

Processing Elements and Datapath

Instructor: Siting Liu

School of Information Science and Technology (SIST), ShanghaiTech University

Shanghai Institute of Semiconductor Innovation (SISI)

<https://faculty.sist.shanghaitech.edu.cn/liust/EE219/EE219.html>



Computer Architecture - EE219
faculty.sist.shanghaitech.edu.cn

立志成才 报国强民



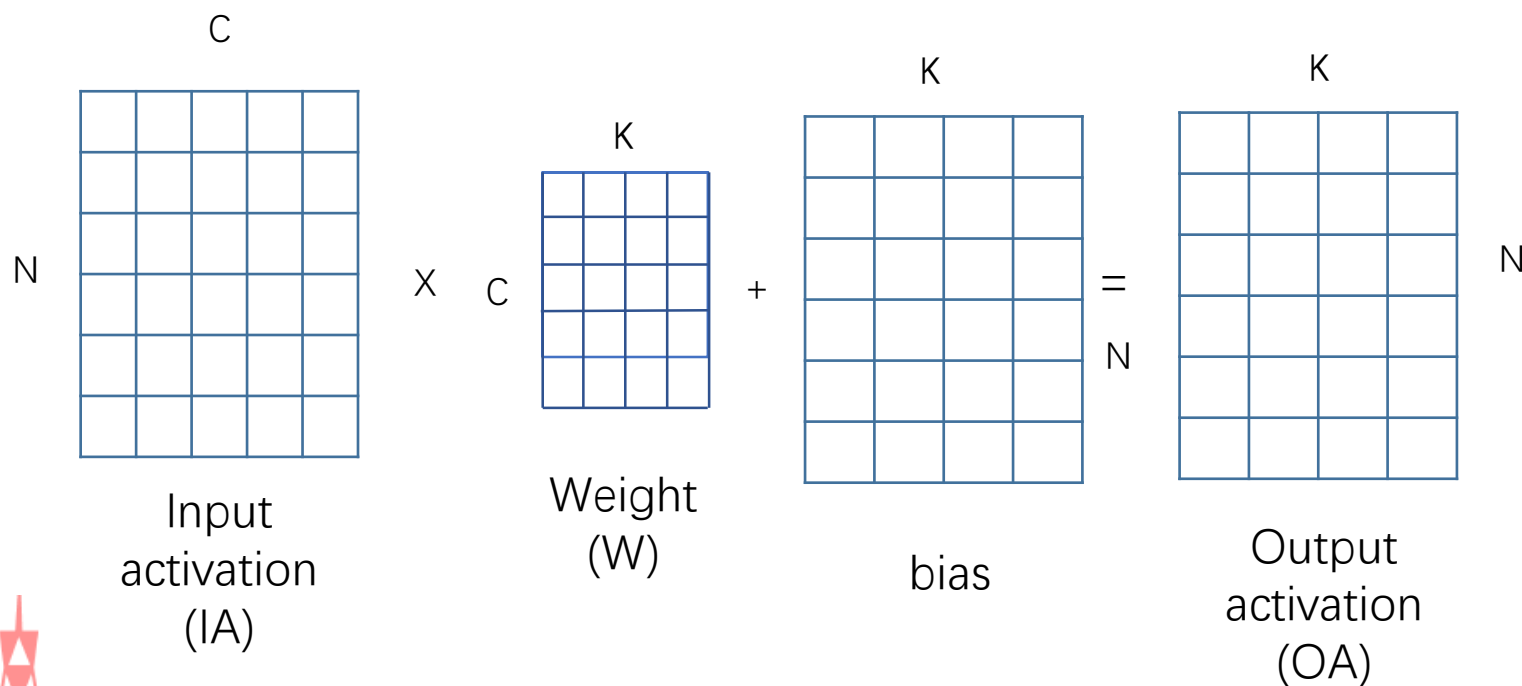
Review

- Algorithm analysis
- Model Compression
- Kernel computation
- Mainly matrix multiplication
- Datapath design (processing elements & memory consideration)



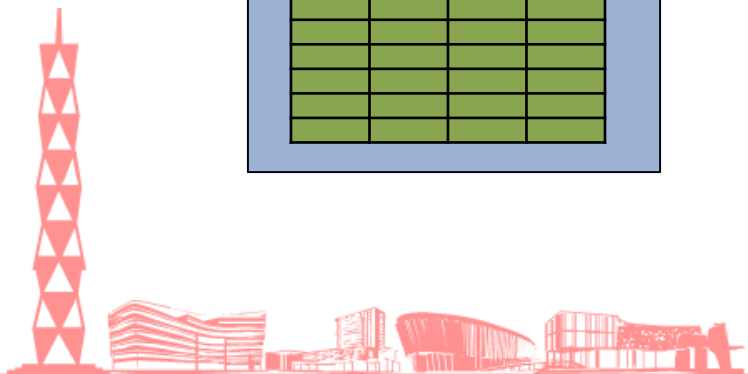
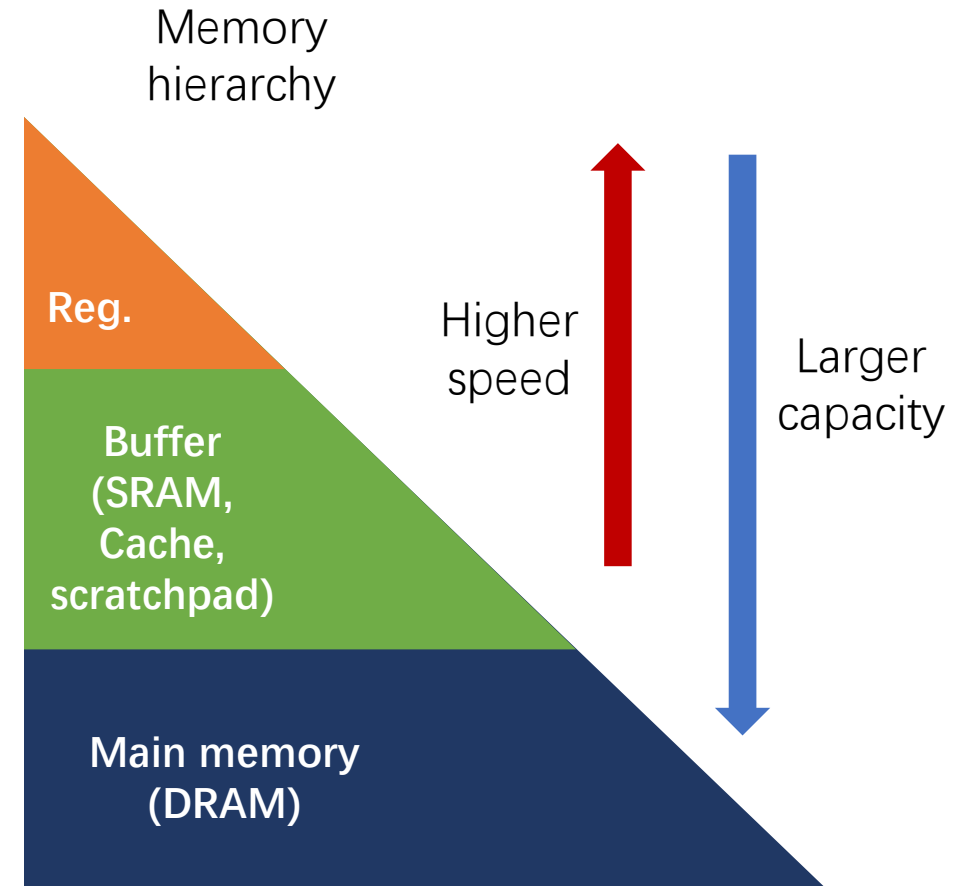
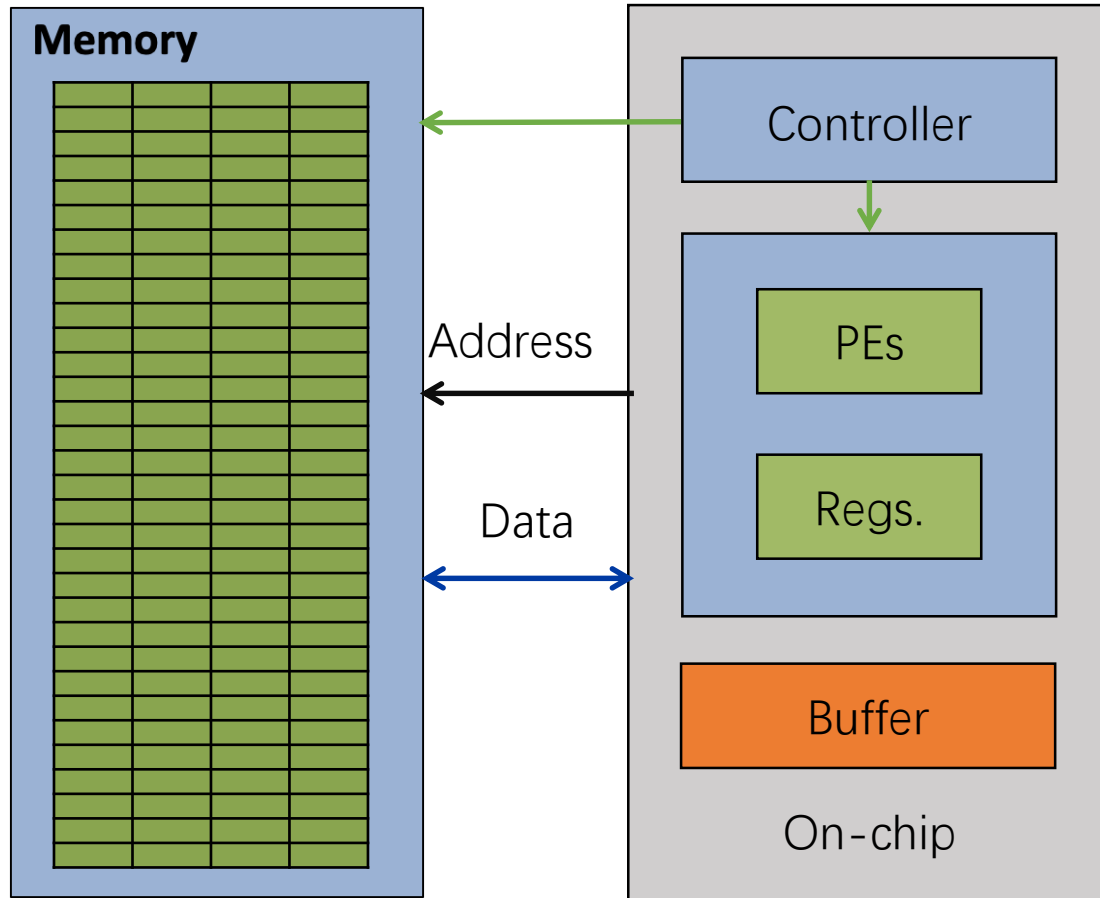
Targeting the Common Cases

- GEMM



```
for (n=0; n<N; n++) {  
    for (k=0; k<K; k++){  
        OA[n, k] = b[n, k];  
        for (c=0; c<C; c++){  
            OA[n, k] = OA[n, k] + IA[n, c]*W[c, k];  
        }  
        OA[n, k] = Act(OA[n, k]);  
    }  
}
```

General Digital System (Simplified)



Datapath – Basic PE

```
for (m=0; m<M; m++) {  
  for (n=0; n<N; n++){  
    OA[m, n] = b[m, n];  
    for (k=0; k<K; k++){  
      OA[m, n] = OA[m, n] + X[m, k]*W[k, n];  
    }  
    OA[m, n] = Act(OA[m, n]);  
  }  
}
```



$C += A * B$



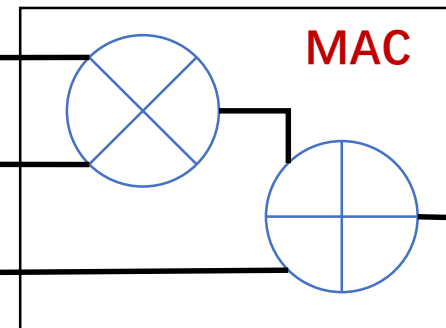
Memory read

Regs.

Filter weight

Fmap/activation

Partial sum



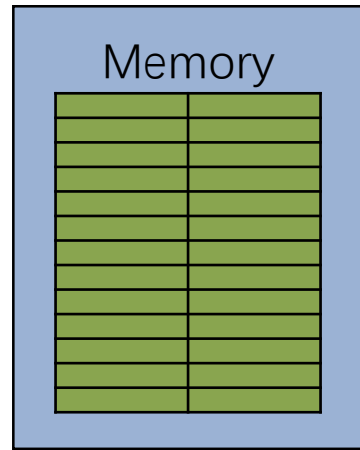
Memory write/read
Updated **partial sum**

Reg.



Output-stationary Dataflow

- Maximize output reuse



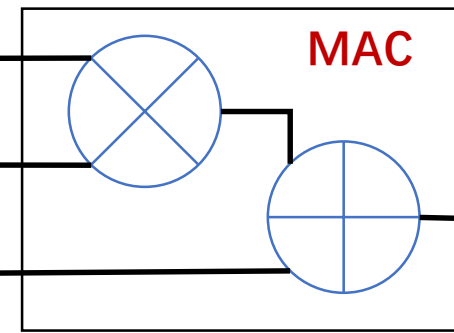
Memory read



Filter weight

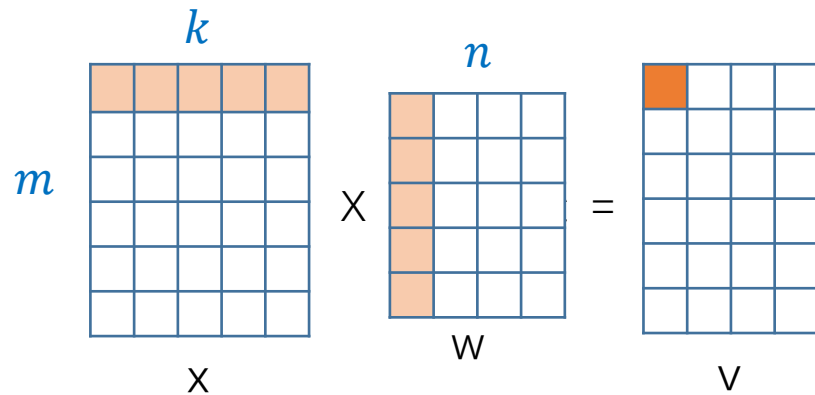
Fmap/activation

Partial sum



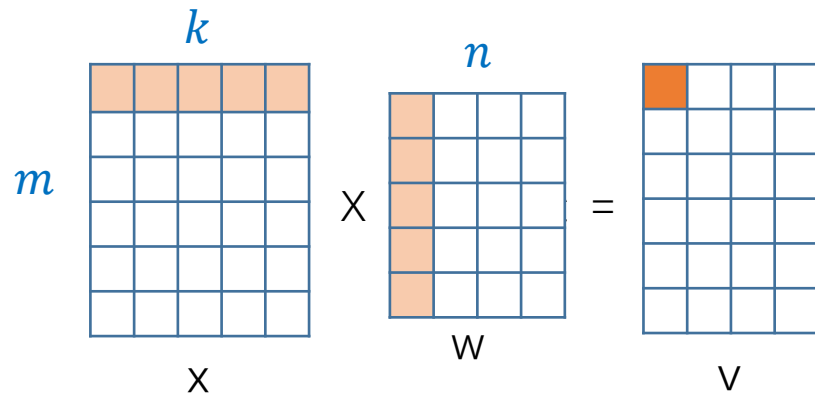
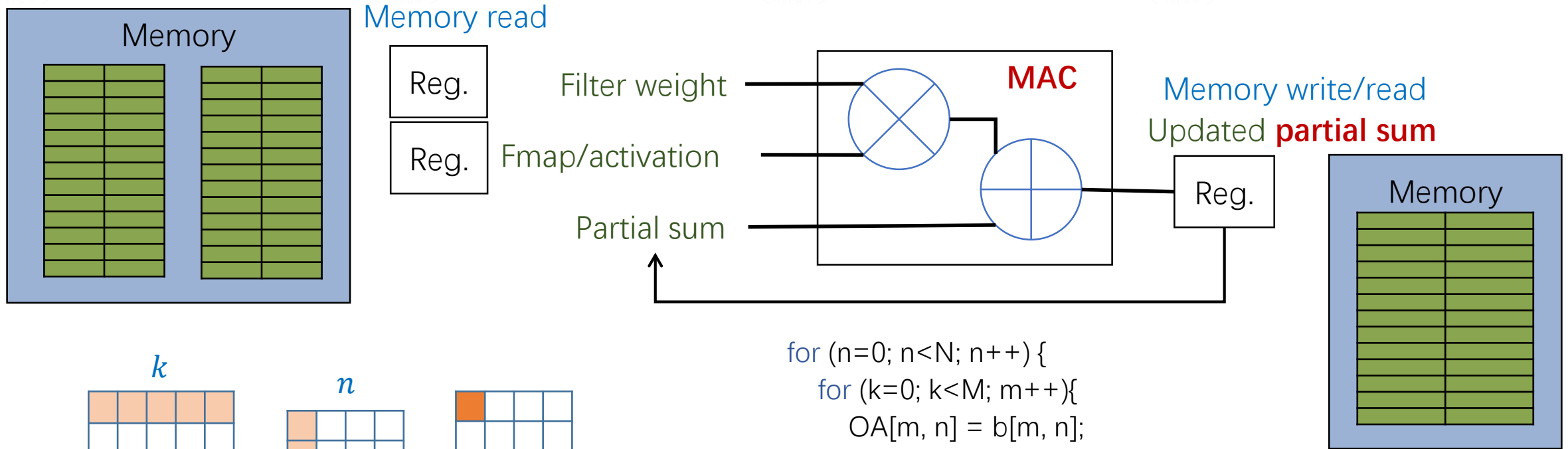
Memory write/read

Updated **partial sum**



Output-stationary Dataflow (Cont'd)

- Maximize output reuse

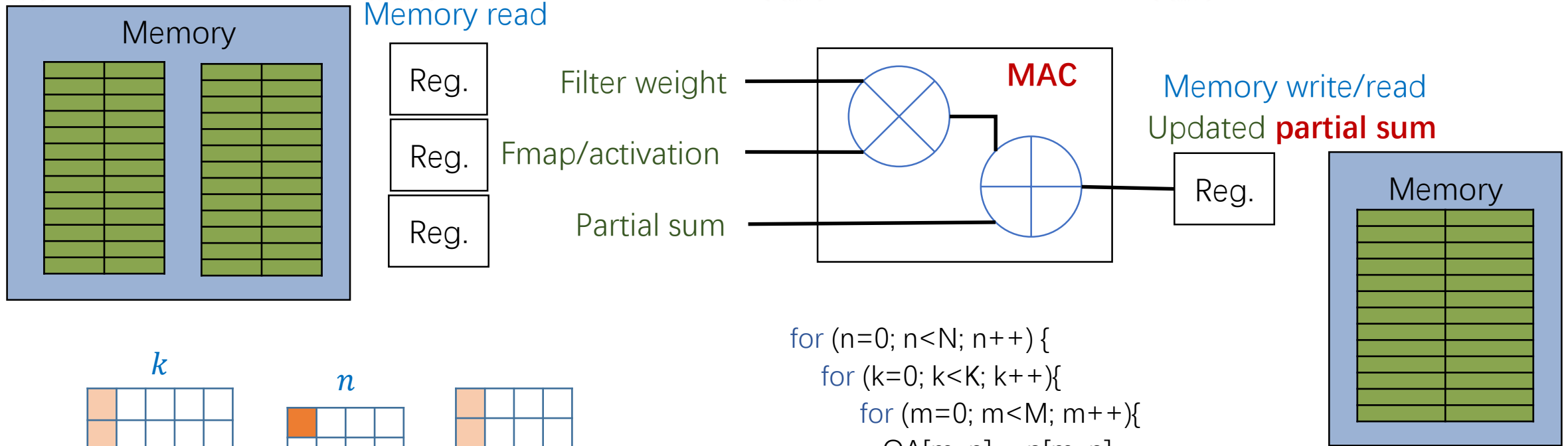


```

for (n=0; n<N; n++) {
  for (k=0; k<M; k++) {
    OA[m, n] = b[m, n];
    for (k=0; k<K; k++) {
      OA[m, n] = OA[m, n] + X[m, k]*W[k, n];
    }
    OA[m, n] = Act(OA[m, n]);
  }
}
    
```

Weight-stationary Dataflow

- Maximize weight reuse

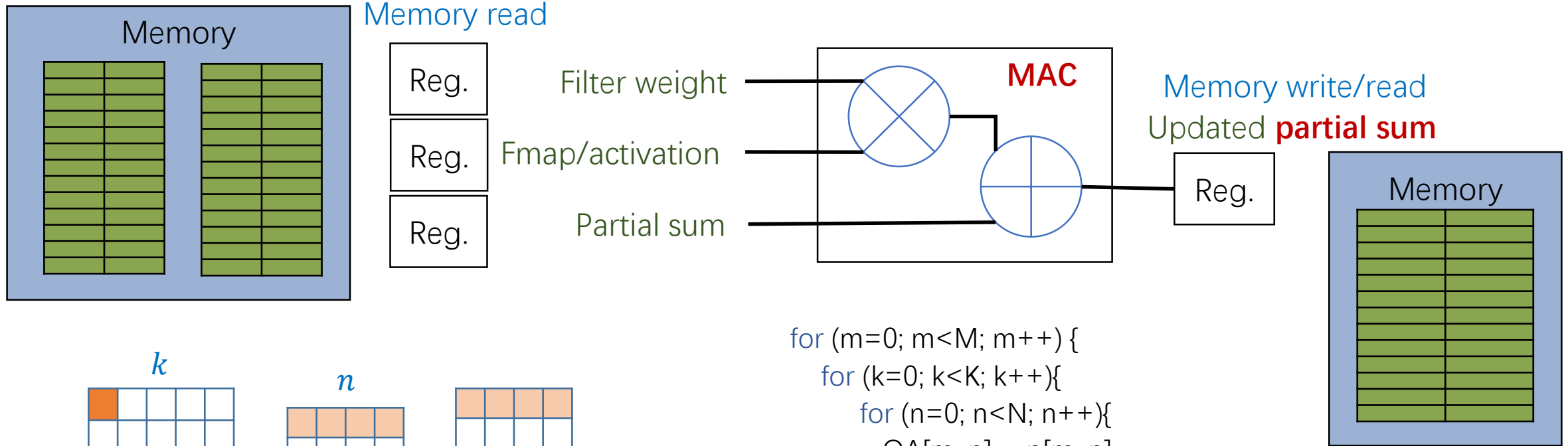


```

for (n=0; n<N; n++) {
  for (k=0; k<K; k++) {
    for (m=0; m<M; m++) {
      OA[m, n] = p[m, n];
      OA[m, n] = OA[m, n] + X[m, k]*W[k, n];
    }
    OA[m, n] = Act(OA[m, n]);
  }
}
    
```

Input-stationary Dataflow

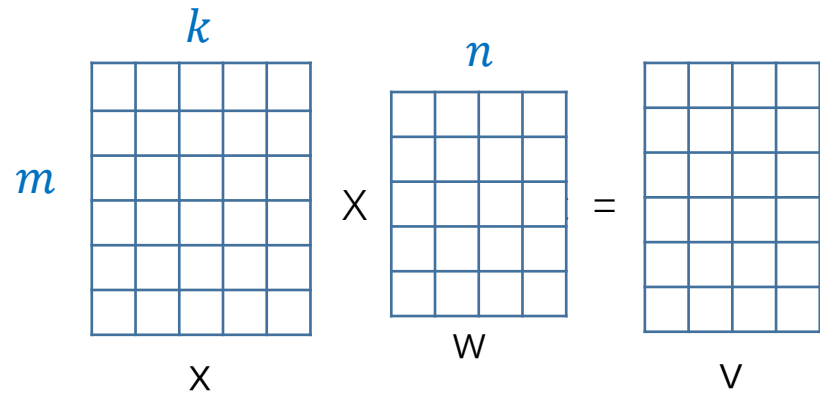
- Maximize input reuse



```

for (m=0; m<M; m++) {
  for (k=0; k<K; k++){
    for (n=0; n<N; n++){
      OA[m, n] = p[m, n];
      OA[m, n] = OA[m, n]+ X[m, k]*W[k, n];
    }
    OA[m, n] = Act(OA[m, n]);
  }
}
    
```

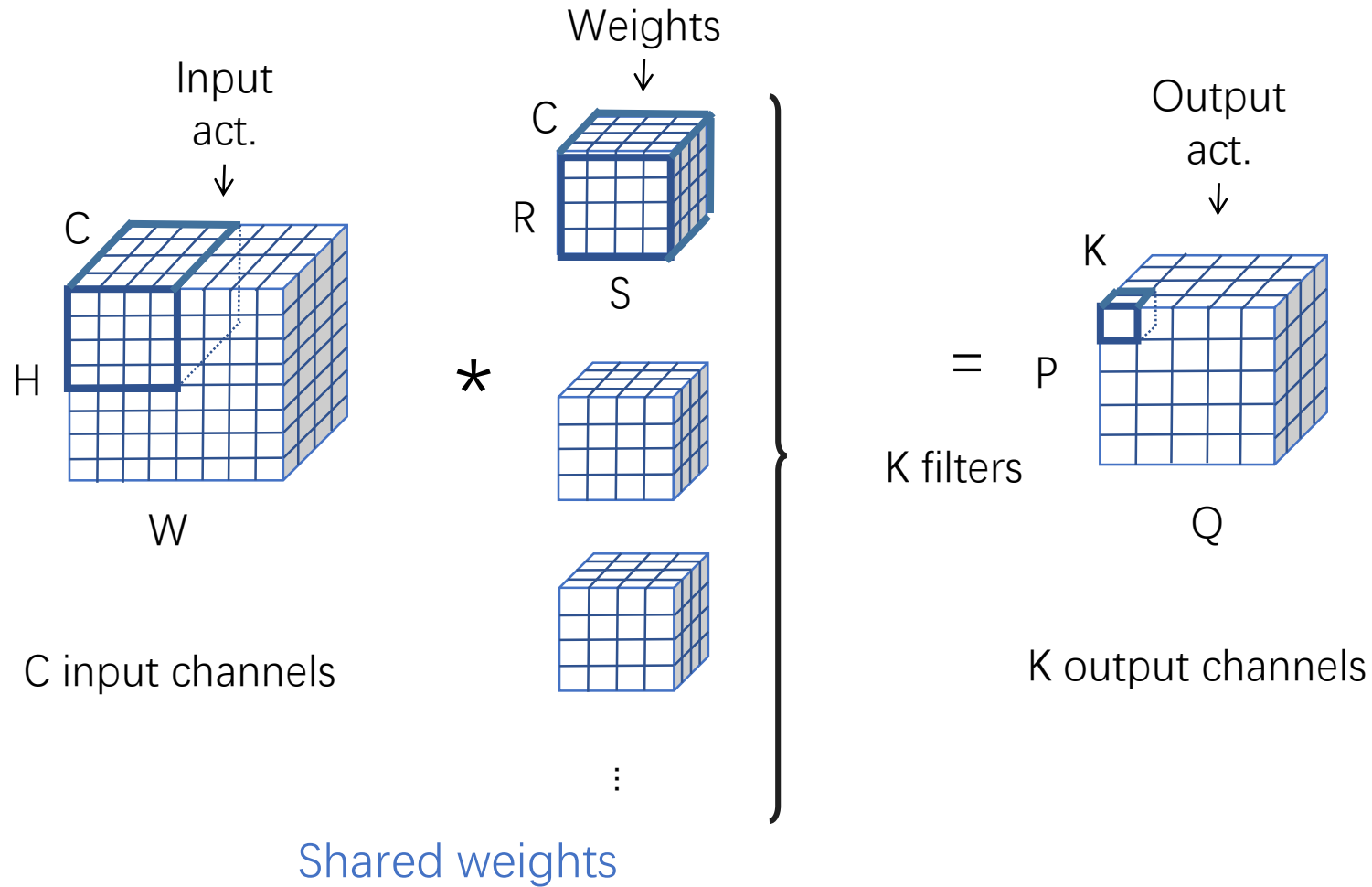
Data Reuse in GEMM



- Total MACs
- Input activations
- Weight
- Output Activations



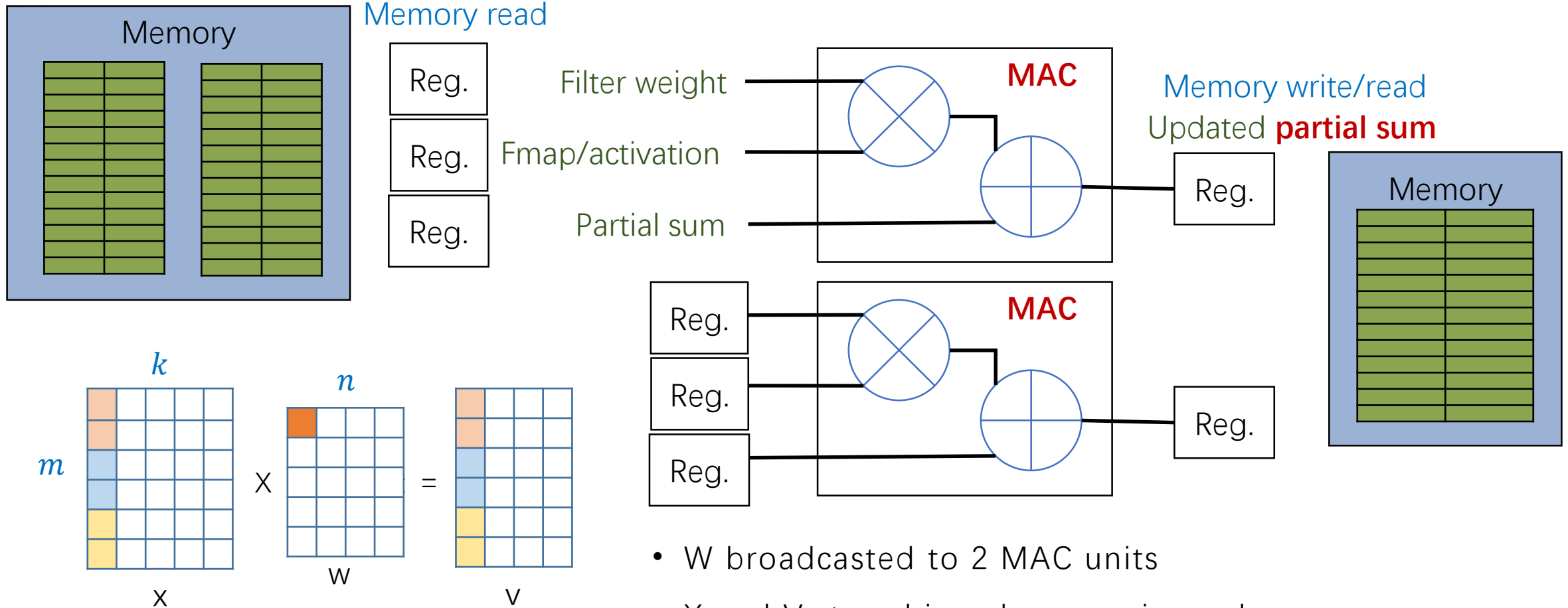
Data Reuse in Convolution



- Total MACs
- Input activations
- Weight
- Output Activations

Parallelization

- Weight-stationary for example



- W broadcasted to 2 MAC units
- X and V stored in column-major order

Summary

- Matrix multiplication can be implemented with basic PE or a MAC unit;
- Different dataflow (order of execution/for-loop) change the number of memory access, and it affects the energy cost and execution time;
- Increase the number of PEs (parallelism) improve the performance with ideal data layout and sufficient bandwidth.

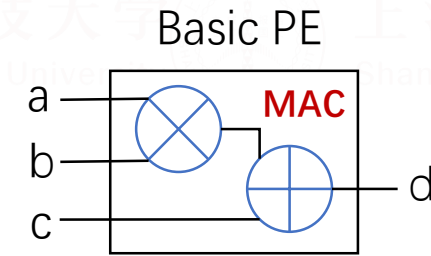
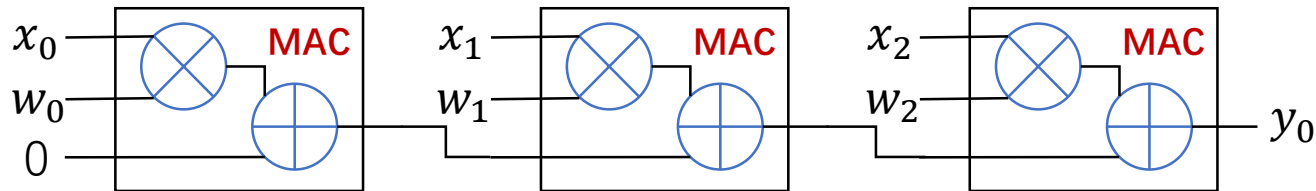


Dataflow Accelerator

- 1D convolution for example

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{c} * \\ \end{array} \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array}$$

$X \qquad \qquad \qquad W \qquad \qquad \qquad Y$

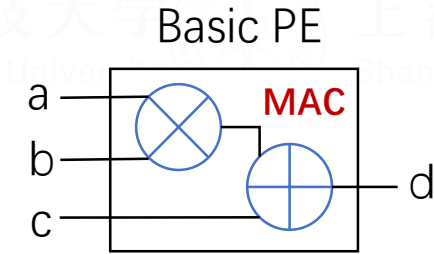
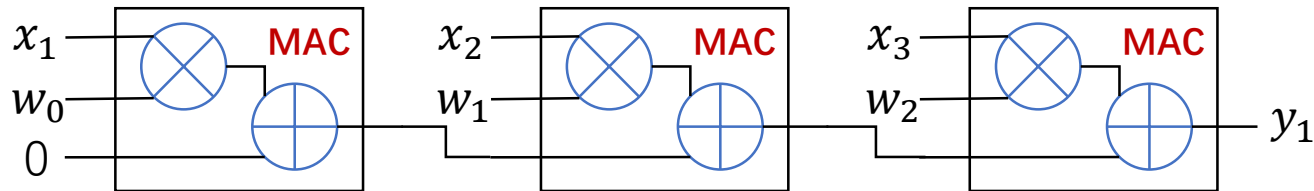


Dataflow Accelerator

- 1D convolution for example

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{c} * \\ \end{array} \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array}$$

$X \qquad \qquad \qquad W \qquad \qquad \qquad Y$

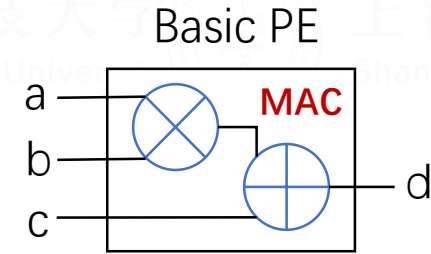
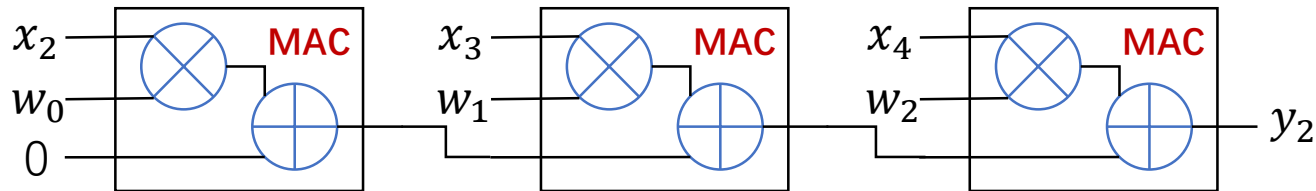


Dataflow Accelerator

- 1D convolution for example

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{c} * \\ \end{array} \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array}$$

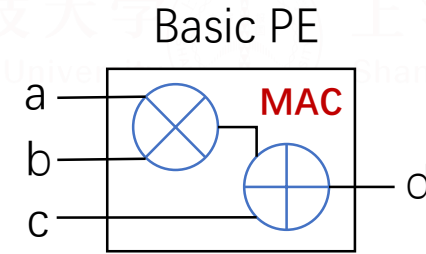
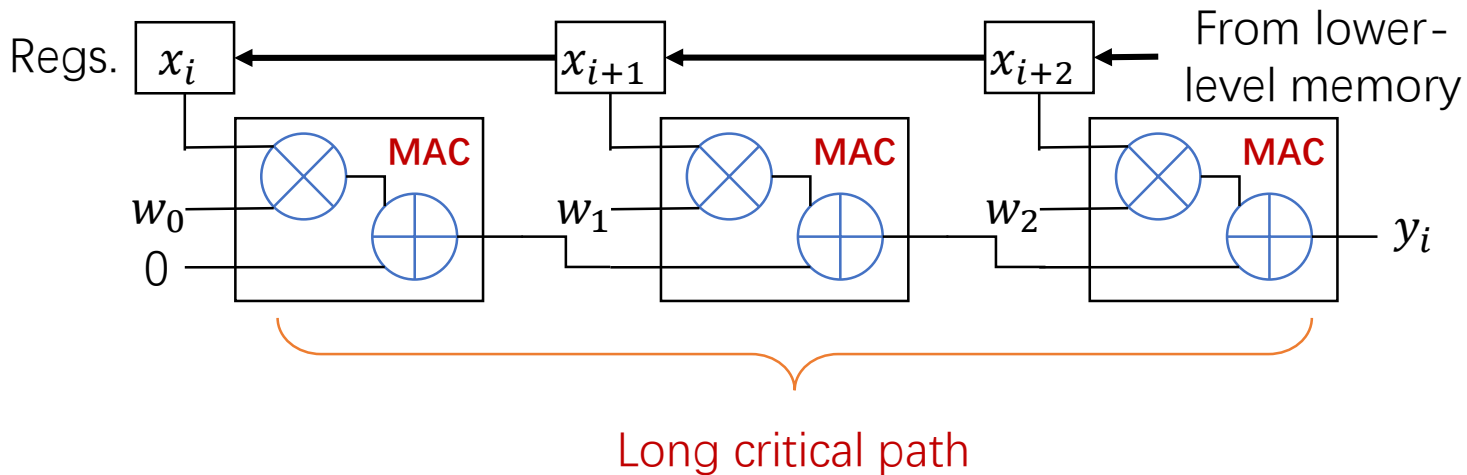
$X \qquad \qquad \qquad W \qquad \qquad \qquad Y$



Dataflow Accelerator

- 1D convolution for example

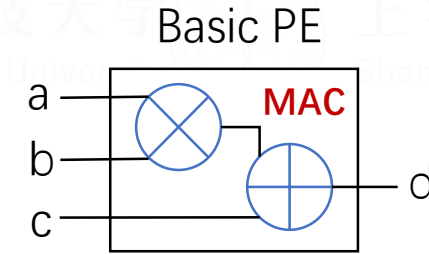
$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{c} X \end{array} \quad * \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} \quad \begin{array}{c} W \end{array} \quad = \quad \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array} \quad \begin{array}{c} Y \end{array}$$



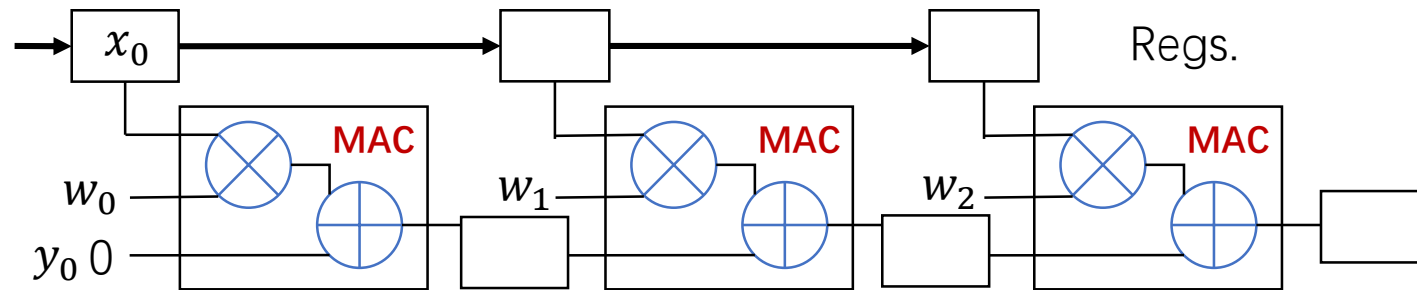
Dataflow Accelerator (Cont'd)

- 1D convolution for example

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \underset{X}{*} \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} \quad \underset{W}{=} \quad \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array} \quad \underset{Y}$$



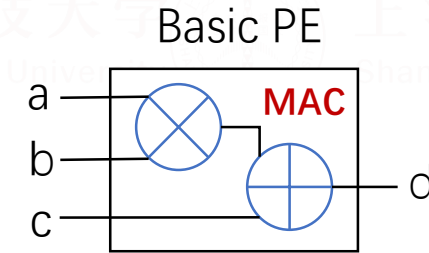
From lower-level memory



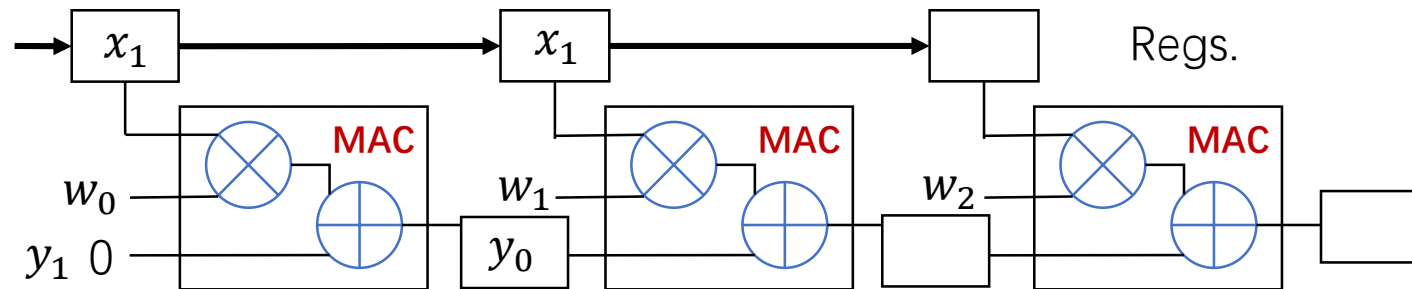
Dataflow Accelerator (Cont'd)

- 1D convolution for example

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{c} X \end{array} \quad * \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} \quad \begin{array}{c} W \end{array} = \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array} \quad \begin{array}{c} Y \end{array}$$



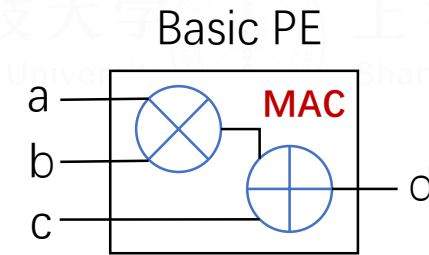
From lower-level memory



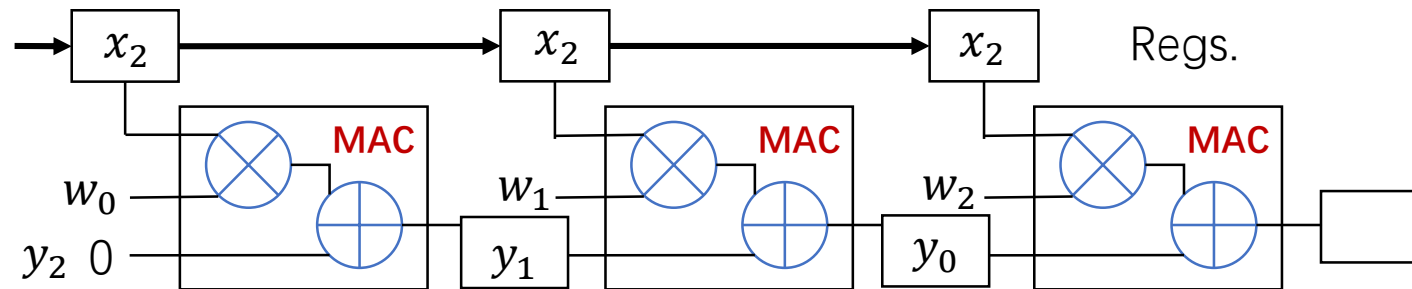
Dataflow Accelerator (Cont'd)

- 1D convolution for example

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{c} X \end{array} \quad * \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} \quad \begin{array}{c} W \end{array} = \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array} \quad \begin{array}{c} Y \end{array}$$



From lower-level memory

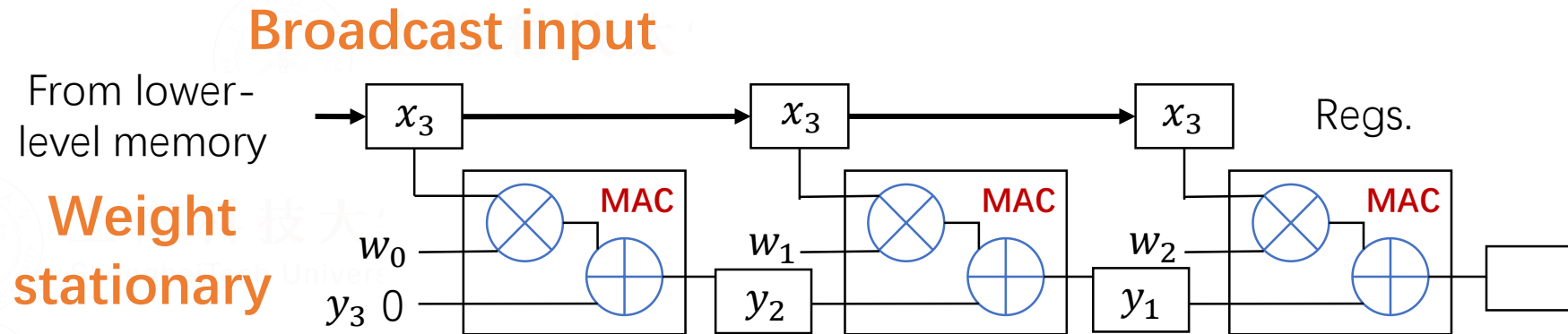
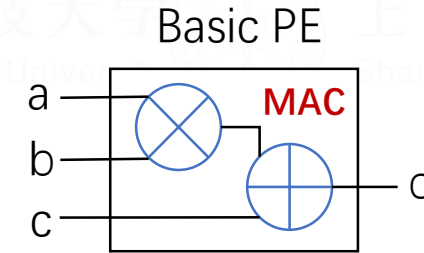


Dataflow Accelerator (Cont'd)

- 1D convolution for example

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{c} \times \\ \\ \end{array} \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array}$$

$X \qquad \qquad \qquad W \qquad \qquad \qquad Y$



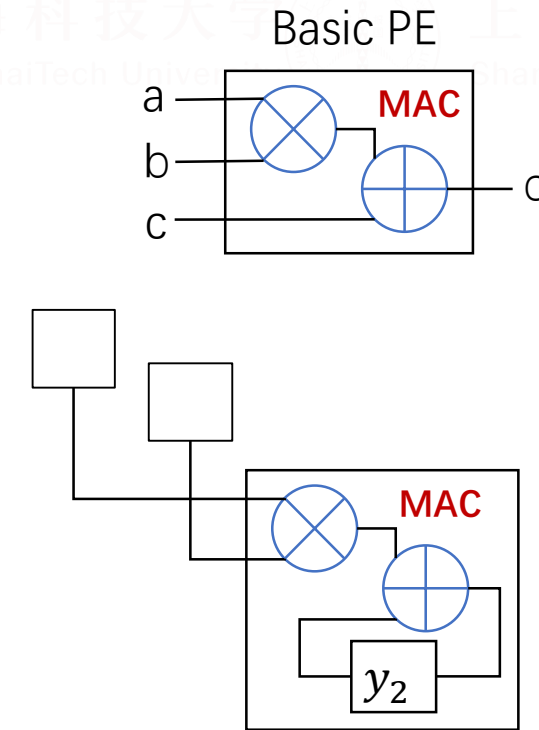
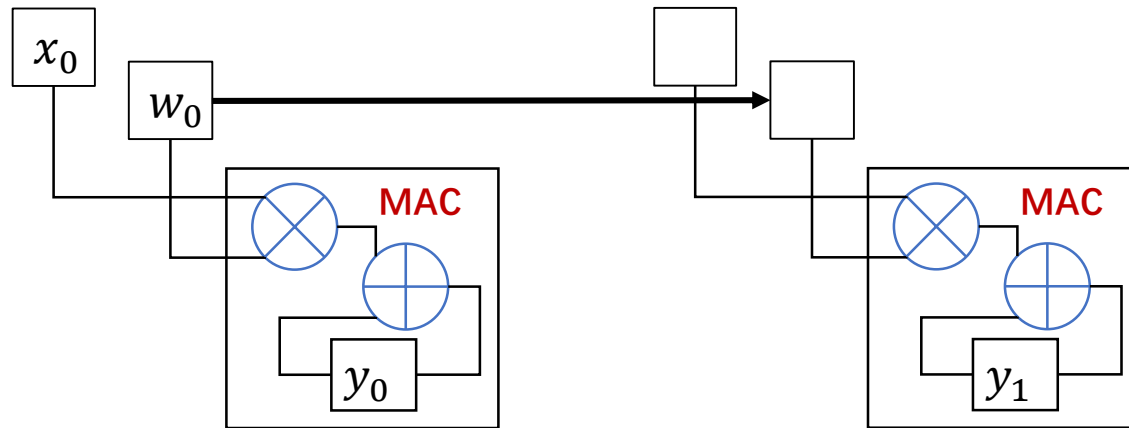
**Pipelined/Systolic
partial sum
propagation**

Dataflow Accelerator (Cont'd)

- 1D convolution for example (output-stationary)

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array}$$

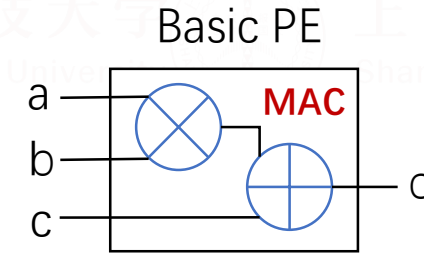
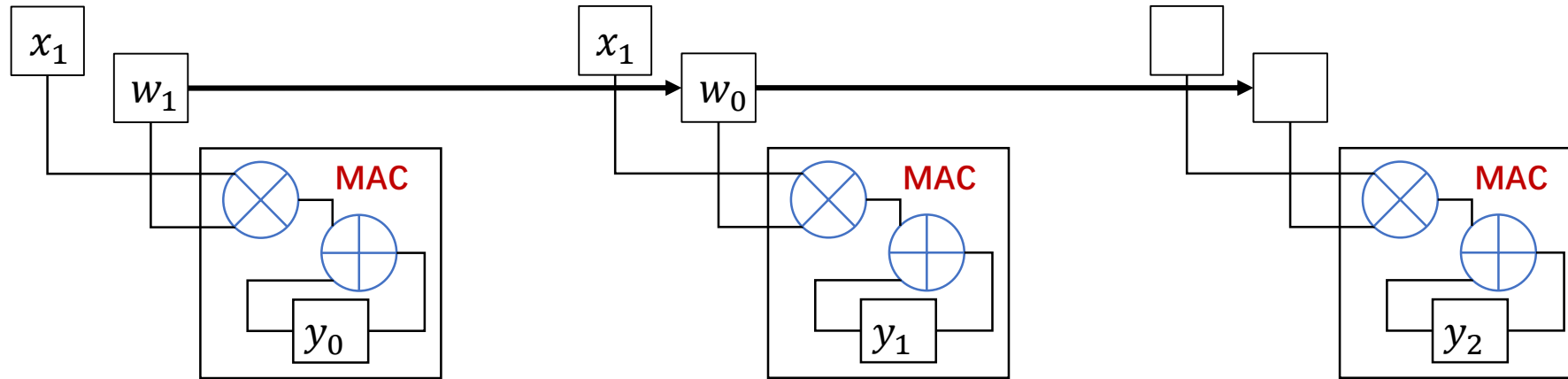
$X \qquad \qquad \qquad W \qquad \qquad \qquad Y$



Dataflow Accelerator (Cont'd)

- 1D convolution for example (output-stationary)

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{c} X \end{array} \quad * \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} \quad \begin{array}{c} W \end{array} = \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array} \quad \begin{array}{c} Y \end{array}$$

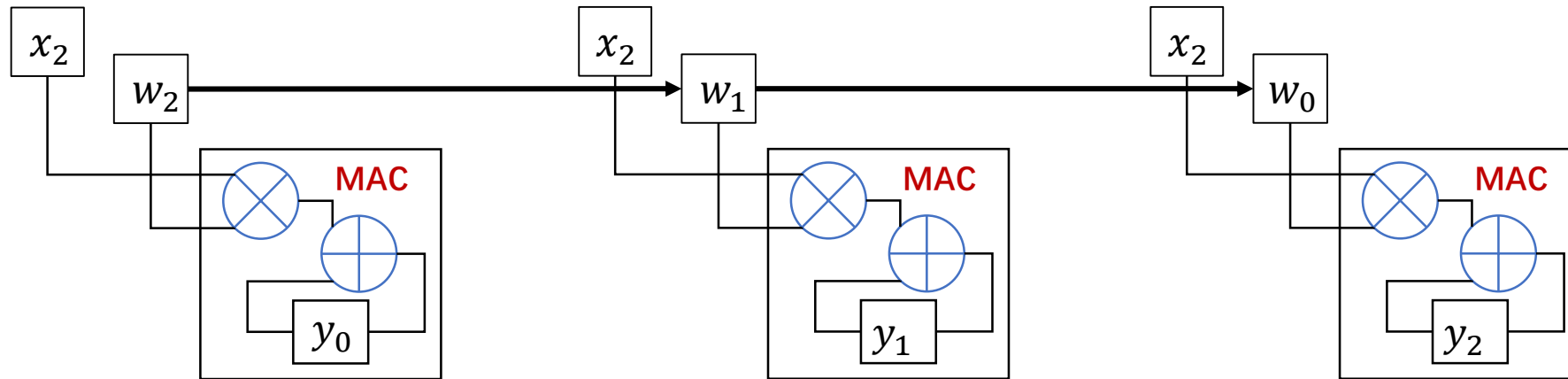
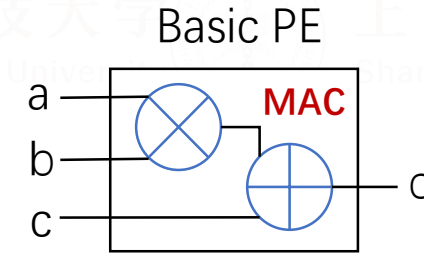


Dataflow Accelerator (Cont'd)

- 1D convolution for example (output-stationary)

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{c} \times \\ \end{array} \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array}$$

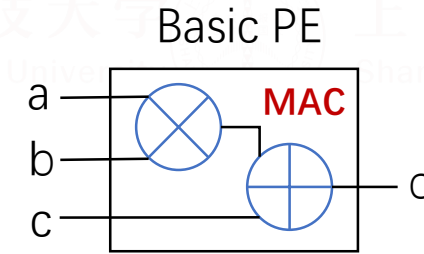
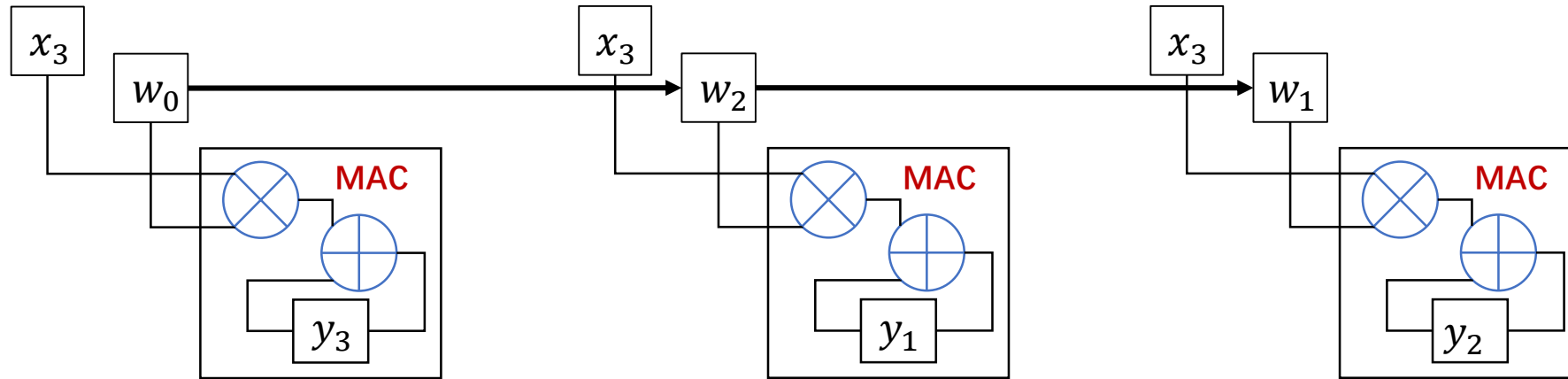
$X \qquad \qquad \qquad W \qquad \qquad \qquad Y$



Dataflow Accelerator (Cont'd)

- 1D convolution for example (output-stationary)

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{c} X \end{array} \quad * \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} \quad \begin{array}{c} W \end{array} = \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array} \quad \begin{array}{c} Y \end{array}$$

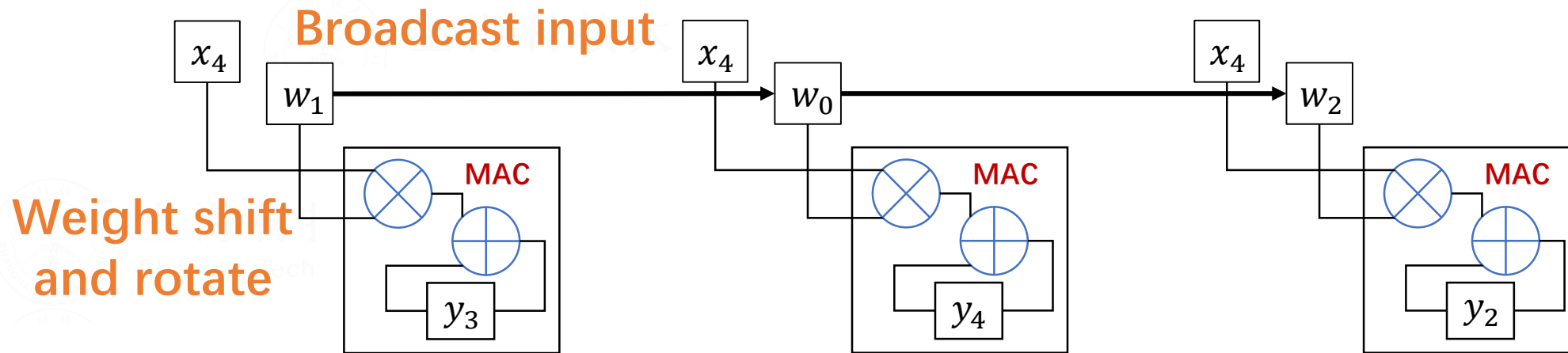
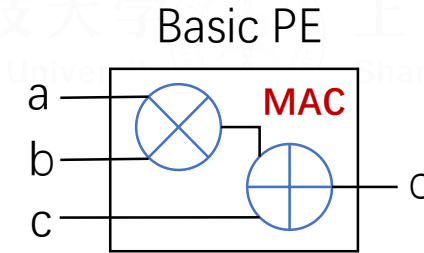


Dataflow Accelerator (Cont'd)

- 1D convolution for example (output-stationary)

$$\begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & x_2 & x_3 & \dots \\ \hline \end{array} \quad \begin{array}{c} \times \\ \end{array} \quad \begin{array}{|c|c|c|} \hline w_0 & w_1 & w_2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline y_0 & y_1 & \dots \\ \hline \end{array}$$

$X \qquad \qquad \qquad W \qquad \qquad \qquad Y$



Output stationary with overwrite

Dataflow Accelerator – 2D Conv.

- Row-stationary

x_0	x_1	x_2	x_3	x_4
x_5	x_6	x_7	x_8	x_9
x_{10}	x_{11}	x_{12}	x_{13}	x_{14}
x_{15}	x_{16}	x_{17}	x_{18}	x_{19}
x_{20}	x_{21}	x_{22}	x_{23}	x_{24}

 $*$

w_0	w_1	w_2
w_3	w_4	w_5
w_6	w_7	w_8

 $=$

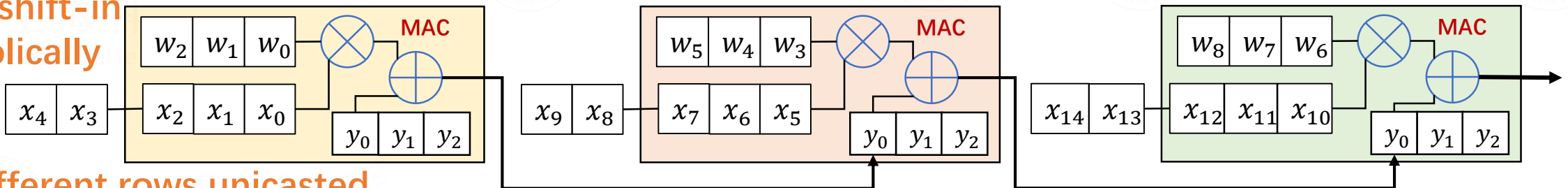
y_0	y_1	y_2
y_3	y_4	y_5
y_6	y_7	y_8

Input shift-in
systolically

Weights unicast to each PE,
and stay in each PE

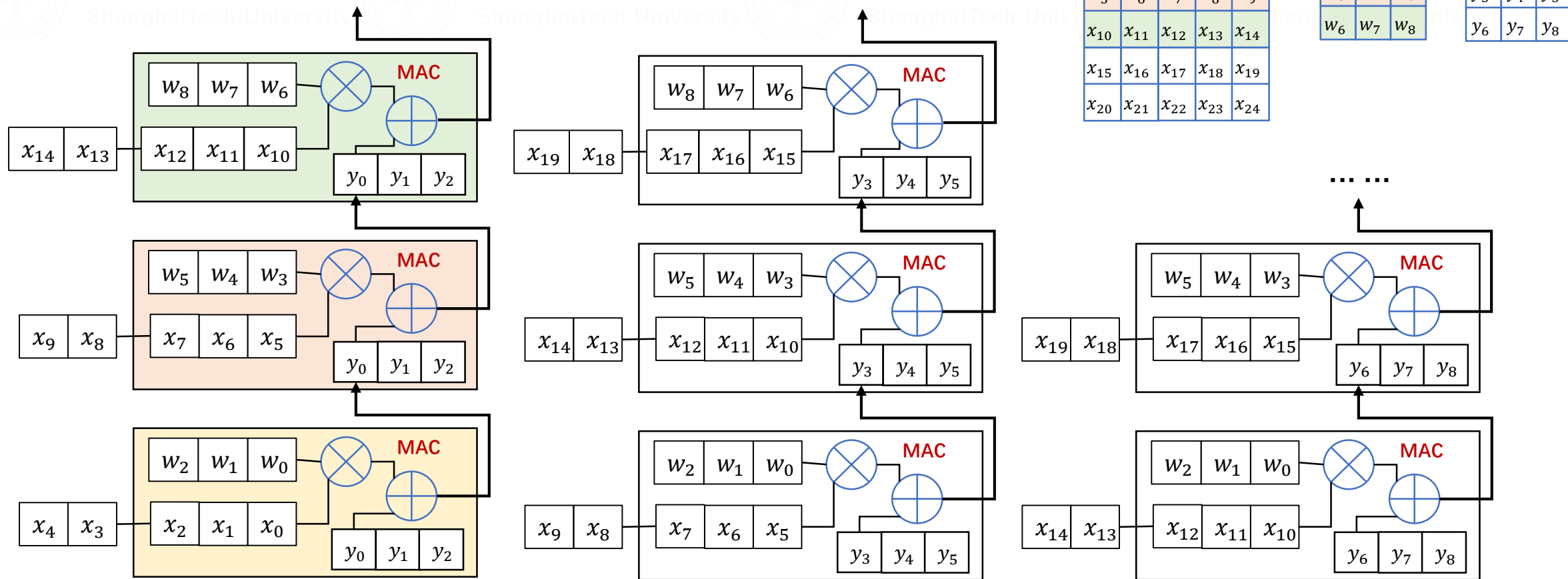
Output-stationary in each PE,
and then shift out systolically

Different rows unicast
to each PE



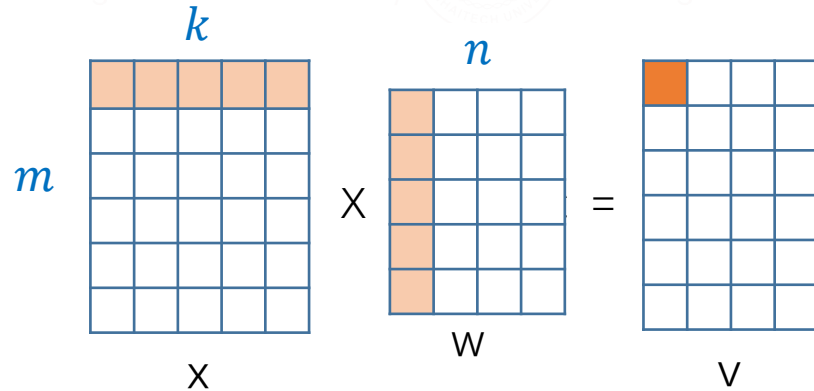
Dataflow Accelerator – 2D Conv. (Cont'd)

- Higher reuse ratio with more PE units



Problem Formulation

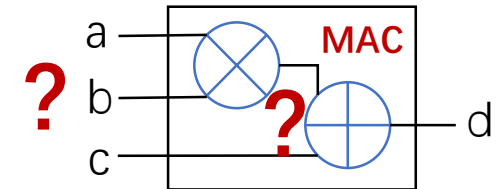
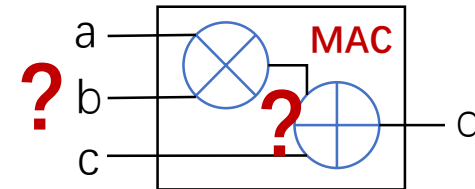
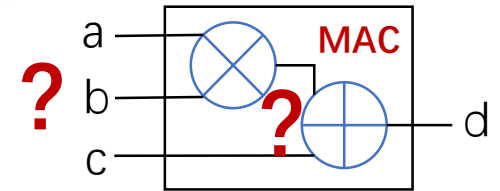
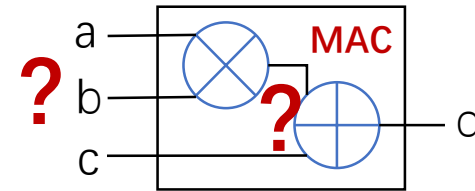
- Matrix multiplication as an example



```
for (m=0; m<M; m++) {
  for (n=0; n<N; n++){
    OA[m,n] = b[m,n];
    for (k=0; k<K; k++){
      OA[m, n] = OA[m, n] + IA[m, k]*W[k, n];
    }
  }
}
```

In each loop $l=[m,n,k]$, a multiplication and an addition are performed.

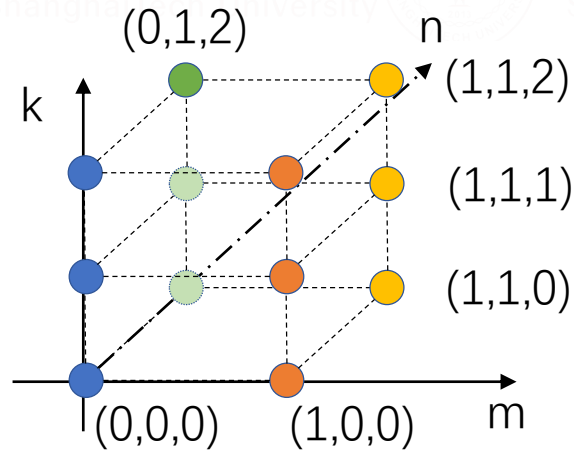
Basic PE



- In total mnk computation tasks, to map to the available PEs
- With the inputs ready at the time when the tasks are executed

Problem Formulation (Cont'd)

- Matrix multiplication as an example

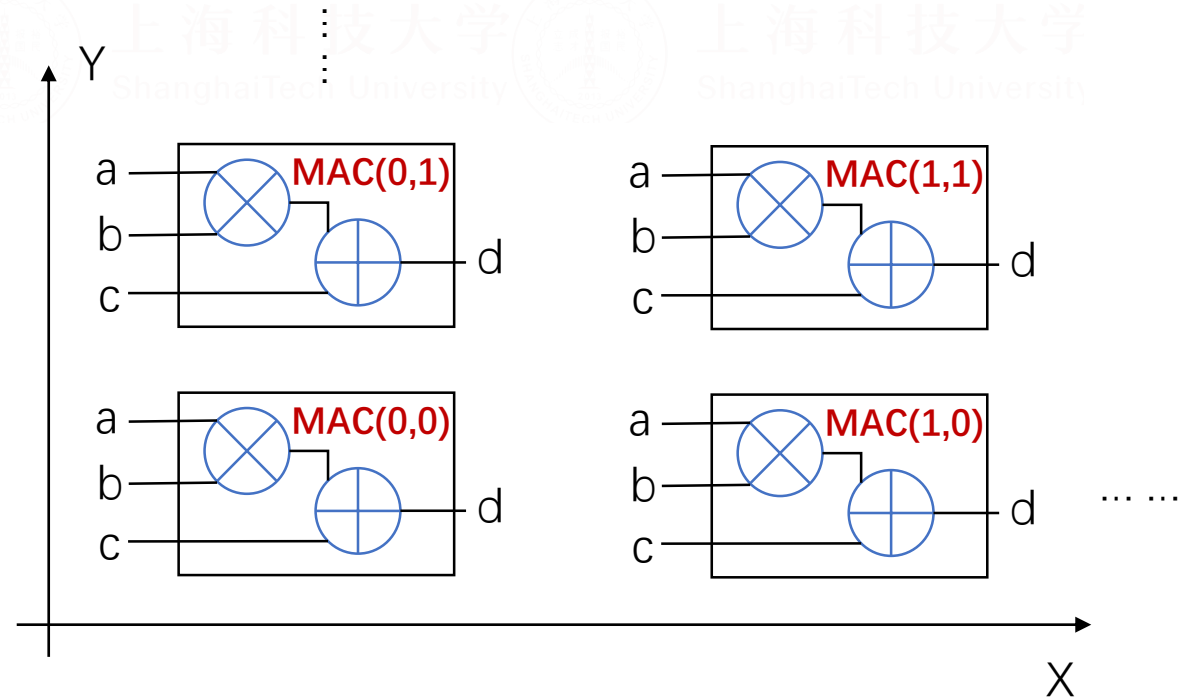


```

for (m=0; m<M; m++) {
  for (n=0; n<N; n++){
    OA[m,n] = b[m,n];
    for (k=0; k<K; k++){
      OA[m, n] = OA[m, n] + IA[m, k]*W[k, n];
    }
  }
}

```

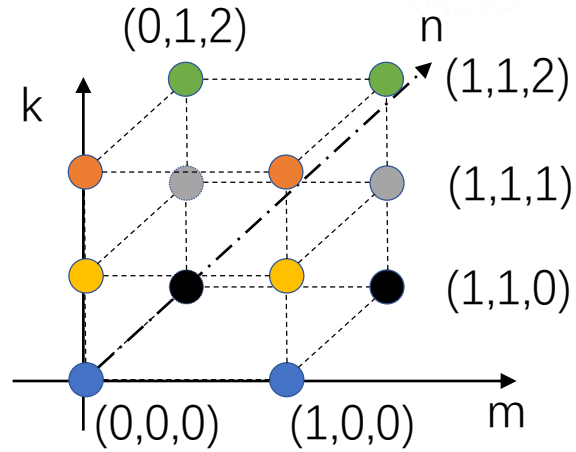
In each loop $l=[m,n,k]$,
a multiplication and an
addition are performed.



- In total $m*n*k$ computation tasks,
to map to the available PEs
- With the inputs ready at the time
when the tasks are executed

Example 1

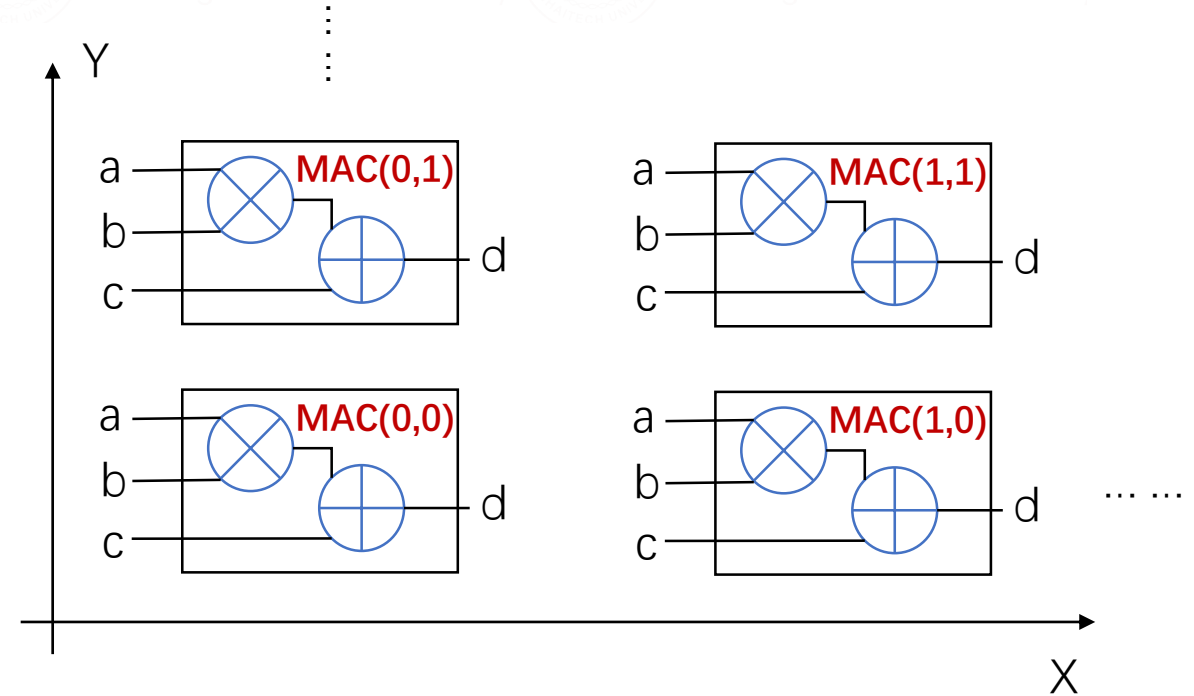
- Matrix multiplication as an example (weight-stationary)



Dependence matrix:

Projection matrix:

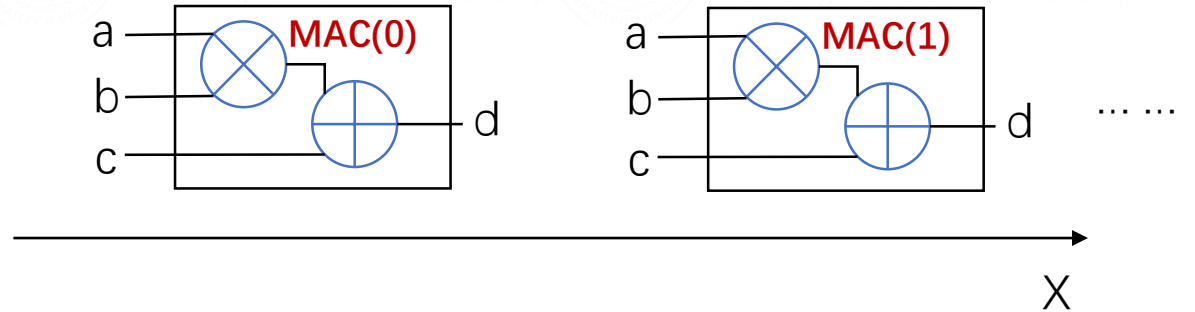
Scheduling matrix:



Example 2

- 1-D Conv.

```
for n {  
  for k {  
     $y[n] = y[n] + x[n-k] * w[k];$   
  }  
}
```



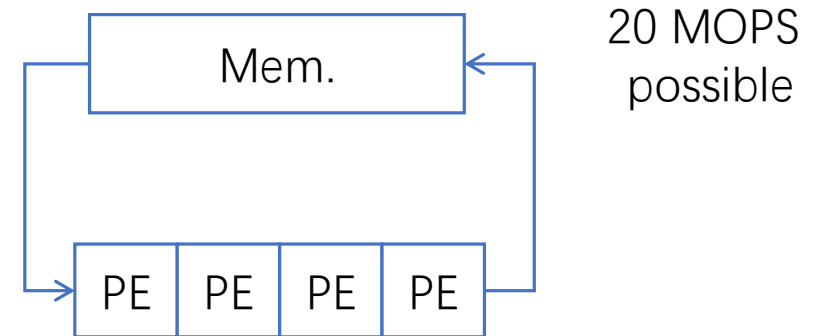
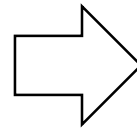
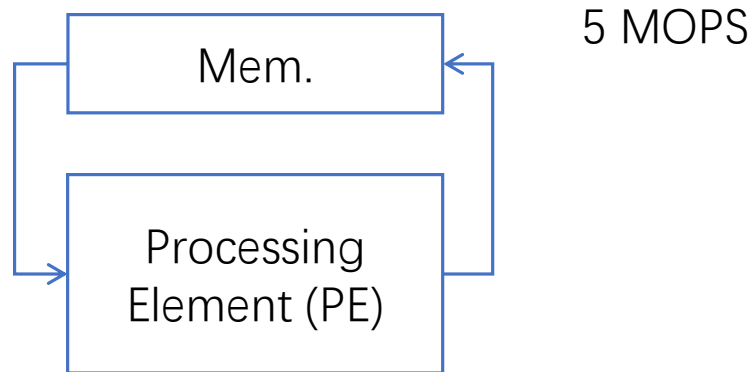
Dependence matrix:

Projection matrix:

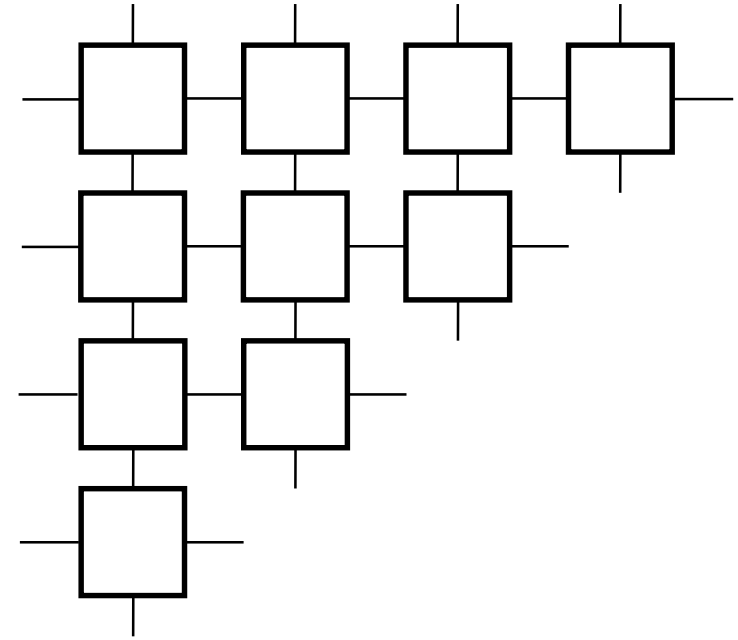
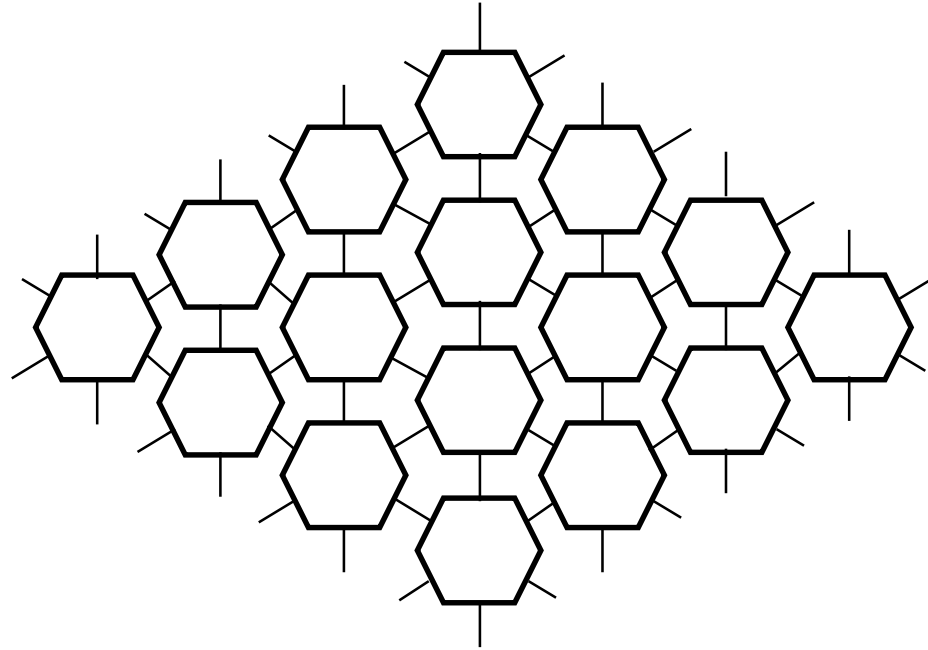
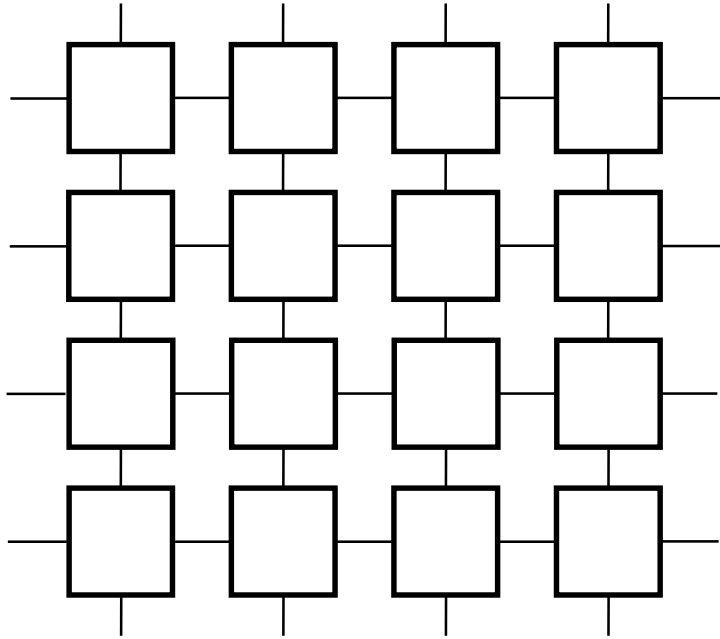
Scheduling matrix:

Systolic Array

- Bandwidth $\propto$ perimeter vs. Computation $\propto$ area
- Compute-bound vs. bandwidth-bound

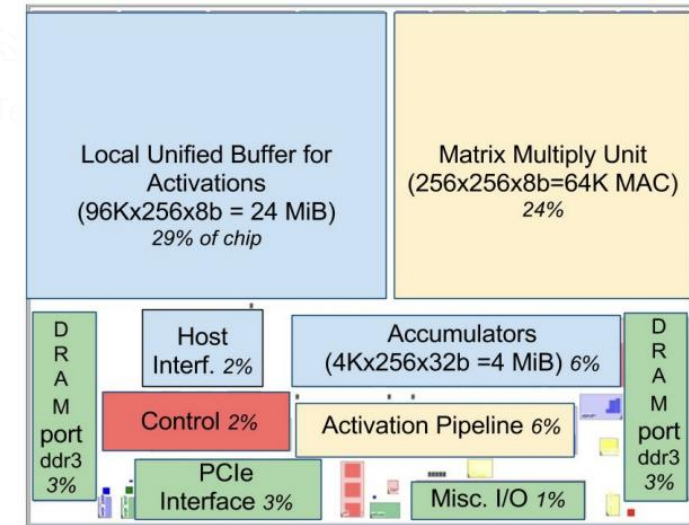
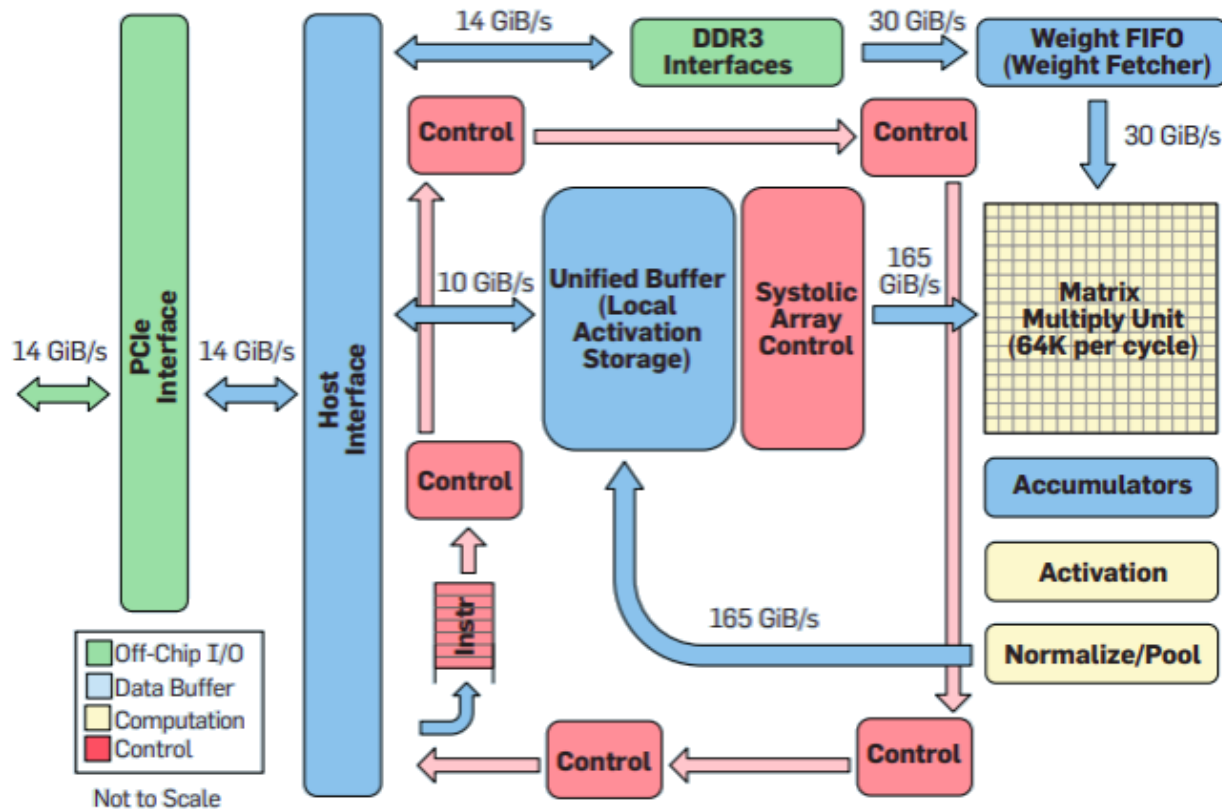


Systolic Array with Various Topologies



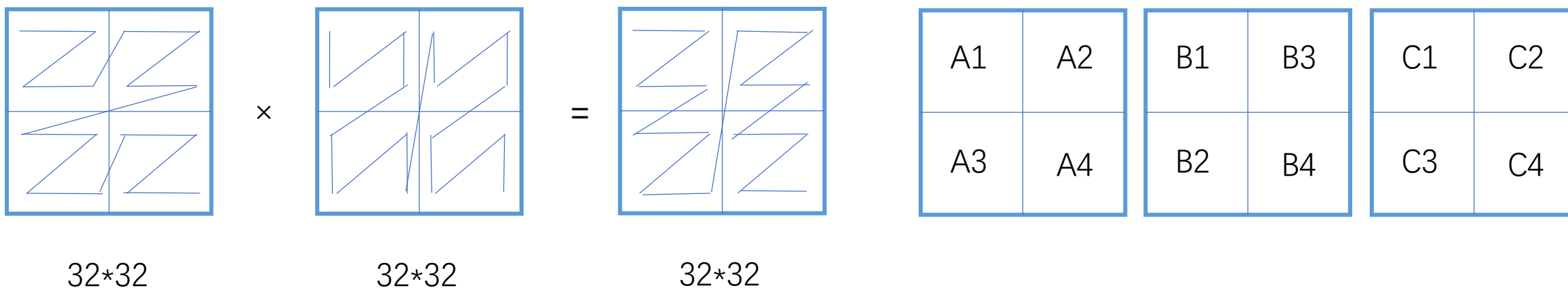
Systolic Array – Design Case

- Google TPU v1



- Matrix unit produces one 256-element partial sum per clock cycle
- Matrix unit holds one 64 KiB tile of weights plus one for double-buffering (to hide the 256 cycles it takes to shift a tile in)
- 2048*2 accumulator units to match MMU (to deal to Wmatrix larger than 256*256)

Tiling



$$C1 = A1 * B1 + A2 * B2$$

$$C2 = A1 * B3 + A2 * B4$$

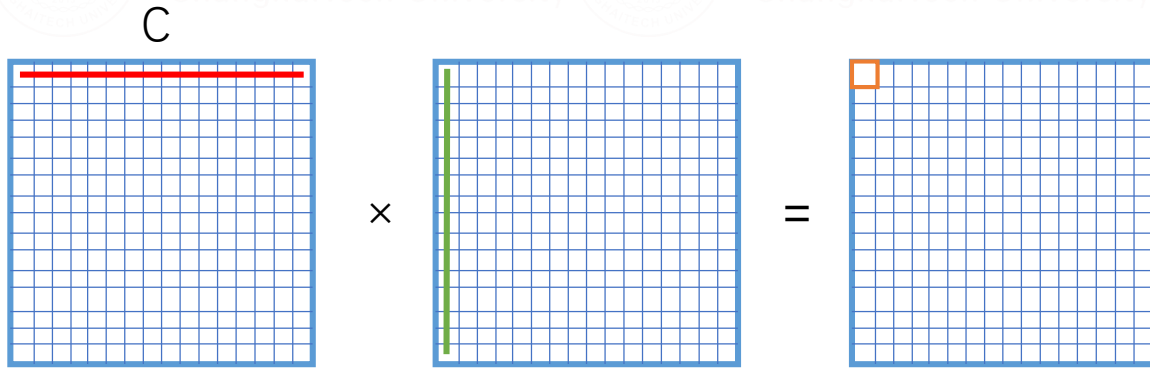
$$C3 =$$

$$C4 =$$

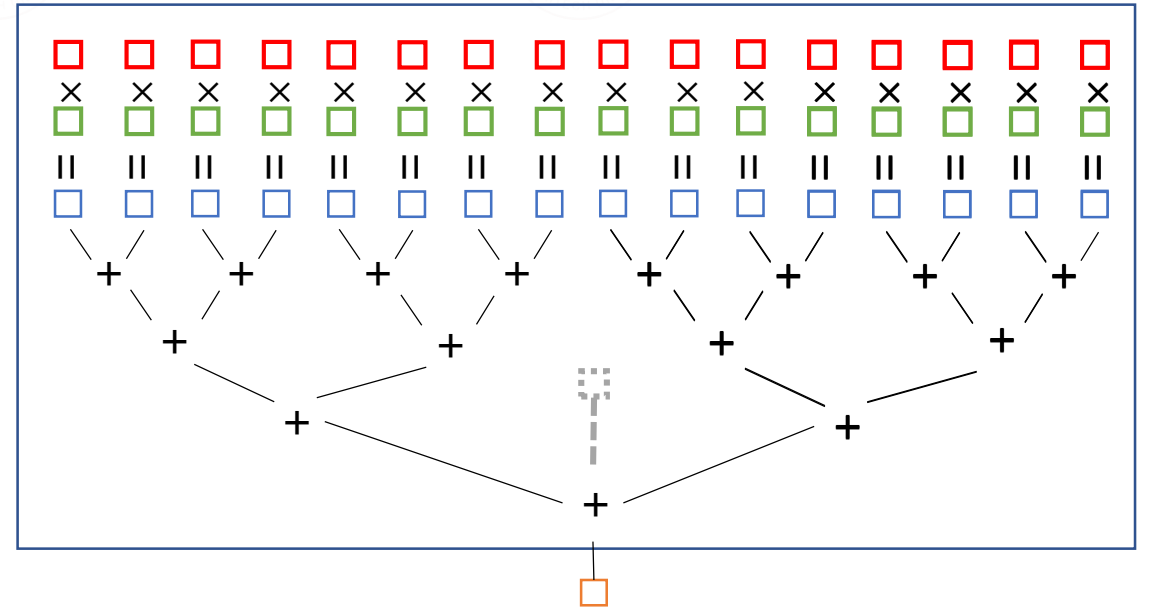
- Extra accumulators required
- Need to decide the dataflow

Adder Tree-based Design

- Multiply-and-add/accumulate by **adder tree**



- Only deal with fixed C (16)
- What if $C < 16$?
- What if $C > 16$? **Tiling!**



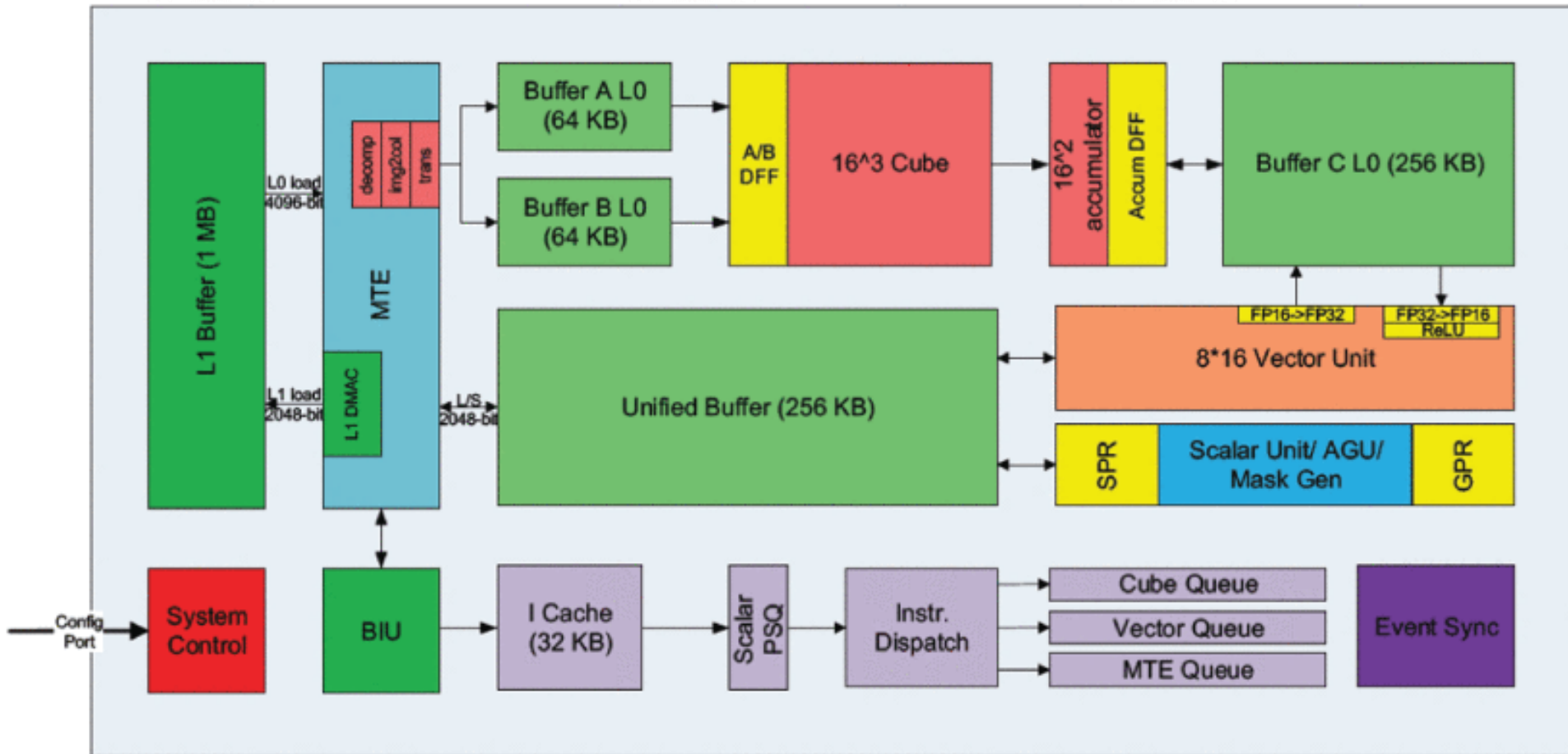
16 element-wise multiply-and-add

Memory access

PPA

Adder Tree – Design Case

- AI core inside Ascend AI processor (DaVinci architecture)



H. Liao et al., "Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper," 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2021, pp. 789-801.

Summary

- Various methods to implement GEMM and Conv.
- They have different memory access patterns/requirements

