

da4ml: Distributed Arithmetic for Machine Learning

Chang Sun

Caltech, PMA

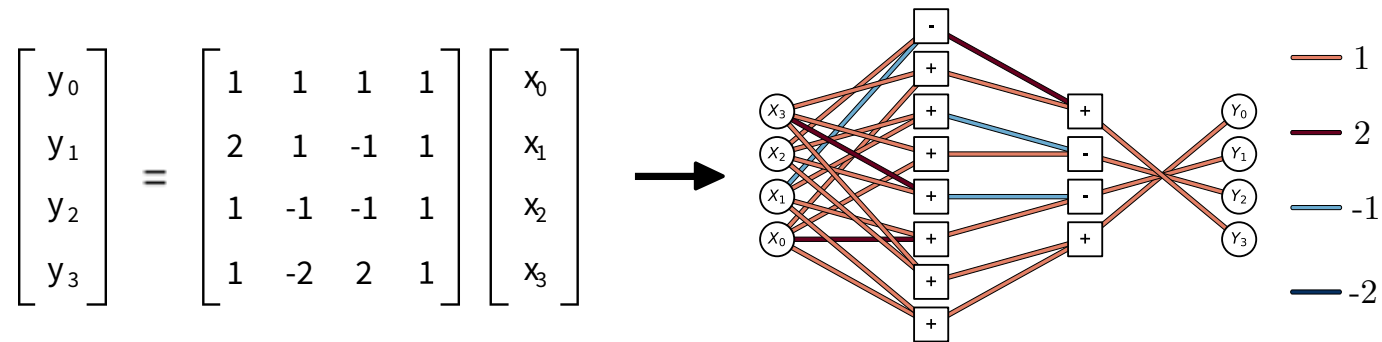
Distributed Arithmetic

What is Distributed Arithmetic (DA)?

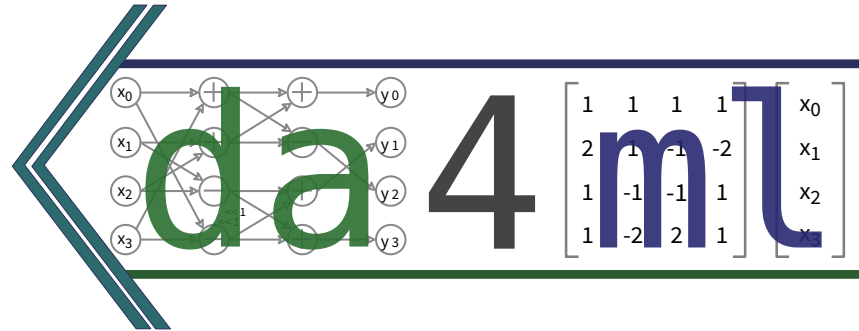
Implementing constant multiplication(-accumulation) as shift-add operations. In Fast ML applications, the operation in concern is **constant-matrix-vector multiplication (CMVM)** (i.e., in `Dense` , `Conv` , `EinsumDense` , `RNN` family, etc).

How is it useful?

When `II=1` is required for the individual CMVM operation, DA can be more efficient than the default HLS implementation.



da4ml: Distributed Arithmetic for Machine Learning



da4ml [[code](#), [doc](#), [paper](#)] is a library implementing neural networks on FPGAs with DA.

There are two main components:

- CMVM optimizer with graph-based pre-optimization and common sub-expressions elimination (CSE) to reduce the firmware footprint for the CMVM operation.
- Low-level symbolic tracing utility for generating combinational/fully pipelined network/sub-network in **HDL** or **HLS** code.

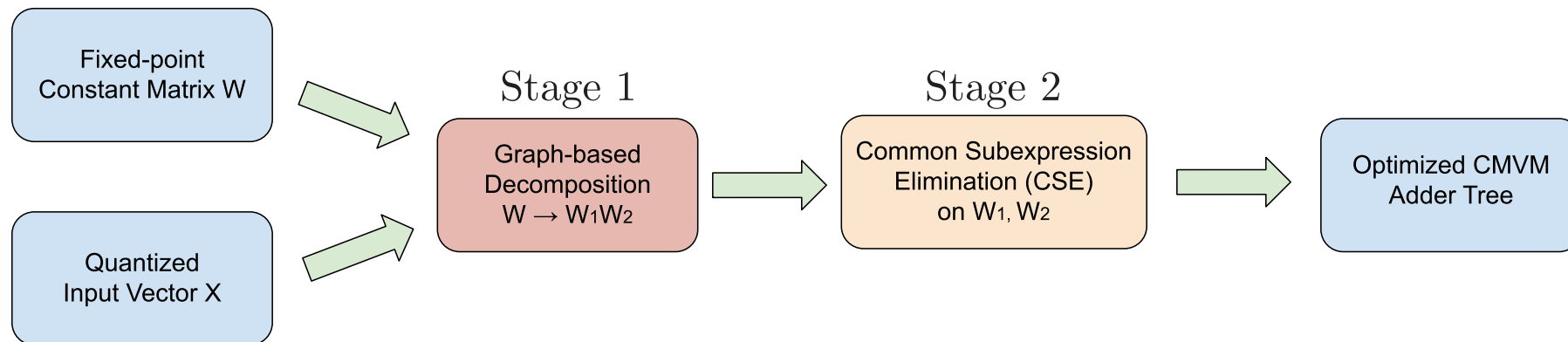
da4ml: Distributed Arithmetic for Machine Learning

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:

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

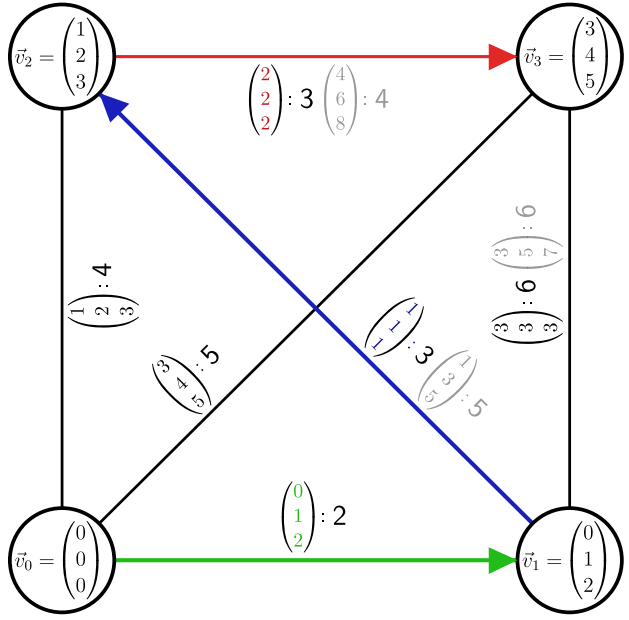
Canonical Signed Digit (CSD) is a unique representation of the number such that:

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

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:

1. **Graph Construction:** A graph is constructed where each column vector of the matrix are the vertices.
 - o Distance $(v_1, v_2) = \min(\# \text{CSD digits in } v_1 \pm v_2)$
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.



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

da4ml: Distributed Arithmetic for Machine Learning

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 $a \pm (b \ll s)$
3. **Substitution:** Implement the found subexpression as a single operation, replace all its occurrences

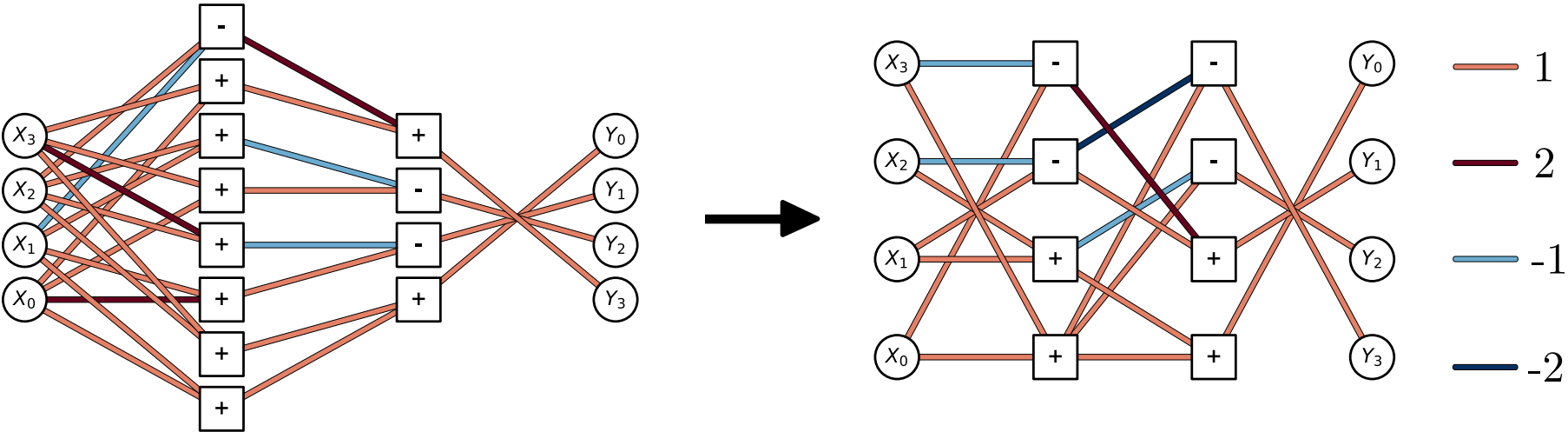
The selection of the most common subexpression in step 2 is weighted by the bitwidths of the two operands in practice

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

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

Distributed Arithmetic Instruction Set (DAIS)

DAIS is the internal representation used by da4ml: a RISC-like, assembly-style DSL that is SSA. Only ~10 instructions are used to perform 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.

DAIS Program Binary Representation

Header

<code>n_in, n_out</code>	<code>int32[2]</code>
<code>len(ops)</code>	<code>int32</code>
<code>inp_shift</code>	<code>int32[shape[0]]</code>
<code>out_idxs</code>	<code>int32[shape[1]]</code>
<code>out_shifts</code>	<code>int32[shape[1]]</code>
<code>out_negs</code>	<code>int32[shape[1]]</code>

ops: Op[`len(ops)`]

Op[0]

<code>opcode</code>	<code>int32</code>
<code>id0</code>	<code>int32</code>
<code>id1</code>	<code>int32</code>
<code>data_higher</code>	<code>int32</code>
<code>data_lower</code>	<code>int32</code>
<code>dtype</code>	<code>int32[3]</code>
<small>(signed, integer_bits, fractional_bits)</small>	

...
Op[1]
...

Header size: $12 + 4 \times n_{in} + 4 \times n_{out}$ bytes
Operator size: $\text{len(ops)} \times 8$ bytes

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

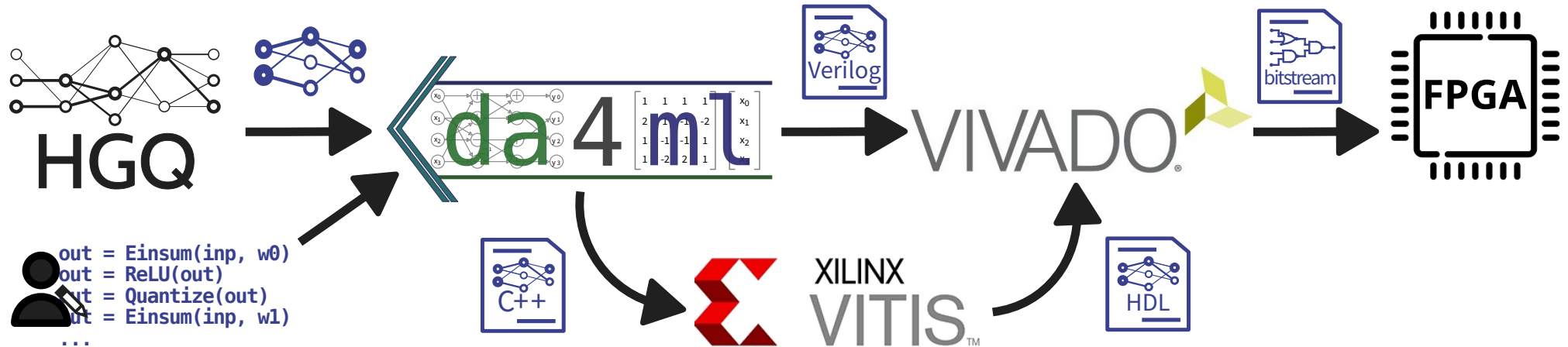
As hls4ml plugin

- 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()

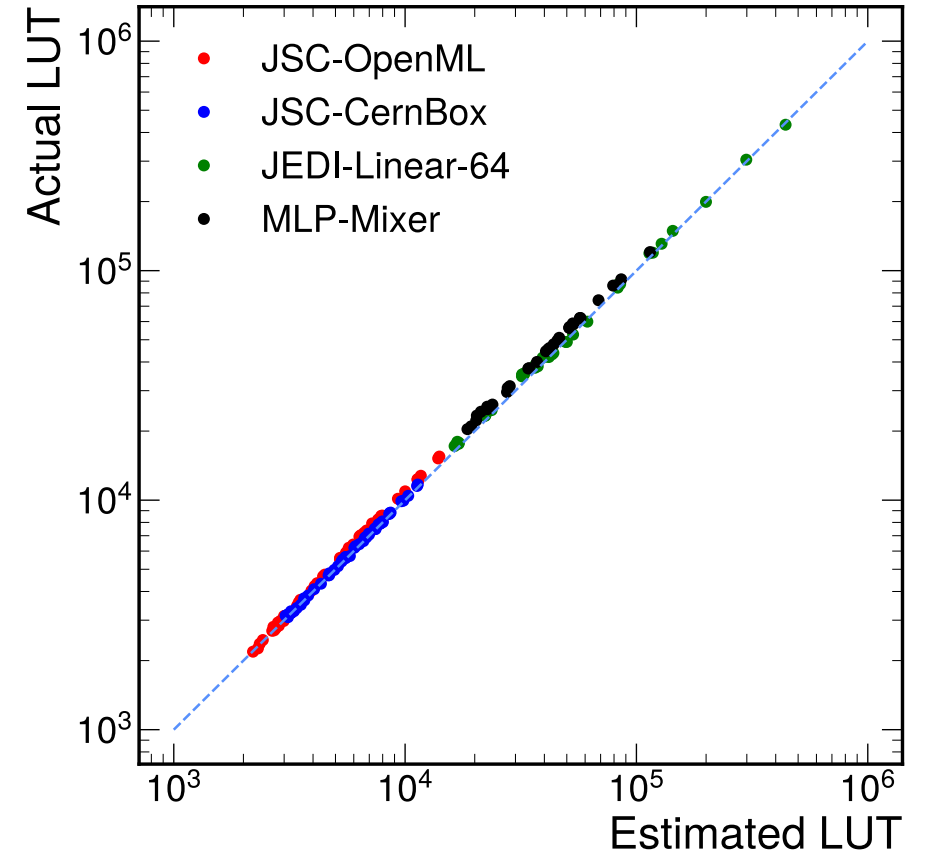
verilog_model._compile() # verilator compile and python bind
verilog_model.predict(data_inp) # run inference
```

Standalone code generation

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

```
> print(verilog_model)

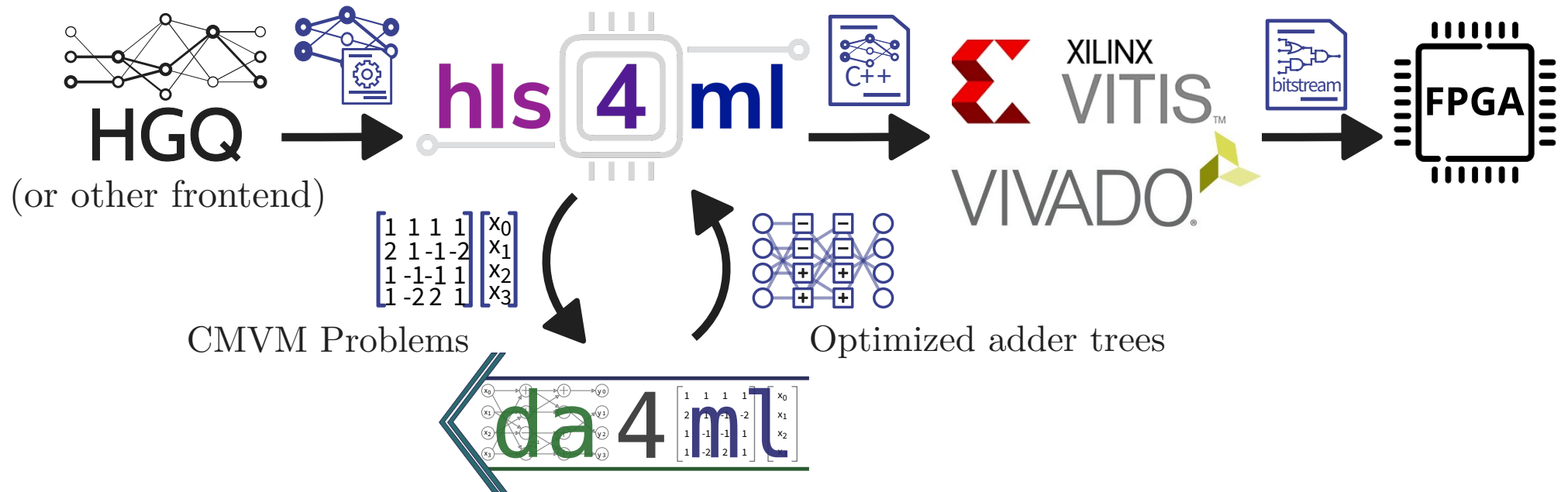
Top Function: hlsmodel
=====
20 (160 bits) -> 24 (486 bits)
combinational @ delay=(21.0, 22.0)
Estimated cost: 1551 LUTs
Emulator is compiled with OpenMP (d9f5cd8e6c99)
```



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.

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

$N \times N \times n\text{bit matrix} \times 8\text{ bits input vector}$

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

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)

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+hls4ml+DA	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