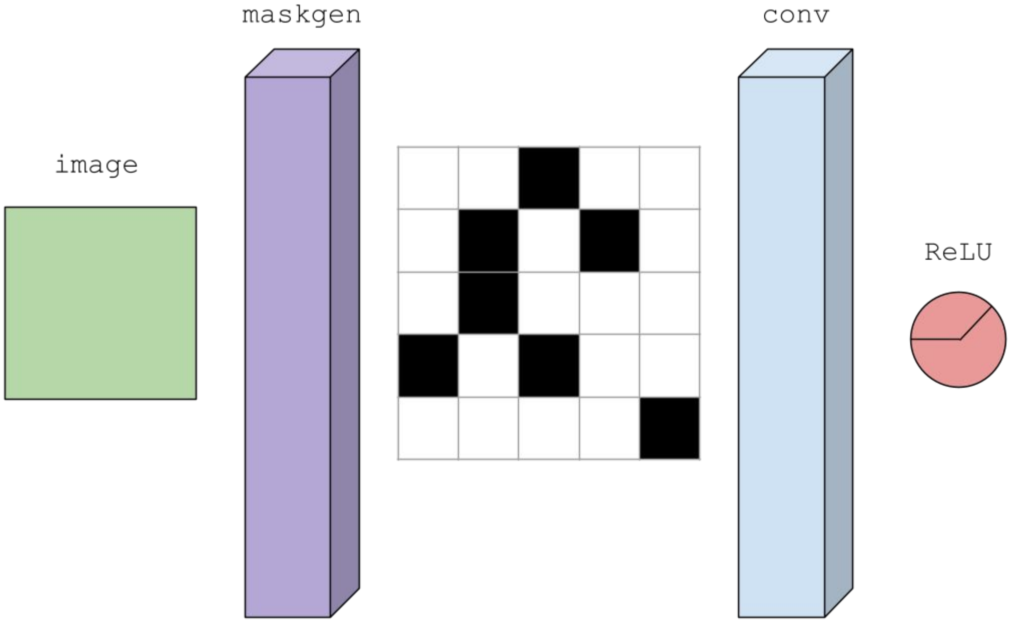
Deep Learning Hardware Deployment Internship

Zero Skipping Mask Generation

Young Jae Kim
2018-15653
Department of Electrical and Computer Engineering

Estimating Where the Zeros Will Be...



The CNN network

Layer	Filter	Input	Output	Mask
1	3x3	128x128x1	128x128x64	N/A
2	1x1	128x128x64	128x128x32	1x1x64x4, 1x1x4x32
3	3x3	128x128x32	128x128x32	1x1x32x4, 3x3x4x4, 1x1x4x32
4	3x3	128x128x32	128x128x32	1x1x32x4, 3x3x4x4, 1x1x4x32
5	3x3	128x128x32	128x128x32	1x1x32x4, 3x3x4x4, 1x1x4x32
6	3x3	128x128x32	128x128x32	1x1x32x4, 3x3x4x4, 1x1x4x32
7	1x1	128x128x32	128x128x64	1x1x32x4, 1x1x4x64
8	3x3	128x128x64	128x128x4	N/A

Is It Even Worth It?

Layer	Filter	Input	Output	Mask
3	3x3	128x128x32	128x128x32	1x1x32x4, 3x3x4x4, 1x1x4x32

To calculate the output feature map (OFM) requires

$$3 \times 3 \times 32 \times 32 \times 128 \times 128 = 150\text{M calculations}$$

To calculate the mask using our mask generation model architecture requires

$$128 \times 128 \times 32 \times 4 + 128 \times 128 \times 3 \times 3 \times 4 + 128 \times 128 \times 4 \times 32 = 4.8\text{M calculations}$$

Is It Even Worth It?

Layer	Filter	Input	Output	Mask
3	3x3	128x128x32	128x128x32	1x1x32x4, 3x3x4x4, 1x1x4x32

Calculation cost of calculating one 1x1x1 output in the OFM

$$3 \times 3 \times 32 \times 32 = 288$$

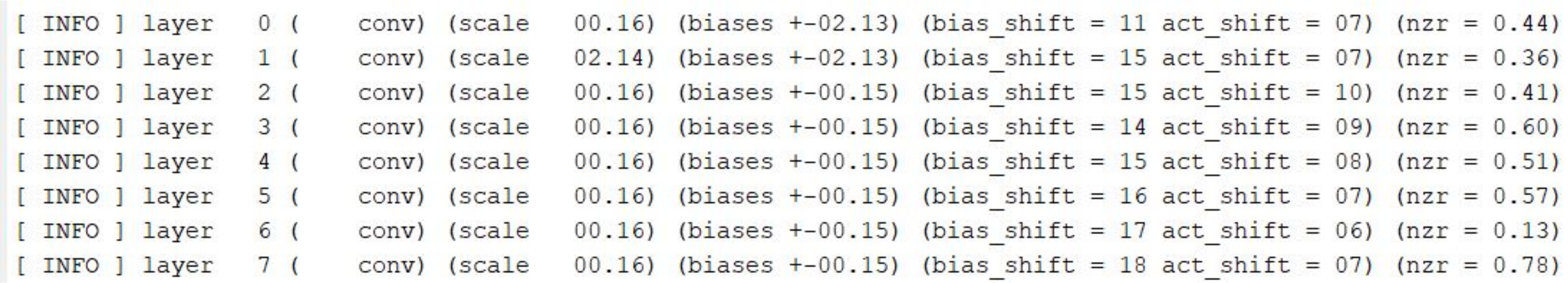
Number of zeros that need to be predicted In order for the calculation to “break even”

$$\frac{4784128(=\text{multiplications for mask calculation})}{288} = 16,611.6$$

$$\frac{16,611.6}{128 \times 128 \times 32(=\text{number pixels in OFM})} \approx 0.0317 = 3.17\%$$

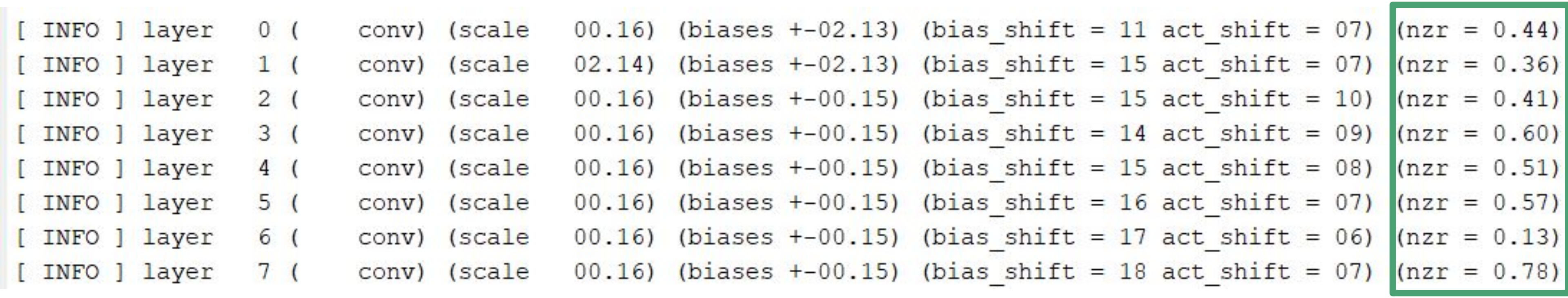
At least 3.17% of the outputs should be 0 in order for the mask generation calculation to break even

Is It Even Worth It?



```
[ INFO ] layer   0 (     conv) (scale   00.16) (biases +-02.13) (bias_shift = 11 act_shift = 07) (nzr = 0.44)
[ INFO ] layer   1 (     conv) (scale   02.14) (biases +-02.13) (bias_shift = 15 act_shift = 07) (nzr = 0.36)
[ INFO ] layer   2 (     conv) (scale   00.16) (biases +-00.15) (bias_shift = 15 act_shift = 10) (nzr = 0.41)
[ INFO ] layer   3 (     conv) (scale   00.16) (biases +-00.15) (bias_shift = 14 act_shift = 09) (nzr = 0.60)
[ INFO ] layer   4 (     conv) (scale   00.16) (biases +-00.15) (bias_shift = 15 act_shift = 08) (nzr = 0.51)
[ INFO ] layer   5 (     conv) (scale   00.16) (biases +-00.15) (bias_shift = 16 act_shift = 07) (nzr = 0.57)
[ INFO ] layer   6 (     conv) (scale   00.16) (biases +-00.15) (bias_shift = 17 act_shift = 06) (nzr = 0.13)
[ INFO ] layer   7 (     conv) (scale   00.16) (biases +-00.15) (bias_shift = 18 act_shift = 07) (nzr = 0.78)
```

Is It Even Worth It?



```
[ INFO ] layer   0 (      conv) (scale   00.16) (biases +-02.13) (bias_shift = 11 act_shift = 07) (nzr = 0.44)
[ INFO ] layer   1 (      conv) (scale   02.14) (biases +-02.13) (bias_shift = 15 act_shift = 07) (nzr = 0.36)
[ INFO ] layer   2 (      conv) (scale   00.16) (biases +-00.15) (bias_shift = 15 act_shift = 10) (nzr = 0.41)
[ INFO ] layer   3 (      conv) (scale   00.16) (biases +-00.15) (bias_shift = 14 act_shift = 09) (nzr = 0.60)
[ INFO ] layer   4 (      conv) (scale   00.16) (biases +-00.15) (bias_shift = 15 act_shift = 08) (nzr = 0.51)
[ INFO ] layer   5 (      conv) (scale   00.16) (biases +-00.15) (bias_shift = 16 act_shift = 07) (nzr = 0.57)
[ INFO ] layer   6 (      conv) (scale   00.16) (biases +-00.15) (bias_shift = 17 act_shift = 06) (nzr = 0.13)
[ INFO ] layer   7 (      conv) (scale   00.16) (biases +-00.15) (bias_shift = 18 act_shift = 07) (nzr = 0.78)
```

All outputs from each layer have more than 3.17% of zero-ness

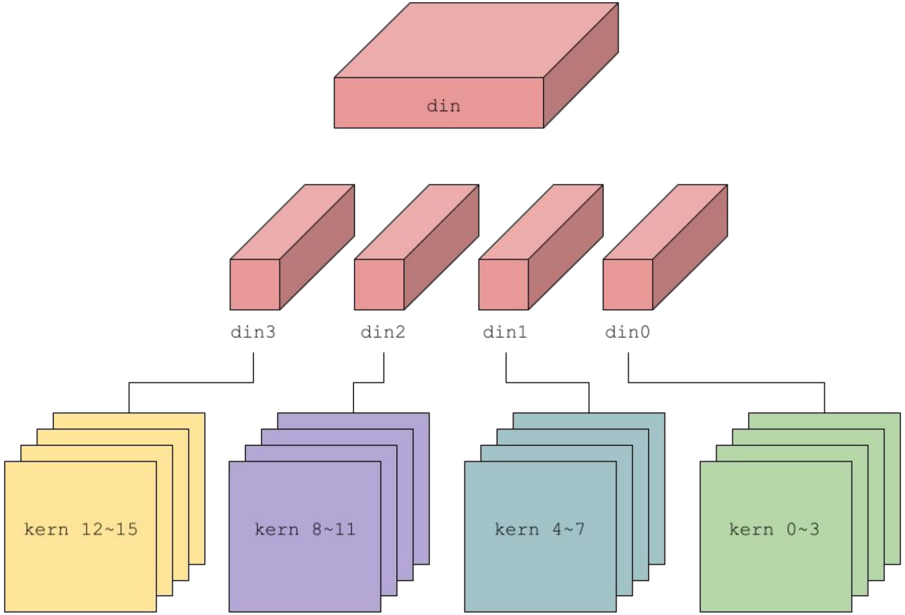
Yes, zero skipping via mask generation is worth it

Implementation Specifications/Constraints

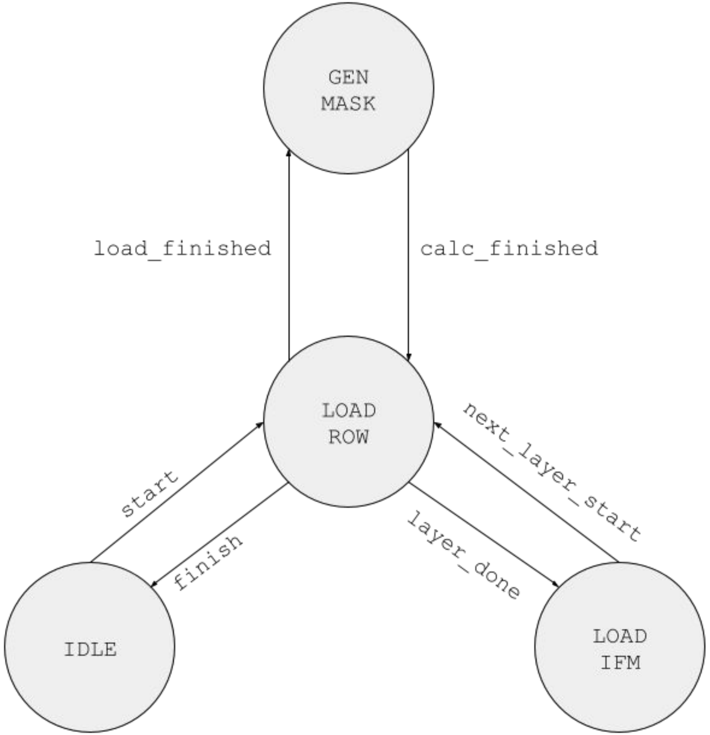
Mask_maker

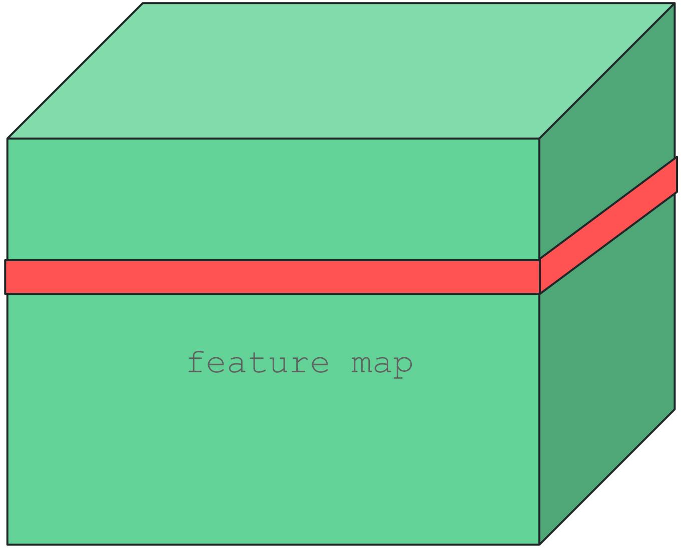
Perform convolution with 1x4x4 section (din) of feature map

16 total kernels

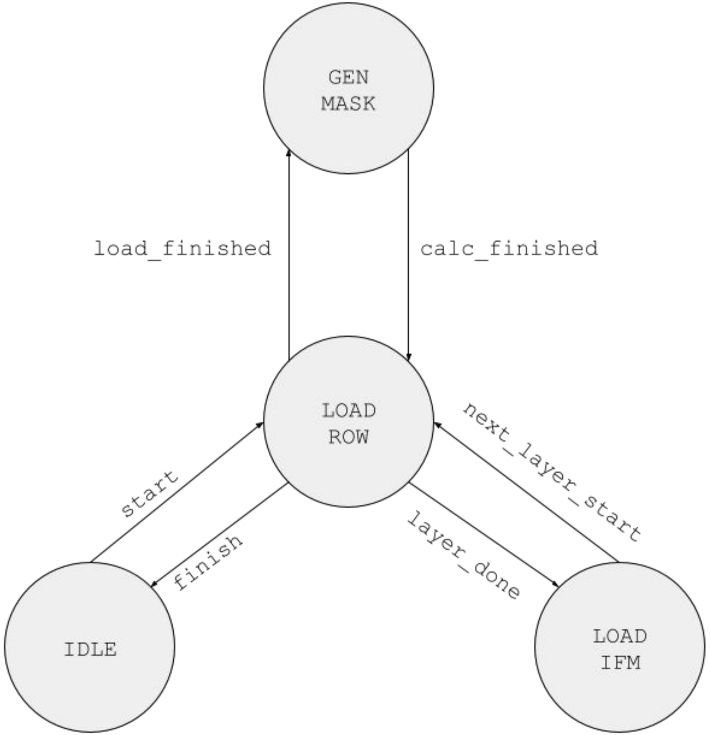


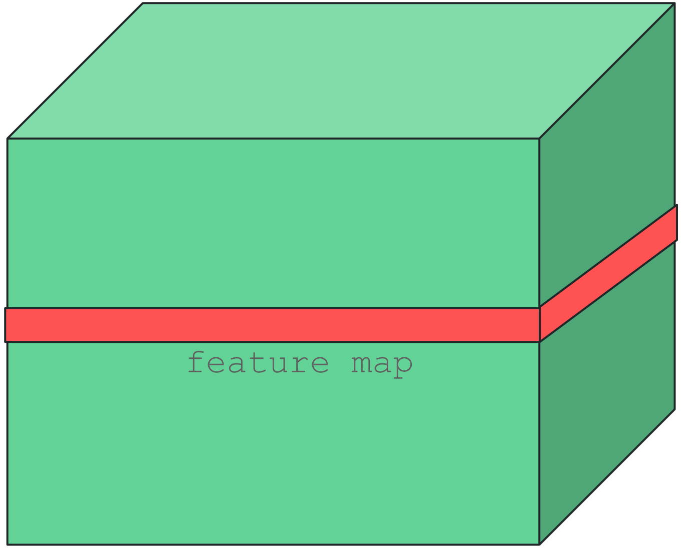
Maskgen FSM


GEN MASK
load_finished
calc_finished
LOAD ROW
start
finish
next_layer_start
layer_done
IDLE
LOAD IFM

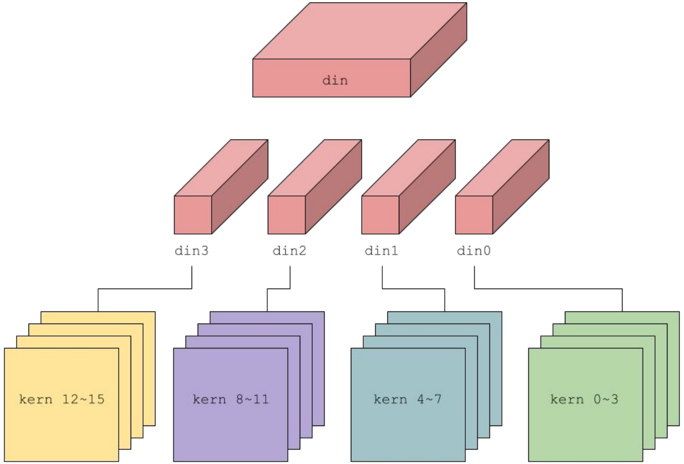

feature map

Maskgen FSM


GEN MASK
load_finished
calc_finished
LOAD ROW
start
finish
next_layer_start
layer_done
IDLE
LOAD IFM


feature map

Layer 1 Convolution Operations



input : 128 x 128 x 32

output : 128 x 128 x 4

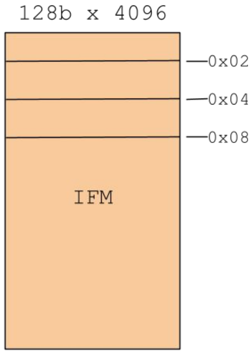
weight : 1 x 1 x 32 x 4

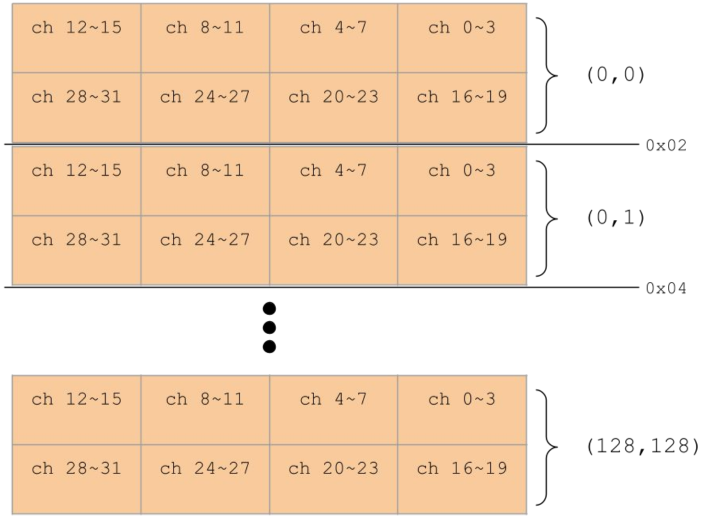
din only takes in 32 bits, so only 4 pixels (4 x 8 bit = 32 bit) channel-wise enter at a time

Accumulate occurs after 8 “batches” of 4 pixels

Inputs

Input feature map from SW code (convout)



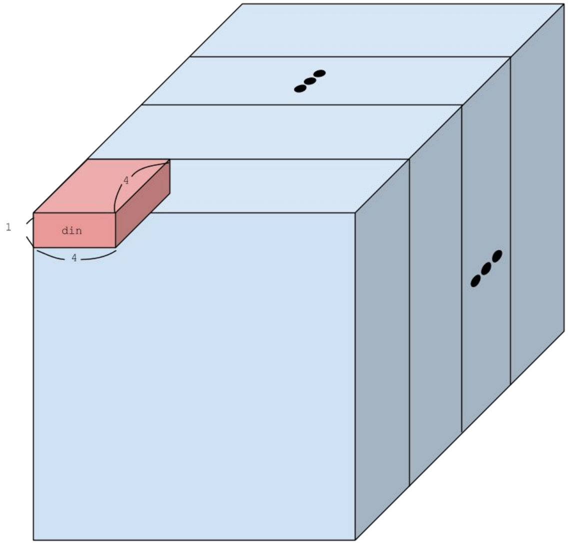


ch 12~15	ch 8~11	ch 4~7	ch 0~3
ch 28~31	ch 24~27	ch 20~23	ch 16~19

ch 12~15	ch 8~11	ch 4~7	ch 0~3
ch 28~31	ch 24~27	ch 20~23	ch 16~19

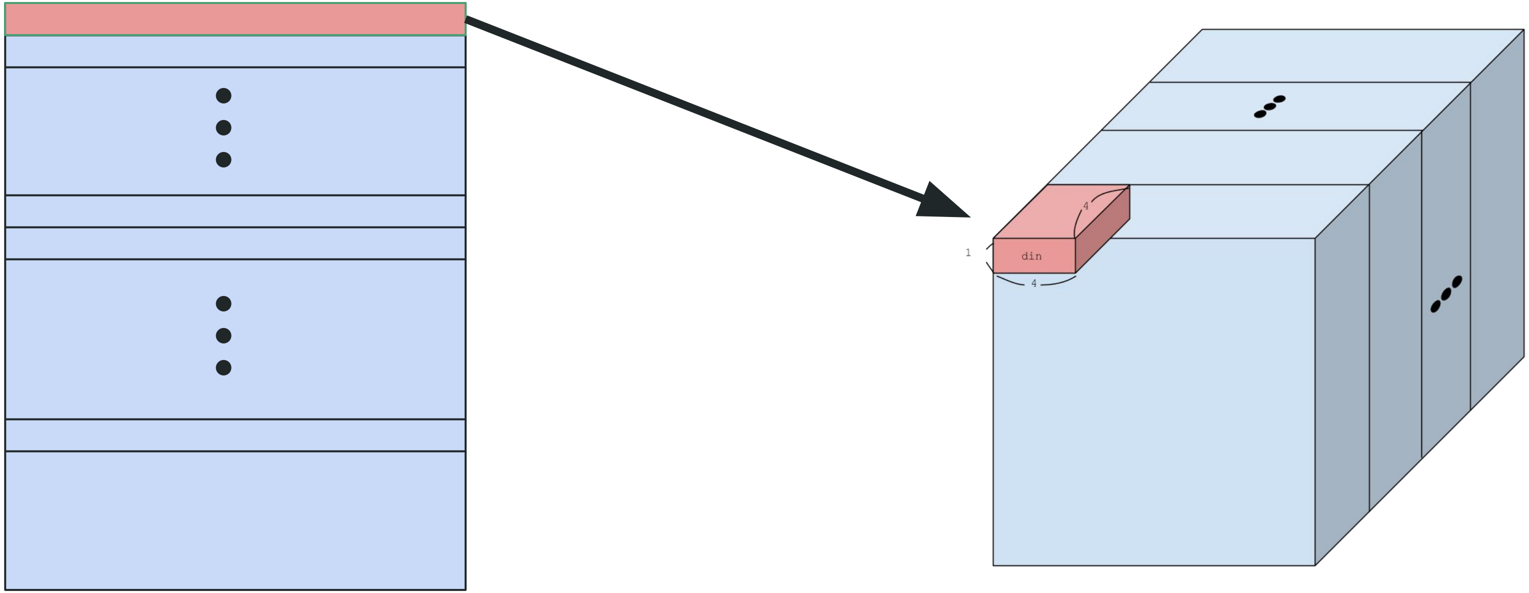
ch 12~15	ch 8~11	ch 4~7	ch 0~3
ch 28~31	ch 24~27	ch 20~23	ch 16~19

≠



Inputs

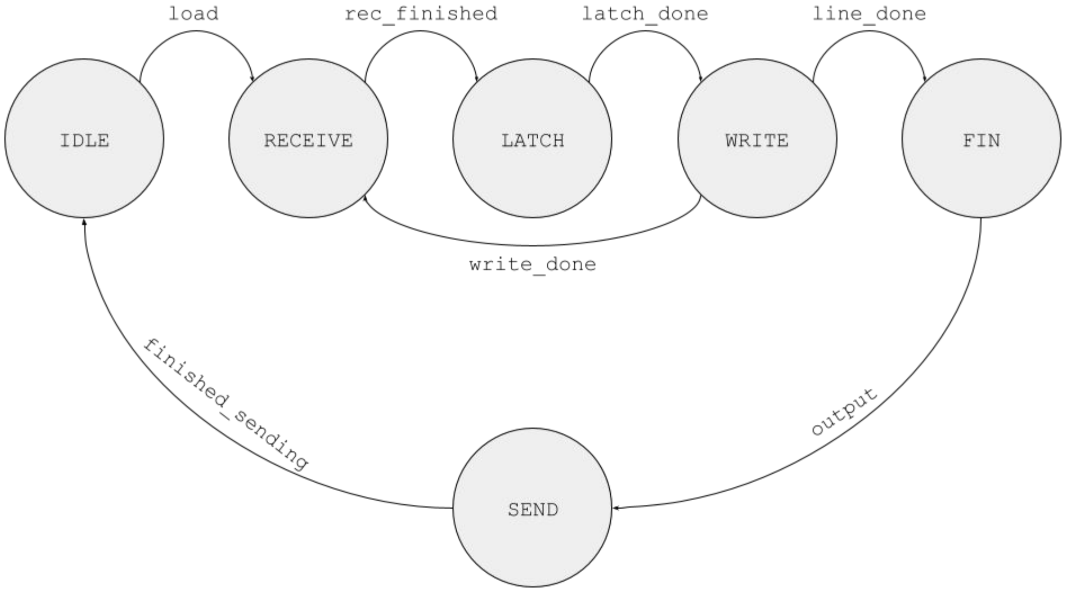
Input feature map from SW code (convout)



Layer 1 Line Buffer

Holds 1 row worth of data (128x1x4 bytes)

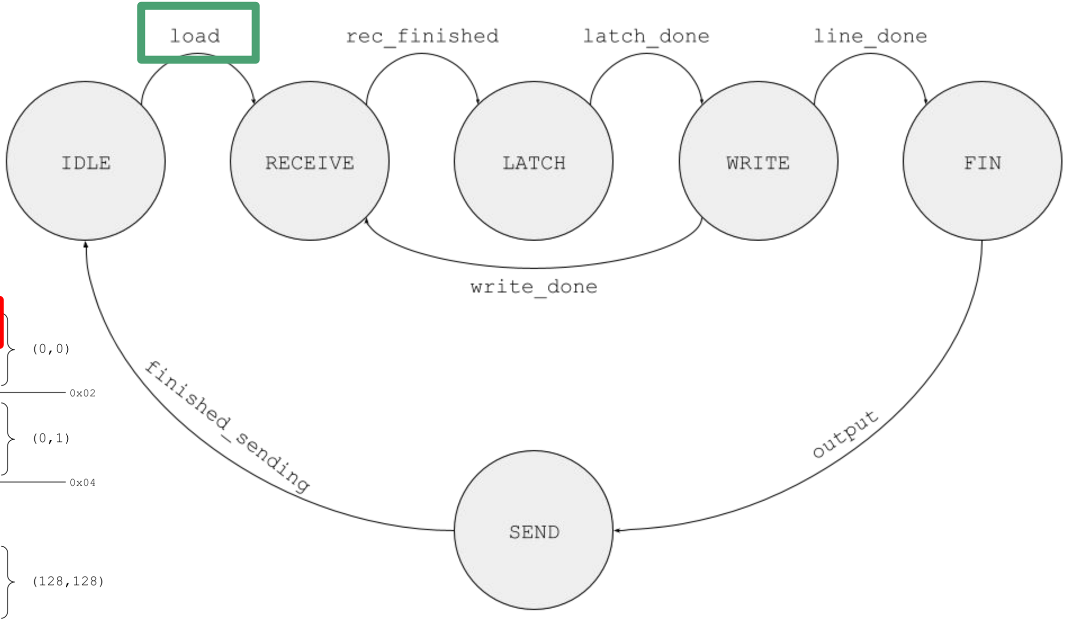
Internally restructures data so 1 address holds 128b



Layer 1 Line Buffer

In RECEIVE state, line buffer receives data(128b) from un-organized IFM.

Inputs stored in temporary buffer.

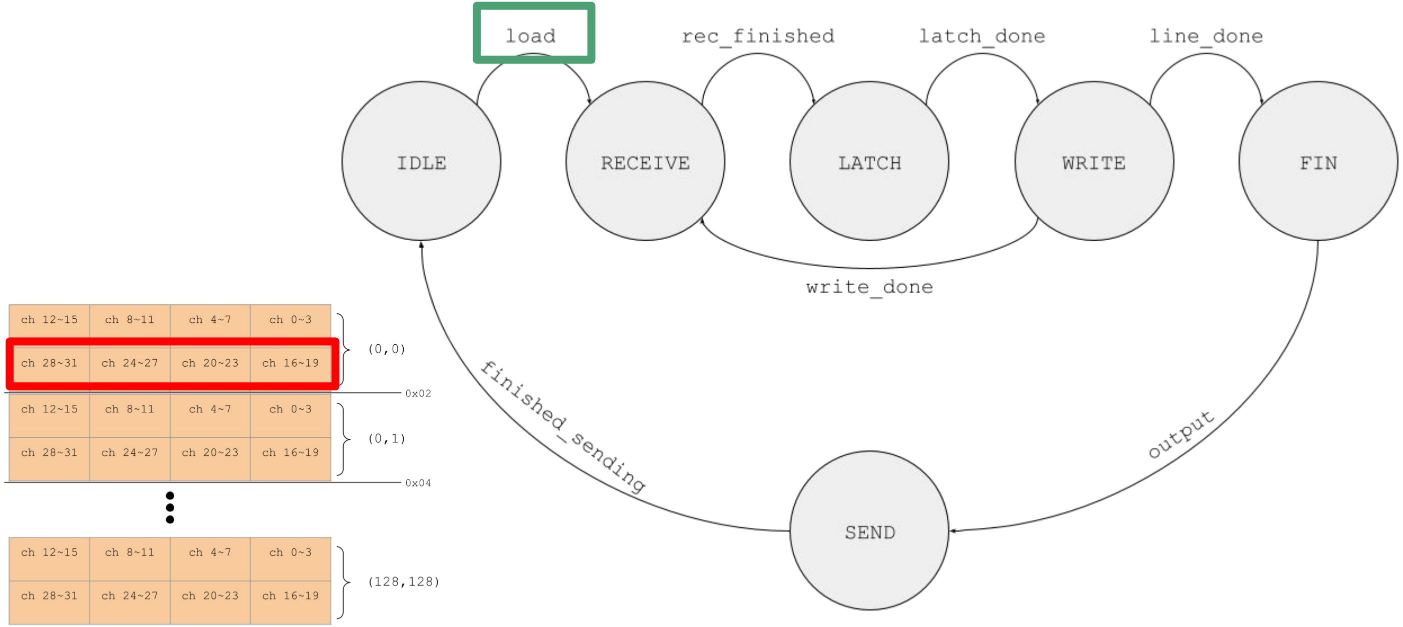


ch 12~15	ch 8~11	ch 4~7	ch 0~3	(0,0)	
ch 28~31	ch 24~27	ch 20~23	ch 16~19		0x02
ch 12~15	ch 8~11	ch 4~7	ch 0~3	(0,1)	
ch 28~31	ch 24~27	ch 20~23	ch 16~19		0x04
⋮					
ch 12~15	ch 8~11	ch 4~7	ch 0~3	(128,128)	
ch 28~31	ch 24~27	ch 20~23	ch 16~19		

Layer 1 Line Buffer

In RECEIVE state, line buffer receives data(128b) from un-organized IFM.

Inputs stored in temporary buffer.

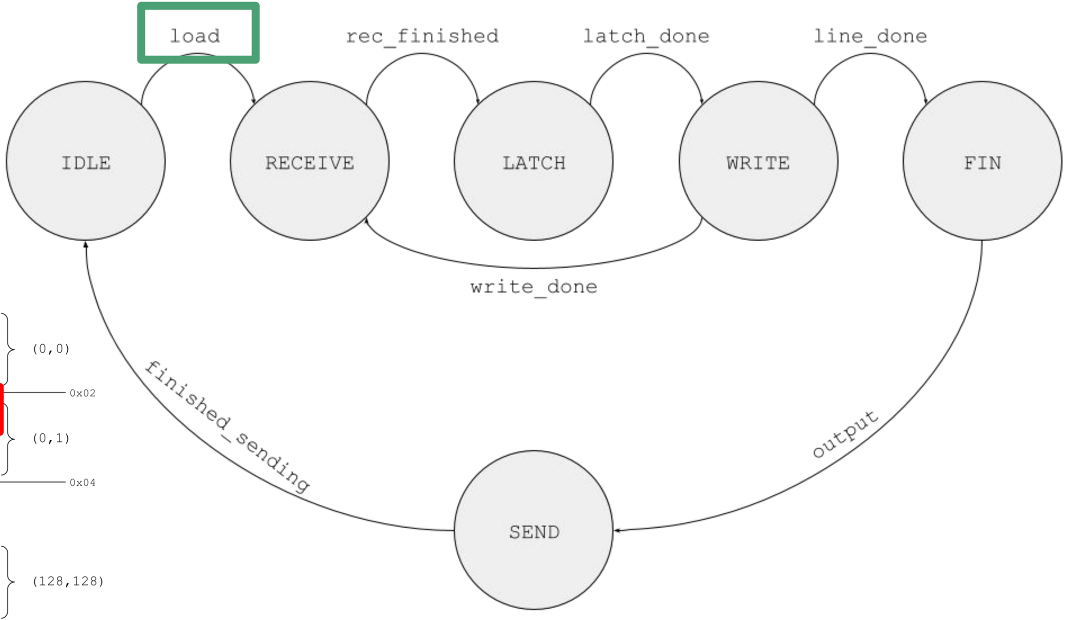


ch 12~15	ch 8~11	ch 4~7	ch 0~3	(0,0)
ch 28~31	ch 24~27	ch 20~23	ch 16~19	
				0x02
ch 12~15	ch 8~11	ch 4~7	ch 0~3	(0,1)
ch 28~31	ch 24~27	ch 20~23	ch 16~19	
				0x04
⋮				
ch 12~15	ch 8~11	ch 4~7	ch 0~3	(128,128)
ch 28~31	ch 24~27	ch 20~23	ch 16~19	

Layer 1 Line Buffer

In RECEIVE state, line buffer receives data(128b) from un-organized IFM.

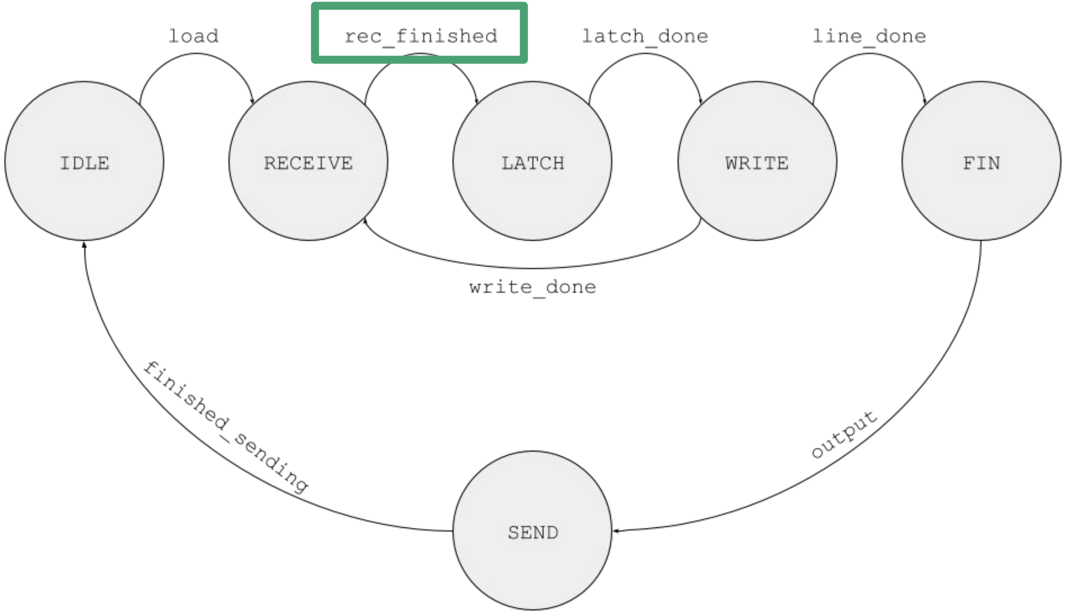
Inputs stored in temporary buffer.



ch 12~15	ch 8~11	ch 4~7	ch 0~3	(0,0)	
ch 28~31	ch 24~27	ch 20~23	ch 16~19		0x02
ch 12~15	ch 8~11	ch 4~7	ch 0~3	(0,1)	
ch 28~31	ch 24~27	ch 20~23	ch 16~19		0x04
⋮					
ch 12~15	ch 8~11	ch 4~7	ch 0~3	(128,128)	
ch 28~31	ch 24~27	ch 20~23	ch 16~19		

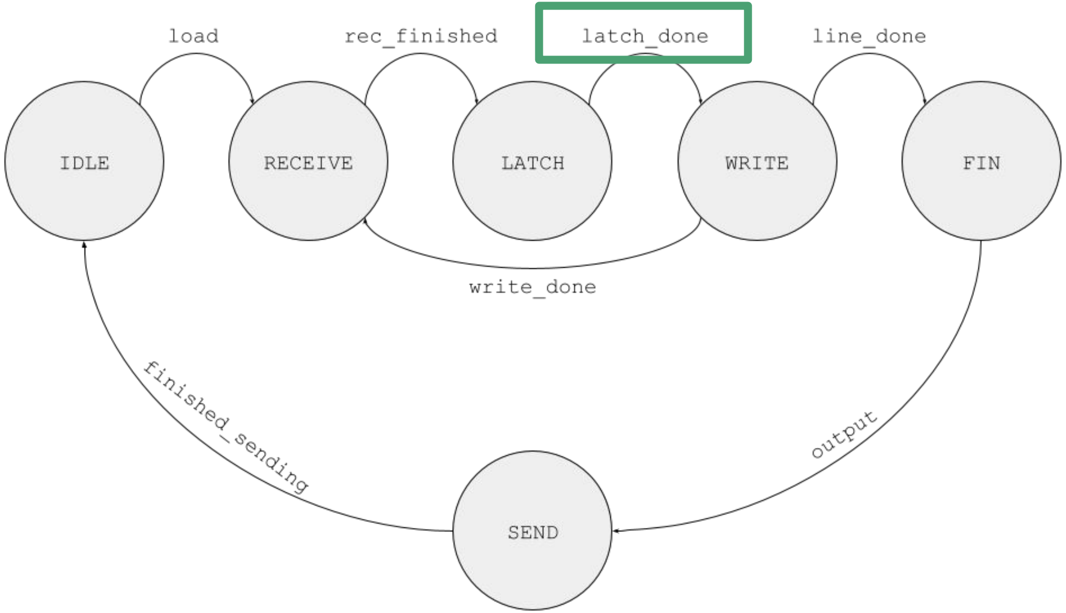
Layer 1 Line Buffer

When 128b from the IFM received 8 times, go to LATCH state



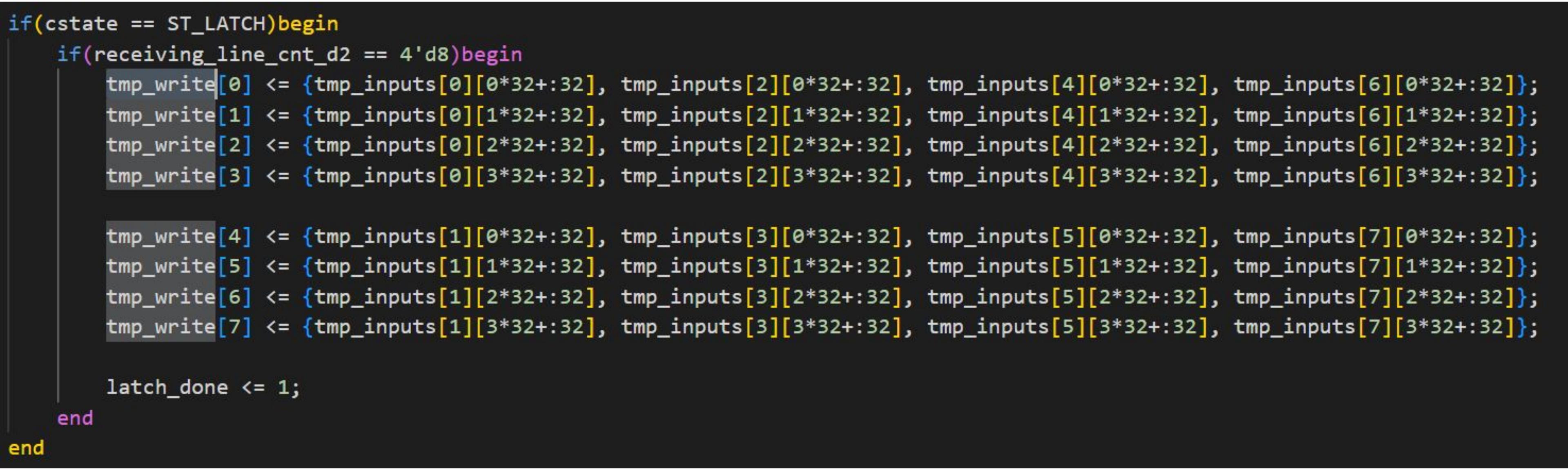
Layer 1 Line Buffer

In LATCH state, reorganize the received weights so that 1 address holds data for one 1x4x4 convolution.



Layer 1 Line Buffer

After weights reorganized, move onto WRITE state

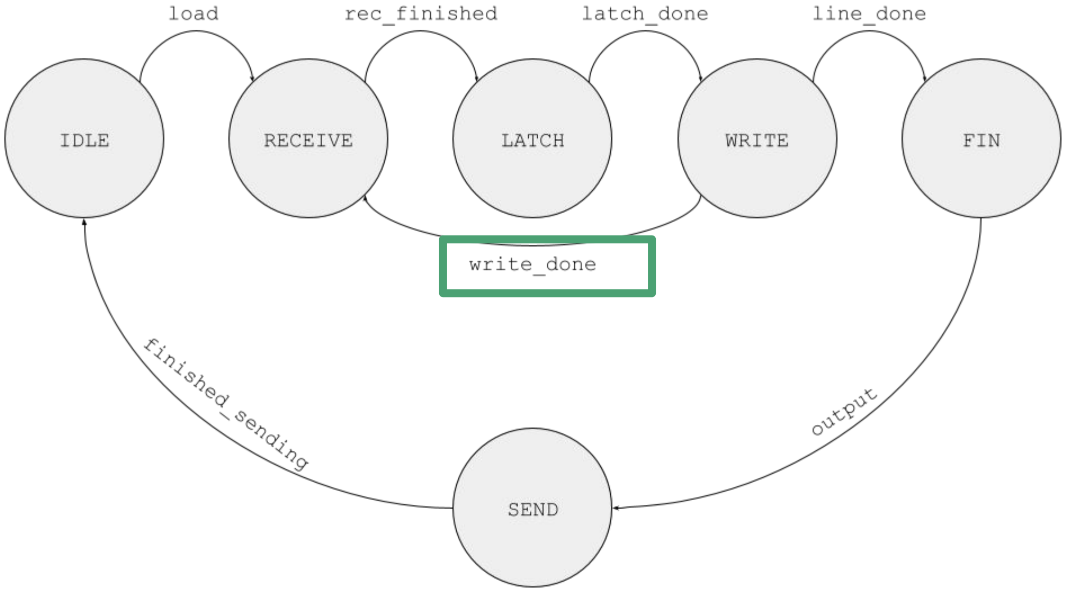


```
if(cstate == ST_LATCH)begin
    if(receiving_line_cnt_d2 == 4'd8)begin
        tmp_write[0] <= {tmp_inputs[0][0*32+:32], tmp_inputs[2][0*32+:32], tmp_inputs[4][0*32+:32], tmp_inputs[6][0*32+:32]};
        tmp_write[1] <= {tmp_inputs[0][1*32+:32], tmp_inputs[2][1*32+:32], tmp_inputs[4][1*32+:32], tmp_inputs[6][1*32+:32]};
        tmp_write[2] <= {tmp_inputs[0][2*32+:32], tmp_inputs[2][2*32+:32], tmp_inputs[4][2*32+:32], tmp_inputs[6][2*32+:32]};
        tmp_write[3] <= {tmp_inputs[0][3*32+:32], tmp_inputs[2][3*32+:32], tmp_inputs[4][3*32+:32], tmp_inputs[6][3*32+:32]};

        tmp_write[4] <= {tmp_inputs[1][0*32+:32], tmp_inputs[3][0*32+:32], tmp_inputs[5][0*32+:32], tmp_inputs[7][0*32+:32]};
        tmp_write[5] <= {tmp_inputs[1][1*32+:32], tmp_inputs[3][1*32+:32], tmp_inputs[5][1*32+:32], tmp_inputs[7][1*32+:32]};
        tmp_write[6] <= {tmp_inputs[1][2*32+:32], tmp_inputs[3][2*32+:32], tmp_inputs[5][2*32+:32], tmp_inputs[7][2*32+:32]};
        tmp_write[7] <= {tmp_inputs[1][3*32+:32], tmp_inputs[3][3*32+:32], tmp_inputs[5][3*32+:32], tmp_inputs[7][3*32+:32]};

        latch_done <= 1;
    end
end
```

Layer 1 Line Buffer

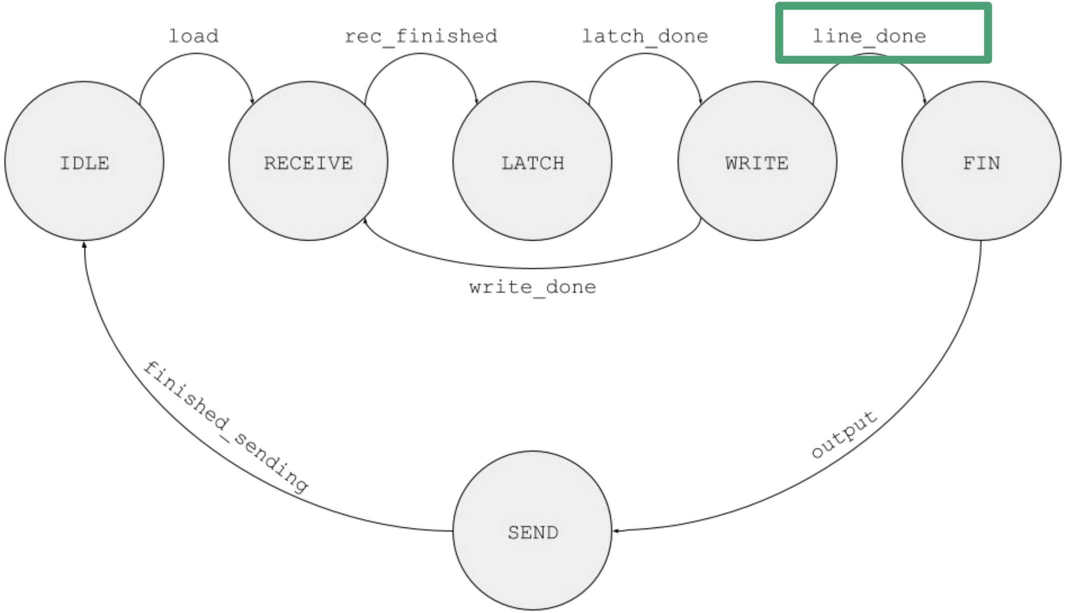
In WRITE state, write the 8 lines of restructured data.

If the entire row is not yet complete, go to RECEIVE state and repeat



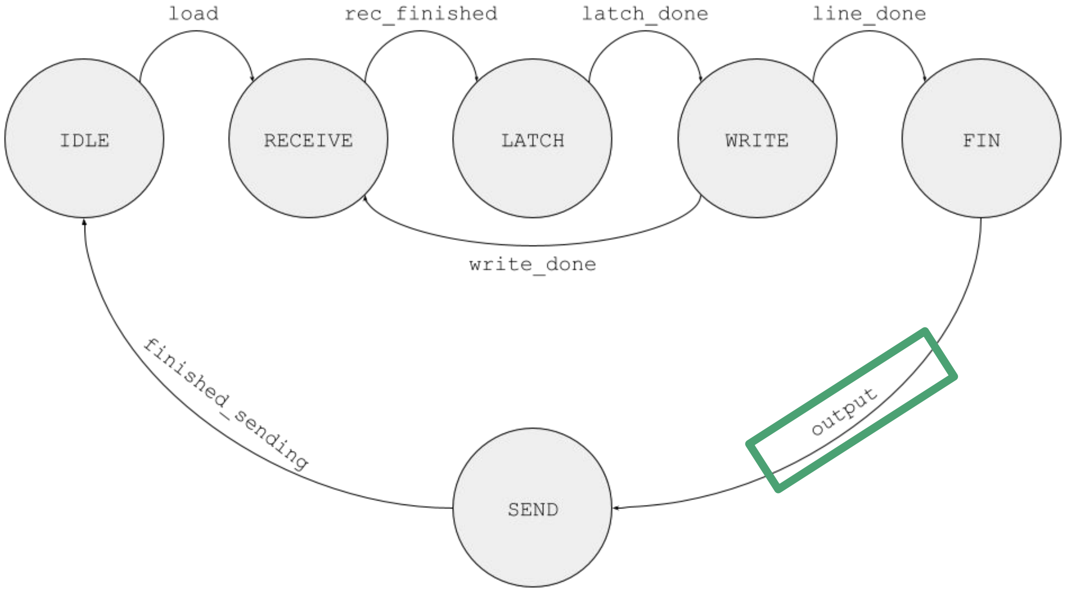
Layer 1 Line Buffer

If entire row has been received and written, go to FIN state (finished receiving)



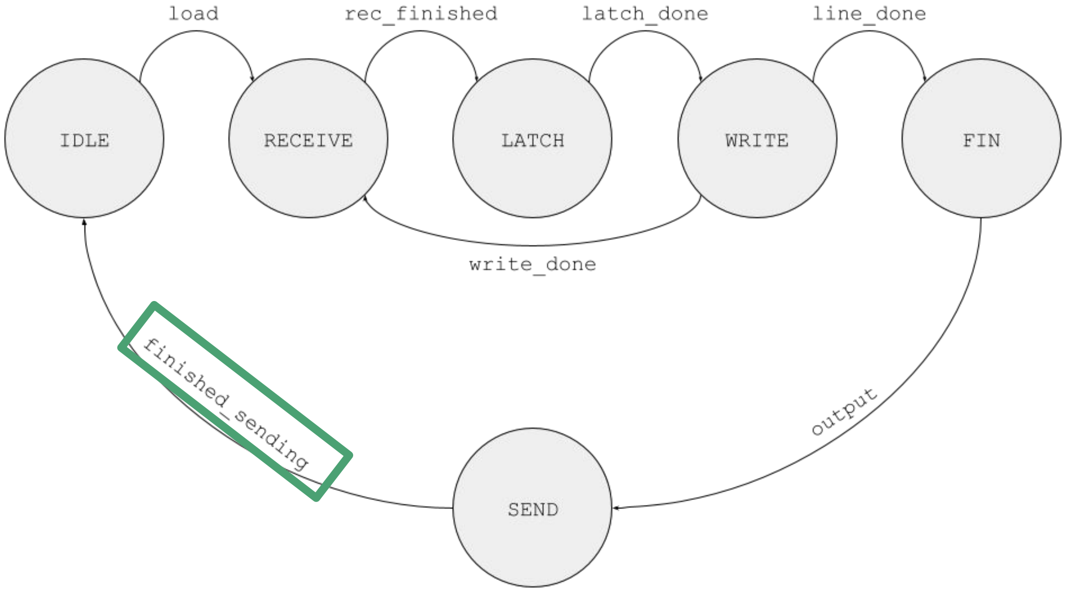
Layer 1 Line Buffer

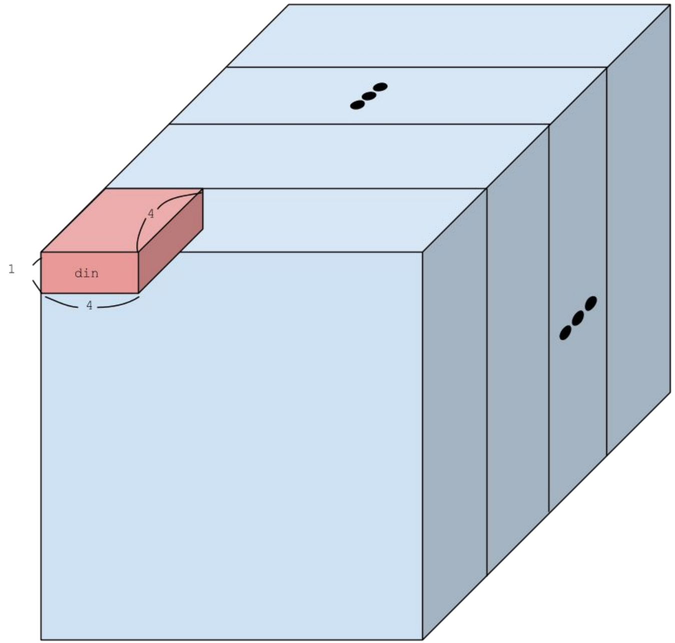
In FIN state, if we get the signal to start reading the reorganized data, move to SEND state



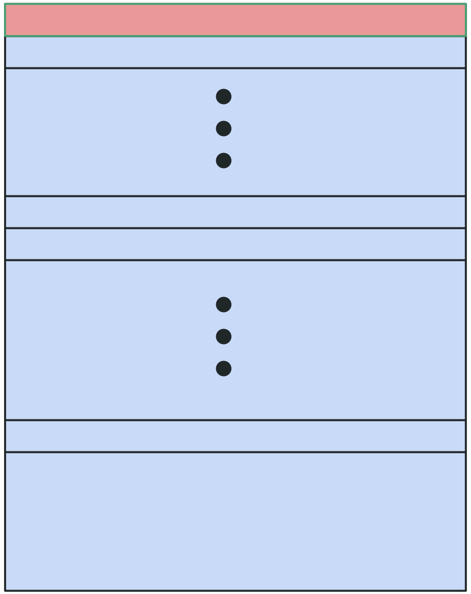
Layer 1 Line Buffer

In SEND state, keep outputting the restructured data until entire row outputted

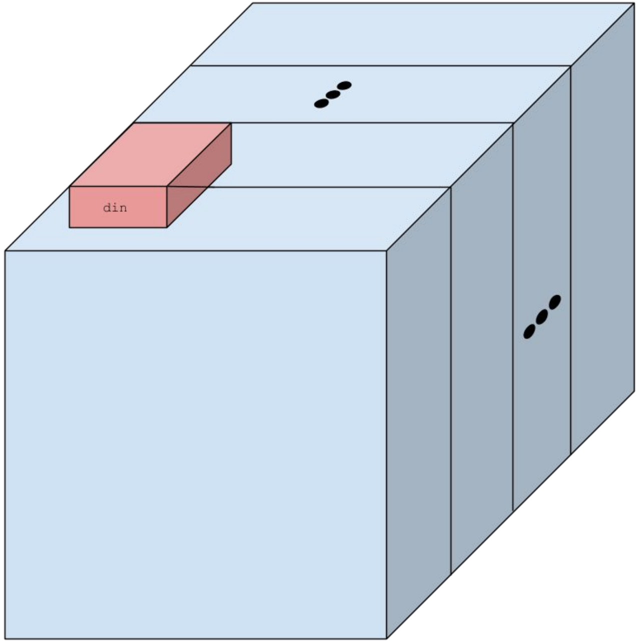




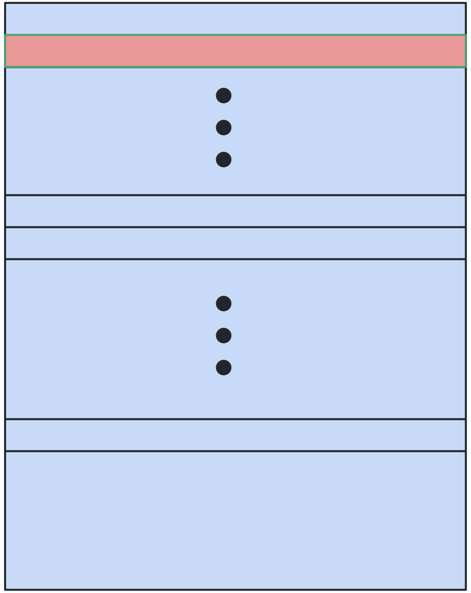
ch_batch_idx = 0

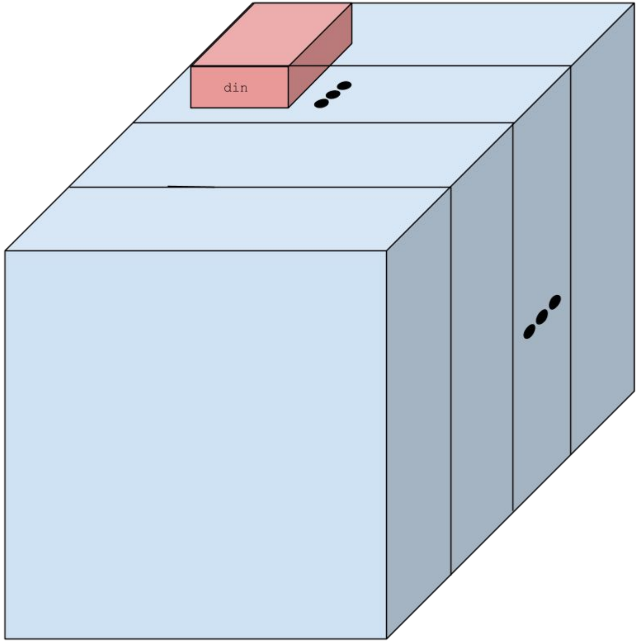


Line buffer FSM in “output” state

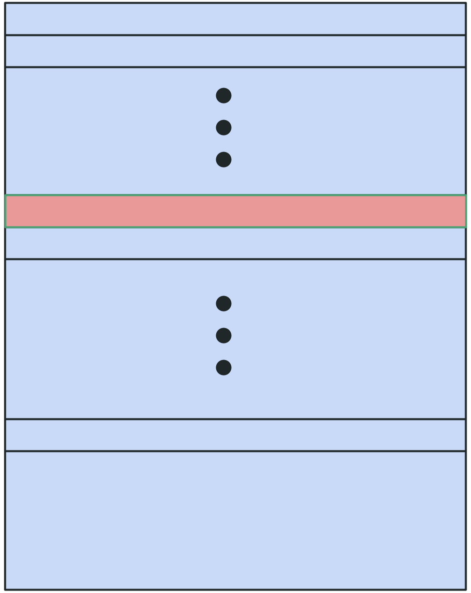


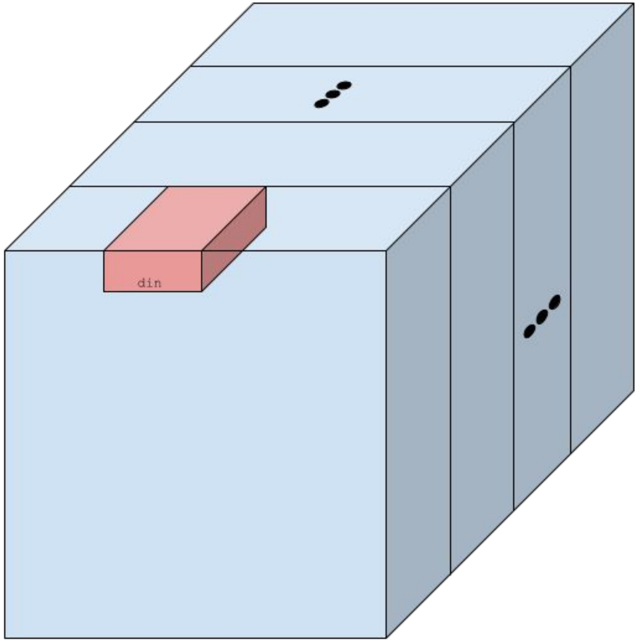
ch_batch_idx = 1



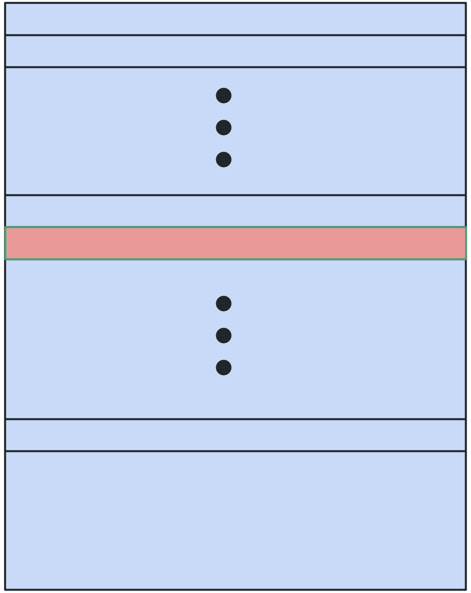


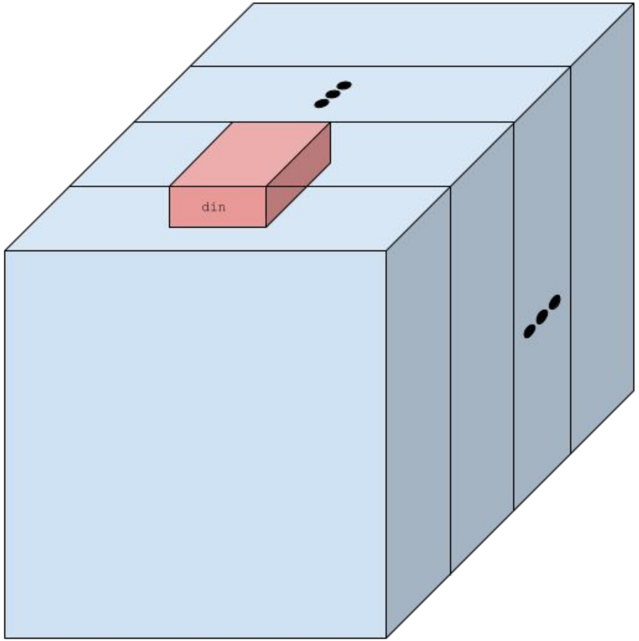
ch_batch_idx = 7



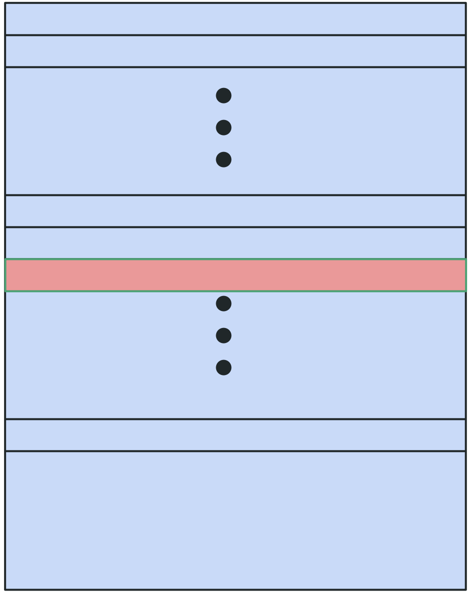


`ch_batch_idx = 0`





ch_batch_idx = 1



Layer 2 Convolution Operations

input : 128 x 128 x 4

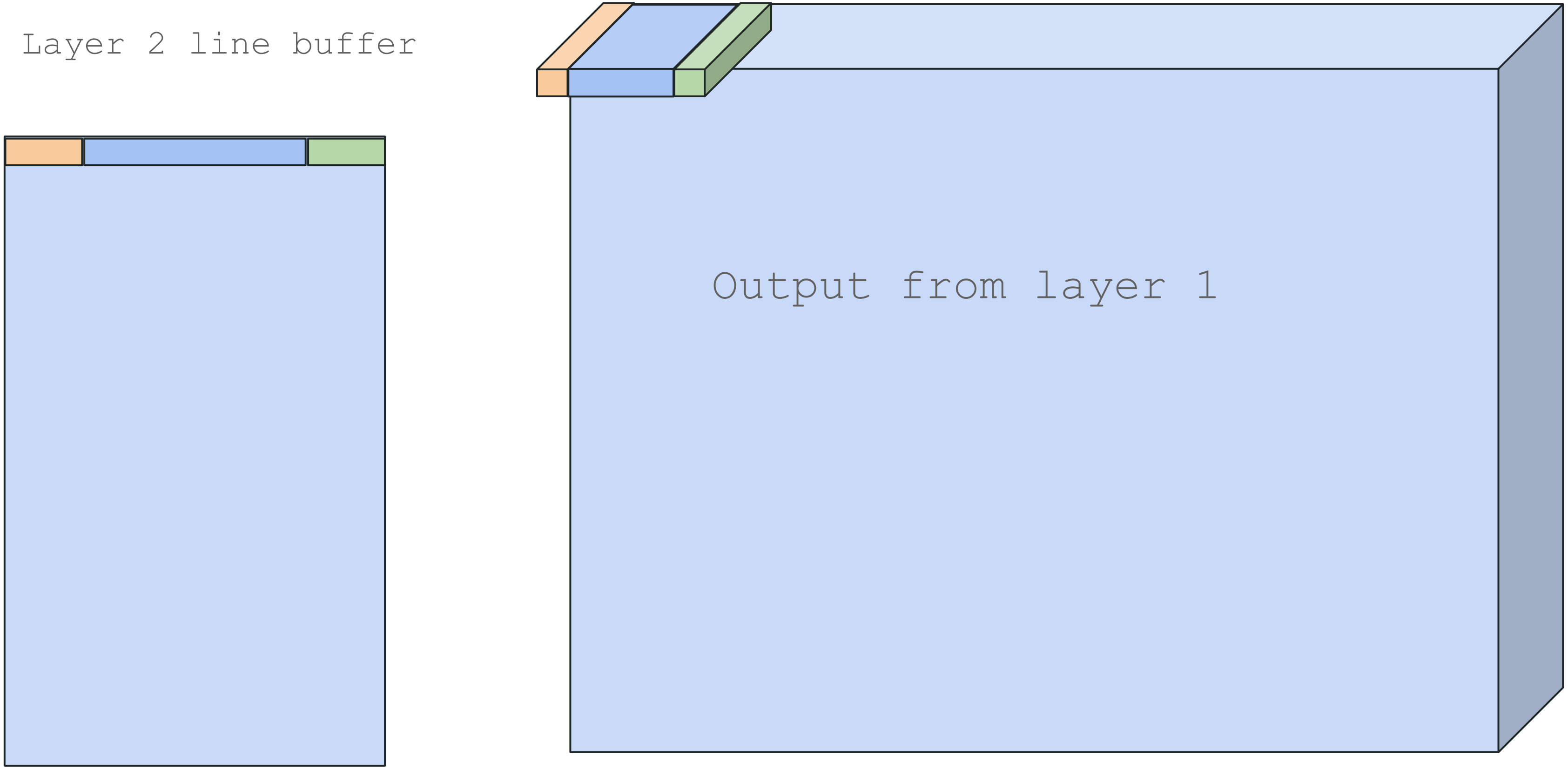
output : 128 x 128 x 4

weights : 3 x 3 x 4 x 4

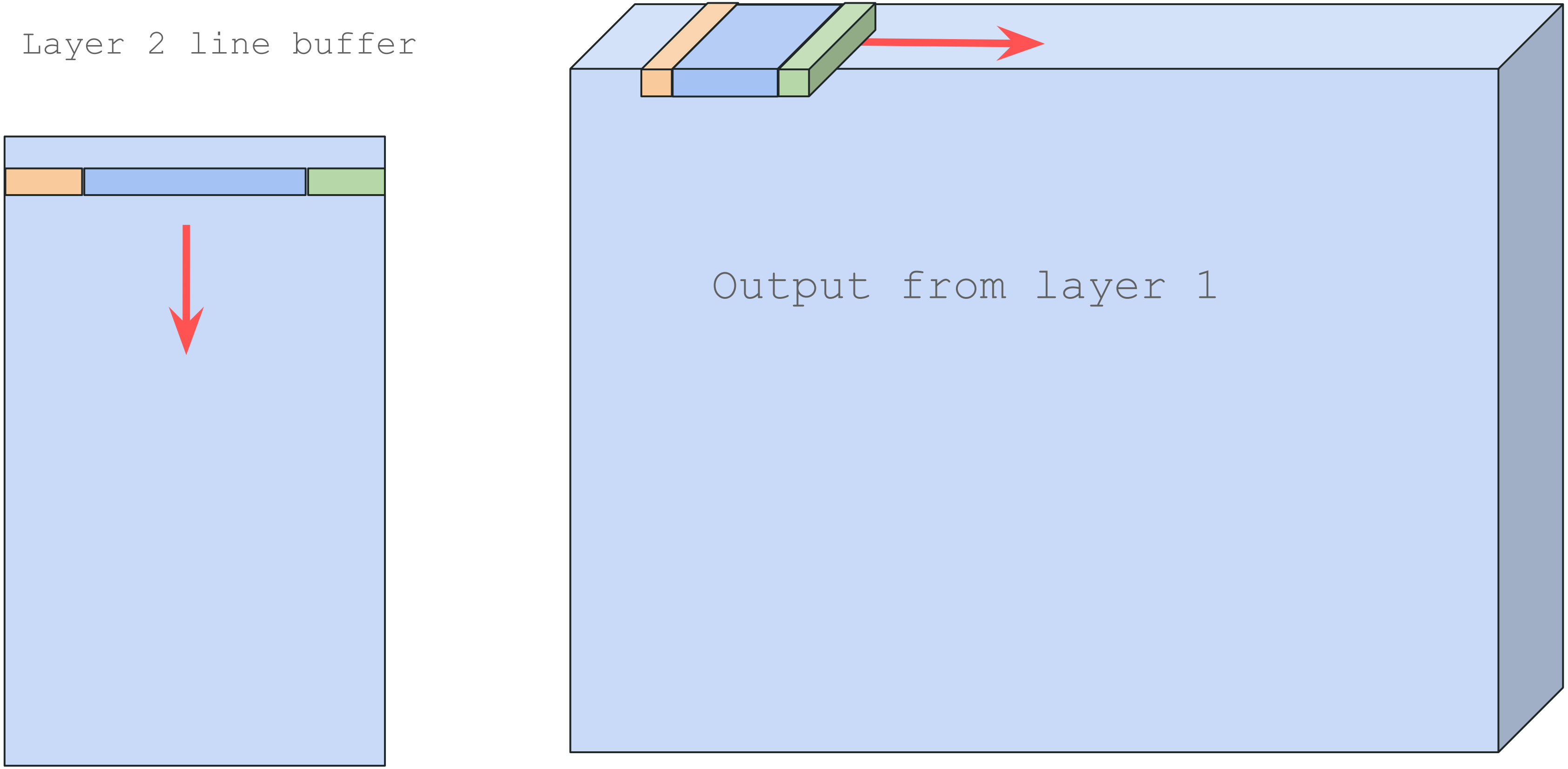
Now kernel size is 3x3, so line buffer needs to hold 3 rows (above, current, below)

FSM controls when to increase row, col, etc

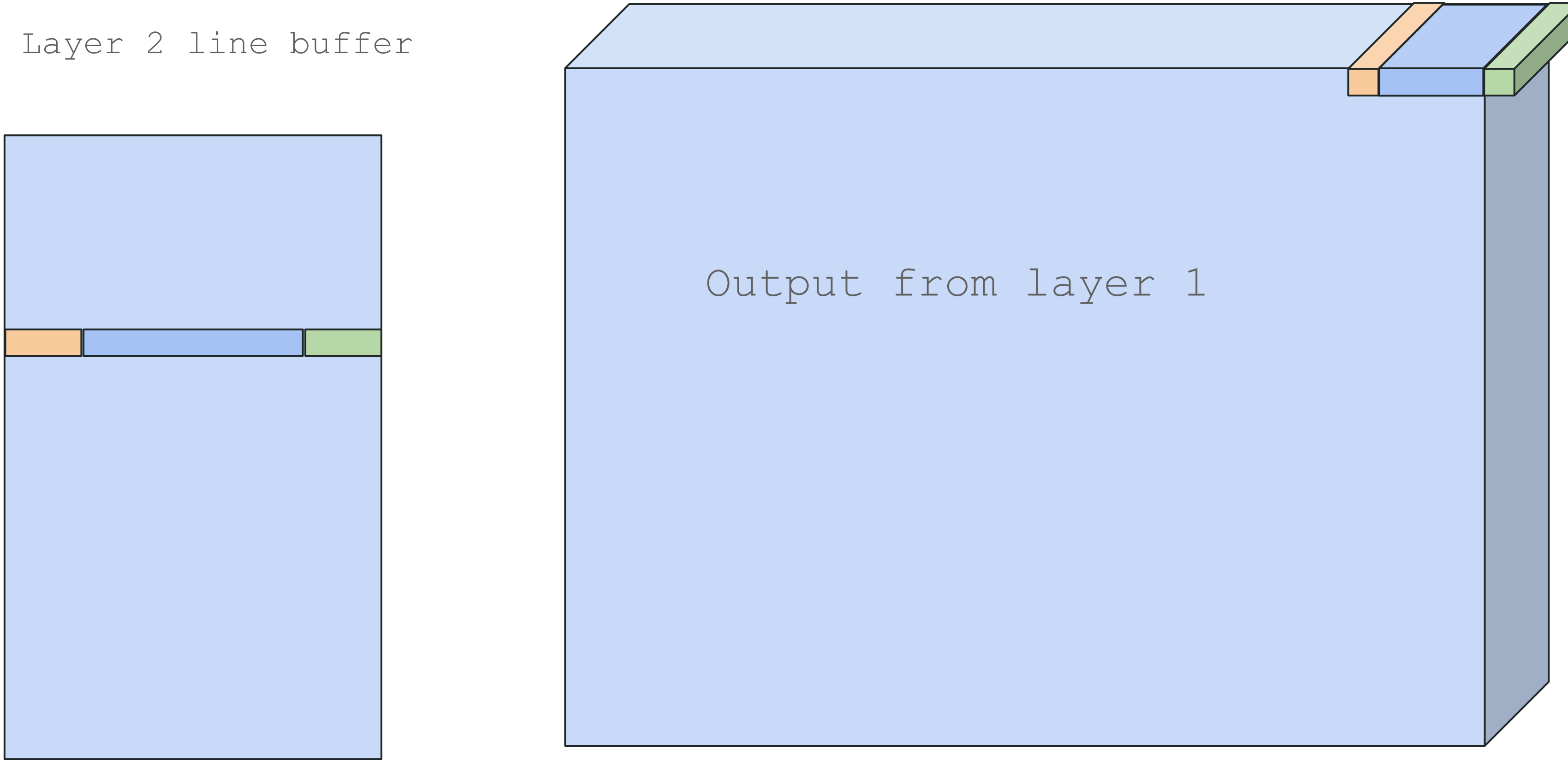
Layer 2 Line Buffer



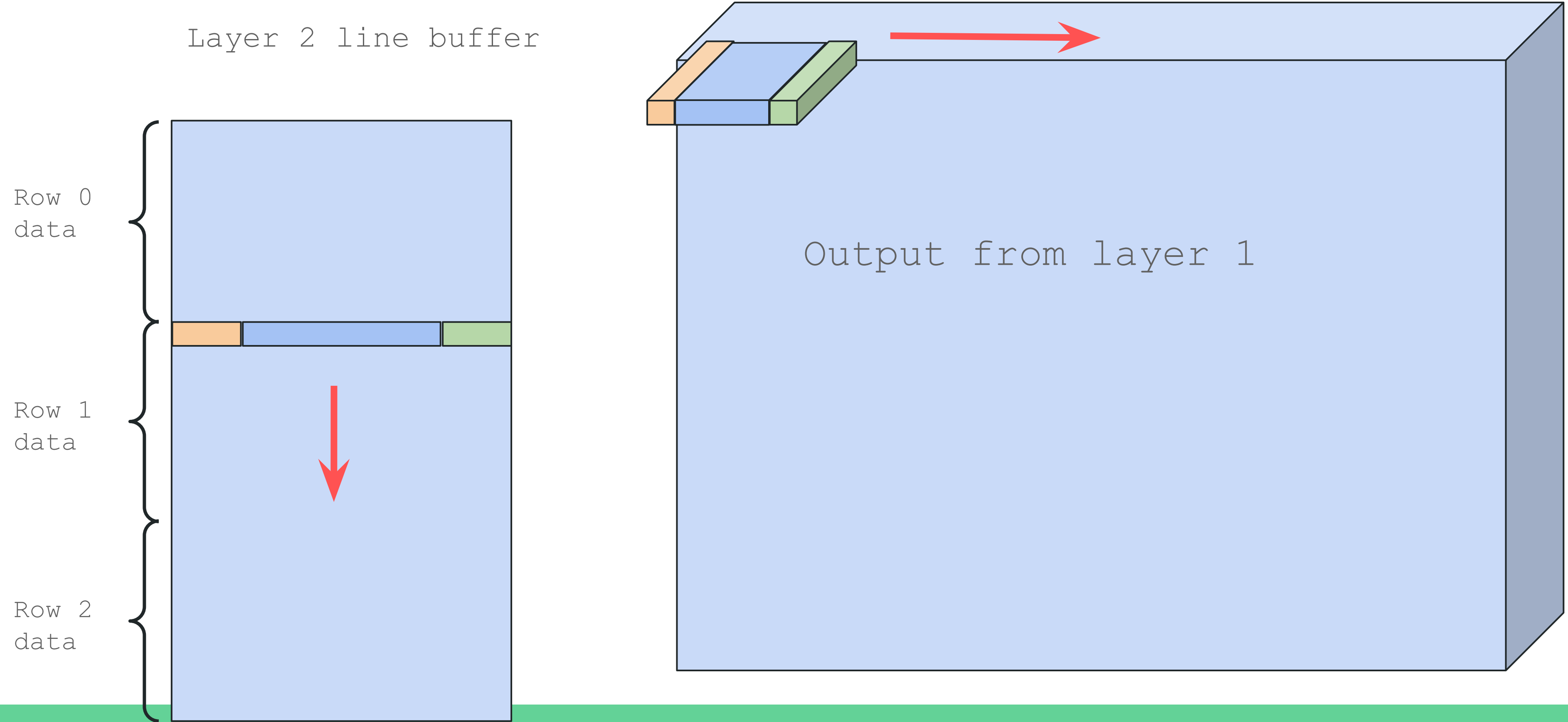
Layer 2 Line Buffer



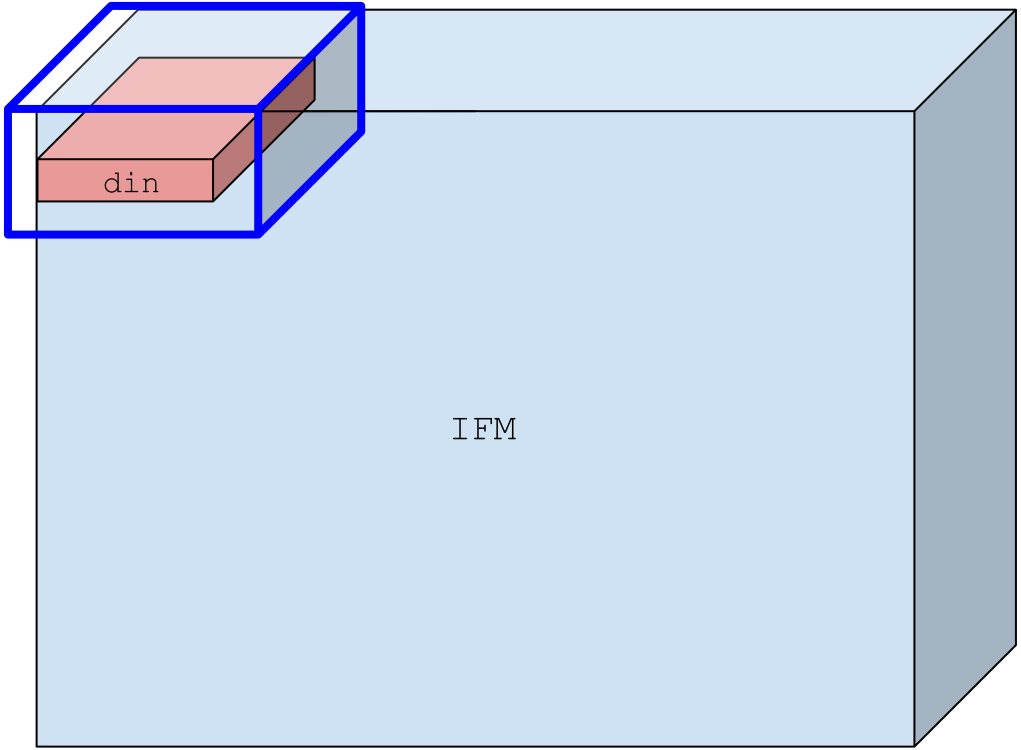
Layer 2 Line Buffer



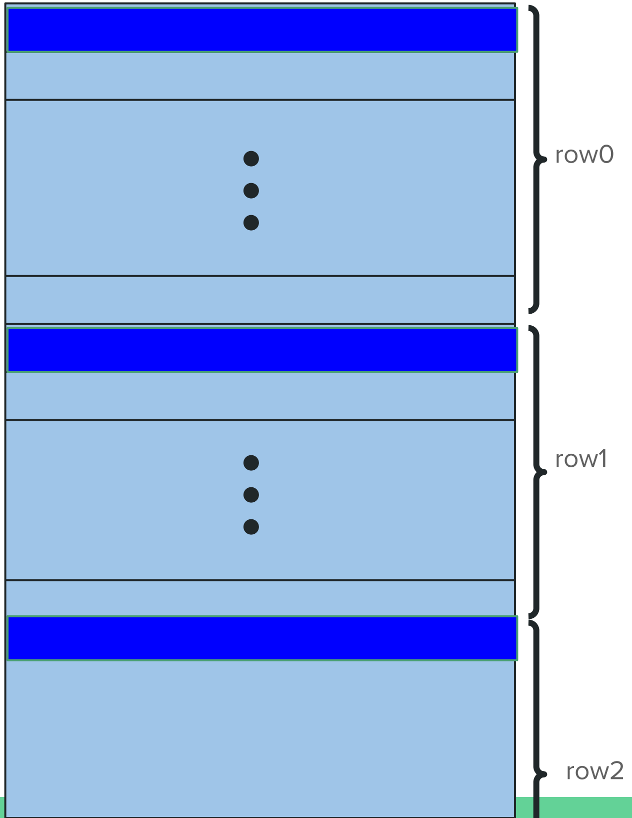
Layer 2 Line Buffer



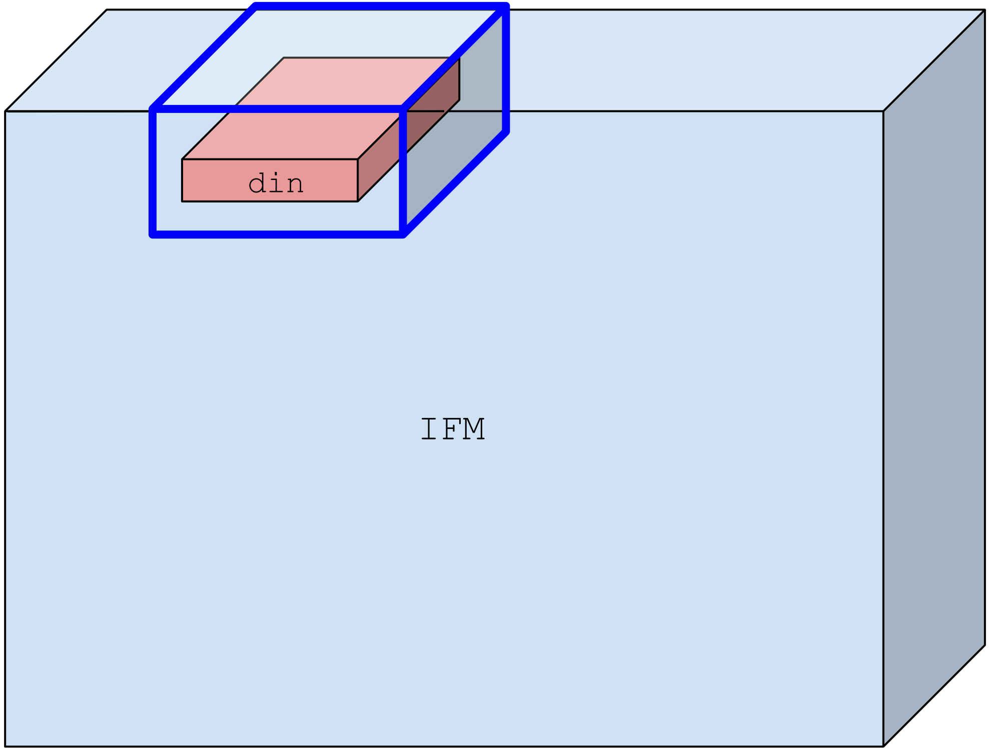
Layer 2 Convolution Operations

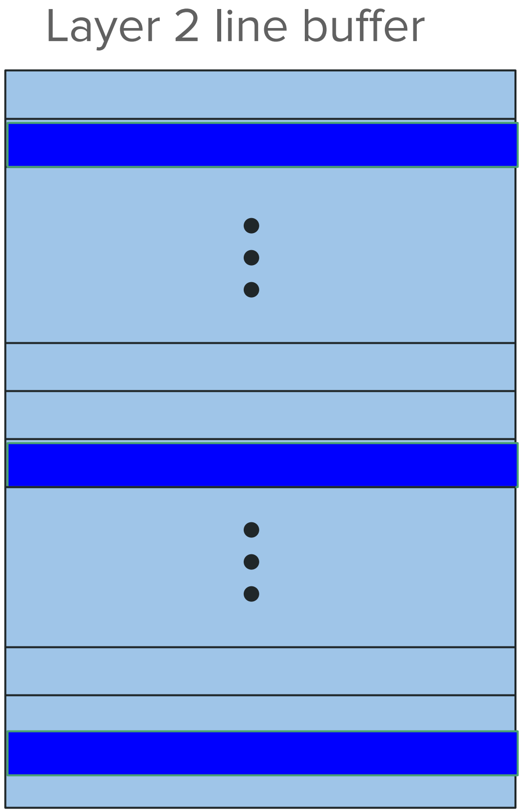


Layer 2 line buffer

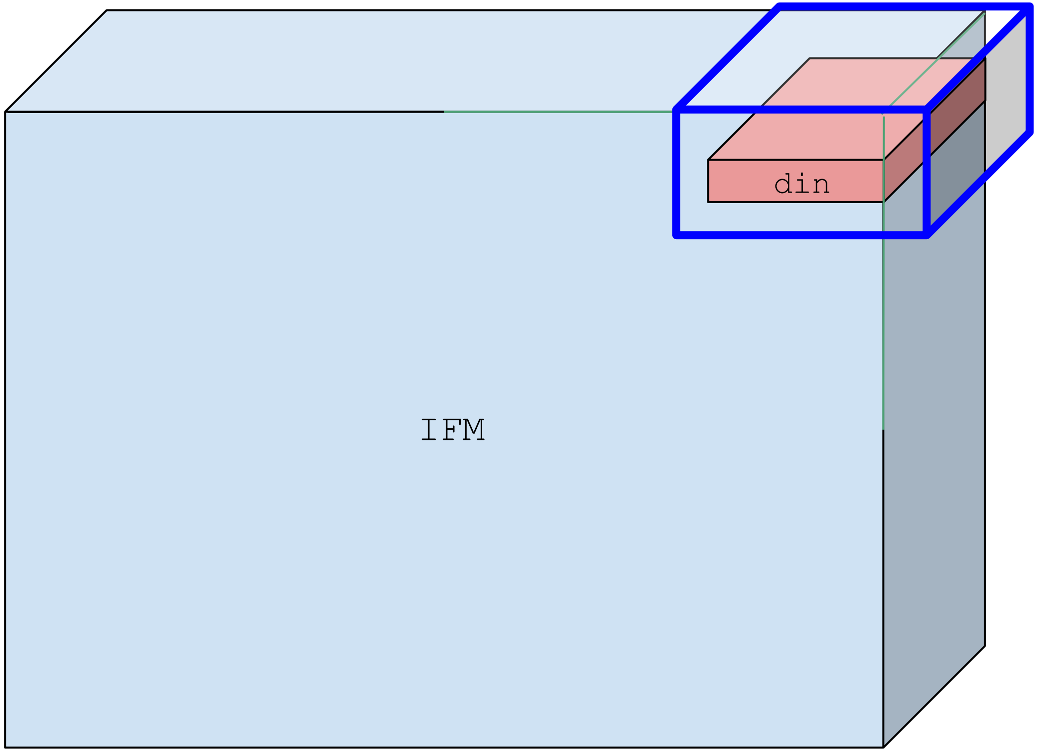


Layer 2 Convolution Operations

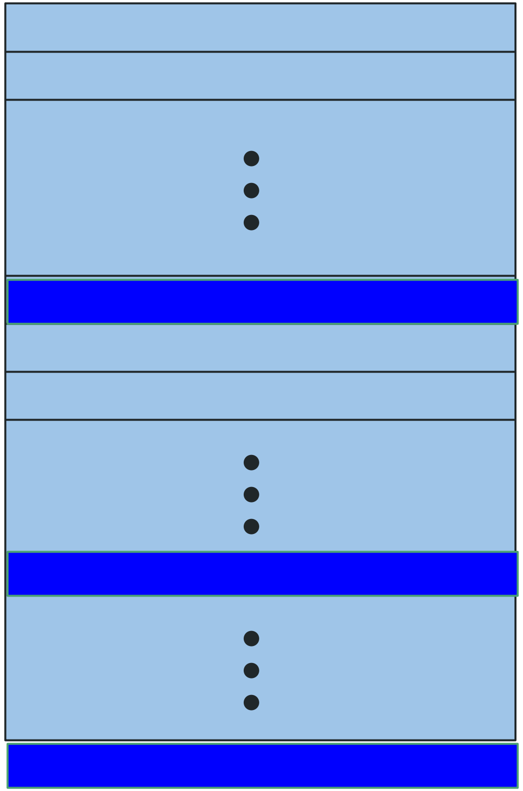




Layer 2 Convolution Operations



Layer 2 line buffer



Layer 3 Convolution Operations

input : 128 x 128 x 4

output : 128 x 128 x 32

weights : 1 x 1 x 4 x 32

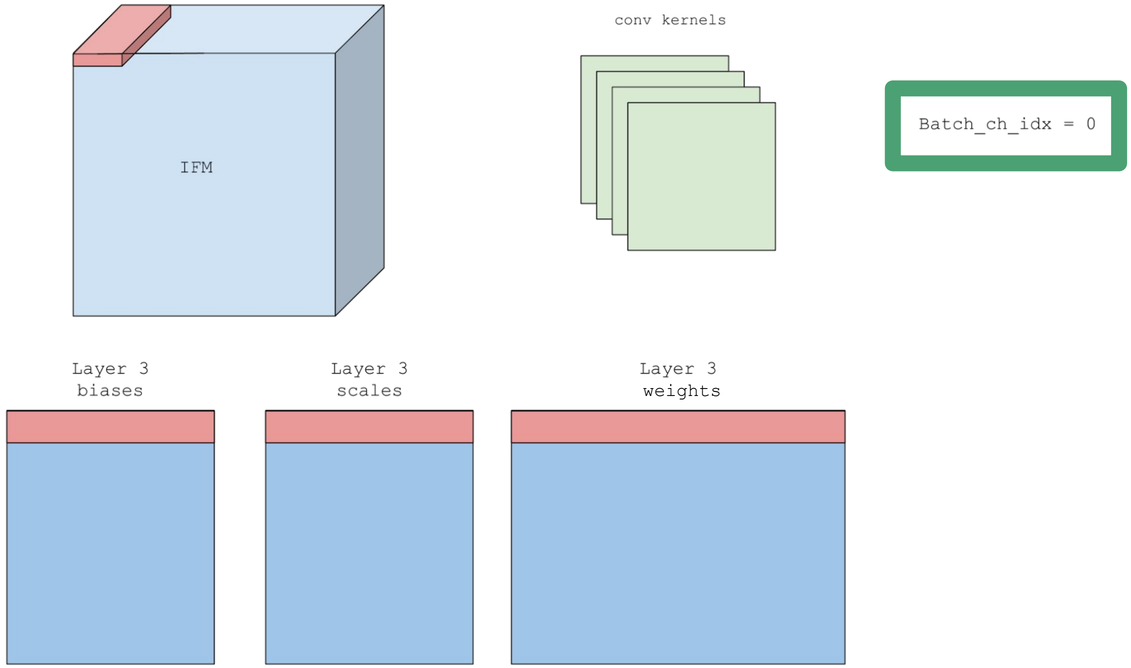
There are 32 filters, so bias and scale must be changed accordingly (in layers 1 and 2, they were static)

batch_ch_idx same as in layer 1, but used to index bias/scales

Fully pipelined accumulate (1 output per clk period)

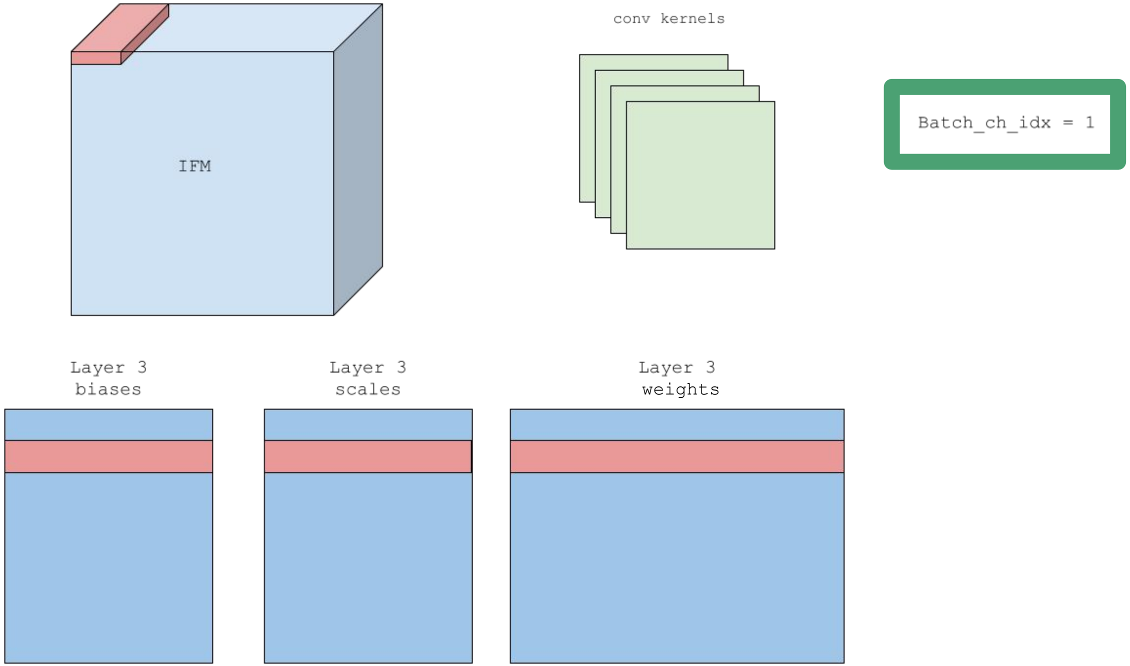
Layer 3 Convolution Operations

Red block = input to kernel



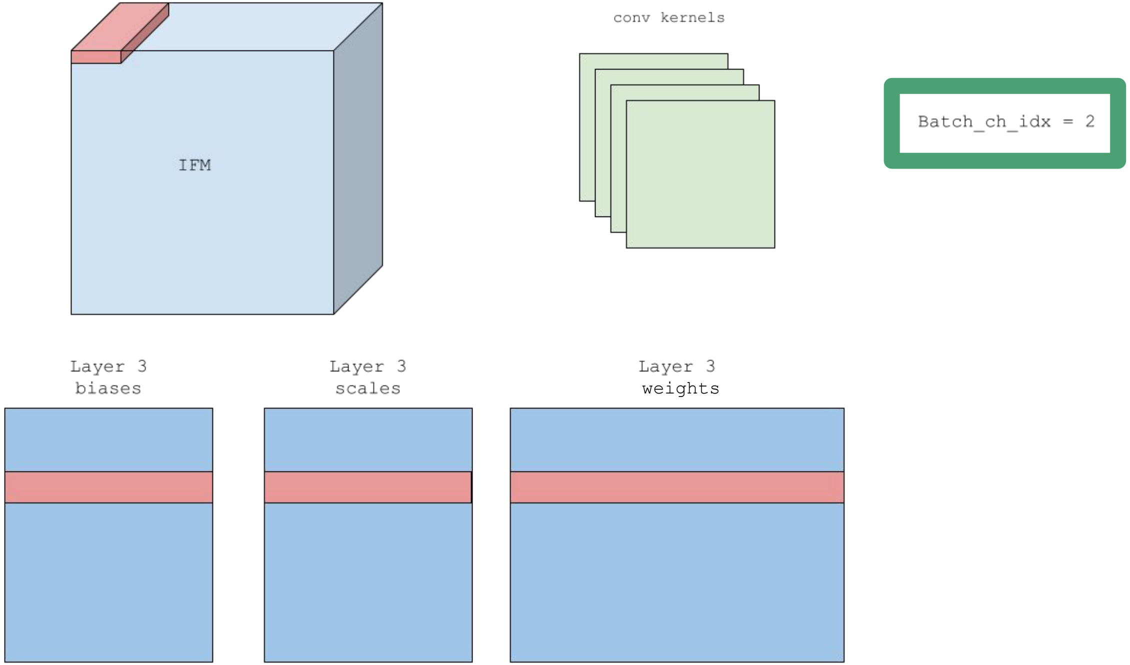
Layer 3 Convolution Operations

Red block = input to kernel



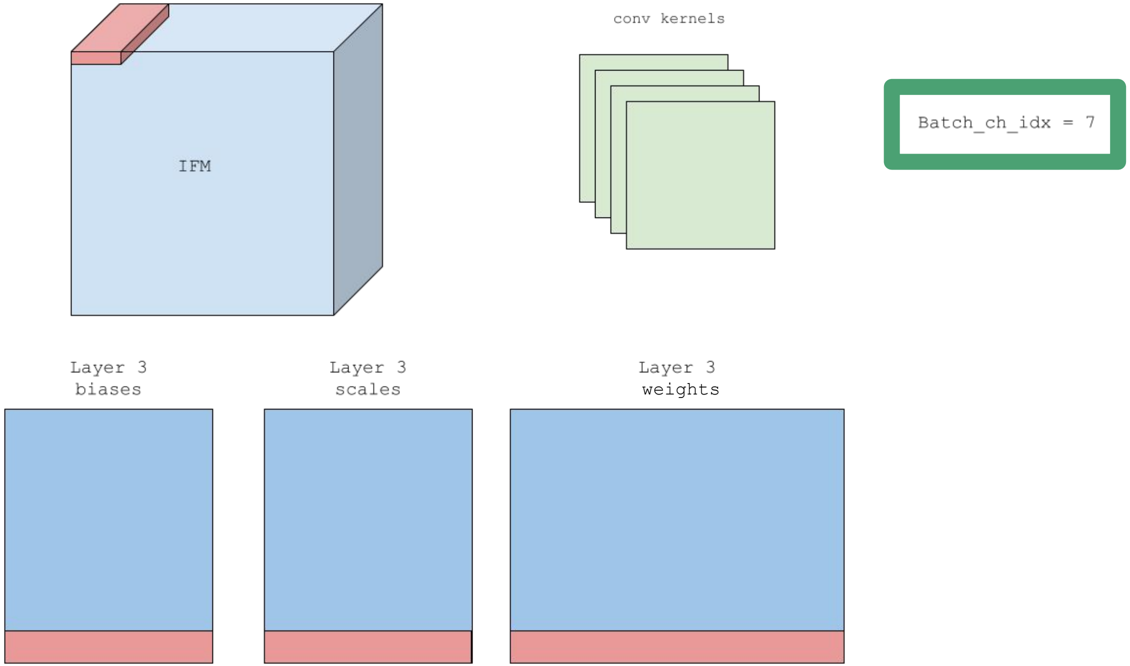
Layer 3 Convolution Operations

Red block = input to kernel



Layer 3 Convolution Operations

Red block = input to kernel



IFM

conv kernels

Batch_ch_idx = 7

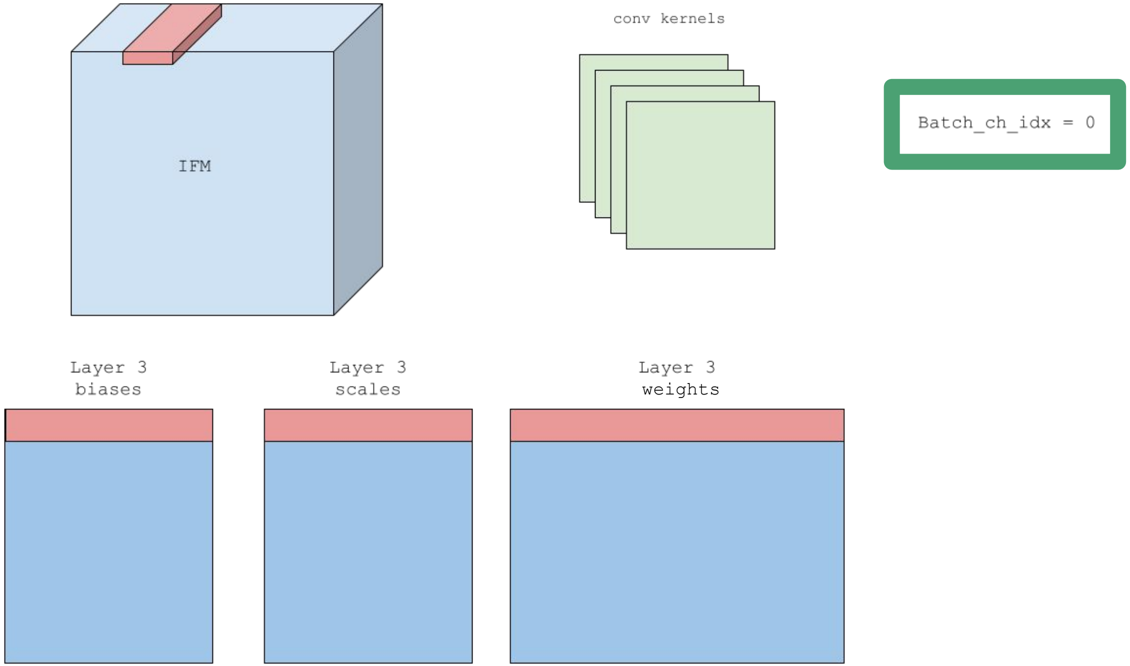
Layer 3
biases

Layer 3
scales

Layer 3
weights

Layer 3 Convolution Operations

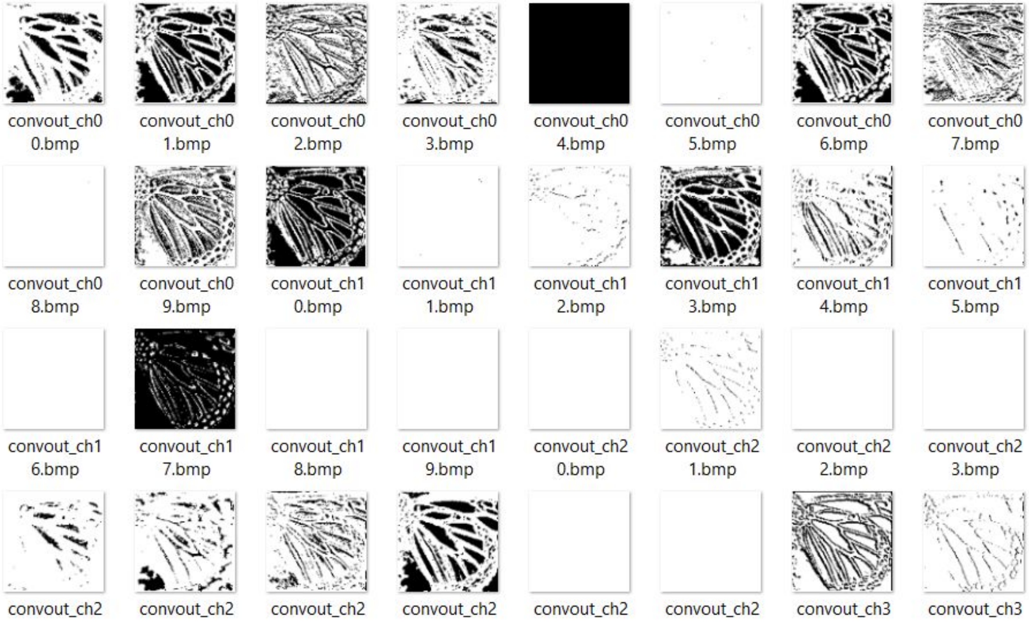
Red block = input to kernel



Increment col/row and repeat!

Masking

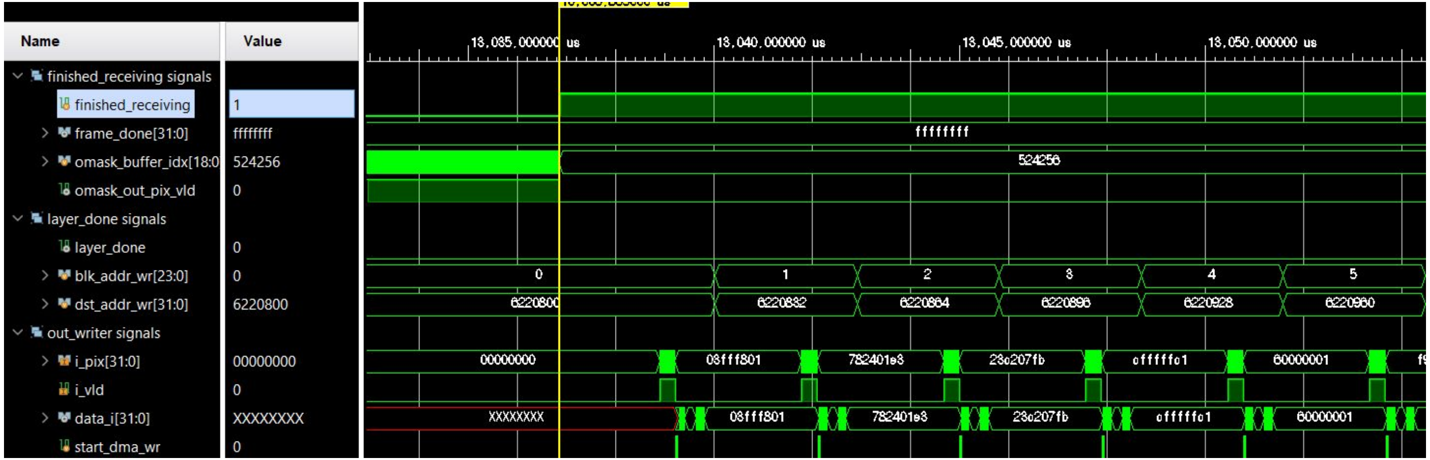
From the third layer, all elements less than or equal to 0 are set to 0, and all elements other than that are set to 255.



Outputting and Writing to Memory

Write mask to memory once all calculations are done

To write to memory, requires bursts of data to be sent at a time

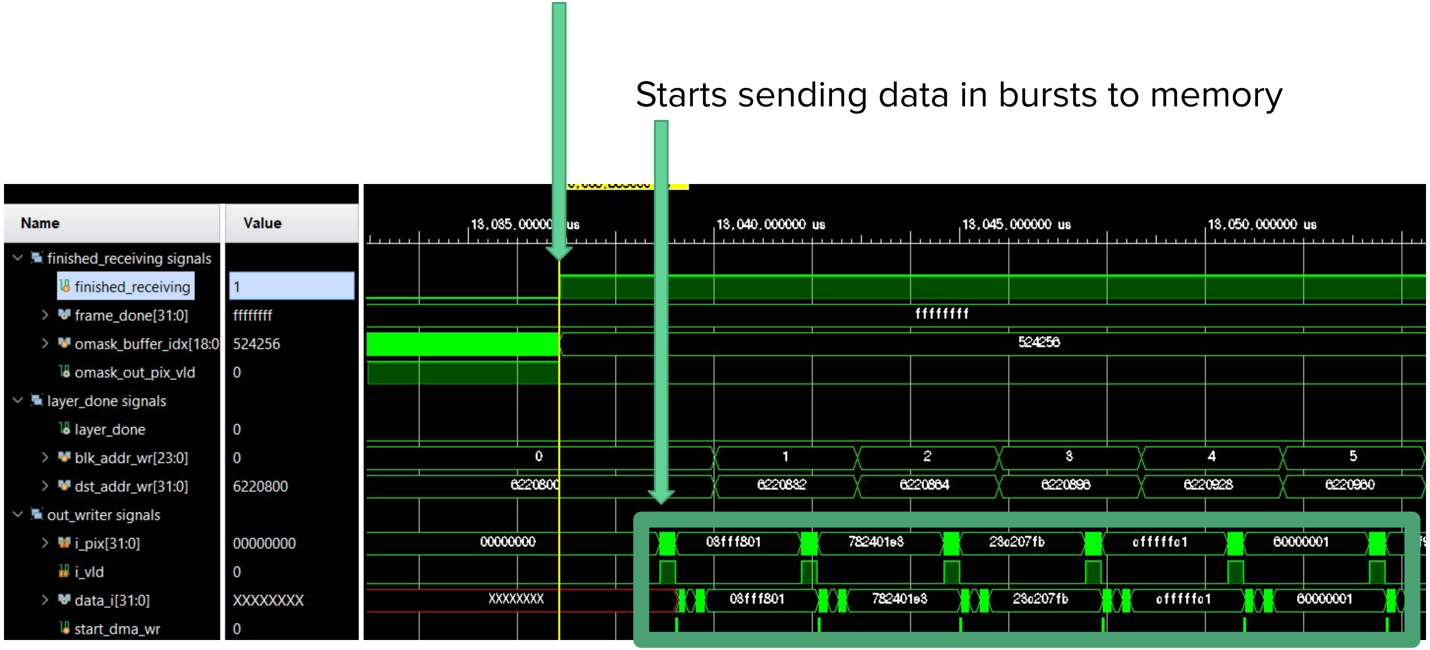


Name	Value
finished_receiving signals	
finished_receiving	1
frame_done[31:0]	ffffffff
omask_buffer_idx[18:0	524256
omask_out_pix_vld	0
layer_done signals	
layer_done	0
blk_addr_wr[23:0]	0
dst_addr_wr[31:0]	6220800
out_writer signals	
i_pix[31:0]	00000000
i_vld	0
data_i[31:0]	XXXXXXXX
start_dma_wr	0

Outputting and Writing to Memory

Finished receiving
signal asserted

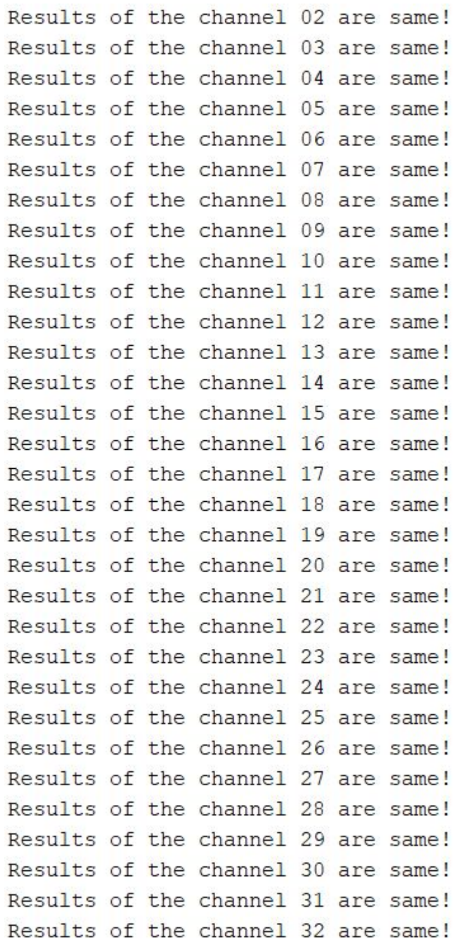
Starts sending data in bursts to memory



Name	Value
finished_receiving signals	
finished_receiving	1
frame_done[31:0]	ffffffff
omask_buffer_idx[18:0	524256
omask_out_pix_vld	0
layer_done signals	
layer_done	0
blk_addr_wr[23:0]	0
dst_addr_wr[31:0]	6220800
out_writer signals	
i_pix[31:0]	00000000
i_vld	0
data_i[31:0]	XXXXXXXX
start_dma_wr	0

Checking Results

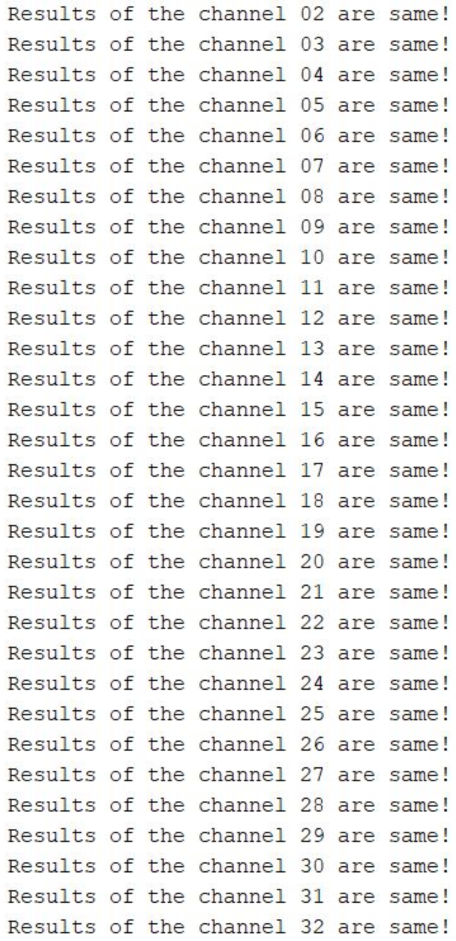
The outputted OFM’s from the last layer are saved as a bmp file and checked against the outputted masks from the matlab SW



```
Results of the channel 02 are same!
Results of the channel 03 are same!
Results of the channel 04 are same!
Results of the channel 05 are same!
Results of the channel 06 are same!
Results of the channel 07 are same!
Results of the channel 08 are same!
Results of the channel 09 are same!
Results of the channel 10 are same!
Results of the channel 11 are same!
Results of the channel 12 are same!
Results of the channel 13 are same!
Results of the channel 14 are same!
Results of the channel 15 are same!
Results of the channel 16 are same!
Results of the channel 17 are same!
Results of the channel 18 are same!
Results of the channel 19 are same!
Results of the channel 20 are same!
Results of the channel 21 are same!
Results of the channel 22 are same!
Results of the channel 23 are same!
Results of the channel 24 are same!
Results of the channel 25 are same!
Results of the channel 26 are same!
Results of the channel 27 are same!
Results of the channel 28 are same!
Results of the channel 29 are same!
Results of the channel 30 are same!
Results of the channel 31 are same!
Results of the channel 32 are same!
```

Checking Results

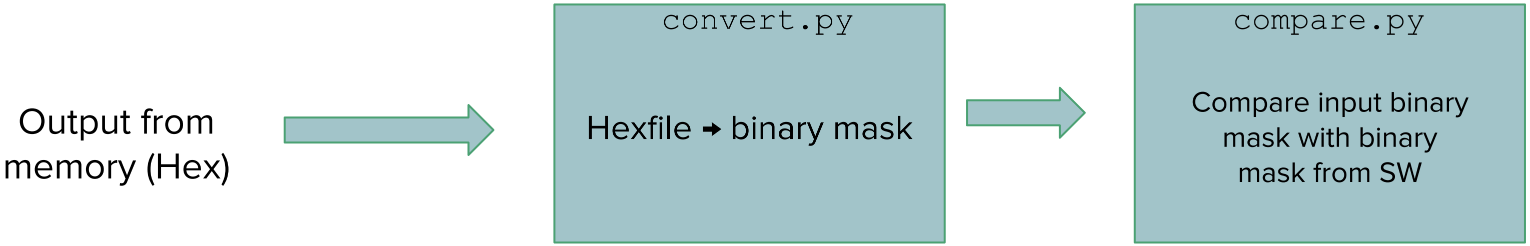
Outputs are verified for all layers (3,4,5, and 6)



```
Results of the channel 02 are same!
Results of the channel 03 are same!
Results of the channel 04 are same!
Results of the channel 05 are same!
Results of the channel 06 are same!
Results of the channel 07 are same!
Results of the channel 08 are same!
Results of the channel 09 are same!
Results of the channel 10 are same!
Results of the channel 11 are same!
Results of the channel 12 are same!
Results of the channel 13 are same!
Results of the channel 14 are same!
Results of the channel 15 are same!
Results of the channel 16 are same!
Results of the channel 17 are same!
Results of the channel 18 are same!
Results of the channel 19 are same!
Results of the channel 20 are same!
Results of the channel 21 are same!
Results of the channel 22 are same!
Results of the channel 23 are same!
Results of the channel 24 are same!
Results of the channel 25 are same!
Results of the channel 26 are same!
Results of the channel 27 are same!
Results of the channel 28 are same!
Results of the channel 29 are same!
Results of the channel 30 are same!
Results of the channel 31 are same!
Results of the channel 32 are same!
```

Checking results (extra, just in case)

Output converted to binary, and compared with SW output



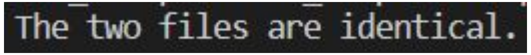
Output from memory (Hex)

`convert.py`

Hexfile → binary mask

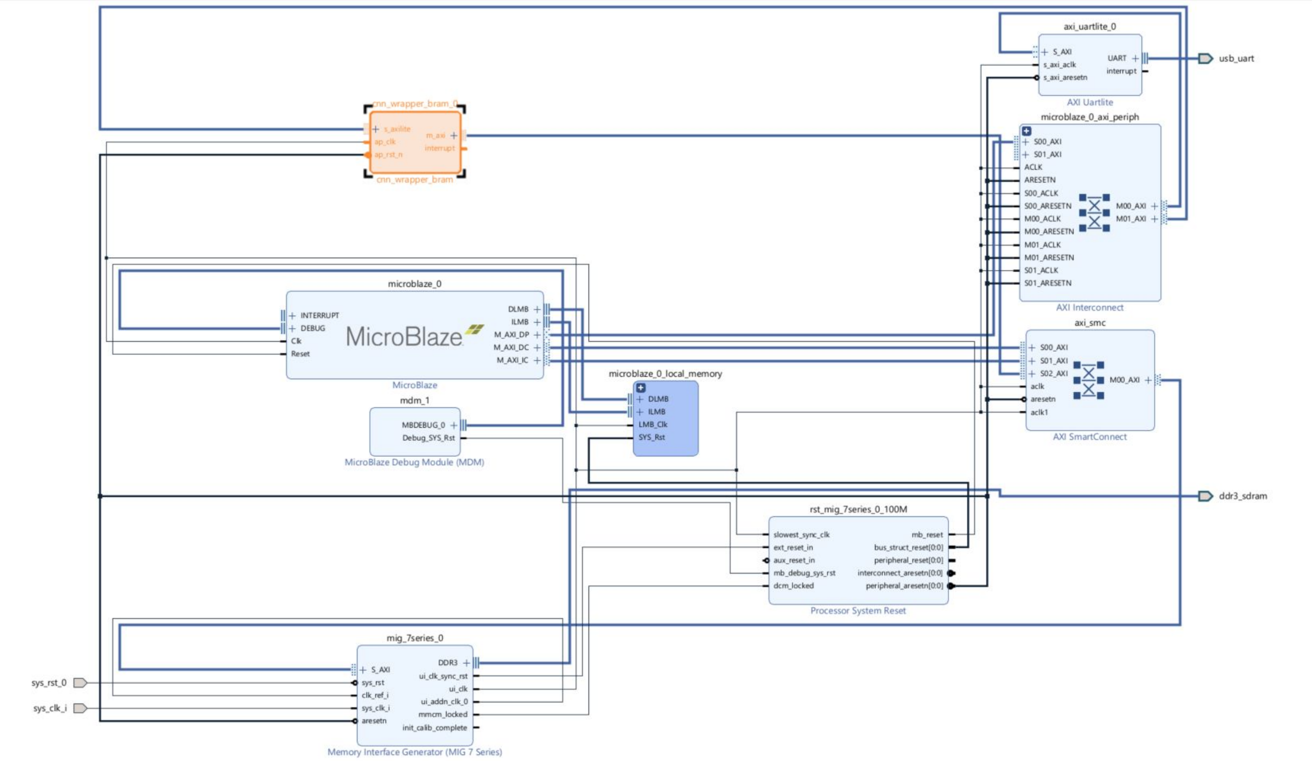
`compare.py`

Compare input binary mask with binary mask from SW



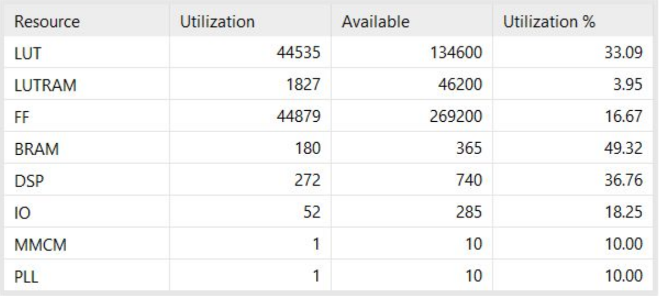
```
The two files are identical.
```

Synthesized and Implemented Design

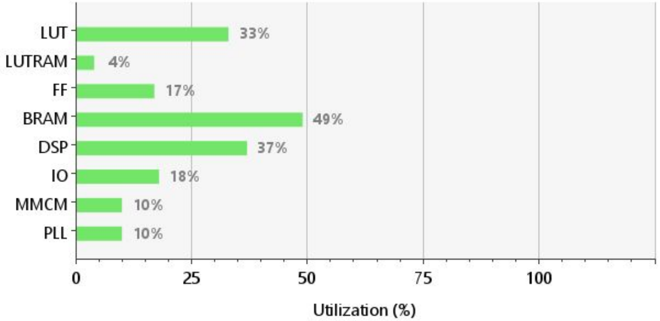


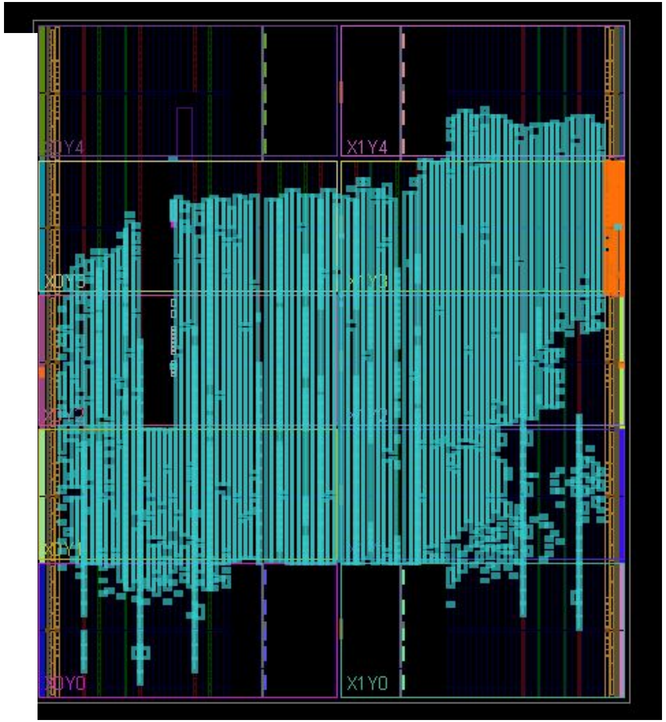
Synthesized and Implemented Design

Nexys Video
FPGA board



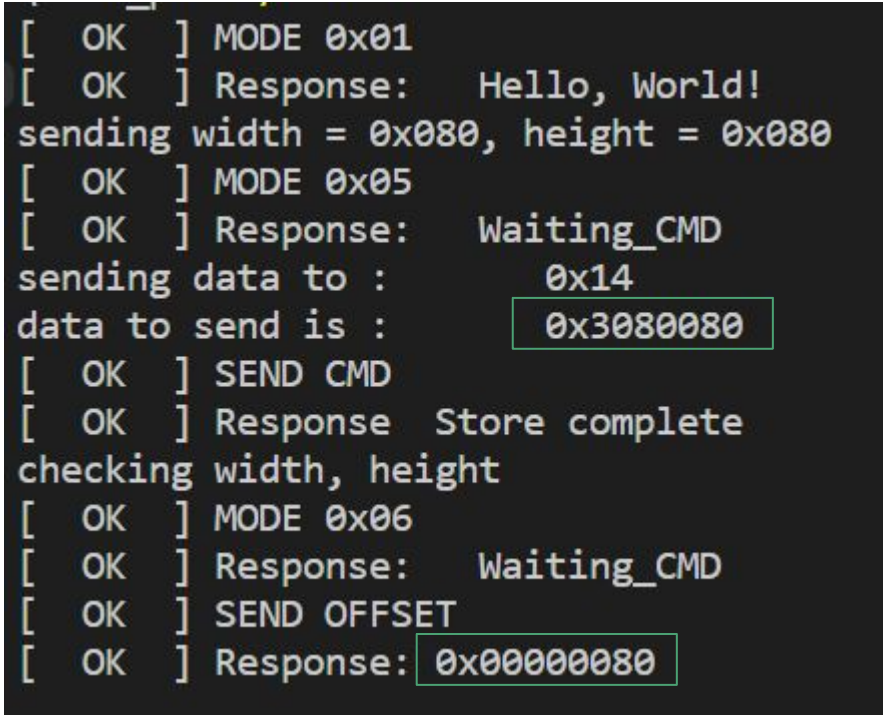
Resource	Utilization	Available	Utilization %
LUT	44535	134600	33.09
LUTRAM	1827	46200	3.95
FF	44879	269200	16.67
BRAM	180	365	49.32
DSP	272	740	36.76
IO	52	285	18.25
MMCM	1	10	10.00
PLL	1	10	10.00





TODO

Get it running on an FPGA...

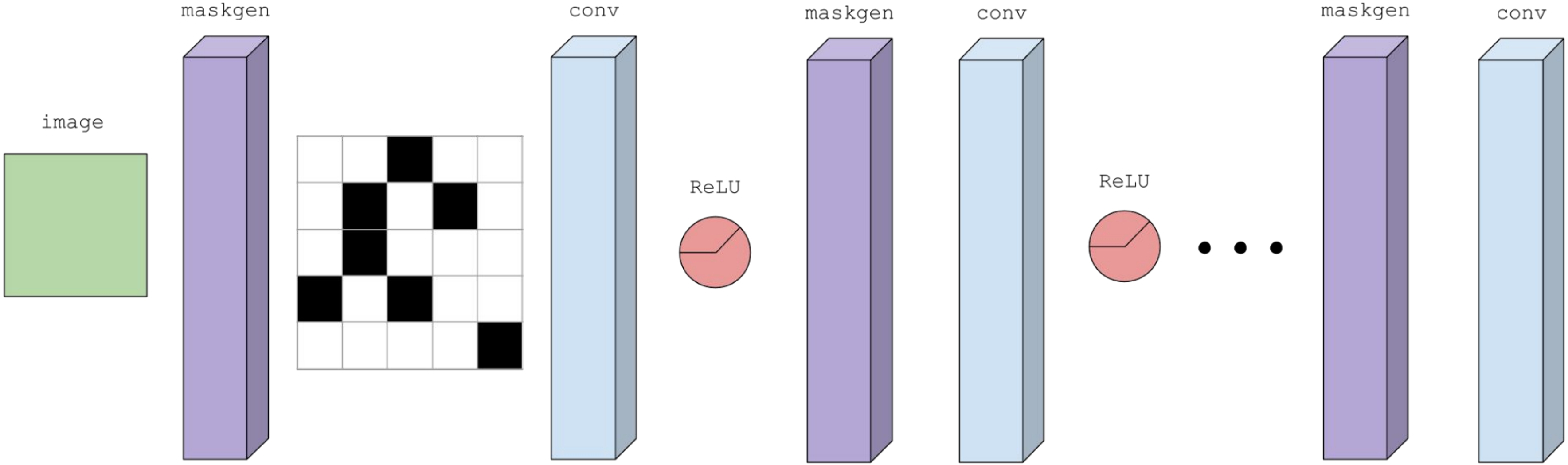


```
[   OK   ] MODE 0x01
[   OK   ] Response:   Hello, World!
sending width = 0x080, height = 0x080
[   OK   ] MODE 0x05
[   OK   ] Response:   Waiting_CMD
sending data to :          0x14
data to send is :          0x3080080
[   OK   ] SEND CMD
[   OK   ] Response  Store complete
checking width, height
[   OK   ] MODE 0x06
[   OK   ] Response:   Waiting_CMD
[   OK   ] SEND OFFSET
[   OK   ] Response: 0x00000080
```

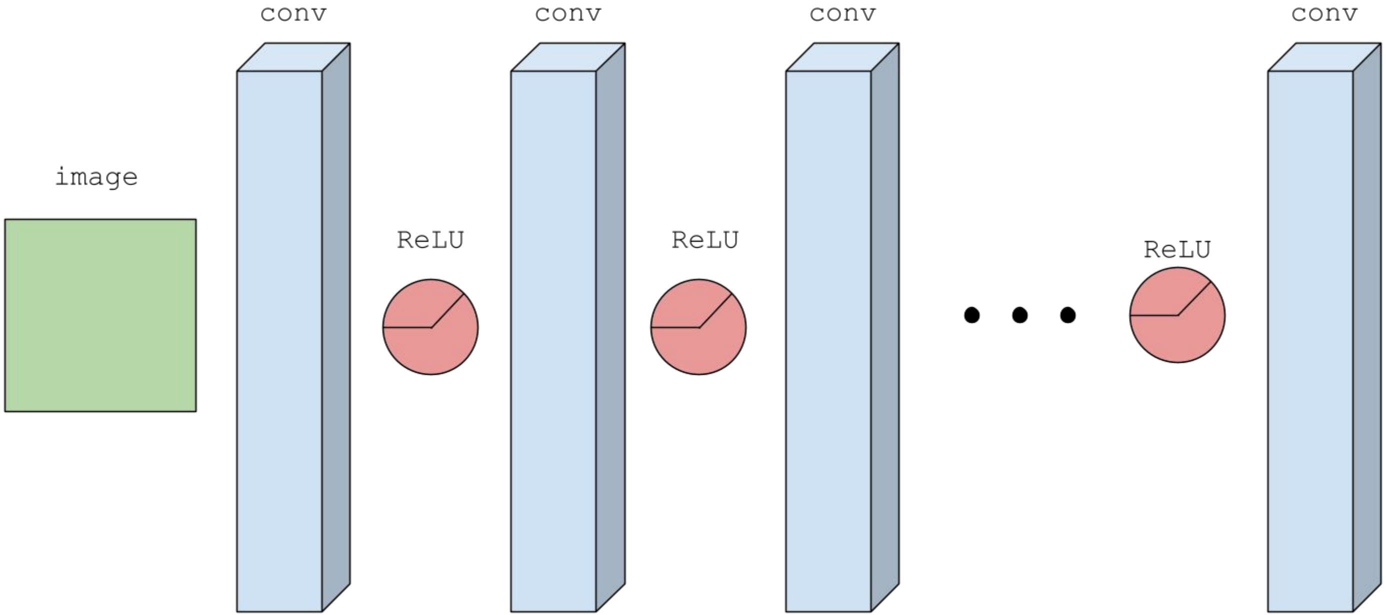
THANK
YOU

Extra slides

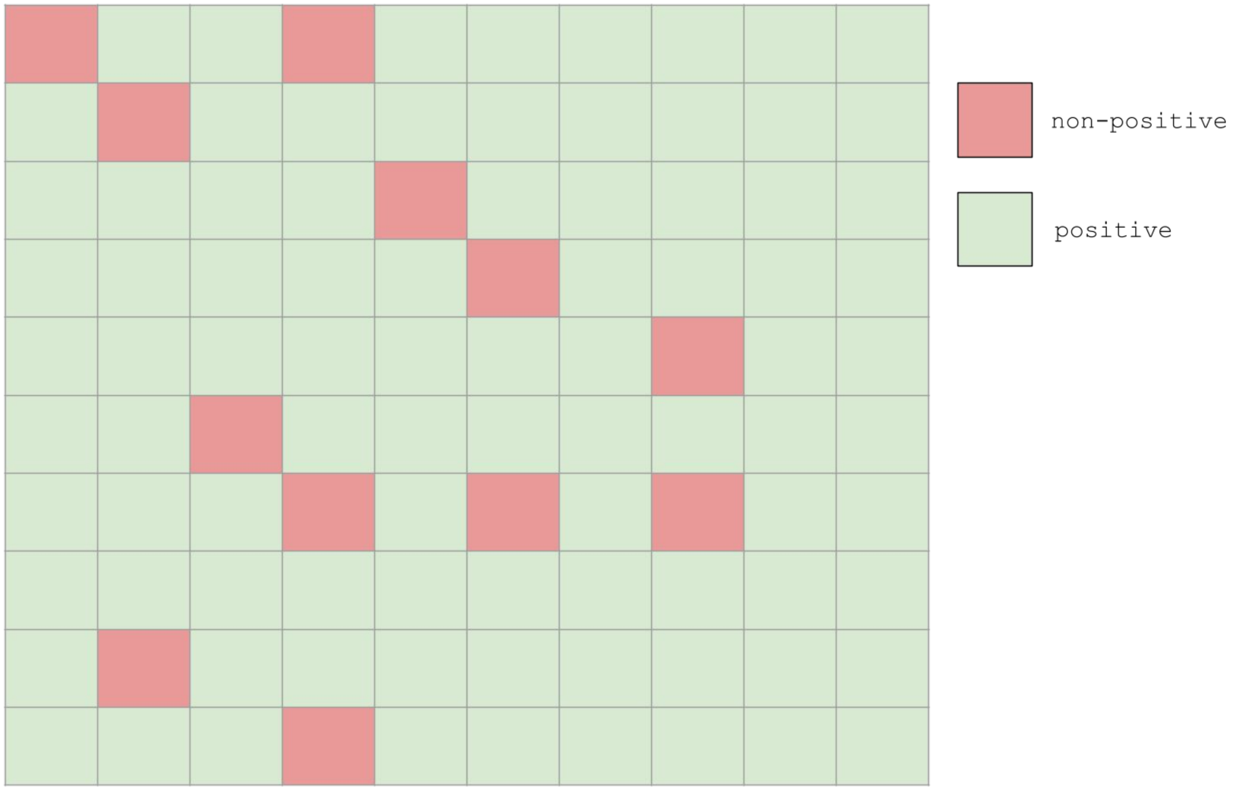
Zero Skipping via Mask Generation - The Big Picture



Why Zero Skipping?



After Conv Layer



🟥	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩
🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩
🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟥	🟩	🟥	🟩	🟥	🟩	🟩
🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟩	🟩

🟥 non-positive

🟩 positive

After ReLU

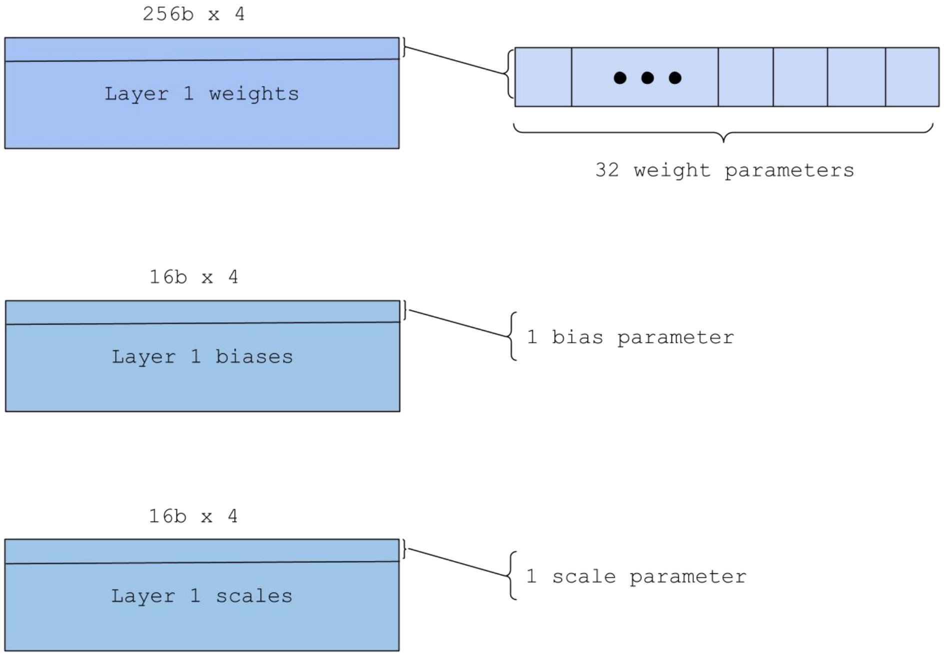
0	🟩	🟩	0	🟩	🟩	🟩	🟩	🟩	🟩
🟩	0	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩	0	🟩	🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩	🟩	0	🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩	🟩	🟩	🟩	0	🟩	🟩
🟩	🟩	0	🟩	🟩	🟩	🟩	🟩	🟩	🟩
🟩	🟩	🟩	0	🟩	0	🟩	0	🟩	🟩
🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
🟩	0	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
🟩	🟩	🟩	0	🟩	🟩	🟩	🟩	🟩	🟩

🟥 non-positive

🟩 positive

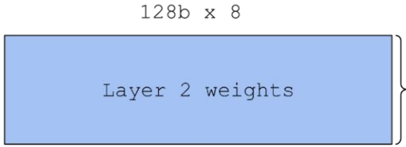
Inputs

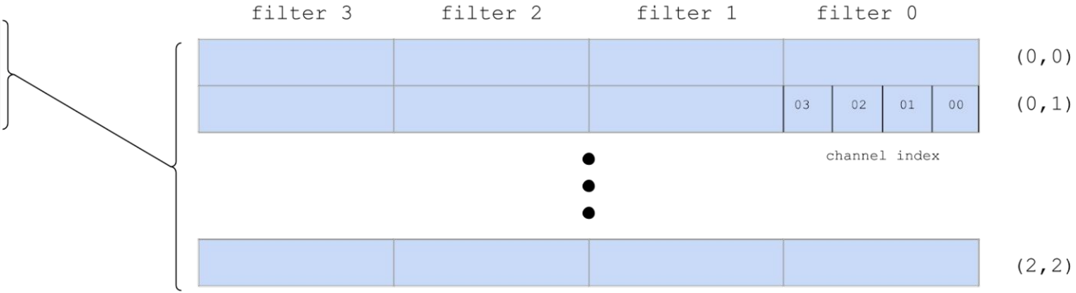
Layer 1 conv weights are 1x1x32x4

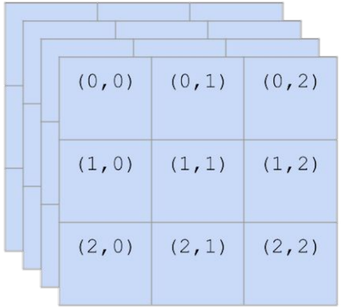


Inputs

Layer 2 conv weights are 3x3x4x4



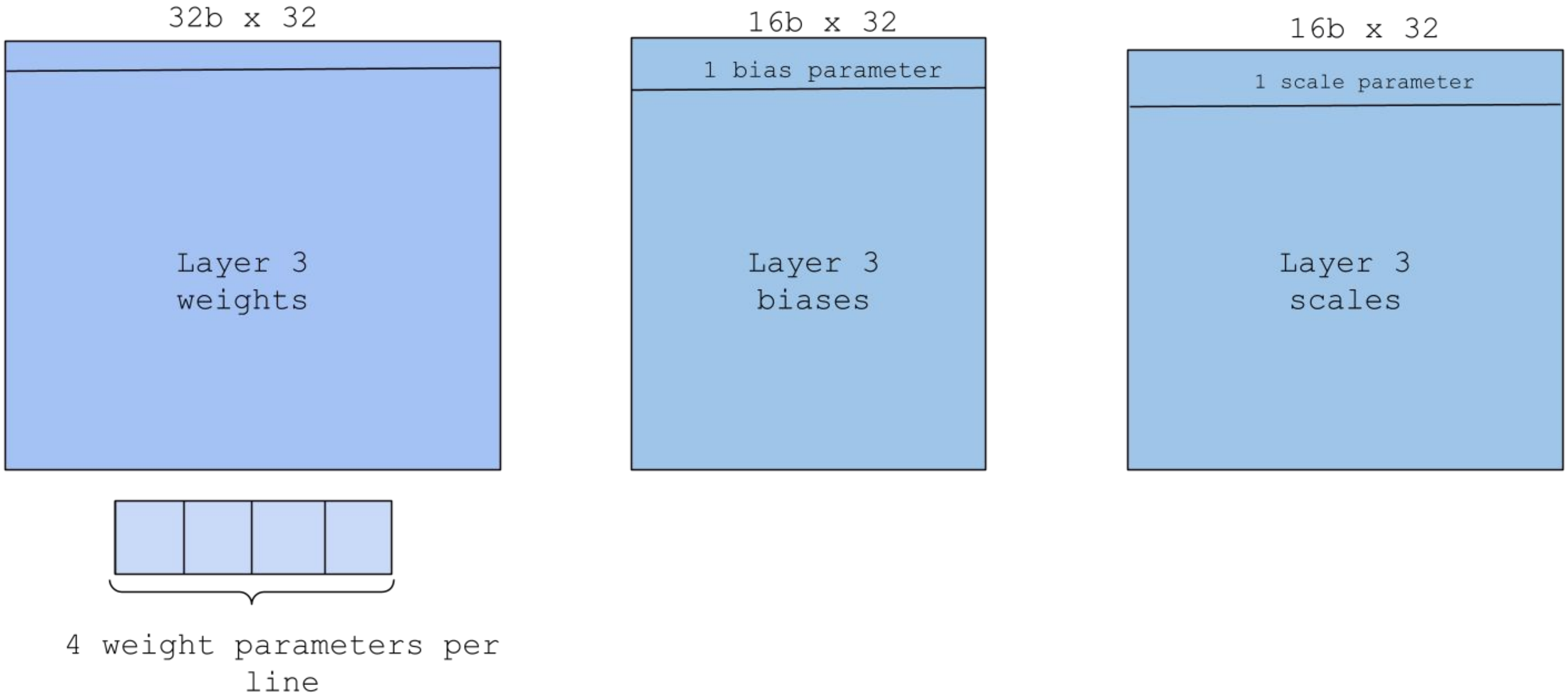




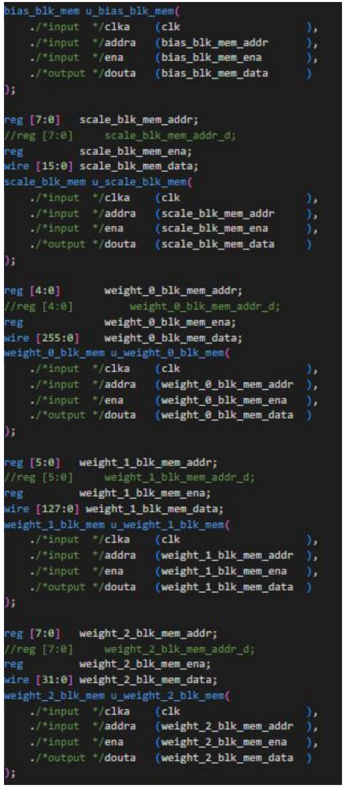
*biases and scales are organized as the same as in layer 1

Inputs

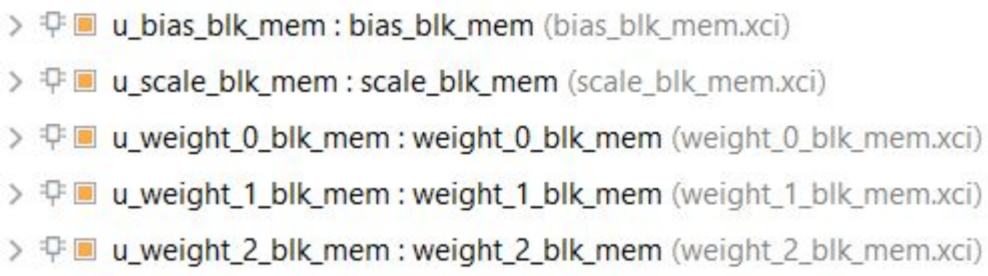
Layer 3 conv weights are 1x1x4x32



Initializing Memory with ROMs



```
bias_blk_mem u_bias_blk_mem(
    ./*input  */clka    (clk                   ),
    ./*input  */addra   (bias_blk_mem_addr     ),
    ./*input  */ena     (bias_blk_mem_ena      ),
    ./*output */douta   (bias_blk_mem_data     )
);

reg [7:0]   scale_blk_mem_addr;
//reg [7:0]     scale_blk_mem_addr_d;
reg         scale_blk_mem_ena;
wire [15:0] scale_blk_mem_data;
scale_blk_mem u_scale_blk_mem(
    ./*input  */clka    (clk                   ),
    ./*input  */addra   (scale_blk_mem_addr    ),
    ./*input  */ena     (scale_blk_mem_ena     ),
    ./*output */douta   (scale_blk_mem_data    )
);

reg [4:0]       weight_0_blk_mem_addr;
//reg [4:0]         weight_0_blk_mem_addr_d;
reg             weight_0_blk_mem_ena;
wire [255:0]    weight_0_blk_mem_data;
weight_0_blk_mem u_weight_0_blk_mem(
    ./*input  */clka    (clk                   ),
    ./*input  */addra   (weight_0_blk_mem_addr ),
    ./*input  */ena     (weight_0_blk_mem_ena  ),
    ./*output */douta   (weight_0_blk_mem_data )
);

reg [5:0]   weight_1_blk_mem_addr;
//reg [5:0]     weight_1_blk_mem_addr_d;
reg         weight_1_blk_mem_ena;
wire [127:0] weight_1_blk_mem_data;
weight_1_blk_mem u_weight_1_blk_mem(
    ./*input  */clka    (clk                   ),
    ./*input  */addra   (weight_1_blk_mem_addr ),
    ./*input  */ena     (weight_1_blk_mem_ena  ),
    ./*output */douta   (weight_1_blk_mem_data )
);

reg [7:0]   weight_2_blk_mem_addr;
//reg [7:0]     weight_2_blk_mem_addr_d;
reg         weight_2_blk_mem_ena;
wire [31:0] weight_2_blk_mem_data;
weight_2_blk_mem u_weight_2_blk_mem(
    ./*input  */clka    (clk                   ),
    ./*input  */addra   (weight_2_blk_mem_addr ),
    ./*input  */ena     (weight_2_blk_mem_ena  ),
    ./*output */douta   (weight_2_blk_mem_data )
);
```



u_bias_blk_mem : bias_blk_mem (bias_blk_mem.xci)

u_scale_blk_mem : scale_blk_mem (scale_blk_mem.xci)

u_weight_0_blk_mem : weight_0_blk_mem (weight_0_blk_mem.xci)

u_weight_1_blk_mem : weight_1_blk_mem (weight_1_blk_mem.xci)

u_weight_2_blk_mem : weight_2_blk_mem (weight_2_blk_mem.xci)

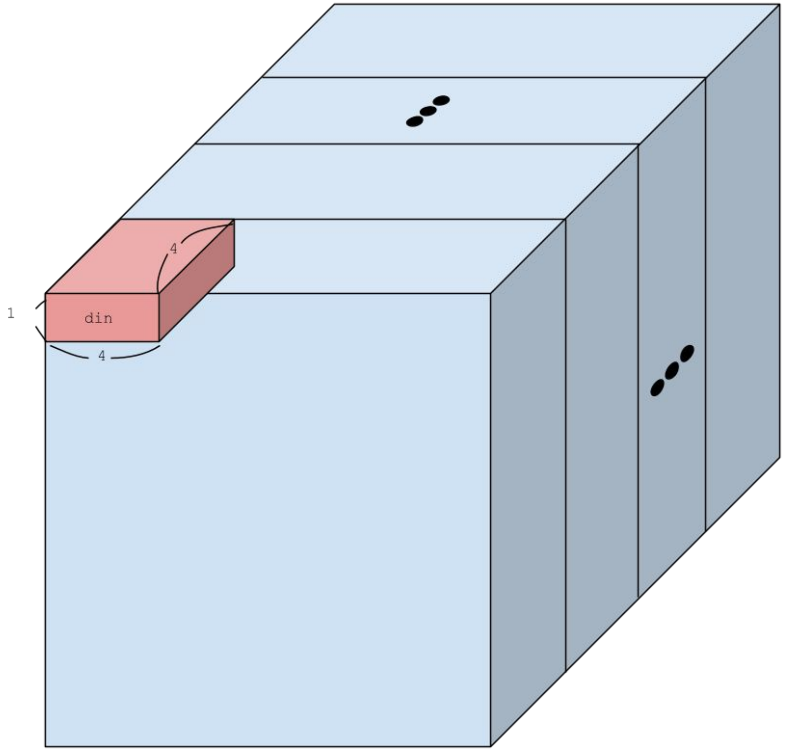
5 ROMs

1 for all scales, biases, weights

3 for all weights for each layer

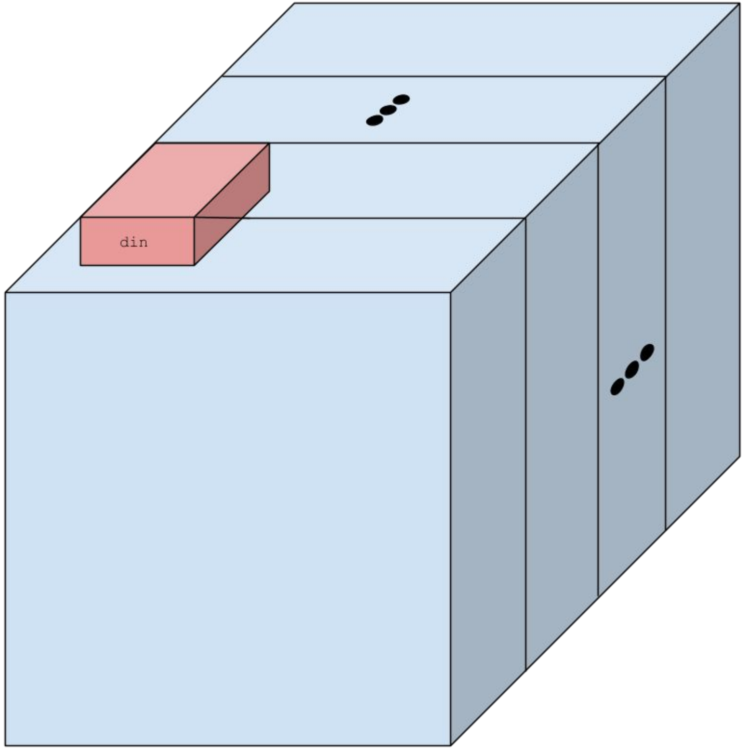
See extra slides for detailed file structure

Layer 1 Convolution Operations



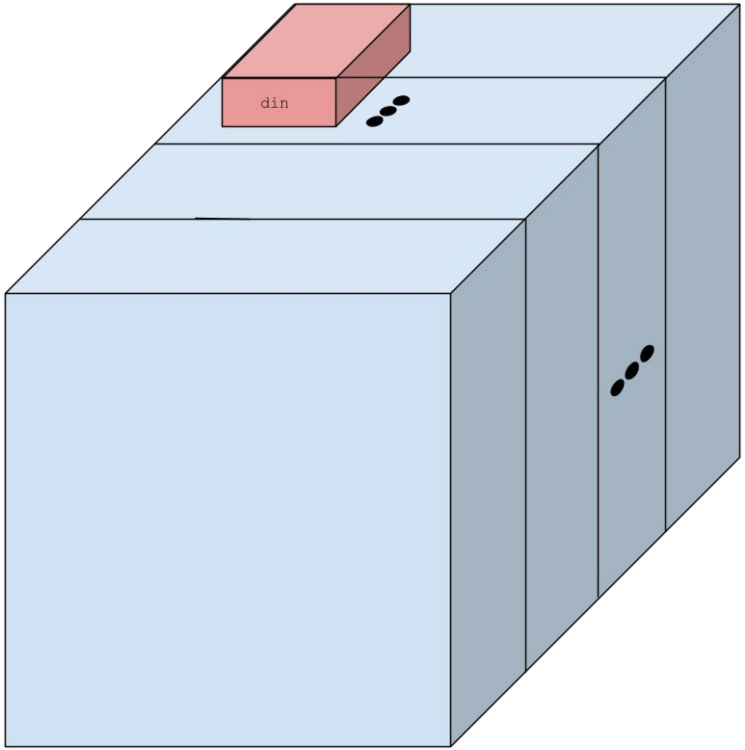
ch_batch_idx = 0

Layer 1 Convolution Operations



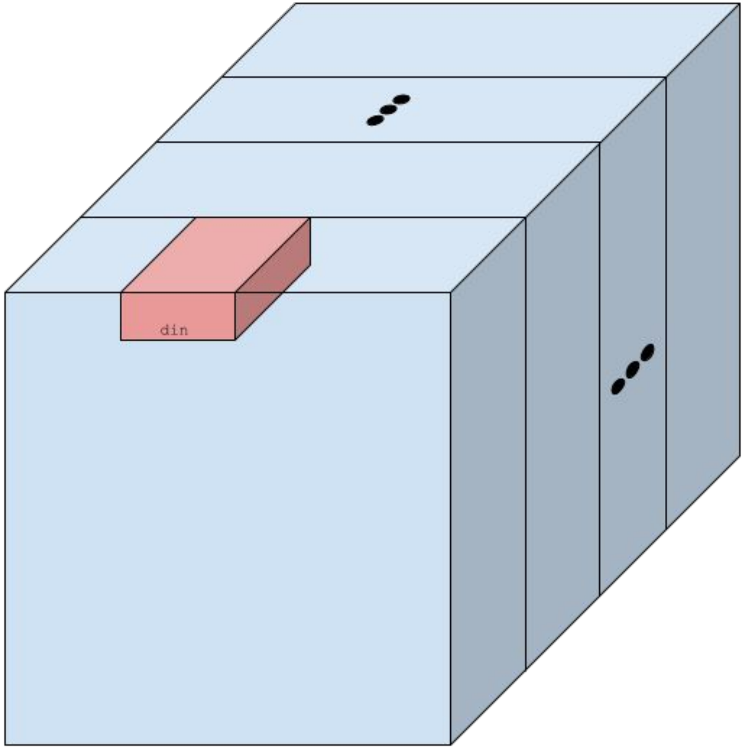
`ch_batch_idx = 1`

Layer 1 Convolution Operations



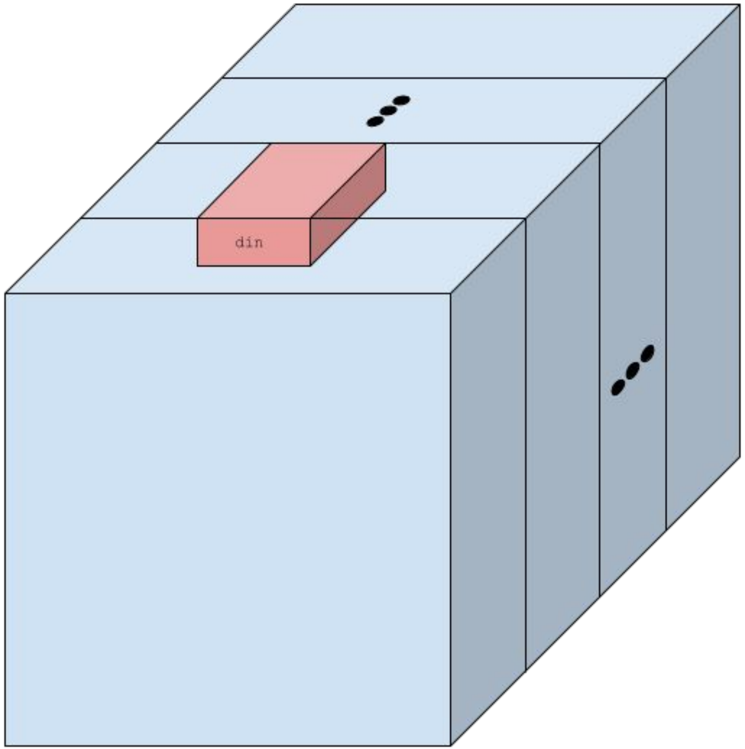
ch_batch_idx = 7

Layer 1 Convolution Operations



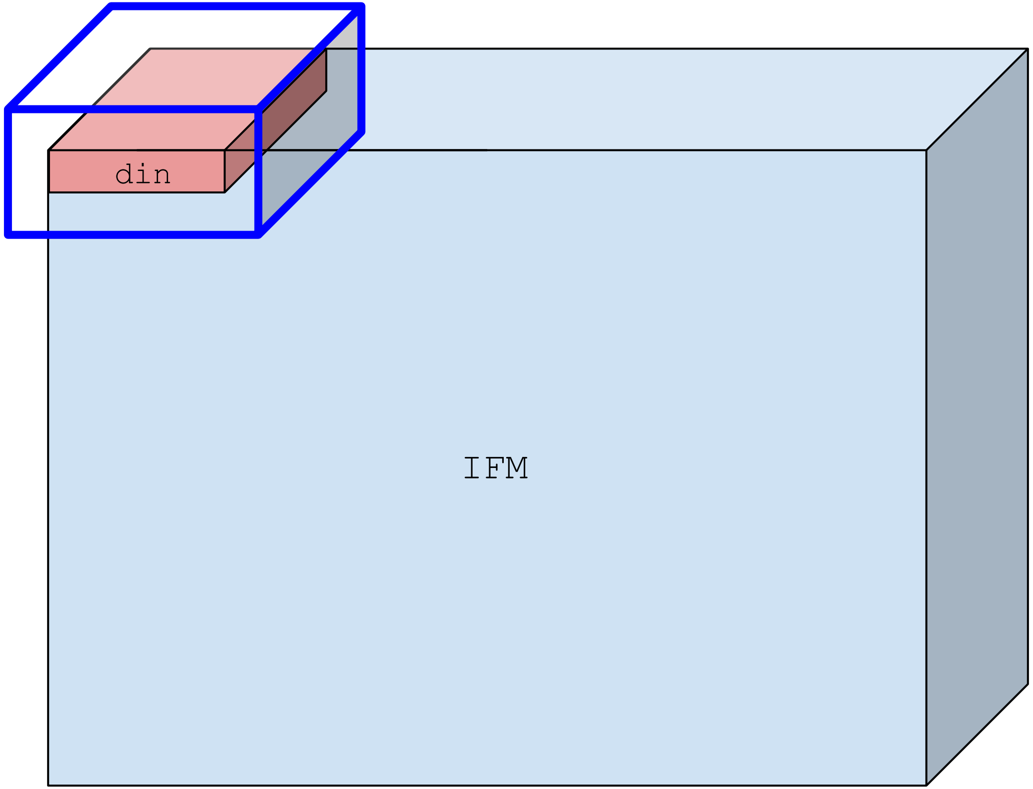
ch_batch_idx = 0

Layer 1 Convolution Operations

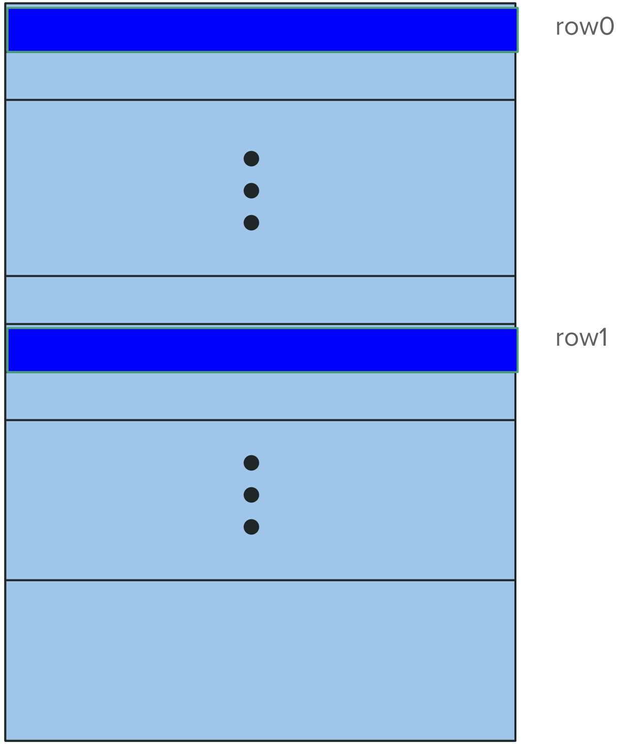


ch_batch_idx = 1

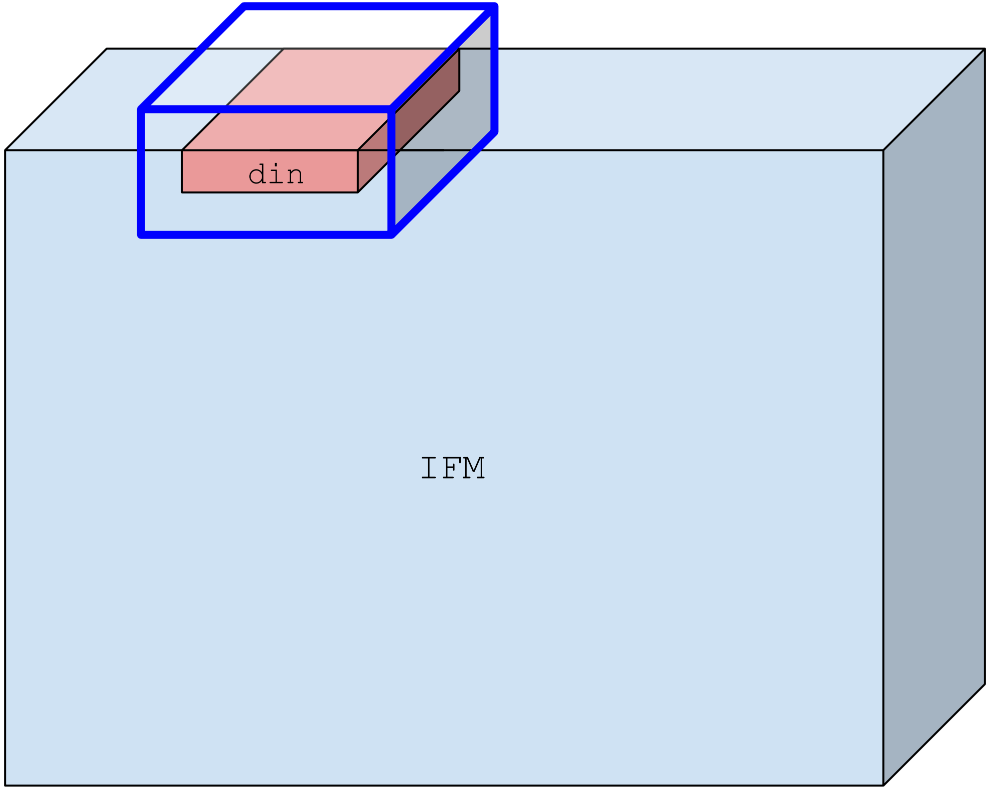
Layer 2 Convolution Operations



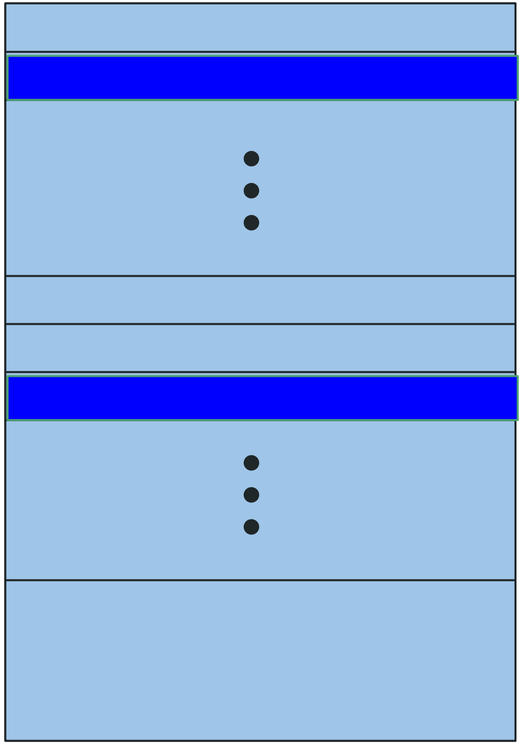
Layer 2 line buffer



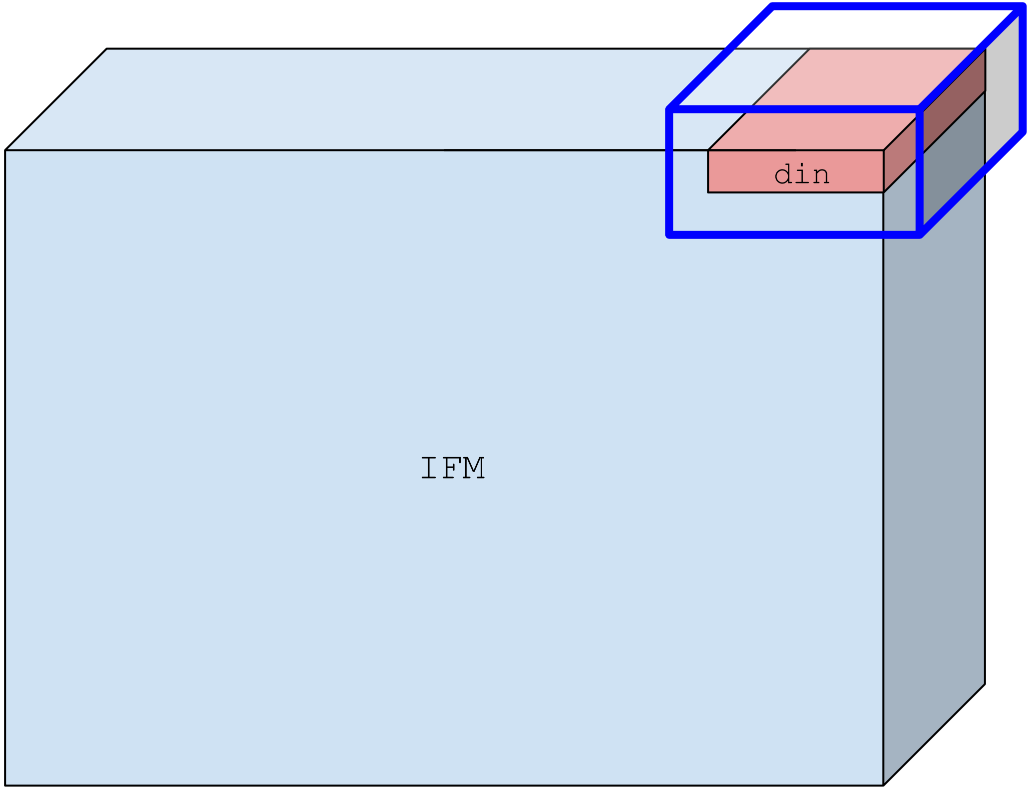
Layer 2 Convolution Operations



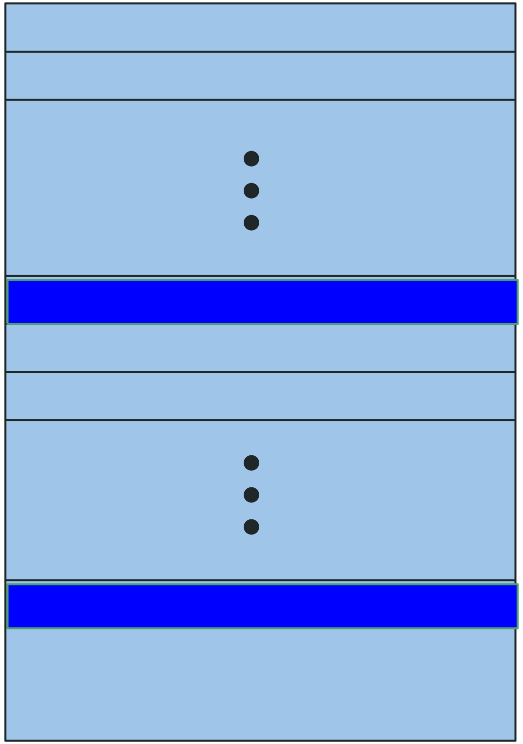
Layer 2 line buffer



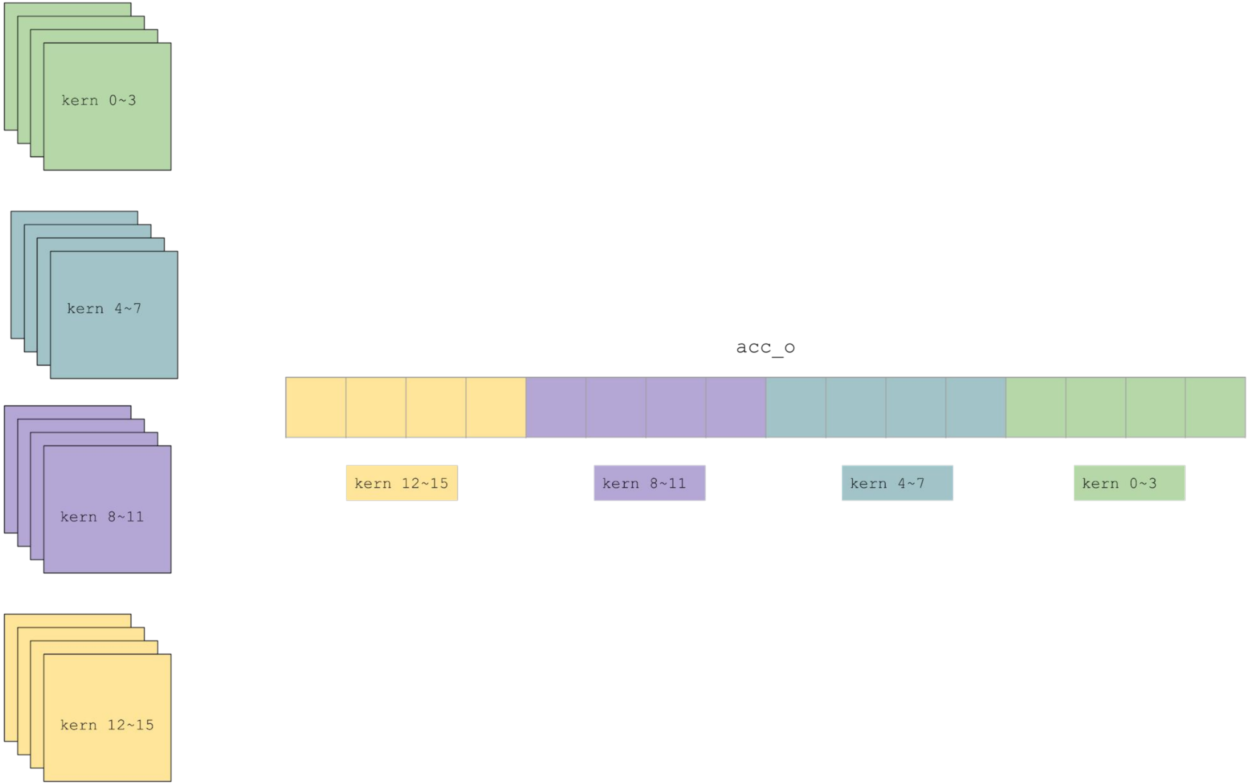
Layer 2 Convolution Operations



Layer 2 line buffer



Kernel outputs structure


kern 0~3
kern 4~7
kern 8~11
kern 12~15
acc_o
kern 12~15
kern 8~11
kern 4~7
kern 0~3