南開大學
Nankai University

计算机学院

并行程序设计实验报告

体系结构相关编程实验

姓名：蔡沅宏

学号：2213897

专业：计算机科学与技术

2024 年 3 月 24 日

目录

1 问题概述

1.1 任务一

计算给定 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. 逐列访问元素的平凡算法。
2. cache 优化算法。

1.2 任务二

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 (链式)。
2. 超标量优化算法 (指令级并行)，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 实验环境

	ARM	x86
CPU 型号	华为鲲鹏 920 服务器	12th Gen Intel Core i7-12700H
CPU 主频	2.6Ghz	2.3Ghz
一级缓存	64KB	1.2MB
二级缓存	512KB	11.5MB
三级缓存	48MB	24.0MB

表 1: 实验环境

3 算法设计及原理分析

3.1 任务一：n*n 矩阵于与向量内积

3.1.1 平凡算法设计思路

对于矩阵 A 和向量 x 的内积，即取矩阵 A 的第一行和向量 x 做内积，得到结果的第一个元素；然后取矩阵 A 的第二行和向量 x 做内积，得到结果的第二个元素；如此类推，直到计算完所有行。

算法的实现可以使用循环结构来实现。具体来说，通过两个嵌套的循环，外层循环遍历矩阵的每一列，内层循环遍历每一行。

3.1.2 cache 优化算法设计思路

对于原始朴素的逐列访问算法来说，CPU 会一次读入连续的一段数据到缓存中，其中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，开销是很大的，这会在很大程度上降低程序运行的效率。

而 cache 优化就是旨在利用缓存，减少内存访问次数，从而提高计算效率。计算机系统中的缓存以缓存行（cache line）为单位加载数据。通过重用缓存中的数据，减少对内存的访问，尽量避免因为缓存未命中而导致的性能损失。

所以重新设计算法对于数据的访问顺序，使得矩阵和向量的访问方式符合缓存的读取模式，尽量减少缓存未命中。

3.1.3 进阶任务：采用循环展开（unroll）技术，降低循环操作对性能的影响

为了能够降低循环访问过程中，条件判断，指令跳转等额外开销，我们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，将循环体内的多次迭代计算展开成多份计算，以减少循环迭代的开销。在矩阵与向量内积计算中，通过一次性处理多个元素的计算，减少循环开销，利用多条流水线同时作业，发挥 CPU 超标量计算的性能。

3.2 任务二：n 个数求和

3.2.1 平凡算法设计思路

使用循环遍历 n 个数，将每个数累加到 sum 中。

3.2.2 超标量优化算法设计思路

对于给定的问题，要求计算 N 个数的和，对于常规的顺序算法而言，由于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，因此考虑使用多链路的方法对传统的链式累加方法进行改进，即设置多个临时变量，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。

4 代码设计

这一部分内容仅展示算法实现代码的关键部分，详细代码已上传至Github。

注：所有时间计算均通过 chrono 库的相关函数完成，这里就不再展示。

4.1 任务一：n*n 矩阵于与向量内积

4.1.1 逐列访问元素的平凡算法

逐列访问元素的平凡算法

```
ull* columnVectorInnerProduct() {//ull是自定义类型unsigned long long int
    static ull result[N] = { 0 }; // 内积结果存储数组
    for (int col = 0; col < N; ++col) {
        for (int row = 0; row < N; ++row) {
            result[row] += matrix[row][col] * vector[col];
        }
    }
    return result;
}
```

4.1.2 cache 优化算法

cache 优化算法

```
ull* columnVectorInnerProductCacheOptimized() {
    static ull result[N] = { 0 }; // 内积结果存储数组
    for (int row = 0; row < N; ++row) {
        for (int col = 0; col < N; ++col) {
            result[row] += matrix[row][col] * vector[col];
        }
    }
    return result;
}
```

4.1.3 进阶任务：采用循环展开 (unroll) 技术，降低循环操作对性能的影响

循环展开优化算法

```
void unroll()
{
    using namespace std::chrono;
    auto start = steady_clock::now();
    for (int l = 0; l < LOOP; l++) //LOOP表示循环次数
    {
        for (int i = 0; i < N; i++) //N表示矩阵维数
            sum[i] = 0;
        for (int j = 0; j < N; j += 10){
            int tmp0 = 0, tmp1 = 0, tmp2 = 0, tmp3 = 0, tmp4 = 0, tmp5 = 0, tmp6 = 0,
                tmp7 = 0, tmp8 = 0, tmp9 = 0;
            for (int i = 0; i < N; i++){
                tmp0 += a[j + 0] * b[j + 0][i]; tmp1 += a[j + 1] * b[j + 1][i];
                tmp2 += a[j + 2] * b[j + 2][i]; tmp3 += a[j + 3] * b[j + 3][i];
                tmp4 += a[j + 4] * b[j + 4][i]; tmp5 += a[j + 5] * b[j + 5][i];
                tmp6 += a[j + 6] * b[j + 6][i]; tmp7 += a[j + 7] * b[j + 7][i];
                tmp8 += a[j + 8] * b[j + 8][i]; tmp9 += a[j + 9] * b[j + 9][i];
            }
            sum[j + 0] = tmp0;  sum[j + 1] = tmp1;
            sum[j + 2] = tmp2;  sum[j + 3] = tmp3;
            sum[j + 4] = tmp4;  sum[j + 5] = tmp5;
            sum[j + 6] = tmp6;  sum[j + 7] = tmp7;
            sum[j + 8] = tmp8;  sum[j + 9] = tmp9;
        }
    }
    auto end = steady_clock::now();
    duration<double, milli> duration = end - start;
    cout << "unroll: " << duration.count()  << "ms" << endl;
}
```

4.2 任务二：n 个数求和

4.2.1 逐个累加的平凡算法

逐个累加的平凡算法

```
void ordinary()
{
    auto start = chrono::high_resolution_clock::now();
    for (int l = 0; l < LOOP; l++)
    {
        ull sum = 0;        //ull是前面define定义的unsigned long long int
        for (int i = 0; i < N; i++)
            sum += a[i];
    }
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, std::milli> duration = end - start;
    cout << "ordinary: " << duration.count() << "ms" << endl;
}
```

4.2.2 超标量优化算法（指令级并行）

超标量优化算法（指令级并行）

```
void optimize()
{
    auto start = chrono::high_resolution_clock::now();
    for (int l = 0; l < LOOP; l++)
    {
        ull sum = 0;
        for (int i = 0; i < N - 1; i += 2)
            sum1 += (a[i] + a[i + 1]);  //两路链式累加
    }
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, std::milli> duration = end - start;
    cout << "optimize: " << duration.count() << "ms" << endl;
}
```

5 实验分析

5.1 任务一：n*n 矩阵于与向量内积

由上述算法设计模块所示，为解决此问题共设计了三种算法，为体现其性能差异，我在 ARM 架构的华为云鲲鹏服务器上进行测试，实验测试数据如表2所示（单位为毫秒 ms）

n	Iterations	Ordinary	Cache	Unroll	n	Iterations	Ordinary	Cache	Unroll
10	100	0.052	0.051	0.045	1000	1	5.87	4.79	4.07
40	100	0.831	0.754	0.586	2000	1	26.85	20.43	17.79
80	100	2.94	2.81	2.12	3000	1	56.61	44.39	33.82
100	100	4.73	4.52	2.97	4000	1	223.96	83.77	70.01
300	10	4.41	4.26	3.34	5000	1	353.19	141.25	101.34
500	10	13.24	11.87	10.09	6000	1	516.74	197.28	138.87

表 2: 在鲲鹏服务器上不同算法性能测试结果 (单位:ms)

根据测试数据观察到，在规模 n 小于 500 时，逐行和逐列访问方式的效率差异较小，但通过循环展开优化算法可以看出性能具有较为明显的提升。当规模超过 2000 时，三种算法之间的性能差异愈发明显，普通算法的时间消耗增长明显高过其他两种算法，而这时 cache 优化算法的优异性能逐渐展现，这表明缓存优化发挥了重要作用。而 unroll 优化算法因为可以在一个循环内利用多条流水线并行执行指令，因此能够在一定程度上优化代码运行效率，所以在性能体现上会比前面两种普通串行算法性能更加优越。

那么透过现象看本质，我们接下来讨论问什么会出现以上现象：首先，对数据增长特点和差异进行分析，并联系鲲鹏服务器本身参数，鲲鹏服务器的各级缓存大小分别为 64KB、512KB 和 48MB，由于在这里，我使用的是 unsigned long long int 数据类型，其每个数据将占用 10B 的空间，那么我们想要填满第一级缓存，就需要 6400 个数据，对应矩阵维数为 80 维，同理，填满第二级需要 51200 个数据，对应维数约为 230 维，填满第三级对应。因此，在规模较小的情况下，普通方法也能有较好的一级缓存命中率，所以在上面的表格中，在维数小于 100 时，普通方法与 cache 优化方法几乎没有什么性能差别；当规模小于 300 时，因为一二级缓存的访问速度都较快，所以性能上差异不是很大。当规模超过 500 时，L2 缓存命中率逐渐下降，逐列访问方式需更多地访问 L3 缓存，增加访存开销，因此两种方法的差距逐渐显现。当规模超过 3000，逐列访问方式的 L3 缓存命中率也降低，导致需要到内存中寻找数据，极大增加访存开销，进一步加剧了两种方法的访问效率差异。

在以上分析的基础上，为进一步探究不同平台对于算法效率的影响，于是我在个人电脑上进行了同样的代码测试，因为有了以上分析结果的基础，我在个人电脑上进行测试时，只抓取了几个重要的数据，因为 12th Gen Intel Core i7-12700H 的各级缓存分别是：1.2MB、11.5MB 和 24.0MB。对应的能容纳的矩阵维数分别约是：350 维、1050 维、2500 维，所以我选取了 n 分别为 100、200、400、800、1500、2000、2500、3000、3500、400、5000 进行测试，得到了表3的结果（单位为微秒）：

n	Iteration	Ordinary	Cache	Unroll
100	10	169	176	122
200	10	699	686	469
400	1	347	279	193
800	1	2045	1136	774
1500	1	9084	4081	2683
2000	1	16030	7213	4649
2500	1	27521	11316	6424
3000	1	33172	15509	10533
3500	1	54646	20539	14969
4000	1	93395	26380	18643
5000	1	107386	41639	27531

表 3: x86 架构下的三种算法性能测试结果 (单位: 微秒)

观察以上数据我们发现，当考虑算法的运行速度时，CPU 的一级和二级缓存大小的确起着关键作用。本机 CPU 的一级缓存（L1 缓存）和二级缓存（L2 缓存）容量更大，相比之下，鲲鹏服务器具有更大容量的三级缓存（L3 缓存）。在大多数情况下，算法的优化更倾向于利用快速访问的一级和二级缓存，因为这些缓存与处理器更接近，可以更快获取数据，而不必经过更慢的主内存访问。这样一来，因为本机具有更大的一级和二级缓存时，算法能够更有效地利用缓存，所以在处理数据时会表现出更好的性能。当然，当鲲鹏服务器在处理大量数据和复杂任务时，仍然会体现出很好的整体性能表现。

5.2 任务二：n 个数求和

接下来讨论 n 个数求和的两种算法的差异。表4性能测试结果是在 x86 平台上的测验结果。为了更好实现算法，我们采用的实验数据规模均为 2 的 n 次幂，下表 n 表示幂的次数。

n	Ordinary	optimize
7	0.0006	0.0005
8	0.0012	0.001
10	0.0042	0.0033
12	0.0111	0.0102
14	0.0442	0.0404
16	0.1827	0.1658
20	2.6615	2.3938
24	43.2978	38.5235
28	654.608	596.833
29	1613.89	1426.84
30	2865.38	2353.43
31	76603.1	67701.2

表 4: x86 架构下的两种算法性能测试结果 (单位:ms)

由表中的数据可以看出，无论是链式累加的方法还是多链路展开的方法，由于都属于线性时间效率的方法，因此随着问题规模的翻倍，时间也近似翻倍。采用双链路展开的超标量优化方法的时间效率高于普通的链式累加方法，这是因为，通过多链路的方法，将相互联系的累加解耦成了两路不相关的问题，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，证明超标量优化确实起到了作用。但是同时，我们也注意到了几个特别现象，比如在 n 比较小时，两者的性能差别并不是很大，在 n=31 时，程序运行时间突然变得非常长，个人认为是此时的数规模太大，而导致内存空间需要过多使用，导致运行效率急剧下降。

有了以上对于算法性能测试的结果，接下来我们对其加速能力进行更细致地分析，并刨除 n=31 的异常值，表5是根据上表数据得到的加速比：

n	7	8	10	12	14	16	20	24	28	29	30
加速比（%）	16.7	16.7	21.4	8.14	8.60	9.25	10.06	11.03	8.83	11.59	17.87

表 5: x86 架构下优化算法对于普通算法的性能加速比 (单位:ms)

根据以上加速比变化，我们发现，加速比是在不停变化中的，在规模处于 n<10 时，加速比有一个较高的值，这个异常现象可能跟时间计算误差有关，而当数据规模 n>10 时，我们发现加速比就较为稳定，基本呈现缓步上涨的趋势，因此，我们选择在转折点和后面加速比稳定后最高的转折点用 VTune 来分析各级缓存的命中情况，结果如表6所示：

缓存	n=10	n=30
L1 cache 命中率	>99%	>99%
L2 cache 命中率	>99%	>84.24%
L3 cache 命中率	>99%	>71.63%

表 6: x86 架构下优化算法对于普通算法的性能加速比 (单位:ms)

由上表，我们发现问题规模在 n=10 时，所有的数据几乎全部在 L1 cache 命中，L2 cache 和 L3 cache 在过程中几乎没有访问，且命中率也几乎在 100%。而当问题规模 n>30 时，虽然 L1 cache 的命中率还是接近 100%，但是已经需要对二级和三级缓存进行多次访问，且二三级缓存的命中率均较低，这说明，除了对缓存进行访问，还会对内存产生多次访问，这将会极大的影响程序运行的时间，这也就不难理解为什么之后的运行速度会突然下降如此之多。

6 总结

6.1 实验总结

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。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能够在较大问题规模下具有很好的性能表现，其原因是能够充分利用 cache 的缓存，提高数据在缓存中的命中率，进而降低了访存导致的额外开销。在数组求和的实验中，我们采用了超标量的优化方法，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，这样能够充分利用 CPU 的多条流水线同时作业，能够明显提升程序性能。

6.2 任务完成总结

本次实验学习过程中，除了完成基本要求外，我还额外设计了运用循环展开技术（unroll）的代码，并且学习并使用华为云鲲鹏服务器完成代码性能测试，同时，对在不同平台（x86 平台和 ARM 平台）上运行时算法的性能进行了分析与比较，同时探索了现代计算机体系结构中 cache 和超标量对程序性能的影响，并用 VTune 工具帮助辅助对程序进行更细致的 profiling，并分析了 profiling 结果与性能表现间的关系。