南開大學
Nankai University

计算机学院

并行程序设计实验报告

CPU 架构相关编程

姓名：童汉鑫

学号：2311995

专业：计算机科学与技术

2025 年 3 月 28 日

目录

Abstract

实验 1：计算给定 n*n 矩阵的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与 cache 优化算法。

实验 2：计算 n 个数的和，考虑两种算法设计思路：a. 逐个累加的平凡算法（链式）。b. 超标量优化算法（指令级并行），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。

1 实验环境

- CPU: AMD Ryzen 7 8845H w/ Radeon 780M Graphics
- 指令集架构: x86 架构
- GPU: NVIDIA GeForce RTX 4060 Laptop GPU GDDR6 @ 8GB (128 bits)
- 内存容量: 24GB
- 操作系统及版本: Windows 11 家庭中文版
- 编译器: GNU C++ Compiler
- 编译环境: MinGW-w64

2 实验一：n*n 矩阵向量内积

2.1 算法介绍

2.1.1 平凡算法

平凡算法是按列计算结果 sum 的每个元素。对于结果向量 sum 中的每一个位置 i：它会遍历矩阵 b 的第 i 列（即 b[0][i], b[1][i], b[2][i], ...），将该列的每个元素 b[j][i] 与向量 a 的对应元素 a[j] 相乘，然后把所有这些乘积加起来，得到 sum[i] 的最终值。简单说就是：逐个计算 sum 的元素，每个元素的计算需要遍历 b 的一整列和整个 a 向量。

对于较小的 n，单次算法执行耗时极短，可能低于计时器能够精确分辨的最小时间单位。为了获得更有意义的测量结果，我们将关键计算重复执行 times 次，累积一个更长、更易精确测量的时间段，然后通过除以 times 来估算单次执行的平均耗时，从而提高计时的精度。

伪代码

Algorithm 1 平凡矩阵向量乘法 (按列计算)

Input: n x n 矩阵 B, n 维向量 a, 整数 n (维度)

Output: n 维结果向量 sum

1: FOR i FROM 0 TO n-1:
2: $sum[i] \leftarrow 0$
3: FOR j FROM 0 TO n-1:
4: b_ji $\leftarrow B[j][i]$
5: a_j $\leftarrow a[j]$

6: $sum[i] \leftarrow sum[i] + \text{b\_ji} * \text{a\_j}$
7: **return** sum

逐列访问平凡算法

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

2.1.2 cache 优化算法

cache 优化算法通过交换内外两层循环，改变了访问矩阵 b 的顺序。原来是按列访问（内存不连续），优化后变成按行访问（内存地址连续）。这样可以更好地利用 CPU 缓存：当加载 b 的一个元素时，它附近（同一行）的其他元素很可能也被一起加载进缓存了，内层循环马上就能用到它们，减少了从慢速主内存读取数据的次数，从而提高了计算速度。

伪代码

Algorithm 2 Cache 优化的矩阵向量乘法

Input: n x n 矩阵 B, n 维向量 a, 整数 n (维度)
Output: n 维结果向量 sum

1: FOR i FROM 0 TO n-1:
2: $sum[i] \leftarrow 0$
3: FOR j FROM 0 TO n-1:
4: $\text{a\_j} \leftarrow a[j]$
5: FOR i FROM 0 TO n-1:
6: $\text{b\_ji} \leftarrow B[j][i]$
7: $sum[i] \leftarrow sum[i] + \text{b\_ji} * \text{a\_j}$
8: **return** sum

cache 优化算法

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

2.2 算法性能比较和分析

通过测量算法核心步骤的运行时间，来比较平凡算法和 cache 优化算法的性能，并且多次测量求平均值以减小误差，运行时间越短说明算法性能越好。

表 1: 不同问题规模下平凡算法与优化算法的运行时间对比

问题规模 n	平凡算法用时/ms	优化算法用时/ms
500	1.15661	1.15675
1000	5.42023	4.08070
1500	15.5648	10.0797
2000	29.8611	16.3725
2500	45.7855	27.9064
3000	83.8521	37.2222
3500	111.392	51.0185
4000	147.805	66.4926
4500	109.878	89.2187
5000	145.403	110.181
5500	157.182	124.28 0
6000	206.324	148.755
6500	236.582	177.222
7000	292.233	214.963
7500	301.654	246.810
8000	373.010	279.846
8500	410.785	317.007
9000	477.374	340.634
9500	492.875	378.490
10000	599.338	414.595

以问题规模 n 为横坐标，用时/ms 为纵坐标绘制图像如2.1所示。

通过折线图，我们可以更加直观的看出两种算法运行的时间差异，以及随着问题规模的变化，从两张图我们可以看出，随着问题规模 n 增大，两种算法运行时间差异越来越大。

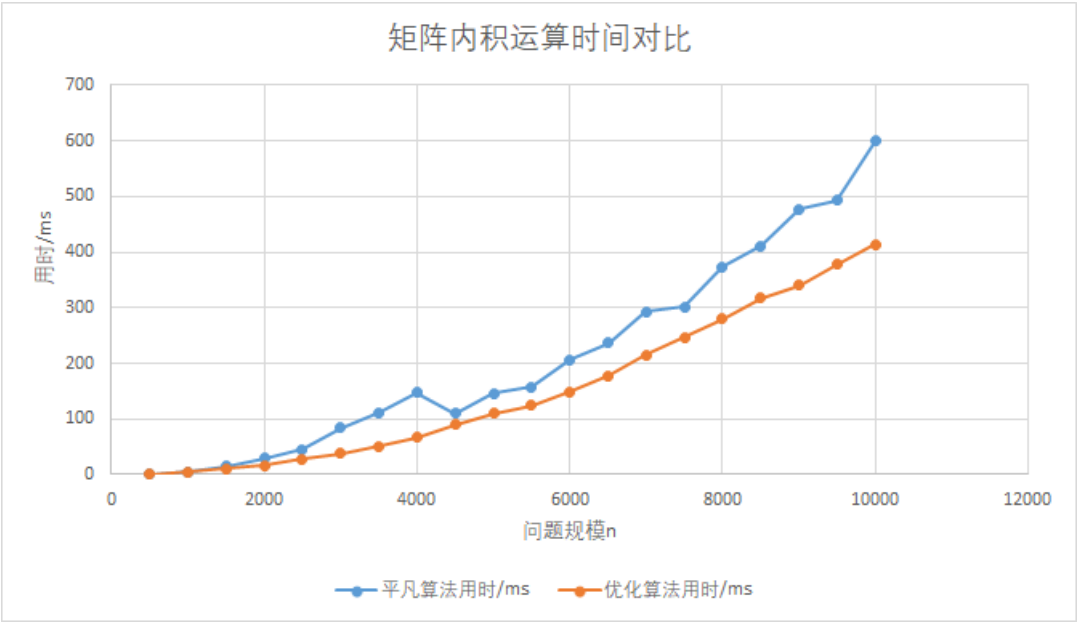

图 2.1: 矩阵内积运算时间对比

2.3 汇编代码深度分析

平凡算法和 cache 优化算法的汇编代码如下：

平凡算法的汇编代码

```
mov     eax, DWORD PTR [rbp-48]
cdqe
lea     rdx, [0+rax*8]
mov     rax, QWORD PTR [rbp-56]
add     rax, rdx
mov     rax, QWORD PTR [rax]
mov     edx, DWORD PTR [rbp-44]
movsx   rdx, edx
sal     rdx, 2
add     rax, rdx
mov     edx, DWORD PTR [rax]
mov     eax, DWORD PTR [rbp-48]
cdqe
lea     rsi, [0+rax*4]
mov     rax, QWORD PTR [rbp-64]
add     rax, rsi
mov     eax, DWORD PTR [rax]
mov     esi, DWORD PTR [rbp-44]
movsx   rsi, esi
sal     rsi, 2
mov     rdi, QWORD PTR [rbp-72]
add     rdi, rsi
mov     ecx, DWORD PTR [rdi]
imul    edx, eax
add     edx, ecx
mov     DWORD PTR [rdi], edx
add     DWORD PTR [rbp-48], 1
```

cache 优化算法的汇编代码

```
  mov     eax, DWORD PTR [rbp-48]
  cdqe
  lea     rdx, [0+rax*4]
  mov     rax, QWORD PTR [rbp-72]
  add     rax, rdx
  mov     ecx, DWORD PTR [rax]
  mov     eax, DWORD PTR [rbp-44]
  cdqe
  lea     rdx, [0+rax*8]
  mov     rax, QWORD PTR [rbp-56]
  add     rax, rdx
  mov     rax, QWORD PTR [rax]
  mov     edx, DWORD PTR [rbp-48]
  movsx   rdx, edx
```

```
15  sal     rdx, 2
16  add     rax, rdx
17  mov     edx, DWORD PTR [rax]
18  mov     eax, DWORD PTR [rbp-44]
19  cdqe
20  lea     rsi, [0+rax*4]
21  mov     rax, QWORD PTR [rbp-64]
22  add     rax, rsi
23  mov     eax, DWORD PTR [rax]
24  imul    edx, eax
25  mov     eax, DWORD PTR [rbp-48]
26  cdqe
27  lea     rsi, [0+rax*4]
28  mov     rax, QWORD PTR [rbp-72]
29  add     rax, rsi
30  add     edx, ecx
31  mov     DWORD PTR [rax], edx
32  add     DWORD PTR [rbp-48], 1
```

对比两种算法的汇编代码核心部分（特别是访问 b[j][i] 的指令序列），Cache 优化算法（内循环变量为 i）在计算 b[j][i] 的地址时，使用了由外层循环 j 决定的、在内循环中不变的行基地址（通过 [rbp-44] -> lea rdx, [0+rax*8] -> ... -> mov rax, QWORD PTR [rax] 计算得到），并加上了随内层 i 线性增长的偏移量（通过 [rbp-48] -> movsx rdx, edx -> sal rdx, 2 计算得到）。这种基地址 + 线性偏移的方式导致对 b 的内存访问是连续的（按行访问），从而能够高效利用 CPU 缓存行，一次内存读取就能将后续需要的多个元素载入缓存，提高缓存命中率。相反，平凡算法（内循环变量为 j）在内循环中不断改变 j 来计算不同的行基地址，导致对 b 的内存访问是跳跃的（按列访问），无法有效利用缓存行，导致频繁的缓存未命中和从主内存加载数据，因此效率较低。

3 实验二：n 个数求和

3.1 算法介绍

3.1.1 平凡算法

按顺序遍历向量 a 中的每一个元素，并将每个元素的值依次累加到同一个总和变量 sum 上。为了更好测量用时，故从 n = 114514 开始。

伪代码

Algorithm 1 顺序累加求和 (平凡算法)

Input: n 维整数向量 a, 整数 n (维度)

Output: 向量 a 中所有元素的总和result_sum

1: result_sum ← 0
2: FOR i FROM 0 TO n-1:
3: result_sum ← result_sum + $a[i]$
4: **return** result_sum

n 个数求和平凡算法

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

3.1.2 两路链式算法

两路链式算法使用两个独立的累加变量，在循环中同时累加数组中交替位置（如一个累加偶数位，一个累加奇数位）的元素，以打破单变量累加的依赖性，允许 CPU 更高效地执行加法，最后再将两个累加变量的结果相加得到总和。

伪代码

Algorithm 2 两路链式累加求和 (优化算法)

Input: n 维整数向量 a, 整数 n (维度)

Output: 向量 a 中所有元素的总和result_sum

1: $sum0 \leftarrow 0$
2: $sum1 \leftarrow 0$
3: $i \leftarrow 0$
4: FOR i FROM 0 UP TO n-2, INCREMENT BY 2:
5: $sum[0] \leftarrow sum0 + a[i]$
6: $sum[1] \leftarrow sum1 + a[i+1]$
7: IF i < n :
8: $sum[0] \leftarrow sum0 + a[i]$
9: result_sum $\leftarrow sum0 + sum1$
10: **return** result_sum

n 个数求和两路链式优化算法

```
long long sum0 = 0; // 偶数索引
long long sum1 = 0; // 奇数索引
int i = 0;
for (; i < n - 1; i += 2){
    sum0 += a[i];
    sum1 += a[i + 1];
}
if (i < n)sum0 += a[i];
sum += sum0 + sum1;
```

3.2 算法性能比较和分析

通过测量算法核心步骤的运行时间，并且多次测量求平均值以减小误差，运行时间越短说明算法性能越好。

表 2: 不同问题规模下平凡算法与优化算法的运行时间对比

问题规模 n	平凡算法用时/ms	优化算法用时/ms
114514	0.149644	0.128832
229028	0.303905	0.258730
458056	0.61546	0.523719
916112	1.17248	1.04957
1832224	2.33806	2.07843
3664448	4.66196	4.15505
7328896	9.38218	8.31618
14657792	18.4746	16.4175
29315584	37.4669	32.5765
58631168	74.4822	64.7785

以问题规模 n 为横坐标，用时/ms 为纵坐标绘制图像如3.2所示。

通过折线图我们可以看到优化算法用时低于平凡算法，并且随着问题规模的增大优化效果更明显。

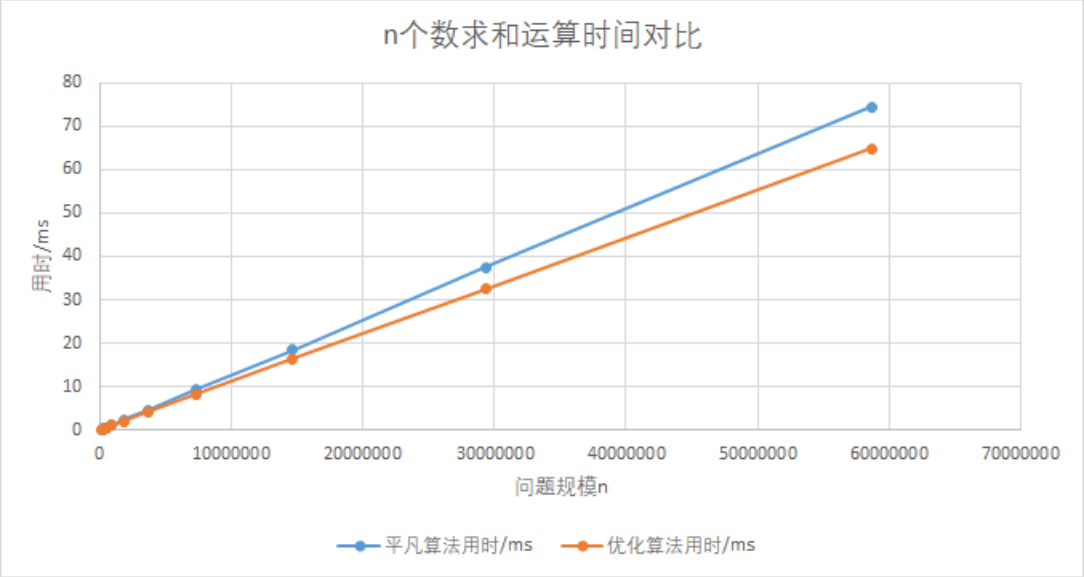

图 3.2: n 个数求和运算时间对比

3.3 汇编代码深度分析

平凡算法和两路链式优化算法的汇编代码如下：

平凡算法的汇编代码

```
mov     eax, DWORD PTR [rbp-4]
cdqe
lea     rdx, [0+rax*4]
mov     rcx, QWORD PTR [rbp-16]
add     rcx, rdx
```

```
    mov     edx, DWORD PTR [rcx]
    movsx   rdx, edx
    add     rbx, rdx
    add     DWORD PTR [rbp-4], 1
```

两路链式优化算法的汇编代码

```
mov     eax, DWORD PTR [rbp-4]
cdqe
lea     rdx, [0+rax*4]
mov     rcx, QWORD PTR [rbp-16]
add     rcx, rdx
mov     edx, DWORD PTR [rcx]
movsx   rsi, edx
add     rbx, rsi
lea     rdi, [rcx+4]
mov     edx, DWORD PTR [rdi]
movsx   rsi, edx
add     r12, rsi
add     DWORD PTR [rbp-4], 2
```

优化算法的汇编通过使用两个不同的寄存器（如 rbx 和 r12）作为独立的累加器，分别执行对 a[i] (add rbx, rsi) 和 a[i+1] (add r12, rsi) 的加法，打破了平凡算法中下一轮加法必须等待上一轮结果更新到单一累加器（如 rbx) 的依赖链，这使得 CPU 能够利用其指令级并行能力（如流水线或多个加法单元）同时或交错处理这两个独立的加法操作，从而提高了计算吞吐量。

GitHub 链接 Github