CS205 C/C++ Programming - Project 4

Name: 钟元吉(Zhong Yuanji)

SID: 12012613

Part 1 - Analysis

> On the basis of the project 3, improve the multiply method between two matrices by not only SIMD and OpenMP. Also, give a function to generate a radom matrix. Run and compare the time cost on Intel core and Arm core.

本次项目基于第三次项目的矩阵结构体及方法上，通过对代码的进一步改进和优化，尤其是在优化矩阵乘法上，通过直接使用项目三中的已进行一次访存优化的乘法方法、进行两次访存优化的乘法方法、和同时使用SIMD和OpenMP进行同时运算的乘法方法，对这三种方法进行测试、比较运行时间，进行时间复杂度的分析。同时程序中额外为生成随机矩阵的制定了两种方法，通过时间作为种子和通过指定种子生成数值介于`-N`至`N`(N通过传参给定)的随机任意大小的矩阵。以上方法需要同时兼容Intel内核和Arm内核，为了避免代码的重复，并避免在不同系统下程序名称不同的问题，程序对SIMD在Intel与Arm上不同的部分进行宏判断自动识别内核，并进行宏定义上的变化，使方法便于使用。

1.针对Project3的部分改进

数据类型的选取

在第三个Project中由于片面地考虑到设计中的交互式矩阵计算器在使用中通过输入的矩阵格式化字符串不会超过`int`的范围，使用了`int`来存储矩阵的行数和列数。但在本次Project中，由于构建的矩阵中涉及到64k*64k以及可能出现的更大规模的矩阵，其索引值已经超过`int`的数据类型，因此将行数和列数的存储与所有函数中索引值的计算使用`size_t`来表示，考虑到32位系统和64位系统对`size_t`的定义不同，进而单个变量占内存大小不同。我们起初想通过定义全局变量来计算并记录单个矩阵占用的内存大小:

```
const size_t __SIZEM = sizeof(Matrix);
```

但由于新增的函数的声明放在与之前不同的文件中，且都需要使用这个变量，这个声明放在头文件(就算使用了宏防止重复包含)也会在关联`.o`文件时，产生变量重定义的错误，因此我们使用宏定义对32位和64位分别定义：

```
#define __SIZEF __SIZEOF_FLOAT__
#if __SIZEOF_POINTER__ == 4 // 32-bits also __SIZEOF_SIZE_T__ == 4
#define __SIZEM 12
#else // __SIZEOF_POINTER__ == 8 // 64-bits also __SIZEOF_SIZE_T__ == 8
#define __SIZEM 24
#endif
```

进而方便后续对矩阵申请内存。

矩阵的复制

由于在矩阵结构体对象中的数据以基础数据类型的指针存储，不是类对象的指针存储，因此矩阵结构体对象在深度复制过程中，通过`for`循环来复制矩阵中的数据会产生索引值的计算和寻址并消耗较多时间，因此我们可以使用`memcpy`来实现对应内存的深度拷贝，本次改进采用以下代码：

```
memcpy(ret->data, mat->data, row * col * __SIZEF);
```

(对于C++中的某些情况，`memcpy`并不总是最好的方案，如果是类对象的指针存储，`memcpy`将对所存对象的所有内容一字不差地拷贝，如果所存对象的类中有对索引的次数记录的指针，则仅会拷贝其指针，不对索引次数增加，容易产生所存对象的数据内存多次释放的问题，而使用`for`循环来赋值会调用该类的赋值函数，则可以增加索引次数，避免重复释放内存)

2. 矩阵乘法的改进

(1) 使用最基础的`i-j-k`循环的无访存优化方法：

如果把两个矩阵写在纸上，在我们手算矩阵乘法时，我们往往会对乘积后的矩阵逐个位置求值，得到整个矩阵。通过这一中思路，我们可以设计出最简单的、最慢的、无优化矩阵乘法方法，例如将矩阵数据`data`乘上`data2`得到`data_`的代码：

```
for (size_t i = 0; i < row; ++i)
    for (size_t j = 0; j < col2; ++j)
        for (size_t k = 0; k < col; ++k)
            data_[i * col2 + j] += data[i * col + k] * data2[k * col2 + j];
```

这种方法使用我们使用下面公式中的`i、j、k`则我们可以称为`i-j-k`循环矩阵乘法(`col`为第一个矩阵的列数)：

$$(A \times B)_{ij} = \sum_{k=1}^{col} A_{ik} B_{kj}$$

在这种循环顺序中，可以通过使用指针的方式来做到小部分的访存优化，例如可以将结果的第`i`行第`j`列用指针记录下来：

```
float *at = data_[i * col2 + j];
```

并移出关于`k`的循环，在赋值时通过指针`at`来修改，这样可以减少对索引值的计算，进而较小地提高效率，但是每次赋值仍然需要3次从地址中读取数据。由于循环可以交换顺序，我们可以通过交换顺序后的循环，来减少地址中读取数据的次数。

(2) 使用`i-k-j`循环的简单访存优化方法：

我们只需要将循环顺序交换一下，不使关于`k`的循环成为最后一个循环，就可以对`data[i * col + k]`与`data2[k * col2 + j]`之一进行固定，然后放在最后一个循环外，进而减少1次读取数据，例如(这同时是我第三次Projct使用的方法)：

```
for (size_t i = 0; i < row; ++i)
    for (size_t k = 0; k < col; ++k)
    {
        float d1 = data[i * col + k];
        for (size_t j = 0; j < col2; ++j)
            data_[i * col2 + j] += d1 * data2[k * col2 + j];
    }
```

在本次项目中，这种方法成为`matmul_plain`方法。**很多人在做访存优化时到这一步就不再思考了**，但我想知道是否可以将访存优化做到极致呢?

(3) 使用`k-i-j`循环的全面访存优化方法：

前面一种方法仅仅只对赋值式右边第一项进行优化，如果我们对赋值式每一项都进行优化，则有如下代码：

```
for (size_t k = 0; k < col; ++k)
{
    float *d2 = data2 + k * col2;
    for (size_t i = 0; i < row; ++i)
    {
        float d1 = data[i * col + k];
        float *d3 = data_ + i * col2;
        for (size_t j = 0; j < col2; ++j)
            *(d3 + j) += d1 * *(d2 + j);
    }
}
```

在本次项目中，这种方法成为`matmul_improved_sa`方法(sa表示storage access optimization)。这种方法虽然在矩阵乘法赋值式中仍然需要2次读取数据，但通过这种方法减少了索引计算中的乘法次数，在对较大的矩阵进行计算时耗时减少效果明显。

// 如图：

(4) 使用SIMD和OpenMP进行优化：

前面的优化都是对矩阵乘法访存方面的优化，这一类优化只能在一定程度上减轻`for`循环的内容，而无法减少`for`循环的次数。通过使用SIMD中提供的4个或8个浮点数整体运算方法，以及OpenMP提供的`for`循环多核并行命令，可以进一步提高程序效率、减少计算的耗时。

当前我们使用SIMD和OpenMP方法将会遇到以下问题：

- 对于矩阵乘法中第二个矩阵，我们所需的列向量数据索引不是连续的，不能直接进行读取为`__m256`或`float32x4_t`变量
- 对于矩阵乘法中两个矩阵的行、列数不是4或8的倍数，直接读取容易越界到下一行

针对这几个问题，我们又以下解决方案：

- 通过将矩阵乘法中第二个矩阵的所需列向量数据拷贝到临时变量中，得到连续的索引值，这虽然会浪费部分内存，但可以为后续读取提供方便：

```
for (size_t c = 0; c < col2; ++c)
{
    // Copy the line c from oth
    for (size_t r = 0; r < FLOOR; ++r)
        copy2[r] = data2[r * col2 + c];
    ...
}
```

- 当矩阵乘法中两个矩阵的行、列数不是4或8的倍数时，我们可以先向下取到4或8的倍数，将这部分的数据使用SIMD提供的方法计算，其余部分则通过逐个单独计算。选取这种方法而不是置零并继续使用SIMD提供的方法计算，是因为剩下的数据如果读取为`__m256`或`float32x4_t`变量，则需要再次拷贝到临时变量，增加拷贝的时间和内存，而剩下的部分可能不大，而这样导致了更大的计算量，故没有取置零的方法。

3. (伪)随机矩阵的生成

(1) 通过随机种子生成相对固定的(随机)矩阵

在C语言中，我们通常使用`rand()`来生成随机的无符号整数，但是，生成的无符号整数一般介于`0`和`RAND_MAX`(一般是`0x7fff`)之间，即该整数不超过32767。我们知道`float`以科学计数法可以表示的整数部分远远超过这个数值，而且可以表示一定精度的小数，因此我们需要对这样取得的随机数进行一定的变形。

通过指定一个随机的正数的最大值`N`，我们可以通过`(2 * rand() / (__f)RAND_MAX - 1) * N`来生成在`-N`与`N`之间的随机数。

在生成随机矩阵之前，我们可以同过以下两行代码来指定随机数产生的种子：

```
srand(seed); // Get rand by time: seed=(int)time(NULL)
rand();      // Ignore the first one
```

第二行中选择**不记录第一次产生的随机数**，原因是随机数生成器产生的第一个数和种子数有非常大的关联，为了排出第一个数恰好为种子数的可能、防止通过第一个数反推种子进而得到所有伪随机序列、以及与第二种方法对应，我们将弃用第一次产生的随机数。但是由于由用户选取的随机数存在一定的必然性，例如较多人会选取`0,1,2,...,10`等常见数字，进而造成这样生成的矩阵相对较为固定。那么将在下面的方法中生成相对较为随机的矩阵。

(2) 通过时间种子生成(伪)随机矩阵

在这一种方法中矩阵生成时将取与时间相关的种子，这样理论上可以实现每次随机的矩阵都不一样的情况。但是随着我们实践发现，在同一个程序中取随机的两个规模较小且相同的矩阵，得到了完全相同的两个矩阵。这是因为通过随机生成矩阵方法的耗时非常短(在1秒之内)，以至于前后两次通过`time(NULL)`**获取的时间完全相同**，于是**传入了相同的种子**，两个矩阵就完全相同，失去了随机性。

为了避免这种情况发生，我们需要判断获取当前时间是否需要传入作为种子，于是创建了一个全局变量：

```
int __TIME_SEED = -1;
```

当该变量未初始化时，使其初始化为当前时间，否则将其赋值为该时间种子下的第一个随机数，即：

```
if (__TIME_SEED == -1) // Relate with time
    __TIME_SEED = (int)time(NULL);
else // Generate a random seed only depend on time
{
    srand(__TIME_SEED);
    __TIME_SEED = rand();
}
```

这里有两个考虑的地方：我们指定其赋值为自身种子下的第一个随机数是因为考虑到用户可能会更换使用两种产生随机矩阵的方法，因此在实现通过时间种子生成(伪)随机矩阵时，需要先将种子更改为前一次记录值。另外，如果存在两次使用时间对变量`__TIME_SEED`赋值，例如第1次和第n次，则如果前n次随机矩阵规模较小时，第1次和第n次得到的矩阵之间的间隔相等，进而导致两矩阵完全相同，将失去随机性。因此程序除了初始化后第一次赋值为时间外，将其赋值为前一次种子下的第一个随机数，同时又避免了与方法(1)中的矩阵数据(去除第一次随机数)的关联。

Part 2 - Code

文件说明

文件名	内容解释
Matrix.h	Project3-矩阵结构体、宏定义与函数头
MoreFunc.h	Project4-改进矩阵乘法、生成随机矩阵方法
MatrixFunc.c	Project3-矩阵结构体有关的函数
MoreFunc.c	Project4-改进矩阵乘法、生成随机矩阵方法
TestForData.c	Project4-三种矩阵乘法耗时对比

文件名	内容解释
Test.c	Project4-随机矩阵显示、两种矩阵乘法耗时对比

部分代码展示

本次项目中涉及的函数声明：

```
// Multiply without optimization by order i->k->j
int matmul_plain(const Matrix *mat, const Matrix *oth, Matrix *ret);
// Multiply with only storage access optimization by order k->i->j
int matmul_improved_sa(const Matrix *mat, const Matrix *oth, Matrix *ret);
// Multiply with optimization by using SIMD and OpenMP
int matmul_improved(const Matrix *mat, const Matrix *oth, Matrix *ret);
// Generate random matrix with seed from -100 to 100
int rand_matrix_seed(Matrix *ret, __f N, size_t row, size_t col, size_t seed);
// Generate random matrix with time seed from -100 to 100
int rand_matrix(Matrix *ret, __f N, size_t row, size_t col);
```

由于测试代码较长，这里只展示上面函数的定义代码`MoreFunc.c`，测试代码请参考Part 3和源文件`TestForData.c`、`Test.c`。

注：将`float`类型定义为`__f`，如果需要使用`double`等其他类型时，便于转换；通过宏判断识别是否为`Arm`内核，通过宏定义转换在`Intel`内核中的对应函数名，避免对`matmul_improved`函数定义的重复。

```
#include <stdlib.h>
#include <time.h>
#include "../inc/MoreFunc.h"

#include <omp.h>   // openMP
#ifdef __aarch64__ // Arm core
#include <arm_neon.h>
#define __m256 float32x4_t
#define _mm256_setzero_ps() vdupq_n_f32(0)
#define _mm256_loadu_ps vld1q_f32
#define _mm256_add_ps vaddq_f32
#define _mm256_mul_ps vmulq_f32
#define _mm256_storeu_ps vst1q_f32
#define _N_ 4
#else // Intel core
#include <immintrin.h>
#define _N_ 8
#endif

// Variable about random seed
int __TIME_SEED = -1;

// Multiply without optimization by order i->k->j
int matmul_plain(const Matrix *mat, const Matrix *oth, Matrix *ret)
{
    __Check_and_reset_Matrix_ret;
    // Initial and multiply by order i->k->j
```

```
    for (size_t i = 0; i < row; ++i)
        for (size_t k = 0; k < col; ++k)
        {
            __f d1 = data[i * col + k];
            for (size_t j = 0; j < col2; ++j)
                data_[i * col2 + j] += d1 * data2[k * col2 + j];
        }
    return 1;
}
// Multiply with only storage access optimization by order k->i->j
int matmul_improved_sa(const Matrix *mat, const Matrix *oth, Matrix *ret)
{
    __Check_and_reset_Matrix_ret;
    // Initial and multiply by order k->i->j
    for (size_t k = 0; k < col; ++k)
    {
        __f *d2 = data2 + k * col2;
        for (size_t i = 0; i < row; ++i)
        {
            __f d1 = data[i * col + k];
            __f *d3 = data_ + i * col2;
            for (size_t j = 0; j < col2; ++j)
                *(d3 + j) += d1 * *(d2 + j);
        }
    }
    return 1;
}
// Multiply with optimization by using SIMD and OpenMP
int matmul_improved(const Matrix *mat, const Matrix *oth, Matrix *ret)
{
    __Check_and_reset_Matrix_ret;
    size_t FLOOR = (size_t)(col / _N_) * _N_;
    __f copy2[FLOOR];
#pragma omp parallel for
    for (size_t c = 0; c < col2; ++c)
    {
        // Copy the line c from oth
        for (size_t r = 0; r < FLOOR; ++r)
            copy2[r] = data2[r * col2 + c];
        // Calculate the answer at [r,c]
        __f sum[_N_];
        __m256 Prod;
        for (size_t r = 0; r < row; ++r)
        {
            Prod = _mm256_setzero_ps();
            for (size_t j = 0; j < FLOOR; j += _N_)
                Prod = _mm256_add_ps(Prod, _mm256_mul_ps(
                                                _mm256_loadu_ps(data + (r * row +
j)),
                                                _mm256_loadu_ps(copy2 + j)));
            _mm256_storeu_ps(sum, Prod);
            __f *target = data_ + (r * col2 + c);
            for (int i = 0; i < _N_; ++i)
```

```
                *target += sum[i];
            for (size_t k = FLOOR; k < col; ++k)
                *target += data[r * row + k] * data2[k * col + c];
        }
    }
    return 1;
}
// Generate random matrix with seed from -N to N
int rand_matrix_seed(Matrix *ret, __f N, size_t row, size_t col, size_t seed)
{
    if (!ret)
        return -3;
    if (row <= 0 || col <= 0)
        return -5;
    if (!ret->data || ret->row != row || ret->col != col)
    {
        if (ret->data)
            free(ret->data);
        ret->row = row, ret->col = col;
        ret->data = (__f *)malloc(row * col * __SIZEF);
    }
    __f *data_ = ret->data;
    srand(seed); // Get rand by time: seed=(int)time(NULL)
    rand();      // Ignore the first one
    for (size_t i = 0; i < row; ++i)
        for (size_t j = 0; j < col; ++j)
            // data_[i * row + j] = 2 * rand() - RAND_MAX;
            data_[i * row + j] = (2 * rand() / (__f)RAND_MAX - 1) * N;
    return 1;
}
// Generate random matrix with time seed from -100 to 100
int rand_matrix(Matrix *ret, __f N, size_t row, size_t col)
{
    if (__TIME_SEED == -1) // Relate with time
        __TIME_SEED = (int)time(NULL);
    else // Generate a random seed only depend on time
    {
        srand(__TIME_SEED);
        __TIME_SEED = rand();
    }
    return rand_matrix_seed(ret, N, row, col, __TIME_SEED);
}
```