南開大學
Nankai University

计算机学院

并行程序设计作业报告

体系结构相关及性能测试

姓名：徐文斌

学号：2010234

专业：计算机科学与技术

2022 年 3 月 15 日

目录

1 cache 优化

1.1 实验介绍

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。

1.2 程序设计思路

1.2.1 题目分析

本实验关键在于令访存模式具有更好的空间局部性，从而发挥 cache 的能力。平凡算法逐列访问矩阵元素，一步外层循环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。但是后者的访存模式具有很好空间局部性，可以令 cache 的作用得以发挥。此外，算法需要进行大量的循环，为了降低循环的影响，使用 2 阶循环展开对 cache 优化算法进行再次优化。

1.2.2 算法设计

平凡算法

```
for (int i = 0; i < n; i++)
    sum[i] = 0.0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
        sum[i] += arr[j][i] * vect[j];
}
```

cache 优化算法

```
for (int i = 0; i < n; i++)
    sum[i] = 0.0;
for (int j = 0; j < n; j++) {
    for (int i = 0; i < n; i++)
        sum[i] += arr[j][i] * vect[j];
}
```

cache 优化算法 2 阶循环展开

```
for (int i = 0; i < n; i++)
    sum[i] = 0.0;
for (int j = 0; j < n; j++) {
```

```
4       for (int i = 0; i < n - 1; i += 2) {
5           sum[i] += arr[j][i] * vect[j];
6           sum[i + 1] += arr[j][i + 1] * vect[j];
7       }
8       if (n % 2)
9           sum[n - 1] += arr[j][n - 1] * vect[j];
10  }
```

1.3 实验环境说明

实验平台分别为鲲鹏服务器的 ARM 平台和个人 PC 使用 intel i5 处理器的 x86 平台。两平台缓存参数如下（未列出指令缓存大小）:

平台 \ 缓存层级	L1	L2	L3
ARM	64KB	512KB	48MB
x86	256KB	1MB	8MB

1.4 实验方案设计

首先，从 n=50 开始以 50 为步长逐渐增大问题规模至 12000，分别测试三个算法的运行时间，从而对程序运行时间有一个宏观的认识。然后选取其中较有代表性的问题规模结合相应工具进行更加细致的分析。

1.5 实验结果呈现及分析

首先在 arm 平台中进行测试，运行时间随问题规模增长的曲线如图1.1所示。

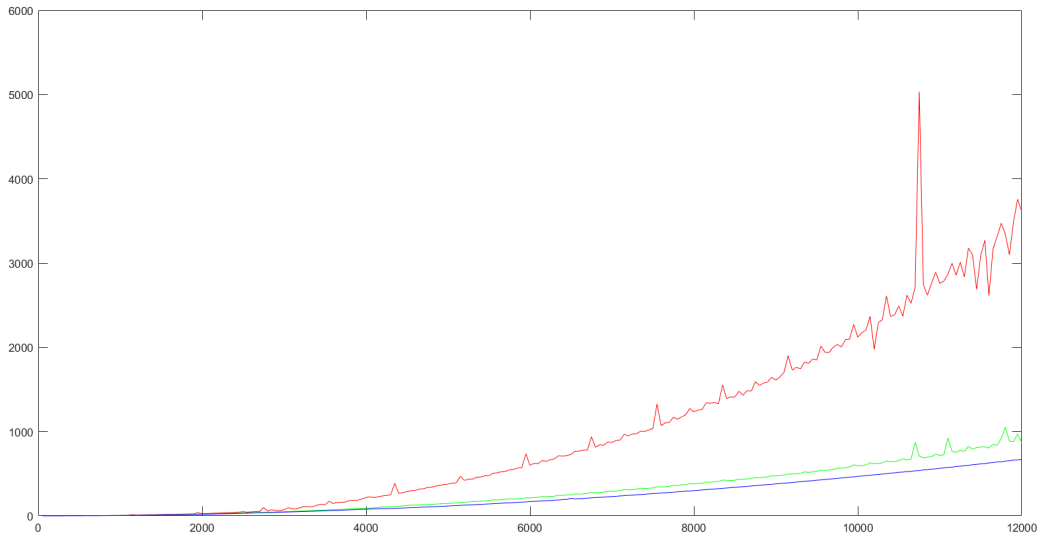

图 1.1: 运行时间（ms）随问题规模变化图
r: 平凡算法 g: 优化算法 b: 优化算法 2 阶循环展开

由上图可以看出，三种算法的增长速度均为 O(n^2) 级别。问题规模较大时，代表平凡算法的红色曲线耗时最长，代表优化算法的绿色曲线位于中间，代表优化算法 2 阶循环展开的蓝色曲线耗时最短。问题规模较小时，三者相差较小。

针对上图，我们选取问题规模为 10、100、1000、5000、10000 进行更加细致的分析，在 arm 平台和 x86 平台中对比运行。如表1所示。

平台	算法\规模	10	100	1000	5000	10000
ARM	平凡算法	0.000627904	0.0557563	6.31702	376.019	2115.24
	优化算法	0.000573197	0.0493405	5.09753	149.046	587.847
	2 阶循环展开的优化算法	0.000524659	0.0441626	4.545	113.72	456.979
x86	平凡算法	0.000281705	0.0256379	7.48	234.374	1036.73
	优化算法	0.000260439	0.0236379	2.194	53.362	219.503
	2 阶循环展开的优化算法	0.000221939	0.0188199	1.99467	47.903	192.655

表 1: cache 优化性能测试结果（单位:ms）

结合上表，可以看到，x86 平台上程序运行性能优于 arm 平台程序运行性能，可能是因为 x86 平台有更大的 L1 和 L2 缓存，因此访存时间更短造成的。接下来分析平凡算法与优化算法的差异的原因。我们可以看到在问题规模为 10 和 100 时，平凡算法用时与优化算法用时相近。而随着问题规模的增大，平凡算法用时/优化算法用时逐渐增大。当问题规模为 10000 时，arm 平台中二者之比达到了 3.6。为了验证缓存是产生差异的原因之一，使用 perf 工具对 arm 平台运行的程序进行缓存命中分析，结果如表2所示。

算法 \ 规模	10	100	1000	5000	10000
平凡算法	0.03%	0.29%	2.06%	4.29%	4.24%
优化算法	0.03%	0.43%	0.34%	0.41%	0.46%
2 阶循环展开的优化算法	0.03%	0.42%	0.35%	0.36%	0.42%

表 2: arm 平台 perf 测试程序 L1 缓存缺失率

可以看到，当问题规模较小时，平凡算法和优化算法的 cache 缺失率大致相近，这可能时因为问题规模较小时，cache 可以容纳所有的数组元素，此时平凡算法和优化算法的用时大致相同。而随着问题规模逐渐增大，cache 空间不足，此时访存方式对程序起了更重要的作用，逐行访存的优化算法的 cache 命中率更高，故而性能更好。

根据表2，我们看到，优化算法和 2 阶循环展开的优化算法在各个规模下的 cache 缺失率大致相同。但是结合表1，不难发现，在每个规模下循环展开优化算法的运行时间均小于普通的优化算法的运行时间，当 n=10000 时，arm 平台中普通优化算法和循环展开优化算法之比为 1.3。接下来利用 perf 分析普通优化算法和循环展开优化算法的 IPC，结果如表3。

算法 \ 规模	10	100	1000	5000	10000
优化算法	2.42	2.54	2.28	2.10	2.13
2 阶循环展开的优化算法	2.63	2.74	2.49	2.38	2.40

表 3: cache 优化实验 arm 平台 perf 测试程序 IPC

可以看到，各个规模下的循环展开的优化算法的 IPC 均大于普通优化算法的 IPC。因此我们有理由相信普通优化算法和循环展开的优化算法性能差异是由于循环展开增大了 IPC 所造成的。

2 超标量优化

2.1 实验介绍

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。

2.2 程序设计思路

2.2.1 题目分析

超标量优化的关键在于尽可能减少指令之间的依赖性，从而使得处理器中的流水线尽可能的充满，从而提升程序性能。

2.2.2 算法设计

平凡算法

```
double sum = 0.0;
for (int i = 0; i < n; i++)
    sum += a[i];
return sum;
```

两路链式累加

```
double sum = 0.0, sum1 = 0.0, sum2 = 0.0;
for (int i = 0; i < n; i += 2) {
    sum1 += a[i];
    sum2 += a[i + 1];
}
sum = sum1 + sum2;
return sum;
```

树形递归算法

```
int lengthB = (n >> 1);
double *b = new double[lengthB];
for (int i = 0; i < lengthB; i++)
    b[i] = a[2 * i] + a[2 * i + 1];
for (int stage = 2; stage <= lengthB; stage <<= 1) {
    int step = (stage >> 1);
```

```
7       for (int i = 0; i < lengthB; i += stage) {
8           b[i] += b[i + step];
9       }
10  }
11  double sum = b[0];
12  delete[] b;
13  return sum;
```

2.3 实验环境说明

实验环境同 cache 优化实验环境。

2.4 实验方案设计

设置实验规模为 2 的整数幂，考虑到三种算法时间复杂度均为 O(n) 级别，分别测试规模为 2^5、2^{10}、2^{15}、2^{20}、2^{25} 时平凡算法和两路链式累加算法及树形递归算法运行时间，并对结果加以分析。为确保结果的准确，规模较小时多次运行取平均值。

2.5 实验结果呈现及分析

根据实验方案进行实验，算法运行时间结果如表4所示。

平台	算法\规模	2^5	2^{10}	2^{15}	2^{20}	2^{25}
ARM	平凡算法	0.000204821	0.00555102	0.177297	5.77991	193.555
	2 路链式累加	0.000132413	0.00280638	0.0891956	2.91754	104.421
	树形递归算法	0.000242172	0.00495452	0.16116	6.15649	258.365
x86	平凡算法	0.0000743285	0.00245249	0.0712532	2.50075	77.4
	2 路链式累加	0.0000540676	0.00126968	0.0403346	1.21844	41.915
	树形递归算法	0.000145895	0.00250091	0.0806765	3.98933	129.653

表 4: 超标量优化算法性能测试（单位:ms）

可以看到，无论在 arm 平台还是 x86 平台，2 路链式累加算法相比于平凡算法均具有较好的性能，几乎有 2 倍的性能提升，而树形递归算法的性能大体上却没有想象中的那样好。

重点分析 arm 平台程序运行情况，首先我们利用 perf 对程序运行的 IPC 进行检测。结果如表5所示。

算法 \ 规模	2^5	2^{10}	2^{15}	2^{20}	2^{25}
平凡算法	1.10	1.08	1.07	1.08	1.06
2 路链式累加	1.43	1.68	1.68	1.52	1.29
树形递归算法	2.21	2.20	2.11	1.80	1.47

表 5: 超标量优化实验 arm 平台 perf 测试程序 IPC

由上表可以看出，2 路链式累加的 IPC 明显优于平凡算法的 IPC，即 2 路链式累加算法充分的运用了超标量处理器的多条流水线。且平凡算法程序和 2 路链式累加算法代码相近，总指令数也应相近。因此，2 路链式累加算法程序性能要好于平凡算法程序性能。

但是，我们可以明显地看到树形递归算法程序的 IPC 更优，但是树形递归算法程序却并未表现出与之相应的性能。利用 perf 得到 2^{15}、2^{20}、2^{25} 问题规模下程序单次运行需要的指令数，结果如表6所示。

算法 \ 规模	2^{15}	2^{20}	2^{25}
平凡算法	519474	23124978	940686658
2 路链式累加	408605	16281162	840023296
树形递归算法	924912	36234393	1360120796

表 6: 超标量优化实验 arm 平台 perf 测试程序运行指令数

显然，单次运行树形递归算法所需的指令数较平凡算法有明显的增加。这可能是因为树形递归算法与平凡算法相比，其进行的加法操作总数量和平凡算法相同，且其算法更加复杂，从而导致程序执行的指令条数更多。此外，结合 cache 优化实验得到的经验，我们也可以基本判断出树形递归算法相比于平凡算法及 2 路链式累加算法的空间局部性更差，平均访存耗时会更长。故而其性能相比于平凡算法没有较好的提升。

3 实验源码项目 GitHub 链接

实验源码项目 GitHub 链接