CS205 Project 2: Matrix multiplication

Name: Yankai Xu

SID: 12011525

Part 1 - Analysis

The problem is to multiply two matrices in two files, and output the result to another file. A simple simulation to matrix multiplication can do the work well. However, the largest matrix provided is 2048 * 2048, which is a big number. And we don't know the type (int or float or double) of numbers in matrix.

To solve these problems, generic programming is used in the matrix class. And to improve the speed, four implementations of matrix multiplication are involved in the program. Also, `std::chrono` are used to calculate the running speed of each implementation.

The four methods are listed below:

1. A simple matrix multiplication, calculated by first locate result[i, j], and sum the product of A[i, k] and B[k, j].
2. A little improvement of method 1. The ijk loop, as method 1 mentioned, are accessing the memory of B discontinuously. So, by changing the ijk loop to ikj loop, we make memory access continuously, which can improve the speed.
3. Multi-thread matrix multiplication. It' obliviously that result[i, j] only depends on the ith row of A and jth col of B. Therefore, matrix multiplication can run parallel. `std::thread` is used in this method, and I used 4 threads when testing speed.
4. Another Multi-thread matrix multiplication. In this method, `openmp` are used instead of `std::thread`.

There are another methods which can accelerate the multiplication, but modern compiler(for instance, g++) already can do them better than me when I use optimize options 3.

The methods I know are listed below:

1. SIMD. SIMD is a type of parallel processing by using special instructions and registers in cpu. By running `g++ -O3 -S main.cpp -lpthread` and searching in the result assembly language file `main.s,` I found 128 `xmm` words, which indicates that the SIMD register is already in use.
2. Loop Unrolling. As I know little asm language, I can't read the asm file directly to find out whether the compiler did loop unrolling or not. So, I tested it by manually unroll a loop:

```
#include <iostream>
#include <vector>
#include <fstream>
```

```cpp
#include <exception>
#include <functional>
#include <chrono>
#include <iomanip>
#include <thread>

#pragma GCC optimize(3, "Ofast", "inline")

template<typename T>
class Matrix {
private:
    size_t rows, cols;
    std::vector<T> base;

    inline size_t locate(const size_t &row, const size_t &col) const {
        return row * cols + col;
    }

public:

    Matrix() : rows(0), cols(0), base() {}

    Matrix(size_t &row, size_t &col) : rows(row), cols(col), base(row * c

    Matrix multiply_impl_a(Matrix &other) {
        if (cols != other.rows) throw std::invalid_argument("A rows != B
        Matrix res(rows, other.cols);
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < other.cols; j++) {
                for (int k = 0; k < cols; k++) {
                    res(i, j) += (*this)(i, k) * other(k, j);
                }
            }
        }
        return res;
    }

    Matrix<T> multiply_impl_b(Matrix &other) {
        if (cols != other.rows) throw std::invalid_argument("A rows != B
        Matrix res(rows, other.cols);
        for (int i = 0; i < rows; i++) {
            for (int k = 0; k < cols; k++) {
                T t = (*this)(i, k);
                for (int j = 0; j < other.cols; j++) {
                    res(i, j) += t * other(k, j);
                }
            }
        }
        return res;
    }

    Matrix<T> multiply_impl_c(Matrix &other) {
        if (cols != other.rows) throw std::invalid_argument("A rows != B
        Matrix res(rows, other.cols);
        std::thread threads[4];
        for (int x = 0; x < 4; x++) {
            threads[x] = std::thread([this, &other, &res, x]() {
                for (int i = x * rows / 4; i < (x + 1) * rows / 4; i++) {
                    for (int k = 0; k < cols; k++) {
                        T t = (*this)(i, k);
                        for (int j = 0; j < other.cols; j++) {
                            res(i, j) += t * other(k, j);
                        }
                    }
                }
            });
        }
        for (auto &v: threads) v.join();
        return res;
    }

    Matrix<T> multiply_impl_d(Matrix &other) {
        if (cols != other.rows) throw std::invalid_argument("A rows != B
        Matrix res(rows, other.cols);
#pragma omp parallel for
        for (int i = 0; i < rows; i++) {
            for (int k = 0; k < cols; k++) {
                T t = (*this)(i, k);
                for (int j = 0; j < other.cols; j++) {
                    res(i, j) += t * other(k, j);
                }
            }
        }
        return res;
    }

    T &operator()(const size_t i, const size_t j) {
        return base[locate(i, j)];
    }

    friend std::istream &operator>>(std::istream &in, Matrix &mat) {
        size_t cnt = 0;
        for (std::string line; std::getline(in, line);) {
            std::istringstream iss(line);
            for (T num; iss >> num;) {
                mat.base.push_back(num);
            }
            if (cnt == 0) mat.cols = mat.base.size();
            cnt++;
        }
        mat.rows = cnt;
        return in;
    }

    friend std::ostream &operator<<(std::ostream &out, Matrix &mat) {
        for (int i = 0; i < mat.rows; i++) {
            for (int j = 0; j < mat.cols - 1; j++) {
                out << mat(i, j) << " ";
            }
            out << mat(i, mat.cols - 1) << std::endl;
        }
        return out;
    }
};

template<typename T>
T counter(double *time, const std::function<T()> &func) {
    auto start = std::chrono::steady_clock::now();
    T res = func();
    auto end = std::chrono::steady_clock::now();
    *time = std::chrono::duration<double, std::milli>(end - start).count(
    return res;
}

int main(int argc, char **argv) {

    if (argc != 4) {
        std::cout << "Invalid arguments" << std::endl;
        return 0;
    }
    std::ifstream in_a((std::string(argv[1])));
    std::ifstream in_b((std::string(argv[2])));

    Matrix<double> a, b; //generic, can be easily changed to float or dou
    in_a >> a;
    in_b >> b;
    in_a.close();
    in_b.close();
    double time;
    Matrix<double> res = counter<Matrix<double>>(&time, [&a, &b]() {
        return a.multiply_impl_d(b); //code provided are testing implemen
    });

    std::ofstream out(std::string(argv[3]), std::ios::trunc | std::ios::o
    out << std::setprecision(2);
    out << std::setiosflags(std::ios::fixed);

    out << res;
    out.flush();
    out.close();
    std::cout << "impl_a: " << time << "ms" << std::endl;

    return 0;
}
```

Part 3 - Result and Verification

The running time may strongly depend on my device. For example, the time of running

```
$./matmul mat-A-2048.txt mat-B-2048.txt out.txt
```

changes between 800ms and 1000ms. So, each test case will run several times and the result listed below will be the one which is closet to average.

Test case #1:

Matrices' type are set to float, using method multiply_impl_d.

This test case only shows that the program is running correctly.

```
  $./matmul mat-A-32.txt mat-B-32.txt out32.txt
  impl_d: 0.154572ms
```

`out.txt`: `60279.16 67331.41 ......`

Test case #2:

Matrices' type are set to float, using method multiply_impl_d.

```
  $./matmul mat-A-2048.txt mat-B-2048.txt out.txt
  impl_d: 486.651ms
```

`out.txt`: `5252342.00 5053434.00 ......`

Test case #3:

Matrices' type are set to double, using method multiply_impl_d.

```
  $./matmul mat-A-2048.txt mat-B-2048.txt out.txt
  impl_d: 825.208ms
```

`out.txt`: `5252342.29 5053432.25 ......`

It's clear that using float lost 2 or 3 digits of precision here.

Test case #4:

From now, to comparing the speed, all tests are going to use 2048 to 2048 matrices, type set to double.

`out.txt` also won't be listed below, as each answer is exactly the same.

Using method multiply_impl_a.

```
  $./matmul mat-A-2048.txt mat-B-2048.txt out.txt
```

```
impl_a: 62173.4ms
```

Test case #5:

Using method multiply_impl_b.

```
$./matmul mat-A-2048.txt mat-B-2048.txt out.txt
impl_b: 3707.36ms
```

By changing ijk loop to ikj loop, the program runs 20 times faster.

Test case #6:

Using method multiply_impl_c.

```
$./matmul mat-A-2048.txt mat-B-2048.txt out.txt
impl_c: 1275.29ms
```

By using `std::thread`, the program runs 2 times faster than method b.

Test case #7:

Using method multiply_impl_d.

```
$./matmul mat-A-2048.txt mat-B-2048.txt out.txt
impl_d: 850.34ms
```

Using openmp is little faster (about 400ms) than using std::thread. Maybe that is because compilers often do better than ordinary programmers...... After all, maybe it chooses a better thread count and does better optimization than `std::thread`.

Part 4 - Difficulties and Solutions

4.1 Multi Threading

It's my first time using `std::thread`, and I hadn't studied the operating system course before. So I accidentally created mat.Cols thread at first, and caused many errors. To solve these problems, I studied something about thread, and my first solution was creating a thread pool. However, c++ doesn't have a thread pool is stl, write my own thread pool was a little hard for me. At last, I found that the problem is trying to use multi threading in inner loop. If I create threads outside all loops, I don't need to delete old thread and create new one anymore. So I simply use 4 threads to divide A.rows, and it works well.

4.2 I/O

Beside the actual calculation, read and write a matrix to file is also a big problem. They take two or three times more than calculation. I thought about using a more efficient way to read and write, such as do i/o redirection and use `get()` and `put()`, or use c-style file i/o to accelerate. However, I didn't make the improvement in the end. That's because the cost of acceleration is the lost of expandability. I' m now using `std::fstream` to open file, and overloading `operator<<` and `operator>>` to interact with `std::ostream` and `std::istream`. That means if your matrix is in a string rather than a file, is easy to change `std::ifstream` to `std::istringstream`, or even `std::cin`.

4.3 Element Storage and Access

To speed up the multiplication, I use 1-D array rather than 2-D array. That means we can't access the matrix simply with mat[i][j], as overloading `operator[]` need to return something. So I did a little study and find out some libraries are using mat(i, j) to access. So I overloaded `operator()` and it works well.