ECE284 Fall 21 W4S1

Low-power VLSI Implementation for Machine Learning

Prof. Mingu Kang

UCSD Computer Engineering

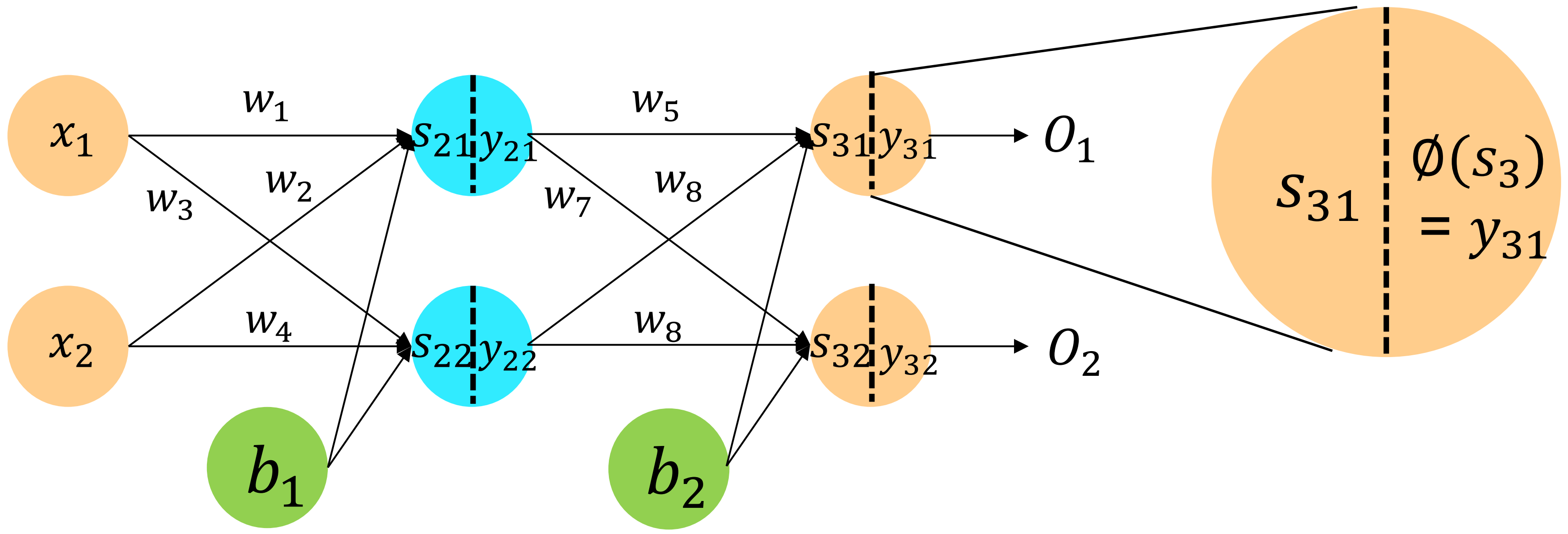
[Example1] Customized Gradient

- Try multiple functions whether auto gradient is generated
- A certain functions (e.g., MyReLU) requires manual definition of gradient
- Define the customized gradient
- .apply() is required to enable the customized static-method based class

[Example2] Gradient for Quantization

- Gradient is defined by using the equation in PACT (W3S2)
- Normalization added for better accuracy (unlike post-training quantization)
- Two gradients (input gradient, alpha gradient) are returned.
- This is because the function had two inputs, input and alpha

Remind: Back Propagation (Example)



in the next page

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial s_{21}} * \frac{\partial s_{21}}{\partial w_1} = \frac{\partial L}{\partial y_{21}} * \frac{\partial y_{21}}{\partial s_{21}} * \frac{\partial s_{21}}{\partial w_1} = \boxed{\frac{\partial L}{\partial y_{21}}} * \cancel{\emptyset'(s_{21})} * x_1$$

$$\boxed{\frac{\partial L}{\partial w_{L,i\to j}} = y_{L,i} * \frac{\partial L}{\partial y_{(L+1),j}}}$$

[Example] Clone Example

[Example3] Customized Cost Function

- Assume that **your hardware/circuitry consumes more power when weight is larger**
- Cost1: consider the accuracy of inference
- Cost2: customized to consider minimizing the weight magnitude
- The ratio (α) between Cost1 and Cost2 (i.e., cost1 + $\alpha \times$cost2) needs to be chosen to balance these two costs
- The cost2 should be differentiable function

Cost1 included in loss (10 Epochs)

```
Epoch: 10          Training Loss1: 0.039878
Epoch: 10          Training Loss2: 701.935702
Epoch: 10          Training Loss: 0.039878
Test accuracy: 99%
```

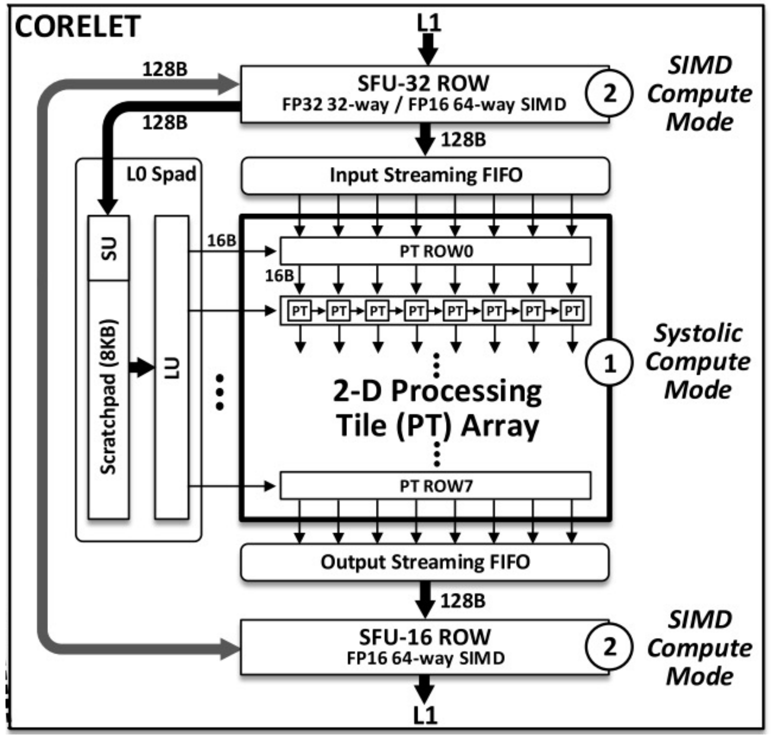
Cost1 + Cost2 included in loss (10 Epochs)

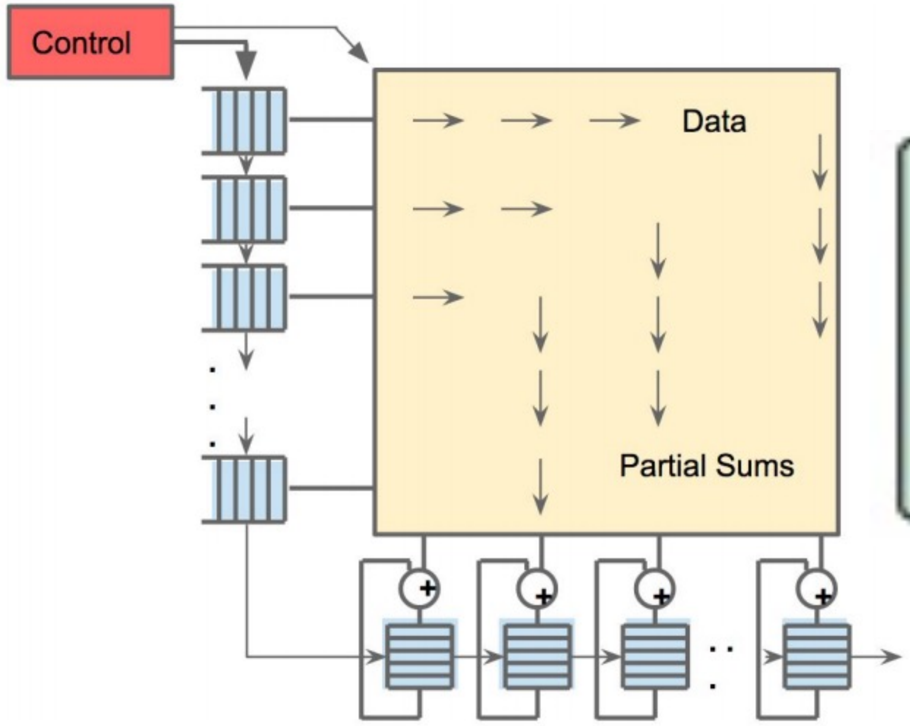
```
Epoch: 10          Training Loss1: 0.245277
Epoch: 10          Training Loss2: 7.226559
Epoch: 10          Training Loss: 0.606605
Test accuracy: 96%
```

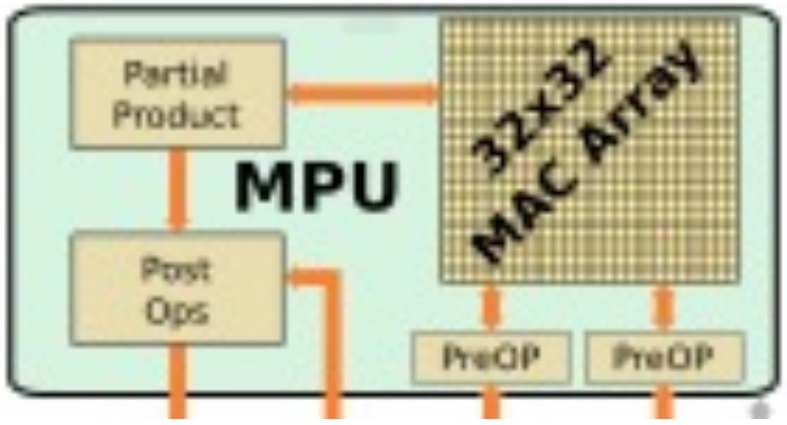
[HW4_prob2] VGG16 Quantization-aware Training

- Quantization-aware training with 4bits for both weight and quantization
- Check the alpha is changing over time
- Weight and alpha are co-optimized
- Check the recovered psum (such as example1 in W3S2) has similar value to the un-quantized original psum

2D Systolic Architecture



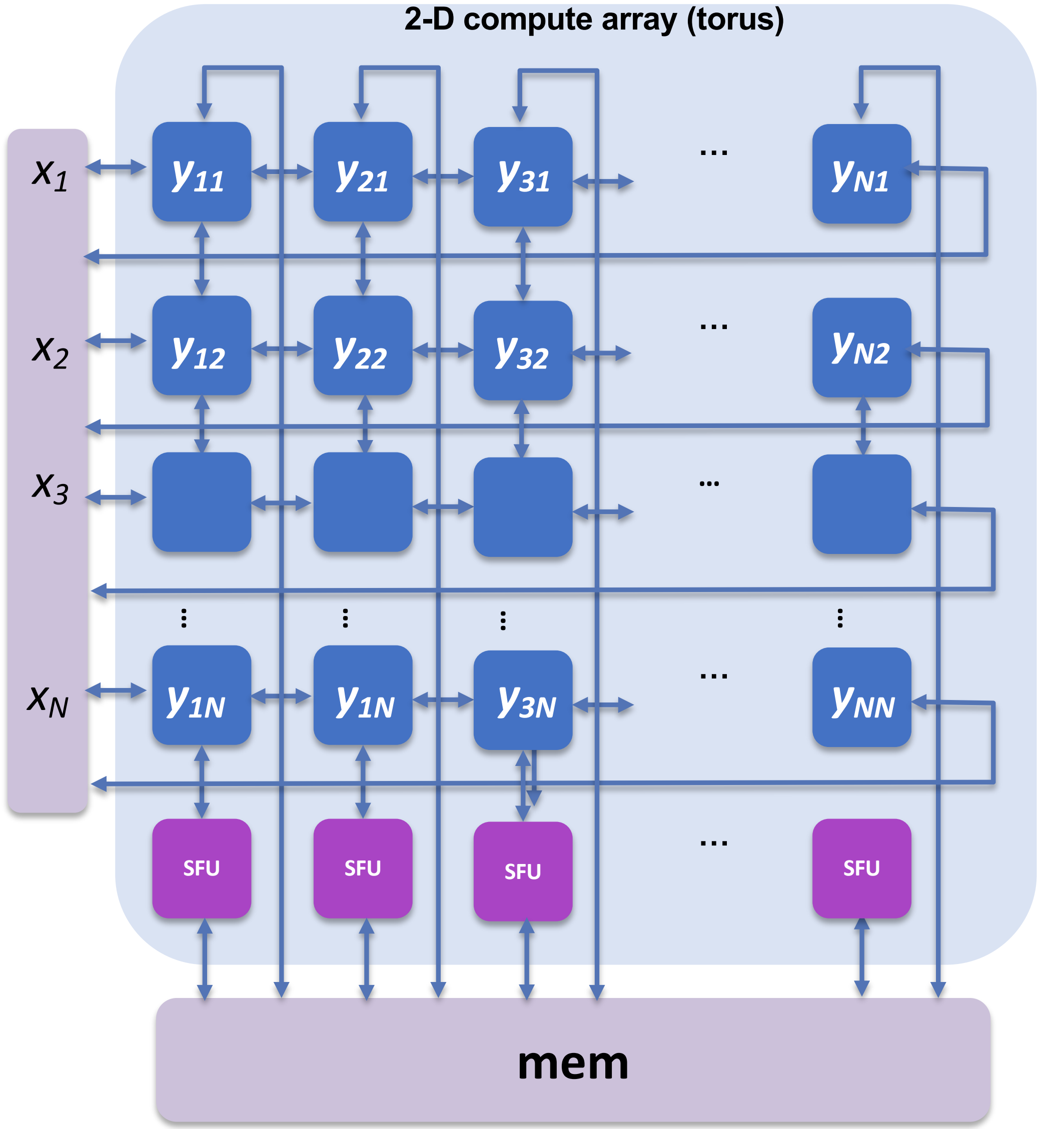


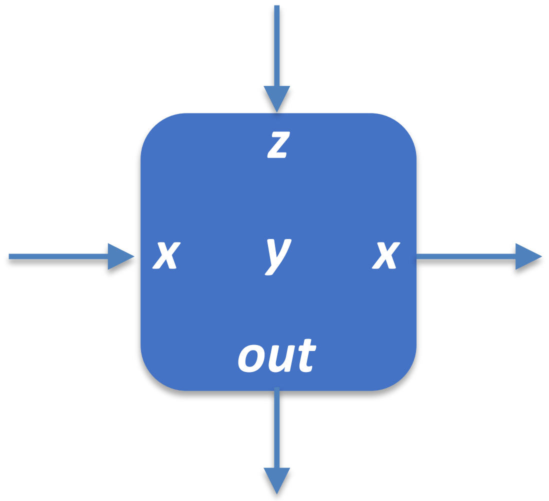


IBM Rapid **Google TPU** **Intel Nervana**

J.Oh, “A 3.0 TFLOPS 0.62V Scalable Processor Core for High Compute Utilization AI Training and Inference“
https://arxiv.org/pdf/1704.04760.pdf
https://fuse.wikichip.org/news/3270/intel-axes-nervana-just-two-months-after-launch/

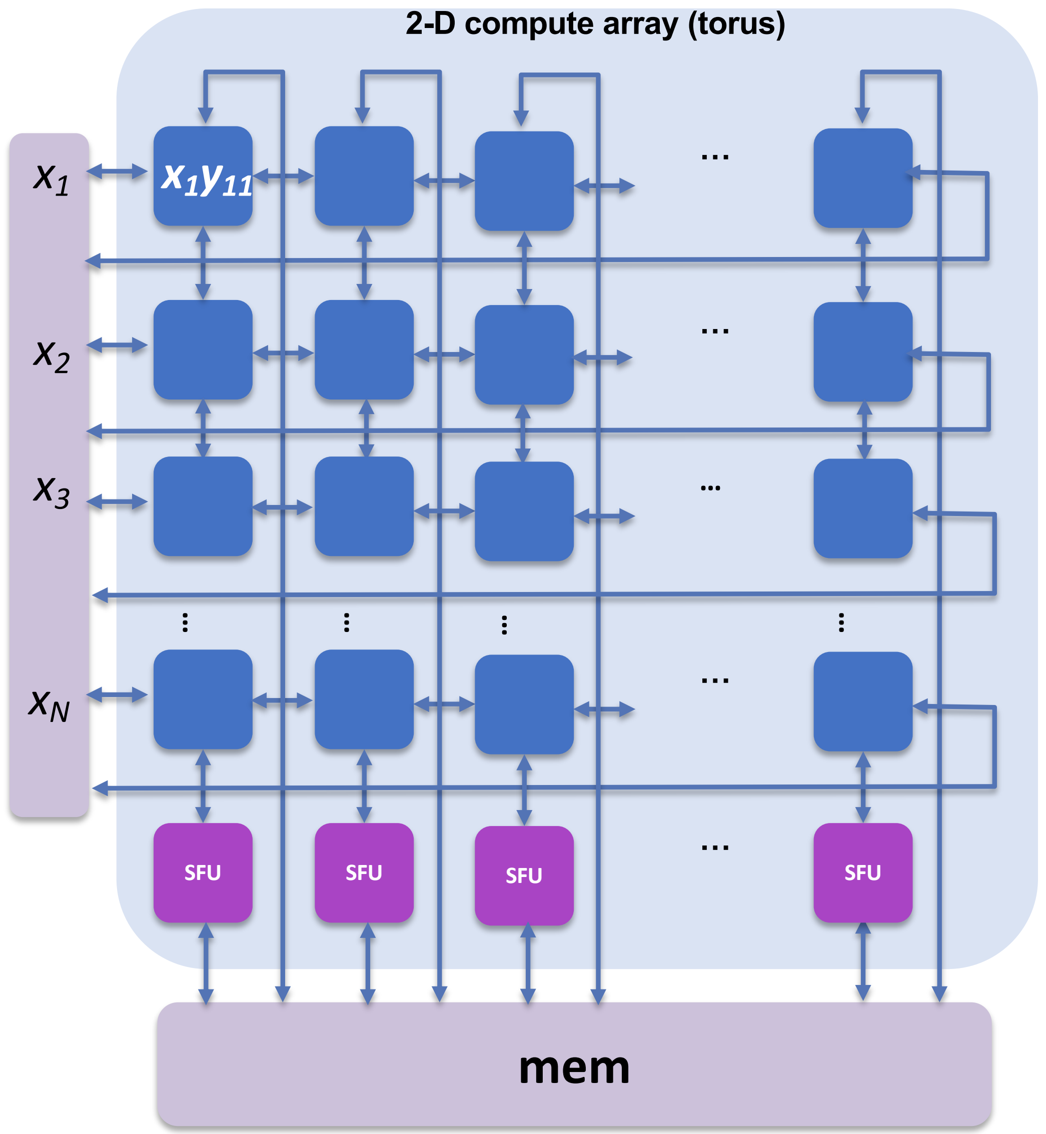
2D Systolic Array Architecture





$$out = z + xy$$

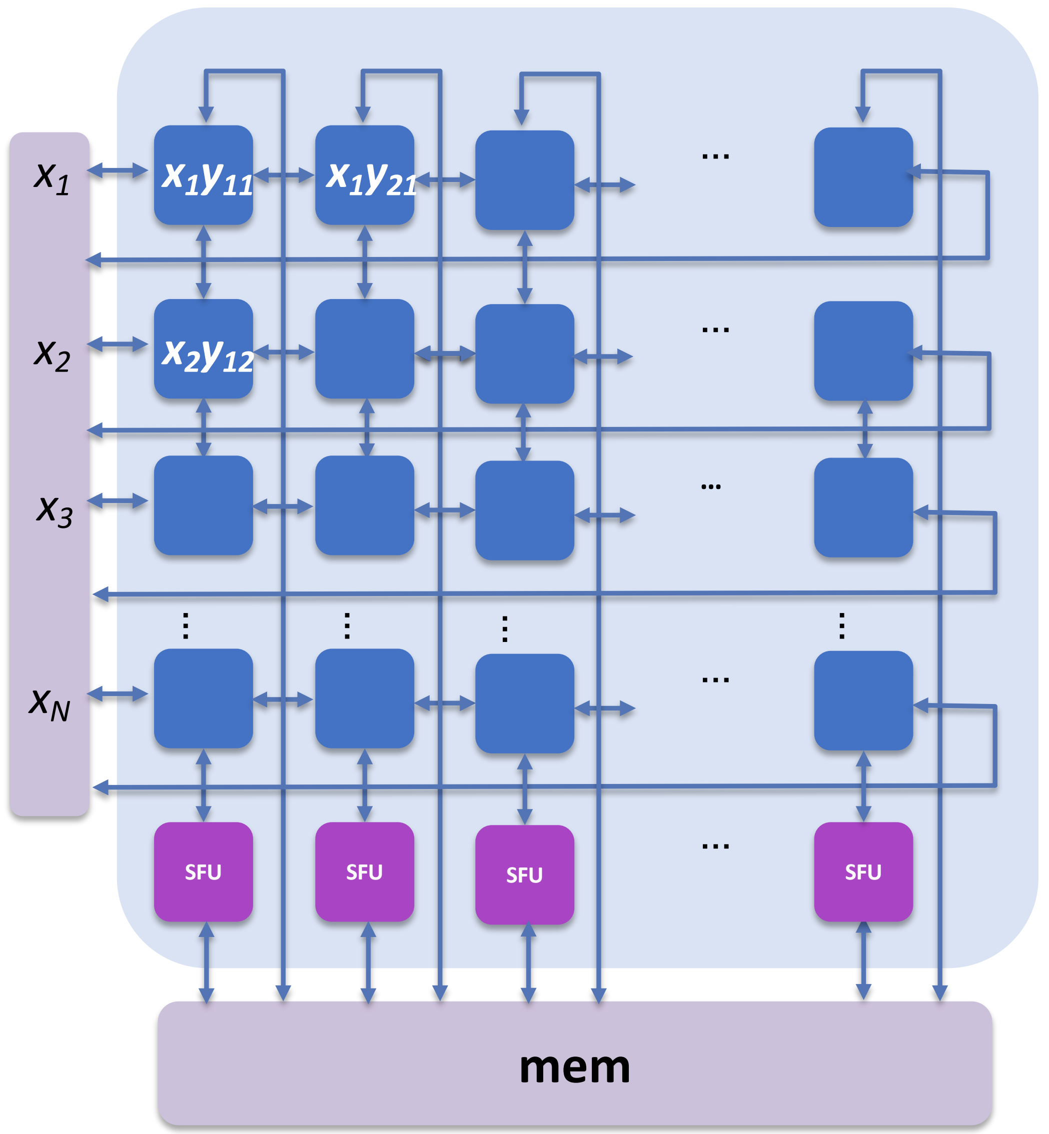
2D Systolic Array Architecture



1-st cycle

$0 + x_1y_{11}$

2D Systolic Array Architecture

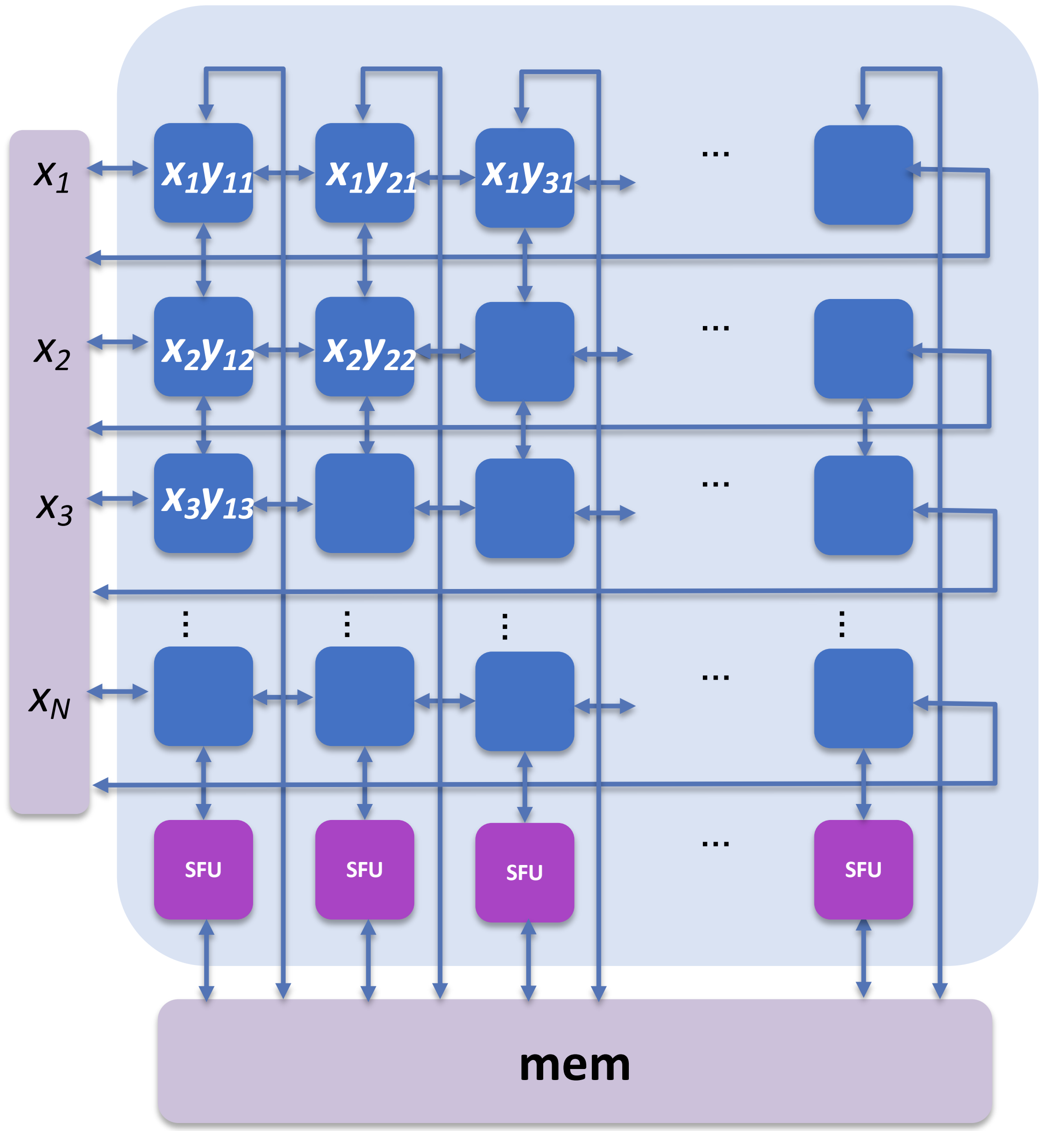


$0 + x_1y_{11}$ $0 + x_1y_{21}$

$x_1y_{11} + x_2y_{12}$

2-nd cycle

2D Systolic Array Architecture



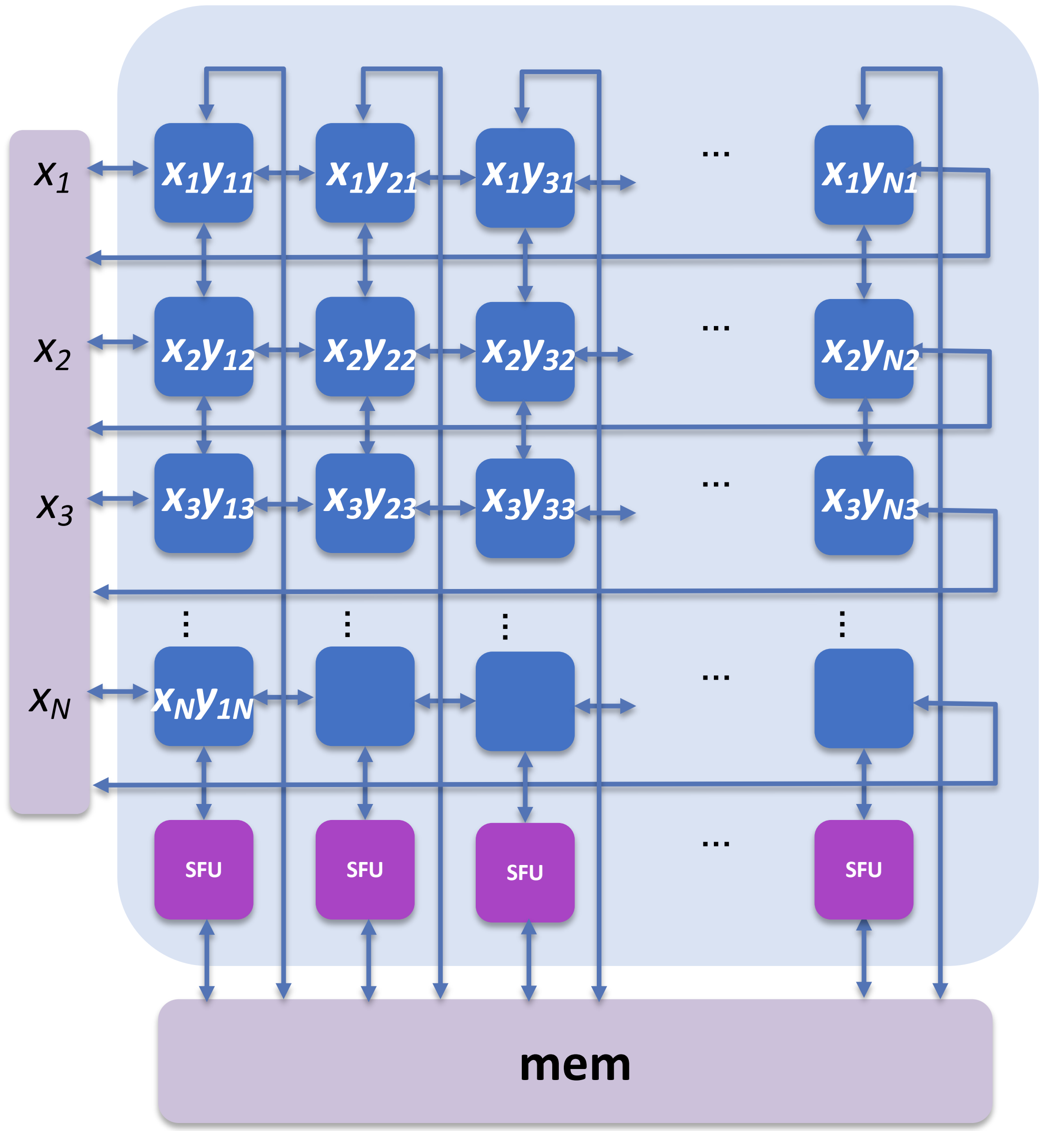
$0 + x_1y_{11}$ $\quad$ $0 + x_1y_{21}$ $\quad$ …

$x_1y_{11} + x_2y_{12}$ $\quad$ $x_1y_{21} + x_2y_{22}$

$x_1y_{11} + x_2y_{12} + x_3y_{13}$

3-rd cycle

2D Systolic Array Architecture



$0 + x_1y_{11}$ $\quad$ $0 + x_1y_{21}$ $\quad \dots$

$x_1y_{11} + x_2y_{12}$ $\quad$ $x_1y_{21} + x_2y_{22}$

$x_1y_{11} + x_2y_{12} + x_3y_{13}$ $\quad$ $x_1y_{21} + x_2y_{22} + x_3y_{23}$

$\vdots$ $\quad$ $\vdots$

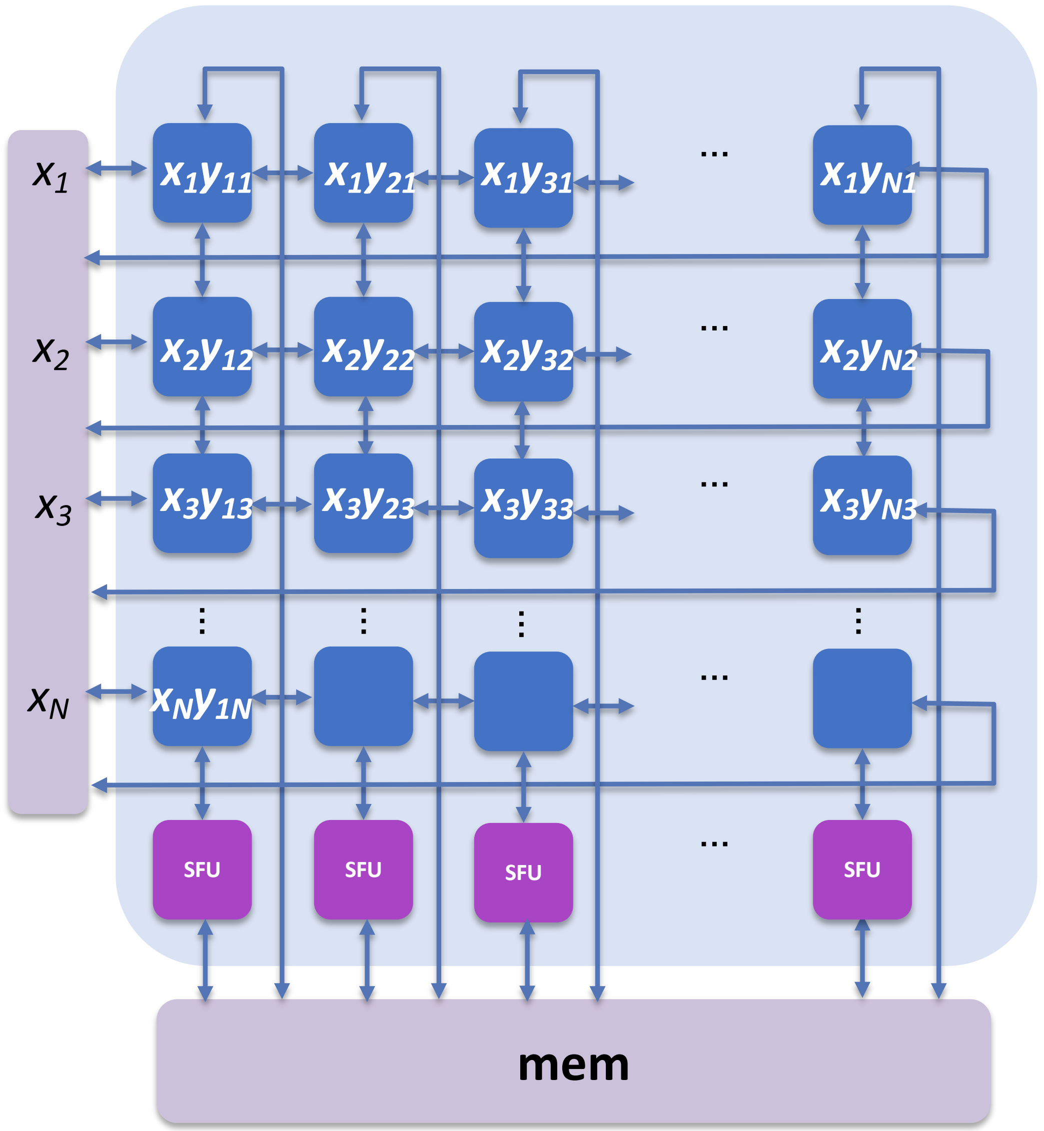
$x_1y_{11} + \dots\ x_{(N-1)}y_{1(N-1)}$ $\quad$ $x_1y_{21} + \dots\ x_{(N-1)}y_{2(N-1)}$

$x_1y_{11}\ \dots + \dots\ x_Ny_{1N}$

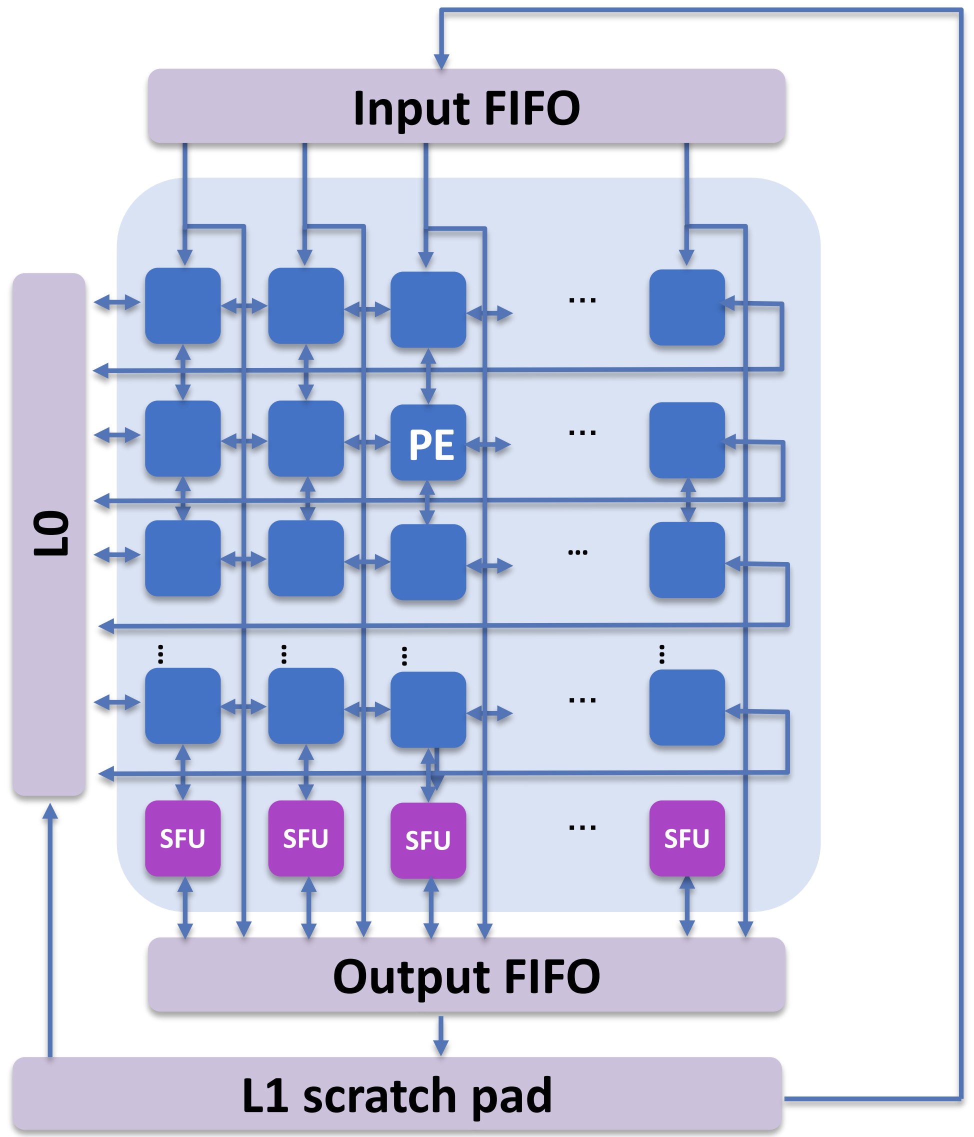
N-th cycle

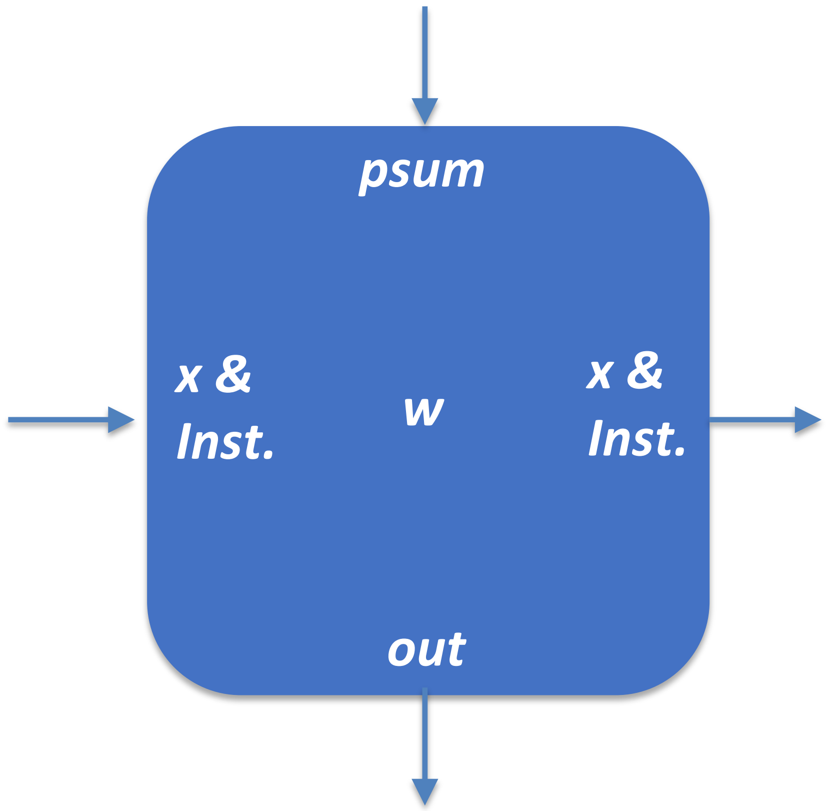
2D Systolic Array Architecture

Row #	*t = 0*	1	2	3	4	5
1	x_{11}	x_{21}	x_{31}	x_{41}		
2		x_{12}	x_{22}	x_{31}	x_{42}	
3			x_{13}	x_{23}	x_{31}	x_{43}
4				...	...	...
5					x_{1N}	x_{2N}



Complete Architecture and Instruction Flow





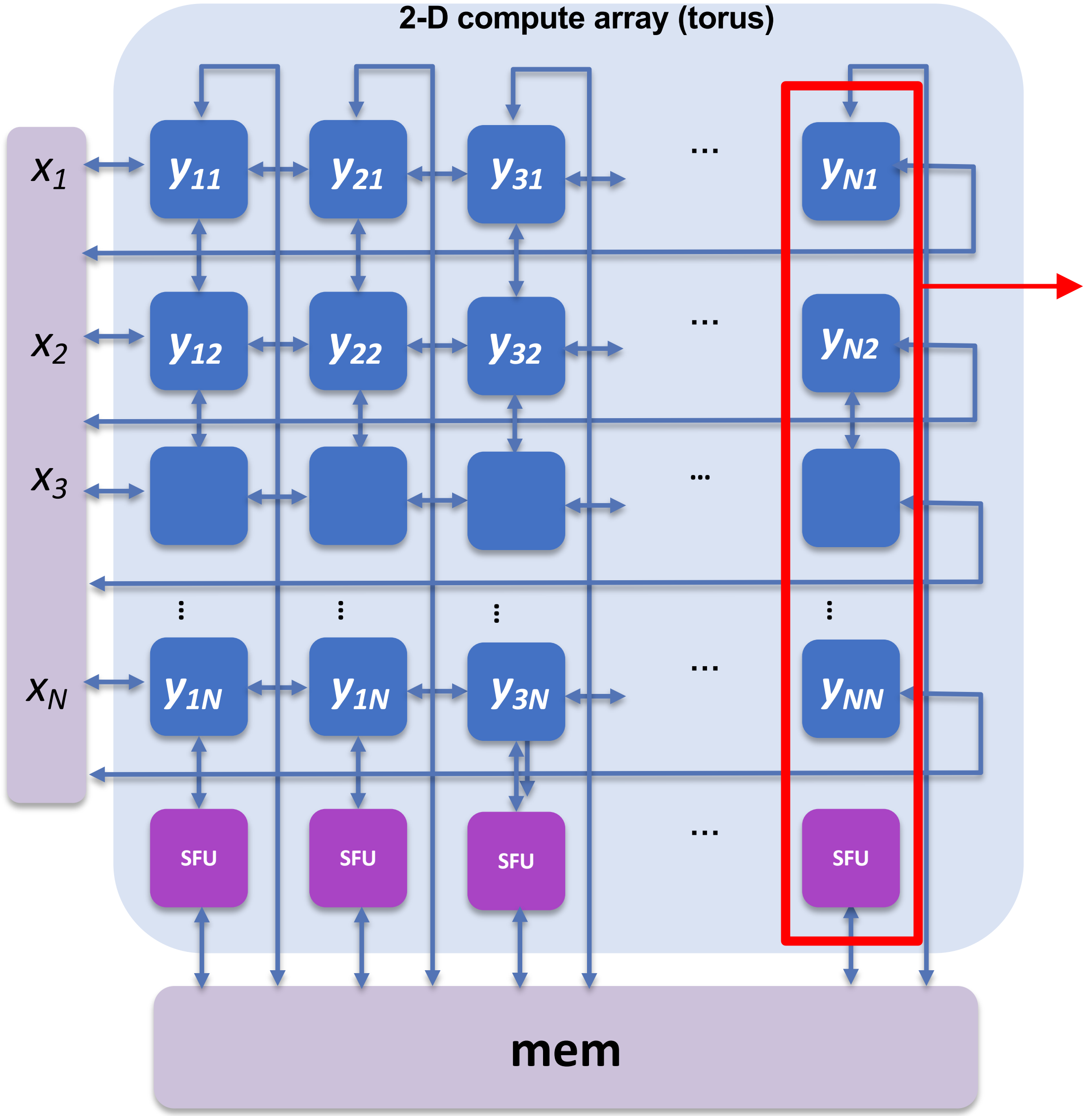
Data and Instruction flow

J.Oh, S. Lee, M. Kang, at al., “A 3.0 TFLOPS 0.62V Scalable Processor Core for High Compute Utilization AI Training and Inference”, VLSI symp 2020

Why 2D Systolic Array ?

- High data reuse in the local register
- Pipeline for each PE -> horizontal and vertical flow is pipelined -> high frequency
- Architecture regularity
- Efficient data movement -> each PE transfers the data to the neighbor creating a bus
- Simple instruction flow
- Optimal for dense matrix multiplication

1D Vector Processor

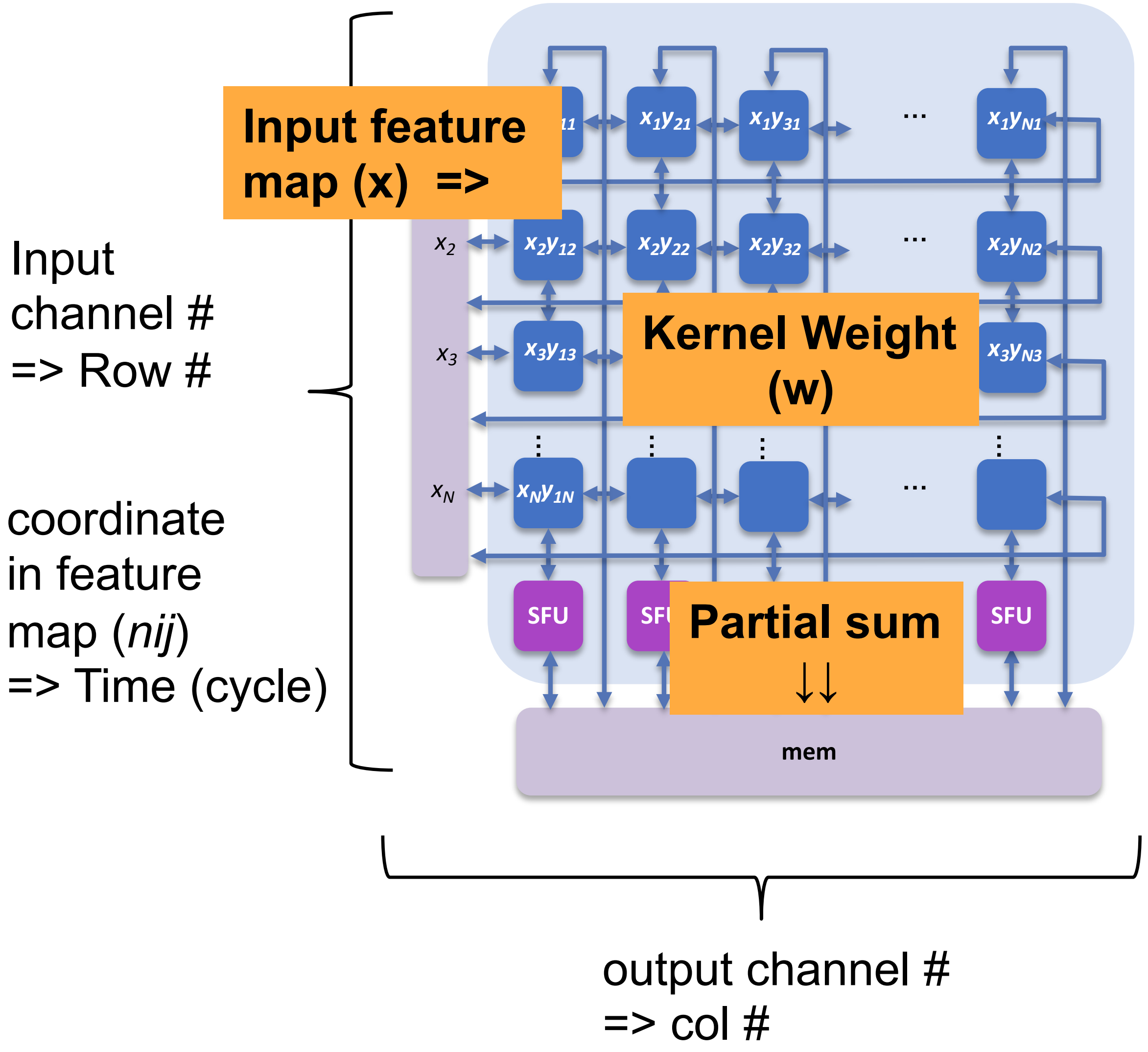


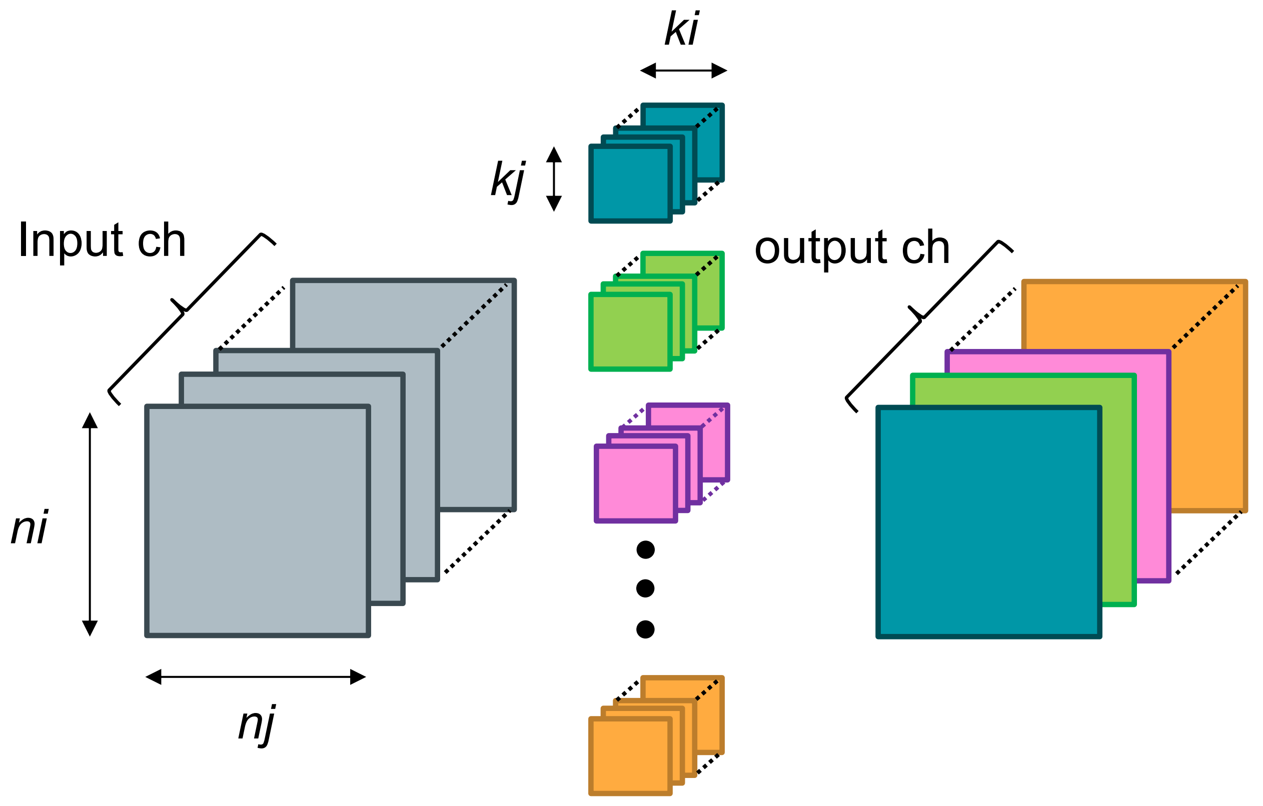
Merged as a single dot product processor

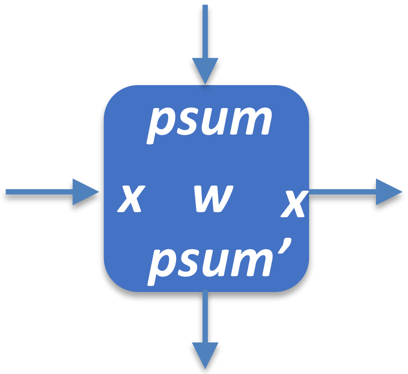
e.g., NVDIA NVDLA
Microsoft Brainwave

How to Enable Convolution ? (Window sliding)

Algorithm to Hardware Mapping (Weight Stationary mapping)

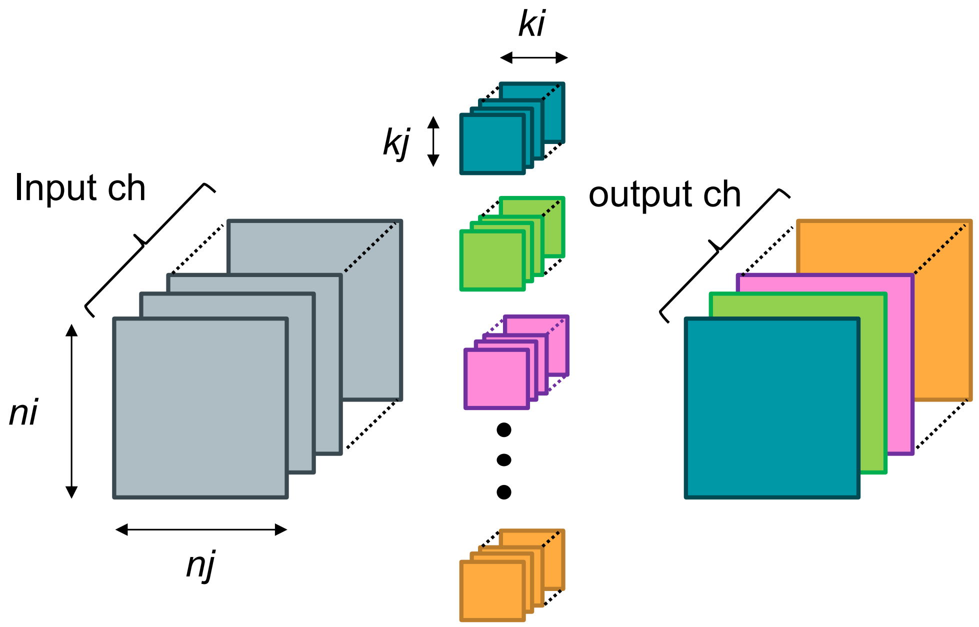






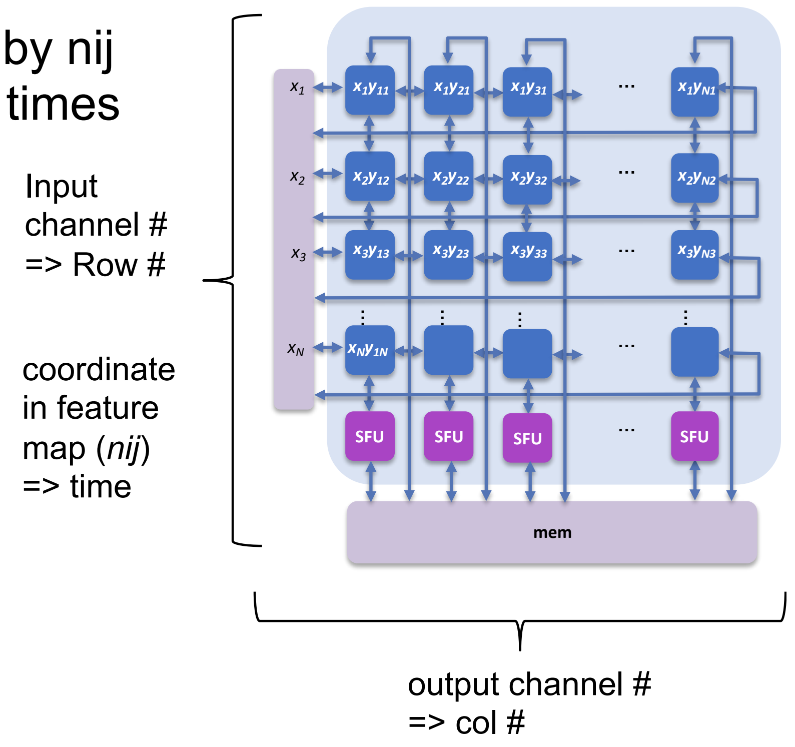
$$psum' = psum + xw$$

Algorithm to Hardware Mapping (Conv to 2D array)



Assumption: 3X3 kernel, 16X16 input feature map
64 in / out channels

Weight is reused by nij
(49 – thousands) times



Matrix multiplication

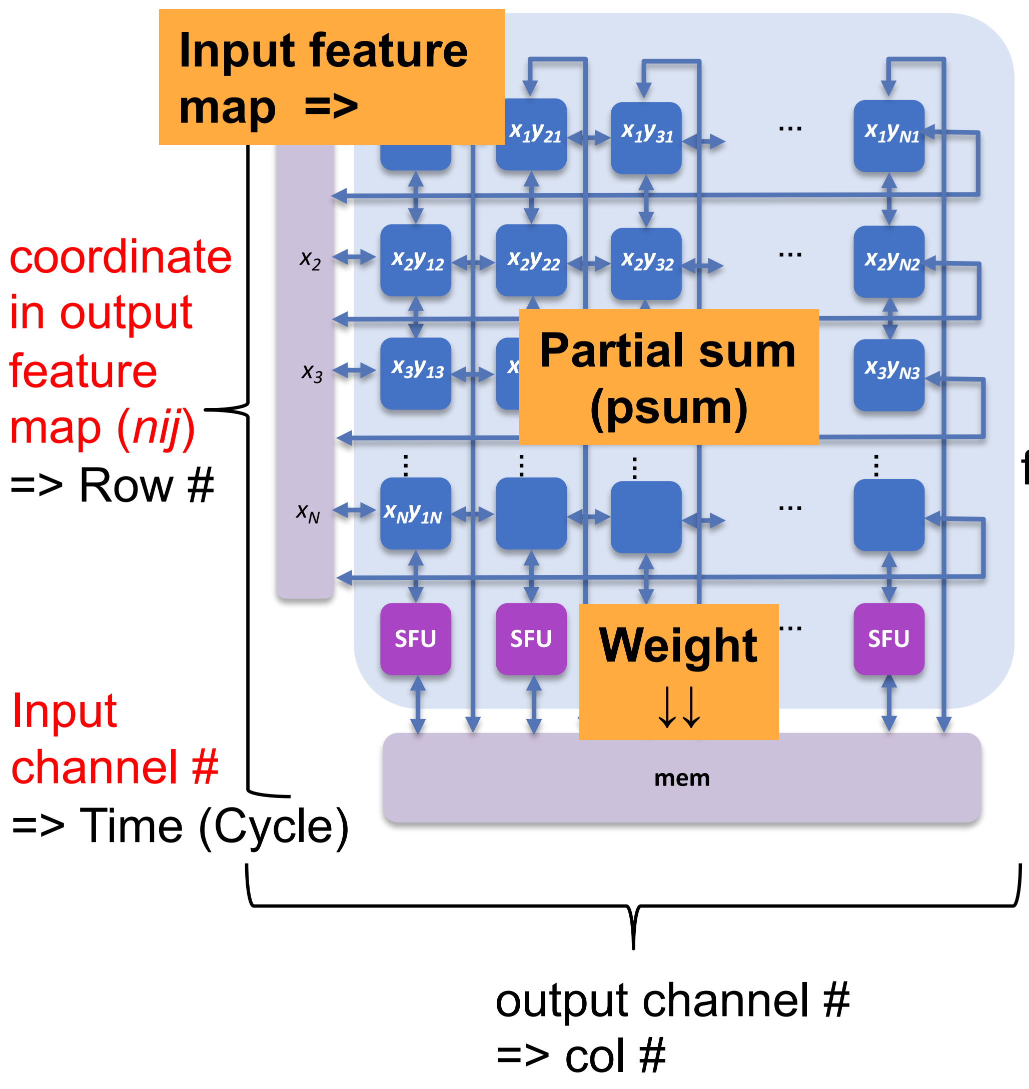
```
for kij = 0:8 (time, renew all the weights in registers)
    for out_ch = 0:63 (col #)
       for in_ch = 0:63 (row #)
           for nij = 0:255 (time for horizontal input)
                 psum(out_ch, kij, nij)  += w(out_ch, in_ch, kij) * x(in_ch, nij)
```

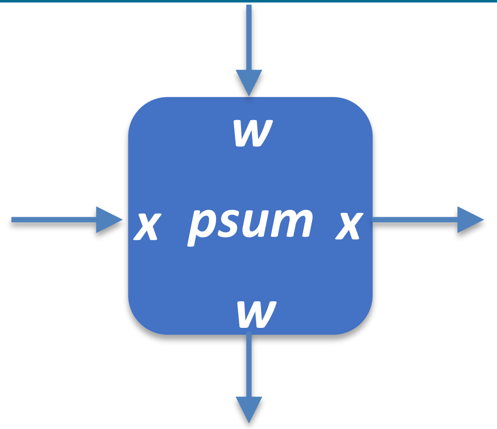
Accumulation (SFU)

```
for nij = 0:255 (output index)
   for kij = 0:8 (time)
        output (out_ch, nij) += psum(out_ch, kij, nij')
```

- Note nij' = f(nij, kij) is shifted index of nij for conv.
- Matmult and acc can be processed simultaneously.

Output Stationary mapping



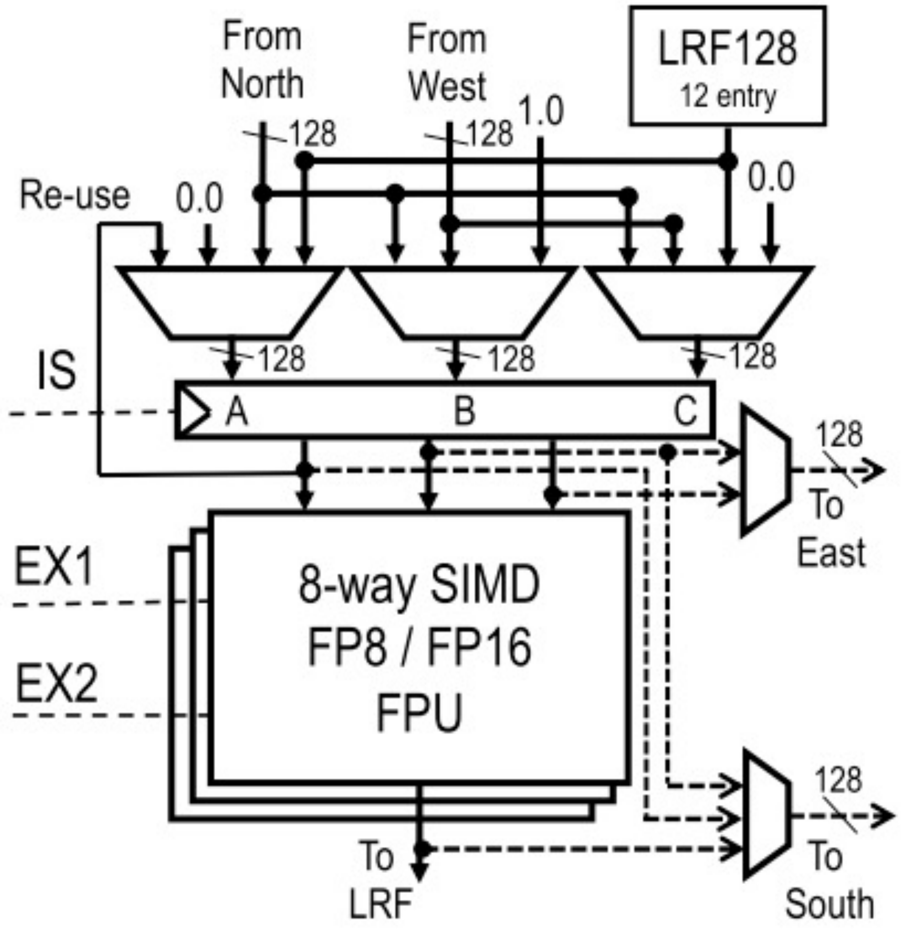


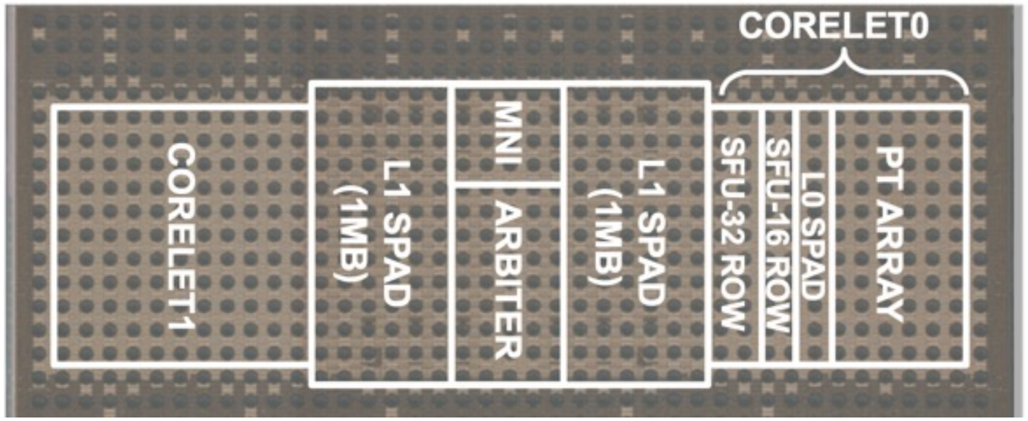
$$psum' = psum + xw$$

```
for out_ch = 0:63 (col #)
  for nij_out = 0:63 (row #)
    for in_ch = 0:2 (time for horizontal input)
      for kij = 0:8 (time)
        psum(out_ch, nij_out) += w(out_ch, in_ch, kij) * x(in_ch, nij')
```

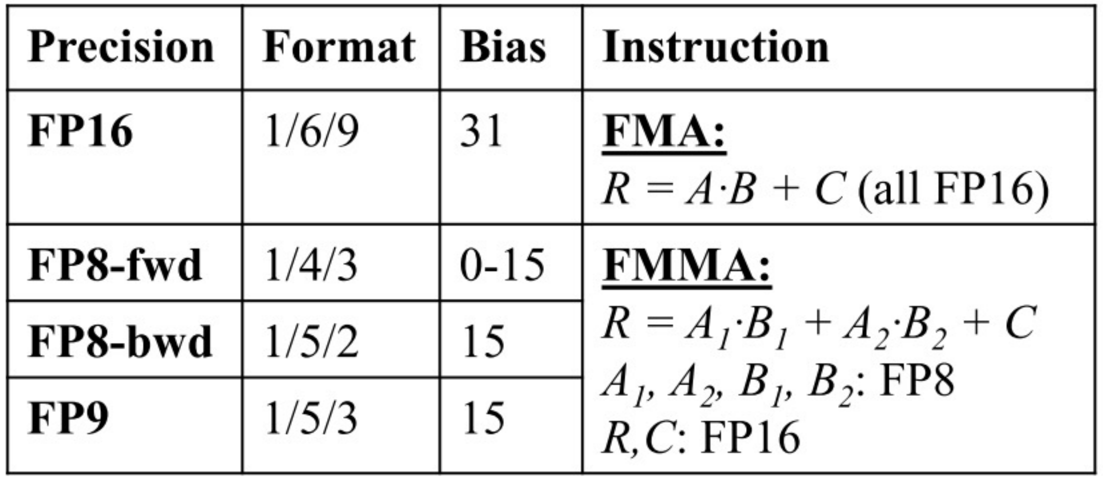
- Note nij' = f(nij, kij) is shifted index of nij for conv
- This is optimal when the input and output channel numbers are not large enough to fill the array, e.g., first layer

Flexibility / Programmability in Data Flow





14nm AI accelerator, VLSI20



Precision	**Format**	**Bias**	**Instruction**
FP16	1/6/9	31	**FMA:** $R = A \cdot B + C$ (all FP16)
FP8-fwd	1/4/3	0-15	**FMMA:** $R = A_1 \cdot B_1 + A_2 \cdot B_2 + C$ A_1, A_2, B_1, B_2: FP8 R, C: FP16
FP8-bwd	1/5/2	15	
FP9	1/5/3	15	