CS205 C/C++ Programming -- Project 2 Simple Matrix Multiplication

Name: 陈长信(Chen Changxin)

SID: 12210731

Part1--问题分析

本项目旨在实现通过C以及Java语言实现矩阵乘法，并分析他们各自在不同矩阵`size`情况下计算速度等方面的表现。

关于矩阵乘法，本项目主要通过计算式`result[i][j] += matrix1[i][k] * matrix2[k][j]`进行计算，即通过$O(n^3)$的复杂度计算矩阵乘法，然后Java中通过`nanoTime()`计算时间，C中通过`clock()`计算运行时间，最后通过文档输出比较时间。

由于本项目侧重探究矩阵乘法的速度的各种影响因素以及探究可能存在的优化方案，故淡化了对其余细节的处理，比如手动输入矩阵以及对矩阵是否能进行乘法等。

探究部分：

探究点一：遍历顺序对实现矩阵乘法速度的影响，通过对内存地址访问先后顺序的更换来分析其对矩阵乘法速度的影响。

探究点二：探究O1,O2,O3优化对矩阵计算速度的提升情况，想通过分析优化的具体机制理解对运算速度的影响。

探究点三：探究`random()`随机函数取值范围以及结合O3优化的机制对矩阵计算速度的影响。

探究点四：通过修改部分Java代码以及C语言对于循环展开以及单双重指针的应用来探究对运算速度的影响，同时结合O3的优化结果进一步进行内部优化机制的探索。

Part2--基础实验

Java部分设计思路

1. 利用矩阵乘法$O(n^3)$复杂度完成函数，利用`result[i][j] += matrix1[i][k] * matrix2[k][j]`关键式

```
public static void multiplyMatrices(float[][] matrix1, float[][] matrix2,
float[][] result, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

2. 利用随机数生成的方式，在0-upperbound之间生成随机数

```
public static void enterMatrixElements(float[][] matrix, float upperbound) {
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[0].length; j++) {
            Random random = new Random();
            float randomNumber = random.nextFloat();
            randomNumber = randomNumber * upperbound;
            matrix[i][j] = randomNumber;
         }
     }
}
```

3. 利用系统的`nanoTime`方法可以得到纳秒级别的时间，更为精确

```
long starttime = System.nanoTime();
multiplyMatrices(matrix1, matrix2, result, testcases[i]);
long endtime = System.nanoTime();

long elapsedTime = endtime - starttime;
double elapsedTimeInSeconds = (double) elapsedTime / 1_000_000_000.0;
```

4. 利用`append = true`的连续输入，对`testcases = {10, 100, 200, 500, 1000, 2000, 3000, 5000, 10000}`中所有`testcase`矩阵的大小都作乘法计算时间的测量

```
FileWriter writer = new FileWriter("matrix_multiplication_time.txt", true);
writer.write("Time taken for matrix multiplication: " + timeInSeconds + "
seconds.(sizeof " + n + ")\n");
writer.close();
System.out.println("Matrix multiplication time written to
matrix_multiplication_time.txt successfully.");
```

C部分设计思路

1. 与Java相似，利用矩阵乘法$O(n^3)$复杂度完成函数。（这里想法是想要与Java尽量公平的比较时间，故使用双指针，单指针的实现比较在后续进行实现）

```
void multiplyMatrices(float **matrix1, float **matrix2, float **result, int
n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            for (int k = 0; k < n; k++)
            {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

2. 同样的，我们使用生成随机数的方式对矩阵进行赋值，这里思路是用`rand`生成随机整数除以`rand`的最大值然后乘上`upperbound`，这里默认数据从下界为`0`

```
for (int i = 0; i < testcases[testcase]; i++)
{
     for (int j = 0; j < testcases[testcase]; j++)
     {
          float randnumber1 = 1.0 * rand() / RAND_MAX * upper;
          matrix1[i][j] = randnumber1;
     }
}
```

3. 利用系统内部的时间，计算矩阵乘法函数运行的时间，并且通过外部文件输出

```
clock_t start = clock();
multiplyMatrices(matrix1, matrix2, result, testcases[testcase]);
clock_t end = clock();
double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

FILE *fp;
fp = fopen("output.txt", "w");
if (fp == NULL)
{
     printf("无法打开");
     return 1;
}

fprintf(fp, "Sizeof %d ", testcases[testcase]);
fprintf(fp, "Time taken(seconds): ");
fprintf(fp, "%.10f\n", time_spent);
```

数据分析部分

Java部分时间结果

```
Time taken for matrix multiplication: 3.52E-5 seconds.(sizeof 10)
Time taken for matrix multiplication: 0.0051628 seconds.(sizeof 100)
Time taken for matrix multiplication: 0.0276693 seconds.(sizeof 200)
Time taken for matrix multiplication: 0.1401756 seconds.(sizeof 500)
Time taken for matrix multiplication: 1.3561175 seconds.(sizeof 1000)
Time taken for matrix multiplication: 39.5528459 seconds.(sizeof 2000)
Time taken for matrix multiplication: 149.3848645 seconds.(sizeof 3000)
Time taken for matrix multiplication: 872.0138086 seconds.(sizeof 5000)
```

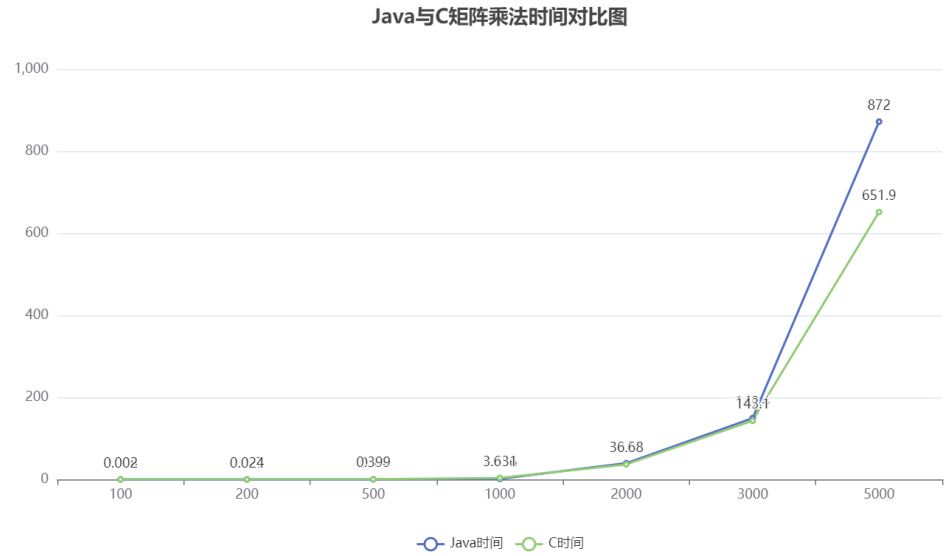
C部分时间结果

YU++ > Project2-Matrix > output.txt

```
1  Sizeof 10 Time taken(seconds): 0.0000060000
2  Sizeof 100 Time taken(seconds): 0.0028660000
3  Sizeof 200 Time taken(seconds): 0.0241240000
4  Sizeof 500 Time taken(seconds): 0.3999340000
5  Sizeof 1000 Time taken(seconds): 3.6312590000
6  Sizeof 2000 Time taken(seconds): 36.6829370000
7  Sizeof 3000 Time taken(seconds): 143.0909250000
8  Sizeof 5000 Time taken(seconds): 651.9533410000
```

通过数据对比，我们发现C语言的运行速度相较于Java略快一点，我认为主要有三点原因：首先是C语言开数组空间上更加节省，可以更加灵活地管理内存，可以使用手动内存分配和释放，避免了Java的垃圾回收机制带来的开销。在矩阵乘法等需要大量内存操作的场景下，这种差异可能会导致C语言的性能优势。其次是C语言对内存访问更加直接，更容易优化以利用CPU缓存。相比之下，Java程序的内存访问可

能更加抽象，可能会导致缓存未命中率更高，从而降低性能。最后就是C语言作为编译型语言是直接将其编译为本地机器码，而Java作为半解释半编译型的语言，需要通过JVM虚拟机进行解释，需要进行额外的解释和转换步骤。



通过上述折线图也可以体现出时间上的差距，基础部分由于设备算力的原因没有继续扩大矩阵范围，如果继续扩大矩阵范围将会看到更加明显的计算速度差距。

Part3--探究实验

1.遍历顺序的影响

当我们修改遍历顺序的时候(使用`ikj`的遍历顺序而非`ijk`的遍历顺序)，发现计算矩阵乘法的速度有显著提升

```
public static void multiplyMatrices(float[][] matrix1, float[][] matrix2,
float[][] result, int n) {
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < n; k++) {
            for (int j = 0; j < n; j++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

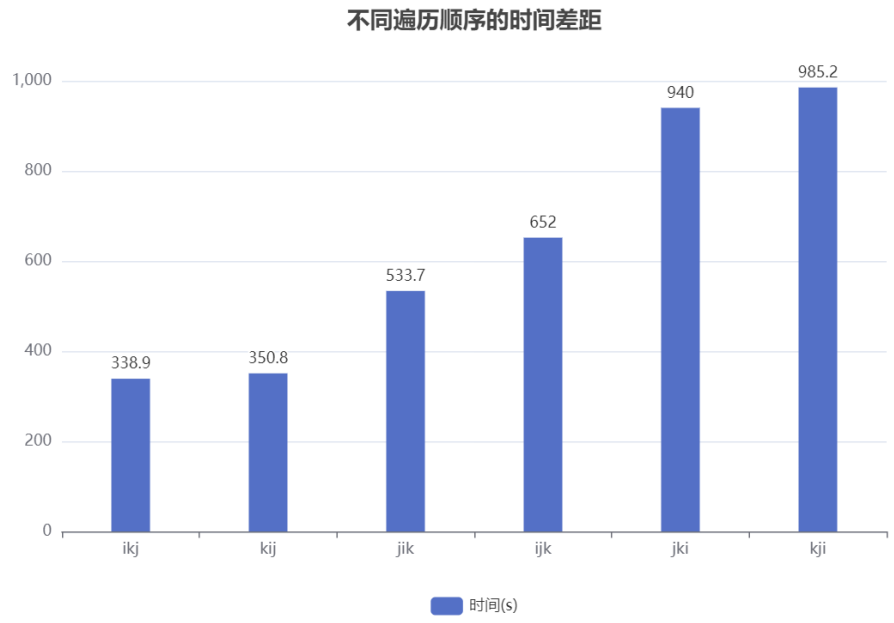
```
Time taken for matrix multiplication: 3.54E-5 seconds.(sizeof 10)
Time taken for matrix multiplication: 0.0051633 seconds.(sizeof 100)
Time taken for matrix multiplication: 0.007301 seconds.(sizeof 200)
Time taken for matrix multiplication: 0.0490394 seconds.(sizeof 500)
Time taken for matrix multiplication: 0.3822181 seconds.(sizeof 1000)
Time taken for matrix multiplication: 4.5878687 seconds.(sizeof 2000)
Time taken for matrix multiplication: 13.0141288 seconds.(sizeof 3000)
Time taken for matrix multiplication: 56.7019867 seconds.(sizeof 5000)
Time taken for matrix multiplication: 445.9132828 seconds.(sizeof 10000)
```

拿`size = 5000`的数据进行对比，在`ijk`的遍历中，时间达到了将近726s，但是在调整顺序之后达到了57s，时间效率提高了近12倍。这是由于例如`ijk`遍历的时候，如果计算式是`result[i][j] += matrix1[i][k] * matrix2[k][j];`不管是在Java语言或者在我用双重指针实现的C语言代码中，对于外层的指针存储的是每一个内层指针的地址值，所以每次最里面的内循环`k`在得到`matrix2[k][j]`时候都会找一个新的地址值，然后再找到这个地址中的某一个元素，所以当一行中的数据量比较大的时候，

每一次都可能会出现`cache miss`就是缓存无法命中，导致很大的局限性，拖慢了运行速度，但是如果是在最内层循环中只是每次在行内访问即在一个基地址值下面进行访问，比如在`ikj`中`j`只出现在`result[k][j]`以及`matrix2[k][j]`的列中，所以每次跳转都是行内跳转，速度就会快很多。

下面用`c`语言给各个遍历顺序进行测试(`size = 5000`)，并按照时间从高到低排序

```
//改变遍历顺序
Sizeof 5000 Time taken(seconds): 985.1759540000(kji)
Sizeof 5000 Time taken(seconds): 940.0466920000(jki)
Sizeof 5000 Time taken(seconds): 651.9533410000(ijk)
Sizeof 5000 Time taken(seconds): 533.6735080000(jik)
Sizeof 5000 Time taken(seconds): 350.8210050000(kij)
Sizeof 5000 Time taken(seconds): 338.8972960000(ikj)
```



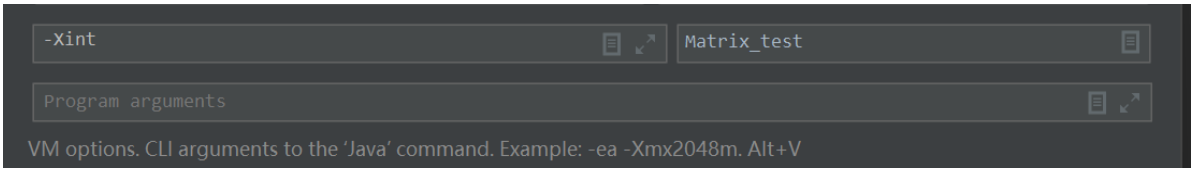
由此可见，在`j`为最内层的情况下时间明显较短，因为在行之间跳转的时间复杂度在$O(n^2)$数量级，而以`k`最内层时候在`matrix2[k][j]`时候需要进行跳转，时间明显变长，在$O(n^3)$数量级，最后在`i`在最内层时候`result[k][j]`以及`matrix2[k][j]`均要进行`n^3`次数级别的跳转，故时间更长。可见以上数据与分析较为相符。

但是这里有一个问题，就是为什么反而在改变了遍历顺序之后Java的速度提高非常明显，而C提高较小，导致甚至同样在计算`size = 5000`的情况下，Java可以达到56s的时间，而C却需要消耗338s?

解答1：基于这个问题，通过查阅资料，我发现Java通过了`JIT`即时编译器进行了优化，即`Just In Time`，实际上`JIT`编译器自动会给代码编译带来一定的优化空间，比如相类型的语言结构只用编译一遍，因此这样的比较实际上不算公平，所以同时我对C语言也开了一定的优化，比较结果相对公平。所以下表按照`ikj`的遍历顺序再次进行测试，我们发现C在各个维度上速度明显快于Java，符合我们的预期。

```
Sizeof 2000 Time taken(seconds): 1.4034000000
Sizeof 3000 Time taken(seconds): 5.9273570000
Sizeof 5000 Time taken(seconds): 29.7301110000
Sizeof 10000 Time taken(seconds): 237.3885010000
```

延伸：但实际上如果只是给C语言开-O3或者-O2优化，C和Jave的优化力度不太一样，所以这里我尝试禁用JIT中的优化操作，这样的比较应该会相对更加客观



基于此再对Java的运行速度进行测试，发现时间运行非常慢。

```
Time taken for matrix multiplication: 2.87E-5 seconds.(sizeof 10)
Time taken for matrix multiplication: 0.0242797 seconds.(sizeof 100)
Time taken for matrix multiplication: 0.1677167 seconds.(sizeof 200)
Time taken for matrix multiplication: 2.5281189 seconds.(sizeof 500)
Time taken for matrix multiplication: 21.7589809 seconds.(sizeof 1000)
Time taken for matrix multiplication: 174.5794712 seconds.(sizeof 2000)
Time taken for matrix multiplication: 606.9435735 seconds.(sizeof 3000)
```

所以综上所述，在各个情况下Java的表现情况都相对比C要差，但是必须要承认在改变遍历顺序之后对程序的优化效果非常明显，本质就是把行之间的跳转次数量级从$O(n^3)$降为了$O(n^2)$。

2.gcc的o1,o2,o3对计算速度的影响

同样的，我们使用`ikj`的遍历顺序，使用`gcc -O3 Matrix.c`等命令测试运行速度

```
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ gcc -O3 Matrix.c
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ ./a.out
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ gcc -O3 Matrix.c
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ ./a.out
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ gcc -O2 Matrix.c
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ ./a.out
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ gcc -O1 Matrix.c
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ ./a.out
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ gcc Matrix.c
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ ./a.out
```

```
//开了O3之后的ikj
Sizeof 5000 Time taken(seconds): 30.2980170000
Sizeof 5000 Time taken(seconds): 31.7328350000
//开了O2之后的ikj
Sizeof 5000 Time taken(seconds): 52.8532930000
//开了O1之后的ikj
Sizeof 5000 Time taken(seconds): 86.0347600000
//无优化的ikj
Sizeof 5000 Time taken(seconds): 351.7132860000
```

发现如上结果，在开了优化之后运行速度得到了显著提高，O2,O3等优化策略实际上就是通过命令对代码进行比如循环中的重排，代码移动等，但是这样会导致编译时间较长，以获取更高更快的性能，最终代码运行效率较高。

为了证明这一点，我利用`-ftime-report`的代码编译时间也做了测试

这里均以`size = 5000`的`ikj`遍历顺序进行实验。

```
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ gcc -o Matrix Matrix.c -ftime-report

 Time variable                     usr           sys          wall           GGC
  phase setup                  :  0.00 (  0%)   0.00 (  0%)   0.00 (  0%)  1298k ( 45%)
  phase parsing                :  0.01 (100%)   0.00 (  0%)   0.01 ( 50%)  1146k ( 40%)
  phase finalize               :  0.00 (  0%)   0.00 (  0%)   0.01 ( 50%)     0 (  0%)
  lexical analysis             :  0.01 (100%)   0.00 (  0%)   0.00 (  0%)     0 (  0%)
  parser (global)              :  0.00 (  0%)   0.00 (  0%)   0.01 ( 50%)   596k ( 21%)
  TOTAL                        :  0.01          0.00          0.02          2855k
```

```
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ gcc -o Matrix -O1 Matrix.c -ftime-report

 Time variable                                 usr           sys          wall           GGC
  phase setup                              :  0.00 (  0%)   0.00 (  0%)   0.00 (  0%)  1298k ( 37%)
  phase parsing                            :  0.01 ( 33%)   0.00 (  0%)   0.01 ( 25%)  1580k ( 45%)
  phase opt and generate                   :  0.02 ( 67%)   0.00 (  0%)   0.02 ( 50%)   659k ( 19%)
  callgraph functions expansion            :  0.01 ( 33%)   0.00 (  0%)   0.01 ( 25%)   461k ( 13%)
  callgraph ipa passes                     :  0.01 ( 33%)   0.00 (  0%)   0.01 ( 25%)   112k (  3%)
  df reg dead/unused notes                 :  0.00 (  0%)   0.00 (  0%)   0.01 ( 25%)  6624  (  0%)
  preprocessing                            :  0.01 ( 33%)   0.00 (  0%)   0.01 ( 25%)   516k ( 15%)
  tree PTA                                 :  0.00 (  0%)   0.00 (  0%)   0.01 ( 25%)  2632  (  0%)
  tree CCP                                 :  0.01 ( 33%)   0.00 (  0%)   0.00 (  0%)  3448  (  0%)
  CSE                                      :  0.01 ( 33%)   0.00 (  0%)   0.00 (  0%)    72  (  0%)
  TOTAL                                    :  0.03          0.00          0.04          3548k
```

```
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ gcc -o Matrix -O2 Matrix.c -ftime-report

Time variable                                   usr           sys          wall           GGC
 phase setup                    :   0.00 (  0%)   0.00 (  0%)   0.00 (  0%)  1298k ( 35%)
 phase parsing                  :   0.00 (  0%)   0.01 ( 50%)   0.01 ( 33%)  1580k ( 42%)
 phase opt and generate         :   0.01 (100%)   0.01 ( 50%)   0.02 ( 67%)   833k ( 22%)
 callgraph functions expansion  :   0.01 (100%)   0.00 (  0%)   0.01 ( 33%)   599k ( 16%)
 callgraph ipa passes           :   0.00 (  0%)   0.01 ( 50%)   0.01 ( 33%)   148k (  4%)
 preprocessing                  :   0.00 (  0%)   0.00 (  0%)   0.01 ( 33%)   516k ( 14%)
 parser (global)                :   0.00 (  0%)   0.01 ( 50%)   0.00 (  0%)   766k ( 21%)
 tree operand scan              :   0.00 (  0%)   0.01 ( 50%)   0.00 (  0%)    44k (  1%)
 tree CCP                       :   0.00 (  0%)   0.00 (  0%)   0.01 ( 33%)  3832  (  0%)
 tree PRE                       :   0.01 (100%)   0.00 (  0%)   0.00 (  0%)    15k (  0%)
 rest of compilation            :   0.00 (  0%)   0.00 (  0%)   0.01 ( 33%)  9464  (  0%)
 TOTAL                          :   0.01          0.02          0.03         3722k
```

```
daniel@Daniel-Chen:~/YU++/Project2-Matrix$ gcc -o Matrix -O3 Matrix.c -ftime-report

Time variable                                   usr           sys          wall           GGC
 phase setup                    :   0.00 (  0%)   0.00 (  0%)   0.00 (  0%)  1298k ( 32%)
 phase parsing                  :   0.01 ( 25%)   0.00 (  0%)   0.02 ( 40%)  1580k ( 39%)
 phase opt and generate         :   0.03 ( 75%)   0.00 (  0%)   0.03 ( 60%)  1118k ( 28%)
 callgraph functions expansion  :   0.02 ( 50%)   0.00 (  0%)   0.03 ( 60%)   884k ( 22%)
 callgraph ipa passes           :   0.01 ( 25%)   0.00 (  0%)   0.00 (  0%)   148k (  4%)
 df reaching defs               :   0.01 ( 25%)   0.00 (  0%)   0.00 (  0%)     0  (  0%)
 preprocessing                  :   0.00 (  0%)   0.00 (  0%)   0.01 ( 20%)   516k ( 13%)
 parser (global)                :   0.01 ( 25%)   0.00 (  0%)   0.01 ( 20%)   766k ( 19%)
 tree CCP                       :   0.01 ( 25%)   0.00 (  0%)   0.00 (  0%)  3832  (  0%)
 tree FRE                       :   0.00 (  0%)   0.00 (  0%)   0.01 ( 20%)    13k (  0%)
 forward prop                   :   0.01 ( 25%)   0.00 (  0%)   0.00 (  0%)   480  (  0%)
 scheduling 2                   :   0.00 (  0%)   0.00 (  0%)   0.01 ( 20%)    11k (  0%)
 rest of compilation            :   0.00 (  0%)   0.00 (  0%)   0.01 ( 20%) 10184  (  0%)
 TOTAL                          :   0.04          0.00          0.05         4007k
```

从上面三图中可以明显看出从无优化到-O3优化，总编译的Total时间从2855k逐步提高到了4007k，注意这里的k指的是kilo-milliseconds，即千分之一秒。结合所有情况的运行表现情况，可以得出在开优化之后编译时间变长，但是程序运行速度以及性能明显提高。

3.数据范围对计算速度的影响

Java测试

将数据随机范围扩大到`200`进行计算（为节省时间，这部分均是通过`ikj`的遍历顺序）

```
Time taken for matrix multiplication: 3.51E-5 seconds.(sizeof 10)
Time taken for matrix multiplication: 0.0046064 seconds.(sizeof 100)
Time taken for matrix multiplication: 0.0038508 seconds.(sizeof 200)
Time taken for matrix multiplication: 0.0549258 seconds.(sizeof 500)
Time taken for matrix multiplication: 0.414679 seconds.(sizeof 1000)
Time taken for matrix multiplication: 4.0505597 seconds.(sizeof 2000)
Time taken for matrix multiplication: 12.0873294 seconds.(sizeof 3000)
Time taken for matrix multiplication: 58.53855 seconds.(sizeof 5000)
Time taken for matrix multiplication: 451.987009 seconds.(sizeof 10000)
```

接着将随机范围扩大到`20000`进行计算

```
Time taken for matrix multiplication: 3.74E-5 seconds.(sizeof 10)
Time taken for matrix multiplication: 0.0047894 seconds.(sizeof 100)
Time taken for matrix multiplication: 0.0046591 seconds.(sizeof 200)
Time taken for matrix multiplication: 0.0497853 seconds.(sizeof 500)
Time taken for matrix multiplication: 0.4176594 seconds.(sizeof 1000)
Time taken for matrix multiplication: 4.1657338 seconds.(sizeof 2000)
Time taken for matrix multiplication: 12.761667 seconds.(sizeof 3000)
Time taken for matrix multiplication: 58.0100977 seconds.(sizeof 5000)
Time taken for matrix multiplication: 458.4890264 seconds.(sizeof 10000)
```

在`java`的测试下，分别对比在`0-10`,`0-200`,`0-20000`随机数计算情况下时间的差别，发现时间变化并不大。

C测试

```
//开优化之后的range = 10
Sizeof 10000 Time taken(seconds): 42.7309020000
//开优化之后的range = 200
Sizeof 10000 Time taken(seconds): 39.6417390000
//开优化之后的range = 20000
Sizeof 10000 Time taken(seconds): 37.7225970000
//开优化之后的range = 2000000
Sizeof 10000 Time taken(seconds): 38.9802560000
```

同样的，在开了O3优化之后计算速度与数据的范围仍然无较大影响。

但是由于前期的测试想到O3优化可能会对部分计算策略也进行优化，于是通过小矩阵对其进行了再次测试

```
//range = 10
Sizeof 2000 Time taken(seconds): 21.0346610000
//range = 200
Sizeof 2000 Time taken(seconds): 20.9226330000
//range = 20000
Sizeof 2000 Time taken(seconds): 20.8851450000
//range = 2000000
Sizeof 2000 Time taken(seconds): 20.8568830000
```

发现结果一如既往，在即使矩阵数字`range`发生大变化时候也对计算速度无明显影响。

我认为主要原因可能是在矩阵乘法主要消耗的时间在于行列直接的跳转与遍历，而非这些简单的相乘以及加法运算，所以如果在数据范围上进行改变并不会对运算速度有明显的影响。

4.通过对源代码的部分修改影响运行速度

Java部分

```java
public static float[][] multiplyMatrices(float[][] matrix1, float[][] matrix2) {
    int m1Rows = matrix1.length;
    int m1Cols = matrix1[0].length;
    int m2Cols = matrix2[0].length;

    float[][] result = new float[m1Rows][m2Cols];

    for (int i = 0; i < m1Rows; i++) {
        for (int j = 0; j < m2Cols; j++) {
            for (int k = 0; k < m1Cols; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
             }
        }
    }
    return result;
}
```

我通过减少传入的参数数据量，减少一个传入的数组，在运行时候速度得到了一定的提高

```
Time taken for matrix multiplication: 3.23E-5 seconds.(sizeof 10)
Time taken for matrix multiplication: 0.0048354 seconds.(sizeof 100)
Time taken for matrix multiplication: 0.022138 seconds.(sizeof 200)
Time taken for matrix multiplication: 0.1402743 seconds.(sizeof 500)
Time taken for matrix multiplication: 1.1402487 seconds.(sizeof 1000)
Time taken for matrix multiplication: 8.2719492 seconds.(sizeof 1500)
Time taken for matrix multiplication: 31.7674427 seconds.(sizeof 2000)
Time taken for matrix multiplication: 120.5784127 seconds.(sizeof 3000)
Time taken for matrix multiplication: 725.651203 seconds.(sizeof 5000)
```

C部分

探究点一

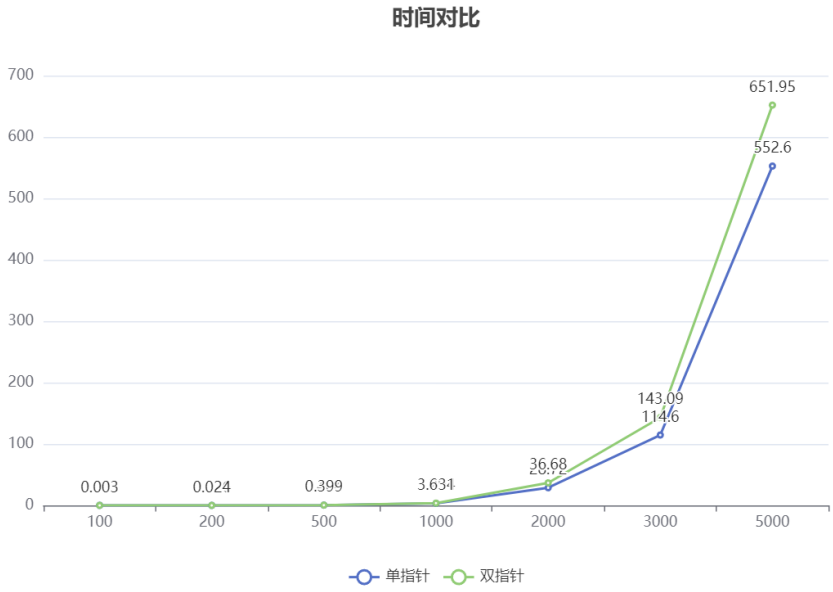
首先我对二层指针做了修改，将数据在内存中的存储方式进行了修改，均用单指针进行了数据存储处理。

```
int *matrix1 = (int *)malloc(testcases[testcase] * testcases[testcase] *
sizeof(int));
int *matrix2 = (int *)malloc(testcases[testcase] * testcases[testcase] *
sizeof(int));
int *result = (int *)malloc(testcases[testcase] * testcases[testcase] *
sizeof(int));
for (int i = 0; i < testcases[testcase]; i++)
{
    for (int j = 0; j < testcases[testcase]; j++)
    {
         float num = (float)1.0 * rand() / RAND_MAX * upper;
         *(matrix1 + i * testcases[testcase] + j) = num;
    }
}
```

然后同时修改了矩阵乘法的算法

```
void matrixMultiplication(int *mat1, int *mat2, int *result, int n)
{
    int i, j, k;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            *(result + i * n + j) = 0;
            for (k = 0; k < n; k++)
            {
                *(result + i * n + j) += *(mat1 + i * n + k) * *(mat2 + k * n +
j);
            }
        }
    }
}
```

```
Sizeof 100 Time taken(seconds): 0.0031120000
Sizeof 200 Time taken(seconds): 0.0260610000
Sizeof 500 Time taken(seconds): 0.4037980000
Sizeof 1000 Time taken(seconds): 3.2935240000
Sizeof 2000 Time taken(seconds): 28.7160060000
Sizeof 3000 Time taken(seconds): 114.5535940000
Sizeof 5000 Time taken(seconds): 552.5899540000
```



以同样的方式进行测试并进行对比，测试的结果如上图所示，对比原来的双重指针的算法，我们发现运算速度有了一定程度的提高，为什么使用单指针存储二维数组的运算会更缓慢呢?

借鉴前面研究关于存储机制以及缓存命中的思想，我认为如果使用单指针存储的二维数组在内存上连续的，这有利于 CPU 缓存的预取和缓存行的加载；而二维数组使用二维指针访问时，可能会导致更多的缓存未命中，因为二维数组的不同行可能在内存中不是连续存储的，需要不断访问新的地址值，这会增加内存访问的开销。另外，使用一维数组通常会比使用二维指针更简洁，减少了对指针的间接引用，也减少了额外的指针算术运算。

同样的，根据前面有关于循环顺序的启示，在交换了循环的顺序之后，我也进行了测试，结果如下

```
Sizeof 100 Time taken(seconds): 0.0031390000
Sizeof 200 Time taken(seconds): 0.0270330000
Sizeof 500 Time taken(seconds): 0.4189690000
Sizeof 1000 Time taken(seconds): 3.4291170000
Sizeof 2000 Time taken(seconds): 26.7710660000
Sizeof 3000 Time taken(seconds): 88.3367040000
Sizeof 5000 Time taken(seconds): 401.6571260000
```

同样的，运行时间也明显缩短，这也再次验证了我们之前得到的结论，指针在访问内存时候跳转的次数以及缓存命中的成功率会影响到运行的速度。通过代码来看的话

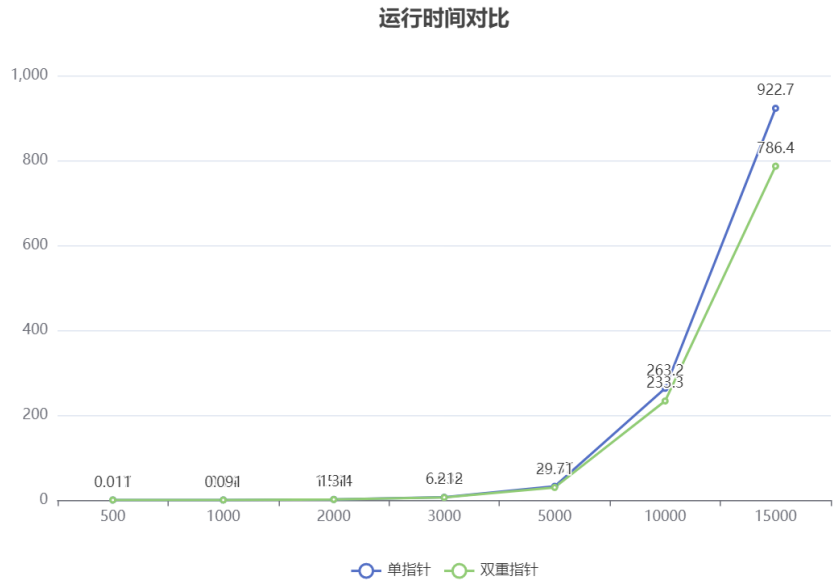
```
*(result + i * n + k) += *(mat1 + i * n + j) * *(mat2 + j * n + k);
```

这样的计算式内层循环变量k每次只用+1，所以只会近邻元素的访问，从而避免了`*(result + i * n + j) += *(mat1 + i * n + k) * *(mat2 + k * n + j);`原来代码中每次最内层循环k都需要访问查询内存中新的基地址下的值，访问次数达到了n^3级别，而修改之后只有n^2级别，这样的话大大降低了内存访问跳转的时间，从而提高了运行速度。

在知道了双重指针与单指针的存储方式可能带来差别，结合之前对O3优化制度的探究，我想知道O3的优化是否也同样进行了内存管理上的优化，于是进了下进一步的测试。

```
//开了O3之后的ikj(单指针)
Sizeof 10 Time taken(seconds): 0.0000010000
Sizeof 100 Time taken(seconds): 0.0001380000
Sizeof 200 Time taken(seconds): 0.0011670000
Sizeof 500 Time taken(seconds): 0.0173600000
Sizeof 1000 Time taken(seconds): 0.1403870000
Sizeof 2000 Time taken(seconds): 1.5443930000
Sizeof 3000 Time taken(seconds): 6.6346220000
Sizeof 5000 Time taken(seconds): 32.3704510000
Sizeof 10000 Time taken(seconds): 263.1916640000
Sizeof 15000 Time taken(seconds): 922.7067830000
```

```
//开了O3之后的ikj(双指针)
Sizeof 10 Time taken(seconds): 0.0000000000
Sizeof 100 Time taken(seconds): 0.0000740000
Sizeof 200 Time taken(seconds): 0.0006000000
Sizeof 500 Time taken(seconds): 0.0108140000
Sizeof 1000 Time taken(seconds): 0.0906230000
Sizeof 2000 Time taken(seconds): 1.3094650000
Sizeof 3000 Time taken(seconds): 6.2120660000
Sizeof 5000 Time taken(seconds): 29.7053250000
Sizeof 10000 Time taken(seconds): 233.3061320000
Sizeof 15000 Time taken(seconds): 786.3636550000
```



我们发现在进行优化的时候实际上双重指针的写法表现情况更好，也就是O3对于双重指针写法的优化效率更高，这是为什么?

由于两个函数代码写法上差异较小，所以我们暂且忽略代码细节上的差别，我认为原因可能是双重指针的存储方式可能使得优化后的编译器更容易进行循环展开或者对内循环进行优化，以利用现代处理器的流水线并行性。然而按照单指针的存储方式在一个同样的栈中进行跳转反而变得不好进行优化，优化策略体现不明显。

探究点二

了解到，进行循环的展开可能会影响运行的速度，于是我进行了一部分测试，即将源代码函数进行了一部分更改。

```
void multiplyMatrices(float **matrix1, float **matrix2, float **result, int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int k = 0; k < n; k++)
        {
            for (int j = 0; j < n; j += 2)
            {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
                result[i][j + 1] += matrix1[i][k] * matrix2[k][j + 1];
            }
        }
    }
}
```

在进行了部分展开之后，我发现运行速度确实有了一定程度上的提高。

```
//没有进行循环展开
Sizeof 3000 Time taken(seconds): 73.0735940000
Sizeof 5000 Time taken(seconds): 338.8972960000
//进行了循环展开
Sizeof 3000 Time taken(seconds): 69.8615220000
Sizeof 5000 Time taken(seconds): 331.8737160000
```

其实我认为对于循环展开，它将循环体内的代码复制多次，以减少循环控制开销和分支预测开销。通过展开循环，可以在每次迭代中执行更多的指令，这样就从而提高指令级并行性和减少循环的迭代次数，进而提高程序的运行速度。

基于此以及前面关于O3优化的研究，通过查阅资料了解到O3的一部分优化功能实际上就是对循环进行展开，为了验证这一点，我给是否进行循环展开进行了测试。

```
//进行循环展开，开O3
Sizeof 5000 Time taken(seconds): 32.0582090000
Sizeof 10000 Time taken(seconds): 257.5698990000
Sizeof 15000 Time taken(seconds): 894.0243390000
//没有进行循环展开，开O3
Sizeof 5000 Time taken(seconds): 29.1989050000
Sizeof 10000 Time taken(seconds): 234.7919020000
Sizeof 15000 Time taken(seconds): 809.7075360000
```

结果意外的是，我进行了手动部分循环展开，但是在打开优化之后反而运行速度下降，这是为什么？

我认为可能有两点原因：一是因为展开循环可能会增加代码的大小，导致指令缓存未命中和数据缓存未命中的次数增加，从而影响性能。二是手动展开循环可能会与编译器的自动优化相冲突，编译器可能无法正确地理解代码，没有进行普通循环中更加高效的循环展开，导致性能反而降低。

Part4--项目总结以及主要困难解决方案

1. 本项目代码部分不算困难，即实现普通矩阵乘法，难点在于探究不同因素对于矩阵乘法速度的影响，我探究的因素包括使用的语言，使用的嵌套格式顺序，计算数据范围以及使用的优化。
2. 本项目探究涉及O3优化等操作，根据ChatGPT指引以及查找各种资料进行探究使得目的性更强，比如可以更深刻理解优化的具体机制以及缓存命中等内部机制。
3. 在探究时间过程尤其是进行时间比较的时候，影响因素较多，所以我尽量让函数传入参数、循环的写法等代码细节维持平衡这样比较较为客观。

Part5--补充说明

1. 本项目源代码包含所有探究步骤的源代码，为保证正确运行，部分代码进行了注解，详细可见代码内的解释行。
2. 本项目部分有关时间计算的代码框架来源于ChatGPT。
3. 会同时附C语言以及Java代码。