CMSC 5743

Efficient Computing of Deep Neural Networks

Implementation 03: Winograd

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

(Latest update: September 2, 2024)

2024 Fall

Overview

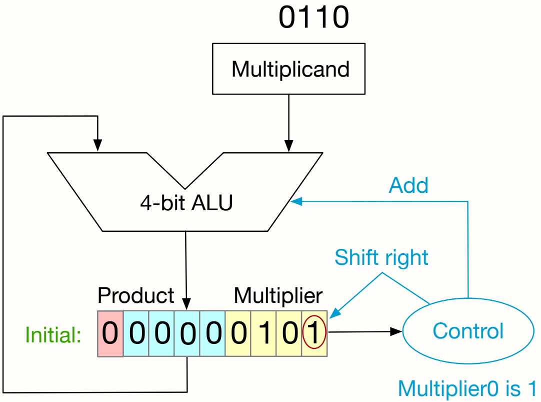
Overview

Introduction

Motivation of This Lecture

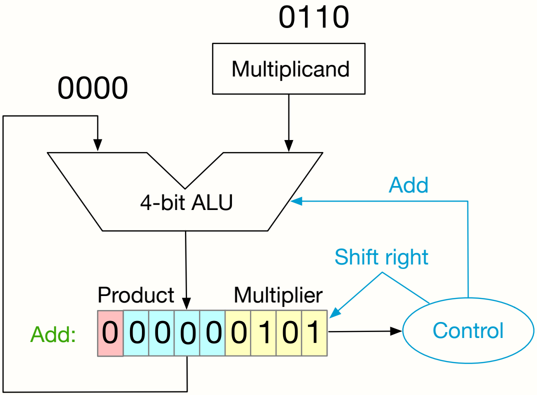
- Reduce multiplication #
- Don't care about addition #

Multiplication Procedure:



- Reduce multiplication #
- Don't care about addition #

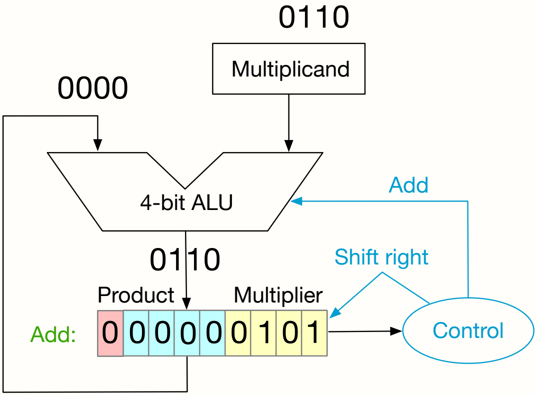
Multiplication Procedure:



Motivation of This Lecture

- Reduce multiplication #
- Don't care about addition #

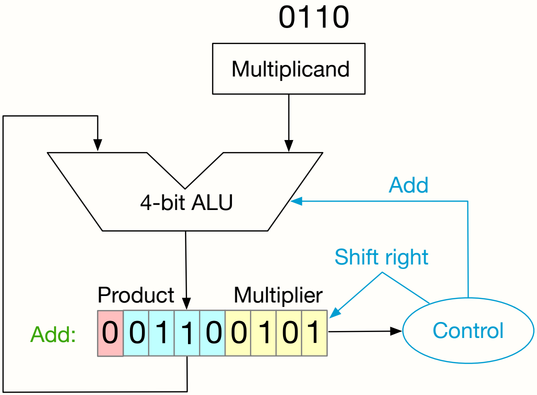
Multiplication Procedure:



Motivation of This Lecture

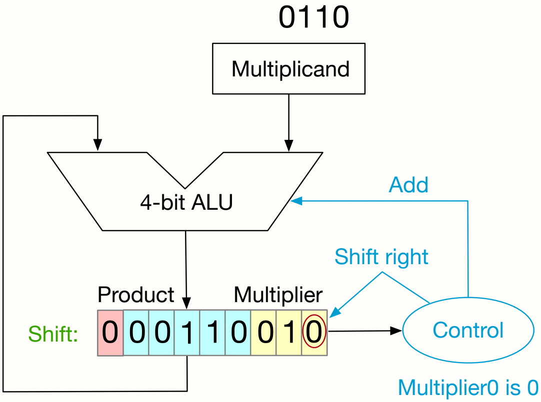
- Reduce multiplication #
- Don't care about addition #

Multiplication Procedure:



- Reduce multiplication #
- Don't care about addition #

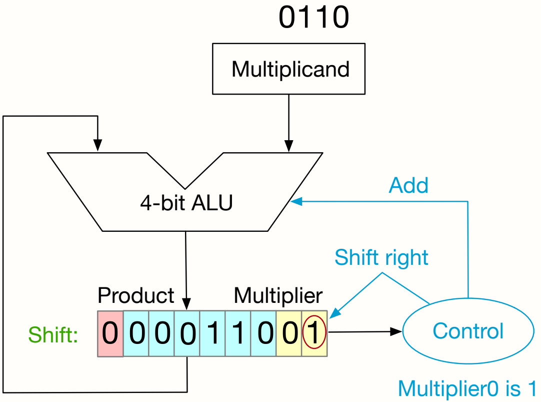
Multiplication Procedure:



Motivation of This Lecture

- Reduce multiplication #
- Don't care about addition #

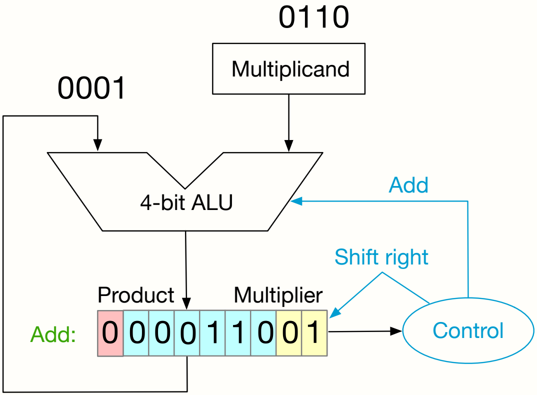
Multiplication Procedure:



Motivation of This Lecture

- Reduce multiplication #
- Don't care about addition #

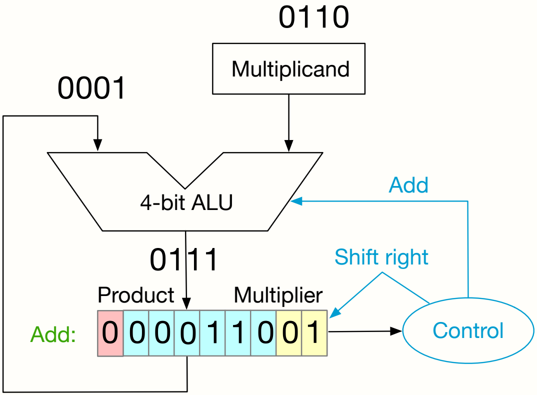
Multiplication Procedure:



Motivation of This Lecture

- Reduce multiplication #
- Don't care about addition #

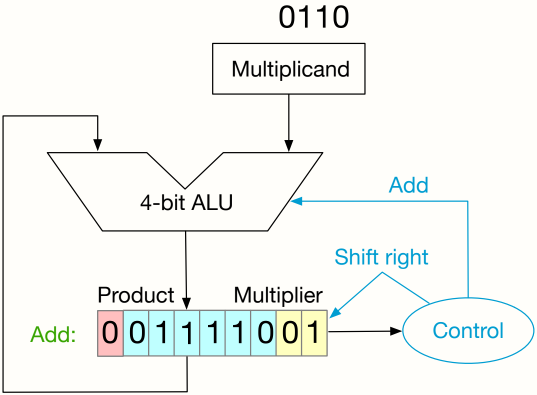
Multiplication Procedure:



Motivation of This Lecture

- Reduce multiplication #
- Don't care about addition #

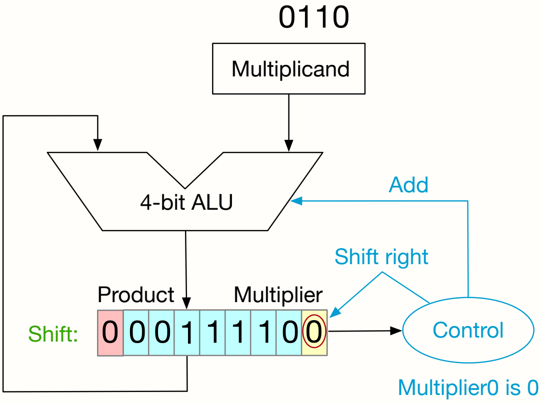
Multiplication Procedure:



Motivation of This Lecture

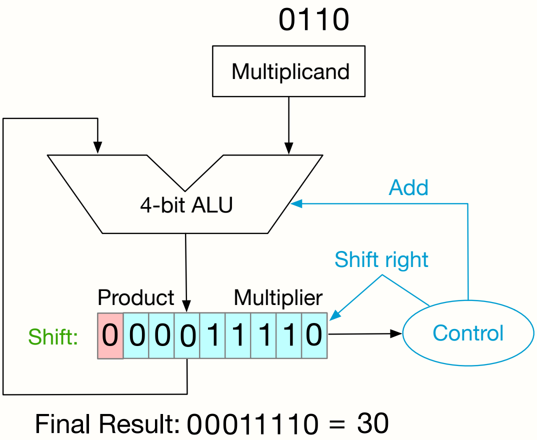
- Reduce multiplication #
- Don't care about addition #

Multiplication Procedure:



- Reduce multiplication #
- Don't care about addition #

Multiplication Procedure:



Overview

1. Introduction
2. Strassen
3. Winograd

Strassen

Matrix Multiplication: Naive Algorithm

Naive Matrix Multiplication

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

1: **for all** $i \in 1, \cdots, N$ **do**
2:     **for all** $j \in 1, \cdots, N$ **do**
3:         $C_{ij} = \sum_{t=1}^{N} A_{it} \cdot B_{tj}$;
4:     **end for**
5: **end for**
6: **return** C;

- Time Complexity: $\mathcal{O}(N^3)$

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

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} \boldsymbol{B}_{11} & \boldsymbol{B}_{12} \\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22} \end{bmatrix}, \quad \boldsymbol{C} = \begin{bmatrix} \boldsymbol{C}_{11} & \boldsymbol{C}_{12} \\ \boldsymbol{C}_{21} & \boldsymbol{C}_{22} \end{bmatrix},$$

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

$$\begin{bmatrix} \boldsymbol{C}_{11} & \boldsymbol{C}_{12} \\ \boldsymbol{C}_{21} & \boldsymbol{C}_{22} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_{11}\boldsymbol{B}_{11} + \boldsymbol{A}_{12}\boldsymbol{B}_{21} & \boldsymbol{A}_{11}\boldsymbol{B}_{12} + \boldsymbol{A}_{12}\boldsymbol{B}_{22} \\ \boldsymbol{A}_{21}\boldsymbol{B}_{11} + \boldsymbol{A}_{22}\boldsymbol{B}_{21} & \boldsymbol{A}_{21}\boldsymbol{B}_{12} + \boldsymbol{A}_{22}\boldsymbol{B}_{22} \end{bmatrix}$$

Matrix Multiplication: Recursive Algorithm

Recursive Matrix Multiplication

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

function $\mathtt{M}(A, B)$
  if A is 1×1 **then**
    return $A_{11} \cdot B_{11}$;
  end if
  for all $i \in \{1, 2\}$ **do**
    for all $j \in \{1, 2\}$ **do**
      $C_{ij} = \mathtt{M}(A_{i1}, B_{1j}) + \mathtt{M}(A_{i2}, B_{2j})$;
    end for
  end for
  return $\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$;
end function

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:

$$T(n) = \Theta(N^{\log_2^8}) = \Theta(N^3).$$

Background: Asymptotic Order

- $f = \mathcal{O}(g)$: f grows no faster than g
- $f = \Theta(g)$: f grows at the same rate as g
- $f = \Omega(g)$: f grows at least as fast as g
- Note: $\Theta(g) = \mathcal{O}(g) \wedge \Omega(g)$

Background: Asymptotic Order

- $f = \mathcal{O}(g)$: f grows no faster than g
- $f = \Theta(g)$: f grows at the same rate as g
- $f = \Omega(g)$: f grows at least as fast as g
- Note: $\Theta(g) = \mathcal{O}(g) \wedge \Omega(g)$

Limit definitions:

- $f = \mathcal{O}(g)$ if $\lim_{n \to +\infty} \frac{f(n)}{g(n)} < \infty$, including 0
- $f = \Omega(g)$ if $\lim_{n \to +\infty} \frac{f(n)}{g(n)} > 0$, including ∞
- $f = \Theta(g)$ if $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = c, c \in (0, \infty)$
- Note: $\lim_{n \to +\infty}$ means for all $n \geq N$ for some constant N

Master Theorem

Let $T(n)$ be a monotonically increasing function that satisfies

$$T(n) = \begin{cases} c, & \text{if } N = 1; \\ aT(\frac{n}{b}) + f(n), & \text{if } N > 1. \end{cases}$$

where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta(n^d)$ when $d \geq 0$, then:

$$T(n) = \begin{cases} \Theta(n^d), & \text{if } a < b^d \\ \Theta(n^d logn), & \text{if } a = b^d \\ \Theta(n^{log_b a}), & \text{if } a > b^d \end{cases}$$

http://cse.unl.edu/~choueiry/S06-235/files/MasterTheorem-HandoutNoNotes.pdf

Master Theorem

Let $T(n)$ be a monotonically increasing function that satisfies

$$T(n) = \begin{cases} c, & \text{if } N = 1; \\ aT(\frac{n}{b}) + f(n), & \text{if } N > 1. \end{cases}$$

where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta(n^d)$ when $d \geq 0$, then:

$$T(n) = \begin{cases} \Theta(n^d), & \text{if } a < b^d \\ \Theta(n^d logn), & \text{if } a = b^d \\ \Theta(n^{log_b a}), & \text{if } a > b^d \end{cases}$$

In Strassen's case, $a = 8, b = 2, d = 2 \rightarrow a > b^d$.

http://cse.unl.edu/~choueiry/S06-235/files/MasterTheorem-HandoutNoNotes.pdf

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:

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix}, \quad \boldsymbol{N} = \begin{bmatrix} \boldsymbol{E} & \boldsymbol{F} \\ \boldsymbol{G} & \boldsymbol{H} \end{bmatrix}$$

Then, we calculate the intermediate matrices:

$$\begin{aligned}
\boldsymbol{S}_1 &= (\boldsymbol{B} - \boldsymbol{D})(\boldsymbol{G} + \boldsymbol{H}) \\
\boldsymbol{S}_2 &= (\boldsymbol{A} + \boldsymbol{D})(\boldsymbol{E} + \boldsymbol{H}) \\
\boldsymbol{S}_3 &= (\boldsymbol{A} - \boldsymbol{C})(\boldsymbol{E} + \boldsymbol{F}) \\
\boldsymbol{S}_4 &= (\boldsymbol{A} + \boldsymbol{B})\boldsymbol{H} \\
\boldsymbol{S}_5 &= \boldsymbol{A}(\boldsymbol{F} - \boldsymbol{H}) \\
\boldsymbol{S}_6 &= \boldsymbol{D}(\boldsymbol{G} - \boldsymbol{E}) \\
\boldsymbol{S}_7 &= (\boldsymbol{C} + \boldsymbol{D})\boldsymbol{E}.
\end{aligned}$$

Strassen Algorithm[1]

The final Strassen algorithm results are:

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{E} & \boldsymbol{F} \\ \boldsymbol{G} & \boldsymbol{H} \end{bmatrix} = \begin{bmatrix} \boldsymbol{S}_1 + \boldsymbol{S}_2 - \boldsymbol{S}_4 + \boldsymbol{S}_6 & \boldsymbol{S}_4 + \boldsymbol{S}_5 \\ \boldsymbol{S}_6 + \boldsymbol{S}_7 & \boldsymbol{S}_2 - \boldsymbol{S}_3 + \boldsymbol{S}_5 - \boldsymbol{S}_7 \end{bmatrix}.$$

[1] Jason Cong and Bingjun Xiao (2014). "Minimizing computation in convolutional neural networks". In: *Proc. ICANN*, pp. 281–290.

Strassen Algorithm

Algorithm Strassen's Algorithm

function STRASSEN$(\boldsymbol{M}, \boldsymbol{N})$
  if $\boldsymbol{M}$ is 1×1 **then**
    return $M_{11}N_{11}$;
  end if
  Let $\boldsymbol{M} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix}$ and $\boldsymbol{N} = \begin{bmatrix} \boldsymbol{E} & \boldsymbol{F} \\ \boldsymbol{G} & \boldsymbol{H} \end{bmatrix}$;
  Set $\boldsymbol{S}_1 =$ STRASSEN$(\boldsymbol{B} - \boldsymbol{D}, \boldsymbol{G} + \boldsymbol{H})$;
  Set $\boldsymbol{S}_2 =$ STRASSEN$(\boldsymbol{A} + \boldsymbol{D}, \boldsymbol{E} + \boldsymbol{H})$;
  Set $\boldsymbol{S}_3 =$ STRASSEN$(\boldsymbol{A} - \boldsymbol{C}, \boldsymbol{E} + \boldsymbol{F})$;
  Set $\boldsymbol{S}_4 =$ STRASSEN$(\boldsymbol{A} + \boldsymbol{B}, \boldsymbol{H})$;
  Set $\boldsymbol{S}_5 =$ STRASSEN$(\boldsymbol{A}, \boldsymbol{F} - \boldsymbol{H})$;
  Set $\boldsymbol{S}_6 =$ STRASSEN$(\boldsymbol{D}, \boldsymbol{G} - \boldsymbol{E})$;
  Set $\boldsymbol{S}_7 =$ STRASSEN$(\boldsymbol{C} + \boldsymbol{D}, \boldsymbol{E})$;
  return $\begin{bmatrix} \boldsymbol{S}_1 + \boldsymbol{S}_2 - \boldsymbol{S}_4 + \boldsymbol{S}_6 & \boldsymbol{S}_4 + \boldsymbol{S}_5 \\ \boldsymbol{S}_6 + \boldsymbol{S}_7 & \boldsymbol{S}_2 - \boldsymbol{S}_3 + \boldsymbol{S}_5 - \boldsymbol{S}_7 \end{bmatrix}$;
end function

Time Complexity Analysis

- Strassen algorthm 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:

$$T(n) = \Theta(N^{\log_2^7}) = \Theta(N^{2.8074}).$$

Strassen Algorithm in MNN[2]

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;
}
```

[2] Xiaotang Jiang et al. (2020). "MNN: A Universal and Efficient Inference Engine". In: *Proc. MLSys*.

Overview

Winograd

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

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

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

Winograd Algorithm[3]

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

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

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

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

Winograd Algorithm[3]

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:

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

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

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

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

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

[3]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:

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

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

Winograd Algorithm[5]

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

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

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

Winograd Algorithm[6]

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:

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

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

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

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

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

Winograd Algorithm[8]

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:

$$Y = A^T \left[[GgG^T] \odot [B^T dB] \right] A \qquad (8)$$

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

[8] 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:

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

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.

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

Winograd Algorithm[8]

4.3. F(4x4,3x3)

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

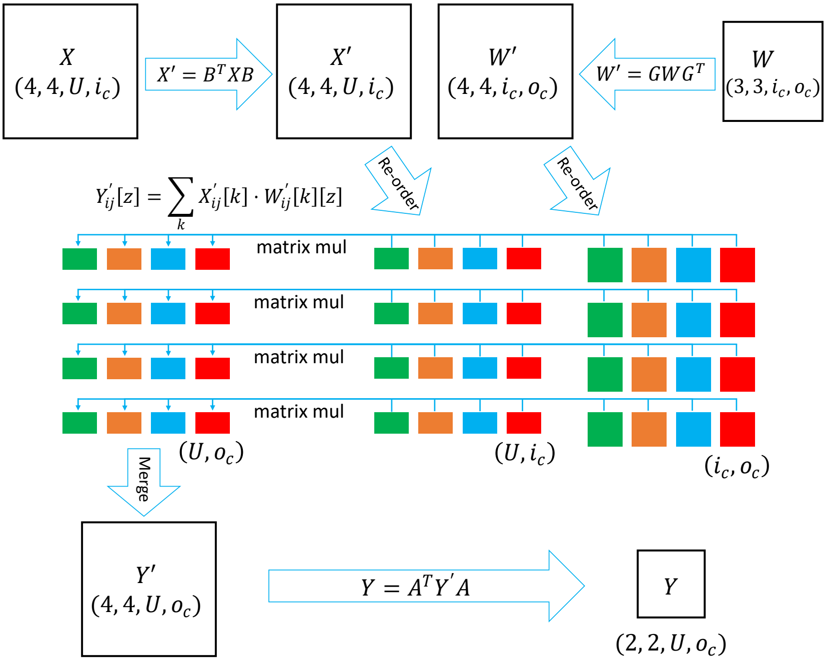
$$
\begin{aligned}
B^T &= \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix} \\
G &= \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6}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

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.

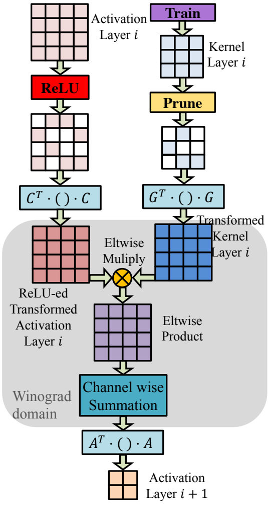
[8] Andrew Lavin and Scott Gray (2016). “Fast Algorithms for Convolutional Neural Networks”. In: *Proc. CVPR*, pp. 4013–4021.

Optimized Winograd algorithm in MNN



Sparse Winograd

Training in the Winograd Domain



(a)

Producing 4 output pixels:

Direct Convolution:

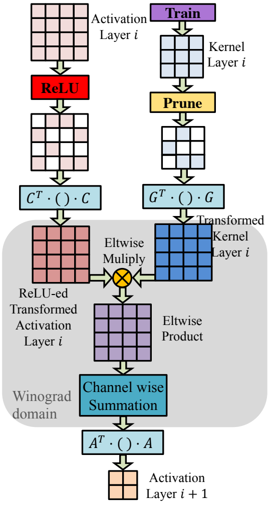
- 4*9=36 multiplications (**1x**)

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:

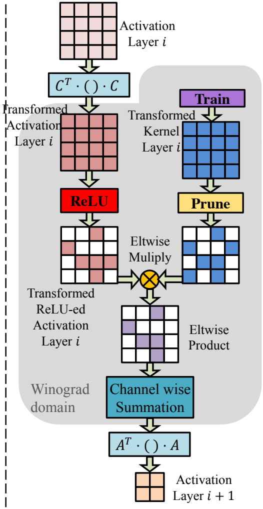
- 4*9=36 multiplications (**1x**)
- sparse weight [NIPS'15] (**3x**)
- sparse activation (relu) (**3x**)
- Overall saving: **9x**

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:

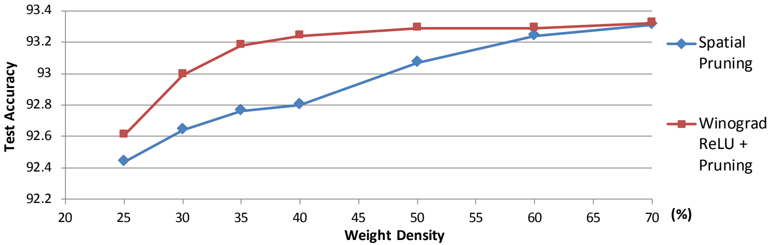
- 4*9=36 multiplications (**1x**)
- sparse weight [NIPS'15] (**3x**)
- sparse activation (relu) (**3x**)
- Overall saving: **9x**

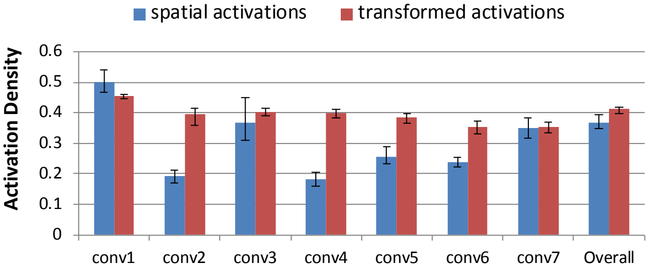
Winograd convolution:

- 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