L11: Systolic Array and Winograd Algorithm

Cong (Callie) Hao
callie.hao@ece.gatech.edu

Assistant Professor
ECE, Georgia Institute of Technology

Sharc-lab @ Georgia Tech https://sharclab.ece.gatech.edu/
Outline

- Why Systolic Array
- Systolic Array for Matrix Multiplication
- Systolic Array for Convolution
- Winograd Algorithm
Our Approach for GEMM So Far

\[
\begin{array}{cccc}
\end{array}
\]

\[
\begin{array}{cccc}
B[0][3] & B[0][2] & B[0][1] & B[0][0] \\
\end{array}
\]

PE *+ PE *+ PE *+ PE *+
PE *+ PE *+ PE *+ PE *+
PE *+ PE *+ PE *+ PE *+
PE *+ PE *+ PE *+ PE *+
Our Approach for GEMM So Far


B[0][3] B[0][2] B[0][1] B[0][0]
Our Approach for GEMM So Far

\[
\begin{align*}
\end{align*}
\]
Our Approach for GEMM So Far

Accumulate at each PE

Our Approach for GEMM So Far

Accumulate at each PE


B[0][3]  B[0][2]  B[0][1]  B[0][0]
Our Approach for GEMM So Far

Problem?
Our Approach for GEMM So Far

Problem?

- High routing complexity
- High fan-out
- Lower clock frequency
- High wire energy

This is one essential limit factor for high parallelism
**Solution: Reuse Data from Neighbor PE**

|---------|---------|---------|---------|

Get $A[0][0]$ from PE next to it instead of from Memory?

<table>
<thead>
<tr>
<th>B[0][3]</th>
<th>B[0][2]</th>
<th>B[0][1]</th>
<th>B[0][0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE *+</td>
<td>PE *+</td>
<td>PE *+</td>
<td>PE *+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PE *+</td>
<td>PE *+</td>
<td>PE *+</td>
<td>PE *+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PE *+</td>
<td>PE *+</td>
<td>PE *+</td>
<td>PE *+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PE *+</td>
<td>PE *+</td>
<td>PE *+</td>
<td>PE *+</td>
</tr>
</tbody>
</table>
Solution: Reuse Data from Neighbor PE

Cycle 1


B[0][3]  B[0][2]  B[0][1]  B[0][0]
Solution: Reuse Data from Neighbor PE

Cycle 2
Solution: Reuse Data from Neighbor PE

Cycle 3


B[0][3] B[0][2] PE B[0][1] PE B[0][0] PE

B[1][0] A[0][0]

PE

PE

B[3][0] PE

A[0][2] A[0][3]
Solution: Reuse Data from Neighbor PE

Cycle 4
Solution: Reuse Data from Neighbor PE

Cycle 5
Solution: Reuse Data from Neighbor PE

Accumulation completed, write back (or, read out)

Cycle 5
Question: how do we do read-out to avoid complex routing?

Solution: Reuse Data from Neighbor PE

Cycle 5

Accumulation completed, write back (or, read out)
Solution: Also shift out the results

Solution: Reuse Data from Neighbor PE

Some logic to read the correct results
A Dynamic View of Systolic Array

Input 1

Input 2

PE Array
A Dynamic View of Systolic Array
A Dynamic View of Systolic Array
A Dynamic View of Systolic Array
A Dynamic View of Systolic Array
A Dynamic View of Systolic Array
A Dynamic View of Systolic Array
A Dynamic View of Systolic Array

Reached its peak performance
Some Variants

\[ \text{Callie Hao | Sharc-lab @ Georgia Institute of Technology} \]

\[ \text{https://sharclab.ece.gatech.edu/} \]
Some Variants

Try a simple 3x3 matrix multiplication to figure it out!

Reference:
https://www.youtube.com/watch?v=vADVh1ogNoo
Systolic Array For Convolution

• Reference:

L1: for(o = 0; o < O; o++) // Output feature #
L2: for(i = 0; i < I; i++) // Input feature #
L3: for(c = 0; c < C; c++) // Feature column
L4: for(r = 0; r < R; r++) // Feature row
L5: for(p = 0; p < K; p++) // Kernel weight
L6: for(q = 0; q < K; q++) // Kernel height
OUT[o][r][c] += W[o][i][p][K] * IN[i][r+p][c+q];
Systolic Array For Convolution

Input:

\[ \rightarrow PE_{0,0} \rightarrow PE_{0,1} \rightarrow PE_{0,2} \]

\[ \rightarrow PE_{1,0} \rightarrow PE_{1,1} \rightarrow PE_{1,2} \]

\[ \rightarrow PE_{2,0} \rightarrow PE_{2,1} \rightarrow PE_{2,2} \]

Output:

\[ \text{previous or current PE} \]

\[ \text{Input} \]

\[ \text{weight} \]

\[ OUT_{xy} \]

\[ \text{Input to next PE} \]

\[ \text{Output from previous PE} \]
Google’s Tensor Process Unit (TPU)

Reference: “In-Datacenter Performance Analysis of a Tensor Processing Unit”, ISCA 2017

Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.

Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.
Google’s Tensor Process Unit (TPU)

Reference: “In-Datacenter Performance Analysis of a Tensor Processing Unit”, ISCA 2017

- One Unit has a systolic array that contains $256 \times 256 = \text{total } 65,536 \text{ ALUs}$.
- Can process 65,536 multiply-and-adds for 8-bit integers every cycle.
- Given clock frequency 700MHz, a TPU can compute $65,536 \times 700,000,000 = 46 \times 10^{12}$ multiply-and-add operations or 92 Teraops per second ($92 \times 10^{12}$).

Data Reuse in Systolic Array

- Input / Output / Weight Stationary – which data stays in the memory?

Output Stationary
Weight Stationary

Unrolled convolution windows

Pre-fill weights
Input Stationary

Unrolled Weight Matrices

Input Stationary

Pre fill inputs
Systolic Array Pros and Cons

• Pros
  o Makes multiple uses of each data item -> reduced need for fetching/refetching -> **better use of memory bandwidth**
  o High **concurrency** -> high **performance**
  o **Regular design** (both data and control flow)
  o Simple, low circuit complexity, high circuit efficiency

• Cons
  o Not good at exploiting **irregular parallelism**
  o **Highly specialized**: not many algorithms can be expressed as matrix multiplications -> complicated software support
Resources for Systolic Array

• Recommended Paper:

• Systolic Array Implementation:
  o https://github.com/PSCLab-ASU/Systolic-CNN
  o https://github.com/spcl/gemm_hls

• Read more about TPU:
Winograd Algorithm

- Winograd Algorithm has been used to speed up FIR filter computation since at least 1980 (Shmuel Winograd. Arithmetic complexity of computations, volume 33. Siam, 1980. 1, 2, 3, 4)
  - Reduce the number of multiplications, which is more expensive than additions
  - Some computations can be pre-computed in advance

An r-tap FIR filter $F(m, r)$; $m=2$, $r=3$ in this example

Inputs $d_0, d_1, d_2, d_3$

Param. $g_0, g_1, g_2$

Outputs $y_0, y_1$

$y_0 = d_0 \cdot g_0 + d_1 \cdot g_1 + d_2 \cdot g_2$

$y_1 = d_1 \cdot g_0 + d_2 \cdot g_1 + d_3 \cdot g_2$

Total $2 \times 3 = 6$ multiplications
Winograd Algorithm

An r-tap FIR filter $F(m, r)$; $m=2$, $r=3$ in this example

\[ y_0 = d_0 \cdot g_0 + d_1 \cdot g_1 + d_2 \cdot g_2 \]
\[ y_1 = d_1 \cdot g_0 + d_2 \cdot g_1 + d_3 \cdot g_2 \]

Total $2 \times 3 = 6$ multiplications

\[ F(2, 3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \]

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

An r-tap FIR filter $F(m, r); m=2, r=3$ in this example

\[
y_0 = d_0 \cdot g_0 + d_1 \cdot g_1 + d_2 \cdot g_2 \\
y_1 = d_1 \cdot g_0 + d_2 \cdot g_1 + d_3 \cdot g_2
\]

Total $2 \times 3 = 6$ multiplications

\[
F(2, 3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}
\]

- Total $2 + 3 - 1 = 4$ multiplications
- But with more additions

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

Can be pre-computed
Winograd Algorithm

An r-tap FIR filter $F(m, r)$; $m=2$, $r=3$ in this example

$$F(2, 3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}$$

$$g = \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix}^T$$
$$d = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix}^T$$

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

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

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$
Winograd Algorithm

An r-tap FIR filter \( F(m, r); m=2, r=3 \) in this example

\[
F(2, 3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}
\]

\[
Y = A^T [(Gg) \odot (B^T d)]
\]

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

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

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

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

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

Known in advance

Element-wise multiplication
Winograd Algorithm in Convolution

A minimal 1D algorithm $F(m, r)$ can be nested to obtain a minimal 2D algorithm, $F(m \times m, r \times r)$ as follows:

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

- $g$ is an $r \times r$ filter
- $d$ is an $(m + r - 1) \times (m + r - 1)$ image tile

An example of a 4x4 image tile with 3x3 kernel; output is 2x2
Winograd Algorithm in Convolution

\[ Y = A^T \left[ G g G^T \right] \odot \left[ B^T d B \right] A \]

Liu, Xinheng, et. al "WinoCNN: Kernel sharing Winograd systolic array for efficient convolutional neural network acceleration on FPGAs.", IEEE ASAP, 2021
Winograd Algorithm + Systolic Array

Liu, Xinheng, et. al "WinoCNN: Kernel sharing Winograd systolic array for efficient convolutional neural network acceleration on FPGAs." IEEE ASAP, 2021

PE Architecture

System Architecture
Winograd Algorithm for CNN References

• References:
Summary on Systolic Array

- Very **efficient**, simple design, few data movement, high throughput (at peak)

- Very **specific**: may not be applicable for many problems

- Can be fine-grained or course-grained
  - Fine-grained: matrix multiplication at value level; each PE is one multiply-accumulate
  - Course-grained: each PE is a big function (e.g., Winograd for one tile)
    - Each PE can have its own local buffer and control logic