南開大學
Nankai University

计算机学院

并行程序设计第一次试验报告

CPU 架构相关编程

姓名：刘迪乘

学号：2312810

专业：计算机科学与技术

2025 年 3 月 28 日

目录

1 实验内容 **2**

2 实验环境 **2**

3 基础要求 **2**

3.1 算法设计 . . . 2

3.1.1 n*n 矩阵与向量内积 . . . 2

3.1.2 n 个数求和 . . . 2

3.2 编程实现 . . . 3

3.2.1 n*n 矩阵与向量内积 . . . 3

3.2.2 n 个数求和 . . . 3

3.3 性能测试 . . . 4

3.3.1 n*n 矩阵与向量内积 . . . 4

3.3.2 n 个数求和 . . . 5

3.4 结果分析 . . . 5

3.4.1 n*n 矩阵与向量内积 . . . 5

3.4.2 n 个数求和 . . . 5

4 进阶要求 **6**

4.1 利用 perf 工具进行更加细致的 profiling . . . 6

4.1.1 n*n 矩阵与向量内积 . . . 6

4.1.2 n 个数求和 . . . 6

4.2 不同编译器优化力度对实验结果的影响 . . . 7

4.2.1 n*n 矩阵与向量内积 . . . 7

4.2.2 n 个数求和 . . . 8

4.3 unroll 优化 . . . 8

1 实验内容

- 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:
  1. 逐列访问元素的平凡算法
  2. cache 优化算法
- 计算 n 个数的和，考虑两种算法设计思路:
  1. 逐个累加的平凡思路（链式）
  2. 超标量优化算法 (指令级并行)，如最简单的两路链式累加；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

本次实验选取的操作系统为 Windows11 下的 WSL（Linux 的版本为 24.04)。
硬件参数如下:

硬件名称 \ 参数	naive-conv
CPU 型号	AMD R7 9700X
CPU 核心线程数	8 核 16 线程
主频	5GHZ
内存容量	16*2GB
L1Cache	640kb
L2Cache	8MB
L3Cache	32MB

表 1: 涉及硬件参数

编译器版本: gcc version 13.3.0 (Ubuntu 13.3.0-6ubuntu2 24.04)
实验代码开源于: https://github.com/July-h5kf3/NKU-parallel-computing

3 基础要求

3.1 算法设计

3.1.1 n*n 矩阵与向量内积

对于逐列访问的平凡算法我们直接采取双层循环，外层枚举列，内层循环得到每一列得到的结果。

对于 cache 优化算法，同样采取双层循环，外层枚举行，内层循环仅计算出仅向每个内积累加一个乘法结果。这种计算方式的访存模式与行主存储匹配，具有很好空间局限性，令 cache 作用得以发挥。

3.1.2 n 个数求和

对于逐个累加的平凡思路直接循环链式求和即可
优化有两种方法:

1. 多链路式，例如将奇数项偶数项分别求和最后求和得到结果
2. 递归式，将给定数据两两相加，得到 $\frac{n}{2}$ 个中间结果，再将上一步得到的中间结果两两相加，得到 $\frac{n}{4}$ 个中间结果。依次类推，$\log n$ 个步骤后得到最后一个值即为最终结果。

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++)
{
    sum[i] = 0.0;
    for(int j = 0;j < n;j++)
    {
        sum[i] += (b[j][i] * a[j])
    }
}
```

Cache 优化算法

```
for(int i = 0;i < n;i++)
    sum[i] = 0.0;
for(int i = 0;i < n;i++)
{
    for(int j = 0;j < n;j++)
    {
        sum[j] += b[i][j] * a[i];
    }
}
```

3.2.2 n 个数求和

逐个累加平凡算法

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

多链路式优化算法

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

递归优化算法

```
function recursion(n)
{
    if(n == 1)return;
    else
    {
        for(int i = 0;i < n/2;i++)
        {
            a[i] += a[n-i-1];
        }
        n = n/2;
        recursion(n);
    }
}
```

3.3 性能测试

3.3.1 n*n 矩阵与向量内积

对于数据生成，由于程序性能与 cache 相关，而实验采用的硬件 cache 具有多个层次，每个层次的 cache 大小有固定规模，因此我们设置了不同的问题规模来测试 cache 优化对程序的性能的提升。

具体而言，我设置 n 从 10 到 5000 递增，以求将三个层级的 cache 完全覆盖。其中 n 在 0-300 的数据范围内以 10 递增，而大于 300 时则以 100 递增。且由于数据较小的时候程序完成较快，因此采用重复实验取平均的方式计算时间。

得到程序的运行时间对比图如下：

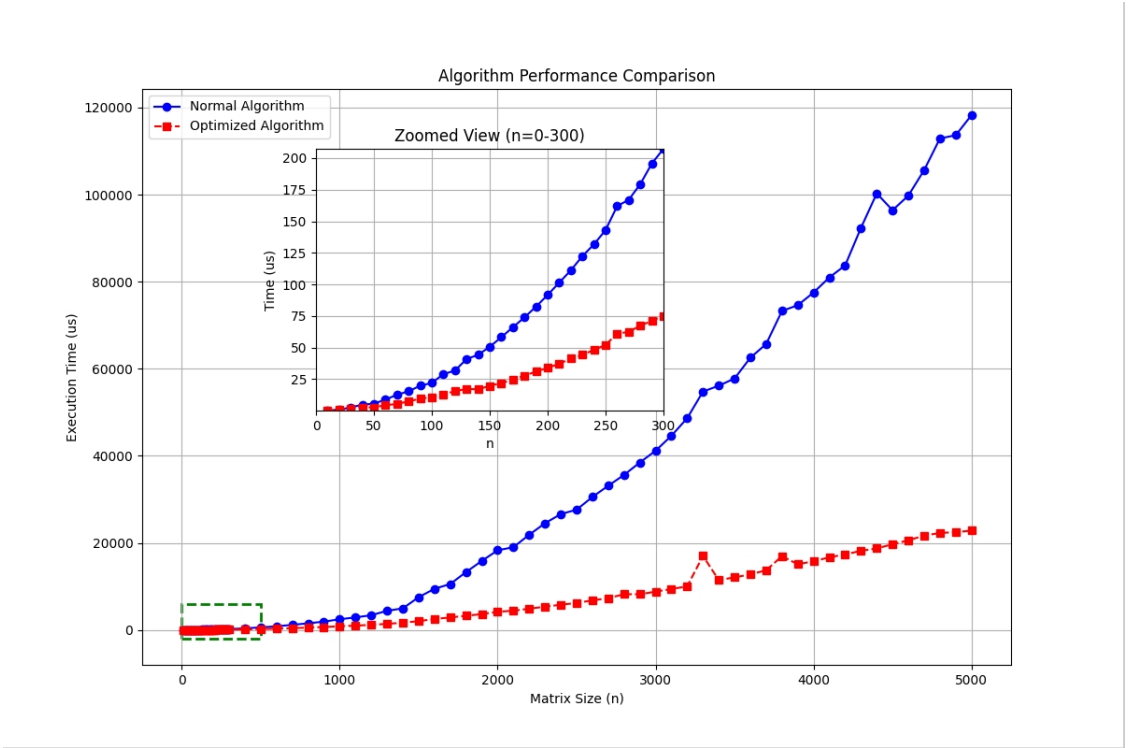

图 3.1: Matrix_product 的执行时间随问题规模的变化

3.3.2 n 个数求和

同样设置了不同的问题规模来测试不同算法的性能，其中多链路式的算法分别采取了 2，4，8 链。n 从 2^3 到 2^{30}，每次 n 都乘以 2，保证 n 时 2 的整数幂。对于每个 n，均重复 100 次取平均以得到更加准确的时间结果。

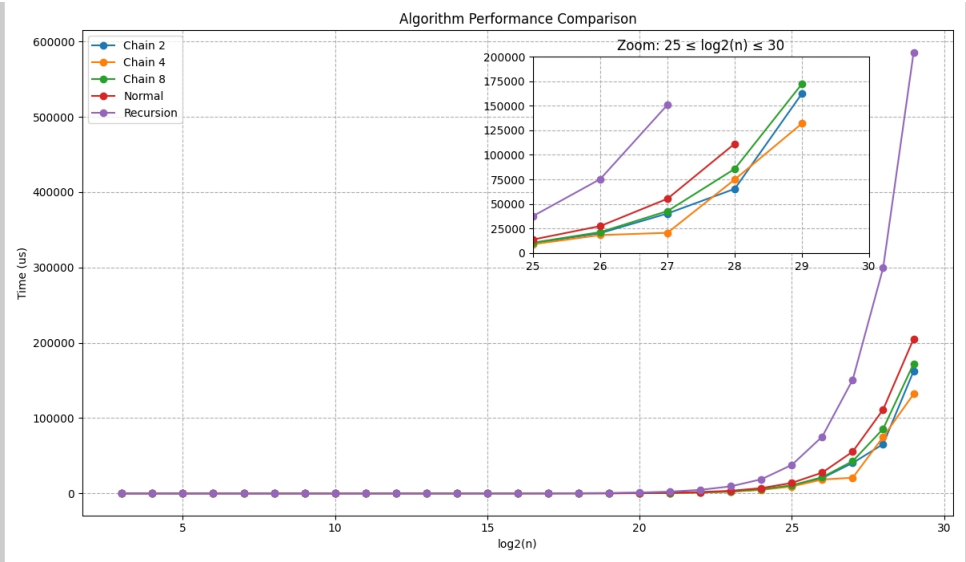

图 3.2: 各个算法在不同数据规模下的执行时间

3.4 结果分析

3.4.1 n*n 矩阵与向量内积

从运行计时的结果图来看，随着数据规模的不断增大，两种算法执行时间的差异也越来越大，这是因为优化的行主访问模式相较于平凡方法的列主访问方式，更加与行主存储匹配，性能更好。

其中我发现在某些 n 值附近出现了凸起 (n = 280,n = 3200 等)，这可能是因为数据存储的大小恰好处于各级 cache 的边缘，触发了缓存临界效应，使得缓存颠簸加剧，CPU 会频繁替换缓存行 (Cache line)，导致刚加载的数据被立即逐出，大大降低了 Cache 的命中率（这一点会在进阶要求中详细展示数据）。

3.4.2 n 个数求和

从运行结果图 3.2 可以看出来，循环 unroll 的方式对程序性能的提升更大，而递归的方式甚至不如平凡的方法。

这主要是因为，函数调用时的堆栈开销以及递归的不断跳转操作使得产生大量的不连续内存的访问，从而使得指令的缓存命中率降低。为了佐证这一点，在后续进阶要求中也做了相关的 cache 命中率的测试。

且反直觉的是，在一定数据范围内，并非 unroll 的链数越多，程序的性能越好。

这是因为，一方面，每次循环展开会增加生成的指令数量。如果展开层数过多，循环体的代码量可能超出 CPU 的 L1Cache 中的指令缓存的容量，导致缓存的频繁换入换出；另一方面循环展开会增加并行使用的寄存器数量。若超出了 CPU 的物理寄存器数量，编译器就需要额外插入寄存器溢出代

码，从而增加开销。在进阶要求部分，我将通过 pref 工具测试不同展开层数的 L1 未命中率以及 CPI 来展示不同展开层数的效率。

4 进阶要求

4.1 利用 perf 工具进行更加细致的 profiling

4.1.1 n*n 矩阵与向量内积

这一部分我一方面对平凡算法以及 cache 优化后的算法的 cache 命中率进行了测试，从而说明其优化的内层原理。

algo\events	L1-dcache-loads	L1-dcache-load-misses
normal_algo	41.13%	73.86%
optimizer_algo	36.90%	16.40%
Events_total	16965297791	682422781

表 2: Cache 事件统计

由此可见，Cache 优化算法的 L1-dcache-load-misses 事件显著少于 normal 算法，印证了我们对采取行主访问模式可以提升缓存的命中率从而提升程序的运行性能的解释。

另一方面，对 cache 优化算法选取了 n 从 3000 到 4000 以步长为 50 对 cache 命中率，以及 IPC 进行测试从而探究突变出现的原因。结果如下：

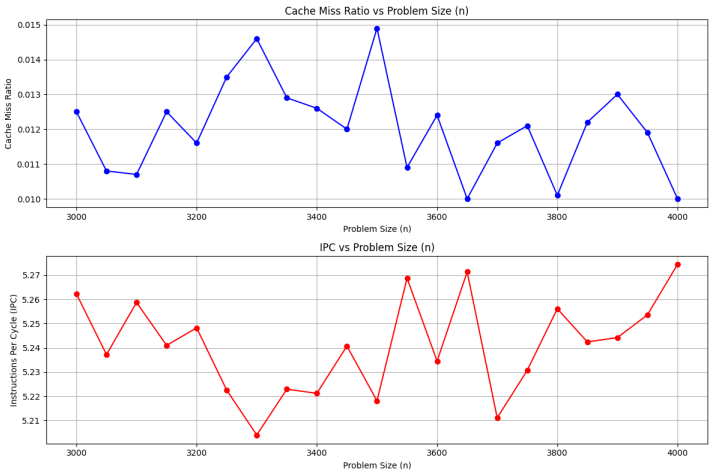

图 4.3: Cache 命中率以及 IPC

显而易见的是，随着 n 的增大，Cache Miss Ratio 以及 IPC 在 3200-3400 这个区间内出现了明显异常，而通过计算可知这个区间恰好也处于 L2-L3Cache 的临界处，发生了缓存临界效应，从而使得缓存颠簸加剧，降低 Cache 命中率，IPC，使得程序性能下降。

4.1.2 n 个数求和

这一部分，我直接测试了各个算法在 $n = 2^{25}$ 下，各个事件的数量，见下表：

algo\events	L1-dcache-loads	L1-dcache-load-misses	Miss-ratio	Cycles	Instructions	CPI
noraml	749736468	12932278	0.0172	349367284	1284942563	0.2718
chain_add_2	647453986	12878494	0.0198	311800696	1184052383	0.2633
chain_add_4	663803891	12855609	0.0193	326057553	1267185415	0.2573
chain_add_8	638637826	12852000	0.0201	321275132	1241902484	0.2586
chain_add_16	627975258	12917017	0.0205	310289612	1229903186	0.2522
chain_add_32	620101501	12855822	0.0207	303840732	1223189865	0.2484
chain_add_512	616210070	12950184	0.0210	335475485	1218416257	0.2753
recursion	1016535105	17059165	0.0167	489059835	2358122288	0.2073

表 3: 各个事件在 n=2^{25} 下的数目

可以发现，各个优化算法在 CPI 指标上较平凡算法有了显著提升，说明我们采取的优化充分利用了 CPU 的流水线设计，得到了一定的收益。

而递归算法的各个事件数均显著高于其它算法，这是它慢于其它的算法的主要原因，不断地调用跳转使得其指令数较其它算法更多，使得程序的性能下降，甚至不如平凡算法。

对于提高 unroll 的层数但是性能提升不大甚至更劣，可以看见链路从 2 到 4 再到 8，CPI 有一个先减少后增加的趋势 (到 512 链路时甚至为 0.2753 劣于平凡算法)，这是由于随着链路的增加，寄存器数目不足从而发生冲突，降低了程序性能；从 Miss-ratio 同样也可以看出来同样存在先减后增加的特点，这体现了因为寄存器冲突造成的缓存利用率下降。

4.2 不同编译器优化力度对实验结果的影响

编译器提供了不同的编译优化选项来提升程序的性能，主要包括以下三种优化：

- -O1: 平衡优化和编译时间，删除部分未使用的代码，简化算术运算，合并相同表达式，性能大约提升 10-20%
- -O2: 更加激进的优化，包含循环优化、指令调度，函数内联小型函数，性能提升 20%-40%
- -O3: 最为激进的优化包含向量化等高级优化，性能可能进一步提升 5%-15%

4.2.1 n*n 矩阵与向量内积

结果如下：

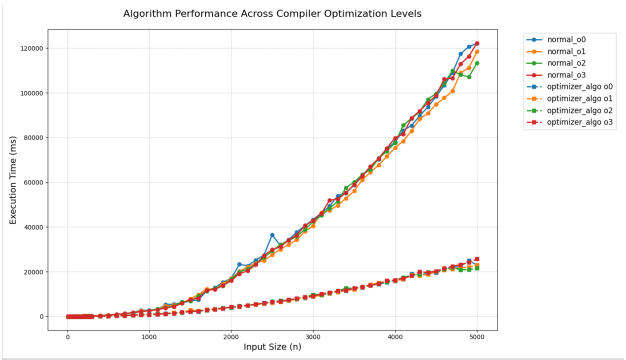

图 4.4: 编译器优化性能

发现相较于平凡算法，编译器优化对于 Cache 优化算法的提升较小，仅在数据较大时较为明显，甚至出现了有时候不如不优化的情况。也并非像直觉认为一样，o3 优化一定比 o2,o1 好，这与程序的数据范围以及算法均相关，但是总体而言提升不如算法变化带来的提升。

4.2.2 n 个数求和

由于这个实验中，平凡算法和优化算法的差别不是很大，因此仅以平凡算法作为 baseline，采取不同的编译器优化，得到如下结果

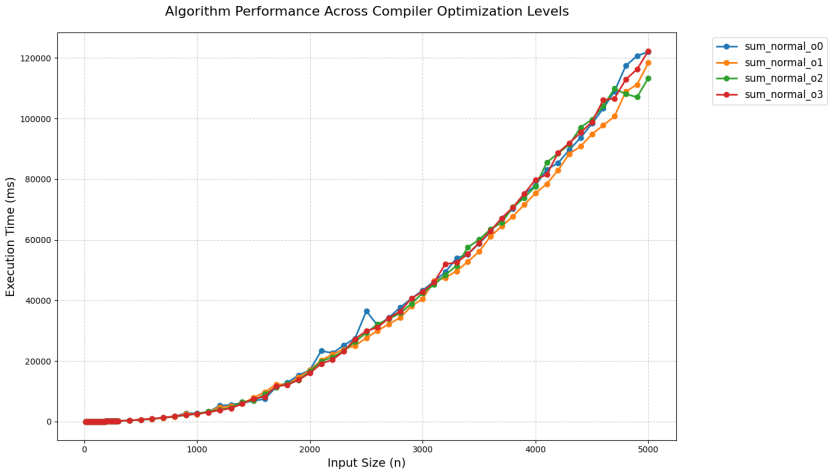

图 4.5: 编译器优化性能

可以发现，这样的优化之后程序的性能得到了较大的提升，在 o2 优化下，甚至比一些 unroll 算法的性能更加，这可能是因为编译器优化与 unroll 一样都采用了循环优化，且编译器层面的优化带来的冗余更小

4.3 unroll 优化

对 n*n 矩阵与向量内积做了 unroll 优化，代码如下:

递归优化算法

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

```
            sum[j] += a[i][j] * b[i];
        }
    }
    return;
}
```

测试程序性能，时间随任务规模图如下:

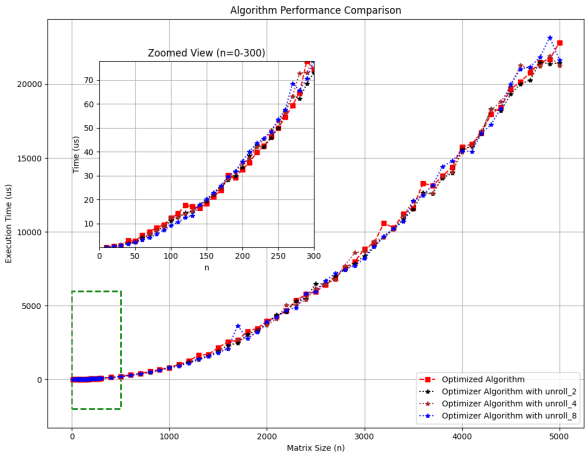

图 4.6: 性能图

实验表明，性能与一般的 Cache 优化（以四路为例）相比大相径庭，没有实质性的提升。

性能也没有随着链路的增加而得到显著提升，二者线性相关性不大。这可以用我们上面对 n 个数之和的多链路式提高层数但是性能提升不大的理论进行解释。

参考文献