中山大學

SUN YAT-SEN UNIVERSITY

并行程序设计与算法实验

Lab0-环境设置与串行矩阵乘法

姓 名 李源卿

学 号 22336128

学 院 计算机学院

专 业 计算机科学与技术

2025 年 3 月 28 日

1 实验目的

- 理解并行程序设计的基本概念与理论。
- 掌握使用并行编程模型实现常见算法的能力。
- 学习评估并行程序性能的指标及其优化方法。

2 实验内容

- 设计并实现以下矩阵乘法版本：
  - 使用 C/C++ 语言实现一个串行矩阵乘法。
  - 比较不同编译选项、实现方式、算法或库对性能的影响：
    * 使用 Python 实现的矩阵乘法。
    * 使用 C/C++ 实现的基本矩阵乘法。
    * 调整循环顺序优化矩阵乘法。
    * 应用编译优化提高性能。
    * 使用循环展开技术优化矩阵乘法。
    * 使用 Intel MKL 库进行矩阵乘法运算。
- 生成随机矩阵 A 和 B，进行矩阵乘法运算得到矩阵 C。
- 衡量各版本的运行时间、加速比、浮点性能等。
- 分析不同实现版本对性能的影响。

3 实验思路以及实现

- 数据初始化生成的均是在 [0,1] 服从均匀分布的浮点数：

```
// c/c++
#include <random>
random_device rd;  // 随机种子
mt19937 gen(rd()); // 随机数引擎
uniform_real_distribution<> dis(0.0, 1.0); // 均匀分布 [0, 1)
// 初始化矩阵
void initi(vector<vector<double>>&matrix,int rows,int cols){
    for (int i = 0; i < rows; ++i) {
        vector<double> row(cols);
        for (int j = 0; j < cols; ++j) {
```

```
            row[j] = dis(gen);
        }
        matrix.push_back(row);
    }
```

```python
# Python
import random
def generate_random_matrix(rows, cols):
    """
    生成随机矩阵
    """
    return [[random.random() for _ in range(cols)] for _ in
       range(rows)]
```

- 计时器

```c
//c语言
#include <sys/time.h>
gettimeofday(&start, NULL); // 开始计时
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            m, n, k, alpha, A, k, B, n, beta, C, n);
gettimeofday(&end, NULL); // 结束计时
```

```cpp
//c++
#include <chrono>
using namespace std::chrono;
auto start = high_resolution_clock::now();
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            m, n, k, alpha, A, k, B, n, beta, C, n);
auto end = high_resolution_clock::now();
```

```python
#Python
import time
start=time.time()
C = matrix_multiply(A, B)
end=time.time()
```

为了让 MKL 中的矩阵也初始化为 [0,1] 之间的数，我稍微修改了老师给的代码：

```
//MKL
for (i = 0; i < (m * k); i++) {
```

```
    A[i] = (double)rand() / RAND_MAX;
}

for (i = 0; i < (k * n); i++) {
    B[i] = (double)rand() / RAND_MAX;
}

for (i = 0; i < (m * n); i++) {
    C[i] = 0.0;
}
```

- 使用 Python 实现的矩阵乘法：

```
# Python矩阵乘法(仅给出关键部分)
for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                result[i][j] += A[i][k] * B[k][j]
    return result
```

- 使用 Numpy 实现的矩阵乘法：
  Numpy 中的矩阵乘法底层是通过 c/c++ 实现的，效率较高。

```
# Python矩阵乘法(仅给出关键部分)
import numpy as np
import time
m, k, n = 1000,1000 , 1000
A = np.random.rand(m, k)
B = np.random.rand(k, n)
start=time.time()
C = np.dot(A, B)
end=time.time()
print("time cost:",end-start)
```

- 使用 c++ 实现的普通矩阵乘法：

```
//c++实现普通矩阵乘法
vector<vector<double>> matrixMultiply(const vector<vector<double>>& A, const vector<vector<double>>& B) {
    int rows_A = A.size();
    int cols_A = A[0].size();
```

```
    int rows_B = B.size();
    int cols_B = B[0].size();
    // 检查矩阵维度是否可乘
    if (cols_A != rows_B) {
        cerr << "Error: Matrix dimensions do not match for multiplication!" << endl;
        return {};
    }
    vector<vector<double>> result(rows_A, vector<double>(cols_B, 0.0));
    // 矩阵乘法计算
    for (int i = 0; i < rows_A; ++i) {
        for (int j = 0; j < cols_B; ++j) {
            for (int k = 0; k < cols_A; ++k) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}
```

- 尝试调整上述 c++ 代码的循环顺序：
  我们其实可以发现，由于计算机一般是按行优先存储，所以在第三层循环里，B[k][j] 和 B[k+1][j] 在物理内存上距离较远，就很有可能造成 cache 的 miss，增加不必要的 I/O 时间。所以思路就是让 k 和 j 循环调换位置。

```
//c++实现普通调整过顺序的矩阵乘法
// 调整后的循环顺序：i -> k -> j(其他的都和前面一样)
for (int i = 0; i < rows_A; ++i) {
    for (int k = 0; k < cols_A; ++k) {
        double a = A[i][k];                 // 缓存A[i][k]，减少重复访问
        for (int j = 0; j < cols_B; ++j) {  // 遍历B的某一行的全部元素
            result[i][j] += a * B[k][j];
        }
    }
}
```

- 循环展开

循环展开最明显的益处在于减少了循环次数，并且由于一次性取出很多变量，从而减少了内存访问。展开的方式有多种，我首先尝试了对 j 做两路展开，k 做四路展开：

```
//c++实现循环展开（j两路，k四路）
...(其他的都和前面展示的一样)
for (int i = 0; i < rows_A; ++i) {
    int j = 0;
    // 每次处理2个j（列）以减少循环次数
    for (; j <= cols_B - 2; j += 2) {
        double sum1 = 0.0, sum2 = 0.0;
        int k = 0;
        // 每次处理4个k（展开因子=4）
        for (; k <= cols_A - 4; k += 4) {
            // 预加载A的元素
            const double a0 = A[i][k];
            const double a1 = A[i][k+1];
            const double a2 = A[i][k+2];
            const double a3 = A[i][k+3];

            // 为两个不同的j值计算乘积并累加
            sum1 += a0 * B[k][j] + a1 * B[k+1][j] + a2 * B[k+2][j] + a3 * B[k+3][j];
            sum2 += a0 * B[k][j+1] + a1 * B[k+1][j+1] + a2 * B[k+2][j+1] + a3 * B[k+3][j+1];
        }
        // 处理剩余k值
        for (; k < cols_A; ++k) {
            const double a = A[i][k];
            sum1 += a * B[k][j];
            sum2 += a * B[k][j+1];
        }
        result[i][j] = sum1;
        result[i][j+1] = sum2;
    }
    // 处理剩余j值
    for (; j < cols_B; ++j) {
        double sum = 0.0;
        int k = 0;
        // 同样展开k循环
```

```
            for (; k <= cols_A - 4; k += 4) {
                sum += A[i][k] * B[k][j] + A[i][k+1] * B[k+1][j]
                    +
                       A[i][k+2] * B[k+2][j] + A[i][k+3] * B[k
                          +3][j];
            }
            // 处理剩余k值
            for (; k < cols_A; ++k) {
                sum += A[i][k] * B[k][j];
            }
            result[i][j] = sum;
        }
    }
    ...
```

我还尝试了对 j 做四路展开，k 做八路展开：确实更快了，但是效果就没有那么立竿见影了，由于原理是一样的，我就不把代码放到报告里了。同时我还对调整过循环顺序的矩阵乘法做过循环展开，也不放在实验报告里了，对于调整过循环顺序的矩阵乘法，只需要简单做一些展开就能达到不错的效果。

- Intel MKL：
  这个代码已经给我们了，*cblas_dgemm* 函数可以执行 $C = alpha * A * B + beta * C$，那我们只需要把 alpha 设置为 1.0，beta 设置为 0 就好了。

```
//MKL
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
    m, n, k, alpha, A, k, B, n, beta, C, n);
```

4 实验结果

4.1 矩阵乘法的定义：

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$$

- 每个 C_{ij} 的计算需要：
  - n 次乘法 $(A_{ik} \times B_{kj})$
  - $n-1$ 次加法（累加求和）
- 因此，每个 C_{ij} 需要 $2n-1$ 次浮点运算。

4.2 整个矩阵 C 的计算：

- C 有 $n \times n$ 个元素，因此总运算次数为：

$$n^2 \times (2n - 1) = 2n^3 - n^2$$

- 当 n 较大时（如 $n \geq 100$），n^2 相比 $2n^3$ 可忽略不计，因此近似为：

$$\text{FLOPs} \approx 2n^3 = 2{\times}1000^3 = 2 \times 10^9$$

4.3 理论峰值浮点性能 (Theoretical Peak FLOPS)

公式如下：

$$\text{Peak FLOPS} = \text{Num\_Cores} \times \text{Clock\_Speed (Hz)} \times \text{FLOPS\_per\_Cycle}$$

- **Num_Cores**：处理器核心数。
- **Clock_Speed**：处理器主频（Hz）。
- **FLOPS_per_Cycle**：每个时钟周期能执行的浮点运算数。

先进技术

英特尔® Volume Management Device (VMD)	是
英特尔® 高斯和神经加速器	2.0
英特尔® 智音技术	是
英特尔® 语音唤醒	是
英特尔® 高质音频	是
MIPI SoundWire*	1.1
英特尔® Adaptix™ 技术	是
支持英特尔® 傲腾™ 内存‡	是
英特尔® Speed Shift Technology	是
英特尔® 睿频加速技术‡	2.0
英特尔® 超线程技术‡	是
指令集	64-bit
指令集扩展	Intel® SSE4.1, Intel® SSE4.2, Intel® AVX2, Intel® AVX-512
空闲状态	是
温度监视技术	是

图 1: CPU 支持的指令集

AVX-512（Advanced Vector Extensions 512）是 Intel 推出的单指令多数据（SIMD）指令集扩展，旨在显著提升 CPU 的并行浮点和整数运算能力。它通过 512 位宽向

量寄存器，允许单条指令同时处理多达 16 个单精度（32 位）或 8 个双精度（64 位）浮点数。由于支持 AVX-512，每个周期能执行的浮点运算为 512/64=8。

我采用了最大睿频频率（见图 2）：

CPU 规格

内核数	4
总线程数	8
最大睿频频率	4.20 GHz
缓存	8 MB Intel® Smart Cache
总线速度	4 GT/s
可配置的 TDP-up 频率	2.40 GHz
可配置的 TDP-up	28 W
可配置的 TDP-down 频率	900 MHz
可配置的 TDP-down	12 W
英特尔® 深度学习提升	是

图 2: 笔记本电脑的 CPU 规格

故峰值性能为 $4 \times 4.2 \times 10^9 \times 8 = 134,400,000,000$ 次每秒

版本	实现描述	运行时间	相对加速比	绝对加速比	浮点性能	峰值性能百分比
1	Python	199.45377s	—	1	10,027,386	0.00219%
2	C/C++	19.387s	10.3	10.3	103,161,913	0.0768%
3	调整循环顺序	7.44672s	2.60	26.8	268,574,621	0.1998%
4	循环展开 (j 四 +k 八)	4.58089s	1.63	43.5	436,596,382	0.3248%
5	编译优化 (Ofast)	1.86329s	2.46	107	1,073,370,221	0.799%
6	Intel MKL	0.017153s	109	11628	116,597,679,706	86.75%
7	Numpy	0.017150s	1	11629	116,618,075,801	86.77%

5 实验分析

5.1 浮点性能优化分析

从上面的表格可以看出，如果不调用 Intel MKL 或 numpy，我们峰值性能百分比是非常感人的。我认为原因有两个：

- 首先我们的程序是跑在单核上的，这导致有三个核是在空闲的，效率拉满也只有 25% 了。
- 其次是我们的程序没有使用 AVX 指令集，导致 CPU 中有大量的寄存器也是空闲的。

5.2 编译优化分析

版本	实现描述	运行时间	相对加速比	绝对加速比
1	O	3.33031s	—	1
2	O1	3.56201s	0.93	0.93
3	O2	1.89658s	1.88	1.76
4	O3	1.72805s	1.10	1.93
5	Ofast	1.72292s	1	1.93

根据上表，我们可以看到 O 和 O1 优化是差不多效果，而 O2,O3,Ofast 优化会快一些，但是彼此间差不多。这说明 O 优化做的一些事情导致了速度的加快，而 O1 相比 O 优化做的事情则对程序的运行效率没有明显帮助。O2,O3,Ofast 同理。我猜测 O 和 O1 对于此程序的优化主要是循环展开，O2，O3，Ofast 则是数组访问加速。