南開大學
Nankai University

计算机学院

并行程序设计实验报告

体系结构相关及性能测试

姓名：黄天昊

学号：2011763

专业：计算机科学与技术

2023 年 3 月 12 日

目录

1 平台配置

本实验共在 2 个平台上测试运行，分别是 Windows+x86 和 Linux+x86。下面给出 2 个平台的配置：

- CPU 型号：Intel(R) Core(TM) i5-10210U
- CPU 架构：x86_64
- CPU 频率：min:1.60GHz, max:2.11GHz
- L1 cache：256KB
- L2 cache：1.0MB
- L3 cache：6.0MB
- 内存大小：16384MB RAM

(a) Windows+x86

- CPU 型号：Intel(R) Core(TM) i5-10210U
- CPU 架构：x86_64
- CPU 频率：2112.006MHz
- L1d cache：32768 Byte
- L1i cache：32768 Byte
- L2 cache：262144 Byte
- L3 cache：6291456 Byte
- 内存大小：12952056 kB

(b) Linux+x86

图 1.1: 测试平台配置

2 Cache 优化

2.1 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，并进行实验对比：

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。

2.2 实验设计

2.2.1 初始化

对于本问题，测试数据人为设定固定值即可。需要注意的是，测试数据过于特别可能无法得到程序真实性能。以本题为例，若向量 a 的第一个元素为 0，那么在该向量和矩阵 b 相乘时，矩阵 b 第一行的所有元素的值全部失去意义。又例如，当元素值全为整数时程序计算用时会相对较少，但这不符合绝大多数应用场景。因此，我们采用 float 浮点类型数据来初始化矩阵和向量。

2.2.2 规模

问题规模的确定因素为执行时间和执行次数。本实验的规模设计为：对于 n (0,1000]，以 100 为跨度进行精细测试；对于 n (1000,10000]，以 1000 为跨度进行稀疏测试。

2.2.3 时间

测量工具 虽然 clock() 函数可以跨平台使用，但由于其精度不高，我们分别在 Windows 和 Linux 平台中使用 QueryPerformance 和 gettimeofday()。具体头文件和使用格式见代码。

提高精度 当问题规模很小时，即使高精度计时器也难以得到准确的时间，此时需要重复多次测量后取平均值。重复测量的次数随 n 的增大而减小，我们使用 freq 来设置重复测量的次数（见代码）。

2.3 程序测试结果及分析

我们分别在 Windows+x86 环境、Linux+x86 环境两个环境下测试。

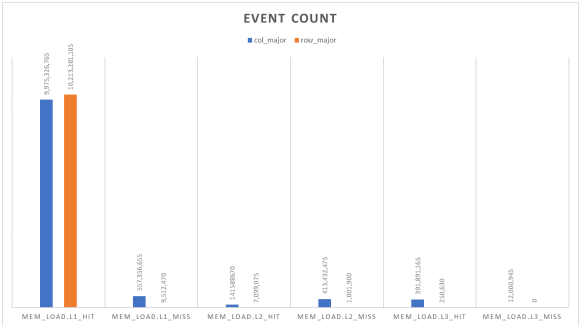
2.3.1 Windows+x86

实验结果 对于该环境，我们只需在 CodeBlocks 中使用 GCC 编译器 Build-Run 即可看到结果，如图2.2所示。值得注意的是，为防止编译器自动优化而无法得到真实结果，需在 CodeBlocks 的 Compiler Settings 中取消其默认勾选 [-O2] 和 [-s]。

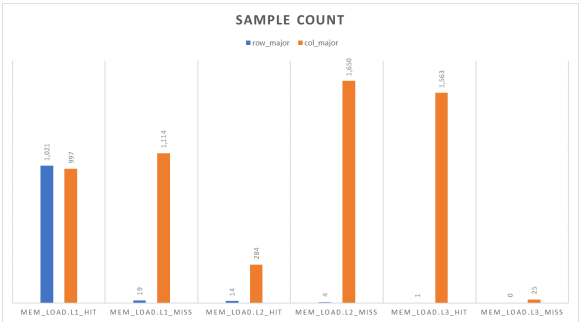
	100	200	300	400	500	600	700	800	900	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
优化前	0.0401852	0.157177	0.322898	0.700935	1.23416	1.64875	2.38179	3.21216	4.05181	6.3763	42.7834	107.644	308.845	282.387	419.3	577.034	767.75	979.731	1264.45
优化后	0.0422368	0.142107	0.290688	0.538691	0.854908	1.18513	1.60363	2.11199	2.7798	3.5701	15.4442	32.412	59.9456	86.1273	124.655	164.444	209.01	256.726	329.863

图 2.2: 在 Windows+x86 环境下执行结果

分析 对于 Windows+x86 平台，我们使用 Vtune 分析 Cache 命中率。主要关注三级 Cache：L1、L2 和 L3，以及它们的 HIT 和 MISS。具体见图2.3。可以看到，在 Event Count 中优化后算法在 L1 Cache



(a) Event Count



(b) Sample Count

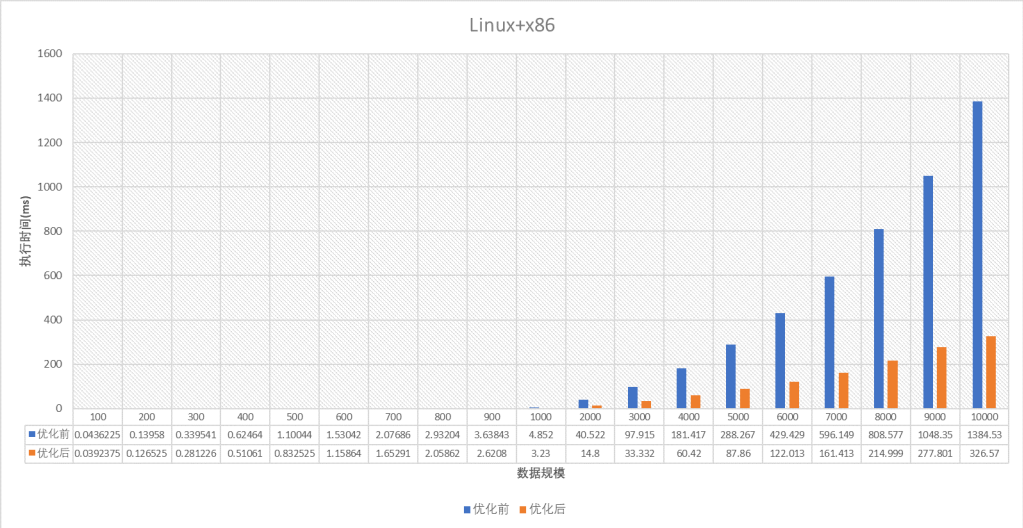
图 2.3: Vtune 分析结果

的 HIT 高于优化前，MISS 在各级 Cache 都低于优化前；在 Sample Event 中优化后算法在 L1 HIT

中和优化前相差不大，并在 L2 HIT 和 L3 HIT 中远高于优化前。由于 HIT 代表预测稍后步骤的能力，HIT 越高程序执行效率越高，故 Vtune 分析结果符合实验结果，即优化后的算法效率更高。

2.3.2 Linux+x86

实验结果 为测试该环境，我首先在 CodeBlocks 中写好代码，并在本机下载了 WSL 和 Ubuntu。这样就可以在 Ubuntu 中使用 g++ 命令编译.cpp 文件，并在 Linux+x86 环境下完成测试。在 Ubuntu 中进入目标文件夹后，编译命令为 g++ -O0 main.cpp -o lab1_linux。与 Windows+x86 环境相同，我们要使用-O0 指令取消所有优化，以得到最真实的实验结果。编译成可执行文件 lab1_linux 后，使用./命令运行可执行文件。最终实验结果如图2.4所示。



	100	200	300	400	500	600	700	800	900	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
优化前	0.0436225	0.13958	0.339541	0.62464	1.10044	1.53042	2.07686	2.93204	3.63843	4.852	40.522	97.915	181.417	288.267	429.429	596.149	808.577	1048.35	1384.53
优化后	0.0392375	0.126525	0.281226	0.51061	0.832525	1.15864	1.65291	2.05862	2.6208	3.23	14.8	33.332	60.42	87.86	122.013	161.413	214.999	277.801	326.57

图 2.4: 在 Linux+x86 环境下执行结果

3 超标量优化

3.1 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。

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。

3.2 实验设计

3.2.1 整体规划

本实验共 4 种算法，分别是平凡算法（链式）、多链路式算法、递归算法和二重循环算法，其中后 3 种均为优化算法。设计规划如下：首先固定问题规模，然后执行并测试不同算法；然后逐渐增大问题规模。随后比较 Windows 和 Linux 平台上执行程序的性能，并探讨编译器不同优化力度对性能的影响。

3.2.2 初始化

与前一实验相同，我们人为指定浮点数作为数组元素。除此之外需要注意的是，数组大小 n 需要取为 2 的幂，这是因为归约的优化算法需要反复二分，若 n 不是 2 的幂则必须处理边界，否则会出现错误的计算结果。另外，由于递归和二重循环的优化算法会改变原有数组，为保证不同算法处理的数组相同从而控制变量，需要在每次调用各算法之前刷新一遍数组。

3.2.3 循环展开 (unroll)

由于几个算法实现的基本方式都是循环，额外开销很大，故采用循环展开 (unroll) 策略，在每个循环步进行多次加法运算，从而降低比较操作的执行次数。由于本例中问题规模始终是 2 的幂，我们采用 4 为步长进行展开。具体代码见 lab2_win 和 lab2_linux 中 main.cpp 里对数组的初始化等。

3.2.4 规模

与 Cache 优化实验相比较，本实验在 $n < 10000$ 时得到的执行时间极短，因此考虑将问题规模扩大到 2000000 以内。此外，n 必须是 2 的幂。

3.2.5 时间

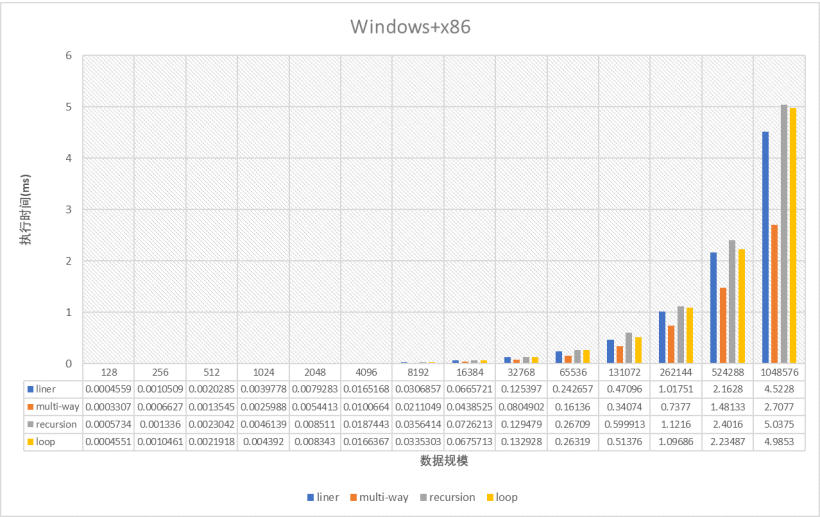
由于执行时间非常短,我们需要比 Cache 优化中更多的重复次数,因此我将重复次数设置为 2000000/n。

3.3 程序测试结果及分析

我们分别在 Windows+x86 环境、Linux+x86 环境下测试。

3.3.1 Windows+x86

实验结果 执行过程和 Cache 优化实验完全相同，因此不再赘述。程序执行结果如图3.5所示。



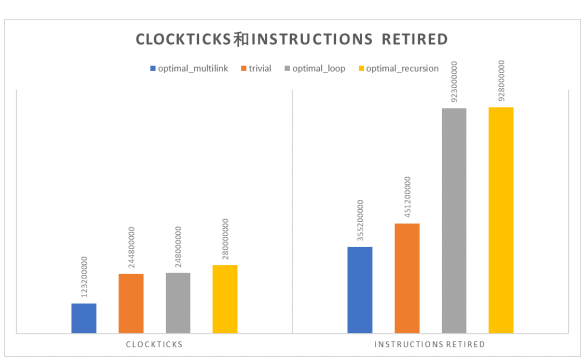
	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576
liner	0.0004559	0.0010509	0.0020285	0.0039778	0.0079283	0.0165168	0.0306857	0.0665721	0.125397	0.242657	0.47096	1.01751	2.1628	4.5228
multi-way	0.0003307	0.0006627	0.0013545	0.0025988	0.0054413	0.0100664	0.0211049	0.0438525	0.0804902	0.16136	0.34074	0.7377	1.48133	2.7077
recursion	0.0005734	0.001336	0.0023042	0.0046139	0.008511	0.0187443	0.0356414	0.0726213	0.129479	0.26709	0.599913	1.1216	2.4016	5.0375
loop	0.0004551	0.0010461	0.0021918	0.004392	0.008343	0.0166367	0.0335303	0.0675713	0.132928	0.26319	0.51376	1.09686	2.23487	4.9853

图 3.5: 在 Windows+x86 环境下执行结果

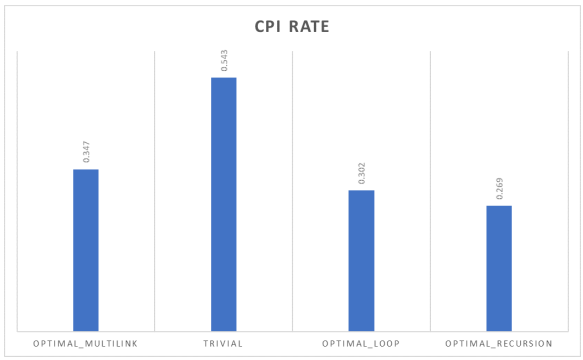
可以看到，多路链式的优化算法最好，其效率约为优化前的 2 倍；递归函数和二重循环的优化算法在该环境下效率低于原算法，我认为这是由于递归函数调用在指令的控制转移、压栈和恢复栈等步

骤上开销过大，且二重循环相比一重循环复杂度更大所导致。同时我认为还与平台环境有关，比如接下来的 Linux+x86 环境下递归和二重循环的优化方法都实现了优化，达到了更高效率。

分析 使用 Vtune 进行分析，对于我们主要关注总体执行的周期数 Clockticks、执行指令数 Instructions Retired 和 CPI，也就是每条指令执行的周期数，如图3.6所示。可以看到，Vtune 的分析结果符合实



(a) Clockticks 和 Instructions Retired



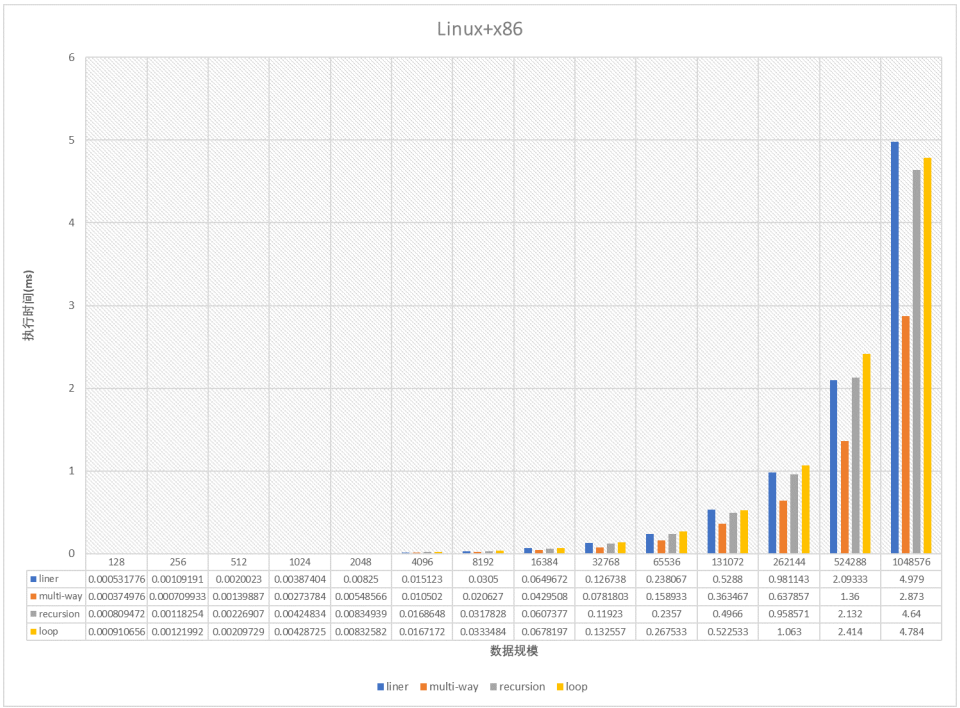
(b) CPI

图 3.6: Vtune 分析结果

验结果，这也解释了为什么多路链式算法最优越，而递归和二重循环算法并不优越。执行周期数越少、指令数越少、每条指令执行的周期数越少，算法越优越。可以看到的是，递归和二重循环算法的优越性体现在了它们的 CPI 上，它们的 CPI 甚至低于多路链式算法。

3.3.2 Linux+x86

实验结果 执行过程和 Cache 优化实验完全相同，因此不再赘述。程序执行结果如图3.7所示。



	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576
liner	0.000531776	0.00109191	0.0020023	0.00387404	0.00825	0.015123	0.0305	0.0649672	0.126738	0.238067	0.5288	0.981143	2.09333	4.979
multi-way	0.00037496	0.000709933	0.00139887	0.00273784	0.00548566	0.010502	0.020627	0.0429508	0.0781803	0.158933	0.363467	0.637857	1.36	2.873
recursion	0.000809472	0.00118254	0.00226907	0.00424834	0.00834939	0.0168648	0.0317828	0.0607377	0.11923	0.2357	0.4966	0.958571	2.132	4.64
loop	0.000910656	0.00121992	0.00209729	0.00428725	0.00832582	0.0167172	0.0333484	0.0678197	0.132557	0.267533	0.522533	1.063	2.414	4.784

图 3.7: 在 Linux+x86 环境下执行结果

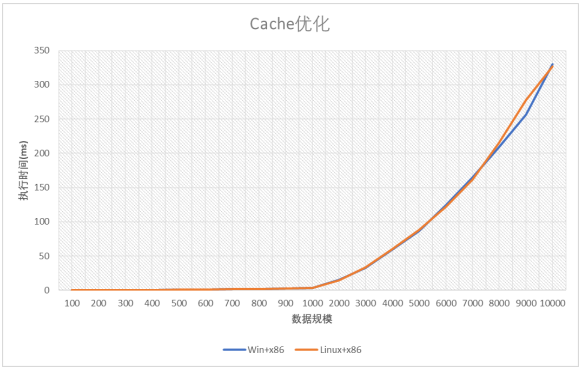
可以看到，在 Linux+x86 环境下程序的执行结果最理想，3 个优化算法都达到了最终目的，都比原算法效率高。这再次证明不同环境下执行同一个程序可能造成不同的结果。

4 性能对比

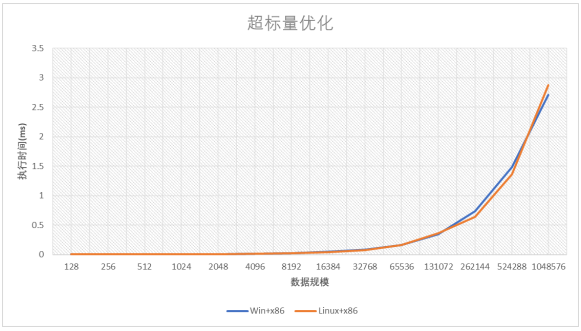
4.1 操作系统对比

4.1.1 Windows 和 Linux

本着控制变量的原则，我们控制指令集架构为 x86，并使用两组实验的结果进行比较。Cache 优化实验中我们选择优化后算法，超标量优化实验中选择多路链式算法。经统计，结果如图4.8所示。可以



(a) Cache 优化



(b) 超标量优化

图 4.8: Windows 和 Linux 对比

看到，Windows 和 Linux 操作系统在两个实验中的表现差异并不明显，在数据规模较小时 Linux 系统占据细微优势，数据规模较大时 Windws 系统占据优势。

4.2 平台对比

由于鲲鹏服务器还不能使用，所以目前并不能进行这项研究，之后如果条件具备再补上这部分实验。

5 代码链接

GitHub 链接为https://github.com/Skyyyy0920/Parallel-Programming