**5X5 행렬 곱 계산 코드와 성능 개선 방법**

**보고서 개요**

1. TPU 모델 수정 결과

2. JZ 명령어를 활용한 5X5 행렬 곱 계산 수행 코드와 이에 대한 설명.

3. 성능 향상 방법

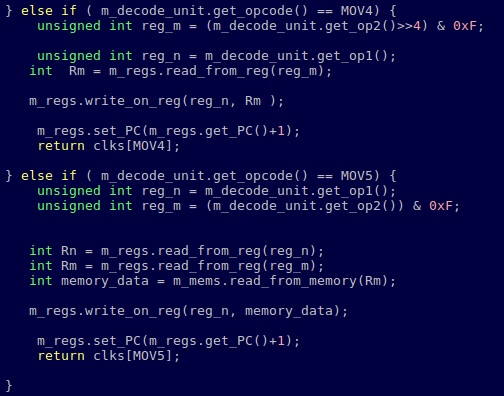
4. JZ 명령어를 사용하지 않은 5X5 Matrix 코드의 성능

5. JZ, MOV2, MOV5, MUL 명령어를 사용하지 않은 5X5 Matrix 코드의 성능

6. 이에 대한 고찰.

1. **TPU 모델 수정 결과**

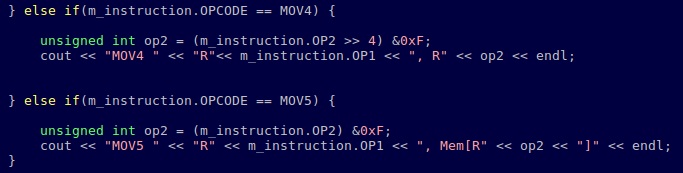
<CExecute.cpp>에서의 MOV4,5



MOV4 : 1000 / Rn / Rm 0000

MOV5 : 1001 / Rn / 0000 Rm

<CDecode.cpp>에서의 MOV4,MOV5



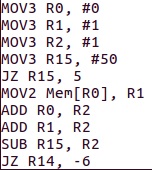
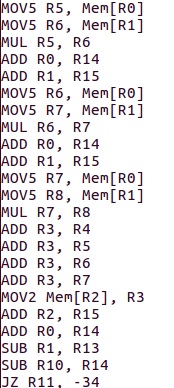
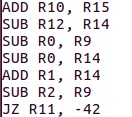
MOV4 : Rn 🡨 Rm

MOV5 : Rn 🡨 Mem[Rm]

2. **JZ 명령어를 활용한 5X5 행렬 곱 계산 수행 코드**

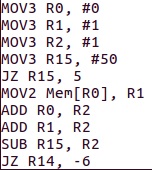
TPU 명령어 파일 = Matrix55.bin 라인수 : 61

<**Matrix55.bin의 전체 어셈블리 코드**>

<**코드에 대한 설명**>

**과정 1** : 가장 먼저 수행한 것은 Memory의 M[0] ~ M[49]까지 행렬A, B의 값들을 넣은 것입니다.

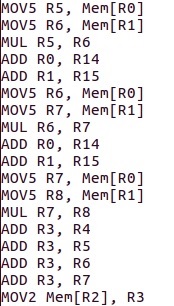
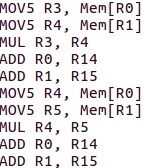
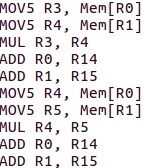
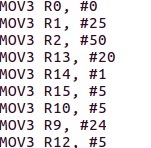


행렬 A,B의 값들은 1~50이므로 레지스터 R1의 값을 1로 초기화한 뒤, MOV2 Mem[R0] R1 명령어를 이용하여 R0, R1값들을 1씩 높여가며 50번 반복 합니다.

JZ R15 5 // JZ R14 -6 명령어로 Loop를 만듭니다.

과정 1 결과 : M[0~49] : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50

**과정 2** : 우선 행렬 C의 C0를 계산하는 코드를 짰습니다. C0는 행렬 A의 1행과 행렬 B의 1열을 각각 곱한 뒤 더한 결과값입니다.



ADD R0 R14 🡪 행렬A의 1행을 모두 계산하기 위해 Mem[R0]에서 R0의 값을 1씩 높여감.

ADD R1 R15 🡪 행렬B의 1열을 모두 계산하기 위해 Mem[R1]에서 R1의 값을 5씩 높여감.

R0=행렬 A의 값을 불러오기 위한 레지스터 🡪 MOV5 R3, Mem[R0] 🡪 행렬A의 값을 R3저장

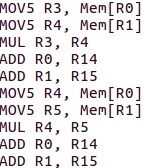
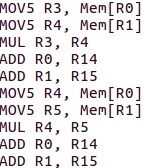
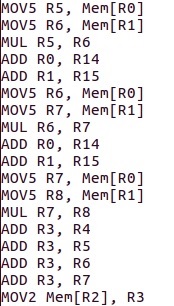
R1=행렬 B의 값을 불러오기 위한 레지스터 🡪 MOV5 R4, Mem[R1] 🡪 행렬B의 값을 R4저장

R2=행렬 C의 값을 저장하기 위한 레지스터 🡪 MOV2 Mem[R2], R3 🡪 최종 결과값 C0 = R3를

행렬C C0 = Mem[R2]에 저장

과정 2 결과 : M[50] : 590

**과정 3** : 과정 2에서 R2 즉, 행렬 C의 1열을 모두 계산하기 위해서 R2를 5씩 높여가며 5번 반복.

<추가>

ADD R0 R14

SUB R1 R13

ADD R2 R15

SUB R10 R14

JZ R11 -34

<추가>

JZ R10 33

과정 3 결과 : M[50~74] : 590 0 0 0 0 1490 0 0 0 0 2390 0 0 0 0 3290 0 0 0 0 4190 0 0 0 0

과정 2의 양쪽 끝에 JZ 명령어를 추가하여 사이의 코드가 5번 반복되도록 한다.

추가된 SUB R1 R13 🡪 과정 2가 5번 반복된다고 했을 때 1번 수행시에 행렬B를 나타내는 R2

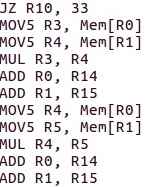
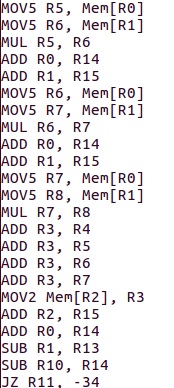
가 25에서 45로 커진 상황이므로 다시 25로 돌려주기 위해 -20.

즉, 행렬A를 나타내는 R0는 꾸준히 1씩 더해지고 행렬B를 나타내는 R1은

(25🡪30🡪35🡪40🡪45)🡪(25🡪30🡪……)을 반복하므로 행렬A는 모든 행과 열이 계산되어지고

행렬B는 1열만 계산되어진다.

**과정 4** : 과정 3을 5번 수행하고 나면 행렬B를 나타내는 R1을 1씩 더하여 (26🡪31🡪36🡪41🡪46), (27🡪32🡪37🡪42🡪47), (28🡪33🡪38🡪43🡪48), (29🡪34🡪39🡪44🡪49) 또한 계산되도록 5번 반복하여 행렬C 전체 행과 열을 구합니다.

과정 3의 양쪽 끝에 JZ 명령어를 추가하여 사이의 코드가 5번 반복되도록 한다.

ADD R10 R15 🡪과정 3을 수행하고 나면 R10은 0이 되어 있으므로 과정 3을 5번 추가적으로

반복하기 위해 R10을 5로 되돌린다.

SUB R0 R9 & SUB R0 R14 🡪 R0 - 25 🡪 행렬A를 나타내는 R0는 과정 3을 한 번 수행하고 나

면 25가 되어 있으므로 다시 행렬A : M[0~24]이 계산되어 지려면 R0 = 0으로 되돌아 가야한다.

<추가>

JZ R12 41

<추가>

ADD R10 R15

SUB R12 R14

SUB R0 R9

SUB R0 R14

ADD R1 R14

SUB R2 R9

JZ R11 -42

ADD R1 R14 🡪 과정 3을 한 번 수행하고 나면 행렬B의 두번째 행으로 과정 3을 반복해야

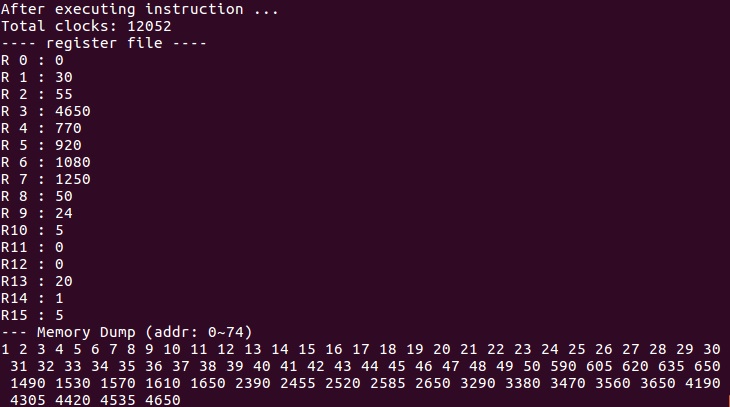
하므로 R1 + 1 을 해야 두 번째 행이 계산되어 지고 그 다음 또 1을 더하면 세 번째 행이 계

산 되어진다.

SUB R2 R9 🡪 계산 결과값을 넣는 R2은 행렬C를 나타낸다. 과정3을 한 번 수행하면 R2 = 75

가 되어있다. 두 번째 행의 C1부터 다시 계산되어지려면 R2 – 24이 되어야 한다.

**최종 결과**



3. **성능 향상 방법**

**방법 1** : 클럭수가 큰 JZ 명령어를 사용하지 않기.

JZ 명령어를 사용하지 않게 되면 입력해야 할 tpu 명령어 라인 수는 비약적으로 커지지만 Total Clocks는 반대로 작아지게 될 것입니다. JZ를 사용하더라도 결국에는 반복되어지 명령어들은 JZ를 사용하지 않았을 때와 마찬가지로 수행됩니다. 또한, JZ명령어는 Compare 명령어로 그 결과값이 나오기 전에는 어떠한 일도 수행될 수 없기 때문에 JZ 명령어는 사용하지 않는 것이 가장 크게 성능 향상을 시키는 방법이라고 할 수 있습니다. 따라서, JZ 명령어를 사용하지 않는 것은 클럭수를 줄이는 데 효과적인 방법입니다.

**방법 2** : MUL 명령를 ADD, SUB 명령어로 대체.

MUL 명령어는 클럭수가 30으로 이때까지 사용한 명령어들 중 가장 큰 클럭수를 가집니다. ADD, SUB 명령어는 각각 클럭수가 4이므로 ADD, SUB 명령어를 7번 이하로 사용하여 MUL 명령어를 대체할 수 있다면 MUL 명령어를 사용하는 것보다 ADD, SUB 명령어를 사용하는 것이 성능 향상에 도움될 것입니다. 실제로는 MUL명령어를 ADD, SUB로 대체하기 위해서는 곱하려는 수를 MOV4 명령어로 다른 레지스터에 옮긴 뒤 ADD, SUB 명령어로 원래 레지스터에 옮긴 레지스터의 값을 더하거나 빼기를 합니다. MUL 명령어의 클럭수 30에서 MOV4 명령어의 클럭수 2를 빼면 28입니다. ADD, SUB 명령어를 7번 사용하면 MUL 명령어를 사용하는 것과 같아지므로 ADD, SUB 명령어를 6번 반복하여 사용하면 MUL 명령어를 사용하는 것보다 ADD, SUB 명령어를 사용하는 것이 성능 향상에 도움이 됩니다.

**방법 3** : MOV2, MOV5 명령어 MOV0, MOV1 명령어로 대체.

MOV0,1,2,5 명령어 모두 레지스터🡪메모리, 메모리🡪레지스터 형태로 데이터를 옮기는 명령어입니다. 다만 MOV2,5는 레지스터를 거쳐서 메모리로 옮기거나 메모리에서 데이터를 빼오기 때문에 클럭수가 MOV0,1보다 4만큼 높습니다. 옮기거나, 빼오고자 하는 메모리의 순번이 높지 않다면 MOV0,1 명령어로 대체할 수 있을 것입니다.

4. **JZ 명령어를 사용하지 않은 5x5 Matrix 코드의 성능.**

TPU 명령 파일 = MatrixExceptJZ.bin 라인 수 = 986

MatrixExceptJZ.bin의 명령어 라인 수는 986입니다. 너무 길기 때문에 모든 명령어를 보여드리는 것이 불가능하기 때문에 JZ 명령어를 사용했을 때와의 차이점에 대해서 설명드리겠습니다.

<**과정1**> : 메모리 M[0~49]에 행렬 A, B의 값을 넣는 과정을 JZ 명령어 없이 수행.

JZ 명령어를 사용했을 때의 코드에서

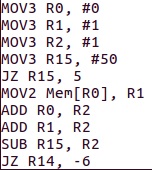
JZ R15 5 // JZ R14 -6 코드를 지우고

그 사이의 코드를 50번 반복합니다.

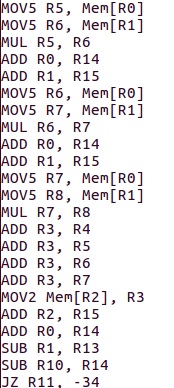
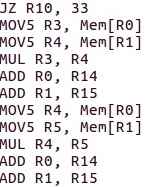
JZ 명령어가 없어짐에 따라 필요가 없어진

MOV3 R15 #50 & SUB R15 R2 명령어를

지웁니다.



<**과정2**> : 5X5 Matrix 수행을 JZ 명령어 없이하는 과정



ADD R10 R15

SUB R12 R14

SUB R0 R9

SUB R0 R14

ADD R1 R14

SUB R2 R9

JZ R11 -42

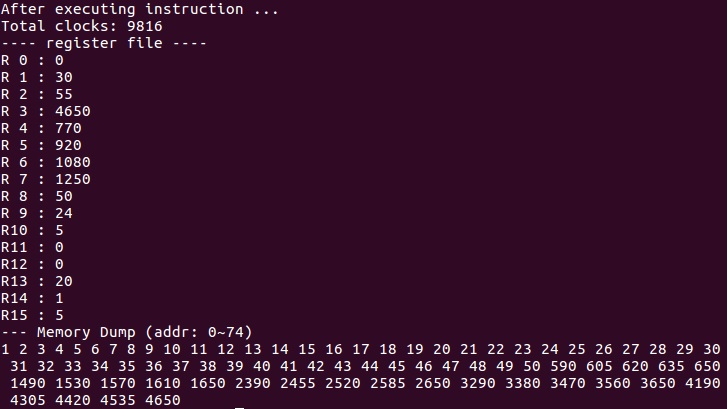
JZ R12 41

큰 Loop 안의 작은 Loop : JZ R10 33 // JZ R11 -34 를 지우고 안의 내용이 5번 반복되도록 4번 복사&붙여넣기 합니다.

다음, 큰 Loop : JZ R12 41 // JZ R11 -42 를 지우고 안의 내용이 5번 반복되도록 4번 복사&붙여넣기 합니다.

JZ 명령어와 관련된 SUB R12 R14 & SUB R10 R14 & ADD R10 R15 명령어들을 삭제합니다.

<**결과**>



Total Clocks : 12052 🡪 9816

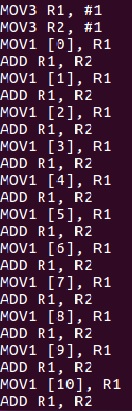
JZ 명령어와 JZ 명령어와 관련된 명령어를 없애는 것만으로 2,236만큼의 클럭수가 줄어드는 것을 확인할 수 있습니다.

5. **JZ, MOV2, MOV5, MUL 명령어를 사용하지 않은 5x5 Matrix 코드의 성능**

TPU 명령어 파일 = MatrixJZMOV1.bin 라인수 = 981

위와 마찬가지로 전과 달라진 점을 중점으로 설명하도록 하겠습니다.

<**과정1**> : 메모리 M[0~49]에 행렬 A, B의 값을 넣는 과정을 JZ, MOV2 명령어 없이 수행.



MOV2 명령어보다 MOV1 명령어가 클럭수가 4만큼 작습니다.

또한, direct로 메모리의 위치를 가리키므로 레지스터 R0에 대한

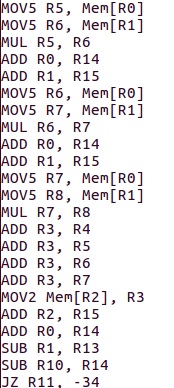
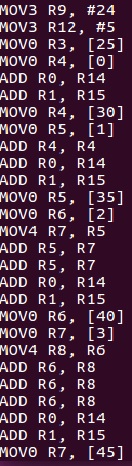
정보를 활용하지 않아도 되기 때문에 클럭수는 더욱 줄어듭니다.

MOV2 명령어를 사용할 때에는 레지스터 R0를 이용하여 메모리의 위치를 정했고 R0, R1를 1씩 높여가며 메모리에 행렬A, B의 값을 넣었습니다. ( MOV2 Mem[R0] R1 )

MOV2 명령어를 사용하지 않으려면 MOV1을 이용하여 메모리의 위치를 직접 값으로 지정해줍니다.

MOV1 [direct] R1 🡪 direct : 0 ~ 49

<**과정2**> : 5X5 Matrix 수행을 JZ 명령어 없이하는 과정

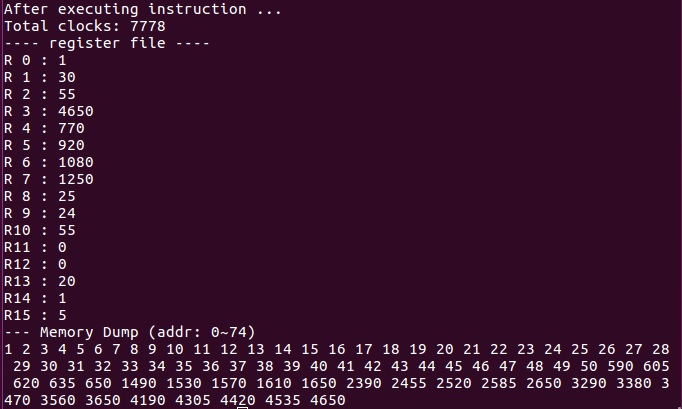
Matrix 수행 과정에서의 MOV5 명령어 MOV0 명령어로 모두 대체합니다. MOV0 명령어는 direct하게 메모리의 주소를 지정하기 때문에 거의 모든 MOV0 명령어들의 메모리 주소를 일일이 수정해야 합니다.

MOV5 🡪 MOV0 : 레지스터 R0, R1로 말해주던 메모리의 위치를 MOV0 명령어를 사용하면서 direct하게 입력.

MUL 🡪 ADD : 행렬A X 행렬B를 수행할 때 행렬 A의 성분의 값이 7 이하라면 MUL 명령어를 ADD 명령어로 대체 가능합니다.

<바뀌기 전> <바뀐 후>

<**결과**>



Total clcks : 12052 🡪 7778 ( JZ & MOV2,5 & MUL 명령어 사용 vs 미사용 )

전체 클럭수가 총 4274만큼 줄어든 것을 확인할 수 있습니다.

6. **이에 대한 고찰**

JZ 명령어를 없애는 것만으로도 2,236의 클럭수가 줄어들었습니다. JZ 명령어로 Loop를 표현하기 위해서는 Loop에 씌울 코드들을 사이에 두고 위와 아래에 JZ 명령어를 사용해야 합니다. 이는 곧 한 루프를 표현하기 위해 적어도 2번의 JZ 명령어가 실행되어야 한다는 것이고, Loop가 2회 실행된다면 4번 이상 JZ 명령어가 실행되게 됩니다. 이렇게 되는 이유는 JZ 명령어가 지정된 값이 0일 때에만 Jump를 하기 때문입니다. 만약, 지정된 값이 0이 아닐 때에만 Jump를 하게 하는 명령어가 있다면, Loop를 표현할 때 한 개의 명령어로도 Loop를 표현할 수 있을 것입니다. 이는 곳 Jump를 하는 명령어의 실행수가 절반으로 줄어들 것을 의미합니다. 이렇게 된다면, 클럭수는 크게 줄어들 것으로 생각됩니다.

MUL 명령어는 저희가 사용하고 있는 명령어들 중 클럭수가 30으로 가장 큽니다. 그래서 MUL 명령어의 클럭수를 줄이는 것이 가장 효율적인 것으로 보입니다. 하지만 MUL 명령어를 대체하기 위해서는 ADD, SUB 명령어로 대체해야 되는데 ADD, SUB 명령어의 클럭수는 4이므로 MUL을 대체할 때에 한계가 있습니다. 또, ADD, SUB 명령어로 자기 자신을 반복적으로 더하거나, 빼기 위해서는 MOV4 명령어로 다른 레지스터에 그 데이터를 옮겨놓아야할 필요가 있습니다. MOV4 명령어의 클럭수는 2이므로 ADD, SUB를 7번 사용한다면 28클럭수이므로 곱하는 수, 두개의 값 중 하나가 8이하일 때에만 대체가 가능합니다.

MOV2,5를 MOV0, 1로 대체하는 경우, MOV2,5는 레지스터를 거쳐서 값을 받아오므로 클럭수가 MOV0,1보다는 크기는 하지만, 반복적인 규칙이 있을 경우 MOV2,5 명령어들이 코드를 표현하기에는 더욱 편하다고 볼 수 있습니다. 물론, 성능 향상면에서는 MOV0,1이 좀 더 좋다고 볼 수 있습니다.

이 3가지의 성능 향상 방법으로 약 70% 정도 성능을 향상시켰습니다. 하지만, 그 과정에서 코드를 짜는 시간이 너무 길었습니다. 그러므로, 성능을 향상시킬 수 있더라도 거기에 투자하는 시간 대비 효율을 따져서 그 접합점에서 코드를 짜는 것이 좀 더 효율적이라고 생각합니다.