**고급소프트웨어실습1 10주차 과제**

컴퓨터공학 20172141 김미소

**[실습 1번]**

**(i) 다음과 같은 프로토타입을 가지는 함수를 작성하라.**

**MultiplySquareMatrices\_1(double \*pDestMatrix, double \*pLeftMatrix, double \*pRightMatrix, int MatSize);**

텍스트, 스크린샷, 실내, 화면이(가) 표시된 사진

자동 생성된 설명

2차원 배열을 곱하는 방법을 1차원 배열 버전으로 바꾸어 작성한 것이다. 2차원 배열이었다면 pDestMatrix[i][j] = pLeftMatrix[i][k]\*pRightMatrix[k][j]로 계산되었을 것이므로 이를 1차원 배열 버전으로 바꾼 것이다.

**(ii) 이제 메모리 접근의 응집성을 높여 행렬 간의 곱셈 계산의 속도를 향상시켜주는 함수MultiplySquareMatrices\_2(\*)를 작성하라. 이 함수의 인자들은 앞의 1번 함수와 동일하며, 함수 수행 후 A 행렬과 B 행렬의 내용은 함수 수행 직전과 동일해야 한다.**

텍스트, 스크린샷, 모니터, 화면이(가) 표시된 사진

자동 생성된 설명

메모리를 접근하는 패턴이 성능에 지대한 영향을 미치기 때문에 메모리 응집성을 고려하여 순서대로 하나씩 메모리 접근이 일어날 수 있게 해야 한다. 제일 좋은 방법은 pRightMatrix를 전치시키는 것이다. 그러나 함수 수행 후 A 행렬과 B 행렬의 내용이 같아야 하므로 별도의 배열을 하나 선언하여 사용하였다.

**(iii) 위의 MultiplySquareMatrices\_2(\*) 함수를 기반으로 하여 loop unrolling 기법을 적용하여 보자. 이때 가장 안쪽의 for 문을 m개 단위로 풀어 어떤 m에 대해 가장 좋은 성능을 내는지 시간을 측정하여 보자.**

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

4개 단위와 8개 단위로 풀어 loop unrolling을 수행하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

각 방법의 수행결과는 위와 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 방법은 4번째 방법에서 16개에 대하여 loop unrolling을 한 것인데 8개 단위로 loop unrolling 한 것보다 더 오래 걸렸다. 따라서 m은 8이 가장 적절하다.

**[실습 2번]**

**(i) DEGREE+1개의 DEGREE차 다항식의 계수 ai를 랜덤하게 생성하여 double 타입의 배열 변수 a[i]에 저장하라(i = 0,1, · · · , DEGREE). 다음 N X개 만큼 x 변수 값을 랜덤하게 생성하여(0과 1 사이의 값으로), i번째 값 xi를 double 타입의 배열 값 x[i]에 저장하라(i = 0,1, · · ·, N X-1).**

텍스트이(가) 표시된 사진

자동 생성된 설명

**(ii) 다음 C/C++ 언어의 수학 라이브러리가 제공하는 pow(\*, \*) 함수를 사용하여 각각의 x[i]에 대해 위에서 설정한 DEGREE차 다항식을 계산하여 double 타입의 배열 값 y[i]에 저장해주는 다음과 같은 프로토타입의 함수를 작성하라(각 인자의 의미는 분명함).**

**void Eval\_Poly\_Naive(double y[], double x[], int n x, double a[], int deg);**

텍스트이(가) 표시된 사진

자동 생성된 설명 

오른쪽 식을 구현한 함수이다.

**(iii) 다음 Horner’s rule을 사용하여 위의 함수와 동일한 작업을 해주는 다음과 같은 함수를**

**작성하라.**

**void Eval\_Poly\_Horner(double y[], double x[], int n x, double a[], int deg);**

텍스트이(가) 표시된 사진

자동 생성된 설명



위 식을 구현한 함수이다.

**(iv) 다음 적절한 함수를 작성하여 위에서 각각 두 함수를 통하여 구한 y[i] 값이 같음을 보여라.**

텍스트이(가) 표시된 사진

자동 생성된 설명

이 함수를 통해 같은 y[i]를 가졌는지 확인할 수 있고 실행 결과

텍스트이(가) 표시된 사진

자동 생성된 설명

모든 값이 같았음을 확인할 수 있었다.

**(v) 이제 DEGREE는 10 정도, 그리고 N\_X는 1,048,576개 이상 크게 설정한 후, 위 두 함수의 수행 시간을 비교하라.**

텍스트이(가) 표시된 사진

자동 생성된 설명

수행 결과 두 방법의 수행시간은 상당히 많이 차이 났다.

**[실습 3번]**

**(i) Taylor series에 기반을 둔 다음 식은 널리 알려진 무한 급수의 합이다.**

****

**이 식에서 주어진 x에 대하여 앞에서 n+1개의 항만 사용하여, ex 값을 근사적으로 계산해주는 함수를 작성하라. 이 함수는 double 타입의 x와 int 타입의 n값을 입력 인자로 받아들여, double 타입의 연산을 수행하여 위의 식의 항을 순서대로 더하여 그 결과를 double 타입으로 리턴해주어야 한다. 이때 위의 급수의 합을 Horner’s method를 사용하여 계산하라.**

텍스트이(가) 표시된 사진

자동 생성된 설명

**(ii) 이 식의 첫 25개 항을 float 타입의 연산을 사용하여 e−8.3을 계산해보자. 정확한 값은 2.485168×10−4인데, 여러분의 계산은 얼마나 정확하게 이 값을 구했는가? 만약 적지 않은 오차가 있다면 이러한 결과가 나온 이유는 무엇일까?**



Float 타입 계산 결과 정확한 값과 많은 차이가 있는 값이 도출되었다. Float 타입 부동 소수점 표현의 한계가 있기 때문에 계산 과정에서 오차가 생기기 때문이다.

**(iii) 그러한 문제를 극복하려면 e−8.3 값을 어떻게 구할지 더 좋은 방법을 제안하고 실험한 후 그 결과를 기술하라.**

Float 타입을 double 타입으로 바꾸어 오차를 줄이고 taylor series는 근사적으로 계산하는 방법이기 때문에 N을 더 늘려 정확도를 높일 수 있다. 따라서 N을 50으로 늘려 정확도를 높인 결과

텍스트이(가) 표시된 사진

자동 생성된 설명

정확한 값이 도출되었음을 확인하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**[숙제 1번]**

**(i) 우선 적절히 큰 n을 선택하여 난수를 발생시켜 적절한 구간의 샘플 데이터 xi (i=1,2, · · · ,n)를 생성하라. 다음 각각 , σ2, 그리고 σ1을 계산해주는 함수를 작성하라 (float 타입의 연산을 사용하되, 만약 실험 중 double 타입의 연산이 필요하면 이유를 기술하고 사용할 것).**

텍스트이(가) 표시된 사진

자동 생성된 설명

샘플 데이터 xi를 생성하는 코드이다. Fixed가 1일 때 0.0001정도 차이나는 비슷한 숫자들을 샘플 데이터로 생성하고 fixed가 0인 경우 난수를 생성한다.

텍스트이(가) 표시된 사진

자동 생성된 설명 

를 계산하는 함수이다. 샘플 데이터의 값을 모두 더하고 샘플 데이터 개수로 나누어 준다.

텍스트이(가) 표시된 사진

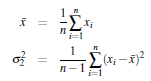
자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

오른쪽 식의 방법으로 분산 값을 구하는 함수이다.

텍스트이(가) 표시된 사진

자동 생성된 설명



평균을 이용하여 분산을 구하는 방법을 구현한 함수이다.

**(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? (즉 수학적으로 동일한 두 식이 컴퓨터상에서는 얼마나 다를 수 있는지를 스스로 파악하는 것이 목적임. 일반적으로 후자의 방법이 더 위험하다고 알려져 있음. 참고로 분산 값은 음수일 수 없음)**

- Fixed = 1, 즉 비슷한 숫자로 이루어진 샘플 데이터에 대하여

텍스트이(가) 표시된 사진

자동 생성된 설명

10만개의 데이터에 대하여 첫번째 방법과 두번째 방법을 통해 얻은 결과 값은 차이가 굉장히 컸다.

- Fixed = 0, 즉 무작위 난수로 이루어진 샘플 데이터에 대하여

텍스트이(가) 표시된 사진

자동 생성된 설명

10만개의 데이터에 대하여 첫번째 방법과 두번째 방법을 통해 얻은 결과 값의 차이가 비슷한 숫자로 이루어진 데이터에 비하여 적었다.

이러한 결과가 나온 것은 이 연산이 float 타입의 부동 소수점 연산을 기반으로 하여 오차가 생겨나게 되고, 비슷한 숫자들로 이루어진 데이터를 이용하면 ‘비슷한 숫자끼리의 뺄셈’이 발생하여 오차가 생겨나기 때문이다. 즉 비슷한 숫자들로 이루어진 데이터보다 일반적인 무작위 난수로 이루어진 데이터가 계산 결과의 오차가 더 적다.

실험 결과를 보았을 때 분산을 더 정확하게 계산한 것은 두번째(σ2) 방법이다. 그 이유를 생각해보면 첫번째(σ1) 방법에서  이 부분에서 ‘비슷한 숫자끼리의 뺄셈’이 발생할 수 있기 때문이다. Init\_hw1의 tmp를 float 타입에서 double 타입으로 바꾸었을 때 첫번째(σ1) 방법에서 분산이 음수가 나오기도 했다. 따라서 두번째(σ2) 방법이 좀 더 정확한 방법이다.

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

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

위에서 사용했던 결과를 보면 n = 100000인 경우에, 두번째(σ2) 방법이 더 빠르게 분산 값을 계산하였다.

**[숙제 2번]**

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

텍스트이(가) 표시된 사진

자동 생성된 설명

근의 공식을 이용하여 실근을 구하는 함수이다.

**(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 확인함으로써 심각한 문제가 발생하였다는 것을 증명하라.**

B에 비하여 a, c가 아주 작은 숫자인 경우 는 b와 비슷한 숫자가 될 것이고

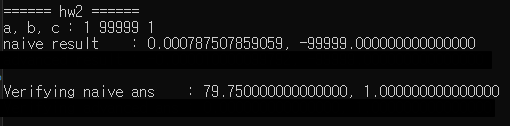
는 결국 ‘비슷한 숫자끼리의 뺄셈’이 될 것이다. 즉 이 과정에서 오차가 발생하여 근을 제대로 구하지 못하는 문제가 발생할 수 있다.

**- a = 0.1, b = 9999, c = 0.1**

텍스트이(가) 표시된 사진

자동 생성된 설명

**- a = 1, b = 99999, c= 1**



**- a = 1, b = -9999, c = 1**

텍스트이(가) 표시된 사진

자동 생성된 설명

세 결과 모두 0이 나오지 않는다.

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

텍스트, 모니터, 화면, 실내이(가) 표시된 사진

자동 생성된 설명



b가 양수인 경우, 을 유리화하여 사용하면 ‘비슷한 숫자끼리의 뺄셈’을 피할 수 있다. b가 음수인 경우에는 에서 ‘비슷한 숫자끼리의 뺄셈’이 발생하므로 이를 유리화하여 사용하면 된다.

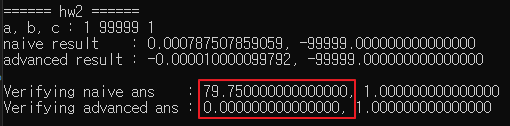
**- a = 0.1, b = 9999, c = 0.1**

텍스트이(가) 표시된 사진

자동 생성된 설명

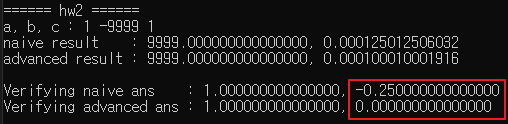
빨간색 네모 친 부분이 개선된 방법으로 계산한 근에 대한 verifying 부분인데 정상적으로 0이 나온 것을 확인할 수 있다.

**- a = 1, b = 99999, c= 1**



빨간색 네모 친 부분이 개선된 방법으로 계산한 근에 대한 verifying 부분인데 정상적으로 0이 나온 것을 확인할 수 있다.

**- a = 1, b = -9999, c = 1**



빨간색 네모 친 부분이 개선된 방법으로 계산한 근에 대한 verifying 부분인데 정상적으로 0이 나온 것을 확인할 수 있다.

개선된 부분을 제외하고 Verifying 부분이 0이 나오지 않는 것은 변수들이 float 타입이기 때문에 발생하는 문제인 것 같다.

**[숙제 3번]**

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

**(1) Loop collapsing**

텍스트이(가) 표시된 사진

자동 생성된 설명

Nested Loop를 single loop로 바꾸는 기법이다. 이 경우에는 2차원 배열에 접근하는 방식을 2차원 loop에서 1차원 loop로 변환한 것이다.

<Debug mode>

텍스트이(가) 표시된 사진

자동 생성된 설명

여기서는 연산을 하지 않기 때문에 수행 결과에 대한 비교는 하지 않았고 최적화 하지 않은 코드보다 최적화 한 코드가 훨씬 빨리 수행되는 것을 확인할 수 있었다.

**(2) Code motion**

텍스트이(가) 표시된 사진

자동 생성된 설명

Loop를 돌 때마다 반복되는 값이 바뀌지 않는 연산을 반복문 밖으로 빼내서 연산량을 줄이는 방법이다. 값이 바뀌지 않는 x = y+z와 x\*x연산이 loop 안에서 계속 반복되므로 반복문 밖으로 빼준다.

<Debug mode>

텍스트이(가) 표시된 사진

자동 생성된 설명

최적화된 코드의 수행시간이 최적화 전 코드의 수행시간보다 빨랐고 결과 값도 같은 것을 확인할 수 있었다.

**(3) Dead code elimination**

텍스트이(가) 표시된 사진

자동 생성된 설명

사용되지 않는 변수나 코드를 없애 최적화하는 방법이다. If(0)은 항상 거짓이므로 실행되지 않는 dead code이기 때문에 지우고 사용되지 않는 dead variable 또한 지워서 코드를 최적화하였다.

<Debug mode>

텍스트이(가) 표시된 사진

자동 생성된 설명

최적화되기 전 코드보다 최적화한 코드의 수행시간이 훨씬 짧음을 확인할 수 있었다. 계산 결과를 비교할 것이 없었기 때문에 계산 결과에 대한 확인은 하지 않았다.

**(4) Induction variable & Strength reduction**

텍스트, 모니터, 스크린샷이(가) 표시된 사진

자동 생성된 설명

곱하기 연산은 더하기 연산에 비해 연산 cost가 크므로 곱하기를 더하기로 바꾸어 최적화하는 방법이다.

<Debug mode>

텍스트이(가) 표시된 사진

자동 생성된 설명

수행 결과, 최적화된 코드의 수행시간이 원래 코드의 수행시간보다 짧아진 것을 확인할 수 있었고 결과값이 다르게 나온 것을 확인할 수 있었다. 그 이유는 float 타입의 부동 소수점이 표현할 수 있는 한계가 있기 때문에 곱셈을 한 것보다 덧셈을 통해 누적되는 오차가 더 크기 때문에 결과값이 다르게 나온다.

**(5) Integer divide optimization**

텍스트이(가) 표시된 사진

자동 생성된 설명

나누기 연산은 shift 연산에 비해 속도가 느리다. 이 방법은 따라서 나누기 연산을 shift 연산으로 바꾸어 속도 향상을 할 수 있도록 한다.

<Debug mode>

텍스트이(가) 표시된 사진

자동 생성된 설명

수행 결과, 최적화된 코드의 수행시간이 원래 코드의 수행시간보다 짧아진 것을 확인할 수 있었고 결과값이 다르게 나온 것을 확인할 수 있었다. 그리고 계산 결과가 같게 나온 것을 확인할 수 있었다.

**- 각 방법의 release mode 수행 결과**

텍스트이(가) 표시된 사진

자동 생성된 설명Release mode에서는 디버깅 정보를 삽입하지 않기 때문에 코드를 최적화시키고 파일의 크기가 줄어들게 된다. 따라서 수행속도가 상당히 빨라진다. Debug mode에 비하여 수행시간이 상당히 빨라진 것을 확인할 수 있었다.

1) Loop collapsing

이 방법에 대하여 수행시간이 정말 미세하게 최적화된 코드의 수행시간이 더 빨랐다.

2) Code motion

이 방법 또한 수행시간이 정말 미세하게 최적화된 코드의 수행시간이 더 빨랐다.

3) Dead code elimination

이 방법은 최적화 전후의 수행시간 차이는 나지 않았다.

4) Induction variable & Strength reduction

이 방법은 최적화를 하고 난 후 수행시간이 더 길어지는 결과를 확인할 수 있었다.

5) Integer divide optimization

이 방법은 최적화된 코드의 수행시간이 더 빠른 것을 확인할 수 있었다.

1번과 3번의 방법은 0에 정말 가까운 시간이 걸리는데, 수행할 때마다 최적화 전후 수행시간이 다르기 때문에 정확하게 어느 것이 더 빠른지 판단하기는 힘들었다.