南開大學
Nankai University

计算机学院

并行程序设第一次实验

体系结构相关及性能测试

姓名：田佳业

学号：2013599

专业：计算机科学与技术

2022 年 3 月 15 日

目录

1 体系结构相关实验分析——cache 优化

1.1 实验任务

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，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。

1.2 题目分析

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;cache 优化算法则改为逐行访问矩阵元素，一步外层循环计算不出任何一个内积，只是向每个内积累加一个乘法结果。后者的访存模式具有很好空间局部性，令 cache 的作用得以发挥。

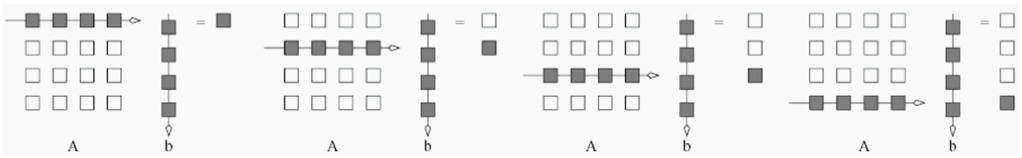


图 1.1: cache 优化算法

1.3 实验思路

实验数据设计: 实验数据采取了人为设置的数据。为了尽可能避免数值大小分布不均匀或数值过大可能对不同规模测试造成的影响的同时保持计算的复杂性，优化了初始值的设定方式：对于问题的规模 N，列向量 $a[i] = N - i$，矩阵根据行的奇偶性设置不同规模的值，具体为：$b[i][j] = (i/2 == 0)?(i + j) : (i - j)$.

实验规模设计: 根据实际运行情况对两种算法规模 N 从 1000 到 10000 之间的运行情况进行了比较，同时对小规模数据与 cache 大小的关系进行分析。

1.4 实验方案

1. 首先在 x86 架构，windows 操作系统上使用 QueryPerformance 和 gettimeofday() 两种方法进行测试后，发现两种计时方法都能有效测量程序时间且结果没有明显差距后，考虑到代码的统一性，选择了跨平台的 gettimeofday() 进行计时，比较了两种算法的运行效率，并用 Vtune 对 cache 命中率进行了分析。

2. 之后在 x86 架构，ubuntu 操作系统通过 perf 对 cache 命中率进行了分析，印证了 Vtune 的结果；同时探究了不同编译优化力度对程序性能的影响。

2. 最后在 ARM 架构，鲲鹏服务器上进行了作业的提交和性能测试，并在同样为 ARM 架构的苹果 M1 芯片上（通过 homebrew 包管理工具安装 gcc 编译器，使用 clion 作为 IDE）进行了测试，并三者的表现进行了分析和比较。

1.5 平台信息和配置

x86 windows: Intel i7 6 核，三级缓存分别为 384KB,1.5MB,12MB。

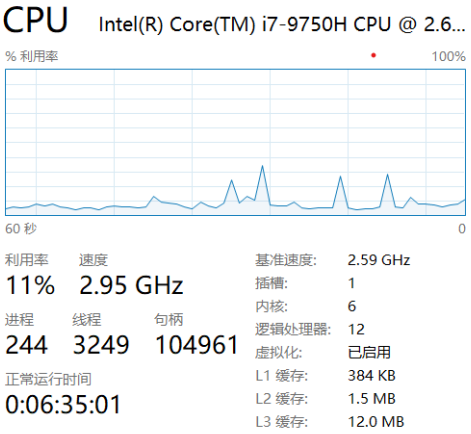


图 1.2: x86 平台 CPU 配置

ubuntu 虚拟机： Intel i7 单核，L1d(数据缓存）和 L1i（指令缓存）均为 32 KB，L2 缓存 256 KB，L3 缓存 12 MB。

鲲鹏服务器： aarch64 对于单核 L1d 和 L1i 均为 64 KB，L2 cache 为 512KB，L3 cache 为 49152KB

macbook： Apple M1，内置 8 核 CPU，集成 4 个高性能大核心以及 4 个高效能小核心。由于 M1 芯片作为较新的芯片，目前多方查阅文献并未得到其详细数据。但据其他非官方来源（AnandTech）[1][2] 显示，关于 M1 的 cache 组成如下：

大核心 L1d: 每核 128KB，共 512KB

大核心 L1i: 每核 192KB，共 768KB

小核心 L1d: 每核 64KB，共 256KB

小核心 L1i: 每核 96KB，共 384KB

大核心 L2:12MB 共用

小核心 L2:4MB 共用

SLC Cache(类似于三级缓存）16MB

相比之下，苹果 M1 的二级缓存是非常大的，同时 192KB 的 L1 指令缓存，是 ARM 公版设计的 3 倍，x86 现有设计的 6 倍。因此，作为新的 ARM 架构芯片,M1 有着独特的处理器设计和性能表现。

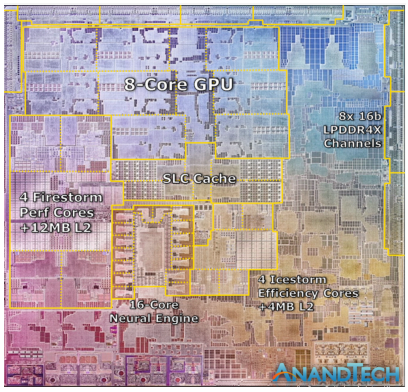


图 1.3: M1 芯片

1.6 实验结果

1.6.1 性能比较

在 Intel x86 架构和 MacBook ARM 架构上分别对两种算法规模 N 从 1000 到 10000 之间的运行情况进行了比较，考虑到实验结果可能具有的偶然性，对每个规模重复运行 3 次取平均值，得到结果如下

在 Intel x86 架构上：

表 1: intel x86 执行时间

规模	平凡算法 (ms)	优化算法 (ms)	优化比率
1000	5.292	3.540	0.6689
2000	27.908	8.775	0.3144
3000	71.575	19.765	0.2761
4000	143.980	35.111	0.2438
5000	222.159	55.161	0.2482
6000	305.396	80.975	0.2651
7000	455.813	107.082	0.2349
8000	681.533	140.447	0.2060
9000	877.176	179.025	0.2040
10000	1155.67	217.169	0.1879

在 M1 ARM 架构上：

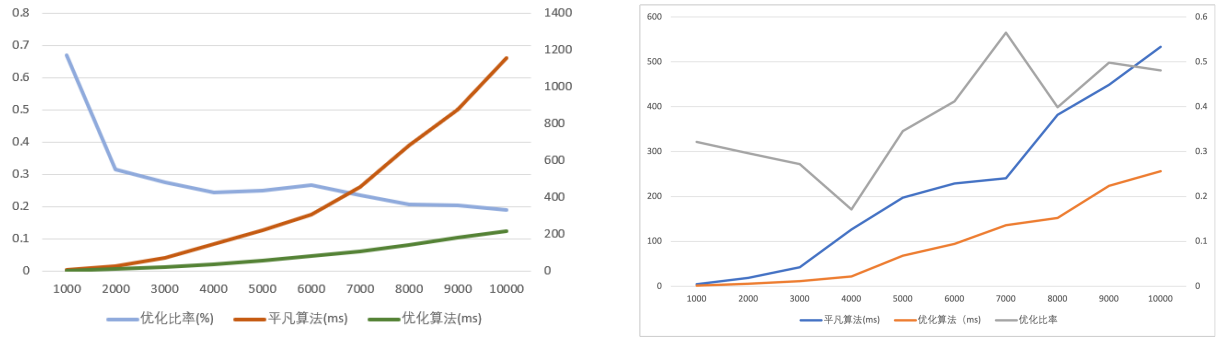
表 2: mac M1 执行时间

规模	平凡算法 (ms)	优化算法 (ms)	优化比率
1000	4.336	1.394	0.3214
2000	18.737	5.549	0.2961
3000	42.121	11.465	0.2721
4000	126.315	21.651	0.1714
5000	197.486	68.312	0.3459
6000	228.851	94.328	0.4122
7000	240.656	135.974	0.5650
8000	382.215	152.362	0.3986
9000	449.058	223.688	0.4981
10000	533.617	256.560	0.4807

从图中我们可以看到，在 x86 平台上优化算法由于对 cache 访问的优化性能明显比平凡算法要好：当 n 相对较大时平凡算法的耗时几乎呈现指数式上升，n=10000 时所需的运行时间已经超过了 1s；而优化算法的性能维持在 200ms 一下。并且随着规模的增大优化算法相比平凡算法的优化比率更小，也就是优化的力度更大。

而在基于 ARM 的 M1 芯片上，平凡算法的执行时间相比 x86 有明显的降低，但优化算法大执行效率当 n 较大时反而比 x86 平台上稍差，优化力度随着 n 的增大也不那么明显。猜测可能是由于 M1 芯片的性能调度导致的：在优化算法执行的过程中并没有使用到高性能的内核。在连续执行 n=7000 以上的平凡算法时 Macbook Pro 的风扇有转动，而平凡算法没有间接的说明了猜测的可能性。同时也可能存在在其他的因素导致了这样的差距。

在鲲鹏服务器上分别执行 n=5000 对应平凡算法和优化算法后得到平凡算法运行时间为 333.855ms 和 135.4ms，显著慢于 Intel x86 和 M1 mac ，主要是由于鲲鹏服务器执行程序时为单核 CPU。尝试



(a) intel x86 的执行时间和优化比率

(b) M1 ARM 的执行时间和优化比率

图 1.4: 不同并行优化算法的执行时间与准确率对比

申请更多节点进行测试（4 个）但被 PBS 拒绝。

1.6.2 与 cache 大小的关系

在 Intel x86 平台上对程序运行时间与以及 cache 大小进行了探究。由于矩阵采用的 double 型变量存储，每个数据占 8 字节，根据计算得知当 n 约为 68.8 时，刚好对应于一级缓存 384KB 的大小。由图中对 n 较小时的数据测量计算得到结果如下：（为减少偶然性每次重复循环目标代码段 50 次，重复测量 3 次）

表 3

n	cost(ms)
10	0.0005
20	0.0017
30	0.0034
40	0.0041
50	0.0064
60	0.0083
70	0.0138
80	0.0153
90	0.0199
100	0.0249

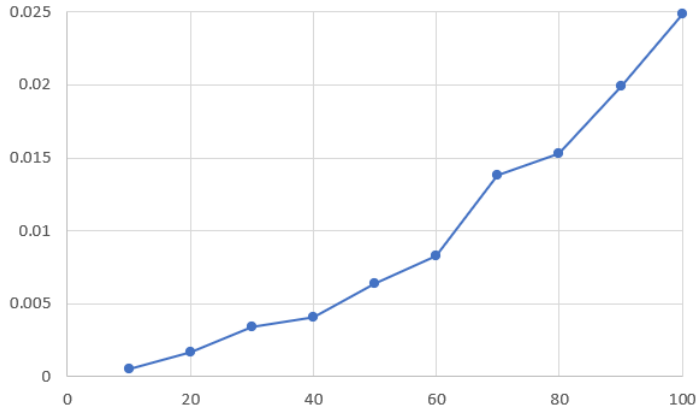


图 1.5: n 较小时与执行时间 L1 关系探究

由结果可知，当 n 超过一级缓存的大小后执行时间有明显的增加。

1.6.3 编译器不同优化选项对性能的影响

在 ubuntu 虚拟机下，对 n=5000 的平凡算法和优化算法的代码进行了不同程度的编译优化，结果如下图所示：（单位：ms)

	default	O0	O1	O2	O3
平凡算法	676	655	726	669	350
优化算法	123	138	36	41	24

表 4: 不同优化力度下性能测试结果 (单位:ms)

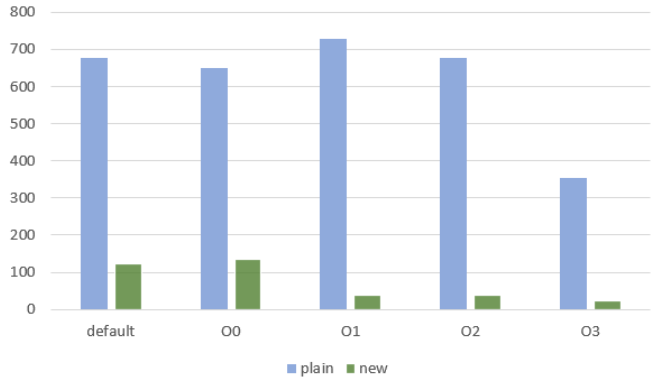


图 1.6: 平凡算法和优化算法编译不同编译优化程度的执行时间

由结果可知，此问题中的编译优化并不能将平凡算法和优化算法的性能差距进行有效的缩小。即便 O3 程度的优化（g++ 官方实际上并不推荐此程度的优化), 将平凡算法的的执行时间缩短到接近原来的一半，但仍旧是优化算法的两倍多。有时在一些情况下 g++ 编译器甚至还会进行负优化。对于优化后汇编代码的分析还可以作进一步深入探究，但限于时间和篇幅在此不再赘述。

1.7 使用 Vtune 和 perf 对 cache 命中率进行分析

在 Intel x86 环境下采用 Vtune 和 perf 对 n=5000 时的 cache 命中率进行了探究：我们能够明显的看到，平凡算法的各级缓存都有较多的 miss，通过 perf 得到 L1 缓存的 miss 甚至达到了 73.13%，而优化算法的 miss 仅有 15.08%。

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample	Precise
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	0	0	25003	
CPU_CLK_UNHALTED.REF_TSC	937,162,008	363	2600000	
CPU_CLK_UNHALTED.REF_XCLK	0	0	25003	
CPU_CLK_UNHALTED.THREAD	0	0	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	40,000,140	5	2000003	
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	2000003	
INST_RETIRED.ANY	0	0	2600000	
MEM_LOAD_RETIRED.L1_HIT	420,183,565	43	2000003	
MEM_LOAD_RETIRED.L1_MISS	26,534,130	54	100003	
MEM_LOAD_RETIRED.L2_HIT	0	0	100003	
MEM_LOAD_RETIRED.L2_MISS	26,819,860	108	50021	
MEM_LOAD_RETIRED.L3_HIT	25,064,170	101	50021	
MEM_LOAD_RETIRED.L3_MISS	1,000,070	3	100007	
UOPS_EXECUTED.THREAD	1,806,759,020	180	2000003	
UOPS_EXECUTED.X87	0	0	2000003	
UOPS_RETIRED.RETIRE_SLOTS	1,553,794,535	155	2000003	

图 1.7: 平凡算法代码使用 Vtune 得到的实验报告

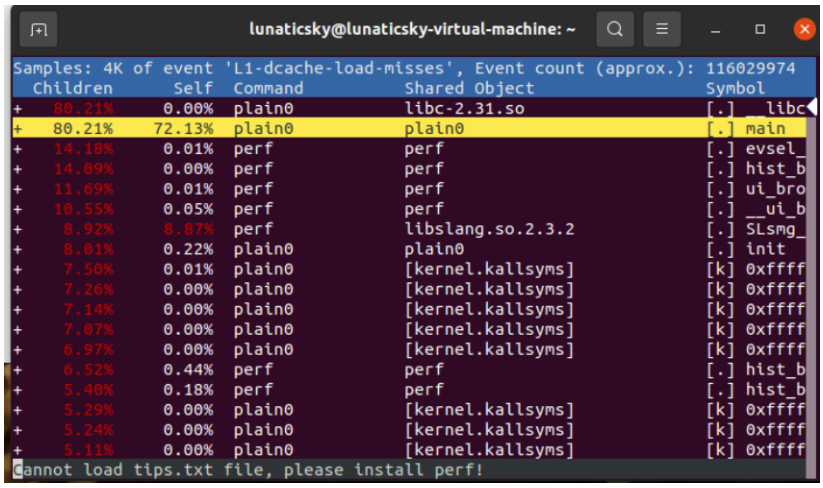


图 1.8: 平凡算法代码使用 perf 得到的 L1-misses

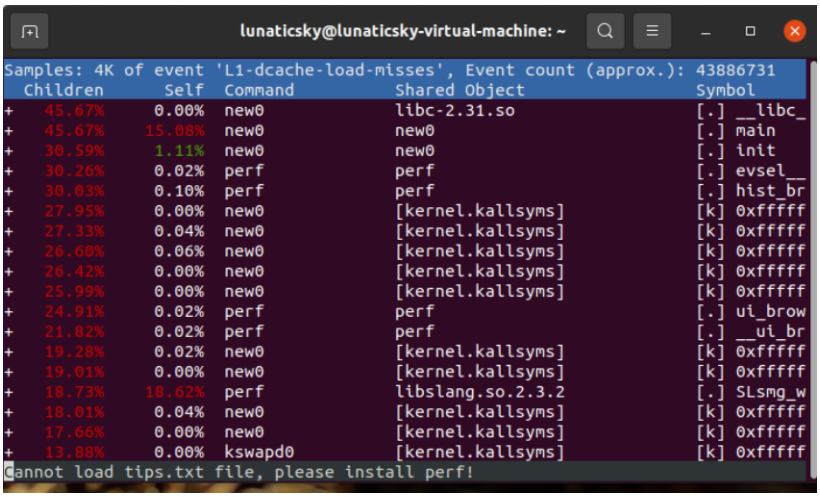


图 1.9: 优化算法代码使用 perf 得到的 L1-misses

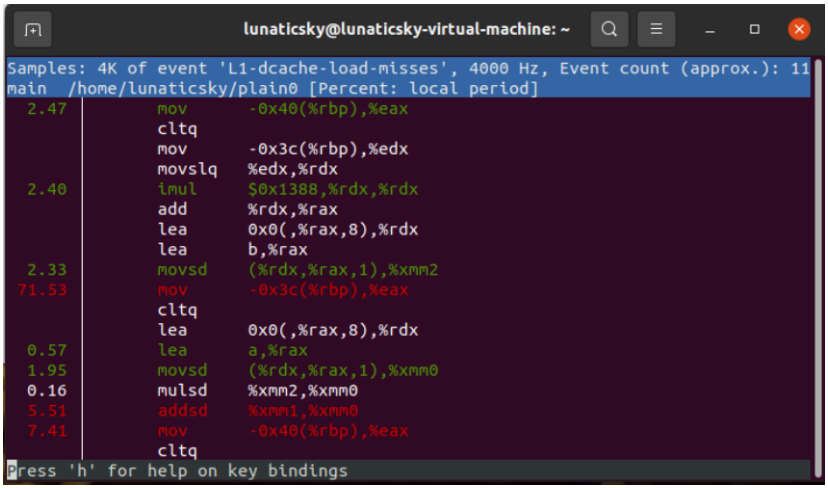


图 1.10: 平凡算法代码得到的导致 cache miss 较多的汇编代码

由 perf 的分析我们也可以看出 cache miss 的发生主要在按照列进行读取数据的过程中。

2 体系结构相关实验分析——超标量优化

2.1 实验任务

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。

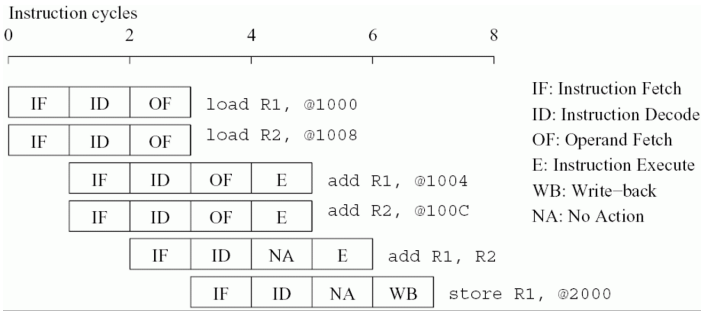


图 2.11: 超标量优化示意

2.2 实验设计和计划

算法设计：题目中要求我们考虑链式算法和并行算法的效率差别，因此我们先考虑算法的设计：根据提示，在本次实验中开始之前设计了平凡算法（逐个累加），二路链式、递归和套用宏模板的算法。在 M1 芯片上进行实验后，发现二路链式在性能上表现最佳。既然二路链式一定程度上能并行的执行不相关的指令，那么四路链式或者八路链式能否做到更好呢？以下实验对此也进行了探究。

程序设计：考虑到便于验证递归和多重循环算法的正确性，我们以 2 的各次幂为程序规模，每个程序内执行 1000 次。同时为了保证公平性，所有算法使用相同的方式创建动态数组，对于所有的算法都在每次计算得出结果后重新进行初始化，并将重新初始化所占用时间一同计入（相同操作不影响对算法的比较。

2.3 实验结果

2.3.1 算法初步测试

首先在 M1 芯片上执行所考虑的算法，结果如下：

表 5: 初步实验结果

2^n	平凡	二路链式	递归	二重循环	宏
4	0.055	0.046	0.074	0.064	0.053
5	0.133	0.104	0.152	0.147	0.134
6	0.299	0.225	0.308	0.332	0.28
7	0.614	0.478	0.625	0.662	0.593
8	1.207	0.942	1.27	1.318	1.271
9	2.442	1.882	2.566	2.717	2.479
10	4.854	3.77	5.078	5.172	4.93
11	9.596	7.633	10.178	10.175	9.981
12	20.163	14.723	20.605	19.885	19.072
13	40.352	29.323	39.072	40.806	38.685
14	74.473	69.079	76.323	77.296	74.68
15	148.46	123.316	164.922	154.615	146.703
16	295.393	258.807	314.29	314.153	295.243

我们能够看到，当 n 较小时即便执行 1000 次所需花费时间也很短，而当 n 过大时会超出 int 的范围。因此我们接下来将着重分析 n=10 到 n=16 之间的情况。将其数据制作成折线图如下：

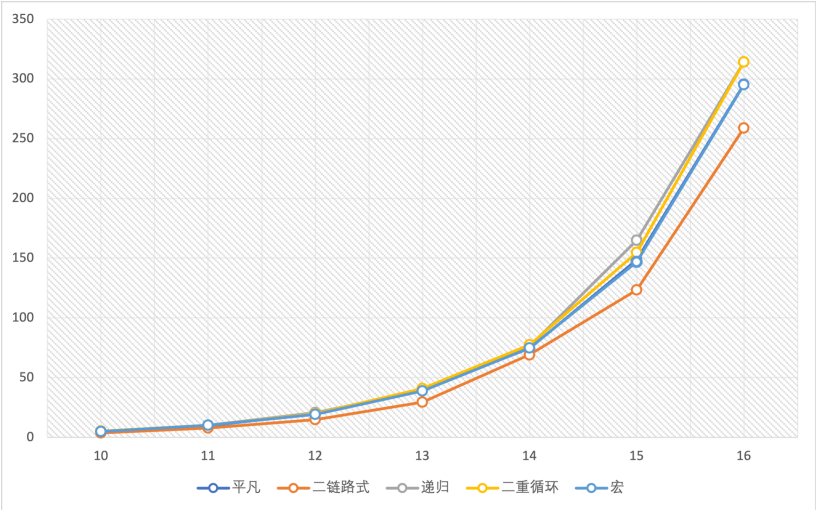


图 2.12: 超标量优化不同算法性能分析

由图我们可以看到：二路链式算法的性能最好；宏和平凡算法的表现基本一致，而二重循环算法和递归算法甚至还不如平凡算法好，尽管其复杂度都为对数级的。猜测可能是由于二重循环和递归将更多的时间浪费在相对较大跨度的寻址和状态的保存上了。

当然，基于循环判断的过程编写更复杂的宏也许能更好的表现 [3]，但可以看出，最基本的循环展开即可较显著的提高累加程序的性能，因此下面对此作更深入的探究：

2.3.2 算法优化

在下面的实验中探索了增加更多的分支链去执行是否能继续优化执行的时间？因此在同样的环境下（M1 芯片，gcc 默认编译优化）进行了测试，实验结果如下：

表 6: 不同链路对应执行时间

2^n	二链路式	四路链式	八路链式
10	3.77	2.906	3.08
11	7.633	5.729	6.085
12	14.723	10.958	12.417
13	29.323	21.814	23.276
14	69.079	43.337	48.55
15	123.316	85.598	92.795
16	258.807	173.16	184.663

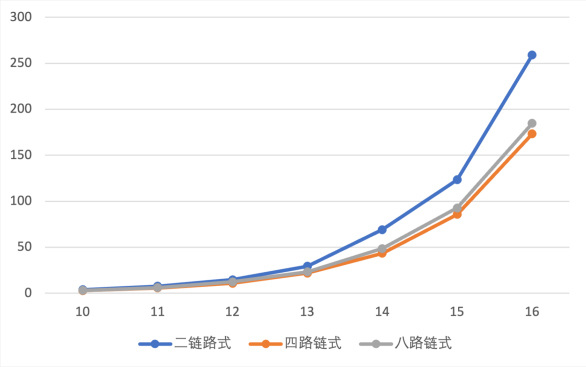


图 2.13: 不同分支链性能分析

我们可以看到，四路优化比二路有了更明显的提升，但八路优化相比四路的提升不再明显，甚至在重复实验得出的结果下略逊于四路优化。循环展开的初衷是减少循环次数以减少对分支的判断，但随着链路的增多并不能继续带来显著的效果，反而会给 CPU 带来更多的操作量（比如数组下标的计算、对中间变量的重新初始化等），违背了减少执行次数的初衷，同时也受限于 CPU 的吞吐量等限制，循环展开对程序性能的提升是有极限的。

2.3.3 数据类型的探讨

之后，对数据为双精度浮点数时的运行效率进行了测试，理论上循环展开对于双精度浮点数的优化不会很明显，但实际的实验中仍旧有较显著的结果。总体上比整型变量的执行速度慢，但对于不同个数分支链路的优化程度与整型变量时相似。

表 7: double 数据类型时不同链路对应执行时间

n	平凡	二路链式	四路链式	八路链式
10	6.126	4.348	2.973	3.12
11	12.452	8.288	7.683	6.192
12	24.948	17.072	12.803	12.294
13	48.002	33.844	26.228	25.061
14	93.7	65.407	48.343	48.623
15	190.373	131.212	95.138	98.717
16	374.765	266.493	180.611	192.282

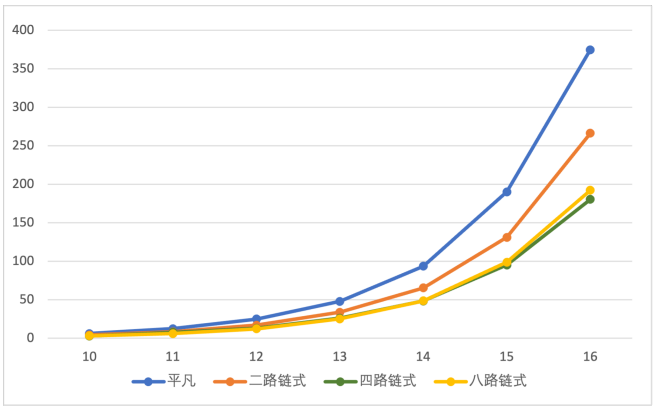


图 2.14: double 数据类型时不同分支链数性能分析

2.3.4 不同平台下的性能表现

最后，将 n=16 (即数据规模为 2 的 16 次方)，数据类型 int 时的情况在三个不同的平台均使用 gcc 默认编译选项运行，得到如下数据：

	Intel x86	鲲鹏 ARM	M1 ARM
平凡算法	374.765	438.6715	340.689
四路链式	180.611	280.3975	141.213

表 8: 不同平台下性能测试结果 (单位:ms)

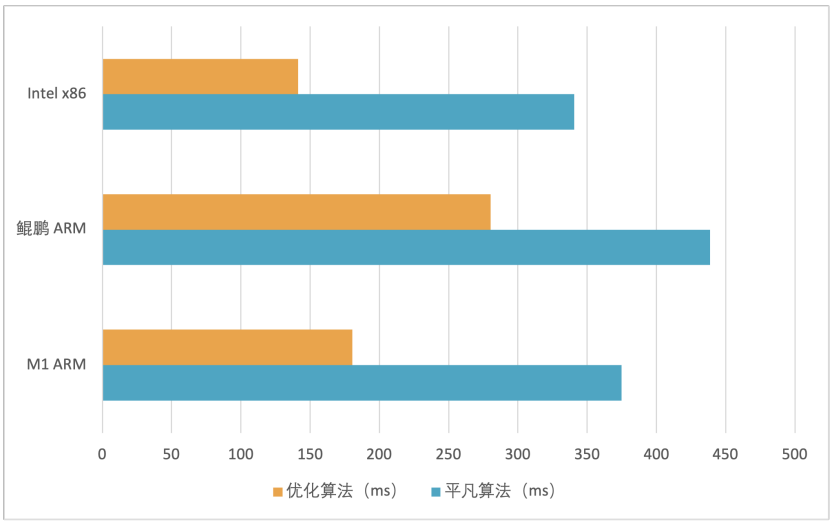


图 2.15: 不同平台下性能测试结果 (单位:ms)

结合第一个实验不同的平台比较结果来看，ARM 架构的 M1 芯片虽然性能强悍，日常使用也非常节能，但在这两个实验中的表现和 x86 还是有一定差距的，而鲲鹏服务器由于资源分配的限制性能也很难与个人笔记本电脑相媲美。

3 源码项目链接

Github 源码

参考文献

[1] [EB/OL]. https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive,.

[2] [EB/OL]. https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested.

[3] [EB/OL]. https://blog.csdn.net/u013471946/article/details/40680965.