南開大學
Nankai University

计算机学院

并行程序设计作业

体系结构相关性能测试

姓名：杜岱玮

学号：2011421

专业：计算机科学与技术

2022 年 3 月 6 日

目录

1 cache 优化

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：

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 实验设计

我对平凡算法和优化算法进行编程实现，然后在鲲鹏服务器上对不同问题规模下两种算法的效率进行对比，并观察不同问题规模对优化效果的影响。

1.2.1 平凡算法

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，代码如下：

逐列访问平凡算法

```
void task11(int n)
{
  for(int i = 0; i < n; i++)
    p[i] = 0;                    //数组p储存计算结果
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
      p[i] += b[j] * a[j][i]; //二维数组a储存矩阵，数组b储存向量
}
```

1.2.2 优化算法

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。代码如下：

逐列访问平凡算法

```
void task12(int n)
{
  for(int i = 0; i < n; i++)
    p[i] = 0;
  for(int j = 0; j < n; j++)
    for(int i = 0; i < n; i++)
      p[i] += b[j] * a[j][i];
}
```

1.2.3 精确计时

由于在较小的问题规模下算法用时很少，为了提高计时精度，我采用了多次运行取平均的方法。我写了一个工具函数 run 用于对各种不同算法计时：

逐列访问平凡算法

```
double run(void (*f)(int), int arg, int times)
{
  struct timeval t1, t2;
    gettimeofday(&t1, NULL);
  while(times--)
    f(arg);
  gettimeofday(&t2, NULL);
  double t = (t2.tv_sec-t1.tv_sec) * 1000000 + t2.tv_usec-t1.tv_usec;
  return t / 1000000;
}
```

run 函数返回以 arg 为参数运行 f 函数 times 次的用时，使得测试各种不同算法的效率变得容易。

1.2.4 测试代码

为了研究不同问题规模下的观察效果，我选择了多个不同问题规模进行测试。

由于较小规模的问题下程序运行时间较短，测试较为不准确，我希望对较小规模的情形运行较多次；而当问题规模较大，运行一次所需时间较长，我希望运行次数少一些。

对于 $n \times n$ 的矩阵，算法每次大约执行 n^2 次。为了使不同规模下的测试时间大致相同，我对矩阵大小为 $n \times n$ 的情形运行 $\frac{tot}{n^2}$ 次，其中 tot 是一个预先设定的常数。

下面是我的测试代码。不同的测试规模存储在数组 test_list1 中，测试结果存储在数组 r1 中。

逐列访问平凡算法

```
void task1()
{
    const int tot = 100000000;
    const int test_list1[7] = { 100,400,700,1000,2000,4000,5000 };
    double r1[7][2];
  for(int i = 0; i < 7; i++)
  {
    int arg = test_list1[i];
    int times = std::min(1000, tot / (arg*arg));
    r1[i][0] = run(task11, arg, times) / times;
    r1[i][1] = run(task12, arg, times) / times;
  }
}
```

1.3 实验数据

1.3.1 硬件参数

我在鲲鹏服务器上单核运行我的测试代码。我通过 lscpu 命令得到下列 CPU 缓存参数：

- L1d cache: 64K
- L1i cache: 64K
- L2 cache: 512K
- L3 cache: 49152K

1.3.2 测试数据

测试所得的数据如下表所示，第一列表示向量大小，第二列第三列分别为两种算法在相应数据上的运行时间。

问题规模	普通算法/s	优化算法/s
100	0.000049	0.000043
400	0.000766	0.000688
700	0.002387	0.002240
1000	0.005170	0.004530
2000	0.022040	0.017440
4000	0.128548	0.078387
5000	0.251250	0.113250

1.4 数据分析

可以发现，优化算法在较大的数据规模上取得了更好的效果。将数据绘制成统计图后这一点更加明显：

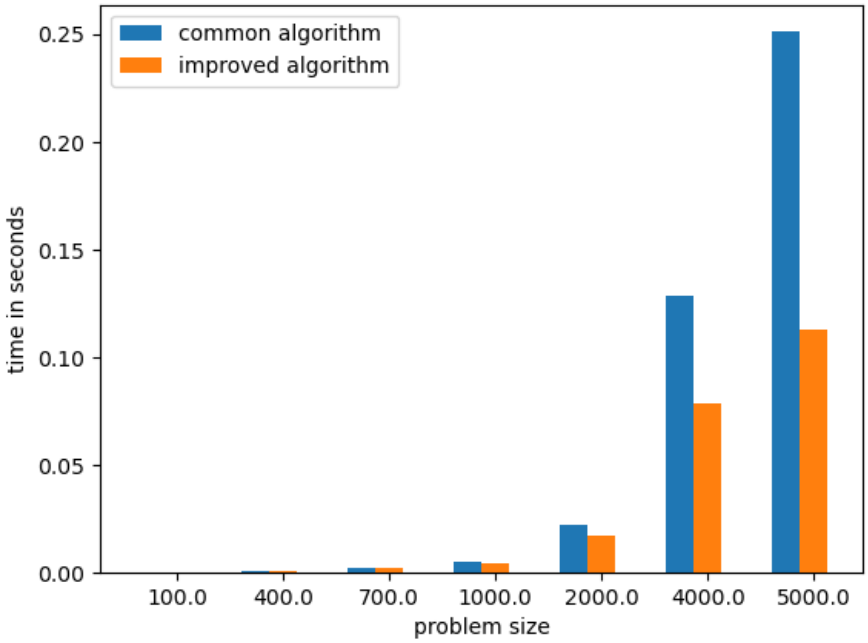

图 1.1: 运行时间

对于小数据上图显示不清楚，我又计算了 cache 优化算法相对于普通算法的提升比例，结果如图：

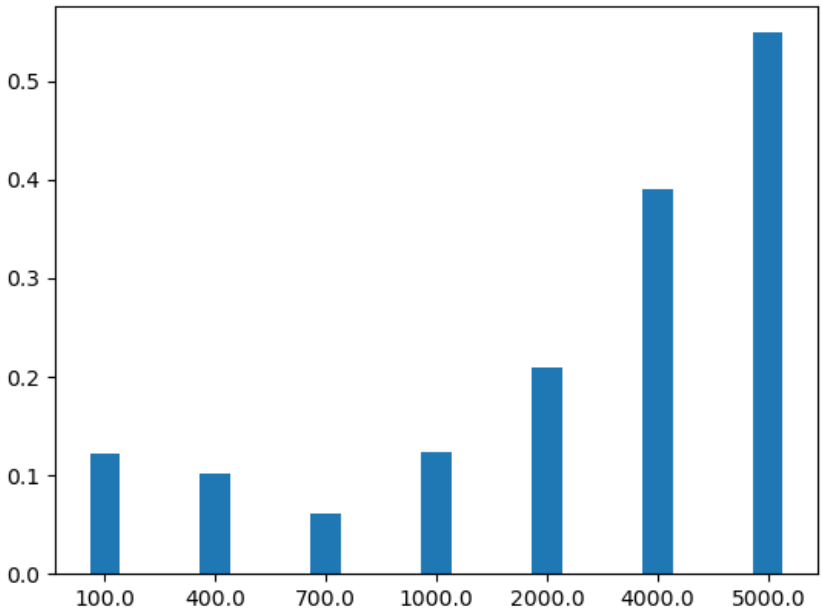

图 1.2: 提升比例

可见，当数据规模在 1000 以下时，提升不明显；当数据规模大于 1000，规模越大提升越明显，特别是当规模为 5000，提升比例超过 0.5。

我认为从缓存方面可以解释提升比例的变化。一级缓存大小为 64K，可以存下 16384 个 int 型整数；二级缓存大小为 512K，可以存下 131072 个 int 型整数。当规模小于 1000，一、二级缓存中足以放下矩阵的相当大一部分，因此普通算法也可以受到缓存的很大优化，导致针对缓存的优化并不明显。当规模大于 1000，优化算法还是可以有效利用缓存；但对于普通算法，每次循环的地址在内存中跳跃很大，很快就跳出被加载进缓存的那部分内存从而导致缓存需要重新加载，因此缓存的利用率很低，加速效果就不理想了。这就解释了为什么数据规模越大优化算法的提升效果越好。

1.5 利用 perf 测量缓存命中率

为了验证我的猜测，我又使用 perf 测量了两种算法在不同数据规模下的缓存不命中率，原始数据如下：

- problem size: 100 400 700 1000 2000 4000 5000
- tot miss count: 873476005
- common alg miss(%): 0.04 0.47 1.46 4.80 9.34 7.95 8.20
- improved alg miss(%): 0.05 0.39 0.40 0.41 0.42 0.40 0.43
- tot load count: 65952188807
- common alg load(%): 0.20 1.98 1.98 1.97 1.97 1.89 1.97
- improved alg load(%): 0.20 1.98 1.98 1.97 1.97 1.89 1.97

由原始数据计算 cache 不命中率，绘制出下图：

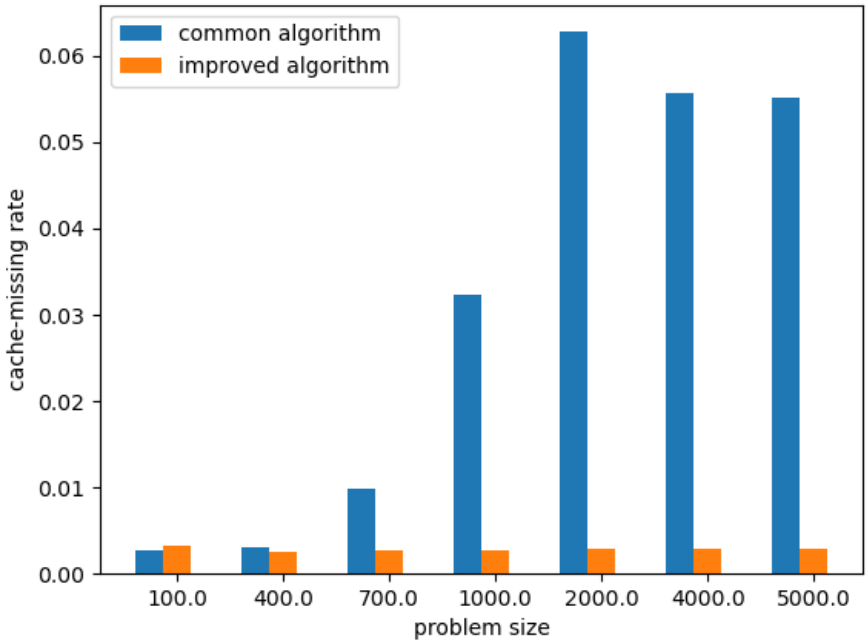

图 1.3: 两种算法的缓存不命中率

很明显，当数据规模大于 1000 时，平凡算法的缓存不命中率大大增加了，而缓存优化算法的缓存不命中率则一直保持在很低的水平；这就解释了为何当数据规模大于 1000 时缓存优化的效果大大增加。

2 超标量优化

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。完成如下作业：

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。

2.2 实验设计

我实现并测试了四类求和算法，包括普通求和算法、多路链式求和算法、递归算法以及改变求和顺序算法。对于每一类算法，我针对不同的数据规模和具体实现分别做了测试，观察其性能差异。

2.2.1 平凡算法

平凡算法逐个访问待加数据，并将它们逐一加到累加器上。代码实现如下：

平凡算法

```
void task21(int n)
{
  int sum = 0;
  for(int i = 0; i < n; i++)
    sum += c[i];
}
```

2.2.2 多路链式累加算法

n 路链式累加算法使用 n 个累加器，每次访问 n 个待加数据，将它们分别加到对应的累加器上。由于这 n 个累加器彼此独立，cpu 可以把针对它们的指令分配给不同的流水线执行，从而达到加速的目的。如果流水线数量足够，n 路链式算法最多可达到 n 倍的加速。下面是两个例子，分别为 2 路和 8 路链式算法：

多路链式累加算法

```
void task22(int n)
{
  int sum = 0, tmp[2] = {};
  int i;
  for(i = 0; i + 2 - 1 < n; i += 2)
    tmp[0] += c[i], tmp[1] += c[i + 1];
  for(i; i < n; i++)
    sum += c[i];
  sum += tmp[0] + tmp[1];
}
void task228(int n)
{
  int sum = 0, tmp[8] = {};
  int i;
  for(i = 0; i + 8 - 1 < n; i += 8)
    tmp[0] += c[i], tmp[1] += c[i + 1],
    tmp[2] += c[i + 2], tmp[3] += c[i + 3],
    tmp[4] += c[i + 4], tmp[5] += c[i + 5],
    tmp[6] += c[i + 6], tmp[7] += c[i + 7];
  for(i; i < n; i++)
    sum += c[i];
  sum += tmp[0] + tmp[1] + tmp[2] +
       tmp[3] + tmp[4] + tmp[5] + tmp[6] + tmp[7];
}
```

2.2.3 递归算法

所有数据两两相加、中间结果再两两相加，依次类推，直至只剩下最终结果。由于各个计算步骤基本互不依赖，理论上该算法可以最大程度地发挥多条流水线的优势。我采用了循环实现，并实现了两个版本，在其中一个版本中运用循环展开技术以减少循环控制语句的开销：

递归算法和带循环展开的递归算法

```
void task231(int n)
{
  while(n > 1)
  {
    int i;
    for(i = 0; i < n/2; i++)
      c[i] += c[n - i - 1];
    n /= 2;
  }
}
void task2316(int n)
{
  while(n > 1)
  {
    int i;
    for(i = 0; i + 16 < n/2; i += 16)
    {
      c[i] += c[n - i - 1];
      c[i + 1] += c[n - i - 1 - 1];
      c[i + 2] += c[n - i - 2 - 1];
      c[i + 3] += c[n - i - 3 - 1];
      c[i + 4] += c[n - i - 4 - 1];
      c[i + 5] += c[n - i - 5 - 1];
      c[i + 6] += c[n - i - 6 - 1];
      c[i + 7] += c[n - i - 7 - 1];
      c[i + 8] += c[n - i - 8 - 1];
      c[i + 9] += c[n - i - 9 - 1];
      c[i + 10] += c[n - i - 10 - 1];
      c[i + 11] += c[n - i - 11 - 1];
      c[i + 12] += c[n - i - 12 - 1];
      c[i + 13] += c[n - i - 13 - 1];
      c[i + 14] += c[n - i - 14 - 1];
      c[i + 15] += c[n - i - 15 - 1];
    }
    for(i; i < n/2; i++)
      c[i] += c[n - i - 1];
    n /= 2;
  }
}
```

2.2.4 改变运算顺序算法

首先运用循环展开，然后在连加表达式中加入括号改变运算顺序，使得不同括号间的运算互不依赖；这事实上是在连加表达式中运用上面提到的递归算法。下面的代码中循环展开长度为 8:

改变运算顺序算法

```
void task248(int n)
```

```
{
  int i, sum = 0;
  for(i = 0; i + 8 < n; i += 8)
  {
    sum += ((c[i] + c[i + 1]) +
    (c[i + 2] + c[i + 3])) +
    ((c[i + 4] + c[i + 5]) +
    (c[i + 6] + c[i + 7]));
  }
  for(i; i < n; i++)
    sum += c[i];
}
```

2.2.5 精确计时

计时方法与上一个实验相同，详见 1.2.3.

2.3 实验数据

在鲲鹏服务器上运行各种算法，所得结果如下表所示：

问题规模	16384	32768	65536	131072	262144	524288	1048576
平凡算法	0.000048	0.000097	0.000194	0.000388	0.000781	0.001566	0.003134
2 路链式	0.000031	0.000063	0.000127	0.000253	0.000507	0.001016	0.002061
3 路链式	0.000024	0.000049	0.000098	0.000195	0.000391	0.000784	0.001605
4 路链式	0.000022	0.000046	0.000091	0.000183	0.000366	0.000734	0.001500
5 路链式	0.000021	0.000043	0.000086	0.000172	0.000345	0.000692	0.001428
6 路链式	0.000020	0.000042	0.000084	0.000167	0.000336	0.000676	0.001384
8 路链式	0.000019	0.000040	0.000079	0.000158	0.000319	0.000642	0.001311
递归算法	0.000060	0.000121	0.000242	0.000484	0.000982	0.001978	0.004042
递归算法（循环展开）	0.000042	0.000088	0.000176	0.000353	0.000705	0.001427	0.002851
改变运算顺序（4 层展开）	0.000020	0.000042	0.000084	0.000168	0.000336	0.000679	0.001421
改变运算顺序（8 层展开）	0.000017	0.000036	0.000073	0.000145	0.000291	0.000589	0.001247
改变运算顺序（16 层展开）	0.000016	0.000033	0.000066	0.000133	0.000273	0.000549	0.001161

2.4 数据分析

对于这个实验来说，CPU 的流水线数量是一个重要的参数；令人遗憾的是，我并不知道这个参数。然而根据我得到数据，可以相当可靠地对此做出估计。

2.4.1 多路链式算法的路数与用时的关系

下图展示了不同数据规模下多路链式算法的路数与用时的关系：

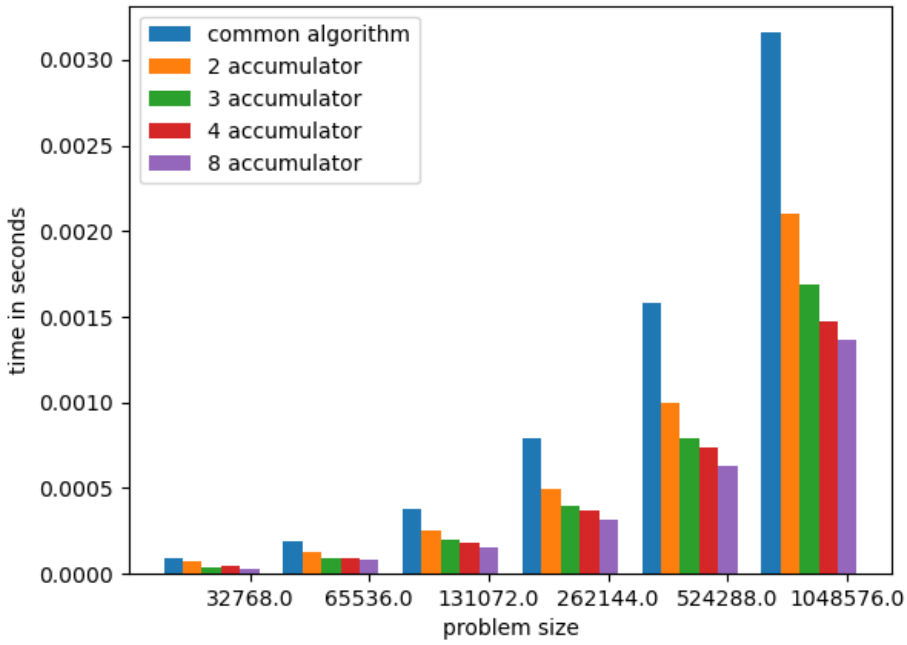

图 2.4: 多路链式算法的用时

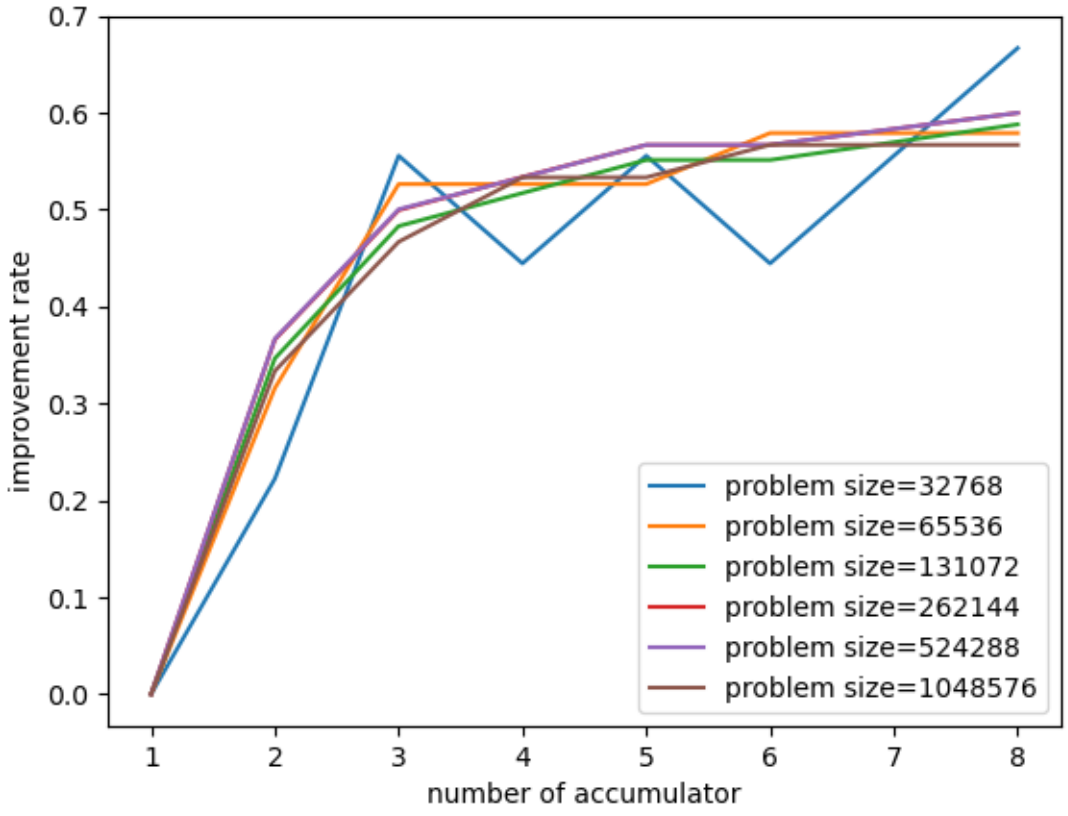

图 2.5: 多路链式算法相对于平凡算法的提升比例

由上图可见，排除掉 n=32768 的异常数据后，多路链式算法的提升率对于各种数据规模是相近的。

链数越多，对性能的提升越大，但提升是有界的。如果 cpu 中有 n 条流水线，则多路链式对性能的提升不可能超过 $\frac{n-1}{n}$。考虑到现实中不太可能达到理想情况，性能提升不可能达到上界。由图可知，当链数增加，提升率收敛在 0.6 左右，在 $\frac{1}{2}$ 和 $\frac{2}{3}$ 之间；所以我据此估计 CPU 中很可能有 3 条流水线。

2.4.2 递归算法的效率不及预期

平凡算法、递归算法、带循环展开的递归算法在不同规模数据上的运行时间对比如图所示：

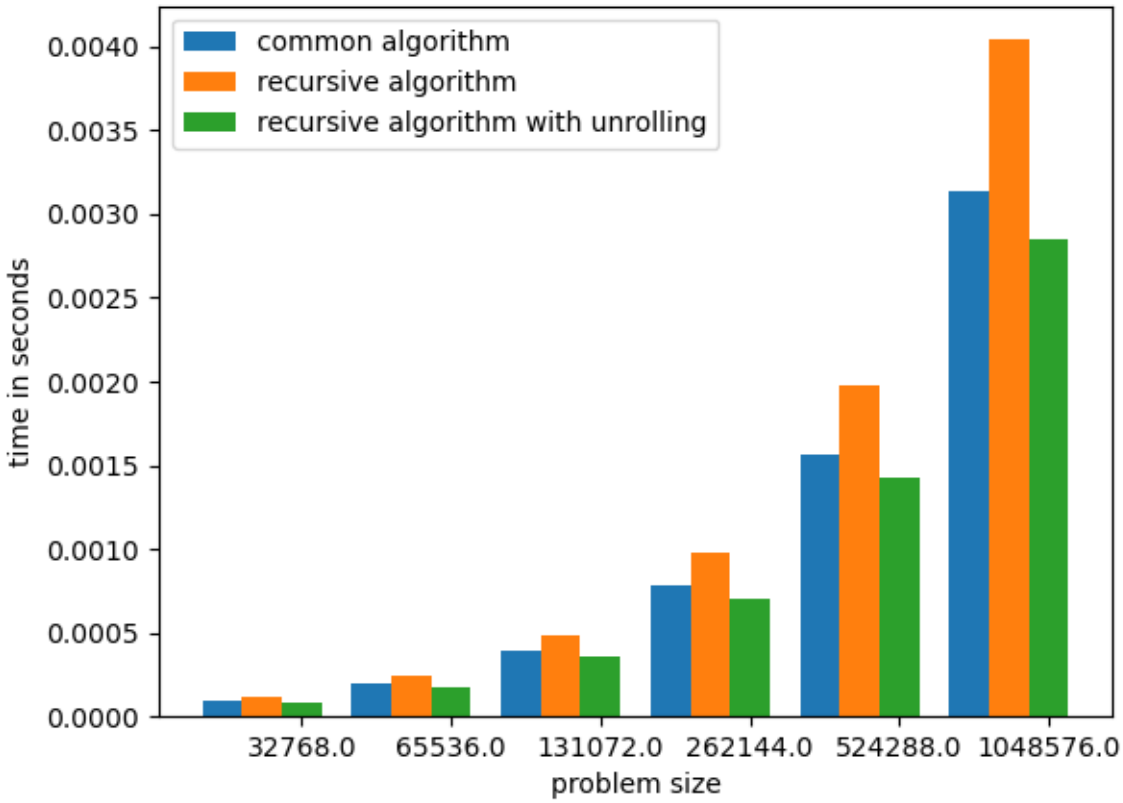

图 2.6: 递归算法与平凡算法的对比

令人意外的是，递归算法在所有数据规模上的效率都不及平凡算法；即使运用了循环展开技术，其效率也仅仅只是稍微高于平凡算法一点点，仍然远逊于多路链式算法。

我认为导致这种情形的关键原因是，递归算法的效率被过多的内存访问操作拖累了。在平凡算法中，程序共访问数组 n 次，而在递归算法中，程序访问数组约 2n 次，因此递归算法的内存访问次数约为平凡算法的二倍。即便顺序访问数组可以很好地利用缓存，过多的内存访问仍是一笔相当大的开销，很可能足以抵消超标量优化带来的优势。

2.4.3 改变运算顺序对性能的提升略优于多路链式算法

改变运算顺序算法通过运用加法结合律改变运算顺序，从而使得程序的指令间依赖降低、更适用于多流水线处理器。改变运算顺序算法的一个关键参数是循环展开因子，即循环每一步的步长；我共实验了 4，8，16 三种循环展开因子，测试结果与平凡算法和多路链式算法的对比如图所示：

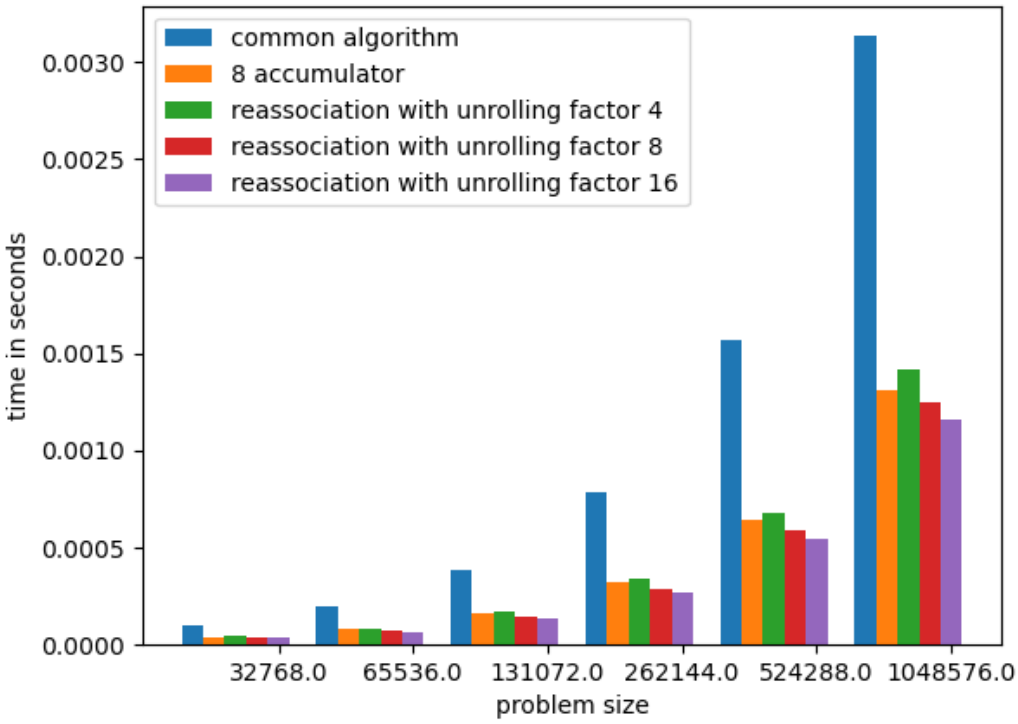

图 2.7: 平凡算法，8 路链式算法，三种改变运算顺序算法对不同数据规模的用时

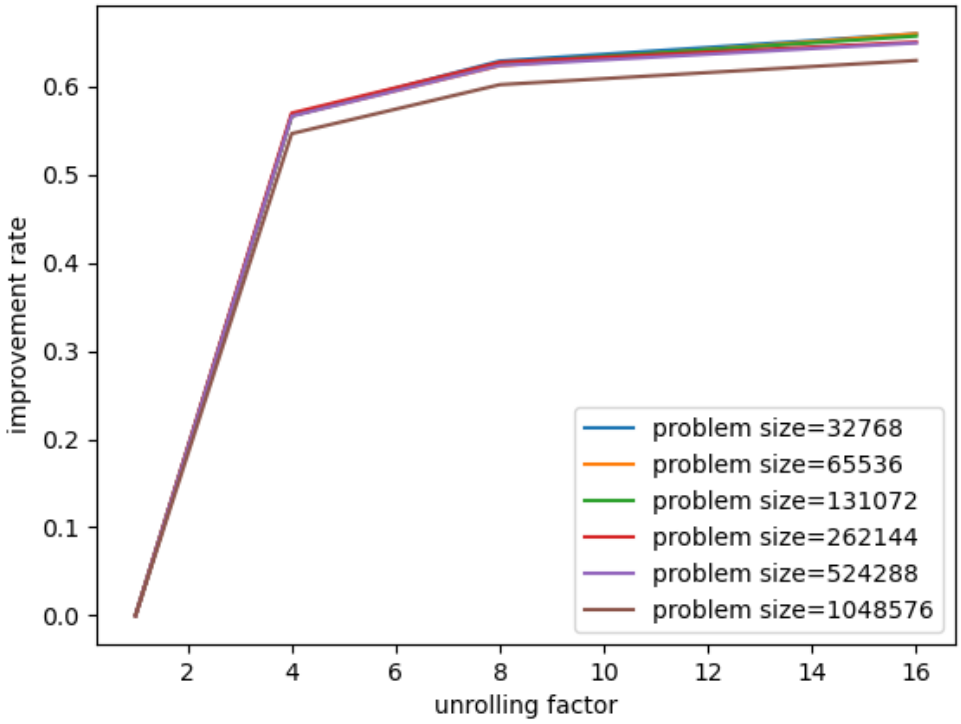

图 2.8: 循环展开因子对改变运算顺序算法相对于平凡算法的提升比例的影响

由图可知，三种改变运算顺序算法与 8 路链式算法在效率上相差不大；或者说，在展开因子足够大的情况下，改变运算顺序算法与多路链式算法相对于平凡算法的提升比例的上界是差不多的。这很合理，因为它们的理论上界都是 $\frac{n-1}{n}$，n 是流水线数量。

同预期一致的是，改变运算顺序算法的性能提升比例随循环展开因子增加而增加。看起来提升比例收敛在 0.6 左右的位置，这同样暗示了 cpu 可能有 3 条流水线（参考 2.4.1）。

循环展开因子为 8 和 16 的改变运算顺序算法比 8 路链式算法略快。我认为这可能是因为改变运算顺序算法的实现更为简洁且没有用到临时变量，所以常数略小。

2.5 总结

鲲鹏服务器的 cpu 很可能有 3 条流水线，多路链式算法和改变运算顺序算法在循环展开因子足够大的情况下可以达到约 $\frac{2}{3}$ 的加速比例；而递归算法由于内存访问消耗过高，加速比例很不理想。