并行程序设计与算法实验0

实验	串行矩阵算法	专业	计算机科学与技术
学号	21311525	姓名	马梓培
Email	mazp@mail2.sysu.edu.cn	完成日期	2024/03/23

1. 实验目的

根据数学定义用C/C++语言实现一个串行矩阵乘法，并通过对比实验分析其性能。
通用矩阵乘法: $C = A \cdot B$, 其中 A 为 $m \times n$ 的矩阵, B 为 $n \times k$ 的矩阵,则其乘积 C 为 $m \times k$ 的矩阵, C 中第 i 行第 j 列元素可由矩阵 A 中第 i 行向量与矩阵 B 中第 j 列向量的内积给出, 即:

$$C_{\mathrm{i,j}} = \sum_{p=1}^{n} A_{i,p} B_{p,j} \tag{1}$$

输入： m, n, k 三个整数, 每个整数的取值范围均为 $[512, 2048]$
问题描述： 随机生成 $m \times n$ 的矩阵 A 及 $n \times k$ 的矩阵 B, 并对这两个矩阵进行矩阵乘法运算, 得到矩阵 C.
输出： A, B, C 三个矩阵, 及矩阵计算所消耗的时间 t 。
要求： 实现多个版本的串行矩阵乘法（可考虑多种语言/编译选项/实现方式/算法/库），填写表格，并对比分析不同因素对最终性能的影响。

2. 环境配置

2.1 安装Ubuntu

1. 安装 VirtualBox: https://www.virtualbox.org/wiki/Downloads
2. 下载 Ubuntu 18.04 镜像文件: http://releases.ubuntu.com/18.04/
3. 在 VirtualBox 中创建虚拟机实例, 并调整其资源, 如 CPU 核、内存、硬盘容量等。
4. 在虚拟机实例中的虚拟光驱中加载 Ubuntu 镜像文件(iso 文件),并安装 Ubuntu 操作系统。

2.2 OpenMPI 命令行安装：

```
sudo apt-get update
sudo apt-get install libopenmpi-dev -y
sudo apt-get install vim -y
```

2.3 Intel oneAPI Math Kernel Library(MKL)命令行安装：

```
wget https://registrationcenter-download.intel.com/akdlm/IRC_NAS/86d6a4c1-c998-4c6b-9fff-ca004e9f7455/l_onemkl_p_2024.0.0.49673_offline.sh

sudo sh ./l_onemkl_p_2024.0.0.49673_offline.sh
```

修改环境变量：

```
source /opt/intel/oneapi/setvars.sh
```

可以输入env | grep SETVARS_COMPLETED来检查一下，输出结果是SETVARS_COMPLETED=1，就证明配置成功了！

```
env | grep SETVARS_COMPLETED
```

3. 串行矩阵算法实现

3.1 Python

通用矩阵乘法实现：

```
for i in range(m):
    for j in range(k):
        for p in range(n):
            C[i, j] += A[i, p] * B[p, j]
```

记录运行时间和GFLOPS：

```
# 计算总的浮点运算次数（这里只计算乘法）
total_flops = m * n * k
# 计算GFLOPS
gflops = total_flops / (elapsed_time * 1e9)
```

3.2 Naive

c++ naive 版本就是3个for循环实现通用矩阵乘法

```
void matrix_multiply(float* A, float* B, float* C, int m, int n, int k) {
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < k; ++j) {
            float sum = 0;
            for (int x = 0; x < n; ++x) {
                sum += *(A + i * n + x) * *(B + x * k + j);
            }
            *(C + i * k + j) = sum;
        }
    }
}
```

3.3 Change Order

在简单的代码实现中，由于矩阵以行主序方式存储，对于计算C矩阵中的元素值，采取的是先列后行的顺序。这样虽然能保证对A矩阵元素的连续访问，但对于B和C矩阵而言，访问模式就显得不那么高效了。特别是在最内层的循环中，对于k维度，访问B[j+k*m]时会产生较大的跨步访问。为了改善这种情况，可以考虑交换i和j维度的循环顺序，以利用数据的空间局部性，提高数据复用率。

```
void matrixMultiply_change_order(float *A, float *B, float *C, int m, int n, int k)
{
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            float a = A[i * n + j];
            for (int x = 0; x < k; x++) {
                C[i * k + x] += a * B[j * k + x];
            }
        }
    }
}
```

3.4 编译优化

在针对简单版本的代码，可以首先采取一种直接的方法，即应用一系列编译优化选项，如`-O3 -fomit-frame-pointer -march=armv8-a -ffast-math`，来促使编译器最大限度地实施自动向量化，以提高代码效率。
终端命令：

```
g++ matrix_mul.cpp -O3 -fomit-frame-pointer -march=native -ffast-math -o
matrix_mul.out
```

编译选项解释：

- `-O3`：这是一个优化级别设置。它告诉编译器使用尽可能多的优化技术来提高程序的执行速度。它比-O2级别做得更多，可能会让编译时间变长，但是目的是为了让最终的程序运行得更快。
- `-fomit-frame-pointer`：这个选项会让编译器尝试不使用帧指针(frame pointer)，这样可以节省一些寄存器资源，可能会让程序运行得更快一些，特别是在寄存器非常宝贵的情况下。
- `-march=armv8-a`：这个选项指定了目标CPU的架构类型，在这个例子中是ARMv8-A架构。这意味着编译器会生成专门为这种类型的CPU优化过的代码，利用该架构的特定功能来提高性能。
- `-ffast-math`：这个选项放宽了一些数学计算的精度和标准规则，让编译器可以采取一些额外的数学相关优化措施来加速计算过程，但可能会牺牲一定的精确度。

3.5 循环展开

在行主序存储下，通过循环展开并同时执行多个列的乘加操作可以提高效率。具体做法是，选择A矩阵的相同行中的元素a0, a1, a2, a3，这些元素分别与B矩阵的对应列中的相同元素进行乘法操作，然后将结果累加到C矩阵的相应行上。这种方式允许同时处理多个C矩阵的列，从而提高性能。下面是具体实现的代码示例，但要注意，对于列数不能被4整除的情况，还需要额外处理剩余的列。

```
void matrixMultiply_change_order_unrolled4(float *A, float *B, float *C, int m, int n, int k) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            float a = A[i * n + j];
            for (int x = 0; x < (k / 4) * 4; x += 4) {
                // 主循环展开4次
                C[i * k + x] += a * B[j * k + x];
                C[i * k + x + 1] += a * B[j * k + x + 1];
                C[i * k + x + 2] += a * B[j * k + x + 2];
                C[i * k + x + 3] += a * B[j * k + x + 3];
            }
            // 处理剩余元素
            for (int x = (k / 4) * 4; x < k; x++) {
                C[i * k + x] += a * B[j * k + x];
            }
        }
    }
}
```

3.6 Intel MKL

通过使用mkl_malloc来为三个矩阵分配对齐的内存空间，从而实现了对MKL性能的优化。接下来，采用cblas_dgemm函数来进行矩阵的乘法操作。这里，CblasRowMajor用于指明数组的存储顺序为行主序，而CblasNoTrans则表明不对任何矩阵进行转置操作。通过将alpha和beta参数分别设定为1.0和0.0，实现了C矩阵等于A矩阵与B矩阵乘积的计算。具体的核心代码段展示了这一过程。

```
double *A, *B, *C;
A = (double *)mkl_malloc( m*k*sizeof( double ), 64 );
B = (double *)mkl_malloc( k*n*sizeof( double ), 64 );
C = (double *)mkl_malloc( m*n*sizeof( double ), 64 );

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

4. 实验结果

4.1 Python

运行结果：

```
PP0 ) python3 matrix_mul.py
Matrix multiplication took 447.659 seconds.
GFLOPS: 0.002
```

Python的运行效率相对较低主要是因为它是一种解释型语言，具有动态类型系统，这意味着代码在运行时逐行转换为机器码且变量类型的检查也在运行时进行，增加了额外的开销。此外，Python的标准实现使用了全局解释器锁（GIL），限制了其在多核CPU上的并行执行能力。高级抽象和自动内存管理等特性虽然提高了开发效率，但也隐藏了底层的复杂性，从而增加了运行时的开销。这些因素共同作用，导致了Python相对于某些编译型语言在运行效率上的不足。

4.2 Naive

运行结果：

```
PP0 ) g++ matrix_mul.cpp -o matrix_mul.out
PP0 ) ./matrix_mul.out
Matrix multiplication took 4.09568 seconds.
GFLOPS: 0.524329
```

C++的运行效率高主要是因为它是一种编译型语言，其代码在程序运行前就被编译成了直接由硬件执行的机器码，消除了运行时转换的开销。C++提供了低级的编程能力，允许开发者直接管理内存和系统资源，这种直接控制硬件的能力减少了额外的抽象层，从而提高了效率。

4.3 Change Order

运行结果：

```
PP0 ) g++ matrix_mul.cpp -o matrix_mul_change_order.out
PP0 ) ./matrix_mul_change_order.out
Matrix multiplication took 2.78495 seconds.
GFLOPS: 0.771104
```

通过交换i和j维度的循环顺序，以利用数据的空间局部性，提高数据复用率，从而提升运行效率。

4.4 编译优化

运行结果：

```
PP0 ) g++ matrix_mul.cpp -O3 -fomit-frame-pointer -march=native -ffast-math -o
matrix_mul.out
PP0 ) ./matrix_mul.out
Matrix multiplication took 0.146415 seconds.
GFLOPS: 14.6671
```

经过编译优化后，运行效率大大提升。

4.5 循环展开

```
PP0 ) g++ matrix_mul.cpp -O3 -fomit-frame-pointer -march=native -ffast-math -o
matrix_mul_change_order_unrolled4.out
PP0 ) ./matrix_mul_change_order_unrolled4.out
Matrix multiplication took 0.143962 seconds.
GFLOPS: 14.917
```

可以看到，循环展开后，运行效率略微相较于change order有些许提升。

4.6 Intel MKL

```
xiaoma@xiaoma-virtual-machine:~/桌面/Parallel-Programming/PP0$ g++ -o intel_mm.out
intel_mm.cpp -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread -lm -ldl
xiaoma@xiaoma-virtual-machine:~/桌面/Parallel-Programming/PP0$ ./intel_mm.out
Matrix multiplication took 0.059867 seconds.
GFLOPS: 35.8709
```

可以看到，Intel MKL令运行效率大大提升。只需0.0598s即可完成[1024, 1024]方阵的矩阵相乘。同时，GFLOPS也高达35.8709。

4.7 表格对比

版本	实现描述	运行时间	相对加速比	绝对加速比	GFLOPS	峰值性能百分比
1	Python	447.659s	1.0	1.0	0.002	0.0009765625%
2	C/C++	4.09568s	262.1645	262.1645	0.524329	0.2560200195%
3	调整循环顺序	2.78495s	1.470649	385.552	0.771104	0.3765156249%
4	编译优化	0.14641s	19.02091	7333.549	14.6671	7.1616699219%
5	循环展开	0.14396s	1.017038	7458.5	14.917	7.2836914062%
6	Intel MKL	0.05986s	2.404699	17935.45	35.8709	17.515087890%

这个表格详细比较了六种不同实现方法对于串行矩阵乘法性能的影响。我们可以逐项分析：

1. **Python实现**：
   - 执行时间最长（447.659秒），性能最低（GFLOPS为0.002）。
   - 这是因为Python作为一种解释型语言，执行时的开销比编译型语言大；它还进行动态类型检查，增加了额外的运行时负担。
2. **C/C++实现**：
   - 显著减少了执行时间至4.09568秒，GFLOPS提升到0.524329。
   - C/C++是编译型语言，能够更直接地管理内存和执行，这就减少了运行时的转换和检查，导致性能大幅提升。
   - C/C++的直接内存访问和控制意味着可以进行更精细的性能优化。
3. **调整循环顺序的实现**：
   - 执行时间进一步降低至2.78495秒，GFLOPS提升到0.771104。
   - 这种优化利用了数据的空间局部性，通过优化数据访问模式以增加缓存命中率，提高效率。
4. **编译优化**：
   - 采用编译优化后，执行时间缩短至0.146415秒，GFLOPS急剧上升至14.6671。
   - 编译器优化如`-O3`、`-march=native`等，通过自动向量化等手段极大地提升了代码效率。
5. **循环展开**：
   - 执行时间略微减少至0.143962秒，GFLOPS提升至14.917。
   - 循环展开通过减少循环次数和增加每次循环的工作量，降低了循环开销，实现了轻微的性能提升。
6. **Intel MKL库**：
   - 执行时间最短，仅为0.05986秒，GFLOPS最高，达到了35.8709。

- MKL库是专为性能优化而设计的数学库，提供了高度优化的数学函数，特别是矩阵操作，能够充分利用硬件的特性，如SIMD指令集。

总体来说，每一种优化都在不同程度上提高了性能。这从GFLOPS的提升和执行时间的减少可以看出。从编程语言层面到编译优化，再到算法级别的调整和专门的数学库，都是提升性能的重要因素。尤其值得注意的是，硬件加速（通过MKL）提供了巨大的性能提升，这表明在现代高性能计算中，利用专业的库和硬件特性是非常关键的。

5 实验感想

在这次的并行程序设计与算法实验中，我深入探讨了串行矩阵乘法的多种实现方式，并对比了它们在性能上的差异。通过本实验，我不仅加深了对矩阵乘法的理解，还学习了如何通过不同的编程技术和优化策略来提高程序的执行效率。

首先，我体会到了编程语言选择对性能的重大影响。比如，用Python实现的矩阵乘法，因其解释性质和动态类型系统，其性能远不如用C++实现的版本。这一点在实验结果中表现得非常明显，Python版本的运行时间远长于C++版本，这让我意识到在处理计算密集型任务时，选择合适的编程语言是多么重要。

此外，我还学习到了多种代码优化技术，如循环展开、编译器优化标志的使用，以及更改循环顺序以利用数据的空间局部性。这些技术都在不同程度上提升了程序的运行效率。特别是编译优化和循环展开，它们让我了解到编译器的强大以及编写高效代码时需要考虑的底层细节。

通过本实验，我还了解到了现代软件开发中常用的一些工具和库，比如Intel MKL。使用这样的库可以极大地简化开发过程，并利用库作者的专业知识来提升程序的性能，这对我来说是一个非常宝贵的学习经验。

总的来说，这次实验不仅仅是对串行矩阵乘法算法的一次实践，更是一次深刻的学习和成长经历。它让我认识到了在软件开发中，理论知识与实践技巧的重要性，以及持续探索和学习新技术的必要性。我期待将这次实验中学到的知识应用到未来的项目中，继续提升自己的技术能力。