_resource_stream<Layer10OutPipe, Layer11OutPipe, config11>> output;
    task_sequence<nnet::softmax_stream<Layer11OutPipe, Layer13OutPipe, softmax_config13>> softmax;

    // hls-fpga-machine-learning insert layers

    fc1.async(w2, b2);
    relu1.async();
    fc2.async(w5, b5);
    relu2.async();
    fc3.async(w8, b8);
    relu3.async();
    output.async(w11, b11);
    softmax.async();

}
```

New backend: Catapult

- Siemens EDA Catapult is widely used for ASIC design, and also FPGAs from various companies (include eFPGAs)
- A big effort was made (generally by Siemens employees) to port the Vivado HLS code to Catapult, with our support
- Functionality is generally on par with Vivado and Vitis HLS
- Note: one needs to use the hls4ml packaged with Catapult—currently the Catapult code packaged with hls4ml is missing the pragmas
    - Changes may come to this

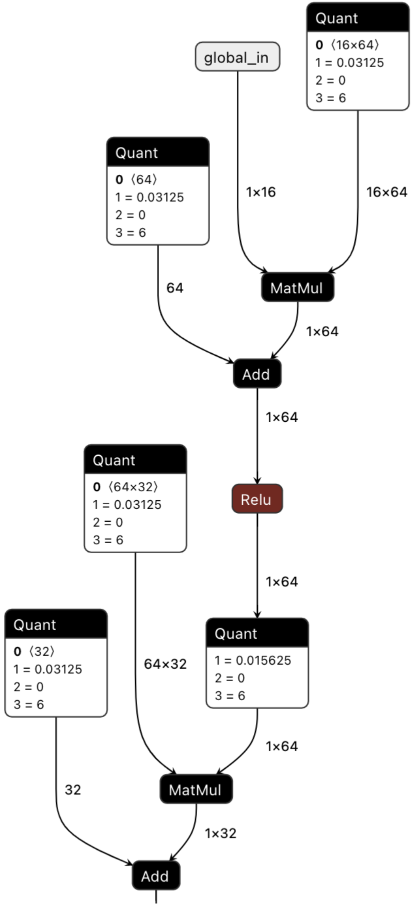
QONNX parsing

- One of the products of the cooperation with the FINN group has been proposing a simple but flexible method to represent uniform quantization in ONNX: QONNX
  - lightweight: only 3 operators (Quant, BipolarQuant, Trunc)
  - abstract: not tied to any implementation
- Fused quantize-dequantize (QDQ) format

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

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

where s is scale and z is zero offset.



qonnx package

- As part of our collaboration with the FINN group, we made a package of common utilities, https://github.com/fastmachinelearning/qonnx
  - constant folding, shape inference, etc.—“cleaning”
    - when parsing ONNX can assume we know the shape
  - channels-first to channels-last conversion for CNNs
    - do not need to support channels-first in hls4ml for ONNX
  - Other common utilities, like Gemm to MatMul and Add
    - do not need to support Gemm explicitly in hls4m
  - Evaluate QONNX graphs for functional analysis

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.

Improved direct PyTorch parsing

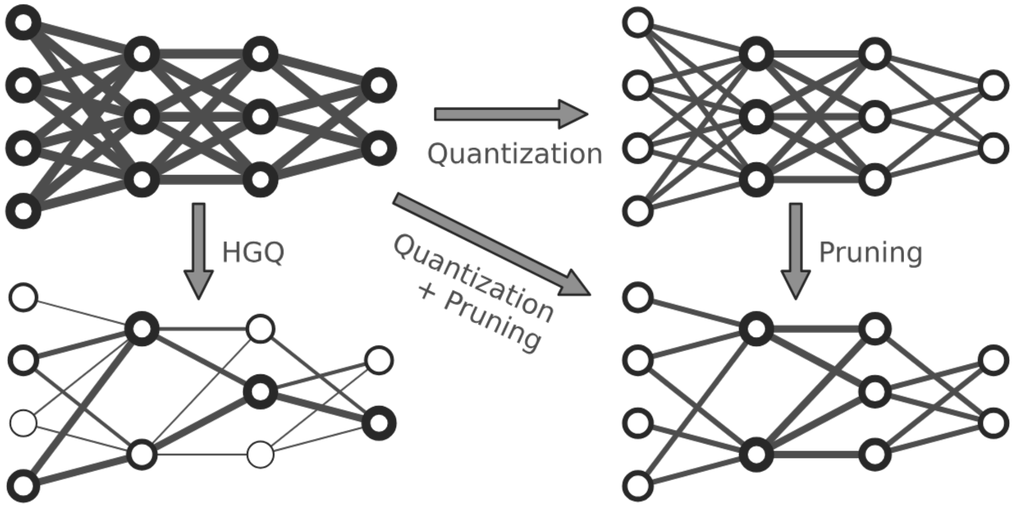
- PyTorch has become more popular than Keras
- hls4ml, however, has traditionally focused more on Keras
- PyTorch, including Brevitas, is supported via (Q)ONNX
- Significantly improved direct PyTorch parsing, too.
- As for QONNX, we use a channels-first to channels-last conversion step for CNNs
  - (A few additional improvements have been merged after rel 1.0)

Automatic precision inference

- The default behavior of `config_from_(keras|pytoch|onnx)_model` has changed so that in “name” granularity, the precision set for all the values is “auto”.
  - The default precision is only used as a backup when no better precision can be chosen
- With an “auto” precision, the accumulator size is set to never overflow or truncate, only using the input and weight precisions.
  - The weight values are not used, just the weight types
- Warning: precisions can get quite wide when using post-training quantization.
- One can set max_precision to limit the precision width, but generally it may be better to explicitly set some precisions in the configuration

Other improvements

- [High Granularity Quantization (HGQ)](): per-weight or per-activation quantization



- [Hardware-aware Optimization API](): hardware-aware pruning and weight sharing to reduce model footprint and computational requirements

Looking forward

- We are updating the testing environment especially for synthesis
  - Synthesis environment uses deprecated command-line interface, and does not exercise all the backends
- Keras v3 support will be added; some question on QKeras progress
- We have a pull request to be able to write out the hls4ml internal representation
- Vitis backend structure will be updated to not inherit from Vivado
- Intel/Altera engineers have recommended some oneAPI improvements, and currently tracing and profiling is not supported for oneAPI
- Vitis Accelerator and oneAPI Accelerator backends are planned (with completion depending on availability of effort)
- We want to make releases more frequent

Backups

(Q)ONNX ingestion

- Quant nodes are applied both on the data flow and on weights.
- Introduce explicit Constant nodes for the weights.
  - This more easily handles Quant nodes between constant values (initializers) and operations
- Make extensive use of optimizers to convert graph to a synthesizable code
  - ONNX nodes are converted to hls4ml nodes that closely match the ONNX nodes.
  - Optimizers convert to standard Dense, Conv2D, etc