Kernel Computation

Chia-Chi Tsai (蔡家齊)
cctsai@gs.ncku.edu.tw
AI System Lab
Department of Electrical Engineering
National Cheng Kung University

Outline

- Overview
- Matrix Multiplication Based Convolution
- Tiling for Optimizing Performance
- Computation Transform Optimizations

Outline

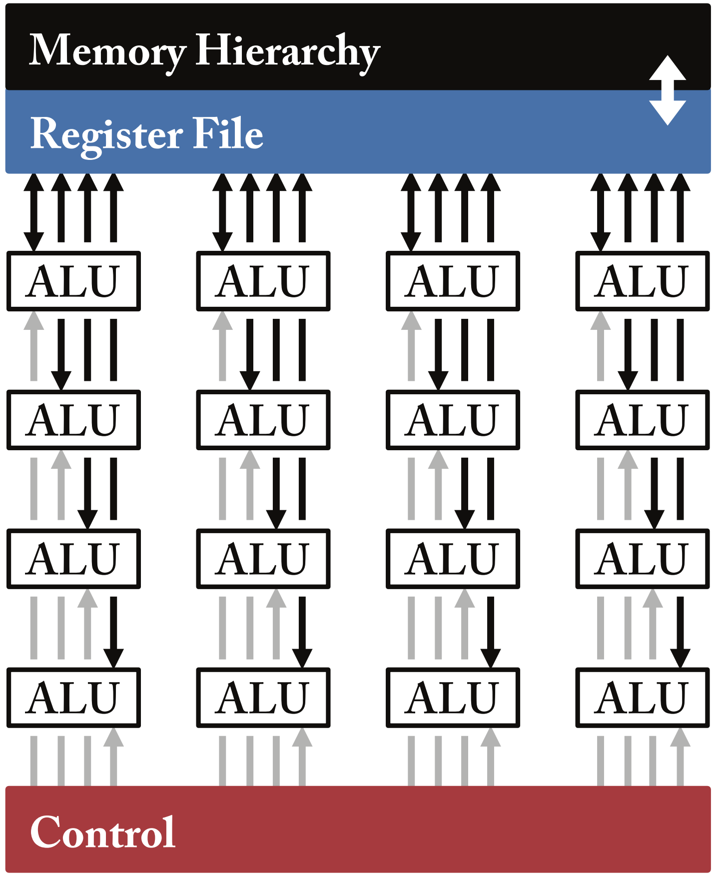
- Overview
- Matrix Multiplication Based Convolution
- Tiling for Optimizing Performance
- Computation Transform Optimizations

Fundamental Computation of AI

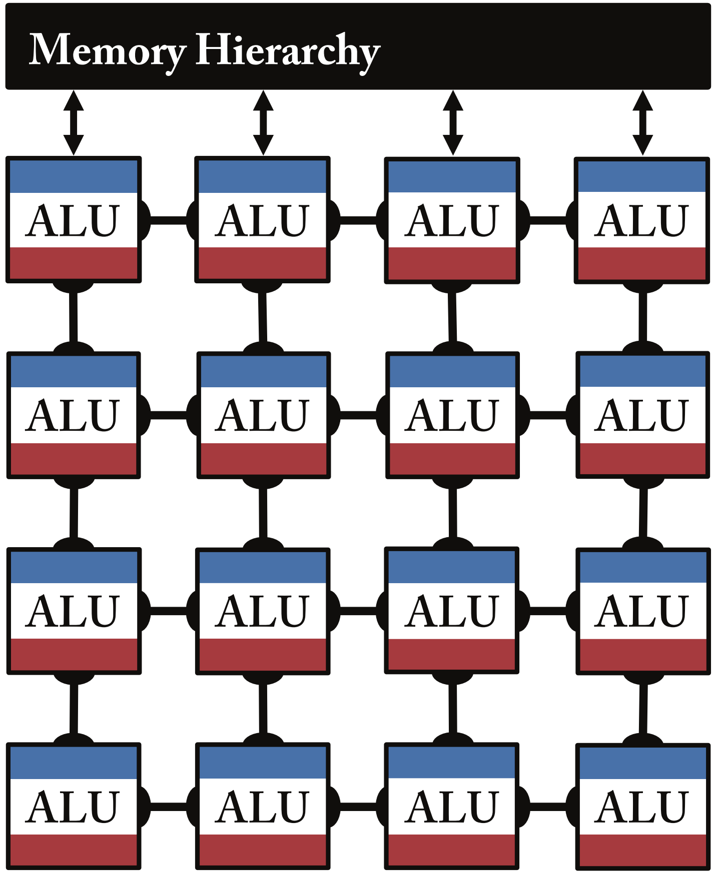
- Fundamental computation
  - Convolution Layer
  - Fully Connected Layer
  - Consist of Multiply-and-accumulate(MAC) operations
- Flexibility and parallelization is the key
- Temporal and spatial parallelism

Temporal and Spatial Architecture

Temporal Architecture
(SIMD/SIMT)



Spatial Architecture
(Dataflow Processing)



Temporal Architecture

- Centralized control for a large number of arithmetic logic units (ALUs)
  - Fetch data from the memory hierarchy
  - Cannot communicate directly with each other
- Commonly seem in CPUs and GPUs
  - Vector instructions(SIMD)
  - Parallel threads(SIMT)

Spatial Architecture

- Allow for communication between ALUs
- Use dataflow processing
  - ALUs form a processing chain so that they can pass data from one to another directly
- Each ALU may can have its own control logic and local memory
  - Scratchpad or register file
- ALU with its own local memory → Processing Engine(PE)
- Commonly seem for processing DNNs in ASIC- and FPGA-based designs

Temporal System Support for DNNs

- With the rise in popularity of DNN, many programmable temporal systems (i.e., CPUs and GPUs) started adding features that target DNN processing
- Intel Knights Landing CPU
  - Special vector instructions
  - Performed multiple fused multiply accumulate operations
- Nvidia PASCAL GP100 GPU
  - 16-bit floating point (fp16) arithmetic
  - Perform two fp16 operations on a single precision core
- Nvidia VOLTA GV100 GPU
  - Special compute unit for performing matrix multiplication and accumulation
  - Individual instructions that perform many MAC operations

Systems of DNN processing

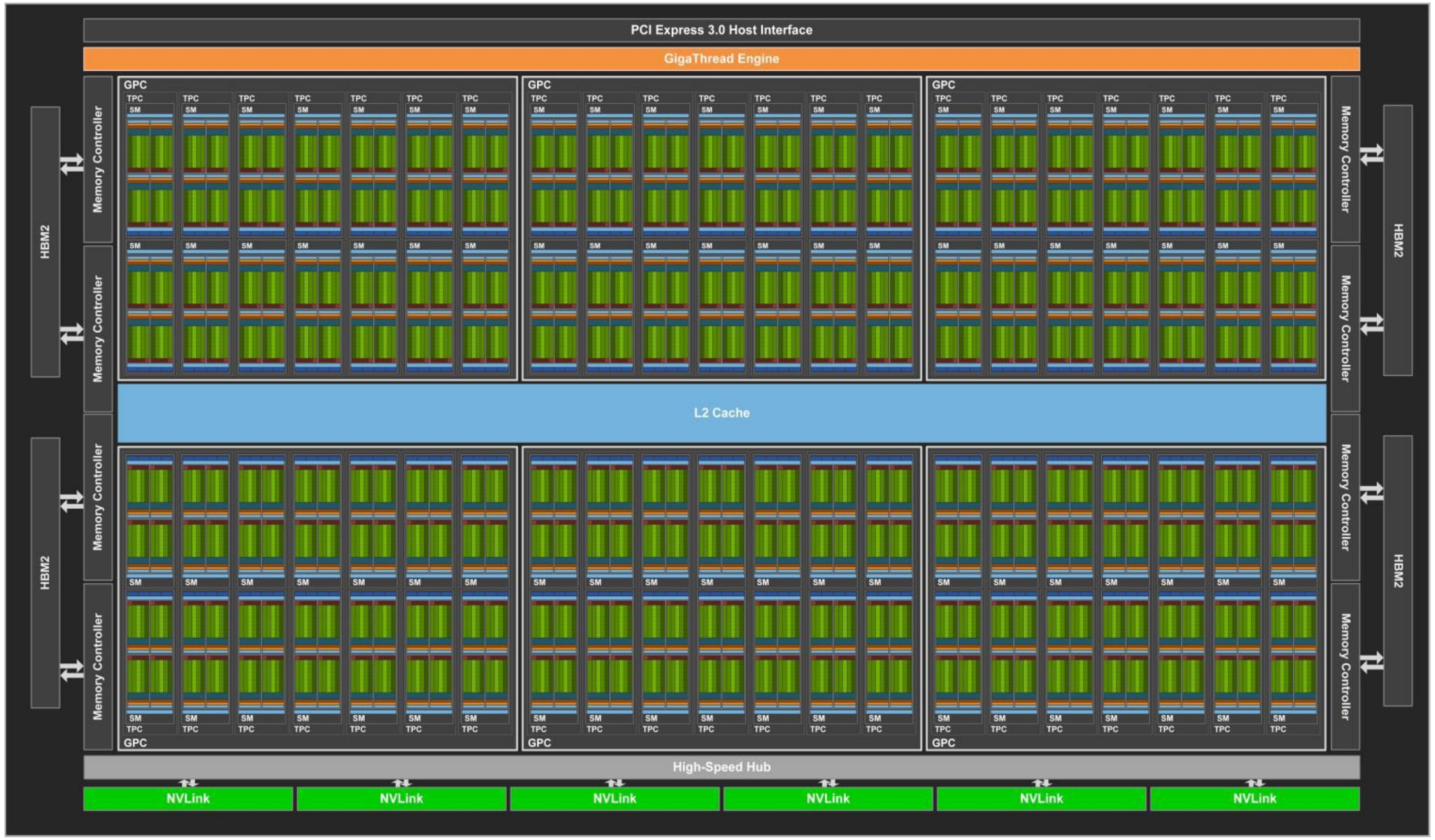
- Facebook’s Big Basin custom DNN server
- Nvidia’s DGX-1
- Apple’s A Series
- Nvidia’s Tegra
- Samsung’s Exynos

Things to Learn

- For temporal architecture
  - How DNN algorithms can be mapped optimized on these platforms
  - How **computational transforms** on the kernel can reduce the number of **multiplications** to **increase throughput**
  - How the computation (e.g., MACs) can be ordered (i.e., tiled) to improve memory subsystem behavior
- For spatial architecture
  - How dataflows can increase data reuse from low-cost memories in the memory hierarchy to reduce energy consumption
  - How other architectural features can help optimize data movement

Volta GV100

- 84 SM Units
- 120 TFLOPS(FP16)
- 400 GFLOPS/W(FP16)



Volta GV100 Streaming Multiprocessor(SM)

- 8 Tensor Cores per SM
- 640 Tensor Cores

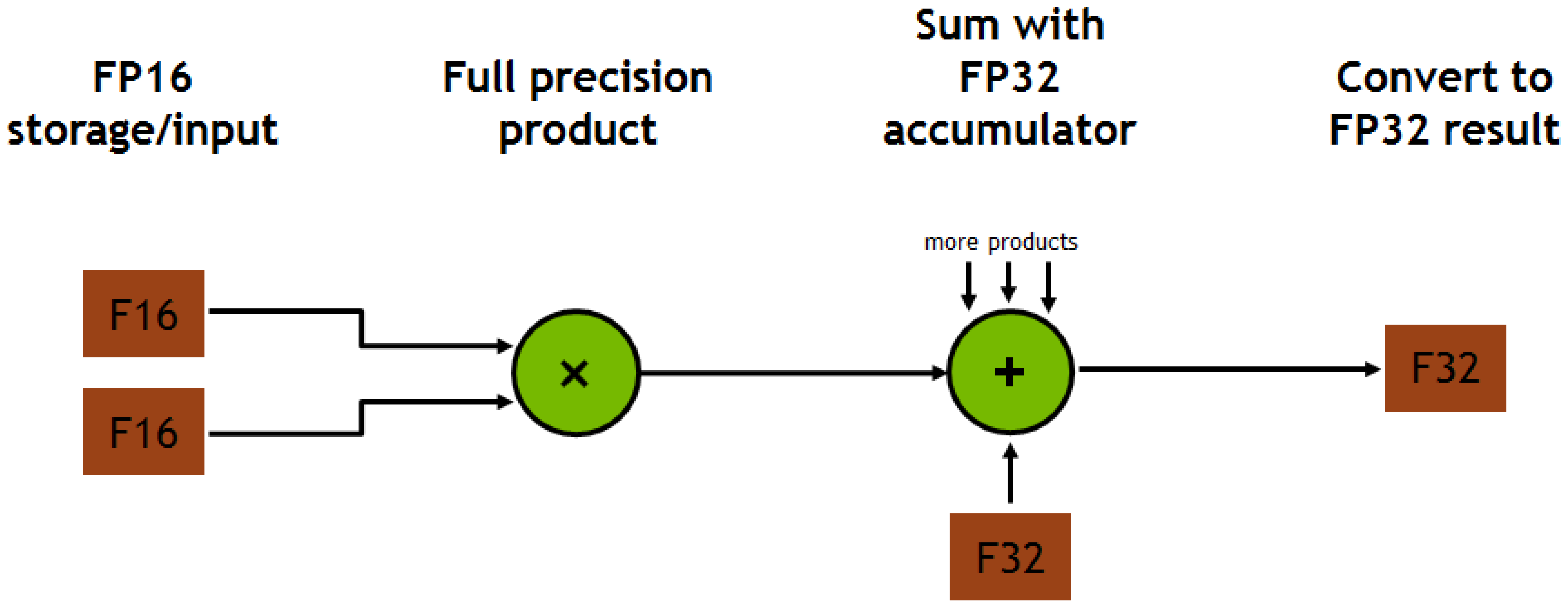
GV100 – Tensor Cores

$$\underbrace{D}_{\text{FP16 or FP32}} = \underbrace{\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}}_{\text{FP16}} \underbrace{\begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}}_{\text{FP16}} + \underbrace{\begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}}_{\text{FP16 or FP32}}$$

Tensor Core 4x4x4 matrix multiply and accumulate

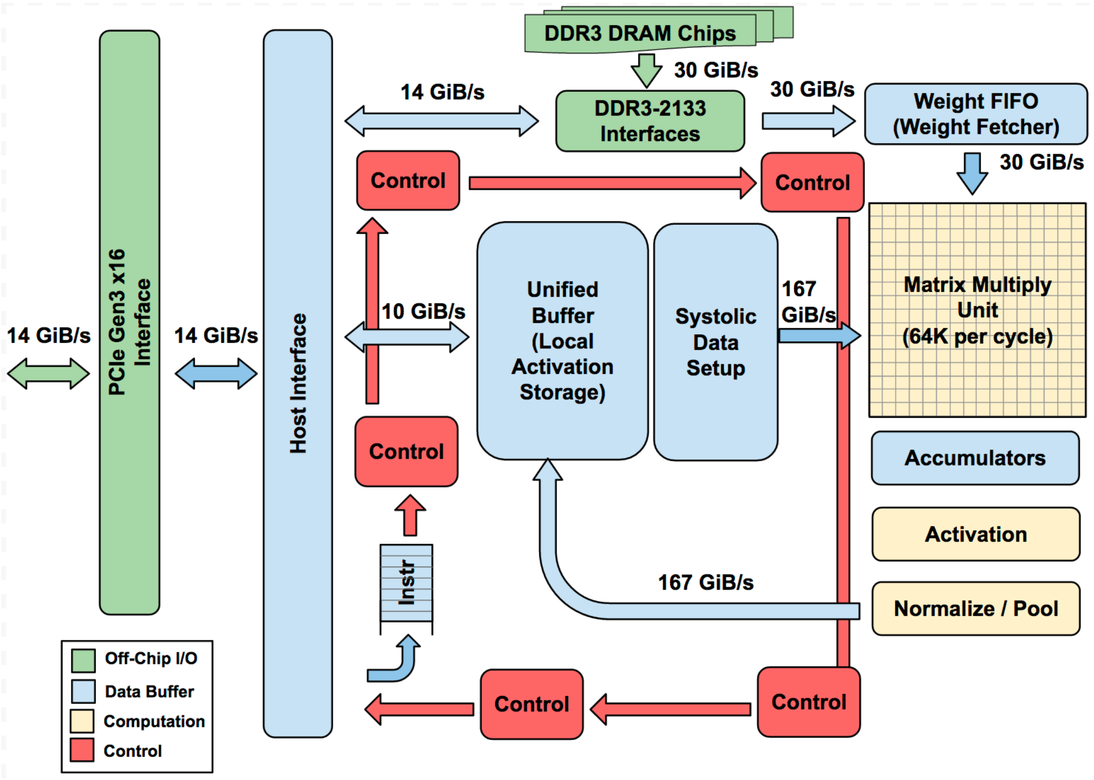
- New opcodes – Matrix Multiply Accumulate (HMMA)
- Number of FP16 operands? Inputs 48, Outputs 16
- 64 Multiplies/clock
- 64 Adds/clock

GV100 – Tensor Cores Operation

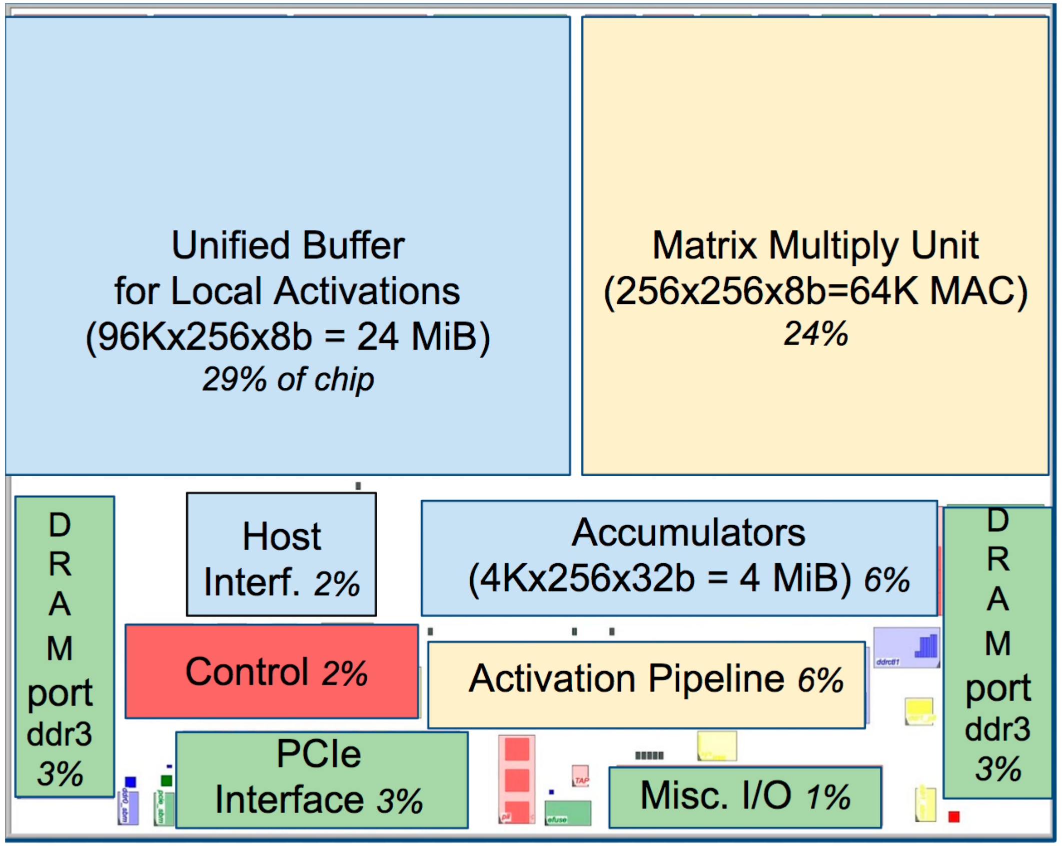


TPU

- Top-Level Architecture



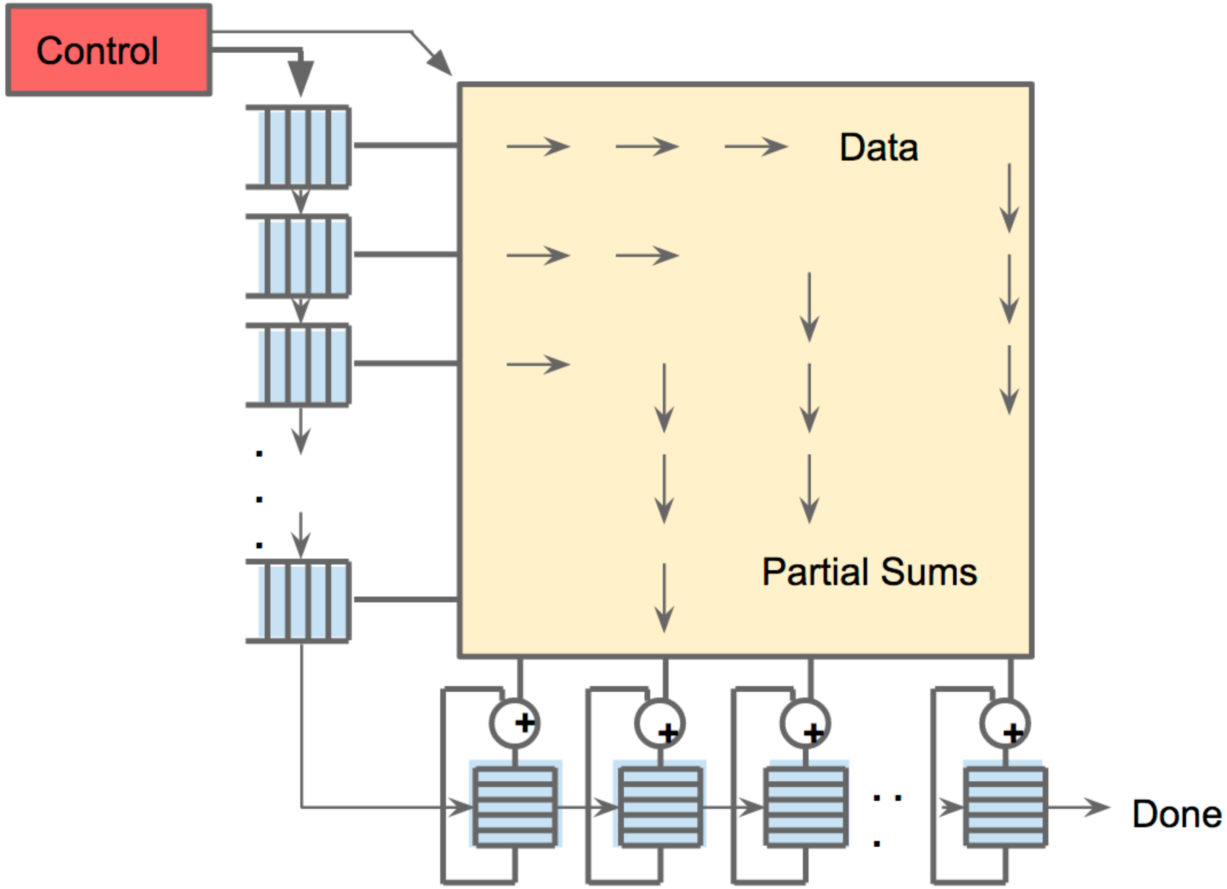
- Floorplan



Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., ... & Yoon, D. H. (2017, June). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture* (pp. 1-12).

TPU

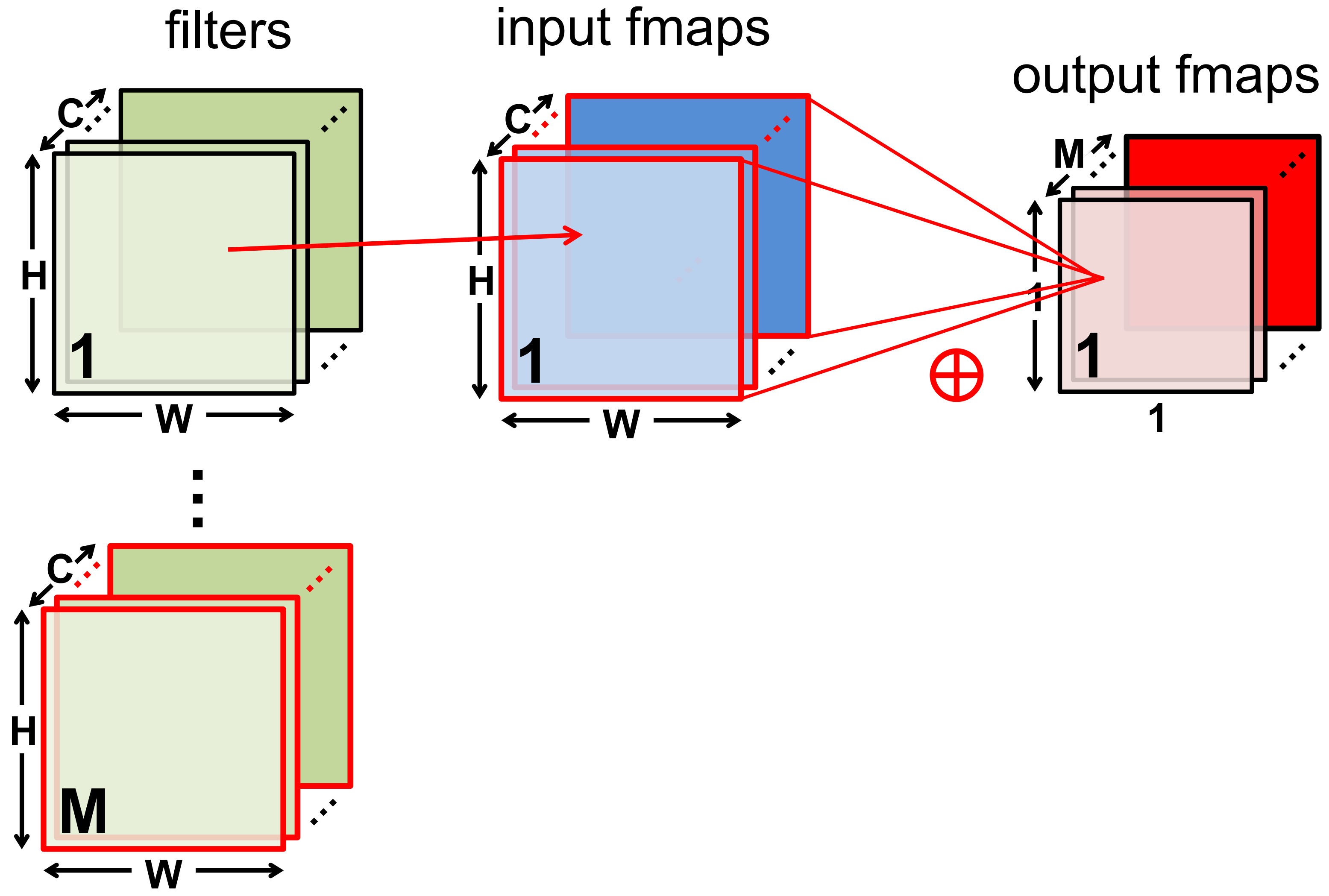
- Systolic data flow of Matrix Multiply Unit



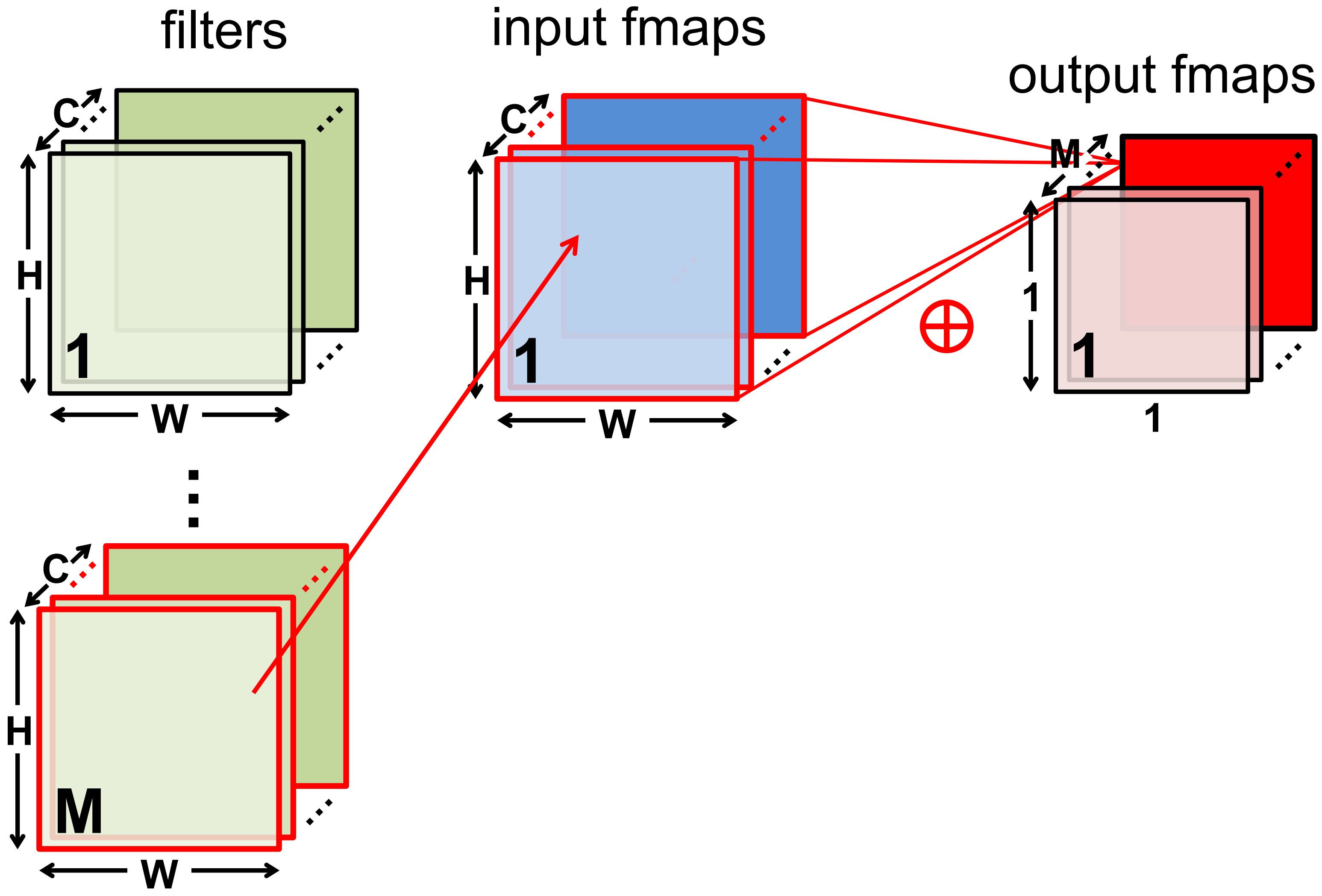
Outline

- Overview
- Matrix Multiplication Based Convolution
- Tiling for Optimizing Performance
- Computation Transform Optimizations

Fully-Connected Layer



Fully-Connected Layer



Flattened 2D Dot Product

Filter | Input Feature map | Output Feature map

$$\begin{bmatrix} \mathbf{F}_{11} & \mathbf{F}_{12} \\ \mathbf{F}_{21} & \mathbf{F}_{22} \end{bmatrix} \bullet \begin{bmatrix} \mathbf{I}_{11} & \mathbf{I}_{12} \\ \mathbf{I}_{21} & \mathbf{I}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{O}_{11} \end{bmatrix}$$

2D dot product

$$\Rightarrow$$

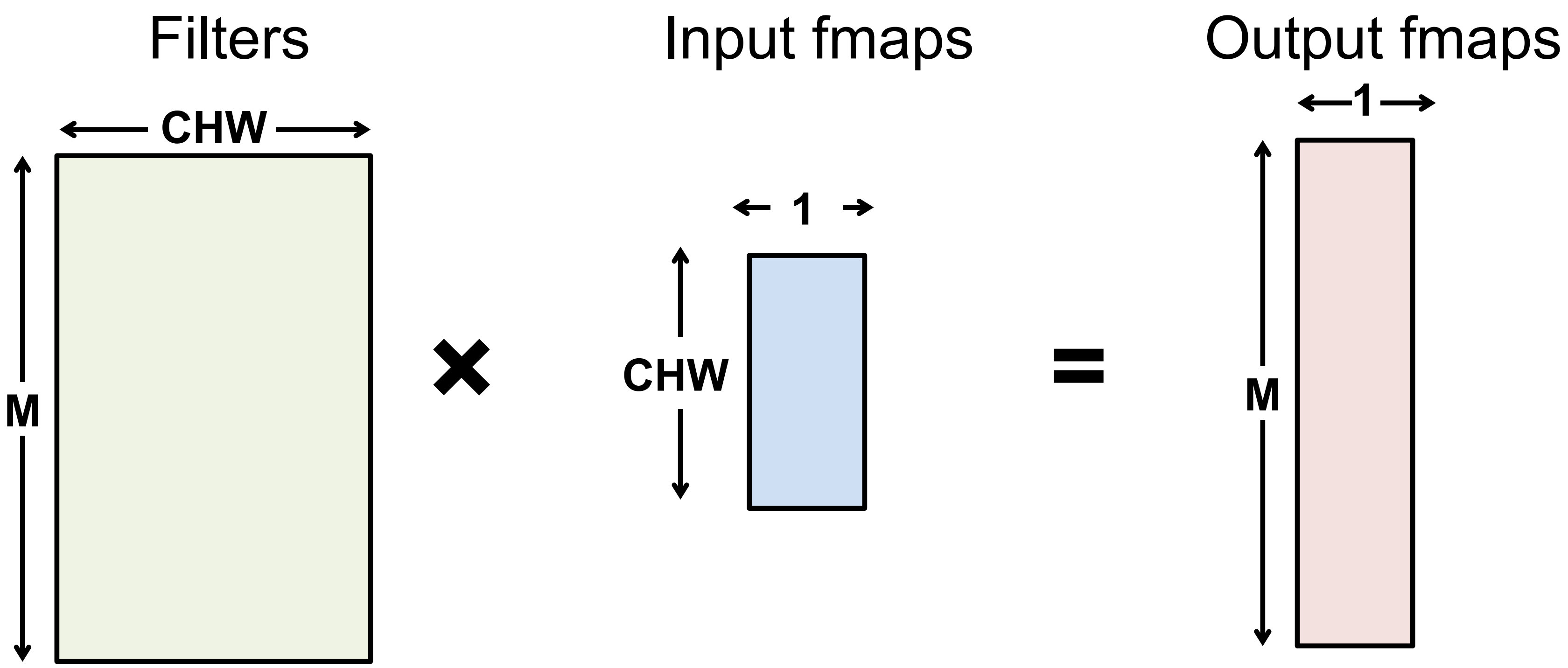
Flattened Filter | Flattened Input Feature map | Output Feature map

$$\begin{bmatrix} \mathbf{F}_{11} & \mathbf{F}_{12} & \mathbf{F}_{21} & \mathbf{F}_{22} \end{bmatrix} \bullet \begin{bmatrix} \mathbf{I}_{11} \\ \mathbf{I}_{12} \\ \mathbf{I}_{21} \\ \mathbf{I}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{O}_{11} \end{bmatrix}$$

2D dot product

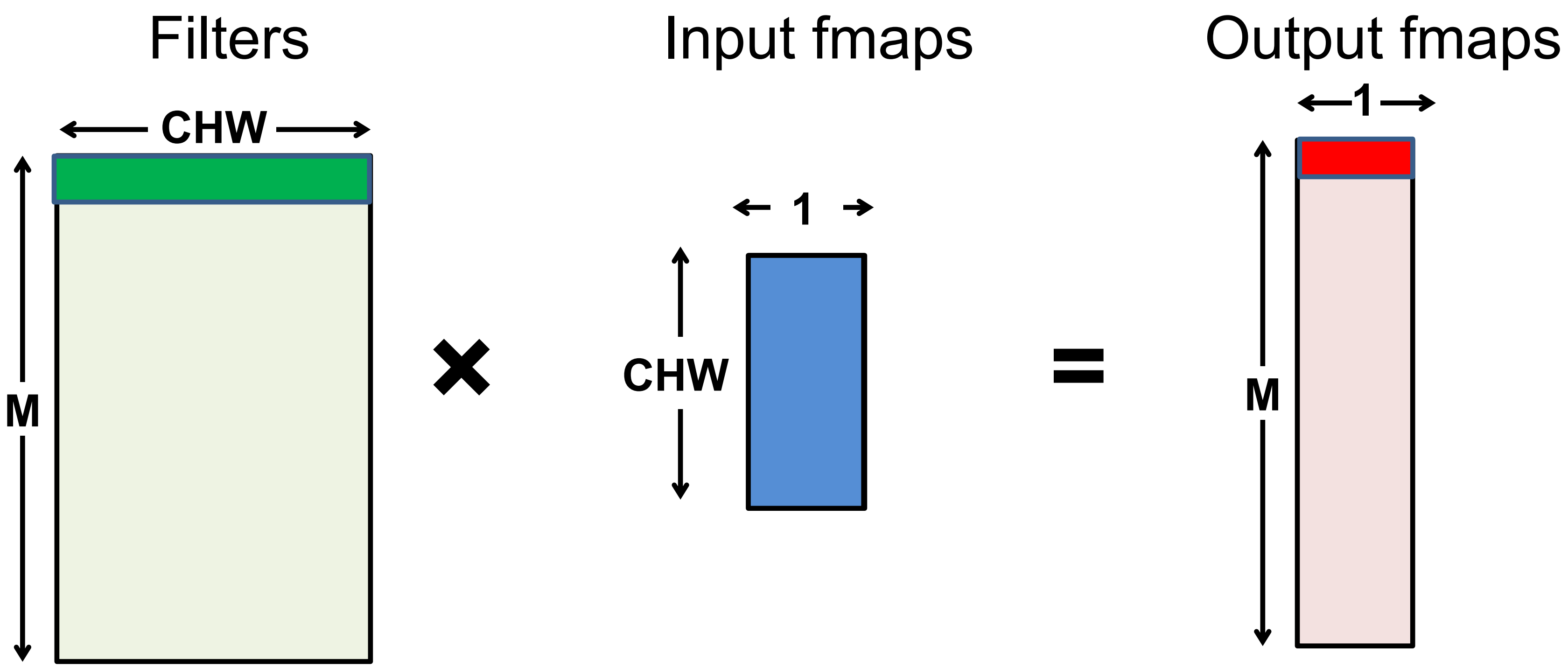
Fully-Connected Layer

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



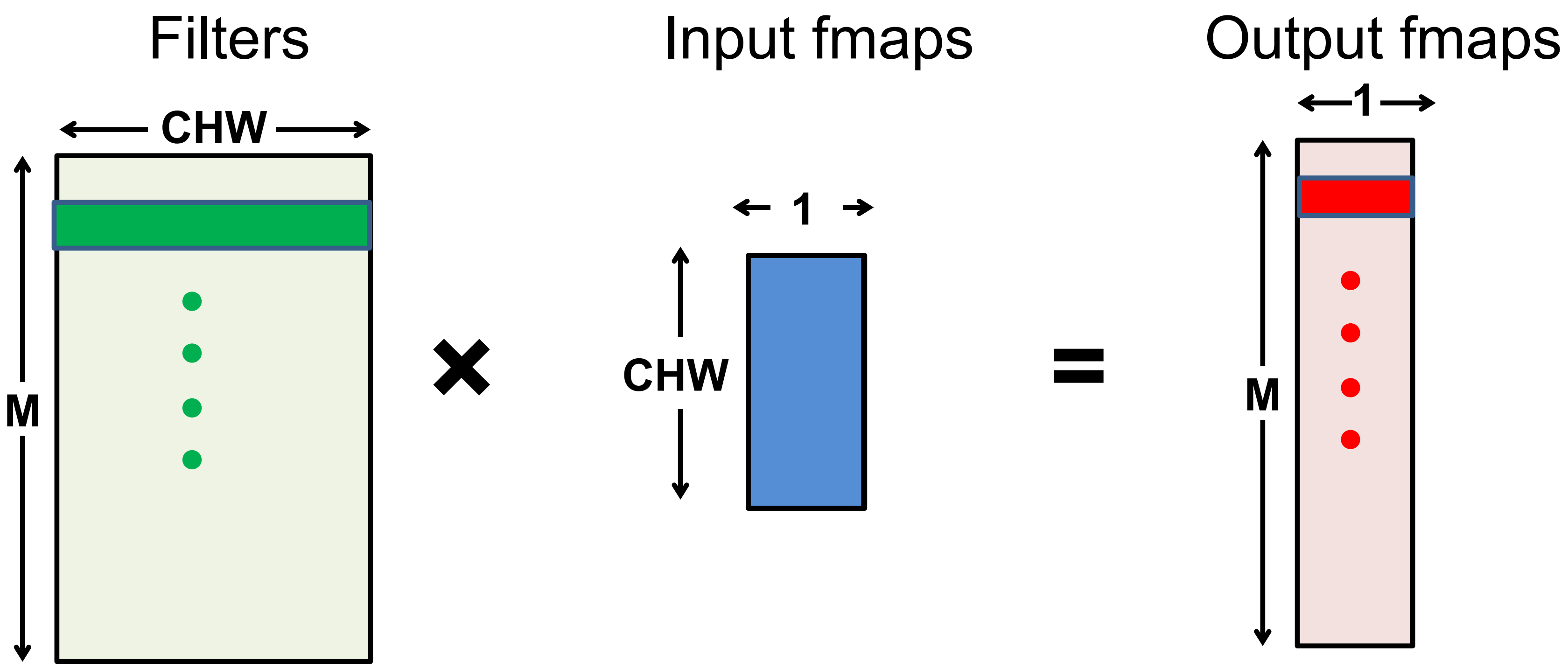
Fully-Connected Layer

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

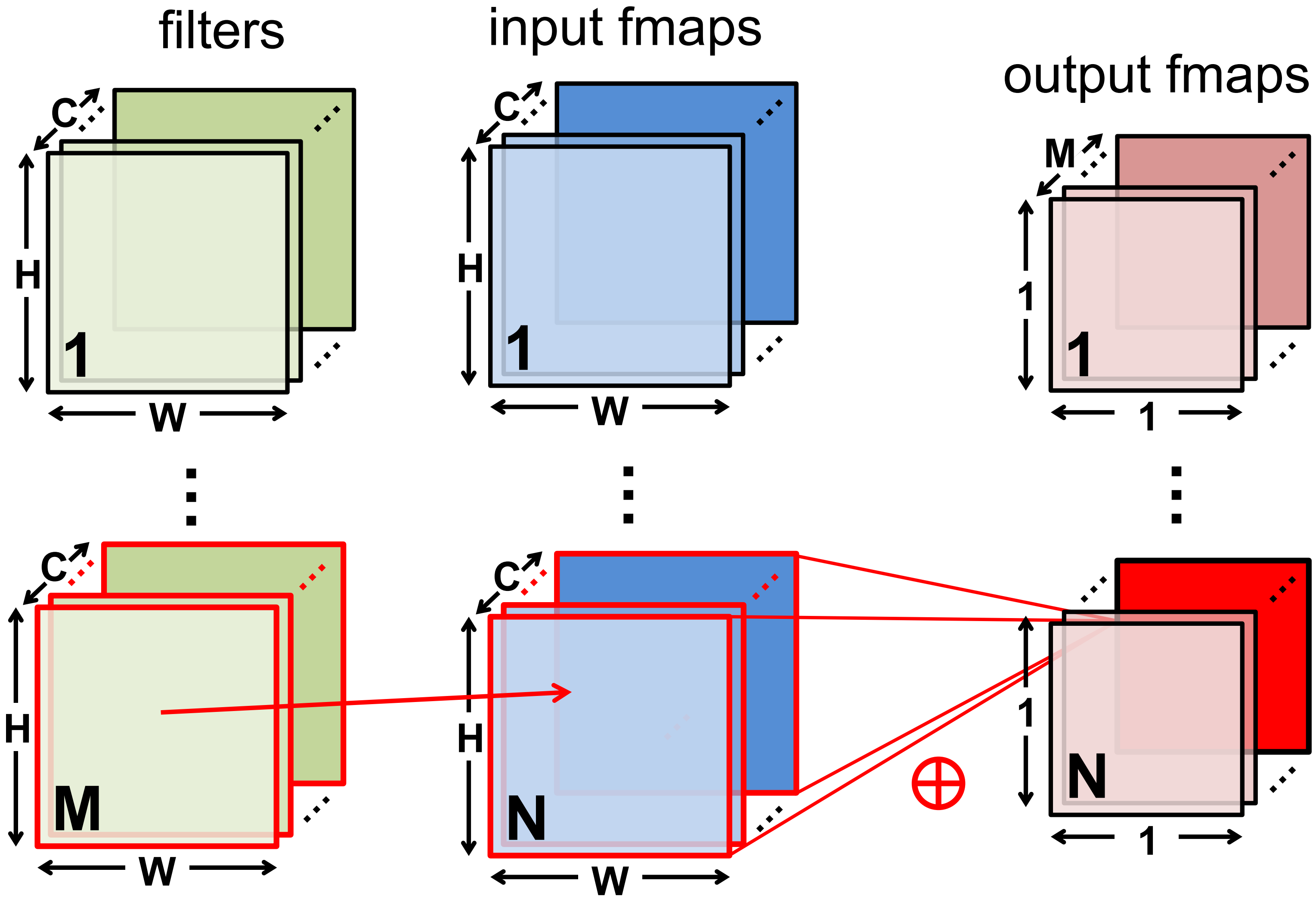


Fully-Connected Layer

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

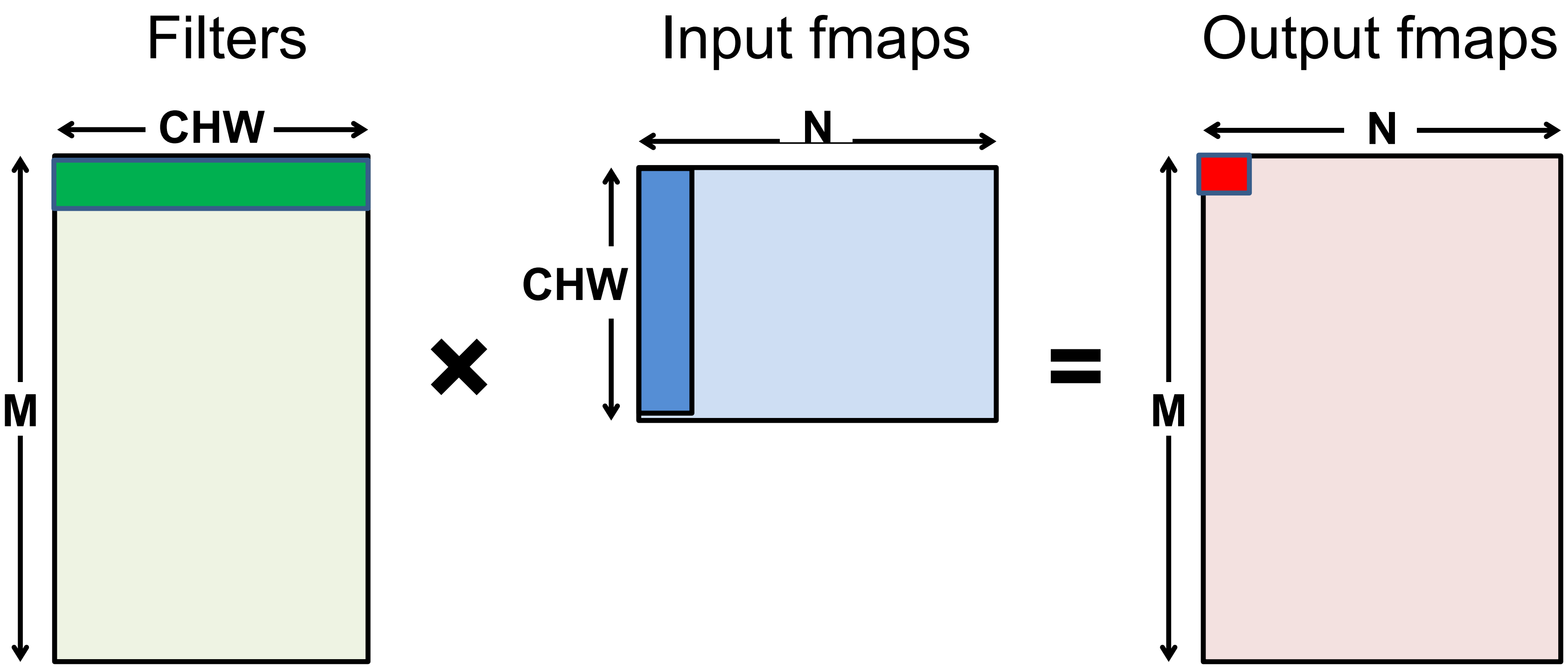


Fully-Connected Layer



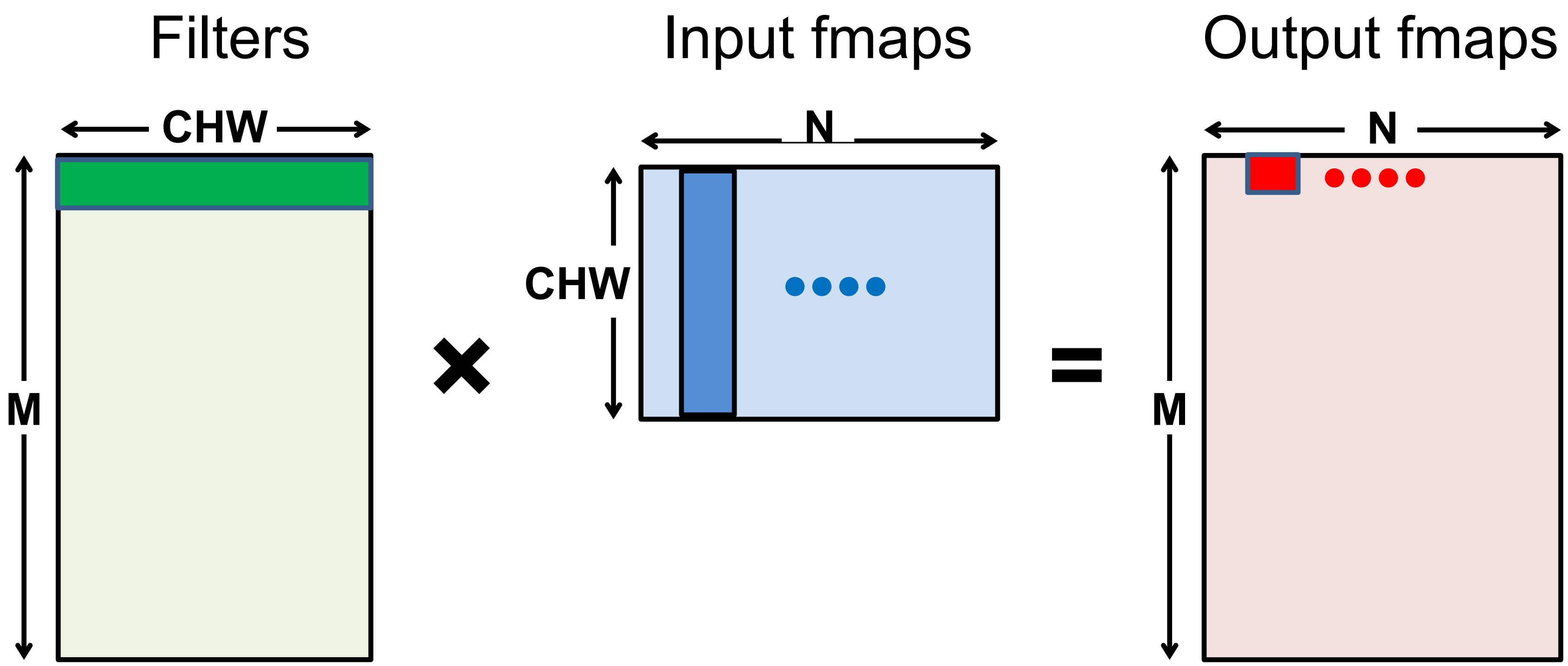
Flattened Fully-Connected Layer

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



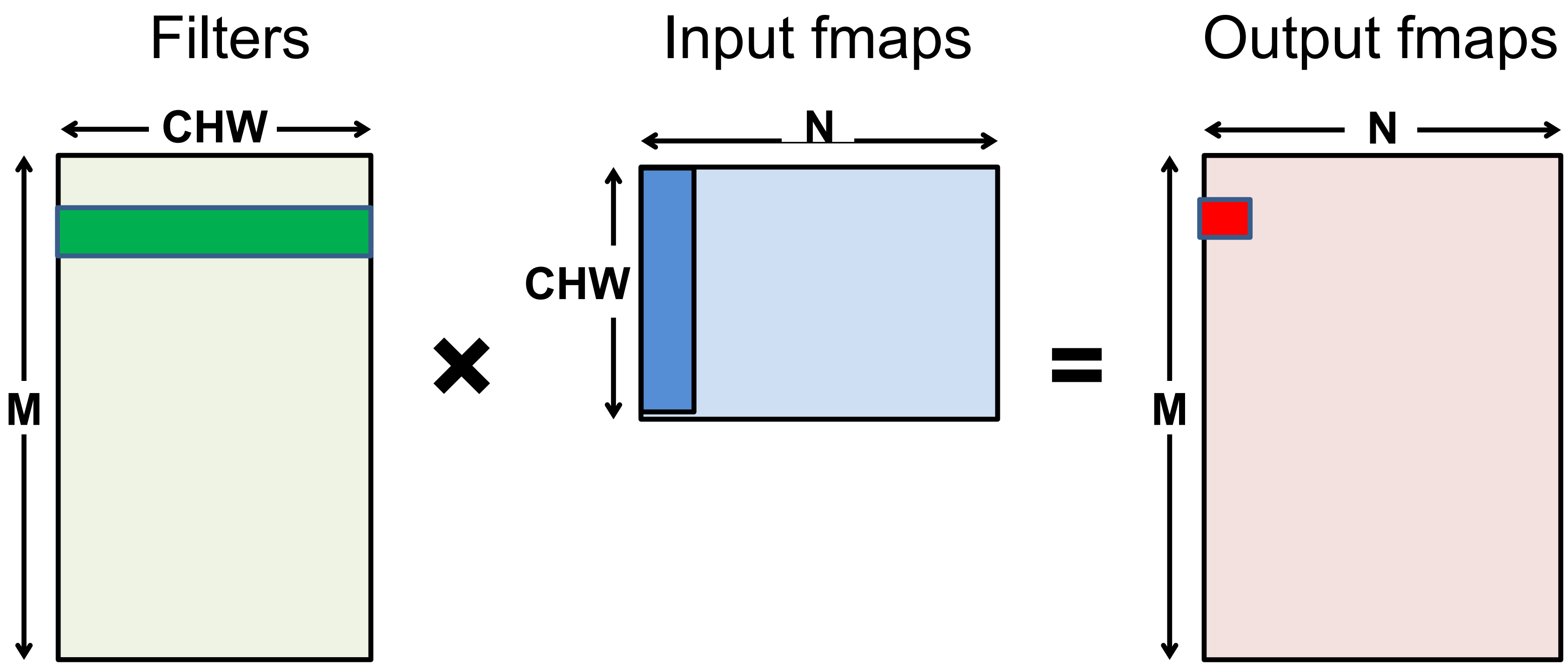
Flattened Fully-Connected Layer

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



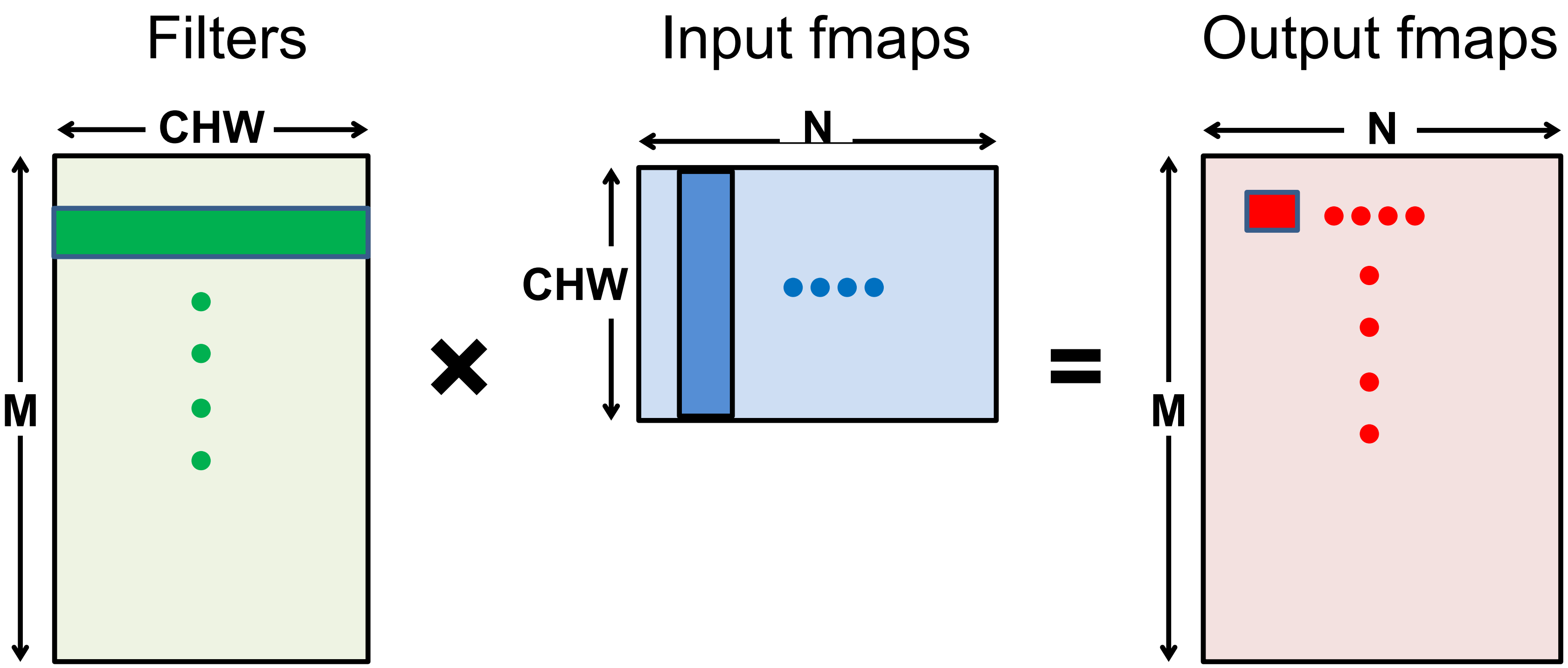
Flattened Fully-Connected Layer

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

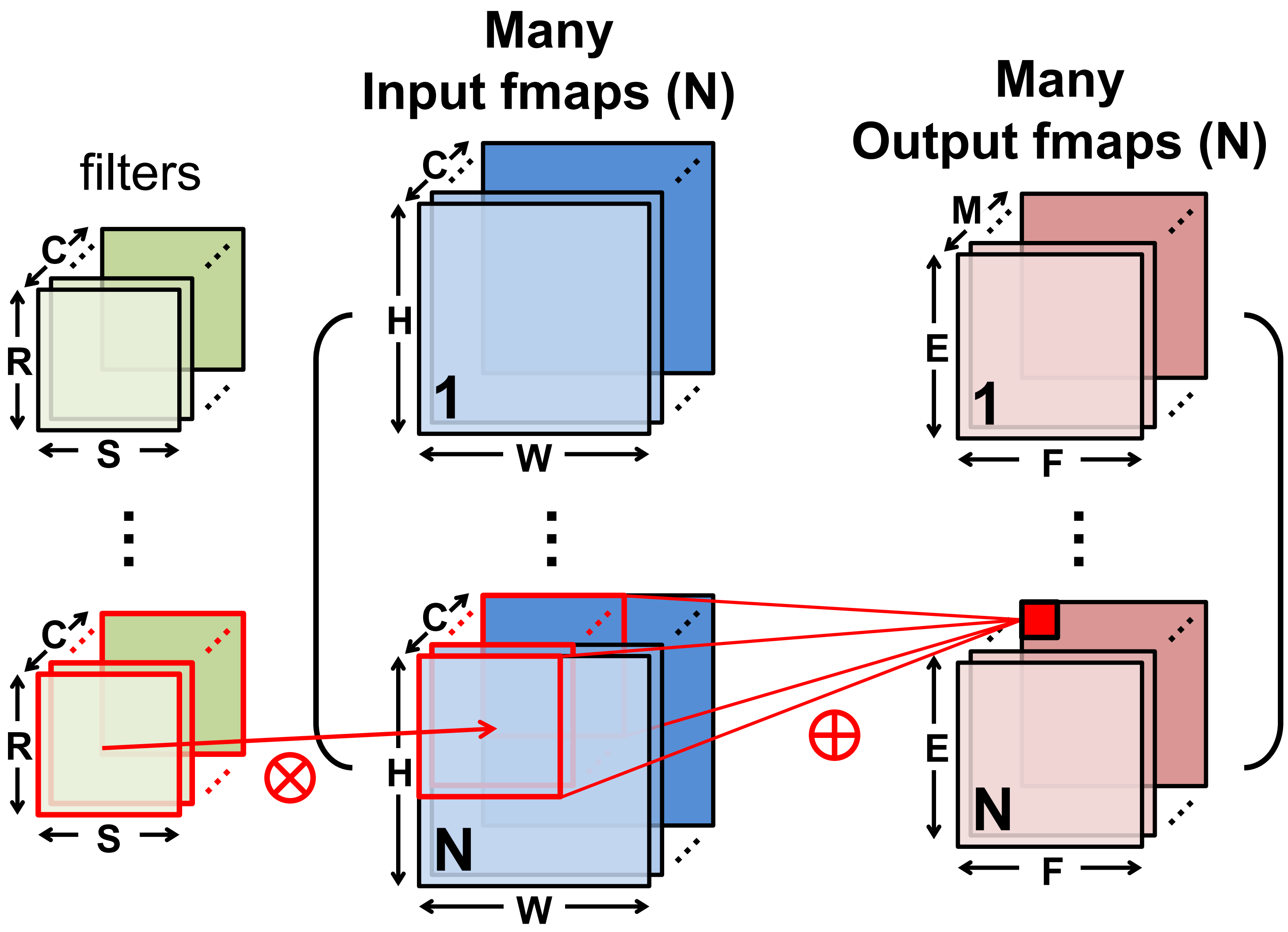


Flattened Fully-Connected Layer

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



Convolution 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++) {
                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]);
            }
        }
    }
}
```

for each output fmap value

Convolve a window and apply activation

Shape Parameter	Description
N	Number of **input fmaps/output fmaps** (batch size)
C	Number of 2-D **input fmaps /filters** (channels)
H	Height of **input fmap** (activations)
W	Width of **input fmap** (activations)
R	Height of 2-D **filter** (weights)
S	Width of 2-D **filter** (weights)
M	Number of 2-D **output fmaps** (channels)
E	Height of **output fmap** (activations)
F	Width of **output fmap** (activations)

How much temporal locality for naïve implementation? None

Convolution Layer

Filter		Input Feature map		Output Feature map
$\begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix}$	$*$	$\begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix}$	$=$	$\begin{bmatrix} O_{11} & O_{12} \\ O_{21} & O_{22} \end{bmatrix}$

Computation Steps

1 $\begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} \\ O_{21} & O_{22} \end{bmatrix} \Rightarrow \begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{11} \\ I_{12} \\ I_{21} \\ I_{22} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix}$

2 $\begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} \\ O_{21} & O_{22} \end{bmatrix} \Rightarrow \begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{12} \\ I_{13} \\ I_{22} \\ I_{23} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix}$

Convolution Layer

Filter, Input Feature map, Output Feature map

$$\begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} * \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} \\ O_{21} & O_{22} \end{bmatrix}$$

Computation Steps

3
$$\begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} \\ O_{21} & O_{22} \end{bmatrix} \Rightarrow \begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{21} \\ I_{22} \\ I_{31} \\ I_{32} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix}$$

4
$$\begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} \\ O_{21} & O_{22} \end{bmatrix} \Rightarrow \begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{22} \\ I_{23} \\ I_{32} \\ I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix}$$

Flattened Convolution Layer

Filter, Input Feature map, Output Feature map

$$\begin{bmatrix}F_{11} & F_{12}\\ F_{21} & F_{22}\end{bmatrix} * \begin{bmatrix}I_{11} & I_{12} & I_{13}\\ I_{21} & I_{22} & I_{23}\\ I_{31} & I_{32} & I_{33}\end{bmatrix} = \begin{bmatrix}O_{11} & O_{12}\\ O_{21} & O_{22}\end{bmatrix}$$

Decompose

$$\begin{bmatrix}F_{11} & F_{12} & F_{21} & F_{22}\end{bmatrix} \bullet \begin{bmatrix}I_{11}\\ I_{12}\\ I_{21}\\ I_{22}\end{bmatrix} = \begin{bmatrix}O_{11} & O_{12} & O_{21} & O_{22}\end{bmatrix}$$

$$\begin{bmatrix}F_{11} & F_{12} & F_{21} & F_{22}\end{bmatrix} \bullet \begin{bmatrix}I_{12}\\ I_{13}\\ I_{22}\\ I_{23}\end{bmatrix} = \begin{bmatrix}O_{11} & O_{12} & O_{21} & O_{22}\end{bmatrix}$$

$$\begin{bmatrix}F_{11} & F_{12} & F_{21} & F_{22}\end{bmatrix} \bullet \begin{bmatrix}I_{21}\\ I_{22}\\ I_{31}\\ I_{32}\end{bmatrix} = \begin{bmatrix}O_{11} & O_{12} & O_{21} & O_{22}\end{bmatrix}$$

$$\begin{bmatrix}F_{11} & F_{12} & F_{21} & F_{22}\end{bmatrix} \bullet \begin{bmatrix}I_{22}\\ I_{23}\\ I_{32}\\ I_{33}\end{bmatrix} = \begin{bmatrix}O_{11} & O_{12} & O_{21} & O_{22}\end{bmatrix}$$

Flattened Convolution Layer

$$\begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{11} \\ I_{12} \\ I_{21} \\ I_{22} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix} \qquad \begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{12} \\ I_{13} \\ I_{22} \\ I_{23} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix}$$

$$\begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{21} \\ I_{22} \\ I_{31} \\ I_{32} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix} \qquad \begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix} \bullet \begin{bmatrix} I_{22} \\ I_{23} \\ I_{32} \\ I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix}$$

Integrate

$$\begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix} \times \begin{bmatrix} I_{11} & I_{12} & I_{21} & I_{22} \\ I_{12} & I_{13} & I_{22} & I_{23} \\ I_{21} & I_{22} & I_{31} & I_{32} \\ I_{22} & I_{23} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix}$$

Matrix Multiply (Toeplitz Matrix))

Flattened Convolution Layer

Filter, Input Feature map, Output Feature map

$$\begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} * \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} \\ O_{21} & O_{22} \end{bmatrix}$$

Convolution

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

Matrix Multiply (Toeplitz Matrix))

$$\begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix} \times \begin{bmatrix} I_{11} & I_{12} & I_{21} & I_{22} \\ I_{12} & I_{13} & I_{22} & I_{23} \\ I_{21} & I_{22} & I_{31} & I_{32} \\ I_{22} & I_{23} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix}$$

Flattened Convolution Layer

Filter | Input Feature map | Output Feature map

$$\begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} * \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} \\ O_{21} & O_{22} \end{bmatrix}$$

Convolution

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

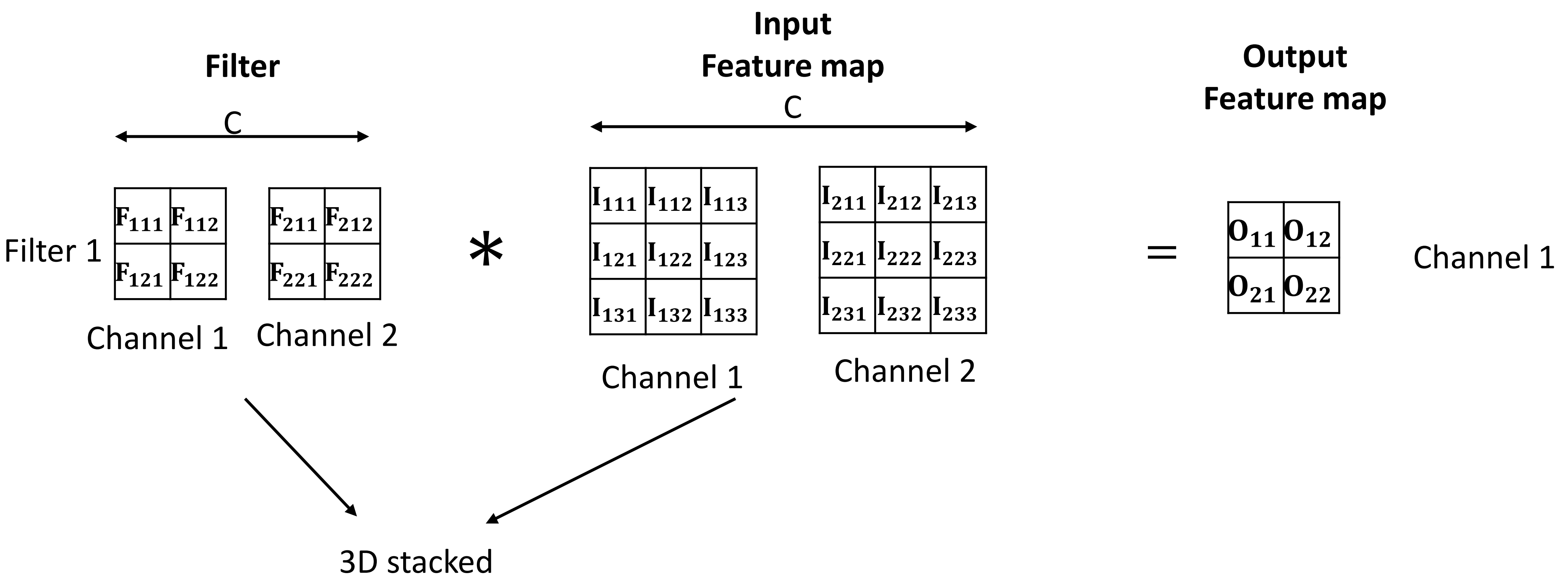
Matrix Multiply (Toeplitz Matrix))

$$\begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix} \times \begin{bmatrix} I_{11} & I_{12} & I_{21} & I_{22} \\ I_{12} & I_{13} & I_{22} & I_{23} \\ I_{21} & I_{22} & I_{31} & I_{32} \\ I_{22} & I_{23} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix}$$

Data is repeated(redundant data)

Flattened Convolution Layer

- Multiple Input Channels



Flattened Convolution Layer

- Multiple Input Channels

Input Feature map
as Toeplitz matrix
(w/ redundant data)

Filter

Output
Feature map

$$
\begin{array}{c}
\begin{array}{cc} \text{Channel 1} & \text{Channel 2} \end{array} \\
\text{Filter 1}\ \begin{bmatrix} F_{111} & F_{112} & F_{121} & F_{122} & F_{211} & F_{212} & F_{221} & F_{222} \end{bmatrix}
\end{array}
\times
\begin{array}{c}
\begin{bmatrix}
I_{111} & I_{112} & I_{121} & I_{122} \\
I_{112} & I_{113} & I_{122} & I_{123} \\
I_{121} & I_{122} & I_{131} & I_{132} \\
I_{122} & I_{123} & I_{132} & I_{133} \\
\hline
I_{211} & I_{212} & I_{221} & I_{222} \\
I_{212} & I_{213} & I_{222} & I_{223} \\
I_{221} & I_{222} & I_{231} & I_{232} \\
I_{222} & I_{223} & I_{232} & I_{233}
\end{bmatrix}
\end{array}
\begin{array}{l} \text{Channel 1} \\ \\ \\ \\ \text{Channel 2} \end{array}
=
\begin{bmatrix} O_{11} & O_{12} & O_{21} & O_{22} \end{bmatrix}\ \text{Channel 1}
$$

Toeplitz
converted inputs

Flattened Convolution Layer

- Multiple Output Channels

Filter (C)

Filter 1: $\begin{bmatrix} F_{1111} & F_{1112} \\ F_{1121} & F_{1122} \end{bmatrix}$ $\begin{bmatrix} F_{1211} & F_{1212} \\ F_{1221} & F_{1222} \end{bmatrix}$

Filter 2: $\begin{bmatrix} F_{2111} & F_{2112} \\ F_{2121} & F_{2122} \end{bmatrix}$ $\begin{bmatrix} F_{2211} & F_{2212} \\ F_{2221} & F_{2222} \end{bmatrix}$

Channel 1 Channel 2

$*$

Input Feature map (C)

$\begin{bmatrix} I_{111} & I_{112} & I_{113} \\ I_{121} & I_{122} & I_{123} \\ I_{131} & I_{132} & I_{133} \end{bmatrix}$ $\begin{bmatrix} I_{211} & I_{212} & I_{213} \\ I_{221} & I_{222} & I_{223} \\ I_{231} & I_{232} & I_{233} \end{bmatrix}$

Channel 1 Channel 2

$=$

Output Feature map

$\begin{bmatrix} O_{111} & O_{112} \\ O_{121} & O_{122} \end{bmatrix}$ Channel 1

$\begin{bmatrix} O_{211} & O_{212} \\ O_{221} & O_{222} \end{bmatrix}$ Channel 2

Flattened Convolution Layer

- Multiple Output Channels

Input Feature map
as Toeplitz matrix (w/ redundant data)

Filter

Output Feature map

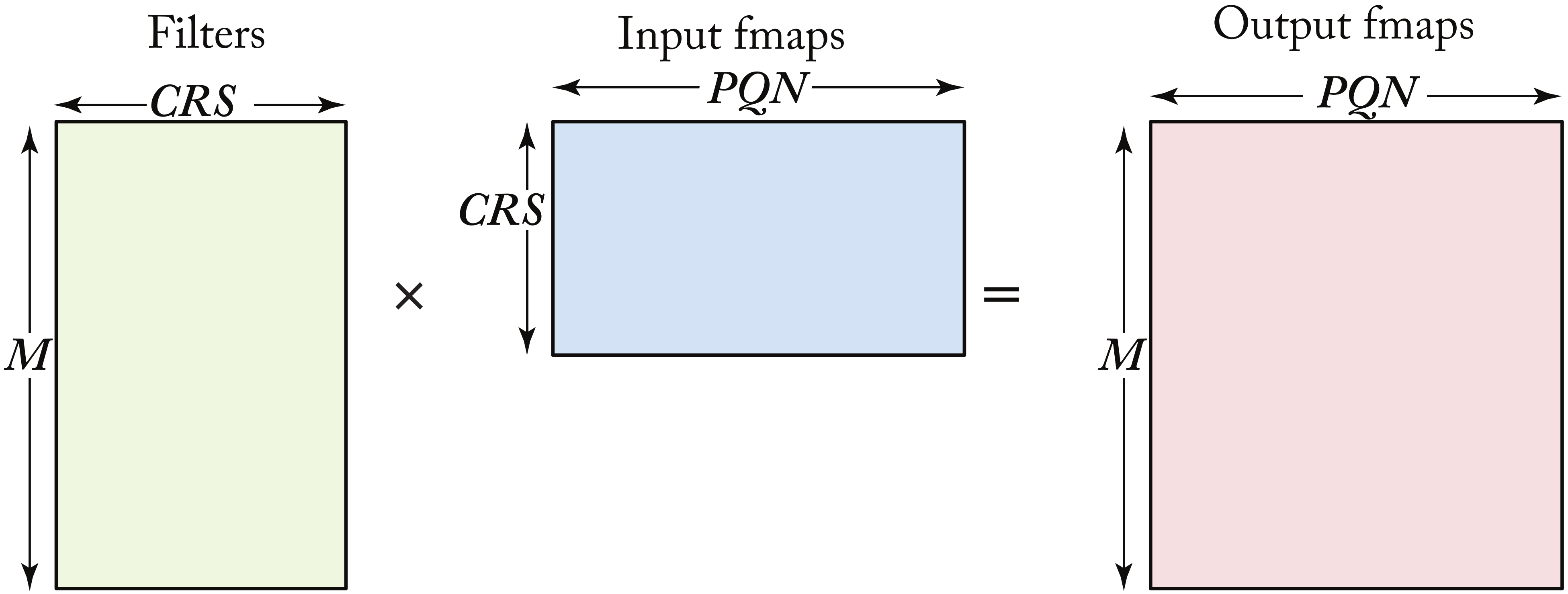
$$
\begin{array}{c|cccc|cccc|}
 & \multicolumn{4}{c}{\text{Channel 1}} & \multicolumn{4}{c}{\text{Channel 2}} \\
\text{Filter 1} & F_{1111} & F_{1112} & F_{1121} & F_{1122} & F_{1211} & F_{1212} & F_{1221} & F_{1222} \\
\text{Filter 2} & F_{2111} & F_{2112} & F_{2121} & F_{2122} & F_{2211} & F_{2212} & F_{2221} & F_{2222}
\end{array}
\times
\begin{array}{|cccc|l}
I_{111} & I_{112} & I_{121} & I_{122} & \\
I_{112} & I_{113} & I_{122} & I_{123} & \text{Channel 1} \\
I_{121} & I_{122} & I_{131} & I_{132} & \\
I_{122} & I_{123} & I_{132} & I_{133} & \\
\hline
I_{211} & I_{212} & I_{221} & I_{222} & \\
I_{212} & I_{213} & I_{222} & I_{223} & \\
I_{221} & I_{222} & I_{231} & I_{232} & \text{Channel 2} \\
I_{222} & I_{223} & I_{232} & I_{233} &
\end{array}
=
\begin{array}{|cccc|l}
O_{111} & O_{112} & O_{121} & O_{122} & \text{Channel 1} \\
\hline
O_{211} & O_{212} & O_{221} & O_{222} & \text{Channel 2}
\end{array}
$$

Toeplitz
converted inputs

Flattened Convolution Layer

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

$$P = \frac{H - R + U}{U}, Q = \frac{W - S + U}{U}$$



Weight Replication of CONV

- Since the filter size is typically much smaller than the size of the input feature map
  - We can also convert convolution into a matrix multiply by replicating the filter weights to correspond to the filter weight convolutional reuse
- Result in **sparse matrix**
  - Inefficiency in storage
  - Complex memory access pattern

Weight Replication of CONV

Filter
as Toeplitz matrix (w/ redundant data)

Input Feature map

Output Feature map

Channel 1 $\begin{bmatrix} I_{11} & I_{12} & I_{13} & I_{21} & I_{22} & I_{23} & I_{31} & I_{32} & I_{33} \end{bmatrix} \times \begin{bmatrix} F_{11} & 0 & 0 & 0 \\ F_{12} & F_{11} & 0 & 0 \\ 0 & F_{12} & 0 & 0 \\ F_{21} & 0 & F_{11} & 0 \\ F_{22} & F_{21} & F_{12} & F_{11} \\ 0 & F_{22} & 0 & F_{12} \\ 0 & 0 & F_{21} & 0 \\ 0 & 0 & F_{22} & F_{21} \\ 0 & 0 & 0 & F_{22} \end{bmatrix} = \begin{bmatrix} O_{11} & O_{21} & O_{21} & O_{22} \end{bmatrix}$ Channel 1

Toeplitz
converted inputs

Weight Replication vs. Input Replication

$$
\begin{bmatrix} I_{11} & I_{12} & I_{13} & I_{21} & I_{22} & I_{23} & I_{31} & I_{32} & I_{33} \end{bmatrix}
\times
\begin{bmatrix}
F_{11} & 0 & 0 & 0 \\
F_{12} & F_{11} & 0 & 0 \\
0 & F_{12} & 0 & 0 \\
F_{21} & 0 & F_{11} & 0 \\
F_{22} & F_{21} & F_{12} & F_{11} \\
0 & F_{22} & 0 & F_{12} \\
0 & 0 & F_{21} & 0 \\
0 & 0 & F_{22} & F_{21} \\
0 & 0 & 0 & F_{22}
\end{bmatrix}
$$

Weight Replication

- Required Memory$= 9 + 9 \times 4 = 45$

$$
\begin{bmatrix} F_{11} & F_{12} & F_{21} & F_{22} \end{bmatrix}
\times
\begin{bmatrix}
I_{11} & I_{12} & I_{21} & I_{22} \\
I_{12} & I_{13} & I_{22} & I_{23} \\
I_{21} & I_{22} & I_{31} & I_{32} \\
I_{22} & I_{23} & I_{32} & I_{33}
\end{bmatrix}
$$

Input Replication

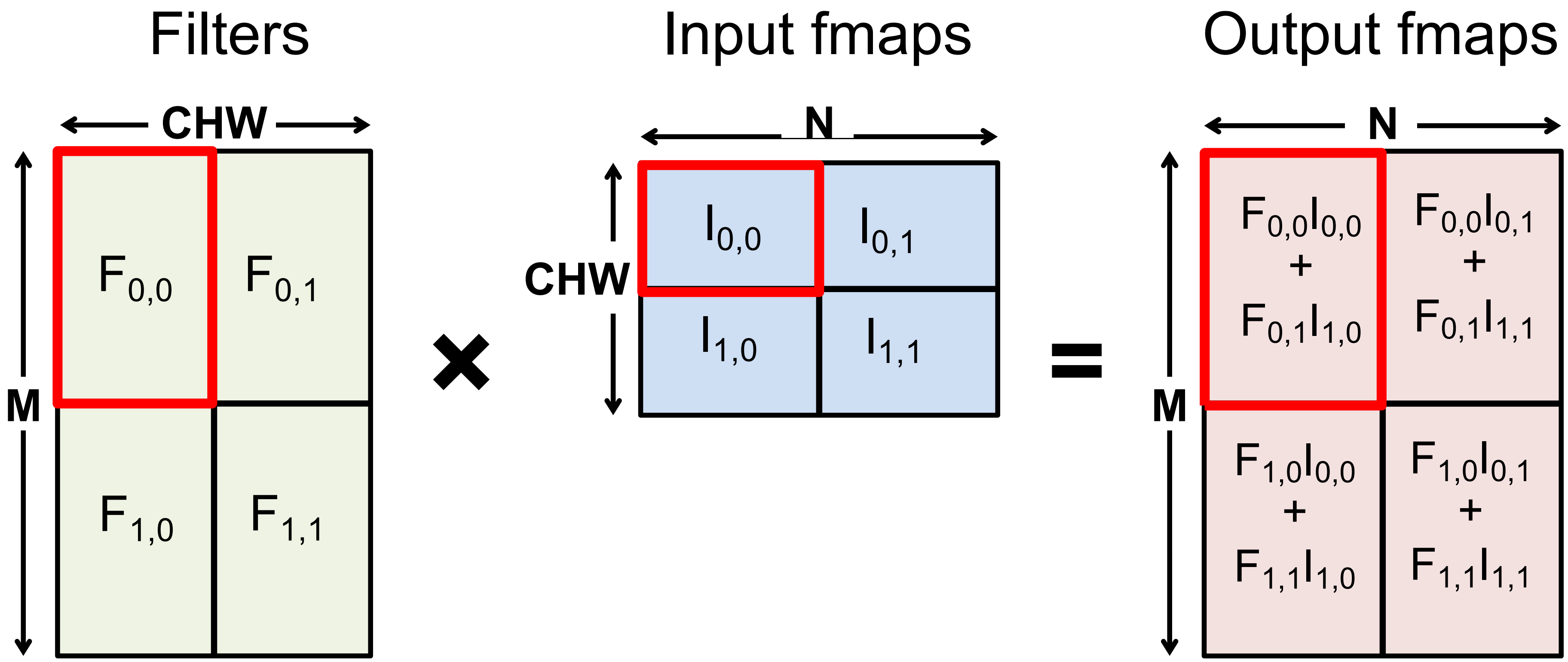
- Required Memory$= 4 + 4 \times 4 = 20$

Outline

- Overview
- Matrix Multiplication Based Convolution
- Tiling for Optimizing Performance
- Computation Transform Optimizations

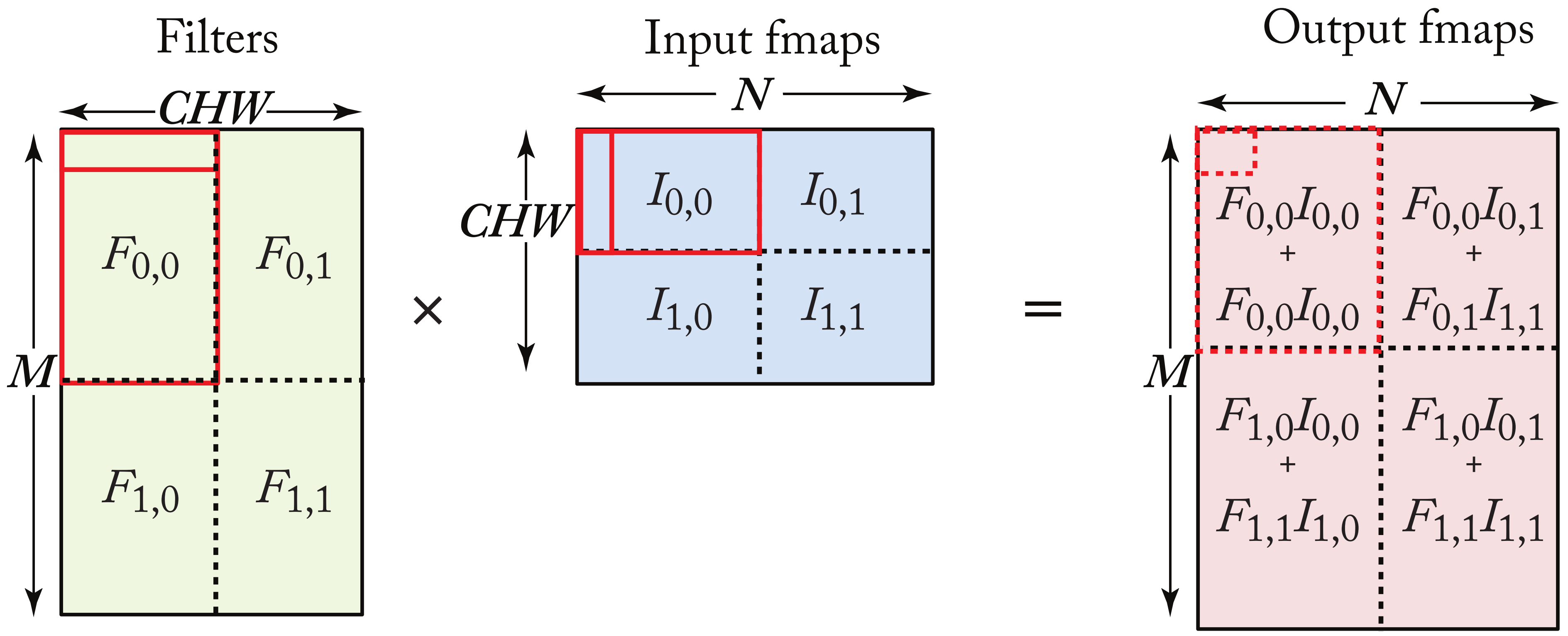
Tiled Computation of Matrix Multiplication

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



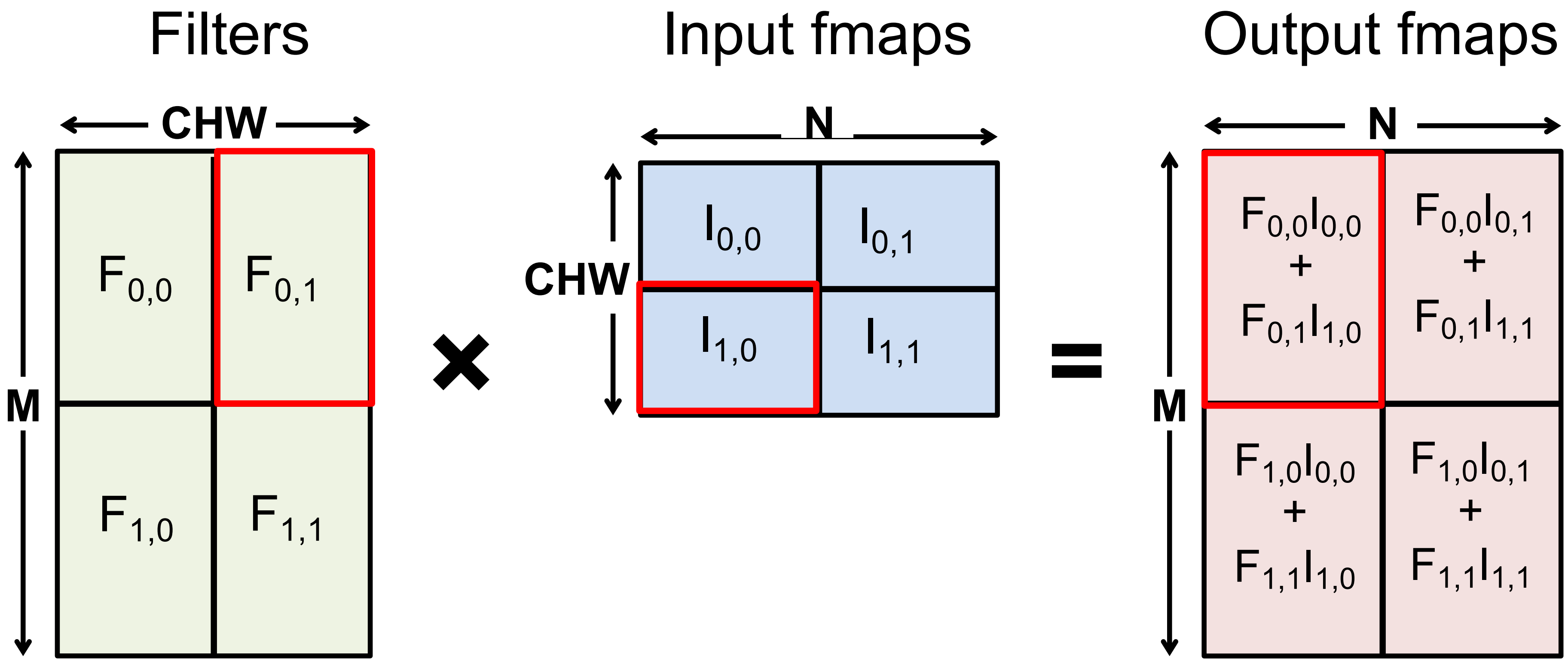
Tiled Computation of Matrix Multiplication

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



Tiled Computation of Matrix Multiplication

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



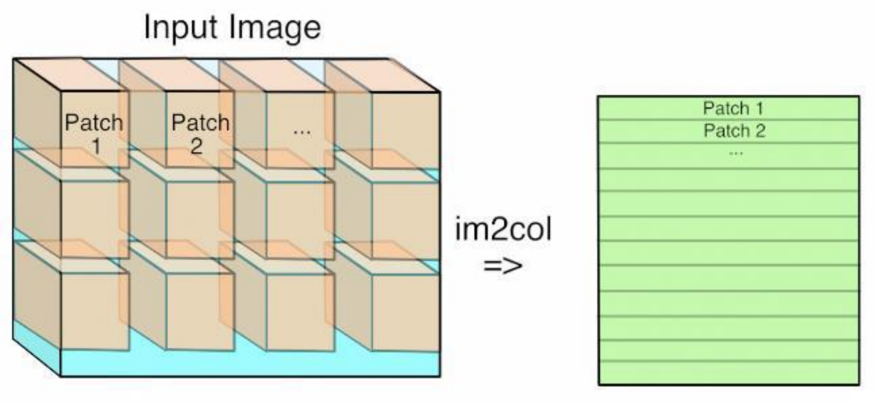
Matrix Multiplication

- 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
- Attempt to maximize reuse of the values held in the smaller, faster, and more energy-efficient memories

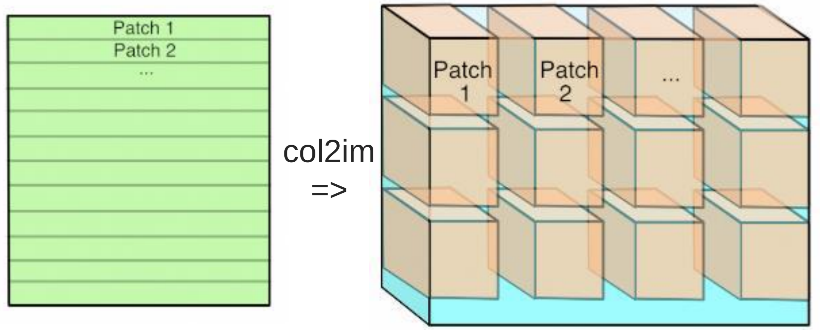
GEMM Based Convolution

- Flatten input data and kernels, solve the convolution as a matrix multiplication problem

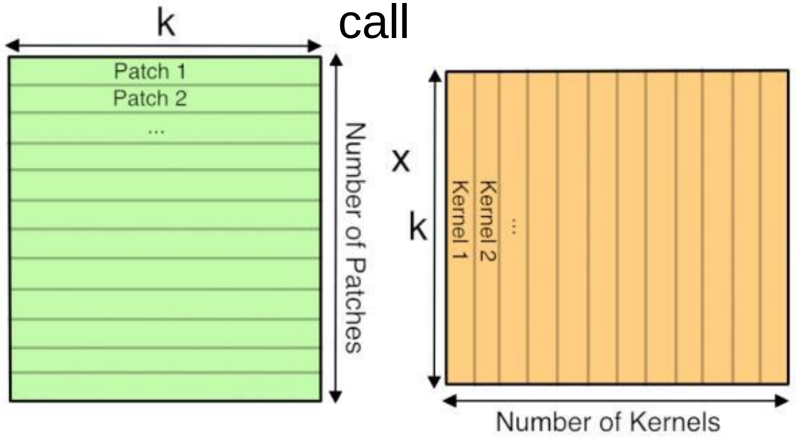
Step1: data flattening



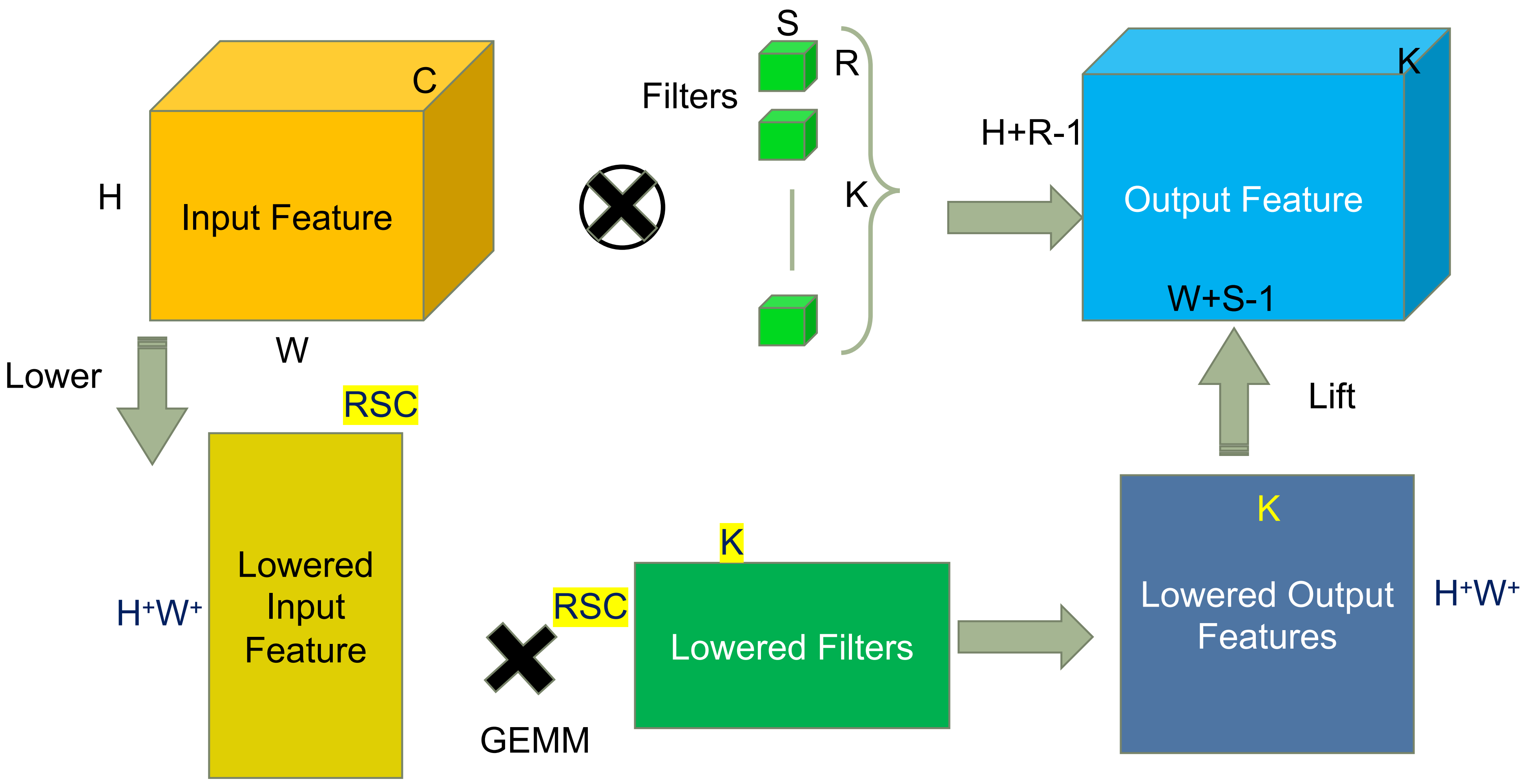
Step3: data unflattening



Step2: matrix multiply
Usually mapped into a BLAS (Basic Linear Algebra Subprogram) call

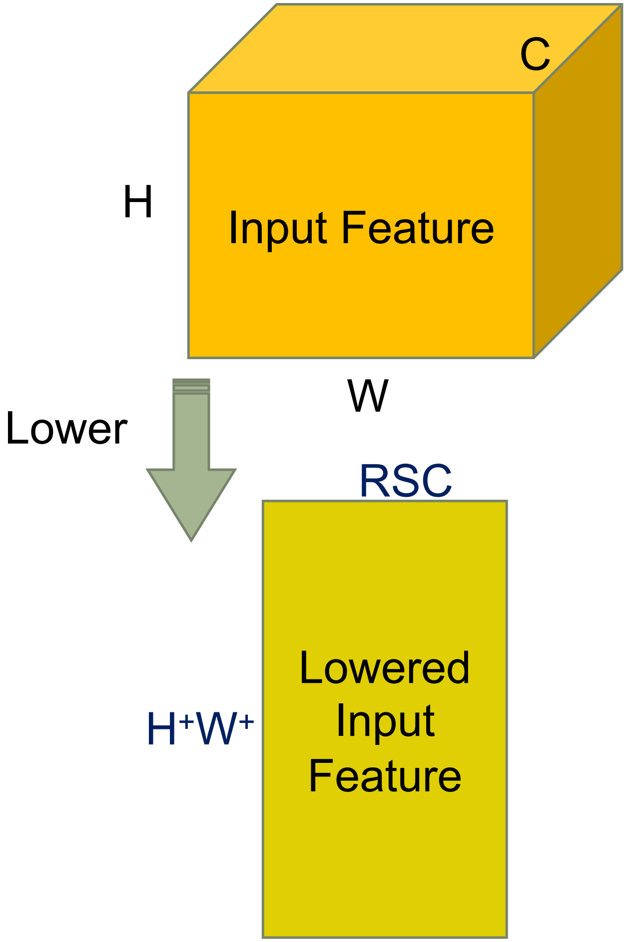


Im2Col



C
H
Input Feature
W
Filters
S
R
K
H+R-1
K
Output Feature
W+S-1
Lower
RSC
H⁺W⁺
Lowered Input Feature
RSC
K
Lowered Filters
GEMM
Lift
K
Lowered Output Features
H⁺W⁺

What Really Happened?



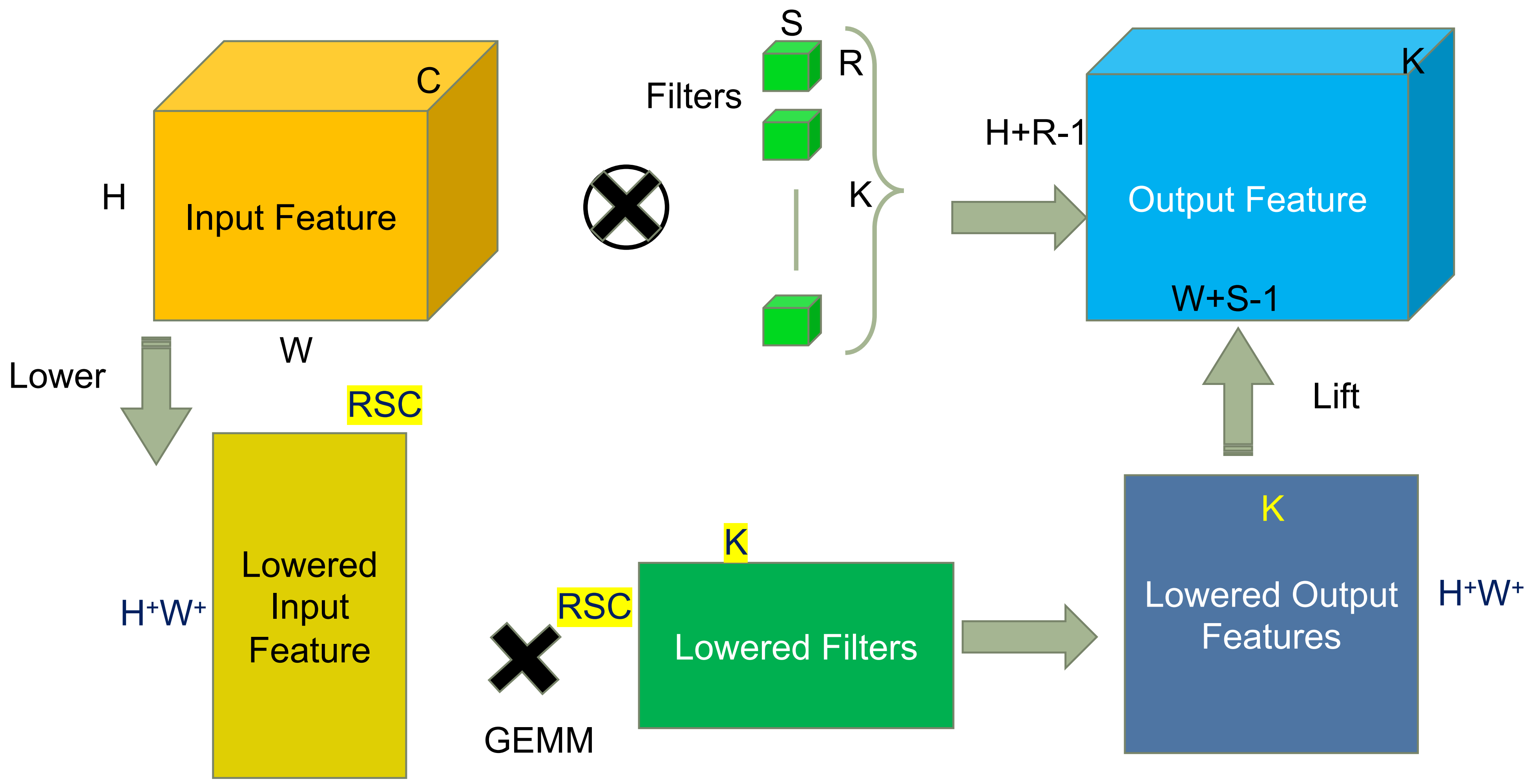
What was the tensor size?

What is lowered matrix means?

What does lowering mean?

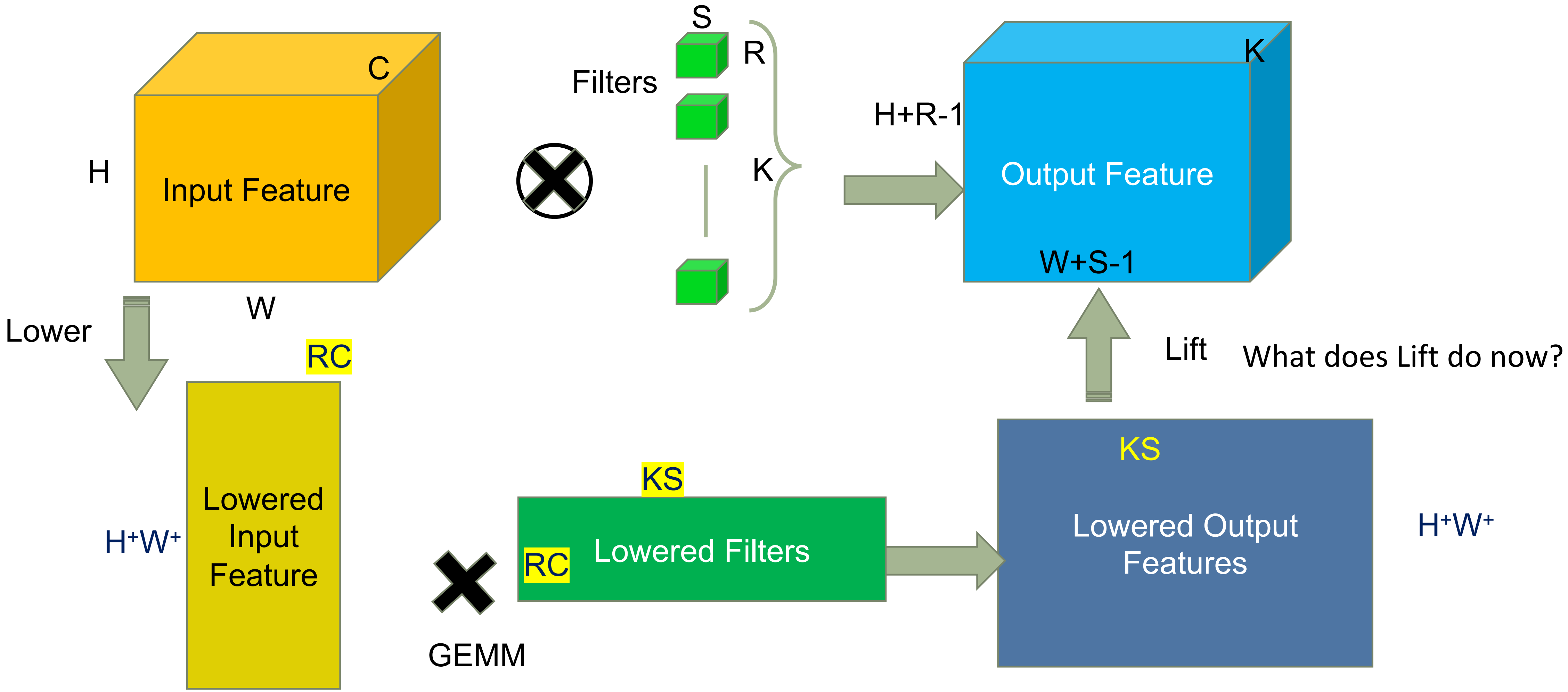
What is the impact on Memory?

Im2Col



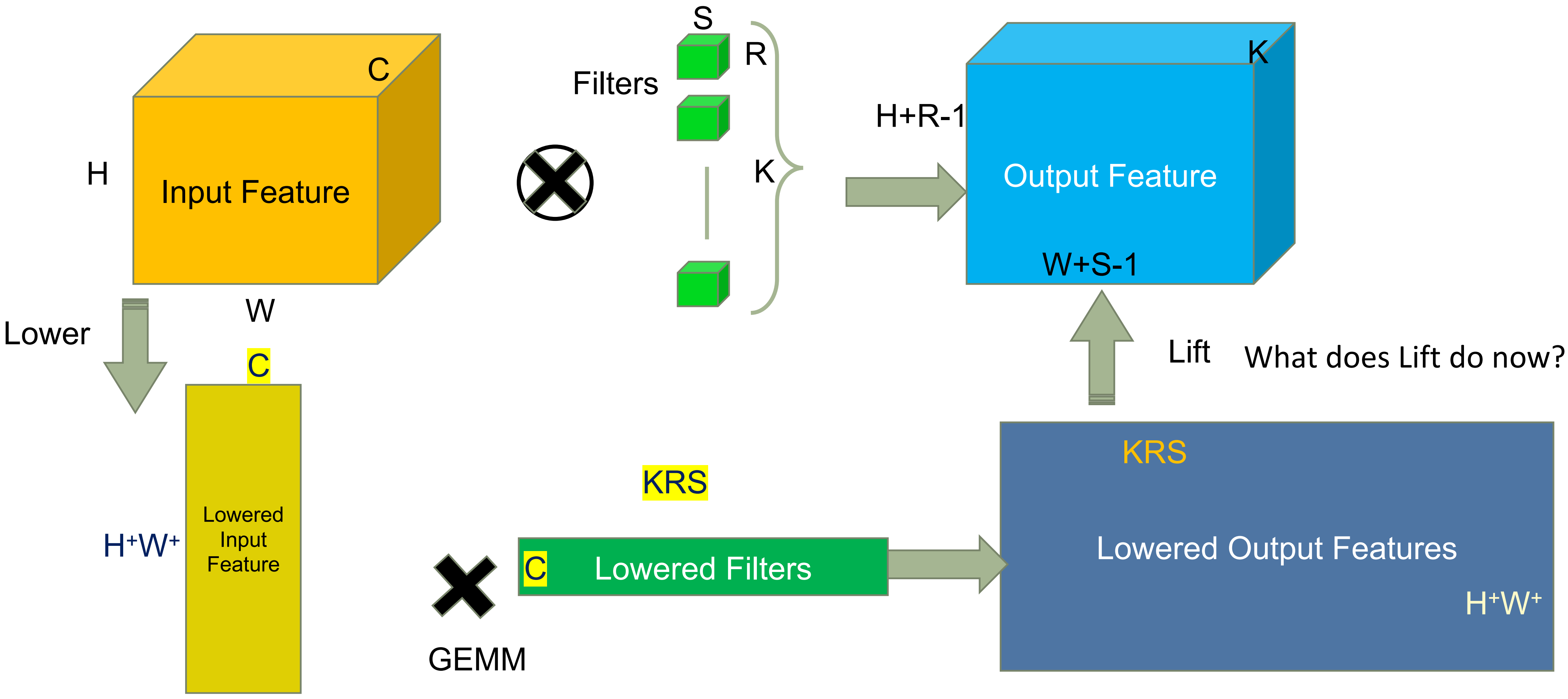
C
H
Input Feature
W
Filters
S
R
K
H+R-1
K
Output Feature
W+S-1
Lower
RSC
Lowered
Input
Feature
H+W+
K
RSC
Lowered Filters
GEMM
Lift
K
Lowered Output
Features
H+W+

Do We Have Other Options?



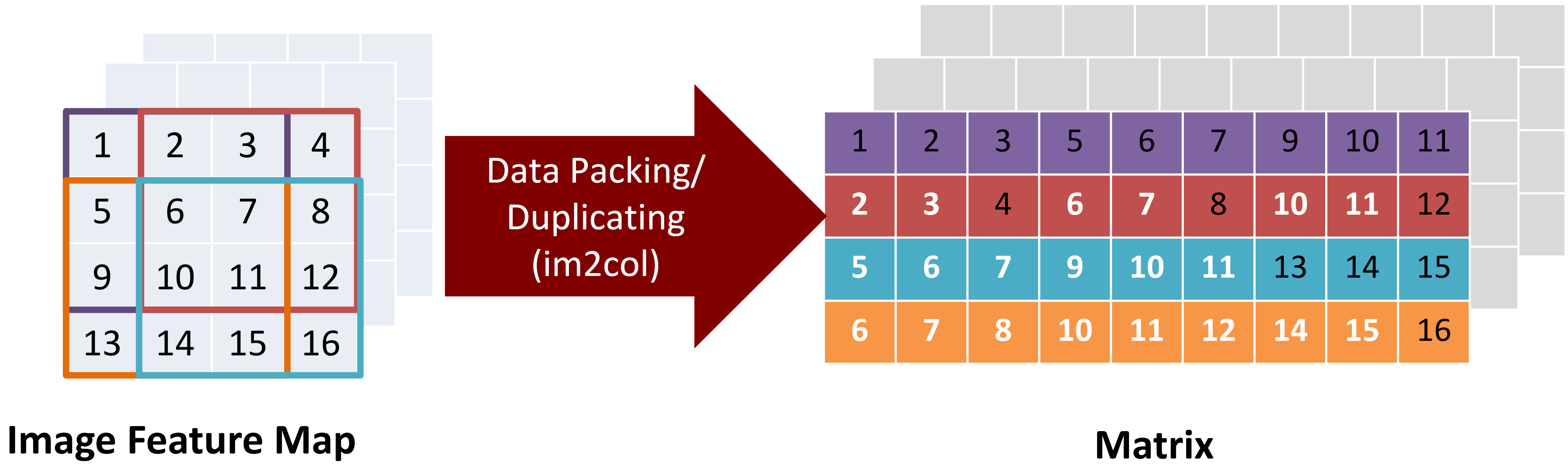
13

Do We Have Other Options?



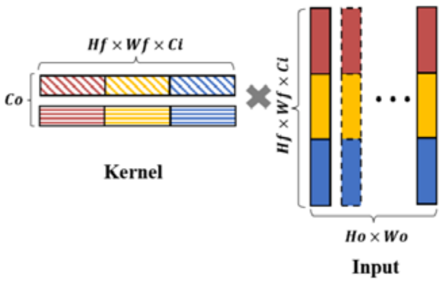
Conventional Approach Penalizes Memory To Optimize Performance

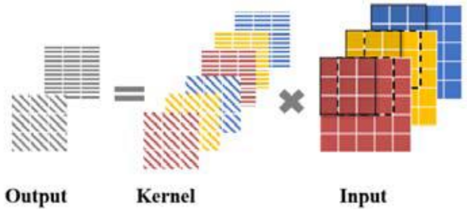
- Casts convolution to Matrix-Matrix-Multiplication, in order to utilize high performance MMM implementations in **Basic-Linear-Algebra-Subroutines(BLAS)** libraries



Direct Convolution is Better

- Higher performance, zero memory overheads





	Matrix-Matrix-Multiplication	Direct Convolution
Packing	**Yes** • Over 10x additional memory for image data • Performance penalty	**No** • Zero memory overheads • No performance penalty
Computation Performance	**Less than** expected theoretic peak of GEMM	Close to system’s theoretic peak

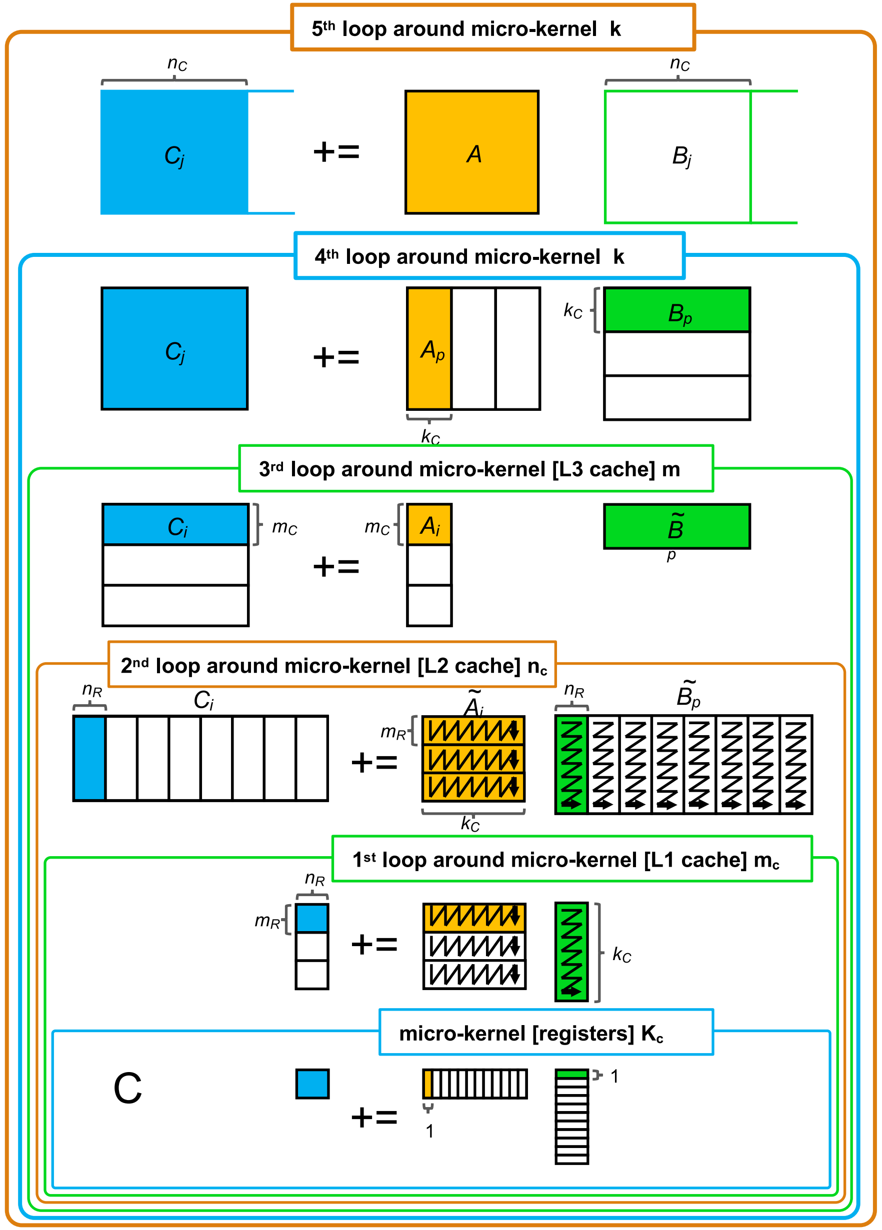
GEMM Loops

- Rules for each new character
  - Buffers
  - Re-fetch rate
- m_r n_r k_c m_c n_c m k n
- m_0 n_0 k_0 m_1 n_1 m_2 k_1 n_2

for $j_c = 0, \ldots, n-1$ **in steps of** n_c
  for $p_c = 0, \ldots, k-1$ **in steps of** k_c
    $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$
    for $i_c = 0, \ldots, m-1$ **in steps of** m_c
      $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$

      for $j_r = 0, \ldots, n_c - 1$ **in steps of** n_r
        for $i_r = 0, \ldots, m_c - 1$ **in steps of** m_r

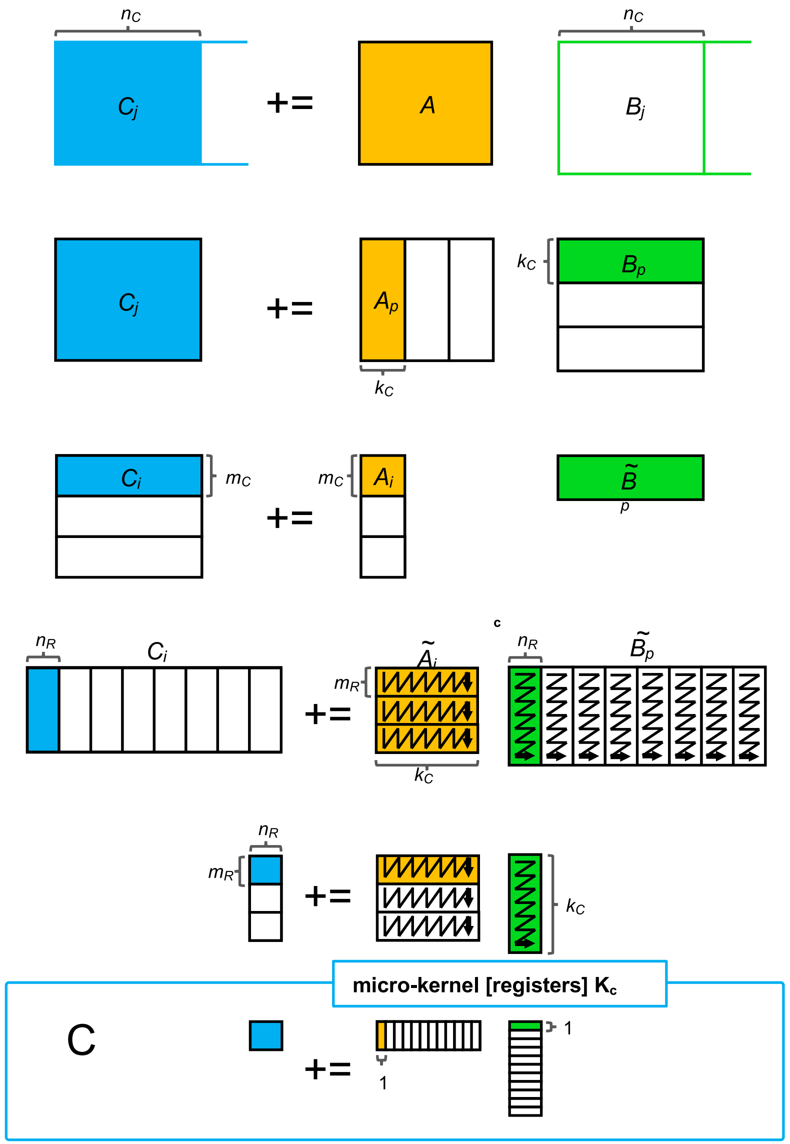
          for $p_r = 0, \ldots, k_c - 1$ **in steps of** 1
            $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$
              $\mathrel{+}= A_c(i_r : i_r + m_r - 1, p_r)$
              $\cdot\ B_c(p_r, j_r : j_r + n_r - 1)$



GEMM Loops

- Rules for each new character
  - Buffers
  - Re-fetch rate
- m_r n_r k_c
- m_0 n_0 k_0

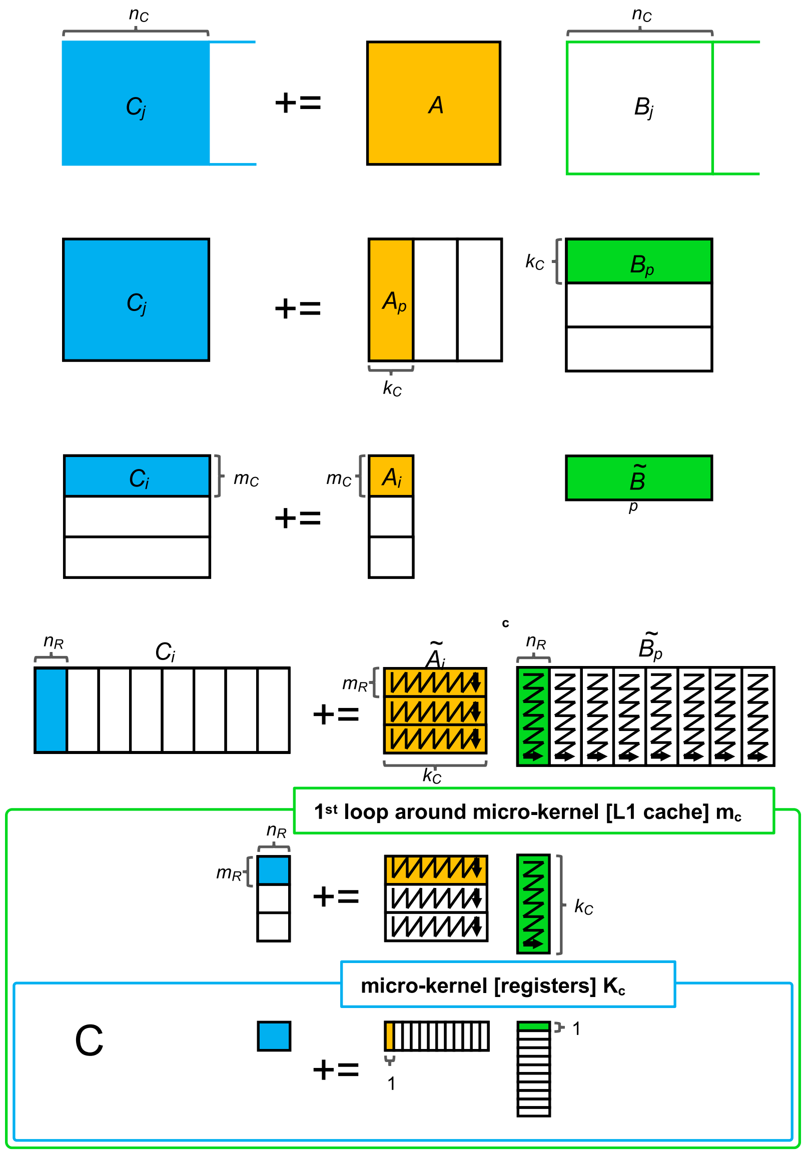
$$
\begin{array}{l}
\textbf{for } j_c = 0, \ldots, n-1 \textbf{ in steps of } n_c \\
\quad \textbf{for } p_c = 0, \ldots, k-1 \textbf{ in steps of } k_c \\
\quad\quad B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \to B_c \\
\quad\quad \textbf{for } i_c = 0, \ldots, m-1 \textbf{ in steps of } m_c \\
\quad\quad\quad A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \to A_c \\
\hline
\quad\quad\quad \textbf{for } j_r = 0, \ldots, n_c - 1 \textbf{ in steps of } n_r \\
\quad\quad\quad\quad \textbf{for } i_r = 0, \ldots, m_c - 1 \textbf{ in steps of } m_r \\
\quad\quad\quad\quad\quad \textbf{for } p_r = 0, \ldots, k_c - 1 \textbf{ in steps of } 1 \\
\quad\quad\quad\quad\quad\quad C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \\
\quad\quad\quad\quad\quad\quad\quad += A_c(i_r : i_r + m_r - 1, p_r) \\
\quad\quad\quad\quad\quad\quad\quad\quad \cdot \; B_c(p_r, j_r : j_r + n_r - 1)
\end{array}
$$



GEMM Loops

- Rules for each new character
  - Buffers
  - Re-fetch rate
- m_r n_r k_c m_c
- m_0 n_0 k_0 m_1

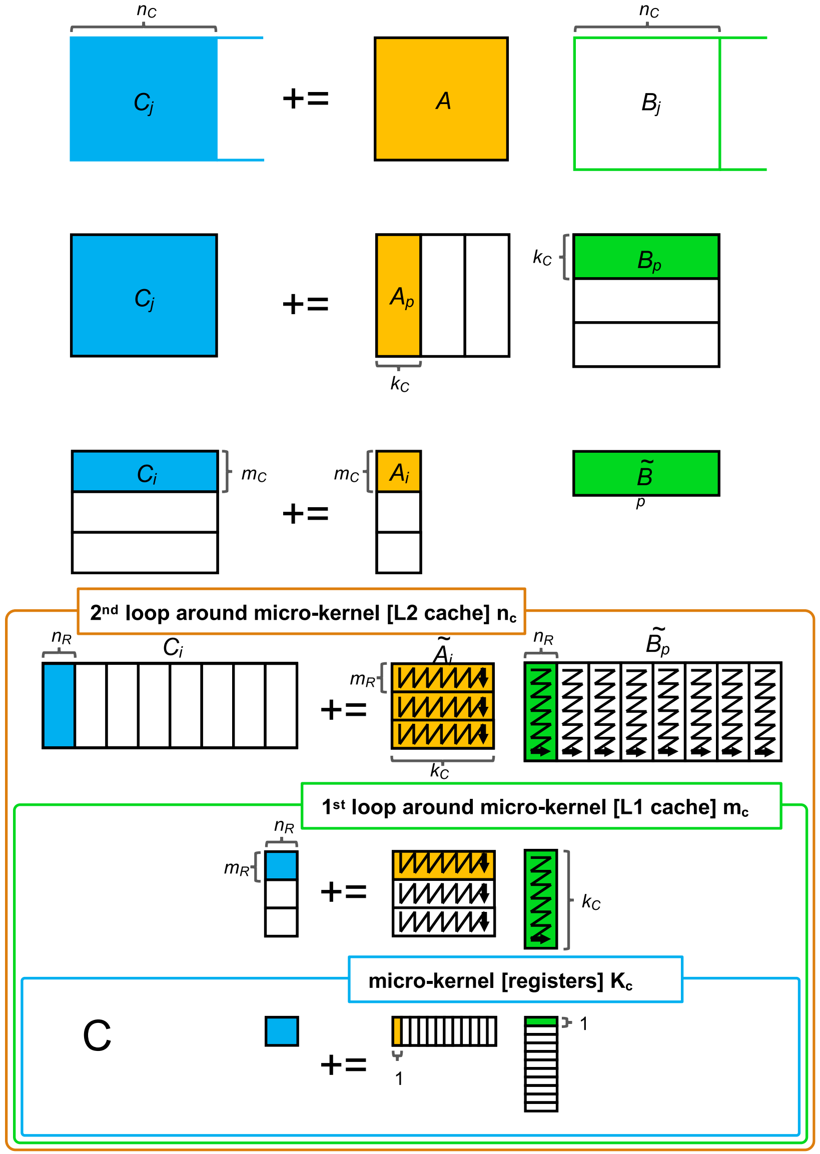
$$
\begin{array}{l}
\textbf{for } j_c = 0, \ldots, n-1 \textbf{ in steps of } n_c \\
\quad \textbf{for } p_c = 0, \ldots, k-1 \textbf{ in steps of } k_c \\
\quad\quad B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c \\
\quad\quad \textbf{for } i_c = 0, \ldots, m-1 \textbf{ in steps of } m_c \\
\quad\quad\quad A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c \\
\quad\quad\quad \textbf{for } j_r = 0, \ldots, n_c - 1 \textbf{ in steps of } n_r \\
\quad\quad\quad\quad \textbf{for } i_r = 0, \ldots, m_c - 1 \textbf{ in steps of } m_r \\
\quad\quad\quad\quad\quad \textbf{for } p_r = 0, \ldots, k_c - 1 \textbf{ in steps of } 1 \\
\quad\quad\quad\quad\quad\quad C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \\
\quad\quad\quad\quad\quad\quad\quad \mathrel{+}= A_c(i_r : i_r + m_r - 1, p_r) \\
\quad\quad\quad\quad\quad\quad\quad\quad \cdot \; B_c(p_r, j_r : j_r + n_r - 1)
\end{array}
$$



GEMM Loops

- Rules for each new character
  - Buffers
  - Re-fetch rate
- m_r n_r k_c m_c n_c
- m_0 n_0 k_0 m_1 n_1

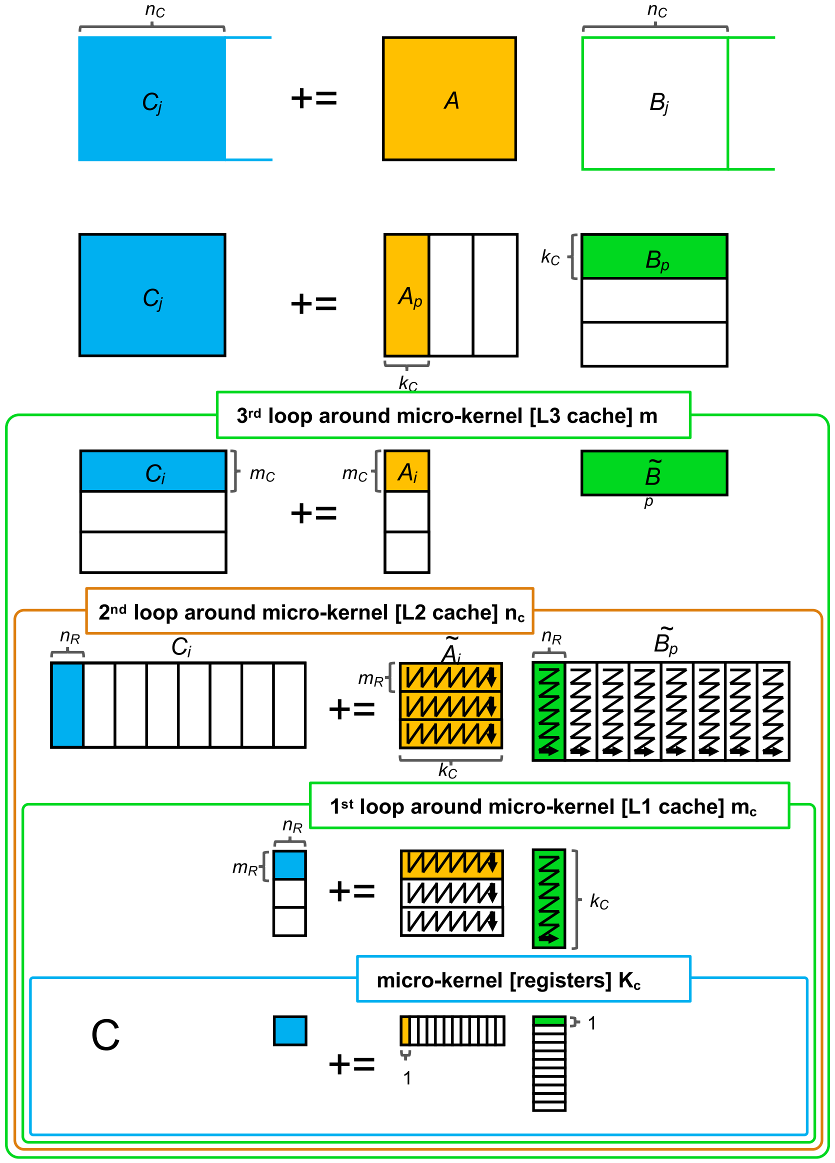
for $j_c = 0, \ldots, n-1$ **in steps of** n_c
  for $p_c = 0, \ldots, k-1$ **in steps of** k_c
    $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$
    for $i_c = 0, \ldots, m-1$ **in steps of** m_c
      $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$
        for $j_r = 0, \ldots, n_c - 1$ **in steps of** n_r
          for $i_r = 0, \ldots, m_c - 1$ **in steps of** m_r
            for $p_r = 0, \ldots, k_c - 1$ **in steps of** 1
              $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$
              $\mathrel{+}= A_c(i_r : i_r + m_r - 1, p_r)$
              $\cdot \; B_c(p_r, j_r : j_r + n_r - 1)$



GEMM Loops

- Rules for each new character
  - Buffers
  - Re-fetch rate
- m_r n_r k_c m_c n_c m
- m_0 n_0 k_0 m_1 n_1 m_2

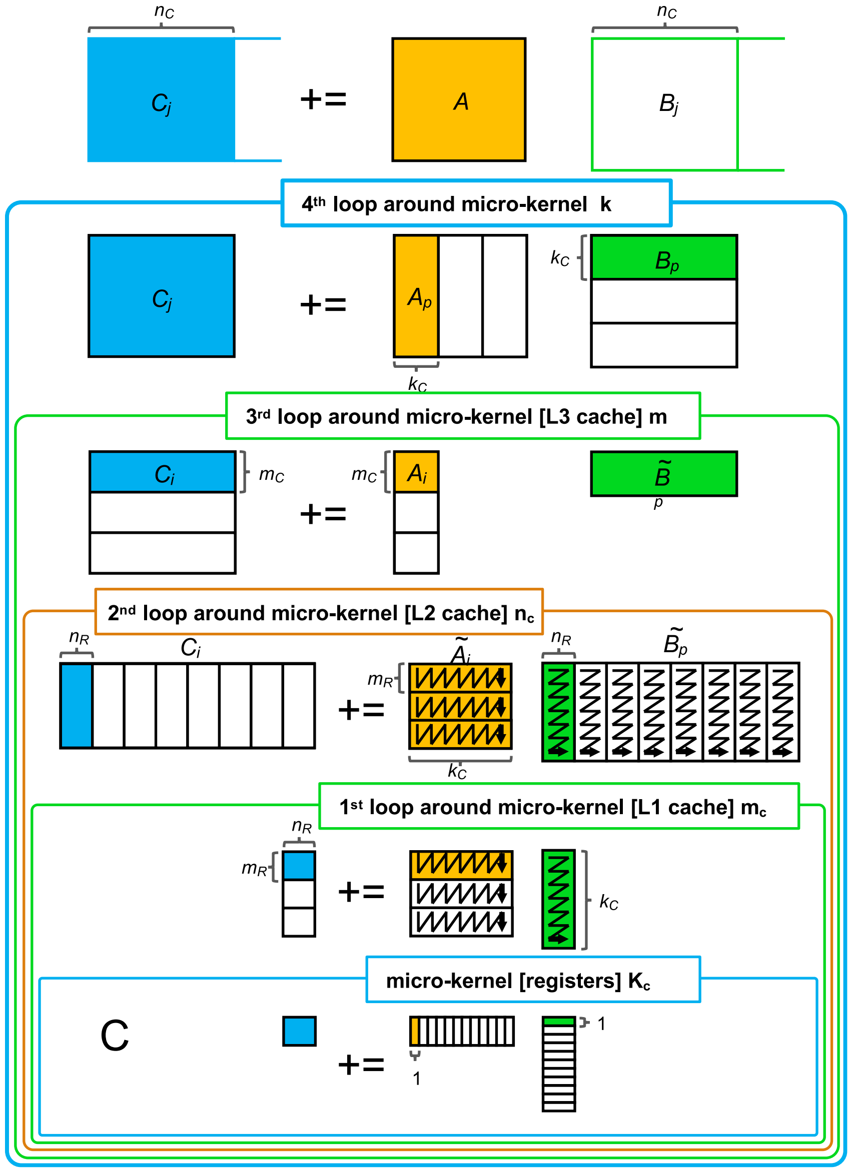
for $j_c = 0, \ldots, n-1$ **in steps of** n_c
  for $p_c = 0, \ldots, k-1$ **in steps of** k_c
    $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$
    for $i_c = 0, \ldots, m-1$ **in steps of** m_c
      $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$
      for $j_r = 0, \ldots, n_c - 1$ **in steps of** n_r
        for $i_r = 0, \ldots, m_c - 1$ **in steps of** m_r
          for $p_r = 0, \ldots, k_c - 1$ **in steps of** 1
            $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$
              $\mathrel{+}= A_c(i_r : i_r + m_r - 1, p_r)$
              $\cdot\ B_c(p_r, j_r : j_r + n_r - 1)$



GEMM Loops

- Rules for each new character
  - Buffers
  - Re-fetch rate
- m_r n_r k_c m_c n_c m k
- m_0 n_0 k_0 m_1 n_1 m_2 k_1

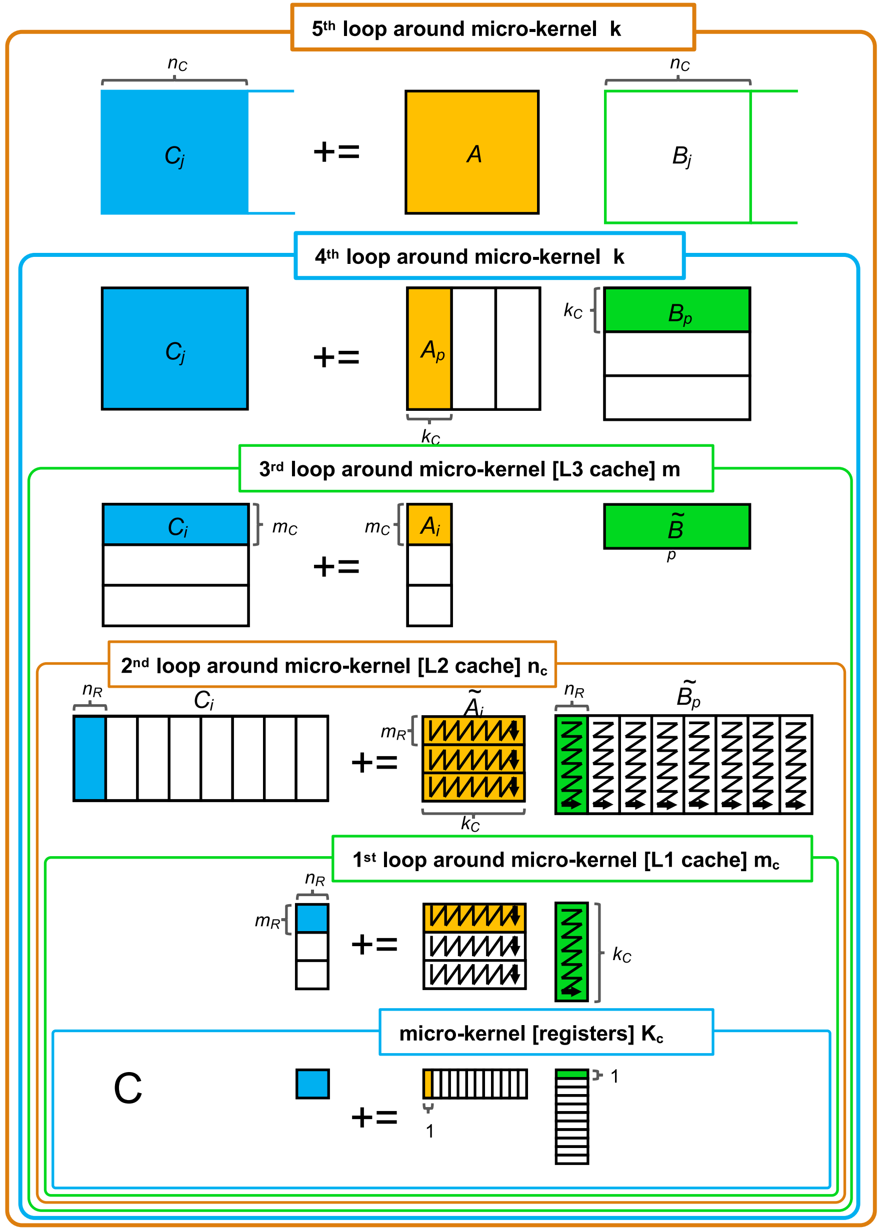
$$
\begin{array}{l}
\textbf{for } j_c = 0, \ldots, n-1 \textbf{ in steps of } n_c \\
\quad \textbf{for } p_c = 0, \ldots, k-1 \textbf{ in steps of } k_c \\
\quad\quad B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \to B_c \\
\quad\quad \textbf{for } i_c = 0, \ldots, m-1 \textbf{ in steps of } m_c \\
\quad\quad\quad A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \to A_c \\
\hline
\quad\quad\quad \textbf{for } j_r = 0, \ldots, n_c - 1 \textbf{ in steps of } n_r \\
\quad\quad\quad\quad \textbf{for } i_r = 0, \ldots, m_c - 1 \textbf{ in steps of } m_r \\
\hline
\quad\quad\quad\quad\quad \textbf{for } p_r = 0, \ldots, k_c - 1 \textbf{ in steps of } 1 \\
\quad\quad\quad\quad\quad\quad C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \\
\quad\quad\quad\quad\quad\quad\quad \mathrel{+}= A_c(i_r : i_r + m_r - 1, p_r) \\
\quad\quad\quad\quad\quad\quad\quad \cdot\ B_c(p_r, j_r : j_r + n_r - 1)
\end{array}
$$



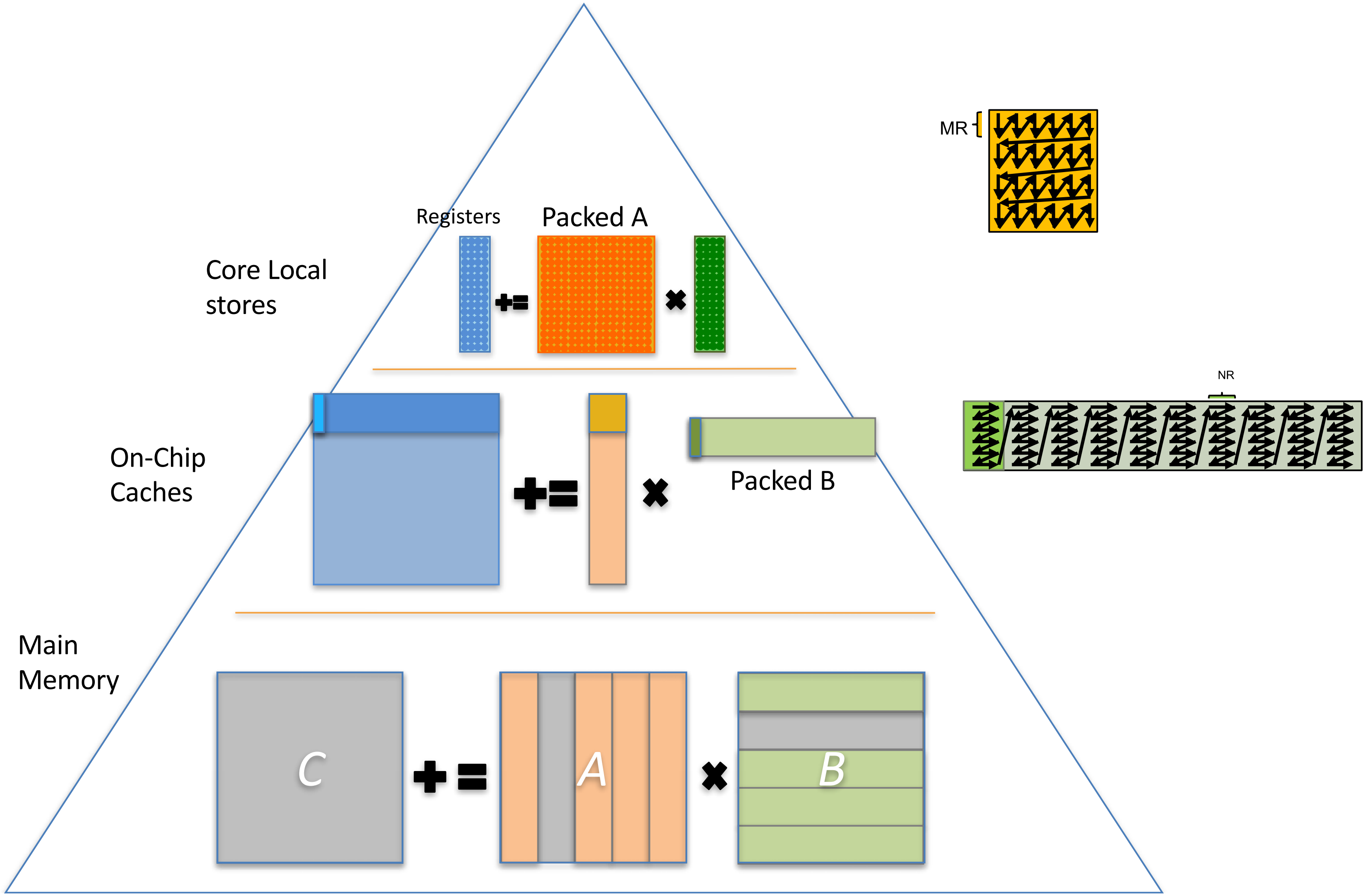
GEMM Loops

- Rules for each new character
  - Buffers
  - Re-fetch rate
- m_r n_r k_c m_c n_c m k n
- m_0 n_0 k_0 m_1 n_1 m_2 k_1 n_2

$$
\begin{array}{l}
\textbf{for } j_c = 0, \ldots, n-1 \textbf{ in steps of } n_c \\
\quad \textbf{for } p_c = 0, \ldots, k-1 \textbf{ in steps of } k_c \\
\quad\quad B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \to B_c \\
\quad\quad \textbf{for } i_c = 0, \ldots, m-1 \textbf{ in steps of } m_c \\
\quad\quad\quad A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \to A_c \\
\hline
\quad\quad\quad \textbf{for } j_r = 0, \ldots, n_c - 1 \textbf{ in steps of } n_r \\
\quad\quad\quad\quad \textbf{for } i_r = 0, \ldots, m_c - 1 \textbf{ in steps of } m_r \\
\hline
\quad\quad\quad\quad\quad \textbf{for } p_r = 0, \ldots, k_c - 1 \textbf{ in steps of } 1 \\
\quad\quad\quad\quad\quad\quad C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \\
\quad\quad\quad\quad\quad\quad\quad \mathrel{+}= A_c(i_r : i_r + m_r - 1, p_r) \\
\quad\quad\quad\quad\quad\quad\quad\quad \cdot\ B_c(p_r, j_r : j_r + n_r - 1)
\end{array}
$$



Memory Hierarchy with GEMM



Tiling For Optimizing Performance

- Improve efficiency at each level of the memory hierarchy
- Tiling can also be applied to parallelize the computation across multiple CPUs or the many threads of a GPU
- Must consider replacement policy of associative caches to achieve optimal performance
  - Least Recently Used(LRU)
  - Dynamic Re-Reference Interval Prediction (DRRIP)
- GEMM library will dynamically select and run the most appropriate implementation
  - Layer shape, hardware platform, etc.

Tiling For Optimizing Performance

- Compilers optimize a user-written program for tiling
- Creating a polyhedral model of the computation and using a Boolean satisfiability (SAT) solver to optimally tile and schedule the program. [1]
- Decouples the basic expression of the algorithm from user-provided annotations that describe the desired scheduling and tiling of the algorithm. [2]
- GCC, LLVM

[1] Lam, M. D., Rothberg, E. E., & Wolf, M. E. (1991). The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, *25*(Special Issue), 63-74.
[2] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, *48*(6), 519-530.

Compiler for DNN Hardwares

- TVM – Compiler for DNN hardware
  - Graph-level and operator-level optimizations for DNN workloads across diverse hardware back-ends
  - Tiling for hiding memory latency
  - High-level operator fusion
    - E.g. performing a CONV layer and ReLU together with one pass through memory
  - Mapping to arbitrary hardware primitives
- Maximize data reuse
  - in the memory hierarchy
  - in parallel computation units

Outline

- Overview
- Matrix Multiplication Based Convolution
- Tiling for Optimizing Performance
- Computation Transform Optimizations

Computation Transformations

- Goal
  - Bitwise same result, but reduce number of operations
    - In DNNs, reduce number of multiplications
  - Improve performance
  - Reduce energy consumption
- Focuses mostly on compute
- May come at the cost of
  - More intermediate results
  - Increased number of additions
  - More irregular data access pattern

Gauss's Multiplication Algorithm

- Complex multiplication

$$(ac - bd) + (bc + ad)i$$

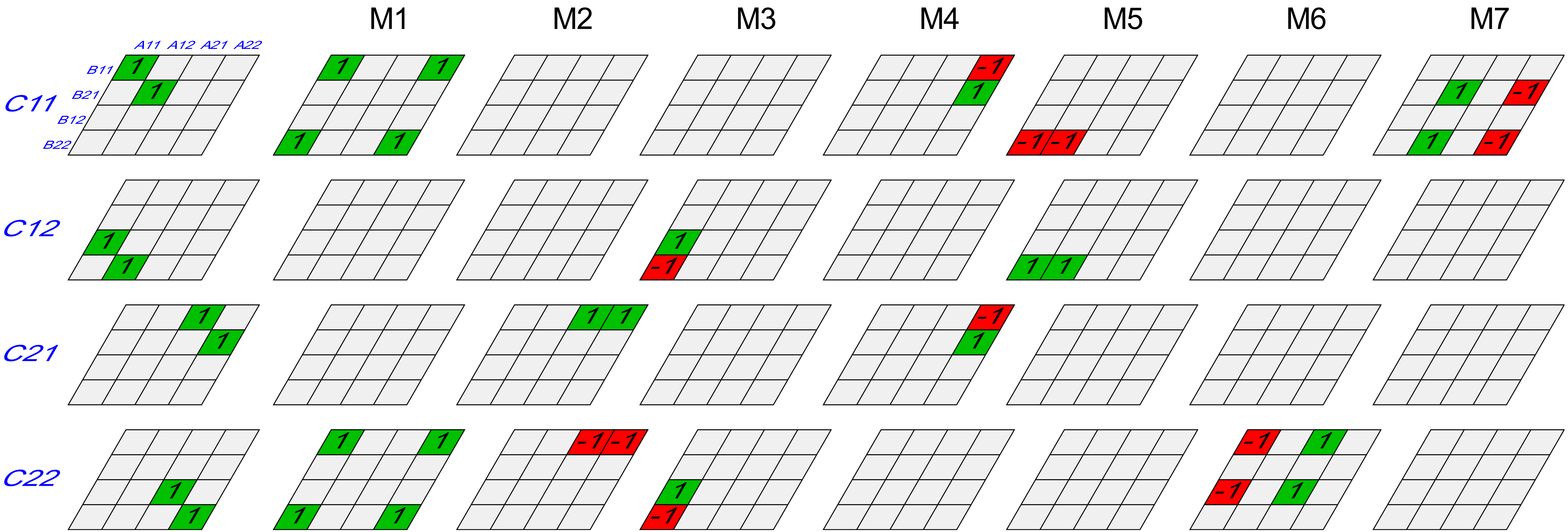
  - 4 multiplications + 3 additions
- Re-associate operations

$$k_1 = c(a + b)$$
$$k_2 = a(d - c)$$
$$k_3 = b(c + d)$$
$$Real\ part = k_1 - k_3$$
$$Imaginary\ part = k_1 + k_2$$

  - 3 multiplications + 5 additions

Strassen Matrix Multiplication Transform

- $\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \times \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$



Strassen Matrix Multiplication Transform

- Matrix multiplication of A and B

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$AB = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & ef + dh \end{bmatrix}$$

  - 8 multiplications and 4 additions
- Re-associate operations

$$M_1 = a(f - h)$$
$$M_2 = h(a + b)$$
$$M_3 = e(c + d)$$
$$M_4 = d(g - e)$$
$$M_5 = (a + d)(e + h)$$
$$M_6 = (b - d)(g + h)$$
$$M_7 = (a - c)(e + f)$$

$$AB = \begin{bmatrix} M_5 + M_4 - M_2 + M_6 & M_1 + M_2 \\ M_3 + M_4 & M_1 + M_5 - M_3 - M_7 \end{bmatrix}$$

  - 7 multiplications and 18 additions, creation of 7 intermediate values

Asymptotic Complexity of Strassen

- Asymptotic complexity of matrix multiplication $\Theta(\mathrm{N}^3), N = 2^n$

$$f(n) = number\ of\ operations\ for\ a\ 2^n \times 2^n\ matrix$$

- Recursive apply Strassen algorithm

$$f(n) = 7f(n-1) + l4^n = (7 + o(1))^n$$

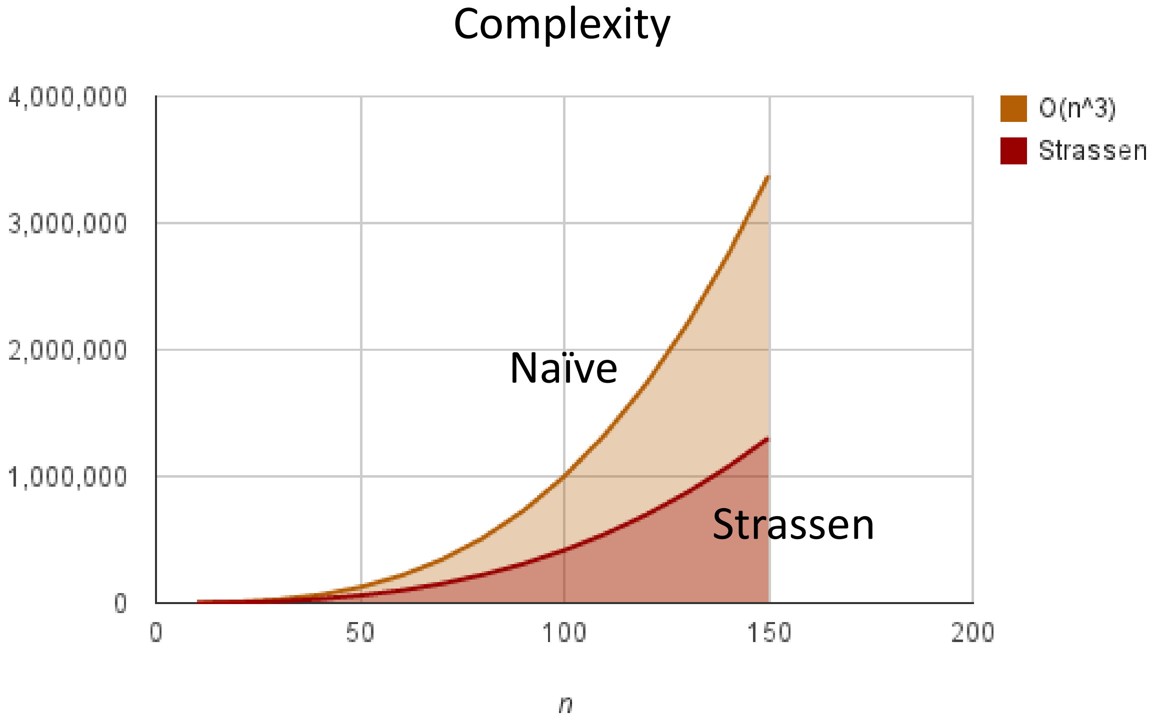
l=some constant depends on the number of additions in Stressen

- Asymptotic complexity using Strassen

$$\Theta\left(\left(7 + o(1)\right)^n\right) = \Theta(N^{\log_2 7 + o(1)}) \approx \Theta(N^{2.8074})$$

Asymptotic Complexity of Strassen

- Reduce the complexity of matrix multiplication from $\Theta(\mathrm{N}^3)$ to $\Theta(N^{2.8074})$ by reducing multiplications
- Comes at the price of reduced numerical stability and requires significantly more memory



Winograd Transform

- Targeting convolutions instead of matrix multiply
- Significantly reduce multiplies
- Achieves varies based on the filter and tile size
- Requires specialized processing depending on the size of the filter and tile
- Winograd hardware typically support only specific tile and filter sizes
    - NVDLA only support 3x3 filters

1D Convolution

Input Filter Output

$$[i_0 \quad i_1 \quad i_2 \quad i_3] \text{ Convolution } [f_0 \quad f_1 \quad f_2] = [o_0 \quad o_1]$$

$$\begin{bmatrix} i_0 & i_1 & i_2 \\ i_1 & i_2 & i_3 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} o_0 \\ o_1 \end{bmatrix}$$

6 multiplications + 4 additions

1D Convolution Using Winograd Transform

- 1D Convolution using Winograd transform

$$\begin{bmatrix} i_0 & i_1 & i_2 \\ i_1 & i_2 & i_3 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} o_0 \\ o_1 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

$$m_1 = (i_0 - i_2)f_0$$
$$m_2 = (i_1 + i_2)\frac{f_0 + f_1 + f_2}{2}$$
$$m_3 = (i_2 - i_1)\frac{f_0 - f_1 + f_2}{2}$$
$$m_4 = (i_1 - i_3)f_2$$

4 multiplications + 12 additions + 2 shifts(divided by 2)
With constant weights
→ 4 multiplications + 8 additions

Linear Algebraic Formulation

Input transform matrix (constant)

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

Filter transform matrix (constant)

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

Output transform matrix (constant)

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

filter matrix

$$f = [f_0 \quad f_1 \quad f_2]^T$$

input matrix

$$i = [i_0 \quad i_1 \quad i_2]^T$$

- Sandwiching those matrices in a chain of matrix multiplies by constant matrices
  - $[GfG^T]$ and $[B^TiB]$
  - Existing in a **Winograd** space
  - GfG^T only need to be performed once, since the filter weights are constant across many applications of the tiled convolution

Linear Algebraic Formulation

Input transform matrix (constant)

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

Filter transform matrix (constant)

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

Output transform matrix (constant)

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

filter matrix

$$f = [f_0 \quad f_1 \quad f_2]^T$$

input matrix

$$i = [i_0 \quad i_1 \quad i_2]^T$$

- Convolution can be performed by combining those matrices with element-wise multiplication
  - $[GfG^T] \odot [B^T iB]$
- Reverse transformation out of the **Winograd** space
  - $Y = A^T\big[[GfG^T] \odot [B^T iB]\big]A$

2D Winograd Transform

- 1D Winograd is nested to make 2D Winograd

Filter, Input Feature map, Output Feature map

$$\begin{array}{|c|c|c|}\hline f_{00} & f_{01} & f_{02} \\ \hline f_{10} & f_{11} & f_{12} \\ \hline f_{20} & f_{21} & f_{22} \\ \hline \end{array} \times \begin{array}{|c|c|c|c|}\hline i_{00} & i_{01} & i_{02} & i_{03} \\ \hline i_{10} & i_{11} & i_{12} & i_{13} \\ \hline i_{20} & i_{21} & i_{22} & i_{23} \\ \hline i_{30} & i_{31} & i_{32} & i_{33} \\ \hline \end{array} = \begin{array}{|c|c|}\hline o_{00} & o_{01} \\ \hline o_{10} & o_{11} \\ \hline \end{array}$$

- Original
  - 36 multiplication
- Winograd
  - 16 multiplication→ 2.25 multiplication reduction

2D Winograd Halo

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

Filter

f_{00}	f_{01}	f_{02}
f_{10}	f_{11}	f_{12}
f_{20}	f_{21}	f_{22}

$\times$

Input Feature map

i_{00}	i_{01}	i_{02}	i_{03}	i_{04}	i_{05}	i_{06}
i_{10}	i_{11}	i_{12}	i_{13}	i_{14}	i_{15}	i_{16}
i_{20}	i_{21}	i_{22}	i_{23}	i_{24}	i_{25}	i_{26}
i_{30}	i_{31}	i_{32}	i_{33}	i_{34}	i_{35}	i_{36}

Halo column

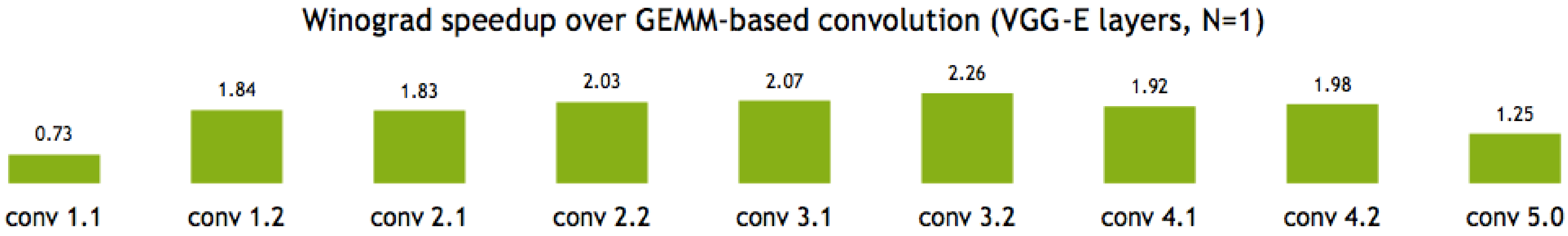
$=$

Output Feature map

o_{00}	o_{01}	o_{02}	o_{03}
o_{10}	o_{11}	o_{12}	o_{13}

Winograd Performance Varies

- Optimal convolution algorithm depends on convolution layer dimensions



  - Meta parameters (data layouts, texture memory) afford higher performance
  - Using texture memory for convolution: 13% inference speedup (GoogLeNet, batch size 1)

Fast Fourier Transform (FFT)

1. Convert filter and input to frequency domain
2. Make convolution a simple multiply
3. Convert back to space domain

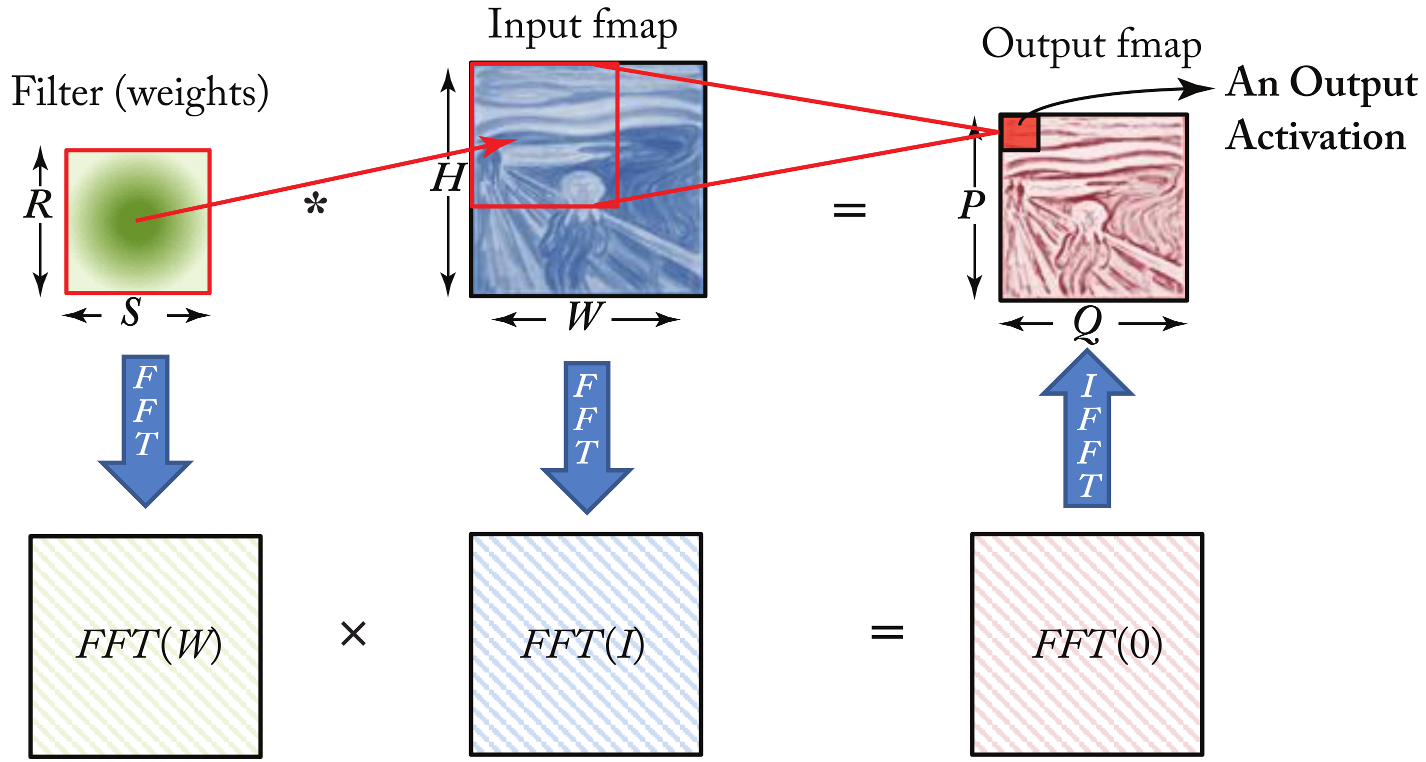
- Follows a similar pattern to the Winograd transform
  - Convert a convolution into a new space where convolution is more computationally efficient

FFT to Accelerate DNN

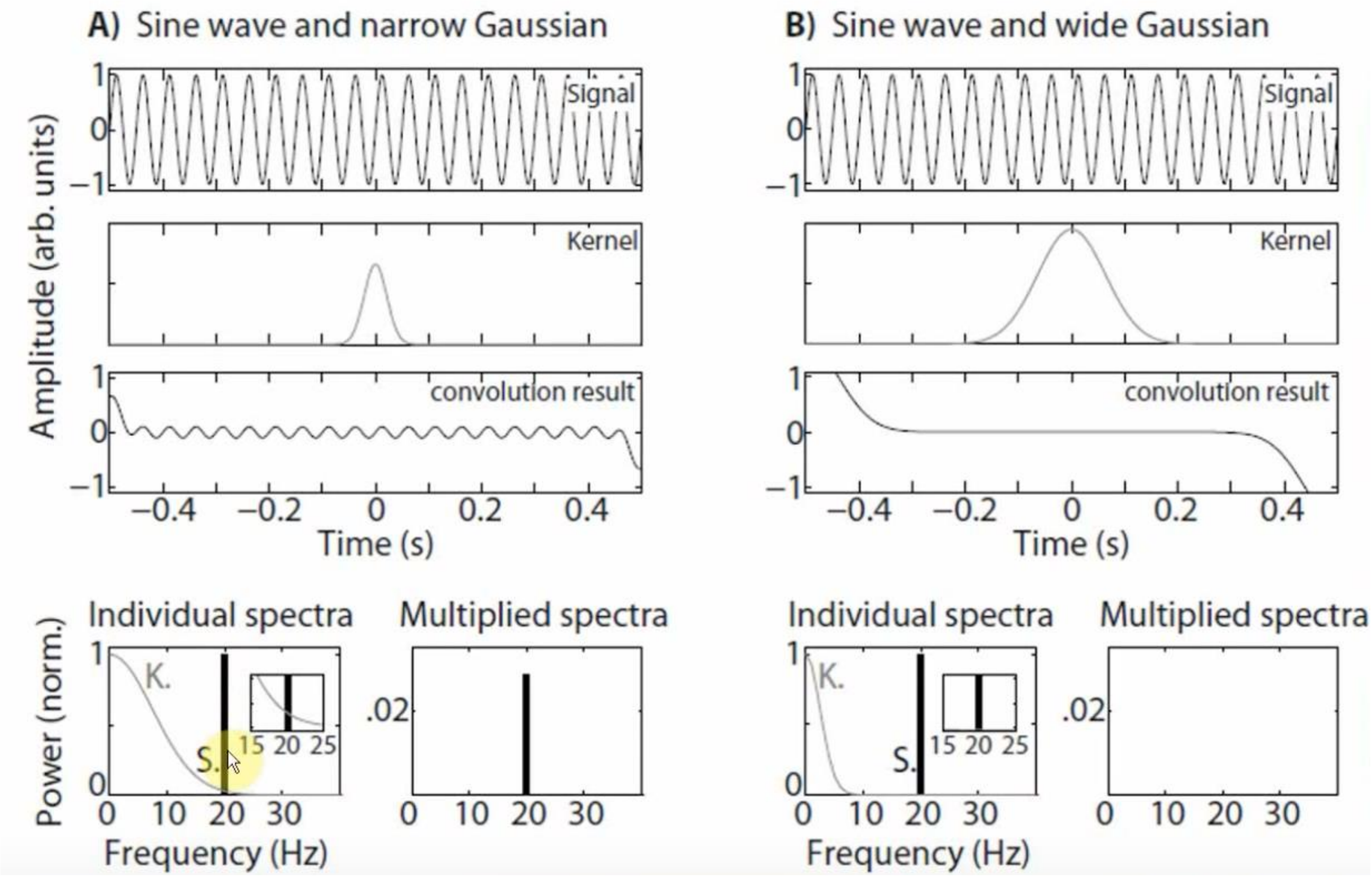
- Convolution in the time domain is equivalent to point-wise multiply in the frequency domain.

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

$\mathcal{F}\{f\}$: Fourier transform of f
$\mathcal{F}\{g\}$: Fourier transform of g
$*$: convolution(not multiplication)



FFT-based Convolution



FFT-based Convolution

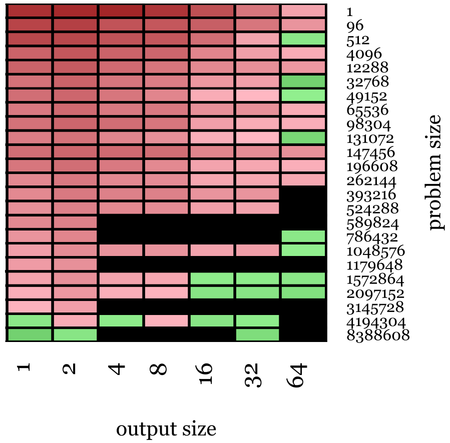


Figure 1: 3×3 kernel (K40m)

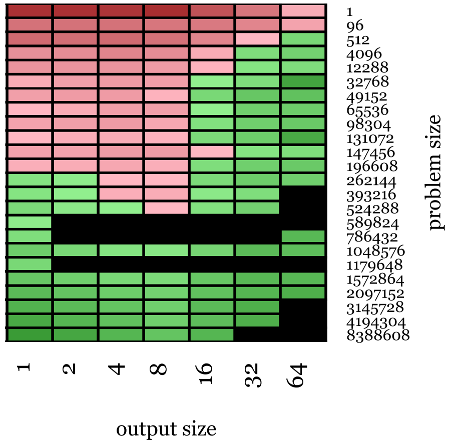


Figure 2: 5×5 kernel (K40m)

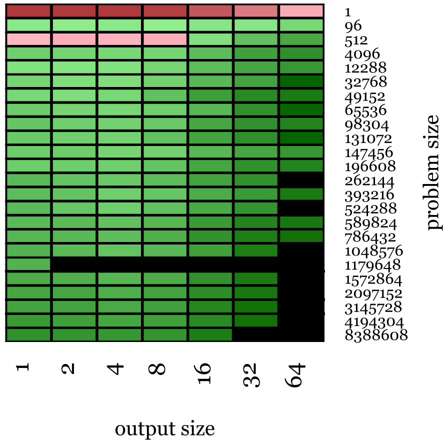


Figure 5: 11×11 kernel (K40m)

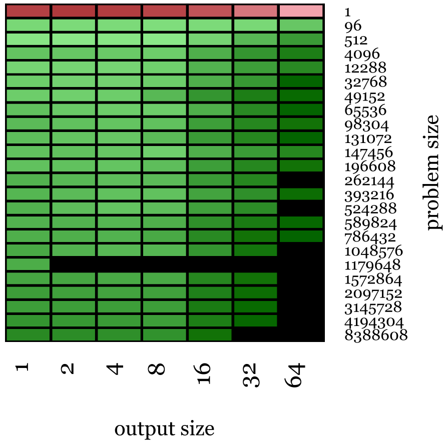


Figure 6: 13×13 kernel (K40m)

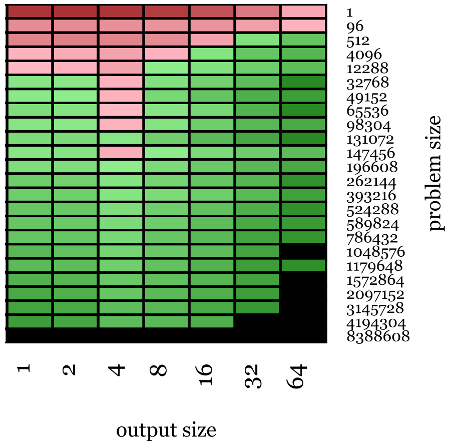


Figure 3: 7×7 kernel (K40m)

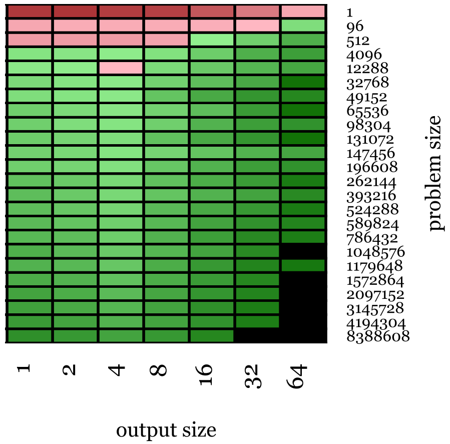
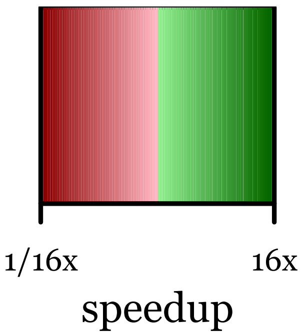


Figure 4: 9×9 kernel (K40m)



Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., & LeCun, Y. (2014). Fast convolutional nets with fbfft: A GPU performance evaluation. *arXiv preprint arXiv:1412.7580.*

FFT Convolution

- Complexity
  - For output size P x Q, Filter size R x S
  - Convert direct convolution $O(PQRS)$ to $O(PQ\log_2 PQ)$

- Drawbacks
  - Computational benefit of FFT decreases with decreasing size of filter
    - Needs $RS > \log_2 PQ$ for there to be a benefit
  - Size of the FFT is dictated by the output feature map size
    - Often much larger than the filter
  - The coefficients in the frequency domain are complex

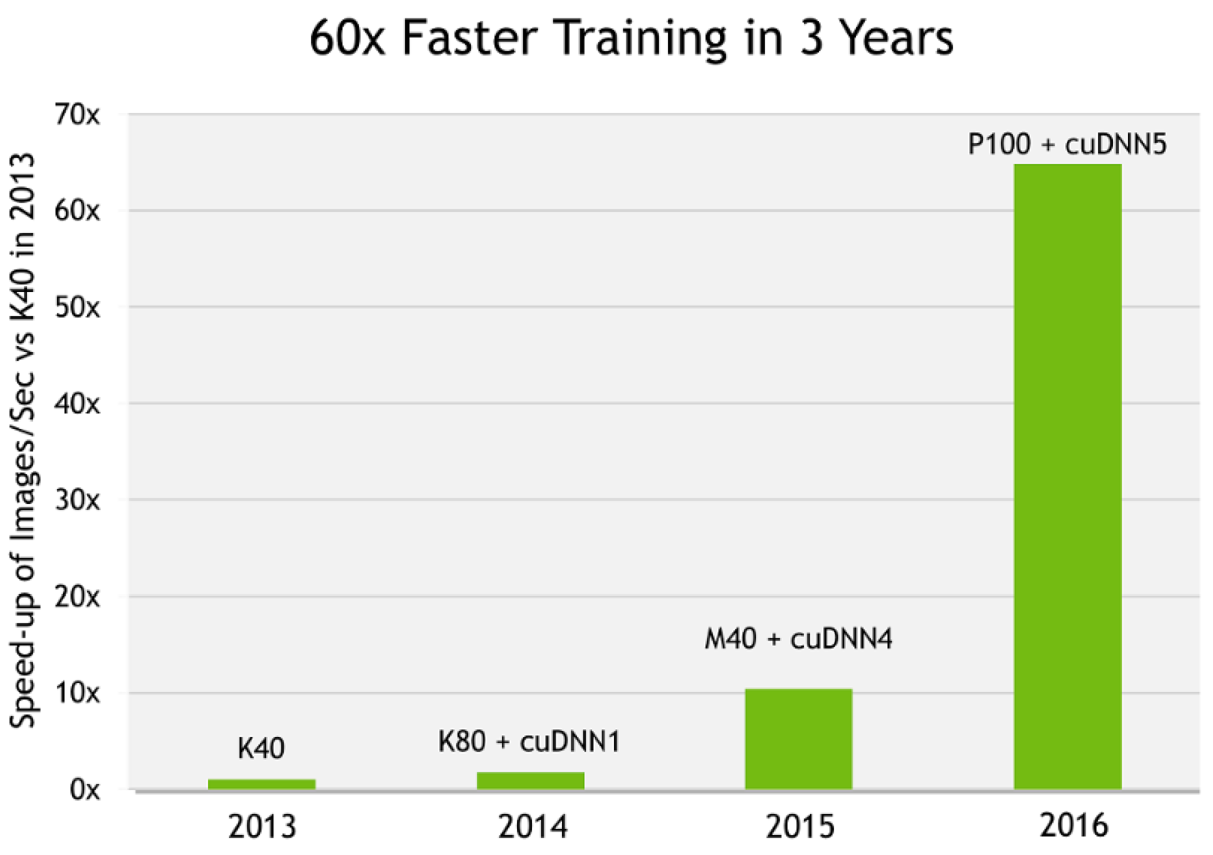
FFT Optimization for DNN Computation

- FFT of the filter can be pre-computed and stored
  - Reduce the number of operations
  - but convolutional filter in frequency domain is much larger than in space domain
- FFT of the input feature map can be computed once to generate multiple channels in the output feature map
- An image contains only real values
  - Its Fourier Transform is symmetric
  - Can be exploited to reduce storage and computation cost
- Can accumulate across channels before performing inverse transform to reduce number of IFFT

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

cuDNN: Speed up with Transformations

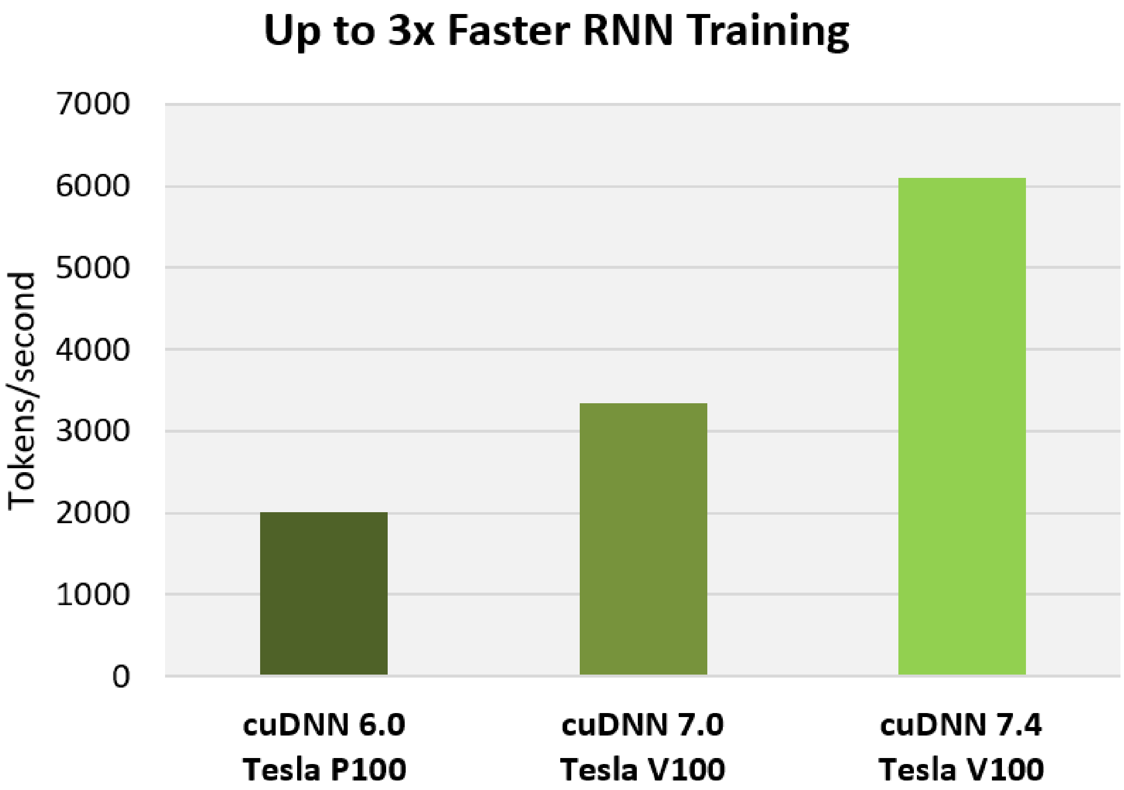


AlexNet training throughput on:

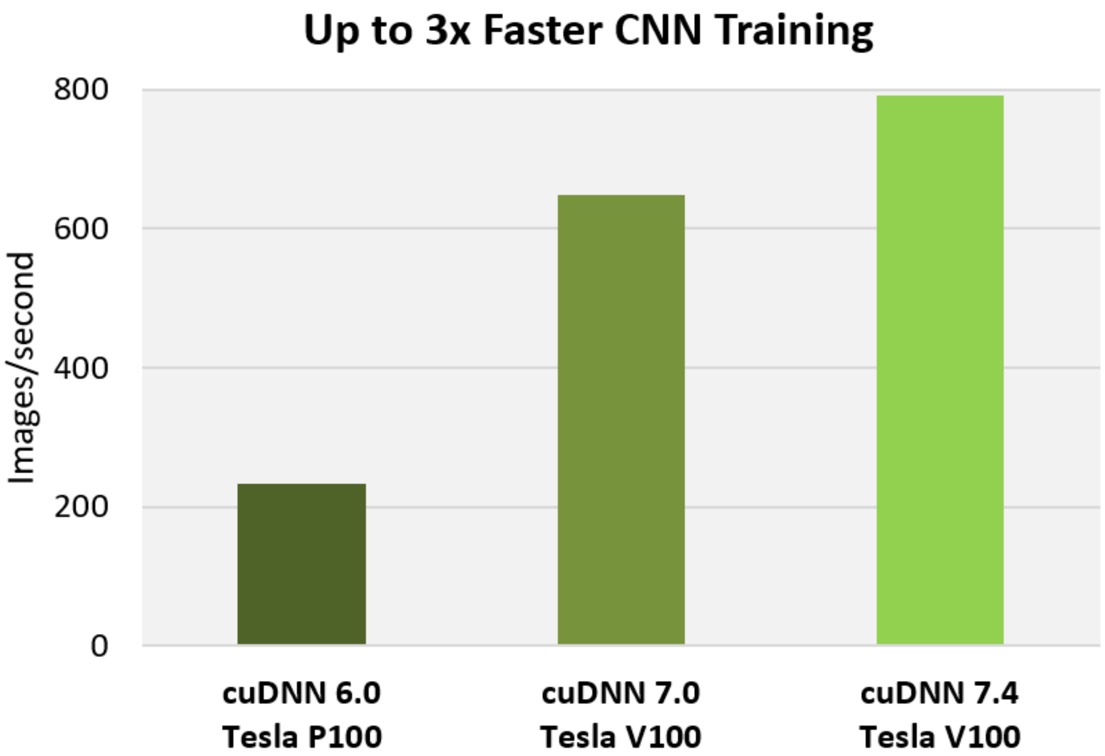
CPU: 1x E5-2680v3 12 Core 2.5GHz. 128GB System Memory, Ubuntu 14.04

M40 bar: 8x M40 GPUs in a node, P100: 8x P100 NVLink-enabled

cuDNN: Speed up with Transformations



TensorFlow performance (tokens/sec), Tesla P100 + cuDNN 6 (FP32) on 17.12 NGC container, Tesla V100 + cuDNN 7.0 (Mixed) on 18.02 NGC container, Tesla V100 + cuDNN 7.4 (Mixed) on 18.10 NGC container, OpenSeq2Seq (GNMT), Batch Size: 64



TensorFlow performance (images/sec), Tesla P100 + cuDNN 6 (FP32) on 17.12 NGC container, Tesla V100 + cuDNN 7.0 (Mixed) on 18.02 NGC container, Tesla V100 + cuDNN 7.4 (Mixed) on 18.10 NGC container, ResNet-50, Batch Size: 128

Selecting A Transform

- Different algorithms might be used for different layer shapes and sizes
  - E.g. FFT for filters greater than 5x5, and Winograd for filters 3x3 and below
- Existing platform libraries dynamically choose the appropriate algorithm for a given shape and size
  - MKL
  - cuDNN

Optimization Tools

- Halide
  - A language for fast, portable computation on images and tensors
  - embedded in C++
  - Support x86, arm, MIPS, RISC-V, PowerPC, CUDA, OpenCL, OpenGl, etc.
- TVM
  - An End to End Machine Learning Compiler Framework for CPUs, GPUs and accelerators
  - Various framework
    - Tensorflow, Keras, etc.
  - Various backend
    - LLVM, C, CPUs, DSPs, GPUs, FPGAs,
- Timeloop
  - A Systematic Approach to DNN Accelerator Evaluation
- Many other optimization tools…

Analysis of Convolution Approaches

	Pro	Con
GEMM	• Generic and stable • Easy to implement (problem mapped into a BLAS call) • Optimized solution if good BLAS is provided	• Additional memory to store the intermediate data • Rely heavily on optimized BLAS
Spatial Domain	• Avoids additional memory copy • Speedy with optimized code	• Rely on individually optimized kernels according to given params, or even given HW architecture
FFT fomain	• Lower computational complexity	• Additional memory to save FFT data • Overhead is big for small kernel size, or large stride

Concluding Remarks

- Temporal platform
  - CPUs and GPUs
- Spatial platforms
  - Communication between ALUs with its own control logic and storage
- Methods that restructure the computations to improve efficiency without any impact on accuracy
  - Nearly bit-wise identical results
  - Reshape computation for efficiency
  - Reduce memory bandwidth
  - Reduce high-cost operations

Concluding Remarks

- Toeplitz transformation
  - Converts a convolution into a matrix multiply by replicating value
  - Widely used in CPUs and GPUs
  - GEMM libraries supports
- Strassen
  - Comes at the price of reduced numerical stability and requires significantly more memory
- Winograd
  - Targeting convolutions instead of matrix multiply
  - Convert to Winograd space, compute, then convert back
- FFT transform
  - Convert filter and input to frequency domain, compute, then convert back