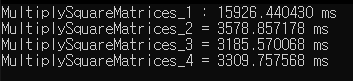
**고소실 10주차 과제 보고서**

20171617 김소연

**실습 1**



원래의 1번 함수를 변형해서 2번 함수를 구현할 때, matrix b를 전치함수로 뒤집어 주었다.

C++ 언어는 2차원 배열에 접근할 때, 행을 우선으로 선형화하여 1차원 메모리에 저장한다.  
기존에 B 행렬에 접근할 때는 같은 행에서 열을 늘려가며 메모리에 접근하였는데, 이는 행 우선인 C++에서 메모리를 이동하는 데에 시간을 소모하게 된다. 따라서, B 행렬을 전치로 뒤집어서 행 우선으로 데이터에 접근하고 계산을 수행하여 코드의 소요시간을 줄였다. 이를 메모리 접근의 응집성을 높였다고 할 수 있다.

3, 4번 함수는 2번 함수에서 loop unrolling 기법을 적용하여, m개의 단위로 for문을 쪼갠 후 수행시간을 비교하였다.

3번은 8개 단위로 쪼개었고, 4번은 16번 단위로 쪼개었다. 그 결과 3, 4번 둘 다 2번보다 수행 시간에 있어 좋은 성능을 보였으나, 3번보다 4번의 수행 시간에 있어서 성능은 좋지 않은 것으로 나타났다.

따라서, m이 크다고 수행 시간이 줄어드는 것은 아니며 적절한 m을 찾아 적용하는 것이 중요하다 할 수 있다.

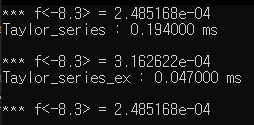
**실습 2**



기존의 다차방정식을 계산하는 방법보다, Horner’s rule을 적용한 방법에서 수행 시간에 있어서 뛰어난 성능을 보여주는 것을 확인할 수 있다.

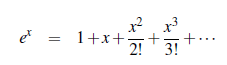
Horner’s rule는 코드에서 반복적으로 수행되는 곱셈을 줄여줌으로서, 코드의 전체적인 수행시간을 줄여주었다. 기존 다차방정식 계산 방법에서 여러 번 수행되는 x의 n승이 그 예라고 할 수 있다.

**실습 3**



이 실험에서는 float과 double 자료형에 있어서 계산 수행 속도의 차이와 정확도의 차이, 그리고 2번과 마찬가지로 horner’s rule을 적용했을 때의 속도 차이를 확인할 수 있었다.

위의 수행 결과는 double으로 구현한 무한 급수의 합을 계산한 결과이고,



아래의 수행결과는 float으로 구현한 horner’s rule을 이용한 무한 급수의 합을 계산한 결과이다.



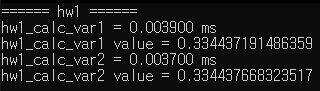
아래의 float으로 구현한, horner’s rule을 적용한 결과의 속도가 확연히 빠른 것을 확인할 수 있다.

하지만, double로 구현한 수행결과가 그 답에 있어서 정확도가 높은 것을 확인할 수 있다.

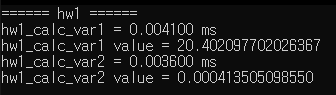
Double은 float보다 더 많은 수의 bit를 가지고 실수를 표현한다. 그렇기에 double 자료형을 이용해 실수를 표현하고 연산을 수행하는 것이 float을 이용하는 것보다 정확도가 높다.

단, double은 그만큼 계산을 수행해야할 비트의 수가 많다. 이는 곧, 속도 면에 있어서 연산 속도가 float보다 느리다는 것을 의미하며, 필요한 메모리가 더 많다는 것을 의미한다. 실습에서처럼 작은 단위의 연산에서 이 논의는 의미가 없을테지만, 딥러닝과 같이 연산을 많이 필요로 하는 작업에서 이는 곧 비용의 문제와 직결된다. 따라서 앞으로도 float이 많이 사용될텐데, 이 float의 연산과정에서 발생하는 오차들을 줄이기위한 프로그래머들의 연구와 노력이 필요할 것이다.

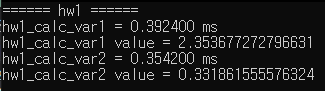
**숙제 1**



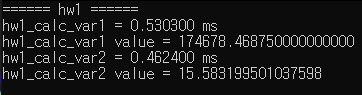
A-1. Random한 1000개의 x를 통해 계산한 결과



A-2. X의 평균과 x가 비슷하도록 고정한 1000개의 x를 통해 계산한 결과



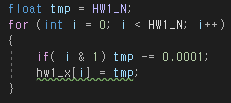
B-1. Random한 100000개의 x를 통해 계산한 결과



B-2. X의 평균과 x가 비슷하도록 고정한 100000개의 x를 통해 계산한 결과

1. **Var1와 var2중 어떤 것이 더 정확한가?**

우선, x의 값들을 고정한 경우를 살펴보자. 다음의 식을 통해 값들을 고정하고 HW1\_N만큼의 x를 생성해주었다.



이를 통하면, 홀수번째의 수만 HW1\_N – 0.0001의 값을 갖고, 짝수번째의 수는 HW1\_N의 값을 갖는다. 따라서 평균값은 HW1\_N – 0.00005정도의 값을 가질 것이다. 이를 통해 분산을 예측해보면 0에 가까운 값을 가질 것으로 예상할 수 있다.



Var2를 계산한 위의 식을 살펴보면, xi에서 x의 평균값을 빼는 식을 확인할 수 있다. 위의 데이터 셋의 경우 상당히 비슷한 숫자들과 뺄셈을 해주는데, 이 과정에서 부동소수점으로 인한 오차가 발생할 것으로 예상하였다.



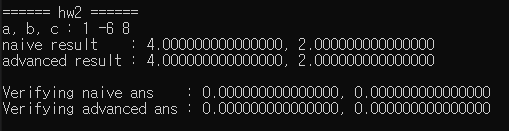
따라서 앞서 해당하는 비슷한 수들의 뺄셈을 피하여, 위의 식을 통해 계산한 Var1이 더 정확하다고 예측하였다.

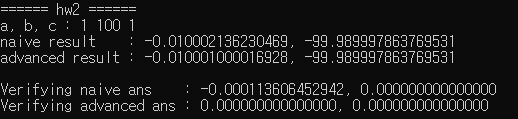
하지만 결과를 보았을 때, var1의 오차가 더 심한 것을 확인할 수 있었으며, 음수가 나오는 경우까지 확인할 수 있었다. 이는 비슷한 숫자들간의 뺄셈으로 발생하는 부동 소수점의 오차의 영향을 능가하는 결함이 있었던 것으로 이해할 수 있다.

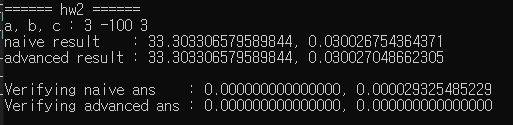
1. **어떤 방법이 더 빠르게 분산을 계산하는가?**

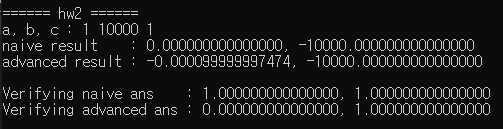
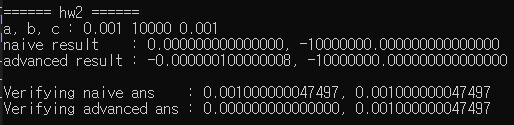
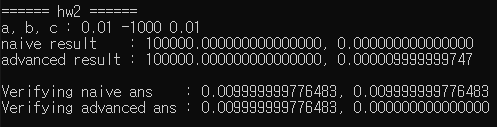
위의 결과를 통해 알 수 있듯이, Var1을 계산하는 과정이 더 오래 걸리는 것을 확인할 수 있다. 이는 Var1의 연산 횟수가 Var2의 연산 횟수보다 더 많기 때문으로 이해할 수 있다.

**숙제 2**

  
C-1. 근을 제대로 산출한 경우



  
C-2. naïve 함수가 산출한 결과에 오차가 존재함

  
  
  
C-3. naïve 함수가 근사한 근을 계산해내지 못함.

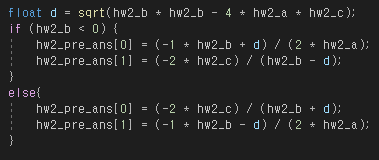
C-2, C-3 에서의 5개의 예시는 본래 알려진 근의 공식으로 문제를 풀었을 때 근사하는 근이 산출되지 않거나 산출한 근이 0에 수렴하지 않는 치명적인 오류가 발생한 경우를 나타낸 것이다.

※C-3에서는 개선한 함수 또한 대입한 결과가 0에 수렴하지 않는 듯 보이는 결과가 보인다. 하지만 C-3의 예시들을 보면B가 매우 큰 값을 갖는데, 이는 b의 값이 커지며 근의 아주 작은 변화로도 대입 결과의 격차가 커지기 때문으로 이해할 수 있다. 또 다른 이유로는 실수간의 계산과정에서 오차가 발생했기 때문으로 예상할 수 있다.

그 결과가 주로 B의 값과 a \* c의 값과의 격차가 심할 때 이러한 오류가 발생함을 확인할 수 있다.

이러한 오류는 에서 를 계산할 때, 4ac가 b^2보다 매우 작은 값일 때, 와 는 매우 비슷한 값이 되어, 부동소수점으로 인한 오류가 발생하는 것으로 이해할 수 있다.

이러한 점을 개선하여 이차방정식의 근을 산출하는 새로운 함수를 구현하였다. 비슷한 값끼리의 뺄셈이 일어나지 않도록 개선한 식이다.



b의 부호에 따라 가 뺄셈인지 덧셈인지 달라지기 때문에, b의 부호에 따라 조건을 달리해주었다.



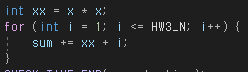
위와 같이 비슷한 숫자간의 뺄셈이 발생할 수 있는 상황에 대해 뺄셈을 덧셈으로 바꾸어 준 것이 위의 코드이다. 결과를 보면 본래의 근의 공식에서 산출하지 못했던 근을, 개선한 식에서는 근사한 값으로 산출하는 것을 확인할 수 있다.

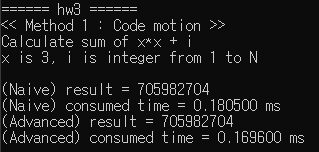
**숙제 3**

1. **Method 1 : Code Motion**

반복문을 수행할 때, 값이 변화가 없는 부분을 미리 계산하고 미리 계산한 수를 대신 대입하여 시간을 최적화하는 방법을 말한다.

Ex)아래 수식을 계산하는 함수

* **Naïve**  
  
* **Advanced**  
  

**Result**  


Advanced된 loop의 속도가 더 빠른 것을 확인할 수 있으며, 이를 통해 코드가 최적화 된 것을 알 수 있다.

1. **Method 2 : Function Inlining**

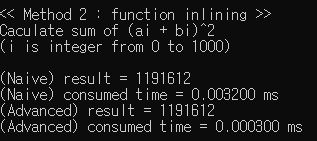
간단한 기능을 하는 함수는 함수를 생성하고 호출하여 결과를 얻는 것보다, 코드 내에서 해당 기능을 수행시켜 코드가 수행되는 시간을 최적화하는 방법이다.

함수를 호출할 때에는 복귀할 위치를 미리 스택에 저장해두고, 사용하던 레지스터를 스택에 저장해둔다. 함수의 수행이 종료되면 레지스터를 스택에서 pop하고 미리 저장해둔 복귀할 주소로 이동하는 과정을 거치는데, 함수를 호출하지 않고 원래 위치에서 해당 기능을 수행함으로서, 함수 호출 과정에서 소모되는 시간을 최적화시키는 방법이다.

Ex)아래 수식을 계산하는 함수

a와 b배열의 수들은 0 ~ 1000 사이의 임의의 정수

* **Naïve**  
  
* **Advanced**  
  

**Result**  


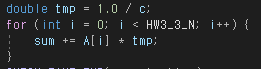
Advanced된 loop의 속도가 더 빠른 것을 확인할 수 있으며, 이를 통해 코드가 최적화된 것을 알 수 있다.

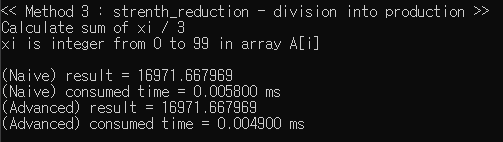
1. **Method 3 : Strength Reduction – 나눗셈을 곱셈으로 수정**

연산에서 나눗셈 연산은 큰 strength를 갖는다. 이러한 나눗셈을 상대적으로 strength가 낮은 곱셈으로 대치하여 코드 수행시간을 줄이는 최적화 기법이다.

Ex)아래 수식을 계산하는 함수

x배열의 수는 0부터 99까지의 임의의 정수

* **Naïve**  
  
* **Advanced**  
  

**Result**  


Advanced된 loop의 속도가 더 빠른 것을 확인할 수 있으며, 이를 통해 코드가 최적화된 것을 알 수 있다.

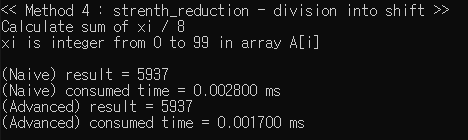
1. **Method 4 : Strength Reduction – 나눗셈을 SHIFT로 수정**

연산에서 나눗셈 연산은 큰 strength를 갖는다. 하지만 특별히 정수의 나눗셈의 경우, 나눗셈보다 상대적으로 strength가 작은 SHIFT로 대체하여 코드의 수행시간을 최적화시키는 기법이다.

Ex)아래 수식을 계산하는 함수

x배열의 수는 0부터 99까지의 임의의 정수

* **Naïve**  
   **(c는 8)**
* **Advanced**  
  

**Result**  


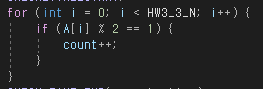
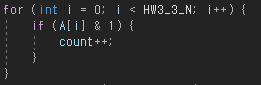
Advanced된 loop의 속도가 더 빠른 것을 확인할 수 있으며, 이를 통해 코드가 최적화된 것을 알 수 있다.

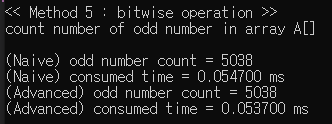
이 예시는 나눗셈을 right shift로 대치했는데, 마찬가지로 곱셈을 left shift로 대치하여 최적화할 수 있다.

1. **Method 5 : Bitwise operation**

많은 연산 중 Bit 단위의 연산은 다른 연산보다 확연히 그 속도가 빠르다. 따라서, bit 단위로 연산을 수행할 수 있는 코드는 bit 연산으로 대치시켜 코드가 수행되는 시간을 최적화시키는 기법이다.

Ex) N개의 요소를 가진 x배열에서 홀수의 개수를 세는 함수. X배열 내의 요소들은 0부터 99까지의 임의의 정수이다.

* **Naïve**  
  
* **Advanced**  
  

**Result**  


Advanced된 loop의 속도가 더 빠른 것을 확인할 수 있으며, 이를 통해 코드가 최적화된 것을 알 수 있다.