CPP_Project4_Matrix_SpeedUp@Jiko

- **Name：纪可鸣**
- **SID：12112813**

仓库网址： https://github.com/JikoSchnee/Cpp_Project4_Matrix_SpeedUp

写在前面

一看到要优化速度，我开始便兴致勃勃地先去找有没有优化的算法，一眼相中斯特拉森算法啃了好久，结果却很难使人满意，并且过程中对内存管理的要求非常冗杂，“性价比”不高。做完整个project后发现硬件优化、O3优化、SIMD指令集优化和OpenMP优化都比算法“性价比”高得多，如此一来我发现这几种优化真的非常管用，几乎可以广泛地用在任何的project上，而且在面对数据非常大的运算时能够获得很理想的优化。而啃斯特拉森算法的过程，以及看于老师对这个project点评的视频过程中对内存管理也有了更深刻的认识，可以说此次project受益匪浅了^_^。

---

*※为方便比较，以硬件优化后的乘法为基础，为了比较更明显，每个模块不会同时使用所有优化方法。在**综合优化**模块中综合前面提到的优化方法提供一个improved方案。*

目录

硬件优化

CPU访问优化

因为CPU访问内存的速度比CPU计算速度慢得多，所以CPU访问内存的优化尤为重要。

人工计算矩阵，一般地是从左往右，从上到下依次计算出结果矩阵每一位的数，如pro3中的代码：

```
for (size_t i = 0; i < mat1->row; ++i) {
    for (size_t j = 0; j < mat2->column; ++j) {
        array[indexResult] = 0;
        for (size_t k = 0; k < mat1->column; ++k) {
            array[indexResult] += mat1->data[i * mat1->column + k] * mat2->data[j
+ mat2->column * k];
        }
        indexResult++;
    }
}
```

但是这样使得mat2->data的读取变得不连续，每次读取都需要跳跃至少`column \* sizeof(float)`个字节，那么是否能通过交换顺序使得内存的访问变得更加连续呢?

```
for (int i = 0; i < r; ++i) {
            for (int k = 0; k < r; ++k) {
                tem = mat1->data[i * r + k];
                for (int j = 0; j < r; ++j) {
                    array[i * r + j] += tem * mat2->data[k * r + j];
                }
            }
        }
```

如此一来在mat1和mat2data上的读取相比原先方案更加连续。

数据测试

`matmul_plain`:

	1	2	3	平均
128×128	8	12	8	9.333333
1k×1k	4325	4495	4227	4349

`硬件优化`：

	1	2	3	平均
128×128	7	8	6	7
1k×1k	2664	2767	2613	2681.333333

可以看到当矩阵足够大时优化效果非常显著。

O3优化

通过增加编译代码的代码量来换取运行时间的缩短。

代码实现

cmake

添加 `set(CMAKE_C_FLAGS "-O3")`：添加O3编译选项。

.c

添加 `#pragma gcc optimize(3)`

数据测试

硬件优化：

	1	2	3	平均
128×128	7	8	6	7
1k×1k	2664	2767	2613	2681.333333

`O3优化+硬件优化`：

	1	2	3	平均
128×128	1	1	1	1
1k×1k	154	147	154	151.666667
8k×8k	131834	151079	143843	142252

可以发现使用O3优化后运行速度是飞一般的增长，并且也是初次在可控时间内跑出了8k级别的矩阵乘法。

OpenMP优化

理论操作起来比较简单，只要`#include "omp.h"`之后再在for上面一行加上`#pragma omp parallel for`即可使多个cpu并行计算，但是实际测速却发现是3.147000左右，明显大于不并行计算，原来是因为多个cpu同时工作使得clock()重复计算了多次时间，因此在这里决定将所有clock()更换为ftime()来计时。

加上OpenMP后发现并没有任何提升，一番查找后才知道原来不仅要在头文件中include，还需要在cmake中调用OpenMP的包：

```
FIND_PACKAGE( OpenMP REQUIRED)
if(OPENMP_FOUND)
    message("OPENMP FOUND")
    set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
endif()
```

硬件优化：

	1	2	3	平均
128×128	7	8	6	7
1k×1k	2664	2767	2613	2681.333333

`硬件优化+OpenMP`：

	1	2	3	平均
128×128	2	2	2	2
1k×1k	528	539	571	546

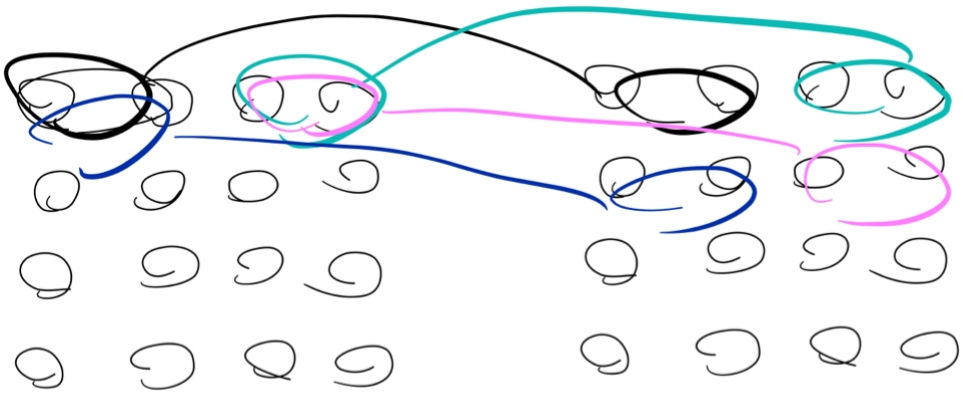
`硬件优化+OpenMP+O3`：

	1	2	3	平均
1k×1k	33	34	32	33
8k×8k	52260	55465	53345	53690

可以看到使用OpenMP优化后花费时间又降了一大层级。

SIMD优化

将第二个矩阵转置，使得寄存器读取数据的时候访问是连续的，然后八个八个地把数据存入寄存器，做乘积，再依次放入存放结果的数组中：



```
float * tem= NULL;
tem = malloc(8*sizeof (int ));
__m256 load1 = _mm256_setzero_ps();
__m256 load2 = _mm256_setzero_ps();
__m256 sum = _mm256_setzero_ps();
size_t index = 0;
size_t row = 0;
size_t i = 0,j = 0;
while (i<mat1->row) {
    for (int k = 0; k < mat1->column; k+=8) {
        load1 = _mm256_loadu_ps(&mat1->data[i*mat1->column+k]);
        load2 = _mm256_loadu_ps(&mat3->data[j*mat3->column+k]);
        sum = _mm256_mul_ps(load1,load2);
        _mm256_storeu_ps(tem, sum);
        for (int l = 0; l < 8; ++l) {
            array[i*c+j] += tem[i];
        }
    }
    j++;
    if (j>=mat3->row){
        j = 0;
        i++;
    }
}
```

数据测试

`matmul_plain`：

	1	2	3	平均
128×128	8	12	8	9.333333
1k×1k	4325	4495	4227	4349

`avx+O3`：

	1	2	3	平均
128×128	7	6	6	6.333333
1k×1k	2641	2734	2679	2684.666667

硬件优化+O3+openMP：

	1	2	3	平均
1k×1k	33	34	32	33
8k×8k	52260	55465	53345	53690

`avx+O3+openMP` （openMP提升不显著）：

	1	2	3	平均
128×128	6	7	6	6.333333
1k×1k	2640	2643	2789	2684.666667

算法优化

Strassen算法

在网上搜索矩阵乘法的优化算法，映入眼帘的首先就是**Strassen算法**。

> *斯特拉森算法把原先普通二阶矩阵相乘需要的8次乘法压缩到7次乘法，而在计算机，乘法运算的耗时远远高于加减运算，所以斯特拉森算法可以将O(d^ 3) 压缩到O(d^2.8)。*
>
> *需要知道的是，斯特拉森算法只是对矩阵分治的算法而不是单独的乘法算法，分治完成时最后使用的还是普通矩阵乘法，在阶数小于等于32（或者64？看过不同的实验结果）时普通的矩阵乘法会有更快的速度，而随着矩阵的阶不断增加，斯特拉森可以提供更快的速度。*

本质为分治思想，将矩阵乘法不断迭代直至目标矩阵阶数小于32或64。

※拓展到其他矩阵：使用补零法向下向右补零，使得矩阵为偶数阶方阵。

```
matrix *strassen(const matrix *mat1, const matrix *mat2) {
    if (mat1->row != mat2->column)return matmul_improved(mat1, mat2);
    size_t row1 = mat1->row;
    size_t row2 = mat2->row;
    size_t column1 = mat1->column;
    size_t column2 = mat2->column;
    if (row1 % 2 != 0 || column1 % 2 != 0)return matmul_improved(mat1, mat2);
    if (mat1->column <= 32)return matmul_improved(mat1, mat2);
    size_t newSize = mat1->row / 2;
    size_t rXc = row1 * column2;
    float *data1 = mat1->data;
    float *data2 = mat2->data;
    float *a11 = malloc(rXc * sizeof(float));
```

```
    float *a12 = malloc(rXc * sizeof(float));
    float *a21 = malloc(rXc * sizeof(float));
    float *a22 = malloc(rXc * sizeof(float));
    float *b11 = malloc(rXc * sizeof(float));
    float *b12 = malloc(rXc * sizeof(float));
    float *b21 = malloc(rXc * sizeof(float));
    float *b22 = malloc(rXc * sizeof(float));
    matrix *c11 = NULL;
    matrix *c12 = NULL;
    matrix *c21 = NULL;
    matrix *c22 = NULL;

    matrix *m1 = NULL;
    matrix *m2 = NULL;
    matrix *m3 = NULL;
    matrix *m4 = NULL;
    matrix *m5 = NULL;
    matrix *m6 = NULL;
    matrix *m7 = NULL;


    for (int i = 0; i < newSize; ++i) {
        for (int j = 0; j < newSize; ++j) {
            a11[i * newSize + j] = data1[i * newSize + j];
            a12[i * newSize + j] = data1[i * newSize + j + newSize];
            a21[i * newSize + j] = data1[(i + newSize) * newSize + j];
            a22[i * newSize + j] = data1[(i + newSize) * newSize + j + newSize];
            b11[i * newSize + j] = data2[i * newSize + j];
            b12[i * newSize + j] = data2[i * newSize + j + newSize];
            b21[i * newSize + j] = data2[(i + newSize) * newSize + j];
            b22[i * newSize + j] = data2[(i + newSize) * newSize + j + newSize];
        }
    }
    matrix *A11 = createMatrix(newSize, newSize, a11);
    free(a11);
    matrix *A12 = createMatrix(newSize, newSize, a12);
    free(a12);
    matrix *A21 = createMatrix(newSize, newSize, a21);
    free(a21);
    matrix *A22 = createMatrix(newSize, newSize, a22);
    free(a22);
    matrix *B11 = createMatrix(newSize, newSize, b11);
    free(b11);
    matrix *B12 = createMatrix(newSize, newSize, b12);
    free(b12);
    matrix *B21 = createMatrix(newSize, newSize, b21);
    free(b21);
    matrix *B22 = createMatrix(newSize, newSize, b22);
    free(b22);

    matrix *tem1 = NULL, *tem2 = NULL;

    tem1 = addMatrix(A11, A22);
    tem2 = addMatrix(B11, B22);
    m1 = strassen(tem1, tem2);
    deleteMatrix(tem1);
```

```
70        deleteMatrix(tem2);
71
72        tem1 = addMatrix(A21, A22);
73        m2 = strassen(tem1, B11);
74        deleteMatrix(tem1);
75
76        tem1 = subMatrix(B12, B22);
77        m3 = strassen(tem1, A11);
78        deleteMatrix(tem1);
79
80        tem1 = subMatrix(B21, B11);
81        m4 = strassen(tem1, A22);
82        deleteMatrix(tem1);
83
84        tem1 = addMatrix(A11, A12);
85        m5 = strassen(tem1, B22);
86        deleteMatrix(tem1);
87
88        tem1 = subMatrix(A21, A11);
89        tem2 = subMatrix(B11, B12);
90        m6 = strassen(tem1, tem2);
91        deleteMatrix(tem1);
92        deleteMatrix(tem2);
93
94        tem1 = subMatrix(A12, A22);
95        tem2 = addMatrix(B21, B22);
96        m7 = strassen(tem1, tem2);
97        deleteMatrix(tem1);
98        deleteMatrix(tem2);
99
100       deleteMatrix(A11);
101       deleteMatrix(A12);
102       deleteMatrix(A21);
103       deleteMatrix(A22);
104       deleteMatrix(B11);
105       deleteMatrix(B12);
106       deleteMatrix(B21);
107       deleteMatrix(B22);
108       c11 = addMatrix(addMatrix(m1, m4), subMatrix(m7, m5));
109       c12 = addMatrix(m3, m5);
110       c21 = addMatrix(m2, m4);
111       c22 = addMatrix(addMatrix(m1, m3), subMatrix(m6, m2));
112       deleteMatrix(m1);
113       deleteMatrix(m2);
114       deleteMatrix(m3);
115       deleteMatrix(m4);
116       deleteMatrix(m5);
117       deleteMatrix(m6);
118       deleteMatrix(m7);
119       float *resultData = malloc(rXc * sizeof(float));
120       for (int i = 0; i < newSize; ++i) {
121           for (int j = 0; j < newSize; ++j) {
122               resultData[i * newSize + j] = c11->data[i * newSize + j];
123               resultData[i * newSize + j + newSize] = c12->data[i * newSize + j];
124               resultData[(i + newSize) * newSize + j] = c21->data[i * newSize +
    j];
```

```
            resultData[(i + newSize) * newSize + j + newSize] = c22->data[i * newSize + j];
        }
    }
    matrix *result = createMatrix(mat1->row, mat2->column, resultData);
    deleteMatrix(c11);
    deleteMatrix(c12);
    deleteMatrix(c21);
    deleteMatrix(c22);
    free(resultData);
    return result;
}
```

数据测试

※其中小于3的矩阵乘法直接使用硬件优化+O3+OpenMP后的乘法

斯特拉森算法（32）：

	1	2	3	平均
128×128	396	424	394	404.666667
1k×1k	67218	68163	68772	68051

但是该算法存在精度丢失过大的问题，且在矩阵没有非常大时优化的时间并没有前面的几个方法来得显著，而矩阵非常大的时候又容易爆内存，得不偿失，因此最终没有把该算法加入综合优化中。

`matmul_improved`综合优化

代码

综合使用了**硬件优化**、**O3优化**以及OpenMP优化：

```
matrix *matmul_improved(const matrix *mat1, const matrix *mat2) {
    if (mat1 == NULL || mat1->column <= 0 || mat1->row <= 0 || mat1->data == NULL) {
        fprintf(stderr, "Error in \"mulMatrix\": Input a invalid matrix(left)");
        printFalse(mat1);
        return NULL;
    } else if (mat2 == NULL || mat2->column <= 0 || mat2->row <= 0 || mat2->data == NULL) {
        fprintf(stderr, "Error in \"mulMatrix\": Input a invalid matrix(right)");
        printFalse(mat2);
        return NULL;
    } else if (mat1->column != mat2->row) {
        fprintf(stderr, "Error in \"mulMatrix\": mat1's column(%d) not equal mat2's row(%d).\n", mat1->column,
                mat2->row);
        return NULL;
    }
    size_t r = mat1->row;
    size_t c = mat2->column;
```

```
17      float *array = NULL;
18      array = malloc(r * c * sizeof(float));
19      if (array == NULL) {
20          fprintf(stderr, "Error in \"mulMatrix\": Fail to malloc space for
    data.\n", mat1->column, mat2->row);
21          return NULL;
22      }
23      float tem = 0;
24
25  #pragma omp parallel
26      {
27  #pragma omp for
28          for (int i = 0; i < r; ++i) {
29              for (int k = 0; k < r; ++k) {
30                  tem = mat1->data[i * r + k];
31                  for (int j = 0; j < r; ++j) {
32                      array[i * r + j] += tem * mat2->data[k * r + j];
33                  }
34              }
35          }
36      }
37
38      matrix *newMatrix = createMatrix(r, c, array);
39      return newMatrix;
40  }
```

数据测试

	1	2	3	平均
16×16				0
128×128				0.01
1k×1k	33	34	32	33
8k×8k	52260	55465	53345	53690

数据生成与测试

数据生成

为方便测试，我将生成的例子直接存在`data.c`文件中，只要调用其中函数就能得到对应的一维数组。

```java
public class float随机生成 {
    public static void main(String[] args) throws IOException {
        int size = (int) Math.pow(8000,2);
        Random rdm = new Random();
        File matrix1 = new
File("C:\\Users\\27449\\IdeaProjects\\lab8e_A\\src\\8k");
        FileWriter fl = new FileWriter(matrix1);
        for (int i = 0;i<size;i++){
            fl.write(rdm.nextFloat(100000)+"f,");
        }
        fl.close();
    }
}
```

但是64kX64k实在太大了，我的cpu好像不支持创建这么大的数组。因此暂时先生成了以下几个例子的函数：

```
float * data16X16();
float * data128X128();
float * data1kX1k();
float * data8kX8k();
```

最后发现貌似只需要最大的8kX8k就不需要前面的了，只需要改变size就行。

```
int main() {
    size_t size = 8000;
    matrix * a = NULL;
    float * data = data8kX8k();
    a = createMatrix(size,size,data);
    struct matrix * m1 = NULL;
    //----要测试的函数----//(预编译)

    //------------------//
    deleteMatrix(m1);
    struct timeb tb;
    ftime(&tb);
    long start_time= tb.time*1000+tb.millitm;
    //----要测试的函数----//

    //------------------//
    ftime(&tb);
    long end_time= tb.time*1000+tb.millitm;
    double rs=end_time-start_time;
    printf("rs=%f\n",rs);
    return 0;
}
```

数据测试

因为会用到openMP，所以clock()并不能适用于所有情况，因此统一使用ftime()来测函数的运行时间。

为了避免编译器“**太聪明**”自动优化了函数，所以每次测试都先跑一遍函数，然后计算接下来三次跑函数的时间取平均值作为该函数的使用时长。

```
    m1 = matmul_plain(a,a);
    deleteMatrix(m1);
    struct timeb tb;
    ftime(&tb);
    long start_time= tb.time*1000+tb.millitm;
    m1 = matmul_plain(a,a);
    ftime(&tb);
    long end_time= tb.time*1000+tb.millitm;
    double rs=end_time-start_time;
    printf("rs=%f\n",rs);
```

OpenBlas数据比较

单位：ms	16×16	128×128	1k×1k	8k×8k	10k×10k
matmul_improved	0	1	32	53690	111500
OpenBlas	10	13	33	9095	16545

可以看到在矩阵较小时openBlas会稍慢些，但矩阵超过1k阶之后openBlas慢慢显现出优势，到10k阶的时候已经差了近10倍。

(附)对pro3中的函数进行的小优化

结构体

`long row` -----> `size_t row`

`size_t` 有更好的兼容性，并且没有正负号，而row和column必须为非负数。

`struct matrix{};` -----> `typedef struct matrix{}matrix;`

更方便，不冗杂。

打印错误

`printf` -----> `fprint(stderr,"...")`

打印错误

createMatrix

更改后：

```
matrix *createMatrix(const size_t r, const size_t c, float * data) {
    if (data == NULL) {
        fprintf(stderr,"Error in \"createMatrix\": Pointer is NULL.\n");
        return NULL;
    } else if (r * c == 0 || data == NULL) {
        fprintf(stderr,"Error in \"createMatrix\": Matrix is empty.\n");
```

```
7           return NULL;
8       }
9       matrix *newMat = NULL;
10      newMat = (matrix *) malloc(sizeof (matrix));
11      if(newMat == NULL){
12          fprintf(stderr,"Error in \"createMatrix\": Fail to malloc space for mat
    pointer.\n");
13          return NULL;
14      }
15      newMat->row = r;
16      newMat->column = c;
17      float *saveData = NULL;
18      saveData = (float *) malloc(r * c * sizeof(float));
19      if(saveData == NULL){
20          fprintf(stderr,"Error in \"createMatrix\": Fail to malloc space for data
    pointer.\n");
21          free(newMat);
22          return NULL;
23      }
24      for (size_t i = 0; i < r * c; ++i)saveData[i] = data[i];
25      newMat->data = saveData;
26      refreshType(newMat);
27      return newMat;
28  }
```

增加了对两次内存申请失败的判断，且第二次申请失败后，释放之前申请成功的内存。如此构成一个更加**“稳健”**的函数

内存申请

许多函数存在类似`float array[r*c]`的申请，这样内存管理会出现纰漏，正确做法应该是：

`float * array = NULL;`

新建指针时需要指向NULL，方便后续判断错误来源。

`array = malloc(r*c*(typeof(float)));`

分配动态内存

`if (!array){...}`

判断是否为空指针，若空了就是申请失败了

后面如果还有其他判断，离开函数之前要把已申请的内存释放