고소실 10주차 보고서

20191571 김세영

1. 실습

1) (1) 자신이 적용한 방법

MultiplySquareMatrices_2: double* pMatB_t를 새로 추가하여 pRightMatrix의 전치행렬이 되도록 동적할당 후 값을 저장하였다. 그후 AxB=C라는 행렬을 계산할 때 B대신 B의 전치행렬인 pMatB_t를 대입하였다.

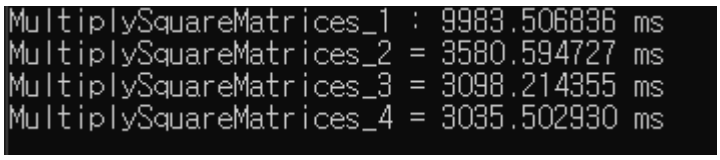
MultiplySquareMatrices_3: 2와 마찬가지로 전치행렬을 적용하였고, 가장 안쪽의 for문을 4개 단위로 풀었다.

MultiplySquareMatrices_4: 3과 마찬가지로 전치행렬을 적용하였고, 가장 안쪽의 for문을 8개 단위로 풀었다.

(2) 프로그램이 효율적인 근거

전치행렬: 기존 행렬의 곱셈에는 pRightMatrix를 col 단위로 접근하였다. 각 행의 원소는 matrix의 col의 개수(width)만큼 떨어져 있기 때문에 응집력이 약하다. 따라서 pRightMatrix의 전치행렬을 새로 추가하여 계산할 때 pRightMatrix대신 대입하면 row 단위로 접근하기 때문에 행렬 계산 시 바로 인접한 원소를 접근하여 응집력이 강해진다. 전치행렬을 추가하는 과정에서 malloc, free, 데이터를 복사하는 overhead가 발생하였음에도 기존보다 실행시간이 적게 걸렸다는 것을 확인할 수 있다.

loop unrolling: for loop과정에서 발생하는 산술연산과 비교연산, branch의 개수가 줄어들기 때문에 실행시간이 줄어든다.



(3) 효과적인 m값

본 실험 결과에는 4개 단위로 풀 때보다 8개 단위로 풀 때의 시간이 더 적게 걸렸다.

(speed up: 1.02)

2) horner's rule로 계산한 결과와 naïve로 계산한 결과는 같지만 실행시간은 horner's rule을 사용한 함수가 더 적게 걸렸다. (Speed up: 10.55)

```
***Polynomial Naive  compute time : 338.812286 ms
***Polynomial Horner compute time : 32.113602 ms
All values are equal
```

CPU에서 곱셈 연산이 덧셈 연산보다 상대적인 부담이 크기 때문에 실행시간을 줄이기 위해서는 곱셈 횟수를 줄여야 한다.

$$y_i = \sum_{i=0}^{n} a_i \cdot x^i$$

naïve의 곱셈 횟수는 $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ 번이다..

```
for (int i = 0; i < n_x; i++) {
    y[i] = a[deg]*x[i]+a[deg-1];//an*x
    for (int j = deg-2; j >= 0; j--) {
        y[i] = y[i] * x[i] + a[j];
    }
}
```

horner's rule은 위와 같은 코드로 구현했는데, naïve와 같은 식을 계산하기 위한 곱셈 횟수는 n(degree)번이다. n이 1보다 큰 자연수일 때 horner's rule이 naïve보다 곱셈 횟수가 적다. 또한 실제 코드에서는 같은 식을 n_x번만큼 계산하기 때문에 실행시간의 차이는 더 벌어질 것이다.

3)

```
*** f<-8.3> = 2.298340e-04
*** f<-8.3> = 2.485168e-04
*** f<-8.3> = 2.485168e-04
```

첫 번째 줄은 float type의 연산을 수행한 것, 두번째 줄은 double type의 연산을 수행한 것, 세번째 줄은 pow()로 e^-8.3의 값을 구한 것이다.

float type으로 계산한 결과와 pow()로 계산한 결과는 소수 첫째 자리부터 다르고, double type으로 계산한 결과와 pow()로 계산한 결과는 소수 6째자리까지 일치했다.

float type 계산이 실제 값과 다른 이유는 나눗셈과 곱셈 연산에서 유효숫자가 적기 때문

에 실제 값과 차이가 생기는 것이다.

```
float e = 1.0+x/n;
for (int i = n-1; i >=1; i--) {
    e = e * x / i + 1.0;
}
return (double)e;
```

float type은 single precision의 계산을 수행하여 유효숫자가 5~7자리이고, double type은 double precision이므로 유효숫자가 15 ~17자리이다.

2. 과제

1)(i) n=1000, init_hw1(0) 일 때 실험을 진행하였다.

```
C:₩Users₩seyoung₩Desktop₩code₩고소실₩10주차₩gososil10_as
====== hw1 ======
hw1_calc_var1 = 0.005100 ms
hw1_calc_var1 value = 0.566828310489655
hw1_calc_var2 = 0.002700 ms
hw1_calc_var2 value = 0.566829085350037
```

두 방법의 계산 결과는 소수 5째자리까지 동일하다. single precision의 유효숫자가 5~7자리이기 때문에 두 방법의 계산 결과는 비교적 정확하다고 할 수 있다.

(ii) n=1000, init_hw1(1)일 때 실험을 진행하였다.

```
C:₩Users₩seyoung₩Desktop₩code₩고소실₩10주차₩gososil10_
====== hw1 ======
hw1_calc_var1 = 0.004400 ms
hw1_calc_var1 value = 4.516868114471436
hw1_calc_var2 = 0.002600 ms
hw1_calc_var2 value = 0.020834824919701
```

두 계산 결과는 상당히 차이가 난다. 이때 두 계산 결과 중 정확한 것은 var2이다.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma_2^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2$$

$$\sigma_1^2 = \frac{1}{n(n-1)}\left(n\sum_{i=1}^{n}x_i^2-\left(\sum_{i=1}^{n}x_i\right)^2\right)$$

var1의 계산 과정에서 비슷한 숫자끼리의 뺄셈이 일어나서 계산결과가 부정확해진다. 밑의 구현한 코드에서 HW1_N * a와 b * b의 뺄셈이 있는데, 두 숫자를 출력해보니 비슷한 값이라는 것을 확인할 수 있다.

```
hw1_var1 = 0.0f;
float a = 0.0f, b = 0.0f;
for (int i = 0; i < HW1_N; i++) {
    a += hw1_x[i] * hw1_x[i];
    b += hw1_x[i];
}
printf("%f %f\n", HW1_N * a, b * b);
hw1_var1 = (float)(HW1_N * a - b * b) / (HW1_N * (HW1_N - 1));
hw1_var1 = sqrt(hw1_var1);
```

```
999939112960.000000 999918731264.000000
```

(iii)각각 n=100000, hw1_init(0)과 hw1_init(1)에서 실험한 결과이다.

```
====== hw1 ======
hw1_calc_var1 = 0.489500 ms
hw1_calc_var1 value = 1.537416577339172
hw1_calc_var2 = 0.339700 ms
hw1_calc_var2 value = 0.577293455600739
```

```
====== hw1 ======
hw1_calc_var1 = 0.245900 ms
hw1_calc_var1 value = 417.945526123046875
hw1_calc_var2 = 0.228500 ms
hw1_calc_var2 value = 3.947556257247925
```

var2가 더 빠르게 분산 값을 계산하는데, var1의 곱셈 횟수가 var2 보다 크기 때문이다.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma_2^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

```
void hw1_calc_var2()
{
    hw1_var2 = 0.0f;
    for (int i = 0; i < HW1_N; i++) {
        hw1_var2 += (hw1_x[i] - hw1_E) * (hw1_x[i] - hw1_E);
    }
    hw1_var2 = hw1_var2 / (HW1_N - 1);

    hw1_var2 = sqrt(hw1_var2);
}
```

var2의 곱셈 횟수는 n번이다.

$$\sigma_1^2 = \frac{1}{n(n-1)}\left(n\sum_{i=1}^{n}x_i^2 - \left(\sum_{i=1}^{n}x_i\right)^2\right)$$

```
void hw1_calc_var1()
{
    hw1_var1 = 0.0f;
    float a = 0.0f, b = 0.0f;
    for (int i = 0; i < HW1_N; i++) {
        a += hw1_x[i] * hw1_x[i];
        b += hw1_x[i];
    }
    hw1_var1 = (float)(HW1_N * a - b * b) / (HW1_N * (HW1_N - 1));
    hw1_var1 = sqrt(hw1_var1);
}
```

var1의 곱셈 횟수는 n+2번이다.

2)

(i) a=1, b=5,c=4일때의 실험결과이다.(x=-1,-4)

```
====== hw2 ======
a, b, c : 1 5 4
naive result    : -1.000000000000000, -4.000000000000000
advanced result : -1.000000000000000, -4.000000000000000

Verifying naive ans    : 0.000000000000000, 0.000000000000000
Verifying advanced ans : 0.000000000000000, 0.000000000000000
```

두 방법 모두 정확하게 실근을 구한다.

(ii),(iii)

-b+root(b^2-4ac)가 비슷한 숫자끼리의 뺄셈이 될 때 문제가 발생한다.

1. a : 1, b : 10000, c : 1

```
====== hw2 ======
a, b, c : 1 10000 1
naive result    : 0.000000000000000, -10000.000000000000000
advanced result : -0.000099999997474, -10000.000000000000000

Verifying naive ans    : 1.000000000000000, 1.000000000000000
Verifying advanced ans : 0.000000000000000, 1.000000000000000
```

2. a : 0.0001, b : 9999, c : 0.0001

```
====== hw2 ======
a, b, c : 0.0001 9999 0.0001
naive result    : 0.000000000000000, -99990000.000000000000000
advanced result : -0.000000010001000, -99990000.000000000000000

Verifying naive ans    : 0.000099999997474, 0.000099999997474
Verifying advanced ans : 0.000000000000000, 0.000099999997474
```

3. a= 0.01 b=5000 c=0.001

```
====== hw2 ======
a, b, c : 0.01 5000 0.001
naive result    : 0.000000000000000, -500000.000000000000000
advanced result : -0.000000200000017, -500000.000000000000000

Verifying naive ans    : 0.001000000047497, 0.001000000047497
Verifying advanced ans : 0.000000000000000, 0.001000000047497
```

세가지 경우 모두 naïve의 f(x) 값이 0이 아니다. advanced에서는 근의 공식을 유리화하여 비슷한 숫자끼리의 뺄셈을 피하고자 하였다. advanced에서 유리화를 한 근 하나는 값이 정확하게 나왔지만, 유리화를 하지 않은 근은 naïve와 마찬가지로 f(x)의 값이 0이 아니다. `(-hw2_b - sqrt(double(hw2_b * hw2_b - 4 * hw2_a * hw2_c)))`

위의 식은 -hw2_b와 -sqrt() 모두 음수이므로 비슷한 숫자끼리의 뺄셈이 아니지만, sqrt() 안의 수가 매우 크기 때문에 sqrt의 계산결과가 정확하지 않게 나온 것으로 추정할 수 있다.

3)

1. recursion 피하기

```
//hw3_1
void hw3_recursion(float i)
{
    if (i < 1000) {
        hw3_ans1_1++;
        hw3_recursion(hw3_ans1_1);
    }
}

void hw3_recursion_2(void)
{
    for (hw3_ans1_2 = 0.0f; hw3_ans1_2 < 1000;) {
        hw3_ans1_2++;
    }

}
```

동일하게 hw3_ans_1_n을 1씩 1000번 증가시키는 과정을 hw3_recursion에서는 recursion으로, hw3_recursion2에서는 반복문으로 구현하였다.

recursion은 추가적인 함수 호출 과정이 발생하므로 recursion대신 for-loop을 사용한 반복문으로 코드를 작성하면 실행시간이 줄어든다. stack overflow가 발생할 위험이 있어 n을 HW3_N대신 1000으로 줄여서 실험을 진행하였다.

debug

```
hw3_recursion = 0.189700 ms
hw3_recursion value = 1000.000000000000000
hw3_recursion_2 = 0.003200 ms
hw3_recursion_2 value = 1000.000000000000000
```

release

```
====== hw3 ======
hw3_recursion = 0.001600 ms
hw3_recursion value = 1000.000000000000000
hw3_recursion_2 = 0.001500 ms
hw3_recursion_2 value = 1000.000000000000000
```

hw3_recursion_2의 수행시간이 hw3_recursion보다 적다. debug에서는 눈에 띄는 차이를 보였으나 release에서는 거의 비슷한 결과가 나왔다. release에서 recursion에 대한 추가적인 코드 최적화가 이루어졌음을 추정할 수 있다. 계산 결과는 두 방법 모두 동일하다.

2. 나눗셈 연산 피하기

```
void hw3_avoid_divide(void)
{
    float a = 100.0f, b = 15.0f, c = 7.0f, d = 13.243f, e = 3.1f;
    hw3_ans2_1 = 0.0f;

    for (int i = 0; i < HW3_N; i++) {

        hw3_ans2_1 += a / b / c / d / e;
    }

}

void hw3_avoid_divide_2(void)
{
    float a = 100.0f, b = 15.0f, c = 7.0f, d = 13.243f, e = 3.1f;
    hw3_ans2_2 = 0.0f;

    b = 1.0 / b;
    c = 1.0 / c;
    d = 1.0 / d;
    e = 1.0 / e;
    for (int i = 0; i < HW3_N; i++) {

        hw3_ans2_2 += a * b * c * d * e;
    }

}
```

hw3_ans_2_n에 a/b/c/d/e만큼 HW3_N번 더하는 과정을 hw3_avoid_divide에서는 나눗셈 연산으로, hw3_avoid_divide_2에서는 a,b,c,d,e의 역수를 구하여 반복문 안에서는 곱셈 연산이 되도록 구현하였다. 따라서 hw3_avoid_divide에서는 반복문마다 나눗셈 연산이 수행되지만 hw3_avoid_divide_2에서는 나눗셈 연산이 반복문 밖에서 한번씩만 수행된다. CPU에서는 곱셈연산보다 나눗셈 연산이 더 느리므로 hw3_avoid_divide2가 hw3_avoid_divide보다 실행시간이 적게 걸릴 것이다.

debug

```
hw3_avoid_divide = 30.356300 ms
hw3_avoid_divide value = 199911.937500000000000
hw3_avoid_divide_2 = 22.802601 ms
hw3_avoid_divide_2 value = 199911.937500000000000
```

release

```
hw3_avoid_divide = 14.935100 ms
hw3_avoid_divide value = 199911.937500000000000
hw3_avoid_divide_2 = 13.784900 ms
hw3_avoid_divide_2 value = 199911.937500000000000
```

hw3_avoid_divide_2의 수행시간이 hw3_avoid_divide보다 적다. 1과 마찬가지로 release일 때는 두 방법의 수행시간 차이가 줄어들었다. 나눗셈 과정에서 추가적인 코드 최적화가 이루어졌다고 추정할 수 있다. 두 방법의 계산 결과는 같다.

3. 매크로(define) 함수 사용

```
void hw3_macro(void) {
    hw3_ans3_1 = 0.0f;
    float a = 5.332f;
    float b = 0.435f;
    float temp;
    for (int i = 0; i < HW3_N; i++) {

        hw3_ans3_1 += foo_2(a,b);
    }
}

void hw3_macro_2(void) {
    hw3_ans3_2 = 0.0f;
    float a = 5.332f;
    float b = 0.435f;
    float temp;
    for (int i = 0; i < HW3_N; i++) {
        temp= foo(a, b);
        hw3_ans3_2 += temp;
    }
}

float foo_2(float a, float b) {
    float temp = a - b;
    temp /= (b + 1);
    temp /= (a + 1.0f);
    return temp;
}
```

```
#define foo(a, b) (((a)-(b)) / ((b)+1) /((a)+1))
```

hw3_ans3_n에 (a-b)/(b+1)/(a+1)를 HW3_N번 더하는 코드를 작성하였다. 수행시간이 오래 걸리게 하여 함수 간의 차이를 보이기 위해 나눗셈 연산을 수행하였다. hw3_macro에서는 계산 과정에서 일반적인 함수인 foo_2를 호출하도록 하였고, hw3_macro_2에서는 #define으로 정의한 foo에서 계산하도록 하였다.

일반적인 함수 호출은 호출 전후로 overhead가 발생한다. 매크로로 함수를 정의하면 컴

파일 과정에서 치환을 하기 때문에 함수 호출 overhead가 발생하지 않아 수행시간이 줄어든다. 다만 함수의 크기가 큰 경우에는 macro가 단순 치환이 되기 때문에 실행 파일의 크기가 늘어날 수 있다는 단점이 있다.

debug

```
hw3_macro = 49.254799 ms
hw3_macro value = 5090881.500000000000000
hw3_macro_2 = 22.930300 ms
hw3_macro_2 value = 5090881.500000000000000
```

release

```
hw3_macro = 11.664200 ms
hw3_macro value = 5090881.500000000000000
hw3_macro_2 = 10.078600 ms
hw3_macro_2 value = 5090881.500000000000000
```

hw3_macro_2의 수행시간이 hw3_macro의 수행시간보다 적다. 두 계산 방식의 결과는 같고, release에서는 두 함수 간의 차이가 줄어들었다. 1,2와 마찬가지로 release에서는 함수 호출 과정의 추가적인 코드 최적화가 일어났음을 추측할 수 있다.

4. loop inversion

```
void hw3_loop_inversion(void) {
    float a = 100.0f, b = 15.0f, c = 7.0f, d = 13.243f, e = 3.1f;
    hw3_ans4_1 = 0.0f;
    int i = 0;
    while (i < HW3_N) {
        hw3_ans4_1 += a / b / c / d / e;
        i++;
    }
}
```

```
void hw3_loop_inversion_2(void) {
    float a = 100.0f, b = 15.0f, c = 7.0f, d = 13.243f, e = 3.1f;
    hw3_ans4_2 = 0.0f;

    int i =0;
    do {
        hw3_ans4_2 += a / b / c / d / e;
        i++;
    } while (i < HW3_N);

}
```

hw3_ans4_n에 a/b/c/d/e를 hw3_N번 더하는 과정을 구현하였다. 수행시간이 오래 걸리게 하여 함수 간의 차이를 보이기 위해 나눗셈 연산을 수행하였다. hw3_loop_inversion은 while문을, hw3_loop_inversion_2는 do while문을 사용하여 구현하였다. int i =0으로 값을 지정하였기 때문에 if문은 생략하였다. hw3_loop_inversion_2는 instruction pipelining에 의해 hw3_loop_inversion보다 수행시간이 적게 걸린다. do while문은 while문에 비해 jump 횟수가 적게 걸리기 때문에 pipeline stall이 걸리는 횟수가 적기 때문이다.

debug

```
hw3_loop_inversion = 30.269400 ms
hw3_loop_inversion value = 199911.937500000000000
hw3_loop_inversion_2 = 30.074499 ms
hw3_loop_inversion_2 value = 199911.937500000000000
```

release

```
hw3_loop_inversion = 12.547400 ms
hw3_loop_inversion value = 199911.937500000000000
hw3_loop_inversion_2 = 10.824000 ms
hw3_loop_inversion_2 value = 199911.937500000000000
```

hw3_loop_inversion_2의 수행시간이 hw3_loop_inversion보다 적게 걸렸고, 두 계산결과의 값은 같다.

5. register 변수 사용

```
void hw3_using_register(void) {
    hw3_ans5_1 = 0.0f;
    float a = 5.332f;
    float b = 0.435f;
    float temp;
    for (int i = 0; i < HW3_N; i++) {
        temp = a - b;
        temp /= (b + 1);
        temp /= (a + 1.0f);
        hw3_ans5_1 += temp;
    }
}
```

```
void hw3_using_register_2(void) {
    hw3_ans5_2 = 0.0f;
    register float a = 5.332f;
    register float b = 0.435f;
    register float temp;
    register int i;
    for (i = 0; i < HW3_N; i++) {
        temp = a - b;
        temp /= (b + 1);
        temp /= (a + 1.0f);
        hw3_ans5_2 += temp;
    }
}
```

3의 foo에서 수행했던 연산을 hw3_using_register에서는 일반적인 float, int형 변수를 사용하여, hw3_using_register_2에서는 register float와 register int 변수를 사용하여 구현하였다. register에 저장된 변수는 메모리에 저장된 변수보다 빠르게 접근할 수 있는데, 특히 loop에서 i는 반복문을 돌 때마다 비교, 덧셈 연산이 수행되므로 register int를 사용하면 수행시간이 빨라질 수 있다.

debug

```
hw3_using_register = 28.191200 ms
hw3_using_register value = 5090881.500000000000000
hw3_using_register_2 = 26.653900 ms
hw3_lusing_register_2 value = 5090881.500000000000000
```

release

```
hw3_using_register = 10.268600 ms
hw3_using_register value = 5090881.500000000000000
hw3_using_register_2 = 9.994500 ms
hw3_lusing_register_2 value = 5090881.500000000000000
```

두 방법의 계산 결과는 같고, hw3_using_register_2의 수행시간이 hw3_using_register보다 적다.