南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计

体系结构相关实验及性能测试

田翔宇 2011748

年级：2020 级

专业：计算机科学与技术

2023 年 3 月 12 日

摘要

对于给定的矩阵乘法和数组求和两个问题，分别考虑采用 cache 优化和超标量优化的方法对串行算法进行加速。在数组求和的实验中，考虑采用超标量的优化方法，即将求和问题转化为了多的变量同时累加最后再求和的方式。实验代码已上传 GitHub。

关键字：cache 优化超标量并行程序设计

目录

一、 问题重述

(一) 矩阵与向量内积

给定一个 n × n 矩阵，计算每一列与给定向量的内积，考虑两种算法设计思路：逐列访问元素的平凡算法和 cache 优化算法，进行实验对比：

1. 对两种思路的算法编程实现
2. 练习使用高精度计时测试程序执行时间，比较平凡算法和优化算法的性能

(二) n 个数求和

计算 n 个数的和，考虑两种算法设计思路：逐个累加的平凡算法（链式）；适合超标量架构的指令级并行算法（相邻指令无依赖），如最简单的两路链式累加，再如递归算法——两两相加、中间结果再两两相加，依次类推，直至只剩下最终结果。完成如下作业：

1. 对两种算法思路编程实现
2. 练习使用高精度计时测试程序执行时间，比较平凡算法和优化算法的性能

二、 实验环境

环境如下：

- 平台：x86
- CPU 型号：Intel(R) Core(TM) i7-10710U
- CPU 频率：1.10GHz
- L1 cache：384KB
- L2 cache：1.5MB
- L3 cache：12.0MB

三、 实验设计及分析

(一) 矩阵与向量内积

1. 实验设计

- 题目背景：矩阵在内存中存储时按照行有限的顺序存储的，即在内存中的矩阵是按行紧密排列的。
- 平凡逐列访问算法
  - CPU 会一次读入连续的一段数据到缓存中，而其中极有可能只包含需要计算一个元素。因此当计算该列的第二个元素的时候，CPU 又需要到更低的缓存或内存中去读取所需要的元素，而访存的时间相较于运算来说开销大很多，这会在很大程度上降低程序运行的效率。

平凡逐列访问算法

```
for (int l = 0; l < LOOP; l++)
{
        for (int i = 0; i < N; i++)
        {
                sum[i] = 0;
                for (int j = 0; j < N; j++)
                sum[i] += a[j] * b[j][i];
        }
}
```

- cache 优化算法
  - 充分利用每次读入的数据，将当前读入的缓存中的一行数据全部进行计算，然后累加到结果数组的对应位置，虽然在这个过程中并没有能够直接计算出结果，但是极大利用了 cache 中的缓存数据，减少去内存中寻找数据的访存时间。

cache 优化算法

```
for (int l = 0; l < LOOP; l++)
{
        for (int i = 0; i < N; i++)
        sum[i] = 0;
        for (int j = 0; j < N; j++)
        for (int i = 0; i < N; i++)
        sum[i] += a[j] * b[j][i];
}
```

- unroll 优化算法
  - 对于逐行访问的算法进行了进一步优化，采用循环展开的方法，在一次循环中，同时计算 2 个位置的值，可以利用多条流水线同时作业，能够降低循环访问过程中，条件判断，指令跳转等额外开销, 发挥 CPU 超标量计算的性能。

unroll 优化算法

```
for (int l = 0; l < LOOP; l++)
{
        for (int i = 0; i < N; i++)
        sum[i] = 0;
        for (int j = 0; j < N; j += 10)
        {
                int tmp0 = 0, ... tmp9 = 0;
                for (int i = 0; i < N; i++)
                {
                        tmp0 += a[j + 0] * b[j + 0][i];
                        //...
                        tmp9 += a[j + 9] * b[j + 9][i];
                }
                sum[j + 0] = tmp0;
```

```
15                          // ...
16                          sum[j + 9] = tmp9;
17                      }
18              }
```

2. 实验分析

为此，我分别设计了三种算法，并在 x86 平台上进行测试，实验测试数据如表1、2、3所示。

N	平凡	cache 优化	unroll
10	0.0005	0.0005	0.0005
20	0.0012	0.0013	0.0008
30	0.0021	0.0020	0.0018
40	0.0037	0.0035	0.0025
50	0.0055	0.0052	0.0038
60	0.0080	0.0076	0.0059
70	0.0113	0.0098	0.0071
80	0.0135	0.0127	0.0093
90	0.017	0.0159	0.0115

表 1: N:10-90

N	平凡	cache 优化	unroll
100	0.0218	0.0196	0.0141
200	0.1092	0.0743	0.0535
300	0.1945	0.1659	0.1193
400	0.3468	0.2923	0.2145
500	0.6207	0.4543	0.3310
600	0.8854	0.6753	0.5010
700	1.1316	0.9737	0.6832
800	1.6017	1.2705	0.9732
900	2.7184	1.5482	1.1949

表 2: N:100-900

数学分析：

- N<200 ： 平凡算法或 cache 优化的访问方式在效率上差别不大，但采用循环展开的优化算法能够取得较大的性能提升
- 200<N<2000 ： 三种方法时间随 N 的规模增长趋势是基本一致
- N>2000 ： 平凡算法的时间迅速增大，比例超过 cache 优化，cache 优化效果明显

理论分析：

N	平凡	cache 优化	unroll
1000	3.0625	1.8898	1.4355
2000	27.9892	7.4019	5.4379
3000	77.3183	16.7841	12.6252
4000	132.119	29.6537	22.9701
5000	230.523	46.3494	34.0457
6000	328.735	66.3655	48.3634
7000	476.718	90.6296	65.3894
8000	687.153	118.433	86.1058
9000	829.281	148.209	107.635

表 3: N:1000-9000

- CPU 的 L1-L3 的各级 cache 大小分别为 384KB，1.5MB 和 12.0MB，数组元素 unsigned long long int 占据 8 个字节，又考虑到 cache 不会只分配给一个程序使用，故猜测转折点 80、200、2000 处 cache 填满。
- 进一步分析：
  - N<200：尽管普通方法的 L1 cache 命中率可能不高，但是由于 L2 cache 的访问速度相对比较快，所以访存速度对整体的时间影响不太大，即两种方法效率差别不大。
  - 200<N<2000：L2 cache 的命中率也会不断下降，使得平凡算法的方法被迫大更大的 L3 cache 中去寻找数据，这就导致了较大的访存开销，也可以看出两种方法的差距逐渐显现出来
  - N>2000：平凡算法的 L3 cache 命中率也会降到很低，迫到内存中去寻找数据，这导致的访存开销极大，因此也使得两种方法的访问效率产生了显著的差异

(二) n 个数求和

1. 实验设计

- 题目背景：求计算 N 个数的和。
- 平凡顺序算法
  - 每次都是在同一个累加变量上进行累加，导致只能调用 CPU 的一条流水线进行处理，无法充分发挥 CPU 超标量优化的性能。

平凡顺序算法

```
for (int l = 0; l < LOOP; l++)
{
        // init();
        ull sum = 0;
        for (int i = 0; i < N; i++)
        sum += a[i];
}
```

- 超标量优化算法
  - 设置多个临时变量，在一个循环内同时用着多个临时变量对多个不同的位置进行累加，达到多个位置并行累加的效果，同时还能够减少循环遍历的步长，降低循环开销。

超标量优化算法

```
for (int l = 0; l < LOOP; l++)
{
        ull sum1 = 0, sum2 = 0;
        for (int i = 0; i < N - 1; i += 2)
        sum1 += a[i], sum2 += a[i + 1];
        ull sum = sum1 + sum2;
}
```

2. 实验分析

由于该问题整体随 n 的增速比较慢，故考虑将所有问题规模都取成 2 的 n 次幂，且当问题规模较小的时候，两种算法并没有显著的时间效率差异，因此可以直接扩大了问题规模，分别测试了 n 从 9 到 28 之间的数据，如表4所示。

n	平凡	cache 优化
9	0.00081	0.00058
10	0.0016	0.00114
11	0.00317	0.00226
12	0.00641	0.0045
13	0.01296	0.01114
14	0.02591	0.01826
15	0.06294	0.0377
16	0.10828	0.07777
17	0.23611	0.14735
18	0.41927	0.34755
19	0.85508	0.60822
20	1.82409	1.31209
21	3.54668	2.69749
22	7.10291	5.02325
23	14.247	9.92355
24	28.1432	20.2914
25	56.5281	40.1385
26	113.289	80.4477
27	227.764	159.198
28	461.814	350.749

表 4: n:9-28

为了进一步探究超标量优化的加速比，我们计算了优化加速比同问题规模的变化情况，如图所示。

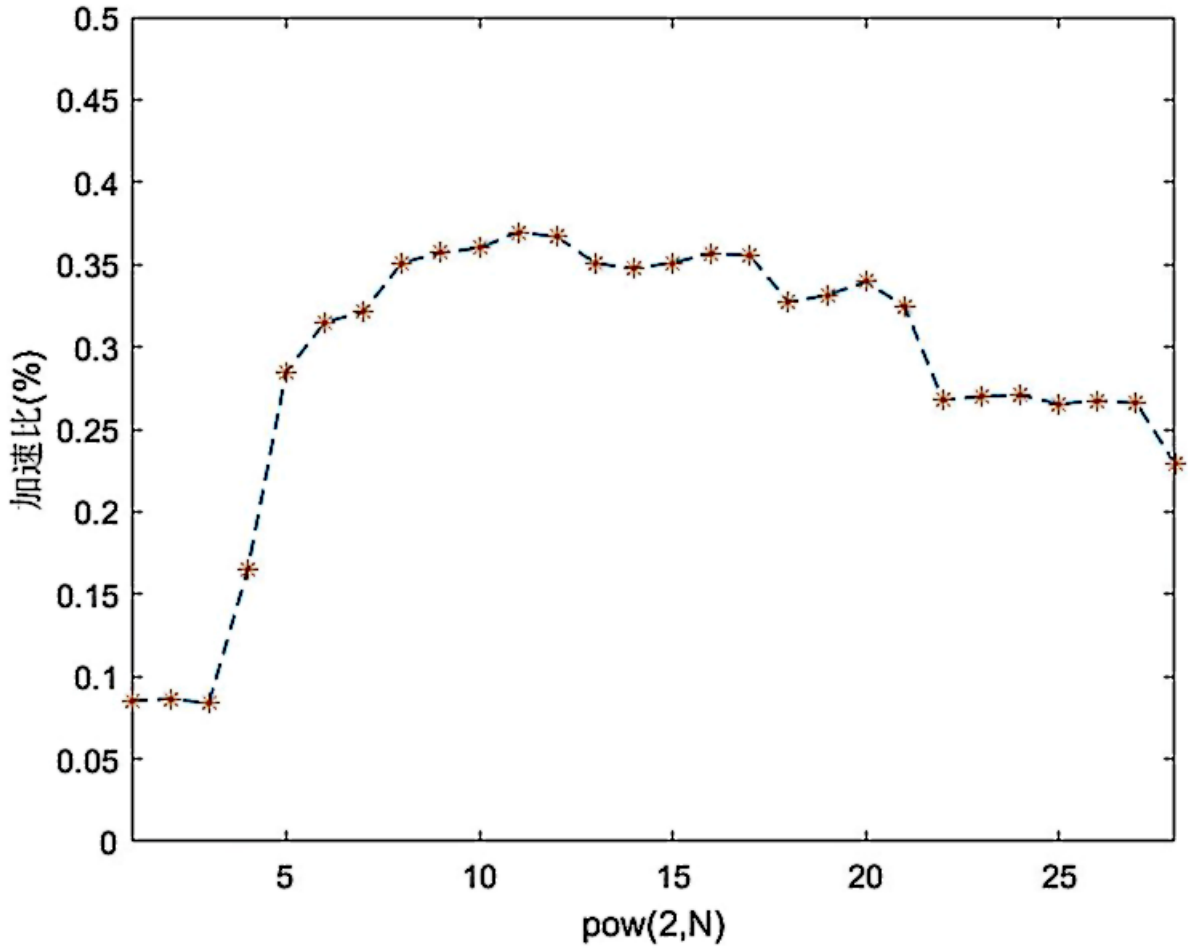

图 1: 加速比

数据分析：

- **随着问题规模的翻倍，两种算法的时间也近似翻倍，是线性时间效率**
- **双链路展开的超标量优化方法的时间效率要显著高于普通的链式累加方法**
  - **通过多链路的方法，将一个累加运算分成两个不相干的累加运算，可以并行处理两个累加，使得 CPU 能够同时调用两条流水线处理问题，实现超标量优化的目的**
- **而当问题规模超过 2 的 16 次幂后，整体呈现一个下降的趋势。原因是问题当问题规模大到一定程度时，缓存明显不足，需要对内存进行访问，导致了较大的访存开销，而这个访存开销占据了程序运行时间比重较大，所以超标量优化的效果被一定程度上减弱。**

参考文献