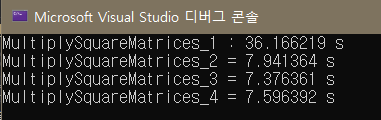
**고급소프트웨어실습1 10주차 과제**

**코드 최적화기법/부동소수점 연산에 대한 소개**

**20171672 이정원**

**실습 문제 1**

**실행 결과**



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

* 2번을 위해서는 C = A \* B를 취할 때 B를 전치행렬로 만드는 방법을 사용하였다. pMatD를 동적할당한 뒤 pRightMatrix의 전치행렬을 할당하였다. 1번과 비교했을 때 시간이 5분의 1로 줄어든 것을 확인할 수 있었다. 3번과 4번은 Loop Unrolling 기법을 적용하였다. Loop Unrolling 기법이란 반복하는 수를 줄여 for loop의 비교 연산이 줄게 되어 비용을 절감하고 속도를 높이는 기법이다. 3번은 i를 4씩, 4번은 i를 16씩 증가하였다. m이 클수록 더 효과적일 줄 알았으나 오히려 시간이 증가하는 경향을 보였다. 실험 결과 m이 4일 때 가장 효과적이었다.

**실습 문제 2**

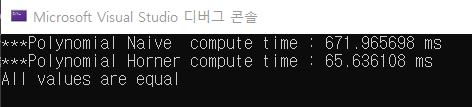
이 문제는 다항식의 값을 효율적으로 계산하는데 있어 적용할 수 있는 Horner’s rule에 관한 문제이다. Horner’s rule이란, 예를 들어, 다음과 같은 5차 다항식을



다음과 같이 계산하는 방식을 말한다.



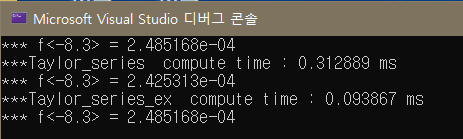
**실험 결과**



* Horner’s Rule을 적용함으로써 곱셈의 수를 줄여 계산량울 줄이게 돼 속도를 높일 수 있었다. 따라서 곱셈과 덧셈이 반복되는 다항식과 같은 계산식을 효율적으로 계산할 수 있는 것이다. 위 실험 결과와 같이 시간이 10배 이상 줄어든 것을 확인할 수 있다.

**실습 문제 3**

Taylor\_series 함수에서는 double 타입의 연산으로 기본 구현, Taylor\_series\_ex 함수에서는 float 타입의 연산으로 Horner’s method을 적용하여 구현하였다.

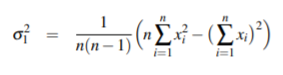


* 실험 결과 기본 구현의 결과가 더 정확한 값이 나왔으나 속도는 느리게, Horner’s Method을 적용한 구현의 결과는 오차가 더 컸지만 속도는 빠르게 나온 것을 확인할 수 있었다. 오차가 생긴 이유는 부동 소수점 연산 수행 시 비슷한 숫자끼리의 뺄셈으로 인해 생긴 것으로 보인다. 또한 double 타입 연산과 float 타입 연산의 차이는 해당 타입이 다루는 메모리 크기가 다른 것에서 비롯되는데, double 타입의 메모리 크기는 8로 속도가 느리고 비용이 크지만 정밀도가 높고, float 타입의 메모리 크기는 4로 정밀도가 낮지만 비용이 작고 속도가 빠른 것이다. 0과 1 사이에는 실수가 무한히 존재하지만 컴퓨터에 의해 유한 개로 표현하려다 보니 생긴 문제를 확인할 수 있었다.

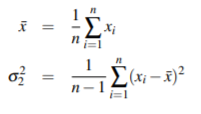
**숙제 1**

**(i) 우선 적절히 큰 n을 선택하여 난수를 발생시켜 적절한 구간의 샘플 데이터 xi (i = 1,2,···,n)를 생성하라. 다음 각각 x¯, σ2, 그리고 σ1을 계산해주는 함수를 작성하라.**

* hw1\_calc\_var1()에는 다음 식을 사용하였다. (식 1)



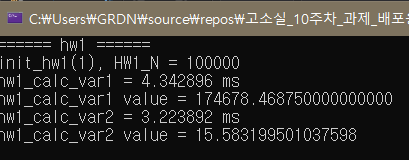
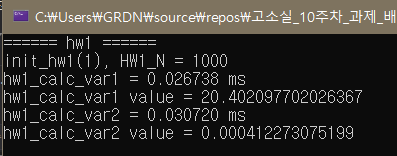
* hw1\_calc\_e()와 hw1\_calc\_var2()에는 다음 식을 사용하였다. (식 2)



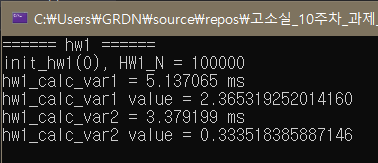
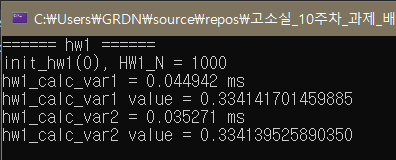
* 샘플 데이터 xi, 즉 hw1\_x에는 init\_hw1에 매개변수 1을 넘길 경우 서로 0.0001 정도밖에 차이가 나지 않는 비슷한 난수들을 생성하고, 0을 넘길 경우 일반적인 난수들을 생성한다.

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

* init\_hw(1): 서로 비슷한 값을 가진 난수들일 경우



* init\_hw1(0): 일반적인 랜덤값을 가진 난수들일 경우



* 서로 비슷한 값들을 가진 난수일 경우 두 식의 결과값이 모두 상당한 차이를 보였고, N이 커질수록 또한 작았을 때보다 더 큰 차이를 보였다. 일반적인 랜덤값을 가진 난수일 경우 두 식의 결과가 전자보다 큰 차이를 보이지는 않았고, N이 작을수록 서로 값이 비슷해지는 것을 보아 N이 작을수록 높은 정확성을 보였다. 이 계산은 부동소수점 연산을 기반으로 하고 있기 때문에 비슷한 숫자 간의 뺄셈을 하면 할수록 정확도는 점점 더 낮아질 확률이 높아진다. 따라서 서로 비슷한 값들을 가진 난수일 경우 계산결과의 오차가 더 크게 나타났다.
* 식 2에서는 난수와 평균값의 차이가 거의 없다면 (비슷한 숫자 간의 뺄셈) 큰 오차가 생길 것이기 때문에 더 위험하다. 따라서 식 1이 더 정확한 연산을 할 것으로 판단된다.

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

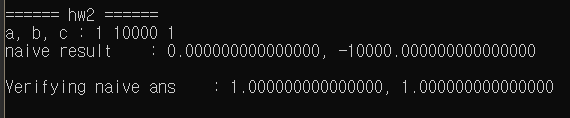
* 정확한 계산을 할수록 시간이 더 걸리기 때문에 식 2가 더 빠르게 분산값을 계산한다. 실험 결과(HW1\_N = 100000)에서도 그렇게 나타났다.

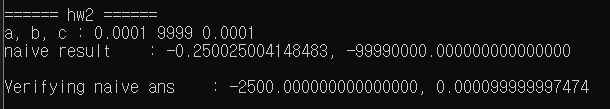
**숙제 2**

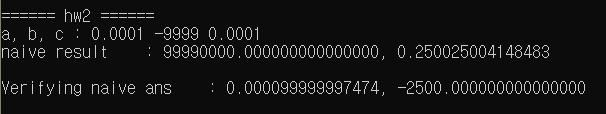
**(i) 콘솔 윈도우에서 임의의 a, b, 그리고 c 값을 읽어들여 중학교에서 배운 근의 공식을 사용하여 두 실근을 구하여 출력하는 프로그램을 작성하라 (편의상 두 개의 실근이 존재하는 경우만 고려).**

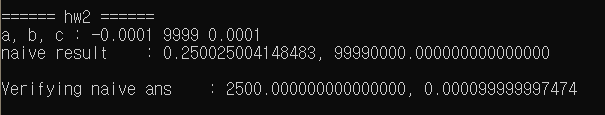
**(ii) 위 프로그램이 심각한 문제를 야기하는 상황을 세 가지 발생시켜라. 즉 그런 문제를 일으킬 a, b, 그리고 c값을 적절히 설정한 후, 위에서 구한 두 근을 다시 f(x)에 대입하여 0이 나오는지 확임함으로써 심각한 문제가 발생하였다는 것을 증명하라.**

* 이차방정식의 근을 구할 때 근의 공식을 이용할 때, a와 c가 b보다 상대적으로 많이 작은 값이면 비슷한 숫자끼리의 뺄셈을 하게 된다. 따라서 수치적인 오차 (-b + root(b\*b – 4ac))가 발생하게 된다.







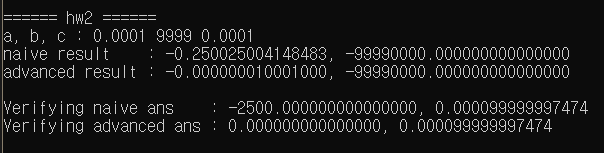


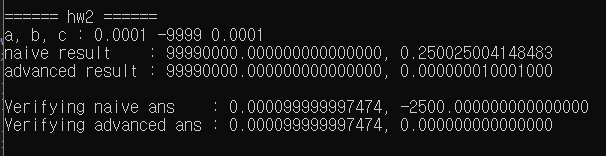
* 위 실험 결과와 같이 함수의 근인지 판별할 때 0이 아닌 것을 확인할 수 있었다.

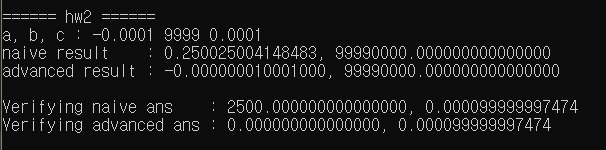
**(iii) 다음 그러한 문제를 완화시킬 수 있는 방법을 사용하여 위의 2차 방정식을 풀어주는 함수를 새롭게 구현한 후, 위의 문제와 동일한 과정를 거쳐 (즉 자신이 구한 근에 대해 f(x) 함수 값을 구하여), 위에서 심각한 문제를 야기한 세 경우 각각에 대해 자신의 두 번째 함수가 안정적으로 근을 구했음을 밝혀라.**



* b가 0보다 클 경우와 작을 경우로 나누어 비슷한 숫자 간의 뺄셈을 피하기 위해 근의 공식을 변형한다. b가 0보다 클 경우, hw2\_pre\_ans[0]를 구할 때 분모와 분자를 바꾸고 분자에 마이너스를 취하고, hw2\_pre\_ans[1]을 구할 때는 아무것도 바꾸지 않는다. b가 0보다 작을 경우 반대로 진행한다. 실험 결과는 다음과 같다.







* Verifying advanced ans에서 근이 0이 나오는 것을 통해 안정적으로 근을 구했음을 확인할 수 있었다.

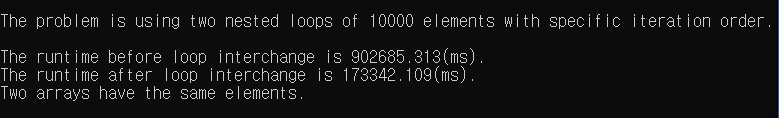
**숙제 3**

TWO\_23 = 10000

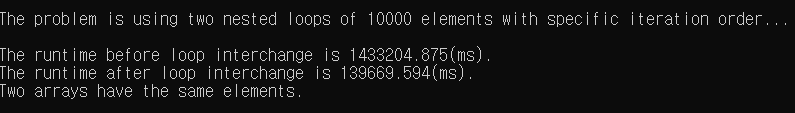
1. **Loop Interchange**

|  |  |
| --- | --- |
| **BEFORE** | **AFTER** |
| for (int j = 0; j < TWO\_23; ++j) {  for (int i = 0; i < TWO\_23; ++i) {  arr1[i][j] = i + j + 1;  }  } | for (int i = 0; i < TWO\_23; ++i) {  for (int j = 0; j < TWO\_23; ++j) {  arr2[i][j] = i + j + 1;  }  } |

< Debug Mode >



< Release Mode >

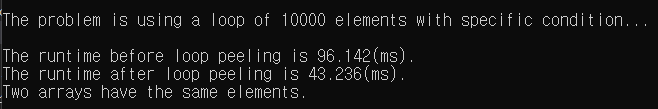


* Loop Interchange 기법은 루프의 순서를 뒤바꾸는 기법이다. 이 기법은 데이터 locality를 개선하고, 메모리 stride를 최소화하고 cache-line hit rate를 최대화하는 데 의의가 있다.
* 위 실험 결과로 보아 Debug Mode에서는 loop interchange 후에 시간이 상당히 줄어든 것을 확인할 수 있다. 그러나 Release Mode에서는 줄어들긴 했으나 Debug Mode와 달리 큰 차이를 보이지 않았다.

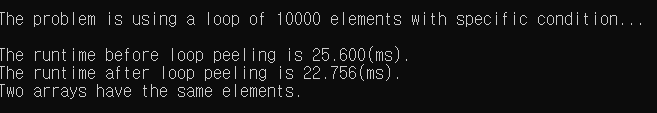
1. **Loop Peeling**

|  |  |
| --- | --- |
| **BEFORE** | **AFTER** |
| for (int i = 0; i < TWO\_23; ++i) {  arr4[i] = (i == 0) ? arr3[i] : arr3[i] + arr4[i - 1];  } | arr5[0] = arr3[0];  for (int i = 1; i < TWO\_23; ++i) {  arr5[i] = arr3[i] + arr5[i - 1];  } |

< Debug Mode >



< Release Mode >

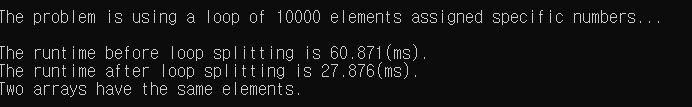


* Loop Peeling 기법은 루프 반복의 수를 줄이고 줄인 것을 루프의 앞이나 뒤에 옮기는 기법이다.
* 위 실험 결과로 보아 Debug Mode에서는 loop peeling 후에 시간이 상당히 줄어든 것을 확인할 수 있다. 그러나 Release Mode에서는 줄어들긴 했으나 Debug Mode와 달리 큰 차이를 보이지 않았다.

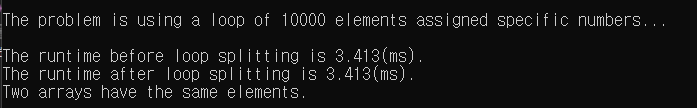
1. **Loop Splitting**

|  |  |
| --- | --- |
| **BEFORE** | **AFTER** |
| for (int i = 0; i < TWO\_23; ++i) {  arr4[i] = 2 \* arr3[i] + 1;  } | for (int i = 0; i < 100; ++i) {  arr5[i] = 2 \* arr3[i] + 1;  }  for (int i = 100; i < 500; ++i) {  arr5[i] = 2 \* arr3[i] + 1;  }  for (int i = 500; i < TWO\_23; ++i) {  arr5[i] = 2 \* arr3[i] + 1;  } |

< Debug Mode >



< Release Mode >

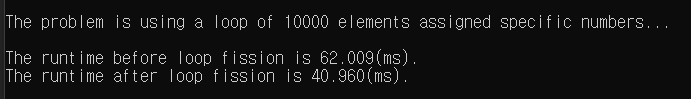


* Loop Splitting 기법은 하나의 루프를 다수의 루프로 쪼개는데 각각 다수의 범위로 쪼개는 기법이다.
* 위 실험 결과로 보아 Debug Mode에서는 loop splitting 후에 시간이 상당히 줄어든 것을 확인할 수 있다. 그러나 Release Mode에서는 Debug Mode와 계속 같은 시간이 나왔다.

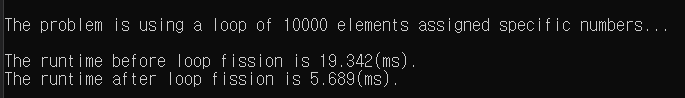
1. **Loop Fission**

|  |  |
| --- | --- |
| **BEFORE** | **AFTER** |
| for (int i = 0; i < TWO\_23; ++i) {  arr4[i] = 2 \* arr3[i];  arr5[i] = 3 \* arr3[i] + 1;  } | for (int i = 0; i < TWO\_23; ++i) {  arr4[i] = 2 \* arr3[i];  }  for (int i = 0; i < TWO\_23; ++i) {  arr5[i] = 3 \* arr3[i] + 1;  } |

< Debug Mode >



< Release Mode >

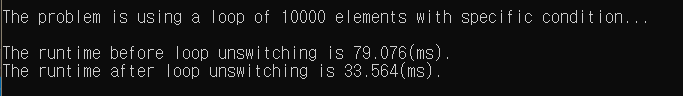


* Loop Fission 기법은 하나의 루프를 다수의 루프로 쪼개는 기법이다. 이 기법은 참조 위치를 효율적으로 사용할 수 있고, 병렬적인 루프를 격리시키고, 각각의 독립적인 루프를 만들어 작업을 분리할 수 있는 데 의의가 있다.
* 위 실험 결과로 보아 Debug Mode와 Release Mode에서 loop Fission 후에 시간이 상당히 줄어든 것을 확인할 수 있다.

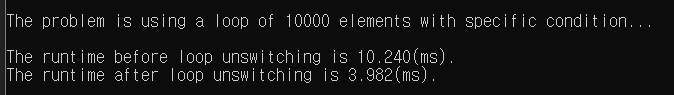
1. **Loop Unswitching**

|  |  |
| --- | --- |
| **BEFORE** | **AFTER** |
| for (int i = 0; i < TWO\_23; ++i) {  arr4[i] = 2 \* arr3[i] + 1;  if(p > 0) arr5[i] = p \* arr3[i] + i;  } | if (p > 0) {  for (int i = 0; i < TWO\_23; ++i) {  arr4[i] = 2 \* arr3[i] + 1;  arr5[i] = p \* arr3[i] + i;  }  }  else {  for (int i = 0; i < TWO\_23; ++i) {  arr4[i] = 2 \* arr3[i] + 1;  }  } |

< Debug Mode >



< Release Mode >



* Loop Unswitching 기법은 루프 안의 조건을 바깥으로 옮겨 그에 따라 루프를 복제하는 기법이다. 이 기법은 조건문 검사의 수를 줄이고 루프의 바디를 단순화시키는 데 의의가 있다.
* 위 실험 결과로 보아 Debug Mode와 Release Mode에서 Loop Unswitching 후에 시간이 상당히 줄어든 것을 확인할 수 있다.
* Debug 모드에서는 실행 파일에 디버깅 정보를 포함하여 언제든지 디버깅할 수 있다. Release 모드 보다 더 큰 메모리를 사용하고 디버그에 필요한 정보들을 실행 시 계속 체크하기 때문에 그만큼 속도가 저하된다. 그러나 Release 모드에서는 디버그에 대한 정보가 없으며, 코드를 최적화하여 실행 파일 크기를 최대한 줄여준다. 초기화를 하지 않으며, 같은 문자열 상수라도 서로 다른 공간에 할당된다. 속도와 크기, 메모리 사용면에서 Debug 모드보다는 좋으며 순수한 코드 자체의 기능만 담긴 파일이다. 이러한 차이점으로 인해 각 기법에 대해 실행 속도의 차이가 드러난 것으로 확인할 수 있었다.