

# 《多核软件设计》实验手册

（2023年11月版）

**郑重声明：**

**1、实验手册中的所有实验均有本人独立编码、调试和测试。**

**2、实验手册中给出的实验数据和结果完全由本人所完成的程序给出。**

**3、本人了解：不按照前两条要求所完成的实验报告已经构成了抄袭或造假行为，本人将承担相应的不良后果。**

姓名： （签名） 学号：

提交日期： 2023年12月16日 总成绩：

本课程以设计竞赛方式评分，执行速度排名与分数的比例如下表。

|  |  |  |  |
| --- | --- | --- | --- |
| 执行速度排名 | 分数 | 执行速度排名 | 分数 |
| 前10%（第1名） | 100% | 前10～30%（第2~4名） | 85% |
| 前40～60%（第5~8名） | 70% | 前60～85%（第9~10名） | 60% |
| 最后15%或程序执行崩溃，或结果不正确（第11名） | | | 50% |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 题号 | 占分 | 程序执行时间(s) | 输入参数 | 运行结果 | 排名 | 得分 |
| 1 | 30 |  |  |  |  |  |
| 2 | 30 |  |  |  |  |  |
| 自主报告 | 20 |  | | | |  |
| 实验报告 | 10 |  | | | |  |
| 平时作业 | 10 |  | | | |  |
| 总分 | | | | | |  | |

指导教师： （签名）

本实验所使用的CPU平台为：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CPU | | | Cache | | | 操作系统 | 编译器 | |
| 型号 | 核数 | 主频 | L1 | L2 | L3 | 版本 | 优化参数 |
|  |  |  |  |  |  |  |  |  |

## 1. 矩阵乘法（30分）

实现单精度浮点的矩阵乘法：，并求矩阵C的迹Trace。

**程序输入输出要求**

Matrix\_mul *N seed*

参数：

*N:* 矩阵大小

*seed*：输入的种子浮点数

输出格式：

Trace

Trace: 结果矩阵C的迹（），用于验证矩阵乘法的正确性。

矩阵产生函数

/\*

Input: a, b are the N\*N float matrix, 0<seed<1, float

This function should initialize two matrixs with rand\_float()

\*/

float rand\_float(float s){

return 4\*s\*(1-s);

}

void matrix\_gen(float \*a,float \*b,int N,float seed){

float s=seed;

for(int i=0;i<N\*N;i++){

s=rand\_float(s);

a[i]=s;

s=rand\_float(s);

b[i]=s;

}

}

1. 【基准程序】

基准矩阵乘法程序如下所示。

//基准矩阵乘法，a,b为输入矩阵的指针，c为输出矩阵的指针，N为矩阵的阶数

void matrix\_multiply(float \*a,float \*b, float \*c, int N){

int i,j,k;

for(i=0;i<N;i++){

for(j=0;j<N;j++){

float sum=0.0;

for(k=0;k<N;k++){

sum+=a[i\*N+k]\*b[k\*N+j]; //矩阵元素访问

}

c[i\*N+j]=sum;

}

}

}

调整矩阵规模，观察不同矩阵规模下串行程序执行时间的变化。

表2-1 不同规模矩阵的串行计算时间

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N* | 512 | 1024 | 2048 | 4096 |
| Trace |  |  |  |  |
| 计算时间(s) |  |  |  |  |

根据表2-1绘制图2-1，其中X轴为N，Y轴为矩阵规模为*N*时的计算时间。

图2-1 不同矩阵规模的计算时间

从图2-1中，可以得到的结论是 。

2.【矩阵分块计算】

可以将矩阵分解为*m*×*m*的矩阵小块，每次完成一对小块的计算，以提高Cache的命中率。如下式所示。

，其中*n=N/m*

L1和L2 Cache容量分别是 KB和 KB。你认为*m*设置为 比较合适（记为*m*0），原因是 。在最优的线程数下，调整*m*的大小，得到表2-3。

表2-2 不同分块大小的计算时间和加速比（*N*=8192）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 分块大小(*m*) | 不分块 | *m*0/2 | *m*0 | 2*m*0 |
| 计算时间(s) | (第1节中的时间) |  |  |  |
| 加速比 | 1.00 |  |  |  |

实验表明，分块大小为 时性能最好。这个结果和你的预期一致吗？ 。

如果不一致，其原因在于 。

3.【使用SIMD指令加速】

使用SSE指令可以一次完成4个浮点乘法操作和加法操作。下图给出了4×4的矩阵乘法操作示意图。可以使用<https://github.com/tommychen74/SIMDLib>提供的内嵌原语实现SSE或者AVX指令。



根据同样的方法，使用AVX指令可以完成8路矩阵的乘法。

结合上述矩阵分块策略，在最优的线程数下，最优的分块数下，使用SSE/AVX指令的性能如表2-4所示。

表2-3 使用SIMD指令的计算时间和加速比

|  |  |  |  |
| --- | --- | --- | --- |
| SIMD指令 | 不使用 | 使用SSE | 使用AVX |
| 计算时间(s) | (第2节中的最优时间) |  |  |
| 加速比 | 1.00 |  |  |

当前最优的执行时间是 s，与第1节中的串行计算相比，加速比达到 。

4.【多线程并行化】

使用多线程并行方法计算矩阵乘法（*N*=4096）。测试不同线程数下，矩阵计算的执行时间。有两种矩阵划分方式，如下所示。你认为哪种方法更好？对比实验的效果如何？





表2-4 不同线程数的计算时间和加速比（*N*=4096）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 线程数(T) | 1 | 2 | 4 | 8 | 16 |
| 计算时间(s) |  |  |  |  |  |
| 加速比 | 1.00 |  |  |  |  |

根据表2-2绘制图2-2，其中X轴为T，Y轴为计算时间。

图2-2 不同线程数的计算时间（*N*=4096）

实验平台的处理器核数是 。当线程数(T)为 时，性能最好，加速比达到 。

5.【性能对比】

画出以上串行、分块、分块+SIMD、分块+SIMD+多线程并行等4种实现策略在*N*=4096的情况下的程序执行时间对比图。

从理论上，矩阵乘法中总共需要 次浮点乘法和加法操作。比较上述6种算法的计算性能（单位GFlops）。

表2-5 4种矩阵乘法的性能比较

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 执行方法 | 串行 | 分块 | 分块+SIMD | 分块+SIMD+多线程 |
| 执行时间(s) |  |  |  |  |
| 加速比 | 1.00 |  |  |  |
| 性能(GFlops) |  |  |  |  |

从表2-5中，你认为哪种并行化策略最为有效？ 。原因是 。

6. 【Strassen方法】

Strassen给出了一个减少矩阵乘法计算量的方法。



按照以下方式计算：

*P1=*(*A11+A22*)(*B11+B22*)

*P2=*(*A21+A22*)*B11*

*P3=A11* (*B12-B22*)

*P4=A22*(*B21-B11*)

*P5=*(*A11+A12*) *B22*

*P6=*(*A21-A11*)(*B11+B12*)

*P7=*(*A12-A22*)(*B21+B22*)

*C11=P1+P4-P5+P7*

*C12=P3+P5*

*C21=P2+P4*

*C22=P1+P3-P2+P6*

可否使用这个方法进一步提高矩阵乘法的性能？如果可以，请说明你的方案以及实现效果。

## 2. 字符文件分析（30分）

输入为一个由ASCII构成的文件file。此文件每行的结尾均为0x0A。文件最后一个字符为0x0A。

输出为该文件的行数、出现次数最多的一行长度，以及对应的行数。如果多个长度满足的行数均相同，且高于其它长度，则显示所有的最长长度和对应行数。

**程序输入**

**filelines** *filepath*

**filelines**：程序名

*filepath:* 输入文件file的路径

**输出格式**

linenum most\_freq\_len most\_freq\_len\_linenum

linenum:文件的总行数

most\_freq\_len: 出现次数最多的一行长度

most\_freq\_len\_linemum：对应的行数

当输入文件为下述内容时：

1

345

123

abcd

abc

abcde

789

输出的结果为：

7 3 4

即此文件共7行，最常出现的行长度为3个字符，3个字符的行数共计4行。

源代码说明：

filelines.cpp 测试程序入口

filelines\_baseline.cpp filelines\_baseline.h 第1节基准程序

find\_most\_freq.cpp find\_most\_freq.h 结果统计程序

filelines\_gen.cpp 产生测试文件程序

Makefile 编译makefile文件

1. 【按块读取文件】

输入文件 ，大小为 MB。

从文件中以*S* KB为单位读取一块，然后对一块中的内容进行统计。下表为*S*设置为不同值时程序执行时间。

|  |  |
| --- | --- |
| *S* (Bytes) | 执行时间 |
| 64 |  |
| 256 |  |
| 1K |  |
| 4K |  |
| 16K |  |
| 256K |  |

在*S*为 时，程序执行时间最短，为 s，程序的吞吐率为 B/s。

2. 【使用SIMD方法统计】

以上节最快的*S*为分块大小。参照《软件优化技术》中程序示例3.24，将一个块内的统计过程修改为SIMD指令以加速统计过程。

使用了 （SSE或AVX）的SIMD指令系统时，程序的执行时间为 s，程序的吞吐率为 MB/s

3. 【读取文件和计算过程并行】

在上节的基础上使用生产者-消费者模型，一个线程用于读取文件块，一个线程用户统计过程。此时程序的执行时间为 s，程序的吞吐率为 MB/s

4. 【比较和分析】

下表对比了4种实现方案的吞吐率。

|  |  |
| --- | --- |
| 方案 | 吞吐率（MB/s） |
| 基准程序 |  |
| 文件分块读入 |  |
| 使用SIMD方法统计 |  |
| 读取文件和计算过程并行 |  |

测试平台的硬盘型号是 。查找手册或实际测试，可以得到测试平台的硬盘读出吞吐率为 MB/s。

请问答：

1）该程序的瓶颈是文件读取还是计算过程？为什么？

2）你的程序是否已经达到了系统的性能极限？为什么？

## 3. 自主报告（20分）

自主报告将选择一个本人有兴趣或者实验室课题相关的**单节点上程序优化问题**，作为研究的主要内容。报告的主题是开源软件性能评测与分析。

“开源性能评测与分析”主题，需要完成以下工作：

1）问题描述；（2分）

2）选择解决同一问题的2种或以上的开源软件，并在自有硬件平台上部署；（3分）

3）选择测试基准数据；（3分）

4）测试和比较基准数据在不同开源软件源代码上的性能，找到性能最优的开源软件；（2分）

5）说明性能最优软件在线程级、数据级、Cache等方面的优化方法；（8分）

6）参考文献；（2分）

报告完成时间：2024年11月17日（第12周周日）