CS205 C/C++ Programming -- Project 3 Improved Matrix Multiplication

Name: 陈长信(Chen Changxin)

SID: 12210731

Part1--问题分析

本题旨在通过各种方式实现矩阵乘法速度的优化。

基于上一个项目的基础，本项目弱化了对随机数范围、遍历顺序等影响因素的探索，重点在于对计算速度的优化。

通过比较 `matmul_plain()` ，以及通过各种途径优化之后的 `matmul_improved()` 函数来比较优化之后不同大小矩阵乘法的速度。

关于 `matmul_plain` 的实现，首先完成对矩阵数据的储存，使用结构体存储 `rows,cols` 以及 `float *data` 变量；

然后使用函数 `struct Matrix *allocate_Matrix(size_t rows, size_t cols) 以及 void deallocateMatrix(struct Matrix *mat)` 完成对矩阵内存的分配以及释放；

用函数 `void generateRandomMatrix(struct Matrix *mat)` 完成对矩阵随机数的赋值；

最后在核心计算部分直接使用 `result->data[i * result->cols + j] += A->data[i * A->cols + k] * B->data[k * B->cols + j]` 来进行矩阵乘法。

优化部分：

1. 使用SIMD方式进行优化。
2. 基于SIMD，加入并行处理OMP进行优化。
3. 在SIMD和OMP的优化基础上，再充分利用并行计算的优势，利用矩阵分割，运用分块矩阵的思想进行优化。
4. 将上述优化策略与OpenBLAS库的运行速度进行分析比较。

由于设备以及优化的限制，我没有进行 `size = 64000` 的测试，最后最大测试 `size` 为 `size = 19200` 。

Part2--基础实验

1. 基本矩阵构造设计思路

1. 使用分配内存的函数，给乘法矩阵以及结果矩阵分配内存空间。

```
//给矩阵分配空间
struct Matrix *allocate_Matrix(size_t rows, size_t cols)
{
    struct Matrix *mat = (struct Matrix *)malloc(sizeof(struct Matrix));
    if (mat == NULL)
    {
        printf("内存分配失败\n");
        return NULL;
    }
    mat->rows = rows;
    mat->cols = cols;
    mat->data = (float *)malloc(rows * cols * sizeof(float));
    if (mat->data == NULL)
    {
        printf("内存分配失败\n");
        free(mat);
        return NULL;
    }
    return mat;
}


int main()
{
    /// ...
    struct Matrix *A = allocate_Matrix(testcases[testcase], testcases[testcase]);
    struct Matrix *B = allocate_Matrix(testcases[testcase], testcases[testcase]);
    struct Matrix *result = allocate_Matrix(testcases[testcase],
testcases[testcase]);
    /// ...
}
```

2. 根据项目经验，数据的规模对矩阵乘法速度较小，故这里弱化对矩阵乘法数据范围的探究，仅给乘法矩阵赋 `0-1` 之间的浮点数随机值。

```
void generateRandomMatrix(struct Matrix *mat)
{
    for (size_t i = 0; i < mat->rows * mat->cols; ++i)
    {
        mat->data[i] = (float)rand() / RAND_MAX;
    }
}

int main()
{
    /// ...
```

```
    generateRandomMatrix(A);
    generateRandomMatrix(B);
    /// ...
}
```

3. 实施矩阵乘法，并且对矩阵乘法的可行性、正确性等进行判断。

```
// Function to perform matrix multiplication
void matmul_plain(const struct Matrix *A, const struct Matrix *B, const struct
Matrix *result)
{
    if (A == NULL || B == NULL || result == NULL)
    {
        printf("输入矩阵不能为空\n");
        return;
    }
    if (A->cols != B->rows)
    {
        printf("矩阵尺寸不兼容\n");
        return;
    }
    if (A->rows != result->rows || B->cols != result->cols)
    {
        printf("结果矩阵尺寸不正确\n");
        return;
    }
    //实施基础矩阵乘法
    size_t i, j, k;
    for (i = 0; i < A->rows; ++i)
    {
        for (k = 0; k < A->cols; ++k)
        {
            for (j = 0; j < B->cols; ++j)
            {
                result->data[i * result->cols + j] += A->data[i * A->cols + k] *
B->data[k * B->cols + j];
            }
        }
    }
}
```

4. 进行时间的测量，并通过.txt进行输出

```
clock_t start = clock();
matmul_plain(A, B, result);
clock_t end = clock();
double elapsed_secs = (double)(end - start) / CLOCKS_PER_SEC;
```

5. 最后对内存空间进行释放，以免内存泄漏

```
// Deallocate memory for a matrix
void deallocateMatrix(struct Matrix *mat)
{
    free(mat->data);
    free(mat);
}

int main()
{
    /// ...
    deallocateMatrix(result);
    deallocateMatrix(A);
    deallocateMatrix(B);
    /// ...
}
```

2. 时间测试

仅进行普通编译的测试即 `gcc Matrix.c` 进行编译，测量的结果如下图所示

```
Time used(size = 16): 0.000017s
Time used(size = 128): 0.008217s
Time used(size = 1000): 4.046370s
Time used(size = 2000): 32.471377s
```

由于当 `size` 达到测试集中8k的时候运行时间过长，没有进行过多的时间测量，直接进入了优化阶段

Part3--优化探索

1. 使用SIMD进行优化

SIMD可以优化矩阵乘法的速度，主要因为它允许同时处理多个数据元素，从而提高了计算的并行性。在矩阵乘法中，有大量的乘法和加法操作需要执行，而这些操作通常都是独立的。通过使用 SIMD 指令，可以一次性处理多个数据元素，从而在相同的时钟周期内完成更多的计算，最后使用 `gcc -o Matrix Matrix.c -DWITH_AVX2 -mavx` 进行编译

256位优化

具体来说，当使用__m256表示使用256位的向量，可以容纳8个单精度浮点数，比如 `k = 0; k < A->cols / 8; ++k` 的内层循环，就是直接把8位并行计算最后 `result->data[i * result->cols + j] += Σsum_vector[x]` 这样的存储数据。

```
#ifdef WITH_AVX2
#include <immintrin.h>
#endif
```

```
size_t i, j, k;

for (i = 0; i < A->rows; ++i)
{
    for (j = 0; j < B->cols; ++j)
    {
        __m256 sum_vec = _mm256_setzero_ps();
        for (k = 0; k < A->cols / 8; ++k)
        {
            //加载8个一组中每个元素的值
            __m256 a_vec = _mm256_loadu_ps(&A->data[i * A->cols + k]);
            __m256 b_vec = _mm256_loadu_ps(&B->data[k * B->cols + j]);
            // 使用 SIMD 指令进行乘法运算
            __m256 mul_vec = _mm256_mul_ps(a_vec, b_vec);
            // 将乘法结果累加到 sum_vec 中
            sum_vec = _mm256_add_ps(sum_vec, mul_vec);
         }
       // 将累加结果存储到 result 矩阵中
      result->data[i * result->cols + j] += sum_vec[0] + sum_vec[1] + sum_vec[2] +
sum_vec[3] + sum_vec[4] + sum_vec[5] + sum_vec[6] + sum_vec[7];
    }
}
```

我进行了SIMD进行了矩阵乘法优化，分别对矩阵size在16,128,1000,2000,8000进行了测速，时间如下图所示，我们发现时间明显有了提升。

```
Time used after improved by SIMD(size = 16): 0.000005s
Time used after improved by SIMD(size = 128): 0.001589s
Time used after improved by SIMD(size = 1000): 0.757465s
Time used after improved by SIMD(size = 2000): 6.200813s
Time used after improved by SIMD(size = 8000): 537.085771s
```

128位优化

同样我也进行了使用128位的SIMD优化，并且进行了运行时间检测

```
size_t i, j, k;

for (i = 0; i < A->rows; ++i)
{
    for (j = 0; j < B->cols; ++j)
    {
        __m128 sum_vec = _mm_setzero_ps(); // 使用 SSE 寄存器进行累加
        for (k = 0; k < A->cols / 4; ++k)
        {
            __m128 a_vec = _mm_loadu_ps(&A->data[i * A->cols + k]);
            // 加载 B 矩阵的 4 个元素到 SSE 寄存器
            __m128 b_vec = _mm_loadu_ps(&B->data[k * B->cols + j]);
            // 使用 SIMD 指令进行乘法运算
            __m128 mul_vec = _mm_mul_ps(a_vec, b_vec);
```

```
            // 将乘法结果累加到 sum_vec 中
            sum_vec = _mm_add_ps(sum_vec, mul_vec);
        }
        result->data[i * result->cols + j] += sum_vec[0] + sum_vec[1] + sum_vec[2] +
sum_vec[3];
    }
}
```

同样，我对矩阵size在16,128,1000,2000,8000进行了测速，测试结果如下

```
//128位
Time used after improved by SIMD(size = 16): 0.000011s
Time used after improved by SIMD(size = 128): 0.002600s
Time used after improved by SIMD(size = 1000): 1.365752s
Time used after improved by SIMD(size = 2000): 12.137461s
Time used after improved by SIMD(size = 8000): 1041.429958s
```

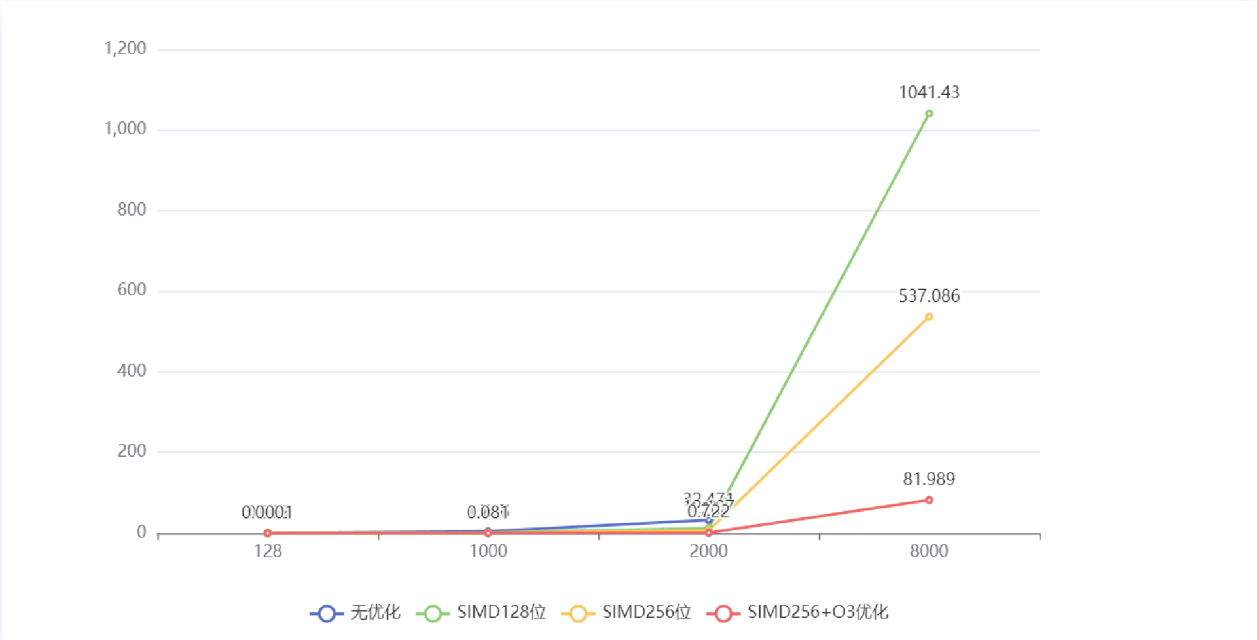
我们发现当使用__m128时候只能并行处理8个单精度浮点数，也会有所优化，但是优化效果不如256位明显。

-O3优化

另外，我在256位计算中加入-O3优化并进行测试

```
//O3
Time used after improved by SIMD(size = 16): 0.000002s
Time used after improved by SIMD(size = 128): 0.000141s
Time used after improved by SIMD(size = 1000): 0.081130s
Time used after improved by SIMD(size = 2000): 0.722139s
Time used after improved by SIMD(size = 8000): 81.989225s
```

然后对于四种不同的乘法思路，我作图对运行进行了对比。



2.使用SIMD+OMP进行优化

基于上述对SIMD的应用，我对此外加了对OMP的应用，即提高并行化效率，添加了语句 `#pragma omp parallel for private(i, j, k) shared(A, B, result) num_thread(3)` 以及 `#pragma omp for schedule(static) private(j, k)` 具体代码如下

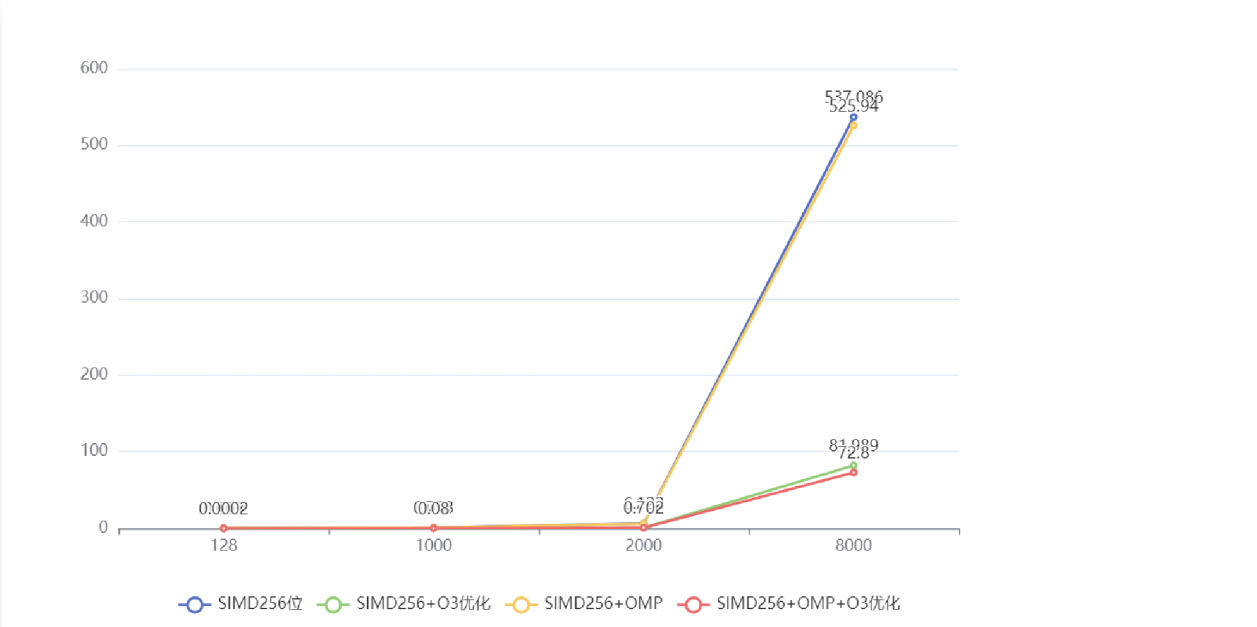
```
//编译时候开启WITH_AVX2
#ifdef WITH_AVX2
#include <omp.h>
#endif

#pragma omp parallel for private(i, j, k) shared(A, B, result) num_thread(3)
#pragma omp for schedule(static) private(j, k)
//以下代码同SIMD代码
// ...
```

最后同时对相同的数据集进行了测试（也对O3优化之后进行了测试），得到如下的运行结果，并做出图表。

```
Time used after improved by SIMD + OMP(size = 16): 0.000007s
Time used after improved by SIMD + OMP(size = 128): 0.001711s
Time used after improved by SIMD + OMP(size = 1000): 0.773088s
Time used after improved by SIMD + OMP(size = 2000): 6.122551s
Time used after improved by SIMD + OMP(size = 8000): 525.939635s

//SIMD+OMP开O3
Time used after improved by SIMD + OMP(size = 16): 0.000001s
Time used after improved by SIMD + OMP(size = 128): 0.000219s
Time used after improved by SIMD + OMP(size = 1000): 0.080449s
Time used after improved by SIMD + OMP(size = 2000): 0.701568s
Time used after improved by SIMD + OMP(size = 8000): 72.799599s
```



通过对比上述折线图发现，在OMP优化加入之后，速度有了一定的提升，但提升速度并不大，在开了O3之后，`size = 8000` 的时候达到了目前最快的72s的运行速度。

3.分块矩阵思路

基于上述思路，我意识到可以尝试利用多的循环，并且尝试利用多核并行运行的优势对速度进行优化。所以引入分块矩阵乘法的思路，是它能够更有效地利用计算资源和内存层次结构，一是分块矩阵乘法通过在内存中加载分块数据，并在 CPU 缓存中进行计算，可以减少内存访问的次数，从而提高计算效率；二是分块矩阵乘法可以很容易地进行并行化。通过将矩阵分成多个块，可以在每个块上独立地进行计算，从而充分利用多核处理器的并行计算能力。

```
size_t i, j, k;
int block_size = 16;
#pragma omp parallel for collapse(4) num_threads(4)
    for (i = 0; i < A->rows; i += block_size)
    {
        for (k = 0; k < A->cols; k += block_size)
        {
            for (j = 0; j < B->cols; j += block_size)
            {
                for (size_t ii = i; ii < i + block_size && ii < A->rows; ++ii)
                {
                    for (size_t jj = j; jj < j + block_size && jj < B->cols; ++jj)
                    {
                        __m256 sum_vec = _mm256_setzero_ps();
                        for (size_t kk = k; kk < k + block_size && kk < A->cols; kk
+= 8)
                        {
                            __m256 a_vec = _mm256_loadu_ps(&A->data[ii * A->cols +
kk]);
                            // 加载 B 矩阵的 8 个元素到 SSE 寄存器
                            __m256 b_vec = _mm256_loadu_ps(&B->data[kk * B->cols +
jj]);
                            // 使用 SIMD 指令进行乘法运算
                            __m256 mul_vec = _mm256_mul_ps(a_vec, b_vec);
                            // 将乘法结果累加到 sum_vec 中
                            sum_vec = _mm256_add_ps(sum_vec, mul_vec);
                        }
                        result->data[ii * result->cols + jj] += sum_vec[0] +
sum_vec[1] + sum_vec[2] + sum_vec[3] + sum_vec[4] + sum_vec[5] + sum_vec[6] +
sum_vec[7];
                    }
                }
            }
        }
    }
```

由于此处探究与 `block_size` 相关的参数，为了忽略冗余部分的影响以及保证 `block_size>total_size` ，故本部分仅对 `size = 128,1280,6400,9600` 进行测试

这里首先使用的是 `block_size = 16` 的情况下，即每size = 16一个分块进行处理，得到以下数据

```
Time used by block(size = 16): 0.000002s
Time used by block(size = 128): 0.000360s
Time used by block(size = 1000): 0.175623s
Time used by block(size = 2000): 1.366710s
Time used by block(size = 8000): 88.277172s
```

在 `block_size = 32` 的情况下，得到以下数据

```
Time used by block(size = 128): 0.000391s
Time used by block(size = 1280): 0.268924s
Time used by block(size = 6400): 36.188913s
Time used by block(size = 9600): 121.185348s
```

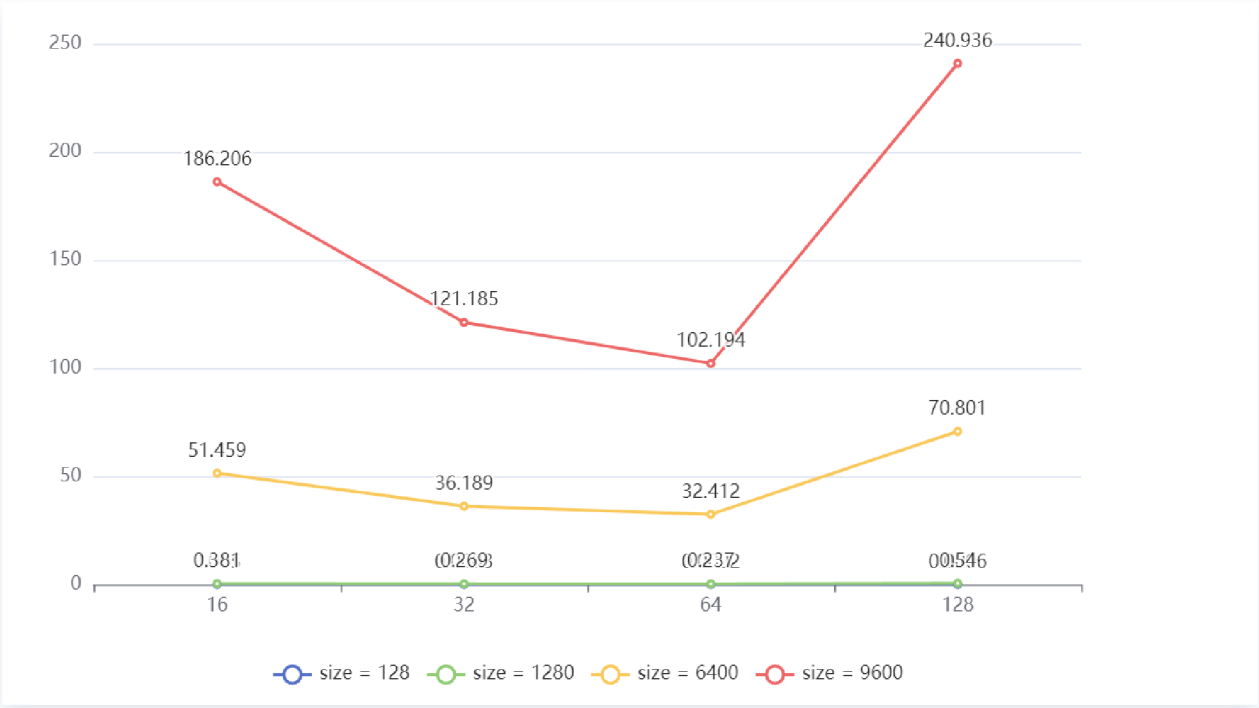
接着，我又对 `block_size = 64,128` 进行了测试，测试结果数据如下图所示

```
block_size = 64
Time used by block(size = 128): 0.000237s
Time used by block(size = 1280): 0.236773s
Time used by block(size = 6400): 32.411744s
Time used by block(size = 9600): 102.194279s

block_size = 128
Time used by block(size = 128): 0.000697s
Time used by block(size = 1280): 0.540089s
Time used by block(size = 6400): 70.800430s
Time used by block(size = 9600): 240.936072s
```

做出表格如下表所示

block_size	size = 128(s)	size = 1280(s)	size = 6400(s)	size = 9600(s)
16	0.005	0.381	51.459	186.206
32	0.0003	0.269	36.189	121.185
64	0.0002	0.237	32.412	102.194
128	0.0006	0.540	70.801	240.936



经过对上述折线图数据的比较，我发现在 `block_size = 64` 的情况下速度明显较快，达到了最高的效率，为什么会这样呢?

我认为是因为，现理器通常具有多级缓存，每个缓存级别都有不同的大小和缓存行大小。如果 `block_size` 太小，无法充分利用缓存行，会导致缓存未命中增加。而 `block_size` 太大时，可能会导致缓存冲突，互相干扰，也会影响效率。所以在一个适中的大小，比如在 `size = 64` 时候能达到一个最高的效率。

4.与OpenBLAS运行速度对比

在配置完 `OpenBLAS` 库之后，在同样的数据生成环境下，使用函数 `cblas_segmm` 对运算速度进行了测试

```
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                    testcases[testcase], testcases[testcase], testcases[testcase],
1.0, A->data, testcases[testcase], B->data, testcases[testcase], 0.0, result->data,
testcases[testcase]);
```

命令行 `gcc -o Matrix Matrix.c -lopenblas -lpthread` 运行结果如下图

```
Time used by openblas(size = 16): 0.000705s
Time used by openblas(size = 128): 0.108166s
Time used by openblas(size = 1000): 0.238416s
Time used by openblas(size = 1280): 0.356676s
Time used by openblas(size = 2000): 0.900937s
Time used by openblas(size = 6400): 19.496987s
Time used by openblas(size = 8000): 32.886871s
Time used by openblas(size = 9600): 61.742155s
```

为了方便对比，以及得到更广泛的测试，我同时将上述方法添加数据集之后进行了重新测试。

```
Time used by openblas(size = 12800): 118.193457s
Time used by block(size = 12800): 245.984860s
Time used by openblas(size = 19200): 340.330374s
Time used by block(size = 19200): 786.198443s
```

我将各个优化方法后的时间整理至表格如图所示（时间单位s）

矩阵size/运行时间(s)	SIMD+OMP优化	SIMD+OMP+分块(block_size = 64)	OpenBLAS
16	/	/	0.0007
128	0.0001	0.0002	0.108
1000	0.085	/	0.238
1280	0.303	0.237	0.357
2000	0.735	/	0.901
6400	62.067	32.412	19.497
8000	81.812	/	32.887
9600	182.291	102.194	61.742
12800	/	245.985	118.193
19200	/	786.198	340.330

我们可以发现OpenBLAS速度性能明显较好，尤其是在大矩阵计算时优势明显，为什么？

1. **优化的底层实现**：OpenBLAS针对多种硬件架构进行了高度优化的实现。
2. **并行化运算**：OpenBLAS中GEMM便是使用SIMD进行指令向量化和多核心并行。在我的实现方法上，在SIMD的向量化过程以及并行方式依然有很多优化空间。
3. **内存访问优化**：OpenBLAS采用了一些内存访问优化技术，分块的目的实际上就是优化访存，通过分块之后让访存都集中在一定区域，能够提高了数据局部性，从而提高cache利用率，性能就会更好。实际上我在矩阵分块的时候对分块的 `block_size` 实验较少，以及对于不同的处理器CPU运行环境应该有对应的分块大小，这样能更加高效提高cache利用率。

Part4--项目总结以及主要困难解决方案

1. 矩阵乘法计算量大，在无法在大O表示法上通过肉眼可见的复杂度降低，尝试通过 `SIMD` 、 `OMP` 以及矩阵分块等思路进行优化，主要利用多核计算的并行计算以及内存访问的优化思路。

Part5--补充说明

1. 本项目部分代码框架来自ChatGPT，核心代码均为本人撰写。
2. 本项目基于上个项目 `CS205 C/C++ Programming -- Project 2 Simple Matrix Multiplication` ，所以忽略了例如遍历顺序等影响以及优化策略，本项目优化部分均采用较为优化的代码结构。