

ECE-GY 9143

Introduction to High Performance Machine Learning

Efficient Convolution

Lecture 12

Parijat Dube

How to stack convolutions

- Suppose we stack two 3×3 conv layers (stride 1)
- Each neuron sees 3×3 region of previous activation map
- **Question:** What is the size of the receptive field of a neuron on the second conv layer (i.e. How big of a region in the input does he sees)?

→ 5 x 5

receptive field of 2nd conv layer

- **Question:** If we stack **three** 3×3 conv layers, what is the size of the receptive field of a neuron in the third layer?

→ 7 x 7

receptive field of 3rd conv layer

→ Three 3×3 convolutions give similar representational power as a single 7×7 convolution

The power of small filters

- Suppose input is $H \times W \times C$ and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)

one CONV with 7×7 filters

Number of weights:

$$= C \times (7 \times 7 \times C) = \underline{\underline{49 C^2}}$$

three CONV with 3×3 filters

Number of weights:

$$= 3 \times C \times (3 \times 3 \times C) = \underline{\underline{27 C^2}}$$

Fewer parameters, more nonlinearity

Number of multiply-adds:

$$= (H \times W \times C) \times (7 \times 7 \times C)$$

$$= \underline{\underline{49 HWC^2}}$$

Number of multiply-adds:

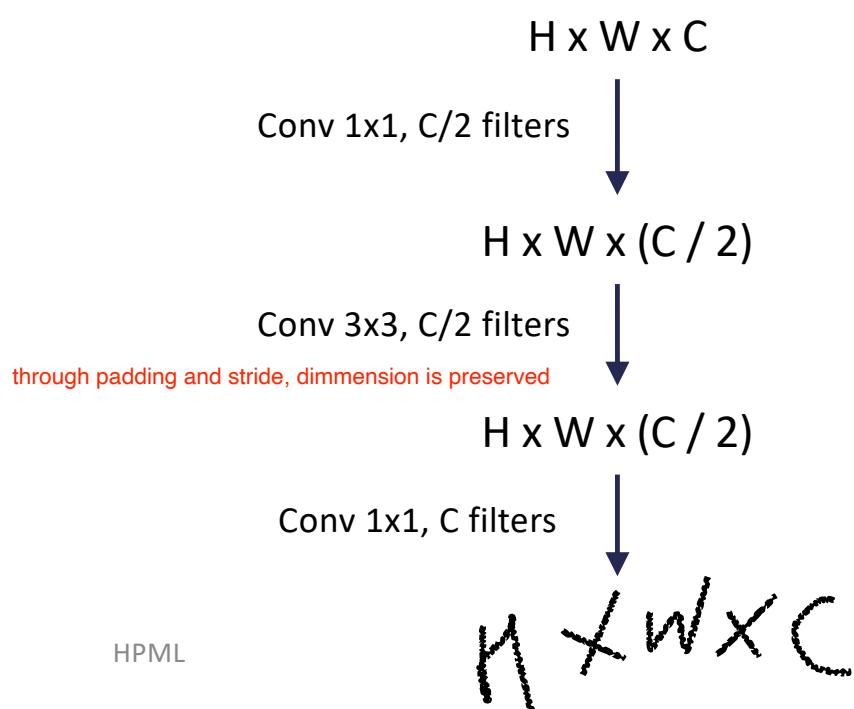
$$= 3 \times (H \times W \times C) \times (3 \times 3 \times C)$$

$$= \underline{\underline{27 HWC^2}}$$

Less compute, more nonlinearity

The power of small filters

- Why stop at 3×3 filters? → use 1×1 !

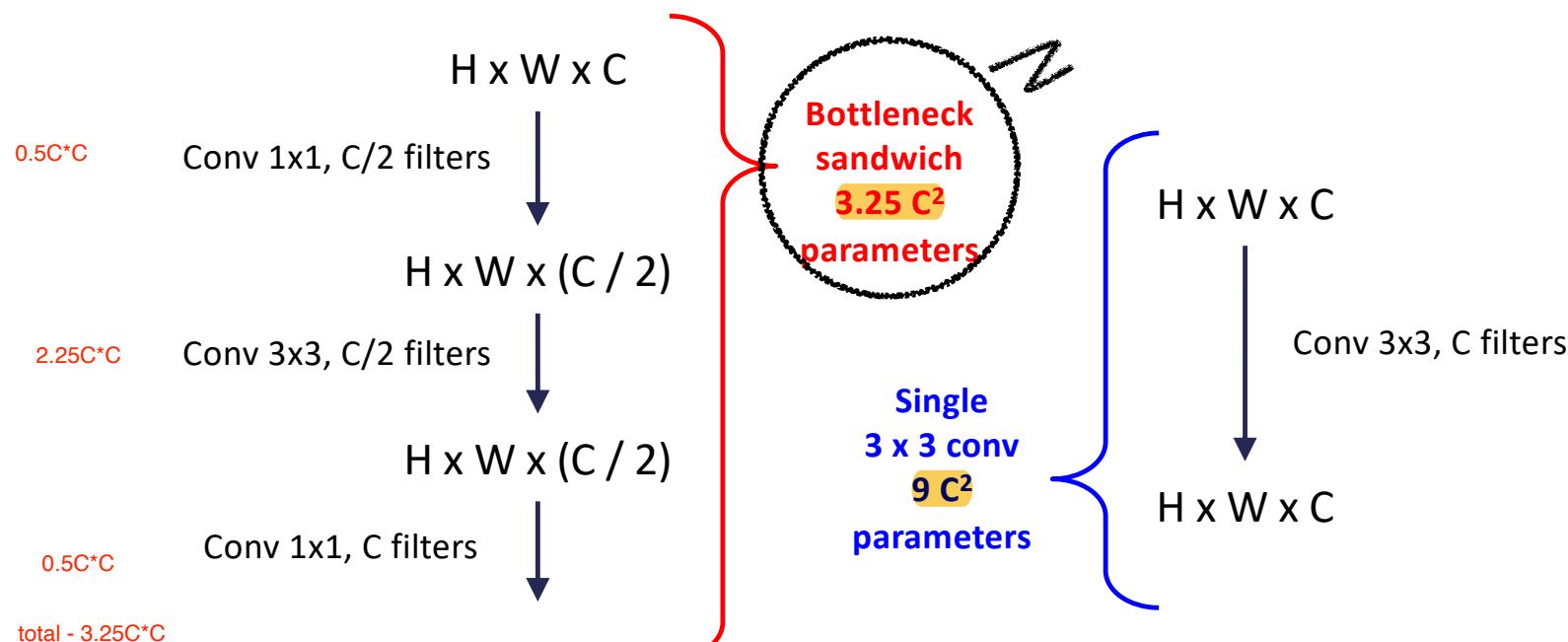


1. “bottleneck” 1×1 conv to reduce dimension
2. 3×3 conv at reduced dimension
3. Restore dimension with another 1×1 conv

[Seen in Lin et al, “Network in Network”, GoogLeNet, ResNet]

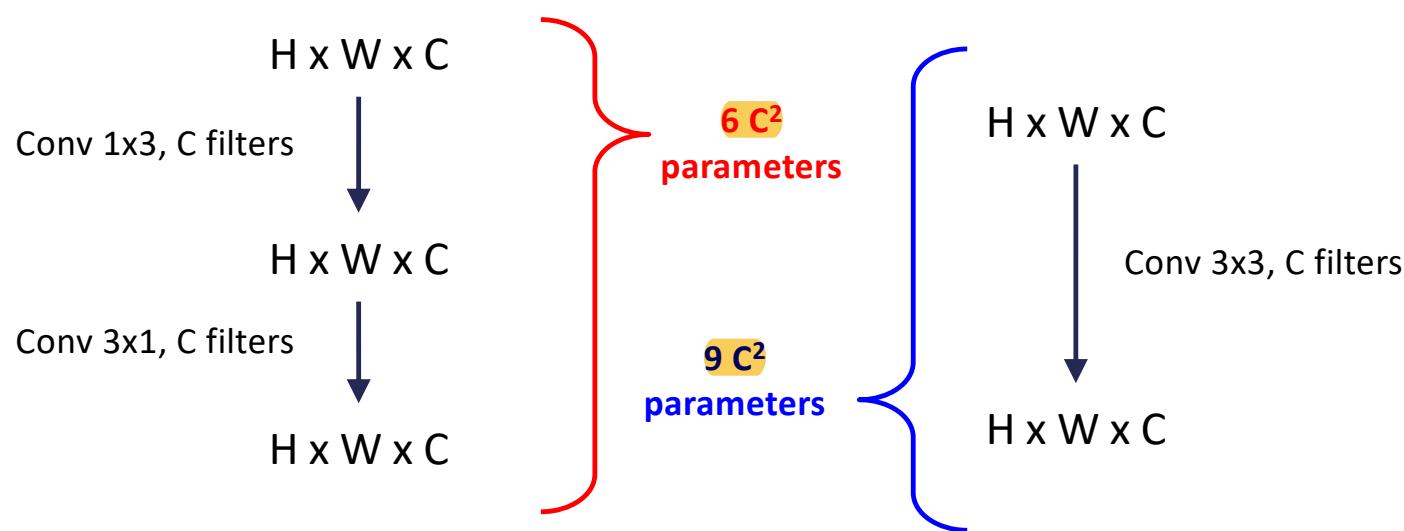
Bottleneck vs single conv

- More nonlinearity, fewer parameters, less compute!
as more activation functions



The power of small filters

- Still using 3×3 filters ... can we break it up?



How to stack convolutions - Recap

- Replace large convolutions (5×5 , 7×7) with stacks of 3×3 convolutions
- 1×1 “bottleneck” convolutions are very efficient why??
- Replace $N \times N$ convolutions into $1 \times N$ and $N \times 1$
- All of the above give fewer parameters, less compute, more nonlinearity
- Szegedy et al, “Rethinking the Inception Architecture for Computer Vision”

Computing NN Layers



Work and Depth Cost Model

- Cost Model for Parallel Algorithms (ignores communication cost) executing on multiple processors

to find out which algorithm is the best

- Assumptions:

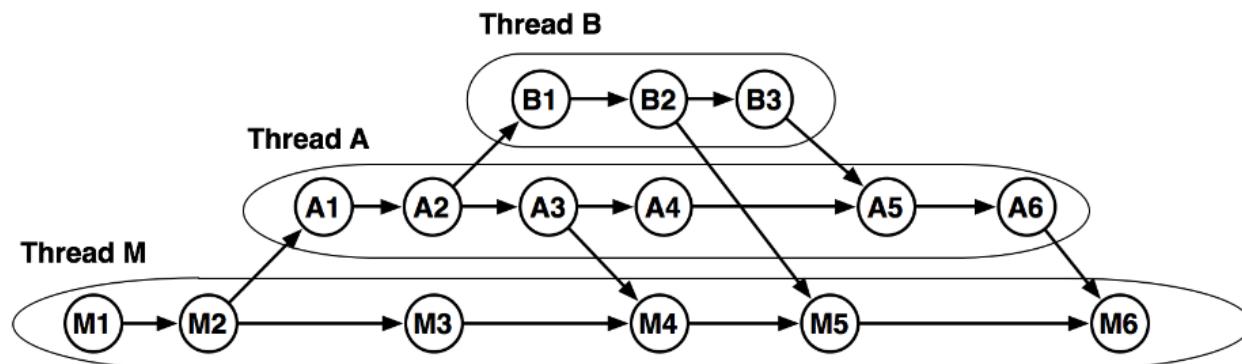
- Communication time among processors ignored
- Memory accesses ignored
- Processors activity is clock-synchronized
- Computations steps are constant size



In the work-depth cost model, work is the total number of operations in an algorithm, and depth is the length of the longest chain of computations. When executing tasks in parallel, the total work is the sum of the work of the tasks, and the total depth is the maximum of the depth of the tasks. When executing tasks sequentially, both the work and depth are summed.

Work and Depth Cost Model

- The **execution of a parallel algorithm** can be represented by a directed acyclic graph - **DAG**
- Directed edges represent **data dependencies**
- Vertices represent a **computation**



from: http://www.cs.cmu.edu/afs/cs/academic/class/15210-f15/www/tapp.html#_dag_representation

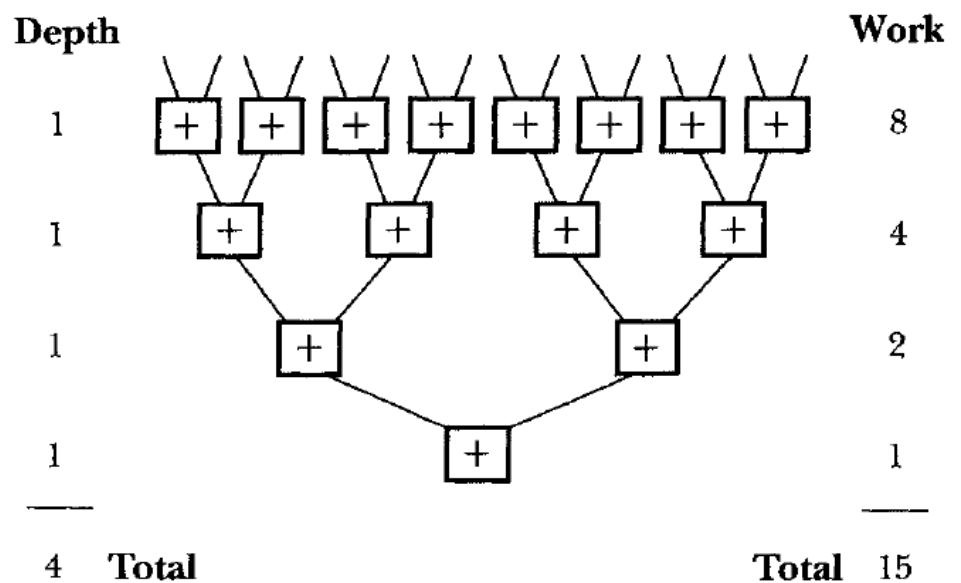
Work is usually viewed as a measure of the total cost of a computation (integral of needed resources over time), and also specifies the running time if the algorithm is executed on a sequential processor. The depth represents the best possible running time, assuming an ideal machine with an unlimited number of processors.

Work and Depth Cost Model (1)

- Work w : the total number of operations executed by a computation
- Depth D (also called **span**): the longest chain of sequential dependencies
- Example:
 - Reduction with SUM function

total work = 15

depth = 4



From: <https://www.cs.cmu.edu/afs/cs/Web/People/blelloch/papers/B85.pdf>

Work and Depth Model (2)

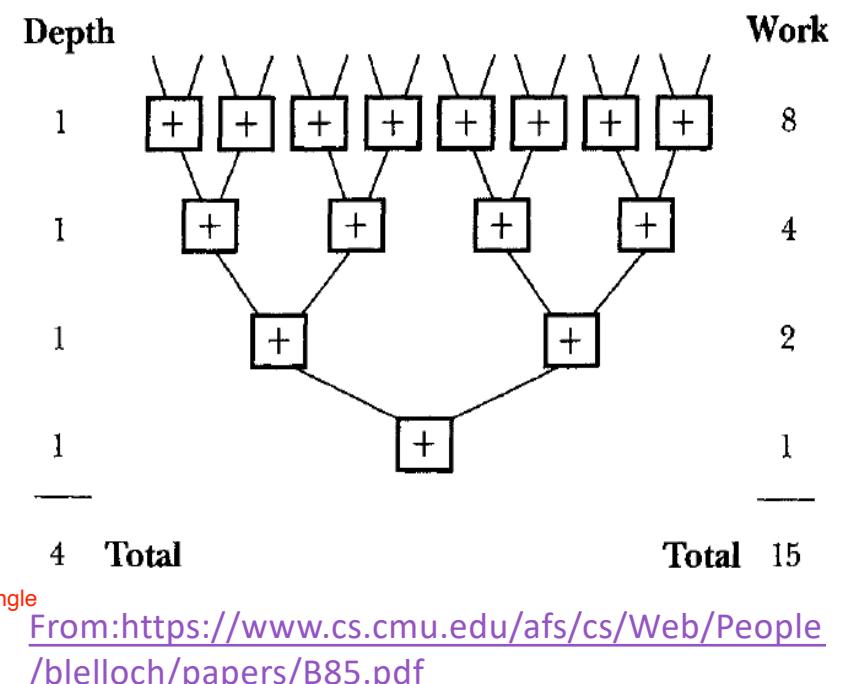
- We assume 1 time unit per compute operation
- T_1 : time units to **compute** work on a single processor
- T_∞ : time units to **compute** work on a infinite number of processors
- Relationship to work and depth:



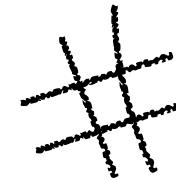
$$\begin{aligned} T_1 &= W \\ T_\infty &= D \end{aligned}$$

(As it will expose sequential dependencies)

(since the entire work is done on single
processor)



From: <https://www.cs.cmu.edu/afs/cs/Web/People/blelloch/papers/B85.pdf>



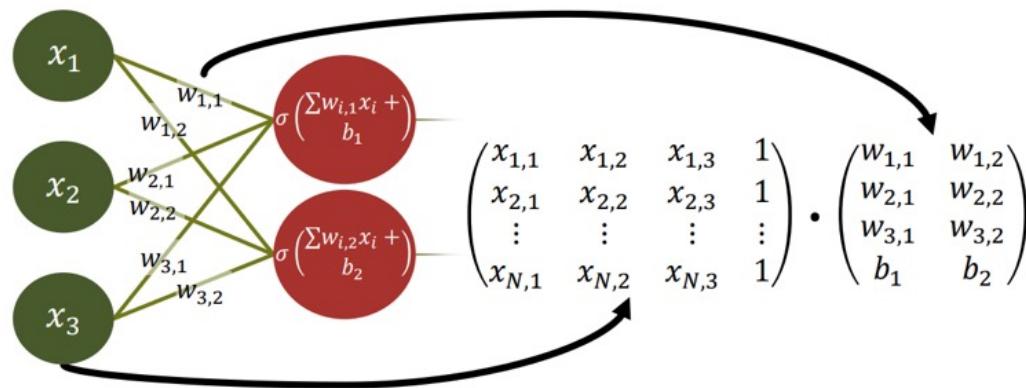
Asymptotic Work-Depth Characteristics

Operator Type	Eval.	Work (W)	Depth (D)
Activation	y	$O(NCHW)$	$O(1)$ can be done parallelly
	∇w	$O(NCHW)$	$O(1)$ No sequential dependency
	∇x	$O(NCHW)$	$O(1)$
Fully Connected	y	$O(C_{out} \cdot C_{in} \cdot N)$	$O(\log C_{in})$
	∇w	$O(C_{in} \cdot N \cdot C_{out})$	$O(\log N)$
	∇x	$O(C_{in} \cdot C_{out} \cdot N)$	$O(\log C_{out})$
Convolution (Direct)	y	$O(N \cdot C_{out} \cdot C_{in} \cdot H' \cdot W' \cdot K_x \cdot K_y)$	$O(\log K_x + \log K_y + \log C_{in})$
	∇w	$O(N \cdot C_{out} \cdot C_{in} \cdot H' \cdot W' \cdot K_x \cdot K_y)$	$O(\log K_x + \log K_y + \log C_{in})$
	∇x	$O(N \cdot C_{out} \cdot C_{in} \cdot H \cdot W \cdot K_x \cdot K_y)$	$O(\log K_x + \log K_y + \log C_{in})$
Pooling	y	$O(NCHW)$	$O(\log K_x + \log K_y)$
	∇w	—	—
	∇x	$O(NCHW)$	$O(1)$
Batch Normalization	y	$O(NCHW)$	$O(\log N)$
	∇w	$O(NCHW)$	$O(\log N)$
	∇x	$O(NCHW)$	$O(\log N)$

Name	Description
N	Minibatch size
C	Number of channels, features, or neurons
H	Image Height
W	Image Width
K_x	Convolution kernel width
K_y	Convolution kernel height

From: Ben-Nun T, Hoefler T.
Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis.

Computing Fully Connected Layers



- Matrix-matrix multiplication
- Perfect for GPU computation
- Useful efficient linear algebra libraries, such as CUBLAS and MKL

Computing Convolutional Layers

different ways to achieve convolution

- Direct Convolution
- Im2col
 - There are highly optimized matrix multiplication routines for just about every platform
 - Can we turn convolution into matrix multiplication?
 - Easy to implement, but big memory overhead or multiple BLAS calls
- FFT Fast fourier transform
 - Big speedups for large kernels
- Winograd
 - Compute minimal complexity convolution over small tiles, which makes them fast with small filters and small batch sizes

Computing Convolution

Assuming convolution of a 4D tensor with a 4D kernel:

- Input tensor (x) shape: $N \times C_{in} \times H \times W$ N = batch size
- Kernel tensor (w) shape: $C_{out} \times C_{in} \times K_y \times K_x$ cout = number of keenels
- Output tensor (y) shape: $N \times C_{out} \times H' \times W'$
- In the general case $W' = \frac{W - K_x + 2P_x}{S_x} + 1$ and $H' = \frac{H - K_y + 2P_y}{S_y} + 1$ for padding P_x, P_y and strides S_x, S_y
- Assuming zero padding and a stride of 1 we obtain $W' = W - K_x + 1$ and $H' = H - K_y + 1$

Computing Convolution - Direct Convolution

```

1: for  $i = 0$  to  $N$  in parallel do for each batch (can be done parallelly)
2:   for  $j = 0$  to  $C_{out}$  in parallel do for each filter (can be done parallelly)
3:     for  $k = 0$  to  $H'$  in parallel do height,width of input (can be done parallelly - as for every k and l there will be one entry in the output map)
4:       for  $l = 0$  to  $W'$  in parallel do
5:         for  $m = 0$  to  $C_{in}$  do for every input channel (as everything is summed up into one value, cant be done parallelly)
6:           for  $k_y = 0$  to  $K_y$  do
7:             for  $k_x = 0$  to  $K_x$  do
8:                $y_{i,j,k,l} += x_{i,m,k+k_y,l+k_x} \cdot w_{j,m,k_y,k_x}$ 
9:             end for
10:            end for
11:          end for
12:        end for
13:      end for
14:    end for
15:  end for

```

total computations -
► Depth: $\log_2 C_{in}$
► Depth: $\log_2 K_y$
► Depth: $\log_2 K_x$
► Work: 1

analogy from the depth of summation

Overall cost:

$$\mathbf{W} = N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$$

$$\mathbf{D} = [\log_2 C_{in}] + [\log_2 K_y] + [\log_2 K_x]$$

HPML

$$D = \log(C_{in} \cdot K_x \cdot K_y)$$

41

Computing Convolution - Im2Col



Image to Column

- There are highly optimized matrix multiplication routines for just about every platform
- Can we turn convolution into matrix multiplication?
- Using *Toeplitz* matrices
 - Also known as `Im2col`
- Idea:
 1. Use the `im2col` operation to transform the input image (or batch) into a matrix
 2. multiply this matrix with a reshaped version of the kernel
 3. At the end reshape this multiplied matrix back to an image with the `col2im` operation

a technique to compute convolution by matrix multiplication

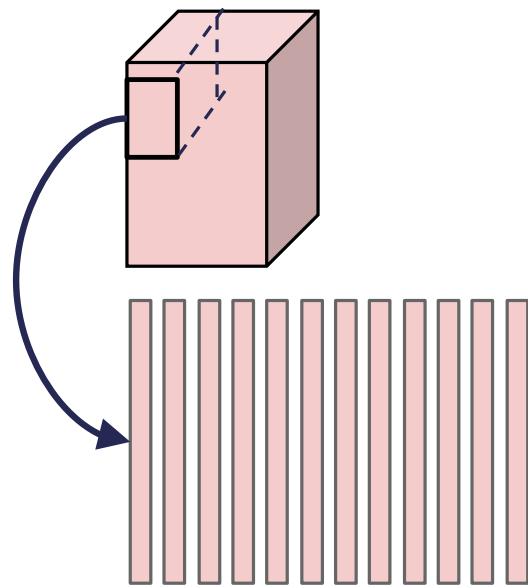
It is a technique to convert image data into a format that is more suitable for efficient computation during convolution operations.

It reshapes the input image or feature map into a matrix, where each column corresponds to a receptive field (also known as a kernel or filter window) of the convolution operation.

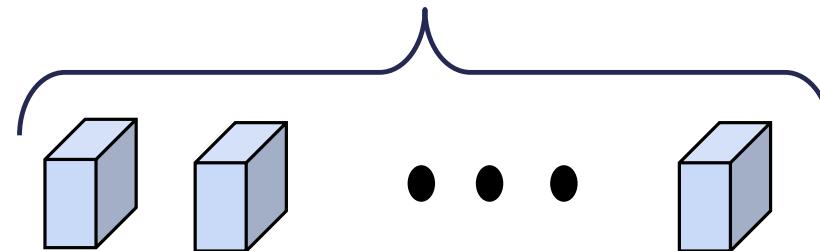
why each column is a receptive field - as it is the area swept by filter at each at each time step. We can use matrix multiplication now

Im2col Details

Feature map: $H \times W \times C$

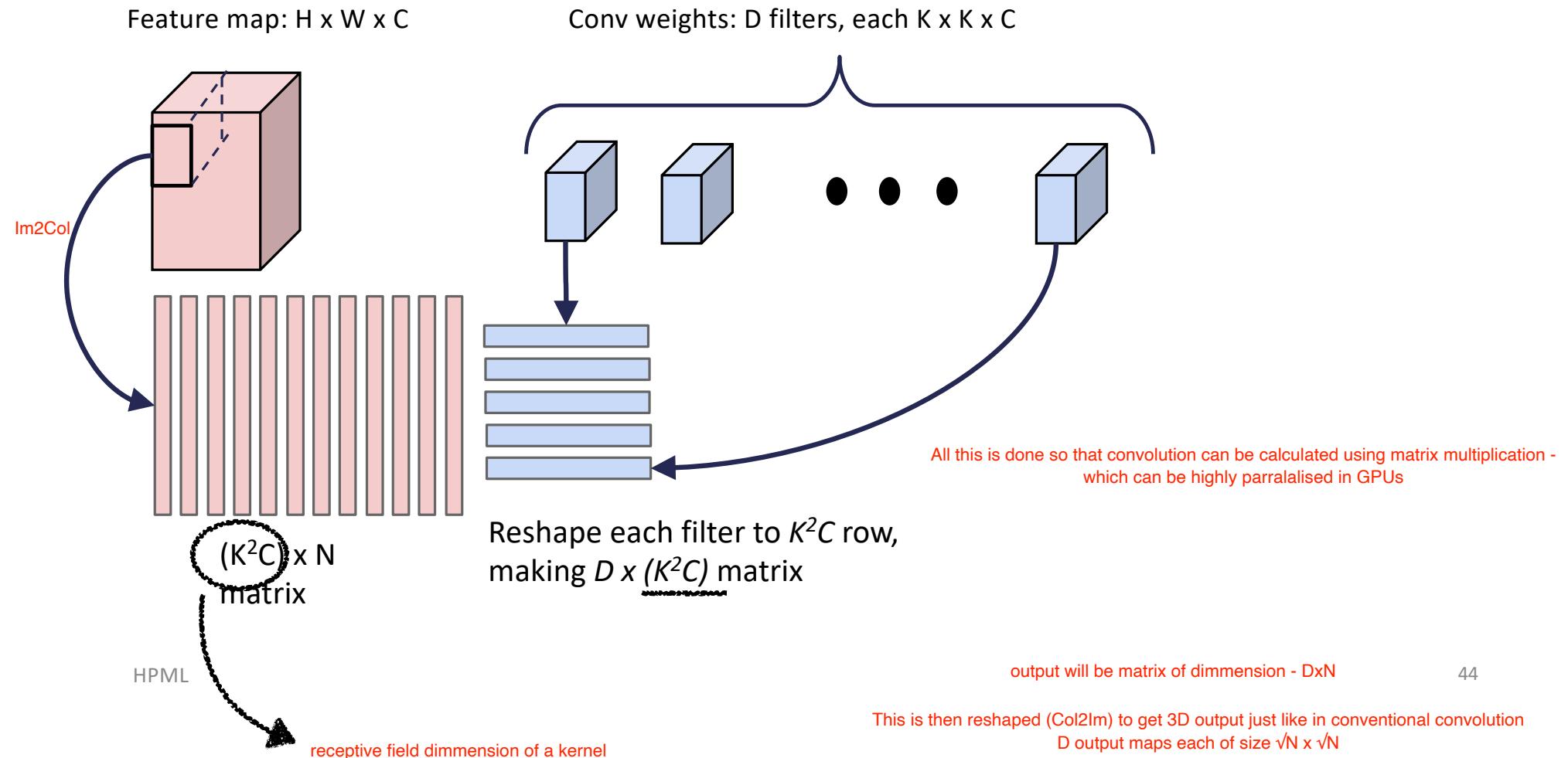


Conv weights: D filters, each $K \times K \times C$



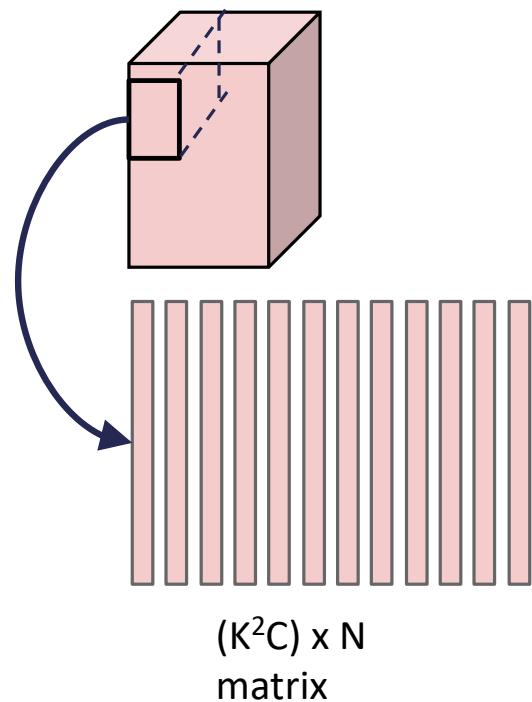
- Reshape $K \times K \times C$ receptive field to column with K^2C elements
- Repeat for all columns to get $(K^2C) \times N$ matrix
- (N receptive field locations)
- Elements appearing in multiple receptive fields are duplicated
→ this uses a lot of memory

Im2col Details

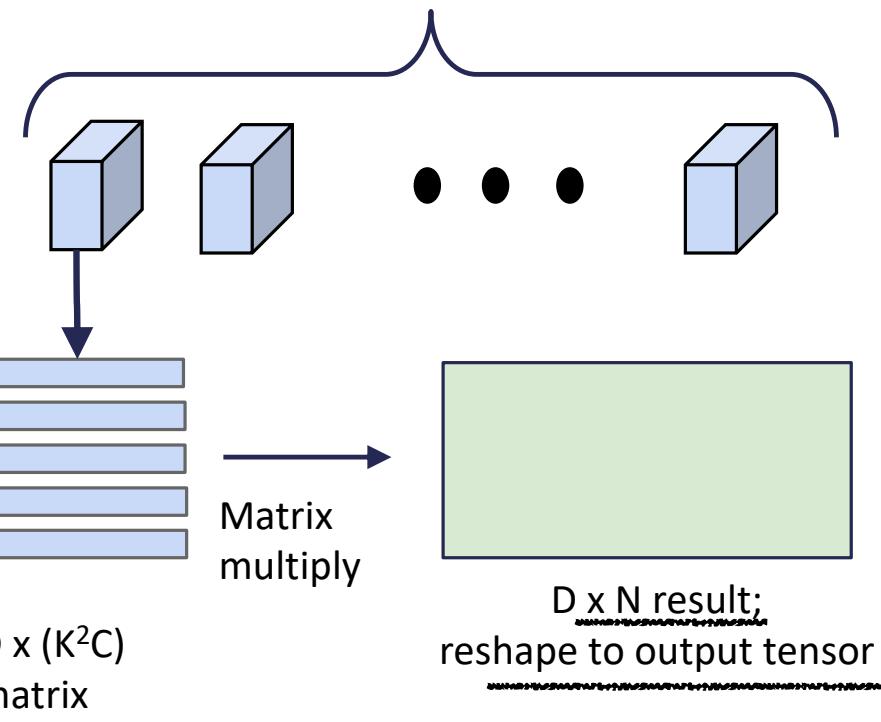


Im2col Details

Feature map: $H \times W \times C$



Conv weights: D filters, each $K \times K \times C$



Im2col – Example with 4x4 input, 3x3 filter and 2x2 output

$W=4$
 $H=4$
 $K=3$
 $W'=2$
 $H'=2$

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

=

α	β
γ	δ

$$\boxed{\beta = bq + cr + ds + ft + gu + hv + jw + kx + ly}$$

(9 multiplications per output value)
 $(9 * 4 = 36$ multiplications for all 4 output values)

A Toeplitz matrix is a special type of matrix in which each descending diagonal from left to right is constant. In other words, all the elements along any given diagonal are identical.

Toeplitz matrix form

W': output width

H': output height

K: kernel size (assumed square)

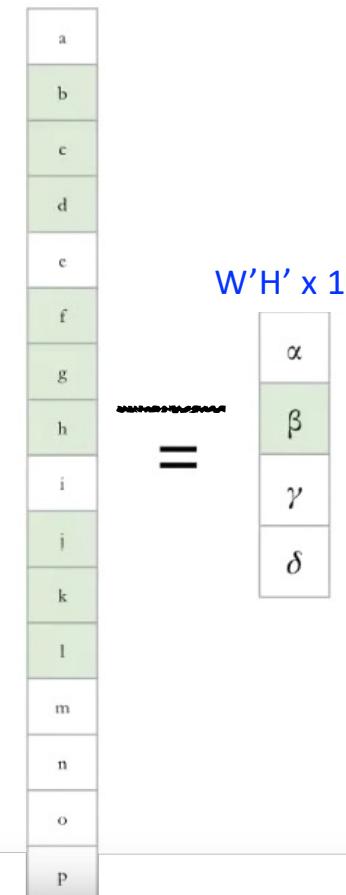
$(W'+K-1)(H'+K-1) \times 1$

$$W' H' \times (W'+K-1)(H'+K-1)$$

q	r	s	t	u	v	w	x	y				
	q	r	s	t	u	v	w	x	y			
			q	r	s	t	u	v	w	x	y	
			q	r	s	t	u	v	w	x	y	

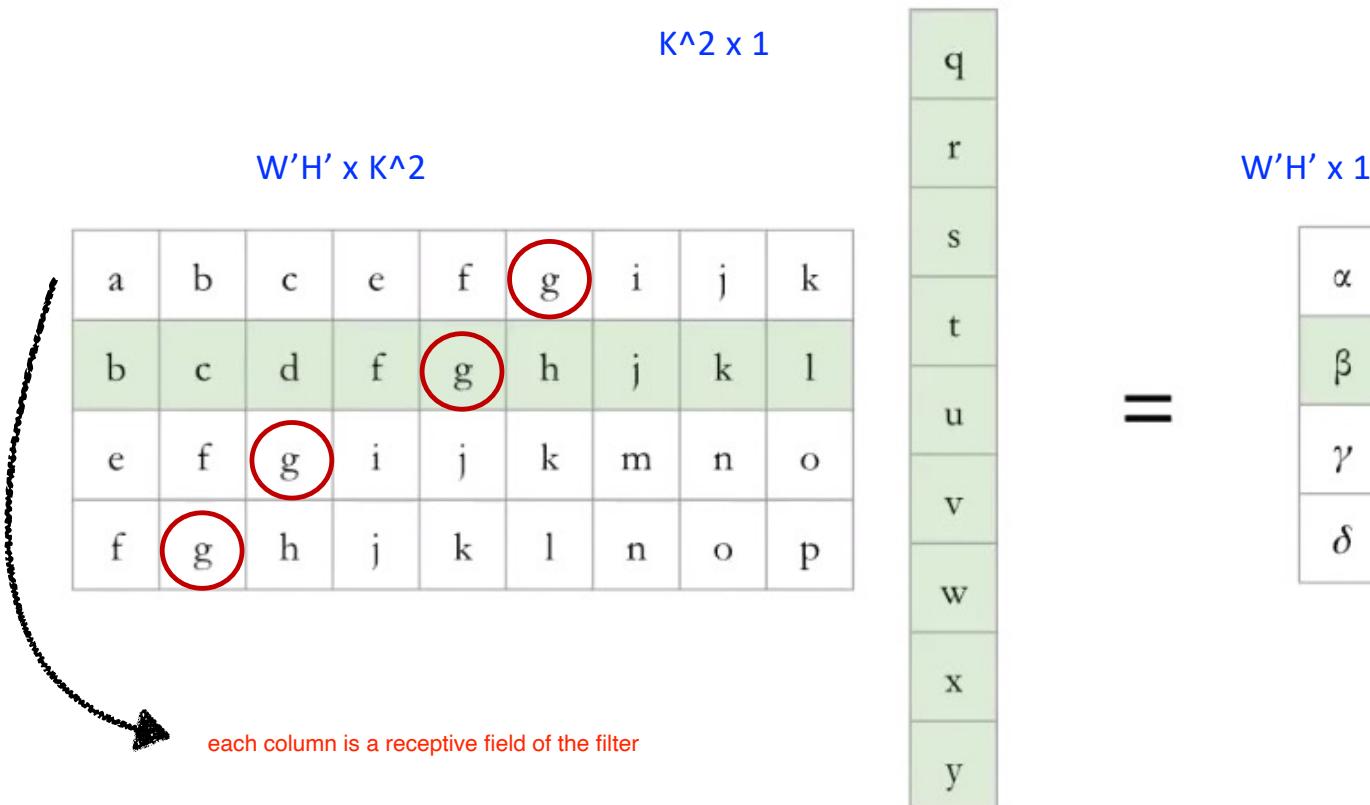
$$\beta = bq + cr + ds + ft + gu + hv + jw + kx + ly$$

(9 multiplications per output value)
 $(9 * 4 = 36$ multiplications for all 4 output values)



Dense matrix re-organization (used in caffe for convolution)

As K^2 is the dimmension of the filter

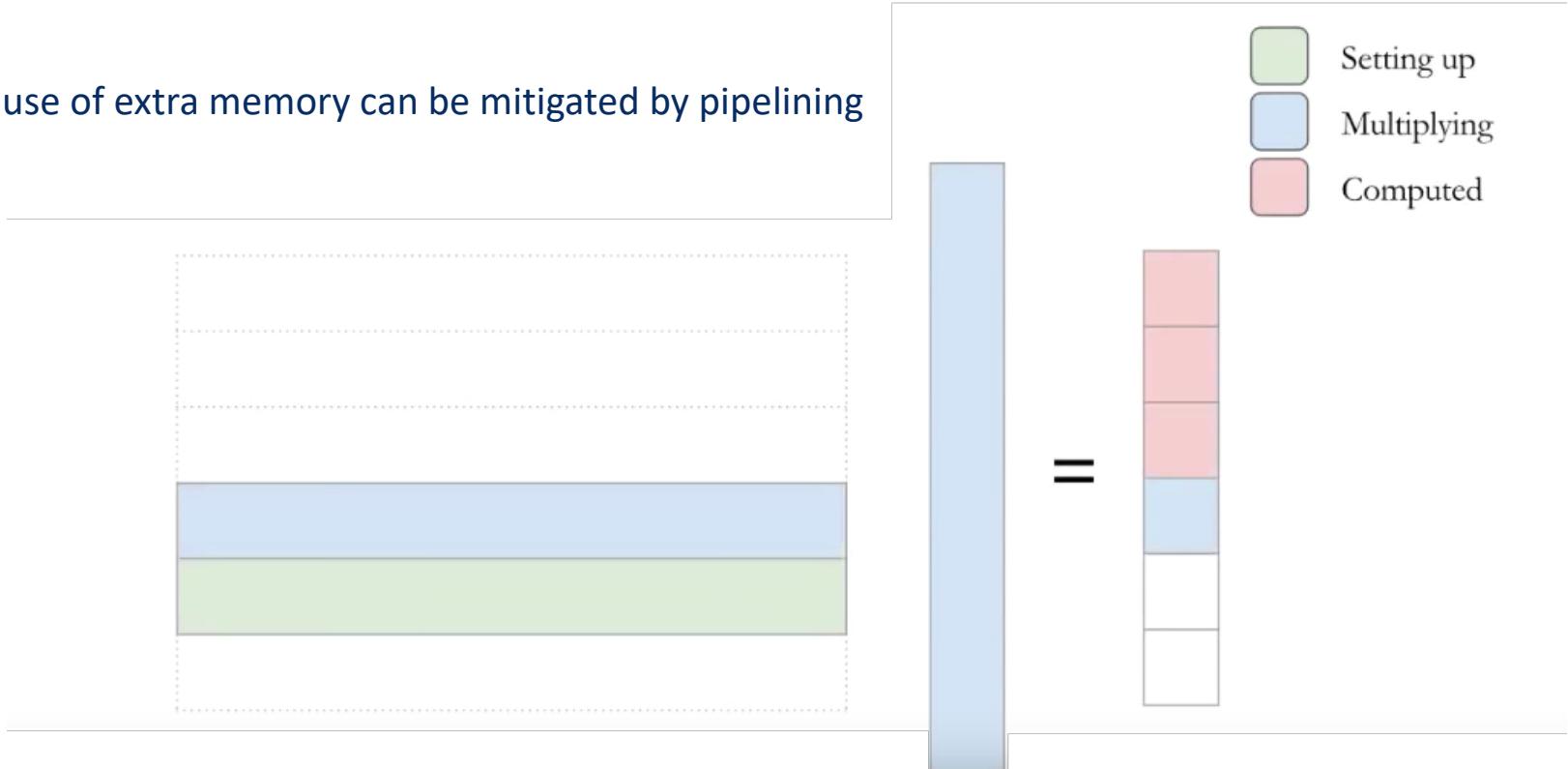


- why?
- At worse, each element is repeated K^2 times
 - Each dot product correspond to the (non-parallelizable) loop in line 5-7 in the direct convolution algorithm
 - Can be parallelized intelligently

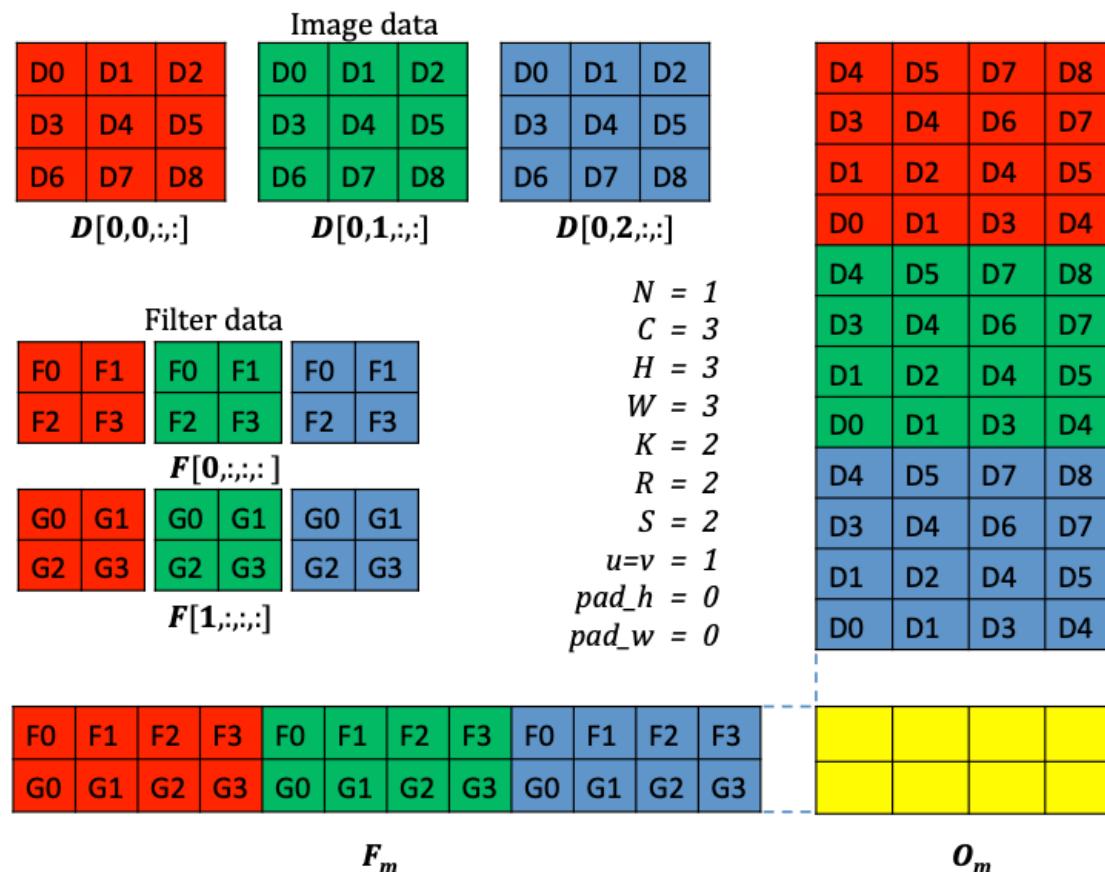
Pipelining Computation

~

- The use of extra memory can be mitigated by pipelining



Im2col - Example with 3 channels



Im2Col Implementation

- Input matrix A is a result of a data-layout transformation, sized $(C_{in} \cdot K_y \cdot K_x) \times (N \cdot H' \cdot W')$
- Kernel matrix F is the reshaped tensor w, with dimensions $C_{out} \times C_{in} \cdot K_y \cdot K_x$
- Output matrix B has a size of $C_{out} \times (N \cdot H' \cdot W')$ and is reshaped to the output
- Overall cost:

$$W = N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$$

$$D = \lceil \log_2 C_{in} \rceil + \lceil \log_2 K_y \rceil + \lceil \log_2 K_x \rceil$$

```

1: for  $i = 0$  to  $C_{in} \cdot K_y \cdot K_x$  in parallel do
2:   for  $j = 0$  to  $N \cdot H' \cdot W'$  in parallel do
3:      $A_{i,j} \leftarrow x\dots$ 
4:   end for
5: end for
6:
7:  $B \leftarrow F \cdot A$ 
8:
9: for  $i = 0$  to  $N$  in parallel do
10:   for  $j = 0$  to  $C_{out}$  in parallel do
11:     for  $k = 0$  to  $H'$  in parallel do
12:       for  $l = 0$  to  $W'$  in parallel do
13:          $y_{i,j,k,l} \leftarrow B\dots$ 
14:       end for
15:     end for
16:   end for
17: end for

```

► im2col. Work: N/A, Depth: N/A (layout only)

► Matrix Multiplication
 ► Work: $C_{out} \cdot (C_{in} \cdot K_y \cdot K_x) \cdot (N \cdot H' \cdot W')$
 ► Depth: $\log_2 (C_{in} \cdot K_y \cdot K_x)$

► col2im. Work: N/A, Depth: N/A (layout only)

Im2Col

- The GEMM method (while processor-friendly) consumes a considerable amount of memory
→ not scalable
- Practical implementations of the GEMM method, such as in CUDNN, implement “implicit GEMM”, in which the Toeplitz matrix is never materialized
- Using Strassen matrix multiplication¹ could reduce the number of operations by up to 47%
 - Use clever arithmetic to reduce complexity to $O(N^{\log_2(7)}) \sim O(N^{2.81})$

¹ V. Strassen. 1969. Gaussian Elimination is Not Optimal. Numer. Math. 13, 4 (1969), 354–356.

Matrix Multiplication using Block Matrices

- Let A and B be two square matrices and C=AxB
- Say A, B, C are matrices of size $2^n \times 2^n$
- Strassen algorithm first partitions them into equally sized block matrices

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

where A_{ij} , B_{ij} , C_{ij} are matrices of size $2^{n-1} \times 2^{n-1}$

How many multiplications are needed to calculate C?



Strassen Algorithm for Matrix Multiplication

- Define new matrices

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22});$$

$$M_2 = (A_{21} + A_{22})B_{11};$$

$$M_3 = A_{11}(B_{12} - B_{22});$$

$$M_4 = A_{22}(B_{21} - B_{11});$$

$$M_5 = (A_{11} + A_{12})B_{22};$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12});$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22}),$$

recursively iterate this division process until
the submatrices degenerate into numbers

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

Uses only 7 multiplications (one for each M_k) instead of 8

.....

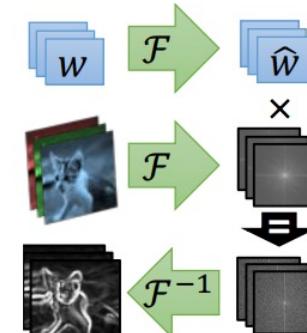
https://en.wikipedia.org/wiki/Strassen_algorithm

Computing Convolution - FFT

- Convolution Theorem: The fourier transform of convolution of f and g is equal to the element-wise product of their Fourier Transforms:

$$\mathcal{F}(f * g) = \mathcal{F}(f) \circ \mathcal{F}(g)$$

- where \mathcal{F} denotes the Fourier Transform and \circ is element-wise multiplication
- Using the **Fast Fourier Transform**, we can compute the Discrete Fourier transform of an N-dimensional vector in $O(N \log N)$ time
- FFT Process:
 1. Compute FFT of weights: $\mathcal{F}(W)$
 2. Compute FFT of image: $\mathcal{F}(X)$
 3. Compute elementwise product: $\mathcal{F}(W) \circ \mathcal{F}(X)$
 4. Compute inverse FFT: $Y = \mathcal{F}^{-1}(\mathcal{F}(W) \circ \mathcal{F}(X))$



FFT

- \hat{x} and \hat{w} are the transformed inputs and kernels reshaped as 3D tensors, obtained by batching 2-dimensional FFTs
- \hat{x}_k and \hat{w}_k are the k-th 2D slice in the tensor
- In practical implementations, \hat{x} and \hat{w} are reshaped to $W \cdot H$ 2D matrices (sized $N \times C_{in}$ and $C_{in} \times C_{out}$ respectively) to transform the point-wise multiplication and sum to a batched complex matrix-matrix multiplication

Algorithm 5 FFT Convolution

```

1: for  $i = 0$  to  $C_{out}$  in parallel do
2:   for  $j = 0$  to  $C_{in}$  in parallel do
3:      $\hat{w}_{i,j} \leftarrow \mathcal{F}(w_{i,j})$ 
4:   end for
5: end for
6: for  $i = 0$  to  $N$  in parallel do
7:   for  $j = 0$  to  $C_{in}$  in parallel do
8:      $\hat{x}_{i,j} \leftarrow \mathcal{F}(x_{i,j})$ 
9:   end for
10: end for
11: for  $i = 0$  to  $N$  in parallel do
12:   for  $j = 0$  to  $C_{out}$  in parallel do
13:     for  $k = 0$  to  $H'$  in parallel do
14:       for  $l = 0$  to  $W'$  in parallel do
15:          $\hat{y}_{i,j} = \sum_{m=0}^{C_{in}} \hat{x}_{i,m} \cdot \hat{w}_{j,m}$ 
16:       end for
17:     end for
18:   end for
19: end for
20: for  $i = 0$  to  $N$  in parallel do
21:   for  $j = 0$  to  $C_{out}$  in parallel do
22:      $y_{i,j} \leftarrow \mathcal{F}^{-1}(\hat{y}_{i,j})$ 
23:   end for
24: end for

```

► 2D FFT. Work: $c \cdot HW \log_2 HW$, Depth: $\log_2 HW$

► 2D FFT. Work: $c \cdot HW \log_2 HW$, Depth: $\log_2 HW$

► Batched MM

► Work: $H \cdot W \cdot N \cdot C_{in} \cdot C_{out}$

► Depth: $\log_2 C_{in}$

► 2D IFFT. Work: $c \cdot H'W' \log_2 H'W'$, Depth: $\log_2 H'W'$

FFT: ↗speedup for ↗filters

- FFT convolutions get a big speedup for larger filters
- Not much speedup for 3x3 filters

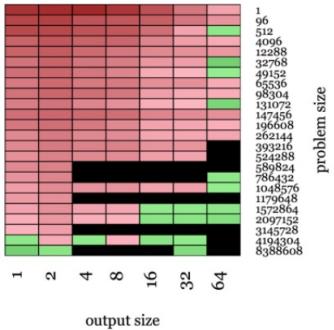


Figure 1: 3×3 kernel (K40m)

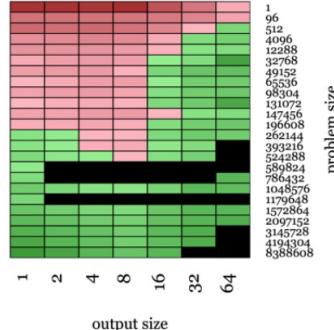


Figure 2: 5×5 kernel (K40m)

speedup is more for larger filters

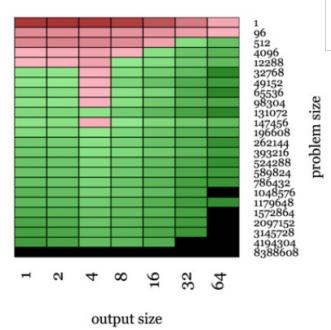


Figure 3: 7×7 kernel (K40m)

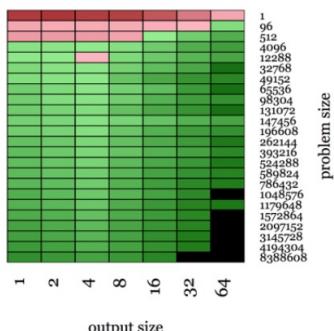


Figure 4: 9×9 kernel (K40m)

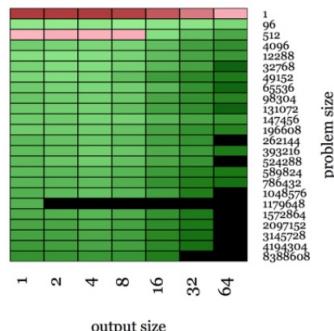


Figure 5: 11×11 kernel (K40m)

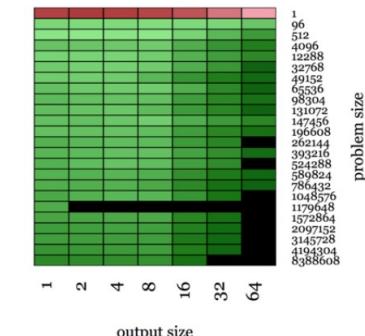
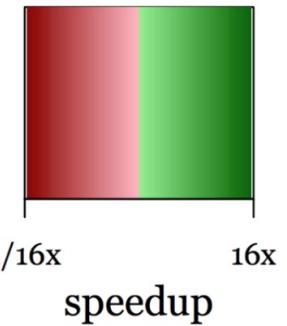
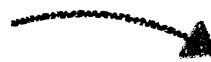


Figure 6: 13×13 kernel (K40m)



Minimal Filtering Algorithm

- The minimal filtering algorithm for computing m outputs with an r -tap filter, which we call $F(m, r)$, requires

$$\mu(F(m, r)) = \underline{m + r - 1}$$


size of the input

multiplications

- The minimal algorithm requires a number of multiplications equal to the number of inputs

Computing Convolution: Winograd minimal filtering algorithm

- Winograd (and its improvements) is the prevalent method used today to perform convolutions
- Very well suited (and only used) for convolution with small filters (e.g., 3×3)
 - Small filters are very extensively used in the latest CNNs architectures
- The number of operations in Winograd convolutions grows quadratically with filter size
- Algorithmic strength reduction: Reduce the number of expensive ops (multiplications) by increasing the number of inexpensive ops (adds)
- Process:
 - Winograd minimal filtering algorithm works on one image tile at a time
 - The tile size and filter size are comparably small
 - The main steps consist of transformation of image tile and filter, element wise product and inverse transformation of the product tile to obtain the output of convolution for that tile
 - This algorithm has a slightly reduced arithmetic complexity compared to direct methods, which makes a huge difference asymptotically and practically in deep CNNs

Winograd - Minimal filtering algorithm

$F(m,r)$

- Input size = $m + r - 1$
 - 4 in this example (d_0, d_1, d_2, d_3)
- Output size = m
 - 2 in this example
- Filter size = r
 - 3 in this example
- In a direct convolution:
 - $O_0 = d_0 * g_0 + d_1 * g_1 + d_2 * g_2$
 - $O_1 = d_1 * g_0 + d_2 * g_1 + d_3 * g_2$
 - need 6 MULs and 4 ADDs:

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

where

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

Data (d's): 4 ADDs
 Filter (g's): 3 ADDs, 2 MULs
 Depend only on filter!
 Why 3 ADDs?
 Outputs (m's): 4 MULs, 4 ADDs

Matrix form for 1-d Winograd filtering

$$Y = A^T [(Gg) \odot (B^T d)]$$

where \odot indicates element-wise multiplication.

For F(2,3) the values are:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = [g_0 \quad g_1 \quad g_2]^T$$

$$d = [d_0 \quad d_1 \quad d_2 \quad d_3]^T$$

Matrix form for 2-d Winograd Filtering

$$Y = A^T [U \odot V] A$$

with $U = GgG^T$ and $V = B^T dB$

Nesting Minimal Filtering Algorithms

A minimal 1D algorithm $F(m, r)$ is nested with itself to obtain a minimal 2D algorithm, $F(m \times m, r \times r)$ like so:

$$Y = A^T \left[[GgG^T] \odot [B^T dB] \right] A \quad (8)$$

where now g is an $r \times r$ filter and d is an $(m + r - 1) \times (m + r - 1)$ image tile. The nesting technique can be generalized for non-square filters and outputs, $F(m \times n, r \times s)$, by nesting an algorithm for $F(m, r)$ with an algorithm for $F(n, s)$.

$F(2 \times 2, 3 \times 3)$ uses $4 \times 4 = 16$ multiplications, whereas the standard algorithm uses $2 \times 2 \times 3 \times 3 = 36$. This is an arithmetic complexity reduction of $\frac{36}{16} = 2.25$. The data transform uses 32 additions, the filter transform uses 28 floating point instructions, and the inverse transform uses 24 additions.

$$I_{\text{padded}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

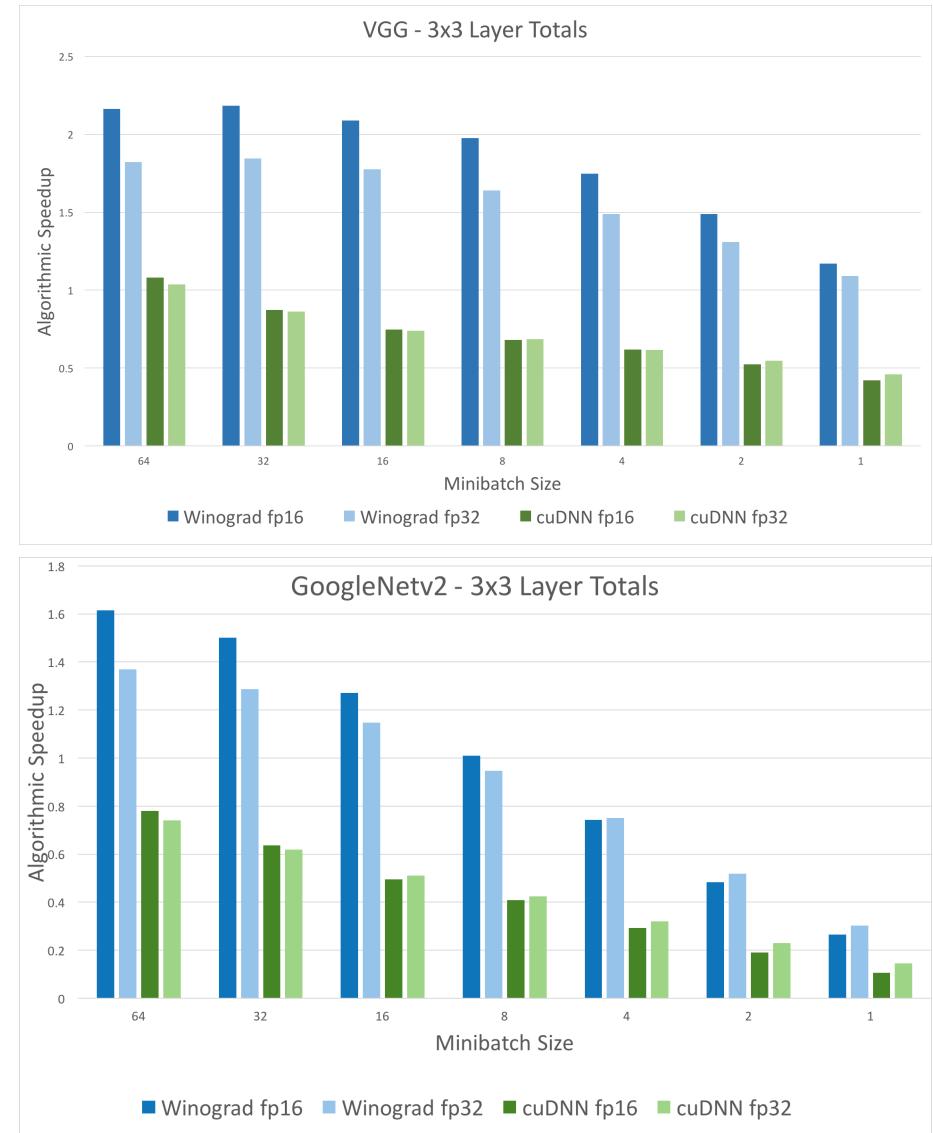
$F(2 \times 2, 3 \times 3)$

$$\text{out} = \begin{bmatrix} o_0 & o_1 \\ o_2 & o_3 \end{bmatrix} \quad \text{filter} = \begin{bmatrix} 1 & 2 & 1 \\ 1 & 2 & 1 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\text{input size} = (m + r - 1) \times (m + r - 1) = 4 \times 4$$

FFT vs Winograd

- Architectures:
 - VGG16: all 3x3 conv layers
 - GoogleNetv2: dominant small (1x1 and 3x3) conv layers
- Benchmarked speed on:
 - NVIDIA Titan X GPU (fixed at 1 Ghz)
 - Intel® Core™ i7 CPU 975 @ 3.33GHz
- Depending on the network architecture, training with Winograd algorithm yields speed-ups of **2-3x** over NVIDIA's cuDNN v4 kernels
- From <https://ai.intel.com/winograd/>



Computing Convolution - Comparison

Method	Work (W)	Depth (D)
Direct	$N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$	$\lceil \log_2 C_{in} \rceil + \lceil \log_2 K_y \rceil + \lceil \log_2 K_x \rceil$
im2col	$N \cdot C_{out} \cdot H' \cdot W' \cdot C_{in} \cdot K_y \cdot K_x$	$\lceil \log_2 C_{in} \rceil + \lceil \log_2 K_y \rceil + \lceil \log_2 K_x \rceil$
FFT	$c \cdot HW \log_2(HW) \cdot (C_{out} \cdot C_{in} + N \cdot C_{out}) + HWN \cdot C_{in} \cdot C_{out}$	$2 \lceil \log_2 HW \rceil + \lceil \log_2 C_{in} \rceil$
Winograd ($m \times m$ tiles, $r \times r$ kernels)	$\alpha(r^2 + \alpha r + 2\alpha^2 + \alpha m + m^2) + C_{out} \cdot C_{in} \cdot P$ ($\alpha \equiv m - r + 1$, $P \equiv N \cdot \lceil H/m \rceil \cdot \lceil W/m \rceil$)	$2 \lceil \log_2 r \rceil + 4 \lceil \log_2 \alpha \rceil + \lceil \log_2 C_{in} \rceil$

- Work-Depth Analysis of Convolution Implementations
- Work and Depth metrics are not always sufficient to reason about absolute performance
 - the Direct and im2col methods exhibit the same concurrency characteristics, even though im2col is faster in many cases, due to high processor utilization and memory reuse (e.g., caching) opportunities
- DNN primitive libraries, such as CUDNN and MKL-DNN, provide a variety of convolution methods and data layouts. In order to assist users in a choice of algorithm, such libraries provide functions that choose the best-performing algorithm given tensor sizes and memory constraints.
 - Internally, the libraries may run all methods and pick the fastest one