어셈블리 프로그램 설계 및 실습

프로젝트 결과 보고서

Term project

학　　과: 컴퓨터공학과

담당교수: 이준환 교수님

학　　번: 2019202052

성　　명: 김호성

1. Introduction

- 구현 할 요소

  - Data set 에서 각각이 의미하는 정보를 나눠주는 함수 (Convolution_func)
  - Convolution_func 함수 이후 Kernel Data 를 저장하는 함수 (Store_Kernel_Data)
  - Store_Kernel_Data 함수 이후 Input Data 를 저장하는 함수 (Store_Matrix_Data)
  - Convolution 연산을 위해 순서에 맞게 곱하기를 진행 하는 함수 (Convolution)
  - Convolution 함수로 저장한 데이터들을 순서에 맞게 덧셈을 진행 하는 함수 (Calculation)
  - MUL instruction 대신 사용할 곱셈 (Radix2_of_Convolution_calculate)
  - Convolution 에서 나온 결과 값을 Max Pooling 하는 함수 (Max_Pooling)

- 일정

  - 11 월 26 일 ~ : 전반적인 코드의 흐름 구상
  - ~11 월 27 일 : Radix 2 Booth Algorithm 구현
  - ~ 11 월 28 일 : Convolution 연산에 필요한 일반화된 공식들 도출
  - ~ 11 월 29 일 Convolution 에 관한 함수들 구현
  - ~ 11 월 30 일 : Max pooling 함수 구현, 프로젝트 결과서 작성

2. Project Specification

- 각 요소 별 설명

- Convolution_func: Data_set 에 있는 초기 데이터를 각각 저장한다. 초기 정보로는 Convolution size, stride size, Input size, Kernel Data, Input Data 등이 있으나, Kernel Data 와 Input Data 의 경우 여러 개의 값을 가져야 하기 때문에 Store_Kernel_Data 와 Store_Matrix_Data 함수로 나누어 값을 저장한다. (r0 = stride, r1 = convolution size, r2 = input size)

- Store_Kernel_Data: 이중 반복문으로 설계했으며, 정방행렬의 특성을 이용하여, Convolution size 보다 작을 때까지 반복하도록 안쪽과 바깥쪽 반복문을 구현하였다. 궁극적으로 Convolution Data 값을 저장하는 역할을 한다. (r11)

- Store_Matrix_Data: 이중 반복문으로 설계했으며, 정방행렬의 특성을 이용하여, Input size 보다 작을 때까지 반복하도록 안쪽과 바깥쪽 반복문을 구현하였다. 궁극적으로 Input Data 를 저장하는 역할을 한다. (r12)

- Convolution: 4 중 반복문으로 설계 했으며, 각각의 레지스터들은 다음과 같은 값들을 의미한다.

r3 = 기준 점의 세로 위치

r4 = 기준 점의 가로 위치

r5 = 현재 위치의 가로 좌표 값

r6 = 현재 위치의 세로 좌표 값

가장 바깥쪽 반복문과 2 번째 바깥쪽 반복문의 경우, matrix size 가 r3(r4) + r1 – 1 보다 클 때까지 반복하도록 구현하였으며, r3(r4)값은 stride 값만큼 증가한다. 또, 가장 안쪽 반복문과 2 번째 안쪽 반복문의 경우, 각각 Convolution size 보다 r6(r5) 값이 작을 때까지 반복하도록 구현하였으며, Input data 의 좌표 값을 구하는 공식의 경우, r2(r3 + r5 + 1 – i) + r4 - r1 + r6 이라는 일반화된 식을 도출했다.

즉, stride condition = matrix size > width position + kernel size – 1 라는 식을 구한 후, 반복문 내부에서 변하는 값을 대입하여 r3(or r4) + r1 – 1 라는 식을 얻어냈고,

Input data position = Input size(stride height position + current height position) +stride width position +current width position 이라는 일반화 된 식을 구한 후, 반복문 내부에서 변화는 값을 대입하여 [r2(r3 + r5 + 1 – i) + r4 - r1 + r6] 라는 식을 얻어 냈다.

이러한 조건에 맞게 반복문을 돌리면서, 가장 안쪽 반복문이 될 때 마다, 값을 저장하는 알고리즘을 구현하였다. 즉 가장 안 쪽 반복문이 한 번 반복될 때 마다 Input data * kernel data 의 값을 저장해주며, 이후에 Calculation 함수에서 좌표에 맞게 다시 값을 대입해주어, Convolution 알고리즘을 마무리한다.

- Calculation: Calculation 함수는 Convolution 함수에서 만들어낸 Convolution_middle_result 값을 조건에 맞게 덧셈하면서 Convolution 함수의 result 값을 도출해낸다. 필요한 조건으론 다음과 같다.

1. 이중 반복문을 만들고, 안쪽 반복문은 kernel size 의 제곱만큼 반복하게끔, 바깥쪽 반복문은 input size = convolution size + (i-1)*stride 이 식을 성립하는 i 번 만큼 반복하게끔 한다.

2. 안쪽 반복문을 탈출한다는 것은 Convolution result 의 값 중 하나가 완성되었음을 의미한다. 즉, 바깥쪽 반복문으로 돌아가기 직전에 안쪽 반복문에서 값을 계속 더해주던 r4 레지스터 값을 Convolution result 에 저장하고 r4 를 초기화 한다.

3. 바깥쪽 반복문이 종료되었다는 것은, Convolution result 가 완성되었다는 의미이므로, Max Pooling 함수로 넘어간다.

- Radix2_of_Convolution_calculate: Radix-2 Booth 알고리즘으로 MUL instruction 을 진행하였다.

Booth Multiplication 이란 이진법으로 표현된 어느 두 변수의 곱셈에서 승수의 마지막 비트와 마지막에서 두 번째 비트를 비교하여 이에 따른 연산을 진행하고 그 연산의 결과에 따라 곱셈의 결과를 얻을 수 있는 곱셈방법을 말한다.

우선 이 방법을 사용하기 위해서는 4 개의 임의의 변수가 필요하다.

이 4 개의 변수를 u, v, x, x-1 이라고 했을 때, u, v, x-1 은 0 으로 초기화하고 x 는 승수가 된다 x 의 마지막 비트와 x-1 을 비교하여 연산을 진행하는데

{x, x-1} = {0,0} 일 경우 u 의 값을 ASR 연산을 하고 v 는 u 의 마지막비트를 최상위 비트로 가지며, LSR 연산을 진행한다.

{x, x-1} = {0,1} 일 경우 u 의 값을 피승수와 전의 u 의 값을 더한 후 ASR 연산을 한다. v 는 u 의 마지막 비트를 최상위 비트로 가지며, LSR 연산을 진행한다.

{x, x-1} = {1,0} 일 경우 u 의 값을 피승수와 전의 u 의 값과 뺀 후 ASR 연산을 한다. v 는 u 의 마지막비트를 최상위 비트로 가지며 LSR 연산을 진행한다.

{x, x-1} = {1,1} 일 경우 u 의 값을 ASR 연산을 하고 v는 u 의 마지막비트를 최상위 비트로 가지며 LSR 연산을 진행한다.

이러한 방법으로 연산이 진행되는데 이 때, 승수의 비트(x)가 32 번째이면 마지막으로 연산을 진행한 후 연산을 멈추고 그 때의 계산된 u와 v의 값을 이어 붙여 64비트짜리 결과 값을 얻는다.

일반적으로 한 비트씩 다른 32 비트짜리 데이터를 곱하고 이 연산을 32 번 반복하여 그 값들을 더해줄 수도 있지만 이러한 방법으로 진행 시 64 비트짜리 데이터가 필요하고 그 값을 하나하나 더해주는 방식이므로 사이클이 많이 사용될 것이다. 그러나 다행스럽게도 제안서에 나와있는 data 의 범위는 -2000 ~ 2000 사이이므로 이 데이터들은 음수일 때 2 의 보수를 취해서 값을 계산한다면, 16 비트를 넘지 않으므로 ARM 의 32bit 를 넘는 일이 발생하지 않는다.

- Max_Pooling: Convolution 과 다른 알고리즘이기 때문에 모든 레지스터를 초기화 한 후 값을 계산한다. Max Pooling 에서 각각의 레지스터는 다음과 같은 용도로 사용되었다.

r0 = result

r1 = Max Pooling size

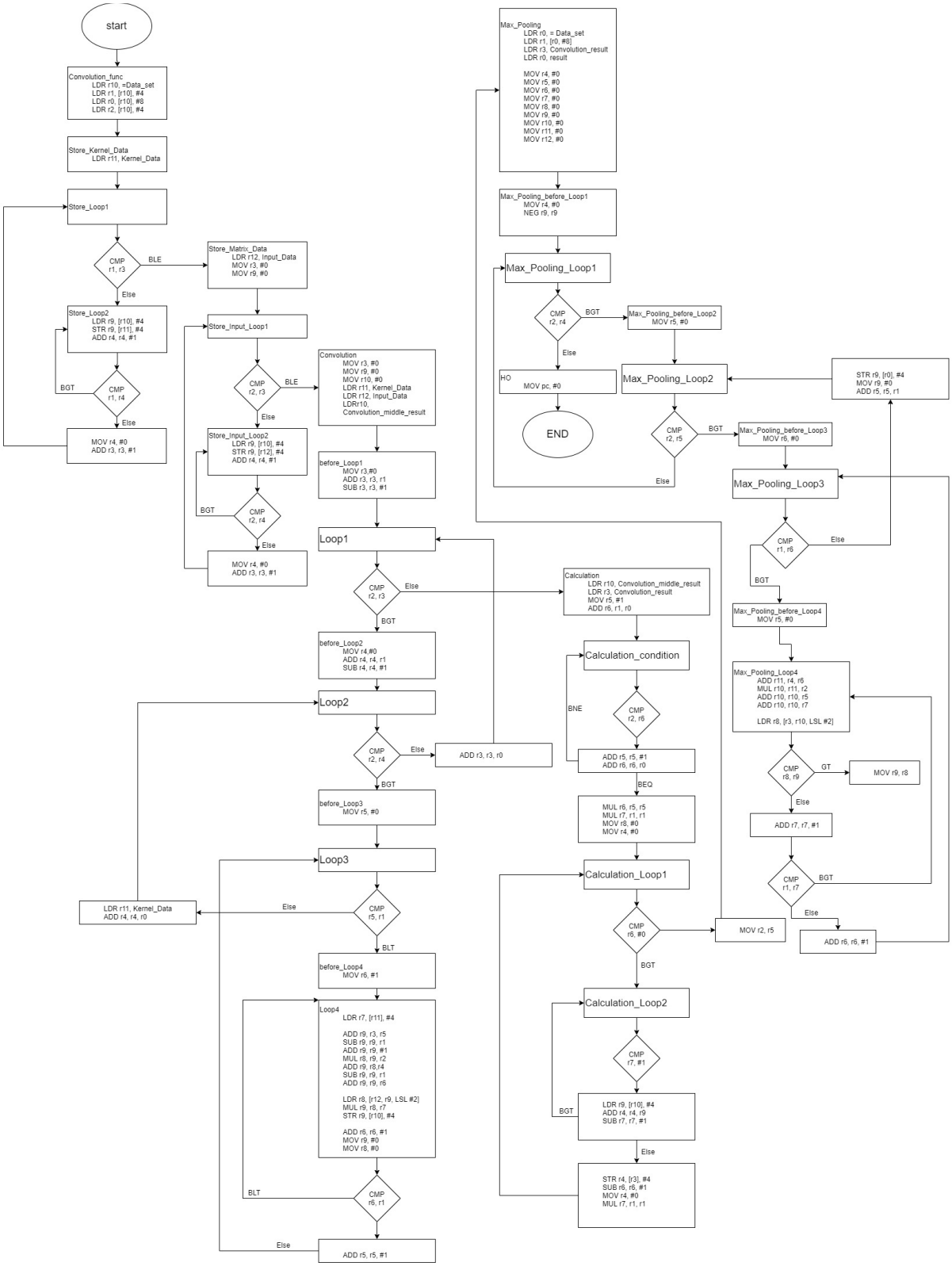
r2 = Convolution Output size

r3 = Convolution Output data

Max Pooling 도 Convolution 과 비슷한 구조로 구현하였으며, 다른 점이 있다면 평행이동 하는 값이 Max pooling size 라는 점이다. 또, Convolution Output Data 의 좌표에 맞게 움직이는 것 또한, Convolution 함수와 동일하다. (위에서 서술한 일반화된 식을 응용한 값이다.)

Max Pooling 의 경우 초기값을 0 에 NEG Instruction 을 적용한 값을 가지며(가장 작은 음수), 가장 안쪽 반복문에서 이전에 들어온 값보다 값이 크다면 새로 들어온 값을 레지스터가 가지고 있게끔 동작한다. 이후 2 번째 안쪽 반복문을 빠져나가기 직전 남아있는 값이 가장 Max Pooling size 안에 들어있는 data 중 가장 큰 값이므로, 그 값을 result 에 저장해준 후 2 번째 안쪽 반복문을 빠져나간다. 이러한 방법으로 반복문을 반복하여, 최종 결과 값을 얻는다.

3. Algorithm


start
Convolution_func
LDR r10, =Data_set
LDR r1, [r10], #4
LDR r0, [r10], #8
LDR r2, [r10], #4
Store_Kernel_Data
LDR r11, Kernel_Data
Store_Loop1
CMP r1, r3
BLE
Else
Store_Matrix_Data
LDR r12, Input_Data
MOV r3, #0
MOV r9, #0
Store_Loop2
LDR r9, [r10], #4
STR r9, [r11], #4
ADD r4, r4, #1
CMP r1, r4
BGT
Else
MOV r4, #0
ADD r3, r3, #1
Store_Input_Loop1
CMP r2, r3
BLE
Else
Store_Input_Loop2
LDR r9, [r10], #4
STR r9, [r12], #4
ADD r4, r4, #1
CMP r2, r4
BGT
Else
MOV r4, #0
ADD r3, r3, #1
Convolution
MOV r3, #0
MOV r9, #0
MOV r10, #0
LDR r11, Kernel_Data
LDR r12, Input_Data
LDRr10, Convolution_middle_result
before_Loop1
MOV r3,#0
ADD r3, r3, r1
SUB r3, r3, #1
Loop1
CMP r2, r3
Else
BGT
Calculation
LDR r10, Convolution_middle_result
LDR r3, Convolution_result
MOV r5, #1
ADD r6, r1, r0
Calculation_condition
CMP r2, r6
BNE
ADD r5, r5, #1
ADD r6, r6, r0
BEQ
MUL r6, r5, r5
MUL r7, r1, r1
MOV r8, #0
MOV r4, #0
Calculation_Loop1
CMP r6, #0
BGT
MOV r2, r5
Calculation_Loop2
CMP r7, #1
BGT
LDR r9, [r10], #4
ADD r4, r4, r9
SUB r7, r7, #1
Else
STR r4, [r3], #4
SUB r6, r6, #1
MOV r4, #0
MUL r7, r1, r1
ADD r6, r6, #1
before_Loop2
MOV r4,#0
ADD r4, r4, r1
SUB r4, r4, #1
Loop2
CMP r2, r4
Else
ADD r3, r3, r0
BGT
before_Loop3
MOV r5, #0
Loop3
CMP r5, r1
Else
LDR r11, Kernel_Data
ADD r4, r4, r0
BLT
before_Loop4
MOV r6, #1
Loop4
LDR r7, [r11], #4
ADD r9, r3, r5
SUB r9, r9, r1
ADD r9, r9, #1
MUL r8, r9, r2
ADD r9, r8,r4
SUB r9, r9, r1
ADD r9, r9, r6
LDR r8, [r12, r9, LSL #2]
MUL r9, r8, r7
STR r9, [r10], #4
ADD r6, r6, #1
MOV r9, #0
MOV r8, #0
CMP r6, r1
BLT
Else
ADD r5, r5, #1
Max_Pooling
LDR r0, = Data_set
LDR r1, [r0, #8]
LDR r3, Convolution_result
LDR r0, result
MOV r4, #0
MOV r5, #0
MOV r6, #0
MOV r7, #0
MOV r8, #0
MOV r9, #0
MOV r10, #0
MOV r11, #0
MOV r12, #0
Max_Pooling_before_Loop1
MOV r4, #0
NEG r9, r9
Max_Pooling_Loop1
CMP r2, r4
BGT
Else
Max_Pooling_before_Loop2
MOV r5, #0
HO
MOV pc, #0
END
Max_Pooling_Loop2
CMP r2, r5
BGT
Else
Max_Pooling_before_Loop3
MOV r6, #0
STR r9, [r0], #4
MOV r9, #0
ADD r5, r5, r1
Max_Pooling_Loop3
CMP r1, r6
Else
BGT
Max_Pooling_before_Loop4
MOV r5, #0
Max_Pooling_Loop4
ADD r11, r4, r6
MUL r10, r11, r2
ADD r10, r10, r5
ADD r10, r10, r7
LDR r8, [r3, r10, LSL #2]
CMP r8, r9
GT
MOV r9, r8
Else
ADD r7, r7, #1
CMP r1, r7
BGT
Else
ADD r6, r6, #1

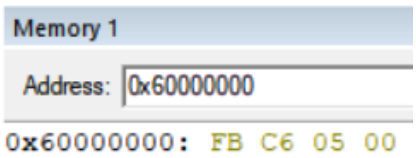
4. Performance & Result

- test case 1

State: 2,341

Code size: 772

State * State * Code size = 4,230,776,932

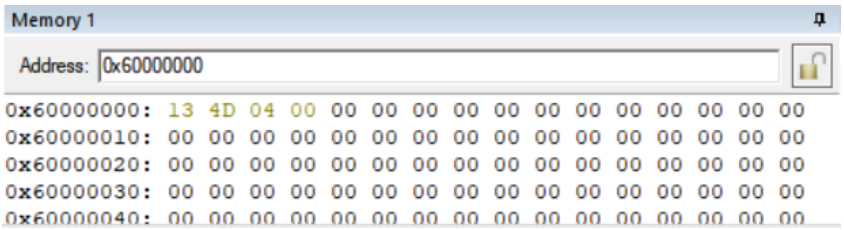


- test case 2

State: 1,918

Code size: 712

State * State * Code size = 2,619,251,488

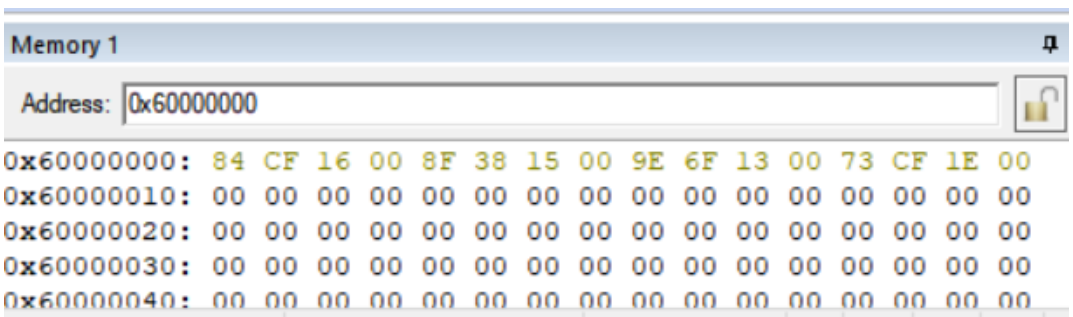


- test case 3

State: 34,989

Code size: 1,112

State * State * Code size = 1,361,343,894,552



5. Problems and solutions that arise

- 레지스터 부족 문제

크게 나누자면 Convolution 함수와 Max Pooling 함수, 총 2 개의 서브함수를 이용하여 구현하였다. 그 중 Convolution 함수의 구성 함수들을 다시 본다면, Convolution 함수와 Calculation 함수, Radix2_of_Convolution_calculate 함수가 존재한다. 여기서 발생하는 문제는 stride 값과 Convolution size, Input size 의 값을 저장하는데 register 3 개, Convolution Data 와 Input Data 를 저장하는데 register 2 개, 4 중 반복문의 카운트 조건으로 register 4 개, MUL 연산을 하는데 register 3 개를 사용하여, 결과적으로 Booth algorithm 을 만들었으나, 대입하지 못하였다. Booth algorithm 을 구현하기 위해선 적어도 register 가 4 개 이상 필요하지만, 고정적인 값을 가지고 있어야 하는 register 가 너무 많았기 때문이다. 이는 4 중 반복문이 아닌 3 중 반복문이나 2 중 반복문으로 줄이고 코드의 길이를 늘렸으면 해결할 수 있다고 생각해 구조를 바꾸려고 노력하였으나, 결국 구조를 바꾸지 못하고 Booth algorithm 을 주석처리한 후 MUL Instruction 을 사용하였다.

- 퍼포먼스 문제

프로젝트의 규모가 커지고 MUL Instruction 을 대체하는 Booth Algorithm 을 사용할 경우 굉장히 많은 횟수로 Booth Algorithm 이 반복되었다. (이는 곧 state 의 크기가 기하급수적으로 커지는 것을 의미한다.)

퍼포먼스를 줄이기 위하여 먼저 Branch 명령어를 이용한 코드 구현을 줄여 퍼포먼스를 향상하였다. 그 대신 조건문을 사용하여 state 를 줄일 수 있었다.

6. Conclusion

- 기대되는 학습 효과

이번 프로젝트를 통해 얻을 수 있는 가장 큰 학습 효과는 레지스터와 메모리를 제어하는 부분에서 실력이 향상되었을 것이라고 생각한다. 지난 과제들과는 다르게 프로젝트에서는 레지스터가 모자라는 경우가 많았고, 그렇기 때문에 레지스터에 있는 값을 메모리에 저장하고 빼는 경우가 자주 있었다. 레지스터는 중요한 자원임을 상기시키는 프로젝트였다.

또, 코드 구현을 위하여 C 언어를 사용하여 먼저 구조의 틀을 잡은 후 어셈블리 코딩을 진행하였다. 아래는 직접 구현한 Convolution 함수이다.

```
void convolution(int c_arr[], int d_arr[], int i, int j) {
    i = sqrt(i);                //sqrt of convolution matrix size   (if number of convolution matrix element is 9 sqrt = 3)
    j = sqrt(j);                //sqrt of data matrix size          (if number of data matrix element is 36 sqrt = 6)
    int stride = 1;             //value of stride
    int result[200] = {NULL,};  //result matrix
    int i2 = 0;                 //i2 = 세로위치
    int j2 = 0;                 //j2 = 가로위치

    for (i2 = 0; j > i2 + i +  - 1; i2 += stride) {
        for (j2 = 0; j > j2 + i +  - 1; j2 += stride) {
            for (int x = 0; x < i; x++) {
                for (int y = 0; y < i; y++) {
                    result[i * x + y] = d_arr[j * (i2 + x) + j2 + y] * c_arr[i * x + y];
                    //printf("result좌표: %3d, d_arr좌표 값: %3d, kernel좌표 값: %3d \n", i * x + y, d_arr[i * (i2 + x) + j2 + y], c_arr[i * x + y]);
                    printf("%5d ", result[i * x + y]);
                }
                printf("\n");
            }
            printf("\n");
        }
    }
}
```

여기에 Visual Studio 라는 프로그램을 사용한다면 디버깅 중 디스어셈블리를 표시할 수 있는 기능이 있는데, 이 기능을 사용하면서, 대략적인 구조를 알 수 있었고, 어셈블리가 C 언어보다 1 차이가 중요하다는 느낌을 받을 수 있었다. 프로젝트를 진행하면서 나눈 단계들은 다음과 같다.

1. C 언어로 Convolution 구현하기 (Max Pooling 의 경우 구조가 비슷할 것이라고 판단함.)

2. MUL Instruction 을 대체할 Radix-2 Booth Algorithm 구현

3. Convolution 과 Max Pooling 을 진행하면서 필요한 일반화된 공식 도출

4. Convolution 알고리즘에 필요한 하위 함수들 어셈블리어로 구현

5. Max Pooling 알고리즘에 필요한 하위 함수들 어셈블리어로 구현

이렇게 다른 언어로 접근하는 방법을 사용해보면서 어려운 문제를 단계를 나누어 하나하나 풀어나가는 게 가장 정확하면서 빠른 방법이라는 것을 느꼈다.