Survey of DNN Hardware

CICS/MTL Tutorial (2017)

Website: http://eyeriss.mit.edu/tutorial.html

Joel Emer, Vivienne Sze, Yu-Hsin Chen

CPUs Are Targeting Deep Learning

Intel Knights Landing (2016)

- 7 TFLOPS FP32
- 16GB MCDRAM– 400 GB/s
- 245W TDP
- 29 GFLOPS/W (FP32)
- 14nm process

Knights Mill: next gen Xeon Phi "optimized for deep learning"

Intel announced the addition of new vector instructions for deep learning (AVX512-4VNNIW and AVX512-4FMAPS), October 2016

Image Source: Intel, Data Source: Next Platform

GPUs Are Targeting Deep Learning

Nvidia PASCAL GP100 (2016)

- 10/20 TFLOPS FP32/FP16
- 16GB HBM – 750 GB/s
- 300W TDP
- 67 GFLOPS/W (FP16)
- 16nm process
- 160GB/s NV Link

Source: Nvidia

Systems for Deep Learning

Nvidia DGX-1 (2016)

- 170 TFLOPS
- 8× Tesla P100, Dual Xeon
- NVLink Hybrid Cube Mesh
- Optimized DL Software
- 7 TB SSD Cache
- Dual 10GbE, Quad IB 100Gb
- 3RU – 3200W

Source: Nvidia

Cloud Systems for Deep Learning

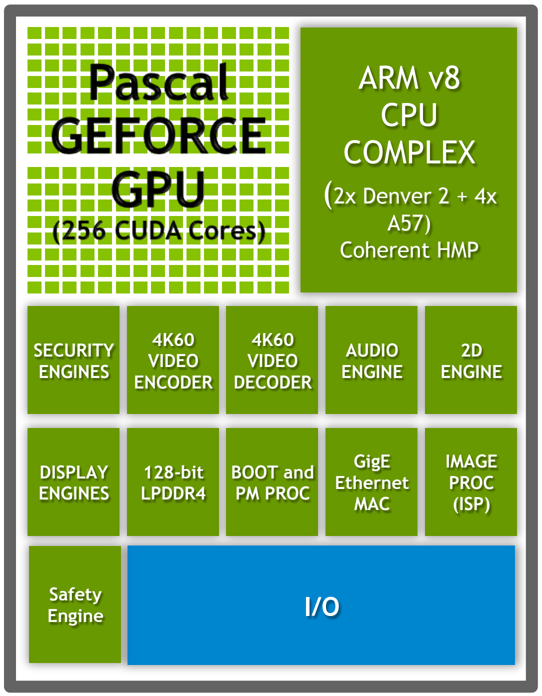
Facebook's Deep Learning Machine

- Open Rack Compliant
- Powered by 8 Tesla M40 GPUs
- 2x Faster Training for Faster Deployment
- 2x Larger Networks for Higher Accuracy

Source: Facebook

SOCs for Deep Learning Inference

Nvidia Tegra - Parker



- GPU: 1.5 TeraFLOPS FP16
- 4GB LPDDR4 @ 25.6 GB/s
- 15 W TDP
  (1W idle, <10W typical)
- 100 GFLOPS/W (FP16)
- 16nm process

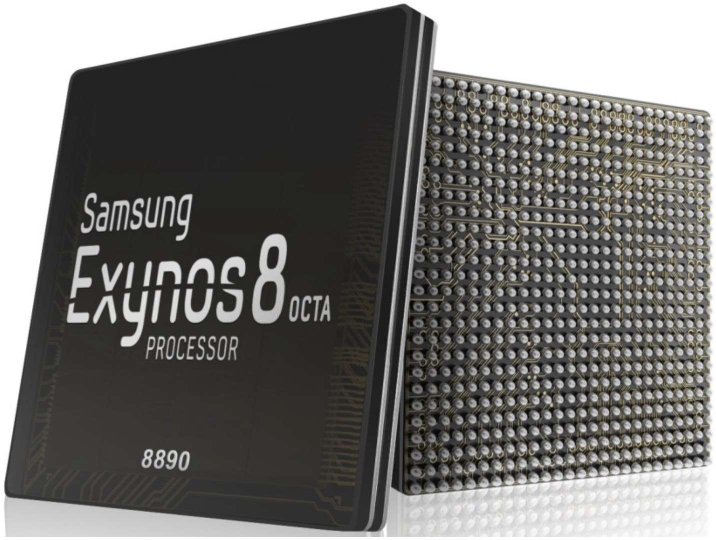
Xavier: next gen Tegra to be an "AI supercomputer"

Source: Nvidia

Mobile SOCs for Deep Learning

Samsung Exynos (ARM Mali)

Exynos 8 Octa 8890



- GPU: 0.26 TFLOPS
- LPDDR4 @ 28.7 GB/s
- 14nm process

Source: Wikipedia

FPGAs for Deep Learning

Intel/Altera Stratix 10

- 10 TFLOPS FP32
- HBM2 integrated
- Up to 1 GHz
- 14nm process
- 80 GFLOPS/W

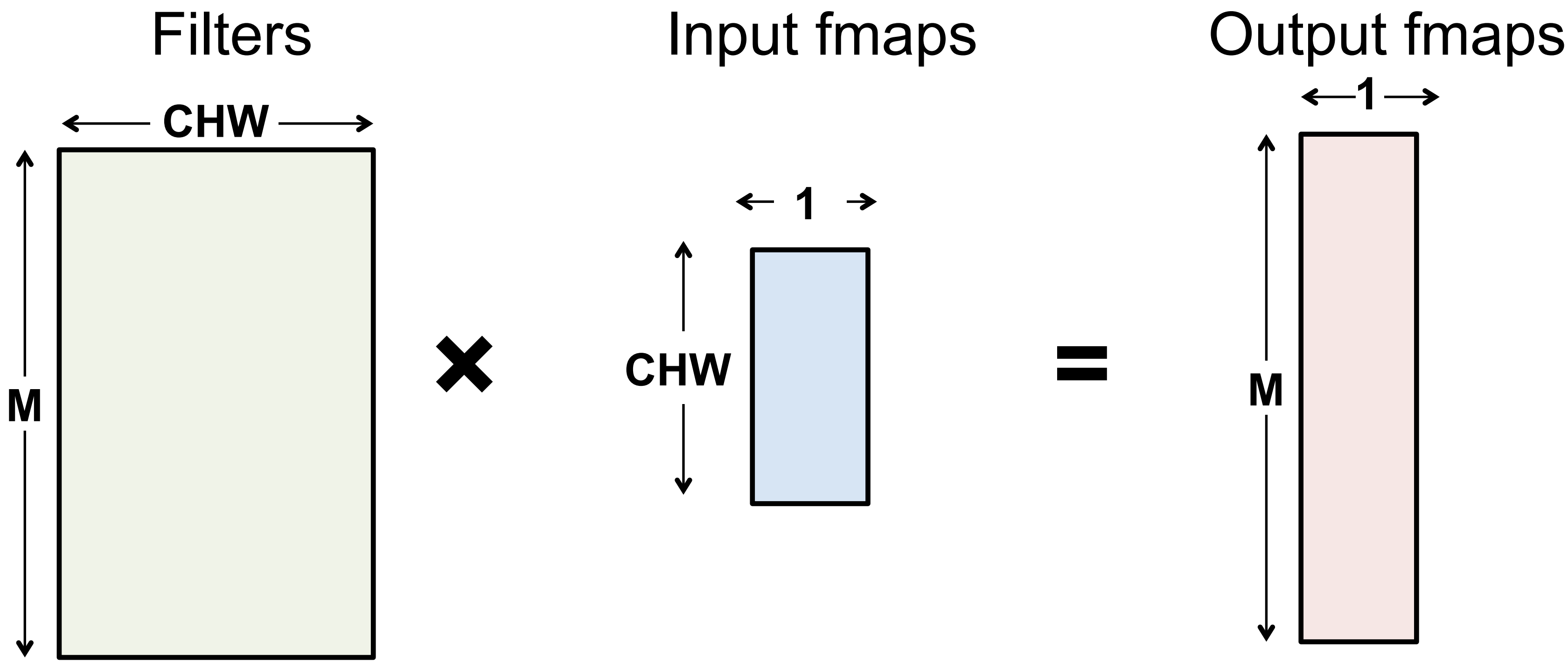
Xilinx Virtex UltraSCALE+

- DSP: up to 21.2 TMACS
- DSP: up to 890 MHz
- Up to 500Mb On-Chip Memory
- 16nm process

Kernel Computation

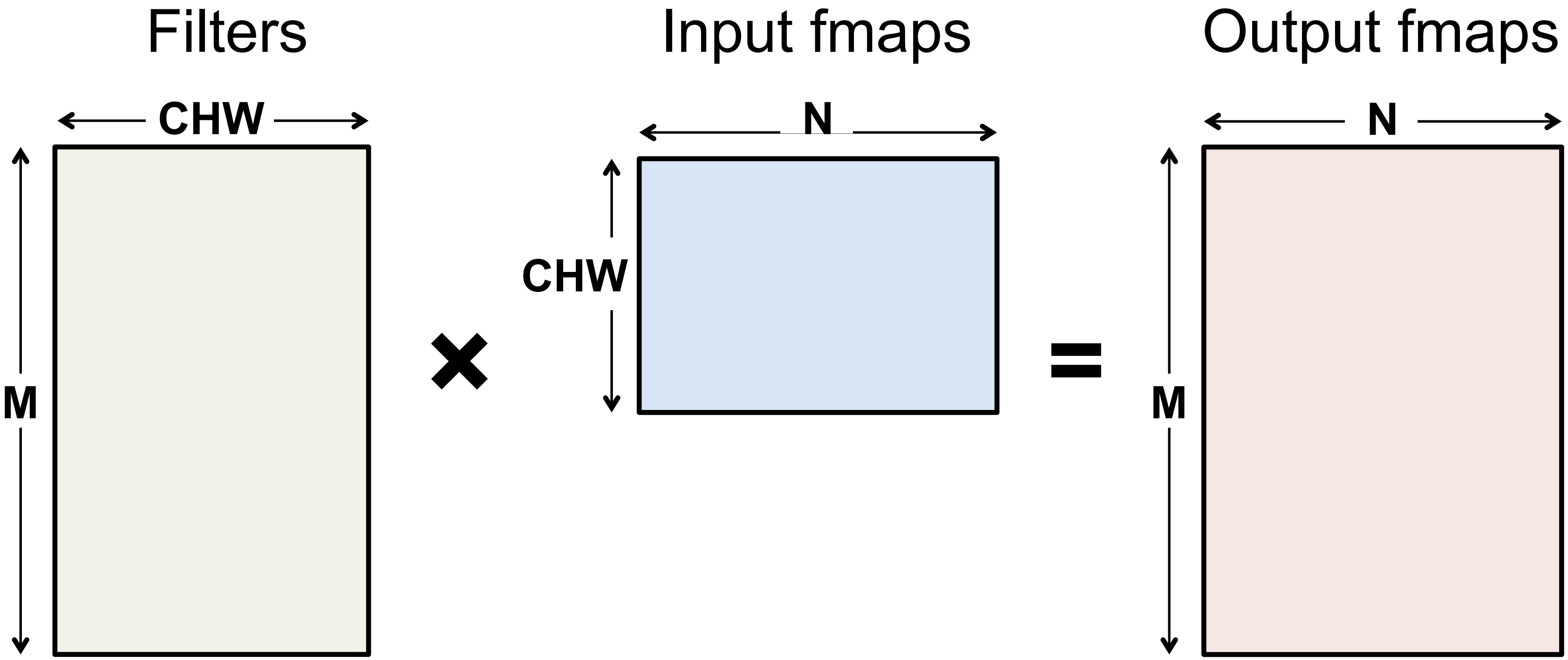
Fully-Connected (FC) Layer

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



Fully-Connected (FC) Layer

- Batching (N) turns operation into a Matrix-Matrix multiply



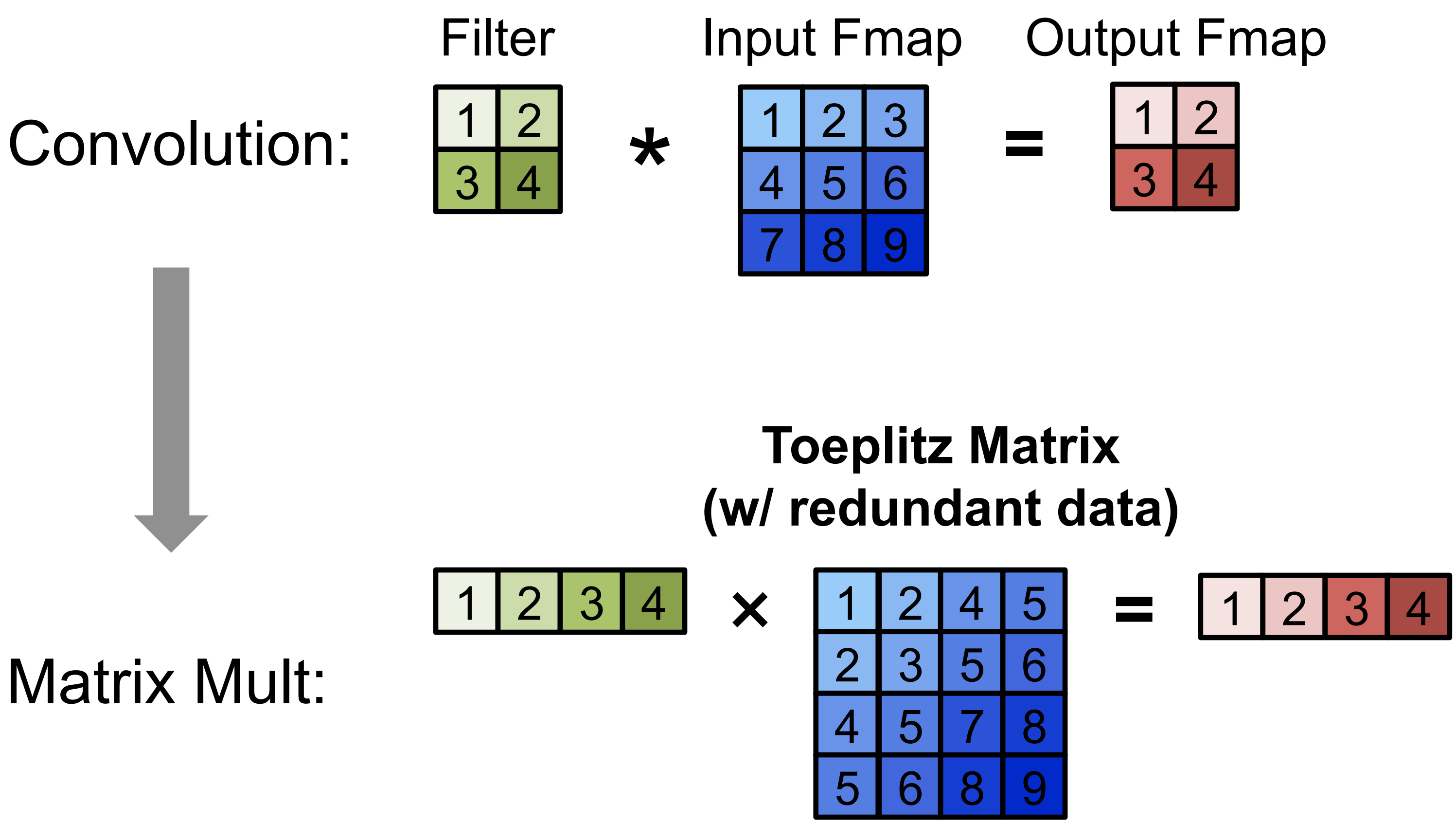
Fully-Connected (FC) Layer

- Implementation: **Matrix Multiplication (GEMM)**
  - **CPU:** OpenBLAS, Intel MKL, etc
  - **GPU:** cuBLAS, cuDNN, etc

- Optimized by tiling to storage hierarchy

Convolution (CONV) Layer

- Convert to matrix mult. using the **Toeplitz Matrix**



Convolution (CONV) Layer

- Convert to matrix mult. using the **Toeplitz Matrix**

Convolution:

Filter		Input Fmap		Output Fmap
$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$	$*$	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$	$=$	$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

Matrix Mult:

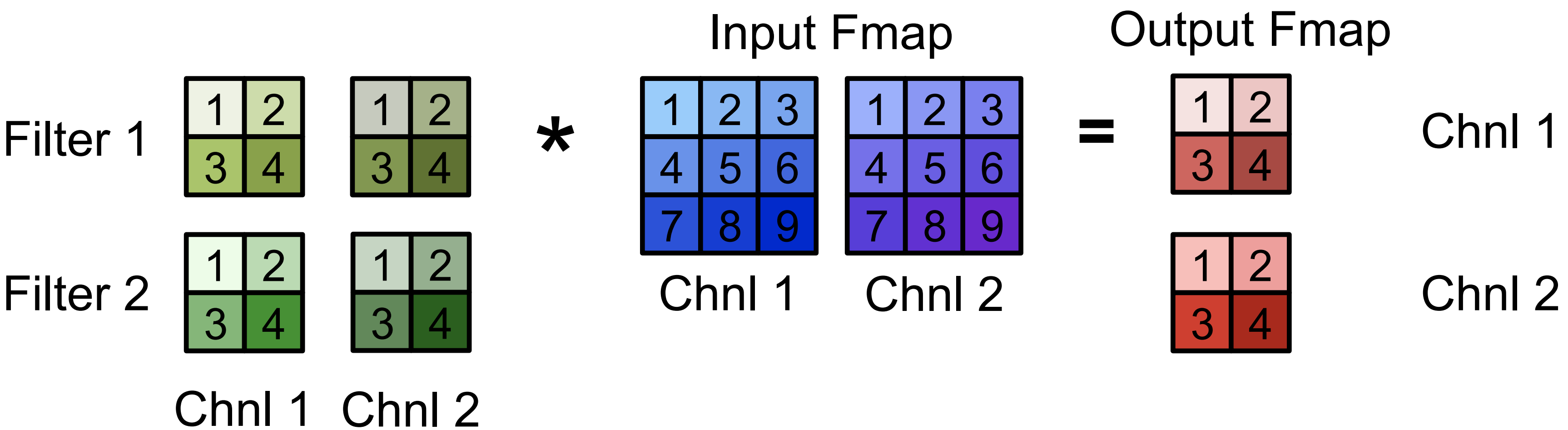
Toeplitz Matrix (w/ redundant data)

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 4 & 5 \\ 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 \\ 5 & 6 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

Data is repeated

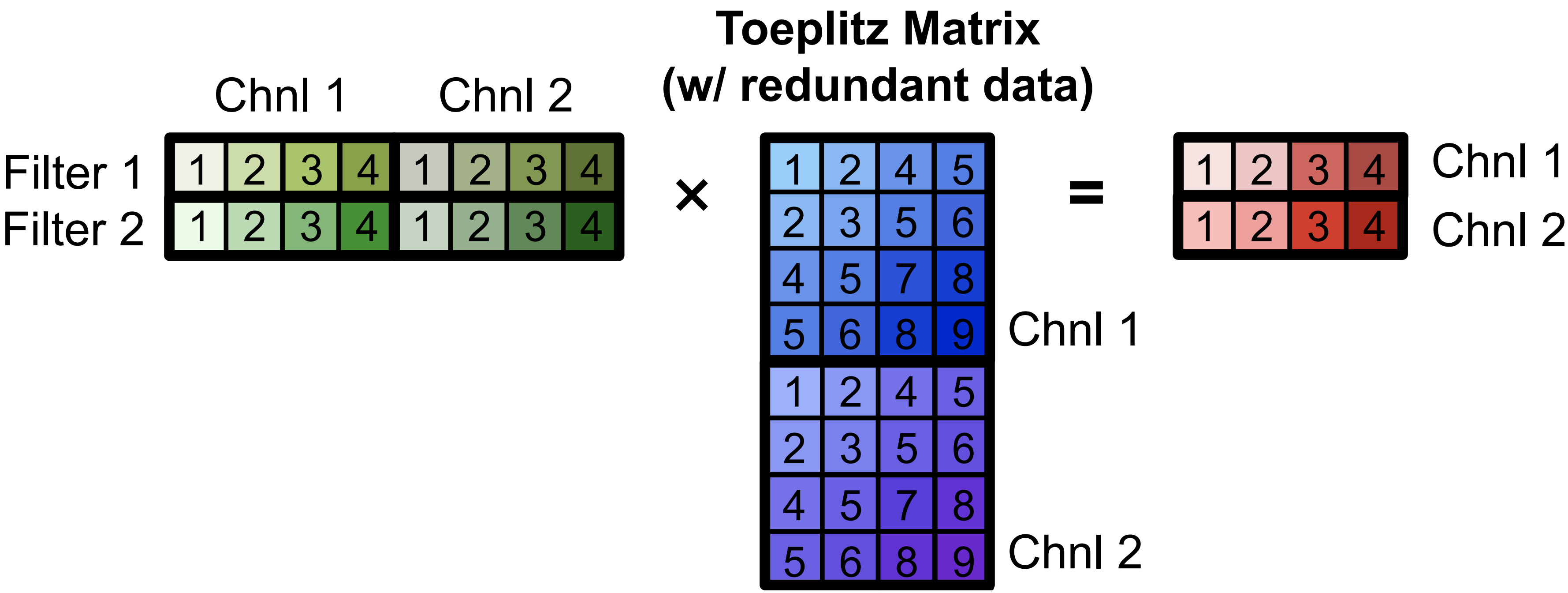
Convolution (CONV) Layer

- Multiple Channels and Filters



Convolution (CONV) Layer

- Multiple Channels and Filters



Computational Transforms

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

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

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)

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

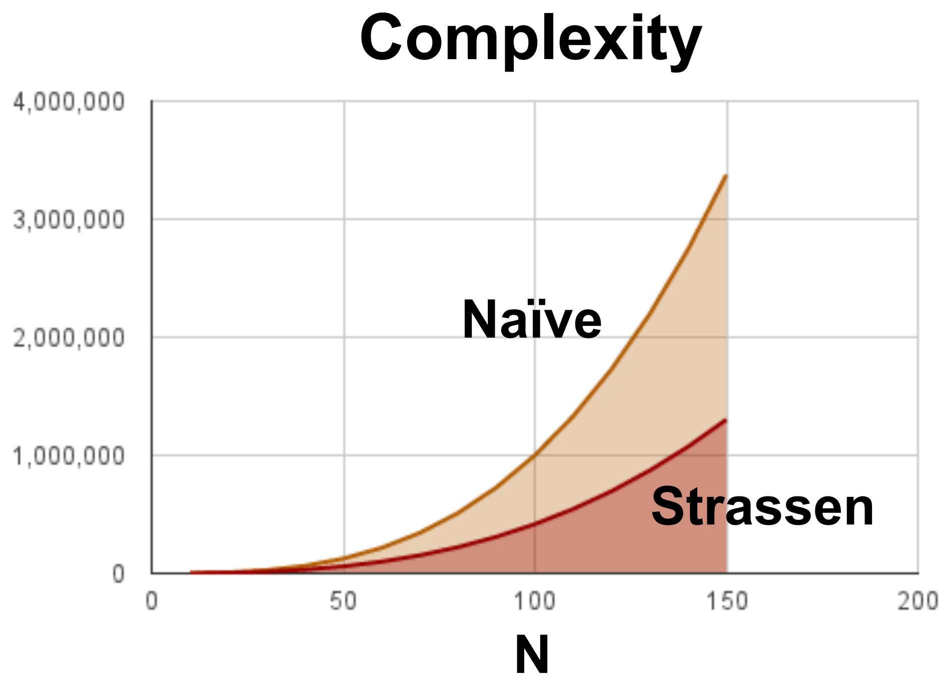
7 multiplications + 18 additions

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

[Cong et al., ICANN, 2014]

Strassen

- Reduce the complexity of matrix multiplication from **$\Theta(N^3)$** to **$\Theta(N^{2.807})$** by reducing multiplication



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/

Winograd 1D – F(2,3)

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

$$F(2,3) = \begin{array}{c} \text{input} \\ \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \end{array} \begin{array}{c} \text{filter} \\ \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} \end{array}$$

6 multiplications + 4 additions

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

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

4 multiplications + 12 additions + 2 shifts

4 multiplications + 8 additions (for constant weights)

[Lavin et al., ArXiv 2015]

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

- 1D Winograd is nested to make 2D Winograd

$$\overset{\text{Filter}}{\begin{bmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ g_{20} & g_{21} & g_{22} \end{bmatrix}} * \overset{\text{Input Fmap}}{\begin{bmatrix} d_{00} & d_{01} & d_{02} & d_{03} \\ d_{10} & d_{11} & d_{12} & d_{13} \\ d_{20} & d_{21} & d_{22} & d_{23} \\ d_{30} & d_{31} & d_{32} & d_{33} \end{bmatrix}} = \overset{\text{Output Fmap}}{\begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix}}$$

Original: 36 multiplications

Winograd: 16 multiplications → 2.25 times reduction

Winograd Halos

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

Filter

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

Input Fmap

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

Halo columns

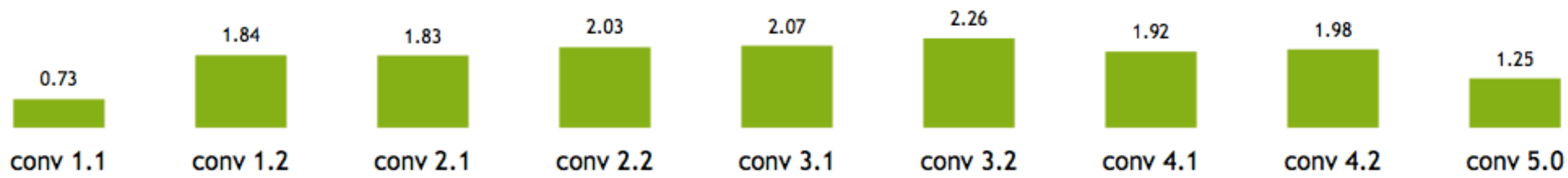
Output Fmap

y_{00}	y_{01}	y_{02}	y_{03}
y_{10}	y_{11}	y_{12}	y_{12}

Winograd Performance Varies

Optimal convolution algorithm depends on convolution layer dimensions

Winograd speedup over GEMM-based convolution (VGG-E layers, N=1)



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 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 $g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$

input $d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$

Transform inputs

$$Y = A^T \Big[[G g G^T] \odot [B^T d B] \Big] A$$

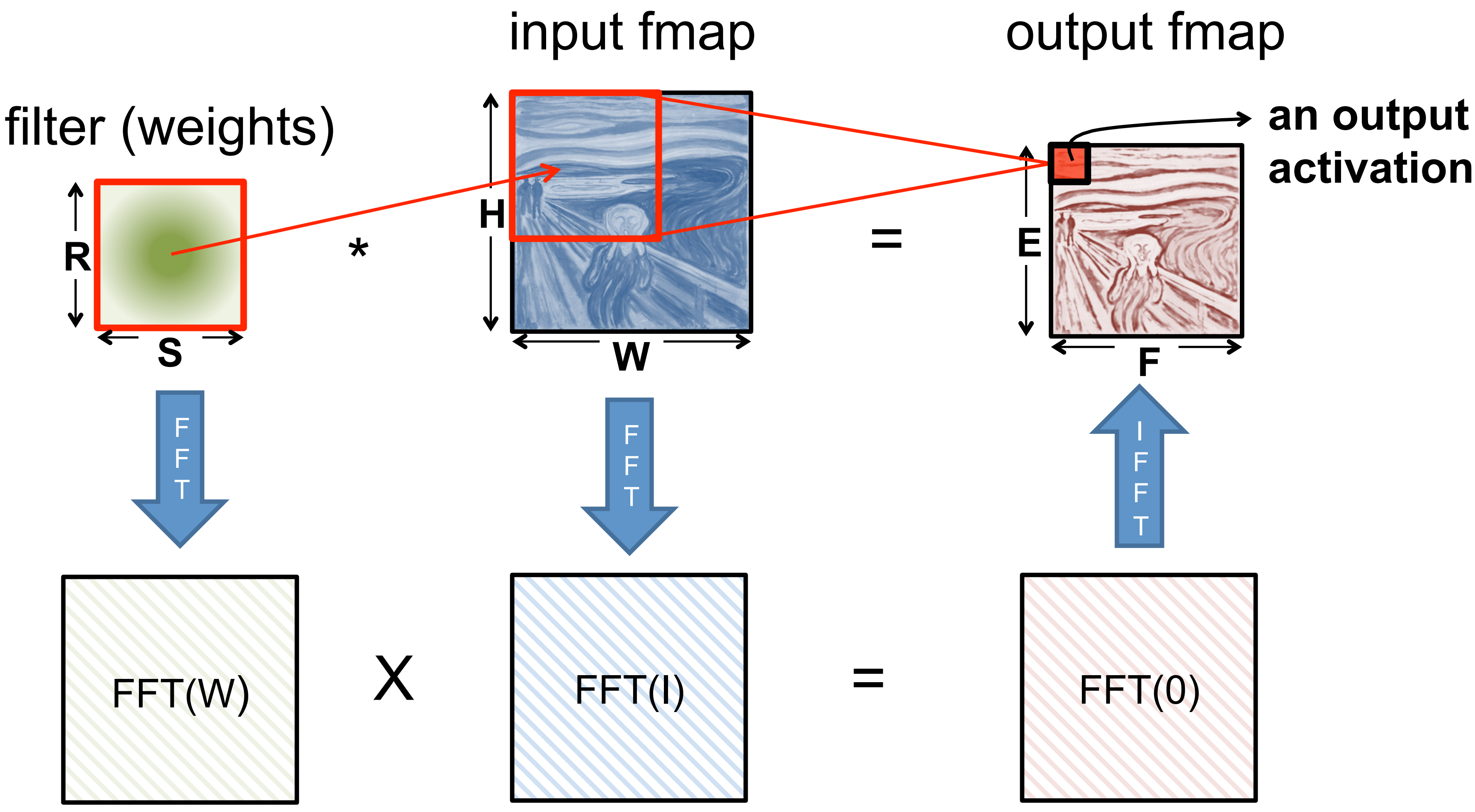
Dot-product

Transform output

GgG^T can be precomputed

[Lavin et al., ArXiv 2015]

FFT Flow


input fmap
output fmap
filter (weights)
an output activation
R
S
H
W
E
F
*
=
F F T
F F T
I F F T
FFT(W)
X
FFT(I)
=
FFT(0)

FFT Overview

- Convert filter and input to frequency domain to make convolution a simple multiply then convert back to time domain.

- Convert direct convolution $O(N_o^2 N_f^2)$ computation to $O(N_o^2 \log_2 N_o)$

- So note that computational benefit of FFT decreases with decreasing size of filter

[Mathieu et al., ArXiv 2013, Vasilache et al., ArXiv 2014]

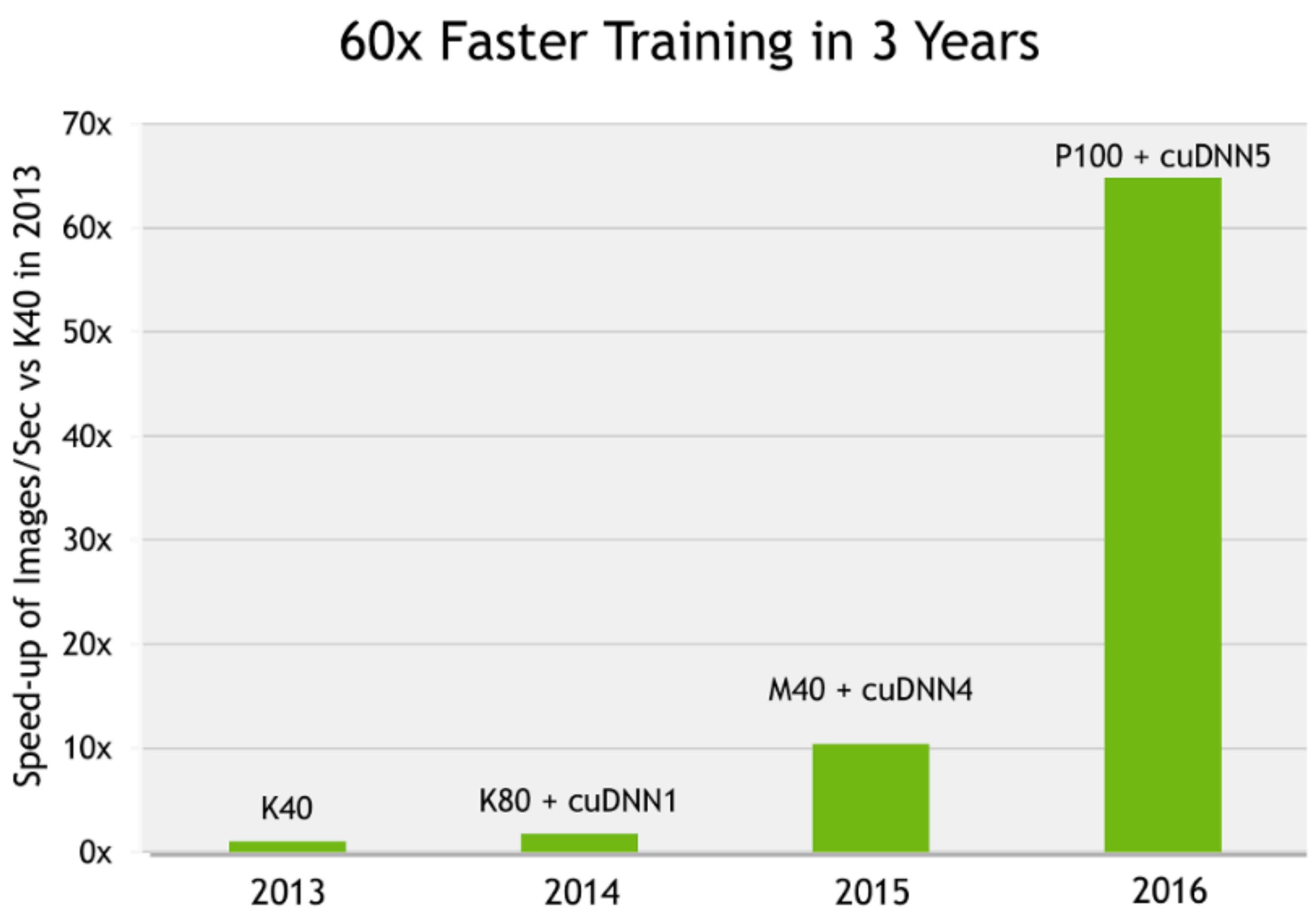
FFT Costs

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

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 time domain
- Can reuse frequency domain version of input for creating different output channels to avoid FFT re-computations

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