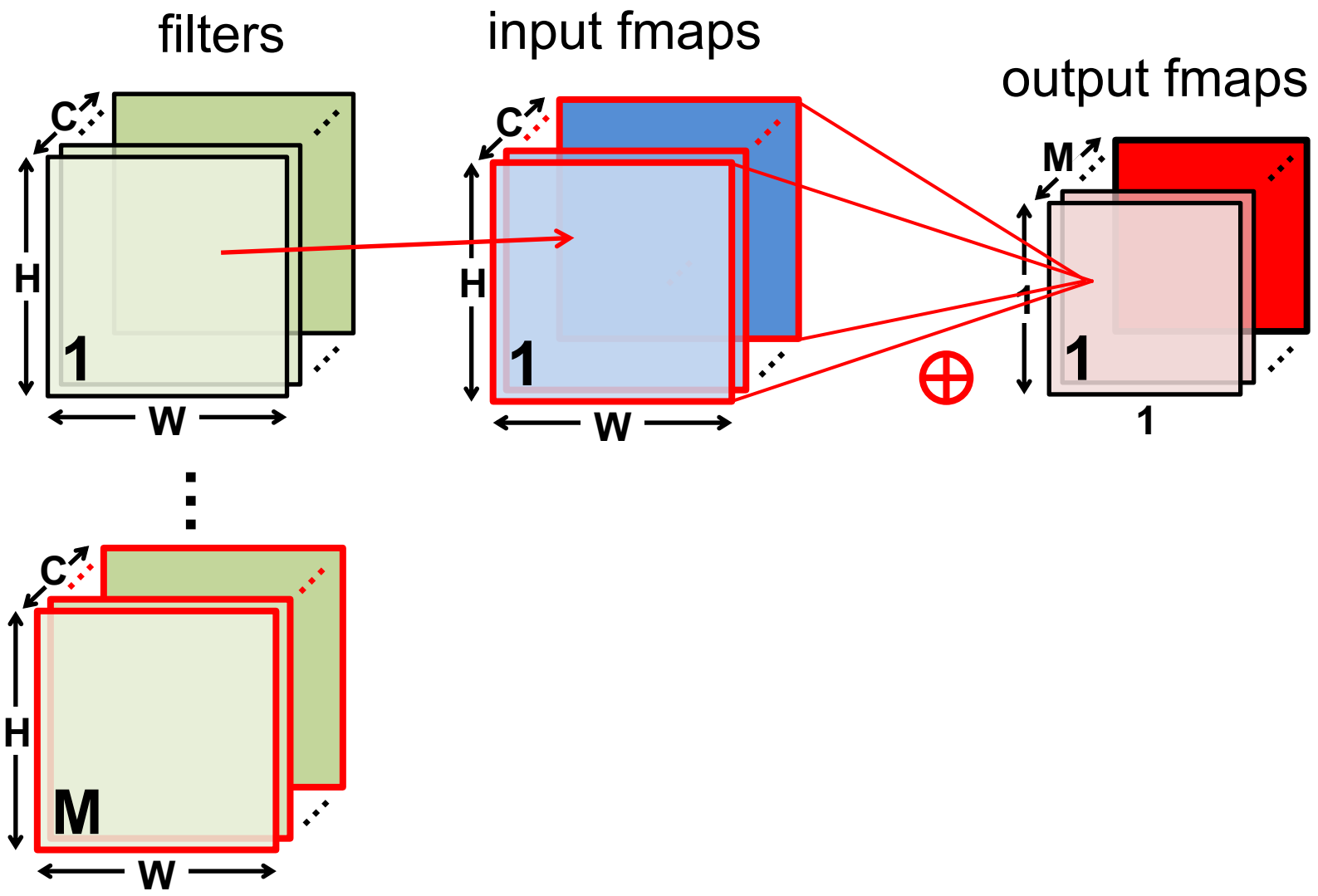
Kernel Computation

ISCA Tutorial (2019)

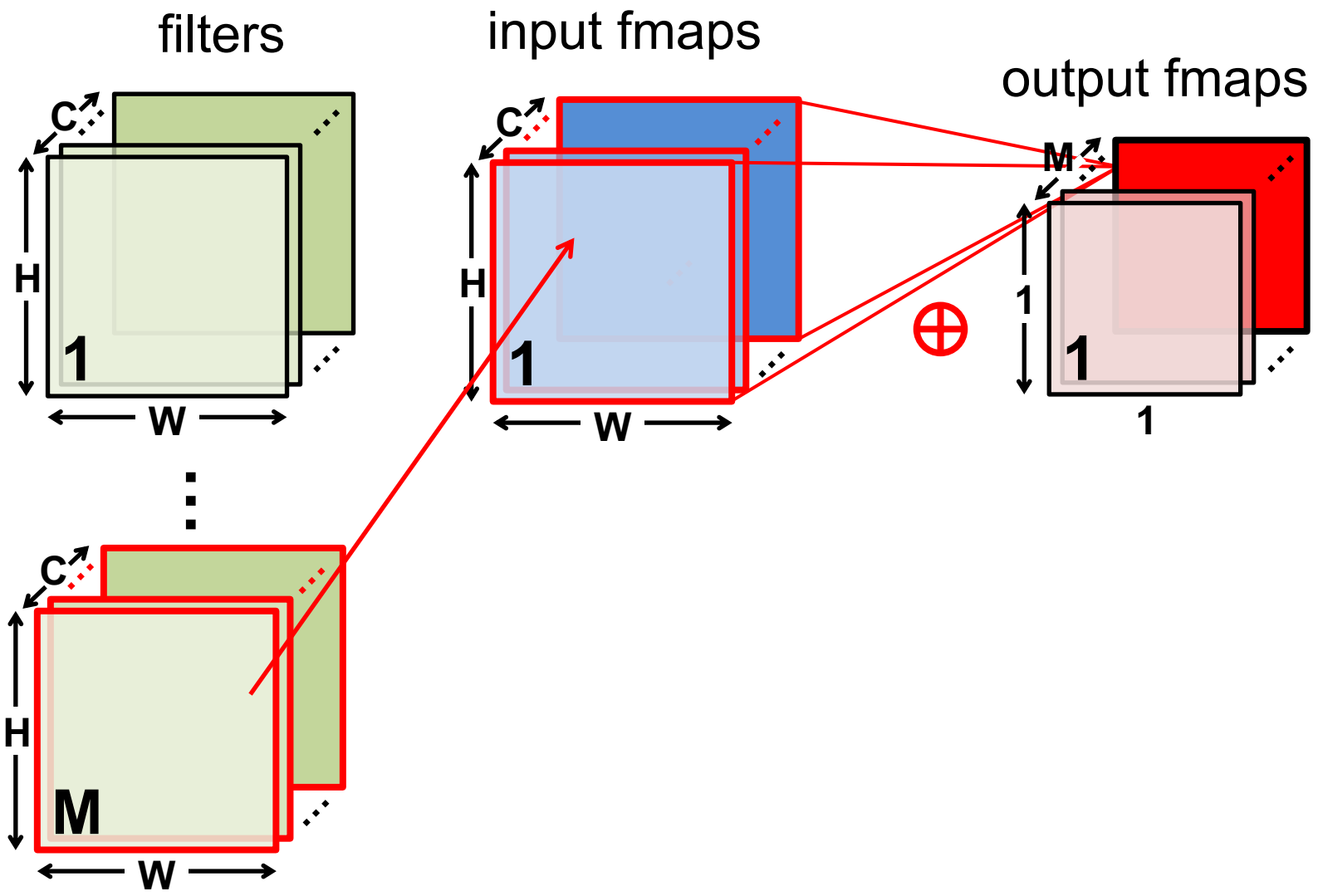
Website: http://eyeriss.mit.edu/tutorial.html

Joel Emer, Vivienne Sze, Yu-Hsin Chen

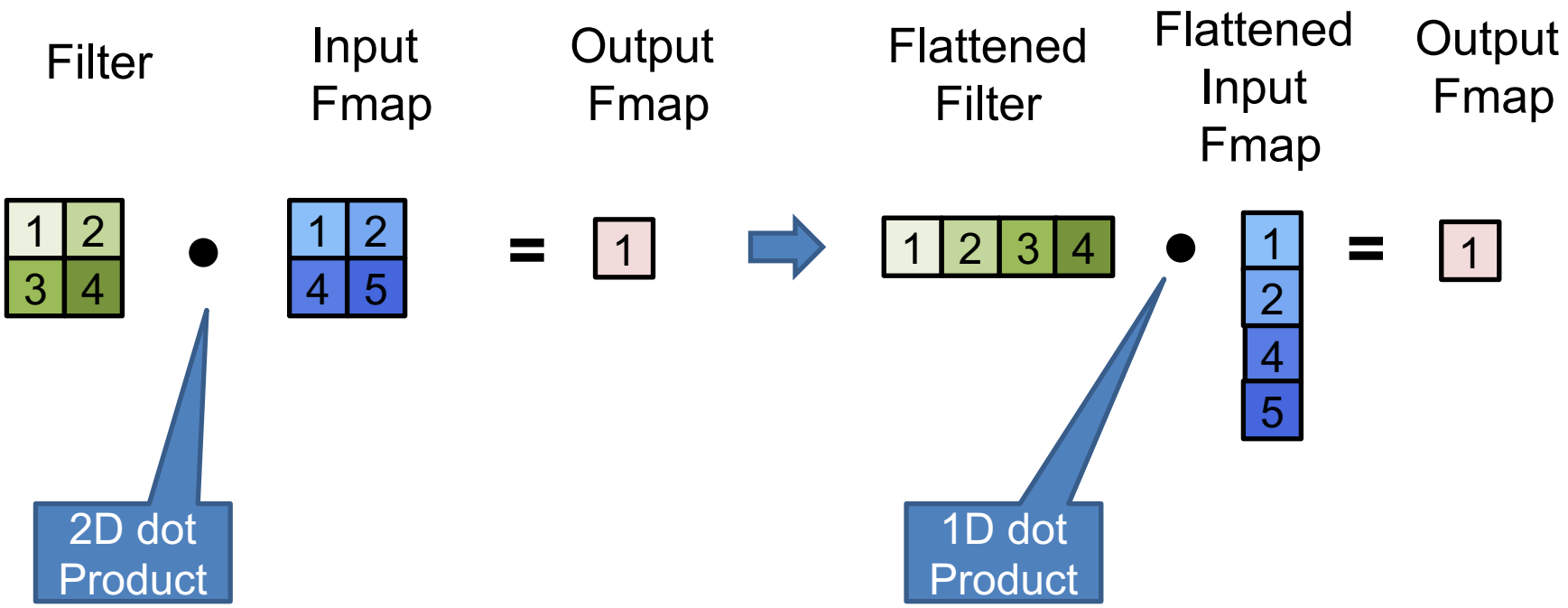
Fully-Connected (FC) Layer



Fully-Connected (FC) Layer

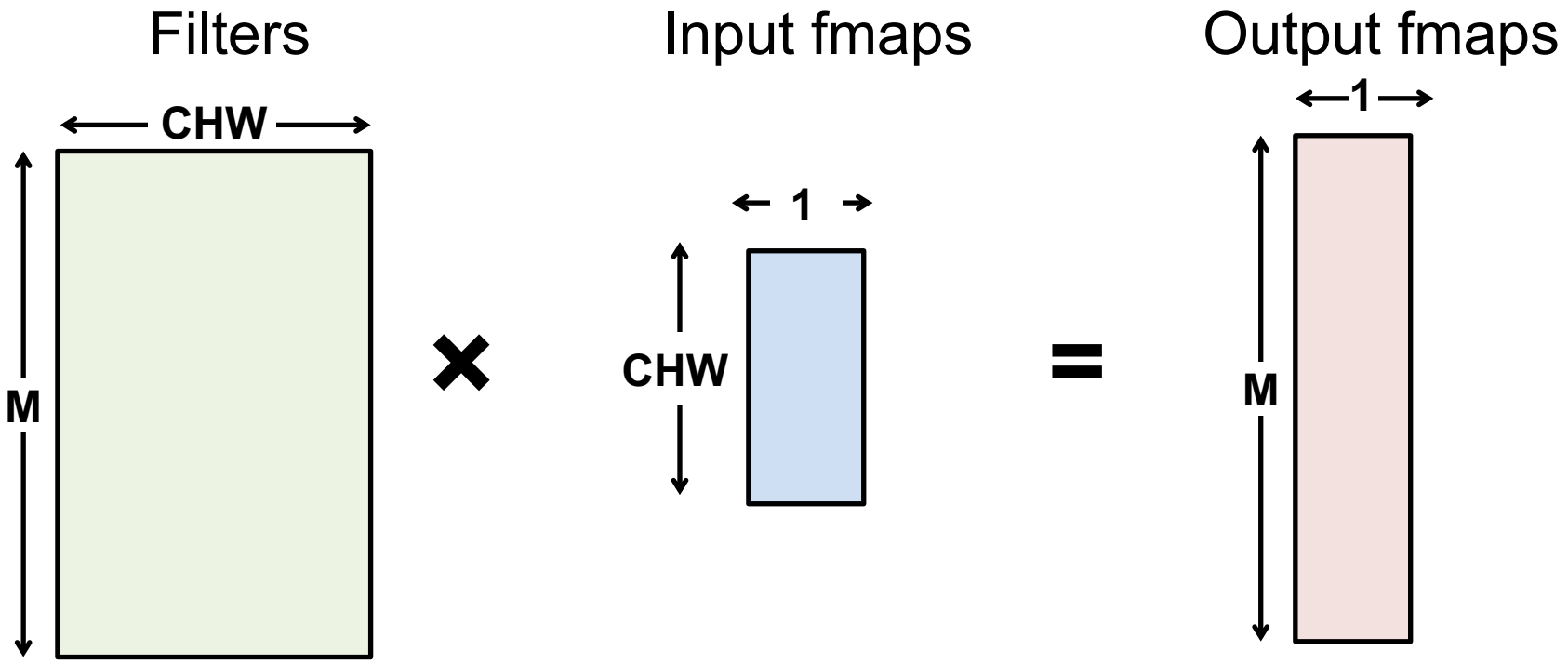


Flattened 2D Dot Product



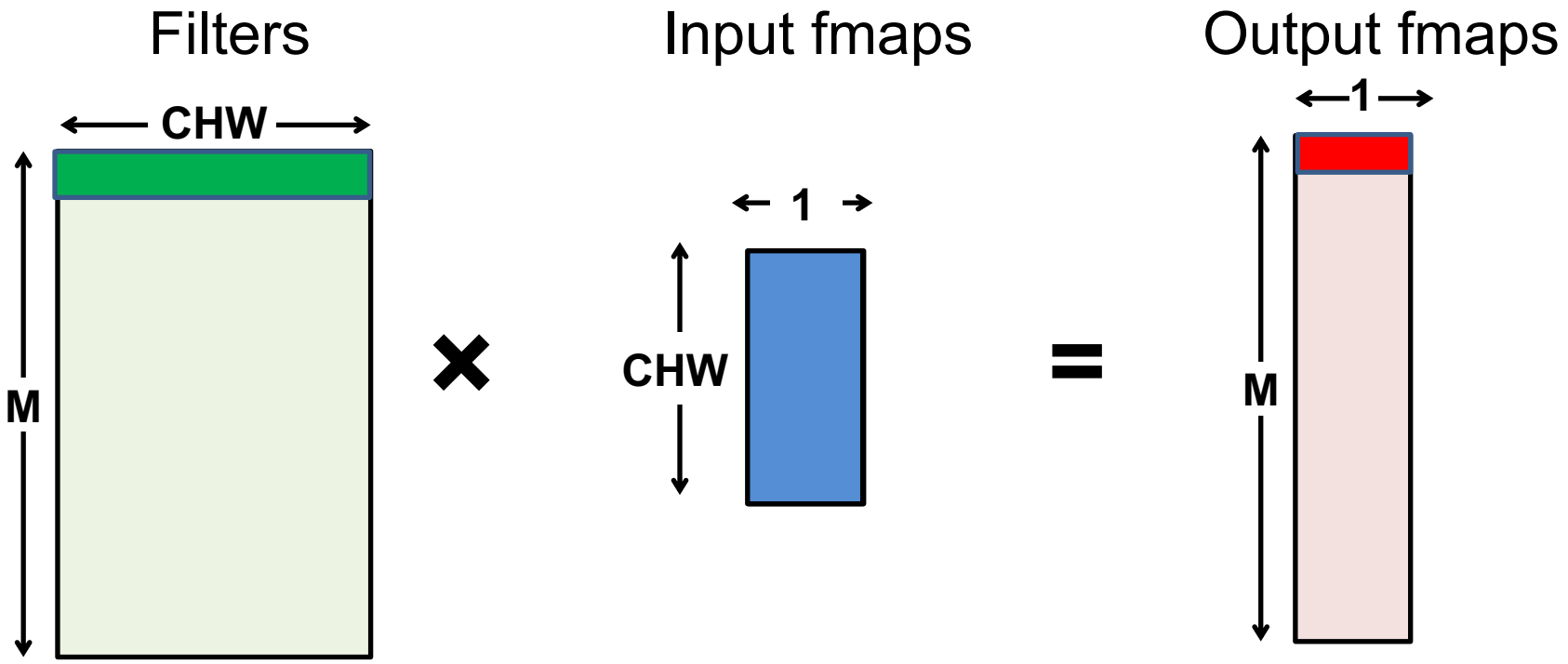
Fully-Connected (FC) Layer

- Matrix–Vector Multiply:
  - Multiply all inputs in all channels by a weight and sum



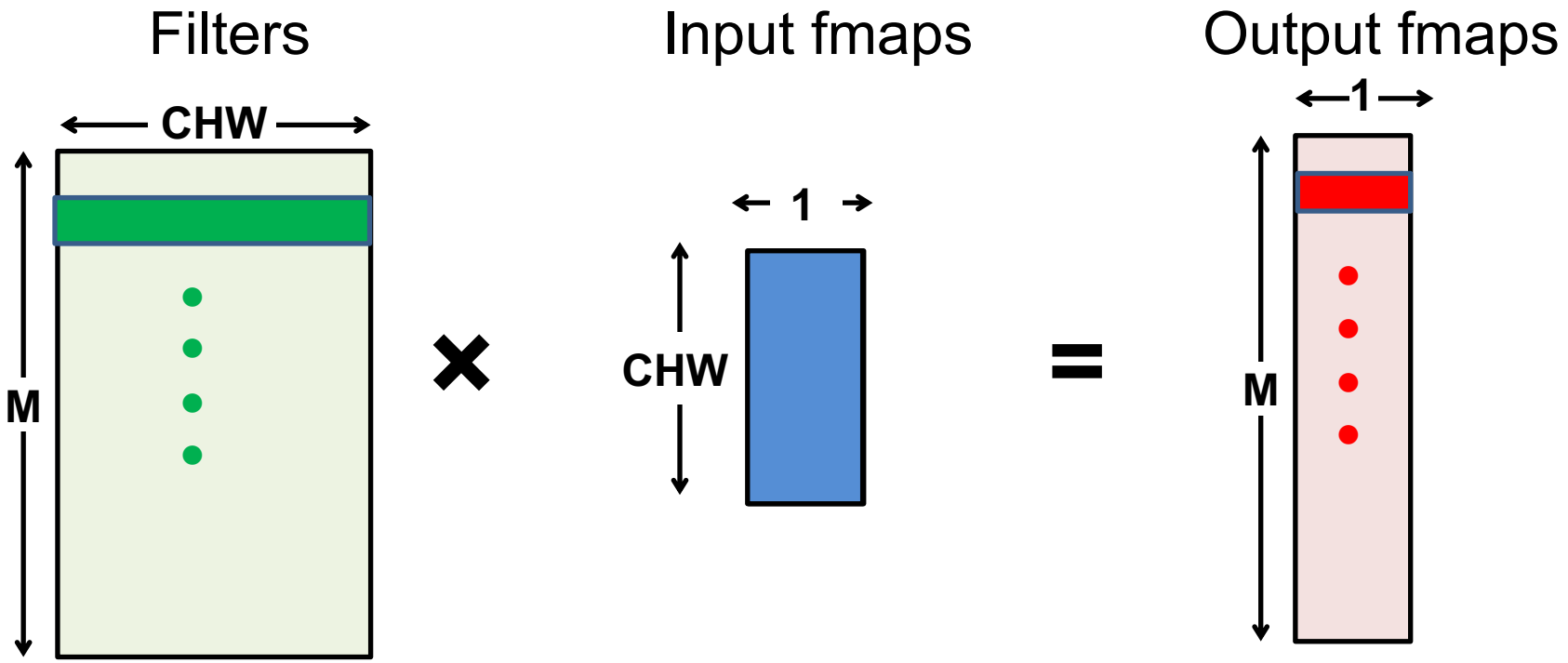
Fully-Connected (FC) Layer

- Matrix–Vector Multiply:
  - Multiply all inputs in all channels by a weight and sum

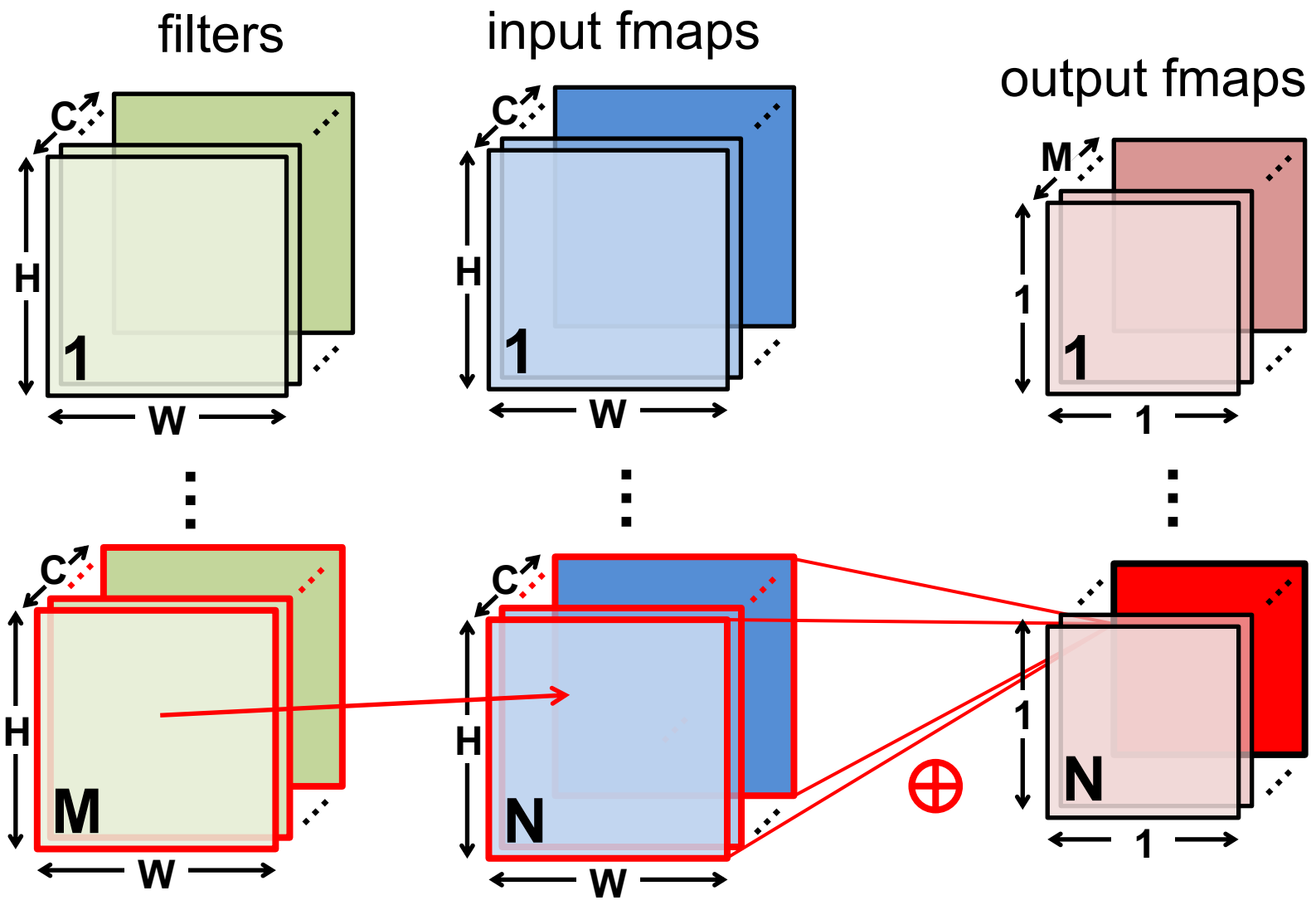


Fully-Connected (FC) Layer

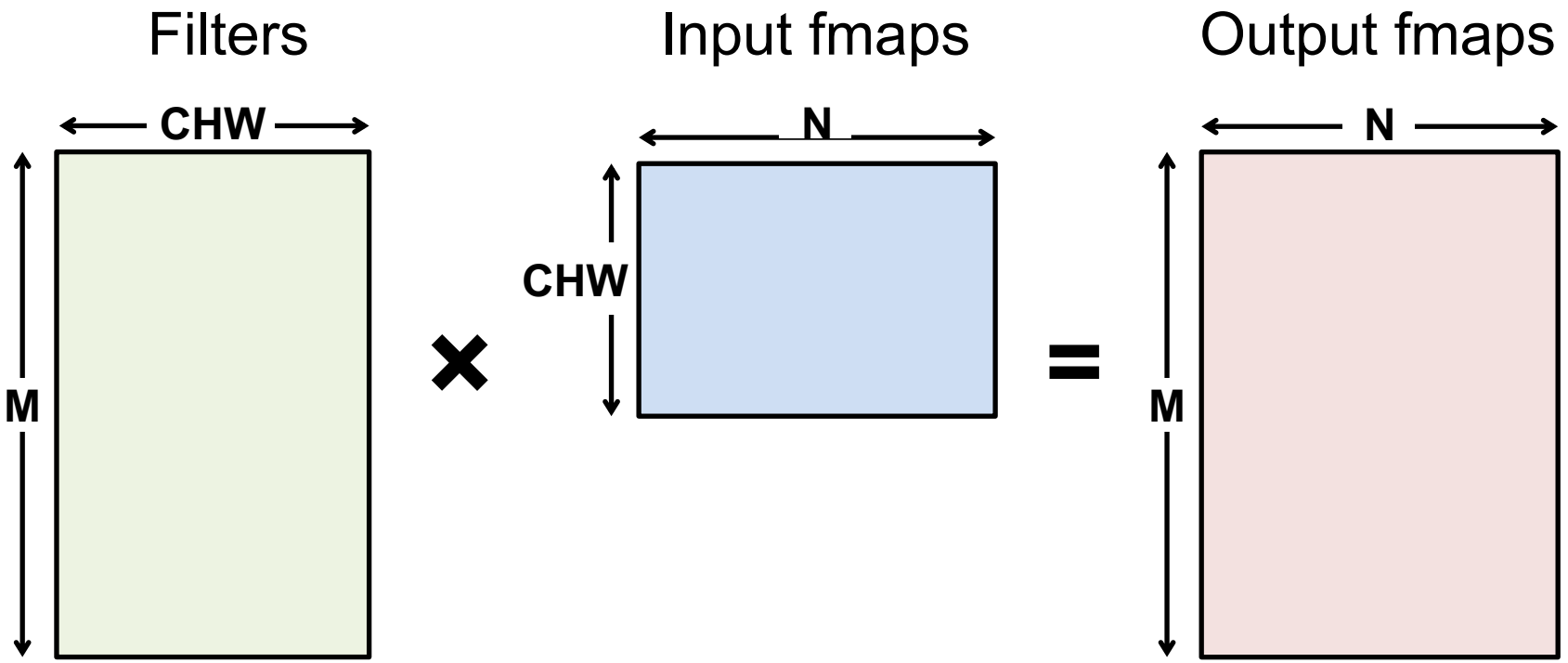
- Matrix–Vector Multiply:
  - Multiply all inputs in all channels by a weight and sum



Fully-Connected (FC) Layer

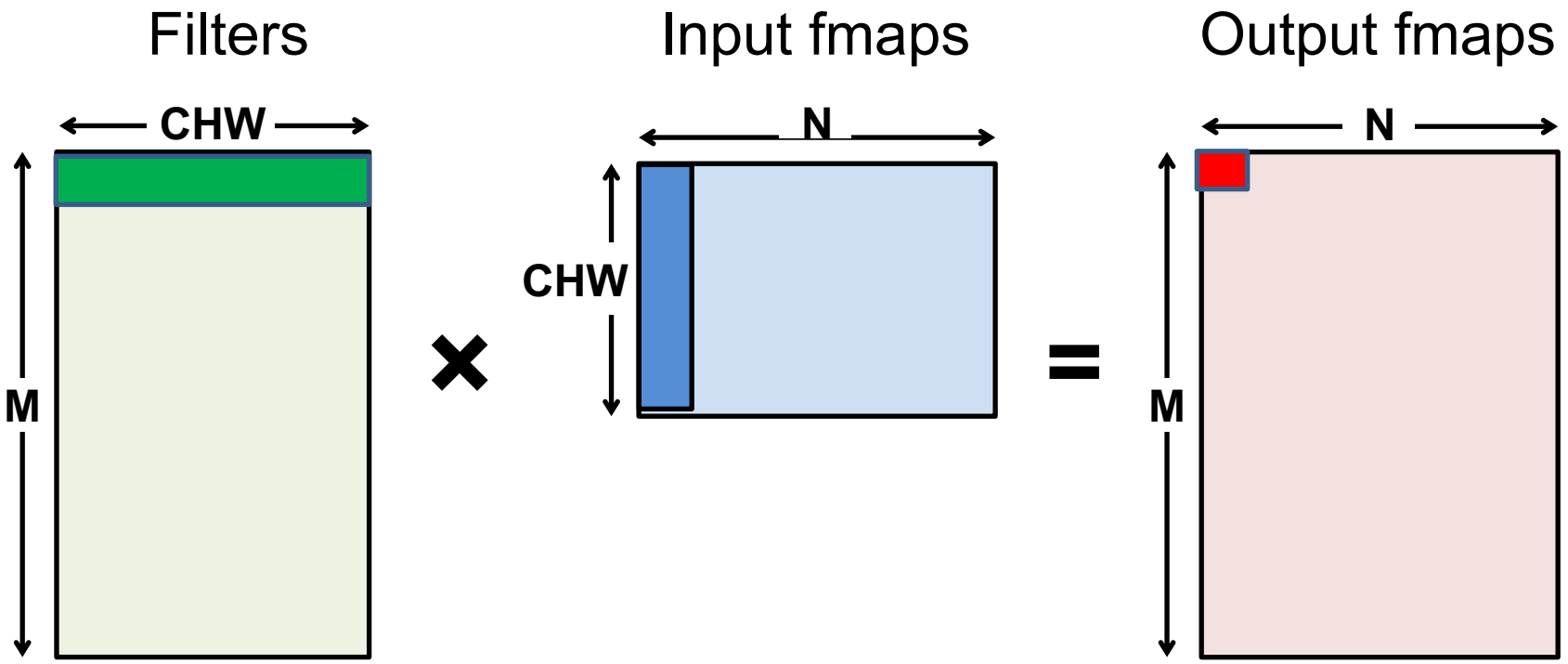


Flattened Fully-Connected Layer



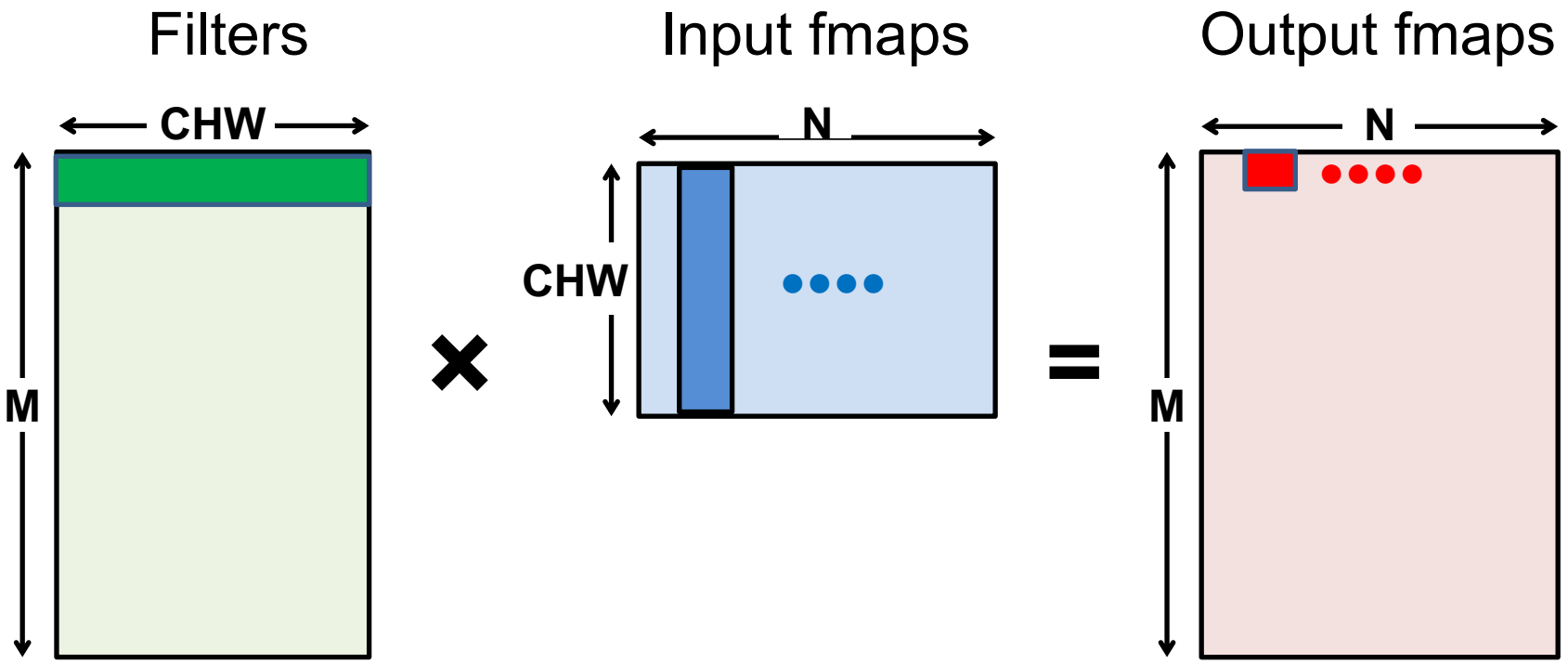
- After flattening, having a batch size of N turns the matrix-vector operation into a matrix-matrix multiply

Fully-Connected (FC) Layer



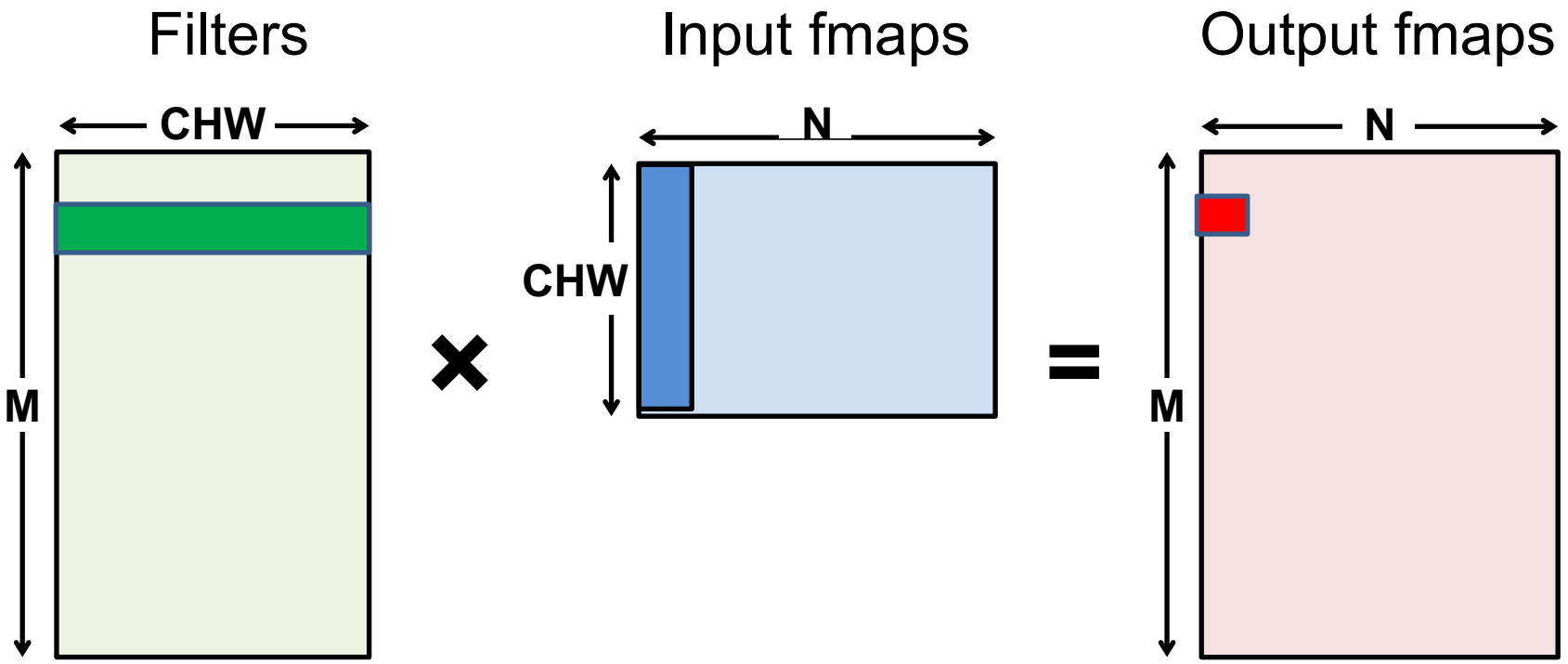
- After flattening, having a batch size of N turns the matrix-vector operation into a matrix-matrix multiply

Fully-Connected (FC) Layer



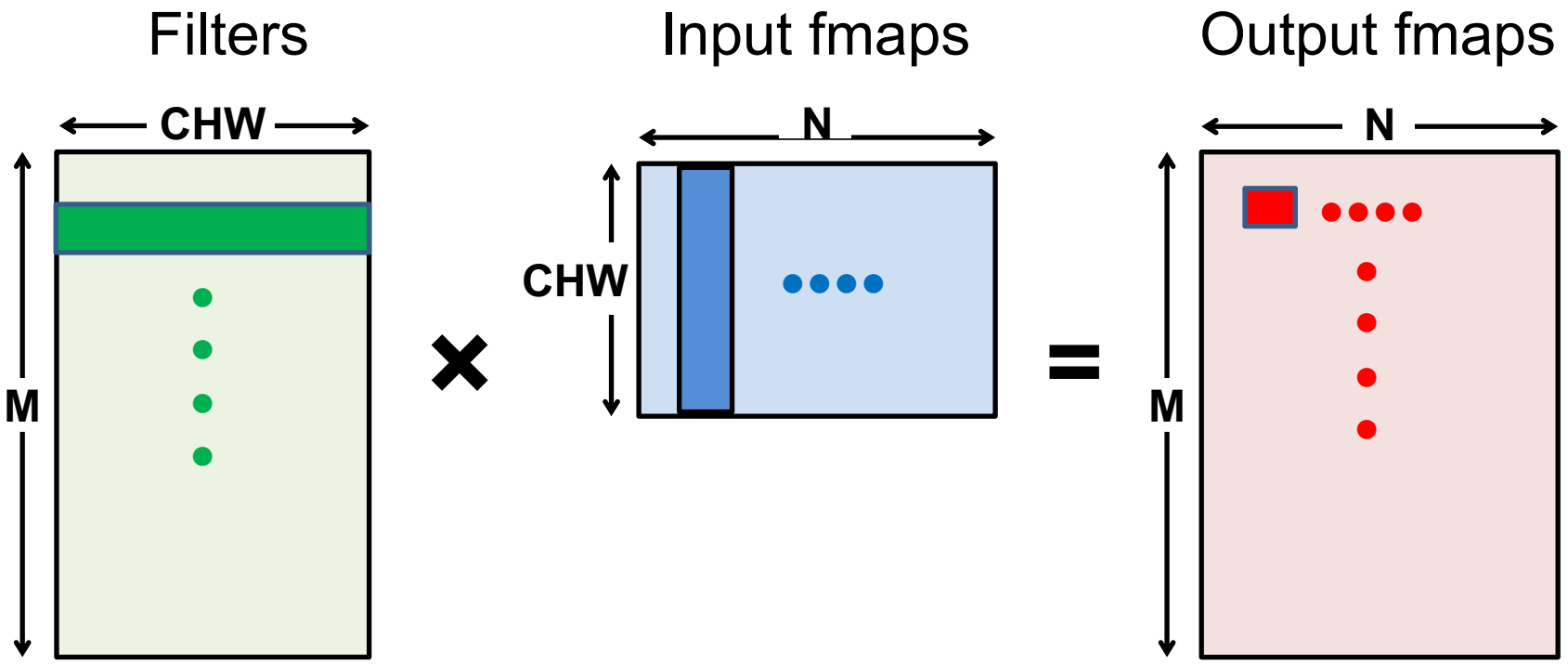
- After flattening, having a batch size of N turns the matrix-vector operation into a matrix-matrix multiply

Fully-Connected (FC) Layer



- After flattening, having a batch size of N turns the matrix-vector operation into a matrix-matrix multiply

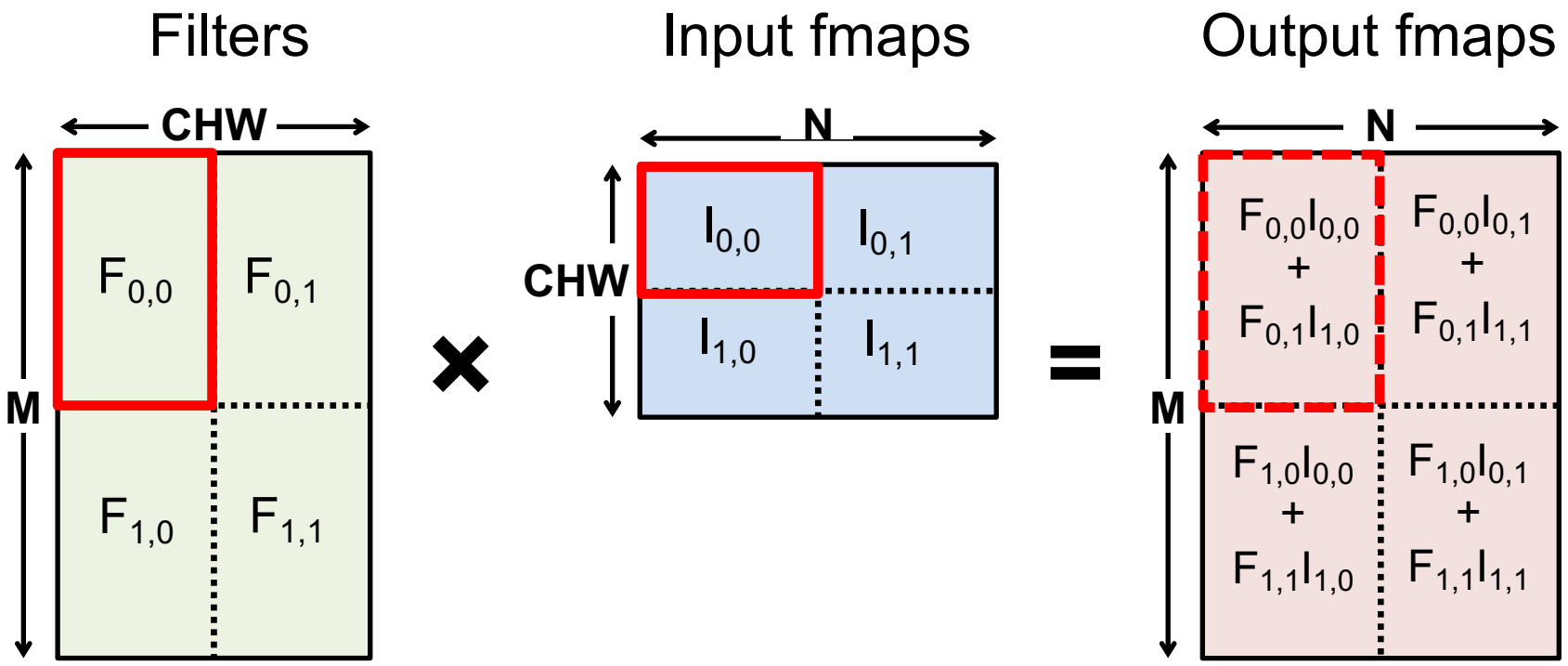
Fully-Connected (FC) Layer



- After flattening, having a batch size of N turns the matrix-vector operation into a matrix-matrix multiply

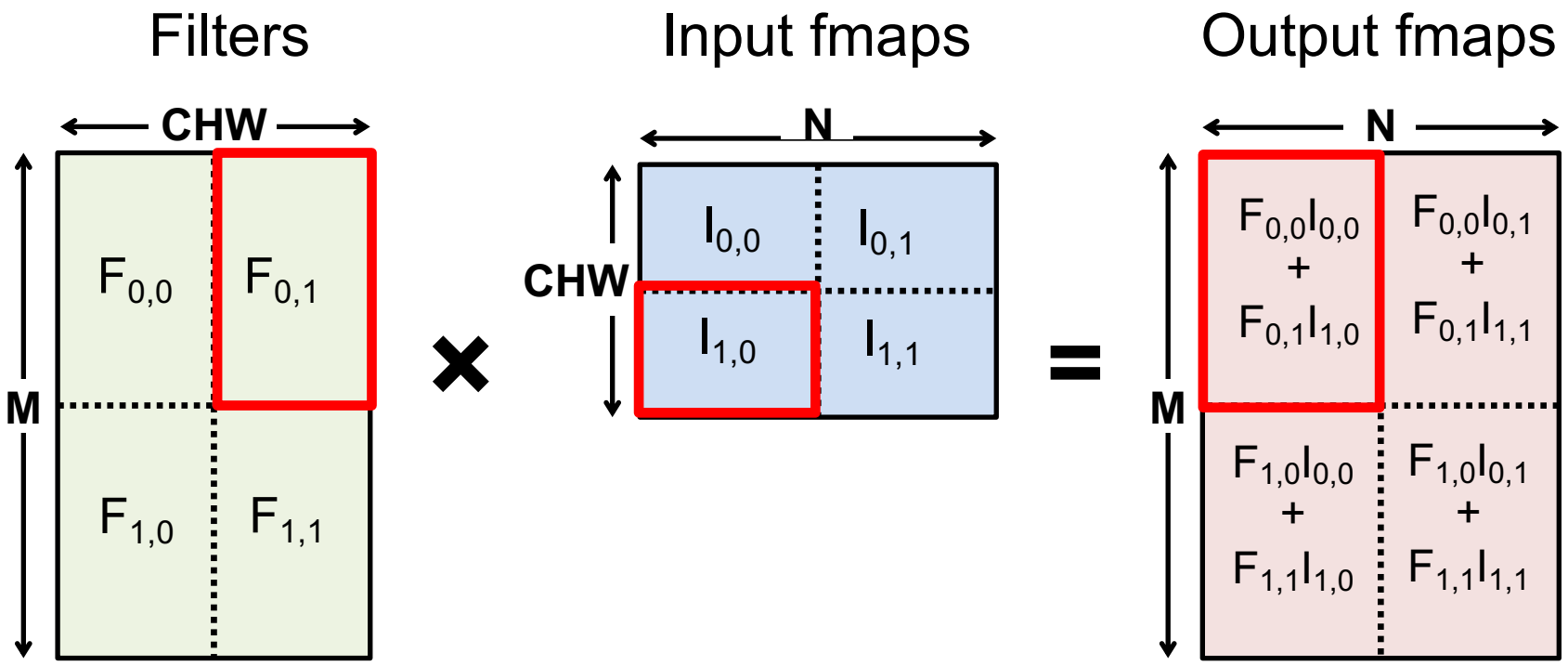
How much temporal locality for naïve implementation? None

Tiled Fully-Connected (FC) Layer



Matrix multiply tiled to fit in cache
and computation ordered to maximize reuse of data in cache

Tiled Fully-Connected (FC) Layer

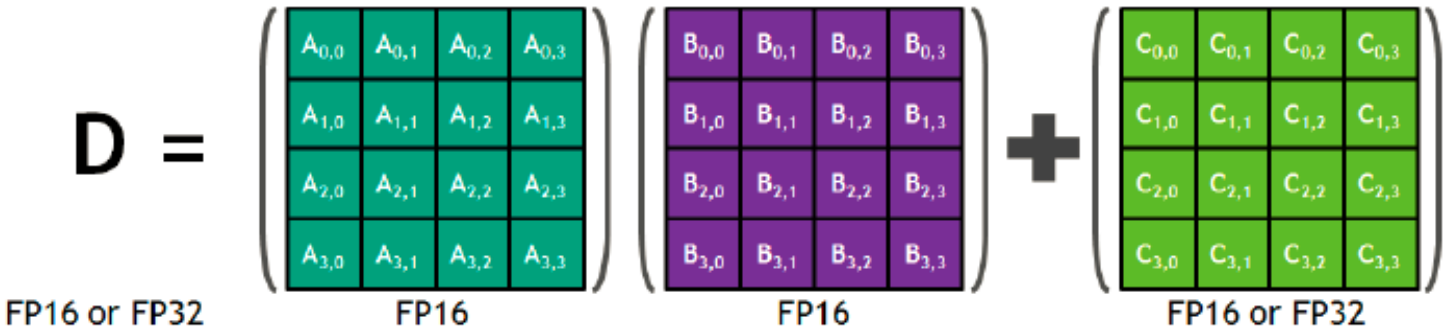


Matrix multiply tiled to fit in cache
and computation ordered to maximize reuse of data in cache

Fully-Connected (FC) Layer

- Implementation: **Matrix Multiplication (GEMM)**
  - **CPU:** OpenBLAS, Intel MKL, etc
  - **GPU:** cuBLAS, cuDNN, etc
- Library will note shape of the matrix multiply and select implementation optimized for that shape.
- Optimization usually involves proper tiling to storage hierarchy

GV100 – “Tensor Core”



New opcodes – Matrix Multiply Accumulate (HMMA)

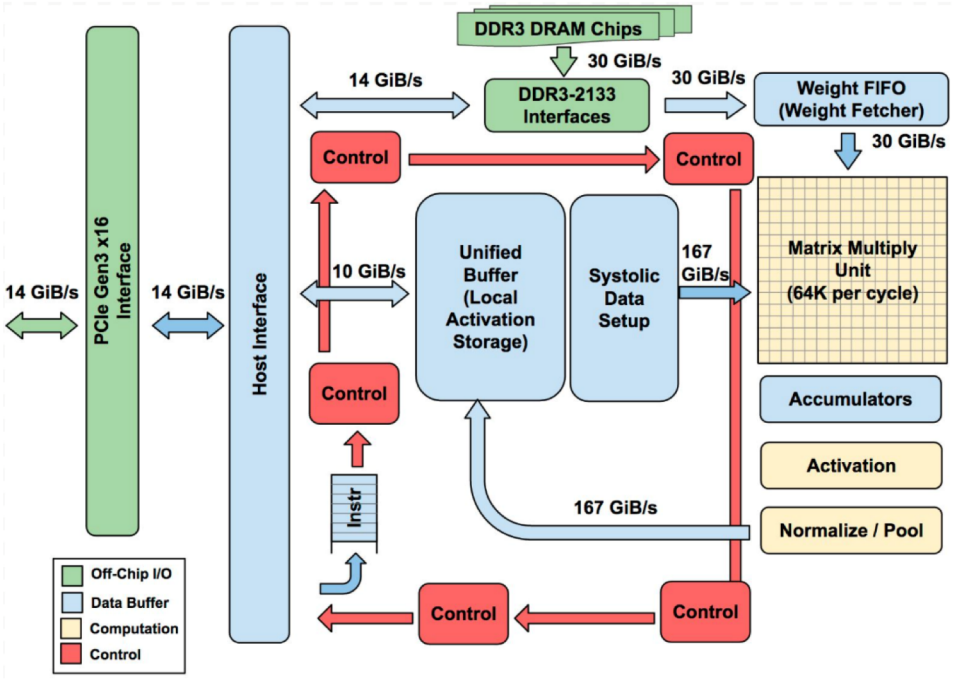
FP16 operands?	Inputs 48 / Outputs 16
Multiplies?	64
Adds?	64

Tensor Core….

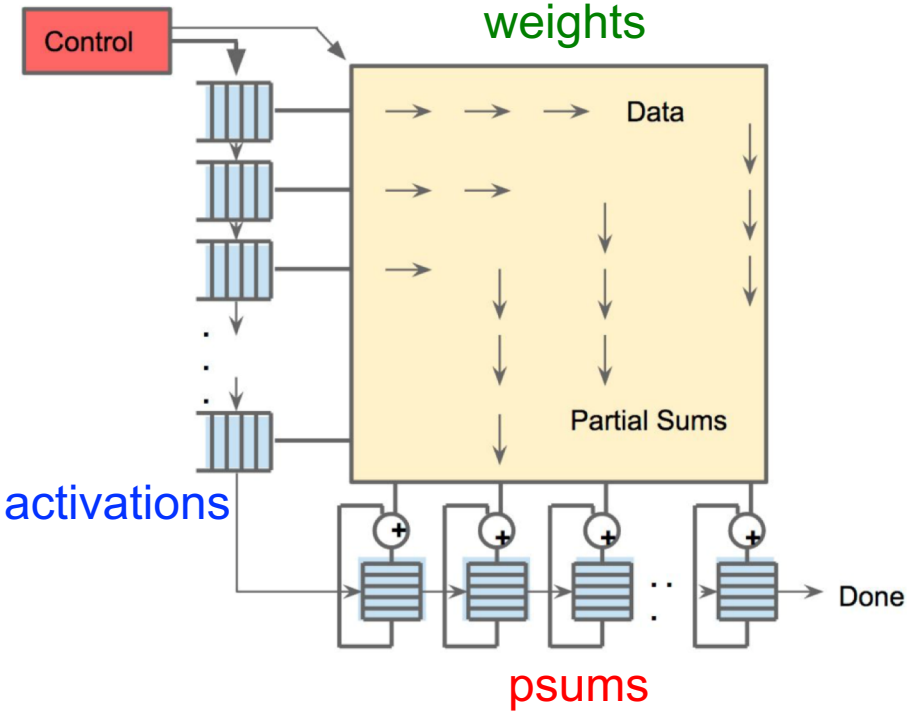
- 120 TFLOPS (FP16)
- 400 GFLOPS/W (FP16)

TPU

Top-Level Architecture

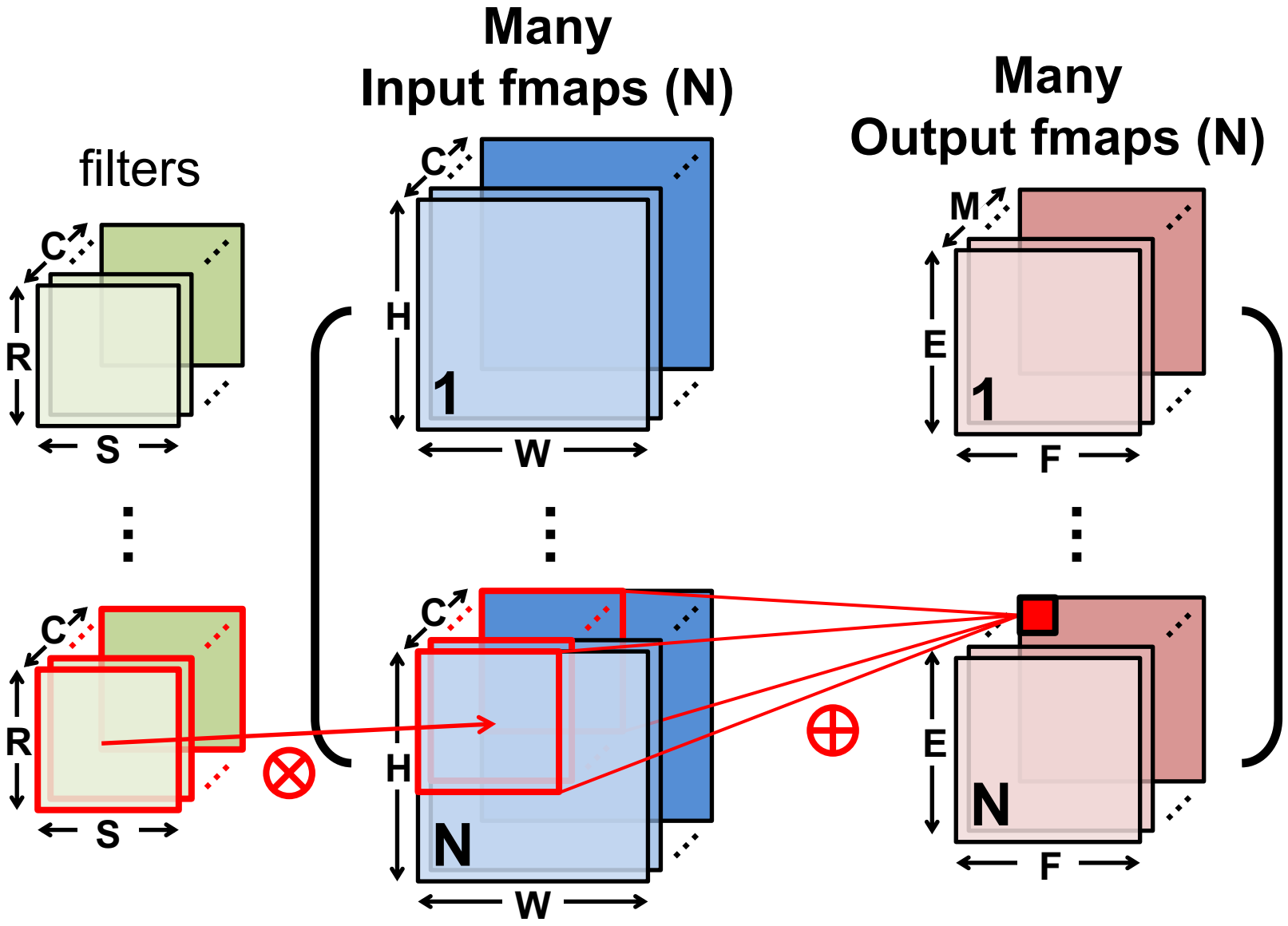


Matrix Multiply Unit



[Jouppi et al., ISCA 2017]

Convolution (CONV) Layer



CONV Layer Implementation

Naïve 7-layer for-loop implementation:

```
for (n=0; n<N; n++) {
    for (m=0; m<M; m++) {
        for (x=0; x<F; x++) {
            for (y=0; y<E; y++) {
```

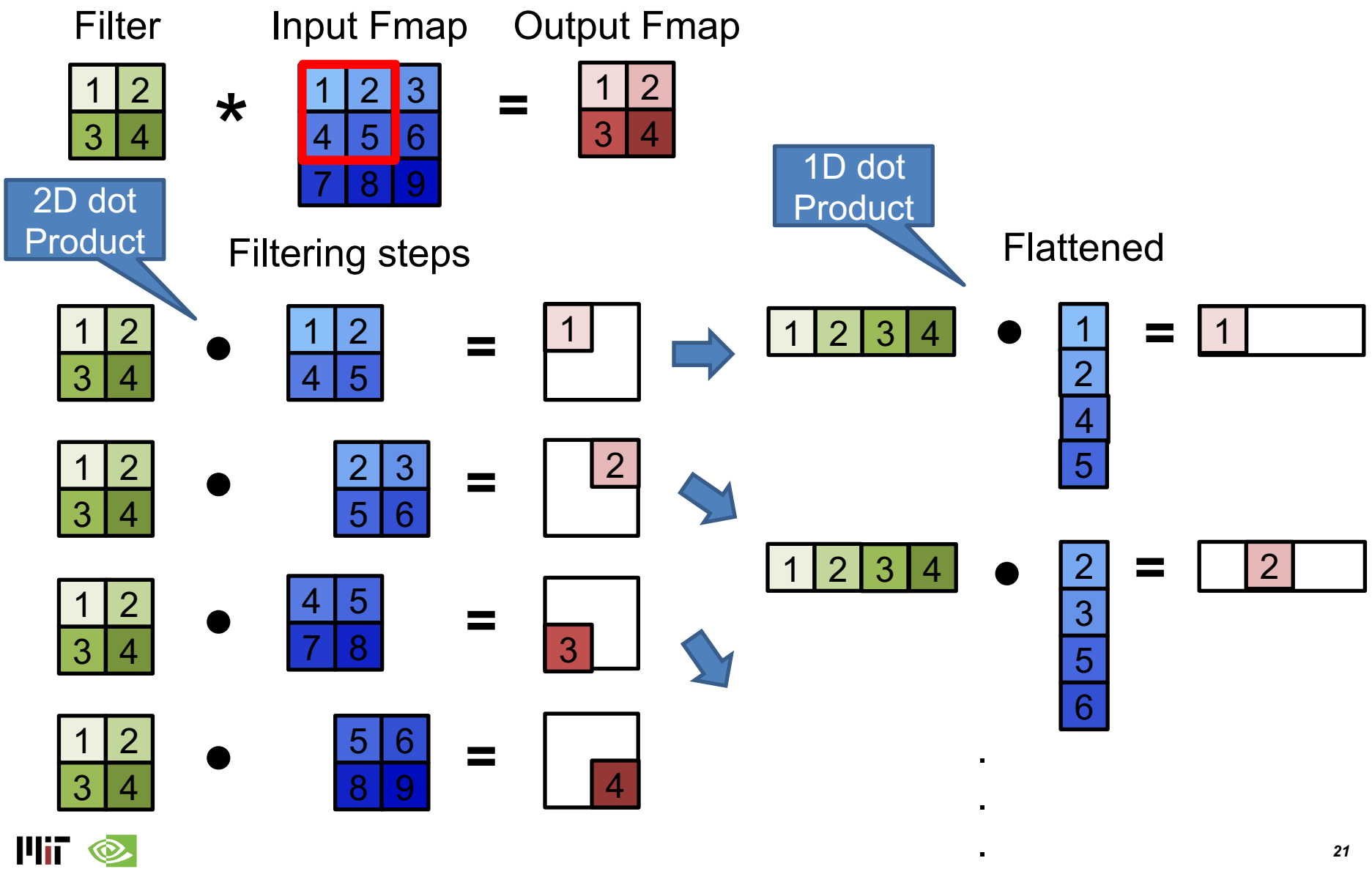
} for each output fmap value

convolve a window and apply activation

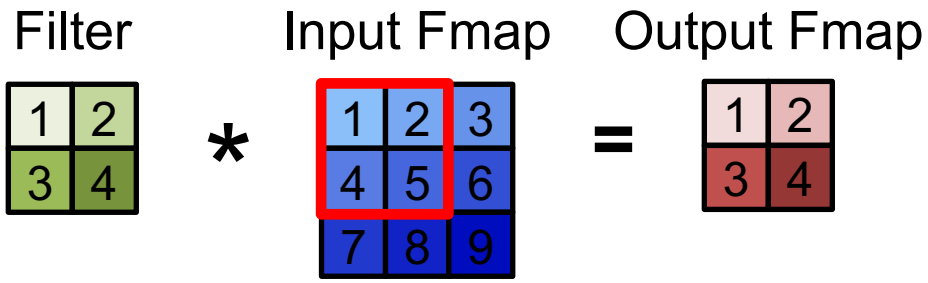
```
                O[n][m][x][y] = B[m];
                for (i=0; i<R; i++) {
                    for (j=0; j<S; j++) {
                        for (k=0; k<C; k++) {
                            O[n][m][x][y] += I[n][k][Ux+i][Uy+j] × W[m][k][i][j];
                        }
                    }
                }

                O[n][m][x][y] = Activation(O[n][m][x][y]);
            }
        }
    }
}
```

Convolution (CONV) Layer

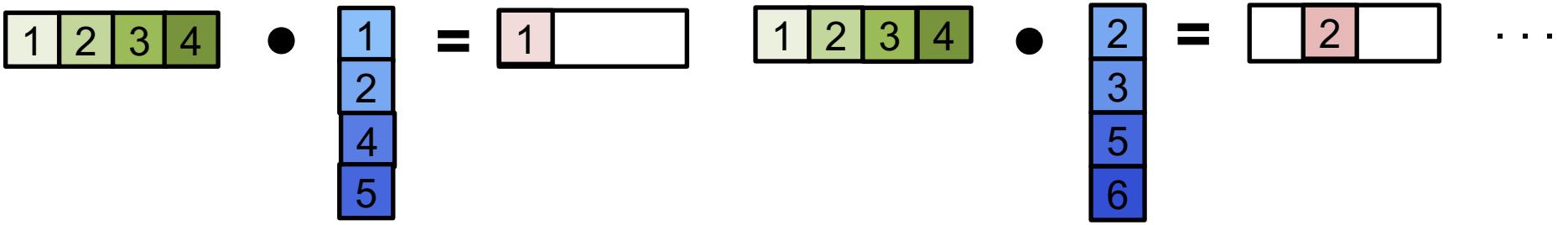


Convolution (CONV) Layer

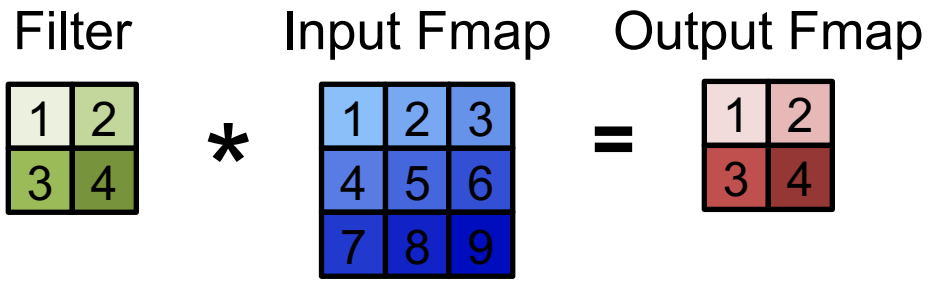


Convolution:

Flattened

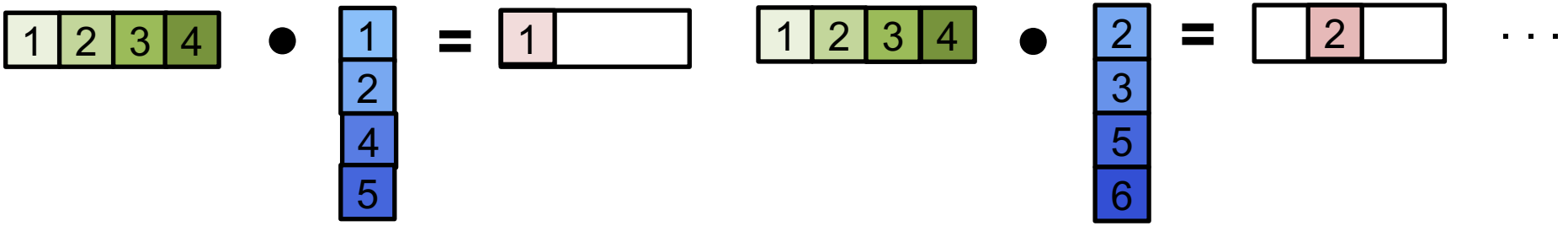


Convolution (CONV) Layer

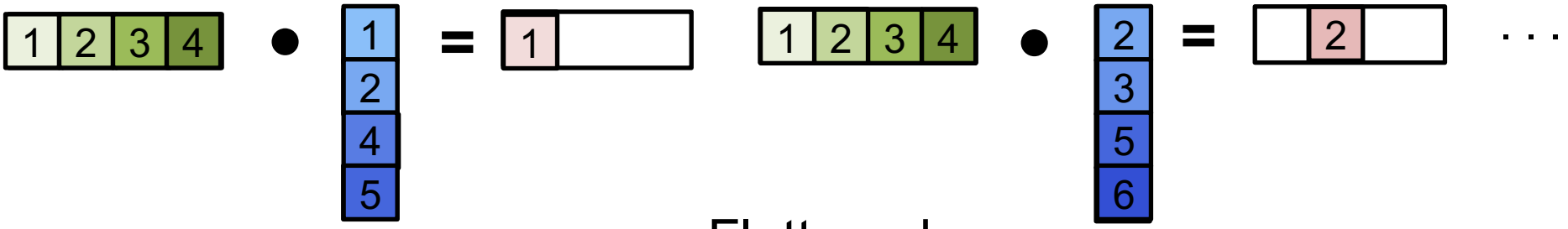


Convolution:

Flattened

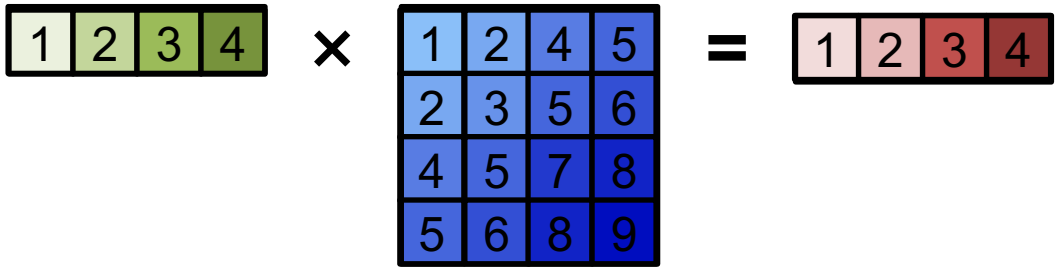


Convolution (CONV) Layer

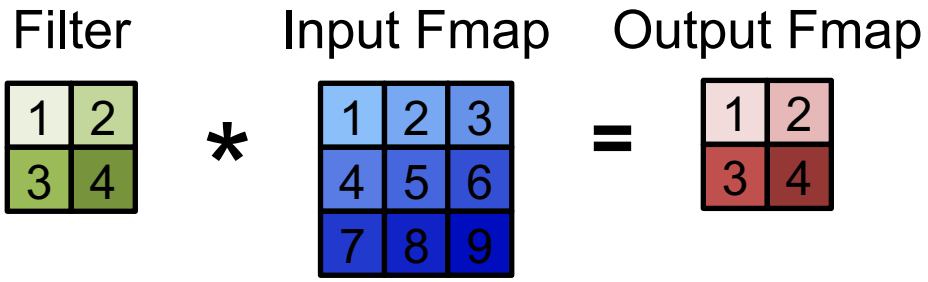


Flattened

Matrix Multiply (by Toeplitz Matrix)

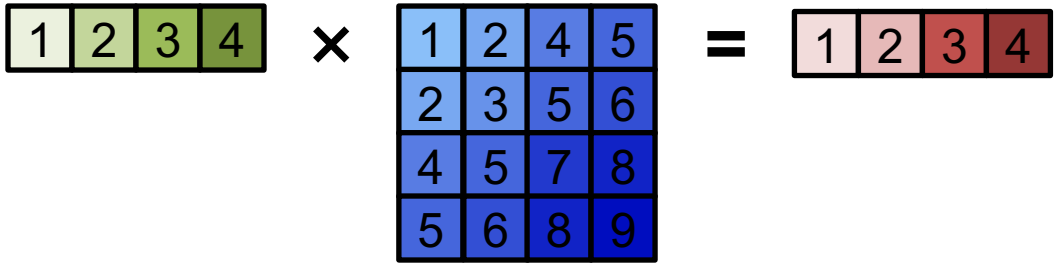


Convolution (CONV) Layer



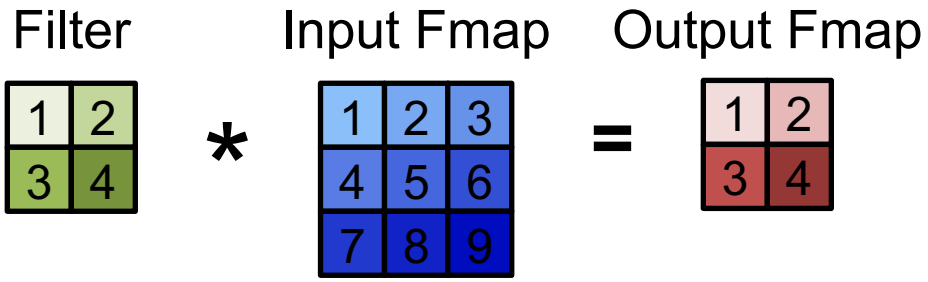
Convolution:

Matrix Multiply (by Toeplitz Matrix)



Convert to matrix multiply using the **Toeplitz Matrix**

Convolution (CONV) Layer



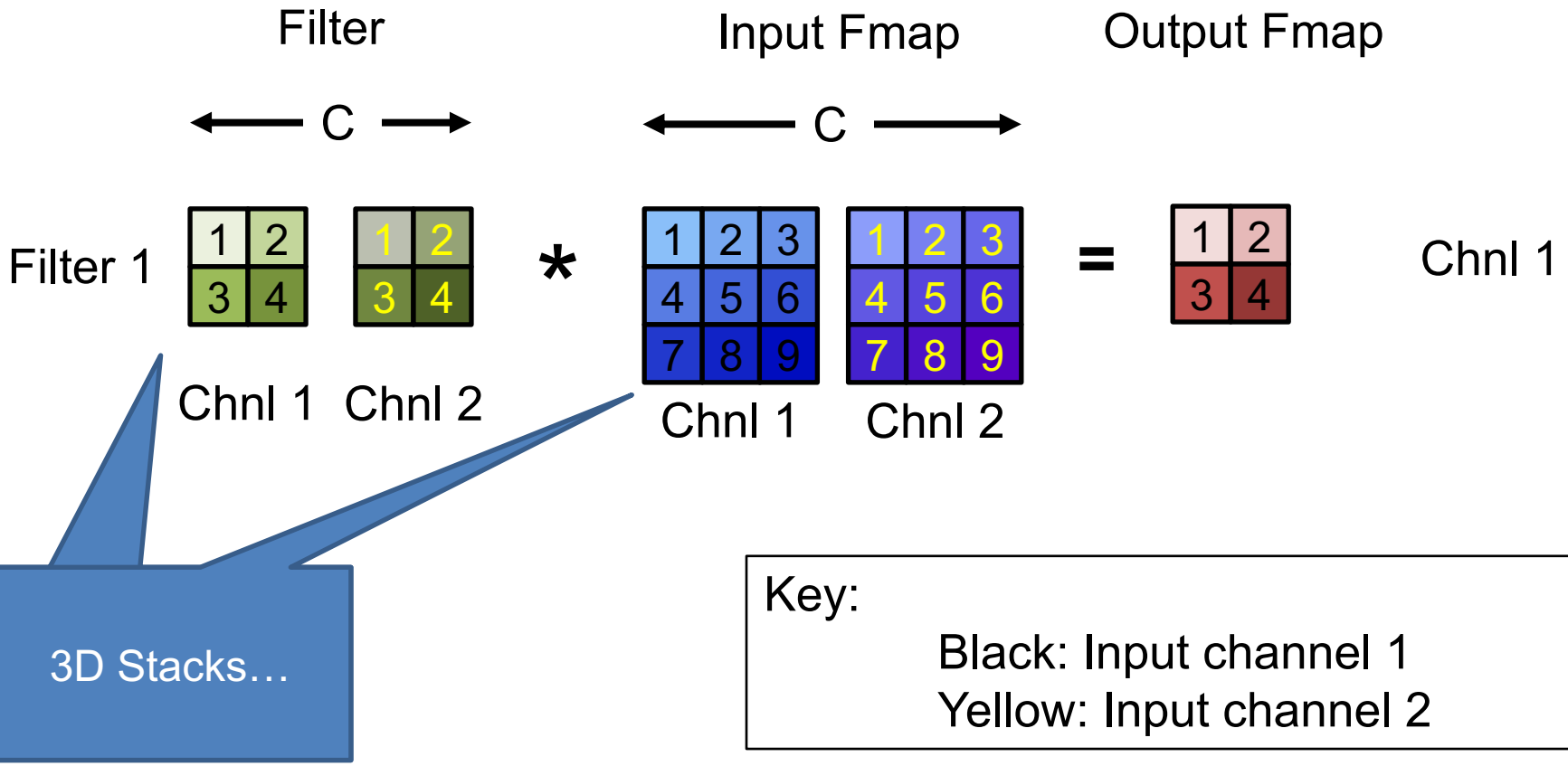
Convolution:

Matrix Multiply (by Toeplitz Matrix)

Data is repeated

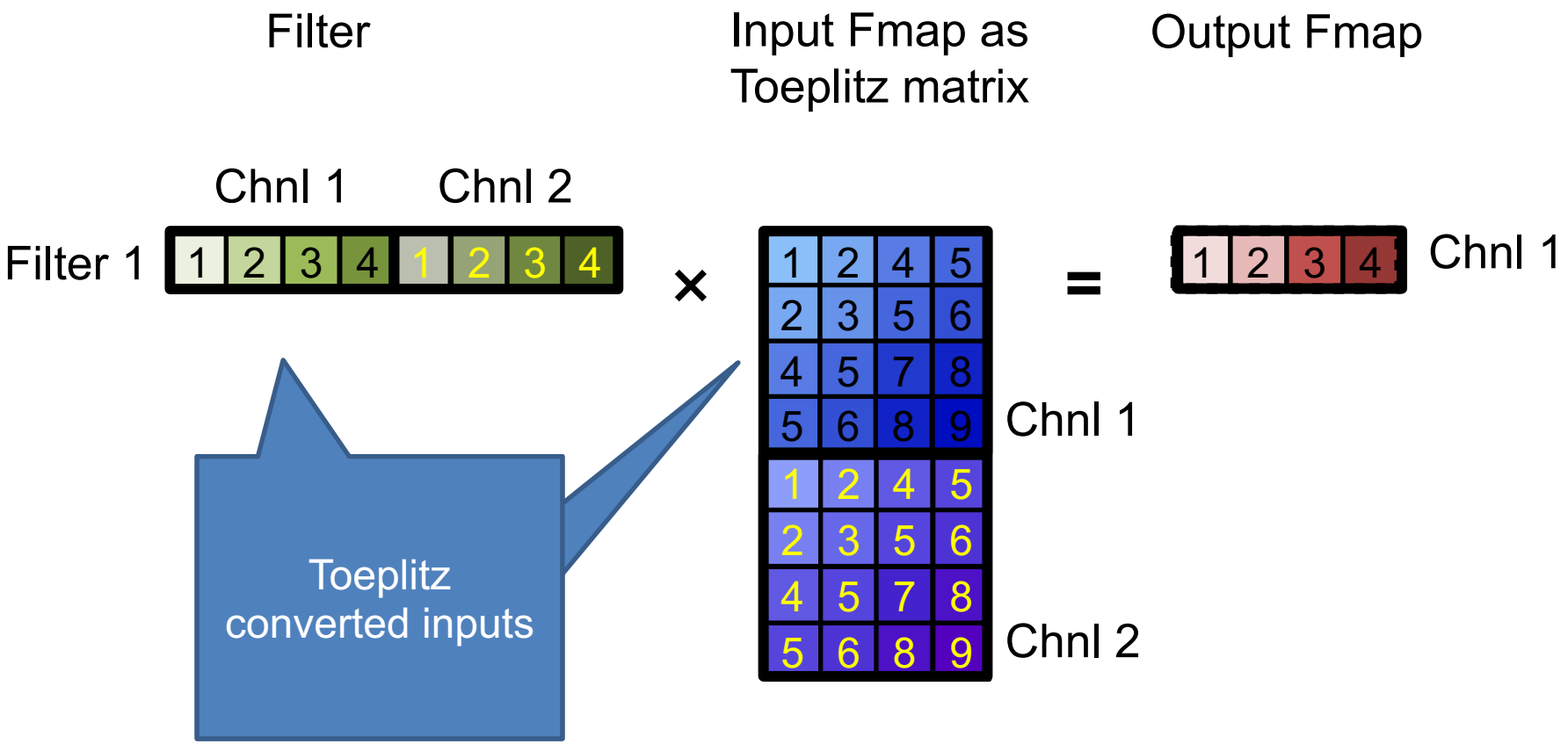
Convolution (CONV) Layer

- Multiple Input Channels



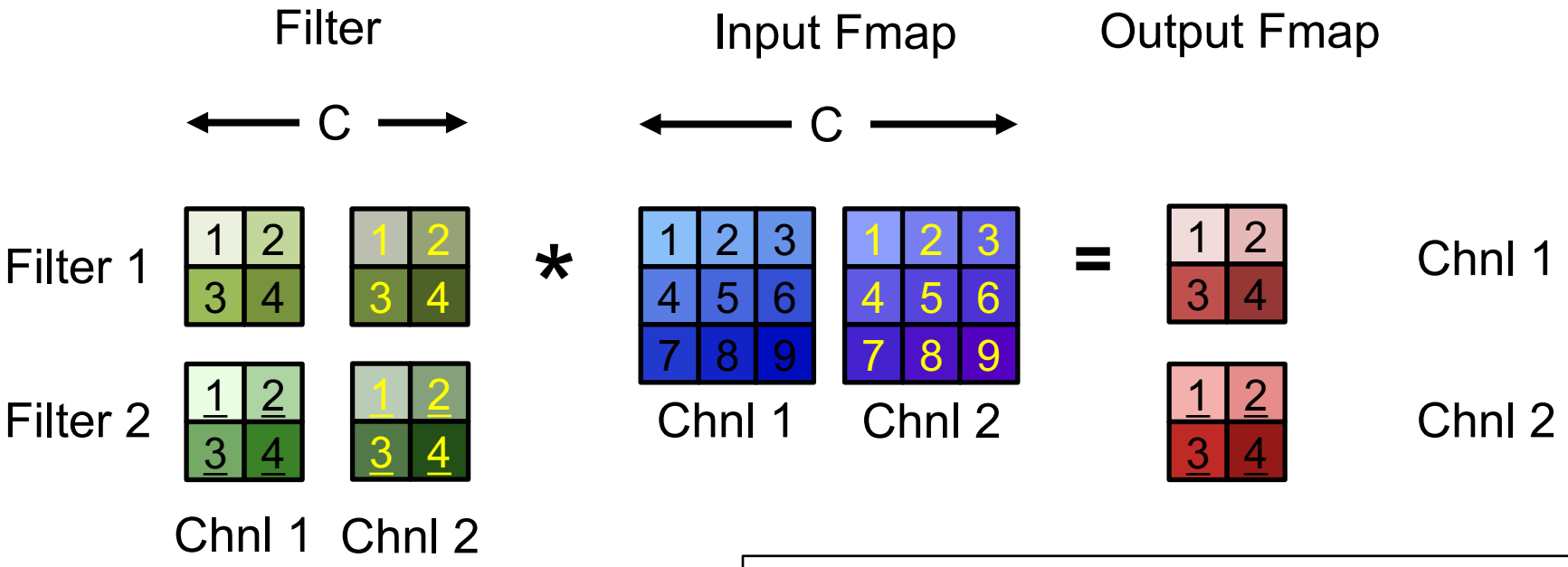
Convolution (CONV) Layer

- Multiple Input Channels



Convolution (CONV) Layer

- Multiple Input Channels and Output Channels

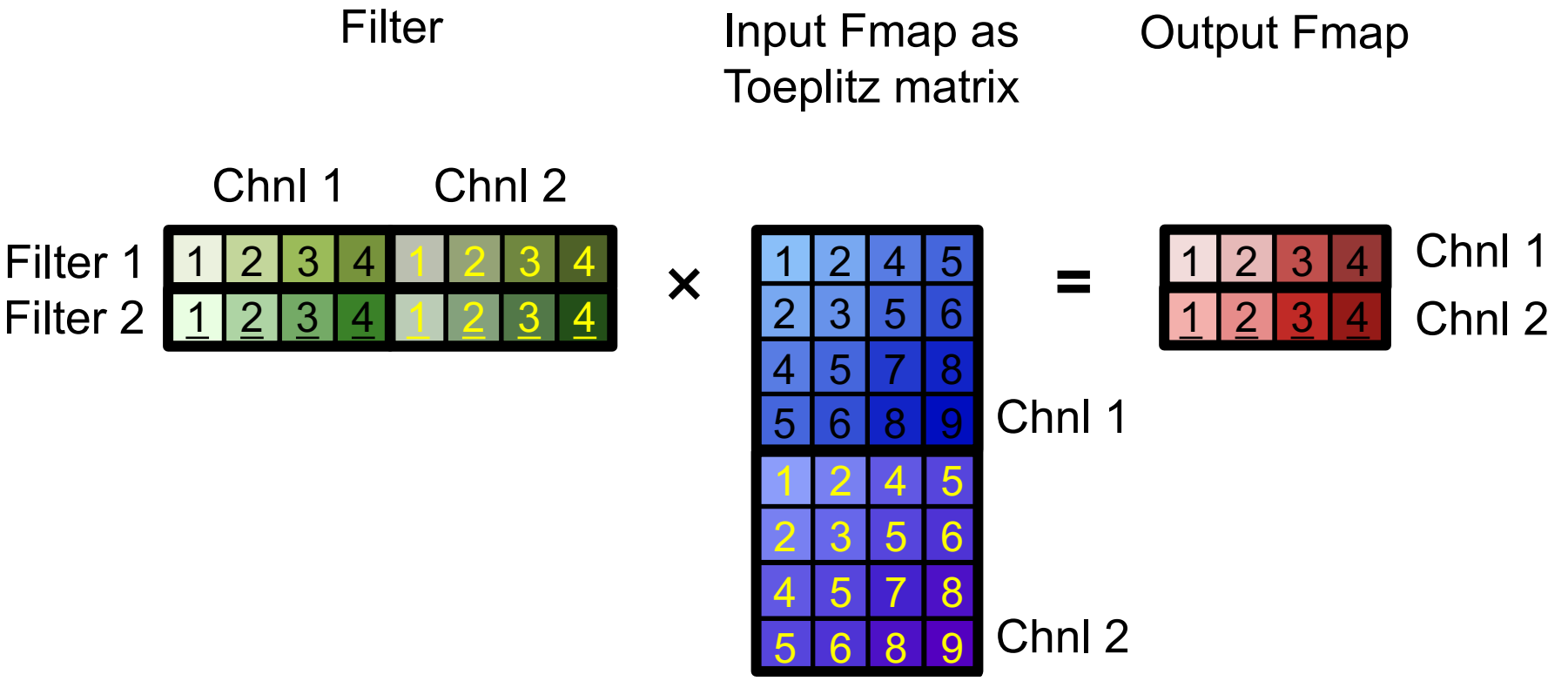


Key:

Black: Input channel 1
Yellow: Input channel 2
Underlined: Output channel 2

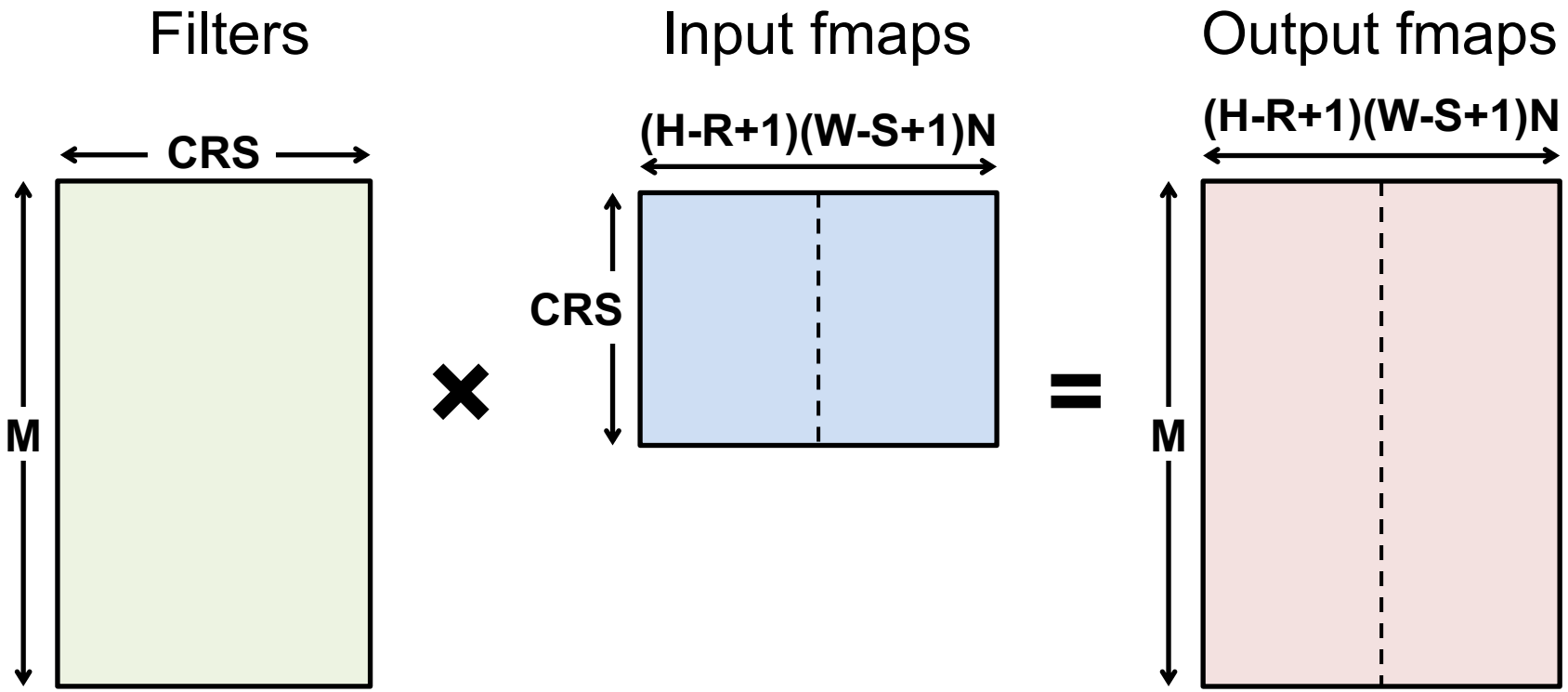
Convolution (CONV) Layer

- Multiple Input Channels and Output Channels



Convolution (CONV) Layer

- Dimensions of matrices for matrix multiply in convolution layers with batch size N



N=2 in example

Computational Transforms

Computation Transformations

- Goal: Bitwise same result, but reduce number of operations
- Focuses mostly on compute

Gauss's Multiplication Algorithm

$$(a + bi)(c + di) = (ac - bd) + (bc + ad)i.$$

4 multiplications + 3 additions

$$k_1 = c \cdot (a + b)$$

$$k_2 = a \cdot (d - c)$$

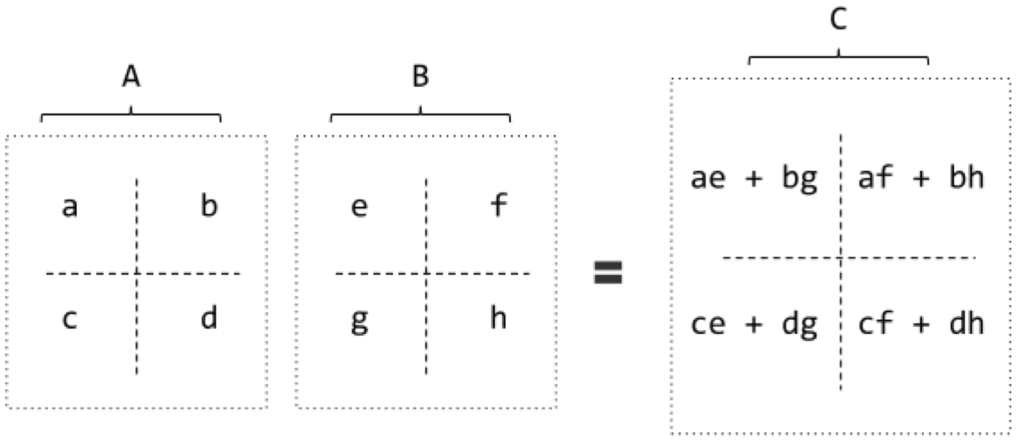
$$k_3 = b \cdot (c + d)$$

$$\text{Real part} = k_1 - k_3$$

$$\text{Imaginary part} = k_1 + k_2.$$

3 multiplications + 5 additions

Strassen



8 multiplications + 4 additions

P1 = a(f – h)
P2 = (a + b)h
P3 = (c + d)e
P4 = d(g – e)

P5 = (a + d)(e + h)
P6 = (b - d)(g + h)
P7 = (a – c)(e + f)

$$AB = \begin{bmatrix} P5 + P4 - P2 + P6 & P1 + P2 \\ P3 + P4 & P1 + P5 - P3 - P7 \end{bmatrix}$$

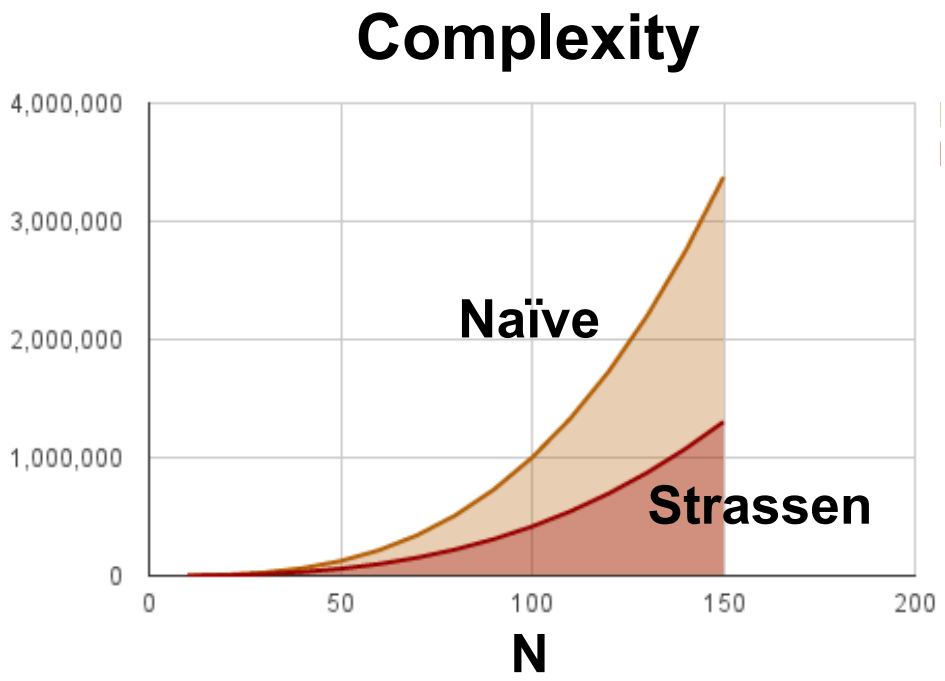
7 multiplications + 18 additions

7 multiplications + 13 additions (for constant B matrix – weights)

[Cong et al., ICANN, 2014]

Strassen

- Reduce the complexity of matrix multiplication from $\Theta(N^3)$ to $\Theta(N^{2.807})$ by reducing multiplications



Comes at the price of reduced numerical stability and requires significantly more memory

Image Source: http://www.stoimen.com/blog/2012/11/26/computer-algorithms-strassens-matrix-multiplication/

Winograd 1D – F(2,3)

- Targeting convolutions instead of matrix multiply
- Notation: F(size of output, filter size)

input filter

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}$$

6 multiplications + 4 additions

[Lavin et al., CVPR 2016]

Winograd 1D – F(2,3)

- Targeting convolutions instead of matrix multiply
- Notation: F(size of output, filter size)

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

input filter

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

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

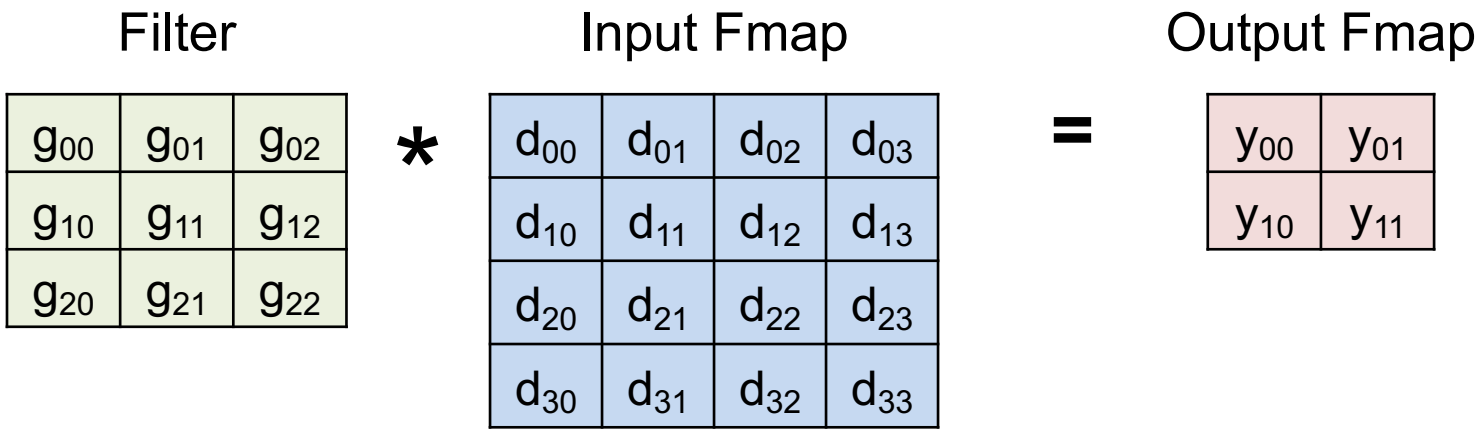
4 multiplications + 12 additions + 2 shifts

4 multiplications + 8 additions (for constant weights)

[Lavin et al., CVPR 2016]

Winograd 2D - F(2x2, 3x3)

- 1D Winograd is nested to make 2D Winograd

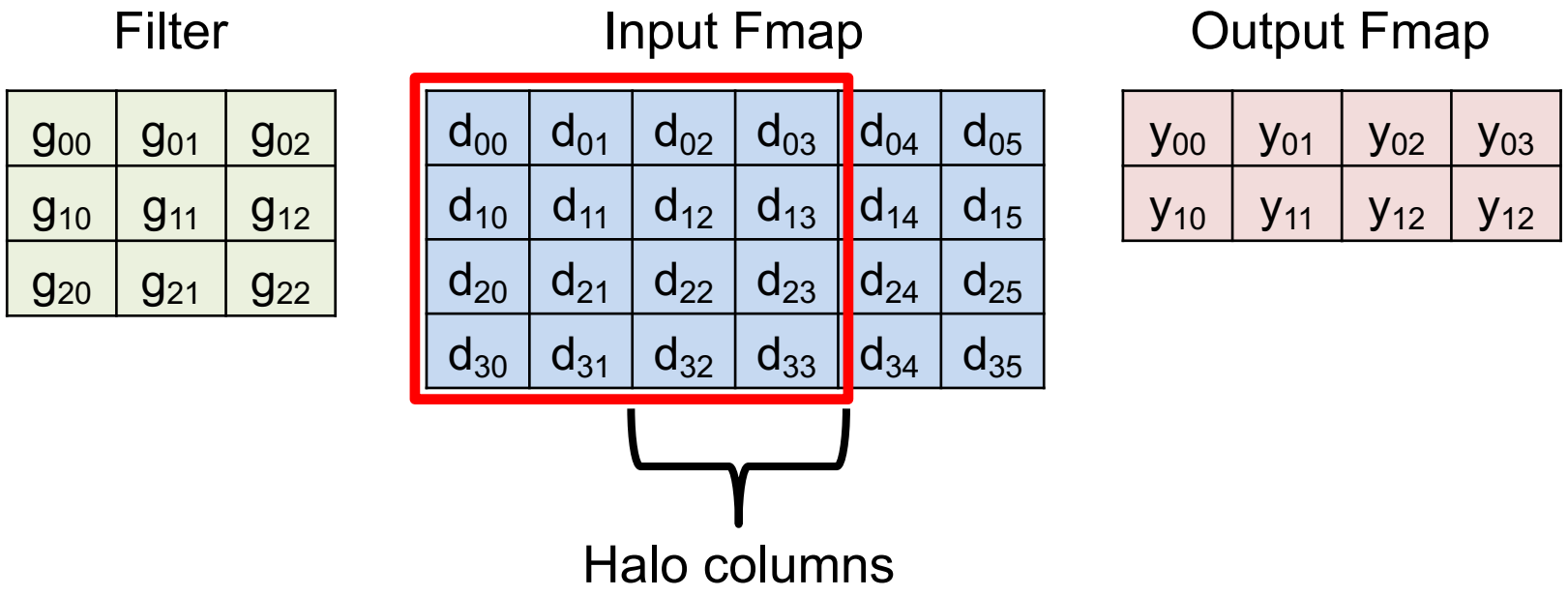


Original: 36 multiplications

Winograd: 16 multiplications → 2.25 times reduction

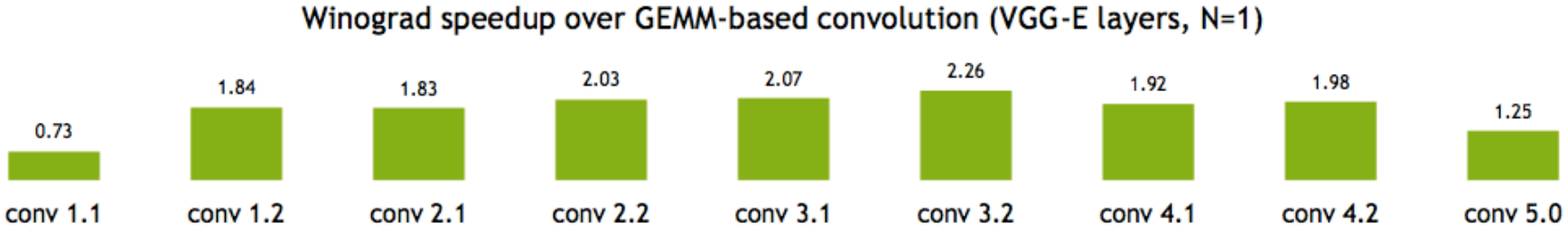
Winograd Halos

- Winograd works on a small region of output at a time, and therefore uses inputs repeatedly



Winograd Performance Varies

Optimal convolution algorithm depends on convolution layer dimensions



Meta-parameters (data layouts, texture memory) afford higher performance

Using texture memory for convolutions: 13% inference speedup

(GoogLeNet, batch size 1)

Source: Nvidia

Winograd Summary

- Winograd is an optimized computation for convolutions
- It can significantly reduce multiplies
  - For example, for 3x3 filter by 2.25X
- But, each filter size (and output size) is a different computation.

Winograd as a Transform

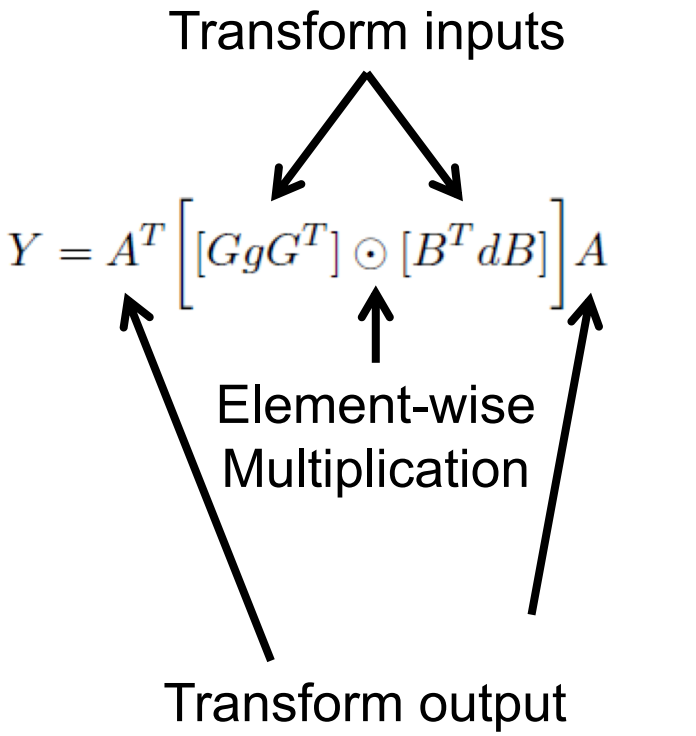
$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

filter $g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$

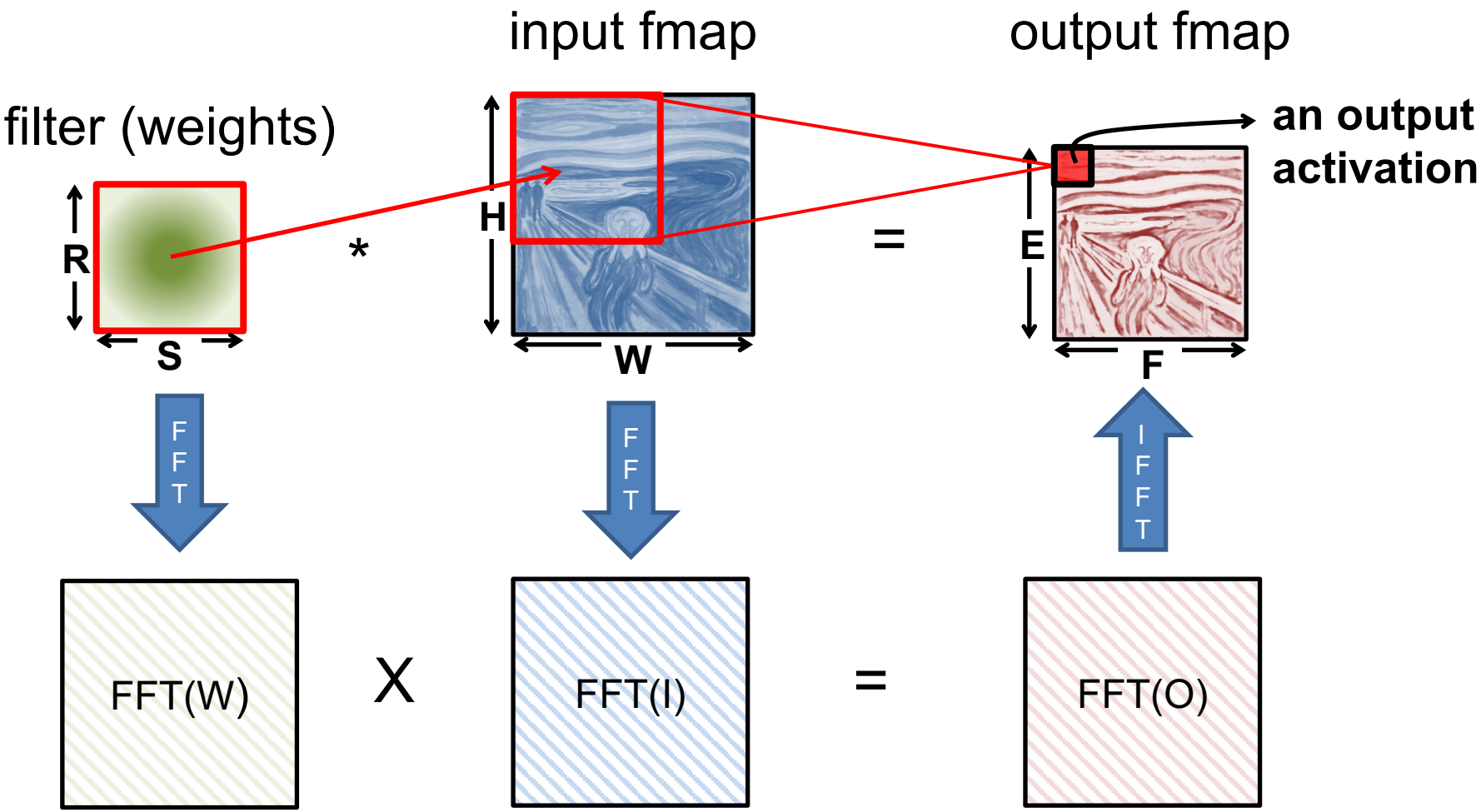
input $d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$



GgG^T can be precomputed

[Lavin et al., CVPR 2016]

Fast Fourier Transform (FFT) Flow



FFT Overview

- Convert filter and input to frequency domain to make convolution a simple multiply then convert back to space domain.

- Convert direct convolution $O(N_o^2 N_f^2)$ computation to $O(N_o^2 \log_2 N_o)$

- So note that computational benefit of FFT decreases with decreasing size of filter

[Mathieu et al., ArXiv 2013, Vasilache et al., ArXiv 2014]

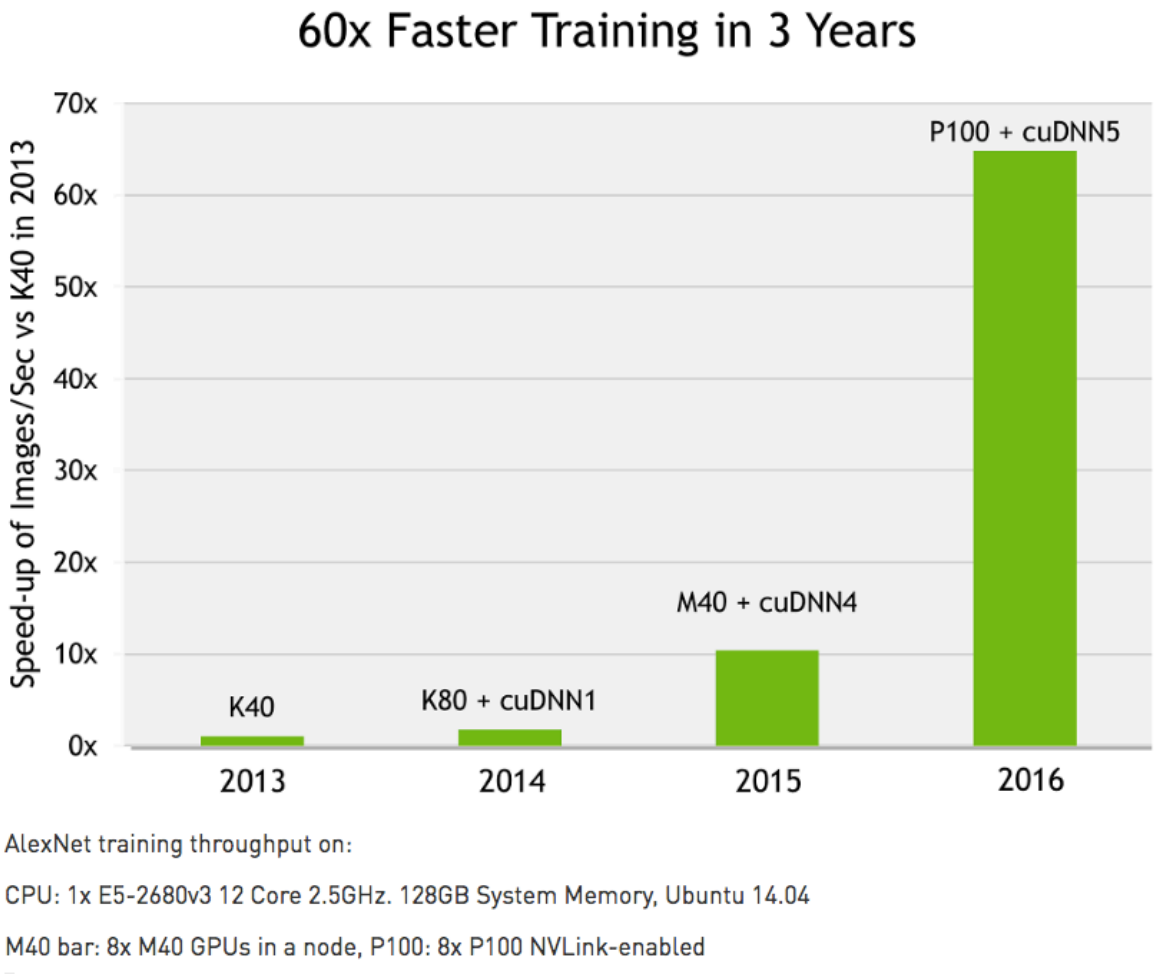
FFT Costs

- Input and Filter matrices are ‘0-completed’,
  - i.e., expanded to size E+R-1 x F+S-1
- Frequency domain matrices are same dimensions as input, but complex.
- FFT often reduces computation, but requires much more memory space and bandwidth

Optimization opportunities

- FFT of real matrix is symmetric allowing one to save ½ the computes
- Filters can be pre-computed and stored, but convolutional filter in frequency domain is much larger than in space domain
- Can reuse frequency domain version of input for creating different output channels to avoid FFT re-computations
- Can accumulate across channels before performing inverse transform to reduce number of IFFT

cuDNN: Speed up with Transformations



Source: Nvidia

Optimization Tools

- Halide
- TVM
- Timeloop
- ….