CENG 5030
Energy Efficient Computing

Implementation 03: Winograd

Bei Yu
CSE Department, CUHK
byu@cse.cuhk.edu.hk
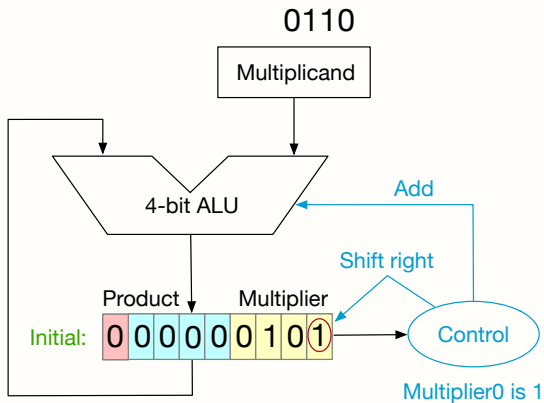
(Latest update: October 16, 2023)

2023 Fall

Introduction

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

Multiplication Procedure:

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

Multiplication Procedure:

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

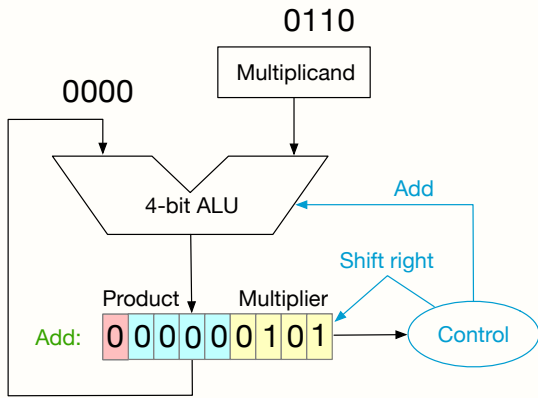## Multiplication Procedure:

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

Multiplication Procedure:

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

Multiplication Procedure:

- Reduce multiplication #
- Don't care about addition #
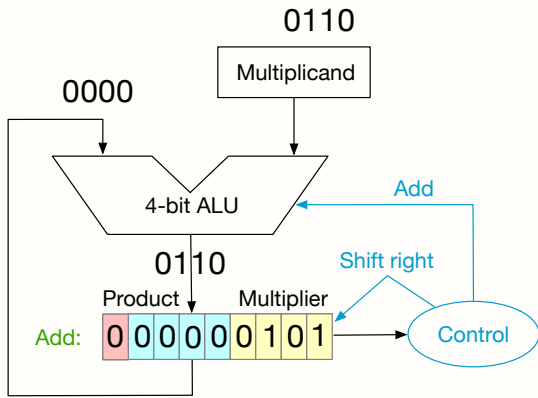
Multiplication Procedure:

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

Multiplication Procedure:

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

Multiplication Procedure:

- Reduce multiplication #
- Don't care about addition #
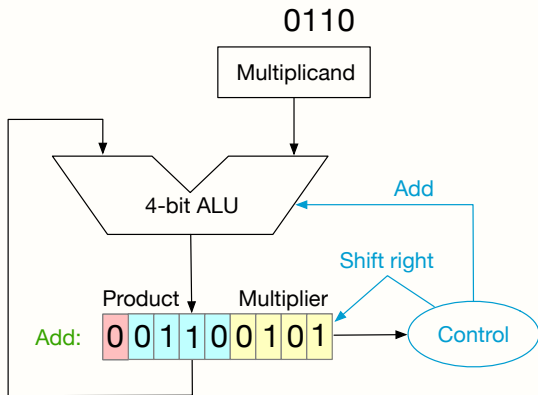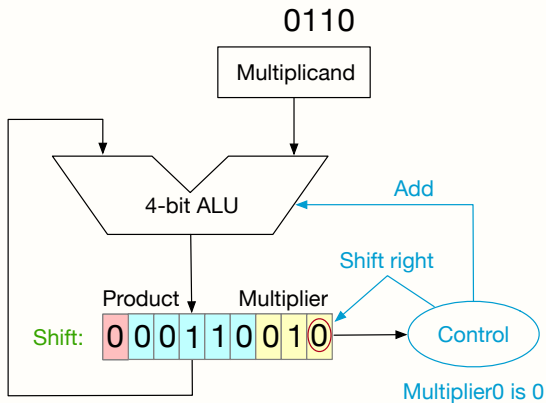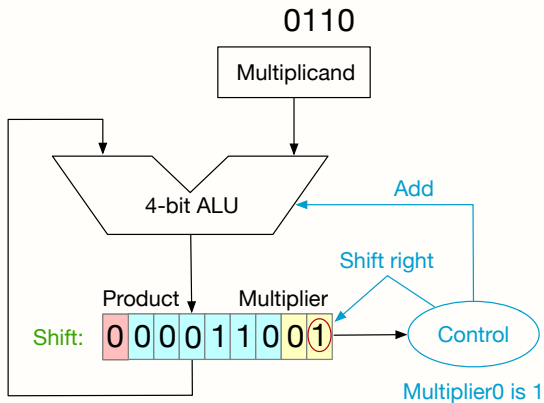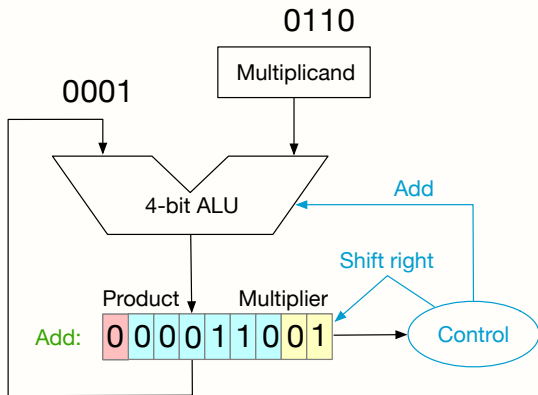
Multiplication Procedure:

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

Multiplication Procedure:

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

Multiplication Procedure:

0110

Multiplicand

4-bit ALU

Add

Shift right

Product   Multiplier

Shift: 0 0 0 0 1 1 1 0

Control

Final Result: 00011110 = 30

Overview

Strassen

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

To compute $C = AB$, we first partition A, B and C into equal-sized blocked matrices such that

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

where $A_{ij}, B_{ij}, C_{ij} \in \mathbb{R}^{\frac{N}{2} \times \frac{N}{2}}$. We then have:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

Recursive Matrix Multiplication

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

```
 1: function M(A, B)
 2:     if A is 1 × 1 then
 3:         return A₁₁ · B₁₁;
 4:     end if
 5:     for all i ∈ {1, 2} do
 6:         for all j ∈ {1, 2} do
 7:             C_ij = M(A_i1, B_1j) + M(A_i2, B_2j);
 8:         end for
 9:     end for
10:     return [ C₁₁  C₁₂ ]
            [ C₂₁  C₂₂ ];
11: end function
```

The recursive algorithm can be formulated as:

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

- 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:

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

- $f = \mathcal{O}(g)$: 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: $\Theta(g) = \mathcal{O}(g) \wedge \Omega(g)$

- $f = \mathcal{O}(g)$: 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 $\lim\limits_{n \to +\infty} \dfrac{f(n)}{g(n)} < \infty$, including 0

- $f = \Omega(g)$ if $\lim\limits_{n \to +\infty} \dfrac{f(n)}{g(n)} > 0$, including ∞

- $f = \Theta(g)$ if $\lim\limits_{n \to +\infty} \dfrac{f(n)}{g(n)} = c, c \in (0, \infty)$

- Note: $\lim\limits_{n \to +\infty}$ 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:

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

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:

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

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

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

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

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 Strassen algorithm results are:

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

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

Algorithm Strassen's Algorithm

1: **function** STRASSEN(M, N)
2:     **if** M is 1×1 **then**
3:         **return** $M_{11}N_{11}$;
4:     **end if**
5:     Let $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ and $N = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$;
6:     Set S_1= STRASSEN($B - D$, $G + H$);
7:     Set S_2= STRASSEN($A + D$, $E + H$);
8:     Set S_3= STRASSEN($A - C$, $E + F$);
9:     Set S_4= STRASSEN($A + B$, H);
10:    Set S_5= STRASSEN(A, $F - H$);
11:    Set S_6= STRASSEN(D, $G - E$);
12:    Set S_7= STRASSEN($C + D$, E);
13:    **return** $\begin{bmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{bmatrix}$;
14: **end function**

- 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:

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

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** ($\downarrow 13.5\%$)

```cpp
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

$$\mu(F(m, r)) = m + r - 1 \qquad (3)$$

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

$$\mu(F(m \times n, r \times s)) = \mu(F(m, r))\mu(F(n, s))$$
$$= (m + r - 1)(n + s - 1) \qquad (4)$$

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

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.

Fast filtering algorithms can be written in matrix form as:

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

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

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

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

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

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

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

[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.

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.

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:

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

where

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

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

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.

4.3. F(4x4,3x3)

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

$$B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix}$$

$$G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix} \tag{15}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix}$$

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.

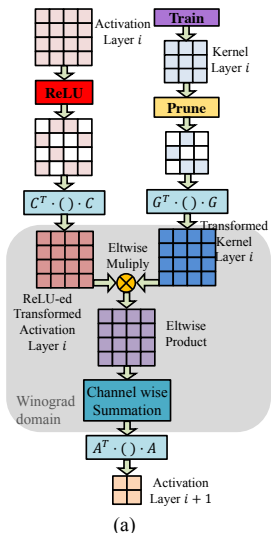$$Y'_{ij}[z] = \sum_{k} X'_{ij}[k] \cdot W'_{ij}[k][z]$$
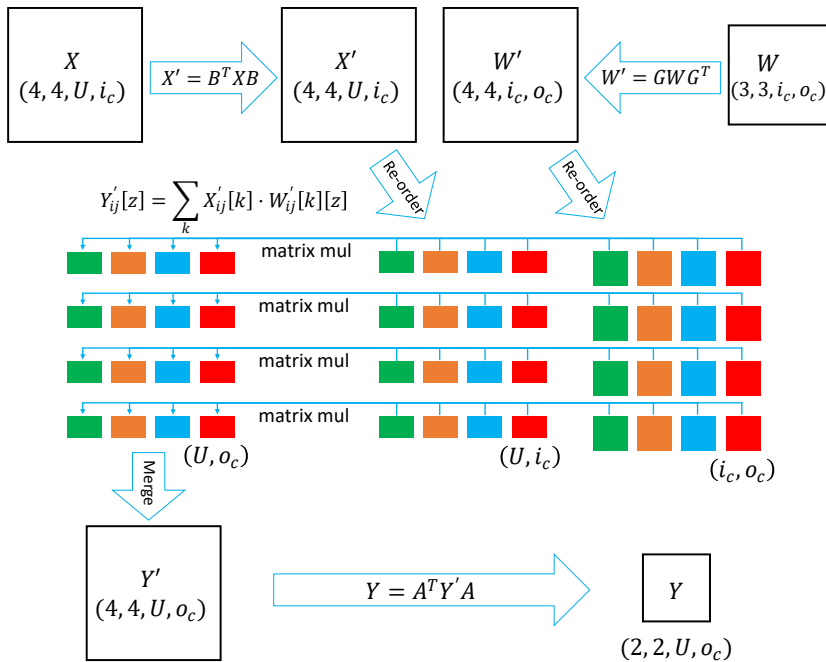
Sparse Winograd

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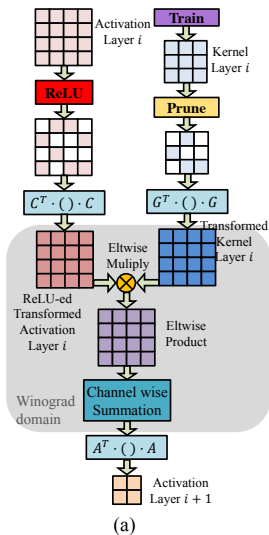
Liu et al. "Efficient Sparse-Winograd Convolutional Neural Networks", submitted to ICLR 2017 workshop

Training in the Winograd Domain
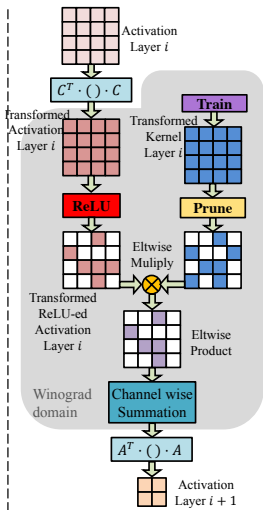

(a)

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**

Result

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

26/26