Report CSMC5743 LAB2

LOU YI MING 1155215795

September 2024

1 Question 1

From this experiment, I changed the value of I,J,K to 256, 512, and 1024 just like the experiment implemented in the **LAB 1**. Then I implement the **Strassen** Algorithm to test the average running time for the matrices with different sizes. The result is quite obvious: **Strassen** algorithm lags far behind by direct matrix multiplication in performance, the results are shown in Table 1.

Table 1: Performance Compare

Algorithm	**I**	**J**	**K**	**Average_Time**
matmul	256	256	256	0.01
matmul	512	512	512	0.081
matmul	1024	1024	1024	0.866
matmul_ijk	256	256	256	0.0017
matmul_ijk	512	512	512	0.0094
matmul_ijk	1024	1024	1024	0.0777
matmul_AT	256	256	256	0.02
matmul_AT	512	512	512	0.155
matmul_AT	1024	1024	1024	1.40
matmul_BT	256	256	256	0.0017
matmul_BT	512	512	512	0.0096
matmul_BT	1024	1024	1024	0.076
Strassen	256	256	256	1.211
Strassen	512	512	512	8.539
Strassen	1024	1024	1024	60.384

1.1 Result Analysis

By comparing the results of my experiments, I analyzed the reason why the Strassen algorithm might be slower. Matrix multiplication needs a significant amount of data access and storage operations. The use of the cache system might be influenced by the recursive nature of the Strassen algorithm which may result in poor cache locality. As the matrix is recursively divided into sub-matrices, the data may no longer be efficiently utilized by the cache, leading to frequent cache misses and consequently increasing memory access overhead.

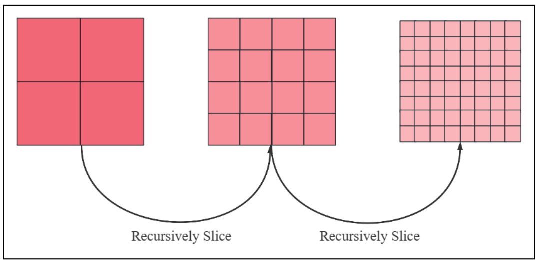

Figure 1: Strassen Process

2 Question 2

For my implementation, it can be noticed that **Winograd** runs faster on average than direct convolution and the result is shown in table 2. The parameters are height = 56; width = 56; channels = 3; out_channels = 64; kernel_size = 3; batch_size = 1; stride = 1; padding = 0;

Table 2: Performance Compare Winograd

Algorithm	**Average_Time**
Original Im2col	0.0091
Winograd	0.0024

2.1 Result Analysis

To analyze the reason why **Winograd** is faster, the theoretical explanation is that it reduces the number of multiplications and increases the number of additions (e.g. F(2,3) can reduce the times of multiplication from 6 to 4). As computers perform addition and shifting more faster than multiplication, so this algorithm can save a lot of time.

```
float D00 = im2col_data[b * out_height * out_width * 4 + (h * out_width + w) * 4]; // Compute D00 to D30
float D10 = im2col_data[b * out_height * out_width * 4 + ((h + 1) * out_width + w) * 4];
float D20 = im2col_data[b * out_height * out_width * 4 + (h * out_width + (w + 1)) * 4];
float D30 = im2col_data[b * out_height * out_width * 4 + ((h + 1) * out_width + (w + 1)) * 4];

float k0 = kernels[c * kernel_size * kernel_size + k]; // Compute K0 to K2
float k1 = kernels[c * kernel_size * kernel_size + k + 1];
float k2 = kernels[c * kernel_size * kernel_size + k + 2];

float M0 = (D00 - D20) * k0; // Compute M0 to M3
float M1 = (D10 + D20) * (k0 + k1 + k2) / 2.0f;
float M2 = (D20 - D10) * (k0 - k1 + k2) / 2.0f;
float M3 = (D10 - D30) * k2;

float r0 = M0 + M1 + M2; // Compute r0 and r1 as the result. followed by formula
float r1 = M1 - M2 - M3;
```

Just like the code shown above, we firstly compute **D00** to **D30**, the kernel **k0** to **k2** and then address the values for **M0** to **M3**. The result **r0** and **r1** can be computed by using **M0** to **M3**. In this whole computing process, we just need to compute **M1** and **M2** once but we can reuse them in both process to compute **r1** and **r2**.