어셈블리프로그램 설계 및 실습

Term Project 제안서

Inverse Matrix

학　　과: 컴퓨터공학과

담당교수: 이준환 교수님

분　　반: 화 5, 목 6, 7

학　　번: 2015722068

성　　명: 남윤창

1. 제목 & 목표

A. Title

Inverse Matrix

B. Object

임의의 부동소수점(Floating point) 데이터로 이루어진 NxN(1 <= N <= 20) 정방행렬에 대한 역행렬(Inverse Matrix)을 구하는 어셈블리 코드를 작성하는 것이다. FPU instruction인 VMUL, VDIV를 사용하지 않고 구하고 최대한 최적화하여 state를 줄이는 것을 목표로 한다.

2. 함수 알고리즘

A. Inverse Matrix

우선 기본적으로 NxN 정방행렬에서의 역행렬을 구하는 방법으로는 여인자(Cofactor)를 사용하는 방법과 가우스-조던 소거법(Gauss-Jordan Elimination)이 있다. 여인자는 너무 복잡하다고 판단하여 강의자료에 나온 가우스-조던 소거법을 채택하여 구현할 예정이다. 가우스-조던 소거법은 2.-B 섹션에 기술하였다.

본 프로젝트의 조건은 역행렬이 항상 존재하고 0인자가 없다고 가정하는 것이다. 영인자란 두 행렬 A, B의 곱이 0이 되는 경우를 말한다. 이런 경우 역행렬을 갖지 않는다. 역행렬이 항상 존재한다는 것은 모든 행렬을 가역행렬(Invertible Matrix)이라는 뜻이다. 2x2로 예를 들자면 판별식 ad – bc 가 0이 아닌 경우 즉 determinant det(A)가 0이 아닌 경우를 뜻한다

본 프로젝트에서는 Code Size는 고려 대상이 아니고 State로만 평가하기 때문에 Case를 나누어 일반화된 공식이 있는 경우(2x2, 3x3 등) 식을 간단하게 처리할 예정이다. 2x2 와 3x3의 경우 determinant를 구해 빠르게 연산할 것이고 4x4부터 20x20까지는 일반화된 Gauss-Jordan방식을 사용하여 연산 할 것이다.

B. Gauss-Jordan Elimination

우선 가우스-조던 소거법은 가우스-소거법과 조던 소거법을 혼용하여 역행렬을 구하는 방법이다. 가우스 소거법은 행렬의 하 삼각행렬을 모두 0으로 만들고 대각행렬을 1로 만드는 방법이고 조던 소거법은 상삼각행렬을 모두 0으로 만들어 기존 행렬을 단위 행렬로 만듦으로써 기존 행렬을 단위행렬로 만들어 역행렬을 구하는 방법이다.

가우스-조던 소거법에는 3가지의 행 연산이 사용된다.

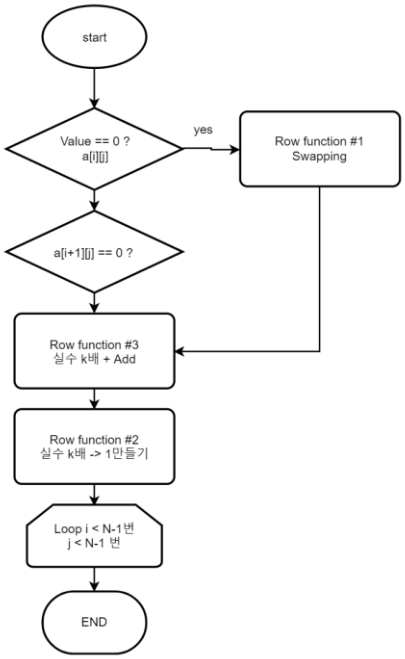
두 행을 R1, R2라고 하자. 첫 째로는 R1과 R2를 바꾸는 연산이 있다. 행을 바꾸는 연산은 기준이 되는 노드가 값이 0인 경우 특정 행과 swap하여 연산이 가능하도록 바꾸는 연산이다. 예를 들어 R1의 a11값이 0 이고 a21값이 0이 아니라면 0 인 값은

나눗셈이 불가능 하므로 아래 a21값과 swapping을 통해 가우스 소거가 가능한 형태로 바꾸는 연산이다.

두 번째 연산은 특정 행 R1에 실수 k값을 곱하는 연산이다. 우리는 기존의 행렬을 단위 행렬로 만들 필요가 있다. 따라서 a11의 값이 5인경우 이를 1/5배 하여 1로 만들어주어야 한다. 이럴 경우 5에 실수 1/5를 곱하여 1 즉 대각행렬을 1로 만들어 주는 작업을 할 때 사용할 것이다.

마지막으로는 특정 행 R1을 실수 k배 하여 R2에 더해주는 연산이다. 이 연산은 하삼각행렬 혹은 상삼각행렬을 모두 0으로 만들어 줄 때 사용된다 예를 들어 a11값이 5이고 a21값이 10인 경우 a11값에 실수 -2배를 하여 a21에 더해주는 연산을 진행한다. 이 경우 a21값이 0 이 된다. 물론 이 행동은 행 전체에 수행해 주어야 한다.

이 3가지 연산을 모두 각 function으로 구현하여 연산을 수행할 것이다. 3가지 function을 선택하는 algorithm은 다음과 같다.



기준 행의 값이 0인 경우를 먼저 비교하여 0이라면 실수 k배를 진행할 수 없기 때문에 가장 우선적으로 비교한 후, 상, 하 삼각행렬을 모두 0으로 만드는 row function을 2번째로 진행하고 마지막으로 본인을 1로 만들어 버리는 row function을 마지막으로 진행한다. 이를 정방행렬 N에 따라 N-1번 반복하여 전체 행렬에 대한 역행렬을 구할 수 있다. 이 때 Register의 여유가 있다면 Unrolling을 사용하여 State를 줄일 수 있다.

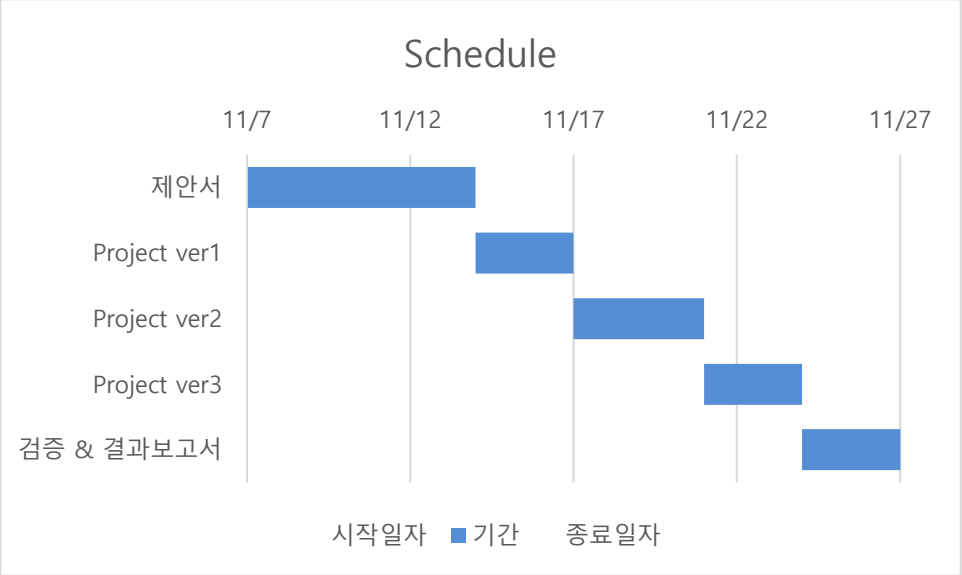
C. Floating Point Multiply & Division

본 프로젝트에서 가장 중점적으로 다루게 되는 Floating point의 곱셈 연산과 나눗셈

연산이다. 곱셈 연산의 순서는 다음과 같다. 덧셈 연산과 달리 Exponent를 조정하지 않는다. 그리고 Mantissa끼리 곱한다. Exponent끼리는 더한다. 소수 이하 자리는 아직 Specification에서 명시되어 있지 않아 버리는 쪽으로 한다. 이후 계산 결과를 Normalization하여 표현한다. 나눗셈도 Exponent를 조정하지 않는다. Mantissa끼리는 나눗셈을 실행하고 Exponent끼리는 뺄셈을 실행한다. 소수 이하 자리는 마찬가지로 버림을 한다. 오차범위는 아직 공지된 것이 없어 기다릴 예정이다. 이후 결과를 Normalization하여 값을 도출한다. 이 때 덧셈과 뺄셈은 기존 Floating Point과제로 수행한 내용을 약간 수정하여 function으로 사용할 것이고 곱셈은 Booth Multiplication을 이용해 구현할 예정이다.

덧셈과 뺄셈은 값을 더할 때 사용되고 곱셈은 앞서 말한 row function의 실수 k배 하는 경우에 사용하게 된다. 나눗셈은 실수 k배의 k값을 구하기 위한 용도로 사용된다. A11, A21값이 몇 배 차이 나는지 알기 위한 용도로 사용 할 수 있다.

3. 일정



4. 예상되는 문제점 & 검증 전략

기존에는 State를 검사하지 않아 코드 내용을 내가 원하는 대로 자유롭게 구현하며 성능을 크게 고려하지 않았다. 반면 프로젝트는 State가 작을수록 좋기 때문에 이를 최대한 최적화해야 한다. Loop Unrolling 과 Barrel Shifter의 응용을 통해 최대한 구현을 할 것이다.

기존에는 Register가 모자라서 LDM, STM으로 Memory Access를 억지로 해야 하는 경우가 없었다. 반면 행렬은 Size가 너무 커 13개의 user define Register에서 모든 연산을 진행할

수가 없다. 따라서 LDM, STM을 사용하여 연산중인 Register를 stack을 사용하여 작업해야 하는 경우가 생긴다. 이때 명확한 알고리즘이 없다면 서로 겹쳐 헷갈릴 수 있다. 직접 손으로 register 사용을 디자인하며 Register의 낭비 및 혼선 없이 구현하도록 한다.

Floating Point의 곱셈과 나눗셈의 연산 자체가 난이도가 높은 function으로 생각된다. 이를 구현하고 정확히 검증을 먼저 하여 실제 프로젝트에서 제대로 사용할 수 있도록 설계한다. 즉 Sub Function을 먼저 구현한 후 상위 함수를 구현하는 방식으로 진행 할 것이다. Assembly Code는 눈으로 보면 다 맞는 말 같다. 따라서 명확한 test set과 결과를 천천히 Debugging하며 따라가서 연산 결과가 맞는지 최종적으로 확인한다.

5. 참고문헌

[1] 부동소수점 산술연산

https://m.cafe.daum.net/kpucomjjang/D7M4/30?q=D_7_beATEAlUg0&

[2] 역행렬

https://www.youtube.com/watch?v=6EuwKnsNQSg

[3] 가우스-조던 소거법

https://www.youtube.com/watch?v=RgpjVOFzpdk