



南開大學  
Nankai University

南开大学

计算机学院和密码与网络空间安全学院

《并行程序设计》实验报告

---

作业二：CPU 架构相关编程

---

姓名：梁景铭

学号：2312632

专业：计算机科学与技术

指导教师：王刚

2025 年 3 月 30 日

## 摘要

本实验主要针对矩阵列向量内积与数组求和两类数值计算问题，设计并实现了平凡算法、缓存优化、超标量优化（包括多链累加与递归分治）以及循环展开（unroll）等多种策略。通过在不同硬件平台和操作系统上的高精度性能测试，探讨了缓存利用和指令级并行对程序性能的影响，验证了优化策略在提高计算效率方面的有效性。实验代码及图片已全部上传至：

[https://github.com/eprogressing/NKU\\_COSC0025\\_Parallel](https://github.com/eprogressing/NKU_COSC0025_Parallel)

关键字：平凡算法，缓存优化，超标量优化，循环展开，性能测试

## 目录

|          |                           |          |
|----------|---------------------------|----------|
| <b>1</b> | <b>基础要求</b>               | <b>1</b> |
| 1.1      | 问题重述                      | 1        |
| 1.2      | 实验环境                      | 1        |
| 1.3      | $n \times n$ 矩阵与向量内积      | 1        |
| 1.3.1    | 优化原理分析                    | 1        |
| 1.3.2    | 代码对比分析                    | 1        |
| 1.3.3    | 性能测试对比                    | 2        |
| 1.4      | $n$ 个数求和                  | 3        |
| 1.4.1    | 优化原理分析                    | 3        |
| 1.4.2    | 代码对比分析                    | 4        |
| 1.4.3    | 性能测试对比                    | 4        |
| <b>2</b> | <b>进阶要求</b>               | <b>5</b> |
| 2.1      | 不同操作系统对比                  | 5        |
| 2.2      | unroll 优化                 | 6        |
| 2.2.1    | 优化原理分析                    | 6        |
| 2.2.2    | unroll 优化的关键代码            | 6        |
| 2.2.3    | unroll 和平凡算法，cache 优化算法对比 | 7        |
| 2.3      | profiling                 | 7        |
| 2.4      | 更多算法设计思路                  | 8        |
| 2.4.1    | 多核和 SIMD 优化               | 8        |
| 2.4.2    | 缓存无关算法                    | 8        |

# 1 基础要求

## 1.1 问题重述

基础实验的主要任务是对两类常见的数值计算问题进行算法设计、实现与性能优化，并通过高精度计时对比不同算法在实际运行中的效率表现。具体来说，实验内容涵盖如下两方面：

### 1. 矩阵列向量内积计算问题

在该部分中，给定一个  $n \times n$  的矩阵和一个向量，要求计算矩阵每一列与该向量的内积。为此，实验将实现并对比如下两种算法：

- ▶ **平凡算法**：直接逐列访问矩阵中的每个元素，按常规方式计算内积；
- ▶ **Cache 优化算法**：通过调整内存访问顺序，优化缓存使用，从而提高运算效率。

### 2. 数组求和问题

针对一组包含  $n$  个数的数组，实验要求计算其总和。为此，设计了两种不同的求和策略：

- ▶ **链式累加法**：按顺序逐个累加，得到最终和；
- ▶ **超标量优化方法**：采用指令级并行技术，如两路链式累加，或者利用递归分治策略（即两两相加、逐步合并）实现高效求和。

实验全部代码及图片已上传到 github 上。

## 1.2 实验环境

| 参数       | ARM 架构   | x86 架构             |
|----------|----------|--------------------|
| CPU 型号   | 华为鲲鹏 920 | AMD Ryzen 9 7945HX |
| 基础主频     | 2.6GHz   | 2.50 GHz           |
| L1 Cache | 64KB     | 1.0MB              |
| L2 Cache | 512KB    | 16.0 MB            |
| L3 Cache | 49152KB  | 64.0MB             |

## 1.3 $n \times n$ 矩阵与向量内积

### 1.3.1 优化原理分析

矩阵在内存中按行连续存储，而原始逐列访问算法每次仅利用缓存中的部分数据，频繁的访存操作大大增加了延时。为此，我们首先采用 Cache 优化策略，将加载到缓存的一整行数据用于所有内积计算，从而充分利用缓存，减少访存次数。

### 1.3.2 代码对比分析

平凡算法直接在内层循环中进行内积累加，每次更新 `result[i]` 时都需要重新加载矩阵数据。由于矩阵 `a` 在内存中是按行连续存储的，逐列访问不能充分利用缓存的连续性，从而频繁触发访存操作，增加了延迟。

为充分利用缓存，每次加载一整行数据后，我们调整了循环顺序：首先初始化所有 `result` 元素，然后以行为外层循环，内层循环更新所有对应列的结果。这样可以在加载一行数据后，利用缓存中连续的数据同时更新多个 `result` 元素，减少访存次数，提高性能。下面给出代码及注释说明：

### 平凡算法

```
1 // 对于每个列索引 i, 初始化 result[i] 并逐行累加内积
2 for (int i = 0; i < n; i++) {
3     result[i] = 0.0;           // 初始化当前结果
4     for (int j = 0; j < n; j++)
5         result[i] += a[j][i] * b[j]; // 累加来自每一行的乘积
6 }
```

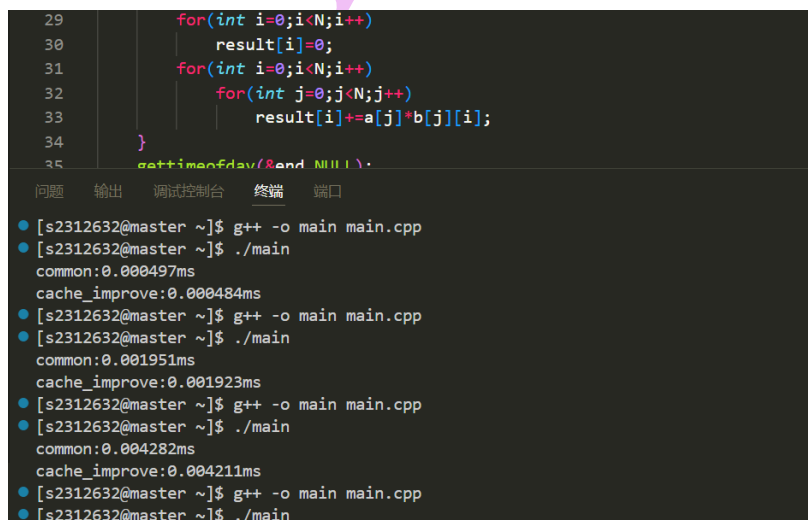
### Cache 优化算法

```
1 // 先对所有结果进行初始化
2 for (int i = 0; i < n; i++)
3     result[i] = 0.0;
4
5 // 外层循环按行遍历, 充分利用每行连续存储的数据
6 for (int j = 0; j < n; j++) {
7     // 内层循环更新所有列对应的结果
8     for (int i = 0; i < n; i++)
9         result[i] += a[j][i] * b[j]; // 利用当前行的数据更新 result
10 }
```

### 1.3.3 性能测试对比

对两种算法在 ARM 架构上的华为鲲鹏服务器进行测试。由于篇幅原因, 如何用 vscode 远程连接服务器, 及下文 linux 系统的安装不做详细赘述。

本次性能测试实验将矩阵大小  $N$  分为三个组: 第一组为  $N = 10$  至  $90$  (步长为  $10$ ), 代表小规模问题; 第二组为  $N = 100$  至  $900$  (步长为  $100$ ), 代表中等规模问题; 第三组为  $N = 1000$  至  $9000$  (步长为  $1000$ ), 代表大规模问题。以下为  $N=10,20,30$  时的代码运行截图, 以作证明:



```
29     for(int i=0;i<N;i++)
30         result[i]=0;
31     for(int i=0;i<N;i++)
32         for(int j=0;j<N;j++)
33             result[i]+=a[j]*b[j][i];
34     }
35     gettimeofday(&end,NULL);

问题 输出 调试控制台 终端 窗口
• [s2312632@master ~]$ g++ -o main main.cpp
• [s2312632@master ~]$ ./main
common:0.000497ms
cache_improve:0.000484ms
• [s2312632@master ~]$ g++ -o main main.cpp
• [s2312632@master ~]$ ./main
common:0.001951ms
cache_improve:0.001923ms
• [s2312632@master ~]$ g++ -o main main.cpp
• [s2312632@master ~]$ ./main
common:0.004282ms
cache_improve:0.004211ms
• [s2312632@master ~]$ g++ -o main main.cpp
• [s2312632@master ~]$ ./main
```

图 1.1: 代码运行截图

| Group 1 |          |               | Group 2 |          |               | Group 3 |           |               |
|---------|----------|---------------|---------|----------|---------------|---------|-----------|---------------|
| N       | common   | cache_improve | N       | common   | cache_improve | N       | common    | cache_improve |
| 10      | 0.000497 | 0.000484      | 100     | 0.05064  | 0.048206      | 1000    | 5.96454   | 4.88232       |
| 20      | 0.001951 | 0.001923      | 200     | 0.199102 | 0.190763      | 2000    | 26.7959   | 19.9792       |
| 30      | 0.004282 | 0.004211      | 300     | 0.457049 | 0.436359      | 3000    | 57.3313   | 49.9696       |
| 40      | 0.007992 | 0.007824      | 400     | 0.821042 | 0.764997      | 4000    | 225.9123  | 86.3145       |
| 50      | 0.012765 | 0.012626      | 500     | 1.36919  | 1.19199       | 5000    | 350.2519  | 148.8251      |
| 60      | 0.17142  | 0.16661       | 600     | 1.91770  | 1.72783       | 6000    | 522.2582  | 211.1875      |
| 70      | 0.24564  | 0.23699       | 700     | 2.43425  | 2.21932       | 7000    | 855.6485  | 284.1634      |
| 80      | 0.30169  | 0.29172       | 800     | 3.63680  | 3.09371       | 8000    | 1396.1347 | 386.5762      |
| 90      | 0.40249  | 0.38815       | 900     | 4.75876  | 3.92989       | 9000    | 1600.1537 | 470.4953      |

表中列出了 `common` 与 `cache_improve` 两种算法在不同规模  $N$  下的运行时间。总体而言，随着  $N$  的增加，两种方法的时间均随运算规模上升，但 `cache_improve` 表现出更优的性能。对于小规模（如  $N \leq 100$ ），两者差异较小，主要因为矩阵较小导致缓存不命中带来的影响有限。随着  $N$  增大，`cache_improve` 逐渐展现出更好的缓存局部性，从而比 `common` 更快，尤其在千级或更大规模时差距更为明显。在实验过程中，我发现在  $N > 1000$  过后，程序运行的速度显著变慢。总体结论是，随着问题规模增长，优化访存顺序能显著提升矩阵乘法的性能。

将实验数据分为三组 10-90,100-900,1000-9000 分别进行可视化对比，如下图所示：

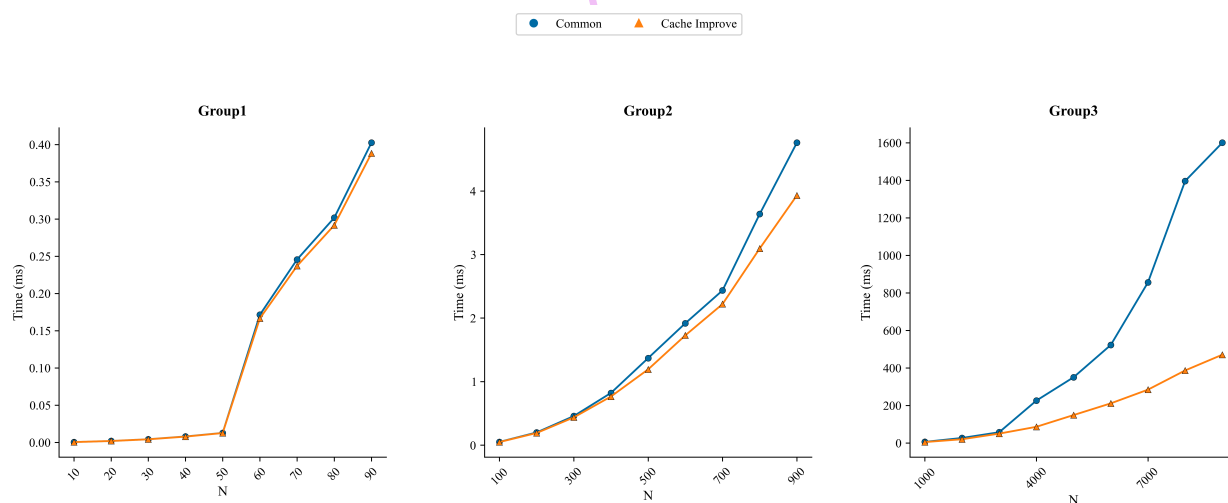


图 1.2: 平凡算法和 cache 优化时间运行可视化对比

## 1.4 n 个数求和

### 1.4.1 优化原理分析

在处理  $N$  个数求和的问题时，传统的顺序算法往往只使用一个累加器变量，这意味着所有的加法操作都在同一条流水线上进行，从而无法利用 CPU 的超标量执行优势。为了解决这一瓶颈，可以采用一种多链路的方法——在同一次循环中同时使用多个临时累加器，将不同数据段的累加操作并行执行。这样不仅能够使得计算过程并行化，还能通过循环展开减少循环迭代次数，进一步降低循环开销。而

递归分治策略通过分治思想将数组分为两部分逐步合并，同样实现了并行加速的效果。

需要注意的是，为了使实验具有公平性，传统的单链累加方法也应当采用相同程度的循环展开。这样一来，各方法在相似的循环结构下进行比较，能更准确地反映出多链路并行累加在充分利用 CPU 并行计算能力方面的优势。

### 1.4.2 代码对比分析

#### 平凡算法

```
1 // 平凡算法：依次遍历每个元素，使用单一累加器进行累加
2 for (int i = 0; i < n; i++)
3     sum += a[i]; // 累加每个元素到sum中
```

#### 链式累加

```
1 // 链式累加：通过循环展开同时使用两个累加器进行累加
2 for (int i = 0; i < n; i += 2)
3 {
4     sum1 += a[i];    // 将当前元素累加到sum1中
5     sum2 += a[i+1]; // 将下一个元素累加到sum2中
6 }
7 // 循环结束后，将两个累加器的结果相加，得到最终的和
8 sum = sum1 + sum2;
9
10 //四累加器同理，这里不再进行关键代码的演示
```

#### 递归分治策略

```
1 // 递归分治策略：通过递归合并数组两端对称的元素
2 void circle(double *a, long long n) {
3     if (n == 1) return; // 基本情况：当数组中只有一个元素时结束递归
4     else {
5         // 将数组前半部分与后半部分的对称元素相加
6         for (int i = 0; i < n / 2; i++) {
7             a[i] += a[n - i - 1]; // 合并对称位置的两个数
8         }
9         // 递归处理合并后的前半部分数组
10        circle(a, n / 2);
11    }
12 }
```

### 1.4.3 性能测试对比

对四种算法在 ARM 架构上的华为鲲鹏服务器进行测试，测试数据我一开始想从 2 的 0 次方开始，但是发现最开始的数据几乎没有差别，都是约等于 0ms，所以选择了从 2 的 10 次幂一直测试到 2 的

26 次幂，代码运行如下：

```
n=1024 common 0.0007ms
n=1024 cache_improve2 0.0005ms
n=1024 cache_improve4 0.0003ms
n=1024 circle 0.0016ms
-----
n=2048 common 0.0014ms
n=2048 cache_improve2 0.001ms
n=2048 cache_improve4 0.0006ms
n=2048 circle 0.0032ms
-----
n=4096 common 0.0028ms
n=4096 cache_improve2 0.0019ms
n=4096 cache_improve4 0.0012ms
n=4096 circle 0.0063ms
```

图 1.3: 代码运行截图

结果分析如下：

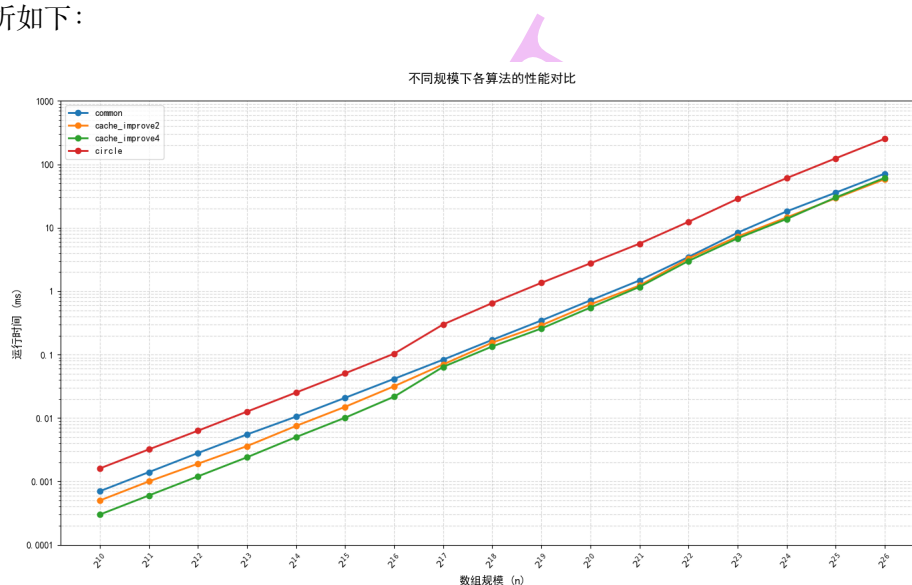


图 1.4: 运行时间对比图

cache\_improve4 凭借四路循环展开和缓存优化全程最优，在大规模数据  $\geq 2^{26}$  时耗时仅为 common 的 60%-80%、circle 的 3%-5%。cache\_improve2 次之，耗时比 common 低 20%-30%。circle 因递归调用和内存访问模式劣势耗时最高，当  $n=2^{26}$  时达 253.8ms，较 cache\_improve4 慢 4.1 倍。

## 2 进阶要求

### 2.1 不同操作系统对比

为了进一步对比平台之间运行时间的差距，我在三个平台上测试了程序的运行速度：分别为 Linux 系统下的 x86 平台、Windows 系统下的 x86 平台以及 Linux 系统下的 ARM 平台（鲲鹏服务器）。每次测试时，问题规模  $n$  按照倍增方以 2 为底的指数级变化。

另外，我在鲲鹏服务器上采用了 STL 的 vector 来动态分配数组，来和最初只设定单独  $N$  值的运行代码进行比较，发现动态分配数组之后运行时间会明显增加，因为篇幅原因，具体操作不再说明。

系统性能对比分析

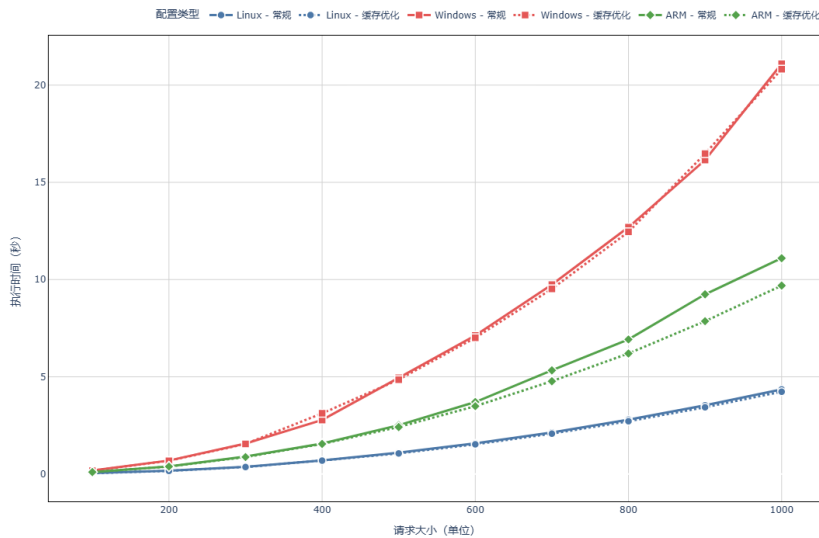


图 2.5: 不同操作系统下系统性能对比分析

结果如下：Linux 在整体性能和扩展性上表现最优，其平凡算法及缓存优化耗时均显著低于其他系统，尤其在缓存优化后执行效率提升最为明显；Windows 系统整体效率偏低，耗时约为 Linux 的 4-5 倍且缓存优化效果微弱。所有系统均呈现请求规模与耗时的正相关性，其中 Linux 增长斜率最平缓。

## 2.2 unroll 优化

### 2.2.1 优化原理分析

unroll 算法通过将循环体中需要重复执行的语句“展开”为多份，可以减少循环迭代次数，从而降低这些控制指令所占用的 CPU 资源。循环展开后，每次执行更多计算，减少了条件判断和跳转指令的频率，从而降低了由分支预测失败或跳转延迟带来的性能损耗。

在 unroll 函数中，通过一次循环计算了 10 个不同位置的值，这些累加器彼此独立。这样，编译器和 CPU 都有机会将这些不相关的运算并行调度到多个执行单元上，同时执行多条指令。外层循环控制“逐列”处理，内层循环对每个位置进行累加。由于每次外层循环处理 10 个元素，所以外层循环的迭代次数相较于没有展开时降低了 10 倍，从而减少了循环控制指令的开销。

### 2.2.2 unroll 优化的关键代码

```

1  for(int i=0; i<N; i++)
2      result[i] = 0;
3  for(int j=0; j<N; j+=5)
4  {
5      int tmp0=0, tmp1=0, tmp2=0, tmp3=0, tmp4=0;
6      for(int i=0; i<N; i++)
7      {
8          tmp0 += a[j+0] * b[j+0][i];

```



```
9      tmp1 += a[j+1] * b[j+1][i];
10     tmp2 += a[j+2] * b[j+2][i];
11     tmp3 += a[j+3] * b[j+3][i];
12     tmp4 += a[j+4] * b[j+4][i];
13 }
14 result[j+0] = tmp0;
15 result[j+1] = tmp1;
16 result[j+2] = tmp2;
17 result[j+3] = tmp3;
18 result[j+4] = tmp4;
19 }
```

### 2.2.3 unroll 和平凡算法, cache 优化算法对比

在这里我选择  $n \times n$  矩阵与向量内积的 group2 进行性能对比, 用折线图对比如下, 可以观察到性能提升了约 17.3%。

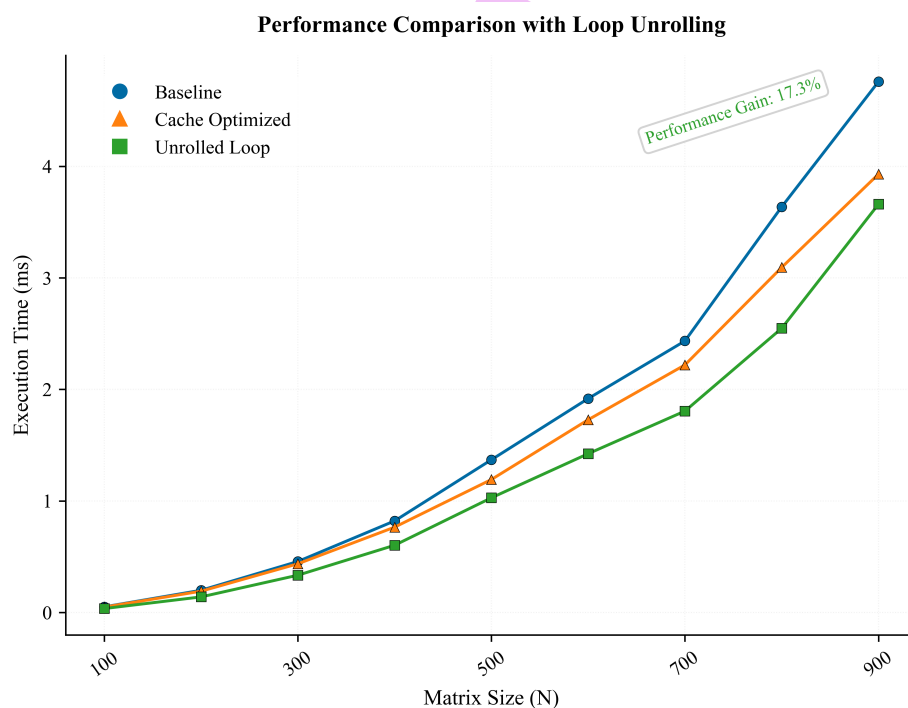


图 2.6: unroll 和平凡算法, cache 优化算法对比

## 2.3 profiling

因为电脑是 AMD 的 CPU, 无法利用 vtune 进行 profiling, 我选择在 linux 系统上利用 perf 工具测试。准备工作需要将原来测试的代码进行文件分割, 即把单独的方法写在一个程序里方便测试。安装 perf 之后, 用 g++ debug 之后, 运行如下代码:

```
1 sudo perf stat -e cycles,instructions ./your profile
```

```
2 sudo perf record ./your profile
3 sudo perf report
```

实在没有办法贴图了，这里我就直接用表格来展示两个程序分别 profiling 之后的结果：

表 1: Problem 1

| Category      | Missing Percentage (%) | Cycles      | Instructions  | CPI   |
|---------------|------------------------|-------------|---------------|-------|
| common        | 50.79                  | 745,611,075 | 1,403,352,186 | 0.531 |
| cache_improve | 29.56                  | 212,332,964 | 506,565,156   | 0.419 |
| unroll        | 23.50                  | 23,421,203  | 54,021,184    | 0.436 |

表 2: Problem 2

| Category      | Missing Percentage (%) | Cycles      | Instructions  | CPI   |
|---------------|------------------------|-------------|---------------|-------|
| common        | 20.20                  | 546,832,827 | 1,010,126,170 | 0.541 |
| cache_improve | 18.93                  | 523,465,362 | 926,086,997   | 0.565 |

## 2.4 更多算法设计思路

查阅资料，思考更多算法设计思路来探索计算机体系结构中 cache 和超标量对程序性能的影响。

### 2.4.1 多核和 SIMD 优化

SIMD[1] 是一种并行计算技术，允许单个指令同时对多个数据元素执行相同的操作。它通过硬件支持的向量寄存器（如 x86 架构的 AVX、ARM 架构的 NEON）实现数据级并行，是现代 CPU 提升计算吞吐量的核心手段。

SIMD 通过扩展寄存器位宽，将多个标量数据打包为向量，单条指令即可完成所有数据批量运算。

### 2.4.2 缓存无关算法

缓存无关算法 [2] 是一类在设计算法时无需依赖具体缓存大小和层次结构的算法。该类算法通常采用递归分治策略，自然地利用数据局部性，从而在多级缓存系统中实现高效的数据传输和计算。正因其具有高度自适应性，缓存无关算法能够在不同硬件平台上取得较优的性能，而无需针对特定硬件进行复杂的调优。

## 参考文献

- [1] R. Sprangler, *SIMD Programming Manual for Linux and Windows*, Apress, 2012.
- [2] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, *Cache-Oblivious Algorithms*, in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, 1999, pp. 285–297.