Designing CNN Accelerators
Day 3

Hyoukjun Kwon
(hyoukjun@gatech.edu)

Georgia Institute of Technology
Synergy Lab (http://synergy.ece.gatech.edu)

@SNU

Dec 28, 2017

Day 3 Agenda

- **CNN Accelerator Dataflow**
  - Understanding Dataflow
  - Weight-Stationary
  - Row-Stationary
- **Network-on-Chip**
  - Topology
  - NoC Topology for DNN Accelerators
- **Processing Element**
  - Structure Overview
  - Processing Element Array

Revisiting: Convolutional Layer Computation

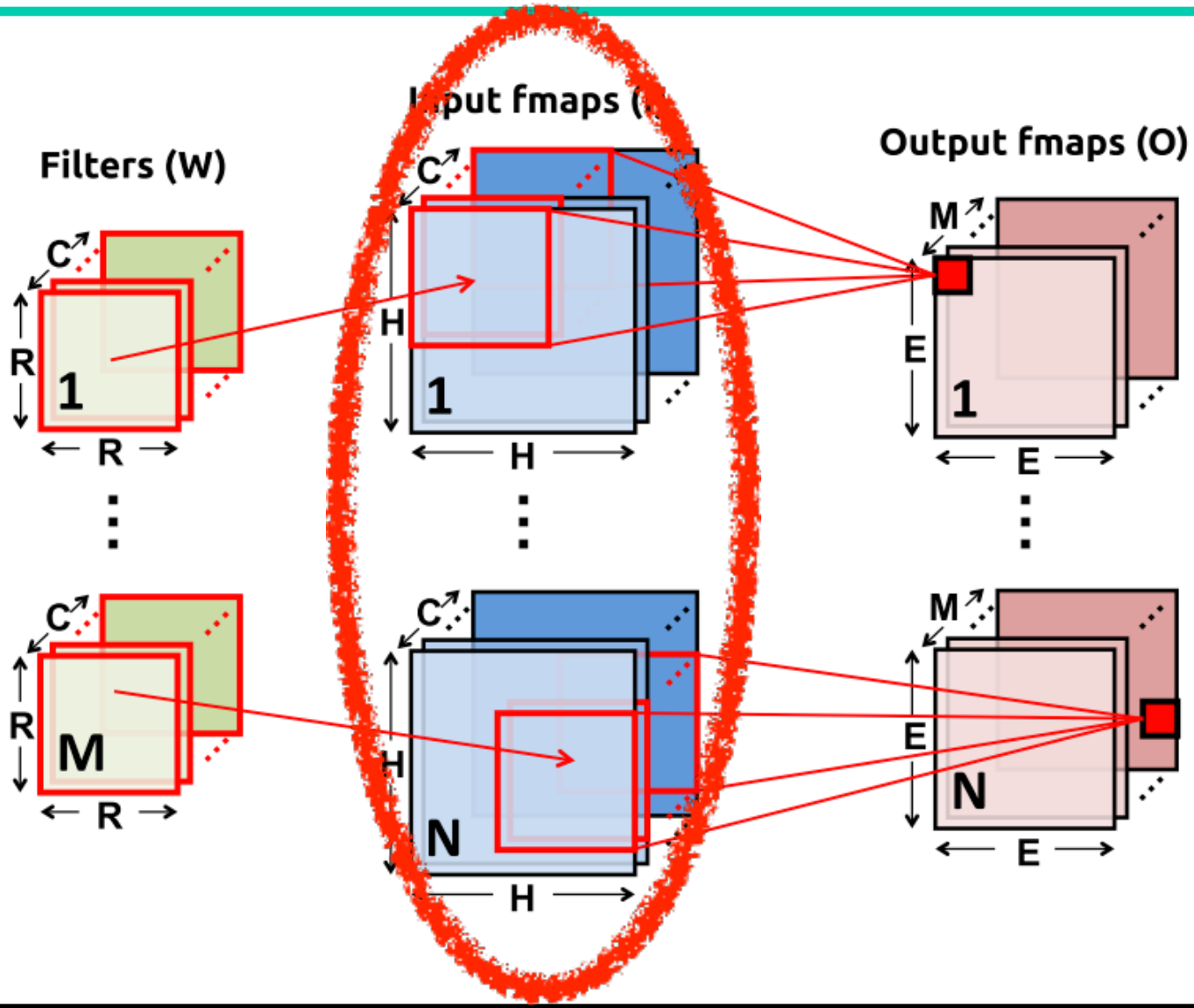
```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) {  // Input feature map row
        for(x=0; x<H; x++) {  // Input feature map column
          for(j=0; j<R; j++) {  // Weight filter row
            for(i=0; i<R; i++) {  // Weight filter column
              O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```

Accumulation **Multiplication**

Analysis: Conv Layer Computation

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) {  // Input feature map row
        for(x=0; x<H; x++) {  // Input feature map column
          for(j=0; j<R; j++) {  // Weight filter row
            for(i=0; i<R; i++) {  // Weight filter column
              O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```

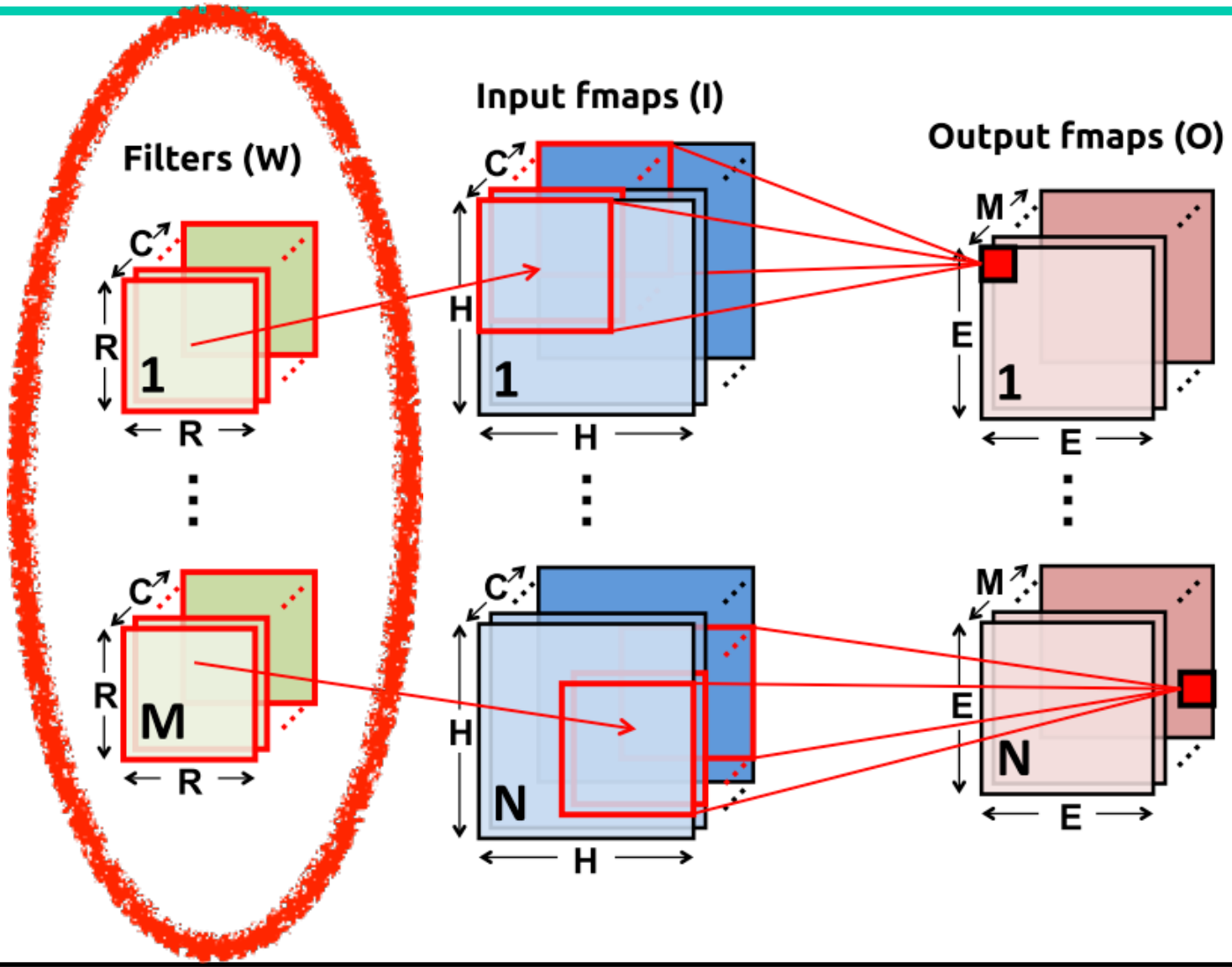
Analysis: Conv Layer Computation



Analysis: Conv Layer Computation

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) {  // Input feature map row
        for(x=0; x<H; x++) {  // Input feature map column
          for(j=0; j<R; j++) {  // Weight filter row
            for(i=0; i<R; i++) {  // Weight filter column
              O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```

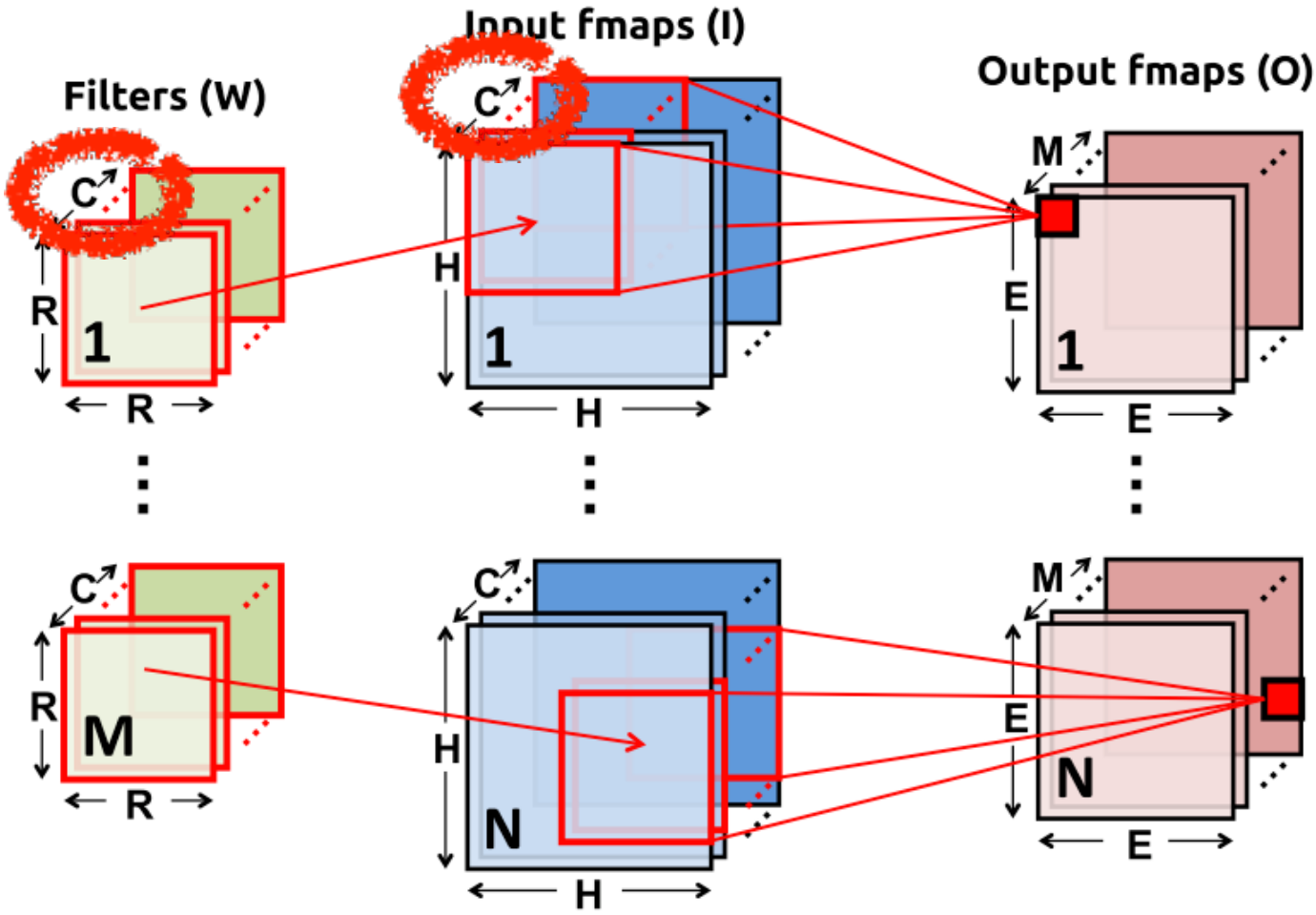
Analysis: Conv Layer Computation



Analysis: Conv Layer Computation

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) { // Input feature map row
        for(x=0; x<H; x++) { // Input feature map column
          for(j=0; j<R; j++) { // Weight filter row
            for(i=0; i<R; i++) { // Weight filter column
              O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```

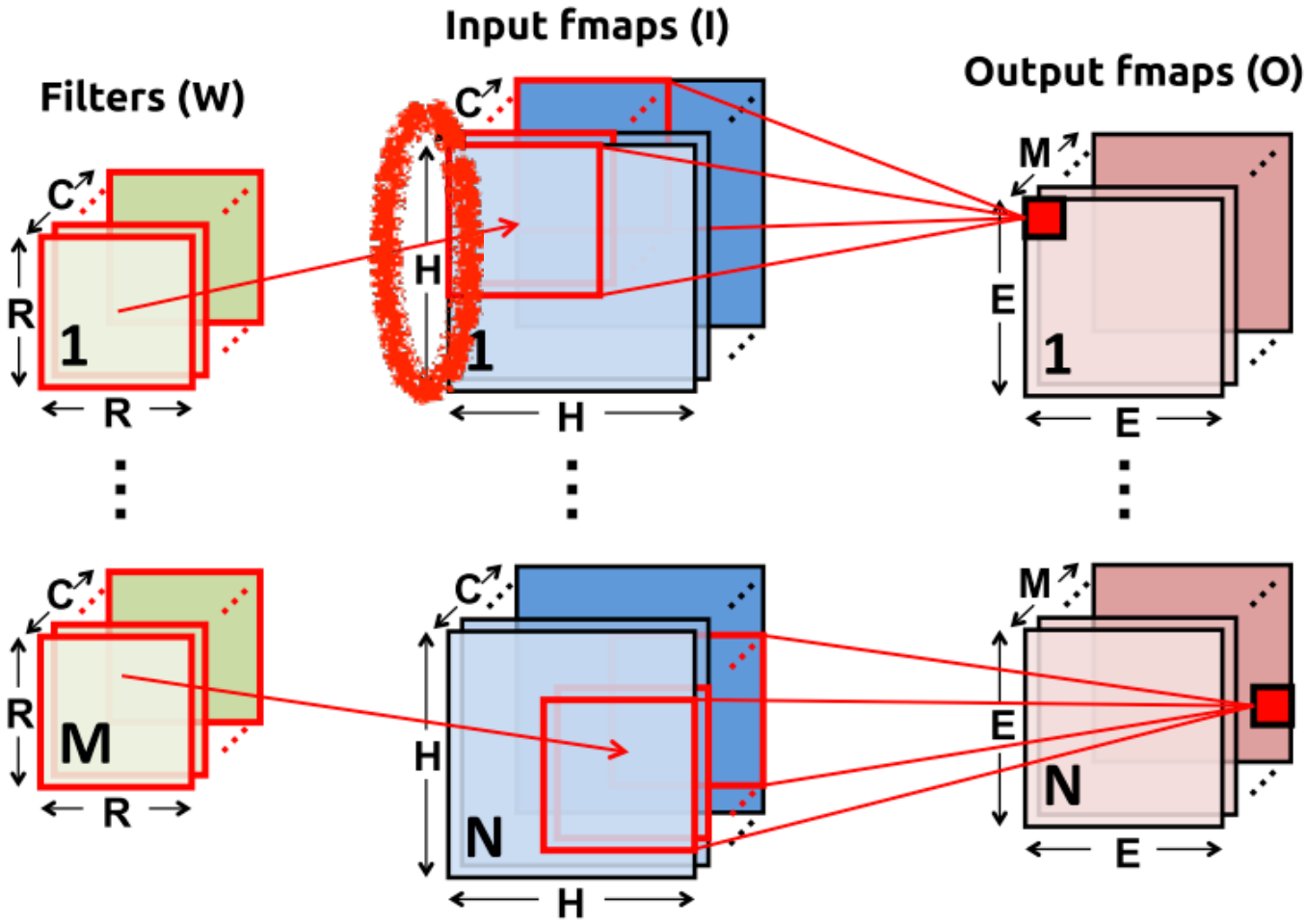
Analysis: Conv Layer Computation



Analysis: Conv Layer Computation

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) {  // Input feature map row
        for(x=0; x<H; x++) {  // Input feature map column
          for(j=0; j<R; j++) {  // Weight filter row
            for(i=0; i<R; i++) {  // Weight filter column
              O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```

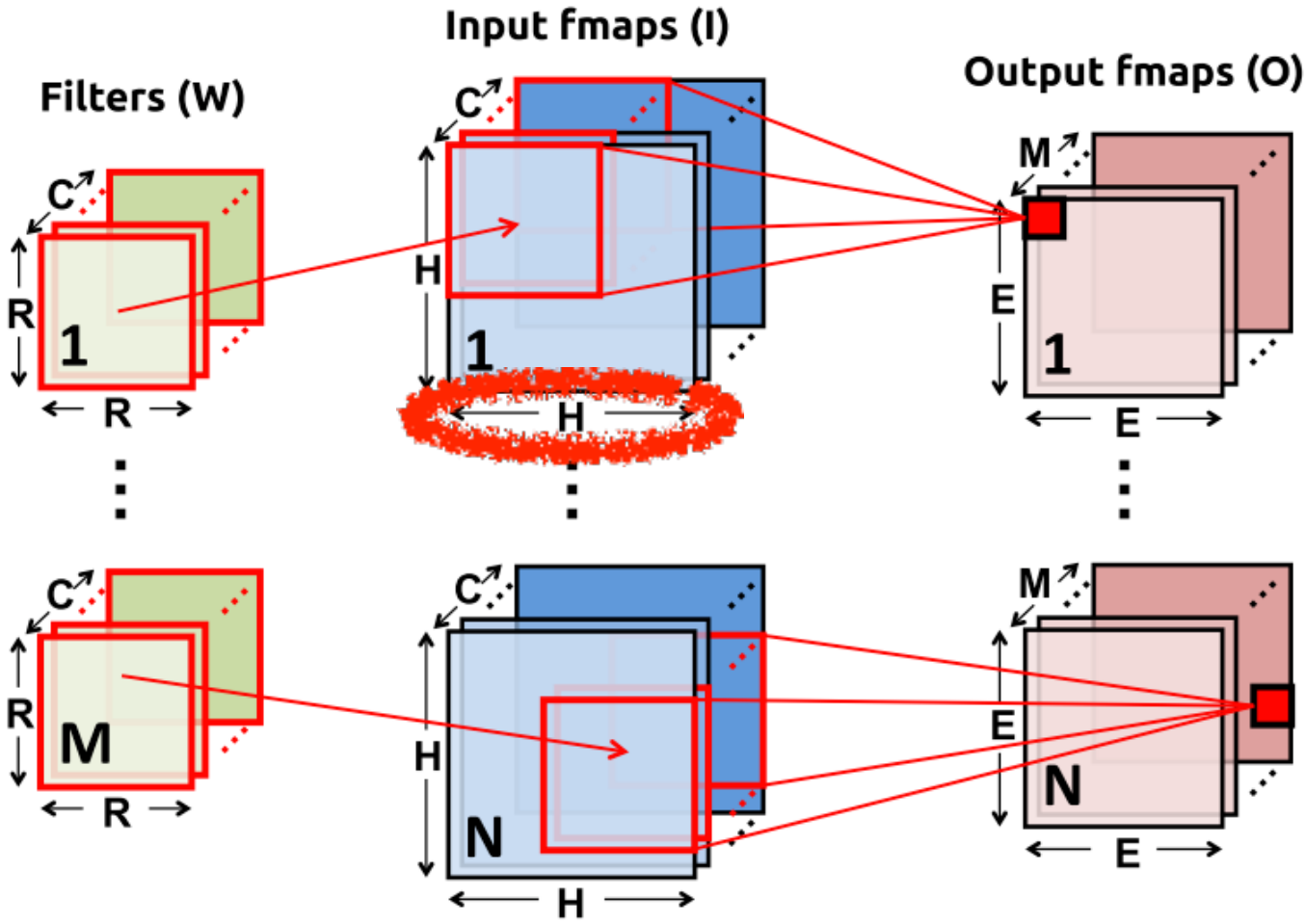
Analysis: Conv Layer Computation



Analysis: Conv Layer Computation

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) { // Input feature map row
        for(x=0; x<H; x++) { // Input feature map column
          for(j=0; j<R; j++) { // Weight filter row
            for(i=0; i<R; i++) { // Weight filter column
              O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```

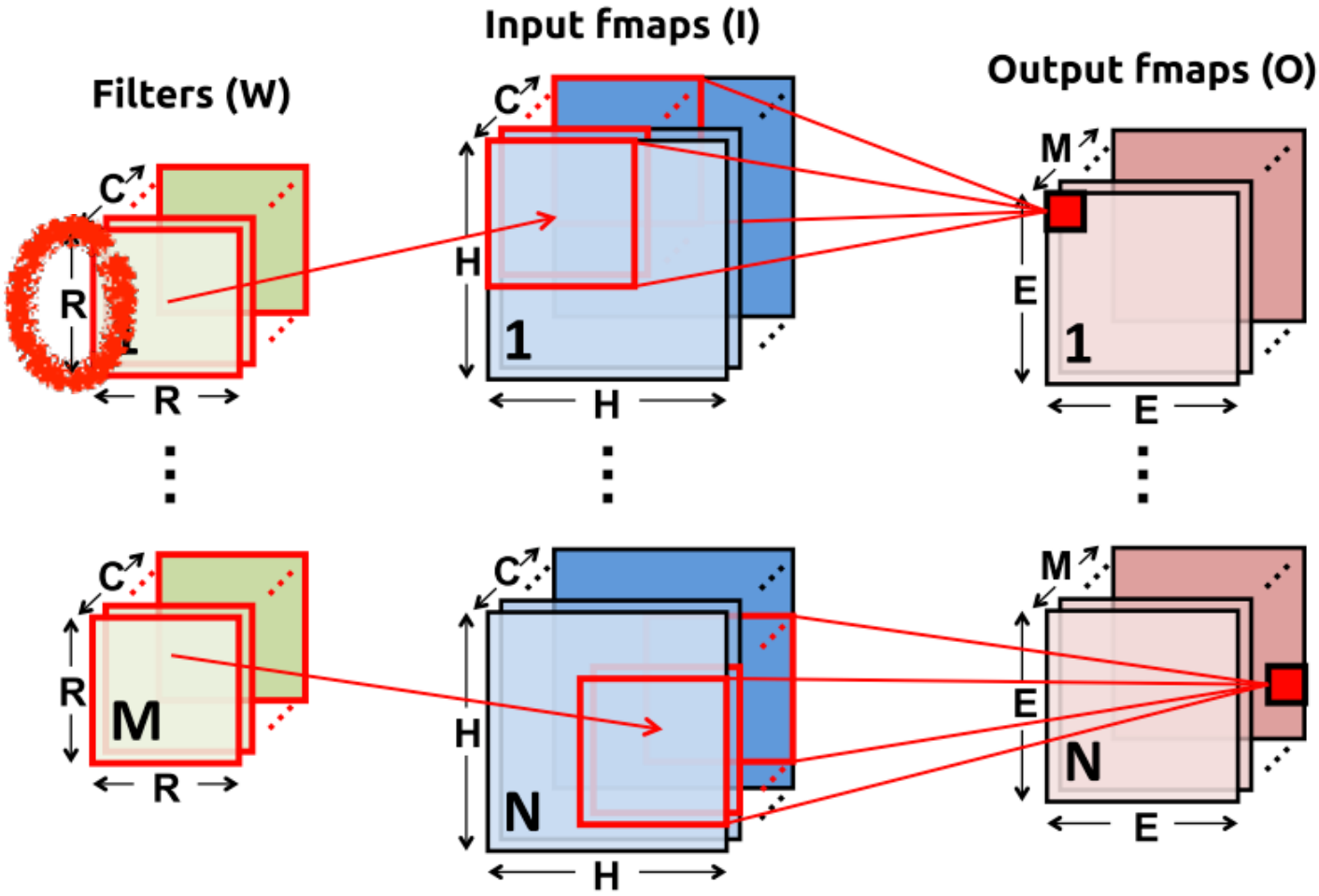
Analysis: Conv Layer Computation



Analysis: Conv Layer Computation

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) {  // Input feature map row
        for(x=0; x<H; x++) {  // Input feature map column
          for(j=0; j<R; j++) {  // Weight filter row
            for(i=0; i<R; i++) {  // Weight filter column
              O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```

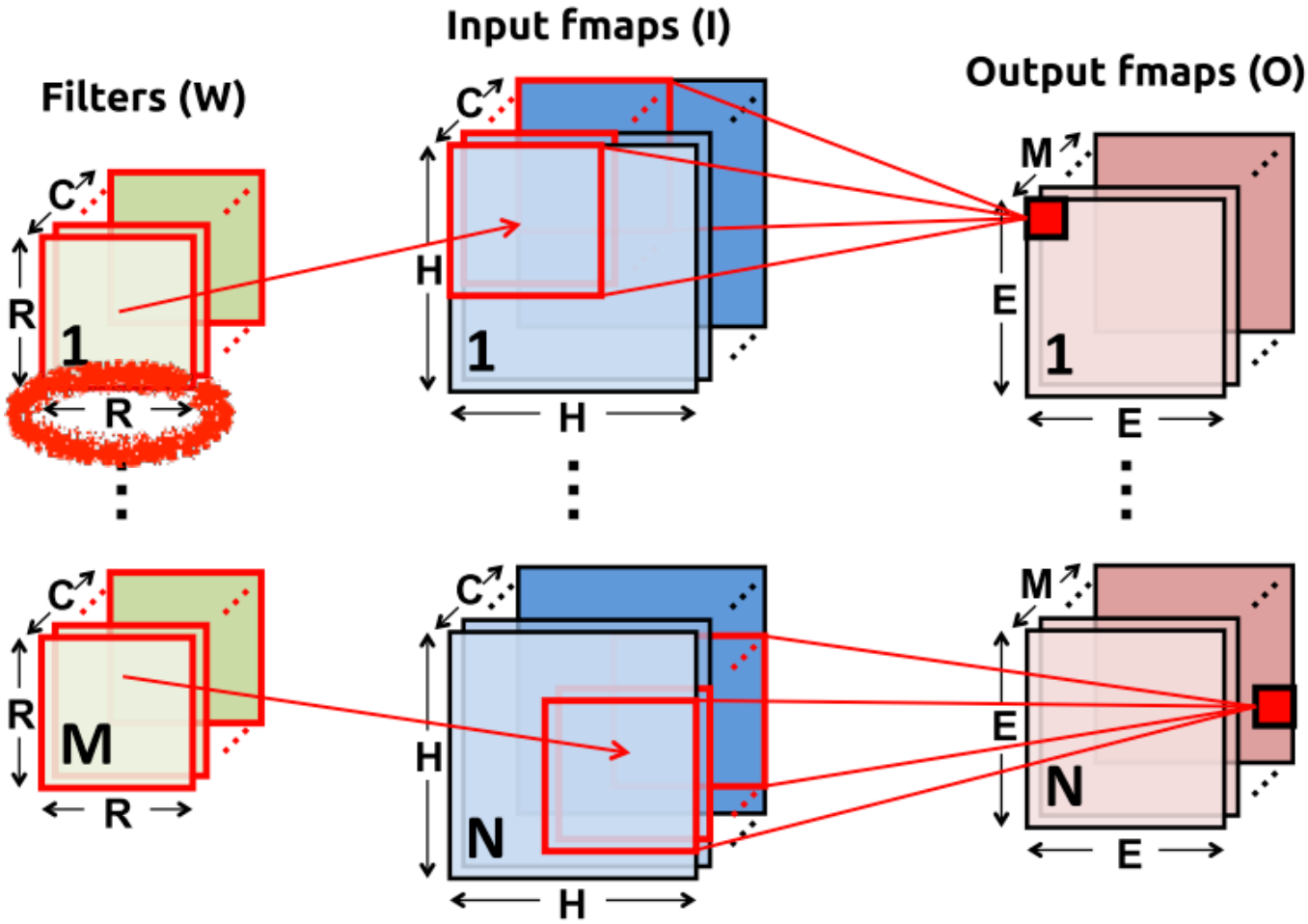
Analysis: Conv Layer Computation



Analysis: Conv Layer Computation

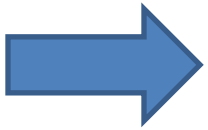
```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) {  // Input feature map row
        for(x=0; x<H; x++) {  // Input feature map column
          for(j=0; j<R; j++) {  // Weight filter row
            for(i=0; i<R; i++) {  // Weight filter column
              O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```

Analysis: Conv Layer Computation



Analysis: Conv Layer Computation

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) {  // Input feature map row
        for(x=0; x<H; x++) {  // Input feature map column
          for(j=0; j<R; j++) {  // Weight filter row
            for(i=0; i<R; i++) {  // Weight filter column
              O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```



6D Convolution (if process over one image)

Understanding Dataflow

- **Convolutional Layer Dataflow**
  - Data communication pattern based on computation order of convolutions
  - Significantly affects the performance and energy efficiency of a CNN acclerator

- **Elements of Dataflow**
  - Iteration order: Changing for-loop order
  - Blocking: Splitting a loop into multiple chunks
  - Data reuse strategy: Data reuse pattern

Elements of Dataflow

- **Iteration Order**

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) {  // Input feature map row
        for(x=0; x<H; x++) {  // Input feature map column
          for(j=0; j<R; j++) {  // Weight filter row
            for(i=0; i<R; i++) {  // Weight filter column
              O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```

No dependence between outputs

Elements of Dataflow

- **Blocking**

```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) {  // Input feature map row
        for(x=0; x<H; x++) {  // Input feature map column
          for(j=0; j<R; j++) {  // Weight filter row
            for(i=0; i<R; i++) {  // Weight filter column
              O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```

Elements of Dataflow

- **Blocking**

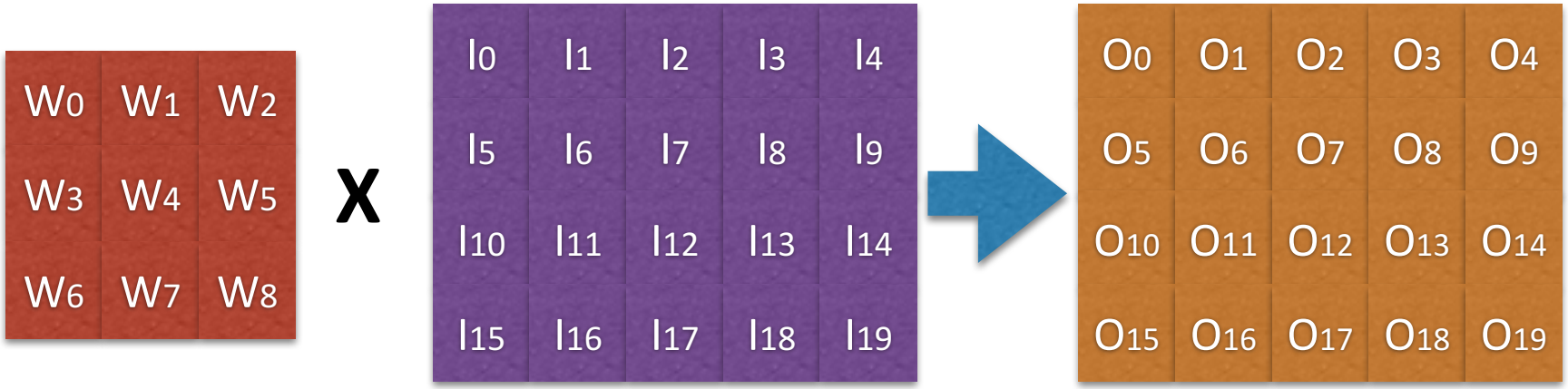
```
for(n=0; n<N; n++) { // Input feature maps (IFMaps)
  for(m=0; m<M; m++) { // Weight Filters
    for(c=0; c<C; c++) { // IFMap/Weight Channels
      for(y=0; y<H; y++) {  // Input feature map row
        for(x=0; x<H; x++) {  // Input feature map column
          for(blk=0; blk<R/BlkSz; blk++) {  // Block
            for(j=blk*BlkSz; j<BlkSz; j++) {// Weight filter row
              for(i=0; i<R; i++) {  // Weight filter column
                O[n][m][x][y] += W[m][c][i][j] * I[n][c][y][x]}}}}}}}
```

Why split a loop into blocks?

Elements of Dataflow

- **Data Reuse Strategy**
  - **Weight Stationary (WS):**
    - Try to maximize weight reuse
  - **Output Stationary (OS)**
    - Try to maximize partial sum (Psum) reuse
  - **Row Stationary (RS)**
    - Try to maximize inter-PE weight/IFMap/Psum reuse
  - **No Local Reuse (Global Reuse)**

Data Reuse Strategy Example - WS



Weight Filter

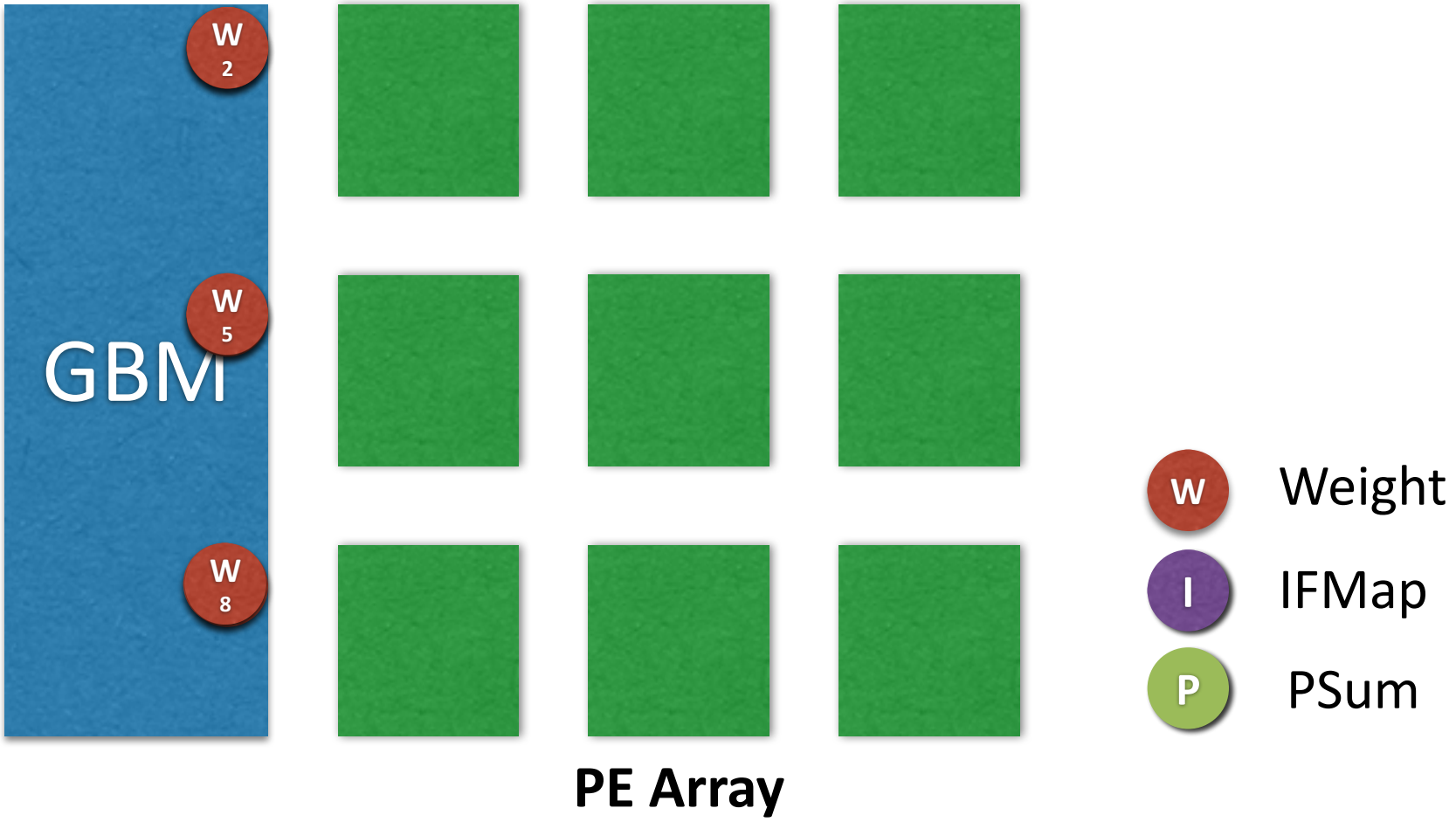
Input Feature Map

Output Feature Map

For simplicity, assume only one channel and one weight filter

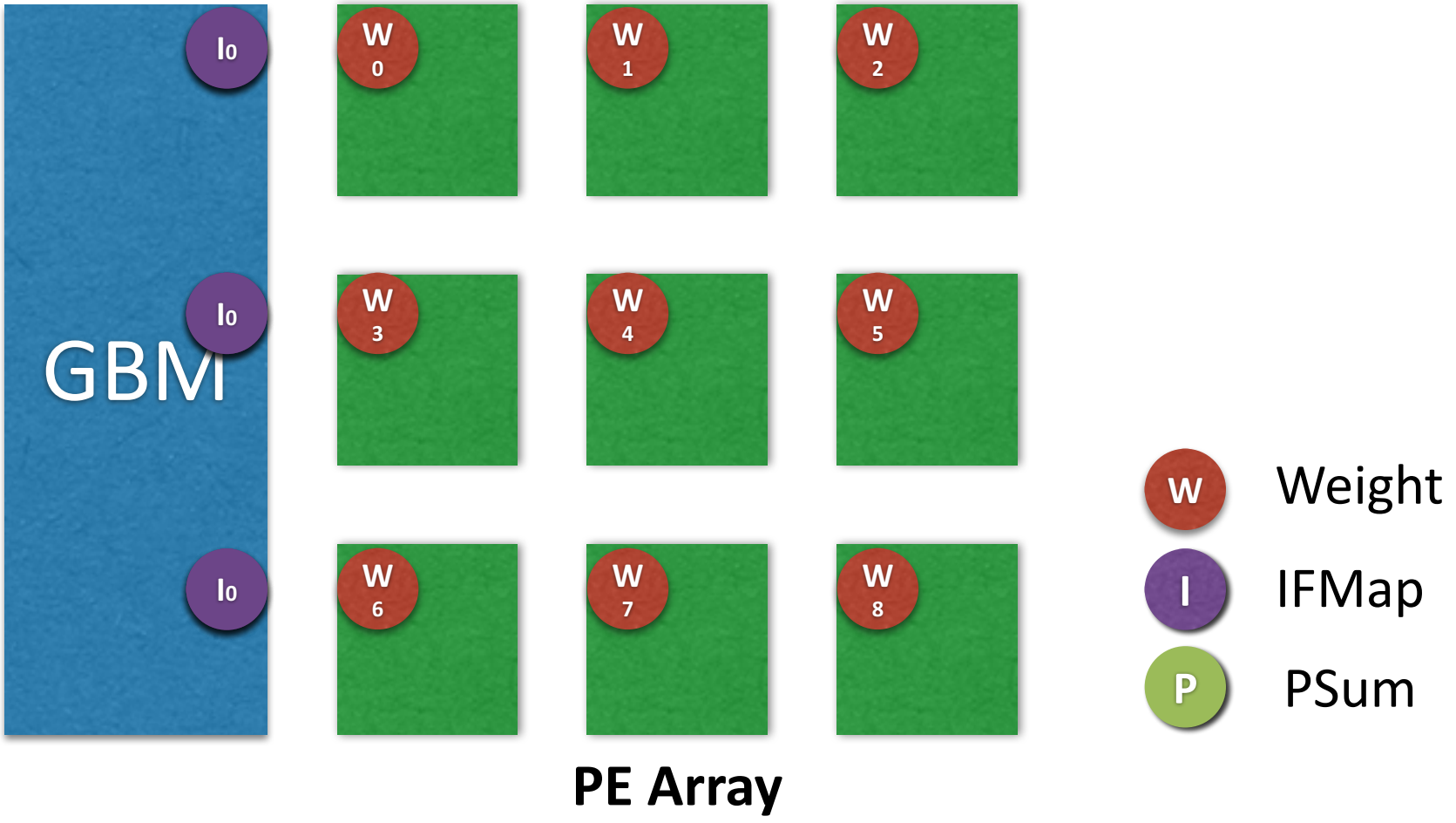
Data Reuse Strategy Example - WS

1. Weight Distribution



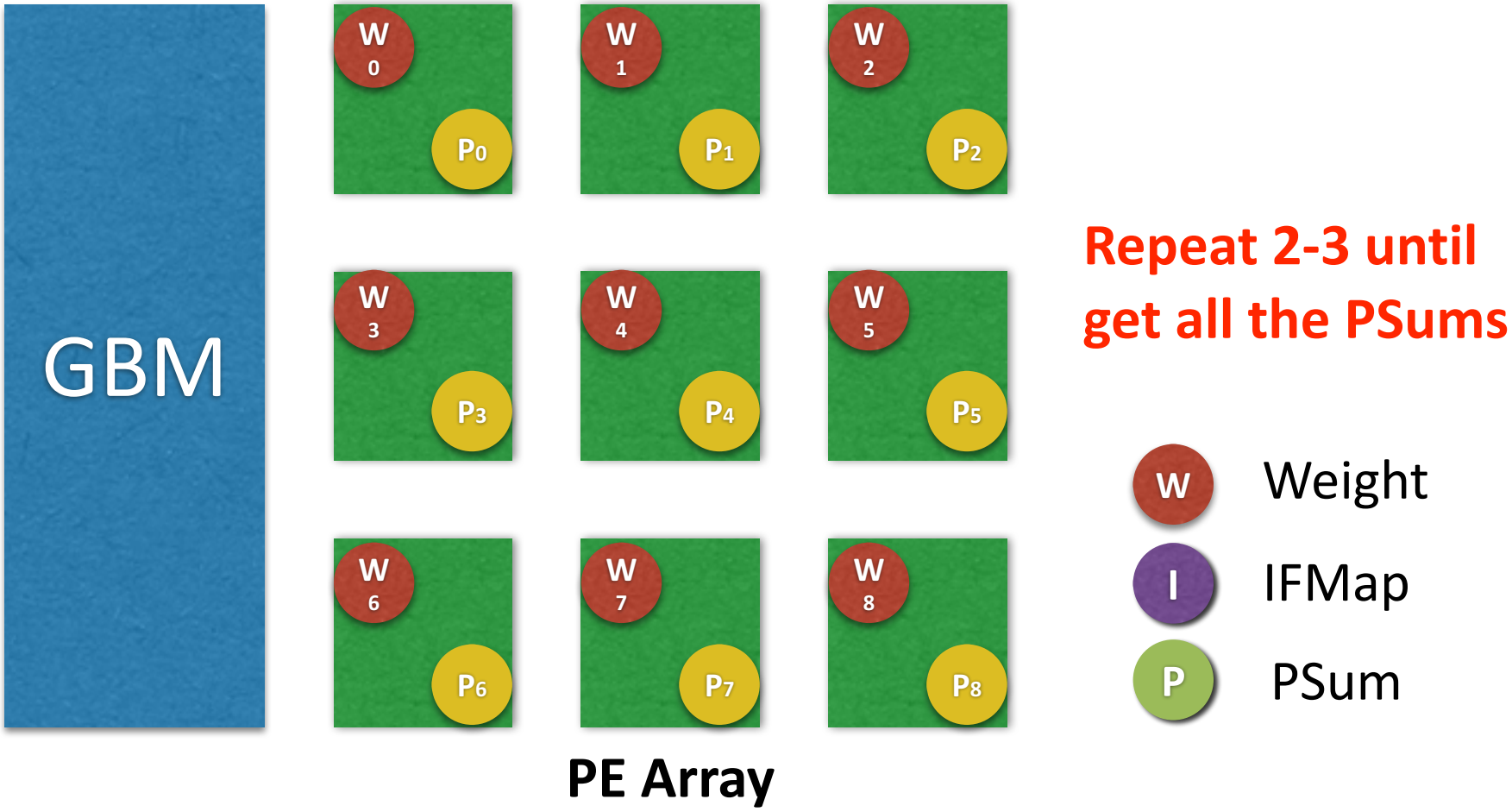
Data Reuse Strategy Example - WS

2. IFMap Distribution

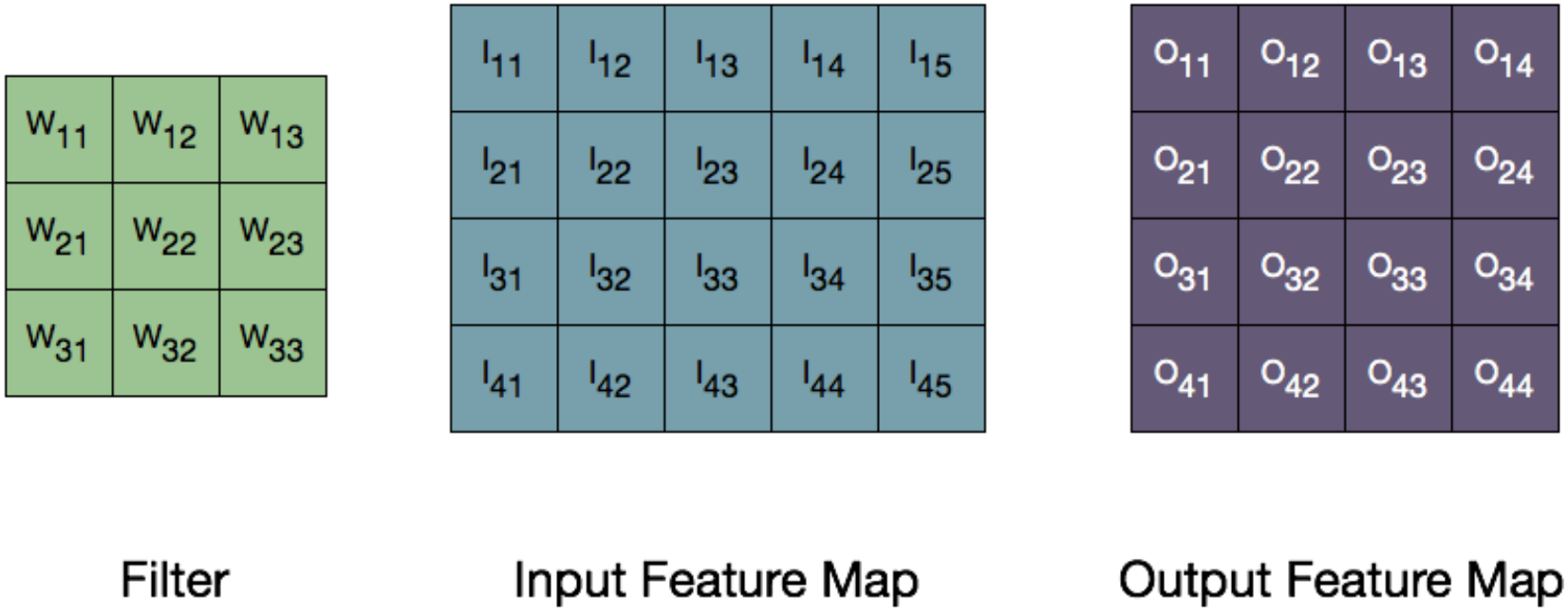


Data Reuse Strategy Example - WS

3. Partial Sum Gather



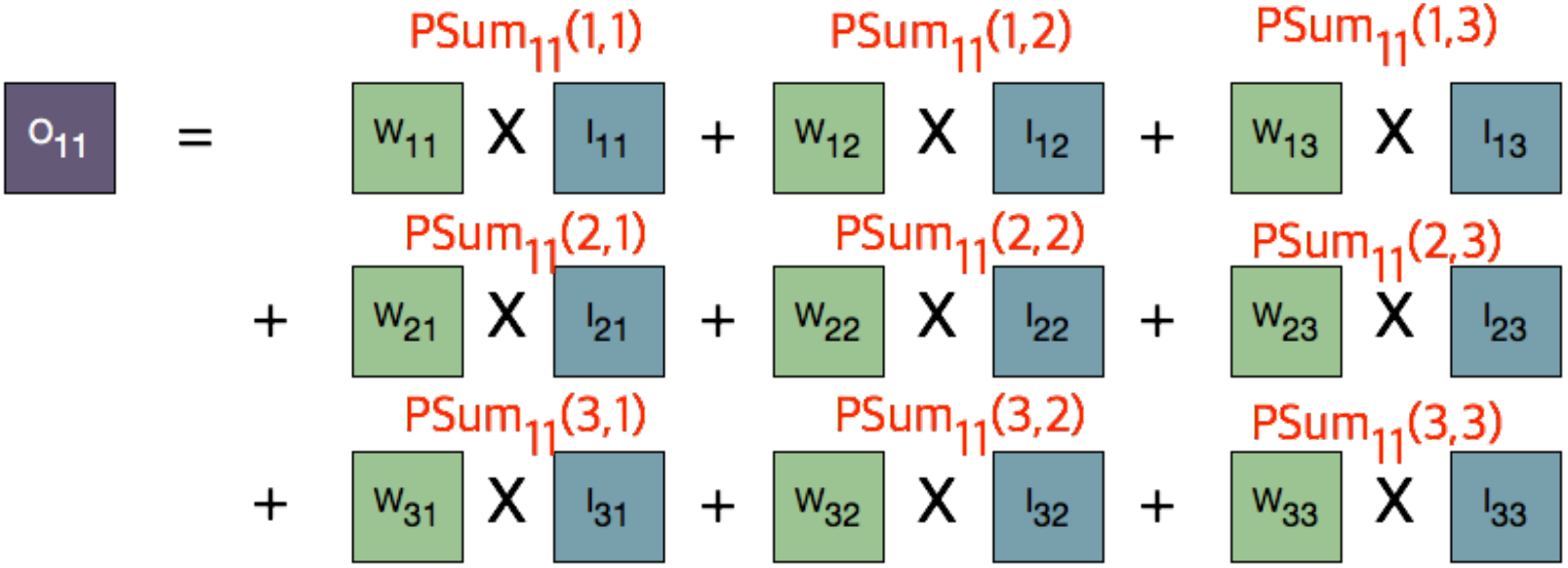
Data Reuse Strategy Example - RS



For simplicity, assume only one channel and one weight filter

Data Reuse Strategy Example - RS

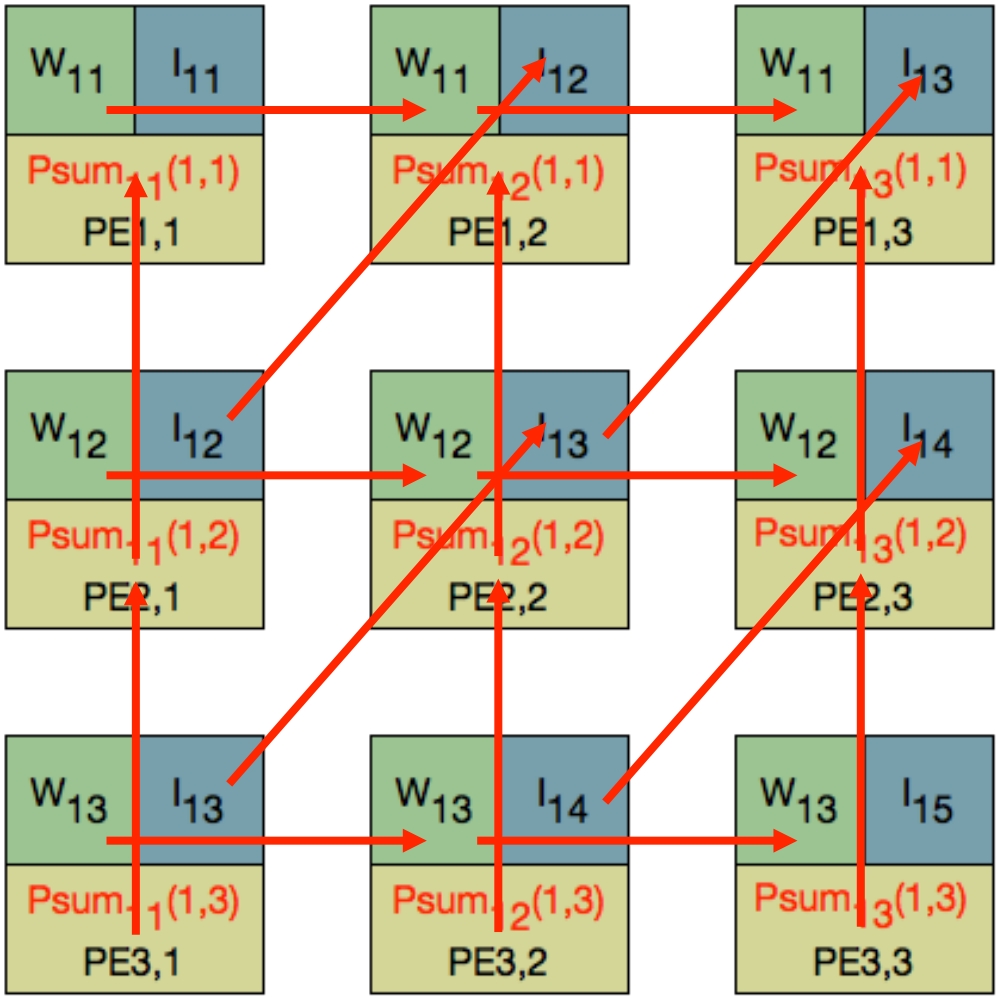
- **Conventions**



$PSum_{ij}(a,b)$: A partial sum for O_{ij} that is generated by W_{ab}

Data Reuse Strategy Example - RS

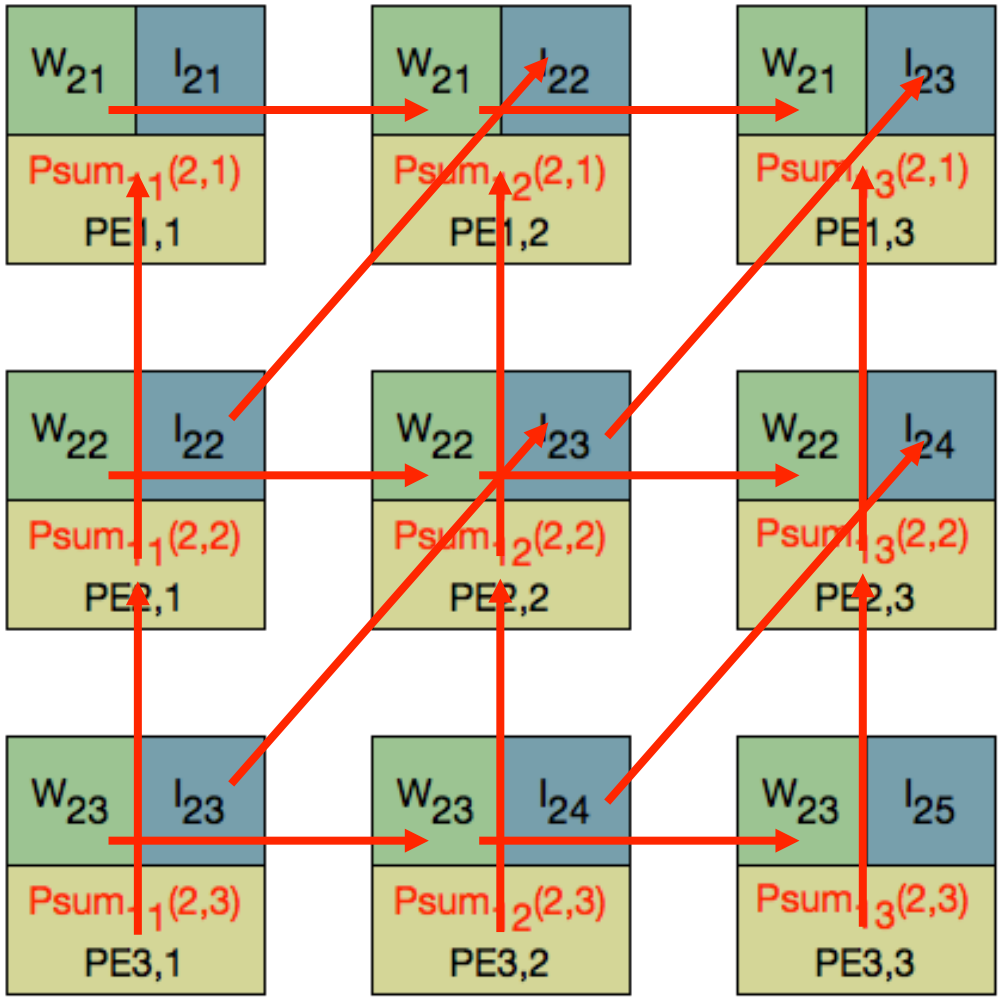
Clk = 0

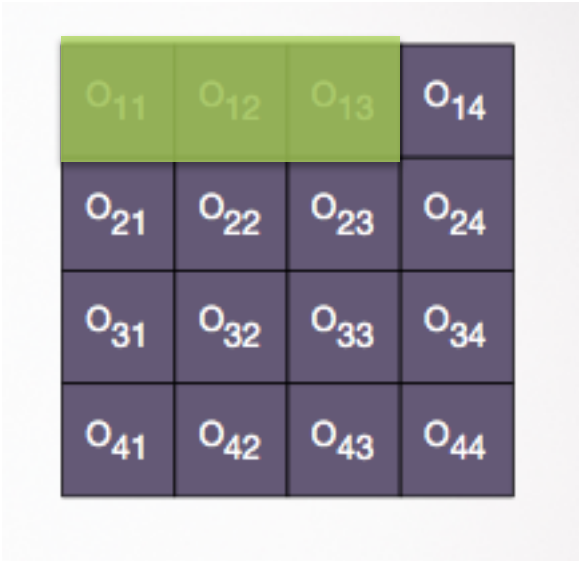


$$O_{11} = W_{11} \times I_{11} + W_{12} \times I_{12} + W_{13} \times I_{13} + W_{21} \times I_{21} + W_{22} \times I_{22} + W_{23} \times I_{23} + W_{31} \times I_{31} + W_{32} \times I_{32} + W_{33} \times I_{33}$$

Data Reuse Strategy Example - RS

Clk = 1

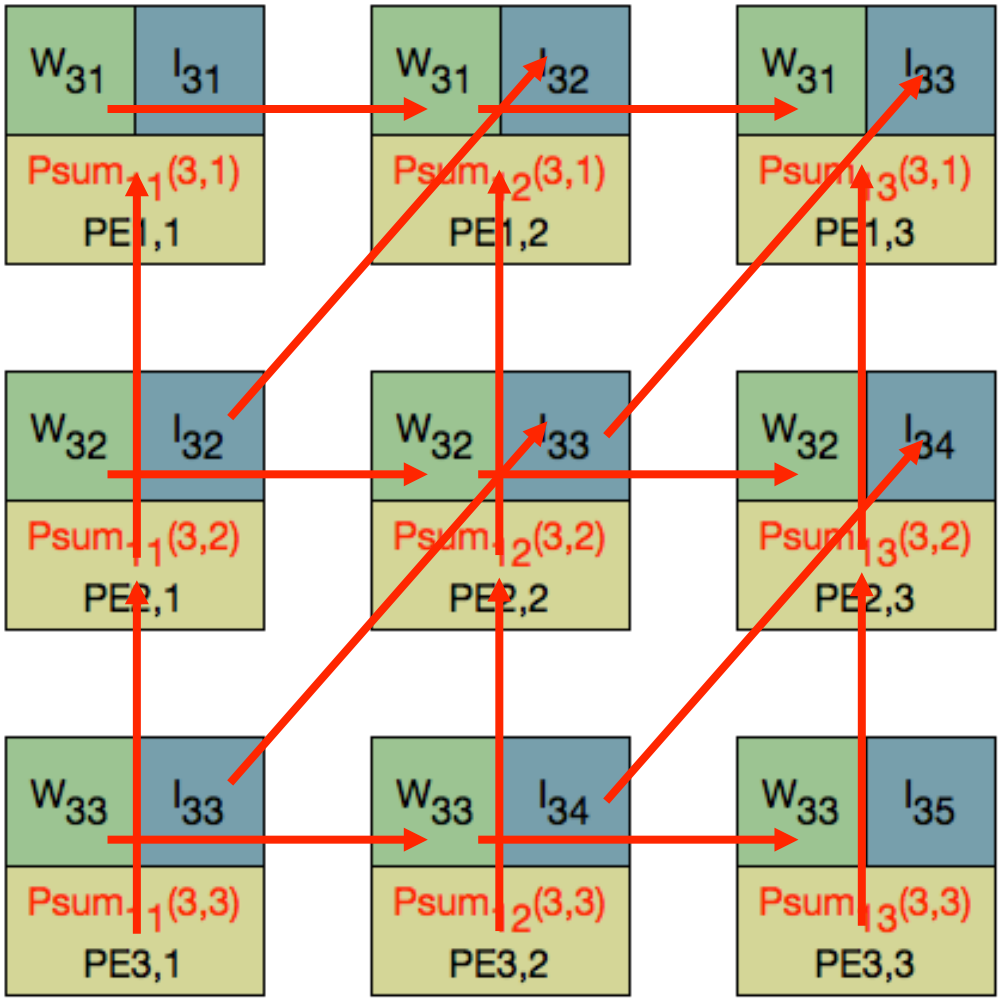




$$o_{11} = w_{11} \times i_{11} + w_{12} \times i_{12} + w_{13} \times i_{13} + w_{21} \times i_{21} + w_{22} \times i_{22} + w_{23} \times i_{23} + w_{31} \times i_{31} + w_{32} \times i_{32} + w_{33} \times i_{33}$$

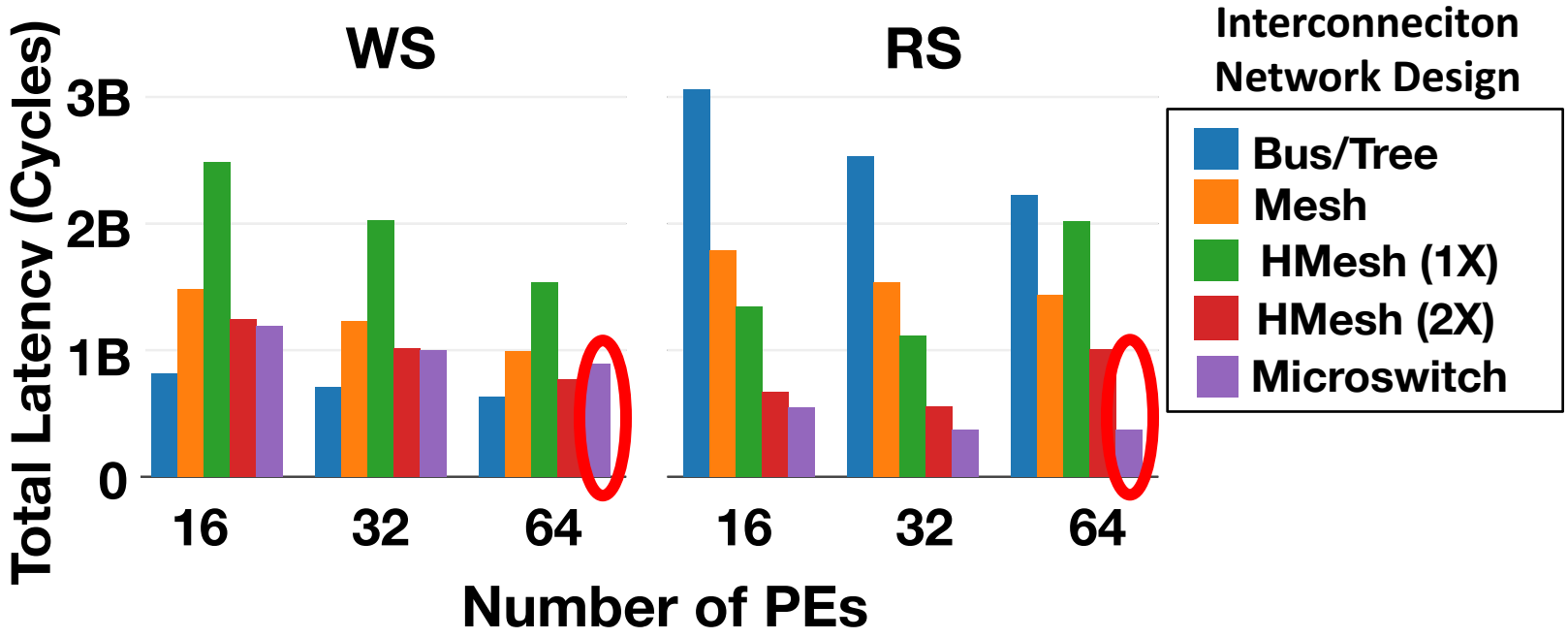
Data Reuse Strategy Example - RS

Clk = 2



$$O_{11} = W_{11} \times I_{11} + W_{12} \times I_{12} + W_{13} \times I_{13}$$
$$+ W_{21} \times I_{21} + W_{22} \times I_{22} + W_{23} \times I_{23}$$
$$+ W_{31} \times I_{31} + W_{32} \times I_{32} + W_{33} \times I_{33}$$

Performance Impact of Dataflow

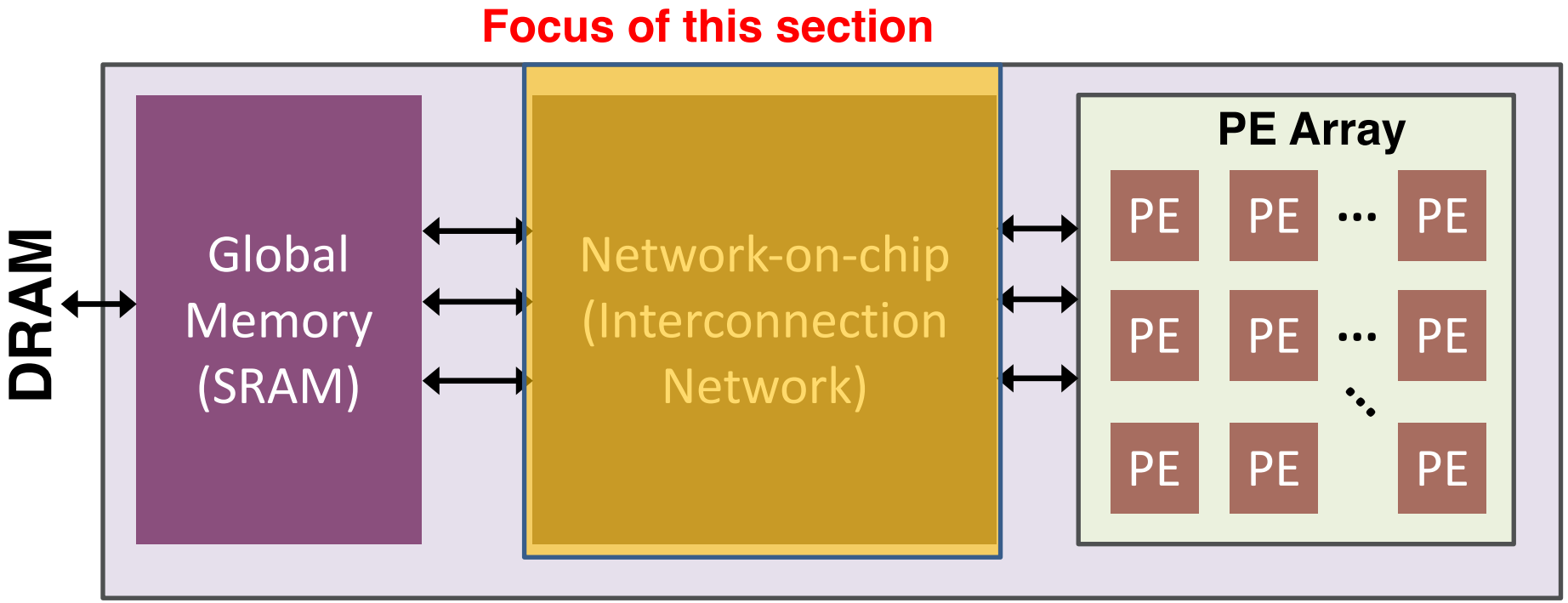


RS reduces the latency 63% with a good interconnection network (Microswitch)

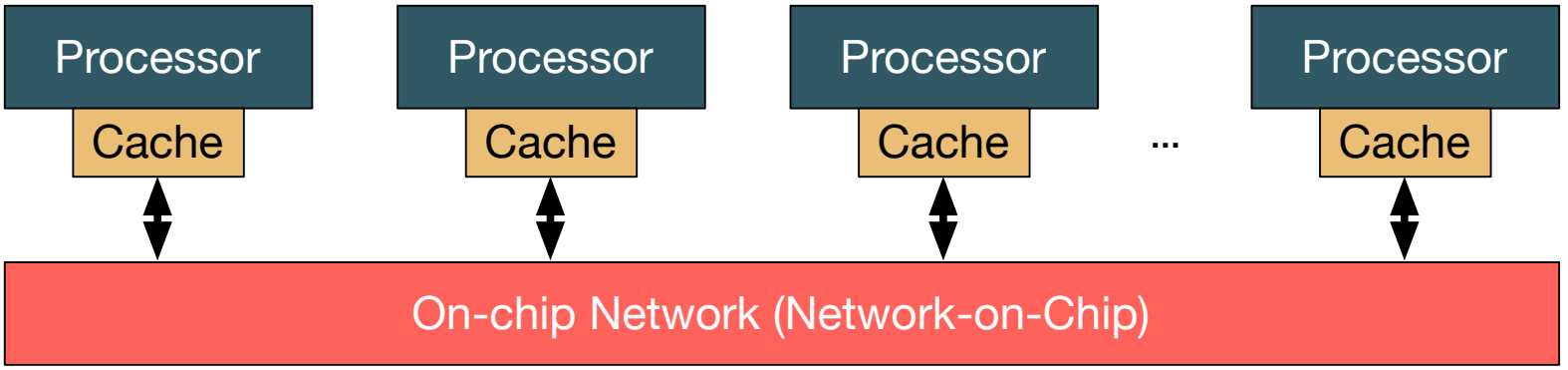
Day 3 Agenda

- **CNN Accelerator Dataflow**
  - Understanding Dataflow
  - Weight-Stationary
  - Row-Stationary
- **Network-on-Chip**
  - Topology
  - NoC Topology for DNN Accelerators
- **Processing Element**
  - Structure Overview
  - Processing Element Array

Network-on-Chip (NoC)



Network-on-Chip (NoC)

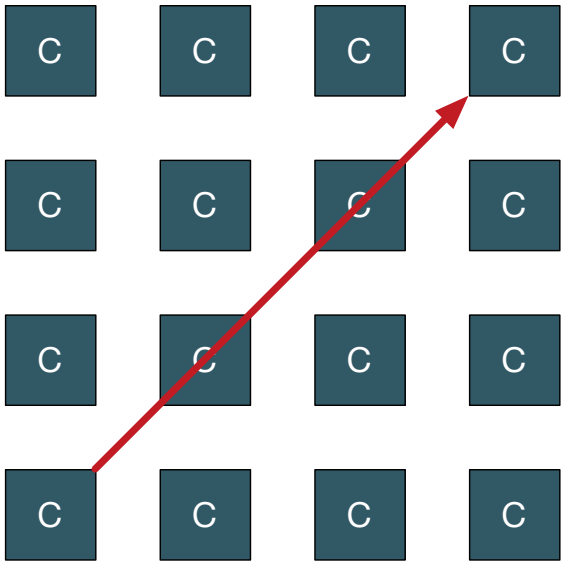


- Emerged to provide connectivity among cores in chip multi-processors (CMPs).
- Bus is good enough for a small number of cores (~=4)
- For larger number of cores/PEs (GPUs, accelerators), we need scalable solutions.

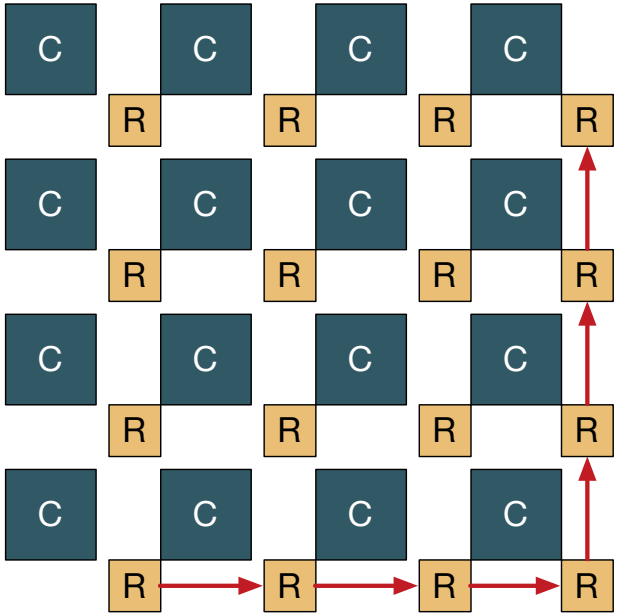
Network-on-Chip (NoC)

- **Why do we need NoC?**

- Provide larger bandwidth
- Implement QoS features
- Reduce critical path delay



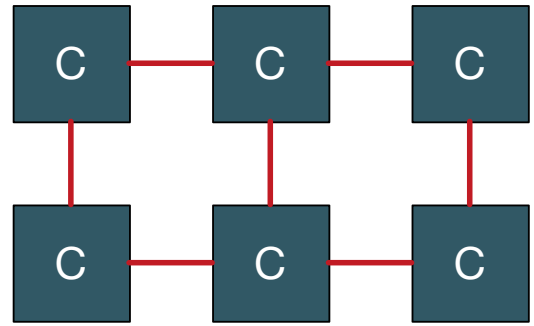
vs.

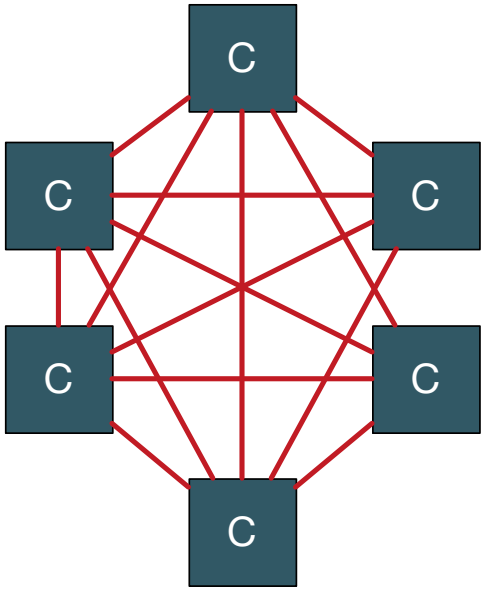


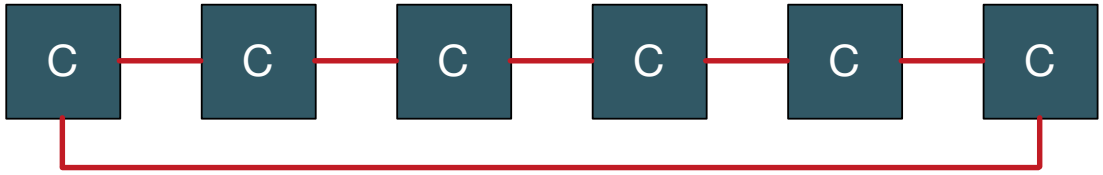
*** C: core , R: router**

Topology

- **Network-on-Chip Topology**
- How do we interconnect cores?



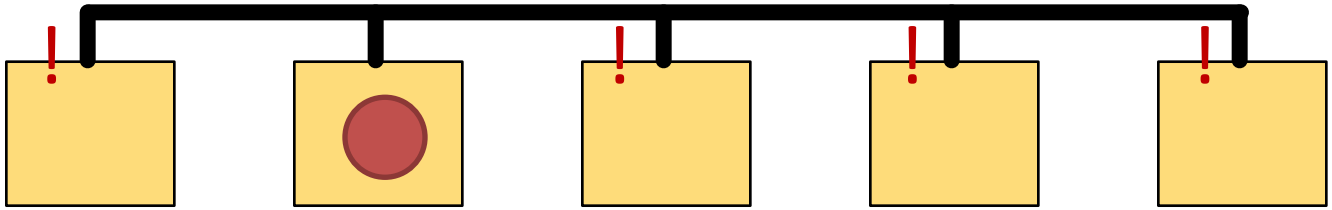




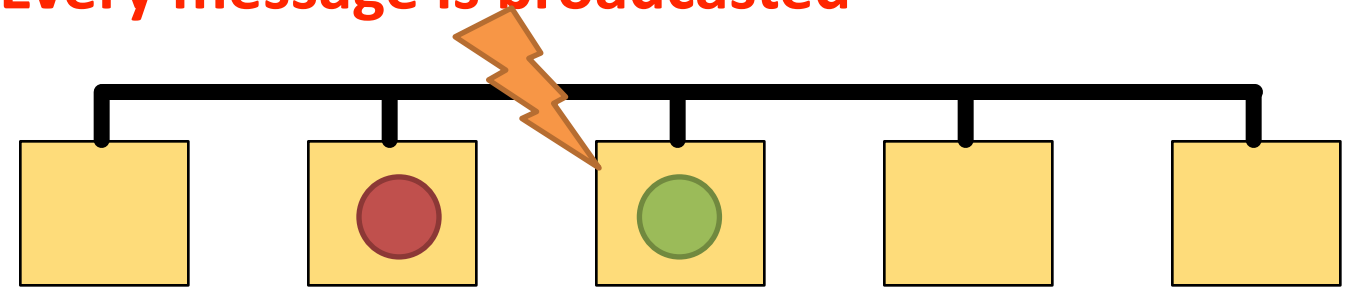
Trade-off exists: delay, bandwidth, area/power overhead, etc.

Topology

- **Bus**
  - Shared wires among all the nodes (nodes can be cores, memory controllers, or PEs)



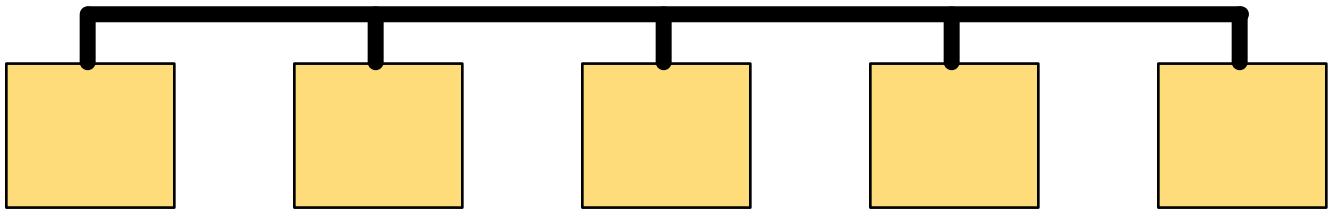
Every message is broadcasted



Arbitration required

Topology

- **Bus**



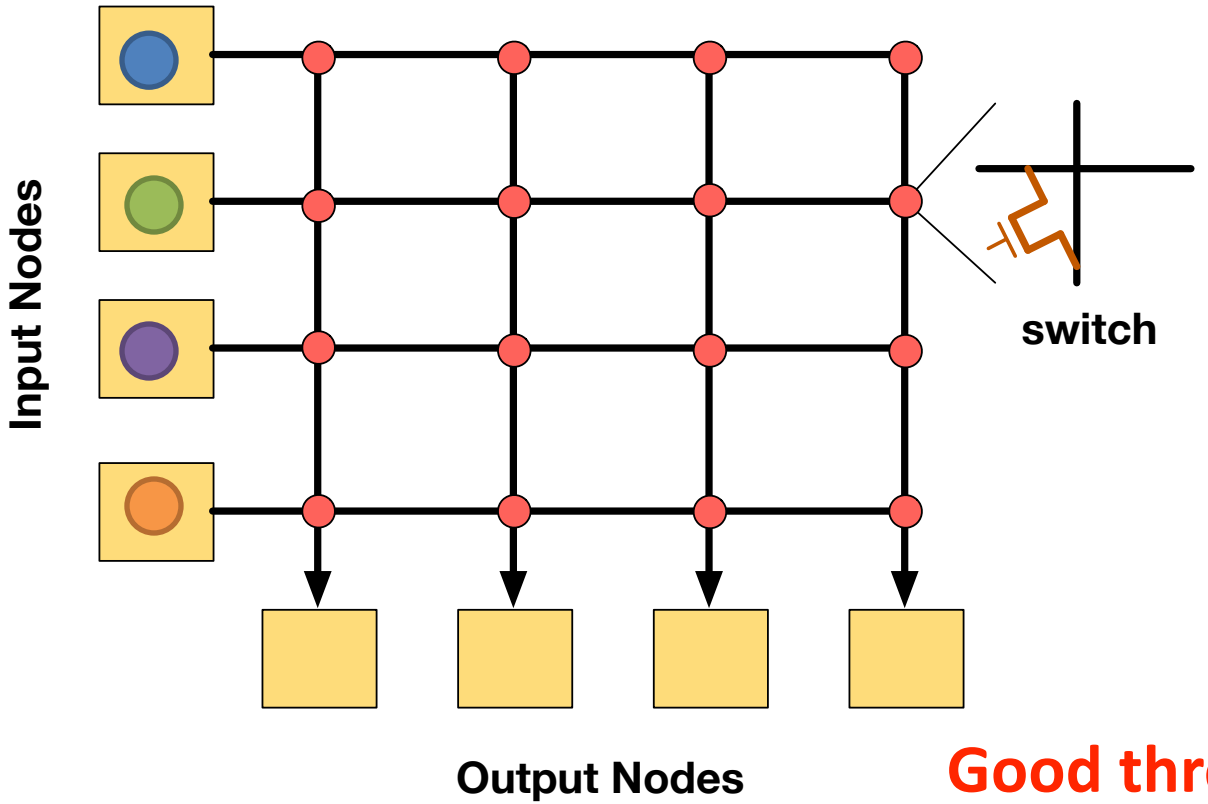
- **Pros**
  - Cost-effective (area and power)
  - Easily supports multicasting/broadcasting

Cons?

- **Cons**
  - Low bandwidth
  - Energy inefficiency (always broadcast)
  - Long control path because of a centralized arbiter
  - Lacks scalability

Topology

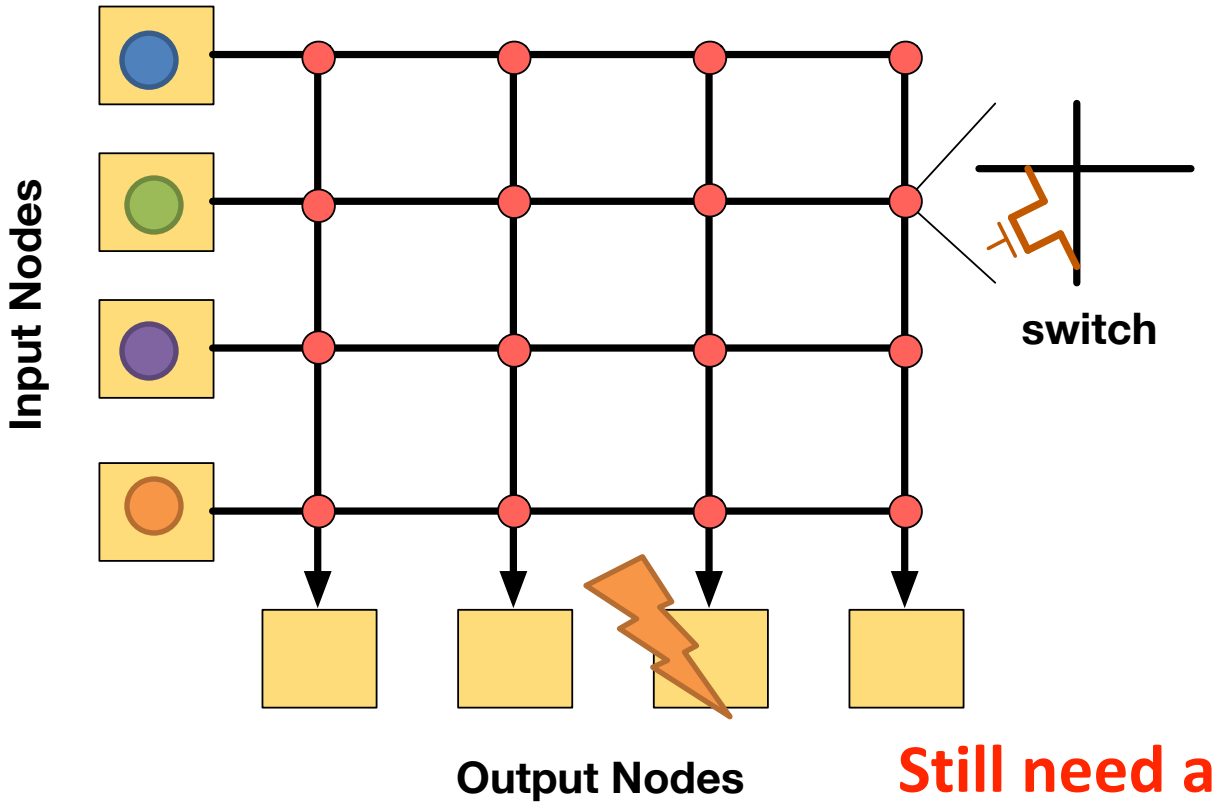
- **Crossbar**
  - Input/output nodes interconnected with an array of switches



Good throughput!

Topology

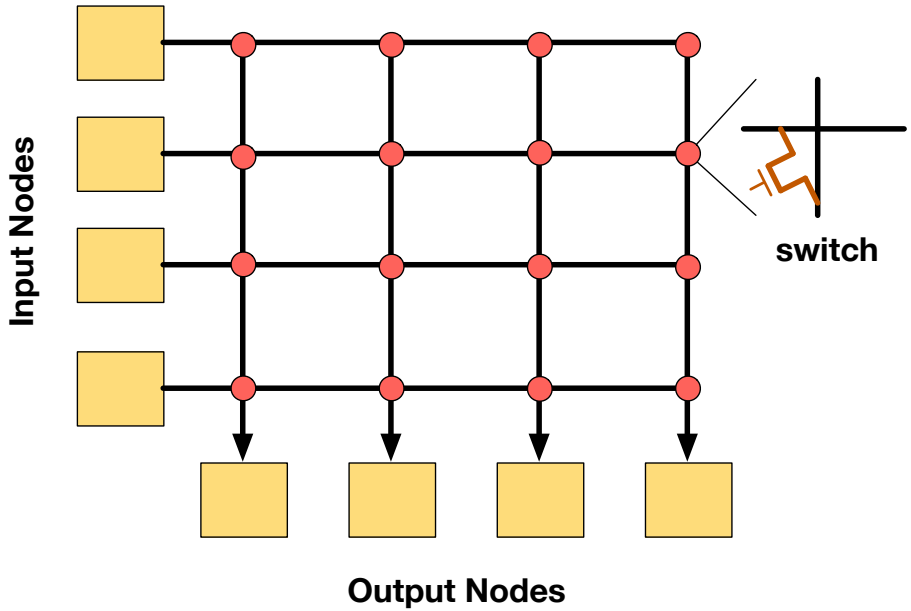
- **Crossbar**
  - Input/output nodes interconnected with an array of switches



Still need arbitration

Topology

- **Crossbar**

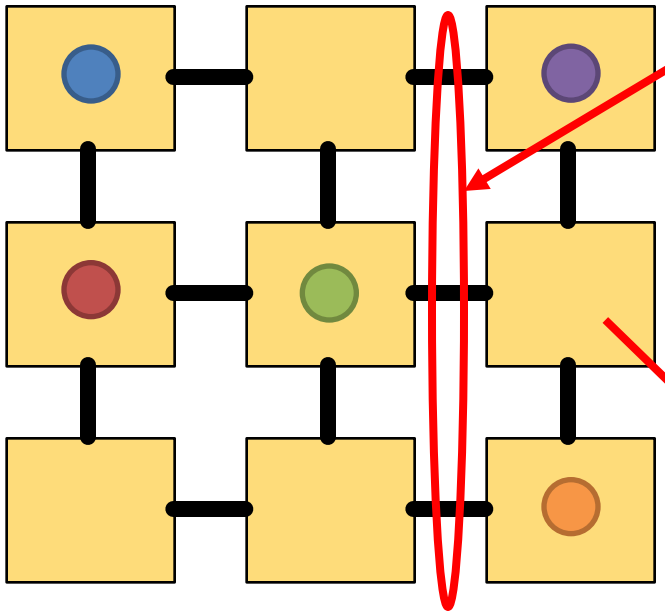


- **Pros**
  - High-throughput

- **Cons**
  - Quadratic area/power overhead
  - Arbitration cost (one arbiter per an output node)

Topology

- **Mesh**
  - Nodes are interconnected with adjacent nodes



High bisection bandwidth

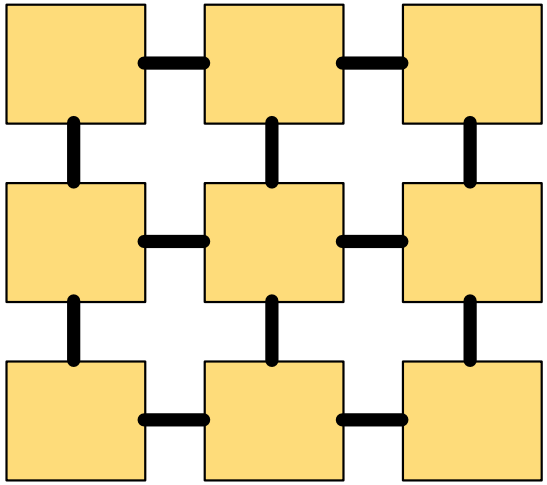
Conflict?

No. Mesh routers have separate input queues for each port

Hop-by-hop traversal

Topology

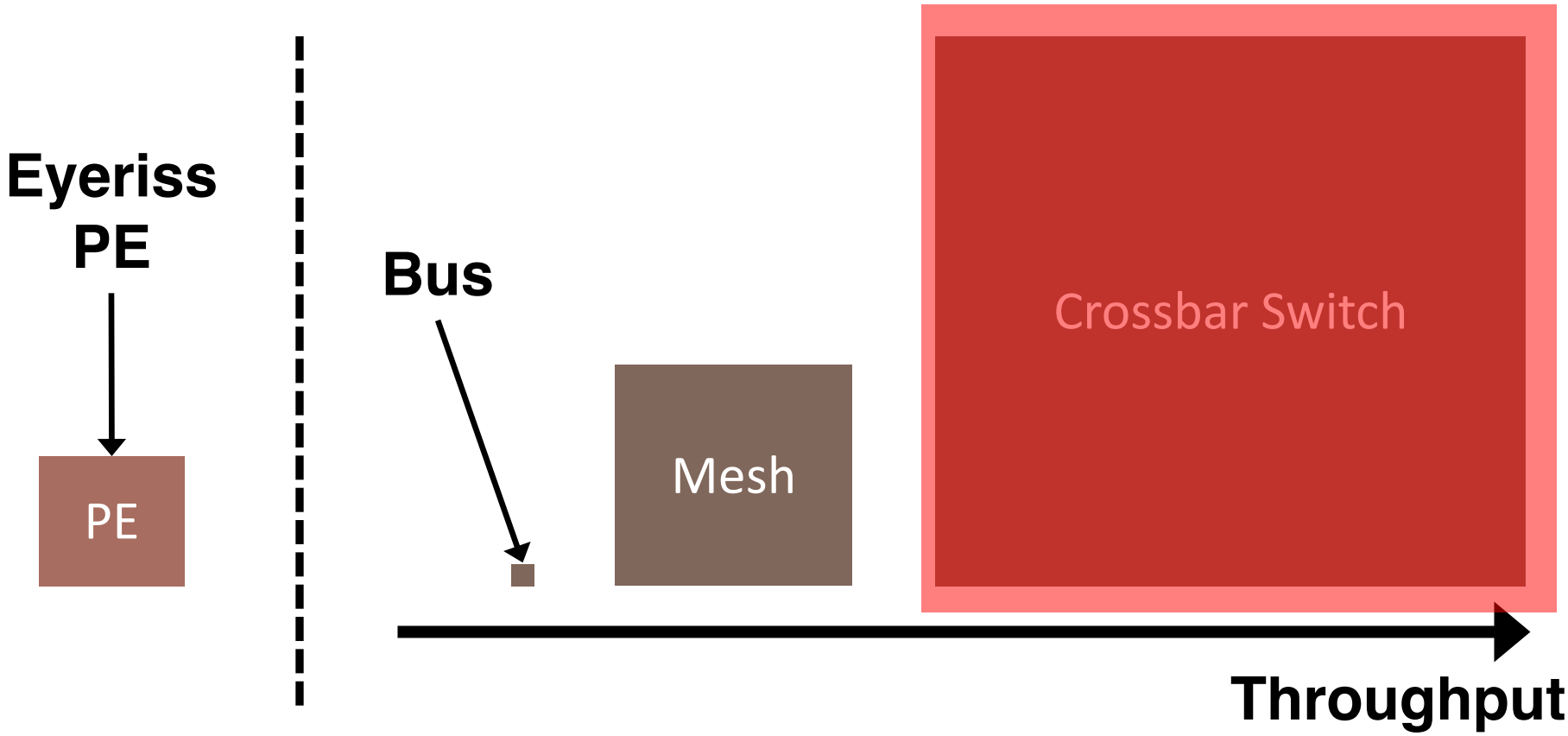
- **Mesh**



- **Pros**
  - Provides high bisection bandwidth
  - Involves small wire overhead
- **Cons**
  - Routers are expensive in area and power
  - Requires high delay in cycles (hop-by-hop traversal)
  - Doesn't support broadcasting

NoC Topology for DNN Accelerators

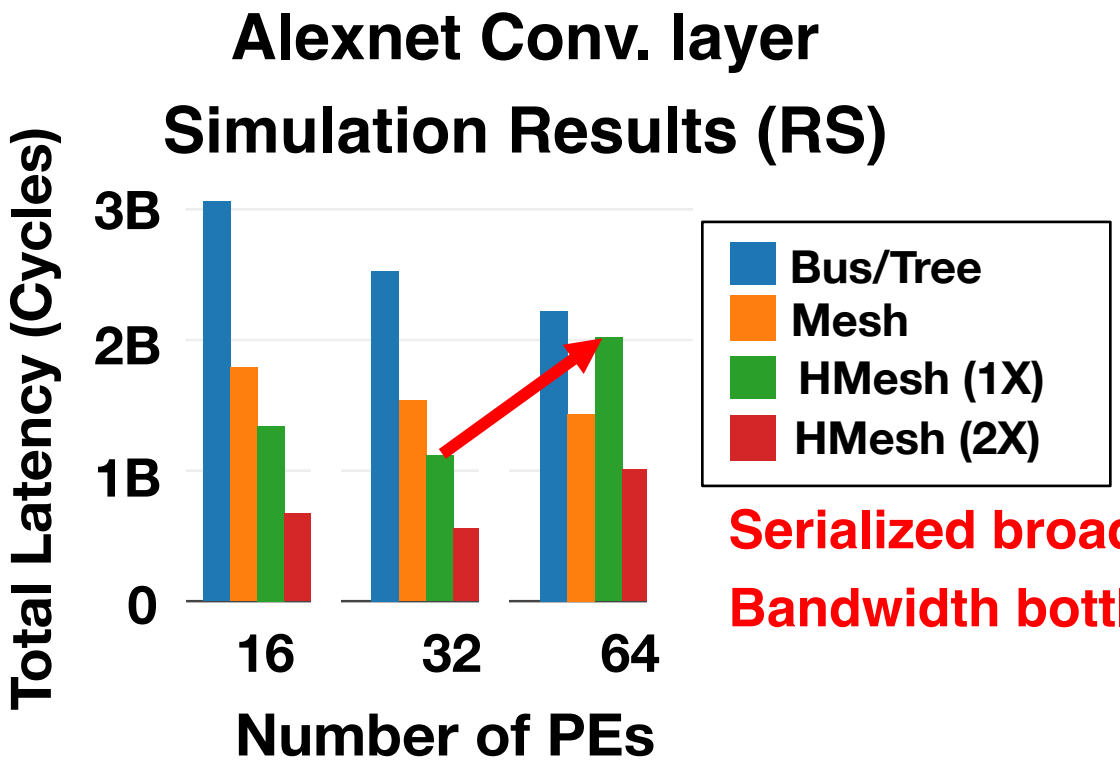
- **Relative Area Overhead Compared to Tiny PEs**



Size of squares of NoC: Total area divided by the number of PEs (256 PEs)

NoC Topology for DNN Accelerators

- **Bandwidth**

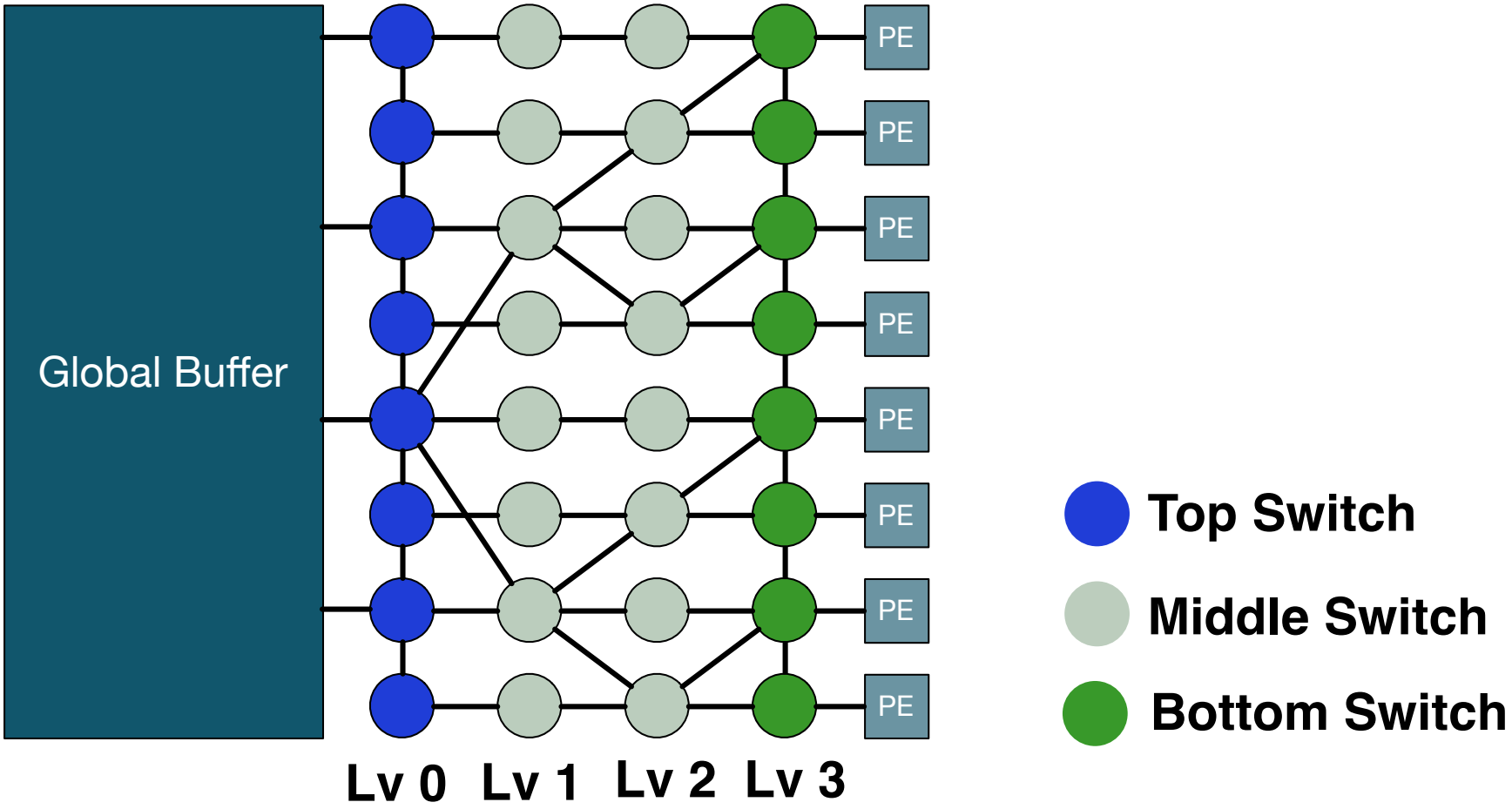


Serialized broad-/multi-casting

Bandwidth bottleneck at top level

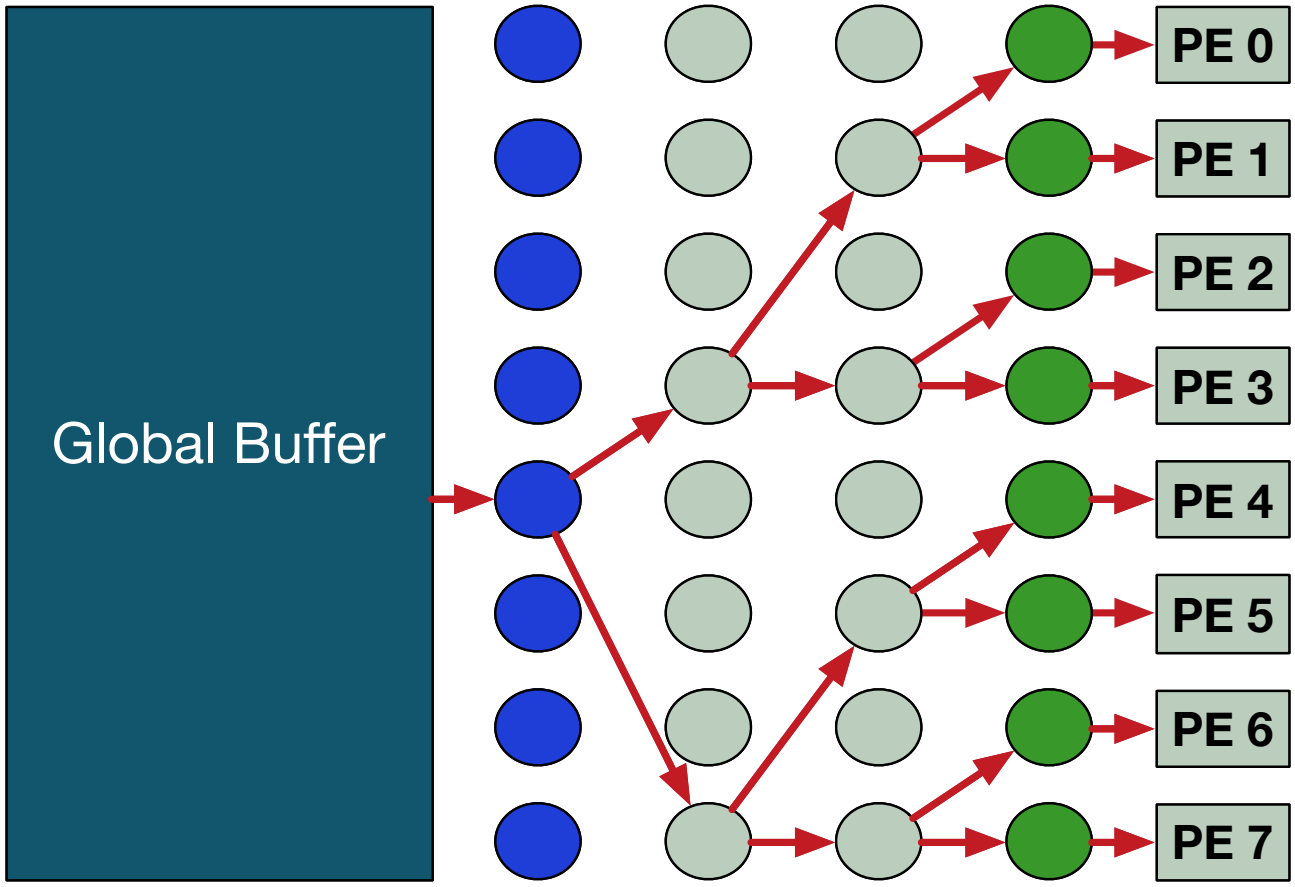
Bus provides low bandwidth for DNN traffic

Microswitch NoC



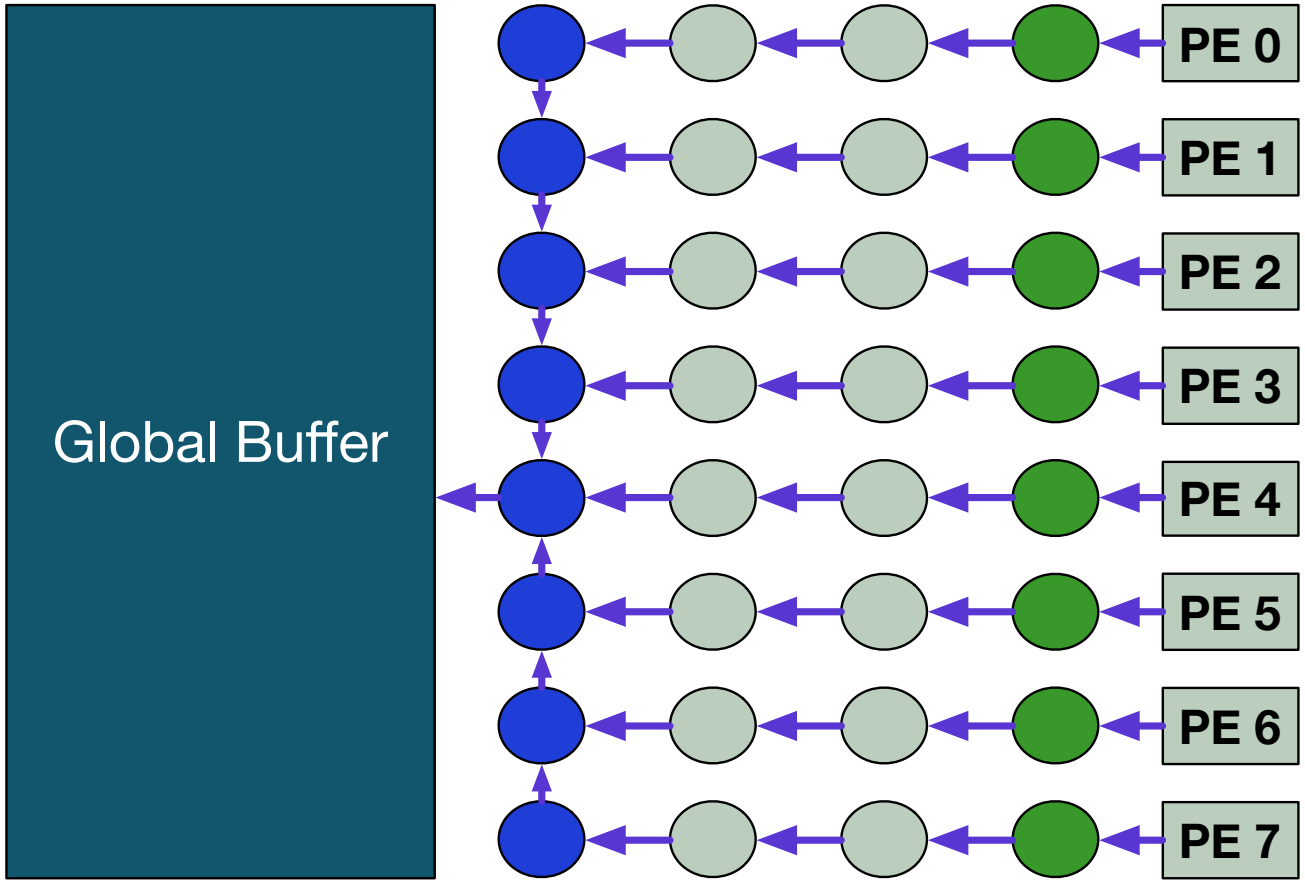
- **Distribute communication to tiny switches**

Routing: Scatter Traffic



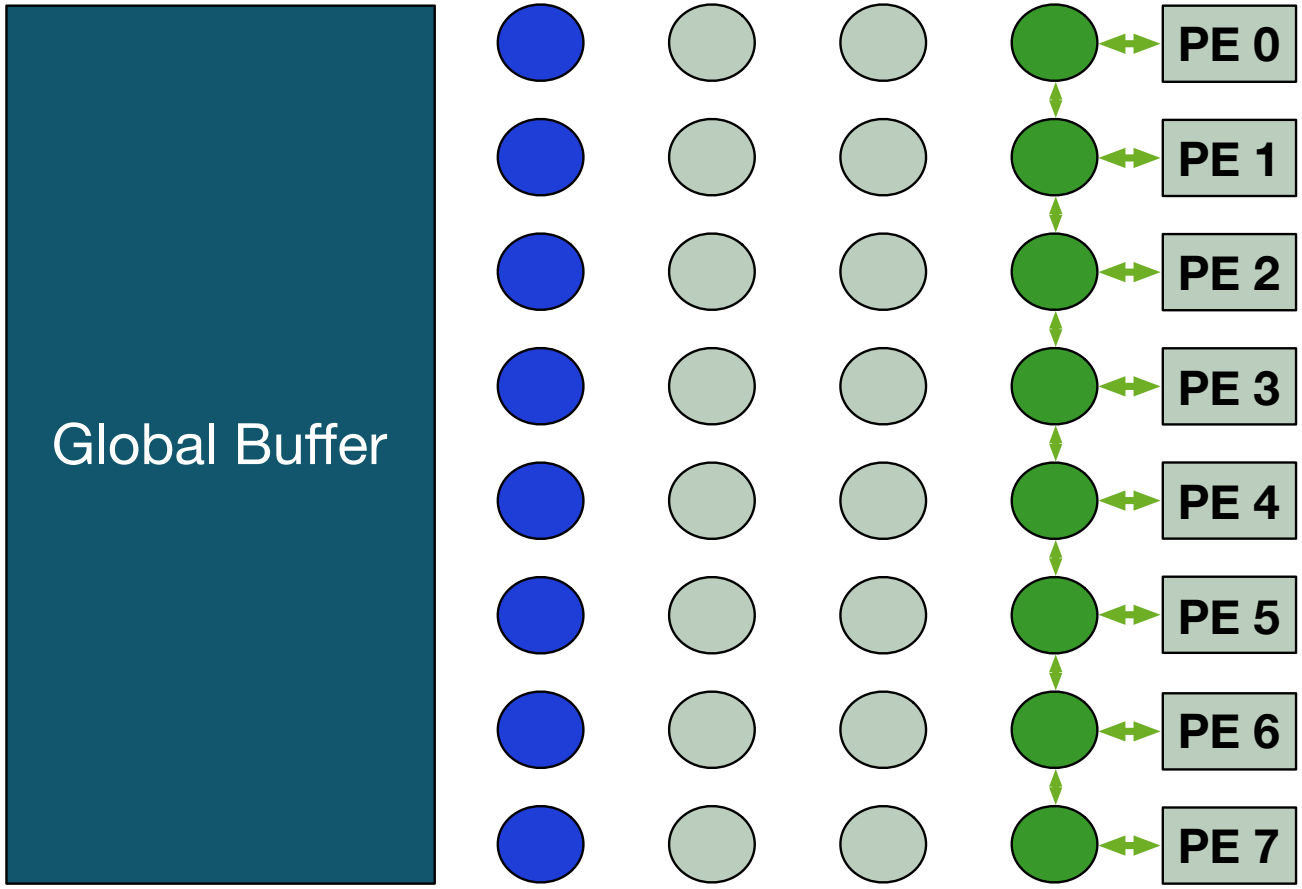
- **Tree-based broad/multicasting**

Routing: Gather Traffic



- **Multiple pipelined linear network**
- **Bandwidth bound to GBM write bandwidth**

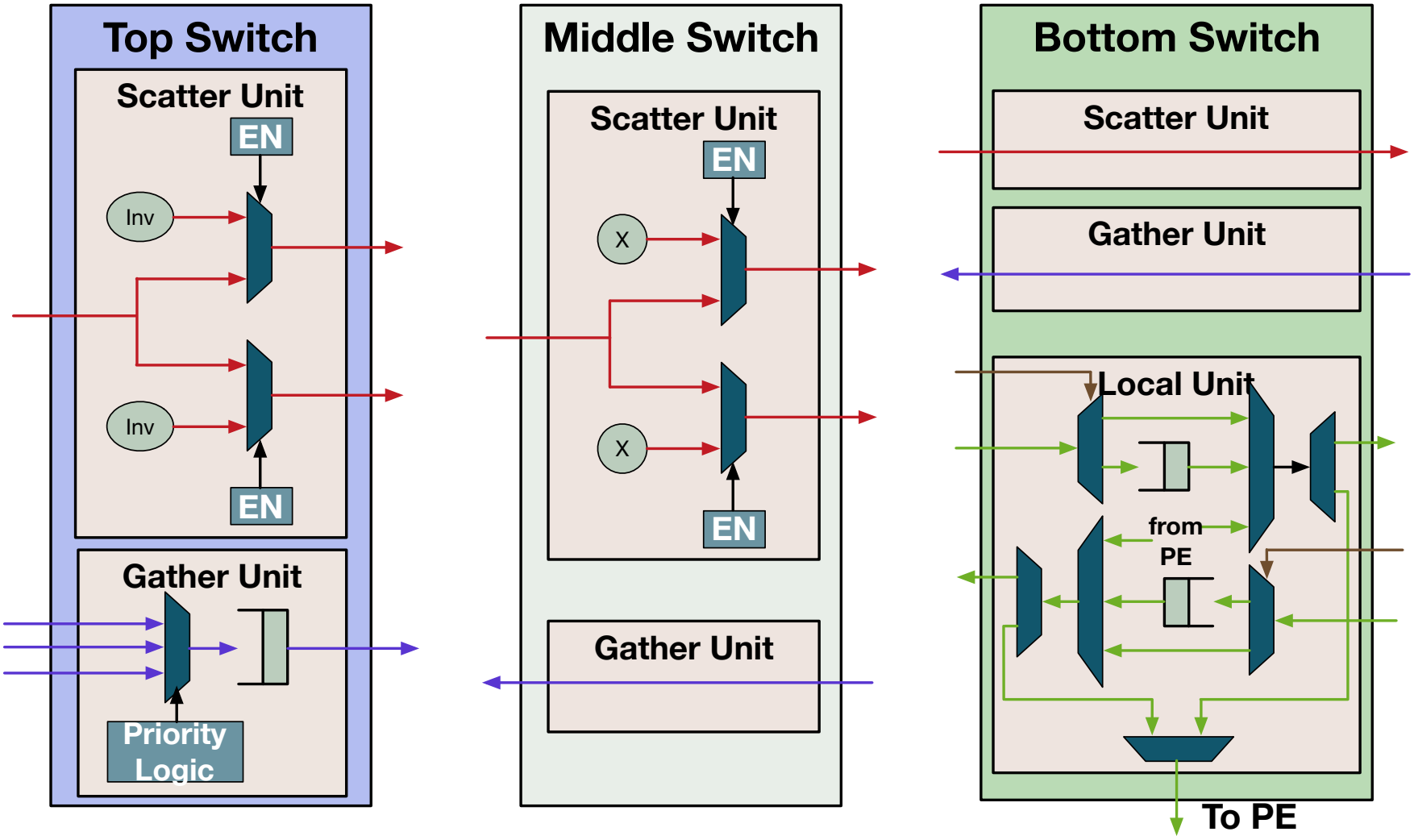
Routing: Local Traffic



- **Linear single-cycle multi-hop (SMART*) network**

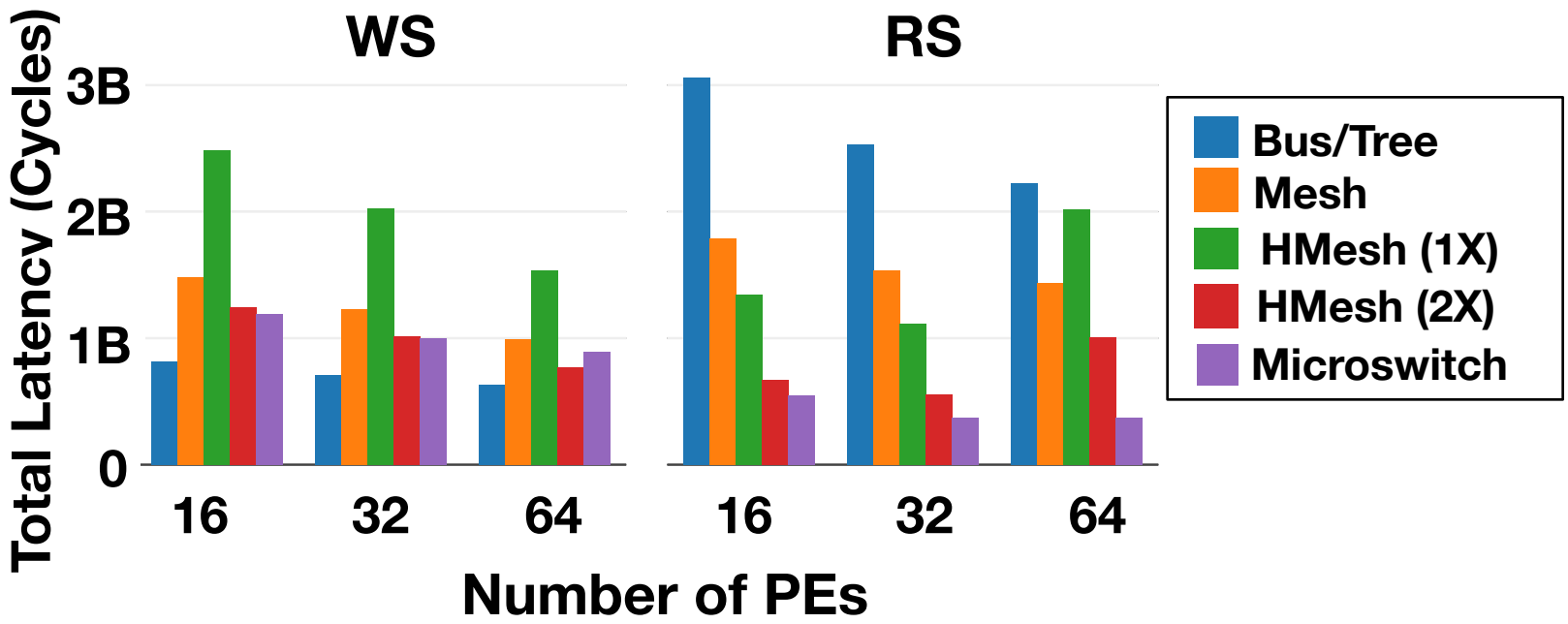
H. Kwon et al., OpenSMART: Single cycle-Multi-hop NoC Generator in BSV and Chisel, ISPASS 2017
T. Krishna et al., *Breaking the On-Chip Latency Barrier Using SMART,* HPCA 2013

Microarchitecture: Microswitches



Performance of Microswitch NoC

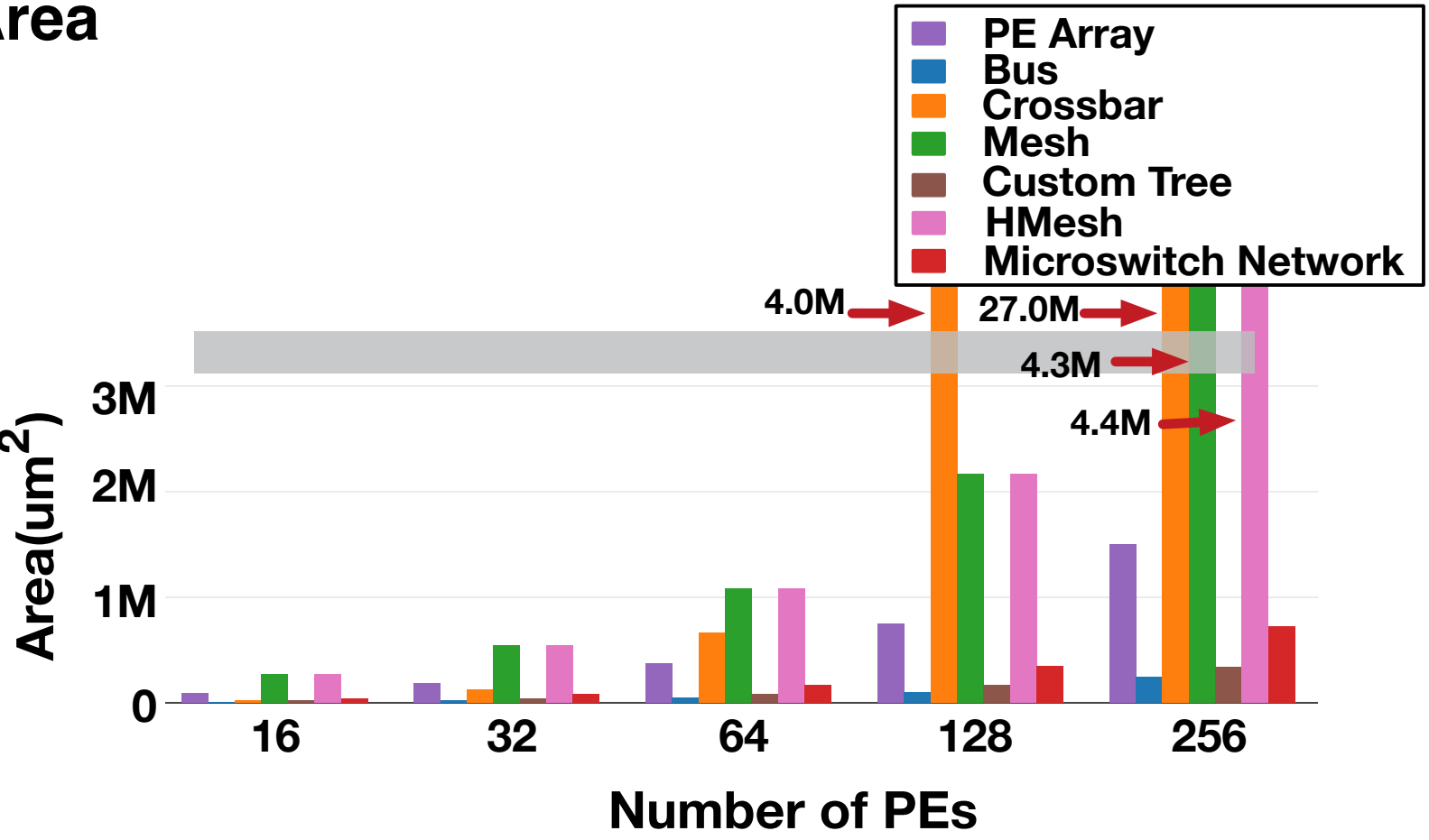
- **Latency (Entire Alexnet Convolutional layers)**



Microswitch reduces the latency by 61% compared to mesh

Overhead of Microswitch NoC

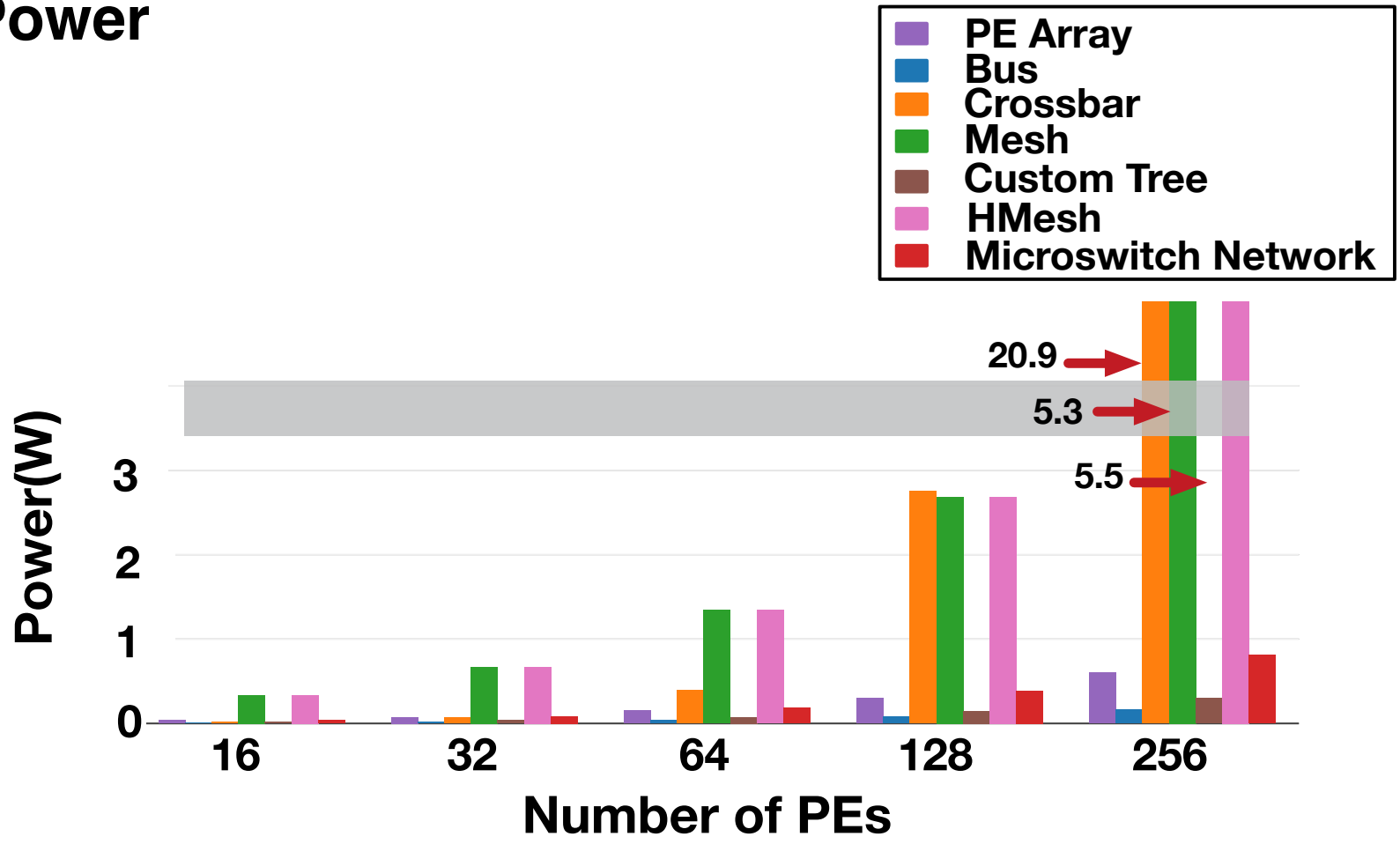
- **Area**



Microswitch NoC requires 16% area of mesh

Overhead of Microswitch NoC

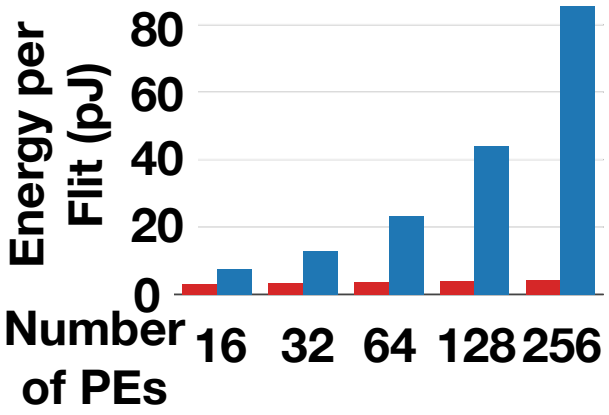
- **Power**

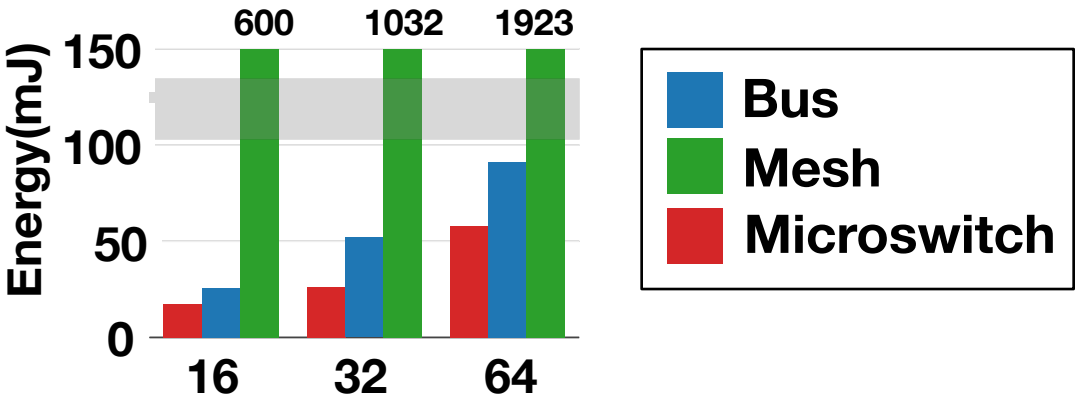


Microswitch NoC only consumes 12% power of mesh

Overhead of Microswitch NoC

- **Energy**





Buses always need to broadcast, even for unicast traffic

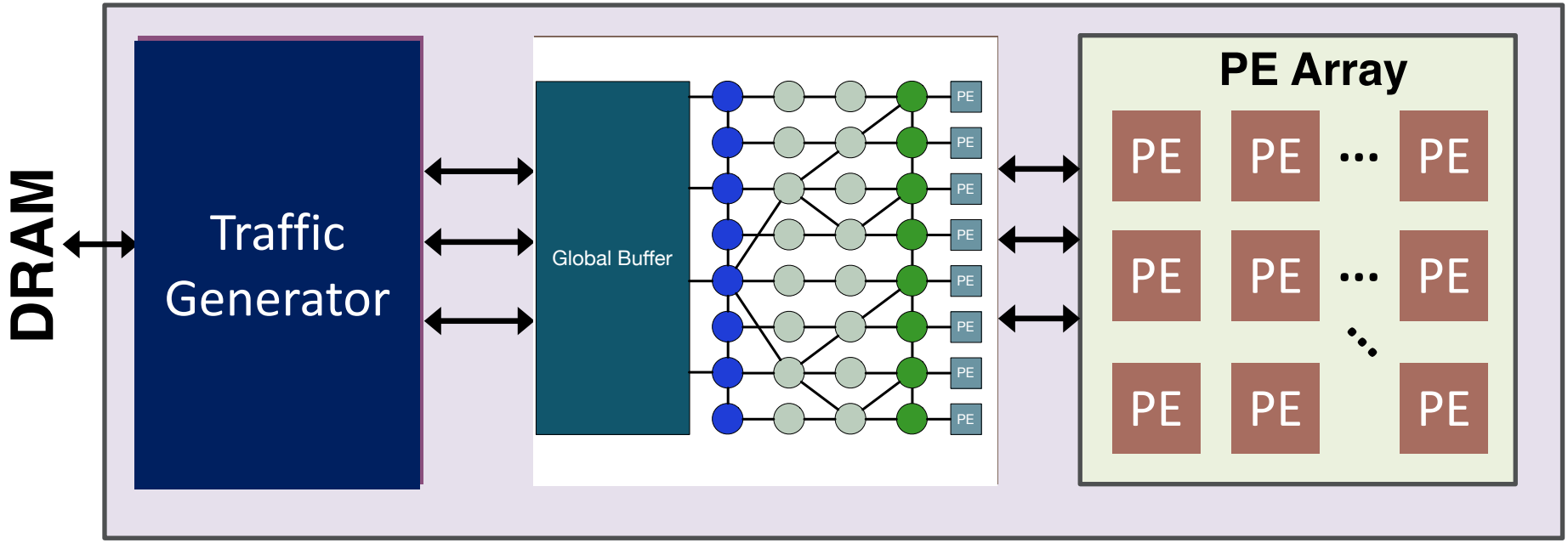
Microswitch NoC enables only necessary links

Day 3 Agenda

- **CNN Accelerator Dataflow**
  - Understanding Dataflow
  - Weight-Stationary
  - Row-Stationary
- **Network-on-Chip**
  - Topology
  - NoC Topology for DNN Accelerators
- **Processing Element**
  - Structure Overview
  - Processing Element Array

[Lab] WS Processing Elements

- **Our WS Accelerator Model**

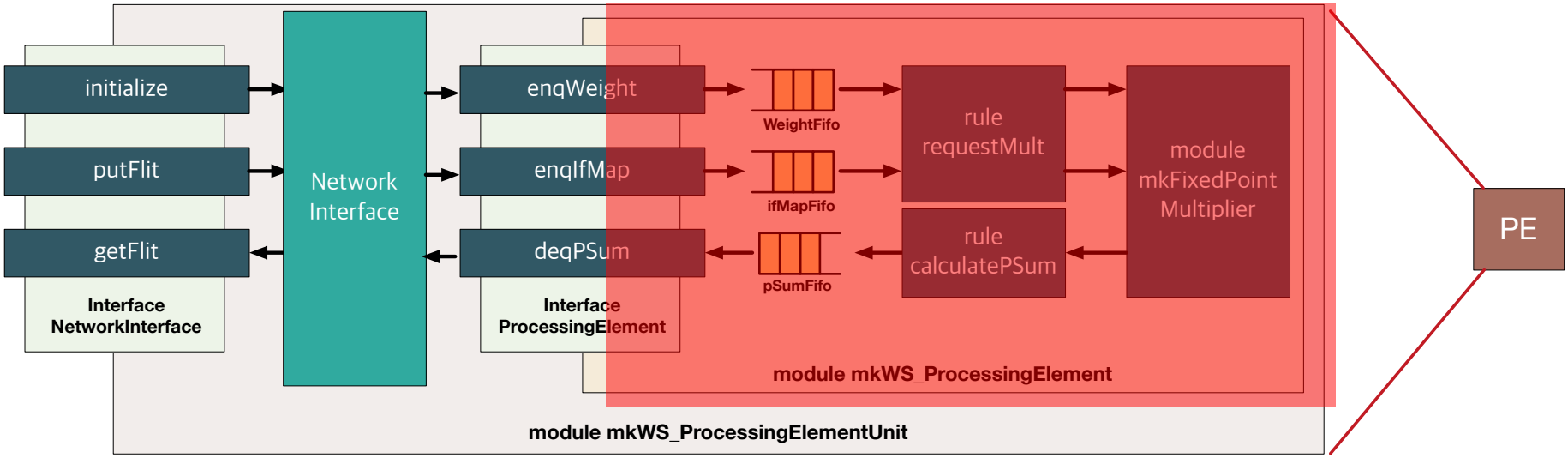


"Push" model: Global buffer pushes data to PE array

GBM is master, and PEs are slaves.

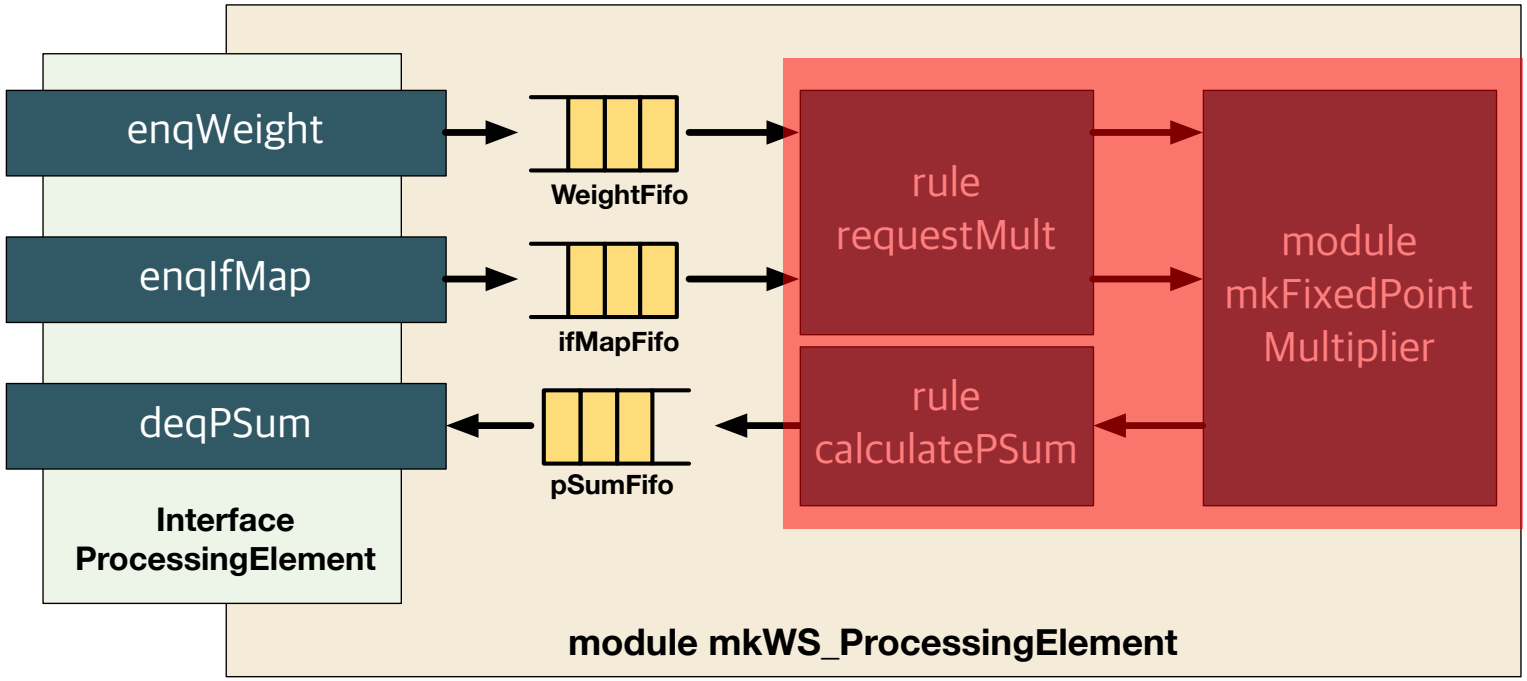
[Lab] WS Processing Elements

- **Weight-stationary(WS) Processing Element**



[Lab] WS Processing Elements

- **Weight-stationary(WS) Processing Element**

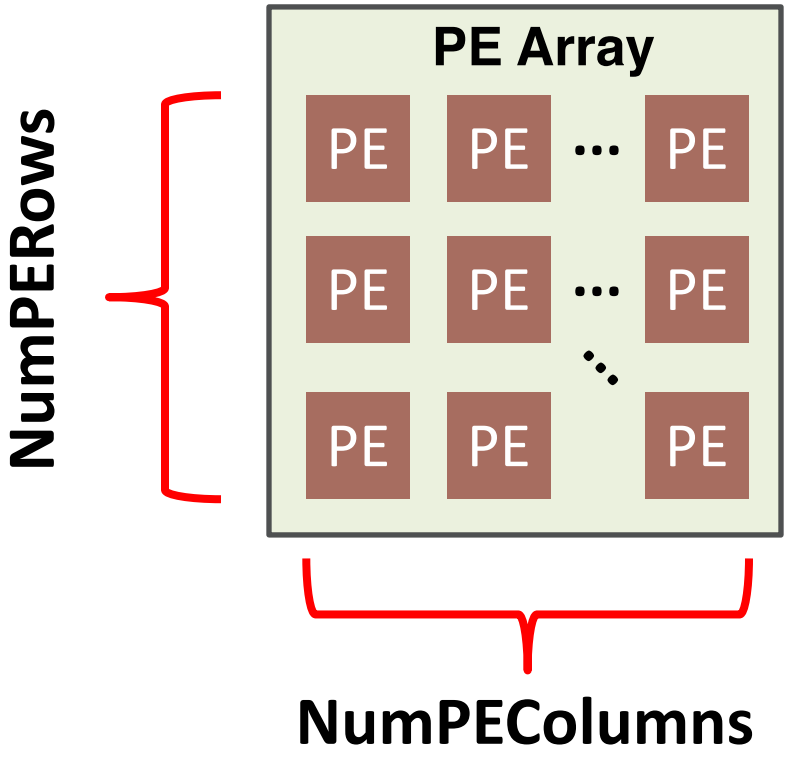


Replacing fake multiplier with your fixed point multiplier

TODOs are annotated in the source code

[Lab] Processing Element Array

- **Instantiating Module Array**



How to instantiate an array of modules? **Use Vector**

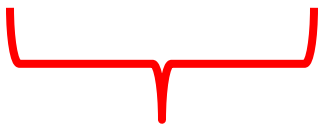
[Lab] Processing Element Array

- **Instantiating Module Array**
  - Example) a 4 x 5 register array

```
Vector#(4, Vector#(5, Reg#(Bit#(16)))) regArray;
```

Does it work? No, need to instantiate the modules

```
Vector#(4, Vector#(5, Reg#(Bit#(16)))) regArray
<- replicateM(replicateM(mkReg(0)));
```



replicateM(Instantiation Ins): apply Ins for all the elements in the vector

[Lab] Connecting Components

- **Connecting two methods**

```
interface DataIfc;
  method Action putData(Data data);
  method ActionValue#(Data) getData;
endinterface

module mkTopModule();
  DataIfc moduleA <- mkModuleA;
  DataIfc moduleB <- mkModuleB;
  ...
endmodule
```

How to connect putData of A and getData of B?

[Lab] Connecting Components

- **Connecting two methods**

```
module mkTopModule();
  DataIfc moduleA <- mkModuleA;
  DataIfc moduleB <- mkModuleB;

  rule connectModules;
    let bData <- moduleB.getData;
    moduleA.putData(bData);
  endrule

endmodule
```

[Lab] Connecting Components

- **Connecting two methods**

```
module mkTopModule();
  DataIfc moduleA <- mkModuleA;
  DataIfc moduleB <- mkModuleB;

  mkConnection(moduleB.getData, moduleA.putData);

endmodule
```

[Lab] Connecting Components

- **For-loop Usage 1**
  - Iterate over a Bit#(N) type

```
module mkExModule();
  Reg#(Bit#(32)) exampleReg <- mkReg(0);

  rule checkBits;
    for(Integer idx = 0; idx< 32; idx = idx +1) begin
      if(exampleReg[idx] == 1) begin
        $display(“bit %d is one!”, idx);
      end
    end
  endrule

endmodule
```

[Lab] Connecting Components

- **For-loop Usage 2**
  - Iterate over a vector

```
module mkExModule();
  Vector#(4, Reg#(Bit#(32))) exampleRegs
                                        <- replicateM(mkReg(0));

  rule checkRegs;
    for(Integer idx = 0; idx< 4; idx = idx +1) begin
      if(exampleReg[idx] == 1) begin
        $display("register[%d]'s value is one!", idx);
      end
    end
  endrule

endmodule
```

[Lab] Connecting Components

- **For-loop Usage 3**
  - Define multiple rules

```
module mkExModule();
  Vector#(4, Reg#(Bit#(32))) exampleRegs
                                    <- replicateM(mkReg(0));

  for(Integer idx = 0; idx< 4; idx = idx +1) begin
    rule checkRegs;
      exampleRegs[idx] <= fromInteger(idx);
    endrule
  end

endmodule
```

describe four rules

[Lab] Connecting Components

- **For-loop Usage 4: Revisiting mkConnection example**

```
interface DataIfc;
  method Action putData(Data data);
  method ActionValue#(Data) getData;
endinterface

module mkTopModule();
  DataIfc moduleA <- mkModuleA;
  DataIfc moduleB <- mkModuleB;
  ...
endmodule
```

[Lab] Connecting Components

- **For-loop Usage 4**
  - Define connections

```
module mkExModule();
  Vector#(4, DataIfc) moduleAs <- replicateM(mkModuleA);
  Vector#(4, DataIfc) moduleBs <- replicateM(mkModuleB);

  for(Integer idx = 0; idx< 4; idx = idx +1) begin
    mkConnection(moduleAs[idx].putData,
                 moduleBs[idx].getData);
  end

endmodule
```

What about interconnecting 2D array of modules?

[Lab] Processing Element Array

- **Files to edit**
  - **WS_PE.bsv:** Add float multiplier
  - **WS_PE_Array.bsv:** Construct a PE array
  - **TopModule.bsv:** Interconnect PE array and NoC

- **How to run compilation and testbench?**
  - **Compilation:** > make
  - **Testbench:** > make run

[Lab] Processing Element Array

- **Print-out Messages**
  - You will see the number of injected weight, input feature maps (IfMaps), and partial sums (PSums).

```
Elapsed cycle time:          0
Num injected Weights:          0
Num injected IfMaps:          0
Num received PSums:          0
[WS_PE_Array]Initialize
Elapsed cycle time:      10000
Num injected Weights:         32
Num injected IfMaps:       8416
Num received PSums:       8390
Elapsed cycle time:      20000
Num injected Weights:         32
Num injected IfMaps:      16832
Num received PSums:      16812
```

[Lab] Processing Element Array

- **Questions**
  1) Why does the number of injected weight remain constant for a while?
  2) Why the number of injected IfMaps and PSums are simliar?
  3) Why the number of injected IfMaps and PSums are not exactly same?

[Demo] Hardware Synthesis and PnR