南開大學
Nankai University

计算机学院

并行程序设计作业报告

体系结构相关及性能测试报告

姓名：莫骐骥

学号：2010237

专业：计算机科学与技术

2022 年 3 月 15 日

目录

1 概述及实验环境介绍

此次实验内容分为两个大部分，cache 优化和超标量优化。我计划对比分析程序在 X86 架构和 ARM 架构上的性能，对比分析 g++ 编译器不同优化力度对程序性能的影响，纵向分析不同问题规模时程序的性能表现。此次实验我的学习目标为以下几点：

1. 学会使用鲲鹏服务器。
2. 进一步熟悉 git 和服务器的使用方法。
3. 学会使用 perf 分析程序性能。
4. 学会使用 gettimeofday 函数精确计时。
5. 学会通过从多个角度对比分析实验结果。
6. 总结出影响程序性能的几个重要指标。

本次实验环境配置：电脑配置为联想小新 Air14，CPU 参数为 Intel i5-1035G1，一级数据缓存大小为 4*48KB，一级指令缓存大小为 4*32KB，二级缓存大小为 4*512KB。X86 架构下程序性能表现均是虚拟机中 Ubuntu20.04.3（运行内存 4GB）环境下使用 g++ 编译器的运行结果反映，ARM 架构下程序性能表现均为使用鲲鹏服务器的结果。

本实验中的所有程序使用了一个整型数组记录不同的数据规模，通过遍历数组实现运行一次程序即可重复对每个数据规模进行实验且列出完整实验结果表格。可能是由于电脑的缓存较小，在进行测试时偶尔会出现在测试至某一数据规模时顿住的情况，故在有对比度的前提下适当缩小了最大数据规模，进行重复实验。在 cahce 优化实验中最大数据规模为 4096，在超标量优化实验中最大数据规模为 16384.

2 cache 优化

2.1 实验内容重述

这是此次实验的第一部分内容。实验内容为：给定一个 N×N 矩阵，计算每一列与给定向量的内积。为了简化问题，向量的每个方向的分向量长度均设为随机数，矩阵的每个元为所在位置的行数与列数之和。

2.2 平凡算法

在看到实验指导书中有关平凡算法的内容后，我认为平凡算法很别扭，但我当时这样想的原因仅源于对先访问列后访问行这一遍历方式的不适应，并不知道这样的遍历方式会导致大量的 cache missed。

平凡算法简述：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。程序核心代码如下：

```
1  gettimeofday(&tv1,nullptr);
2  gettimeofday(&tv2,nullptr);
3  while((tv2.tv_usec-tv1.tv_usec)<10000){
4      round ++;
```

```
    for(int i = 0;i < N;i ++){
        res[i] = 0;
        for(int j = 0;j < N;j ++){
            res[i]+=matrix[j][i]*array[j];
    }
}
gettimeofday(&tv2,nullptr);
}
```

此算法在 X86 架构下且不使用编译器优化的实验结果如下表所示：

数组规模	重复次数	总运行时间（单位微秒）	单次运行时间（单位微秒）
2	761	10012	13.1564
4	748	10011	13.3837
8	667	10007	15.003
16	763	10005	13.1127
32	532	10006	18.8083
64	264	10021	37.9583
128	84	10076	119.952
256	15	10459	697.267
512	4	11182	2795.5
1024	1	16379	16379
2048	1	79183	79183
4096	1	398060	398060

表 1: 平凡算法在 X86 架构下的性能

由表1可以看出，随着数组规模的增大，单次运行时间增幅显著，在优化算法小节会进行对比分析。

2.3 优化算法

cache 优化算法思路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，由于这样的访存方式契合数据在内存中的存储方式——行主存储。由于设计计算机时遵循了这样一个基本准则：程序接下来（时间局部性）很可能会用到的指令和数据与最近访问过的指令和数据在物理上是临近存放的（空间局部性）。故这样的访存模式具有很好空间局部性，令 cache 的作用得以发挥。

核心代码如下：

```
gettimeofday(&tv1,nullptr);
gettimeofday(&tv2,nullptr);
while((tv2.tv_usec-tv1.tv_usec)<10000){
    round ++;
    for(int i = 0;i < N;i ++){
```

```
6           res[i] = 0;
7       }
8       for(int j = 0;j < N;j ++){
9           for(int i = 0;i < N;i ++){
10              res[i]+=matrix[j][i]*array[j];
11          }
12  }
13  gettimeofday(&tv2,nullptr);
14  }
```

首先在 Ubuntu 环境下测试 X86 架构的性能，实验结果表2:

数组规模	重复次数	总运行时间（单位微秒）	单次运行时间（单位微秒）
2	824	10006	12.1432
4	908	10010	11.0242
8	782	10003	12.7916
16	742	10012	13.4933
32	554	10003	18.056
64	322	10175	31.5994
128	120	10027	83.5583
256	35	10077	287.914
512	9	11139	1237.67
1024	3	13314	4438
2048	1	18993	18993
4096	1	76181	76181

表 2: cache 优化算法在 X86 架构下的性能

登录鲲鹏服务器，测试 cache 优化算法在 ARM 架构下的表现，实验结果如表3所示:

数组规模	重复次数	总运行时间（单位微秒）	单次运行时间（单位微秒）
2	176358	10000	0.056703
4	87606	10000	0.114147
8	28958	10000	0.345328
16	7733	10000	1.29316
32	1922	10004	5.20499
64	491	10018	20.4033
128	125	10050	80.4
256	30	10147	338.233
512	8	11145	1393.12
1024	2	11300	5650
2048	1	23158	23158

4096	1	95375	95375

表 3: cache 算法在 ARM 架构下的性能

2.4 对比分析

比较两种算法在 X86 架构下的性能表现。

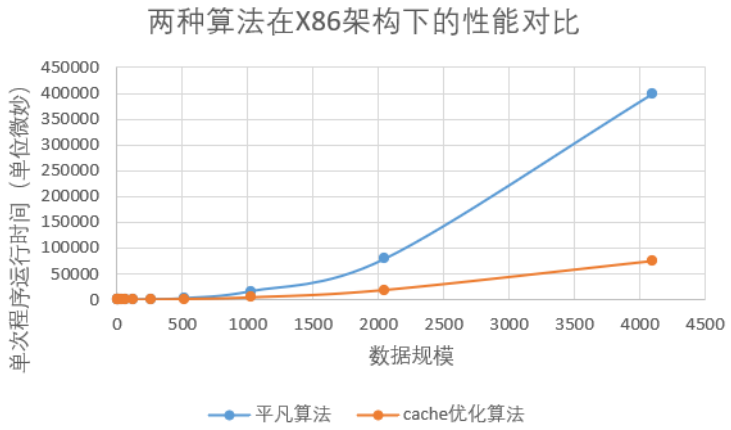
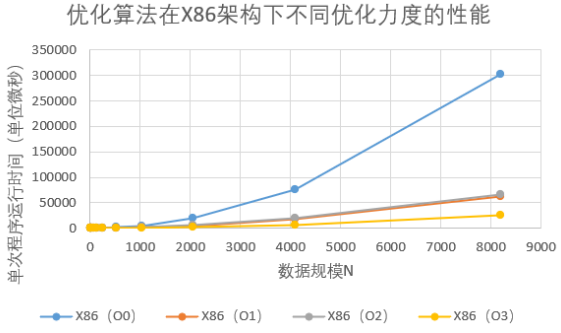


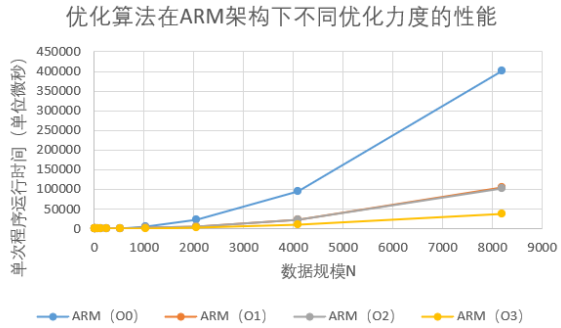
图 2.1: Caption

数据规模较小时，由于 cache 命中率不是程序运行时间的主要决定因素，优化算法的优势并不明显。当数据规模增大时，两种算法的性能差距越来越大，在数据规模为 4096 时达到接近五倍差距。

再比较不同优化级别的优化算法在两种架构下的性能。



(a) 优化算法在 X86 架构下不同优化力度的性能

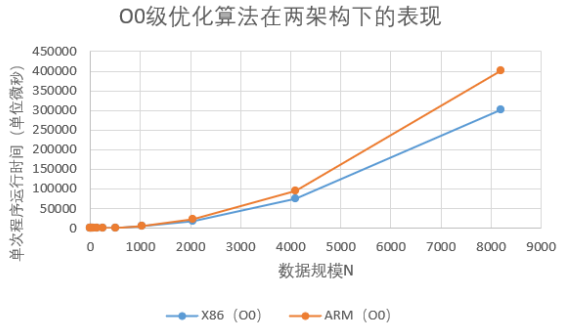


(b) 优化算法在 ARM 架构下不同优化力度的性能

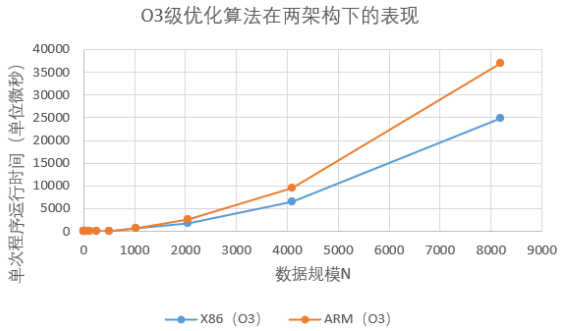
图 2.2: 优化算法在两架构下不同优化级别的性能表现

分析图 2.2，优化可以明显提升程序运行效率，对比不同的优化级别，随着优化级别增加，程序优化幅度也随之增加。在 X86 架构下，O3 级别的优化相比于 O0 级的程序运行效率提高 12 倍，在 ARM 架构下提高 10.8 倍。

进一步对比同一优化级别的优化算法在两架构下的性能。分别以分析 O0 级别和 O3 级别为例。



(a) O0 级优化算法在两架构下的性能



(b) O3 级优化算法在两架构下的性能

图 2.3: 相同算法在 ARM 和 X86 架构下的性能对比

2.5 perf 分析 cache 命中率

进一步使用 perf 性能分析工具来定量分析平凡算法与编译器不同优化力度的 cache 优化算法的 cache 命中率。由于我的电脑不支持虚拟化 cpu 计数器，所以我只能在鲲鹏服务器下测试。结果如下表所示。

算法	cache-misses	cache-references	cycles	miss-rate
平凡算法	101900796	1844897937	8169970150	0.055233839
O0cache 优化	4937883	1949128807	2107811782	0.00253338
O1cache 优化	6640686	537483779	787447917	0.012355138
O2cache 优化	6745923	538131746	765390782	0.012535821

表 4: perf 分析结果

由上表可见，平凡算法的 cache 丢失率是优化算法的 cache 丢失率的五倍。使用 perf 分析：O1，O2 级优化大大降低了总 cache 规模，对于平凡算法，99.74% 的 cache-miss 发生在实现向量内积的核心代码部分，优化算法则是 90.42% 的 cache-miss 发生在核心代码部分，与预想偏差较大。

2.6 小结

优化算法相比于平凡算法提高了 cache 命中率，大大加快了程序运行效率。纵向对比编译器的优化，编译器的优化效果也十分显著，O3 级优化相比于 O0 级优化性能提升五倍左右，但这在数据规模较小时效果并不明显。横向对比 ARM 架构下的程序性能，发现于 X86 架构下的程序性能差距并不大，但在数据规模较小时加速明显。在 cache 命中率方面，实验结果与预想结果不太一致，平凡算法没有导致很大比例的 cache 丢失，可能与我的程序中将向量和矩阵声明为全局变量有关。

3 超标量优化

3.1 实验内容重述

实验内容是计算 n 个数的和。使用如下几种算法设计思路：

- 逐个累加的平凡算法 (链式)。

- 两路链式累加。
- 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。有两种实现方式，一种是用函数实现，另一种是用二重循环实现。

为了满足优化算法逻辑正确执行，实验数据规模皆为 2 的整数幂。另外，数组中的数使用随机数初始化。经过前期测试，发现我所设计的最大数据规模（16384）使用 g++ O0 级优化的平凡算法，执行速度仍十分迅速（单次执行大概 60-70 微秒），故在实验过程中对于每个数据规模都重复核心计算至运行时间达到 10000 微秒，计算其每次计算的平均值作为实验结果。

3.2 平凡算法

平凡算法即遍历数组逐个累加。
核心代码如下：

```
    gettimeofday(&t1,nullptr);
    gettimeofday(&t2,nullptr);
    while(t2.tv_usec-t1.tv_usec<10000){
        int sum = 0;
        for(int j = 0;j < size[i];j ++){
            sum+= array[j];
    }
    round++;
    gettimeofday(&t2,nullptr);
    }
```

X86 架构下 g++ 不优化实验结果：

数组规模	重复次数	总运行时间（单位微秒）	单次运行时间（单位微秒）
2	703	10005	14.2319
4	761	10012	13.1564
8	696	10009	14.3807
16	717	10011	13.9623
32	764	10009	13.1008
64	601	10004	16.6456
128	639	10009	15.6635
256	671	10010	14.918
512	618	10003	16.1861
1024	610	10003	16.3984
2048	454	10005	22.0374
4096	367	10020	27.3025
8192	251	10026	39.9442
16384	144	10043	69.7431

表 5: 逐个累加算法在 X86 架构 O0 级优化实验结果

由实验结果可以看出，当数组规模较小时，运行时间几乎相同。再结合 ARM 架构下的实验结果进行分析。

数组规模	重复次数	总运行时间（单位微秒）	单次运行时间（单位微秒）
2	246895	10000	0.040503
4	230965	10000	0.043297
8	176157	10000	0.056768
16	120593	10000	0.082924
32	73816	10000	0.135472
64	39966	10000	0.250213
128	21966	10000	0.455249
256	11506	10000	0.869112
512	5815	10000	1.71969
1024	2926	10002	3.41832
2048	1474	10000	6.78426
4096	738	10002	13.5528
8192	371	10020	27.0081
16384	186	10045	54.0054

表 6: 逐个累加算法在 ARM 架构下 O0 级优化的实验结果

观察 ARM 架构下同一程序的实验结果发现在数据规模较小时并没有出现增大数据规模但运行时间几乎不变的情况，而是随时间递增，再结合 O3 级优化进一步分析。总对比图如下：

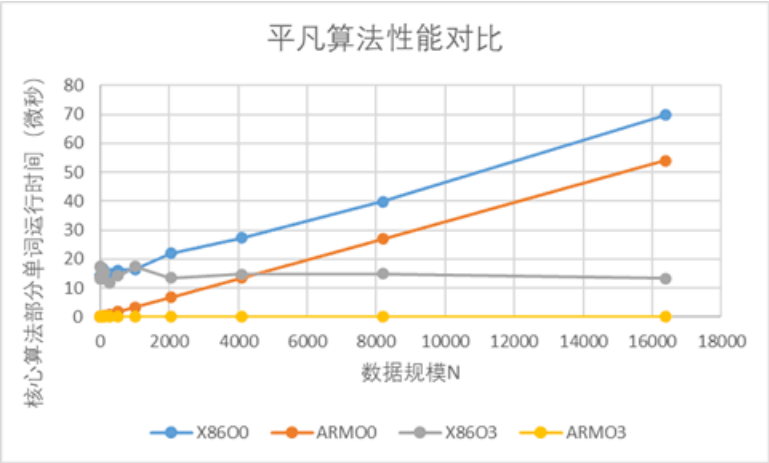


图 3.4: 双架构平凡算法性能对比

根据图3.4所示实验结果，对比 ARM 和 X86 架构下程序性能，在编译器优化力度相同的情况下表现为 ARM 架构普遍优于 X86 架构。分析原因是此实验程序行为单一，复杂性相对较低，发挥出 ARM 架构的优势，对比 O0 和 O3 级编译器优化，发现优化效果在 ARM 架构下最为显著，如图中黄

线, 即使随着数据规模增大，核心算法的运行时间保持在很低范围:0.0375-0.0379 微秒。经过编译器优化后，在 X86 架构下的程序出现了数据规模较低时运行时间异常增加的情况，多次重复实验后发现不是偶然，这可能与编译器对循环结构、寄存器、指令进行了优化有关。

3.3 两路链式累加

两路链式累加相比于平凡算法的改动就是在遍历数组时改变步长，采用了循环展开策略，将两次循环步的运算合并到一次中，在每次循环中进行两次计算，分别保留奇数列和偶数列的累加结果，最后再将二路累加结果求和，从而实现超标量计算。

核心代码如下:

```
    gettimeofday(&t1,nullptr);
    gettimeofday(&t2,nullptr);
    while(t2.tv_usec-t1.tv_usec<10000){
        int sum1 = 0, sum2 = 0,sum;
        for (int j = 0;j < size[i]; j += 2){
        sum1 += array[j];
        sum2 += array[j + 1];
        }
        sum = sum1 + sum2;
        round++;
        gettimeofday(&t2,nullptr);
    }
```

数据规模	两路累加		平凡算法	
	X86O0	ARMO0	X86O0	ARMO0
2	12.9187	0.039647	14.2319	0.040503
4	13.2772	0.041659	13.1564	0.043297
8	13.0692	0.050613	14.3807	0.056768
16	14.2493	0.067879	13.9623	0.082924
32	13.7972	0.097997	13.1008	0.135472
64	15.0648	0.159165	16.6456	0.250213
128	15.3497	0.286821	15.6635	0.455249
256	14.6	0.533163	14.918	0.869112
512	16.1484	1.02365	16.1861	1.71969
1024	19.1224	2.00944	16.3984	3.41832
2048	17.2712	3.98486	22.0374	6.78426
4096	24.0553	7.92552	27.3025	13.5528
8192	29.2807	15.6928	39.9442	27.0081
16384	44.6652	31.3668	69.7431	54.0054

表 7: 两路链式累加与平凡算法性能对比

由表7和图3.5所示，程序运行时间随数据规模线性增加。随着数据规模的增大，两路累加算法的效率优势逐渐凸显。本小节的比较是在 g++ 编译器 O0 级优化的前提下进行的。实际上，两路算法的实验结果出现了与上一小节类似的情况：在 ARM 架构下进行编译器不同优化力对比实验时，在实验数据范围（2-16384）内，核心算法单次运行时间随数据规模增大而发生的变化及其微小，分析是由于程序指令单一，充分发挥了 ARM 架构的优势，又进行了完善的编译器优化，使程序性能达到瓶颈；在 X86 架构下，小数据规模的实验结果又出现了波动现象，分析原因与上一小节类似。

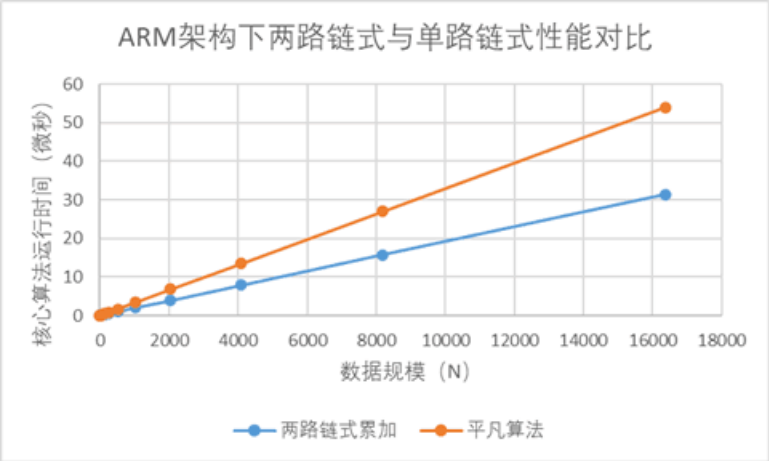


图 3.5: ARM 架构下两算法核心运行时间对比

3.4 递归算法

将数组元素两两相加，得到 n/2 个中间结果；将上一步得到的中间结果两两相加，得到 n/4 个中间结果; 依此类推，log(n) 个步骤后得到一个值即为最终结果。可使用以下策略实现。第一种方法是用一个递归函数，每次递归使遍历区间折半。第二种方法是使用一个二重循环，外部循环使每次遍历区间折半，内部循环遍历折半后的区间，将另一半区间的数据累加前半区间。以上两种方法均实现循环展开策略，将多次循环步的计算任务合并至依次，在内部使用小循环遍历。

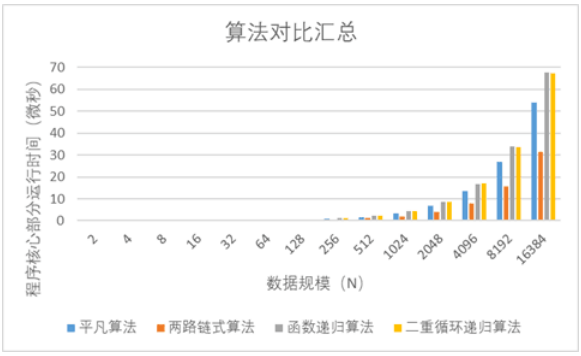
函数递归核心代码如下：

```
1     void recursion(int* a,int n)
2     {
3         if (n == 1)
4             return;
5         else{
6             for (int i = 0; i < n / 2; i++)
7                 a[i] += a[n - i - 1];
8             n = n / 2;
9             recursion(a,n);
10        }
11    }
```

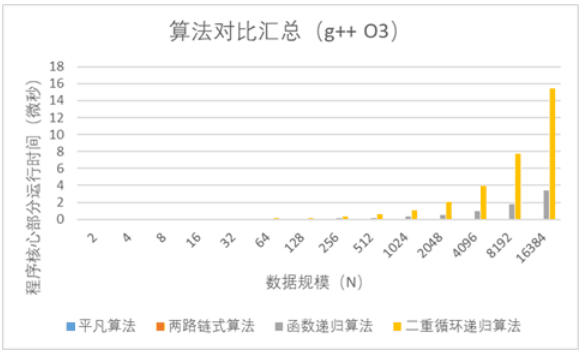
二重循环递归核心代码如下：

```
1      void func(int* a,int n)
2      {
3          for (int m = n; m > 1; m /= 2)
4              for (int i = 0; i < m / 2; i++)
5                  a[ i ] = a[i * 2] + a[i * 2 + 1] ;
6      }
```

直接比较平凡算法、两路链式算法与两种递归算法的性能（ARM 架构下）：



(a) 编译器不优化时算法性能对比汇总



(b) g++O3 优化时算法性能对比汇总

图 3.6: 超标量优化算法对比汇总

分析：相较于链式累加算法，g++ 编译器对递归算法的优化作用并不明显。g++ 在 O3 级别优化时，对单路链式累加算法可提供 1400 倍、对两路链式累加算法可提供 800 倍的优化效果，而作用于函数递归算法只提升了 20 倍、二重循环递归算法只提升了 4 倍优化效果。而且递归算法在本实验中并没有显示出优化作用，无论是否有编译器优化，单次运行时间反而比平凡算法更长，编译器优化后的递归算法与链式累加算法的差距进一步拉大，已不在一个数量级上。

使用 perf 分析：

算法	instructions	branch-misses	cache-misses	cache-references	miss-rate
平凡算法 O0	673813091	96058	13802	347397557	3.97297E-05
两路链式算法 O0	797710553	84825	17573	398039285	4.41489E-05
函数实现递归算法 O0	944477522	152846	15017	425285351	3.53104E-05
二重循环递归算法 O0	965242019	168335	13526	390824972	3.46088E-05

表 8: perf hardware 分析结果

分析上表可见，在本实验中 cache 命中率已不是区分程序运行快慢的指标。分析上表，结合图3.6，两种递归算法的共同特点是指令数更多，且分支预测错误更多，从数据来看后者体现得更加明显。针对两种不同的递归算法，认为函数递归造成的额外开销可能也是导致程序运行效率低的原因之一；二重循环的内部循环语句是将两个位置的数据存放到另一个位置，具有前后依赖关系，可能导致了超流水线处理的作用无法发挥。

4 总结

我从此次实验作业中受益匪浅，基本完成了一开始时我设定的目标。但由于个人硬件知识的匮乏和尚未完全摸清门道，很多地方仍不知如何下手，比如掌握 perf 的使用仍是我的一大问题，还有 vim 的使用等等。除了工具的使用之外，我会在接下来的学习中加强对底层知识的掌握，这样做既可以更好地完成作业任务，也可以与计算机组成原理、操作系统、编译原理等课程贯通起来。

本次实验的代码实现