南開大學
Nankai University

计算机学院

并行程序设计实验报告

cache 优化和超标量优化方法的探究

姓名：刘心源

学号：2112614

专业：计算机科学与技术

2023 年 3 月 12 日

目录

1 实验环境

1.1 实验所用架构

	ARM	
CPU 型号	Apple Silicon M1 Pro	
CPU 主频	3.2GHz	
cache	大核	小核
L1 cache	每核 128KB，共 512KB	每核 128KB，共 512KB
L2 cache	12MB 共用	4MB 共用
L3 cache	16MB（包括 GPU 等）	

表 1: 实验环境

1.2 实验所用操作系统

实验过程使用 Apple Silicon M1Pro 上的 Ubuntu 22.04.2 Linux ARM64 虚拟机进行性能耗时的测量，编译器使用 g++ 进行编译。

2 程序代码实现

由于篇幅原因，全部代码已经上传个人的Github

3 体系结构相关实验分析——cache 优化

3.1 问题重述与分析

计算给定的 n*n 矩阵的每一列与给定向量的内积，考虑两种算法设计思路：

1. 逐列访问平凡算法
2. cache 优化算法
3. unroll 循环展开算法

3.2 算法设计

针对给定问题，由于矩阵在内存中存储时是按行存储的。按行访问，则实际上是连续读取，刚好大多次都能够 cache 命中；而按列读取，则是非连续读取，cache 命中的理论可能性不大，所以需要访问主存的次数增多，于是，所花费时间变多。对于原始的逐列访问算法来说，CPU 会一次读入连续的一段数据到 cache 中，但是其中可能只包含一个需要计算的元素，因此在计算该列的第二个元素时，CPU 需要重新读入所需要的元素，访存时间较大，极大地降低程序运行效率。因此考虑改进算法。

3.2.1 平凡算法设计思路

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；

3.2.2 cache 优化算法设计思路

cache 优化算法则是逐行访问矩阵元素，一步外层循环无法计算出任何一个内积，只是向每个内积累加一个乘法结果。这种算法的访存模式具有很好的空间局部性，极大地利用了 cache 中的缓存数据，减少访存时间。

3.2.3 unroll 优化算法设计思路

为了降低循环访问过程中，条件判断、指令跳转等额外耗时，我们可以采用循环展开的方式，在一次循环中计算多个位置的值，利用多条流水线同时工作。

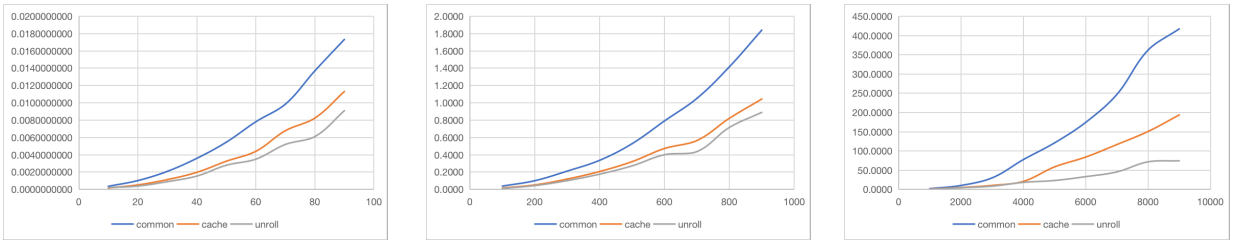
3.3 实验分析

我们设计三种算法，并编写代码在 Apple M1 pro 的 Unbuntu20.04 虚拟机上进行测试，测试数据如下：

n	common	cache	n	common	cache	unroll	n	common	cache	unroll
10	0.0004	0.0002	100	0.0389	0.0170	0.0121	1000	2.3559	1.3310	1.1531
20	0.0010	0.0005	200	0.1013	0.0524	0.0448	2000	10.5918	5.3239	4.6674
30	0.0021	0.0011	300	0.2123	0.1201	0.1020	3000	30.3316	10.6942	8.7591
40	0.0036	0.0020	400	0.3367	0.2076	0.1769	4000	77.7402	21.1333	18.5588
50	0.0055	0.0033	500	0.5287	0.3214	0.2728	5000	121.1070	58.7636	23.4518
60	0.0078	0.0044	600	0.7917	0.4732	0.4016	6000	174.3990	84.3518	33.4516
70	0.0099	0.0068	700	1.0518	0.5652	0.4378	7000	246.9520	116.8450	45.8288
80	0.0137	0.0082	800	1.4161	0.8220	0.7163	8000	362.0650	150.7640	71.8750
90	0.0173	0.0113	900	1.8404	1.0442	0.8907	9000	417.2860	193.7450	74.6580

表 2: 不同规模下三种方法运行时间

结合实验数据，我们可以绘制出在不同数据规模情况下时间随数据规模的变化情况，如图3.1所示。



(a) 数据规模 N<=100 时　(b) 数据规模 100<=N<1000 时　(c) 数据规模 1000<=N<10000 时

图 3.1: 不同算法时间随着问题规模的变化

由测试数据我们可以看出，在 N<200 的规模下，逐行访问或者逐列访问在效率上有一定差距但是三种方式差别不大，函数增长速度也相差不大。在 200<N<2000 的数据规模时，逐行访问比逐列访问的增长速度要慢，但是此时三种方法的增长趋势还是基本相同的。在 N>2000 后，逐行访问和逐列访问的方式所需时间已经有较大差别，逐列访问的耗时增长速度急剧增大，明显超越逐行访问的方法，证明此时 cache 优化起到了显著作用，但是逐行访问与采用循环展开方式时间差距较小。当 N>4000 后，逐行访问与采用循环展开方式时间差距迅速拉大。

根据数据进行理论分析，由于该 CPU 的 L1 cache 大小为 512KB，L2 cache 大小为 12MB。对于数组元素为 unsigned long long int 来说，每一个元素占据 8 个字节。故填满 L1 cache 所需数据规模约为 256，填满 L2 cache 所需数据规模为 1254。图像走势也与此差不多。在 N<1254 时，即使 common 算法的 L1 cache 命中率已经比较低，但是由于 L2 cache 访存速度较快，所以访存速度对整体影响不大，当数据规模上升到 2000 时，L2 cache 的命中率不断下降，使得 common 算法被迫到芯片共用的 L3 cache 和内存中寻找数据，查找速度迅速降低，这样导致访存开销是巨大的，使得两种方法的访问效率产生了显著差异。

对于循环展开方法对逐列访问方式的优化，由于循环展开可以在一个循环周期中利用多条流水线，并行执行多条相同指令，故能在一定程度上优化耗时。我们可以利用 perf 比较两种方法的 CPI，得到证明。如表 3 所示。

	common	optimize
CPI	0.4980	0.4753

表 3: common 和 unroll 方法的 CPI 对比

4 体系结构相关实验分析——超标量优化

4.1 题目相关与分析

计算 n 个数的和，考虑两种算法设计思路：

1. 逐个累加的平凡算法
2. 适合超标量架构的指令级并行算法（相邻指令无依赖）

4.2 算法设计

对于常规的顺序算法而言，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，无法发挥 CPU 超标量优化的性能，因此考虑多链路的方式对于传统的链式相加的方法进行改进。设置多个临时变量，在一个循环中利用多个临时变量，对于多个不同位置进行累加，可以在多条流水线上同时作业，达到多个位置并行累加的效果，同时也能减少利用循环展开的技术降低循环的比较次数，减少开销。

本题在数据规模较小的时候采用 LOOP 循环对算法的核心计算重复合适次数，以提高性能测试的精度。在问题规模方面采用双链路并行算法，考虑到流水线为 2 条，选取 2 的 n 次幂作为数据规模的值。

由于多链路方法使用了循环展开技术降低循环的额外开销，为了保证实验准确性，我们对于普通的链式累加方法也要进行同样比例的循环展开，控制实验变量保证实验的合理性。

通过对比不同数据规模下的两种方法运行时间，可以探究优化多条流水线对时间的加速与问题规模的变化情况，并寻找分析其中的原因。同时计算出不同数据规模下优化算法相对于平凡算法的加速比，并试图从 cache 及内存的角度给出合理解释。

4.2.1 平凡算法设计思路

采取链式算法，将给定元素依次累加到结果变量即可。为了保证实验准确性，对其进行 unroll 改进。

经过循环展开后的平凡算法

```
for(int k = 1; k <= LOOP ; k++)
{
    unsigned long long int sum = 0;
    for (int i = 0; i < n - 1; i += 2)
    {
        sum += c[i];
        sum += c[i+1];
    }
}
```

4.2.2 并行算法设计思路——多链路超标量

选取合理的步长（为实验验证方便选取步长 =2），然后两路链式并行累加，最后将两路链式的累加结果加到结果变量中即可。

二路链路并行算法

```
for(int k = 1; k <= LOOP; k++)
{
    unsigned long long int sum1 = 0;
    unsigned long long int sum2 = 0;
        for(int i = 0;i < n;i += 2)
        {
            sum1 += c[i];
            sum2 += c[i+1];
        }
    unsigned long long int and = sum1+sum2;
}
```

4.3 实验分析

为方便算法的实现，我们所有问题规模都取为 2 的 n 次幂，由于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，因此我们从 n=10 的稍大规模开始进行测试并记录数据。

n	common	optimize	n	common	optimize
10	0.0015	0.0013	19	0.7260	0.6119
11	0.0027	0.0023	20	1.4072	1.2442
12	0.0056	0.0039	21	2.7531	2.4158
13	0.0190	0.0139	22	5.5737	4.9894
14	0.0322	0.0196	23	10.9682	9.7498
15	0.0530	0.0387	24	21.8257	19.4370
16	0.0959	0.0753	25	43.6045	39.0337
17	0.1770	0.1514	26	88.2018	77.5922
18	0.3795	0.3031	27	178.3560	154.9790

表 4: common 算法和 optimize 算法耗时随数据规模变化

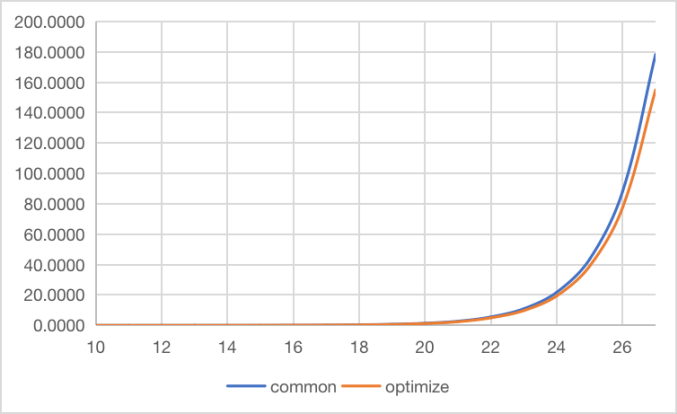


图 4.2: 两种方法耗时随数据规模变化的图

由表中数据可以看出，无论是链式相加还是多路展开超标量的方法，都属于线性时间效率的方法，随着问题规模不断翻倍，时间也近似于翻倍。使用双链路展开的超标量优化方法时间效率在数据规模较大的时候，显著高于普通的链式相加的方法。因为通过多链路的处理方式，让原本相互关联的累加变成了互不相关的操作，使得 CPU 可以同时调用两条流水线处理问题，从而实现了超标量优化的目的。为了验证超标量的优化确实起到了作用，我使用 perf 检测了链式累加与多路累加的 IPC。根据表中的数据，可以得到多链路累加方法的 IPC 要低于链式累加，即在同一个时钟周期中，多链路累加执行的指令数要多于链式累加，这也证明我们使用多条流水线实现了超标量优化的目的。

	common	optimize
CPI	0.667	0.498
IPC	1.5	2

表 5: 链式累加和多链路累加的 CPI 与 IPC 对比

为了进一步研究超标量优化的加速比，我们绘制优化加速比与数据规模之间的关系图。如图4.3所示

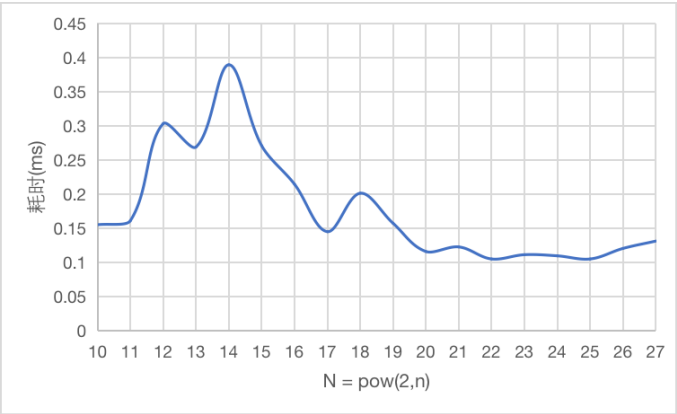


图 4.3: 优化加速比随数据规模增大的变化

通过4.3中实验数据可以发现，当问题规模较小的时候，多路展开算法的优化加速比较低，但是随着问题规模增大，算法的优化加速比呈现先增加后降低的趋势。其增加的原因主要是通过多链路的处理方式，将累加拆分成两个不相关的部分，使得 CPU 可以同时调用两条流水线处理问题，利用 CPU

的超标量优化加速算法。问题规模为 2^14 时，超标量优化水平达到最高点。问题规模在介于 2^10 到 2^17 之间基本保持较高水平，当问题规模较大的时候，整体呈现出下降趋势。

对于该现象，我们初步猜测是问题规模较大导致的缓存不足，由于需要经常对内外存进行访问导致较大的访存开销。在 n 超过 27 后，耗时迅速增加，优化效果不明显，说明数据规模较大时访存开销急剧增大，超标量优化的效果变得不明显。为验证上述想法.

5 实验总结

对于给定的矩阵内积的问题，我们考虑采用 cache 优化的方法对原有的朴素串行算法进行加速，通过对比平凡算法和 cache 优化算法，可以明显对比出逐行访问在较大数据规模的情况下具有比逐列访问更好的性能。cache 优化算法能够充分的利用 cache 的缓存，提高数据在缓存中的命中率，进而降低不断访问内存的额外开销。对于数组求和的问题，我们采用超标量的优化方法，将求和问题转化为了多条流水线同时执行累加最后再求和，这样充分利用 CPU 的多条流水线作业，利用 CPU 的超标量特性，提升程序的性能。

在两个问题中我都采用了循环优化的方式，在把一次循环进行展开，可以利用多条流水线同时作业，也可以加快程序的执行，减少耗时开销。这种思路和超标量优化的思路相似。

同时在分析程序的 CPI 时，我学习了 perf 的使用，在 Linux 系统上使用 perf 对程序的 instructions 和 clocks 进行记录，并可以计算出 CPI 和 IPC。通过这些数据就可以看出优化方法比原来的串行方法执行效率更高。