어셈블리프로그램 설계 및 실습 - term project

Matrix convolution

결과 보고서

학과: 컴퓨터정보공학부

담당교수: 이준환 교수님

학 번: 2018202050

이 름: 송영욱

목차

Introduction

1) 프로젝트 설명

Floating point로 이루어진 64-by-64 행렬 데이터(First_data)를 66-by-66으로 **Padding**하는 과정 거치고 3-by-3의 행렬 데이터(Second_data)와 행렬 곱 과정을 통해 **Convolution**을 진행하고 **Sub_sampling**과정을 통해 32-by-32행렬을 구하는 프로젝트이다.

2) 목적

Floating pointer의 연산을 이해하고 연산 능력을 향상시킨다. 특히 행렬 곱셈에서 Floating pointer를 MUL instruction이 아닌 Booth algorithm을 이용한 MUL기능 구현을 통해 Assembly coding 능력을 향상시킨다.

3) 일정

<일정 계획표>

	11/22	11/23	11/24	11/25	11/26	11/27	11/28	11/29	11/30
제안서 작성									
코드 작성									
코드 검증									
결과 보고서 작성									

<실제 일정표>

	11/22	11/23	11/24	11/25	11/26	11/27	11/28	11/29	11/30
제안서 작성									
코드 작성									
코드 검증									
결과 보고서 작성									

계획서에선 코드작성을 빨리 시작하려고 했으나 다른 프로젝트들과 겹쳐서 늦게 시작했다.

그래도 일정에 여유가 좀 있어서 계획한 코드 작성기간과 비슷한 기간동안 작성할 수 있었다.

하지만 검증시간과 검증 후 코드 수정을 계획한 기간이 짧아지는 문제가 있었다.

Project Specification

이번 프로젝트는 전체적인 흐름부터 파악을 해야 이해하기가 쉽다.

프로젝트에 있는 큰 흐름은 총 3가지로 Padding, Convolution, Sub_sampling이다.

하지만 다르게 생각해서 Padding, Convolution, Sub_sampling을 한 번에 했다.

<Padding & Convolution>

Padding은 64-by-64의 행렬을 가장 가까운 값을 복사해서 66-by-66크기의 행렬로 확장하는 것이다. Padding을 하는 이유는 다음 과정인 Convolution 때문이다.

Convolution은 3-by-3인 Second_data와 행렬 곱을 통해 연산을 해야 하는데 Padding 하기전의 행렬의 (0,0) 같은 경우는 주변에 값이 있는 것이 아니기 때문이다.

A	B
C	D

예를 들어 다음과 같은 2-by-2 행렬이 있다고 생각하면

A의 Convolution을 진행할 주변의 값들이 없다.

그러므로 Padding을 통해 아래 그림같이 확장시켜준다.

A	A	B	B
A	A	B	B
C	C	D	D
C	C	D	D

2-by-2를 4-by-4 행렬로 확장했다.

확장된 값은 가장 가까운 값으로 복사된다.

이렇게 확장이 되면 A를 Convolution을 진행할 주변 값들이 생겼다.

하지만 프로젝트 수행 도중 든 생각이 Convolution을 위한 Padding이라면 Padding을 통해 확장될 값은 이미 알고 있기 때문에 굳이 Memory에 Padding값을 저장하지 않고, 바로 Convolution을 진행한다면 State를 많이 줄일 수 있을 것 같다고 판단했다. 그래서 따로 Padding하는 과정 없이 바로 Convolution을 진행했다.

Convolution은 크게 4가지로 경우를 나누어 생각했다. 각 경우를 일부분을 통해서 어떻게 계산 진행됐는지 확인할 수 있다. Padding은 **가장 가까운 값을 복사한다**는 점을 생각하면서 계산한다.

Convolution을 진행하기 위해선 행렬 곱이 어떤 식으로 진행되는 지부터 파악해야 한다.

예를 들어 3-by-3행렬로 생각해볼 수 있다.

A	B	C
D	E	F
G	H	I

X

a	b	c
d	e	f
g	h	i

위의 행렬로 예를 들면 A * a + B * b + C * c + D * d + E * e + F * f + G * g + H * h + I * i 로 계산한다.

<Convolution의 4가지 경우>

1) 첫번째 줄 왼쪽 꼭지점 (0,0)

A	A	B
A	A	B
C	C	D

2) 첫번째 줄 모서리(0,2N) (0<N<32)

A	B	C
A	B	C
D	E	F

3) 중간 왼쪽 모서리 (2N,0) (0<N<32)

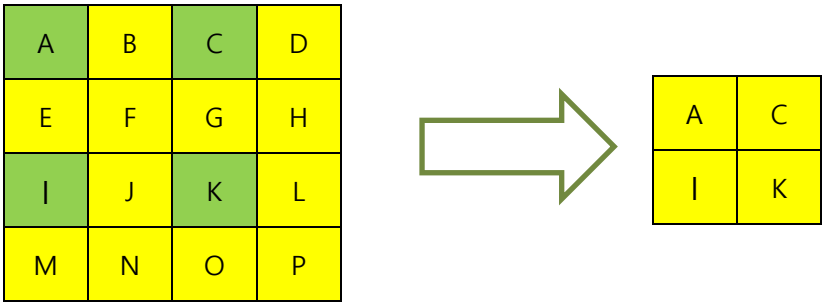
A	A	B
C	C	D
E	E	F

4) 중간값 (2X,2Y) (0<X<32, 0<Y<32)

A	B	C
D	E	F
G	H	I

홀수 행과 홀수 열은 계산을 생략해도 되기 때문에 경우에서 제외했다.

Sub_sampling



Sub_sampling은 Convolution의 결과 값을 짝수 행, 짝수 열만 뽑아서 Matrix_result에 저장한다.

과정을 거치면 64-by-64 행렬이 32-by-32로 줄어든다.

하지만 Convolution을 진행하면서 Matrix_result에 바로 저장을 해주면 Sub_sampling도 동시에 진행이 가능하다.

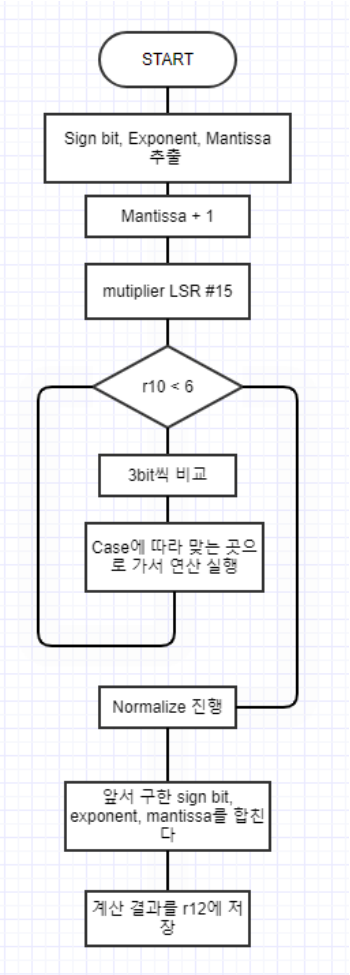
Algorithm

처음에 Matrix_first 와 Matrix_second 값을 레지스터에 불러온다.

첫번째 줄 왼쪽 꼭지점 (0,0) 계산을 위해 곱을 총 9 번 진행한다. 이때 나온 계산 값을 바로 Matrix result 에 저장을 한다. 주의할 점은 Padding 을 따로 하지 않았기 때문에 Padding 값을 계산하기 위해서 가장 가까운 값으로 계산한다. 이렇게 (0,0)을 구한다.

첫번째 줄 모서리(0,2N) (0<N<32)는 값을 읽을 때 짝수만 읽어야 하므로 8byte 씩 이동시켜가면서 값을 읽었다. 마찬가지로 행렬 곱으로 계산을 하고 Matrix_result 에 바로 값을 저장한다. Loop 를 통해서 31 번 반복을 하면 첫번째 줄을 계산할 수 있다.

중간 값은 (2N,0) (0<N<32)도 마찬가지로 홀수 행, 홀수 열은 읽지 않는다 그러므로 한 줄을 건너뛰어야 한다. 그러므로 #260 을 통해 읽을 주소를 건너뛰었다. 그렇게 건너뛰어 가면서 Loop 문을 통해 31 번 반복을 통해 전체를 구할 수 있었다.



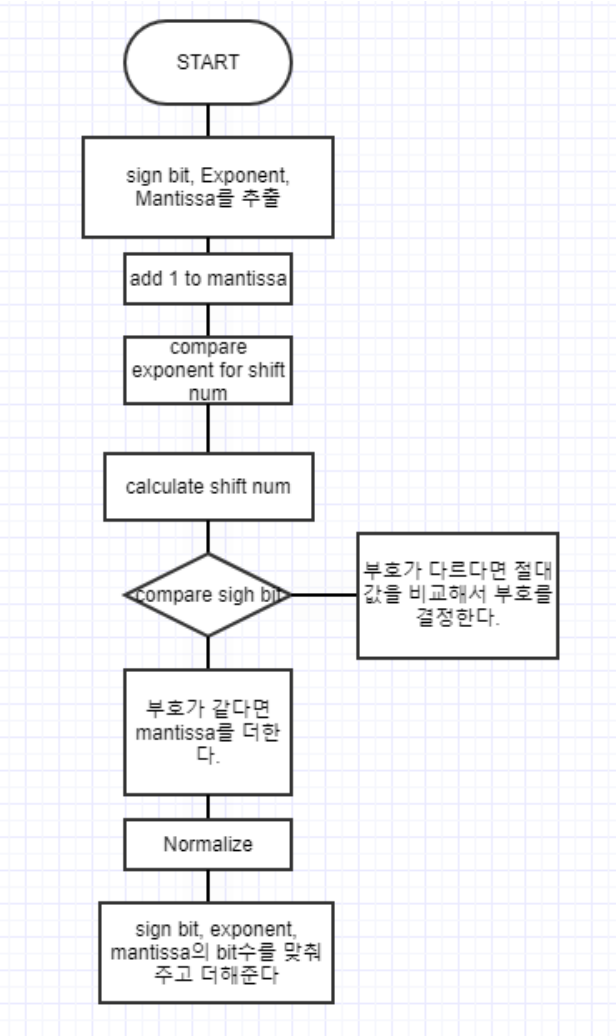
LSR #15 은 state 를 줄이기위해 해줬다.

기본적으로 주어진 값이 0~255 사이의 정수라면 하위 16bit 은 무조건 0 이기 때문에 16 개를 지워도 값이 변하지 않는다.

하지만 Booth algorithm 을 진행하면서 하위 bit 에 0 을 추가해주기 때문에 결과적으로 16 개를 지우고 1 개를 추가해줘서 15 개만 지웠다.

radix-4 로 구현을 했기 때문에 3bit 씩 비교해서 맞는 경우에 가서 연산을 진행했다.

결과를 r12 에 저장을 하고 바로 ADD 함수로 간다.



Multiplication 과 마찬가지로 sign bit 과 exponent, Mantissa 를 추출한다.

그다음 mantissa 에 1 을 더한다.

exponent 비교를 통해 shift num 을 구해준다.

shift num 이 0 이 아닌경우 Exponent 값이 작은 값이 mantissa 값을 shift num 만큼 LSR 해준다.

Performance & Result

Project Specification애서 Performance는 code size * state * state라고 명시 되어있다.

```
linking...
Program Size: Code=18128 RO-data=0 RW-data=0 ZI-data=0
".\Objects\Matrix_convolution.axf" - 0 Error(s), 2 Warning(s).
Build Time Elapsed:  00:00:00
```

PC $	0x000004AC
Mode	Supervisor
States	1742914
Sec	0.00000000

Performance를 계산해보면 18128 * 1742914 * 1742914 = 55,068,317,704,186,688으로 계산할 수 있다.

Result는 앞서 나눈 4개의 경우를 확인해서 Convolution값을 확인하고 그 값들이 Sub_sampling되었는지 확인했다

1) <첫번째 줄 왼쪽 꼭지점> (0,0)

167	167	167
167	167	167
166	166	165

0.075	0.124	0.075
0.124	0.204	0.124
0.075	0.124	0.075

둘을 행렬 곱을 통해 계산하면 아래 표처럼 값이 나오는데 이것들의 합을 구하면 166.651이다.

12.525	20.708	12.525
20.708	34.068	20.708
12.45	20.584	12.375

2) 첫번째 줄 모서리(0,2)

167	165	163
167	165	163
165	164	163

0.075	0.124	0.075
0.124	0.204	0.124
0.075	0.124	0.075

둘을 행렬 곱을 통해 계산하면 아래 표처럼 값이 나오는데 이것들의 합을 구하면 164.726이다.

12.525	20.46	12.225
20.708	33.66	20.212
12.375	20.336	12.225

```
0x60000000: 166.651         164.726         166.042         168.972         128.103
```

첫번째 줄 왼쪽 꼭지점인 (0,0)값이 Matrix_result의 (0,0)에 저장된 것을 볼 수 있다.

그리고 첫번째 줄의 모서리인 (0,2)의 값도 (0,1)에 저장되었다.

2) 중간 모서리(2,0)

166	166	165
165	165	165
165	165	165

0.075	0.124	0.075
0.124	0.204	0.124
0.075	0.124	0.075

둘을 행렬 곱을 통해 계산하면 아래 표처럼 값이 나오는데 이것들의 합을 구하면 165.199이다.

12.45	20.584	12.375
20.46	33.66	20.46
12.375	20.46	12.375

```
0x60000078: 134.398         132.322         165.199
```

중간 모서리인 (2,0)값이 Matrix_result의 (1,0)에 저장된 것을 볼 수 있다.

4) 중간 값(2,2)

165	164	163
165	165	164
165	166	167

0.075	0.124	0.075
0.124	0.204	0.124
0.075	0.124	0.075

둘을 행렬 곱을 통해 계산하면 아래 표처럼 값이 나오는데 이것들의 합을 구하면 164.876이다.

12.375	20.336	12.225
20.46	33.66	20.336
12.375	20.584	12.525

```
0x60000080: 165.199        164.876
```

중간 모서리인 (2,2)값이 Matrix_result의 (1,1)에 저장된 것을 볼 수 있다.

나머지 값들은 모두 반복문으로 반복되기 때문에 옳은 값이 들어갔다고 할 수 있다.

Conclusion

이번 프로젝트는 Floating pointer 연산이 핵심이었다.

맨 처음 계획은 Padding을 무조건 해야 한다고 생각하고 Padding한 값들을 다른 주소에 따로 저장을 하고 행렬 곱 연산을 진행하려고 했었다. 그 다음 Convolution을 진행하는데 굳이 Padding이 아니더라도 그 위치의 값을 이미 알고 있기 때문에 할 필요가 없을 것 같다는 생각이 들었다.

그렇게 Padding을 진행하지 않고 Convolution을 진행했다. 9가지 경우로 Convolution을 나누어서 계산을 하고 난 다음 Sub_sampling까지 진행을 다하고 난 뒤에 보니 state수가 너무 커진 것 같은 느낌이 들었다. 어떻게 줄일까 고민을 하고 있었는데 실습 수업에서 조교님이 Convolution한 값을 바로 Matrix_result에 넣어도 된다는 식으로 말씀을 하신 것을 듣고 바로 수정을 했다.

수정하는 과정도 그렇고 처음 진행할 때도 그렇고 multiplication과 add를 진행하고 나니 남는 레지스터가 없었다. 그렇게 해서 원래 사용하지 않으려 했던 sp인 r13하나를 가지고 이중 Loop를 구현하려고 하는 것이 어려움이 많았다.

레지스터 하나를 가지고 Loop하나를 만드는 것은 쉬웠지만 이중 Loop을 하려고 하니 조건을 주는 것이 많이 어려웠다. 그렇게 해서 생각해낸 것이. 두번째 Loop에 들어가기 전의 r12값을 메모리에 저장해두고 Loop를 탈출했을 때 다시 값을 불러와 비교를 통해 구현할 수 있었다.

메모리에 왔다 갔다 하는게 비효율적인 것은 알았지만, 레지스터 줄일 방법이 떠오르지 않아서 이런 선택을 했다.

add는 앞선 실습시간에 구현을 해서 금방 할 수 있었지만, MUL을 사용하지 않고 booth algorithm을 사용해 구현하는 것에 시간이 엄청 오래 걸렸다. 다른 수업들 에서도 booth algorithm을 배웠지만 어떻게 돌아가는지는 알아도 assembly를 통해서 구현하는 것이 어려움이 많아 시간이 많이 걸렸다.

그리고 Floating point로 Data들이 주어지는데 이것들이 맞게 계산된 건지 확인하기 위해서 일일이 값을 바꿔서 행렬 곱을 진행하는데 개수도 많고 하나를 구하는데 9번 곱하고 더해야 하는 수고스러움이 있었다. 계산기를 쓰기는 했지만 계산실수가 잦아 계속 계산하는 과정에서도 시간이 많이 걸린 것이 아쉽다.

그런 과정들을 통해 수정을 하니 Convolution을 하는 경우도 4가지로 줄일 수 있었고 code size도 엄청 줄이고, 특히 State가 많이 줄었다. 성능을 계산하는데 State는 제곱으로 들어가기 때문에 상당히 중요하다고 판단했다.

이번 프로젝트를 통해 가장 크게 느낀 것은 프로젝트를 진행함에 있어서 큰 흐름부터 파악을 해야 한다고 생각했다. 바로 무작정 구현하는 것이 아니라 Padding을 해야 하는 이유, Convolution을 하는 이유를 고려해보면 결국은 result를 구하기 위함인데 굳이 안 해도 될 것을 해서 시간도 오래 걸리고 성능도 안 좋게 나왔다.

학습적으로 생각해보면 Floating point의 연산을 하는 것에 익숙해져서 계산을 쉽게 할 수 있을것 같고 Booth algorithm같은 경우에 다른 수업시간에도 하는 알고리즘이기 때문에 한 번 더 알고리즘을 이해하는데 도움이 되었다.

물론 레지스터를 효율적으로 관리하는 것은 하지 못했지만, 효율적인 방법은 아니지만 레지스터가 1개 있을 때 이중 Loop를 구현하는 법 그리고 읽어올 주소를 계산하면서 Convolution을 수행해야 해서 어떻게 값들을 더하고 빼서 값들을 불러올지 이해할 수 있었다.