组成原理实验课程第 5 次实验报告

实验名称	矩阵乘法			班级	李涛老师
学生姓名	申宗尚	学号	2213924	指导老师	董前琨
实验地点	实验 A306		实验时间	2024.5.18	

1、 实验目的

1. 熟悉并掌握矩阵乘法的优化方法
2. 初步了解 CUDA 编程在矩阵乘法中的应用
3. 掌握在不同环境中进行性能测试和分析的方法
4. 总结优化过程中的问题和解决方法
5. 学习使用 Taisan 服务器编程

2、 实验内容

1、个人 PC 电脑实验要求如下:

使用个人电脑完成，不仅限于 visual studio、vscode 等。

在完成矩阵乘法优化后，测试矩阵规模在 1024~4096，或更大维度上，至少进行 4 个矩阵规模 维度的测试。

如 PC 电脑有 Nvidia 显卡，建议尝试 CUDA 代码。

在作业中需总结出不同层次，不同规模下的矩阵乘法优化对比，对比指标包括计算耗时、运行性能、加速比等。

在作业中总结优化过程中遇到的问题和解决方式。

2、在 Taishan 服务器上使用 vim+gcc 编程环境，要求如下:

在 Taishan 服务器上完成，使用 Putty 等远程软件在校内登录使用，服务器 IP:222.30.62.23， 端口 22，用户名 stu+学号，默认密码 123456，登录成功后可自行修改密码。

在完成矩阵乘法优化后(使用 AVX 库进行子字优化在 Taishan 服务器上的软件包环境不好配置，可以不进行此层次优化操作，注意原始代码需要调整)，测试矩阵规模在 1024~4096，或更大维度上，至少进行 4 个矩阵规模维度的测试。

在作业中需总结出不同层次，不同规模下的矩阵乘法优化对比，对比指标包括计算耗时、运行 性能、加速比等。

在作业中需对比 Taishan 服务器和自己个人电脑上程序运行时间等相关指标，分析一下不同电脑上的运行差异的原因，总结在优化过程中遇到的问题和解决方式。

3、 实验过程

- **个人 PC 电脑：**

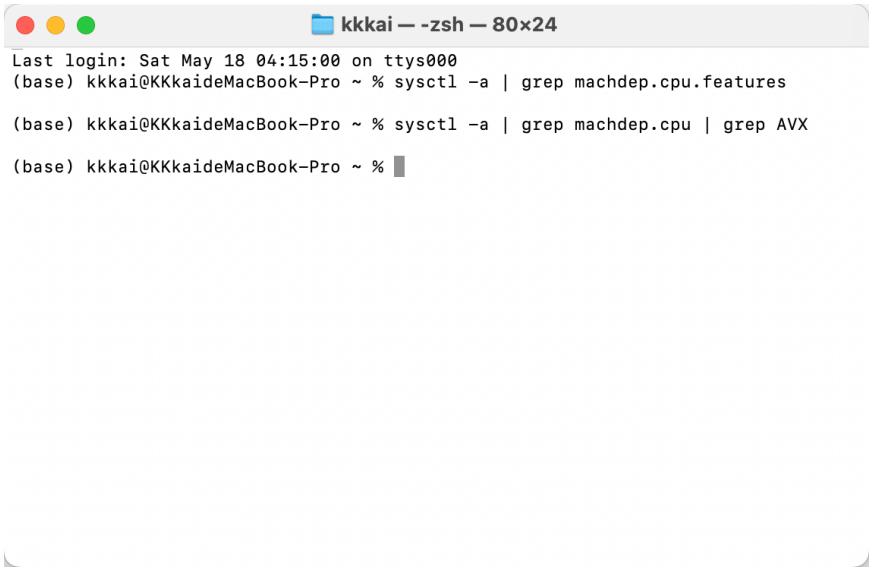
(1) MacOS 操作系统

在这部分，我将先对于矩阵乘法及其四个可能的优化进行简单的介绍和具体代码的实现，同时进行结果的数据分析。

然而，在我开始之前，我对 macbook 是否可以支持 AVX 指令集报怀疑态度，因此，我先尝试了查询硬件，通过 sysctl 指令查看 cpu 信息，在第一行指令 sysctl -a | grep machdep.cpu.features 后，如果支持 AVX，应该会：输出包含 AVX1.0 或 AVX2.0，然而，什么也没有。

第二条指令 sysctl -a | grep machdep.cpu | grep AVX 后，通过 grep 指令特定查询”AVX”，还是没有任何输出。

因此可以断定，MacOS(苹果 M2 芯片)，是不支持 AVX 指令集的



(2) Windows10 操作系统

首先，根据矩阵乘法，书写基础代码如下：

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <immintrin.h>
#include <cstring>
using namespace std;

#define REAL_T double

// 计算并输出计算时间和每秒浮点运算次数
void printFlops(int A_height, int B_width, int B_height, clock_t
start, clock_t stop) {
    double elapsed = double(stop - start) / CLOCKS_PER_SEC;
    cout << "Time:\t" << elapsed << " s" << endl;
    REAL_T flops = (2.0 * A_height * B_width * B_height) / 1E9 /
elapsed;
    cout << "Performance:\t" << flops << " GFLOPS" << endl << endl;
}

// 拷贝矩阵
void copyMatrix(int n, REAL_T *S_A, REAL_T *S_B, REAL_T *S_C, REAL_T
*D_A, REAL_T *D_B, REAL_T *D_C) {
    memcpy(D_A, S_A, n * n * sizeof(REAL_T));
    memcpy(D_B, S_B, n * n * sizeof(REAL_T));
    memcpy(D_C, S_C, n * n * sizeof(REAL_T));
}
```

```
// 用随机生成数初始化矩阵
void initMatrix(int n, REAL_T *A, REAL_T *B, REAL_T *C) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            A[i + j * n] = rand() / REAL_T(RAND_MAX);
            B[i + j * n] = rand() / REAL_T(RAND_MAX);
            C[i + j * n] = 0;
        }
// 定义算法
void dgemm(int n, REAL_T *A, REAL_T *B, REAL_T *C) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            REAL_T cij = C[i + j * n];
            for (int k = 0; k < n; k++) {
                cij += A[i + k * n] * B[k + j * n];
            }
            C[i + j * n] = cij;
        }
}
    int main() {
    srand(int(time(0)));
    REAL_T *A, *B, *C, *a, *b, *c;
    clock_t start, stop;
    int n = 1024; // 矩阵规模
    for (int i = 1; i < 5; i++) {
        if (i != 1)
            n += 1024;
        A = new REAL_T[n * n];
        a = new REAL_T[n * n];
        B = new REAL_T[n * n];
        b = new REAL_T[n * n];
        C = new REAL_T[n * n];
        c = new REAL_T[n * n];
        initMatrix(n, A, B, C); // 构造原始矩阵
        cout << "dimension: " << n << endl << endl;
        copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
        cout << "original calculation performance: \n";
        start = clock();
        dgemm(n, a, b, c);
        stop = clock();
        double origin_time = stop - start;
        printFlops(n, n, n, start, stop);
        // 释放内存
        delete[] A;
```

```
        delete[] a;
        delete[] B;
        delete[] b;
        delete[] C;
        delete[] c;
    }
    return 0;
}
```

这段代码中，主函数动态分配矩阵内存，调用函数初始化矩阵并逐步增加矩阵大小，实现 1024，2048，3072，4096 阶的矩阵乘法运算，同时，通过前后两次 clock()函数调用，输出运行时间和加速比。

随后，我们介绍四种矩阵乘法的优化算法和给出代码实现：

(1).子字并行(AVX Optimized DGEMM)

通过利用 AVX 指令集，并行处理多个浮点数，提高计算效率。AVX 指令可以一次处理 4 个 double 类型的数据。（理论上应该具有 4 倍的加速比）；

使用的操作指令有：

_mm256_load_pd()　_mm256_store_pd()

_mm256_add_pd()　_mm256_mul_pd()

使用 AVX 指令_mm256_load_pd 加载 4 个 double 数据，_mm256_store_pd 存储 4 个 double 数据，_mm256_add_pd 和_mm256_mul_pd 进行加法和乘法运算。

具体代码：

```
    // 子字并行优化方法
void avx_dgemm(int n, REAL_T *A, REAL_T *B, REAL_T *C) {
    for (int i = 0; i < n; i += 4)
        for (int j = 0; j < n; ++j) {
            __m256d cij = _mm256_load_pd(C + i + j * n);
            for (int k = 0; k < n; k++) {
                cij = _mm256_add_pd(
                       cij,
                       _mm256_mul_pd(_mm256_load_pd(A + i + k * n),
                                     _mm256_broadcast_sd(B + k + j * n))
                );
            }
            _mm256_store_pd(C + i + j * n, cij);
        }
}
```

(2).指令并行(Unrolled AVX Optimized DGEMM)

通过指令运行的流水线原理，将多发射乱序执行通过循环展开，复制 4 份后，在编译优化时将后三次循环进行指令级并行，从而对于 B[i][j]，可以在四次循环中反复利用，主要使用操作指令：_mm256_broadcast_pd()。

同时，在前定义 Unroll 为循环展开次数，控制指令依次发射执行。（理论上相比 AVX，至少拥有一倍的加速比）

循环展开减少了循环控制的开销，使得 CPU 能够更高效地执行计算指令，提高了算法的整体性能。

具体代码：

```
        // 指令并行的优化方法
void pavx_dgemm(int n, REAL_T *A, REAL_T *B, REAL_T *C) {
    for (int i = 0; i < n; i += 4 * UNROLL)
        for (int j = 0; j < n; ++j) {
            __m256d cij[UNROLL];
            for (int x = 0; x < UNROLL; ++x)
                cij[x] = _mm256_load_pd(C + i + 4 * x + j * n);
            for (int k = 0; k < n; k++) {
                __m256d b = _mm256_broadcast_sd(B + k + j * n);
                for (int x = 0; x < UNROLL; ++x)
                    cij[x] = _mm256_add_pd(
                            cij[x],
                            _mm256_mul_pd(_mm256_load_pd(A + i + 4 * x +
k * n), b));
            }
            for (int x = 0; x < UNROLL; ++x)
                _mm256_store_pd(C + i + x * 4 + j * n, cij[x]);
        }
}
```

(3).分块优化(Blocked AVX Optimized DGEMM)

通过将原始大矩阵分块成小矩阵进行运算，在一个子矩阵（块）被 cache 替换出去之前，最大限度的对其进行数据访问。从而，由于时间局部性，在相邻时间内 cache 内元素被多次利用，减少搜索主存的次数，提高 cache 命中率，实现优化效果。

在 block_gemm 函数中，调用 do_block(int n, int si, int sj, int sk, REAL_T* A, REAL_T* B, REAL_T* C) 处理每个块。在 do_block 函数中，通过 si, sj, sk 参数指定块的起始位置，使用与 Unrolled AVX 相同的指令处理每个块内的计算。

具体代码：

```
        // 分块的优化方法
void do_block(int n, int si, int sj, int sk, REAL_T *A, REAL_T *B,
REAL_T *C) {
    for (int i = si; i < si + BLOCKSIZE; i += UNROLL * 4)
        for (int j = sj; j < sj + BLOCKSIZE; ++j) {
            __m256d c[UNROLL];
            for (int x = 0; x < UNROLL; ++x)
                c[x] = _mm256_load_pd(C + i + 4 * x + j * n);
            for (int k = sk; k < sk + BLOCKSIZE; ++k) {
                __m256d b = _mm256_broadcast_sd(B + k + j * n);
                for (int x = 0; x < UNROLL; ++x)
                    c[x] = _mm256_add_pd(
                            c[x],
                            _mm256_mul_pd(_mm256_load_pd(A + i + 4 * x +
k * n), b));
            }
```

```
            for (int x = 0; x < UNROLL; ++x)
                _mm256_store_pd(C + i + x * 4 + j * n, c[x]);
        }
}

void block_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C) {
    for (int sj = 0; sj < n; sj += BLOCKSIZE)
        for (int si = 0; si < n; si += BLOCKSIZE)
            for (int sk = 0; sk < n; sk += BLOCKSIZE)
                do_block(n, si, sj, sk, A, B, C);
}
```

(4).多处理器优化(OpenMP DGEMM)

综合利用多核处理器进行优化。

由于现代 CPU 基本都是多核的，使用多核技术可以同时运行多次运算，从而达到优化目的。线程数一般默认为 4，也可以自行修改配置，在我的代码中，采用默认的 4 个处理器进行并行。（所以应有 4 倍的加速比）

主要语句：#pragma omp parallel for: 使用 OpenMP 将循环并行化

OpenMP 允许将循环并行化，将计算任务分配到多个处理器核心上同时进行。通过在块矩阵乘法的外层循环中添加 OpenMP 指令，可以让每个处理器核心负责处理不同的块，进一步提高计算效率。

具体代码：

```
    // 多处理器并行的优化方法
void omp_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C) {
#pragma omp parallel for
    for (int sj = 0; sj < n; sj += BLOCKSIZE)
        for (int si = 0; si < n; si += BLOCKSIZE)
            for (int sk = 0; sk < n; sk += BLOCKSIZE)
                do_block(n, si, sj, sk, A, B, C);
}
```

**（do_block）同上面分块算法。

***GPU 优化**

由于电脑未配备 NVIDIA 显卡，不使用 CUDA 进行优化

最终实验结果见 **4.实验结果分析。**

- **Taishan 服务器：**

泰山服务器（Taishan Server）是由华为推出的一系列高性能计算服务器，其基于 ARM 架构，采用华为自主研发的鲲鹏（Kunpeng）处理器，相比传统的 x86 架构能效比高、性能出色，可以完成高性能计算。同时，其具有灵活扩展性，支持灵活的扩展配置，从而在市场上得到了广泛应用，涵盖多个行业。

在本次实验中，需要登陆 Taishan 处理器，完成 4 个维度不同的矩阵乘法测试。由于使用 AVX 库进行子字优化在 Taishan 服务器上的软件包环境不好配置，不进行此层次优化操作。

删减后的代码如下：

```
#include<stdio.h>
#include<stdlib.h>
```

```
#include<time.h>
#include<string.h>
#include<omp.h>
#define REAL_T double
#define UNROLL 4
#define BLOCK_SIZE 32
//计算并输出计算时间和每秒浮点运算次数
void printFlops(int A_height, int B_width, int B_height, clock_t
start, clock_t stop) {
    printf("Time: %ld.%ld s\n", (stop - start) / CLOCKS_PER_SEC,
(stop -
                                                        start) %
CLOCKS_PER_SEC);
    REAL_T flops = (2.0 * A_height * B_width * B_height) / 1E9 /
((stop
                                                          - start) /
(CLOCKS_PER_SEC * 1.0));
    printf("Performance: %f FLOPS\n\n", flops);
}
// 随机生成浮点数构造原始矩阵
void initMatrix(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            A[i + j * n] = rand() / (REAL_T) RAND_MAX;
            B[i + j * n] = rand() / (REAL_T) RAND_MAX;
            C[i + j * n] = 0;
        }
}
//定义算法
void origin_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j){
            REAL_T cij = C[i + j * n];
            for (int k = 0; k < n; k++) {
                cij += A[i + k * n] * B[k + j * n];
            }
            C[i + j * n] = cij;
        }
}
//多处理器并行的优化方法
void block_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
    for (int i = 0; i < n; i += BLOCK_SIZE) {
        for (int j = 0; j < n; j += BLOCK_SIZE) {
            for (int k = 0; k < n; k += BLOCK_SIZE) {
```

```
            for (int ii = i; ii < i + BLOCK_SIZE; ++ii) {
                for (int jj = j; jj < j + BLOCK_SIZE; ++jj) {
                    REAL_T cij = C[ii + jj * n];
                    for (int kk = k; kk < k + BLOCK_SIZE; ++kk) {
                        cij += A[ii + kk * n] * B[kk + jj * n];
                    }
                    C[ii + jj * n] = cij;
                }
            }
        }
    }
}
void omp_gemm(int n, REAL_T* A, REAL_T* B, REAL_T* C) {
#pragma omp parallel for collapse(2) default(none) shared(A, B, C, n)
    for (int j = 0; j < n; j += BLOCK_SIZE) {
        for (int i = 0; i < n; i += BLOCK_SIZE) {
            for (int k = 0; k < n; k += BLOCK_SIZE) {
                for (int ii = i; ii < i + BLOCK_SIZE; ++ii) {
                    for (int jj = j; jj < j + BLOCK_SIZE; ++jj) {
                        REAL_T cij = C[ii + jj * n];
                        for (int kk = k; kk < k + BLOCK_SIZE; ++kk) {
                            cij += A[ii + kk * n] * B[kk + jj * n];
                        }
                        C[ii + jj * n] = cij;
                    }
                }
            }
        }
    }
}
int main() {
    srand((unsigned int)time(NULL)); REAL_T* A, * B, * C;
    clock_t start, stop;
    int n = 1024; // 矩阵规模
    for (int i = 1;i < 5; i++) {
        if (i != 1)
            n += 1024;
        A = (REAL_T*)malloc(n * n * sizeof(REAL_T)); B =
(REAL_T*)malloc(n * n * sizeof(REAL_T)); C = (REAL_T*)malloc(n * n *
sizeof(REAL_T)); //构造原始矩阵
        printf("demension: %d\n\n", n);
        initMatrix(n, A, B, C); // 从原始矩阵拷贝数据 printf("origin
caculation performance:\n"); start = clock();
```

```
        origin_gemm(n, A, B, C);
        stop = clock();
        double origin_time = stop - start;
        printFlops(n, n, n, start, stop);
        initMatrix(n, A, B, C); // 从原始矩阵拷贝数据 printf("block
caculation performance:\n"); start = clock();
        block_gemm(n, A, B, C);
        stop = clock();
        double block_time = stop - start;
        printFlops(n, n, n, start, stop);
        initMatrix(n, A, B, C); // 从原始矩阵拷贝数据 printf("openmp
caculation performance:\n"); start = clock();
        omp_gemm(n, A, B, C);
        stop = clock();
        double omp_time = stop - start;
        printFlops(n, n, n, start, stop);
        printf("block speedup: %f\n", origin_time / block_time);
        printf("omp speedup: %f\n\n", origin_time / block_time);
        printf("------------------------------\n\n");
        free(A);
        free(B);
        free(C);
    } }
```

这段代码实现了矩阵乘法的三种不同方法，并评估其性能：基本矩阵乘法（origin_gemm）、块矩阵乘法（block_gemm）和使用 OpenMP 进行并行优化的块矩阵乘法（omp_gemm）。这些方法用于在不同规模的矩阵上进行测试，并输出计算时间和浮点运算次数（FLOPS）。

在实际连接中，我们使用 putty 进行服务器连接。

```
stu2213924@parallel542-taishan200-1:~$ vim main.c
stu2213924@parallel542-taishan200-1:~$ ls
main  main.c
stu2213924@parallel542-taishan200-1:~$ gcc -o main main.c
stu2213924@parallel542-taishan200-1:~$ ls
main  main.c
stu2213924@parallel542-taishan200-1:~$ ./main
```

先进行 vim 和 gcc 的代码编写和编译 然后运行。

结果见 **4、实验结果分析**

4、 实验结果分析

在第一部分个人 PC 电脑中，最终运行结果如下：

1024 维：

```
dimension: 1024

original calculation performance:
Time:   5.337 s
Performance:    0.402377 GFLOPS

AVX calculation performance:
Time:   1.892 s
Performance:    1.13503 GFLOPS

PAVX calculation performance:
Time:   1.408 s
Performance:    1.5252 GFLOPS

block calculation performance:
Time:   1.277 s
Performance:    1.68166 GFLOPS

OpenMP calculation performance:
Time:   0.337 s
Performance:    6.37236 GFLOPS

AVX speedup: 2.82082
PAVX speedup: 3.79048
block speedup: 4.17933
OpenMP speedup: 15.8368
```

2048 维：

```
dimension: 2048

original calculation performance:
Time:   153.619 s
Performance:    0.111834 GFLOPS

AVX calculation performance:
Time:   58.212 s
Performance:    0.295126 GFLOPS

PAVX calculation performance:
Time:   31.302 s
Performance:    0.548843 GFLOPS

block calculation performance:
Time:   10.68 s
Performance:    1.6086 GFLOPS

OpenMP calculation performance:
Time:   2.88 s
Performance:    5.96523 GFLOPS

AVX speedup: 2.63896
PAVX speedup: 4.90764
block speedup: 14.3838
OpenMP speedup: 53.3399
```

3072 维：

```
dimension: 3072

original calculation performance:
Time:   707.444 s
Performance:    0.0819599 GFLOPS

AVX calculation performance:
Time:   268.184 s
Performance:    0.216203 GFLOPS

PAVX calculation performance:
Time:   189.269 s
Performance:    0.306347 GFLOPS

block calculation performance:
Time:   35.369 s
Performance:    1.63935 GFLOPS

OpenMP calculation performance:
Time:   12.996 s
Performance:    4.46153 GFLOPS

AVX speedup: 2.63791
PAVX speedup: 3.73777
block speedup: 20.0018
OpenMP speedup: 54.4355
```

4096 维：

```
dimension: 4096

original calculation performance:
Time:   1752.16 s
Performance:    0.0784396 GFLOPS

AVX calculation performance:
Time:   757.436 s
Performance:    0.181453 GFLOPS

PAVX calculation performance:
Time:   469.466 s
Performance:    0.292756 GFLOPS

block calculation performance:
Time:   99.689 s
Performance:    1.37868 GFLOPS

OpenMP calculation performance:
Time:   41.118 s
Performance:    3.34255 GFLOPS

AVX speedup: 2.31328
PAVX speedup: 3.73225
block speedup: 17.5763
OpenMP speedup: 42.613
```

指标：（1）Time 运行时间，程序的总运行时间，由运行函数前后 clock()得到。

（2）Performance 每秒浮点数运算次数，通过 printFlops 函数得到。

（3）speedup 加速比，由原始算法运行时间/优化算法运行时间得到。

*（4）运行性能 = 1/运行时间，与运行时间大同小异，因此未输出。

根据对结果的分析，可以得到以下结论：

随着四次算法的逐步优化，运行时间逐渐减少，每秒浮点数运算次数增加，加速比增大。优化是有效且逐步变得更优的。

但，几乎所有优化都比预估的优化速度比更小，可能是实际过程中存在硬件/电脑实际内存分配的误差和随机性误差的原因。

在第二部分 Taishan 服务器中，最终运行结果如下：

1024 维:

```
demension: 1024

origin caculation performance:
Time: 18.126588 s
Performance: 0.118471 FLOPS

block caculation performance:
Time: 7.531830 s
Performance: 0.285121 FLOPS

openmp caculation performance:
Time: 7.550113 s
Performance: 0.284431 FLOPS

block speedup: 2.406665
omp speedup: 2.406665
```

2048 维:

```
demension: 2048

origin caculation performance:
Time: 216.626838 s
Performance: 0.079306 FLOPS

block caculation performance:
Time: 60.282863 s
Performance: 0.284988 FLOPS

openmp caculation performance:
Time: 59.979417 s
Performance: 0.286429 FLOPS

block speedup: 3.593506
omp speedup: 3.593506
```

3072 维:

```
demension: 3072

origin caculation performance:
Time: 781.138535 s
Performance: 0.074228 FLOPS

block caculation performance:
Time: 204.540460 s
Performance: 0.283475 FLOPS

openmp caculation performance:
Time: 202.997686 s
Performance: 0.285629 FLOPS

block speedup: 3.818993
omp speedup: 3.818993
```

4096 维:

```
block caculation performance:
Time: 485.31938 s
Performance: 0.283361 FLOPS

openmp caculation performance:
^[[3~Time: 479.800618 s
Performance: 0.286450 FLOPS

block speedup: 3.947870
omp speedup: 3.947870
```

通过观察在 Taishan 服务器上的运行数据可以发现，虽然分块算法明显比原始算法有优化提升，但是多核算法并没有比分块算法提升多少。同时，相同维度相同算法的运行时间，服务器上的运行时间明显大于个人 PC 电脑上的运行时间。推测理由如下：

(1)多核算法未提升性能：由于之前在 visual studio 就是没有开启 OpenMP 支持所以导致的类似这个数据，可能是服务器默认不支持多核。

也有可能是由于算法在并行过程中运用了深层嵌套，限制了并行化的效果。

同时，负载不平衡，可能导致某些线程计算较快，而其他线程仍在工作，服务器上的硬件资源未被完全调用起来，导致的算法未提升性能。

(2)个人 PC 算法更快：由于服务器是通过 SSH 连接，数据的传输和显示需要通过协议转发，中间过程应该非常耗时，产生延迟。

同时，Taishan 服务器使用了 ARM 架构的 CPU，这种处理器在单核性能上可能不如专为高性能计算优化的桌面 CPU。这可能导致即便在多核利用率高的情况下，单核运算性能仍然低于个人电脑。

而且，服务器通常配置有更高容量且更注重错误纠正的内存，但这些内存在速度和延迟上可能不如专为高速计算设计的桌面内存。服务器的操作系统也更偏向于稳定性和安全性，而不是算法的优化性，从而无法比个人 PC 更优

5、 过程中遇到的问题：

(1).AVX 指令集

在实验过程中，因为 AVX 指令集而耗费了大量时间，由于对于操作系统的不熟悉，在发现自己的 Macbook 无法使用 AVX 指令集前实在有点摸不到头脑，最终发现问题并转战 windows 系统。

同时，由于代码是直接部分给出，对于代码的理解也花费了一定的时间，之前没有接触过 256 位的寄存器，通过 CSDN 和 Google，对于该部分进行了较为系统地学习，方便理解代码。

(2)分块优化的块大小选择

在实现分块矩阵优化时，分块的块大小十分影响优化的最终效果，需要确定合理的分块策略和块大小，因此在实验中进行了多次测试，确定了最终合适的大小和策略。

(3)并行计算，OpenMP 的学习

由于是第一次接触 OpenMP，（之前虽然学习过多线程和多进程，但是是使用 python 的库），对于 C++实现比较陌生，通过网络进行了部分学习，看明白了代码，大概之后也会有用处

(4)Taishan 服务器的连接，putty 使用

第一次实际操作远程连接 SSH 服务器，之前的汇编语言课程是使用华为的线上可视化界面进行服务器的调用，因此对 putty 的使用比较陌生，进行了部分学习，同时在熟悉的交互式界面通过 vi 和 gcc 进行代码的书写和编译，有一定难度。

(5)时间的等待!!!

代码虽然确实很好，但是矩阵维度实在太大，每一次优化前的跑代码都是一次折磨，但同时磨练了坚韧的毅力。

6、 总结感想

在本次实验中，通过对矩阵乘法的优化方法进行了深入研究和实践，不仅熟悉了矩阵乘法的基础实现，还系统学习了多种优化技术，包括 AVX 指令集、指令并行、分块优化和多处理器优化（OpenMP）。

在个人 PC 电脑上的实验结果表明，通过逐步引入 AVX 指令、指令并行、分块优化

和多处理器优化，矩阵乘法的计算性能得到了显著提升。实际测试中，随着算法的优化，运行时间明显减少，每秒浮点运算次数（GFLOPS）增加，加速比也逐步增大。这验证了优化方法的有效性。

通过在个人 PC 和 Taishan 服务器上进行对比实验，发现虽然 Taishan 服务器在某些优化算法上的性能提升显著，但整体性能不及个人 PC。分析认为，可能由于服务器默认不支持多核优化，ARM 架构的单核性能不如桌面 CPU，以及通过 SSH 连接造成的数据传输延迟等原因导致。

在实验中遇到了 Macbook 不支持 AVX 指令集的问题，转而在 Windows 系统上实现并测试了 AVX 优化算法。通过对 AVX 指令集的学习和应用，深入理解了其在并行处理浮点数运算中的优势和具体使用方法。

同时，第一次接触 OpenMP 进行并行优化，通过实践理解了其基本原理和实现方法。在分块矩阵乘法中应用 OpenMP 进行多线程优化，虽然在 Taishan 服务器上的效果不显著，但在个人 PC 上还是看到了性能的提升。

总的来说，本次实验不仅加深了我对矩阵乘法及其优化方法的理解，还提高了我在不同编程环境和硬件平台上进行性能测试和分析的能力。通过解决实际问题，学会了如何灵活应对硬件和软件环境的限制，并在实践中逐步完善自己的代码和算法优化技巧。未来的学习和工作中，这些经验将对我继续探索高性能计算和优化算法提供重要的帮助。