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
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

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:

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{array}
\times
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{array} =
\begin{array}{ccc}
1 & 2 & 3 \\
3 & 4 &
\end{array}
\]

Toeplitz Matrix (w/ redundant data)

\[
\begin{array}{ccccc}
1 & 2 & 3 & 4 \\
2 & 3 & 5 & 6 \\
4 & 5 & 7 & 8 \\
5 & 6 & 8 & 9
\end{array}
\times
\begin{array}{ccc}
1 & 2 & 4 & 5 \\
2 & 3 & 5 & 6 \\
4 & 5 & 7 & 8 \\
5 & 6 & 8 & 9
\end{array} =
\begin{array}{ccccc}
1 & 2 & 3 & 4 \\
2 & 3 & 4 &
\end{array}
\]
Convolution (CONV) Layer

• Convert to matrix mult. using the Toeplitz Matrix

\[
\begin{array}{ccc}
\text{Filter} & \times & \text{Input Fmap} \\
1 & 2 & 3 \\
3 & 4 & 6 \\
\end{array}
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{array}
= \begin{array}{cc}
1 & 2 \\
3 & 4 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array}
\begin{array}{cccc}
1 & 2 & 4 & 5 \\
2 & 3 & 5 & 6 \\
4 & 5 & 7 & 8 \\
5 & 6 & 8 & 9 \\
\end{array}
= \begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array}
\]

Toeplitz Matrix (w/ redundant data)

Data is repeated
Convolution (CONV) Layer

- Multiple Channels and Filters

Filter 1

Filter 2

Input Fmap

Output Fmap

Chnl 1

Chnl 2
Convolution (CONV) Layer

- Multiple Channels and Filters

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

\[k_1 = c \cdot (a + b)\]
\[k_2 = a \cdot (d - c)\]
\[k_3 = b \cdot (c + d)\]

Real part = \(k_1 - k_3\)
Imaginary part = \(k_1 + k_2.\)

- 3 multiplications + 5 additions
Strassen

\[ \begin{align*}
\text{P1} &= a(f - h) \\
\text{P2} &= (a + b)h \\
\text{P3} &= (c + d)e \\
\text{P4} &= d(g - e) \\
\text{P5} &= (a + d)(e + h) \\
\text{P6} &= (b - d)(g + h) \\
\text{P7} &= (a - c)(e + f)
\end{align*} \]

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

8 multiplications + 4 additions

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

Complexity

Naïve
Strassen

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

Winograd 1D – F(2,3)

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

\[
F(2, 3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix}
\]

6 multiplications + 4 additions

\[
m_1 = (d_0 - d_2)g_0 \\
m_2 = (d_1 + d_2) \frac{g_0 + g_1 + g_2}{2} \\
m_4 = (d_1 - d_3)g_2 \\
m_3 = (d_2 - d_1) \frac{g_0 - g_1 + g_2}{2}
\]

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

\[
\begin{array}{c|c|c|c|}
\text{Filter} & \text{Input Fmap} & \ast & \text{Output Fmap} \\
\hline
\begin{array}{c|c|c|}
\text{g}_{00} & \text{g}_{01} & \text{g}_{02} \\
\text{g}_{10} & \text{g}_{11} & \text{g}_{12} \\
\text{g}_{20} & \text{g}_{21} & \text{g}_{22} \\
\end{array} & \begin{array}{c|c|c|c|}
\text{d}_{00} & \text{d}_{01} & \text{d}_{02} & \text{d}_{03} \\
\text{d}_{10} & \text{d}_{11} & \text{d}_{12} & \text{d}_{13} \\
\text{d}_{20} & \text{d}_{21} & \text{d}_{22} & \text{d}_{23} \\
\text{d}_{30} & \text{d}_{31} & \text{d}_{32} & \text{d}_{33} \\
\end{array} = & \begin{array}{c|c|}
\text{y}_{00} & \text{y}_{01} \\
\text{y}_{10} & \text{y}_{11} \\
\end{array}
\end{array}
\]

**Original:** 36 multiplications

**Winograd:** 16 multiplications \(\rightarrow\) 2.25 times reduction
Winograd Halos

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

\[
\begin{array}{ccc}
g_{00} & g_{01} & g_{02} \\
g_{10} & g_{11} & g_{12} \\
g_{20} & g_{21} & g_{22}
\end{array}
\]

\[
\begin{array}{ccccccc}
d_{00} & d_{01} & d_{02} & d_{03} & d_{04} & d_{05} \\
d_{10} & d_{11} & d_{12} & d_{13} & d_{14} & d_{15} \\
d_{20} & d_{21} & d_{22} & d_{23} & d_{24} & d_{25} \\
d_{30} & d_{31} & d_{32} & d_{33} & d_{34} & d_{35}
\end{array}
\]

\[
\begin{array}{cccc}
y_{00} & y_{01} & y_{02} & y_{03} \\
y_{10} & y_{11} & y_{12} & y_{12}
\end{array}
\]

Halo columns
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)
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

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

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

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

filter \[ g = [g_0 \ g_1 \ g_2]^T \]

input \[ d = [d_0 \ d_1 \ d_2 \ d_3]^T \]

\[ Y = A^T \left[ [GgG^T] \odot [B^T dB] \right] A \]

Transform inputs

Dot-product

Transform output

GgG^T can be precomputed

[Lavin et al., ArXiv 2015]
FFT Flow

filter (weights)

input fmap

output fmap

an output activation

FFT(W) \times 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 $O(N_o^2N_f^2)$ computation to $O(N_o^2\log_2N_o)$

- 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 $E+R-1 \times F+S-1$

• 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 $\frac{1}{2}$ 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

60x Faster Training in 3 Years

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