고급소프트웨어실습I

10주차 보고서

20171646 박태윤

(실습1)

행렬의 곱을 계산(C = A x B)하는 코드를 작성한다. 4개의 함수를 작성하는데, MultiplySquareMatrices\_1함수는 인자로 받은 행렬 pLeftMatrix와 pRightMatrix를 곱하여 pDesMatrix에 결과로 저장하는 함수이다. MultiplySquareMatrices\_2함수도 역시 두 행렬의 곱을 pDesMatrix에 저장하는 함수인데, MultiplySquareMatrices\_1와는 달리 행렬 메모리 접근 방식을 다르게 하여 조금 더 효율적인 계산이 이루어질 수 있도록 구현한다. MultiplySquareMatrices\_3은 loop unrolling을 사용하여 효율적인 계산이 이루어질 수 있도록 구현하였는데, 반복문은 16개 단위로 풀었다. 마지막 함수인 MultiplySquareMatrices\_4역시 loop unrolling을 사용하여 코드를 작성하였는데, loop unrolling을 단순하게 많은 단위로 푼다고 효율적인 계산이 이루어질 수 없음을 보이기 위해 3과 달리 64개 단위로 풀었다. 코드를 실행하면 다음과 같은 결과를 얻을 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

실행할 때 마다 어느 정도의 오차는 존재하지만, 단순하게 계산하는 1번 방법에 비해 2, 3, 4번이 효율적임을 확인할 수 있으며 더 많은 단위로 loop unrolling을 한 4번이 3번에 비해 조금 더 시간이 많이 걸리는 것을 확인할 수 있다.

(실습 2)

다항식의 값을 계산하는 프로그램을 작성한다. 이 때 두 가지 방법을 사용하는데, pow함수를 사용하여 각각의 x[i]에 대해 계산을 하는 Eval\_Poly\_Naive함수와 Honor’s rule을 사용하여 계산하는 Eval\_Poly\_Horner함수를 구현한다. Honor’s rule은 y = a5x 5 +a4x 4 +a3x 3 +a2x 2 +a1x+a와 같은 다항식을 y = ((((a5x+a4)x+a3)x+a2)x+a1)x+a 처럼 계산하는 방법을 의미한다. 두 함수를 이용하여 같은 다항식들에 대해 계산을 하였을 때, Check\_Difference함수에 의해 두 함수에서 다항식을 정상적으로 계산하여 구한 값이 같다면 All values are equal이란 메시지가 뜨고 프로그램이 종료될 것이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

정상적으로 계산을 했음을 All value are equal메시지로 알 수 있다. 계산 결과는 같지만 Naive함수보다 Horner’s rule을 이용한 함수가 계산 속도가 훨씬 빠른 것 또한 알 수 있다.

(실습 3)

Taylor series에 기반을 둔 무한 급수의 합을 계산하는 프로그램을 작성한다. Horner’s method를 사용하여 무한 급수의 합을 계산하는데, 이 때 두 가지 함수를 작성한다. 이는 float자료형을 사용하여 계산하는 Taylor\_series와 double자료형을 사용하여 계산하는 Taylor\_series\_ex인데, 두 함수의 결과를 출력한 뒤 math.h라이브러리에 있는 pow함수를 이용하여 실제 결과를 출력한 뒤 값을 비교한다. 프로그램을 실행하면 다음과 같은 결과를 얻을 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

float자료형을 사용하여 값을 계산했을 경우 실제와는 다른 결과를 얻는 것을 확인할 수 있다. double자료형을 사용하면 유효숫자의 자리 수가 늘어나 float에 비해 더 정확한 결과를 얻을 수 있기 때문에 다음과 같은 결과가 나타난 것이다.

(과제 1)

두 가지 방법을 이용하여 분산을 구하는 프로그램을 작성한다.

텍스트, 시계이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

자료의 평균은 hw1\_calc\_e함수에서, 1번 분산은 hw1\_calc\_var1함수에서, 2번 분산은 hw1\_calc\_var2함수에서 계산이 된다. 코드는 다음과 같이 작성하였다.

void hw1\_calc\_e()

{

hw1\_E = 0.0f;

for (int i = 0; i < HW1\_N; i++) {

hw1\_E += hw1\_x[i];

}

hw1\_E = hw1\_E / HW1\_N;

}

void hw1\_calc\_var1()

{

hw1\_var1 = 0.0f;

double val = 0.0f;

for (int i = 0; i < HW1\_N; i++) {

hw1\_var1 += pow(hw1\_x[i] - hw1\_E, 2.0);

}

hw1\_var1 = hw1\_var1 / (HW1\_N - 1.0);

}

void hw1\_calc\_var2()

{

hw1\_var2 = 0.0;

double tmp1 = 0.0, tmp2 = 0.0;

for (int i = 0; i < HW1\_N; i++) {

tmp1 += pow(hw1\_x[i], 2.0);

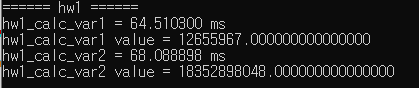
tmp2 += hw1\_x[i];

}

hw1\_var2 = (HW1\_N \* tmp1 - pow(tmp2, 2.0)) / (HW1\_N \* (HW1\_N - 1.0));

}

처음에 주어져 있던 float자료형으로 계산을 하면 다음과 같은 결과를 얻을 수 있다. 자료는 100만개로 설정하였다.



두 함수를 이용하여 계산한 분산이 같은 데이터를 사용함에도 매우 다른 값을 나타냄을 확인할 수 있다. double자료형으로 계산을 하면 다음과 같은 결과를 얻을 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

앞선 결과에 비해 어느 정도 유사한 결과를 나타냄을 확인할 수 있다. 또한 1번 함수의 계산 속도가 2번에 비해 더 빠른 것을 확인할 수 있다. 값의 정확도는 1번 함수가 더 높다고 볼 수 있는데, 이유는 2번 식에서



이 계산을 하는 과정에서 비슷한 수의 뺄셈이 일어날 가능성이 존재하기 때문이다.

double타입을 사용하지 않으면 많은 데이터를 가지고 분산을 구할 때, 유효 숫자가 부족한 경우가 발생하여 정확하지 않은 값을 구할 수 있기 때문에 사용을 하였다.

(2번 과제)

이차방정식 에서 a, b, c를 입력 받아 근을 구하는 프로그램을 작성한다. 함수를 두 개 구현하는데, 첫번째 함수는 근의 공식을 그대로 이용하여 두 근을 구하는 함수이고 두번째 함수는 프로그램으로 근의 공식을 이용해 이차방정식의 근을 구할 때 잘못된 근을 구할 수 있는 경우를 찾아 이를 완화시켜 더 적절한 근을 구할 수 있는 함수이다. 코드는 다음과 같이 구현하였다.

/\* hw2 \*/

void hw2\_naive()

{

hw2\_naive\_ans[0] = ((-1.0) \* hw2\_b + sqrt(pow(hw2\_b, 2.0) - (4 \* hw2\_a \* hw2\_c))) / (2 \* hw2\_a);

hw2\_naive\_ans[1] = ((-1.0) \* hw2\_b - sqrt(pow(hw2\_b, 2.0) - (4 \* hw2\_a \* hw2\_c))) / (2 \* hw2\_a);

}

void hw2\_safe()

{

hw2\_pre\_ans[0] = (-2.0 \* hw2\_c) / (sqrt(pow(hw2\_b, 2.0) - 4.0 \* hw2\_a \* hw2\_c) + hw2\_b);

float tmp = hw2\_pre\_ans[0] \* 2.0 \* hw2\_a;

hw2\_pre\_ans[1] = (2.0 \* hw2\_c) / tmp;

}

근의 공식을 이용하여 hw2\_naive처럼 근을 구할 때 생길 수 있는 문제는 근의 공식 에서 를 봤을 때, 이 4ac에 비해 월등히 큰 경우 는 곧 비슷한 숫자 간의 뺄셈이 되는 경우에서 발생하게 된다. 따라서 이를 유리화 하여 는 로, 는 (앞에서 구한 근을 이용하여 계산하였다)로 근을 구하였다. 문제를 야기할 수 있는 세 가지 경우에 대해 프로그램을 실행시켰고 다음과 같은 결과를 도출하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

근이 적절한지를 확인하기 위해 구한 근을 식에 다시 대입해서 결과를 도출하는 Verifying ans부분을 보면 확실히 근의 공식을 유리화 하여 계산했을 때와 그렇게 하지 않았을 때 결과가 다른 것을 확인할 수 있다. 근의 공식을 유리화 하면 두 근중 한 근은 문제를 야기할 수 있는 경우에 매우 적절한 근을 구하는 것을 확인할 수 있다. 하지만 두 근 모두 적절하다고는 볼 수 없는 결과를 얻었는데 이는 자료형을 float으로 지정하여 표현할 수 있는 유효숫자의 개수가 적기 때문이다. 3번째 경우 (a = 2, b = 50000000, c = 5)에서 자료형을 double로 바꾸어 다시 계산하면 다음과 같은 결과를 얻을 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

유리화를 하여 근의 공식을 사용하는 방법에서 적절한 두 실근을 구하는 것을 확인할 수 있다.

(과제 3)

1. 조건문 분리

같은 동작을 하는 조건문이라도 변형을 하면 코드 최적화가 가능하다. 이에 대한 코드는 다음과 같이 작성하였다.

void if\_not\_break() {

int num = 99;

for (int i = 0; i < 1000000000; i++) {

if (num < 10) {}

else if (num < 20) {}

else if (num < 30) {}

else if (num < 40) {}

else if (num < 50) {}

else if (num < 60) {}

else if (num < 70) {}

else if (num < 80) {}

else if (num < 90) {}

else if (num < 100) {}

}

}

void if\_break() {

int num = 99;

for (int i = 0; i < 1000000000; i++) {

if (num < 50) {

if (num < 30) {

if(num < 10){}

else {}

}

else {

if(num < 40){}

else {}

}

}

else {

if (num < 80) {

if (num < 60) {}

else {}

}

else {

if(num < 90){}

else{}

}

}

}

}

같은 동작을 하는 조건문에 대한 함수를 다음과 같이 두 개 작성한 뒤 동작 시간을 측정하였고, 다음과 같은 결과를 얻었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위의 경우는 num을 99로 세팅하여 if\_not\_break에서 조건문 비교를 10번, if\_break에서는 그보다 더 적은 수로 수행을 하는 경우를 설정하였다. 결과에서 확인할 수 있듯, 같은 동작을 하는 조건문이라도 코드를 어떻게 작성하냐에 따라 수행시간이 달라지는 것을 확인할 수 있다.

1. shift 나눗셈

비용이 많이 드는 ‘/’연산자를 이용한 나눗셈 대신 비트 연산자 ‘>>’를 활용하여 나눗셈을 진행하면 코드 최적화를 할 수 있다. 이에 대한 코드는 다음과 같이 작성하였다.

void divide() {

int res;

for (int i = 0; i < 1000000; i++) {

res = i / 4;

}

}

void shift\_divide() {

int res;

for (int i = 0; i < 1000000; i++) {

res = i >> 2;

}

}

0~999999까지의 i를 4로 나누어 res에 저장하는 같은 기능을 하는 두 함수이다. 4로 나누는 부분에서 ‘/’로 나누는지, >>연산자를 사용하여 계산을 하는지에 대한 차이가 존재한다. 시간 측정을 했을 때 다음과 같은 결과를 얻을 수 있었다.



shift연산자를 사용하여 같은 나눗셈을 진행한 결과가 더 빠른 것을 확인할 수 있다.

1. bit 연산

OR, AND, XOR등 빠르게 실행될 수 있는 연산을 이용하면 코드 최적화를 할 수 있다. 이에 대한 코드는 다음과 같이 작성하였다.

void not\_bit() {

int num = 3;

for (int i = 0; i < 100000000; i++) {

if (num % 2 == 1) {}

else if (num % 2 == 0) {}

}

}

void bit() {

int num = 3;

for (int i = 0; i < 100000000; i++) {

if (num & 1) {}

else if (!(num & 1)) {}

}

}

num이 짝수인지 홀수인지를 구별하는 기능을 하는 두 함수를 구현하였다. not\_bit은 ‘%’연산자를, bit함수는 bit연산인 ‘&’연산자를 이용하여 같은 횟수만큼 연산을 진행하였고, 시간을 측정하였을 때 이와 같은 결과를 얻을 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

같은 기능을 하는 함수라도 bit연산을 활용했을 때 더 속도가 빠른 것을 확인할 수 있다.

1. 반복문 최적화

반복문은 연산 속도가 매우 느리기 때문에 사용하지 않을 수 있다면 사용하지 않는 것이 코드최적화면에서는 좋다. 이에 대한 코드는 다음과 같이 작성하였다.

void loop() {

int res = 0;

for (int i = 0; i <= 500000; i++)

res += i;

printf("loop res : %d\n", res);

}

void not\_loop() {

int res = 0;

int num = 500000;

res = (num \* (num + 1)) / 2;

printf("not\_loop res : %d\n", res);

}

0 ~ 499999까지의 합을 구해서 res에 저장하는 같은 기능을 하는 두 함수이다. loop는 반복문을 활용하여 이를 구하였고 not\_loop는 공식을 사용하여 이를 구하였다. 코드를 실행시켰을 때 다음과 같은 결과를 얻을 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

같은 값을 도출하였는데, loop를 사용하지 않는 경우가 연산속도가 훨씬 빠른 것 확인할 수 있다.

1. 함수 호출 최적화

함수 호출은 비용이 꽤나 들기 때문에 이를 조정하면 코드최적화를 할 수 있다. 이에 대한 코드는 다음과 같이 작성하였다.

void func() {

int res = 0;

for (int i = 0; i < 1000000; i++) {

res += sum(1, 2);

}

printf("func : %d\n", res);

}

void not\_func() {

int res = 0;

for (int i = 0; i < 1000000; i++) {

res += (1 + 2);

}

printf("not\_func : %d\n", res);

}

1000000번 동안 (1 + 2)를 res에 더하는 같은 기능을 하는 함수이다. func에서는 sum이라는 인자로 받은 두 정수를 더해서 리턴하는 함수를 이용하여 이를 계산하였고 not\_func에서는 함수 호출 없이 계산하였다. 이를 실행시키면 다음과 같은 결과를 얻을 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

같은 결과를 도출하였는데, 함수 호출 없이 계산을 하는 방법이 훨씬 빠른 것을 확인할 수 있다.

(Debug & Release)

앞선 실습, 과제의 프로그램 구동 결과는 Debug모드로 실행시켰을 때 나타난 결과이다. 같은 과제 프로그램을 Debug가 아닌 Release모드로 실행시켰을 때는 다음과 같은 결과를 얻을 수 있다. (자료형은 float타입으로 구동)

텍스트이(가) 표시된 사진

자동 생성된 설명

분명 앞선 보고서의 내용에 기술된 코드와 같은 코드를 실행시켰는데, 전체적으로 실행 시간이 모두 줄은 것을 확인할 수 있다. 그 이유는 release로 빌드를 하면 dubug로 빌드 할 때와 달리 디버그에 필요한 정보들을 체크하지 않고 코드를 최적화시켜 실행 파일 크기를 줄이고 이로 인해 debug빌드 보다 더 빠르게 실행시킬 수 있기 때문입니다.