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：冯佳荟

学号：2310424

专业：计算机科学与技术

2025 年 3 月 30 日

目录

1 实验内容

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：
   (a) 逐列访问元素的平凡算法。
   (b) cache 优化算法。
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。

2 实验环境

硬件名称	对应参数
CPU 名称	13th Gen Intel Core i5-13500H
CPU 核心数	12
线程数	16
L1 高速缓存	1.1MB
L2 高速缓存	9.0MB
L3 高速缓存	18.0MB
基准频率	2.60GHZ

表 1: 硬件配置情况

代码开源于：https://github.com/feng05-2/parallel-programe

3 基础要求

3.1 算法设计

3.1.1 n*n 矩阵与向量内积

将 n*n 矩阵放在二维数组 A[n][n] 中，给定的向量存于一维数组 a[n] 中，并声明数组 s[n] 用于存放求取内积后得到的向量，并将 s[n] 初始化为全 0 数组。

- 逐列访问矩阵元素的平凡算法

  基于按列访问矩阵元素的原则，采用双重 for 循环，外循环为列 j，内循环为行 i，每步内循环计算出来 s[j] 的部分积在内层累加，一步外循环完成后即得到一个内积 s[j]。

- 逐行访问矩阵元素的 cache 优化算法

  基于按行访问矩阵元素的原则，同样采用双重 for 循环，此时外循环为行 i, 内循环为列 j, 一步外循环结束后得到 n 个内积的部分积，将其赋值给 s[j],n 次外循环结束后，s[j] 的部分积累加完成得到内积 s[j]。

  由于矩阵按行存储，cache 将数据读入缓存时，会基于空间局部性原理会预取与需要读取的数据相邻的元素，即每次会缓存同一行的数据，所以采用逐行遍历可以提高 cache 的缓存命中率，减少从内存中再次读取数据到缓存的次数，实现对算法的优化。

3.1.2 n 个数求和

由于测试数据都是人为生成，我们在测试时将元素个数都取为 2 的幂，由此简化算法设计。

- 逐个累加的平凡算法

  将 n 个数存入数组 a[n] 中，定义并初始化整型变量 sum=0, 每读入一次 a[i]，令 sum=sum+a[i]，n 次读入后，即可完成求和得到 sum。

- 两路链式累加

  将 n 个数存入数组 a[n] 中，然后将奇数项和偶数项分别累加得到 sum1 和 sum2，再将 sum1 和 sum2 相加得到 sum。

- 递归算法

  将 n 个数存入数组 a[n] 中后，从 a[0] 开始，将数组中的数据两两求和（相邻两项），再将得到的结果相邻两项两两求和，如此进行下去，直到只有一个结果即为 n 个数的和。

3.2 编程实现

3.2.1 n*n 矩阵与向量内积

逐列访问平凡算法

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

逐行访问 cache 优化算法

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

3.2.2 n 个数求和

逐个累加的平凡算法

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

```
    cin>>a[i];
    sum=sum+a[i];
    }
```

两路链式累加

```
int sum1=0,sum2=0;
for(int i=0;i<n/2;i++){
    sum1=sum1+a[2*i];
    sum2=sum2+a[2*i+1];
    }
sum=sum1+sum2;
```

递归算法

```
void sum(int* a, int n) {
    if (n==1) return;
    for (int i=0;i<n/2;i++) {
        a[i]=a[2*i]+a[2*i+1];
    }
    n=(n+1)/2;
    sum(a,n);
}
```

3.3 性能测试

利用 window.h 中的 QueryPerformance() 测试算法执行时间来进行性能测试。

3.3.1 n*n 矩阵与向量内积

- 程序的性能与 cache 相关，而现代 CPU cache 有多个层次，每个层次有固定的规模，因此在性能测试中设置一系列的问题规模对应设备各级 CPU 大小，选取 n=0-5000 递增测试，以研究问题规模对程序性能产生的影响及变化趋势；
- 本问题计算较为简单，当矩阵规模较小时，程序运算时间较短，因此在测试时重复运行待测函数后求平均运行时间，解决计时函数精度不够的问题。

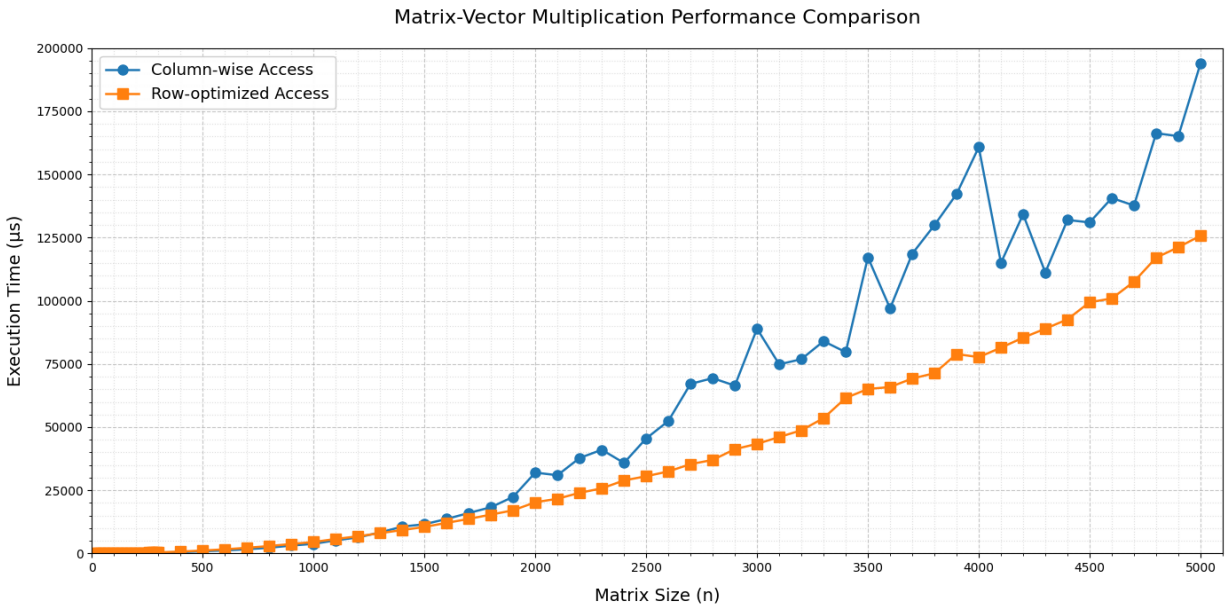

图 3.1: 两种算法在不同数据规模下的执行时间

3.3.2 n 个数求和

- 方便算法设计，测试数据都设置为 2 的幂；
- 同样设置不同的问题规模（n 从 2 到 2^{24}）测试程序性能;
- 每个算法核心重复计算 100 次提高测试精度。

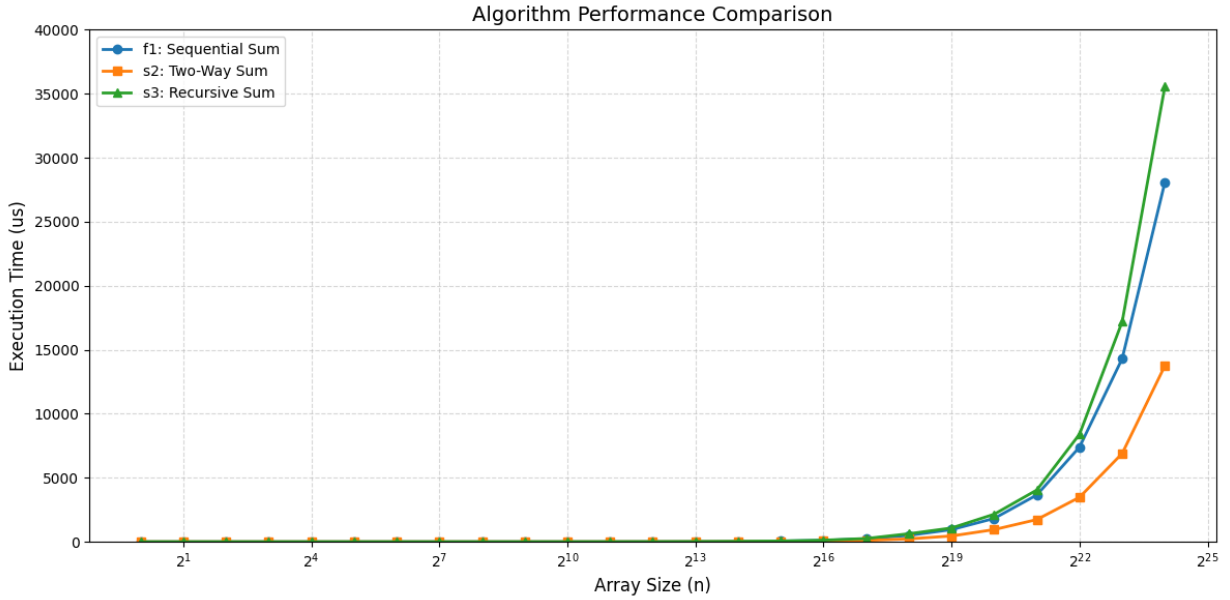

图 3.2: 三种算法在不同数据规模下的执行时间

3.4 结果分析

3.4.1 n*n 矩阵与向量内积

根据性能测试结果我们可以看出逐行访问的 cache 优化算法在大部分测试范围内都显著优于逐列访问的平凡算法，主要原因在于：

- 连续内存访问模式提高缓存行利用率：平凡算法按列访问元素，每次访问的内存地址不连续，如果 n 较大，即行的长度过大，则每次访问跳跃地址过大可能会触发新的缓存行加载，每次加载一次缓存行可能只利用第一个元素导致利用率极低；而 cache 优化算法访问的内存地址是连续的，每次加载的缓存行可包含后续要使用的多个元素，利用率接近 100
- 顺序访问更容易被预期器预测：现代 CPU 的硬件预取器会预测程序的内存访问模式来提前加载可能需要的数据到缓存中，逐列访问是非连续模式难以预测，而逐行访问容易被识别，可减少内存访问延迟；
- 内存带宽高效利用：当矩阵规模超过某一级缓存的容量时，性能差异会被进一步放大。正如图中的波动：当 n<100 时，数据完全载入 L1 缓存，两种算法的性能接近，可当 n>200 时，数据规模逐步溢出 L1/L2 缓存，平凡算法每次列访问都需要从更高层缓存或主存中加载数据，导致性能急剧下降；而对于 cache 优化算法，即使超出 L3 缓存，由于连续的访问模式，仍旧可以通过缓存的批量加载数据等来优化性能。

3.4.2 n 个数求和

从测试结果可以看出，递归算法并没有实现优化，执行时间反而显著高于其他两种算法，两路链式累加的优化效果较为显著，下面对 3 中算法的测试结果进行分析：

- 平凡算法：加法操作存在严重的数据依赖，CPU 无法并行执行多个操作；
- 递归算法：每次递归调用时保存寄存器状态、传递参数等会导致额外开销，同时递归调用导致栈内存频繁访问，与数据数组竞争空间，并且递归条件判断增加了分支预测错误率，这些性能劣势使得递归算法执行效率最低；
- 两路链式累加：拆分两个独立变量 sum1、sum2，消除了数据依赖，允许 CPU 并行执行两个加法操作，充分利用超标量流水线。

根据以上分析，递归算法在实际应用中应避免使用；在问题规模较小，如 n<=2^{16} 时，两路链式累加的优势显著；但随着问题规模 n 的增大，这种差距逐渐缩小，原因在于：在大规模数据下，内存带宽成为瓶颈，CPU 无法充分掩盖访存延迟，此时平凡算法则成为更稳定的选择。

4 进阶要求

4.1 unroll 优化算法

4.1.1 n*n 矩阵与定向量的内积

循环展开侧重于将多次循环迭代合并为一次来降低循环控制开销。

基础的 cache 优化算法尽管改善了逐列访问造成的缓存命中率低的问题，但是循环的开销还是很大，我们通过将多次循环合并为一次对算法进行优化，代码修改如下：

循环展开后的 cache 算法

```
for (int i = 0; i < n; i++) {
    int sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0;
    int j = 0;
    for (; j <= n - 4; j += 4) {
        sum0 += A[i*n + j]   * a[j];
        sum1 += A[i*n + j+1] * a[j+1];
        sum2 += A[i*n + j+2] * a[j+2];
        sum3 += A[i*n + j+3] * a[j+3];
    }
    int sum = sum0 + sum1 + sum2 + sum3;
    for (; j < n; j++) {
        sum += A[i*n + j] * a[j];
    }
    s[i] = sum;
}
```

测试 unrolling 后的代码执行时间并与原算法进行对比：

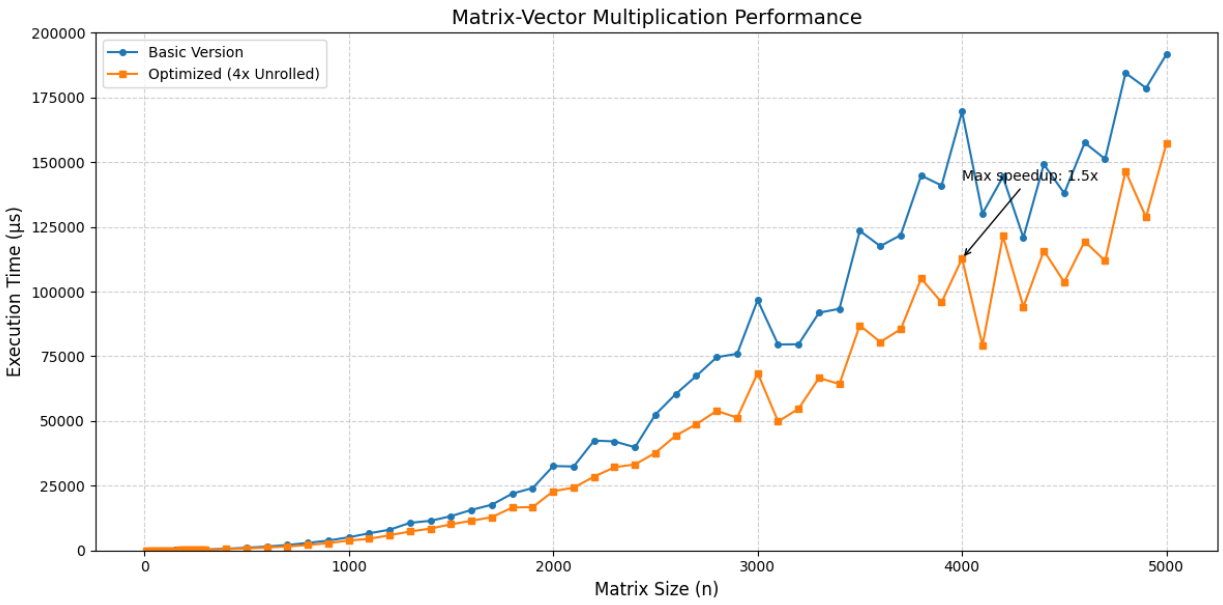

图 4.3: cache 算法 unroll 前后执行时间随问题规模的变化

循环展开后的优化版本带来了持续的性能提升，执行时间曲线全程位于基础 cache 算法曲线的下方，在 n=3000 左右时优化效果最为明显，成功实现优化主要依赖于以下几个方面：

- 现代 CPU 的分支预测如果发生错误会导致大量的流水线清空，而 4 路展开将分支分支大大减少；
- 使用了独立的累加变量，消除了原有程序对数据的依赖，且如本设备的 CPU，拥有 4-6 个整数 ALU 端口，4 路展开能够更好地利用多发射能力；
- 每次迭代加载的 4 个元素有很大概率位于同一缓存行，减少了缓存缺失。

4.1.2 n 个数的和

对 n 个数求和的平凡算法进行循环展开后的代码实现如下：

循环展开后的平凡算法

```
    int sum=0;
    int i;
    for(i = 0; i < n/4; i++) {
        sum= sum + a[4*i];
        sum= sum + a[4*i+1];
        sum= sum + a[4*i+2];
        sum= sum + a[4*i+3];
    }
    for(int j=4*i;j<n;j++){
      sum=sum+a[j];
  }
  return sum;
```

同样利用 window.h 中的 QueryPerformance() 测试算法执行时间来进行性能测试，具体测试方法与基础要求中的测试方法一致。

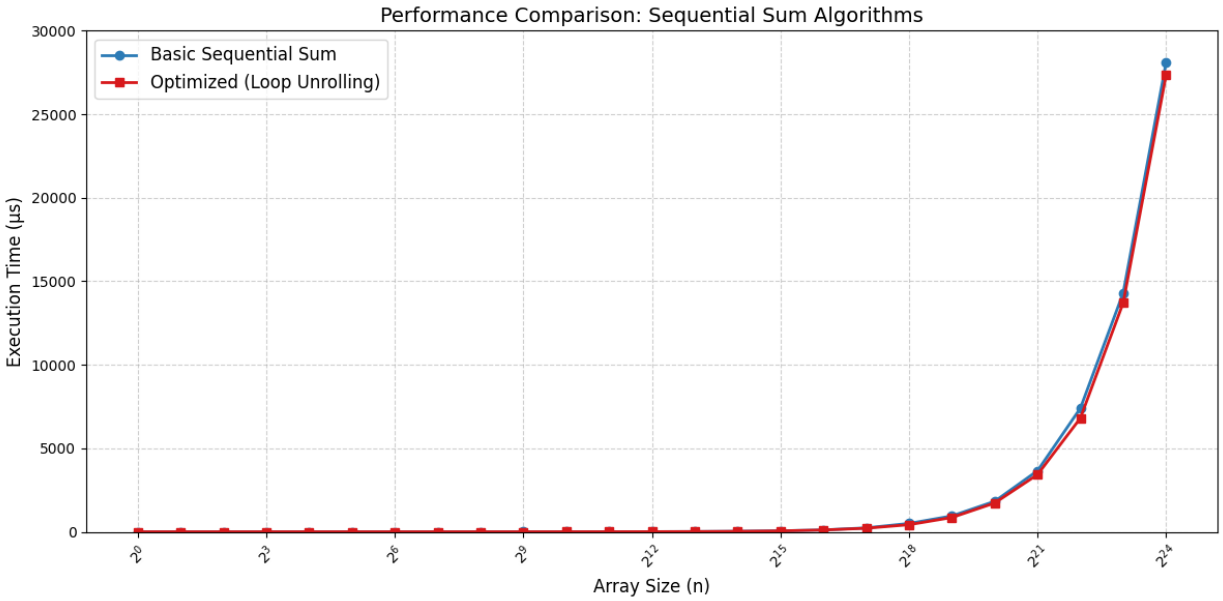

图 4.4: unrolling 前后的平凡算法执行时间

由测试结果可知，单纯的循环展开并未带来显著的性能提升，原因在于：

- 现代 CPU 的指令级并行使手动循环展开的边际效益降低，当 $n>2^{20}$ 时，性能还会受到内存带宽的限制；
- 循环展开的计算强度过低，每个元素只做一次加法，无法充分利用循环展开减少的指令开销，并且单纯的减少迭代次数而未采用多变量，仍严重依赖数据，限制了指令级并行。