南開大學
Nankai University

计算机学院

并行程序设计作业 1

体系结构相关及性能测试

姓名：林语盈

学号：2012174

专业：计算机科学与技术

2022 年 3 月 10 日

目录

1 Cache 优化

1.1 作业题目

计算给定 n*n 矩阵的每一列与给定向量的内积，两类算法设计思路：
a) 逐列访问元素的平凡算法；
b) cache 优化算法。

1.2 程序设计

由于矩阵按行主次序存储，平凡算法逐列访问，优化算法逐行访问。理论上，在同等的问题规模下，cache 优化算法的的运行的时间更短。

平凡算法

```
// 逐列访问矩阵元素:外层循环每次计算出一列的内积结果
for (int i = 0; i < N; i++) {
  inner_product1[i] = 0.0;
  for (int j = 0; j < N; j++)
    inner_product1[i] += a[j][i] * b[j];
}
```

cache 优化算法

```
//逐行访问矩阵元素:一步外层循环不计算出任何一个内积，只向每个内积累加一个乘法结果，
当外层循环结束后得到全部结果。
for (int i = 0; i < N; i++)
  inner_product2[i] = 0.0;
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    inner_product2[j] += a[i][j] * b[i];
```

1.3 实验平台

实验平台		
处理器	AMD Ryzen 5 Mobile 4500U	华为鲲鹏
CPU 时钟主频	1 389.65MHz	200-2600MHz
操作系统	Windows10	Linux
指令架构	x86	ARM v8(aarch64)
编译器	GCC 9.2.0	GCC 9.3.1
L1 Data cache	6 x 32 KB (8-way, 64-byte line)	64KB
L1 Instruction cache	6 x 32 KB (8-way, 64-byte line)	64KB
L2 cache	6 x 512 KB (8-way, 64-byte line)	512KB
L3 cache	2 x 4 MB (16-way, 64-byte line	49152KB
内存大小	16GB	191GB

1.4 实验方案

1.4.1 平凡算法与优化算法的程序执行时间

矩阵规模 N 分别取 100、200、400、600、800、1024、2048、4096、6144、8192、10240 时分别进行试验。

为了确保数据的可靠性，使用高精度计时，同时在 $N < 2048$ 时，采用 100 次做平均；在 N 较大时，采用 10 次做平均的时间。

1.4.2 探讨编译器不同优化力度对性能的影响

取 N=1024 时，采用 100 次重复的平均值。

采用不优化/Od、最大优化（优选大小）/O1、最大优化（优化速度）/O2、优化（优选速度）/Ox 优化分别测试。

1.4.3 浮点数运算次序对不同对结果的影响

取 N=1024 时，采用 100 次重复的平均值。

分别测试加法和乘法运算的次序不同时，对程序运行结果是否有影响。

1.4.4 测试不同操作系统和不同指令集对程序性能的影响

分别在华为鲲鹏服务器（Linux 系统、ARM 指令集）与笔记本本地（Windows10 操作系统，x86 指令集）下测试程序的运行时间，并作对比。

1.5 实验结果及分析

1.5.1 测试矩阵规模不同时的程序执行时间

实验结果如表所示：

表 1: 程序执行时间（ms）

N	100	200	400	600	800	1024
平凡算法	0.030635	0.151624	0.469765	1.03909	1.97351	3.34865
cache 优化算法	0.023203	0.1105	0.334465	0.761412	1.39908	2.279
N	2048	4096	6144	8192	10240	
平凡算法	28.8281	120.275	311.162	645.618	2168.46	
cache 优化算法	8.7854	35.7367	79.7823	142.118	224.145	

两种算法的执行时间，如图所示

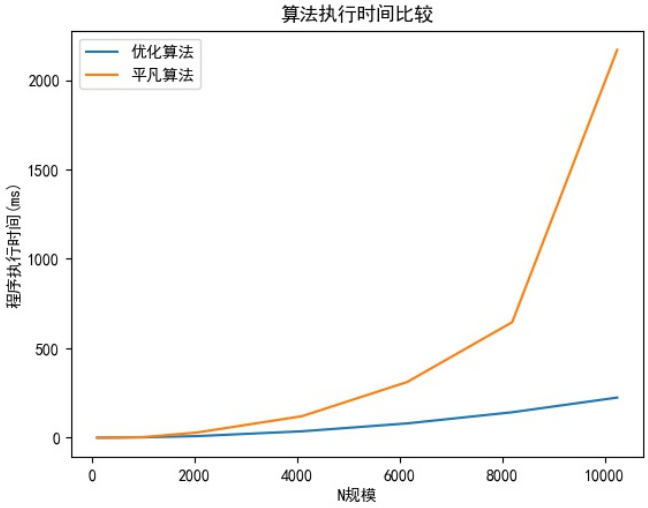

图 1.1: 两种算法的执行时间

cache 算法的优化效果，如图所示

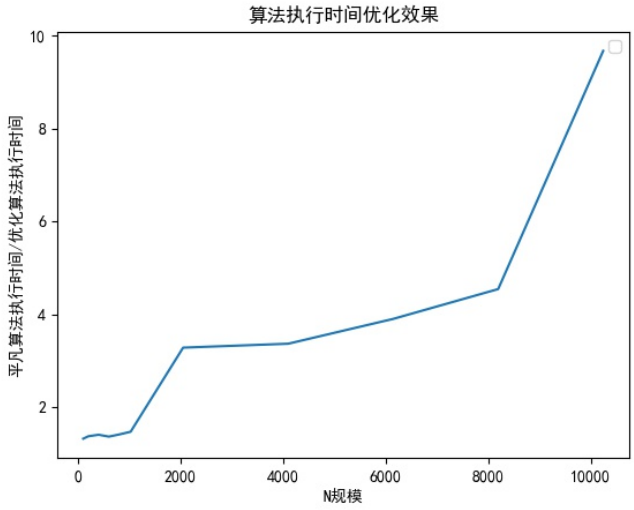

图 1.2: cache 算法的优化效果

可以看到，在同等规模下，cache 优化的算法比平凡算法的程序执行时间具有明显优势，随着 N 规模的增大，时间被等倍放大，优势更加明显，这基本与预期相同。但是在实验过程中发现，不同时间运行时，相同算法的程序运行时间亦有差别，这可能有多种原因，如程序运行时机器的缓存占用情况、计时误差等。

1.5.2 编译器不同优化力度对性能的影响

$N = 1024$, 实验结果如下表所示。可以看到，O1 优化对平凡算法的程序运行时间并无明显效果，但对优化算法有显著效果；O2 和 Ox 优化差别不大，均对算法的运行时间有一定程度的优化，而对优化算法的加速略大于对平凡算法的加速。

表 2: 编译器优化的影响

	平凡算法	优化算法
/Od	4.51645	2.39194
/O1	5.1532	0.890424
/O2	2.55507	0.883307
/Ox	2.98659	0.869442

1.5.3 浮点数运算次序对不同对结果的影响

预期中，因为浮点数精度的影响，可能会导致运算结果不同。但在本次实验中，经检验，改变浮点数的运算次序，对计算结果影响不大。原因可能是 N 的规模较小或是浮点数值的值精度较小。

1.5.4 测试不同操作系统和不同指令集对程序性能的影响

矩阵规模 N 分别取 1024、2048 进行试验，采用 10 次做平均的时间，下表分别列出 Windows（x86 指令集）和 Linux 平台（ARM 指令集）下的运行时间差异。

表 3: 不同平台和指令集下的程序执行时间，单位：ms

	Windows(x86)		Linux(ARM)	
	平凡算法	Cache 优化算法	平凡算法	Cache 优化算法
N=1024	3.34865	2.279	7.3991	5.6393
N=2048	28.8281	8.7854	31.3364	23.3732

可以看到，ARM 架构相比 x86 架构普遍运行时间较长，且 Cache 优化的效果不如 x86，尤其是当 N=2048 时，x86 上的 Cache 优化效果达到 3 倍左右，但 ARM 下却仅有 1.5 倍。
在 windows 系统中，由于 AMD 处理器的限制，无法使用 VTune 进行性能分析，经过多次尝试，也无法使用 AMD Prof 进行程序性能的分析。
在 Linux 中，Cache 命中率如下表所示：

表 4: L1 Cache 命中率统计

	L1-dcache-loads		L1-dcache-load-misses		命中率	
	平凡算法	Cache 优化算法	平凡算法	Cache 优化算法	平凡算法	Cache 优化算法
N=1024	214,768,075	214,478,188	5,335,050	755,013	97.516%	99.648%
N=2048	857,043,945	856,986,324	35,514,972	2,989,334	95.856%	99.651%

可以看到，在 N 相同的情况下，L1 缓存总 load 数相当，而优化算法的缺失值要远远小于平凡算法，即命中率要远远高于平凡算法。随着 N 的增加，L1 缓存总 load 数增加，而优化算法命中率的优势亦随之增加。从程序运行的时间来看，优化算法的运行的时间更短，cache 命中率更高。这与先前的预想相同，优化算法的访存模式与行主存储匹配，具有更好的空间局部性，令 cache 的作用得以发挥。

1.6 总结

通过上述实验，比较了平凡算法于 Cache 优化算法的程序运行时间，探讨了编译器不同优化力度对性能的影响，探究了浮点数计算次序对不同结果的影响，测试的不同操作系统和不同指令集对程序性能的影响，并运用 perf 对程序的 cache 命中率进行分析。

2 超标量优化

2.1 作业题目

计算 n 个数的和，两类算法设计思路:

a) 逐个累加的平凡算法（链式）

b) 超标量优化算法，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.

2.2 程序设计

分别实现了逐个累加的平凡算法，2 路链式、4 路链式、通过递归函数实现的递归算法、通过双层循环实现的递归算法。多路链式即相当于多步循环展开。

平凡算法

```
// n个数逐个累加
ans = 0.0;
    for (int i = 0; i < N; i++) {
        ans += a[i];
    }
```

2 路链式

```
// 采用2路链式，每次循环步中，分别对两路进行累加。
double sum1=0.0, sum2=0.0;
    for (int i = 0; i < N; i += 2) {
        sum1 += a[i];
        sum2 += a[i + 1];
    }
    ans = sum1 + sum2;
```

4 路链式

```
// 采用4路链式，每次循环步中，分别对四路进行累加。
double sum1=0.0, sum2=0.0, sum3=0.0, sum4=0.0;
    for (int i = 0; i < N; i += 4) {
        sum1 += a[i];
        sum2 += a[i + 1];
        sum3 += a[i + 2];
        sum4 += a[i + 3];
    }
```

```
    ans = sum1 + sum2 + sum3 + sum4;
```

递归算法 (函数)

```
// 采用递归算法，不断将问题划分为二，依次递归计算序列的和，使用递归函数实现。
void recursion(int n){
    if (n == 0)
        return;
    if(n%2)
        for (int i = 0; i <=n / 2; i++)
            a[i] += a[n / 2+ i +1];

    else
        for (int i = 0; i < n / 2; i++)
            a[i] += a[n / 2+ i+1];
    recursion(n/2);

}
```

递归算法 (循环)

```
// 采用递归算法，算法同上，使用循环实现，最终a[0]中为结果。
for (int m = N; m > 0; m /= 2)
    if(m%2)
        for (int i = 0; i <=m / 2; i++)
            a[i] += a[m / 2+ i +1];
    else
        for (int i = 0; i < m / 2; i++)
            a[i] += a[m / 2+ i+1];
ans4 = a[0];
```

2.3 实验平台

实验平台		
处理器	AMD Ryzen 5 Mobile 4500U	华为鲲鹏
CPU 时钟主频	1 389.65MHz	200-2600MHz
操作系统	Windows10	Linux
指令架构	x86	ARM v8(aarch64)
编译器	GCC 9.2.0	GCC 9.3.1

2.4 实验方案

2.4.1 平凡算法与优化算法的程序执行时间

在验证了程序正确性的前提下，测试程序的运行时间。

矩阵规模 N 分别取 2^{20}、2^{21}、2^{22}、2^{23} 时分别进行试验。

为了确保数据的可靠性，使用高精度计时，采用 10 次做平均的时间。

2.4.2 测试不同操作系统和不同指令集对程序性能的影响

分别在华为鲲鹏服务器（Linux 系统、ARM 指令集）与笔记本本地（Windows10 操作系统，x86 指令集）下测试程序的运行时间，并作对比。

2.5 实验结果及分析

2.5.1 测试矩阵规模不同时的程序执行时间

表 5: ARM 程序运行时间

	N=1024*1024	N=1024*2048	N=1024*4096	N=1024*8192
平凡算法	5.7217ms	11.4986ms	23.2893ms	47.4498ms
二路链式	4.5863ms	9.3896ms	20.7771ms	42.8392ms
四路链式	1.723ms	3.574ms	8.1418ms	17.0311ms
递归（循环）	5.8744ms	12.6151ms	29.0388ms	65.1184ms
递归（函数）	5.6223ms	11.5591ms	24.1455ms	49.9635ms

从表中可以看到，从运行时间上看，递归（循环）> 递归（函数）≒ 平凡算法 > 二路链式 > 四路链式。多路链式的结构因为其超标量的结构，利用多条流水线，使多路可以并行执行，拥有更大的加速比，其中在一定限度内，随着多路的增加，优化效果随之增加。

链路数也会影响流水线级数，即每条流水线执行的步骤数，这二者共同决定了程序的运行时间，此外链路数亦受到硬件架构的影响，如 ARM7 是冯 · 诺依曼结构，采用了典型的三级流水线，而 ARM9 则是哈佛结构，采用五级流水线技术，而 ARM11 则更是使用了 7 级流水线。

递归的方式运行时间较长，尤其是使用循环形式，这可能是因为无论函数递归调用或是两层循环都使运行过程中产生大量不必要的计算，如循环步的控制、函数调用时的出入栈操作等，反而取得了更差的效果，这一点在下表中有更明显的体现。

使用 perf 调查程序的指令数和周期数，如下表所示：

表 6: 程序指令数与循环数比较

N=2^23	指令数	循环数	指令数/循环数
平凡算法	1,989,243,270	1,345,183,036	1.48
二路链式	1,360,100,736	766,844,482	1.77
四路链式	1,213,294,770	541,278,114	2.24
递归（循环）	3,415,310,182	1,665,383,932	2.05
递归（函数）	3,331,425,712	1,390,426,822	2.40

从表中可以看到，平凡算法、二路链式、四路链式的指令数比较少，而递归算法的指令数是其将近 3 倍，这也说明在函数调用过程中出现的大量其它的运行步骤，而平凡算法的指令数多于链式，这也是因为平凡算法进行的较多的循环步。

关于指令数/循环数的值，随着链路数的增加而增加，递归算法也达到了 2 以上，这就说明了流水线对程序执行时间的优化，可递归算法因执行的其它指令较多，故程序总时间上并未显示出优势。

2.5.2 测试不同操作系统和不同指令集对程序性能的影响

在 x86 指令集下，程序的运行时间如下表所示：

表 7: x86 程序运行时间

	N=1024*1024	N=1024*2048	N=1024*4096	N=1024*8192
平凡算法	1.5435ms	4.50719ms	9.36259ms	17.8227ms
二路链式	1.33041ms	3.54535ms	7.82711ms	15.6099ms
四路链式	0.56285ms	1.36227ms	3.13559ms	6.51731ms
递归（循环）	1.55627ms	3.83029ms	8.69988ms	17.6998ms
递归（函数）	0.99311ms	2.5815ms	5.81042ms	12.4939ms

与 arm 指令集上相比，递归（函数）的相对运行时间明显减少，而其他的相对运行时间基本相同，这可能是由于指令集的不同，使函数调用时的命令执行不同。
X86 有三种常用调用约定，cdecl(C 规范)/stdcall(WinAPI 默认)/fastcall 函数调用约定。其中：

- Cdecl 调用规范：参数从右往左一次入栈，调用者实现栈平衡，返回值存放在 EAX 中。
- stdcall 调用规范：参数从右往左一次入栈，被调用者实现栈平衡，返回值存放在 EAX 中。
- fastcall 调用规范：参数 1、参数 2 分别保存在 ECX、EDX ，剩下的参数从右往左一次入栈，被调用者实现栈平衡，返回值存放在 EAX 中。

arm64 函数调用约定:

- arm64 位调用约定采用 AAPCS64。
- 参数 1 参数 8 分别保存到 X0 X7 寄存器中，剩下的参数从右往左一次入栈，被调用者实现栈平衡，返回值存放在 X0 中。

此外，ARM 指令集下程序的运行时间普遍大于 x86 指令集下的程序运行时间，这可能是由于 x86 架构比较复杂，普遍多核，时钟频率也较高。

2.6 总结

通过上述实验，分别测试了多路链式（即循环的分步展开）、递归算法、平凡算法的运行时间，并对其进行了分析比较，还比较了不同指令集架构下的运行时间，使用 perf 工具进行分析，进一步看到了超标量优化的效果。

3 附录: 程序完整代码

程序完整代码

参考文献