中山大学计算机院本科生实验报告

（2025学年春季学期）

课程名称：并行程序设计
批改人：

实验	环境设置与串行矩阵乘法	专业（方向）	计算机科学与技术
学号	22336087	姓名	胡瑞康
Email	hurk3@mail2.sysu.edu.cn	完成日期	2025.3.10

1 实验目的

1. 掌握串行矩阵乘法的基本实现方法，对比不同编程语言（Python与C++）的性能差异
2. 探索循环顺序调整、编译器优化等技术对计算性能的影响
3. 验证高性能数学库（如OpenBLAS）在密集矩阵运算中的加速效果
4. 通过GFLOPS指标量化评估不同实现方案的浮点运算效率

2 实验过程和核心代码

2.1 核心代码

2.1.1 Python传统三重循环

文件实现在`1.py`

随机矩阵生成

```
def generate_matrix(rows, cols):
    # 生成 rows x cols 随机矩阵（元素取值0～1浮点数）
    return [[random.random() for _ in range(cols)] for _ in range(rows)]
```

矩阵计算函数

```
def matrix_multiply(A, B):
    m = len(A)
    n = len(A[0])
    p = len(B[0])
    # 初始化结果矩阵
    C = [[0.0 for _ in range(p)] for _ in range(m)]
    for i in range(m):
        for j in range(p):
            sum_val = 0.0
            for k in range(n):
                sum_val += A[i][k] * B[k][j]
            C[i][j] = sum_val
    return C
```

问题：Python解释执行慢；B矩阵按列访问导致缓存命中率低

2.1.2 C++传统三重循环

文件实现在`2.cpp`

随机矩阵生成

```
Matrix generate_matrix(int rows, int cols) {
    Matrix mat(rows, vector<double>(cols));
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            mat[i][j] = static_cast<double>(rand()) / RAND_MAX;
    return mat;
}
```

矩阵计算函数

```
Matrix matrix_multiply(const Matrix &A, const Matrix &B) {
    int m = A.size();
    int n = A[0].size();
    int p = B[0].size();
    Matrix C(m, vector<double>(p, 0.0));
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            double sum = 0.0;
            for (int k = 0; k < n; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
    return C;
}
```

问题：B矩阵按列访问导致缓存命中率低

2.1.3 C++循环顺序优化

文件实现在`3.cpp`

矩阵计算函数

```
Matrix matrix_multiply(const Matrix &A, const Matrix &B) {
    int m = A.size();
    int n = A[0].size();
    int p = B[0].size();
    Matrix C(m, vector<double>(p, 0.0));
    for (int i = 0; i < m; i++) {
        double* c_row = &C[i][0]; // 缓存当前行指针
        for (int k = 0; k < n; k++) {
            double temp = A[i][k];
            const double* b_row = &B[k][0]; // 缓存B中第k行指针
            for (int j = 0; j < p; j++) {
                c_row[j] += temp * b_row[j];
            }
        }
    }
    return C;
```

优化：通过调整循环顺序提升空间局部性，B矩阵按行访问

2.1.4 OpenBLAS加速

文件实现在`4.cpp`

将二维矩阵转换为一维数组（行优先顺序），方便调用`cblas_dgemm`函数。

```
vector<double> convert_to_array(const Matrix &mat) {
    int rows = mat.size();
    int cols = mat[0].size();
    vector<double> array(rows * cols);
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            array[i * cols + j] = mat[i][j];
    return array;
}
```

使用 BLAS 的 cblas_dgemm 进行矩阵乘法。

C = alpha * A * B + beta * C, 这里 alpha = 1.0, beta = 0.0。

参数说明：矩阵存储采用 row-major，A 为 MxN, B 为 NxK, C 为 MxK。

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            M, K, N, 1.0, A_arr.data(), N, B_arr.data(), K, 0.0, C_arr.data(), K);
```

优势：调用高度优化的BLAS库函数，利用SIMD指令和多线程并行

2.1.5 统一评估脚本

定义一个列表用于存储编译运行命令

```
experiments = [
{
    "version": "1",
    "description": "Python",
    "compile": None,
    "run": f"python3 1.py {m} {n} {k}"
},
{
    "version": "2",
    "description": "C/C++ (传统三重循环实现) ",
    "compile": "g++ 2.cpp -O0 -o 2",
    "run": f"./2 {m} {n} {k}"
},
...
```

通过`main`函数来调用各个程序运行，并计算出实验结果表格

```
def main():
    results = []
    print("开始编译和运行各个版本……")
    for exp in experiments:
        print(f"\n=== 版本 {exp['version']}: {exp['description']} ===")
        # 如果有编译命令，则先编译
        if exp["compile"]:
            print("正在编译...")
```

```
            if ret != 0:
                print(f"编译失败，错误信息：\n{err}")
                sys.exit(1)
            else:
                print("编译成功。")
        # 运行命令
        print("正在运行...")
        ret, out, err = run_command(exp["run"])
        if ret != 0:
            print(f"运行失败，错误信息：\n{err}")
            sys.exit(1)
        # 打印完整输出（可选）
        print("程序输出：")
        print(out)
        # 提取运行时间
        t = extract_time(out)
        if t is None:
            print("未检测到 'Time taken' 输出。")
            t = 0.0
        else:
            print(f"检测到运行时间：{t} seconds")
        # 计算浮点性能（GFLOPS），如果运行时间有效则计算，否则为0
        if t > 0:
            gflops = flops / (t * 1e9)
        else:
            gflops = 0.0
        # 计算峰值性能百分比
        peak_percent = (gflops / device_peak_gflops) * 100 if device_peak_gflops > 0 else
0.0
        results.append({
            "version": exp["version"],
            "description": exp["description"],
            "time": t,
            "gflops": gflops,
            "peak_percent": peak_percent
        })

    # 找出最快的时间（非零值）
    valid_times = [r["time"] for r in results if r["time"] > 0]
    if not valid_times:
        print("没有检测到有效的运行时间数据。")
        sys.exit(1)
    fastest = min(valid_times)

    # 构造 Markdown 表格
    md_table = []
    header = "|版本|实现描述|运行时间(sec.)|相对加速比|绝对加速比|浮点性能(GFLOPS)|峰值性能百分
比|"
    separator = "|---|---|---|---|---|---|---|"
    md_table.append(header)
    md_table.append(separator)

    # 获取版本1的运行时间（用于计算绝对加速比）
    version1_time = next((r["time"] for r in results if r["version"] == "1" and r["time"] >
0), None)
```

```
    for i, r in enumerate(results):
        # 计算相对加速比（相对于前一版本）
        rel_speedup = ""
        if r["time"] > 0:
            if i > 0 and results[i-1]["time"] > 0:  # 如果有前一个版本且时间有效
                rel_speedup = results[i-1]["time"] / r["time"]
            else:
                rel_speedup = 1.0  # 第一个版本的相对加速比为1

        # 计算绝对加速比（相对于版本1）
        abs_speedup = ""
        if r["time"] > 0 and version1_time is not None and version1_time > 0:
            abs_speedup = version1_time / r["time"]  # 使用版本1的时间计算绝对加速比

        # 格式化各项数据（时间和加速比保留6位小数，峰值百分比保留4位小数）
        time_str = f"{r['time']:.6f}" if r["time"] > 0 else ""
        rel_str = f"{rel_speedup:.6f}" if isinstance(rel_speedup, float) else ""
        abs_str = f"{abs_speedup:.6f}" if isinstance(abs_speedup, float) else ""
        gflops_str = f"{r['gflops']:.6f}" if r["time"] > 0 else ""
        peak_str = f"{r['peak_percent']:.4f}%" if r["time"] > 0 else ""
        row = f"|{r['version']}|{r['description']}|{time_str}|{rel_str}|{abs_str}|
{gflops_str}|{peak_str}|"
        md_table.append(row)

    # 输出 Markdown 表格
    print("\n最终生成的 Markdown 表格：\n")
    md_result = "\n".join(md_table)
    print(md_result)
```

2.2 实验过程

查询得到本机的AMD R7 8845H的GFLOPS为843.8（参考地址https://gadgetversus.com/processor/amd-ryzen-7-8845h-gflops-performance/）

使用M=1024 N=512 K=512参数，分别测试了Python传统实现，C++传统实现，C++调整循环顺序，以及对C++调整循环顺序使用O1-O3的编译优化，最后测试了OpenBLAS的加速实现。

调用`evaluae.py`进行评测构造结果表格

3 实验结果

版本	实现描述	运行时间(sec.)	相对加速比	绝对加速比	浮点性能(GFLOPS)	峰值性能百分比
1	Python	25.053343	1.000000	1.000000	0.021429	0.0025%
2	C/C++（传统三重循环实现）	1.858630	13.479468	13.479468	0.288853	0.0342%
3	调整循环顺序	1.093190	1.700189	22.917647	0.491105	0.0582%
4	编译优化（-O1）	0.078597	13.908818	318.757394	6.830688	0.8095%
5	循环展开（-O2）	0.075788	1.037068	330.573115	7.083889	0.8395%
6	O3优化（-O3）	0.049876	1.519508	502.308567	10.764027	1.2757%
7	OpenBLAS（4.cpp -lopenblas）	0.013348	3.736676	1876.964219	40.221678	4.7667%

性能跃迁关键点：

1. Python版（v1）受限于解释型语言特性，性能比C++低13倍
2. 循环顺序优化（v3）带来70%性能提升，验证内存访问模式的重要性
3. 编译优化（v4-v6）通过指令级并行和循环展开实现13-22倍加速
4. OpenBLAS（v7）利用硬件级优化实现1876倍绝对加速，达到4.77%峰值利用率

4 4. 实验感想

在本次实验中，由于使用的是AMD R7 8845H处理器，考虑到硬件兼容性和优化支持的问题，直接使用Intel oneAPI Math Kernel Library (MKL)可能无法充分发挥性能优势。此外，Intel MKL主要针对Intel处理器进行了深度优化，而在AMD平台上可能无法达到最佳效果。因此，为了确保实验结果的公平性和准确性，我选择了开源的OpenBLAS库作为替代方案。

OpenBLAS是一个高度优化的BLAS（Basic Linear Algebra Subprograms）实现，支持多种硬件架构，包括AMD和Intel处理器。它通过多线程并行计算、SIMD指令集加速等技术，在密集矩阵运算中表现出色。与Intel MKL类似，OpenBLAS也提供了高效的矩阵乘法实现（如`cblas_dgemm`函数），并且其开源特性使得它更易于集成和调试。

在实验过程中，我通过调用OpenBLAS的`cblas_dgemm`函数实现了矩阵乘法，并观察到显著的性能提升。相比传统的C++实现，OpenBLAS版本的运行时间大幅缩短，达到了4.77%的峰值性能利用率，这验证了高性能数学库在计算密集型任务中的重要性。同时，OpenBLAS的跨平台特性也为未来在不同硬件环境下的实验提供了更大的灵活性。