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：刘宇轩

学号：2012677

专业：计算机科学与技术

2022 年 3 月 11 日

目录

1 问题重述

1.1 体系结构相关实验分析——cache 优化

计算给定 n*n 矩阵的每一列和给定向量的内积，考虑两种算法设计思路：

1. 逐列访问元素的平凡算法
2. cache 的优化算法

1.2 体系结构相关实验分析——超标量优化

计算 n 个数的和，考虑两种算法的设计思路

1. 逐个累加的平凡算法
2. 超标量优化算法，如最简单的两路链式累加，或两两相加后中间结果再两两相加的递归算法

2 实验环境

	ARM	x86
CPU 型号	华为鲲鹏 920 处理器	Intel Core i7-11800H
CPU 主频	2.6GHz	2.3GHz
L1 Cash	64KB	48K
L2 Cash	512KB	1.25MB
L3 Cash	48MB	24MB

3 实验设计及分析

3.1 cache 优化

3.1.1 实验设计

针对给定的问题，由于矩阵在内存中存储时按照行有限的顺序存储的，也就是说，在内存中的矩阵是按行紧密排列的。因此，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。

因此我们考虑改进算法，采用逐行访问的 cache 优化算法，即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。

同时，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，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，发挥 CPU 超标量计算的性能。

3.1.2 实验分析

为此，我们分别设计了三种算法，并在 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1所示

n	ordinary	potimize	n	ordinary	potimize	unroll	n	ordinary	potimize	unroll
10	0.0005	0.0005	100	0.051	0.048	0.035	1000	6.02	4.87	4.11
20	0.0019	0.0019	200	0.198	0.190	0.139	2000	26.81	20.32	17.71
30	0.0043	0.0042	300	0.455	0.437	0.351	3000	56.65	44.67	33.75
40	0.0080	0.0076	400	0.817	0.778	0.634	4000	224.93	83.84	69.20
50	0.0127	0.0125	500	1.371	1.204	1.069	5000	347.65	144.42	104.60
60	0.0172	0.0166	600	1.888	1.741	1.377	6000	518.29	205.09	136.93
70	0.0245	0.0237	700	2.457	2.192	1.820	7000	849.74	279.10	186.99
80	0.0301	0.0291	800	3.688	3.159	2.675	8000	1389.16	382.21	259.99
90	0.0402	0.0389	900	4.735	3.899	3.489	9000	1588.23	461.34	340.04

表 1: 不同规模下 ordinary,optimize 和 unroll 三种方法的运行时间

结合实验数据，绘制出在不同问题规模下的，三种方法的时间随问题规模的变化情况，如图3.1所示，并进行进一步分析

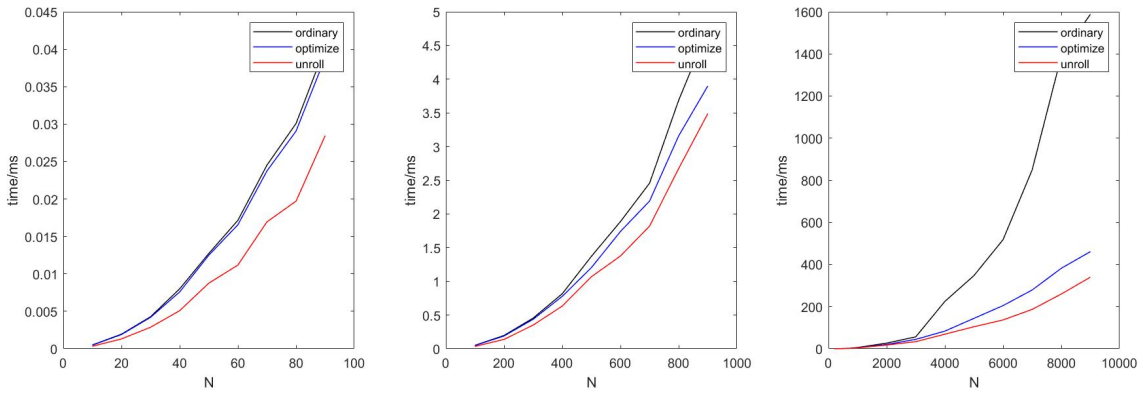

图 3.1: 算法时间随问题规模的变化

根据测试数据我们可以看出，在 N<300 的规模下，逐行或逐列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。而当 300<N<3000 时，逐行访问要比逐列访问的增长速度慢，但在此规模下，三种方法的增长趋势还是基本相同的。当 N>3000 之后，逐列访问的增长速度要明显超越了逐行访问的方法，证明此时 cache 优化起到了显著作用。

由数据可以进行理论分析，由于 CPU 的 L1-L3 的各级 cache 大小分别为 64KB，512KB 和 48MB，对于数组元素为 unsigned long long int 而言，每一个元素占据 10 个字节，因此，填满各级 cache 的元素规模大概在 80，200，2000。而图像的大概走势也恰恰能够印证这一假设，即在 N<300 下，尽管普通方法的 L1 cache 命中率可能已经很低，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，即两种逐行和逐列访问的方法效率差别不大。而当 300<N<3000 这个区间上时，L2 cache 的命中率也会不断下降，使得逐列访问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。而当数据规模超过 3000 之后，逐列访问的 L3 cache 命中率也会降到很低，也就是说逐列访问方法被迫到内存中去寻找数据，这导致的访存开销是非常大的，因此也使得两种方法的访问效率产生了显著的差异。

为了证明上述假设，我们分别采集了数据规模在 80，300 和 4000 的两组实验，通过 VTune 分析

其各级缓存的访存次数和命中率，如表2所示。当 N=80 时，可以看到，逐列访问方法的 L1 cache 未命中率要显著高于逐行访问，导致其 L2 cache 的访问次数增多，但 L2 cache 大部分情况命中，所以两种方法效率差距不大。当 N=300 时，逐列访问 L2 cache 未命中率要显著高于逐行访问，导致其 L3 cache 的访问次数也显著高于逐行访问，但 L3 cache 绝大情况下都已命中。而 L3 cache 的访问时间相对开销较大，因此两种方法的时间差异开始显现出来。当 N=4000 时，我们可以看到，逐列访问方法的的 L3 cache 访问次数要远远高出逐行访问的方法，而且 L3 cache 的未命中率也高达 27.1%，需要由大量的内存访问，时间开销会非常大，这也是两种方法产生巨大差距的主要原因。

ordinary						
n	L1 Hit	L1 Miss	L2 Hit	L2 Miss	L3 Hit	L3 Miss
80	7.81E+07	3.19E+05	3.11E+05	7.91E+03	2.64E+02	1.20E+01
300	1.45E+08	1.31E+07	1.28E+07	2.13E+05	1.63E+05	1.50E+02
4000	1.49E+08	1.58E+07	1.19E+07	3.93E+06	2.39E+06	8.89E+05
optimize						
n	L1 Hit	L1 Miss	L2 Hit	L2 Miss	L3 Hit	L3 Miss
80	9.14E+07	4.08E+03	3.86E+03	1.37E+02	1.20E+02	0.00E+00
300	1.83E+08	5.62E+04	5.11E+04	5.12E+03	9.36E+02	0.00E+00
4000	1.45E+08	8.02E+04	7.65E+04	3.76E+03	3.57E+02	8.70E+01

表 2: 不同规模下 ordinary 和 optimize 各级缓存访问次数和命中情况

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内利用多条流水线并行执行相同指令，因此能够在一定程度上优化代码运行效率。这一点通过比较两种方法的 CPI 也能够得到印证，如表3所示。

	ordinary	potimize
CPI	0.4975	0.4761

表 3: optimize 和 unroll 方法的 CPI 对比

3.2 超标量优化

3.2.1 实验设计

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。由于多链路方法使用了循环展开技术在一定程度上降低了循环的额外开销，为了保证实验的准确性，我们对普通的链式累加方法也要进行同样比例的循环展开，控制实验的可变因素，使得实验结果具有合理的对比性。

通过对比在不同实验规模下的两种方法的运行时间，探究优化加速比同问题规模的变化情况，并分析其中的内在原因。此外，还将会探究在 x86 架构下，Windows 和 Linux 两种系统对于处理同样规模的问题所需要消耗的绝对时间，以及优化的加速比的情况。

3.2.2 实验分析

为了方便算法的实现，我们的所有问题规模都取成 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，因此我们直接扩大了问题规模，分别测试了从 n=9 到 28 之间的 20 组数据，如表4所示。

n	ordinary	potimize	n	ordinary	potimize
9	0.0014	0.0009	19	1.4321	0.9576
10	0.0028	0.0018	20	2.8962	1.9128
11	0.0055	0.0035	21	5.7369	3.8739
12	0.0114	0.0070	22	13.064	9.5665
13	0.0221	0.0143	23	25.980	18.966
14	0.0445	0.0290	24	52.023	37.945
15	0.0891	0.0578	25	104.15	76.496
16	0.1780	0.1145	26	208.23	152.58
17	0.3573	0.2302	27	415.32	304.85
18	0.7140	0.4804	28	803.69	619.20

表 4: optimize 和 unroll 方法耗时随问题规模变化对比

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要显著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。为了能够证明超标量优化确实起到了作用，我们检测了链式累加方法和多链路累加方法的 CPI，如表5所示。根据表中的数据，可以得到，多链路累加的方法所达到的 CPI 要低于链式累加，也就是说在同一个时钟周期内，多链路累加方法执行的指令数要多于链式累加，这也证明了我们确实调用了 CPU 的多条流水线实现了超标量优化的目的。

	ordinary	optimize
CPI	0.667	0.496

表 5: 链式累加和多链路累加的 CPI 对比

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3.2所示。

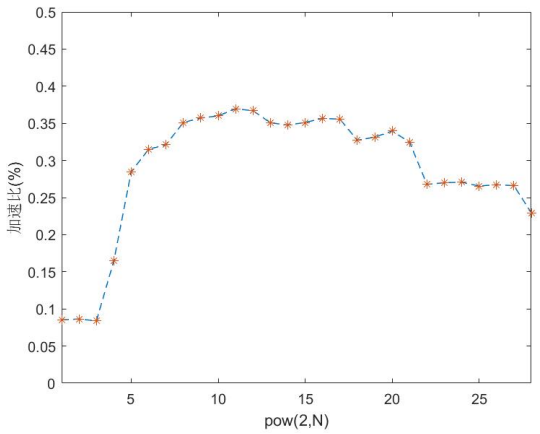

图 3.2: 优化加速比随问题规模的变化

通过实验数据可以发现，当问题规模较小的时候，算法的优化加速比比较低，但随着问题规模的增大，算法的优化加速比呈现先增加再保持最后降低的趋势。其增加的原因主要是由于多链路方法将累加拆分成了两个不相关的部分，能够利用 CPU 的超标量优化，使得算法获得逐渐增高的加速比。当问题问题规模 2^{10} 到 2^{17} 之间的时候，优化加速比基本保持一个稳定的状态，说明此时的超标量优化已经达到一个上限。而当问题规模超过 2^{17} 后，整体呈现一个下降的趋势，猜测是由于问题规模过大，导致缓存不足，由于需要经常进行内外存的访问，导致了较大的访存开销，而这个访存开销占据了程序运行的大部分时间，所以超标量优化的效果被一定程度上减弱。为了验证上述想法，我们利用 VTune 对下降最显著的点分析各级缓存的访问和命中情况，选取了 2^{22} 这个点分析，并与 2^{12} 这个最高点进行对比，如表6所示。

n	L1 cache 命中率	L2 cache 命中率	L3 cache 命中率
12	>99%	>99%	>99%
22	>99%	94.75%	82.16%

表 6: 不同问题规模下各级缓存的命中情况

通过数据可以得到，当问题规模在 2^{12} 时，所有的问题几乎全部在 L1 cache 命中，L2 cache 和 L3 cache 在过程中几乎没有访问，而且命中率也几乎在 100%。而当问题规模达到 2^{22} 时，虽然 L1 cache 的命中率还是接近 100%，但是已经有了很多的 L2 cache 和 L3 cache 访问，而且 L3 cache 的命中率只有 80%，会出现很多的内存访问，这将会极大的影响程序运行的时间，因此印证了我们上面的猜测。

4 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。实验的相关代码已上传GitHub。