CASE SCHOOL
OF ENGINEERING
CASE WESTERN RESERVE
UNIVERSITY

CSDS 451: Designing High Performant Systems for AI

Lecture 10

9/25/2025

Sanmukh Kuppannagari

sanmukh.kuppannagari@case.edu

https://sanmukh.research.st/

Case Western Reserve University

Outline

- Convolution as Matrix Multiplication – Motivation
- Im2Col and Kn2Row Algorithms
- Scalar Matrix Multiplication Algorithm

Reading Materials

- Papers on Im2Col, Implicit Im2Col
  - **Original Im2Col:** Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., & Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*.
  - **Implicit Im2Col Nvidia Doc:** https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html
  - **Implicit Im2Col:** S. Lym, D. Lee, M. O'Connor, N. Chatterjee, and M. Erez, "Delta: Gpu performance model for deep learning applications with in-depth memory system traffic analysis," in 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2019, pp. 293–303
  - **Implicit Im2Col:** Zhou, Yangjie, et al. "Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators." *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021.

Reading Materials

- **Kn2Row:** Vasudevan, A., Anderson, A., & Gregg, D. (2017, July). Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)* (pp. 19-24). IEEE.
- **Scalar Matrix Multiplication:** Ofir, A., & Ben-Artzi, G. (2022). Smm-conv: Scalar matrix multiplication with zero packing for accelerated convolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 3067-3075).

Outline

- Convolution as Matrix Multiplication – Motivation
- Im2Col and Kn2Row Algorithms
- Scalar Matrix Multiplication Algorithm

Key Operation in CNN

- Forward Propagation:

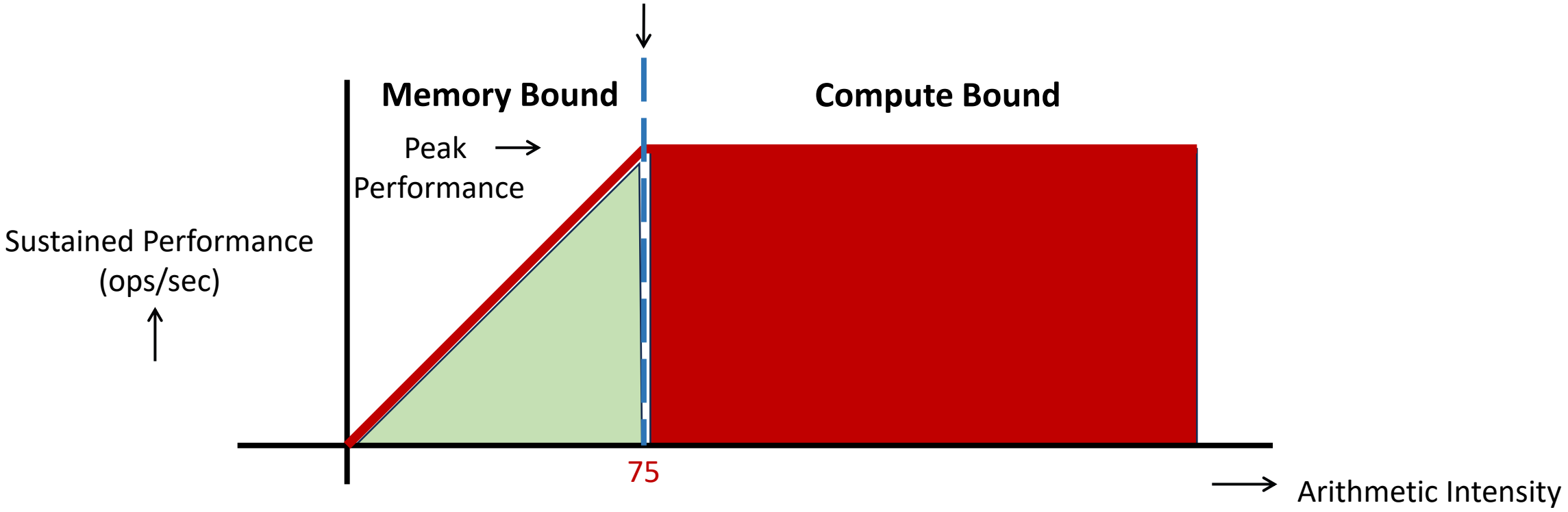
$$O_l = Conv(W_l, I_l)$$

- Backward Propagation

$$\frac{\delta E(W)}{\delta W_l} = Conv(I_l, \frac{\delta E(W)}{\delta O_l})$$

$$\frac{\delta E(W)}{\delta O_l} = Conv(W_{180}, \frac{\delta E(W)}{\delta O_{l+1}})$$

CNN on A100 Roofline Plot (assuming infinite cache)



Which area will the CNN Performance lie in? Compute Bound

Implications

- Essentially a Compute Bound Problem, assuming large cache
- Objective: How to ensure efficiency is $O(1)$, i.e., all compute units are fully utilized
- If cache is limited
  - Objective 1: How to improve data reuse so that compute units have enough to work on?
  - Objective 2: How to ensure efficiency is $O(1)$, i.e., all compute units are fully utilized
- Another approach: increase Number of Devices
  - + Larger On-chip memory
  - + Higher bandwidth
  - - Communication/Synchronization overheads

Convolution as Matrix Multiplications

- Transform the input features and kernels into matrices and perform matrix multiplication.
- Theoretically, matrix multiplication algorithms achieve cost optimality and good data reuse
- Practically, every hardware vendor provides optimized implementations of matrix operations

Block Matrix Multiplication

- Data Reuse Factor:

- Total Computations - $O(\sqrt{S}^3 \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}})$
- Total data transfers – $O(\sqrt{S} \times \sqrt{S} \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}} \times \frac{N}{\sqrt{S}})$ transfers

- Data Reuse Factor: $O(\sqrt{S})$

Blocked Matrix Multiplication

- Serial Algorithm - $O(n^3)$

- Parallel Algorithm:
- Number of steps n/b
- Work done by each processor in each step: $O(b)$

- Cost of the algorithm: $O\left(n^2 \times \left[\frac{n}{b} \times b\right]\right) = O(n^2 \times n) = O(n^3)$

- Cost of the algorithm ~ serial complexity. So work optimal.

BLAS – Basic Linear Algebra Subprograms

- A specification describing a set of low level routines for performing common linear algebra operations
- Hardware vendors develop optimized versions for their platform,
- While still conforming to the specification enabling portability

BLAS – Basic Linear Algebra Subprograms

- Three levels of operations:
- Level 1: Vector Operations such as dot products, vector addition, scaling, etc.
- Level 2: Matrix Vector Operations such as matrix-vector multiplication or
- Level 3: Matrix-Matrix Operations such as matrix multiplications, scaling, etc.

GEMM – Generalized Matrix Matrix Multiplication

- Matrix Operation of the form

$$C \leftarrow \alpha op(A).op(B) + \beta C$$

- α, β: scalar values
- A, B, C: matrices

- op: transpose operation that is applied if an appropriate parameter is set

GEMM – Generalized Matrix Matrix Multiplication

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float           *C, int ldc)
```

GEMM – Generalized Matrix Matrix Multiplication

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float           *C, int ldc)
```

Parameters that determine whether to transpose the matrix or not:

transa = CUBLAS_OP_N -> No transposition
transa = CUBLAS_OP_T -> Transpose
transa = CUBLAS_OP_H -> Hermitian Transpose

GEMM – Generalized Matrix Matrix Multiplication

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float           *C, int ldc)
```

Dimensions of the matrices

GEMM – Generalized Matrix Matrix Multiplication

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float        *alpha,
                           const float        *A, int lda,
                           const float        *B, int ldb,
                           const float        *beta,
                           float              *C, int ldc)
```

Scalar Values α, β in the equation

GEMM – Generalized Matrix Matrix Multiplication

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float           *C, int ldc)
```

Pointers to the matrices (stored as a 1D array), leading dimension of the matrix

float A[12] = {1,2,3, 4,5,6, 7,8,9, 10,11,12};

Lda = 3; -> // 3x4 matrix

GEMM – Generalized Matrix Matrix Multiplication

- Several variants exist – different data types, batched matrix multiplications, etc.

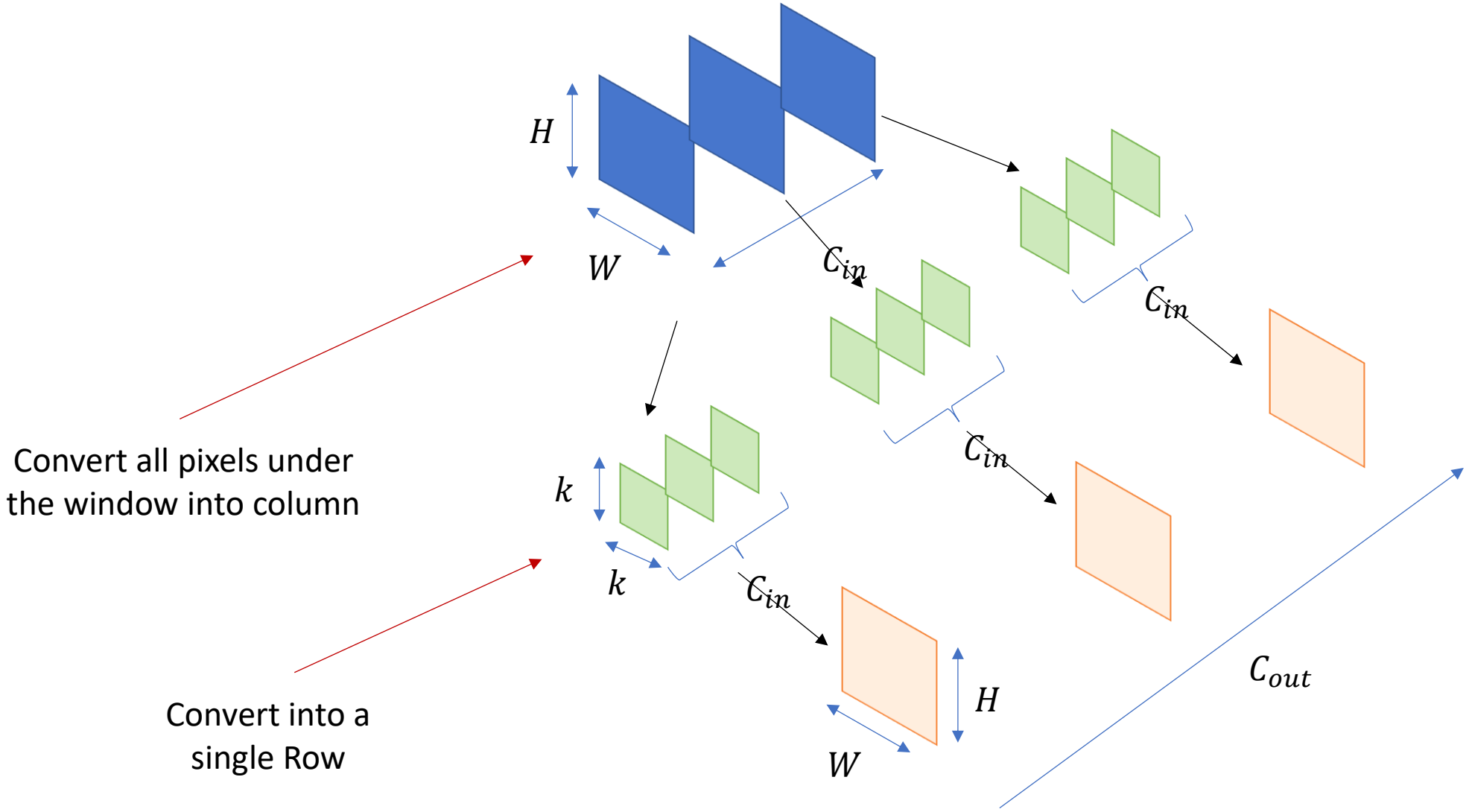
- Ungraded HW Assignment: Try out an example on HPC on CPU or GPU
  - Find out what library to use for the CPU and the GPU
  - Find out how to properly setup the matrices and make the call

- We will discuss the AMD library for BLAS when we study AMD HIP

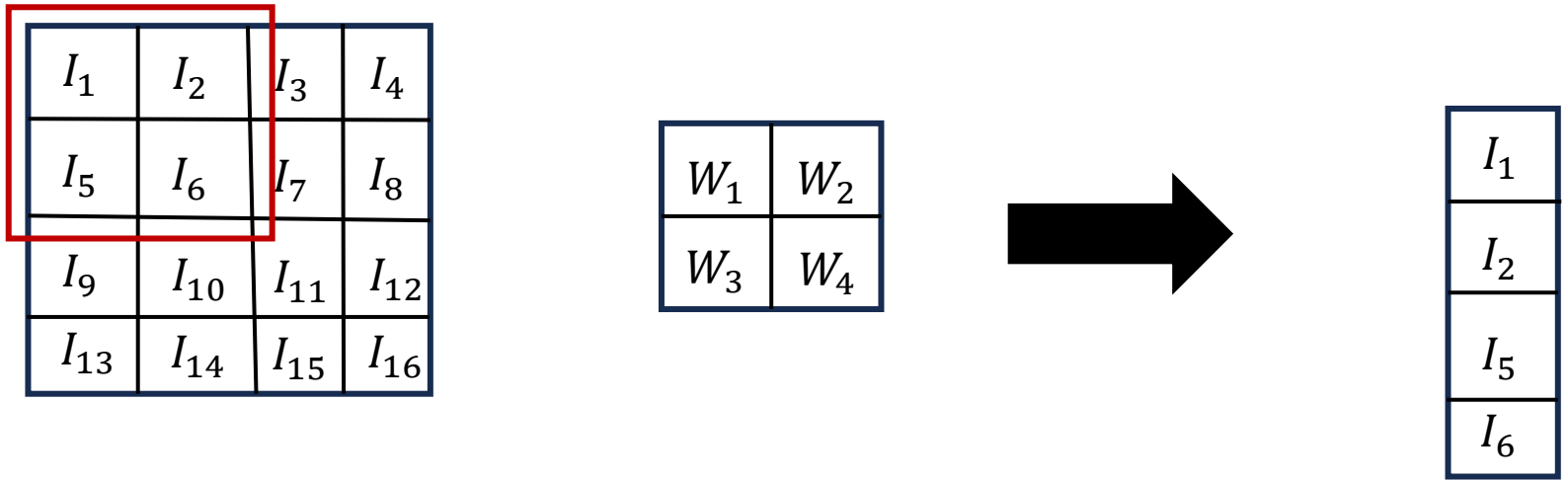
Outline

- Convolution as Matrix Multiplication – Motivation
- Im2Col and Kn2Row Algorithms
- Scalar Matrix Multiplication Algorithm

Im2col Algorithm

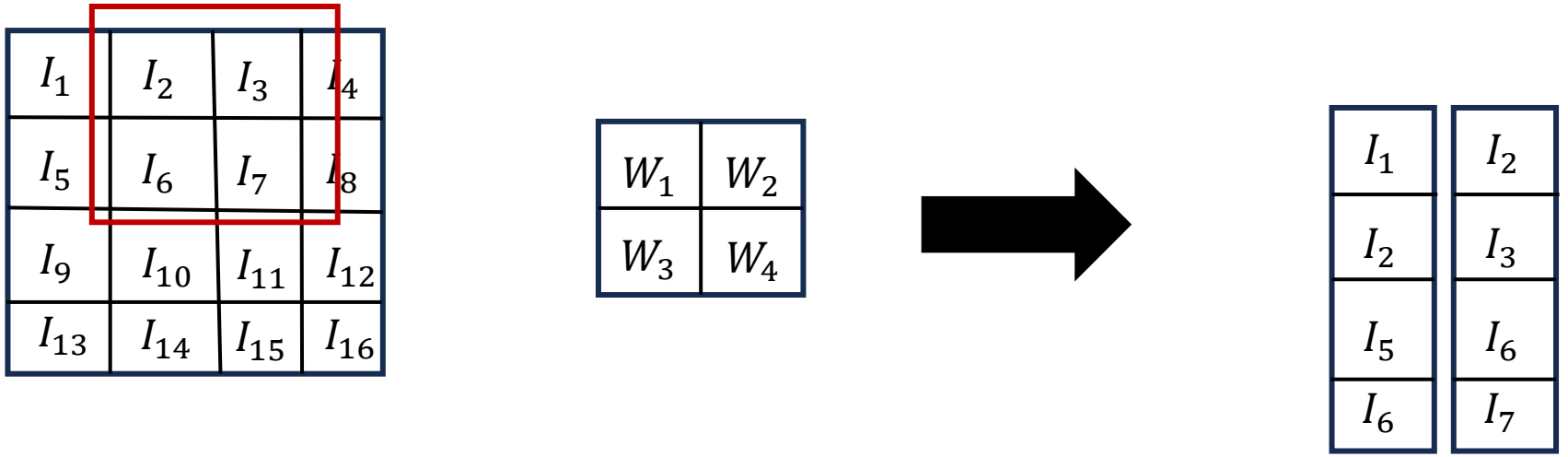


Im2col Algorithm



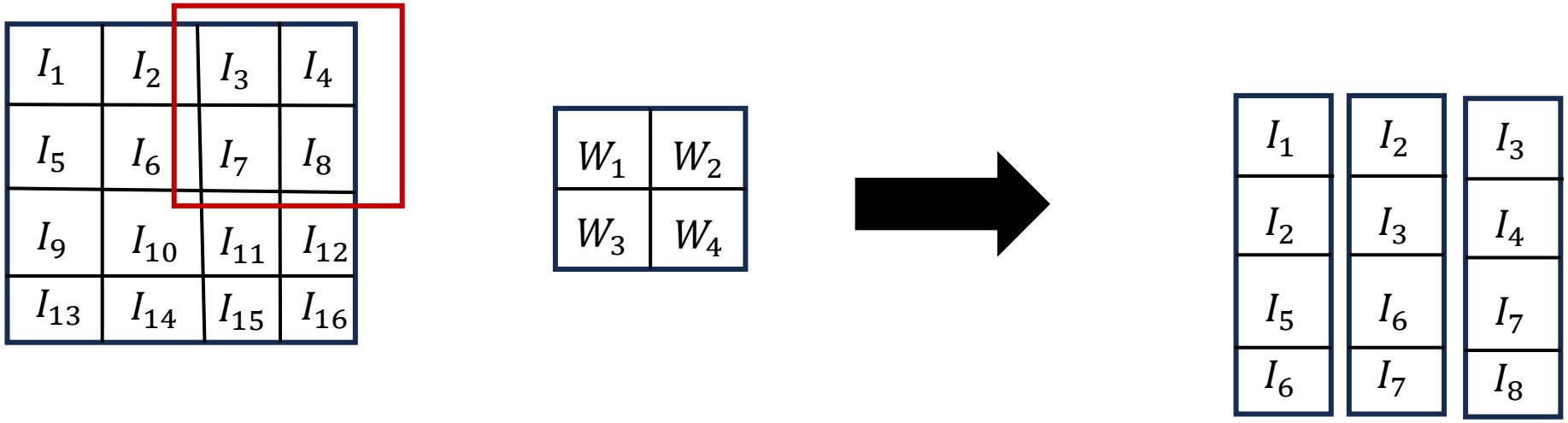
Single Input Single Output Channel

Im2col Algorithm



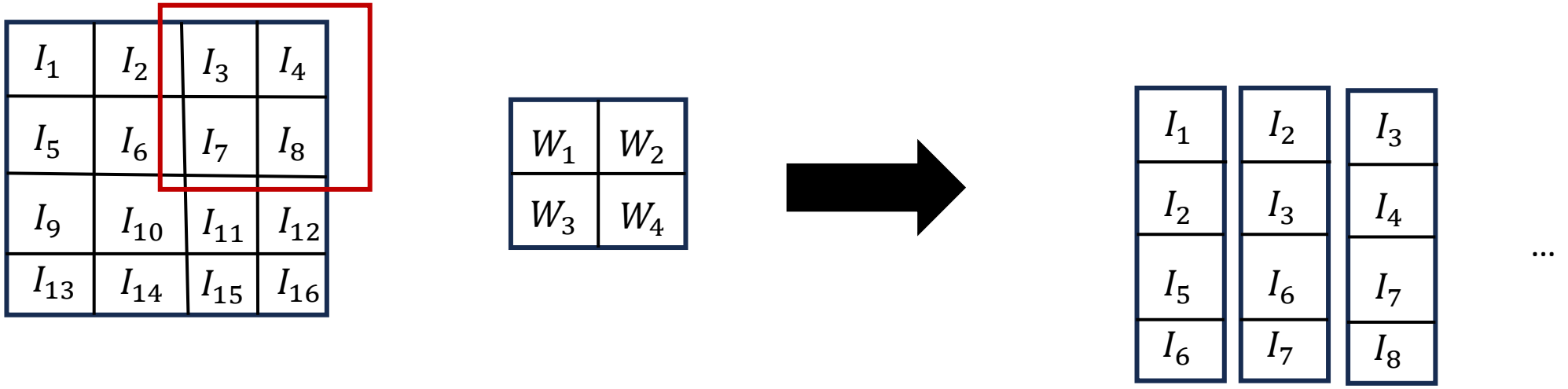
Single Input Single Output Channel

Im2col Algorithm



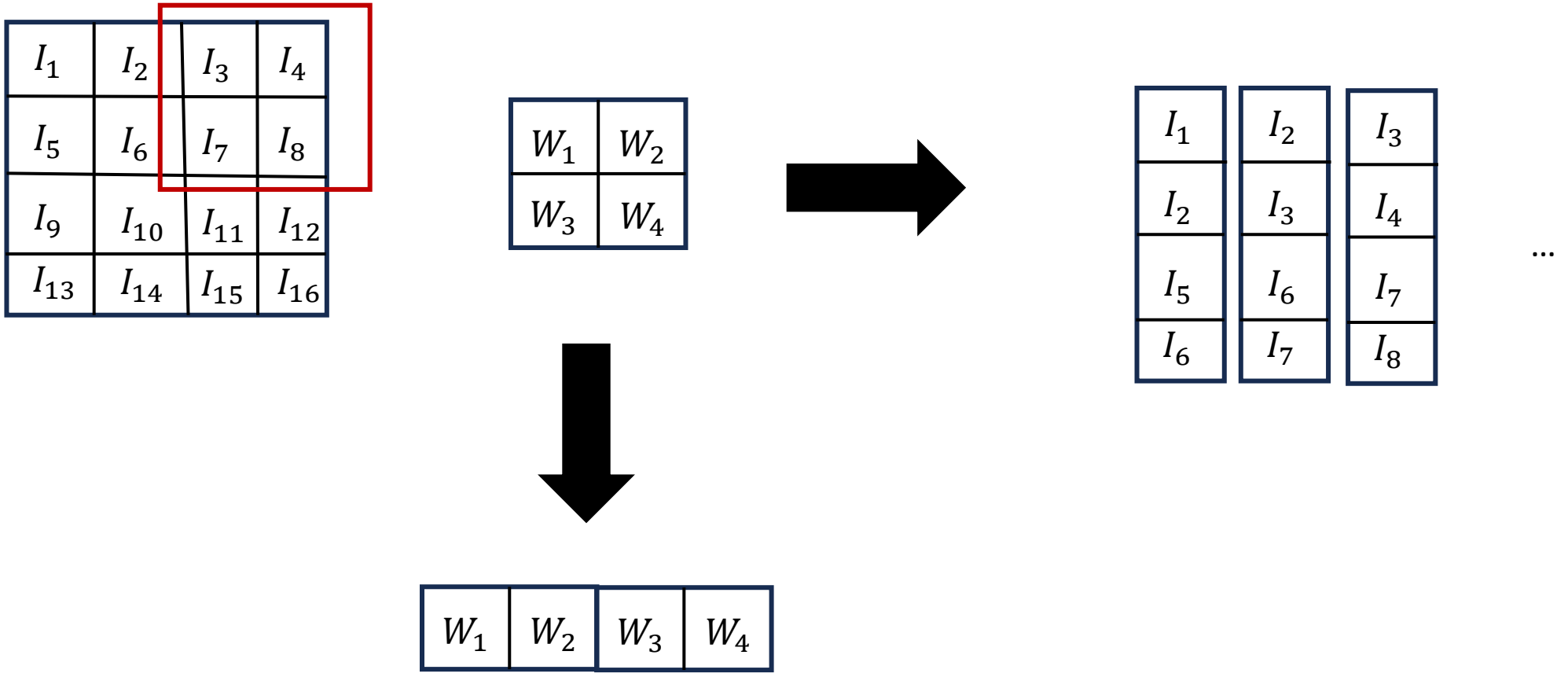
Single Input Single Output Channel

Im2col Algorithm



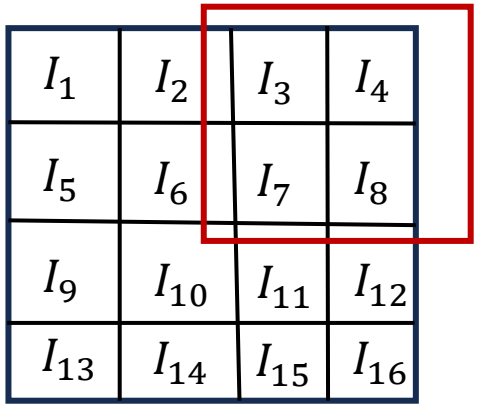
Single Input Single Output Channel

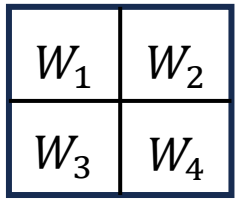
Im2col Algorithm

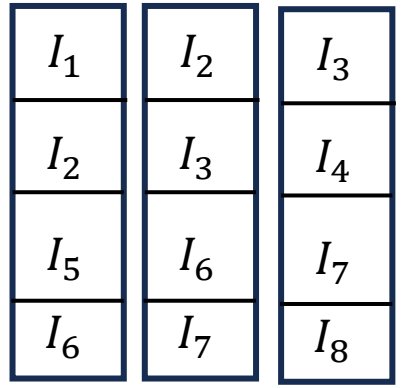


Single Input Single Output Channel

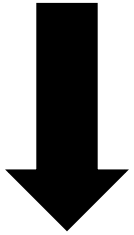
Im2col Algorithm



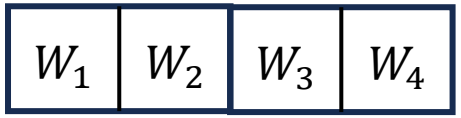




...

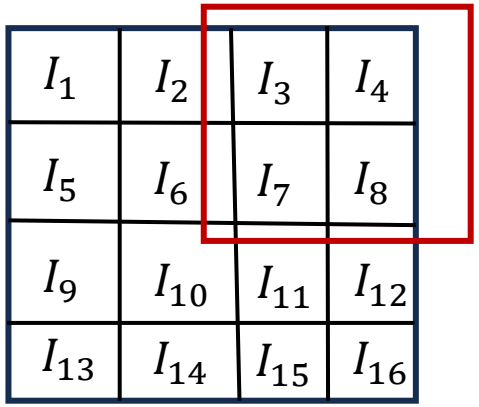


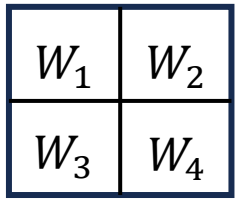
How many columns?

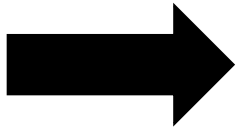


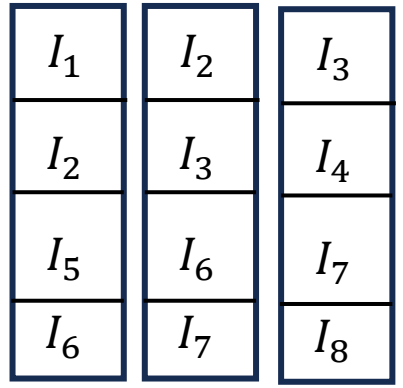
Single Input Single Output Channel

Im2col Algorithm





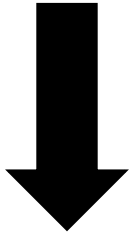


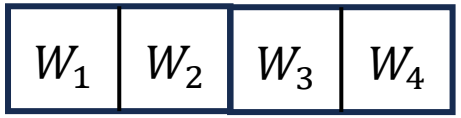


...

How many columns?

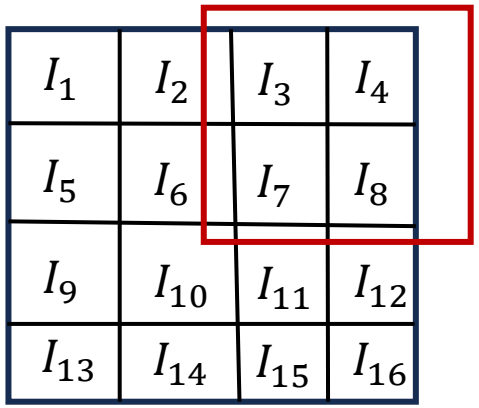
$H \times W$

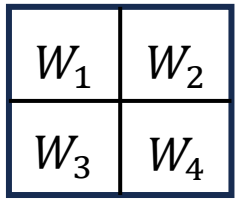


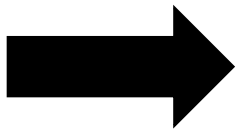


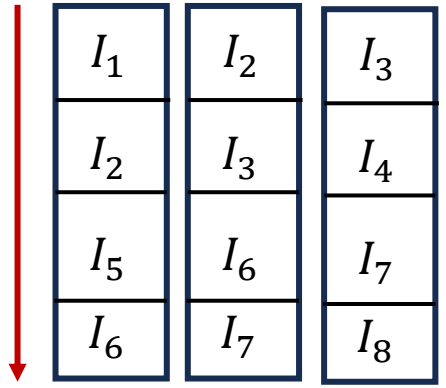
Single Input Single Output Channel

Im2col Algorithm

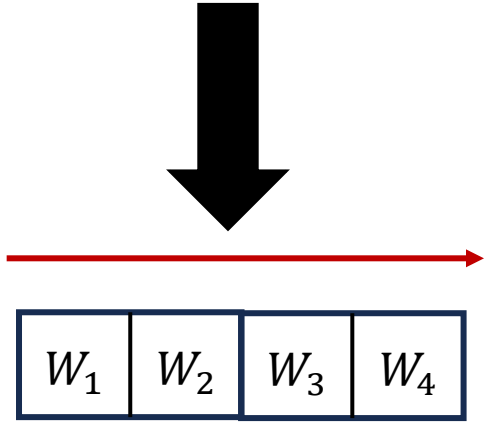








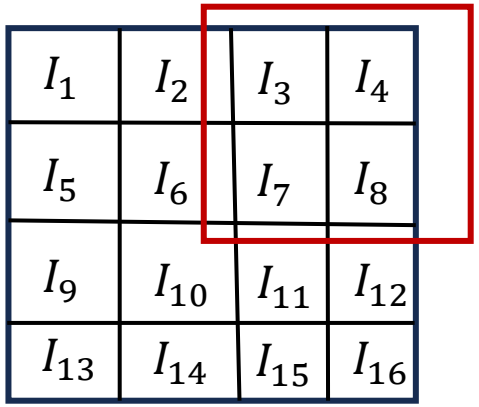
...

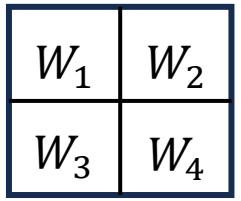


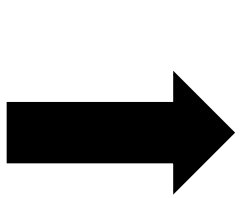
What is the inner dimension of the matrix multiplication?

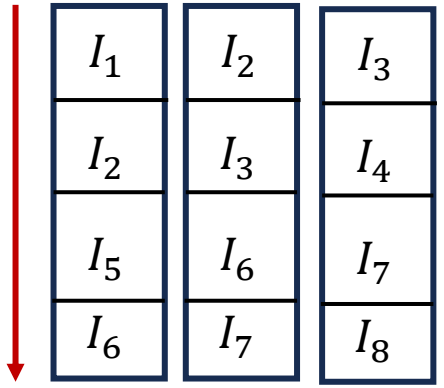
Single Input Single Output Channel

Im2col Algorithm

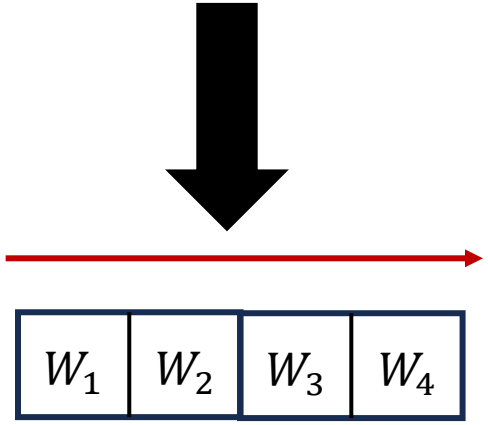








...



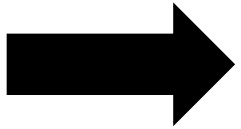
What is the inner dimension of the matrix multiplication? $k \times k$

Single Input Single Output Channel

Im2col Algorithm

I_1	I_2	I_3	I_4
I_5	I_6	I_7	I_8
I_9	I_{10}	I_{11}	I_{12}
I_{13}	I_{14}	I_{15}	I_{16}

I_1	I_2	I_3	I_4
I_5	I_6	I_7	I_8
I_9	I_{10}	I_{11}	I_{12}
I_{13}	I_{14}	I_{15}	I_{16}

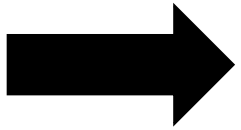


I_1	I_2	I_3
I_2	I_3	I_4
I_5	I_6	I_7
I_6	I_7	I_8

I_1	I_2	I_3
I_2	I_3	I_4
I_5	I_6	I_7
I_6	I_7	I_8

W_1	W_2
W_3	W_4

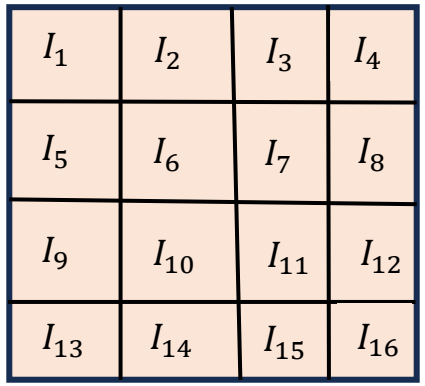
W_1	W_2
W_3	W_4

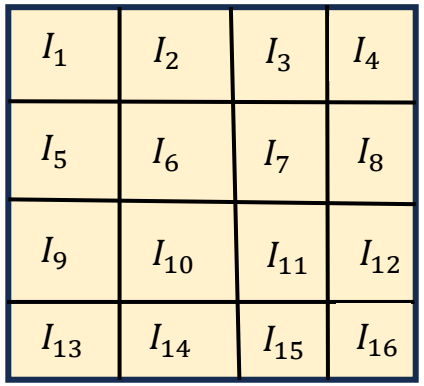


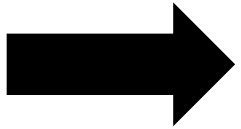
W_1	W_2	W_3	W_4	W_1	W_2	W_3	W_4

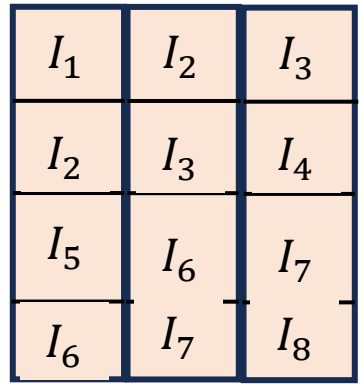
Multiple Input Single Output Channel

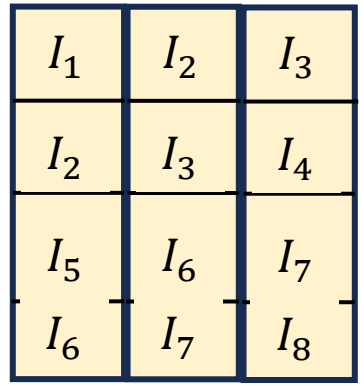
Im2col Algorithm



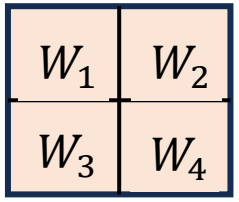


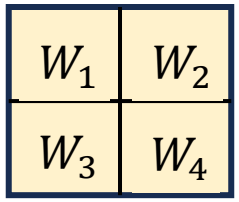


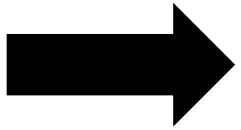


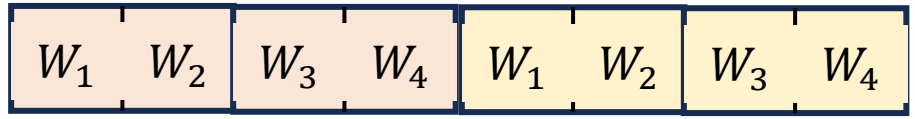


Notice: Input channels increase the row dimension of weight matrix and column dimension of the input feature matrix



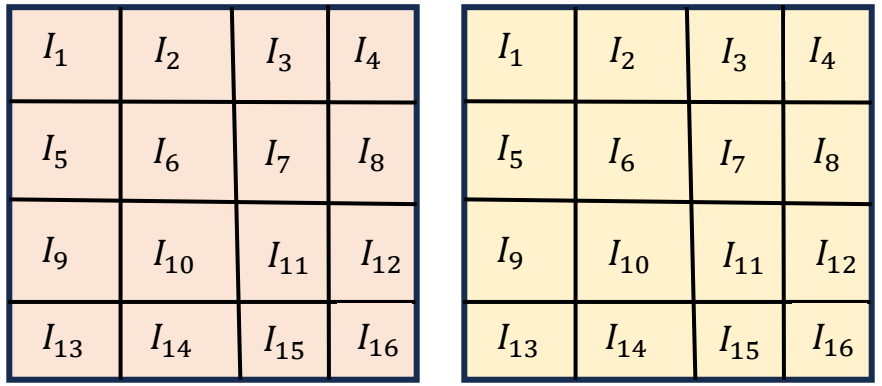


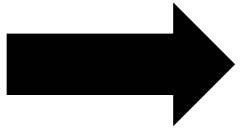


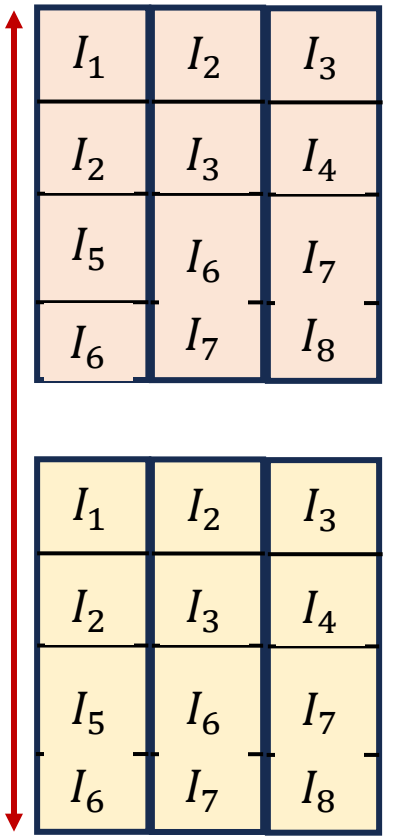


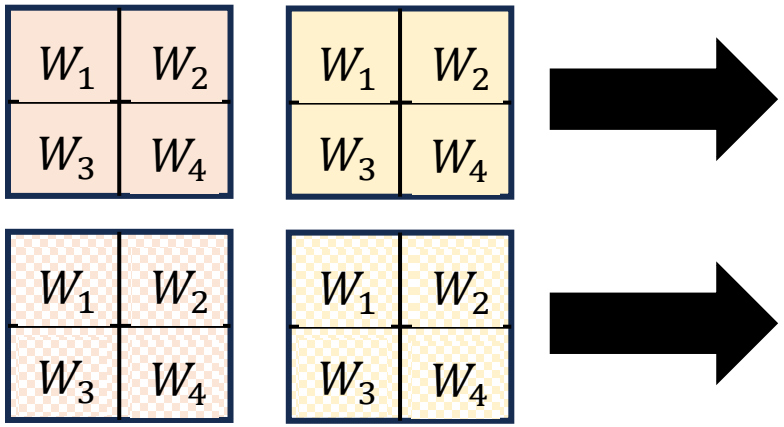
Multiple Input Single Output Channel

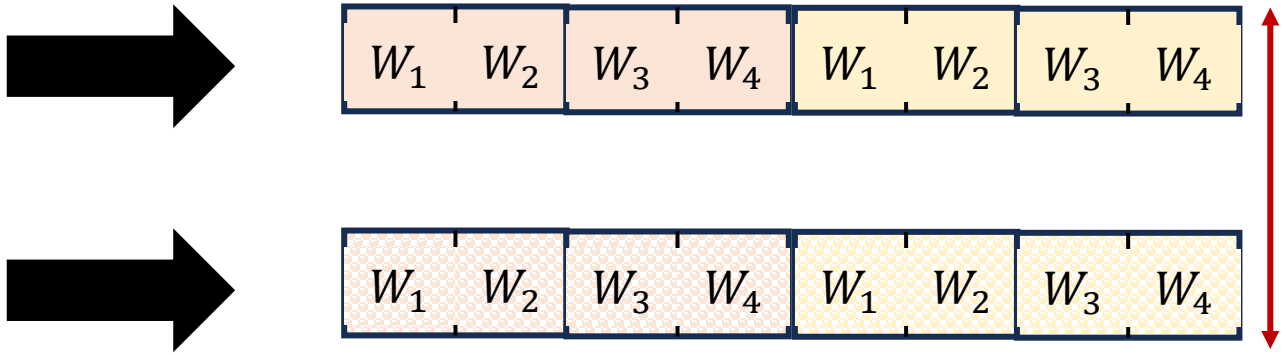
Im2col Algorithm











Multiple Input Single Multiple Channel

Im2Col Algorithm

- Input Matrix -$(k \times k \times C_{in}) \times (H \times W)$
  - $k \times k$ increase in size
- Filter Matrix - $(\mathrm{C}_{out}) \times (k \times k \times C_{in})$
  - No increase in size
- Output Matrix - $C_{out} \times (H \times W)$
  - No increase in size

Implicit GEMM based Im2Col

- In practice, the input matrix is not duplicated
- Rather tiles/blocks are fetched into the shared cache of SMP by indexing appropriate addresses

Im2col Algorithm

I_1	I_2	I_3	I_4
I_5	I_6	I_7	I_8
I_9	I_{10}	I_{11}	I_{12}
I_{13}	I_{14}	I_{15}	I_{16}

I_1	I_2	I_3	I_4
I_5	I_6	I_7	I_8
I_9	I_{10}	I_{11}	I_{12}
I_{13}	I_{14}	I_{15}	I_{16}

W_1	W_2
W_3	W_4

W_1	W_2
W_3	W_4

W_1	W_2
W_3	W_4

W_1	W_2
W_3	W_4

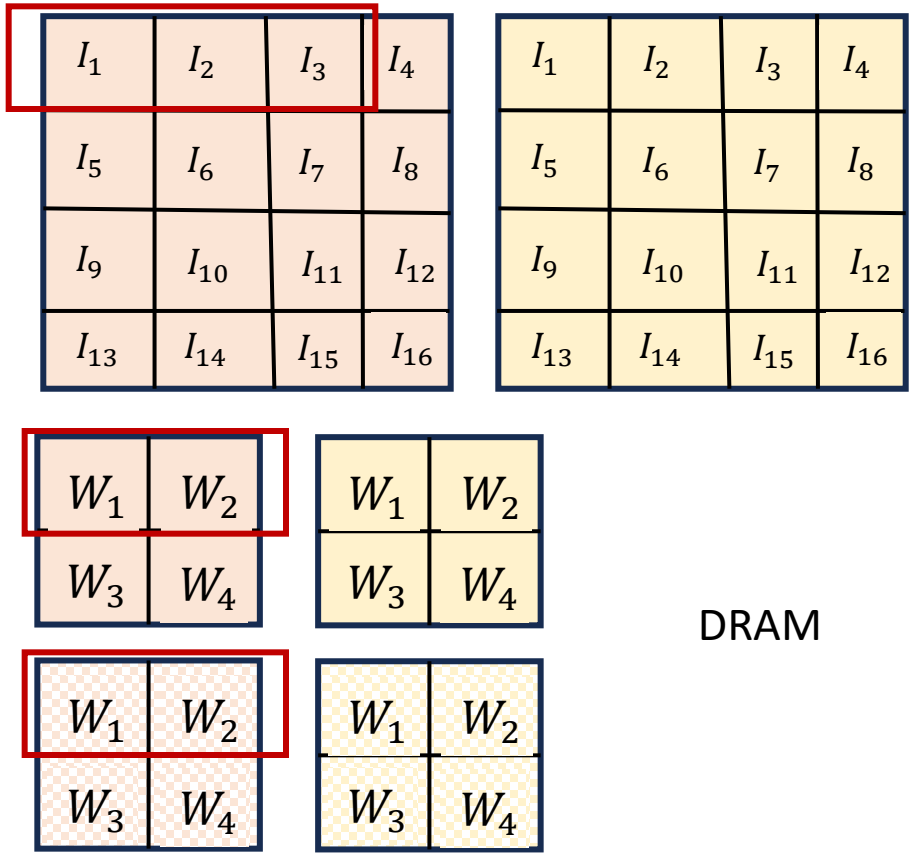
DRAM

Assume blocked matrix multiplication with block sizes of 2×2

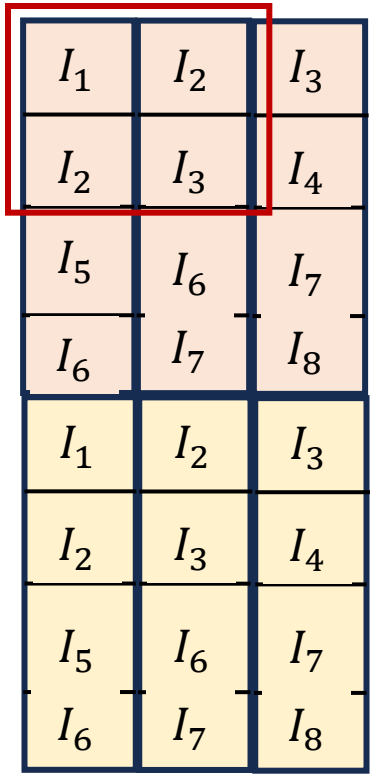
I_1	I_2	I_3
I_2	I_3	I_4
I_5	I_6	I_7
I_6	I_7	I_8
I_1	I_2	I_3
I_2	I_3	I_4
I_5	I_6	I_7
I_6	I_7	I_8

W_1	W_2	W_3	W_4	W_1	W_2	W_3	W_4
W_1	W_2	W_3	W_4	W_1	W_2	W_3	W_4

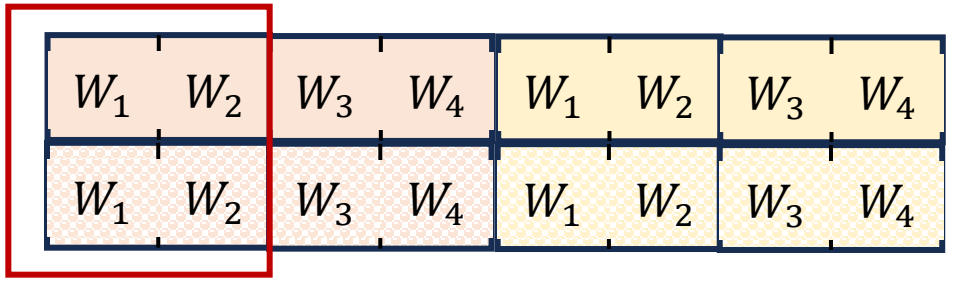
Im2col Algorithm



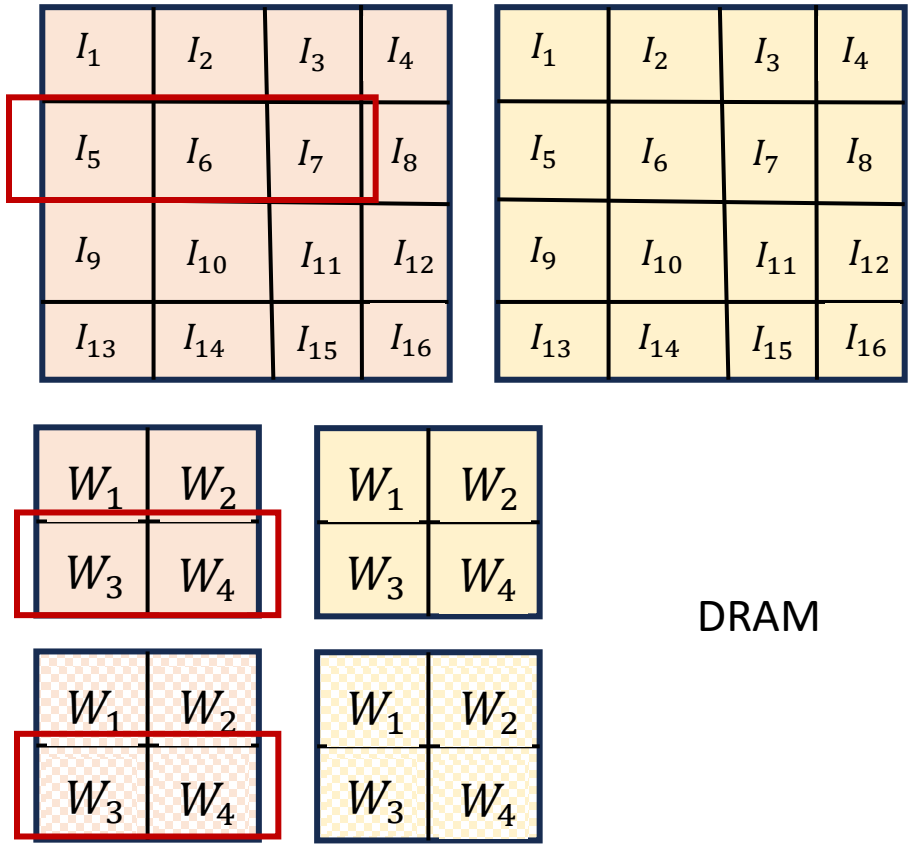
Shared Memory/Cache



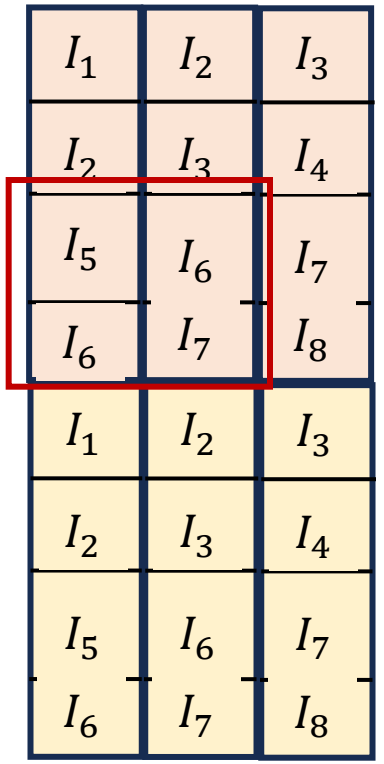
Shared Memory/Cache



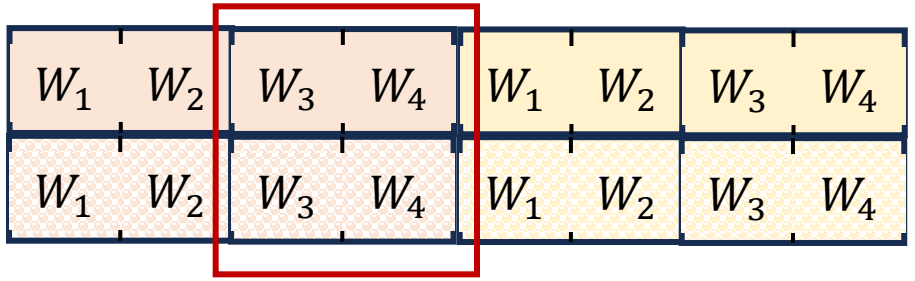
Im2col Algorithm



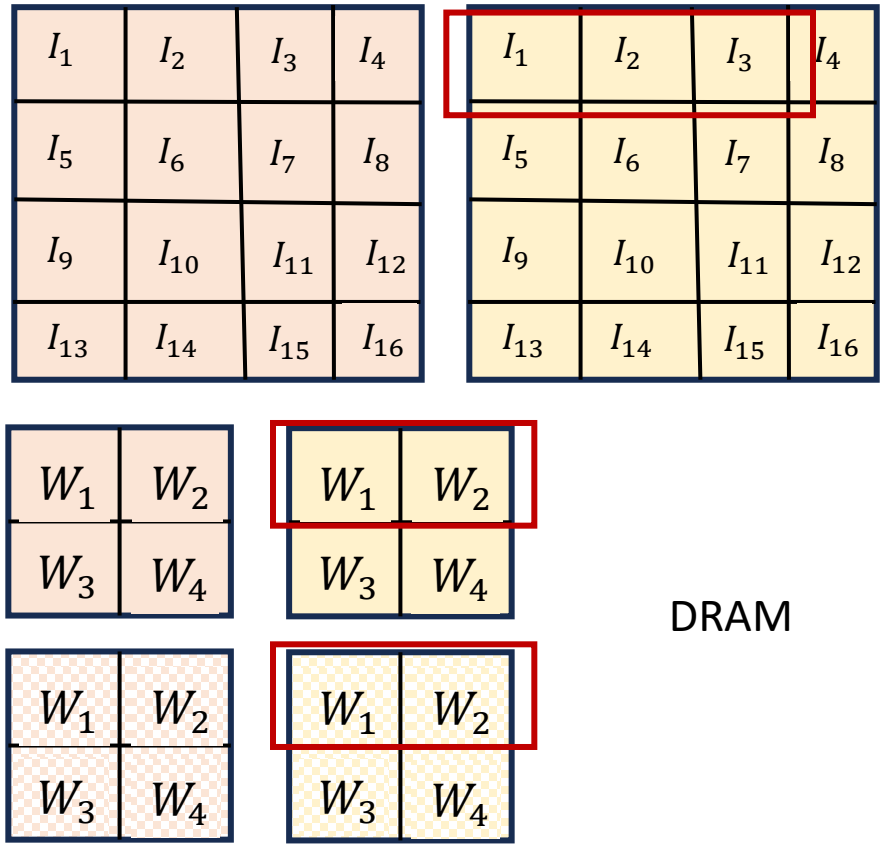
Shared Memory/Cache



Shared Memory/Cache



Im2col Algorithm



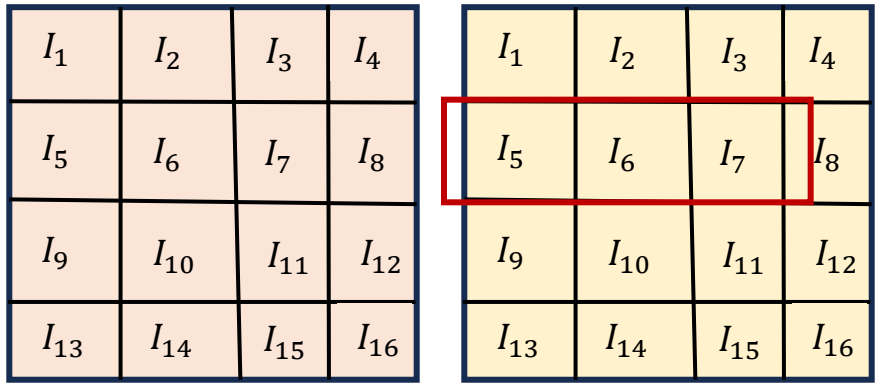
Shared Memory/Cache

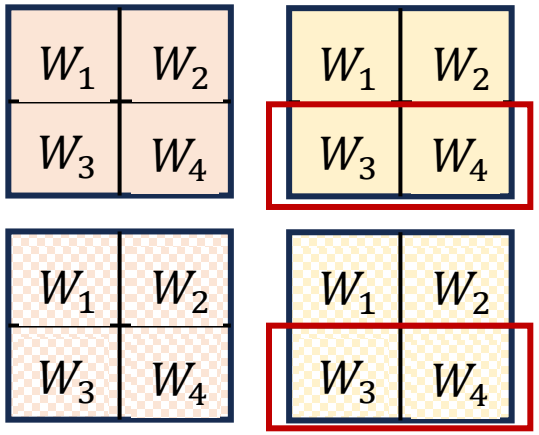
I_1 I_2 I_3
I_2 I_3 I_4
I_5 I_6 I_7
I_6 I_7 I_8

Shared Memory/Cache

W_1 W_2 W_3 W_4 W_1 W_2 W_3 W_4
W_1 W_2 W_3 W_4 W_1 W_2 W_3 W_4

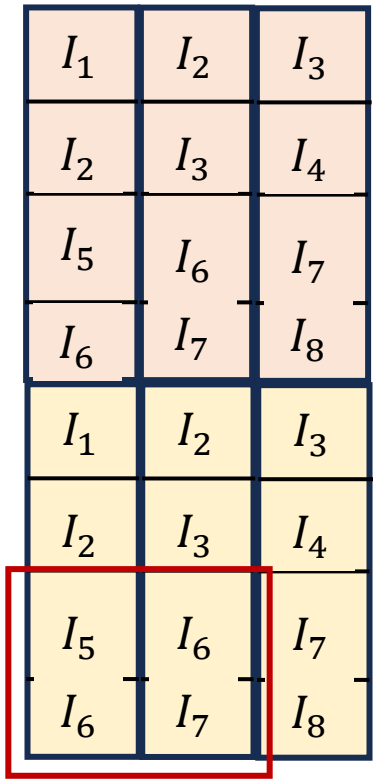
Im2col Algorithm



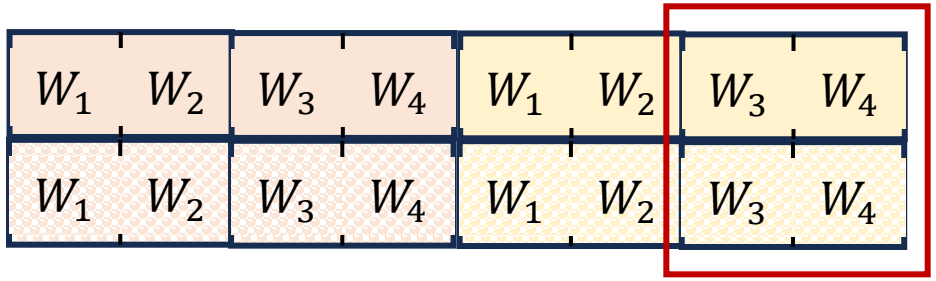


DRAM

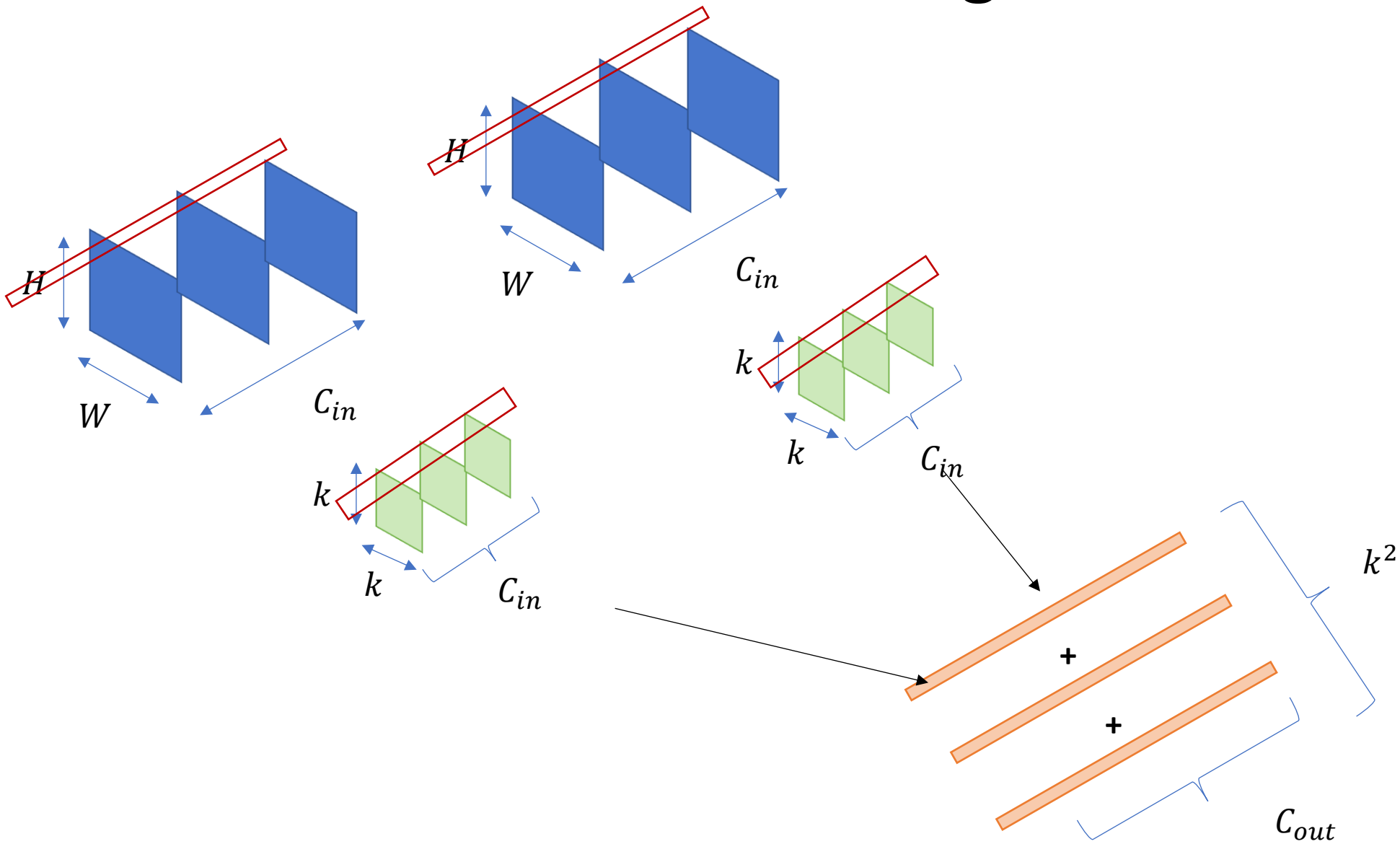
Shared Memory/Cache



Shared Memory/Cache



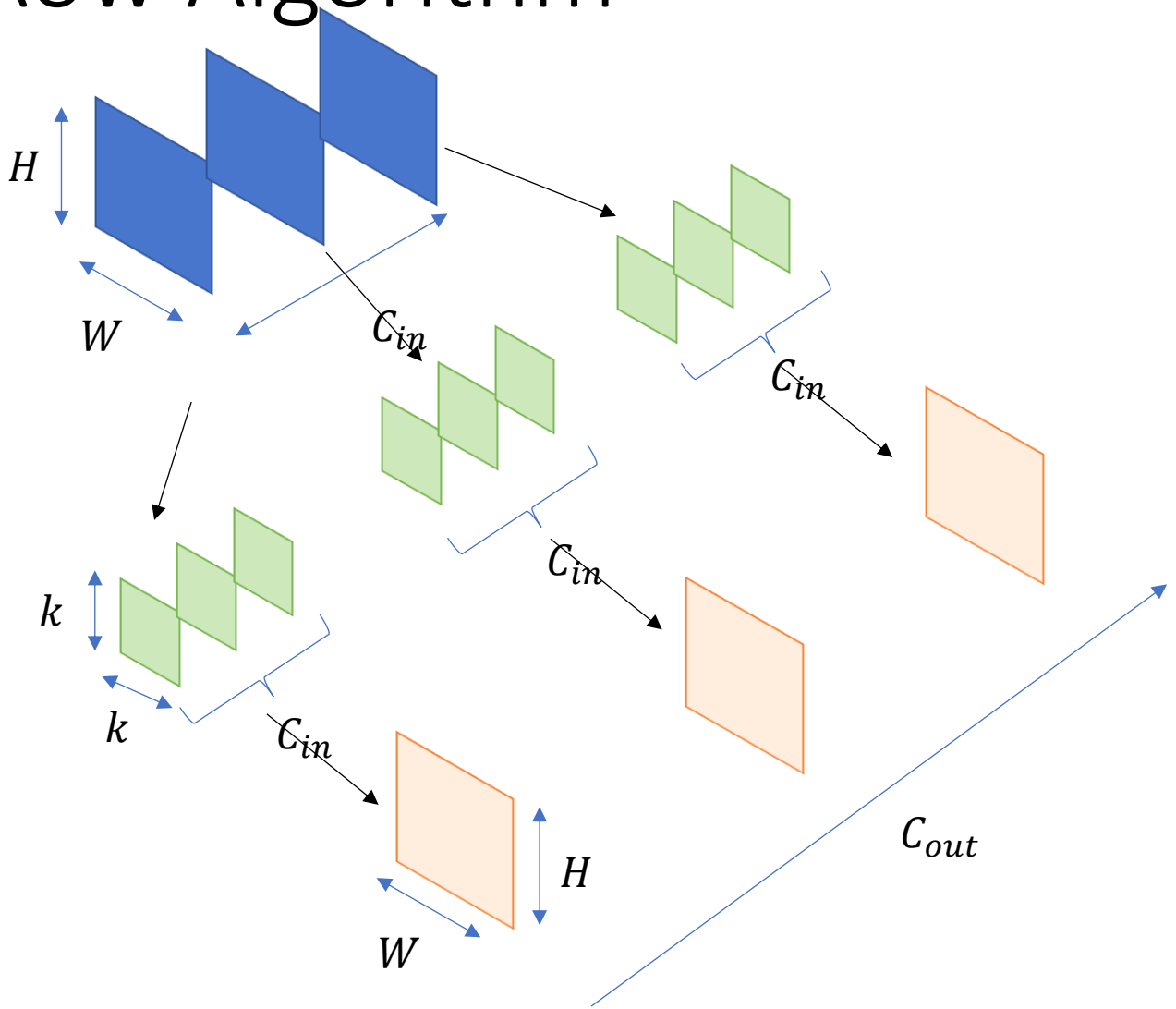
Kn2Row Algorithm



Kn2Row Algorithm

Kernel Matrix

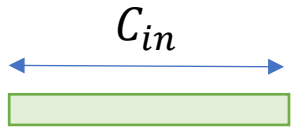
- Take a strip of C_{in} pixels of kernels of same output channel to form a row
- Create k^2 rows for each output channel
- Repeat the above for all output channels

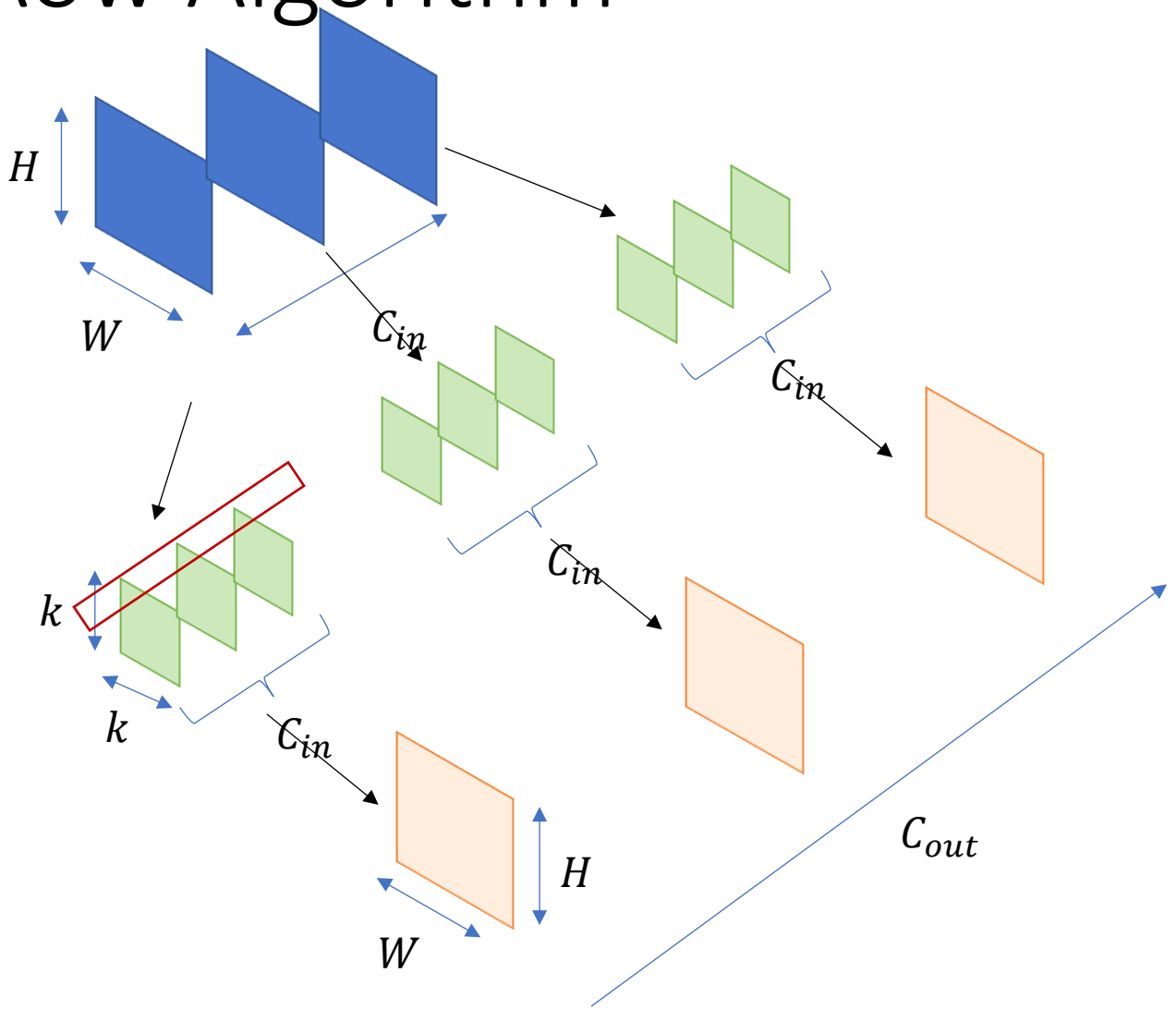


Kn2Row Algorithm

Kernel Matrix

- Take a strip of C_{in} pixels of kernels of same output channel to form a row
- Create k^2 rows for each output channel
- Repeat the above for all output channels

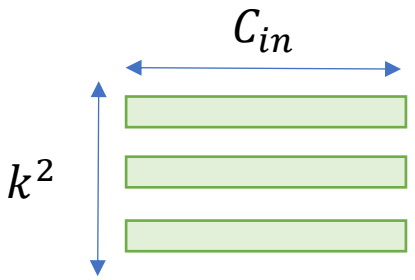


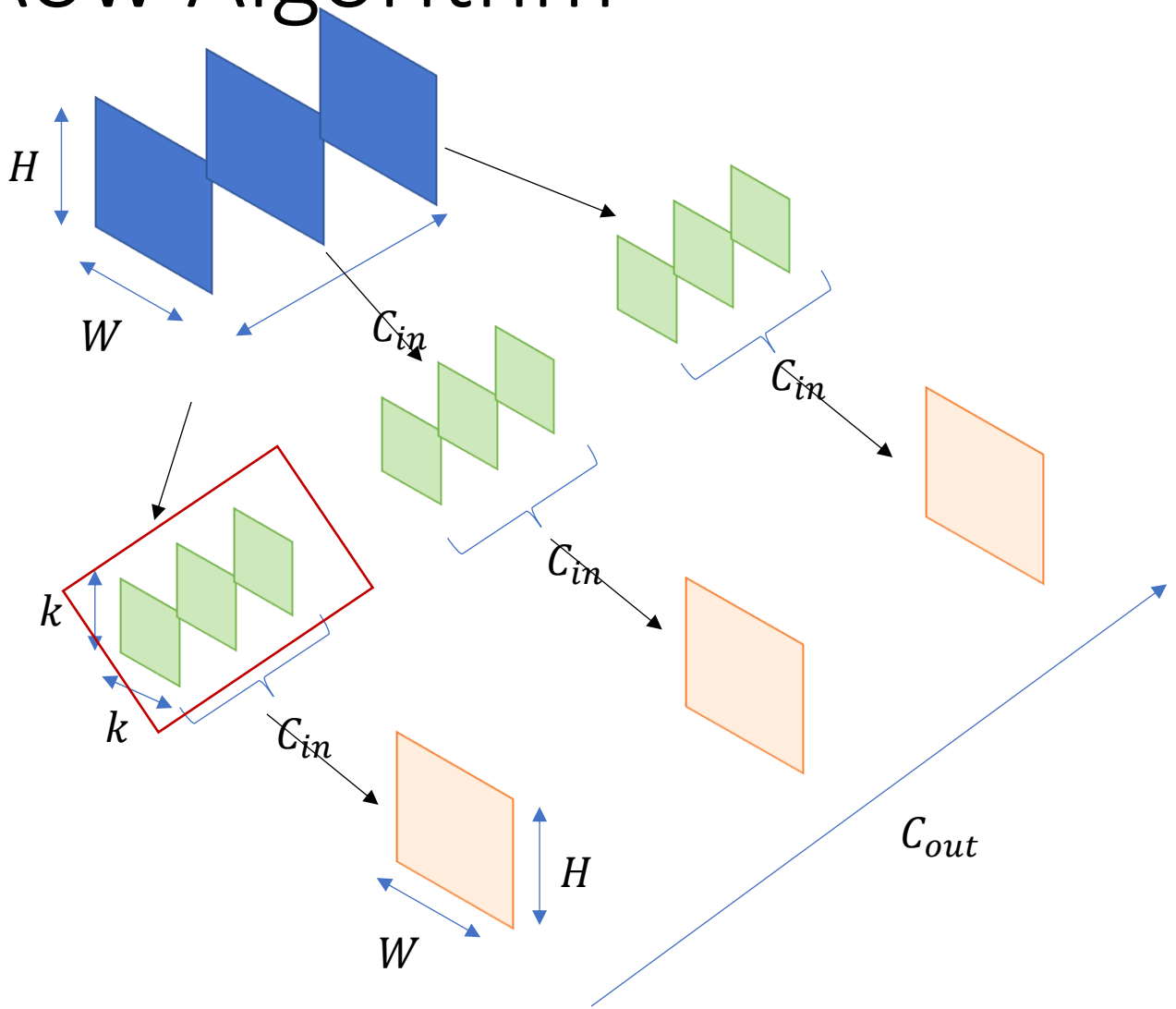


Kn2Row Algorithm

Kernel Matrix

- Take a strip of C_{in} pixels of kernels of same output channel to form a row
- Create k^2 rows for each output channel
- Repeat the above for all output channels

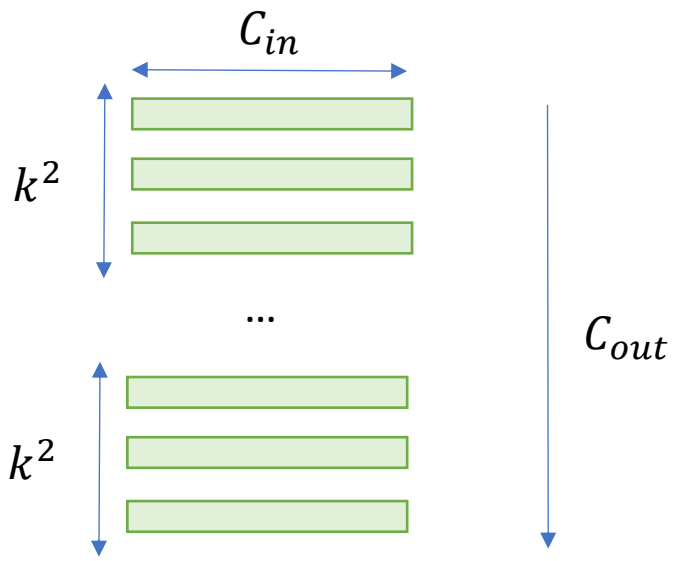


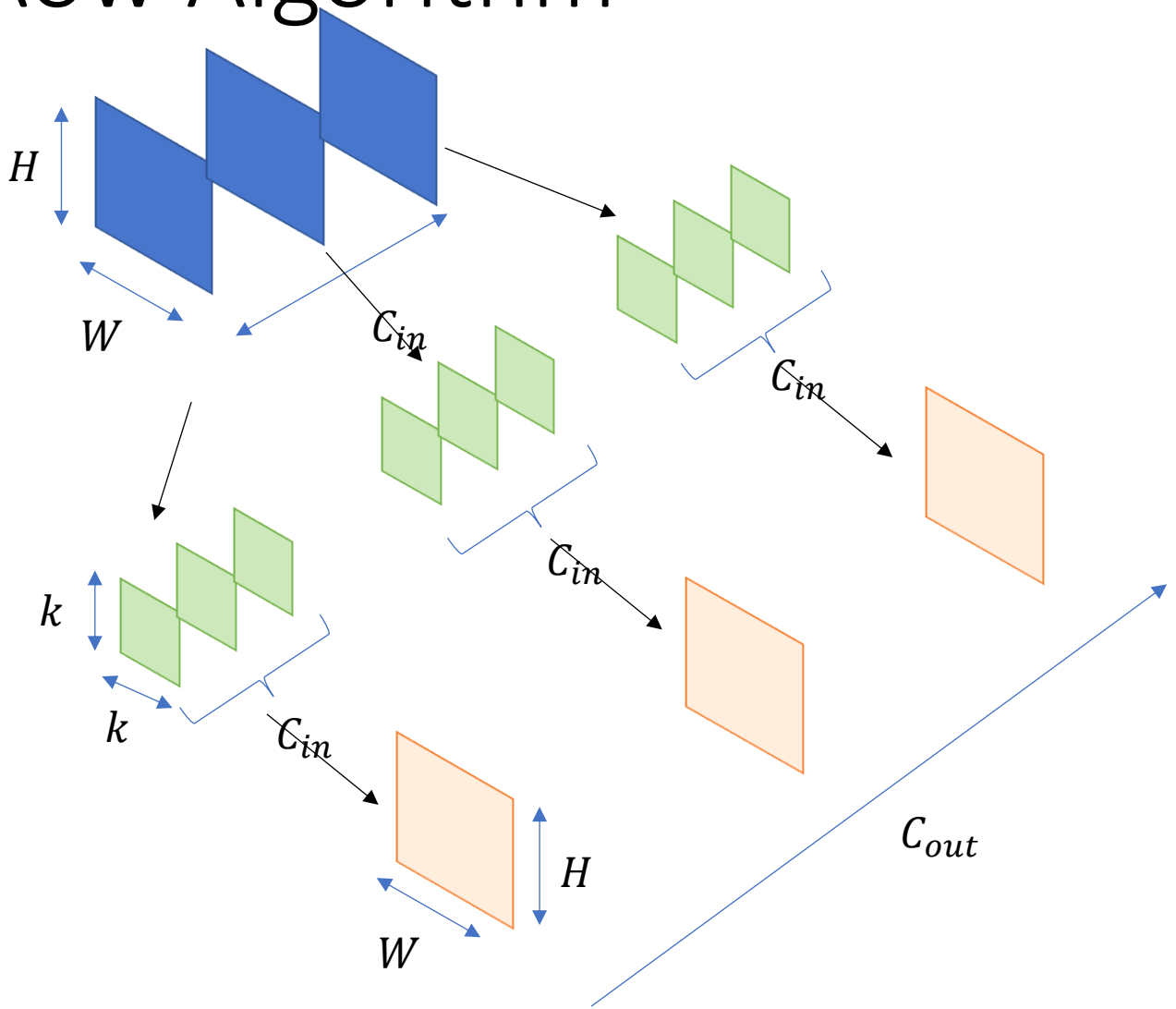


Kn2Row Algorithm

Kernel Matrix

- Take a strip of C_{in} pixels of kernels of same output channel to form a row
- Create k^2 rows for each output channel
- Repeat the above for all output channels

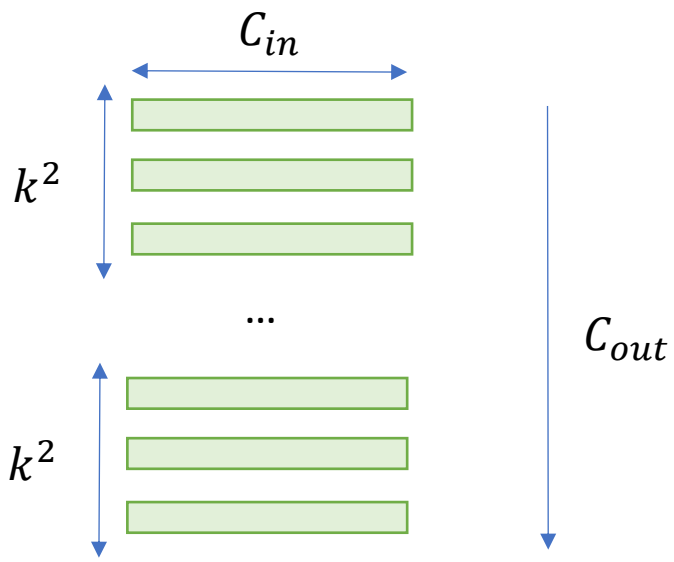


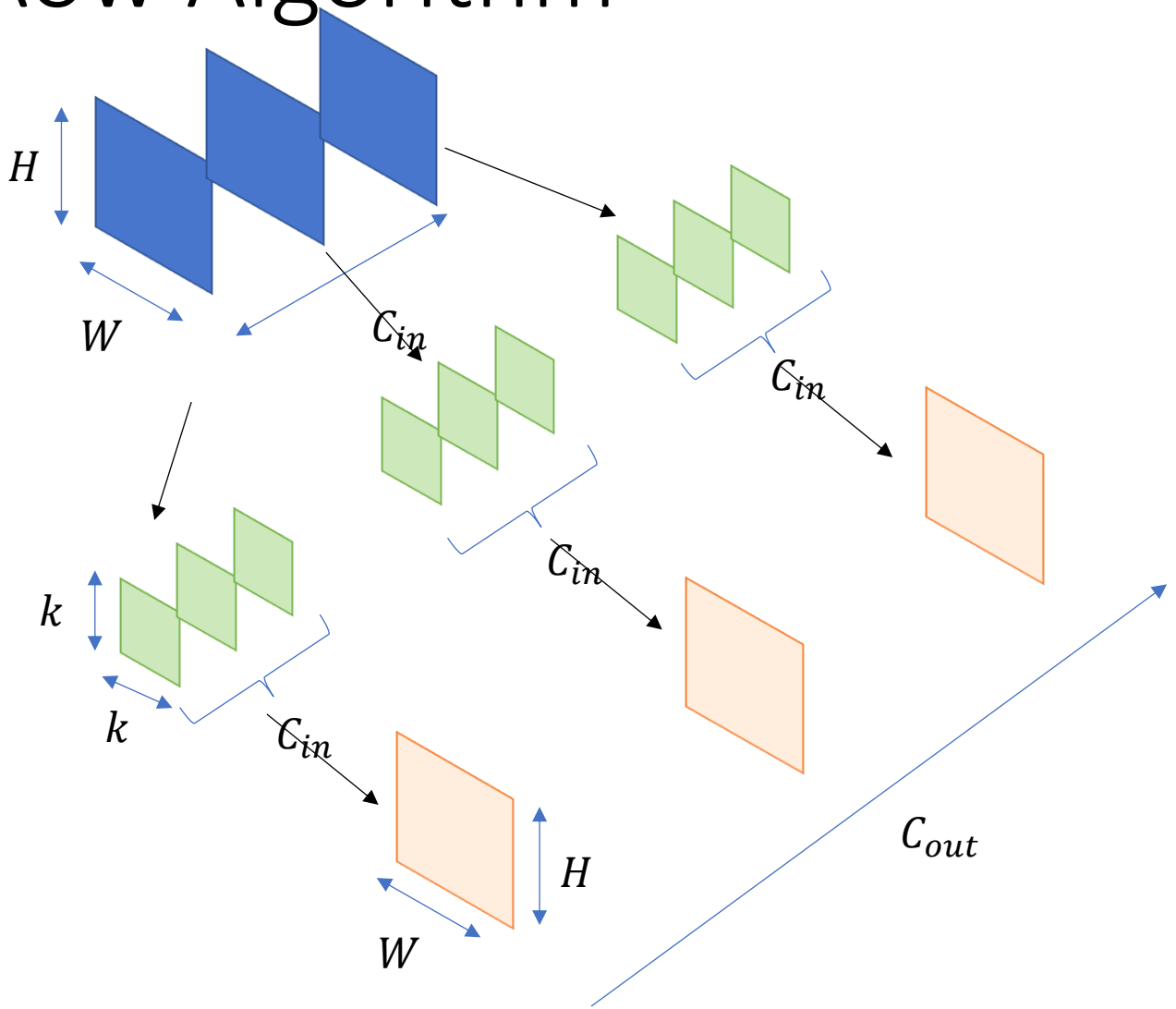


Kn2Row Algorithm

Kernel Matrix Size

$$(C_{out} \times k^2) \times C_{in}$$

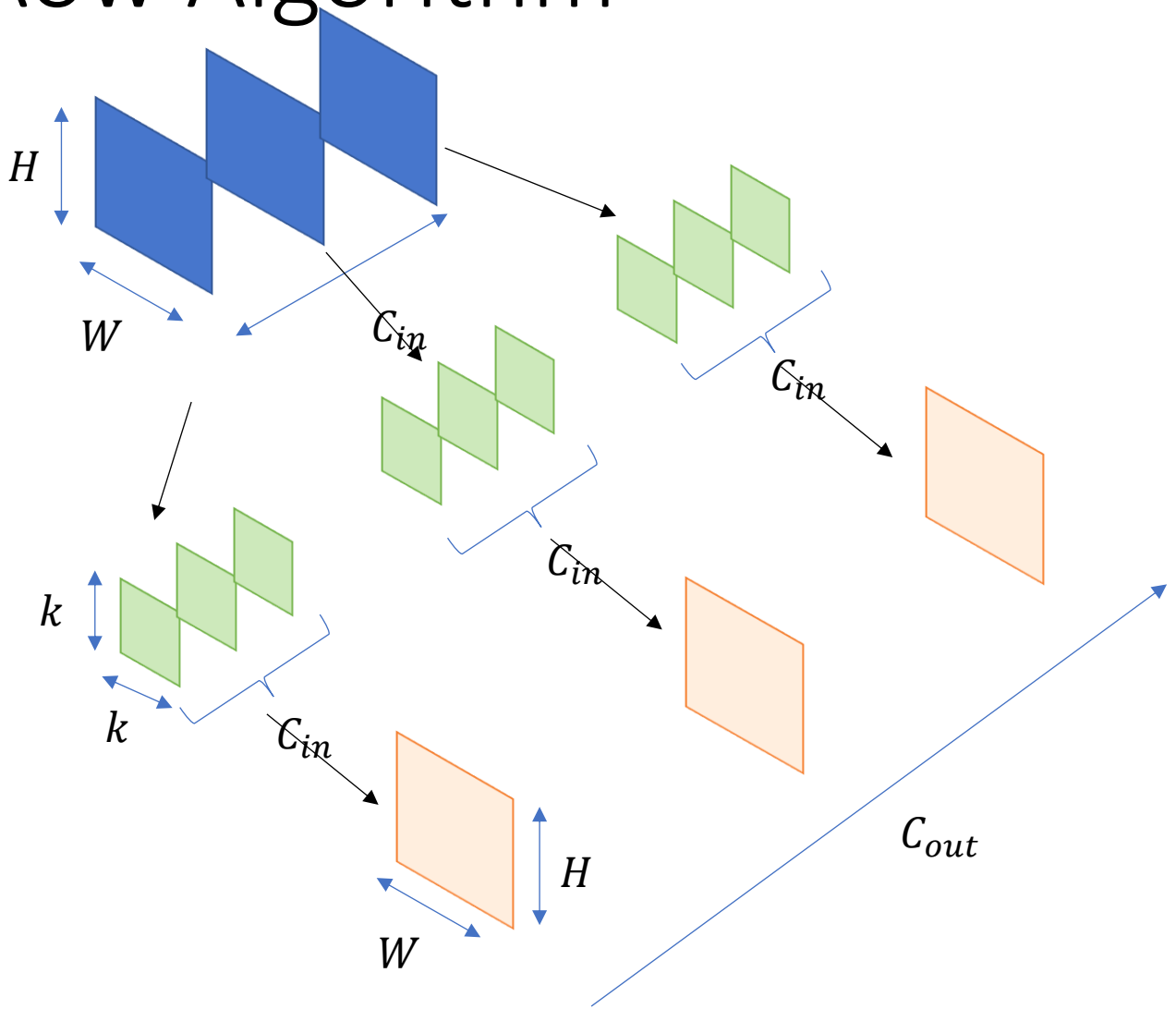




Kn2Row Algorithm

Input Matrix

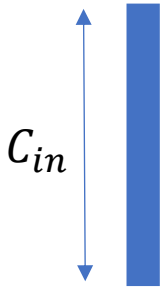
- Take a strip of C_{in} pixels of images to form a column
- Create $H \times W$ such column

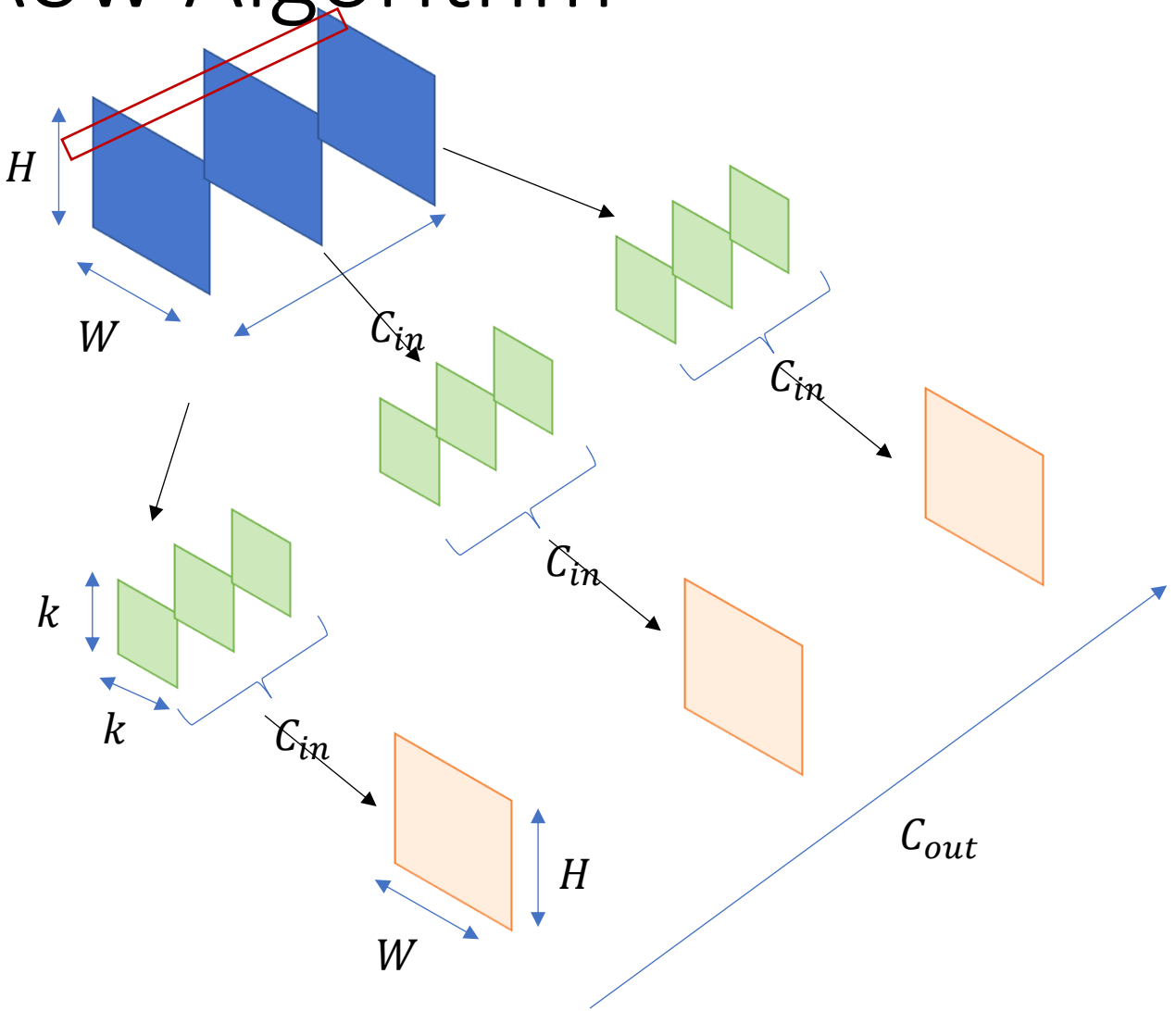


Kn2Row Algorithm

Input Matrix

- Take a strip of C_{in} pixels of images to form a column
- Create $H \times W$ such column

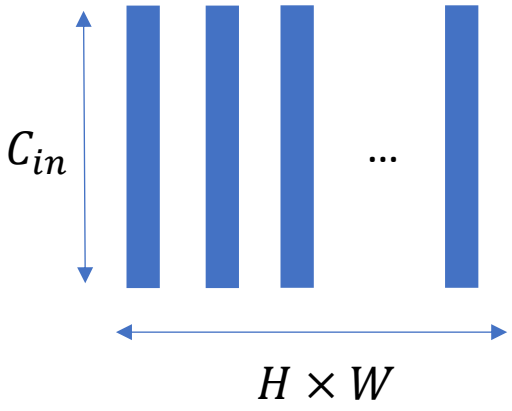


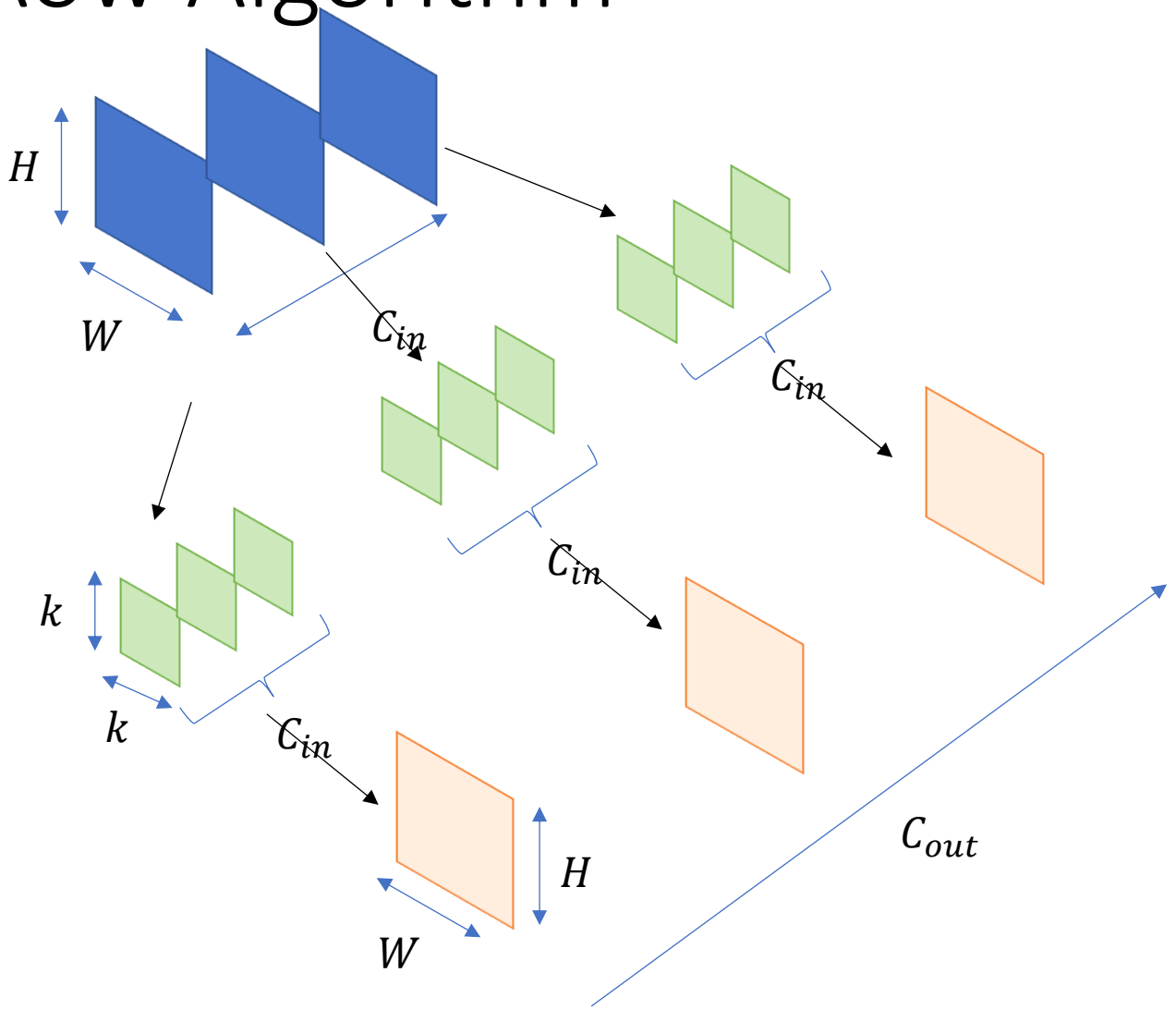


Kn2Row Algorithm

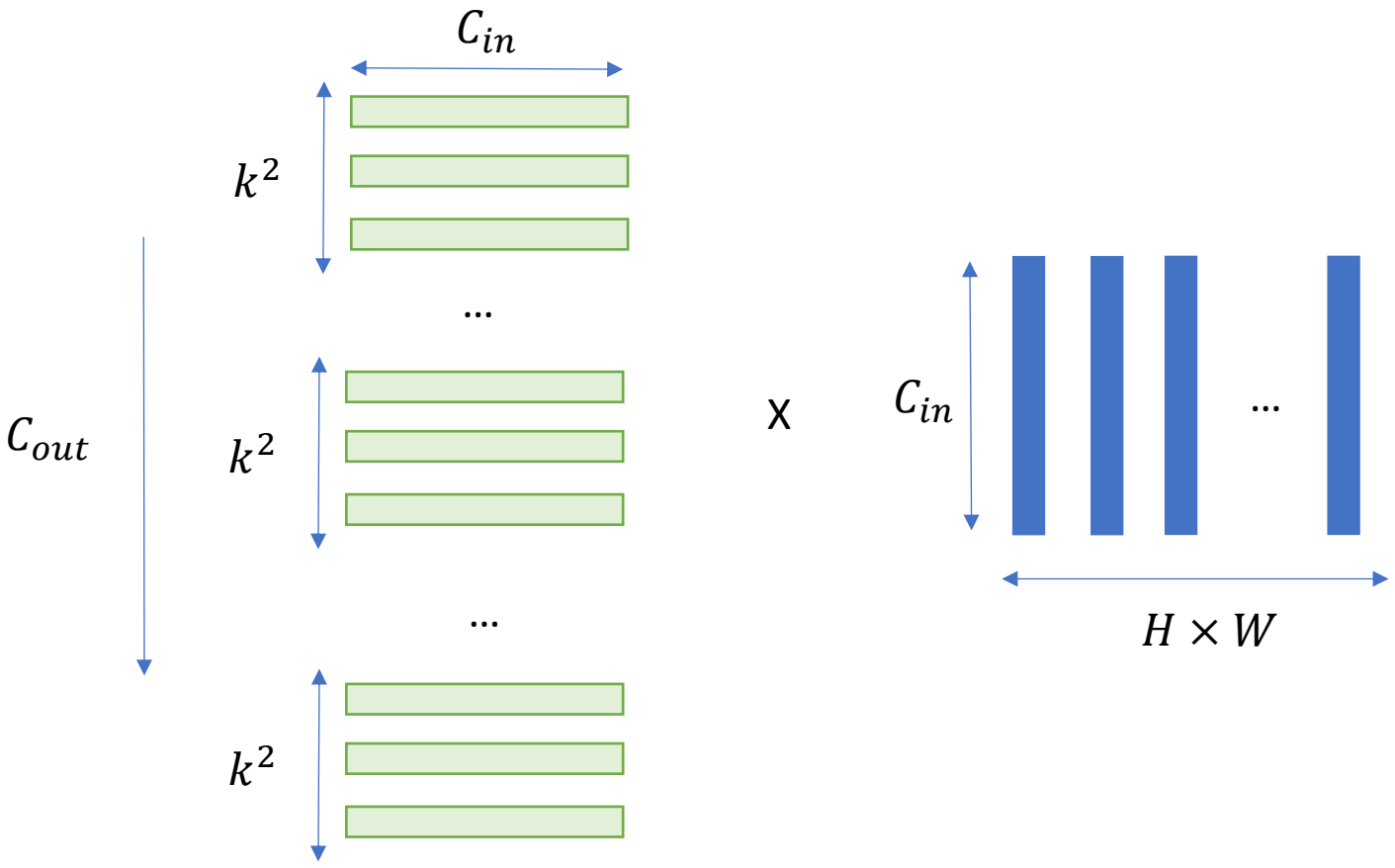
Input Matrix

- Take a strip of C_{in} pixels of images to form a column
- Create $H \times W$ such column

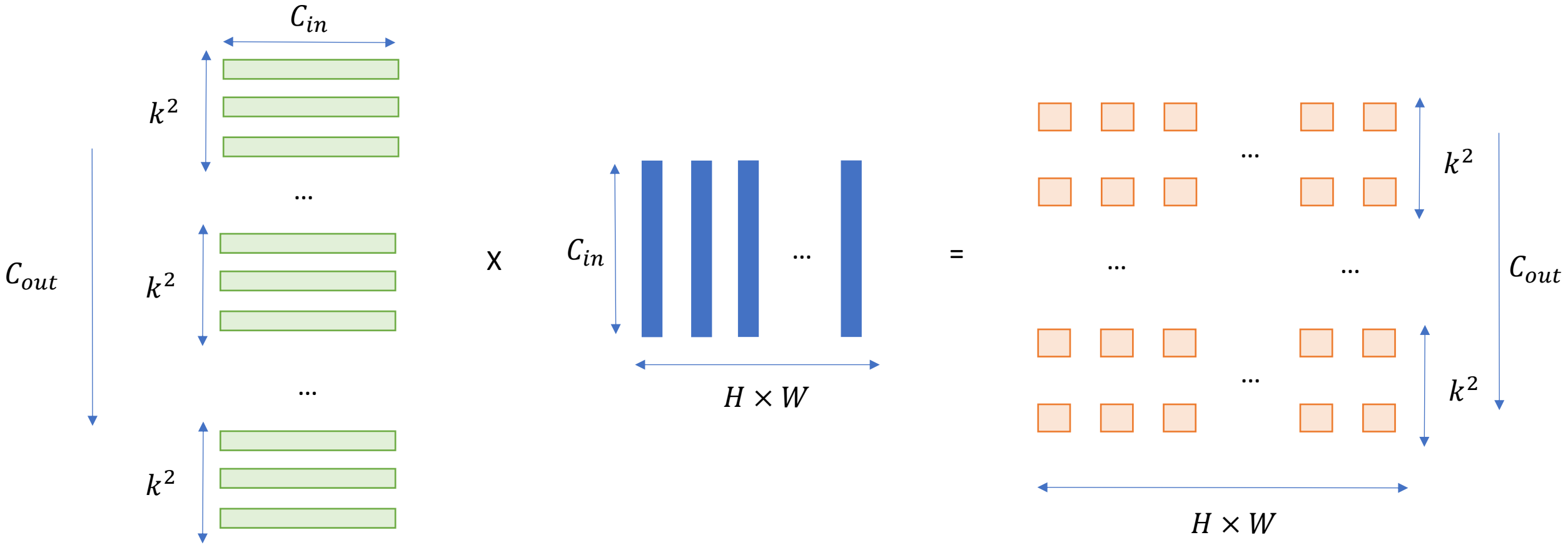




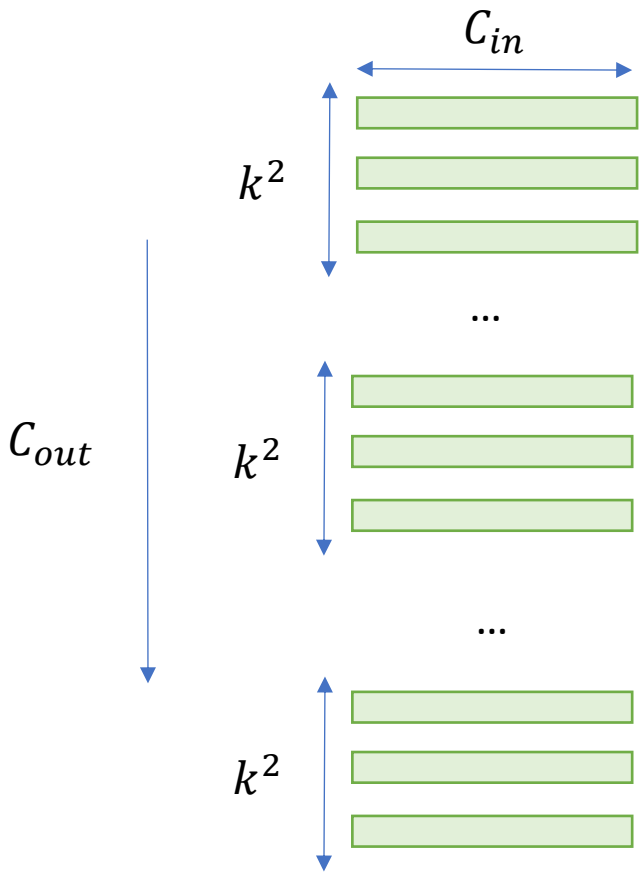
Kn2Row Algorithm



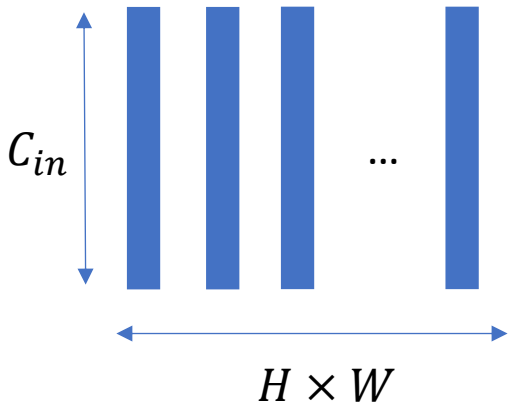
Kn2Row Algorithm



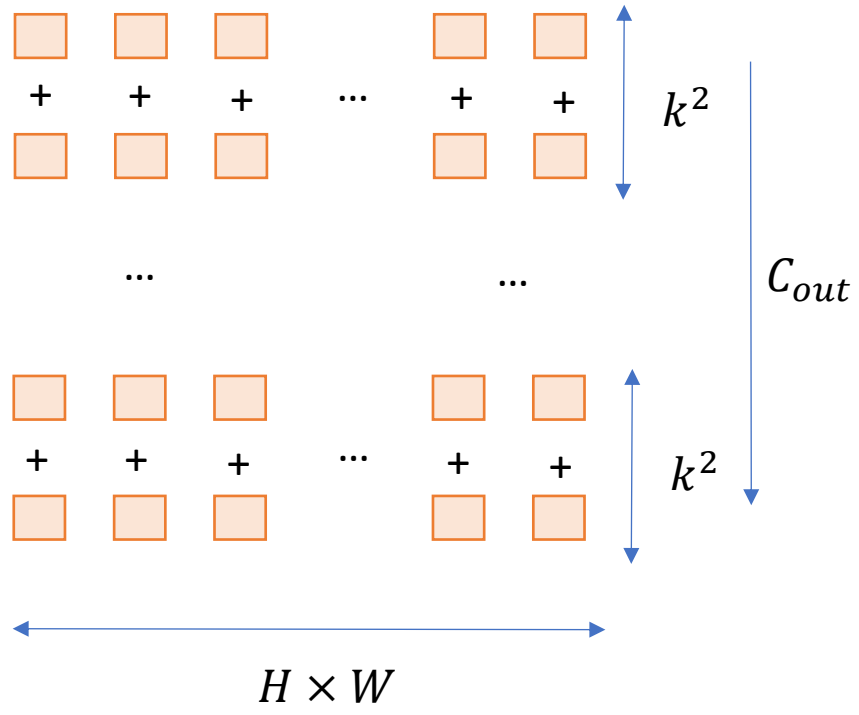
Kn2Row Algorithm



X

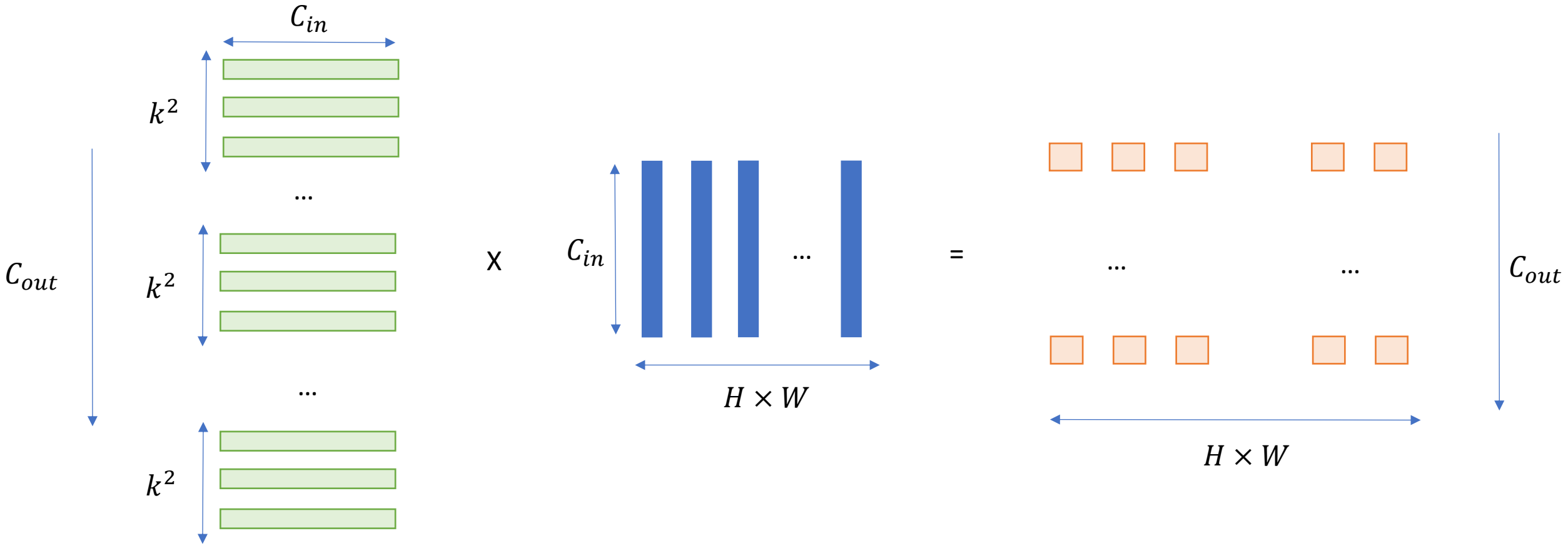


=



*The sums above are shift-add,

Kn2Row Algorithm



Kn2Row Algorithm

- Input Matrix -$(C_{in}) \times (H \times W)$
- Filter Matrix - $\left(\mathrm{C}_{out} \times k^2\right) \times (C_{in})$
  - No increase in size
- Output Matrix -$(H \times W) \times (C_{out} \times k^2)$
  - $k \times k$ increase in size

Outline

- Convolution as Matrix Multiplication – Motivation
- Im2Col and Kn2Row Algorithms
- Scalar Matrix Multiplication Algorithm

Scalar Matrix Multiplication

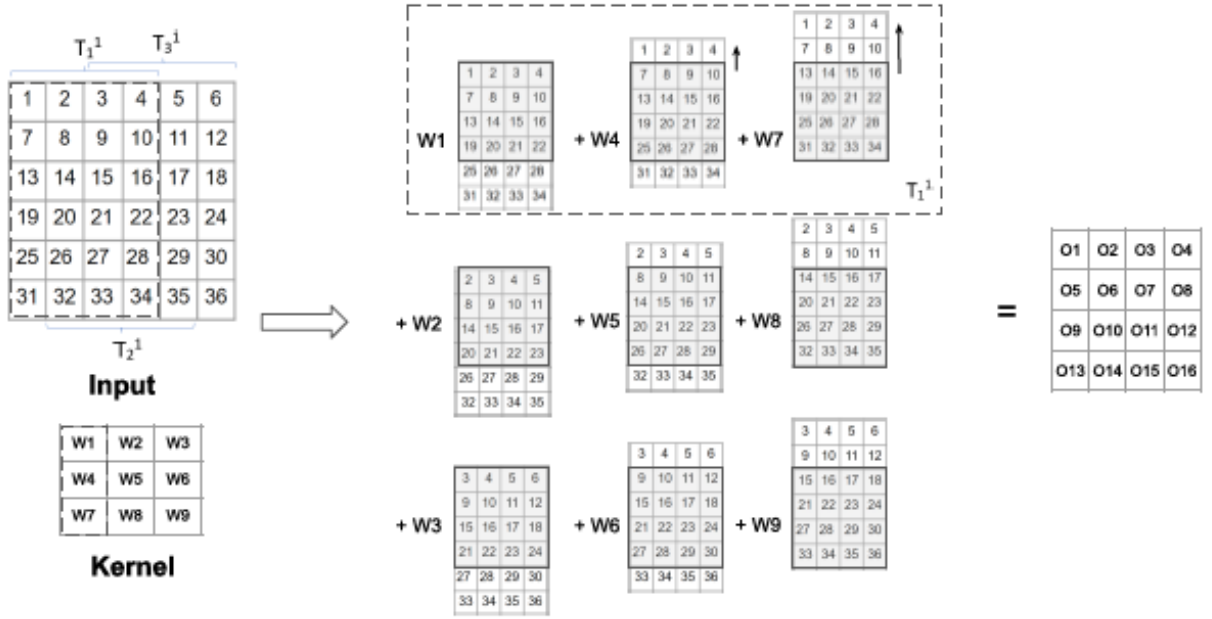
- Approach till now
  - Transform input image and kernels into matrix
  - Directly use BLAS kernels for Matrix multiplication

- No explicit parallelization needed
  - BLAS takes care of that

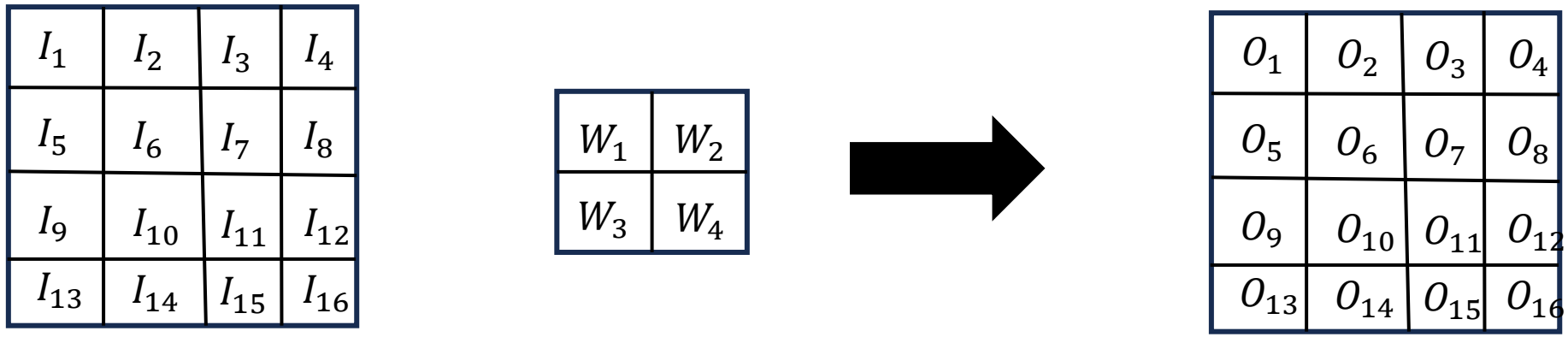
- Challenge - k^2 increase in memory requirements (input or output)

Scalar Matrix Multiplication

- Key Idea: Output image is a linear combination of “portions” of input image with weights as kernels
- Avoid matrix multiplication, perform scaling and matrix addition operations
  - Scaling: Multiply entire vector/matrix with a single scalar value

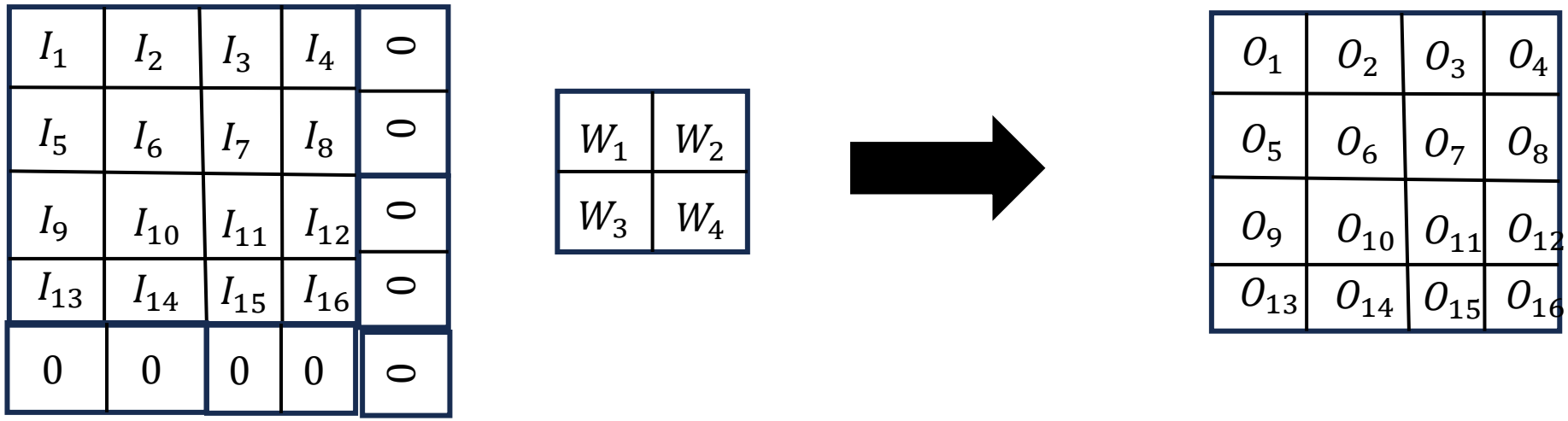


Scalar Matrix Multiplication



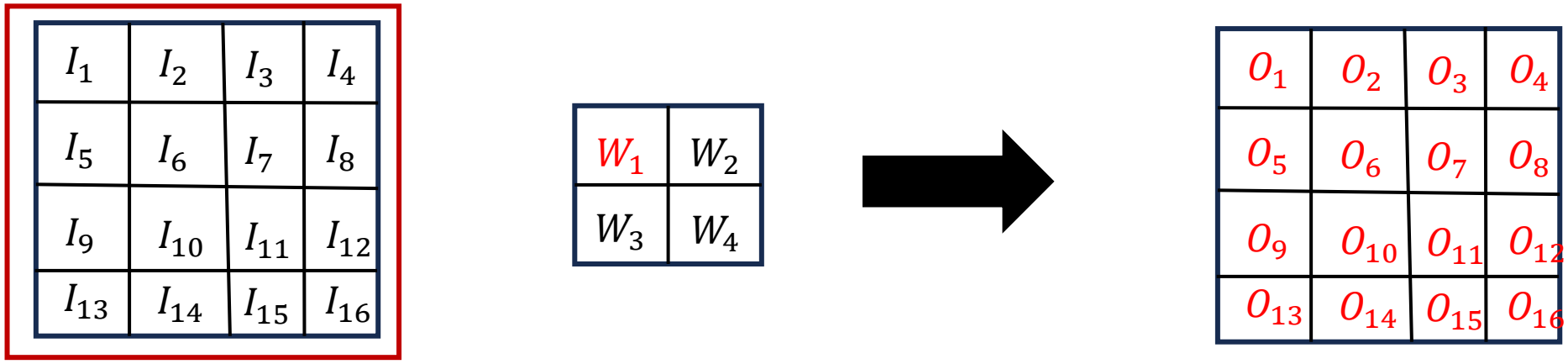
Single Input, Single Output Channel

Scalar Matrix Multiplication

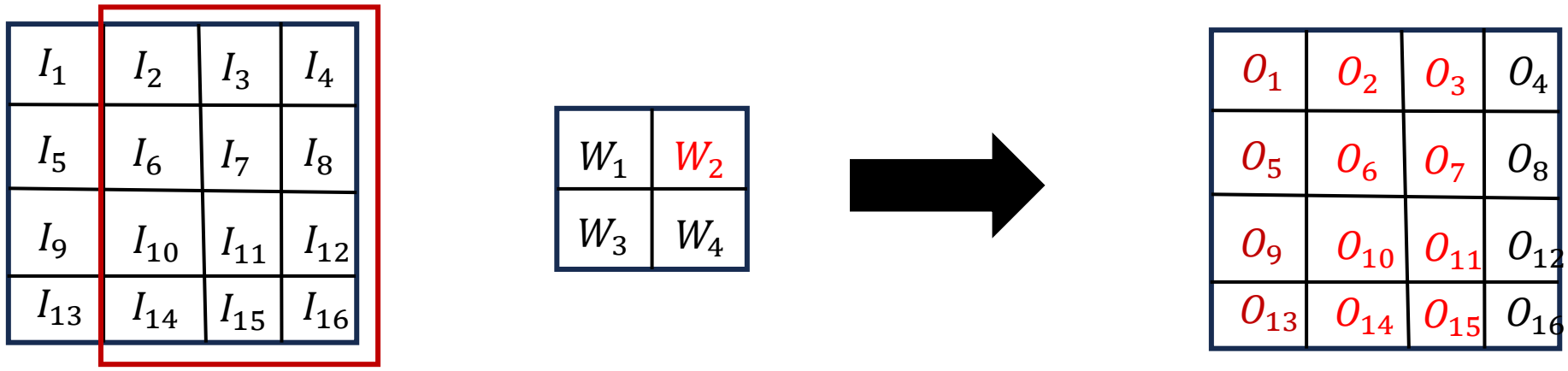


Single Input, Single Output Channel

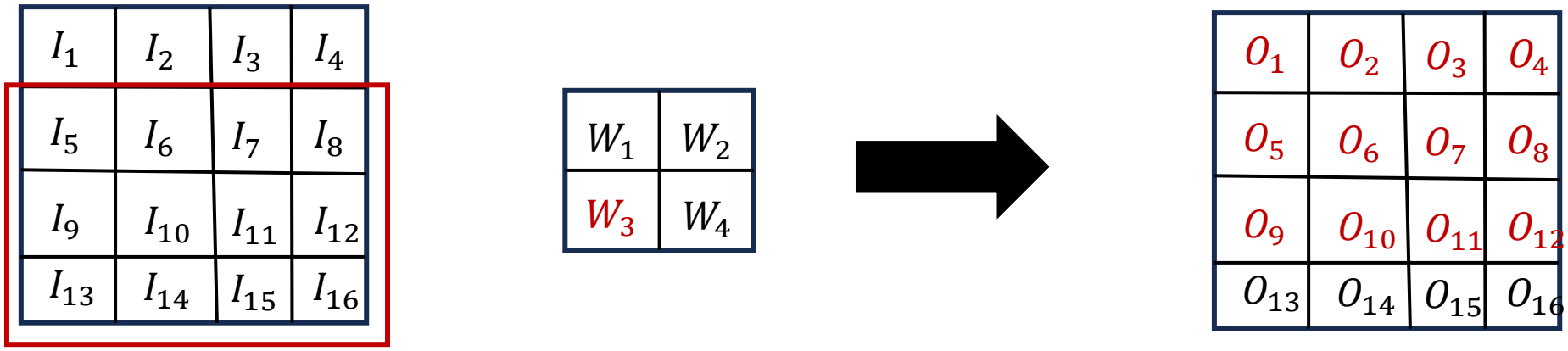
Scalar Matrix Multiplication



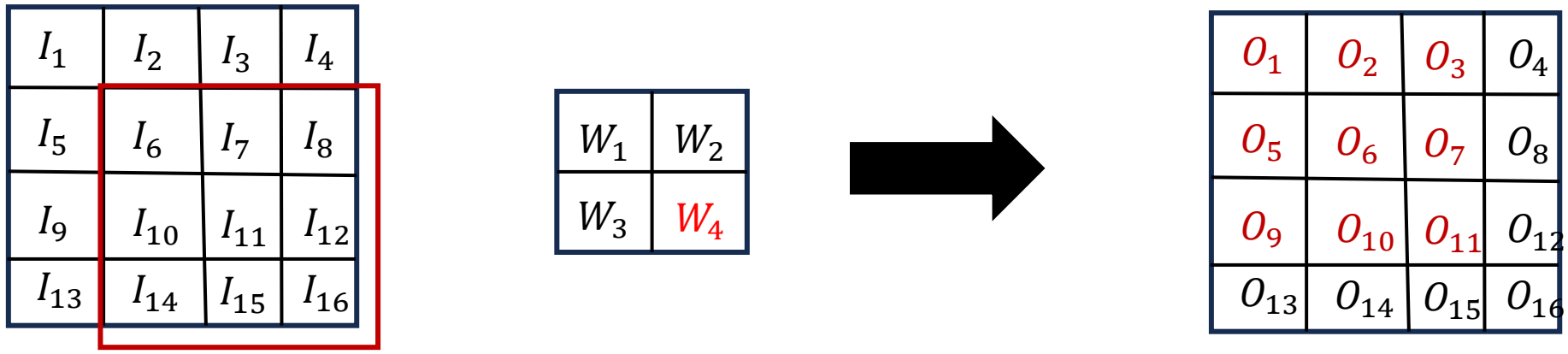
Scalar Matrix Multiplication



Scalar Matrix Multiplication

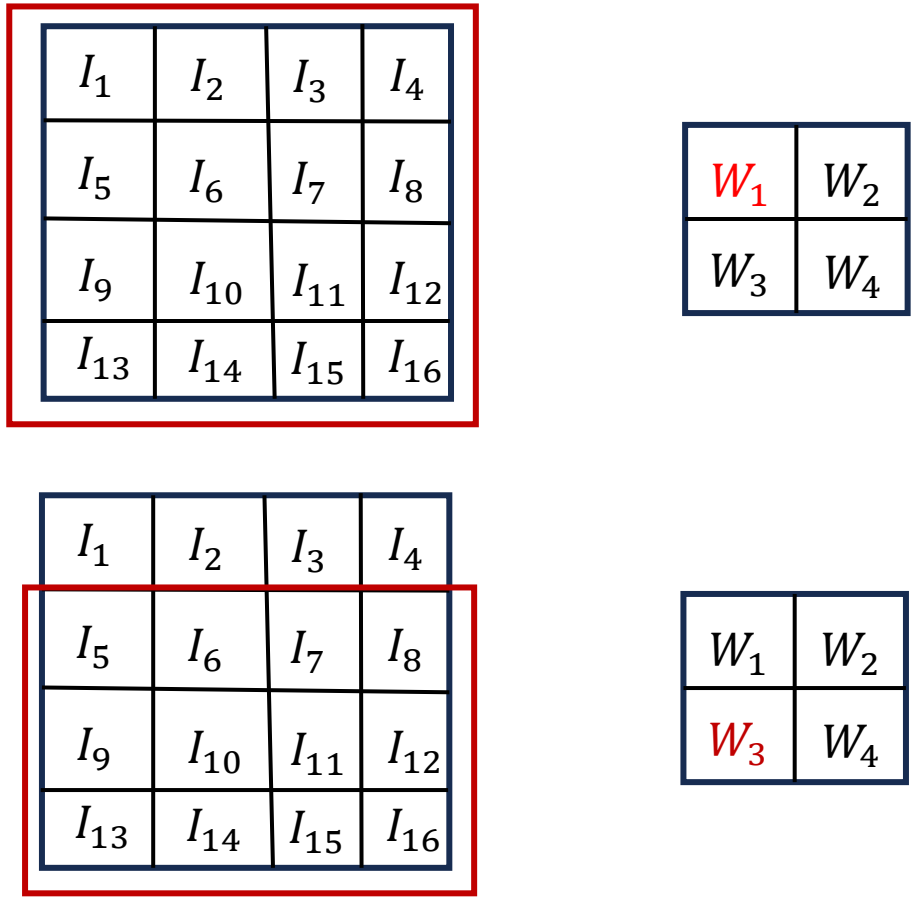


Scalar Matrix Multiplication



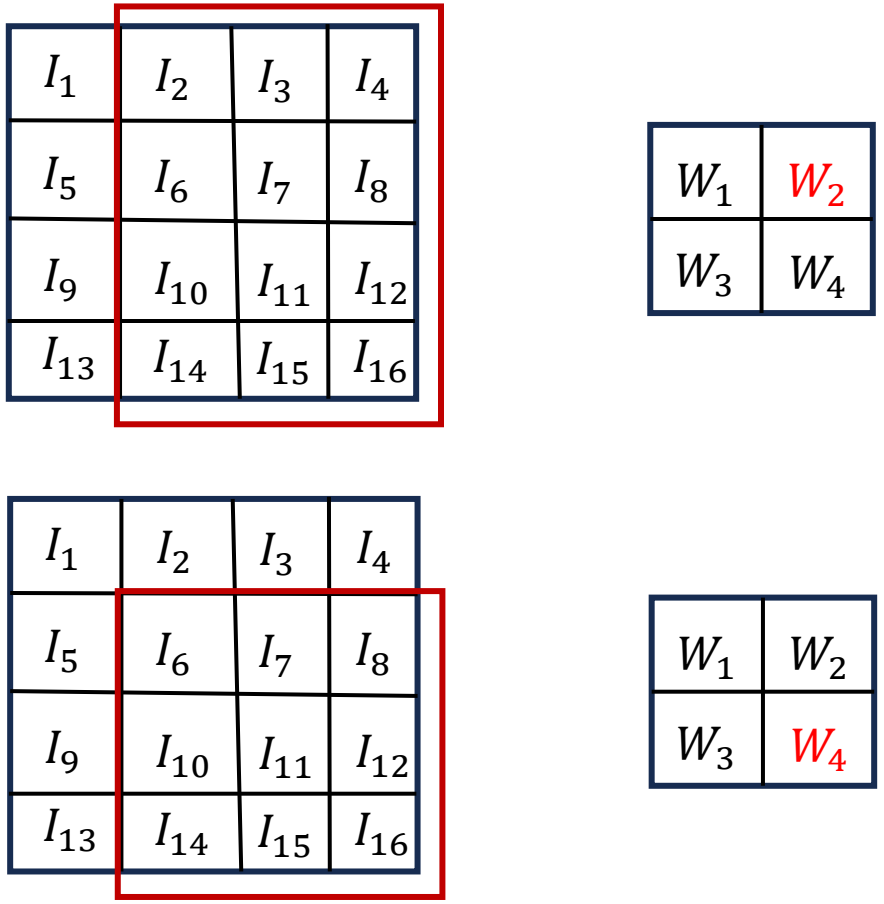
Scalar Matrix Multiplication

- Observe: weights along the same column operate on similar inputs



Scalar Matrix Multiplication

- Observe: weights along the same column operate on similar inputs



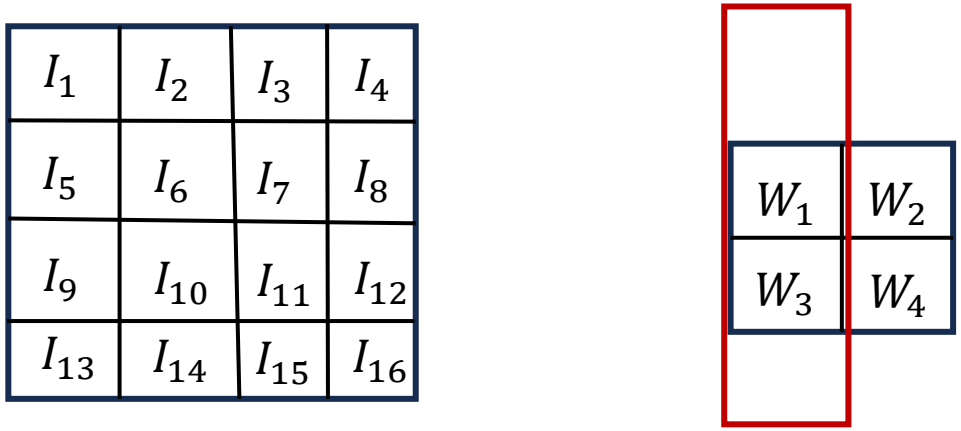
Scalar Matrix Multiplication

- Algorithm (single input/output channel):

- For each column j of kernel
  - Load a slice of input into cache: T_j
  - Load a slice of output into cache: O_j
  - For each row i of the kernel with weight $w = K[i][j]$
    - $O_j \leftarrow O_j + T_j[i:] * w$
  - Store O_j back

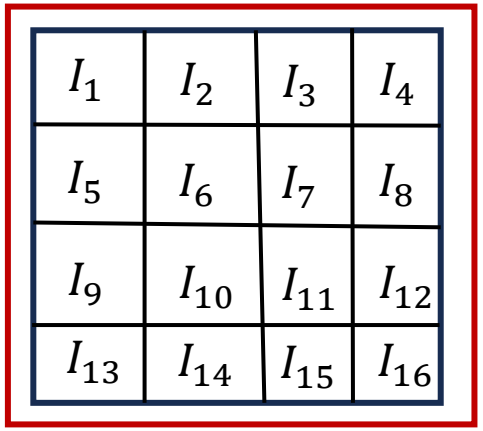
Scalar Matrix Multiplication

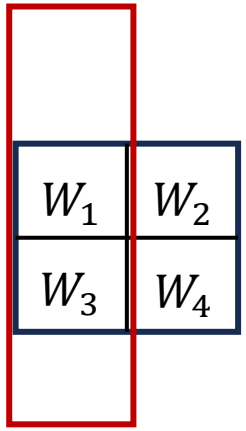
- For each column j



Scalar Matrix Multiplication

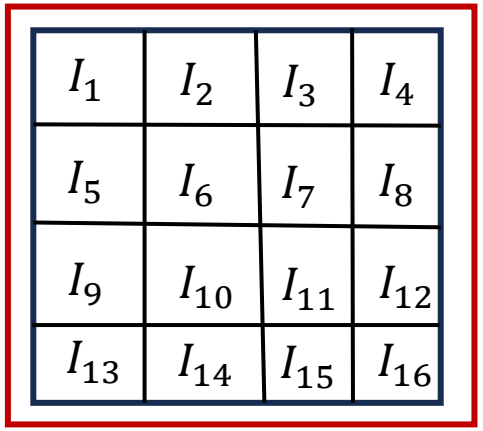
- Load a slice of input: T_j

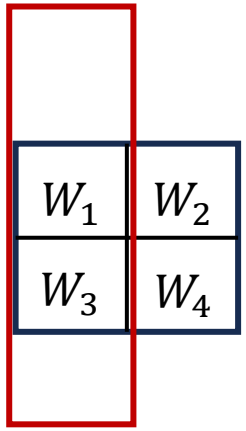


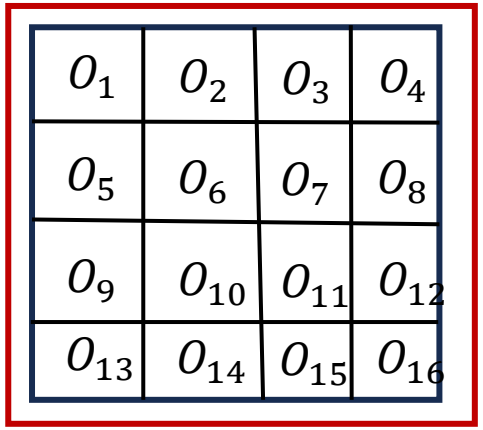


Scalar Matrix Multiplication

- Load a slice of output: O_j

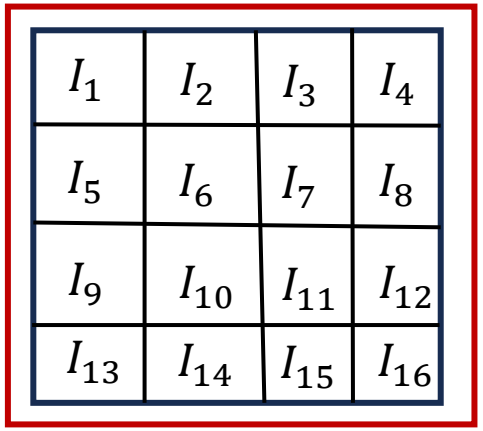


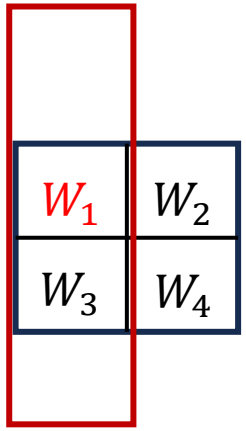


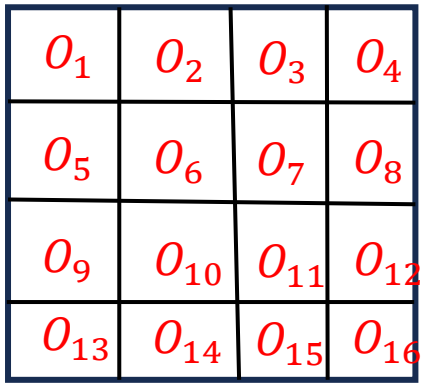


Scalar Matrix Multiplication

- For each row i of the kernel with weight $w = K[i][j]$
  - $O_j \leftarrow O_j + T_j[i:] * w$

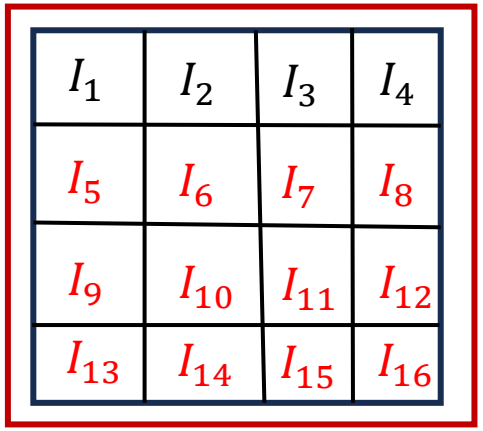


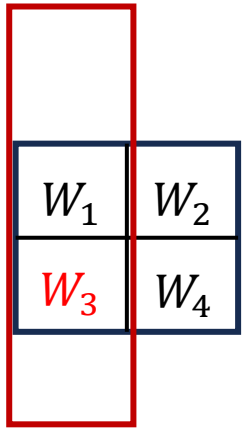


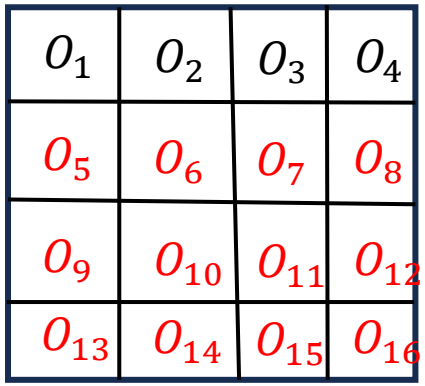


Scalar Matrix Multiplication

- For each row i of the kernel with weight $w = K[i][j]$
  - $O_j \leftarrow O_j + T_j[i:] * w$







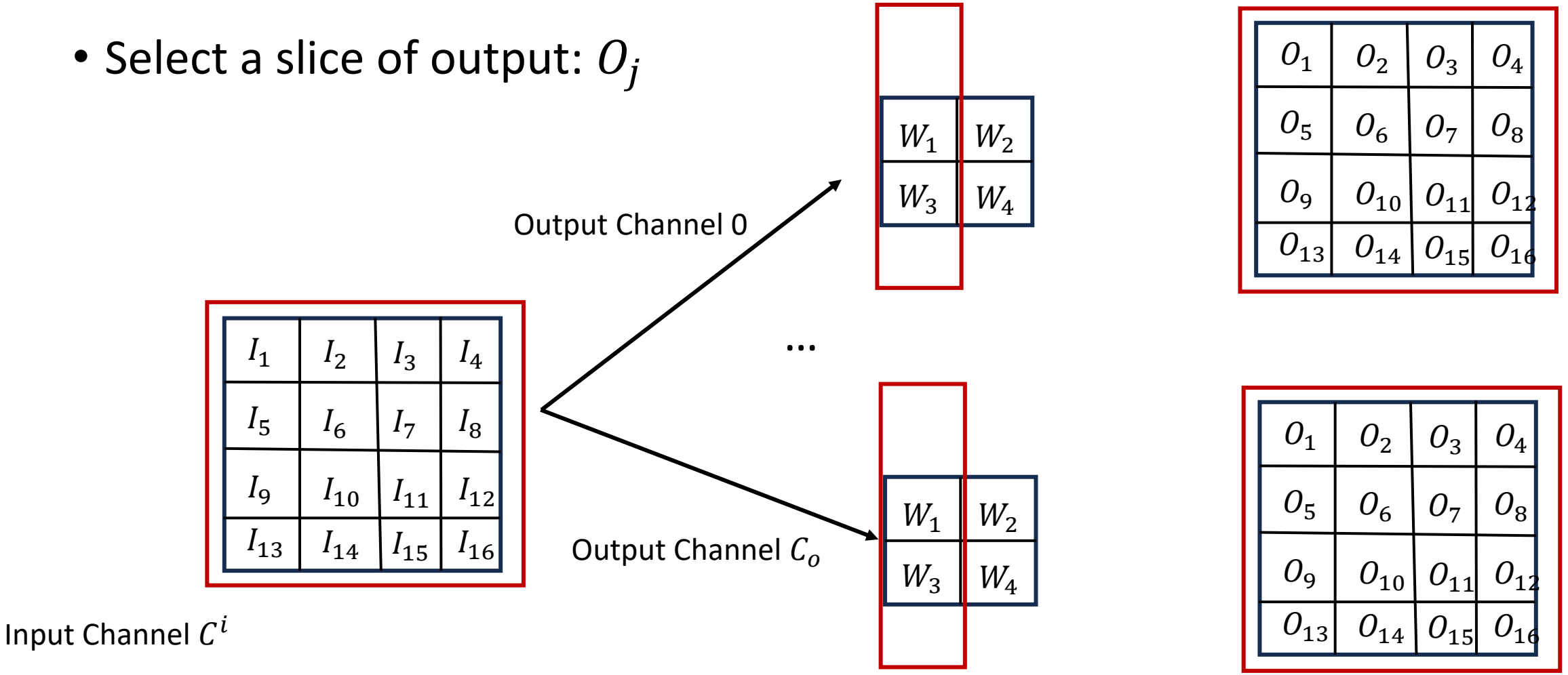
Scalar Matrix Multiplication

- Algorithm (c_{in} input, c_{out} output channels):

- For each input channel c^i
  - For each column j of kernel
    - Load a slice of input into cache: $T_j^{c^i}$
    - For each output channel c^0
      - Load a slice of output into cache: $O_j^{c^o}$
      - For each row i of the kernel with weight $w = K[i][j][c^i][c^o]$
        - $O_j^{c^o} \leftarrow O_j^{c^o} + T_j^{c^i}[i:] * w$
      - Store $O_j^{c^o}$ back

Scalar Matrix Multiplication

- Select a slice of output: O_j



Scalar Matrix Multiplication

- Claim of the paper
- Memory requirement = $H \times W$ vs $C_{in} \times K^2 \times H \times W$
- Technically correct, but may not be very useful
  - We care more about data transfers and computation time

Scalar Matrix Multiplication

- Can we parallelize this algorithm?

- Ungraded HW Assignment: Apply one of the data parallelism techniques to come up with a parallel algorithm (see if you can design an algorithm that has no write conflicts)

- For each input channel c^i
  - For each column j of kernel
    - Load a slice of input into cache: $T_j^{c^i}$
    - For each output channel c^0
      - Load a slice of output into cache: $O_j^{c^o}$
      - For each row i of the kernel with weight $w = K[i][j]\left[c^i\right][c^o]$
        - $O_j^{c^o} \leftarrow O_j^{c^o} + T_j^{c^i}[i:] * w$

Im2Col Algorithm

- Input Matrix -$(k \times k \times C_{in}) \times (H \times W)$
  - $k \times k$ increase in size -> leads to increase in memory transfers
- Filter Matrix - $(\mathrm{C}_{out}) \times (k \times k \times C_{in})$
  - No increase in size
- Output Matrix - $C_{out} \times (H \times W)$
  - No increase in size

Kn2Row Algorithm

- Input Matrix -$(C_{in}) \times (H \times W)$
- Filter Matrix - $\left(\mathrm{C}_{out} \times k^2\right) \times (C_{in})$
  - No increase in size
- Output Matrix -$(H \times W) \times (C_{out} \times k^2)$
  - $k \times k$ increase in size -> leads to increase in memory transfers

Scalar Matrix Multiplication

- Input Matrix -$(C_{in}) \times (H \times W)$
  - No increase in size
- Filter Matrix - $(\mathrm{C}_{out}) \times (k \times k \times C_{in})$
  - No increase in size
- Output Matrix - $C_{out} \times (H \times W)$
  - No increase in size
- Performs Matrix Scaling and Addition Operations

Next Class

- 9/30 Lecture 11
  - Accelerating Convolutional Neural Networks: Exploring Sparsity in Convolutional Neural Networks

CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

Thank You

- Questions?

- Email: sanmukh.kuppannagari@case.edu