

[CSE 4152] 고급 소프트웨어 실습 I

『고급 프로그래밍 기법』

12주차: 코드 최적화 기법/부동 소수점 연산에 대한 소개

담당교수: 컴퓨터공학과 임인성 (AS-905, 02-705-8493, ihm@sogang.ac.kr)

담당조교: 김영욱 (AS-914, 02-711-5278, kimyu7@sogang.ac.kr)

1 Loop Unrolling 기법 적용을 통한 성능 향상

이번 절의 내용은 [C. Hecker, “PowerPC Compilers: Still Not So Hot”, *Game Developers*, 1996]의 글에 기반을 두고 있다. 아래에 주어진 3행 3열 행렬에 3차원 벡터를 반복적으로 곱해주는 간단한 코드를 살펴보자.

```

:
#include <windows.h>
#define CHECK_TIME_START QueryPerformanceFrequency ((_LARGE_INTEGER*)&freq);
QueryPerformanceCounter(( _LARGE_INTEGER*)&start)
#define CHECK_TIME_END(a) QueryPerformanceCounter(( _LARGE_INTEGER*)&end);
a=(float)((float) (end - start)/freq)

#define NVECTORS 8388608
double *pSVec, *pDVec, pMat[3][3];

// Listing 1
void TransformVectors_L1(double *pDestVectors, const double (*pMatrix)[3],
    const double *pSourceVectors, int NumberOfVectors) {

    int Counter, i, j;
    for (Counter = 0; Counter < NumberOfVectors; Counter++) {
        for (i = 0; i < 3; i++) {
            double Value = 0;
            for(j = 0; j < 3; j++) {
                Value += pMatrix[i][j] * pSourceVectors[j];
            }
            *pDestVectors++ = Value;
        }
        pSourceVectors += 3;
    }
}

void init_MatVec(void) {
    pSVec = (double *) malloc(sizeof(double)*NVECTORS*3);
    pDVec = (double *) malloc(sizeof(double)*NVECTORS*3);
    // initialize the input vectors pointed by pSVec here.
    :
    // initialize the matrix pointed by pMat here.
    :
}

void main(void) {
    _int64 start, freq, end;
    float resultTime = 0;

    init_MatVec();

    CHECK_TIME_START;
    TransformVectors_L1(pDVec, pMat, pSVec, NVECTORS);
    CHECK_TIME_END(resultTime);
    printf("TransformVectors_L1: %f(s).\n", resultTime);
}

```

`TransformVectors_L1(*)` 함수는 `pMatrix`가 가리키는 3행 3열 행렬에 `pSourceVectors`가 가리키고 있는 지점부터 시작하여 저장된 NVECTORS개의 (x, y, z) 벡터를 반복적으로 곱해 그 결과를 `pDestVectors`가 가리키고 있는 메모리 영역에 순서대로 저장해주는 일을 한다. 전반적으로 보아 큰 문제가 없는 잘 짜여진 코드라 할 수 있다. 하지만 코드 최적화 관점에서 보면 성능을 향상시킬 여지가 있어 보인다.

이 함수 대신에 다음 함수를 살펴보자.

```
// Listing 5
void TransformVectors_L5(double *pDestVectors, const double (*pMatrix)[3],
                        const double *pSourceVectors, int NumberOfVectors) {
    int Counter;
    double Value0, Value1, Value2;

    for (Counter = 0; Counter < NumberOfVectors; Counter++) {
        Value0 = pMatrix[0][0] * pSourceVectors[0];
        Value0 += pMatrix[0][1] * pSourceVectors[1];
        Value0 += pMatrix[0][2] * pSourceVectors[2];
        *pDestVectors++ = Value0;
        Value1 = pMatrix[1][0] * pSourceVectors[0];
        Value1 += pMatrix[1][1] * pSourceVectors[1];
        Value1 += pMatrix[1][2] * pSourceVectors[2];
        *pDestVectors++ = Value1;
        Value2 = pMatrix[2][0] * pSourceVectors[0];
        Value2 += pMatrix[2][1] * pSourceVectors[1];
        Value2 += pMatrix[2][2] * pSourceVectors[2];
        *pDestVectors++ = Value2;
        pSourceVectors += 3;
    }
}
```

여기서는 이전 함수에서의 안쪽의 두 개의 for-loop을 단순히 풀어서 코딩을 하였는데, 언뜻 보면 상당히 수준이 낮은 프로그램처럼 보인다. 하지만 실제 8,388,608개의 (x, y, z) 벡터를 3행 3열 행렬에 곱하는 과정을 3.4 GHz Intel Core i7 CPU에서 시간을 측정을 한 결과 각각 0.223184초와 0.116088초가 소요가 되었다. 즉 후자의 함수가 약 두 배정도 빠르다고 할 수 있는데, 어떻게 이러한 현상이 발생하였을까?

이 두 번째 함수는 코드 최적화 기법 중 *loop unrolling* 기법을 적용하여 for-loop를 변환한 예이다. 어셈블러 프로그래밍 과목에서 배운 기계어 수준에서의 for-loop 처리 과정을 생각해보면, 매번 for-loop이 반복될 때마다 약각의 산술 연산과 비교 연산이 수행이 된다. 어떻게 보면 상당히 작은 계산량에 해당한다고 생각할 수 있지만, 이는 단순히 그러한 문제를 넘어서 프로그램의 수행에 부정적인 영향을 미치는 요소로 작용한다. 즉 그러한 계산량도 문제지만 for-loop 안에서의 한 번 수행되는 계산량이 작다면, 이는 다시 말에서 자신의 코드가 수행이 될 때 상당한 빈도로 (conditional) branch가 발생하게 되고, 이는 특히 최근의

고성능 CPU처럼 instruction pipeline을 사용하여 수행속도를 향상시키려는 프로세서의 경우 심각한 문제로 작용하게 된다. 따라서 적절히 규모가 작은 for-loop를 단순히 풀어버림으로서 그러한 문제를 완화시킬 수가 있다.

그리면 과연 규모가 큰 for-loop도 무조건 모두 풀어 상당히 긴 프로그램을 생성해야 할까? 이 경우 컴파일 된 후의 프로그램 크기가 커져 성능에 부정적인 영향을 미치게 되는데, 실제 수행 시간에 영향을 미치는 요소는 매우 많아 이러한 질문에 한 마디로 답을 하기 어렵다. 자신이 적용한 코드 최적화 기법이 얼마나 효과가 있을지는 사용하는 프로세서의 특성에 영향을 많이 받기 때문에, 해당 프로세서의 자세한 메뉴얼을 참조하여 그 특성을 파악해야 한다. 예를 들어, 몇 년전에 많이 사용되었던 어떤 Intel CPU의 경우 일반적으로 for-loop을 8개 정도씩 풀 때 성능이 가장 좋다고 메뉴얼에 기술되어 있는데, 이러한 CPU라면, 예를 들어 8,000천번 반복이 되는 for-loop이라면, 8개씩 풀어 1,000번이 반복되는 for-loop으로 변환하면 될 것이다.

표 1에는 이 두 함수(Listing 1과 Listing 5), 그리고 아래에 주어진 세 번째 함수(Listing 6)를 서로 다른 컴파일러를 사용하여 컴파일한 후 PowerPC 기반의 프로세서에서 수행 시간을 측정한 결과를 보여주고 있다. 비교적 이전 자료이나 현재의 컴퓨팅 환경에서도 시사하는 바가 크다고 할 수 있다.

Table 1. Transform Cycle Counts

Compiler	Listing 1	Listing 2	KAPed 1 (not shown)	Listing 4	Listing 5	Listing 6
CodeWarrior	40.7	50.5	50.9	34.3	29.7	19.6
Symantec C++	76.6	94.9	82.8	50.9	31.9	25.7
Motorola C++	34.5	47.4	39.5	33.2	30.8	20.6
Apple's MrCpp	52.0	65.0	56.2	36.1	28.8	19.5
Microsoft VC++	41.6	49.3	42.8	31.9	21.9	22.7

표 1: 사용 컴파일러와 코드 최적화 정도에 따른 수행 속도 비교 ([C. Hecker, “PowerPC Compilers: Still Not So Hot”, *Game Developers*, 1996]에서 참조).

```
// Listing 6
void TransformVectors_L6(float *pDestVectors, const float (*pMatrix)[3],
    const float *pSourceVectors, int NumberOfVectors) {
    int Counter;
    float Value, _Krr1, _Krr2;

    for (Counter = 0; Counter < NumberOfVectors; Counter++) {
        _Krr1 = pMatrix[0][0] * pSourceVectors[0];
        _Krr2 = pMatrix[1][0] * pSourceVectors[0];
        Value = pMatrix[2][0] * pSourceVectors[0];
        _Krr1 += pMatrix[0][1] * pSourceVectors[1];
        _Krr2 += pMatrix[1][1] * pSourceVectors[1];
        Value += pMatrix[2][1] * pSourceVectors[1];
        _Krr1 += pMatrix[0][2] * pSourceVectors[2];
        _Krr2 += pMatrix[1][2] * pSourceVectors[2];
        Value += pMatrix[2][2] * pSourceVectors[2];
        *pDestVectors++ = _Krr1;
        *pDestVectors++ = _Krr2;
        *pDestVectors++ = Value;
        pSourceVectors += 3;
    }
}
```

컴파일러 제작 기술을 향상으로 인하여 ‘괜찮은’ 컴파일러인 경우 컴파일러가 적지 않은 최적화 작업을 통하여 효율적인 코드를 생성해주나, 프로그래머 입장에서는 가급적 컴파일러가 좋은 코드를 생성할 수 있도록 코딩을 하는 것이 중요하다. 또한 ‘괜찮지 않은’ 컴파일러를 사용할 경우 이러한 프로그래머의 노력은 성능에 큰 차이를 발생시킬 것이다. 아래의 C. Hecker의 글에서 인용한 문단을 읽어보기 바란다.

Why did it make such a big difference? I have no idea, and the only explanation I can come up with is that you need to hold your compiler's hand on any piece of code you care about. The changes I made for Listings 5 and 6 are very obvious (to a human, if not a compiler). I'm basically just stating explicitly where variables are accessed, where possible aliasing can occur, and which variables are constant throughout a loop iteration. These are all things the compiler is supposed to do for us, so we can work on more important stuff, like design and algorithms, or assembly language code for our most inner loops. We're supposed to trust the compiler will do a respectable job, without having to optimize every line of our code (an impossible task for all but the smallest programs).

2 다차원 배열 접근 시 Stride의 크기가 주는 영향

이번에는 다음과 같은 간단한 코드를 살펴보자. 여기서 두 함수 `MinStride_1(*)`과 `MinStride_2(*)`가 계산해주는 내용은 동일하다. 다만 2차원 배열의 각 원소를 계산하는 순서만 다를 뿐이다.

```
:

#define MATDIM2 8192
double MatA[MATDIM2] [MATDIM2], MatB[MATDIM2] [MATDIM2], MatC[MATDIM2] [MATDIM2];

void MinStride_1(double c[] [MATDIM2], double a[] [MATDIM2],
                 double b[] [MATDIM2]) {
    for (int i = 0; i < MATDIM2; i++)
        for (int j = 0; j < MATDIM2; j++)
            c[i] [j] = c[i] [j] + a[i] [j] * b[i] [j];
}

void MinStride_2(double c[] [MATDIM2], double a[] [MATDIM2],
                 double b[] [MATDIM2]) {
    for (int j = 0; j < MATDIM2; j++)
        for (int i = 0; i < MATDIM2; i++)
            c[i] [j] = c[i] [j] + a[i] [j] * b[i] [j];
}

void init_MinStride(void) {
    srand((unsigned) time(NULL));

    for(int i = 0; i < MATDIM2; i++)
        for(int j = 0; j < MATDIM2; j++) {
            MatA[i] [j] = (double) rand() / ((double) RAND_MAX);
            MatB[i] [j] = (double) rand() / ((double) RAND_MAX);
            MatC[i] [j] = (double) rand() / ((double) RAND_MAX);
        }
    }

void main(void) {
    _int64 start, freq, end;
    float resultTime = 0;

    init_MinStride();
    CHECK_TIME_START;
    MinStride_1(MatC, MatA, MatB);
    CHECK_TIME_END(resultTime);
    printf("MinStride_1: %f(s).\n", resultTime);
    printf("%lf\n", MatC[5] [9]);

    init_MinStride();
    CHECK_TIME_START;
    MinStride_2(MatC, MatA, MatB);
    CHECK_TIME_END(resultTime);
    printf("MinStride_2: %f(s).\n", resultTime);
```

```

    printf("%lf\n", MatC[5][9]);
}

```

별 차이가 없어 보이는데도 불구하고, 3.4 GHz Intel Core i7 CPU 상에서 이 두 함수의 수행 시간을 측정해본 결과 놀랍게도 각각 0.265968초와 4.862961초가 소요가 되었다. 어떻게 이러한 일이 발생할 수 있을까? 이는 바로 프로그램을 수행하면서 메모리를 접근하는 패턴이 성능에 지대한 영향을 미칠 수 있다는 좋은 예라 할 수 있다. 잘 알다시피 C/C++ 언어는 2차원 배열을 행 우선(row-major) 방식으로 선형화하여 1차원 메모리에 저장을 한다. 따라서 함수 `MinStride_1(*)`과 `MinStride_2(*)`를 다시 살펴보면, 첫 번째 함수에서는 세 개의 배열에 각각에 대해 순서대로 하나씩 메모리 접근이 일어나고 있는 반면, 두 번째 함수의 경우 각 배열에 대해 매번 8,192·8 바이트 만큼씩 떨어진 지점을 접근함을 알 수 있다.

즉 메모리 접근 시의 지역성(locality) 또는 응집성(coherence)에 관한 문제로서, 컴퓨터 구조 시간에 배운 프로세서의 캐시 메모리의 작동 구조를 고려할 때 능히 그러한 일이 발생할 수 있을 것이라 추측할 수 있다. 즉 전자의 경우가 캐시의 효율을 높여주는 코드라 할 수 있다.(자세한 내용은 본 강의 시 다시 설명). 표 2는 어떤 컴퓨터 시스템의 각 계층별 메모리 접근 시간을 비교하고 있다(만약 여기서 사용하는 CPU의 클락 속도가 1 GHz라면, 어떤 메모리를 접근하는 시간이 10 ns일 경우 10 클락 사이클 동안 기다리고 있어야 한다는 것을 의미함). 이 결과를 보면 자신의 코드가 수행이 될 때 가급적 비용이 작은 메모리를 접근하도록 하는 것이 수행 속도에 큰 영향을 끼칠 수 있음을 알 수 있다. 따라서 반드시 명심해야 할 것은 내가 작성하는 프로그램이 수행될 때 어떠한 패턴으로 메모리에 접근하는지를 항상 염두에 두고, 가급적 공간적 응집성(spatial coherence)과 시간적 응집성(temporal coherence)을 높일 수 있도록 주의를 기울이자.

Level	Access Time	Typical Size	Technology	Managed By
Registers	1-3 ns	?1 KB	Custom CMOS	Compiler
Level 1 Cache (on-chip)	2-8 ns	8 KB-128 KB	SRAM	Hardware
Level 2 Cache (off-chip)	5-12 ns	0.5 MB - 8 MB	SRAM	Hardware
Main Memory	10-60 ns	64 MB - 1 GB	DRAM	Operating System
Hard Disk	3,000,000 - 10,000,000 ns	20 - 100 GB	Magnetic	Operating System/User

표 2: 메모리 계층별 접근 속도 비교 (<http://arstechnica.com/gadgets/2002/07/caching/2/>에서 참조).

지금까지 코드 최적화에 관한 문제에 대하여 극히 간략히 살펴 보았다. 이 문제는 상당

히 범위가 넓으며 적지 않은 경험과 감을 필요로 한다. 요즈음은 인터넷이 좋은 자료가 많이 있으므로 잘 활용하기 바라며, 마지막으로 아래 두 함수의 차이를 비교해 보자.

```
char *Implementation_1(int c) {
    switch(c) {
        case 0: return "EQ"; case 1: return "NE"; case 2: return "CS";
        case 3: return "CC"; case 4: return "MI"; case 5: return "PL";
        case 6: return "VS"; case 7: return "VC"; case 8: return "HI";
        case 9: return "LS"; case 10: return "GE"; case 11: return "LT";
        case 12: return "GT"; case 13: return "LE"; case 14: return "";
        default: return 0;
    }
}

char *Implementation_2(int c) {
    if ((unsigned) c >= 15) return 0;
    return "EQ\ONE\OCS\OCC\OMI\OPL\OVS\OVC\OHI\OLS\OGE\OLT\OGT\OLE\O\O"
           + 3*c;
}
```

3 부동 소수점 숫자의 표현

*IEEE Standards for Floating-Point Arithmetic (IEEE 754)*는 1985년 Institute of Electrical and Electronics Engineers (IEEE)에 제정한 부동 소수점 숫자 (floating-point number)의 표현과 그에 기반을 둔 연산에 대한 표준이다. 대부분의 프로세서가 채택하고 있는 부동 소수점 숫자 및 연산에 대한 기준으로서, 가장 최신 표준은 IEEE 754-2008이다. 여러분이 C/C++ 프로그래밍 시 `float`나 `double` 타입의 변수를 사용할 경우 프로세서 내부적으로는 이 기준에 맞게 숫자가 표현되고 연산이 수행되게 된다. 한 개의 부동 소수점 숫자가 컴퓨터 내부에 표현이 될 때 몇 바이트를 사용하는지는 자신이 어떤 정밀도를 원하는지, 또한 어떤 프로세서를 사용하는지에 따라 다르다. 표 3은 http://en.wikipedia.org/wiki/IEEE_floating_point에서 참조한 자료로서, 각각 2, 4, 8, 그리고 16 바이트를 사용하는 부동 소수점 숫자에 대한 정보를 제공하고 있다(이 외에도 현재 모바일 디바이스에 장착되는 그래픽스 프로세서에서는 3 바이트를 사용하여 부동 소수점 숫자를 표현하기도 한다).

Name	Common name	Base	Digits	E min	E max	Notes	Decimal digits	Decimal E max
binary16	Half precision	2	10+1	-14	+15	storage, not basic	3.31	4.51
binary32	Single precision	2	23+1	-126	+127		7.22	38.23
binary64	Double precision	2	52+1	-1022	+1023		15.95	307.95
binary128	Quadruple precision	2	112+1	-16382	+16383		34.02	4931.77
decimal32		10	7	-95	+96	storage, not basic	7	96
decimal64		10	16	-383	+384		16	384
decimal128		10	34	-6143	+6144		34	6144

표 3: 몇 가지 부동 소수점 숫자 표현의 예.

이제 일반 PC용 프로세서 상에서 `float` 타입의 변수에 해당하는, 4 바이트를 사용하는, single-precision 부동 소수점 숫자의 표현에 대하여 살펴 보자. 임의의 숫자는 다음과 같은 형태로 표현이 가능한데,

$$V = (-1)^S \times (1.f_1 f_2 f_3 \dots)_2 \times 2^E$$

예를 들어, 1.1은 다음과 같이 표현할 수 있다.

$$1.1 = (-1)^0 \times (1.000\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ \dots)_2 \times 2^0$$

여기서 부호를 결정짓는 S 는 V 가 양수일 경우 0 값, 그리고 음수일 경우 1 값을 가지며, f_i 는 모두 0 또는 1 값을 가지는 2진수로서, 정수 값을 가지는 지수 e 를 사용하여, 항상 $(1 \cdots)$ 처럼 정수 부분을 1로 정규화한 형태로 표현이 가능하다. 많은 숫자들이 소수점 이하 숫자들이 무한 반복 되거나, 정확히 표현하기 위하여 상당한 갯수의 이진수 f_i 를 필요로 한다. 그러나 컴퓨터에서는 유한 개의 비트만을 사용하여 숫자를 표현해주어야 하기 때문에 소수점 이하 유한 개의 이진수만 저장하게 된다. 즉, 위의 V 를 다음과 같이 ‘근사적으로’ 표현하여 4 바이트, 즉 32 비트에 저장하게 된다(그림 1 참조).

$$V \approx (-1)^S \times (1.f_1f_2f_3 \cdots f_{23})_2 \times 2^{E-127}$$

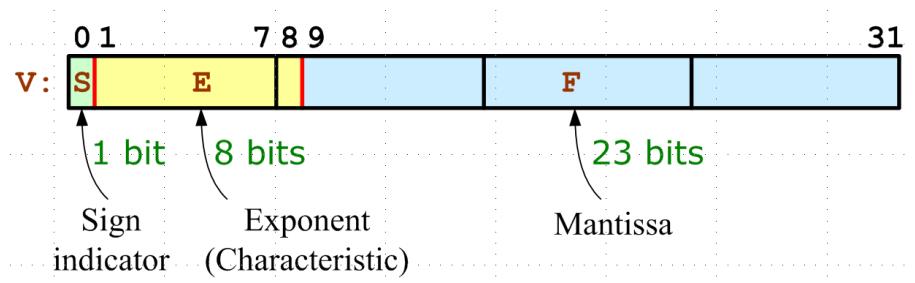


그림 1: Single-precision 부동 소수점 숫자의 표현.

여기서 S 는 앞에서와 동일한 값으로서 한 비트를 사용하여 (Bit 0)에 저장을 하며, 소수점 이하 23개의 이진수 값, 즉 가수(mantissa)를 Bit 9에서 Bit 31까지의 23개 비트 영역에 저장을 한다. 이 23이라는 숫자가 바로 single precision 숫자가 제공할 수 있는 정밀도를 결정하며, 이는 10진수로 따질 경우 6-8자리 정도의 정밀도에 해당한다. 마지막으로 지수(exponent)는 Bit 1부터 Bit 8까지 8개의 비트 영역에 저장을 하는데, 음의 지수도 표현하기 위하여 127 만큼 편향(bias) 시킨 값 E ($0 \leq E \leq 255$)를 저장한다. 따라서 가수 F 에 저장되는 23개의 비트는 다음과 같은 의미를 갖는다: $F = 1 \cdot 2^0 + f_1 \cdot 2^{-1} + f_2 \cdot 2^{-2} + \cdots + f_{23} \cdot 2^{-23}$.

실제로는 지수 E 에 저장되는 값에 따라 위의 표현이 의미하는 바가 그림 2에 도시한 것처럼 세분되는데 자세한 것은 관련 자료를 참조하기 바란다.

이러한 표현 방식을 사용하면 정상적인 상태에서 32비트 부동 소수점 숫자가 표현해줄 수 있는 가장 큰 절대값과 가장 작은 절대값은 다음과 같은데, 자신이 사용하는 프로그래밍

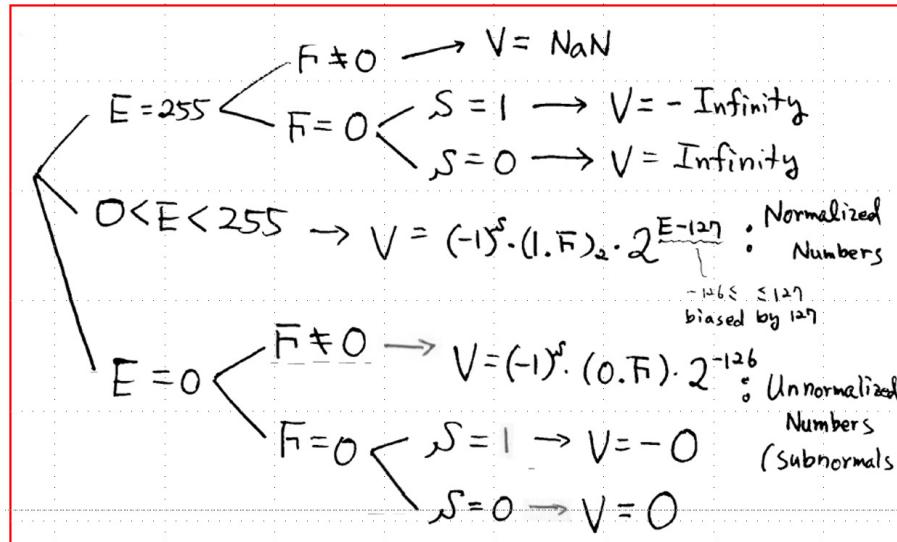


그림 2: 지수 값에 따른 부동 소수점 숫자의 의미.

시스템에서의 최대, 최소 값은 시스템 include 파일인 `float.h`에 정확히 기술되어 있다.

$$V_{max} = (1.11\cdots 1)_2 \cdot 2^{127} = \{(10.00\cdots 0)_2 - 2^{-23}\} \cdot 2^{127} = (1 - 2^{-24}) \cdot 2^{128} \approx 3.4 \cdot 10^{38}$$

$$V_{min} = \begin{cases} (1.00\cdots 0)_2 \cdot 2^{-126} = 2^{-126} \approx 1.8 \cdot 10^{-38} (N) \\ (0.00\cdots 1)_2 \cdot 2^{-126} = 2^{-23} \cdot 2^{-126} = 2^{-149} \approx 1.4 \cdot 10^{-45} (\text{SUBN}) \end{cases}$$

4 비슷한 숫자끼리의 뺄셈 문제

4.1 문제의 원인 및 심각성

앞에서 살펴본 바와 같이 무한개가 존재하는 실수를 유한개의 비트만을 사용하여 표현하려 하니, 제한된 범위의, 그리고 제한된 정밀도를 가지는 실수들만 표현할 수 있게 된다. 그리고 이렇게 ‘부정확한’ 숫자를 유한개의 비트를 사용하여 연산을 하므로 부동 소수점 숫자 및 연산을 사용할 경우 항상 오차가 발생을 하며, 중요한 문제 중의 하나는 부동 소수점 연산을 수행하면서 그 피해를 어떻게 하면 최소화하여 강건한(robust) 소프트웨어를 제작할 것인가 하는 것이다. 가장 널리 알려진 심각한 문제 중의 하나가 **비슷한 숫자끼리의 뺄셈** 시 발생하는 문제이다. 다음과 같은 간단한 프로그램을 살펴보자.

```
void main () {
    float g, x, y, z;

    g = 1.1;
    x = 123456.7890;
    y = x + g;
    z = y - x;
    printf("*** g = %f\n*** z = %f\n", g, z);
}
```

두 말할 것 없이 g와 z 모두 1.1이 출력이 되어야 하는데, 3.4 GHz Intel Core i7 CPU상에서 이 프로그램을 수행시켜 본 결과 다음과 같은 결과를 얻었다.

```
C:\Windows\system32\cmd.exe
*** g = 1.100000
*** z = 1.101563
계속하려면 아무 키나 누르십시오 . . .
```

즉 z 값이 1.1이 아닌 1.101563이 출력이 되었는데, 어떻게 이러한 일이 벌어졌을까? 그 이유는 바로 그림 3을 보면서 한 단계씩 따라 가보면 쉽게 이해할 수 있는데, 우선 다음과 같은 사실을 명심하자.

$$1.1 \approx (-1)^0 \times (1.000\ 1100\ 1100\ 1100\ 1100\ 1101)_2 \times 2^0$$

$$123456.7890 \approx (-1)^0 \times (1.111\ 0001\ 0010\ 0000\ 0110\ 0101)_2 \times 2^{16}$$

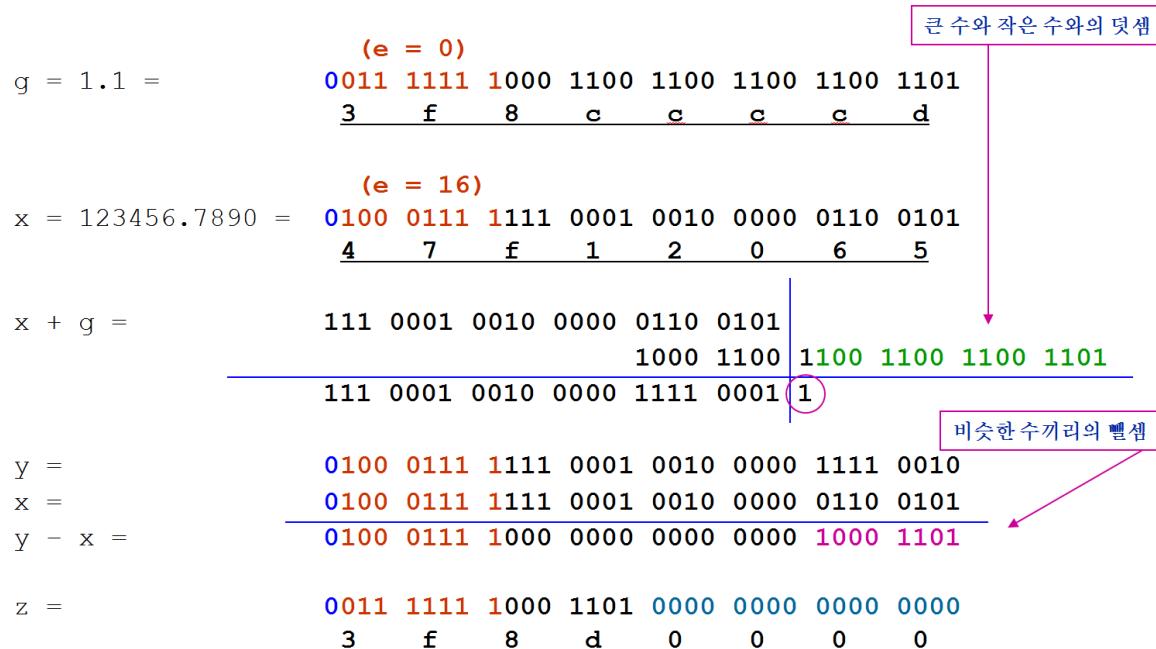


그림 3: 부동 소수점 연산 과정의 예.

$z = y - z$; 문장 수행 시 비슷한 숫자끼리의 뺄셈이 발생하는데, 결과 숫자의 앞 부분이 0이되고(0.00… 100…), 이를 다시 정규화해주는 과정에서 왼쪽 쉬프트가 발생하면서 오른쪽에 아무 의미 없는 0이 삽입되어 그 결과 정밀도가 훼손되게 된다. 이 과정을 이해하였다 면, 두 숫자가 비슷하면 비슷할 수록 문제의 심각성을 커지게 됨을 알 수 있다. 그림 4의 또 다른 간단한 예를 살펴보자.

<pre> float abc, def, ghi; scanf("%f %f", &abc, &def); printf("abc = %20.8e\ndef = %20.8e\n", abc, def); ghi = 1.0 - abc*def; printf("ghi = %20.8e\n", ghi); // 4e-8 ghi = 1.0 - 1.0002*0.9998; printf("ghi = %20.8e\n", ghi); // 4e-8 </pre>	<p style="text-align: center;">Intel Core i7 CPU M620</p> <pre> 1.0002 0.9998 abc = 1.00020003e+000, def = 9.99800026e-001 ghi = -1.96032914e-008 ghi = 3.99999998e-008 Press any key to continue </pre> <p style="text-align: center;">A cheap SHARP calculator</p> <pre> 0.00000004 </pre>
--	--

그림 4: 또 다른 비슷한 숫자끼리의 뺄셈의 예.

4.2 비슷한 숫자끼리의 뺄셈을 피하기 위한 노력

이제 비슷한 숫자끼리의 뺄셈이 야기하는 문제와 그 심각성을 이해하였다 면, 그러한 문제를 최소화하려는 노력이 필요하다. 다음은 비슷한 숫자끼리의 뺄셈을 피함으로서 부동 소수점

연산의 정확성을 높일 수 있는 간단한 몇 가지 예이다.

- $b > 0$ 이고 $b^2 \gg |ac|$ 일 때, 식 $-b + \sqrt{b^2 - 4ac}$ 를 식 $\frac{-4ac}{b + \sqrt{b^2 - 4ac}}$ 으로 대치함.
- $x + \delta, x > 0$ 이고 $|\delta| \ll |x|$ 일 때, 식 $\sqrt{x + \delta} - \sqrt{x}$ 을 식 $\frac{\delta}{\sqrt{x + \delta} + \sqrt{x}}$ 으로 대치함.
- $|\delta| \ll |x|$ 일 때, 식 $\cos(x + \delta) - \cos x$ 을 식 $-2 \sin \frac{\delta}{2} \sin(x + \frac{\delta}{2})$ 으로 대치함.
- $|x| \approx 0$ 일 때, $x - \sin x \rightarrow x - (x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots)$ 으로 대치함.

물론 항상 이렇게 문제를 최소화할 수 있는 것은 아니다. 예를 들어, 컴퓨터 상에서 수치적으로 어떤 함수의 미분값을 계산하는 문제를 고려하자. 종종 주어진 함수 $y = f(x)$ 의 식을 정확하게 모르는 경우가 많은데(예를 들어, 분 단위로 변화하는 주식 지수를 우리가 아는 간단한 수식으로 표현하는 것은 불가능하다), 이 경우 함수 값을 여러 지점에서 샘플링 한 후 이를 이용하여 임의의 x 에서의 미분값을 추정하게 된다. 수치 미분을 하는 방식에는 여러 가지가 있지만, 가장 간단한 방법은 다음과 같으며, 다른 방법도 기본적으로 별반 차이가 크지 않다.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

이는 forward difference라는 방식에 기반을 둔 수치 미분 방법인데, 즉 적절한 h 값을 설정한 후, 미분값을 구하려는 지점 x 와 $x+h$ 에서의 함수값 $f(x+h)$ 와 $f(x)$ 를 샘플링하여 미분값을 추정하게 된다. 문제는 잘 알다시피 이론적으로는 h 값이 작을수록 추정값이 정확한 미분값에 수렴을 하지만, 그러면 그럴 수록 실제 컴퓨터상에서의 계산 시에는 위 식의 분자에서 비슷한 숫자끼리의 뺄셈이 발생하게 된다는 문제가 있다. 따라서 어떠한 h 값을 사용하는가가 수치 미분값의 정확도에 지대한 영향을 미치는데, 실제로 이론적인 오차와 부동 소수점 연산으로 인한 오차의 합을 최소화해주는 h 값을 추정하여 사용하기도 한다.

4.3 부주의한 부동 소수점 연산으로 인한 피해 사례

다음은 부동 소수점 연산 시 부주의한 프로그래밍으로 인한 (또 다른 원인으로 인한) 실제 피해 사례이다. 잘 읽어 보고 앞으로 부동 소수점 연산이 필요한 소프트웨어를 제작할 때에는 항상 언제나 ‘오차’가 발생할 수 있고, 경우에 따라 이는 매우 심각한 문제를 발생 시킬 수 있다는 점을 명심하고, 이를 방지하기 위하여 고급 개발자로서 자신이 생각할 수 있는 모든 노력을 기울이기 바란다.

사례 1: On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi

Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people. ... It turns out that the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors. Specifically, the time in tenths of second as measured by the system's internal clock was multiplied by 1/10 to produce the time in seconds. This calculation was performed using a 24 bit fixed point register. In particular, the value 1/10, which has a non-terminating binary expansion, was chopped at 24 bits after the radix point. The small chopping error, when multiplied by the large number giving the time in tenths of a second, led to a significant error. Indeed, the Patriot battery had been up around 100 hours, and an easy calculation shows that the resulting time error due to the magnified chopping error was about 0.34 seconds (The number 1/10 equals $1/2^4 + 1/2^5 + 1/2^8 + 1/2^9 + 1/2^{12} + 1/2^{13} + \dots$). In other words, the binary expansion of 1/10 is 0.000110011001100110011001100.... Now the 24 bit register in the Patriot stored instead 0.00011001100110011001100 introducing an error of 0.00000000000000000000000011001100... binary, or about 0.000000095 decimal. Multiplying by the number of tenths of a second in 100 hours gives $0.000000095 \times 100 \times 60 \times 60 \times 10 = 0.34$.) A Scud travels at about 1,676 meters per second, and so travels more than half a kilometer in this time. This was far enough that the incoming Scud was outside the “range gate” that the Patriot tracked. Ironically, the fact that the bad time calculation had been improved in some parts of the code, but not all, contributed to the problem, since it meant that the inaccuracies did not cancel.

사례 2: An egregious example of roundoff error is provided by a short-lived index devised at the Vancouver stock exchange (McCullough and Vinod 1999). At its inception in 1982, the index was given a value of 1000.000. After 22 months of recomputing the index and truncating to three decimal places at each change in market value, the index stood at 524.881, despite the fact that its “true” value should have been 1009.811.

사례 3: Other sorts of roundoff error can also occur. A notorious example is the fate of the Ariane rocket launched on June 4, 1996 (European Space Agency 1996). In the 37th second of flight, the inertial reference system attempted to convert a 64-bit floating-point number to a 16-bit number, but instead triggered an

overflow error which was interpreted by the guidance system as flight data, causing the rocket to veer off course and be destroyed.

사례 4: 지난 실습에서 Gaussian Elimination with Scaled Partial Pivoting 방법을 사용하여 선형 방정식 $Ax = b$ 의 근을 구해보았다. 샘플로 제공한 `General_n.txt` 파일의 $n \times n$ 크기의 A 행렬의 원소 a_{ij} 들은 -1과 1 사이의 uniform random number를 발생시켜 설정을 하였고, b 벡터의 원소 b_i 는 근 x 의 모든 원소 x_i 가 1.2가 되도록 설정을 하였다. $n = 16$ 인 경우 위 방법을 사용하여 구한 결과는 다음과 같은데,

```
x[0] = 1.199997, x[1] = 1.200026, x[2] = 1.200008, x[3] = 1.199975,
x[4] = 1.200016, x[5] = 1.200004, x[6] = 1.200013, x[7] = 1.199994,
x[8] = 1.199995, x[9] = 1.200004, x[10] = 1.200018, x[11] = 1.200005,
x[12] = 1.200001, x[13] = 1.199981, x[14] = 1.200006, x[15] = 1.199998
```

일반적으로 5-7 자리 정도의 유효 숫자만 표현할 수 있는 single-precision의 계산을 수행하였다는 사실을 고려하면 비교 정확한 근을 구한 것으로 판단된다.

반면 또다른 샘플로 제공한 `Hilbert_n.txt` 파일의 $n \times n$ 크기의 A 행렬의 원소 a_{ij} 들은 다음과 같이 설정을 하였는데,

$$a_{ij} = \frac{1}{i + j - 1}$$

$n = 16$ 인 경우에 대하여 구한 해는 다음과 같다.

```
x[0] = 1.201568, x[1] = 1.173602, x[2] = 1.260496, x[3] = 1.244426,
x[4] = 1.403662, x[5] = 0.444684, x[6] = 0.752322, x[7] = 1.805505,
x[8] = 2.525773, x[9] = 1.637263, x[10] = 0.707327, x[11] = -0.007822,
x[12] = 0.139721, x[13] = 1.551384, x[14] = 2.224458, x[15] = 1.141567
```

$n = 16$ 인 매우 작은 크기의 선형 방정식에 대해서도 매우 부정확하게 근이 구해졌음을 알 수 있는데, 이러한 형태의 선형 방정식에 대해서는 문제 크기가 커 질수록 더 의미 없는 결과를 얻게 된다. 이는 선형 방정식의 근을 구해주는 수치 알고리즘의 결함보다는 선형 방정식의 A 행렬 자체가 수치적으로 매우 불안정해서 (ill-conditioned) 발생하는 문제로서, 이러한 경우 문제 해결 과정에서 Hilbert 행렬이 나타나지 않도록 해법 자체에 대한 설계를 바꾸거나 하는 등의 노력이 필요하다. 이러한 수치적인 문제가 발생할 수도 있다는 사실을 인지하지 못한

상태에서 소프트웨어를 개발할 때 심각한 문제가 발생할 수가 있기 때문에, 수치 계산에 기반을 둔 문제 해결 과정에서는 항상 그 수치적인 정확도에 문제를 일으킬 수 있는 여러 요인에 대하여 세심한 주의를 기울여야 할 것이다.

5 실습 문제

다음은 코드 최적화 기법 및 부동 소수점 연산에 관한 실습 문제이다.

실습 문제 1

- (i) 다음과 같은 프로토타입을 가지는 함수를 작성하라.

```
MultiplySquareMatrices_1(double *pDestMatrix, double *pLeftMatrix,
                        double *pRightMatrix, int MatSize);
```

이 함수는 `double` 타입의 원소를 가지는 `MatSize` 행 `MatSize` 열 행렬 A 와 B 를 곱해 C 에 저장해주는 역할을 하는데 (즉, $C = A \times B$), A , B , 그리고 C 행렬은 각각 포인터 변수 `pLeftMatrix`, `pRightMatrix`, 그리고 `pDestMatrix`를 통하여 전달 받는다.

본 실험에서는 다음과 같이,

```
#define MATDIM 1024
double *pMatA, *pMatB, *pMatC;
```

$1,024 \times 1,024$ 행렬을 사용하여 실험을 하며, 전역 변수로 세 행렬에 대한 포인터 변수를 정의한 후, 다음 함수를 사용하여

```
void init_MatMat(void) {
    double *ptr;

    pMatA = (double *) malloc(sizeof(double)*MATDIM*MATDIM);
    pMatB = (double *) malloc(sizeof(double)*MATDIM*MATDIM);
    pMatC = (double *) malloc(sizeof(double)*MATDIM*MATDIM);

    srand((unsigned) time(NULL));

    ptr = pMatA;
    for (int i = 0; i < MATDIM*MATDIM; i++)
        *ptr++ = (double) rand() / ((double) RAND_MAX);

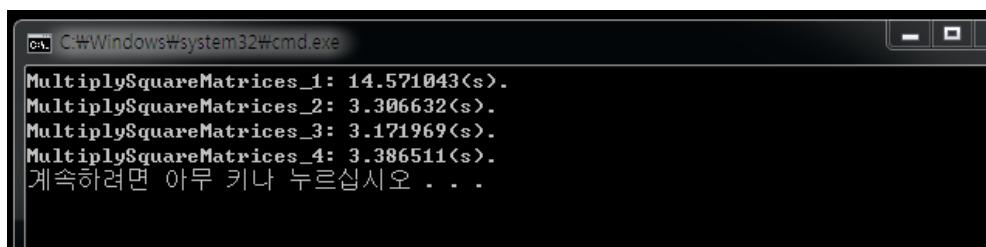
    ptr = pMatB;
    for (int i = 0; i < MATDIM*MATDIM; i++)
```

```
*ptr++ = (double) rand() / ((double) RAND_MAX);  
}
```

세 행렬에 대한 메모리를 할당 받아 A 행렬과 B 행렬을 초기화 한다. 다음, 여러분의 함수에서 다음과 같이 자신이 작성한 함수를 호출하며, 이때 걸리는 시간을 측정한다(측정 방법은 조교가 설명).

```
MultiplySquareMatrices_1(pMatC, pMatA, pMatB, MATDIM);
```

- (ii) 이제 메모리 접근의 응집성을 높혀 행렬 간의 곱셈 계산의 속도를 향상시켜주는 함수 `MultiplySquareMatrices_2(*)`를 작성하라. 이 함수의 인자들은 앞의 1번 함수와 동일하며, 함수 수행 후 A 행렬과 B 행렬의 내용은 함수 수행 직전과 동일해야한다. (힌트: C 행렬의 한 원소 계산 시 A 행렬과 B 행렬의 원소들을 어떠한 순서대로 접근을 하는지, 어떻게 하면 응집성을 가지고 접근하도록 변형할 수 있는지를 잘 고려할 것 ← 전치 행렬!)
- (iii) 위의 `MultiplySquareMatrices_2(*)` 함수를 기반으로 하여 loop unrolling 기법을 적용하여 보자. 이때 가장 안쪽의 for 문을 m 개 단위로 풀어 어떤 m 에 대해 가장 좋은 성능을 내는지 시간을 측정하여 보자. 참고로 다음은 두 개의 서로 다른 m 값을 사용하여 loop unrolling 방법을 적용한 `MultiplySquareMatrices_3(*)` 함수와 `MultiplySquareMatrices_4(*)` 함수를 포함하여 3.4 GHz Intel Core i7 CPU상에서 시간을 측정한 결과를 보여주고 있다.



```
C:\Windows\system32\cmd.exe
MultiplySquareMatrices_1: 14.571043(s).
MultiplySquareMatrices_2: 3.306632(s).
MultiplySquareMatrices_3: 3.171969(s).
MultiplySquareMatrices_4: 3.386511(s).
계속하려면 아무 키나 누르십시오 . . .
```

[주의] 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.

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

$$y = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

다음과 같이 계산하는 방식을 말한다.

$$y = (((a_5x + a_4)x + a_3)x + a_2)x + a_1)x + a_0$$

- (i) DEGREE+1개의 DEGREE차 다항식의 계수 a_i 를 랜덤하게 생성하여 double 타입의 배열 변수 $a[i]$ 에 저장하라 ($i = 0, 1, \dots, \text{DEGREE}$). 다음 N_X개 만큼 x 변수 값을 랜덤하게 생성하여 (0과 1 사이의 값으로), i 번째 값 x_i 를 double 타입의 배열 값 $x[i]$ 에 저장하라 ($i = 0, 1, \dots, \text{N}_X-1$). 이제 각 x_i 에 대해 다음과 같은 식을 계산하려 한다.

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

- (ii) 다음 C/C++ 언어의 수학 라이브러리가 제공하는 $\text{pow}(*, *)$ 함수를 사용하여 각각의 $x[i]$ 에 대해 위에서 설정한 DEGREE차 다항식을 계산하여 double 타입의 배열 값 $y[i]$ 에 저장해주는 다음과 같은 프로토타입의 함수를 작성하라(각 인자의 의미는 분명함).

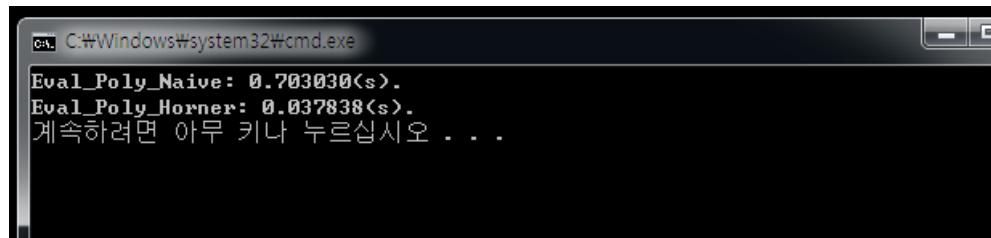
```
void Eval_Poly_Naive(double y[], double x[], int n_x, double a[],
int deg);
```

- (iii) 다음 Horner's rule을 사용하여 위의 함수와 동일한 작업을 해주는 다음과 같은 함수를 작성하라.

```
void Eval_Poly_Horner(double y[], double x[], int n_x, double a[],
int deg);
```

- (iv) 다음 적절한 함수를 작성하여 위에서 각각 두 함수를 통하여 구한 $y[i]$ 값이 같음을 보여라.

- (v) 이제 DEGREE는 10 정도, 그리고 N_X는 1,048,576개 이상 크게 설정한 후, 위 두 함수의 수행 시간을 비교하라. 과연 여러분도 다음과 같은 정도의 차이를 보이는 실험 결과를 얻었는가?



A screenshot of a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The window displays the following text:
Eval_Poly_Naive: 0.703030(s).
Eval_Poly_Horner: 0.037838(s).
계속하려면 아무 키나 누르십시오 . . .

[주의] 각 항목의 실험을 통하여 발견한 사실을 기술하여 프로그램과 함께 제출할 것.

실습 문제 3

이 문제는 부동 소수점 연산 수행 시 가급적 피해야 하는 ‘비슷한 숫자 끼리의 뺄셈’을 다른 형태의 연산으로 대치하는 것에 관한 것이다.

- (i) Taylor series에 기반을 둔 다음 식은 널리 알려진 무한 급수의 합이다.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

이 식에서 주어진 x 에 대하여 앞에서 $n+1$ 개의 항만 사용하여, 즉 $\frac{x^n}{n!}$ 항까지만 사용하여 e^x 값을 근사적으로 계산해주는 함수를 작성하라. 이 함수는 `double` 타입의 x 와 `int` 타입의 n 값을 입력 인자로 받아들여, `double` 타입의 연산을 수행하여 위의 식의 항을 순서대로 더하여 그 결과를 `double` 타입으로 리턴해주어야 한다. 이때 위의 급수의 합을 Horner's method를 사용하여 계산하라. 즉, 예를 들어, $n=3$ 인 경우 다음과 같이 ‘효율적’으로 계산할 것.

$$e^x \approx 1 + \frac{x}{1} \left(1 + \frac{x}{2} \left(1 + \frac{x}{3} \right) \right)$$

- (ii) 이 식의 첫 25개 항을 `float` 타입의 연산을 사용하여 $e^{-8.3} \approx 1 + (-8.3) + \frac{(-8.3)^2}{2} + \dots + \frac{(-8.3)^{24}}{24!}$ 을 계산해보자. 정확한 값은 2.485168×10^{-4} 인데, 여러분의 계산은 얼마나 정확하게 이 값을 구했는가? 만약 적지 않은 오차가 있다면 이러한 결과가 나온 이유는 무엇일까?
- (iii) 그러한 문제를 극복하려면 $e^{-8.3}$ 값을 어떻게 구할지 더 좋은 방법을 제안하고 실험한 후 그 결과를 기술하라. 과연 여러분도 개선된 방법을 사용한 후 다음과 같은 정도의 차이를 보이는 실험 결과를 얻었는가?

```
C:\Windows\system32\cmd.exe
*** f<-8.3> = 4.872486e-003
*** f<-8.3> = 2.485174e-004
계속하려면 아무 키나 누르십시오 . . .
```

[주의] 각 항목의 실험을 통하여 발견한 사실을 기술하여 프로그램과 함께 제출할 것.

6 숙제

제출 마감: ?월 ?일 오후 ?시 정각

제출물 및 방법: 조교가 실습 시간에 공지

이번 주의 숙제는 다음과 같다.

숙제 1

역시 ‘비슷한 숫자 끼리의 뺄셈’의 위험성에 관한 문제이다. 주어진 샘플 데이터 x_i ($i = 1, 2, \dots, n$)에 대하여, 평균 \bar{x} 과 분산 σ_2^2 는 다음과 같이 계산할 수가 있다.

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

이때 분산은 다음과 같은 방법으로도 계산을 할 수가 있다.

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

- (i) 우선 적절히 큰 n 을 선택하여 난수를 발생시켜 적절한 구간의 샘플 데이터 x_i ($i = 1, 2, \dots, n$)를 생성하라. 다음 각각 \bar{x} , σ_2 , 그리고 σ_1 을 계산해주는 함수를 작성하라 (float 타입의 연산을 사용하되, 만약 실험중 double 타입의 연산이 필요하면 이유를 기술하고 사용할 것).
- (ii) 두 분산 값 계산 방법의 결과가 상당히 차이가 나게 해주는 샘플 데이터를 생성한 후, 계산 결과를 비교분석하라. 분산을 어떻게 계산한 것이 더 정확한 것으로 판단되는가? (즉 수학적으로 동일한 두 식이 컴퓨터상에서는 얼마나 다를 수 있는지를 스스로 파악하는 것이 목적이임. 일반적으로 후자의 방법이 더 위험하다고 알려져 있음. 참고로 분산 값은 음수일 수 없음)
- (iii) 충분히 큰 n 에 대하여 두 방법 중 어떤 방법이 더 빠르게 분산 값을 계산하는가?

숙제 2

다음과 같은 이차 방정식 $f(x) = ax^2 + bx + c = 0 (a \neq 0)$ 의 근을 구하는 문제를 생각하자.

- (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 c 값을 적절히 설정한 후, 위에서 구한 두 근을 다시 $f(x)$ 에 대입하여 0이 나오는지 확인함으로서 심각한 문제가 발생하였다는 것을 증명하라.
- (iii) 다음 그러한 문제를 완화시킬 수 있는 방법을 사용하여 위의 2차 방정식을 풀어주는 함수를 새롭게 구현한 후, 위의 문제와 동일한 과정을 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.

숙제 3

전체 이론 강의에서는 Loop Unrolling과 Loop Fusion과 같은 간단한 코드 최적화(Code Optimization) 기법의 적용이 종종 상당히 효과적으로 수행 시간을 단축할 수 있다는 사실에 대하여 살펴보았다. 이러한 최적화 기법에 대한 이해도를 높이기 위하여,

- (i) 자료 조사를 통하여 일반적으로 널리 사용되고 있는 원시 언어 수준(Source Code Level)의 코드 최적화 기법에 대하여 공부를 한 후,
- (ii) Loop Unrolling과 Loop Fusion 기법을 제외한 나머지 기법 중 5개를 선정하여,
- (iii) 각 방법 별로 해당 방법의 효과를 최대한 보일 수 있는 C/C++ 코드를 작성한 후,
- (iv) 해당 기법 적용 전과 적용 후의 시간을 비교한 내용을 보고서에 명기한 후 그러한 결과가 나온 이유를 상세히 설명하라.

[주의]

- 숙제 제출 시 보고서에 각 적용 최적화 기법 별로 코드/실험 결과/분석 자료를 분명하게 기술하라.
- 코드 작성 시 데이터는 C/C++ 언어의 float 타입의 single-precision floating-point number를 사용하고, 모든 계산이 single-precision floating-point operation을 통하여 수행되도록 주의하라.
- Visual Studio에서 플랫폼을 x64로 설정한 후, DEBUG 모드와 RELEASE 모드 각각에 대하여 실험하라.
- 시간 측정은 조교가 실험 시간에 설명하는 방법을 사용하라.
- 크기가 큰 배열을 정의하기 위해서 Visual Studio 상에서 적절히 스택의 크기를 증가 시킬 것. (방법은 조교가 실험 시간에 설명함)
- 코드 최적화 기법 적용 후 계산 결과에 어떠한 변화가 있는지 확인하라. (예를 들어, 그림 6의 실험 결과 예에서 합에 대한 두 결과가 미묘하게 다름을 볼 수 있음) 결과에 조금이라도 차이가 발생할 경우 그 이유를 설명하라.

```

#define TWO_23 (1 << 23)

void loop_fusion(void) {
    float Array_1[TWO_23], Array_2[TWO_23], Array_3[TWO_23], Array_4[TWO_23];
    float sum_1, sum_2, sum_3, sum_4, sum;
    float run_time;

    printf("*****\n");
    printf("*      Loop Fusion      *\n");
    printf("*****\n");

    srand(time(NULL));

    for (int i = 0; i < TWO_23; i++) {
        Array_1[i] = rand() - RAND_MAX / 2.0f; Array_2[i] = rand() - RAND_MAX / 2.0f;
        Array_3[i] = rand() - RAND_MAX / 2.0f; Array_4[i] = rand() - RAND_MAX / 2.0f;
    }
    printf("\nThe problem is to add four arrays of %d elements...%n", TWO_23);

    ///////////////////////////////////////////////////////////////////
    CHECK_TIME_START;
    sum_1 = 0.0f;
    for (int i = 0; i < TWO_23; i++) sum_1 += Array_1[i];

    sum_2 = 0.0f;
    for (int i = 0; i < TWO_23; i++) sum_2 += Array_2[i];

    sum_3 = 0.0f;
    for (int i = 0; i < TWO_23; i++) sum_3 += Array_3[i];

    sum_4 = 0.0f;
    for (int i = 0; i < TWO_23; i++) sum_4 += Array_4[i];

    sum = sum_1 + sum_2 + sum_3 + sum_4;
    CHECK_TIME_END(run_time);

    printf("The runtime using four separate loops is %.3f(ms).\n", run_time * 1000);
    printf("The sum is %e.%n", sum);
    ///////////////////////////////////////////////////////////////////

    ///////////////////////////////////////////////////////////////////
    CHECK_TIME_START;
    sum = 0.0f;
    for (int i = 0; i < TWO_23; i++)
        sum += Array_1[i] + Array_2[i] + Array_3[i] + Array_4[i];

    CHECK_TIME_END(run_time);

    printf("The runtime after loop fusion is %.3f(ms).\n", run_time * 1000);
    printf("The sum is %e.%n", sum);
    ///////////////////////////////////////////////////////////////////

    printf("*****\n");
}

```

그림 5: Loop Fusion 기법에 대한 실험 코드 예.

```
ca C:\WINDOWS\system32\cmd.exe
*****
*      Loop Fusion      *
*****
The problem is to add four arrays of 8388608 elements...
The runtime using four separate loops is 70.835(ms).
The sum is 9.901496e+07.

The runtime after loop fusion is 19.255(ms).
The sum is 9.899250e+07.

*****
계속하려면 아무 키나 누르십시오 . . .
```

그림 6: 그림 5 코드의 수행 결과