

Tutorial on HGQ and da4ml

Chang Sun

Caltech, PMA

Backgrounds - Fixed Point Numbers

What is a fixed point number?

Short answer: An integer with some power-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:

$$5.28125 = 0101.01001 \quad (k=1, i=3, f=5)$$

The representable range is

$$[-k \cdot 2^{i-1}, 2^{i-1} - 2^{-f}], \text{ with step } 2^{-f}.$$

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

$$\begin{aligned} 0.0859375 &= 0.0001011 & : (k=0, i=-3, f=7) \\ -24. &= 111000. & : (k=1, i=5, f=-3) \end{aligned}$$

(Using two's complement representation here)

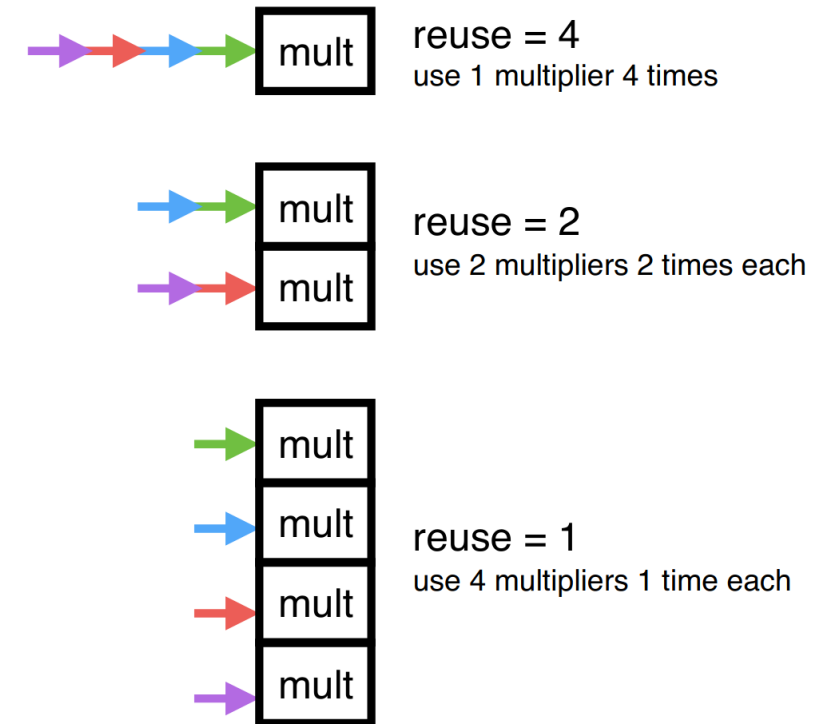
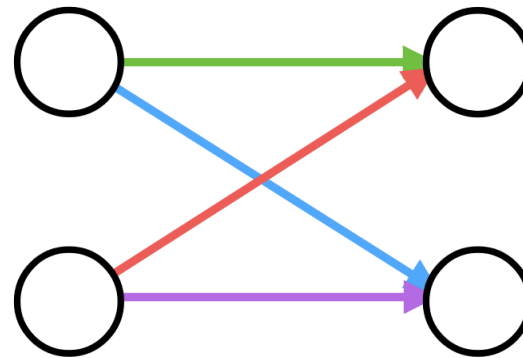
Backgrounds - CMVM on FPGAs

For a constant matrix vector multiplication, how can resource be reused (hls4ml as an example)?

`reuse_factor` in hls4ml controls the II required for an individual CMVM operation. Everyone has probably seen this famous figure regarding how reuse factor works -- is it always the case?

(Spoiler: **No** for latency strategy)

Unrelated: this figure shows an outer product, which is unsupported by hls4ml.



Backgrounds - CMVM on FPGAs

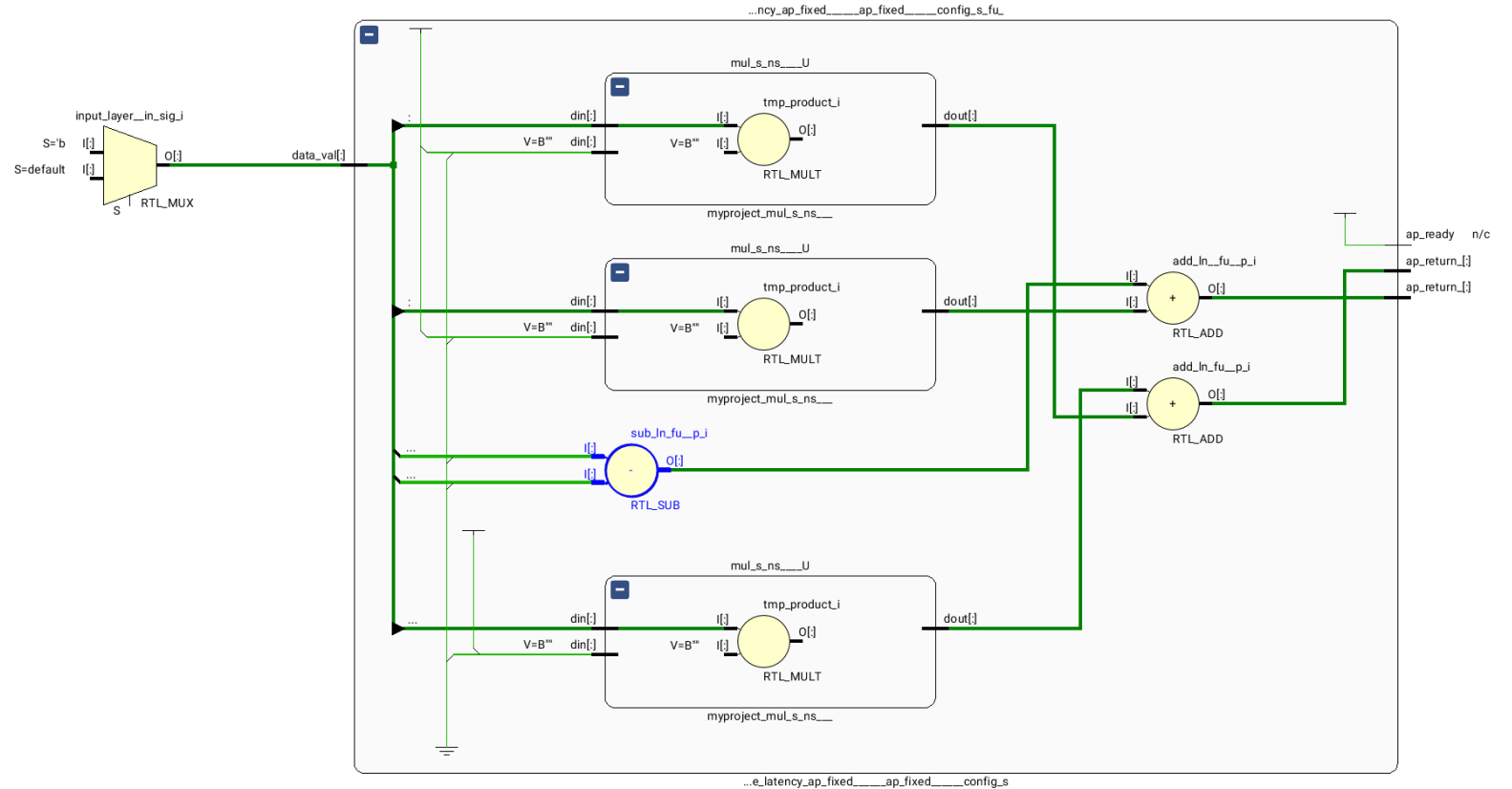
Toy Example 1

A 1-dense-layer NN, kernel size: 2×2 , no bias, $II=1$:

Operation:

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} 21 & 23 \\ 22 & 24 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

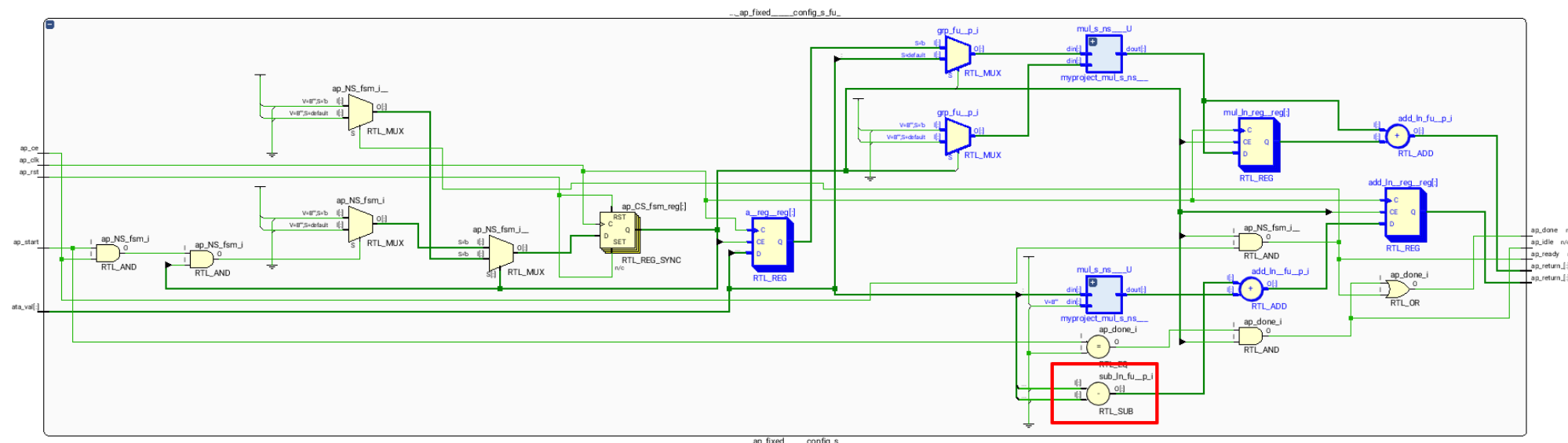
Here, as $24 = 32 - 8$,
 $24 \cdot x_1$ is implemented as a **subtractor** instead of a real multiplier.



Backgrounds - CMVM on FPGAs

Toy Example 1

The same network, but with II (**reuse_factor**)=2:



Here, the $24 \cdot x_1$ is still implemented as a **subtractor**, and one **multiplier** is used **only once**. The other **multiplier** is reused two times with a finite state machine. **Auxiliary muxing and registering logics** are added automatically, as expected.

Backgrounds - CMVM on FPGAs

Toy Example 2

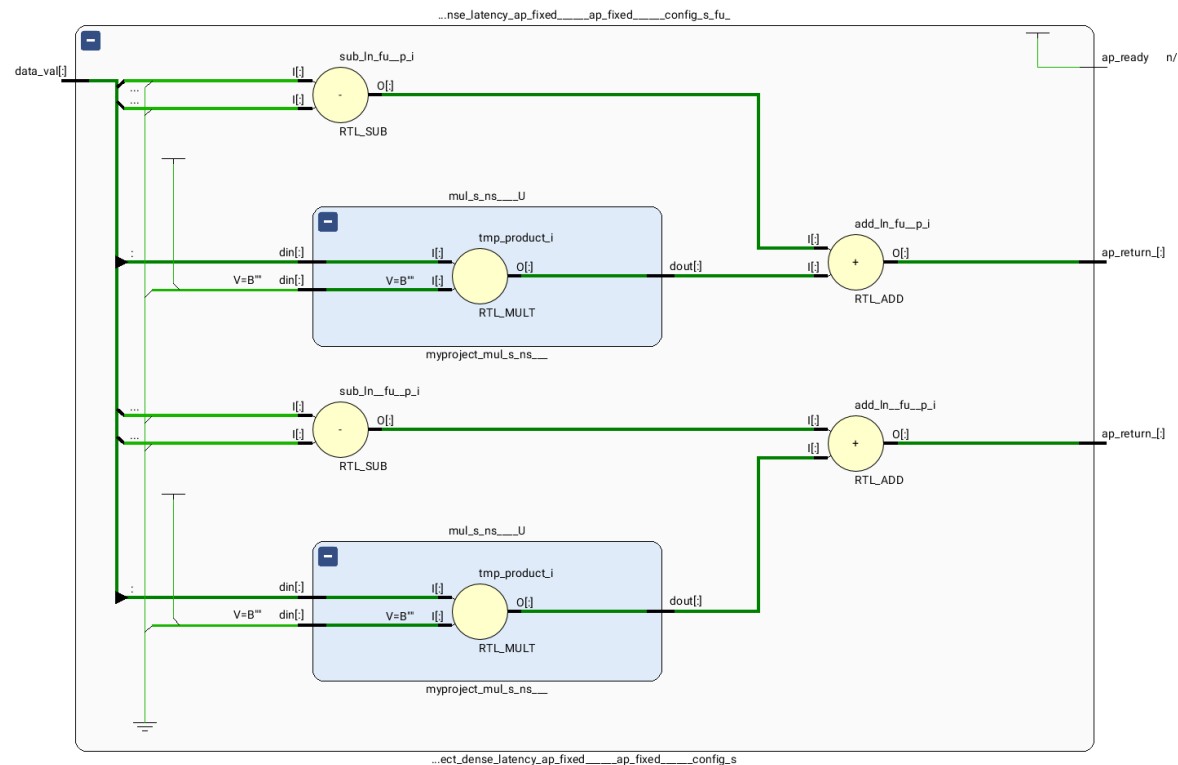
A 1-dense-layer NN, kernel size: 2×2 , no bias, $II=1$ or 2:

Operation:

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} 77 & 12 \\ 85 & 14 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

Both $12 \cdot x_1$ and $14 \cdot x_1$ are implemented as subtractors this time ($12 = 16 - 4$, $14 = 16 - 2$), and the compiler refused to comply with the `II=2` constraint.

Transposing the matrix makes no difference.



Backgrounds - CMVM on FPGAs

Toy Example 3 (1 and 2 together)

2-layer-dense-NN with no bias and set II=2:

Operation:

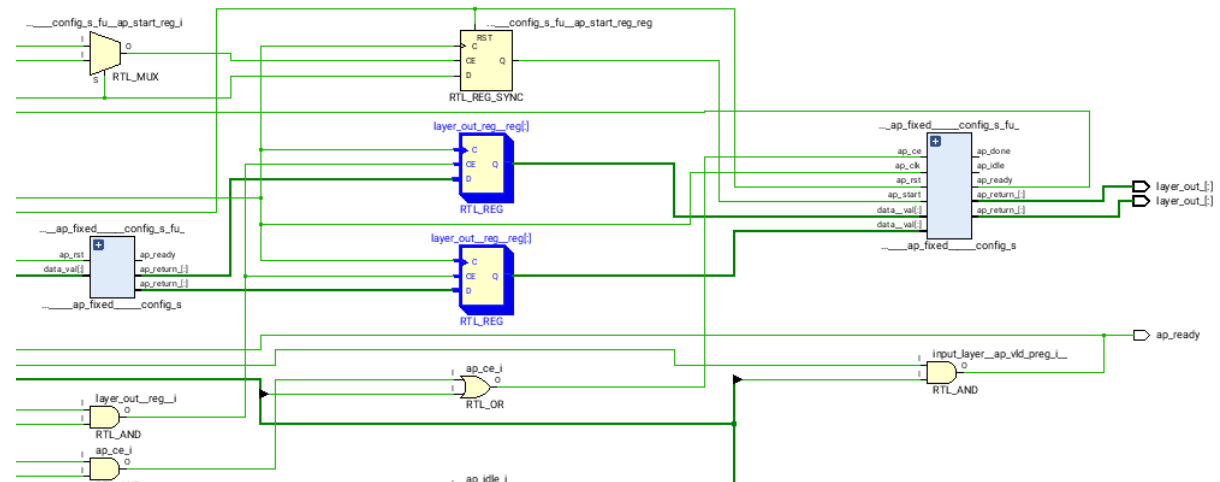
$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} 21 & 23 \\ 22 & 24 \end{pmatrix} \begin{pmatrix} 77 & 12 \\ 85 & 14 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

The **modules** for the two layers are exactly as-is to the case where they are synthesized individually. However,

extra registers is added to "make"

II=2 for the first layer which has

internal **II=1**



When adders/subtractors are used instead of multipliers?

When the multiplication to that weight can be represented as a single addition/subtraction operation \equiv only two bits in the minimal signed representation of the weight.

As 0-bit weight is trivial, 1-bit weight is simply a wire/negation, they also do not use multiplier. Hence **when the number of digits is smaller or equal to two**, Vitis HLS 2025.1 (Likely also older versions) do not use multipliers, and **reuse factor will not be useful for those operations**.

How often does this happen?

Very often \rightarrow see next page.

Distribution of number of signed digits in weights

As a simplified model, for a weight w of W bits, we assume the weights are **signed** and **uniformly distributed** across the range of representable values. With these assumptions, the **ratio of weights has ≤ 2 signed digits** is shown in the right.

As realistically, the weights are closer to a gaussian distribution, the actual ratio will **usually be much higher**

Bitwidth	Ratio ≤ 2	Bitwidth	Ratio ≤ 2
3	100.0%	8	34.4%
4	100.0%	9	22.7%
5	87.5%	10	14.5%
6	68.8%	11	9.0%
7	50.0%	12	5.5%

Takeaways - In latency strategy:

1. In the latency strategy, the network's implementation depends on **the values of the weights**, not just the declared bitwidths.
2. **Most of the low-latency NN for L1T** are LUT heavy due to the **low number of digits** in the weights.
3. **Use reuse factors with great caution.** They do not always (or, **usually not**) do what you expect from the simplified model. As a very rough approximation, consider **LUT consumption unchanged or slightly increased** due to muxing overhead and **DSP consumption divided by the reuse factor** when increasing it (DSP usage \sim real multiplier usage, but not always).
4. **II=1** almost always has the lowest LUT consumption.

Just use $II=1$ (in most cases)

Since resource reusing is not as effective as commonly expected in low-bitwidths, we focus on optimizing the designs for `reuse_factor=1`

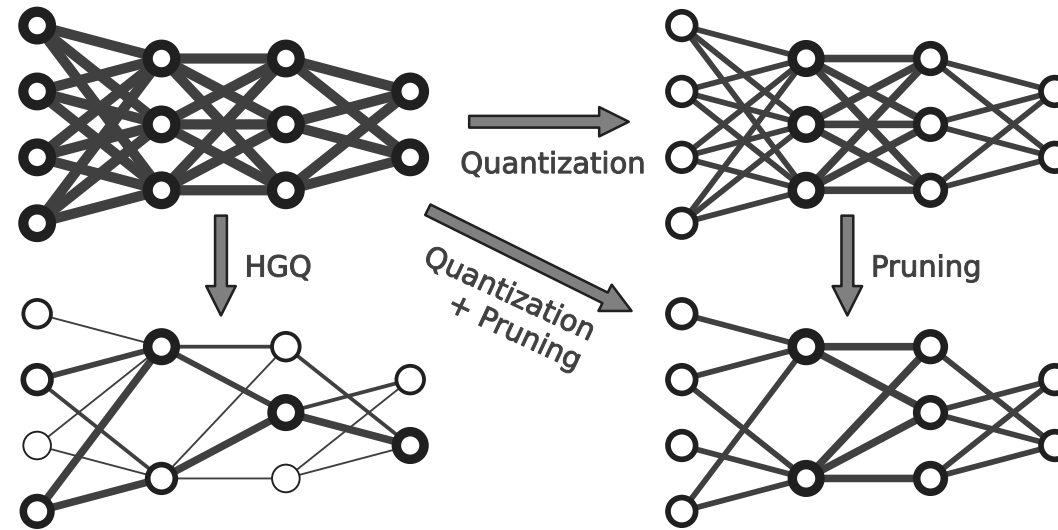
$II=1$ in CMVM opens up more opportunities for optimization: all operation can be fully heterogeneous now due to the lack of resource multiplexing.

Everything in this tutorial will target **CMVM $II=1$** case. Note that this is different to **Network $II=1$** , as reusing on the CMVM kernel level is still possible and does not introduce as significant overhead as inside the CMVM kernel.

High Granularity Quantization

What is HGQ

High Granularity Quantization (HGQ) is a framework and an algorithm that quantizes neural networks at arbitrary fine granularity. HGQ2 [[code](#), [doc](#), [paper](#)] will be used for this tutorial.



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

High Granularity Quantization

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 to QKeras -- up to two OOM.
- Model is **bit-exact** up to the floating point representation limit
- Supports most of the common layers
 - Core layers: `Dense/EinsumDense/Conv/DepthwiseConv` (with BatchNorm fused)
 - Pooling and binary operation (e.g., `(Global)*PoolingND/Sum/Einsum/Dot`)
 - Arbitrary nonlinear look-up table
 - Complex sugar layers (e.g., `MultiHeadAttention/LinformerAttention`)
- Keras v3 based, supporting all `jax/tensorflow/torch` backends
- seamless `da4ml` and `hls4ml` integration

High Granularity Quantization

Differentiable quantization

This gradient is the same to the one used for [LSQ](#). We use the LSQ derivation here for brevity; the derivation we used is different, see [paper](#) for more details.

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

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

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

$$\mathcal{L} = \mathcal{L}_0 + \beta \cdot \text{EBOPs} + \gamma \cdot \sum \text{bitwidths}$$

High Granularity Quantization

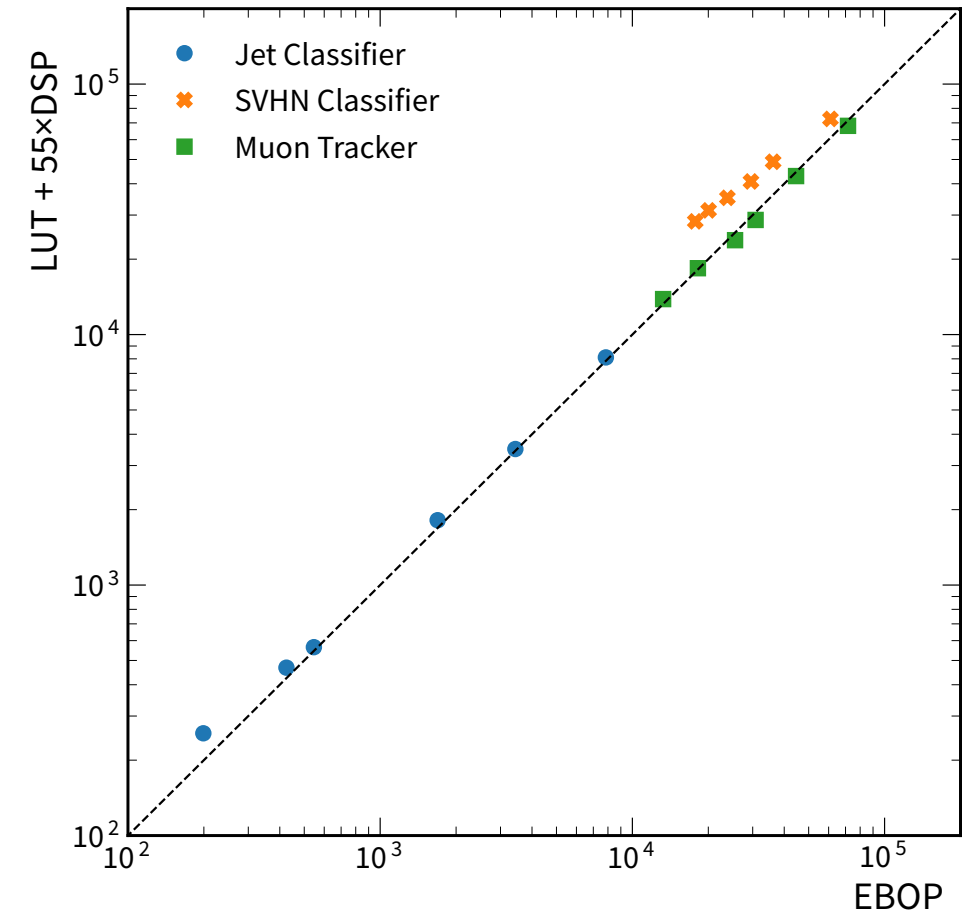
Effective Bit Operations

$$\text{EBOPs} = \sum \text{bw}_i \cdot \text{bw}_j$$

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

In HGQ2, we also add EBOPs for `ADD/SUB`, as well as a very rough estimation of the table size for look-ups into the calculation.

If `da4ml` is not used, on Xilinx FPGAs,
 $\text{EBOPs} \approx \text{LUT} + 55 \cdot \text{DSP}$ with hls4ml synthesized models



High Granularity Quantization - Concepts

Quantization

Quantization is the mapping of real number (or fp number) to a fixed-point representation in this context. In HGQ2, it is handled by the `quantizers` library.

There are two major types of values in concern:

- **Weight-like:** The parameters **known** at synthesis time.
- **Activation-like:** The parameters **not known** at synthesis time.

For the activation-like values, other than the bit-width, one needs to decide the exact behavior when casting something to it: how **rounding** and **overflow** are handled.

While arbitrary float-point quantization is also supported in HGQ2, we focus on the fixed-point case for now due to backend support status.

High Granularity Quantization - Concepts

Overflow and Rounding

In HGQ2, 3 overflow and 3 rounding modes are supported:

Overflow:

- **WRAP** - drop bits beyond MSB, **no overhead**, but ~trashes the results if happens
- **SAT** - saturation, clamps the values but **high overhead**
- **SAT_SYM** - **SAT** but symmetric - lower bound is $-k \cdot (2^i - 2^{-f})$ instead of $-k \cdot (2^i)$

Rounding:

- **TRN** - drops bits beyond LSB / floor, **no overhead**
- **RND** - rounding, ties up - same as adds 0.5 beyond the LSB before **TRN**
- **RND_CONV** - rounding, ties to even.

High Granularity Quantization - Concepts

bit-exact

A model/operation that is mapped to FPGA 1:1, without any precision loss.

quantized

Not a strict definition, but a value is considered quantized if it has sufficiently low bitwidth. However, generic input data (float point) is considered **strictly unquantized**.

Usually, they come from either explicit **quantizers** or **lookup-tables**. Values are **not** quantized in general if they come from **multiplying** two values.

non-trivial operation

All operations needs some kind of arithmetic operation. Essentially, anything that is not a 1. negation or 2. shift by a power-of-two. **ADD/SUB** are considered **semi-trivial**.

High Granularity Quantization - Consideration

Requirements

1. The model **should** be able to map to **bit-exactly**
2. All bitwidths in the design **must** be kept reasonable.

Rules

1. Fixed-point representation **must** be used throughout the design with sufficient bits to avoid numerical precision loss.
2. Any values engaged in highly non-trivial operations **must** be quantized, and **should** be quantized for other non-trivial operations when possible.

High Granularity Quantization - Consideration

Why?

Rule #1: If the model is not bit-exact, significant performance degradation would likely to occur with aggressive quantization.

Though, this is not always the case, especially when the model is not heavily quantized to begin with. Though, it is always more useful if the quantization framework provides a WYSIWYG (What You See Is What You Get) experience.

Rule #2: Otherwise, resource consumption will blow up.

This part probably won't need more explanation ;)

High Granularity Quantization - Consideration

Quantization in HGQ2

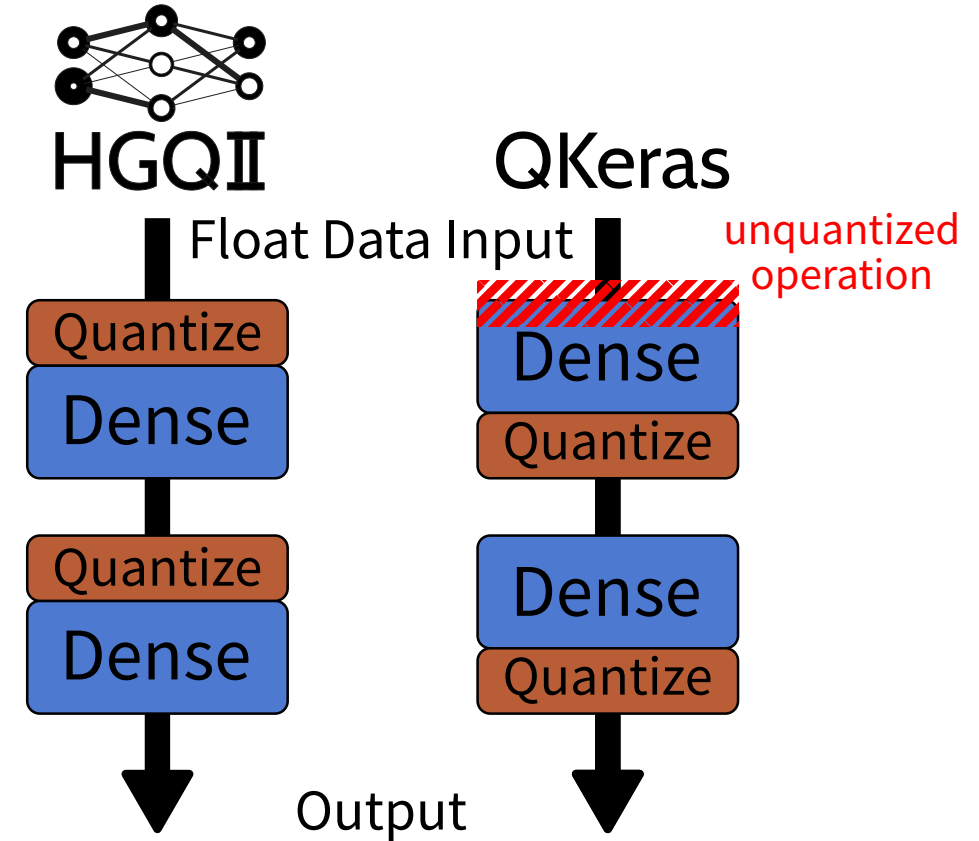
HGQ2 is designed to be flexible and provide a set of APIs like the native `Keras v3`. All of the following can be overridden, but **by default**:

- 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



High Granularity Quantization - Consideration

Quantization in HGQ2

Quantizers are configured in 4 places in HGQ2: `data lane`, `weight`, `bias`, and `table`:
`data lane` is Activation-like, whereas all others are weight-like.

Two parametrization of the quantizers are supported:

- `kif`: keep_negative, integers, fractional
- `kbi`: keep_negative, bitwidth (excl. sign), integers

If `WRAP` is used, the former is **should** be used for the activation-like quantizers (`data lane`). If `SAT/SAT_SYM` is used, the difference is minor.

Quantization in HGQ2

What's special about **WRAP** overflow mode for **data lane** quantizers?

Unlike **SAT/SYM**, **WRAP** is **not** enforced in the training time. During training, the number of integer bits is estimated based on the actual value range the network sees. Post training, one needs to **finalized the data lane quantizers' bitwidths with a calibration dataset**, where the integer values are updated.

If **kbi** is used, since **b** is fixed, the precision may change with this operation, leading to discrepancy in model accuracy before/after calibration.

Quantizer Configuration

Unlike QKeras, there is **generally no need** to instantiate quantizers manually in HGQ2. If proper `Q-` layers are used, quantization for all datapaths and weights will be handled automatically.

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

Quantizer Configuration

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

When a `QuantizerConfig` object is instantiated, for each of its attributes, the value is determined in the following hierarchy:

1. **Explicit Value:** Explicitly provided values are used as-is.
2. **Scope Value:** The innermost `QuantizerConfigScope` that matches `place` and `q_type` (`kif` or `kbi`) of the current `QuantizerConfig` object is used.
3. **Default Value:** Framework default values are used for each `place` and `q_type`.

High Granularity Quantization - User Guide

Quantizer Configuration

`default_q_type` argument controls should `kif` or `kbi` quantizer be used for a particular `place` by default. Not to be confused with `q_type`, which is a selector (default: `all`).

```
1 with (  
2     QuantizerConfigScope(place='all', overflow_mode='SAT_SYM'),  
3     QuantizerConfigScope(  
4         place='datalane',  
5         default_q_type='kif',  
6         overflow_mode='WRAP'  
7     ),  
8     LayerConfigScope(enableEbops=True, beta0=1e-5)  
9 ):  
10 oq_conf = QuantizerConfig('kif', 'datalane', fr=fr, ir=ir)  
11 model = keras.Sequential([  
12     QDense(64, activation='relu'),  
13     QDense(32, activation='relu'),  
14     QDense(32, activation='relu'),  
15     QDense(5, enable_oq, oq_conf=oq_conf),  
16 ])
```

High Granularity Quantization - User Guide

Quantizer Configuration

```
1 with (  
2   QuantizerConfigScope(place='all', overflow_mode='SAT_SYM'),  
3   QuantizerConfigScope(  
4     place='datalane',  
5     default_q_type='kif',  
6     overflow_mode='WRAP'  
7   ),  
8   LayerConfigScope(enable_ebops=True, beta0=1e-5)  
9 ):  
10  oq_conf = QuantizerConfig('kif', 'datalane', fr=fr, ir=ir)  
11  model = keras.Sequential([  
12    QDense(64, activation='relu'),  
13    QDense(32, activation='relu'),  
14    QDense(32, activation='relu'),  
15    QDense(5, enable_oq, oq_conf=oq_conf),  
16  ])
```

Not found
(except overflow_mode)

Match

Still Unfound

Unfound (except ir and fr)

Match

For kernel/bias:
Not match

High Granularity Quantization - User Guide

Quantizer Configuration

Please refer to the docstring embedded for detailed explanation over the parameters.

In the scope variable, one can also configure the **initial bitwidths** with `k0, i0, f0, b0`. In the **configuration scope**, the initial bitwidths is insensitive to `kbi` or `kif` parametrization -- setting `i0=2, f0=2` will automatically set `b0=4`, and vice versa. (not true when explicitly defining with `QuantizerConfig`.)

Another set of attributes need attention is `heterogeneous_axis` and `homogeneous_axis`. **Only one of these can be set to a tuple at a time**, and the other is automatically derived based on the rank of the tensor to be quantized. Only over the axis defined in the `heterogeneous_axis`, quantization bit-width will be different for each element. If none is set, maximum granularity will be used (all weights, skip batch dimension for activations).

Model Definition

As one may have noticed, `iq: input quantizer` and `oq: output quantizer` can also be enabled or disabled for each layer, or controlled by a scope (no selection though). By default, `iq` is enabled, and `oq` is disabled.

`beta`, the parameter controls the trade-off between quantization error and model accuracy, can also be defined in the scope. However, it is recommended to use a `BetaScheduler` from `hgq.utils.sugar` instead to schedule it dynamically during training.

Model Definition


In general, use layers from `hgq.layers` for model definition **as much as possible**. When using external layers, check:

- Is this layer's operation trivial (e.g., simple element arrangement)
- Is there any multiplication involved? If so, stop, and
 - Consider if you can find a quantized version, or construct a quantized equivalent with existing quantized building blocks - even MHA can be constructed this way!
 - Proceeding with layers with highly non-trivial operations will likely to **break automatic conversion both in `da4ml` and `hls4ml`**

High Granularity Quantization - User Guide

Bonus: Emulate QKeras

If, for any reason, one want to replicate `QKeras` behavior:

-  `auto_pow2` behavior is different: `QKeras auto_pow2` returns a floating point, and we found this behavior not really useful on FPGAs.

To emulate a `QKeras` model:

- Turn on output quantizer, and turn off input quantizer for `Q-` layers.
- If `alpha=1`:
 - Set `trainable=False, overflow_mode='SAT', round_mode='RND_CONV'`
 - Set desired bitwidths with `k0, i0, f0`.
- To emulate "correct" `auto_pow2` behavior:
 - Set all quantizer type to `kbi`
 - Set `k0` to `signed`, `bc` to `hgq.constraints.MinMax(b, b)`
 - Set `overflow_mode='WRAP'` and `round_mode='RND_CONV'` for all quantizers.

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 GPU utilization**:

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.

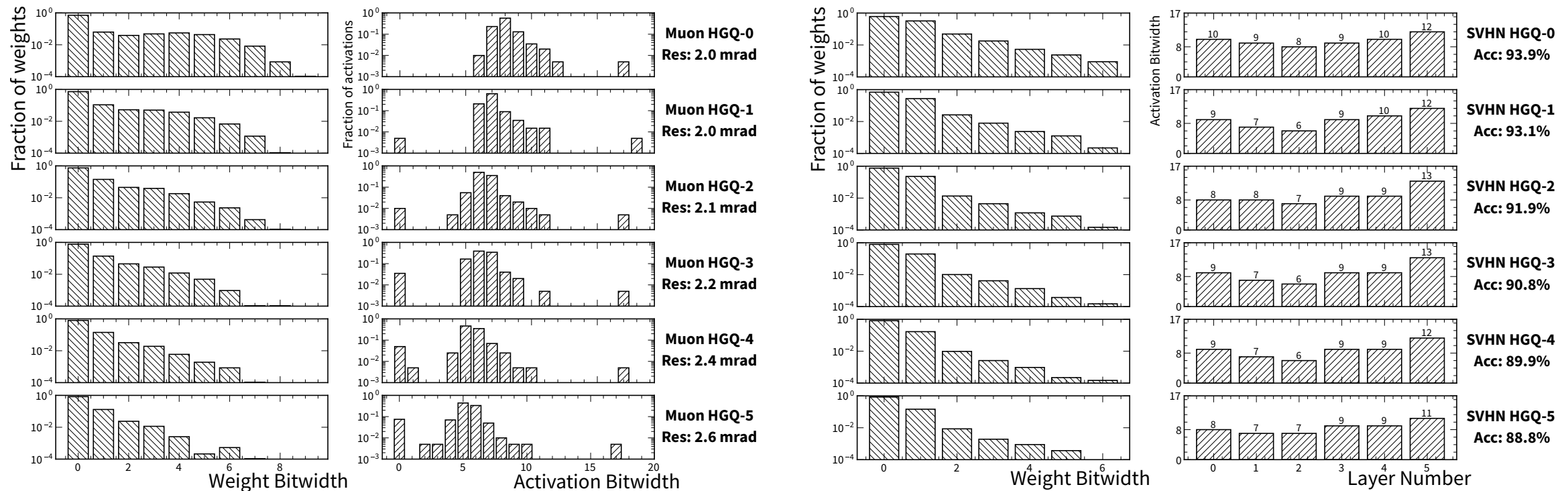
Full Examples

https://github.com/calad0i/HGQ2/blob/master/example/small_jet_tagger.ipynb

<https://github.com/calad0i/JEDI-linear>

High Granularity Quantization - Resultant Bitwidths

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):

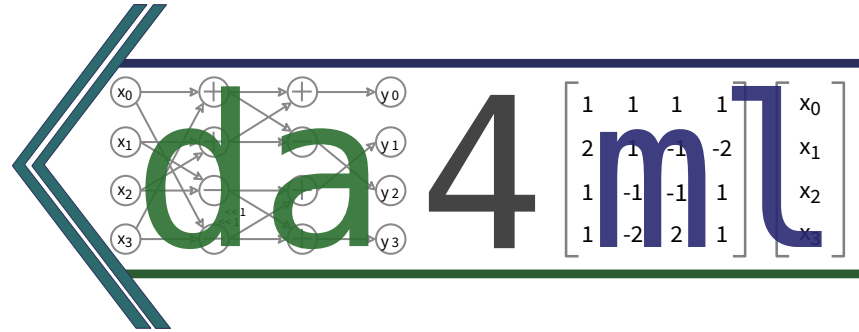


What is Distributed Arithmetic?

In this context, we refer to the method of implementing the CMVM operation as adder graph a form of distributed arithmetic (DA). As shown before, when the number of signed digits ≤ 2 for the weights, Vitis HLS automatically implements DA.

However, the implementation there is not optimal: with $ll=1$, more optimization can be performed as the computation graph constructed naively contains many redundant operations.

da4ml: Distributed Arithmetic for Machine Learning



da4ml [[code](#), [doc](#), [paper](#)] is a library for implementing distributed arithmetic (DA) based algorithms on FPGAs. It is an alternative to `hls4ml` for most of the `II=1` use cases, or it can also function as a plugin for `hls4ml`. It has two major components:

- A fast and performant constant-matrix-vector multiplications (CMVM) optimizer to implement them as efficient adder graphs, with Common sub-expressions elimination (CSE) and graph-based pre-optimization to reduce the firmware footprint.
- Low-level symbolic tracing utility with code generator for combinational/fully pipelined logics in HDL or HLS code.

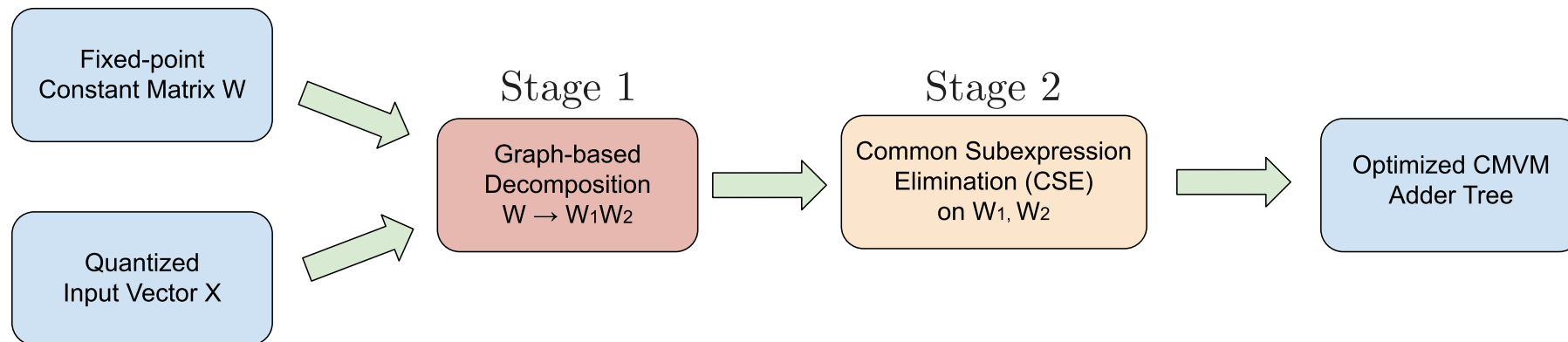
da4ml: Distributed Arithmetic for Machine Learning

CMVM Optimization

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



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:

$$\begin{aligned} 7 &= \{ 0, 1, 1, 1 \} = 4 + 2 + 1 \\ &= \{ 1, 0, -1, 1 \} = 8 - 2 + 1 \\ &= \{ 1, -1, 1, 1 \} = 8 - 6 + 4 + 1 \\ &= \{ 1, 0, 0, -1 \} = 8 - 1 // \text{ This is CSD} \end{aligned}$$

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

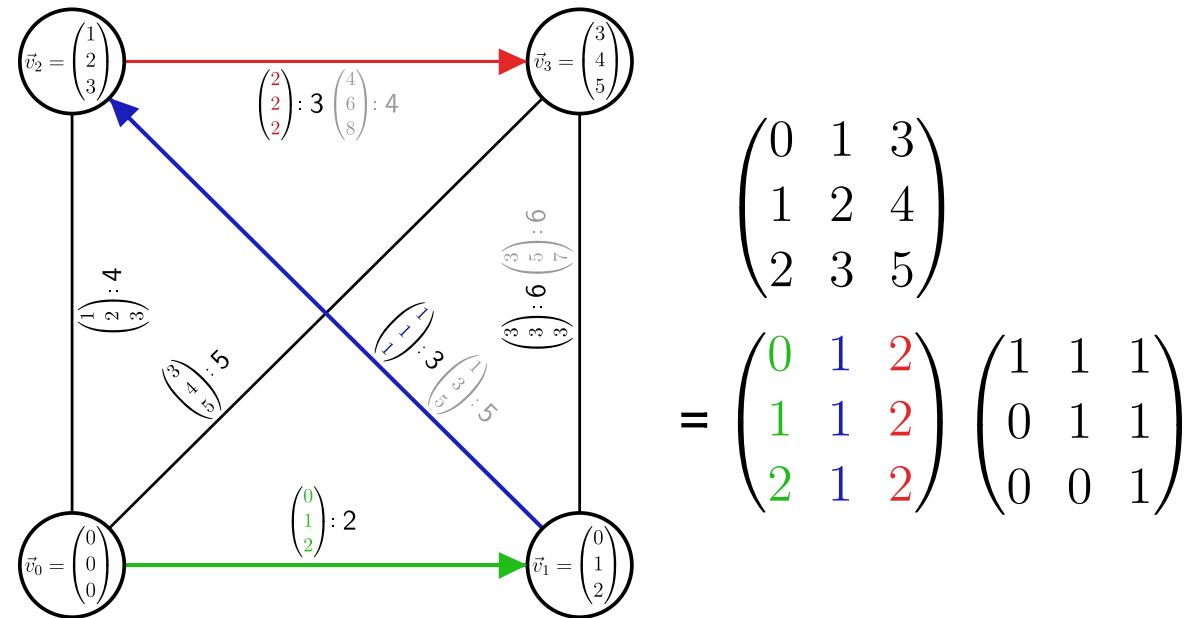
da4ml: Distributed Arithmetic for Machine Learning

CMVM Optimization - Stage 1

The first stage aims to exploit high-level structural similarities between the columns of the constant matrix W .

Three steps are involved:

- Graph Construction:** A graph is constructed where each column vector of the matrix are the vertices.
 - Distance $(v_1, v_2) = \min(\# \text{CSD digits in } v_1 \pm v_2)$
- Minimum Spanning Tree:** An MST is constructed for this graph with Prim's algorithm. This tree connects the columns in a way that minimizes the "cost" of transforming one column from the others.
- Matrix Decomposition:** The original matrix W is then decomposed into the product of two matrices, $W = W_1 W_2$, following the MST constructed.



da4ml: Distributed Arithmetic for Machine Learning

Stage 2: Common Subexpression Elimination (CSE)

The second stage applies a greedy Common Subexpression Elimination (CSE) algorithm to the decomposed matrices W_1 and W_2 independently.

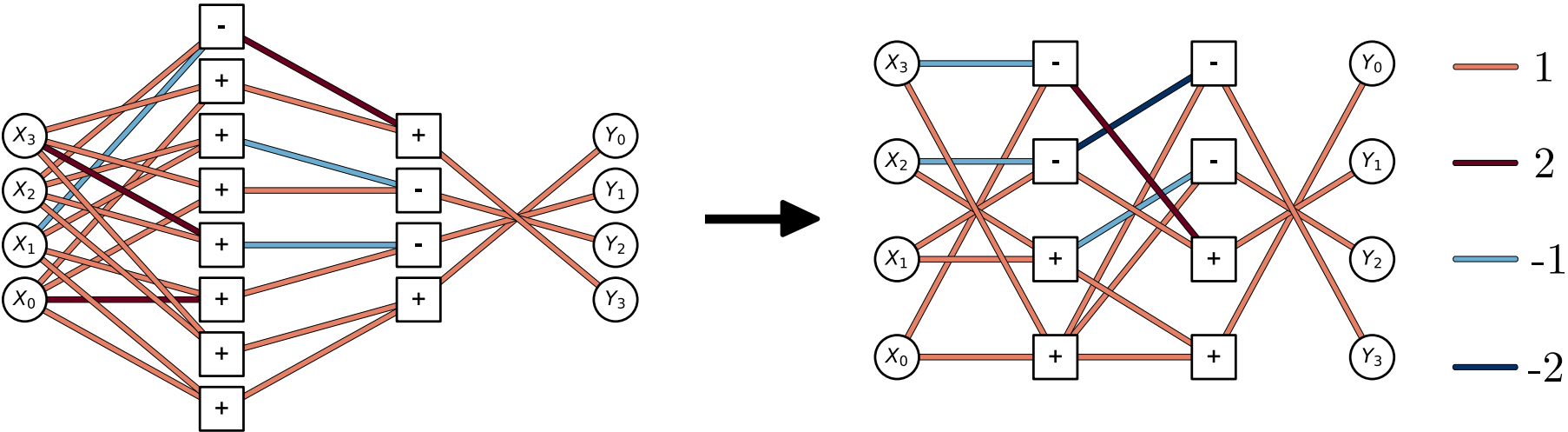
- Convert to CSD:** The matrix elements are first converted into the CSD representation.
- Greedy subexpression finding:** iteratively identifies the most frequently occurring two-term subexpression of the form $a \pm (b \ll s)$
- Substitution:** Implement the found subexpression as a single operation, replace all its occurrences.
- Bitwidth weighting:** The selection of the best subexpression to eliminate is weighted by the bitwidths of the operands.

$$\begin{aligned}
 \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & 1 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 2 & 1 & -1 & -2 & 0 \\ 0 & -1 & -1 & 0 & 1 \\ 0 & -2 & 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_0+x_3 \end{bmatrix} \\
 &= \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 2 & 1 & -1 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & -2 & 2 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_0+x_3 \\ x_1+x_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 2 & -2 & 0 & 0 & 1 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 0 & -2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_3 \\ x_0+x_3 \\ x_1+x_2 \\ x_1-x_2 \end{bmatrix}
 \end{aligned}$$

da4ml: Distributed Arithmetic for Machine Learning

Example

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



da4ml: Distributed Arithmetic for Machine Learning

da4ml can be used mainly in two ways:

Standalone

- Direct, explicit manual symbolic tracing
- Automatic tracing of a HGQ2 trained model, if compatible
- `II=1` is required; not supporting general non-linear activations; higher FF usage

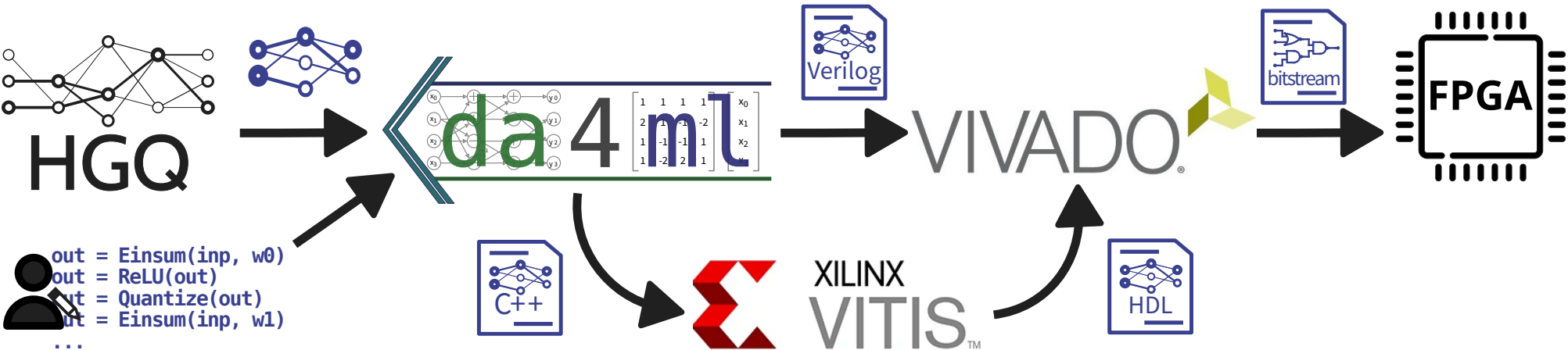
As hls4ml plugin

- Only offload the implementation of the CMVM operations from `hls4ml`, use `hls4ml` native template library for other operations.
- Significantly slower synthesis (HLS); less granular computation graph control/operators provided; sometimes higher LUT usage and worse timing

da4ml: Distributed Arithmetic for Machine Learning

Workflows - standalone

The framework generates standalone HDL/HLS project in this workflow.



Workflows - Direct symbolic tracing

da4ml provides a symbolic tensor with some built-in operators and overloaded `numpy` operations. Similar to the keras functional API, one can define a combinational logic by "replay" the required operation on the symbolic tensor

```
def operation(inp):
    inp = quantize(inp, 1, 7, 0) # Always quantize the input first!
    out1 = relu(inp)
    out2 = inp[:, 1:3].transpose()
    out2 = np.repeat(out2, 2, axis=0) * 3 + 4
    out2 = np.amax(np.stack([out2, -out2 * 2], axis=0), axis=0)
    out3 = quantize(out2 @ out1, 1, 10, 2)
    out = einsum('ijk,ij->ik', w, out3)
    return out

inp = FixedVariableArrayInput((4, 5))
out = operation(inp)
comb_logic = comb_trace(inp, out)
```

Workflows - Automatic HGQ2 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 = keras.Input((4, 5))
out = QEInsumDenseBatchnorm('bij,jk->bik', (4,6), bias_axes='k')(inp)
out1 = QMaxPool1D(pool_size=2)(out)
out = keras.ops.concatenate([out, out1], axis=1)
out1, out2 = out[:, :3], out[:, 3:]
out = keras.ops.einsum('bik,bjk->bij', out1, out1 - out2[:, :1])
model = keras.Model(inp, out)
inp, out = trace_model(model)

comb_logic = comb_trace(inp, out)
```

Standalone code generation

da4ml can generate fully pipeline/combinational `Verilog` code, or `HLS` code in Vitis HLS (`ap_types`) or `hlslibs` (`ac_types`, not tested) formats from a `comb_logic` traced.

For `verilator` code generation, verification by verilator is natively supported.

```
# can also be HLSModel
verilog_model = VerilogModel(comb_logic, 'vmodel', '/tmp/verilog', latency_cutoff=5)
verilog_model.write()

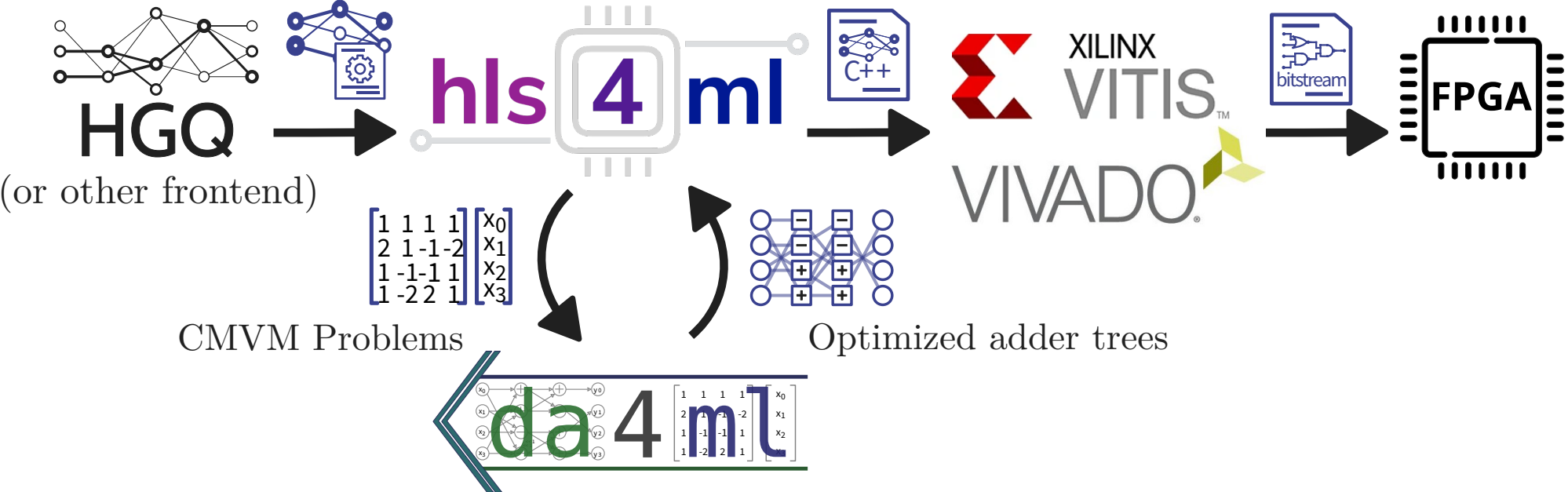
verilog_model._compile() # verilator compile and python bind
verilog_model.predict(data_inp) # run inference
```

Tip: The `VerilogModel/HLSModel` objects gives pretty accurate LUT (and FF) usage when instantiated (~10% for LUTs, 20% for FF). Make good use of them.

da4ml: Distributed Arithmetic for Machine Learning

Workflows - As hls4ml plugin

For existing hls4ml workflows, `da4ml` can be enabled by setting `strategy` to `distributed_arithmetic`.



Workflows - As hls4ml plugin

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.

We show the Performance gain by using HGQ2 and da4ml together on a few benchmark tasks.

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

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+da4ml (HLS)	76.9%	44.1	12,682	0	19,056	702.	1
HGQ+da4ml (RTL)	76.5%	23.1	6,165	0	7,207	736.	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
NeuraLUT-Assemble [4]	76.0%	2.1	1,780	0	540	940.	1
TreeLUT [5]	75.6%	2.7	2,234	0	347	735.	1

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+da4ml (RTL)	75.2%	37.8	8,703	0	10,008	503.	1
HGQ+da4ml (RTL)	75.0%	27.4	5,636	0	6,218	656.	1
NeuraLUT-Assemble [4]	75.0%	5.7	8,539	0	1,332	352.	1
NeuraLUT-Assemble [4]	75.0%	7.0	8,535	0	2,717	994.	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

hls4ml jet tagger dataset (particle based)

SOTA on tasks requiring ~100k LUTs.

Implementation	N	Accuracy	LUT	DSP	FF	Latency [ns]	F_max [MHz]	II [cc]
JL (HGQ+da/RTL)	32	79.0%	80. k	0	136. k	79	299.	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+da/RTL)	8	66.5%	79. k	0	136. k	73	303.	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

Some personal thoughts on designing NN for FPGAs

Static \gg dynamic dataflow. If there is any way the dataflow can be made dynamic without catastrophic performance loss, it should be explored.

- If the overhead is not OOM, sometimes naive solution like padding can excel dynamic control
- If dynamic dataflow is needed, find as much constraints as possible. The less freedom the dataflow has, the easier it is to optimize.

Aim for CMVM, not MVM. Multiplying with constants is much easier than with variables, especially in the HGQ+DA setting (low bw -> just one/zero **ADD/SUB in most cases)**