南開大學
Nankai University

计算机学院

并行程序设计实验报告

矩阵计算有关的并行加速算法

姓名：高祎珂

学号：2011743

专业：计算机科学与技术

2022 年 3 月 15 日

目录

1 实验一

1.1 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，进行实验对比. 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，而 cache 优化算法按行访问矩阵元素，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。因为在程序内部，矩阵是以二维数组形式存在，按行存储，后者令访存模式具有更好的空间局部性，从而发挥 cache 的能力。

1.2 实验平台介绍

ARM		X 86	
架构	aarch64	CPU 型号	Intel Core i7-1065G7 CPU
CPU 数量	96	内核数	4
CPU max MHz	2600	线程数	8
CPU min MHz	200	基本频率	1.5GHz
L1dcache	64K	L1 cache	320KB
L1icache	64K	L2cache	2.0MB
L2cache	512K	L3cache	8.0MB
L3cache	49152K	最大内存大小	64GB

1.3 ARM 平台

1.3.1 代码

源码链接在这里 Github

逐列访问平凡算法

```
1  void ordinary()
2  {
3      unsigned long diff;
4      struct timeval tv_begin,tv_end;
5      gettimeofday(&tv_begin,NULL);
6      int cir=1000;
7      for(int k=0;k<cir;k++)
8      {
9          for(int j=0;j<N;j++)
10         {
11             res[j]=0;
12             for(int i=0;i<N;i++)
13                 res[j]+=mat[i][j]*arr[i];
14         }
15     }
16     gettimeofday(&tv_end,NULL);
17     diff = 1000000 * (tv_end.tv_sec-tv_begin.tv_sec)+
           tv_end.tv_usec-tv_begin.tv_usec;
18     cout<<(diff/1000.0)/cir<<"ms"<<endl;
19 }
```

cache 优化算法

```
void youhua()
{
    unsigned long diff;
    struct timeval tv_begin,tv_end;
    gettimeofday(&tv_begin,NULL);
    int cir=1000;
    for(int k=0;k<cir;k++)
    {
        for(int i=0;i<N;i++) res[i]=0;
        //memset(res,0,sizeof(int)*N);
        for(int j=0;j<N;j++)
            for(int i=0;i<N;i++)
                res[i]+=mat[j][i]*arr[j];
    }
    gettimeofday(&tv_end,NULL);
    diff = 1000000 *
        (tv_end.tv_sec-tv_begin.tv_sec)+tv_end.tv_usec-tv_begin.tv_usec;
    cout<<(diff/1000.0)/cir<<"ms"<<endl;
}
```

1.3.2 程序运行结果

数据规模	平凡算法运行时间/ms	cache 优化算法运行时间/ms	加速比
100	0.05422	0.05142	1.05445
200	0.2171	0.20538	1.05706
300	0.4873	0.4616	1.05568
400	0.88096	0.82308	1.07032
500	1.40098	1.295	1.08184
600	2.0842	1.8607	1.12012
700	2.9895	2.53052	1.18138
800	3.87792	3.29566	1.17667
900	5.10648	4.17374	1.22348
1000	7.07402	5.18464	1.36442
1100	7.511	6.321	1.18826
2100	31.113	23.693	1.31317
3100	64.287	51.801	1.24104
4100	183.959	90.287	2.03749
5100	358.586	146.88	2.44135
6100	569.638	211.414	2.69442
7100	871.257	282.497	3.08413
8100	979.214	367.195	2.66674
9100	1345.1	463.653	2.90109

1.4 X86 平台

1.4.1 代码

源码链接可在这里找到 Github

逐列访问平凡算法

```
void ordinary(int n)
{
    long long head, tail,freq ;
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    QueryPerformanceCounter((LARGE_INTEGER*)&head);
    int times=0;
    while(times<n)
    {
        for(int i = 0; i < N; i++)
        {
          sum[i]=0.0;
          for(int j = 0; j < N; j++)
            sum[i] += b[j][i] * a[j];
        }
         times++;
    }
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    cout<<(tail-head)*1000.0/(n*freq)<<"ms"<<endl;
}
```

cache 优化算法

```
void youhua(int n)
{
    long long head, tail,freq ;
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    QueryPerformanceCounter((LARGE_INTEGER*)&head);
     int times=0;
    while(times<n)
    {
        for(int i = 0; i < N; i++)
            sum[i] = 0.0;
        for(int j = 0; j < N; j++)
            for(int i=0;i<N;i++)
                sum[i] += b[j][i] * a[j];
        times++;
    }
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    cout<<(tail-head)*1000.0/(freq*n)<<"ms"<<endl;
}
```

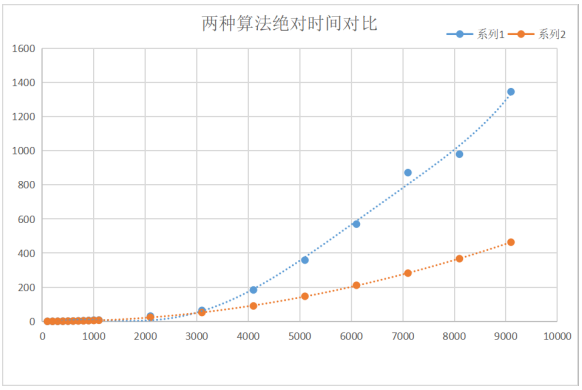
程序在设计时，设计了不同的问题规模，对于不同问题规模设计了不同的重复执行次数，问题规模较小时，重复执行次数较大，问题规模较大时，就缩减了重复执行次数，这样也便于程序执行总时间的合理性。

1.4.2 程序执行结果

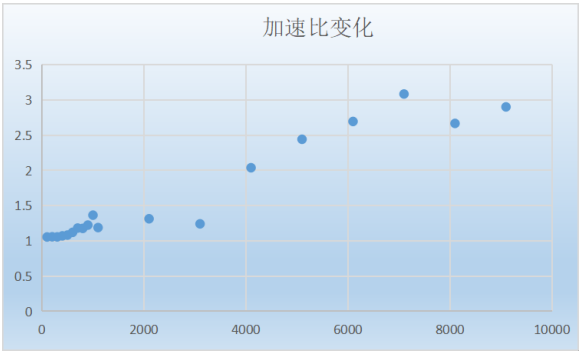
数据规模	平凡算法运行时间/ms	cache 优化算法运行时间/ms	加速比
100	0.0291	0.0298	0.97651
200	0.1148	0.1158	0.991364
300	0.3028	0.3016	1.00398
400	0.5237	0.483	1.08427
500	0.9002	0.738	1.21978
600	1.3699	1.0793	1.26925
700	1.9496	1.4744	1.3223
800	2.7027	2.04	1.32485
900	3.674	2.4605	1.49319
1000	4.1047	3.0688	1.33756
1100	5.1215	3.8966	1.31435
2100	20.1825	13.7928	1.46326
3100	61.8444	28.6999	2.15486
4100	123.433	50.263	2.45574
5100	194.051	78.2931	2.47852
6100	292.81	110.163	2.65796
7100	445.965	150.822	2.9569
8100	609.16	216.534	2.81323
9100	808.811	287.126	2.81692

1.5 程序结果分析

对于 ARM 平台，程序执行结果，作图分析如下：



(a) 两种算法绝对时间对比

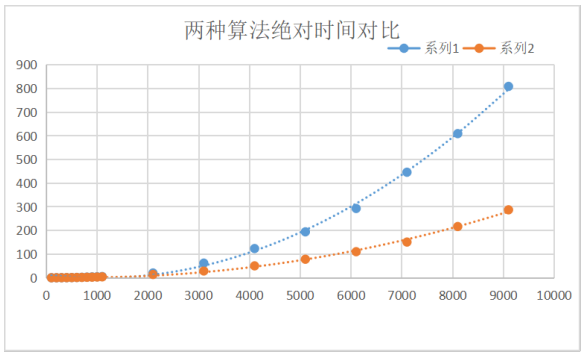


(b) 加速比

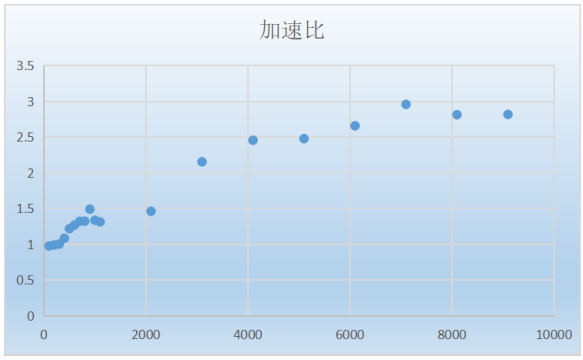
图 1.1: ARM 不同算法的执行时间对比

分析上面两幅图可知，cache 算法优化与原来按列访问的平凡算法相比，对于运行时间的确有了优化，且在问题规模较大时，加速比也在不断上升，这个优化效果更加明显，矩阵规模达到 7000+ 时，cache 算法的优势几乎是平凡算法的 3 倍。出现这种情况从理论上分析来说是因为采用 Cache 优化算法，是 cache 命中率增大，不用每进行一次运算就进行一次内存读取，极大减小了读取内存时间，具体分析还可通过 VTnue 工具来定性描述。

对于 X 86 平台程序执行结果，作图分析如下：



(a) 两种算法绝对时间对比

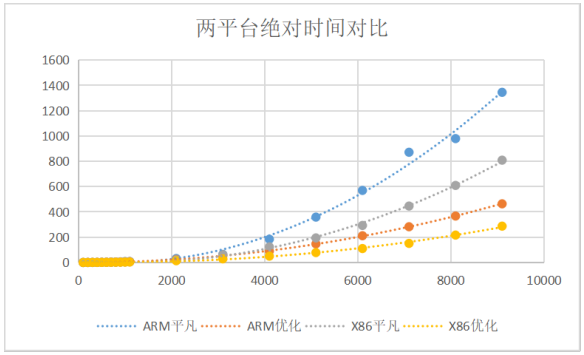


(b) 加速比

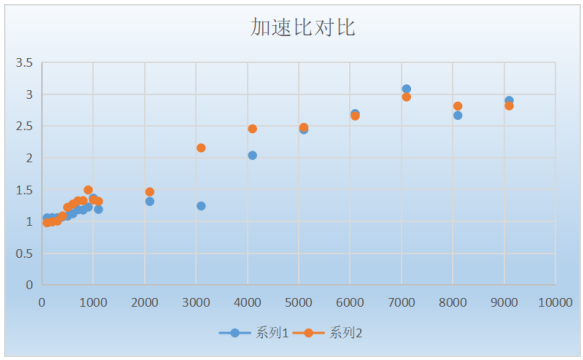
图 1.2: X 86 不同算法的执行时间对比

整体规律表现基本与 ARM 平台相同，加速比变化趋势与两种算法的绝对时间对比变化趋势都几乎相同，但是 X 86 平台有一个异常情况，在数据规模较小时，出现了优化算法效果不如平凡算法的情况，按理来说不应该出现这种状况，使用 VTune 工具查看一下具体发生了什么。

ARM 和 X 86 平台对比分析



(a) 两个平台绝对时间对比



(b) 加速比

图 1.3: 两个平台性能对比

整体而言，两者变化趋势几乎一样，但是横向对比，本机由于 cache 较大，性能会比在鲲鹏服务器上跑的 arm 平台性能更优，主要是读取内存的时间较短，可用 VTune 工具具体查看分析。

2 实验二

2.1 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。

实验平台与实验一相同

2.2 ARM 平台

2.2.1 代码

源码链接可在这里找到 Github

逐个累加平凡算法

```
void ordinary()
{
    unsigned long diff;
    struct timeval tv_begin,tv_end;
    gettimeofday(&tv_begin,NULL);
    while(times<cir)
    {
        for(int i=0;i<scale;i++)
            sum+=a[i];
            times++;
    }
    gettimeofday(&tv_end,NULL);
    diff = 1000000 * (tv_end.tv_sec-tv_begin.tv_sec)+
        tv_end.tv_usec-tv_begin.tv_usec;
    cout<<(diff/1000.0)/cir<<"ms"<<endl;
}
```

两路链式累加算法

```
void youhua()
{
    unsigned long diff;
    struct timeval tv_begin,tv_end;
    gettimeofday(&tv_begin,NULL);
    while(times<cir)
    {
        sum=0;
        long sum1 = 0;
        long sum2 = 0;
        for (int i = 0;i < scale; i += 2)
        {
            sum1 += a[i];
            sum2 += a[i + 1];
        }
        sum = sum1 + sum2;
        times++;
    }
    gettimeofday(&tv_end,NULL);
    diff = 1000000 * (tv_end.tv_sec-tv_begin.tv_sec)+
        tv_end.tv_usec-tv_begin.tv_usec;
    cout<<(diff/1000.0)/cir<<"ms"<<endl;
}
```

程序在设计时，设计了不同的问题规模，对于不同问题规模设计了不同的重复执行次数，问题规模较小时，重复执行次数较大，问题规模较大时，就缩减了重复执行次数，利用 gettimeofday 计时函数来计算所用时间，这样也便于程序执行总时间的合理性。

2.2.2 程序运行结果

数据规模	平凡算法运行时间/ms	优化算法运行时间/ms	加速比
10	3e-05	2e-05	1.5
210	0.00065	0.00042	1.54762
410	0.00126	0.00092	1.36957
610	0.00186	0.00121	1.53719
810	0.00246	0.00161	1.52795
1010	0.004	0.002	2
11010	0.035	0.022	1.59091
21010	0.066	0.042	1.57143
31010	0.097	0.063	1.53968
41010	0.13	0.082	1.58537
51010	0.167	0.106	1.57547
61010	0.19	0.123	1.54472
71010	0.227	0.142	1.59859
81010	0.252	0.162	1.55556
91010	0.284	0.183	1.55191

对于表格数据分析，随着数据规模的增大，两种算法运行时间都在不断延长，但是加速比并没有明显增大，基本维持在 1.5 左右，

2.3 X86 平台

2.3.1 代码

源码链接可在这里找到 Github

逐个累加平凡算法

```
void ordinary()
{
        long long head, tail,freq ;
        QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
        QueryPerformanceCounter((LARGE_INTEGER*)&head);
        int repeat=0;
        while(repeat<cir)
        {
            for(int i=0;i<scale;i++)
            sum+=a[i];
            repeat++;
        }
        QueryPerformanceCounter((LARGE_INTEGER *)&tail);
        double space=(tail-head)*1000.0/(freq*repeat);
        cout<<space<<"ms"<<endl;
}
```

两路链式累加算法

```
void youhua()
{
        long long head2, tail2;
        QueryPerformanceCounter((LARGE_INTEGER*)&head2);
        repeat=0;
        while(repeat<cir)
        {
            long sum1 = 0;
            long sum2 = 0;
            for (int i = 0;i < scale; i += 2)
            {
                sum1 += a[i];
                sum2 += a[i + 1];
            }
            sum = sum1 + sum2;
            repeat++;
        }
        QueryPerformanceCounter((LARGE_INTEGER *)&tail2);
        double space1=(tail2-head2)*1000.0/(freq*repeat);
}
```

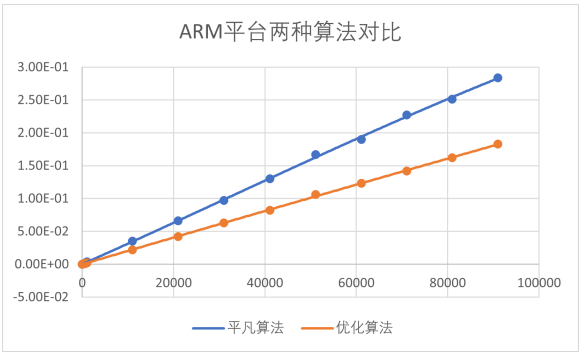
算法设计思路基本与 ARM 平台相同，只是更换了计时函数

2.3.2 程序运行结果

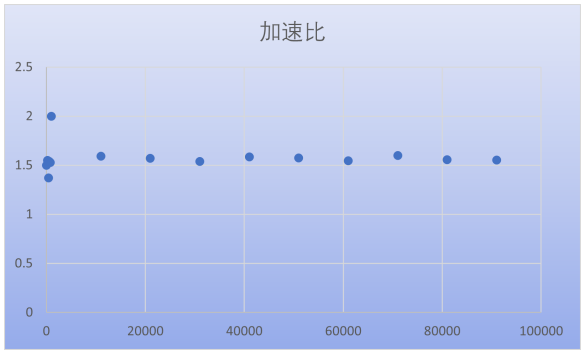
数据规模	平凡算法运行时间/ms	优化算法运行时间/ms	加速比
10	1.7e-05	1.4e-05	1.21429
210	0.00042	0.000251	1.67331
410	0.00089	0.000483	1.84265
610	0.001358	0.000707	1.92079
810	0.001626	0.000943	1.72428
1010	0.0019	0.0012	1.58333
11010	0.0216	0.0128	1.6875
21010	0.0405	0.0247	1.63968
31010	0.1113	0.0361	3.0831
41010	0.1294	0.0585	2.21197
51010	0.1434	0.0805	1.78137
61010	0.1453	0.0875	1.66057
71010	0.1503	0.1141	1.31727
81010	0.1779	0.1131	1.57294
91010	0.2001	0.141	1.41915

2.4 程序结果分析

对于 ARM 平台，程序执行结果，作图分析如下：



(a) 两种算法绝对时间对比

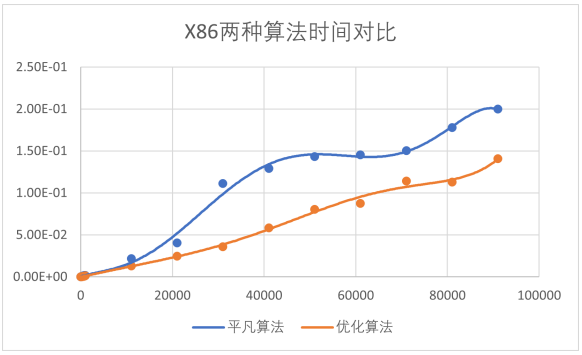


(b) 加速比

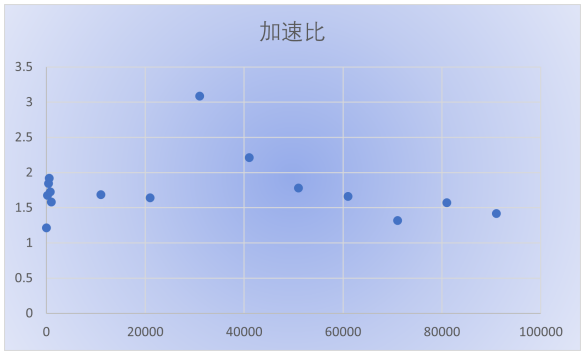
图 2.4: ARM 不同算法的执行时间对比

分析上面两幅图可知，cache 算法优化与原来按列访问的平凡算法相比，对于运行时间的确有了优化，且在问题规模较大时，加速比也在不断上升，这个优化效果更加明显，矩阵规模达到 7000+ 时，cache 算法的优势几乎是平凡算法的 3 倍。出现这种情况从理论上分析来说是因为采用 Cache 优化算法，是 cache 命中率增大，不用每进行一次运算就进行一次内存读取，极大减小了读取内存时间，具体分析还可通过 VTnue 工具来定性描述。

对于 X 86 平台程序执行结果，作图分析如下：



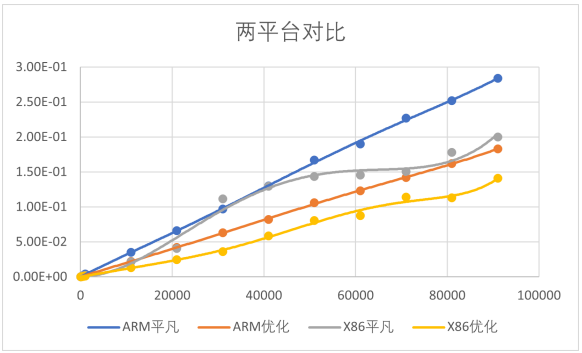
(a) 两种算法绝对时间对比



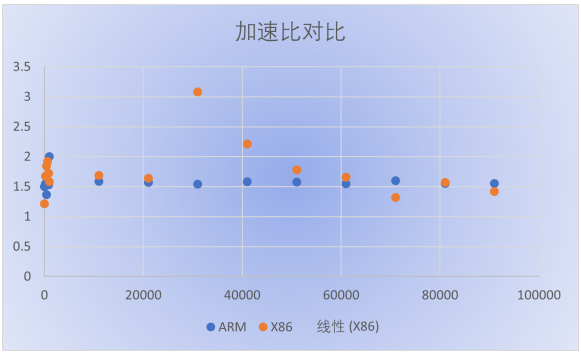
(b) 加速比

图 2.5: X 86 不同算法的执行时间对比

整体规律表现基本与 ARM 平台相同，加速比变化趋势与两种算法的绝对时间对比变化趋势都几乎相同，但是 X 86 平台有一个异常情况，在数据规模较小时，出现了优化算法效果不如平凡算法的情况，按理来说不应该出现这种状况，使用 VTune 工具查看一下具体发生了什么。ARM 和 X 86 平台对比分析



(a) 两平台绝对时间对比



(b) 加速比

图 2.6: 两平台性能对比

整体而言，两者变化趋势几乎一样，但是横向对比，本机由于 cache 较大，性能会比在鲲鹏服务器上跑的 arm 平台性能更优，主要是读取内存的时间较短，可用 VTune 工具具体查看分析。

2.5 循环展开优化

对于此实验又进行了循环展开优化，采用了每次循环增加迭代次数，用不同的迭代次数来看循环展开的优势。

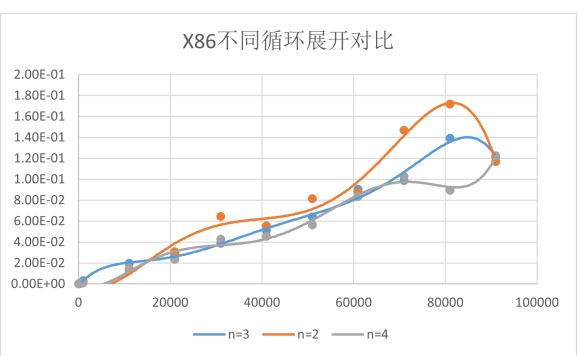
伪代码如下

循环展开伪代码

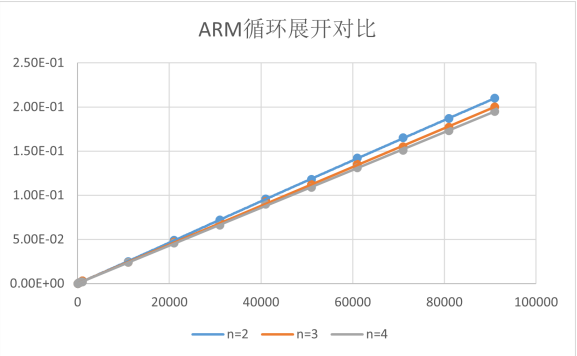
```
void unroll()
{
        timepoint head;
        repeat=0;
        while(repeat<cir)
        {
            long sum1 = 0;
            long sum2 = 0;
            .....
            long sumn=0;
            for (int i = 0;i < scale; i += n)
            {
                sum1 += a[i];
                sum2 += a[i + 1];
                .....
                sumn += a[i + n - 1];
            }
            sum = sum1 + sum2 + ...+sumn;
            repeat++;
        }
        timespoint tail;
        double space1=(tail-head)/repeat;

}
```

不同超标量下的运行时间结果如下：



(a) 不同循环展开对比



(b) 加速比

图 2.7: 两平台的不同循环展开对比

3 结果分析

3.1 VTune 分析

不同平台，不同算法最后运行时间有所不同，这是因为在两个平台上不同 cache 大小，采用不同的代码编写，线程不同，对 cache 的利用率也不同。运用 VTnue 工具对 ARM 平台不同算法的 L1,L2,L3cache 利用率分析如下表所示：

	实验一平凡算法	实验一优化算法	实验二平凡算法	实验二优化算法
	Hardware Event Count			
L1_HIT	1,268,435,498	1,287,742,845	378,585	16,026,744
L1_MISS	11,722,887	1,842,975	27,360	154,164
L2_HIT	9,482,208	1,763,232	10,335	91,622
L2_MISS	2,240,853	79,926	17,061	62,685
L3_HIT	2,065,698	71,103	28,935	26,706
L3_MISS	97,485	41,979	5,679	22,221

从表中可以看出，优化算法之所以快，要么是因为大幅度增大了 L1cache_HIT，或者大幅减小了 L1cache_MISS, 这样使读取内存不至于过多浪费，极大地利用了 cache 的特性。从这里我们也可以看出，如果想要提高程序的有异性，在设计算法时，我们也要考虑并行设计，以达到更好的效果。