6.5930/1
Hardware Architectures for Deep Learning

Computational Transforms

February 28, 2024

Joel Emer and Vivienne Sze

Massachusetts Institute of Technology
Electrical Engineering & Computer Science

FC – Vector Operation Counts

$$O_m = I_{chw} \times F_{m,chw}$$

$$O_m = I_{chw1,chw0} \times F_{m,chw1,chw0}$$

Order: m, chw1, <u>chw0</u>

Factors: M, C*H*W/L, L

L == Lanes

```
// Level 2 loops
for m in [0, M):
  for chw1 in [0, C*H*W/L):
// Level 1 loops
    parallel_for chw0 in [0, L):
      o[m] += i[L*chw1+chw0] * f[C*H*W*m + L*chw1+chw0];
```

Vector operation on L lanes

FC – Vector Operation Counts

$$O_m = I_{chw1,chw0} \times F_{m,chw1,chw0}$$

```
// Level 2 loops
for m in [0, M):
  for chw1 in [0, C*H*W/L):
// Level 1 loops
    parallel_for chw0 in [0, L):
      o[m] += i[L*chw1+chw0] * f[C*H*W*m + L*chw1+chw0];
```

L == Lanes

How many MACs? M *(C*H*W/L) * L = M*C*H*W

How many reads of “inputs” C*H*W*M

How many reads of “weights” C*H*W*M

How many writes of “outputs” M

Measuring reads/writes in units of 32-bit integers

Compute Intensity (MACs/Read)

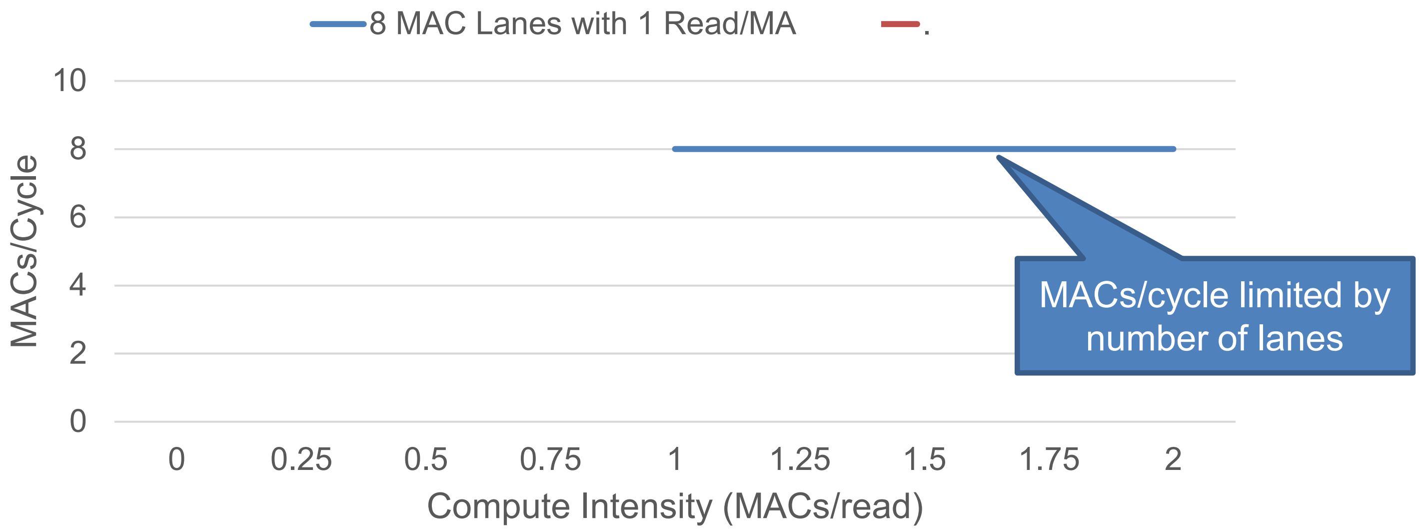
```
// Level 2 loops
for m in [0, M):
  for chw2 in [0, C*H*W/L:
// Level 1 loops
    parallel_for chw1 in [0, L):
      o[m] += i[L*chw2+chw1] * f[C*H*W*m + L*chw2+chw1];
```

L == Lanes

MACs/Read? $$\frac{C*H*W*M}{C*H*W*M + C*H*W*M} \sim \frac{1}{2}$$

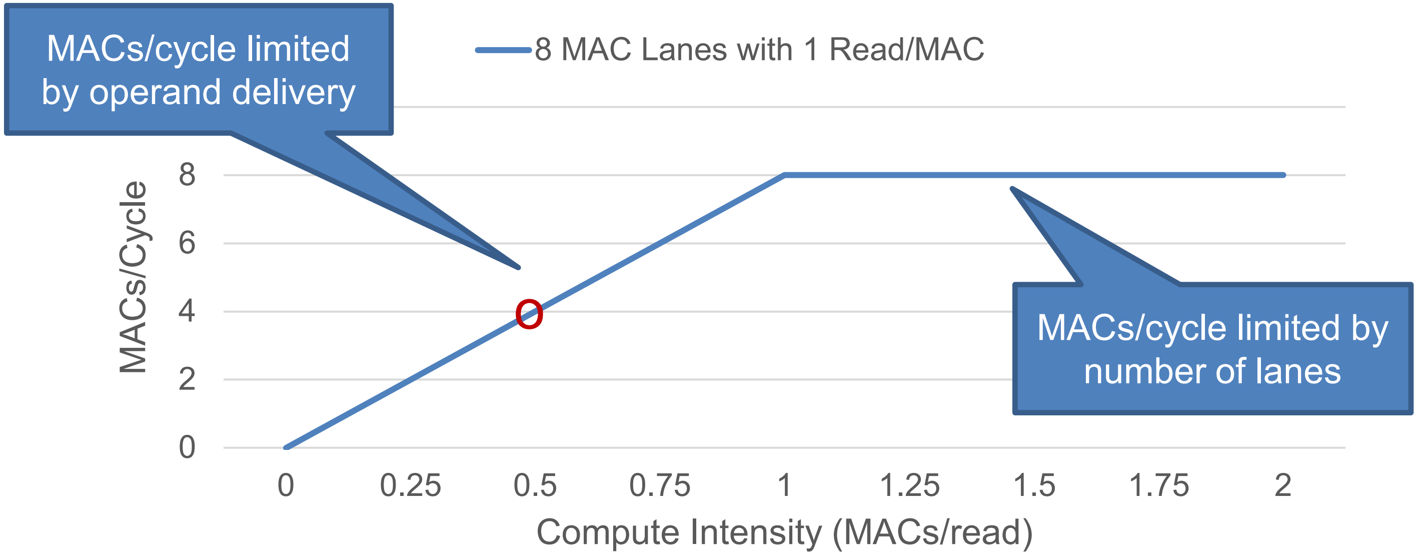
If system can support 1 Read/MAC will system run at full throttle? No

Roofline Model



Williams, Samuel, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." Communications of the ACM 52.4 (2009): 65-76.

Roofline Model



Where will the previous slide's code be? Compute intensity = 1/2

How can we change the compute intensity? Code changes, e.g., different splitting, loop inversions?

FC – Reordered + loop invariant hoisted

$$O_m = I_{chw} \times F_{m,chw}$$

$$O_{m2,m1} = I_{chw} \times F_{m2,m1,chw}$$

Order: m2, chw, m1

Factors: M/L, C*H*W, L

```
// Level 2 loops
for m2 in [0, M/L):
   for chw in [0, C*H*W):
// Level 1 loops
      parallel_for m1 in [0, L):
         o[m2*L+m1] += i[chw] * f[CHW*(m2*L+m1) + chw]
```

L == Lanes

Strided?

Strided?

FC – Reordered + loop invariant hoisted

L == Lanes

```
// Level 2 loops
for m2 in [0, M/L):
   for chw in [0, C*H*W):
// Level 1 loops
     parallel_for m1 in [0, L):
        o[m2*L+m1] += i[chw] * f[CHW*(m2*L+m1) + chw]
```

L == Lanes

```
// Level 2 loops
for m2 in [0, M/L):
  for chw in [0, C*H*W):
    i_chw = i[chw]
// Level 1 loops
      parallel_for m1 in [0, L):
         o[m2*L+m1] += i_chw * f[CHW*(m2*L+m1) + chw]
```

Loop invariant hoisted

Use loop invariant in register

FC – Operation Counts

$$O_{m2,m1} = I_{chw} \times F_{m2,m1,chw}$$

```
// Level 2 loops
for m2 in [0, M/L):
  for chw in [0, C*H*W):
    i_chw = i[chw]
// Level 1 loops
      parallel_for m1 in [0, L):
        o[m1*L+m0] += i_chw * f[CHW*(m1*L+m0) + chw]
```

L == Lanes

How many MACs?	C*H*W*M
How many reads of “inputs”	C*H*W*M/L
How many reads of “weights”	C*H*W*M
How many writes of “outputs”	M

Measuring reads/writes in units of 32-bit integers

Compute Intensity

```
// Level 2 loops
for m1 in [0, M/L):
  for chw in [0, C*H*W):
    i_chw = i[chw]
// Level 1 loops
      parallel_for m1 in [0, L):
        o[m1*L+m0] += i_chw * f[CHW*(m1*L+m0) + chw]
```

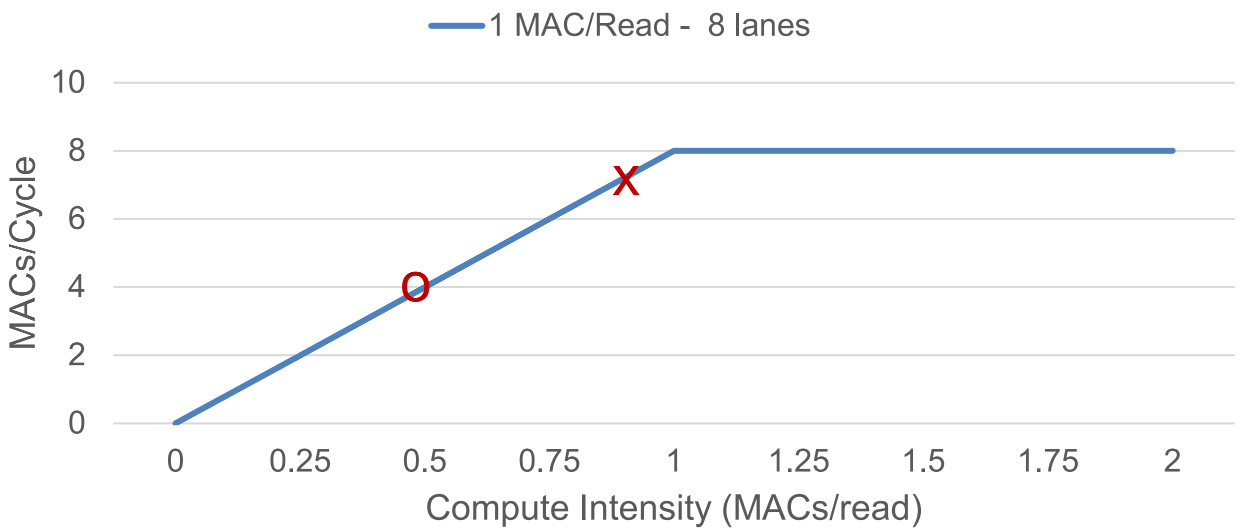
MACs/Read?

$$\frac{C*H*W*M}{C*H*W*M + C*H*W*M/L} \sim \frac{L}{1+L}$$

If system can support 1 Read/MAC will system run at full throttle?

No

Roofline Model



Where will the previous slide's code be? Compute intensity = L/(1+L) = 8/9

Why might points be below the line? Other overheads (e.g. instructions, stalls)

Is being on the flat part always best? Not necessarily…

Computation Transformations

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

Gauss's Multiplication Algorithm

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

4 multiplications + 3 additions

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

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

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

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

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

3 multiplications + 5 additions

Strassen

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}_{C}$$

8 multiplications + 4 additions

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

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

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

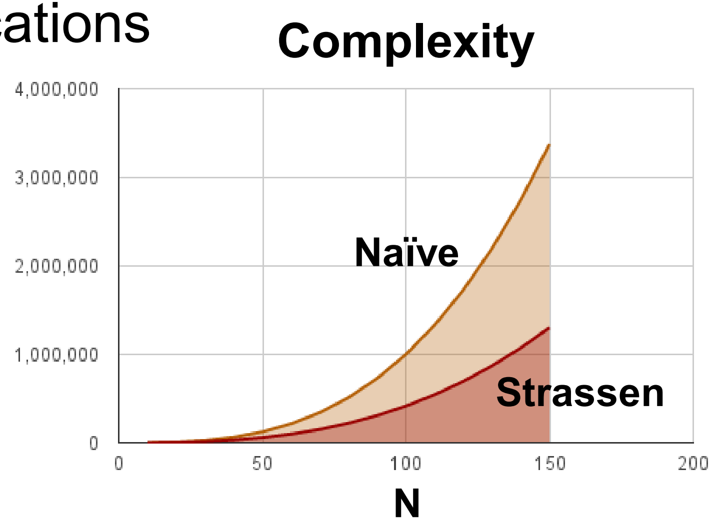
7 multiplications + 18 additions

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

[Cong et al., ICANN, 2014]

Strassen

- Reduce the complexity of matrix multiplication from **Θ(N³)** to **Θ(N^2.807)** by reducing multiplications



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

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

Python to C++ Chart

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak
1	Python	25,552.48	0.005	1	—	0.00%
2	Java	2,372.68	0.058	11	10.8	0.01%
3	C	542.67	0.253	47	4.4	0.03%
4	Parallel loops	69.80	1.969	366	7.8	0.24%
5	Parallel divide-and-conquer	3.80	36.180	6,727	18.4	4.33%
6	+ vectorization	1.10	124.914	23,224	3.5	14.96%
7	+ AVX intrinsics	0.41	337.812	62,806	2.7	40.45%
8	Strassen	0.38	361.177	67,150	1.1	43.24%

[**Leiserson**, There's plenty of room at the top, *Science*, 2020]

Tensor Computations

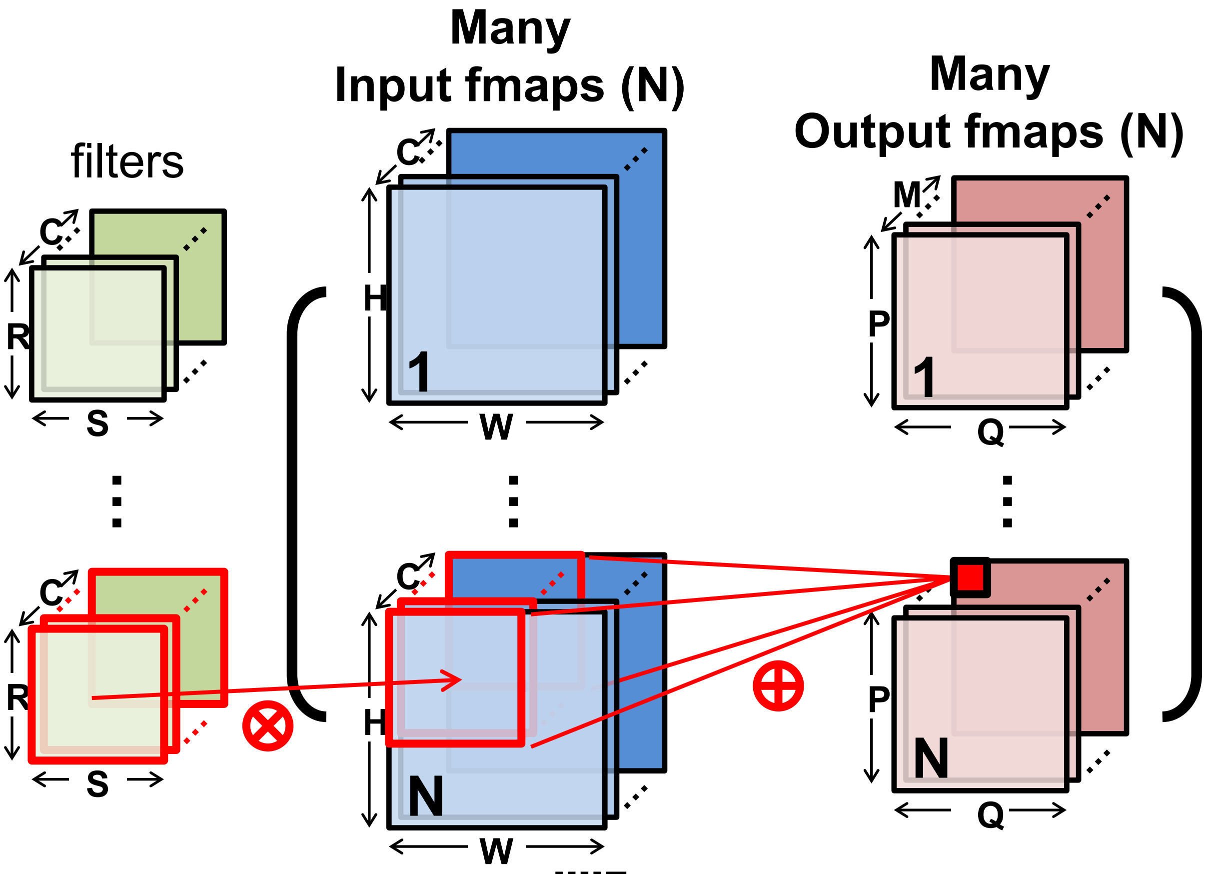
Matrix Multiply

$$O_{n,m} = I_{n,chw} \times F_{m,chw}$$

CONV Layer

$$O_{n,m,p,q} = I_{n,c,Up+r,Uq+s} \cdot F_{m,c,r,s}$$

Convolution (CONV) Layer



Batch Size (N)

CONV Layer Implementation

Naïve 7-layer for-loop implementation:

```
for n in [0..N):
    for m in [0..M):
        for q in [0..Q):
            for p in [0..P):
```

for each output fmap value

```
O[n][m][p][q] = B[m];
for c in [0..C):
    for r in [0..R):
        for s in [0..S):
            O[n][m][p][q] += I[n][c][Up+r][Uq+s]
                           × F[m][c][r][s];

O[n][m][p][q] = Activation(O[n][m][p][q]);
```

convolve a window and apply activation

Winograd 1D – F(2,3)

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

inputs / filter / outputs

$$F(2,3) = \begin{bmatrix} i_0 & i_1 & i_2 \\ i_1 & i_2 & i_3 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} o_0 \\ o_1 \end{bmatrix}$$

6 multiplications + 4 additions

[Lavin et al., CVPR 2016]

Winograd 1D – F(2,3)

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

$$\text{F(2,3)} = \underbrace{\begin{bmatrix} i_0 & i_1 & i_2 \\ i_1 & i_2 & i_3 \end{bmatrix}}_{\text{inputs}} \underbrace{\begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix}}_{\text{filter}} = \underbrace{\begin{bmatrix} k_1 + k_2 + k_3 \\ k_2 - k_3 - k_4 \end{bmatrix}}_{\text{outputs}}$$

$$k_1 = (i_0 - i_2) f_0$$

$$k_2 = (i_1 + i_2) \frac{f_0 + f_1 + f_2}{2}$$

$$k_3 = (i_2 - i_1) \frac{f_0 - f_1 + f_2}{2}$$

$$k_4 = (i_1 - i_3) f_2$$

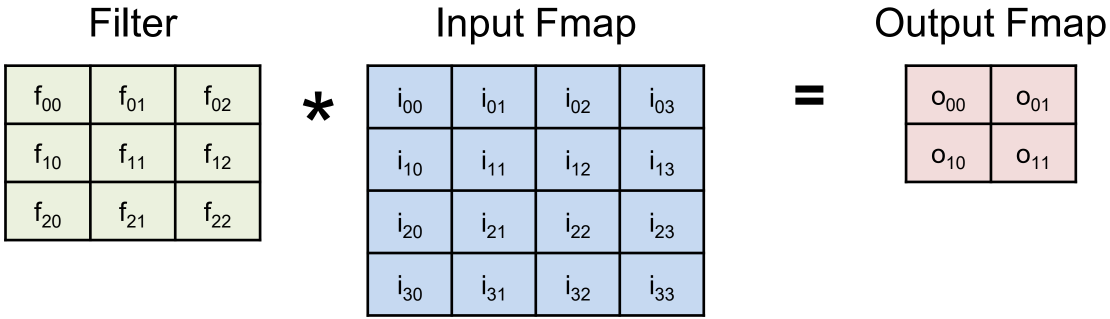
4 multiplications + 12 additions + 2 shifts

4 multiplications + 8 additions (for constant weights)

[Lavin et al., CVPR 2016]

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

- 1D Winograd is nested to make 2D Winograd

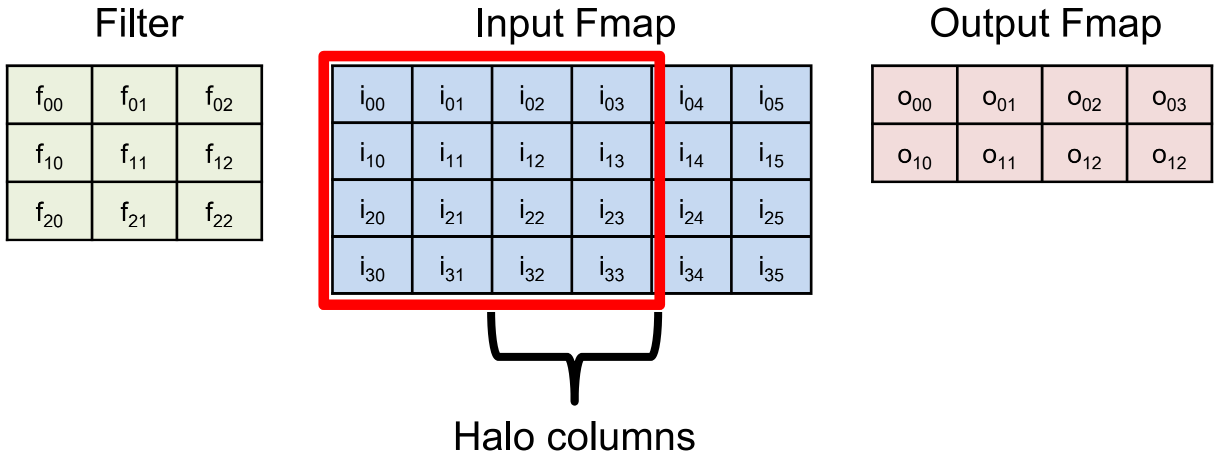


Original: 36 multiplications

Winograd: 16 multiplications → 2.25 times reduction

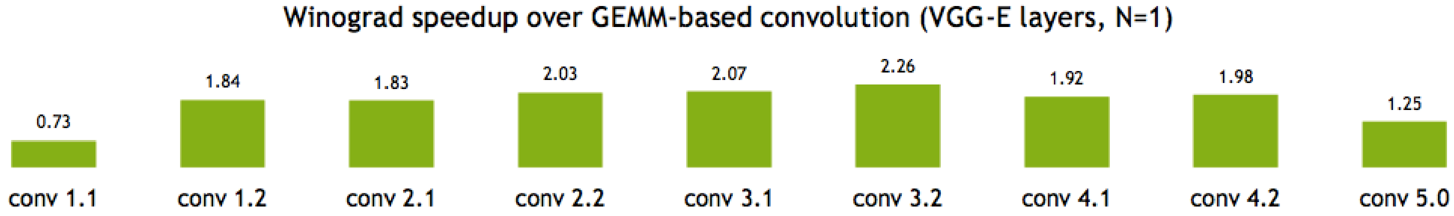
Winograd Halos

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



Winograd Performance Varies

Optimal convolution algorithm depends on convolution layer dimensions



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

Using texture memory for convolutions: 13% inference speedup

(GoogLeNet, batch size 1)

Source: Nvidia

Winograd Summary

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

Winograd as a Transform

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

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

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

filter $f = \begin{bmatrix} f_0 & f_1 & f_2 \end{bmatrix}^T$

input $i = \begin{bmatrix} i_0 & i_1 & i_2 & i_3 \end{bmatrix}^T$

Transform inputs

$$Y = A^T \left[\left[G f G^T \right] \odot \left[B^T i B \right] \right] A$$

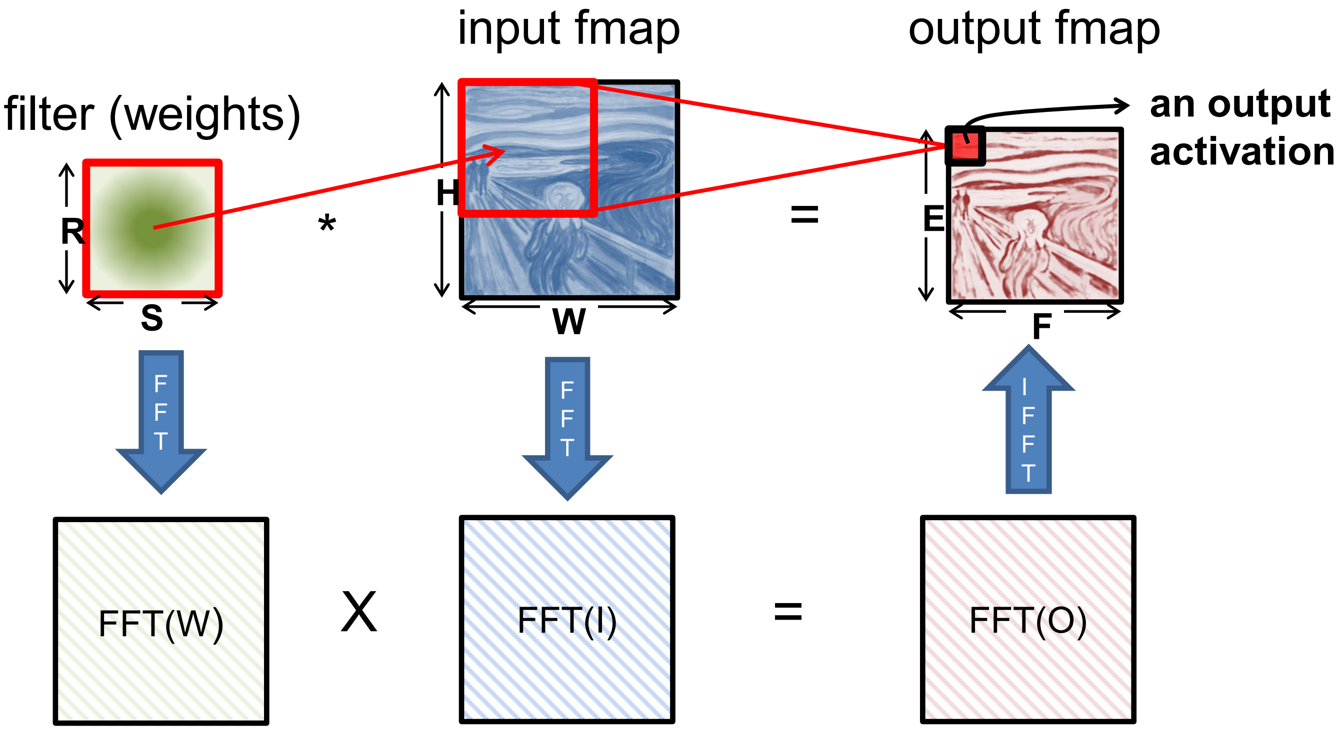
Element-wise Multiplication

Transform output

Note: GfG^T can be precomputed

[**Lavin**, *ArXiv* 2015]

Fast Fourier Transform (FFT) Flow



FFT Overview

- Convert filter and input to frequency domain to make convolution a simple multiply then convert back to space domain.
- Convert direct convolution $O(N_o^2 N_f^2)$ computation to $O(N_o^2 \log_2 N_o)$
- Note that computational benefit of FFT decreases with decreasing size of filter

[**Mathieu**, *ArXiv* 2013], [**Vasilache**, *ArXiv* 2014]

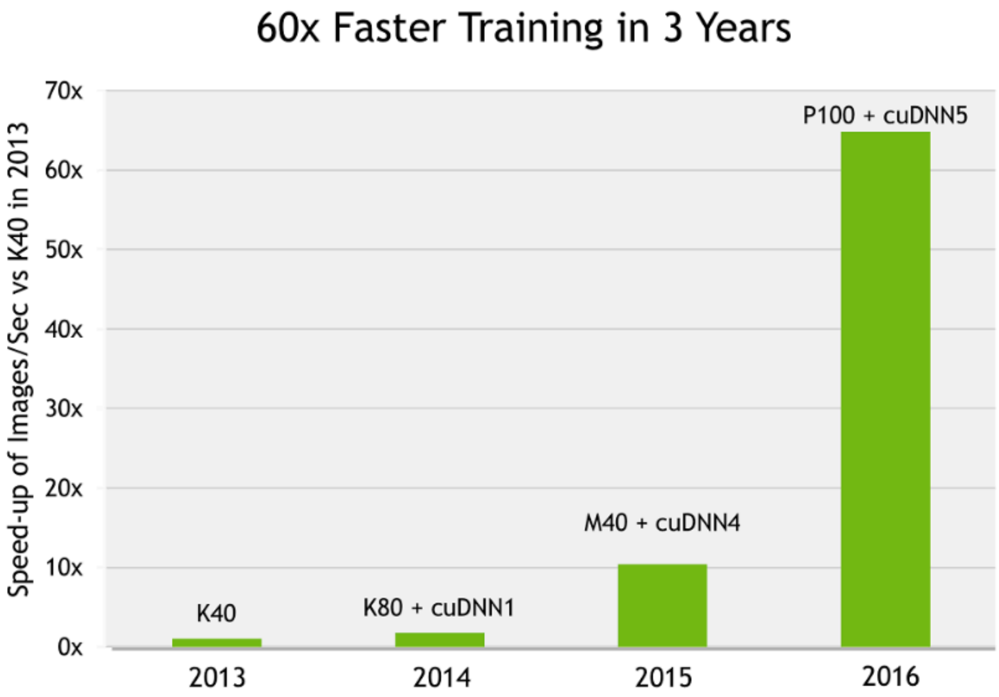
FFT Costs

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

Optimization opportunities

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

cuDNN: Speed up with Transformations



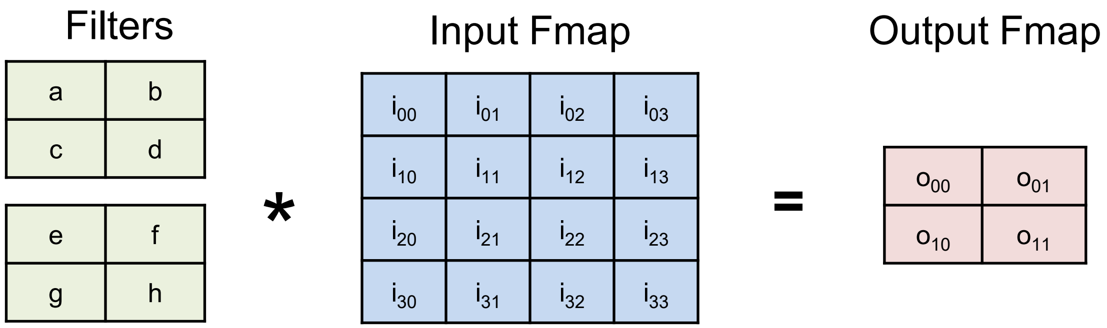
AlexNet training throughput on:

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

Source: Nvidia

UCNN – Convolution (Simplified)

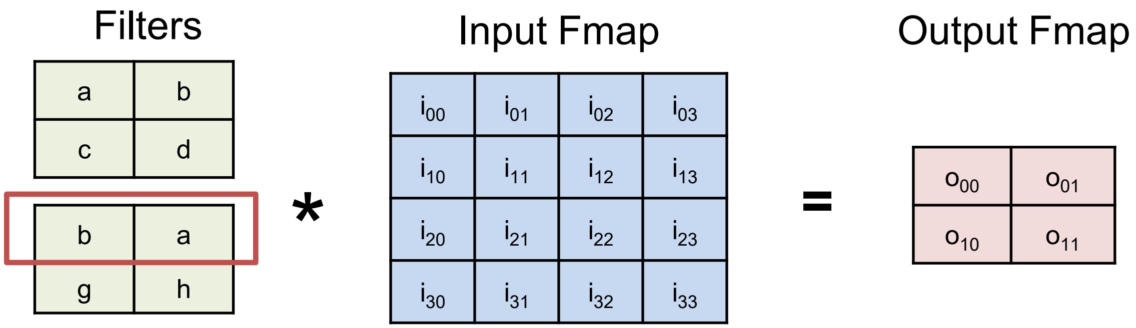


$$o_{00} = a\, i_{00} + b\, i_{01} + \mathrm{c}\, i_{10} + d\, i_{11} + e\, i_{00} + f\, i_{01} + g\, i_{10} + \mathrm{h}\, i_{11}$$

7 additions
8 multiplications

[**Hegde**, *ISCA* 2018]

UCNN – Convolution (Simplified)



$$o_{00} = a\, i_{00} + b\, i_{01} + \mathrm{c}\, i_{10} + d\, i_{11} + e\, i_{00} + f\, i_{01} + g\, i_{10} + \mathrm{h}\, i_{11}$$

$$o_{00} = a\, i_{00} + b\, i_{01} + \mathrm{c}\, i_{10} + d\, i_{11} + \mathrm{b}\, i_{00} + a\, i_{01} + g\, i_{10} + \mathrm{h}\, i_{11}$$

$$o_{00} = (a+b) i_{00} + (a+b)\, i_{01} + \mathrm{c}\, i_{10} + d\, i_{11} + g\, i_{10} + \mathrm{h}\, i_{11}$$

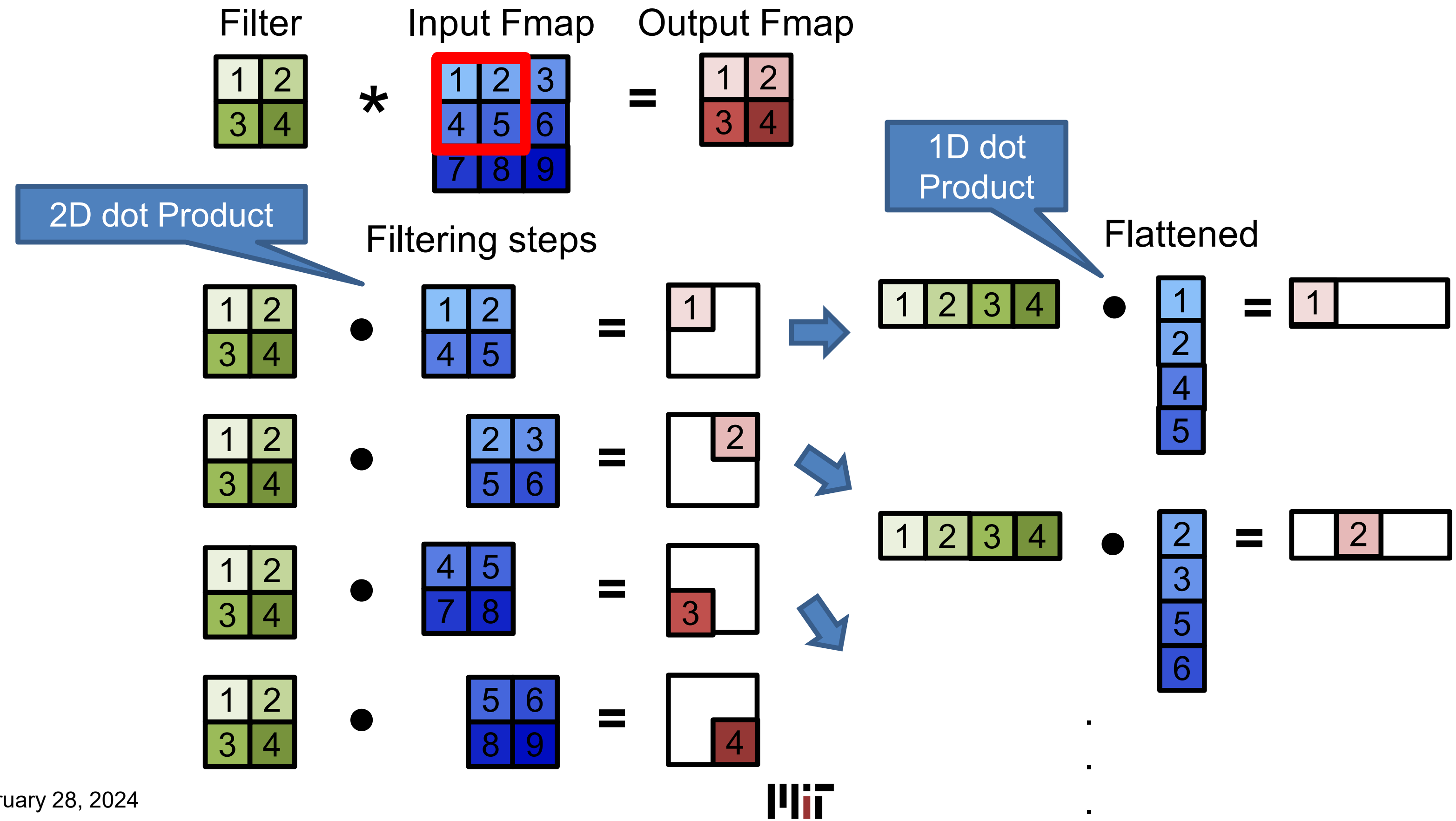
7 additions
8 multiplications

→

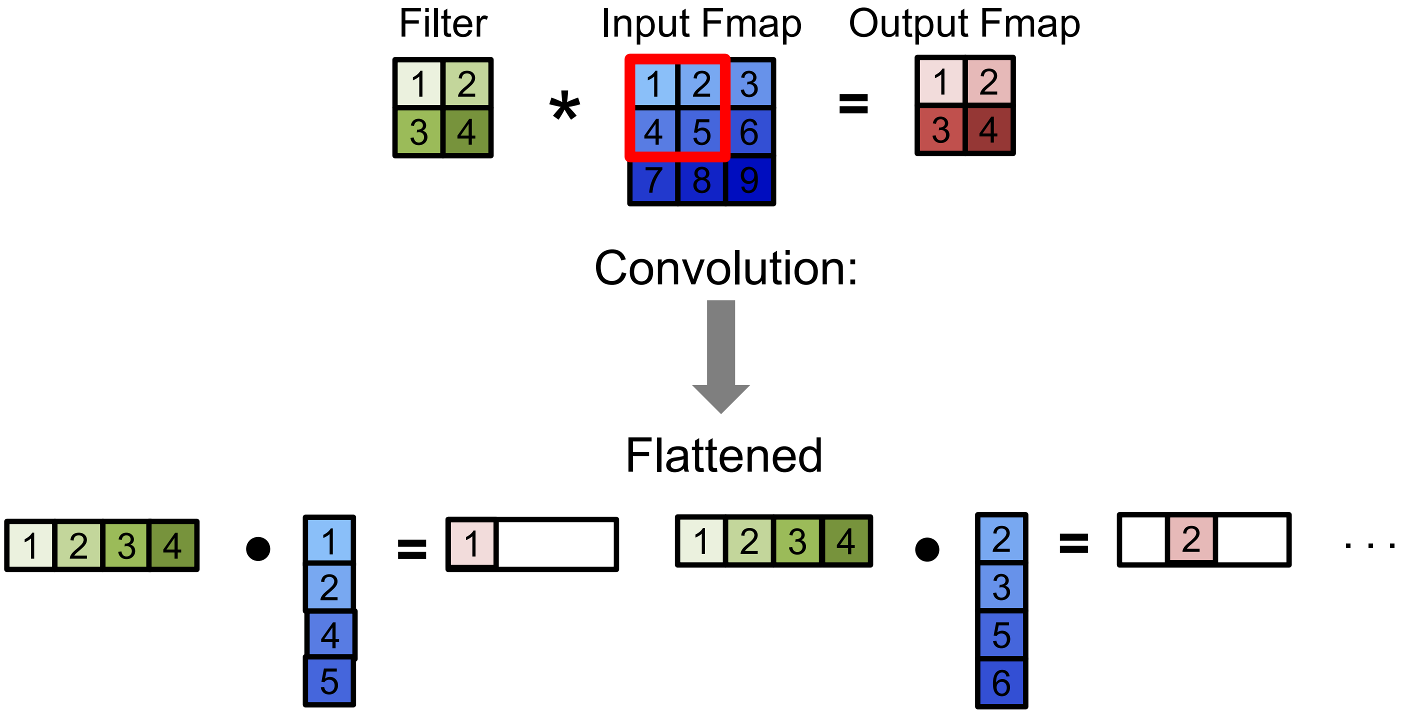
6 additions
6 multiplications

[**Hegde**, *ISCA* 2018]

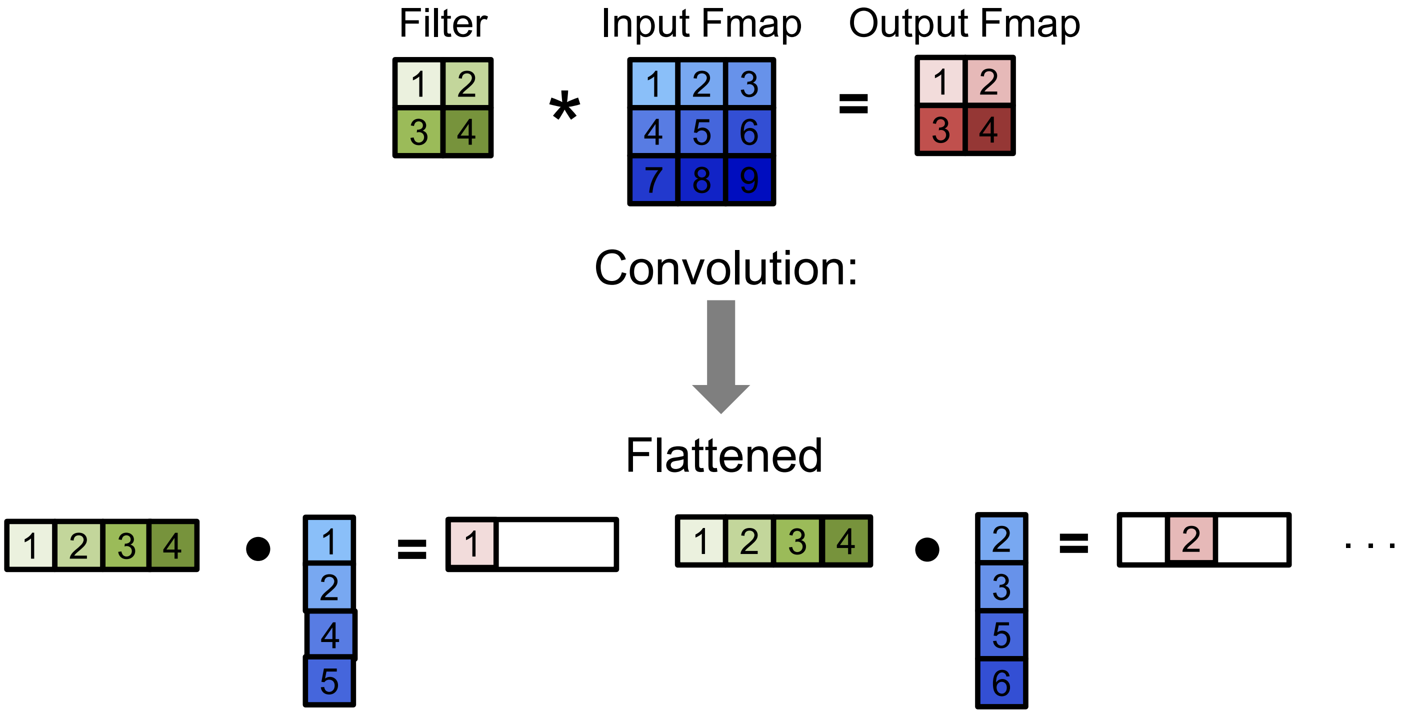
Convolution (CONV) Layer



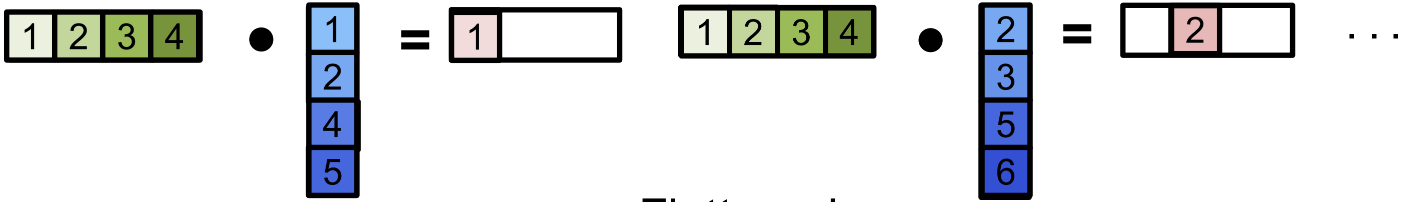
Convolution (CONV) Layer



Convolution (CONV) Layer

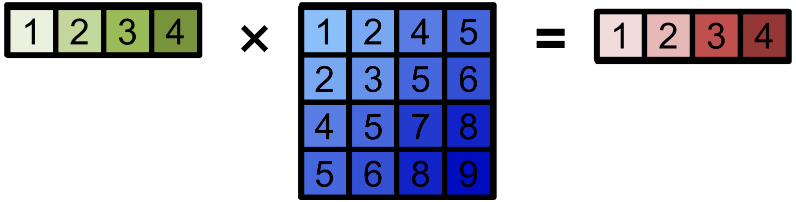


Convolution (CONV) Layer

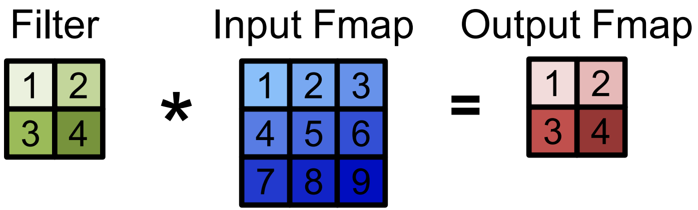


Flattened

Matrix Multiply (by Toeplitz Matrix)

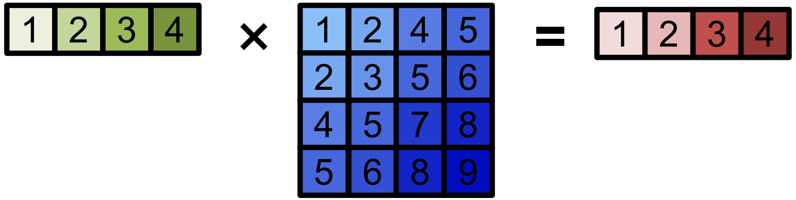


Convolution (CONV) Layer



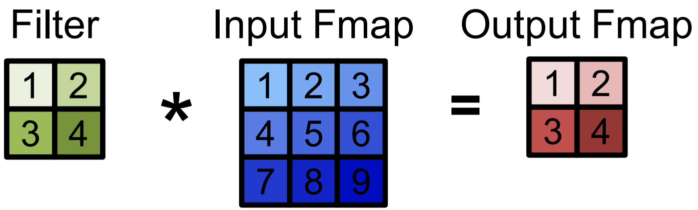
Convolution:

Matrix Multiply (by Toeplitz Matrix)



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

Convolution (CONV) Layer



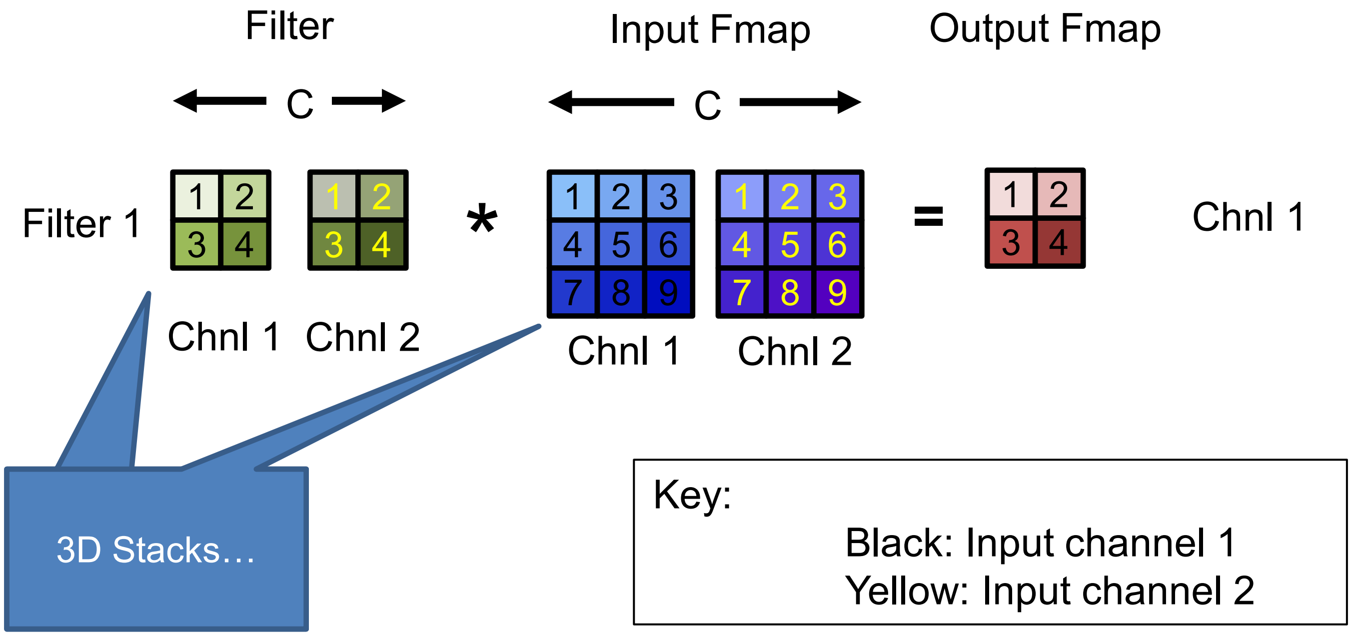
Convolution:

Matrix Multiply (by Toeplitz Matrix)

Data is repeated

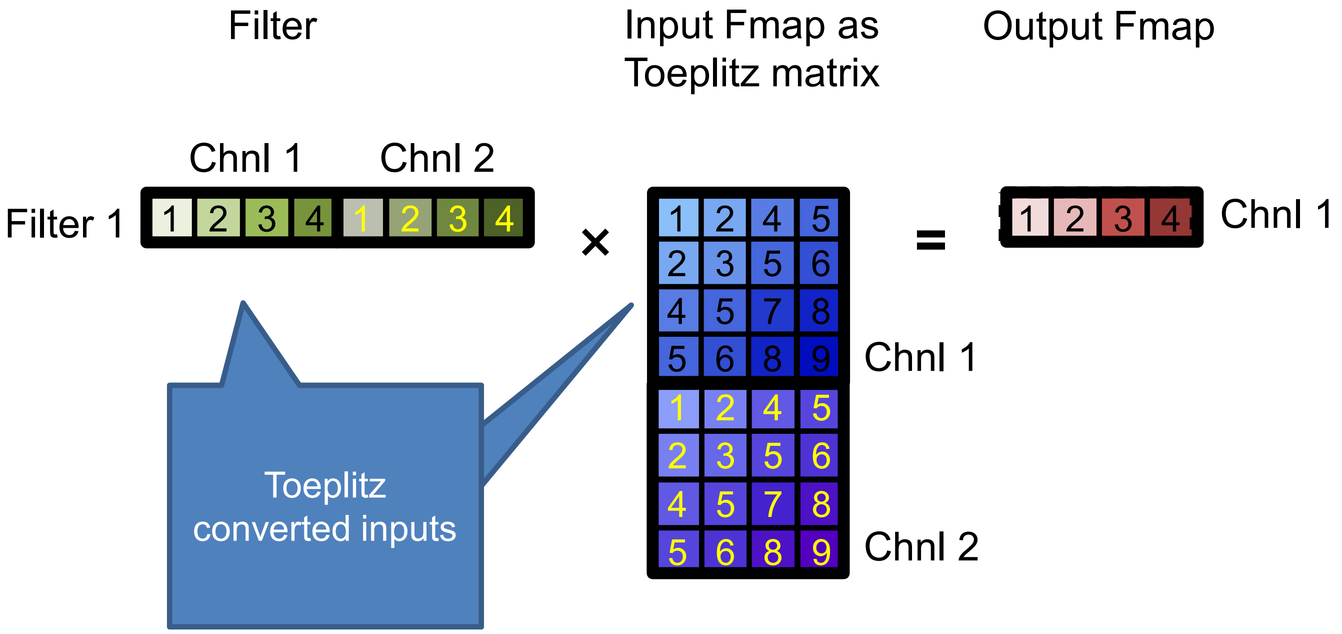
Convolution (CONV) Layer

- Multiple Input Channels



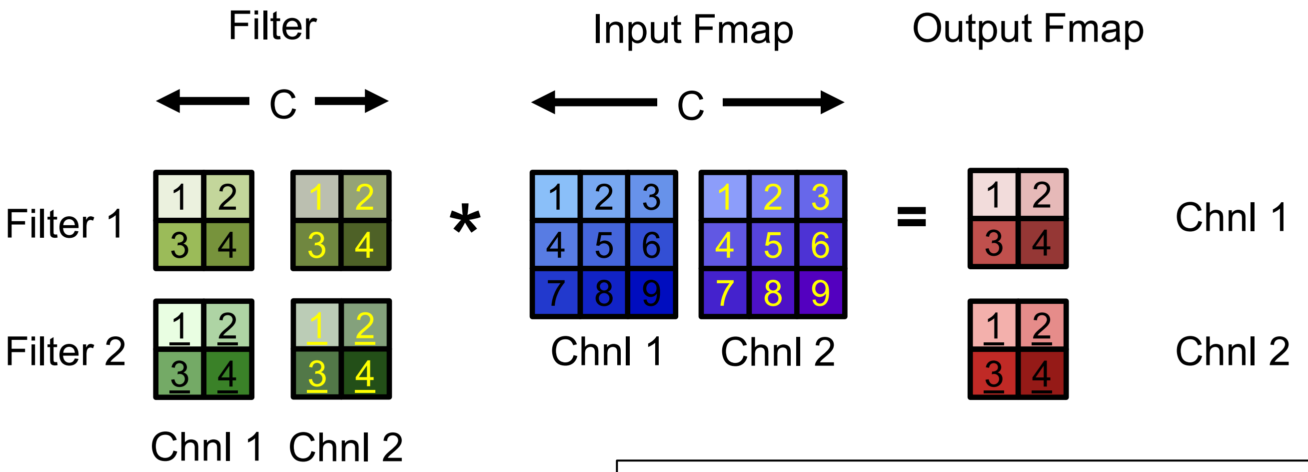
Convolution (CONV) Layer

- Multiple Input Channels



Convolution (CONV) Layer

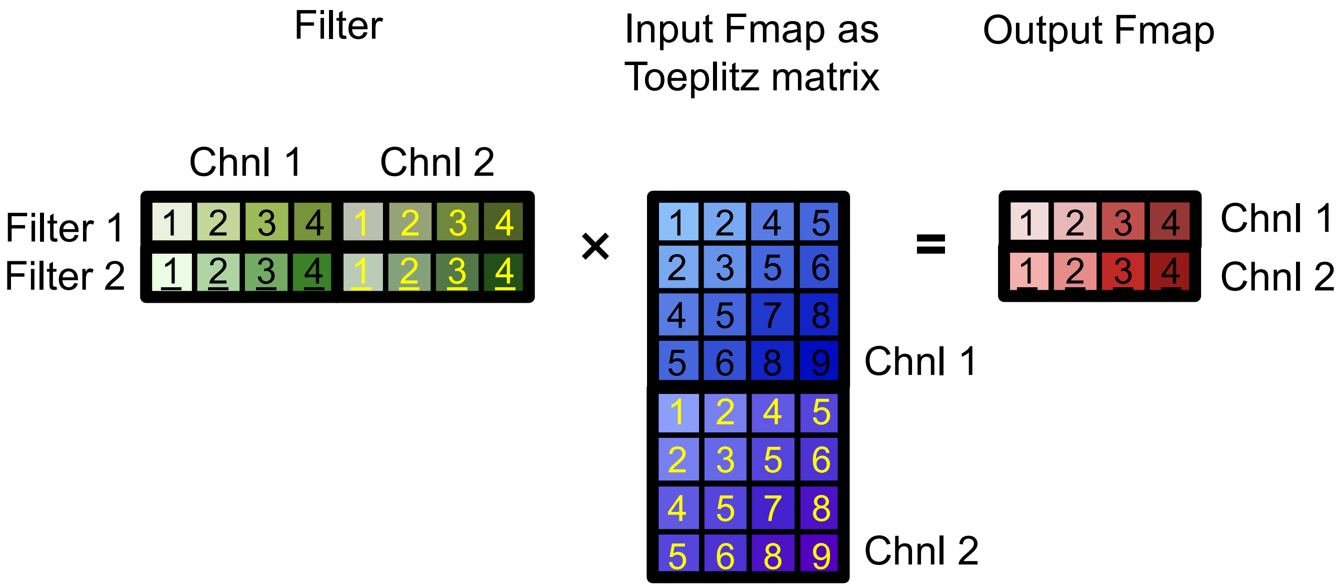
- Multiple Input Channels and Output Channels



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

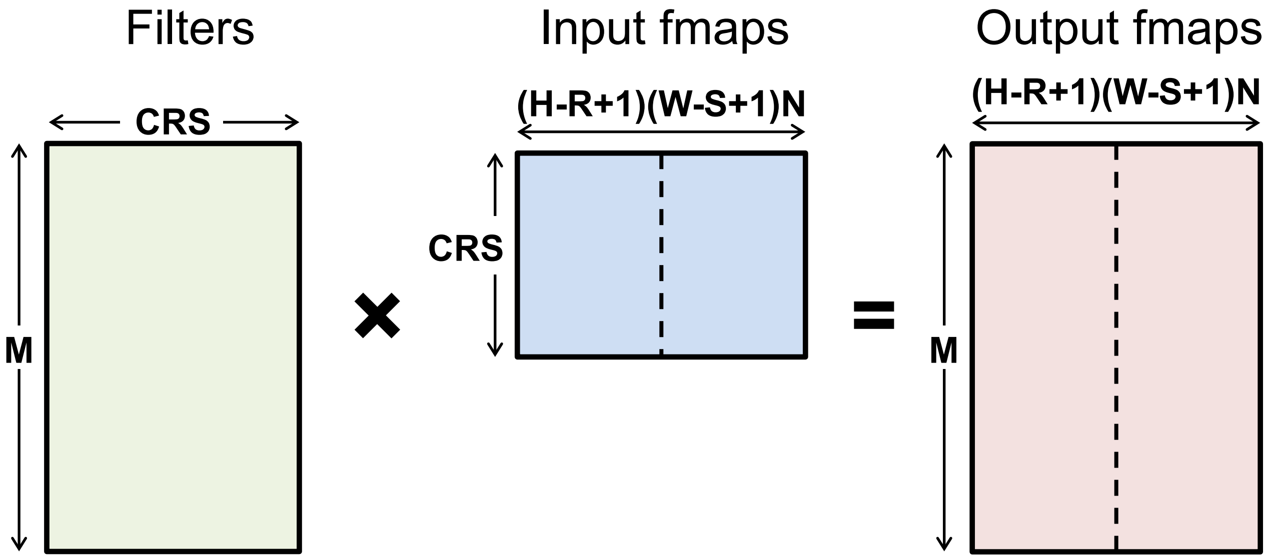
Convolution (CONV) Layer

- Multiple Input Channels and Output Channels



Convolution (CONV) Layer

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



N=2 in example

1-D Toeplitz Convolution Einsum

$$O_{n,m,p,q} = I_{n,c,Up+r,Uq+s} \times F_{m,c,r,s}$$

Simplify to 1-D with N=1, C=1, M=1, U=1

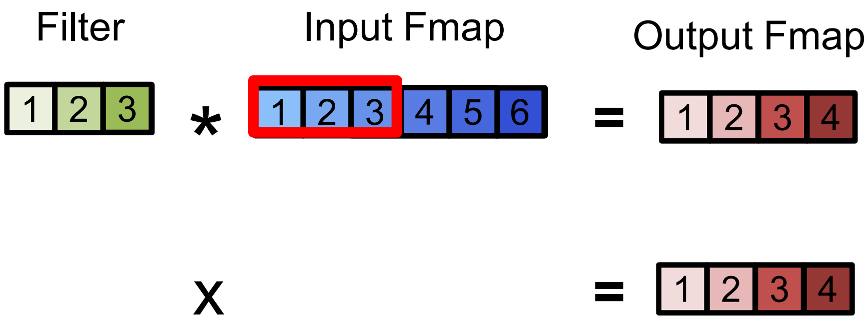
$$O_q = I_{q+s} \times F_s$$

Break into two steps

$$T_{q,s} = I_{q+s}$$

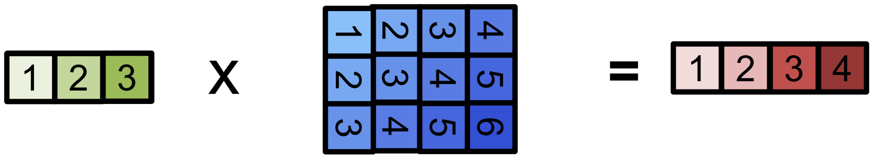
$$O_q = T_{q,s} \times F_s$$

1-D Toeplitz Convolution Einsum

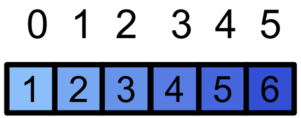


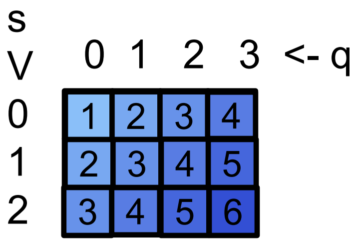
1-D Toeplitz Convolution Einsum

Filter Input Fmap Output Fmap



$$I_{q,s} = I_{q+s}$$





q	s	q+s
0	0	0
0	1	1
0	2	2
1	0	1
1	1	2
1	2	3
2	0	2
2	1	3

2-D Toeplitz Convolution Einsum

$$O_{m,p,q} = I_{c,p+r,q+s} \times F_{m,c,r,s}$$

Break out Toeplitz conversion

$$T_{c,p,q,r,s} = I_{c,p+r,q+s}$$

Flatten ranks

$$T'_{pq,crs} = T_{c,p,q,r,s}$$

$$F'_{m,crs} = F_{m,c,r,s}$$

$$O_{m,pq} = T'_{pq,crs} \times F'_{m,crs}$$

Next Lecture: GPUs

Thank you!