SparsePixels: Efficient Convolution for Sparse Data on FPGAs

Ho Fung Tsoi [Penn]

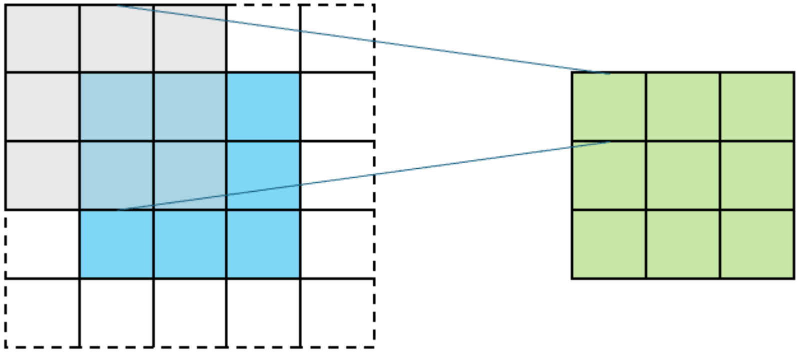
joint work with:

Dylan Rankin [Penn], Vladimir Loncar [CERN], Philip Harris [MIT]

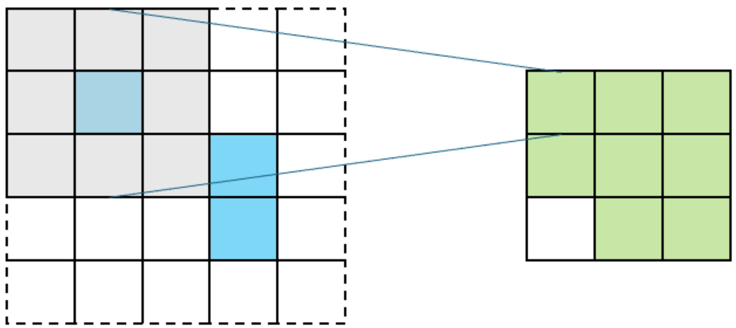
Coordinating Panel on Advanced Detectors (CPAD 2025) at Penn, Oct 7-10

Overview

- Problem statement
- SparsePixels framework
  - Algorithms, HLS implementation
  - FPGA latency/resource scaling
- Experiments

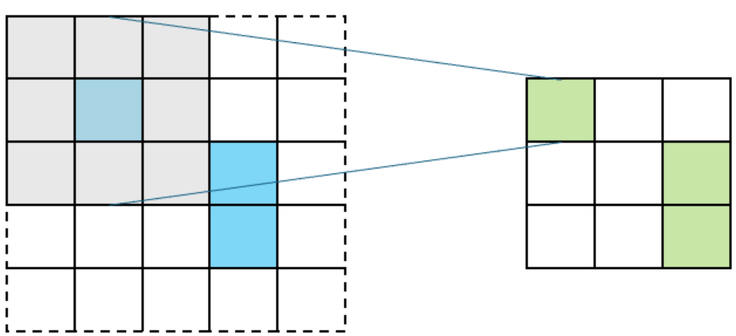


Standard CNN



Standard CNN on sparse input

- Do computation on all pixels no matter what



Sparse CNN on sparse input

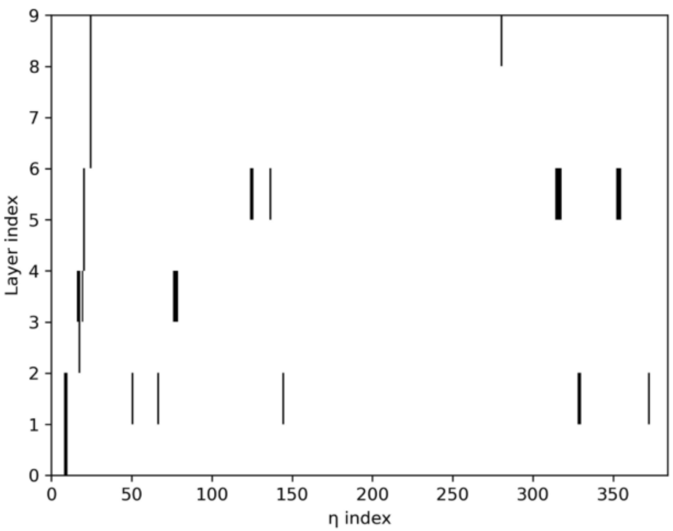
- Do computation on active pixels only

- Here, 3 out of 9 pixels are active
- What if there are 3000 input pixels and only 30 are active

What if the inputs become extremely sparse, say <1% of input pixels are active?

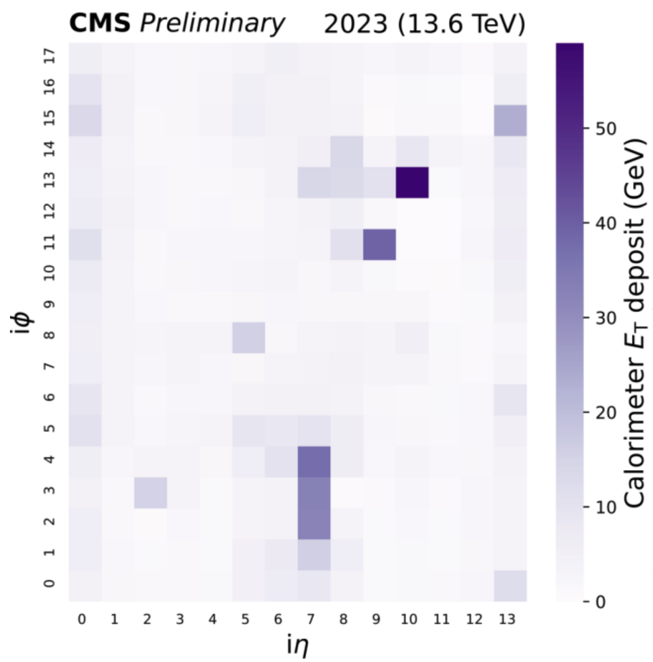
➢ Most computations are wasted on inactive pixels if using standard CNNs

Muon tracking



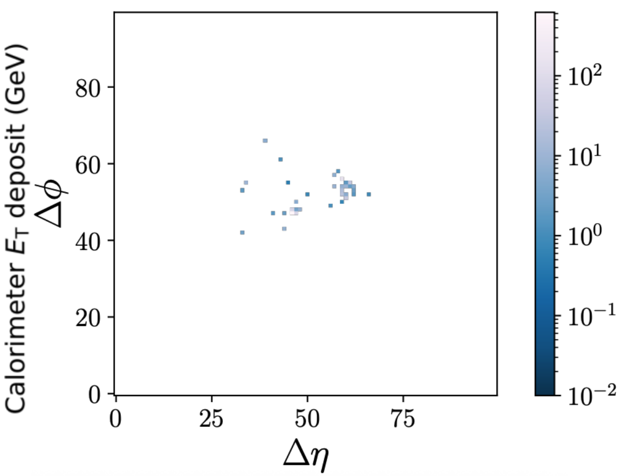
[Comput.Softw.Big Sci. 7 (2023) 1, 8]

Calorimeter image

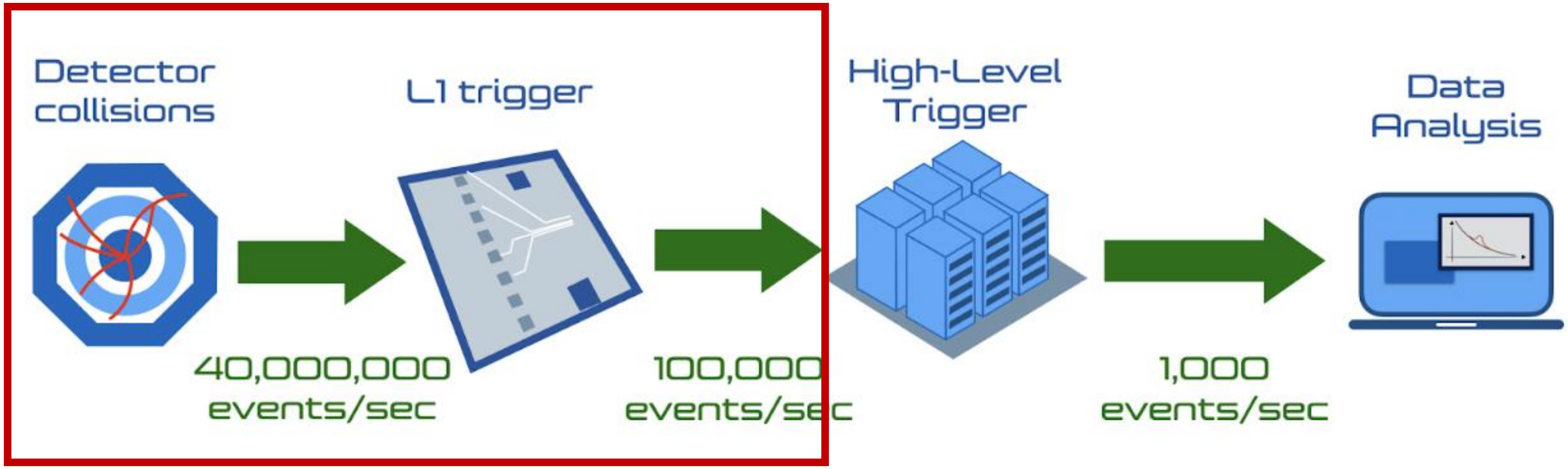


[CERN-CMS-DP-2023-086]

Jet image



[Eur.Phys.J.C 80 (2020) 1, 58]



Whereabout

- Environments requiring low inference latency, e.g.,
  - Level-1 trigger at CERN LHC experiments <O(1 μs)
  - Triggering in neutrino experiments <O(1 ms)
- Need efficient hardware for deployment → FPGAs

Problem statement

Question

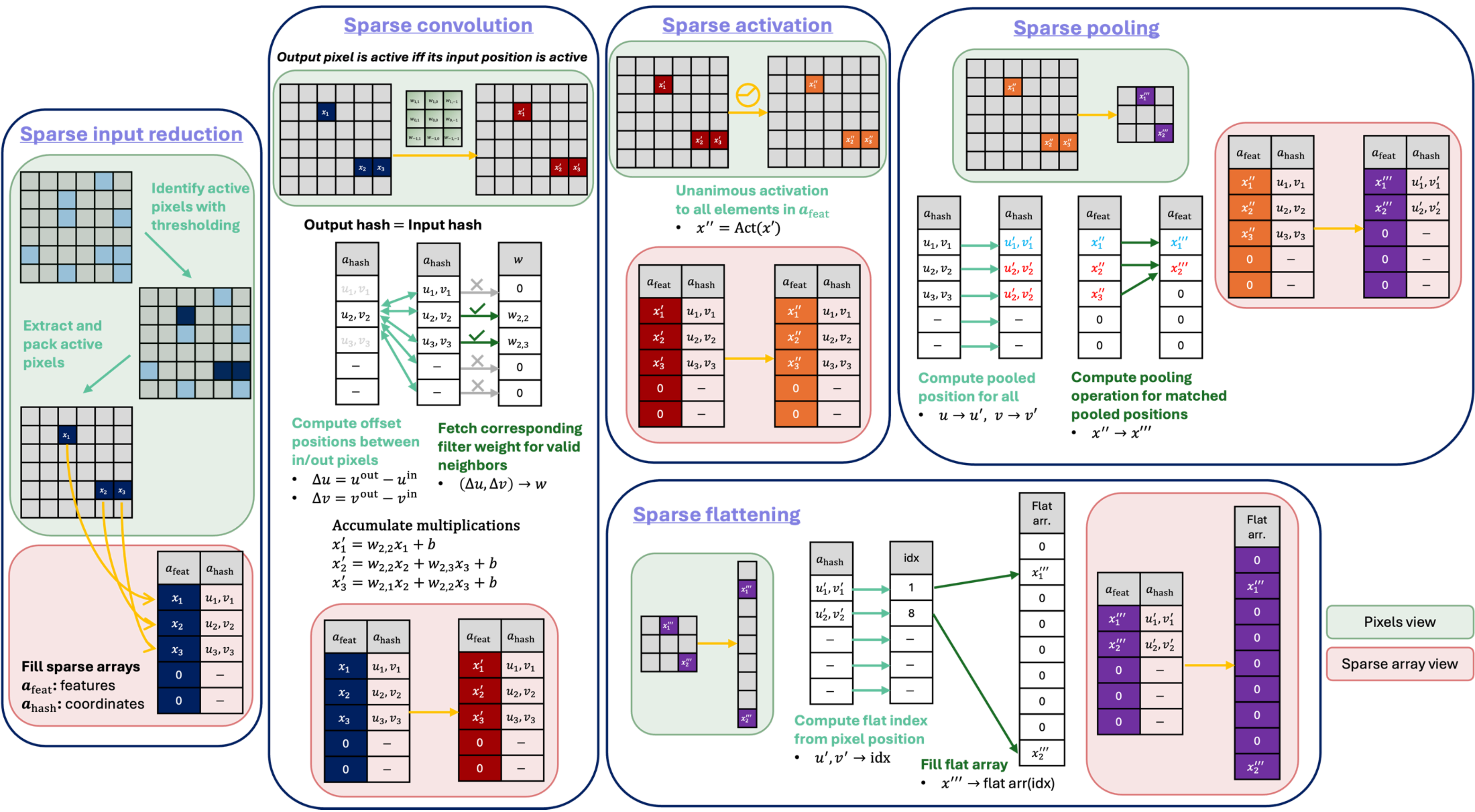
Can we improve CNN processing efficiency on FPGAs for image-like data with sparse features?

Applications in HEP

Trigger with tight latency constraints

- Muon tracking at LHC
- Calorimeter object detection at LHC
- Neutrino detection in LArTPC

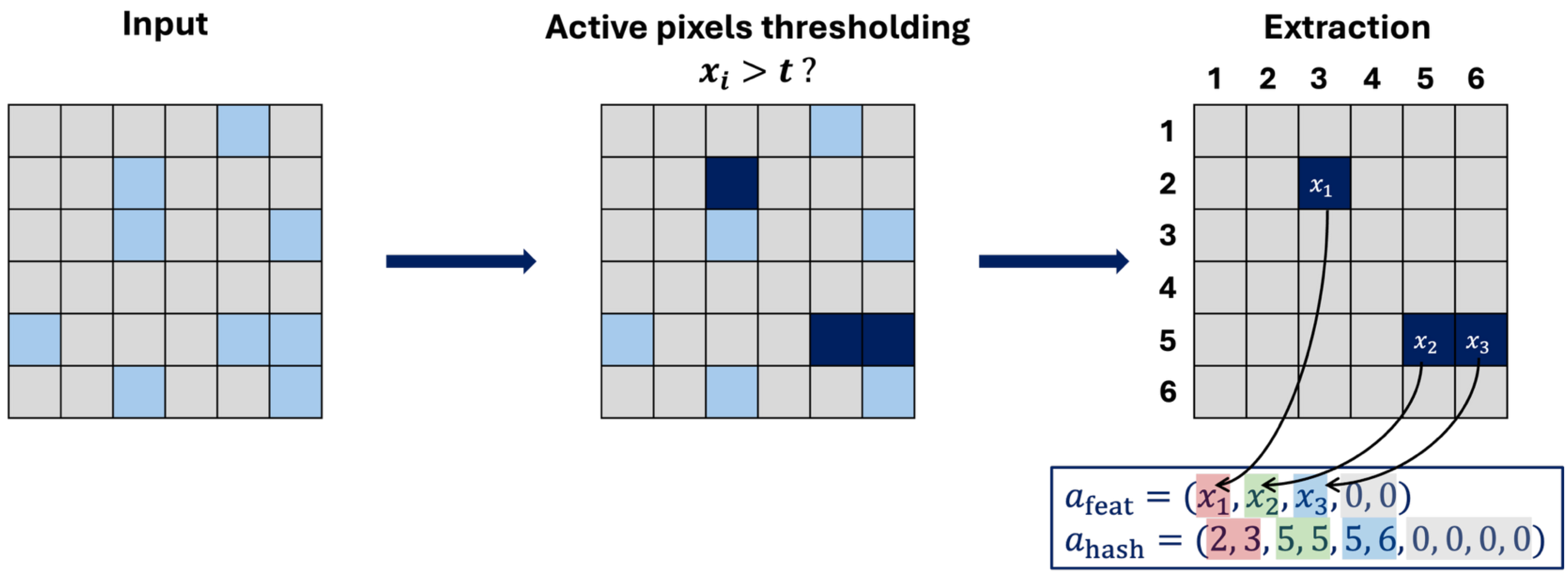
SparsePixels framework



- Core: five new layers to construct sparse CNNs

1. Sparse input reduction

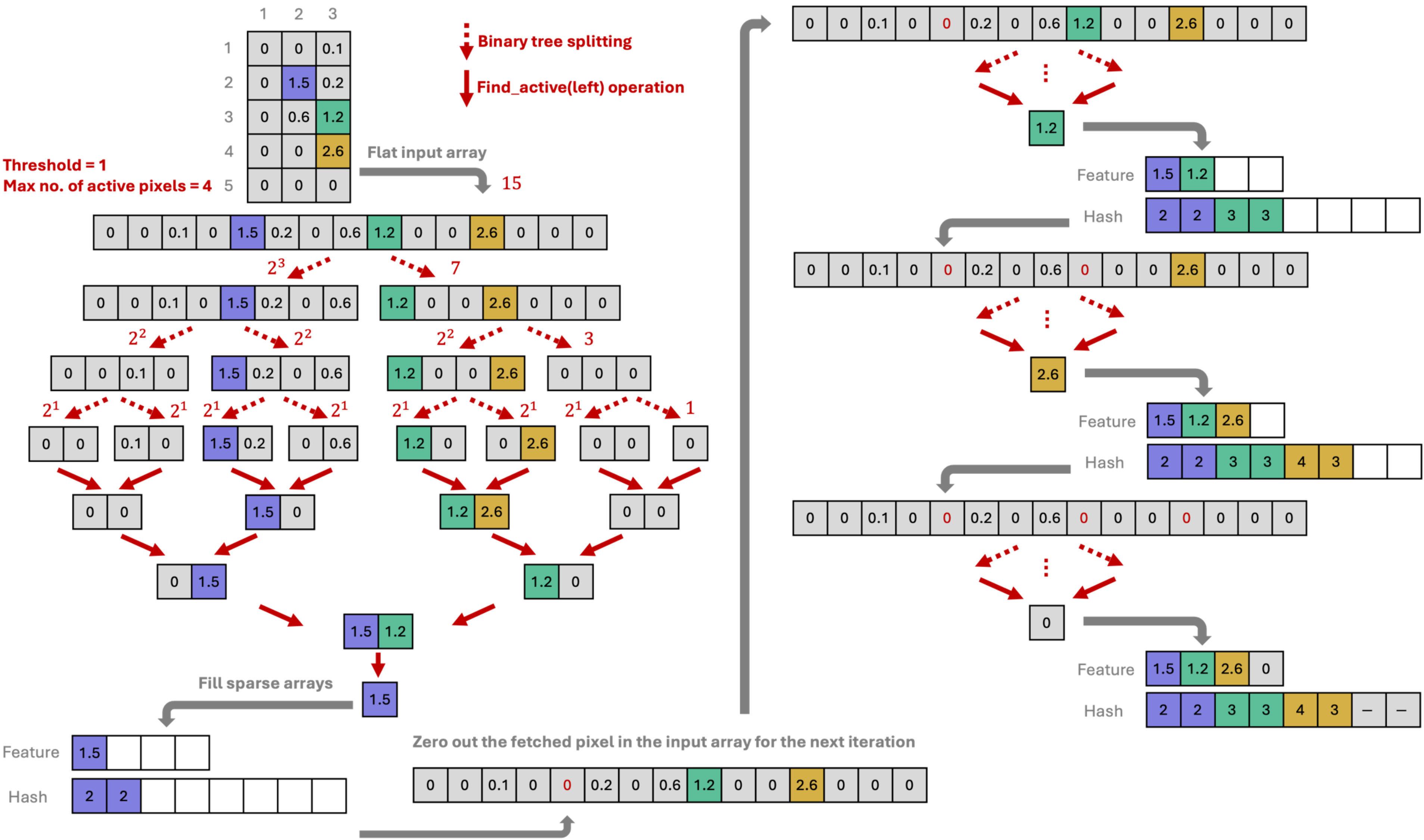
Dynamic storage of active input pixels



- **Active pixels:** pixels with feature value > a predefined threshold
- **Idea:** create two sparse arrays to dynamically store active pixels
  - Fix a max number of active pixels to be considered in the dataset $N_{\text{active}}^{\text{max}} \ll H \times W \times C$
  - a_{feat}: features of active pixels (array length = $N_{\text{active}}^{\text{max}} \times C$)
  - a_{hash}: height/width of active pixels (array length = $N_{\text{active}}^{\text{max}} \times 2$)
  - All subsequent sparse layers will operate on a_{feat}, a_{hash} only

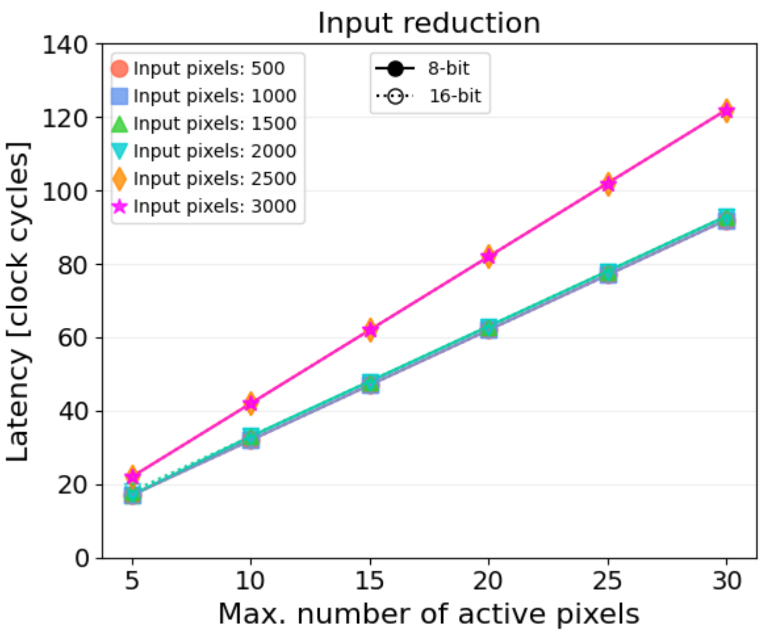
- **Challenge:** locations of active pixels vary event by event, this creates so many possible combinations, HLS synthesis will need to include all these paths if a simple loop over the input pixels is used for extraction

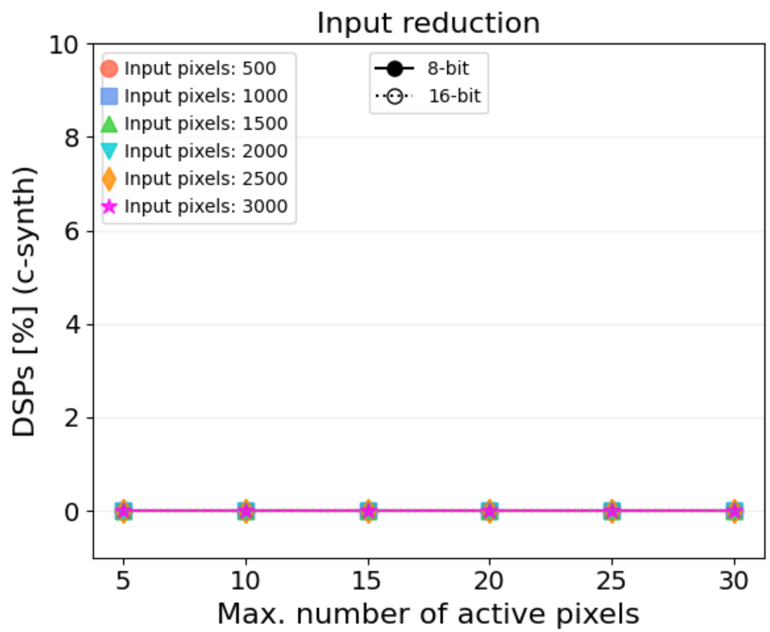
Sparse input reduction algorithm



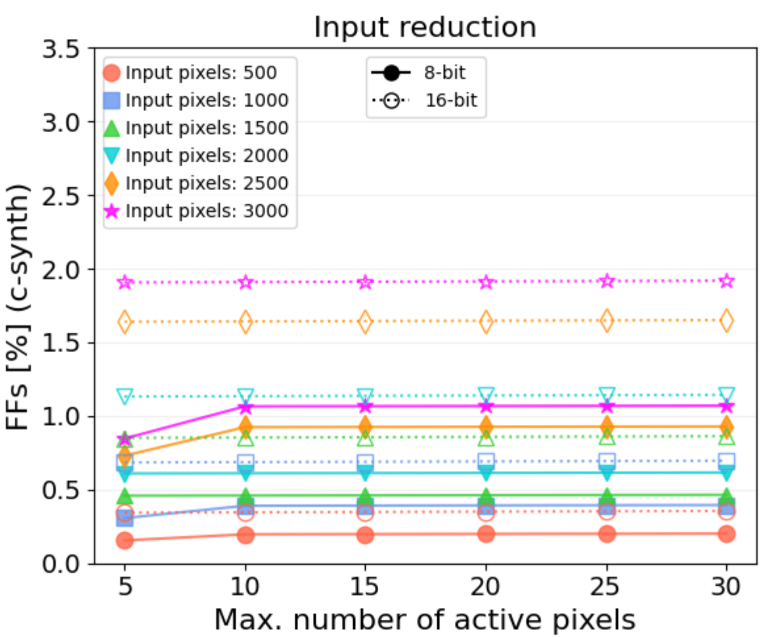
- Reduction tree depth (dashed arrow part) is $\mathrm{ceil}(\log_2(H \times W))$ (= 4 here)

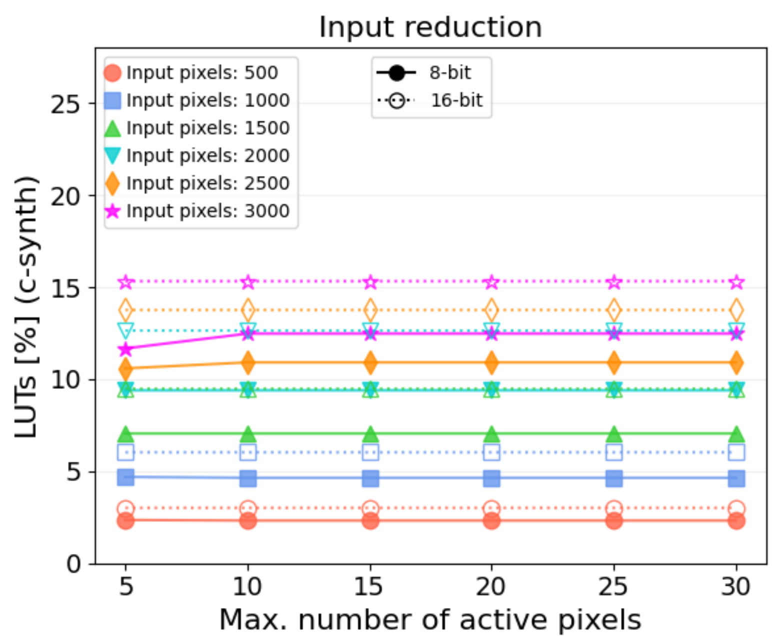
FPGA latency/resource scaling





These are **C-synth** results which could have overestimation in actual resource utilization



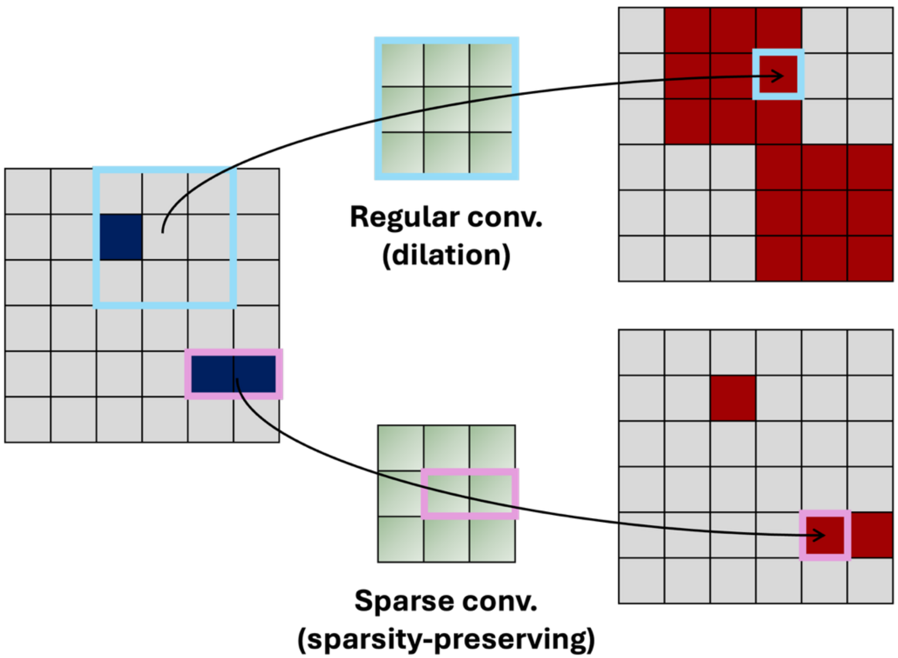


The discrete latency dependence on input size is due to ~ $\mathrm{ceil}(\log_2(H \times W))$

- Solid: 8-bit
- Dashed: 16-bit

2. Sparse convolution

Dilation in standard convolution

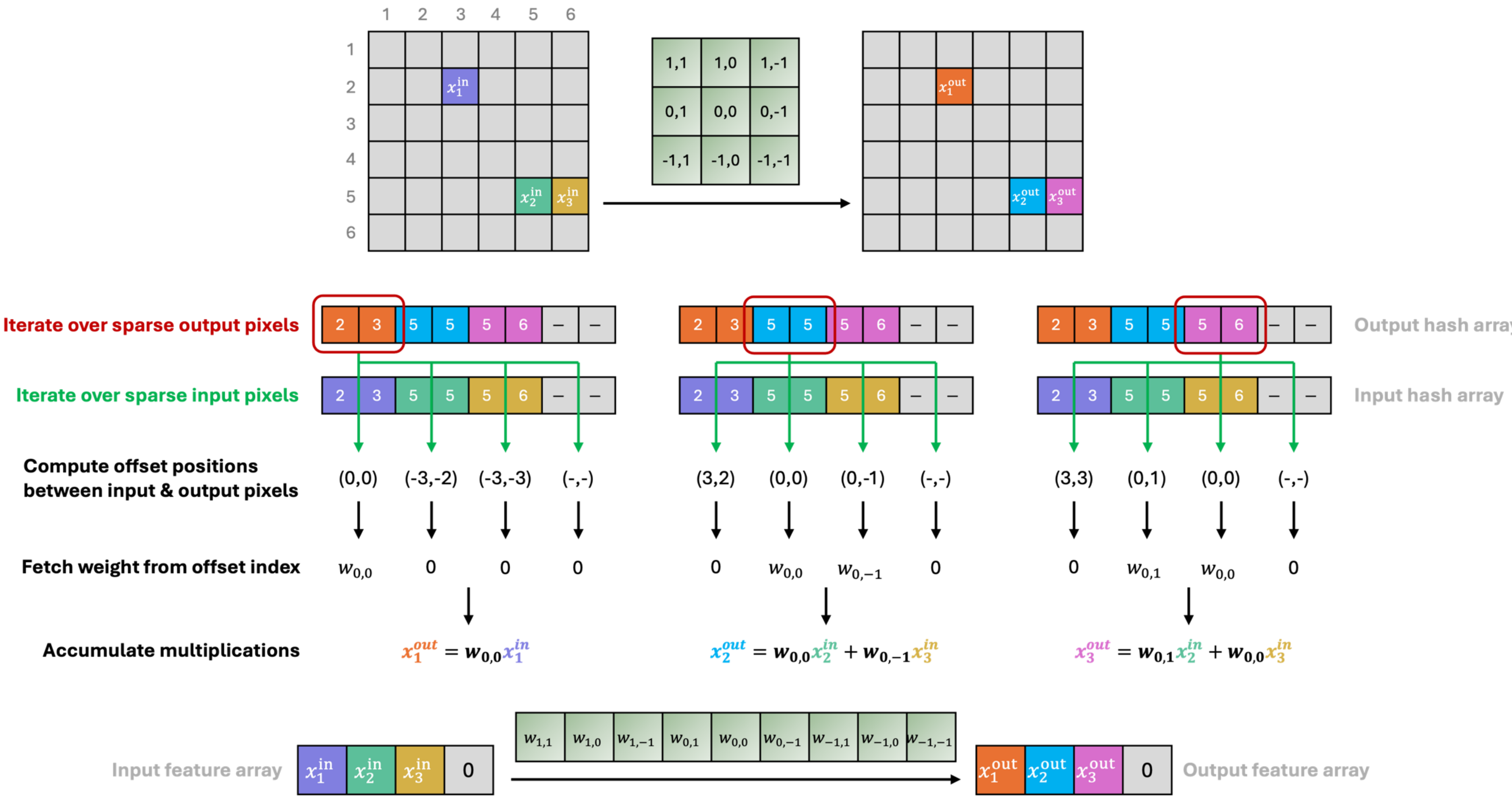


Standard convolution will quickly dilate active pixels and make data not sparse anymore and it loses the meaning of efficient computations on sparse data

Solution: consider a special class of CNNs that preserves the sparsity pattern between in/out

- An output pixel is active iff the input pixel at the same location is active
- At the cost of further constraining pixels communicating to their neighbors

Sparse convolution algorithm

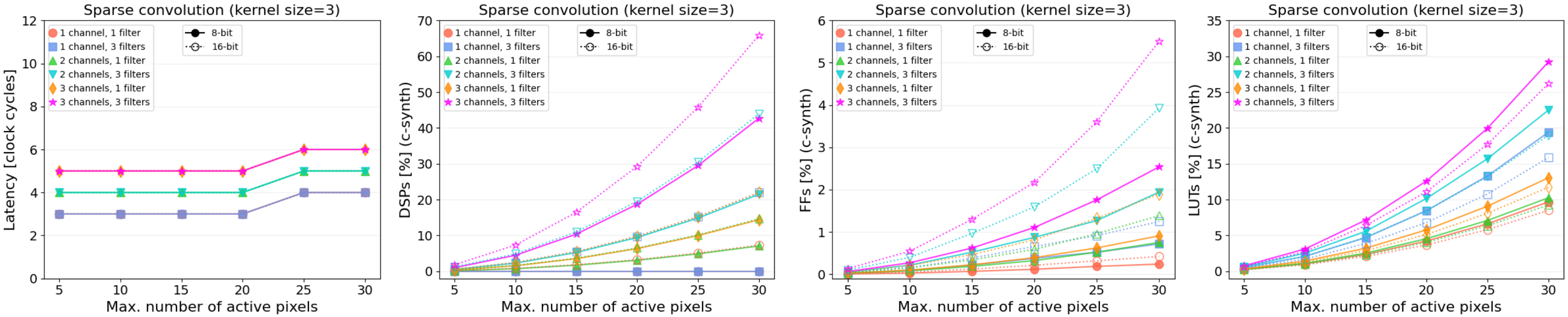


Number of iterations to perform one conv layer (nested loops of multiplications)

- **Standard conv:** $H \times W \times C_{\text{in}} \times C_{\text{out}} \times K \times K$
- **Sparse conv:** $N_{\text{active}}^{\max} \times N_{\text{active}}^{\max} \times C_{\text{in}} \times C_{\text{out}}$ (not scale directly with kernel size K)

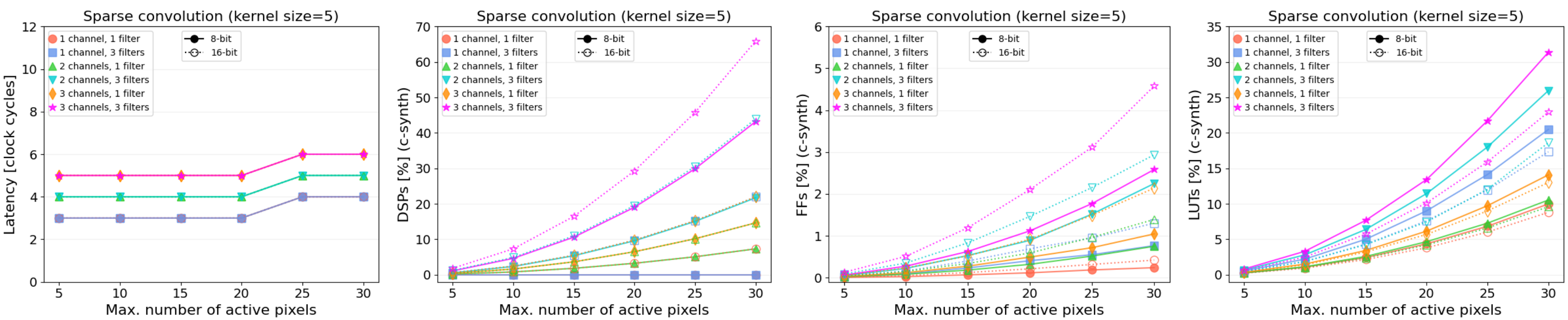
FPGA latency/resource scaling

These are **C-synth** results which could have overestimation in actual resource utilization



Kernel size = 3

Kernel size = 5



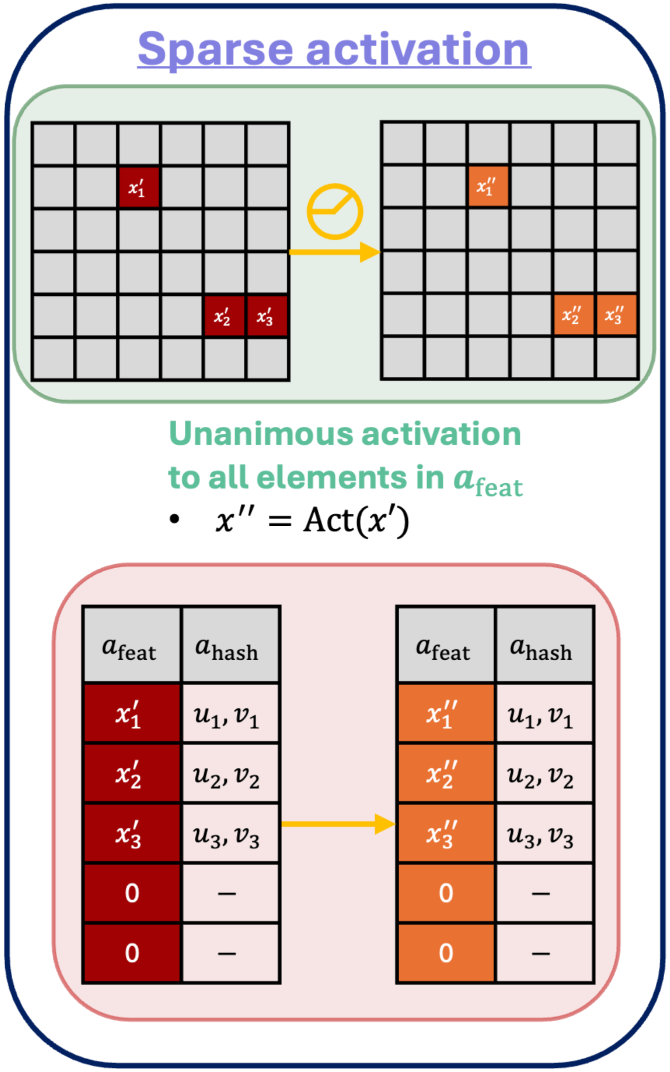
- The sparse conv algorithm does not loop over all kernel weights, so it does not directly scale with kernel size

3. Sparse activation

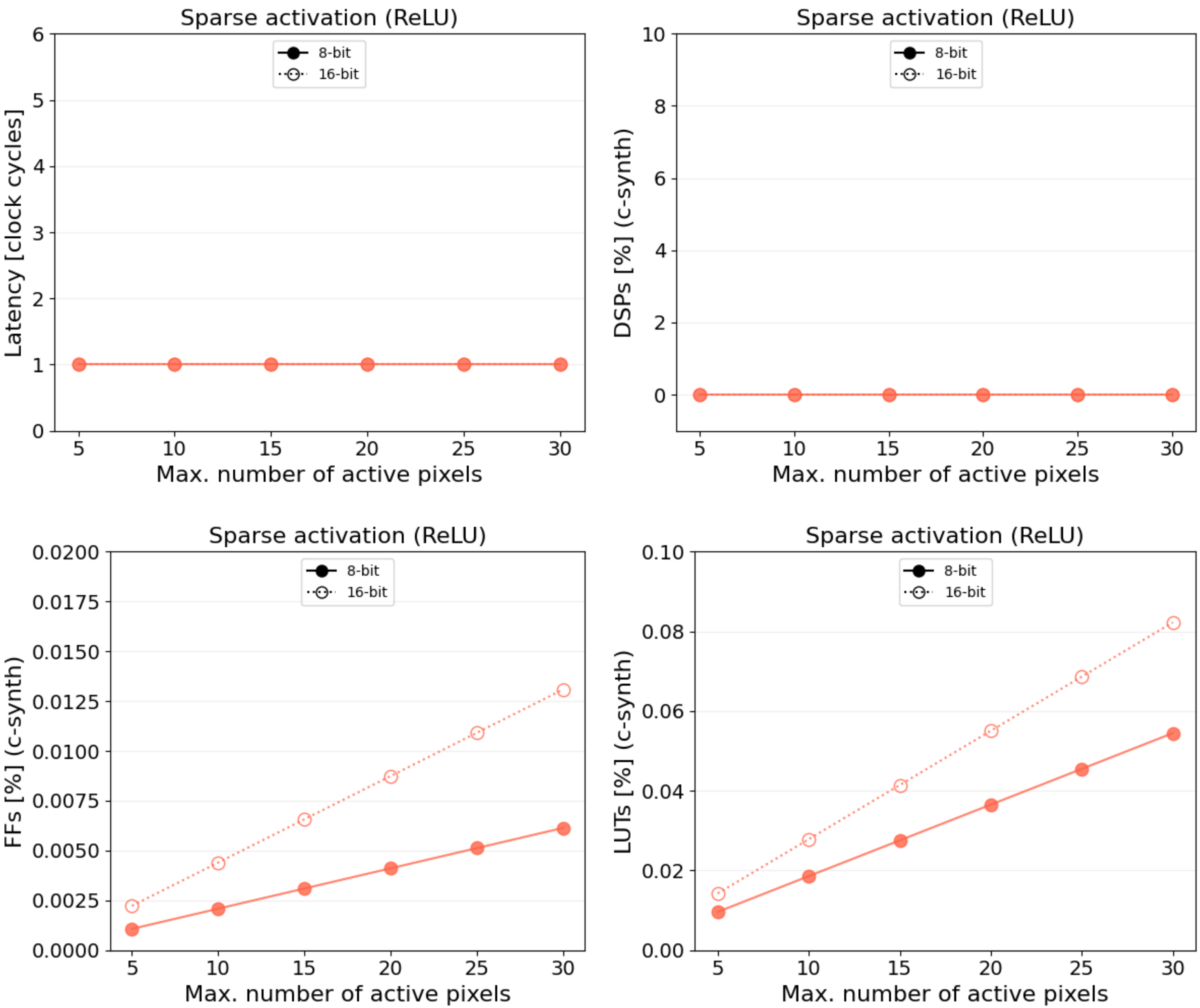
Sparse activation

Apply element-wise activation to sparse feature array

- **Standard conv:** $H \times W \times C$ elements
- **Sparse conv:** $N_{\text{active}}^{\max} \times C$ elements



FPGA latency/resource scaling

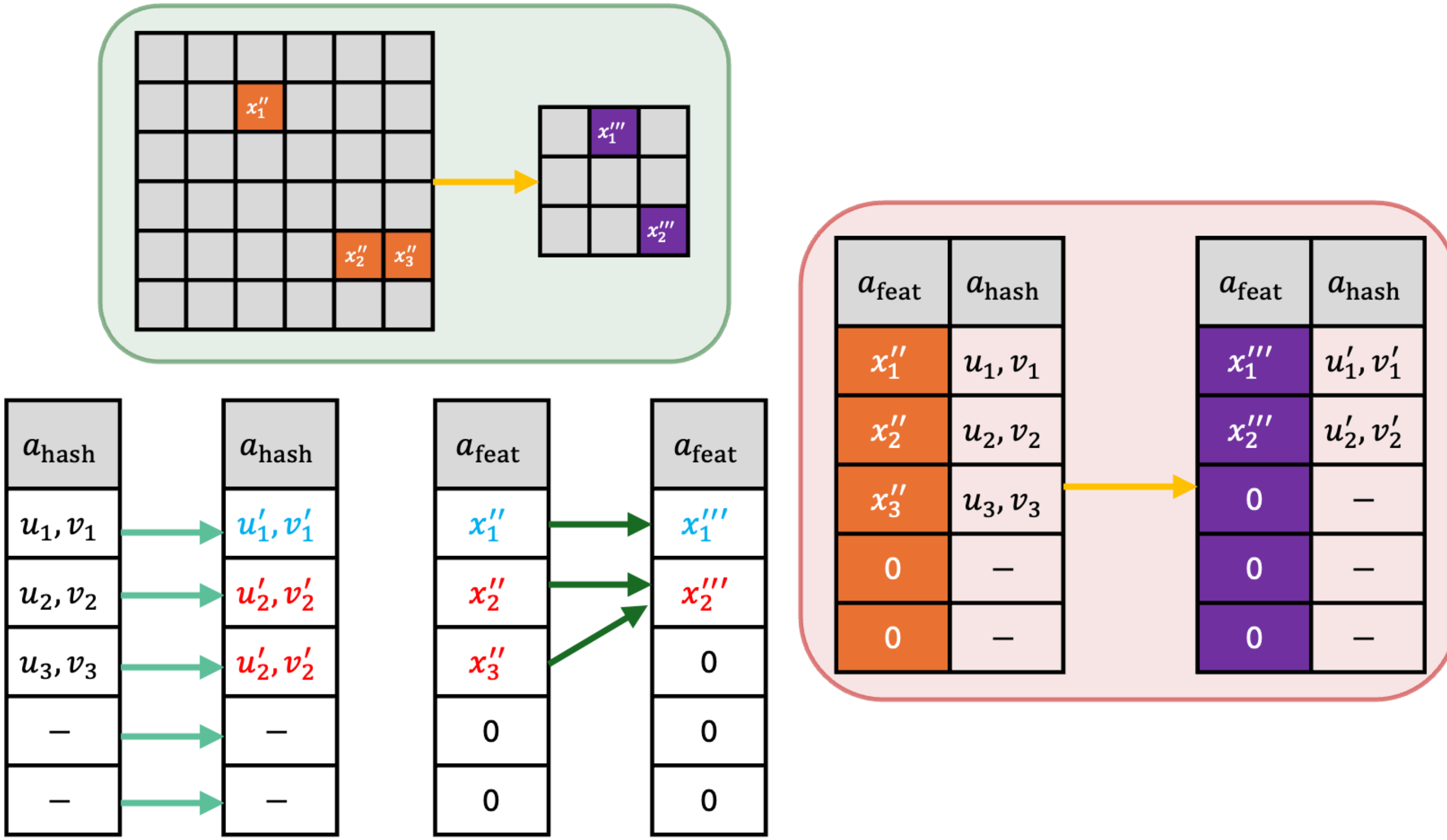


These are **C-synth** results which could have overestimation in actual resource utilization

4. Sparse pooling

Sparse pooling

Sparse pooling



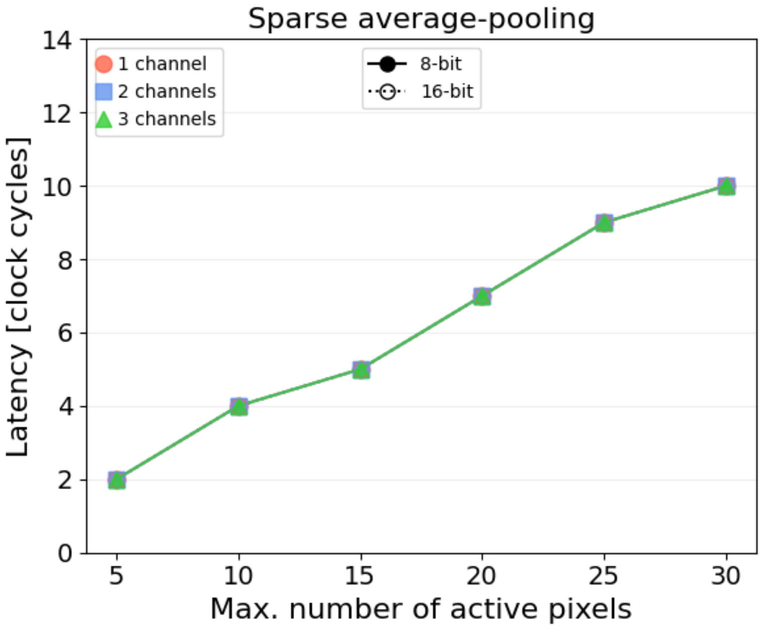
Compute pooled position for all

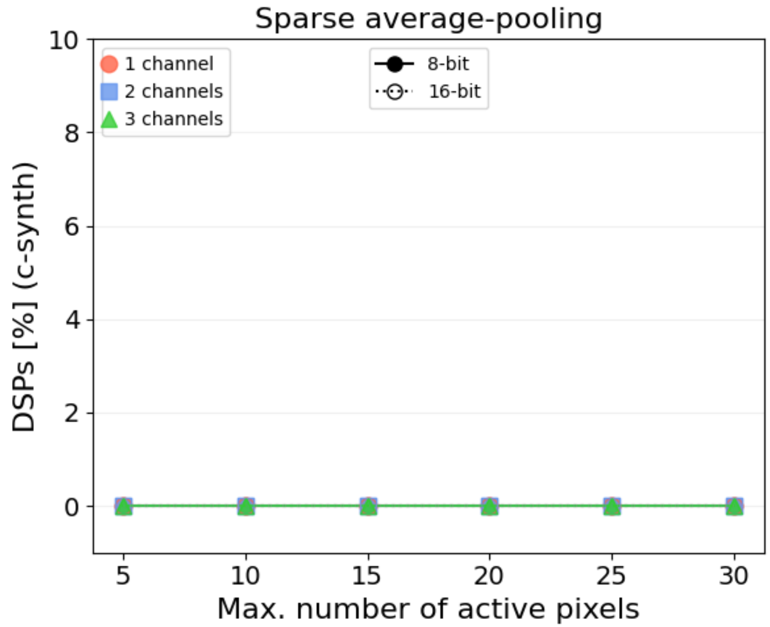
- $u \to u', \ v \to v'$

Compute pooling operation for matched pooled positions

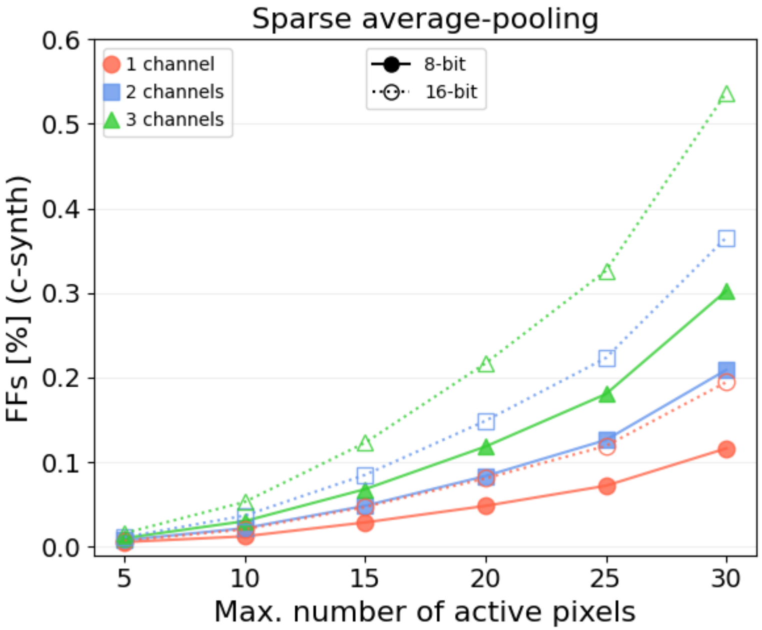
- $x'' \to x'''$

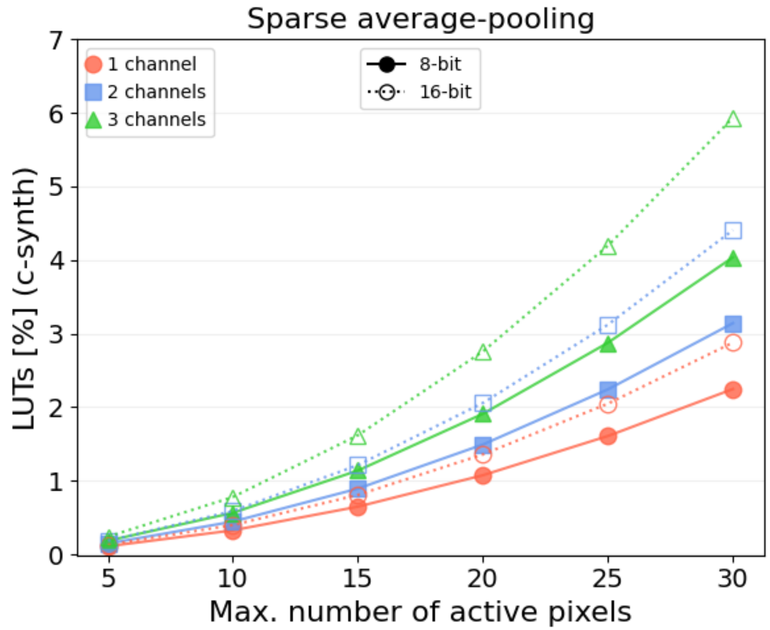
FPGA latency/resource scaling





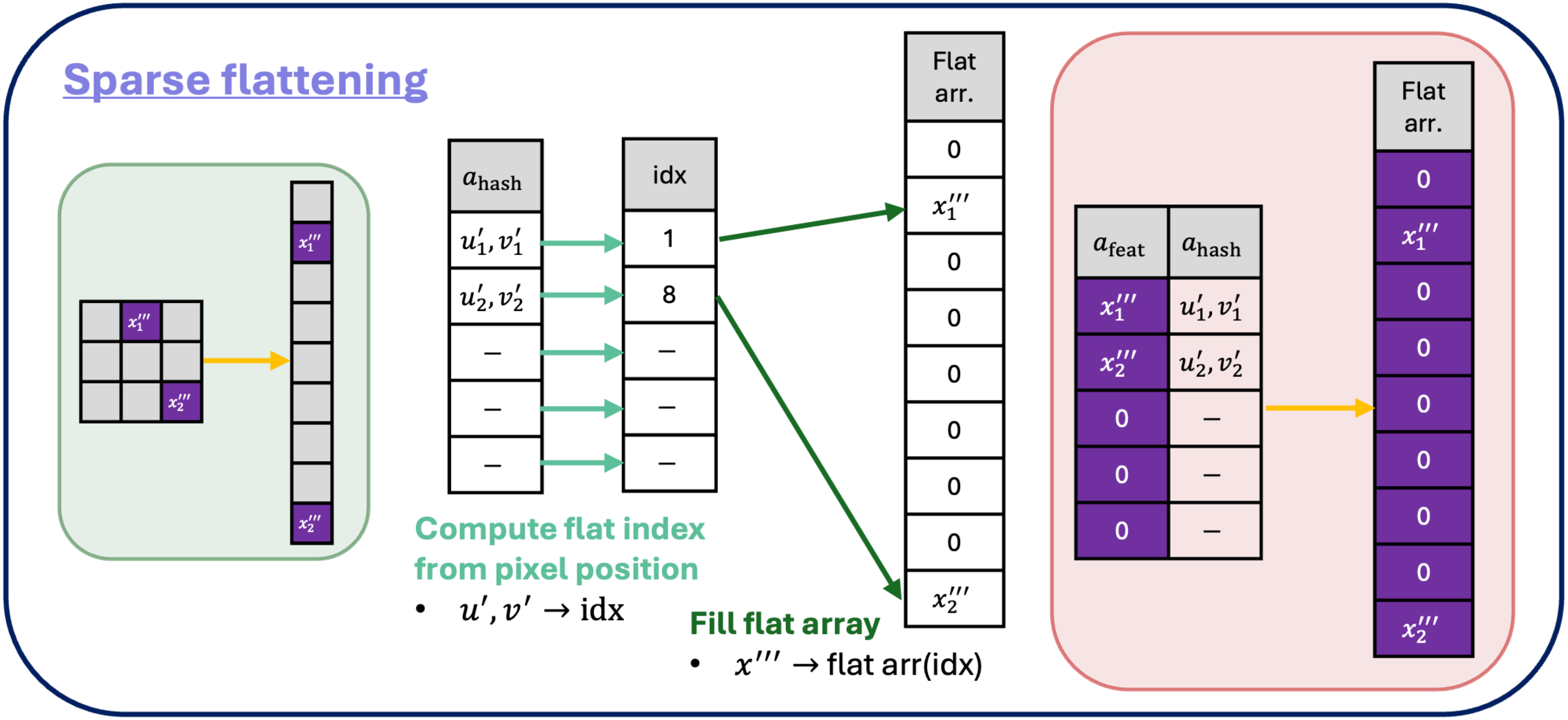
These are **C-synth** results which could have overestimation in actual resource utilization



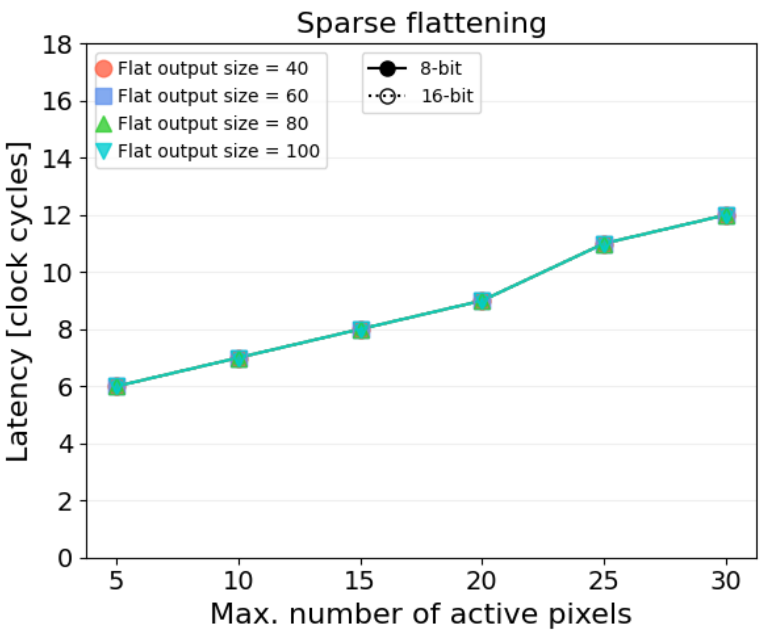


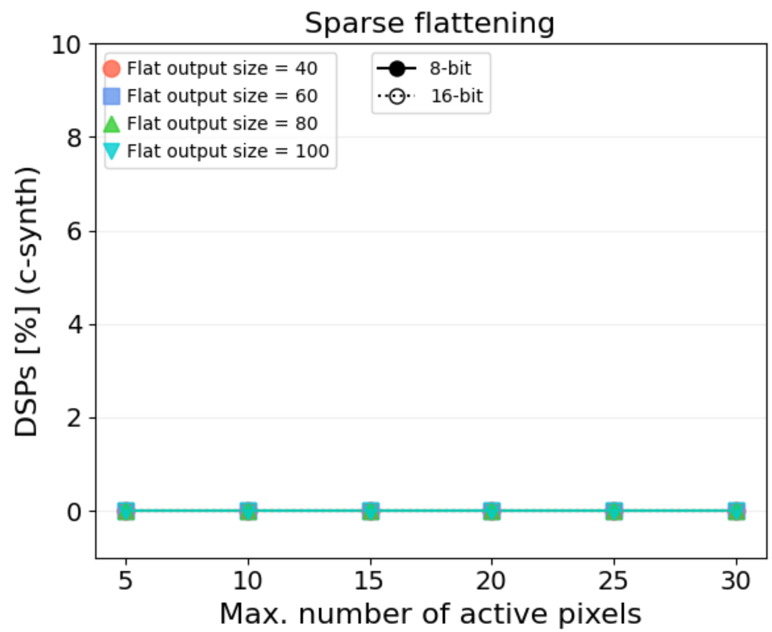
5. Sparse flattening

Sparse flattening

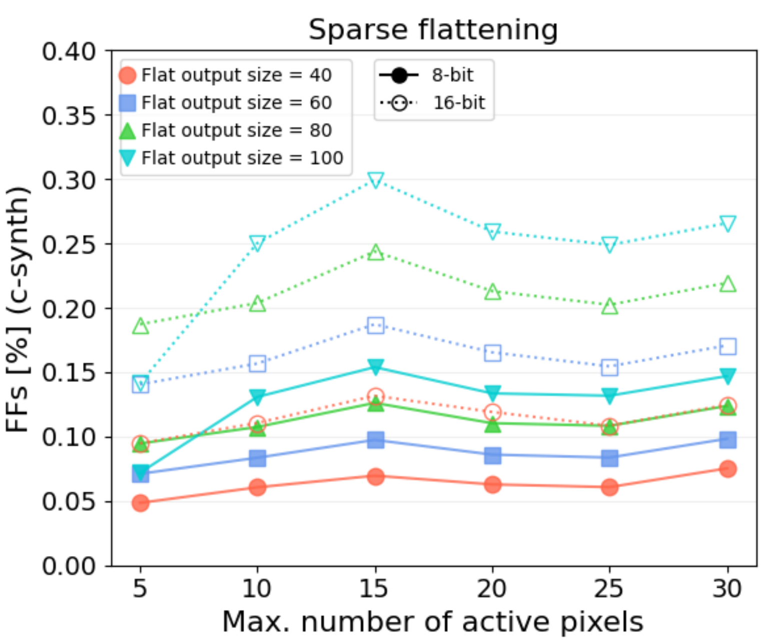


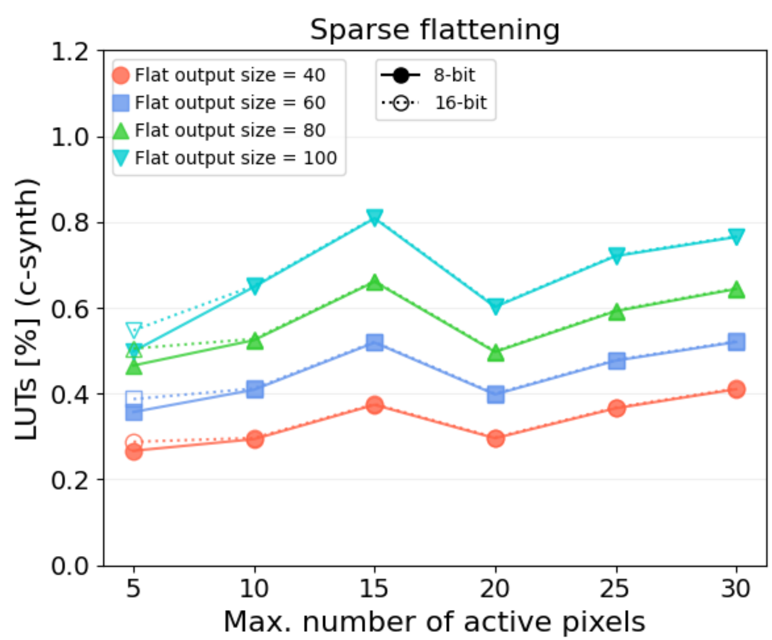
FPGA latency/resource scaling





These are **C-synth** results which could have overestimation in actual resource utilization

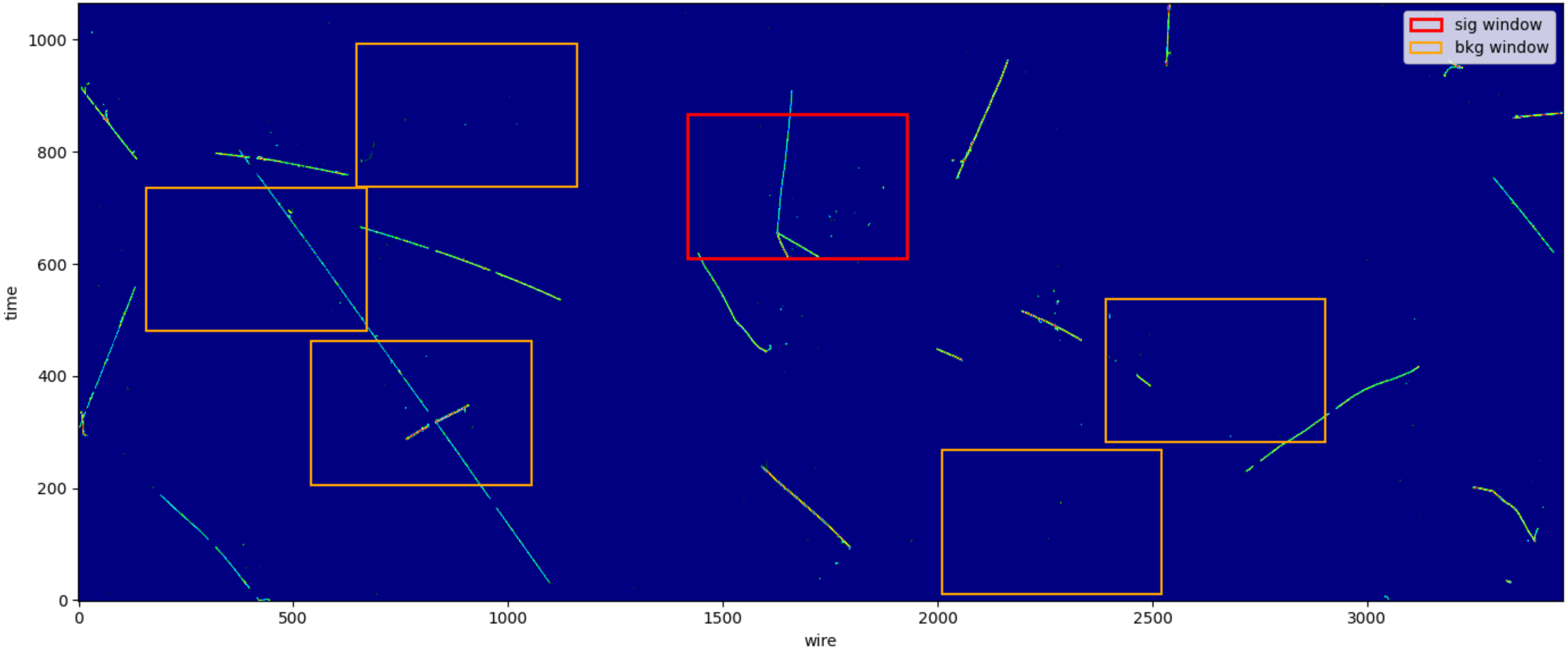




Partial unrolling implemented, so not monotonic curves here

Dataset demo: MicroBooNE open samples

Dataset



MicroBooNE open samples dataset [github: uboone/OpenSamples dataset: 10.5281/zenodo.7262009]

- Simulated neutrino interactions overlaid on top of cosmic ray data in LArTPC
- Readout time window is O(1 ms) [2010.08653]
- Input size: 6400 time ticks * 3456 wires (or 2400 wires in other two planes)

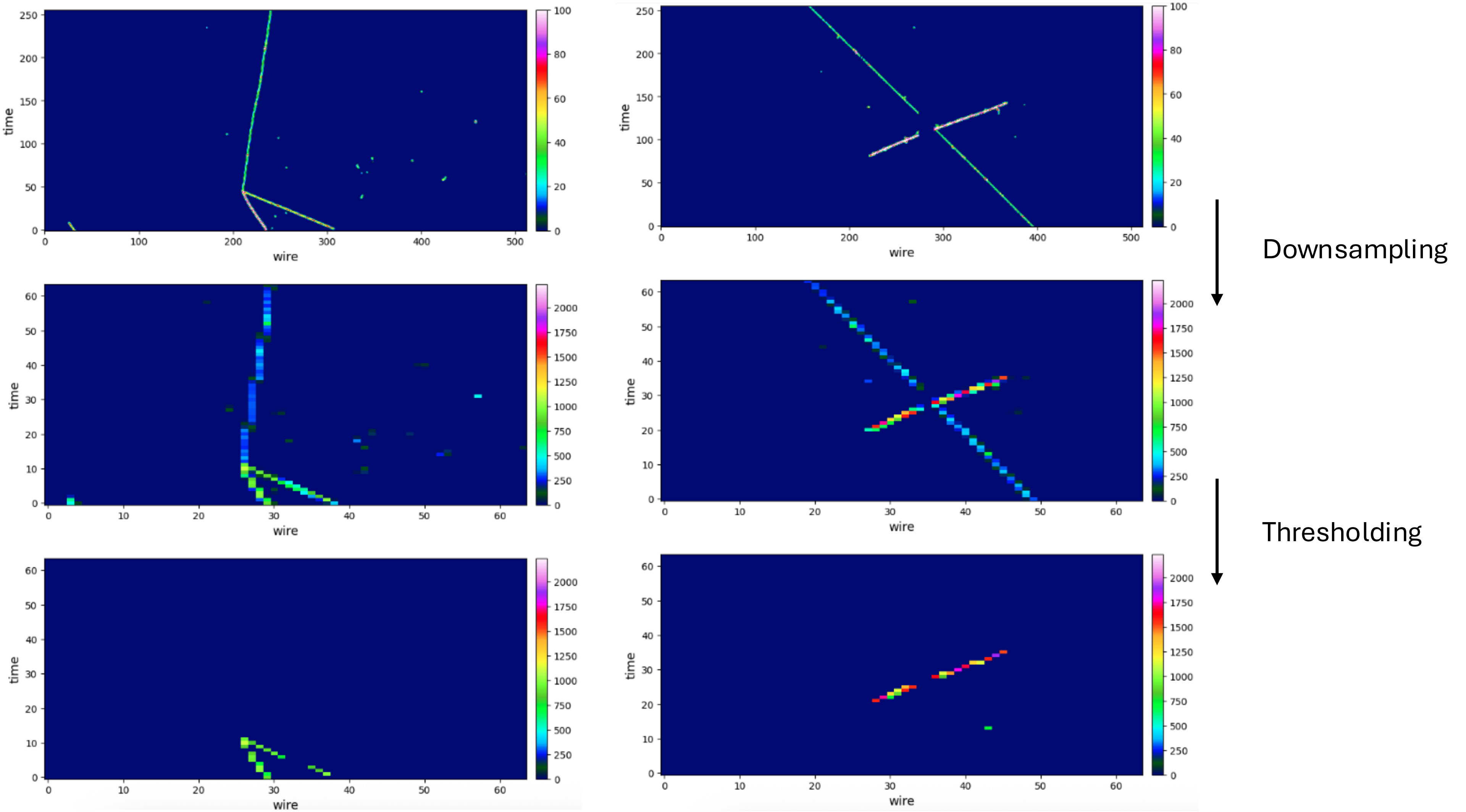
Labeling for a binary classification

- Window size 256*512
- Windows containing a MC truth neutrino vertex -> signal
- Windows randomly cropped elsewhere -> background

Dataset

A signal window

A background window



- Downsampling to reduce window size -> 64*64
- Thresholding to suppress low-energy hits and further sparsify

Models

Standard CNN (QKeras layers)

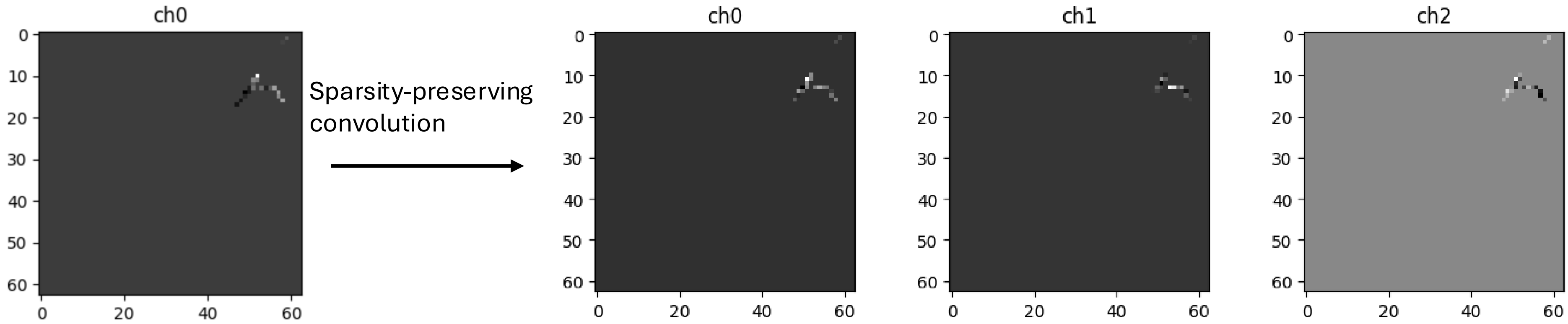
```
Model: "cnn_full"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 x_in (InputLayer)           [(None, 63, 63, 1)]       0

 conv1 (QConv2D)             (None, 63, 63, 1)         50

 relu1 (QActivation)         (None, 63, 63, 1)         0

 conv2 (QConv2D)             (None, 63, 63, 3)         150

 relu2 (QActivation)         (None, 63, 63, 3)         0

 pool1 (AveragePooling2D)    (None, 9, 9, 3)           0

 flatten (Flatten)           (None, 243)               0

 dense1 (QDense)             (None, 16)                3904

 relu3 (QActivation)         (None, 16)                0

 dense2 (QDense)             (None, 1)                 17

 sigmoid (Activation)        (None, 1)                 0

=================================================================
Total params: 4121 (16.10 KB)
Trainable params: 4121 (16.10 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

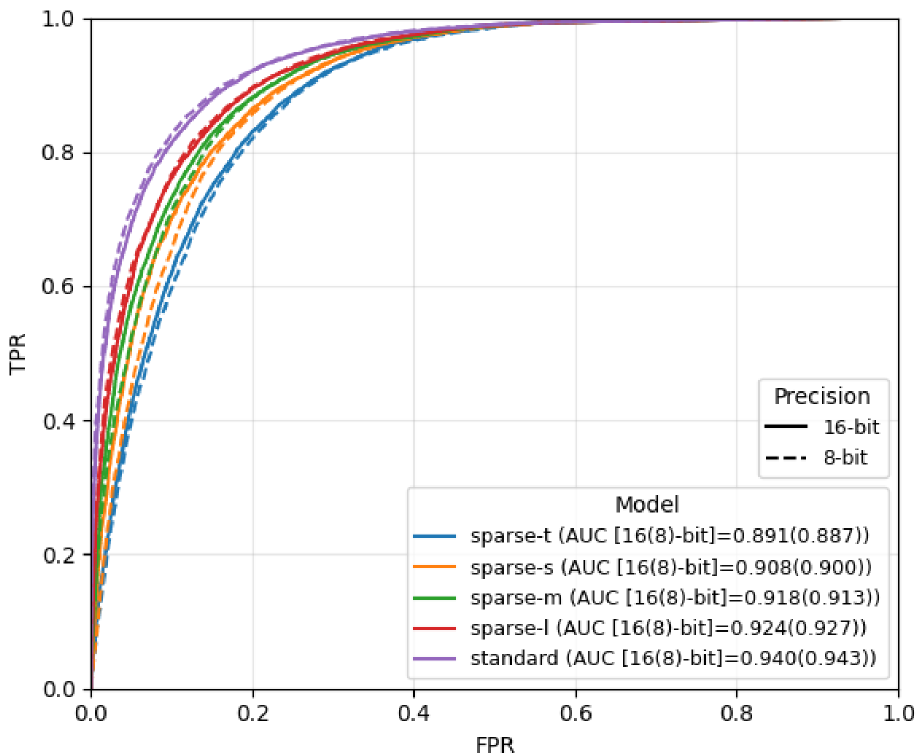
Sparse CNN (our SparsePixels layers)

```
Model: "cnn_sparse"
__________________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================================
 x_in (InputLayer)           [(None, 63, 63, 1)]          0         []

 input_reduce (InputReduce)  ((None, 63, 63, 1),          0         ['x_in[0][0]']
                              (None, 63, 63, 1))

 conv1 (QConv2DSparse)       (None, 63, 63, 1)            50        ['input_reduce[0][0]',
                                                                     'input_reduce[0][1]']

 relu1 (QActivation)         (None, 63, 63, 1)            0         ['conv1[0][0]']

 conv2 (QConv2DSparse)       (None, 63, 63, 3)            150       ['relu1[0][0]',
                                                                     'input_reduce[0][1]']

 relu2 (QActivation)         (None, 63, 63, 3)            0         ['conv2[0][0]']

 pool1 (AveragePooling2DSpa  ((None, 9, 9, 3),            0         ['relu2[0][0]',
 rse)                         (None, 9, 9, 1))                       'input_reduce[0][1]']

 flatten (Flatten)           (None, 243)                  0         ['pool1[0][0]']

 dense1 (QDense)             (None, 16)                   3904      ['flatten[0][0]']

 relu3 (QActivation)         (None, 16)                   0         ['dense1[0][0]']

 dense2 (QDense)             (None, 1)                    17        ['relu3[0][0]']

 sigmoid (Activation)        (None, 1)                    0         ['dense2[0][0]']

==================================================================================================
Total params: 4121 (16.10 KB)
Trainable params: 4121 (16.10 KB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________________________
```

- Same base model architecture
- For sparse CNN, consider four $N_{\text{active}}^{\max}$: 8 (tiny), 12 (small), 16 (medium), 20 (large)

Performance





Same base model architecture

- **Standard** CNN [standard]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 8$ [sparse-t]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 12$ [sparse-s]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 16$ [sparse-m]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 20$ [sparse-l]

- Some performance loss as expected, but will see great efficiency gain in HLS implementation

Our HLS implementation for sparse CNN

```
template <class data_T, class hash_T, int N_h, int N_w, int N_c, int N_sparse>
void sparse_input_reduce(data_T input_arr[N_h * N_w * N_c],
                         data_T threshold,
                         data_T sparse_arr_feat[N_sparse * N_c],
                         hash_T sparse_arr_hash[N_sparse * 2]) {
```

```
template <class data_T, class res_T, class hash_T, class w_T, class b_T, int N_sparse, int n_chan, int n_filt, int ker_size>
void sparse_conv(data_T sparse_arr_feat_in[N_sparse * n_chan],
                 res_T sparse_arr_feat_out[N_sparse * n_filt],
                 hash_T sparse_arr_hash[N_sparse * 2],
                 w_T w[ker_size * ker_size * n_chan * n_filt],
                 b_T b[n_filt]) {
```

```
template <class data_T, class res_T, int N_sparse, int n_chan>
void sparse_relu(data_T sparse_arr_feat_in[N_sparse * n_chan], res_T sparse_arr_feat_out[N_sparse * n_chan]) {
```

```
template <class data_T, class res_T, class hash_T, int N_sparse, int n_chan, int pool_size>
void sparse_pooling_avg(data_T sparse_arr_feat_in[N_sparse * n_chan],
                        res_T sparse_arr_feat_out[N_sparse * n_chan],
                        hash_T sparse_arr_hash_in[N_sparse * 2],
                        hash_T sparse_arr_hash_out[N_sparse * 2]) {
```

```
template<class data_T, class hash_T, int n_height, int n_width, int n_chan, int N_sparse>
void sparse_flatten(data_T sparse_arr_feat[N_sparse * n_chan],
                    hash_T sparse_arr_hash[N_sparse * 2],
                    data_T flat_arr[n_height * n_width * n_chan]) {
```

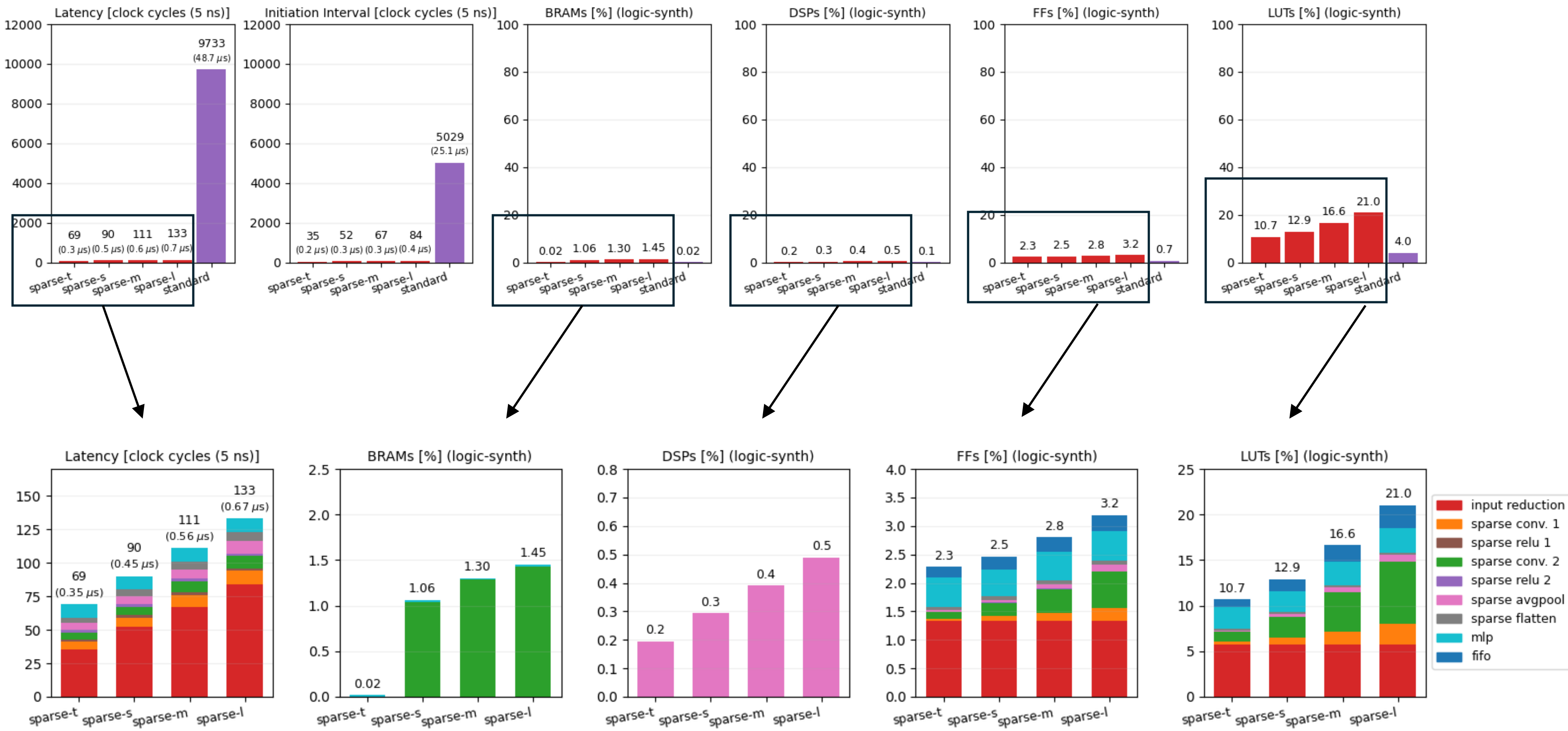
- ^ the five sparse operations illustrated earlier
- Will eventually integrate into hls4ml for auto conversion

Model latency and resource utilization

Standard CNN: io_stream, fully serialized
Sparse CNN: io_parallel, fully parallelized

Model bit width = 8

Standard CNN vs sparse CNNs



Per-module breakdown in sparse models

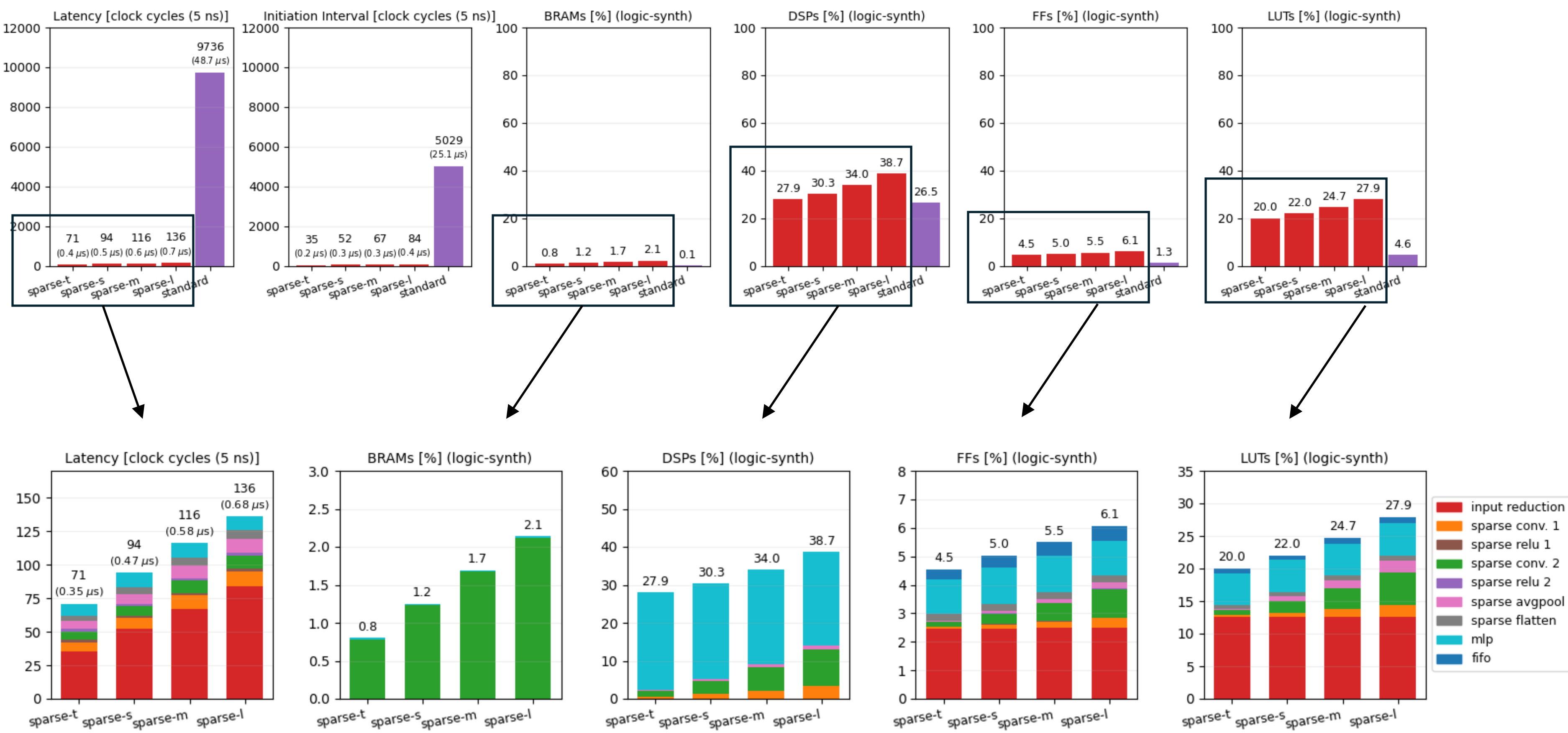
- Sparse CNNs are orders of magnitude faster than standard CNN of same base arch, given both are well within the available resources
- Now these sparse CNNs are fully parallelized, but one could potentially tune the reuse factors to lower the resources

Model latency and resource utilization

Standard CNN: io_stream, fully serialized
Sparse CNN: io_parallel, fully parallelized

Model bit width = 16

Standard CNN vs sparse CNNs



Per-module breakdown in sparse models

- Sparse CNNs are orders of magnitude faster than standard CNN of same base arch, given both are well within the available resources
- Now these sparse CNNs are fully parallelized, but one could potentially tune the reuse factors to lower the resources

Summary

SparsePixels developed for implementing efficient CNNs for sparse data on FPGAs with low-latency demands

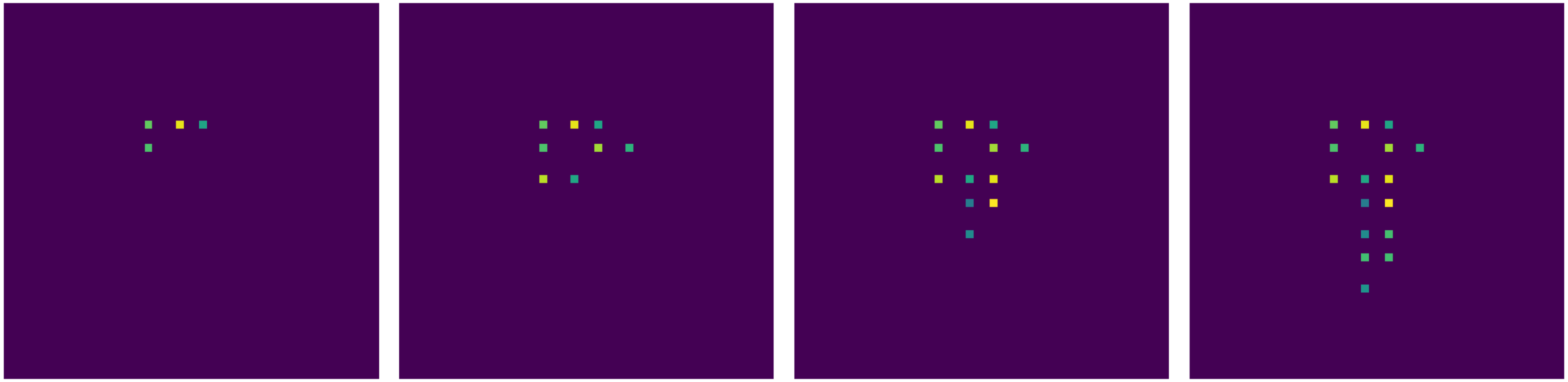
- Dynamically pick up sparse active pixels and only do computations on them ($N_{\text{active}}^{\text{max}} \times N_{\text{active}}^{\text{max}} \ll H \times W$)
- Allows for parallelization for large input size while not possible for standard CNNs
- Could be orders of magnitude faster than standard CNN of same size, with small performance loss
- Demonstrated on microBooNE neutrino OpenSamples (LHC jet and MNIST in backup)
- Python library developed to support training sparse CNNs
- C++ modules developed to support HLS implementation

Outlook

- Current implementation is fully parallelized, reuse factor tuning and streaming could be developed in future
- Paper/codes will be out soon (stay tuned!!)
- Will eventually integrate into hls4ml

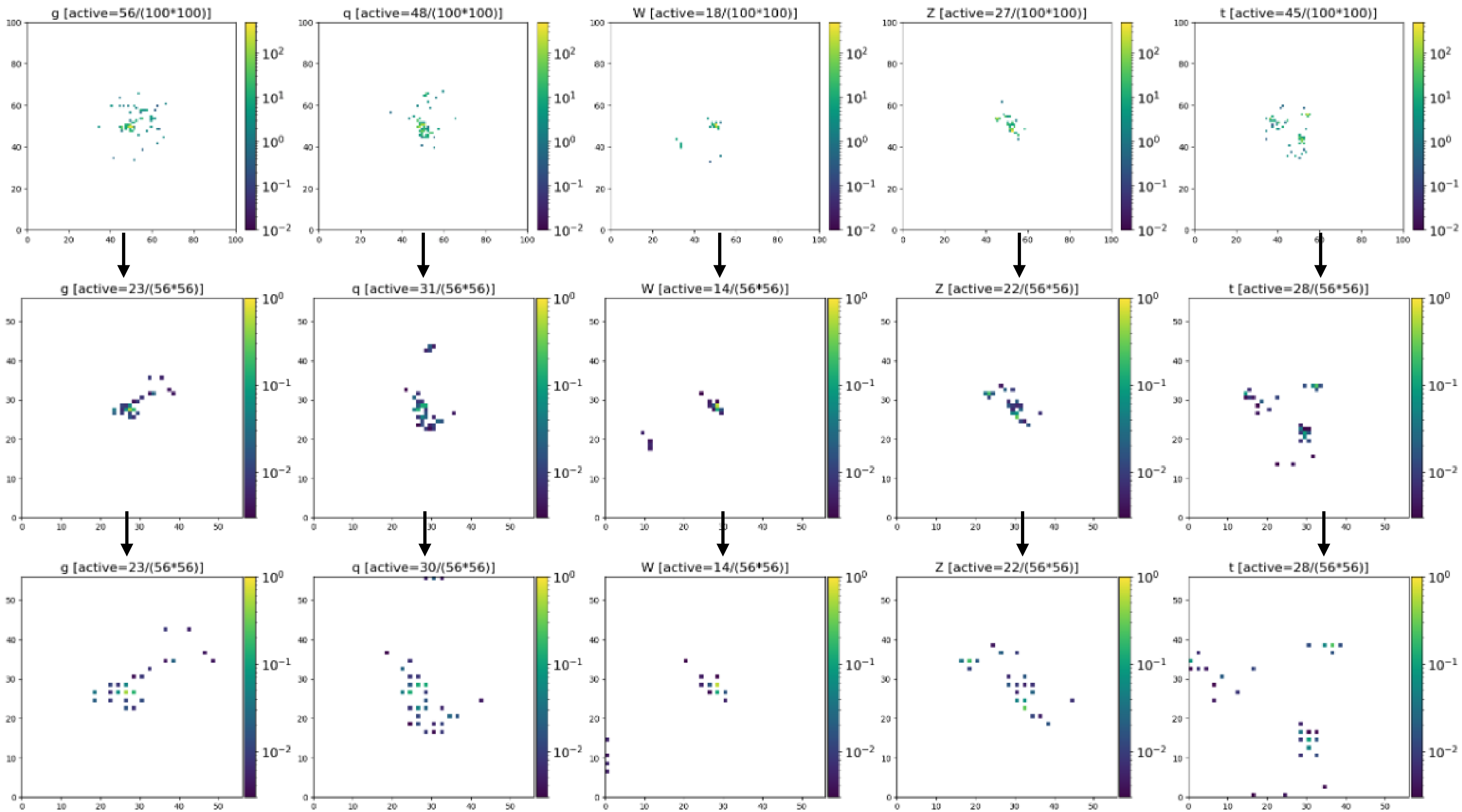
Backup

Choose how many active pixels to consider



Dataset demo: hls4ml LHC jet

Dataset



Original hls4ml LHC jet dataset (150 particles) [10.5281/zenodo.3602260]

- 100*100*1 input pixels, 5 classes (g, q, W, Z, t)

Pooled and inflated to sparsify

- 56*56*1 input pixels

Models

Standard CNN (QKeras layers)

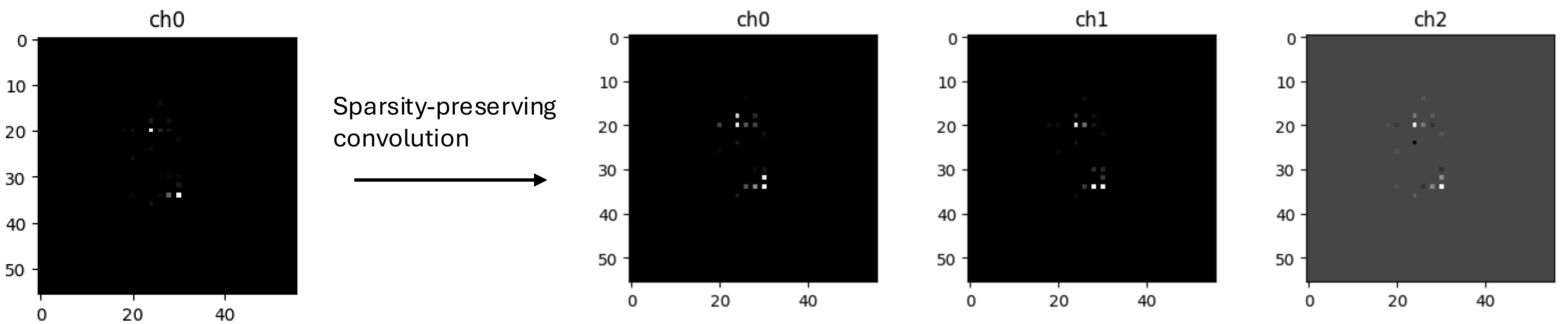
```
Model: "cnn_full"
______________________________________________________________________
 Layer (type)                 Output Shape              Param #
======================================================================
 x_in (InputLayer)            [(None, 56, 56, 1)]       0

 conv1 (QConv2D)              (None, 56, 56, 3)         246

 relu1 (QActivation)          (None, 56, 56, 3)         0

 conv2 (QConv2D)              (None, 56, 56, 1)         244

 relu2 (QActivation)          (None, 56, 56, 1)         0

 pool1 (AveragePooling2D)     (None, 14, 14, 1)         0

 flatten (Flatten)            (None, 196)               0

 dense1 (QDense)              (None, 20)                3940

 relu3 (QActivation)          (None, 20)                0

 dense2 (QDense)              (None, 5)                 105

 softmax (Activation)         (None, 5)                 0

======================================================================
Total params: 4535 (17.71 KB)
Trainable params: 4535 (17.71 KB)
Non-trainable params: 0 (0.00 Byte)
______________________________________________________________________
```

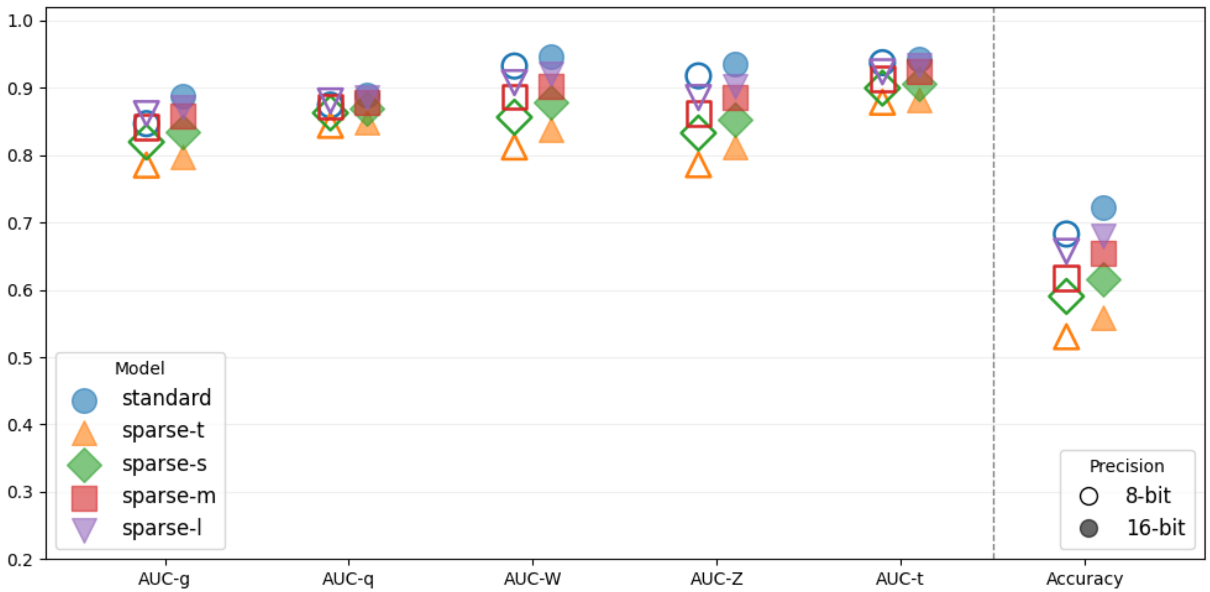
Sparse CNN (our SparsePixels layers)

```
Model: "cnn_sparse"
__________________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================================
 x_in (InputLayer)           [(None, 56, 56, 1)]          0         []

 input_reduce (InputReduce)  ((None, 56, 56, 1),          0         ['x_in[0][0]']
                              (None, 56, 56, 1))

 conv1 (QConv2DSparse)       (None, 56, 56, 3)            246       ['input_reduce[0][0]',
                                                                     'input_reduce[0][1]']

 relu1 (QActivation)         (None, 56, 56, 3)            0         ['conv1[0][0]']

 conv2 (QConv2DSparse)       (None, 56, 56, 1)            244       ['relu1[0][0]',
                                                                     'input_reduce[0][1]']

 relu2 (QActivation)         (None, 56, 56, 1)            0         ['conv2[0][0]']

 pool1 (AveragePooling2DSpa  ((None, 14, 14, 1),          0         ['relu2[0][0]',
 rse)                         (None, 14, 14, 1))                     'input_reduce[0][1]']

 flatten (Flatten)           (None, 196)                  0         ['pool1[0][0]']

 dense1 (QDense)             (None, 20)                   3940      ['flatten[0][0]']

 relu3 (QActivation)         (None, 20)                   0         ['dense1[0][0]']

 dense2 (QDense)             (None, 5)                    105       ['relu3[0][0]']

 softmax (Activation)        (None, 5)                    0         ['dense2[0][0]']

==================================================================================================
Total params: 4535 (17.71 KB)
Trainable params: 4535 (17.71 KB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________________________
```

- Same base model architecture
- For sparse CNN, consider four $N_{\text{active}}^{\max}$: 8 (tiny), 12 (small), 16 (medium), 20 (large)

Performance





Same base model architecture

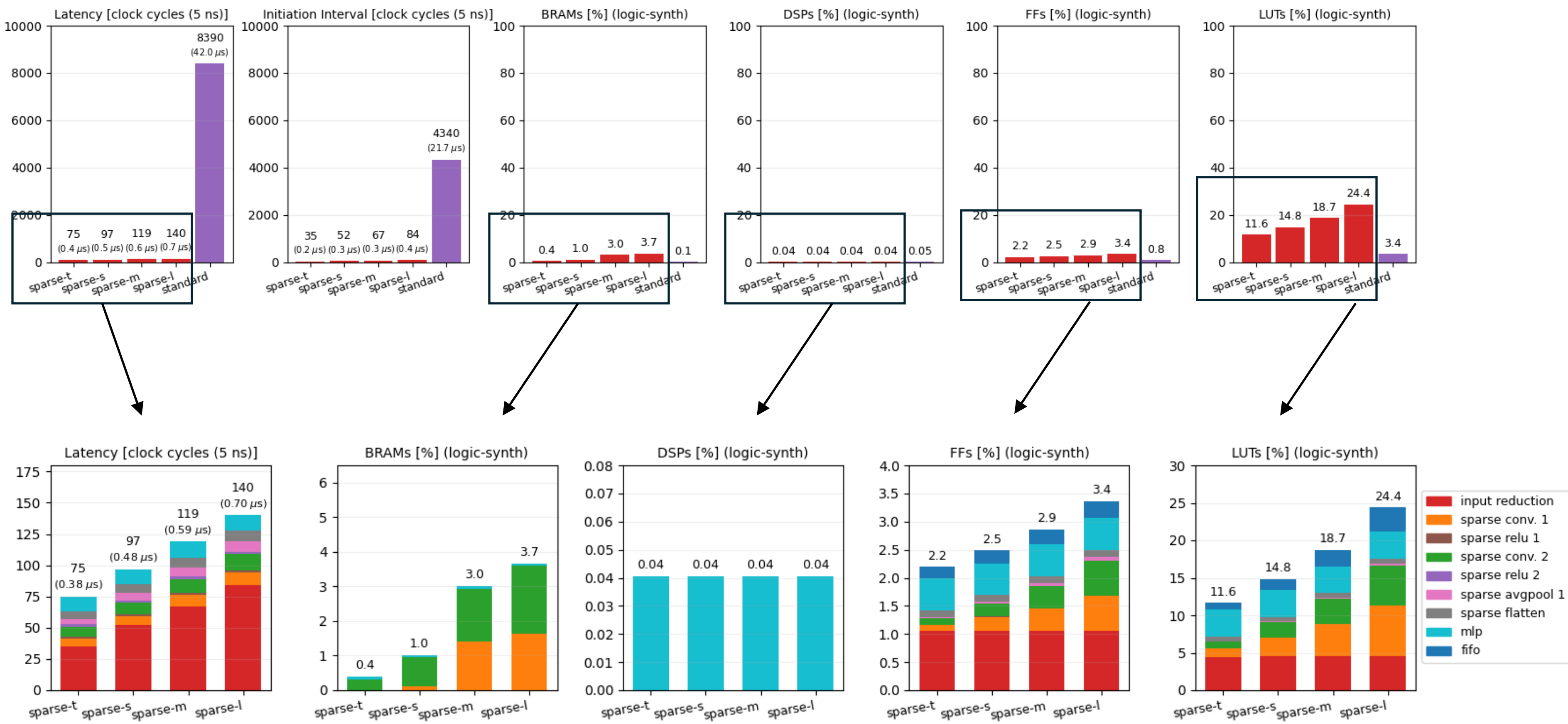
- **Standard** CNN [full]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 8$ [sparse-t]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 12$ [sparse-s]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 16$ [sparse-m]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 20$ [sparse-l]

Model latency and resource utilization

Standard CNN: io_stream, fully serialized
Sparse CNN: io_parallel, fully parallelized

Model bit width = 8

Standard CNN vs sparse CNNs



Per-module breakdown in sparse models

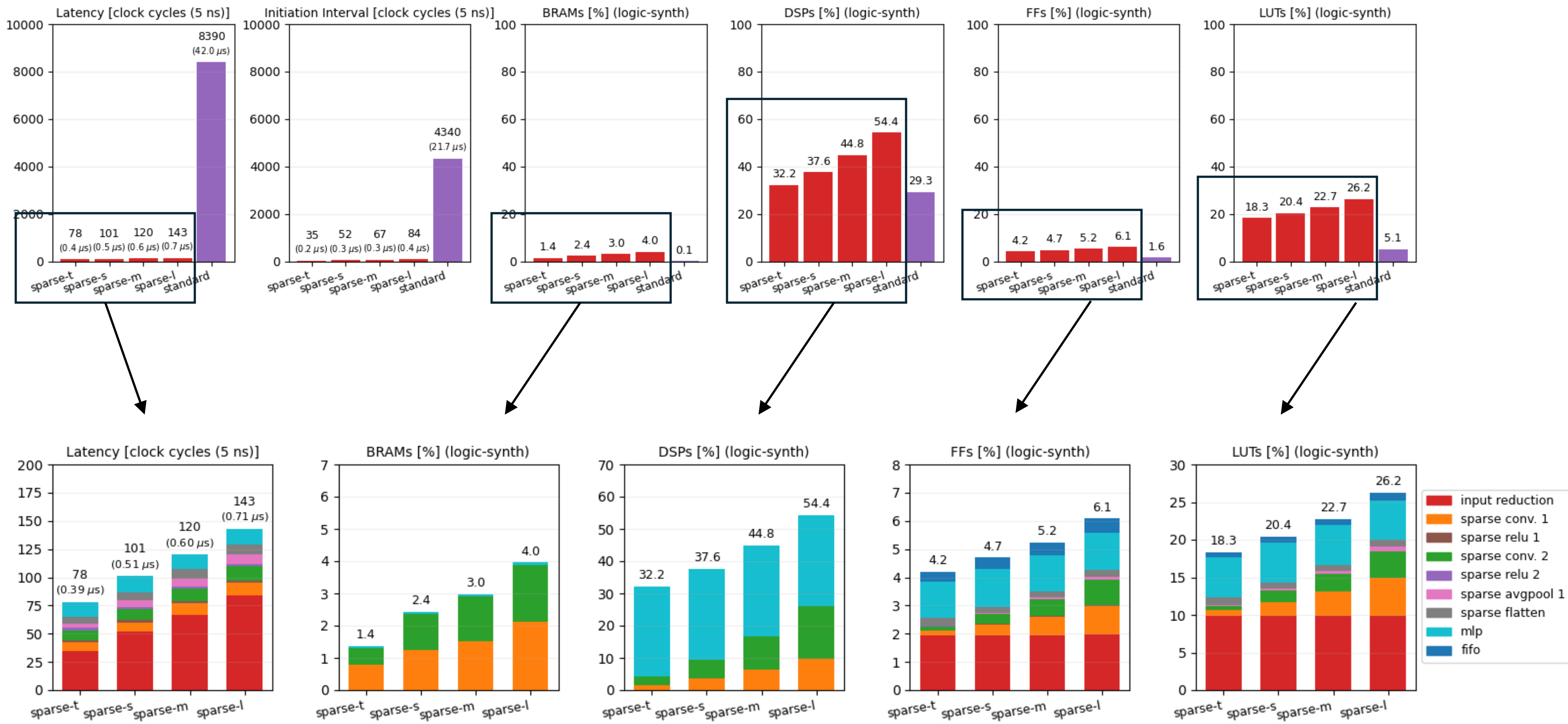
- Sparse CNNs are orders of magnitude faster than standard CNN of same base arch, given both are well within the available resources
- Now these sparse CNNs are fully parallelized, but one could potentially tune the reuse factors to lower the resources

Model latency and resource utilization

Standard CNN: io_stream, fully serialized
Sparse CNN: io_parallel, fully parallelized

Model bit width = 16

Standard CNN vs sparse CNNs

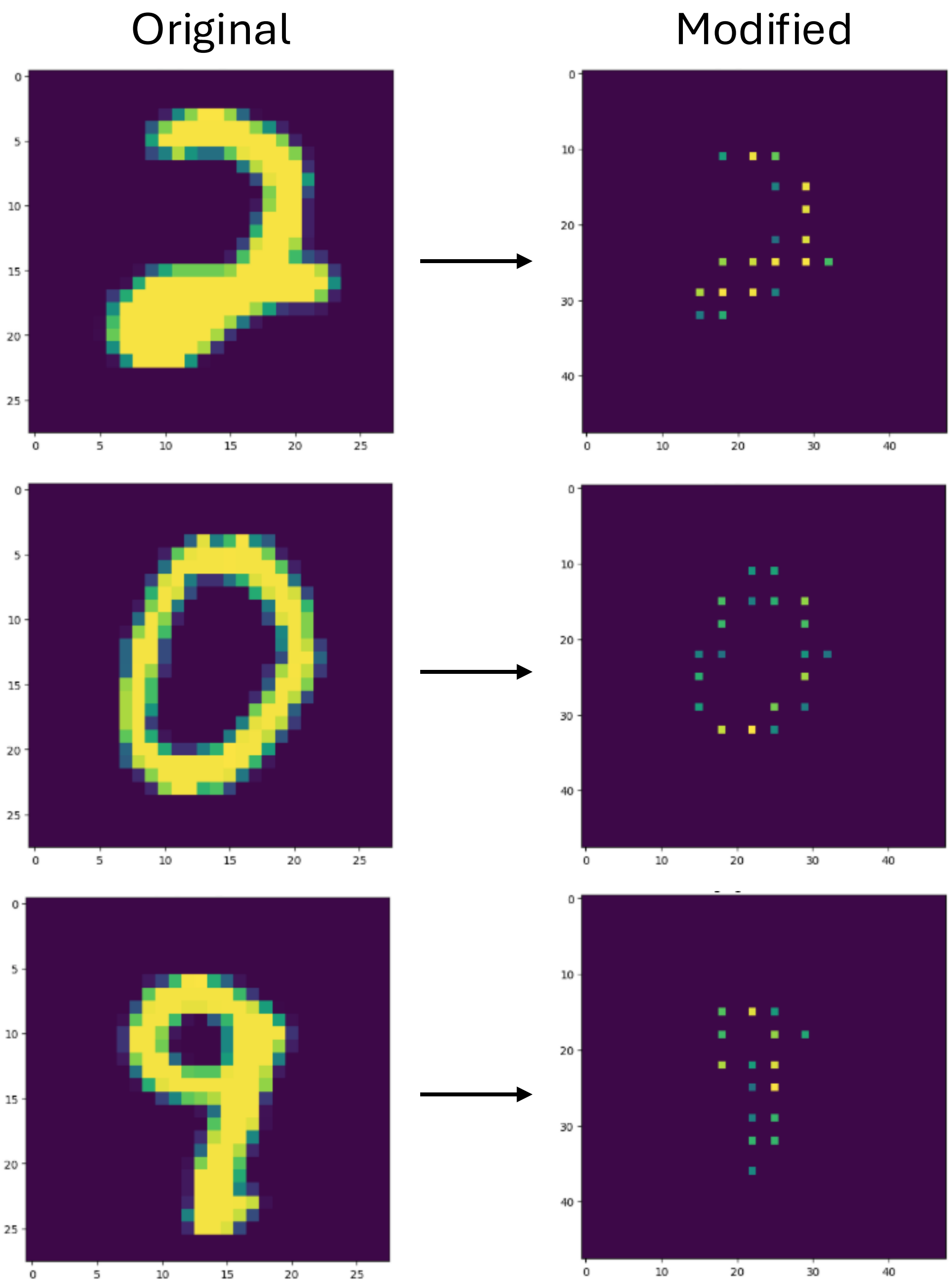


Per-module breakdown in sparse models

- Sparse CNNs are orders of magnitude faster than standard CNN of same base arch, given both are well within the available resources
- Now these sparse CNNs are fully parallelized, but one could potentially tune the reuse factors to lower the resources

Dataset demo: sparsified MNIST

Dataset



Original MNIST dataset

- 28*28*1 input pixels

Modified to emulate sparse datasets

- Pooled + padded+ inflated
- 48*48*1 input pixels

Our python library to support sparse CNN training

Standard CNN (QKeras layers)

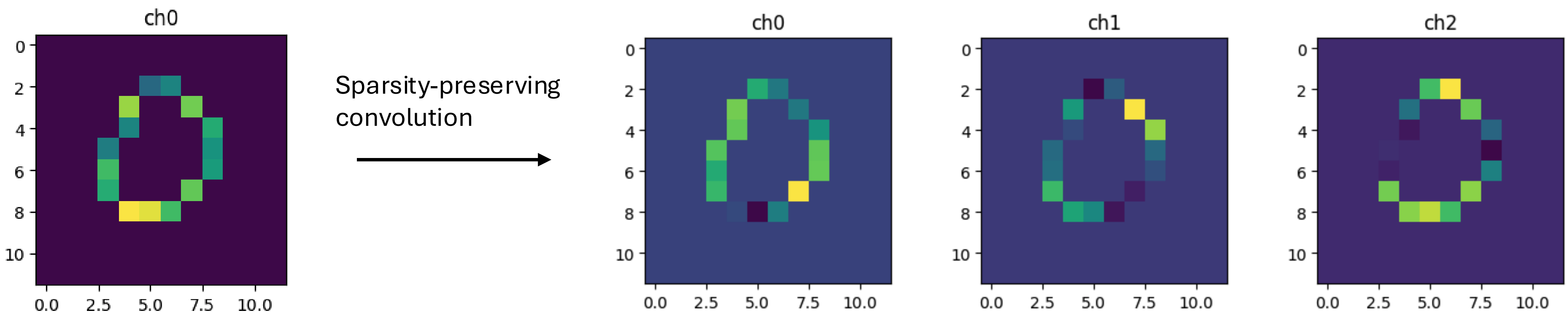
```
Model: "cnn_full"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 x_in (InputLayer)           [(None, 48, 48, 1)]       0

 conv1 (QConv2D)             (None, 48, 48, 1)         50

 relu1 (QActivation)         (None, 48, 48, 1)         0

 pool1 (AveragePooling2D)    (None, 12, 12, 1)         0

 conv2 (QConv2D)             (None, 12, 12, 3)         78

 relu2 (QActivation)         (None, 12, 12, 3)         0

 pool2 (AveragePooling2D)    (None, 6, 6, 3)           0

 flatten (Flatten)           (None, 108)               0

 dense1 (QDense)             (None, 36)                3924

 relu3 (QActivation)         (None, 36)                0

 dense2 (QDense)             (None, 10)                370

 softmax (Activation)        (None, 10)                0

=================================================================
Total params: 4422 (17.27 KB)
Trainable params: 4422 (17.27 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

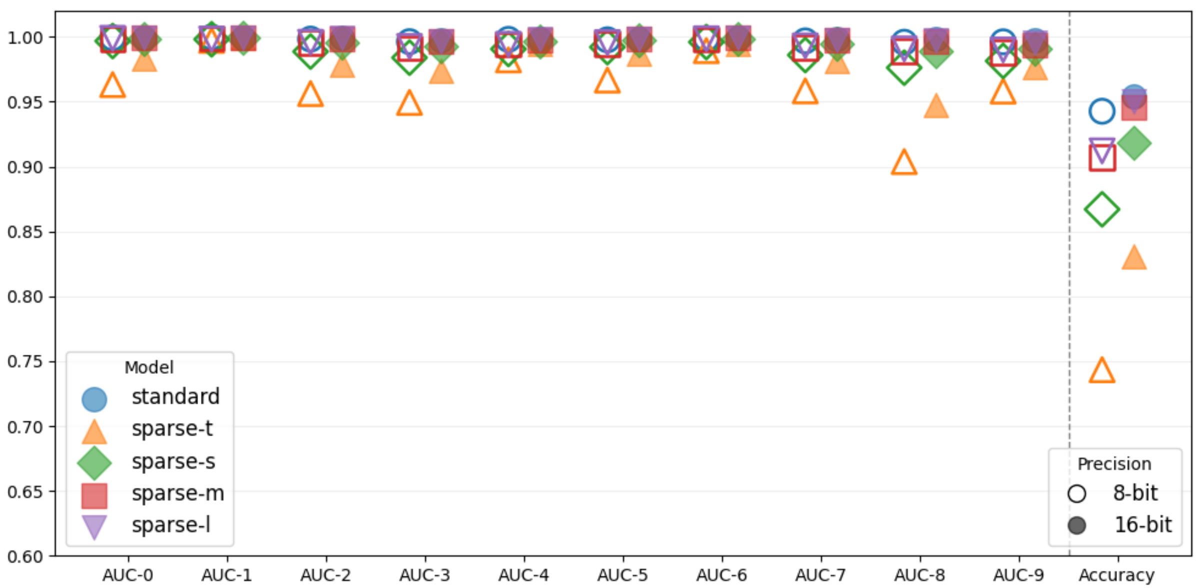
Sparse CNN (our SparsePixels layers)

```
Model: "cnn_sparse"
__________________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================================
 x_in (InputLayer)           [(None, 48, 48, 1)]          0         []

 input_reduce (InputReduce)  ((None, 48, 48, 1),          0         ['x_in[0][0]']
                              (None, 48, 48, 1))

 conv1 (QConv2DSparse)       (None, 48, 48, 1)            50        ['input_reduce[0][0]',
                                                                     'input_reduce[0][1]']

 relu1 (QActivation)         (None, 48, 48, 1)            0         ['conv1[0][0]']

 pool1 (AveragePooling2DSpa  ((None, 12, 12, 1),          0         ['relu1[0][0]',
 rse)                         (None, 12, 12, 1))                     'input_reduce[0][1]']

 conv2 (QConv2DSparse)       (None, 12, 12, 3)            78        ['pool1[0][0]',
                                                                     'pool1[0][1]']

 relu2 (QActivation)         (None, 12, 12, 3)            0         ['conv2[0][0]']

 pool2 (AveragePooling2DSpa  ((None, 6, 6, 3),            0         ['relu2[0][0]',
 rse)                         (None, 6, 6, 1))                       'pool1[0][1]']

 flatten (Flatten)           (None, 108)                  0         ['pool2[0][0]']

 dense1 (QDense)             (None, 36)                   3924      ['flatten[0][0]']

 relu3 (QActivation)         (None, 36)                   0         ['dense1[0][0]']

 dense2 (QDense)             (None, 10)                   370       ['relu3[0][0]']

 softmax (Activation)        (None, 10)                   0         ['dense2[0][0]']

==================================================================================================
Total params: 4422 (17.27 KB)
Trainable params: 4422 (17.27 KB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________________________
```

- Same base model architecture
- For sparse CNN, consider four $N_{\text{active}}^{\text{max}}$: 8 (tiny), 12 (small), 16 (medium), 20 (large)

Performance





Same base model architecture

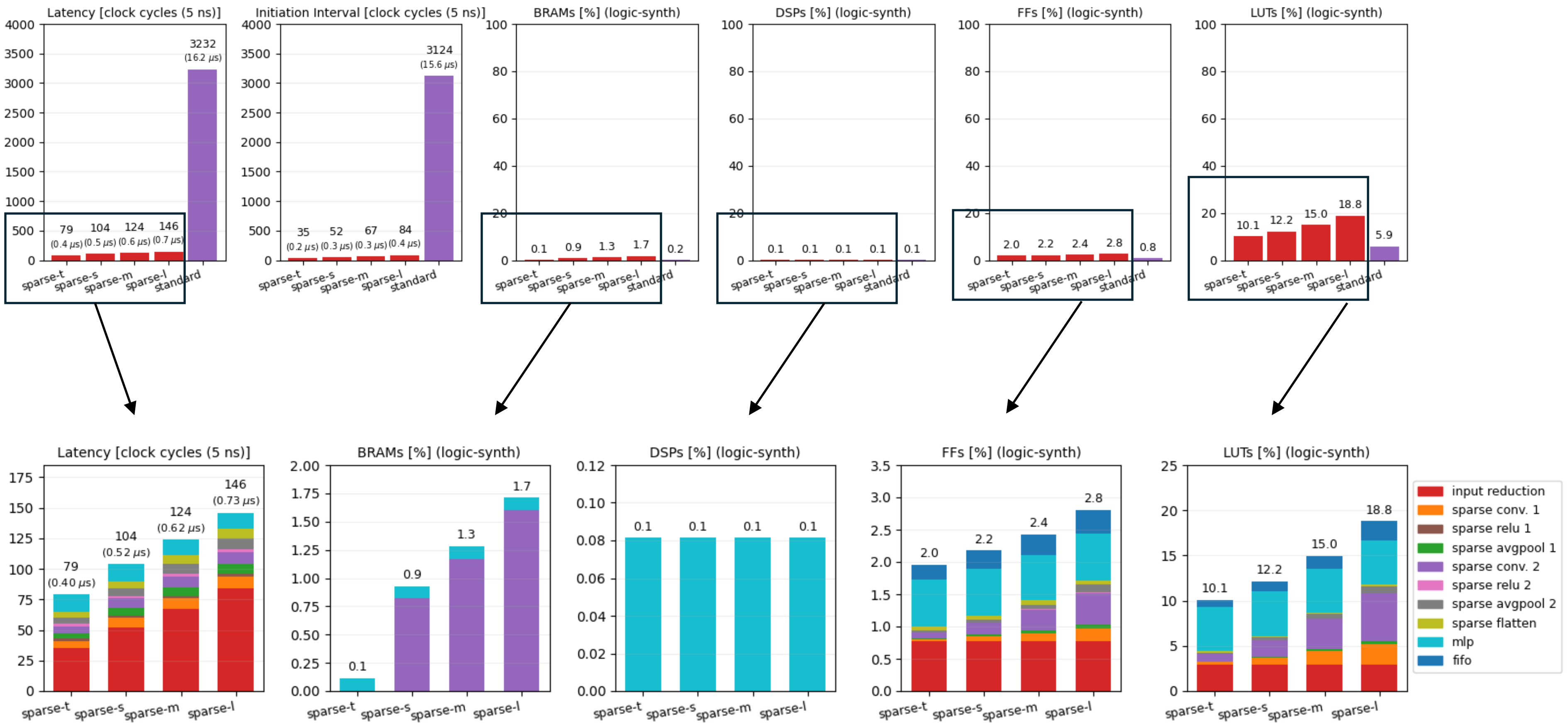
- **Standard** CNN [full]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 8$ [sparse-t]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 12$ [sparse-s]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 16$ [sparse-m]
- **Sparse** CNN with $N_{\text{active}}^{\text{max}} = 20$ [sparse-l]

Model latency and resource utilization

Standard CNN: io_stream, fully serialized
Sparse CNN: io_parallel, fully parallelized

Model bit width = 8

Standard CNN vs sparse CNNs



Per-module breakdown in sparse models

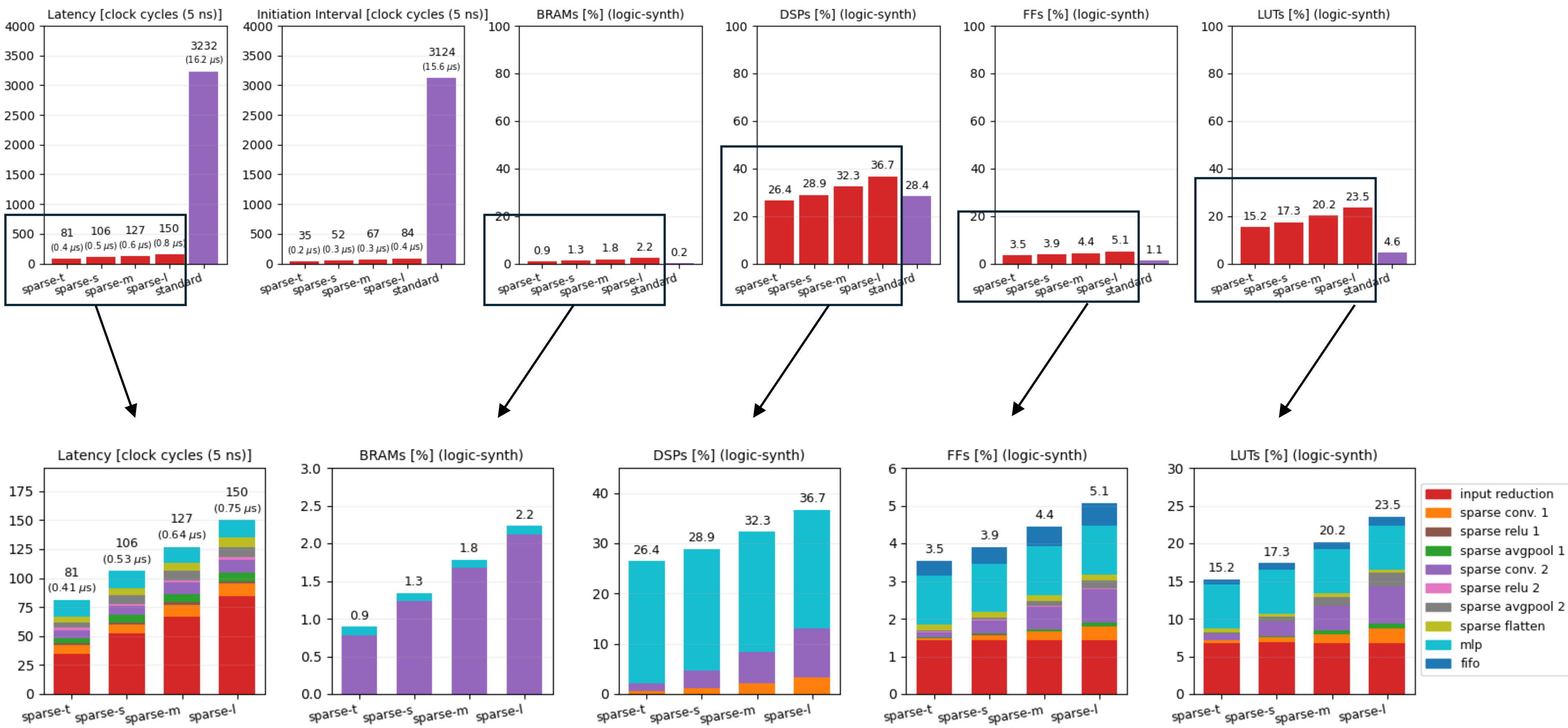
- Sparse CNNs are orders of magnitude faster than standard CNN of same base arch, given both are well within the available resources
- Now these sparse CNNs are fully parallelized, but one could potentially tune the reuse factors to lower the resources

Model latency and resource utilization

Standard CNN: io_stream, fully serialized
Sparse CNN: io_parallel, fully parallelized

Model bit width = 16

Standard CNN vs sparse CNNs



Per-module breakdown in sparse models

- Sparse CNNs are orders of magnitude faster than standard CNN of same base arch, given both are well within the available resources
- Now these sparse CNNs are fully parallelized, but one could potentially tune the reuse factors to lower the resources