南開大學
Nankai University

计算机学院

并行程序设计实验报告

矩阵向量内积与 n 个数求和问题研究

姓名：卢艺晗

学号：2213583

专业：计算机科学与技术

2024 年 3 月 27 日

目录

1 实验平台信息

指标	值
CPU 型号	Intel® Core™ i7-14650HX
CPU 核数	16
CPU 主频	2.20GHz
L1/L2/L3 大小	1.4MB/24.0MB/30.0MB
内存容量	32.0GB
操作系统及版本	windows11 家庭中文版
编译器及版本	Code::Blocks 20.03 gcc 8.1.0

表 1: 实验平台信息

源码 GitHub 链接： GitHub/Lucyannn/Parallel Design

2 基础要求

2.1 实验设计

2.1.1 n*n 矩阵与向量内积

（1）算法设计思路

问题设计平凡算法和 cache 优化算法进行对比，以探究如何利用 cache 优化程序性能。平凡算法逐列访问矩阵元素，一步外层循环（内存循环一次完整执行）计算出一个内积结果；cache 优化算法则改为逐行访问矩阵元素，一步外层循环计算不出任何一个内积，只是向每个内积累加一个乘法结果。

（2）测试数据及问题规模

测试数据采用人为生成，对于 n*n 矩阵中的每一个元素 b[i][j], 令 b[i][j]=i+j; 对 n 维向量 a 中的每个元素 a[i]=i+1。

问题规模对应三个缓存的大小，根据我的 cpu 三级缓存大小，经计算，对应 L1 临界时的 n 大小在 200 前后，对应 L2 临界时的 n 大小在 1800 前后，对应 L3 临界时的 n 大小在 3000 前后。故设计三组（每组十个）问题规模，分布在三个临界值附近, 详见“性能测试”部分的表格。

（3）观测指标

对于三组问题规模，分别绘制算法执行时间与问题规模相关的折线图，对比应用到不同缓存级别时，两种算法的性能差异。

2.1.2 n 个数求和

（1）算法设计思路

问题设计平凡算法和 cache 优化算法进行对比，以探究如何利用 cache 优化程序性能。平凡算法采用链式方法，对数组元素逐个累加；cache 优化算法则采用两路链式累加。

（2）测试数据及问题规模

测试数据采用人为生成，对于数组 a[n] 中的每一个元素，a[i]=i*i。

根据三个缓存的大小，设置问题规模在 2 的 5 次方到 2 的 25 次方，测试并分析三级缓存对于算法性能的影响。详见“性能测试”部分的表格。

（3）观测指标

统计两种算法在不同问题规模下各自的执行时间，并计算优化算法相对平凡算法的加速比，分析缓存对性能差异的影响。

2.2 编程实现

2.2.1 n*n 矩阵与向量内积

(1) 平凡算法（仅展示关键代码）

n*n 矩阵与向量内积平凡算法

```
    long long head=0.0, tail=0.0 , freq=0.0 ;
    QueryPerformanceFrequency((LARGE_INTEGER*)&freq );
    QueryPerformanceCounter((LARGE_INTEGER*)&head);
    for(int t=1;t<=count;t++){          //多次运行取平均值
    for ( int i = 0; i < N; i++) {
    sum[ i ] = 0.0;
    for ( int j = 0; j < N; j++)
    sum[ i ] += b[ j ][ i ]*a[j];
    }
    }
   QueryPerformanceCounter((LARGE_INTEGER*)&tail);
   cout<<(tail-head)*1000.0/freq/count<<" ms"<<endl;//计算平均每次执行时间
```

(2)cache 优化算法（仅展示关键代码）

n*n 矩阵与向量内积 cache 优化算法

```
    long long head=0.0, tail=0.0 , freq=0.0 ;
    QueryPerformanceFrequency((LARGE_INTEGER*)&freq ) ;
    QueryPerformanceCounter((LARGE_INTEGER*)&head) ;
    for(int t=1;t<=count;t++){          //多次运行取平均值
    for (int i = 0;i< N;i++) {
    sum[ i ] = 0.0;
    for (int j = 0;j< N;j++)
    for( i=0;i<N;i++)
    sum[ i ] += b[ j ][ i ]*a[j];
    }
    }
   QueryPerformanceCounter((LARGE_INTEGER*)&tail);
   cout<<(tail-head)*1000.0/freq/count<<" ms"<<endl;   //计算平均每次执行时间
```

2.2.2 n 个数求和

(1) 平凡算法（仅展示关键代码）

n 个数求和平凡算法

```
    long long head=0.0, tail=0.0 , freq=0.0 ;
    QueryPerformanceFrequency((LARGE_INTEGER*)&freq ) ;
    QueryPerformanceCounter((LARGE_INTEGER*)&head) ;
```

```
    for(int t=1;t<=count;t++){          //多次运行取平均值
    for(int i=0;i<N;i++){
    sum[i]+=a[i];}
    }
    QueryPerformanceCounter((LARGE_INTEGER*)&tail);
    cout<<(tail-head)*1000.0/freq/count<<" ms"<<endl;
```

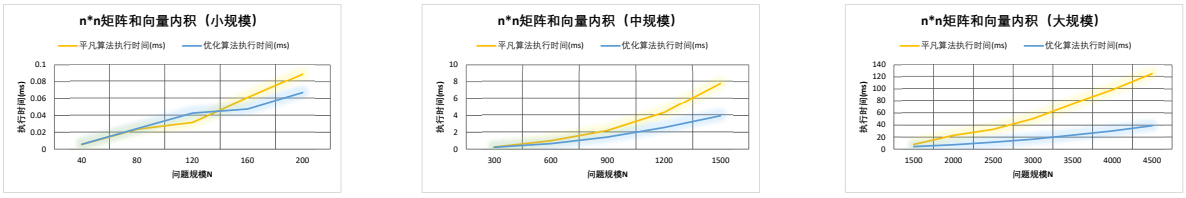
(2)cache 优化算法（仅展示关键代码）

n 个数求和 cache 优化算法

```
    long long head=0.0, tail=0.0 , freq=0.0 ;
    double sum1 = 0.0, sum2 = 0.0;
    QueryPerformanceFrequency((LARGE_INTEGER*)&freq ) ;
    QueryPerformanceCounter((LARGE_INTEGER*)&head) ;
    for(int t=1;t<=count;t++){          //多次运行取平均值
    for(int i=0;i<N;i+=2){
    sum1 += a[i];
    sum2 += a[i + 1];}
    }
    sum = sum1 + sum2;
    QueryPerformanceCounter((LARGE_INTEGER*)&tail);
    cout<<(tail-head)*1000.0/freq/count<<" ms"<<endl;
```

2.3 性能测试

2.3.1 n*n 矩阵与向量内积



(a) 小规模下两种算法执行时间比较. (b) 中规模下两种算法执行时间比较. (c) 大规模下两种算法执行时间比较.

图 2.1: n*n 矩阵与向量内积两种算法性能对比

问题规模（N）	执行次数	平凡算法平均执行时间（ms）	优化算法平均执行时间（ms）
小规模			
40	100	0.005429	0.005755
80	100	0.023387	0.024393
120	100	0.031299	0.042651
160	100	0.060696	0.047615
200	100	0.088799	0.067109
中规模			
300	30	0.234117	0.22201
600	30	0.972267	0.65006
900	30	2.17153	1.4322
1200	30	4.27977	2.56144
1500	30	7.78676	3.95768
大规模			
1500	10	7.84772	4.23502
2000	10	23.0312	7.19726
2500	10	32.9726	11.5355
3000	10	50.5607	16.3872
3500	10	75.1869	23.2514
4000	10	98.6189	30.1086
4500	10	125.427	38.8909

表 2: n*n 矩阵与向量内积测试数据记录

2.3.2 n 个数求和

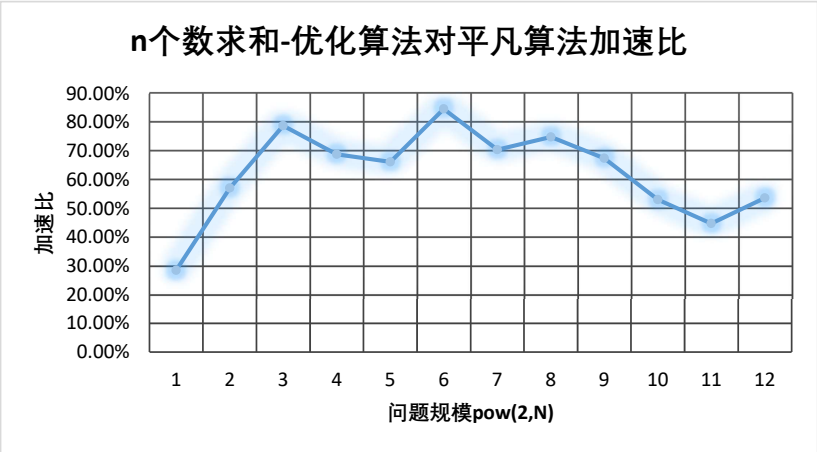


图 2.2: 不同问题规模下优化算法对平凡算法加速比

问题规模 (pow(2,N))	执行次数	平凡算法平均执行时间 (ms)	优化算法平均执行时间 (ms)	加速比
2	20	1.60E-05	1.60E-05	28.50%
4	12	5.83E-05	2.50E-05	57.10%
6	8	0.000238	5.00E-05	78.90%
8	6	0.00075	0.000233	68.90%
10	5	0.00296	0.001	66.20%
12	4	0.036225	0.00555	84.70%
14	3	0.049333	0.014567	70.50%
16	3	0.233633	0.058667	74.90%
18	2	0.7096	0.2307	67.50%
20	2	1.97825	0.9285	53.10%
22	2	8.1186	4.47795	44.80%
24	2	32.5417	15.0787	53.70%

表 3: n 个数求和两种算法测试数据记录

2.4 结果分析

2.4.1 n*n 矩阵与向量内积

由于计算机中存储数组为同一行元素相邻存放，优化算法为逐行访问，很好地符合了内存中这种规律，降低访存开销，相对平凡算法的逐列访问具有更好的性能，具体分析如下：

如图 2.1(a) 所示，在小规模问题中，在 N<=160 前，数据能主要存储在 cache L1 中，L1 访存效率很高，因此优化算法和平凡算法的执行时间相差不大，甚至偶然时可能因为数据实际分布情况导致优化算法执行时间略长。当 N>60 后，超出 L1 的容量，部分需要访问 L2，优化算法开始显现优势；

如图 2.1(b) 所示，在中规模问题中，访存主要在 L1 和 L2。数据规模越大，L2 访存率越高，优化算法由于降低访存开销，相对平凡算法的优势也越大。

如图 2.1(c) 所示，在大规模问题中，N<=3000 之前，访存主要涉及 L1,L2,L3，由于 L3 的访存效率相对较低，因此两种平凡算法的执行时间相对优化算法已经将近翻了一番。当 N 超过 3000 以后，缓存容量不够，内存的访存率升高，由于内存的访存效率远低于缓存，优化算法的性能开始远优于平凡算法，如图中 N=4500 处数据所示，优化算法的执行时间达到了平凡算法的近 1/3。

综上，优化算法的访存模式具有很好地空间局部性，降低访存开销，因此应用到较远地缓存或内存时，优化算法的优势更加明显。当数据规模巨大时，使用优化算法提升性能十分重要。

2.4.2 n 个数求和

该问题的优化算法采用超标量优化。相对于平凡算法的单链式，这种双链式的算法采用以空间换时间的策略，从而提高性能。但是由于对硬件资源的更多消耗，它的加速效果并没有呈线性增长趋势。

如图 2.2 所示，随着问题规模的增大，优化算法对平凡算法的加速比呈现先增大后降低的趋势。当规模较小时，访存主要涉及 L1（及 L2），访存效率本身相对较高，因此双链式相对单链式的性能提升仅在 20% 左右。当规模增大，访存涉及了 L3，访存效率相对较低，双链式两链同时进行就显现出巨大优势，最高达约 84.70%。但是当问题规模再扩大，可能由于超标量优化本身消耗空间资源较多的原因，访问内存比例上升，使得加速比受限在 50% 上下。整体来看，超标量优化还是对性能起到了很大提升作用。

3 进阶要求

3.1 使用 COMPILER EXPLORER 平台对比不同编译器导致性能差异的原因

3.1.1 设计思路

对“n 个数求和”问题的两种算法在不同编译器上对比。分别进行一次横向对比和一次纵向对比。横向对比是对比两个不同的编译器最新版本的比较（x86 msvc v19.latest 对比 x86-64 gcc 13.2）；纵向对比是同一编译器不同版本的比较（x86-64 gcc 13.2 对比 x86-64 gcc 9.5），从而深入探究性能差异的原因。

3.1.2 测试代码

n 个数求和平凡算法

```
int main(){
int a[10240];
int sum=0;
for (int i = 0;i < 10240;i += 2) {
sum+=a[i];  }
return 0;   }
```

n 个数求和优化算法

```
int main(){
int a[10240];
int sum=0,sum1 = 0, sum2 = 0;
for (int i = 0;i < 10240;i += 2) {
sum1 += a[i];
sum2 += a[i + 1];    }
sum = sum1 + sum2;
return 0;   }
```

3.1.3 汇编代码和性能差异分析

（1）不同编译器上体现出的共性

根据汇编相关知识，整数的加减算术运算等指令开销相对较小，而条件跳转、涉及内存访存的数据传输指令等开销相对较大，效率较低。据此可以从汇编代码中分析两种算法性能差异的原因。

在不同的编译器上，共性是 cache 优化算法的循环次数减半，而每次循环的步长为 2，计算密度提高。计算密度的提高有助于提高处理器执行单元的利用率，循环次数的减半使得条件跳转这种开销相对较大的指令数量降低，也在一定程度上提高了运行效率。

（2）不同编译器上体现出的差异

纵向对比 gcc 的两个版本，gcc 9.5 和 gcc 13.2 主要在两处有差异，一是跳转指令不同，二是 gcc 9.5 的输出包含了静态初始化和析构代码，尽管与代码逻辑无关，但是在程序启动时会产生轻微的开销。可能由于本测试案例程序相对简单，两个版本的汇编代码并未在逻辑上产生过多的不同。综合 gcc9.5 和 gcc13.2 的整体表现来看，更新版本的编译器通过使用更好的循环优化（如循环向量化）、支持 SIMD 等现代并行式指令集来提高程序性能。

横向对比 gcc 13.2 和 mscv 19.latest 两个版本，它们编译得到的汇编代码量有很大差异前者的汇编代码 34 行，后者 58 行。虽然最新版本的编译器都能支持更好的循环优化、使用内联函数来减少函数调用开销、更好地利用现代 cpu 特性，但是从汇编代码中能够发现，它们在实现这些时的具体策略和细节有所不同。综上导致的缓存利用、指令调用等差异最终导致了同一算法在不同编译器上也会表现出不同的性能。

4 实验总结和思考

4.1 对比两个实验的异同

（1）相同点

两个实验都通过 cache 优化算法，从访存角度研究提升算法性能的方法，且取得显著成效。在实验设计上，都设置不同规模的问题来细致观察和分析各级缓存对于程序效率的影响，并通过多次实验取平均值的方法降低偶然误差的影响。

（2）不同点

n*n 矩阵与向量内积的 cache 优化是利用矩阵在内存中同行相邻，以及现代 cpu 常有的整行数据预读取的优势，减小更换访地址距离的方式来优化的；n 个数求和的 cache 优化算法是利用超标量优化，采用双链式将求和分成了两个相对独立的过程并行执行，从而提高了程序整体的执行时间。

4.2 心得体会

通过此次实验，我明白了以下几点：

（1）通过优化算法来提高性能，在逻辑设计时要基于计算机硬件的一些特性（例如更好地利用缓存）来设计程序执行的具体思路；

（2）在程序编写时，可以通过降低循环次数、提高运算密度等方法，发挥汇编不同指令效率不同的特点；

（3）程序实际执行的性能与平台也有着密切关系。不同的编译器意味着不同的算法优化策略、不同的指令集和不同的代码量等，这些都从方方面面影响着程序的性能。