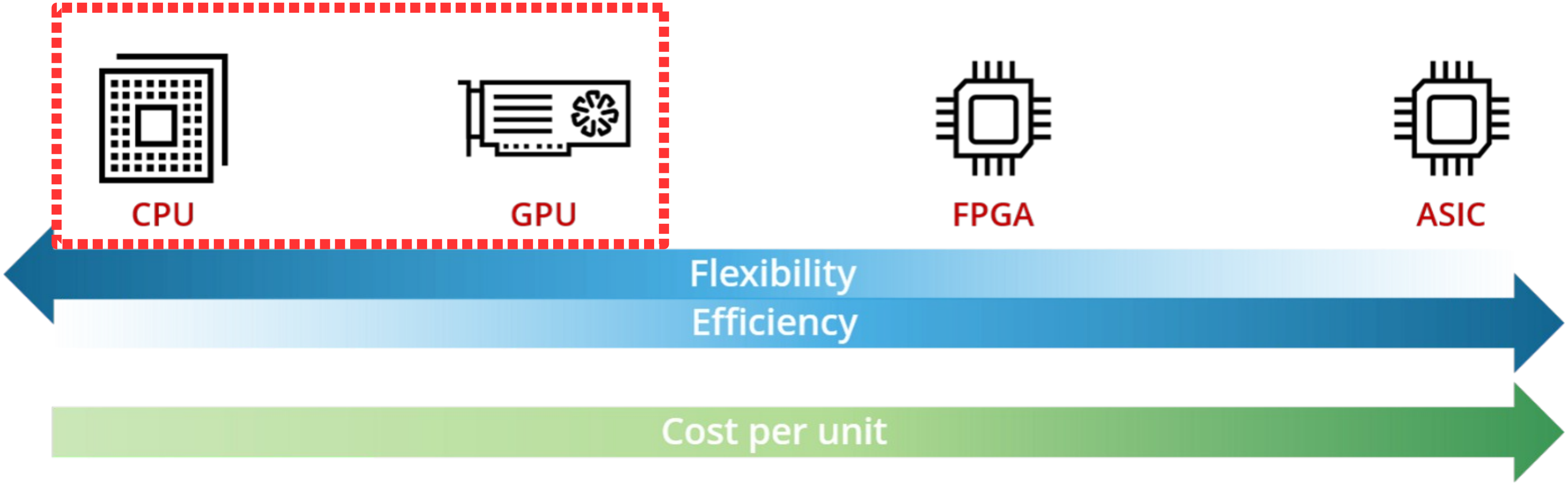
Data Flow Techniques

Workshop Agenda

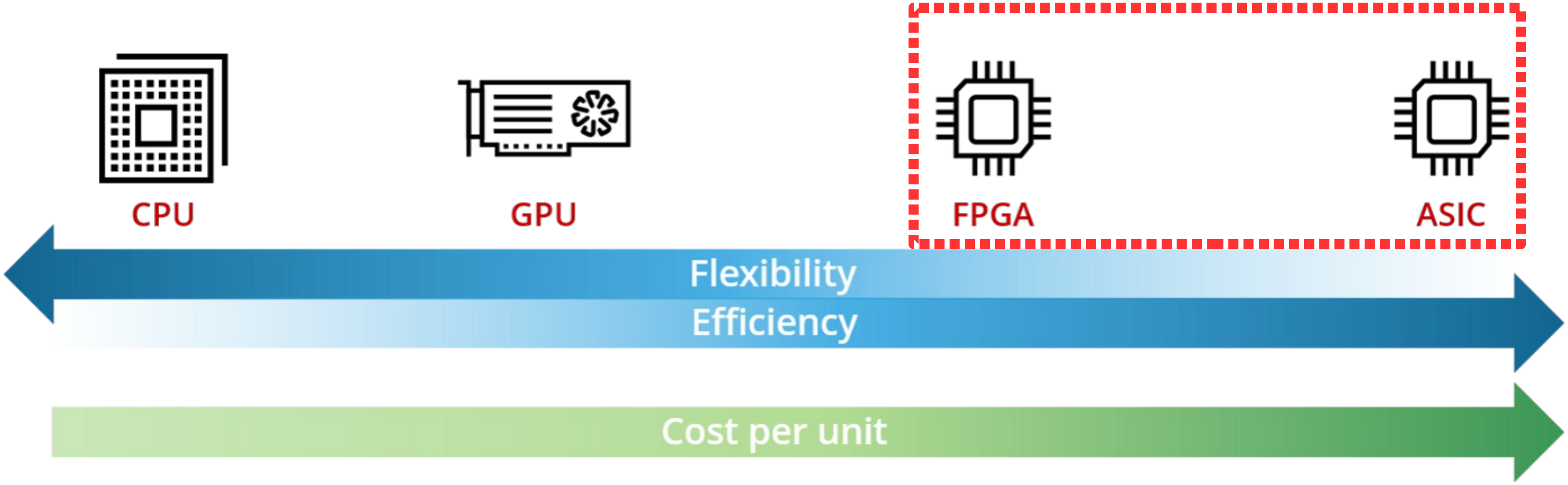
- Lecture 1: Domain Specific Architectures
- Lecture 2: Kernel computation
- Lecture 3: Data-flow techniques
- Lecture 4: DNN accelerators architectures

Optimizing DNN Computation



- Transformation to leverage
  - Optimized libraries
  - Reduce the number of multiplications

Optimizing DNN Computation



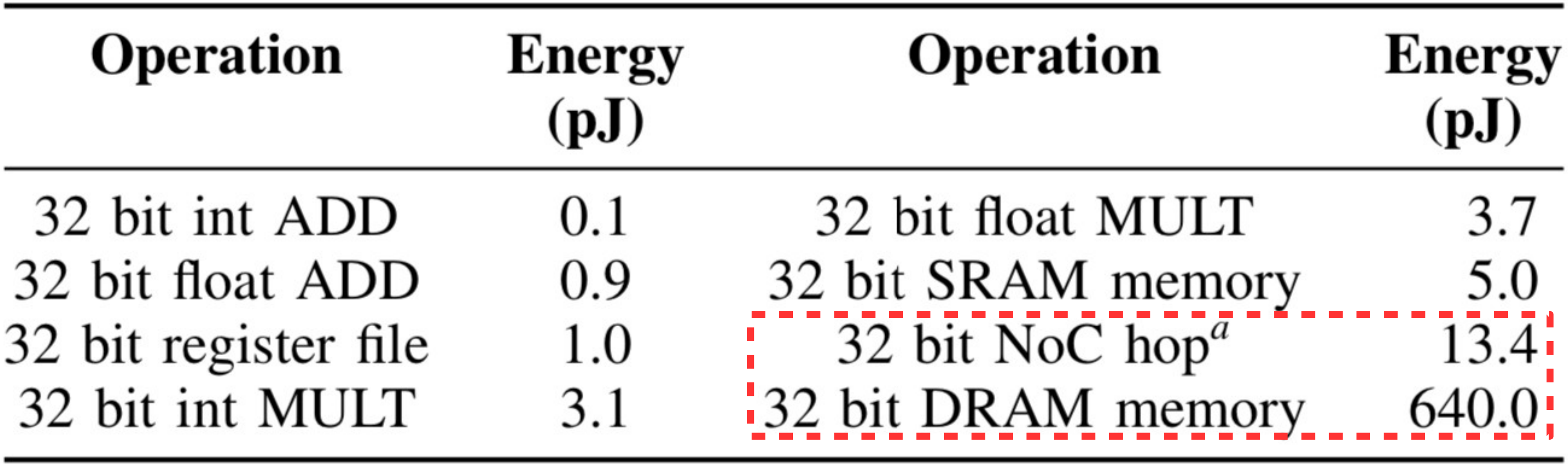
- Designing specialized HW
  - To continue improving perfomance and energy efficiency in important computational domains
  - Hennessy & Patterson, Turing Award Lectuter 2018

Design Space

- DS for DNN HW Accelerators is large
  - No constraints on the order of execution of MACs
  - HW designer has a lot of flexibility
  - Constraints
    - Number of PEs
    - Storage capacity

General Design Objective

- Reduce data movement
  - Always valid recipe
  - Memory access much more power hangry than MACs

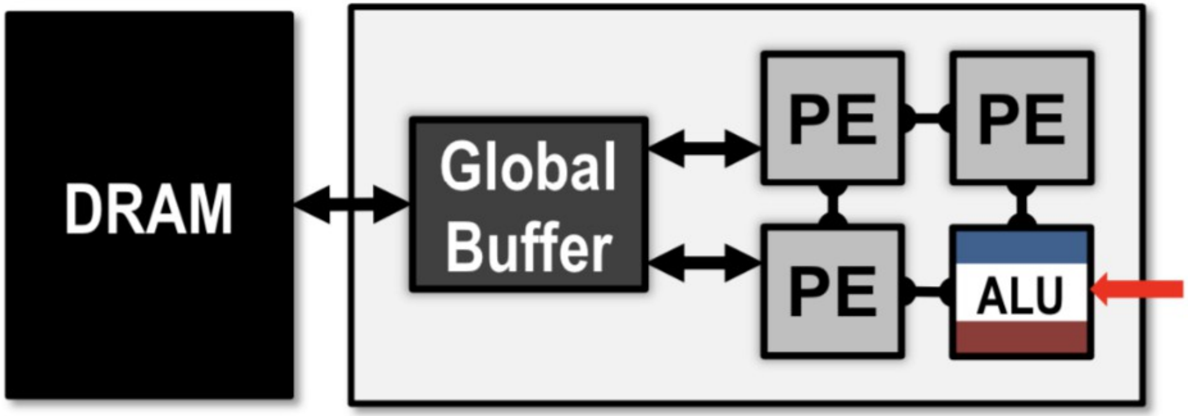


Operation	Energy (pJ)	Operation	Energy (pJ)
32 bit int ADD	0.1	32 bit float MULT	3.7
32 bit float ADD	0.9	32 bit SRAM memory	5.0
32 bit register file	1.0	32 bit NoC hop[a]	13.4
32 bit int MULT	3.1	32 bit DRAM memory	640.0

45 nm CMOS process

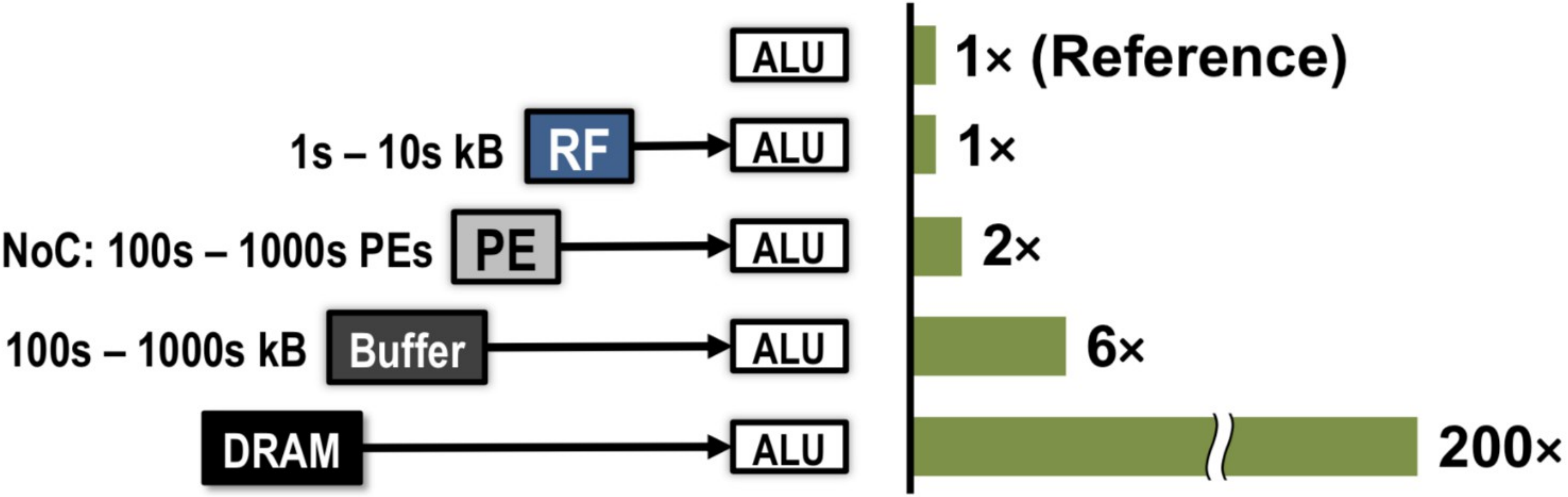
[M. Horowitz, “Energy table for 45nm process,” Stanford VLSI Wiki.]

The Cost of Moving Data



fetch data to run a MAC here

Normalized Energy Cost*



For a 65 nm technology node

Reduce Data Movement

- Two main ways
  - Reducing the number of times values are moved from source with high energy cost
    - DRAM, large on-chip buffer
  - Reducing the cost of moving each value
    - Reducing the data bit-width

Onother Design Objective

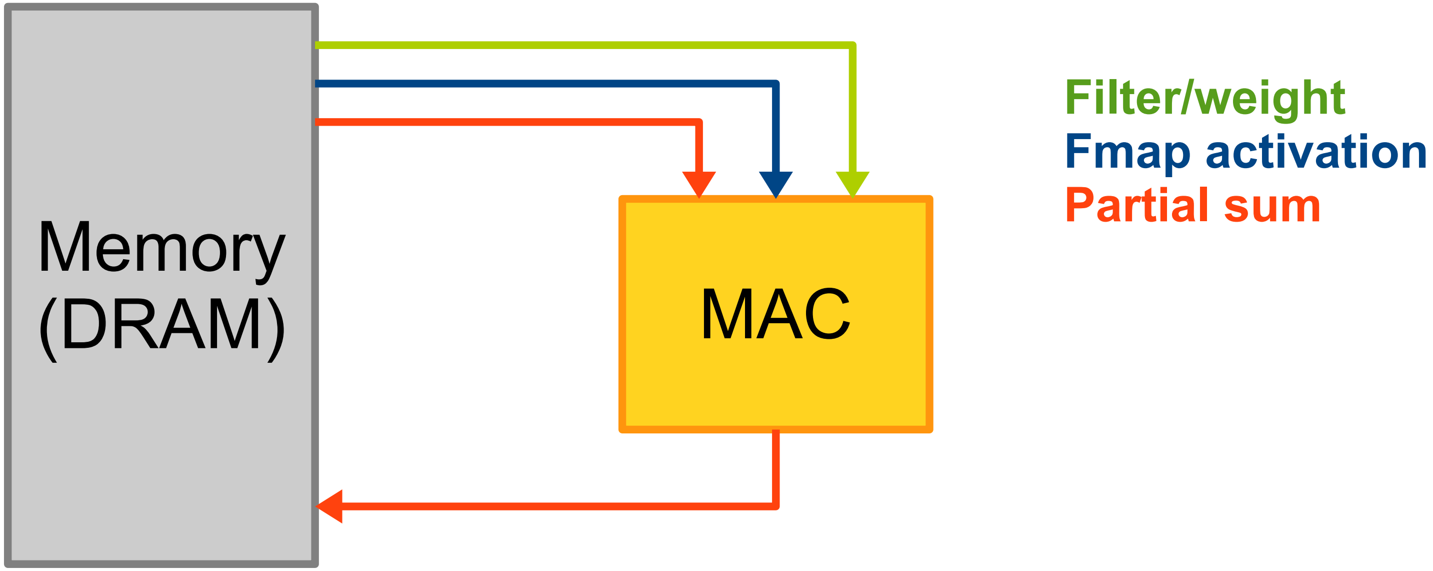
- Allocating work to as many PEs as possible
  - So as they can work in parallel
- Minimizing the PEs idle cycles
  - Assuring enough bandwidth to deliver data to PEs

Key Properties of DNN to Leverage

- Good news
  - Many MACs per layer w/o constraints in the execution order → **Parallelism exploitation simple to achive**
- Bad news
  - PEs are data hungry
  - Data have to be delivered to PEs → **Data movement issue**

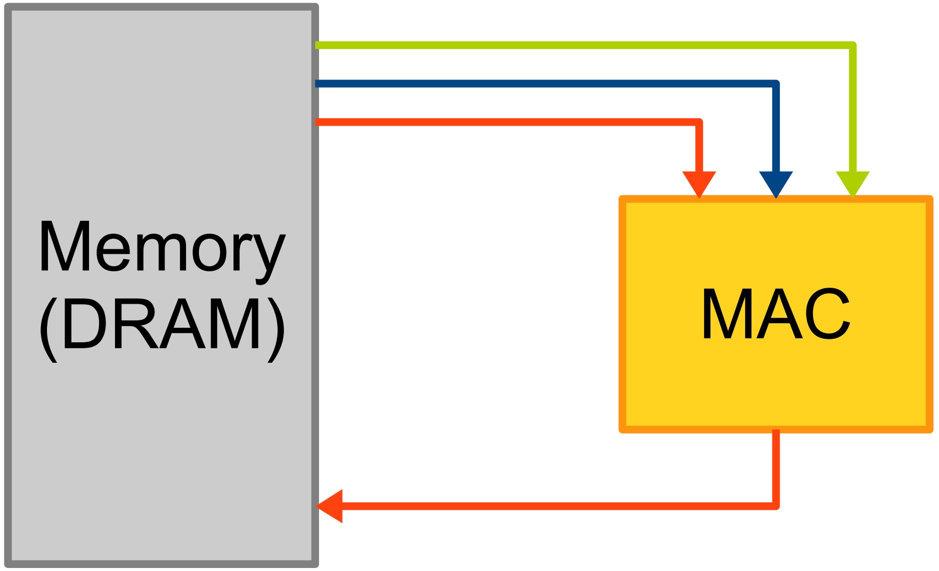
Kernel Operation in DNNs

- Load from memory
  - Filters, input feature map, partial sum
- Store to memory
  - Updated partial sum, output feature map



Memory Bottleneck

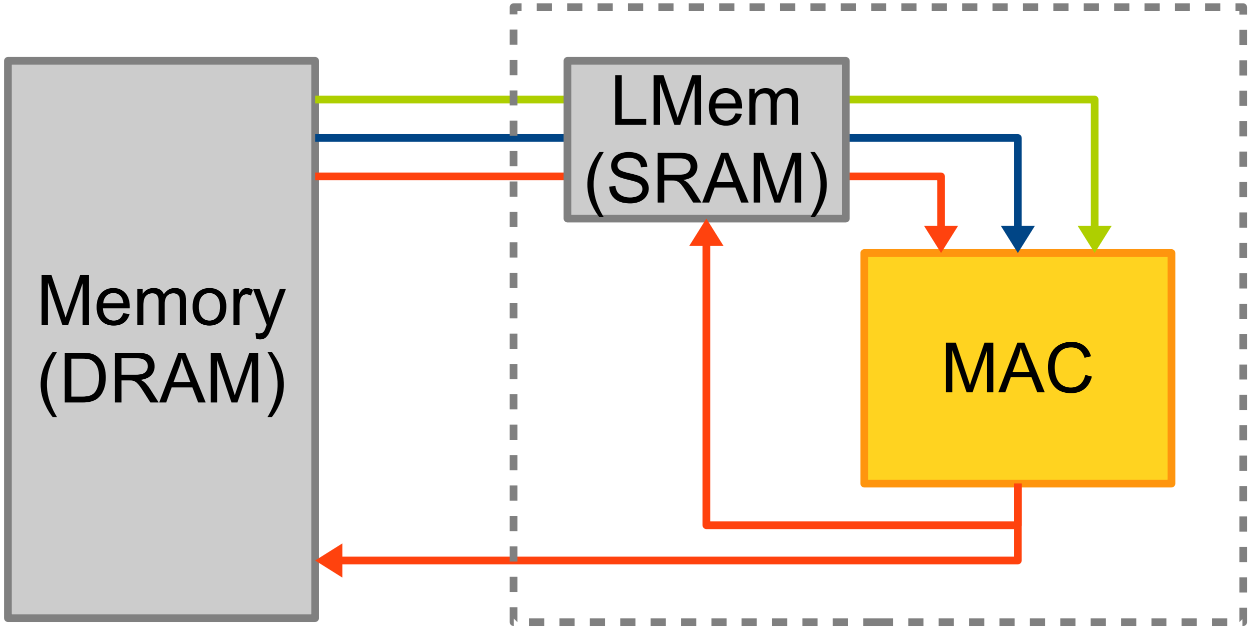
- **Worst case:** all memory accesses are DRAM accesses
  - AlexNet 724M MACs → **2896M DRAM accesses**



Filter/weight
Fmap activation
Partial sum

Local Memory

- Use small, fast and more energy-efficient local memory for storing reused data



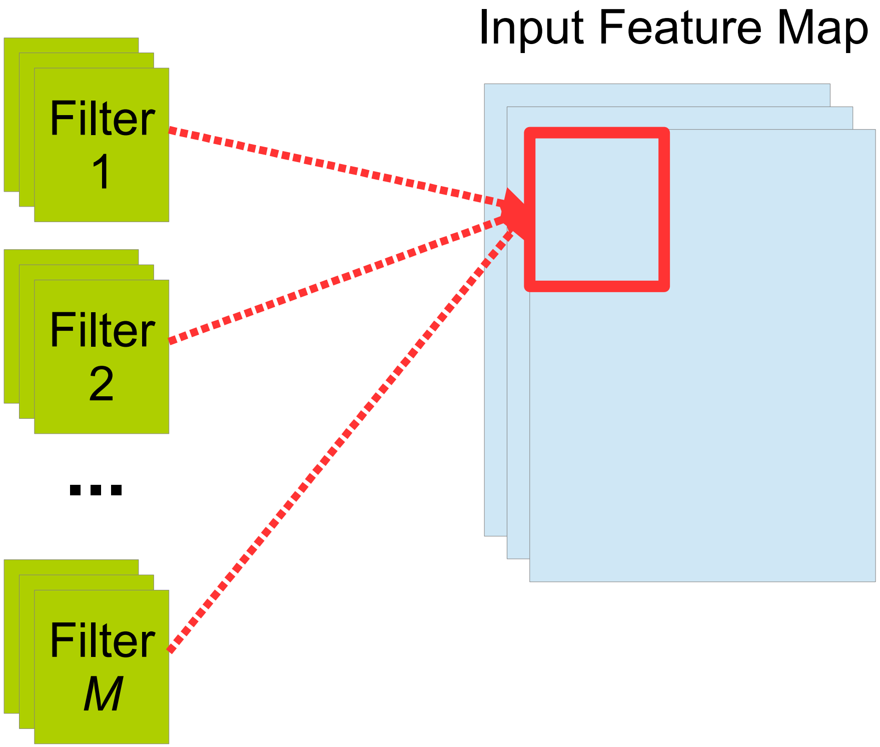
Filter/weight
Fmap activation
Partial sum

Data Movement Issue

- Exploiting ***data reuse***
  - The same piece of data is often reused for multiple MACs
- Three forms of data reuse
  - Input feature map reuse
  - Filter reuse
  - Convolutional reuse

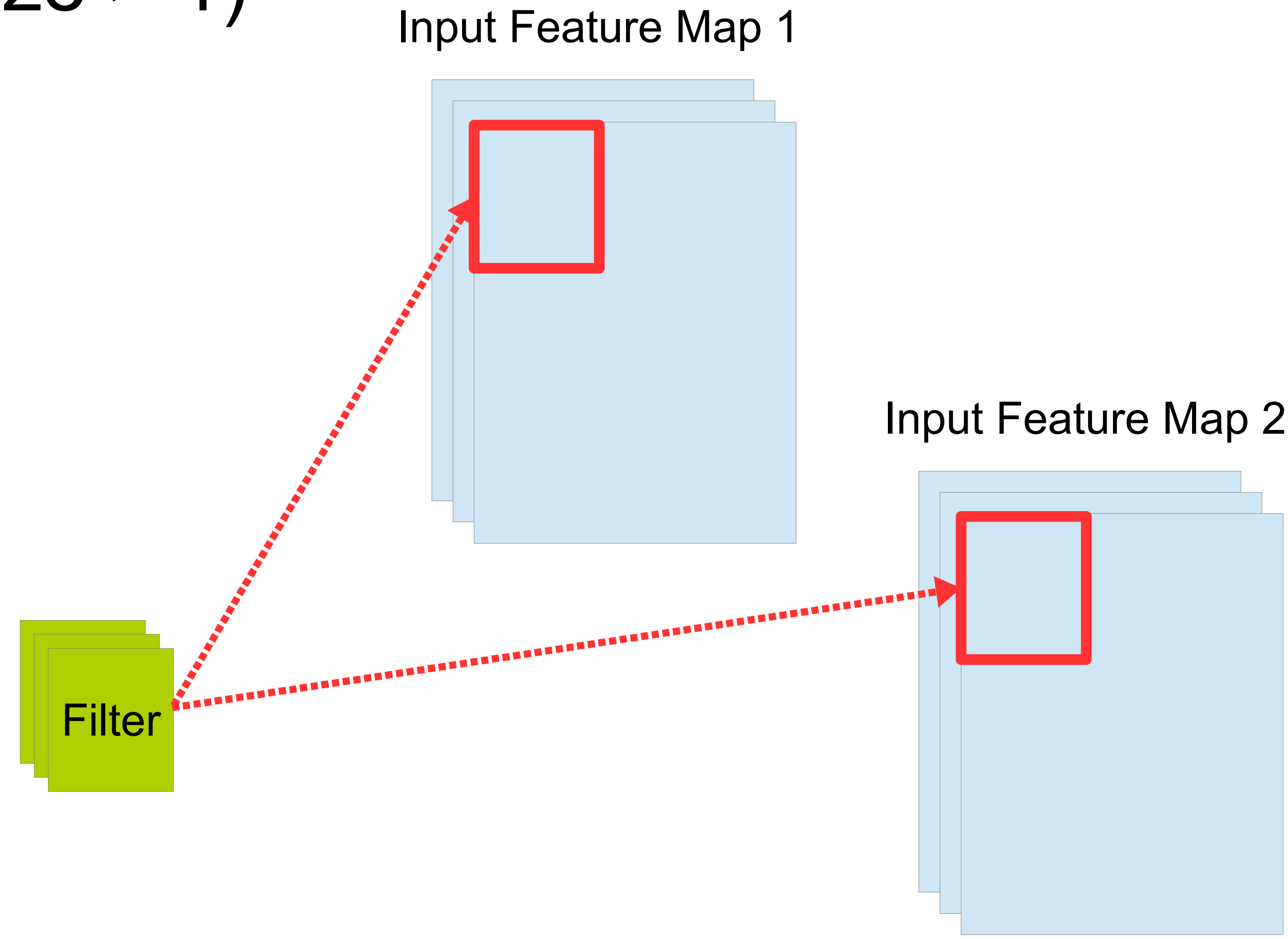
Input Feature Map Reuse

- The same ifmap is used by several filters
  - Different filters applied to the same ifmap
  - Each input activation is reused *M* times



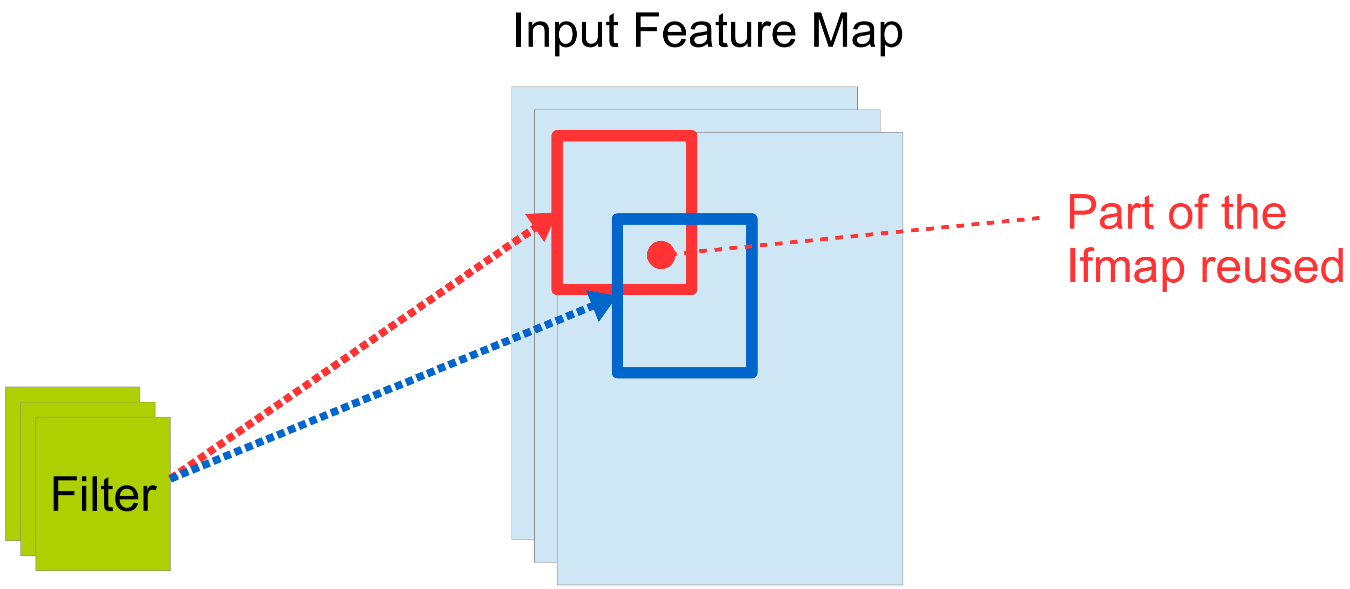
Filter Reuse

- The same filter is applied to different ifmap (batch size > 1)



Convolutional Reuse

- The same filter is applied to different parts of the ifmap
  - The filter is reused
  - Part of the ifmap is reused



Exploiting Data Reuse

- AlexNet
  - Worst case: **2896M** DRAM accesses
  - Best case: **61M** DRAM accesses (47x)
    - If all data reuse is exploited

Data Reuse Types

- Data read once from a large expensive memory
- **Temporal Reuse**
  - Store data to a small cheap memory and reuse data several times
- **Spatial Reuse**
  - Send the same data to multiple PEs and reuse data at distict PEs

The Right Reuse Choice

- Which is the best data reuse choice?
  - It depends by the layer shape!
- DNN layers vary dramaticaly
  - Across different DNNs
  - Within the same DNN
- Optimization must be performed on a layer basis
- HW needs to be able to support different configurations
  - Flexibility *vs.* Cost *vs.* Efficiency dilemma

Terminology

- Mapping
- Dataflow

Mapping

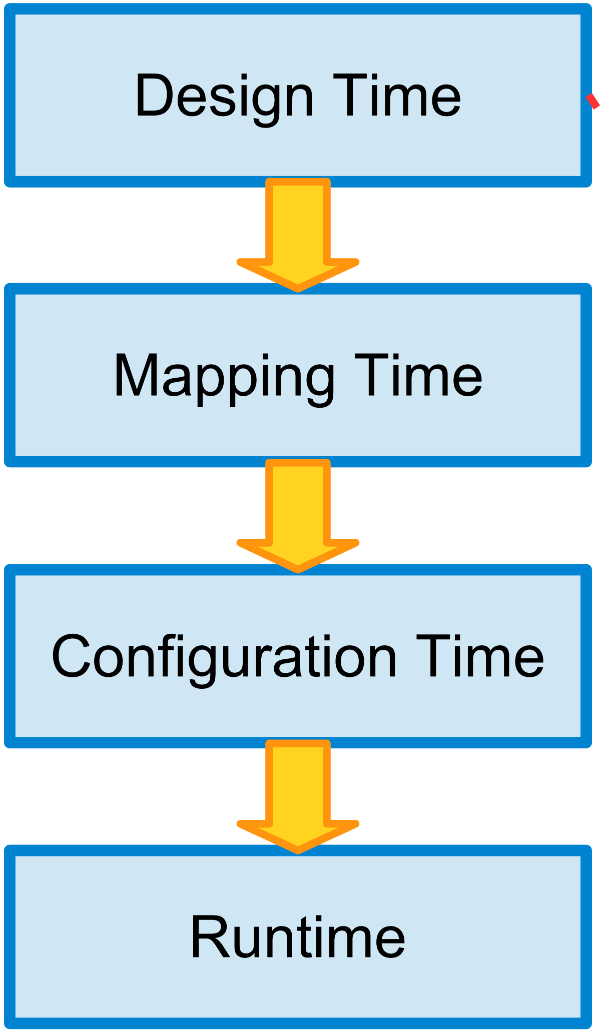
- Configure the HW to minimize energy maintaining high performance
  - Find the optimal ***mapping***
- Mapping
  - Execution order of the MACs
    - **Temporally**: serial order on the same PE
    - **Spatially**: across many parallel PEs
  - How to tile and move data across the memory hierarchy

Dataflow

- Many possible spatio-temporal ordering of MACs
  - Many different mappings theoretically possible
- HW can support a limited number of orderings
- **Dataflow**
  - Rules that determine the possible spatio-temporal orderings (*i.e.*, mappings) supported by the HW

Design & Use of DNN HW Accelerators

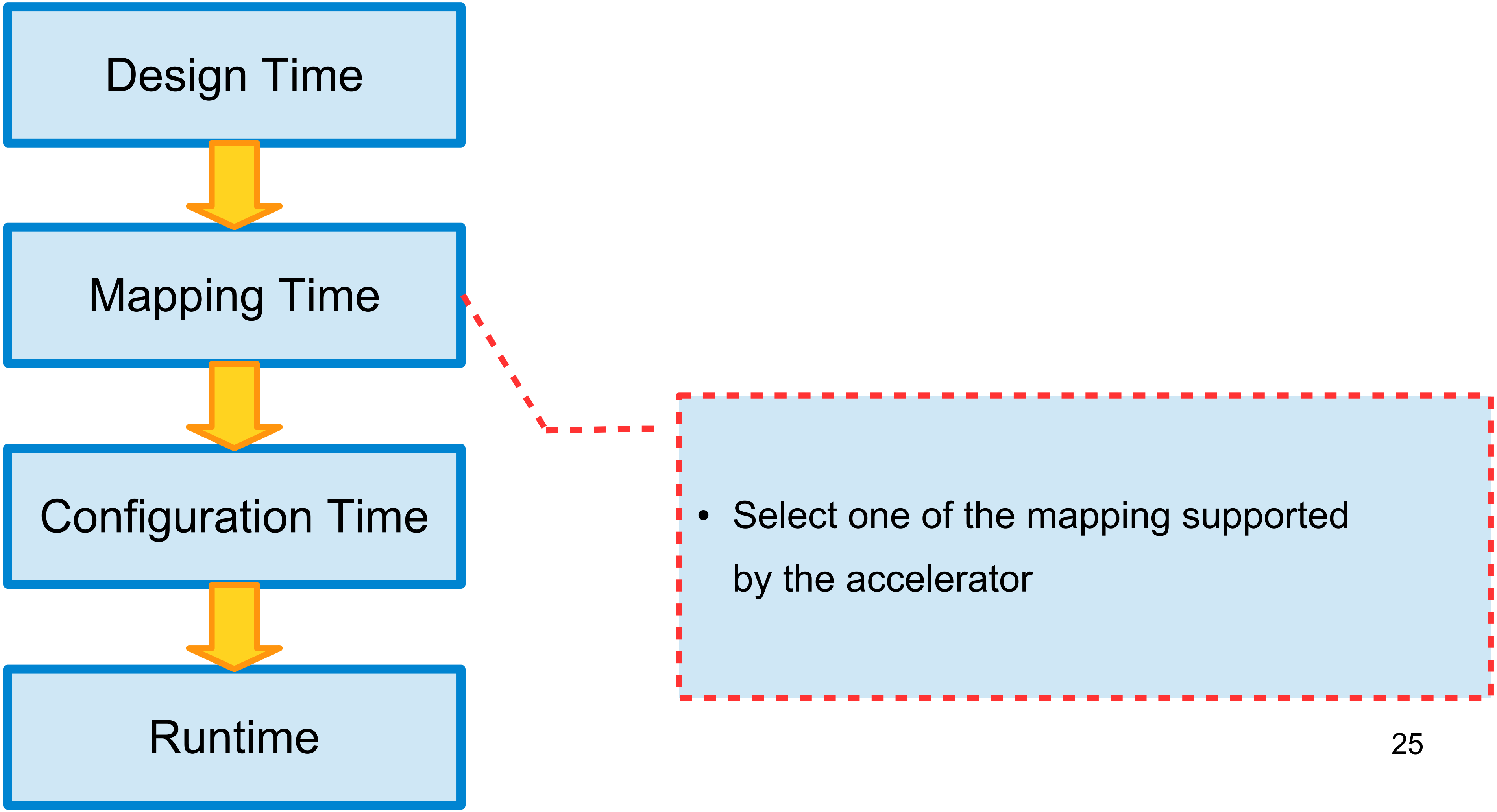
- Steps involved during the design and use of DNN accelerators



- Dataflow/dataflows supported
- Number of PEs
- Number of MACs per PE
- Memory hierarchy (levels and storage)
- Data delivery patterns supported by the NoC

Design & Use of DNN HW Accelerators

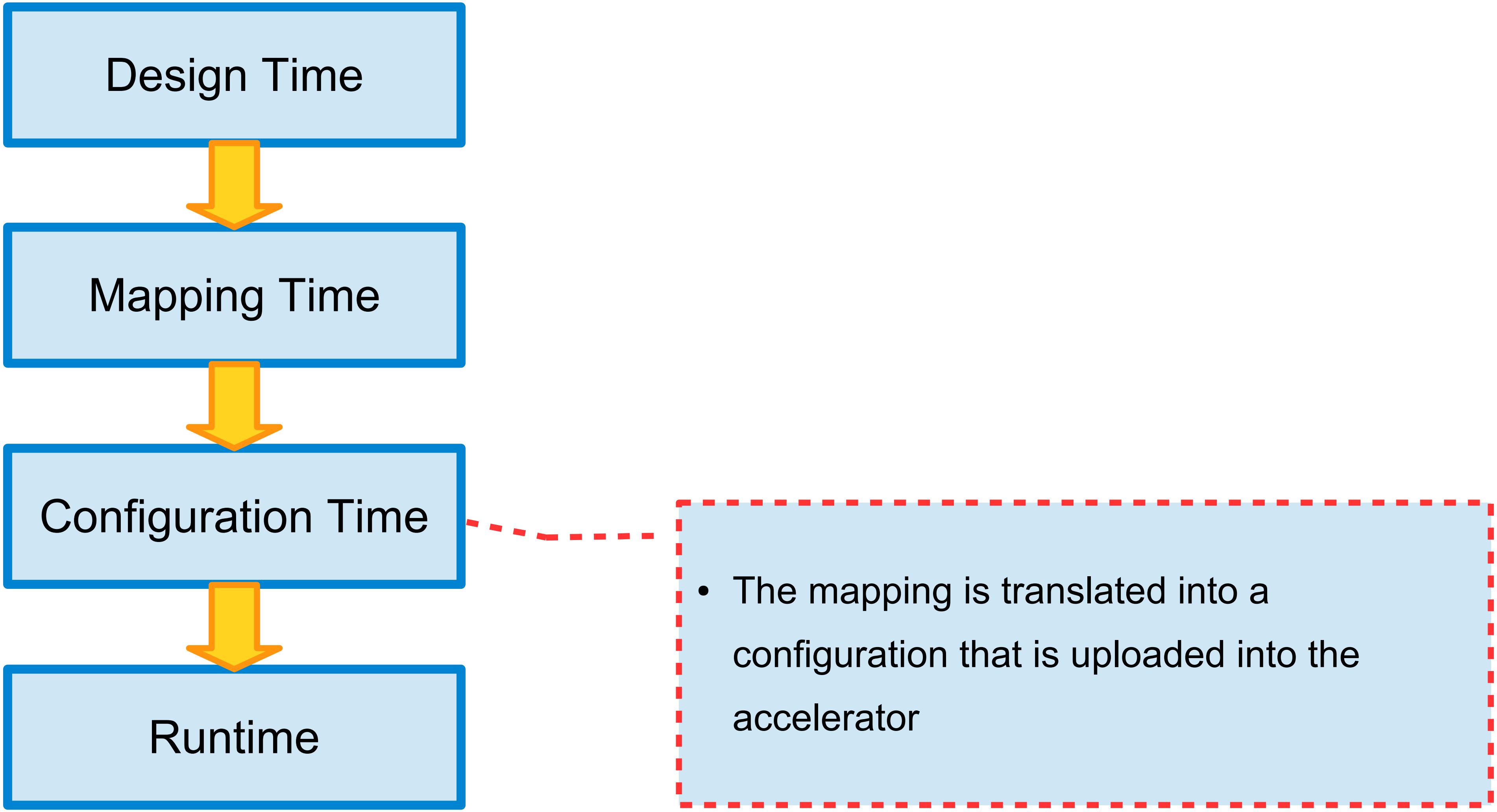
- Steps involved during the design and use of DNN accelerators



- Select one of the mapping supported by the accelerator

Design & Use of DNN HW Accelerators

- Steps involved during the design and use of DNN accelerators



Design Time

Mapping Time

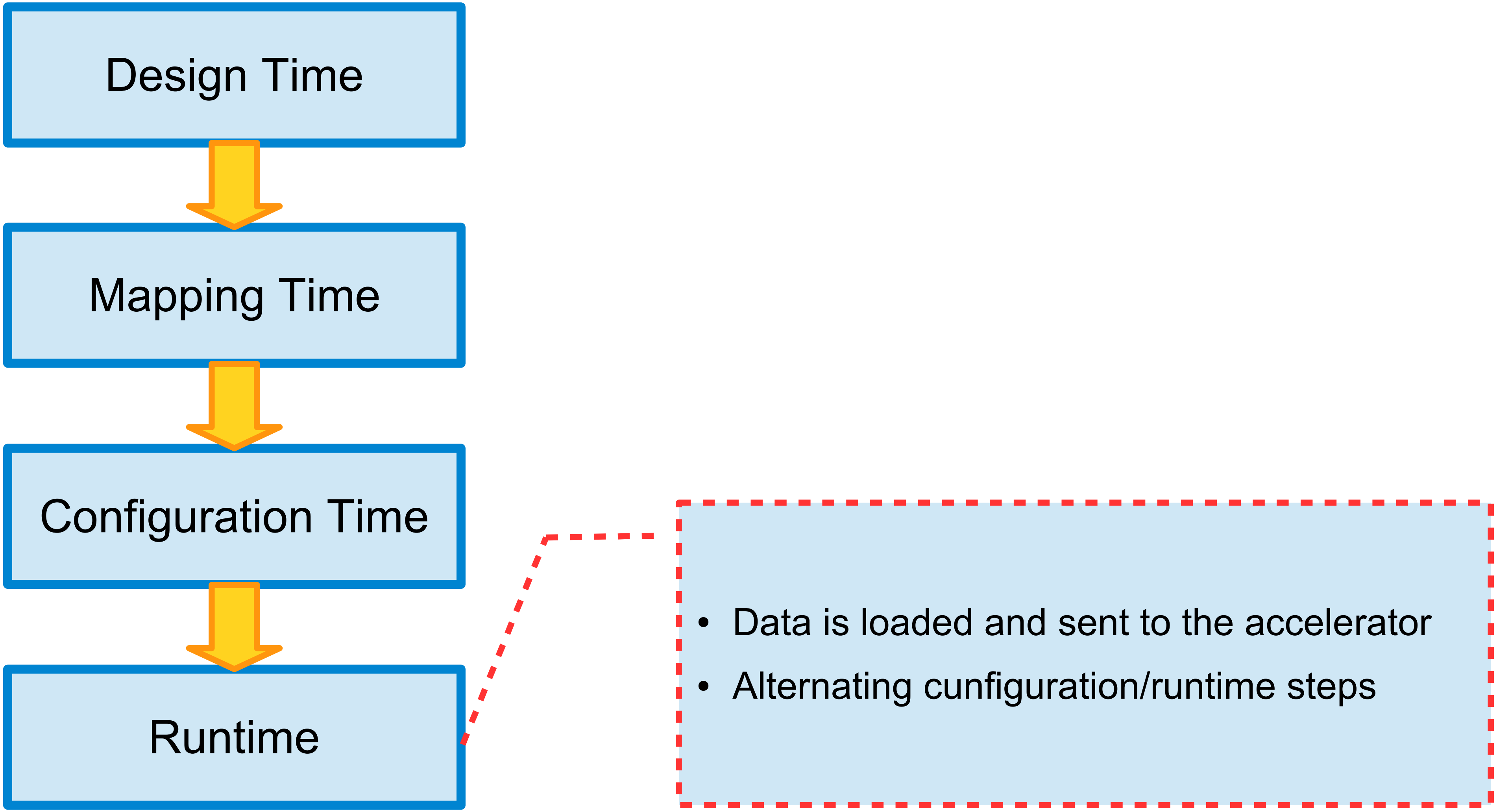
Configuration Time

Runtime

- The mapping is translated into a configuration that is uploaded into the accelerator

Design & Use of DNN HW Accelerators

- Steps involved during the design and use of DNN accelerators



Design Time

Mapping Time

Configuration Time

Runtime

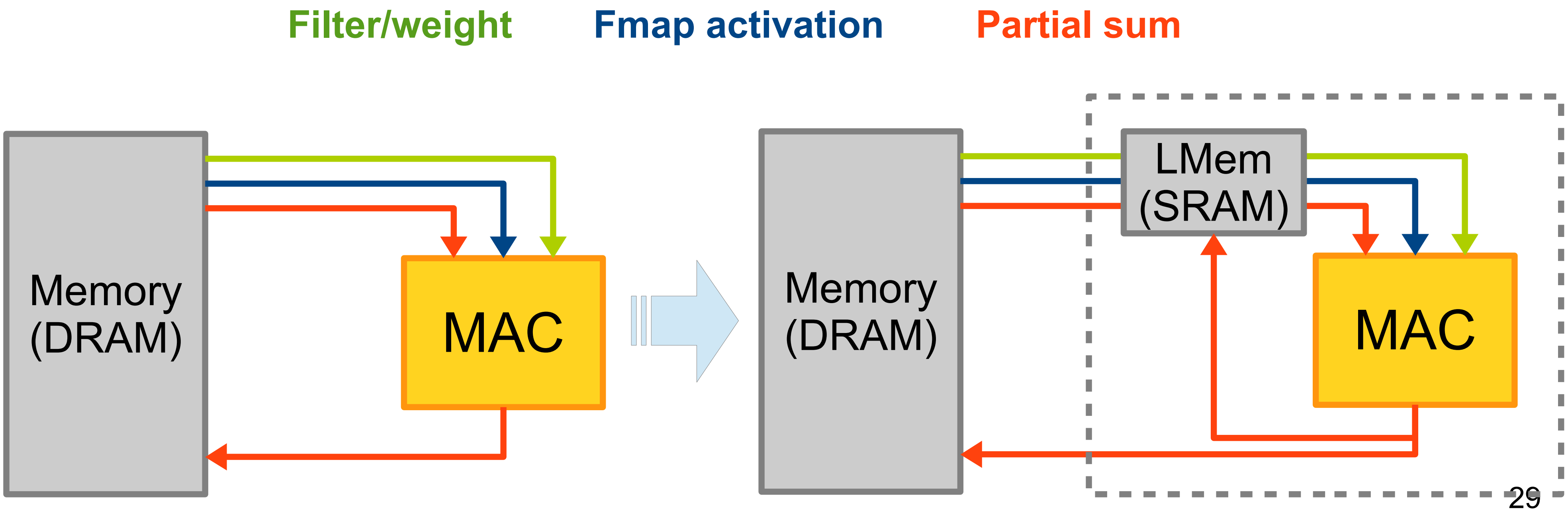
- Data is loaded and sent to the accelerator
- Alternating cunfiguration/runtime steps

Exploiting Data Reuse

- Temporal reuse
- Spatial reuse

Temporal Reuse

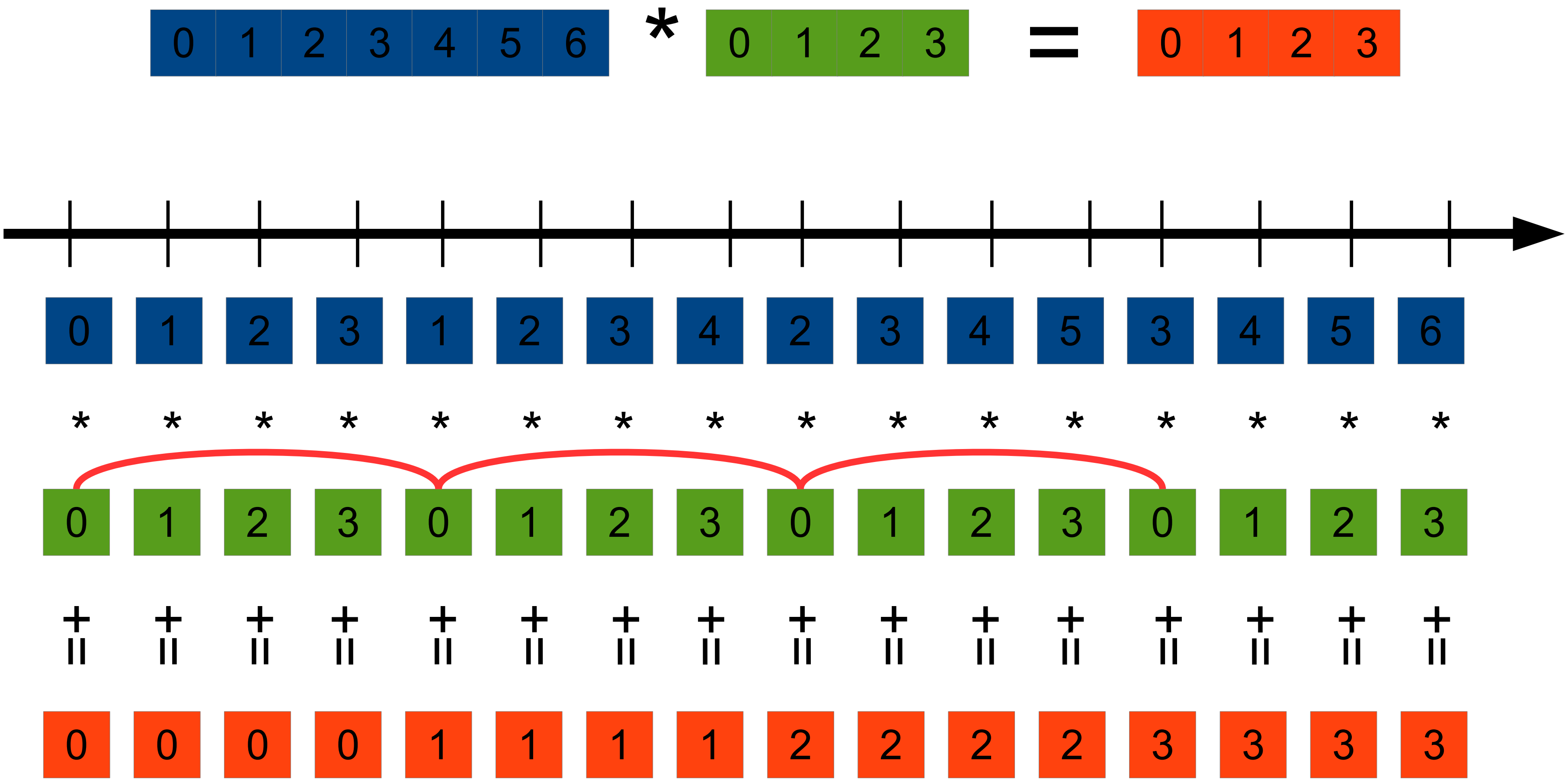
- The same data is used more than once by the same PE
- Exploited by adding itermediate memory level



Temporal Reuse Distance

- Intermediate memory size < Main memory size
  - Data into the intermediate memory may be replaced by new data
  - Data can be reused if it is not replaced by new data
- Temporal Reuse Distance
  - Number of data accesses in between the access to the same data
  - Depends on operation ordering

Reuse Distance

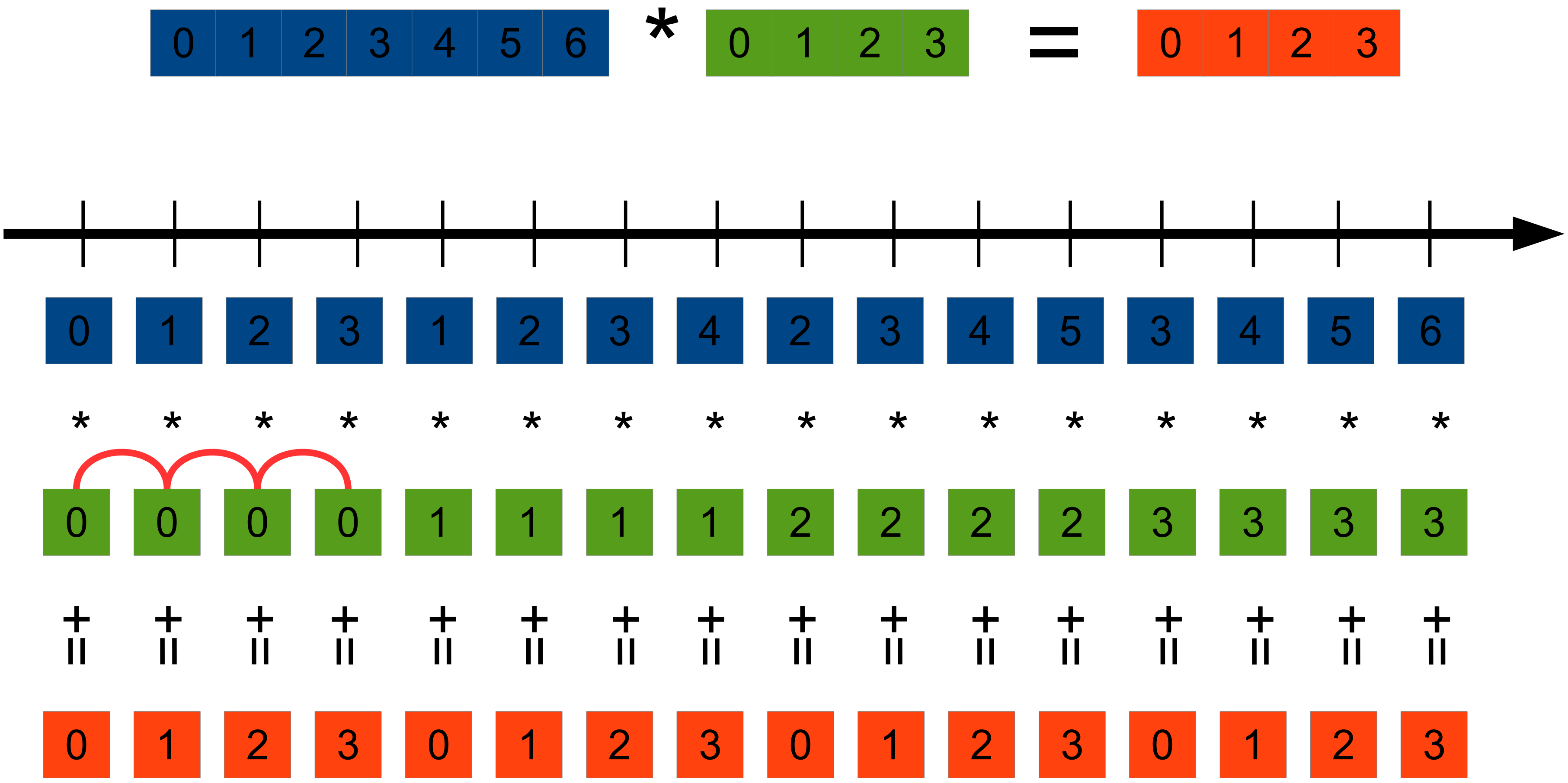


0 1 2 3 4 5 6 * 0 1 2 3 = 0 1 2 3

0	1	2	3	1	2	3	4	2	3	4	5	3	4	5	6
*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
+ =	+ =	+ =	+ =	+ =	+ =	+ =	+ =	+ =	+ =	+ =	+ =	+ =	+ =	+ =	+ =
0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3

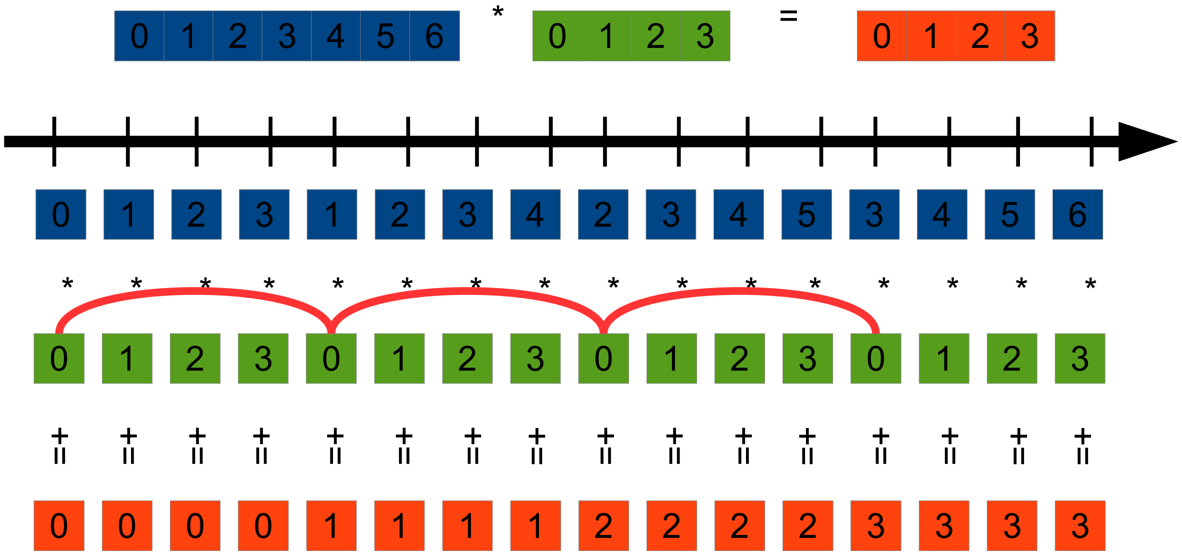
Weight Reuse Distance = 4

Reuse Distance

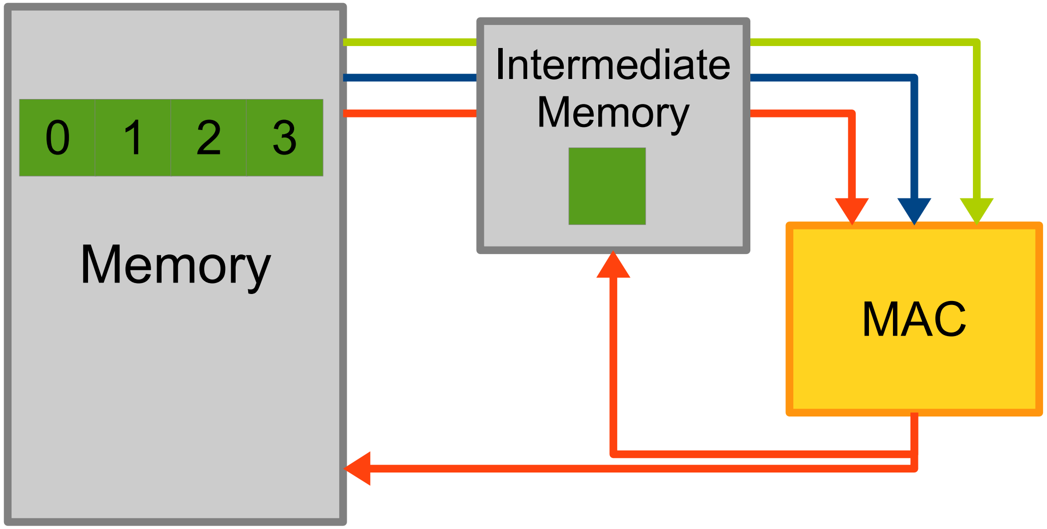


Weight Reuse Distance = 1

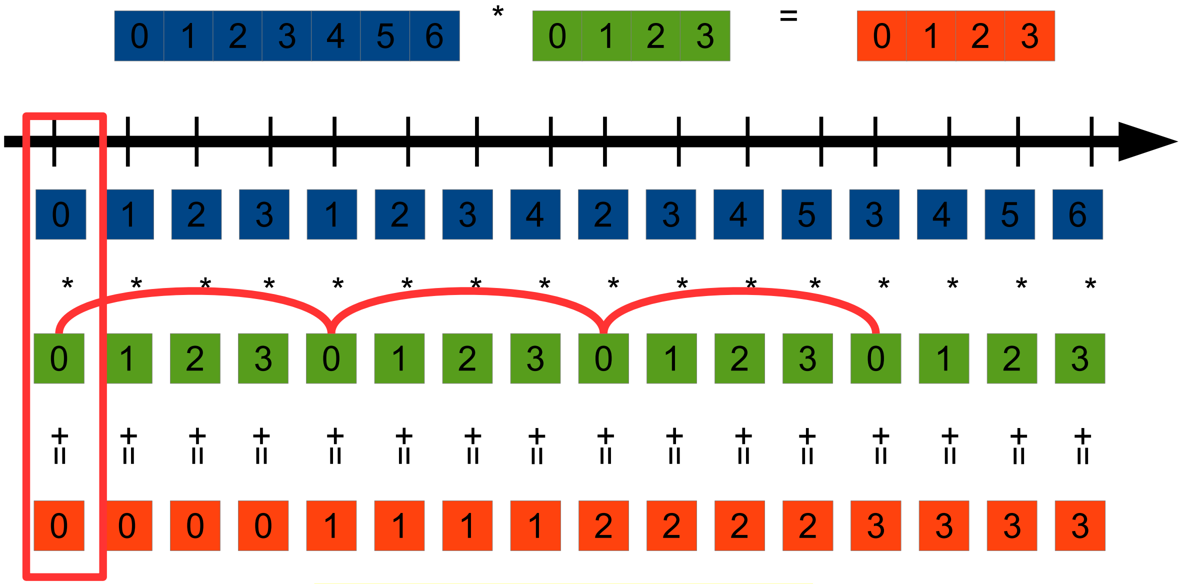
Reuse Distance *vs.* Storage Capacity



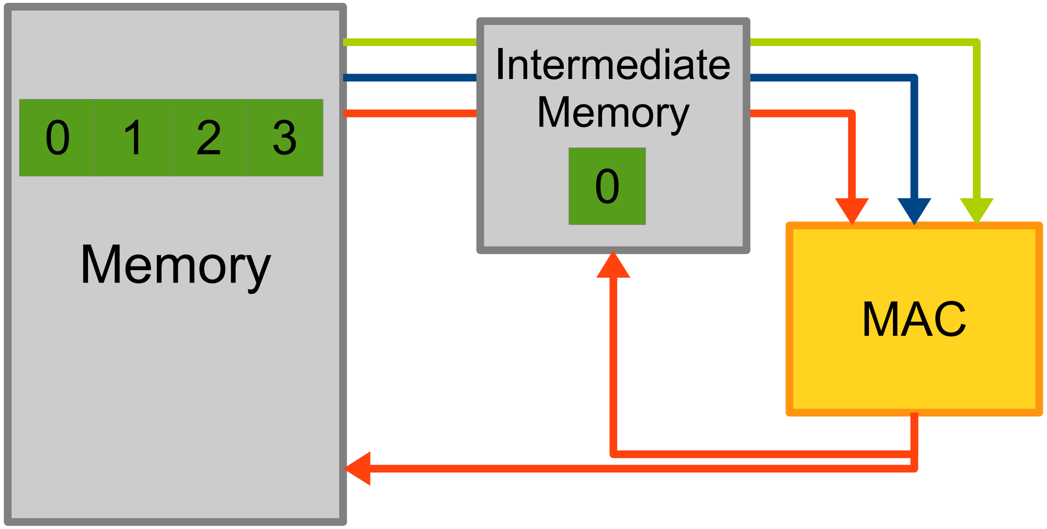
Weight Reuse Distance = 4



Reuse Distance *vs.* Storage Capacity



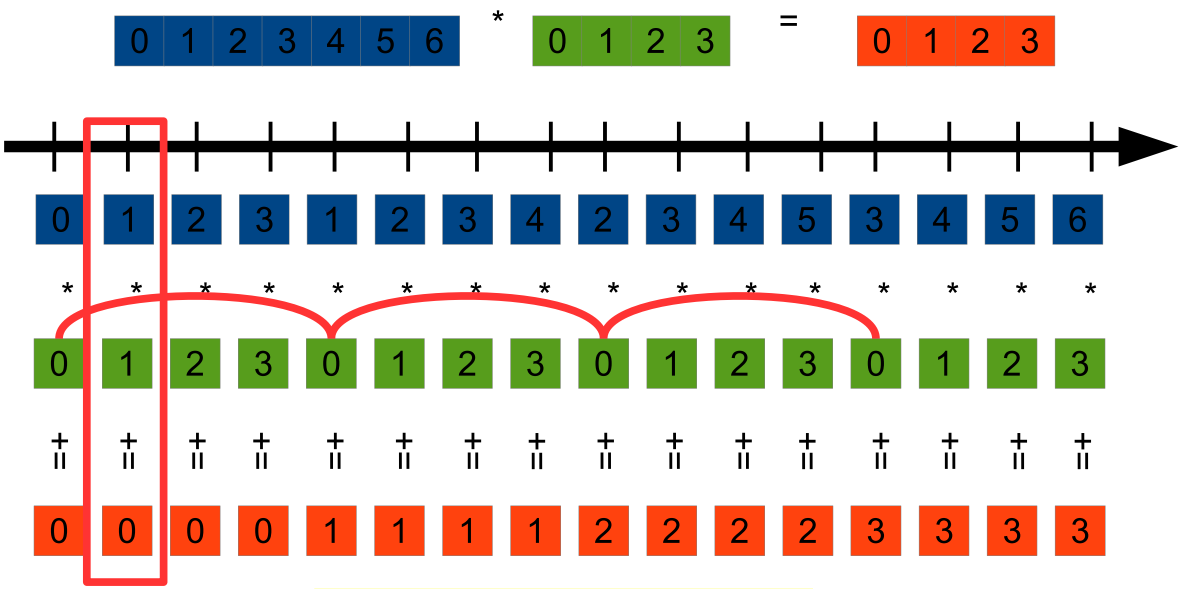
Weight Reuse Distance = 4



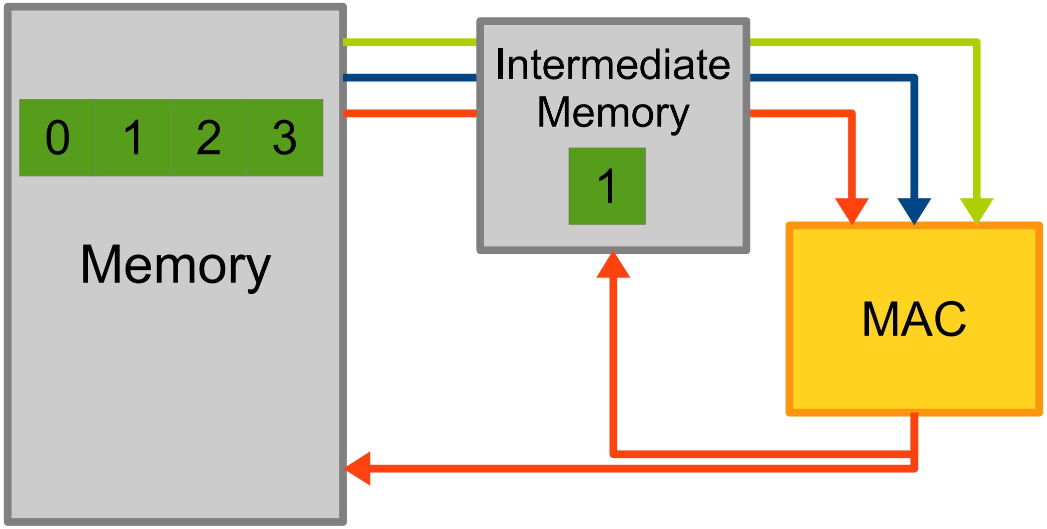
Weight 0 fetched from Memory

(No reuse)

Reuse Distance *vs.* Storage Capacity

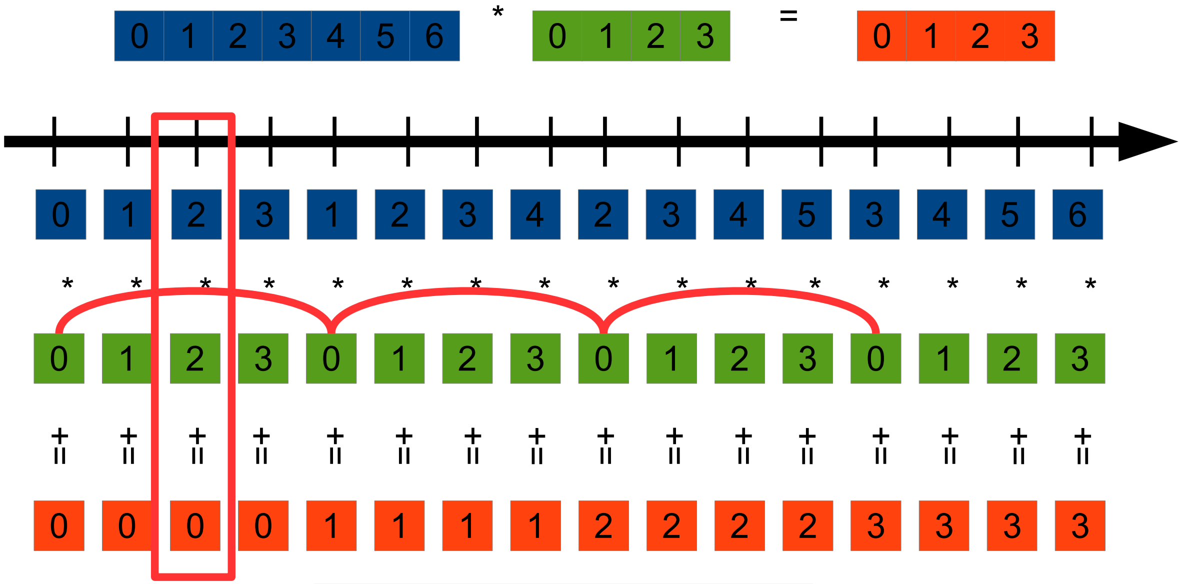


Weight Reuse Distance = 4

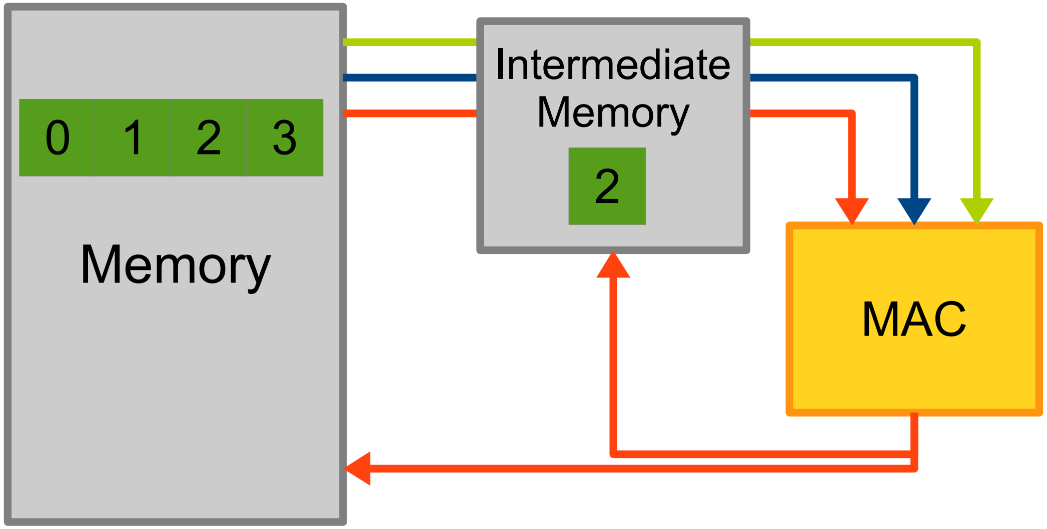


Weight 1 fetched from Memory
(No reuse)

Reuse Distance *vs.* Storage Capacity

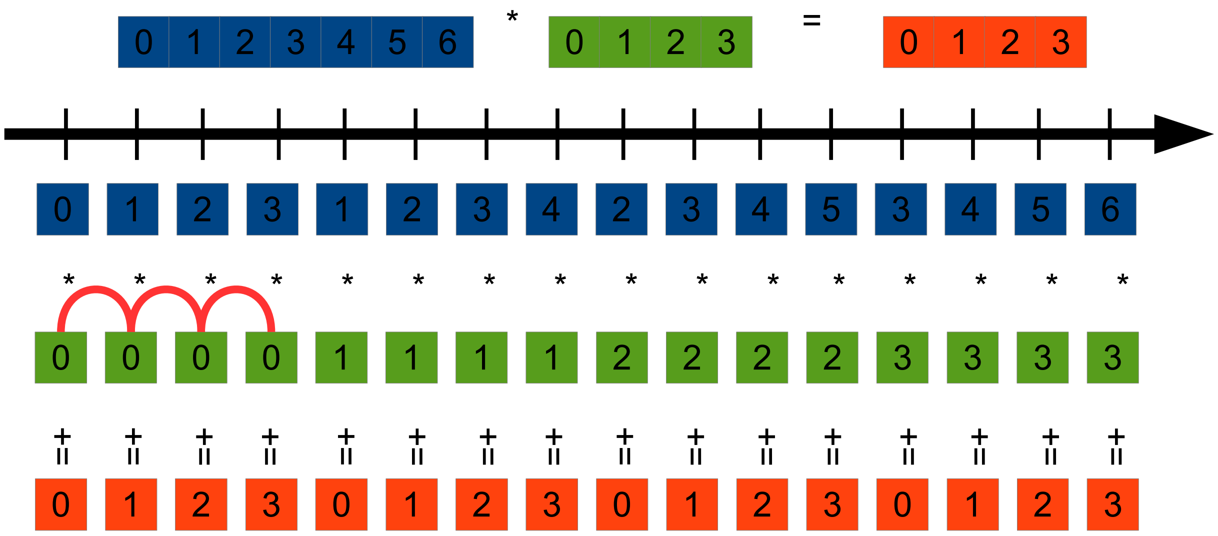


Weight Reuse Distance = 4

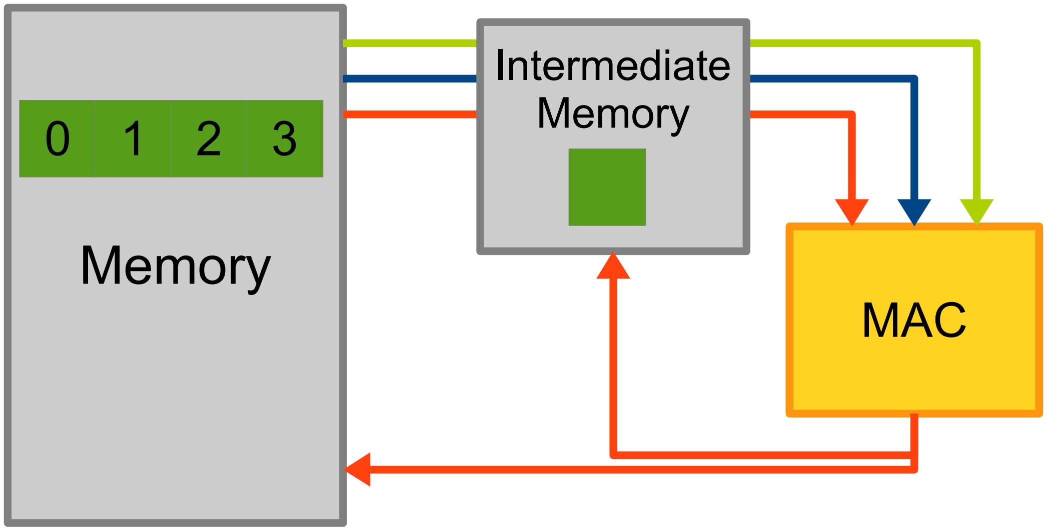


Weight 2 fetched from Memory
(No reuse)

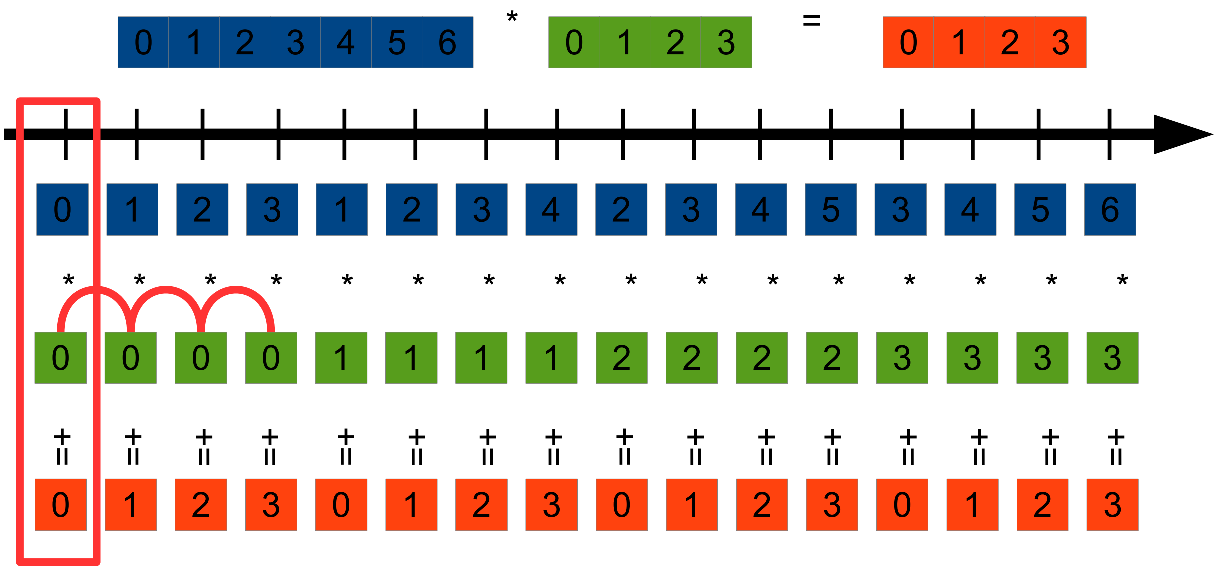
Reuse Distance *vs.* Storage Capacity



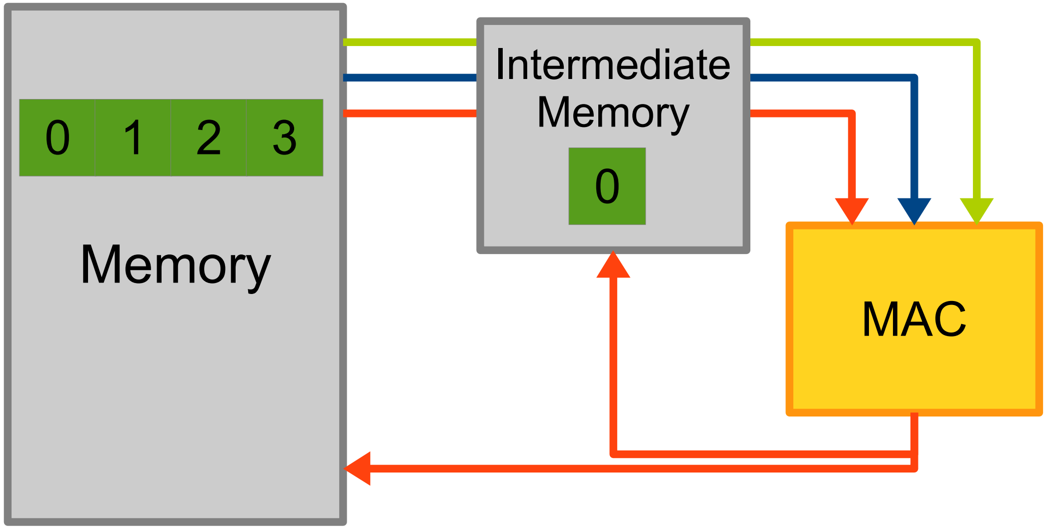
Weight Reuse Distance = 1



Reuse Distance *vs.* Storage Capacity

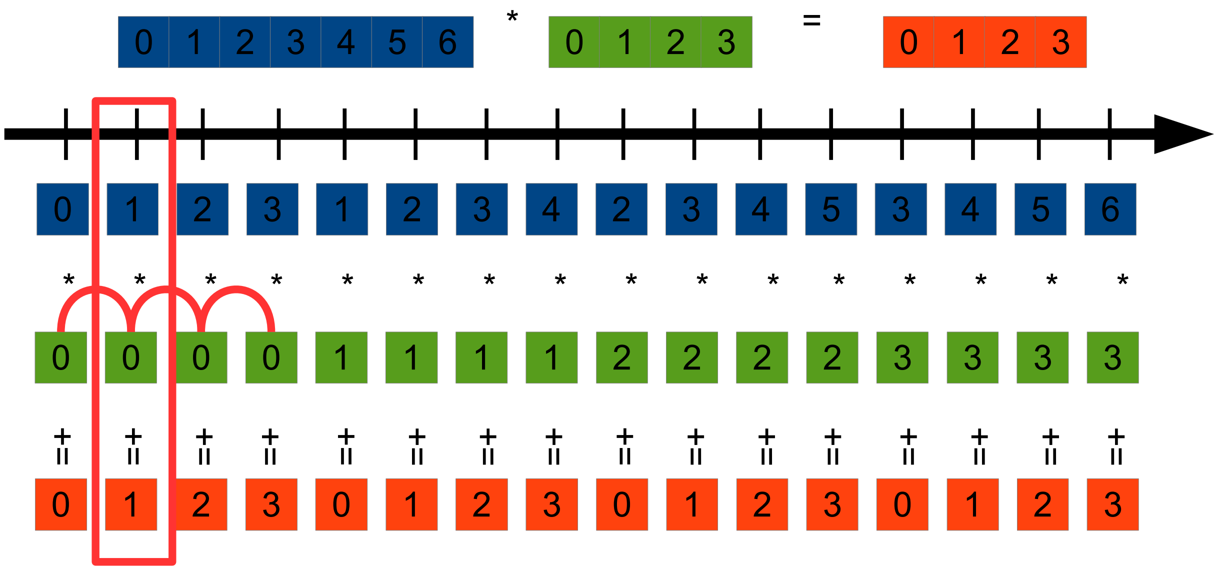


Weight Reuse Distance = 1

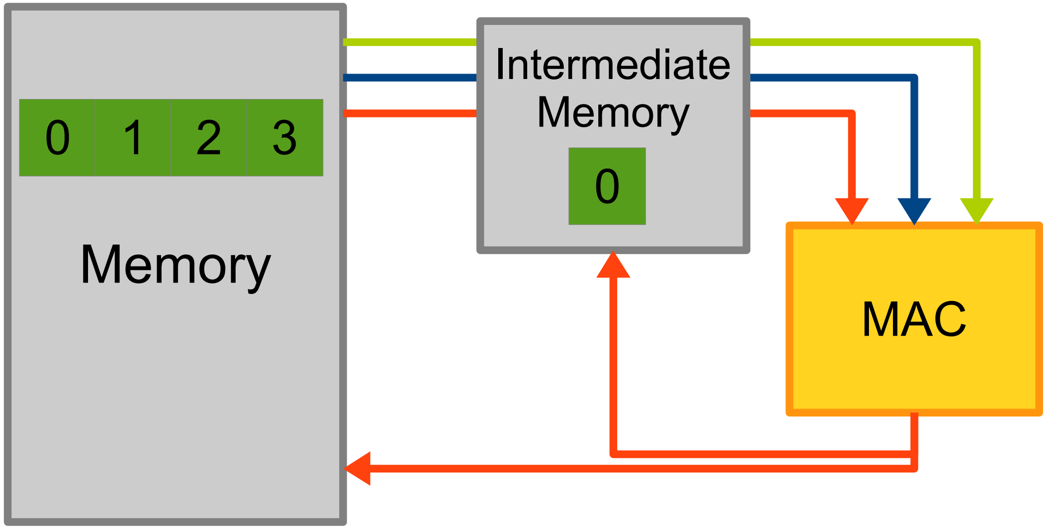


Weight 0 fetched from Memory
(No reuse)

Reuse Distance *vs.* Storage Capacity

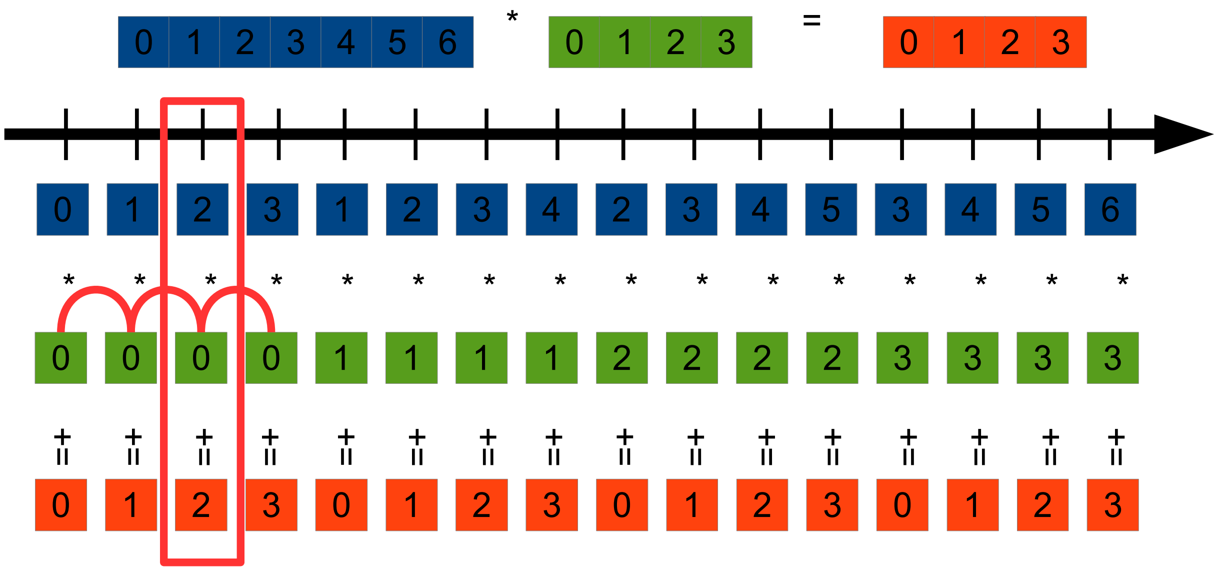


Weight Reuse Distance = 1

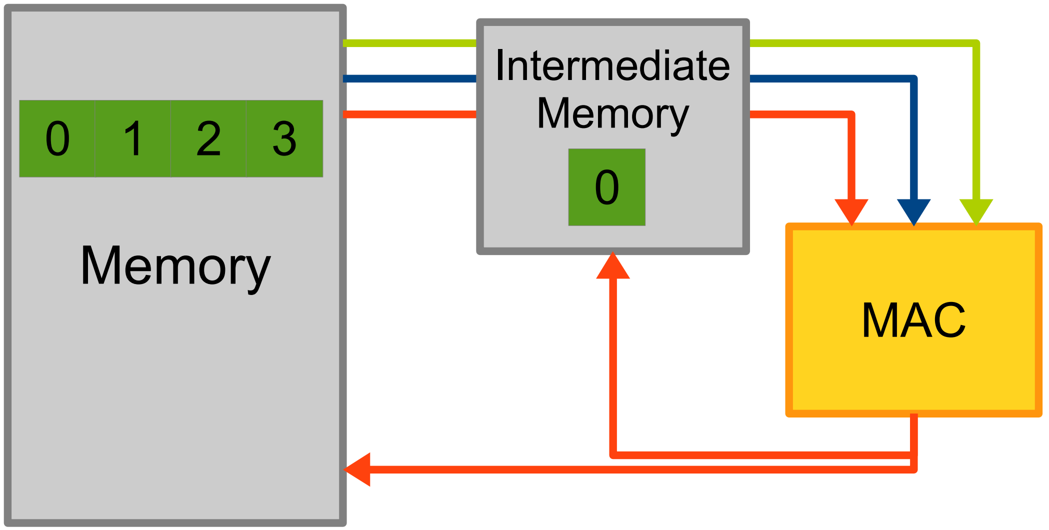


Weight 0 fetched from Intermediate Memory (Reuse)

Reuse Distance *vs.* Storage Capacity

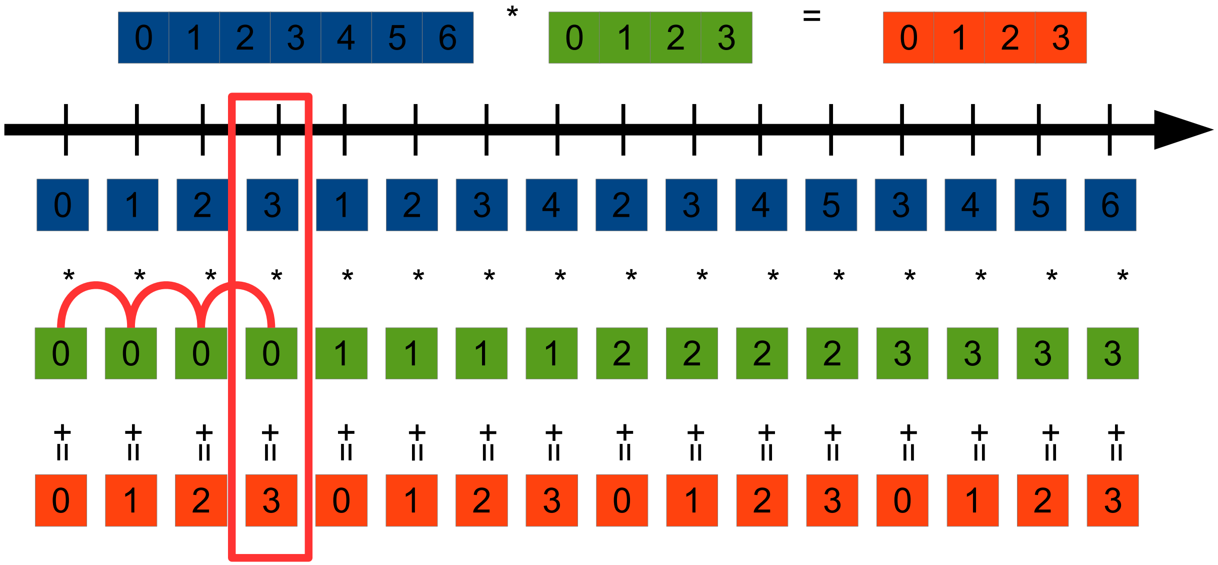


Weight Reuse Distance = 1

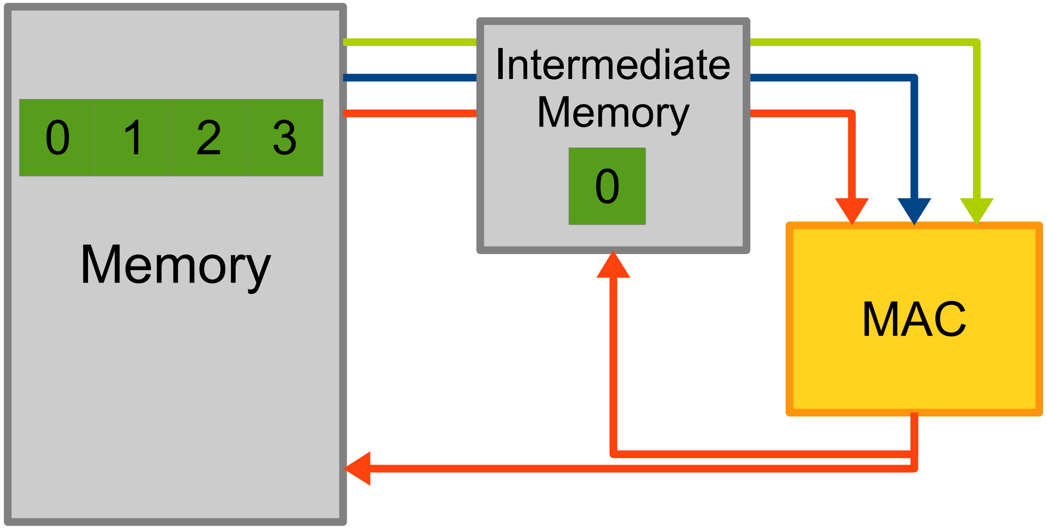


Weight 0 fetched from Intermediate Memory (Reuse)

Reuse Distance *vs.* Storage Capacity



Weight Reuse Distance = 1



Weight 0 fetched from Intermediate Memory (Reuse)

Temporal Reuse – Summary

- If temporal reuse distance for a data type is smaller than or equal to the storage capacity of the intermediate memory level, temporal reuse can be exploited
  - Storage capacity of intermediate memory limits the maximum reuse distance where temporal reuse can be exploited
  - Increase intermediate memory capacity to improve temporal reuse → Average cost per access increases

Spatial Reuse

- The same data value is used by more PEs
- Exploitation
  - Data is read once from memory and multicast to all the PEs
- Advantages
  - Reduce the number of memory access
  - Reduce the required bandwidth from the memory

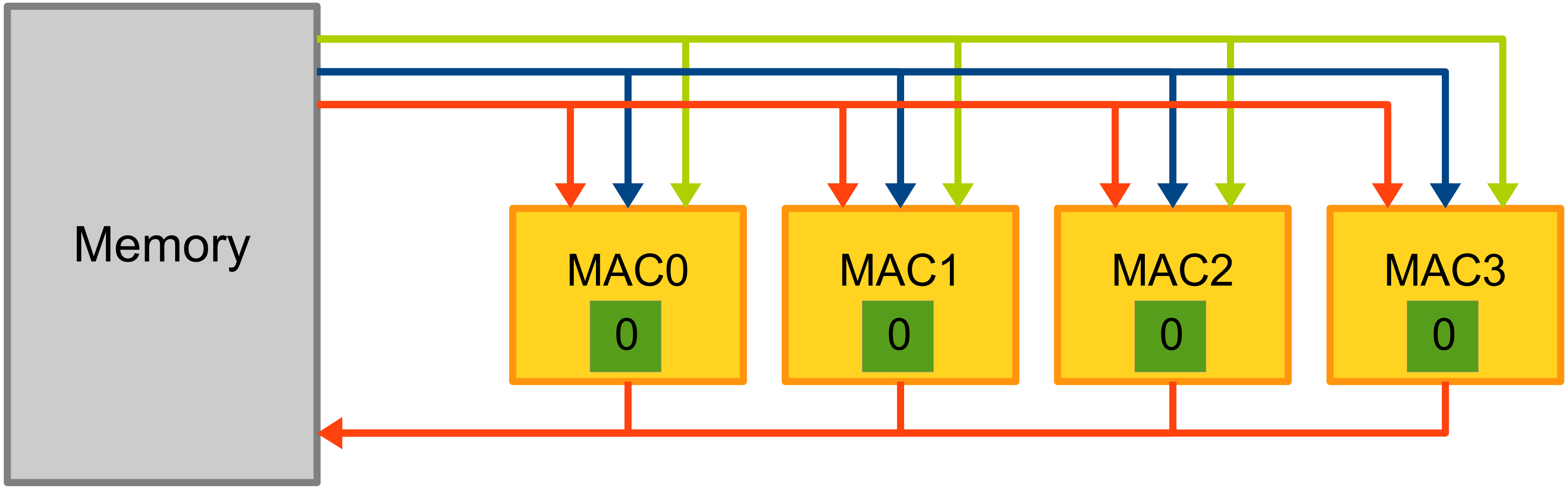
Spatial Reuse and Storage Capacity

- The same data value is used by more PEs
- Two cases
  - PE without storage capacity
    - Data must arrive at the PE at the same clock cycle
  - PE with storage capacity
    - Data must arrive the PE within a given *reuse distance*

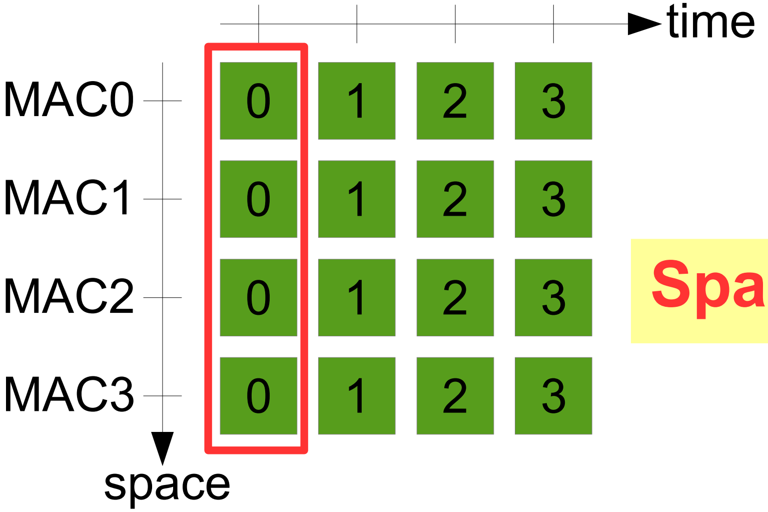
Spatial Reuse Distance

- Max number of data accesses in between any pair of PEs that access the same data value
  - Depends on the ordering of operations

Spatial Reuse *vs.* Storage Capacity



0 1 2 3 4 5 6 * 0 1 2 3 = 0 1 2 3



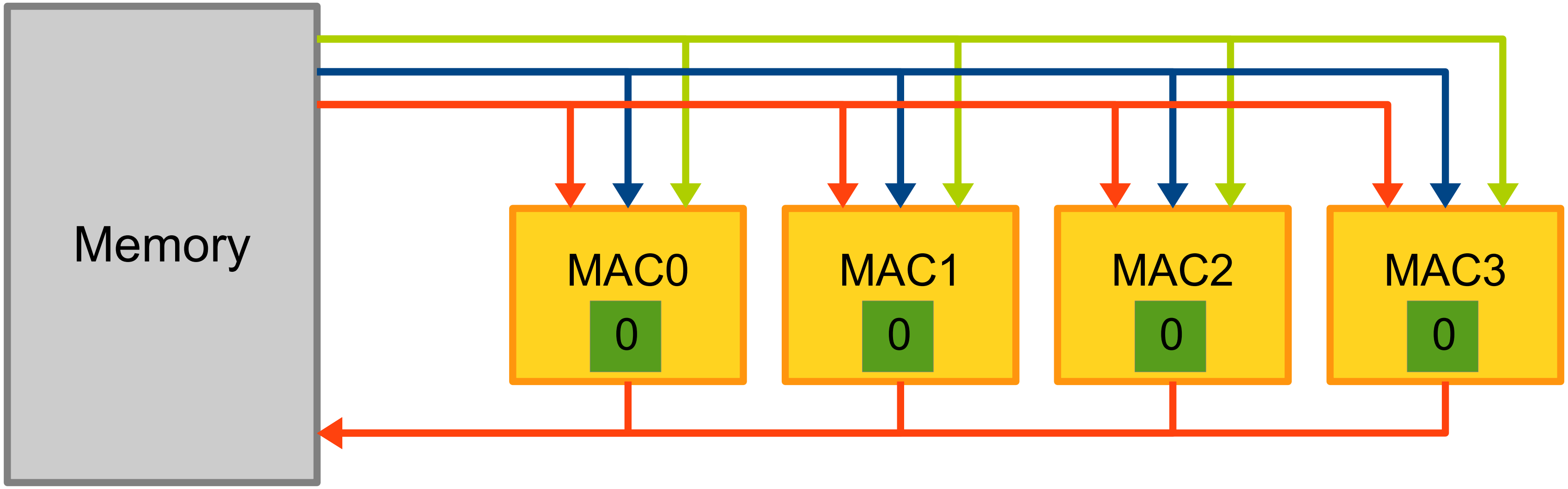
MAC0	0	1	2	3
MAC1	0	1	2	3
MAC2	0	1	2	3
MAC3	0	1	2	3

Operation Ordering 1

One memory access to access weight 0 used by all the four MACs

Spatial Reuse Distance = 0

Spatial Reuse *vs.* Storage Capacity



0	1	2	3	4	5	6

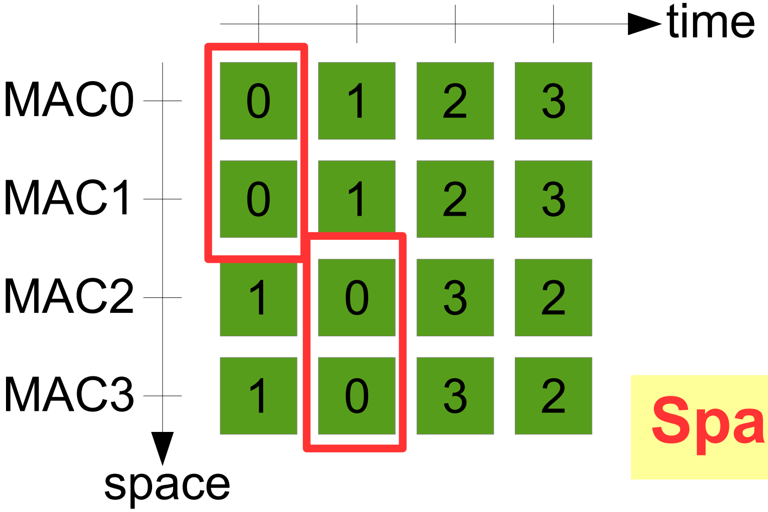
*

0	1	2	3

=

0	1	2	3

Operation Ordering 2

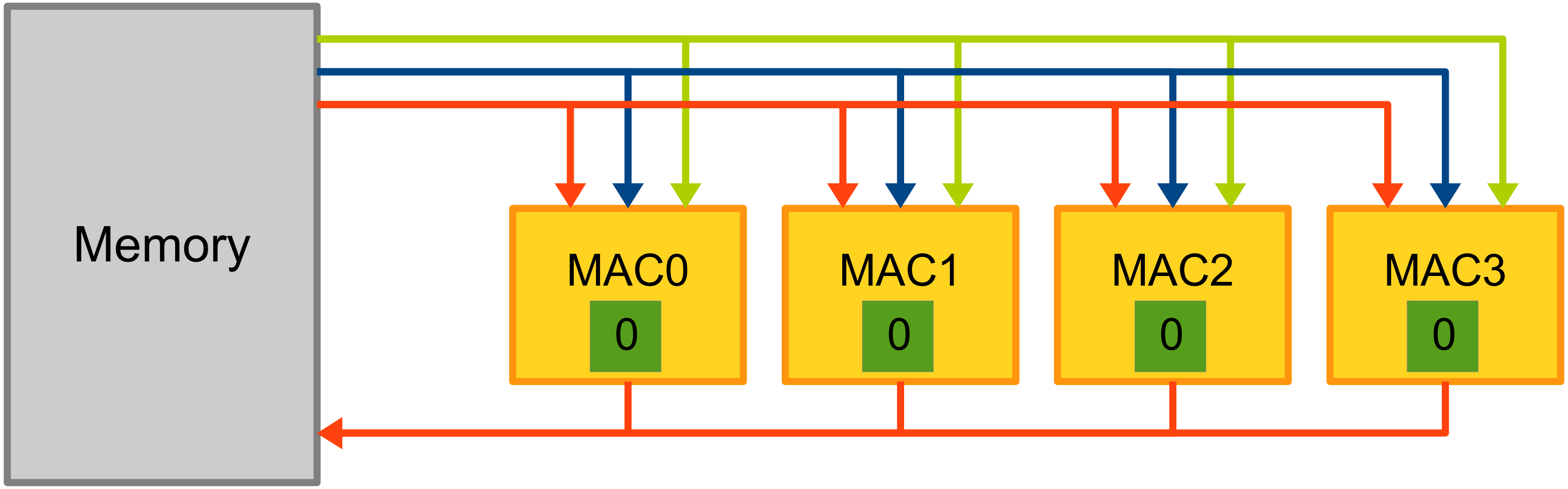


MAC0	0	1	2	3
MAC1	0	1	2	3
MAC2	1	0	3	2
MAC3	1	0	3	2

One memory access to access weight 0 used by all the four MACs: by MAC0 and MAC1 at one cyle and by MAC2 and MAC3 in the subsequent cycle

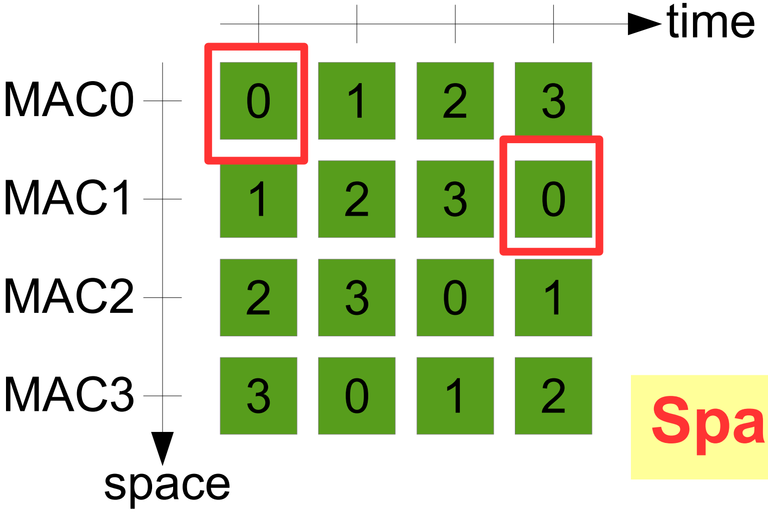
Spatial Reuse Distance = 1

Spatial Reuse *vs.* Storage Capacity



0 1 2 3 4 5 6 * 0 1 2 3 = 0 1 2 3

Operation Ordering 3



MAC0	0	1	2	3
MAC1	1	2	3	0
MAC2	2	3	0	1
MAC3	3	0	1	2

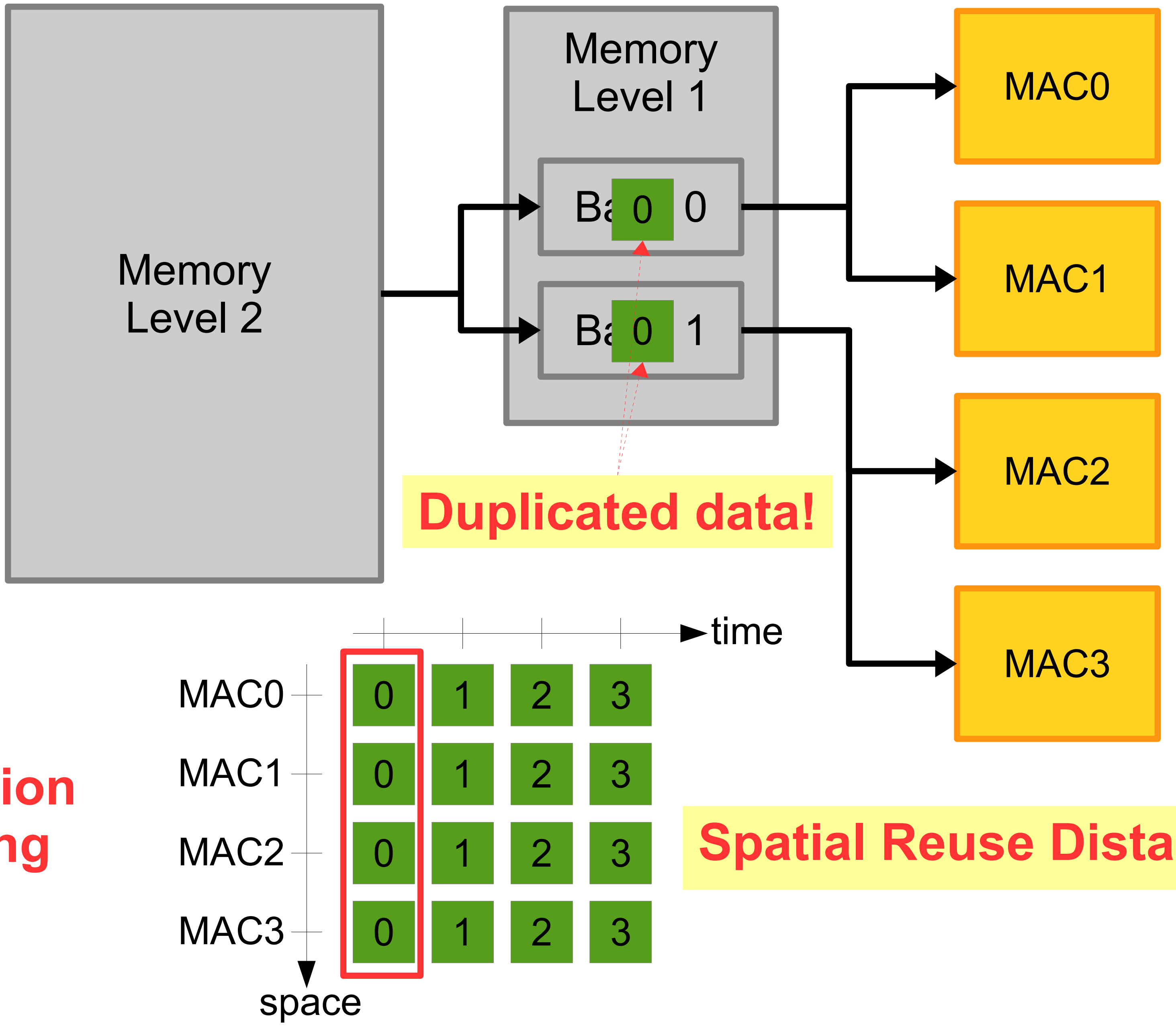
One memory access to access weight 0 that can be reused within one cycle by one MAC at most

Spatial Reuse Distance = 3

Role of the NoC

- NoC plays a key role
  - Spatial reuse involves routing data
  - NoC is responsible for data delivering
  - NoC must support different traffic patterns

Spatial Reuse at Different Levels



Duplicated data!

Operation Ordering

MAC0	0	1	2	3
MAC1	0	1	2	3
MAC2	0	1	2	3
MAC3	0	1	2	3

Spatial Reuse Distance = 0

Reuse Exploitation

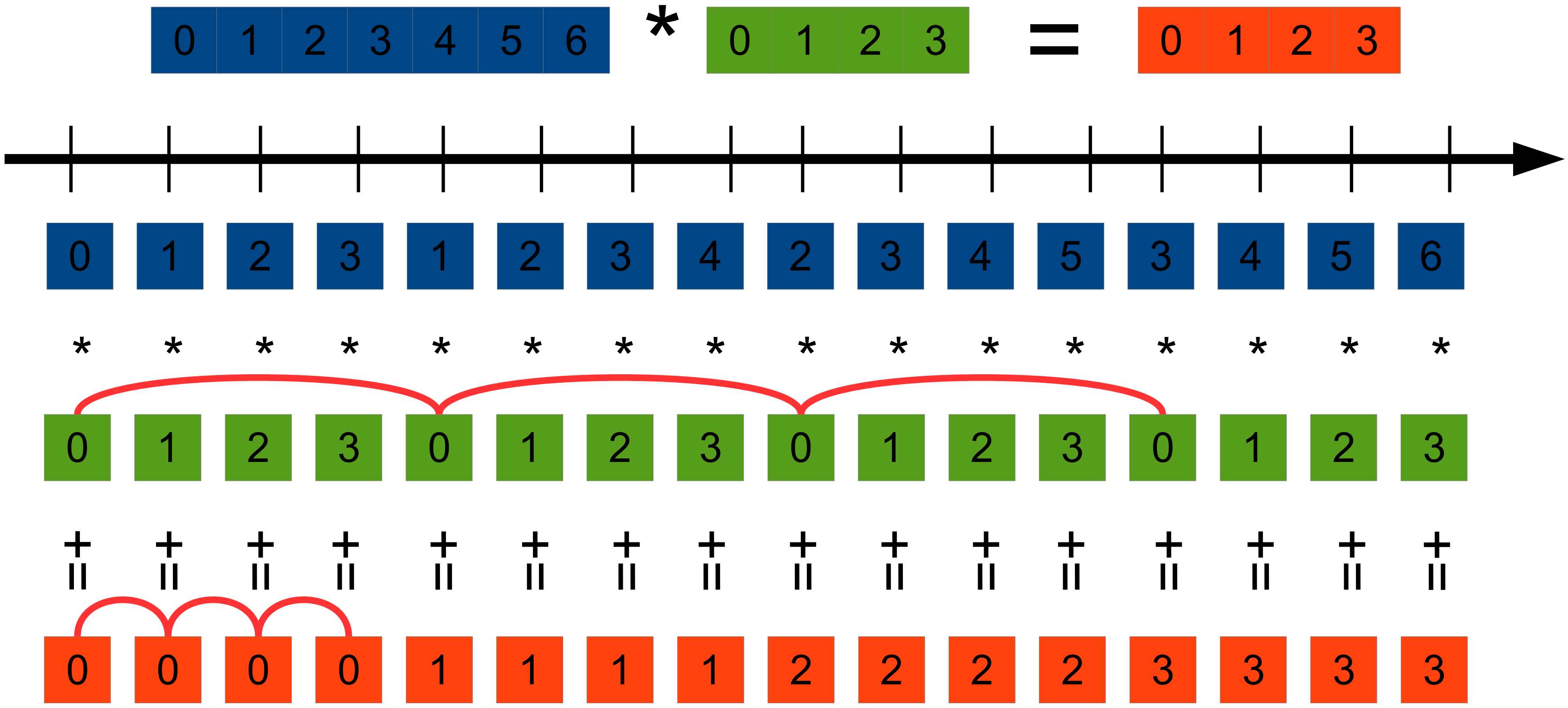
- Operation ordering determines the reuse distance

- Reuse distance impacts the exploitation of temporal/spatial reuse

- Need of reducing the reuse distance

Reuse Exploitation

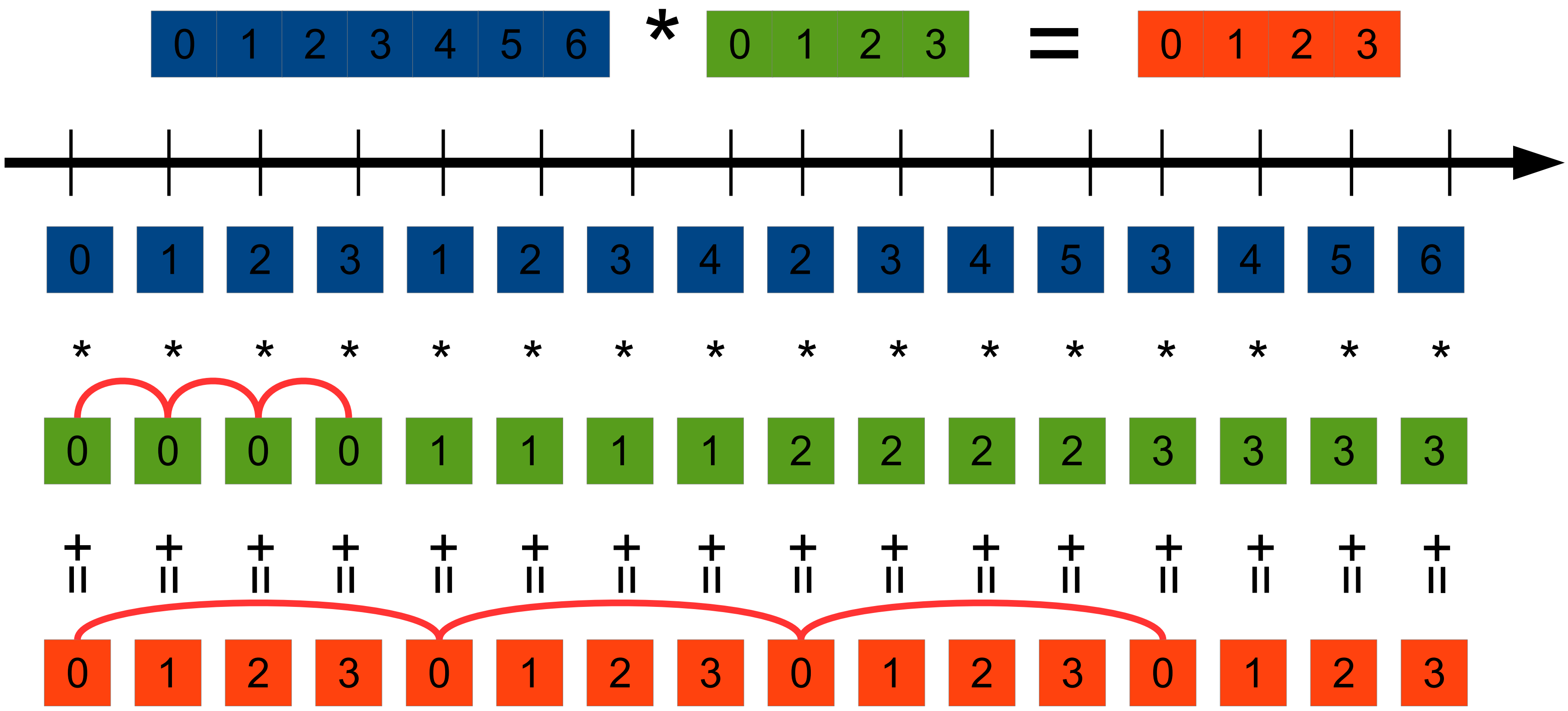
- No way to minimize reuse distance for all data types simultaneously



Reuse Distance = 4 **Reuse Distance = 1**

Reuse Exploitation

- No way to minimize reuse distance for all data types simultaneously



0 1 2 3 4 5 6 * 0 1 2 3 = 0 1 2 3

0 1 2 3 1 2 3 4 2 3 4 5 3 4 5 6

* * * * * * * * * * * * * * * *

0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

+= += += += += += += += += += += += += += += +=

0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

Reuse Distance = 1 **Reuse Distance = 4**

Reuse Exploitation

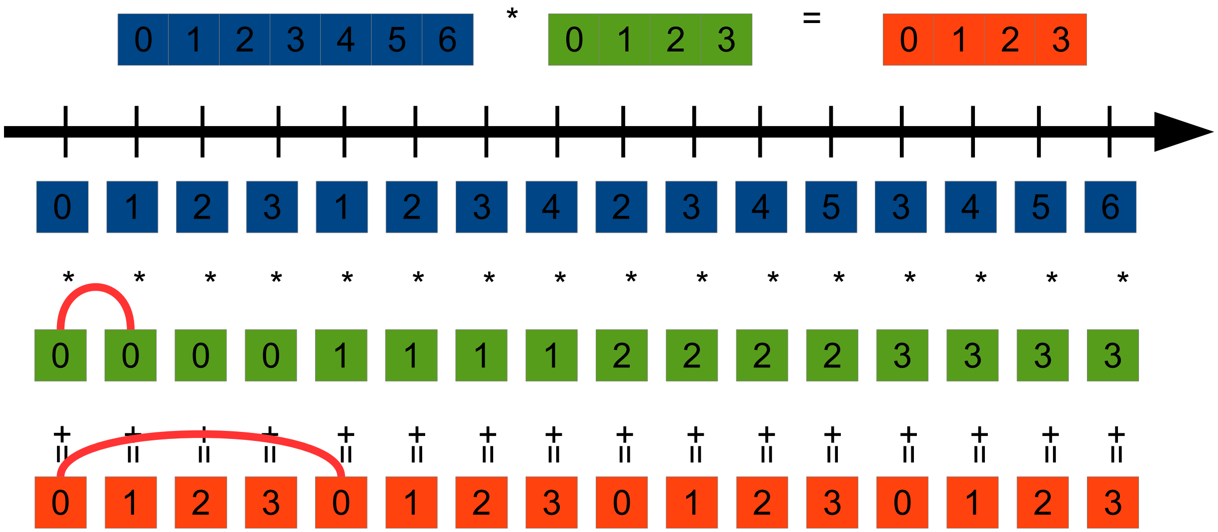
- How do we reduce the reuse distance?
  - Data Tiling
    - **Temporal Tiling**: for temporal reuse
    - **Spatial Tiling**: for spatial reuse
  - Tiling design decision
    - Tile size
    - Dimension being tiled

Temporal Tiling

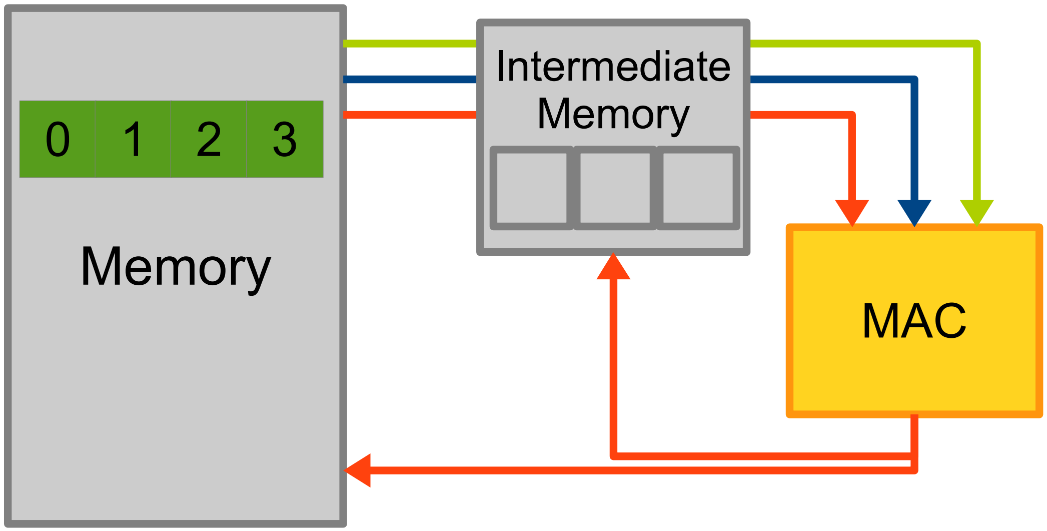
- Ordering operations to increase temporal reuse

Temporal Tiling – Example

Untiled Ordering



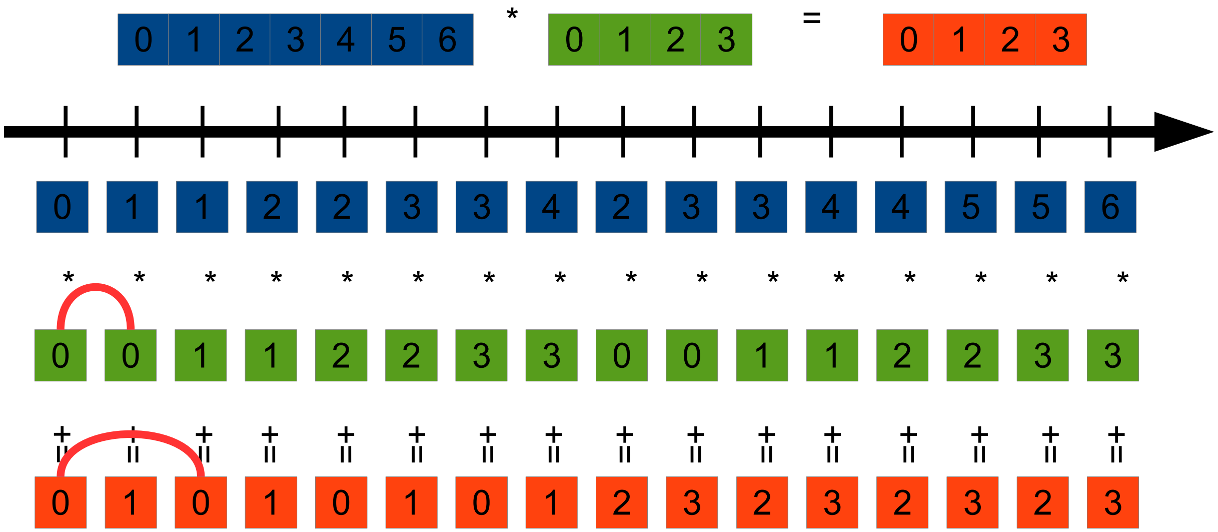
Weight Reuse Distance = 1
Partial sum Reuse Distance = 4



Only temporal reuse of weights
Weight reused 4 times

Temporal Tiling – Example

Tiled Ordering



0 1 2 3 4 5 6 * 0 1 2 3 = 0 1 2 3

0 1 1 2 2 3 3 4 2 3 3 4 4 5 5 6

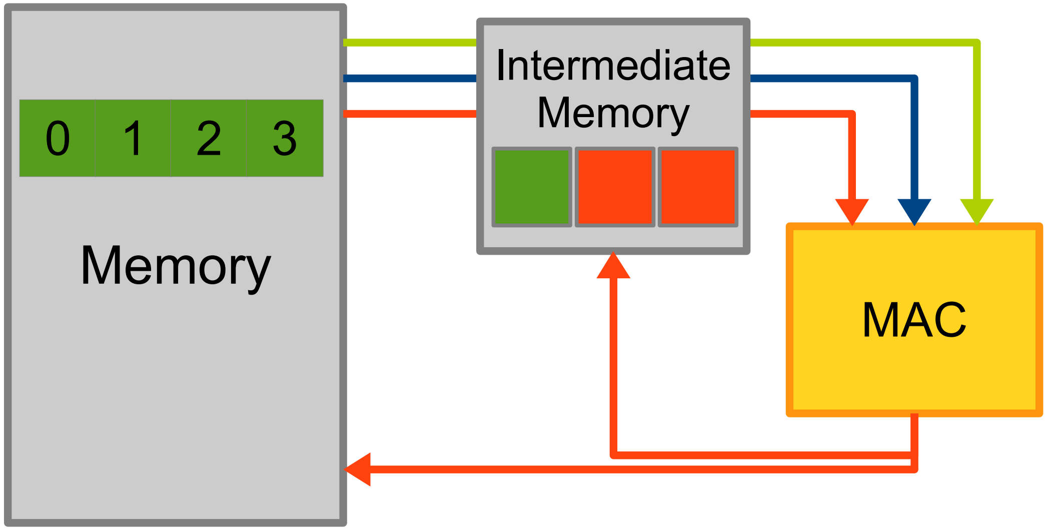
* * * * * * * * * * * * * * * *

0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3

+= += += += += += += += += += += += += += += +=

0 1 0 1 0 1 0 1 2 3 2 3 2 3 2 3

Weight Reuse Distance = 1
Partial sum Reuse Distance = 2



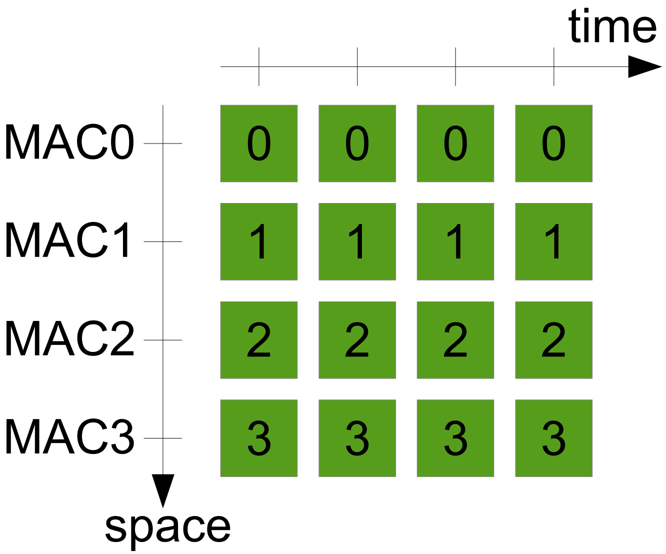
Temporal reuse of weights and partial sums

One weight and a tile of two partial sums are reused twice

Spatial Tiling

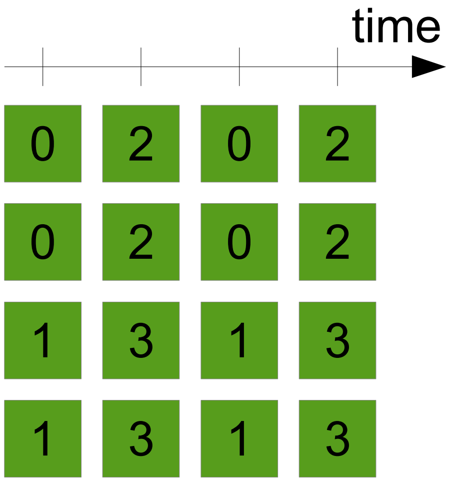
- Ordering operations to increase spatial reuse
  - Reusing the same data by as many PEs as possible
  - Reducing the reuse distance so that one multicast can serve as many PEs as possible
    - Given a certan amount of local storage

Spatial Tiling – Example



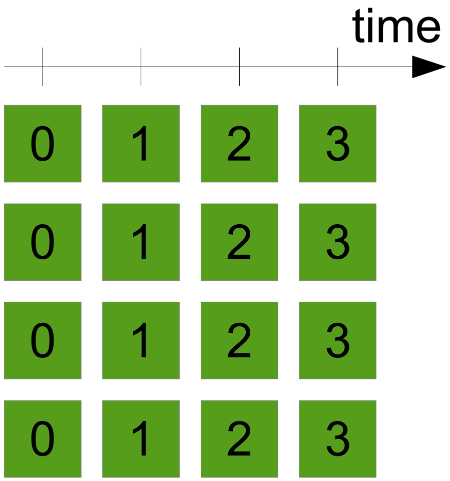
MAC0	0	0	0	0
MAC1	1	1	1	1
MAC2	2	2	2	2
MAC3	3	3	3	3

No spatial reuse



0	2	0	2
0	2	0	2
1	3	1	3
1	3	1	3

Medium degree of spatial reuse



0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

High degree of spatial reuse

Dataflow

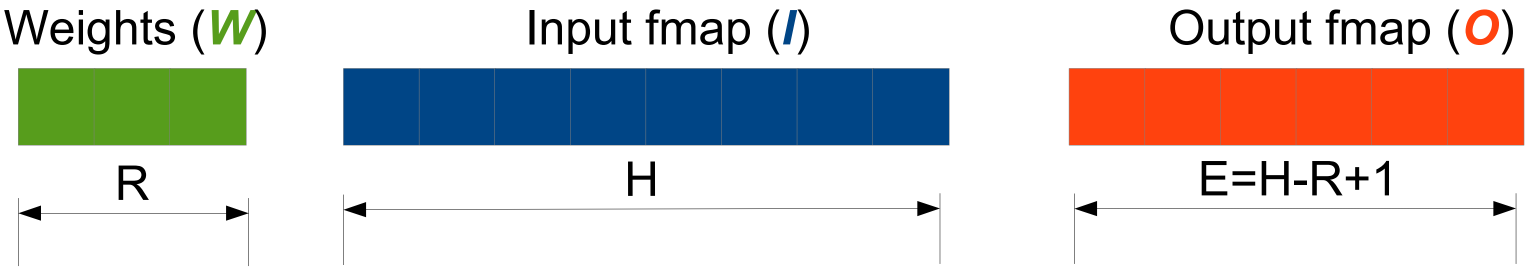
$$O_{nmpq} = \left(\sum_{crs} I_{nc(p+r)(q+s)} F_{mcrs} \right) + b_m$$

- No ordering no notion of parallelism
- Dataflow
  - Specifies an **ordering** and calculations run in **parallel**

Describing a Dataflow

- Loop nests can be used to describe a dataflow
- Dataflow taxonomy
  - Output Stationary
  - Weight Stationary
  - Input Stationary

Output Stationary Dataflow



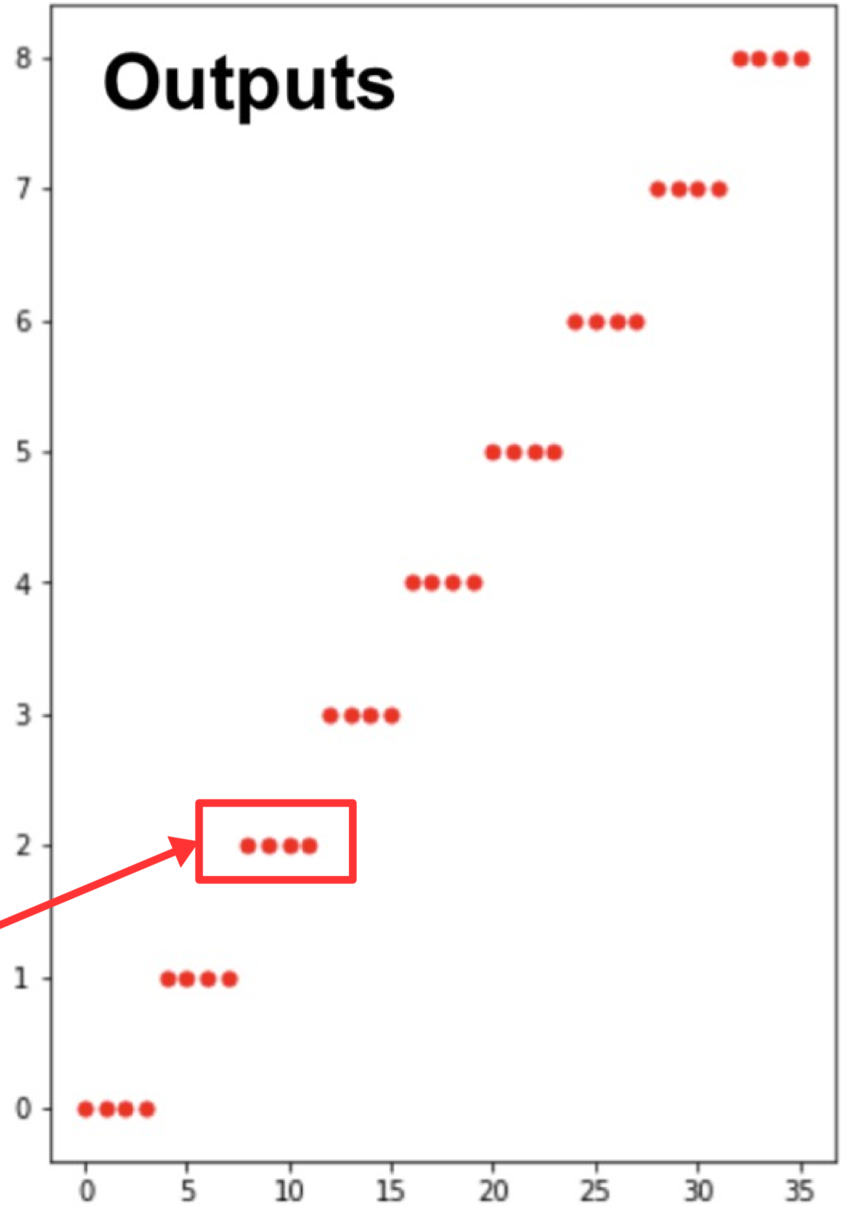
```
int W[R];
int I[H];
int O[E];

for (e=0; e<E; e++)
   for (r=0; r<R; r++)
      O[e] += W[r] * I[e+r];
```

Output Stationary – Outputs

```
int W[R];
int I[H];
int O[E];

for (e=0; e<E; e++)
 for (r=0; r<R; r++)
  O[e] += W[r] * I[e+r];
```

Example:
H=12, R=4, E=9

Same **output** reused repeatedly (R times)

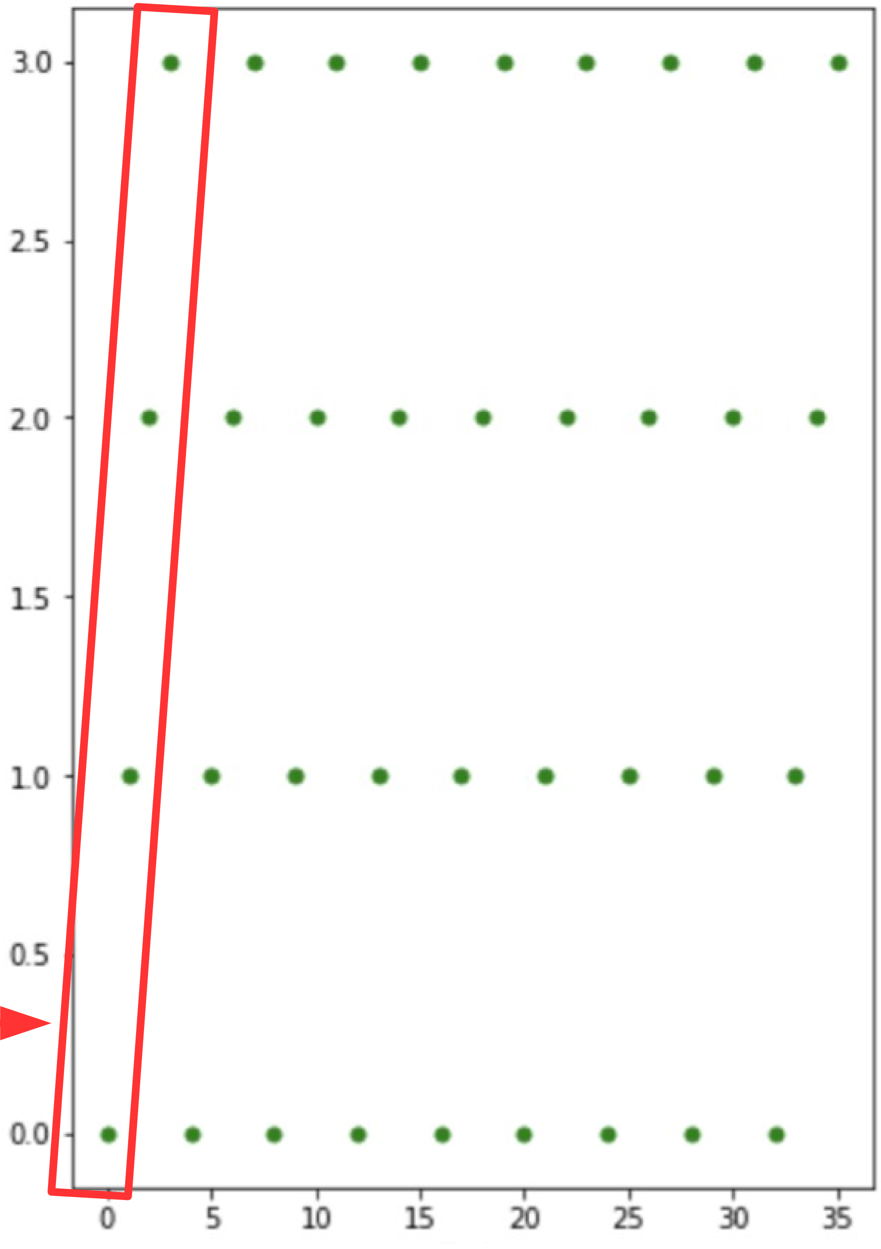


Output Stationary – Weights

```
int W[R];
int I[H];
int O[E];

for (e=0; e<E; e++)
 for (r=0; r<R; r++)
  O[e] += W[r] * I[e+r];
```

Example:
H=12, R=4, E=9

All **weights** reused repeatedly

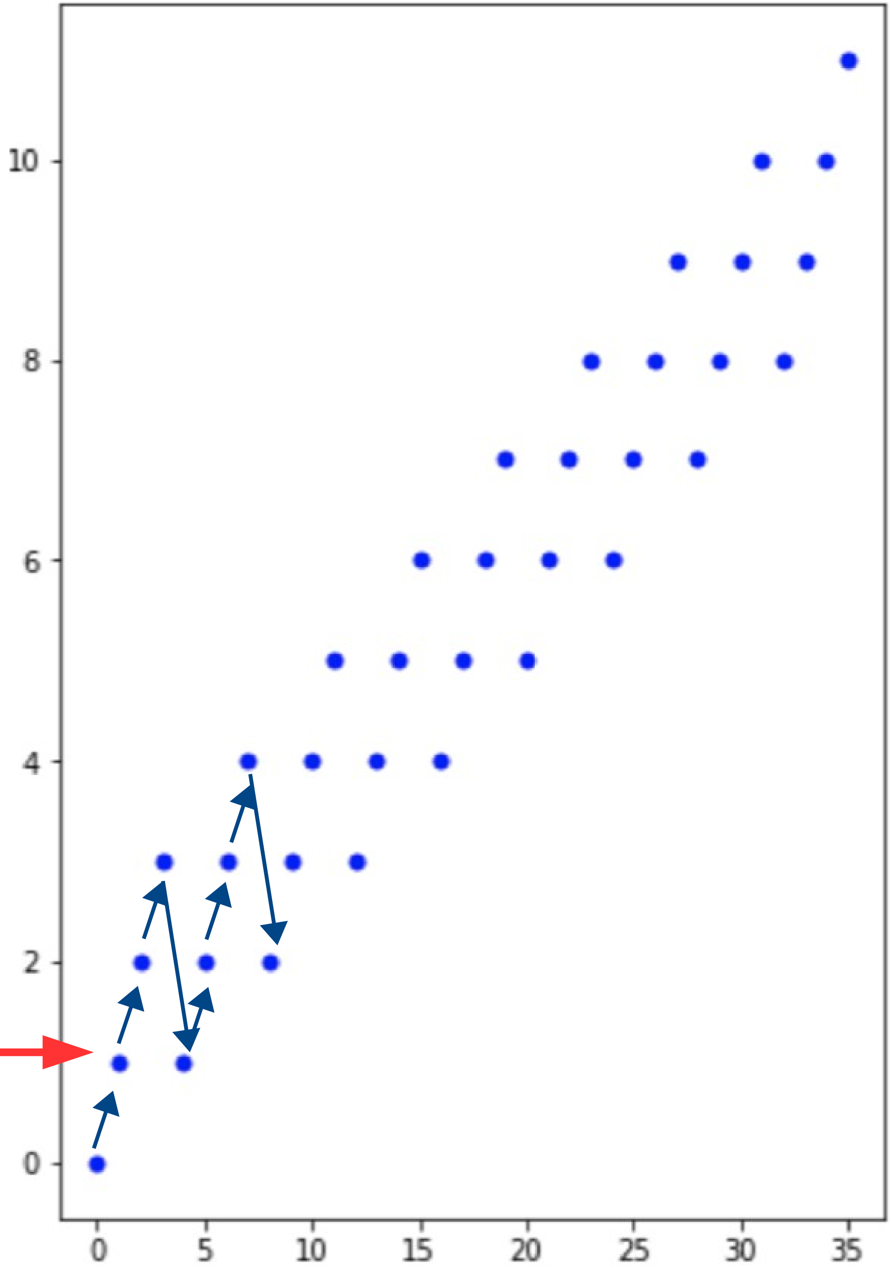


Output Stationary – Inputs

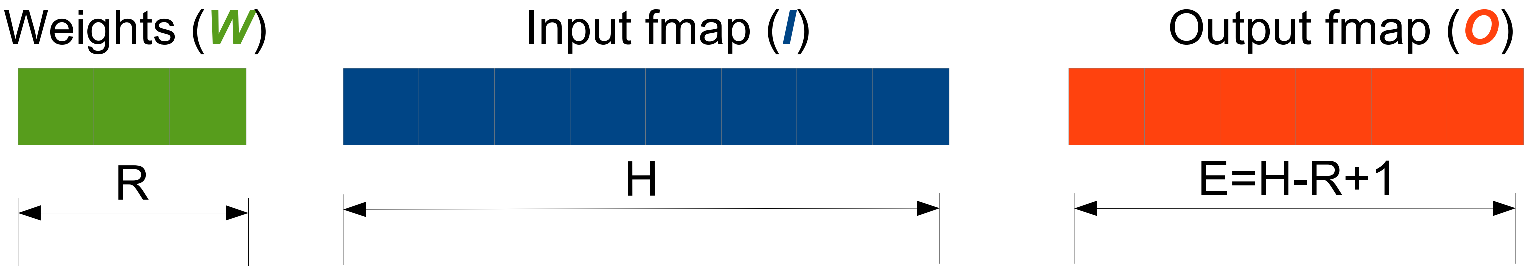
```
int W[R];
int I[H];
int O[E];

for (e=0; e<E; e++)
 for (r=0; r<R; r++)
  O[e] += W[r] * I[e+r];
```

Example:
H=12, R=4, E=9

Sliding window of **inputs**
(size R)



Weight Stationary Dataflow

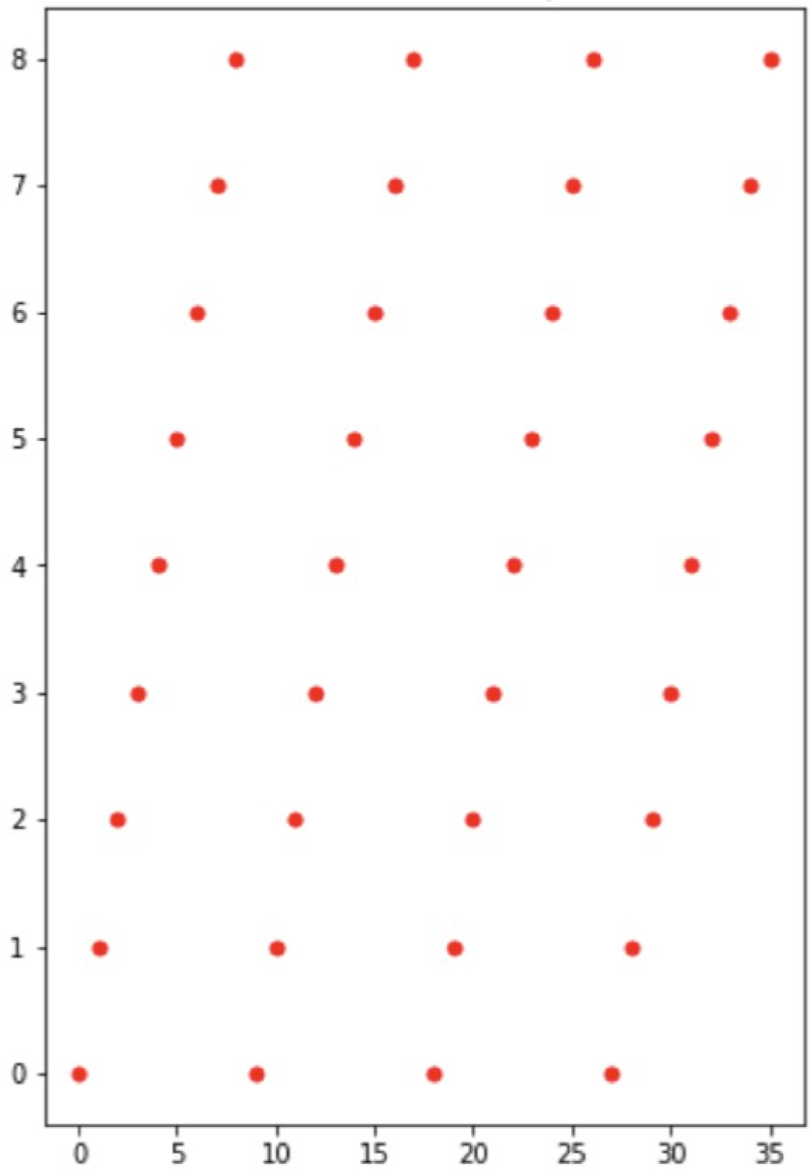


```
int W[R];
int I[H];
int O[E];

for (r=0; r<R; r++)
  for (e=0; e<E; e++)
    O[e] += W[r] * I[e+r];
```

Weight Stationary – Outputs

```
int W[R];
int I[H];
int O[E];

for (r=0; r<R; r++)
 for (e=0; e<E; e++)
  O[e] += W[r] * I[e+r];
```

Example:
H=12, R=4, E=9



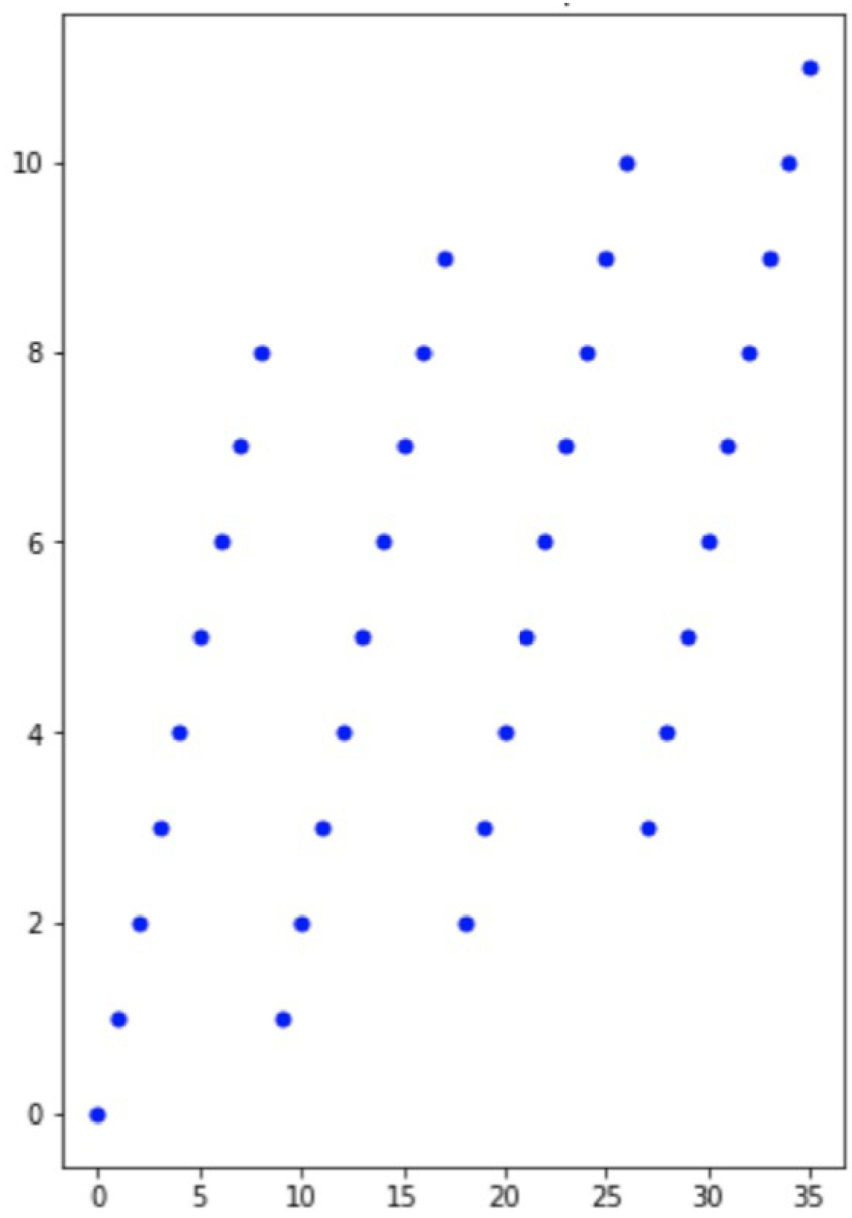
Fixed window of **outputs**
(size E)

Weight Stationary – Outputs

```
int W[R];
int I[H];
int O[E];

for (r=0; r<R; r++)
 for (e=0; e<E; e++)
  O[e] += W[r] * I[e+r];
```

Example:
H=12, R=4, E=9

Large sliding window of **inputs** (size E)

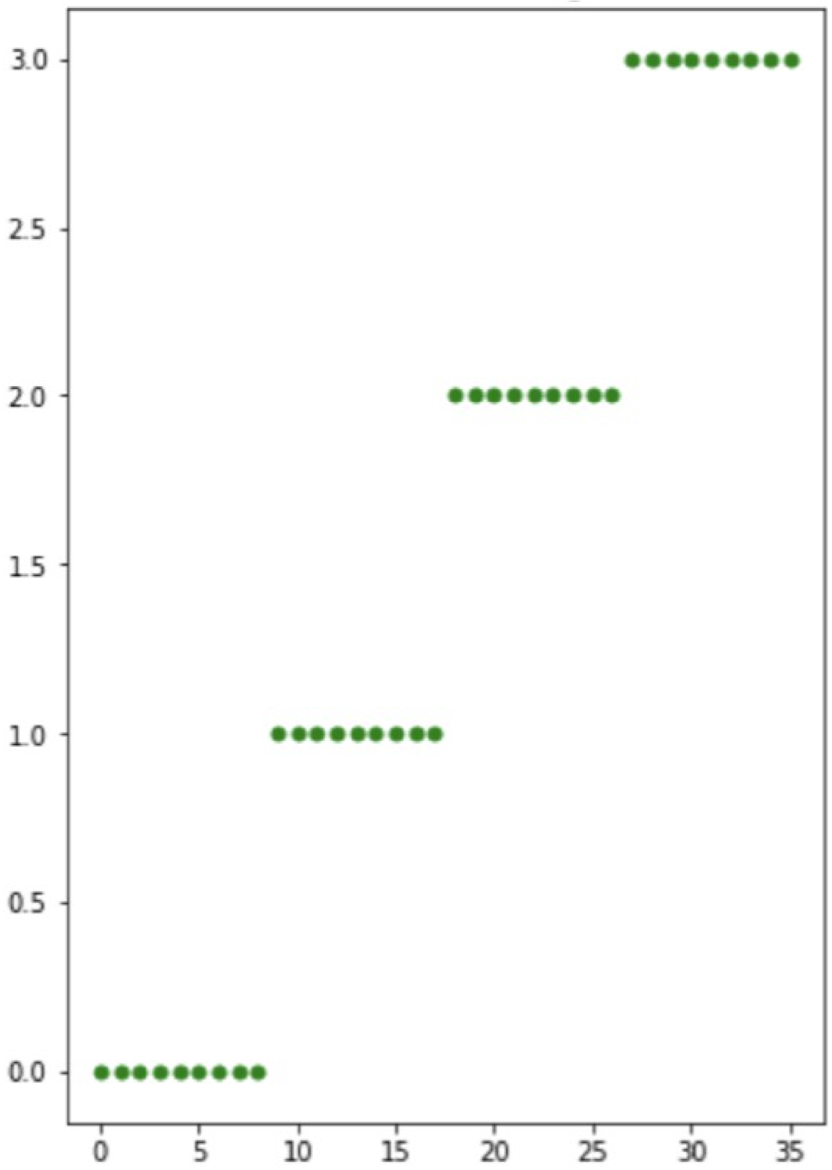


Weight Stationary – Outputs

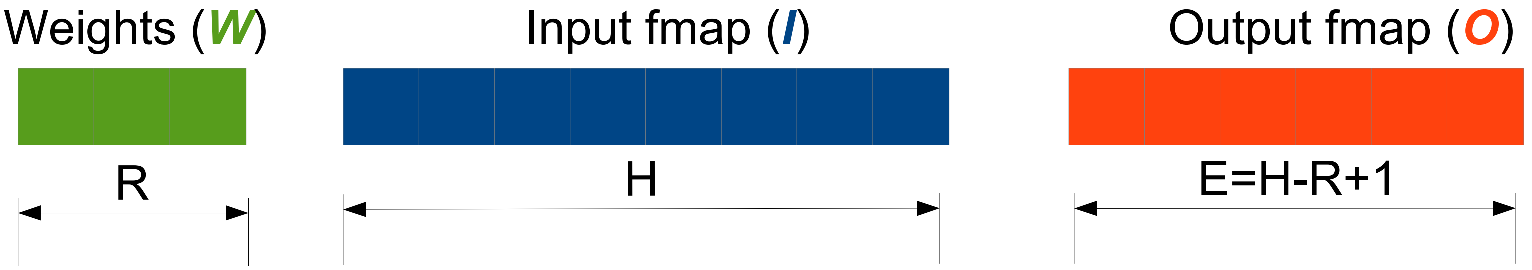
```
int W[R];
int I[H];
int O[E];

for (r=0; r<R; r++)
 for (e=0; e<E; e++)
  O[e] += W[r] * I[e+r];
```

Example:
H=12, R=4, E=9

Same **weight** reused repeatedly (E times)



Input Stationary Dataflow



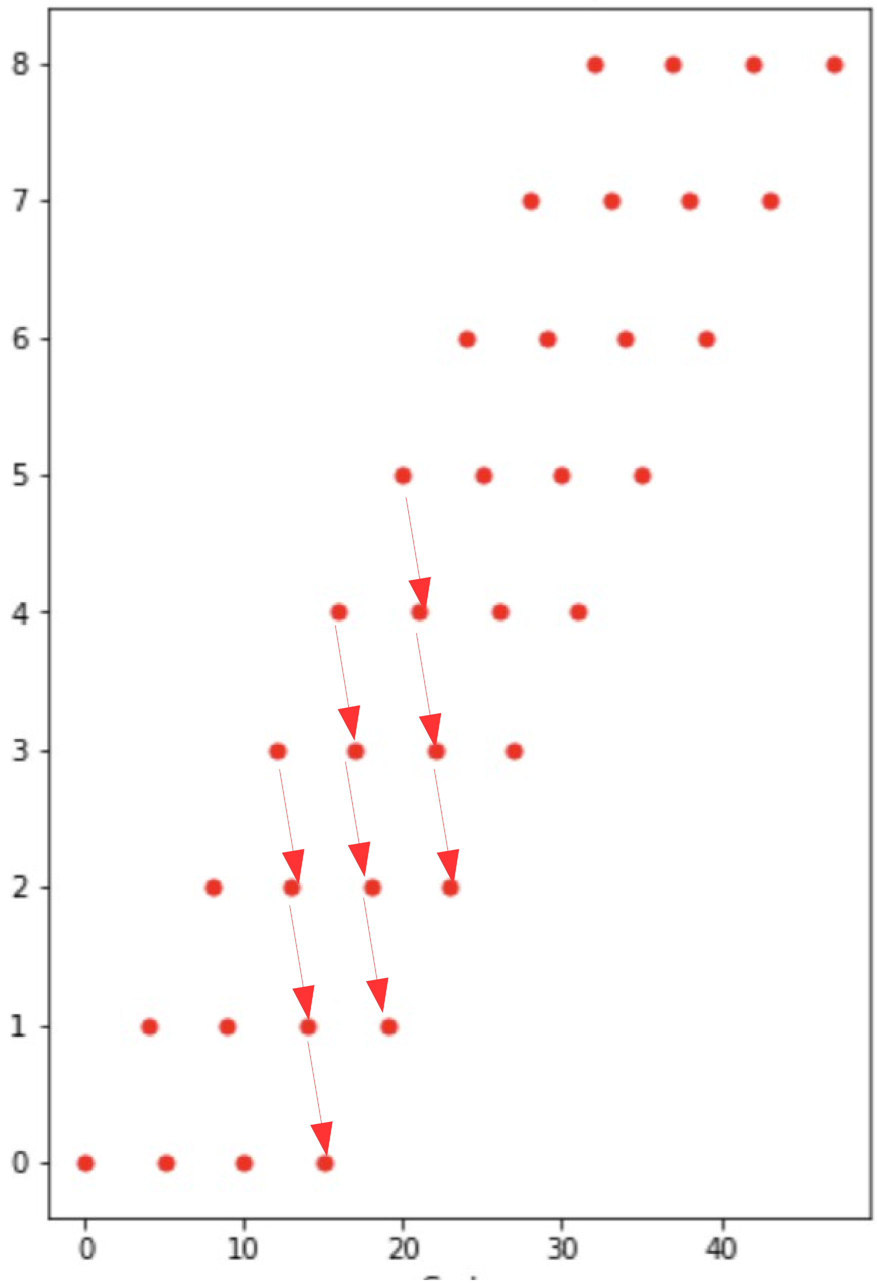
```
int W[R];
int I[H];
int O[E];

for (h=0; h<H; h++)
  for (r=0; r<R; r++)
    O[h-r] += W[r] * I[h];
```

Input Stationary – Outputs

```
int W[R];
int I[H];
int O[E];

for (h=0; h<H; h++)
 for (r=0; r<R; r++)
  O[h-r] += W[r] * I[h];
```

Example:
H=12, R=4, E=9

Sliding window of **outputs**
(size R)

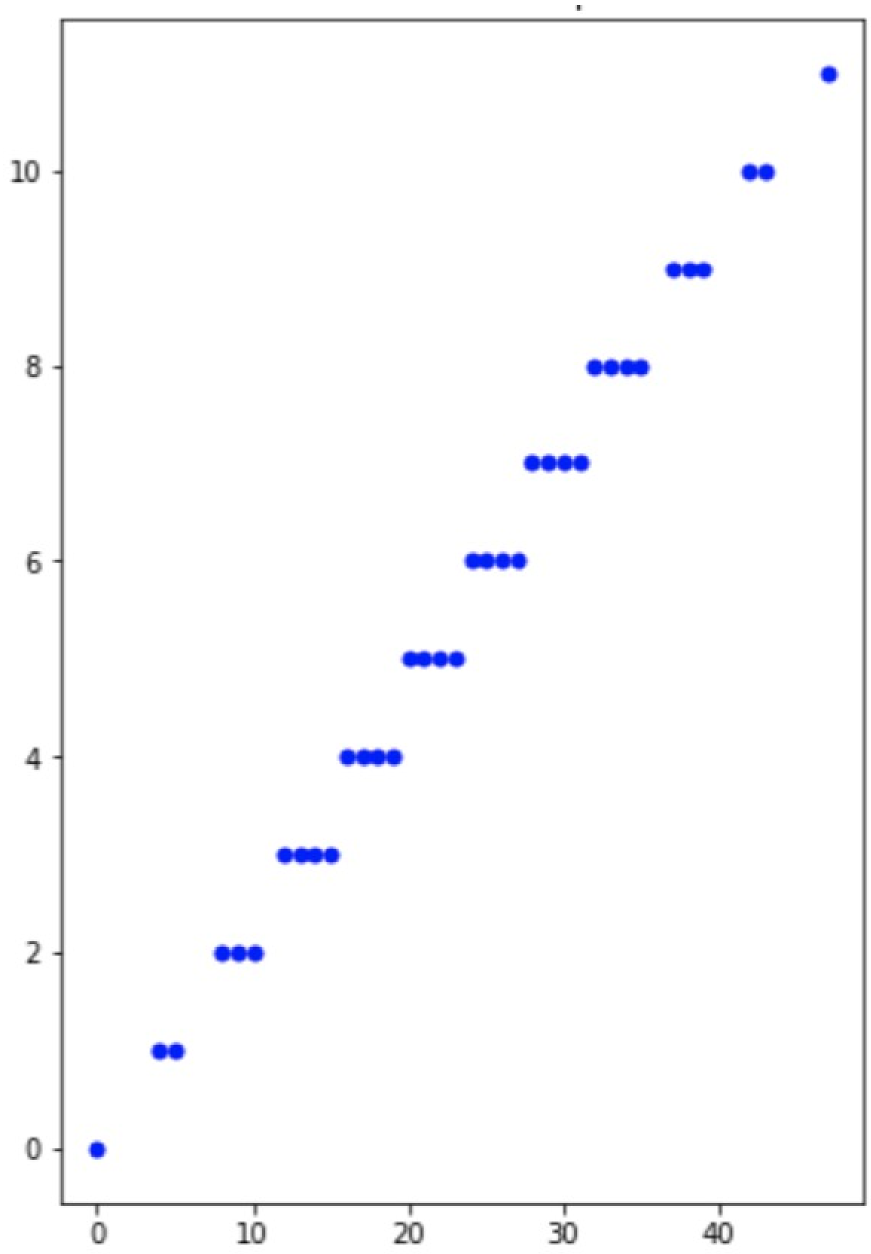


Input Stationary – Inputs

```
int W[R];
int I[H];
int O[E];

for (h=0; h<H; h++)
 for (r=0; r<R; r++)
  O[h-r] += W[r] * I[h];
```

Example:
H=12, R=4, E=9

Inputs reused repeatedly (R times)

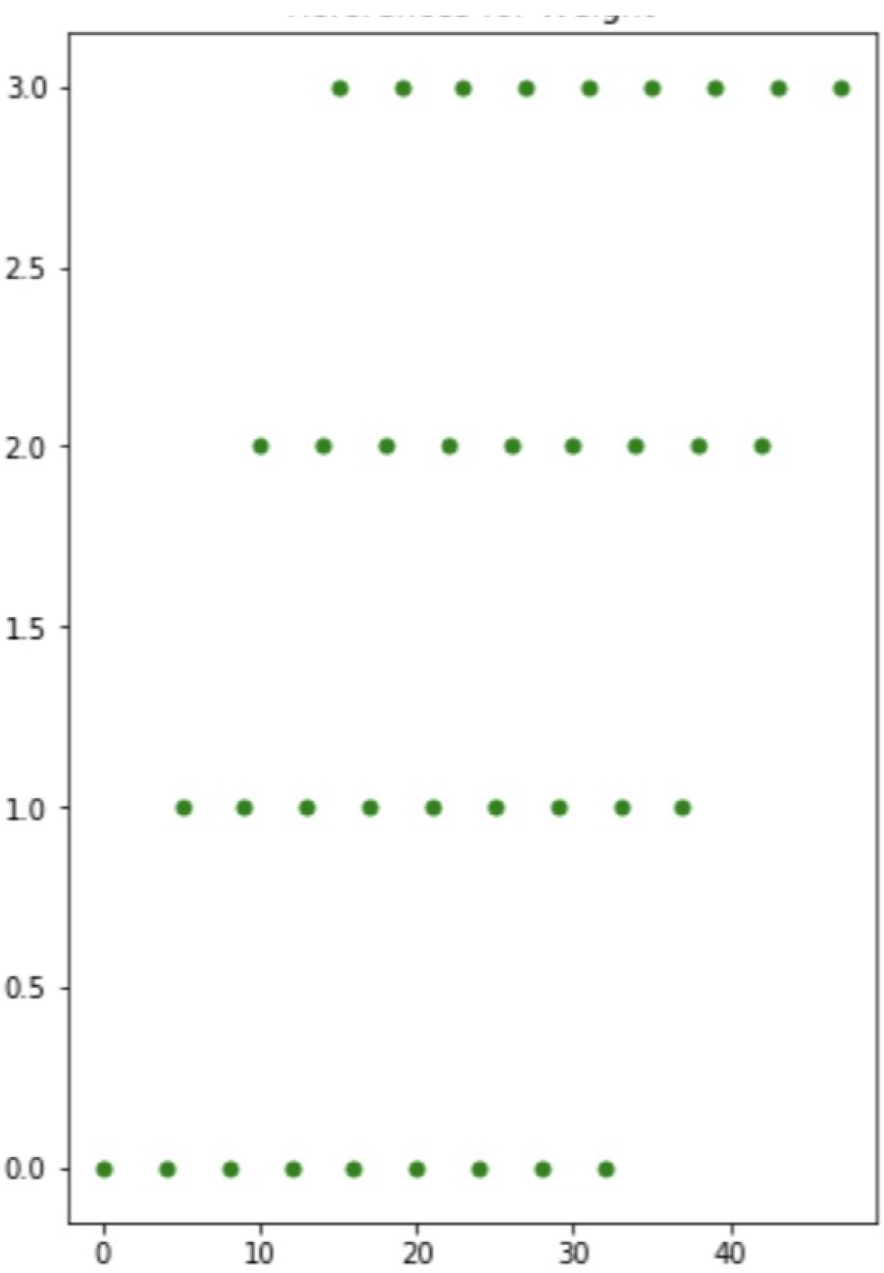


Input Stationary – Weights

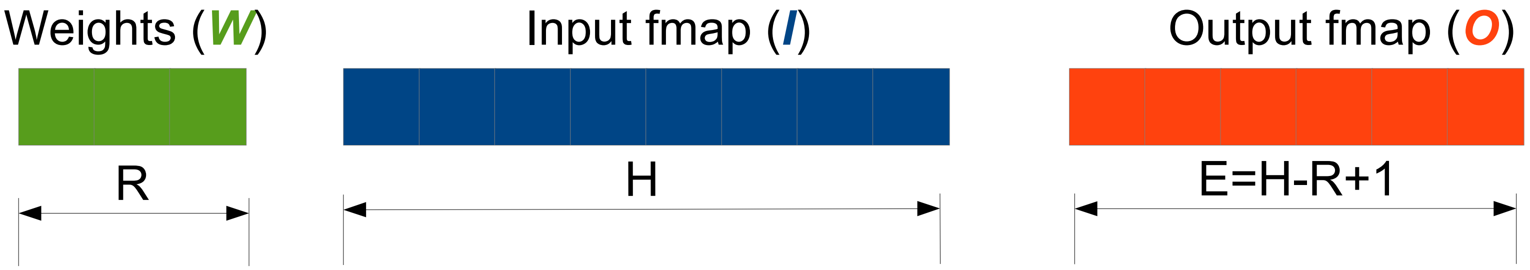
```
int W[R];
int I[H];
int O[E];

for (h=0; h<H; h++)
 for (r=0; r<R; r++)
  O[h-r] += W[r] * I[h];
```

Example:
H=12, R=4, E=9

Weights reused in large window (size R)



Weight Stationary DF with Tiling

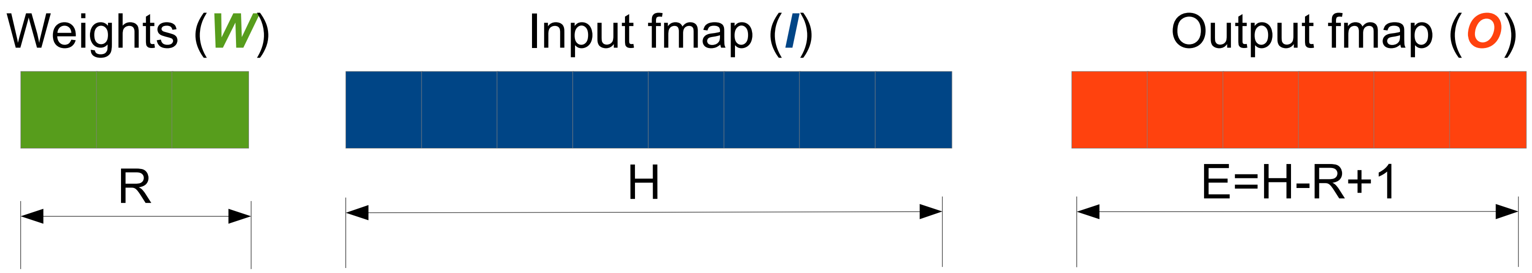


E1 tiles of E0 elements

```
int W[R];
int I[H];
int O[E];

for (e1=0; e1<E1; e1++)
 for (r=0; r<R; r++)
  for (e0=0; e0<E0; e0++) {
    e = e1*E0+e0;
    O[e] += W[r] * I[e+r];
  }
```

Weight Stationary DF with Parallel Processing and Tiling

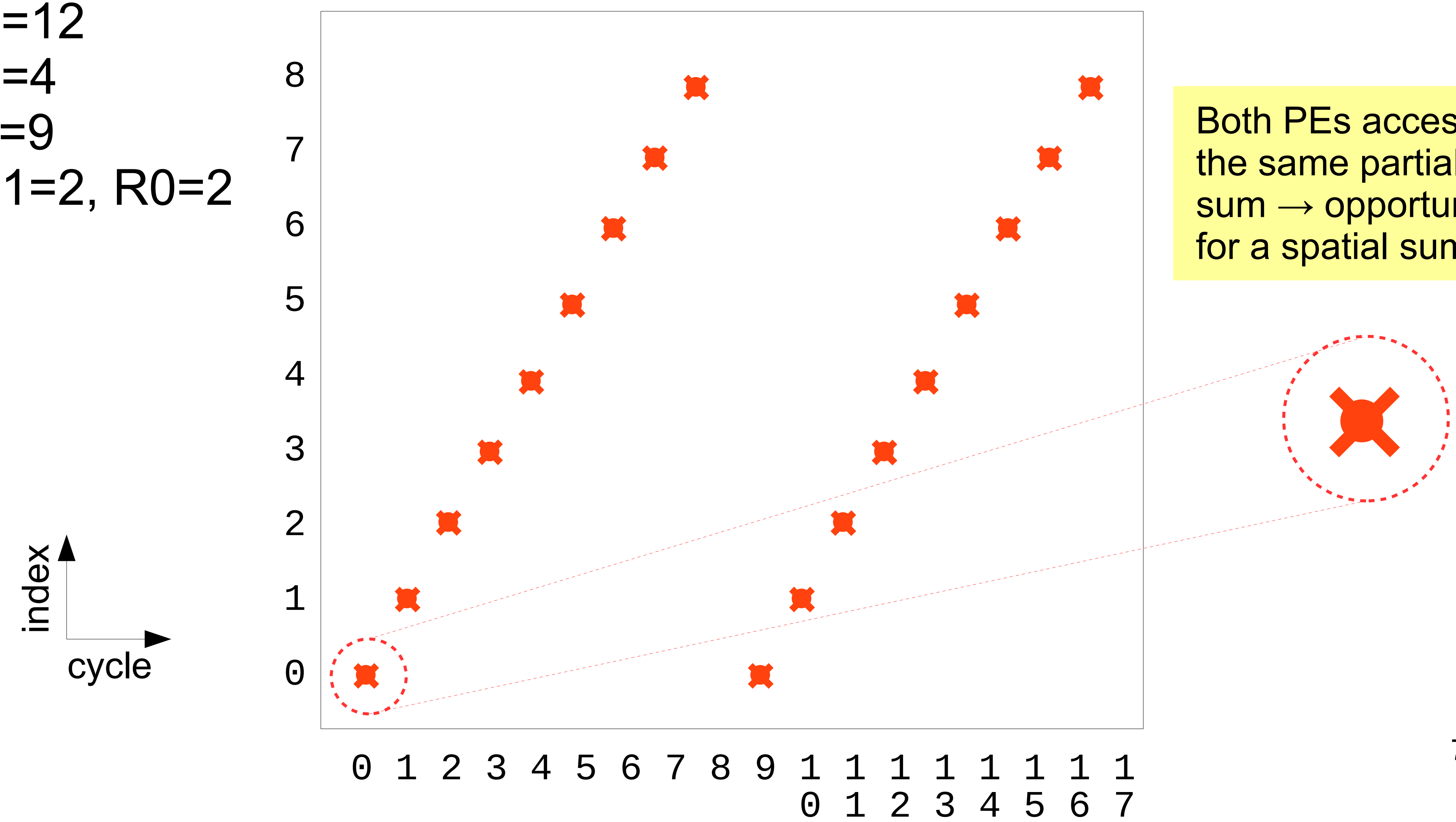
R1 tiles of R0 elements



```
int W[R];
int I[H];
int O[E];

for (r1=0; r1<R1; r1++)
 for (e=0; e<E; e++)
  parallel-for (r0=0; r0<R0; r0++) {
    r = r1*R0+r0;
    O[e] += W[r] * I[e+r];
  }
```

Weight Stationary DF with Parallel Processing and Tiling – Outputs

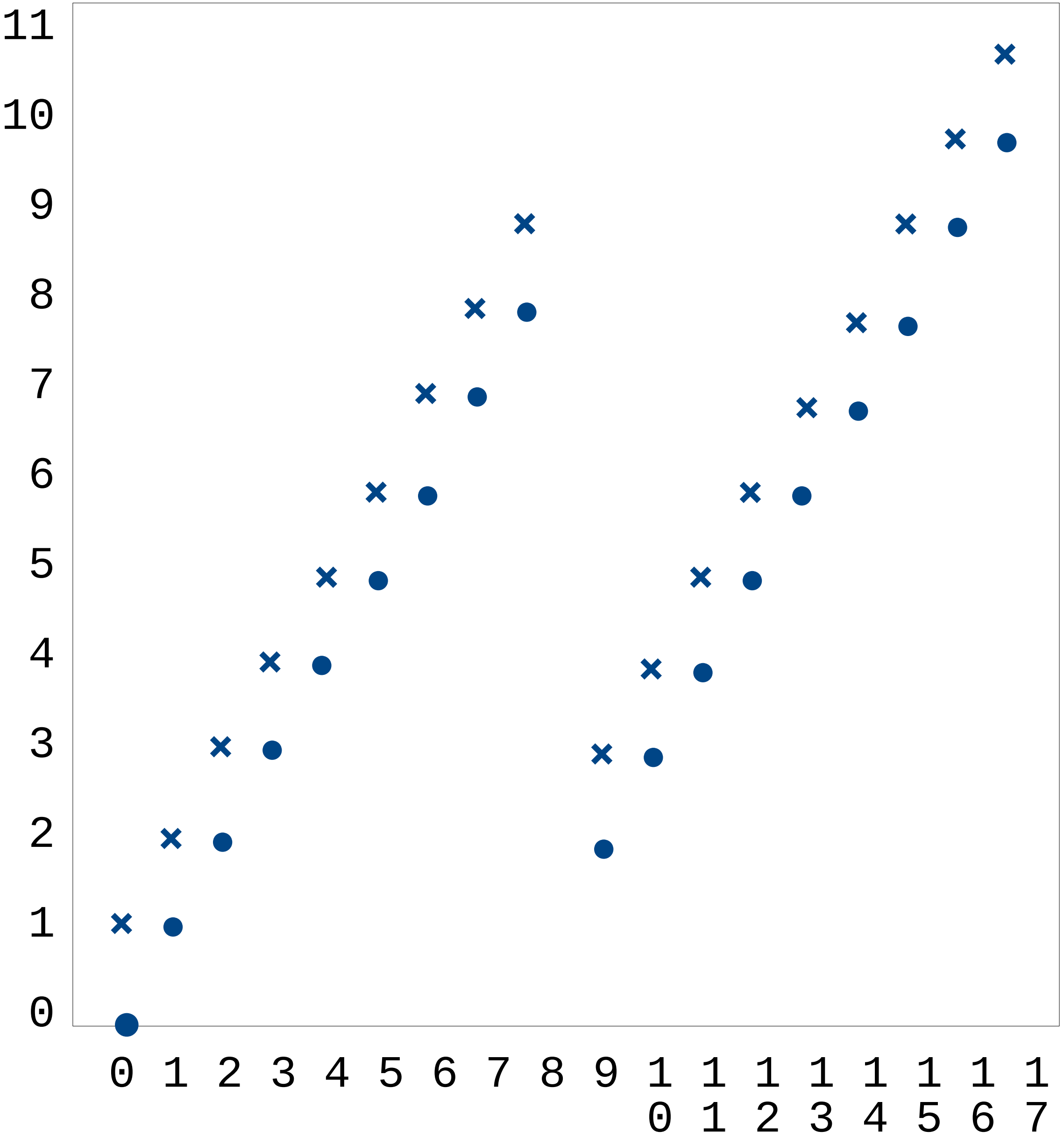
Example:
H=12
R=4
E=9
R1=2, R0=2



Both PEs access the same partial sum → opportunity for a spatial sum

Weight Stationary DF with Parallel Processing and Tiling – Inputs

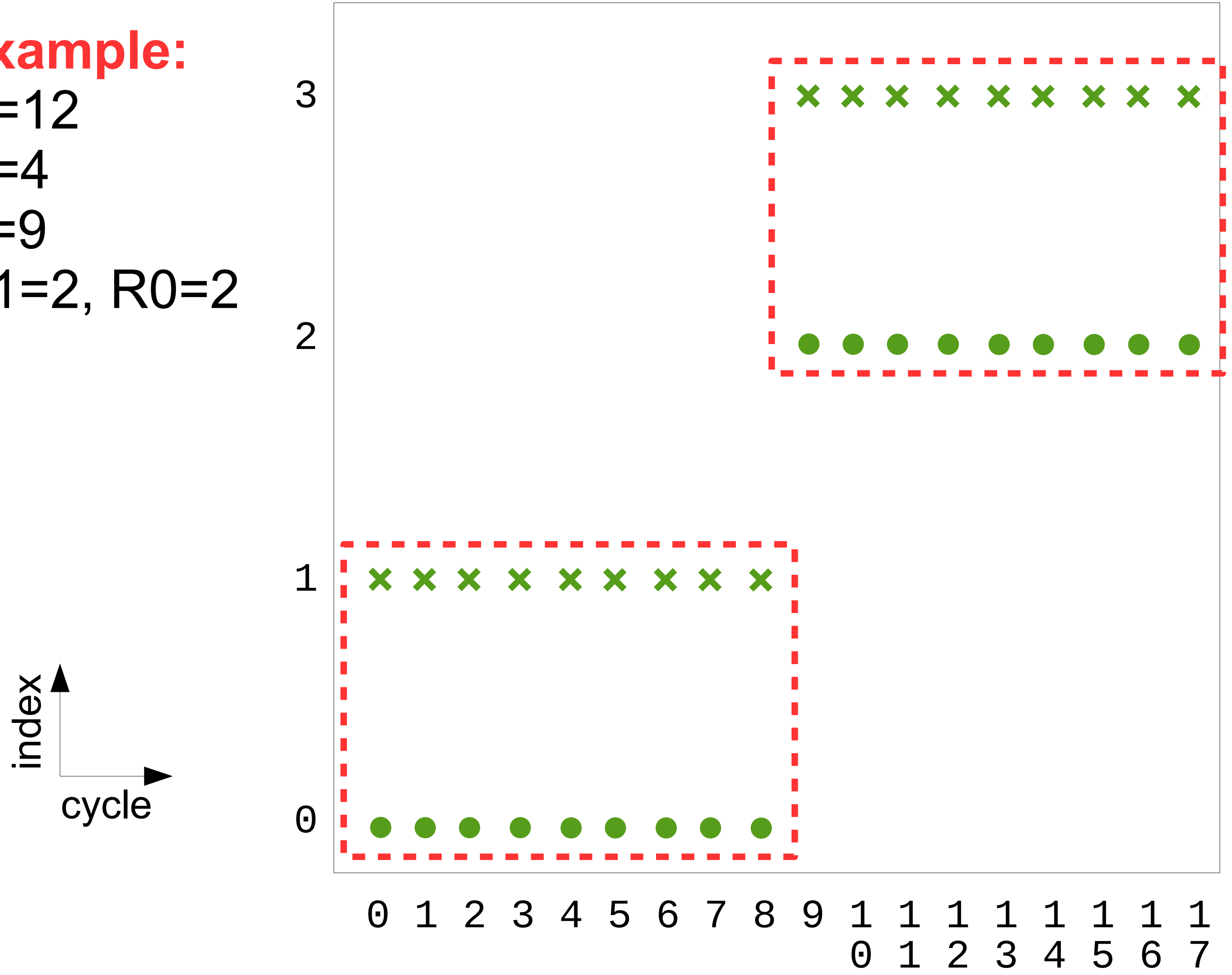
Example:
H=12
R=4
E=9
R1=2, R0=2



Two PEs use the same activation in successive cycles → opportunity for inter-PE communication or opportunity for broadcast the activation and use it in the next cycle (need for local storage)

Weight Stationary DF with Parallel Processing and Tiling – Weights

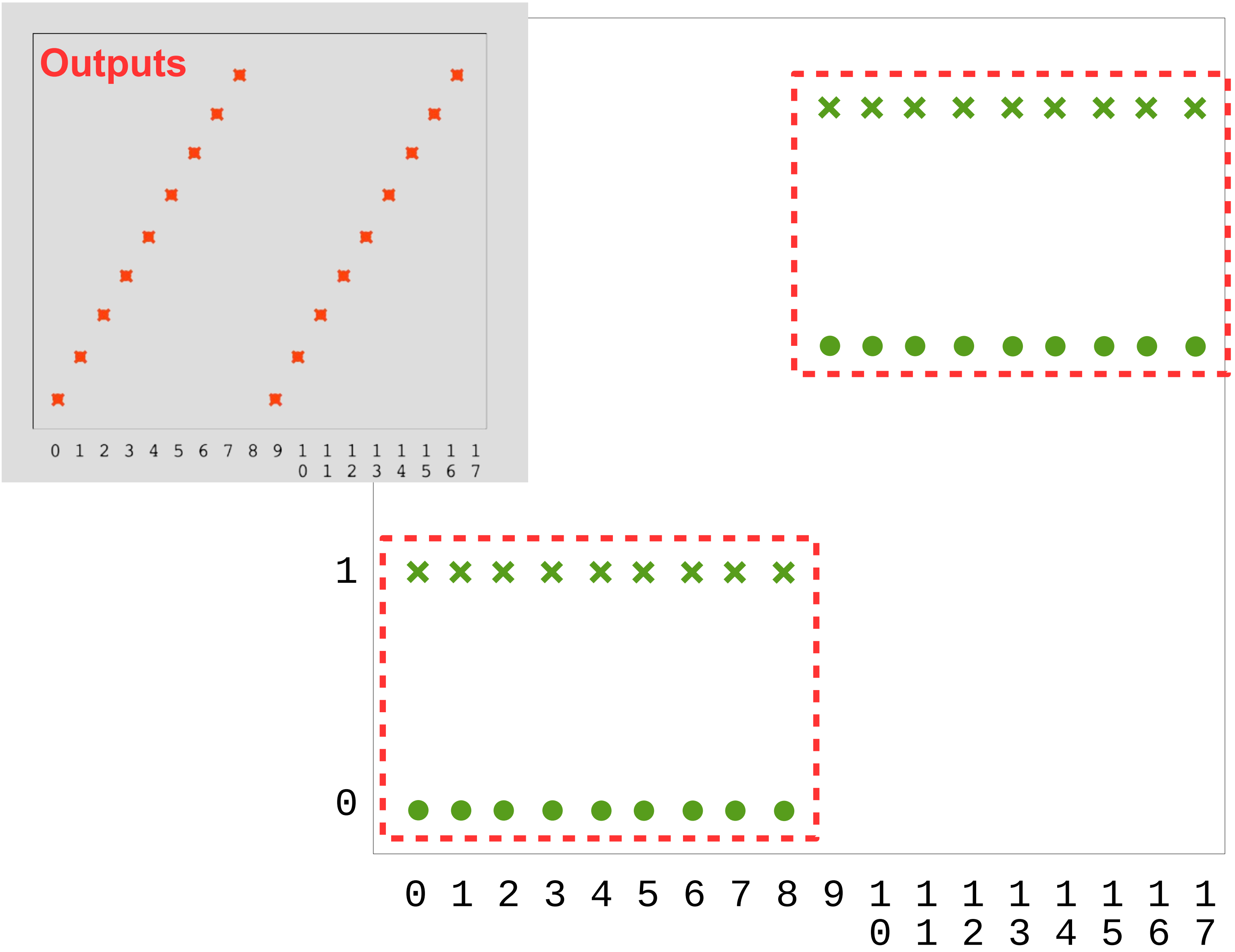
Example:
H=12
R=4
E=9
R1=2, R0=2



Two tiles of weights

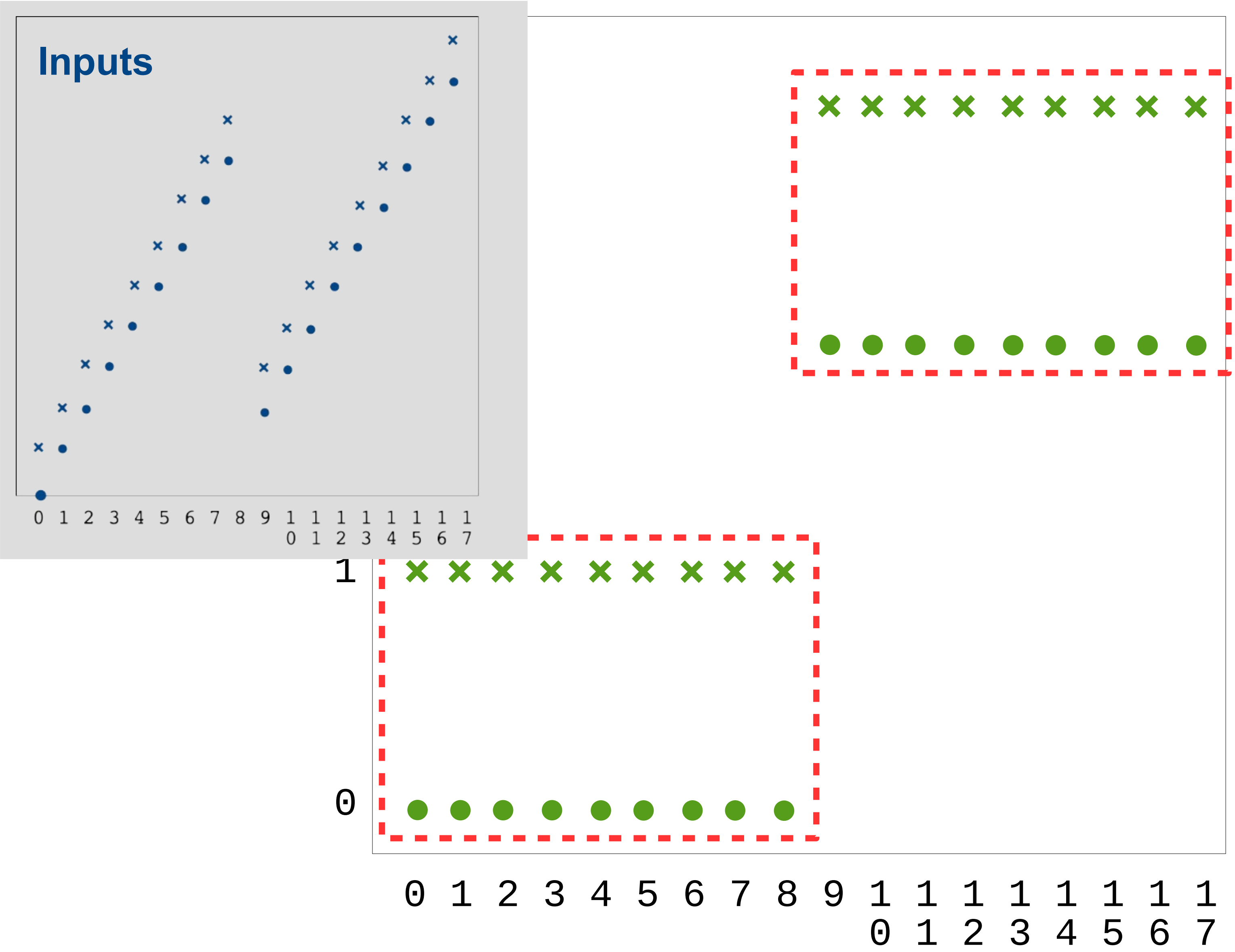
- weights 0 and 1 used in the first 9 cycles
- weights 2 and 3 used in the subsequent 9 cycles

Weight Stationary DF with Parallel Processing and Tiling – Weights



Both tiles of weights contribute to the same outputs → **outputs have a long tile-related reuse distance of 9 cycles**

Weight Stationary DF with Parallel Processing and Tiling – Weights



Most inputs are used by both tiles → **inputs have a long tile-related reuse distance of 8 cycles**

Dataflow & Loop Nest – Summary

- Loop nest allows to describe a dataflow
- Order of the loops
  - Defines the prioritization among the data types
- Number of the loops
  - Defines the tiling
- Temporal (for) and spatial (parallel-for)
  - Define temporal and spatial caracteristics

Dataflow & Loop Nest – Observation

- Loop bounds
  - Are not an attribute of the dataflow
  - Depends on the architecture
  - Can be limited by
    - Size of the data dimension
    - The storage capacity for the temporal loops
    - Reachable PEs by the multicast network for spatial loops
- Loop bounds determined by the optimization process tge finds the optimal mapping

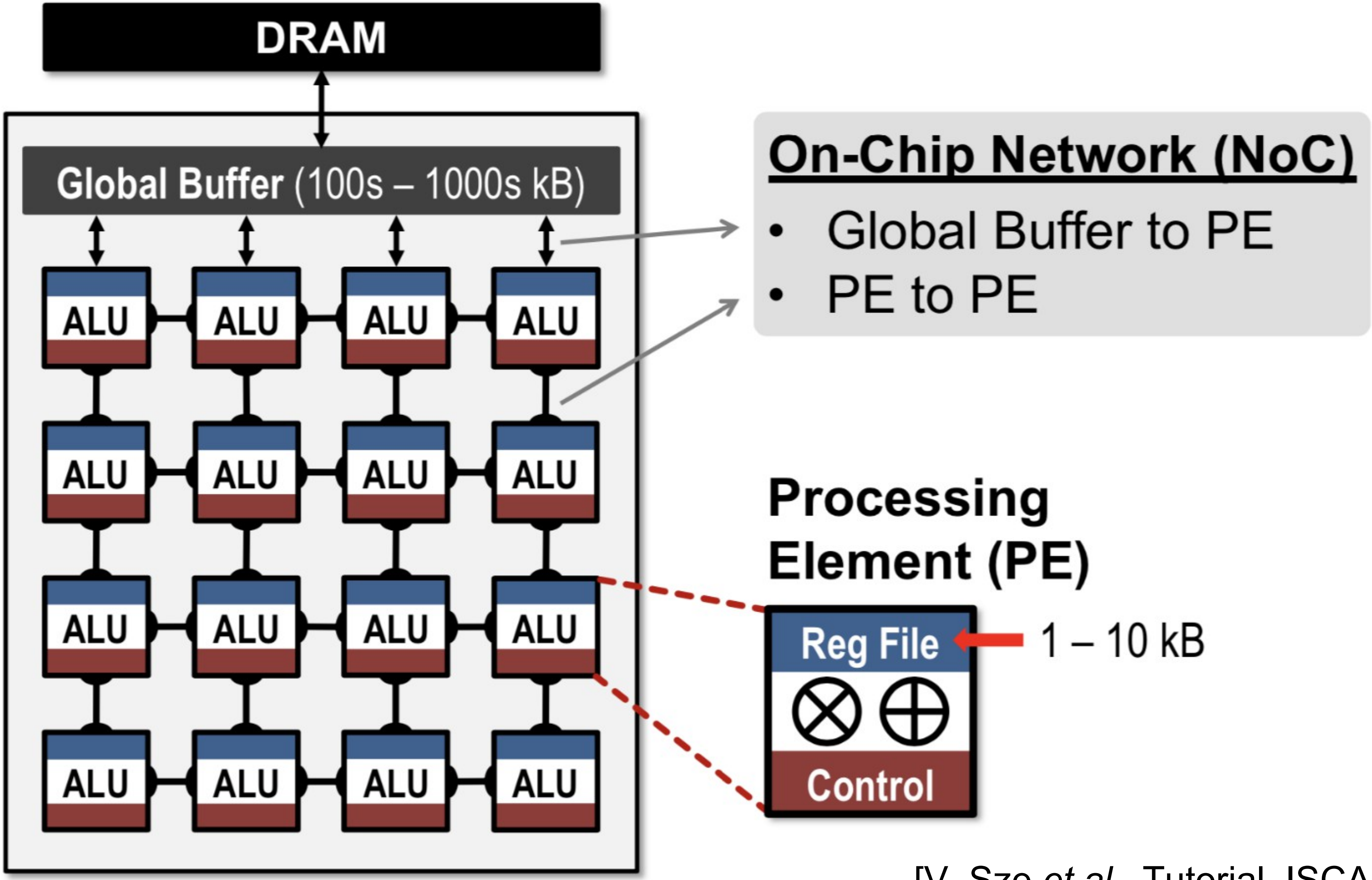
Dataflow Taxonomy

- Recent DNN accelerators can be classified according their supported dataflow
  - Weight Stationary
  - Output Stationary
  - Input Stationary
  - Row Stationary

Recent Work by Dataflow

- Weight Stationary
  - NVDLA [http://nvdla.org/index.html]
  - TPU [https://cloud.google.com/tpu/docs/tpus]
  - Simba [Shao *et al.*, MICRO'19]
- Output Stationary
  - DaDianNao [Y. Chen *et al.*, MICRO'14]
  - DianNao [Y. Chen *et al.*, ASPLOS'14]
  - ShiDianNao [Du *et al.*, ISCA'15]
- Input Stationary
  - SCNN [Parashar, ISCA'17]
- Row Stationary
  - Eyeriss v1 [Y.-H. Chen et al., JSSC'16]
  - Eyeriss v2 [Y.-H. Chen et al., JETCAS'19]

Generic DNN Accelerator Architecture



On-Chip Network (NoC)

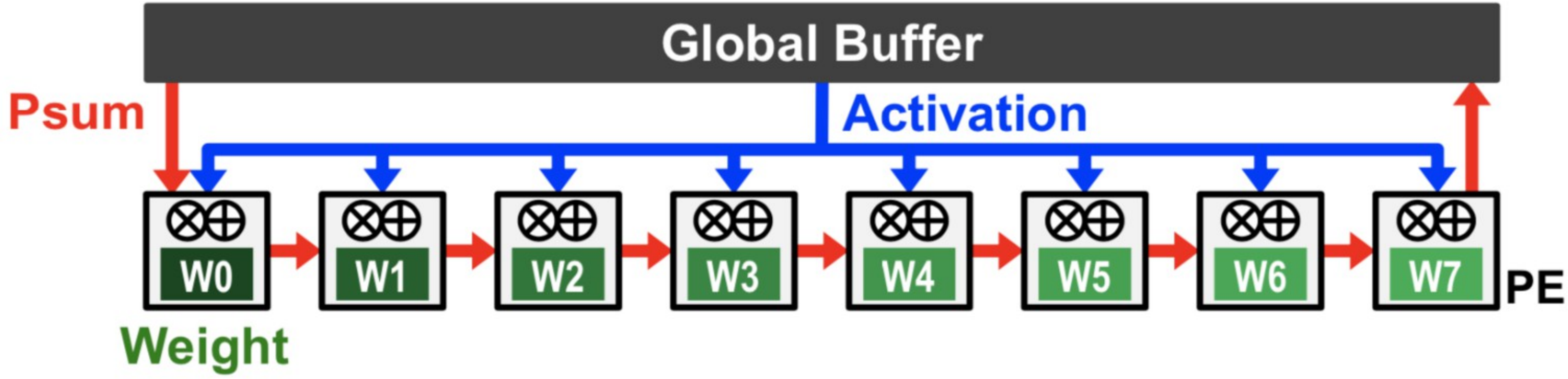
- Global Buffer to PE
- PE to PE

Processing Element (PE)

[V. Sze *et al.*, Tutorial, ISCA'19]

Weight Stationary

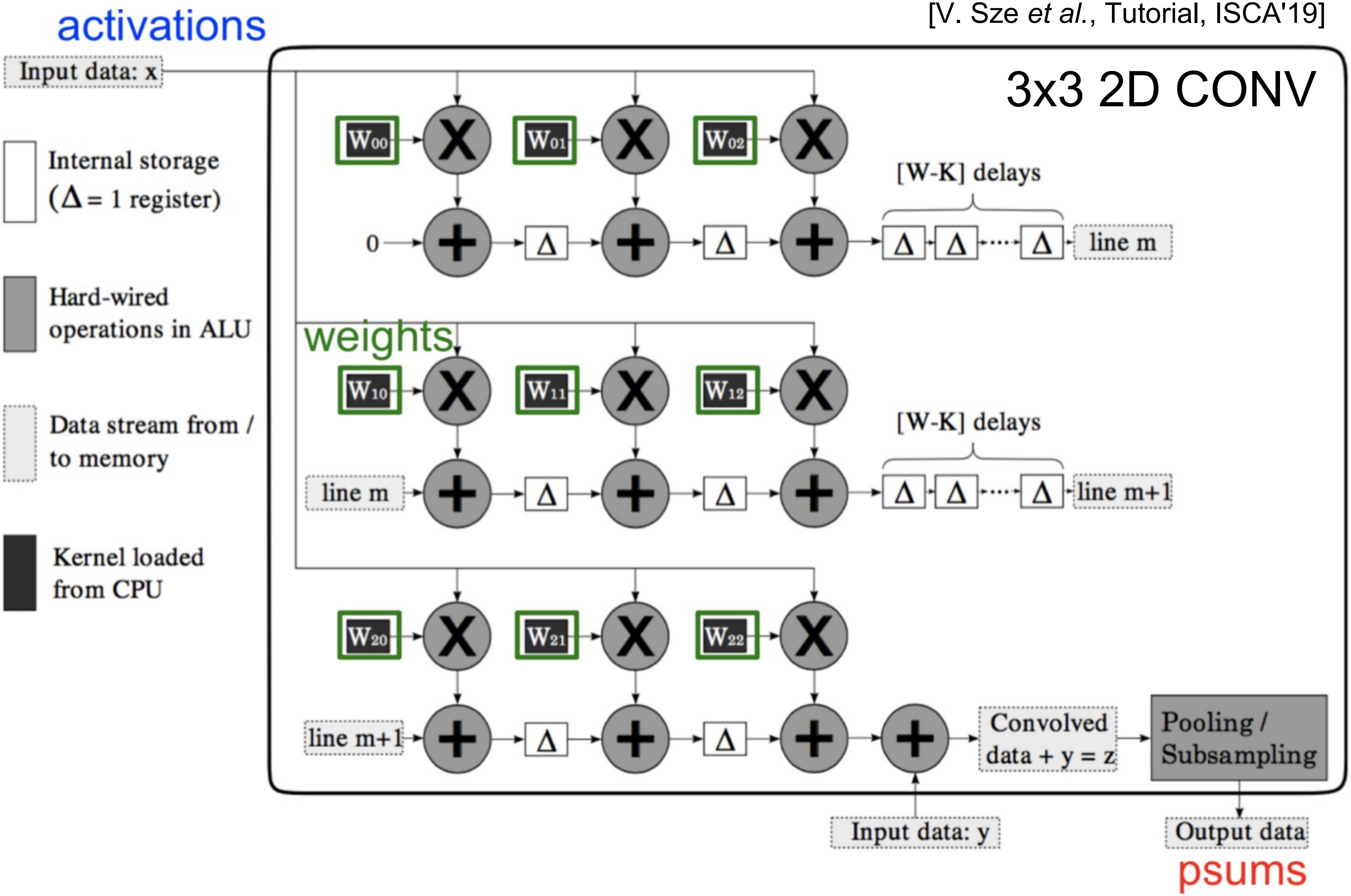
[V. Sze *et al.*, Tutorial, ISCA'19]



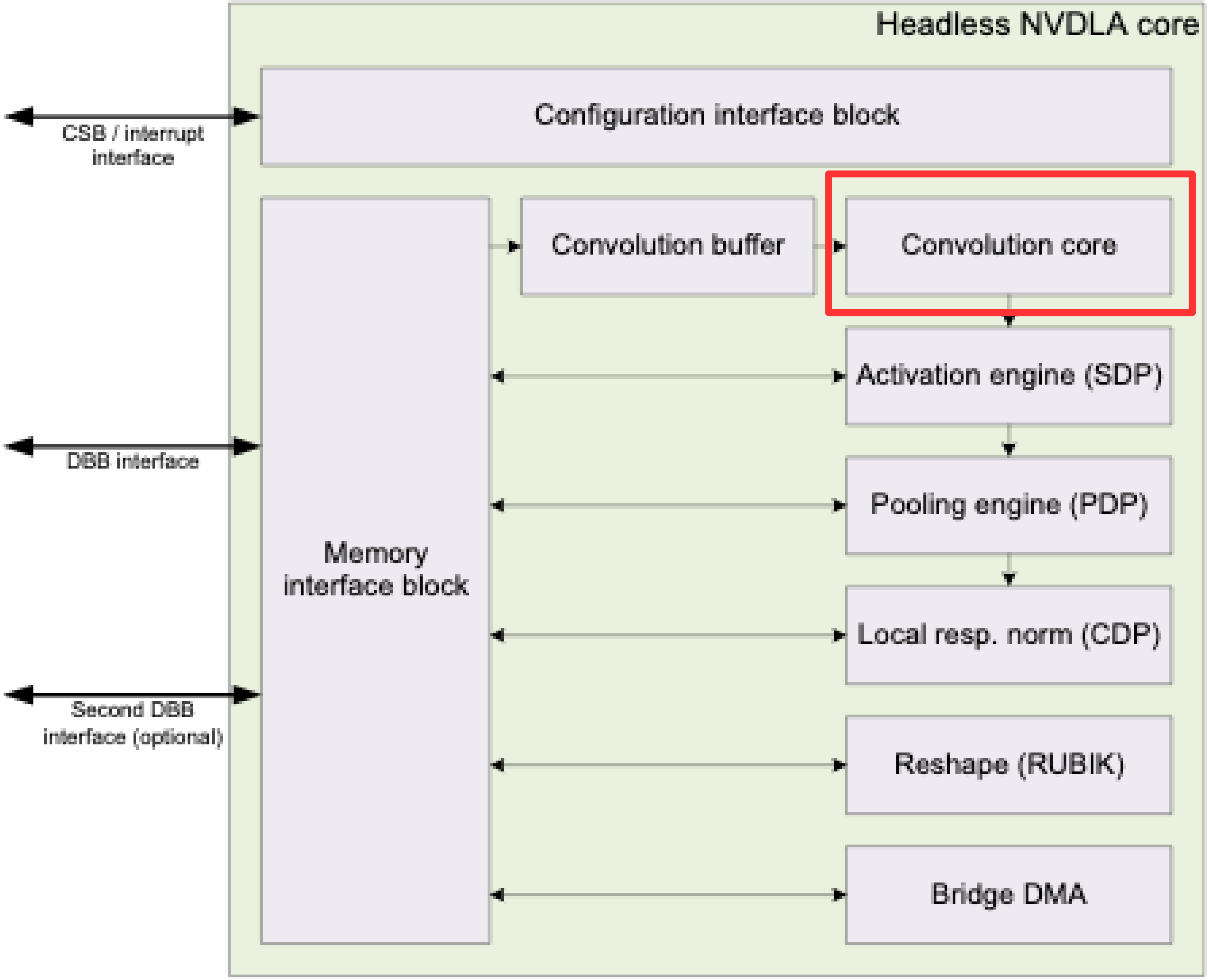
- **Minimize** **weight** read energy consumption
  - Maximize convolutional and filter reuse of weights
- **Broadcast** **activations** and accumulate **partial sums** **spatially** across the PE array

Weight Stationary – neuFlow

[V. Sze *et al.*, Tutorial, ISCA'19]

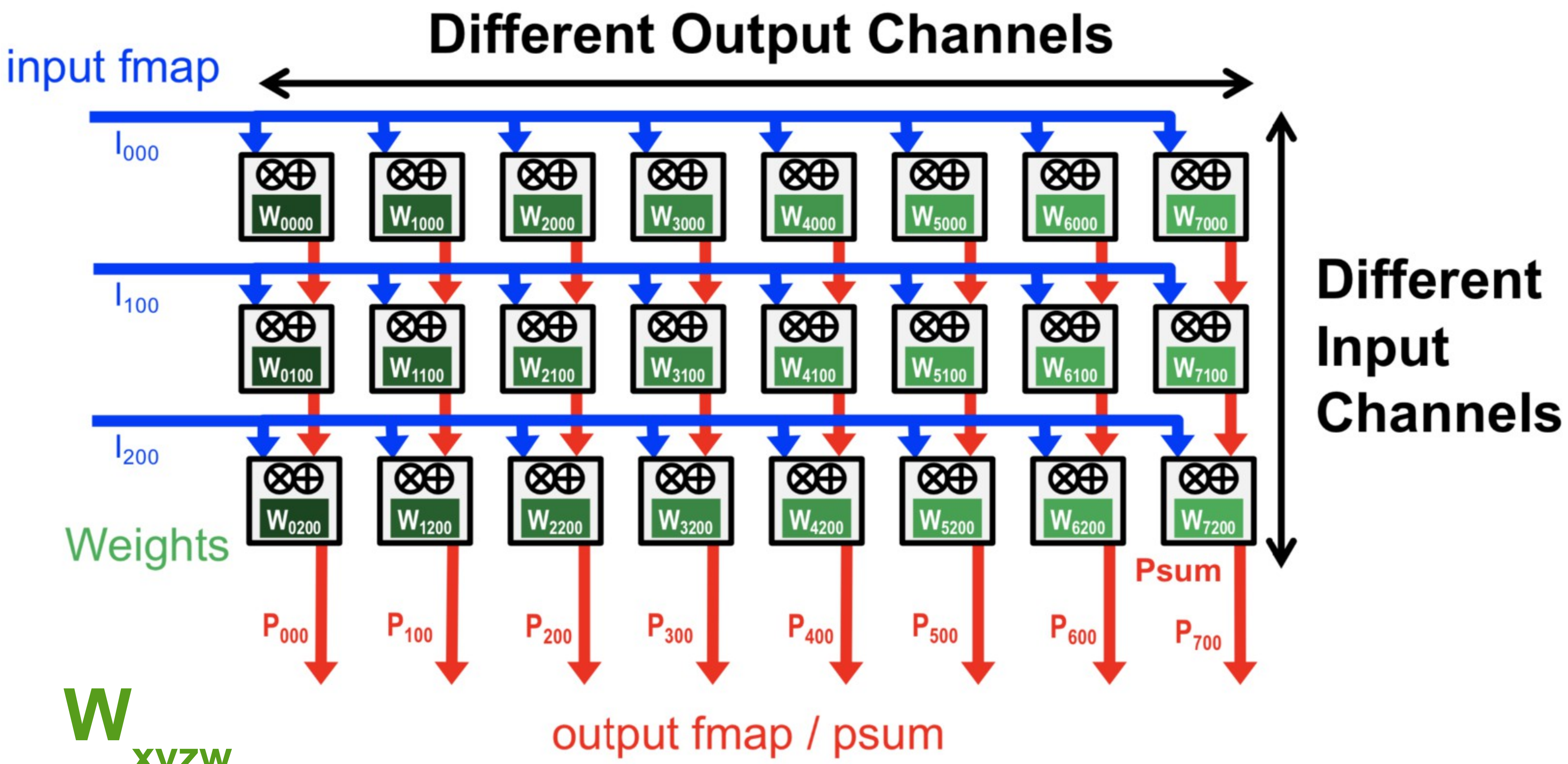


Weight Stationary – NVDLA



[http://nvdla.org/hw/v1/hwarch.html]

Weight Stationary – NVDLA


Different Output Channels
input fmap
I_{000}
I_{100}
I_{200}
W_{0000} W_{1000} W_{2000} W_{3000} W_{4000} W_{5000} W_{6000} W_{7000}
W_{0100} W_{1100} W_{2100} W_{3100} W_{4100} W_{5100} W_{6100} W_{7100}
W_{0200} W_{1200} W_{2200} W_{3200} W_{4200} W_{5200} W_{6200} W_{7200}
Different Input Channels
Weights
Psum
P_{000} P_{100} P_{200} P_{300} P_{400} P_{500} P_{600} P_{700}
output fmap / psum

W_{xyzw}

Filter|Channel|Col|Row

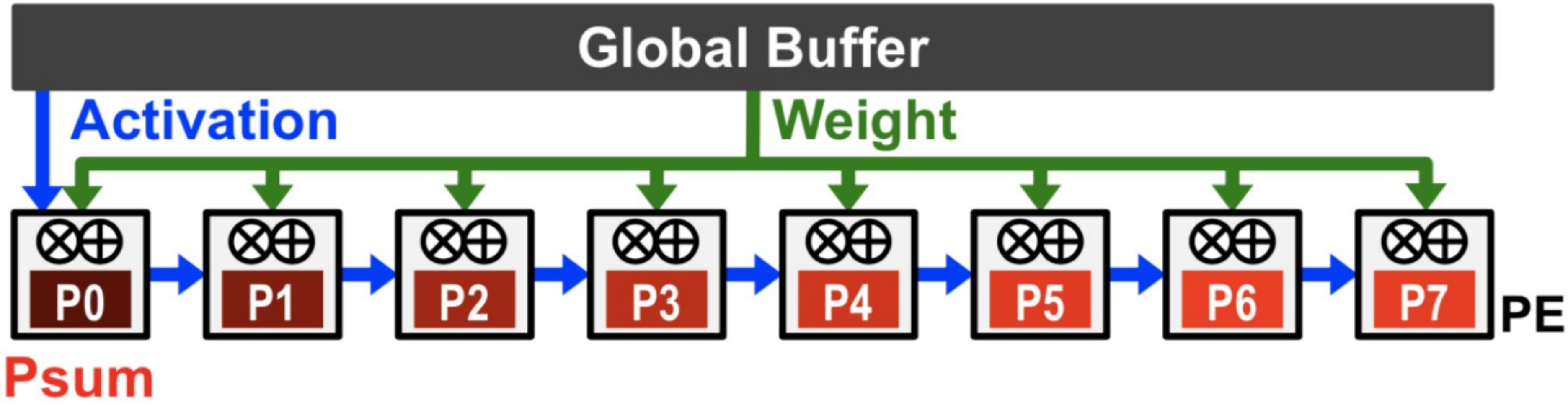
[V. Sze *et al.*, Tutorial, ISCA'19]

NVDL Dataflow as a Loop Nest

```
// I[C,H,W]
// F[M,C,R,S]
// O[M,P,Q]

parallel-for (m=0; m<M; m++)
 parallel-for (c=0; c<C; c++)
  for (r=0; r<R; r++)
   for (s=0; s<S; s++)
    for (p=0; p<P; p++)
     for (q=0; q<Q; q++)
      O[m,p,q] += F[m,c,r,s] *
                         I[c,p+r,q+s];
```

Output Stationary

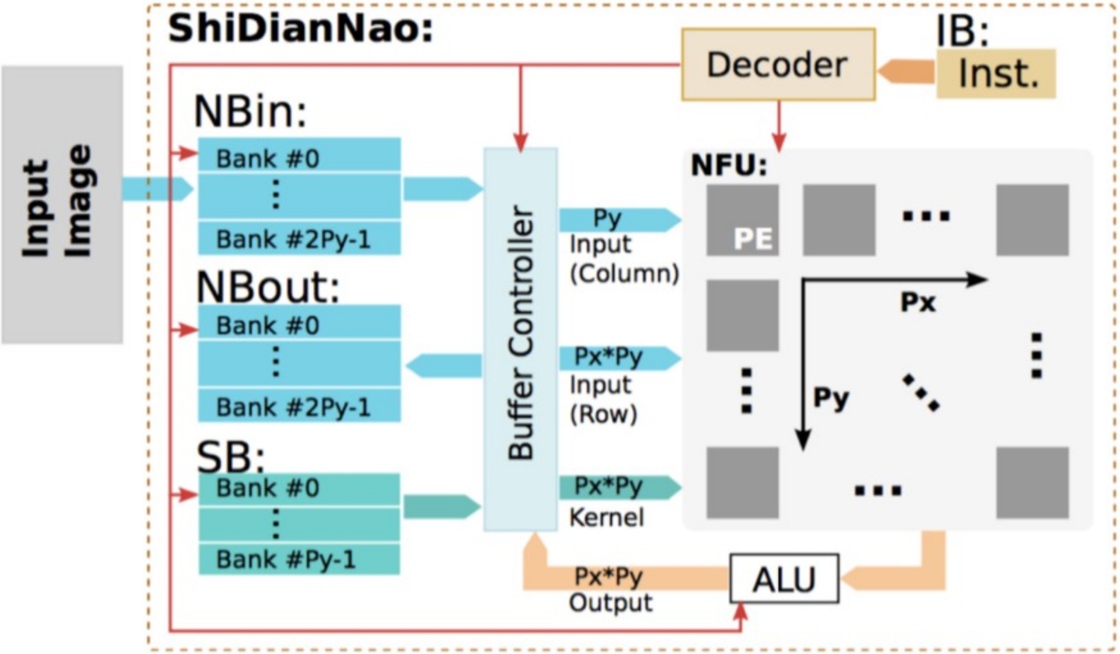
[V. Sze *et al.*, Tutorial, ISCA'19]



- **Minimize** partial sum R/W energy consumption
  - maximize local accumulation
- **Broadcast/Multicast** filter weights and **reuse** activations **spatially** across the PE array

Ouput Stationary – ShiDianNao

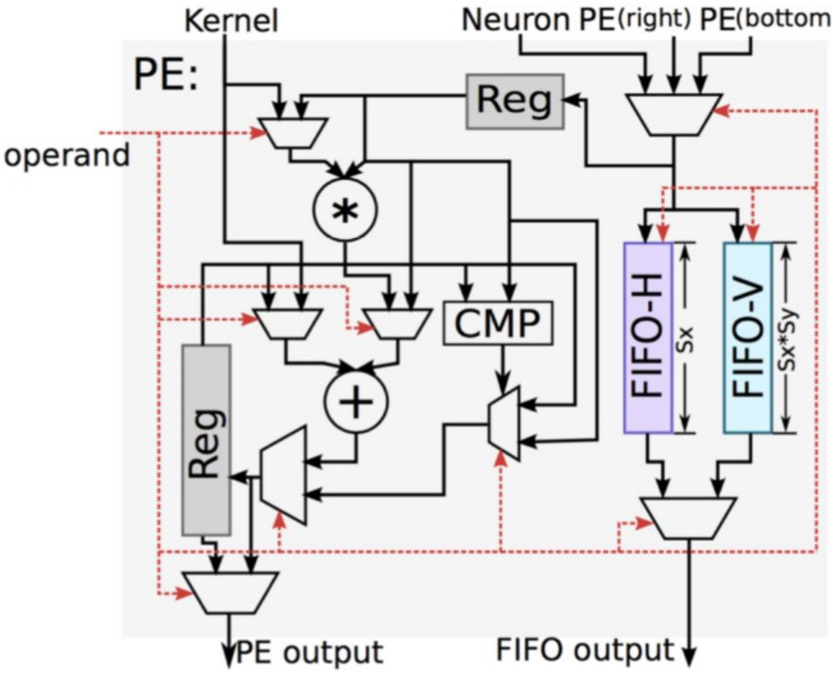
Top-Level Architecture



PE Architecture

weights

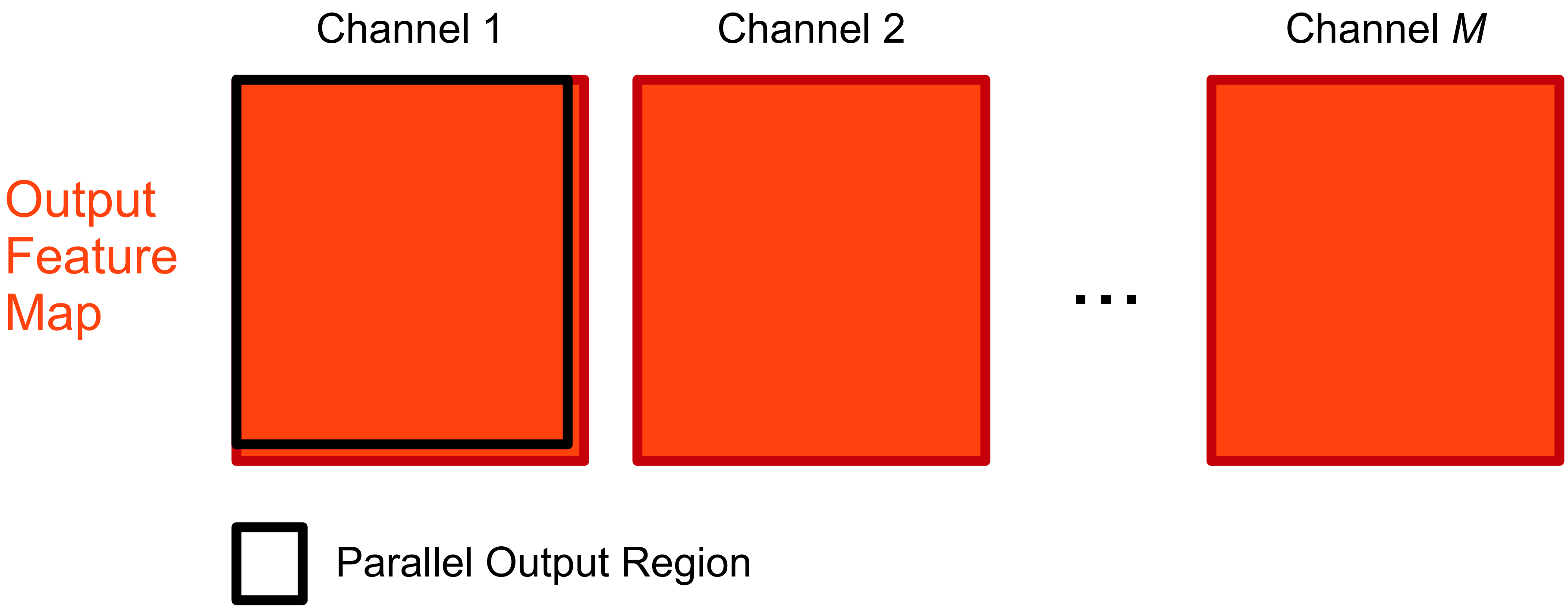
activations



psums

[V. Sze *et al.*, Tutorial, ISCA'19]
[Du et al., ISCA'15]

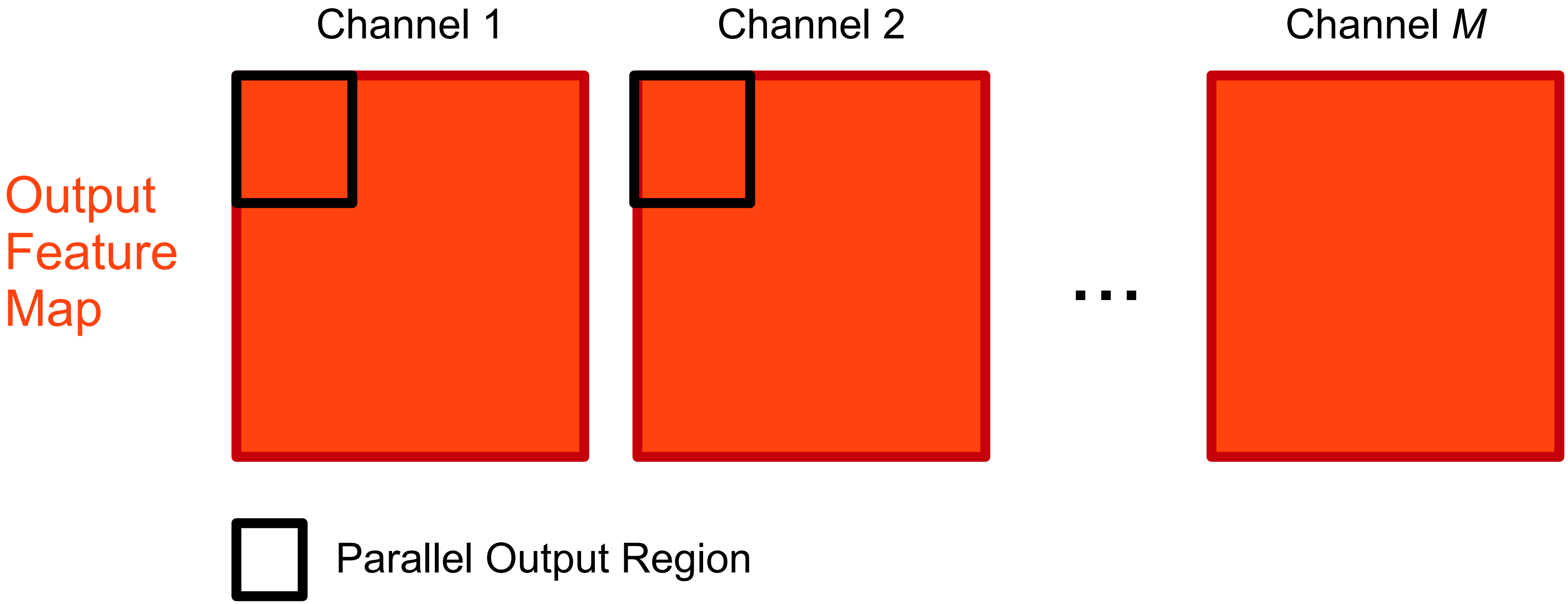
Output Stationary – Variation 1



Single output channel
Multiple output activations

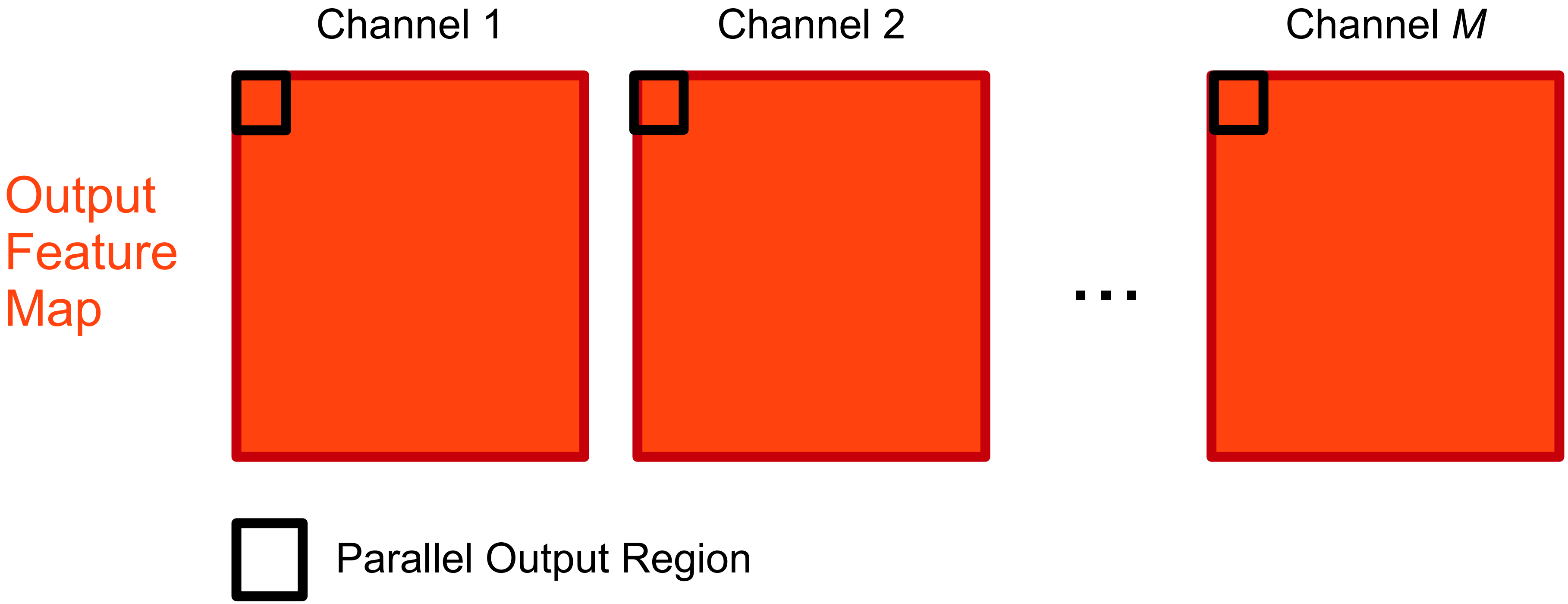
(used in CONV layers)

Output Stationary – Variation 2



Multiple output channels
Multiple output activations

Output Stationary – Variation 3

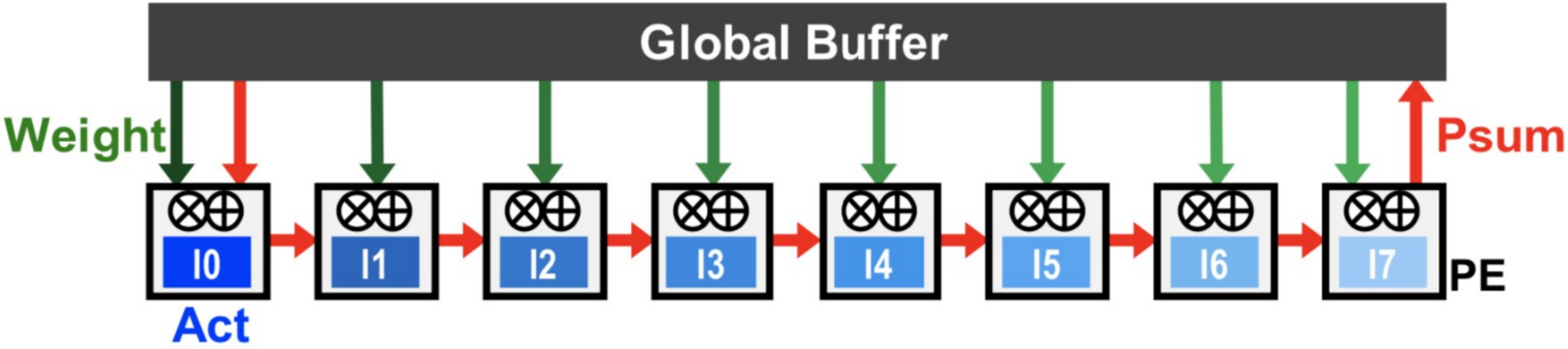


Multiple output channels
Single output activation

(used in FC layers)

Input Stationary

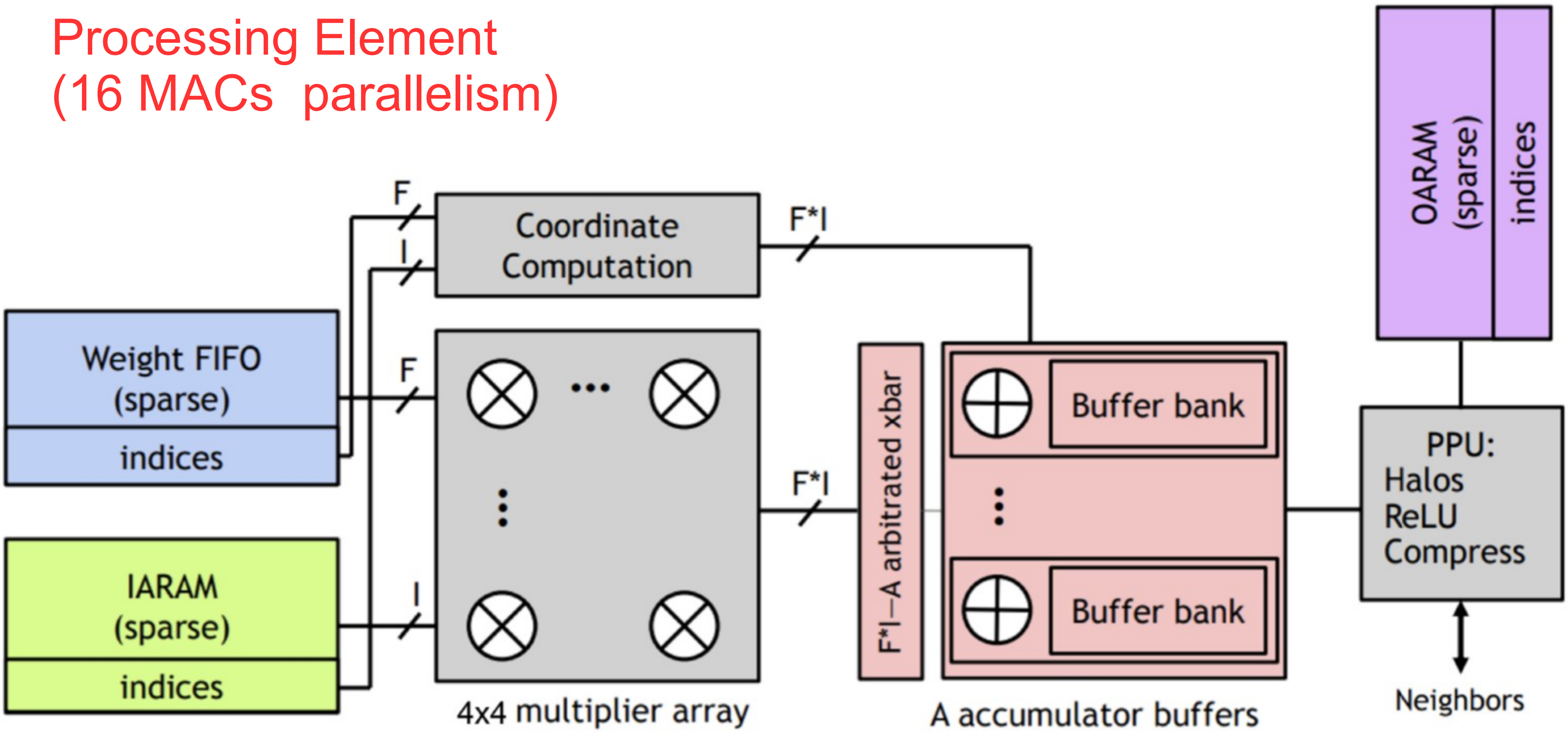
[V. Sze *et al.*, Tutorial, ISCA'19]



- Minimize **activation** read energy consumption
  - Maximize input fmap reuse
- **Unicast weights** and **accumulate psums spatially** across the PE array

Input Stationary – SCNN

Processing Element
(16 MACs parallelism)

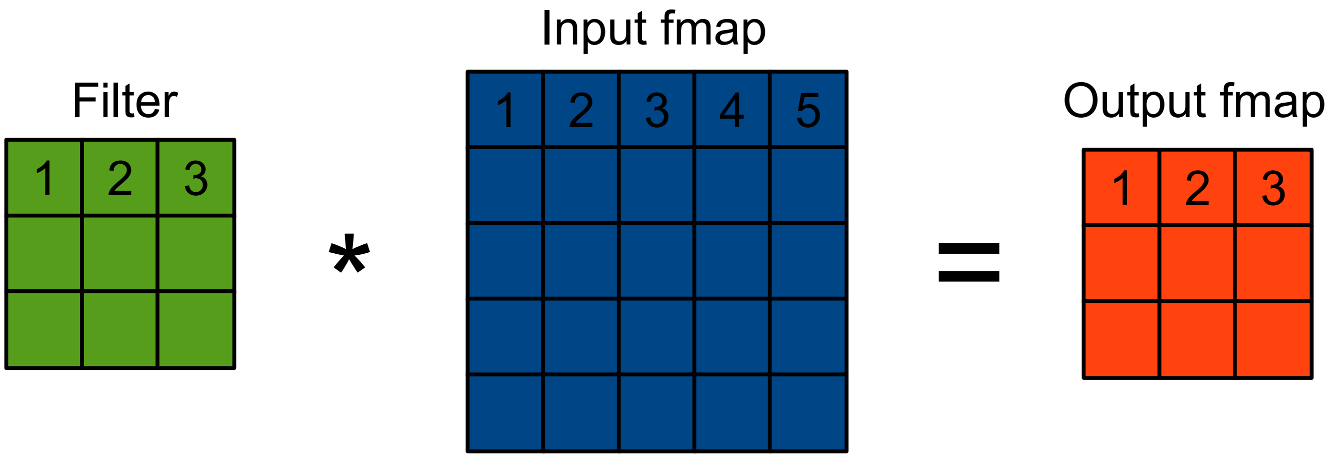


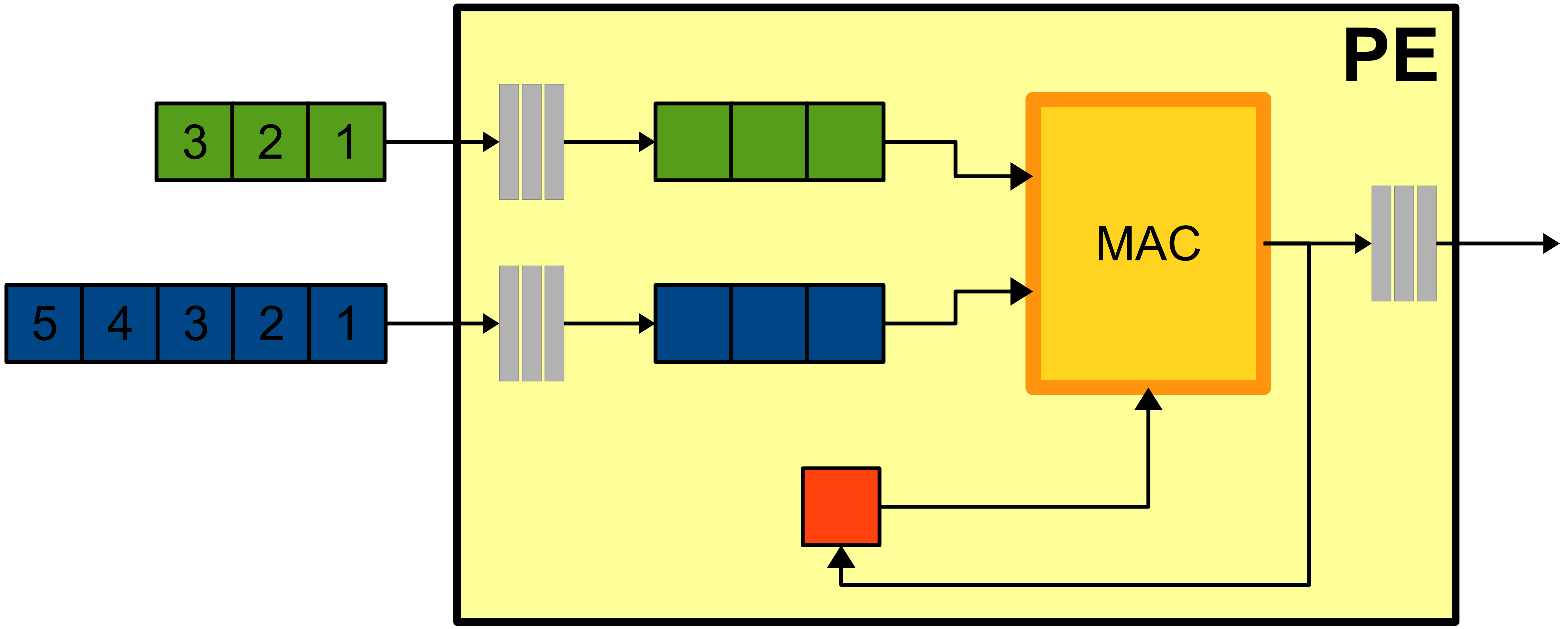
[V. Sze *et al.*, Tutorial, ISCA'19]
[Parashar, ISCA'17]

Row Stationary

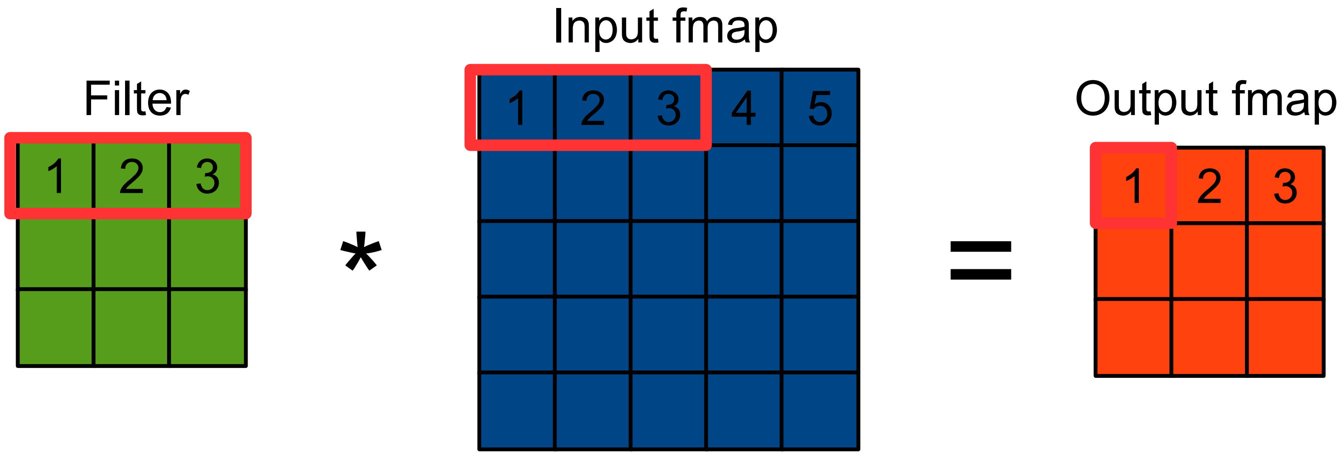
- Aims to maximize the reuse for all types of data
- Assigns the processing of a 1D row convolution into each PE
  - Keeps the row of a filter stationary into the PE
  - Streams input activations into the PE
  - PE does the MACs for each sliding window at a time → Use just one memory space for the accumulation of the partial sums
  - Overlap of input activations between different sliding windows → Input activations reused

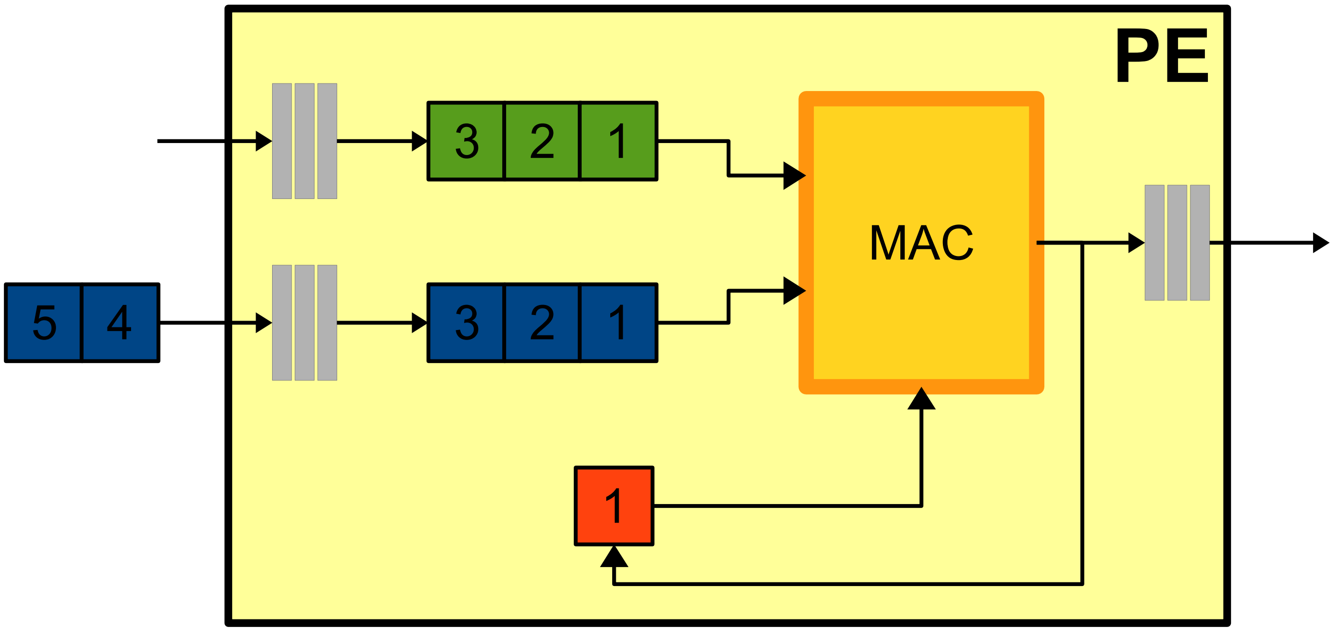
Row Stationary – 1D Conv



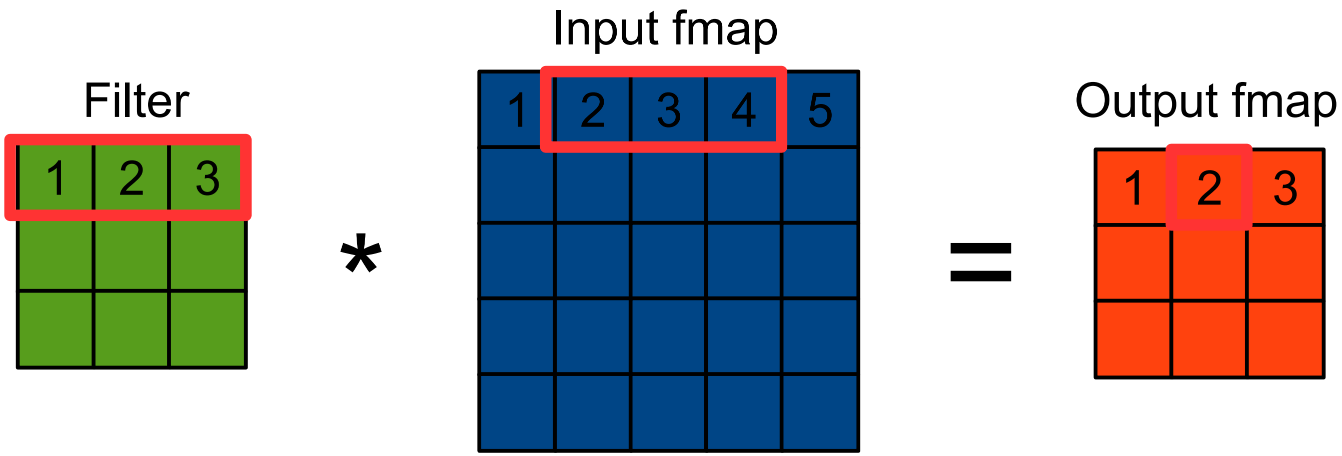


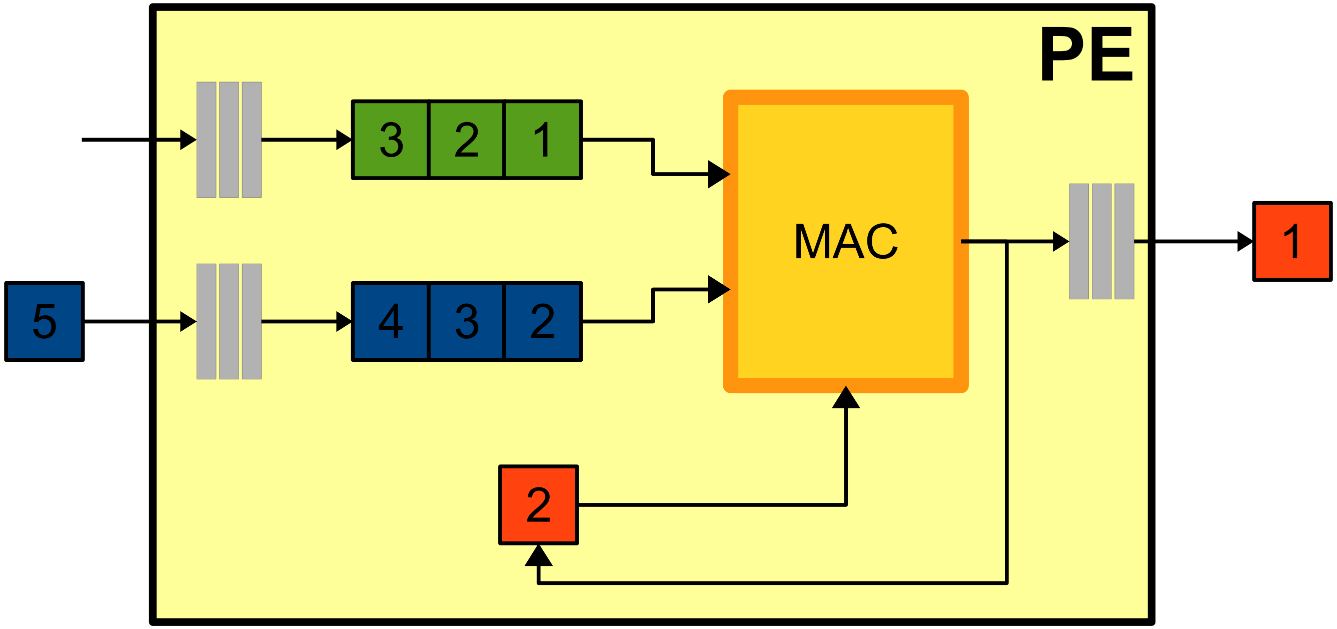
Row Stationary – 1D Conv



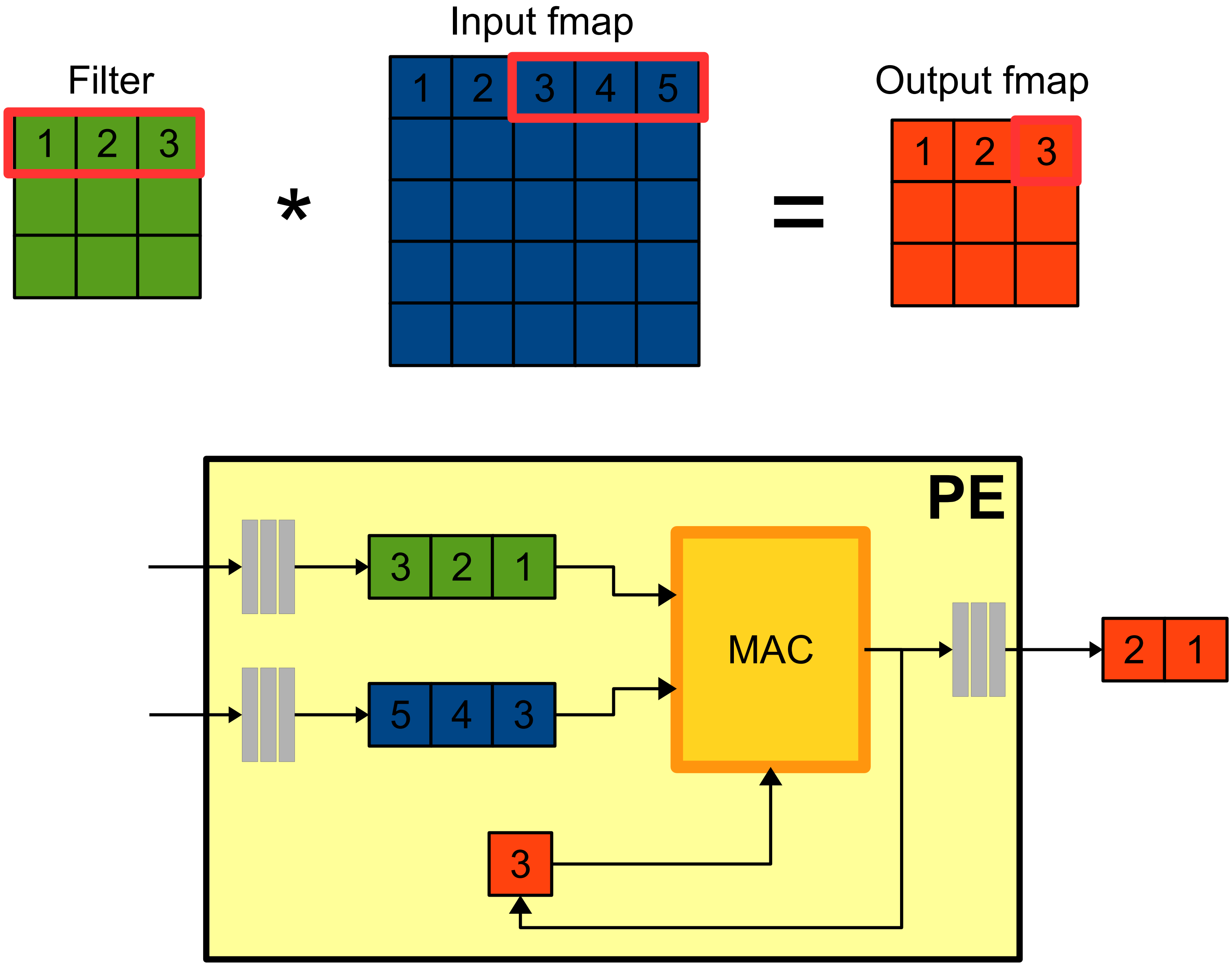


Row Stationary – 1D Conv

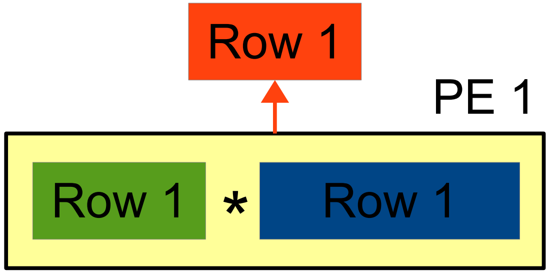


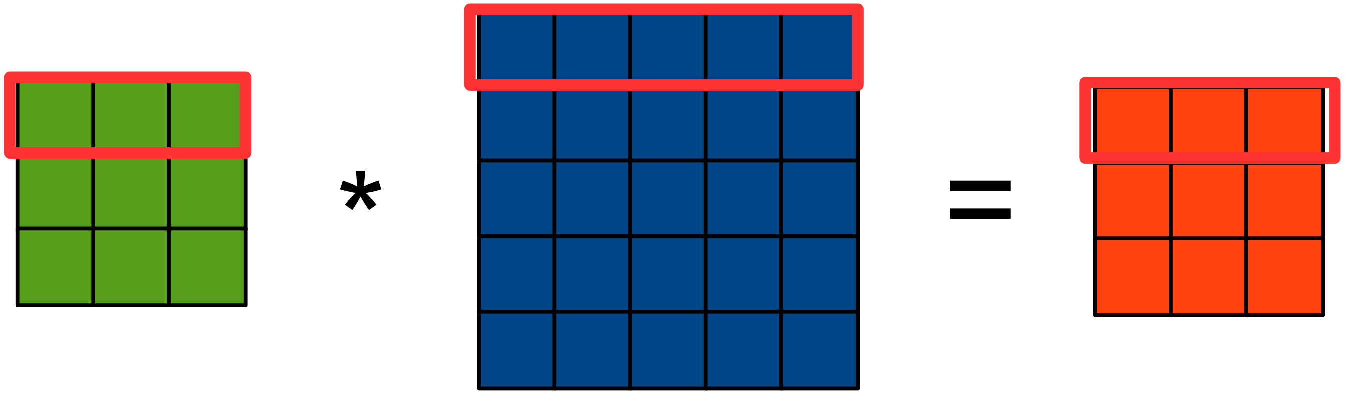


Row Stationary – 1D Conv

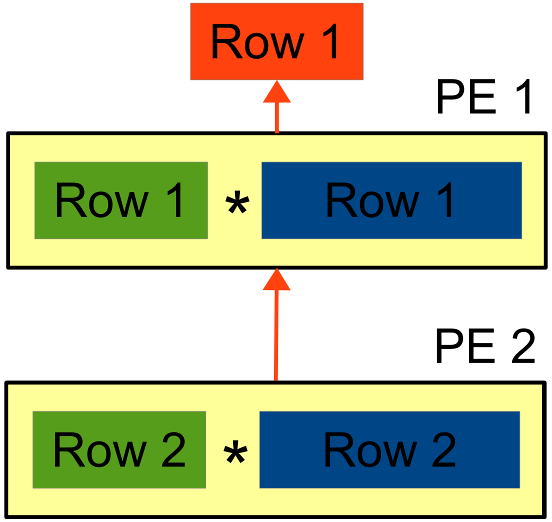


Row Stationary – 2D Conv





Row Stationary – 2D Conv



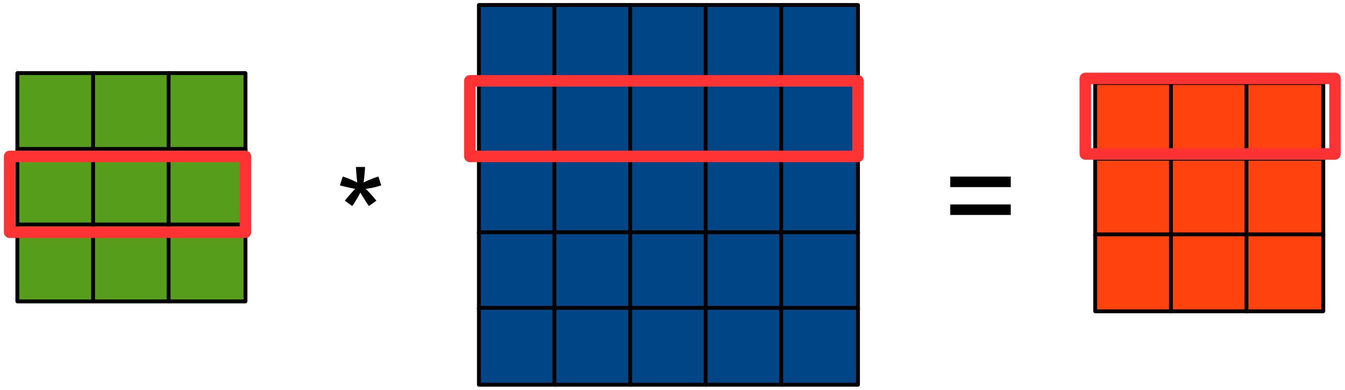
Row 1

PE 1

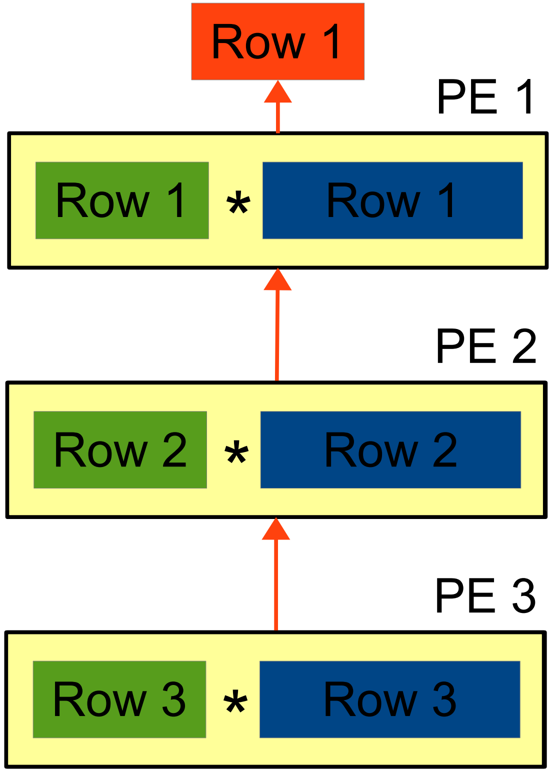
Row 1 * Row 1

PE 2

Row 2 * Row 2



Row Stationary – 2D Conv



Row 1

PE 1

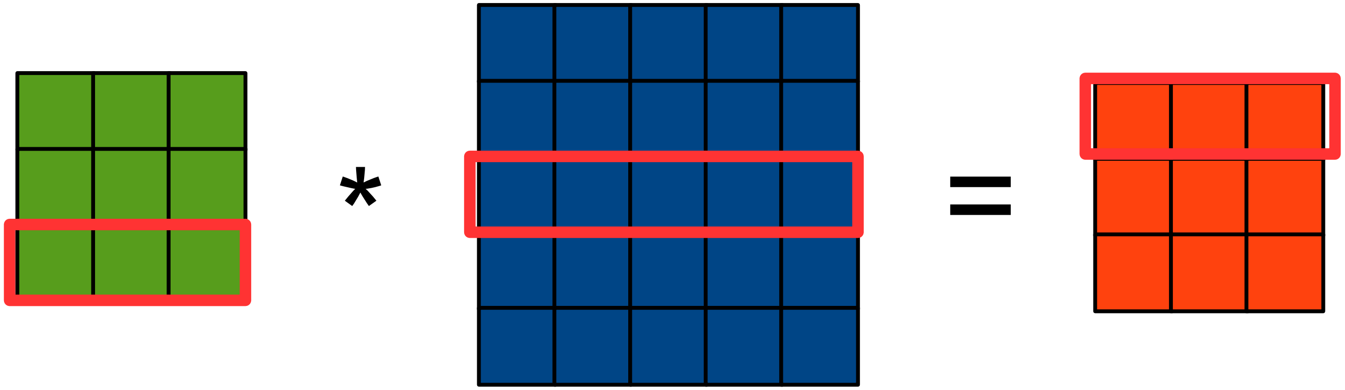
Row 1 * Row 1

PE 2

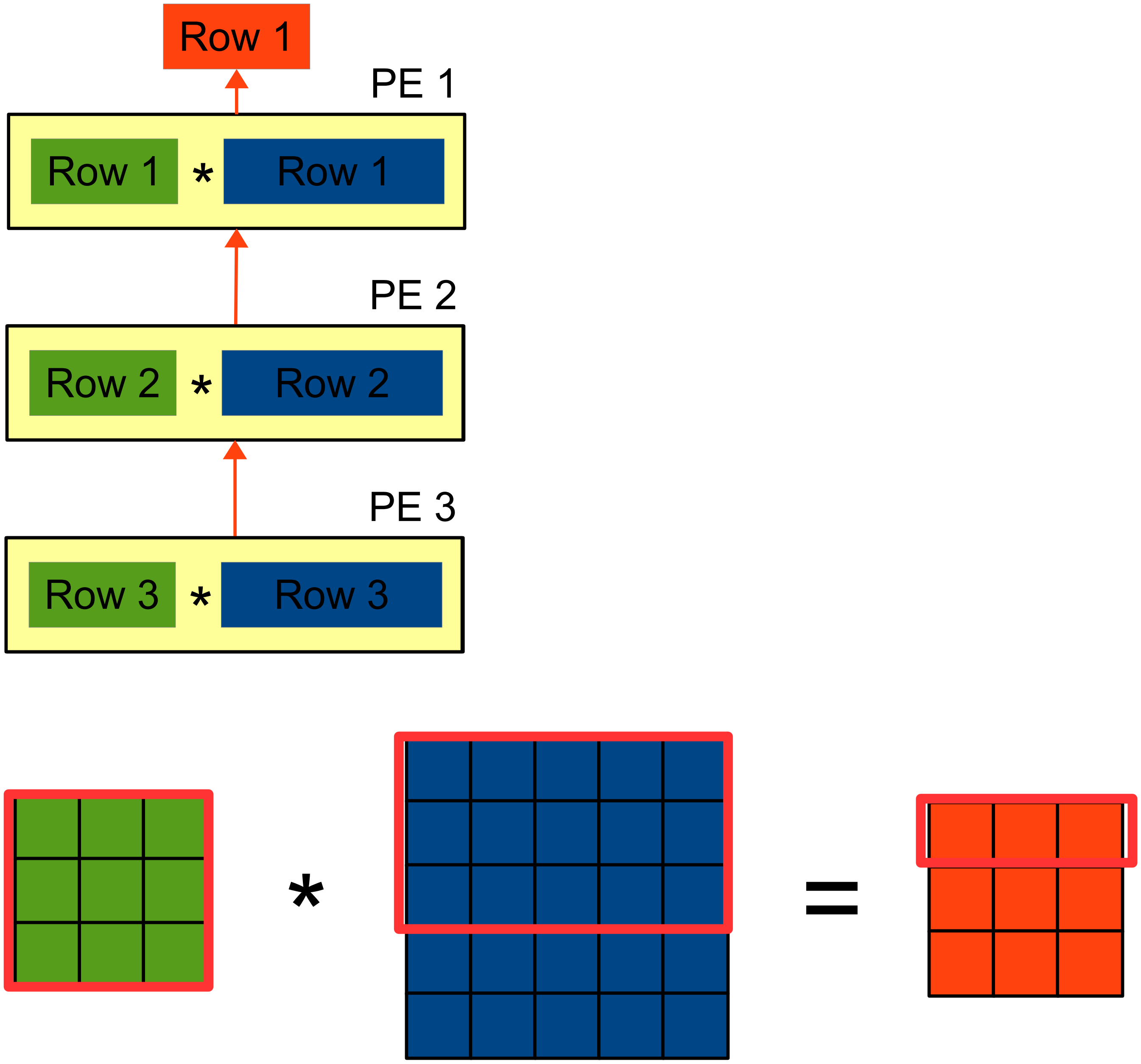
Row 2 * Row 2

PE 3

Row 3 * Row 3



Row Stationary – 2D Conv



Row 1

PE 1

Row 1 * Row 1

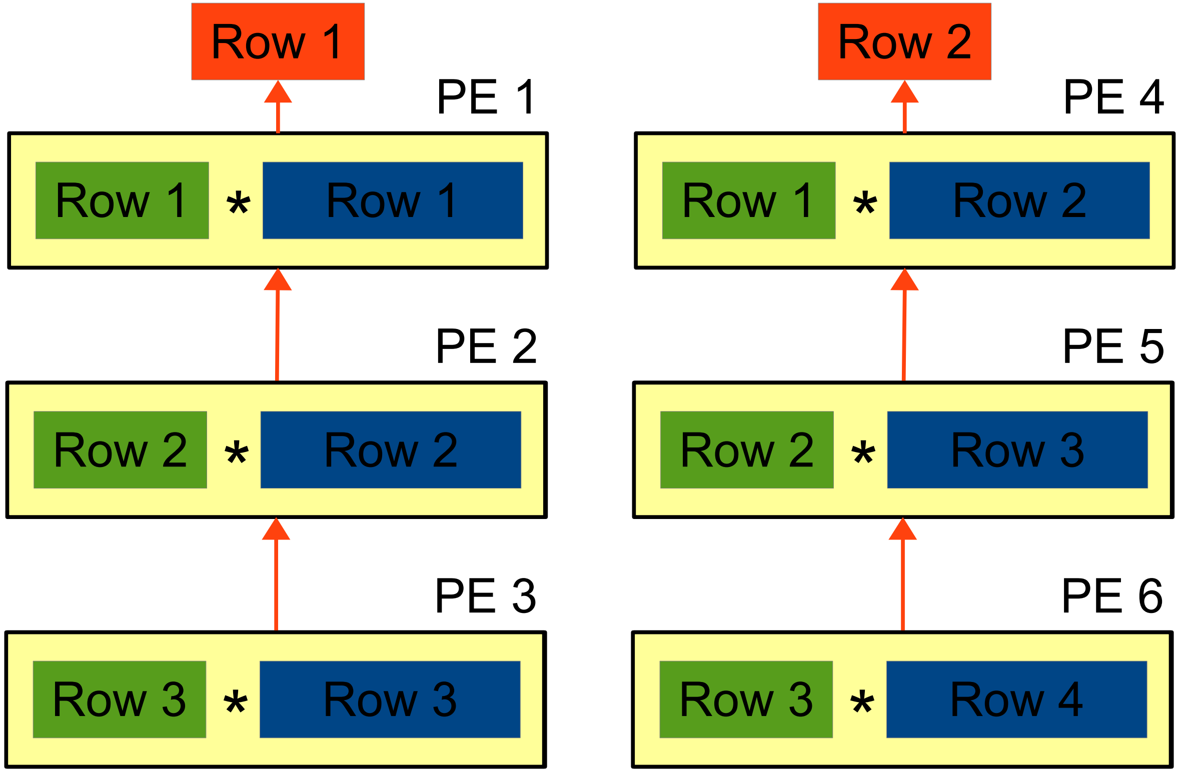
PE 2

Row 2 * Row 2

PE 3

Row 3 * Row 3

Row Stationary – 2D Conv



Row 1

PE 1

Row 1 * Row 1

PE 2

Row 2 * Row 2

PE 3

Row 3 * Row 3

Row 2

PE 4

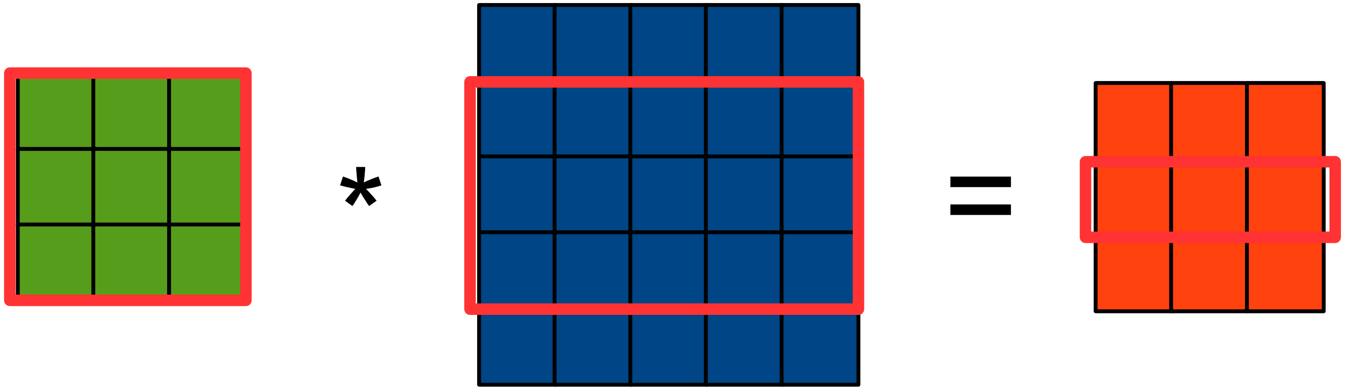
Row 1 * Row 2

PE 5

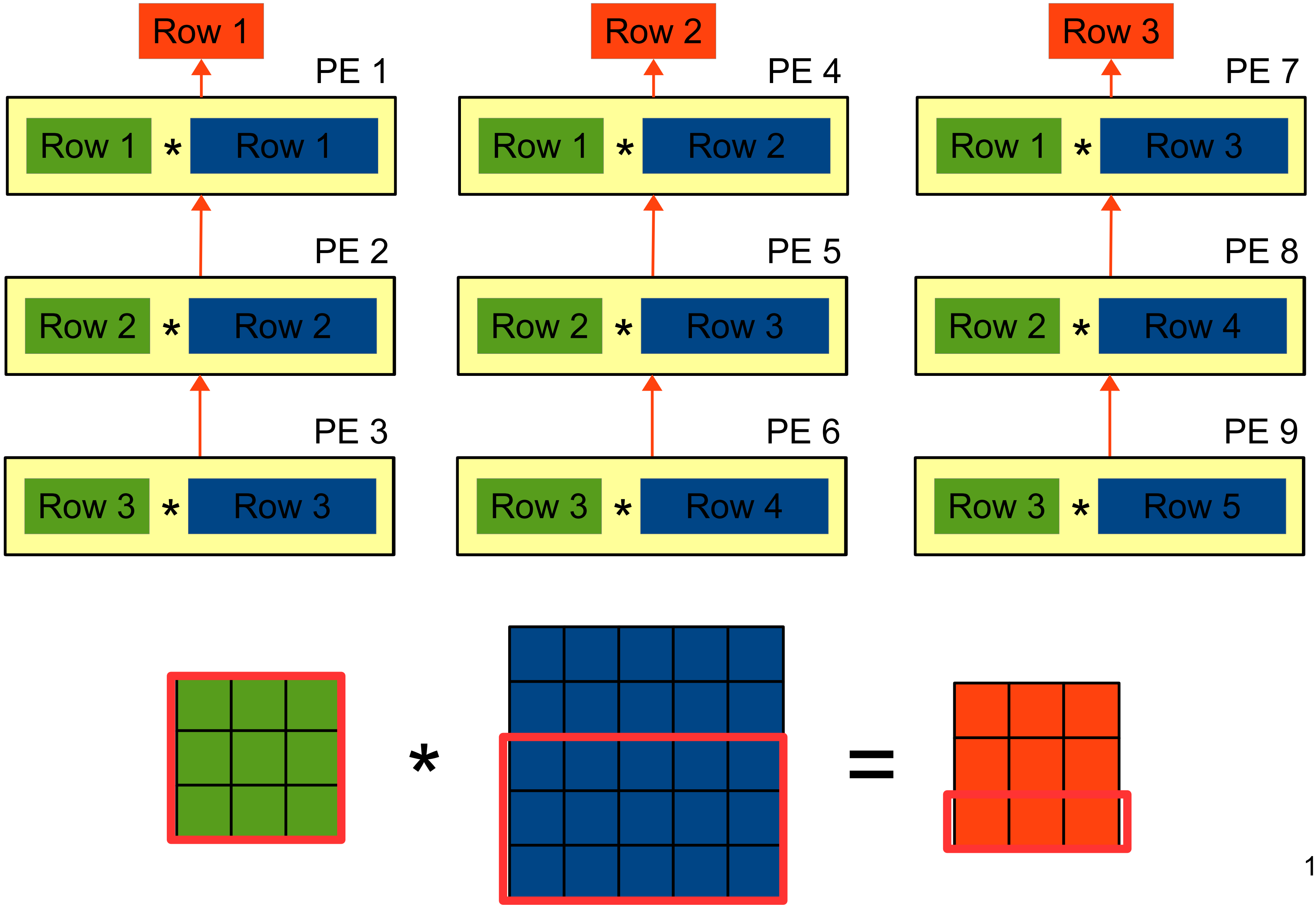
Row 2 * Row 3

PE 6

Row 3 * Row 4



Row Stationary – 2D Conv



Row 1

PE 1

Row 1 * Row 1

PE 2

Row 2 * Row 2

PE 3

Row 3 * Row 3

Row 2

PE 4

Row 1 * Row 2

PE 5

Row 2 * Row 3

PE 6

Row 3 * Row 4

Row 3

PE 7

Row 1 * Row 3

PE 8

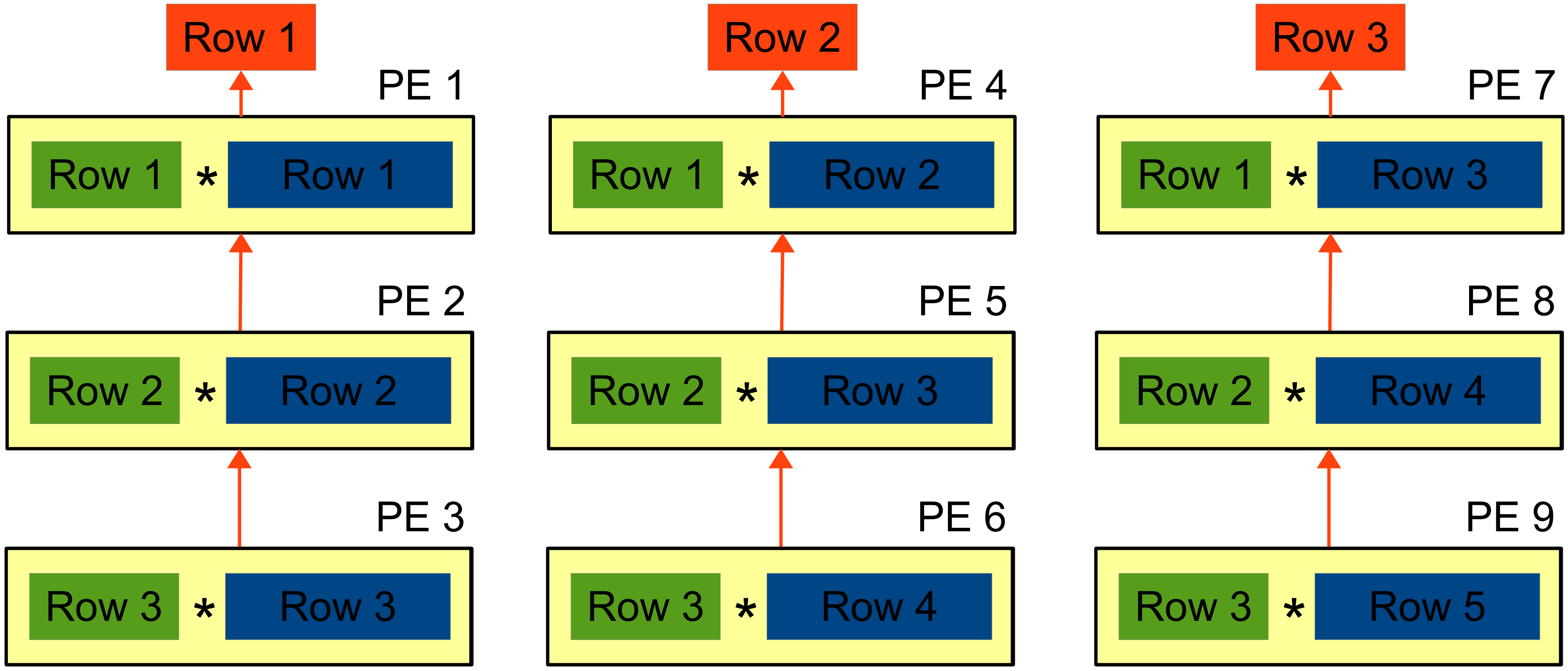
Row 2 * Row 4

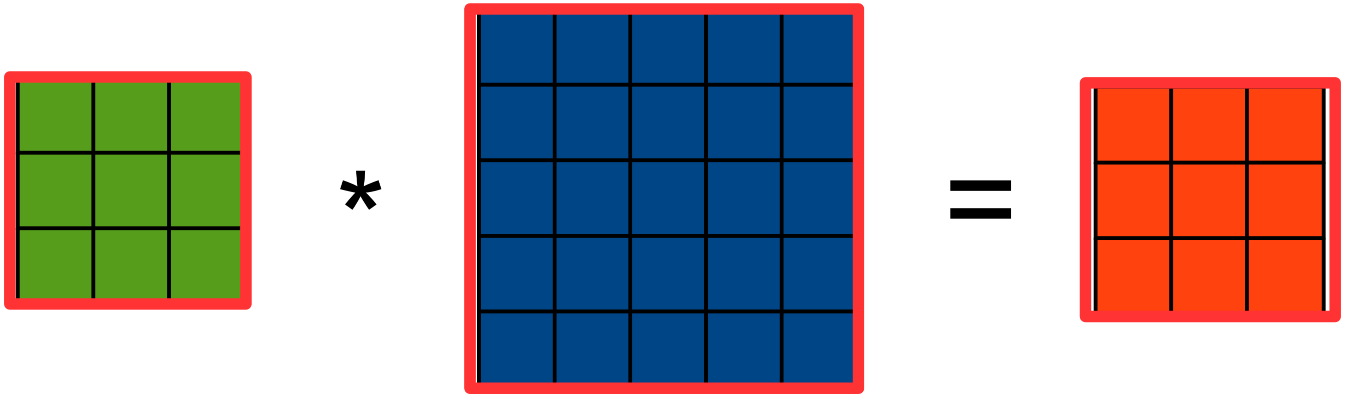
PE 9

Row 3 * Row 5

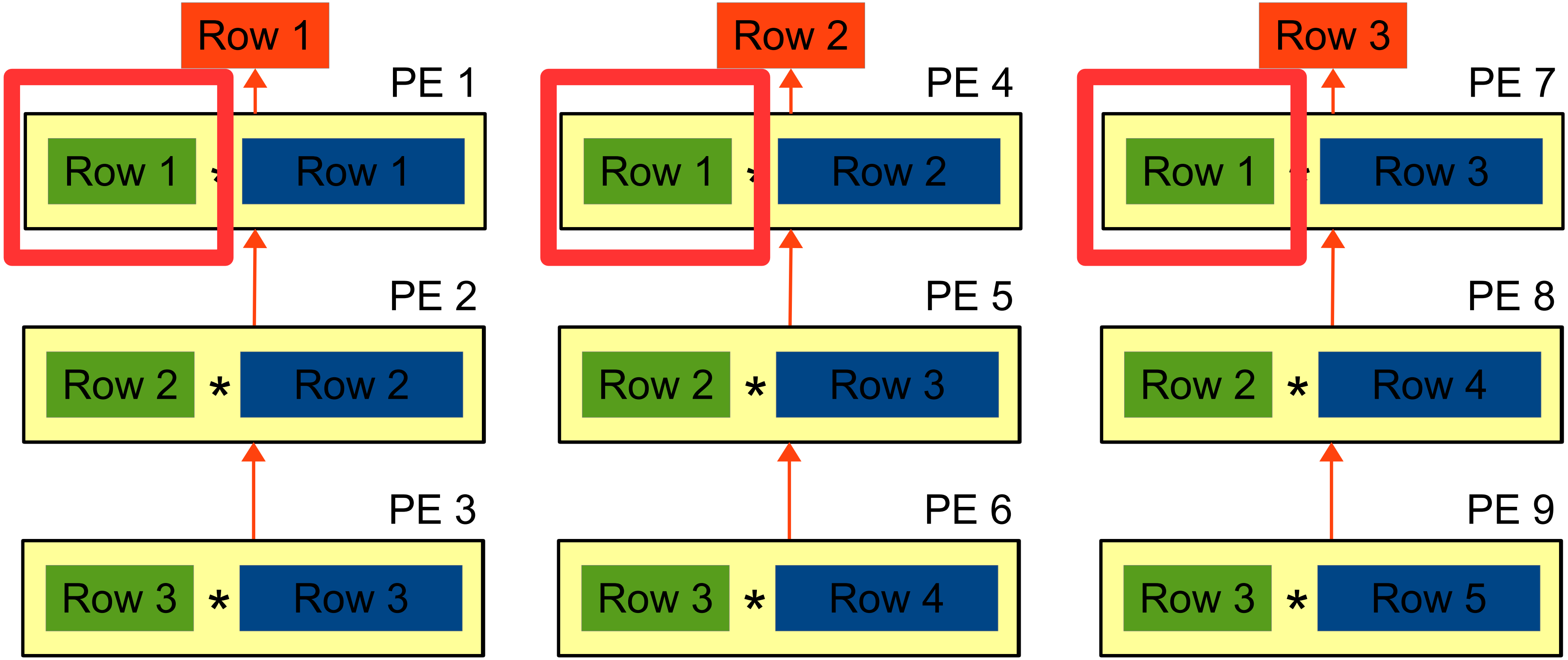
* =

Row Stationary – 2D Conv



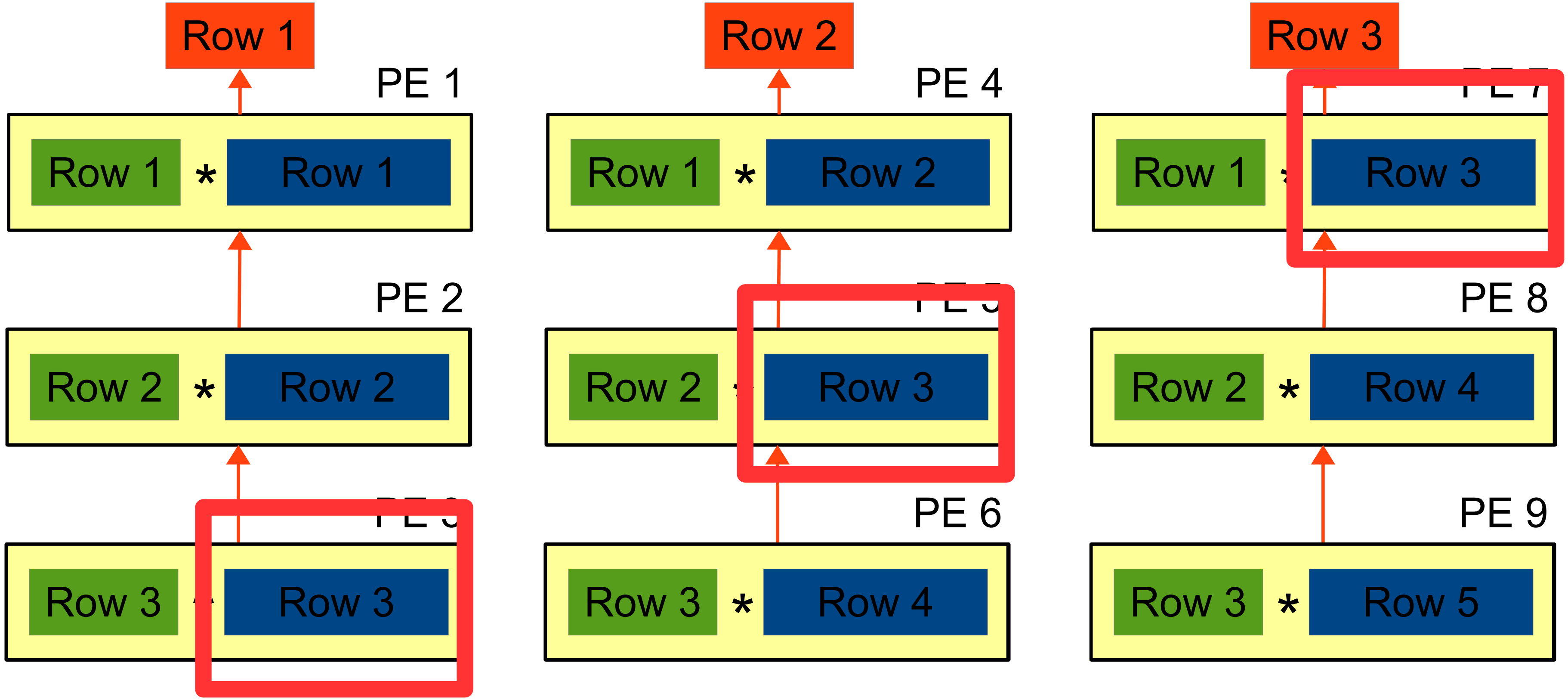


Row Stationary – Reuse Opportunities



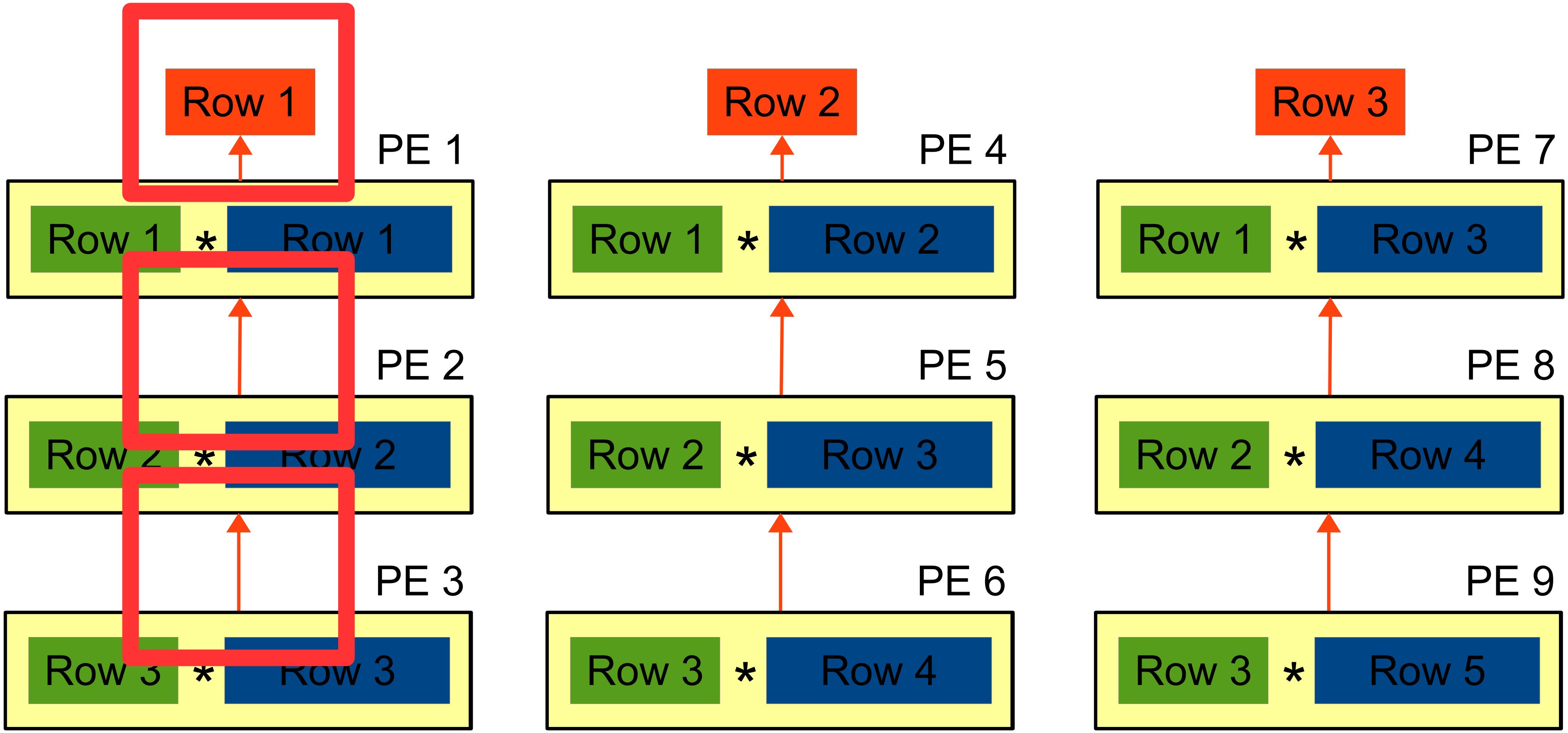
Each row filter reused across multiple PEs horizontally

Row Stationary – Reuse Opportunities



Each row of input activations reused across multiple PE diagonally

Row Stationary – Reuse Opportunities



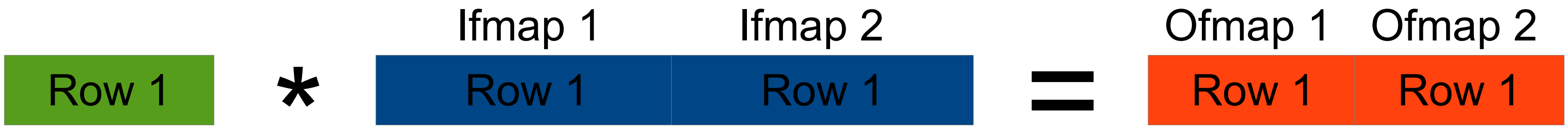
Each row of partial sums are accumulated across the PE vertically

Row Stationary – High Dimensional CONV

- Problem
  - Multiple **ifmaps**? Multiple **filters**? Multiple **channels**?
- Solution
  - **Multiple rows can be mapped onto the same PE**
  - Concatenating or interleaving data of different **ifmaps**, **filters**, **channels**

Row Stationary – High Dimensional CONV

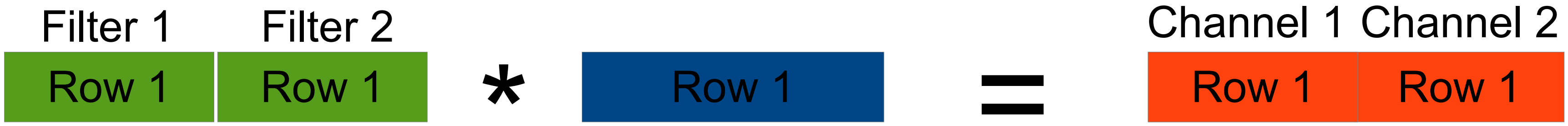
- Multiple input feature maps



Ifmap 1 Ifmap 2 Ofmap 1 Ofmap 2

Row 1 * Row 1 Row 1 = Row 1 Row 1

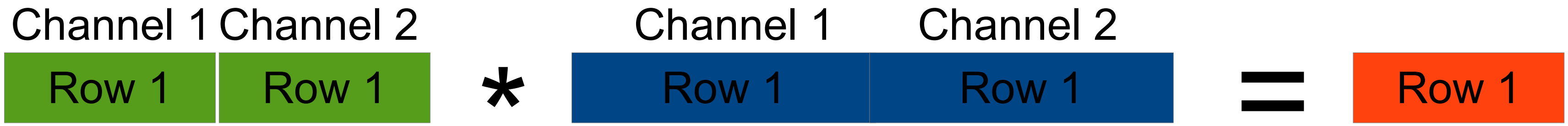
- Multiple filters



Filter 1 Filter 2 Channel 1 Channel 2

Row 1 Row 1 * Row 1 = Row 1 Row 1

- Multiple channels

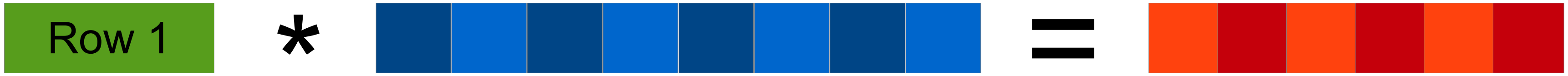


Channel 1 Channel 2 Channel 1 Channel 2

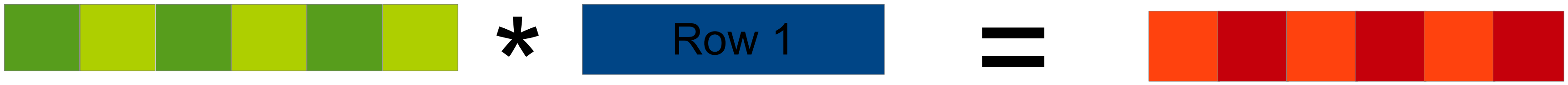
Row 1 Row 1 * Row 1 Row 1 = Row 1

Row Stationary – High Dimensional CONV

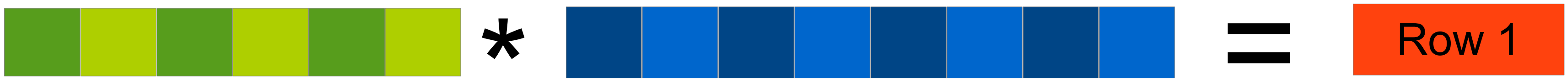
- Multiple input feature maps



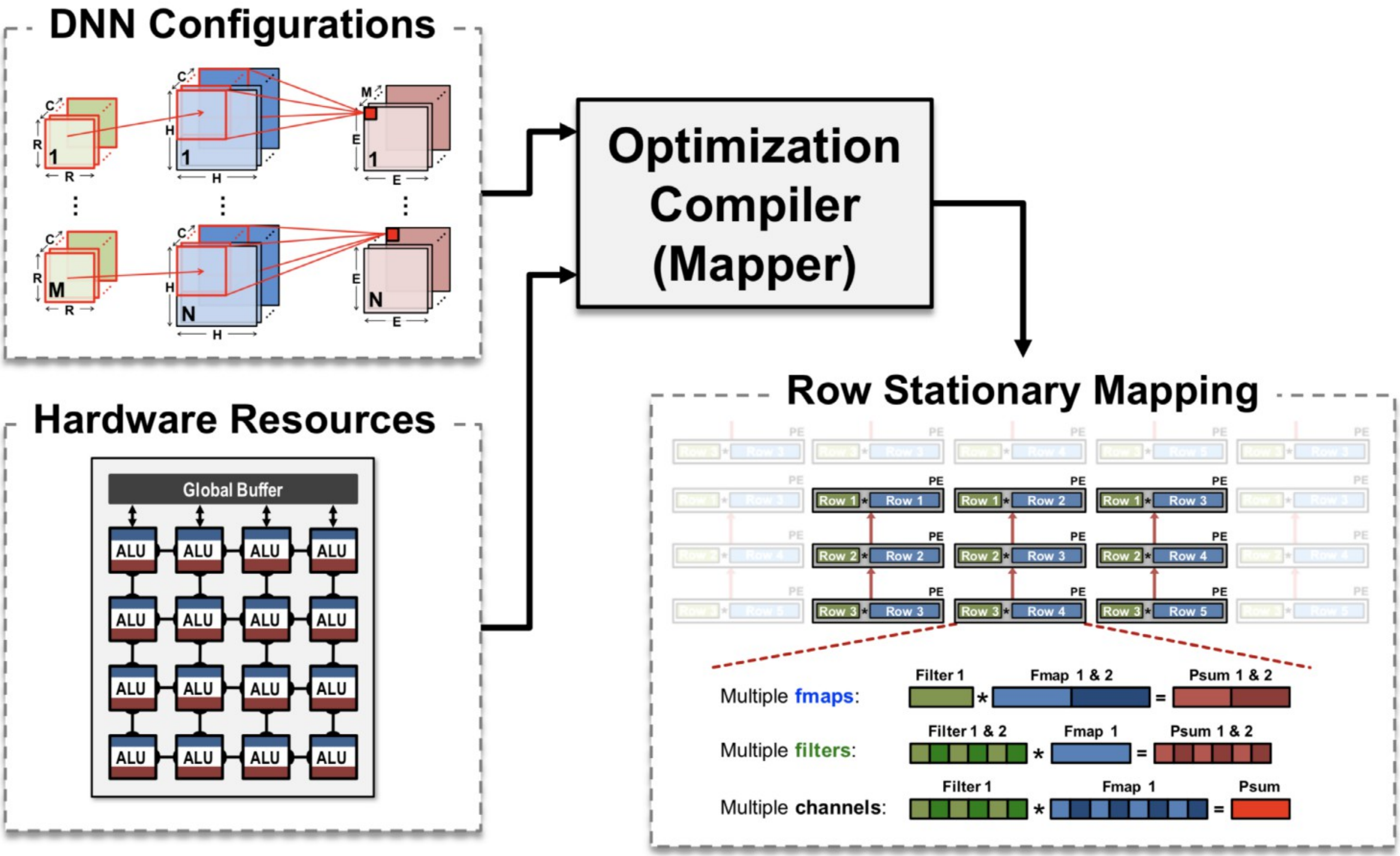
- Multiple filters



- Multiple channels



Optimal Mapping in Row Stationary



DNN Configurations

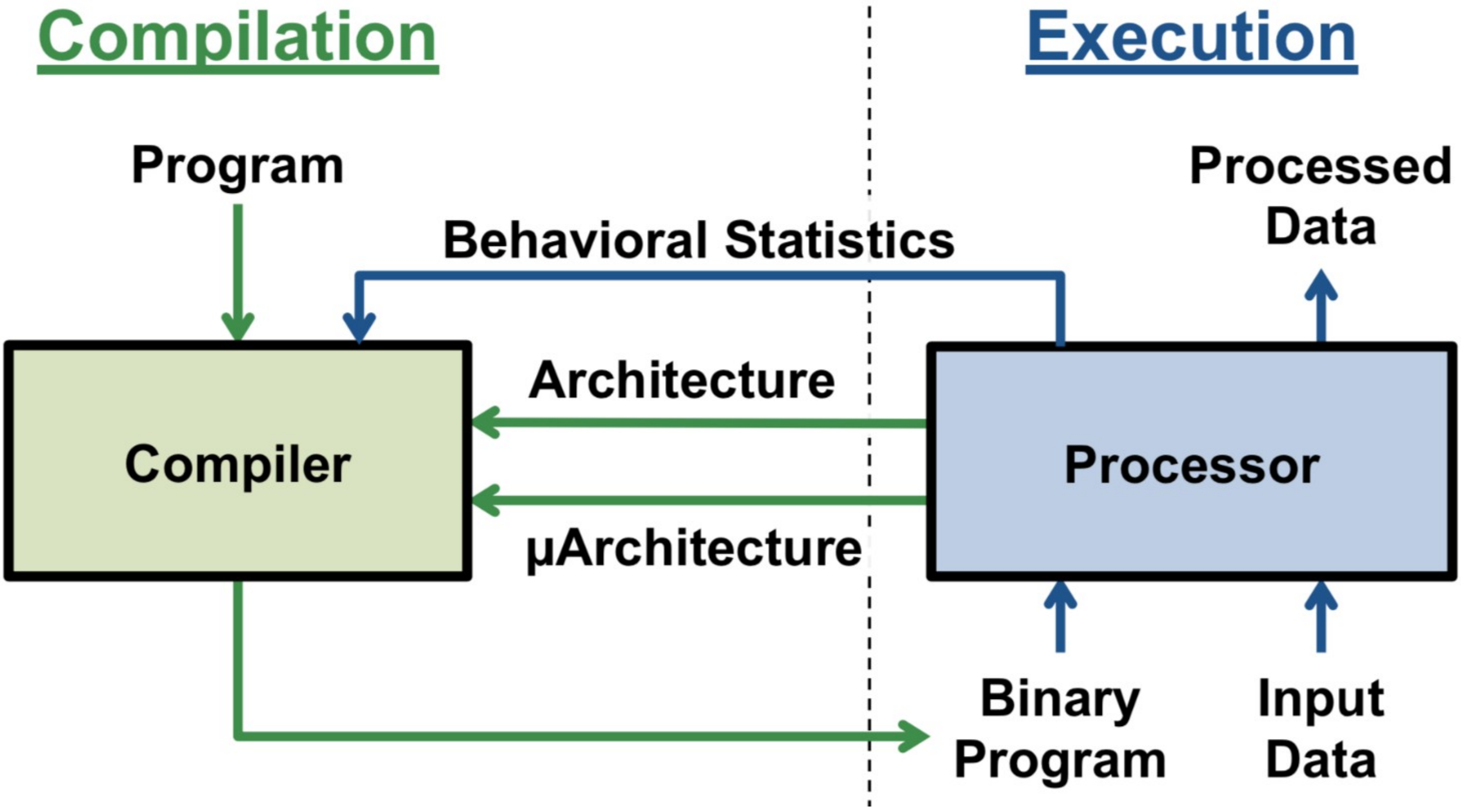
Hardware Resources

Optimization Compiler (Mapper)

Row Stationary Mapping

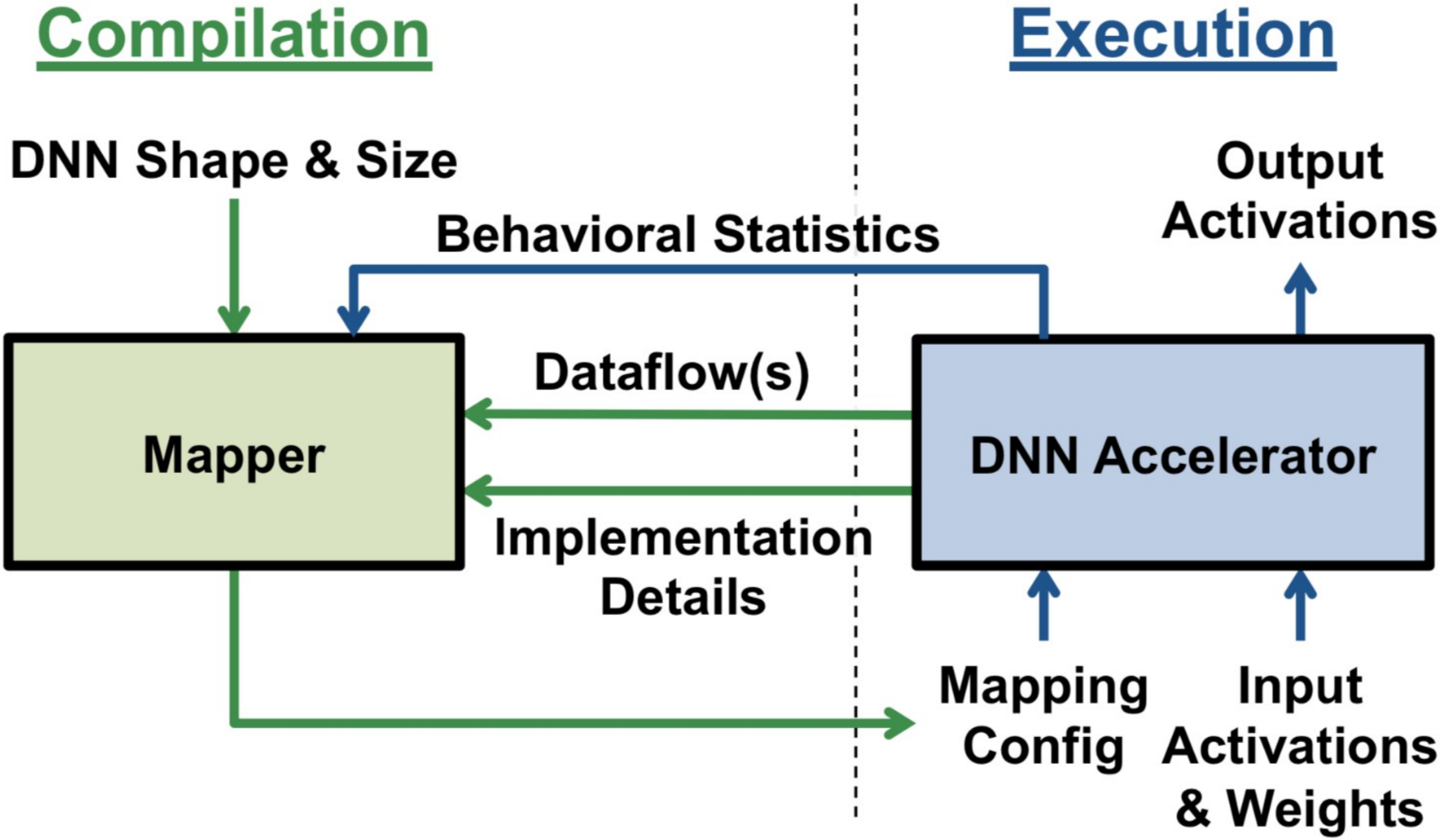
[Chen *et al.*, ISCA'16] [V. Sze *et al.*, Tutorial, ISCA'19]

Computer Architecture Analogy



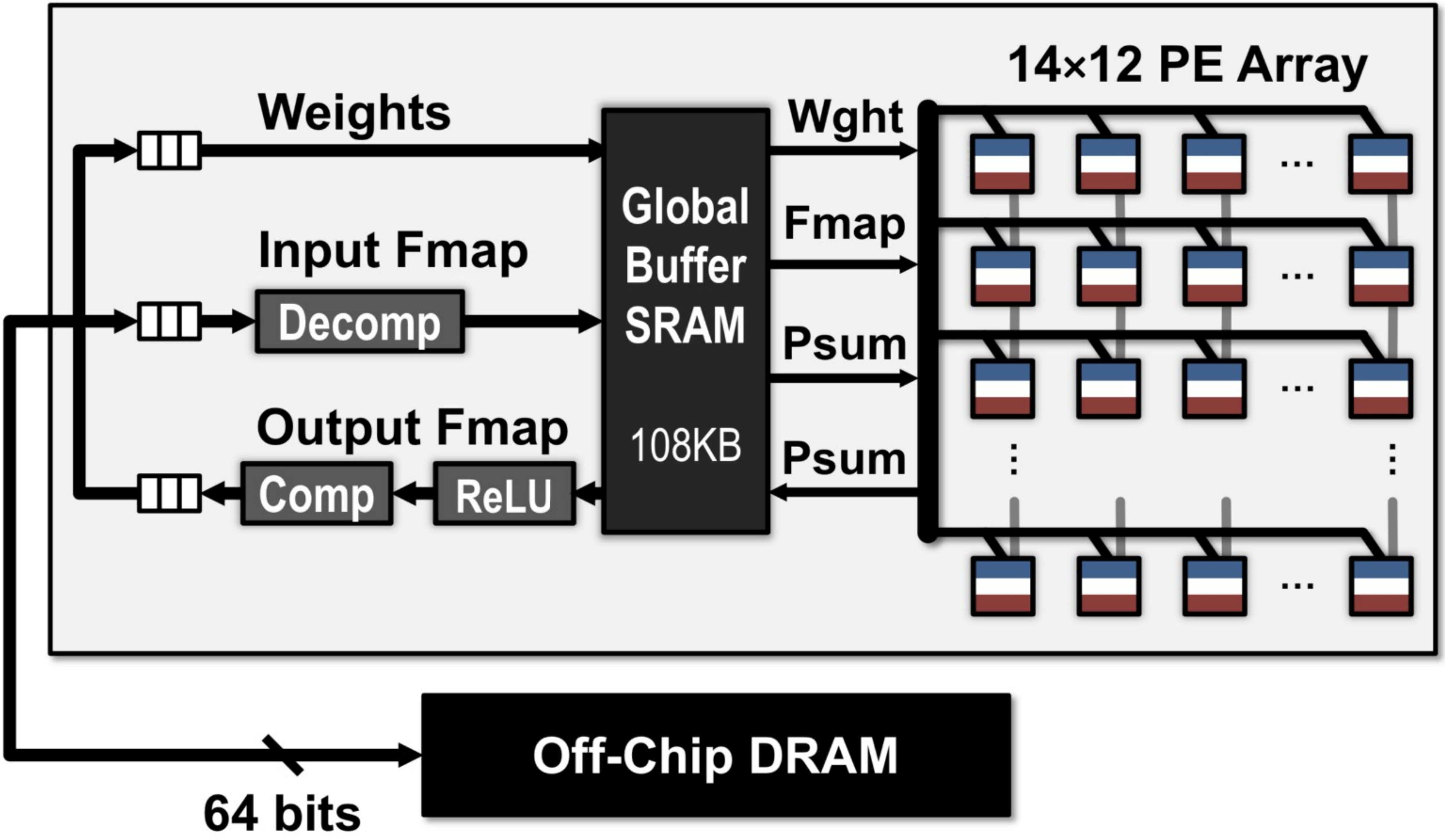
[Chen *et al.*, Micro Top-Picks 2017]

Computer Architecture Analogy



[Chen *et al.*, Micro Top-Picks 2017]

Eyeriss



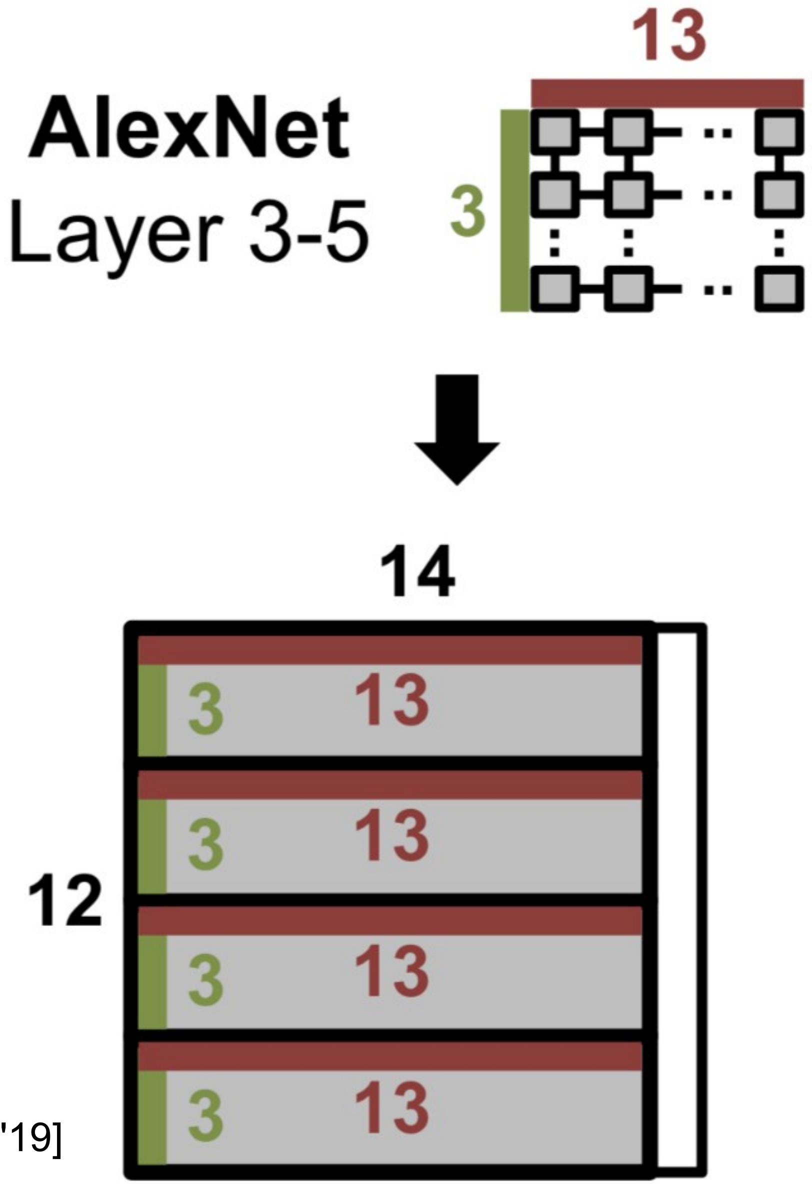
[Chen *et al.*, ISSCC 2016]

Problem 1

- ***How can the fixed-size PE array accomodate different layer shapes?***
- Solution
  - Replication
    - Filter height and ofmap height < rows and cols of PE array
  - Folding
    - Filter height and/or ofmap height > rows and cols of PE array

Replication – Example

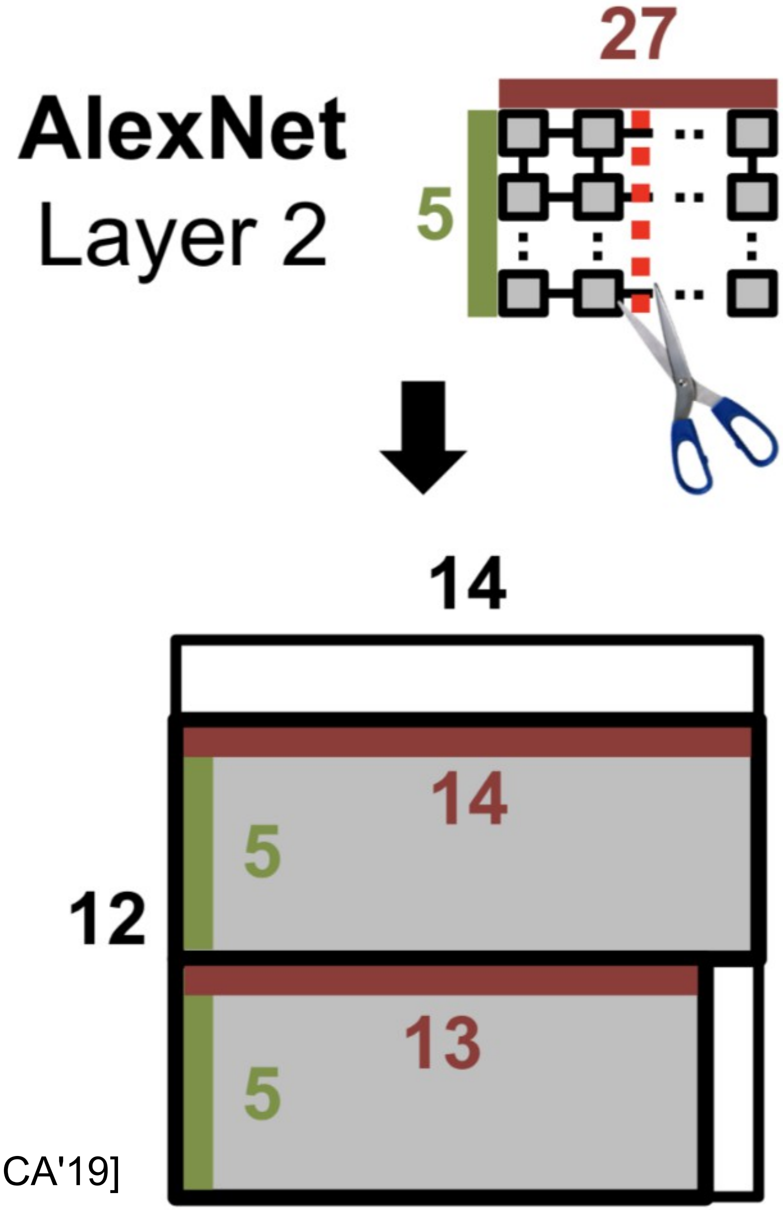
- AlexNet Layers 3 and 4 only use 13x3 PE array
  - The structure can be replicated 3 times
  - Running different channels and/or filters in each replication
  - Unused PEs are clock gated



[V. Sze *et al.*, Tutorial, ISCA'19]

Folding – Example

- AlexNet layer 2 requres 27x5 PE array
  - Can be fold into 14x5 and 13x5



[V. Sze *et al.*, Tutorial, ISCA'19]

Problem 2

- Data are passed in very specific pattern
  - Change with different shape configurations
- ***How can the fixed design pass data in different patterns?***
- Solution
  - **Using a custom multicast network**

Summary

- Keyword for improving efficiency
  - Minimizing data movement
  - Exploiting data reuse
- Dataflow taxonomy
  - **Output Stationary**: minimizing movement of **psums**
  - **Weight Stationary**: minimizing movement of **filters/weights**
  - **Input Stationary**: minimizing movement of **ifmaps**