南開大學
Nankai University

计算机学院

并行程序设计第 **2.1** 次作业

矩阵与向量内积

姓名：丁屹

学号：2013280

专业：计算机科学与技术

2022 年 3 月 14 日

目录

CPU Maximum Frequency	2600 MHz
CPU Minimum Frequency	200 MHz
L1d 缓存	64 KiB
L1i 缓存	64 KiB
L2 缓存	512 KiB
L3 缓存	49152 KiB
内存大小	191.3 GiB

表 1: 鲲鹏 arm 平台硬件配置信息

1 问题

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

2 程序实现

源码链接：https://github.com/ArcanusNEO/Parallel-Programming/tree/master/1/0
头文件位于 inc/，源文件位于 src/

逐列访问平凡算法

```
for (int i = 1; i <= matrix.col(); ++i)
  for (int j = 1; j <= matrix.row(); ++j)
    ans += (long long) matrix(j, i) * vec(i);
```

cache 优化算法

```
for (int i = 1; i <= matrix.row(); ++i)
  for (int j = 1; j <= matrix.col(); ++j)
    ans += (long long) matrix(i, j) * vec(j);
```

- 为了便于调整数据规模的同时保证数据分布紧凑，矩阵采用一维数组模拟，封装到 matrix_t 类中，使用 operator() 访问元素
- 使用 C++11 的 chrono::high_resolution_clock 高精度计时函数测量运行时间
- ordinary 采用逐列访问的平凡算法
- cache 采用 cache 优化算法
- 使用 cmake 构建

3 实验平台配置

华为鲲鹏 arm 平台部分硬件参数如表 1 所示。arm 服务器系统环境为 4.14.0 内核的 openEuler，本次实验使用基于 clang 的华为 bisheng 编译器构建。

CPU 型号	AMD Ryzen 7 4800HS with Radeon Graphics
CPU Maximum Frequency	2900 MHz
CPU Minimum Frequency	1400 MHz
L1d 缓存	256 KiB
L1i 缓存	256 KiB
L2 缓存	4 MiB
L3 缓存	8 MiB
内存大小	16 GiB

表 2: 本地 x86 平台硬件配置信息

本地 x86 平台部分硬件参数如表 2 所示。本地 x86 系统环境为 5.16.14 内核的 Arch Linux，本次实验使用 GNU GCC 编译，并且使用 perf 进行性能测试。

4 实验方案设计

4.1 测试脚本

测试脚本位于 bin/，”run” 是 x86 架构下的脚本，”run-pbs” 是鲲鹏服务器平台的脚本

4.2 测试数据

使用 gen-data 生成数据，规模 n 作为第一个参数传入，使用 mt1937 生成随机数。
生成了一组测试文件位于 res/，文件名形如 n.in
使用 conf/in.conf 配置输入数据路径和重复测试次数，其路径作为待测程序的第一个参数传入。

4.3 测试方法

分别测试 ordinary 和 cache 两个程序。读入定义测试输入文件路径和重复测试遍数的配置文件以便自动完成测试。程序会向标准输出打印测试结果。

重复测试代码

```
for (int _counter = 0; _counter < T; ++_counter) { // T 为重复测试遍数
  ans      = 0;
  auto t1 = chrono::high_resolution_clock::now();
  func(ans, matrix, vec);
  auto t2  = chrono::high_resolution_clock::now();
  auto sec = chrono::duration_cast<chrono::duration<double>>(t2 - t1);
  ret += sec.count() / T;
}
```

n	repeat	ordinary O0	cache O0	ordinary O1	cache O1	ordinary O2	cache O2	ordinary O3	cache O3
10	1000	0.000003921510	0.000003898680	0.000002001930	0.000001964040	0.000000327890	0.000000339470	0.000000319410	0.000000344110
20	1000	0.000014620950	0.000014602760	0.000007023470	0.000006842360	0.000000501310	0.000000657150	0.000000467850	0.000000641370
30	1000	0.000032228360	0.000032184710	0.000015241810	0.000014904280	0.000000763690	0.000000774170	0.000000691040	0.000000777670
40	950	0.000056860989	0.000056846558	0.000026680400	0.000026098295	0.000001164211	0.000001254295	0.000001005126	0.000001245632
50	900	0.000089057611	0.000088343433	0.000041381567	0.000040420867	0.000001678011	0.000001547344	0.000001410889	0.000001549533
60	850	0.000128093988	0.000127123306	0.000059313106	0.000057921706	0.000002270624	0.000002428812	0.000001873753	0.000002413847
70	800	0.000173778637	0.000172767013	0.000080434662	0.000078577600	0.000003020475	0.000002645638	0.000002430088	0.000002644038
80	750	0.000227279400	0.000224837240	0.000104802547	0.000102348533	0.000004033467	0.000003892293	0.000003213960	0.000003860040
90	500	0.000286074980	0.000285214100	0.000132402900	0.000129312160	0.000005021300	0.000004114980	0.000003812880	0.000004122680
100	100	0.000353027200	0.000351141100	0.000163288700	0.000159662300	0.000005953000	0.000005611200	0.000004950000	0.000005456300
200	100	0.001409264900	0.001407091700	0.000649411800	0.000634192400	0.000023487600	0.000021635600	0.000020187600	0.000021334400
300	100	0.003158901300	0.003158875000	0.001459003600	0.001424701100	0.000049816900	0.000046524400	0.000046958800	0.000045225600
400	75	0.005616461067	0.005714338133	0.002596611067	0.002590574133	0.000097343200	0.000084646933	0.000085758667	0.000082299600
500	50	0.008824118800	0.008933916800	0.004065168800	0.004048300400	0.000160591000	0.000133844200	0.000149521400	0.000131648000
750	25	0.020230814800	0.020136468000	0.009124470400	0.009196836800	0.000597472000	0.000290120400	0.000539312400	0.000290214000
1000	10	0.035702858000	0.036474819000	0.016245198000	0.016802713000	0.000869622000	0.000562403000	0.000798831000	0.000522889000
2000	10	0.145383594000	0.150944694000	0.065200668000	0.071015338000	0.004459454000	0.002142993000	0.004385257000	0.002140263000
3000	5	0.336490798000	0.349886098000	0.156467236000	0.168448982000	0.012147040000	0.005809428000	0.011584076000	0.006152252000
2500	5	0.228733876000	0.240887894000	0.102909028000	0.113171246000	0.007623726000	0.003432526000	0.007216164000	0.003632204000
3500	5	0.458229866000	0.478521286000	0.208403398000	0.230962040000	0.015620854000	0.007486212000	0.014909178000	0.007414828000
4000	5	0.589321022000	0.625729100000	0.264033642000	0.301310122000	0.023590688000	0.009625902000	0.023382362000	0.009711832000

表 3: 鲲鹏 arm 平台不同优化等级下的测试结果（时间单位：s）

5 实验结果及分析

5.1 鲲鹏 arm 平台

通过横向比较表 3 的优化级别可以得知，对于平凡算法，优化级别从 O0 到 O1 产生了 1 倍多的性能提升，从 O1 到 O2 能产生大约了 10 倍多的性能提升，而 O3 相对于 O2 的性能提升幅度很小，效果不明显。对于 cache 优化算法的结论也类似。

通过横向比较表 3 两种算法可以得知，对于 O0 优化，两种算法性能差别不大，甚至 cache 优化算法在大数据量下处于劣势。

在小数据量下，两种算法性能差别同样不大。

在中等数据量下，各种优化等级 cache 优化算法相比于平凡算法都有些微的性能优势。

在大数据量下，对于 O0 和 O1 优化，cache 优化算法相比于平凡算法处于劣势；对于 O2 和 O3 优化，cache 优化算法相比于平凡算法有 1 倍有余的性能提升。

通过纵向比较表 3 可以得知，由于算法使用 int 存储矩阵元素，$4 \times 100 \times 100$ B $<$ 64 KiB = L1d size $< 4 \times 200 \times 200$ B，所以在 n = 200 时矩阵大小突破了 L1d 缓存大小，相对于 n = 100 数据规模翻 4 倍，运行时间成 4 倍有余；而 n = 100 与 n = 50 相比，运行时间恰好大约成 4 倍。

$4 \times 300 \times 300$ B $<$ 512 KiB = L2 size $< 4 \times 400 \times 400$ B，所以在 n = 400 时矩阵大小突破了 L2 缓存大小，此时低优化级别下 cache 优化算法和平凡算法性能差别不大，而高优化级别下 cache 优化算法对比于平凡算法有较大性能优势。

5.2 本地 AMD x86 平台

由表 4 可以看到 cache 优化算法对于平凡算法的优势随着数据量增大而变大，大数据量下有近一倍的差距。

n	repeat	ordinary O3	cache O3
10	1000	0.000001673545	0.000001451582
20	1000	0.000001366916	0.000001404933
30	1000	0.000001596258	0.000001526239
40	950	0.000001965164	0.000001941371
50	900	0.000002400503	0.000002393746
60	850	0.000003021801	0.000002977994
70	800	0.000003590614	0.000003584309
80	750	0.000004489092	0.000004473932
90	500	0.000005746814	0.000005276076
100	100	0.000015520090	0.000014127370
200	100	0.000022226890	0.000021095470
300	100	0.000054151200	0.000046500850
400	75	0.000097331360	0.000082775467
500	50	0.000154308000	0.000128123180
750	25	0.000403876120	0.000289532600
1000	10	0.000932460100	0.000527480600
2000	10	0.003278459500	0.002260003800
2500	5	0.008431128400	0.003538072200
3000	5	0.009731245600	0.004834529600
3500	5	0.015965485400	0.006590205400
4000	5	0.014293912400	0.008455307800

表 4: 本地 AMD x86 平台 O3 优化下的测试结果（时间单位：s）

5.2.1 使用 perf + hotspot 分析 cache 命中率

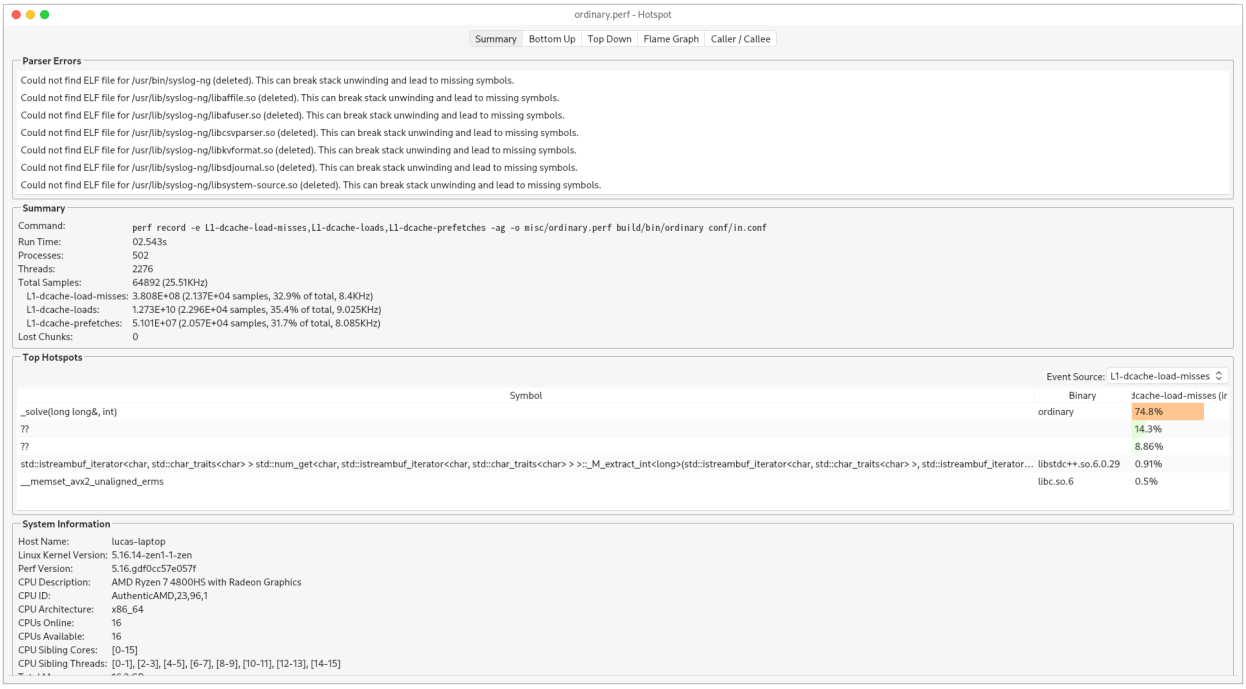

图 5.1: 平凡算法的总体 cache 命中率

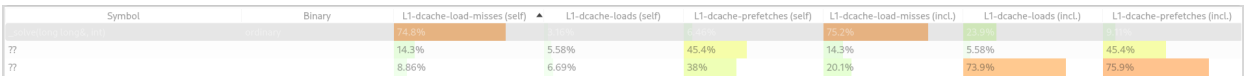

图 5.2: 平凡算法的 _solve 函数 cache 命中率

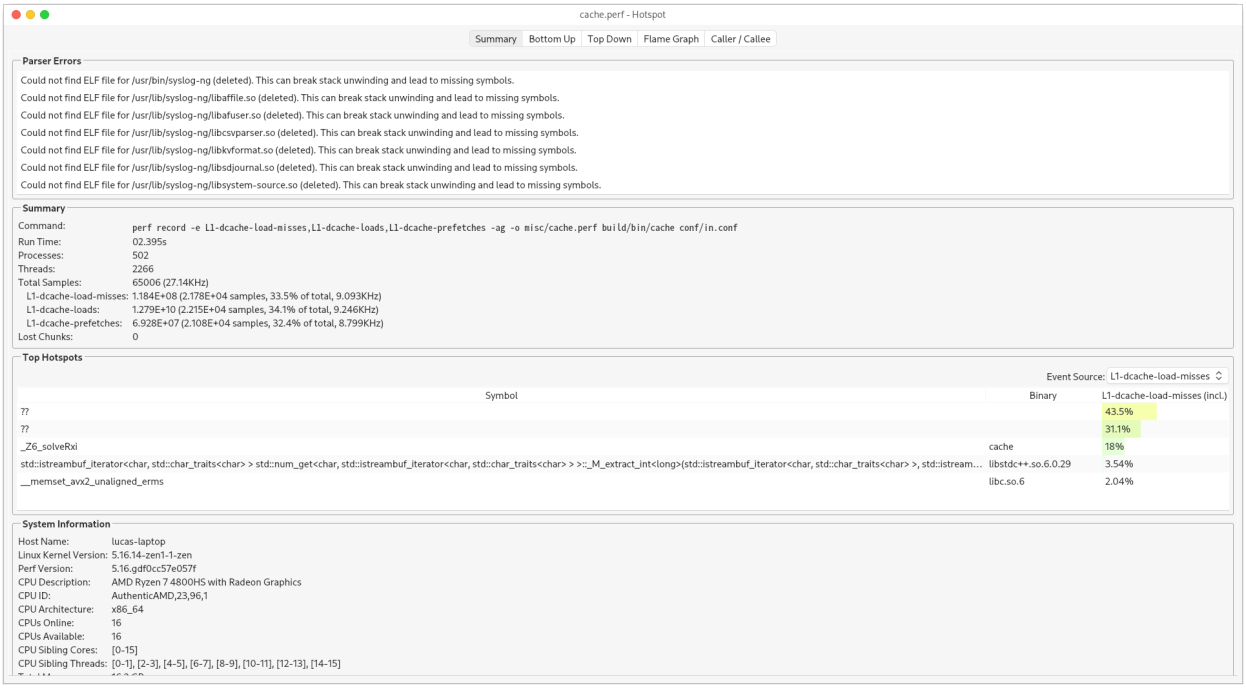

图 5.3: cache 优化算法的总体 cache 命中率

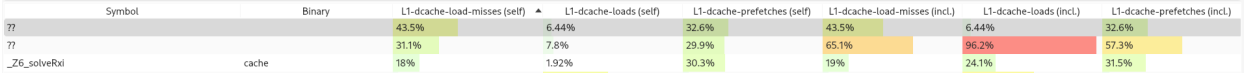

图 5.4: cache 优化算法的 _solve 函数 cache 命中率

n	repeat	x86 ordinary	x86 cache	arm ordinary	arm cache
10	1000	0.000001673545	0.000001451582	0.000000319410	0.000000344110
20	1000	0.000001366916	0.000001404933	0.000000467850	0.000000641370
30	1000	0.000001596258	0.000001526239	0.000000691040	0.000000777670
40	950	0.000001965164	0.000001941371	0.000001005126	0.000001245632
50	900	0.000002400503	0.000002393746	0.000001410889	0.000001549533
60	850	0.000003021801	0.000002977994	0.000001873753	0.000002413847
70	800	0.000003590614	0.000003584309	0.000002430088	0.000002644038
80	750	0.000004489092	0.000004473932	0.000003213960	0.000003860040
90	500	0.000005746814	0.000005276076	0.000003812880	0.000004122680
100	100	0.000015520090	0.000014127370	0.000004950000	0.000005456300
200	100	0.000022226890	0.000021095470	0.000020187600	0.000021334400
300	100	0.000054151200	0.000046500850	0.000046958800	0.000045225600
400	75	0.000097331360	0.000082775467	0.000085758667	0.000082299600
500	50	0.000154308000	0.000128123180	0.000149521400	0.000131648000
750	25	0.000403876120	0.000289532600	0.000539312400	0.000290214000
1000	10	0.000932460100	0.000527480600	0.000798831000	0.000522889000
2000	10	0.003278459500	0.002260003800	0.004385257000	0.002140263000
2500	5	0.008431128400	0.003538072200	0.011584076000	0.006152252000
3000	5	0.009731245600	0.004834529600	0.007216164000	0.003632204000
3500	5	0.015965485400	0.006590205400	0.014909178000	0.007414828000
4000	5	0.014293912400	0.008455307800	0.023382362000	0.009711832000

表 5: 本地 AMD x86 平台 O3 优化下的测试结果（时间单位：s）

由图 5.1、5.2 可以看到，平凡算法的 _solve 函数中 cache miss 率高达 75%；而在图 5.3、5.4 中可以看到，cache 优化算法的 _solve 函数中 cache miss 率仅有 18%。所以显然可以得出结论，优化前后 cache 命中率的差异巨大。

5.3 交叉对比

从表 5 可以看到 arm 平台对比 x86 平台在小数据量上有所领先，而在大数据量上则是 x86 平台略有优势。可能的原因是：本地 x86 的 L1、L2 缓存较大，相应的延迟相对较高，吞吐率相对更大，小数据量表现不如大数据量好

对于 cache 优化算法 x86 平台略有优势，而对于平凡算法则是 arm 表现更好。可能的原因是：本地 x86 的 L1、L2 缓存延迟相对更高，吞吐率相对更大，对于 cache 优化算法吞吐率优势得以发挥，对于平凡算法延迟劣势凸显。

6 参考文献

[1][3][2][5][6][4]

参考文献

[1] Alexis Zhang. Apple m1 wikipedia. https://zh.wikipedia.org/wiki/Apple_M1, 2020.

[2] Andrei Frumusanu. The 2020 mac mini unleashed: Putting apple silicon m1 to the test. https://www.anandtech.com/print/16252/mac-mini-apple-m1-tested, 2020.

[3] Andrei Frumusanu. Apple announces the apple silicon m1: Ditching x86 - what to expect, based on a14. https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive, 2020.

[4] Erik Engheim. Why is apple' s m1 chip so fast? https://debugger.medium.com/why-is-apples-m1-chip-so-fast-3262b158cba2, 2020.

[5] Veedrac. Measures microarchitectural details. https://github.com/Veedrac/microarchitecturometer, 2020.

[6] 木头龙. 如何看待苹果 m1 芯片跑分超过 i9？. https://www.zhihu.com/question/429951450, 2020.