实验报告

数据结构 *Data Structures*

报告标题: **大规模矩阵计算的性能测试**
学号: 19240212
姓名: 华博文
日期: 2025 年 11 月 18 日

一 实验环境

1 操作系统

Ubuntu 24.04.3 LTS x86_64 , Linux 6.14.0-35-generic , AMD Ryzen 7 7700 (16) @ 5.3GHz , DDR5。

2 编程工具

NeoVim v0.12.0-dev，gcc 13.3.0，Python 3.12.3 (matplotlib == 3.6.3)。

3 其他工具

LaTeX (TeX*studio*)。

二 实验内容及其完成情况

1 实验目的

1. 编写双精度矩阵乘法基准程序，理解计算密集型任务的性能特征；
2. 测试并记录个人计算机能够处理的最大矩阵规模，分析时间和内存瓶颈；
3. 探讨不同操作系统/配置对单进程内存使用的限制与可见性（RSS vs USS）。

2 实现方法

本实验采用以下实现与测量策略：

- **矩阵存储**：行主序，使用单个连续的 `std::vector<double>` 存放元素，按索引计算偏移；
- **矩阵类型**：`Matrix` 类，构造函数 `Matrix(size_t rows, size_t cols)`，提供 `operator()(i, j)` 和 `operator*` 进行矩阵乘法；
- **乘法算法**：朴素三重循环实现，返回新的 `Matrix` 对象；
- **性能测量**：使用 C++ 高精度时钟记录每次乘法所耗时间（秒），每个 N 重复至少 3 次取平均；
- **内存测量**：使用 POSIX `getrusage(RUSAGE_SELF)` 获取进程峰值 RSS（Resident Set Size，单位 KB，报告中转换为 MB）。

3 项目结构

代码在本实验目录下：

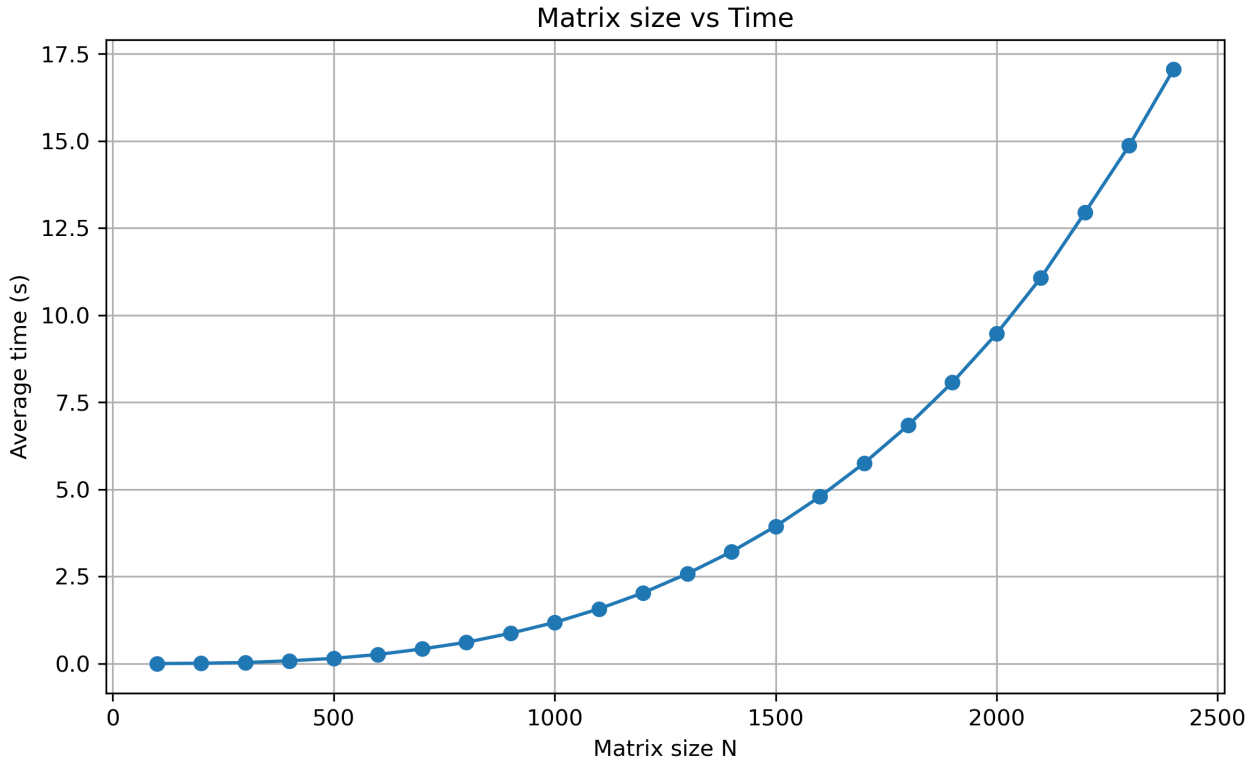
- `src/matrix.h`：`Matrix` 类声明；
- `src/matrix.cpp`：`Matrix` 类实现（**`operator*`**、**`operator()`** 等）；
- `src/main.cpp`：基准运行器，接受参数 `START END STEP REPEATS OUTPUT_CSV`，生成 CSV；
- `src/plot.py`：读取CSV，生成图像`time_vs_n.png`、`mem_vs_n.png`、`comparison.png`。

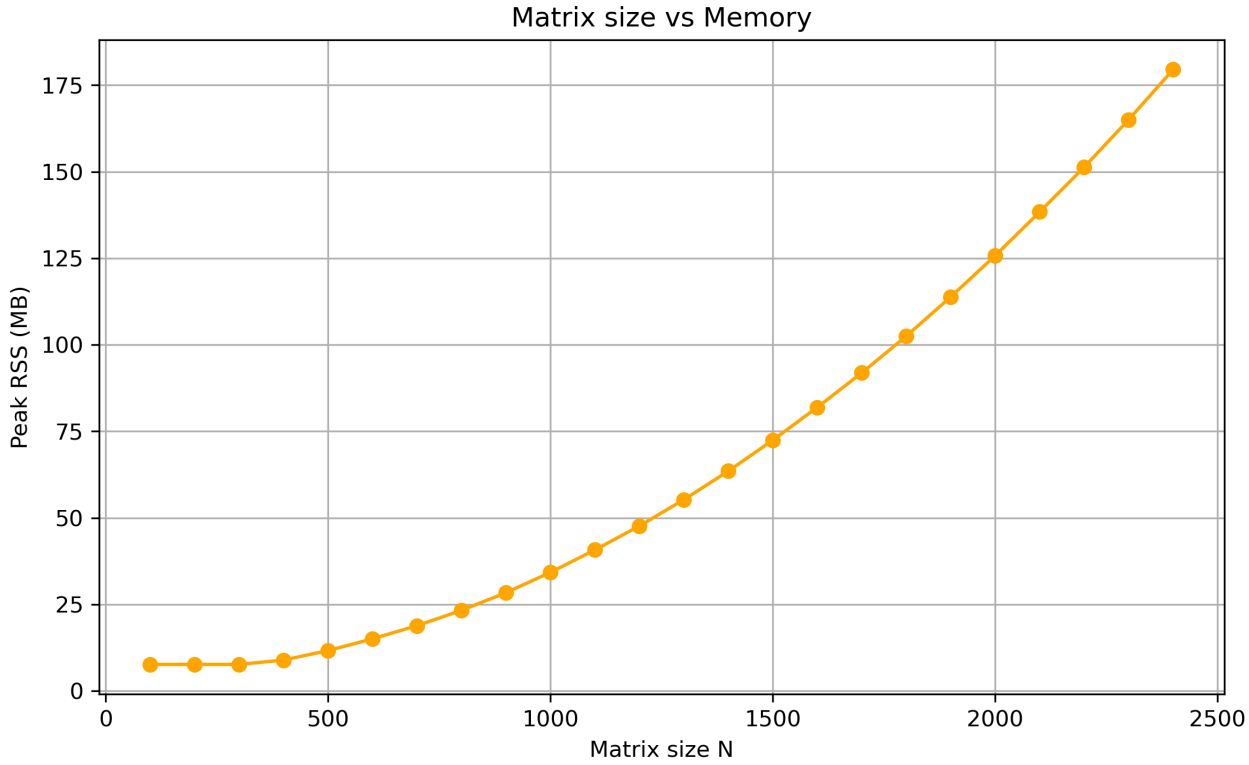
4 运行结果

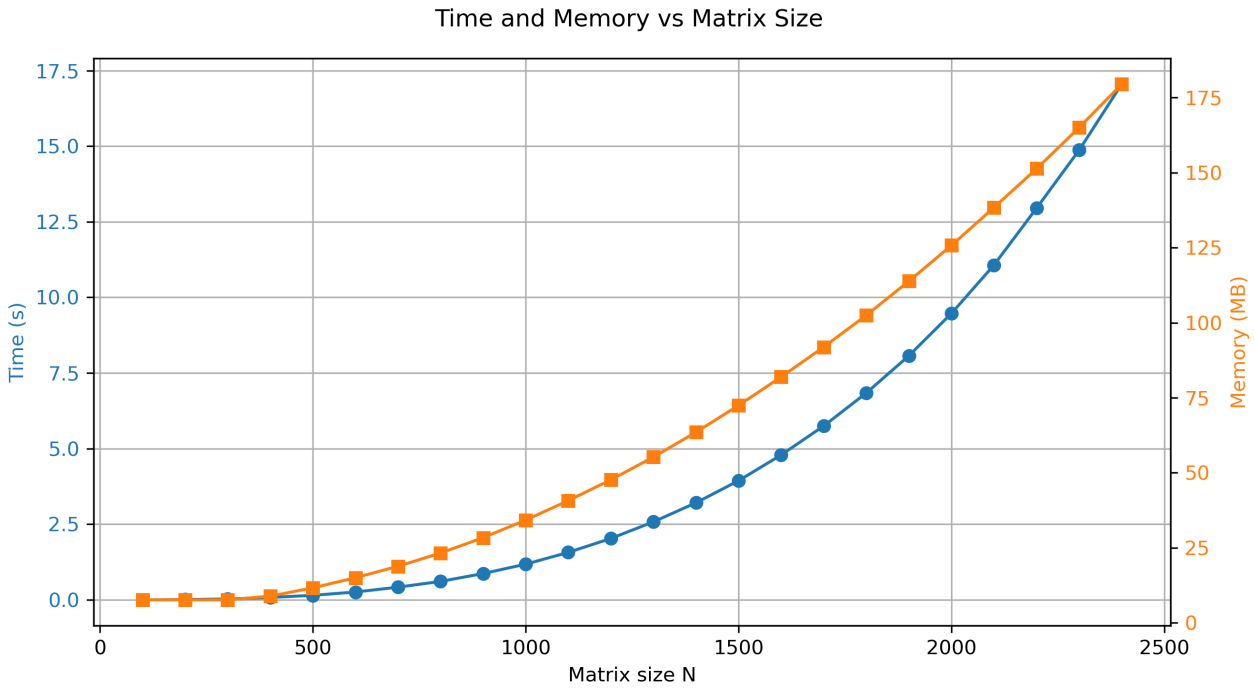
使用 `g++ -O0 -std=c++20 main.cpp matrix.cpp -o matrix_bench` 编译项目，随后运行 `./matrix_bench 100 2400 100 3 results.csv`，在 `results.csv` 中得到如下数据：

N	avg_time_s	peak_rss_MB
100	0.00	7.63
200	0.01	7.63
300	0.03	7.63
400	0.08	8.89
500	0.15	11.64
600	0.26	15.02
700	0.42	18.82
800	0.61	23.25
900	0.87	28.35
1000	1.18	34.23
1100	1.57	40.71
1200	2.03	47.65
1300	2.58	55.23
1400	3.21	63.50
1500	3.94	72.44
1600	4.79	81.92
1700	5.75	91.89
1800	6.84	102.48
1900	8.07	113.84
2000	9.47	125.75
2100	11.07	138.40
2200	12.95	151.32
2300	14.87	165.04
2400	17.05	179.50

由 `python3 plot.py results.csv` 生成的图像：


Matrix size vs Time
Average time (s)
Matrix size N
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
500
1000
1500
2000
2500


Matrix size vs Memory
Peak RSS (MB)
Matrix size N
0
25
50
75
100
125
150
175
0
500
1000
1500
2000
2500



三　数据分析与讨论

1　性能标度

理论上，朴素矩阵乘法的时间复杂度为 $\mathcal{O}(N^3)$；从实验数据可通过对数拟合检验时间随 N^3 增长的趋势。对于小规模（L1 / L2 缓存可容纳）矩阵，缓存命中使实际常数显著更小；当问题规模超过缓存并进入内存访问瓶颈时，时间增长速率加剧。

2　内存限制与崩溃点

峰值 RSS 反映的是进程在系统可见的物理内存占用（包含共享库占用）；如果进程分配失败会抛出 `std::bad_alloc`，程序会在 CSV 中记录崩溃前的最大 N。

3　潜在优化

- 使用块矩阵乘法（blocking / tiling）减少缓存未命中；
- 利用多线程（OpenMP）或调用高性能 BLAS（例如 OpenBLAS / Intel MKL）；
- 避免在基准中频繁分配临时矩阵，可使用预分配输出矩阵并实现 `multiply_to` 接口以排除分配开销对时间测量的影响。

4　结论

本实验提供了一个简单、可重复的矩阵乘法基准框架，满足作业要求：使用双精度、记录时间与峰值内存、给出 CSV 原始数据并绘制对比图表。基准结果显示时间复杂度接近 $\mathcal{O}(N^3)$，内存消耗按 $\mathcal{O}(N^2)$ 增长。要进一步提高性能或扩展测试，可采用阻塞、并行或调用 BLAS 实现。

A 完整代码

1 matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H

#include <cstddef>
#include <vector>

class Matrix {
public:
    Matrix(std::size_t rows, std::size_t cols);

    Matrix(const Matrix &) = default;
    Matrix(Matrix &&) noexcept = default;

    ~Matrix() = default;

    Matrix &operator=(const Matrix &) = default;
    Matrix &operator=(Matrix &&) noexcept = default;

    double &operator()(std::size_t i, std::size_t j) noexcept;
    const double &operator()(std::size_t i, std::size_t j) const
    ↪  noexcept;

    double *operator[](std::size_t i) noexcept;
    const double *operator[](std::size_t i) const noexcept;

    Matrix operator*(const Matrix &other) const;

    std::size_t nrows() const noexcept;
    std::size_t ncols() const noexcept;

    double *data() noexcept;
    const double *data() const noexcept;

    void fill(double v) noexcept;

    double *begin() noexcept { return data(); }
    double *end() noexcept { return data() + (nrows() * ncols()); }
    const double *begin() const noexcept { return data(); }
    const double *end() const noexcept { return data() + (nrows() *
    ↪  ncols()); }
```

```
private:
    std::size_t rows_;
    std::size_t cols_;
    std::vector<double> data_;
};

#endif // MATRIX_H
```

2 matrix.cpp

```
#include <algorithm>
#include <stdexcept>

#include "matrix.h"

Matrix::Matrix(std::size_t rows, std::size_t cols) : rows_(rows),
 ↪  cols_(cols), data_(rows * cols) {}

double &Matrix::operator()(std::size_t i, std::size_t j) noexcept {
 ↪  return data_[i * cols_ + j]; }
const double &Matrix::operator()(std::size_t i, std::size_t j) const
 ↪  noexcept { return data_[i * cols_ + j]; }

double *Matrix::operator[](std::size_t i) noexcept { return &data_[i
 ↪  * cols_]; }
const double *Matrix::operator[](std::size_t i) const noexcept {
 ↪  return &data_[i * cols_]; }

Matrix Matrix::operator*(const Matrix &B) const {
    if (this->ncols() != B.nrows()) {
        throw std::invalid_argument("Incompatible matrix dimensions
         ↪  for multiplication");
    }
    std::size_t R = this->nrows();
    std::size_t K = this->ncols();
    std::size_t Cc = B.ncols();

    Matrix C(R, Cc);
    std::size_t total = R * Cc;
    for (std::size_t i = 0; i < total; ++i)
        C.data()[i] = 0.0;
    for (std::size_t i = 0; i < R; ++i) {
        for (std::size_t k = 0; k < K; ++k) {
```

```
            double aik = (*this)(i, k);
            const double *Brow = &B.data()[k * Cc];
            double *Crow = &C.data()[i * Cc];
            for (std::size_t j = 0; j < Cc; ++j) {
                Crow[j] += aik * Brow[j];
            }
        }
    }

    return C;
}

std::size_t Matrix::nrows() const noexcept { return rows_; }
std::size_t Matrix::ncols() const noexcept { return cols_; }

double *Matrix::data() noexcept { return data_.data(); }
const double *Matrix::data() const noexcept { return data_.data(); }

void Matrix::fill(double v) noexcept { std::fill(data_.begin(),
↪  data_.end(), v); }
```

3 main.cpp

```
#include <chrono>
#include <fstream>
#include <iostream>
#include <string>
#include <sys/resource.h>

#include "matrix.h"

double now_seconds() {
    return std::chrono::duration<double>(
        std::chrono::high_resolution_clock::now().time_since_epoch()
    ).count();
}

size_t get_peak_rss_kb() {
    struct rusage r;
    if (getrusage(RUSAGE_SELF, &r) == 0) {
        return (size_t)r.ru_maxrss;
    }
    return 0;
}
```

```
struct BenchmarkResult {
    double avg_time = 0.0;
    double peak_mb = 0.0;
    std::string notes = "";
    bool oom = false;
    bool skipped = false;
};

static double mean(const std::vector<double> &v) {
    if (v.empty())
        return 0.0;
    double s = 0.0;
    for (double x : v)
        s += x;
    return s / v.size();
}

BenchmarkResult run_benchmark_for_N(size_t N, int repeats) {
    BenchmarkResult res;

    try {
        if (N > (size_t)1e8) {
            res.notes = "N too large (sanity)";
            res.skipped = true;
            return res;
        }

        Matrix A(N, N), B(N, N);
        A.fill(1.0);
        B.fill(1.0);

        std::vector<double> times;
        for (int r = 0; r < repeats; ++r) {
            Matrix C(N, N);

            double t0 = now_seconds();
            C = A * B;
            double t1 = now_seconds();

            times.push_back(t1 - t0);

            size_t peak_kb = get_peak_rss_kb();
            double peak_now_mb = peak_kb / 1024.0;
            if (peak_now_mb > res.peak_mb)
```

```
                res.peak_mb = peak_now_mb;
            }

            res.avg_time = mean(times);

        } catch (const std::bad_alloc &e) {
            res.notes = std::string("OOM: ") + e.what();
            res.oom = true;
        } catch (const std::exception &e) {
            res.notes = std::string("Error: ") + e.what();
        }

        return res;
    }

    int main(int argc, char **argv) {
        if (argc < 6) {
            std::cerr << "Usage: " << argv[0] << " START END STEP REPEATS
             ↪  OUTPUT_CSV\n";
            std::cerr << "Example: " << argv[0] << " 100 500 100 3
             ↪  results.csv\n";
            return 1;
        }

        size_t start = std::stoull(argv[1]);
        size_t end = std::stoull(argv[2]);
        size_t step = std::stoull(argv[3]);
        int repeats = std::stoi(argv[4]);
        std::string out_csv = argv[5];
        if (step == 0) {
            std::cerr << "STEP must be > 0\n";
            return 1;
        }

        std::ofstream fout(out_csv);
        if (!fout) {
            std::cerr << "Cannot open output file " << out_csv << "\n";
            return 1;
        }

        fout << "N,avg_time_s,peak_rss_MB,notes\n";

        for (size_t N = start; N <= end; N += step) {
            std::cout << "Testing N=" << N << "\n";
```

```
        BenchmarkResult res = run_benchmark_for_N(N, repeats);

        if (res.skipped) {
            std::cout << "Skipping N=" << N << " (sanity check)\n";
            fout << N << ",," << "0.00," << res.notes << "\n";
            fout.flush();
            continue;
        }

        if (res.oom) {
            std::cout << "Allocation failed for N=" << N << "\n";
            fout << N << ",, ," << res.notes << "\n";
            fout.flush();
            break;
        }

        if (!res.notes.empty()) {
            std::cout << res.notes << "\n";
        }

        fout << std::format("{},{:.2f},{:.2f},{}\n", N, res.avg_time,
         ↪  res.peak_mb, res.notes);
        fout.flush();
    }

    std::cout << "Benchmark finished. Results in " << out_csv << "\n";
    return 0;
}
```

4 plot.py

```
import csv
import matplotlib.pyplot as plt
import sys


def read_csv(path):
    ns = []
    times = []
    mems = []
    with open(path) as f:
        r = csv.DictReader(f)
        for row in r:
            try:
```

```
                n = int(row["N"])
            except:
                continue
            ns.append(n)
            try:
                times.append(float(row["avg_time_s"]))
            except:
                times.append(float("nan"))
            try:
                mems.append(float(row["peak_rss_MB"]))
            except:
                mems.append(float("nan"))
    return ns, times, mems


def plot_time(ns, times, out="time_vs_n.png"):
    plt.figure(figsize=(8, 5))
    plt.plot(ns, times, marker="o")
    plt.xlabel("Matrix size N")
    plt.ylabel("Average time (s)")
    plt.title("Matrix size vs Time")

    plt.grid(True)
    plt.tight_layout()
    plt.savefig(out, dpi=300)


def plot_mem(ns, mems, out="mem_vs_n.png"):
    plt.figure(figsize=(8, 5))
    plt.plot(ns, mems, marker="o", color="orange")
    plt.xlabel("Matrix size N")
    plt.ylabel("Peak RSS (MB)")
    plt.title("Matrix size vs Memory")

    plt.grid(True)
    plt.tight_layout()
    plt.savefig(out, dpi=300)


def plot_comparison(ns, times, mems, out="comparison.png"):
    fig, ax1 = plt.subplots(figsize=(9, 5))

    ax1.set_xlabel("Matrix size N")
    ax1.set_ylabel("Time (s)", color="tab:blue")
    ax1.plot(ns, times, marker="o", color="tab:blue", label="Time")
```

```
    ax1.tick_params(axis="y", labelcolor="tab:blue")

    ax2 = ax1.twinx()
    ax2.set_ylabel("Memory (MB)", color="tab:orange")
    ax2.plot(ns, mems, marker="s", color="tab:orange",
     ↪  label="Memory")
    ax2.tick_params(axis="y", labelcolor="tab:orange")

    ax1.grid(True)
    fig.suptitle("Time and Memory vs Matrix Size")
    fig.tight_layout()
    plt.savefig(out, dpi=300)


def main():
    if len(sys.argv) < 2:
        print("Usage: python3 plot.py results.csv")
        return
    path = sys.argv[1]
    ns, times, mems = read_csv(path)
    if not ns:
        print("No data found in", path)
        return
    plot_time(ns, times)
    plot_mem(ns, mems)
    plot_comparison(ns, times, mems)
    print("Saved: time_vs_n.png, mem_vs_n.png, comparison.png")


if __name__ == "__main__":
    main()
```