中山大学计算机院本科生实验报告

（2024 学年秋季学期）

课程名称：高性能计算程序设计　　批改人：

实验	通用矩阵乘法	专业（方向）	信息与计算科学
学号	22336049	姓名	陈日康
Email	chenih5@mail2.sysu.edu.cn	完成日期	2024 年 9 月 24 日

1. 实验目的

随机生成 M*N 和 N*K 的两个矩阵 A, B，矩阵元素为单精度浮点数（float），对这两个矩阵做乘法得到矩阵 C。输出 A, B, C 三个矩阵及矩阵计算时间。采用多种语言（C, Python, java）实现乘法，通过编译器优化做对比。

2. 实验过程和核心代码

(1). Python

代码的核心思想是通过嵌套循环实现两个矩阵的乘法，并计算执行时间。先定义矩阵的维数 M=1200, N=1000, K=800，然后生成两个随机的矩阵 A（M*N）和 B（N*K），初始化一个 M*K 的结果矩阵 C。通过三个嵌套的 for 循环，遍历矩阵 A 和 B 的各个元素完成乘法运算。使用 time.time()来测量运算前后的时间差。输出为矩阵 A、B 和乘积矩阵 C 的部分矩阵与执行时间。

```python
# 固定矩阵的维数
M = 1200
N = 1000
K = 800

print("Calculating running time...")
A = np.random.rand(M, N).astype(np.float32)
B = np.random.rand(N, K).astype(np.float32)
C = np.zeros( shape: (M, K), dtype=np.float32)   # 初始化结果矩阵 C

# 执行矩阵乘法并测量时间
start_time = time.time()

# 矩阵乘法
for i in range(M):
    for j in range(K):
        for k in range(N):
            C[i, j] += A[i, k] * B[k, j]

end_time = time.time()
```

(2). Java

在 Matrix 类中，通过构造函数设定矩阵的行数和列数。generateRandomValues 方法利用 Random 类填充矩阵中的每个元素为随机浮点数。multiply 方法则实现了矩阵的乘法运算。在主类 lab0_1 中，定义了三个矩阵 A、B 和 C，初始化它们的大小，并生成随机值。通过测量矩阵乘法的运行时间，最后打印出矩阵 A、B 和乘积矩阵 C 的前两行两列元素以及运算所花费的时间。

```
for (int p = 0; p < C.cols; p++) {
    for (int l = 0; l < this.cols; l++) {
        for (int i = 0; i < C.rows; i++) {
            C.data[i][p] += this.data[i][l] * B.data[l][p];
        }
    }
}
```

(3). C

与 python 的代码大致相同，先定义矩阵的维数 M=1200, N=1000, K=800。通过 generate_random_matrix 函数，随机生成矩阵 A 和 B，通过 malloc 动态分配内存为矩阵 A、B 和结果矩阵 C。矩阵乘法通过 multiply_matrices 函数实现。其中三个嵌套的循环遍历矩阵的行列，依次进行元素乘积求和。使用 clock()函数记录执行时间。最后打印出矩阵 A、B 和乘积矩阵 C 的前两行两列元素以及运算所花费的时间。

```
#define M 1200
#define N 1000
#define K 800

void generate_random_matrix(float** matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = (float)rand() / RAND_MAX;
        }
    }
}

void multiply_matrices(float** A, float** B, float** C, int m, int n, int k) {
    for (int j = 0; j < k; j++) {
        for (int l = 0; l < n; l++) {
            for (int i = 0; i < m; i++) {
                C[i][j] += A[i][l] * B[l][j];
            }
        }
    }
}
```

(4). C (sorted)

这段代码与前一段代码的主要区别在于矩阵乘法的计算顺序不同，两段代码实现的基本功能和结构是相同的，都是进行矩阵乘法并测量执行时间。调整后循环顺序是 for (int i =

0; i < m; i++) → for (int l = 0; l < n; l++) → for (int j = 0; j < k; j++)，即先遍历结果矩阵的行，再遍历矩阵 A 的列和 B 的行，最后遍历结果矩阵的列。

```c
void multiply_matrices(float** A, float** B, float** C, int m, int n, int k) {
    for (int i = 0; i < m; i++) {
        for (int l = 0; l < n; l++) {
            for (int j = 0; j < k; j++) {
                C[i][j] += A[i][l] * B[l][j];
            }
        }
    }
}
```

(5). C (optimized)

输入：g++ -O2 -o lab0_2 lab0_2.c 与 ./lab0_2.exe。

3. 实验结果

(1). 使用 Python 的运行结果如下：

```
D:\Anaconda3\python.exe "D:\Python\python project\HPC\lab0.py"
Calculating running time...
Matrix A:
0.6338 0.8345 ...
0.5364 0.2640 ...
...

Matrix B:
0.0179 0.1179 ...
0.0792 0.0693 ...
...

Matrix C (Result of A * B):
252.5106 258.5497 ...
244.3870 239.3051 ...
...

Run time: 574.1538553237915 seconds
```

(2). 使用 Java 的运行结果如下：

```
PS D:\Code\Java>  d:; cd 'd:\Code\Java'; & 'C:\Program Files\Java\jre1.8.0_421\bin\java.exe' '-cp' 'C:\Users\frank\AppData\Roaming\Code\User\workspaceStorage\8b91137dd8ff5bba9edd5050aa7060a7\redhat.java\jdt_ws\Java_d209beab\bin' 'lab0_1'
Calculating running time...

Matrix A (first 2*2 elements):
0.889522 0.092641 ...
0.671915 0.225995 ...
...

Matrix B (first 2*2 elements):
0.583584 0.448036 ...
0.820560 0.322775 ...
...

Matrix C (Result of A*B, first 2*2 elements):
245.078140 255.716507 ...
255.009735 260.593445 ...
...

Run time: 11.464 seconds.
```

(3). 使用 C 语言的运行结果如下（未优化）：

```
PS D:\Code\C>  & 'c:\Users\frank\.vscode\extensions\ms-vscode.cpptools-1.22.5-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-bvvg3cz4.x3f' '--stdout=Microsoft-MIEngine-Out-s2khjwma.3s3' '--stderr=Microsoft-MIEngine-Error-lkpch0y0.wpn' '--pid=Microsoft-MIEngine-Pid-1wc5x2qv.nm5' '--dbgExe=C:\Program Files\mingw64\bin\gdb.exe' '--interpreter=mi'
Calculating running time...

Matrix A (first 2*2 elements):
0.961394 0.829737 ...
0.446272 0.040345 ...
...

Matrix B (first 2*2 elements):
0.382458 0.414594 ...
0.754570 0.118381 ...
...

Matrix C (Result of A * B, first 2*2 elements):
255.359009 246.384018 ...
244.217545 238.391968 ...
...

Run time: 4.740000 seconds.
```

(4). 使用 C 语言的运行结果如下（调换顺序）：

```
PS D:\Code\C>  & 'c:\Users\frank\.vscode\extensions\ms-vscode.cpptools-1.22.5-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-x03x15b2.szz' '--stdout=Microsoft-MIEngine-Out-cfqx5xme.nuj' '--stderr=Microsoft-MIEngine-Error-odd5meul.zxm' '--pid=Microsoft-MIEngine-Pid-4inm2drr.ro2' '--dbgExe=C:\Program Files\mingw64\bin\gdb.exe' '--interpreter=mi'
Calculating running time...

Matrix A (first 2*2 elements):
0.021058 0.310923 ...
0.419813 0.499496 ...
...

Matrix B (first 2*2 elements):
0.478011 0.874386 ...
0.079257 0.197699 ...
...

Matrix C (Result of A*B, first 2*2 elements):
248.015945 259.541565 ...
259.285736 258.554230 ...
...

Run time: 3.065000 seconds.
```

(5). 使用 C 语言的运行结果如下（编译优化）：

```
PS D:\Code\C> cd test
PS D:\Code\C\test> g++ -O3 -o lab0_2 lab0_2.c
PS D:\Code\C\test> ./lab0_2.exe
Calculating running time...

Matrix A (first 2x2 elements):
0.631519 0.098972 ...
0.470565 0.771966 ...
...

Matrix B (first 2x2 elements):
0.437056 0.748711 ...
0.721427 0.785180 ...
...

Matrix C (Result of A * B, first 2x2 elements):
244.286545 243.860596 ...
251.032913 245.683838 ...
...

Run time: 0.264000 seconds.
```

计算浮点运算次数：外层三层嵌套循环的执行次数为 M*N*K，每次最内层循环有 2 次浮点运算。所以总浮点运算次数为：2*M*N*K，代入给定值：M=1200，N=1000，K=800，则总浮点运算次数为：$2 * 1200 * 1000 * 800 = 1.92 * 10^9$ 次浮点运算。

计算浮点性能（GFLOPS），可使用公式：$\text{GFLOPS} = \frac{\text{Total Floating Point Operations}}{\text{Execution Time(s)}} \times 10^{-9}$，因此各版本的浮点性能为：

Python：$\frac{1.92*10^9}{574.454} \times 10^{-9} = 0.1267$；

Java：$\frac{1.92*10^9}{5.144} \times 10^{-9} = 0.373$；

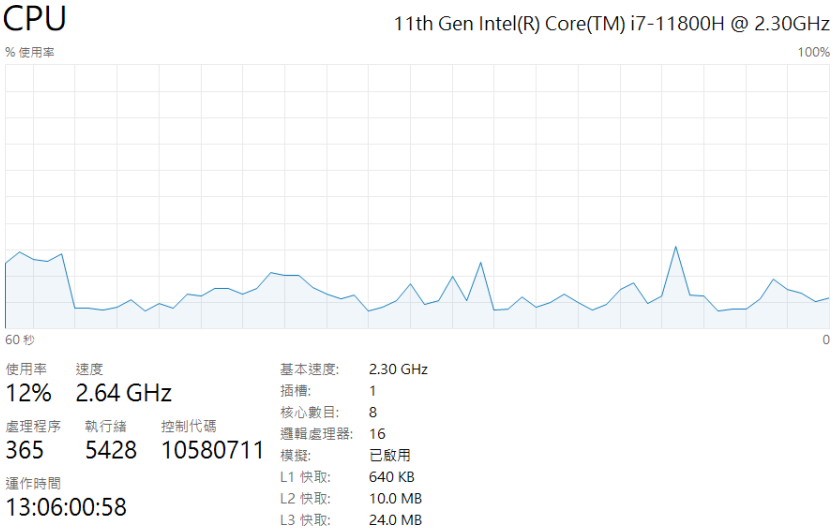
C（未优化）：$\frac{1.92*10^9}{4.74} \times 10^{-9} = 0.405$；

C（调整循环顺序）：$\frac{1.92*10^9}{3.065} \times 10^{-9} = 0.626$；

C（编译优化）：$\frac{1.92*10^9}{0.264} \times 10^{-9} = 7.272$

计算达到峰值性能的百分比：

计算相对峰值性能的百分比公式为：$\text{Performance Percentage} = \frac{\text{Actual GFLOPS}}{\text{Peak GFLOPS}} \times 100\%$，而计算理论峰值浮点性能（Peak GFLOPS）为：核心数量（Number of cores）* 时钟频率（Clock speed, in GHz）* 每个核心每个时钟周期的浮点运算次数（FLOPs per cycle per core）。



由上图可看出，此电脑的核心数量为 8，时钟频率为 2.3 GHz，而 AMD 系列的浮点计算单元通常都为 8，因此 $\text{Peak GFLOPS} = 8 * 2.3\text{GHz} * 8 = 147.2\ \text{GFLOPS}$。因此各版本达到峰值性能的百分比为：

Python：$\frac{0.003}{147.2} \times 100\% = 0.002$；

Java：$\frac{0.373}{147.2} \times 100\% = 0.253$；

C（未优化）：$\frac{0.405}{147.2} \times 100\% = 0.002$；

C（调整循环顺序）：$\frac{0.626}{147.2} \times 100\% = 0.425$；

C（编译优化）：$\frac{7.272}{147.2} \times 100\% = 4.940$

通过上述各项数据计算得出以下表格：

版本	实现	运行时间（s）	相对加速比（相对前一版本）	绝对加速比（相对版本 1）	浮点性能（GFLOPS）	达到峰值性能的百分比
1	Python	574.154	1	1	0.003	0.002
2	Java	11.464	50.083	50.083	0.373	0.253
3	C	4.740	1.085	121.129	0.405	0.275
4	C 调整循环顺序	3.065	1.546	187.326	0.626	0.425
5	C 编译优化	0.264	11.610	2174.826	7.272	4.940

4. 实验感想

这次实验让我对不同编程语言在性能上的差异有了更加深入的理解。简单一个矩阵乘法的代码可以有很多种方法提升性能，过程中我学到的是选择编程语言不仅仅是考虑其性能，还要权衡开发效率、可维护性以及具体应用场景。通过这次高性能计算实验中还让我深刻体会到，优化不仅仅是编写高效代码，还包括对算法和数据结构的深刻理解，以及对编译器的巧妙利用。总括而言，不同编程语言各有其长处，合理选择和运用它们能够极大地提升我们的工作效率和代码质量。