

Accelerators-II

Tushar Krishna

Associate Professor @ Georgia Tech

Visiting Professor @ MIT EECS and CSAIL

Slide Acknowledgments: Michael Pellauer, Angshuman Parashar, Joel Emer, Vivienne Sze, Hyoukjun Kwon, Felix Kao

Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows

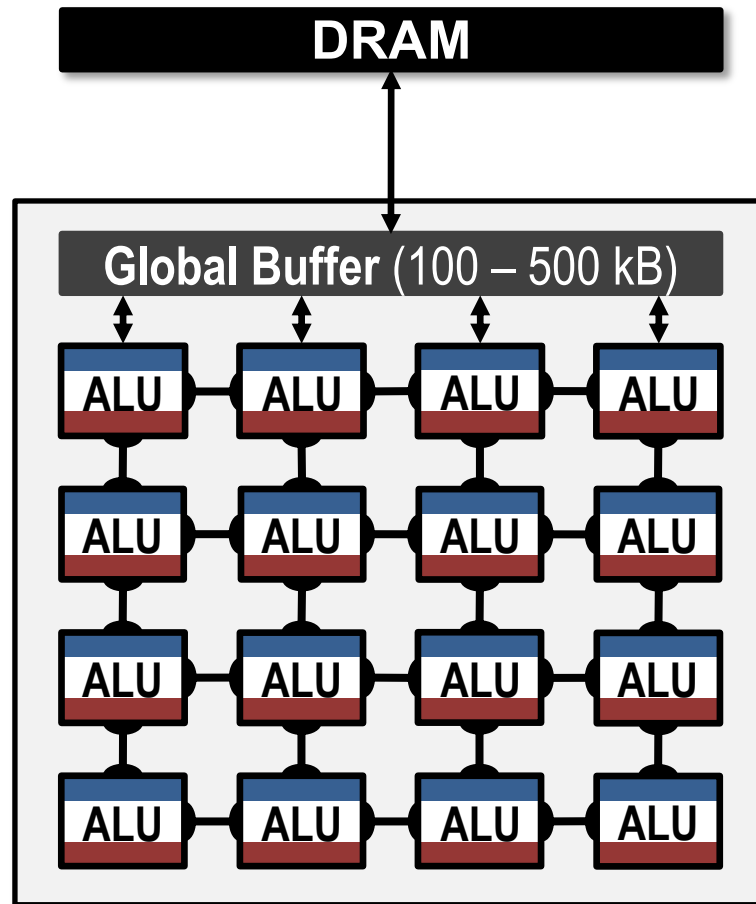
Outline

- **Recap**
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows

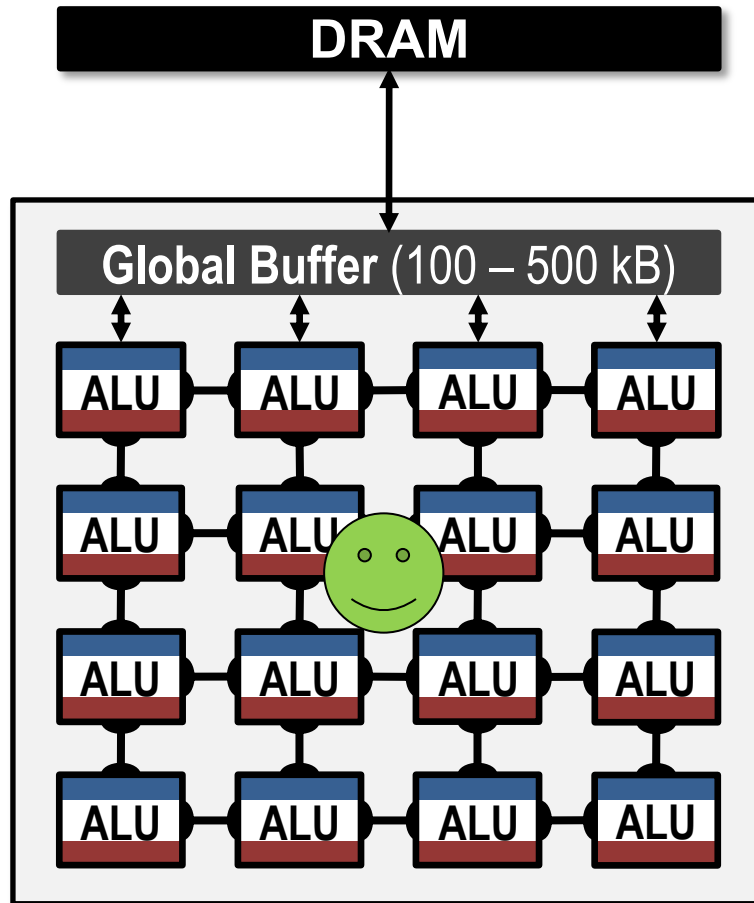
Recap

- Why domain-specific accelerators?
 - High Throughput requirements (workload constraint)
 - Energy costs of Data Movement (technology constraint)
- Why do accelerators help?
 - custom datapaths for the operator(s) of interest (e.g., matrix multiplication)
 - remove control overheads that Turing-complete engines (e.g., CPUs) have such as instruction fetch/decode, speculation, caches, ..

Accelerators



Accelerators



Custom
Datapath

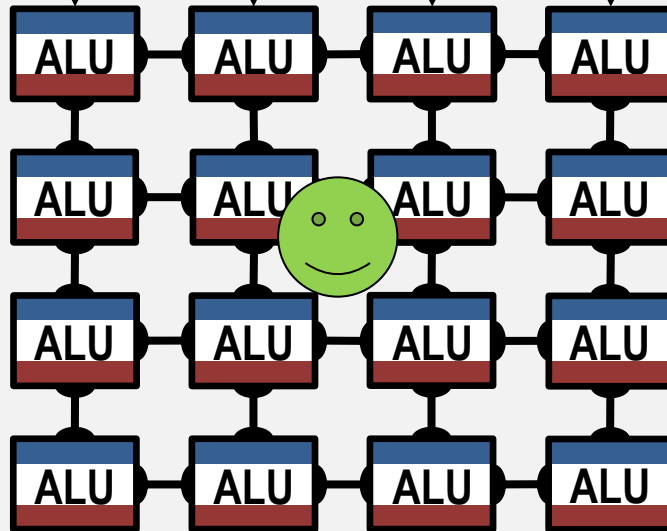
Accelerators

Off-Chip Memory

DRAM



Global Buffer (100 – 500 kB)

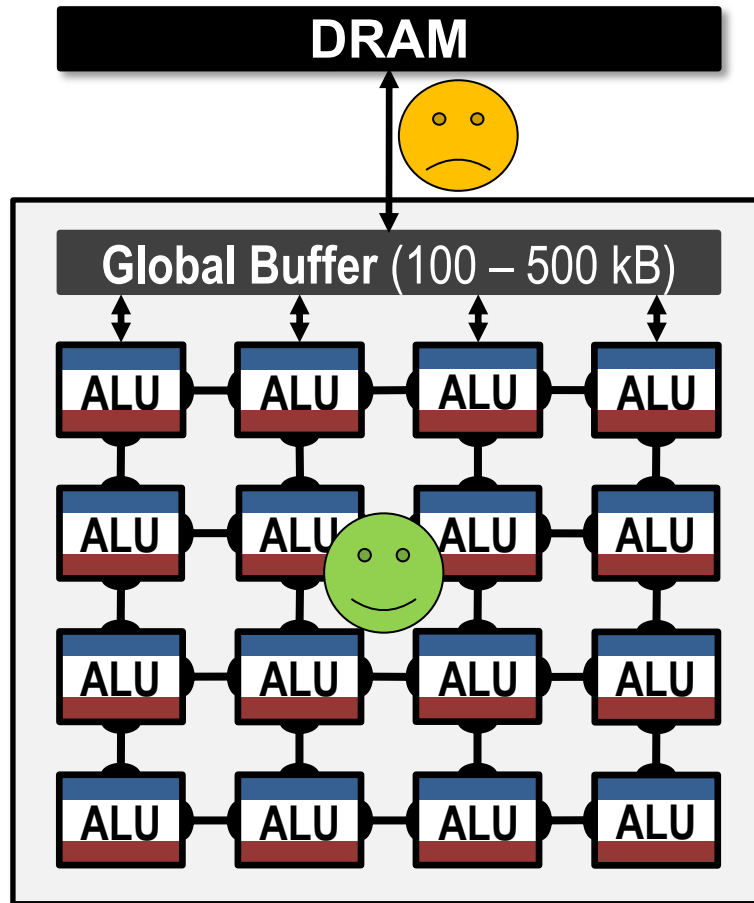


Custom Datapath

Accelerators

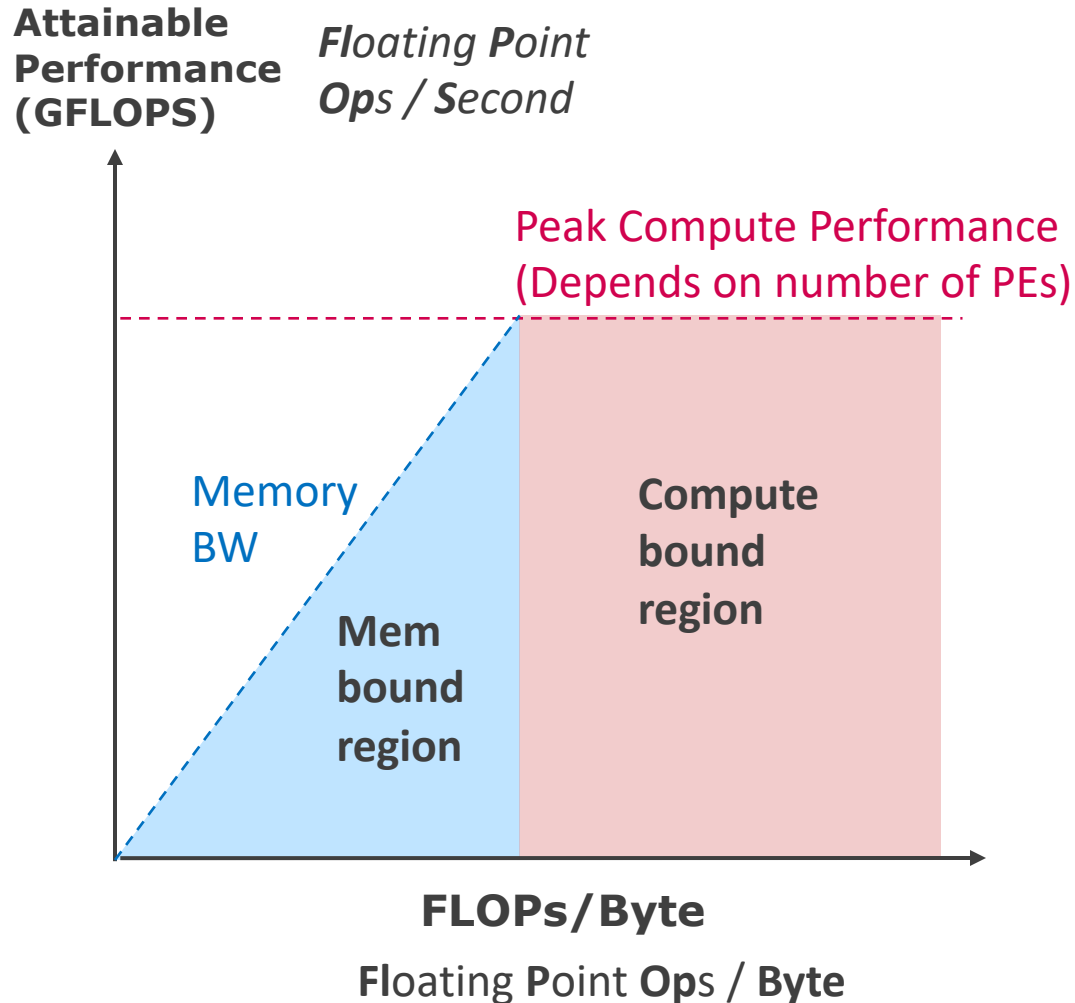
Off-Chip
Memory

Custom
Datapath

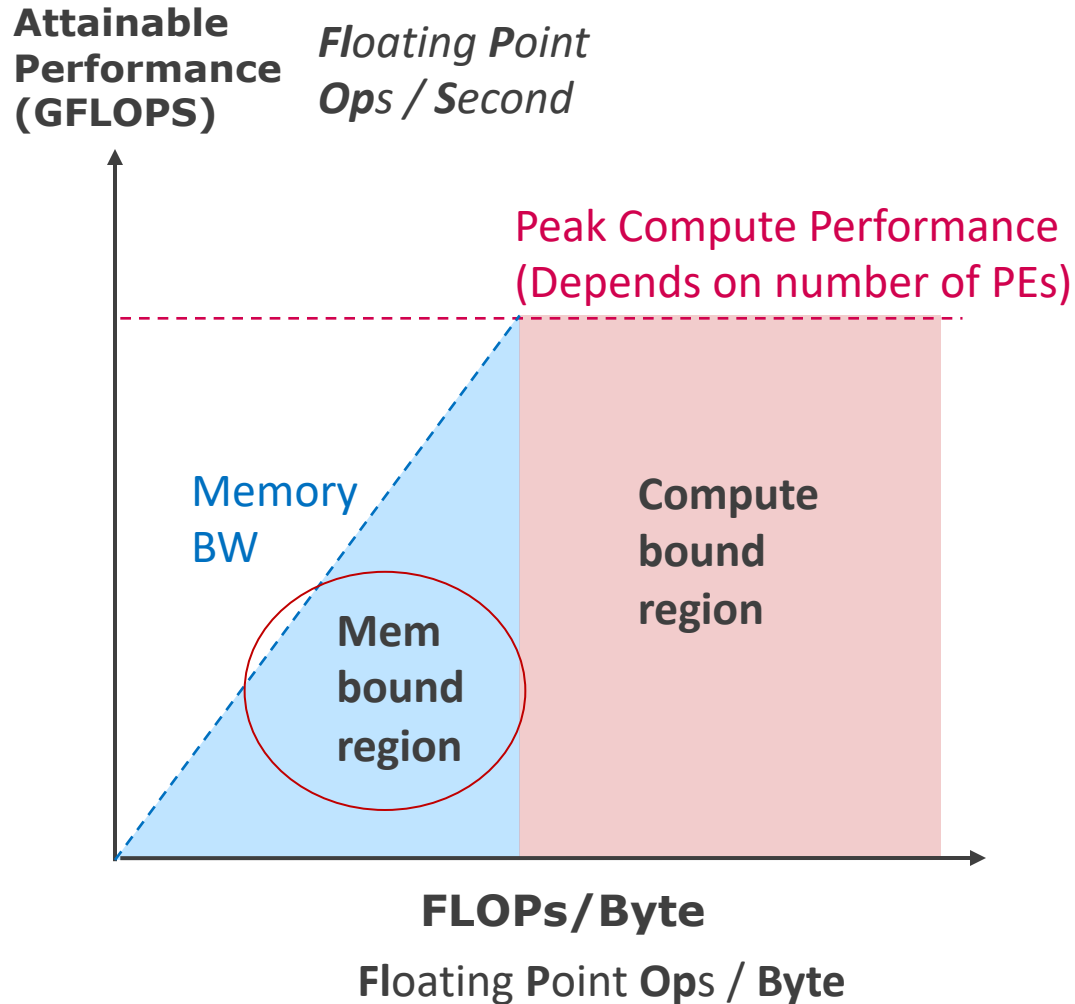


Why does this matter?

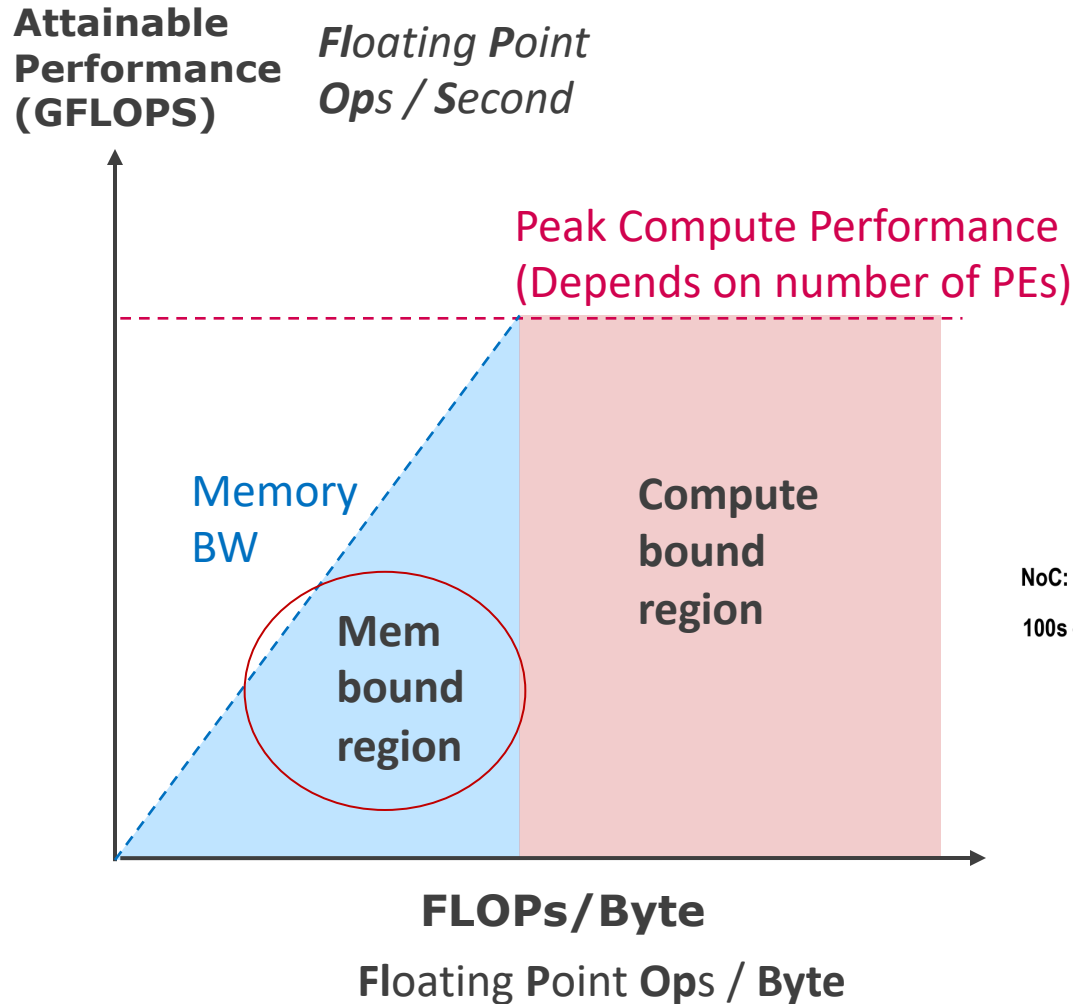
Why does this matter?



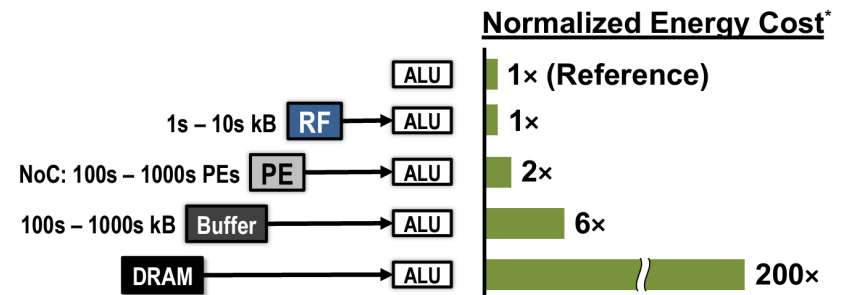
Why does this matter?



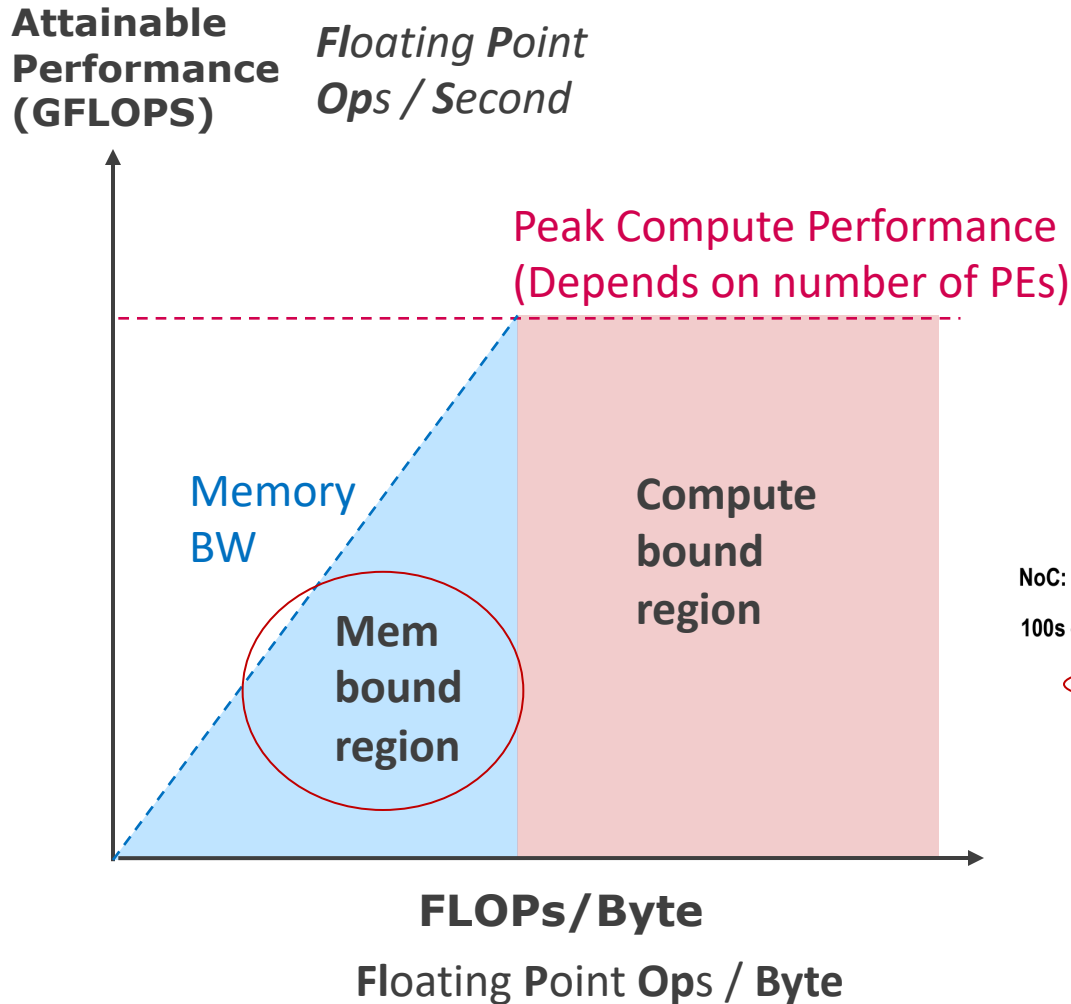
Why does this matter?



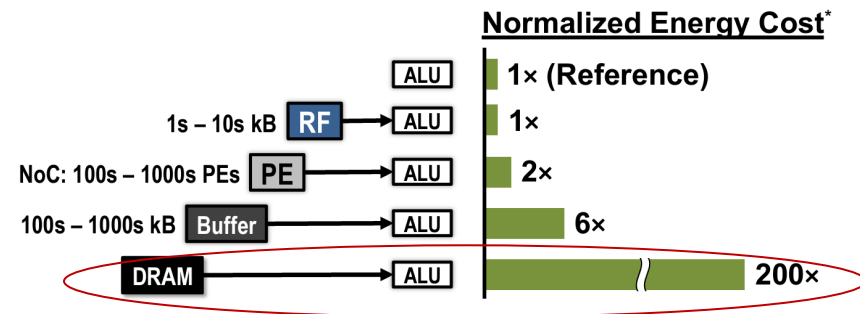
Energy Overheads



Why does this matter?



Energy Overheads



How to reduce BW requirement?

VGG16 conv 3_2	
Multiply Add Ops	1.85 Billion
Weights	590 K
Inputs	803 K
Outputs	803 K

How to reduce BW requirement?

VGG16 conv 3_2	
Multiply Add Ops	1.85 Billion
Weights	590 K
Inputs	803 K
Outputs	803 K



Data
Reuse

How to reduce BW requirement?

VGG16 conv 3_2	
Multiply Add Ops	1.85 Billion
Weights	590 K
Inputs	803 K
Outputs	803 K

Data
Reuse

How to exploit reuse?

How to reduce BW requirement?

VGG16 conv 3_2	
Multiply Add Ops	1.85 Billion
Weights	590 K
Inputs	803 K
Outputs	803 K

Data
Reuse

How to exploit reuse? "Dataflow"

How to reduce BW requirement?

VGG16 conv 3_2	
Multiply Add Ops	1.85 Billion
Weights	590 K
Inputs	803 K
Outputs	803 K

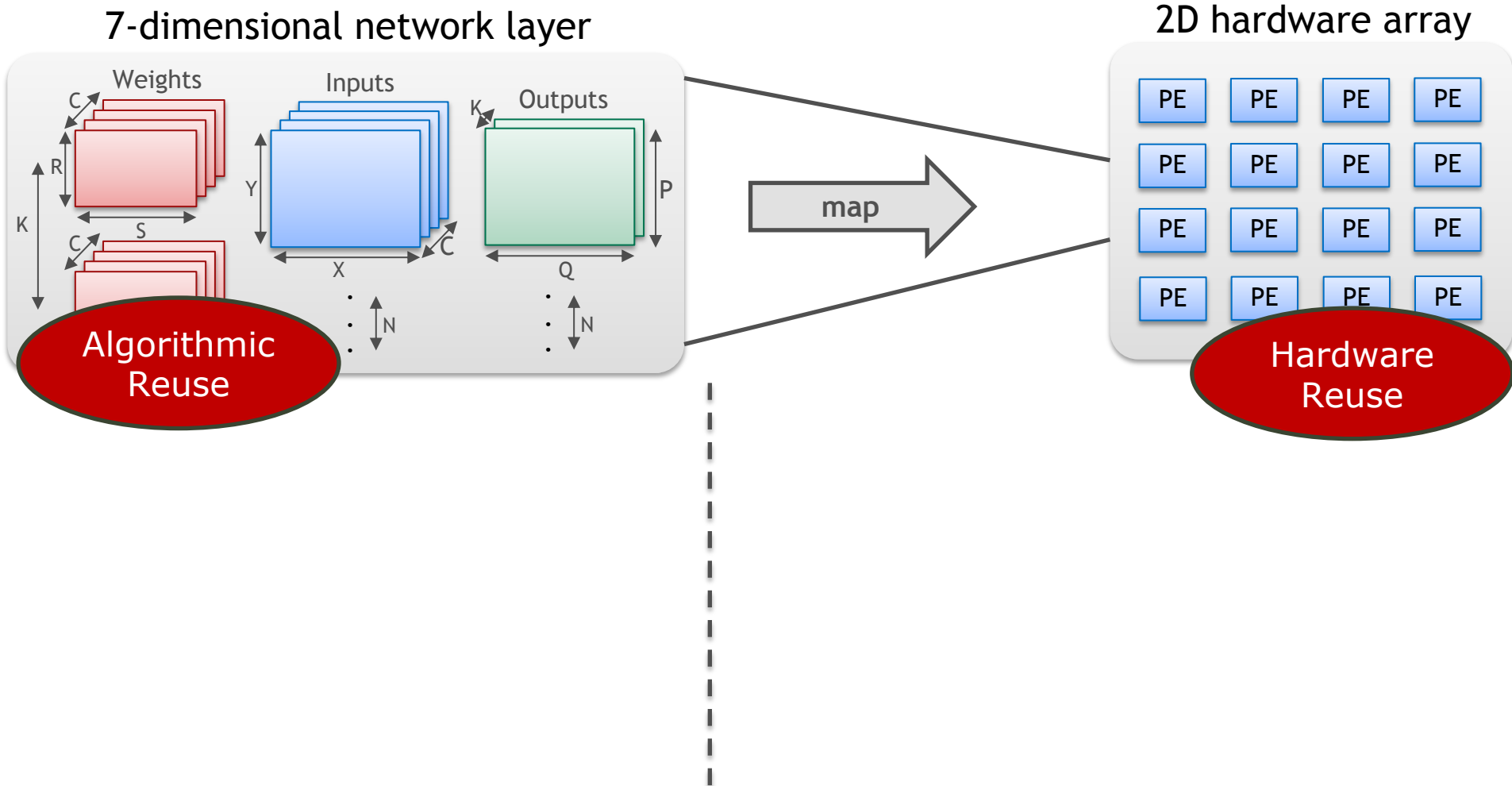
Data
Reuse

How to exploit reuse? “Dataflow”

i.e., fine-grained schedule of computations within DNN accelerators

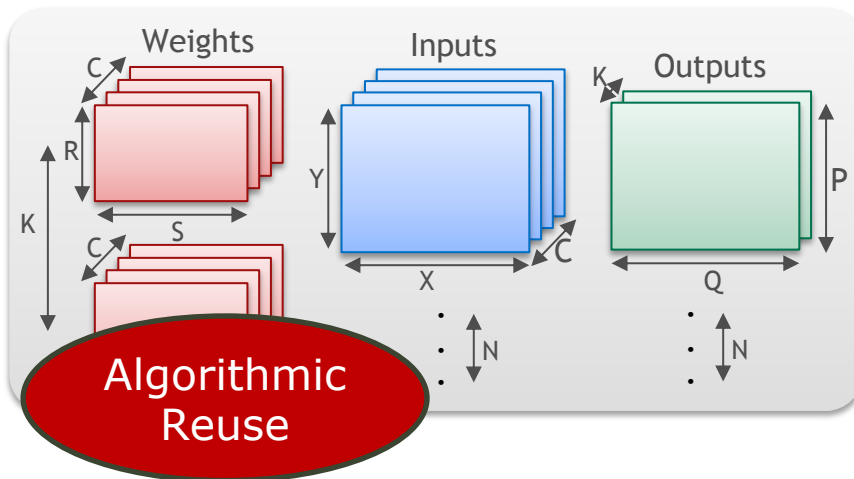
- Computation Order
- Parallelization Strategy

Dataflow Implication: Algorithm Reuse \rightarrow HW Reuse

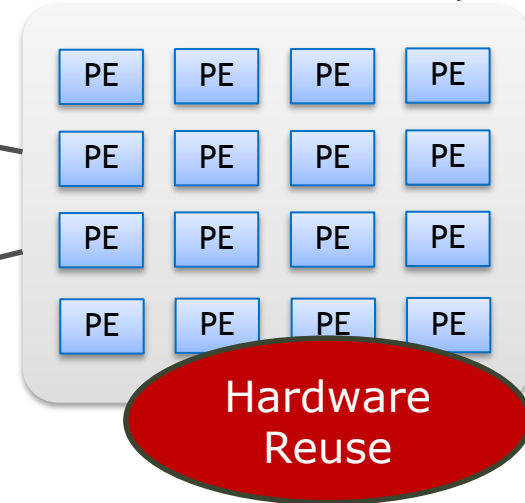


Dataflow Implication: Algorithm Reuse \rightarrow HW Reuse

7-dimensional network layer



2D hardware array



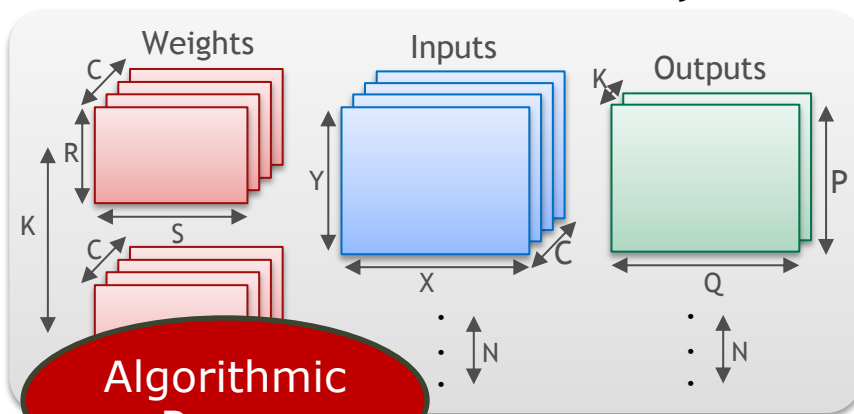
map \rightarrow

- **7D Computation Space**
 - $R * S * X * Y * C * K * N$
- **4D Operand/Result Data Spaces -**
 - Weights - $R * S * C * K$
 - Inputs - $X * Y * C * N$
 - Outputs - $P * Q * K * N$



Dataflow Implication: Algorithm Reuse \rightarrow HW Reuse

7-dimensional network layer



Algorithmic Reuse

- **7D Computation Space**
 - $R * S * X * Y * C * K * N$
- **4D Operand/Result Data Spaces -**
 - Weights - $R * S * C * K$
 - Inputs - $X * Y * C * N$
 - Outputs - $P * Q * K * N$



2D hardware array

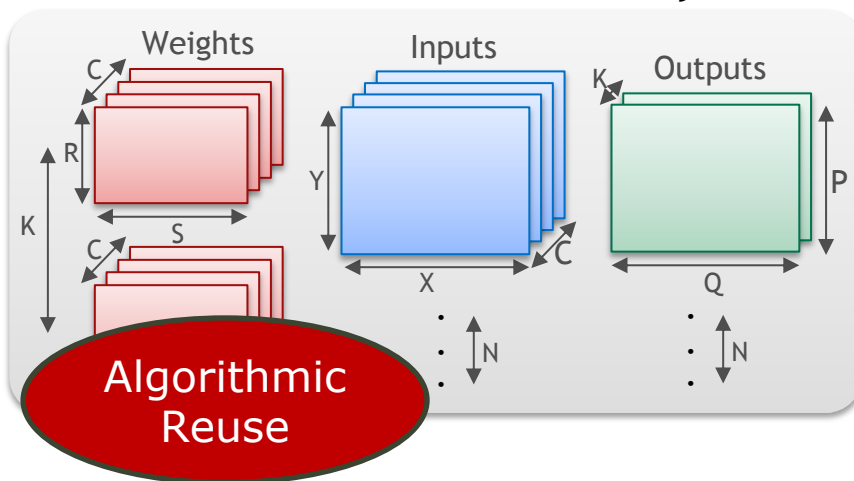


Hardware Reuse

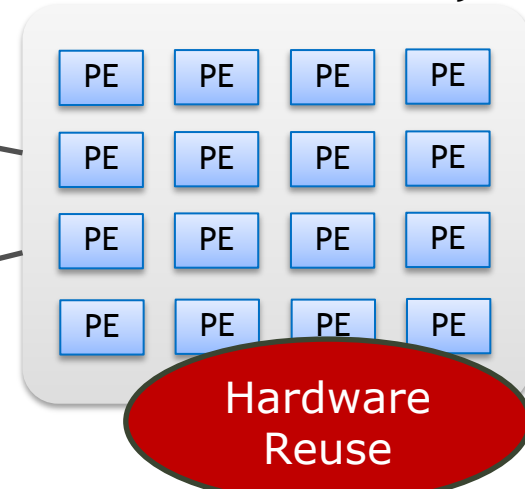
- **HW Design-space**
 - Number of PEs
 - Memory Hierarchy (Sizes and Bandwidths)
 - Interconnect Bandwidth

Dataflow Implication: Algorithm Reuse \rightarrow HW Reuse

7-dimensional network layer



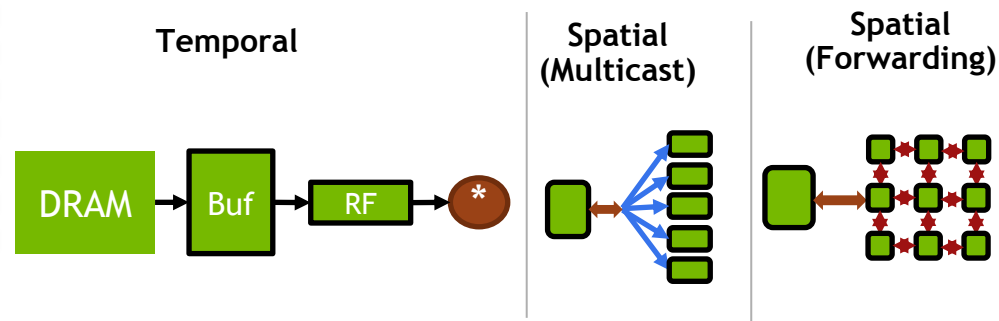
2D hardware array



- **7D Computation Space**
 - $R * S * X * Y * C * K * N$
- **4D Operand/Result Data Spaces -**
 - Weights - $R * S * C * K$
 - Inputs - $X * Y * C * N$
 - Outputs - $P * Q * K * N$

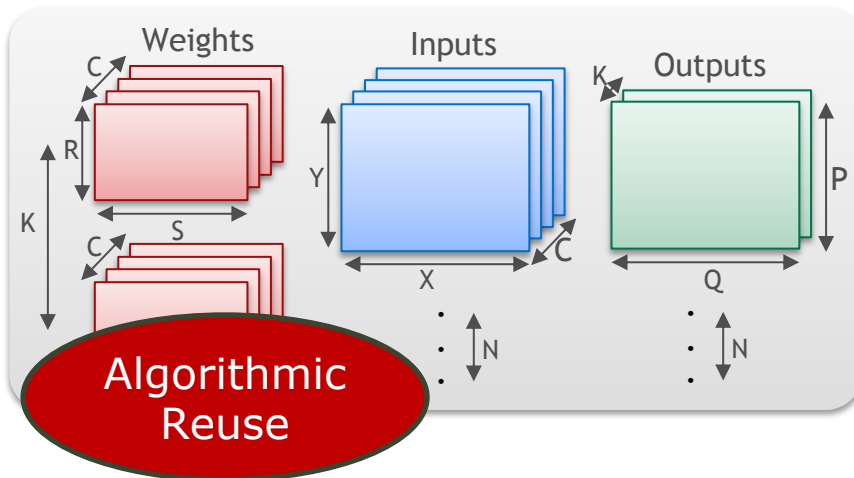
- **HW Design-space**
 - Number of PEs
 - Memory Hierarchy (Sizes and Bandwidths)
 - Interconnect Bandwidth

• **HW Reuse Structures**

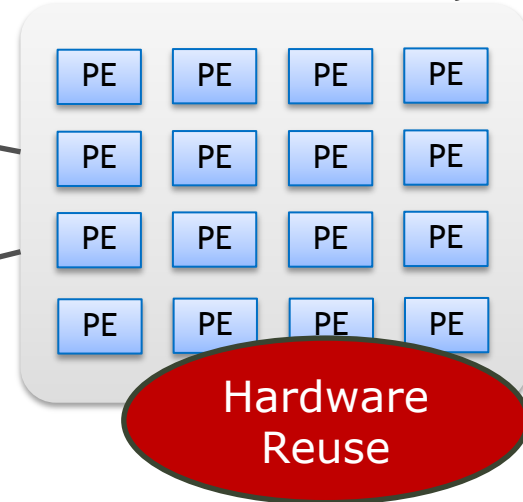


Dataflow Implication: Algorithm Reuse \rightarrow HW Reuse

7-dimensional network layer



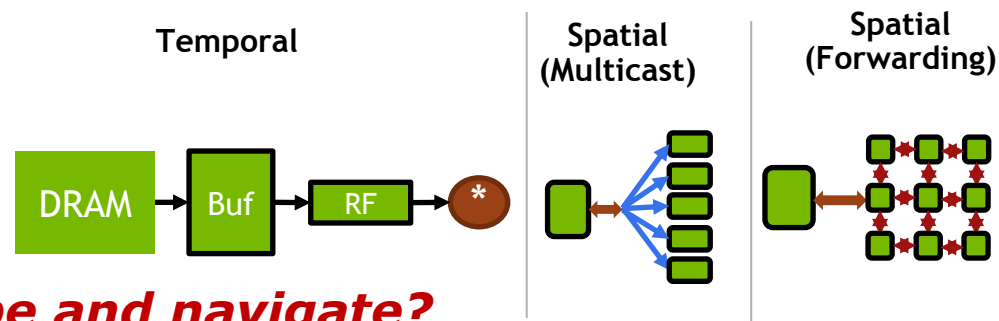
2D hardware array



- **7D Computation Space**
 - $R * S * X * Y * C * K * N$
- **4D Operand/Result Data Spaces -**
 - Weights - $R * S * C * K$
 - Inputs - $X * Y * C * N$
 - Outputs - $P * Q * K * N$

- **HW Design-space**
 - Number of PEs
 - Memory Hierarchy (Sizes and Bandwidths)
 - Interconnect Bandwidth

• **HW Reuse Structures**

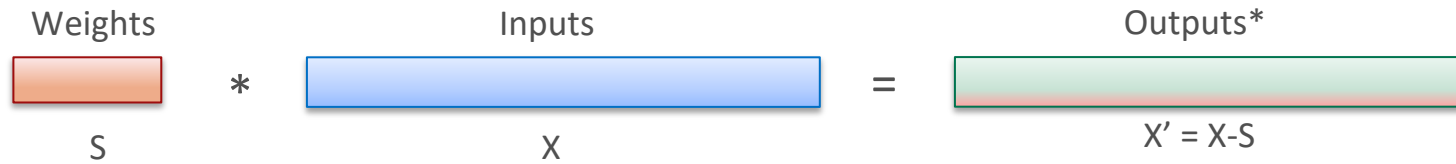


How to describe and navigate?

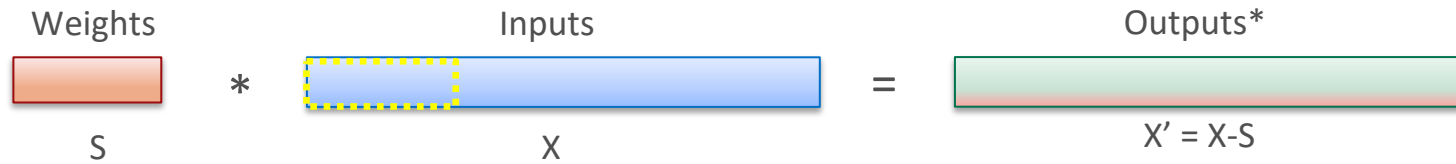
Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows

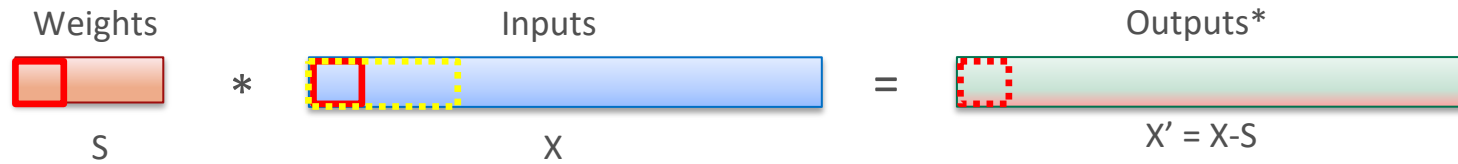
Output Stationary (OS) Dataflow



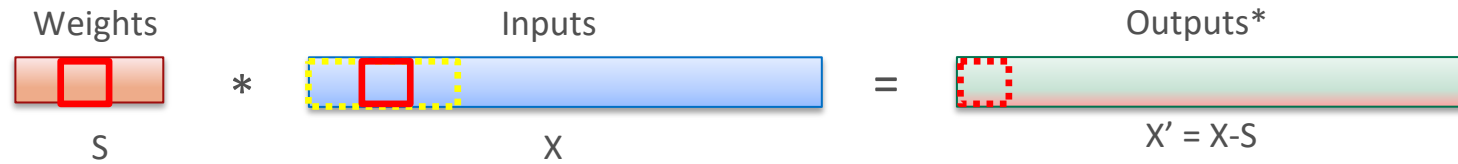
Output Stationary (OS) Dataflow



Output Stationary (OS) Dataflow



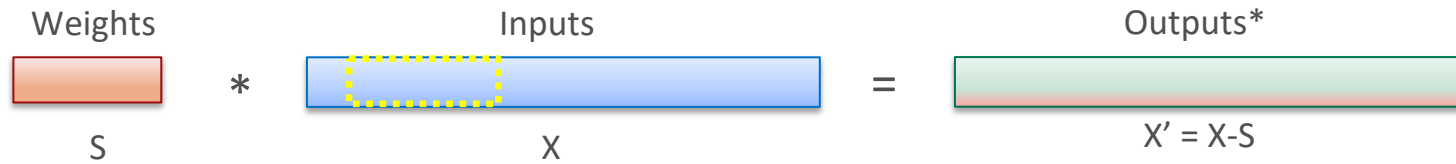
Output Stationary (OS) Dataflow



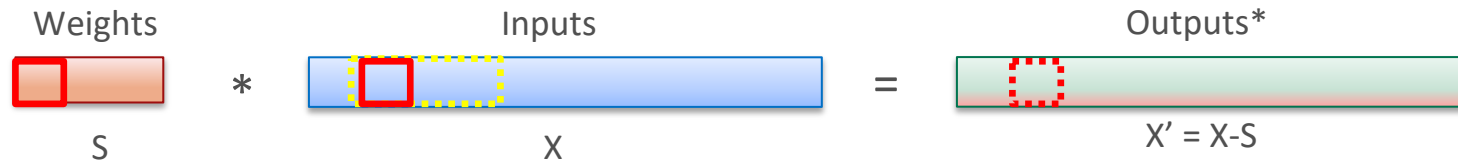
Output Stationary (OS) Dataflow



Output Stationary (OS) Dataflow



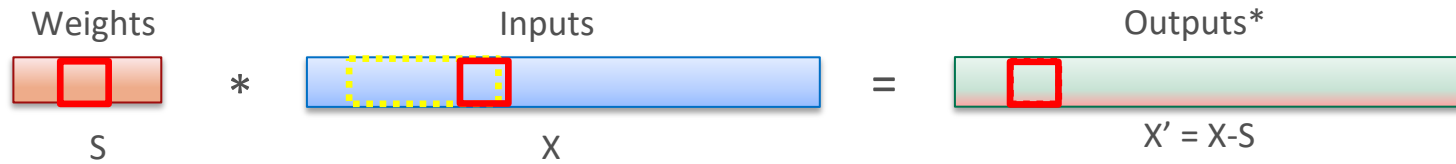
Output Stationary (OS) Dataflow



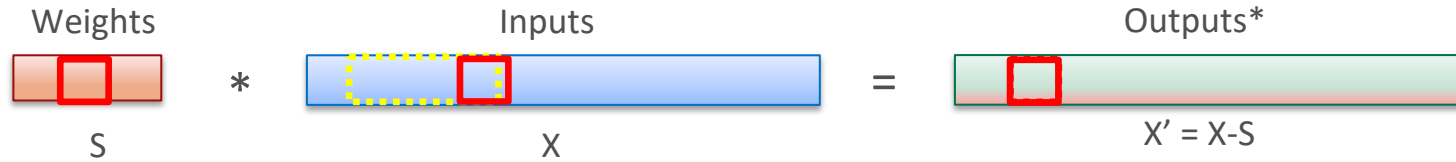
Output Stationary (OS) Dataflow



Output Stationary (OS) Dataflow



Output Stationary (OS) Dataflow



Computation

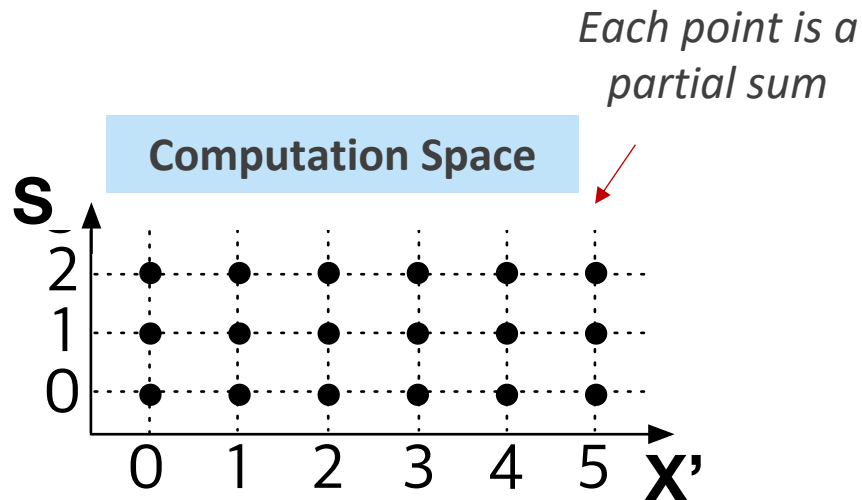
```
for(int x = 0; x < X'; x++)  
  for(int s = 0; s < S; s++)  
    Output[x] += Weight[s] * Input[x+s]
```

Output Stationary (OS) Dataflow

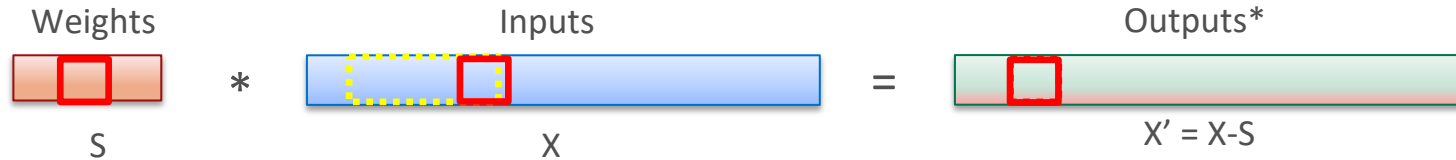


Computation

```
for(int x = 0; x < X'; x++)  
  for(int s = 0; s < S; s++)  
    Output[x] += Weight[s] * Input[x+s]
```

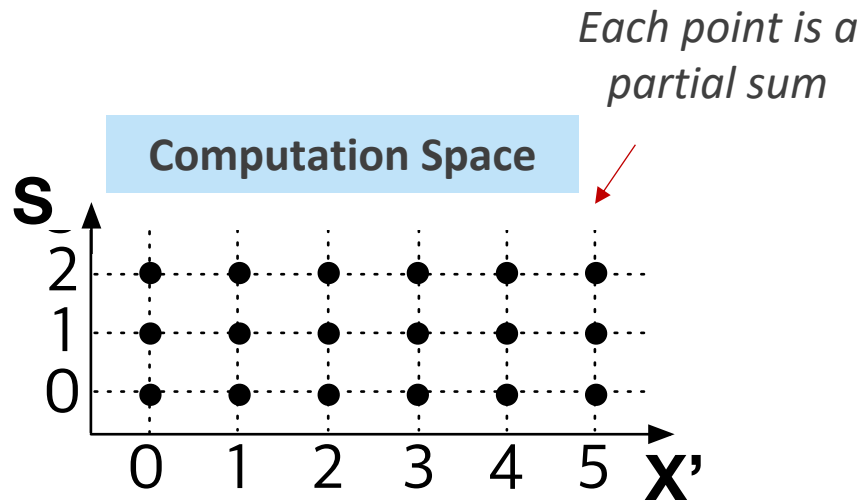
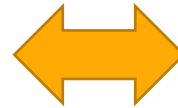


Output Stationary (OS) Dataflow



Computation

```
for(int x = 0; x < X'; x++)  
  for(int s = 0; s < S; s++)  
    Output[x] += Weight[s] * Input[x+s]
```

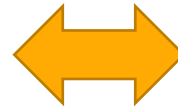


Output Stationary (OS) Dataflow



Computation

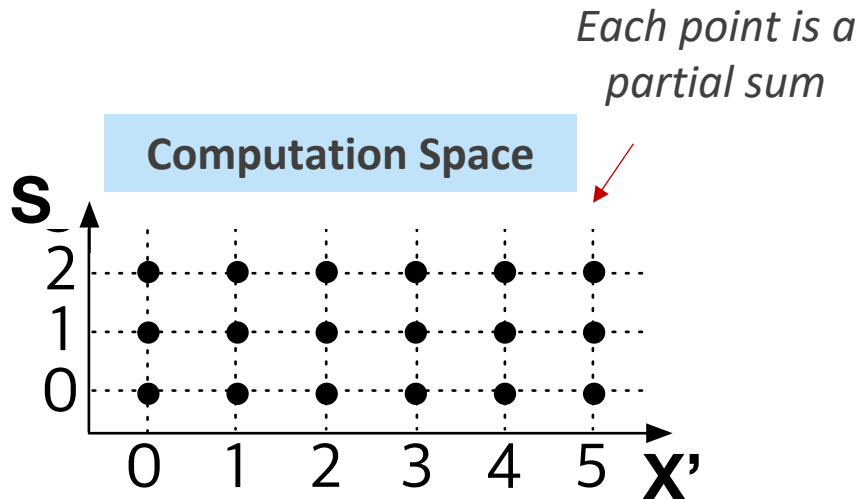
```
for(int x = 0; x < X'; x++)  
  for(int s = 0; s < S; s++)  
    Output[x] += Weight[s] * Input[x+s]
```



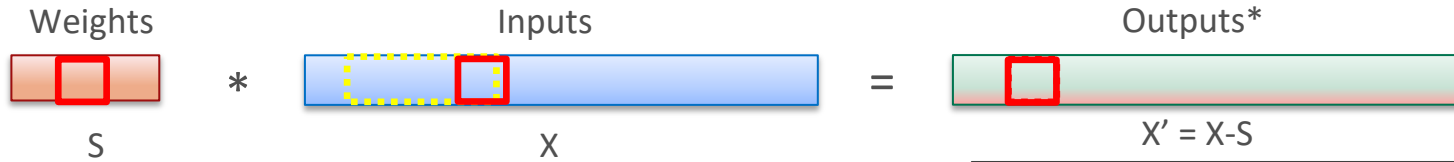
Data

PartialSum[X'][S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]



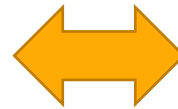
Output Stationary (OS) Dataflow



Computation

```

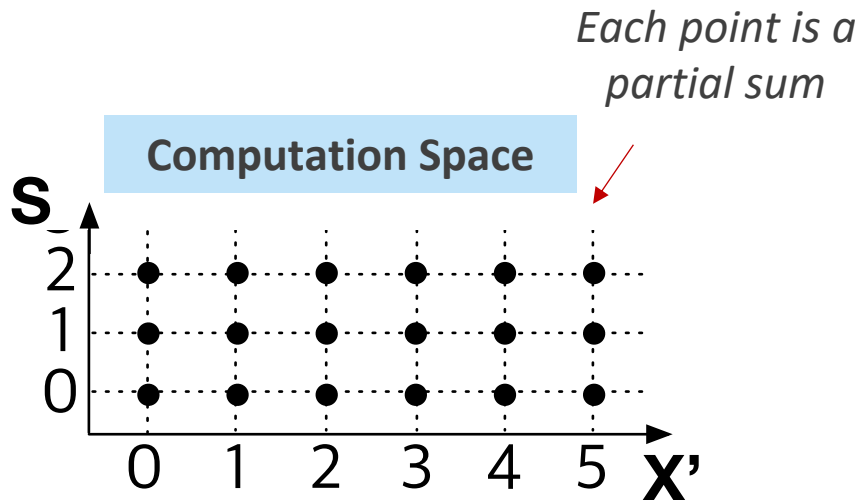
for(int x = 0; x < X'; x++)
  for(int s = 0; s < S; s++)
    Output[x] += Weight[s] * Input[x+s]
  
```



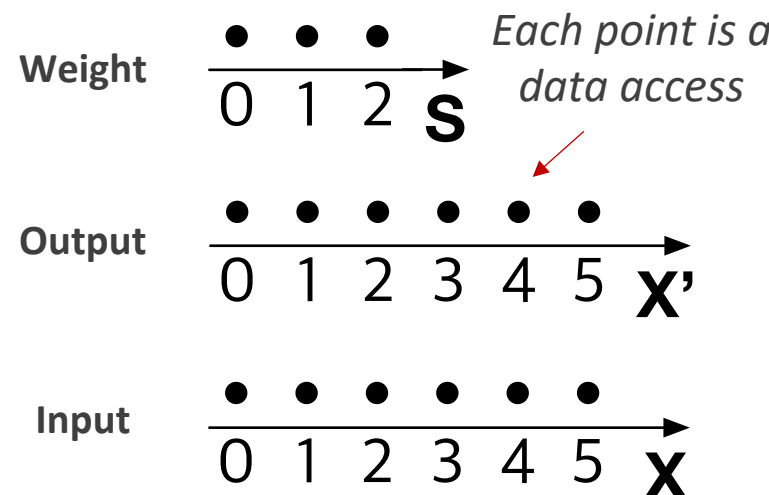
Data

PartialSum[X'][S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]



Data Space

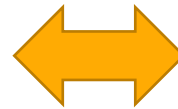


Output Stationary (OS) Dataflow



Computation

```
for(int x = 0; x < X'; x++)
  for(int s = 0; s < S; s++)
    Output[x] += Weight[s] * Input[x+s]
```

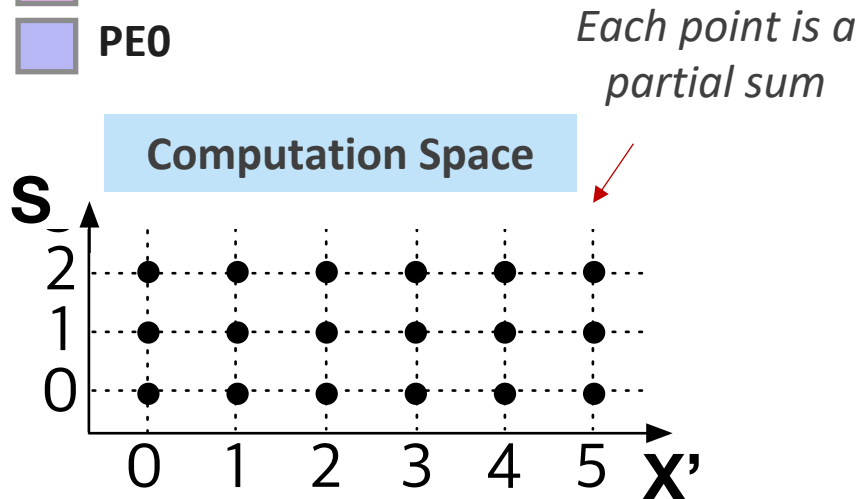


Data

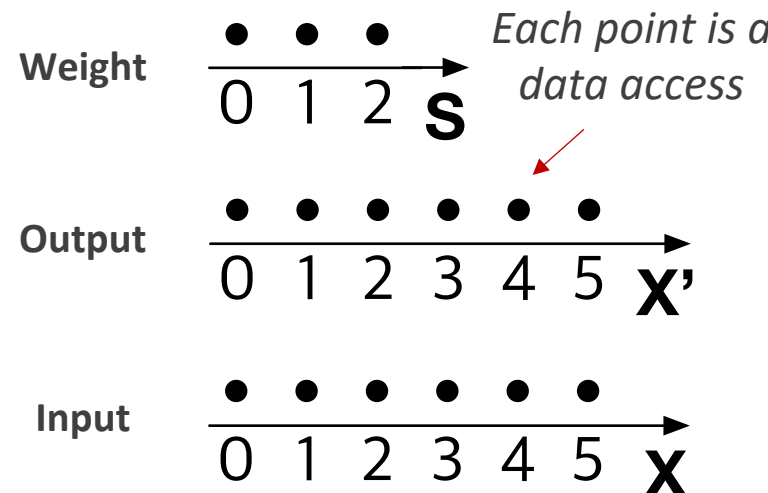
PartialSum[X'][S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

- PE2
- PE1
- PE0



Data Space



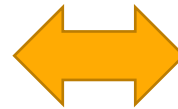
Output Stationary (OS) Dataflow



Computation

```

for(int x = 0; x < X'; x++)
  for(int s = 0; s < S; s++)
    Output[x] += Weight[s] * Input[x+s]
    
```

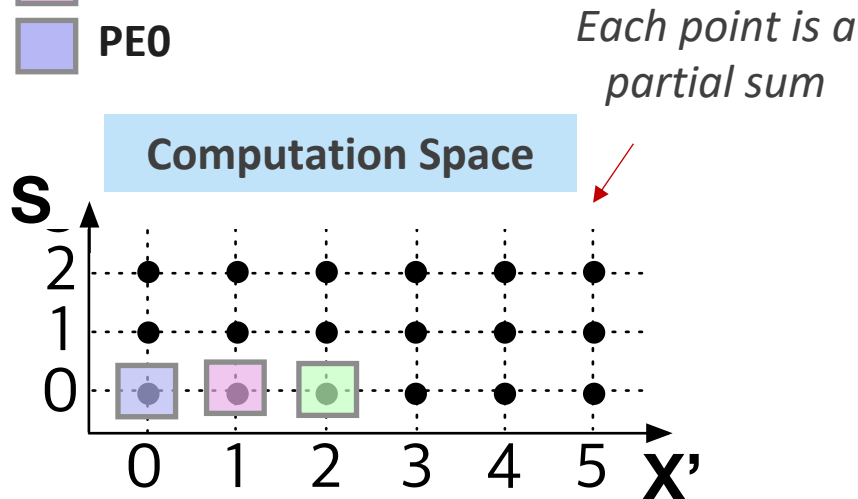


Data

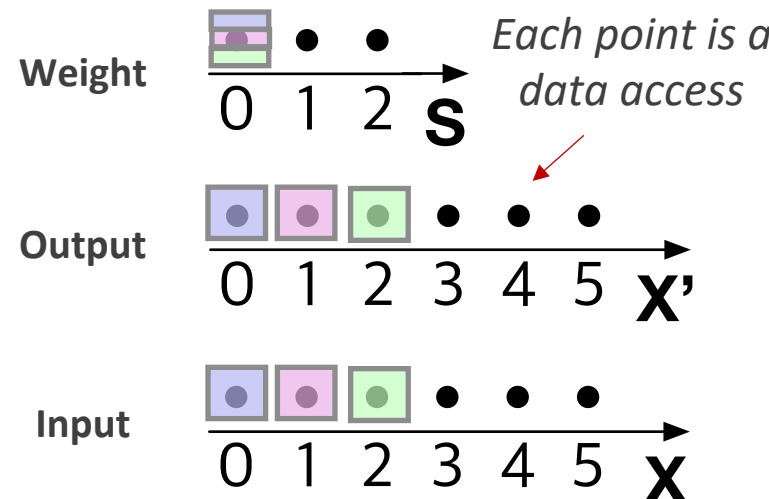
PartialSum[X'][S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

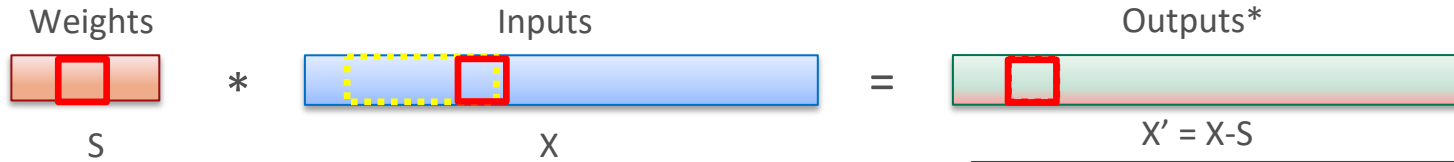
- PE2
- PE1
- PE0



Data Space



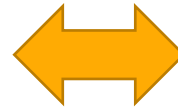
Output Stationary (OS) Dataflow



Computation

```

for(int x = 0; x < X'; x++)
  for(int s = 0; s < S; s++)
    Output[x] += Weight[s] * Input[x+s]
    
```



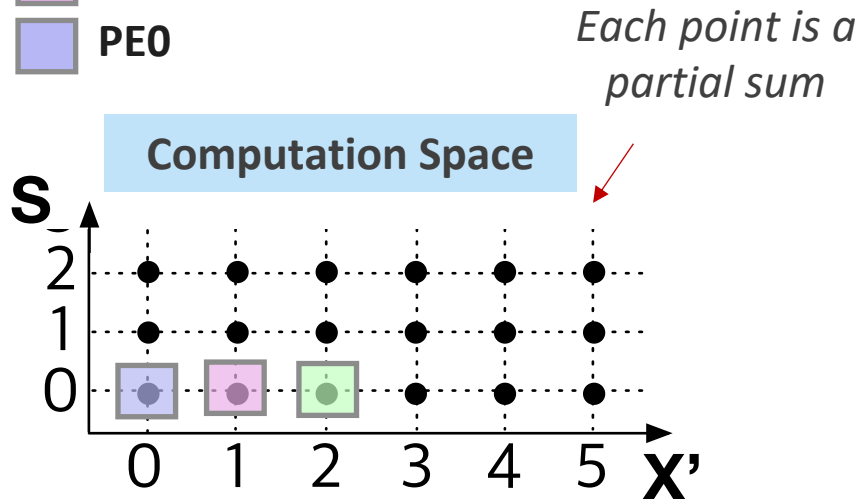
Data

PartialSum[X'][S] needs to access:

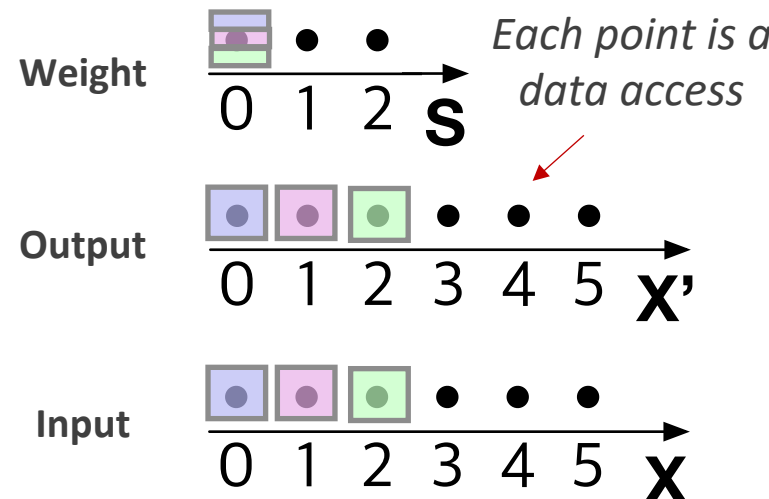
- Weight[s]
- Output[x']
- Input[x'+s]

- PE2
- PE1
- PE0

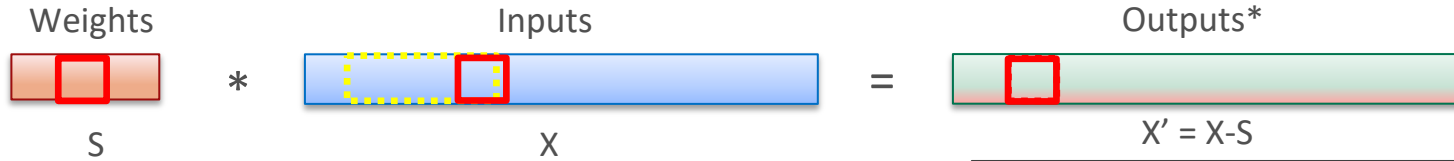
Time = 0



Data Space

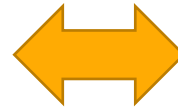


Output Stationary (OS) Dataflow



Computation

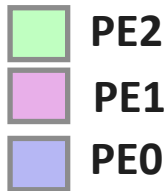
```
for(int x = 0; x < X'; x++)
  for(int s = 0; s < S; s++)
    Output[x] += Weight[s] * Input[x+s]
```



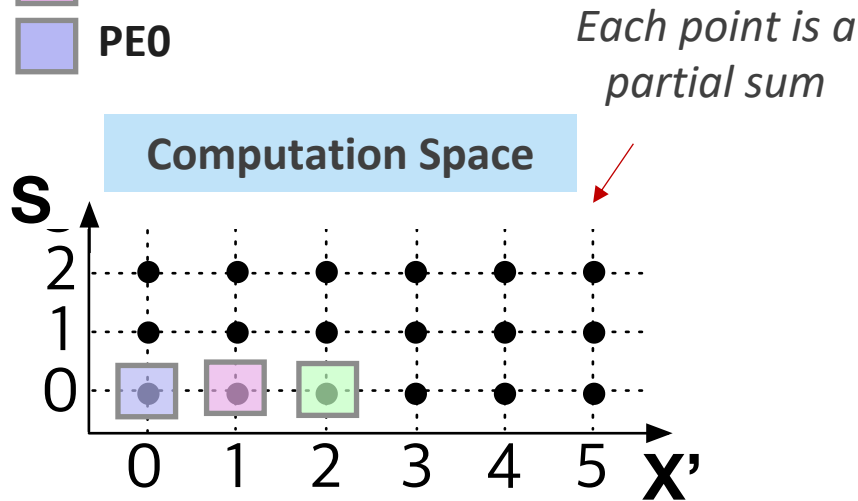
Data

PartialSum[X'][S] needs to access:

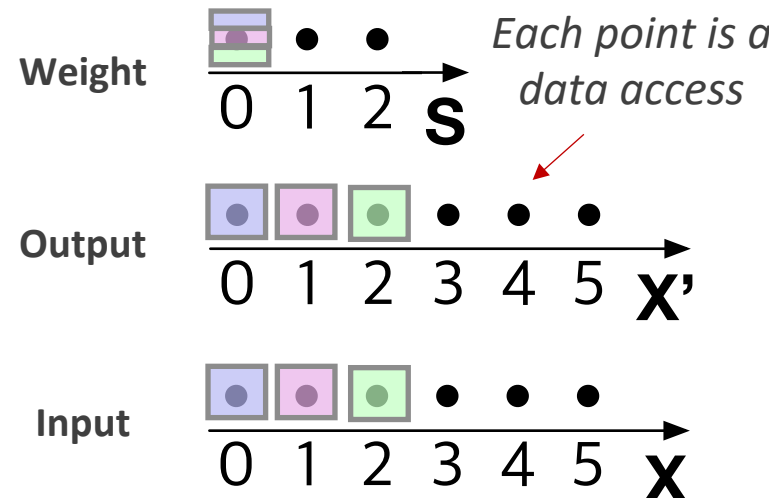
- Weight[s]
- Output[x']
- Input[x'+s]



Time = 1



Data Space

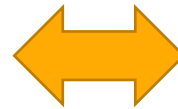


Output Stationary (OS) Dataflow



Computation

```
for(int x = 0; x < X'; x++)
  for(int s = 0; s < S; s++)
    Output[x] += Weight[s] * Input[x+s]
```



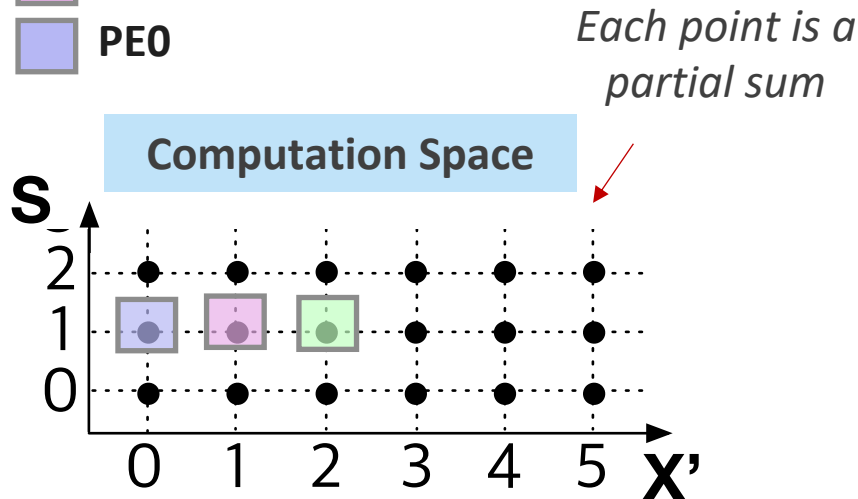
Data

PartialSum[X'][S] needs to access:

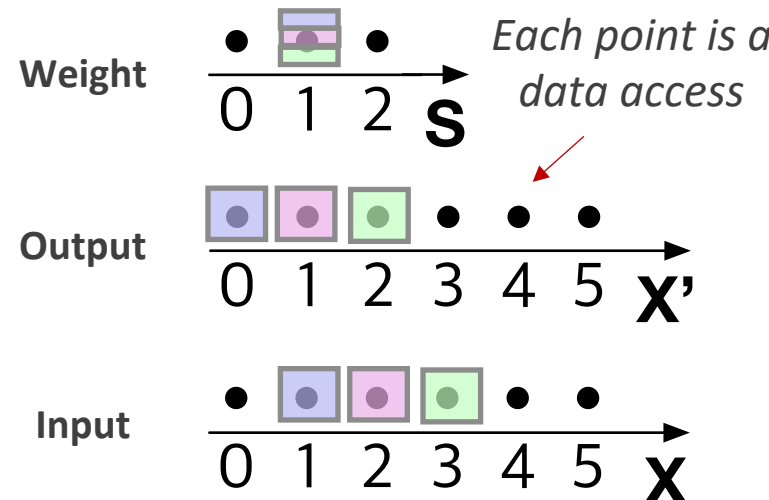
- Weight[s]
- Output[x']
- Input[x'+s]



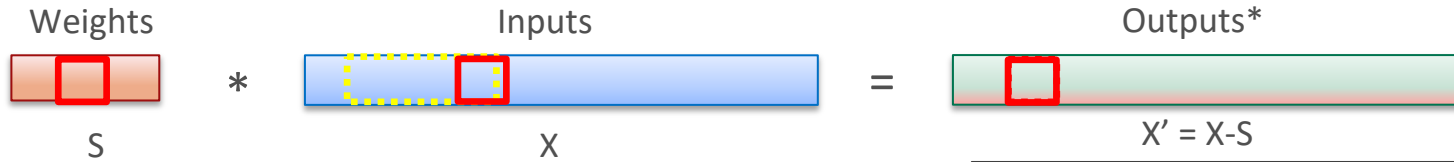
Time = 1



Data Space

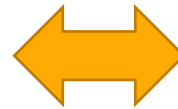


Output Stationary (OS) Dataflow



Computation

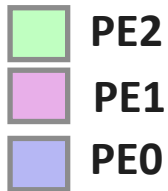
```
for(int x = 0; x < X'; x++)
  for(int s = 0; s < S; s++)
    Output[x] += Weight[s] * Input[x+s]
```



Data

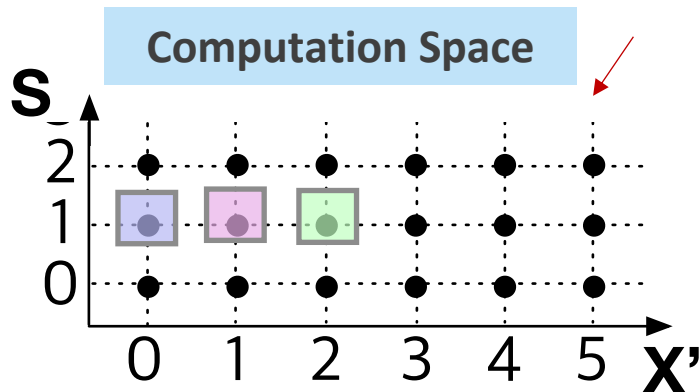
PartialSum[X'][S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

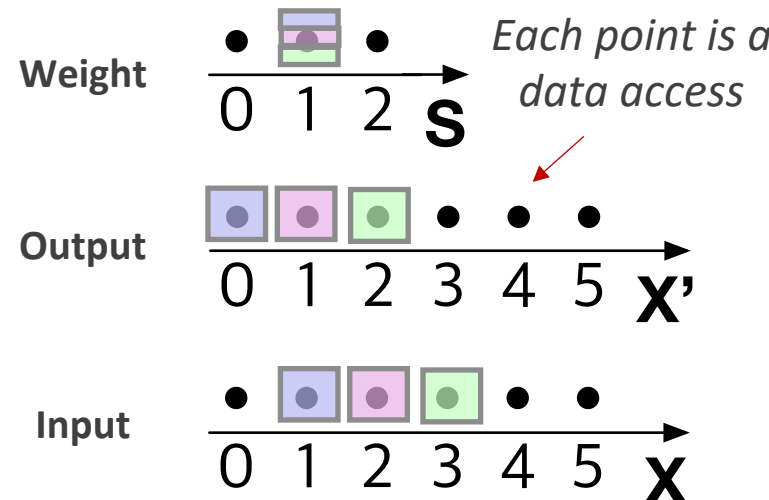


Time = 2

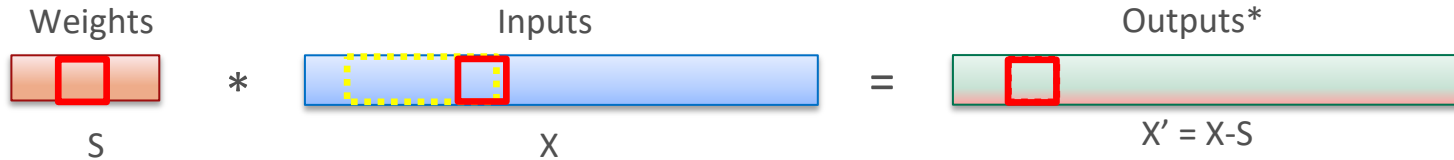
Each point is a partial sum



Data Space



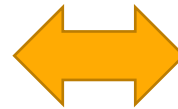
Output Stationary (OS) Dataflow



Computation

```

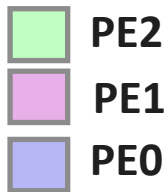
for(int x = 0; x < X'; x++)
  for(int s = 0; s < S; s++)
    Output[x] += Weight[s] * Input[x+s]
  
```



Data

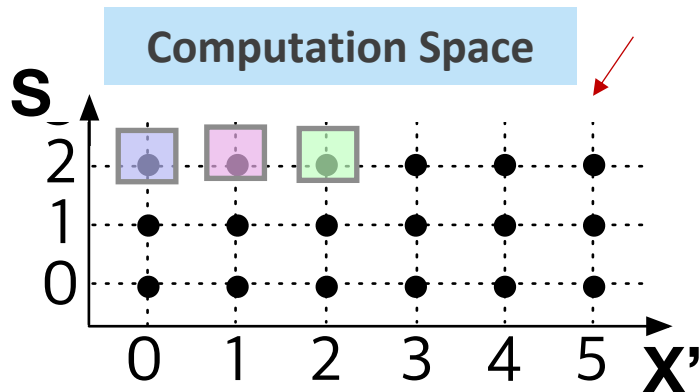
PartialSum[X'][S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

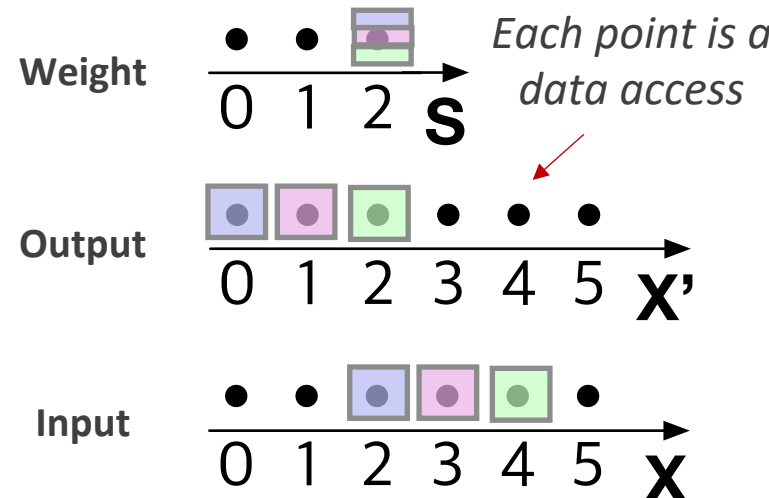


Time = 2

Each point is a partial sum



Data Space

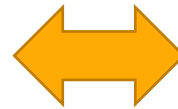


Output Stationary (OS) Dataflow



Computation

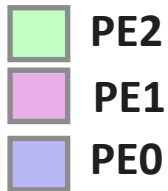
```
for(int x = 0; x < X'; x++)
  for(int s = 0; s < S; s++)
    Output[x] += Weight[s] * Input[x+s]
```



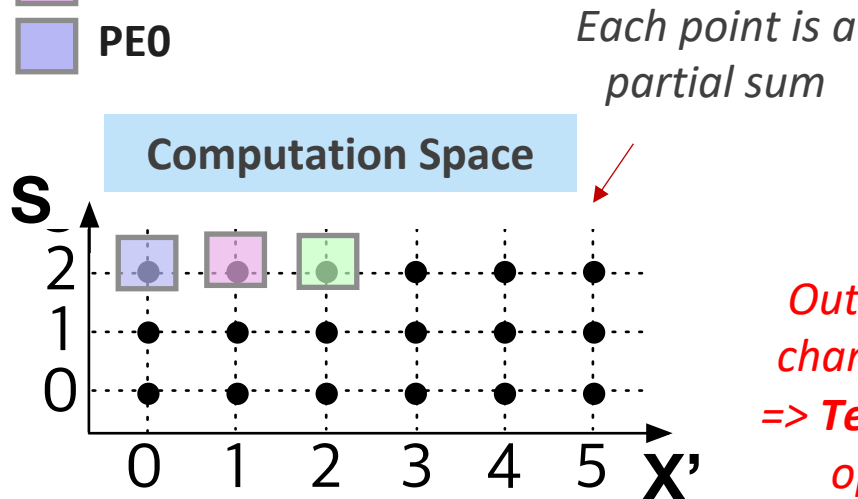
Data

PartialSum[X'][S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

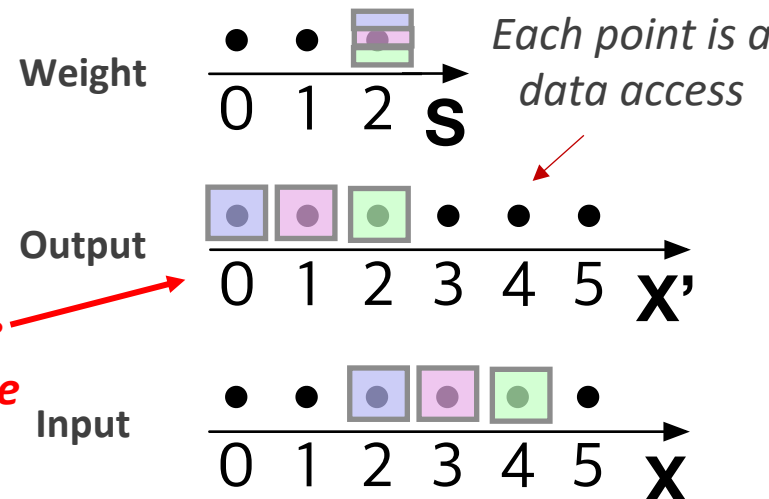


Time = 2



Output does not change over time
=> Temporal reuse opportunity

Data Space



Output Stationary (OS) Dataflow



Computation

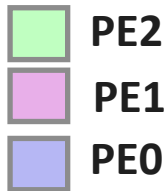
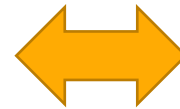
```

for(int x = 0; x < X'; x++)
  for(int s = 0; s < S; s++)
    Output[x] += Weight[s] * Input[x+s]
  
```

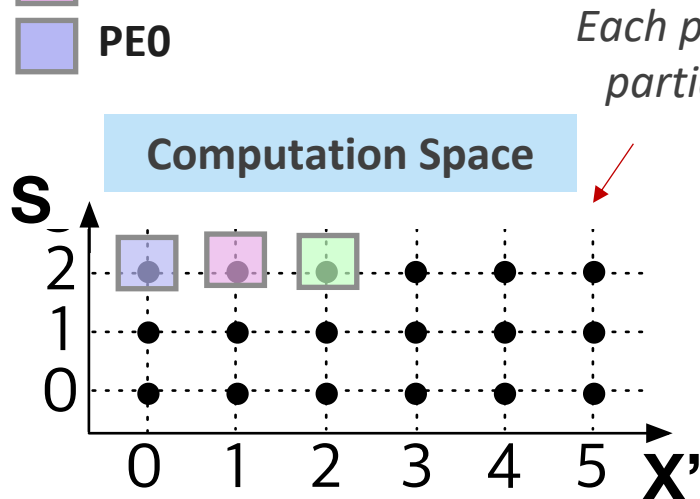
Data

PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]



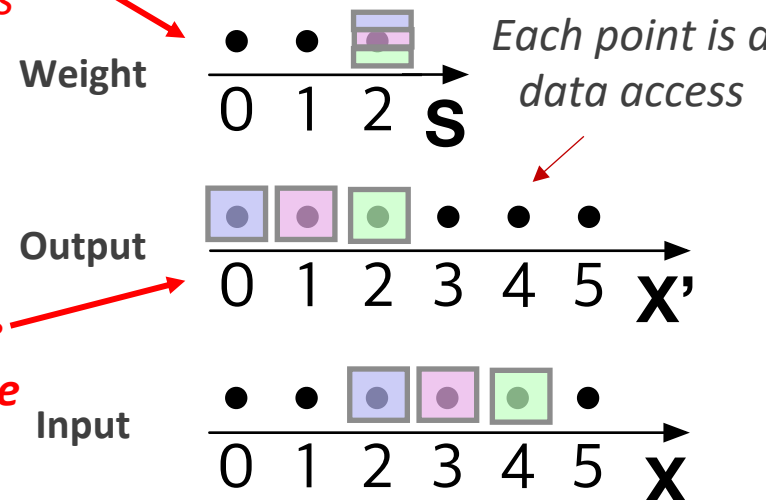
Time = 2



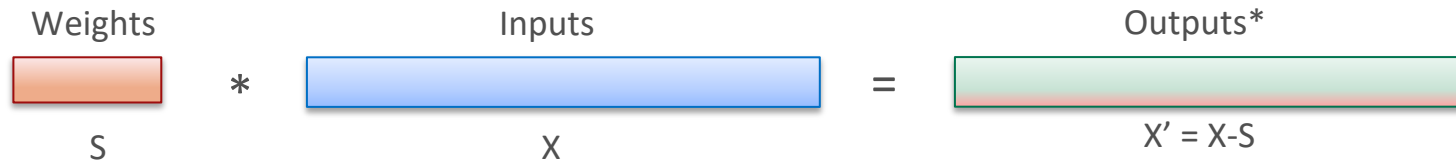
Spatial multicast opportunity for weights

Output does not change over time => Temporal reuse opportunity

Data Space



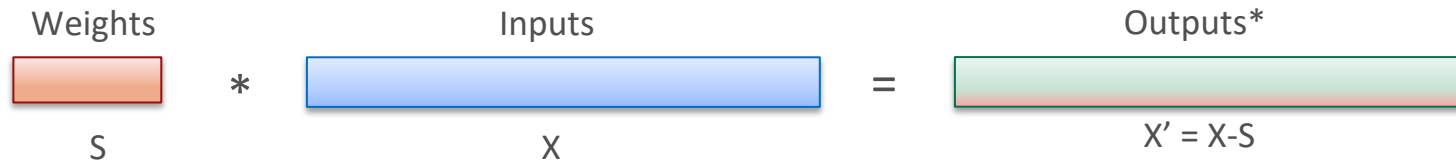
Describing OS dataflow



```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

Describing OS dataflow

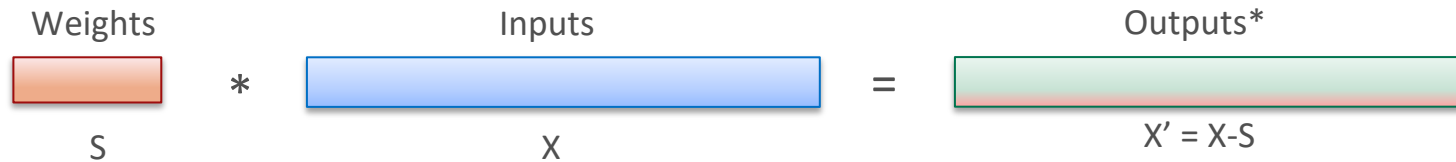


```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

How often does the datapath
change the weight and input?

Describing OS dataflow



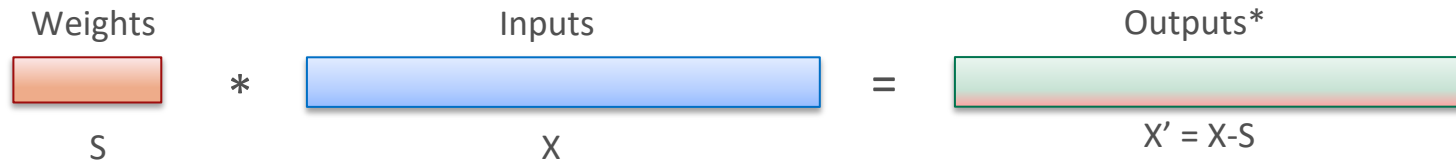
```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

How often does the datapath
change the weight and input?

Every cycle

Describing OS dataflow



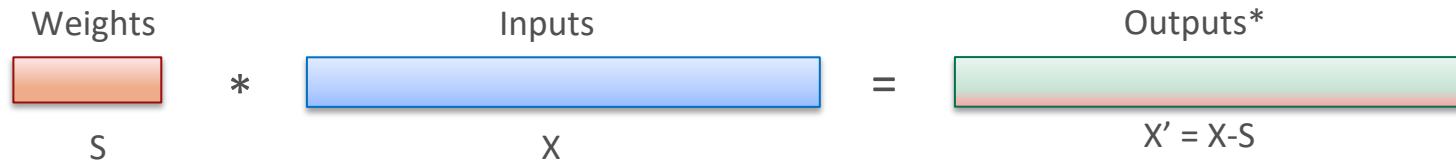
```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

How often does the datapath
change the weight and input?
Output?

Every cycle

Describing OS dataflow



```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

How often does the datapath
change the weight and input?

Output?

Every cycle

Every S cycles: “Output stationary”

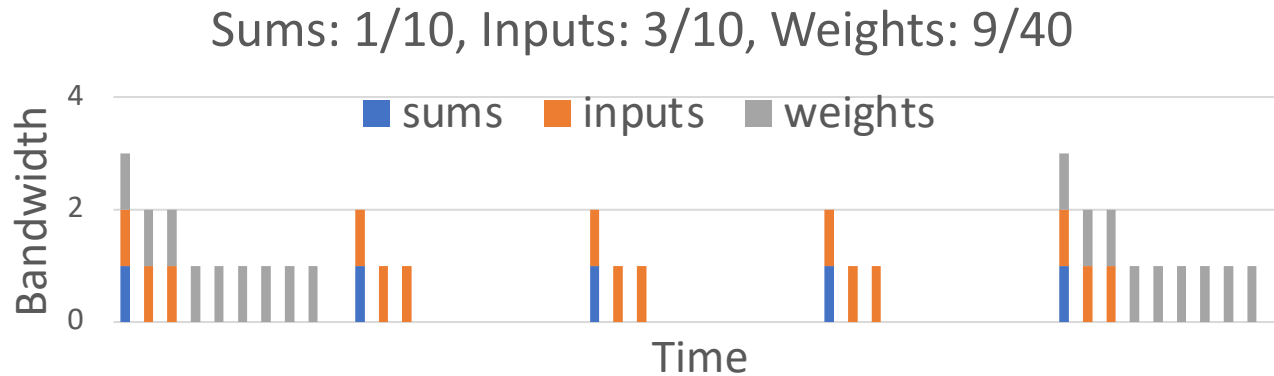
What do we mean by “stationary”?

What do we mean by “stationary”?

The datatype (and dimension) that changes most slowly

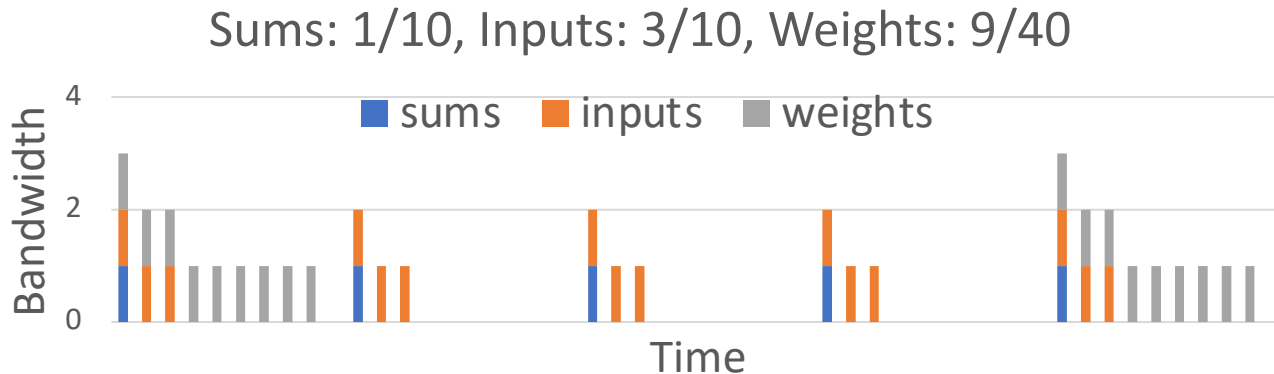
What do we mean by “stationary”?

The datatype (and dimension) that changes most slowly



What do we mean by “stationary”?

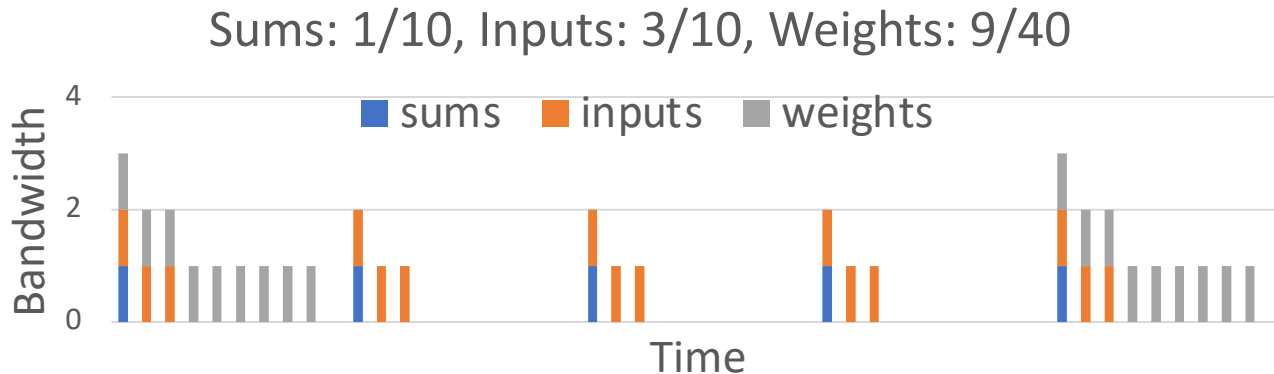
The datatype (and dimension) that changes most slowly



- Imprecise analogy: think of data transfers as a wave with “amplitude” and “period”
 - The stationary datatype has the **longest** period (locally held tile changes most slowly)
 - Note: like waves, may have harmful “interference” (bursts)
 - intermediate staging buffers reduce both bandwidth and energy

What do we mean by “stationary”?

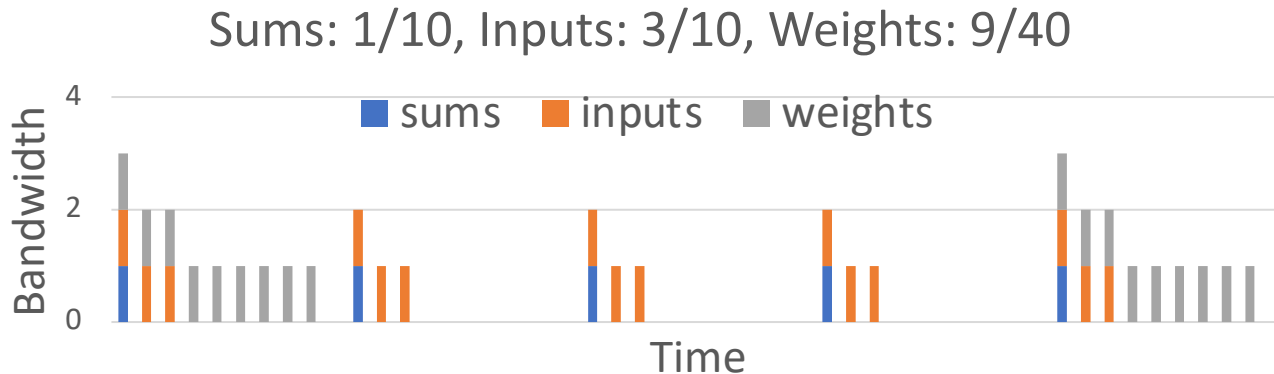
The datatype (and dimension) that changes most slowly



- Imprecise analogy: think of data transfers as a wave with “amplitude” and “period”
 - The stationary datatype has the **longest** period (locally held tile changes most slowly)
 - Note: like waves, may have harmful “interference” (bursts)
 - intermediate staging buffers reduce both bandwidth and energy
- Often corresponds to datatype that is “done with” earliest without further reloads

What do we mean by “stationary”?

The datatype (and dimension) that changes most slowly



- Imprecise analogy: think of data transfers as a wave with “amplitude” and “period”
 - The stationary datatype has the **longest** period (locally held tile changes most slowly)
 - Note: like waves, may have harmful “interference” (bursts)
 - intermediate staging buffers reduce both bandwidth and energy
- Often corresponds to datatype that is “done with” earliest without further reloads
- **Note:** the “stationary” name is meant to give intuition, not to be a complete specification of all the behavior of a dataflow

“Done with” vs “Needs Reload”

```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

“Done with” vs “Needs Reload”

```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

How many times
will $x == 2$?

“Done with” vs “Needs Reload”

```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

How many times
will $x == 2$?

How many times
will $x+s == 2$?

“Done with” vs “Needs Reload”

```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

How many times
will $x == 2$?

How many times
will $x+s == 2$?

How many times
will $s == 2$?

“Done with” vs “Needs Reload”

```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

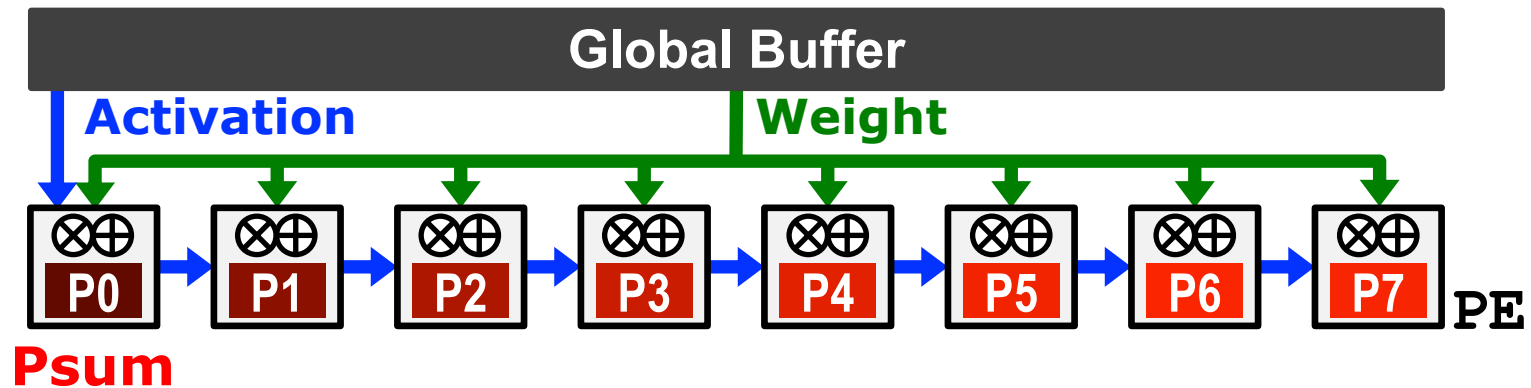
How many times
will $x == 2$?

How many times
will $x+s == 2$?

How many times
will $s == 2$?

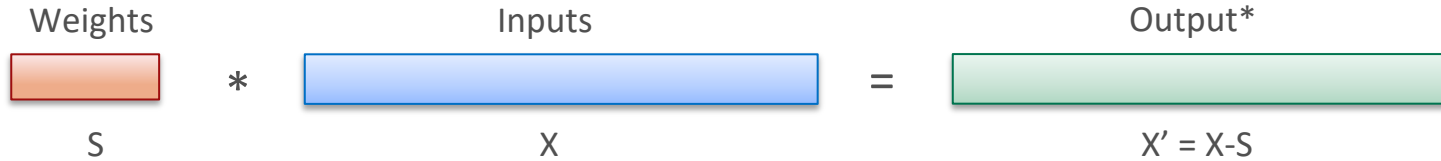
- Temporal distance between re-occurrence dictates buffer size to avoid re-load
- How do you know if a buffer that size is worth it?

OS Dataflow Implementation



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

Weight Stationary (WS) Dataflow



Computation

```
for(int x = 0; x < X'; x++)
  for(int s = 0; s < S; s++)
    Output[x] += Weight[s] * Input[x+s]
```

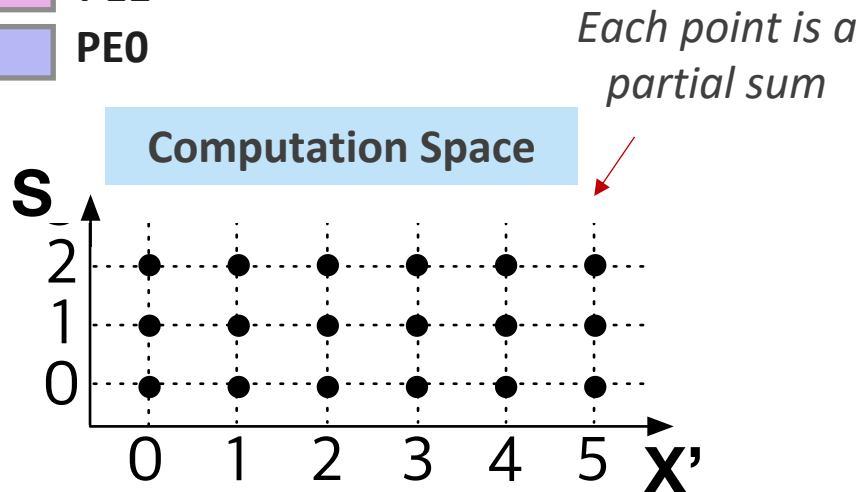


Data

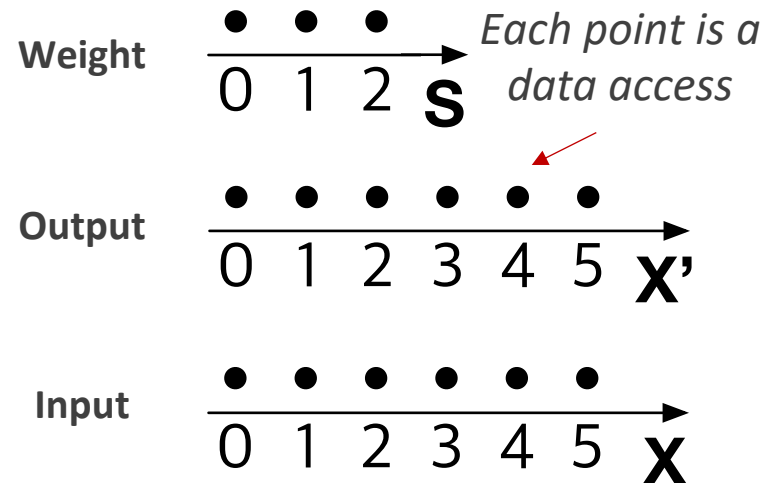
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

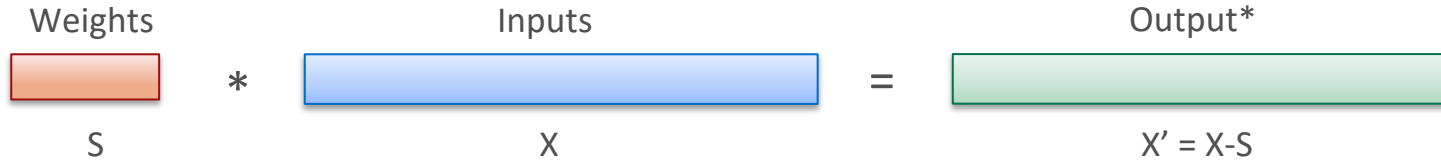
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

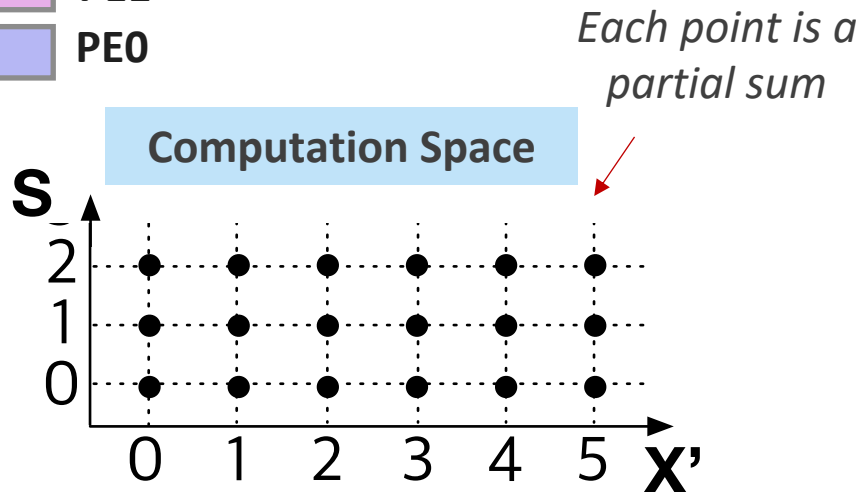
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

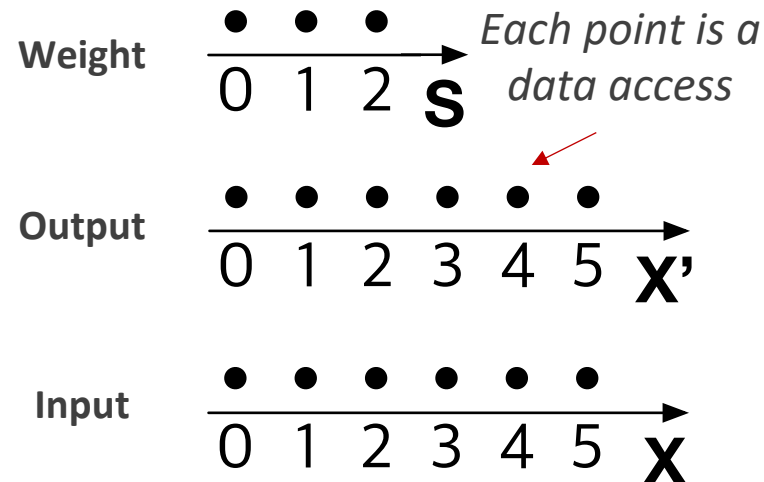
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

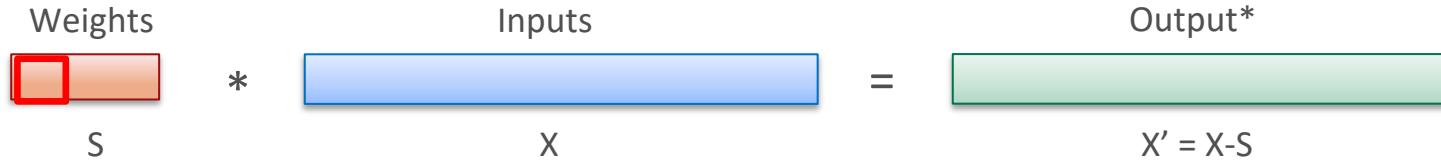
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

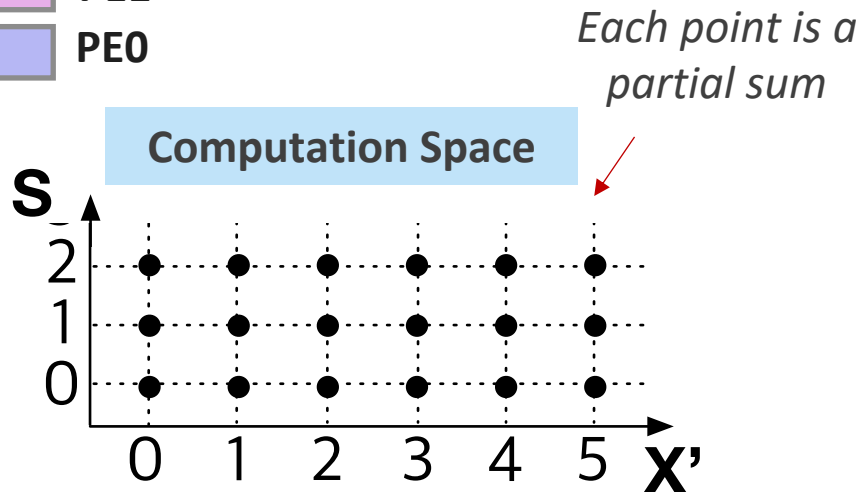
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

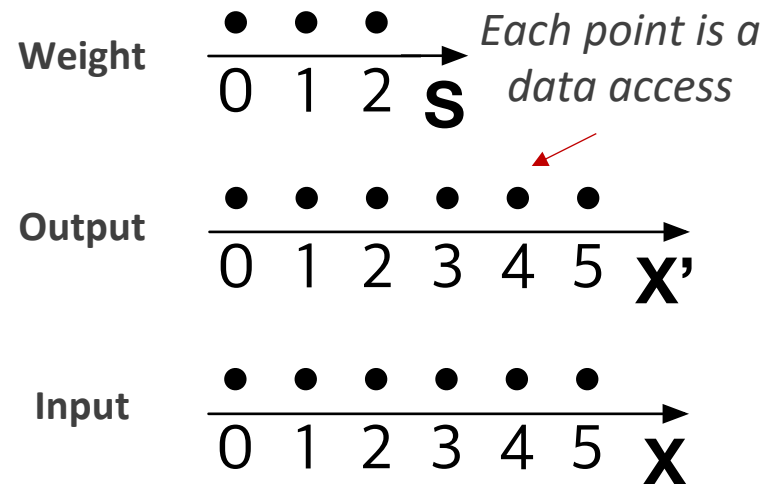
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

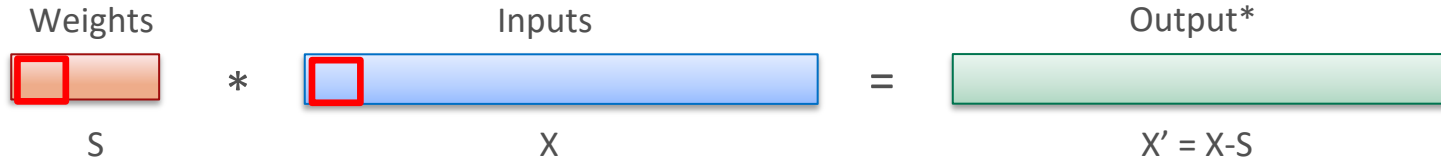
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

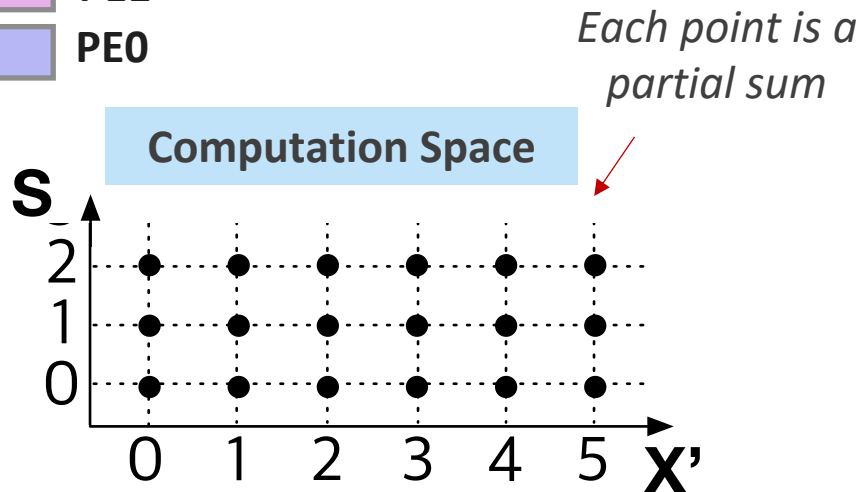
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

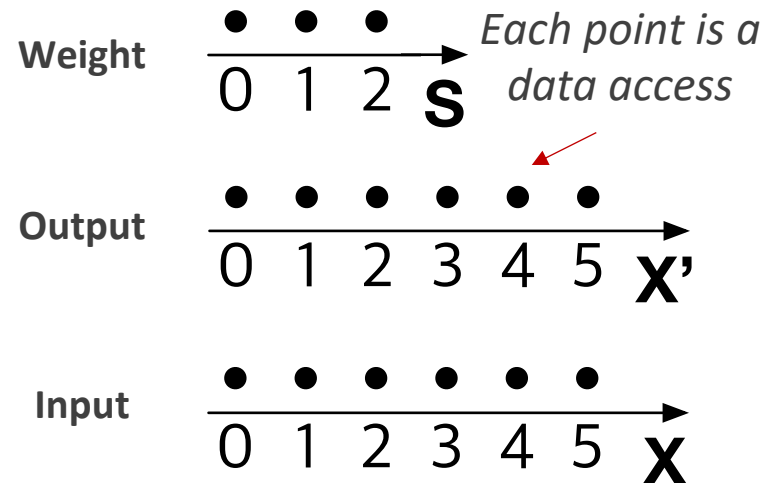
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

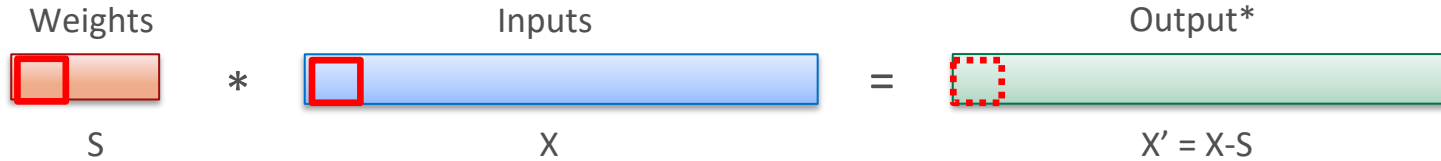
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



```

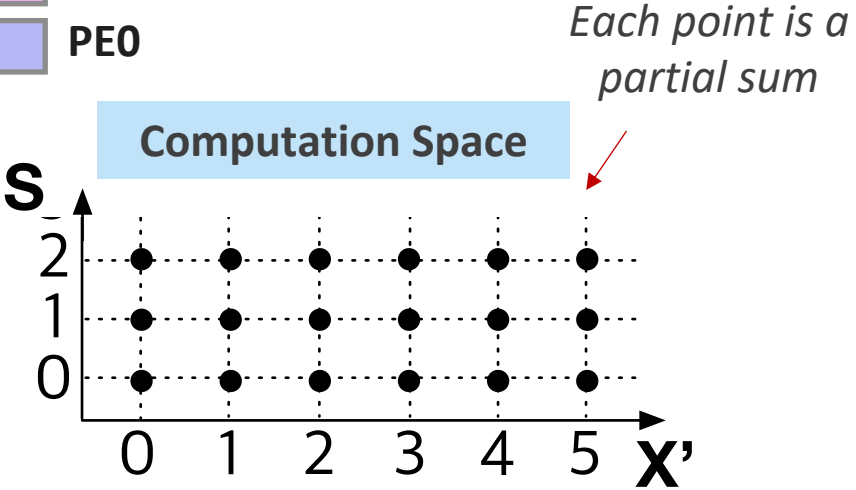
Computation
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
    
```



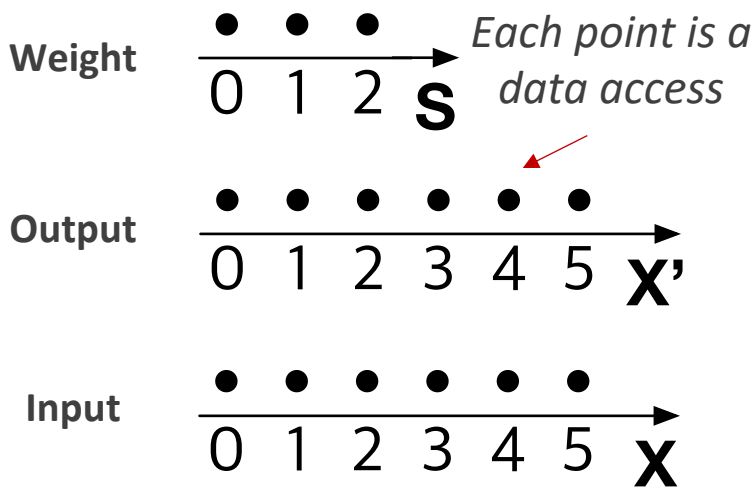
Data
 PartialSum[X'] [S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

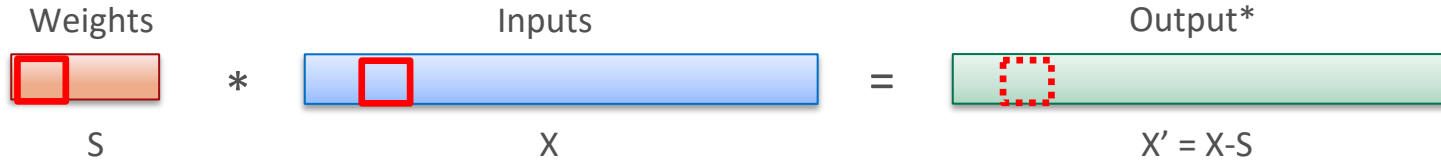
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

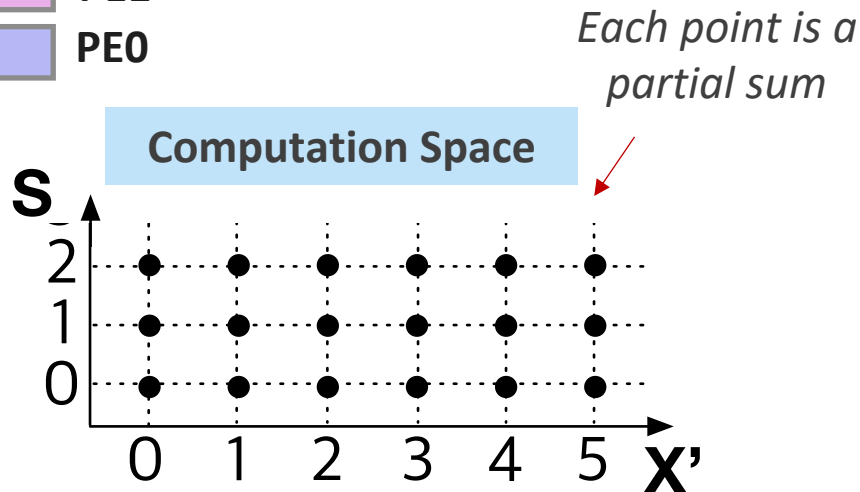
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

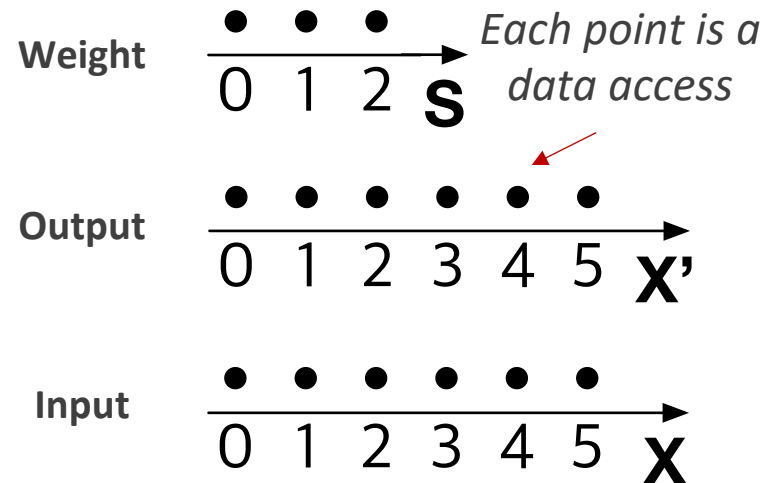
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

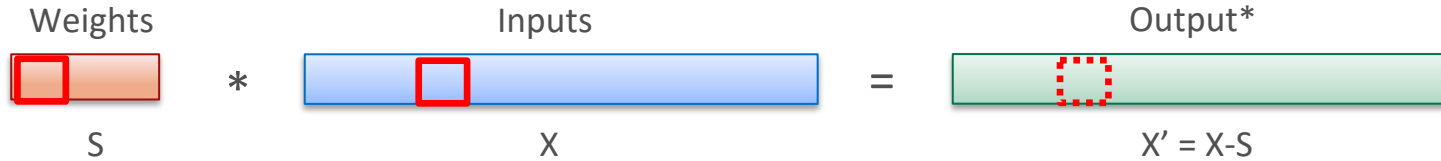
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

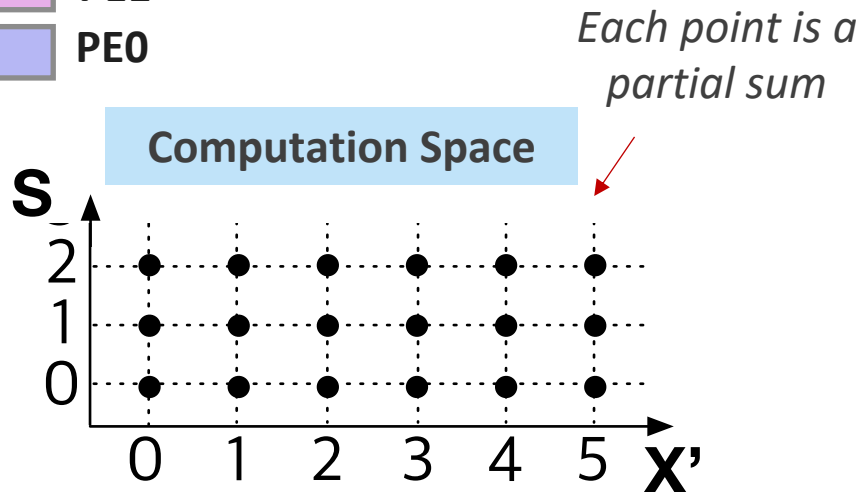
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

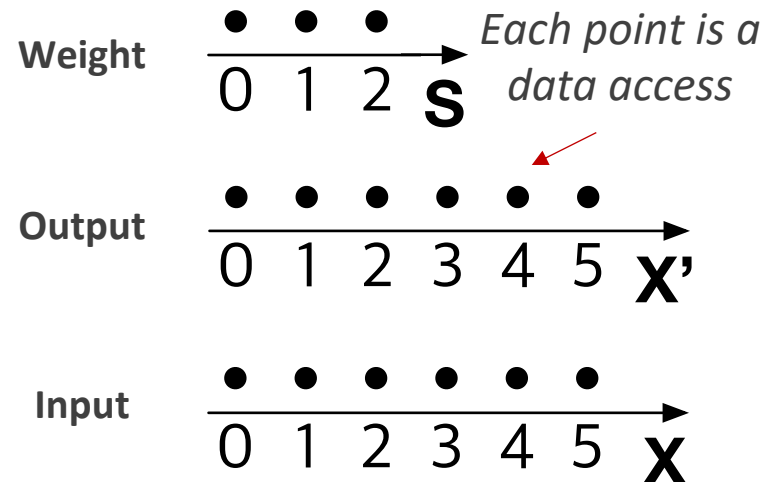
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

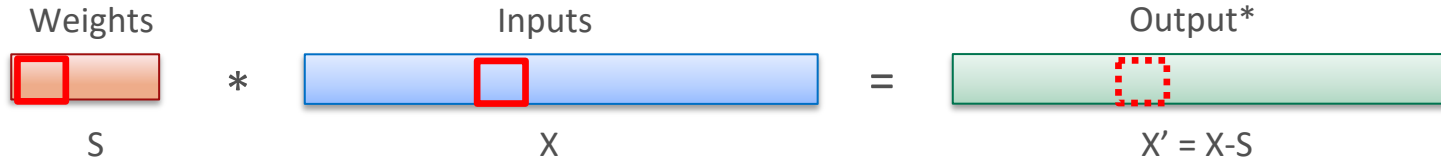
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

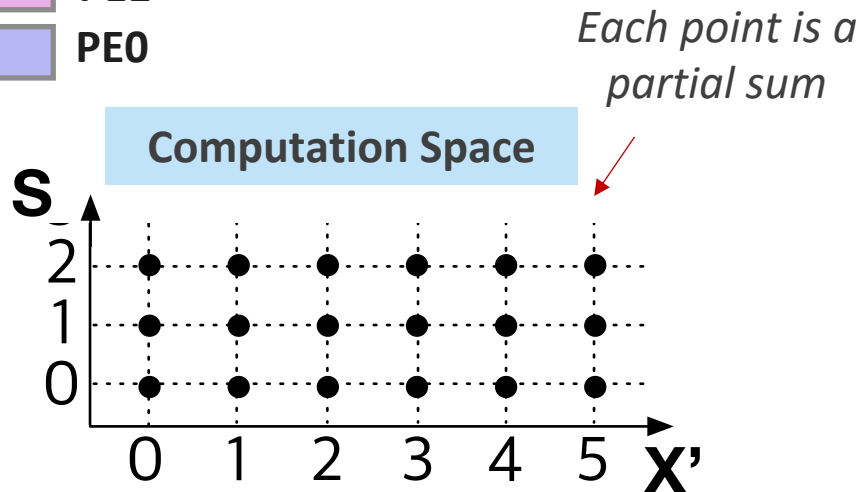
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

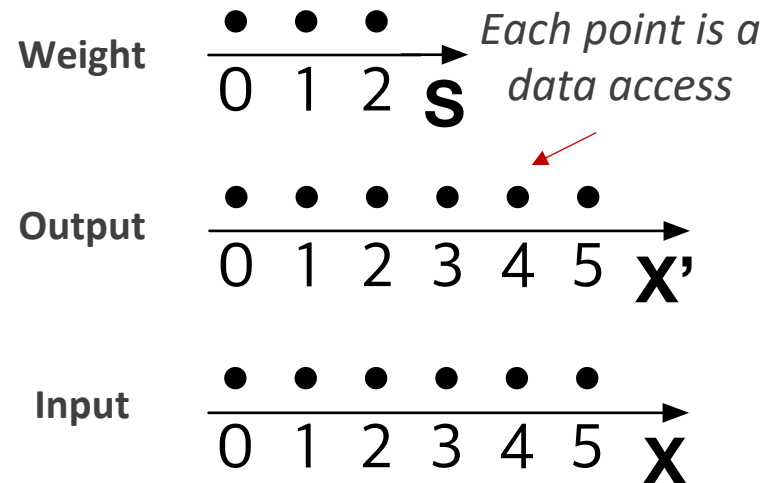
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

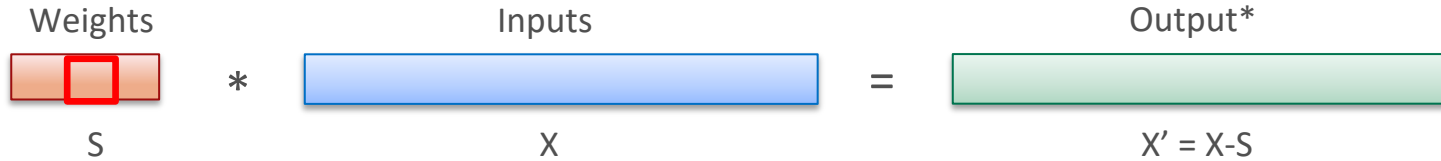
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

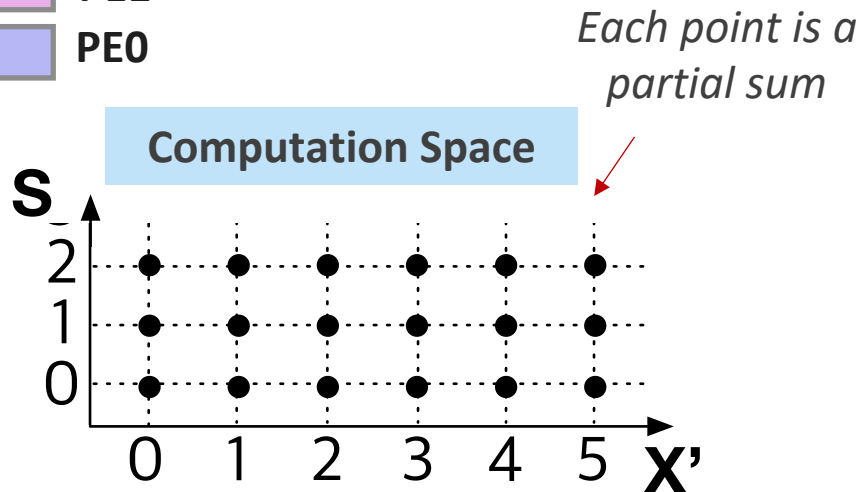
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

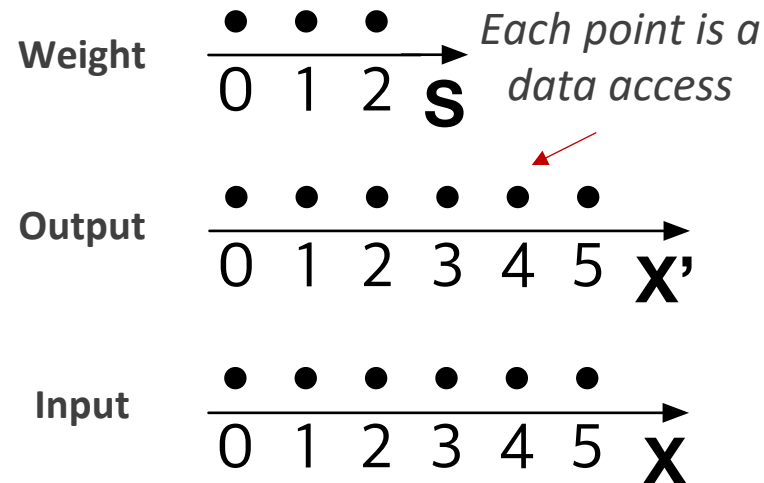
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

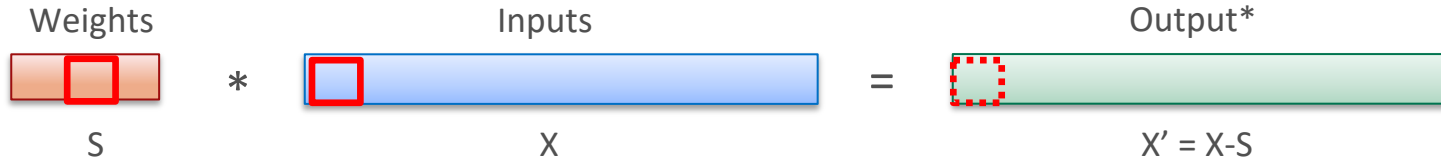
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

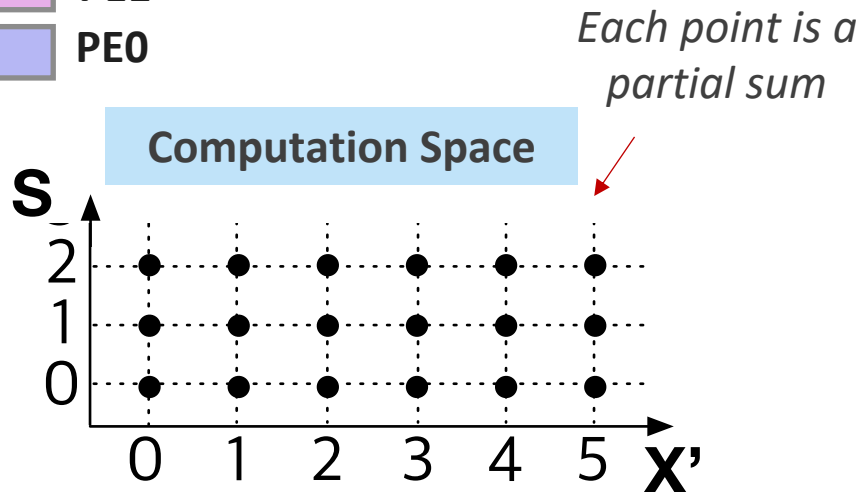
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

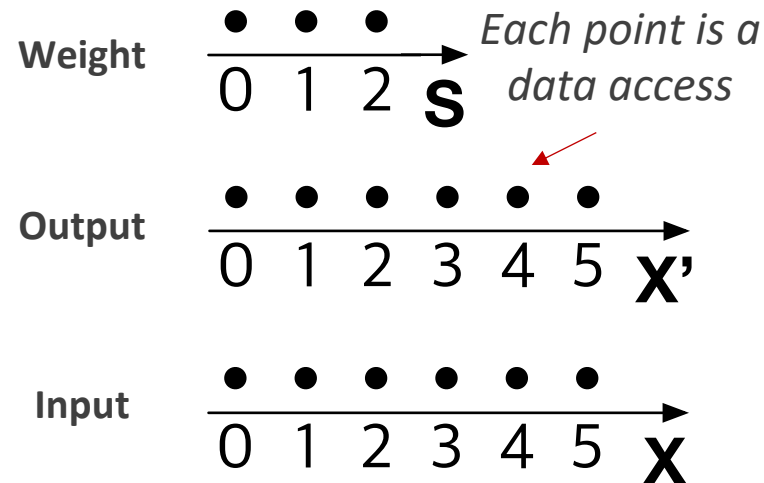
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

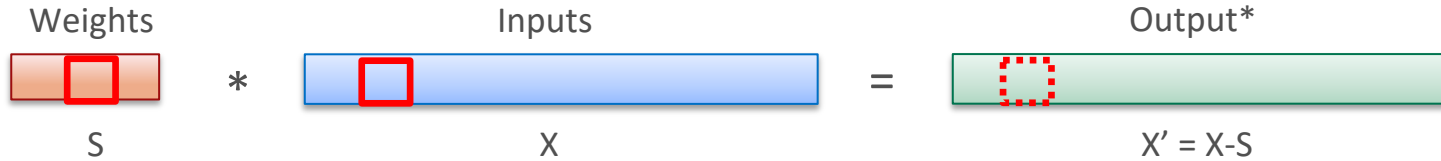
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

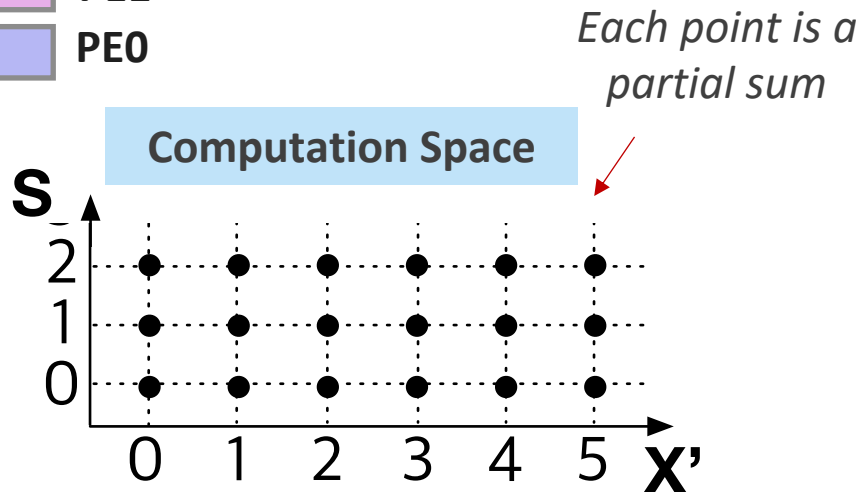
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

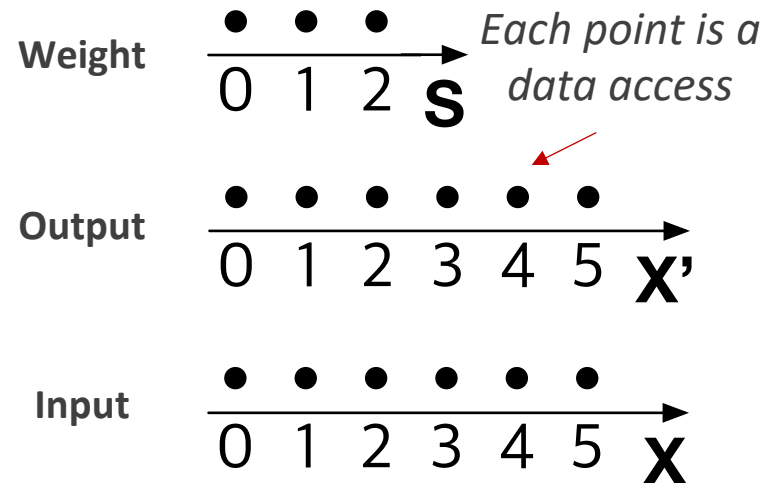
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

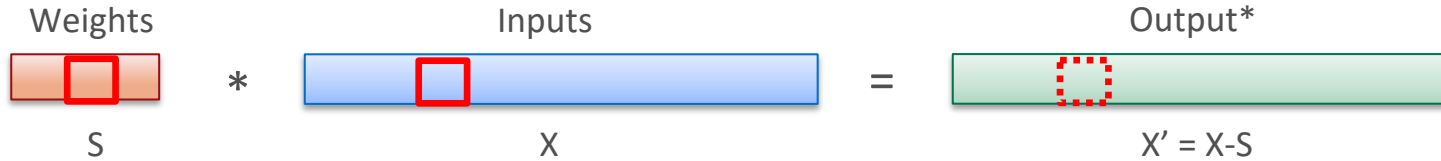
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

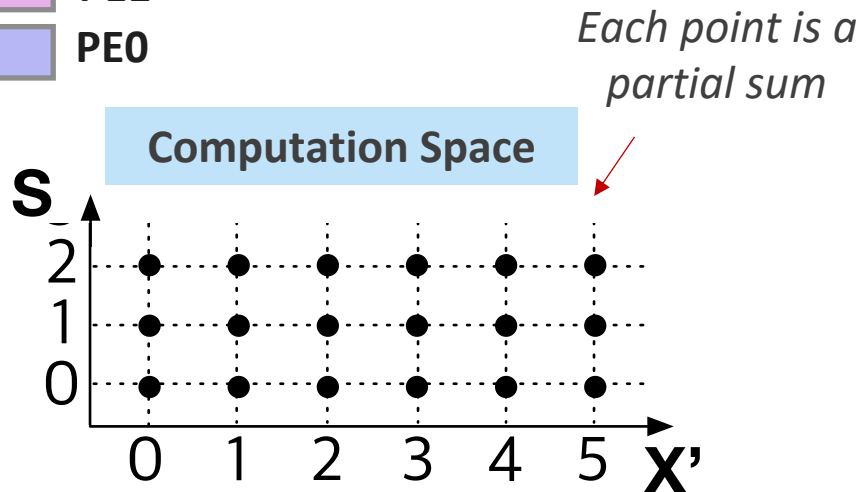
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

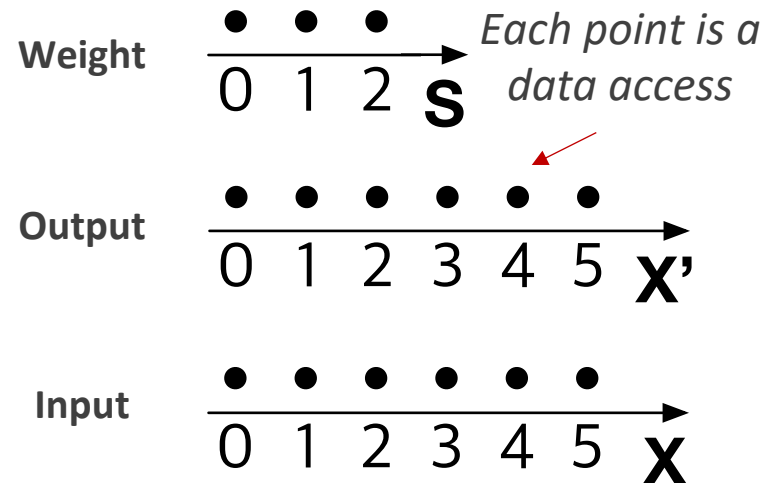
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

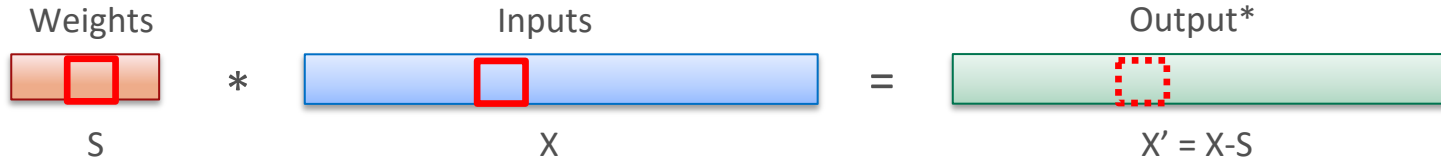
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

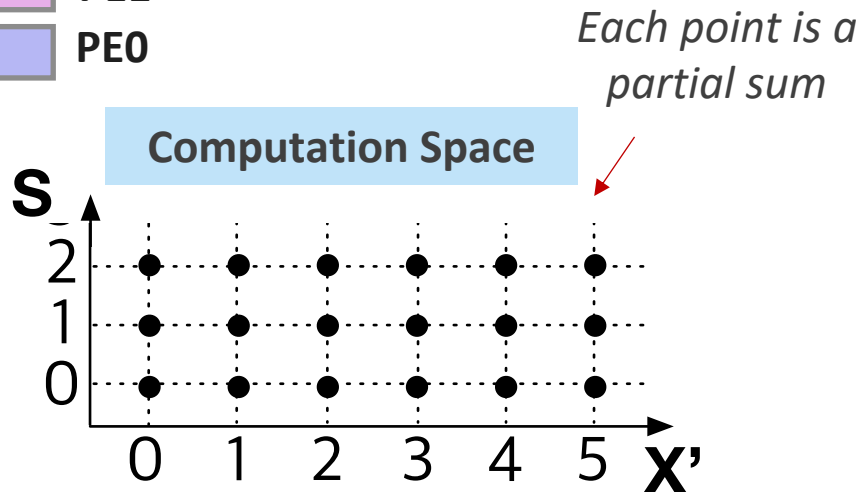
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

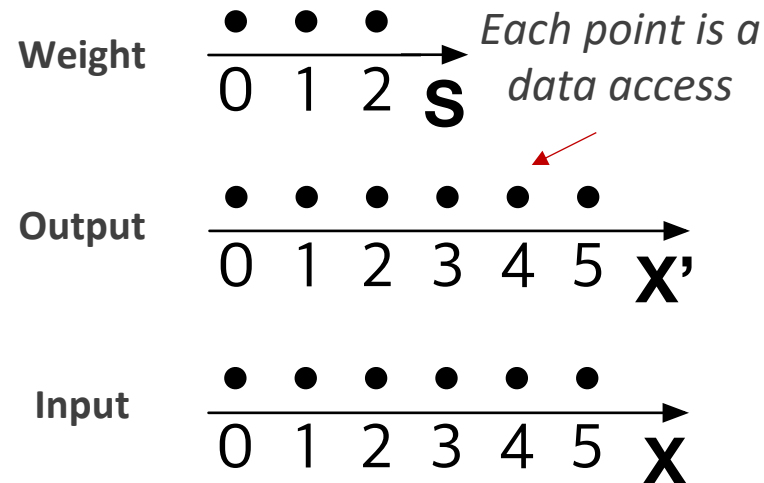
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

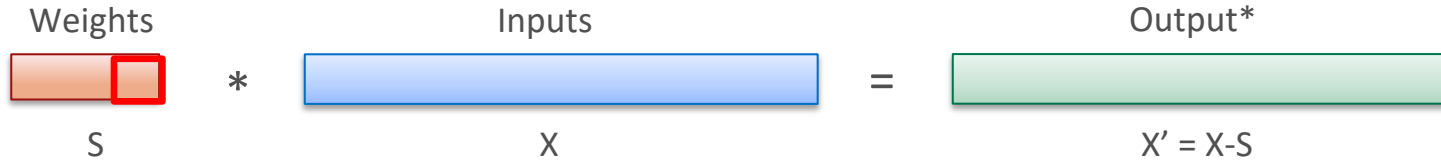
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

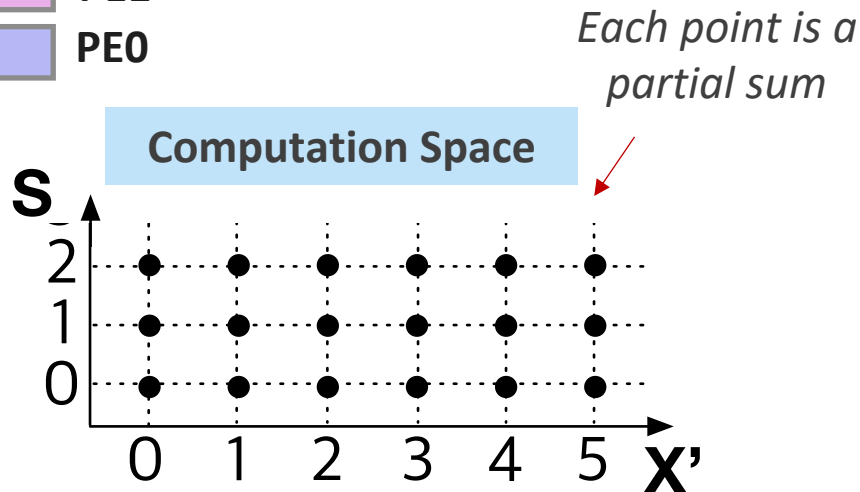
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

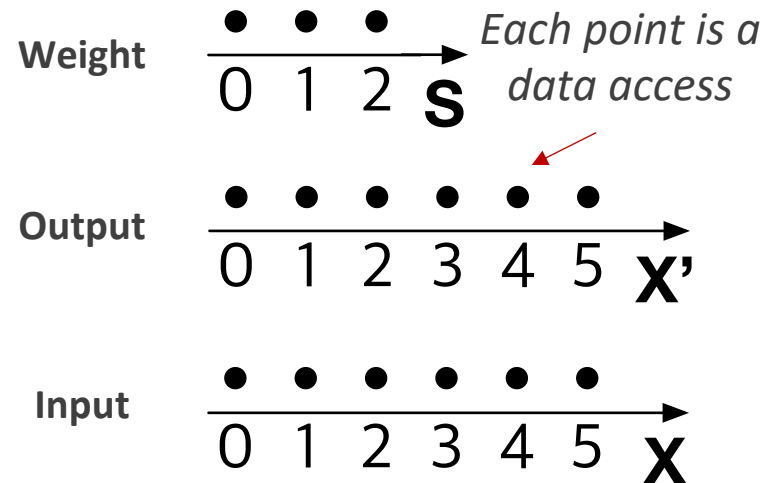
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

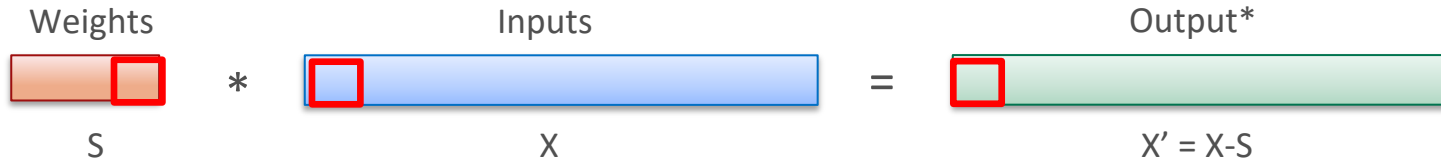
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

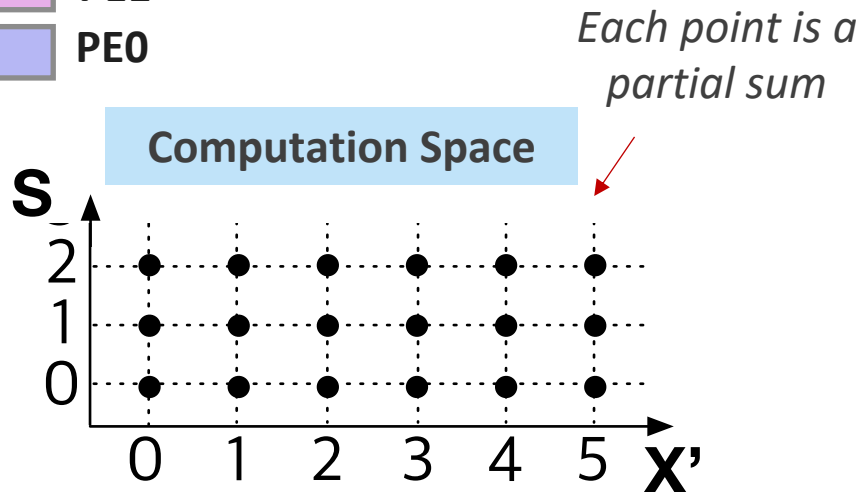
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

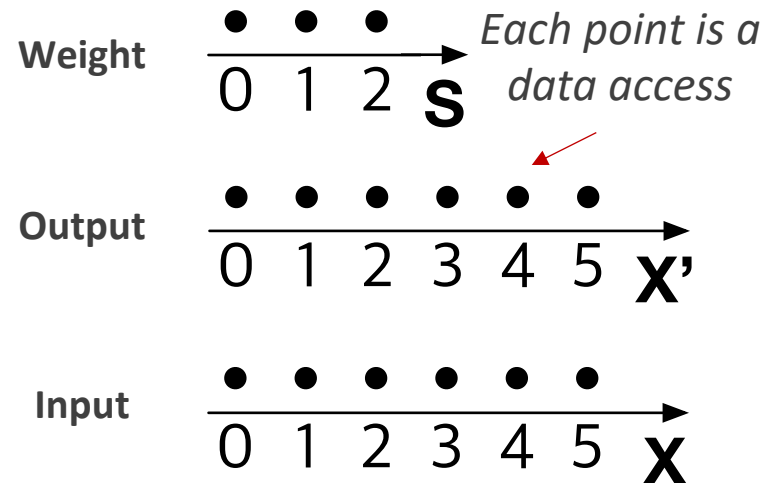
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

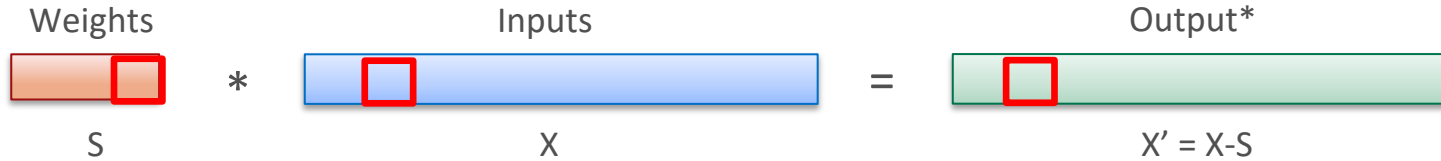
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

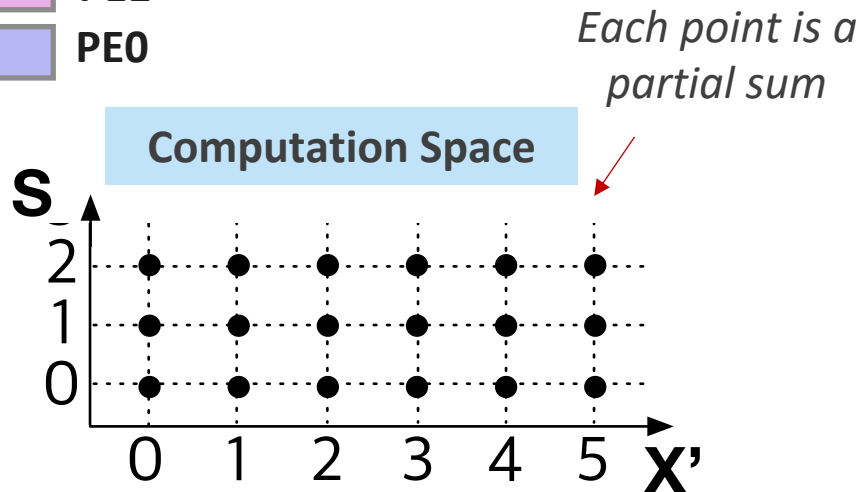
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

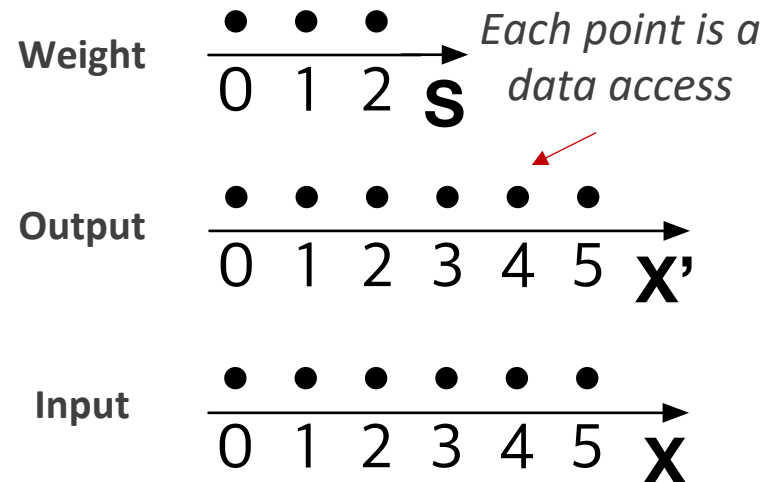
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

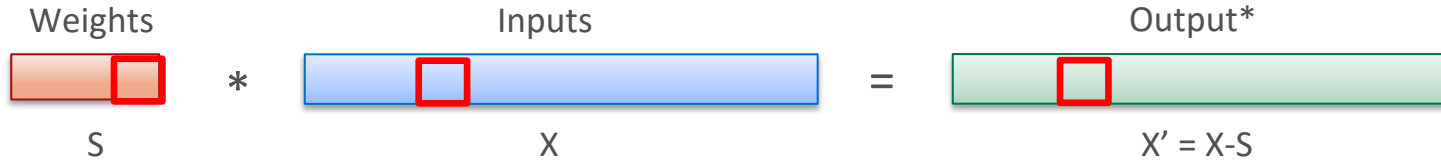
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

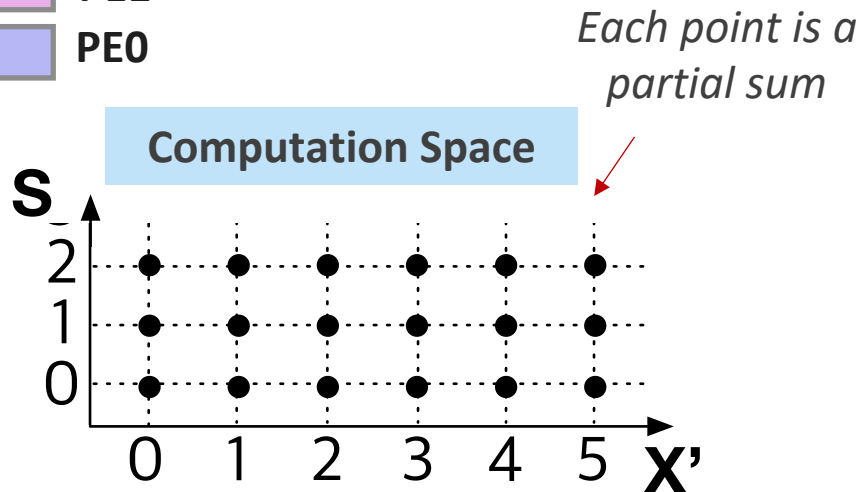
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

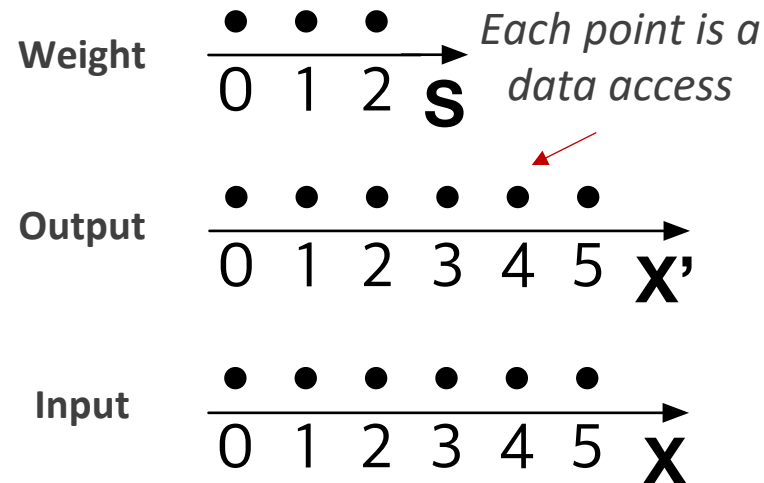
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

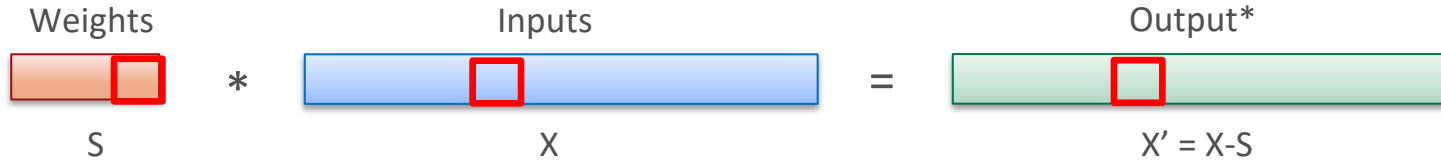
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

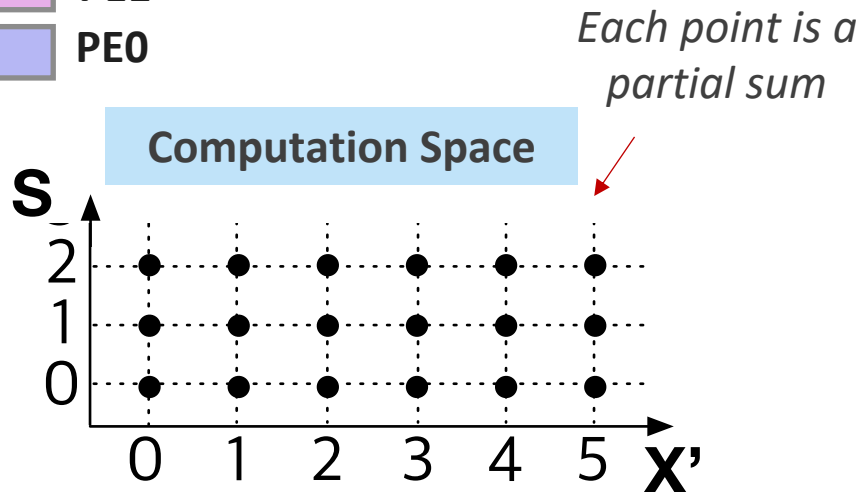
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

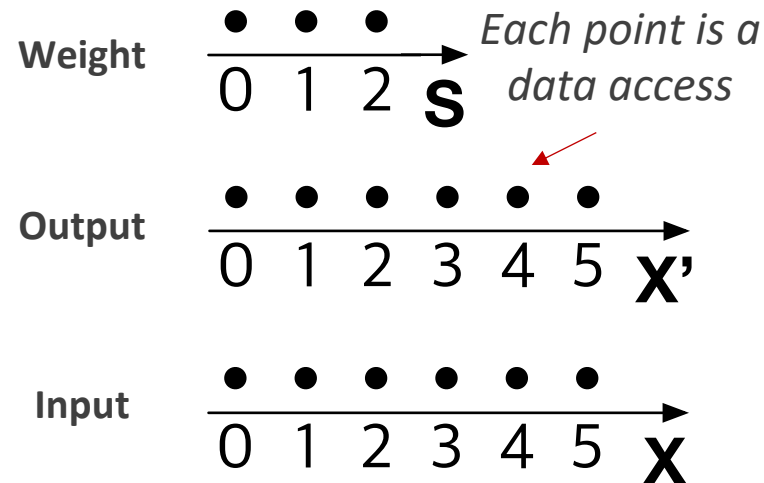
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

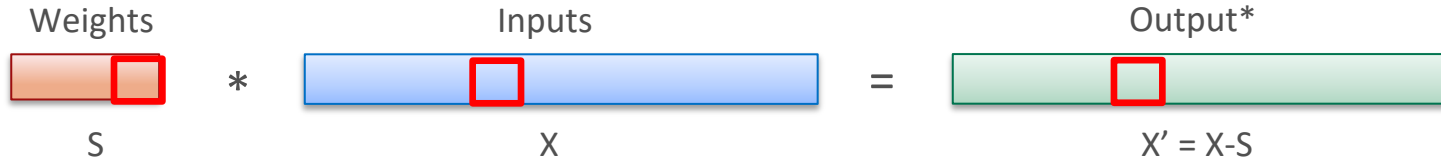
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

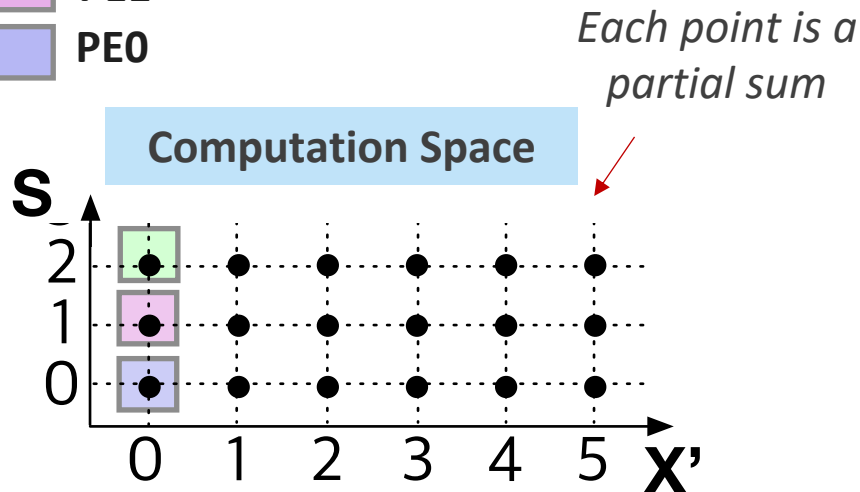
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

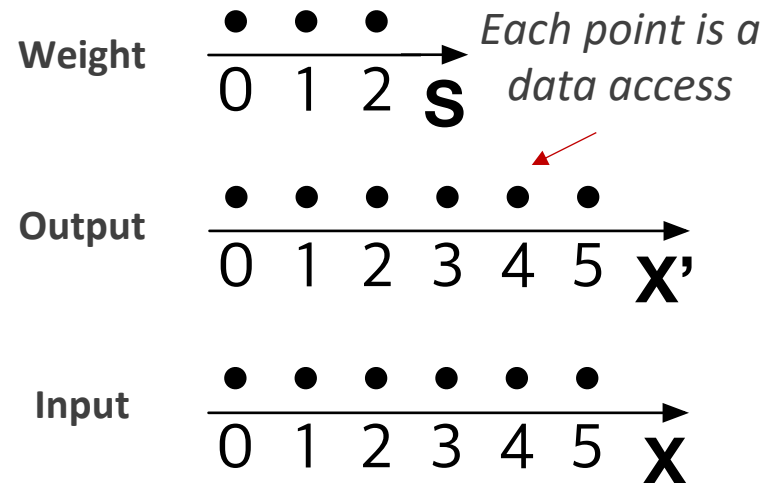
PartialSum[X'] $[S]$ needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

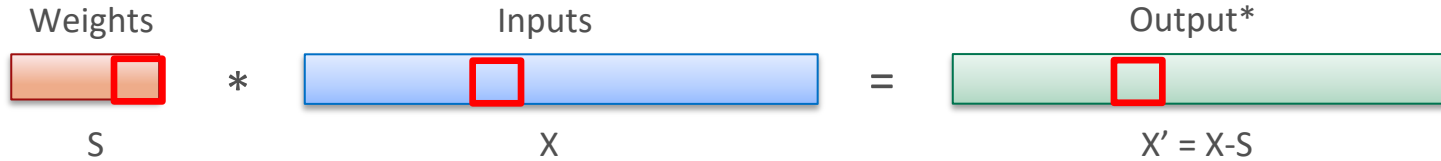
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

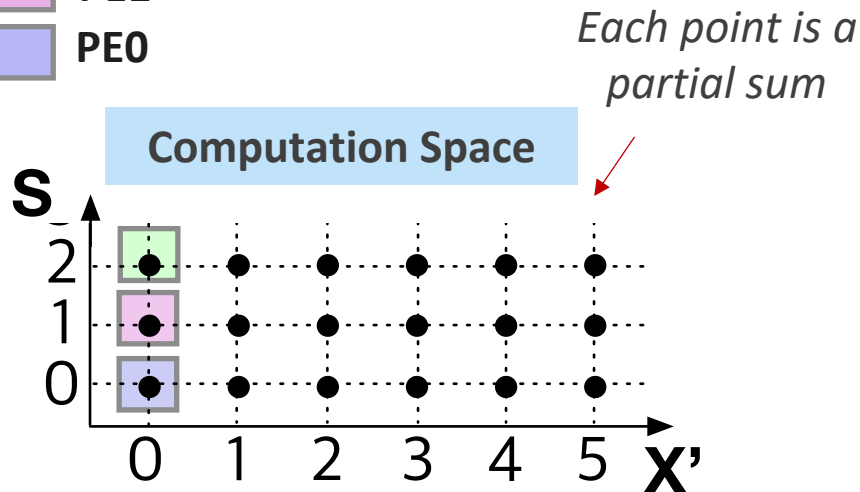
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

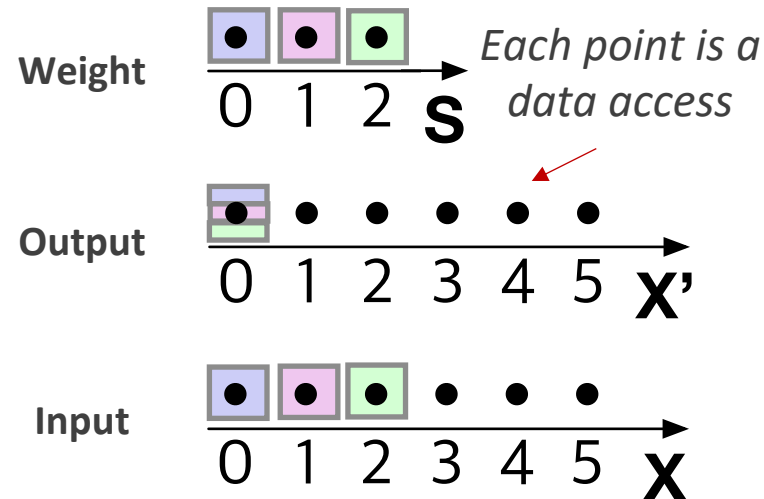
PartialSum[X'] [S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

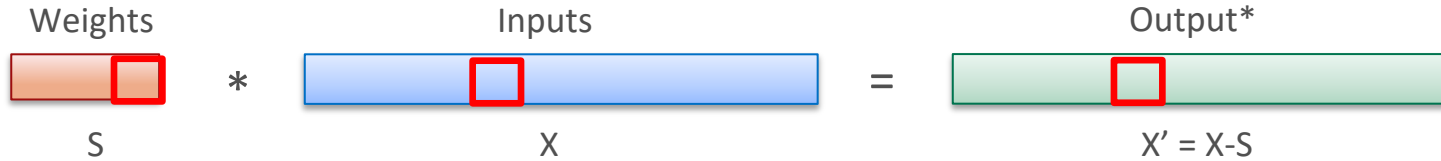
- PE2
- PE1
- PE0



Data Space



Weight Stationary (WS) Dataflow



Computation

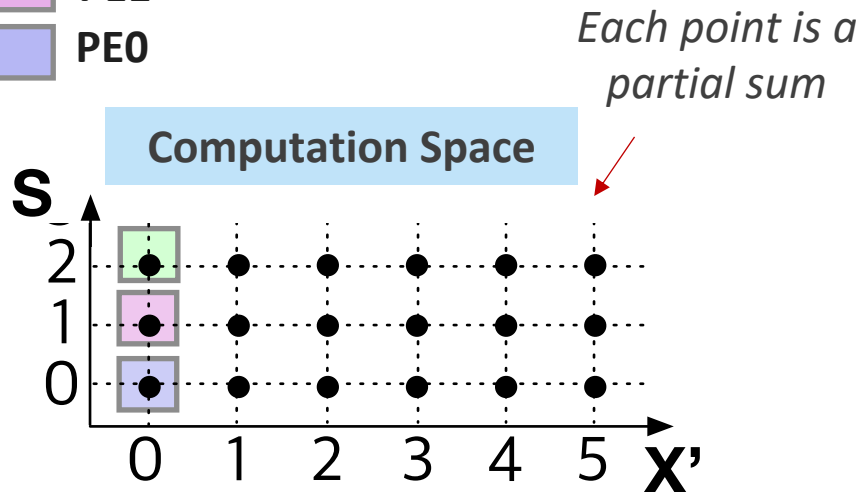
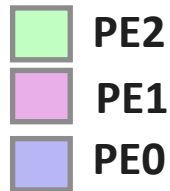
```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

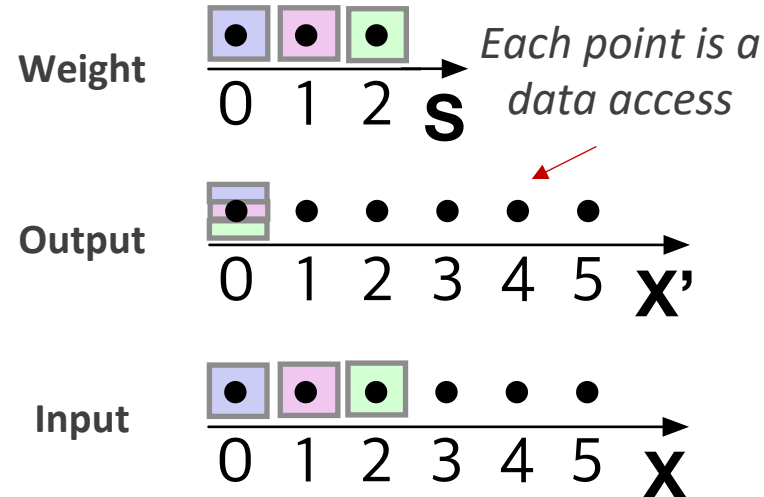
PartialSum[X'] [S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

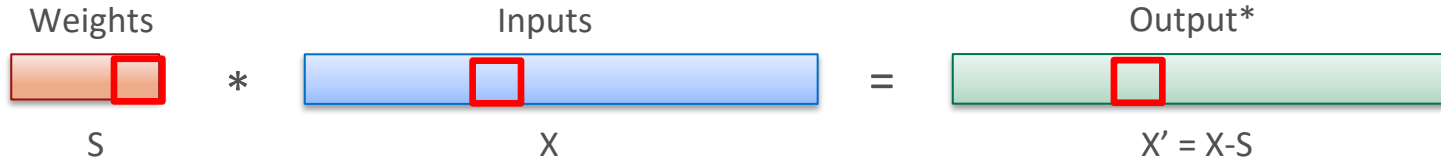
Time = 0



Data Space



Weight Stationary (WS) Dataflow



Computation

```
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
```

Data

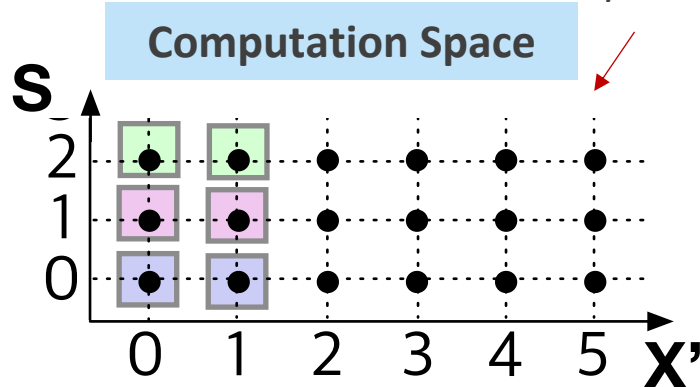
PartialSum[X'] [S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

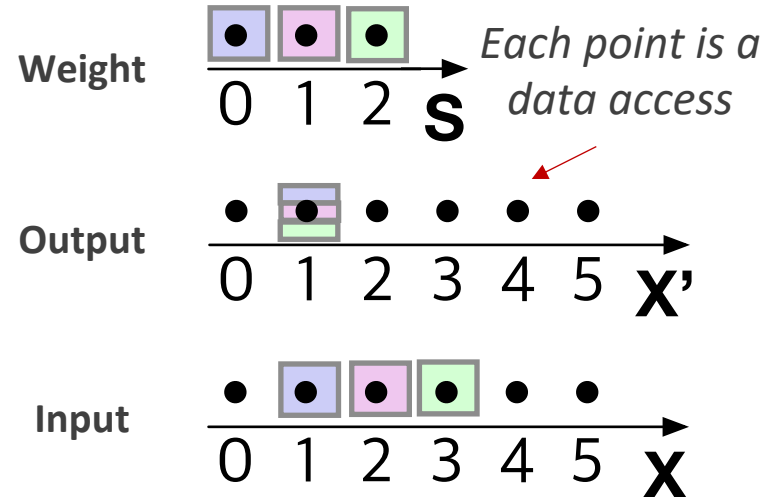
Time = 1

- PE2
- PE1
- PE0

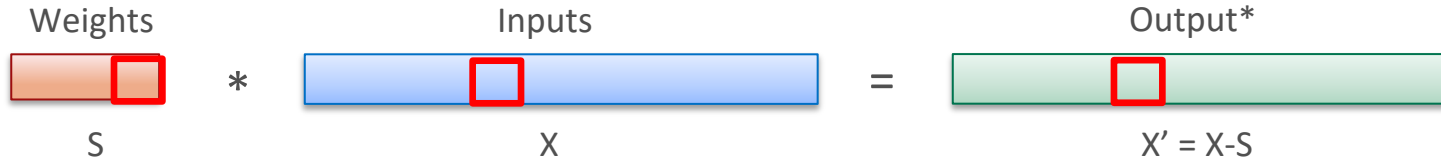
Each point is a partial sum



Data Space



Weight Stationary (WS) Dataflow



```

Computation
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
    
```

Data
 PartialSum[X'] [S] needs to access:

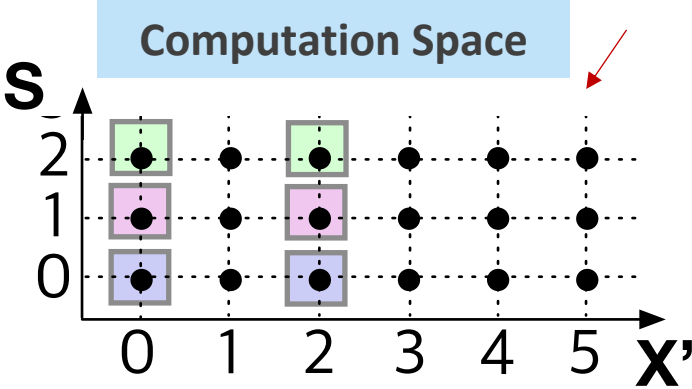
- Weight[s]
- Output[x']
- Input[x'+s]



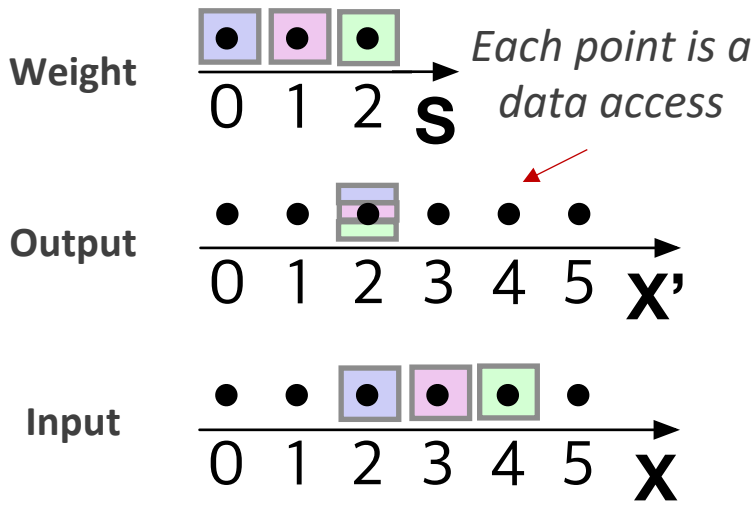
- PE2
- PE1
- PE0

Time = 2

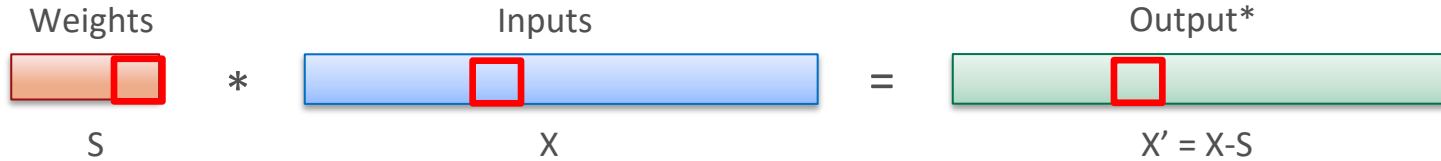
Each point is a partial sum



Data Space



Weight Stationary (WS) Dataflow



```

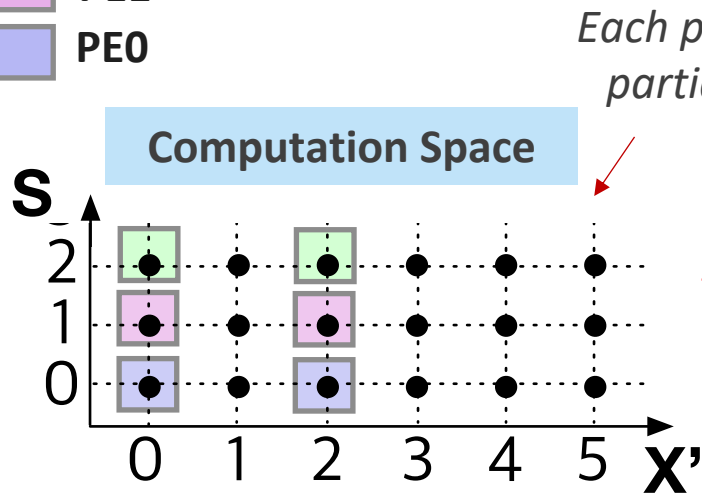
Computation
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
    
```

Data
 PartialSum[X'] [S] needs to access:

- Weight[s]
- Output[x']
- Input[x'+s]

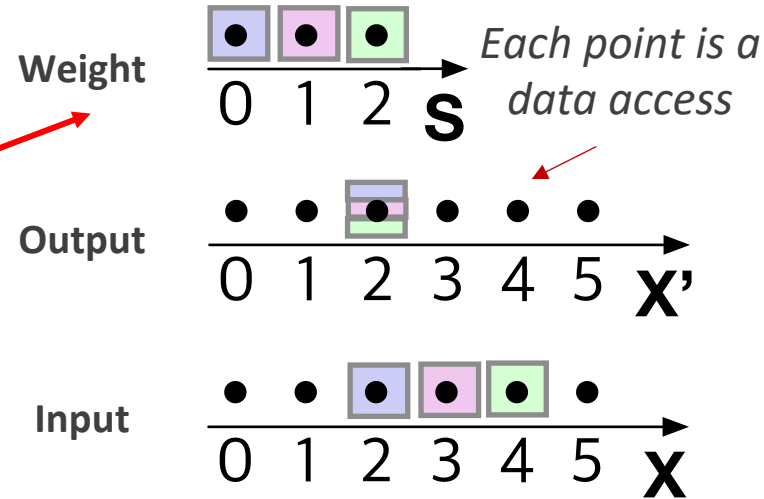
- PE2
- PE1
- PE0

Time = 2

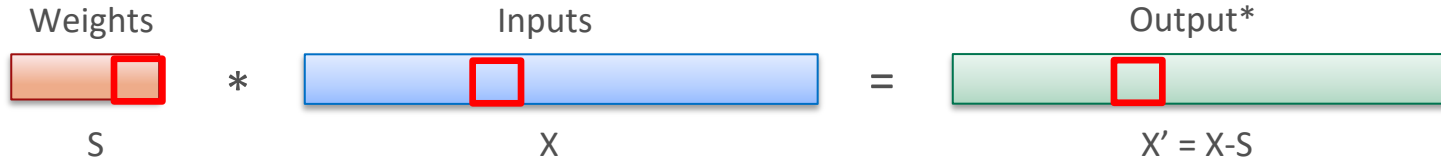


Weight does not change over time
 => **Temporal reuse opportunity**

Data Space



Weight Stationary (WS) Dataflow



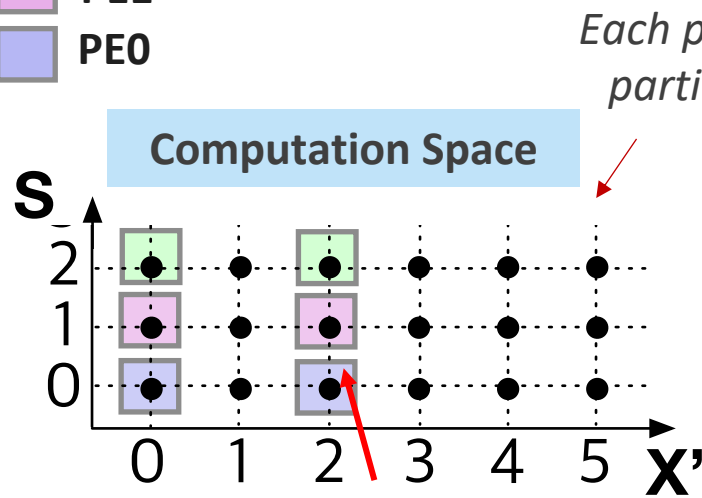
```

Computation
for(int s = 0; s < S; s++)
  for(int x = 0; x < X'; x++)
    Output[x] += Weight[s] * Input[x+s]
    
```

- Data**
 PartialSum[X'] [S] needs to access:
- Weight[s]
 - Output[x']
 - Input[x'+s]

- PE2
- PE1
- PE0

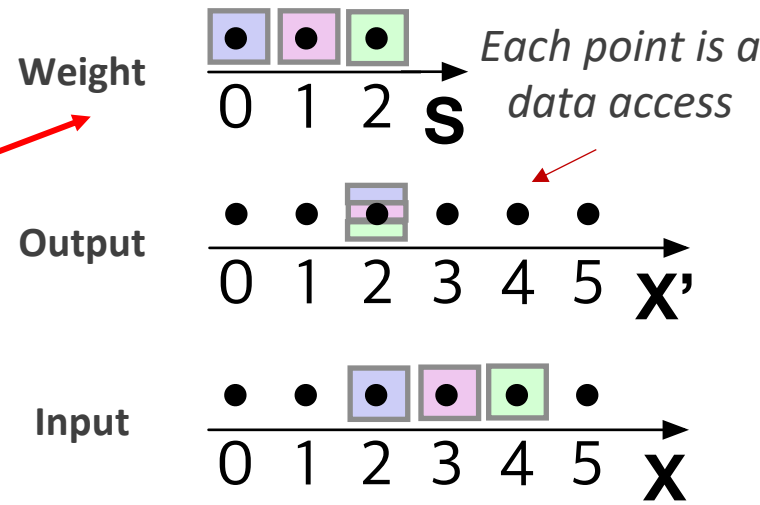
Time = 2



Weight does not change over time
 => **Temporal reuse opportunity**

Need Spatial reduction for output

Data Space



Describing WS Dataflow

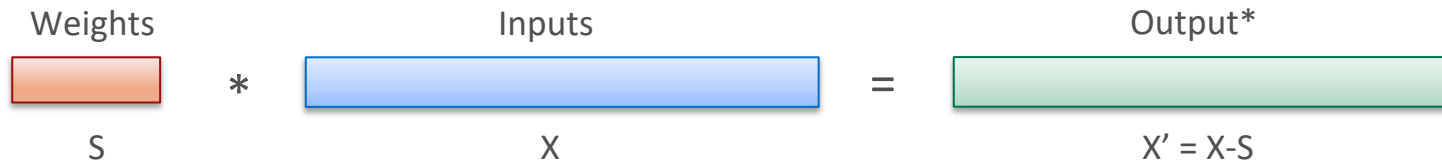


Computation

```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (s = 0; s < S; s++) {
    for (x = 0; x < X'; x++) {
        o[x] += i[x+s]*w[s];
    }
}
```

Describing WS Dataflow



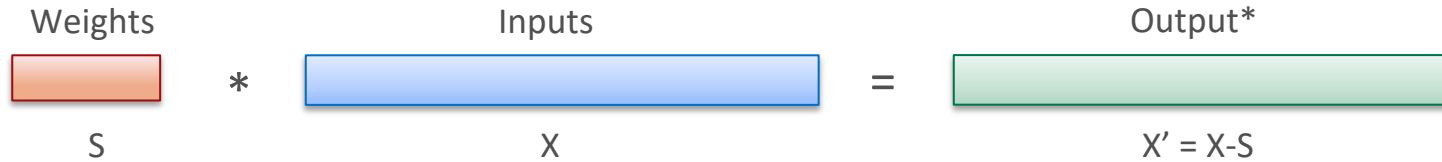
Computation

```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (s = 0; s < S; s++) {
    for (x = 0; x < X'; x++) {
        o[x] += i[x+s]*w[s];
    }
}
```

What about the loop nest makes it weight stationary?

Describing WS Dataflow



Computation

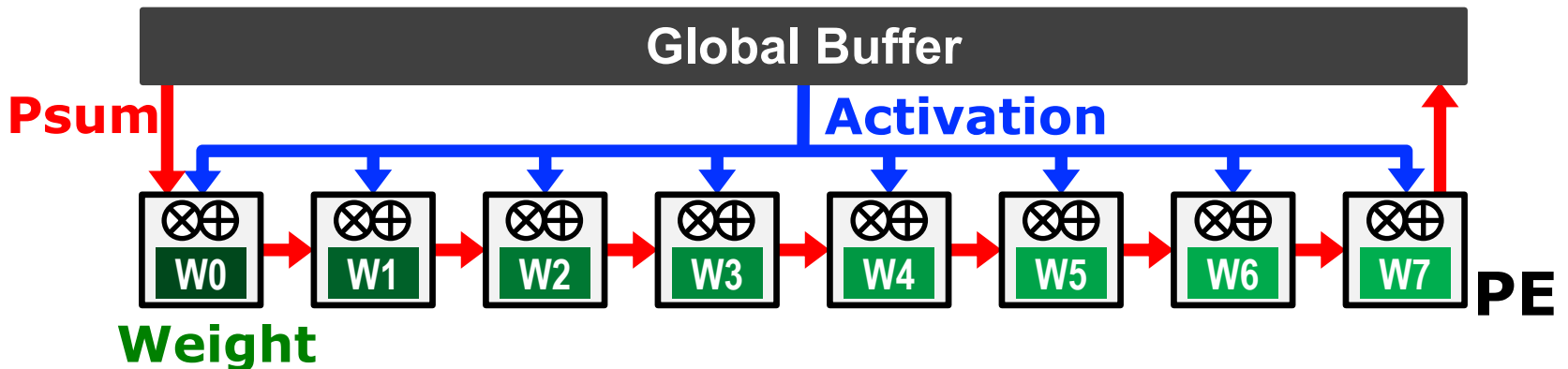
```
int i[X];      # Input activations
int w[S];      # Filter weights
int o[X'];     # Output activations

for (s = 0; s < S; s++) {
    for (x = 0; x < X'; x++) {
        o[x] += i[x+s]*w[s];
    }
}
```

What about the loop nest makes it weight stationary?

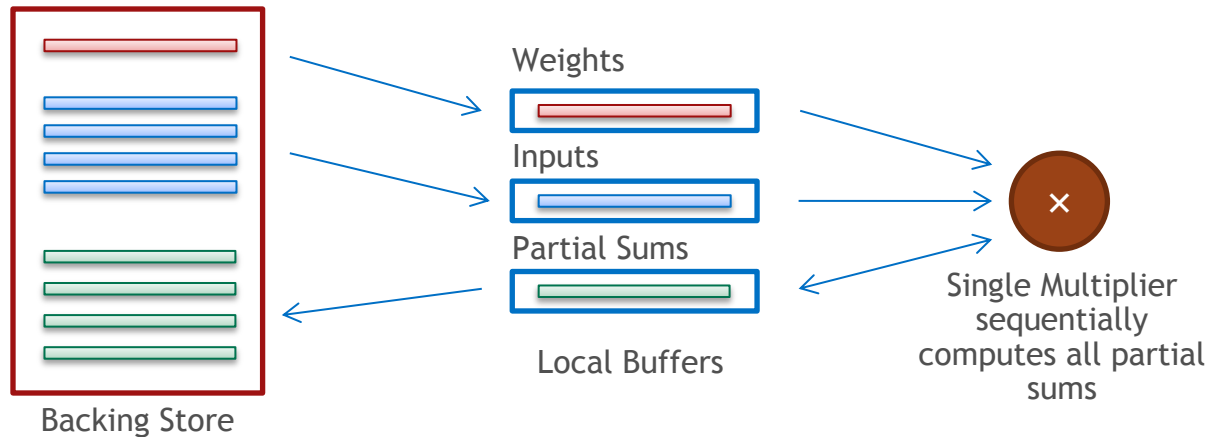
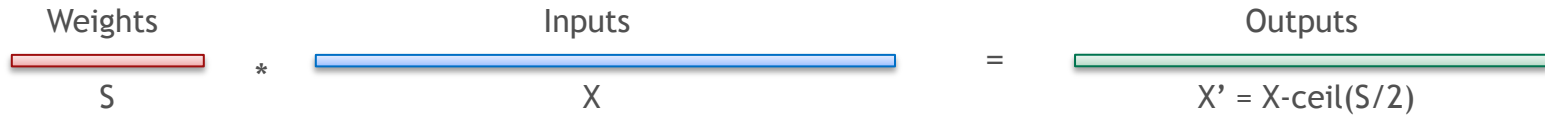
outermost loop is S rank

WS Dataflow Implementation



- **Minimize weight** read energy consumption
 - maximize convolutional and filter reuse of weights
- **Broadcast activations** and **accumulate psums spatially** across the PE array.

Simple Model for Mapping Dataflows to HW



Common metric	Weights	Inputs	Outputs / Partial Sums
Alg. Min. accesses to backing store (MINALG)	S	X	X'
Maximum operand uses (MAXOP)	SX'	SX'	SX'

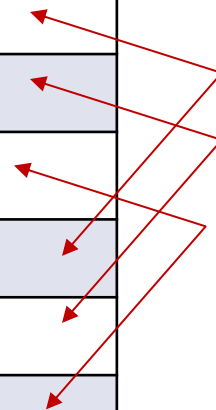
1D Convolution Summary

Hardware Structure	Per Data Type	OS Dataflow Implication	WS Dataflow Implication
Bandwidth to MAC	Weight Fetch Rate	Every Cycle	Every S Cycles
	Input Fetch Rate	Every Cycle	Every Cycle
	Output Fetch Rate	Every S Cycles	Every Cycle
Local Buffer Sizes for Temporal Reuse	Weight Buffer Size	S	1
	Input Buffer Size	S	X'
	Output Buffer Size	1	X'
Total Local Buffer Accesses	Weight Buffer	X'	SX'
	Input Buffer	X'	S
	Output Buffer	SX'	S

1D Convolution Summary

Hardware Structure	Per Data Type	OS Dataflow Implication	WS Dataflow Implication
Bandwidth to MAC	Weight Fetch Rate	Every Cycle	Every S Cycles
	Input Fetch Rate	Every Cycle	Every Cycle
	Output Fetch Rate	Every S Cycles	Every Cycle
Local Buffer Sizes for Temporal Reuse	Weight Buffer Size	S	1
	Input Buffer Size	S	X'
	Output Buffer Size	1	X'
Total Local Buffer Accesses	Weight Buffer	X'	SX'
	Input Buffer	X'	S
	Output Buffer	SX'	S

Why is product always SX'?



1D Convolution Summary

Hardware Structure	Per Data Type	OS Dataflow Implication	WS Dataflow Implication
Bandwidth to MAC	Weight Fetch Rate	Every Cycle	Every S Cycles
	Input Fetch Rate	Every Cycle	Every Cycle
	Output Fetch Rate	Every S Cycles	Every Cycle
Local Buffer Sizes for Temporal Reuse	Weight Buffer Size	S	1
	Input Buffer Size	S	X'
	Output Buffer Size	1	X'
Total Local Buffer Accesses	Weight Buffer	X'	SX'
	Input Buffer	X'	S
	Output Buffer	SX'	S

Why is product always SX'?

Total computations same

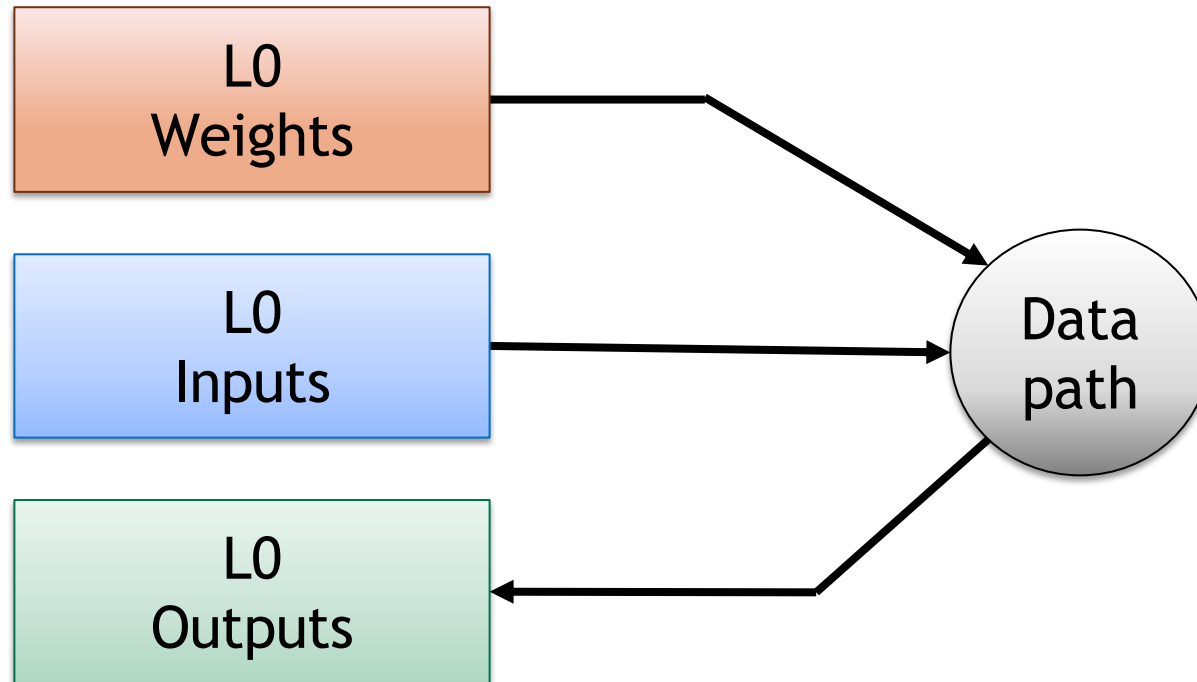
Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
 - Multi-layer Buffering
 - Multiple PEs
 - Full Convolution
- Advanced Dataflows

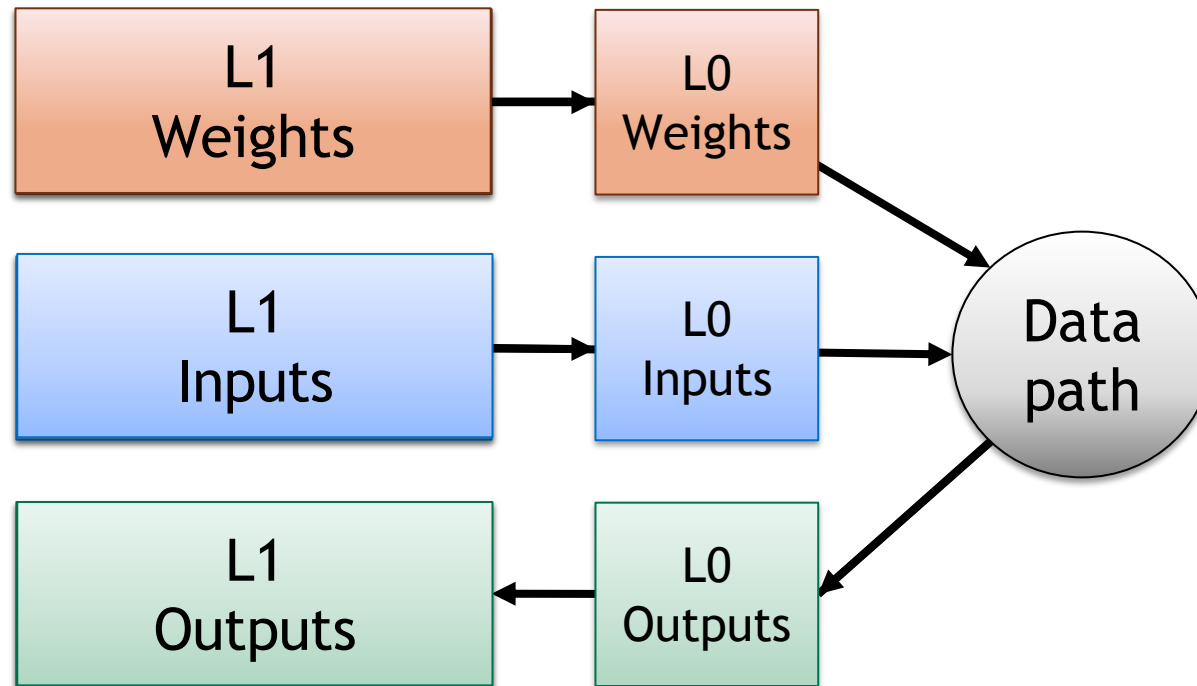
Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
 - Multi-layer Buffering
 - Multiple PEs
 - Full Convolution
- Advanced Dataflows

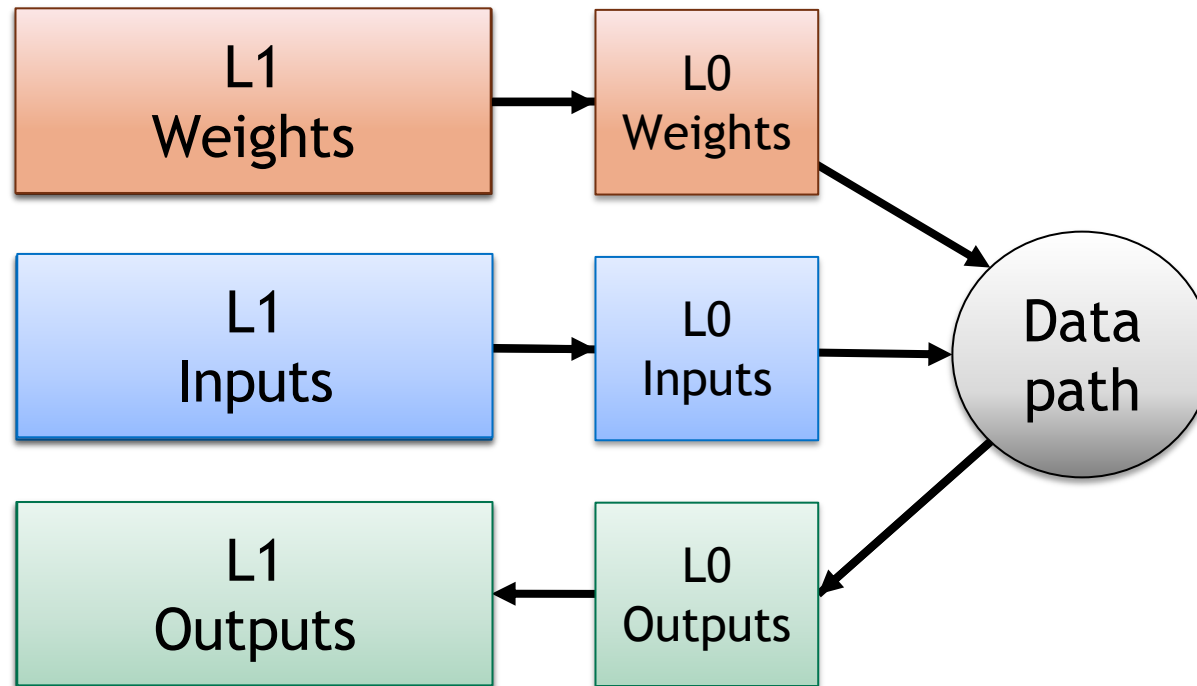
Multi-layer Buffering



Multi-layer Buffering

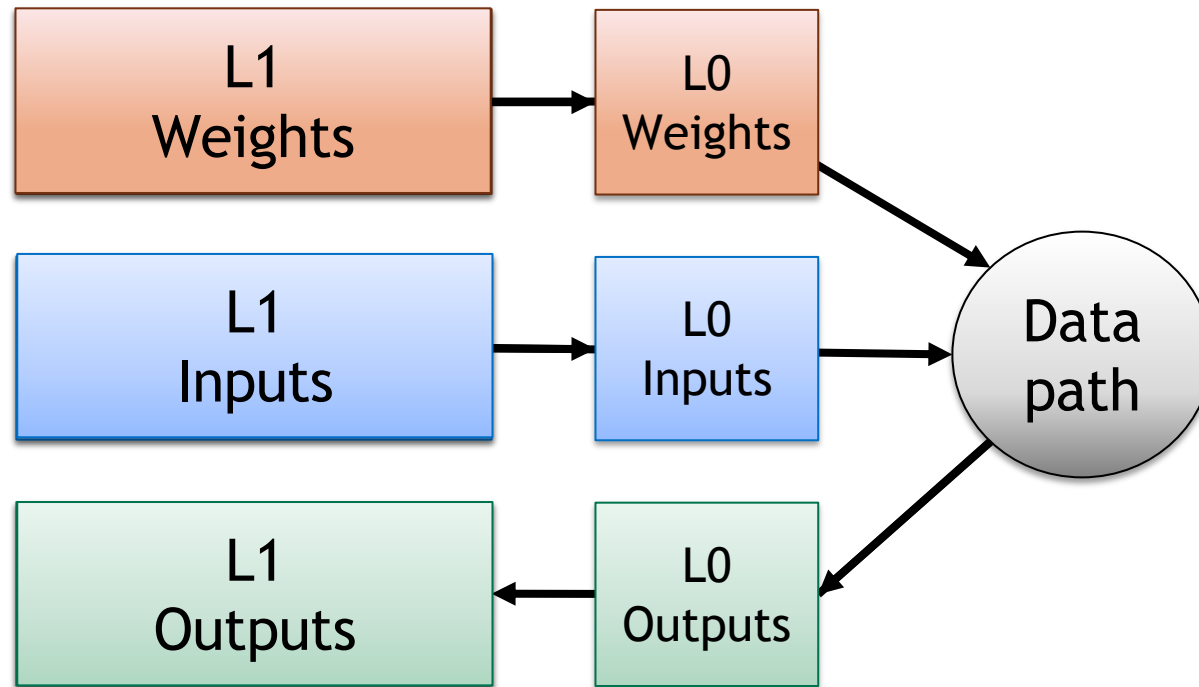


Multi-layer Buffering



How will this be reflected
in the loop nest?

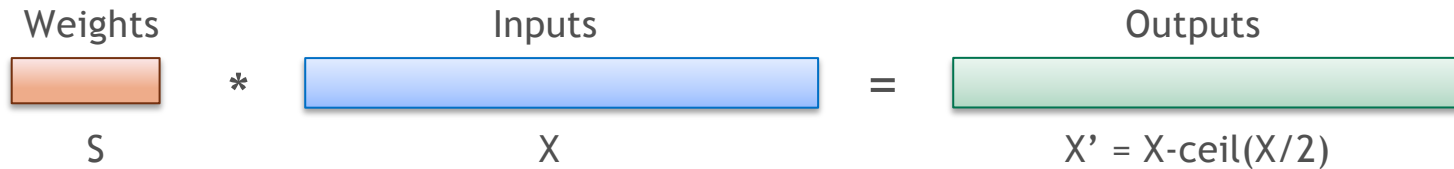
Multi-layer Buffering



How will this be reflected
in the loop nest?

New 'level' of loops

1D Convolution – “Tiled”



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations
```

```
// Level 1
```

```
for (x1 = 0; x1 < X'1; x1++) {
    for (s1 = 0; s1 < S1; s1++) {
        // Level 0
        for (x0 = 0; x0 < X'0; x0++) {
            for (s0 = 0; s0 < S0; s0++) {
                x = x1 * X'0 + x0;
                s = r1 * R0 + r0;
                o[x] += i[x+s] * w[s];
            }
        }
    }
}
```

Note X' and S are factored so:
 $X'_0 * X'_1 = X'$
 $S_0 * S_1 = S$

Buffer sizes

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

The diagram shows the code snippet with four brackets under the innermost loop's index expression: `o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];`. The brackets are under `x1*X'0+x0`, `s1*S0+s0`, `s1*S0+s0`, and `s1*S0+s0`. These four brackets are connected by a single horizontal line, which is then connected by a vertical line to a rectangular box below the code containing the text "Constant over each level 1 iteration".

Constant over each level 1 iteration

- Level 0 buffer size is volume needed in each Level 1 iteration.
- Level 1 buffer size is volume needed to be preserved and re-delivered in future (usually successive) Level 1 iterations.
- A **legal mapping** will fit into the hardware's buffer sizes

Buffer sizes

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

Constant over each level 1 iteration

	Level 0	Level 1
Weights		
Inputs		
Outputs		

Buffer sizes

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

Constant over each level 1 iteration

	Level 0	Level 1
Weights	S0	
Inputs		
Outputs		

Buffer sizes

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

Constant over each level 1 iteration

	Level 0	Level 1
Weights	S0	
Inputs	X'0+S0	
Outputs		

Buffer sizes

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

Constant over each level 1 iteration

	Level 0	Level 1
Weights	S0	
Inputs	X'0+S0	
Outputs	X'0	

Buffer sizes

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

Constant over each level 1 iteration

	Level 0	Level 1
Weights	S0	S
Inputs	X'0+S0	
Outputs	X'0	

Buffer sizes

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

Constant over each level 1 iteration

	Level 0	Level 1
Weights	S0	S
Inputs	X'0+S0	
Outputs	X'0	1

Buffer sizes

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

Constant over each level 1 iteration

	Level 0	Level 1
Weights	S0	S
Inputs	X'0+S0	S
Outputs	X'0	1

Energy Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

Constant over each level 1 iteration

Energy of a buffer access is a function of the size of the buffer

Energy Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

Constant over each level 1 iteration

Energy of a buffer access is a function of the size of the buffer

Each buffer level's energy is proportional the number of accesses at that level

Energy Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

Constant over each level 1 iteration

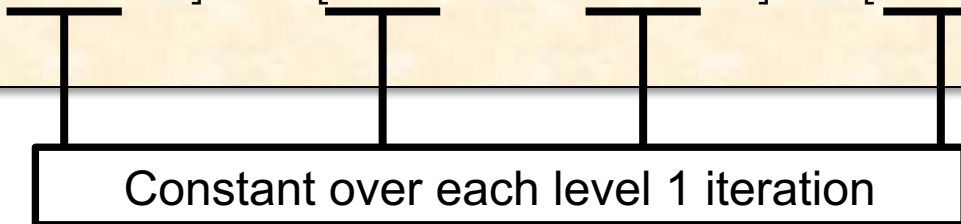
Energy of a buffer access is a function of the size of the buffer

Each buffer level's energy is proportional the number of accesses at that level

For level 0 that is all the operands to the Datapath

Energy Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```



Energy of a buffer access is a function of the size of the buffer

Each buffer level's energy is proportional the number of accesses at that level

For level 0 that is all the operands to the Datapath

For level $L > 0$ there are three components:

Energy Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

Constant over each level 1 iteration

Energy of a buffer access is a function of the size of the buffer

Each buffer level's energy is proportional the number of accesses at that level

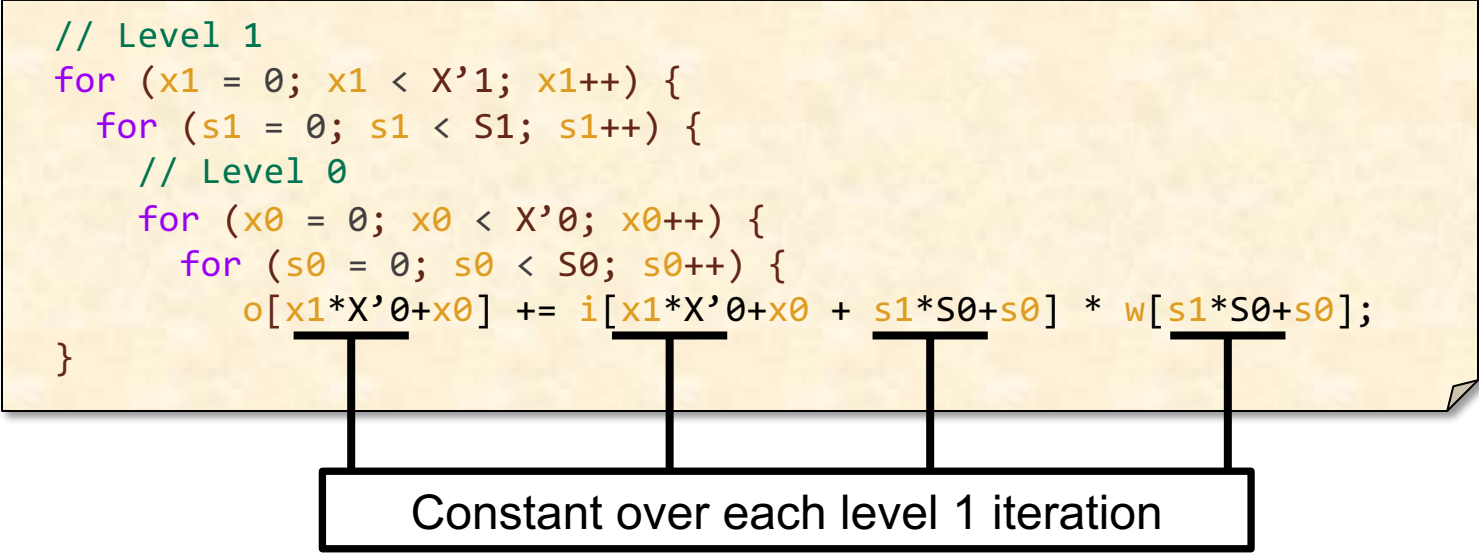
For level 0 that is all the operands to the Datapath

For level $L > 0$ there are three components:

Data arriving from level $L+1$

Energy Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```



Constant over each level 1 iteration

Energy of a buffer access is a function of the size of the buffer

Each buffer level's energy is proportional the number of accesses at that level

For level 0 that is all the operands to the Datapath

For level $L > 0$ there are three components:

Data arriving from level $L+1$

Data that needs to be transferred to level $L-1$

Energy Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

The diagram shows the code snippet with four brackets under the indices of the array access: `x1*X'0+x0`, `x1*X'0+x0 + s1*S0+s0`, `s1*S0+s0`, and `s1*S0+s0`. These brackets are connected by a single horizontal line that points to a box labeled "Constant over each level 1 iteration".

Constant over each level 1 iteration

Energy of a buffer access is a function of the size of the buffer

Each buffer level's energy is proportional the number of accesses at that level

For level 0 that is all the operands to the Datapath

For level $L > 0$ there are three components:

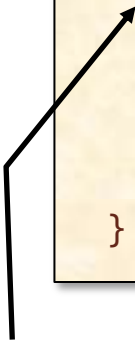
Data arriving from level $L+1$

Data that needs to be transferred to level $L-1$

Data that is returned from level $L-1$

Mapping – Weight Access Costs

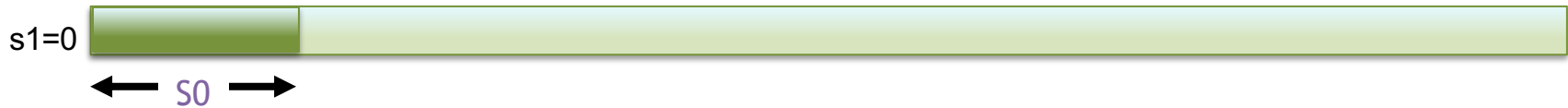
```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S'0+s0];
      }
    }
  }
}
```



Mapping – Weight Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S'0+s0];
      }
    }
  }
}
```

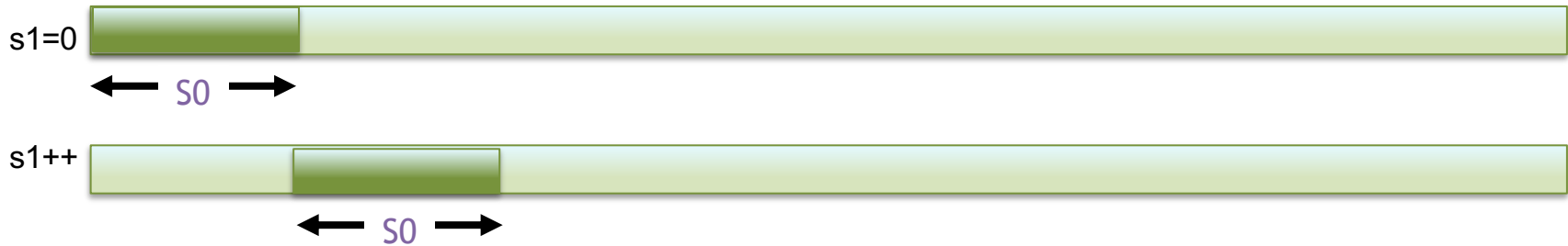
Weights



Mapping – Weight Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S'0+s0];
      }
    }
  }
}
```

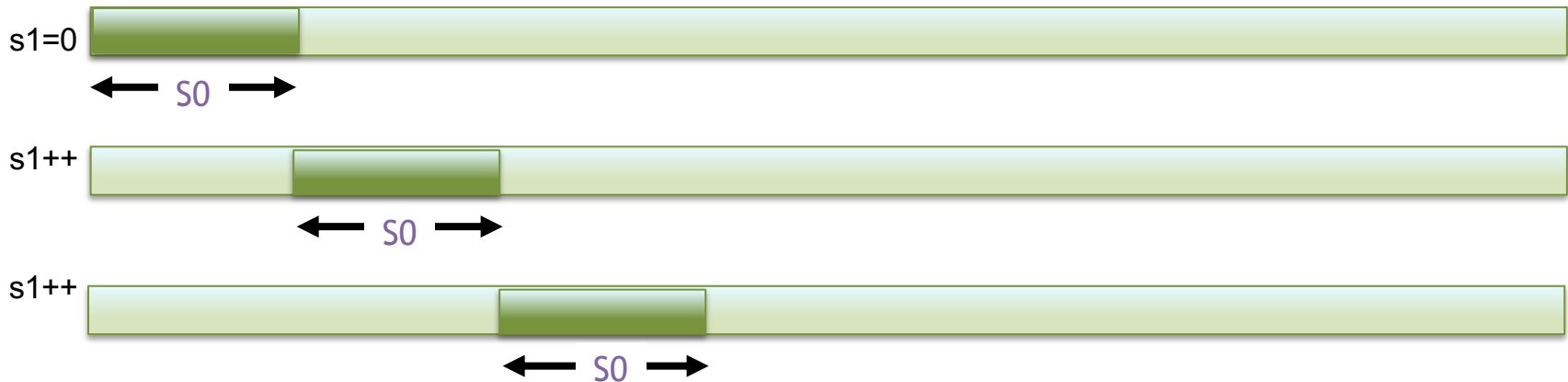
Weights



Mapping – Weight Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S'0+s0];
      }
    }
  }
}
```

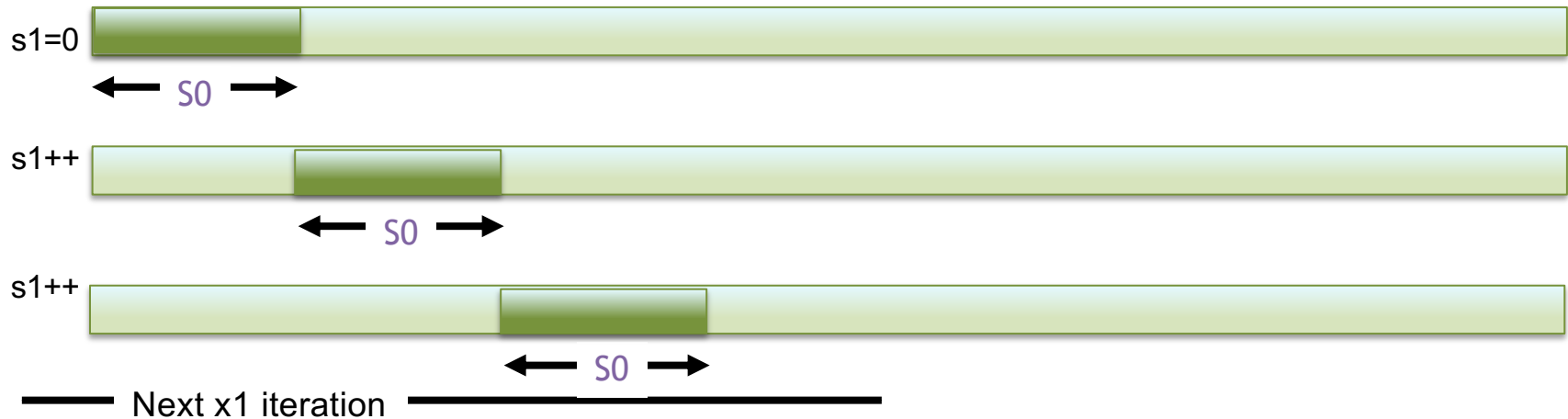
Weights



Mapping – Weight Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S'0+s0];
      }
    }
  }
}
```

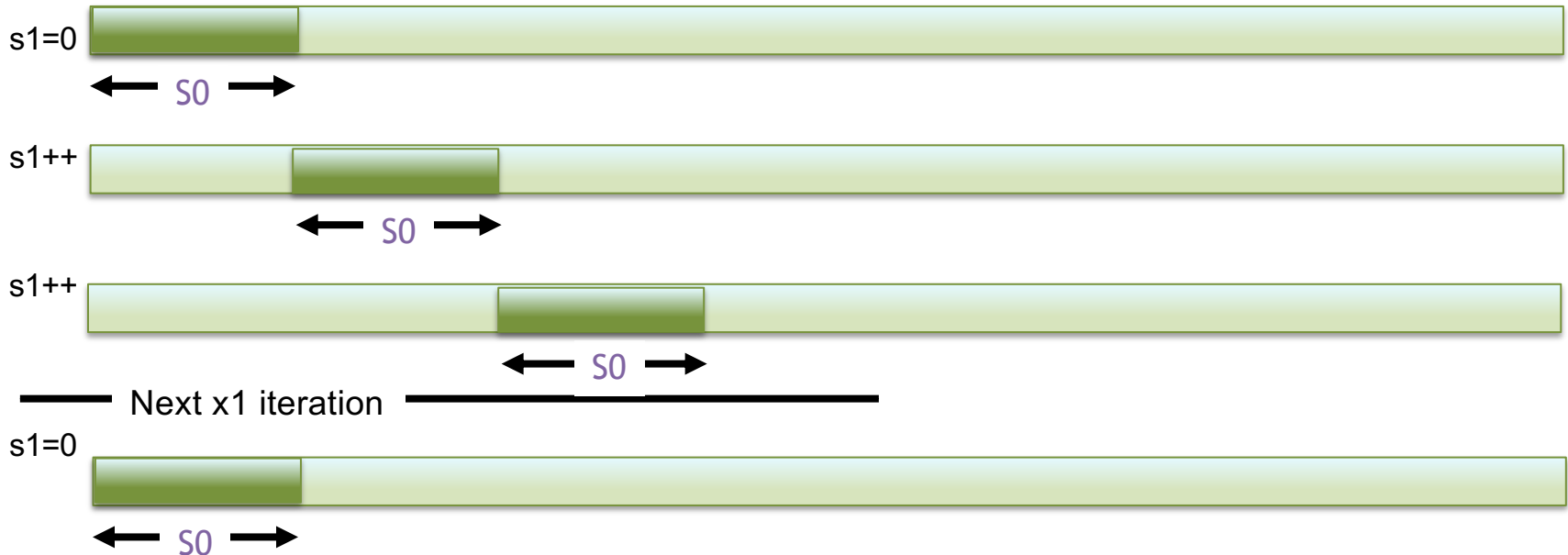
Weights



Mapping – Weight Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0] * w[s1*S'0+s0];
      }
    }
  }
}
```

Weights



Mapping – Weight Access Costs

- Level 0 reads

- Per level 1 iteration -> $X'0 * S0$ weight reads
- Times $X'1 * S1$ level 1 iterations
- Total reads = $(X'0 * S0) * (X'1 * S1) = (X'0 * X'1) * (S0 * S1) = SX'$ reads

- Level 1 to 0 transfers

- Per level 1 iteration -> $S0$ weights transferred
- Times same number of level 1 iterations = $X'1 * S1$
- Total transfers -> $S0 * (X'1 * S1) = X'1 * (S0 * S1) = SX'1$

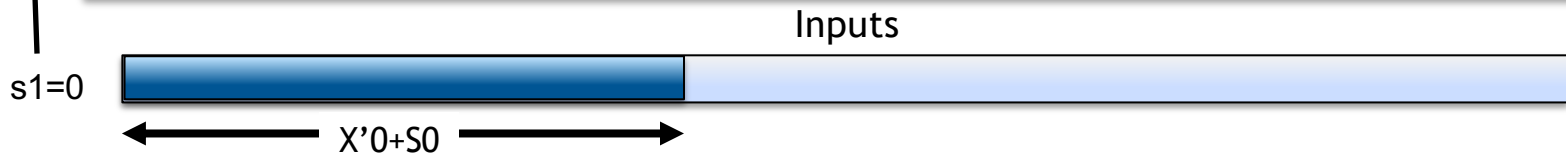
Mapping – Weight Access Costs

- Level 0 reads
 - Per level 1 iteration $\rightarrow X'0 * S0$ weight reads
 - Times $X'1 * S1$ level 1 iterations
 - Total reads = $(X'0 * S0) * (X'1 * S1) = (X'0 * X'1) * (S0 * S1) = SX'$ reads
- Level 1 to 0 transfers
 - Per level 1 iteration $\rightarrow S0$ weights transferred
 - Times same number of level 1 iterations = $X'1 * S1$
 - Total transfers $\rightarrow S0 * (X'1 * S1) = X'1 * (S0 * S1) = SX'1$

Disjoint/partitioned reuse pattern

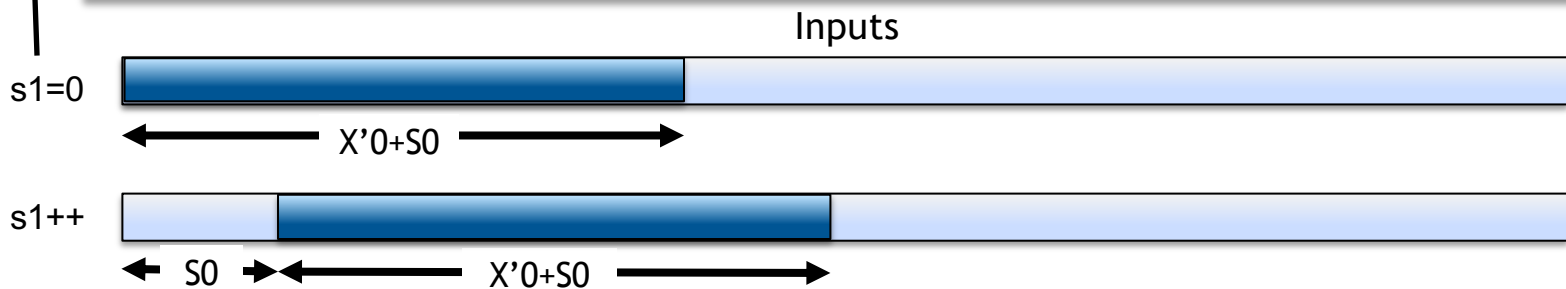
Mapping – Input Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```



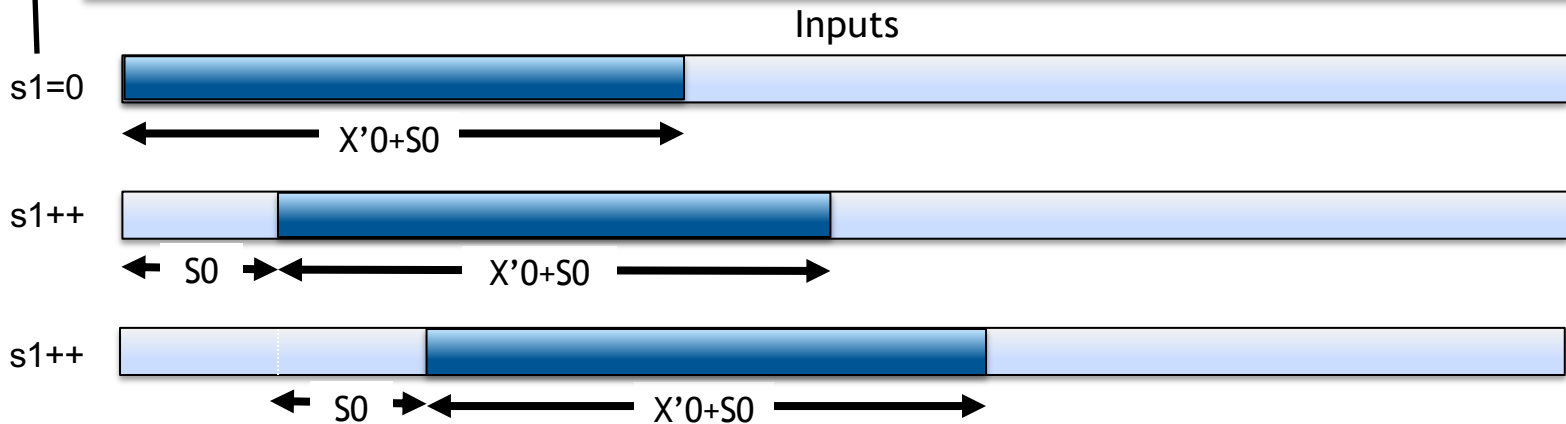
Mapping – Input Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```



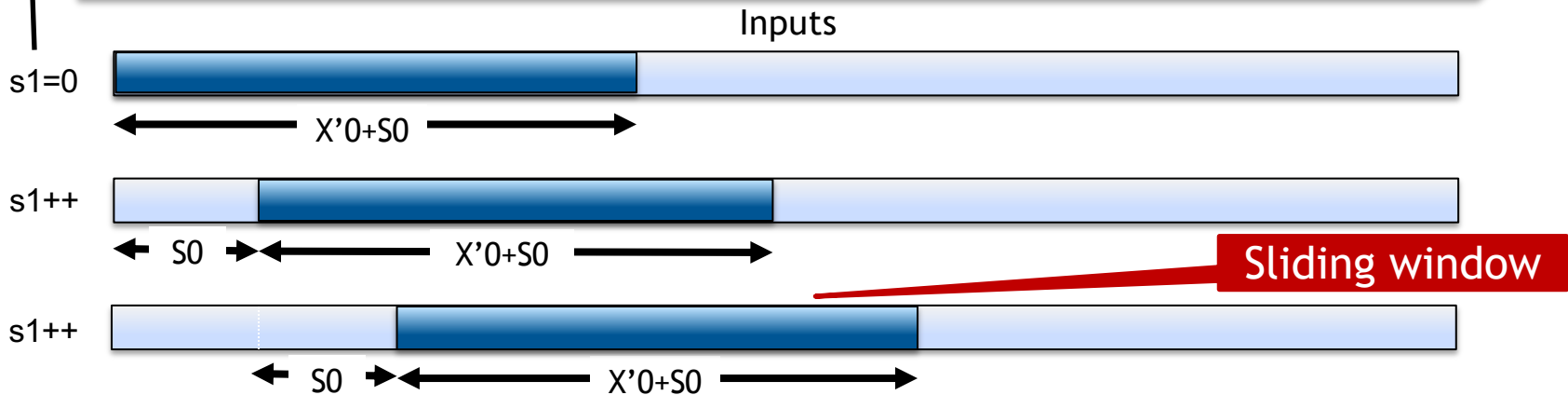
Mapping – Input Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```



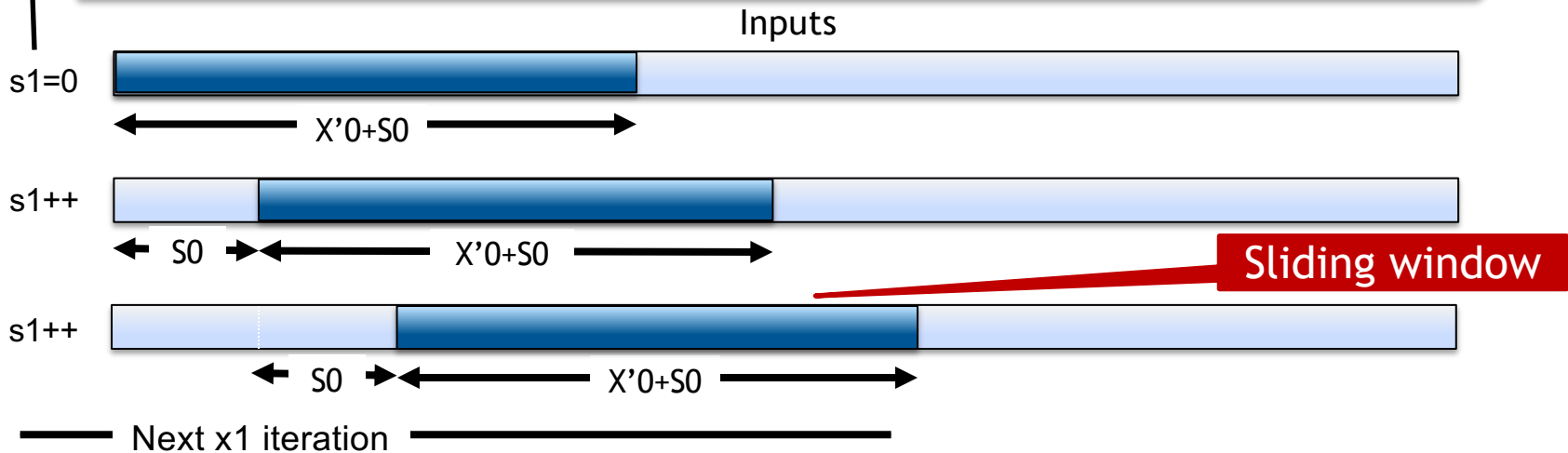
Mapping – Input Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```



Mapping – Input Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```

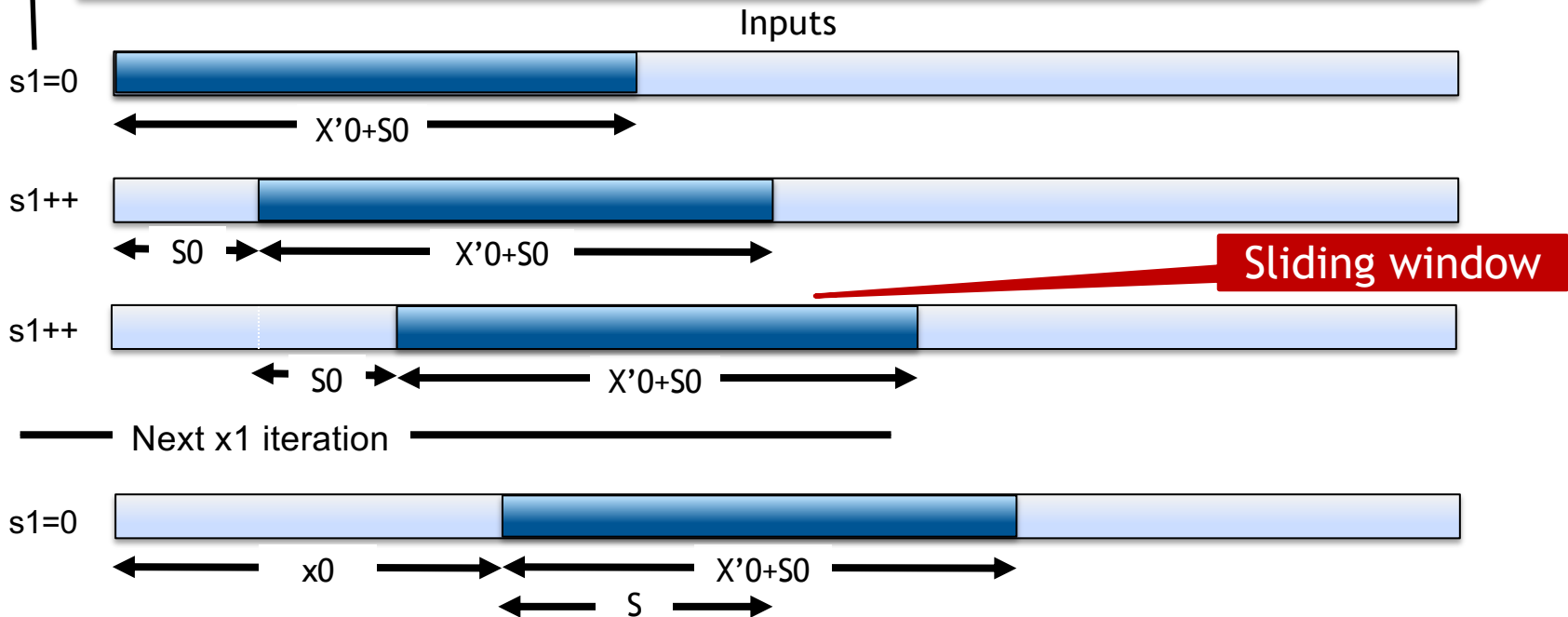


Mapping – Input Access Costs

```

// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}

```

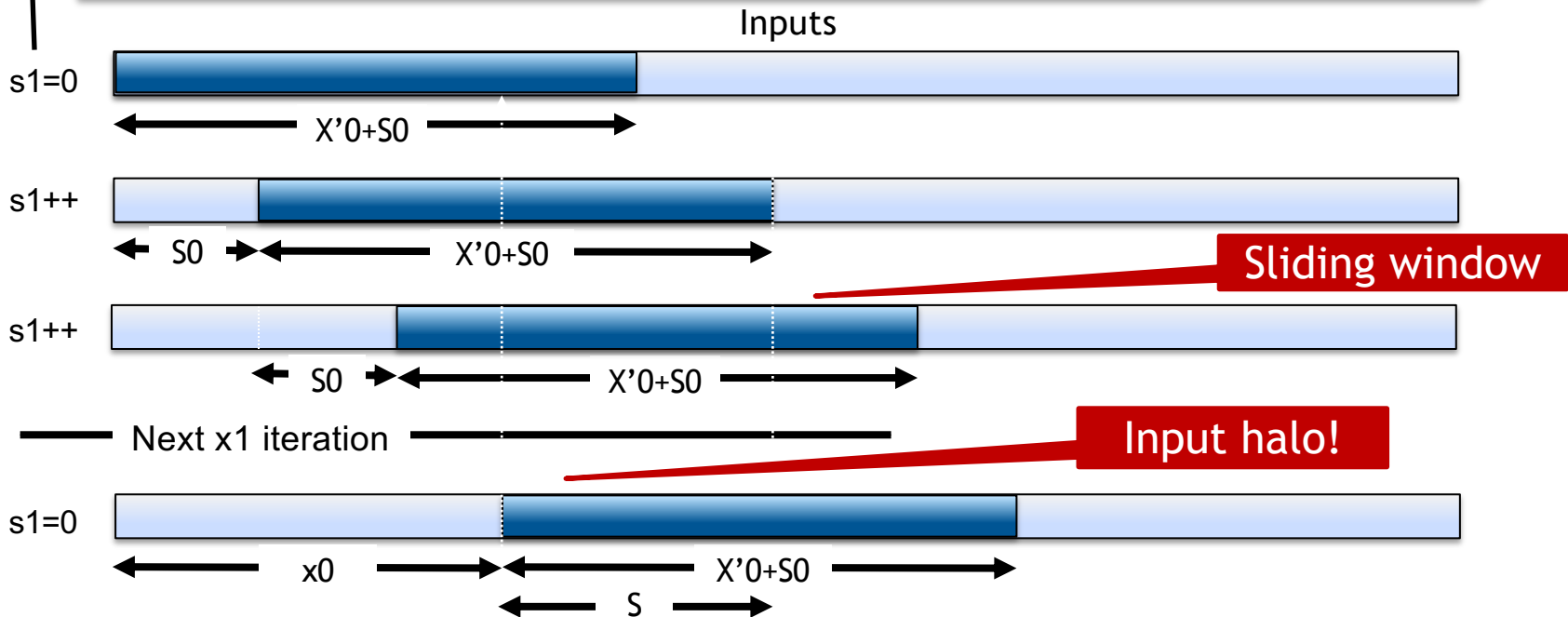


Mapping – Input Access Costs

```

// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}

```



Mapping – Input Access Costs

- Level 0 reads

- Per level 1 iteration $\rightarrow X'0+S0$ inputs reads
- Times $X'1*S1$ level 1 iterations
- Total reads = $X'1*S1*(X'0+S0) = ((X'1*X'0)*S1)+(X'1*(S1*S0))$
= $X'*S1+X'1*S$ reads

–

- Level 1 to 0 transfers

- For $s=0$, $X'0+S0$ inputs transferred
- For each of the following $S1-1$ iterations another $S0$ inputs transferred
- So total per $x1$ iteration is: $X'0+S0*S1 = X'0+S$ inputs
- Times number of $x1$ iterations = $X'1$
- So total transfers = $X'1*(X'0+S) = (X'1*X'0)+X'1*S = X'+X'1*S$

Mapping – Input Access Costs

- Level 0 reads

- Per level 1 iteration $\rightarrow X'0+S0$ inputs reads
- Times $X'1*S1$ level 1 iterations
- Total reads = $X'1*S1*(X'0+S0) = ((X'1*X'0)*S1)+(X'1*(S1*S0))$
= $X'*S1+X'1*S$ reads

–

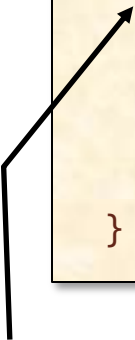
- Level 1 to 0 transfers

- For $s=0$, $X'0+S0$ inputs transferred
- For each of the following $S1-1$ iterations another $S0$ inputs transferred
- So total per $x1$ iteration is: $X'0+S0*S1 = X'0+S$ inputs
- Times number of $x1$ iterations = $X'1$
- So total transfers = $X'1*(X'0+S) = (X'1*X'0)+X'1*S = X'+X'1*S$

Sliding window/partitioned reuse pattern

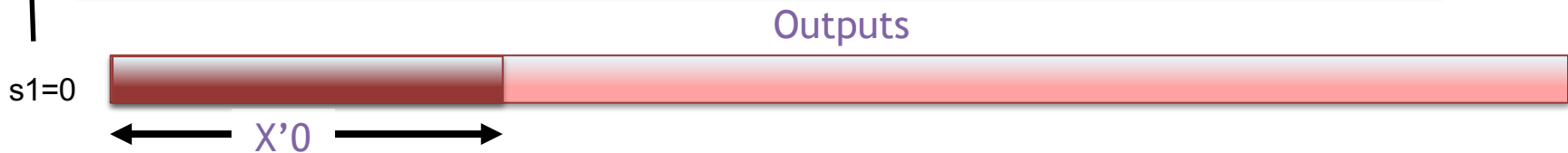
Mapping – Output Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```



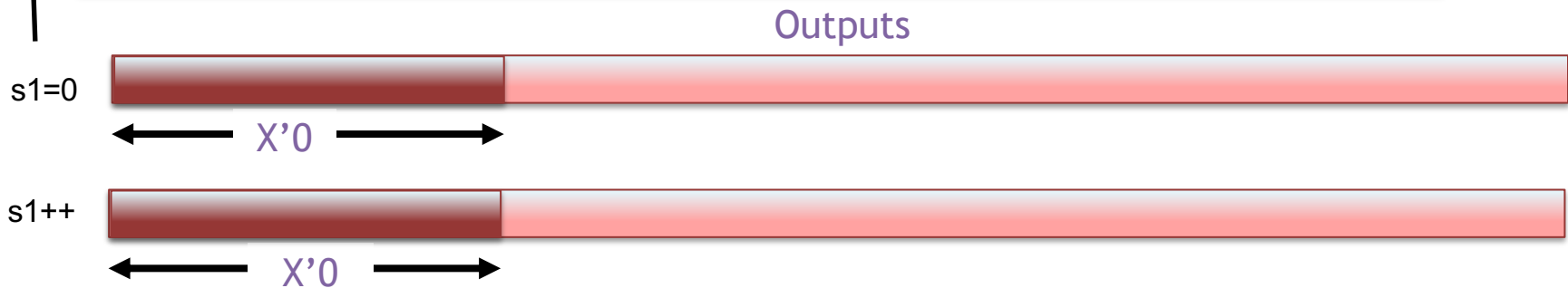
Mapping – Output Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```



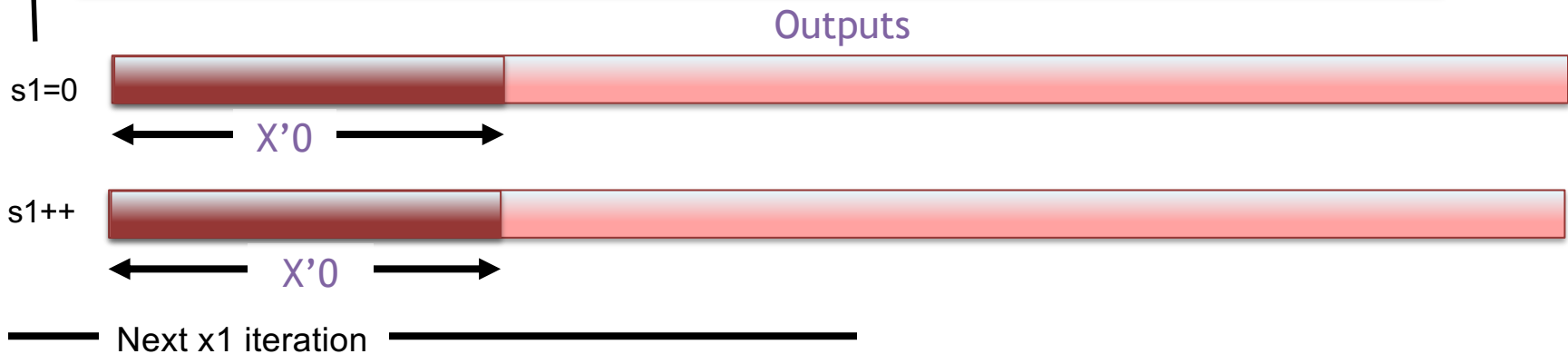
Mapping – Output Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```



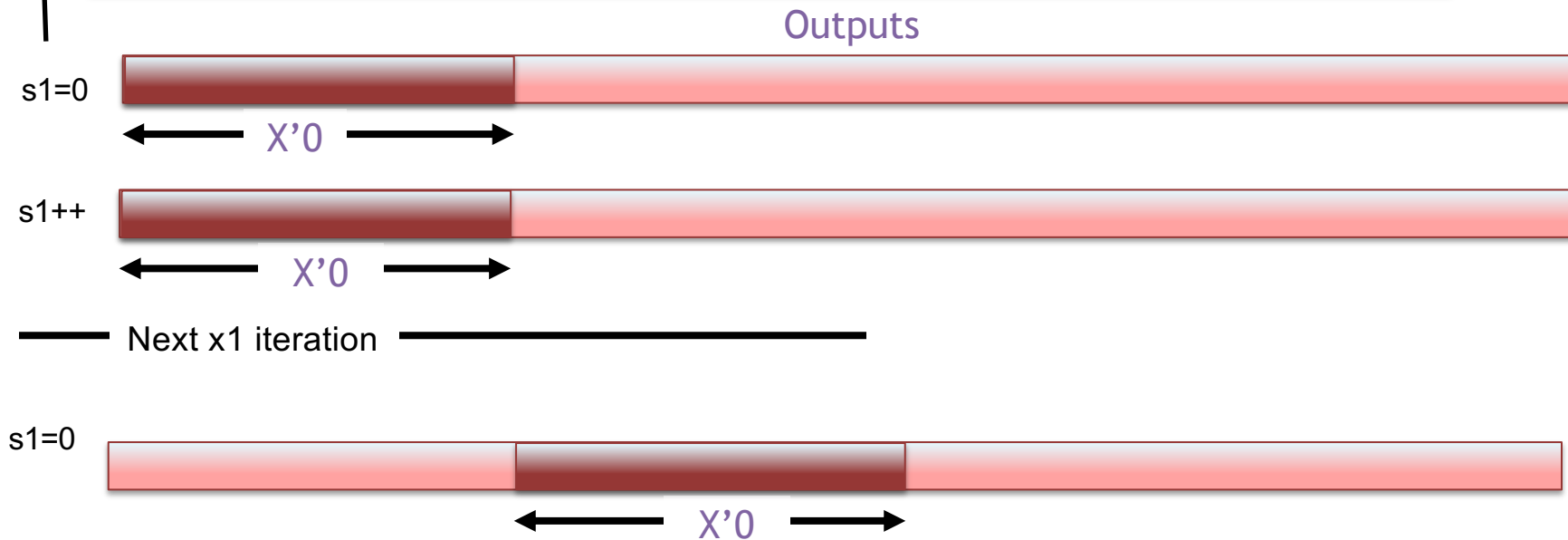
Mapping – Output Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```



Mapping – Output Access Costs

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
      }
    }
  }
}
```



Mapping – Output Access Costs

- Level 0 writes

- Due to level 0 being 'output stationary' only $X'0$ writes per level 1 iteration
- Times $X'1 * S1$ level 1 iterations
- Total writes = $X'0 * (X'1 * S1) = (X'0 * X'1) * S1 = X' * S1$ writes

–

- Level 0 to 1 transfers

- After each $S1$ iterations a completed partial sum for $X'0$ outputs are transferred
- There are $X'1$ chunks of $S1$ iterations
- So total is $X'1 * X'0 = X'$ transfers

Mapping Data Cost Summary

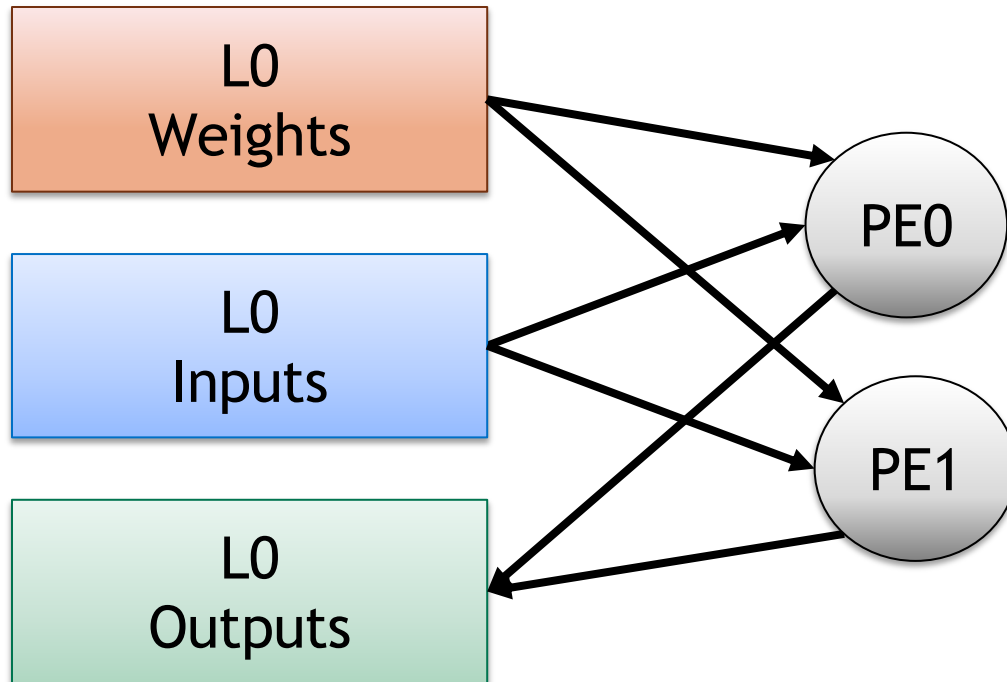
```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
  for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]* w[s1*S0+s0];
      }
    }
  }
}
```

	Level 0	Level 1 to 0 transfers
Weight Reads	SX'	$SX'1$
Input Reads	$X' * S1 + X'1 * S$	$X' + X'1 * S$
Output Reads	N/A	N/A
Output Writes	$X' * S1$	X'

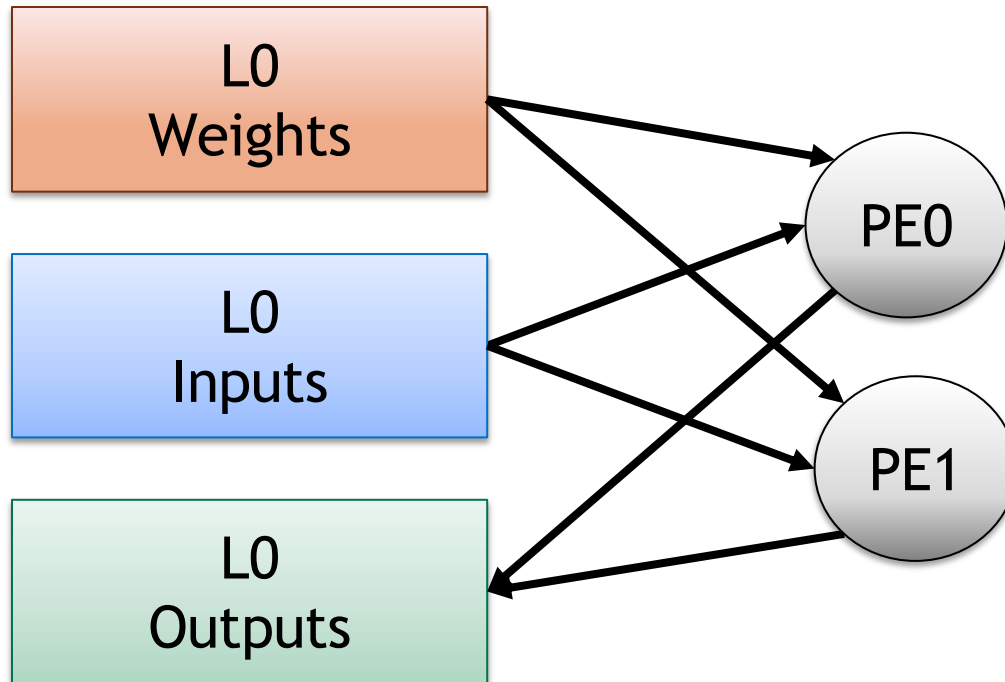
Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
 - Multi-layer Buffering
 - Multiple PEs
 - Full Convolution
- Advanced Dataflows

Spatial PEs

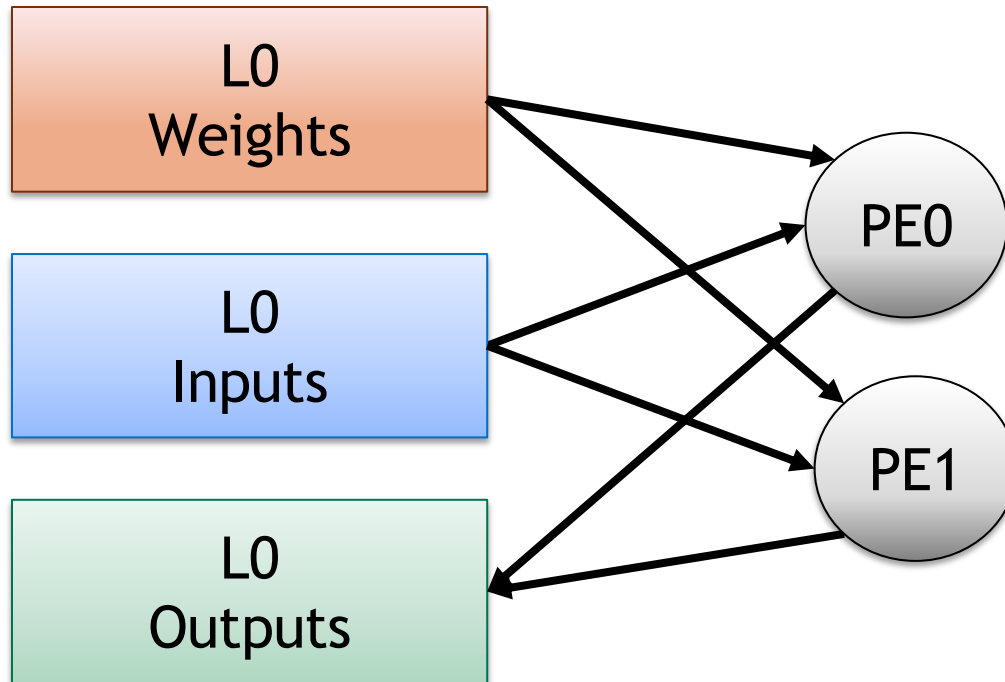


Spatial PEs



How will this be reflected
in the loop nest?

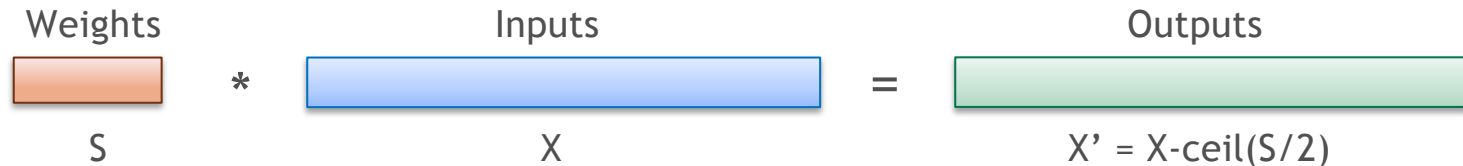
Spatial PEs



How will this be reflected
in the loop nest?

New 'level' of loops

1D Convolution – Partition Outputs

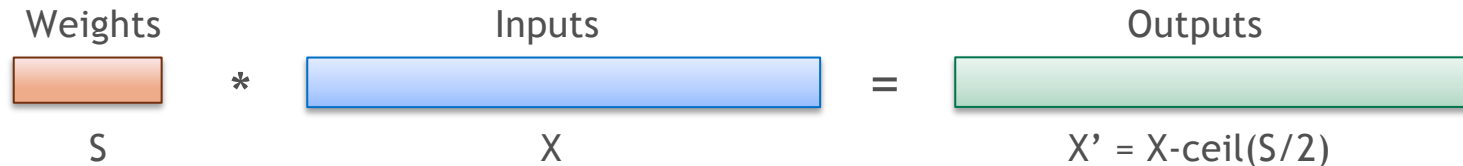


```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations
```

Note:
 $X' \cdot 0 \cdot X' \cdot 1 = X'$
 $S_0 \cdot S_1 = S$

```
// Level 1
parallel-for (x1 = 0; x1 < X'1; x1++) {
  parallel-for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                      * w[s1*S0+s0];
      }
    }
  }
}
```


1D Convolution – Partition Outputs



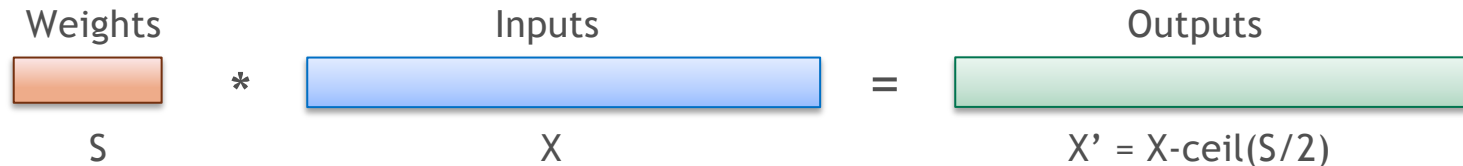
```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations
```

Note:
 $X' \cdot 0 \cdot X' \cdot 1 = X'$
 $S_0 \cdot S_1 = S$

```
// Level 1
parallel-for (x1 = 0; x1 < X'1; x1++) {
  parallel-for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                      * w[s1*S0+s0];
      }
    }
  }
}
```

$X'1 = 2$

1D Convolution – Partition Outputs



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations
```

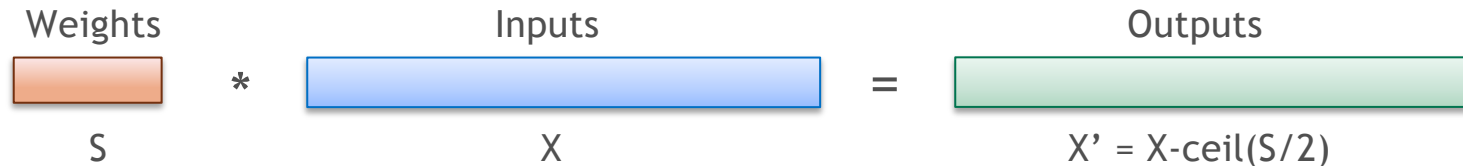
Note:
 $X' \cdot 0 \cdot X' \cdot 1 = X'$
 $S \cdot 0 \cdot S \cdot 1 = S$

```
// Level 1
parallel-for (x1 = 0; x1 < X'1; x1++) {
  parallel-for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
      for (s0 = 0; s0 < S0; s0++) {
        o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                      * w[s1*S0+s0];
      }
    }
  }
}
```

$X' \cdot 1 = 2$

$S \cdot 1 = 1 \Rightarrow s1 = 0$

1D Convolution – Partition Outputs



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations
```

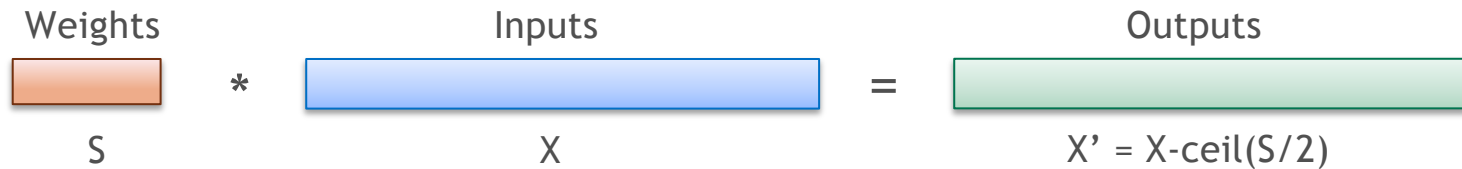
Note:
 $X' \cdot 0 \cdot X' \cdot 1 = X'$
 $S_0 \cdot S_1 = S$

```
// Level 1
parallel-for (x1 = 0; x1 < X'1; x1++) {
parallel-for (s1 = 0; s1 < S1; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
        for (s0 = 0; s0 < S0; s0++) {
            o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                          * w[s1*S0+s0];
        }
    }
}
```

$X'1 = 2$

$S1 = 1 \Rightarrow s1 = 0$

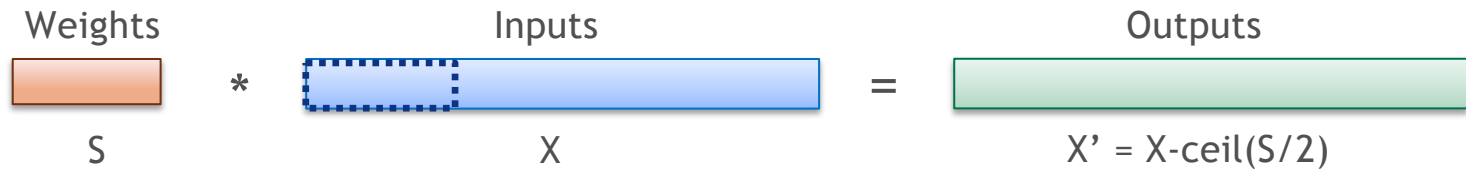
1D Convolution – Partition Outputs



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'0; x0++) {
        for (s0 = 0; s0 < S0; s0++) {
            o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                * w[s1*S0+s0];
        }
    }
}
```

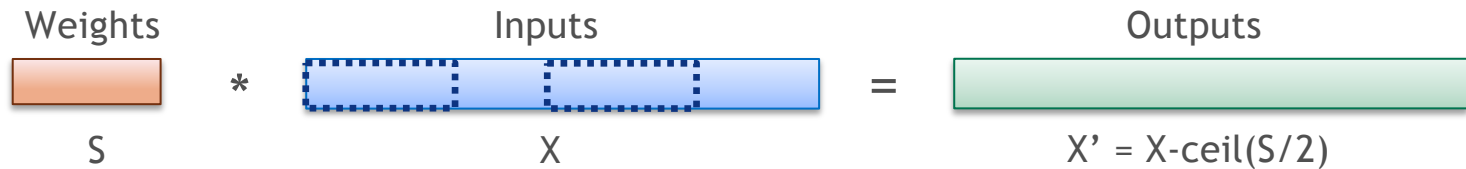
1D Convolution – Partition Outputs



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
  // Level 0
  for (x0 = 0; x0 < X'0; x0++) {
    for (s0 = 0; s0 < S0; s0++) {
      o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                    * w[s1*S0+s0];
    }
  }
}
```

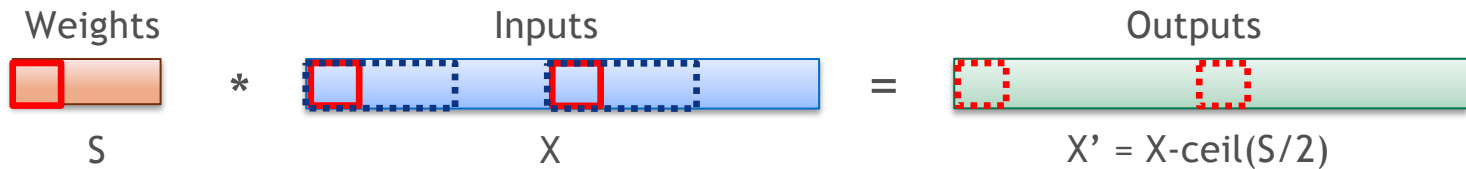
1D Convolution – Partition Outputs



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'0; x0++) {
        for (s0 = 0; s0 < S0; s0++) {
            o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                * w[s1*S0+s0];
        }
    }
}
```

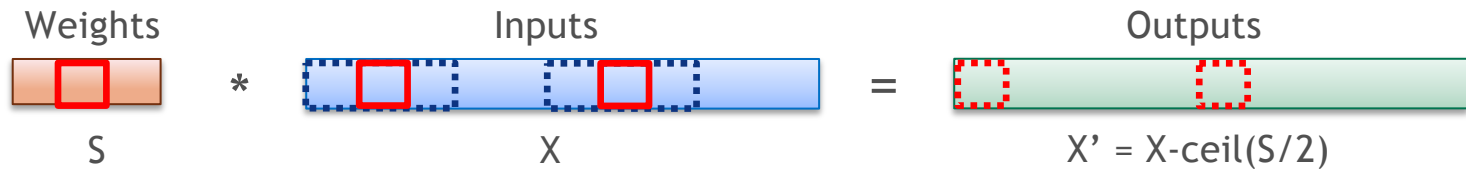
1D Convolution – Partition Outputs



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'; x0++) {
        for (s0 = 0; s0 < S; s0++) {
            o[x1*X'+x0] += i[x1*X'+x0 + s1*S0+s0]
                          * w[s1*S0+s0];
        }
    }
}
```

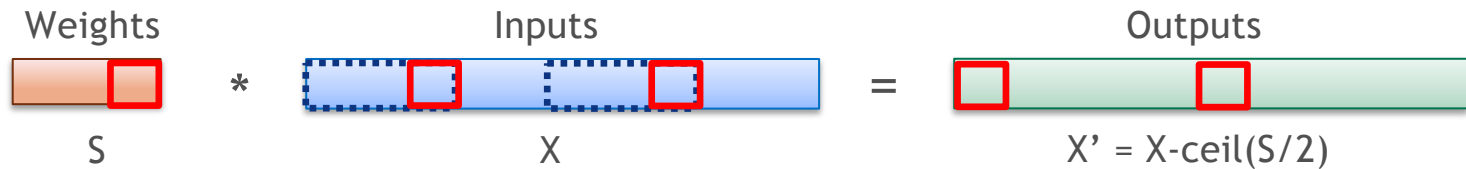
1D Convolution – Partition Outputs



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'; x0++) {
        for (s0 = 0; s0 < S; s0++) {
            o[x1*X'+x0] += i[x1*X'+x0 + s1*S0+s0]
                * w[s1*S0+s0];
        }
    }
}
```

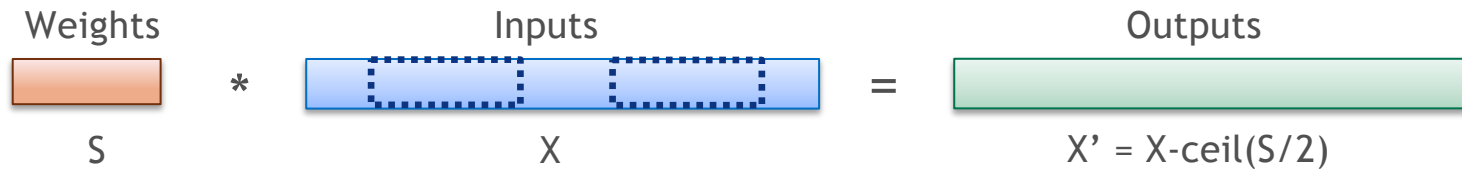

1D Convolution – Partition Outputs



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'0; x0++) {
        for (s0 = 0; s0 < S0; s0++) {
            o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                * w[s1*S0+s0];
        }
    }
}
```

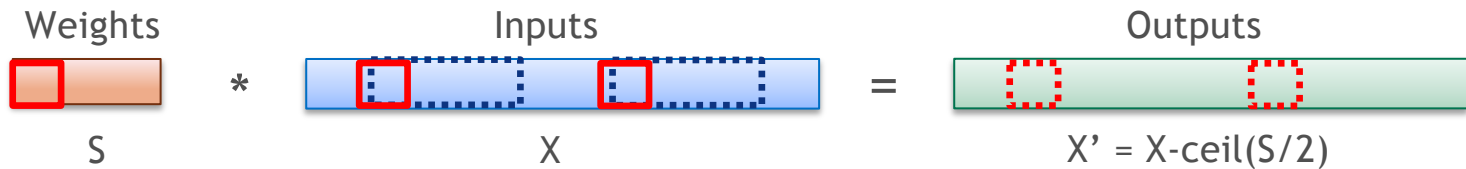
1D Convolution – Partition Outputs



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'0; x0++) {
        for (s0 = 0; s0 < S0; s0++) {
            o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                * w[s1*S0+s0];
        }
    }
}
```

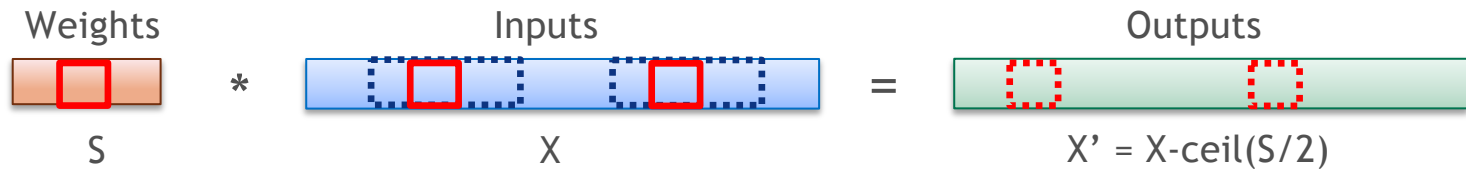
1D Convolution – Partition Outputs



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'0; x0++) {
        for (s0 = 0; s0 < S0; s0++) {
            o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                * w[s1*S0+s0];
        }
    }
}
```

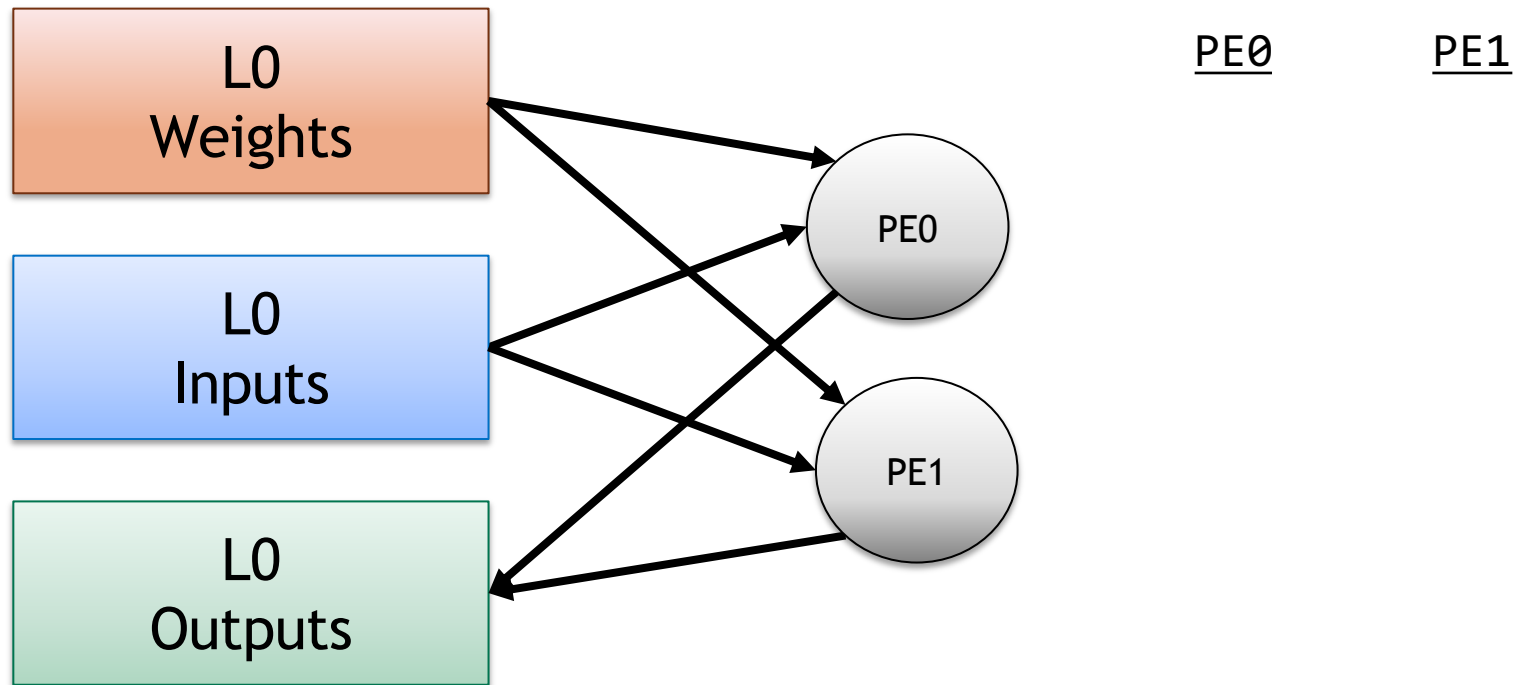
1D Convolution – Partition Outputs



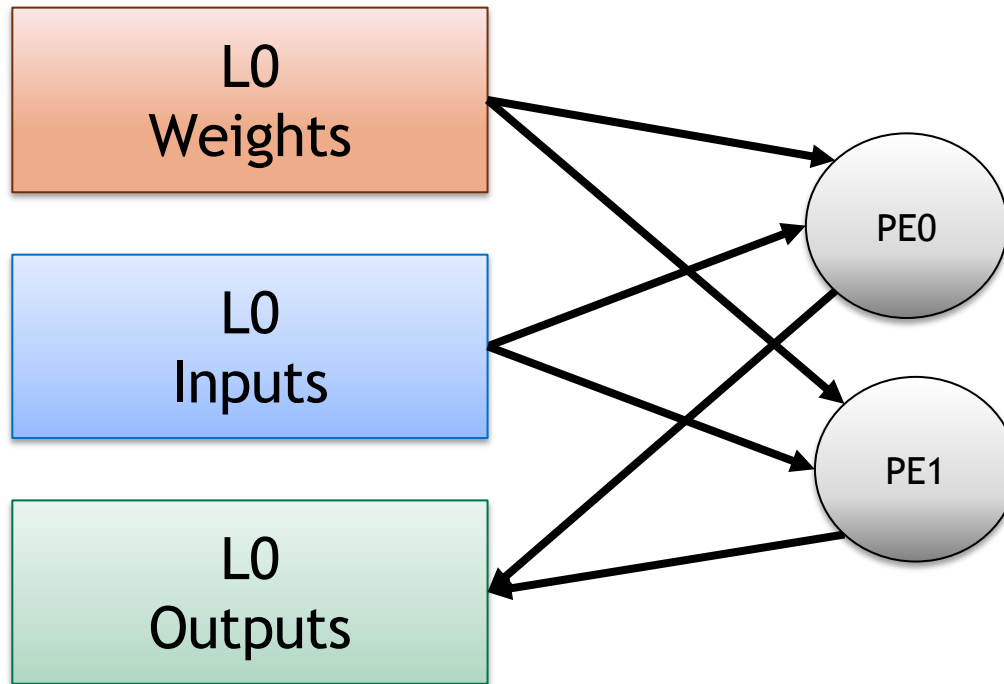
```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'0; x0++) {
        for (s0 = 0; s0 < S0; s0++) {
            o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                * w[s1*S0+s0];
        }
    }
}
```

Spatial PEs



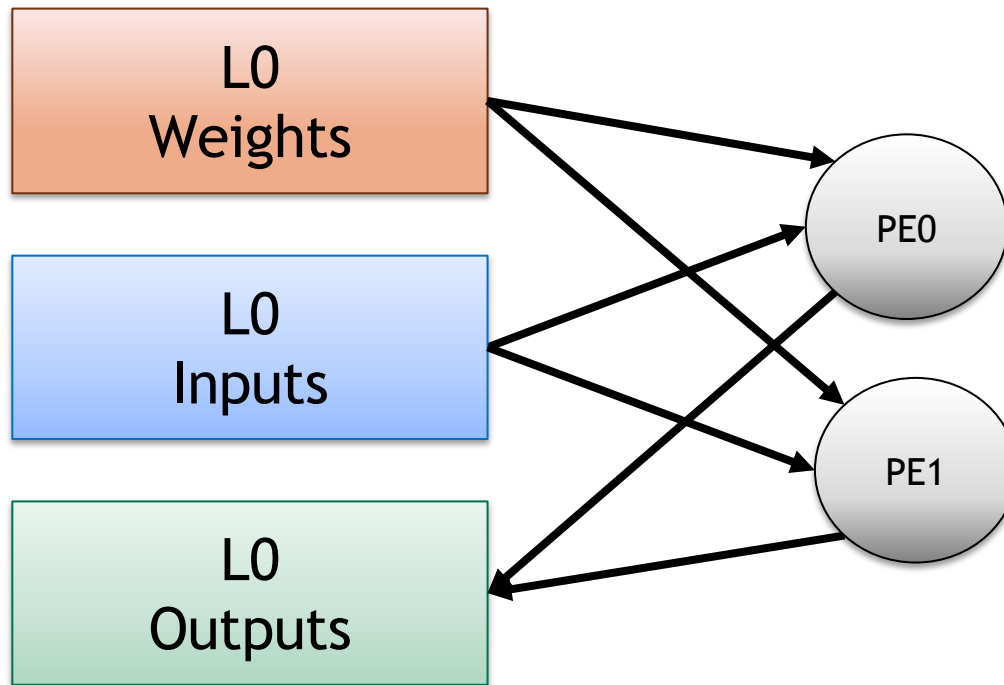
Spatial PEs



PE0
>>w[0]

PE1

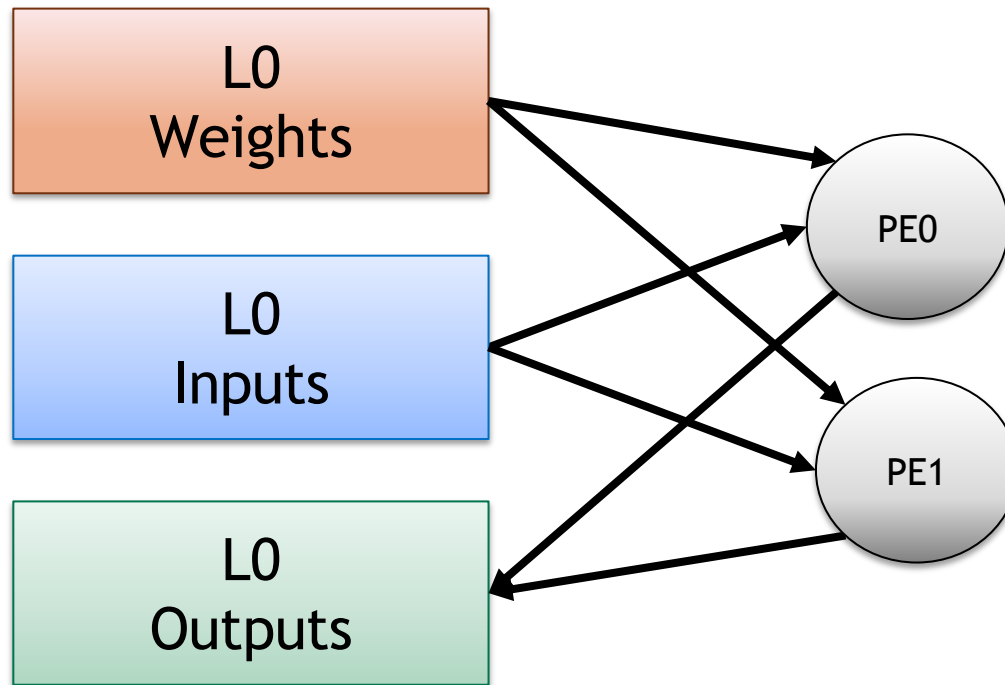
Spatial PEs



PE0
>>w[0]

PE1
>>w[0]

Spatial PEs

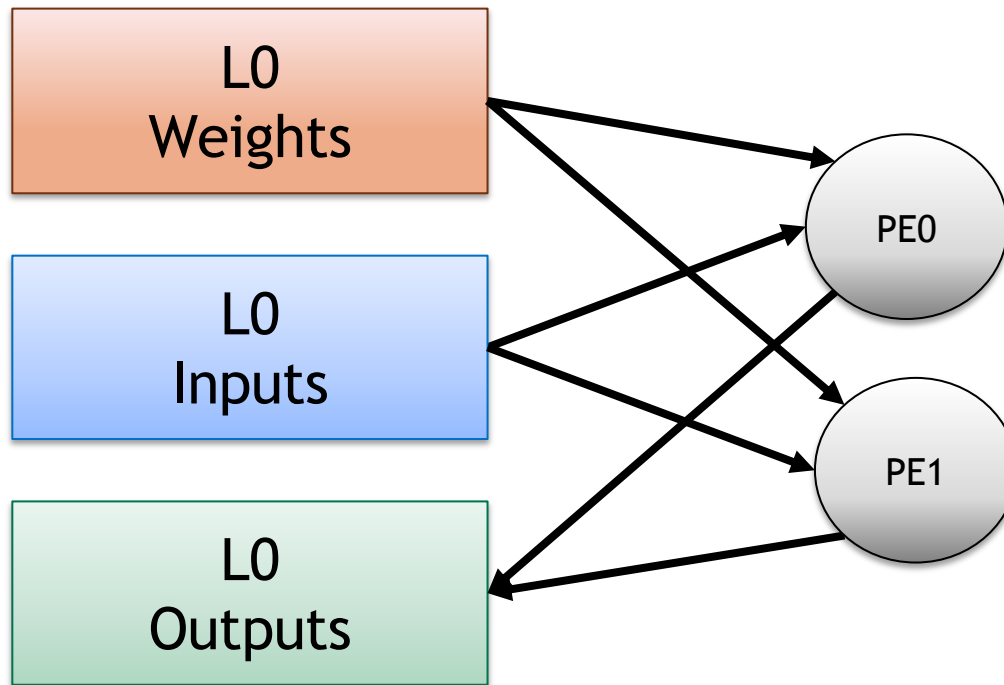


PE0
>>w[0]

PE1
>>w[0]

Implementation
opportunity?

Spatial PEs



PE0
>>w[0]

PE1
>>w[0]

Implementation
opportunity?

Yes, single fetch and
multicast

1D Convolution – Partition Outputs

```
// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'0; x0++) {
        for (s0 = 0; s0 < S0; s0++) {
            o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                          * w[s1*S0+s0];
        }
    }
}
```

1D Convolution – Partition Outputs

```
// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'0; x0++) {
        for (s0 = 0; s0 < S0; s0++) {
            o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                          * w[s1*S0+s0];
        }
    }
}
```

How do we recognize multicast opportunities?

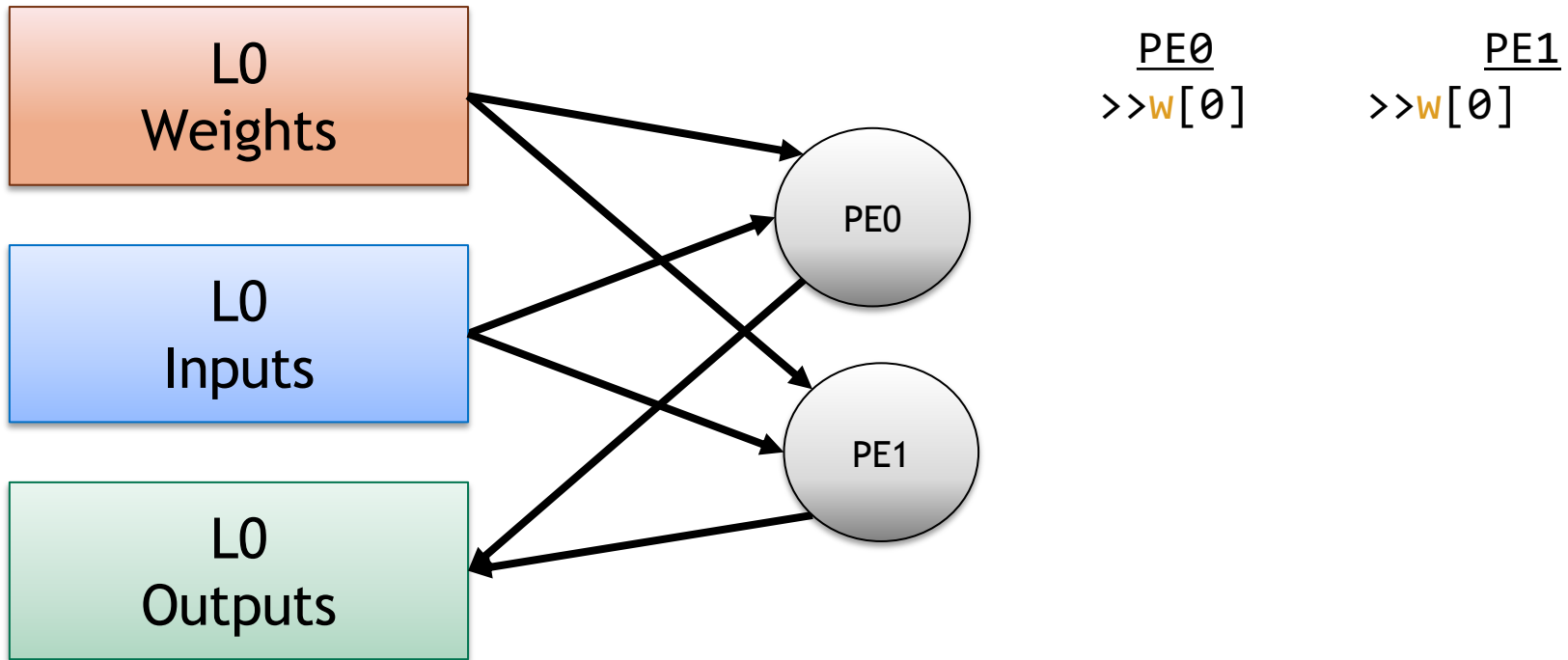
1D Convolution – Partition Outputs

```
// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
  for (x0 = 0; x0 < X'0; x0++) {
    for (s0 = 0; s0 < S0; s0++) {
      o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                    * w[s1*S0+s0];
    }
  }
}
```

How do we recognize multicast opportunities?

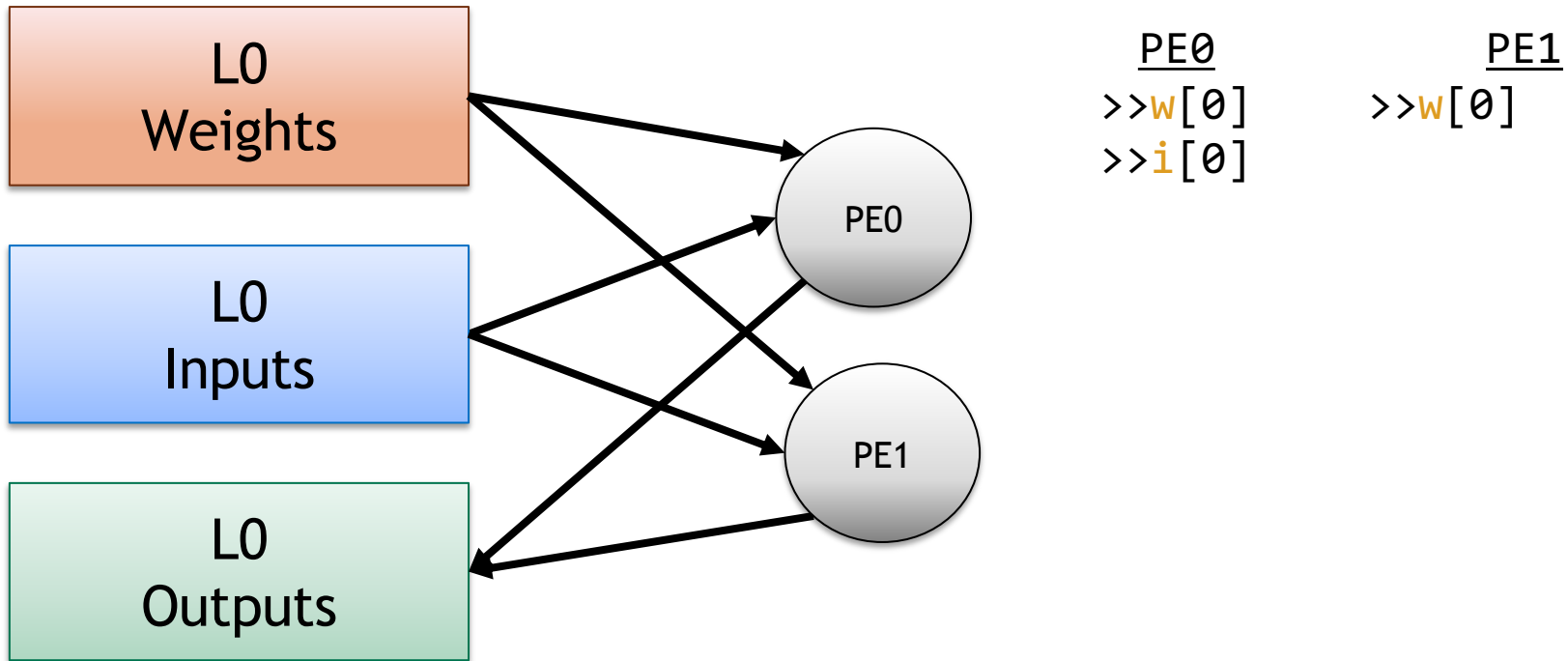
Indices independent of spatial index

Spatial PEs: Partitioned Outputs



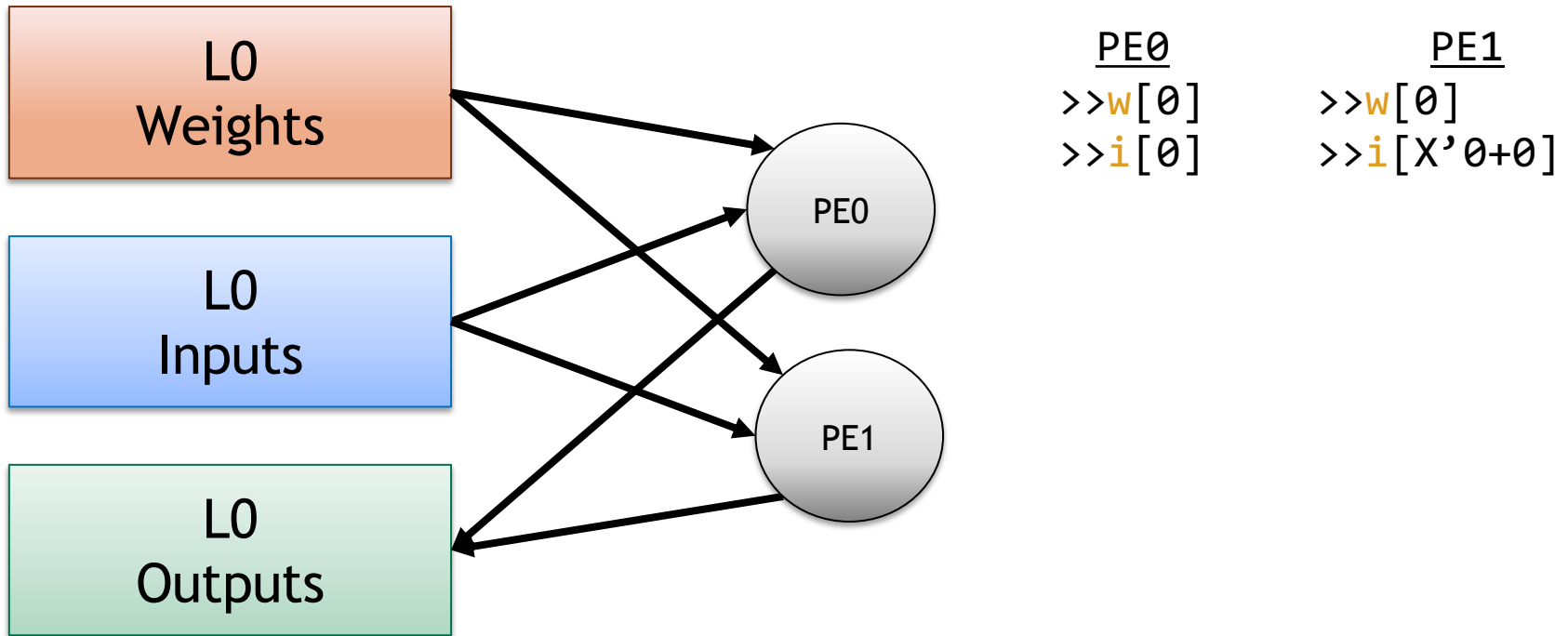
Assuming $S=3$

Spatial PEs: Partitioned Outputs



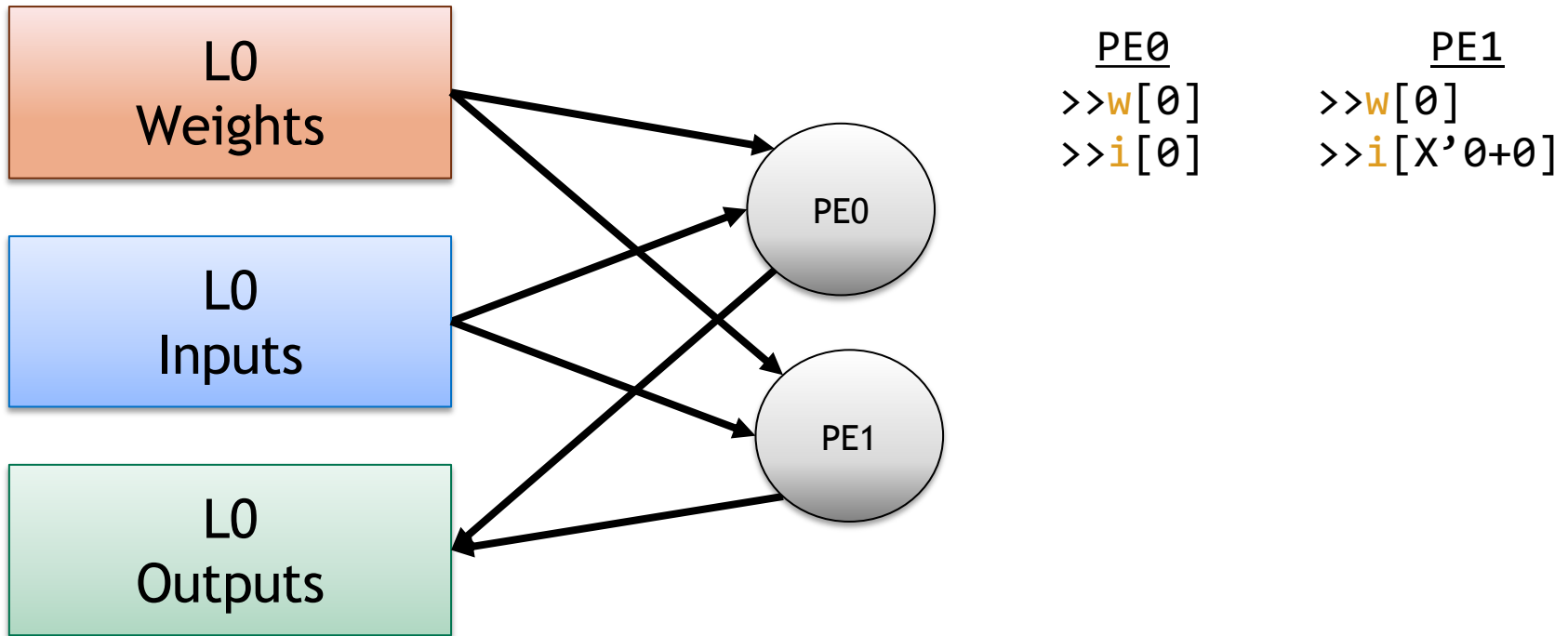
Assuming $S=3$

Spatial PEs: Partitioned Outputs



Assuming S=3

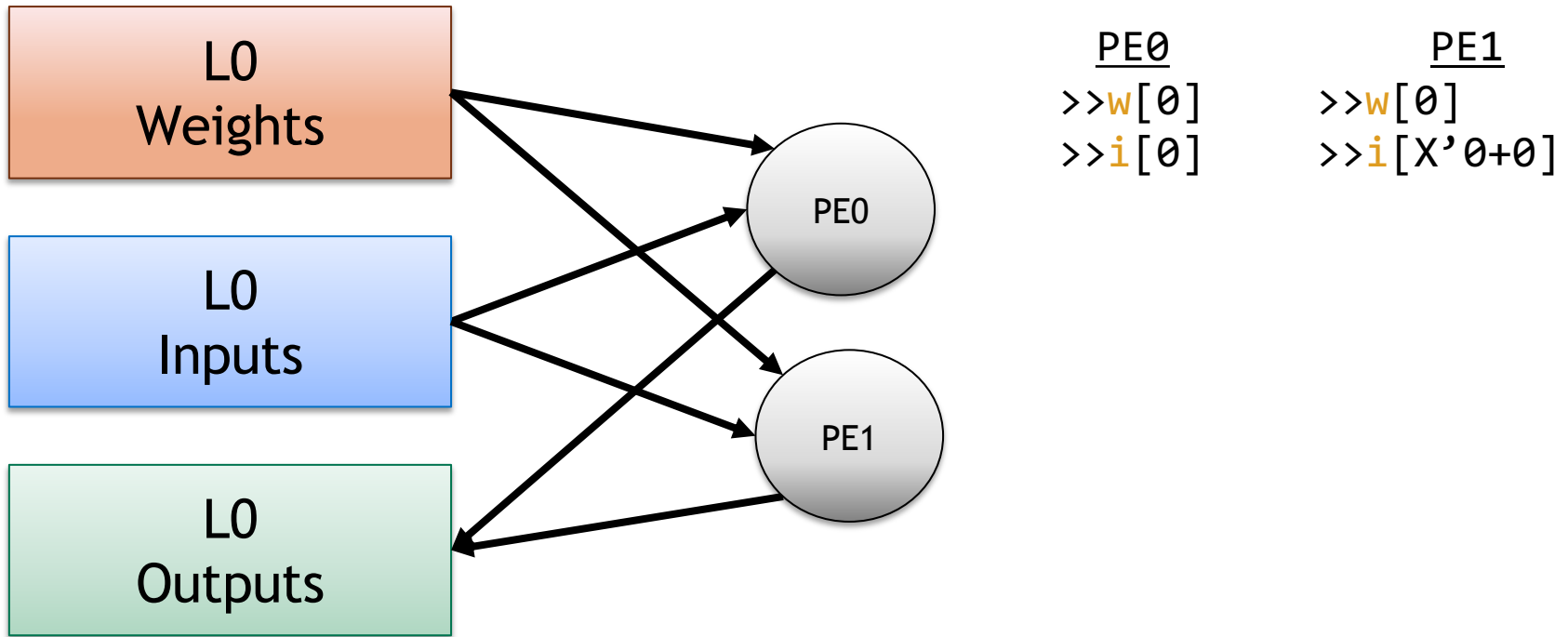
Spatial PEs: Partitioned Outputs



Implementation opportunity?

Assuming $S=3$

Spatial PEs: Partitioned Outputs

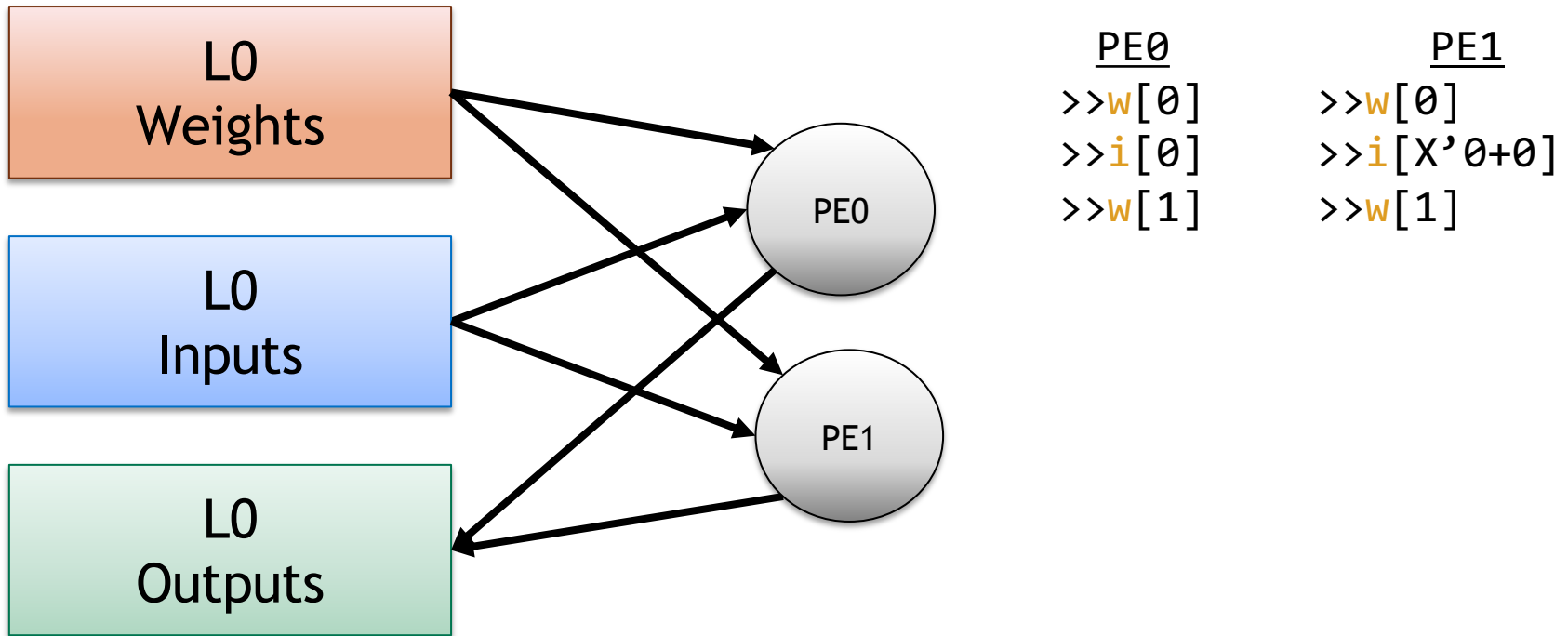


Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs

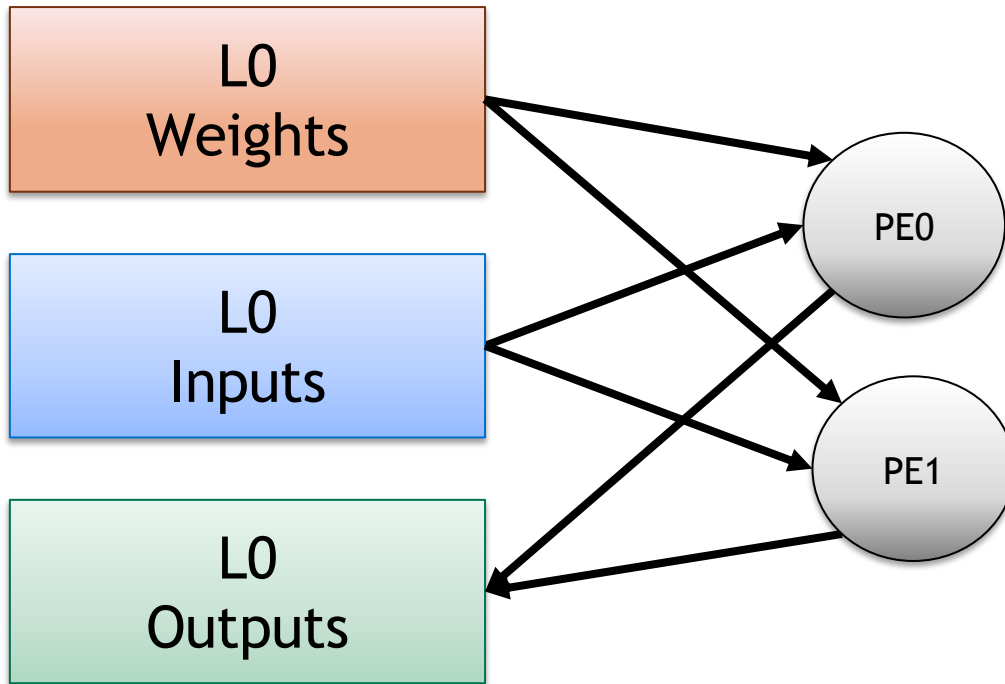


Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs



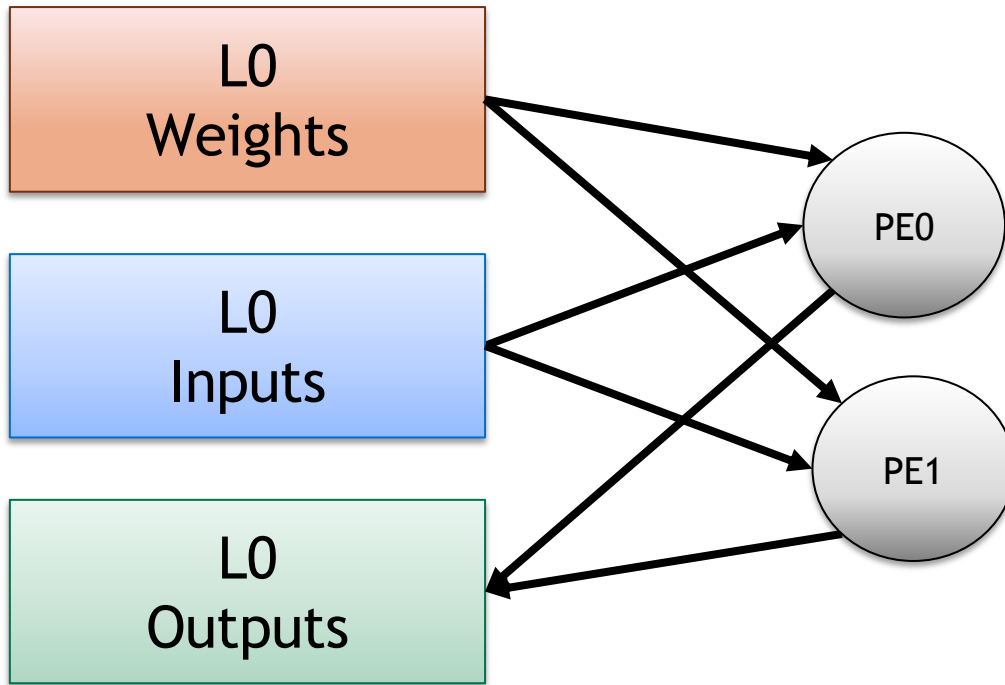
<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[0]
>>i[0]	>>i[X'0+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X'0+1]

Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs



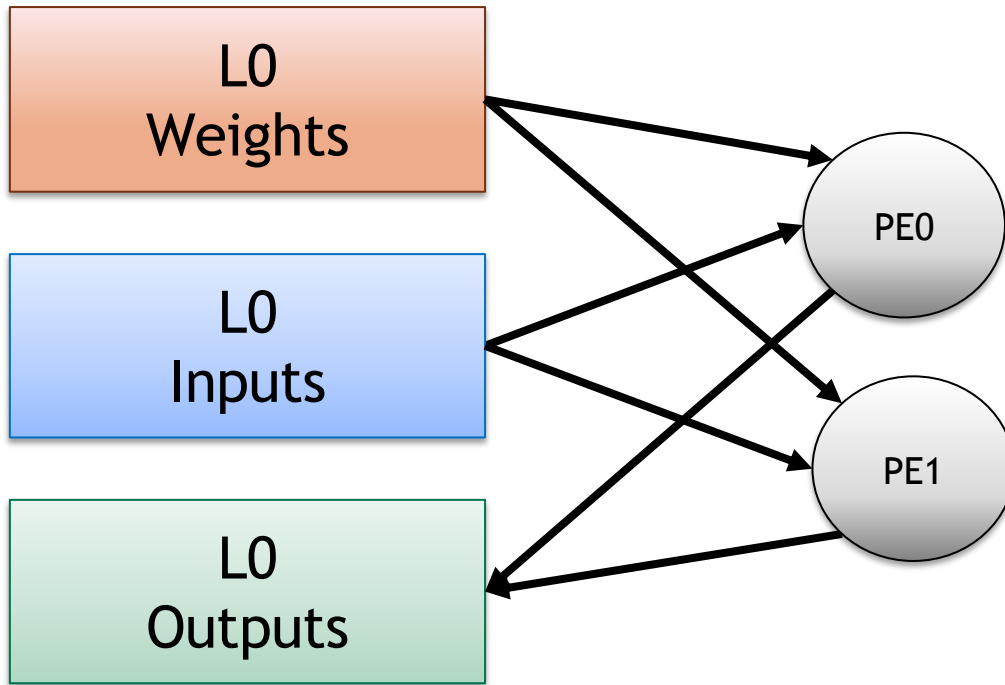
<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[0]
>>i[0]	>>i[X'0+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X'0+1]
>>w[2]	>>w[2]

Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs



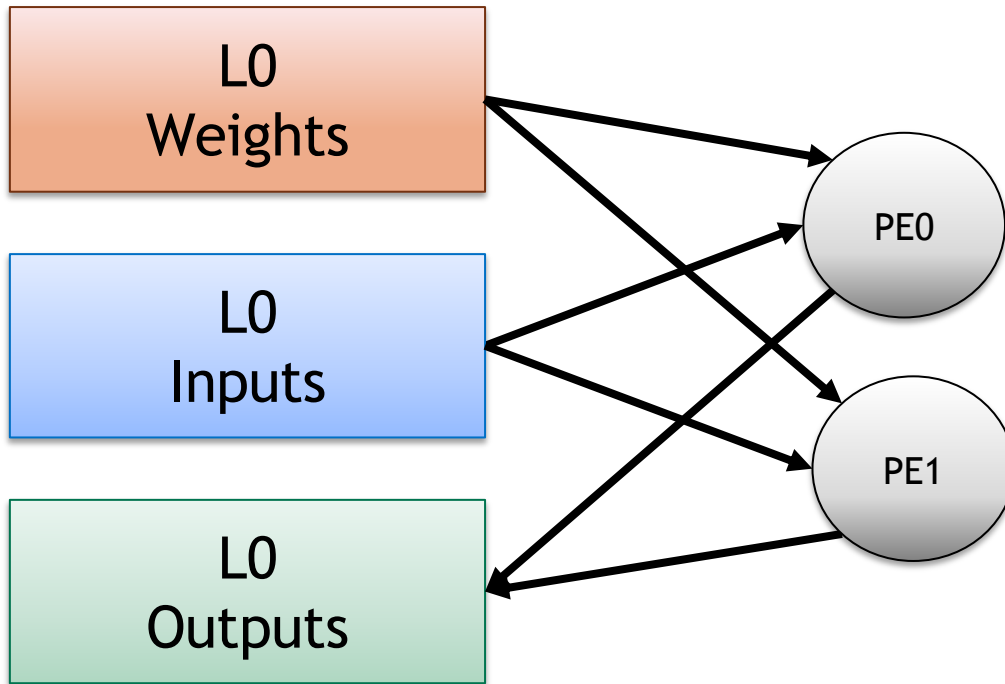
<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[0]
>>i[0]	>>i[X'0+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X'0+1]
>>w[2]	>>w[2]
>>i[2]	>>i[X'0+2]

Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs



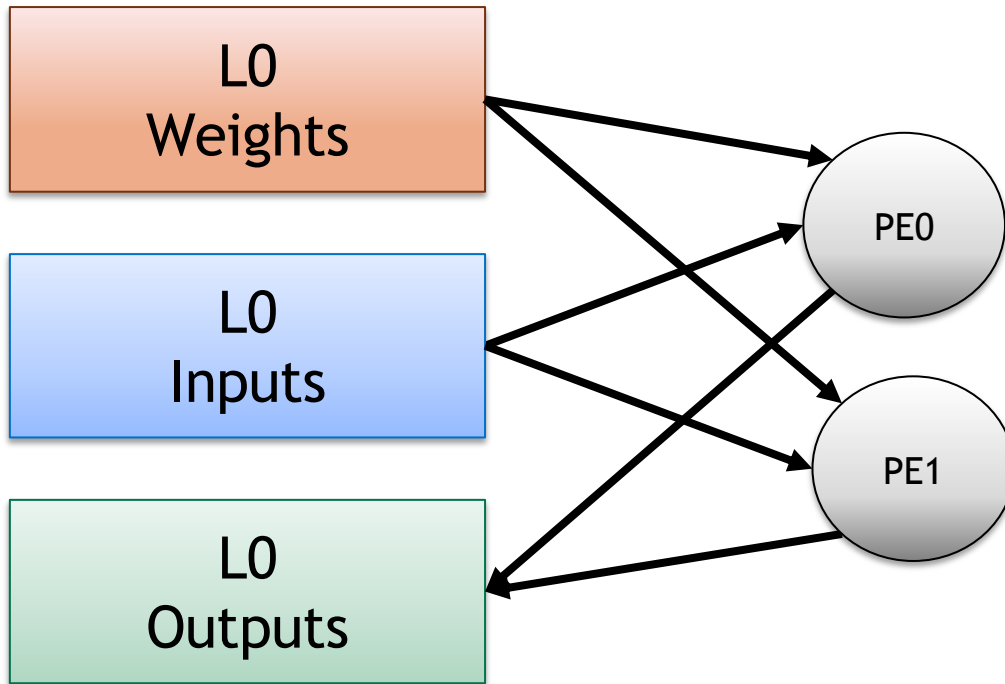
<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[0]
>>i[0]	>>i[X'0+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X'0+1]
>>w[2]	>>w[2]
>>i[2]	>>i[X'0+2]
<<o[0]	<<o[X'0+0]

Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs



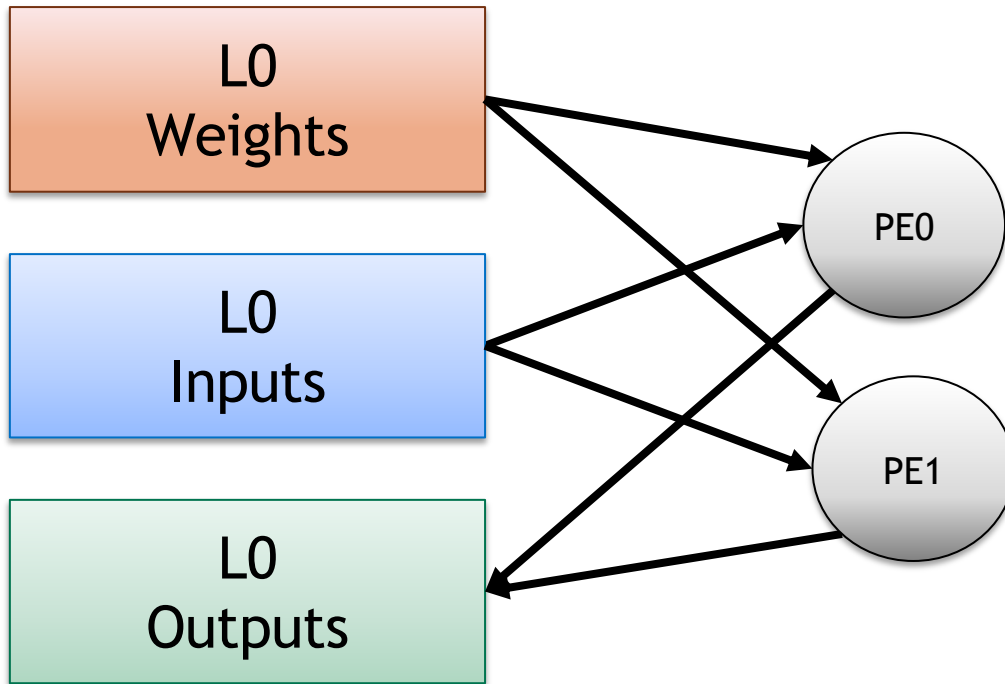
<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[0]
>>i[0]	>>i[X'0+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X'0+1]
>>w[2]	>>w[2]
>>i[2]	>>i[X'0+2]
<<o[0]	<<o[X'0+0]
>>w[0]	>>w[0]

Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs



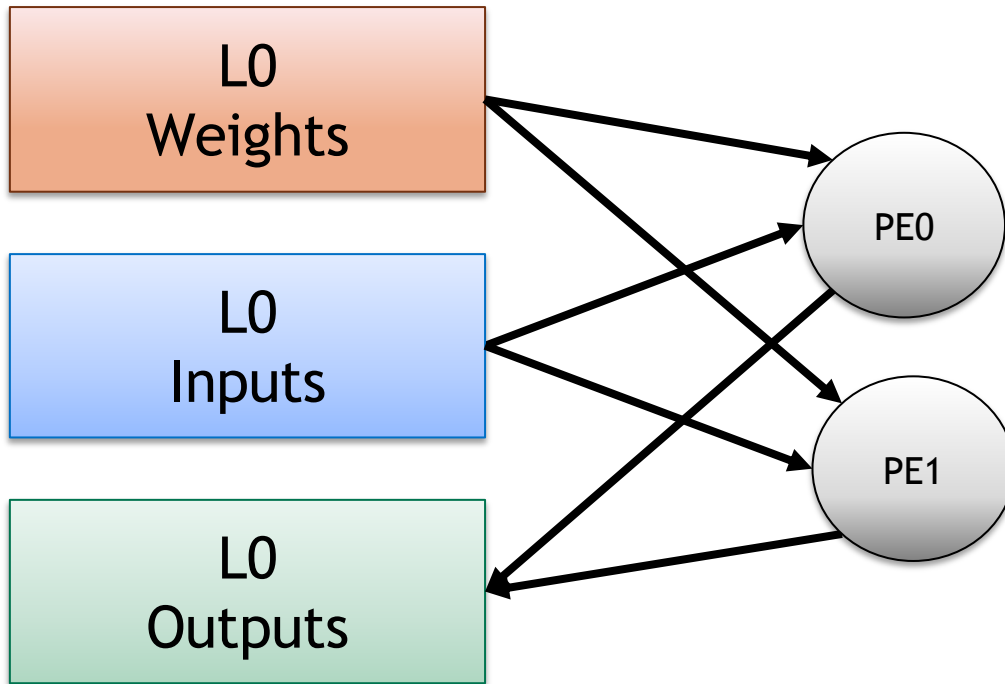
<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[0]
>>i[0]	>>i[X'0+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X'0+1]
>>w[2]	>>w[2]
>>i[2]	>>i[X'0+2]
<<o[0]	<<o[X'0+0]
>>w[0]	>>w[0]
>>i[1]	>>i[X'0+1]

Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs



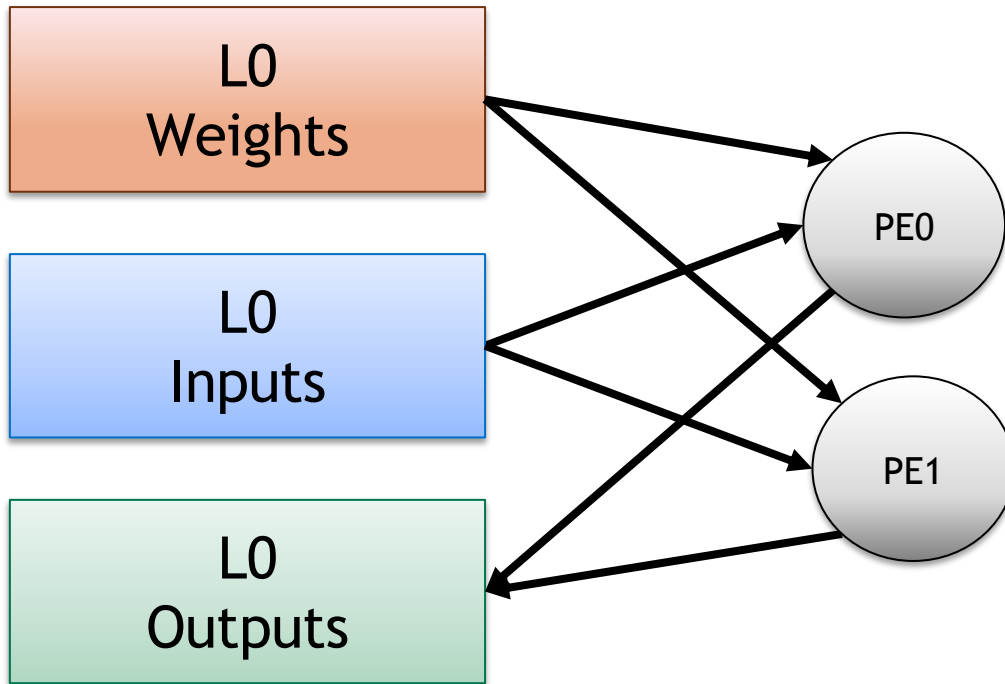
<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[0]
>>i[0]	>>i[X'0+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X'0+1]
>>w[2]	>>w[2]
>>i[2]	>>i[X'0+2]
<<o[0]	<<o[X'0+0]
>>w[0]	>>w[0]
>>i[1]	>>i[X'0+1]
>>w[1]	>>w[1]

Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs



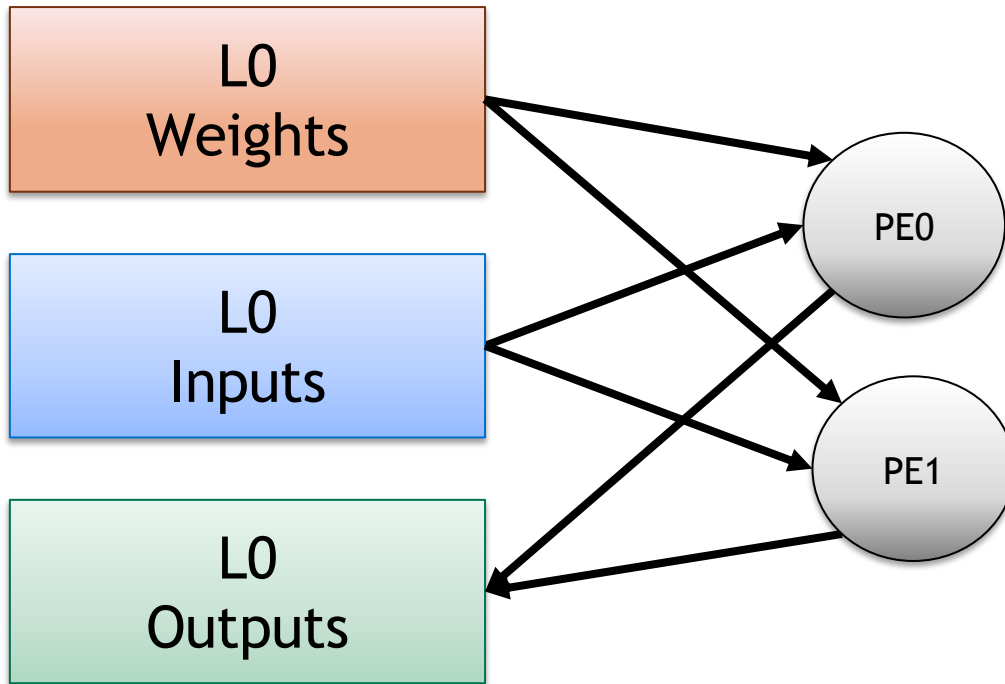
<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[0]
>>i[0]	>>i[X'0+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X'0+1]
>>w[2]	>>w[2]
>>i[2]	>>i[X'0+2]
<<o[0]	<<o[X'0+0]
>>w[0]	>>w[0]
>>i[1]	>>i[X'0+1]
>>w[1]	>>w[1]
>>i[2]	>>i[X'0+2]

Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs



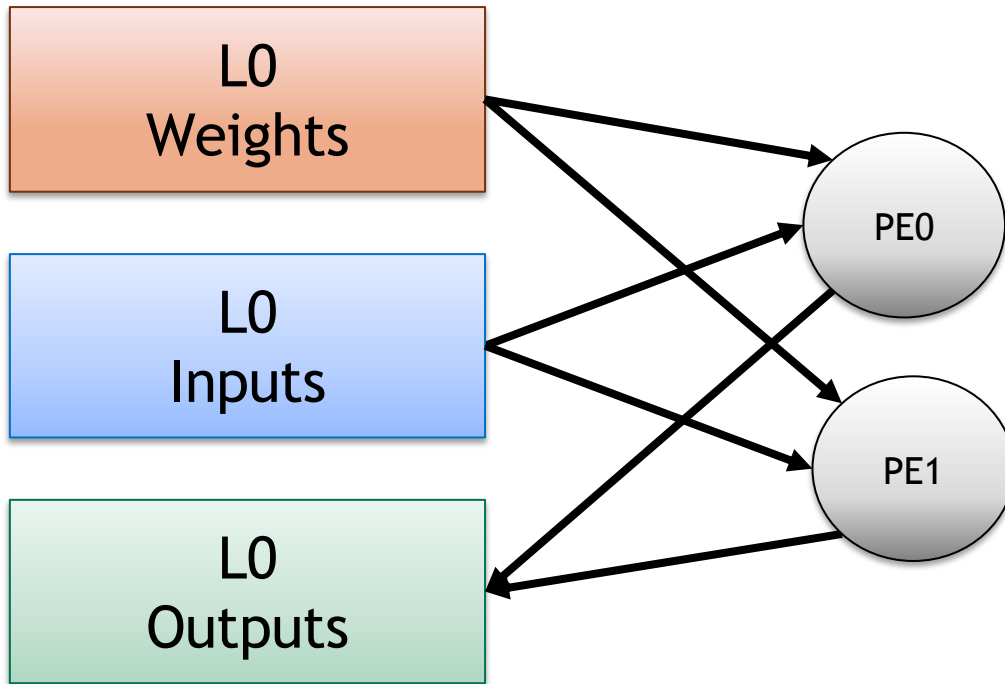
<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[0]
>>i[0]	>>i[X'0+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X'0+1]
>>w[2]	>>w[2]
>>i[2]	>>i[X'0+2]
<<o[0]	<<o[X'0+0]
>>w[0]	>>w[0]
>>i[1]	>>i[X'0+1]
>>w[1]	>>w[1]
>>i[2]	>>i[X'0+2]
>>w[2]	>>w[2]

Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs



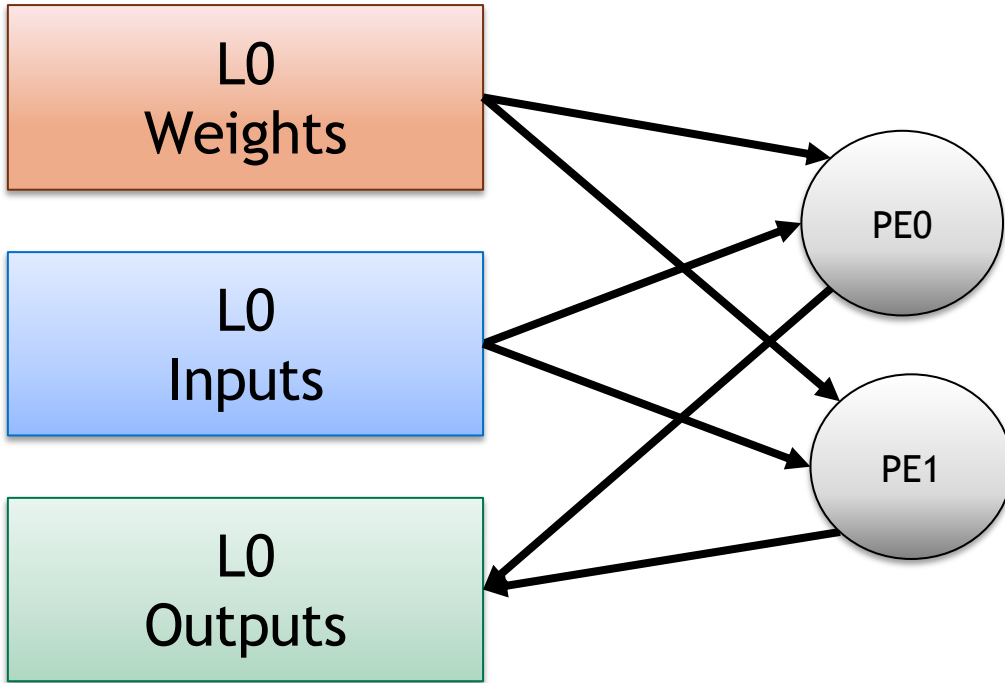
<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[0]
>>i[0]	>>i[X'θ+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X'θ+1]
>>w[2]	>>w[2]
>>i[2]	>>i[X'θ+2]
<<o[0]	<<o[X'θ+0]
>>w[0]	>>w[0]
>>i[1]	>>i[X'θ+1]
>>w[1]	>>w[1]
>>i[2]	>>i[X'θ+2]
>>w[2]	>>w[2]
>>i[3]	>>i[X'θ+3]

Implementation opportunity?

Parallel fetch

Assuming S=3

Spatial PEs: Partitioned Outputs



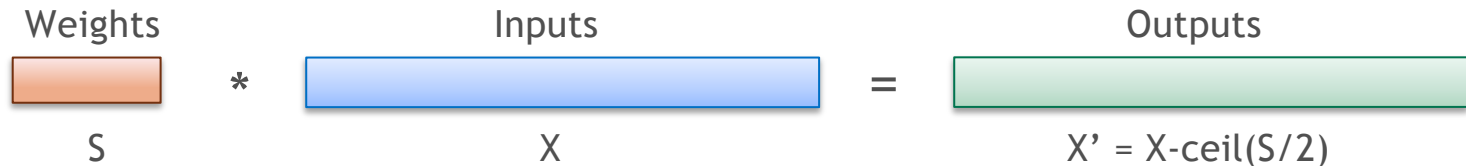
Implementation opportunity?

Parallel fetch

<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[0]
>>i[0]	>>i[X'0+0]
>>w[1]	>>w[1]
>>i[1]	>>i[X'0+1]
>>w[2]	>>w[2]
>>i[2]	>>i[X'0+2]
<<o[0]	<<o[X'0+0]
>>w[0]	>>w[0]
>>i[1]	>>i[X'0+1]
>>w[1]	>>w[1]
>>i[2]	>>i[X'0+2]
>>w[2]	>>w[2]
>>i[3]	>>i[X'0+3]
<<o[1]	<<o[X'0+1]

Assuming S=3

1D Convolution – Partition Weights



```
int i[X];      # Input activations
int w[S];      # Filter Weights
int o[X'];     # Output activations
```

Note:
 $X' * X' = X$
 $S * S = S$

```
// Level 1
parallel-for (s1 = 0; s1 < 2; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'; x0++) {
        for (s0 = 0; s0 < S; s0++) {
            o[x1*X'+x0] += i[x1*X'+x0 + s1*S0+s0]
                * w[s1*S0+s0];
        }
    }
}
```

1D Convolution – Partition Weights

```
// Level 1
parallel-for (s1 = 0; s1 < 2; s1++) {
  // Level 0
  for (x0 = 0; x0 < X'0; x0++) {
    for (s0 = 0; s0 < S0; s0++) {
      o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                    * w[s1*S0+s0];
    }
  }
}
```

1D Convolution – Partition Weights

```
// Level 1
parallel-for (s1 = 0; s1 < 2; s1++) {
  // Level 0
  for (x0 = 0; x0 < X'0; x0++) {
    for (s0 = 0; s0 < S0; s0++) {
      o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                    * w[s1*S0+s0];
    }
  }
}
```

How do we handle same index for output
in multiple PEs?

1D Convolution – Partition Weights

```
// Level 1
parallel-for (s1 = 0; s1 < 2; s1++) {
  // Level 0
  for (x0 = 0; x0 < X'0; x0++) {
    for (s0 = 0; s0 < S0; s0++) {
      o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                    * w[s1*S0+s0];
    }
  }
}
```

How do we handle same index for output **Spatial reduction** in multiple PEs?

1D Convolution – Partition Weights

```
// Level 1
parallel-for (s1 = 0; s1 < 2; s1++) {
  // Level 0
  for (x0 = 0; x0 < X'0; x0++) {
    for (s0 = 0; s0 < S0; s0++) {
      o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                    * w[s1*S0+s0];
    }
  }
}
```

How do we handle same index for output in multiple PEs? **Spatial reduction**

Other multicast opportunities?

1D Convolution – Partition Weights

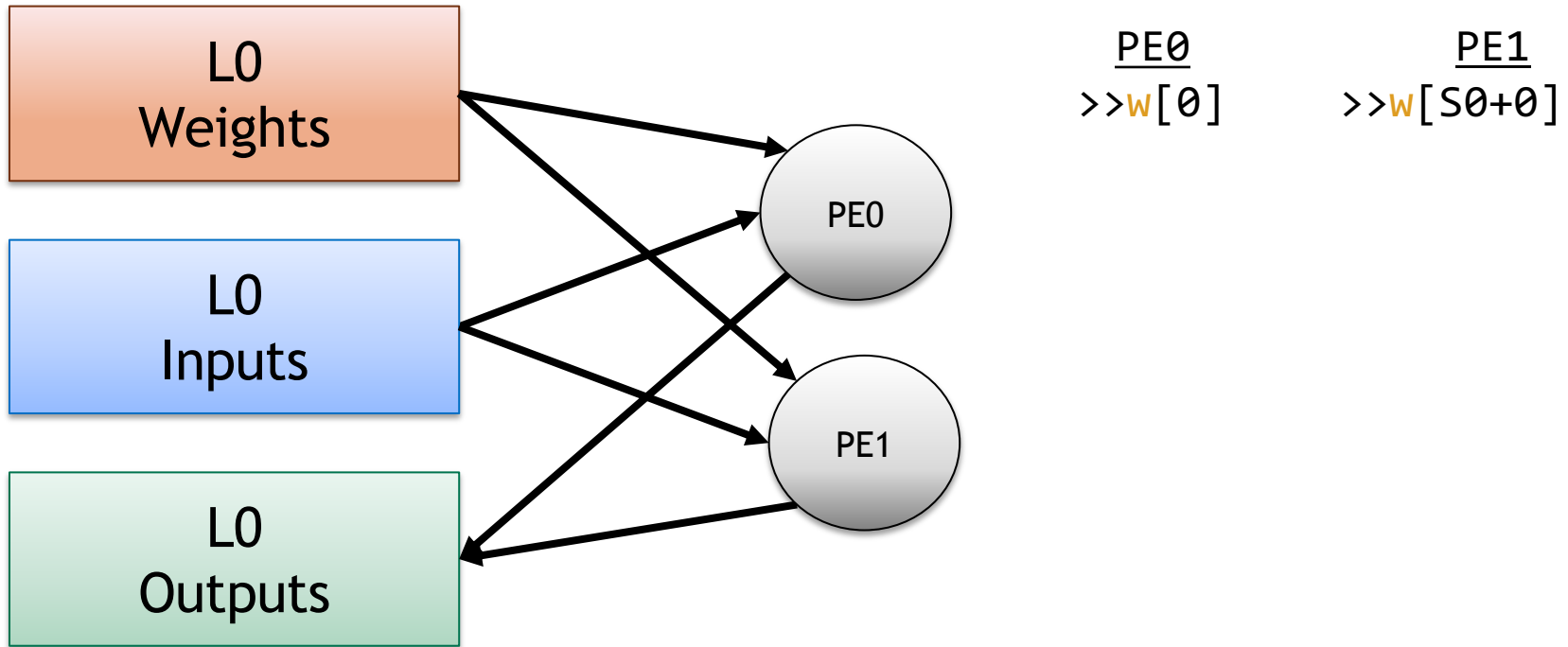
```
// Level 1
parallel-for (s1 = 0; s1 < 2; s1++) {
  // Level 0
  for (x0 = 0; x0 < X'0; x0++) {
    for (s0 = 0; s0 < S0; s0++) {
      o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]
                    * w[s1*S0+s0];
    }
  }
}
```

How do we handle same index for output in multiple PEs? **Spatial reduction**

Other multicast opportunities?

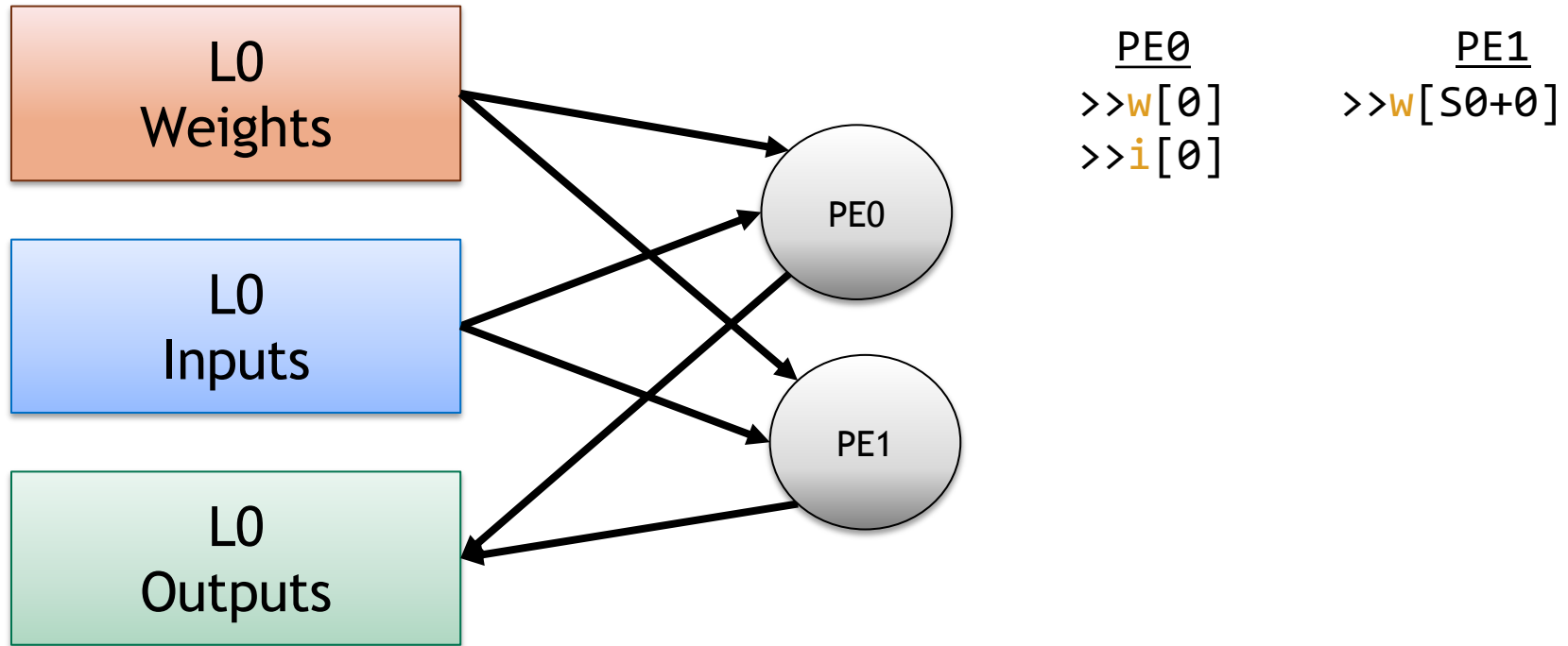
No

Spatial PEs: Partitioned Weights



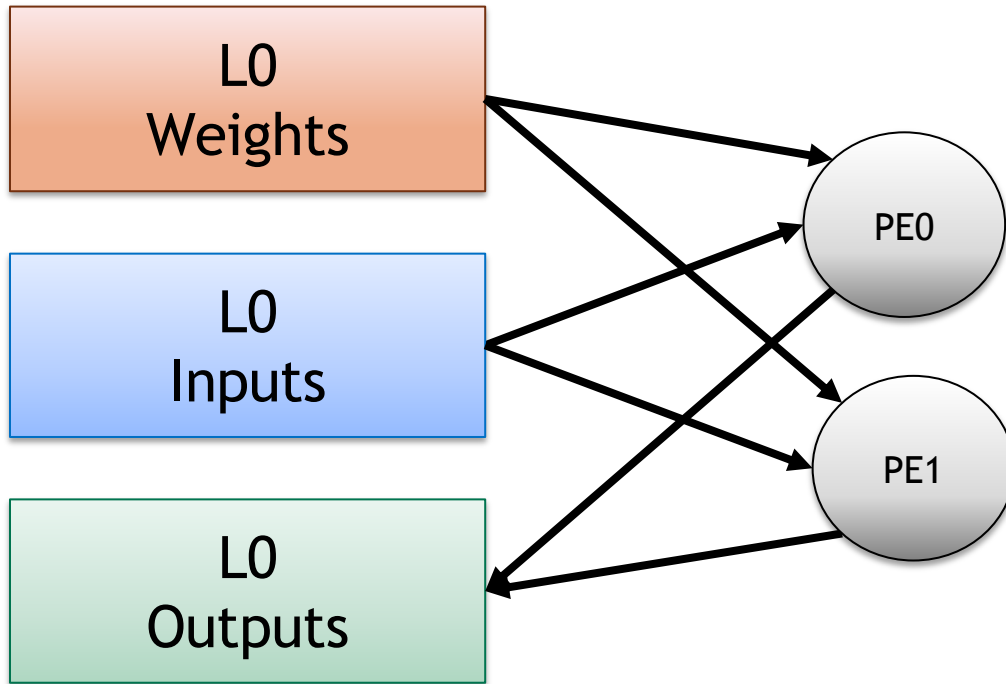
Assuming $S=3$

Spatial PEs: Partitioned Weights



Assuming $S=3$

Spatial PEs: Partitioned Weights

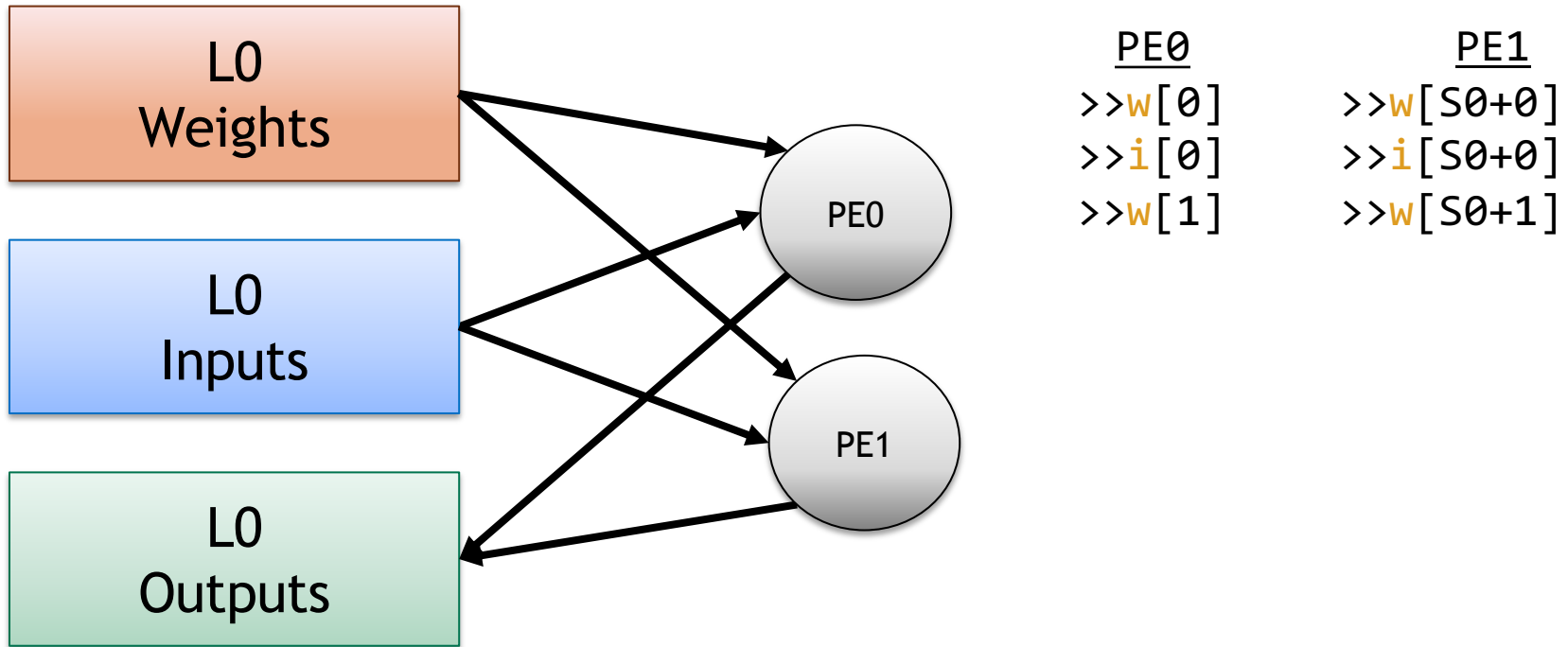


PE0
>>w[0]
>>i[0]

PE1
>>w[S0+0]
>>i[S0+0]

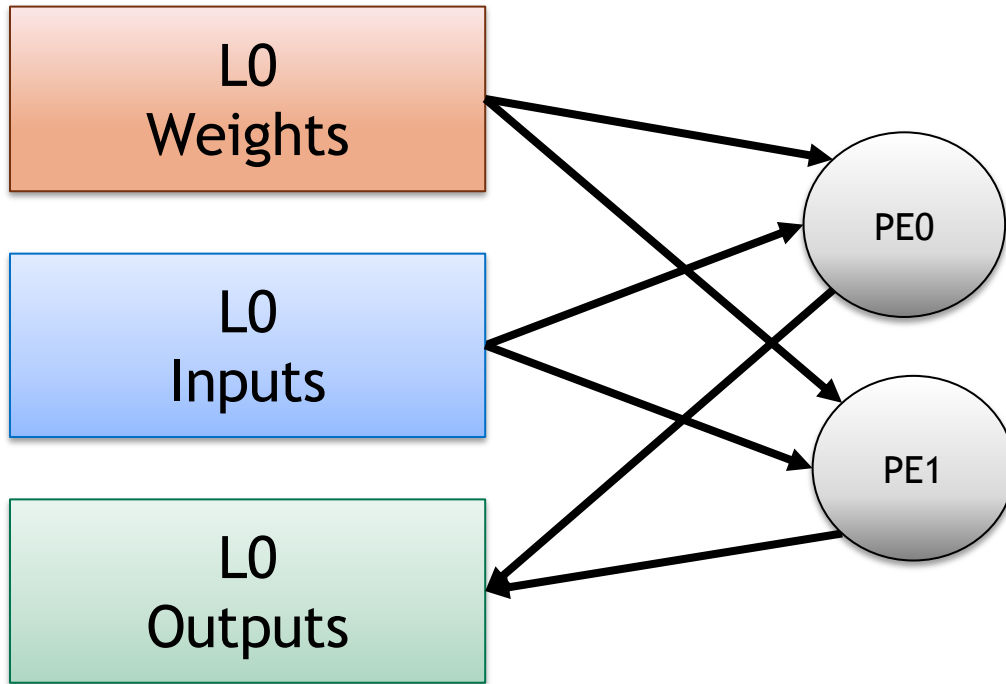
Assuming S=3

Spatial PEs: Partitioned Weights



Assuming S=3

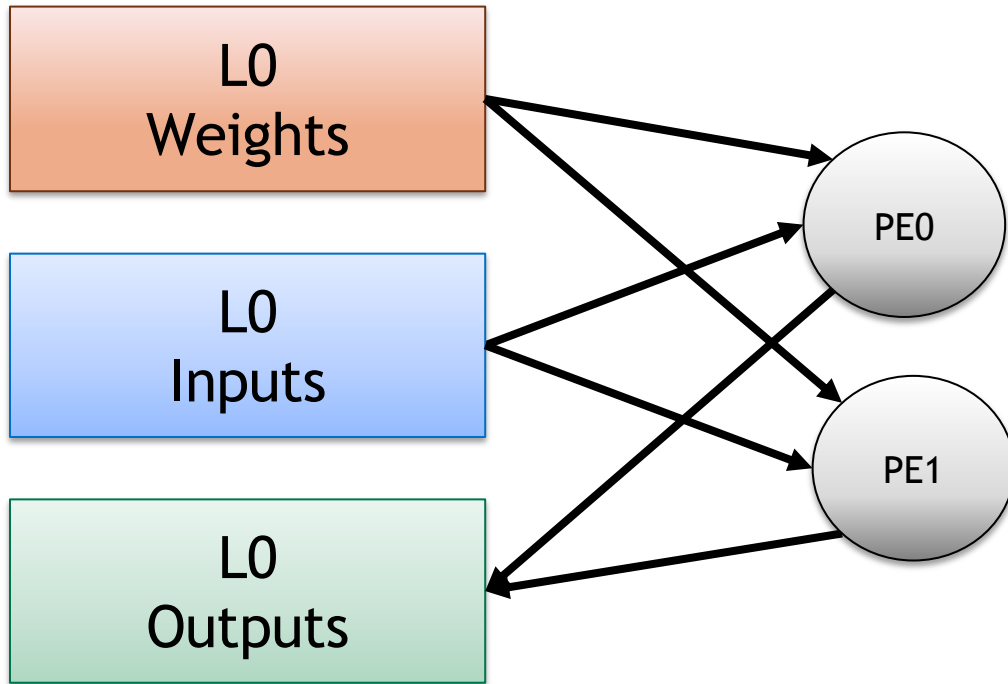
Spatial PEs: Partitioned Weights



<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]

Assuming S=3

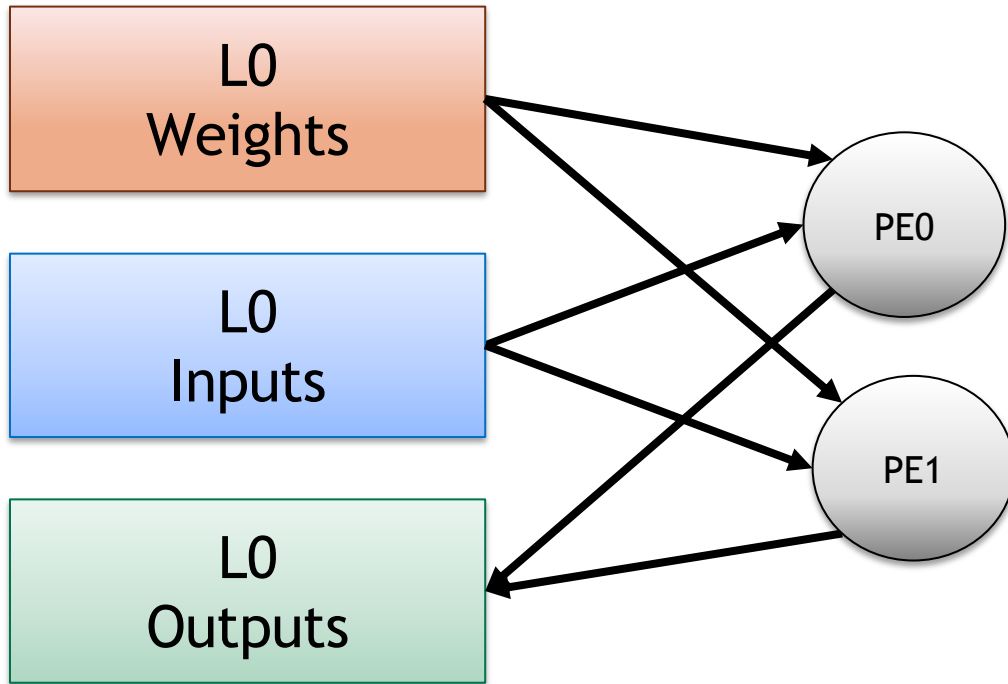
Spatial PEs: Partitioned Weights



<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]

Assuming S=3

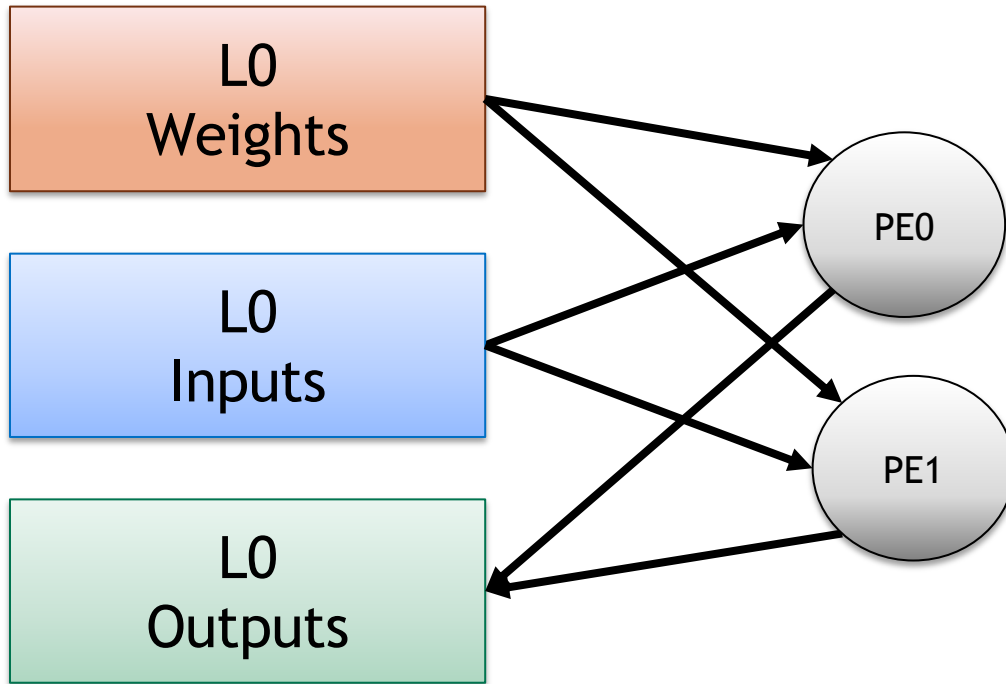
Spatial PEs: Partitioned Weights



<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]

Assuming S=3

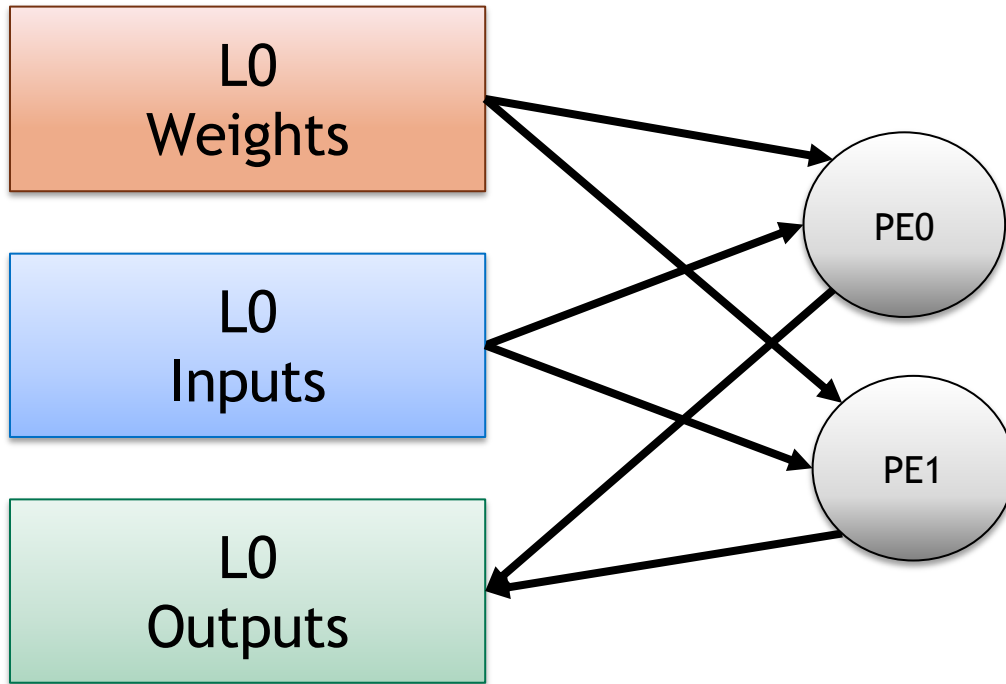
Spatial PEs: Partitioned Weights



<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]
<<o[0]	<<o[0]

Assuming S=3

Spatial PEs: Partitioned Weights

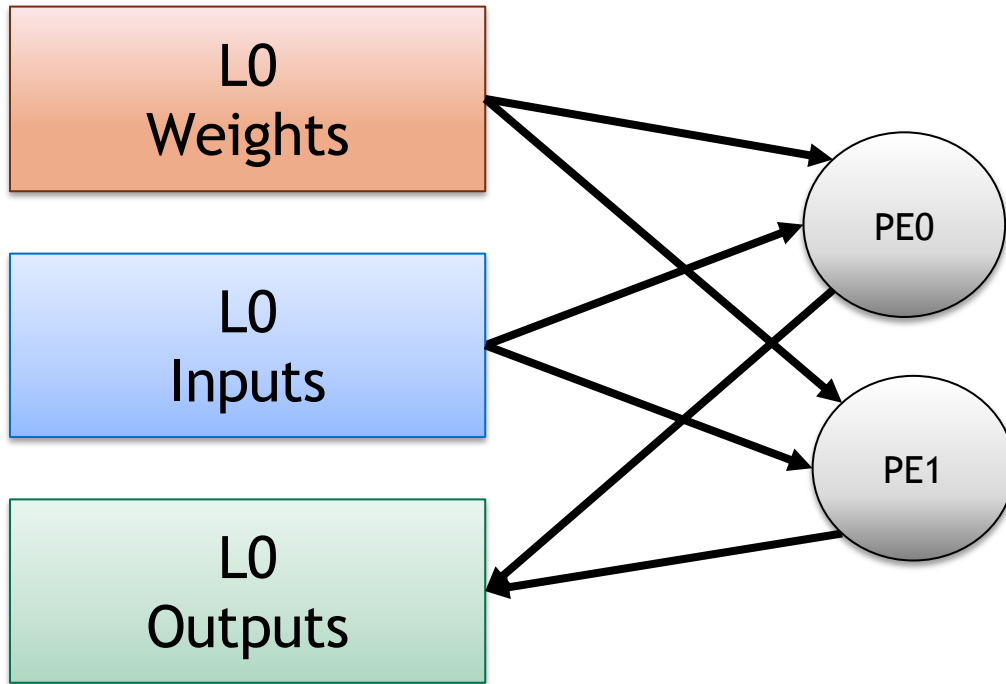


<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]
<<o[0]	<<o[0]

Spatial sum needed?

Assuming S=3

Spatial PEs: Partitioned Weights

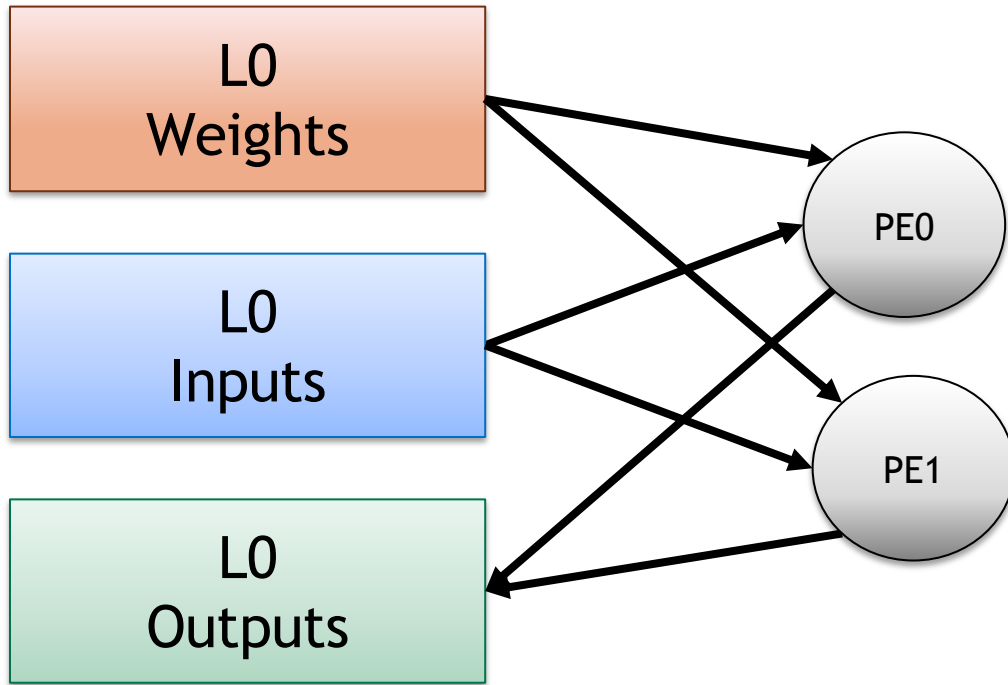


<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]
<<o[0]	<<o[0]

Spatial sum needed? **Yes**

Assuming S=3

Spatial PEs: Partitioned Weights

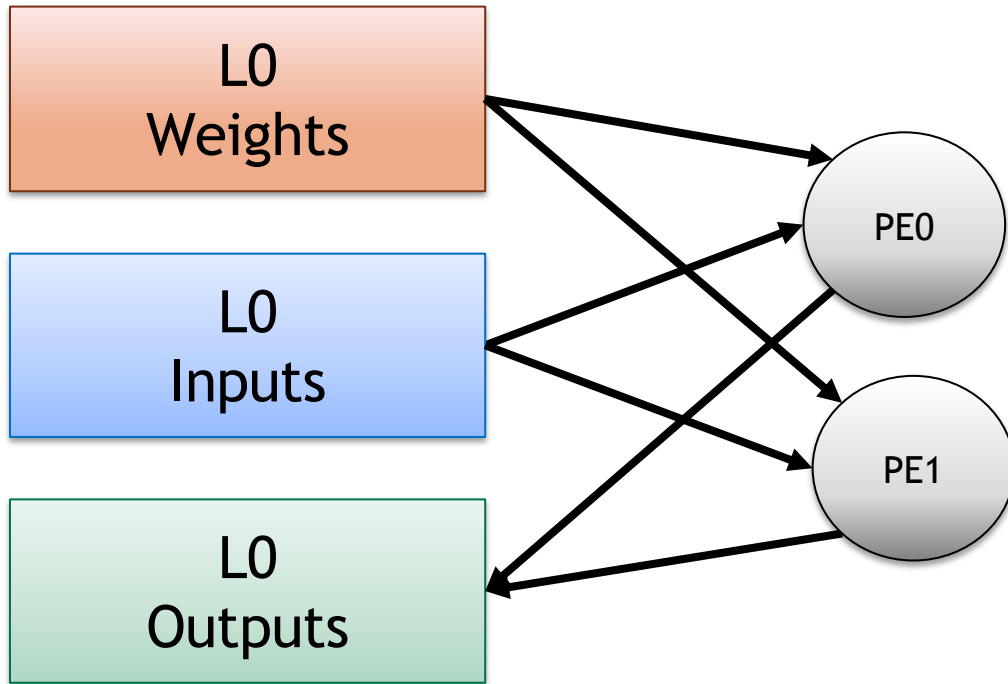


<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]
<<o[0]	<<o[0]
>>w[0]	>>w[S0+1]

Spatial sum needed? **Yes**

Assuming S=3

Spatial PEs: Partitioned Weights

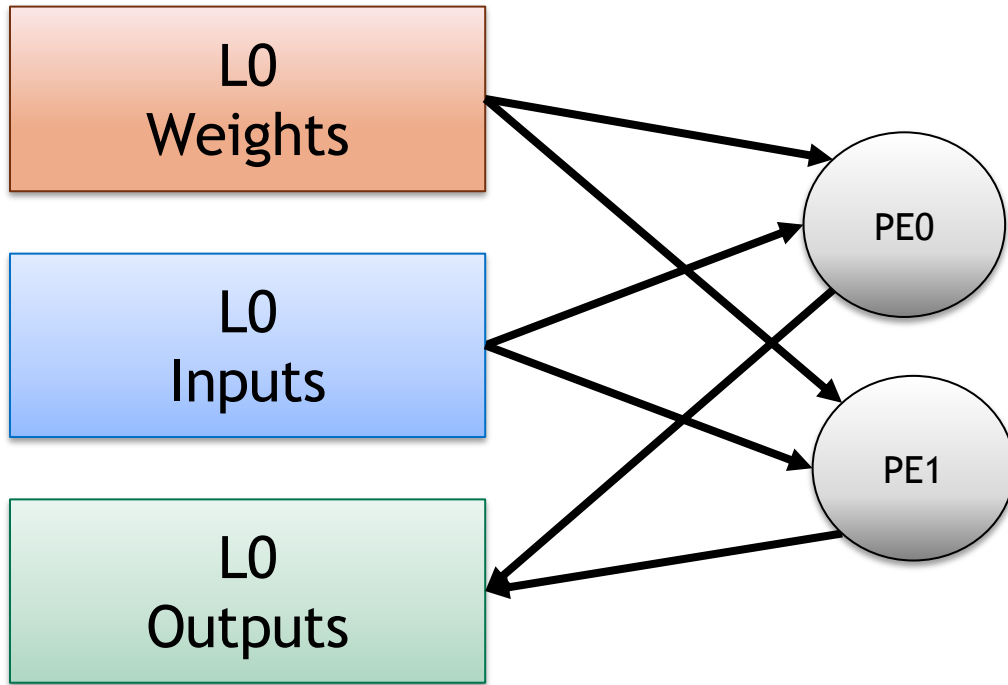


<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]
<<o[0]	<<o[0]
>>w[0]	>>w[S0+1]
>>i[1]	>>i[S0+1]

Spatial sum needed? **Yes**

Assuming S=3

Spatial PEs: Partitioned Weights

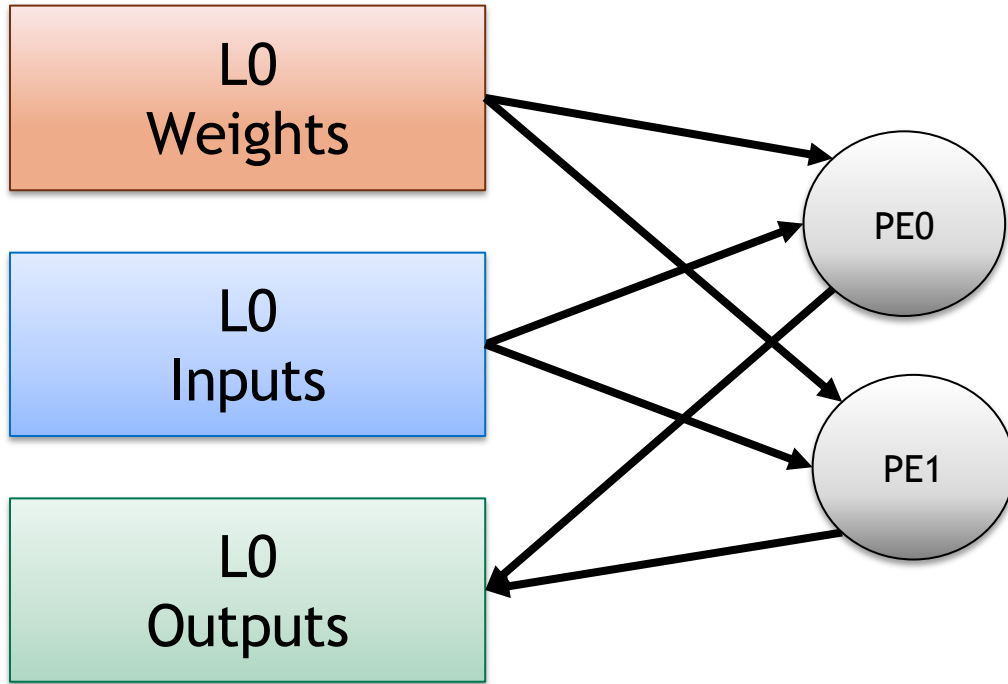


<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]
<<o[0]	<<o[0]
>>w[0]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[1]	>>w[S0+2]

Spatial sum needed? **Yes**

Assuming S=3

Spatial PEs: Partitioned Weights

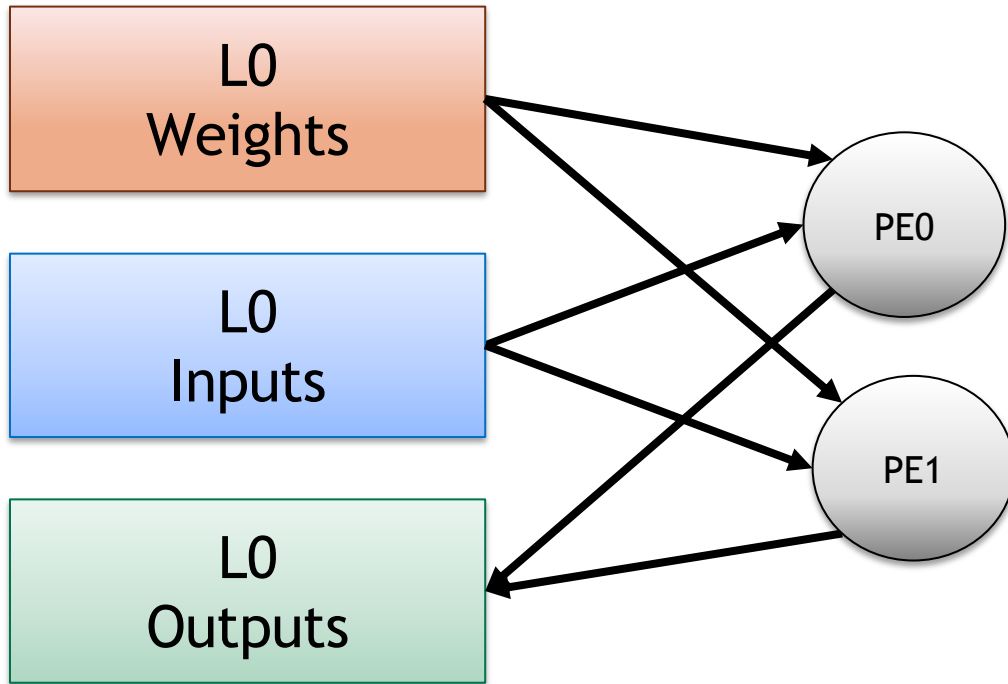


<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]
<<o[0]	<<o[0]
>>w[0]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[1]	>>w[S0+2]
>>i[2]	>>i[S0+2]

Spatial sum needed? **Yes**

Assuming S=3

Spatial PEs: Partitioned Weights

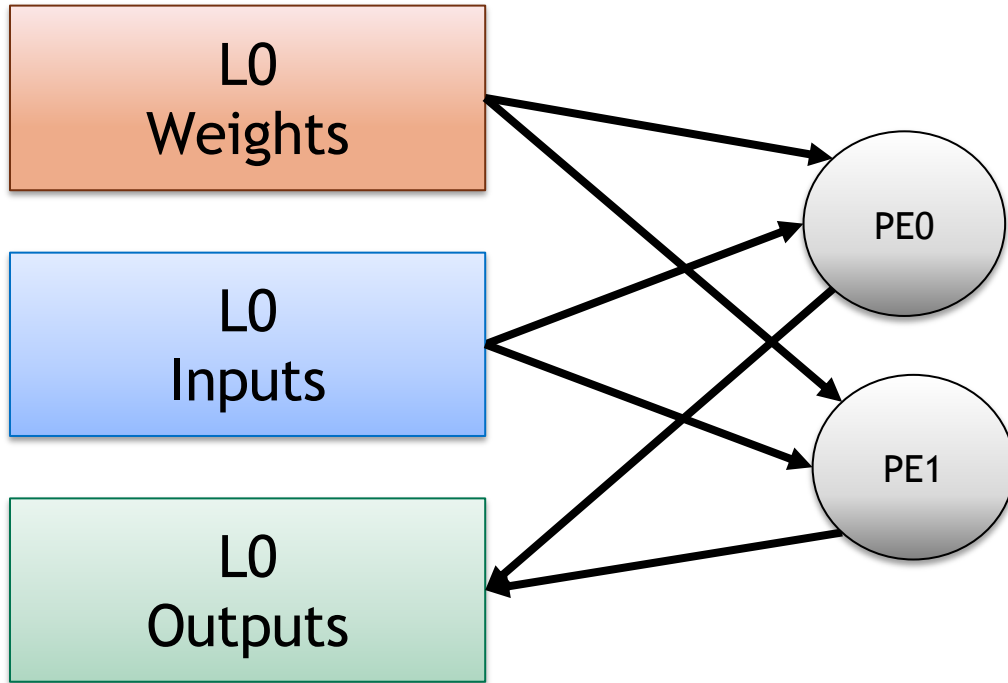


<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]
<<o[0]	<<o[0]
>>w[0]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[1]	>>w[S0+2]
>>i[2]	>>i[S0+2]
>>w[2]	>>w[S0+3]

Spatial sum needed? **Yes**

Assuming S=3

Spatial PEs: Partitioned Weights

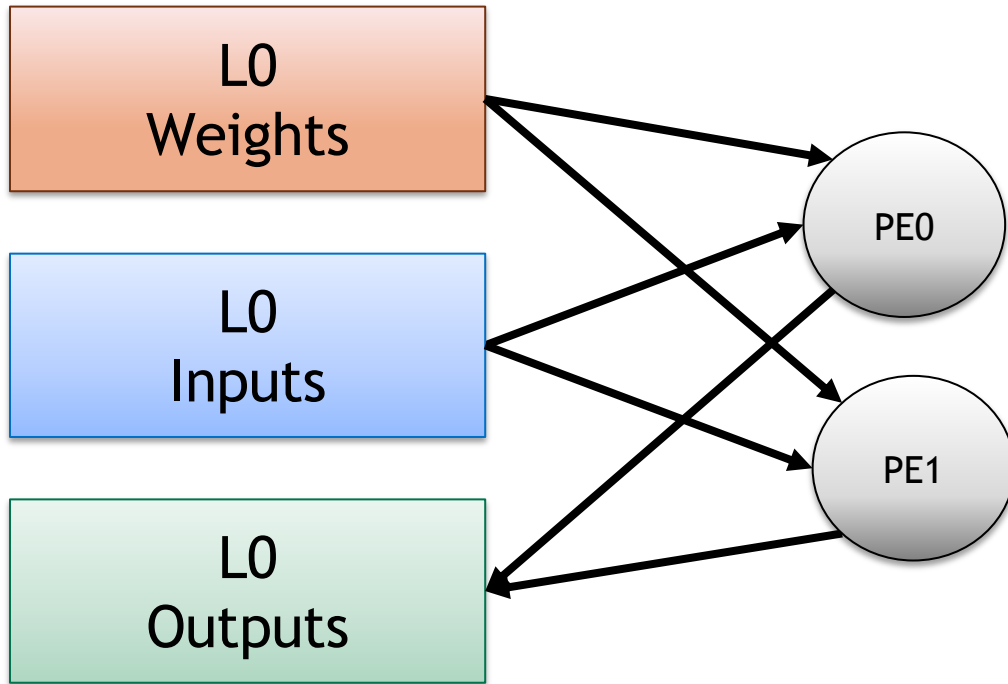


Spatial sum needed? **Yes**

<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]
<<o[0]	<<o[0]
>>w[0]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[1]	>>w[S0+2]
>>i[2]	>>i[S0+2]
>>w[2]	>>w[S0+3]
>>i[3]	>>i[S0+3]

Assuming S=3

Spatial PEs: Partitioned Weights



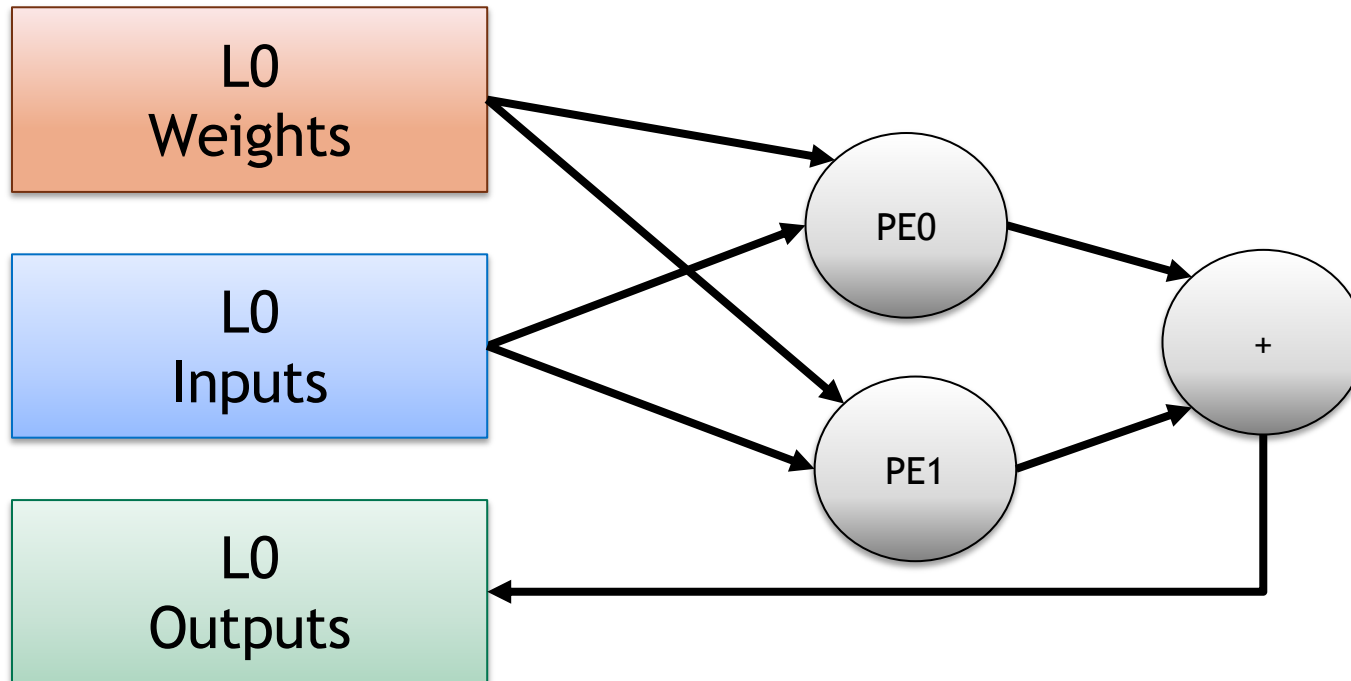
Spatial sum needed?

Yes

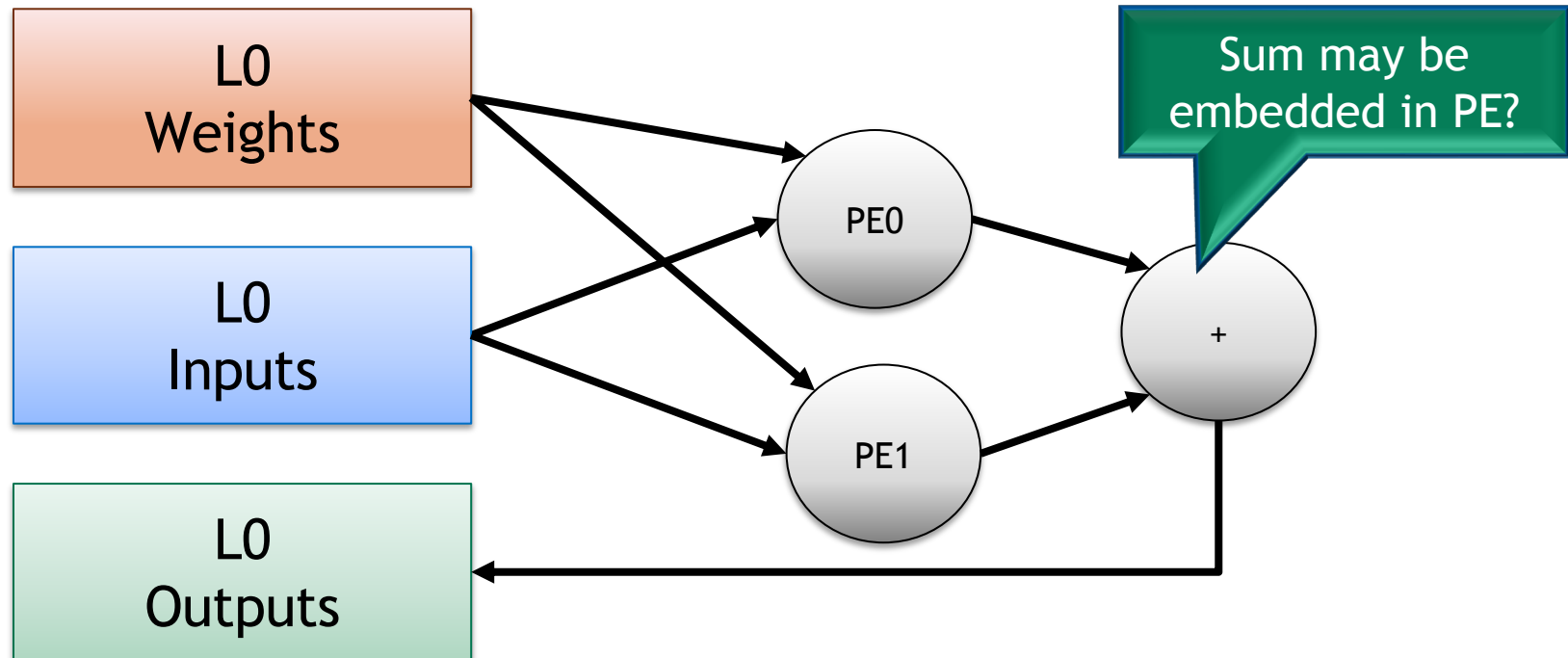
<u>PE0</u>	<u>PE1</u>
>>w[0]	>>w[S0+0]
>>i[0]	>>i[S0+0]
>>w[1]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[2]	>>w[S0+2]
>>i[2]	>>i[S0+2]
<<o[0]	<<o[0]
>>w[0]	>>w[S0+1]
>>i[1]	>>i[S0+1]
>>w[1]	>>w[S0+2]
>>i[2]	>>i[S0+2]
>>w[2]	>>w[S0+3]
>>i[3]	>>i[S0+3]
<<o[1]	<<o[1]

Assuming S=3

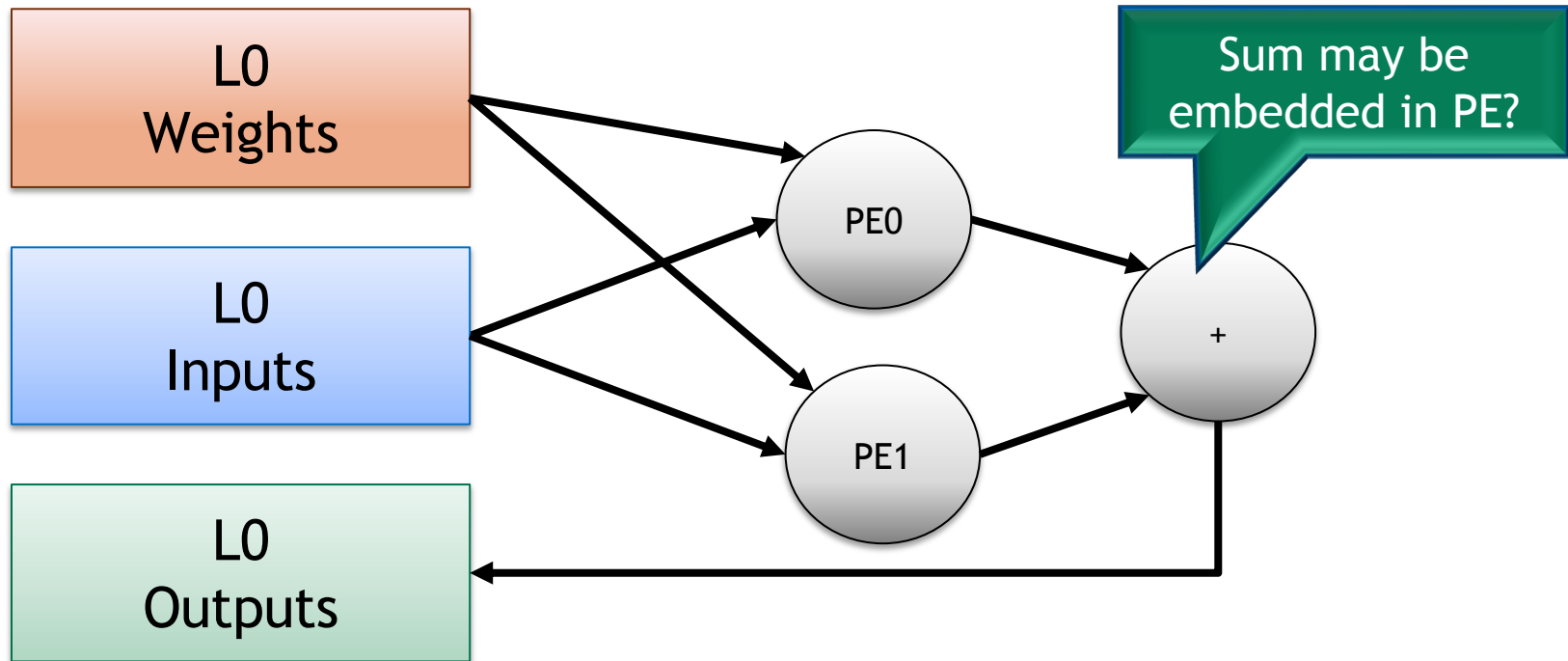
Spatial PEs with Spatial Summation



Spatial PEs with Spatial Summation

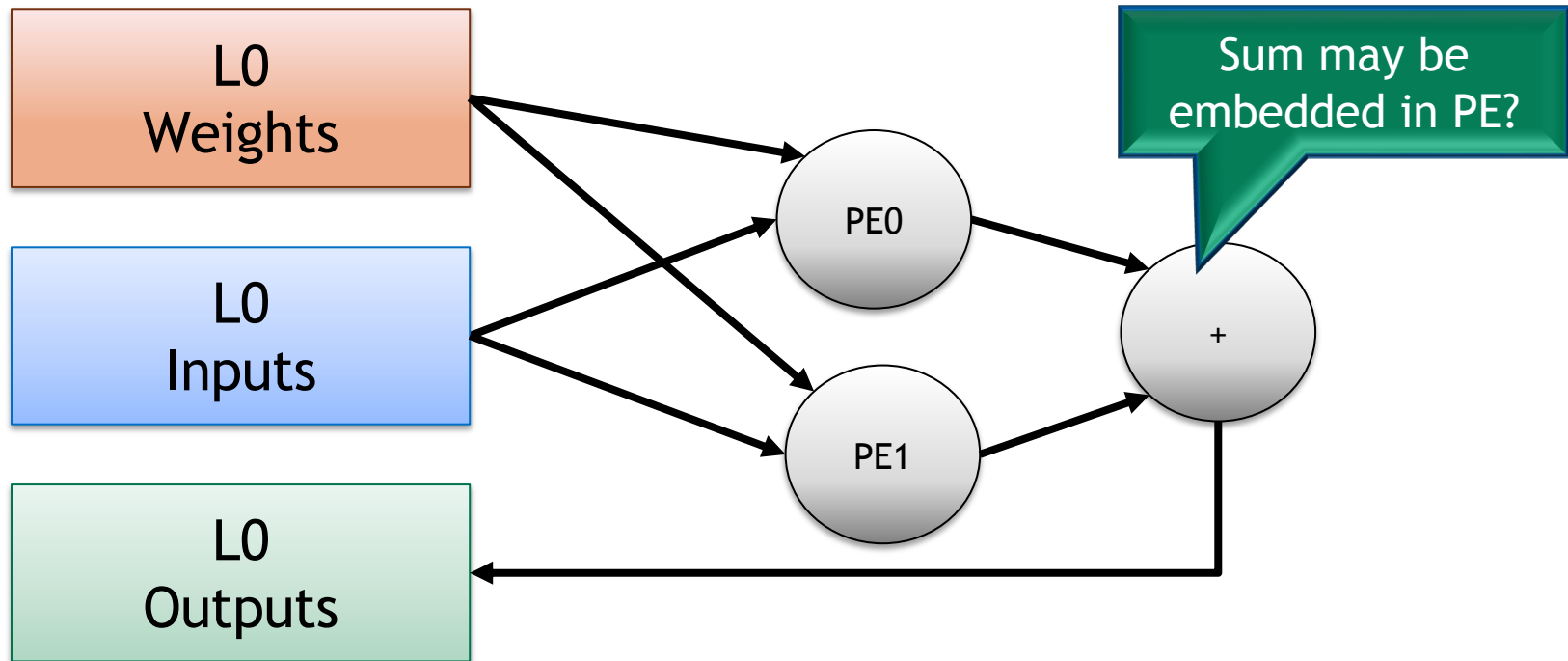


Spatial PEs with Spatial Summation



What if hardware cannot do a spatial sum?

Spatial PEs with Spatial Summation



What if hardware cannot do a spatial sum?

Illegal mapping!

NoC Support

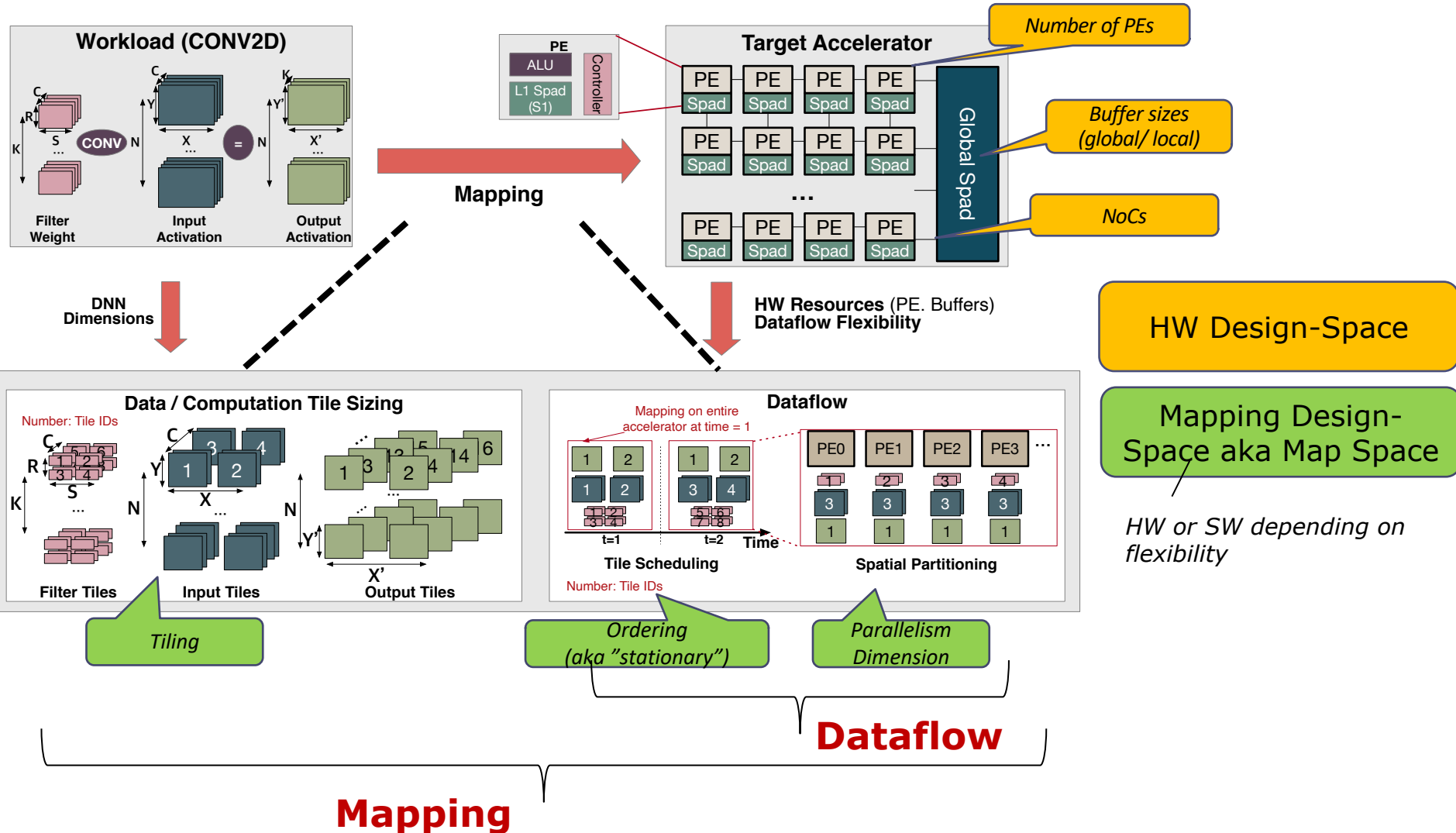
Hardware Structure	Per Data Type	Output-partitioned Dataflow Implication	Weight-partitioned Dataflow Implication
Network-on-Chip for Spatial Reuse	Weight Distribution	Spatial Multicast	Unicast
	Input Distribution	Unicast/Spatial Multicast	Unicast
	Output Collection	Temporal Reduction	Spatial Reduction

More Realistic Loop Nest

```
int i[W];      # Input activations
int w[R];      # Filter Weights
int o[E];      # Output activations

// Level 2
for (x2 = 0; x2 < X'2; x2++) {
  for (s2 = 0; s2 < S2; s2++) {
    // Level 1
    parallel-for (x1 = 0; x1 < X'1; x1++) {
      parallel-for (s1 = 0; s1 < S1; s1++) {
        // Level 0
        for (x0 = 0; x0 < X'0; x0++) {
          for (s0 = 0; s0 < S0; s0++) {
            ...
          }
        }
      }
    }
  }
}
```

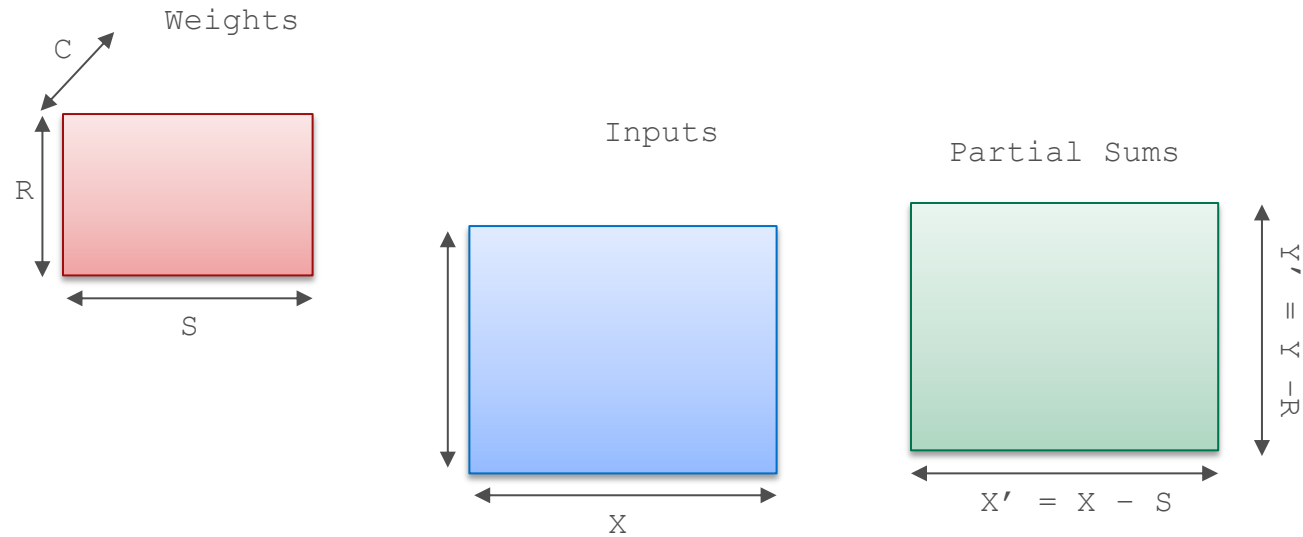
Design-space of a DNN Accelerator



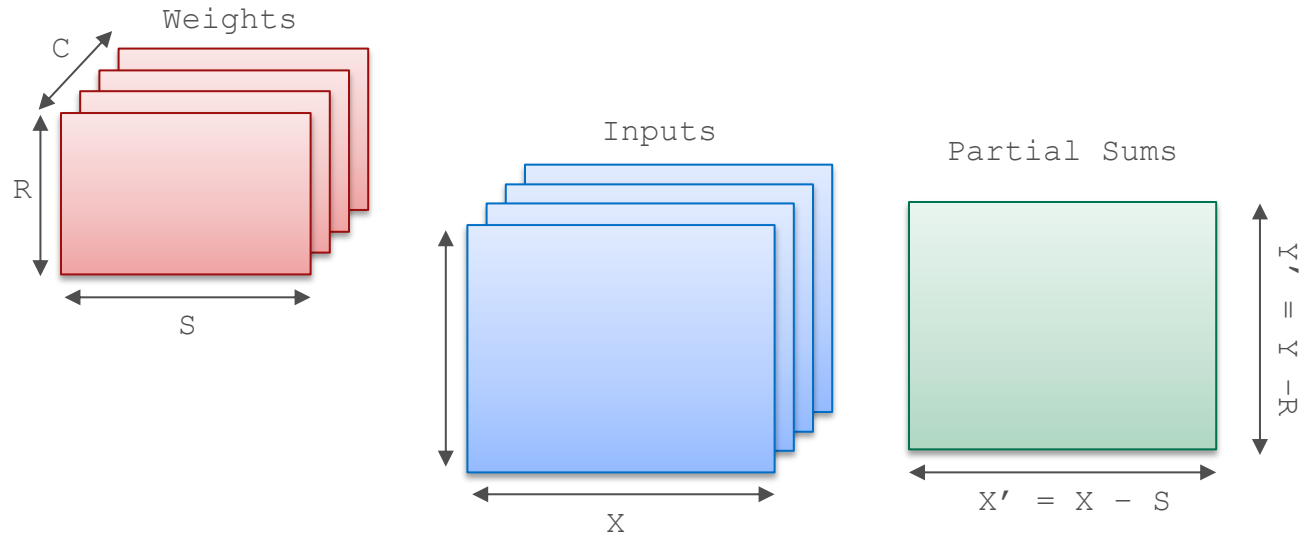
Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
 - Multi-layer Buffering
 - Multiple PEs
 - Full Convolution
- Advanced Dataflows

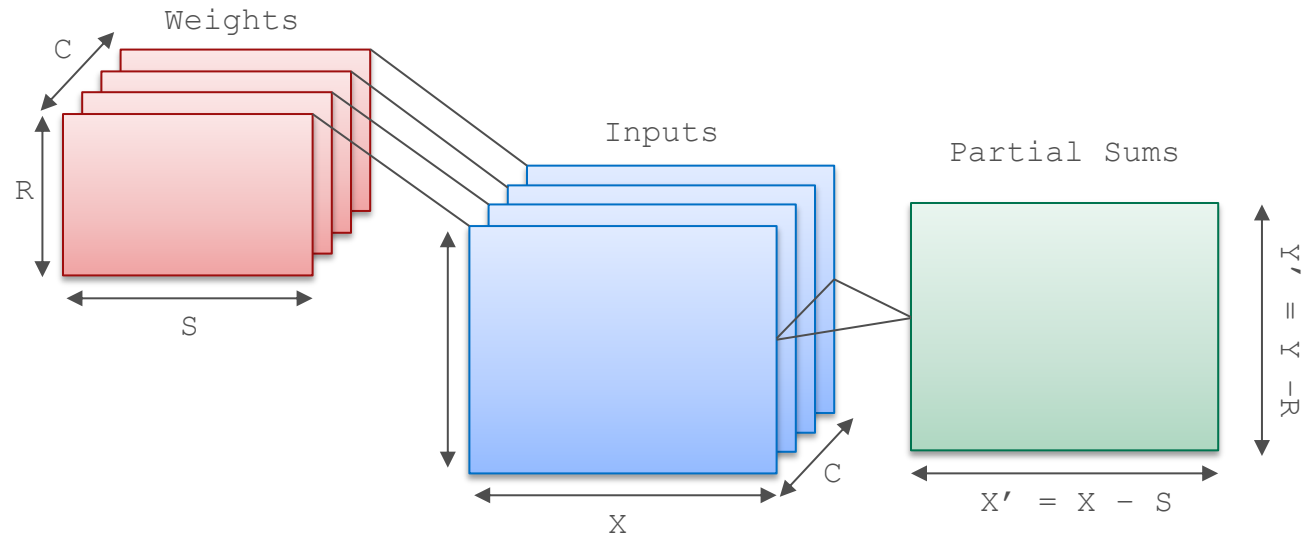
Mapping a Full Convolution



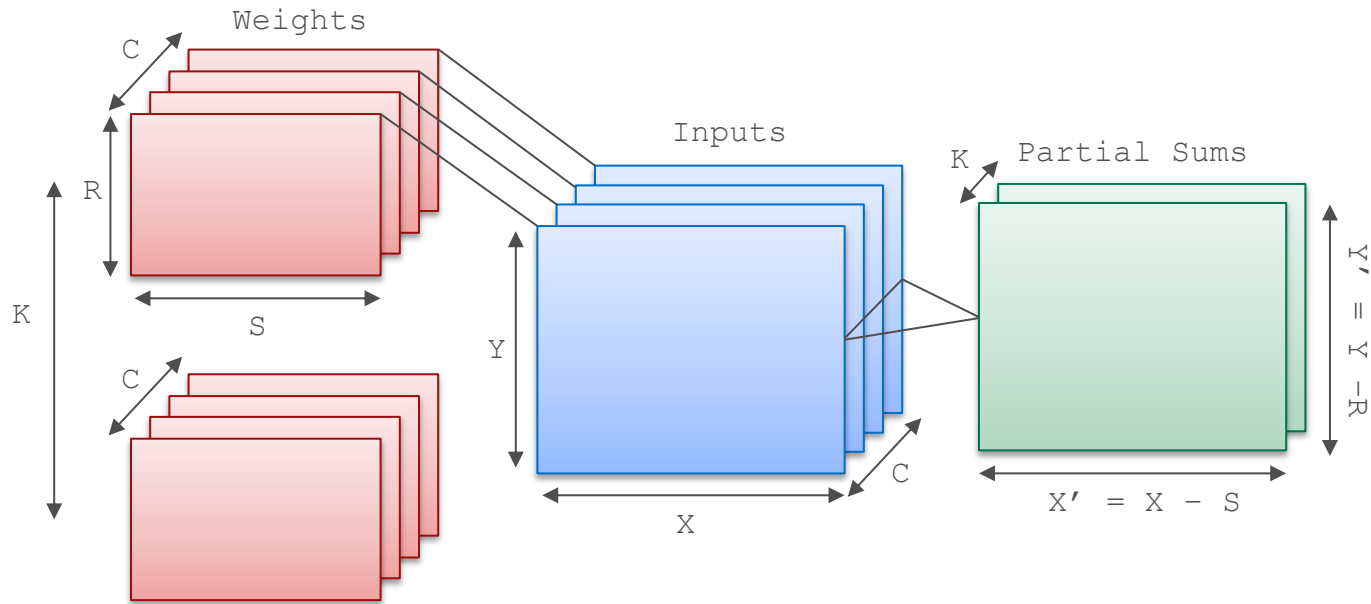
Mapping a Full Convolution



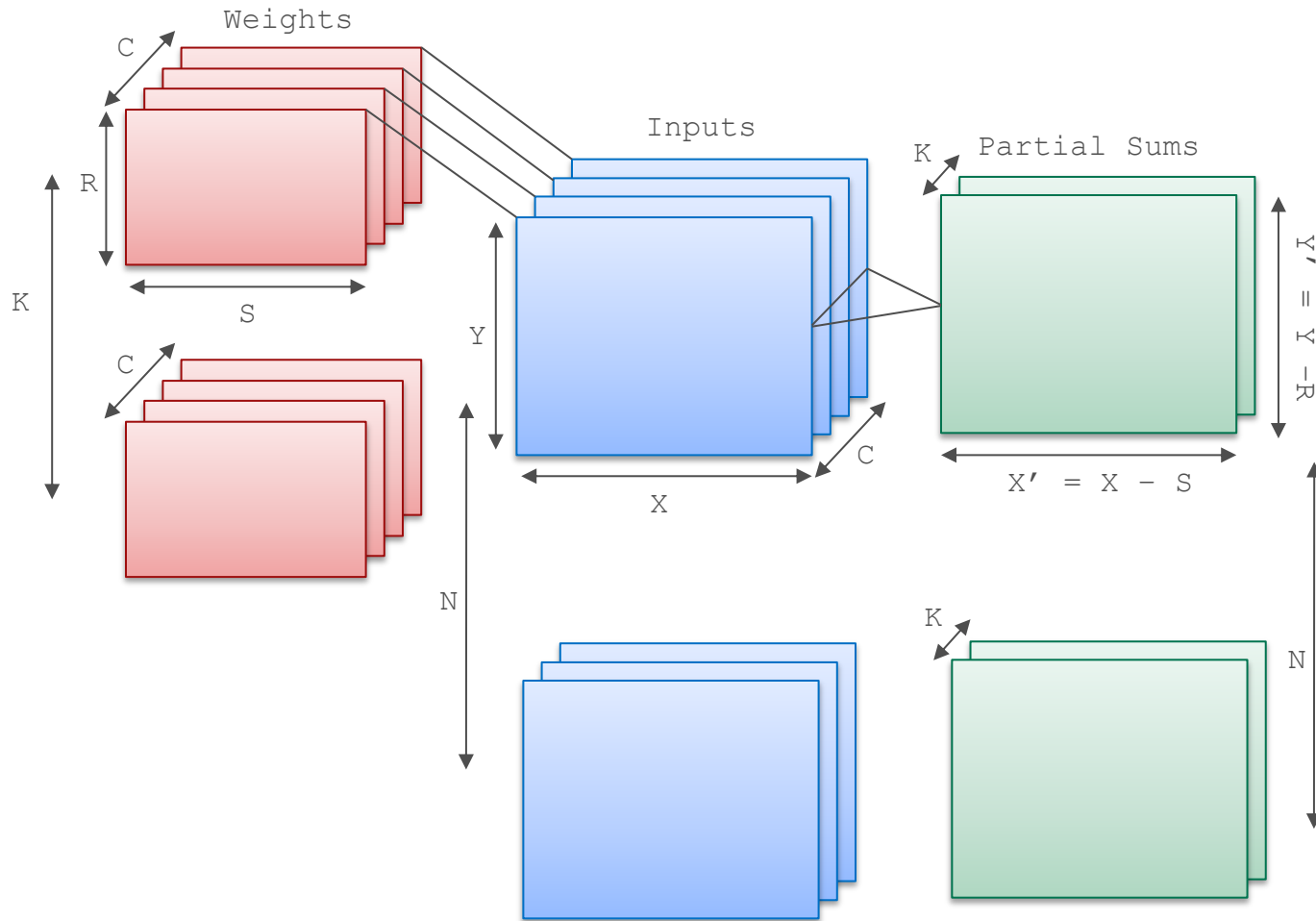
Mapping a Full Convolution



Mapping a Full Convolution



Mapping a Full Convolution



Reference Convolution Layer

```
int i[C][Y][X];      # Input activation channels
int w[K][C][R][S];   # Filter weights (per channel pair)
int o[K][Y'][X'];    # Output activation channels

for (k = 0; k < K; k++) {
    for (y = 0; y < Y'; y++) {
        for (x = 0; x < X'; x++) {
            for (c = 0; c < C; c++) {
                for (r = 0; r < R; r++) {
                    for (s = 0; s < S; s++) {
                        o[k][y][x] += i[c][y+r][x+s]*w[k][c][r][s];
                    }
                }
            }
        }
    }
}
```


A Mapping Representation

- For each temporal and spatial level:
 - Permutation order of all indices
 - Partitioning of data volume for all indices (factoring)
 - $\forall X \in \text{indices} \left(\prod_{l=0}^{\text{maxlevel}} X_l \right) \geq X_{\text{total}}$
 - Data bypass flag per datatype (for temporal layers)

$$(N_l, K_l, C_l, X'_l, Y'_l, R_l, S_l) [I_l, W_l, O_l]$$

How many permutations per level?

How many bypass combinations per level?

Total choices per temporal level?

A Mapping Representation

- For each temporal and spatial level:
 - Permutation order of all indices
 - Partitioning of data volume for all indices (factoring)
 - $\forall X \in \text{indices} \left(\prod_{l=0}^{\text{maxlevel}} X_l \right) \geq X_{\text{total}}$
 - Data bypass flag per datatype (for temporal layers)

$(N_l, K_l, C_l, X'_l, Y'_l, R_l, S_l) [I_l, W_l, O_l]$

How many permutations per level? (# Indices)!

How many bypass combinations per level?

Total choices per temporal level?

A Mapping Representation

- For each temporal and spatial level:
 - Permutation order of all indices
 - Partitioning of data volume for all indices (factoring)
 - $\forall X \in \text{indices} \left(\prod_{l=0}^{\text{maxlevel}} X_l \right) \geq X_{\text{total}}$
 - Data bypass flag per datatype (for temporal layers)

$(N_l, K_l, C_l, X'_l, Y'_l, R_l, S_l) [I_l, W_l, O_l]$

How many permutations per level? (# Indices)!

How many bypass combinations per level? 2^N

Total choices per temporal level?

A Mapping Representation

- For each temporal and spatial level:
 - Permutation order of all indices
 - Partitioning of data volume for all indices (factoring)
 - $\forall X \in \text{indices} \left(\prod_{l=0}^{\text{maxlevel}} X_l \right) \geq X_{\text{total}}$
 - Data bypass flag per datatype (for temporal layers)

$(N_l, K_l, C_l, X'_l, Y'_l, R_l, S_l) [I_l, W_l, O_l]$

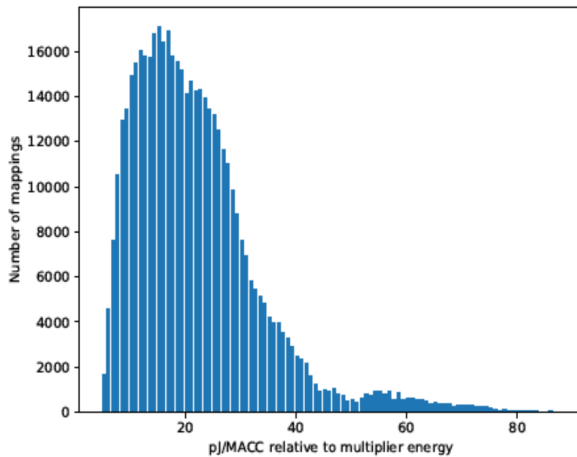
How many permutations per level? (# Indices)!

How many bypass combinations per level? 2^N

Total choices per temporal level? (# Indices)! * 2^N * factorings

Impact of Mappings

VGG conv3 2 Layer. Source: Timeloop



480,000 mappings shown

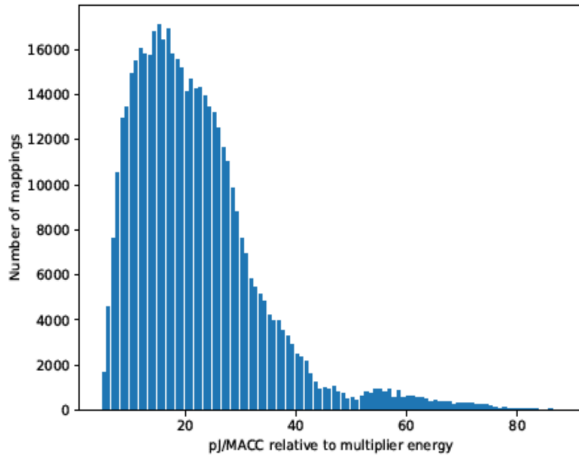
Spread: 19x in energy efficiency

Only 1 is optimal, 9 others within 1%

6,582 mappings have min. DRAM accesses but vary 11x in energy efficiency

Impact of Mappings

VGG conv3 2 Layer. Source: Timeloop



480,000 mappings shown

Spread: 19x in energy efficiency

Only 1 is optimal, 9 others within 1%

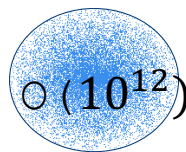
6,582 mappings have min. DRAM accesses but vary 11x in energy efficiency

1-level par.

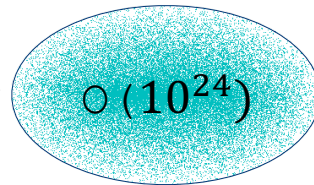
2-level par.

3-level par.

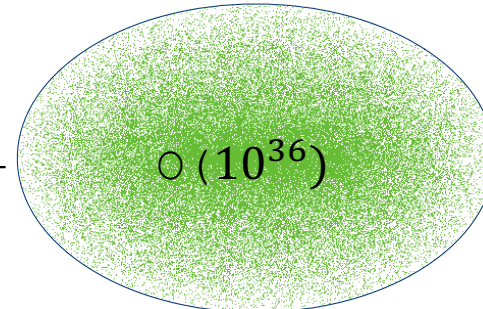
**Immense
Search
space**



+

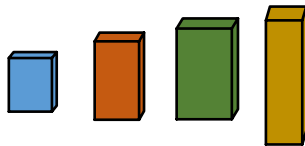


+



Exploring Mappings

- Gigantic space of potential loop orders & factorizations
- Cycle-accurate modeling of realistic dimensions and fabric sizes too slow
- Solution: use an analytic modeling



```
int i[C][Y][X]; # Input activation channels
int w[K][C][R][S]; # Filter weights (per channel pair)
int o[K][Y'][X']; # Output activation channels
```

```
for (k = 0; k < K; k++) {
  for (y = 0; y < Y'; y++) {
    for (x = 0; x < X'; x++) {
      for (c = 0; c < C; c++) {
        for (r = 0; r < R; r++) {
          for (s = 0; s < S; s++) {
            o[k][y][x] += i[c][y+r][x+s]*w[k][c][r][s];
          }
        }
      }
    }
  }
}
```

⋮

DL Operator

Mapping

HW Params

Analytical Cost Model

Latency,
Throughput,
Energy, ...

e.g.,: Timeloop (ISPASS 2019),
MAESTRO (MICRO 2019), ..

Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows
 - Fusion
 - Sparsity

Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows
 - Fusion
 - Sparsity

Not All GEMMs are Compute Bound

Not All GEMMs are Compute Bound

Even in the best case with infinite on-chip storage and large number of PEs.

Not All GEMMs are Compute Bound

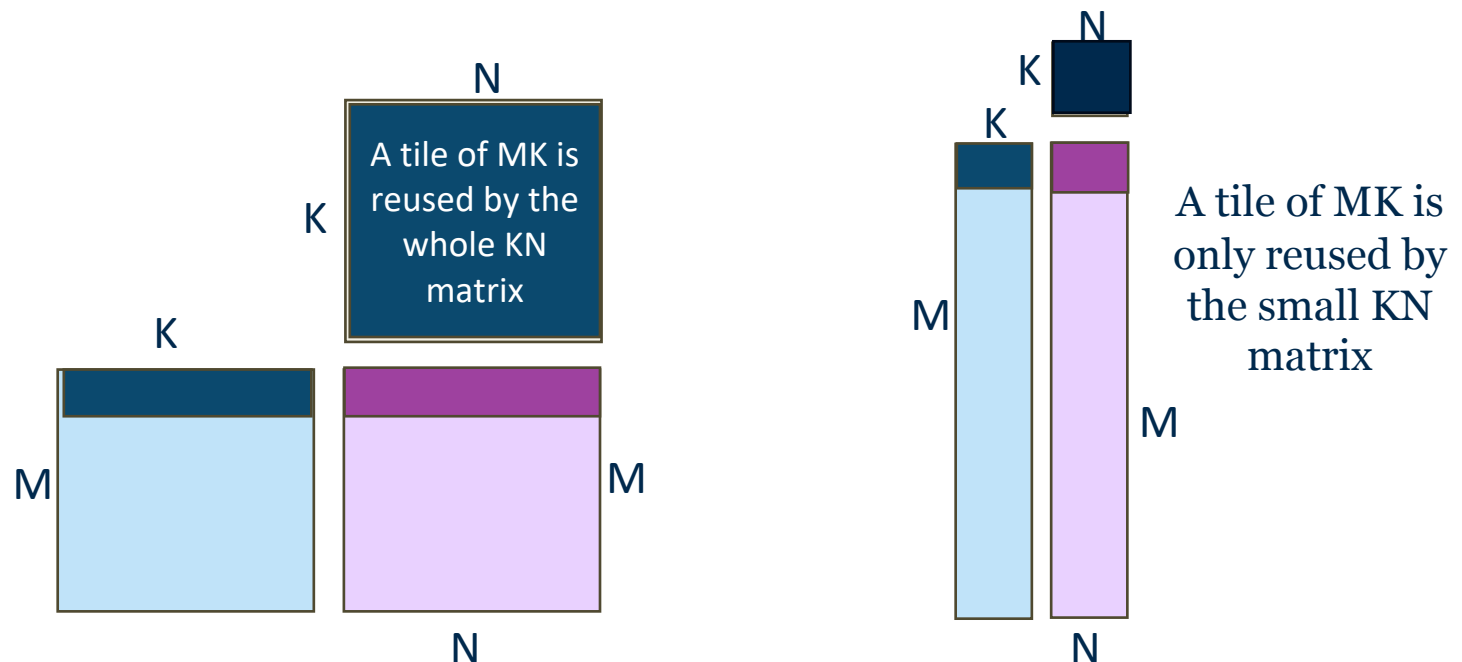
Even in the best case with infinite on-chip storage and large number of PEs.

$$AI_{best_GEMM} = \frac{M \times K \times N}{M \times K + K \times N + M \times N}$$

Not All GEMMs are Compute Bound

Even in the best case with infinite on-chip storage and large number of PEs.

$$AI_{best_GEMM} = \frac{M \times K \times N}{M \times K + K \times N + M \times N}$$



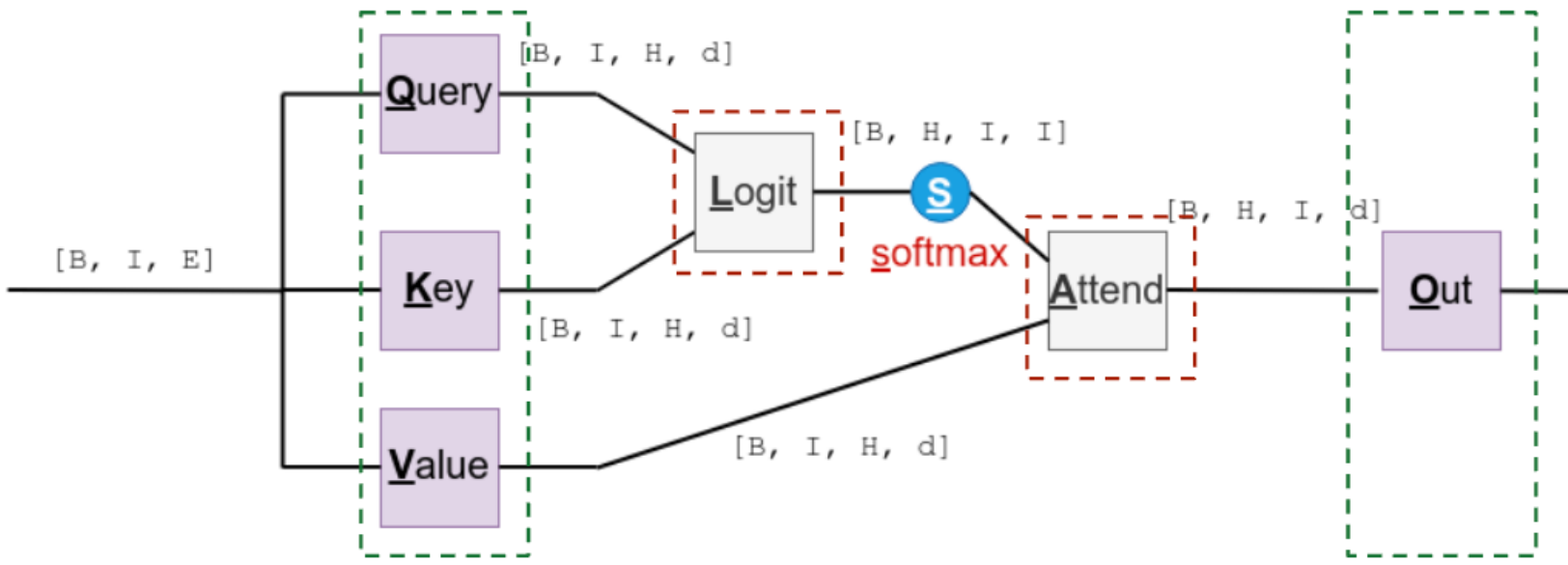
Regular GEMM ($M=1024, K=1024, N=1024$)

$$AI_{best_GEMM} = 341.33 \text{ ops/word}$$

Skewed GEMM ($M=1048576, N=32, K=32$)

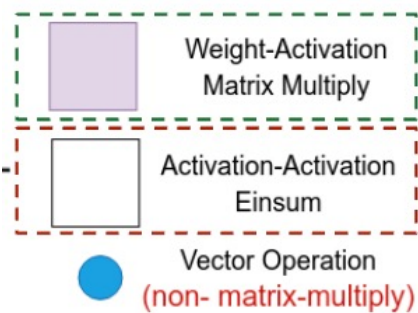
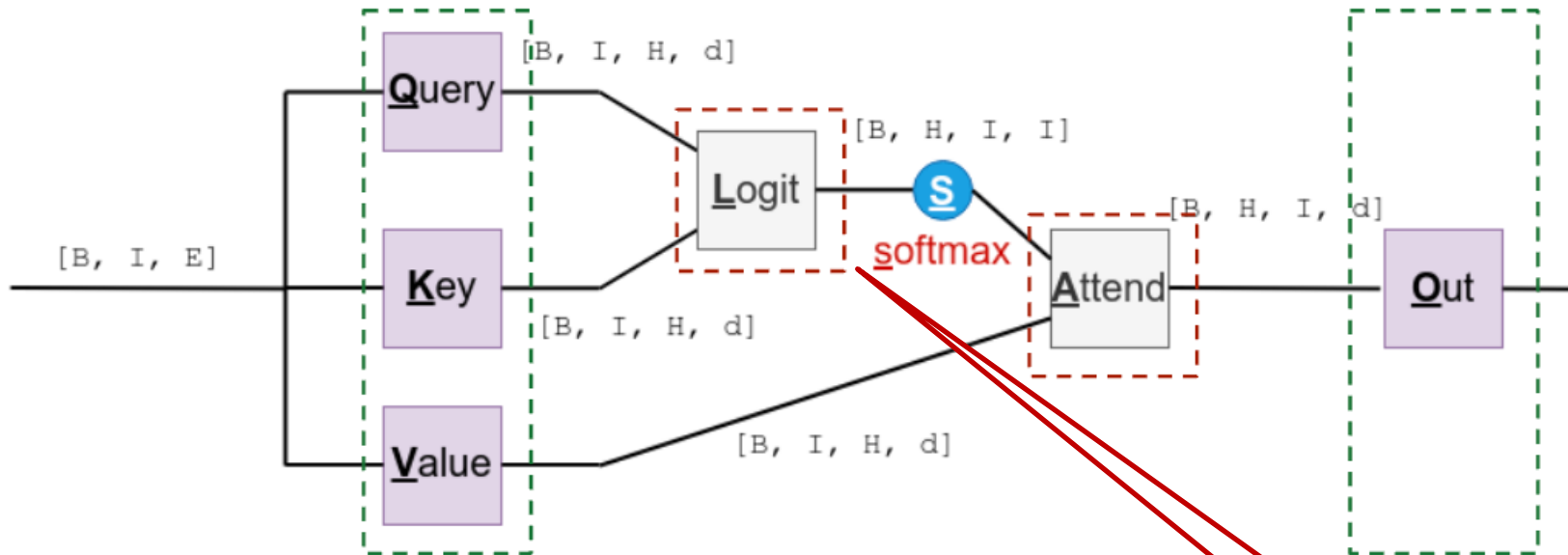
$$AI_{best_GEMM} = 16 \text{ ops/word}$$

GEMMs in Attention Layers



<div style="display: flex; align-items: center; gap: 10px;"> <div style="width: 20px; height: 20px; background-color: #d8bfd8; border: 1px solid black;"></div> <div>Weight-Activation Matrix Multiply</div> </div>	<p>Compute-bound \Rightarrow Intra-operator dataflow to improve reuse is effective</p>
<div style="display: flex; align-items: center; gap: 10px;"> <div style="width: 20px; height: 20px; background-color: white; border: 1px solid black;"></div> <div>Activation-Activation Einsum</div> </div>	<p>Memory-bound \Rightarrow Intra-operator dataflow is not effective</p>
<div style="display: flex; align-items: center; gap: 10px;"> <div style="width: 15px; height: 15px; background-color: #007bff; border-radius: 50%;"></div> <div>Vector Operation (non-matrix-multiply)</div> </div>	

GEMMs in Attention Layers

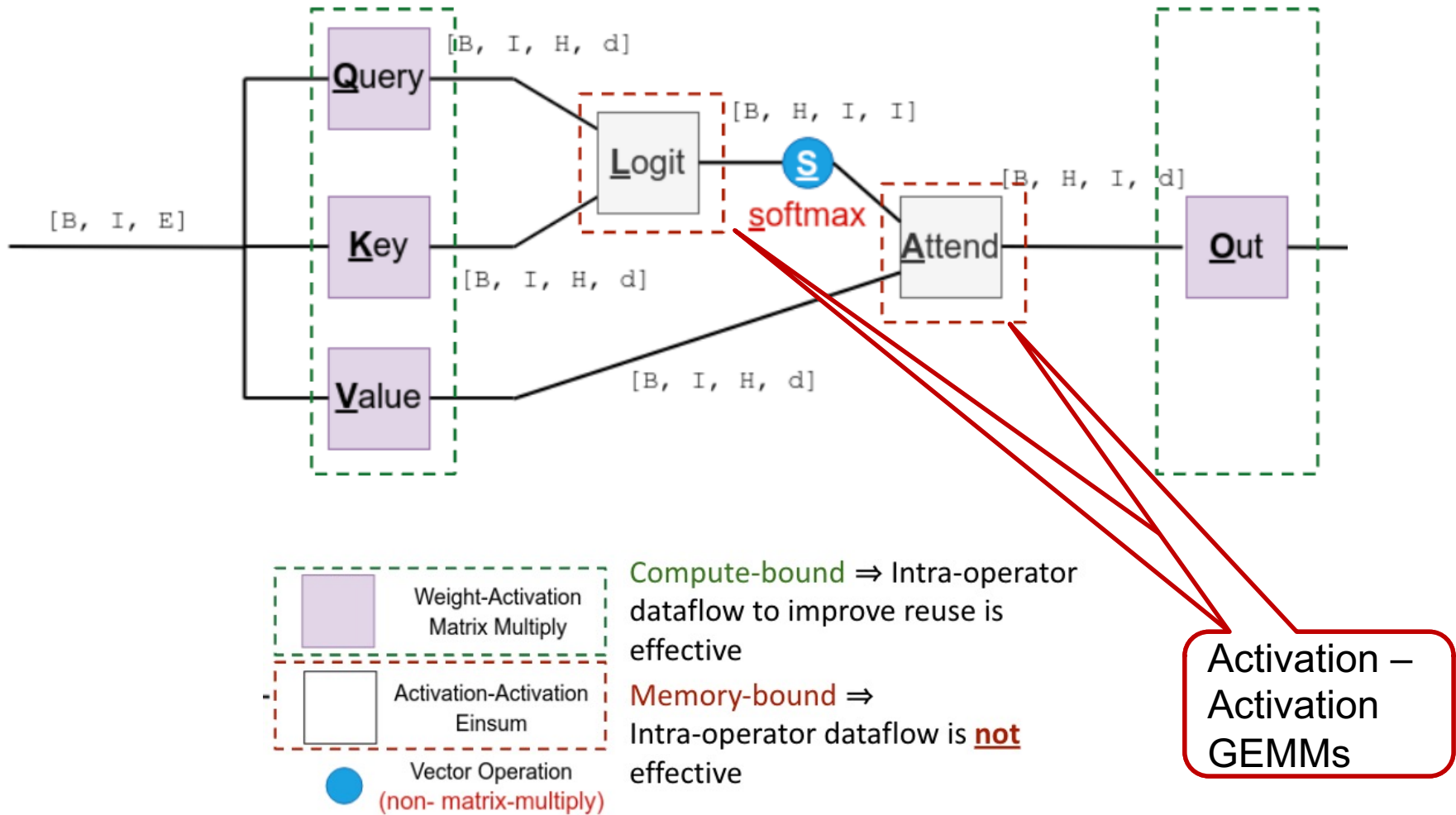


Compute-bound \Rightarrow Intra-operator dataflow to improve reuse is effective

Memory-bound \Rightarrow Intra-operator dataflow is **not** effective



GEMMs in Attention Layers



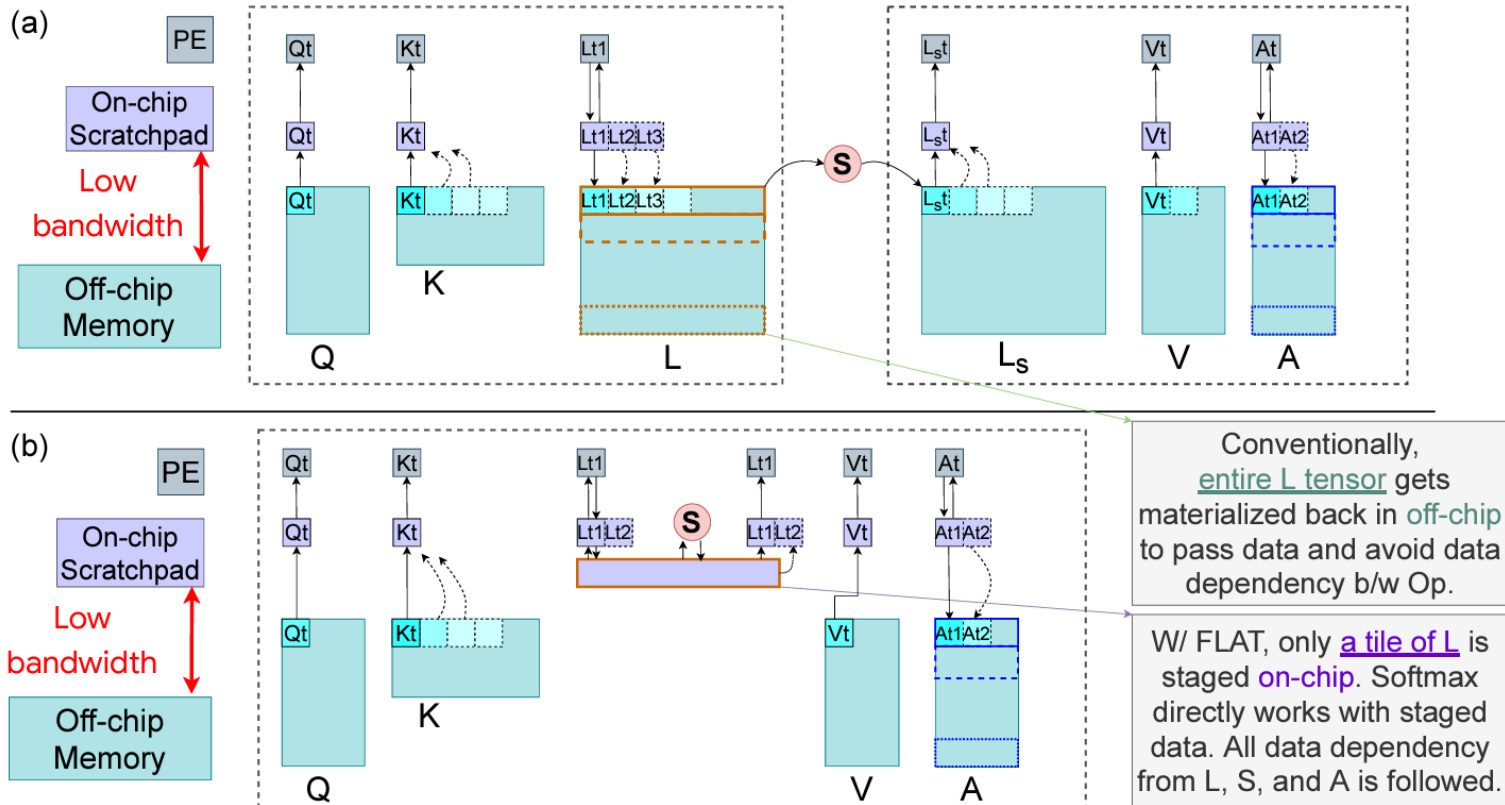
Compute-bound \Rightarrow Intra-operator dataflow to improve reuse is effective

Memory-bound \Rightarrow Intra-operator dataflow is **not** effective

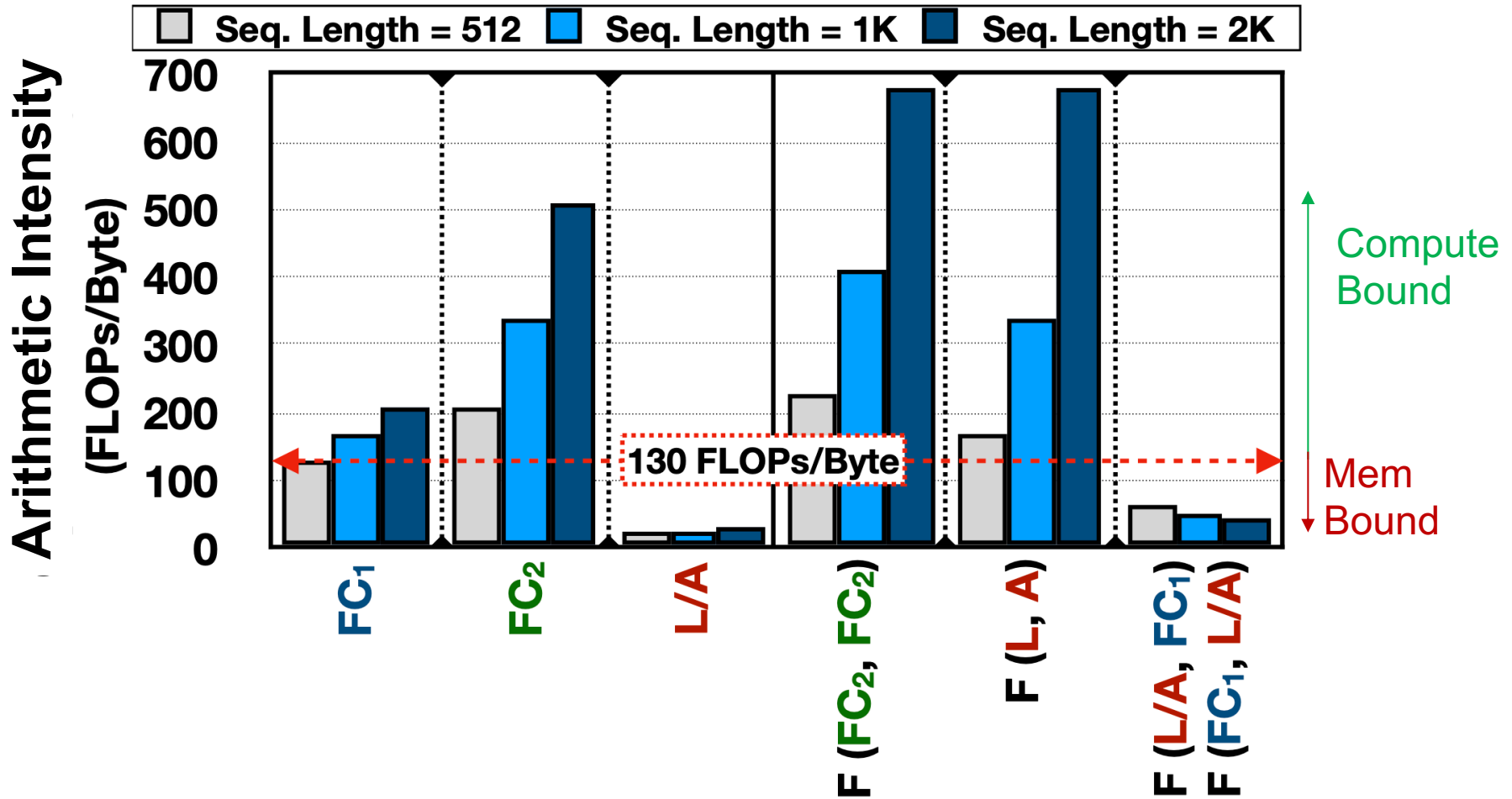
Activation – Activation GEMMs

Opportunity: Fusion

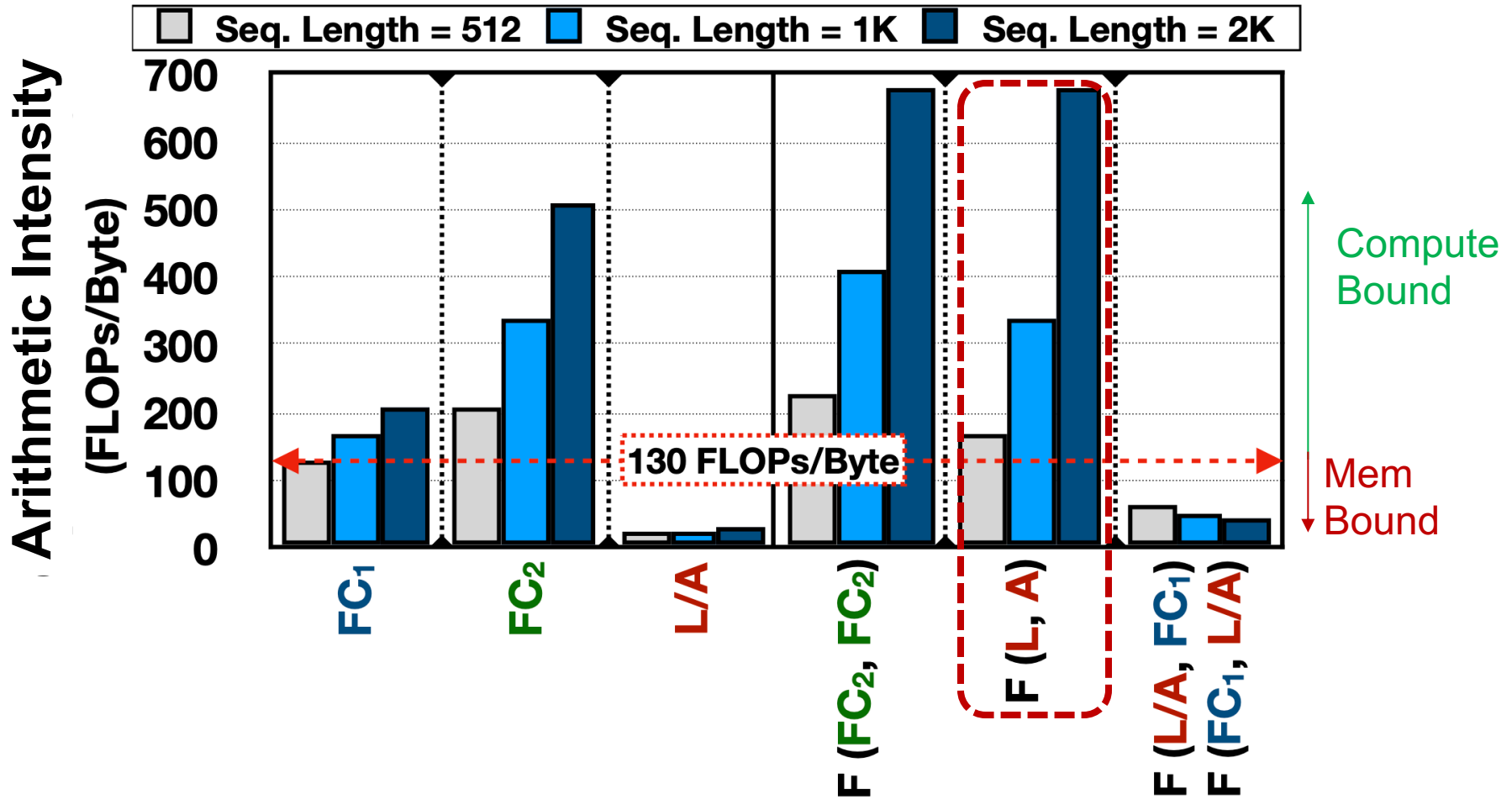
- Key Intuition: "Reuse" the intermediate output immediately



Opportunity: Fusion



Opportunity: Fusion



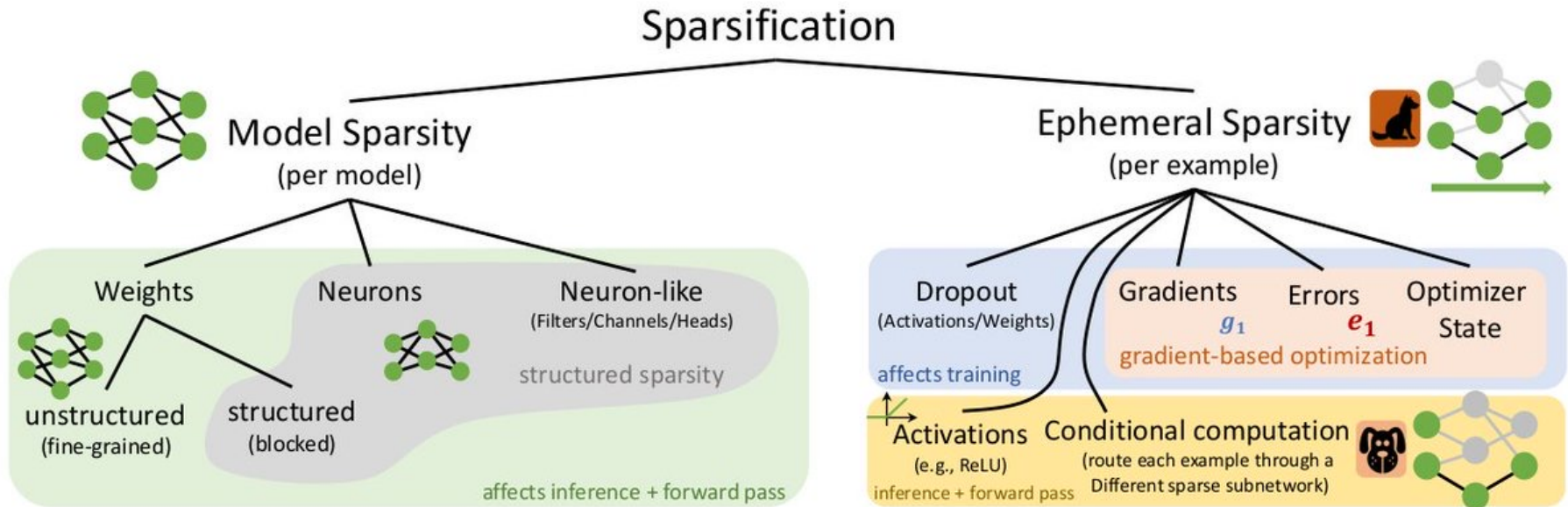
Outline

- Recap
- Dataflows for 1D Convolution
- Getting more realistic
- Advanced Dataflows
 - Fusion
 - Sparsity

Sparsity in DNNs

Source: Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks

Sparsity in DNNs



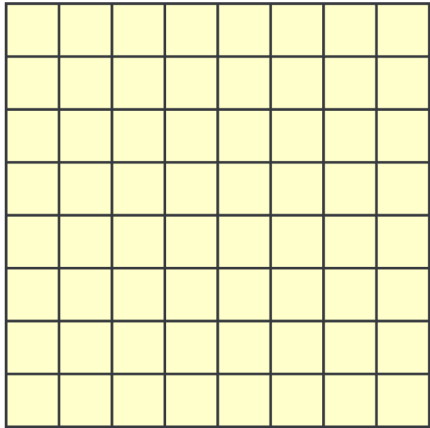
Source: *Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks*

Figure source: <https://hstor.inf.ethz.ch/sparsity-in-dl/>

10-90% sparsity across ML Models today

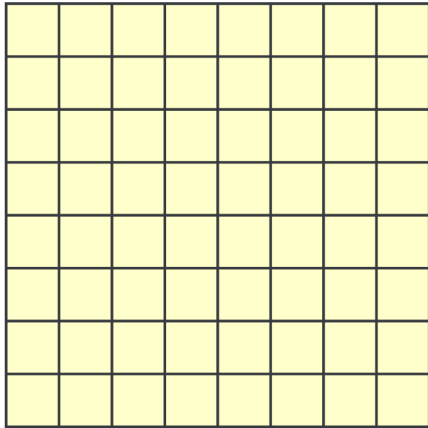
Sparsity Patterns

Sparsity Patterns

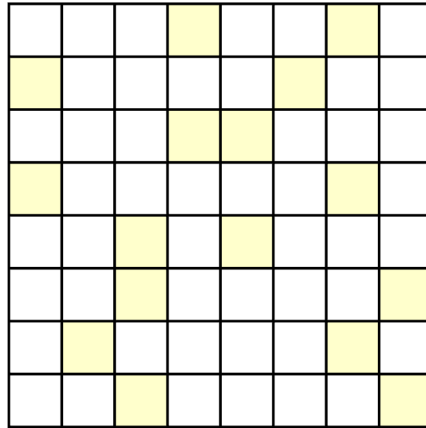


DENSE

Sparsity Patterns

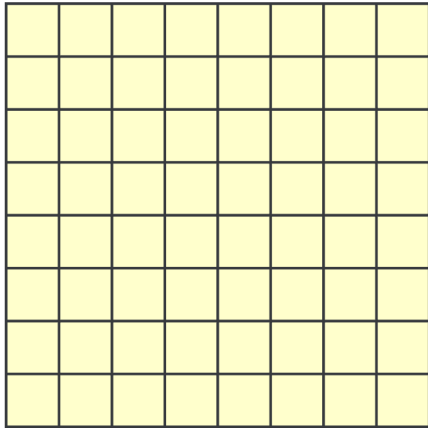


DENSE

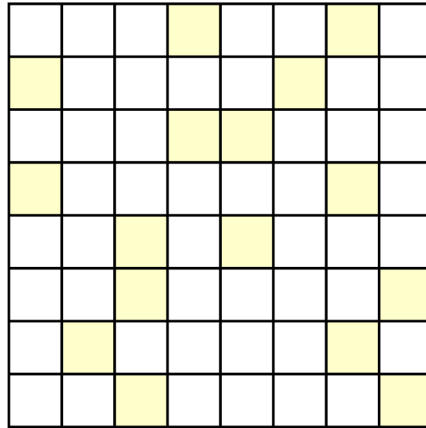


Block Balanced (Eg: N:M)

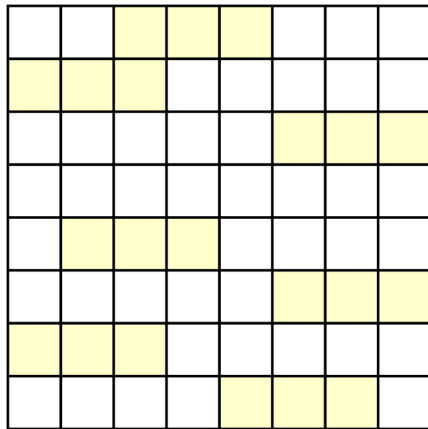
Sparsity Patterns



DENSE

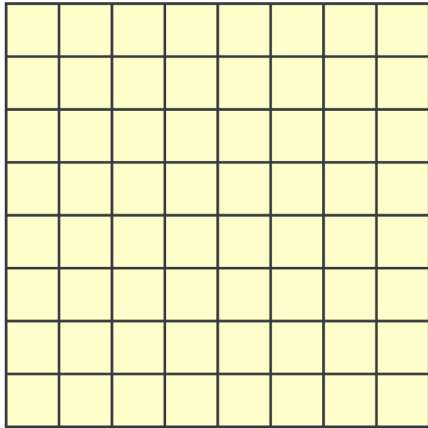


Block Balanced (Eg: N:M)

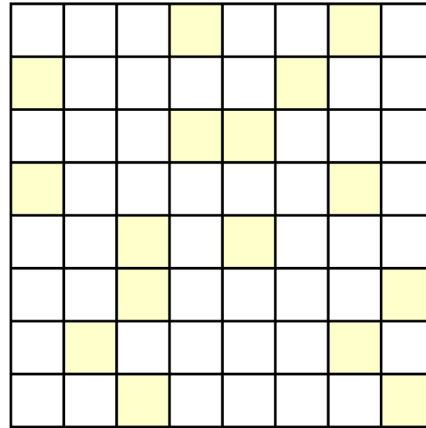


1D Blocks

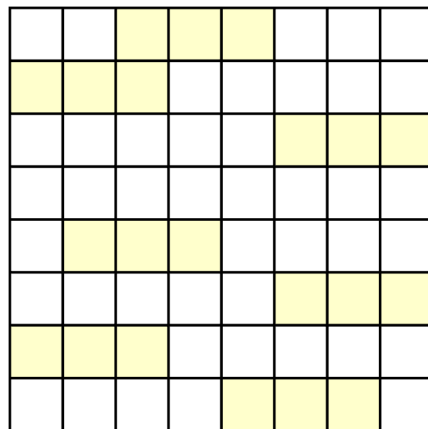
Sparsity Patterns



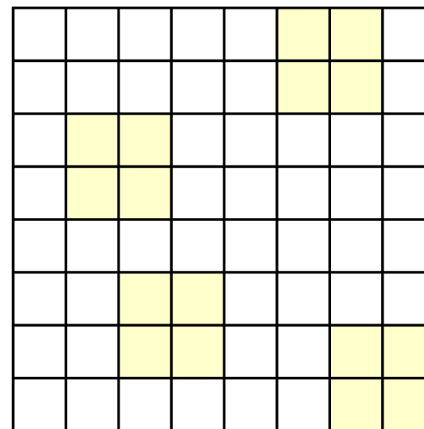
DENSE



Block Balanced (Eg: N:M)

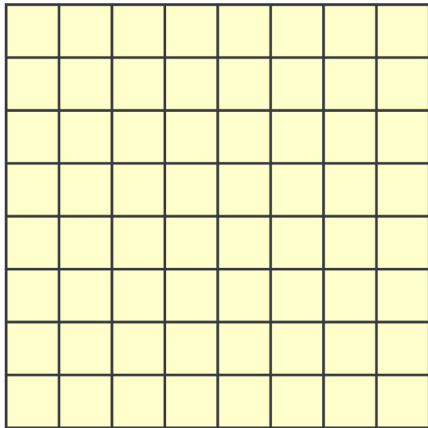


1D Blocks

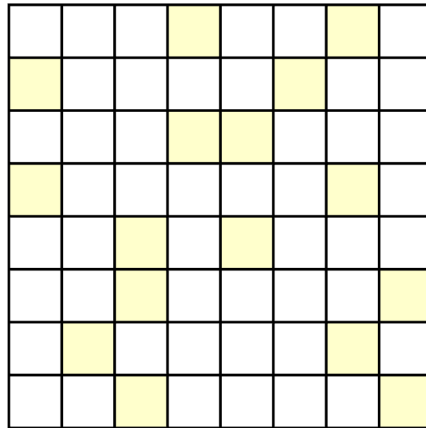


2D Blocks

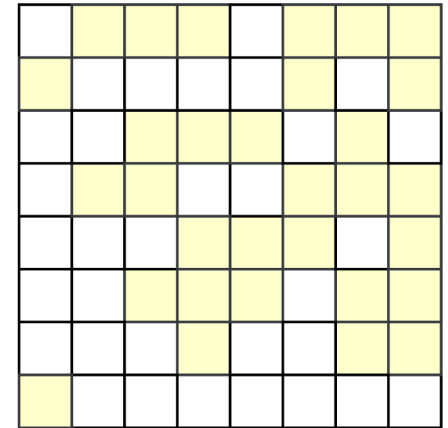
Sparsity Patterns



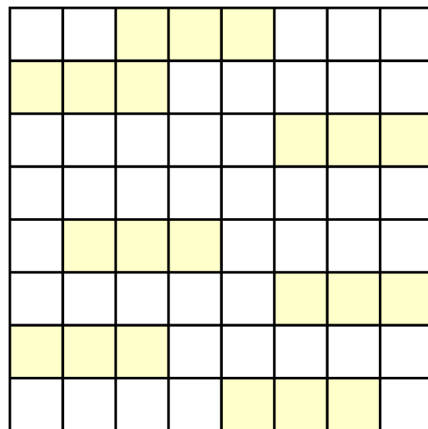
DENSE



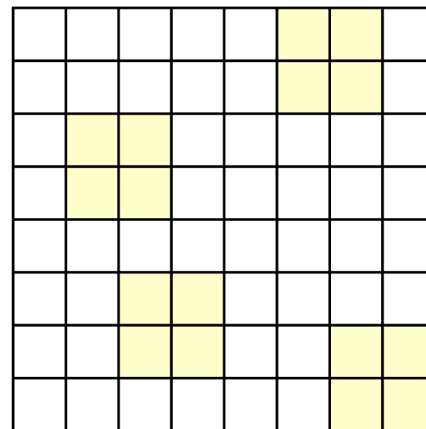
Block Balanced (Eg: N:M)



Unstructured



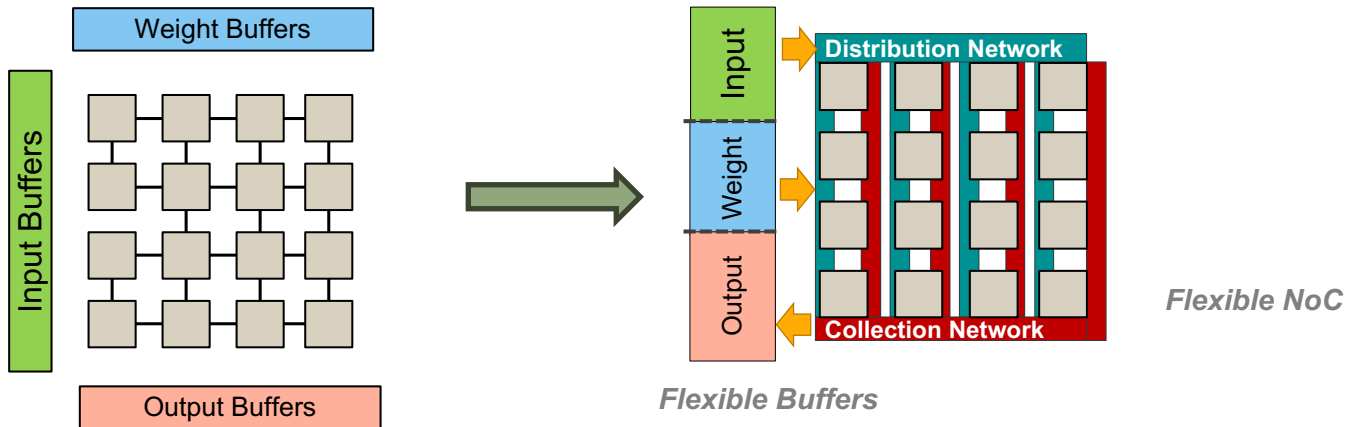
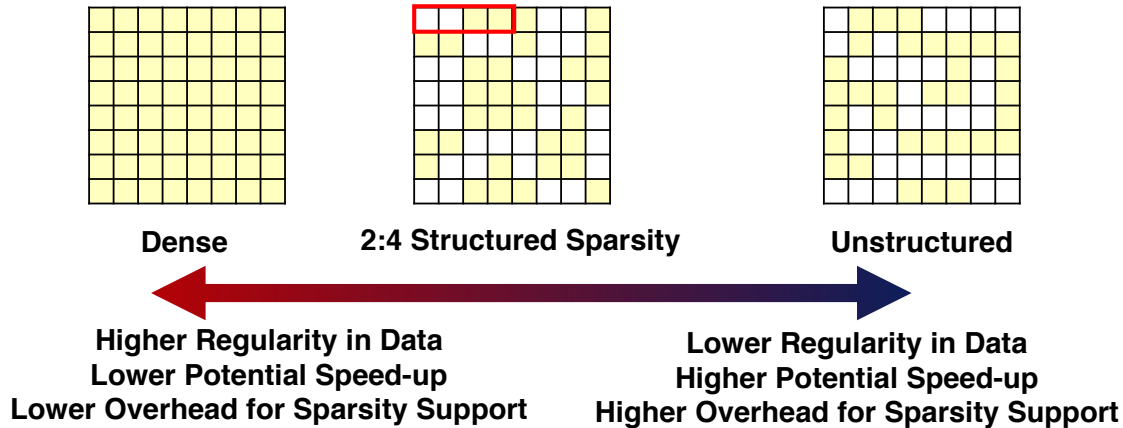
1D Blocks



2D Blocks

Sparse Accelerators

Trade-off Space



Sparse Dataflows

$$A_{MK} \times B_{KN} = C_{MN}$$

The diagram shows the multiplication of two sparse matrices, A_{MK} and B_{KN} , resulting in a sparse matrix C_{MN} .

Matrix A_{MK} (2x4) has non-zero elements at $(0,1)$, $(1,0)$, $(1,2)$, and $(1,3)$.

Matrix B_{KN} (4x3) has non-zero elements at $(0,1)$, $(0,2)$, $(1,0)$, $(1,2)$, $(2,0)$, $(3,0)$, $(3,1)$, and $(3,2)$.

The resulting matrix C_{MN} (2x3) has non-zero elements at $(0,0)$, $(0,2)$, $(1,0)$, $(1,1)$, and $(1,2)$.

- Inner Product
- Outer Product
- Gustavson's

Sparse Dataflows

$$A_{MK} \times B_{KN} = C_{MN}$$

The diagram shows the multiplication of two sparse matrices, A_{MK} and B_{KN} , resulting in a sparse matrix C_{MN} . The matrices are represented as grids of cells. A_{MK} is a 2x4 grid with non-zero elements A_{01} , A_{10} , A_{12} , and A_{13} . B_{KN} is a 4x3 grid with non-zero elements B_{01} , B_{02} , B_{10} , B_{12} , B_{20} , B_{30} , B_{31} , and B_{32} . The resulting matrix C_{MN} is a 2x3 grid with non-zero elements C_{00} , C_{02} , C_{10} , C_{11} , and C_{12} .

- Inner Product
- Outer Product
- Gustavson's

Active area of research!

Thank you!