CS205 C/ C++ Program Design - Project 3

Name: Gizele1

Makefile / CMake

cmake是一个跨平台，跨编译器的工具，它的优点在于能够节约大量的存储空间。在整个时代的大背景下，以cmake作为构建工具的项目也逐渐增多，肯定了我们学习cmake的需求。而且由于cmake是一个跨平台的构建工具，他省去了跨平台带来的构建工具不同而带来的麻烦，使得项目有较好的维护性。

对于初级阶段的C/C++学习者来说，掌握使用cmake将源代码文件生成一个可执行的exe文件是一个最基本的操作。如下代码所示，这样，在命令行输入camke . 程序就会自动在该目录下生成Makefile的文件，接着就可以在命令行输入make来编译源代码生成可执行文件。

```
cmake_minimum_required (VERSION 2.6)
project (main)
add_executable(main main.c)
```

而Makefile则是linux系统下C/C++工程编译的工具，它用于自动编译项目。通过编写Makefile文件，可以实现多文件的自动编译。之前我们在运行文件时使用gcc来运行文件的情况下，我们的源文件数量并不多。而当我们处理多个源文件，有时还会遇到一些链接库的问题，需要在命令行手动输入一些参数，而且链接的顺序有时也会成为一种问题。这么一来编译源文件就会成为令人头疼的问题，在开发大型项目和调试过程中这些问题更加突出。而这些问题通过Makefile都可以解决。但是cmake和Makefile的许多语法甚至比C/C++还要复杂，这里就不多再赘述。

C语言中的文件读写

C语言中的几种文件读写方法

1.头文件stdio.h中包含了许多文件读写的函数，例如：

字符读写函数：fgetc和fputc。该函数主要用于向文件中以字符为单位读写函数，每次读/写一个字符，其中fputc函数有以一个返回值，用于判断是否成功写入文件，若成功写入则返回写入字符，若不成功，则返回EOF

字符串读写函数：fgets和fputs。该函数的功能是从一个指定文件中读取字符串存入字符数组中，当遇到'\n'或EOF时停止。fgets（）返回的是字符数组的首地址。

数据块读写函数：freed和fwrite。该函数用于整段数据的读写，该函数的调用形式为fread(buffer,size,count,fp);其中buffer表示存放读入数据数组的首地址，size表示读入数据块的字节数，count表示需要读写的数据块的块数。fp表示文件的指针。fwrite同理。

格式化读写函数：fscanf和fprintf。该函数的优点在于能够格式化的控制输入和输出，使用规则与scanf和prinf类似，区别在于其读写的对象是磁盘文件。

2.通过建立多线程的方法来进行文件的读写。

计算机在执行程序时，需要和许多IO设备进行交互，而这些IO的运行速度会比CPU和内存慢几个数量级。这时候建立多线程的方法，将文件划分为几个区域，让不同的线程进行读取，借助阻塞队列来实现。

几种读写方法的对比

1.在使用fscanf和fprintf时，发现对于其对于浮点数的存储是不够准确的，这与浮点数本身无法精确表示某一特定的小数的性质所决定的，故在计算过程中必然存在一定的误差。

2.对于是否采用多线程也有以下几点考量：

1.一方面线程数量的增多会增加额外的调度，如果线程的数量大于CPU的数量时，会导致计算速度下降。

2.对于多线程的维护也会成为一个问题，线程需要加锁，会带来项目维护成本的增加。

因此，对于任务类型判断成为一个必要，任务类型可以分为计算密集型和IO密集型。对于计算密集型任务，其主要靠CPU的运算能力，消耗CPU的资源，假如对于此任务采用多线程的方法，会使大量时间消耗在线程的切换上，从而降低效率。而对于IO密集型任务，该任务的特点是CPU的计算量很少，而大部分的时间在等待IO。

对于此次要求完成矩阵乘法，也涉及到大矩阵乘法，该任务可归类为计算密集型任务。因经过测试该project运行2048*2048大型矩阵乘法时，读写的时间约占总程序运行时长的8%左右。故大致可判断其为计算密集型任务，故此处不采用多线程的方法进行读写文件。

程序时间的计算

用插入定时器的方法来测量程序运行的总时间，具体操作是在任务的开头和结尾安插标志。该定时器一般不会受到外界的干扰，但是由于定时器需要一定的处理，额外的时间都会导致时间测量的误差。但是由于该误差与矩阵乘法算法运算时间相差较多个数量级，该时间误差基本可以忽略不计。

同时，在考虑是否加入IO时间时，由于该矩阵的数据存储在磁盘中，当矩阵数据量较大时，会消耗大量的时间在文件读写上，对于时间分布可看下表：

	IO时间（五次测试平均值/s）	**程序运行总时间（五次测试平均值/s）**	**IO/程序运行**
32*32矩阵	0.00000	0.00625	0%
256*256矩阵	0.06250	0.475	13.2%
2048*2048矩阵	4.5	207.80	2%

当然，对于不同算法调试的程序运行总时间相差较大，但对于相同磁盘文件的读写时间差别不大。以上可以作为一个大致参考，但当算法运行时间的数量级降至和IO时间数量级左右时，又需要考虑如何优化IO来进行矩阵乘法的优化。

同样，在测试过程中发现每次测试时间略有不同，有时甚至波动较大，误差大概在2%范围之内。 事实上，与操作系统的调度以及现在的CPU支持动态调配有关。在现代系统上，允许多进程并行运行。这说明系统在很多进程中切换。当系统的资源被配置到另一个进程中，根据资源以及特定系统共享资源的方式，进程可能需要花费一段时间来等待资源。这样，每次程序的运行时长都取决于特定系统对任务的处理，处理器的分配以及资源的请求相关因素。而这些因素具有一定的随机性，这导致了 每次程序运行时长的不一致。

Required Function

根据题目中提出的要求，构造了如下几个方法来实现C语言中的矩阵乘法，首先先构造一个矩阵结构体，如下代码所示：

```
struct mat{
int col;
int row;
float *matrix;
};
```

generate方法实现了产生一个struct mat 类型的给定行和列参数以及矩阵数据的方法

```
struct mat generate(int row, int col, float matrix[])
{
    struct mat *mat1;
    mat1 = (struct mat *)malloc(sizeof(struct mat));
    mat1->matrix = (float *)malloc(sizeof(float) * row * col);
    mat1->col = col;
    mat1->row = row;
    mat1->matrix = &matrix[0];
    return *mat1;
}
```

但是，有时我们希望能够随机产生一个mat，故设计了一个给定行和列的参数随机生成矩阵数据的方法，如下代码所示。rand()函数生成的是伪随机数，它产生的值在程序运行之后便为一个定值。通过以时间作为srand()的参数，能够根据系统时间的不同而产生不同的随机数。

```
struct mat generateR(int row, int col)
{
    struct mat *mat1;
    mat1 = (struct mat *)malloc(sizeof(struct mat *));
    mat1->matrix = (float *)malloc(sizeof(float) * row * col);
    mat1->col = col;
    mat1->row = row;
    srand(time(0));
    for (int i = 0; i < col * row; i++)
    {
        *mat1->matrix = (rand() % 100) / 1.2f;
        mat1->matrix++;
    }
    mat1->matrix = mat1->matrix - col * row;
    return *mat1;
}
```

删除矩阵，指针通过malloc申请动态内存由于“堆”有一个特性——由程序自行管理内存，所以在申请了动态内存之后，需要利用free()自行释放，这是为了避免出现野指针，并且把指向这块内存的指针指向NULL，防止之后的程序再用到这个指针。如果不自行释放的话，就会造成内存泄露——可用内存越来越少，设备速度越来越慢。

```
void deletemat(struct mat *mat)
{
    free(mat);
}
```

copymat的方法生成了一个struct mat类型的与输入参数struct mat1内容相同的矩阵

```
struct mat copymat(struct mat mat1)
{
```

```
    struct mat *mat2;
    mat2 = (struct mat *)malloc(sizeof(struct mat *));
    mat2->matrix = (float *)malloc(sizeof(float) * mat1.col * mat1.row);
    mat2->col = mat1.col;
    mat2->row = mat1.row;
    for (int i = 0; i < (mat1.col) * (mat1.row); i++)
    {
        *mat2->matrix = *mat1.matrix;
        mat1.matrix++;
        mat2->matrix++;
    }
    mat1.matrix = mat1.matrix - mat1.col * mat1.row;
    mat2->matrix = mat2->matrix - mat1.col * mat1.row;
    return *mat2;
}
```

mulmat方法是通过传入两个struct类型的矩阵返回一个struct类型的矩阵，该运算思路和C++中矩阵乘法的思路基本相似，但于此不同的是上个project中通过数组下标的移动来获取对应float的值。而此处展示的是通过编写setvalueofmat方法修改对应下标的矩阵数据的值，通过getvalueofmat的方法获得对应下标的矩阵数据的值。具体方法的应用将于下一节具体讨论。

矩阵乘法速度的优化

本次用C语言进行编写矩阵乘法的与上个project实现矩阵乘法最大的不同在于引入了指针。

通过function获取数据和修改数据

一开始采用面向对象的思维，通过构造方法setvalueofmat（），改变传入的struct* mat确定行和列的数据值和构造方法getvalueofmat(),获取传入的struct mat的确定行和列的数据值。此处所说的行和列是指实体矩阵对应的行和列，在存储矩阵的数据时，仍然使用的是一维数据进行存储。而在C语言中的二维数组实际上是在一维数组的基础上，编译器帮助你做了一些映射。但是在面对大型数组时，多位数组连续内存的访问会成为一个问题，这是由于它们与缓存的关联相互作用。会导致多维数组在这种情况下性能下降，故一般默认采用一维数组。

但是，在这个方法中，通过调用方法的方式访问内存空间，方式非常间接，在运行这个矩阵相乘的方法时，操作系统不停地进入、跳出一个方法，时间就会被耗费在进栈与出栈之中了。故采用inline内联函数的方式，对于内存来说，减少了对栈的进出时间的开销，却扩大了主存的空间来容纳本来在栈里的函数。但是测试发现，采用了内联函数后，对于大型矩阵，程序运行时间有所减少，但是浮动不超过1%，没有起到一个很明显的优化效果。

以下是采用内联函数后的测试时间：

	IO时间（五次测试平均值/s）	**程序运行总时间（五次测试平均值/s）**	**IO/程序运行**
32*32矩阵	0.00000	0.015625	0%
256*256矩阵	0.06250	0.475	13.2%
2048*2048矩阵	4.5	207.80	2%

```
struct mat mulmat(struct mat mat1, struct mat mat2)
{
    struct mat *mat3;
    mat3 = (struct mat *)malloc(sizeof(struct mat));
```

```
    mat3->matrix = (float *)malloc(sizeof(float) * mat1.row * mat2.col);

    if (mat1.col == mat2.row)
    {
        mat3->row = mat1.row;
        mat3->col = mat2.col;
        int m = mat1.row;
        int n = mat2.col;
        int p = mat1.col;
        for (int i = 0; i < m; i++)
        {
            for (int k = 0; k < p; k++)
            {
                float r = getvalueofmat(i,k,mat1);
                for (int j = 0; j < n; j++)
                {
                    float temp = getvalueofmat(i, j, *mat3) + r *
getvalueofmat(k, j, mat2);
                    setvalueofmat(i, j, temp, mat3);
                }
            }
        }
    }

    else
    {
        printf("warning!the dimension do not fit.");
    }
    return *mat3;
}

inline float getvalueofmat(int i, int j, struct mat mat)
{
    float value = 0.f;
    int row = mat.row;
    int col = mat.col;
    int index = i * col + j;
    mat.matrix = mat.matrix + index;
    value = *mat.matrix;
    mat.matrix = mat.matrix - index;
    return value;
}

inline void setvalueofmat(int i, int j, float value, struct mat *mat)
{

    int col = mat->col;
    int index = i * col + j;
    mat->matrix = mat->matrix + index;
    *mat->matrix = value;
    mat->matrix = mat->matrix - index;
}
```

通过指针的移动获取数据

该方法与c++中通过数组的下标移动获取数据并无本质区别，只是形式上变成了指针。同样采用了硬件优化，通过降低循环顺序而改变在访问内存中跳跃的次数，通过降低跳跃次数而使得缓存的命中率升高，进而提高CPU的计算效率。出于严谨的考虑，现仍列举一遍为何调整ijk循环次序能够导致，运算速率的大幅提升，约8倍左右。（但其与pro2中的并没有本质区别）

在此我们用访问内存的跳跃数来衡量数据不连续性带给的CPU效率降低。

对于顺序 ikj —— $2n^2+n$ （二维数组）—— n^2 （一维数组）

顺序 kij —— $3n^2$ （二维数组）—— $2n^2$ （一维数组）

顺序 jik —— n^3+2n^2 （二维数组）—— n^3+n^2+n （一维数组）

顺序 ijk —— n^3+n^2+n （二维数组）—— n^3+n^2-n （一维数组）

顺序 kji —— $2n^3+n$ （二维数组）—— $2n^3$ （一维数组）

顺序 jki —— $2n^3+n^2$ （二维数组）—— $2n^3+n^2$ （一维数组）

测试该方法程序运行的时间如下：

该方法对比起上述函数调用的方法，速度提高了5倍左右。

	IO时间（五次测试平均值/s)	程序运行总时间（五次测试平均值/s)	IO/程序运行
32*32矩阵	0.00000	0.00000	/
256*256矩阵	0.087125	0.14687	55%
2048*2048矩阵	4.41875	40.16691	11%

代码如下：

```
struct mat mulmat(struct mat mat1, struct mat mat2)
{
    struct mat *mat3;
    mat3 = (struct mat *)malloc(sizeof(struct mat));
    mat3->matrix = (float *)malloc(sizeof(float) * mat1.row * mat2.col);

    if (mat1.col == mat2.row)
    {
        mat3->row = mat1.row;
        mat3->col = mat2.col;
        int m = mat1.row;
        int n = mat2.col;
        int p = mat1.col;
        for (int i = 0; i < m; i++)
        {
            for (int k = 0; k < p; k++)
            {
                float r = mat1.matrix[i * n + k];
                for (int j = 0; j < n; j++)
                {
```

```
                    mat3->matrix[i * n + j] += r * mat2.matrix[k * n + j];
                }
            }
        }
    }
    else
    {
        printf("warning!the dimension do not fit.");
    }
    return *mat3;
}
```

-O3编译优化

对比起Pro2中采用硬件优化以及`Strassen`算法的方式对矩阵乘法的速度进行提升，现采用-O3编译优化的方式来提高矩阵乘法的运行速度。

从一个源文件到可执行文件经历了以下几个步骤：

C源程序 - >编译预处理 - >编译 - >汇编程序 - >链接程序 - >可执行文件

其中预处理是指对其中的伪指令（以#开头的指令）和特殊符号进行处理。其中包括宏定义指令，条件编译指令和加载头文件。预处理是将.c文件转化为.i文件。

在编译阶段需要进行三个步骤，包括词法分析、语法分析和语义分析。编译完成后会生成.o文件。

接下来进行的汇编过程就是将汇编语言翻译成机器能够看懂的语言，转化为机器语言代码。

链接就是将不同部分的代码和数据收集和组合成为一个单一文件的过程,这个文件可被加载或拷贝到内存中执行。接着就是生成可执行文件。

-O1,-O2,-O3是用于优化编译过程的，其中O1 提供基础级别的优化，-O2提供更加高级的代码优化,会占用更长的编译时间，-O3提供最高级的代码优化。

对比O1，O2的优化选项会牺牲部分编译速度，除了完成了O1所有的优化之外，还会调用所有支持优化的算法，从而实现总体运行速度的优化。

对比起O2，O3除了执行O2中所有的优化以外，一般是采取向量化的算法来提高程序运行的速度。

测试几组数据时间如下：

	IO时间（五次测试平均值/s）	**程序运行总时间（五次测试平均值/s）**	**IO/程序运行**
32*32矩阵	0.00000	0.00000	/
256*256矩阵	0.062500	0.140626	44.4%
512*512矩阵	0.328125	0.843750	38.9%
1024*1024矩阵	1.234375	5.687500	21.70%
2048*2048矩阵	4.000000	37.656250	10.62%

速度对比起-O3编译优化前提高了108%。但是查阅相关资料，一些数据显示在开启O3优化后能够提升4-5倍的速度，但是此处并没有得到明显提升，故此处存疑。

OpenBLAS的应用

openblas 是一个开源的矩阵计算库，包含了诸多的精度和形式的矩阵计算算法。它包括标量、矢量、矩阵三个层面的运算。OpenBLAS 支持 Fortran 和 C 两个版本，我们所关注的C版本，对应头文件为 `cblas.h`。

对于矩阵乘法的实现，在朴素算法中，当矩阵的规格越大时，运算的效率会大幅降低。这是由于CPU的三层缓存机制所导致的，当缓存中储存不下大量数据时，OS就会到内存中去取数据，导致性能下降。

而在openBLAS中实现矩阵乘法是通过 `gemm` ，其主要原理是通过切分矩阵，将大型矩阵切割成小型矩阵，使得尽可能多的数据被加载到缓存区域中，这样就能够提高效率了。这样一个通过数据重排的操作使得数据仍然是连续的，提高了缓存的命中率，其中从网站上查找到的较为清晰解释这个问题的伪代码如下:

其中 `Copy（）` 操作是一个数据重排的操作，而 `kernel_op（）` 的操作是也是一个矩阵乘法的操作。

```
    For m = 0:M, step = R
        For k = 0:K, step = Q
            For n = 0:N, step = P
                Copy_B(Q,P)
                For mm = 0:R, step = UN
                    Copy_A(UN,Q)
                    kernel_op(A(UN,Q), B(Q,P))
                Endfor
            Endfor
        Endfor
    Endfor
```

在大致了解了openBLAS之后，在linux下进行下载。下载完成后进行编译，测试了它的运算速度如下表所示:

	IO时间（五次测试平均值/s）	**程序运行总时间（五次测试平均值/s）**	**IO/程序运行**
32*32矩阵	0.00000	0.00000	/
256*256矩阵	0.7375	0.7531	97%
512*512矩阵	2.125	2.125	100%
1024*1024矩阵	3.34375	3.46875	96%
2048*2048矩阵	6.8437	7.9531	86%

对比起通过算法实现的矩阵乘法，应用openBLAS实现矩阵乘法程序运行时间中IO的时间占比较高，说明其CPU做计算所消耗的时间非常短，可以通过将两者时间相减来进行大致估量。这就回到了最初讨论文件读写的方法上，现在主要矛盾是IO消耗时间较大，可以运用异步IO或多线程的方法降低时间，但由于时间有限，我在此处并没有实现此功能。

Code

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "function.h"

float a[2048 * 2048] = {0};
float b[2048 * 2048] = {0};
float d[2048 * 2048] = {0};
int read_row(FILE *fp);
int read_col(FILE *fp);

int main(int argc, char *argv[])
{
   clock_t start, finish, finish1, start1;
   double duration, duration1;
   FILE *f1;
   FILE *f2;
   int col1, row1, col2, row2 = 0;
   //f1文件读入
   start = clock();
   f1 = fopen(argv[1], "r");
   if (f1 == NULL)
      perror("Error opening file");
   else
   {
      col1 = read_col(f1);
      row1 = read_row(f1);
      while (!feof(f1))
      {
         for (int i = 0; i < row1 * col1; i++)
         {
            fscanf(f1, "%f", &a[i]);
         }
      }
      fclose(f1);
   }

   //f2文件读入
   f2 = fopen(argv[2], "r");
   if (f2 == NULL)
      perror("Error opening file");
   else
   {
      col2 = read_col(f2);
      row2 = read_row(f2);
      while (!feof(f2))
      {
         for (int i = 0; i < row2 * col2; i++)
         {
            fscanf(f2, "%f", &b[i]);
         }
      }
      fclose(f2);
   }
   finish1 = clock();

   //function generate 的测试
```

```
    struct mat mat1 = generate(row1, col1, a);
    struct mat mat2 = generate(row2, col2, b);
    struct mat mat3 = mulmat(mat1, mat2);

    // //function generateR 的测试
    // struct mat matR = generateR(2,2);
    // printf("the col of matR is:%d, and the row of matR is:%d
\n",matR.col,matR.row);
    // for(int i = 0;i < 4;i++){
    //    printf("%f \n",*matR.matrix);
    //    matR.matrix ++;
    // }
    // matR.matrix = matR.matrix -4;

    //function delete 的测试
    // struct mat * p = &mat1;
    // deletemat(p);
    // printf("the col of mat1 is:%d, and the row of mat1 is:%d
\n",mat1.col,mat1.row);

    // //function copy 的测试
    // struct mat mat3 = copymat( mat2);
    // printf("the col of mat3 is:%d, and the row of mat3 is:%d
\n",mat3.col,mat3.row);
    // for(int i = 0;i < 4;i++){
    //    printf("%f \n",*mat3.matrix);
    //    mat3.matrix ++;
    // }
    // mat3.matrix = mat3.matrix-4;

    // //function getvalue 的测试
    // printf("here is the test for getvalue,mat3[3] is
%f\n",getvalueofmat(1,1,mat2));
    // //function setvalue 的测试
    // struct mat * p2 = &mat2;
    // setvalueofmat(0,0,0.4433f,p2);
    // printf("here test for setvalue function, now mat2[0] is
%f\n",*mat2.matrix);

    //f3文件的写出
    start1 = clock();
    FILE *f3 = fopen(argv[3], "wb+");
    if (f3 == NULL)
       perror("Error opening file");
    else
    {
       for (int i = 0; i < row1; i++)
       {
          for (int j = 0; j < col2; j++)
          {
             fprintf(f3, "%f ", *mat3.matrix);
             mat3.matrix++;
          }
          fprintf(f3, "\n");
       }
    }
    fclose(f3);
    finish = clock();
```

```
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    duration1 = (double)(finish1 - start + finish - start1) / CLOCKS_PER_SEC;
    printf("running time is %f seconds\n", duration);
    printf("the IO time is %f seconds\n", duration1);
}

int read_row(FILE *fp)
{
    int c = 0;
    int row = 0;
    while ((c = fgetc(fp)) != EOF)
    {
        if (c == '\n')
        {
            row++;
        }
    }
    row++;
    rewind(fp);
    return row;
}

int read_col(FILE *fp)
{
    int c = 0;
    int col = 0;
    while ((c = fgetc(fp)) != '\n')
    {
        if (c == ' ')
        {
            col++;
        }
    }
    col++;
    rewind(fp);
    return col;
}
```

function.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "function.h"

struct mat generate(int row, int col, float matrix[])
{
    struct mat *mat1;
    mat1 = (struct mat *)malloc(sizeof(struct mat));
    mat1->matrix = (float *)malloc(sizeof(float) * row * col);
    mat1->col = col;
    mat1->row = row;
    mat1->matrix = &matrix[0];
    return *mat1;
}

struct mat generateR(int row, int col)
```

```
{
    struct mat *mat1;
    mat1 = (struct mat *)malloc(sizeof(struct mat *));
    mat1->matrix = (float *)malloc(sizeof(float) * row * col);
    mat1->col = col;
    mat1->row = row;
    srand(time(0));
    for (int i = 0; i < col * row; i++)
    {
        *mat1->matrix = (rand() % 100) / 1.2f;
        mat1->matrix++;
    }
    mat1->matrix = mat1->matrix - col * row;
    return *mat1;
}

void deletemat(struct mat *mat)
{
    free(mat);
}

struct mat copymat(struct mat mat1)
{
    struct mat *mat2;
    mat2 = (struct mat *)malloc(sizeof(struct mat *));
    mat2->matrix = (float *)malloc(sizeof(float) * mat1.col * mat1.row);
    mat2->col = mat1.col;
    mat2->row = mat1.row;
    for (int i = 0; i < (mat1.col) * (mat1.row); i++)
    {
        *mat2->matrix = *mat1.matrix;
        mat1.matrix++;
        mat2->matrix++;
    }
    mat1.matrix = mat1.matrix - mat1.col * mat1.row;
    mat2->matrix = mat2->matrix - mat1.col * mat1.row;
    return *mat2;
}

struct mat mulmat(struct mat mat1, struct mat mat2)
{
    struct mat *mat3;
    mat3 = (struct mat *)malloc(sizeof(struct mat));
    mat3->matrix = (float *)malloc(sizeof(float) * mat1.row * mat2.col);

    if (mat1.col == mat2.row)
    {
        mat3->row = mat1.row;
        mat3->col = mat2.col;
        int m = mat1.row;
        int n = mat2.col;
        int p = mat1.col;
        for (int i = 0; i < m; i++)
        {
            for (int k = 0; k < p; k++)
            {
                float r = mat1.matrix[i * n + k];
                for (int j = 0; j < n; j++)
```

```
                {
                    mat3->matrix[i * n + j] += r * mat2.matrix[k * n + j];
                }
            }
        }
    }
    else
    {
        printf("warning!the dimension do not fit.");
    }
    return *mat3;
}

inline float getvalueofmat(int i, int j, struct mat mat)
{
    float value = 0.f;
    int row = mat.row;
    int col = mat.col;
    int index = i * col + j;
    mat.matrix = mat.matrix + index;
    value = *mat.matrix;
    mat.matrix = mat.matrix - index;
    return value;
}

inline void setvalueofmat(int i, int j, float value, struct mat *mat)
{

    int col = mat->col;
    int index = i * col + j;
    mat->matrix = mat->matrix + index;
    *mat->matrix = value;
    mat->matrix = mat->matrix - index;
}
```

function.h

```
#pragma once
struct mat{
int col;
int row;
float *matrix;
};
struct mat generate(int row,int col,float matrix[]);
struct mat generateR(int row,int col);
void deletemat(struct mat *mat);
struct mat copymat( struct mat mat1);
struct mat mulmat(struct mat mat1,struct mat mat2);
float getvalueofmat(int i,int j,struct mat mat);
void setvalueofmat(int i,int j,float value,struct mat* mat);
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.16)

project(pro)

aux_source_directory(. DIR_SRCS)


SET(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O3 -Wall")

add_executable(pro ${DIR_SRCS})
```

Reference

参考网址：

1.C语言文件的读写_C语言中文网 (biancheng.net)

2.(33条消息) C/C++快速读写磁盘数据的方法_lusic01的专栏-CSDN博客

3.(33条消息) OpenBLAS学习一：源码架构解析&GEMM分析_frank2679的专栏-CSDN博客

4.(33条消息) -O1,-O2,-O3编译优化知多少*xinianbuxiu*的博客-*CSDN*博客o2优化