CMSC 5743

Efficient Computing of Deep Neural Networks

Implementation 02: GEMM-2

Bei Yu
CSE Department, CUHK
byu@cse.cuhk.edu.hk

(Latest update: February 17, 2023)

Spring 2023

Overview

Overview

1. Strassen
2. Winograd
3. Dataflow Optimization

Strassen

Algorithm 1 Naive matrix multiplication

Input: $A, B \in \mathbb{R}^{n\times n}$
Output: AB
  for $i = 1$ to n **do**
    for $j = 1$ to n **do**
      Set $C_{ij} = \sum_{t=1}^{n} A_{it}B_{tj}$
    end for
  end for
  return C

- Time Complexity: $O(N^3)$

Blockwise Matrix Multiplication

To compute $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$, we first partition $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ into equal=sized blocked matrices such that

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

where $\boldsymbol{A}_{ij}, \boldsymbol{B}_{ij}, \boldsymbol{C}_{ij} \in \mathbb{R}^{\frac{N}{2} \times \frac{N}{2}}$. We then have:

$$\begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} & \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} & \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{bmatrix}$$

Algorithm 2 Recursive matrix multiplication

Input: $A, B \in \mathbb{R}^{n\times n}$
Output: AB

```
function M(A,B)
    if A is 1 × 1 then
        return a_11 b_11
    end if
    for i = 1 to 2 do
        for j = 1 to 2 do
            Set C_ij = M(A_i1, B_1j) + M(A_i2, B_2j)
        end for
    end for
    return (C_11 C_12; C_21 C_22)
end function
```

function $M(A,B)$
  if A is 1×1 **then**
    return $a_{11}b_{11}$
  end if
  for $i = 1$ to 2 **do**
    for $j = 1$ to 2 **do**
      Set $C_{ij} = M(A_{i1}, B_{1j}) + M(A_{i2}, B_{2j})$
    end for
  end for
  return $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$
end function

Time Complexity Analysis

The recursive algorithm can be formulated as:

$$T(N) = \begin{cases} \Theta(1) & \text{if } N = 1 \\ 8T(\frac{N}{2}) + \Theta(N^2) & \text{if } N > 1 \end{cases}$$

This algorithm makes eight recursive calls. Besides, it also adds two $n \times n$ matrices, which requires n^2 time. By Master Theorem, the time complexity of the recursive algorithm is: $T(n) = O(N^{\log_2^8}) = O(N^3)$.

Suppose we need to calculate matrix multiplication $\boldsymbol{M} \times \boldsymbol{N}$, following the idea of blockwise multiplication, we can first split the matrices into:

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \qquad N = \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

Then, we calculate the intermediate matrices:

$$\begin{aligned}
S_1 &= (B - D)(G + H) \\
S_2 &= (A + D)(E + H) \\
S_3 &= (A - C)(E + F) \\
S_4 &= (A + B)H \\
S_5 &= A(F - H) \\
S_6 &= D(G - E) \\
S_7 &= (C + D)E.
\end{aligned}$$

The final results are:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 - S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}.$$

Algorithm 3 Strassen's Algorithm

```
function STRASSEN(M,N)
    if M is 1 × 1 then
        return M_11 N_11
    end if
```

$$\text{Let } M = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \text{ and } N = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

Set $S_1 = \textsc{Strassen}(B - D, G + H)$

Set $S_2 = \textsc{Strassen}(A + D, E + H)$

Set $S_3 = \textsc{Strassen}(A - C, E + F)$

Set $S_4 = \textsc{Strassen}(A + B, H)$

Set $S_5 = \textsc{Strassen}(A, F - H)$

Set $S_6 = \textsc{Strassen}(D, G - E)$

Set $S_7 = \textsc{Strassen}(C + D, E)$

$$\textbf{return} \begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 - S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}$$

end function

Strassen algorithm makes seven recursive calls. Besides, the additions and subtractions take N^2 time. Therefore, Strassen algorithm can be formulated as:

$$T(N) = \begin{cases} \Theta(1) & \text{if } N = 1 \\ 7T(\frac{N}{2}) + \Theta(N^2) & \text{if } N > 1 \end{cases}$$

By Master Theorem, the time complexity of the recursive algorithm is:
$T(n) = O(N^{\log_2^7}) = O(N^{2.8074})$.

Strassen Algorithm

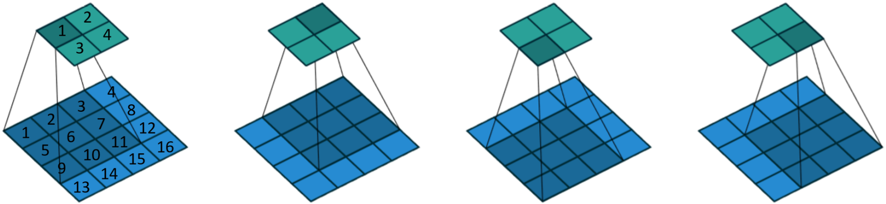
Matrix size	w/o Strassen	w/ Strassen
$(256, 256, 256)$	23	23
$(512, 512, 512)$	191	**176** ($\downarrow$ 7.9%)
$(512, 512, 1024)$	388	**359** ($\downarrow$ 7.5%)
$(1024, 1024, 1024)$	1501	**1299** ($\downarrow$ 13.5%)

```
class XPUBackend final : public Backend {
    XPUBackend(MNNForwardType type, MemoryMode mode);
    virtual ~XPUBackend();
    virtual Execution* onCreate(const vector<Tensor*>& inputs,
                                const vector<Tensor*>& outputs, const MNN::Op* op);
    virtual void onExecuteBegin() const;
    virtual void onExecuteEnd() const;
    virtual bool onAcquireBuffer(const Tensor* tensor, StorageType storageType);
    virtual bool onReleaseBuffer(const Tensor* tensor, StorageType storageType);
    virtual bool onClearBuffer();
    virtual void onCopyBuffer(const Tensor* srcTensor, const Tensor* dstTensor) const;
}
```

Overview

Winograd

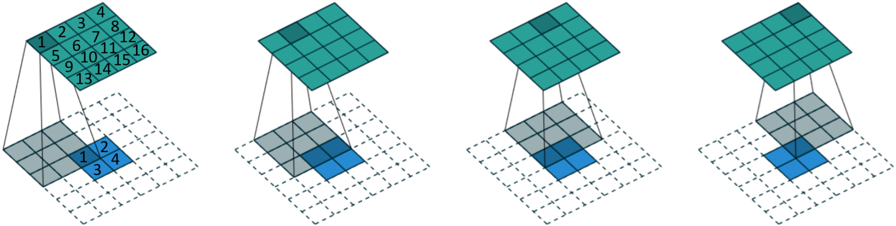
What is Convolution?



The calculation process of convolutional layer

- No padding
- Unit strides
- 3×3 kernel size
- 4×4 input feature map

What is Deconvolution (transposed convolution)?[1]



The calculation process of deconvolutional layer

- 2×2 padding with border of zeros
- Unit strides
- 3×3 kernel size
- 4×4 input feature map

[1]Vincent Dumoulin and Francesco Visin (2016). "A guide to convolution arithmetic for deep learning". In: *arXiv preprint arXiv:1603.07285*.

4. Fast Algorithms

It has been known since at least 1980 that the minimal filtering algorithm for computing m outputs with an r-tap FIR filter, which we call $F(m, r)$, requires

$$\mu(F(m,r)) = m + r - 1 \tag{3}$$

multiplications [16, p. 39]. Also, we can nest minimal 1D algorithms $F(m, r)$ and $F(n, s)$ to form minimal 2D algorithms for computing $m \times n$ outputs with an $r \times s$ filter, which we call $F(m \times n, r \times s)$. These require

$$\begin{aligned} \mu(F(m \times n, r \times s)) &= \mu(F(m,r))\mu(F(n,s)) \\ &= (m+r-1)(n+s-1) \end{aligned} \tag{4}$$

[2] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *Proc. CVPR*, pp. 4013–4021.

Winograd Algorithm[2]

The standard algorithm for $F(2,3)$ uses $2 \times 3 = 6$ multiplications. Winograd [16, p. 43] documented the following minimal algorithm:

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

where

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

[2] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *Proc. CVPR*, pp. 4013–4021.

Winograd Algorithm[2]

Fast filtering algorithms can be written in matrix form as:

$$Y = A^T \left[(Gg) \odot (B^T d) \right] \quad (6)$$

where $\odot$ indicates element-wise multiplication. For $F(2,3)$, the matrices are:

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

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad (7)$$

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

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

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

[2] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *Proc. CVPR*, pp. 4013–4021.

Generalization to 2D cases:
Suppose the input feature map is

$$D = \begin{bmatrix} d_{00} & d_{01} & d_{02} & d_{03} \\ d_{10} & d_{11} & d_{12} & d_{13} \\ d_{20} & d_{21} & d_{22} & d_{23} \\ d_{30} & d_{31} & d_{32} & d_{33} \end{bmatrix}$$

and the kernel is:

$$K = \begin{bmatrix} k_{00} & k_{01} & k_{02} \\ k_{10} & k_{11} & k_{12} \\ k_{20} & k_{21} & k_{22} \end{bmatrix}$$

[3] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *Proc. CVPR*, pp. 4013–4021.

Winograd Algorithm[4]

Using Im2Col function, the convolution process can be defined as:

$$\begin{bmatrix} d_{00} & d_{01} & d_{02} & d_{10} & d_{11} & d_{12} & d_{20} & d_{21} & d_{22} \\ d_{01} & d_{02} & d_{03} & d_{11} & d_{12} & d_{13} & d_{21} & d_{22} & d_{23} \\ d_{10} & d_{11} & d_{12} & d_{20} & d_{21} & d_{22} & d_{30} & d_{31} & d_{32} \\ d_{11} & d_{12} & d_{13} & d_{21} & d_{22} & d_{23} & d_{31} & d_{32} & d_{33} \end{bmatrix} \begin{bmatrix} k_{00} \\ k_{01} \\ k_{02} \\ k_{10} \\ k_{11} \\ k_{12} \\ k_{20} \\ k_{21} \\ k_{22} \end{bmatrix} = \begin{bmatrix} r_{00} \\ r_{01} \\ r_{10} \\ r_{11} \end{bmatrix}$$

[4] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *Proc. CVPR*, pp. 4013–4021.

Winograd Algorithm[5]

We can split the matrices into blocks as:

$$\left[\begin{array}{ccc|ccc|ccc} d_{00} & d_{01} & d_{02} & d_{10} & d_{11} & d_{12} & d_{20} & d_{21} & d_{22} \\ d_{01} & d_{02} & d_{03} & d_{11} & d_{12} & d_{13} & d_{21} & d_{22} & d_{23} \\ \hline d_{10} & d_{11} & d_{12} & d_{20} & d_{21} & d_{22} & d_{30} & d_{31} & d_{32} \\ d_{11} & d_{12} & d_{13} & d_{21} & d_{22} & d_{23} & d_{31} & d_{32} & d_{33} \end{array}\right] \left[\begin{array}{c} k_{00} \\ k_{01} \\ k_{02} \\ \hline k_{10} \\ k_{11} \\ k_{12} \\ \hline k_{20} \\ k_{21} \\ k_{22} \end{array}\right] = \left[\begin{array}{c} r_{00} \\ r_{01} \\ \hline r_{10} \\ r_{11} \end{array}\right]$$

which can be denoted as:

$$\begin{bmatrix} D_{00} & D_{10} & D_{20} \\ D_{10} & D_{20} & D_{30} \end{bmatrix} \begin{bmatrix} \vec{k_0} \\ \vec{k_1} \\ \vec{k_2} \end{bmatrix} = \begin{bmatrix} \vec{r_0} \\ \vec{r_1} \end{bmatrix}$$

[5] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks".

Winograd Algorithm[6]

Then, the we can use 1D winograd algorithm to calculate the blockwise result:

$$\begin{bmatrix} D_{00} & D_{10} & D_{20} \\ D_{10} & D_{20} & D_{30} \end{bmatrix} \begin{bmatrix} \overrightarrow{k_0} \\ \overrightarrow{k_1} \\ \overrightarrow{k_2} \end{bmatrix} = \begin{bmatrix} \overrightarrow{r_0} \\ \overrightarrow{r_1} \end{bmatrix} = \begin{bmatrix} M_0 + M_1 + M_2 \\ M_1 - M_2 - M_3 \end{bmatrix}$$

where

$$\begin{aligned} M_0 &= (D_{00} - D_{20})\overrightarrow{k_0} \\ M_1 &= (D_{10} + D_{20})\frac{\overrightarrow{k_0} + \overrightarrow{k_1} + \overrightarrow{k_2}}{2} \\ M_2 &= (D_{20} - D_{10})\frac{\overrightarrow{k_0} - \overrightarrow{k_1} + \overrightarrow{k_2}}{2} \\ M_3 &= (D_{10} - D_{30})\overrightarrow{k_2} \end{aligned}$$

[6] Andrew Lavin and Scott Gray (2016). “Fast Algorithms for Convolutional Neural Networks”. In: *Proc. CVPR*, pp. 4013–4021.

Winograd Algorithm[7]

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 $F(n, s)$.

$F(2\times2, 3\times3)$ uses $4\times4 = 16$ multiplications, whereas the standard algorithm uses $2 \times 2 \times 3 \times 3 = 36$. This

[7] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *Proc. CVPR*, pp. 4013–4021.

The transforms for $F(3 \times 3, 2 \times 2)$ are given by:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}, G = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \\ 0 & 1 \end{bmatrix} \tag{14}$$

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

With $(3+2-1)^2 = 16$ multiplies versus direct convolution's $3 \times 3 \times 2 \times 2 = 36$ multiplies, it achieves the same $36/16 = 2.25$ arithmetic complexity reduction as the corresponding forward propagation algorithm.

[7] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *Proc. CVPR*, pp. 4013–4021.

Winograd Algorithm[7]

4.3. F(4x4,3x3)

A minimal algorithm for $F(4,3)$ has the form:

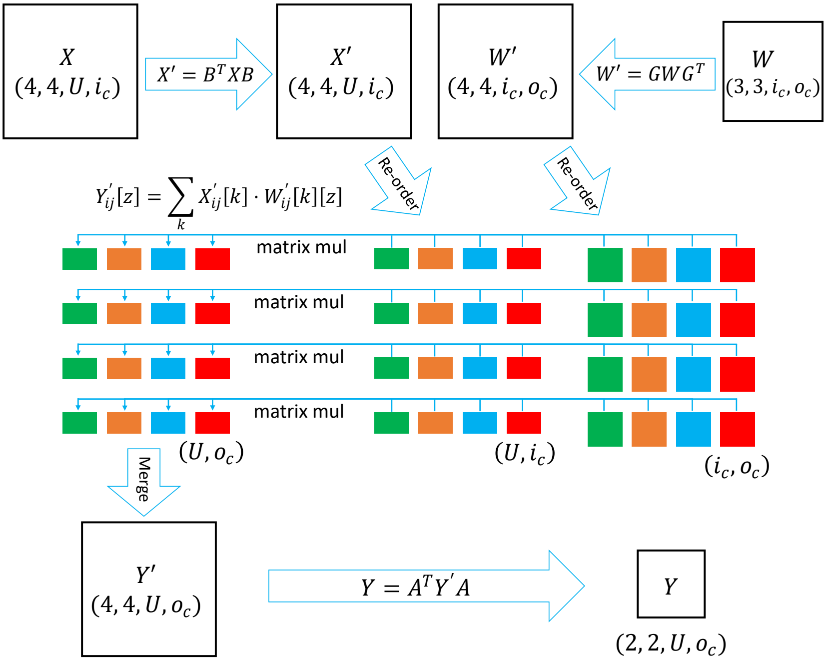
$$
\begin{aligned}
B^T &= \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix} \\
G &= \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix} \\
A^T &= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix}
\end{aligned}
\tag{15}
$$

The data transform uses 12 floating point instructions, the filter transform uses 8, and the inverse transform uses 10.

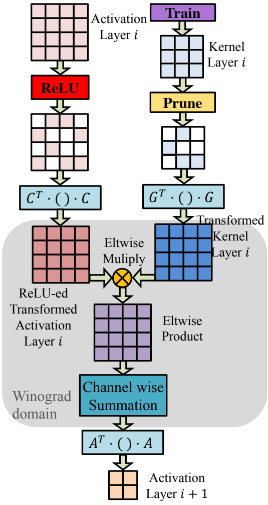
Applying the nesting formula yields a minimal algorithm for $F(4 \times 4, 3 \times 3)$ that uses $6 \times 6 = 36$ multiplies, while the standard algorithm uses $4 \times 4 \times 3 \times 3 = 144$. This is an arithmetic complexity reduction of 4.

[7] Andrew Lavin and Scott Gray (2016). “Fast Algorithms for Convolutional Neural Networks”. In: *Proc. CVPR*, pp. 4013–4021.

Optimized Winograd algorithm in MNN



Training in the Winograd Domain



(a)

Producing 4 output pixels:

Direct Convolution:

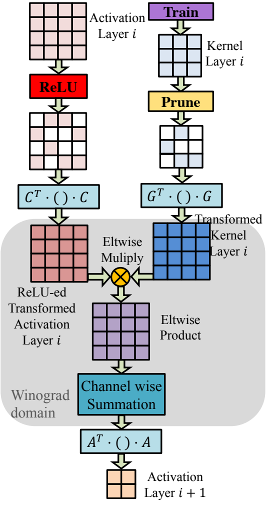
- 4*9=36 multiplications (**1x**)

Winograd convolution:

- 4*4=16 multiplications (**2.25x** less)

Liu et al. “Efficient Sparse-Winograd Convolutional Neural Networks”, submitted to ICLR 2017 workshop

Training in the Winograd Domain



(a)

Producing 4 output pixels:

Direct Convolution:

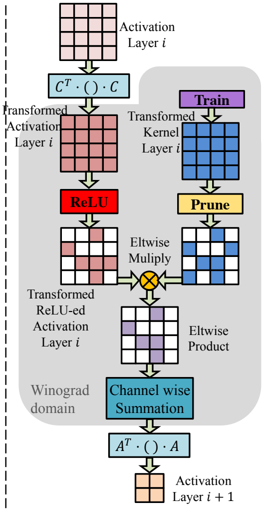
- 4*9=36 multiplications (**1x**)
- sparse weight [NIPS'15] (**3x**)
- sparse activation (relu) (**3x**)
- Overall saving: **9x**

Winograd convolution:

- 4*4=16 multiplications (**2.25x** less)
- dense weight (**1x**)
- dense activation (**1x**)
- Overall saving: **2.25x**

Liu et al. "Efficient Sparse-Winograd Convolutional Neural Networks", submitted to ICLR 2017 workshop

Solution: Fold Relu into Winograd



Producing 4 output pixels:

Direct Convolution:

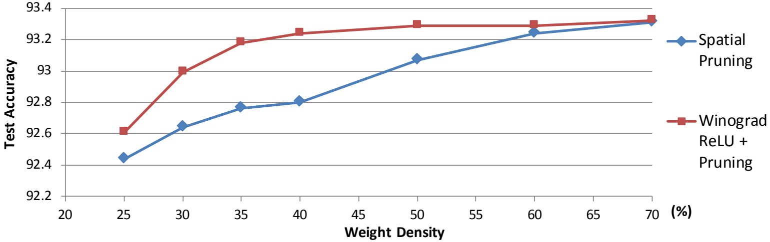
- 4*9=36 multiplications (**1x**)
- sparse weight [NIPS'15] (**3x**)
- sparse activation (relu) (**3x**)
- Overall saving: **9x**

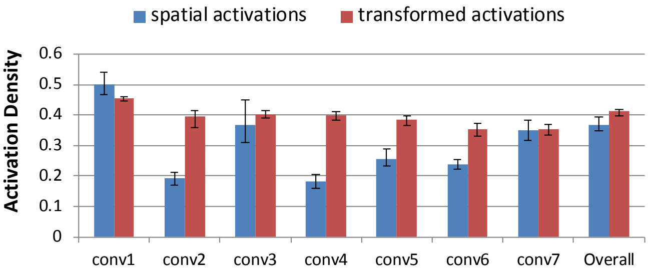
Winograd convolution:

- 4*4=16 multiplications (**2.25x** less)
- sparse weight (**2.5x**)
- dense activation (**2.25x**)
- Overall saving: **12x**

Liu et al. "Efficient Sparse-Winograd Convolutional Neural Networks", submitted to ICLR 2017 workshop

Result





Liu et al. “Efficient Sparse-Winograd Convolutional Neural Networks”, submitted to ICLR 2017 workshop

Overview

1. Strassen
2. Winograd
3. Dataflow Optimization

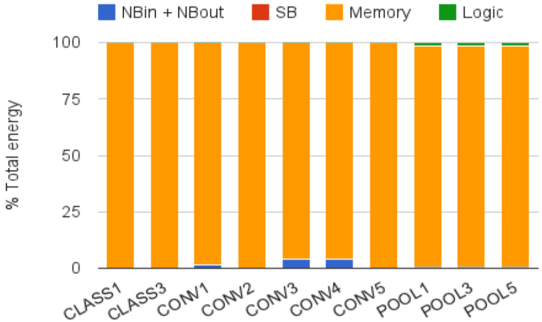
Dataflow Optimization

Case Study 2

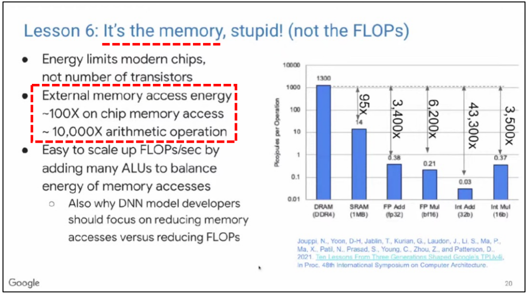
Communication Lower Bound in CNN Accelerators

Memory Bottleneck in CNN Accelerators

- **Memory access consumes most of total energy**
- **CNN accelerators are mostly memory dominant**



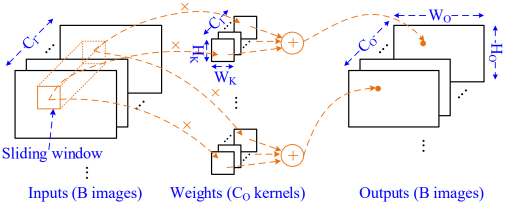
T. Chen et al., DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning, ASPLOS'14



Google slide, one of ten lessons learned from three generations TPUs

Convolutional Layer

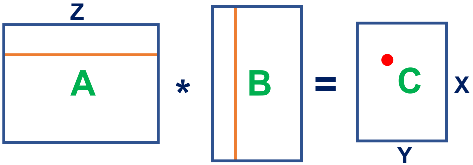
- **Complicated data reuse**
  - **Input reuse**
  - **Sliding window reuse**
  - **Weight reuse**
  - **Output reuse**
- **Finding minimum communication is difficult: huge search space caused by 7 levels of loops and complex data reuse schemes**




```
for (i = 0; i < B; i++)                  //Images in a batch
 for (oz = 0; oz < Co; oz++)             //Output channels
  for (oy = 0; oy < Ho; oy++)            //Output rows
   for (ox = 0; ox < Wo; ox++)           //Output columns
    for (kz = 0; kz < Ci; kz++)          //Input channels
     for (ky = 0; ky < Hk; ky++)         //Kernel rows
      for (kx = 0; kx < Wk; kx++)        //Kernel columns
       out[i][oz][oy][ox] +=
        in[i][kz][oy+ky][ox+kx] * w[oz][kz][ky][kx];
```

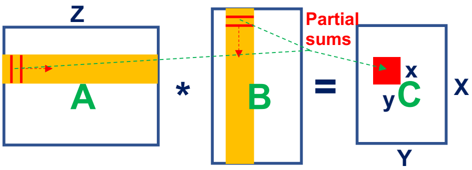
Communication in Matrix Multiplication

- **Naive matrix multiplication**



$$Q = 2XYZ + XY \approx 2XYZ$$

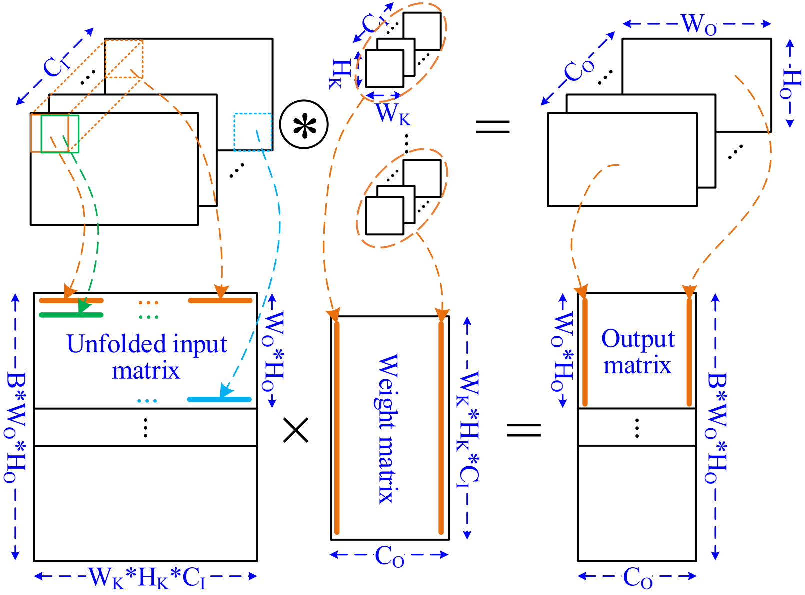
- **Communication-optimal matrix multiplication**



$$Q = \frac{XY}{xy}(xZ + yZ) + XY \approx XYZ\left(\frac{1}{x} + \frac{1}{y}\right) \geq \frac{2XYZ}{\sqrt{xy}} \geq \frac{2XYZ}{\sqrt{S}}$$

S: on-chip memory capacity

Relation between Convolution & Matrix Multiplication (im2col)



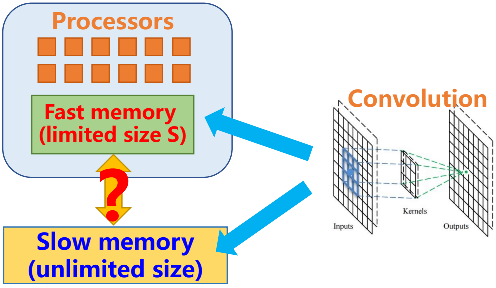
Observations

- Weights and outputs are just reshaped ---- without adding or removing elements
- Inputs are unfolded ---- all sliding windows (having overlapped elements) are explicitly expanded
- Convolution has only one more level of data reuse (sliding window reuse) than matrix multiplication

Communication-optimal convolution
= communication-optimal matrix multiplication + sliding window reuse?

Communication Lower Bound of Convolution

- **Matrix multiplication only used to inspire derivation process, there is not an actual conversion in our implementation**
- **Theoretical derivation based on Red-Blue Pebble Game [1]**



$$Q = \Omega\left(\frac{BW_O H_O C_O W_K H_K C_I}{\sqrt{RS}}\right)$$

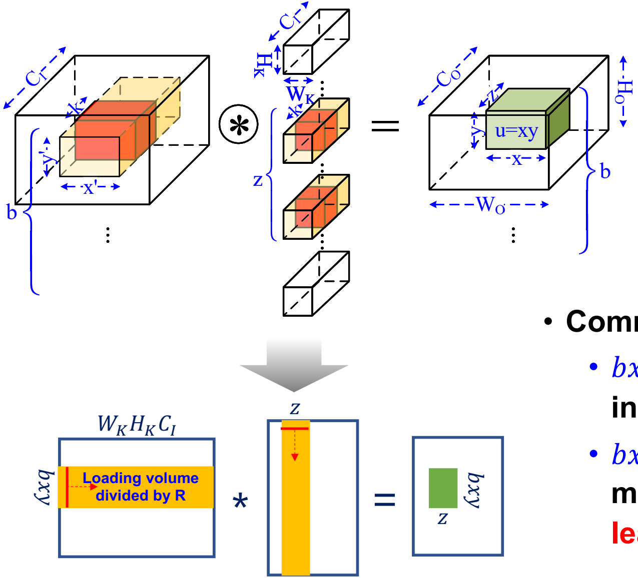
$$R = \frac{W_K H_K}{D_W D_H}$$

W_K & H_K: kernel size
D_W & D_H: stride size

R: max reuse number of each input by sliding window reuse

[1] J.-W. Hong and H. T. Kung, "I/O Complexity: The Red-blue Pebble Game," in ACM Symposium on Theory of Computing (STOC), 1981, pp. 326-333

Communication-optimal Dataflow



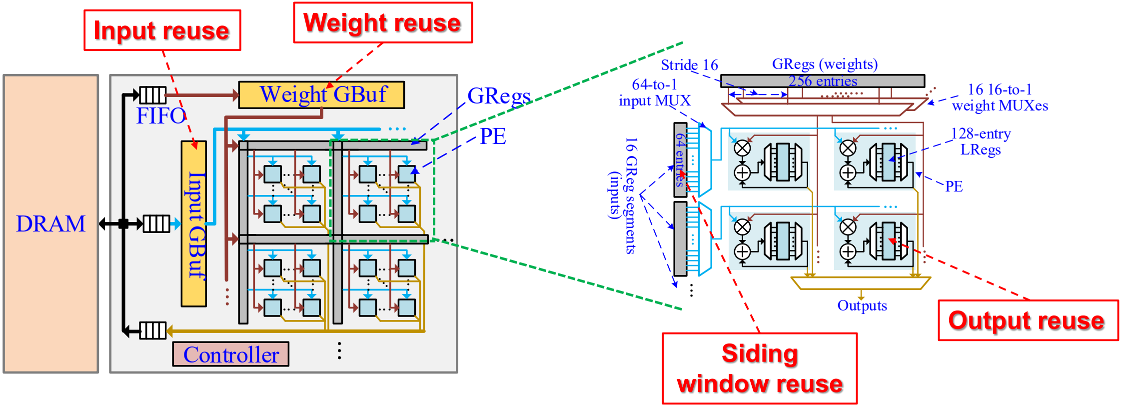
Tiling parameters

$$< b, x, y, z, k >$$

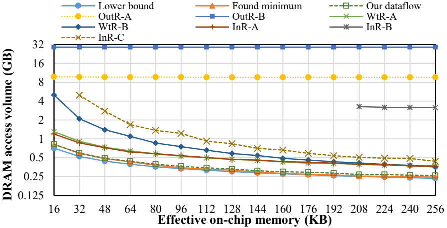
- **Communication-optimal tiling parameters**
  - $bxy \approx Rz$: **balanced loading volumes of inputs & weights**
  - $bxyz \approx S$ **&** $k = 1$: **most of on-chip memory should be for Psums (using least inputs to produce most outputs)**

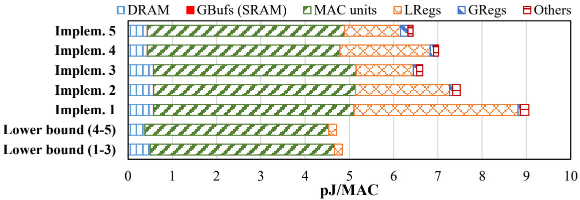
Communication-optimal Architecture

- **Straightforward implementation of communication-optimal dataflow**
- **Elaborate multiplexer structure to adapt to different tiling parameters, no inter-PE data propagation**



Simulation Results





DRAM access: 4.5% more than lower bound, >40% reduction than Eyeriss [1]

Energy consumption: 37-87% higher than lower bound

[1] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in ISCA 2016