고급 소프트웨어 실습 10주차 보고서

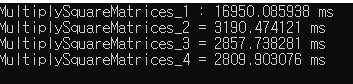
20171693 조병화

**실습**

실습 문제 1

**행렬 계산의 속도 향상을 위하여 (1) 자신이 적용한 방법과 (2) 어떠한 근거로 자신 이 적용한 방법이 더 효율적일지, 그리고 (3) 어떤 m 값에 대해 loop unrolling 방법이 가장 효과적이었는지를 요약하여 프로그램과 함께 제출할 것.**

1. 계산 속도의 향상을 위하여 수업시간에 배운 loop unrolling 방식을 적용해 코드를 작성하였다.
2. Loop 문을 반복할 때마다 항상 추가적으로 발생하는 비교 연산, 산술 연산이 존재하는데, 이를 unrolling을 통해 연산횟수, 비교 횟수를 줄여 프로그램을 더 효율적으로 작성할 수 있다. 따라서 loop unrolling 방법을 적용하였다.



1번째는 속도향상을 위한 어떠한 작업도 하지 않은 상태이고 ,2번째의 경우 메모리 접근의 응집성을 높혀 곱셈 속도를 향상시켰을 때이다. 3,4 번째의 경우 loop unrolling 방식을 적용하여 속도를 향상시킨 것이다. 위를 통해, 각 작업마다 유의미한 속도의 변화가 생겼음을 확인할 수 있었다.

실습 문제 2

이제 DEGREE는 10 정도, 그리고 N X는 1,048,576개 이상 크게 설정한 후, 위 두 함수의 수행 시간을 비교하라. 과연 여러분도 다음과 같은 정도의 차이를 보이는 실험 결과를 얻었는가?



다음은 horner’s rule을 통해 식을 변형시켜 계산한 결과와 naïve하게 계산한 결과를 비교한 것이다. Horner’s rule을 통해 곱셈 횟수가 줄어들어 유의미한 시간의 단축이 있었음을 확인할 수 있었다.

실습 문제 3

**이 식의 첫 25개 항을 float 타입의 연산을 사용하여 e −8.3 ≈ 1+ (−8.3)+ (−8.3) 2 2 +···+ (−8.3) 24 24! 을 계산해보자. 정확한 값은 2.485168 × 10−4인데, 여러분의 계산은 얼마나 정 확하게 이 값을 구했는가? 만약 적지 않은 오차가 있다면 이러한 결과가 나온 이유는 무엇일까? (iii) 그러한 문제를 극복하려면 e −8.3 값을 어떻게 구할지 더 좋은 방법을 제안하고 실험한 후 그 결과를 기술하라. 과연 여러분도 개선된 방법을 사용한 후 다음과 같은 정도의 차이를 보이는 실험 결과를 얻었는가?**



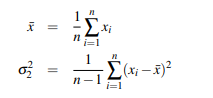
실험의 결과는 다음과 같다.

위의 결과는 순서대로 식을 이용한 경우의 값, 오차를 극복하기 위한 방식을 사용한 값, 원래 나와야 할 값이다. 위의 결과를 통해 주어진 식을 사용하면 적지않은 오차가 발생함을 확인할 수 있다. 따라서 이를 해결하기위해 지수가 음수인 경우와 양수인 경우를 나눠서 다른 계산 방식을 사용하였다. 양수인 경우는 그대로 계산하나, 음수인 경우는 -를 붙여 양수로 만들어 준 뒤, 계산을 완료한 후에 그 역수를 취해주는 방식을 사용하였다. 즉, e^8.3을 계산한 후 그의 역수를 취하여 e^-8.3으로 만들어 준 것이다. 그결과 오차가 매우 크게 감소하였음을 확인할 수 있었다.

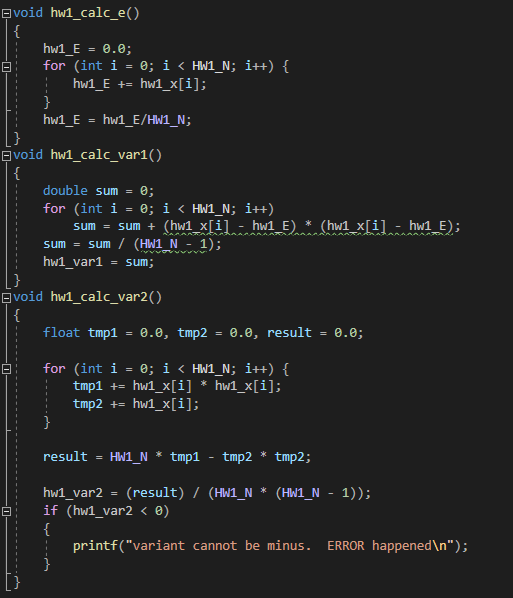
**과제**

과제 1번

**(i) x¯, σ2, 그리고 σ1을 계산해주는 함수를 작성하라. (float 타입의 연산을 사용하되, 만약 실험중 double 타입의 연산이 필요하면 이유를 기술하고 사용 할 것).**



다음의 식을 코드로 구현하면 다음과 같다.



순서대로  **x¯, σ2, 그리고 σ1** 를 구해주는 함수이다.

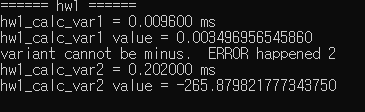
이 때 hw1\_calc\_var2에서 분산의 값이 음수가 나오는지 여부를 확인해보기 위해 리턴 값이 음수일 경우 print문을 출력해 보았다. 이 때 N의 값이 커짐에 따라 float의 범위를 벗어나 연산에 오류가 생기는 경우들도 있었기 때문에 double 타입으로의 변환이 필요할 수 있다.

(ii) **두 분산 값 계산 방법의 결과가 상당히 차이가 나게 해주는 샘플 데이터를 생성한 후, 계산 결과를 비교분석하라. 분산을 어떻게 계산한 것이 더 정확한 것으로 판단되는가?**

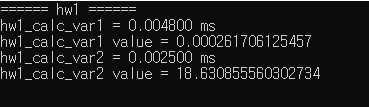
비슷한 숫자끼리의 뺄셈을 할 때 부동 소수점 연산의 오차가 발생하게 된다. 첫번째 식의 경우 각 값에서 평균을 뺀 값을 제곱을 하여 분산을 구하는데, 이 때 평균과 해당 값의 차이가 크지 않은 경우 부동 소수점 연산에서의 오류가 발생할 수 있다. 하지만 두번째 식의 경우 제곱끼리의 차를 구하기 때문에 첫번째 식을 통해 분산을 구하는 경우보다 계산과정에서 부동 소수점 연산에서의 오차가 발생할 일이 더 적다. 따라서 두번째 식이 더 정확할 것으로 판단된다.

다만, 2번 식의 경우 계산과정에서 분산의 값이 음수로 나올 수도 있다는 문제가 있다. 이를 확인해보기 위하여 각 분산 계산 함수의 맨 마지막에 분산이 음수일 경우에 출력문을 하나씩 넣었는데, 값은 N값으로 연산했을 때 2번식으로 구한 분산은 음수 값이 나왔음을 확인할 수 있었다. 따라서 2번 식의 경우에는 값이 음수로 튀어 위험할 수 있다.

이를 통해, double범위 내에서 정확한 연산을 할 수 있는 입력이 주어졌다는 가정하에, 첫번째 식으로 연산하였을 때보다 두번째로 연산하였을 경우 부동 소수점 연산으로 인한 오류가 더 적어 두번째가 더 정확하다고 할 수 있겠다.



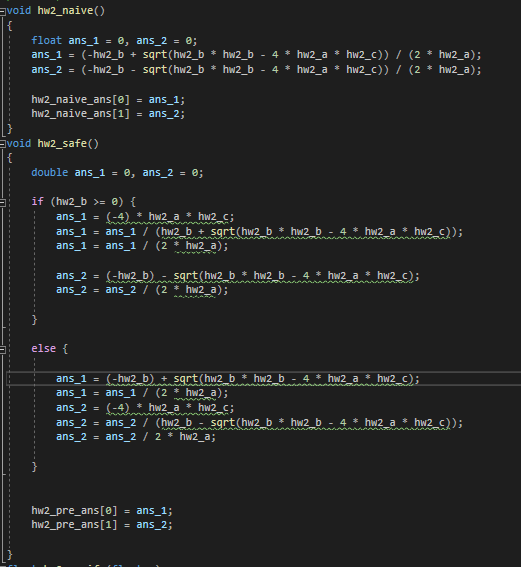
**(iii) 충분히 큰 n에 대하여 두 방법 중 어떤 방법이 더 빠르게 분산 값을 계산하**는가?



모든 경우들에 대하여 1번식이 더 빠르게 분산값을 찾아내는 것을 확인할 수 있었다.

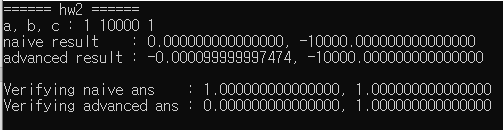
과제 2번

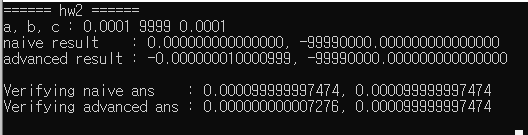
**(i) 콘솔 윈도우에서 임의의 a, b, 그리고 c 값을 읽어들여 중학교에서 배운 근의 공식을 사용하여 두 실근을 구하여 출력하는 프로그램을 작성하라 (편의상 두 개의 실근이 존 재하는 경우만 고려).**

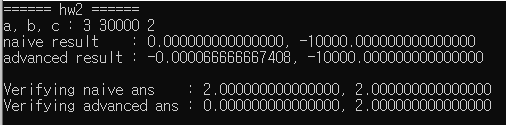


**Naïve함수를 보면 근의 공식을 이용하여 근을 구했음을 확인할 수 있었다.**

**(ii) 위 프로그램이 심각한 문제를 야기하는 상황을 세 가지 발생시켜라. 즉 그런 문제를 일으킬 a, b, 그리고 c값을 적절히 설정한 후, 위에서 구한 두 근을 다시 f(x)에 대입하여 0이 나오는지 확임함으로서 심각한 문제가 발생하였다는 것을 증명하라.**



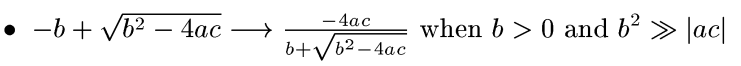




Naïve 한 방식으로 구한 근 보다 advanced 하게 구한 근이 더 정확하게 값을 구했음을 확인할 수 있었다. 다시 대입한 값이 0이 나오지 않기도 하였다.

**(iii) 다음 그러한 문제를 완화시킬 수 있는 방법을 사용하여 위의 2차 방정식을 풀어주는 함수를 새롭게 구현한 후, 위의 문제와 동일한 과정를 거쳐 (즉 자신이 구한 근에 대해 f(x) 함수 값을 구하여), 위에서 심각한 문제를 야기한 세 경우 각각에 대해 자신의 두 번째 함수가 안정적으로 근을 구했음을 밝혀라**

두번째 safe 의 경우



위 식을 적용하여 더 정확하게 연산될 수 있도록 하였다. 유리화를 통해 뺄셈에 의해 일어날 수 있는 비슷한 수의 뺄셈 문제를 해결할 수 있음을 알 수 있었다.

과제 3

**자료 조사를 통하여 일반적으로 널리 사용되고 있는 원시 언어 수준 (Source Code Level) 의 코드 최적화 기법에 대하여 공부를 한 후, (ii) Loop Unrolling과 Loop Fusion 기법을 제외한 나머지 기법 중 5개를 선정하여, (iii) 각 방법 별로 해당 방법의 효과를 최대한 보일 수 있는 C/C++ 코드를 작성한 후, (iv) 해당 기법 적용 전과 적용 후의 시간을 비교한 내용을 보고서에 명기한 후 그러한 결 과가 나온 이유를 상세히 설명하라.**

Loop Unrolling 과 Loop Fusion 을 제외한 나머지 코드 최적화 기법으로 내가 선정한 방법은 다음과 같다.

1. Loop Invariant code Motion
2. Strength Reduction
3. Binary breakdown
4. Local Global
5. Loop Inversion

각각의 방식에 대하여 설명해 보겠다.

1. Loop invariant code motion 방식

반복문 내에서 항상 반복되는 계산들을 루프 밖으로 빼내어 상수값처럼 사용을 하여 계산 횟수를 감소시킨다.

1. Strength reduction 방식

고비용의 연산자들을 저비용의 연산자를 이용하여 연산되도록 코드를 최적화 하는 방법이다. 예를 들어 나누기, 곱하기 연산을 비트연산으로 대체하는 것이다. \*64와 /64를 비트연산으로 대체하여 << 6, >>6으로 사용하여 수행시간을 감소시켰다.

1. Binary breakdown

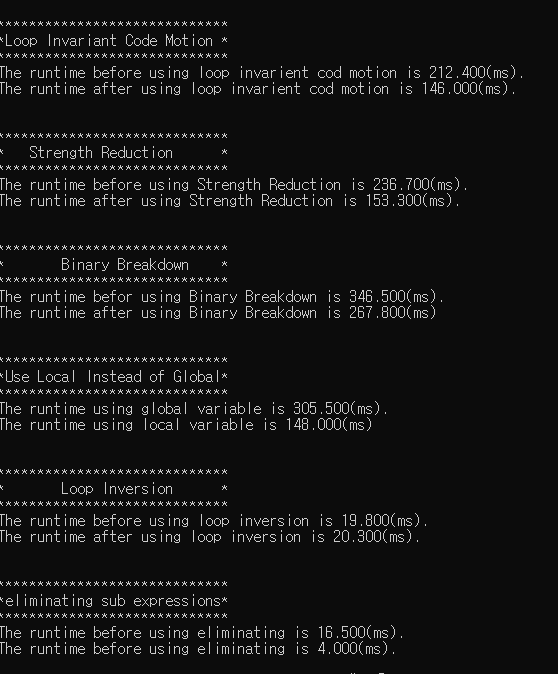
if와 else if를 매우 여러 개 사용하는 경우가 생긴다. 예를 들면 0~7까지의 수를 확인하기 위해 if문 하나, else if문 7개를 쓰게 되는데, 이 경우를 2개로 나눈 뒤 0~3과 4~7의 if-else문으로 나누고 그 안에서 각각의 숫자를 확인하기 위한 if- else if문을 사용하게 된다면, 수행 시간을 많이 줄일 수 있다.

1. Local 변수 대신 global 사용

global 변수가 CPU의 register에 저장될 수 없기 때문에 생기는 연산 속도가 매우 느린데 이를 보완하기 위해 사용하는 방식이다. local 변수는 레지스터를 통해 저장될 수 있기 때문에 연산속도가 global 변수보다 더 빠르다. 연산을 수행 할 때에 global 변수를 사용하는 것보다 local 변수를 통해 레지스터를 활용하는 연산을 수행하면 수행시간을 많이 줄일 수 있을 것이다.

1. Loop inversion

loop inversion은 loop transformation으로 while 반복문을 if문 안의 do while 반목문으로 바꿔 수행하는 것이다. 이를 이용하면 condition을 check 할 때 jump를 하지 않아도 되기 때문에 jump의 사용이 이전보다 적어진다. 이는 instruction의 pipelining을 효과적으로 할 수 있도록 하여 주기 때문에 수행시간이 단축된다.



다음은 각 코드의 실행 결과이다. 시간의 유의미하게 감소함을 확인할 수 있었다.