Quantization and Deployment of Neural Networks on FPGAs

Chang Sun

Caltech, PMA

Quantization on FPGAs

Quantization is a general concept, but we consider fixed-point quantization only here

- Floating point multiplication is **prohibitively expensive** on FPGAs
  - It **may** be possible with minifloat with no normalization for accumulation, etc, but beyond the scope of this presentation.
- Fixed-point multiplication is **cheap** on FPGAs
  - Single DSP48E1/2 on Xilinx FPGAs can perform up to 18x25/27 bits multiplication with fused accumulation.
  - Constant multiplication can be performed with shift-add structures using LUTs and fast carriers

Fixed Point Numbers

Fixed point numbers are integer with some powers-of-two factor

Any fixed point number has three attributes:

1. **Integer bits**, `i`
2. **Fractional bits**, `f`
3. **sign bit** (0/1), `keep_negative, k`

Width/Total bits is the sum of these three.

Example:

5.28125 = `0101.01001` `(k=1, i=3, f=5)`

The representable range is $[-k \cdot 2^{i-1}, 2^{i-1} - 2^{-f}]$, with step 2^{-f}.

Note that this is true, even if both **number integer and fractional bits are negative**, as long as the total number of bits ≥ 1:

```
0.0859375 = 0.0001011 : (k=0, i=-3, f=7)
     -24. = 111000.   : (k=1, i=5, f=-3)
```

(Using two's complement representation here)

Quantization of Neural Networks

What needs to be quantized, and how much?

Forget about neural networks for a moment. Consider the elementary operations:

- `ADD/SUB` between two fixed-point numbers
- `MUL` between two fixed-point numbers
- `SHIFT` arithmetic shift left/right (power-of-two mul/div)
- `MUX` between two fixed-point numbers condition on a bit
- `LOOKUP` into a table of number of width m by index with width n
- `DIV` between two fixed-point numbers
- bit-wise operations

Quantization of Neural Networks

What needs to be quantized, and how much?

Forget about neural networks for a moment. Consider the elementary operations:

- `ADD/SUB` between two fixed-point numbers ~ **max(m,n)**
- `MUL` between two fixed-point numbers ~ **m*n-max(m,n)**
- `SHIFT` arithmetic shift left/right (pow2 mul/div) ~ const shift - **0**; Barrel Shifter - **m*n**
- `MUX` between two fixed-point numbers condition on a bit ~ **max(m,n)**
- `LOOKUP` into a table of number of width m by index with width n ~ **<m*2^n**
- `DIV` between two fixed-point numbers ~ **m*n** but **slow**
- bit-wise operations ~ **max(m,n)**

The numbers **will vary a lot** depending on the impl/device, but holds OOM-ly in general.

What about neural networks?

- Most operations are `ADD/SUB` and `MUL`
  - Dense or anything dense-like (Conv, RNN)
  - Can be translated to `ADD/SUB + SHIFT` for constant multiplications
- `LOOKUP`
  - Nonlinear activations like `sigmoid` and `tanh`. **Not** `ReLU`
- `MUX`
  - Absolute value, max pooling
- `DIV`
  - Normalization layers, generally not used. Can be replaced with look-up of inverse values + multiplications.

Quantization aware training (QAT)

QAT is the essential method to obtain low-bitwidth quantized models with reasonable accuracy.

- Fake quantization during training
  - Straight-through estimator (STE), ($dq(x)/dx = 1$)
  - Noise injection ($q(x) = x + \epsilon$)
  - Various continuous relaxation
- Translate to fixed-point representation post training
  - There won't be real bit-exactness, but can be **almost** bit-exact with proper care.
  - almost: very rare ($< 0.01\%$ in general) mismatches with negligible value changes (1 step in MSB in most cases)

Quantization on the fly

During inference, we will frequently casting higher precision numbers to lower precision numbers, both in fixed-point representations. We will need to decide:

1. What to do when overflow happens?
2. How to round the value?

For 1, practically we have 2(3) choices:

- `WRAP` - drop bits beyond MSB, absolutely no overhead $\rightarrow$ for activations
- `SAT` (`SAT_SYM`) - saturation, clamps the values $\rightarrow$ for weights (no runtime quant.)
  - two substractions and two comparisons, high overhead

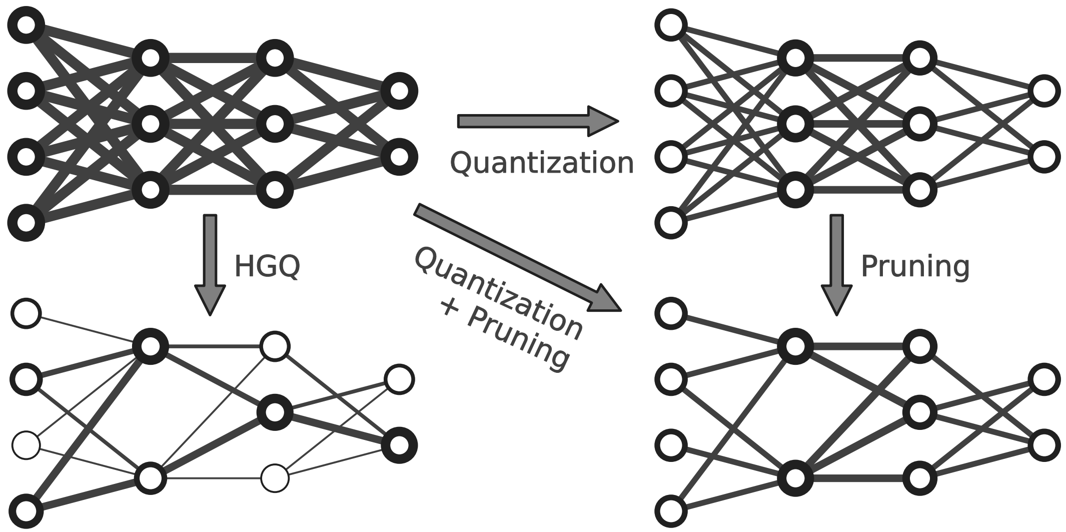
For 2, practically we have 2 choices:

- `TRN` - drops bits beyond LSB / floor, no overhead
- `RND` - rounding, ties up - same as adds 0.5 beyond the LSB before `TRN`

High Granularity Quantization

HGQ is a quantization-aware training framework on Keras 3

HGQ [code, doc, paper] quantizes neural network at **arbitrary fine** granularity. This includes 0 bits and automatic includes pruning as a special case.



In this presentation, we focus on the **per-weight** quantization use case as we require II=1 CMVM operations.

Key Features

- Bitwidths are learnt with gradient descent (pruning included as 0-bit quantization)
- Each weight and activation **may** have its own bitwidth
  - Massive resource compare saving to QKeras -- up to two OOM.
- Models are **almost bit-exact** up to the floating point representation limit
- LUT-aware architecture (`DenseT` and `ConvTXD` layers) support
- Supports most of the common layers
- Keras v3 based, supporting all `jax/tensorflow/torch` backends
- seamless `da4ml` and `hls4ml` integration

High Granularity Quantization

Differentiable quantization

We use a brief form of derivation here for brevity; see paper for detailed version.

Gradient from loss to `f` (drives up bitwidth):

$$\frac{\partial L}{\partial f} = \frac{\partial L}{\partial w} \cdot \frac{\partial w}{\partial f} = \frac{\partial L}{\partial w} \cdot \frac{\partial}{\partial f} 2^{-f} \left[x \cdot 2^{f} \right] = \frac{\partial L}{\partial w} \cdot \log 2 \cdot \left(x - 2^{-f} \left[x \cdot 2^{f} \right] \right)$$

Gradient from resource usage to `f` (drives down bitwidth)

$$\mathcal{L} = \mathcal{L}_0 + \beta \cdot \mathrm{EBOPs} + \gamma \cdot \sum \text{bit-widths}$$

Effective Bit Operations

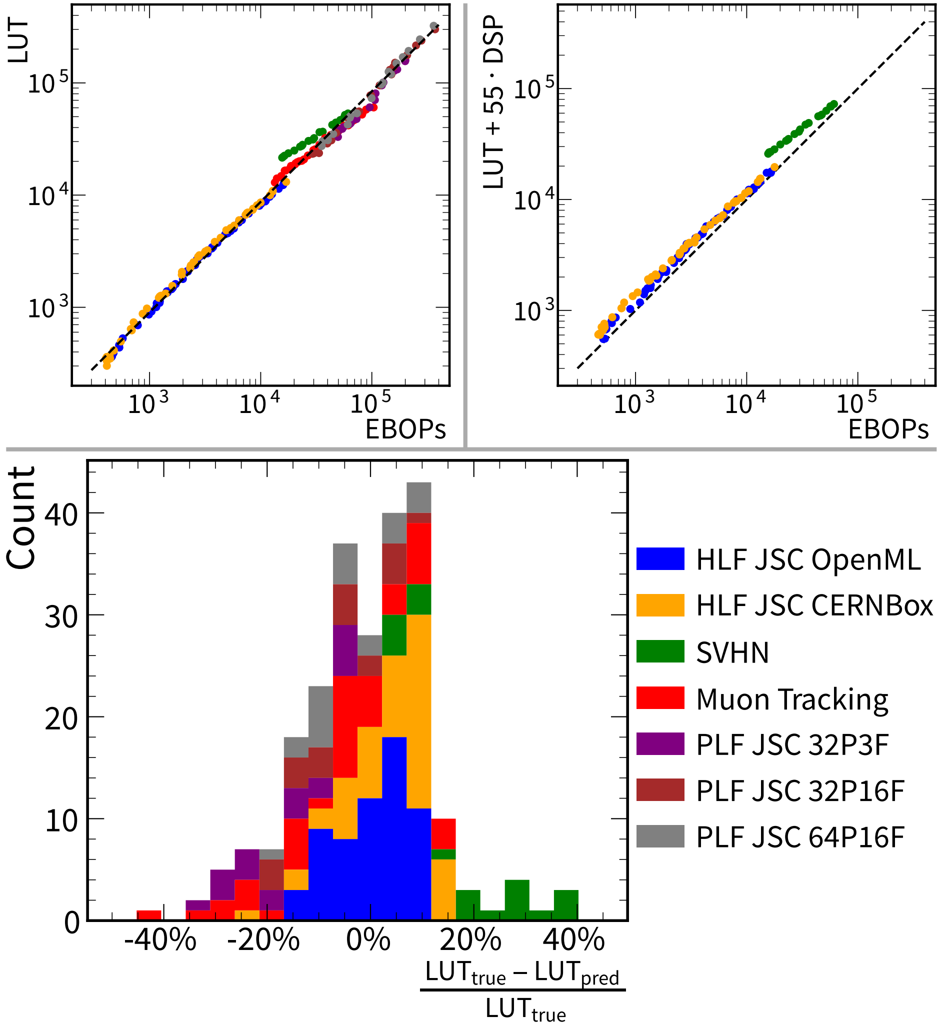
$$\mathrm{EBOPs} = \sum_{\times} \mathrm{bw}_i \cdot \mathrm{bw}_j + \sum_{+} \max(\mathrm{bw}_i, \mathrm{bw}_j)$$

This includes all multiplication and accumulation operations in the network, where bw_i and bw_j are the bitwidths of the two operands.

We also add EBOPs for a rough estimation of the table size for look-ups into the calculation.

Empirical formulas for Xilinx UltraScale+ FPGAs:

- For `hls4ml`, $\mathrm{EBOPs} \approx \mathrm{LUT} + 55 \cdot \mathrm{DSP}$
- For `da4ml`, $\mathrm{LUTs} \approx \exp(0.985 \cdot \log(\mathrm{EBOPs}))$

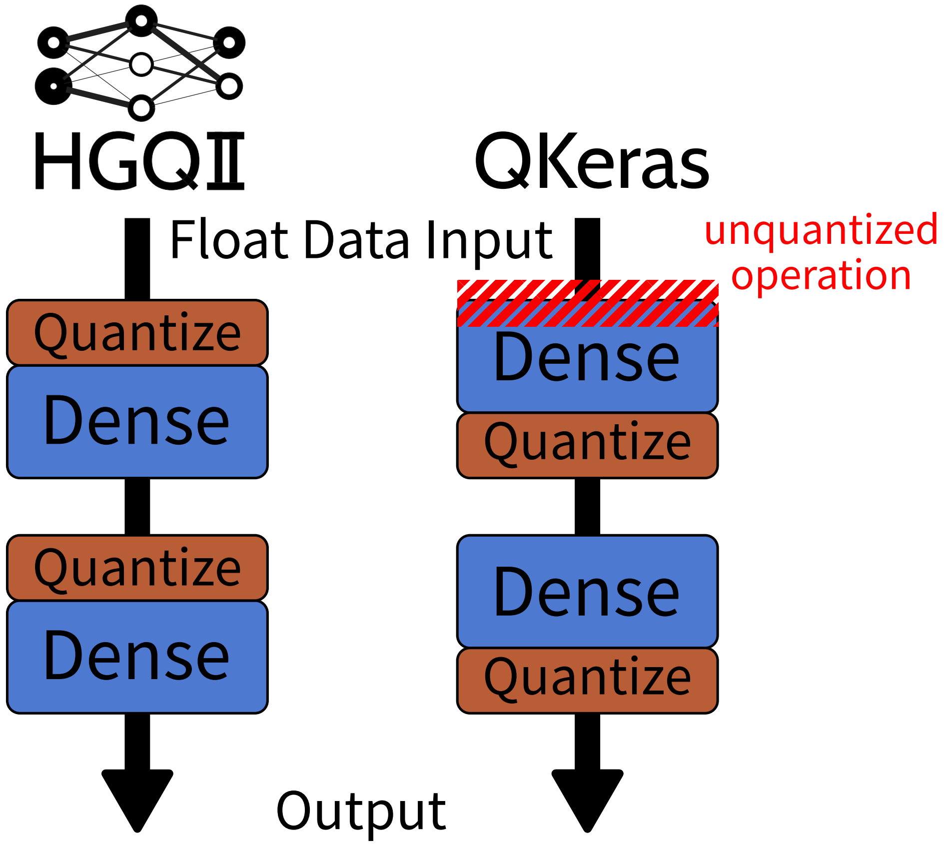


Quantization in HGQ2

HGQ2 is designed to be flexible and provide a set of APIs like the native `Keras v3`. All of the following can be overridden, but **by default**:

- Quantization is applied at the **input** of `Q-` layers, instead of the outputs like `QKeras`.

This ensures all values engaged in non-trivial operations are quantized.



Output quantizer can also be switched on if needed

Do add an input quantizer if you are still using QKeras

Quantizer Configuration

- there is **generally no need** to instantiate quantizers manually in HGQ2.
    - The `Q-` layers in `hgq.layers` API will include quantizers automatically by default for all datapaths that need quantization.
    - Manual quantizer is only needed if one known what they are doing
    - The user has the freedom to enable or disable quantizers at any datapaths. Watchout for footguns:
        - Missing quantization may lead to **bit-width blowing up** at conversion time.
        - Double quantization may lead to **degraded performance/model collapse**.

However, one **should** still configure the quantizers accordingly for the best performance.

Quantization configuration in HGQ2 follows a **hierarchical** approach. The relevant objects are `QuantizerConfigScope` and `QuantizerConfig`.

High Granularity Quantization - User Guide

Quantizer Configuration

Not found
(except overflow_mode)

Still Unfound

Match

Unfound (except ir and fr)

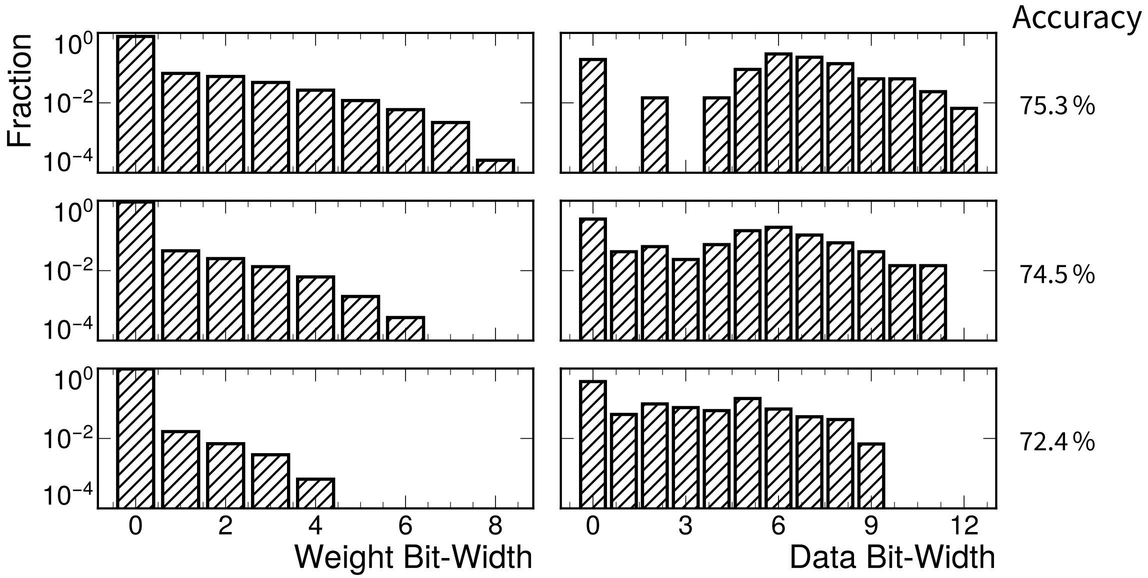
Match

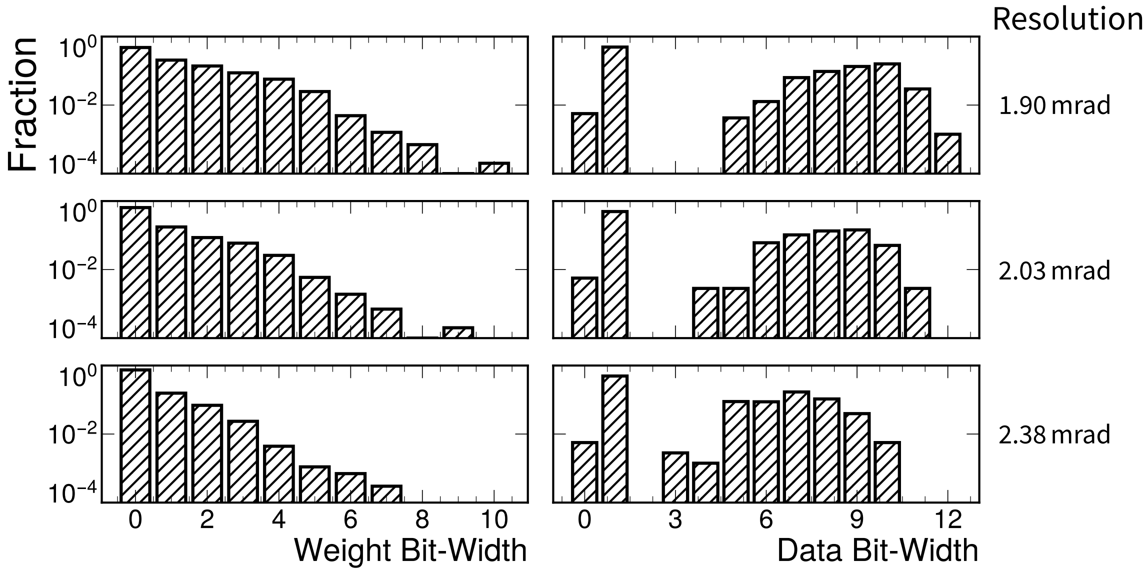
```python
with (
    QuantizerConfigScope(place='all', overflow_mode='SAT_SYM'),
    QuantizerConfigScope(
        place='datalane',
        default_q_type='kif',
        overflow_mode='WRAP'
    ),
    LayerConfigScope(enable_ebops=True, beta0=1e-5)
):
    oq_conf = QuantizerConfig('kif', 'datalane', fr=fr, ir=ir)
    model = keras.Sequential([
        QDense(64, activation='relu'),
        QDense(32, activation='relu'),
        QDense(32, activation='relu'),
        QDense(5, enable_oq, oq_conf=oq_conf),
    ])
```

For kernel/bias:
Not match

Bit-width obtained

When applying HGQ at per-parameter level, the resultant bitwidth is usually a very sparse and peaked at 0/1 bit (pruned/powers-of-2):





FPGA results will be presented in the following slides

HGQ-LUT for Efficient LUT-based Neural Networks

Replacing individual multiplications with generic LUT operations

A dense layer $y_i = \phi(\sum_j W_{ij} \cdot x_j + b_i)$ is replaced by $y_i = \sum_j \mathrm{LUT}_{ij}(x_j)$. Since each LUT are arbitrary functions, we skip the bias term here.

set $\mathrm{LUT}_{ij}(x) = W_{ij} \cdot \phi(x) + b_i$

- W_{ij}, b_i: weights and bias from the current layer
- ϕ: activation function **from the last layer**

$\Rightarrow$ LUT-based layer can exactly represent the original dense layer.

$\Rightarrow$ MLP with such LUT layers can represent any conventional dense MLP

- Train sub-networks with LUT-based cost regularization.
  - For each table, LUT6 $\approx \frac{1}{2} \cdot \left(\mathrm{bw}_{\mathrm{in}} \geq 5 \; ? \; 2^{\max(\mathrm{bw}_{\mathrm{in}}-5,0)} \; : \; \frac{\mathrm{bw}_{\mathrm{in}}}{5} \right) \cdot \mathrm{bw}_{\mathrm{out}}$

HGQ-LUT for Efficient LUT-based Neural Networks

Implementation

- Implemented as `hgq2.layers.DenseT/ConvT1/2/3D`
- LUT_{ij} learnable as individual MLP
  - One hidden layer appears to be sufficient
- GPU-friend implementation with `EinsumDense` layers
- $11\mathrm{s}\ (\mathrm{NeuraLUT\text{-}Assemble}) \rightarrow 20\mathrm{ms}$ required for each epoch of training
  - dataset: JSC CERNBox
  - Hardware 13700k + RTX4090
- SOTA resource efficiency on FPGA on JSC HLS/PLF datasets
- Lower latency than plain HGQ models
- Suitable for parts of networks when per-input number information is high and the input bit-widths are moderate (5-8 bits)

Model Training & Performance Considerations

If there is no special needs, it is **strongly recommended** using the `jax` (preferred) or `tensorflow` (less preferred) backends for model training with the native `model.fit` api for the **best training performance**:

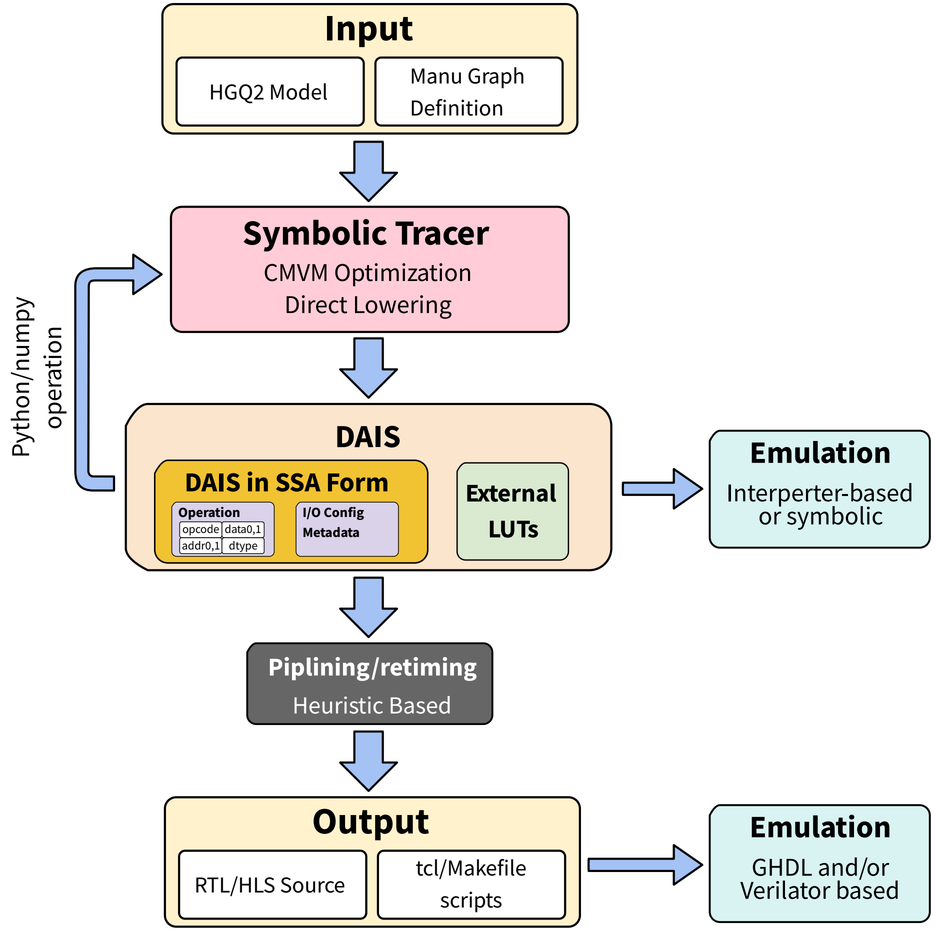
Fake quantization (float-point emulation) used by HGQ introduces many relatively cheap, element-wise calls, and XLA compilation (`jit_compile`) provides massive speedups for these operations by fusing them together.

On the other hand, training with a native `torch` training loop is also possible, though the performance may not be as optimized as with the aforementioned backends. `torch dynamo` support is not yet available.

What is da4ml

da4ml is a static computation graph to RTL/HLS compiler

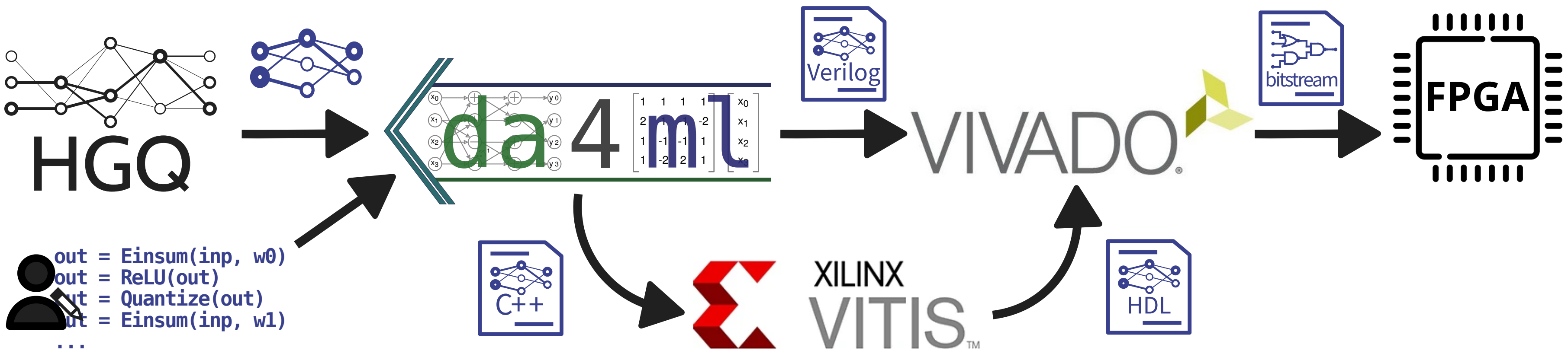
- The computation graph is implemented as fully pipelined or combinational logic
- Distributed Arithmetic (DA) optimization is applied when applicable
- Bit-exact emulation supported via DAIS interpreter and behavioral simulation
- Two usage modes:
  - Standalone code generation
  - As hls4ml plugin



What is da4ml

Two ways of using it:

Standalone

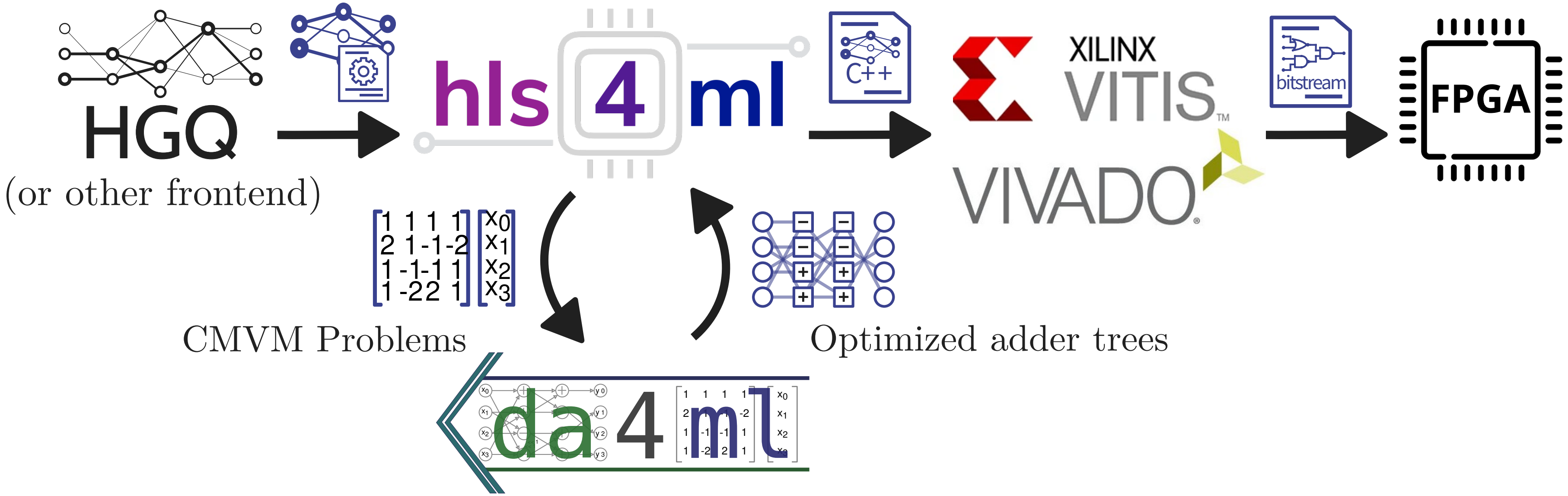


Convert HGQ2, and/or manually traced computation graph to Verilog/VHDL or HLS code.

What is da4ml

Two ways of using it:

As hls4ml plugin



By set `Strategy` to `distributed_arithmetic` in hls4ml, the CMVM operations will be offloaded to da4ml/hls implementation.

Components

da4ml has two main components:

CMVM Optimization

- Optimize CMVM with a custom two-stage DA algorithm
  - Graph-based matrix decomposition
  - Common subexpression elimination
- Generate efficient adder graph for the operation in reasonable time

Computation Graph Compiler

- Arbitrary static computation graph definition via symbolic tracing
- RISC-like internal representation
  - Bit-exact emulation
- Verilog/VHDL, and ap/ac_types HLS code generation

CMVM Optimization

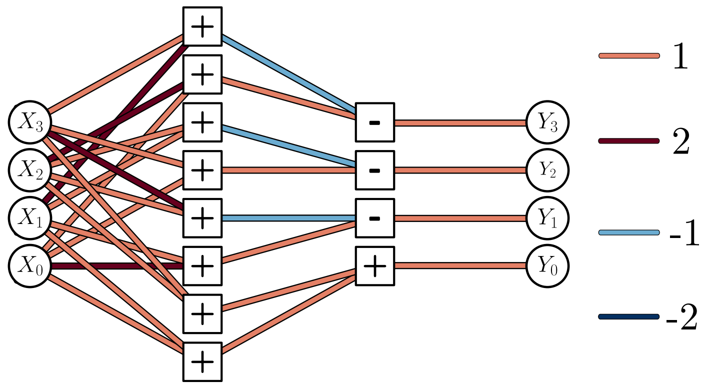
What is Distributed Arithmetic (DA)?

Here, we refer to DA as implementing constant multiplication(-accumulation) as shift-add operations. In Fast ML applications, the operation in concern is usually CMVM (i.e., in `Dense`, `Conv`, `EinsumDense`, `RNN` family, etc).

How is it useful?

When `II=1` is required for the individual CMVM operation, DA can be more efficient than the default HLS implementation. (note: II>1 for Latency strategy usually makes little sense)

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & 1 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \longrightarrow$$

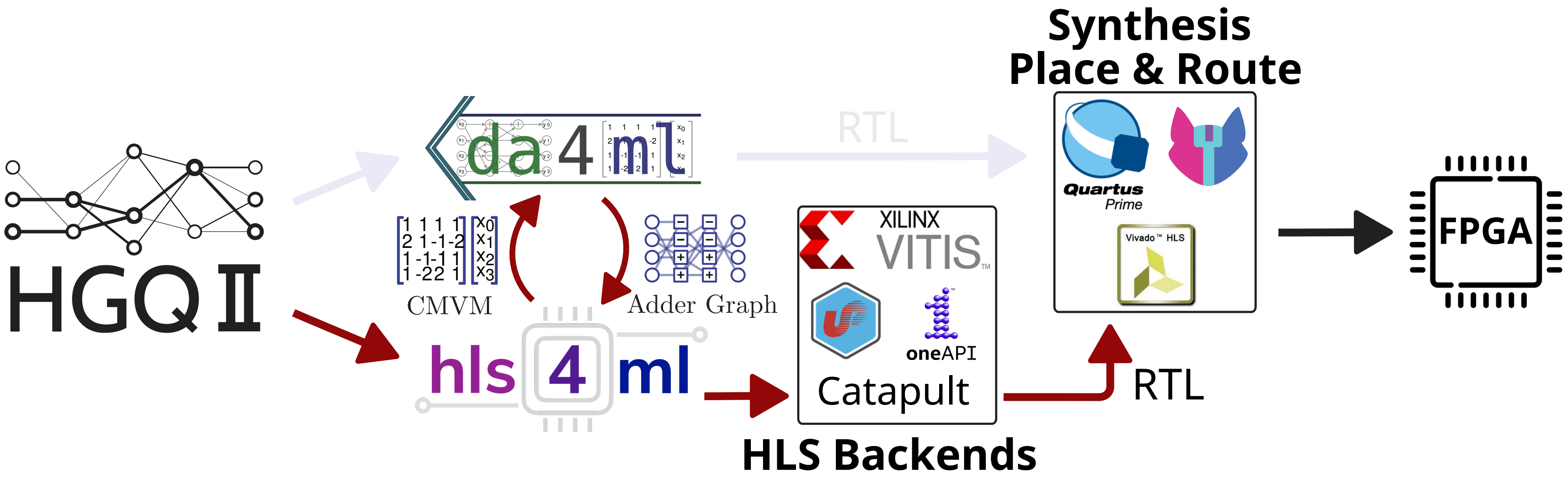


Algorithm Overview

The core of da4ml is an efficient algorithm for optimizing CMVM operations, $y = Wx$, where W is a constant matrix as an efficient adder graph on FPGAs.

The optimization is two-staged, hybrid approach:

- stage 1: High-level Graph-based Decomposition
- stage 2: Greedy Common Subexpression Elimination



CMVM Optimization

Signed Digit Representation

In a signed digit representation, an integer (or fixed point number) is represented by sequence of ternary digits {-1,0,1}, instead of binary {0,1}.

One number have multiple representation:

```
7 = { 0, 1, 1, 1} = 4 + 2 + 1
  = { 1, 0,-1, 1} = 8 - 2 + 1
  = { 1,-1, 1, 1} = 8 - 6 + 4 + 1
  = { 1, 0, 0,-1} = 8 - 1 // This is CSD
```

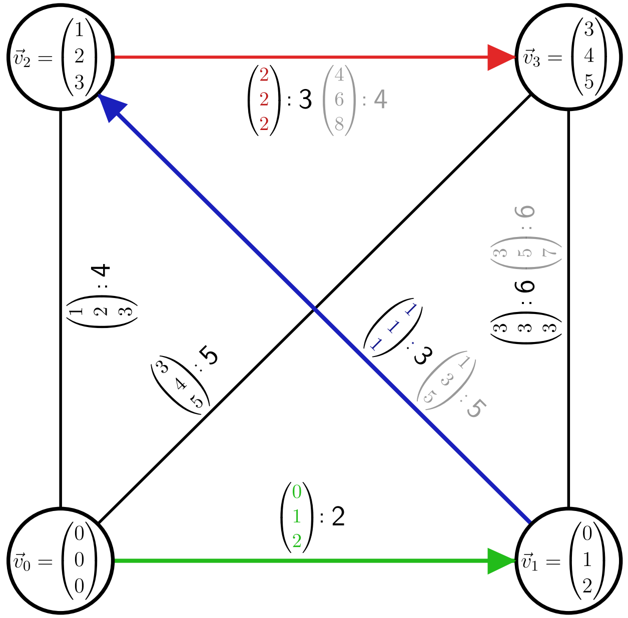
Canonical Signed Digit (CSD) is a unique representation of the number such that:

- No two consecutive digits are non-zero
- Minimal number of non-zero digits is guaranteed

Stage 1: Graph-based Matrix Decomposition

The first stage aims to exploit high-level structural similarities between the columns of the constant matrix W. Three steps are involved:

1. **Graph Construction**: A graph is constructed where each column vector of the matrix are the vertices.
   - Distance (v_1, v_2)=min(#CSD digits in $v_1 \pm v_2$)
2. **Minimum Spanning Tree**: An MST connects the columns in a way that minimizes the total "cost"
3. **Matrix Decomposition**: Decompose W into $W = W_1 W_2$ following the MST constructed.

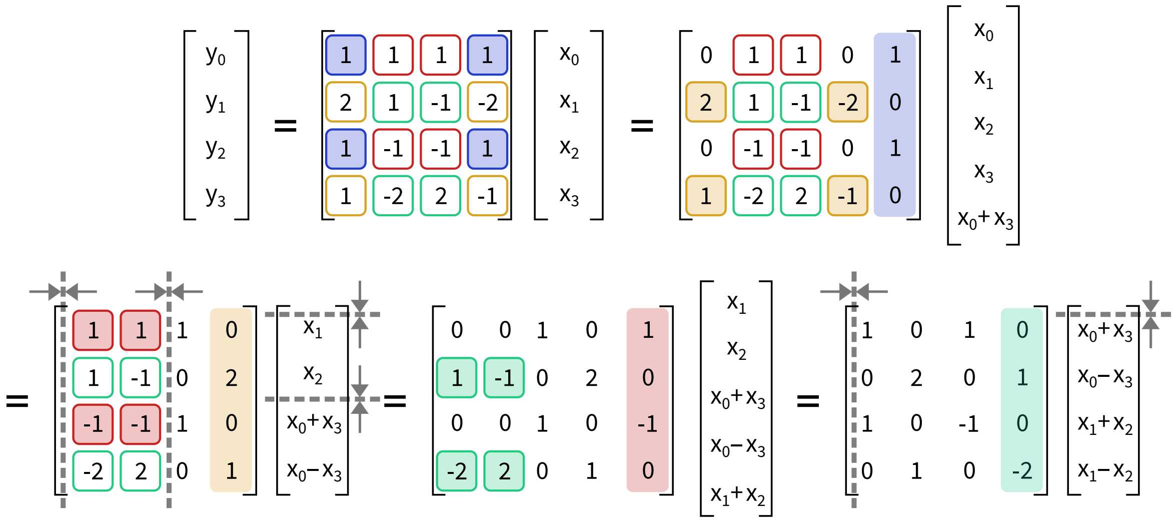


$$\begin{pmatrix}0 & 1 & 3\\ 1 & 2 & 4\\ 2 & 3 & 5\end{pmatrix} = \begin{pmatrix}0 & 1 & 2\\ 1 & 1 & 2\\ 2 & 1 & 2\end{pmatrix}\begin{pmatrix}1 & 1 & 1\\ 0 & 1 & 1\\ 0 & 0 & 1\end{pmatrix}$$

Stage 2: Common Subexpression Elimination (CSE)

The second stage applies a greedy CSE algorithm applied to both W_1 and W_2

1. **Convert to CSD**
2. **Greedy subexpression finding**: iteratively finds the most frequently two-term subexpression $a \pm (b << s)$
3. **Substitution**: Implement the found subexpression as a single operation, replace all its occurrences

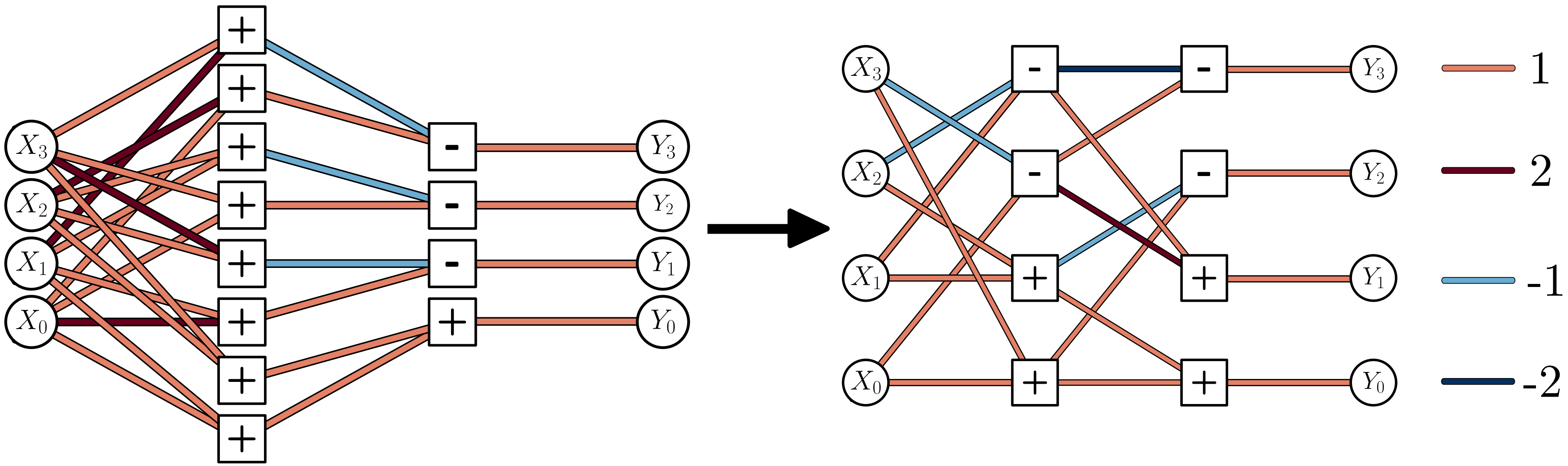


The selection of the most common subexpression in step 2 is weighted by the bitwidths of the two operands in practice

CMVM Optimization

Example

The example h264 matrix will be implemented like left by default. With da4ml, the number of adders/substractor needed is reduced from $12 \to 8$.



Computation Graph Compiler

Overview

- Frontend: Symbolic tracing
  - Internally, the computation graph is always defined via symbolic tracing
- Optimization:
  - There is only CMVM optimization currently, which is applied automatically when applicable on the frontend during tracing
- IR: Distributed Arithmetic Instruction Set (DAIS)
  - RISC-like, assembly-style DSL that is SSA; ~1:1 mapping to RTL/HLS code
  - Bit-exact (i64-based) interpreter provided
- Backend: Code Generation
  - Rather straightforward code emission

Frontend: Symbolic Tracing

Internally, the computation graph is always defined via symbolic tracing.

Consider the following operation:

```python
import numpy as np
from da4ml.trace.ops import quantize, relu

w = np.random.randint(-2**7, 2**7, (4, 5, 6)) / 2**7

def operation(inp):
   inp = quantize(inp, 1, 7, 0)

   out1 = quantize(np.sin(relu(inp)), 1, 0, 7, 'SAT', 'RND')

   out2 = np.repeat(np.abs(inp[:, 1:3].T), 2, axis=0) * 3 + 4

   out3 = quantize(out2 @ out1, 1, 10, 2)
   out3 = np.einsum('ijk,ij->ik', w, out3)

   return np.concatenate([out1, out2, out3], axis=-1)
```

Computation Graph Compiler

Frontend: Symbolic Tracing

The tracing is done by calling `comb_trace` on the input and output of the operation:

```
from da4ml.trace import FixedVariableArrayInput, comb_trace

inp_tensor = FixedVariableArrayInput((4, 5))
out_tensor = operation(inp_tensor) # <- CMVM optimization happens here

comb_logic = comb_trace(inp_tensor, out_tensor)
```

`comb_logic` now contains the traced computation graph with optimized CMVM operations.

Frontend: Symbolic Tracing

da4ml can automatically trace the operations for HGQ2 trained models.

- If `SAT/SAT_SYM` is used for activation, one needs to specify the actual input precision

```
inp_tensor, out_tensor = trace_model(model)
comb_logic = comb_trace(inp_tensor, out_tensor)
```

One can apply further operations on these tensors, to obtain AE score for example:

```
inp_tensor, out_tensor = trace_model(ae_model)
q_input = quantize(inp_tensor, 1, 7, 0)
q_output = quantize(out_tensor, 1, 7, 0)
score = np.sum((q_input - q_output) ** 2)
comb_logic = comb_trace(inp_tensor, score)
```

IR: Distributed Arithmetic Instruction Set (DAIS)

DAIS is the internal representation used by da4ml: a RISC-like DSL that is SSA. 9 instructions are used to perform any high-level operations.

Some examples instructions:

- `copy(inp_buf[id0])`
- `buf[id0] +/- buf[id1] * 2^data0`
- `relu(+/- buf[id0])`
- `buf[data0].bits[0] ? buf[id0] : buf[id1] * 2^data`

Interpreter for binary DAIS dump is also provided for **bit-exact** emulation of generated firmware.

Binary DAIS dump format:

```
- spec_version: i32
- firmware_version: i32
- n_in: i32
- n_out: i32
- n_ops: i32
- n_tables: i32
- inp_shift: i32[n_in]
- out_idxs: i32[n_out]
- out_shifts: i32[n_out]
- out_negs: i32[n_out]
- ops: Op[N]
- table_sizes: i32[n_tables]
- tables: i32[sum(table_sizes)]

Op: - opcode: i32
    - src: i32[2]
    - data: i32[2]
    - dtype(s,i,f): i32[3]
```

Computation Graph Compiler

IR: Distributed Arithmetic Instruction Set (DAIS)

Available instructions in DAIS: (*: allow -x for application on negative input)

```
- -1: copy from input buffer
-  0: shift-add
-  1: shift-subtract
- *2: relu
- *3: WRAP-TRN quantization
-  4: const-add
-  5: const-def
- *6: muxing
-  7: multiplication
-  8: table-lookup
```

Almost all combinational operation can be efficiently mapped to these instructions.

IR: Distributed Arithmetic Instruction Set (DAIS)

- With the generic table-lookup instruction, DAIS can also represent non-linear functions efficiently.
- This includes any lookup-based networks (i.e., NeuraLUT-like ones). Preliminary HGQ-LUT support was added in da4ml v0.5.0beta0 and available for use.
  - Forcing LUTRAM impl on Vivado for now (may change in the future)
  - Corresponding HGQ2 layer: `DenseT`

Computation Graph Compiler

Backend: Code Emission

HDL emission is straightforward from DAIS IR (already SSA and assembly-like). The generated code is fully pipelined (pseudo-delay based pipelining for combinational logic) or combinational.

Standard Verilog/VHDL2008 are supported (no vendor-specific primitives).

- Only `model_input[Y:X]`, `model_output[N:M]`, and `clk` when pipelined: no handshaking is really needed for a II=1 module with fixed latency.
- Built-in support for Verilator/GHDL behavioral simulation.

For HLS codegen, `ap_types` (Vitis/Vivado HLS) and `ac_types` (`hlslibs`, or `OneAPI` *) are supported.

- hls4ml internally uses this path for DA strategy.

*: `OneAPI` doesn't support 1-bit numbers, patched to 2 bits when used. May compromise bit-exactness.

Backend: Code Emission

Generating the model project is straightforward:

```python
from da4ml.codegen import HLSModel, RTLModel

# Or HLSModel for HLS code generation (table-lookup is not supported yet for HLS)
# latency_cutoff is for piplining. Set to negative for combinational logic.
rtl_model = RTLModel(comb_logic, 'vmodel', '/tmp/rtl', flavor='verilog', latency_cutoff=5)
rtl_model.write() # write files to /tmp/rtl
```

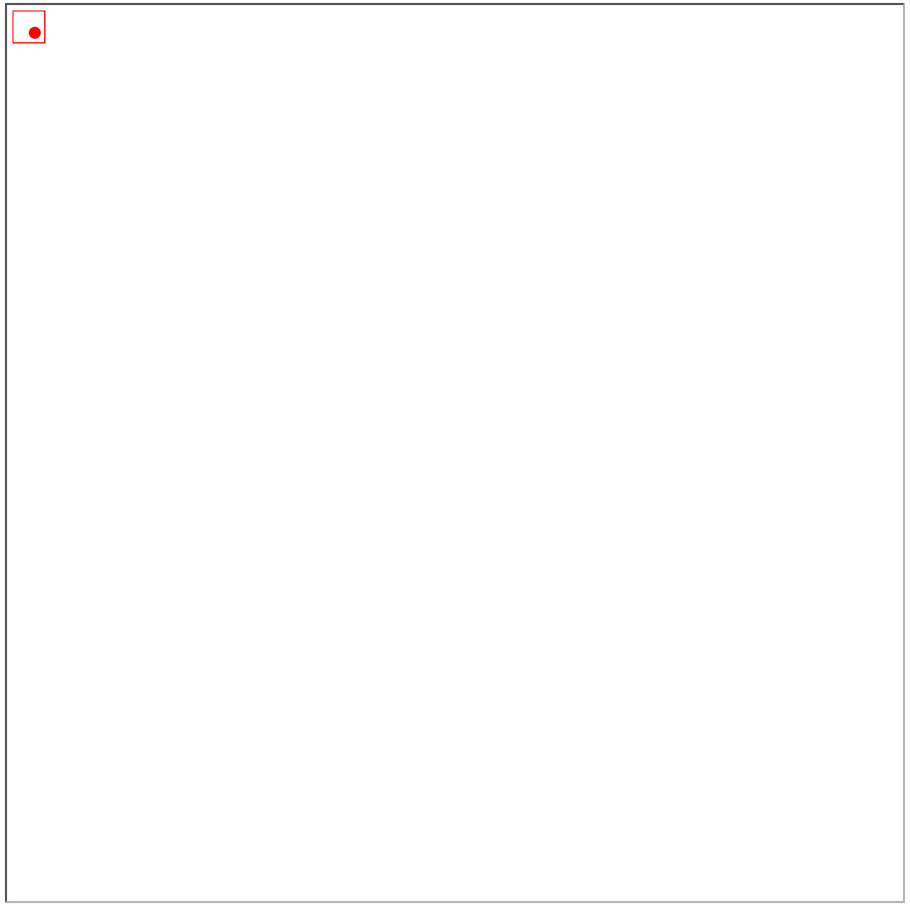
Computation Graph Compiler

Backend: Resource Estimation

The `RTLModel/HLSModel` objects gives good LUT (and FF) usage (~10% for LUTs, ~20% for FF)

```
> print(verilog_model)

Top Function: hlsmodel
====================
20 (160 bits) -> 24 (486 bits)
combinational @ delay=(21.0, 22.0)
Estimated cost: 1551 LUTs
Emulator is compiled with OpenMP (d9f5cd8e6c99)
```



hls4ml Integration

In this mode, only CMVM operations in the `Dense`, `EinsumDense`, and `Conv1/2D` layers will be affected.

```
from hls4ml.converters import convert_from_keras_model

model_hls = convert_from_keras_model(
    model,
    hls_config={'Model': {'Strategy': 'distributed_arithmetic', ...}, ...},
    ...
)

model_hls.write()
```

The obtained project can be used like any other `hls4ml` project.

Tip: `sum(g.attributes.get('da_kernel_cost', 0) * g.attributes.get('parallelization_factor', 1) for g in model_hls.graph.values())` gives a good estimate of the LUTs consumed by the CMVM kernels.

Verification

When directly used, da4ml provides two ways for bit-exact verification:

- DAIS interpreter (`comb_logic.predict(...)`)
  - Needs installation time compilation of the c++ interpreter on the client side as of `v0.5.0beta1`. Will move to pre-compiled binary dist in the future.
  - Fast, good for unit tests, and bit-exact with i64 arithmetic.
- Behavioral simulation on generated source (`hls/rtl_model.predict(...)`)
  - With Verilator/GHDL/HLS cpp headers
  - Verilator can be even faster, but compilation will be slow. HLS inference simulation will be slow both for compilation and execution.

When used as hls4ml plugin, hls4ml's own verification flow can be used as usual.

Performance Gain

Comparison of Latency and Resource Utilization w/ and w/o da4ml

We show a micro-benchmark and some results from the JSC OpenML dataset here.

Please refer to the paper for more details.

Performance Gain - Micro-benchmark

$N \times N \times n$bit matrix $\times$ 8 bits input vector

Impl	Size	LUT	DSP	FF	Latency [ns]
hls4ml	$16 \times 16 \times 8$bit	4319	212	2301	3.05
hls4ml+DA	$16 \times 16 \times 8$bit	4545	0	1618	2.66
hls4ml	$64 \times 64 \times 8$bit	70821	2897	18969	5.63
hls4ml+DA	$64 \times 64 \times 8$bit	63852	0	20509	4.91
hls4ml	$16 \times 16 \times 8$bit	4538	0	1585	1.88
hls4ml+DA	$16 \times 16 \times 4$bit	2268	0	745	2.34
hls4ml	$32 \times 32 \times 4$bit	13550	0	3761	2.44
hls4ml+DA	$32 \times 32 \times 4$bit	7452	0	2465	2.93
hls4ml	$64 \times 64 \times 4$bit	47274	0	14652	3.10
hls4ml+DA	$64 \times 64 \times 4$bit	26715	0	6026	4.18

Performance Gain - JSC OpenML

Impl	Accuracy	Latency	LUT	DSP	FF	F_max [MHz]
hls4ml		57.6 ns	16,081	57	26,484	729.4
da4ml	76.9 %	34.0 ns	12,298	0	13,664	588.6
hls4ml+DA		44.1 ns	12,682	0	19,056	702.2
Latency		67.2 ns	8,548	30	14,418	520.8
da4ml	76.5 %	23.1 ns	6,165	0	7,207	735.8
hls4ml+DA		35.4 ns	6,448	0	10,109	707.2

Combined results

Performance of HGQ2 and da4ml together on a few benchmark tasks compared to other methods:

- JSC OpenML (16 high-level-feature)
- JSC CERN Box (16 high-level-feature)
- hls4ml jet tagging dataset (particle based, 3 features)

JSC OpenML

Higher latency than other pure LUT-based methods, but similar LUT usage and can achieve higher accuracy.

Implementation	Accuracy	Latency [ns]	LUT	DSP	FF	F_{max} [MHz]	II [cc]
`HGQ-LUT`	76.9%	9.2	5,667	0	2,434	650.	1
`HGQ+hls4ml/DA`	76.9%	44.1	12,682	0	19,056	702.	1
`HGQ-LUT`	76.5%	6.5	2,295	0	1,145	771.	1
`HGQ+da4ml`	76.5%	23.1	6,165	0	7,207	736.	1
`HGQ-LUT`	76.0%	6.4	1,293	0	822	777.	1
`HGQ+hls4ml`	76.9%	57.6	16,081	57	26,484	729.	1
`HGQ+hls4ml`	76.5%	67.2	8,548	30	14,418	521.	1
`QKeras+hls4ml` [1]	76.3%	105	5,504	175	3,036	143.	2
`DWN` [2]	76.3%	14.4	6,302	0	4,128	695.	1
`MetaML-Pro` [3]	76.1%	50	13,042	70	N/A	200	1
`TreeLUT` [5]	75.6%	2.7	2,234	0	347	735.	1
`NeuraLUT-Assemble*` [4]	75.5%	3.6	2,036	0	420	558.	1

*: results reproduced after fixing some issues in the original evaluation protocol.

JSC CERN Box

Higher latency than other pure LUT-based methods, but lower LUTs or higher accuracy.

Implementation	Accuracy	Latency [ns]	LUT	DSP	FF	F_{max} [MHz]	II [cc]
`HGQ-LUT`	75.4%	10.1	6,042	0	2,438	593.	1
`HGQ+da4ml`	75.2%	37.8	8,703	0	10,008	503.	1
`HGQ-LUT`	75.1%	9.0	3,391	0	1,675	663.	1
`HGQ+da4ml`	75.0%	27.4	5,636	0	6,218	656.	1
`HGQ-LUT`	74.5%	6.0	1,435	0	903	833.	1
`NeuraLUT-Assemble*` [4]	74.9%	10.3	8,819	0	2,770	680.	1
`AmigoLUT-NeuraLUT` [6]	74.4%	9.6	42,742	0	4,717	520.	1
`PolyLUT-Add` [7]	75.%	16	36,484	0	1,209	315.	1
`NeuraLUT` [8]	75.%	14	92,357	0	4,885	368.	1
`PolyLUT` [9]	75.1%	25	246,071	0	12,384	203.	1
`LogicNets` [10]	72.%	13	37,931	0	810	427.	1

*: results reproduced after fixing some issues in the original evaluation protocol.

hls4ml jet tagger dataset (particle based)

hls4ml jet tagging dataset (particle based), 3 feature per particle, N particles

Implementation	N	Accuracy	LUT	DSP	FF	Latency [ns]	$\mathrm{F}_{\mathrm{max}}$ [MHz]	II [cc]
`JL (HGQ+da4ml)`	32	79.0%	80. k	0	136. k	79	299.	1
`JL (HGQ-LUT)`	32	78.3%	42. k	0	16. k	30	368.	1
`DS (QK+hls)` [11]	32	<75.9%	130. k	434	903. k	359	N/A	2
`GNN (QK+hls)` [11]	32	<75.8%	205. k	2,120	1,162. k	761	N/A	3
`JL (HGQ+da4ml)`	8	66.5%	79. k	0	136. k	73	303.	1
`JL (HGQ-LUT)`	8	65.9%	61. k	0	12. k	40	201.	1
`DS L (QK+hls)` [12]	8	66.6%	135. k	2,458	337. k	140	N/A	3
`DS M (QK+hls)` [12]	8	65.1%	110. k	548	130. k	49	N/A	3
`DS (QK+hls)` [11]	8	<64.0%	95. k	626	386. k	121	N/A	3
`GNN (QK+hls)` [11]	8	<64.9%	160. k	2,120	472. k	192	N/A	3