中山大学计算机院本科生实验报告

（2023 学年春季学期）

课程名称：并行程序设计　　**批改人：**

实验	**环境设置与串行矩阵乘法**	专业（方向）	**计算机科学与技术**
学号	21307082	姓名	**赖耿桂**
Email	laigg@mail2.sysu.edu.cn	完成日期	2024.3.23

1. 实验目的（200 字以内）

①使用 C/C++实现串行矩阵乘法。

输入： m, n, k 三个整数，每个整数的取值范围均为[512, 2048]

问题描述： 随机生成 m × n 的矩阵 A 及 n × k 的矩阵 B，并对这两个矩阵进行矩阵乘法运算，得到矩阵 C.

输出： A, B, C 三个矩阵，及矩阵计算所消耗的时间 t。

②根据表格要求，使用 python 语言实现串行矩阵乘法。

③对矩阵乘法进行优化如调整循环顺序、进行编译优化、循环展开、使用 Intel MKL 库等。

2. 实验过程和核心代码（600 字以内，图文并茂）

2.1 实验环境

系统环境：Linux

虚拟机操作系统：Ubuntu 18.04

所使用的编程语言：C++、Python

2.2 核心代码

2.2.1 Python 版本通用矩阵乘法

在 Generator.py 文件中定义了两个函数，分别用来随机生成指定大小的矩阵（使用二维列表存储数据）和打印矩阵，代码如下：

```
# 随机生成大小为rows * cols 的双精度浮点数矩阵
def random_matrix_generator(rows, cols):
    matrix = []
    for _ in range(rows):
        row = []    # 行
        for _ in range(cols):
```

```
            row.append(random.random() * 10 - 5)       # random.random()函数返回一个[0,1]的浮点数x，x * 10 - 5 代表[-5,5]的浮点数
        matrix.append(row)
    return matrix

# 打印输出指定矩阵
def printMatrix(A):
    for i in range(len(A)):
        if i == 0:
            print("[")
        print(A[i])
        if i == len(A)-1:
            print("]")
        else:
            print("\n")
```

然后在 Matrix_Calculator.py 文件中，定义了一个进行串行矩阵乘法运算并计算运行时间的函数 serial_time，代码如下：

```
import Generator
import time
def serial_time(A, B):
    M, N, K = len(A), len(B), len(B[0]) # 获取矩阵维度M，N，K
    C = [[0.0 for _ in range(K)] for _ in range(M)]   # 初始化结果矩阵C（M*K 矩阵）
    start_time = time.time()    # 记录矩阵乘法开始时刻
    for i in range(M):
        for k in range(K):
            for j in range(N):
                C[i][k] = C[i][k] + A[i][j] * B[j][k]
    end_time = time.time()  # 记录矩阵乘法结束时刻
    Generator.printMatrix(C)    # 打印输出矩阵C
    return end_time-start_time     # 返回矩阵乘法所用时间
```

然后输入 M、N、K，生成大小为 M*N 的矩阵 A 和大小为 N*K 的矩阵 B，并打印输出它们，然后调用函数 serial_time 计算计算 AB 的时间并打印输出，代码如下：

```
print("请输入 3 个正整数 M、N、K（矩阵 A 的大小为 M * K，矩阵 B 的大小为 N * K；M、N、K 的范围为[512,2048]）")
M = int(input("请输入 M："))
N = int(input("请输入 N："))
K = int(input("请输入 K："))

# 生成矩阵 A 并打印输出 A
A = Generator.random_matrix_generator(M,N)
Generator.printMatrix(A)

# 生成矩阵 B 并打印输出 B
B = Generator.random_matrix_generator(N,K)
Generator.printMatrix(B)

STime = serial_time(A, B)   # 使用通用串行矩阵乘法计算 AB 所用的时间
#  ATime = adjust_order_time(A, B)     # 使用调整循环顺序后的串行矩阵乘法计算 AB 所用的时间
```

```
print("使用 python 实现串行矩阵乘法所用时间为："+str(STime)+" s")
```

2.2.2 C++版本通用矩阵乘法

在 Generator.hpp 文件中，同样定义了两个函数用来随机生成指定大小的矩阵和打印指定的矩阵，代码如下：

```cpp
#ifndef GENERATOR_HPP
#define GENERATOR_HPP

#include<iostream>
#include<random>
#include<vector>
using namespace std;

// 随机生成一个 row s* cols 的双精度浮点数矩阵
vector<vector<double>>  random_matrix_generator(int rows, int cols){
    vector<vector<double>> matrix(rows, vector<double>(cols,0));
    // 初始化随机数生成器
    random_device rd;
    default_random_engine eng(rd());
    uniform_real_distribution<double> distr(-50.0,50.0);

    for(int i = 0; i < rows; ++i){
        for(int j = 0; j < cols; ++j){
            matrix[i][j] = distr(eng);
        }
    }
    return matrix;
}

// 打印输出矩阵
void printMatrix(vector<vector<double>> M, int rows, int cols){
    for(int i = 0;i < rows;++i){
        for(int j = 0; j < cols; ++j){
            cout << M[i][j] <<" ";
        }
        cout<<endl;
    }
}
#endif
```

在 Matrix_Calculator.cpp 中同样定义了一个函数 serial_time 来计算使用通用串行矩阵乘法所使用的时间，代码如下：

```cpp
#include "Generator.hpp"
#include <iostream>
#include <ctime>
using namespace std;

// 计算通用串行矩阵乘法所用时间
double serial_time(vector<vector<double>> A, vector<vector<double>> B){
```

```
    clock_t start_time, end_time;   // start_time：记录乘法运算开始的时刻； end_time：记录乘法运算结束的时刻
    int M = A.size(), N = B.size(), K = B[0].size();    // 获取矩阵维度M，N，K
    vector<vector<double>> C(M, vector<double>(K,0));       // C=AB，C 初始化为全 0 矩阵
    start_time = clock();
    for(int i = 0; i < M; ++i){
        for(int k = 0; k < K; ++k){
            for(int j = 0; j < N; ++j){
                C[i][k] += A[i][j] * B[j][k];       // 按照定义计算矩阵乘法的结果
            }
        }
    }
    end_time = clock();
    double using_time = (double)(end_time - start_time)/ CLOCKS_PER_SEC;     // 计算乘法运算所耗费的时间(s)
    printMatrix(C, M, K);
    return using_time;
}
```

然后在 main 函数中，同样先输入 M、N、K，然后生成矩阵 A、B 并打印输出它们，最后使用通用串行矩阵乘法计算 AB，记录所用时间并打印输出，代码如下：

```
int main(){
    int M, N, K;
    cout<<"请输入 3 个整数 M,N,K(矩阵 A 的大小为 M*N,矩阵 B 的大小为 N*K;M,N,K 的范围为[512, 2048]): "<<endl;
    cin >> M >> N >> K;
    vector<vector<double>> A, B;
    // 随机生成矩阵 A、B
    A = random_matrix_generator(M,N);
    B = random_matrix_generator(N,K);
    printMatrix(A, M, N);
    printMatrix(B, N, K);
    double STime = serial_time(A,B);    // STime：使用通用串行矩阵乘法计算 AB 的时间
    cout<<"使用通用串行矩阵乘法所用的时间为："<<STime<<" s"<<endl;
    return 0;
}
```

2.2.3 C++版本调整循环后的矩阵乘法

在 Matrix_Calculator.cpp 中定义了一个函数 adjust_order_time，其用于在调整循环顺序后进行矩阵乘法并计算所耗时间，代码如下：

```
// 计算调整循环顺序后的串行矩阵乘法运算的时间
double adjust_order_time(vector<vector<double>> A, vector<vector<double>> B){
    clock_t start_time, end_time;   // start_time：记录乘法运算开始的时刻； end_time：记录乘法运算结束的时刻
    int M = A.size(), N = B.size(), K = B[0].size();    // 获取矩阵维度M，N，K
    vector<vector<double>> C(M, vector<double>(K,0));       // C=AB，C 初始化为全 0 矩阵
    start_time = clock();
    for(int i = 0; i  <  M; ++i){
        for(int j = 0; j < N; ++j){
            for(int k = 0; k < K;++k){
                C[i][k] += A[i][j] * B[j][k];
```

```
            }
        }
    }
    end_time = clock();
    double using_time = (double)(end_time - start_time)/ CLOCKS_PER_SEC;    // 计算乘法运算所耗费的时间(s)
    printMatrix(C, M, K);
    return using_time;
}
```

同时修改 main 函数：

```
int main(){
    int M, N, K;
    cout<<"请输入 3 个整数 M,N,K(矩阵 A 的大小为 M*N,矩阵 B 的大小为 N*K;M,N,K 的范围为[512, 2048]): "<<endl;
    cin >> M >> N >> K;
    vector<vector<double>> A, B;
    // 随机生成矩阵 A、B
    A = random_matrix_generator(M,N);
    B = random_matrix_generator(N,K);
    printMatrix(A, M, N);
    printMatrix(B, N, K);
//    double STime = serial_time(A,B);    // STime：使用通用串行矩阵乘法计算 AB 的时间
    double ATime = adjust_order_time(A,B);      // ATime：使用调整循环顺序的串行矩阵乘法计算 AB 的时间
//    cout<<"使用通用串行矩阵乘法所用的时间为："<<STime<<" s"<<endl;
     cout<<"使用调整循环顺序后的串行矩阵乘法所用的时间为："<<ATime<<" s"<<endl;
    return 0;
}
```

2.2.4 对通用串行矩阵乘法使用编译优化

需要对 main 函数进行修改（需要在终端对编译命令进行修改，见 2.3.4）：

```
int main(){
    int M, N, K;
    cout<<"请输入 3 个整数 M,N,K(矩阵 A 的大小为 M*N,矩阵 B 的大小为 N*K;M,N,K 的范围为[512, 2048]): "<<endl;
    cin >> M >> N >> K;
    vector<vector<double>> A, B;
    // 随机生成矩阵 A、B
    A = random_matrix_generator(M,N);
    B = random_matrix_generator(N,K);
    printMatrix(A, M, N);
    printMatrix(B, N, K);
    double STime = serial_time(A,B);    // STime：使用通用串行矩阵乘法计算 AB 的时间
//    double ATime = adjust_order_time(A,B);      // ATime：使用调整循环顺序的串行矩阵乘法计算 AB 的时间
    cout<<"进行编译优化后使用通用串行矩阵乘法所用的时间为："<<STime<<" s"<<endl;
//     cout<<"使用调整循环顺序后的串行矩阵乘法所用的时间为："<<ATime<<" s"<<endl;
    return 0;
}
```

附：编译优化等级及简要说明如下：

优化等级	简要说明
-O0	不做任何优化，默认的编译选项
-O1	优化会消耗少多的编译时间，它主要对代码的分支，常量以及表达式等进行优化
-O2	会尝试更多的寄存器级的优化以及指令级的优化，它会在编译期间占用更多的内存和编译时间
-O3	在 O2 的基础上进行更多的优化例如使用伪寄存器网络，普通函数的内联，以及针对循环的更多优化
-OS	相当于-O2.5。是使用了所有-O2 的优化选项，但又不缩减代码尺寸的方法。

2.2.5 对通用串行矩阵乘法进行循环展开

由于共有三层循环，我尝试对三层循环都进行了展开，定义了一个新函数 loop_unrolling_time（间隔为 4），用于计算进行循环展开后的通用矩阵乘法所用时间：

①对最内层循环进行展开：

```
1.   // 计算循环展开串行矩阵乘法所用时间
2.   double loop_unrolling_time(vector<vector<double>> A, vector<vector<double>> B){
3.       clock_t start_time, end_time;   // start_time：记录乘法运算开始的时刻； end_time：记录乘法运算结束的时刻
4.       int M = A.size(), N = B.size(), K = B[0].size();    // 获取矩阵维度M，N，K
5.       vector<vector<double>> C(M, vector<double>(K,0));       // C=AB，C 初始化为全 0 矩阵
6.       start_time = clock();
7.       for(int i = 0; i < M; ++i){
8.           for(int k = 0; k < K; ++k){
9.               for(int j = 0; j < N; j+=4){    // 对最内层循环进行展开，间隔为4
10.                  C[i][k] += A[i][j] * B[j][k];       // 按照定义计算矩阵乘法的结果
11.                  C[i][k] += A[i][j+1] * B[j+1][k];
12.                  C[i][k] += A[i][j+2] * B[j+2][k];
13.                  C[i][k] += A[i][j+3] * B[j+3][k];
```

```
            }
        }
    }
    end_time = clock();
    double using_time = (double)(end_time - start_time)/ CLOCKS_PER_SEC;      // 计算乘法运算所耗费的时间(s)
    printMatrix(C, M, K);
    return using_time;
}
```

②对第二层循环进行展开：

```
// 计算循环展开串行矩阵乘法所用时间
double loop_unrolling_time(vector<vector<double>> A, vector<vector<double>> B){
    clock_t start_time, end_time;   // start_time：记录乘法运算开始的时刻； end_time：记录乘法运算结束的时刻
    int M = A.size(), N = B.size(), K = B[0].size();    // 获取矩阵维度M，N，K
    vector<vector<double>> C(M, vector<double>(K,0));       // C=AB，C 初始化为全 0 矩阵
    start_time = clock();
    for(int i = 0; i < M; ++i){
        for(int k = 0; k < K; k+=4){    //  对第二层循环进行展开，间隔为4
            for(int j = 0; j < N; ++j){
                C[i][k] += A[i][j] * B[j][k];       // 按照定义计算矩阵乘法的结果
                C[i][k+1] += A[i][j] * B[j][k+1];
                C[i][k+2] += A[i][j] * B[j][k+2];
                C[i][k+3] += A[i][j] * B[j][k+3];
            }
        }
    }
    end_time = clock();
    double using_time = (double)(end_time - start_time)/ CLOCKS_PER_SEC;      // 计算乘法运算所耗费的时间(s)
    printMatrix(C, M, K);
    return using_time;
}
```

③对最外层循环进行展开：

```
// 计算循环展开串行矩阵乘法所用时间
double loop_unrolling_time(vector<vector<double>> A, vector<vector<double>> B){
    clock_t start_time, end_time;   // start_time：记录乘法运算开始的时刻； end_time：记录乘法运算结束的时刻
    int M = A.size(), N = B.size(), K = B[0].size();    // 获取矩阵维度M，N，K
    vector<vector<double>> C(M, vector<double>(K,0));       // C=AB，C 初始化为全 0 矩阵
    start_time = clock();
    for(int i = 0; i < M; i+=4){    //  对最外层循环进行展开，间隔为4
        for(int k = 0; k < K; ++k){
            for(int j = 0; j < N; ++j){
                C[i][k] += A[i][j] * B[j][k];       // 按照定义计算矩阵乘法的结果
                C[i+1][k] += A[i+1][j] * B[j][k];
                C[i+2][k] += A[i+2][j] * B[j][k];
                C[i+3][k] += A[i+3][j] * B[j][k];
            }
        }
    }
```

```
    end_time = clock();
    double using_time = (double)(end_time - start_time)/ CLOCKS_PER_SEC;      // 计算乘法运算所耗费的时间(s)
    printMatrix(C, M, K);
    return using_time;
}
```

此外还需要修改 main 函数：

```
int main(){
    int M, N, K;
    cout<<"请输入 3 个整数 M,N,K(矩阵 A 的大小为 M*N,矩阵 B 的大小为 N*K;M,N,K 的范围为[512, 2048]): "<<endl;
    cin >> M >> N >> K;
    vector<vector<double>> A, B;
    // 随机生成矩阵 A、B
    A = random_matrix_generator(M,N);
    B = random_matrix_generator(N,K);
    printMatrix(A, M, N);
    printMatrix(B, N, K);
  //   double STime = serial_time(A,B);     // STime：使用通用串行矩阵乘法计算 AB 的时间
//    double ATime = adjust_order_time(A,B);      // ATime：使用调整循环顺序的串行矩阵乘法计算 AB 的时间
    double LUTime = loop_unrolling_time(A,B);
// cout<<"使用通用串行矩阵乘法所用的时间为："<<STime<<" s"<<endl;
//     cout<<"使用调整循环顺序后的串行矩阵乘法所用的时间为："<<ATime<<" s"<<endl;
  //   cout<<"进行编译优化后使用通用串行矩阵乘法所用的时间为："<<STime<<" s"<<endl;
    cout<<"进行循环展开后使用通用串行矩阵乘法所用的时间为："<<LUTime<<" s"<<endl;
    return 0;
}
```

2.2.6 使用 Intel MKL 库

在 Matrix_Calculator.cpp 中导入头文件 mkl.h：

```
#include"mkl.h"
```

定义一个函数 MKL_cblas_dgemm_time 用来调用 Intel MKL 库并计算矩阵乘法所用时间，代码如下：

```
// 使用 Intel MKL 库，使用 dgemm 进行矩阵计算
double MKL_cblas_dgemm_time(int M, int N, int K, double alpha, double beta){
    double *A, *B, *C;      // 3 个矩阵

    // 分配对齐的内存缓存区
    A = (double*)mkl_malloc(M * N *sizeof(double), 64);
    B = (double*)mkl_malloc(N * K *sizeof(double), 64);
    C = (double*)mkl_malloc(M * K *sizeof(double), 64);

    if (A == NULL || B == NULL || C == NULL) {
     cout<<"\n ERROR: Can't allocate memory for matrices. Aborting... \n\n";
      mkl_free(A);
      mkl_free(B);
      mkl_free(C);
      return 1;
    }
```

```
17.
18.     // 初始化随机数生成器
19.     random_device rd;
20.     default_random_engine eng(rd());
21.     uniform_real_distribution<double> distr(-50.0,50.0);
22.
23.     // 给矩阵A赋值并打印出矩阵A
24.     for(int i = 0; i < M * N; ++i){
25.         A[i] = distr(eng);
26.         cout<<A[i]<<" ";
27.     }
28.     cout<<endl;
29.
30.     // 给矩阵B赋值并打印出矩阵B
31.     for(int i = 0; i < N * K; ++i){
32.         B[i] = distr(eng);
33.     }
34.     cout<<endl;
35.
36.     // 矩阵C初始化为全0矩阵
37.     for(int i = 0; i < M * K; ++i){
38.         C[i] = 0;
39.     }
40.
41.     clock_t start_time, end_time;
42.     start_time = clock();
43.     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, M, K, N, alpha, A, N, B,K, beta, C, K);
    // 调用cblas_dgemm，各参数含义见官网
44.     end_time = clock();
45.
46.     // 打印输出矩阵C
47.     for(int i = 0; i < M * K; ++i){
48.         cout<<C[i]<<" ";
49.     }
50.     cout<<endl;
51.
52.     // 释放分配的缓存
53.     mkl_free(A);
54.     mkl_free(B);
55.     mkl_free(C);
56.     double using_time = (double)(end_time - start_time)/ CLOCKS_PER_SEC;     // 计算乘法运算所耗费的
    时间(s)
57.     return using_time;
58. }
```

注：参照 cblas_dgemm 示范代码，因此使用 mkl_malloc 来分配矩阵的缓存，故不再使用此前生成矩阵的函数。

另外，对 main 函数进行修改，使其能够选择是否调用 Intel MKL 库，代码如下：

```
1.  int main(){
2.     int M, N, K;
3.     cout<<"请输入3个整数M,N,K(矩阵A的大小为M*N,矩阵B的大小为N*K;M,N,K的范围为[512, 2048]): "<<endl;
```

```
4.      cin >> M >> N >> K;
5.      int if_using_mkl = 0;
6.      cout<<"是否使用 Intel MKL 库？若是请输入 1，否则请输入 0： ";
7.      cin>>if_using_mkl;
8.      if(if_using_mkl == 1){
9.          double alpha, beta;
10.         cout<<"接下来请输入两个比例因子 alpha 和 beta，用于计算双精度矩阵的乘积：dgemm. 其计算公式为 C = alpha*AB + beta*C"<<endl;
11.         cout<<"请输入矩阵 A 和 B 乘积的比例因子 alpha： ";
12.         cin>>alpha;
13.         cout<<"请输入矩阵 C 的比例因子 beta： ";
14.         cin>>beta;
15.         double MKL_time = MKL_cblas_dgemm_time(M,N,K,alpha,beta);
16.         cout<<"调用 Intel MKL 库进行矩阵乘法运算所用时间为： "<<MKL_time<<" s"<<endl;
17.     }
18.     else{
19.         vector<vector<double>> A, B;
20.     // 随机生成矩阵 A、B
21.         A = random_matrix_generator(M,N);
22.         B = random_matrix_generator(N,K);
23.         printMatrix(A, M, N);
24.         printMatrix(B, N, K);
25.     //   double STime = serial_time(A,B);     // STime：使用通用串行矩阵乘法计算 AB 的时间
26.     //    double ATime = adjust_order_time(A,B);      // ATime：使用调整循环顺序的串行矩阵乘法计算 AB 的时间
27.         double LUTime = loop_unrolling_time(A,B);
28.     // cout<<"使用通用串行矩阵乘法所用的时间为： "<<STime<<" s"<<endl;
29.     //     cout<<"使用调整循环顺序后的串行矩阵乘法所用的时间为： "<<ATime<<" s"<<endl;
30.     //   cout<<"进行编译优化后使用通用串行矩阵乘法所用的时间为： "<<STime<<" s"<<endl;
31.         cout<<"进行循环展开后使用通用串行矩阵乘法所用的时间为： "<<LUTime<<" s"<<endl;
32.     }
33.     return 0;
34. }
```

2.3 实验过程

2.3.1 Python 版本通用矩阵乘法

```
laigg@laigg-VirtualBox:~/codes/parallel/lab0$ python3 Matrix_Calculator.py
请输入3个正整数M、N、K（矩阵A的大小为M * K，矩阵B的大小为N * K；M、N、K的范围为[512,2048]）
请输入M：1024
请输入N：1024
请输入K：1024
```

2.3.2 C++版本通用矩阵乘法

```
laigg@laigg-VirtualBox:~/codes/parallel/lab0$ g++ -o Matrix_Calculator Matrix_Ca
lculator.cpp
laigg@laigg-VirtualBox:~/codes/parallel/lab0$ ./Matrix_Calculator
请输入3个整数M,N,K(矩阵A的大小为M*N，矩阵B的大小为N*K；M，N，K的范围为[512，2048
]):
1024 1024 1024
```

2.3.3 C++版本调整循环后的矩阵乘法

```
laigg@laigg-VirtualBox:~/codes/parallel/lab0$ g++ -o Matrix_Calculator Matrix_Ca
lculator.cpp
laigg@laigg-VirtualBox:~/codes/parallel/lab0$ ./Matrix_Calculator
请输入3个整数M,N,K(矩阵A的大小为M*N，矩阵B的大小为N*K；M，N，K的范围为[512, 2048
]):
1024 1024 1024
```

2.3.4 对通用串行矩阵乘法使用编译优化

此处要修改编译命令为：

```
1.  g++ -O3 -o Matrix_Calculator Matrix_Calculator.cpp
```

即使用-O3 优化

```
laigg@laigg-VirtualBox:~/codes/parallel/lab0$ g++ -O3 -o Matrix_Calculator Matri
x_Calculator.cpp
laigg@laigg-VirtualBox:~/codes/parallel/lab0$ ./Matrix_Calculator
请输入3个整数M,N,K(矩阵A的大小为M*N，矩阵B的大小为N*K；M，N，K的范围为[512, 2048
]):
1024 1024 1024
```

2.3.5 对通用串行矩阵乘法进行循环展开

```
laigg@laigg-VirtualBox:~/codes/parallel/lab0$ g++ -o Matrix_Calculator Matrix_Ca
lculator.cpp
laigg@laigg-VirtualBox:~/codes/parallel/lab0$ ./Matrix_Calculator
请输入3个整数M,N,K(矩阵A的大小为M*N，矩阵B的大小为N*K；M，N，K的范围为[512, 2048
]):
1024 1024 1024
```

2.3.6 使用 Intel MKL 库

需要在.bashrc 文件中增加一行：

```
1.  sudo gedit .bashrc
2.  source /opt/intel/oneapi/setvars.sh
```

增加该语句后每次打开终端都会自动对当前 shell 对话进行环境变量设置

此处要修改编译命令为：

```
1.  g++ -o Matrix_Calculator Matrix_Calculator.cpp -l mkl_rt
```

增加项

```
1.  -l mkl_rt
```

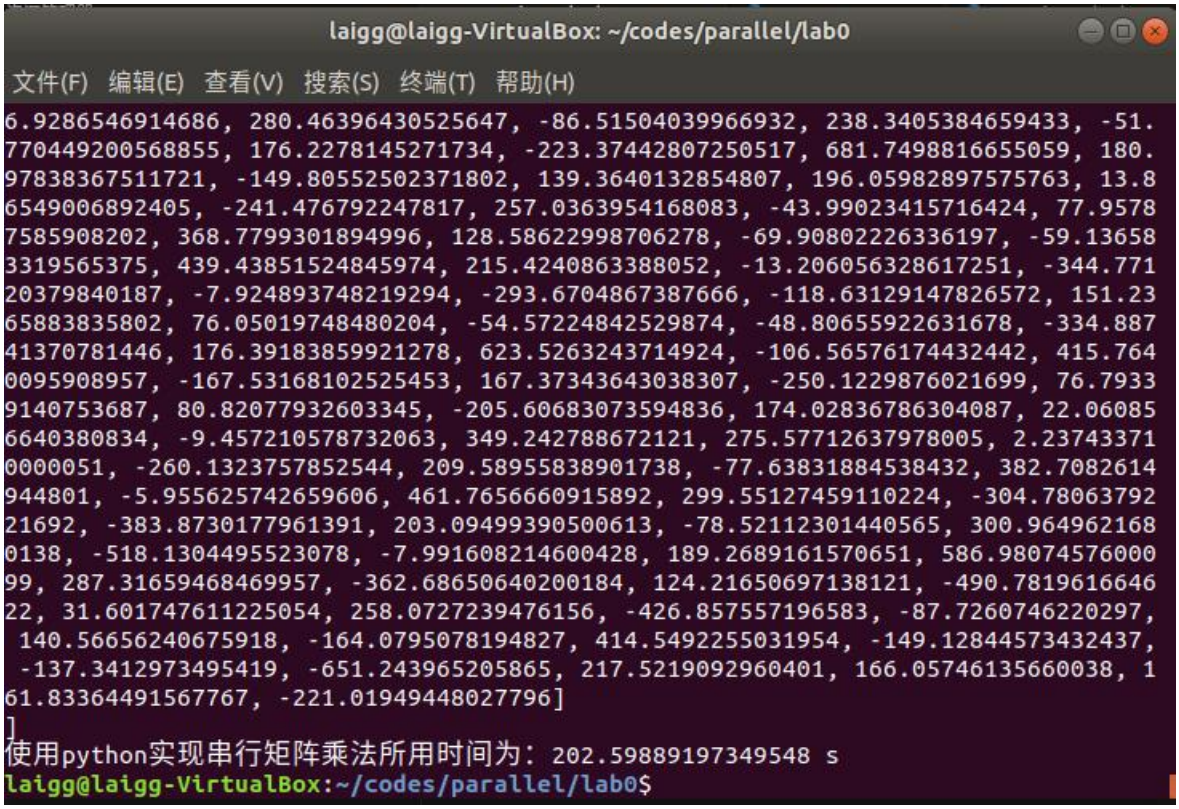
动态链接库

```
laigg@laigg-VirtualBox:~/codes/parallel/lab0$ g++ -o Matrix_Calculator Matrix_Ca
lculator.cpp -l mkl_rt
laigg@laigg-VirtualBox:~/codes/parallel/lab0$ ./Matrix_Calculator
请输入3个整数M,N,K(矩阵A的大小为M*N，矩阵B的大小为N*K；M，N，K的范围为[512, 2048
]):
1024 1024 1024
是否使用Intel MKL库？若是请输入1，否则请输入0：1
接下来请输入两个比例因子alpha和beta，用于计算双精度矩阵的乘积：dgemm. 其计算公式
为C = alpha*AB + beta*C
请输入矩阵A和B乘积的比例因子alpha：1.0
请输入矩阵C的比例因子beta：0.0
```

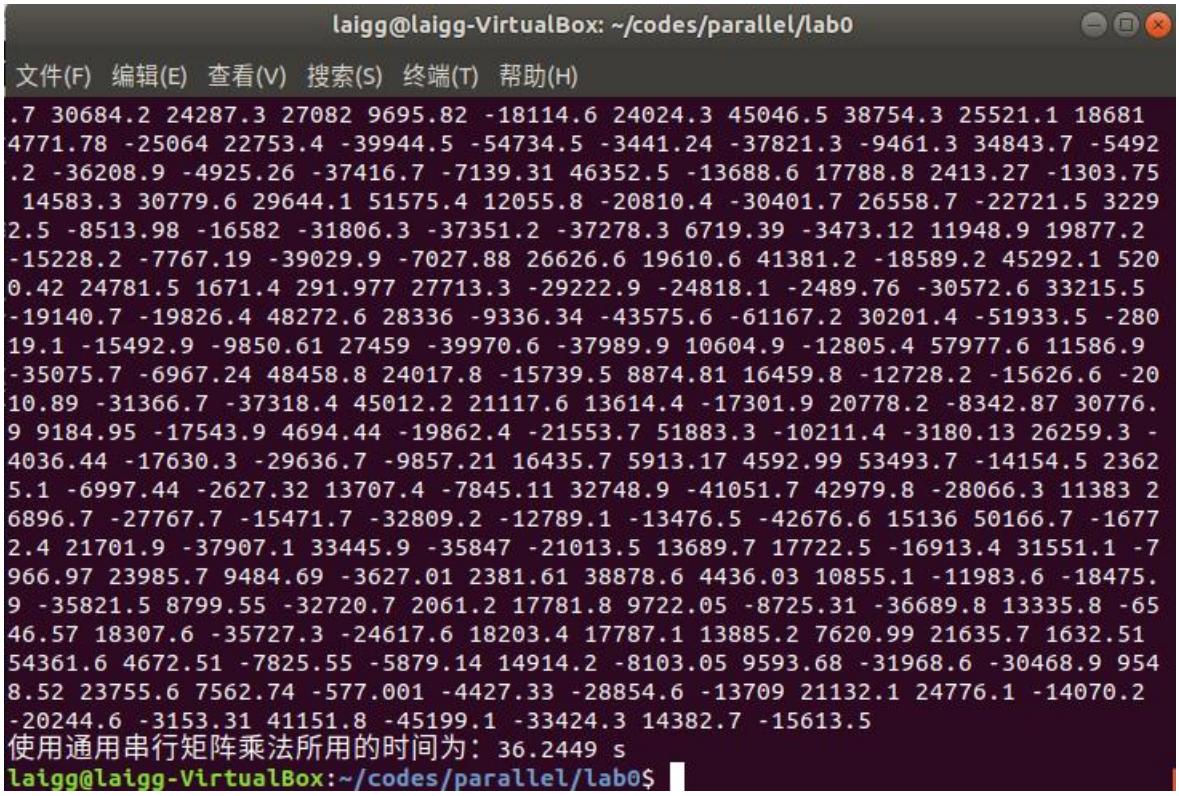
由于实现的是简单的串行矩阵乘法，因此将 alpha 设置为 1.0，beta 设置为 0.0 也即 C=AB 即可。

3. 实验结果（500 字以内，图文并茂）

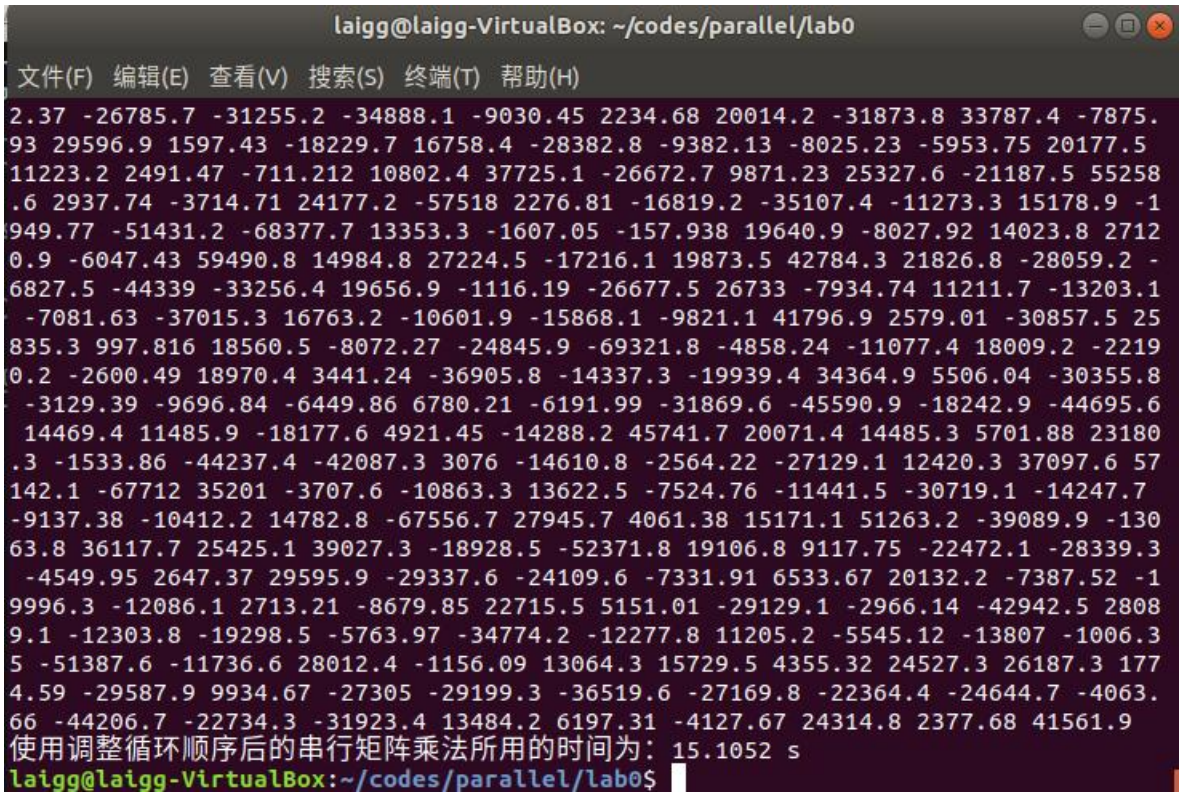
3.1 Python 版本通用矩阵乘法



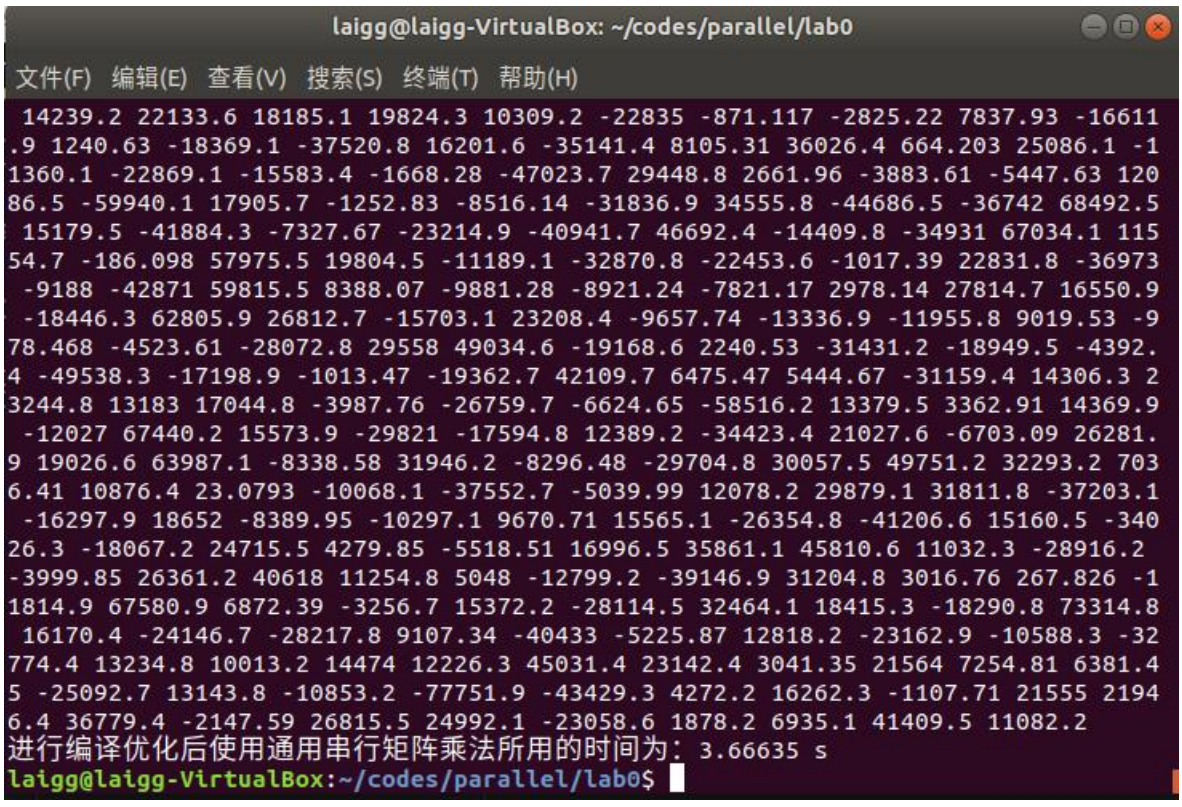
3.2 C++版本通用矩阵乘法



3.3 C++版本调整循环顺序后矩阵乘法

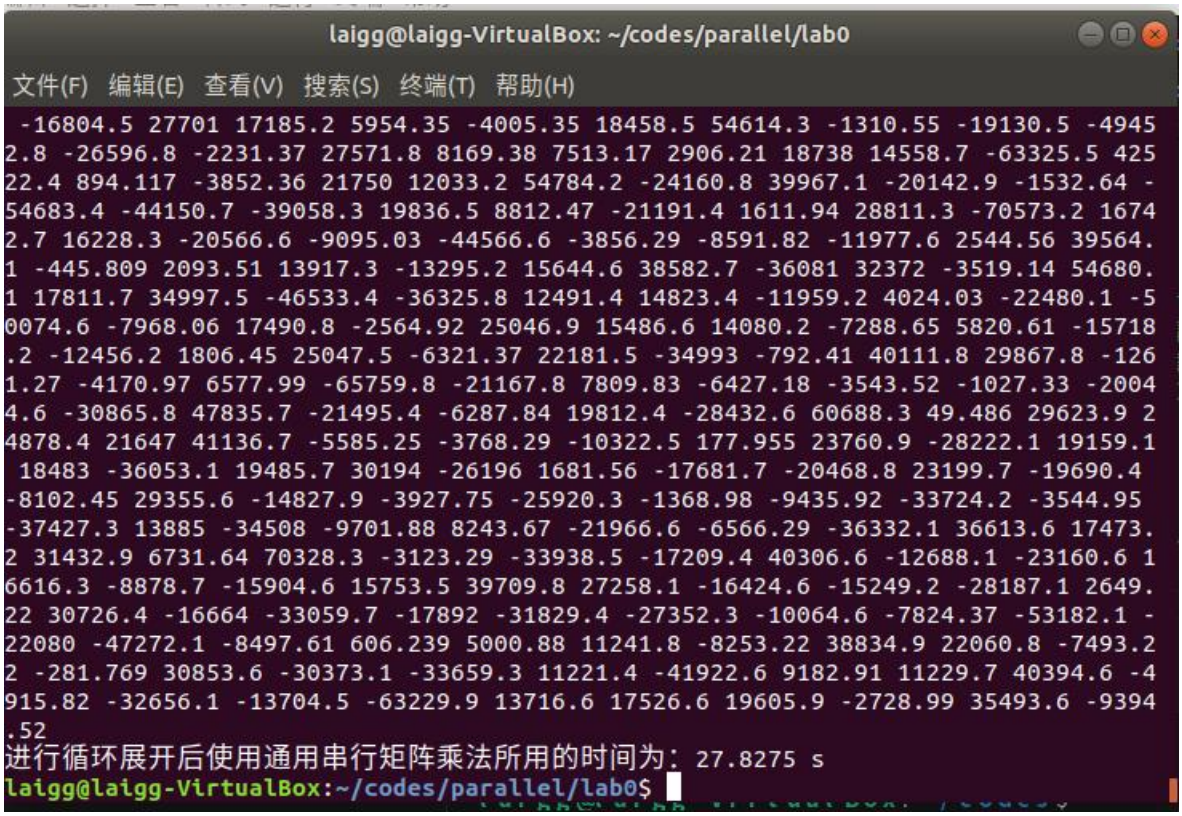


3.4 对通用串行矩阵乘法使用编译优化

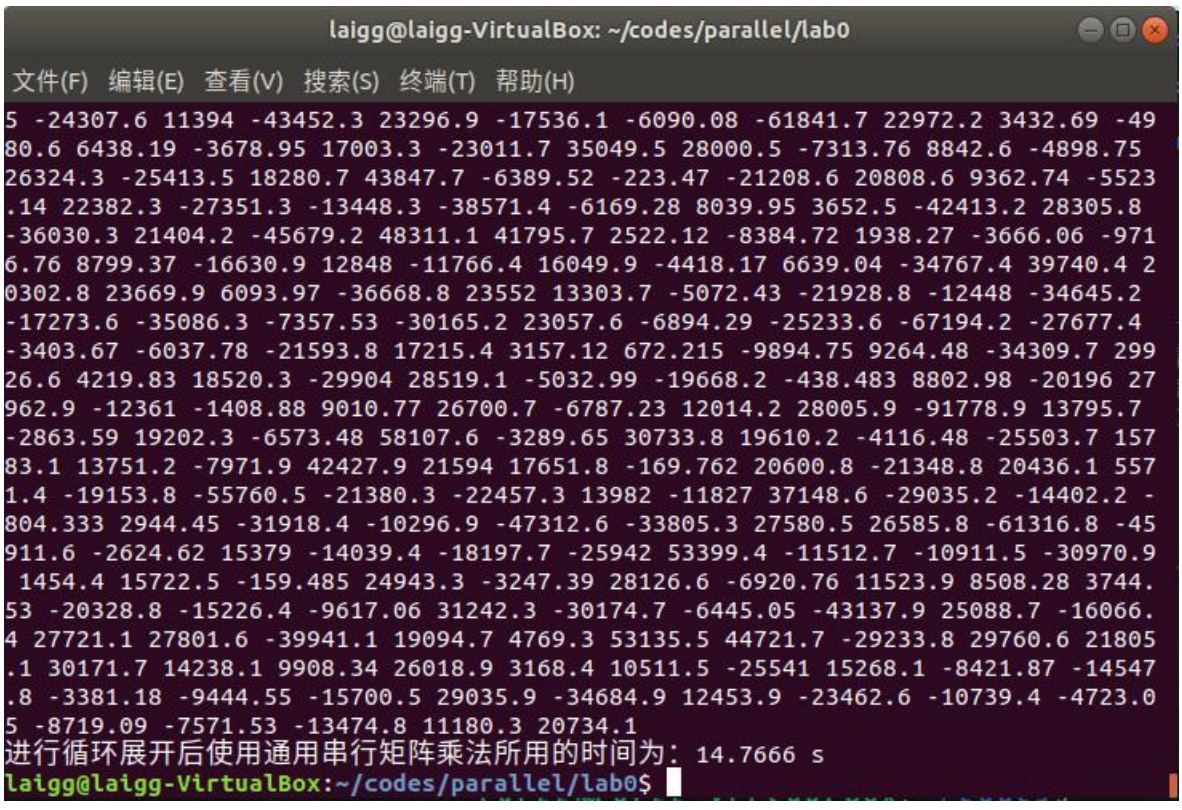


3.5 对通用串行矩阵乘法进行循环展开

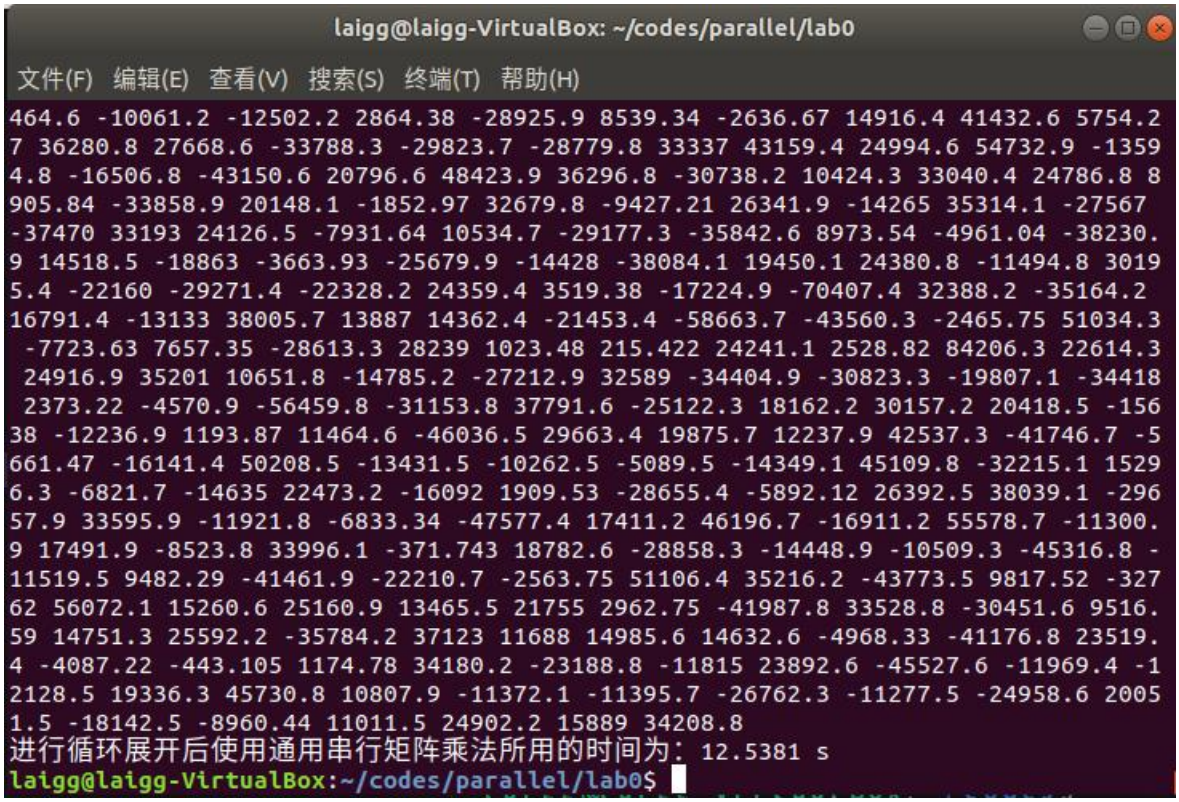
①对最内层循环进行展开：



②对第二层循环进行展开：



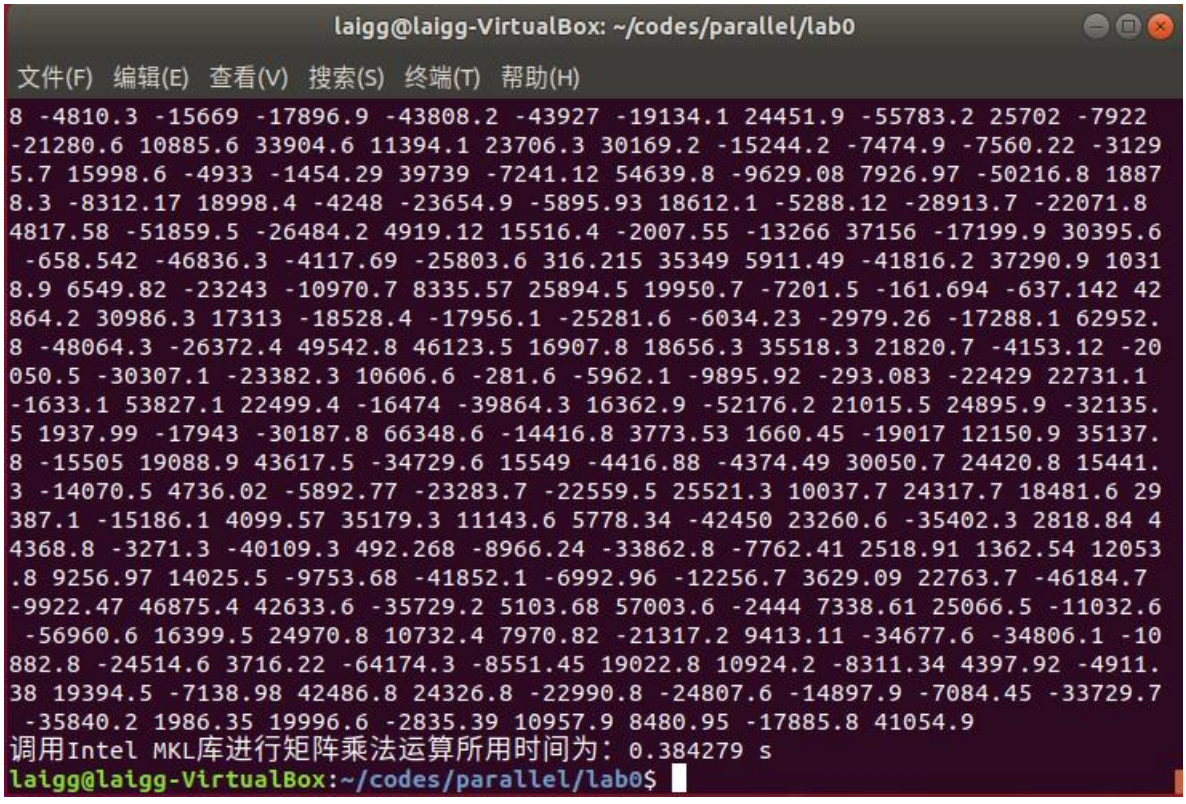
③对最外层循环进行展开：



通过比较可以看出对最外层循环进行展开效果最好，因为其减少的循环次数最多。

（在下表中取提高效率最多的“对最外层循环进行展开”的时间）

3.6 使用 Intel MKL 库



（下表记录的是 1024*1024 的矩阵 A 乘以 1024*1024 的矩阵 B 的运行时间）

在终端输入以下命令，查看 CPU 信息：

```
1.  lscpu
```

```
laigg@laigg-VirtualBox:~$ lscpu
架构:                    x86_64
CPU 运行模式:            32-bit, 64-bit
字节序:                  Little Endian
CPU:                     4
在线 CPU 列表:           0-3
每个核的线程数:          1
每个座的核数:            4
座:                      1
NUMA 节点:               1
厂商 ID:                 GenuineIntel
CPU 系列:                6
型号:                    140
型号名称:                11th Gen Intel(R) Core(TM) i5-1155G7 @ 2.50GHz
步进:                    2
CPU MHz:                 2496.000
BogoMIPS:                4992.00
超管理器厂商:            KVM
虚拟化类型:              完全
L1d 缓存:                48K
L1i 缓存:                32K
L2 缓存:                 1280K
L3 缓存:                 8192K
NUMA 节点0 CPU:          0-3
标记:                    fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc
 rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervis
or lahf_lm abm 3dnowprefetch invpcid_single fsgsbase bmi1 avx2 bmi2 invpcid rdseed clflushopt arat md_clear flush_l1d arch_capabilities
laigg@laigg-VirtualBox:~$
```

在查阅 Intel 官方文档 APP-for-Intel-Core-Processors.pdf 后，该型号的 CPU 的频率为 4.5GHz 而我的虚拟机有 4 个 CPU，每个 CPU 的核数为 1、线程数为 1，浮点运算单元为 8，因此峰值 GFLOPS 为 144。

各版本 GLFOPS 的计算（以 Python 版本为例）：对 A 的一行和 B 的一列，需要进行 1024 次乘法和 1023 次加法，A 共有 1024 行，B 共有 1024 列，因此共有(1024+1023)*1024*1024 次浮点运算。而运行时间为 202.59889197349548s，因此可算出其浮点性能为 (1024+1023)*1024*1024*10^(-9)/202.59889197349548，保留小数点后 6 位为 0.010595 GFLOPS。

版本	实现描述	运行时间 (sec.)	相对加速比	绝对加速比	浮点性能 (GFLOPS)	峰值性能百分比
1	Python	202.59889197349548	1	1	0.010595	0.007358%
2	C++	36.2449	5.5897	5.5897	0.059223	0.041127%
3	调整循环顺序	15.1052	2.3995	13.4125	0.142105	0.098684%
4	编译优化	3.66635	4.1200	55.2590	0.585524	0.406614%

5	循环展开	12.5381	0.2924	16.1587	0.171201	0.118890%
6	Intel MKL	0.384279	32.6276	527.2182	5.585877	3.879081%

4. 实验感想（200 字以内）

在此次实验中，我对串行矩阵乘法的实现以及如何优化有了更深的了解。此外在此次实验中，我也学习了解了计算双精度矩阵乘法 dgemm 的原理及实现。但是在此次实验中我也遇到了一些问题，在安装 Intel MKL 库后如何进行配置和编译上耗费了不少时间。

在此次实验中我也意识到了提高程序的时效性的重要性，而在此之前我只关注程序的正确性，这对我以后的学习和工作有深刻的启发。