组成原理矩阵乘法作业1实验报告

学生姓名：刘修铭　学号：2112492　指导老师：董前琨　班级：李涛老师

一、实验要求

1. 实现矩阵乘法；
2. 完成矩阵乘法的优化，并在1024~4096规模的矩阵上进行测试；
3. 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；
4. 总结优化过程中遇到的问题和解决方式。

二、 矩阵乘法的实现及优化

```cpp
#include <bits/stdc++.h>
#include <chrono>
#include <thread>

using namespace std;
using namespace std::chrono;

// 矩阵乘法（按照定义计算）
vector<vector<int>> matrixMultiplication(const vector<vector<int>>& matrix1, const vector<vector<int>>&
matrix2) {
    int rows1 = matrix1.size();
    int cols1 = matrix1[0].size();
    int rows2 = matrix2.size();
    int cols2 = matrix2[0].size();

    if (cols1 != rows2) {
        cout << "Error: The number of columns in matrix1 should be equal to the number of rows in
matrix2." << endl;
        return {};
    }

    vector<vector<int>> result(rows1, vector<int>(cols2, 0));

    for (int i = 0; i < rows1; i++) {
        for (int j = 0; j < cols2; j++) {
            for (int k = 0; k < cols1; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }

    return result;
}

// 优化的矩阵乘法（分块矩阵乘法）
vector<vector<int>> optimizedMatrixMultiplication(const vector<vector<int>>& matrix1, const
vector<vector<int>>& matrix2) {
    int rows1 = matrix1.size();
    int cols1 = matrix1[0].size();
    int rows2 = matrix2.size();
    int cols2 = matrix2[0].size();

    if (cols1 != rows2) {
        cout << "Error: The number of columns in matrix1 should be equal to the number of rows in
matrix2." << endl;
        return {};
    }
```

```cpp

    vector<vector<int>> result(rows1, vector<int>(cols2, 0));

    // 定义块大小
    const int blockSize = 16;

    for (int i = 0; i < rows1; i += blockSize) {
        for (int j = 0; j < cols2; j += blockSize) {
            for (int k = 0; k < cols1; k += blockSize) {
                // 对子块进行乘法计算
                for (int ii = i; ii < min(i + blockSize, rows1); ii++) {
                    for (int jj = j; jj < min(j + blockSize, cols2); jj++) {
                        for (int kk = k; kk < min(k + blockSize, cols1); kk++) {
                            result[ii][jj] += matrix1[ii][kk] * matrix2[kk][jj];
                        }
                    }
                }
            }
        }
    }

    return result;
}

// 随机生成矩阵
vector<vector<int>> generateMatrix(int rows, int cols) {
    vector<vector<int>> matrix(rows, vector<int>(cols, 0));

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = rand() % 100;  // 生成0到99之间的随机整数
        }
    }

    return matrix;
}

// 打印矩阵
void printMatrix(const vector<vector<int>>& matrix) {
    int rows = matrix.size();
    int cols = matrix[0].size();

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}

// 并行计算矩阵乘法
vector<vector<int>> parallelMatrixMultiplication(const vector<vector<int>>& matrix1, const
vector<vector<int>>& matrix2) {
    int rows1 = matrix1.size();
    int cols1 = matrix1[0].size();
    int rows2 = matrix2.size();
    int cols2 = matrix2[0].size();

    if (cols1 != rows2) {
        cout << "Error: The number of columns in matrix1 should be equal to the number of rows in
matrix2." << endl;
        return {};
    }

    vector<vector<int>> result(rows1, vector<int>(cols2, 0));

    const int numThreads = thread::hardware_concurrency();
    vector<thread> threads;
```

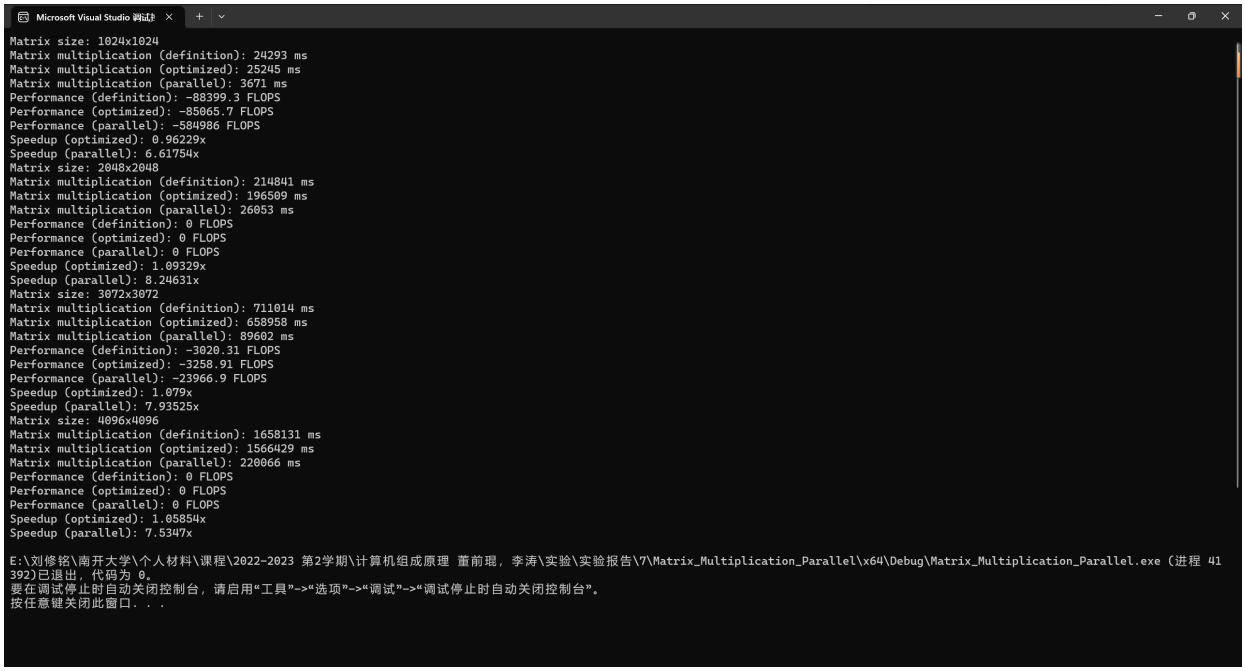
```
110
111     // 创建线程
112     for (int t = 0; t < numThreads; t++) {
113         threads.emplace_back([&, t]() {
114             for (int i = t; i < rows1; i += numThreads) {
115                 for (int j = 0; j < cols2; j++) {
116                     for (int k = 0; k < cols1; k++) {
117                         result[i][j] += matrix1[i][k] * matrix2[k][j];
118                     }
119                 }
120             }
121             });
122     }
123
124     // 等待线程完成
125     for (auto& thread : threads) {
126         thread.join();
127     }
128
129     return result;
130 }
131
132 int main() {
133     const int minSize = 1024;
134     const int maxSize = 4096;
135     const int step = 1024;
136
137     for (int size = minSize; size <= maxSize; size += step) {
138         cout << "Matrix size: " << size << "x" << size << endl;
139
140         vector<vector<int>> matrix1 = generateMatrix(size, size);
141         vector<vector<int>> matrix2 = generateMatrix(size, size);
142
143         // 按照定义计算的矩阵乘法
144         auto startTime = high_resolution_clock::now();
145         vector<vector<int>> result1 = matrixMultiplication(matrix1, matrix2);
146         auto endTime = high_resolution_clock::now();
147         auto duration1 = duration_cast<milliseconds>(endTime - startTime).count();
148         cout << "Matrix multiplication (definition): " << duration1 << " ms" << endl;
149
150         // 优化的矩阵乘法
151         startTime = high_resolution_clock::now();
152         vector<vector<int>> result2 = optimizedMatrixMultiplication(matrix1, matrix2);
153         endTime = high_resolution_clock::now();
154         auto duration2 = duration_cast<milliseconds>(endTime - startTime).count();
155         cout << "Matrix multiplication (optimized): " << duration2 << " ms" << endl;
156
157         // 并行计算矩阵乘法
158         startTime = high_resolution_clock::now();
159         vector<vector<int>> result3 = parallelMatrixMultiplication(matrix1, matrix2);
160         endTime = high_resolution_clock::now();
161         auto duration3 = duration_cast<milliseconds>(endTime - startTime).count();
162         cout << "Matrix multiplication (parallel): " << duration3 << " ms" << endl;
163
164         // 计算运行性能和加速比
165         double performance1 = static_cast<double>(size * size * size * 2) / duration1;
166         double performance2 = static_cast<double>(size * size * size * 2) / duration2;
167         double performance3 = static_cast<double>(size * size * size * 2) / duration3;
168         double speedup2 = static_cast<double>(duration1) / duration2;
169         double speedup3 = static_cast<double>(duration1) / duration3;
170
171         cout << "Performance (definition): " << performance1 << " FLOPS" << endl;
172         cout << "Performance (optimized): " << performance2 << " FLOPS" << endl;
173         cout << "Performance (parallel): " << performance3 << " FLOPS" << endl;
174         cout << "Speedup (optimized): " << speedup2 << "x" << endl;
175         cout << "Speedup (parallel): " << speedup3 << "x" << endl;
176
177         // 检查结果是否一致
```

```
        if (result1 != result2 || result1 != result3) {
            cout << "Error: Matrix multiplication results are different!" << endl;
        }
    }

    return 0;
}
```

本次代码采用运用定义计算矩阵乘法、分块计算矩阵乘法、并行计算矩阵乘法三种方式。

三、优化对比



由运行截图可知，按照定义计算耗时最长，运行性能最差。采用分块计算的优化算法对矩阵乘法有一定加速，但优化程度有限，1024规模时甚至不如定义计算。采用并行计算优化效果最好，加速比接近8。

1. 计算耗时：
    - 按照定义计算的矩阵乘法耗时最长，因为它需要执行三重循环来逐个元素进行计算。
    - 优化的矩阵乘法采用分块矩阵乘法的方法，在矩阵乘法计算中减少了不必要的访存操作，从而提高了计算效率，耗时相对较短。
    - 并行计算矩阵乘法利用多线程进行计算，可以同时进行多个乘法运算，因此具有更快的计算速度，耗时最短。
2. 运行性能：
    - 按照定义计算的矩阵乘法的性能较低，因为它使用了三重循环的嵌套，导致计算复杂度较高。
    - 优化的矩阵乘法通过采用分块矩阵乘法的优化方法，减少了不必要的访存操作，提高了运行性能。
    - 并行计算矩阵乘法利用多线程实现并行计算，充分利用多核处理器的计算能力，因此具有更好的运行性能。
3. 加速比：
    - 优化的矩阵乘法和并行计算矩阵乘法相对于按照定义计算的矩阵乘法都能够取得较好的加速比。
    - 优化的矩阵乘法通过减少不必要的访存操作和利用分块矩阵乘法的优化策略，加速比相对较高。
    - 并行计算矩阵乘法通过利用多线程并行计算的特点，能够进一步提高计算速度，加速比最高。

四、遇到的问题及解决方法

1. 按照定义计算矩阵乘法

- 问题：三重循环嵌套导致计算复杂度高，耗时较长。
- 考虑使用分块矩阵乘法等优化方法来减少不必要的计算和访存操作，从而提高计算效率。

2. 分块矩阵乘法算法

- 问题：实现分块矩阵乘法时，需要确定分块大小和合适的优化策略。
- 解决方法：通过实验和测试来选择合适的分块大小，以达到最佳性能。

3. 并行计算矩阵乘法

- 问题：并行计算可能会引发数据竞争和同步问题，导致结果错误或性能下降。
- 解决方法：使用线程同步机制，如互斥锁（mutex）、信号量（semaphore）等来解决数据竞争和同步问题。确保每个线程访问共享资源时的互斥和同步操作，以保证正确的计算结果和高效的并行计算。