**고급소프트웨어실습1**

**10주차 과제**

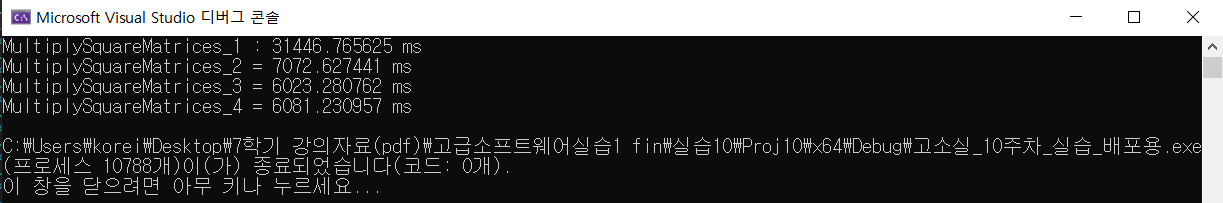
**20161663 허재성**

**실습 결과**

실습은 실습 1, 2 3으로 나누어져 있는데 각 실습 결과를 확인하기 위해서는 각 실습에 해당하는 cpp 파일만을 빌드에 포함시키고 실행하면 사용자가 별도로 입력할 필요 없이 결과를 확인할 수 있다.

**실습 1 결과**

실습 1의 결과는 1\_main.cpp를 실행하면 확인할 수 있다.

****

실습 1에서는 크기가 1024 by 1024인 두 정사각행렬의 곱을 구한다. 총 4가지 방법으로 근을 구하는데 첫 번째 방법은 전형적인 행렬 곱셈을 코드로 옮겨 삼중 루프를 이용하여 곱을 구하는 방법이다. 두 번째 방법은 두 행렬 A, B를 곱할 때 행렬 B의 전치 행렬을 이용하여 곱셈 시간을 획기적으로 줄이는 방법이다. 세 번째, 네 번째 방법은 두 번째 방법에서의 삼중 루프의 가장 안쪽 루프를 Loop Unrolling을 이용하여 최적화하여 실행시간을 좀 더 단축시키는 방법으로 세 번째 방법에서는 m=8, 네 번째 방법에서는 m=16을 이용하였다.

첫 번째 방법은 가장 직관적으로 쉽게 구현할 수 있지만 약 31447ms의 시간이 걸려 성능이 좋지 못하다.

두 번째 방법은 수행 시간이 약 7073ms로 실행속도가 4배 이상 빨라졌다. 첫 번째 방법에서는 삼중 루프의 가장 안쪽 루프(index k)에서 오른쪽 행렬에 접근할 때 pRightMatrix[k \* MatSize + j]로 MatSize씩 건너 뛰어 접근해야 했기 때문에 메모리 접근 속도가 매우 느렸다. 하지만 두 번째 방법에서는 오른쪽 행렬의 전치행렬 pRightTranspose을 구하여 계산했기 때문에 가장 안쪽 루프에서 pRightTranspose[j \* MatSize + k]로 1씩 건너 뛰어 접근할 수 있어서 메모리 접근 속도가 첫 번째 방법에 비해 매우 빠르다. 두 번째 방법에서 pRightTranspose를 동적 할당하고 pRightMatrix를 복사하는 코드가 추가되었음에도 두 번째 방법이 훨씬 빠르게 실행되었다.

세 번째 방법은 두 번째 방법의 삼중 루프의 가장 안쪽 루프를 unrolling하여 8개 단위로 풀어 수행하였다. Loop Unrolling을 이용하면 반복 횟수를 줄일 수 있고 이는 loop문에 수반되는 branch 계산, index 계산, 이동 횟수가 줄어들어 시간을 절약할 수 있음을 의미한다. 첫 번째 방법에서 두 번째 방법으로 계산할 때만큼 극적인 변화를 보여주진 못하지만 수행 시간이 7073ms에서 6023ms로 1.17배정도 빨라졌다.

네 번째 방법은 세 번째 방법과 같지만 16개 단위로 풀어 수행하였다. 반복 횟수가 훨씬 줄어들어 실행 시간도 세 번째 방법보다 감소할 것 같지만 결과는 오히려 조금 더 증가하였다. 16개 단위로 풀어 코드의 양이 많아져서 overhead가 커졌기 때문이다.

정사각행렬의 곱셈의 경우 오른쪽 행렬의 전치행렬을 구하여 곱하면 오른쪽 행렬에 해당하는 메모리에 더 효율적으로 접근하여 계산이 가능하기 때문에 획기적인 속도 향상을 기대할 수 있다. 기본적으로 전치행렬을 이용한 곱셈으로 실행 시간을 크게 줄인 후 적절한 m을 찾아 그 수만큼 loop unrolling을 수행하면 실행 시간을 더 줄일 수 있다. M이 클수록 반복 횟수는 줄어들지만 코드의 양이 많아져서 비효율적일 수 있다.

**실습 2 결과**

실습 2의 결과는 2\_main.cpp를 실행하면 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

실습 2는 10차 다항식을 임의로 생성하고 1048576개의 x를 생성하여 각각 대입하여 다항식의 값을 확인한다. 이 때 단순한 대입으로 계산하는 naïve 방법과 Horner 법칙을 이용한 방법의 수행시간을 비교해본다. Naïve 방법의 경우 한 항의 값을 계산하기 위해 계수와 곱셈 한번 항의 차수만큼 거듭제곱을 수행해야 하여 곱셈의 횟수가 매우 많아지며 다항식 값을 구하기 위해 덧셈 횟수도 차수 + 1만큼 필요하다. 반면 Horner 법칙을 이용하면 곱셈과 덧셈을 각각 차수횟수만큼만 하면 되므로 곱셈의 횟수가 크게 줄어든다. 실제로 결과를 확인해보면 실행시간이 800ms에서 77ms로 10배 이상 빨라졌음을 알 수 있다.

**실습 3 결과**

실습 3의 결과는 3\_main.cpp를 실행하면 확인할 수 있다.

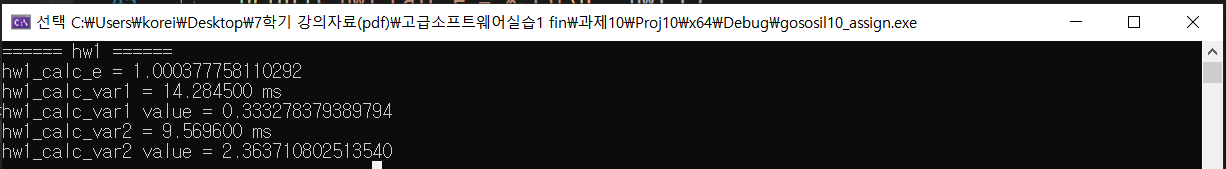
텍스트이(가) 표시된 사진

자동 생성된 설명

Horner 법칙을 이용하여 Taylor 급수로 e-8.3의 값을 구한다. 이 때 반환형이 float인 경우(첫 번째)와 double인 경우(두 번째)를 나누어 계산한다. 항의 계수를 50으로 크게 하여 계산한 결과를 pow를 이용해 계산한 실제 값(세 번째)와 비교한다. Float을 이용한 single precision 방법으로는 실제 값과 일치하지 않았다. Float를 이용할 경우 23개 비트로 정밀도를 계산하여 10진수로 따질 경우 소수점 아래로 6-7자리까지가 유효 숫자가 된다. 그 이후 숫자는 유효하지 않다. 반대로 double을 이용한 방법으로는 52비트로 정밀도를 계산하여 10진수를 따질 경우 소수점 아래로 약 15자리까지는 유효 숫자가 된다. 이로 인해 double로 계산했을 때 원래 값과 같은 값이 나옴을 알 수 있다.

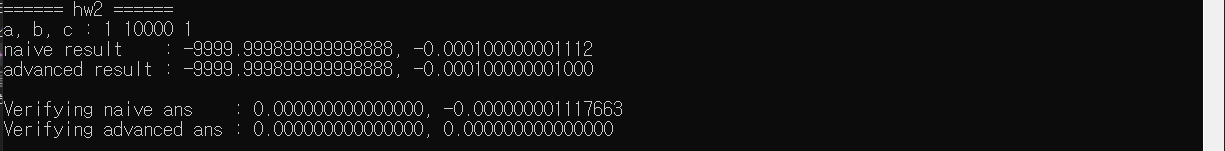
**숙제 결과**

숙제의 경우 실행시키면 숙제 1의 경우 자동으로 결과를 확인할 수 있으며 숙제 2는 이차 방정식의 계수를 입력하여 결과를 확인할 수 있다. 숙제의 경우 정밀도를 높이기 위해 float 대신 double을 사용하였다.

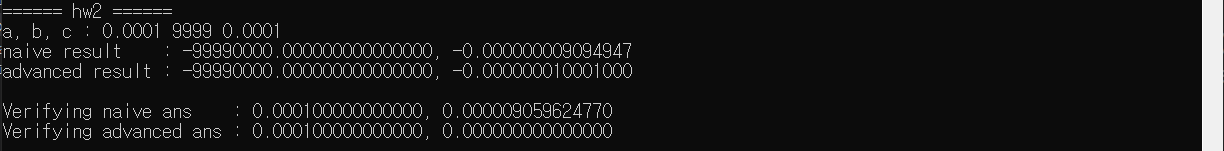
**숙제 1 결과**

10만개의 실수를 임의로 생성하여 분산을 구했다. 분산을 구하는 두 가지 동등한 식을 이용해 수학적으로 동등한 식을 컴퓨터로 계산할 때의 차이를 확인하는 게 목적이다. 숙제 1의 경우 10만개의 실수를 임의로 생성할 때 생성 함수 init\_hw1에 인자로 1을 줄 경우, 숙제에 적합한 실수들이 생성된다. 이를 이용해 계산한 결과 첫 번째 식을 이용한 결과는 정확한 분산을 구할 수 있었지만(엑셀로 해당 데이터에 대한 분산을 구해 비교한 결과 소수점 아래 정밀도 차이는 있지만 대체로 일치했다.) 두 번째 식을 이용한 결과는 실제 분산과 큰 차이가 났다.

두 번째 방법의 경우 각 난수의 제곱의 합에 난수의 개수를 곱한 것과 각 난수의 합의 제곱이 비슷한 값이 되어 그 차가 0에 수렴하기 때문에 부정확한 값을 얻게 됨을 알 수 있다. 실행 시간의 경우 모든 경우 그런 것은 아니지만 두 번째 방법이 조금 더 빨랐다. 첫 번째 방법의 경우 난수의 개수만큼 뺄셈을 수행 후 제곱 또한 그만큼 수행해야 하지만 두번째 방법의 경우 난수의 합의 제곱을 이미 구한 평균을 이용해 계산할 수 있으므로 뺄셈은 한 번만 수행하면 된다. 따라서 두 번째 방법이 실행 시간은 더 적게 걸릴 것을 예측할 수 있다.

**숙제 2 결과**

첫 번째 방정식으로 a=1, b=10000, c=1인 이차방정식을 생각해본다.

****

두 번째 방정식으로 a=0.0001, b=9999, c=0.0001인 이차방정식을 생각해본다.

두 경우 모두 b에 비해 a, c가 매우 작아 sqrt(b2-4ac)가 b와 매우 가깝게 된다. 이 경우 이차 방정식의 두 근 중 한 근(나머지 한 근은 두 방법 모두 같은 방법으로 구하여 같다.)은 계산 과정에서 비슷한 두 수 사이의 뺄셈이 존재하게 되어 오차가 크게 발생하게 된다. 첫 번째 방정식의 경우 naïve한 방법, 즉 비슷한 두 수의 뺄셈을 개선하지 않은 방법으로 근을 구하면 얼핏 보면 비슷해 보이지만 원래 방정식에 대입해보면 0이 되지 않음을 알 수 있고, 두 번째 방정식의 경우는 근도 확실히 다름을 알 수 있고 대입했을 때 값도 0과 큰 차이가 있음을 알 수 있다. Advanced한 방법으로 분자를 유리화하면 비슷한 두 수의 뺄셈이 사라져서 더 안전한 근을 구할 수 있다. 각 방정식의 advanced한 근을 대입한 값을 확인하면 0이 됨을 알 수 있다.