Hardware for Machine Learning

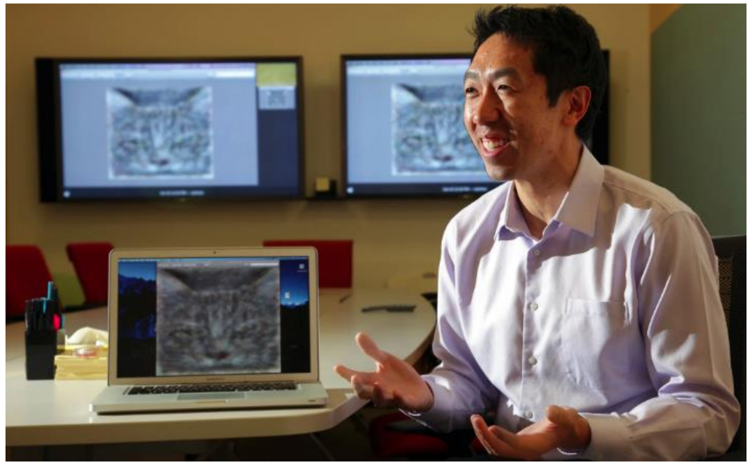
Lecture 6: Dataflow

Sophia Shao

How Many Computers to Identify a Cat? 16,000

“Presented with 10 million digital images found in YouTube videos, what did Google’s brain do? What millions of humans do with YouTube: looked for cats.”

“Nvidia’s AI journey started at Joanie’s Cafe in Palo Alto, California, in 2010. “We got into AI after a breakfast meeting I had with Andrew Ng,” says Dally. At that breakfast, Ng, a well known AI researcher who was working with Google Brain at the time, explained how Google was training AI systems to recognise photos of cats with the help of 16,000 central processing units, or CPUs. After pointing out next to no one has 16,000 CPUs at their disposal, Dally laid down a challenge. “I bet we could do this with way fewer GPUs,” he said to Ng.”



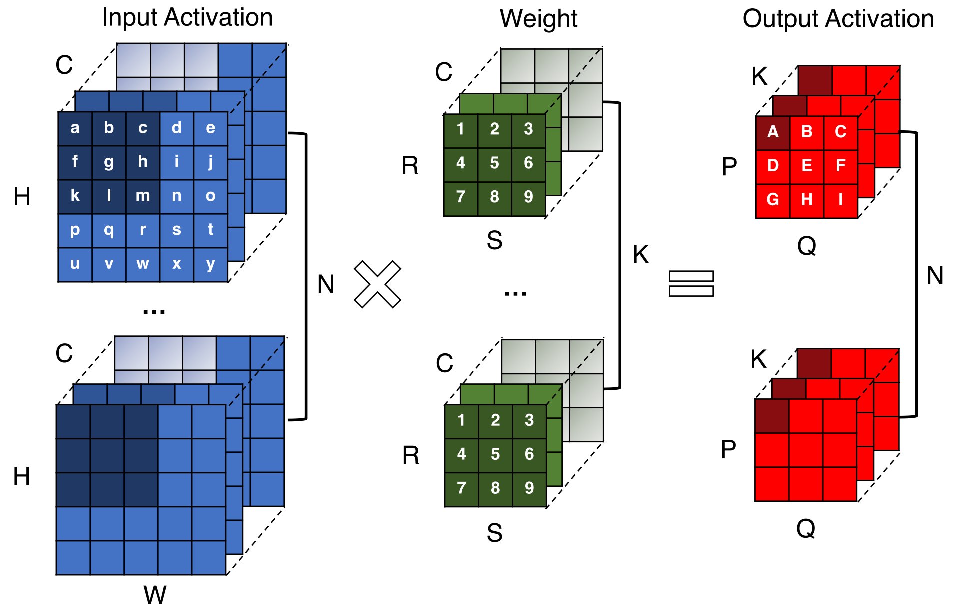
https://www.nytimes.com/2012/06/26/technology/in-a-big-network-of-computers-evidence-of-machine-learning.html

https://www.dclsearch.com/blog/2019/02/nvidias-got-a-cunning-plan-to-keep-powering-the-ai-revolution

Review

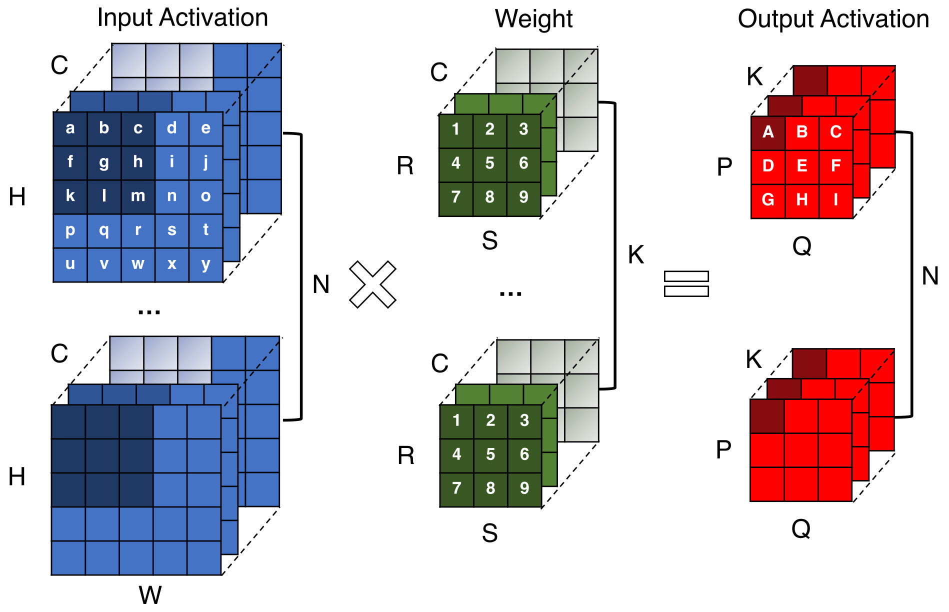
- Deep neural networks typically have a sequence of convolutional, fully-connected, pooling, batch normalization, and activation layers.
- Convolution is one of the fundamental kernel in DNNs.
  - 2-D convolution
  - Stride and padding
  - 3-D convolution with input/output channels
  - Batch size
- Convolution can be calculated in different ways.
  - Direct, GEMM, FFT-based, Winograd-based.

Convolution Loop Nest



```
for (n=0; n<N; n++) {                                    } for each output activation
  for (k=0; k<K; k++) {
    for (p=0; p<P; p++) {
      for (q=0; q<Q; q++) {
        OA[n][k][p][q]= 0;
        for (r=0; r<R; r++) {                            } convolution window
          for (s=0; s<S; s++) {
            for (c=0; c<C; c++) {
              h = p * stride - pad + r;
              w = q * stride - pad + s;
              OA[n][k][p][q] +=
                            IA[n][c][h][w]
                            * W[k][c][r][s];
            }
          }
        }
        OA[n][k][p][q]= Activation(OA[n][k][p][q]);
      }
    }
  }
}
```

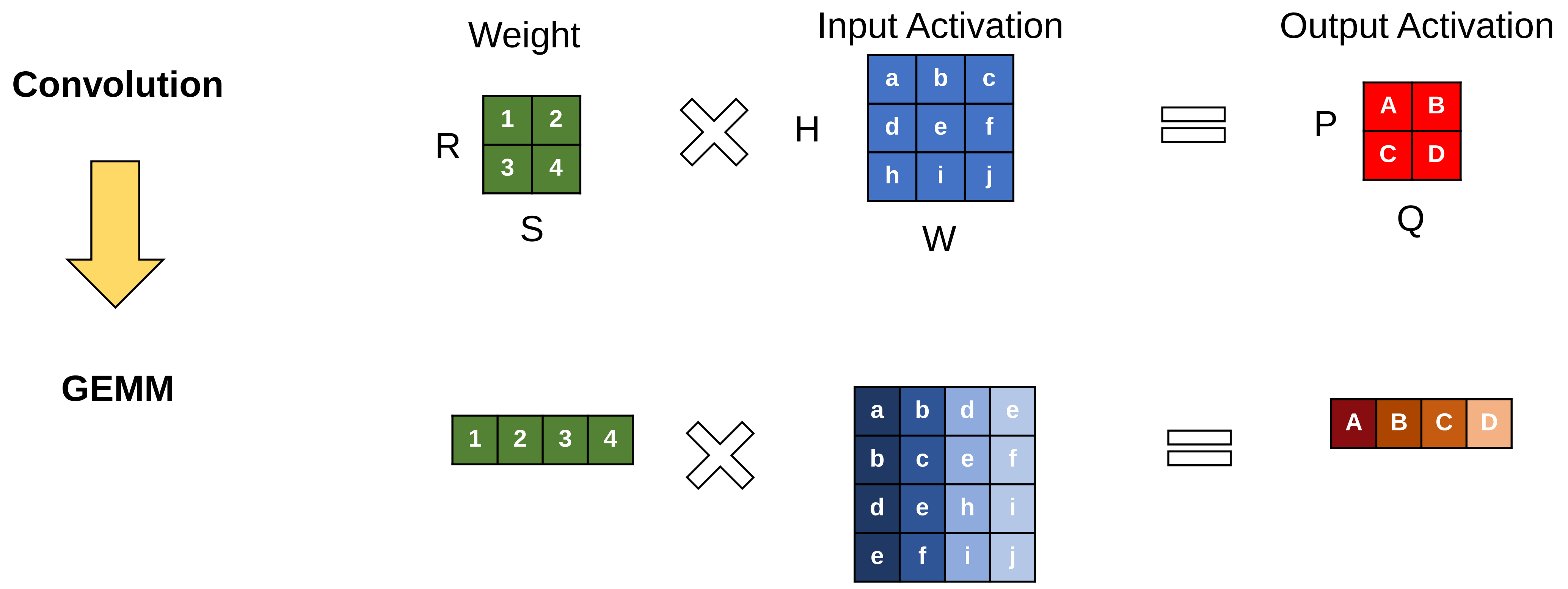
Option 1: Direct Convolution



```
for (n=0; n<N; n++) {
  for (k=0; k<K; k++) {
    for (p=0; p<P; p++) {
      for (q=0; q<Q; q++) {
        OA[n][k][p][q]= 0;
        for (r=0; r<R; r++) {
          for (s=0; s<S; s++) {
            for (c=0; c<C; c++) {
              h = p * stride - pad + r;
              w = q * stride - pad + s;
              OA[n][k][p][q] +=
                                IA[n][c][h][w]
                              * W[k][c][r][s];
            }
          }
        }
        OA[n][k][p][q]= Activation(OA[n][k][p][q]);
      }
    }
  }
}
```

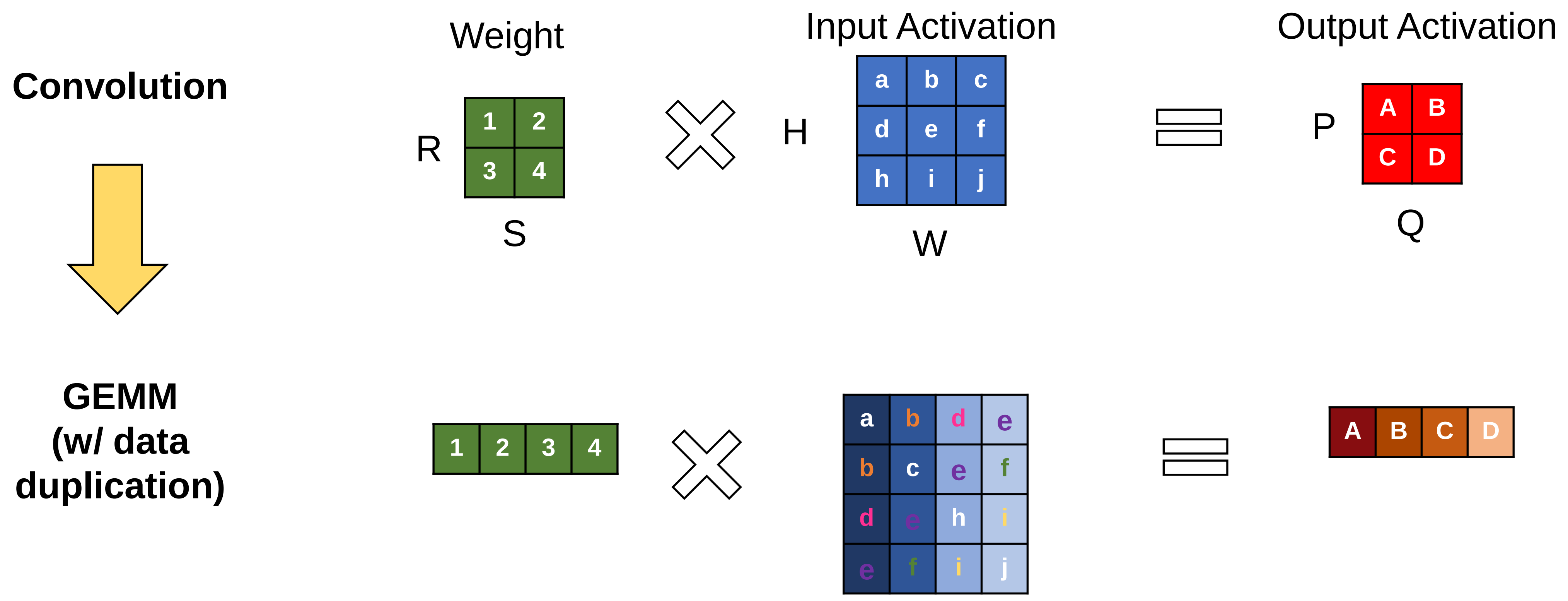
Option 2: GEMM

- Converting convolution to GEMM via **im2col**



Option 2: GEMM

- Converting convolution to GEMM via **im2col**



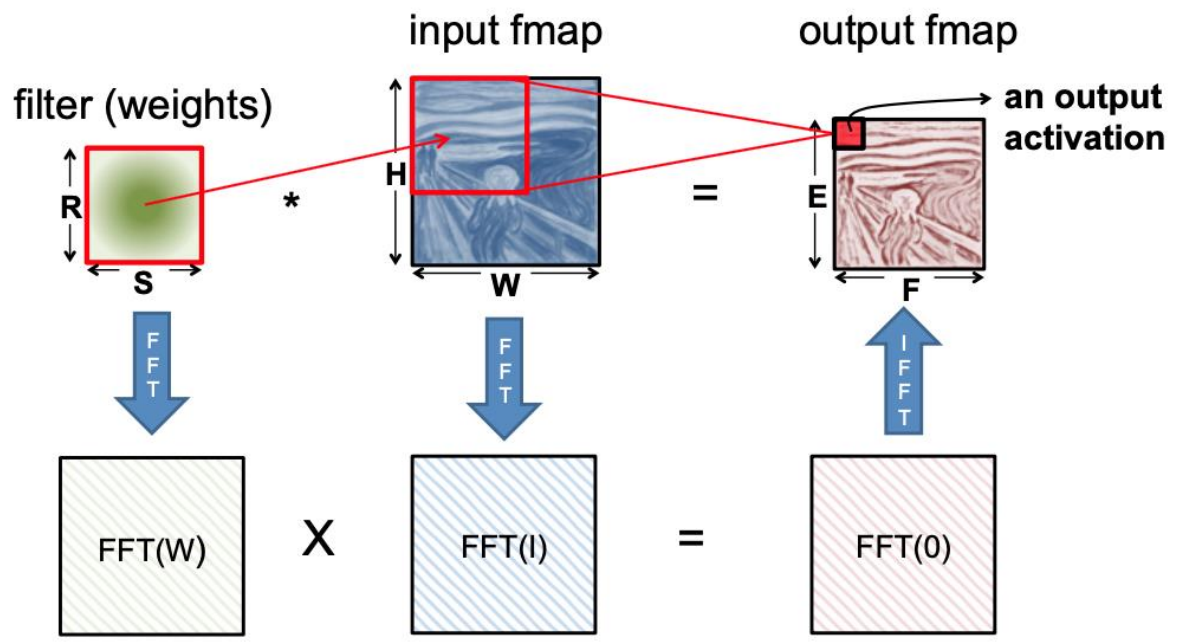
Option 3: FFT-based Convolution

- **Convolution theorem**: convolution in the time domain is equivalent to point-wise multiply in the frequency domain.

$$f * g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\}$$

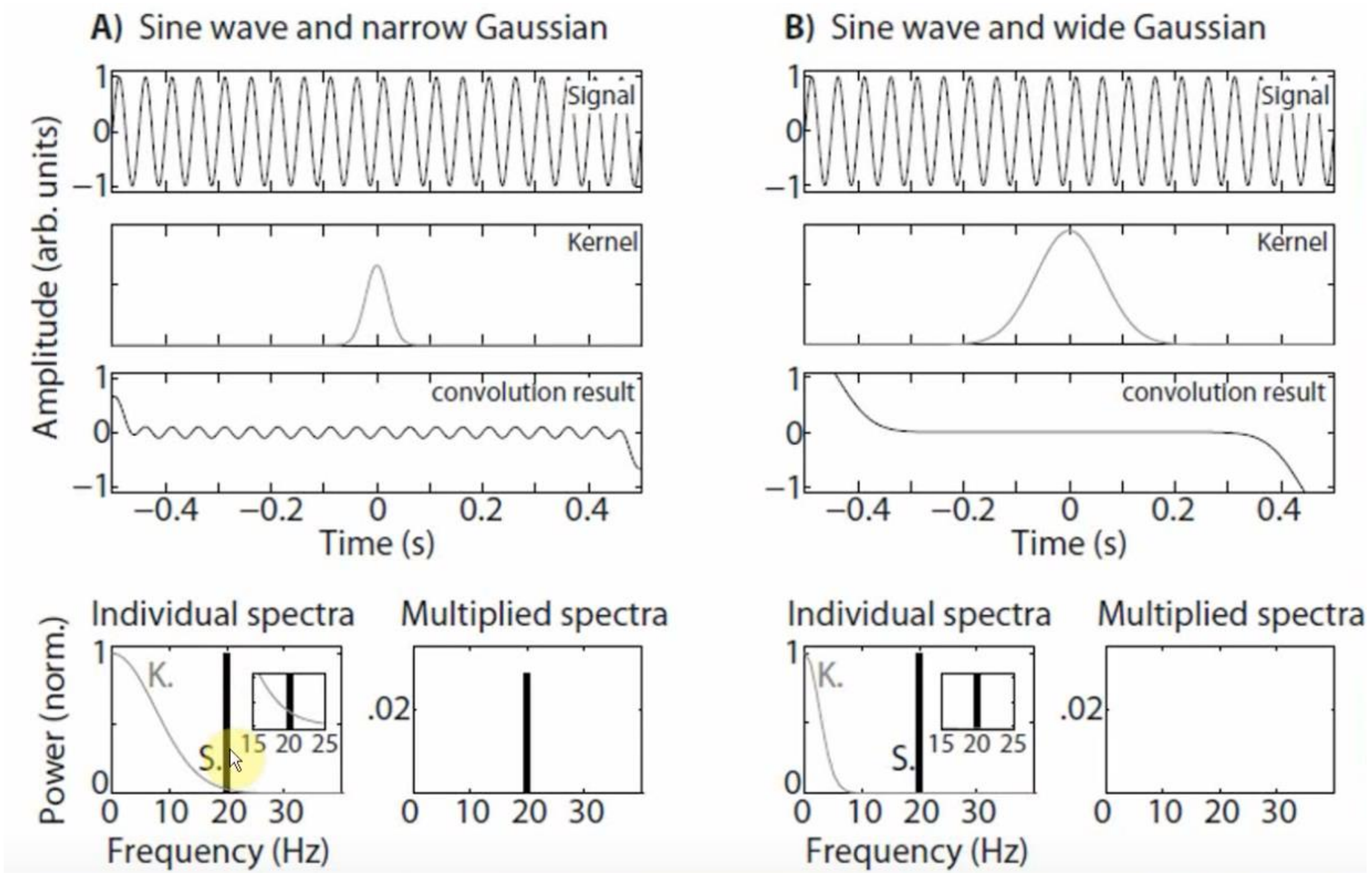
$\mathcal{F}\{f\}$ and $\mathcal{F}\{g\}$ are the Fourier transforms of f and g

The asterisk denotes convolution, not multiplication.



Eyeriss tutorial

Option 3: FFT-based Convolution



Option 3: FFT-based Convolution

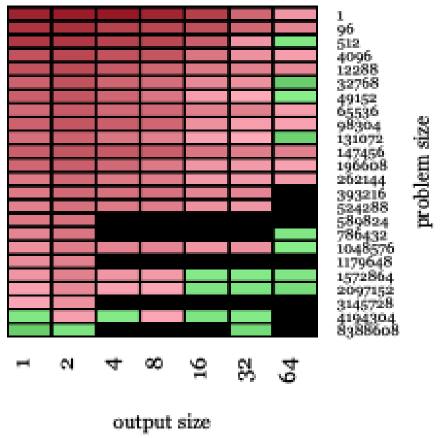


Figure 1: 3 × 3 kernel (K40m)

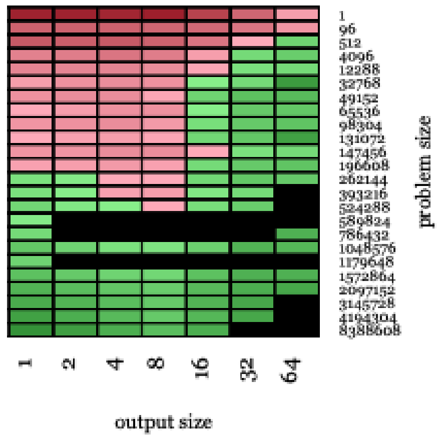


Figure 2: 5 × 5 kernel (K40m)

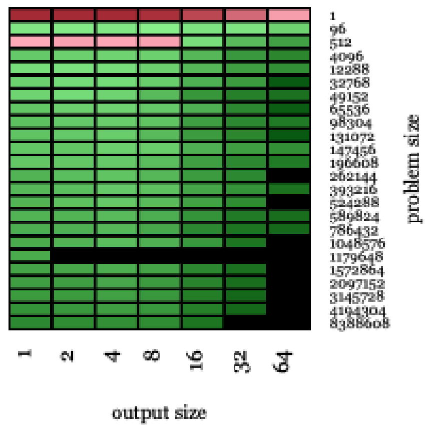


Figure 5: 11 × 11 kernel (K40m)

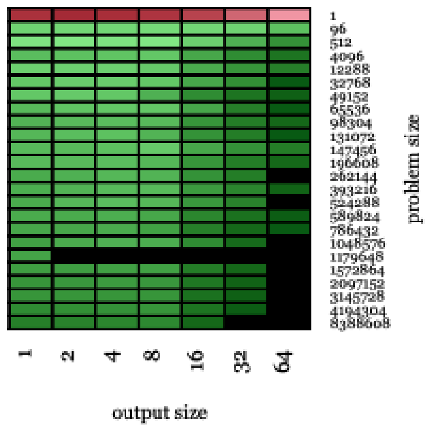


Figure 6: 13 × 13 kernel (K40m)

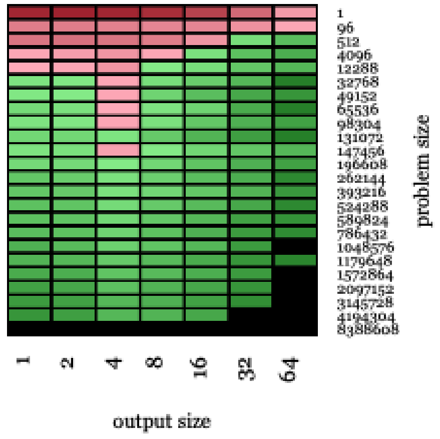


Figure 3: 7 × 7 kernel (K40m)

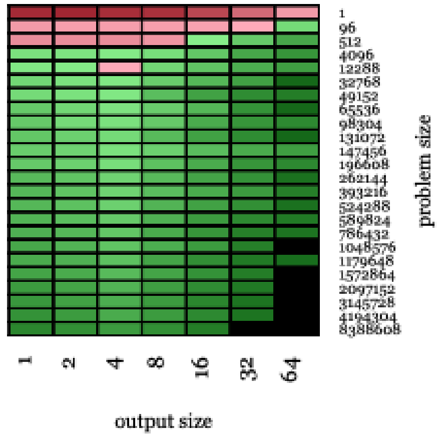
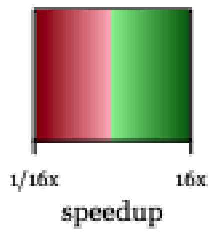


Figure 4: 9 × 9 kernel (K40m)



Fast Convolutional Nets with fbfft: A GPU Performance Evaluation

Option 4: Winograd Transform

- Re-association of intermediate values to reduce # of multiplications.
- Works well for 3x3 convolution.

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} \tag{5}$$

where

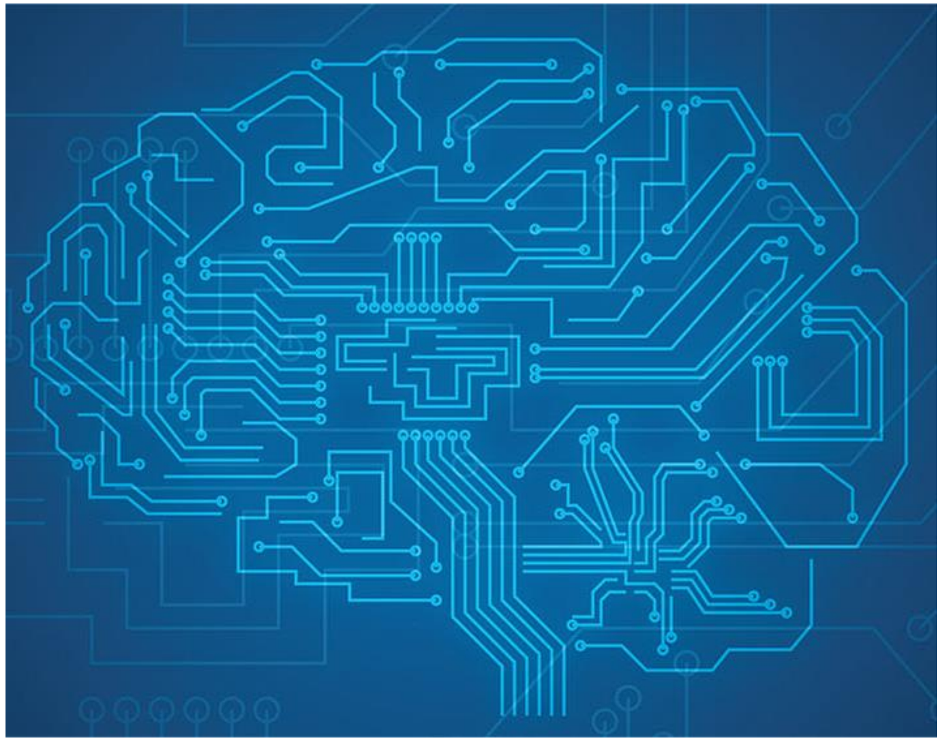
$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$
$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

Before: 6 MULs, 4 ADDs
After:
- IA (d): 4 ADDs
- W (g): 3 ADDs, 2 MULs
- OA (m): 4 MULs, 4 ADDs

$$Y = A^T\left[(Gg) \odot (B^T d)\right] \tag{6}$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$
$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \tag{7}$$
$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$
$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$$
$$d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$$

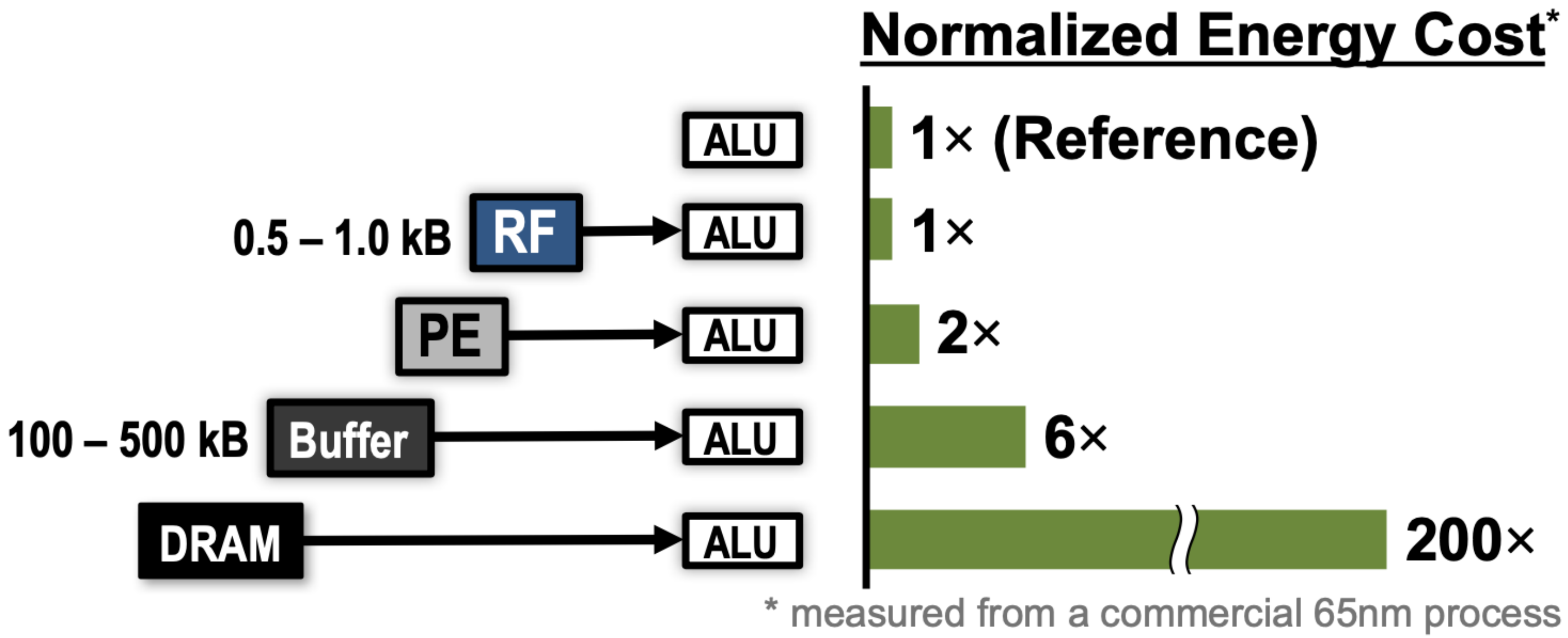
Fast Algorithms for Convolutional Neural Networks



Dataflow

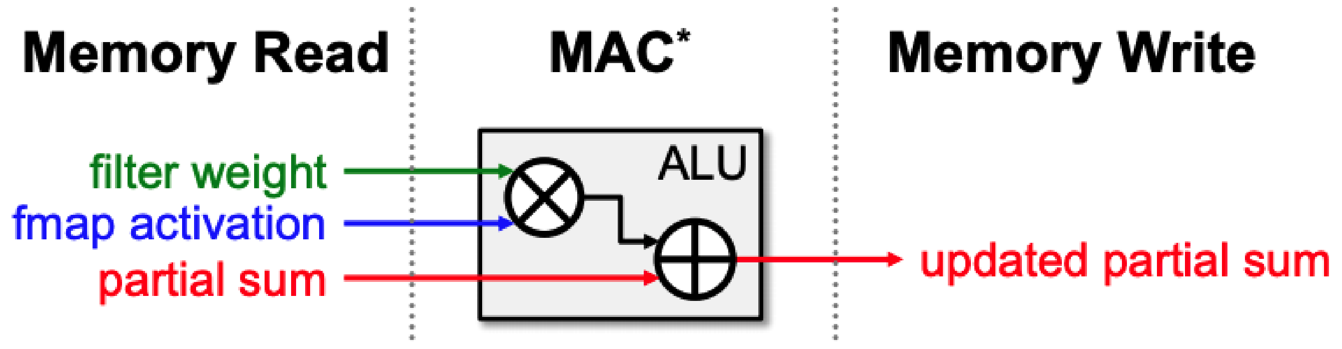
- Core Principles:
  - Locality
  - Parallelism
- Dataflow Taxonomy
  - Output-stationary
  - Weight-stationary

Locality: Data Movement Cost



Eyeriss Tutorial

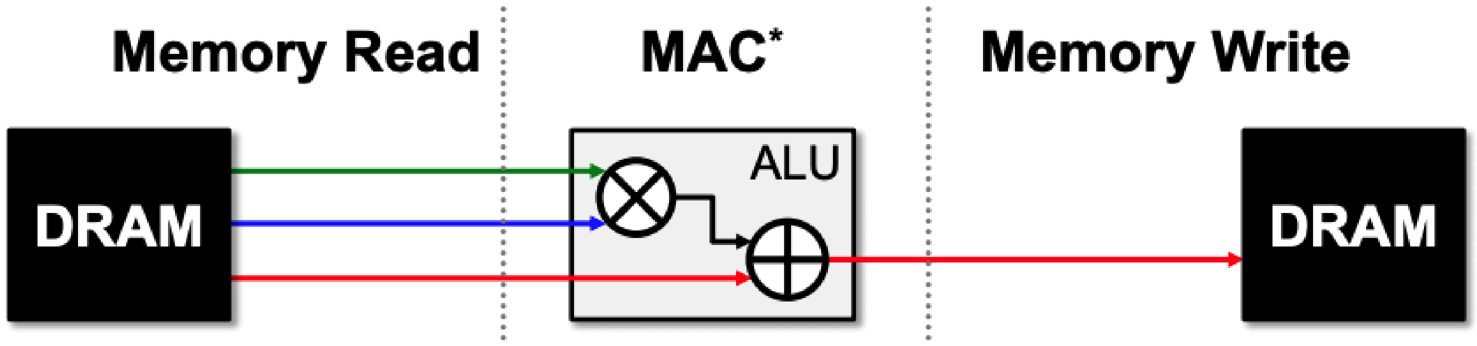
Locality: Memory Access in DNNs



* multiply-and-accumulate

Eyeriss Tutorial

Locality: Memory Access in DNNs



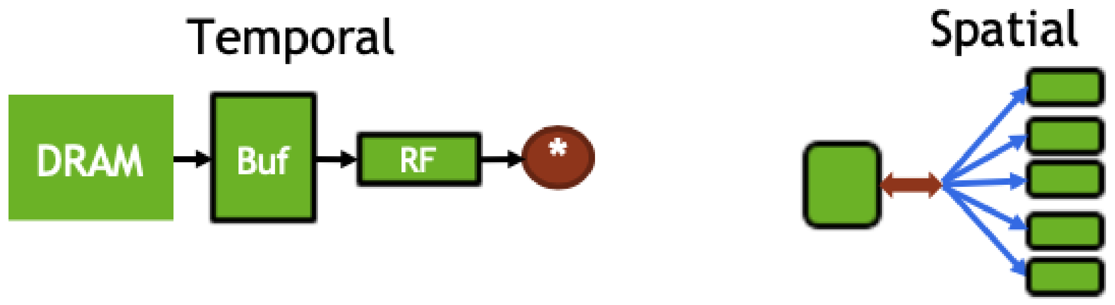
* multiply-and-accumulate

- Worst case: all memory R/W are DRAM accesses
- Example: AlexNet [NIPS 2012] has 724M MACs -> 2896M DRAM accesses required.

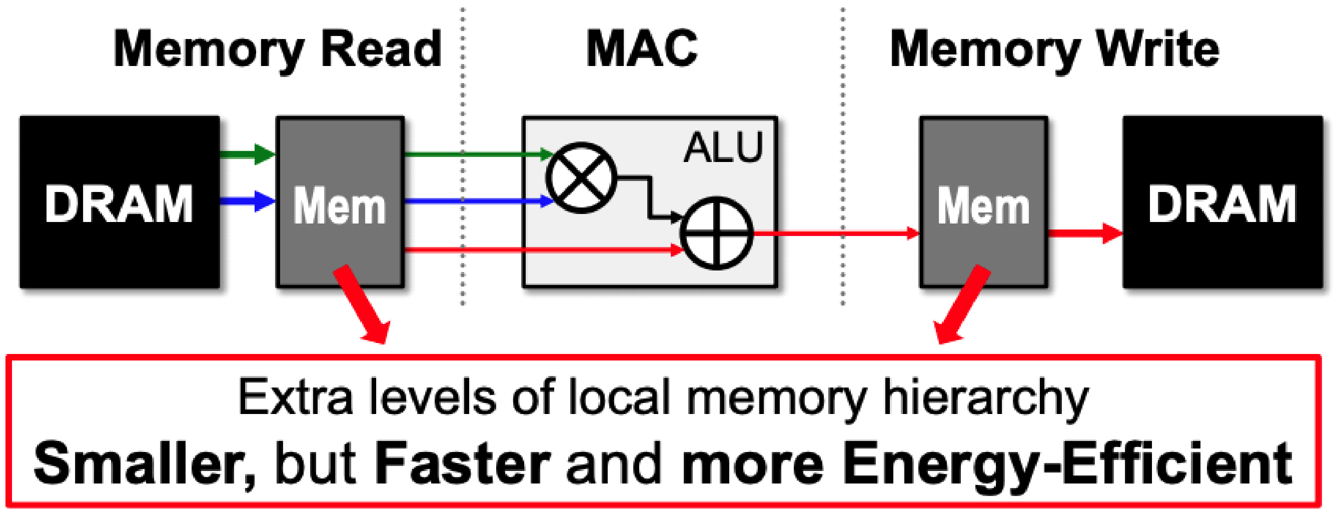
Eyeriss Tutorial

Locality: Temporal and Spatial Reuse

- **Temporal** reuse: the same data is used more than once over time by the same consumer.
- **Spatial** reuse: the same data is used by **more than one consumer** at **different spatial locations** of the hardware.



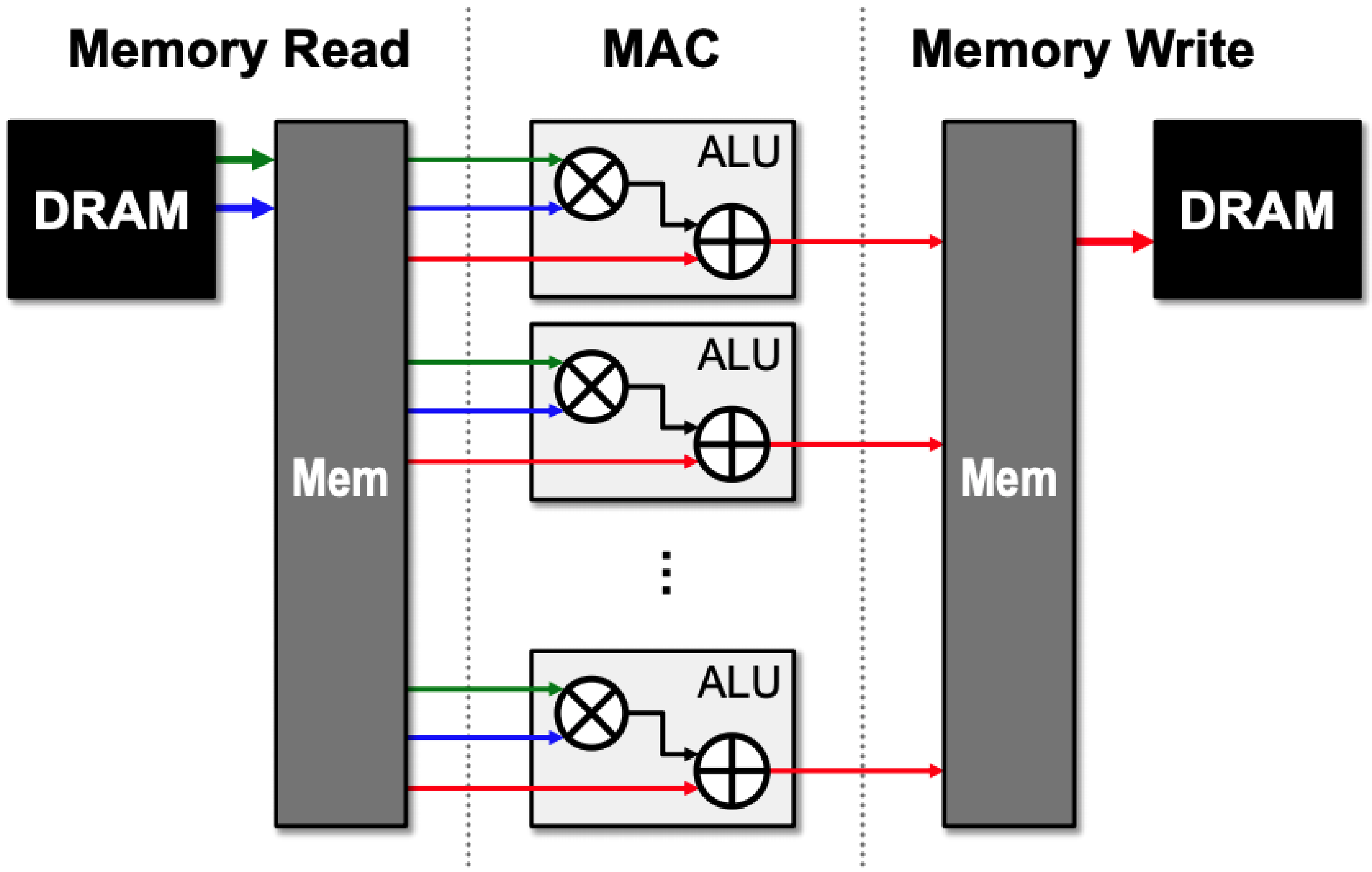
Locality: Memory Hierarchy


Memory Read
MAC
Memory Write
DRAM
Mem
ALU
Mem
DRAM
Extra levels of local memory hierarchy
Smaller, but **Faster** and **more Energy-Efficient**

- **Temporal** reuse: the same data is used more than once over time by the same consumer.

Eyeriss Tutorial

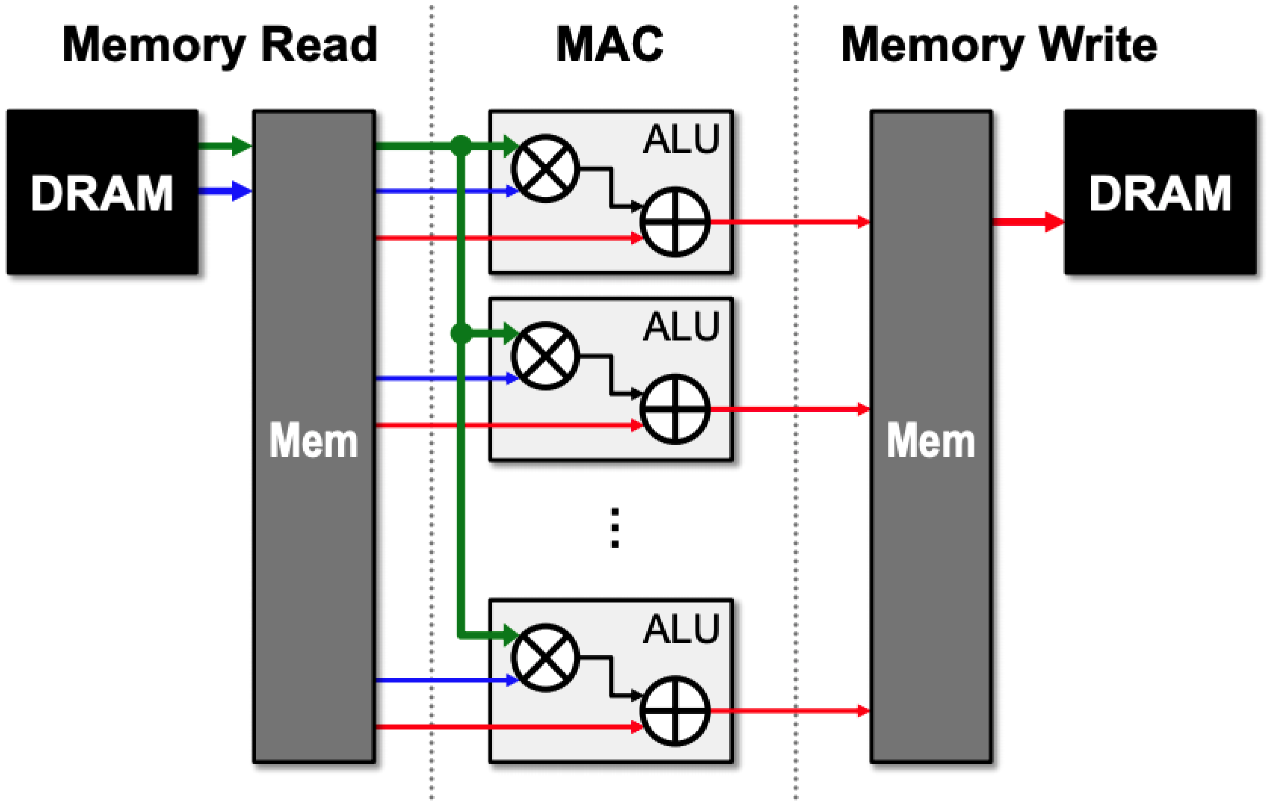
Parallelism: Parallel MAC Units



Eyeriss Tutorial

Parallelism: Spatial Data Reuse

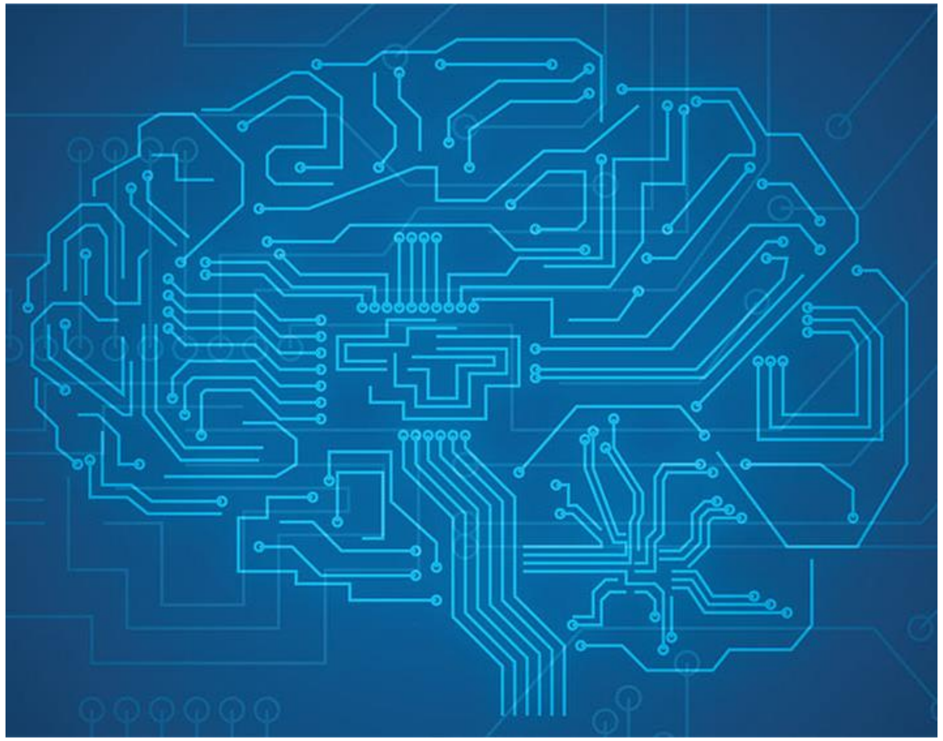
- **Spatial** reuse: the same data is used by **more than one consumer** at **different spatial locations** of the hardware.



Eyeriss Tutorial

Administrivia

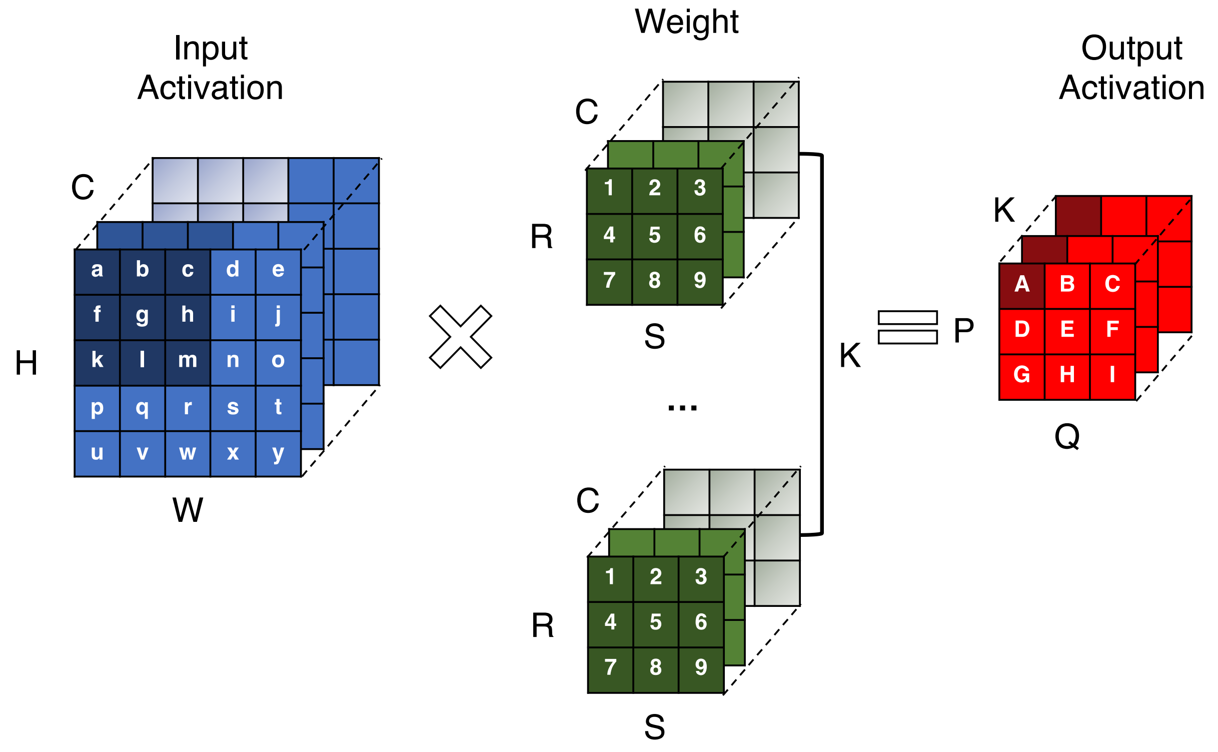
- Lab 2 out today.
  - More involved than Lab 1.
  - Start early.



Dataflow

- Core Principles:
  - Locality
  - Parallelism
- Dataflow Taxonomy
  - Output-stationary
  - Weight-stationary

Data Reuse in DNN

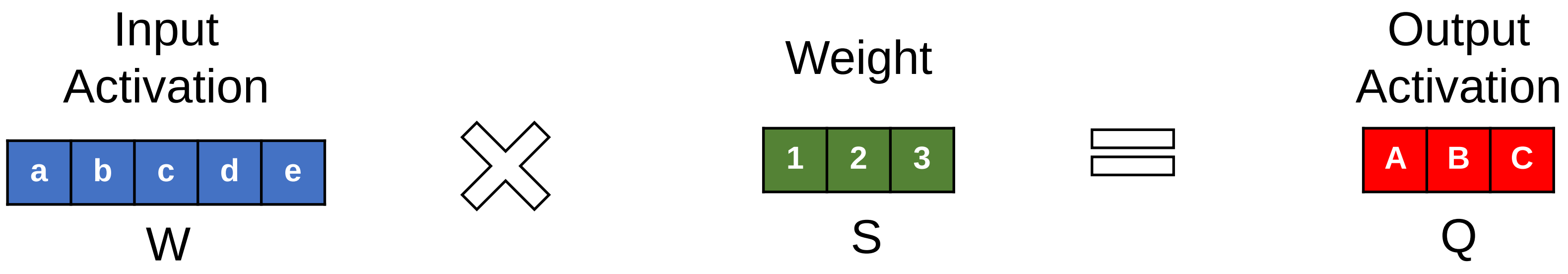


- Total # of MACs:
  - RSCPQK
- Input Activation
  - Size: HWC
  - Reuse:~RSK
- Weight
  - Size: RSCK
  - Reuse: PQ
- Output Activation
  - Size: PQK
  - Reuse: RSC

Dataflow

- Defines the execution order of the DNN operations in hardware.
  - Computation order
  - Data movement order
- Loop nest is a compact way to describe the execution order, i.e., dataflow, supported in hardware.
  - `for`: temporal for, describes the temporal execution order.
  - `spatial_for`: describes parallel execution.

1D Convolution Example



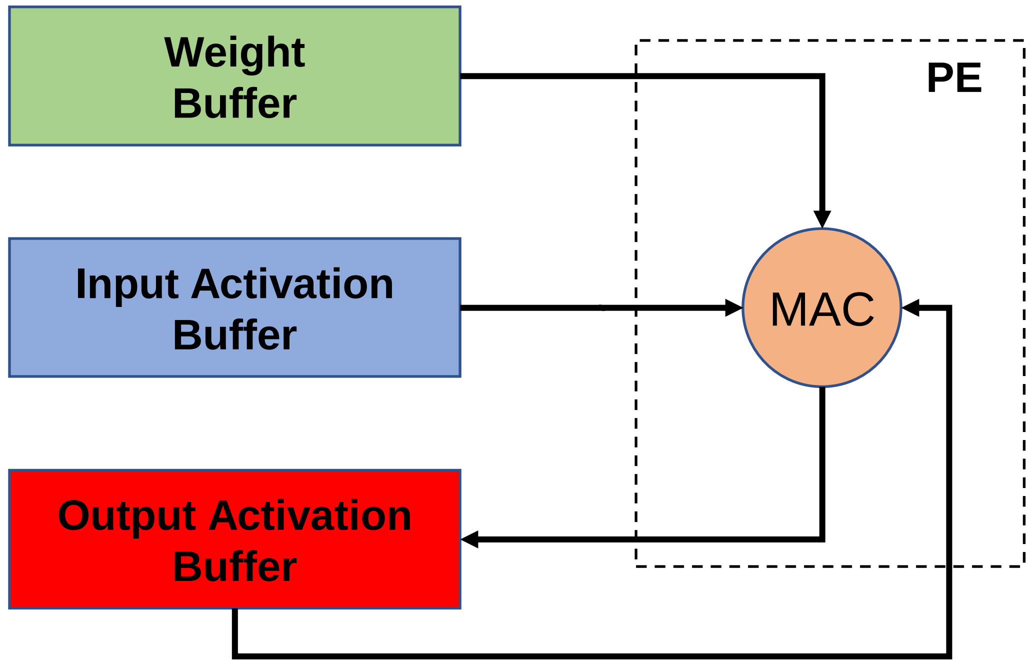
```
for(q=0; q<Q; q++){
  for (s=0; s<S; s++){
    OA[q] += IA[q+s] * W[s];
  }
}
```

Output Stationary (OS) Dataflow

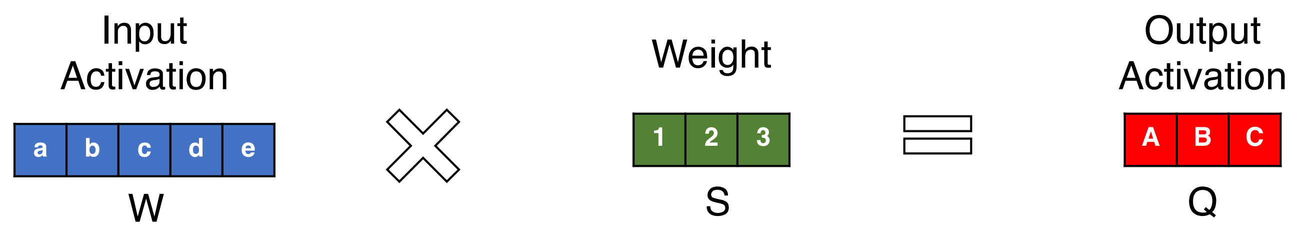
```
for (s=0; s<S; s++){
  for(q=0; q<Q; q++){
    OA[q] += IA[q+s] * W[s];
  }
}
```

Weight Stationary (WS) Dataflow

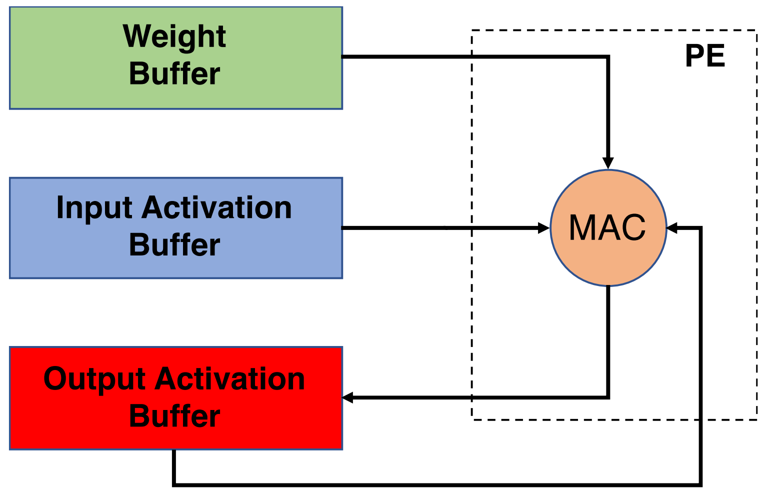
Single Processing Element (PE) Setup



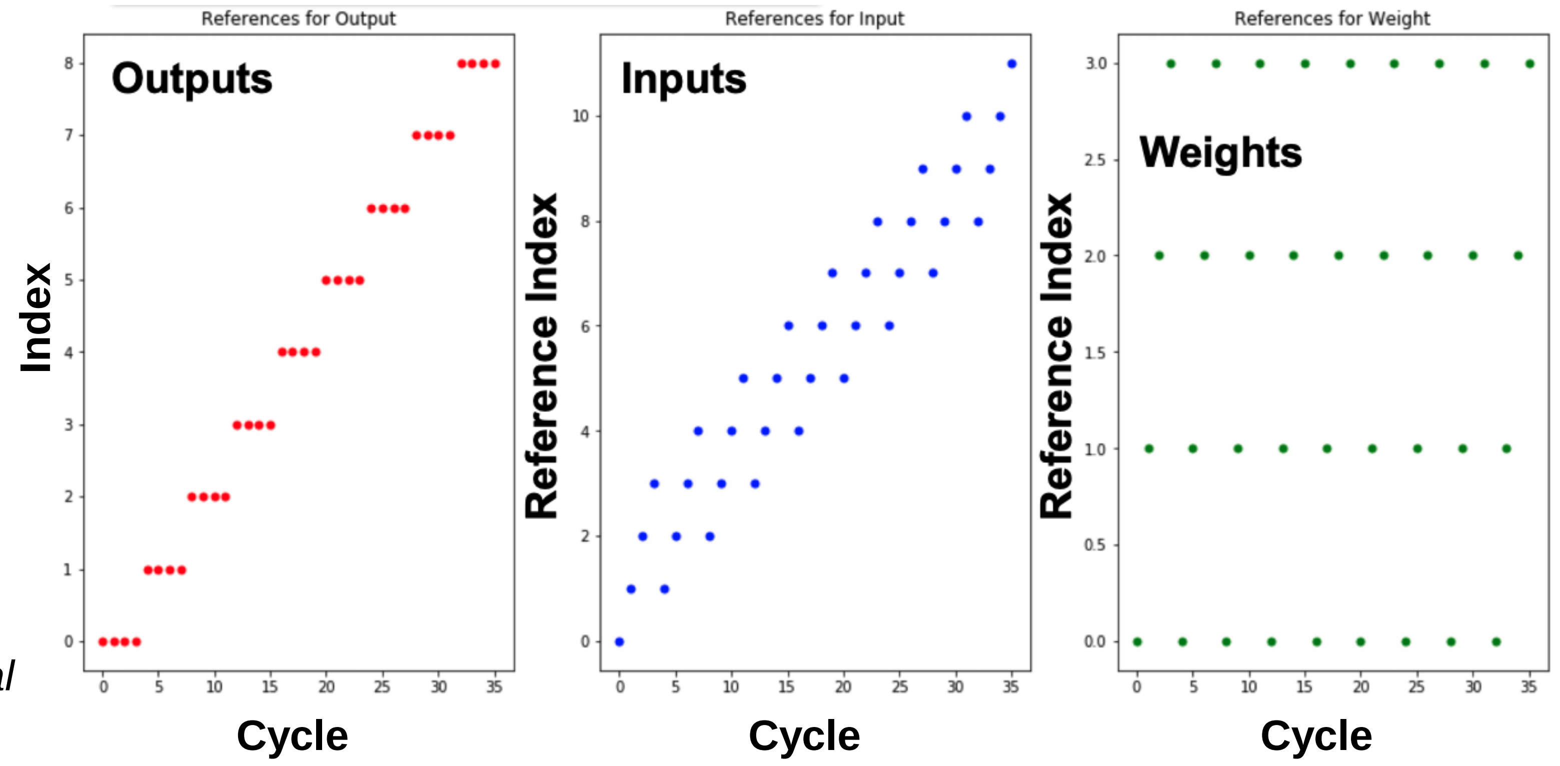
Buffer Access Pattern (OS)



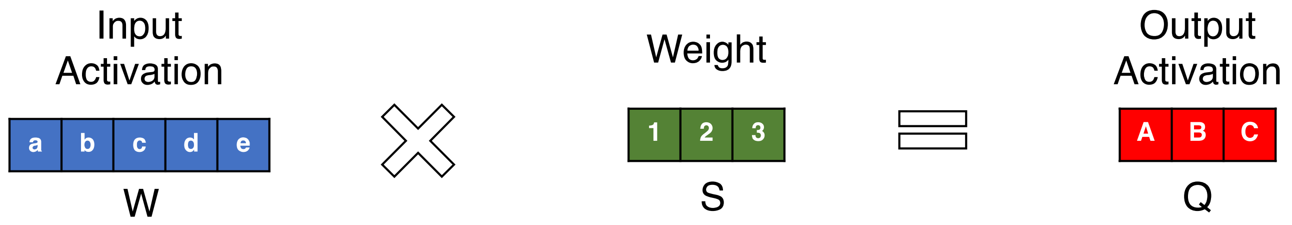
```
for(q=0; q<Q; q++){ // Q =9
  for (s=0; s<S; s++){ // S=4
    OA[q] += IA[q+s] * W[s];
  }
}
```



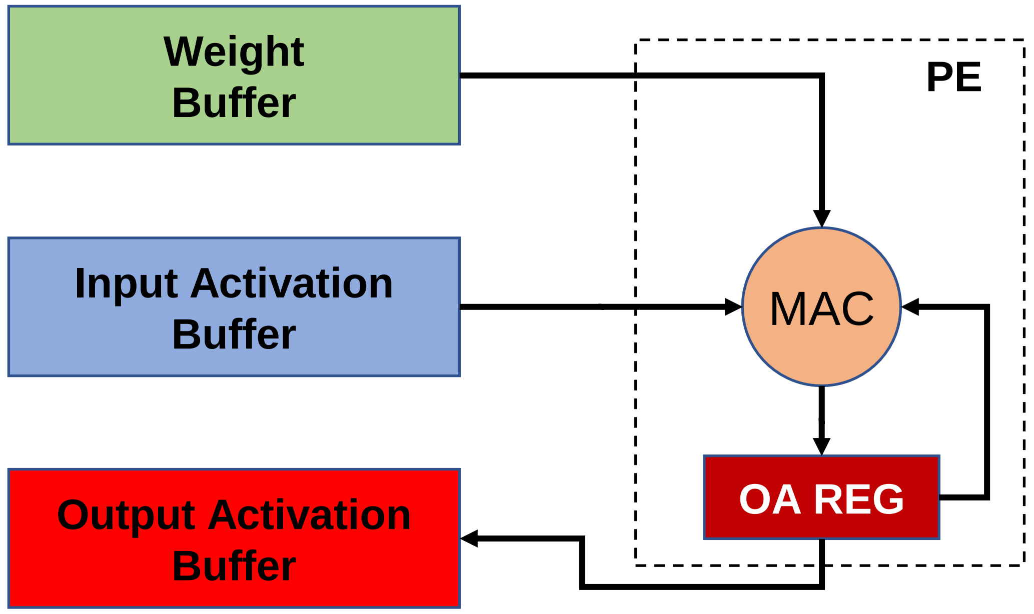
Eyeriss Tutorial

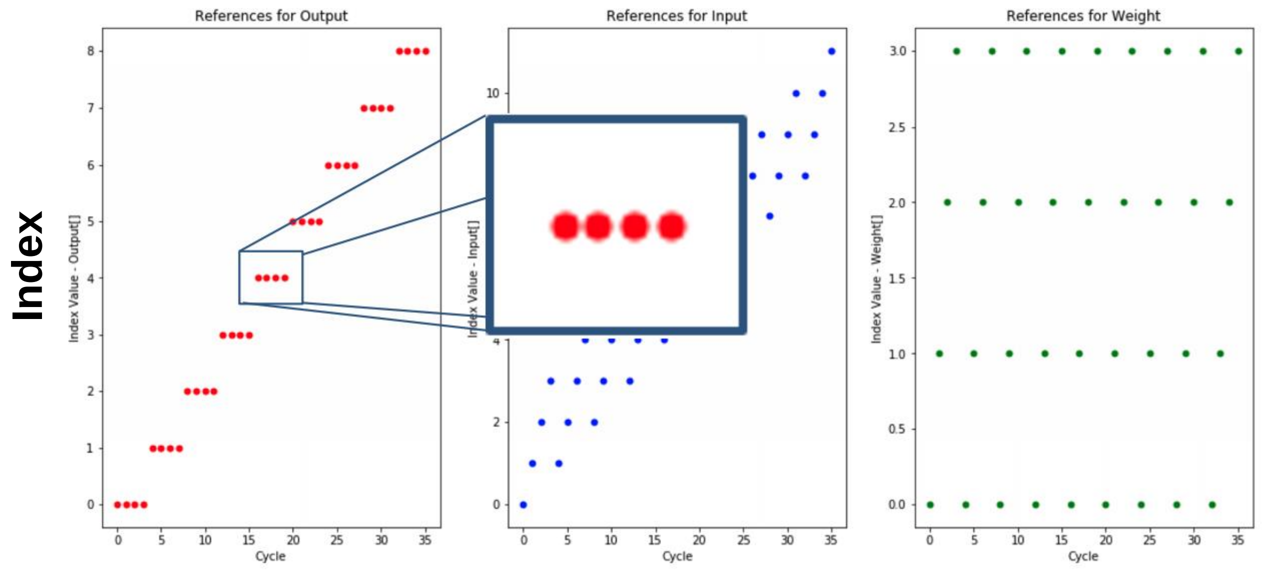


Buffer Access Pattern (OS)



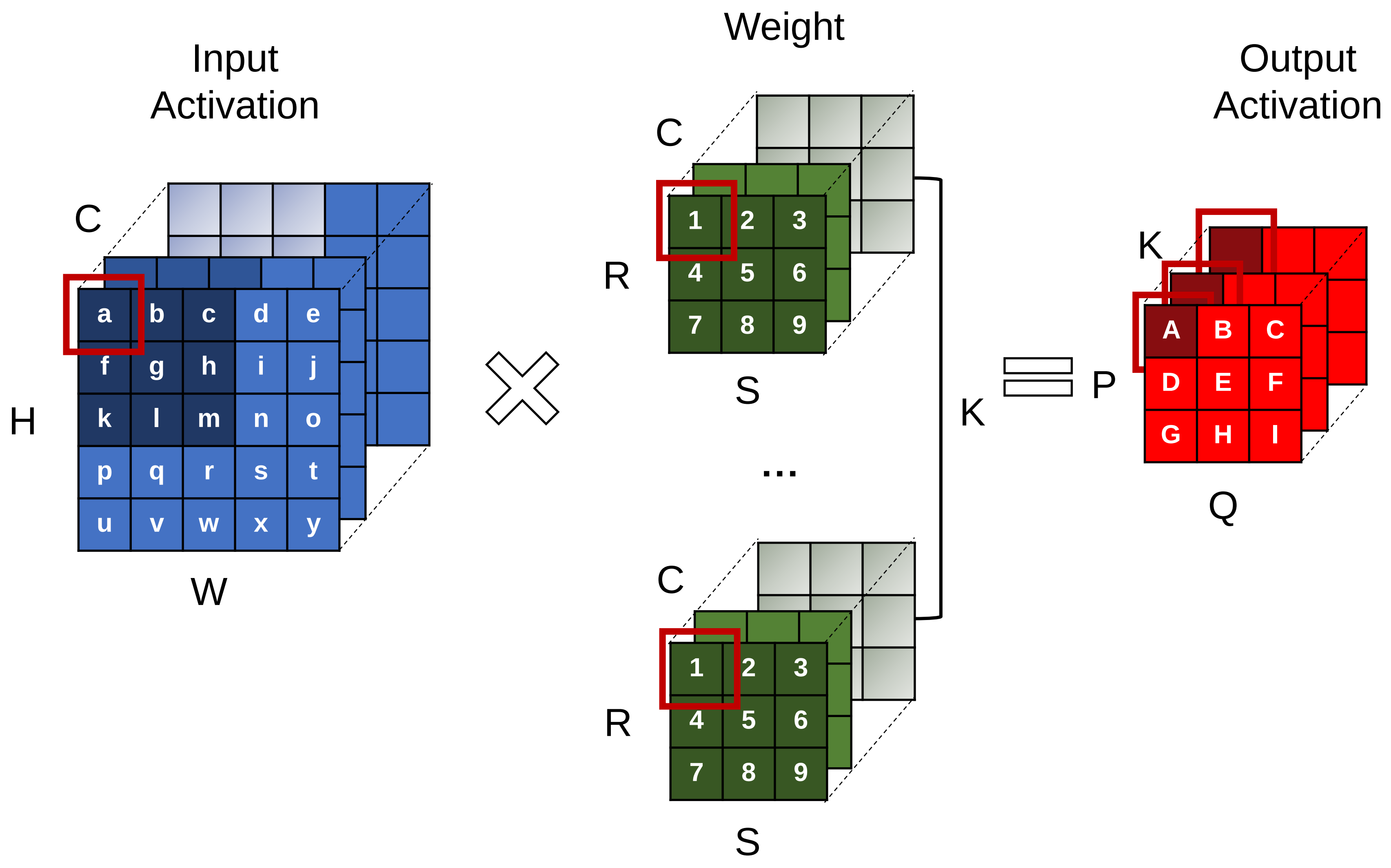
```
for(q=0; q<Q; q++){ // Q =9
  for (s=0; s<S; s++){ // S=4
    OA[q] += IA[q+s] * W[s];
  }
}
```



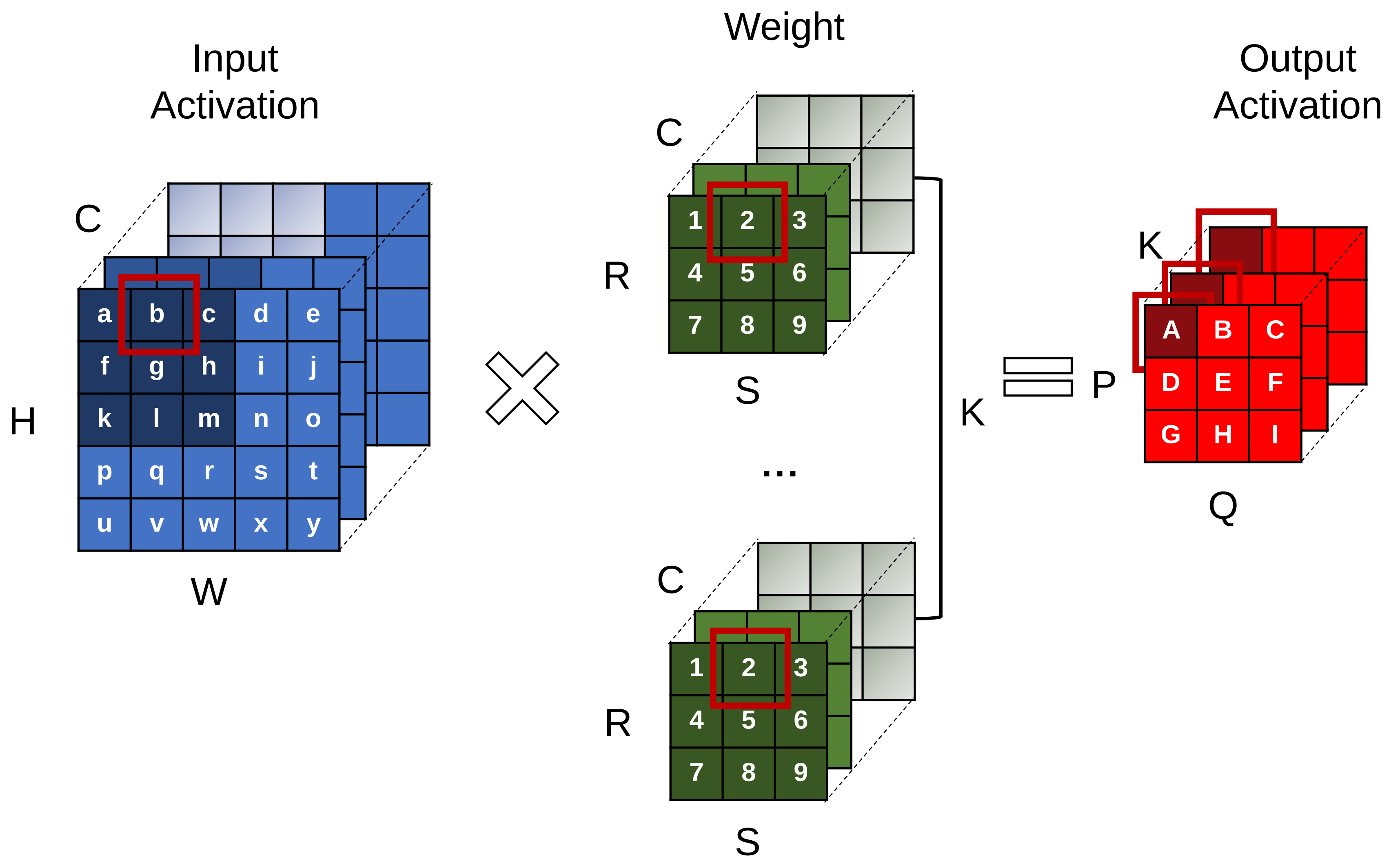


Observation: Single output is reused S times.

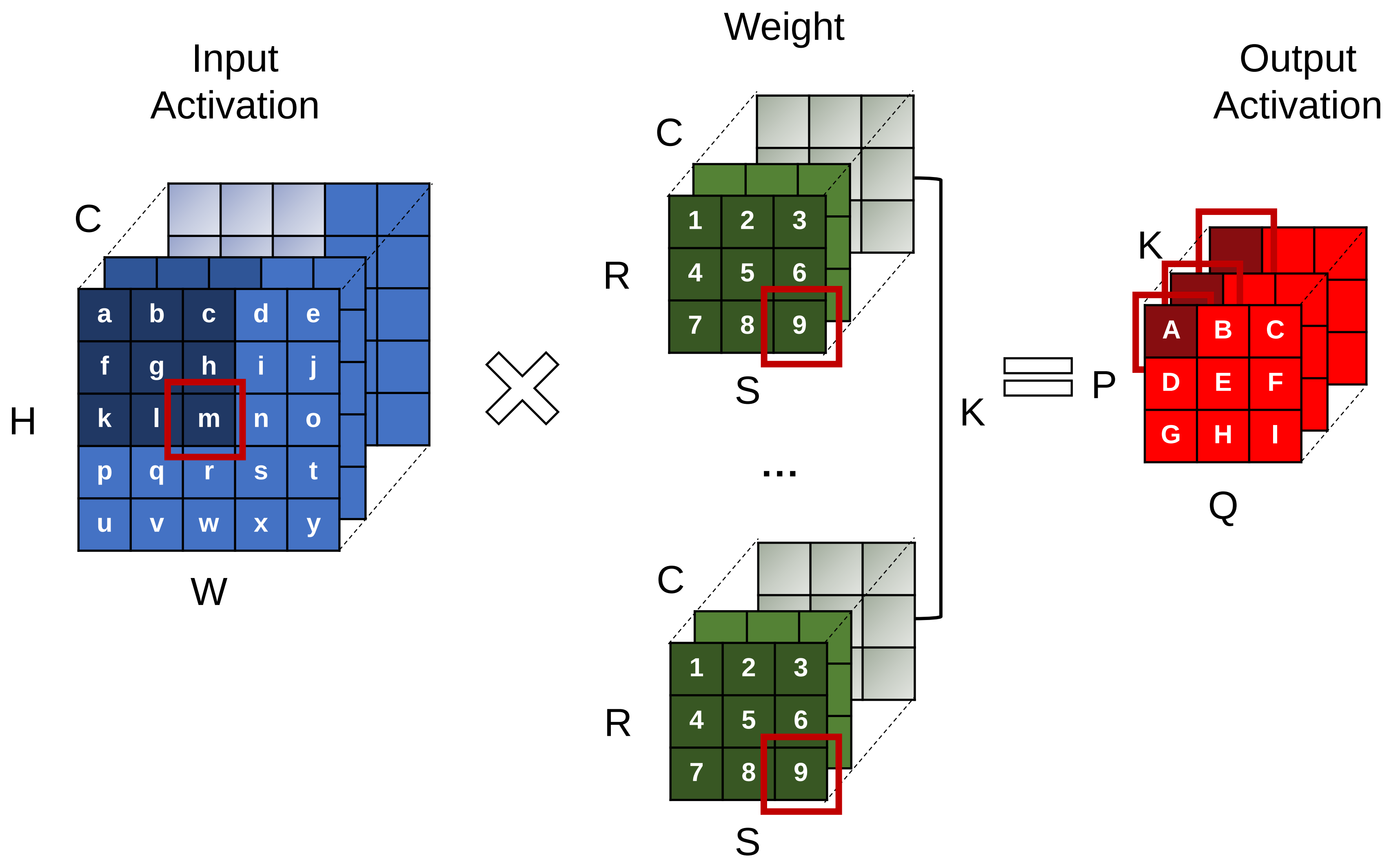
Output Stationary Dataflow



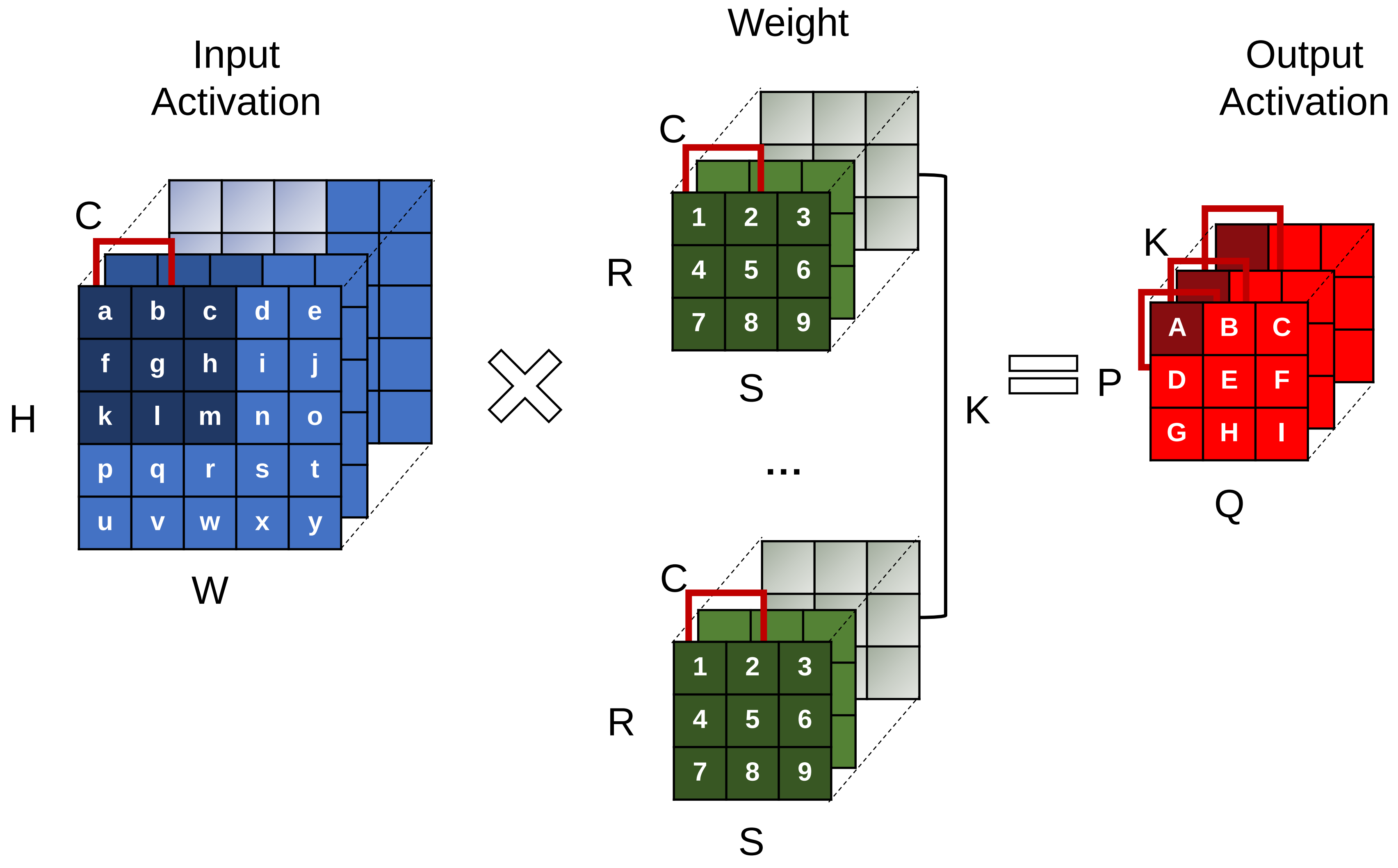
Output Stationary Dataflow



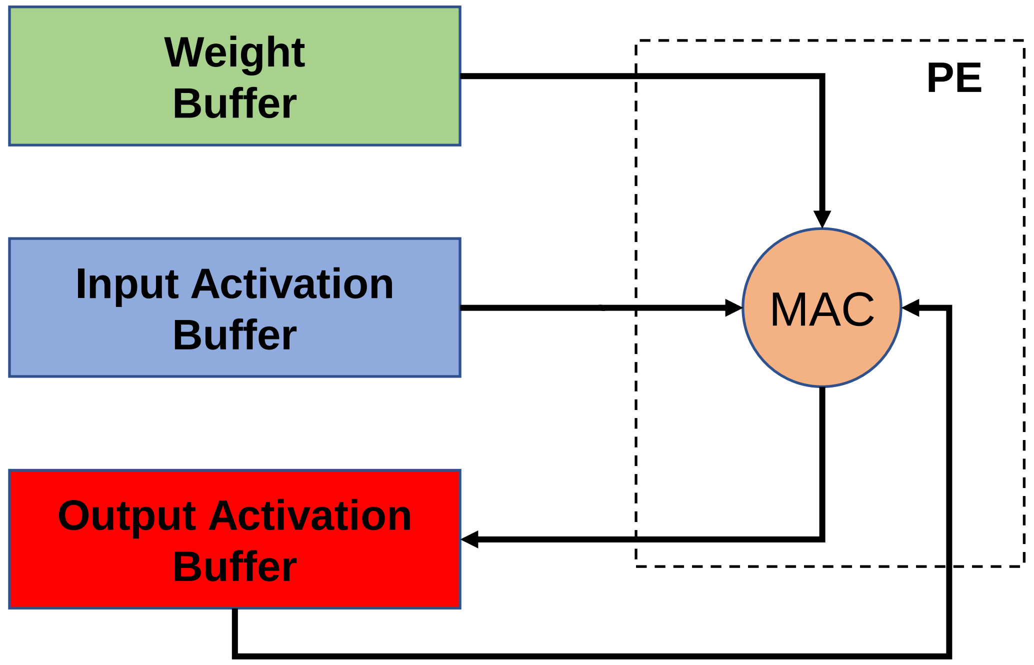
Output Stationary Dataflow



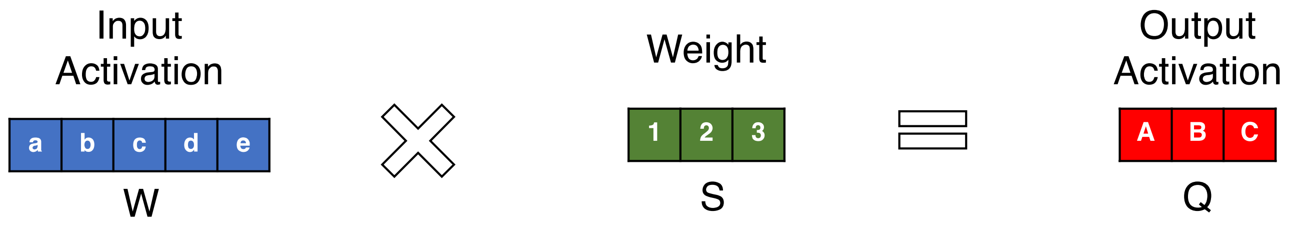
Output Stationary Dataflow



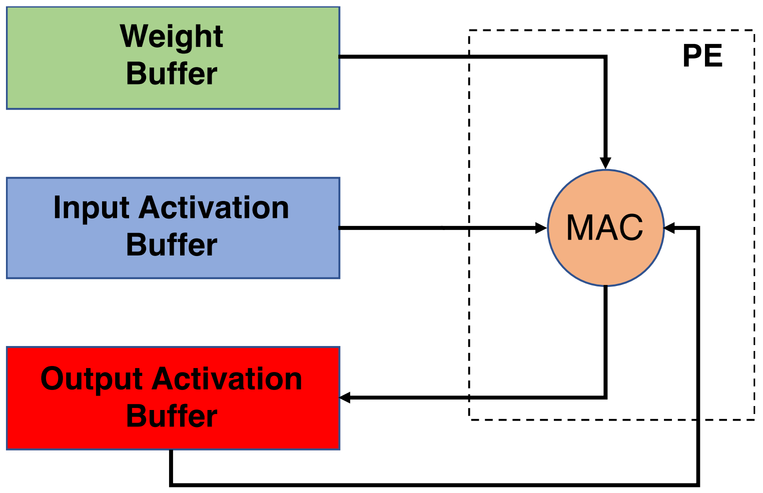
Single Processing Element (PE) Setup

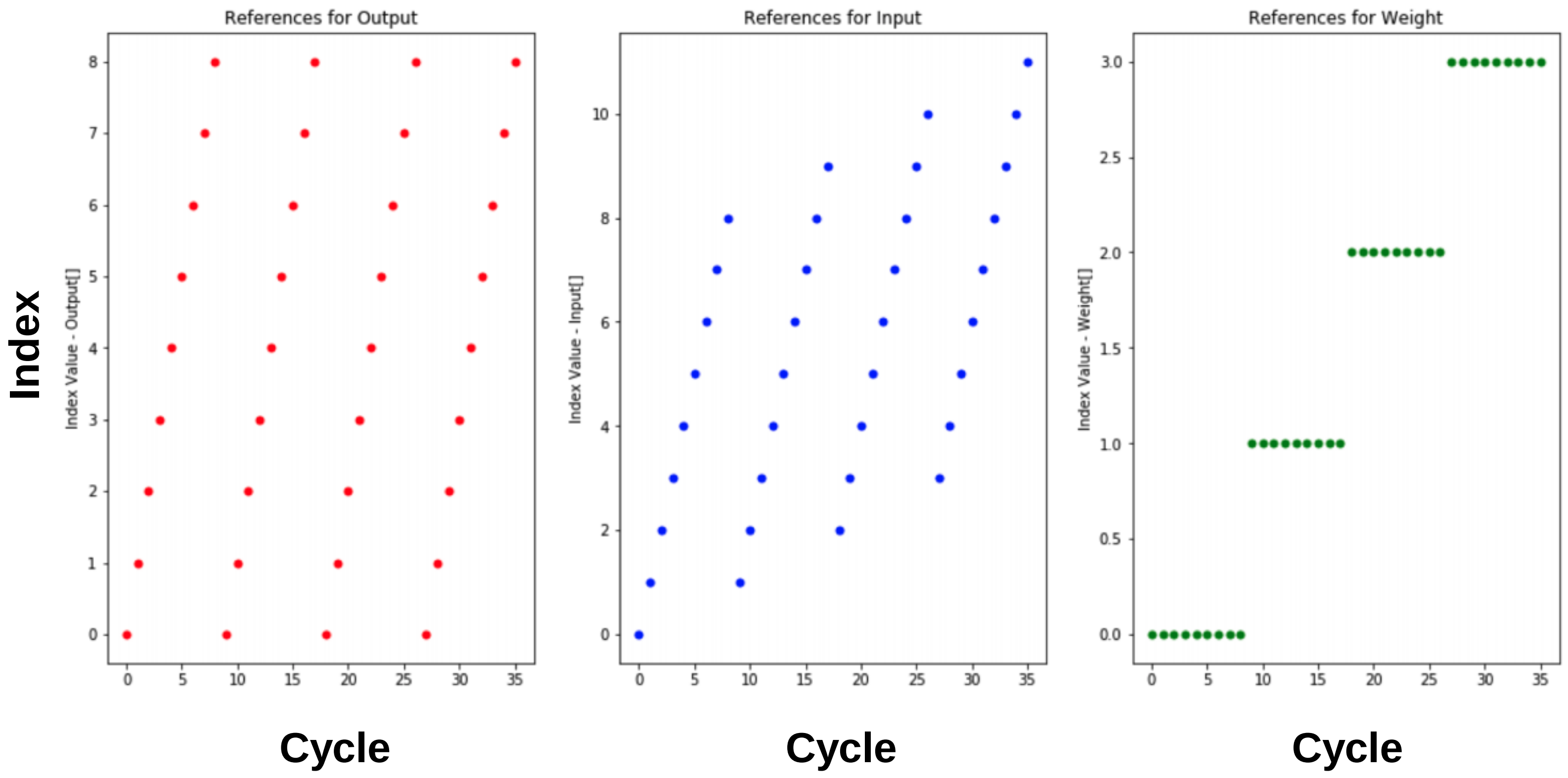


Buffer Access Pattern (WS)



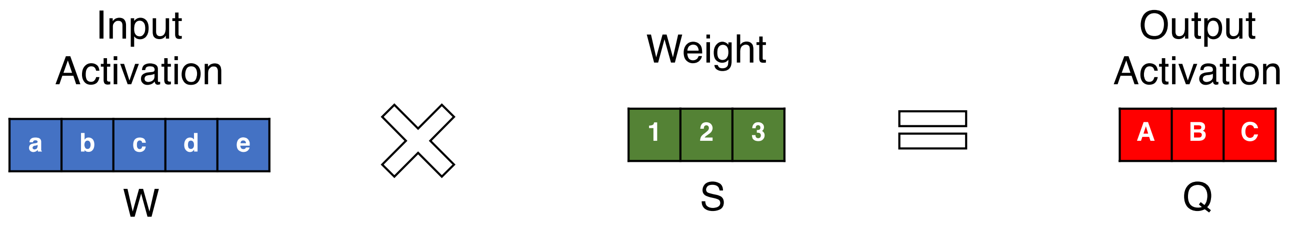
```
for (s=0; s<S; s++){// S=4
  for(q=0; q<Q; q++){// Q =9
    OA[q] += IA[q+s] * W[s];
}
```



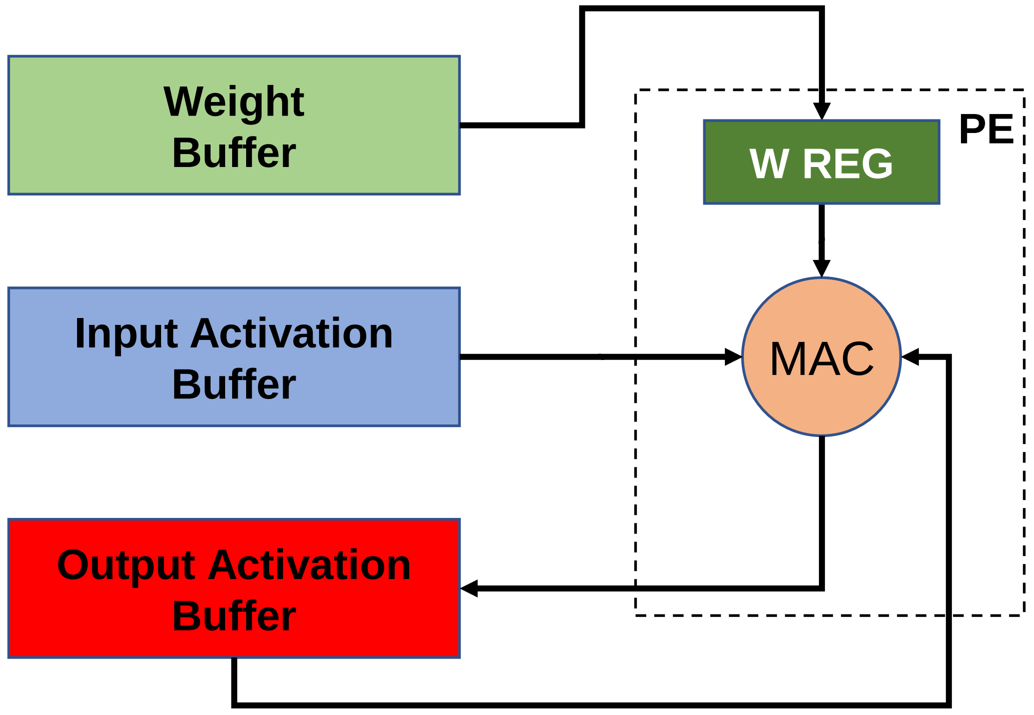


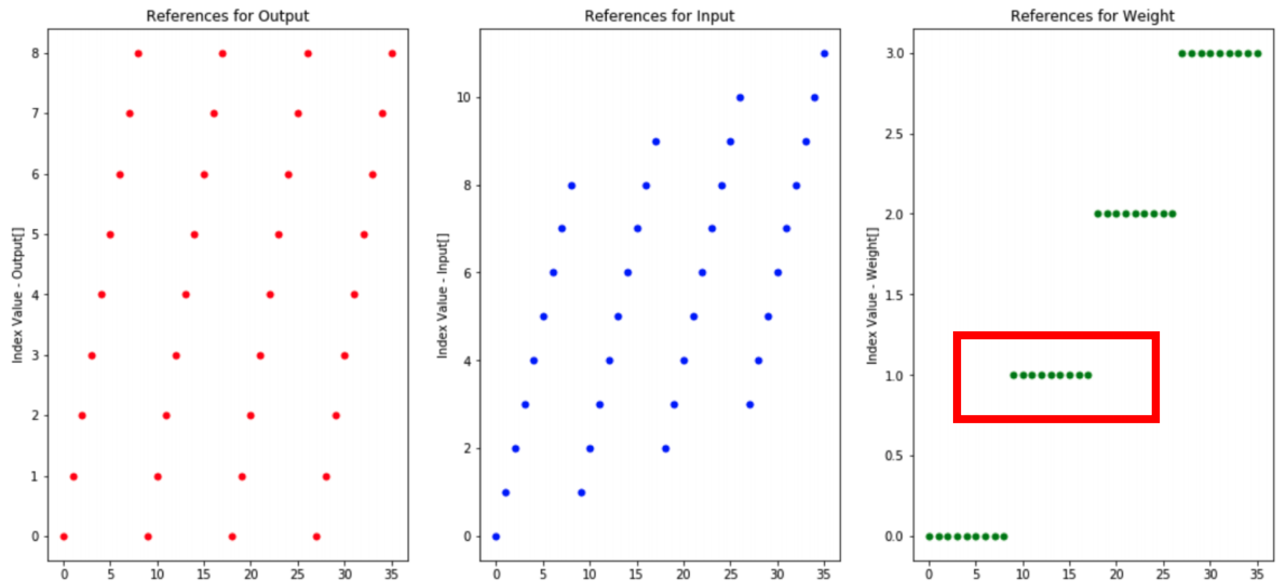
Eyeriss Tutorial

Buffer Access Pattern (WS)



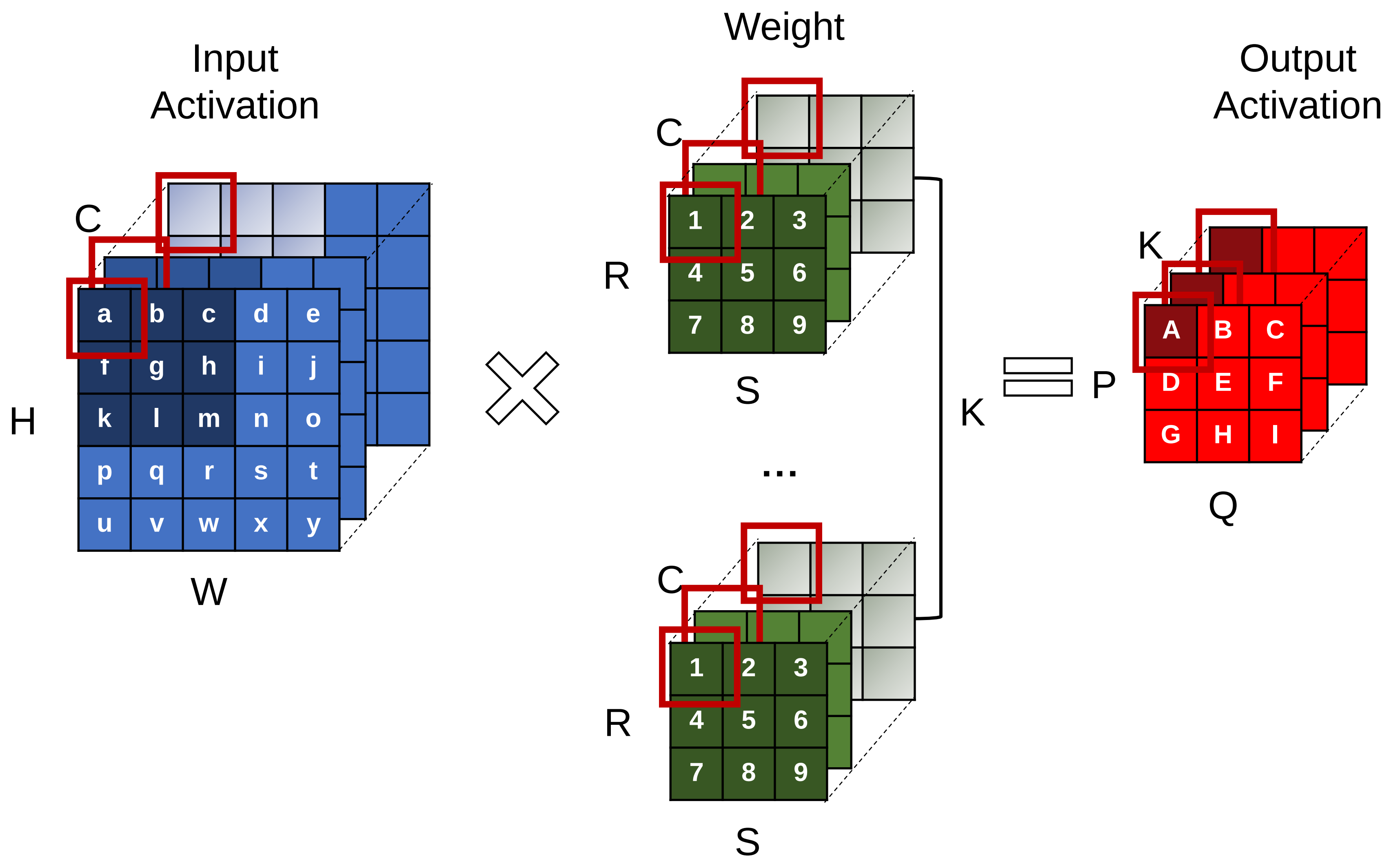
```
for (s=0; s<S; s++){// S=4
   for(q=0; q<Q; q++){// Q =9
      OA[q] += IA[q+s] * W[s];
   }
}
```



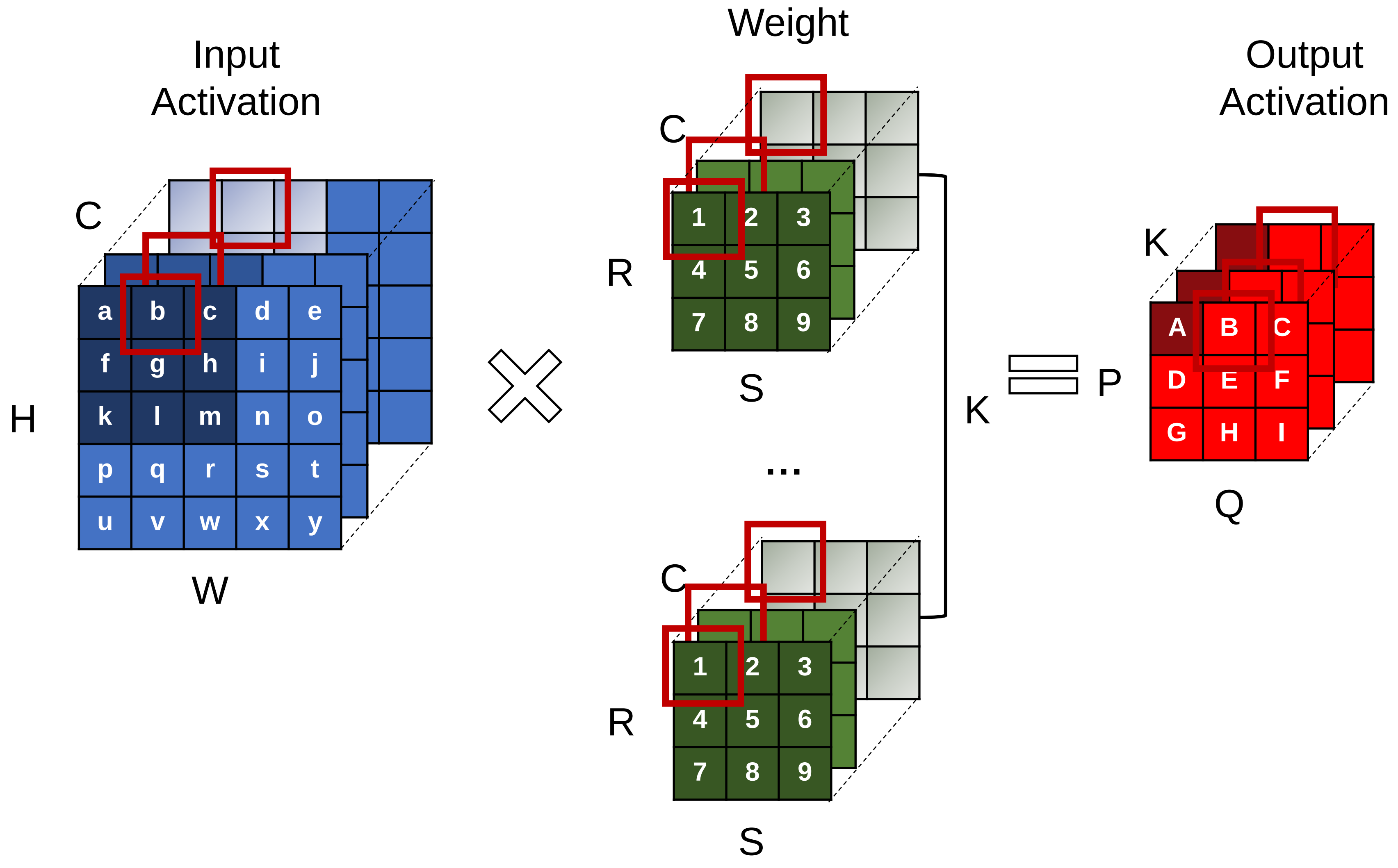


Observation: Single weight is reused Q times.

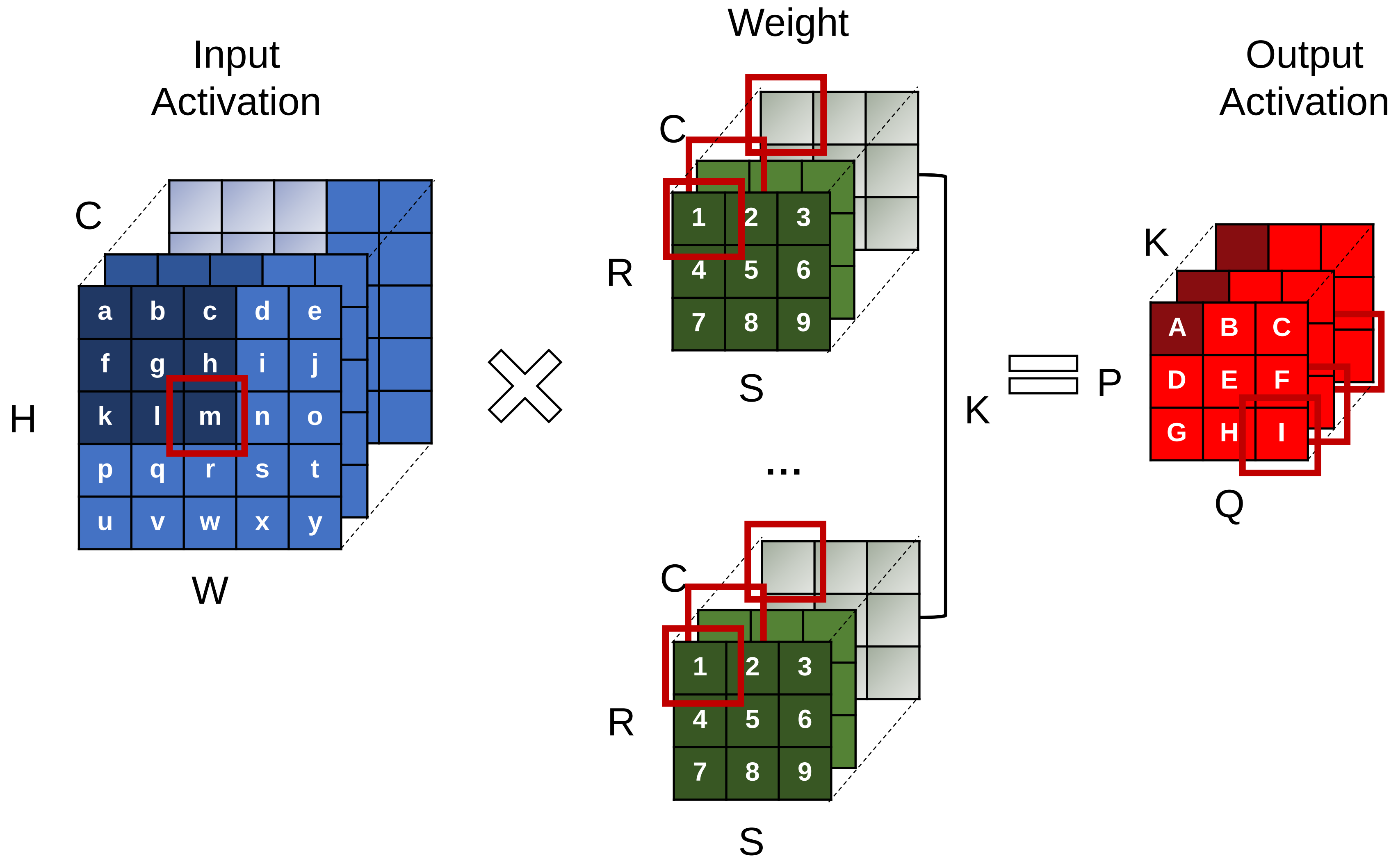
Weight Stationary Dataflow



Weight Stationary Dataflow



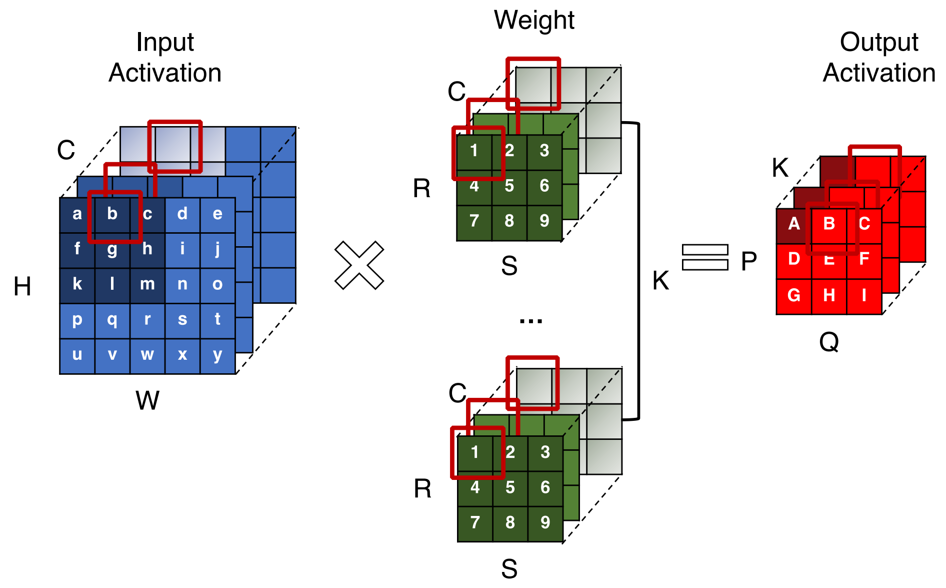
Weight Stationary Dataflow



Review

- Last lecture: core computation in DNN
- This lecture: execution order of the core computation
  - Core principles:
    - Locality and Parallelism
  - Dataflow
    - Defines the execution order of DNNs
    - Represented using a loop nest
      - for: temporal order
      - spatial_for: spatial order/parallelism
    - Output-stationary and weight-stationary dataflow
- Next lecture: hardware realization of the core computation

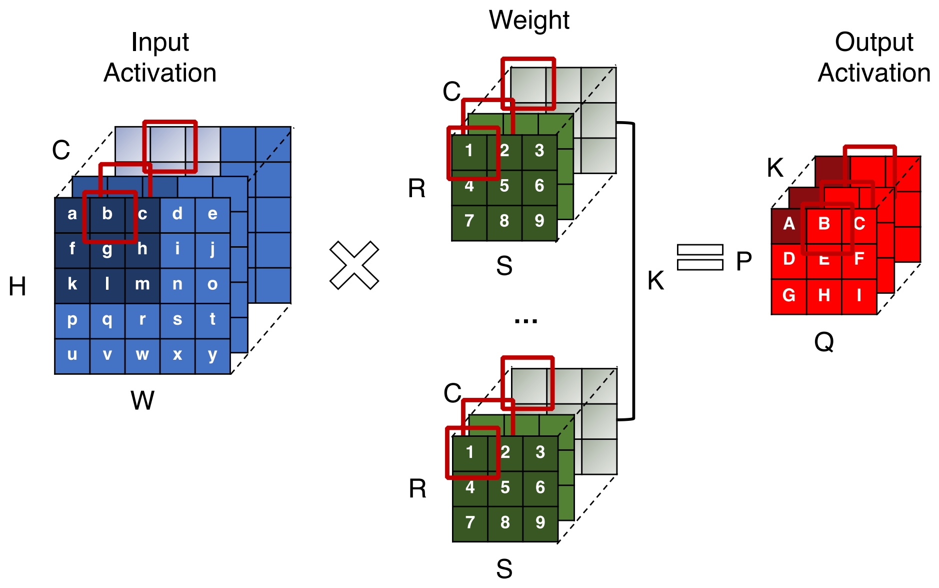
Weight Stationary Dataflow



- What we had before:

```
for (n=0; n<N; n++) {
 for (k=0; k<K; k++) {
  for (p=0; p<P; p++) {
   for (q=0; q<Q; q++) {
    OA[n][k][p][q]= 0;
    for (r=0; r<R; r++) {
     for (s=0; s<S; s++) {
      for (c=0; c<C; c++) {
       h = p * stride - pad + r;
       w = q * stride - pad + s;
       OA[n][k][p][q] +=
            IA[n][c][h][w]
             * W[k][c][r][s];
      }
     }
    }
}
  }
 }
}
```

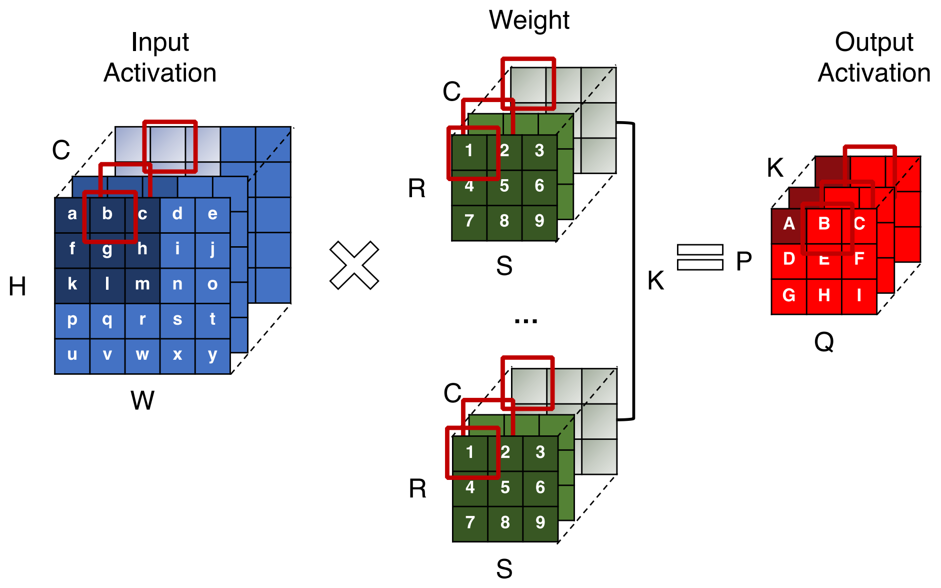
Weight Stationary Dataflow



- Change temporal ordering

```
for (n=0; n<N; n++) {
  for (r=0; r<R; r++) {
    for (s=0; s<S; s++) {
      for (c=0; c<C; c++) {
        for (k=0; k<K; k++) {
          float curr_w = W[r][s][c][k];
          for (p=0; p<P; p++) {
            for (q=0; q<Q; q++) {
              h = p * stride - pad + r;
              w = q * stride - pad + s;
              OA[n][k][p][q] +=
                        IA[n][c][h][w]
                          * curr_w;
            }
          }
        }
      }
    }
  }
}
```

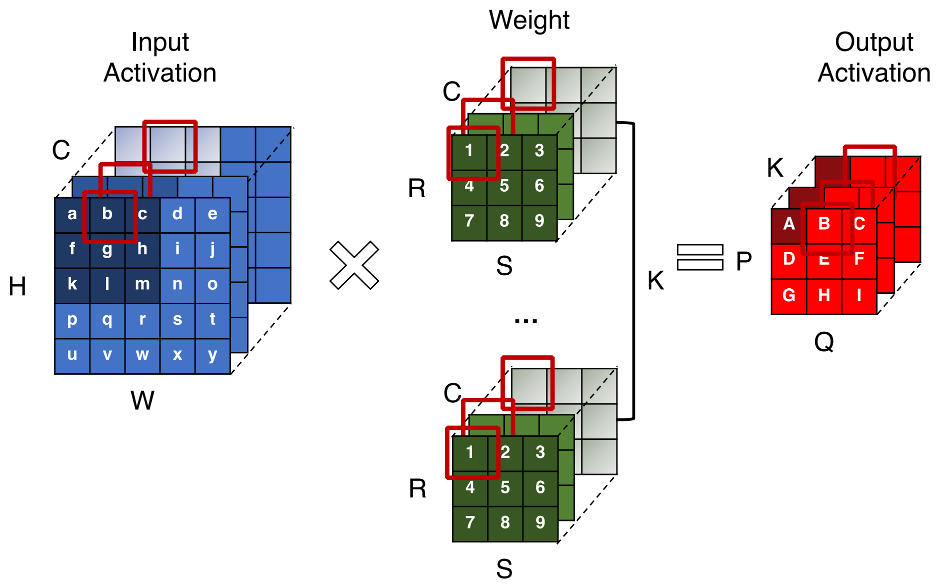
Weight Stationary Dataflow



- Apply spatial parallelism

```
for (n=0; n<N; n++) {
  for (r=0; r<R; r++) {
    for (s=0; s<S; s++) {
      spatial_for (c=0; c<C; c++) {
        spatial_for (k=0; k<K; k++) {
          float curr_w = W[r][s][c][k];
          for (p=0; p<P; p++) {
            for (q=0; q<Q; q++) {
              h = p * stride - pad + r;
              w = q * stride - pad + s;
              OA[n][k][p][q] +=
                   IA[n][c][h][w]
                     * curr_w;
            }
          }
        }
      }
    }
  }
}
```

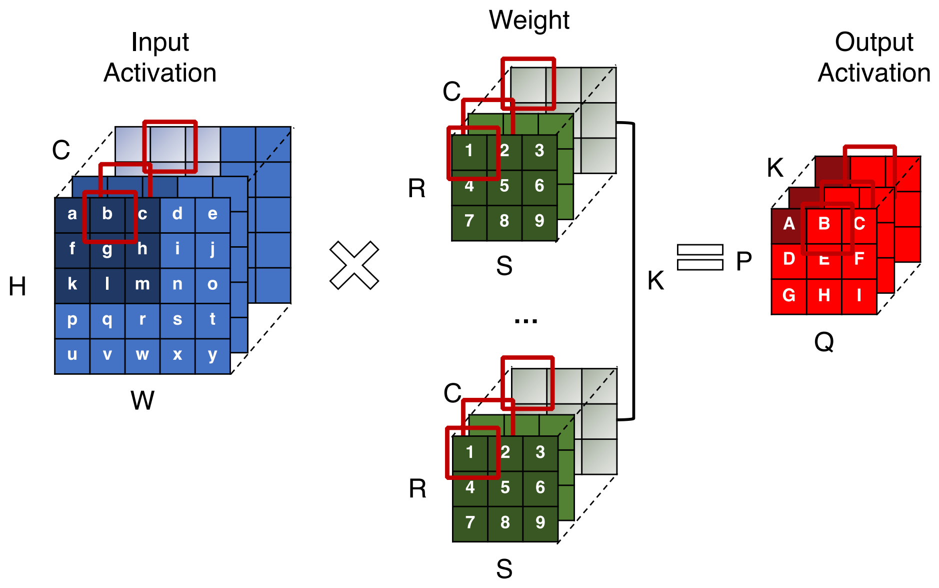
Weight Stationary Dataflow



- Apply temporal tiling

```
for (n=0; n<N; n++) {
  for (r=0; r<R; r++) {
    for (s=0; s<S; s++) {
      for (c_t=0; c_t<C/16; c_t++) {
        for (k_t=0; k_t<K/64; k_t++) {
          spatial_for (c_s=0; c_s<16; c_s++) {
            spatial_for (k_s=0; k_s<64; k_s++) {
              int curr_c = c_t * 16 + c_s;
              int curr_k = k_t * 64 + k_s;
              float curr_w = W[r][s][curr_c][curr_k];
              for (p=0; p<P; p++) {
                for (q=0; q<Q; q++) {
                  h = p * stride - pad + r;
                  w = q * stride - pad + s;
                  OA[n][curr_k][p][q] +=
                          IA[n][curr_c][h][w]
                          * curr_w;
      }}}}
    }
  }
}
```

Weight Stationary Dataflow



- **NVDLA** Dataflow (nvdla.org)

```
for (n=0; n<N; n++) {
  for (r=0; r<R; r++) {
    for (s=0; s<S; s++) {
      for (c_t=0; c_t<C/16; c_t++) {
        for (k_t=0; k_t<K/64; k_t++) {
          spatial_for (c_s=0; c_s<16; c_s++) {
            spatial_for (k_s=0; k_s<64; k_s++) {
              int curr_c = c_t * 16 + c_s;
              int curr_k = k_t * 64 + k_s;
              float curr_w = W[r][s][curr_c][curr_k];
              for (p=0; p<P; p++) {
                for (q=0; q<Q; q++) {
                  h = p * stride - pad + r;
                  w = q * stride - pad + s;
                  OA[n][curr_k][p][q] +=
                              IA[n][curr_c][h][w]
                              * curr_w;
      }}}}
    }
  }
}
```