



# CMSC 5743

## Efficient Computing of Deep Neural Networks

### Implementation 01: GEMM

Bei Yu

CSE Department, CUHK

[byu@cse.cuhk.edu.hk](mailto:byu@cse.cuhk.edu.hk)

(Latest update: September 2, 2024)

2024 Fall



① Convolution Basis

② Im2Col

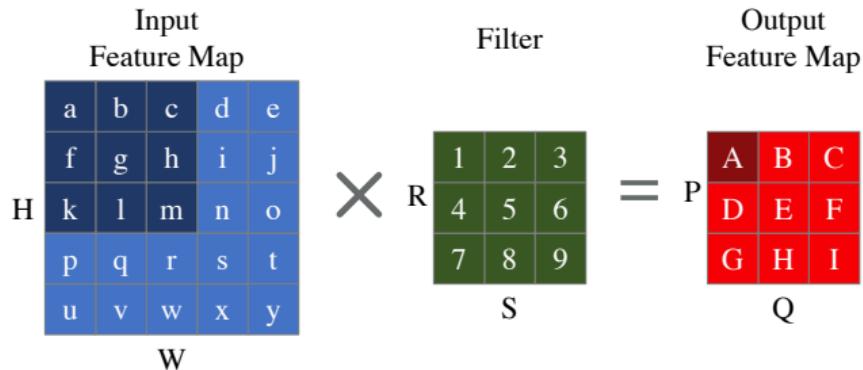
③ Memory-efficient Convolution

④ Memory Layout



# Convolution Basis

# 2D-Convolution



$$\begin{aligned}A = & a \cdot 1 + b \cdot 2 + c \cdot 3 \\& + f \cdot 4 + g \cdot 5 + h \cdot 6 \\& + k \cdot 7 + l \cdot 8 + m \cdot 9\end{aligned}$$

- $\text{H}$ : Height of input feature map
- $\text{W}$ : Width of input feature map
- $\text{R}$ : Height of filter
- $\text{S}$ : Width of filter
- $\text{P}$ : Height of output feature map
- $\text{Q}$ : Width of output feature map

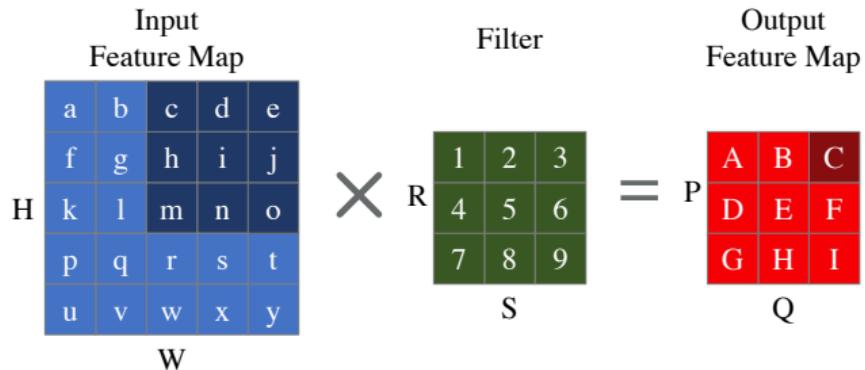
# 2D-Convolution



$$\begin{array}{c} \text{Input} \\ \text{Feature Map} \\ \hline \begin{matrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{matrix} \\ H \quad W \end{array} \times \begin{array}{c} \text{Filter} \\ \hline \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix} \\ R \quad S \end{array} = \begin{array}{c} \text{Output} \\ \text{Feature Map} \\ \hline \begin{matrix} A & B & C \\ D & E & F \\ G & H & I \end{matrix} \\ P \quad Q \end{array}$$

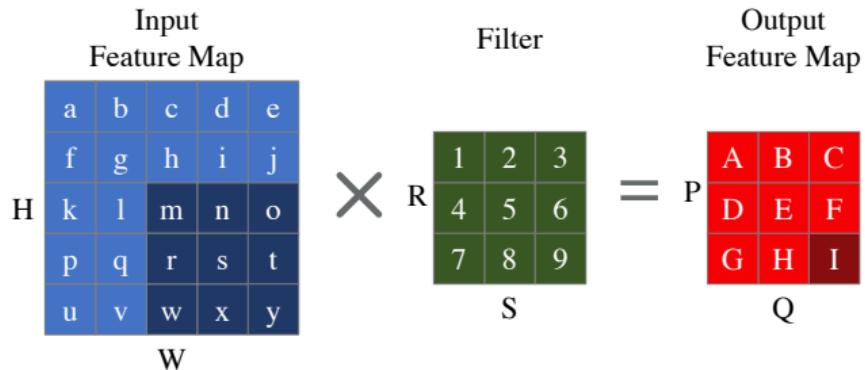
- $H$ : Height of input feature map
- $W$ : Width of input feature map
- $R$ : Height of filter
- $S$ : Width of filter
- $P$ : Height of output feature map
- $Q$ : Width of output feature map
- **stride**: # of rows/columns traversed per step

# 2D-Convolution



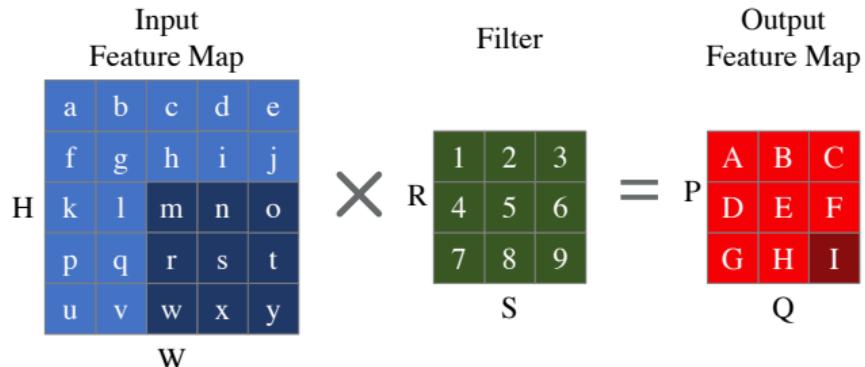
- **H**: Height of input feature map
- **W**: Width of input feature map
- **R**: Height of filter
- **S**: Width of filter
- **P**: Height of output feature map
- **Q**: Width of output feature map
- **stride**: # of rows/columns traversed per step

# 2D-Convolution



- **H**: Height of input feature map
- **W**: Width of input feature map
- **R**: Height of filter
- **S**: Width of filter
- **P**: Height of output feature map
- **Q**: Width of output feature map
- **stride**: # of rows/columns traversed per step

# 2D-Convolution

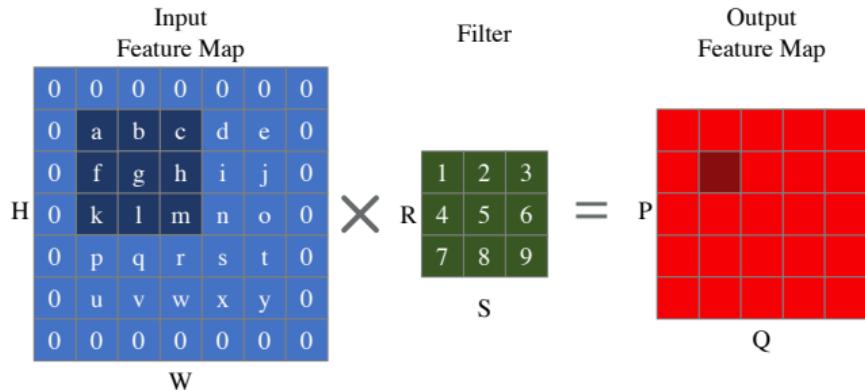


- **H**: Height of input feature map
- **W**: Width of input feature map
- **R**: Height of filter
- **S**: Width of filter
- **P**: Height of output feature map
- **Q**: Width of output feature map
- **stride**: # of rows/columns traversed per step

$$P = \frac{(H - R)}{\text{stride}} + 1;$$

$$Q = \frac{(W - S)}{\text{stride}} + 1.$$

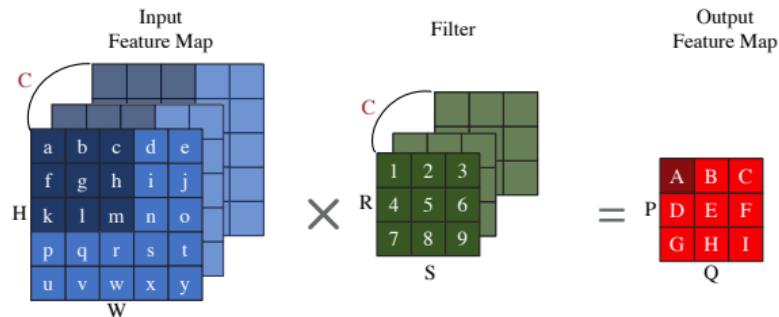
# 2D-Convolution



$$P = \frac{(H - R + 2 \cdot \text{pad})}{\text{stride}} + 1;$$
$$Q = \frac{(W - S + 2 \cdot \text{pad})}{\text{stride}} + 1.$$

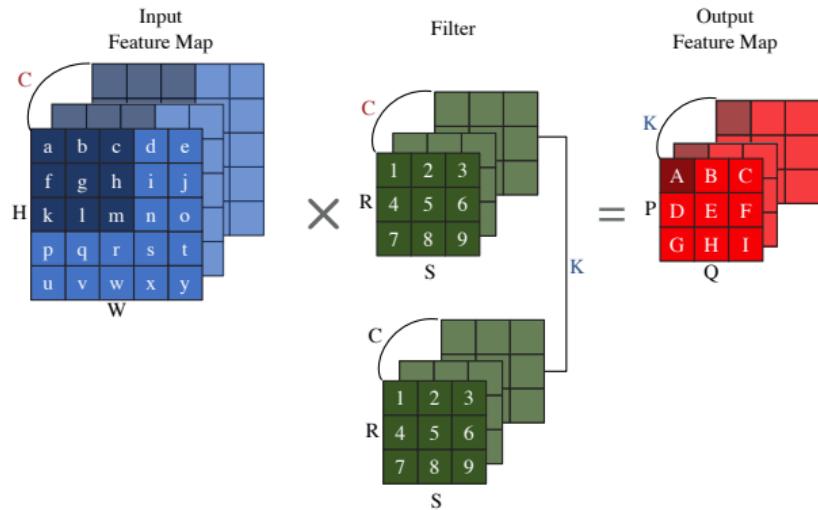
- $H$ : Height of input feature map
- $W$ : Width of input feature map
- $R$ : Height of filter
- $S$ : Width of filter
- $P$ : Height of output feature map
- $Q$ : Width of output feature map
- **stride**: # of rows/columns traversed per step
- **padding**: # of zero rows/columns added

# 3D-Convolution



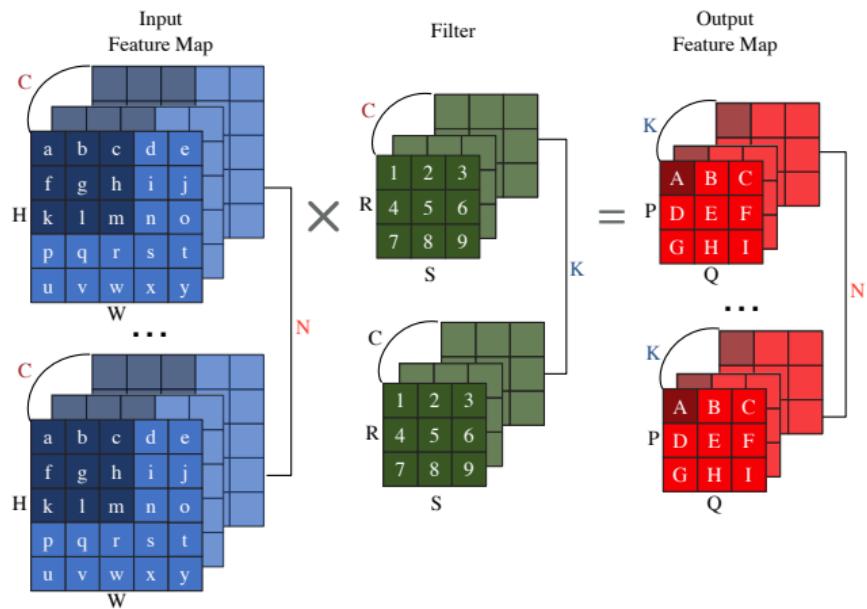
- $\text{H}$ : Height of input feature map
- $\text{W}$ : Width of input feature map
- $\text{R}$ : Height of filter
- $\text{S}$ : Width of filter
- $\text{P}$ : Height of output feature map
- $\text{Q}$ : Width of output feature map
- **stride**: # of rows/columns traversed per step
- **padding**: # of zero rows/columns added
- $\text{C}$ : # of input channels

# 3D-Convolution



- **H:** Height of input feature map
- **W:** Width of input feature map
- **R:** Height of filter
- **S:** Width of filter
- **P:** Height of output feature map
- **Q:** Width of output feature map
- **stride:** # of rows/columns traversed per step
- **padding:** # of zero rows/columns added
- **C:** # of input channels
- **K:** # of output channels

# 3D-Convolution



- **H:** Height of input feature map
- **W:** Width of input feature map
- **R:** Height of filter
- **S:** Width of filter
- **P:** Height of output feature map
- **Q:** Width of output feature map
- **stride:** # of rows/columns traversed per step
- **padding:** # of zero rows/columns added
- **C:** # of input channels
- **K:** # of output channels
- **N:** Batch size



The diagram shows the computation of a 3x3 convolution kernel on a 7x7 input. The input is a 7x7 grid of values from 0 to 2. The kernel is a 3x3 grid of values [1, 0, 0; 1, 1, 1; -1, 0, 1]. The result is a 5x5 output grid. The highlighted part of the input shows the receptive field of the bottom-right output unit, which is calculated by applying the kernel to the 3x3 input window centered at that position.

0	0	0	0	0	0	0
0	2	2	1	1	2	0
0	2	0	1	1	0	0
0	2	0	1	2	0	0
0	1	1	1	1	1	0
0	0	0	1	0	2	0
0	0	0	0	0	0	0

$\times$

1	0	0
1	1	1
1	0	-1

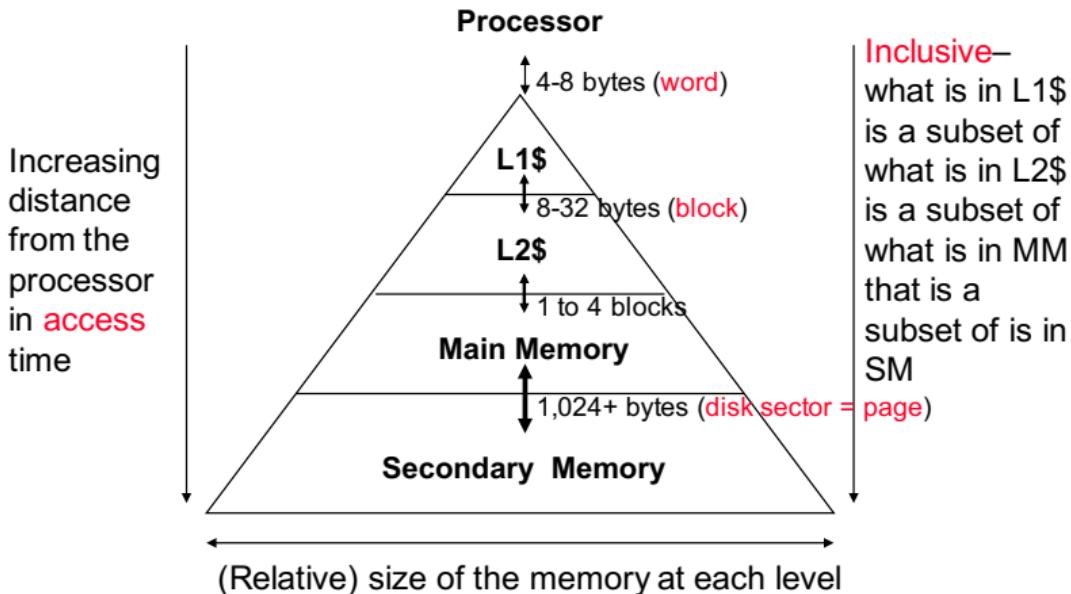
=

4	6	3	5	4
2	6	2	4	4
1	5	3	4	4
2	4	3	3	4
0	2	2	4	3

Direct convolution: No extra memory overhead

- Low performance
- Poor memory access pattern due to geometry-specific constraint
- Relatively short dot product

# Background: Memory System



- **Spatial** locality
- **Temporal** Locality



# Im2Col

# Im2col (Image2Column) Convolution

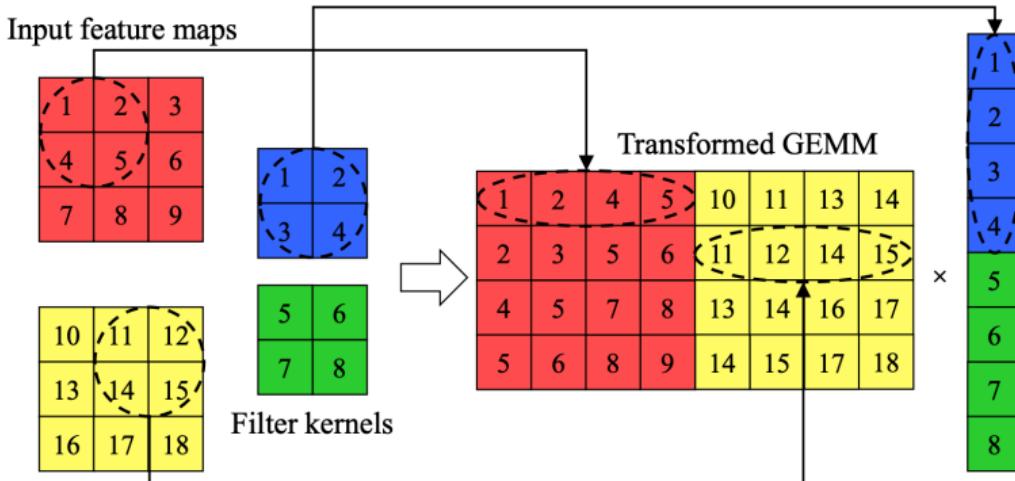


0	0	0	0	0	0	0	0
0	2	2	1	1	2	0	
0	2	0	1	1	0	0	
0	2	0	1	2	0	0	
0	1	1	1	1	1	0	
0	0	0	1	0	2	0	
0	0	0	0	0	0	0	

$$\begin{array}{c} \text{Input Image: } \\ \begin{matrix} 0 & 0 & 0 & 2 & 2 & 0 & 2 & 0 \\ 0 & 0 & 2 & 2 & 1 & 2 & 0 & 1 \\ 0 & 0 & 0 & 2 & 1 & 1 & 0 & 1 & 1 \\ \vdots & & & & & & & \\ 1 & 1 & 0 & 1 & 2 & 0 & 1 & 1 \\ \vdots & & & & & & & \\ 1 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \end{matrix} \quad 25 \times 9 \\ \xrightarrow{\hspace{1cm}} \quad \times \quad = \\ \text{Filter: } \\ \begin{matrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ -1 \end{matrix} \quad 9 \times 1 \end{array}$$

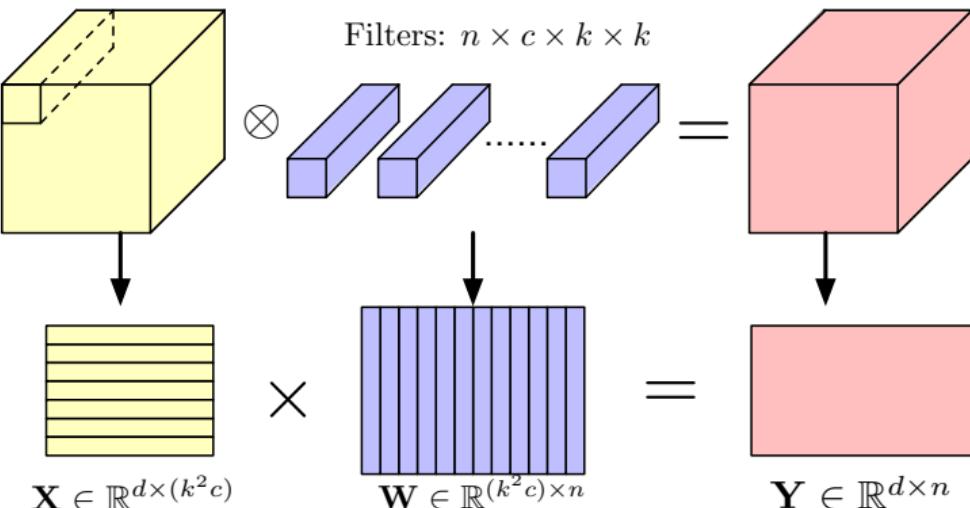
- Large extra memory overhead
- Good performance
- BLAS-friendly memory layout to enjoy SIMD/locality/parallelism
- Applicable for any convolution configuration on any platform

# Im2col (Image2Column) Convolution



- Input channel #: 2
- Output channel #: 1

# Im2col (Image2Column) Convolution

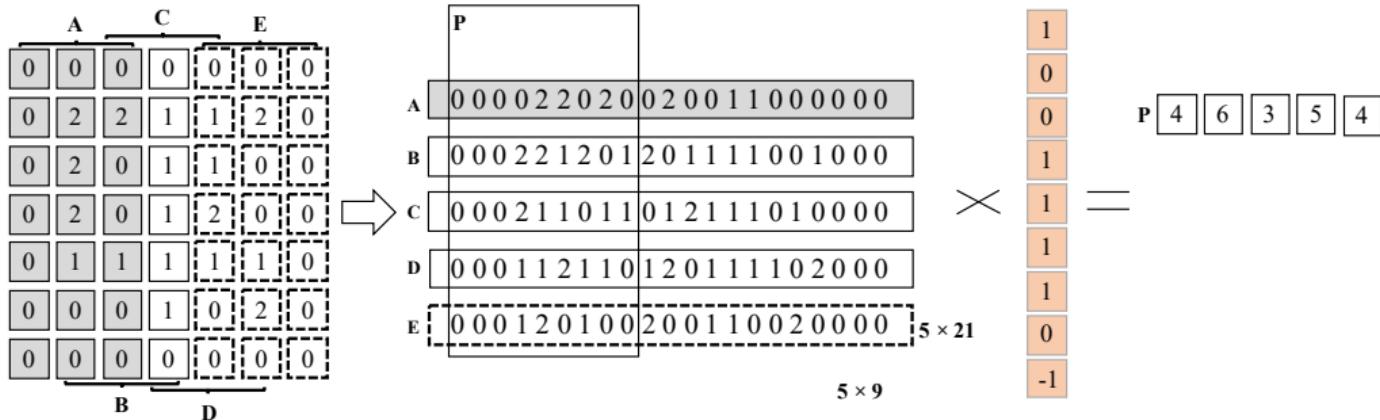


- Transform convolution to **matrix multiplication**
- **Unified** calculation for both convolution and fully-connected layers



# Memory-efficient Convolution

# Memory-efficient Convolution (MEC)

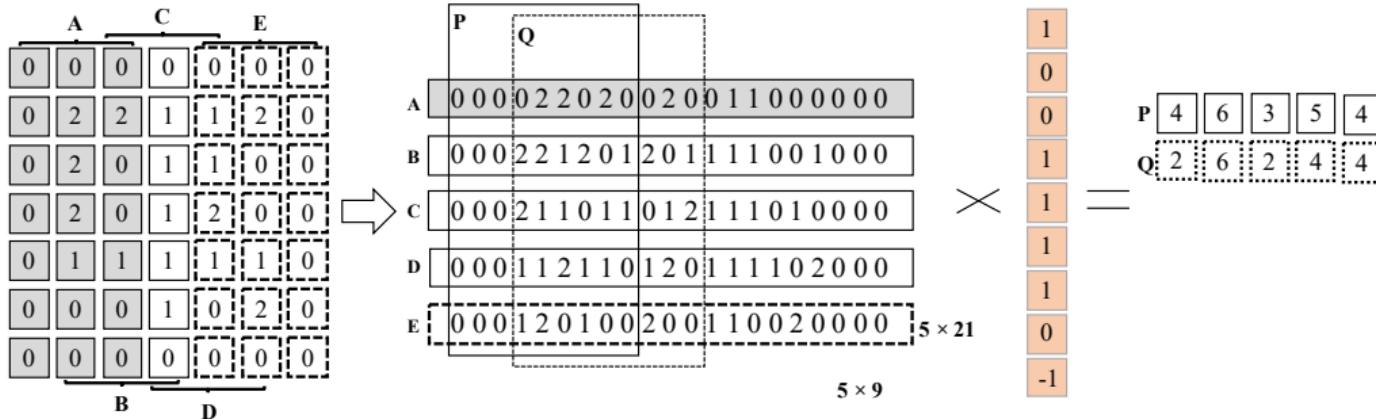


1

- Sub matrices in the lowered matrix will be “**sgemm**” ed in parallel
- Smaller memory foot print, cache locality, and explicit parallelism

<sup>1</sup>Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*.

# Memory-efficient Convolution (MEC)

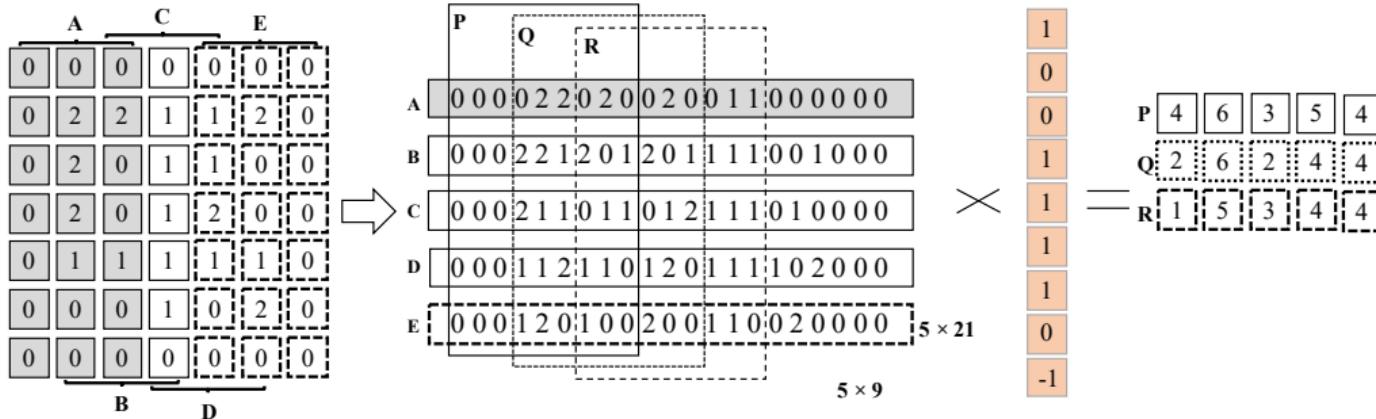


1

- Sub matrices in the lowered matrix will be “**sgemm**” ed in parallel
- Smaller memory foot print, cache locality, and explicit parallelism

<sup>1</sup>Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*.

# Memory-efficient Convolution (MEC)

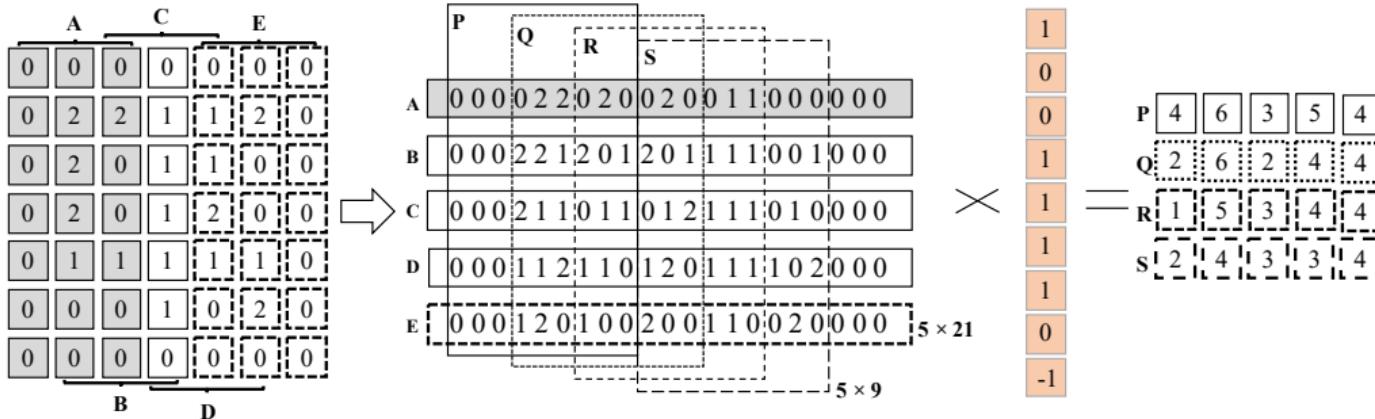


1

- Sub matrices in the lowered matrix will be “**sgemm**” ed in parallel
- Smaller memory foot print, cache locality, and explicit parallelism

<sup>1</sup>Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*.

# Memory-efficient Convolution (MEC)

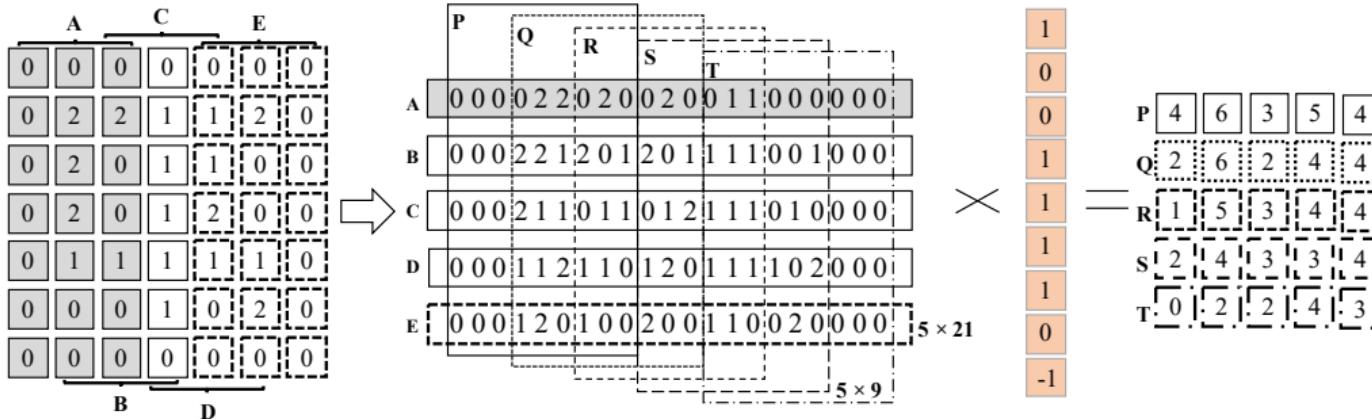


1

- Sub matrices in the lowered matrix will be “**sgemm**” ed in parallel
- Smaller memory foot print, cache locality, and explicit parallelism

<sup>1</sup>Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*.

# Memory-efficient Convolution (MEC)



1

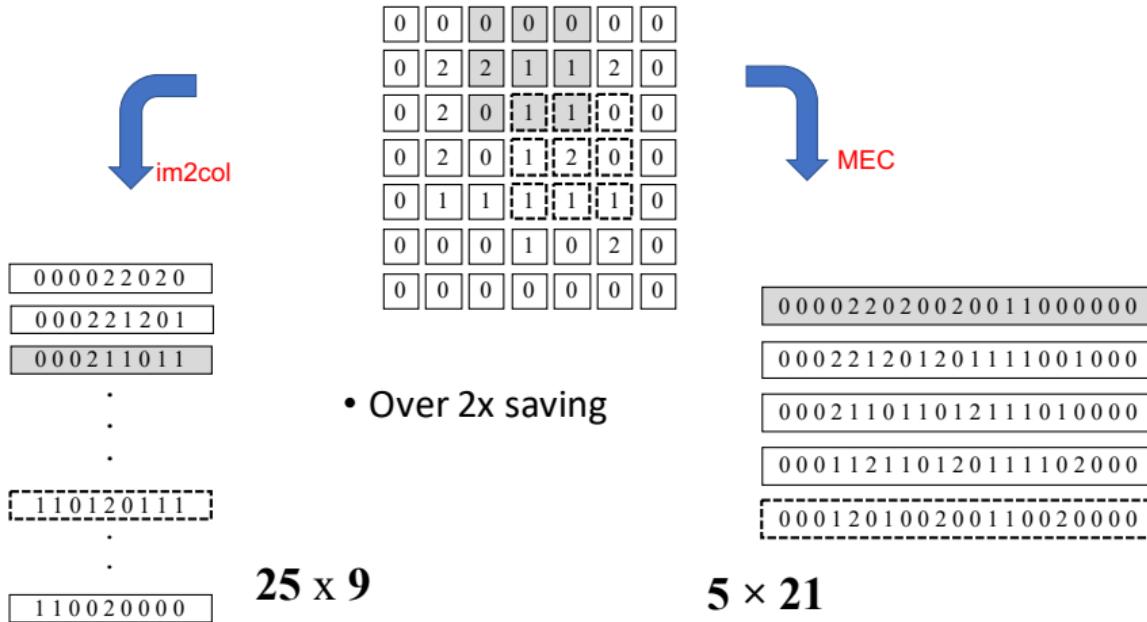
- Sub matrices in the lowered matrix will be “**sgemm**” ed in parallel
- Smaller memory foot print, cache locality, and explicit parallelism

<sup>1</sup>Minsik Cho and Daniel Brand (2017). “MEC: memory-efficient convolution for deep neural network”. In: *Proc. ICML*.

# Memory-efficient Convolution (MEC)



Over  $2\times$  memory saving<sup>2</sup>:



<sup>2</sup>Minsik Cho and Daniel Brand (2017). "MEC: memory-efficient convolution for deep neural network". In: *Proc. ICML*.



# Memory Layout



# Data Layout Formats<sup>3</sup>

- $N$  is the batch size
- $C$  is the number of feature maps
- $H$  is the image height
- $W$  is the image width

## EXAMPLE

$N = 1$

$C = 64$

$H = 5$

$W = 4$

$c = 0$

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

$c = 1$

20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39

$c = 2$

40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59

$c = 30$

600	601	602	603
604	605	606	607
608	609	610	611
612	613	614	615
616	617	618	619

$c = 31$

620	621	622	623
624	625	626	627
628	629	630	631
632	633	634	635
636	637	638	639

$c = 32$

640	641	642	643
644	645	646	647
648	649	650	651
652	653	654	655
656	657	658	659

...

$c = 62$

1240	1241	1242	1243
1244	1245	1246	1247
1248	1249	1250	1251
1252	1253	1254	1255
1256	1257	1258	1259

$c = 63$

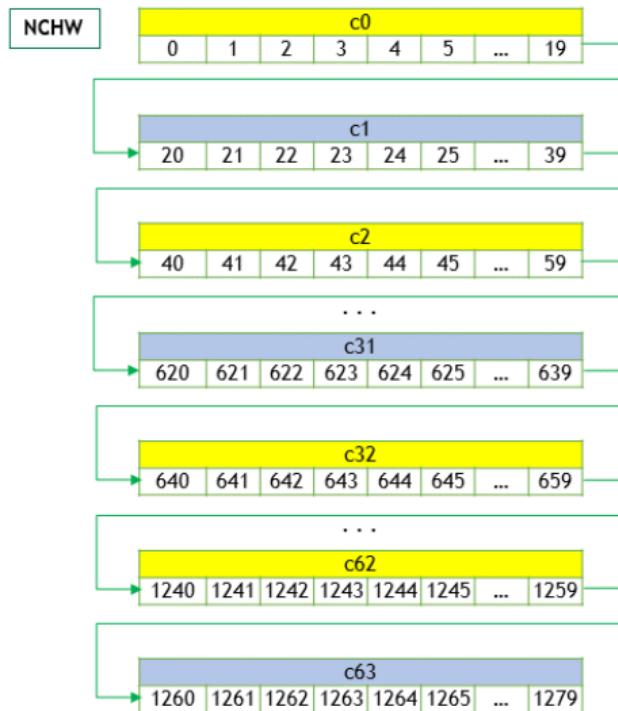
1260	1261	1262	1263
1264	1265	1266	1267
1268	1269	1270	1271
1272	1273	1274	1275
1276	1277	1278	1279

<sup>3</sup><https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html> 17/23

# NCHW Memory Layout



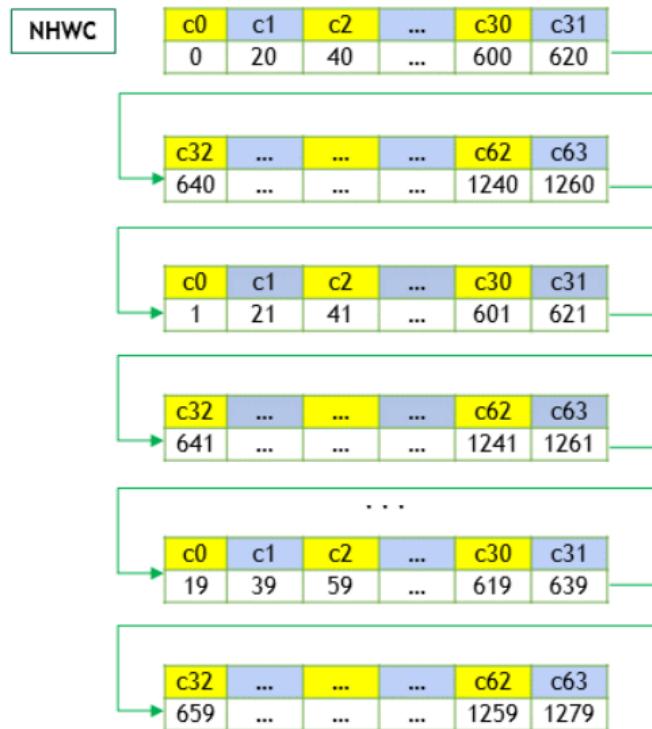
- Begin with first channel ( $c=0$ ), elements arranged contiguously in row-major order
- Continue with second and subsequent channels until all channels are laid out





- Begin with the first element of channel 0, then proceed to the first element of channel 1, and so on, until the first elements of all the C channels are laid out
- Next, select the second element of channel 0, then proceed to the second element of channel 1, and so on, until the second element of all the channels are laid out
- Follow the row-major order of channel 0 and complete all the elements
- Proceed to the next batch (if  $\text{N}$  is  $> 1$ )

# NHWC Memory Layout



# Memory Layout in Im2col

## Image to column operation (im2col)

Slide the input image like a convolution but each patch become a column vector.

Input Image [4x4x3]

		33	34	35	36
	17	18	19	20	40
	2	3	4	6	7
	24	44			
	1	5	6	7	8
	28	48			
	9	10	11	12	
	13	14	15	16	

Result: [12x9]

9 possible  
Sliding window  
positions

1	2	3	5	6	7	9	10	11
2	3	4	6	7	8	10	11	12
5	6	7	9	10	11	13	14	15
6	7	8	10	11	12	14	15	16
17	18	19	21	22	23	25	26	27
18	19	20	22	23	24	26	27	28
21	22	23	25	26	27	29	30	31
22	23	24	26	27	28	30	31	32
33	34	35	37	38	39	41	42	43
34	35	36	38	39	40	42	43	44
37	38	39	41	42	43	45	46	47
38	39	40	42	43	44	46	47	48

Kernel Width:2

Kernel Height:2

Stride:1,

Padding:0

$$W_{out} = (W_{in} - kW + 2^kP)/S + 1$$

$$H_{out} = (H_{in} - kH + 2^kP)/S + 1$$

$$W_{out} = (4 \cdot 2) / 1 + 1 = 3$$

$$H_{out} = (4 \cdot 2) / 1 + 1 = 3$$

2x3 column vector  
[2x2] R, [2x2] G, [2x2] B

We can multiply this result matrix [12x9]

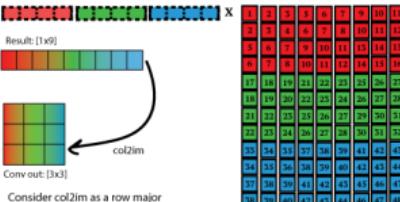
with a kernel [1x12].

result = kernel x matrix

The result would be a row vector [1x9].

We need another operation that will convert

this row vector into a image [3x3].



We get true performance gain

when the kernel has a large number of filters, ie: F=4

and/or you have a batch of images (N=4). Example for the input batch [4x4x3x4], convolved with 4 filters [2x2x3x2].

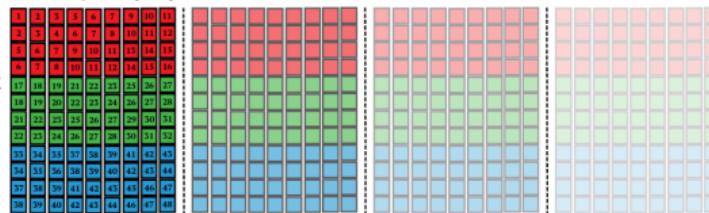
The only problem with this approach is the amount of memory

Reshaped kernel: [1x12]

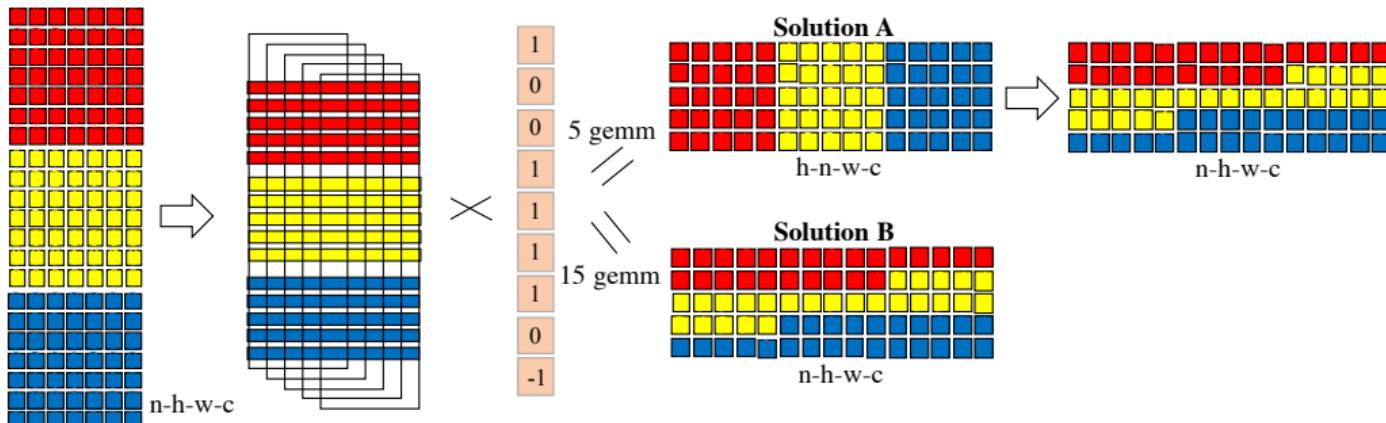
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12

Converted Input batch [12x36]

X

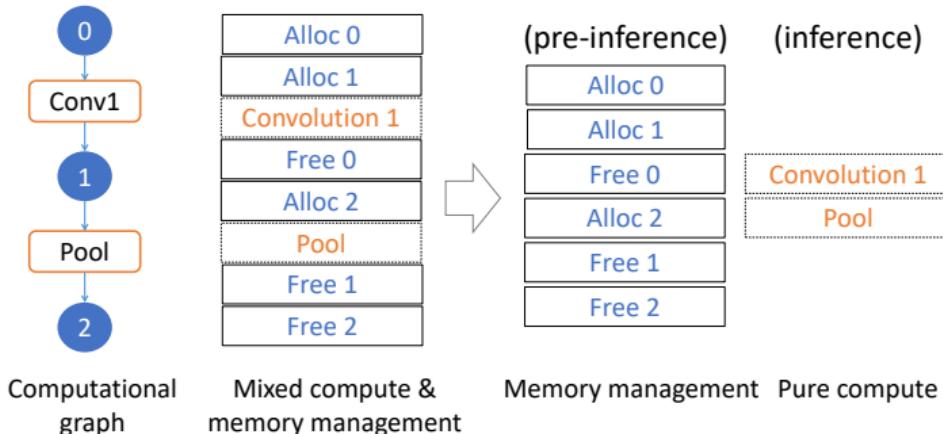


# Memory Layout in Mini-batch MEC



- MEC w. mini-batch: can use  $n-h-w-c$  format
- Fusing convolution+pooling can be another solution

# Memory optimization of MNN



- MNN can infer the exact required memory for the entire graph:
  - virtually walking through all operations
  - summing up all allocation and freeing