Convolutional Neural Networks in practice

Giacomo Domeniconi, Ulrich Finkler

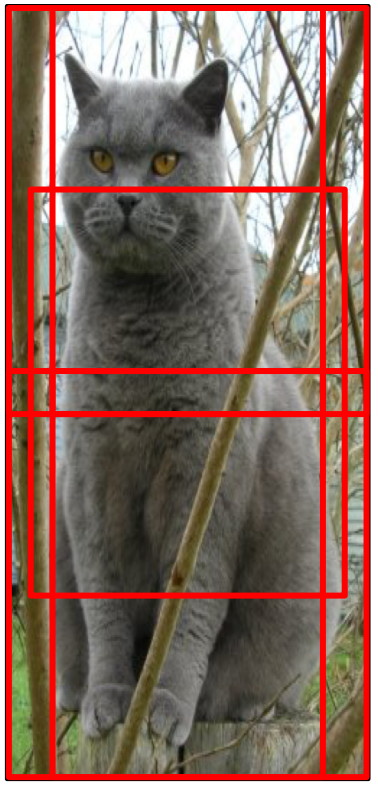
CSCI-GA.3033-022 HPML

Working with CNNs in practice

- Making the most of your data
  - Data augmentation
  - Transfer learning
  - Simple to implement…use it!
  - Especially useful for small datasets
- How to arrange conv layers?
- How to compute conv layers fast?

Data Augmentation - Random crops/scales

- **Training**: sample random crops / scales
  - Pick random L in range [256, 480]
  - Resize training image, short side = L
  - Sample random 224 x 224 patch
- **Testing**: average a fixed set of crops
  - Resize image at 5 scales: {224, 256, 384, 480, 640}
  - For each size, use 10 224 x 224 crops: 4 corners + center, + flips

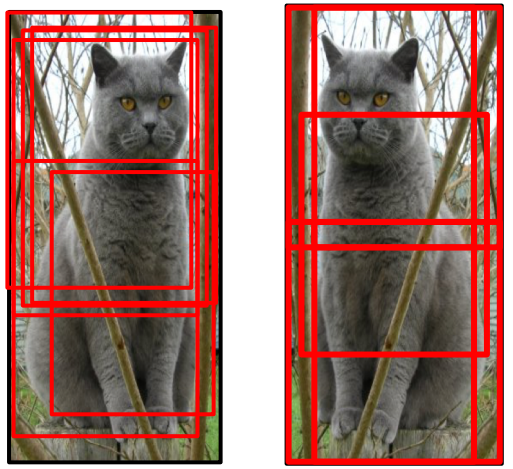


Data Augmentation

- Horizontal Flip
- Jitter
  - Simple: Randomly jitter contrast
  - Complex:
    - Apply PCA to all RGB pixels in training set
    - Sample a “color offset” along principal component directions
    - Add offset to all pixels of a training image
- Random mix/combinations of :
  - Translation
  - Rotation
  - Stretching
  - Shearing
  - Lens distortions
  - ….Go crazy!

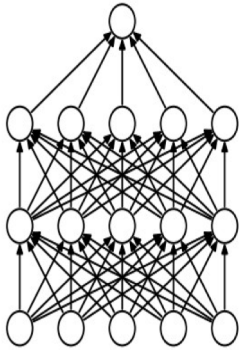
A general Approach

- Training: Add random noise
- Testing: Marginalize the noise

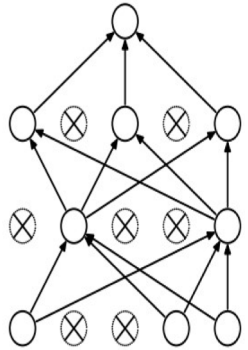


How to generalize the models?

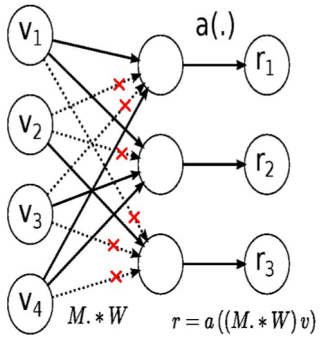
- Data Augmentation
- Dropout
- DropConnect
- Batch normalization
- Model ensembles



(a) Standard Neural Net



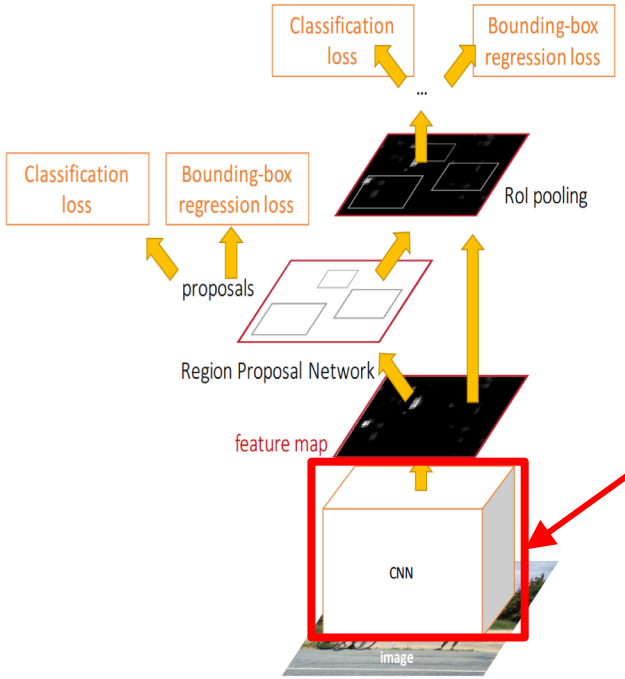
(b) After applying dropout.



Transfer Learning

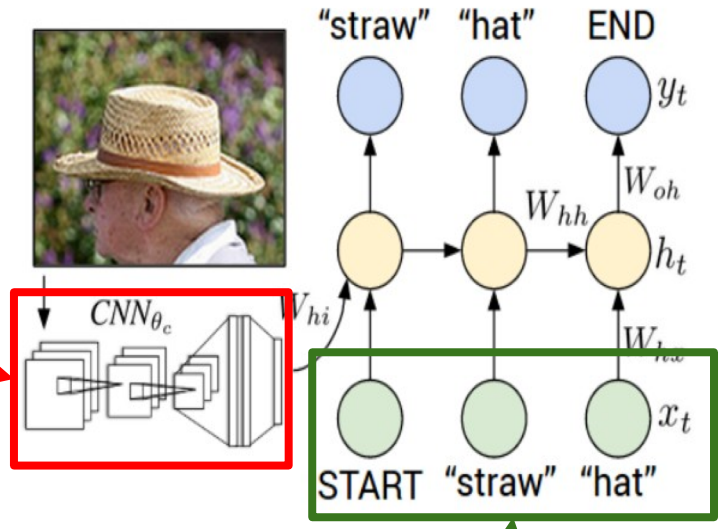
- It is pervasive and it is the norm!

Object Detection (Faster R-CNN)



CNN pretrained on ImageNet

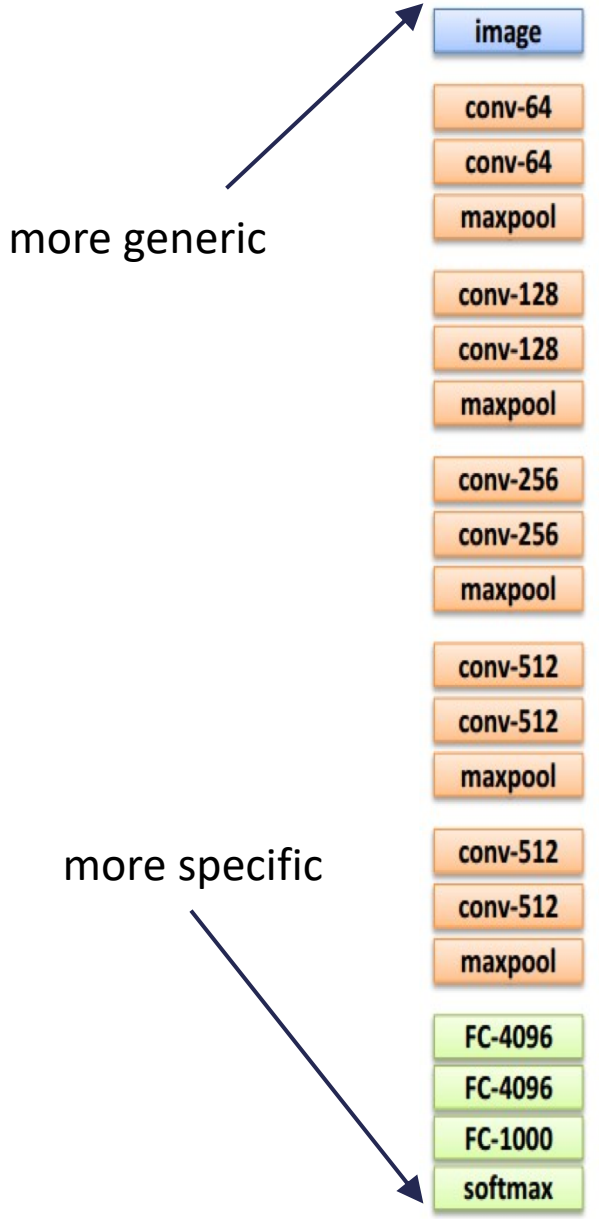
Image Captioning: CNN+RNN



Word vectors pretrained from word2vec

Transfer Learning

	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble…..
quite a lot of data	Fine-tune a few layers	Fine-tune a larger number of layers



How to stack convolutions

- Suppose we stack two 3 x 3 conv layers (stride 1)
- Each neuron sees 3 x 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

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

→7 x 7

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

The power of small filters

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

one CONV with 7 x 7 filters

Number of weights:

= C x (7 x 7 x C) = **49 C^2**

three CONV with 3 x 3 filters

Number of weights:

= 3 x C x (3 x 3 x C) = **27 C^2**

Fewer parameters, more nonlinearity

Number of multiply-adds:

= (H x W x C) x (7 x 7 x C)

= **49 HWC^2**

Number of multiply-adds:

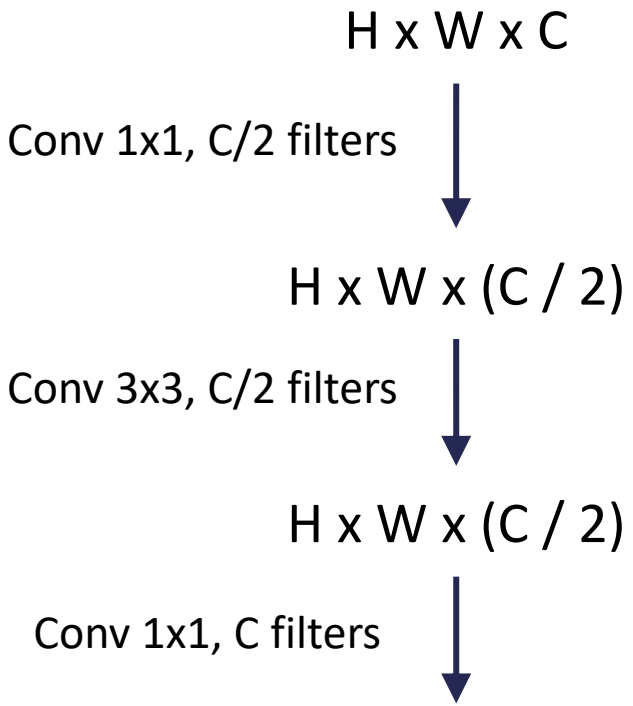
= 3 x (H x W x C) x (3 x 3 x C)

= **27 HWC^2**

Less compute, more nonlinearity

The power of small filters

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

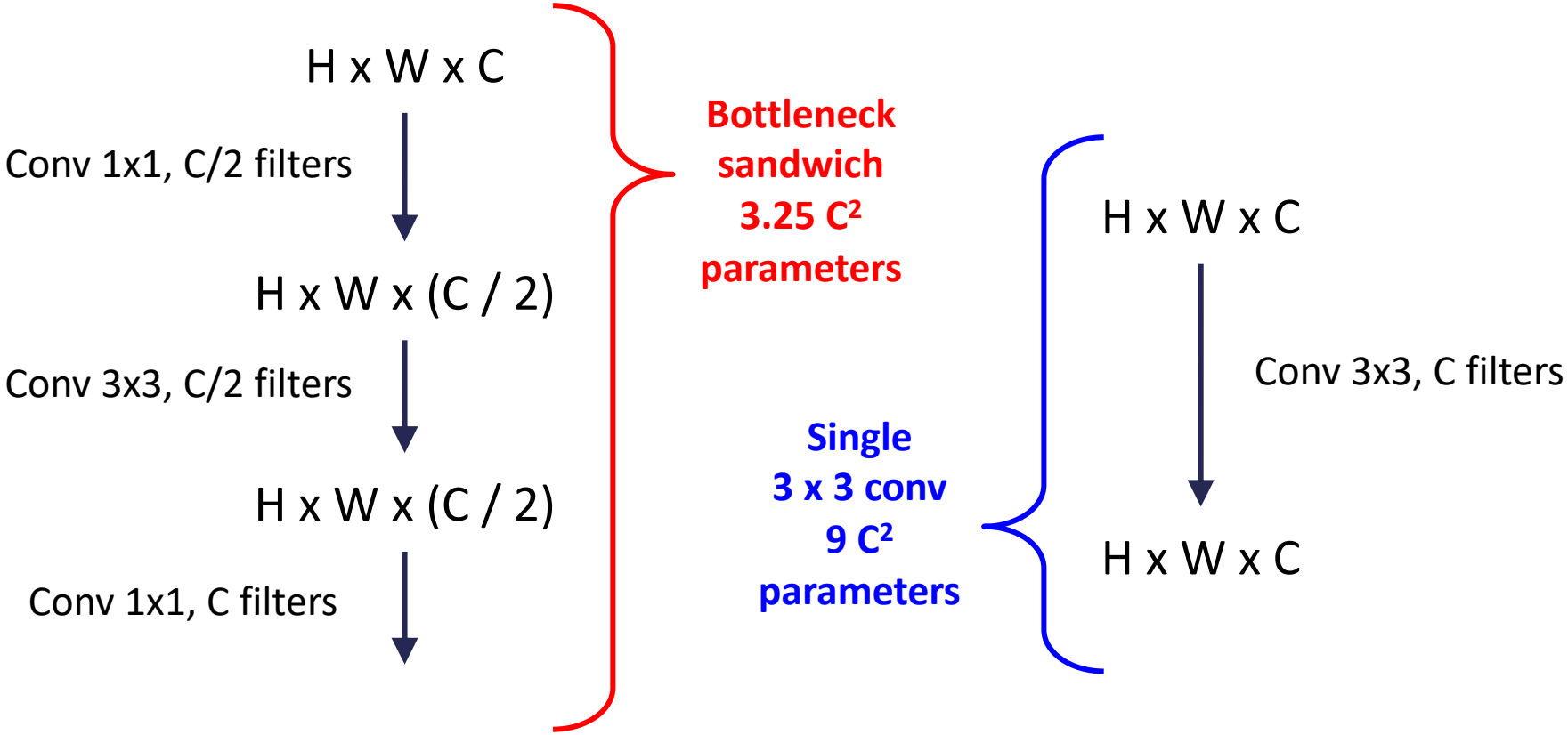


1. "bottleneck" 1 x 1 conv to reduce dimension
2. 3 x 3 conv at reduced dimension
3. Restore dimension with another 1 x 1 conv

[Seen in Lin et al, "Network in Network", GoogLeNet, ResNet]

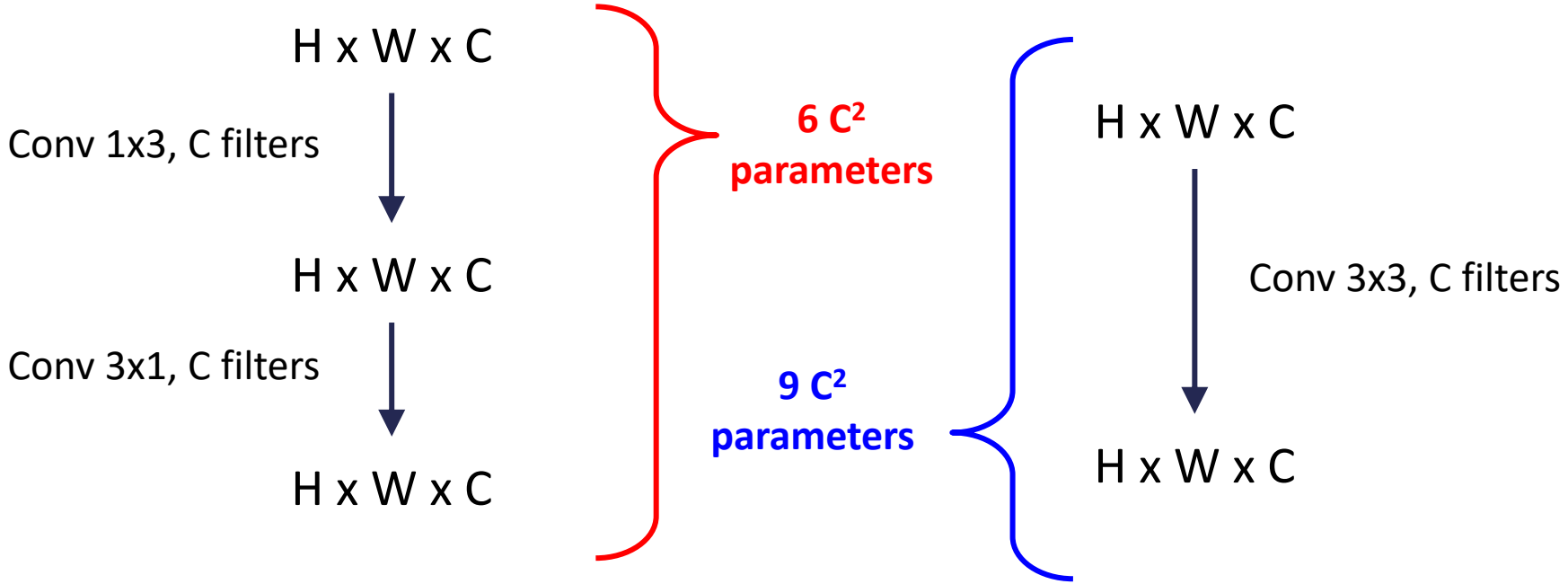
Bottleneck vs single conv

- More nonlinearity, fewer parameters, less compute!



The power of small filters

- Still using 3 x 3 filters … can we break it up?



How to stack convolutions - Recap

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

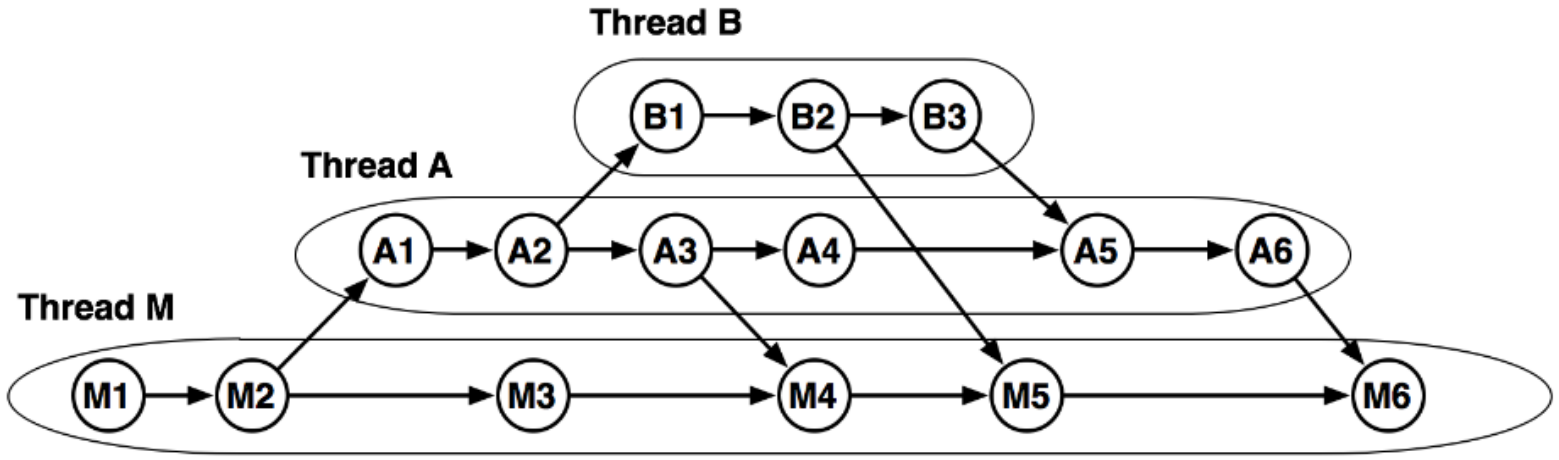
Computing NN Layers

Work and Depth Cost Model

- Cost Model for **Parallel Algorithms** (ignores communication cost) executing on multiple processors
- **Assumptions:**
  - Communication time among processors ignored
  - Memory accesses ignored
  - Processors activity is clock-synchronized
  - Computations steps are constant size

Work and Depth Cost Model

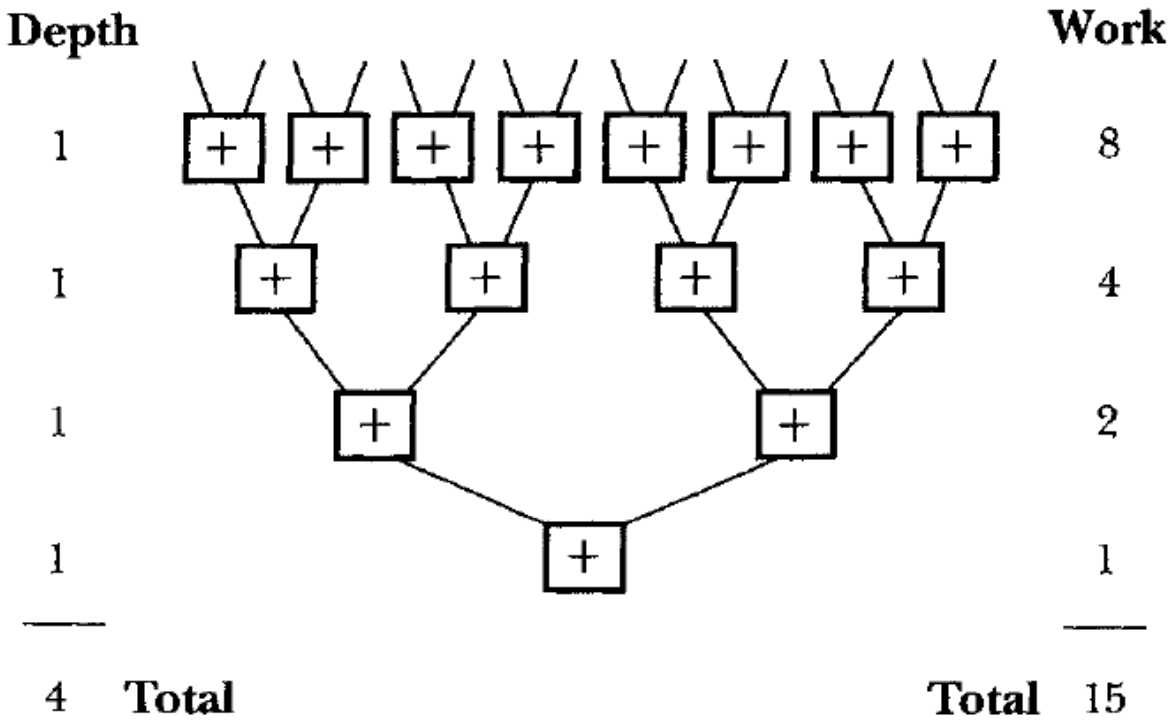
- The **execution of a parallel algorithm** can be represented by a directed acyclic graph - **DAG**
- Directed edges represent **data dependencies**
- Vertices represent a **computation**



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

Work 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



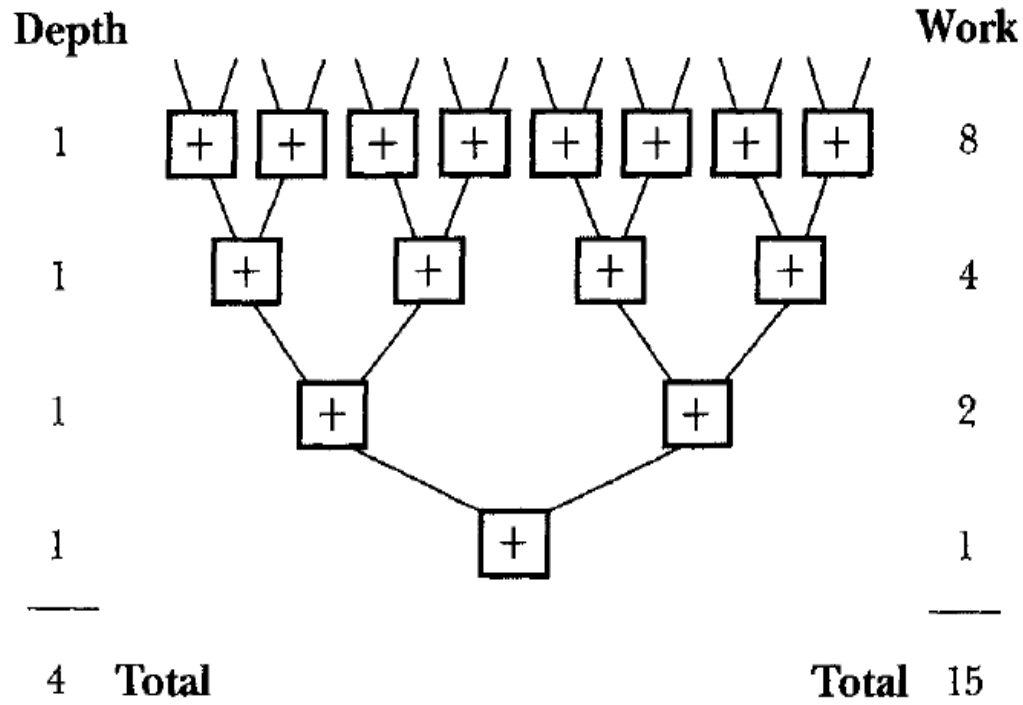
From:https://www.cs.cmu.edu/afs/cs/Web/People/blelloch/papers/B85.pdf

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:

$$T_1 = W$$
$$T_\infty = D$$



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

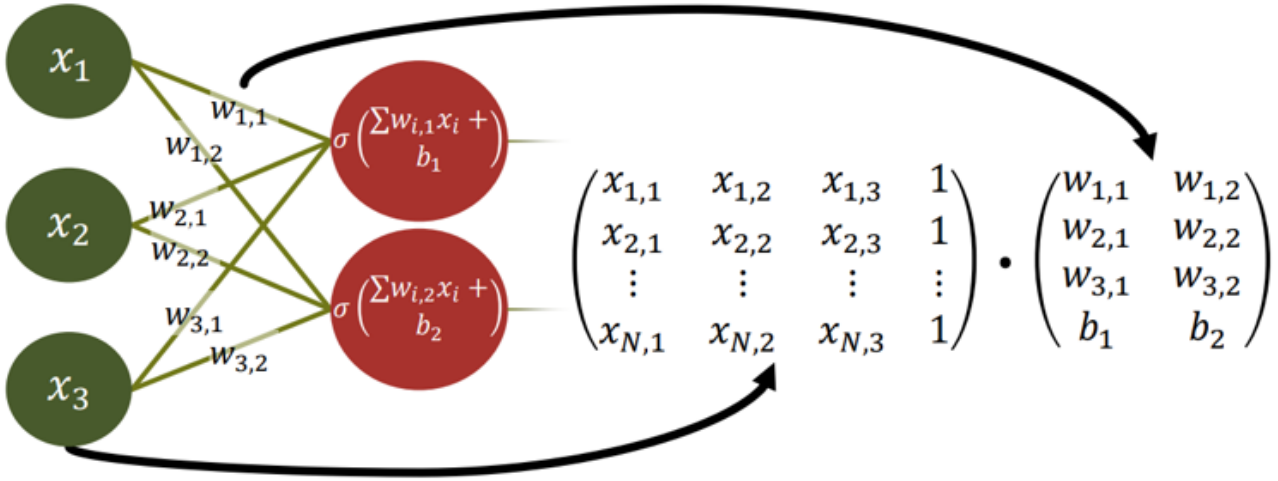
Asymptotic Work-Depth Characteristics

Operator Type	Eval.	Work (W)	Depth (D)
Activation	y	$O(NCHW)$	$O(1)$
	∇w	$O(NCHW)$	$O(1)$
	∇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
- The BLAS GEneral Matrix-Matrix multiplication (GEMM) operator includes scalar factors that enable matrix scaling and accumulation, which can be used when batching groups of neurons

Computing Convolutional Layers

- 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
  - 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: $N \times C_{in} \times H \times W$
- Kernel tensor (w) shape: $C_{out} \times C_{in} \times K_y \times K_x$
- Output tensor (y) shape: $N \times C_{out} \times H' \times W'$
- In the general case $W' = \frac{W - Kx + 2Px}{Sx} + 1$ and $H' = \frac{H - Ky + 2Py}{S_y} + 1$ 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
2:      for j = 0 to C_out in parallel do
3:          for k = 0 to H' in parallel do
4:              for l = 0 to W' in parallel do
5:                  for m = 0 to C_in do                          ▷ Depth: log2 C_in
6:                      for k_y = 0 to K_y do                     ▷ Depth: log2 K_y
7:                          for k_x = 0 to K_x do                 ▷ Depth: log2 K_x
8:                              y_{i,j,k,l} += x_{i,m,k+k_y,l+k_x} · w_{j,m,k_y,k_x}   ▷ Work: 1
9:                          end for
10:                     end for
11:                 end for
12:             end for
13:         end for
14:     end for
15: end for
```

Overall cost:

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

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

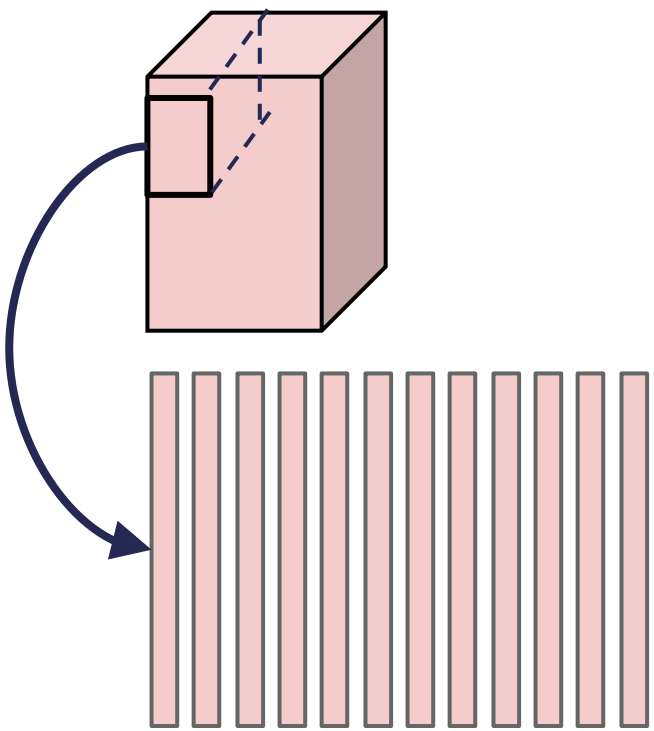
Computing Convolution - Im2Col

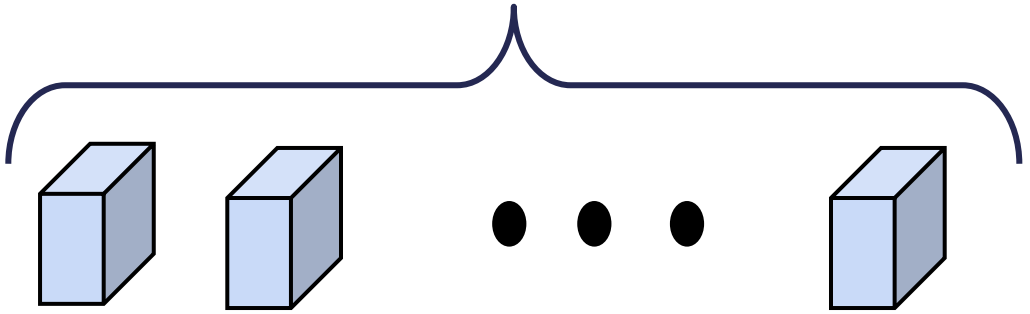
- There are highly optimized matrix multiplication routines for just about every platform
- Can we turn convolution into matrix multiplication?
- Using *Toeplitz* matrices
  - Also known as Im2col
- Idea:
  1. Use the **im2col** operation to transform the input image (or batch) into a matrix
  2. multiply 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

Im2col Details

Feature map: H x W x C

Conv weights: D filters, each K x K x C



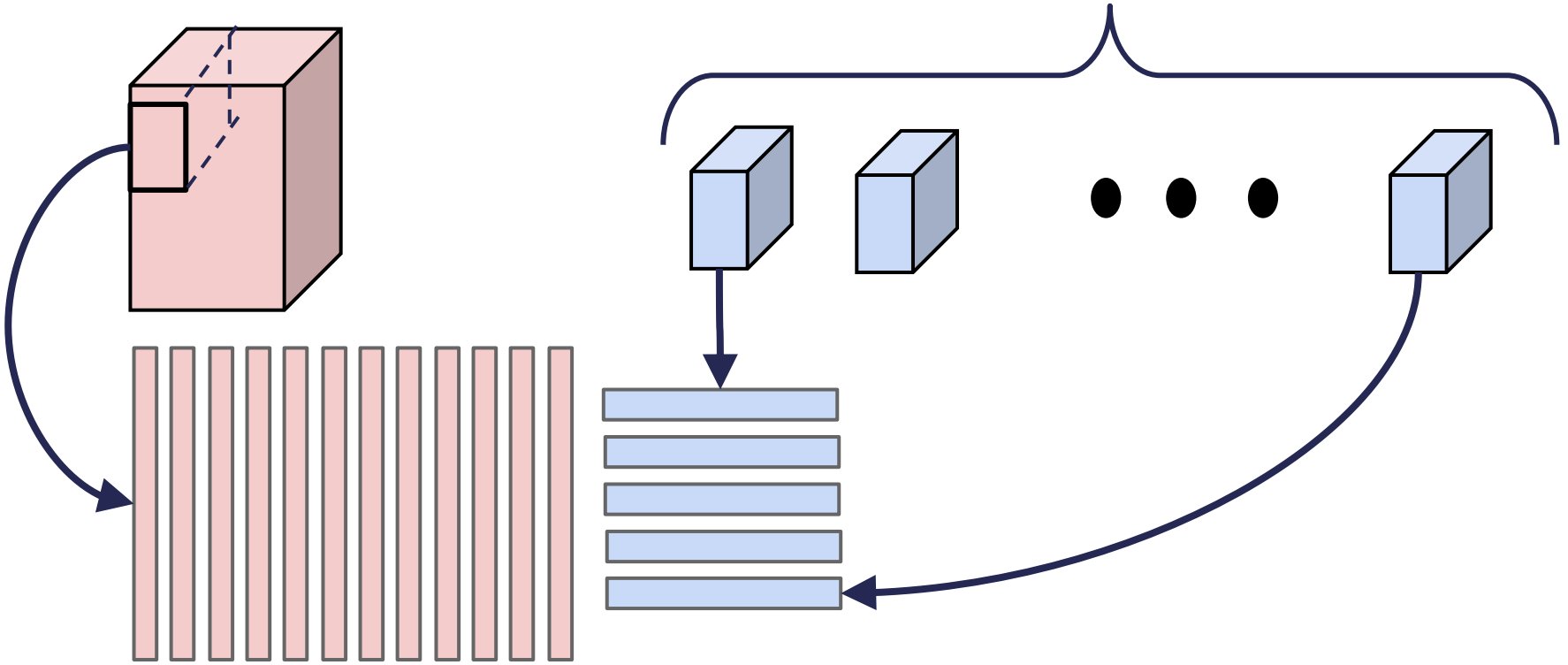


- Reshape K x K x C receptive field to column with K^2C elements
- Repeat for all columns to get $(K^2C) \times N$ matrix
- (N receptive field locations)
- Elements appearing in multiple receptive fields are duplicated
  → this uses a lot of memory

Im2col Details

Feature map: H x W x C

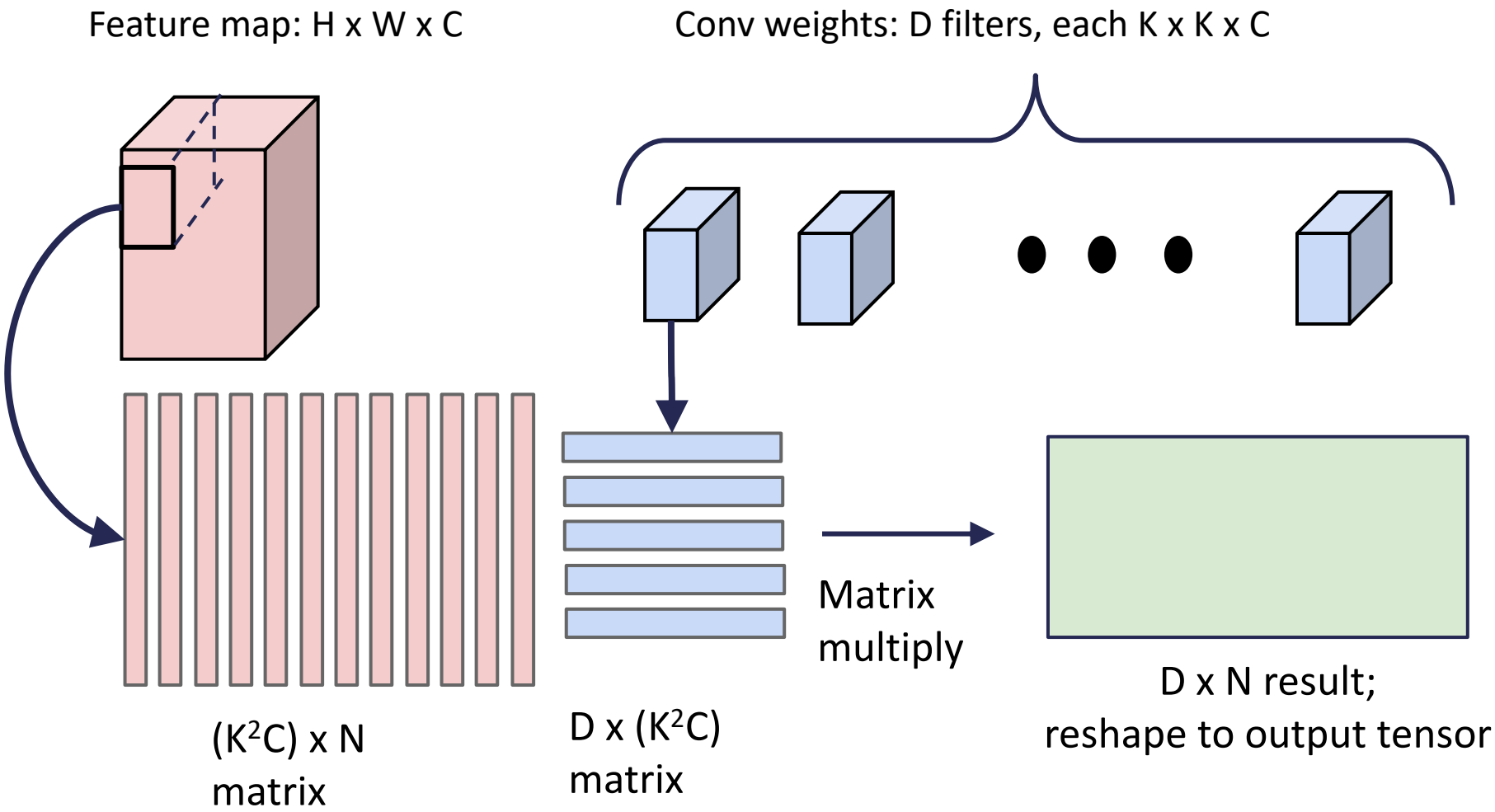
Conv weights: D filters, each K x K x C



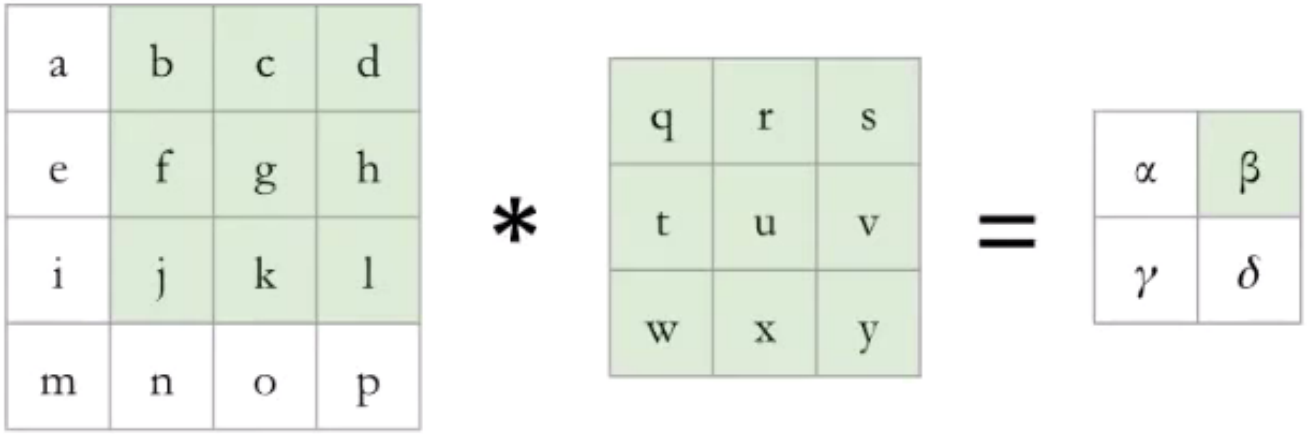
(K^2C) x N matrix

Reshape each filter to K^2C row, making D x (K^2C) matrix

Im2col Details



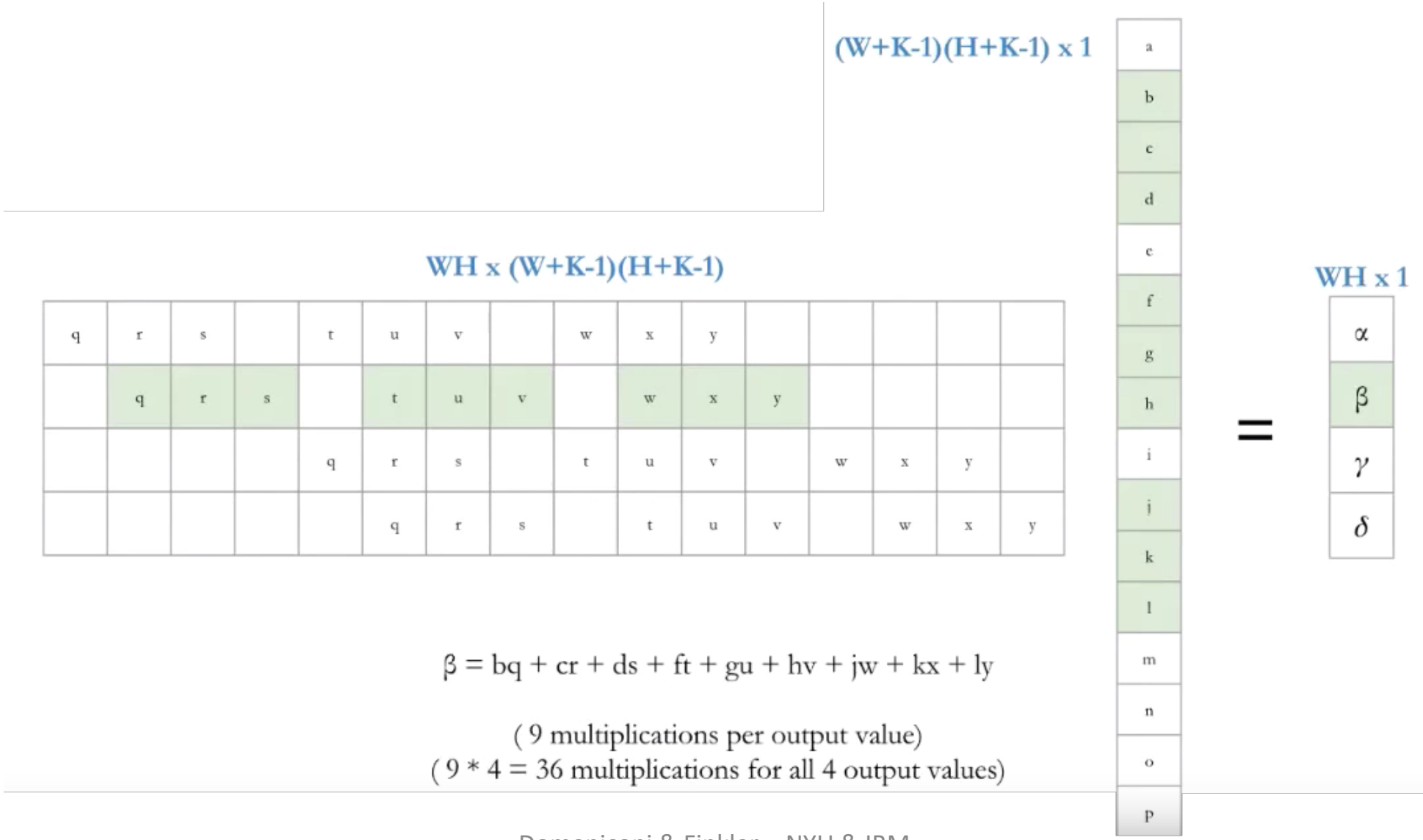
Im2col – Example with 4x4 input, 3x3 filter and 2x2 output



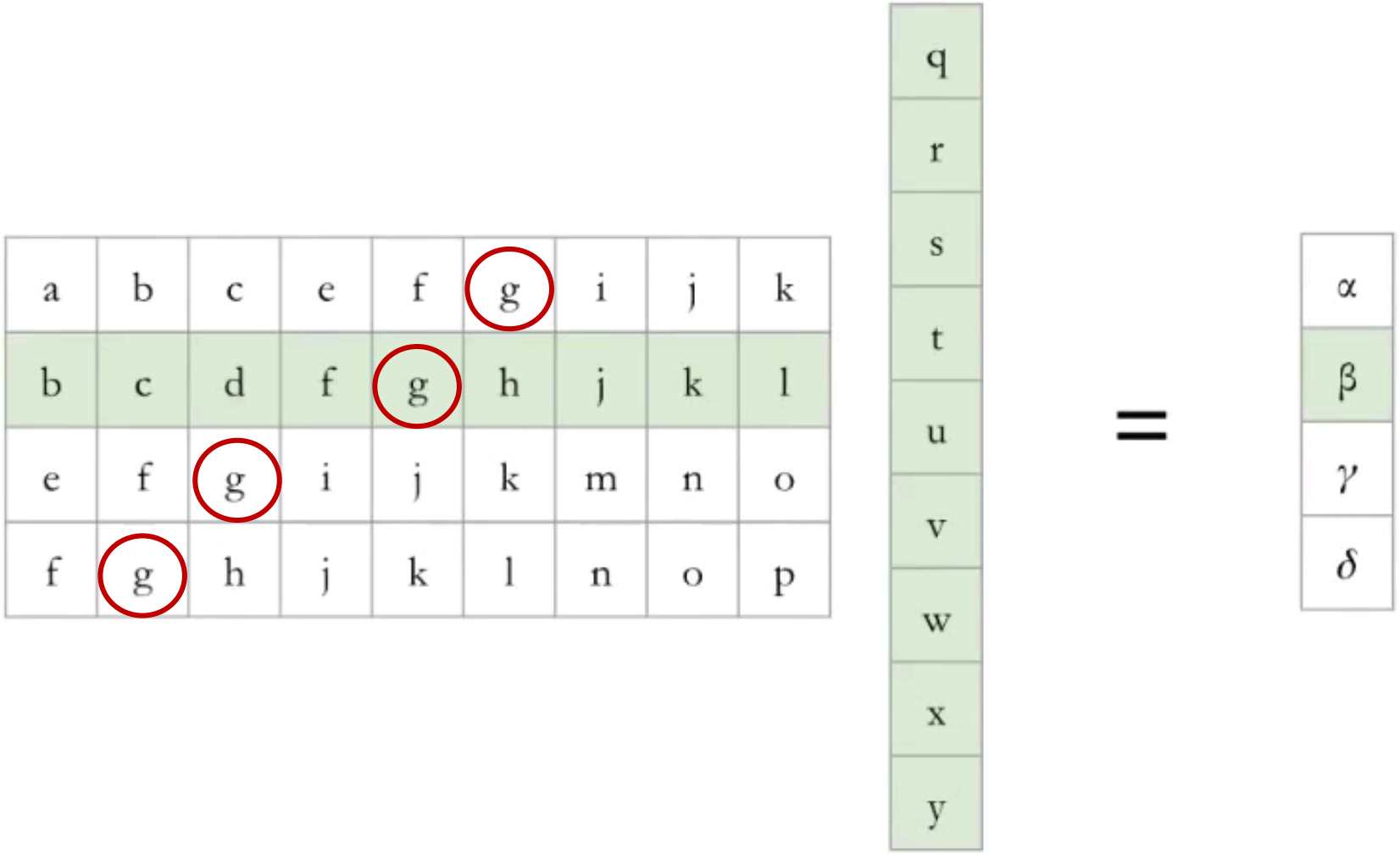
$$\beta = bq + cr + ds + ft + gu + hv + jw + kx + ly$$

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

Toeplitz matrix form



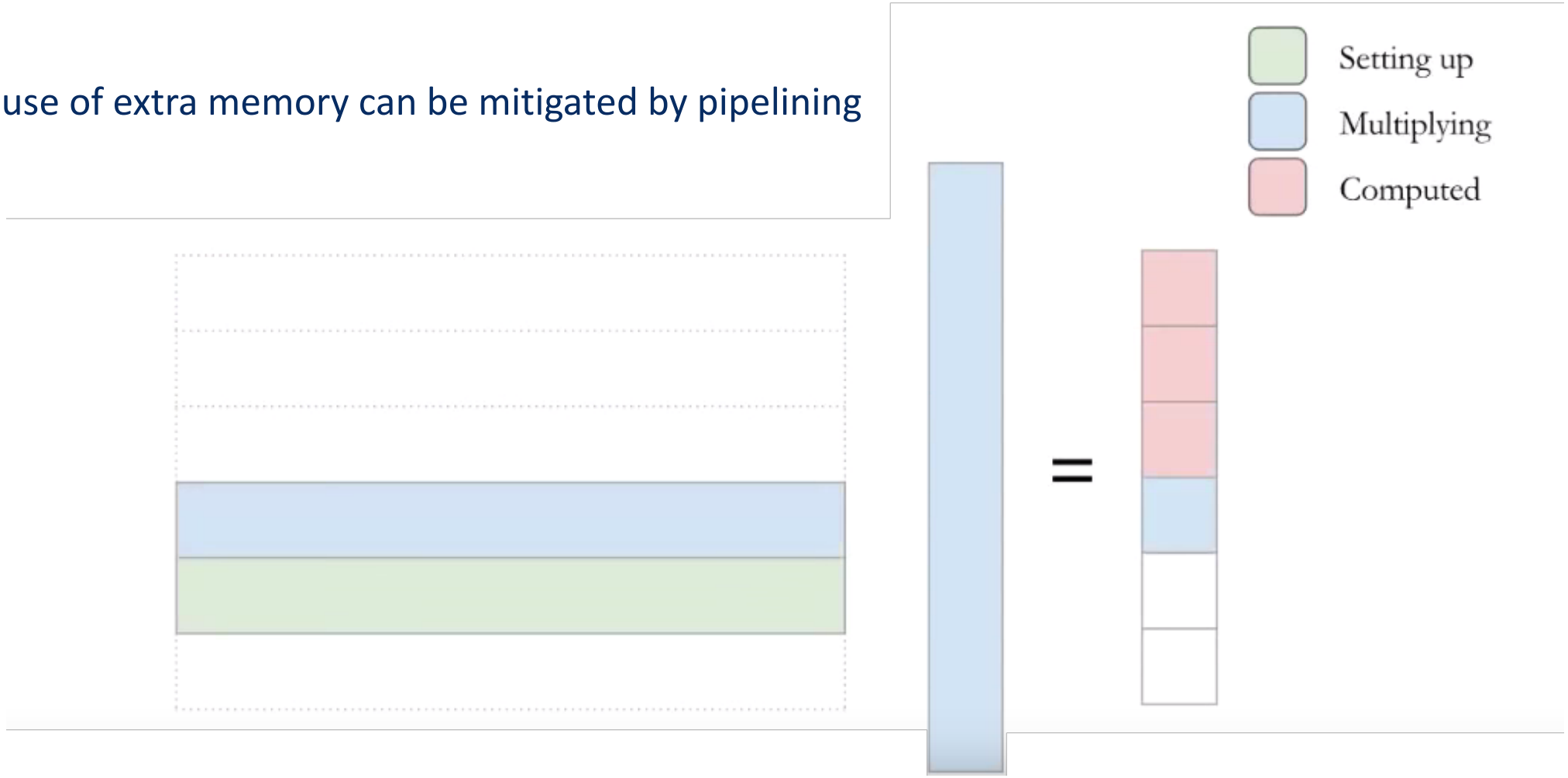
Dense matrix re-organization



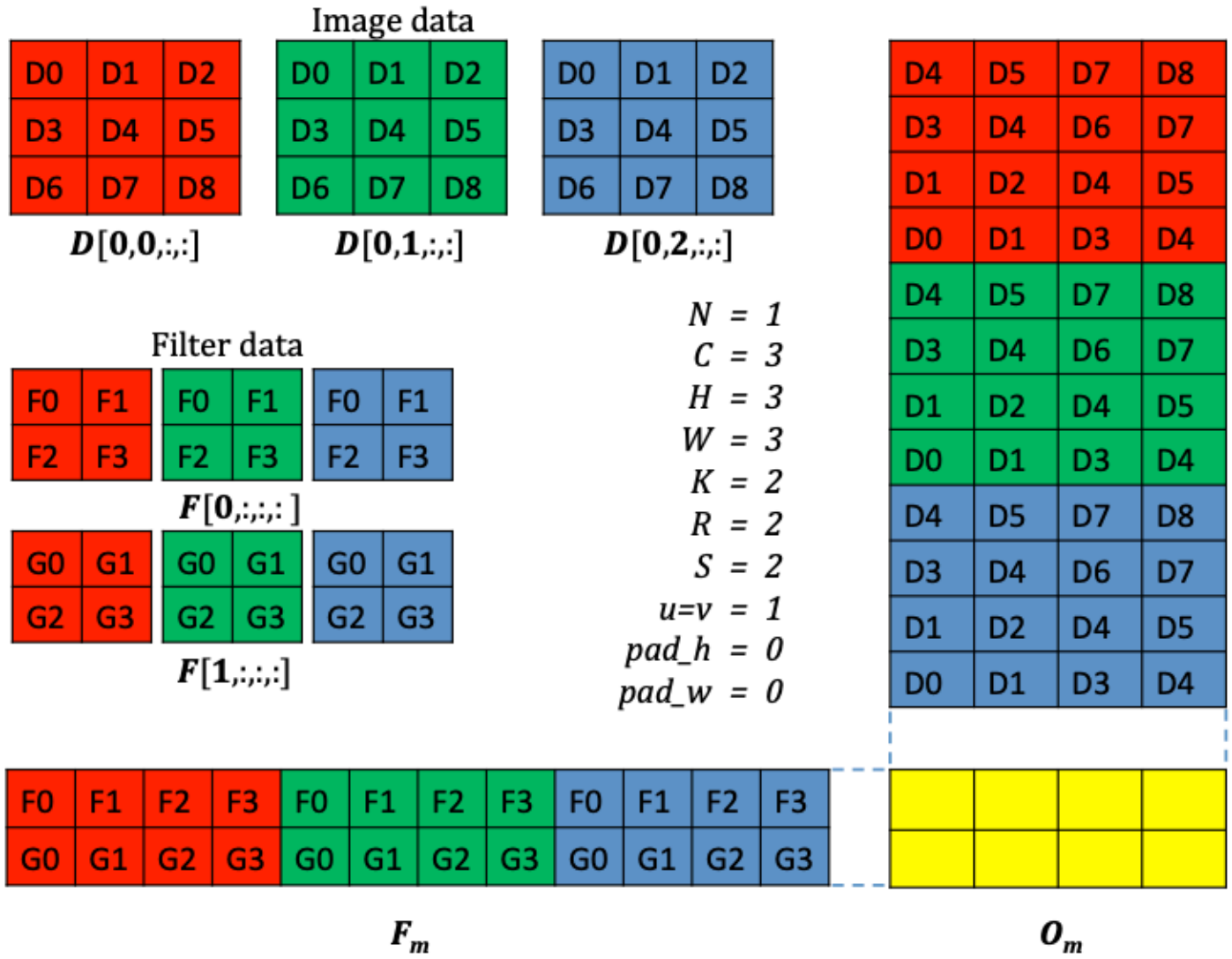
- At worse, each element is repeated K2 times
- Each dot product correspond to the (non-parallelizable) loop in line 5-7 in the direct convolution algorithm
- Can be parallelized intelligently

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:

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

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

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: $A_{i,j} \leftarrow x_{\ldots}$ ▷ im2col. Work: N/A, Depth: N/A (layout only)
4: **end for**
5: **end for**
6: ▷ Matrix Multiplication
7: $B \leftarrow F \cdot A$ ▷ Work: $C_{out} \cdot (C_{in} \cdot K_y \cdot K_x) \cdot (N \cdot H' \cdot W')$
▷ Depth: $\log_2 (C_{in} \cdot K_y \cdot K_x)$
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: $y_{i,j,k,l} \leftarrow B_{\ldots}$ ▷ col2im. Work: N/A, Depth: N/A (layout only)
14: **end for**
15: **end for**
16: **end for**
17: **end for**

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 multiplication*[1] could reduce the number of operations by up to 47%
  - Use clever arithmetic to reduce complexity to $O(N^{log2(7)}) \sim O(N^{2.81})$

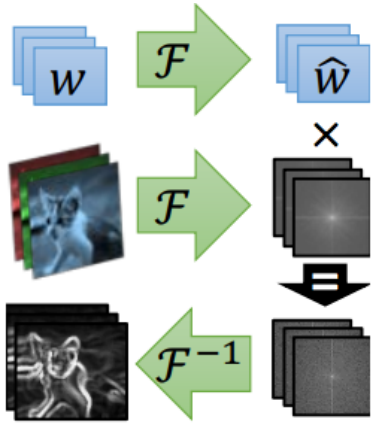
[1] V. Strassen. 1969. Gaussian Elimination is Not Optimal. Numer. Math. 13, 4 (1969), 354–356.

Computing Convolution - FFT

- Convolution Theorem: The 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 - Example

f_0

$$f = [0 \quad 0 \quad 1 \quad 2]$$

f_1

$$g = [\ 10\ 20\ 30\ ]$$

$$f * g = [\ 30\ 80\ ]$$

Dot Product Approach

$$out[0] = 0 * 10 + 0 * 20 + 1 * 20 = 30$$

$$out[1] = 0 * 10 + 1 * 20 + 2 * 20 = 80$$

- 6 Multiplications
- 4 Additions

```
>>> scipy.signal.correlate(f,g,'valid')
array([30, 80])
```

Convolution Theorem Approach

```
>>> f_0 = [0,0,1]
>>> f_1 = [0,1,2]
>>> g = [10,20,30]
>>> gg = [30,20,10]
>>> ifft(fft(f_0)*fft(gg))
array([ 20.+0.j,  10.+0.j,  30.+0.j])
>>> ifft(fft(f_1)*fft(gg))
array([ 50.+0.j,  50.+0.j,  80.+0.j])
```

$$out[0][0] = f[0][0] * g[0]$$
$$out[0][1] = f[0][1] * g[1]$$
$$out[0][2] = f[0][2] * g[2]$$

$$out[1][0] = f[1][0] * g[0]$$
$$out[1][1] = f[1][1] * g[1]$$
$$out[1][2] = f[1][2] * g[2]$$

- 6 Multiplications
- + more operations for FFT/ IFFT, … what gives?

Based on "Fast Algorithms for Convolutional Neural Networks" Andrew Lavin, Scott Gray

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: $\hat{w}_{i,j} \leftarrow \mathcal{F}(w_{i,j})$ ▷ 2D FFT. Work: $c \cdot HW \log_2 HW$, Depth: $\log_2 HW$
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: $\hat{x}_{i,j} \leftarrow \mathcal{F}(x_{i,j})$ ▷ 2D FFT. Work: $c \cdot HW \log_2 HW$, Depth: $\log_2 HW$
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** ▷ Batched MM
15: $\hat{y}_{i,j} = \sum_{m=0}^{C_{in}} \hat{x}_{i,m} \cdot \hat{w}_{j,m}$ ▷ Work: $H \cdot W \cdot N \cdot C_{in} \cdot C_{out}$
16: **end for** ▷ Depth: $\log_2 C_{in}$
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: $y_{i,j} \leftarrow \mathcal{F}^{-1}(\hat{y}_{i,j})$ ▷ 2D IFFT. Work: $c \cdot H'W' \log_2 H'W'$, Depth: $\log_2 H'W'$
23: **end for**
24: **end for**

FFT: ⇗speedup for ⇗filters

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

- From Vasilache et al, Fast Convolutional Nets With fbfft: A GPU Performance Evaluation

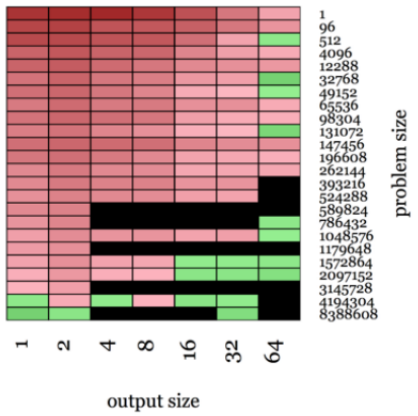


Figure 1: 3 × 3 kernel (K40m)

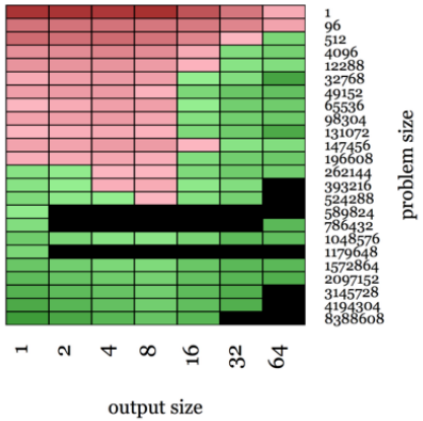


Figure 2: 5 × 5 kernel (K40m)

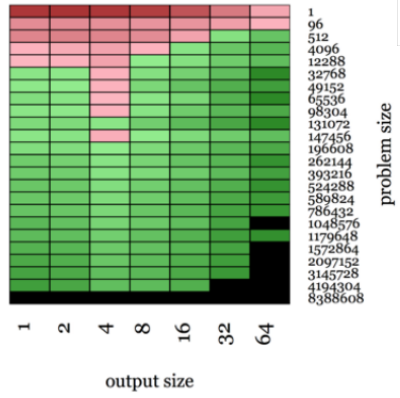


Figure 3: 7 × 7 kernel (K40m)

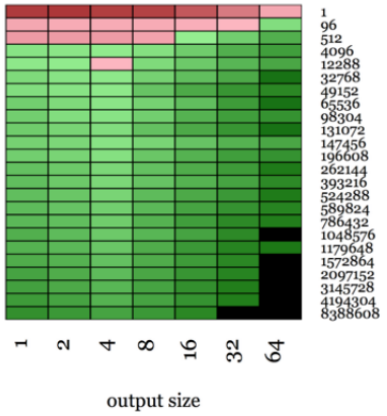


Figure 4: 9 × 9 kernel (K40m)

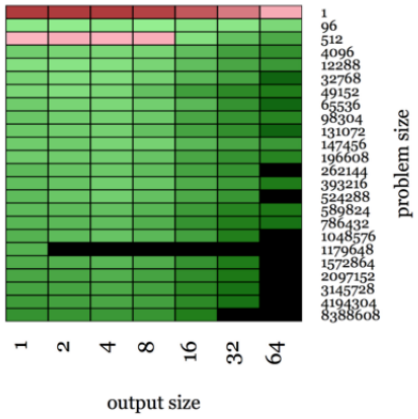


Figure 5: 11 × 11 kernel (K40m)

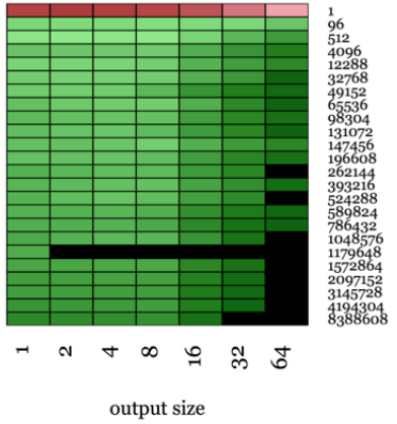
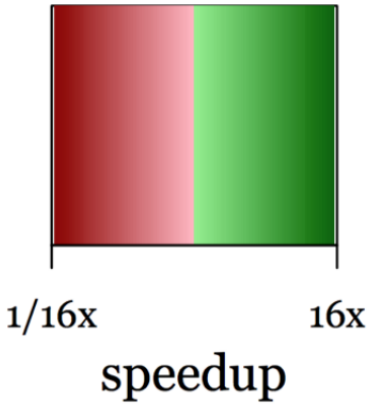


Figure 6: 13 × 13 kernel (K40m)



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:
  - $O_0=d_0*g_0+d_1*g_1+d_2*g_3$
  - $O_1=d_1*g_0+d_2*g_1+d_3*g_3$
  - need 6 MULs and 4 ADDs:

Filter size

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

outputs

(5)

where

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

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

Why 3 ADDs?

Data (d's): 4ADDs

Filter (g's): 3ADDs, 2 MULs

Outputs (m's): 4 MULs, 4ADDs

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:

$$Y = A^T \left[[GgG^T] \odot [B^T dB] \right] A \tag{8}$$

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.

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

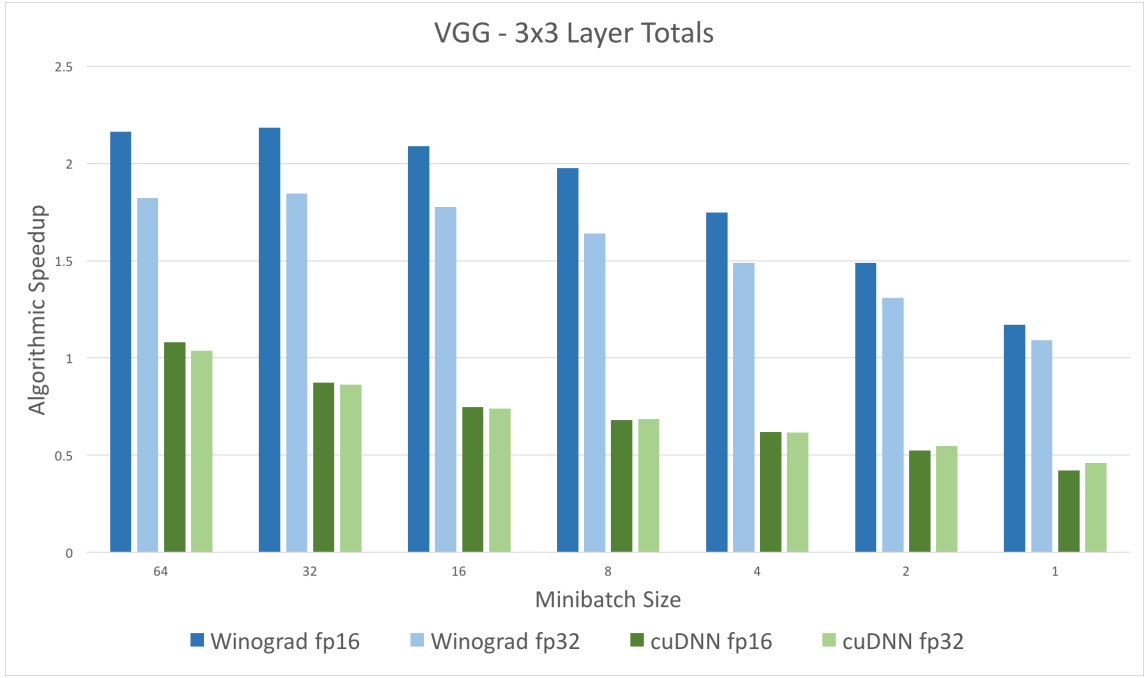
$F(2 \times 2, 3 \times 3)$

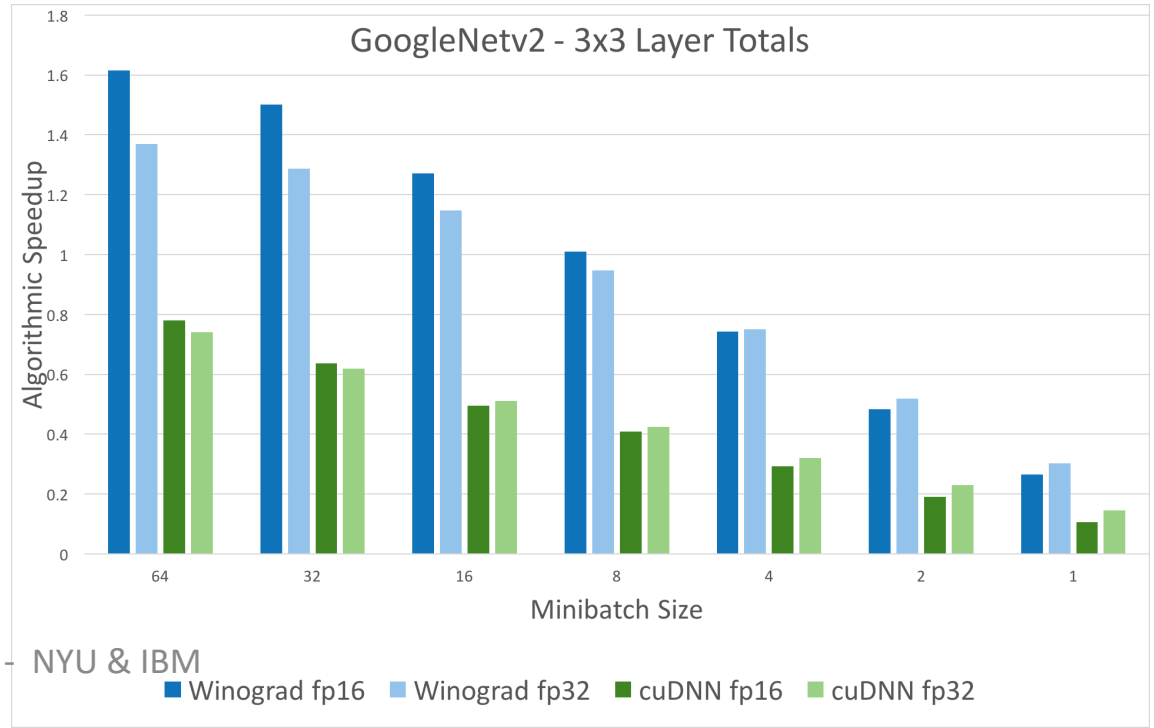
$$out = \begin{bmatrix} o_0 & o_1 \\ o_2 & o_3 \end{bmatrix} \qquad \text{filter} = \begin{bmatrix} 1 & 2 & 1 \\ 1 & 2 & 1 \\ 1 & 2 & 1 \end{bmatrix}$$

input size $= (m + r - 1) \times (m + r - 1) = 4x4$

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	$c \cdot HW \log_2(HW) \cdot (C_{out} \cdot C_{in} + N \cdot C_{in} + N \cdot C_{out}) + HWN \cdot C_{in} \cdot C_{out}$	$2\lceil \log_2 HW \rceil + \lceil \log_2 C_{in} \rceil$
Winograd ($m \times m$ tiles, $r \times r$ kernels)	$\alpha(r^2 + \alpha r + 2\alpha^2 + \alpha m + m^2) + C_{out} \cdot C_{in} \cdot P$ $(\alpha \equiv m - r + 1, \quad P \equiv N \cdot \lceil H/m \rceil \cdot \lceil W/m \rceil)$	$2\lceil \log_2 r \rceil + 4\lceil \log_2 \alpha \rceil + \lceil \log_2 C_{in} \rceil$

- 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

Lesson key points

- Data augmentation
- Transfer learning
- Use/stack small filters
- Work Depth cost model
- Fast computation of convolution
  - Im2col
  - FFT
  - Winograd

- Part of this material has been adapted from the video "How do Convolutional Neural Networks work?" under the Creative Commons License
- Part of this material has been adapted from the video "Fast Convolution Algorithms" under the Creative Commons License
- Part of this material has been inspired by John Kenny's course "Designing, Visualizing and Understanding Deep Neural Networks" @ UC Berkeley (https://bcourses.berkeley.edu/courses/1453965/pages/cs294-129-designing-visualizing-and-understanding-deep-neural-networks)