Lab1 Report

소프트웨어학과 201921085 곽수정

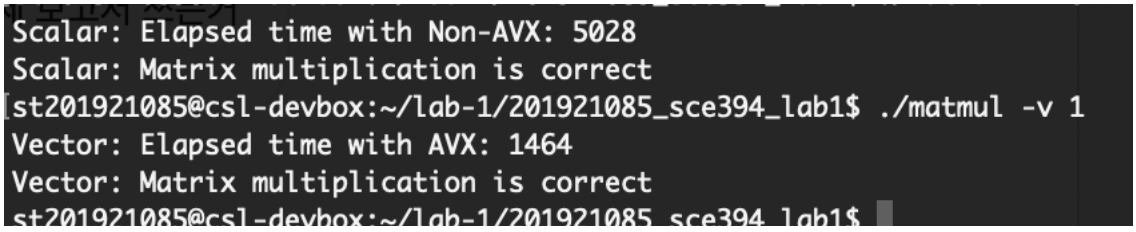
먼저 float 값을 갖는 4 x 4 행렬 두개를 만들기 위해 a와 b에 각각 malloc으로 동적할당 해주었습니다. 이에 맞춰 vector의 타입도 __m256으로 선언하여 사용하였습니다.

첫번째 for문에서는 4 x 4 행렬을 만들기 위해 aligned_alloc 해주었고, 실질적으로 값을 저장해서 전달해줄 목적으로 temp 와 temp1 또한 alligned_alloc 해주었습니다. temp와 temp1에 값을 넣어 temp 는 a 행렬에, temp1 은 b 행렬에 넣어주었습니다.

주석 Non-AVX에서부터는 Scalar version의 행렬 곱을 수행하게 됩니다. 원래 행렬 곱을 계산하는 방법처럼 셀하나씩 serrial 하게 계산할 수 있도록 하였습니다. 계산 결과를 result에 저장하여 answer와 비교하여 일치하는지 확인하고 시간 확인 코드를 작성하였습니다.

주석 AVX에서부터는 Vector version의 행렬 곱을 수행하게 됩니다. 함수 multiply_simple에서 계산하게 되는데 이 함수는 a 행렬의 하나의 행과 matrix(b 행렬과 동일)의 모든 행의 곱을 계산하여 결과 행렬의 하나의 행을 return 하도록 하였습니다. 그리고 a 행렬의 하나의 행에서 첫번째 인덱스부터 끝까지 broadcast 하여 matrix의 벡터와 곱하고 더하는 연산으로 더 빠르게 수행할 수 있도록 하였습니다.

하나의 함수 안에서 a의 모든 행에 대해 순서대로 broadcast하고 matrix와 곱해주어 결과를 얻는 방법도 있었지만 수행해본 결과 연산 속도가 만족할만큼 빠르지 않았습니다. 따라서 a의 1부터 4번째 행까지 벡터 연산이 이루어질 때, 연산 과정이 서로 영향을 끼치지 않으므로 함수 호출 형태로 구현하면 더 빠를 것이라고 생각하여 multiply_simple 을 4번 호출하여 결과를 얻도록 하였습니다. Vector로 수행한 연산 결과도 answer 와 일치하였으며 scalar, vector virsion의 속도 차이는 아래와 같습니다.



위 5028은 scalar 연산, 1464는 vector 연산의 실행 결과입니다. 평균적으로 vector 연산 방법이 scalar 보다 2~4배 빠르게 수행되었습니다. 이는 scalar 연산에서는 곱하기 연산이 64번 일어나며, 그만큼 더하는 연산도 많이 일어나는데, vector 연산에서는 16번 곱하기 연산이 일어나며, 더하는 연산도 그만큼 적게 일어나기 때문에 더 빠른 성능을 보였습니다.

m x n 행렬의 구현은 시간상 하지 못했지만 구현 방법을 생각해 보았을 때, a 행렬과 b 행렬이 곱해지는 경우, b의 행이 만약 길이 8을 넘게 되면 __m256 단위를 넘게 됩니다. 따라서 b 행렬의 열의 길이가 17인 경우를 예로 들면 _mm256_mul_ps 와 같은 함수로 연산이 필요할 때 __m256 데이터 타입 기준으로 벡터를 나눠서 연산해야합니다. 인덱스 0~7은 서로 같은 __m256 데이터에,

인덱스 8~15도 서로 같은 데이터에 담아 사용하고, 남은 마지막 인덱스 16도 __m256으로 선언하고 나머지 쓰이지 않는 칸은 0으로 초기화 합니다.

그러면 행 또는 열의 길이가 달라져도 벡터 연산이 가능하여 여전히 8개의 요소를 한번에 연산 가능하므로 빠른 성능을 가졌을 것으로 예상합니다. 시간상 문제로 m x n 행렬의 방법은 떠올리고 구현은 하지 못했지만 위와 같은 방법으로 구현한다면 벡터 연산이 가능할 것이라고 생각합니다.

- a matrix

1	2	3	4
11	12	13	14
21	22	23	24
31	32	33	34

- b matrix

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

- answer matrix

10	20	30	40
50	100	150	200
90	180	270	360
130	260	390	520