
南开大学

Nankai University

# 南　　开　　大　　学

## 计　算　机　学　院

**并行程序设计**

---

**体系结构相关编程**

**CPU 架构相关编程**

---

孟启轩

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2025 年 3 月 26 日

# 摘要

结合课堂内容以及实验指导书，我对矩阵与向量内积和 n 个数求和两个问题进行平凡算法和优化算法的设计，通过改变问题规模并测试耗时来对比分析性能。


**关键字：cache 并行 CPU**


# 目录

# 一、 实验环境

- 平台：X86
- CPU：AMD Ryzen 7 5800H with Radeon Graphics
- 核心数 8，逻辑处理器 16
- 内存：16GB

# 二、 矩阵与向量内积

## (一) 问题概述

给定一个 n×n 矩阵，计算每一列与给定向量的内积，设计实现逐列访问元素的平凡算法和 cache 优化算法，进行实验对比。

## (二) 算法设计及原理分析

### 1. 平凡算法设计原理

直接按照矩阵与向量相乘的数学定义实现，逐列遍历矩阵，对每一列与向量进行点积运算，通过最基础的嵌套循环结构完成计算，时间复杂度为 $O(n^2)$。

平凡算法核心代码

```
// 平凡算法：逐列访问（外层循环为列，内层循环为行）
for (int col = 0; col < n; col++){
    result[col] = 0;
    for (int row = 0; row < n; row++){
        result[col] += matrix[row][col] * vec[row];
    }
}
```

### 2. cache 优化算法设计原理

外层循环遍历每一行，内层循环遍历该行的所有列，每次循环中对矩阵的访问更接近内存布局，减少缓存失效。同时，向量在内层循环中保持不变，可被缓存重复利用，而结果的累加操作通过连续访问同一 cache 行来合理利用空间局部性，减少主存访问开销。该优化算法最大化利用 cache 的预取机制和块内局部性，显著降低缓存缺失率，虽然时间复杂度仍为 $O(n^2)$，但计算效率得到提升。

cache 优化算法核心代码

```
//cache 优化算法，改为逐行访问矩阵元素：一步外层循环不计算出任何一个内积，只是向每个内积累加一个乘法结果
for (int col = 0; col < n; ++col){
    result[col] = 0;
}

for (int row = 0; row < n; ++row){
```

```
    for (int col = 0; col < n; ++col){
        result[col] += matrix[row][col] * vec[row];
    }
}
```

# 三、 n 个数求和

## (一) 问题概述

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，直至剩下最终结果，比较平凡算法和优化算法的性能。

## (二) 算法设计及原理分析

### 1. 平凡算法设计原理

通过逐个遍历数组元素并累加求和，直接实现最基础的线性求和操作。

平凡算法核心代码

```
// 平凡算法：逐个累加
int A(const int *a, int n){
    int sum = 0;
    for (int i = 0; i < n; i++){
        sum += a[i];
    }
    return sum;
}
```

### 2. 多链路式优化算法设计原理

每次迭代同时处理相邻的两个元素，将累加操作分为 sum1 和 sum2 两条路径，最终合并结果。这种设计减少了循环迭代次数，并利用 CPU 的指令级并行以及减少循环控制开销。

多链路式优化算法核心代码

```
// 优化算法1：两路链式累加
int B(const int *a, int n){
    int sum1 = 0, sum2 = 0;
    for (int i = 0; i < n; i += 2){
        sum1 += a[i];
        sum2 += a[i + 1];
    }
    return sum1 + sum2;
}
```

### 3. 二重循环优化算法设计原理

初始数组被视作包含所有元素的集合，每次将相邻元素两两相加，结果覆盖前半部分，然后问题规模减半，重复此过程直到只剩一个元素。该算法通过分阶段合并子问题实现求和。

二重循环优化算法核心代码

```
// 优化算法2: 非递归的两两相加
void C(int a[], int n){
    for (int m = n; m > 1; m /= 2){
        for (int i = 0; i < m / 2; i++){
            a[i] = a[2 * i] + a[2 * i + 1];
        }
    }
}
```

# 四、 实验结果分析

## (一) 矩阵与向量内积

初始化向量元素为行号，矩阵元素为行号与列号的和。为提升结果的精确度，我在两个算法外层添加一个 for 循环，每次任务计算 10 次，然后将总的时间除以 10 得到每次任务的平均耗时。测试结果如表 1。

表 1: 不同问题规模下平凡算法和 cache 优化算法的耗时对比（单位：秒）

| 问题规模 | 平凡算法耗时 | Cache 优化算法耗时 |
|---|---|---|
| 100 | 0.000 030 9 | 0.000 019 5 |
| 200 | 0.000 082 8 | 0.000 075 2 |
| 500 | 0.000 638 8 | 0.000 465 8 |
| 1000 | 0.002 950 2 | 0.001 853 8 |
| 2000 | 0.015 134 2 | 0.008 984 5 |
| 5000 | 0.103 106 | 0.051 761 1 |
| 8000 | 0.324 402 | 0.134 055 |
| 10 000 | 0.540 251 | 0.193 189 |

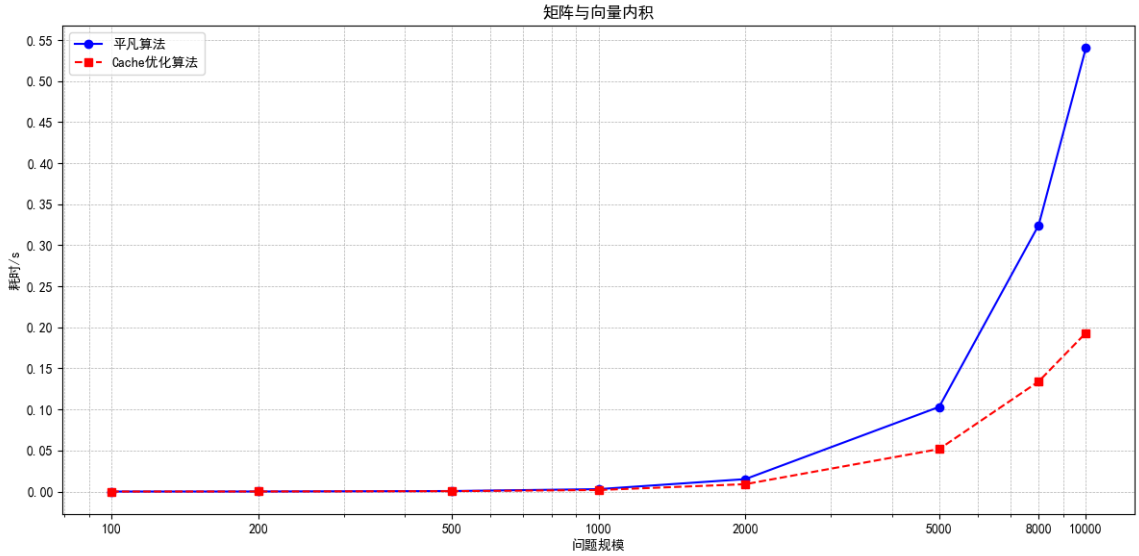


图 1: 不同问题规模下平凡算法和 cache 优化算法的耗时对比

Ryzen 7 5800H 的缓存层级：L1 缓存 512KB，L2 缓存 4MB，L3 缓存 16MB，当然这是 8 个核的，对于本次实验只用到了 1 个核。

当 n 较小时，数据可完全驻留在 L1/L2 缓存中，两种算法性能差异较小，优化算法仅快约 37%。当 n 接近或超过缓存容量时，平凡算法因频繁缓存缺失导致性能骤降，优化算法耗时为平凡算法的 50%，优化算法的性能优势随 n 增大愈发明显。

## (二) cache 优化算法汇编分析

cache 优化算法核心汇编代码

```
获取当前行的基址（matrix[row]）
movq    -40(%rbp), %rax          矩阵指针数组基址
leaq    0(,%rax,8), %rdx         行指针偏移（每行指针间隔8字节）
addq    %rdx, %rax               当前行指针地址
movq    (%rax), %rax             获取当前行的内存基址（matrix[row]）

访问当前元素（matrix[row][col]）
movl    -24(%rbp), %edx          列索引（col）
movslq  %edx, %rdx               32位转64位
salq    $2, %rdx                 列偏移：col × 4（假设元素为int）
addq    %rdx, %rax               当前元素地址
movl    (%rax), %edx             读取matrix[row][col]

获取vec[row]的值
movl    -20(%rbp), %eax          行索引（row）
cltq
leaq    0(,%rax,4), %r8          vec的地址偏移（每个元素4字节）
movq    -48(%rbp), %rax          vec的基址
addq    %r8, %rax                vec[row]的地址
movl    (%rax), %eax             读取vec[row]
```

```
21
22  计算乘积并累加到result[col]
23  imull   %eax, %edx          product = matrix[row][col] * vec[row]
24  movl    -24(%rbp), %eax     列索引（col）
25  cltq
26  leaq    0(,%rax,4), %r8     result的地址偏移
27  movq    -56(%rbp), %rax     result基址
28  addq    %r8, %rax           result[col]的地址
29  movl    (%rax), %ecx        读取原result[col]
30  addl    %ecx, %edx          累加
31  movl    %edx, (%rax)        写回result[col]
```

结合汇编代码分析，外层循环按行遍历，内层按列访问同一行的连续元素，利用空间局部性使缓存预取机制高效加载整行数据，减少缓存未命中；同时，矩阵的行内元素按列连续存储，向量和结果的访问都通过递增索引实现连续内存读写，避免了非连续的跳跃访问。此外，同一行的向量在内层循环中被复用，充分利用时间局部性，减少重复主存访问，而缓存中的当前行数据（包括矩阵块和向量值）在循环中多次复用，降低了缓存替换开销，最终通过减少主存延迟显著提升了计算效率。

## (三) n 个数求和

n 个数我都初始化为 1，测试结果如表 2。

表 2: 不同问题规模下平凡算法、多链路式优化和二重循环优化的耗时对比（单位：微秒）

| 问题规模 | 平凡算法 | 多链路式优化 | 二重循环优化 |
|---|---|---|---|
| $2^{10}$ | 3 | 1 | 2 |
| $2^{12}$ | 12 | 4 | 6 |
| $2^{15}$ | 64 | 25 | 48 |
| $2^{18}$ | 315 | 179 | 402 |
| $2^{20}$ | 933 | 723 | 1582 |
| $2^{22}$ | 3904 | 3428 | 7040 |
| $2^{25}$ | 33 817 | 24 572 | 60 923 |
| $2^{28}$ | 299 390 | 203 479 | 484 153 |
| $2^{30}$ | 1 275 918 | 947 677 | 2 417 648 |

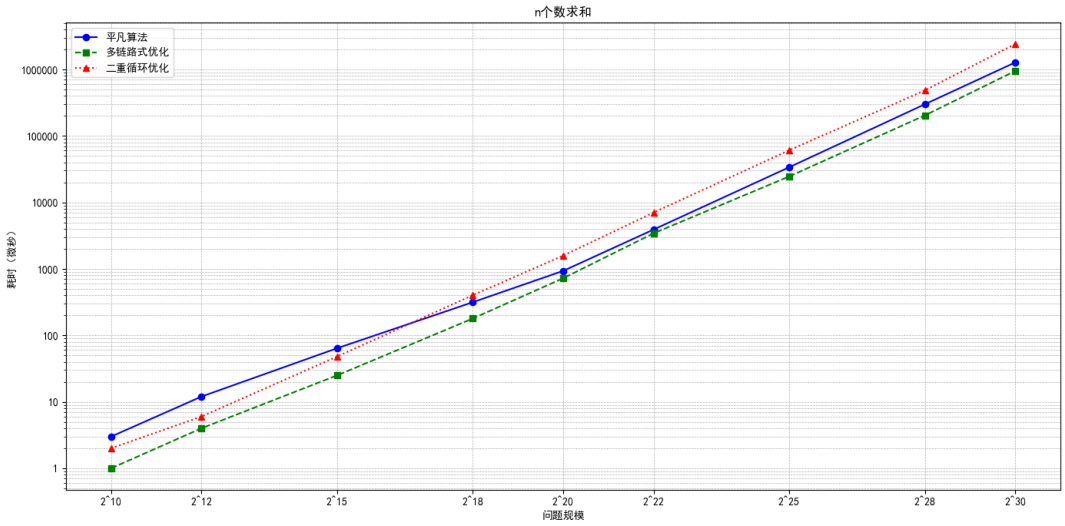


图 2: 不同问题规模下平凡算法、多链路式优化和二重循环优化的耗时对比

实验结果表明，平凡算法凭借顺序访问模式高效利用缓存预取机制，性能比较稳定。多链路式优化算法通过两路累加减少循环迭代次数，在大规模问题时优化十分显著。二重循环优化算法在小规模问题时优化比较明显，在大规模问题时性能崩溃。

二重循环优化算法通过分治思想两两相加，但每次迭代需要访问数组的非连续位置，导致缓存未命中率显著增加。随着问题规模增大，数据量超出缓存容量，虽然理论上时间复杂度为 $O(nlogn)$，但实际操作需要遍历 logn 次数组。此外，算法的循环结构存在数据依赖，导致 CPU 流水线无法有效并行执行指令。

## （四） 多链路式和二重循环汇编分析

双路并行设计利用步长为 2 的循环同时处理相邻元素 a[i] 和 a[i+1]，将内存访问和加法操作拆分为独立指令流，允许超标量架构的多执行单元并行执行。通过寄存器暂存中间结果减少对栈的访问延迟，同时循环迭代次数减半，降低了循环控制开销，从而显著提升指令级并行性和整体性能。

多链路式优化算法核心汇编代码

```
处理a[i]
movl    -12(%rbp), %eax   读取i
cltq
leaq    0(,%rax,4), %rdx  计算a[i]的地址
movq    16(%rbp), %rax
addq    %rdx, %rax
movl    (%rax), %eax      读取a[i]
addl    %eax, -4(%rbp)    累加到sum1

处理a[i+1]
movl    -12(%rbp), %eax   读取i
cltq
addq    $1, %rax          i+1
leaq    0(,%rax,4), %rdx
movq    16(%rbp), %rax
```

```
addq    %rdx, %rax
movl    (%rax), %eax        读取a[i+1]
addl    %eax, -8(%rbp)      累加到sum2
addl    $2, -12(%rbp)       i += 2
```

通过汇编分析,二重循环优化算法的强数据依赖和非连续内存访问导致超标量优化失效。a[2i] 和 a[2i+1] 的地址跳跃式计算破坏缓存局部性，并且 a[i] 要等待前面的读取和加法完成才能计算，后续迭代无法并行；再一个，写入 a[i] 的指令强制等待所有前序操作完成，会导致流水线停滞。这些问题使算法无法利用超标量架构的并行性，性能因内存带宽和依赖延迟而严重退化。

二重循环优化算法核心汇编代码

```
计算a[2i]和a[2i+1]的地址
movl    -8(%rbp), %eax      读取i
addl    %eax, %eax          *2
cltq
leaq    0(,%rax,4), %rdx    地址a[2i]
movq    16(%rbp), %rax
addq    %rdx, %rax
movl    (%rax), %ecx        读取a[2i]

计算a[2i+1]的地址
movl    -8(%rbp), %eax
addl    %eax, %eax
cltq
addq    $1, %rax            *2i +1
leaq    0(,%rax,4), %rdx
movq    16(%rbp), %rax
addq    %rdx, %rax
movl    (%rax), %edx        读取a[2i+1]

相加并写入a[i]
addl    %ecx, %edx          结果 = a[2i] + a[2i+1]
movl    -8(%rbp), %eax
cltq
leaq    0(,%rax,4), %r8     地址a[i]
movq    16(%rbp), %rax
addq    %r8, %rax
movl    %edx, (%rax)        写入a[i]
addl    $1, -8(%rbp)        i++
```



# 五、 总结

对于矩阵与向量内积和 n 个数求和两个问题，分别采用 cache 优化和超标量优化的方法对串行算法进行加速。通过对比平凡算法优化算法，我深刻理解了程序优化的思路与方式，也增强了我对程序测试以及分析的能力。

源代码链接Github