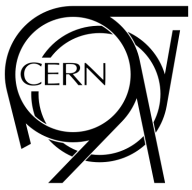
Large and compressed Convolutional Neural Networks on FPGAs with **hls4ml**

Vladimir Lončar
For the FastML team
fastmachinelearning.org

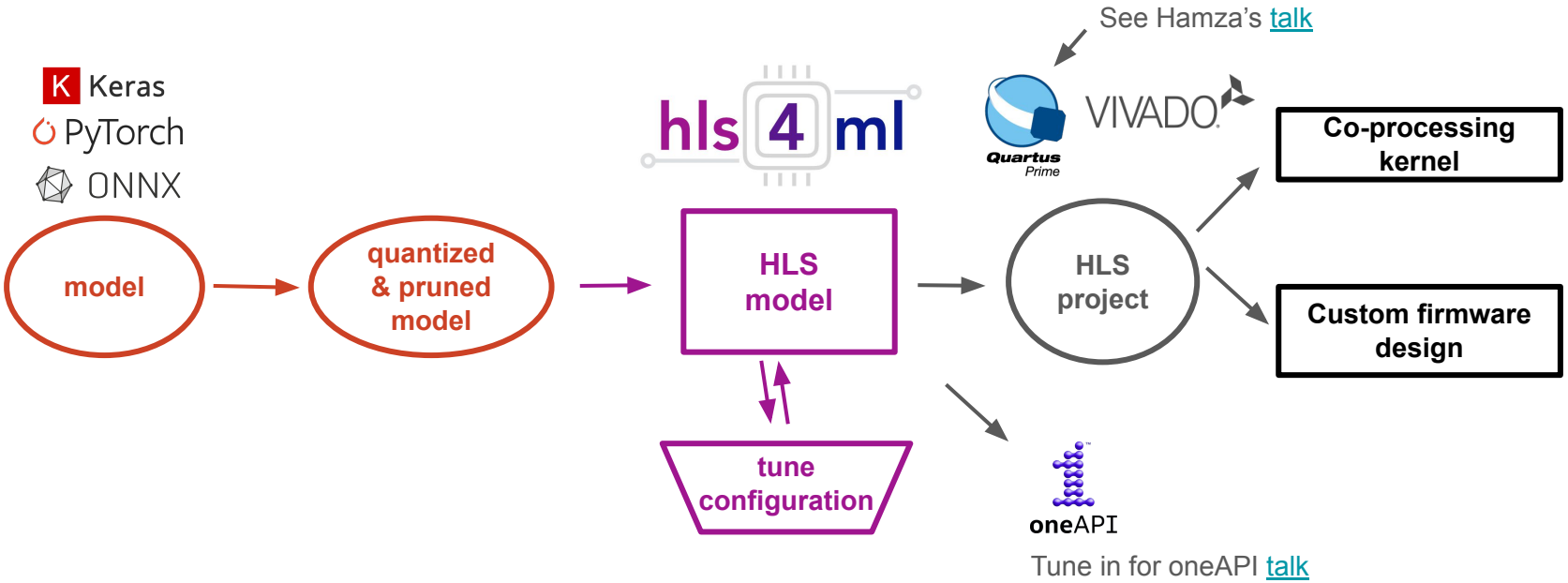


30.11.20
Southern Methodist University

high level synthesis for machine learning

User-friendly tool to automatically build and optimize DL models for FPGAs:

- Reads as input models trained with standard DL libraries
- Comes with implementation of common ingredients (layers, activation functions …)



hls4ml features

On-chip weights
- Much faster access times

User controllable trade-off between resource usage and latency/throughput
- Tuned via `ReuseFactor` and `Strategy`

Supported architectures:
- **DNNs**
  - Zero-suppressed weights
  - Binary/Ternary layers (computation without using DSPs) - arxiv:2003.06308
  - QKeras integration - arxiv:2006.10159
- **CNNs**
  - Convolution and pooling layers NEW
  - Implemented through streams for larger layers
- **Graph NNs**
  - GarNet architecture - arxiv:2008.03601

Challenges of implementing convolution in HLS

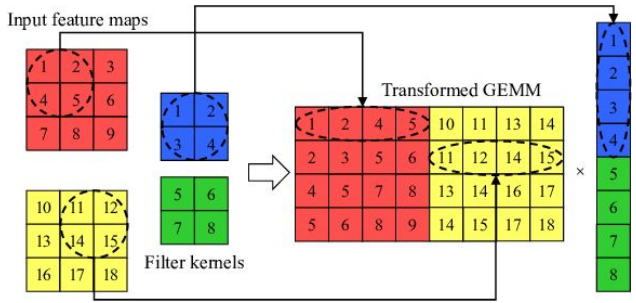
Direct convolution algorithm

- 6 nested loops
- Long latency due to switching context between loops
- Unrolling loops quickly hits the limit

```
for out_c in range(n_filt):
  for i in range(height):
    for j in range(width):
      for c in range(n_chan):
        for fi in range(k_height):
          for fj in range(k_width):
            elem = input[i+fi, j+fj, c]
            w = weights[fi, fj, c, out_c]
            output[i, j, out_c] += elem * w
```

GEMM-based approach

- Build an input matrix and multiply it with weights (*im2col*)
- HxWxC tensor becomes very large as we add filters
  - Hits the `complete` partitioning limit easily
- Partition the array (block or cyclic) into smaller arrays
- Difficult to match access pattern of activation tensors
- block/cyclic partitioning often requires use of % operator
  - Introduces UREM IP cores, ~10 cycles penalty



Implementation using streams

HLS streams

- Represented by `hls::stream<T>` class
- Both blocking and non-blocking `read()` and `write()`

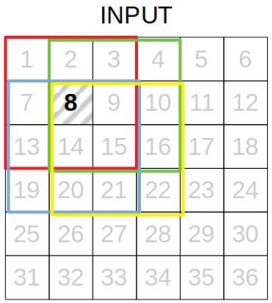
Requires reengineering of all algorithms to read/write sequentially

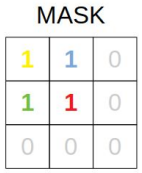
- Not possible to use direct or im2col algorithms

Sequential approach: Collect data from input pixels until we can compute one output

- Store pixel contributions in an array (need to store K x oW (x C) elements)
- Compute the value of output pixel with a single call to matrix-vector multiplication
  - We can reuse existing matrix-vector multiplication used in Dense layer
- Instead of bookkeeping logic, encode instructions into array

Encoding instruction array





For each input pixel, compute the position in sliding window

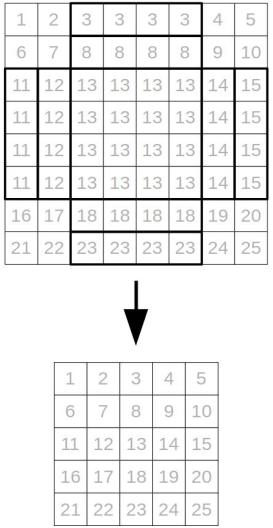
- We can make the instruction array binary

Create a (binary) mask of contributions

- E.g., `{ 1, 1, 0, 1, 1, 0, 0, 0, 0 }`
- Compute output if `mask[k*k-1] = 1`
- Store as K x K array as `ap_uint<K*K>`

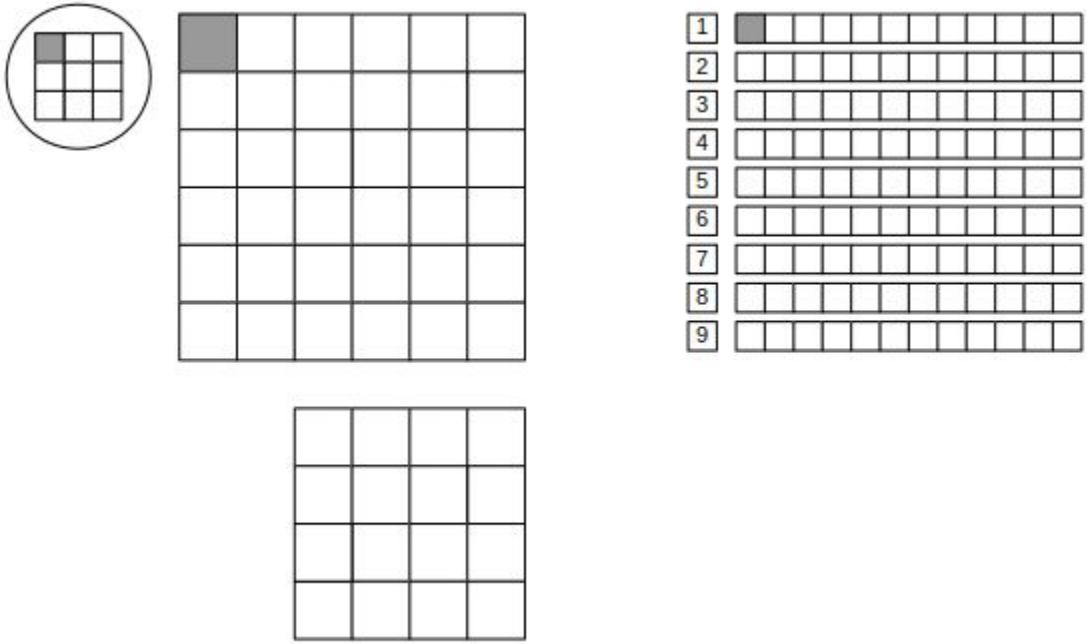
Most of the pixels have the same mask!

- We need to compress the array
- Find the smallest possible image that exhibits the same access pattern
  - E.g., 3x3 kernel with unit stride will have 25 unique patterns
  - We can encode every 3x3 convolution with unit stride using only 25 instructions



Description of Conv2D algorithm

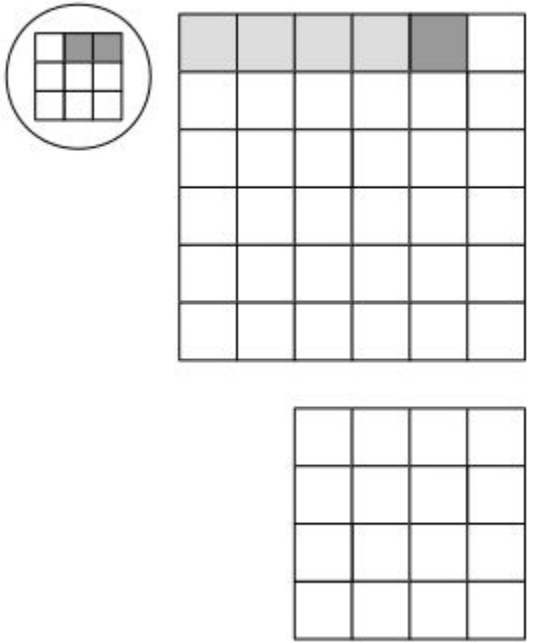
Save pixel contributions based on mask

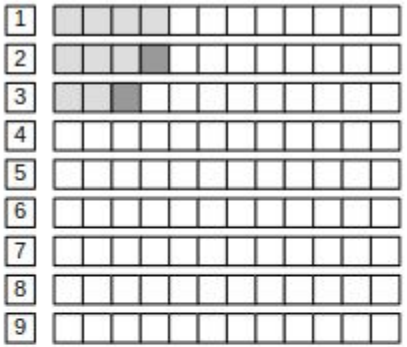


Description of Conv2D algorithm

Save pixel contributions based on mask

- Pixels will have different mask based on their position



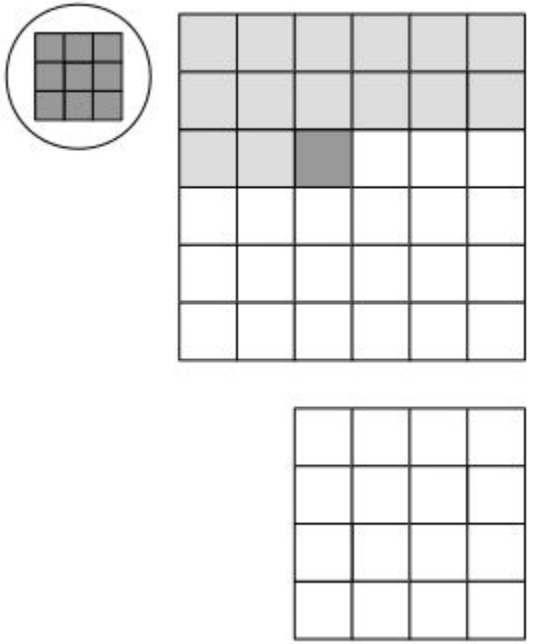


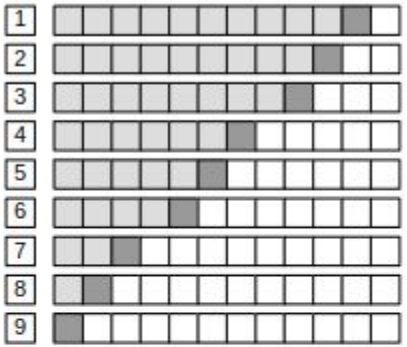
Description of Conv2D algorithm

Save pixel contributions based on mask

- Pixels will have different mask based on their position

If `mask[k*k-1] = 1` we can compute one output pixel





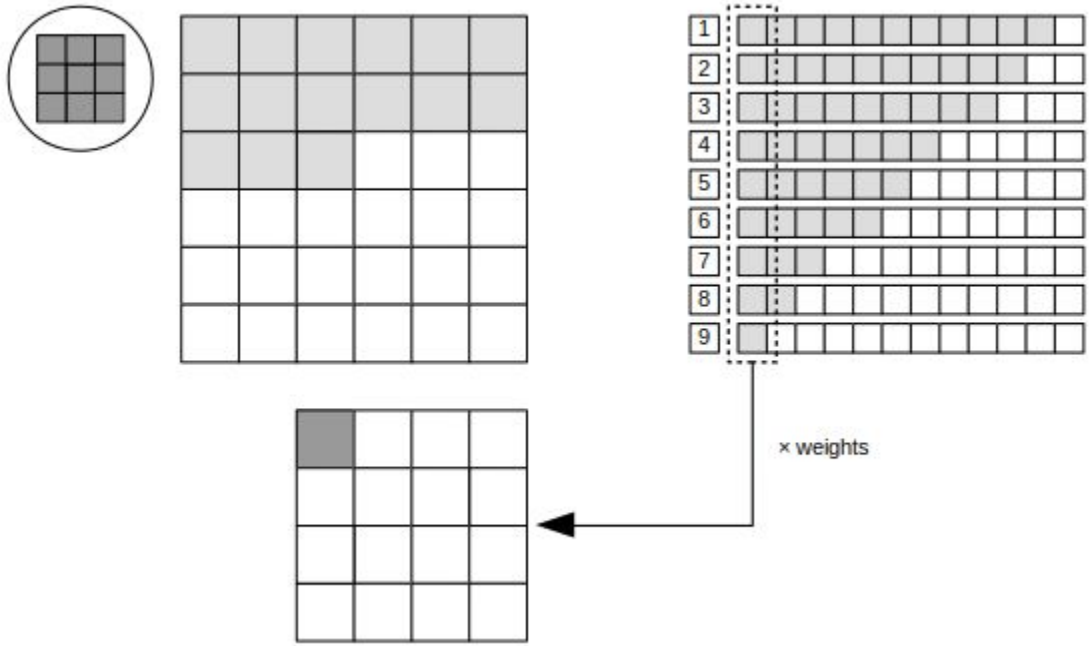
Description of Conv2D algorithm

Save pixel contributions based on mask

- Pixels will have different mask based on their position

If `mask[k*k-1] = 1` we can compute one output pixel

- We use existing matrix-vector multiplication function
  - Gives us support for `ReuseFactor` and `Strategy`
- Repeat until done



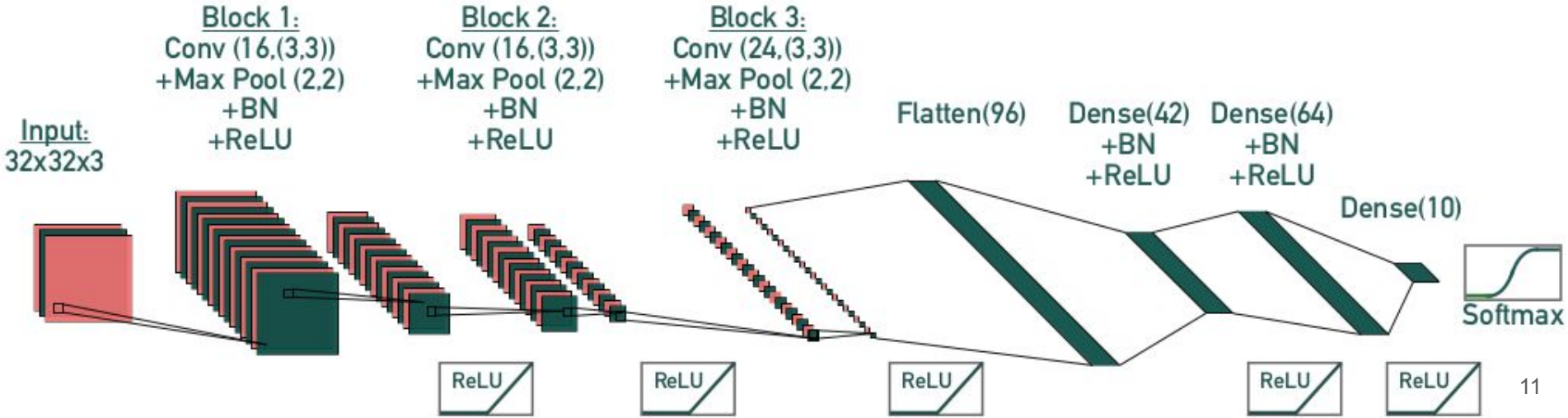
Like this animation? Check out the full version here

Performance evaluation

Street-view house numbers dataset (SVHN)

- 32x32x3 images

Model architecture:



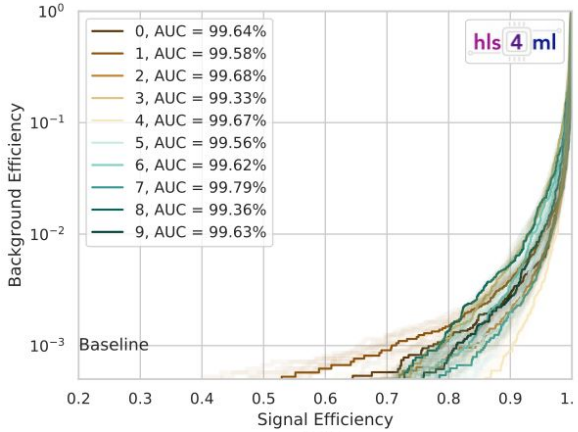
Model performance

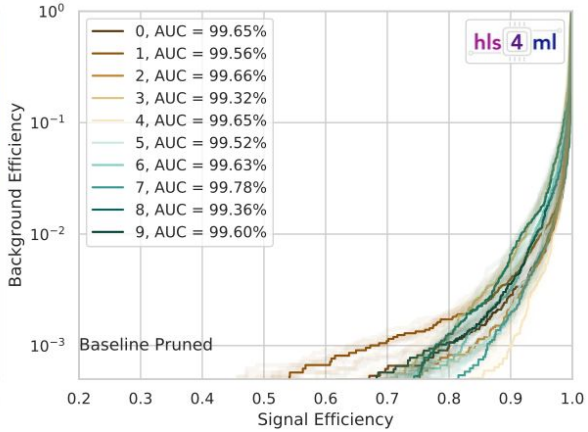
Baseline models

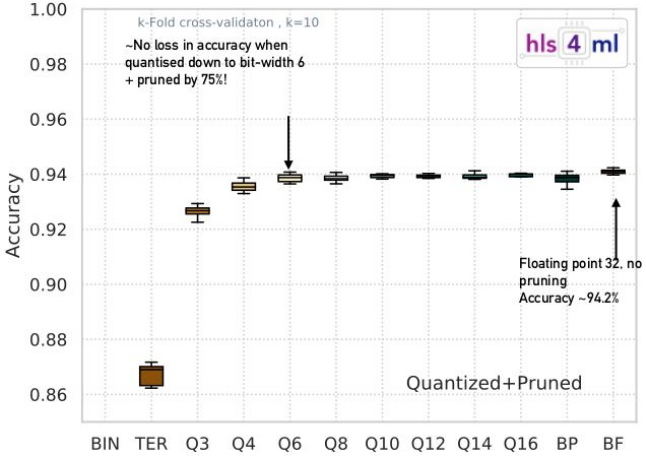
- Full precision (32-bit float)
- Full precision, pruned
  - 75% sparsity
  - Polynomial decay

QKeras models

- Binary (1-bit), Ternary (2-bit), quantized to 3-16 bits
- Pruned
  - 75% sparsity



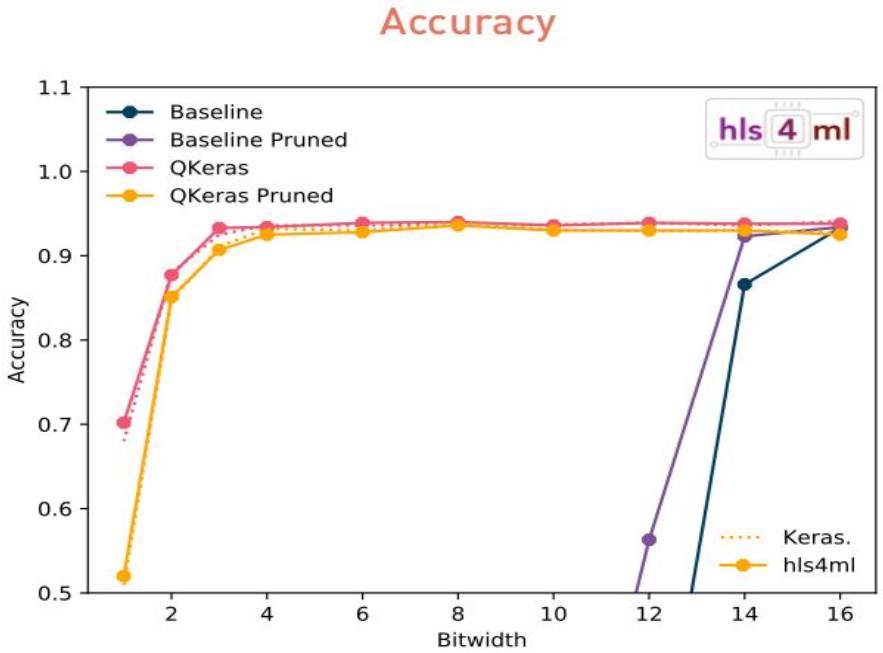


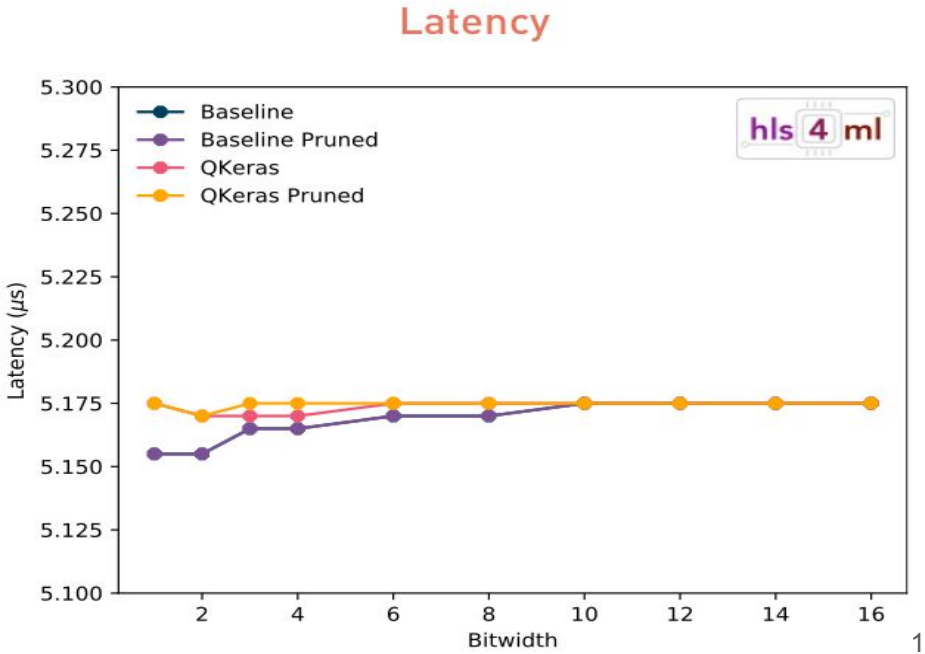


For more hls4ml + QKeras goodness, don't miss Thea's talk!

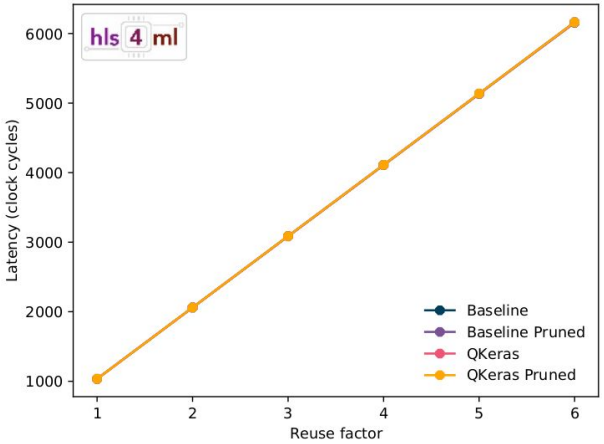
Performance in hls4ml

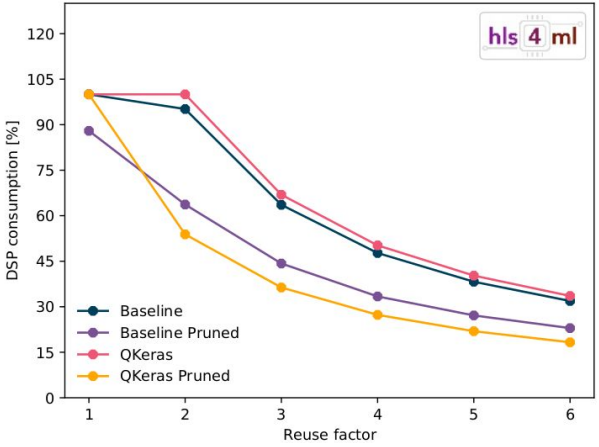
Target device: Xilinx VU9P family, 5ns clock cycle

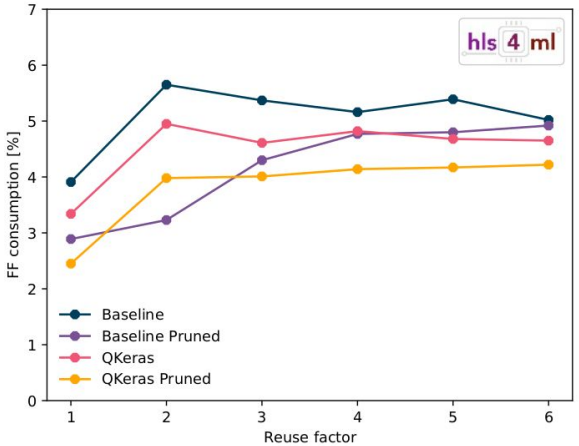


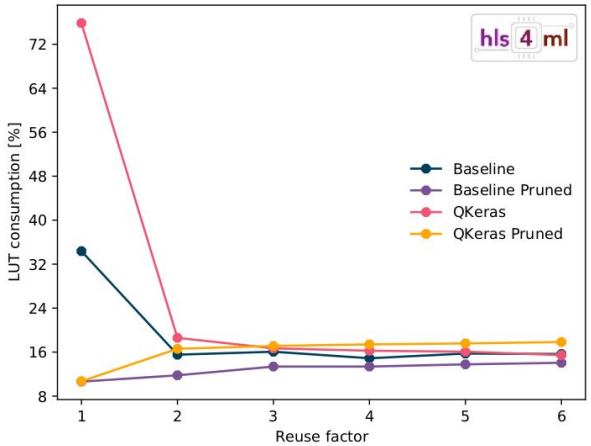


Reuse scan - 16 bits









Summary

New stream based implementation of convolutional layers

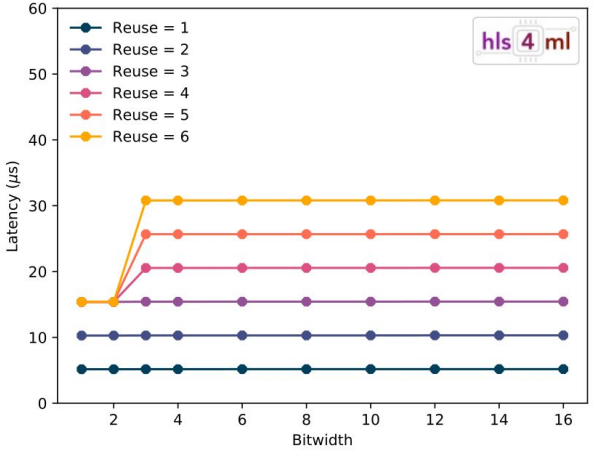
- Many supported layers
- Well integrated will existing **hls4ml** turning knobs
- With pruning using Tensorflow model optimization toolkit (TFMOT) and quantization using QKeras can reach O(μs) latency
- Soon to be in hls4ml master branch, currently available as a pull request

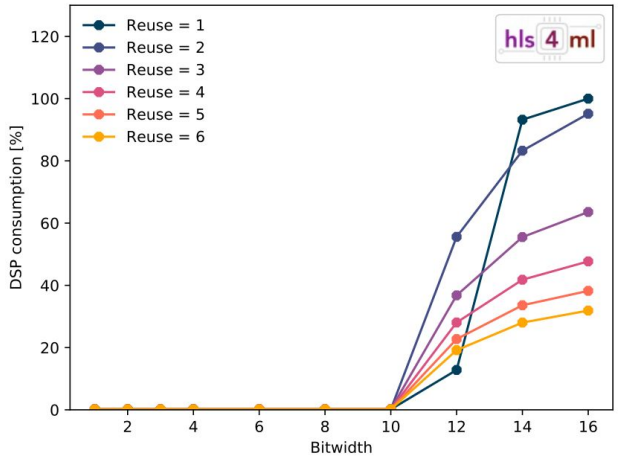
Still to explore:

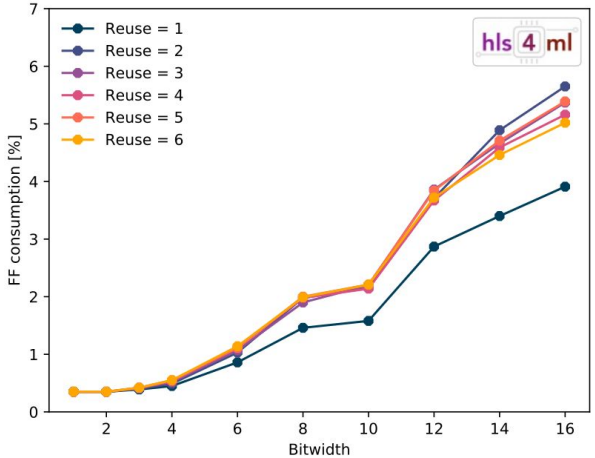
- Tiling can speed up very small models to sub-microsecond latency or scale large models to multiple FPGAs

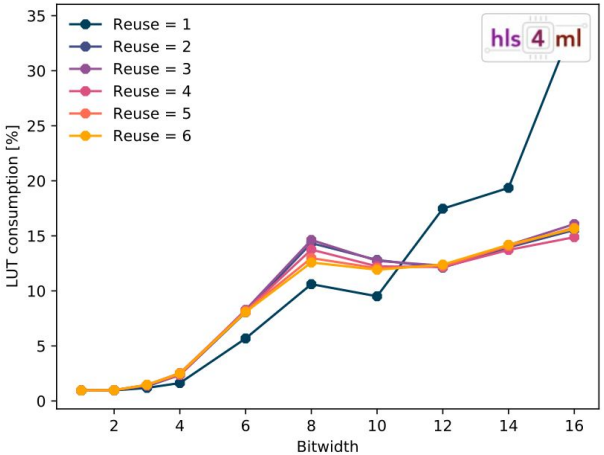
Extras

Reuse/bitwidth scan - Baseline model

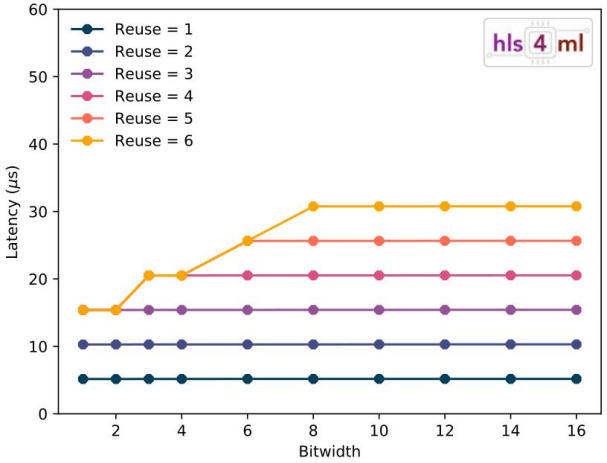


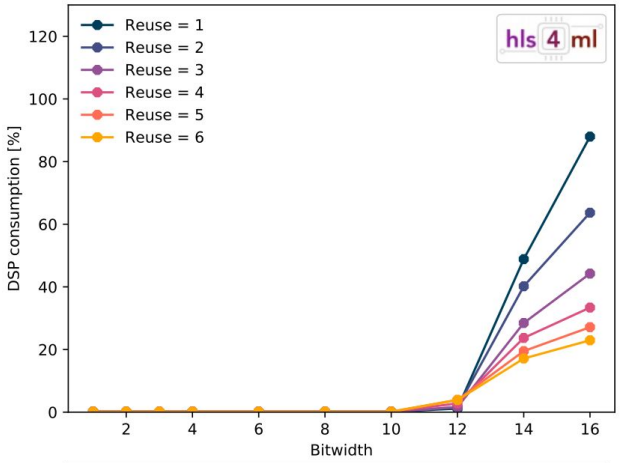


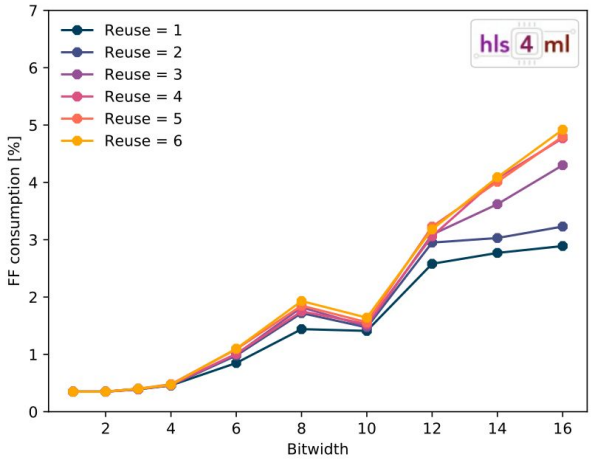


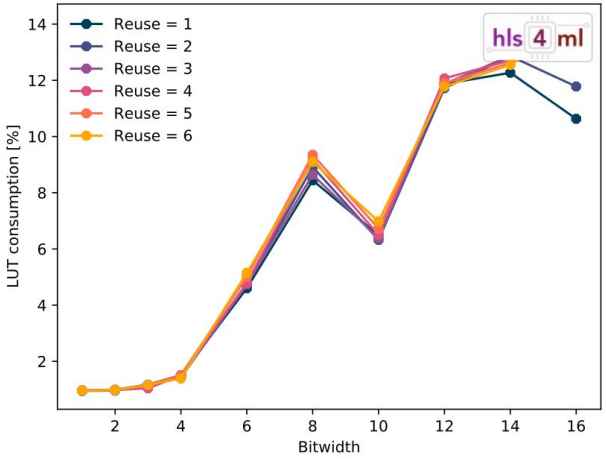


Reuse/bitwidth scan - Baseline, pruned model

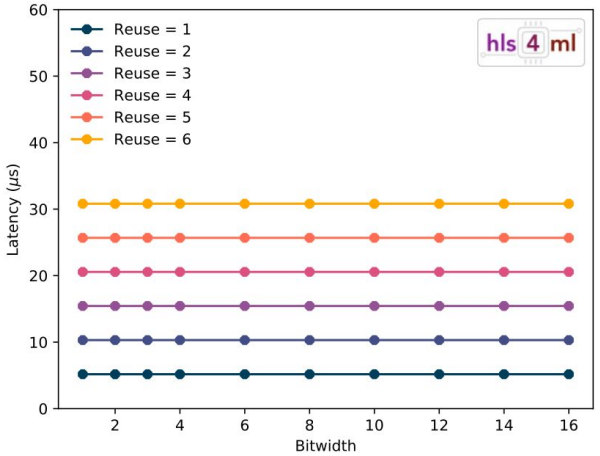


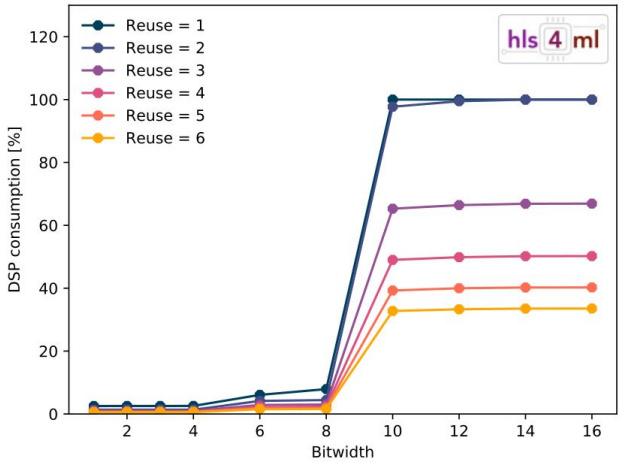


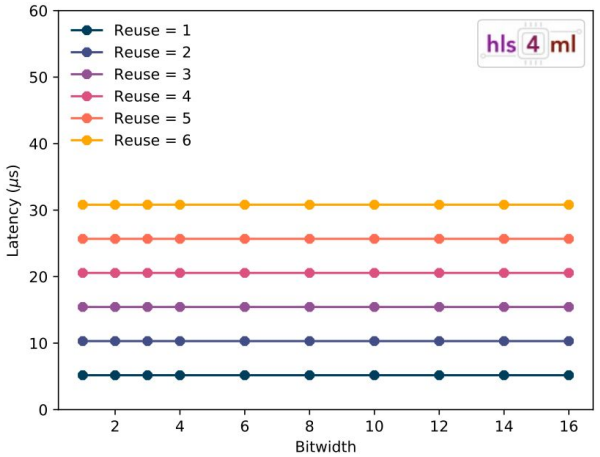


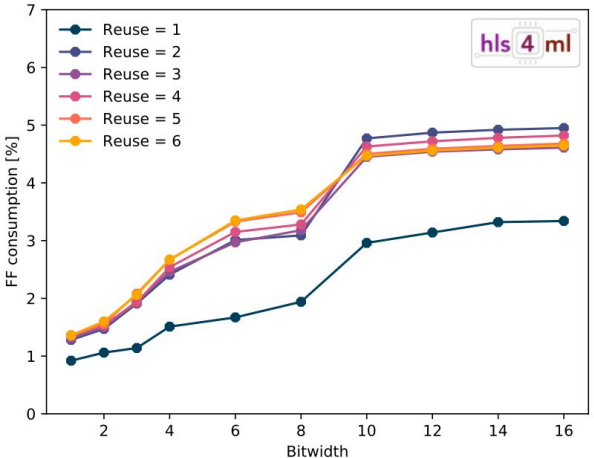


Reuse/bitwidth scan - QKeras model









Reuse scan - QKeras, pruned model