

CS205 C/C++ Program Design - Project 1

姓名：庾立轩

学号：3122001509

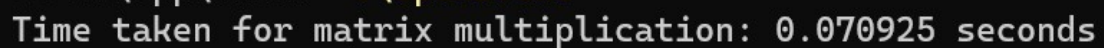
本次 project1 一共上交 5 个代码文件:

1. main.cpp ——>主程序
2. matrix.cpp——>C++语言实现的矩阵类头文件
3. matrix.hpp——>C++语言实现的矩阵类头文件
4. openblas_test.exe——>openblas 运行时间计算程序
5. openblas_test.cpp——>openb.exe 源文件

【最初版本运行时间】 两个 2048×2048 的矩阵相乘用时:80000~90000ms

【最终版本运行时间】 两个 2048×2048 的矩阵相乘用时:30~40ms

【对比】使用 Openblas (网上找的函数) 两个 2048×2048 的矩阵相乘 70.925ms




```
Time taken for matrix multiplication: 0.070925 seconds
```

【要求实现】

1. 矩阵类的定义与设置
2. 大矩阵乘法的实现
3. 矩阵相乘行列不匹配的检测

【时间测量方法】

导入了<chrono> (课程上学的), 用于计算矩阵相乘时间



```
#define TIME_START start=std::chrono::steady_clock::now();
#define TIME_END(NAME) end=std::chrono::steady_clock::now(); \
    duration=std::chrono::duration_cast<std::chrono::milliseconds>(end-start).count(); \
    std::cout<<(NAME)<<"="<<duration<<"ms"<<std::endl;
```

```
auto start = std::chrono::steady_clock::now();
auto end = std::chrono::steady_clock::now();
auto duration = 0L;
```

【效率提升方法】

0.最初版本：for 循环的矩阵相乘

```
Matrix result(rows, mat.cols);

for (int i = 0; i < rows; ++i)
{
    for (int j = 0; j < mat.cols; ++j)
    {
        float sum = 0.0;
        for (int k = 0; k < cols; ++k)
        {
            sum += data[i * cols + k] * mat.data[k * mat.cols + j];
        }
        result.data[i * mat.cols + j] = sum;
    }
}

return result;
```

将【2048*2048】的矩阵相乘，用时 82069ms

```
setting A_rows:2048
setting A_cols:2048
setting B_rows:2048
setting B_cols:2048
A*B=82069ms
```

1. 将编译模式由 Debug 转为 Release

将缓存文件删掉，将编译模式转为 Release.

```

PS D:\YUN\build> cmake -G "MinGW Makefiles" ..\project4\
-- The C compiler identification is GNU 8.1.0
-- The CXX compiler identification is GNU 8.1.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: D:/Down/mingw64/bin/gcc.exe - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: D:/Down/mingw64/bin/c++.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (0.8s)
-- Generating done (0.0s)
-- Build files have been written to: D:/YUN/build
PS D:\YUN\build> cmake -DCMAKE_BUILD_TYPE=Release ..\project4\
-- Configuring done (0.0s)
-- Generating done (0.0s)
-- Build files have been written to: D:/YUN/build

```

重新编译后运行程序(同样是【2048*2048】矩阵相乘), 用时 37890ms

```

PS D:\YUN\build> .\project1.exe
setting A_rows:2048
setting A_cols:2048
setting B_rows:2048
setting B_cols:2048
A*B=37890ms

```

2. 使用 SIMD 指令集加速计算, 将 8 个元素连在一起计算。

```

Matrix SIMD(Matrix mat1, Matrix mat2)
{
    float *mat1_data = mat1.getdata();
    float *mat2_data = mat2.getdata();
    __m256 mat1_num, mat2_num, temp;
    __m256 sum = _mm256_setzero_ps();

    Matrix output(mat1.getrows(), mat2.getcols());
    for (size_t i = 0; i < mat1.getrows(); i++)
    {
        for (size_t j = 0; j < mat2.getcols(); j += 8)
        {
            sum = _mm256_setzero_ps();
            for (size_t k = 0; k < mat1.getcols(); k += 8)
            {
                mat1_num = _mm256_load_ps(&mat1_data[i * mat1.getcols() + k]);
                mat2_num = _mm256_load_ps(&mat2_data[k * mat2.getcols() + j]);
                temp = _mm256_mul_ps(mat1_num, mat2_num);
                sum = _mm256_add_ps(sum, temp);
            }
            _mm256_store_ps(&output.getdata()[i * output.getcols() + j], sum);
        }
    }
    return output;
}

```

再次编译运行，用时 515ms

```

PS D:\YUN\build> .\project1.exe
setting A_rows:2048
setting A_cols:2048
setting B_rows:2048
setting B_cols:2048
A*B_SIMD=515ms

```

3. SIMD 细节优化

将矩阵分为小块，在矩阵乘法的嵌套循环中，k 和 l 用来控制 A 和 B 在同一块中的起始位置，m 和 n 则用来循环处理同一块中的所有元素。这样可以避免跨越不同块的行和列，从而充分利用 CPU 的缓存，提高计算效率。代码如下：

```

Matrix SIMD(Matrix mat1, Matrix mat2)
{
    float *mat1_data = mat1.getdata();
    float *mat2_data = mat2.getdata();
    __m256 mat1_num, mat2_num, temp;
    __m256 sum = _mm256_setzero_ps();

    Matrix output(mat1.getrows(), mat2.getcols());

    for (size_t i = 0; i < mat1.getrows(); i++)
    {
        for (size_t j = 0; j < mat2.getcols(); j += 16)
        {
            sum = _mm256_setzero_ps();
            for (size_t k = 0; k < mat1.getcols(); k += 16)
            {
                for (size_t l = 0; l < 16; l += 8)
                {
                    mat1_num = _mm256_load_ps(&mat1_data[i * mat1.getcols() + k + l]);
                    for (size_t m = 0; m < 16; m += 8)
                    {
                        mat2_num = _mm256_load_ps(&mat2_data[(k + l) * mat2.getcols() + j + m]);
                        temp = _mm256_mul_ps(mat1_num, mat2_num);
                        sum = _mm256_add_ps(sum, temp);
                    }
                }
            }
            _mm256_store_ps(&output.getdata()[i * output.getcols() + j], sum);
        }
    }
    return output;
}

```

分别试过 16 大小的、32 大小的、64 大小的，对比出分为 16 加速效果最好，
用时 310ms

```

PS D:\YUN\build> .\project1.exe
setting A_rows:2048
setting A_cols:2048
setting B_rows:2048
setting B_cols:2048
A*B_SIMD_impr=310ms

```

4. 使用 OpenMP

由于我们程序中有多个嵌套循环，直接使用 OpenMP 可能会导致资源的争用从而
而影响效率，以下为直接使用带来的效果：


```
PS D:\YUN\build> .\project1.exe
setting A_rows:2048
setting A_cols:2048
setting B_rows:2048
setting B_cols:2048
A*B_SIMD_impr=3700ms
```

为了避免线程冲突，我们这里将每个变量都设置为私有，避免线程冲突。

```
#pragma omp parallel for schedule(auto) private(mat1_num, mat2_num, temp, sum)
```

结果用时 36ms

```
PS D:\YUN\build> .\project1.exe
setting A_rows:2048
setting A_cols:2048
setting B_rows:2048
setting B_cols:2048
A*B_SIMD_OpenMp=36ms
```

5. O3 优化

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3" CACHE STRING "C++ flags" FORCE)
```

没什么效果，用时 46ms

```
PS D:\YUN\build> .\project1.exe
setting A_rows:2048
setting A_cols:2048
setting B_rows:2048
setting B_cols:2048
A*B_SIMD_OpenMp=46ms
```

【总结】

本次 project 中让我对优化程序有了更多的了解，包括：

1. SIMD 的使用
2. SIMD 算法的优化
3. OpenMP 的使用及优化
4. CMake 更熟悉的使用，更多的了解
5. 对 OpenBLAS 的了解

【感想】

在本次 project 的完成里，我明白了修改程序的时候一定要留心，或者备份。原本打算进一步实现更多功能，但是不知道改了些什么乘出来的结果原本能打印的莫名其妙无法打印出来。本次 project 有很多不足的地方，换句话说只是一次优化程序的训练，可以说是完全没有别的功能，而且因为算法问题只能算大小在 512 以上的矩阵相乘（而且还要是 4 的倍数），输入的矩阵太大如 16384×16384 会很慢（cpu 使用率拉满），有时候莫名其妙（可能是内存原因）会算不出来，改进空间非常大，我需要更了解计算机及其内存的运作方式。