FINN HLS

Lecturer: Hua-Yang Weng

Date: 2022/09/04

[FPGA'17: FINN: A Framework for Fast, Scalable Binarized Neural Network Inference]
(https://arxiv.org/abs/1612.07119)

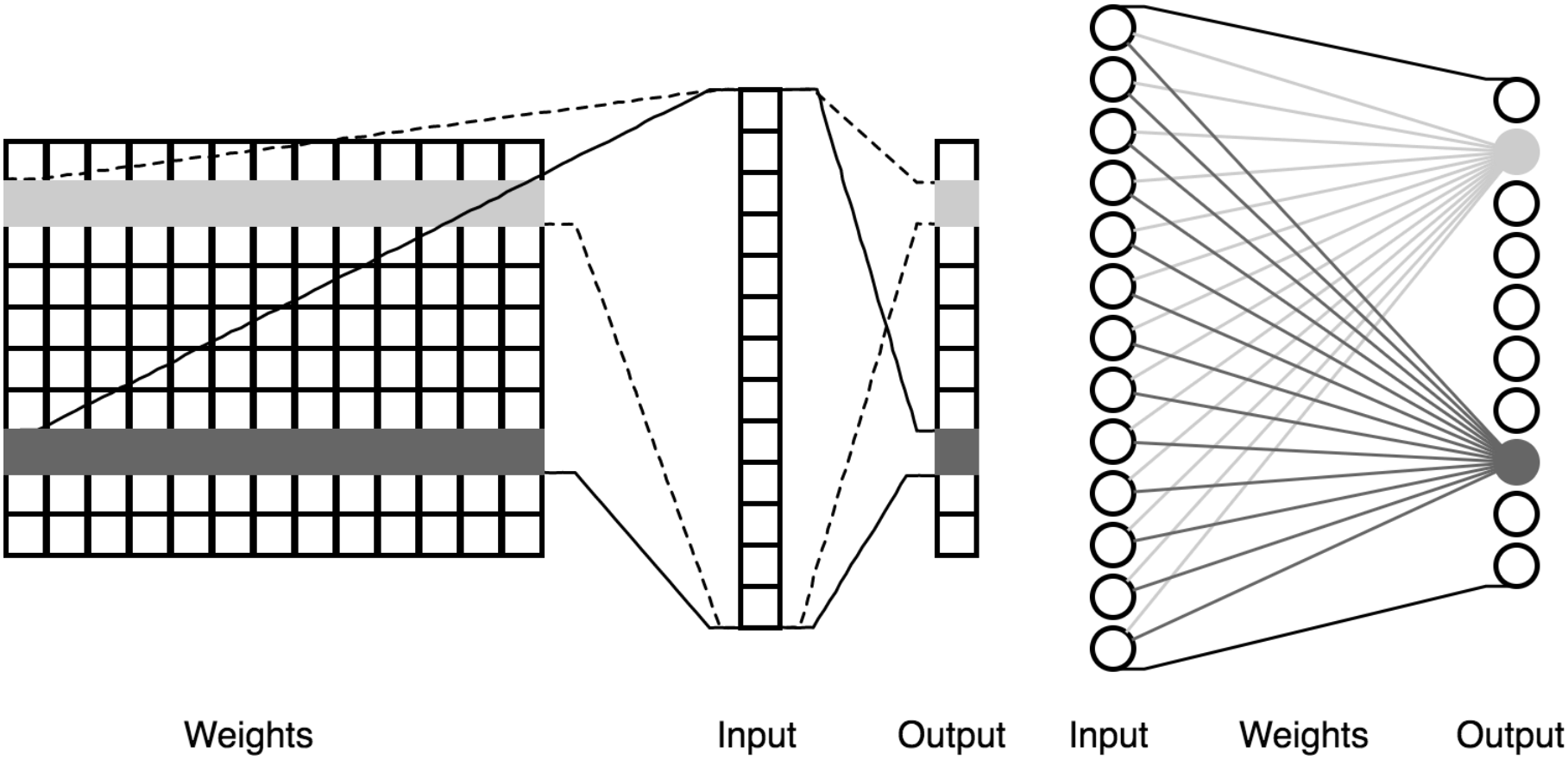
Outline

- Matrix-Vector Accumulation Unit
- Pooling Layer
- Convolution Layer
- FIFO

Outline

- Matrix-Vector Accumulation Unit
- Pooling Layer
- Convolution Layer
- FIFO

Fully Connected Layer

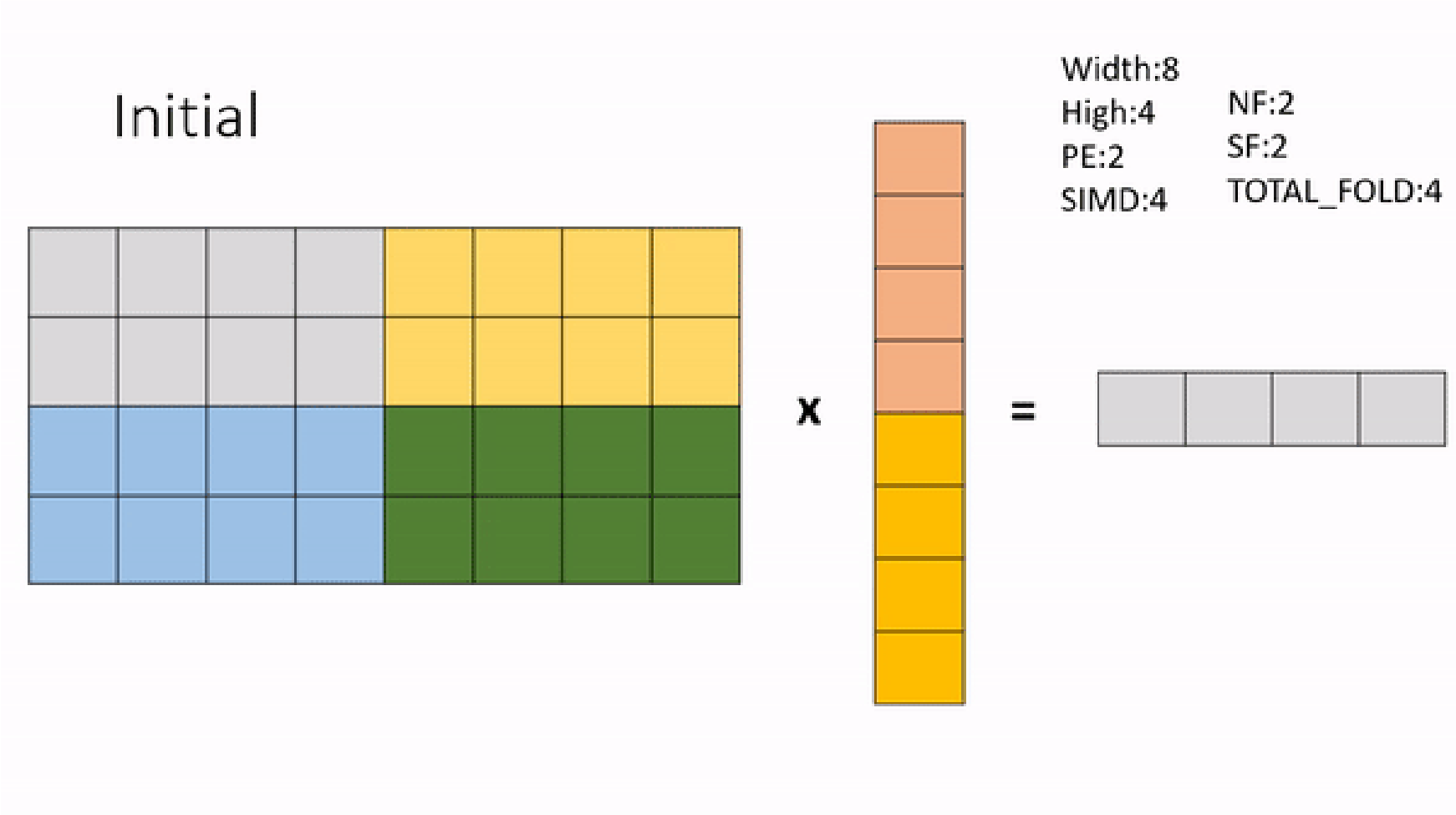


Matrix-Vector Accumulation Unit

- The function performs the multiplication between a weight matrix and the input activation vector.

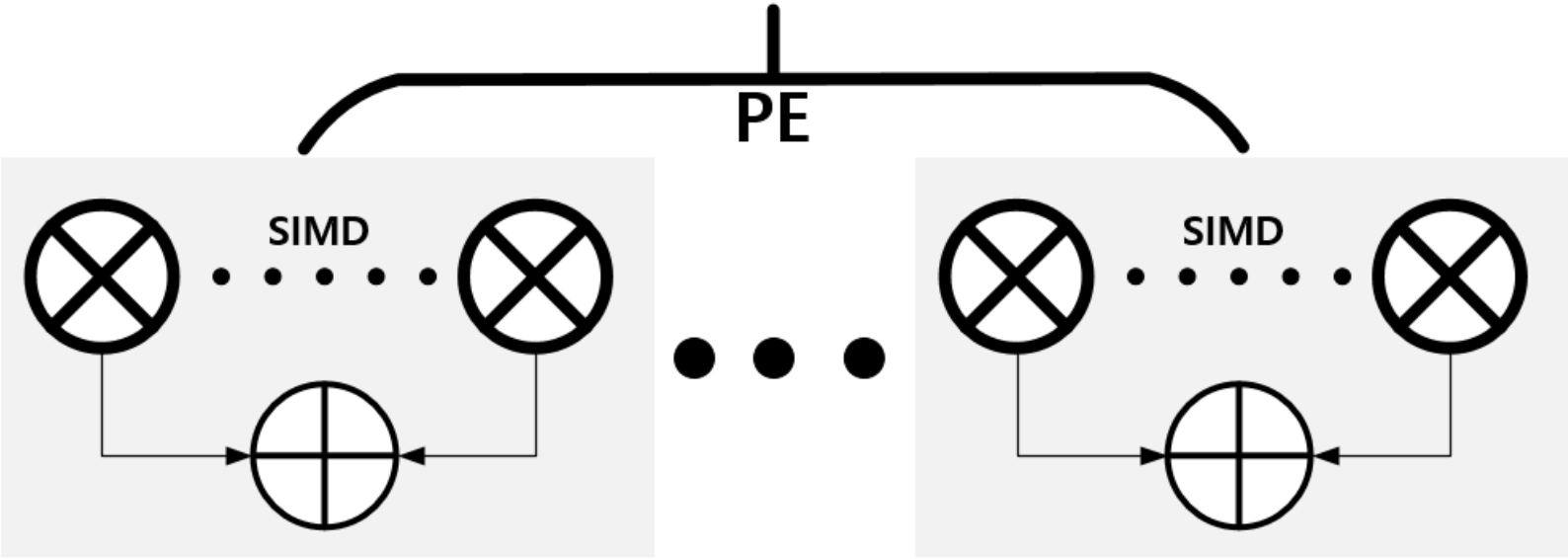
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 30 \\ 40 \\ 50 \end{bmatrix}$$

Example



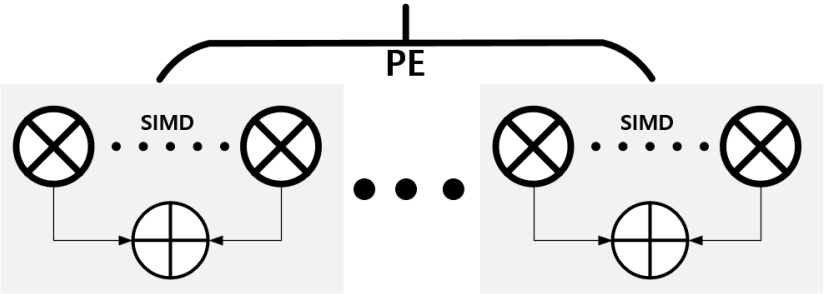
Block Diagram

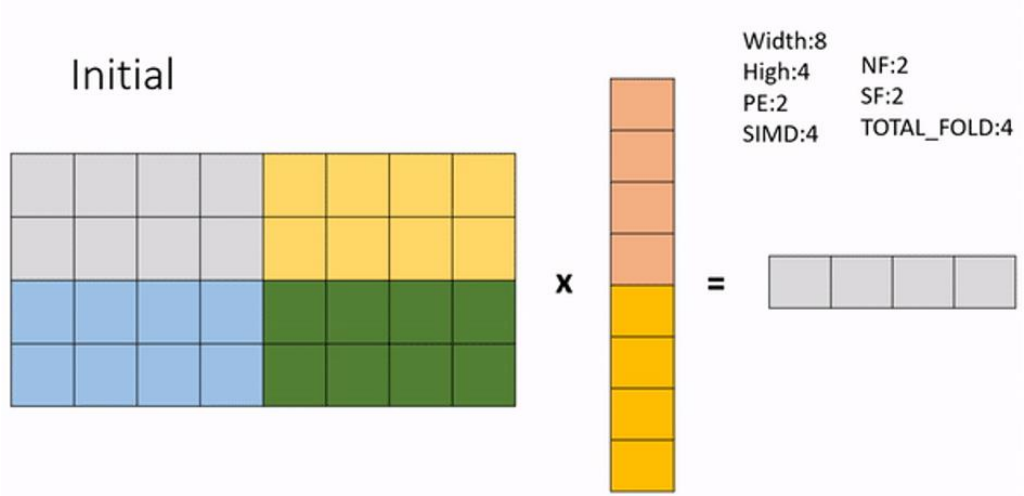
- According to the unroll factor(PE、SIMD) set in python code, the compiler will generate corresponding amount of hardware.



I/O interface

- hls::stream<SIMD*BITWIDTH> &in
- hls::stream<PE*BITWIDTH> &out
- TW const &weights
  - PE*SIMD*BITWIDTH





Constant variables

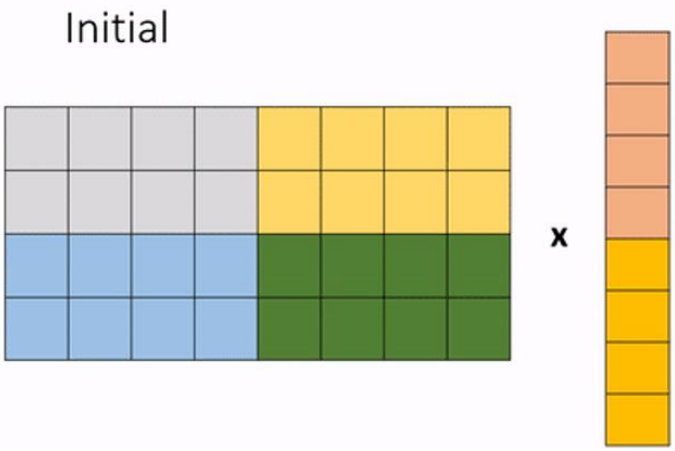
- unsigned const NF = MatrixH / PE;
  - *# of Cycles for the **PEs** to **compute all rows** of the SIMD inputs*
- unsigned const SF = MatrixW / SIMD;
  - *# of Cycles for the **SIMD** inputs to **distribute** all of the workloads of **a row***
- unsigned const TOTAL_FOLD = NF * SF;
  - *# of Cycles for this Matrix-Vector Product*

HLS Code (1/)

- MVAU reads the input activations from input port, and store them into internal buffer

```
    // input vector buffers
    TI  inputBuf[SF];
    for(unsigned  i = 0; i < reps * TOTAL_FOLD; i++) {
#pragma HLS PIPELINE II=1
      TI  inElem;

      if(nf == 0) {
        // read input from stream
        inElem = in.read();
        // store in appropriate buffer for reuse
        inputBuf[sf] = inElem;
      }
      else {
        // reuse buffered input
        inElem = inputBuf[sf];
      }
```

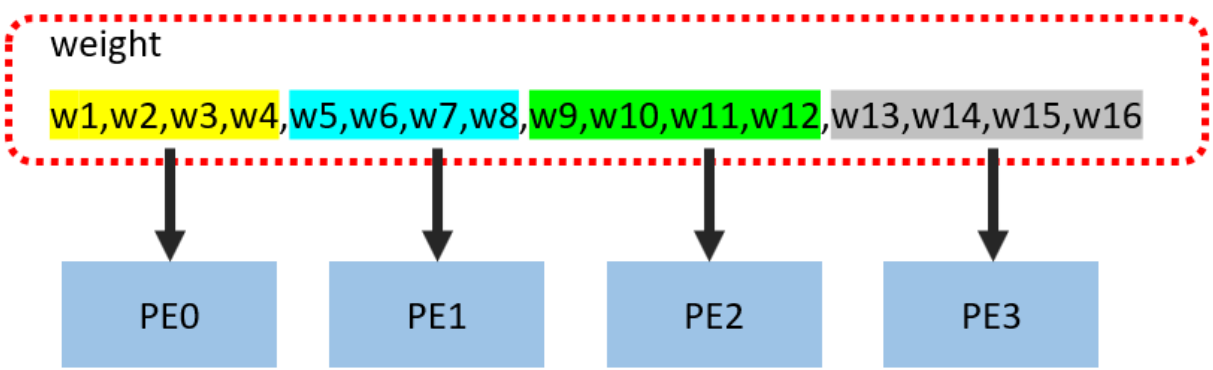
hls::stream<SIMD*BITWIDTH> &in

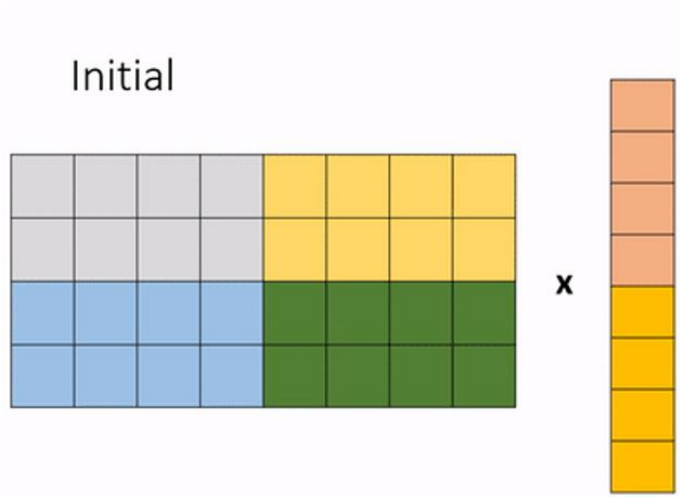


HLS Code (2/)

- MVAU reads the weights and packed them.
  - TW const &weights -> PE*SIMD*BITWIDTH

```
    // read from the parameter stream
    W_packed = weight.read();
    for (unsigned pe = 0; pe < PE; pe++) {
#pragma HLS UNROLL
      w.m_weights[pe] = W_packed((pe+1)*SIMD*TW::width-1,pe*SIMD*TW::width);
    }
```

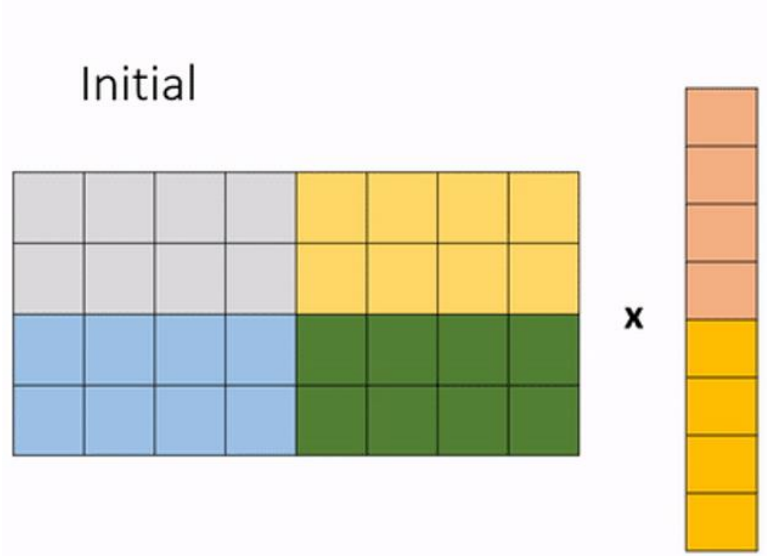




HLS Code (3/)

- Set the accumulation register value to zero.

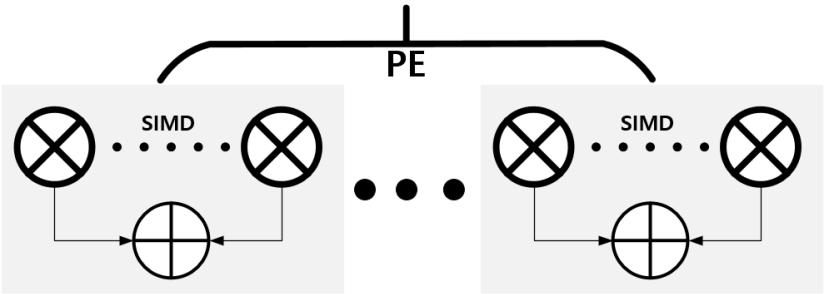
```
    // Threshold Initialisation
    if(sf == 0) {
      for(unsigned pe = 0; pe < PE; pe++) {
#pragma HLS UNROLL
      accu[0][pe] = activation.init(nf, pe);
      }
    }
```

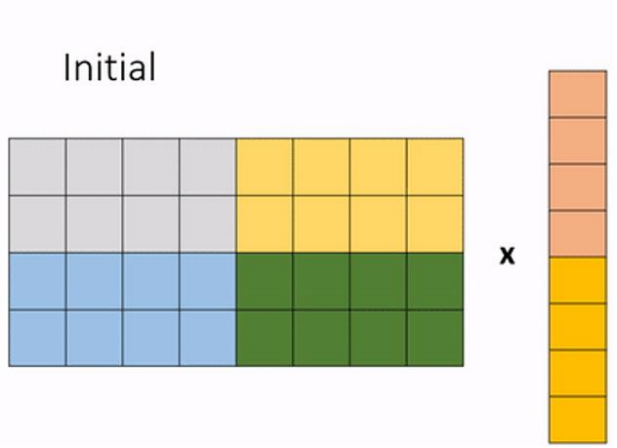


HLS Code (4/)

- Uses MAC unit to calculate
  - # of MAC unit depends on PE value.

```
    // compute matrix-vector product for each processing element
    for(unsigned  pe = 0; pe < PE; pe++) {
#pragma HLS UNROLL
      auto const  act = TSrcI()(inElem, 0);
      auto const  wgt = TWeightI()(w[pe]);
      //auto const  wgt = w[pe];
      accu[0][pe] = mac<SIMD>(accu[0][pe], wgt, act, r, 0);
    }
```



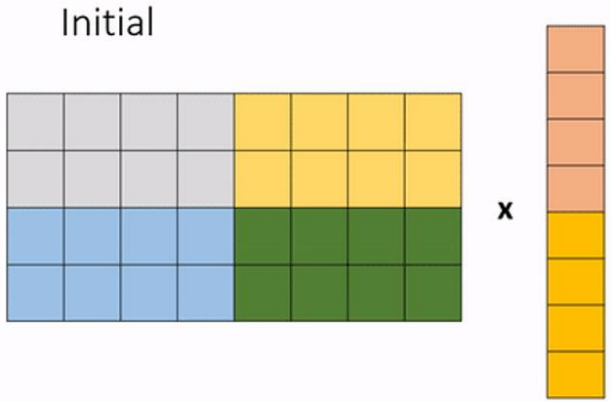


HLS Code (5/)

- Continues with the partial sum if sf < SF
  - Feed the results into activation function.
  - Write the activation function output into output buffer.

```
    ++tile;
    if(++sf == SF) {
      // produce output and clear accumulators
      auto  outElem = TDstI().template operator()<TO>();
      for (unsigned  pe = 0; pe < PE; pe++) {
#pragma HLS UNROLL
      outElem(pe,0,1) = activation.activate(nf, pe, accu[0][pe]);
          }

      out.write(outElem);
```

hls::stream<PE*BITWIDTH> &out



StreamingFCLayer_Batch_3

```
// includes for network parameters
#include "weights.hpp"
#include "activations.hpp"
#include "mvau.hpp"

// defines for network parameters
#define MW1 64
#define MH1 10

#define SIMD1 8
#define PE1 10
#define WMEM1 8

#define TMEM1 0
#define numReps 1
#define WP1 1


void StreamingDataflowPartition_1_StreamingFCLayer_Batch_3(
                    hls::stream<ap_uint<8>> &in0,
                    hls::stream<ap_uint<80>> &weights,
                    hls::stream<ap_uint<80>> &out
                    )
{
#pragma HLS INTERFACE axis port=in0
#pragma HLS INTERFACE axis port=out
#pragma HLS stream depth=64 variable=in0
#pragma HLS stream depth=10 variable=out
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=weights
#pragma HLS stream depth=8 variable=weights
Matrix_Vector_Activate_Stream_Batch<MW1, MH1, SIMD1, PE1, Recast<XnorMul>, Slice<ap_uint<8>>, Identity, ap_uint<1> >
            (in0, out, weights, PassThroughActivation<ap_uint<8>>(), numReps, ap_resource_lut());
}
```

- hls::stream<SIMD*BITWIDTH> &in
- hls::stream<PE*BITWIDTH> &out
- TW const &weights
  - PE*SIMD*BITWIDTH

Last FC layer of mnist classification:

BITWIDTH_IN = 1
BITWIDTH_OUT = 8

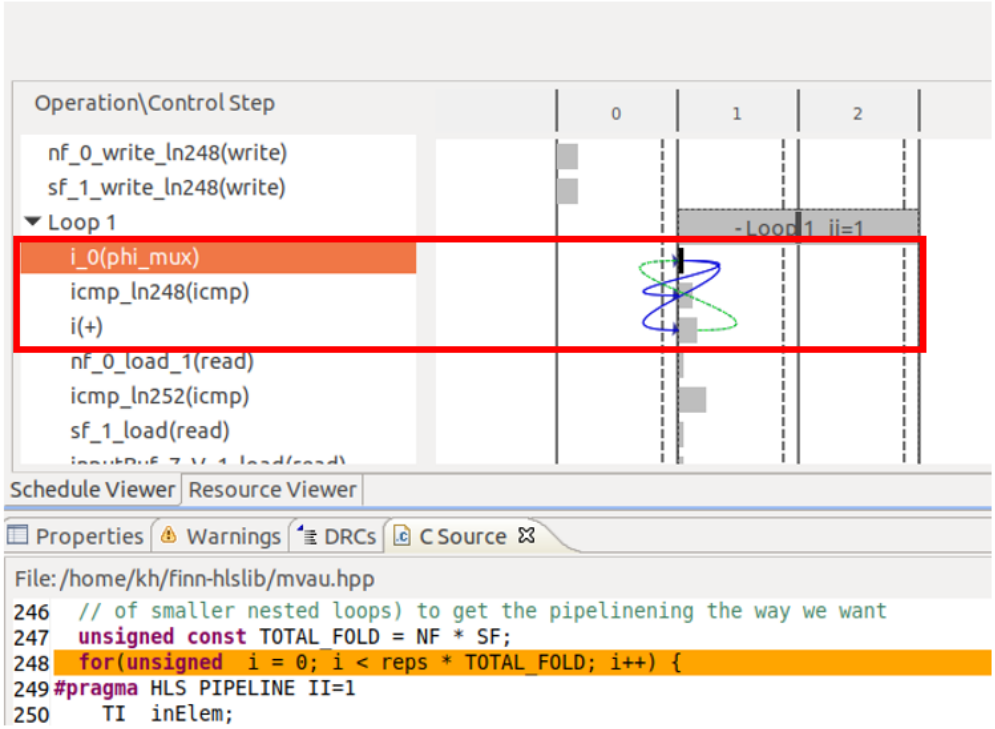
Latency Report

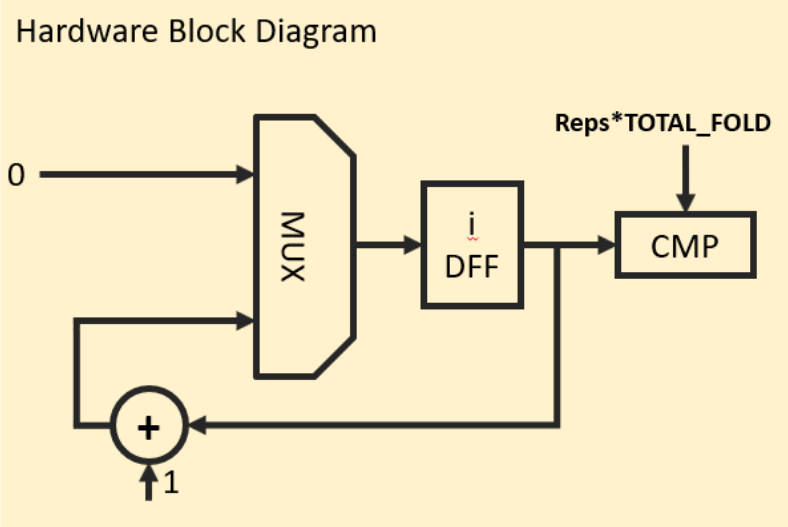
- The total fold is (MH/PE) * (MW/SIMD).
- (10/10)*(64/8)=8.

Performance Profile | Resource Profile

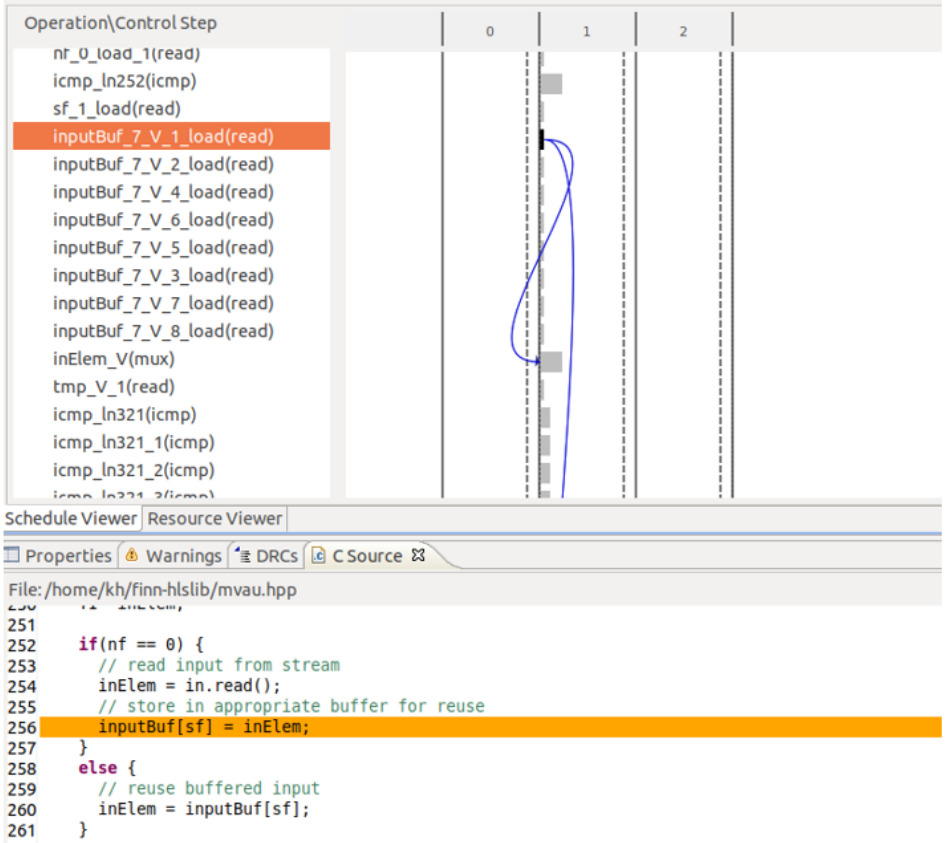
	Pipelined	Latency	Iteration Laten
Matrix_Vector_Activa	-	10	-
Loop 1	yes	8	2

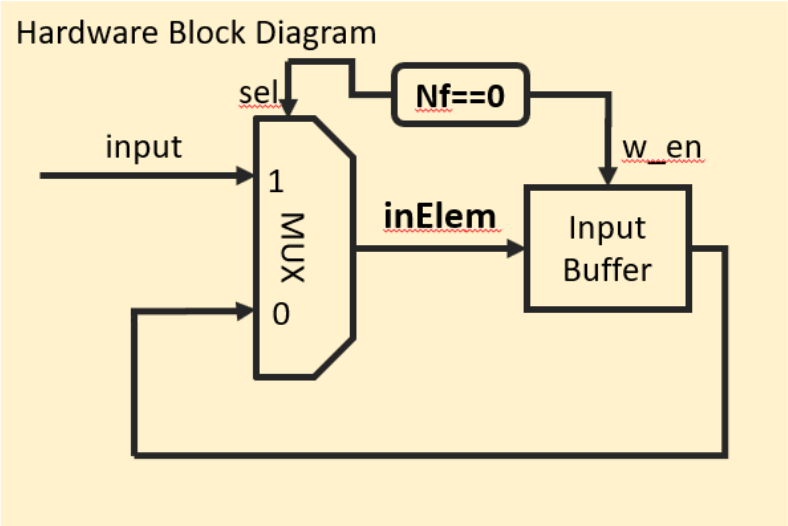
Schedule Viewer Trace(1/)



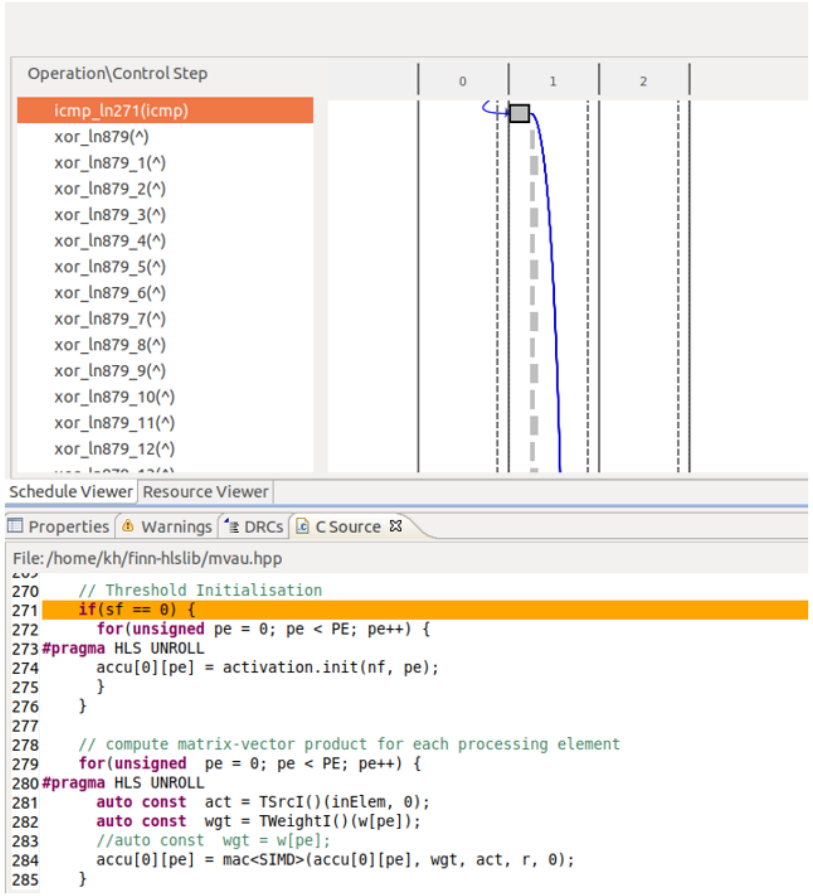


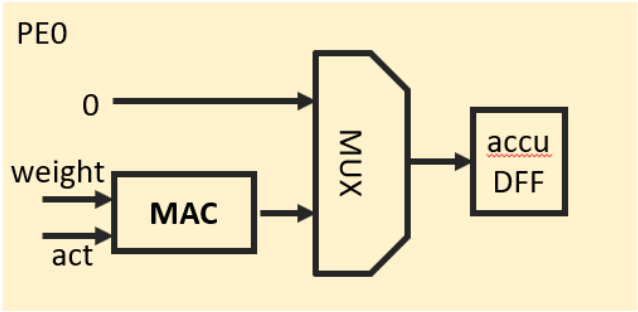
Schedule Viewer Trace(2/)

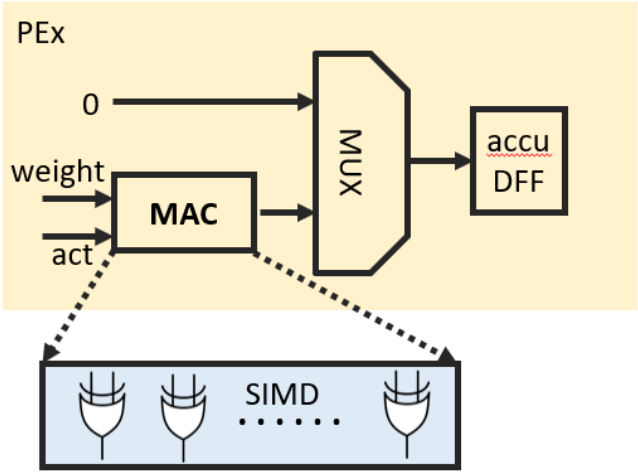




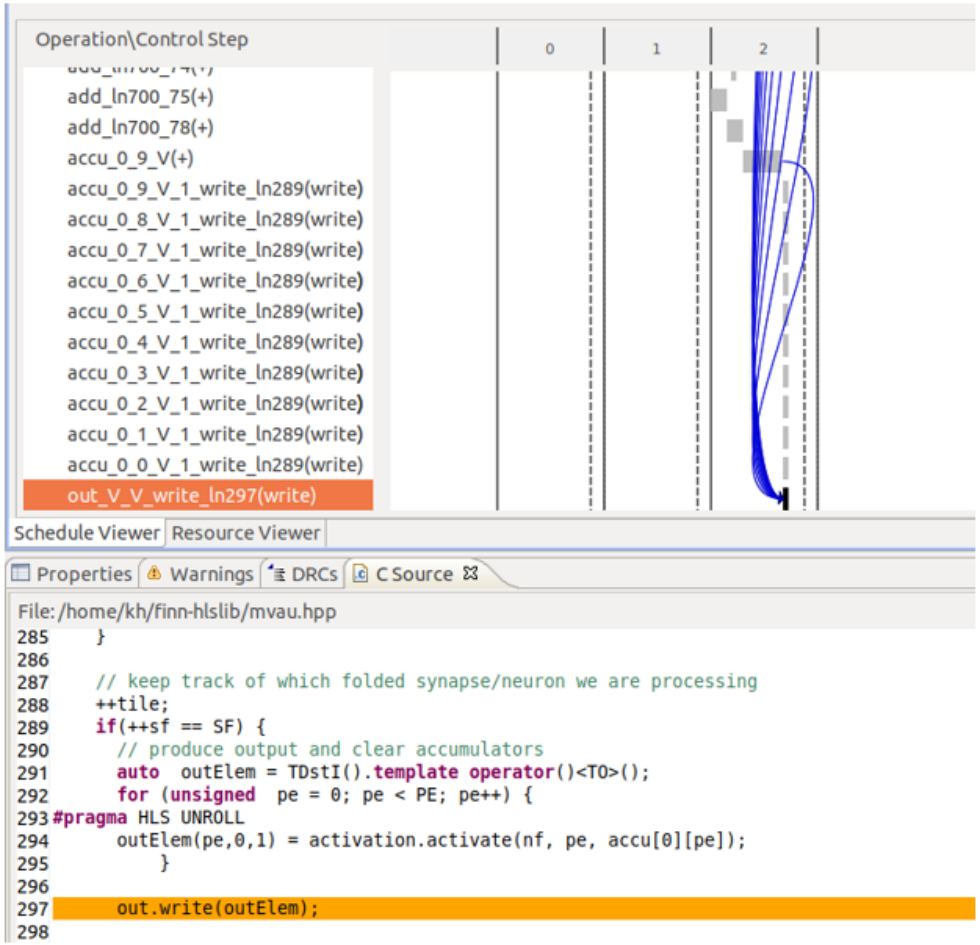
Schedule Viewer Trace(3/)

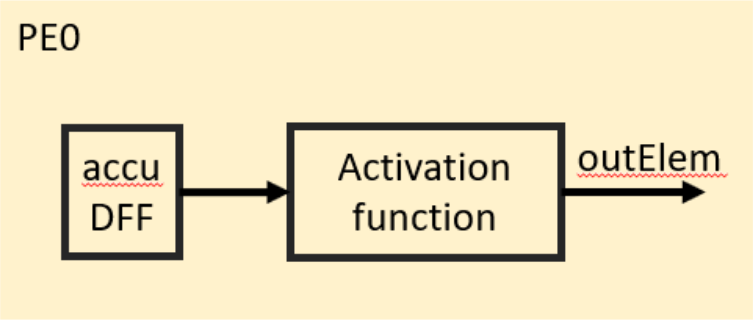


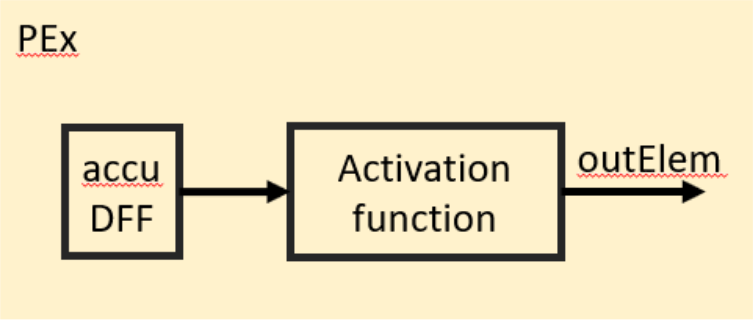




Schedule Viewer Trace(4/)







Outline

- Matrix-Vector Accumulation Unit
- Pooling Layer
- Convolution Layer
- FIFO

Max Pooling Layer

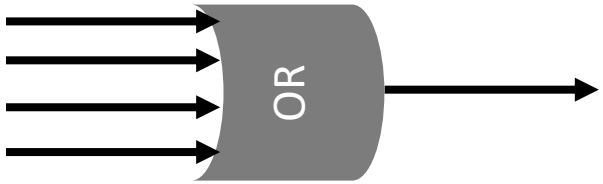
12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

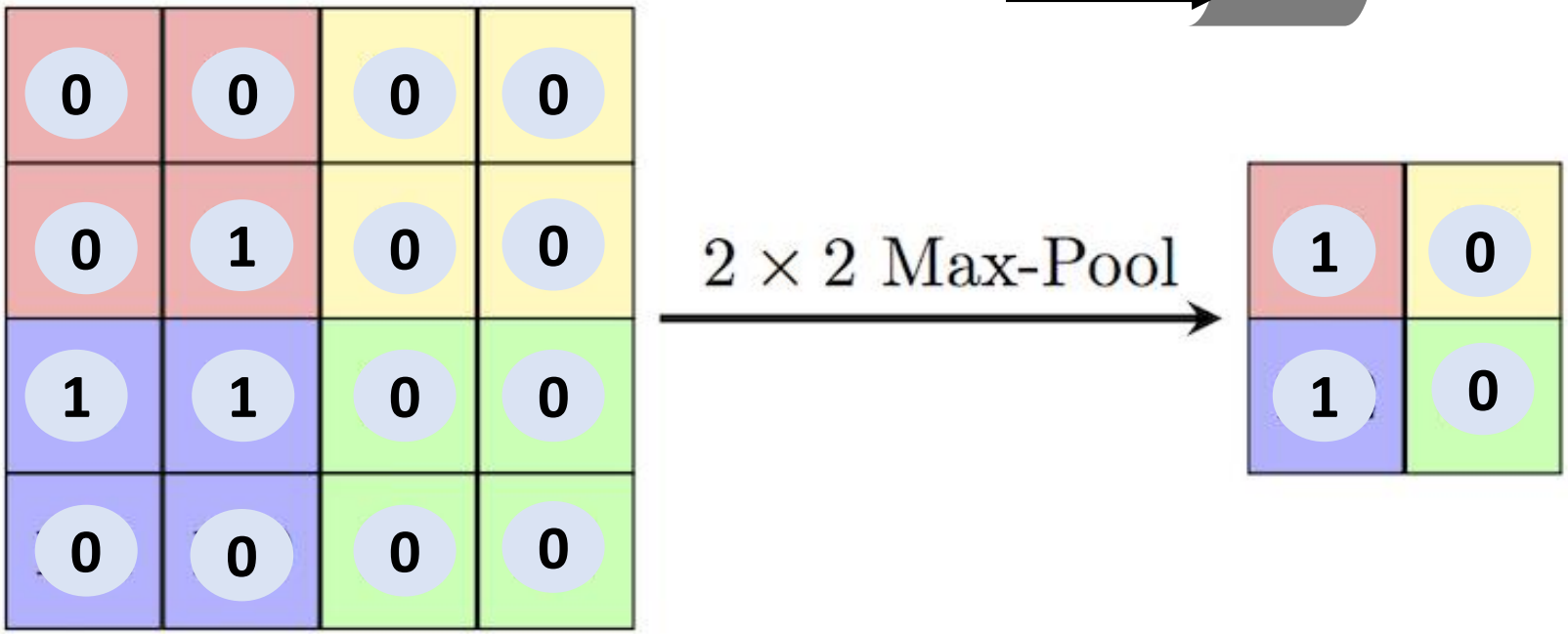
2×2 Max-Pool $\longrightarrow$

20	30
112	37

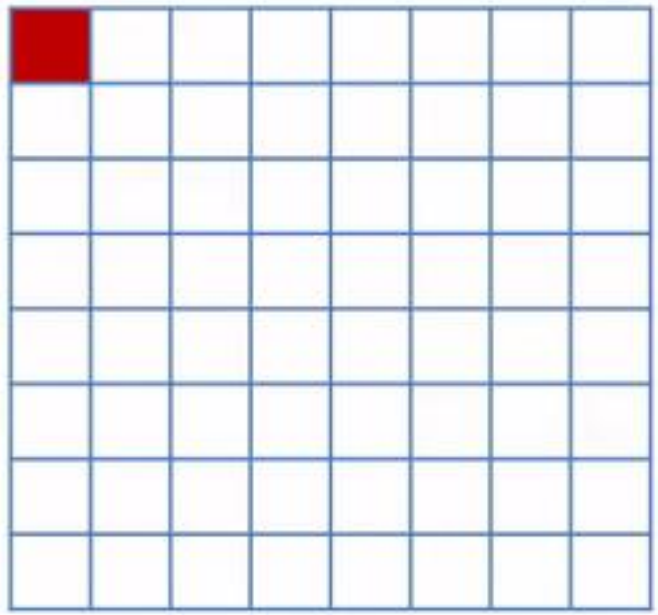
Binary Max Pooling Layer

We can use OR Gate to implement BNN-version.



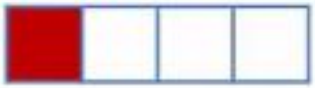


Example

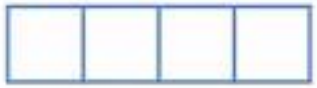


Input Feature map: 8x8
Max pooling kernel: 2x2

Line buffer

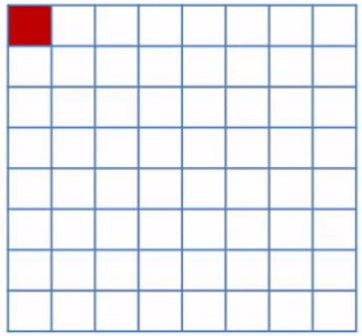


output FIFO



HLS Code

```
void StreamingMaxPool(stream<ap_uint<NumChannels> > & in,
        stream<ap_uint<NumChannels> > & out) {
  CASSERT_DATAFLOW(ImgDim % PoolDim == 0);
  // need buffer space for a single maxpooled row of the image
  ap_uint<NumChannels> buf[ImgDim / PoolDim];
  for(unsigned int i = 0; i < ImgDim / PoolDim; i++) {
#pragma HLS UNROLL
    buf[i] = 0;
  }

  for (unsigned int yp = 0; yp < ImgDim / PoolDim; yp++) {
    for (unsigned int ky = 0; ky < PoolDim; ky++) {
      for (unsigned int xp = 0; xp < ImgDim / PoolDim; xp++) {
#pragma HLS PIPELINE II=1
        ap_uint<NumChannels> acc = 0;
        for (unsigned int kx = 0; kx < PoolDim; kx++) {
          acc = acc | in.read();
        }
        // pool with old value in row buffer
        buf[xp] |= acc;
      }
    }
    for (unsigned int outpix = 0; outpix < ImgDim / PoolDim; outpix++) {
#pragma HLS PIPELINE II=1
      out.write(buf[outpix]);
      // get buffer ready for next use
      buf[outpix] = 0;
    }
  }
}
```



Input Feature map: 8x8
Max pooling kernel: 2x2

Line buffer

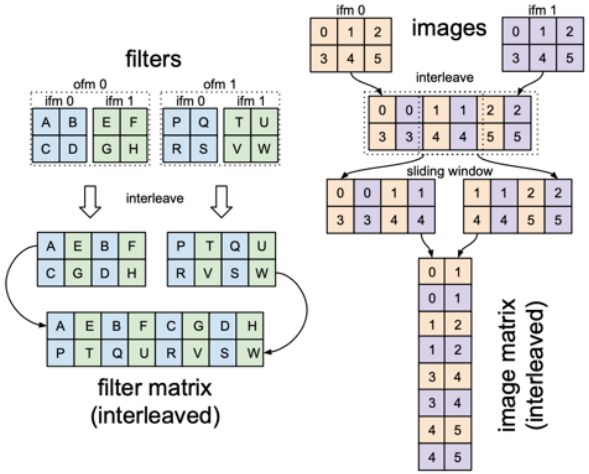
output FIFO

Outline

- Matrix-Vector Accumulation Unit
- Pooling Layer
- Convolution Layer
- FIFO

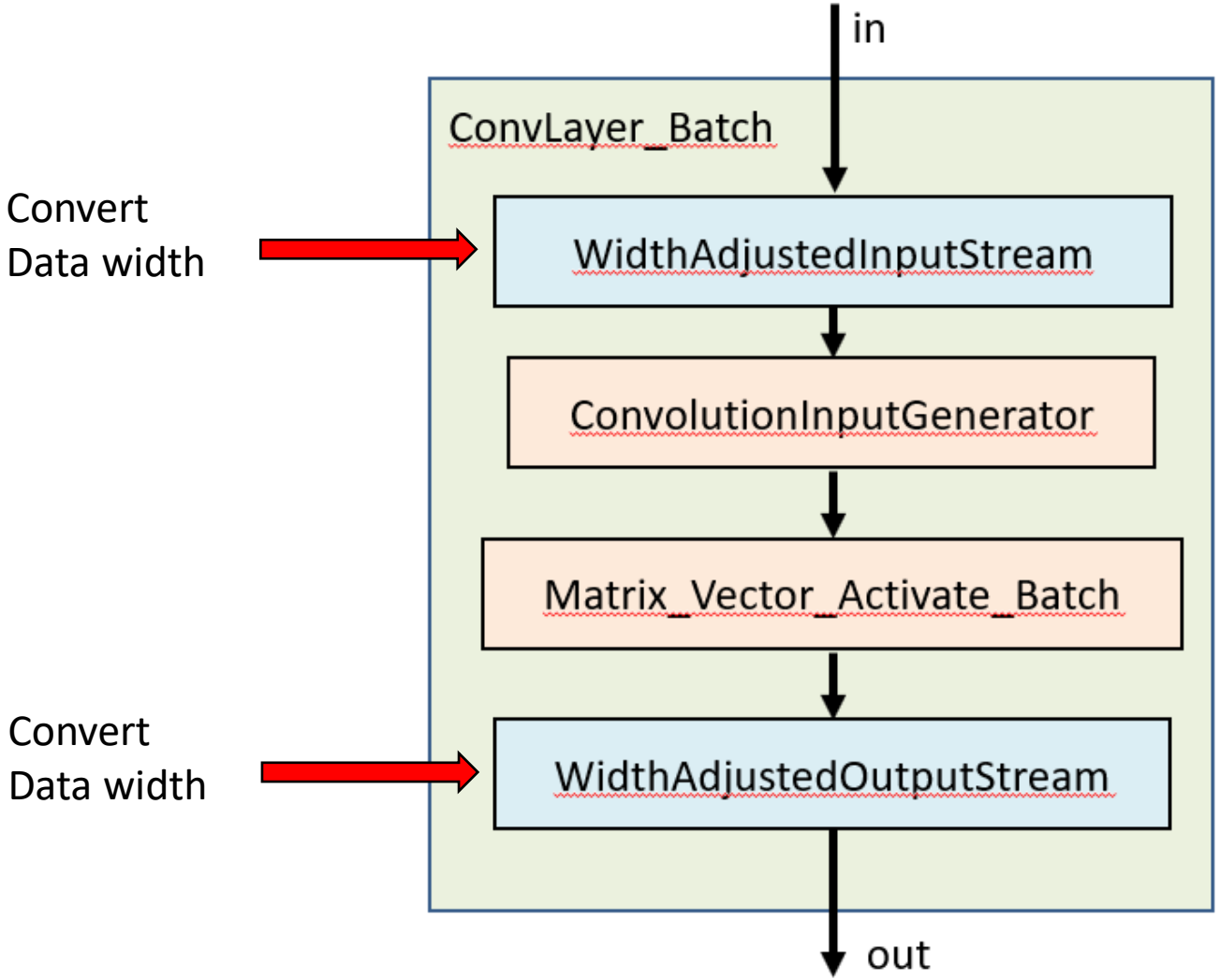
Convolution Layer

- Mainly consist of 2 core units
  - Convolution Input Generator



  - Matrix Vector Activation Batch

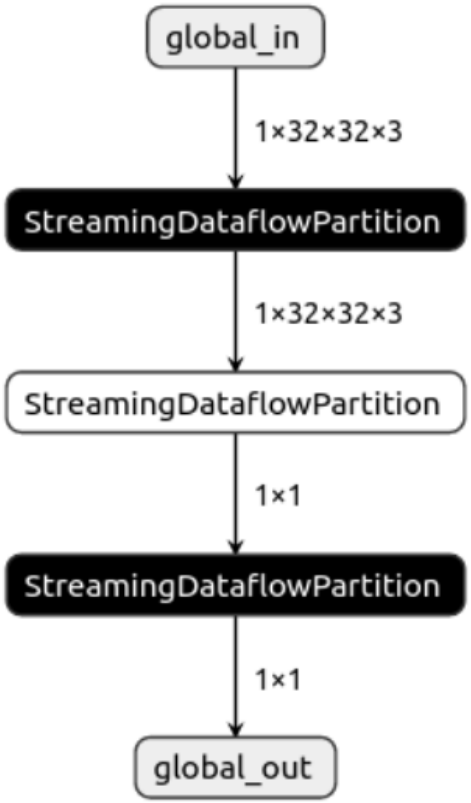
Convolution Layer: Block Diagram



ConvLayer_Batch( )

ConvLayer_batch( ):
ConvolutionInputGenerator (SWU)
Matrix_Vector_Activate_Batch(mvau)

```
template<
        unsigned int ConvKernelDim,
        unsigned int IFMChannels,
        unsigned int IFMDim,
        unsigned int OFMChannels,
        unsigned int OFMDim,

        unsigned int SIMD,              // number of SIMD lanes
        unsigned int PE,                // number of PEs

        typename TSrcI = Identity,      // redefine I/O interpretation as needed for input activations
        typename TDstI = Identity,      // redefine I/O interpretation as needed for output activations
        typename TWeightI = Identity,   // redefine I/O interpretation as needed for weigths

        int InStreamW, int OutStreamW,  // safely deducible (stream width must be int though!)
        typename TW,   typename TA,  typename R
>
    void ConvLayer_Batch(hls::stream<ap_uint<InStreamW>>  &in,
                         hls::stream<ap_uint<OutStreamW>> &out,
                         TW const        &weights,
                         TA const        &activation,
                         unsigned const   reps,
                         R const &r) {
    #pragma HLS INLINE
      unsigned const MatrixW = ConvKernelDim * ConvKernelDim * IFMChannels;
      unsigned const MatrixH = OFMChannels;
      unsigned const InpPerImage = IFMDim*IFMDim*IFMChannels*TSrcI::width/InStreamW;
      WidthAdjustedInputStream <InStreamW, SIMD*TSrcI::width, InpPerImage>  wa_in (in,  reps);
      WidthAdjustedOutputStream <PE*TDstI::width, OutStreamW, OFMDim * OFMDim * (OFMChannels / PE)>  mvOut (out,  reps);
      hls::stream<ap_uint<SIMD*TSrcI::width> > convInp("StreamingConvLayer_Batch.convInp");
      ConvolutionInputGenerator<ConvKernelDim, IFMChannels, TSrcI::width, IFMDim,
                OFMDim, SIMD,1>(wa_in, convInp, reps, ap_resource_dflt());
      Matrix_Vector_Activate_Batch<MatrixW, MatrixH, SIMD, PE, 1, TSrcI, TDstI, TWeightI>
        (static_cast<hls::stream<ap_uint<SIMD*TSrcI::width>>&>(convInp),
         static_cast<hls::stream<ap_uint<PE*TDstI::width>>&>  (mvOut),
         weights, activation, reps* OFMDim * OFMDim, r);
    }
```



NODE PROPERTIES

type	StreamingDataflowPartition
module	finn.custom_op.fpgadataflow
name	StreamingDataflowPartition_1

ATTRIBUTES

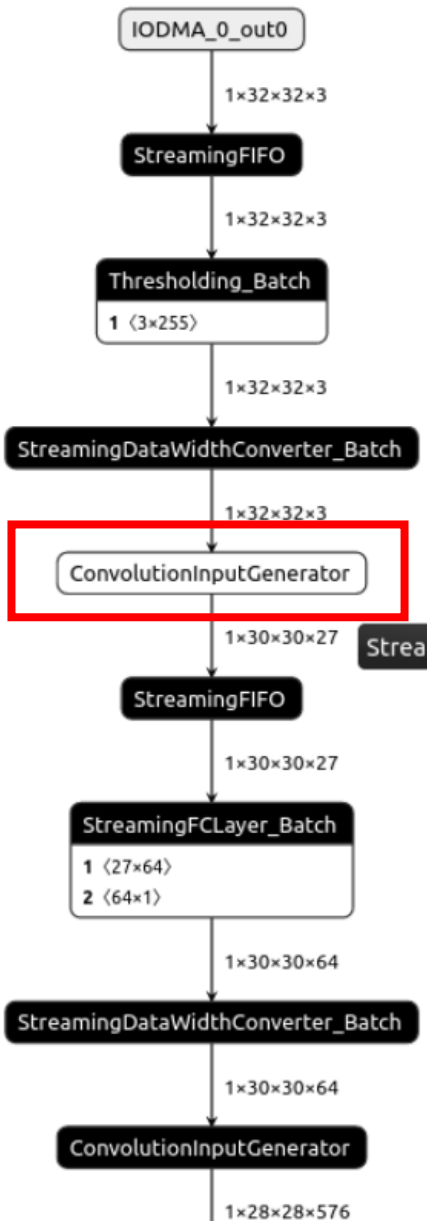
mem_port	
model	/home/yuoto/multimediaIC/FINN/finn_new_vitis/host_build /dataflow_partition2_wbswq3d3/df_model.onnx
partition_id	2
slr	-1

INPUTS

0	name: **StreamingDataflowPartition_0_out0**

OUTPUTS

0	name: **StreamingDataflowPartition_1_out0**



StreamingDataflowPartition_1_ConvolutionInputGenerator_0

NODE PROPERTIES

ATTRIBUTES

backend	fpgadataflow
code_gen_dir_ip ...	/home/yuoto/multimediaIC/FINN/finn_new_vitis/host_build /code_gen_ipgen_StreamingDataflowPartition_1_ConvolutionInput
ConvKernelDim	3, 3
depthwise	0
Dilation	1, 1
IFMChannels	3
IFMDim	32, 32
inputDataType	INT8
	finn_new_vitis/host_build /code_gen_ipgen_StreamingDataflowPartition_1_ConvolutionInput /project_StreamingDataflowPartition_1_ConvolutionInputGeneratc
ip_path	/home/yuoto/multimediaIC/FINN/finn_new_vitis/host_build /code_gen_ipgen_StreamingDataflowPartition_1_ConvolutionInput /project_StreamingDataflowPartition_1_ConvolutionInputGeneratc /sol1/impl/ip
ip_vlnv	xilinx.com:hls:StreamingDataflowPartition_1_ConvolutionInputGen
OFMDim	30, 30
outFIFODepth	128
outputDataType	INT8
partition_id	2

- ConvKernelDim: kernel size [3, 3]
- Dilation: [1, 1]
- Stride:[1, 1]
- IFMChannels(Input Feature Map): 3 (Int8 RGB)
- IFMDIM: [32, 32]
- SIMD: 3

ConvolutionInputGenerator

Stride = 1
IFMDim = 32
OFMDim = 30
multiplying_factor = 1
number_blocks = 4
cycle_write_block = 270
cycle_read_block = 32

counter_internal_block = 0
current_line = 0
current_block_read = 0
current_line_in_block = 0
count_simd = 0
current_block_write = 3
read_block = 3
kx = 0, ky = 0
ofm_x = 0, ofm_y = 0

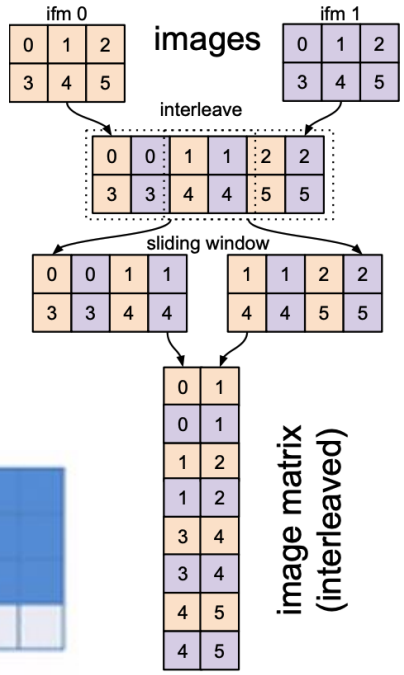
ConvolutionInputGenerator(1/4)

```
template<unsigned int ConvKernelDim,
         unsigned int IFMChannels,
         unsigned int Input_precision,
         unsigned int IFMDim,
         unsigned int OFMDim,
         unsigned int SIMD,
         unsigned int Stride,
         typename R>
void ConvolutionInputGenerator(
        stream<ap_uint<SIMD*Input_precision> > & in,
        stream<ap_uint<SIMD*Input_precision> > & out,
        const unsigned int numReps,
        R const &r) {
  CASSERT_DATAFLOW(IFMChannels % SIMD == 0);
  CASSERT_DATAFLOW(ConvKernelDim % Stride == 0);
  const unsigned int multiplying_factor = IFMChannels/SIMD;
  const unsigned int number_blocks = ConvKernelDim/Stride + 1 ;
  ap_uint<SIMD*Input_precision> inputBuf[number_blocks][Stride * IFMDim * multiplying_factor];
#pragma HLS ARRAY_PARTITION variable=inputBuf complete dim=1
  memory_resource(inputBuf, r);
  const unsigned int cycles_write_block = (OFMDim * ConvKernelDim * ConvKernelDim * multiplying_factor);
  const unsigned int cycles_read_block = Stride * IFMDim * multiplying_factor;
  const unsigned int max_cycles = MAX(cycles_write_block,cycles_read_block);
  const unsigned int baseIter = IFMDim * ConvKernelDim * multiplying_factor// Initial buffer
                              + OFMDim * MAX(cycles_write_block,cycles_read_block);
  unsigned int counter_internal_block = 0;
  unsigned int current_block_write = 0;
  unsigned int next_block_write = 0;
  unsigned int current_line = 0;
  unsigned int read_block = 0;
  unsigned int inp = 0, ofm_y = 0, ofm_x = 0, k_y = 0, k_x = 0, count_simd =0;
```

Define vars

ConvolutionInputGenerator

- **Outer loop:** *BaseIteration* to finish one image matrix
  - (if) First initialize input buffer
  - (else) Then … parallel do
    - write the output in the correct order
    - Read the next block
    - Update count



- Goal: pipelined with II=1

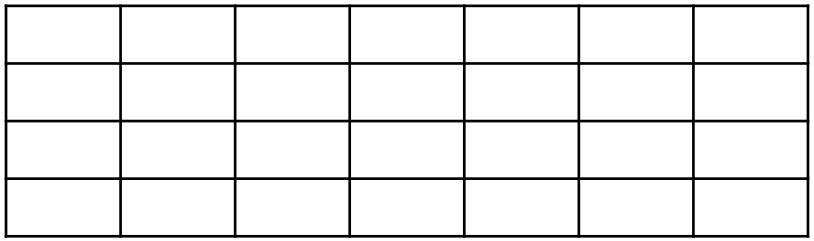
ConvolutionInputGenerator(1/4)

Base Iteration:
(if) First initialize input buffer
(else) Then parallel do 3 ifs

```
#pragma HLS reset variable=inp
  for (unsigned int count_image = 0; count_image < numReps; count_image++) {
    for (unsigned int i = 0; i < baseIter; i++) {
#pragma HLS PIPELINE II=1
      if (inp < IFMDim * ConvKernelDim*multiplying_factor) {// Initial buffer of ConvKe
        ap_uint<SIMD*Input_precision> inElem;
        inElem = in.read();
        inputBuf[current_block_write][current_line] = inElem;
        current_line++;
        inp++;
        if (current_line == Stride * IFMDim * multiplying_factor ) {
          current_line = 0;
          current_block_write++;
          if (current_block_write == number_blocks) {
            current_block_write=0;
          }
          read_block++;
          counter_internal_block = 0;
        }
      } else {
```

IFMDim*factor*stride

kernelDim +1



ConvolutionInputGenerator(1/4)

```
} else {
  if (counter_internal_block < cycles_write_block-1) { // We are writing output, MMV IFMChan per cycle
    unsigned int current_block_read = (current_block_write + 1 + k_y / Stride);
    if (current_block_read >= number_blocks) {
      current_block_read-= number_blocks;
    }
    unsigned int current_line_in_block = ((k_y%Stride) * IFMDim + ofm_x*Stride + k_x)*multiplying_factor + count_simd;
    ap_uint<SIMD*Input_precision> outElem = inputBuf[current_block_read][(current_line_in_block)];
    out.write(outElem);
    count_simd++;
    if (count_simd == multiplying_factor) {
      count_simd=0;
      k_x++;
      if (k_x == ConvKernelDim) {
        k_x = 0;
        k_y++;
        if (k_y == ConvKernelDim) {
          k_y = 0;
          ofm_x ++;
          if (ofm_x == OFMDim) {
            ofm_x = 0;
            ofm_y++;
            if (ofm_y == OFMDim) {
              ofm_y = 0;
              inp = 0;
            }
          }
        }
```

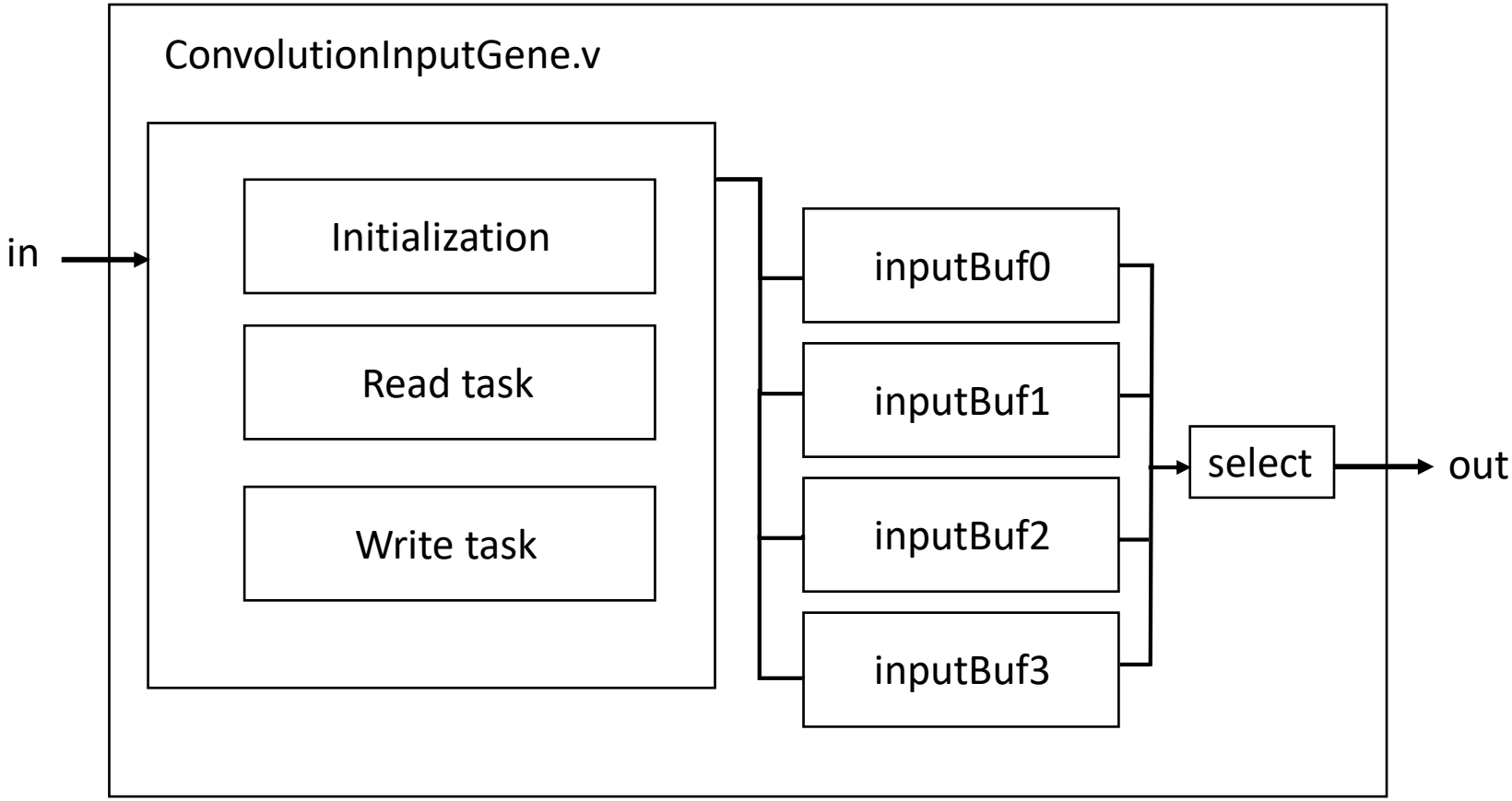
1. if: write output
(within # of write cycles)

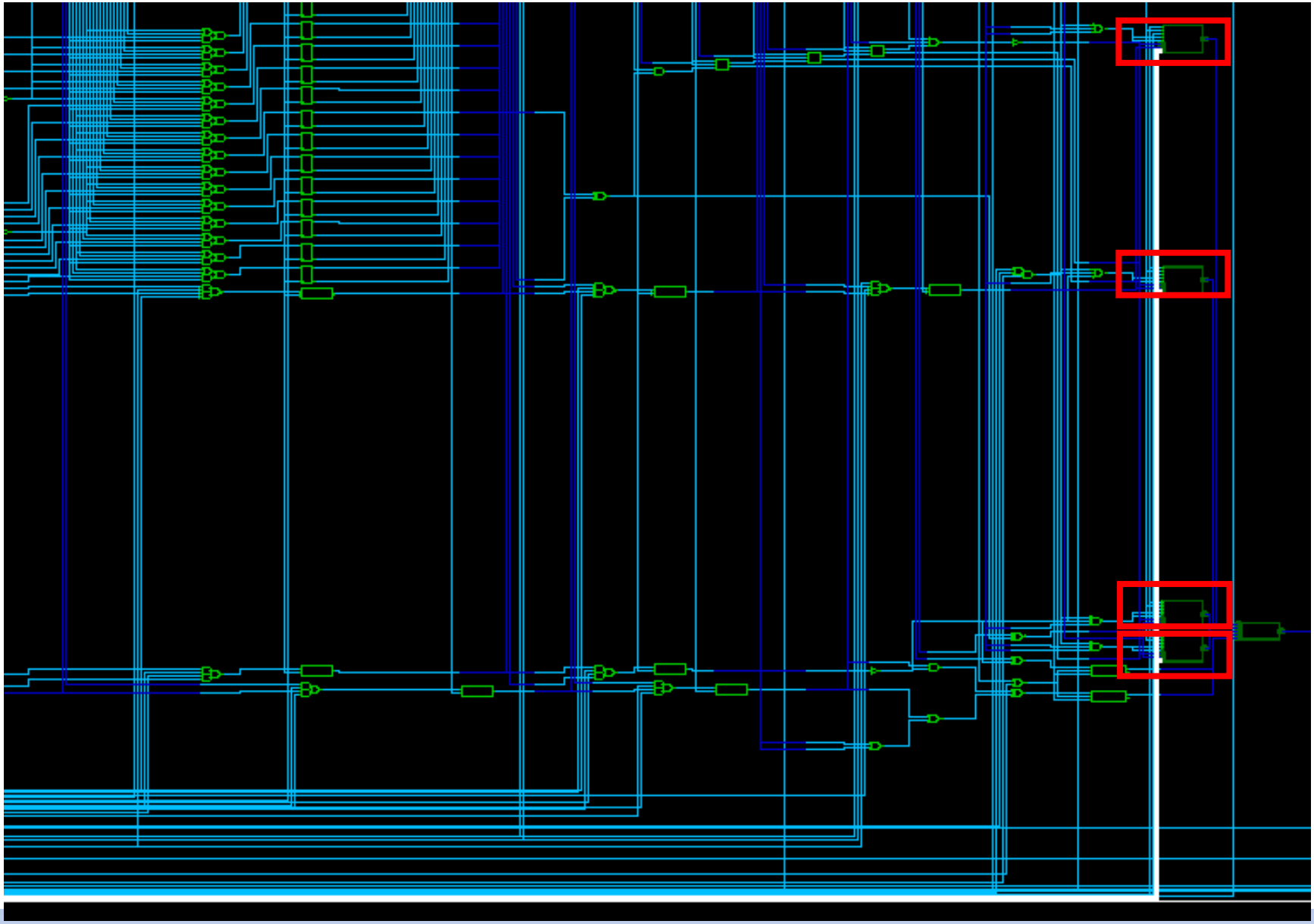
ConvolutionInputGenerator(4/4)

```
        if ((counter_internal_block < cycles_read_block-1) && (read_block<IFMDim/Stride)) { //
          ap_uint<SIMD*Input_precision> inElem;
          inElem = in.read();
          inputBuf[current_block_write][current_line] = inElem;
#pragma AP dependence variable=inputBuf intra false
#pragma AP dependence variable=inputBuf inter false
          current_line++;
          if (current_line == Stride * IFMDim * multiplying_factor) {// We read the whole bloc
            // We filled up a block, let's not read until
            current_line = 0;
            read_block++;
            current_block_write++;
            if (current_block_write == number_blocks) {
              current_block_write=0;
            }
#pragma AP dependence variable=current_block_write intra false
          }
        }
        counter_internal_block++; // = (counter_internal_block +1) % max_cycles;
        if (counter_internal_block == (max_cycles-1)) {
          counter_internal_block = 0;
        }
      }
    } // End base_iter
    read_block = 0;
  } // End count_image
} // End generator
```

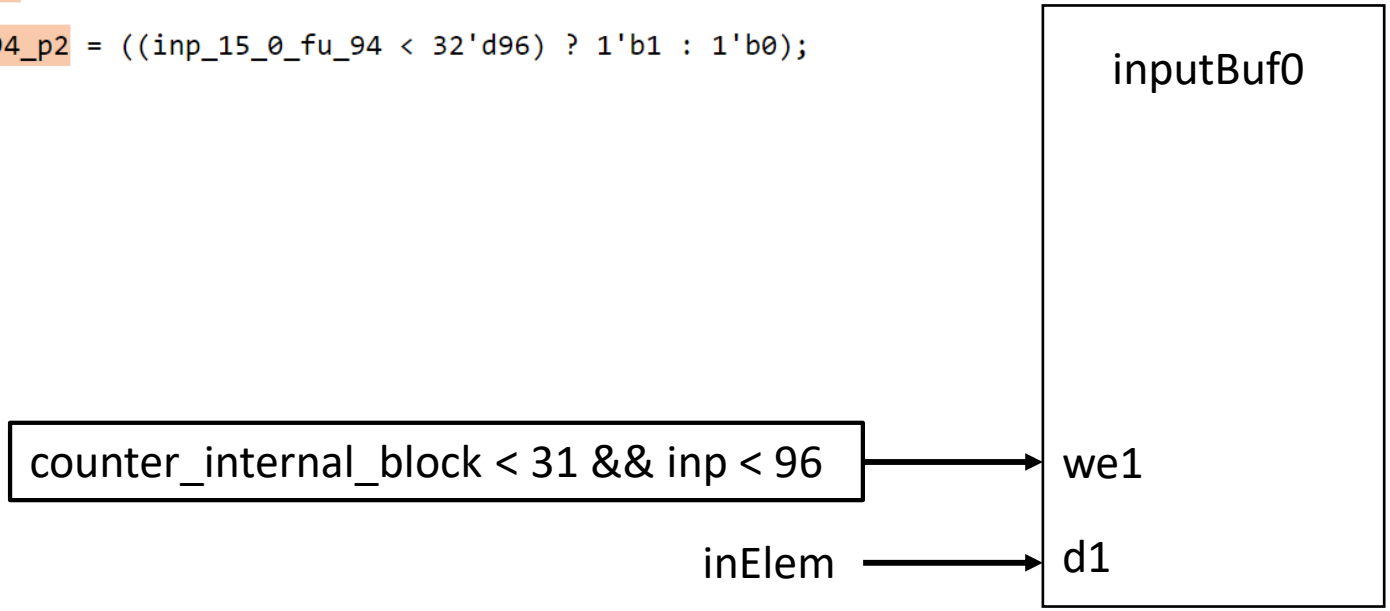
2. if: read input to buffer (within # of read cycles and image)

3. if: initialize the counter when one R/W pair is achieved. (max_cycle is reached)

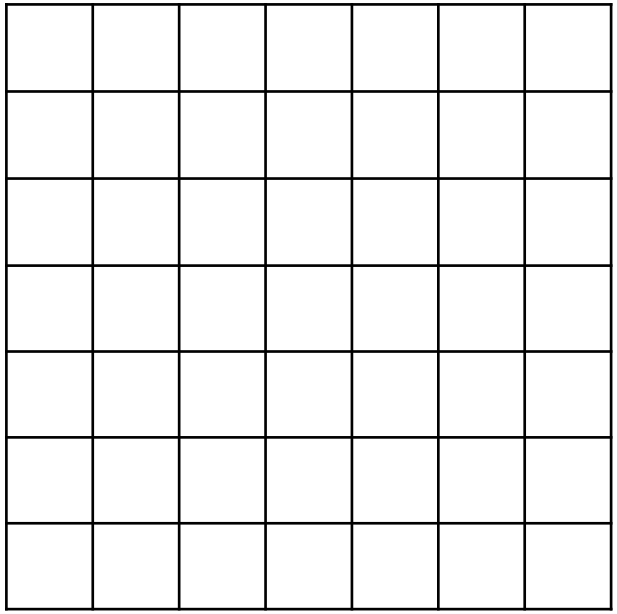

ConvolutionInputGene.v
in
Initialization
Read task
Write task
inputBuf0
inputBuf1
inputBuf2
inputBuf3
select
out



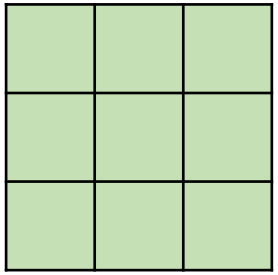
```
always @ (*) begin
    if ((((1'b0 == ap_block_pp0_stage0_11001) & (1'd1 == and_ln244_fu_598_p2) & (icmp_ln198_fu_394_p2 == 1'd0) & (icm
        inputBuf_0_V_we1 = 1'b1;
    end else begin
        inputBuf_0_V_we1 = 1'b0;
    end
end

assign and_ln244_fu_598_p2 = (icmp_ln244_fu_576_p2 & icmp_ln244_1_fu_592_p2);

assign icmp_ln244_fu_576_p2 = ((counter_internal_blo_fu_118 < 32'd31) ? 1'b1 : 1'b0);

assign icmp_ln198_fu_394_p2 = ((inp_15_0_fu_94 < 32'd96) ? 1'b1 : 1'b0);
```



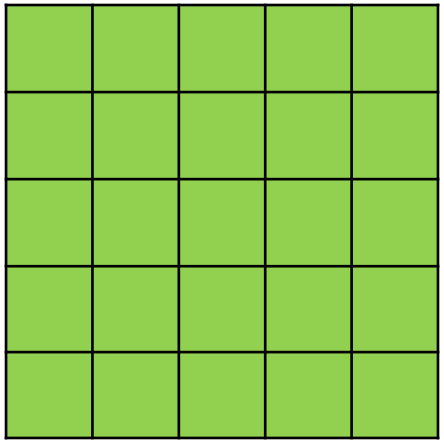
Case Study



Input IFM
Width:7
High:7
Channel:3
SIMD:3
Stride:1



Kernel:3x3



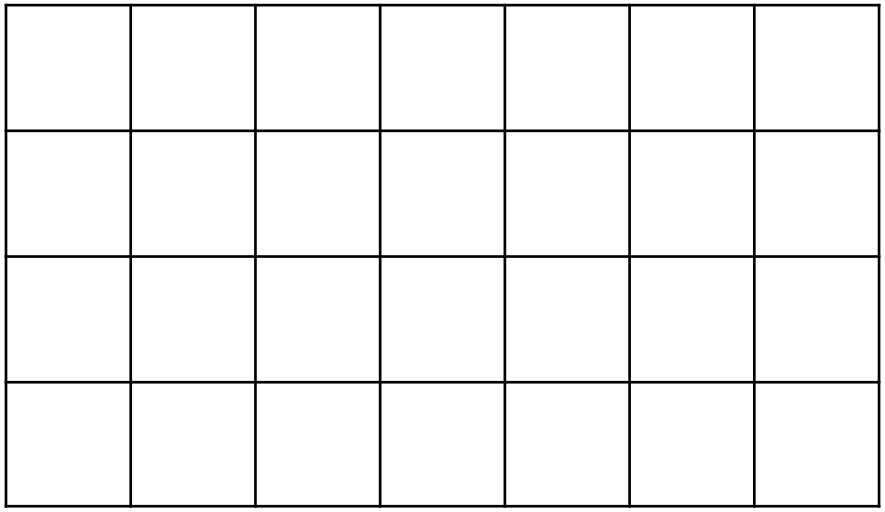
Output OFM
Width:5
High:5
Channel:1

Case Study: inputBuf

- ap_uint<SIMD*Input_precision>
  - inputBuf[number_blocks][Stride * IFMDim * multiplying_factor]
- number_blocks = ConvKernelDim/Stride + 1 = 4
- Stride * IFMDim * multiplying_factor = 7
  - inputBuf[4][7]

multiplying_factor = IFMChannels / SIMD

(# of cycles to pick all data channel-wise)



Case Study: baseIter

- baseIter = IFMDim * ConvKernelDim * multiplying_factor// Initial
  + OFMDim *MAX(cycles_write_block,cycles_read_block);

- IFMDim * ConvKernelDim * multiplying_factor=21
- OFMDim *MAX(cycles_write_block,cycles_read_block)=225

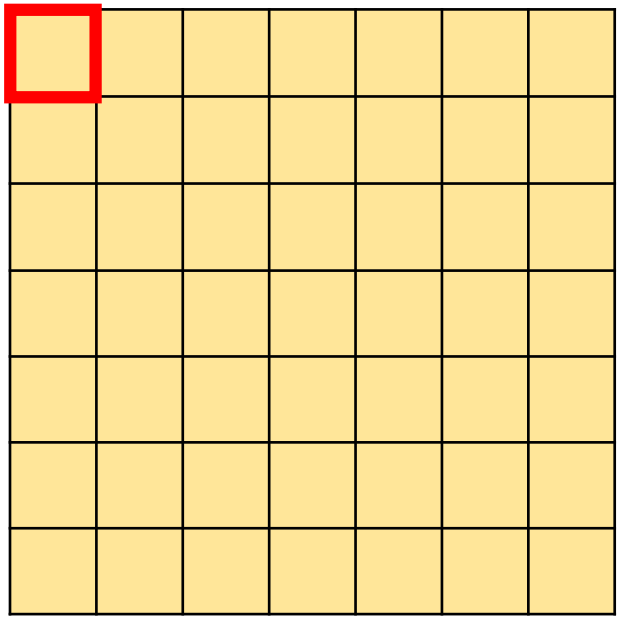
- cycles_write_block = 5 * 3 * 3 * 1 = 45
- cycles_read_block = 1 * 7 * 1 = 7
- baseIter = 21+225 = 246

```
const unsigned int cycles_write_block = (OFMDim * ConvKernelDim * ConvKernelDim * multiplying_factor);
const unsigned int cycles_read_block = Stride * IFMDim * multiplying_factor;
```

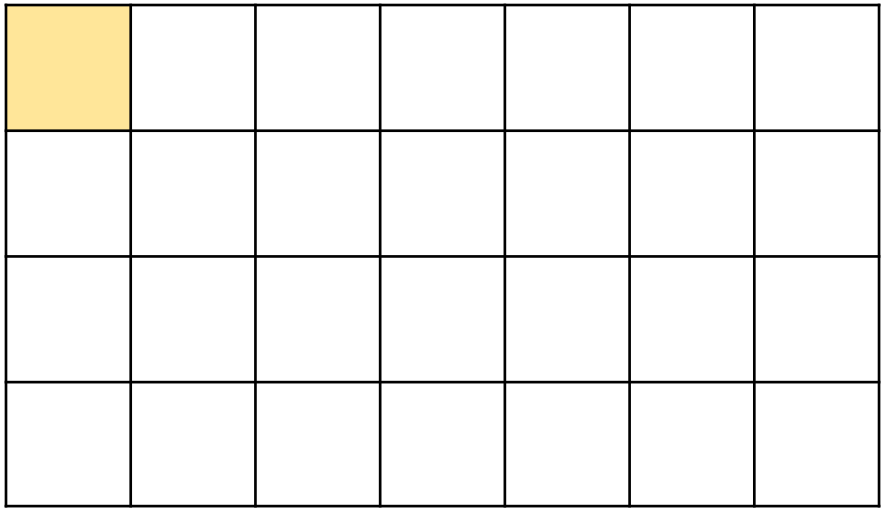
i=0
inp=0

```
if (inp < IFMDim * ConvKernelDim*multiplying_factor) {// Initial buffer of ConvKernelDim lines
  ap_uint<SIMD*Input_precision> inElem;
  inElem = in.read();
  inputBuf[current_block_write][current_line] = inElem;
  current_line++;
  inp++;
  if (current_line == Stride * IFMDim * multiplying_factor ) {
    current_line = 0;
    current_block_write++;
    if (current_block_write == number_blocks) {
      current_block_write=0;
    }
    read_block++;
    counter_internal_block = 0;
  }
} else {
```

Input IFM



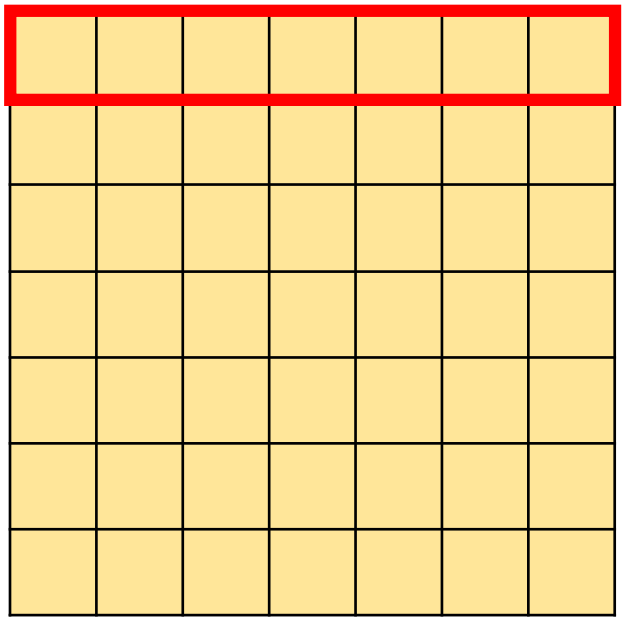
InputBuff



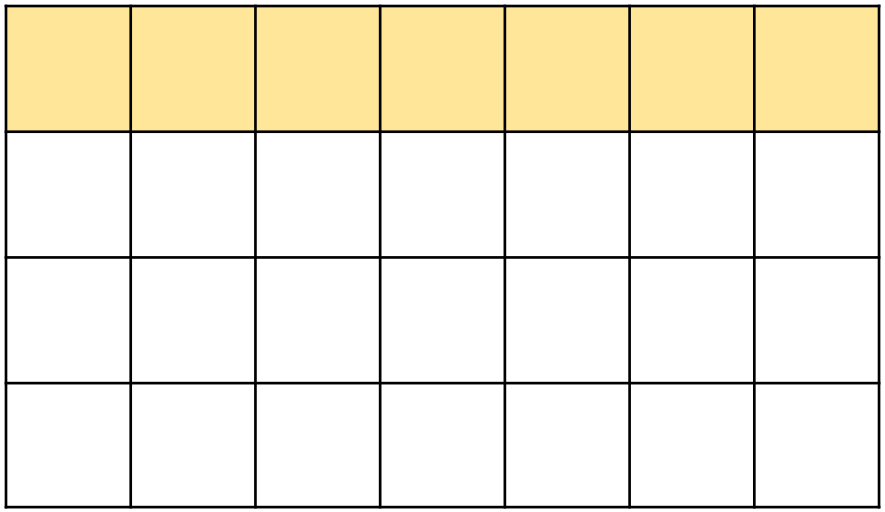
i=6
inp=6

```
if (inp < IFMDim * ConvKernelDim*multiplying_factor) {// Initial buffer of ConvKernelDim lines
  ap_uint<SIMD*Input_precision> inElem;
  inElem = in.read();
  inputBuf[current_block_write][current_line] = inElem;
  current_line++;
  inp++;
  if (current_line == Stride * IFMDim * multiplying_factor ) {
    current_line = 0;
    current_block_write++;
    if (current_block_write == number_blocks) {
      current_block_write=0;
    }
    read_block++;
    counter_internal_block = 0;
  }
} else {
```

Input IFM



InputBuff

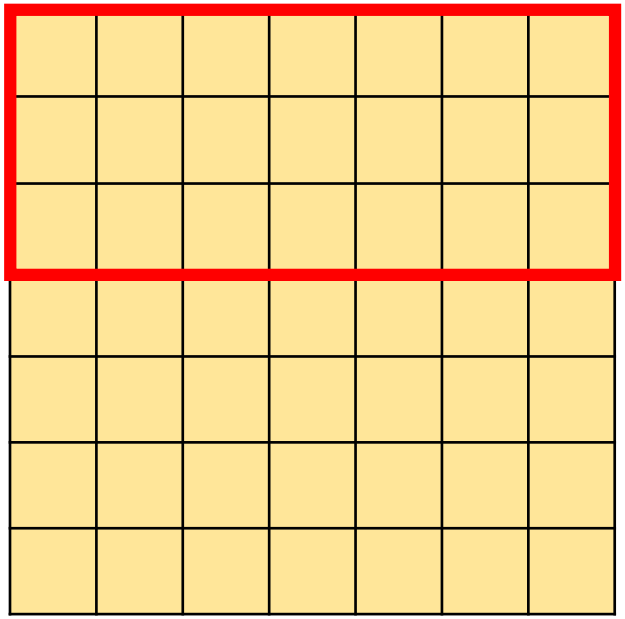


i=20
inp=20

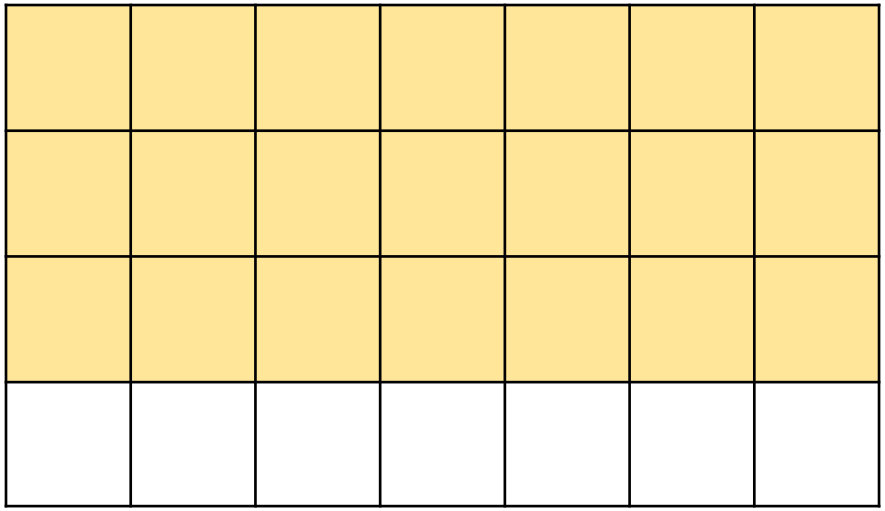
7*3*1=21

```
if (inp < IFMDim * ConvKernelDim*multiplying_factor) {// Initial buffer of ConvKernelDim lines
  ap_uint<SIMD*Input_precision> inElem;
  inElem = in.read();
  inputBuf[current_block_write][current_line] = inElem;
  current_line++;
  inp++;
  if (current_line == Stride * IFMDim * multiplying_factor ) {
    current_line = 0;
    current_block_write++;
    if (current_block_write == number_blocks) {
      current_block_write=0;
    }
    read_block++;
    counter_internal_block = 0;
  }
} else {
```

Input IFM



InputBuff



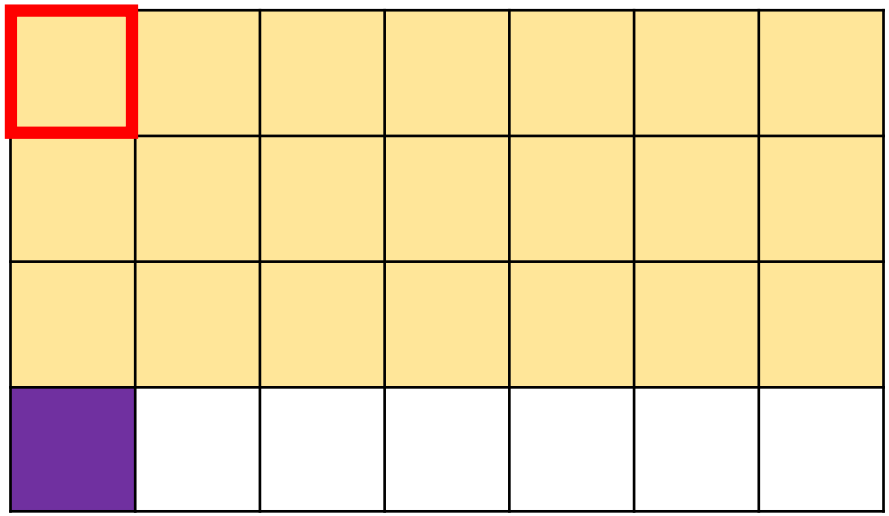
i=21(do else)

```
213        } else {
214          if (counter_internal_block < cycles_write_block-1) { // We are writing output, MMV IFMChan per cycle
215            unsigned int current_block_read = (current_block_write + 1 + k_y / Stride);
216            if (current_block_read >= number_blocks) {
217              current_block_read-= number_blocks;
218            }
219            unsigned int current_line_in_block = ((k_y%Stride) * IFMDim + ofm_x*Stride + k_x)*multiplying_factor + count_simd;
220            ap_uint<SIMD*Input_precision> outElem = inputBuf[current_block_read][(current_line_in_block)];
221            out.write(outElem);
222            count_simd++;
223            if (count_simd == multiplying_factor) {
224              count_simd=0;
225              k_x++;
226              if (k_x == ConvKernelDim) {
227                k_x = 0;
228                k_y++;
229                if (k_y == ConvKernelDim) {
230                  k_y = 0;
231                  ofm_x ++;
232                  if (ofm_x == OFMDim) {
233                    ofm_x = 0;
234                    ofm_y++;
235                    if (ofm_y == OFMDim) {
236                      ofm_y = 0;
237                      inp = 0;
```

0 0

Kx=0
Ky=0
Ox=0
Oy=0

InputBuff



SWU output (image matrix)

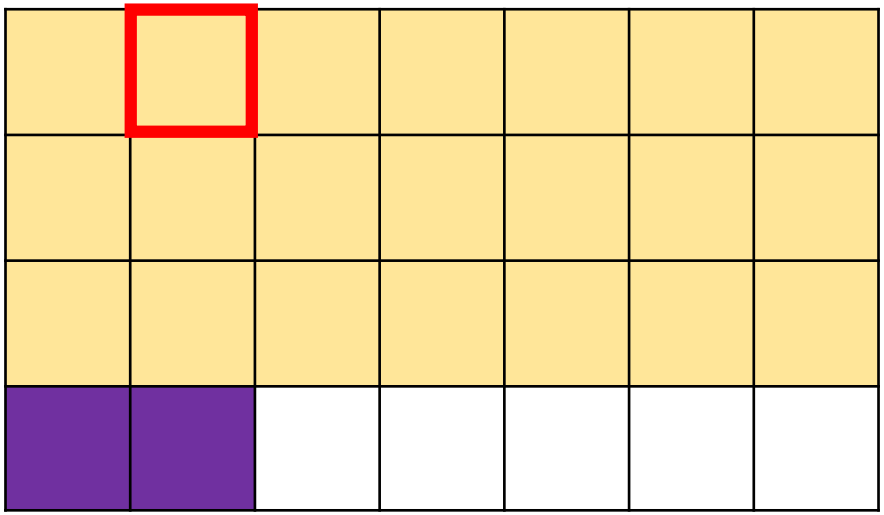
i=22

```
213        } else {
214          if (counter_internal_block < cycles_write_block-1) { // We are writing output, MMV IFMChan per cycle
215            unsigned int current_block_read = (current_block_write + 1 + k_y / Stride);
216            if (current_block_read >= number_blocks) {
217              current_block_read-= number_blocks;
218            }
219            unsigned int current_line_in_block = ((k_y%Stride) * IFMDim + ofm_x*Stride + k_x)*multiplying_factor + count_simd;
220            ap_uint<SIMD*Input_precision> outElem = inputBuf[current_block_read][(current_line_in_block)];
221            out.write(outElem);
222            count_simd++;
223            if (count_simd == multiplying_factor) {
224              count_simd=0;
225              k_x++;
226              if (k_x == ConvKernelDim) {
227                k_x = 0;
228                k_y++;
229                if (k_y == ConvKernelDim) {
230                  k_y = 0;
231                  ofm_x ++;
232                  if (ofm_x == OFMDim) {
233                    ofm_x = 0;
234                    ofm_y++;
235                    if (ofm_y == OFMDim) {
236                      ofm_y = 0;
237                      inp = 0;
```

0 1

Kx=1
Ky=0
Ox=0
Oy=0

InputBuff



SWU output (image matrix)

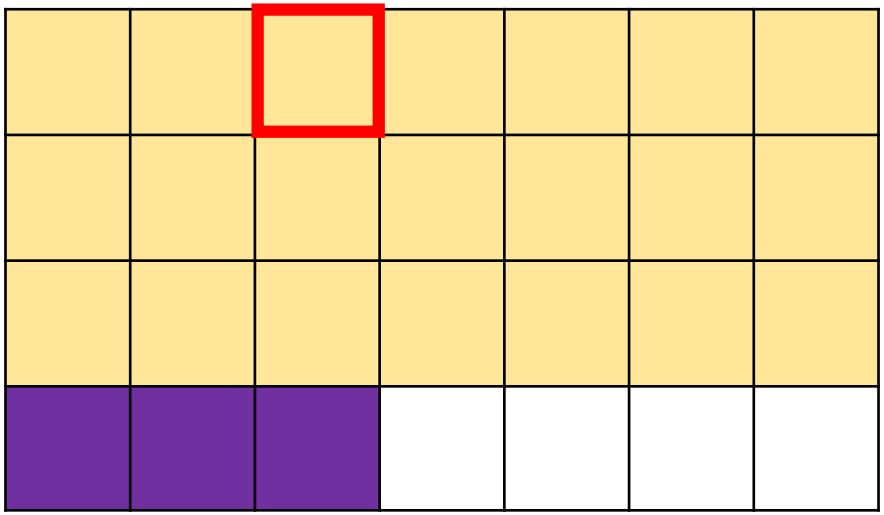
i=23

```
213        } else {
214          if (counter_internal_block < cycles_write_block-1) { // We are writing output, MMV IFMChan per cycle
215            unsigned int current_block_read = (current_block_write + 1 + k_y / Stride);
216            if (current_block_read >= number_blocks) {
217              current_block_read-= number_blocks;
218            }
219            unsigned int current_line_in_block = ((k_y%Stride) * IFMDim + ofm_x*Stride + k_x)*multiplying_factor + count_simd;
220            ap_uint<SIMD*Input_precision> outElem = inputBuf[current_block_read][(current_line_in_block)];
221            out.write(outElem);
222            count_simd++;
223            if (count_simd == multiplying_factor) {
224              count_simd=0;
225              k_x++;
226              if (k_x == ConvKernelDim) {
227                k_x = 0;
228                k_y++;
229                if (k_y == ConvKernelDim) {
230                  k_y = 0;
231                  ofm_x ++;
232                  if (ofm_x == OFMDim) {
233                    ofm_x = 0;
234                    ofm_y++;
235                    if (ofm_y == OFMDim) {
236                      ofm_y = 0;
237                      inp = 0;
```

0 2

Kx=2
Ky=0
Ox=0
Oy=0

InputBuff



SWU output (image matrix)

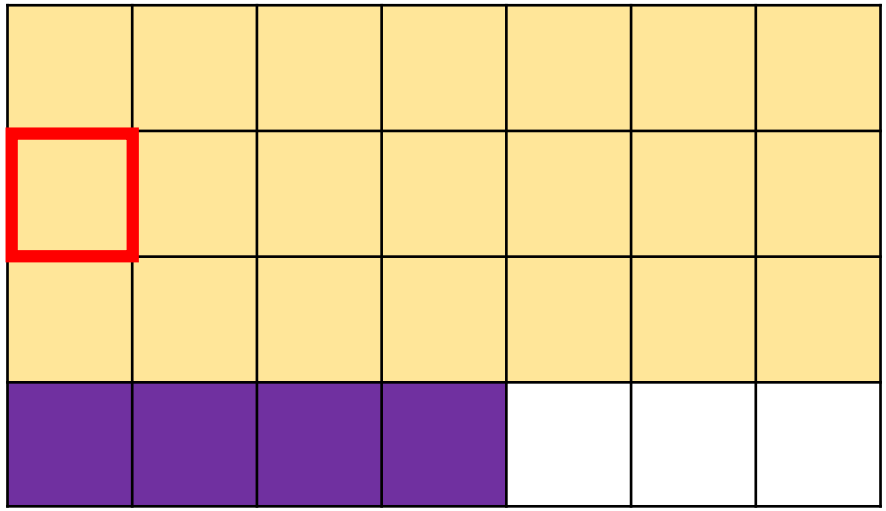
i=24

```
} else {
  if (counter_internal_block < cycles_write_block-1) { // We are writing output, MMV IFMChan per cycle
    unsigned int current_block_read = (current_block_write + 1 + k_y / Stride);
    if (current_block_read >= number_blocks) {
      current_block_read-= number_blocks;
    }
    unsigned int current_line_in_block = ((k_y%Stride) * IFMDim + ofm_x*Stride + k_x)*multiplying_factor + count_simd;
    ap_uint<SIMD*Input_precision> outElem = inputBuf[current_block_read][(current_line_in_block)];
    out.write(outElem);
    count_simd++;
    if (count_simd == multiplying_factor) {
      count_simd=0;
      k_x++;
      if (k_x == ConvKernelDim) {
        k_x = 0;
        k_y++;
        if (k_y == ConvKernelDim) {
          k_y = 0;
          ofm_x ++;
          if (ofm_x == OFMDim) {
            ofm_x = 0;
            ofm_y++;
            if (ofm_y == OFMDim) {
              ofm_y = 0;
              inp = 0;
```

1 0

Kx=0
Ky=1
Ox=0
Oy=0

InputBuff



SWU output (image matrix)

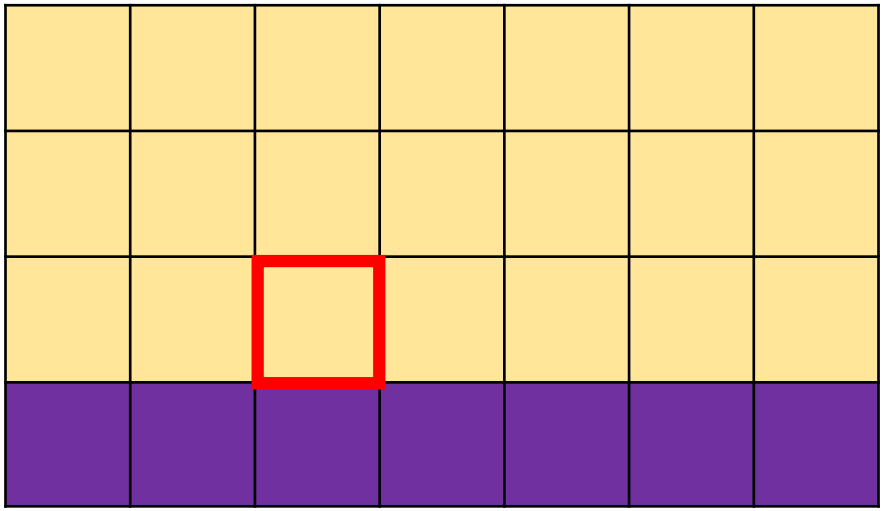
i=29

```
213        } else {
214          if (counter_internal_block < cycles_write_block-1) { // We are writing output, MMV IFMChan per cycle
215            unsigned int current_block_read = (current_block_write + 1 + k_y / Stride);
216            if (current_block_read >= number_blocks) {
217              current_block_read-= number_blocks;
218            }
219            unsigned int current_line_in_block = ((k_y%Stride) * IFMDim + ofm_x*Stride + k_x)*multiplying_factor + count_simd;
220            ap_uint<SIMD*Input_precision> outElem = inputBuf[current_block_read][(current_line_in_block)];
221            out.write(outElem);
222            count_simd++;
223            if (count_simd == multiplying_factor) {
224              count_simd=0;
225              k_x++;
226              if (k_x == ConvKernelDim) {
227                k_x = 0;
228                k_y++;
229                if (k_y == ConvKernelDim) {
230                  k_y = 0;
231                  ofm_x ++;
232                  if (ofm_x == OFMDim) {
233                    ofm_x = 0;
234                    ofm_y++;
235                    if (ofm_y == OFMDim) {
236                      ofm_y = 0;
237                      inp = 0;
```

2 2

Kx=2
Ky=2
Ox=0
Oy=0

InputBuff



SWU output (image matrix)

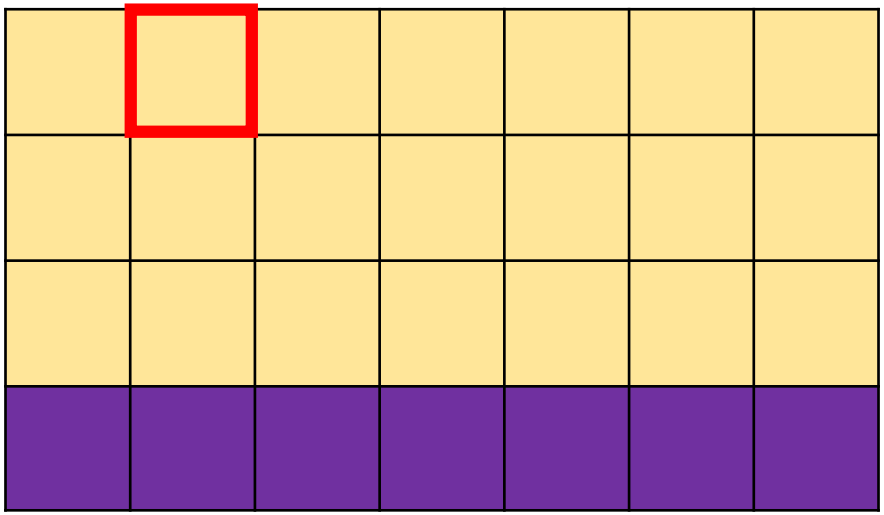
i=30

```
    } else {
      if (counter_internal_block < cycles_write_block-1) { // We are writing output, MMV IFMChan per cycle
        unsigned int current_block_read = (current_block_write + 1 + k_y / Stride);
        if (current_block_read >= number_blocks) {
          current_block_read-= number_blocks;
        }
        unsigned int current_line_in_block = ((k_y%Stride) * IFMDim + ofm_x*Stride + k_x)*multiplying_factor + count_simd;
        ap_uint<SIMD*Input_precision> outElem = inputBuf[current_block_read][(current_line_in_block)];
        out.write(outElem);
        count_simd++;
        if (count_simd == multiplying_factor) {
          count_simd=0;
          k_x++;
          if (k_x == ConvKernelDim) {
            k_x = 0;
            k_y++;
            if (k_y == ConvKernelDim) {
              k_y = 0;
              ofm_x ++;
              if (ofm_x == OFMDim) {
                ofm_x = 0;
                ofm_y++;
                if (ofm_y == OFMDim) {
                  ofm_y = 0;
                  inp = 0;
```

0 1

Kx=0
Ky=0
Ox=1
Oy=0

InputBuff



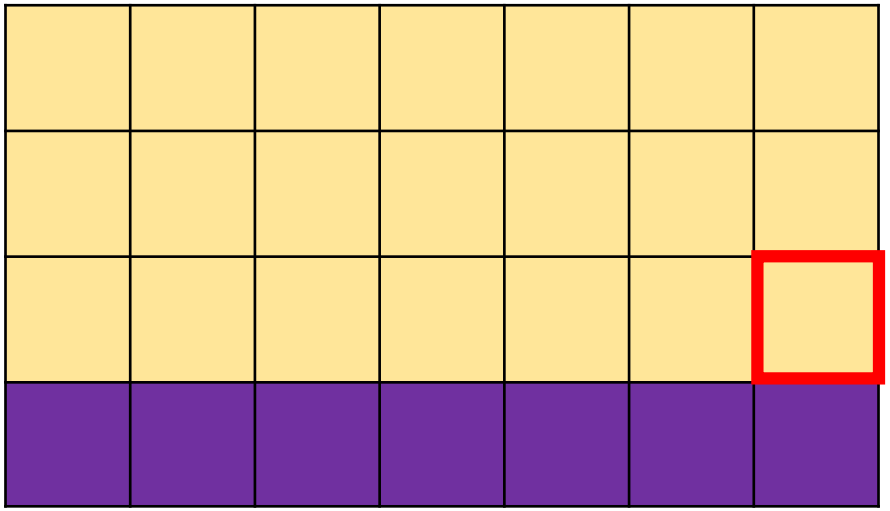
i=65

```
} else {
  if (counter_internal_block < cycles_write_block-1) { // We are writing output, MMV IFMChan per cycle
    unsigned int current_block_read = (current_block_write + 1 + k_y / Stride);
    if (current_block_read >= number_blocks) {
      current_block_read-= number_blocks;
    }
    unsigned int current_line_in_block = ((k_y%Stride) * IFMDim + ofm_x*Stride + k_x)*multiplying_factor + count_simd;
    ap_uint<SIMD*Input_precision> outElem = inputBuf[current_block_read][(current_line_in_block)];
    out.write(outElem);
    count_simd++;
    if (count_simd == multiplying_factor) {
      count_simd=0;
      k_x++;
      if (k_x == ConvKernelDim) {
        k_x = 0;
        k_y++;
        if (k_y == ConvKernelDim) {
          k_y = 0;
          ofm_x ++;
          if (ofm_x == OFMDim) {
            ofm_x = 0;
            ofm_y++;
            if (ofm_y == OFMDim) {
              ofm_y = 0;
              inp = 0;
```

2 6

Kx=2
Ky=2
Ox=4
Oy=0

InputBuff



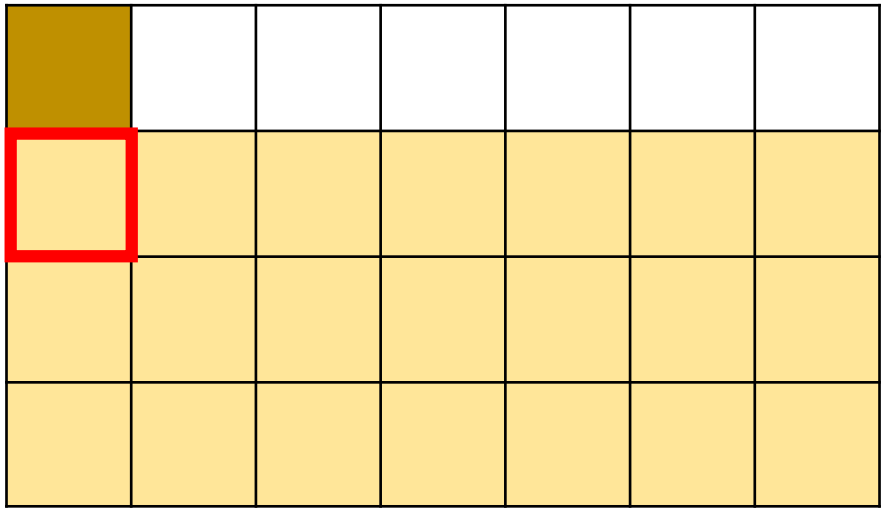
i=66

```
213        } else {
214          if (counter_internal_block < cycles_write_block-1) { // We are writing output, MMV IFMChan per cycle
215            unsigned int current_block_read = (current_block_write + 1 + k_y / Stride);
216            if (current_block_read >= number_blocks) {
217              current_block_read-= number_blocks;
218            }
219            unsigned int current_line_in_block = ((k_y%Stride) * IFMDim + ofm_x*Stride + k_x)*multiplying_factor + count_simd;
220            ap_uint<SIMD*Input_precision> outElem = inputBuf[current_block_read][(current_line_in_block)];
221            out.write(outElem);
222            count_simd++;
223            if (count_simd == multiplying_factor) {
224              count_simd=0;
225              k_x++;
226              if (k_x == ConvKernelDim) {
227                k_x = 0;
228                k_y++;
229                if (k_y == ConvKernelDim) {
230                  k_y = 0;
231                  ofm_x ++;
232                  if (ofm_x == OFMDim) {
233                    ofm_x = 0;
234                    ofm_y++;
235                    if (ofm_y == OFMDim) {
236                      ofm_y = 0;
237                      inp = 0;
```

1 0

Kx=0
Ky=0
Ox=0
Oy=1

InputBuff



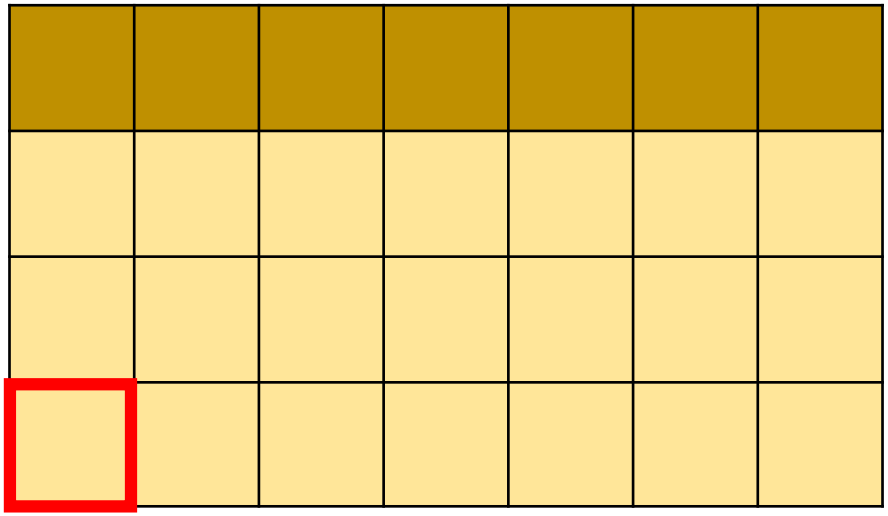
i=72

```
      } else {
        if (counter_internal_block < cycles_write_block-1) { // We are writing output, MMV IFMChan per cycle
          unsigned int current_block_read = (current_block_write + 1 + k_y / Stride);
          if (current_block_read >= number_blocks) {
            current_block_read-= number_blocks;
          }
          unsigned int current_line_in_block = ((k_y%Stride) * IFMDim + ofm_x*Stride + k_x)*multiplying_factor + count_simd;
          ap_uint<SIMD*Input_precision> outElem = inputBuf[current_block_read][(current_line_in_block)];
          out.write(outElem);
          count_simd++;
          if (count_simd == multiplying_factor) {
            count_simd=0;
            k_x++;
            if (k_x == ConvKernelDim) {
              k_x = 0;
              k_y++;
              if (k_y == ConvKernelDim) {
                k_y = 0;
                ofm_x ++;
                if (ofm_x == OFMDim) {
                  ofm_x = 0;
                  ofm_y++;
                  if (ofm_y == OFMDim) {
                    ofm_y = 0;
                    inp = 0;
```

3 0

Kx=0
Ky=2
Ox=0
Oy=1

InputBuff



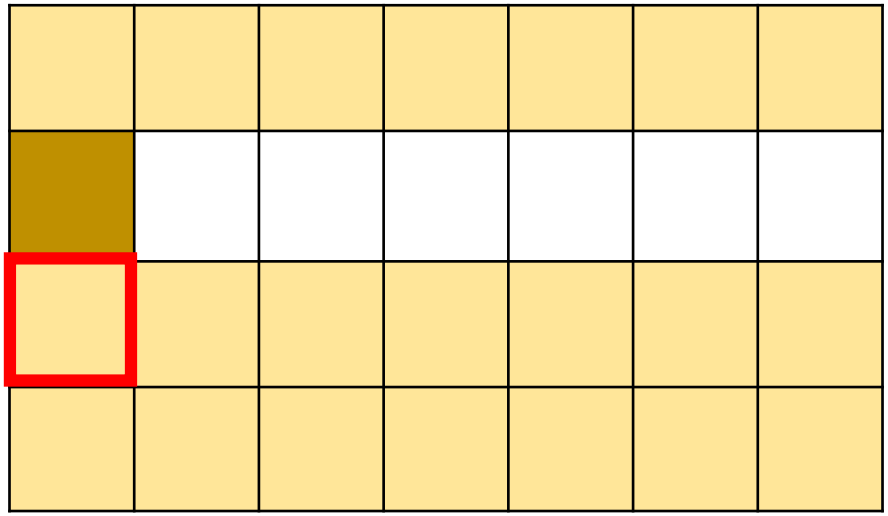
i=111

```
213        } else {
214          if (counter_internal_block < cycles_write_block-1) { // We are writing output, MMV IFMChan per cycle
215            unsigned int current_block_read = (current_block_write + 1 + k_y / Stride);
216            if (current_block_read >= number_blocks) {
217              current_block_read-= number_blocks;
218            }
219            unsigned int current_line_in_block = ((k_y%Stride) * IFMDim + ofm_x*Stride + k_x)*multiplying_factor + count_simd;
220            ap_uint<SIMD*Input_precision> outElem = inputBuf[current_block_read][(current_line_in_block)];
221            out.write(outElem);
222            count_simd++;
223            if (count_simd == multiplying_factor) {
224              count_simd=0;
225              k_x++;
226              if (k_x == ConvKernelDim) {
227                k_x = 0;
228                k_y++;
229                if (k_y == ConvKernelDim) {
230                  k_y = 0;
231                  ofm_x ++;
232                  if (ofm_x == OFMDim) {
233                    ofm_x = 0;
234                    ofm_y++;
235                    if (ofm_y == OFMDim) {
236                      ofm_y = 0;
237                      inp = 0;
```

2 0

Kx=0
Ky=0
Ox=0
Oy=2

InputBuff



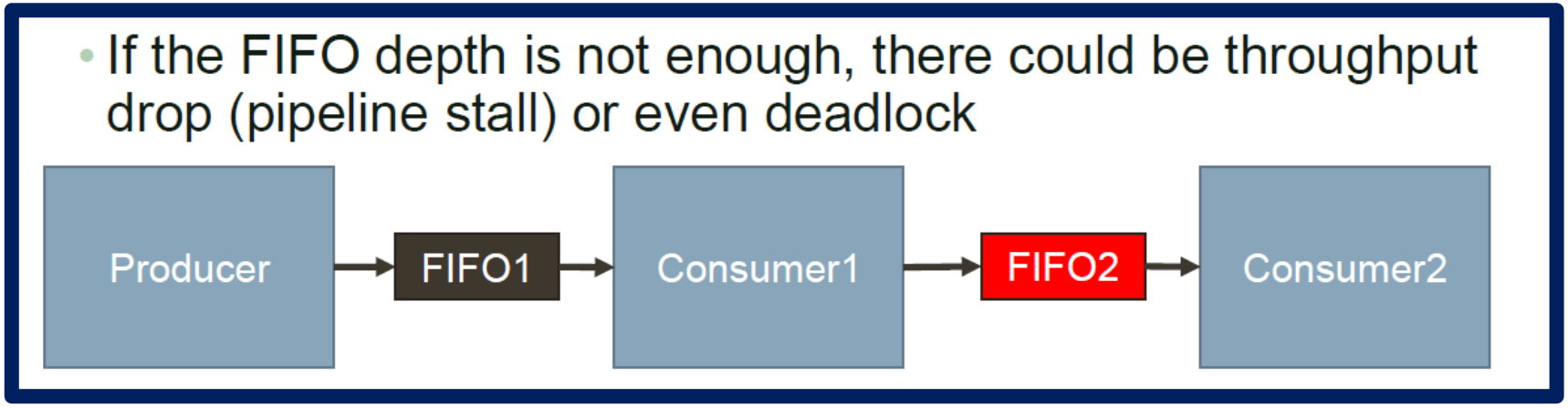
- One more row used!
  - Not conventional conv2d
- Imbalanced Read/Write cycles!

Outline

- Matrix-Vector Accumulation Unit
- Pooling Layer
- Convolution Layer
- FIFO

FIFO

- **Critical in the whole system design.**
  - If the depth of the FIFO is not enough
  - Streaming process might encounter a stall in response to the "full" or "empty" signal.



- *hls::stream* interface
  - *ap_fifo*
  - Tool will make a FIFO with a default depth of 2.

```python
fc_layers = model.get_nodes_by_op_type("StreamingFCLayer_Batch")
# (PE, SIMD, in_fifo_depth, out_fifo_depth, ramstyle) for each layer
config = [
    (16, 49, 16, 64, "block"),
    (8, 8, 64, 64, "auto"),
    (8, 8, 64, 64, "auto"),
    (10, 8, 64, 10, "distributed"),
]
for fcl, (pe, simd, ififo, ofifo, ramstyle) in zip(fc_layers, config):
    fcl_inst = getCustomOp(fcl)
    fcl_inst.set_nodeattr("PE", pe)
    fcl_inst.set_nodeattr("SIMD", simd)
    fcl_inst.set_nodeattr("inFIFODepth", ififo)
    fcl_inst.set_nodeattr("outFIFODepth", ofifo)
    fcl_inst.set_nodeattr("ram_style", ramstyle)

# set parallelism for input quantizer to be same as first layer's SIMD
inp_qnt_node = model.get_nodes_by_op_type("Thresholding_Batch")[0]
inp_qnt = getCustomOp(inp_qnt_node)
inp_qnt.set_nodeattr("PE", 49)
```

How to determine FIFO depths?

- **Depends on** the IO protocol, bandwidth, and the throughput of the data producer/consumer.
  - All these possible factors result in **the difficulty of analytical FIFO depth calculation.**
  - Blindly increase the depth to a great extent, if the budget of the area is infinite

- **Empirically estimate** suitable FIFO depths.
  1. Set all FIFO depths to a large number.
  2. Prepare enough test patterns that can simulate the actual data processing flow
  3. Record the maximum data occupancy of this FIFO.
  4. After the simulation is done, change the size of the FIFO to this empirical depth parameter.
  5. Measure the resulting throughput drop of the system with this new configuration.
  6. If no throughput drop, a more aggressive FIFO depth reduction method may use.

How to determine FIFO depths?

- Use RTL co-simulation collect FIFO depth
  - Confirm the throughput is balanced
  - Switching sub-modules to prevent latency mismatch

- Inevitable compute cycle & Read cycle sync
  - Pattern specific
  - Lower outstanding numbers & burst length

Optimized	FIFO Max Depth
Frame stream	475 → 450→ 2
Point stream	92 → 2
Jacobian stream	1
Residual stream	1

Media IC and System Lab
Graduate Institute of Electronics Engineering
National Taiwan University

Hua-Yang Weng

FIFO Depth in FINN

- **FINN Compiler** (hardware build *ZynqBuild()* phase)
  - ***InsertFIFO( )*** inserts FIFOs with the manually specified depth.
  - FINN also provides ***InsertAndSetFIFODepths( ).***

- *InsertAndSetFIFODepths( )*
  - First set all FIFO depths to a number of **16K** and sends random input image patterns.
  - **PyVerilator** is used for testing the maximum FIFO occupancy.

InsertAndSetFIFODepths( )

```
class InsertAndSetFIFODepths(Transformation):
    """Insert appropriate-depth StreamingFIFOs through RTLSim that preserve
    throughput in the created accelerator.

    Constructor arguments:
    - clk_ns : clock period (used for IP preparation)
    - max_qsrl_depth : FIFOs deeper than this will use Vivado IP instead of
                       Verilog FIFOs (Q_srl.v)
    - max_depth : how deep the "max"-sized FIFOs initially inserted will be
    - swg_exception : call CapConvolutionFIFODepths to make convolution FIFOs
                      smaller where appropriate
    - vivado_ram_style : the StreamingFIFO.ram_style attribute to be used for
                         large FIFOs implemented by Vivado

    Assumed input graph properties:
    - all nodes are fpgadataflow nodes
    - no FIFOs inserted,
    - (inFIFODepth/outFIFODepth attrs will be ignored)

    Output:
    - graph with appropriate-depth FIFOs inserted

    Background:
    Even with all FINN HLS fpgadatflow layers appropriately parallelized, it is
    necessary to insert FIFOs between them to prevent stalls due to bursty
    behavior. The sizes of those FIFOs are hard to predict analytically, so
    we do the following:
    - insert very deep (default 16k deep) FIFOs between all fpgadataflow nodes
    - create stitched design
    - run through rtlsim with stream of multiple random input images (to fill pipeline)
    - keep track of observed maximum occupancy for each FIFO during rtlsim
    - when sim finished, update each FIFO depth to maximum observed occupancy
      and set inFIFODepth/outFIFODepth attrs to 0 on relevant nodes
    """
```

※ **Note**: *ZynqBuild()* class does not default use this FIFO insertion method

- **We have to manually replace it.**

```
# change this
kernel_model = kernel_model.transform(InsertFIFO())

# to this
kernel_model = kernel_model.transform(InsertAndSetFIFODepths(fpgapart=self.fpga_part,
                                                              clk_ns=self.period_ns))
```