
南開大學
Nankai University

计算机学院

并行程序设计实验报告

# CPU 架构编程实验

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 3 月 28 日

# 目录

# 1 实验环境

| CPU 型号 | i9-13900HX |
|---|---|
| CPU 主频 | 2.2GHz |
| L1 缓存 | 2.1MB |
| L2 缓存 | 32.0MB |
| L3 缓存 | 36.0MB |

表 1: 实验环境：x86 架构

# 2 问题一: 矩阵与向量内积

## 2.1 问题重述

给定一个 n × n 矩阵，计算每一列与给定向量的内积，考虑两种算法设计思路，进行实验对比：
1. 逐列访问元素的平凡算法；
2. cache 优化算法。

## 2.2 算法设计

从问题角度出发，现给定了一个 n*n 的矩阵和一个 n 维向量，要求计算每一列与给定向量的内积，自然而然想到的就是去遍历每一列去做相应运算，这就是平凡算法的实现逻辑。由于计算机内部的存储逻辑从上至下依次是寄存器、多级缓存、主存、外存等等，在计算比较大规模的数据时，计算机首先会把数据存储在缓存 cache 中，而 cache 的基本单位是缓存行，原理是**在 cache 中，每次访问内存可以读取一整块连续的数据**。而在计算机中，二维数组是行式存储的，这也就说明，如果按照平凡算法，去遍历每一列的话，会跨越很多数据单元，如果数据规模比较大时，一行数据不能全部存储在一个缓存行中，就不能利用 cache 一次读取一个缓存行的优势，访问次数会更多，也就造成了数据读取操作的时间负担，所以我们引入了 cache 优化算法。优化算法的原理是，遍历每一行与对应向量某一维元素的乘积，例如第一行每个元素都去乘向量的第一个元素，再做累加，这也能解决提出的问题，并且由于这种算法是行遍历的，一次内存访问能读入多个数据，数据读取速率会明显加快。

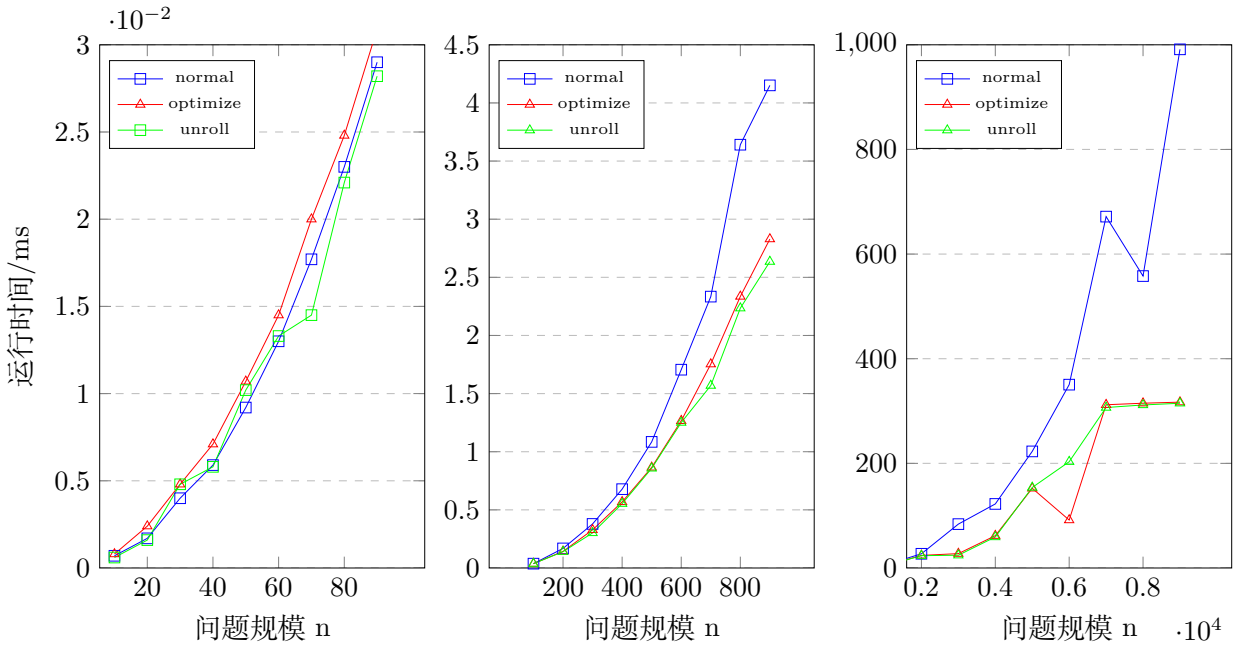
## 2.3 性能分析

### 2.3.1 数据分析

对表 2 中数据进行处理，画出对应的折线图，我们可以看出运行时间随问题规模增长的变化趋势。由折线图可以看出，起初在问题规模 n 比较小的时候，在 n<400 时，两种算法没有明显的差别；当问题规模达到 $10^2$ 时，n>400 时，开始出现差距，优化算法的效率开始高于平凡算法，但总体趋势大致相同；而当 n>3000 时，两种算法开始出现明显差距，优化算法的效率明显高于平凡算法，在 n>3000 时，cache 优化效果显著。随着问题规模越来越大，数据在缓存中的一次访存一个缓冲行的逻辑在我们处理算法时就占了更主要的因素，与列式访问相比，行式访问时由于访问时数据是连续的，故算法效率大大提高。对于 unroll 算法来说，其原理是把原来一路的运算，转换成了多路的计算，根据电脑的 CPU 硬件，我这里选用了五路展开处理，由数据可以看到，其优化效益也很显著，甚至微优于 cache 优化算法。

| n | normal | optimize | unroll | n | normal | optimize | unroll | n | normal | optimize | unroll |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 0.0007 | 0.0008 | 0.0006 | 100 | 0.0373 | 0.0376 | 0.0374 | 1000 | 5.3679 | 3.8158 | 3.7124 |
| 20 | 0.0017 | 0.0024 | 0.0016 | 200 | 0.1683 | 0.1426 | 0.1402 | 2000 | 27.1636 | 24.0919 | 23.5512 |
| 30 | 0.004 | 0.0048 | 0.0048 | 300 | 0.3788 | 0.3279 | 0.3017 | 3000 | 83.8861 | 27.4605 | 24.432 |
| 40 | 0.0059 | 0.0071 | 0.0058 | 400 | 0.6781 | 0.5683 | 0.5532 | 4000 | 122.419 | 61.724 | 59.234 |
| 50 | 0.0092 | 0.0107 | 0.0102 | 500 | 1.0846 | 0.8652 | 0.8571 | 5000 | 222.845 | 152.318 | 153.882 |
| 60 | 0.013 | 0.0145 | 0.0127 | 600 | 1.7056 | 1.266 | 1.251 | 6000 | 350.429 | 91.6682 | 203.11 |
| 70 | 0.0177 | 0.02 | 0.0145 | 700 | 2.3341 | 1.7534 | 1.569 | 7000 | 671.597 | 311.997 | 306.711 |
| 80 | 0.023 | 0.0248 | 0.0221 | 800 | 3.6407 | 2.3354 | 2.2347 | 8000 | 557.983 | 314.942 | 311.607 |
| 90 | 0.029 | 0.0309 | 0.0282 | 900 | 4.1511 | 2.8292 | 2.6352 | 9000 | 991.347 | 316.986 | 314.886 |

表 2: 矩阵与向量内积平凡与优化算法运行时间比较




结合计算机多级存储结构的各自特性以及缓存的机制，通过折线图我们可以推测出：在 n<400 时，数据规模很小，数据可能都存在 L1 cache 中，cache1 命中率为 100%，并且由于第一级缓存的速度很快，所以在问题规模较小的情况下基本上一致相同，也可能存在 L2 cache 中，但由于 L2 的速度较快，所以没多少区别；当数据规模再大一点时，由于数据会存到更低级的缓存中，前面的缓存命中率会下降，所以此时部分数据进入到 L3 cache，而第三级缓存的访问效率会比前两级慢得多，所以对于平凡算法来说，经常会到 L3 中访存数据，在 L3 中访存数据的次数很有可能是线性的，多于优化算法，所以效率会大大下降；当数据规模更大时，L3 命中率也会下降，数据就会被存入主存中，平凡算法需要访问更多次的主存，就导致了更差的效益。

### 2.3.2 cache 分析

我们可以利用 vtune 去分析我们两种算法在不同问题规模下各级 cache 命中情况来验证我们的猜测，我们选取了 n=100、400、1000、3000、8000 的问题规模去 vtune 进行监测，得到上述两表 (表 3)，分别是列式访问的平凡算法和行式访问的优化算法。

例如选取 n=3000 这个问题规模，对比 L1，平凡算法的访存次数更多，但由于 L1 速度很快，所以

| normal | | | | | | |
|---|---|---|---|---|---|---|
| | cache1_hit | cache1_miss | cache2_hit | cache2_miss | cache3_hit | cache3_miss |
| 100 | 4000012 | 0 | 400006 | 0 | 0 | 0 |
| 400 | 8000024 | 0 | 0 | 0 | 0 | 0 |
| 1000 | 38000114 | 1200018 | 800012 | 0 | 200042 | 0 |
| 3000 | 154000462 | 9600144 | 8400126 | 800168 | 200042 | 600252 |
| 8000 | 1186003558 | 65600984 | 62400936 | 4000840 | 0 | 4201764 |

| optimize | | | | | | |
|---|---|---|---|---|---|---|
| | cache1_hit | cache1_miss | cache2_hit | cache2_miss | cache3_hit | cache3_miss |
| 100 | 12000036 | 400006 | 0 | 0 | 0 | 0 |
| 400 | 14000042 | 0 | 0 | 0 | 0 | 0 |
| 1000 | 12000036 | 0 | 0 | 0 | 0 | 0 |
| 3000 | 186000558 | 800012 | 800012 | 200042 | 0 | 200084 |
| 8000 | 1246003738 | 7200108 | 6400096 | 200042 | 0 | 300126 |

表 3: 两种算法在不同规模下多级 cache_hit 分析

L1 并不是决定两种算法差异的主要因素，L2 速度比 L1 慢，并且平凡算法的访存次数很多，所以需要更多时间，L3 也是类似，并且我们可以看到两种算法 L3 级都有着很高的未命中率，说明需要到内存中去读取数据，平凡算法需要到内存读取 600252 量级的数据，而优化算法需要到内存中读取 200084 量级的数据，所以会导致两种算法的时间效益差得很多，优化算法明显优于平凡算法。

# 3 问题二:n 个数求和

## 3.1 问题重述

计算 n 个数的和，考虑两种算法设计思路：

1. 逐个累加的平凡算法（链式）；

2. 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，直至只剩下最终结果。

## 3.2 算法设计

我们再来看第二个问题，要实现 n 个数的求和。首先想到的就是直接逐个遍历 n 个元素，实现 n 个数的相加，但这样只是利用了一条线程，后面的相加操作需要依赖前面计算的结果，需要很多的等待时间，这是很浪费时间的。由于计算机可以同时运行多条线程，我们可以分两条线路求部分和，然后再把部分和加起来，这样，我们利用了两条线程，形成两路链式累加，由于这两路是完全并行的，相邻的指令没有依赖关系，假如说一步相加操作需要一个时钟周期，那么就使得原来 n 个时钟周期完成的任务，我们只需 n/2 个周期就可以完成了，显著提高了运行效率，当问题规模 n 更大时，这种优化效果会更加明显。

经过分析，单纯递归算法的时间复杂度应为 O(nlogn), 我们在此不作数据分析，只将代码实现部分注释在代码块里。

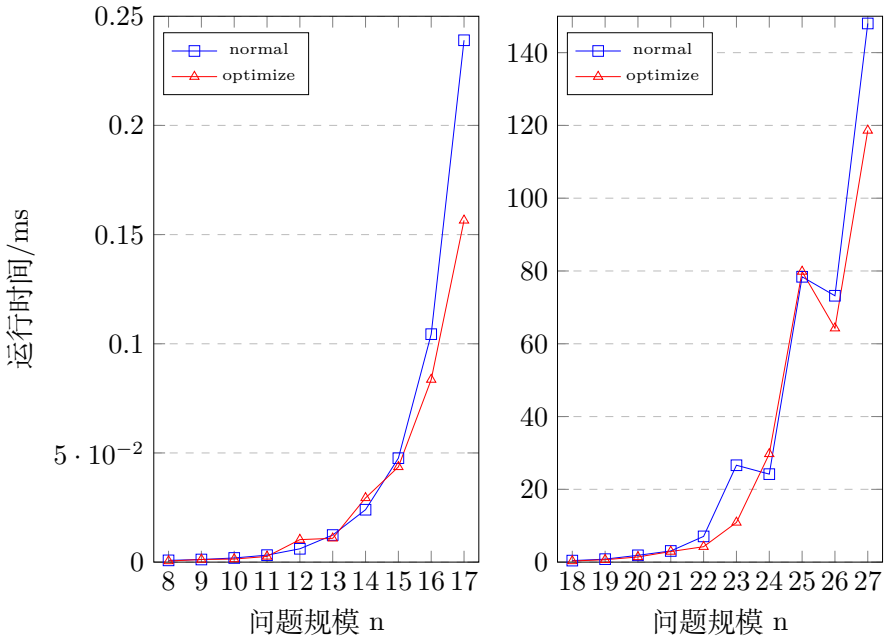
## 3.3 性能分析

### 3.3.1 数据分析

| n | normal | optimize | n | normal | optimize |
|---|---|---|---|---|---|
| 8 | 0.0008 | 0.0005 | 18 | 0.4166 | 0.3093 |
| 9 | 0.0012 | 0.0012 | 19 | 0.8171 | 0.6569 |
| 10 | 0.0019 | 0.0014 | 20 | 1.869 | 1.4207 |
| 11 | 0.0032 | 0.0026 | 21 | 3.0638 | 2.9082 |
| 12 | 0.0061 | 0.0103 | 22 | 7.07 | 4.234 |
| 13 | 0.0124 | 0.0109 | 23 | 26.6202 | 10.9017 |
| 14 | 0.024 | 0.0294 | 24 | 24.1774 | 29.7315 |
| 15 | 0.0476 | 0.0436 | 25 | 78.4004 | 79.8465 |
| 16 | 0.1044 | 0.0836 | 26 | 73.1902 | 64.2721 |
| 17 | 0.239 | 0.1565 | 27 | 148.014 | 118.609 |

表 4: $2^n$ 个数求和平凡与优化算法运行时间比较

我们取数据规模为 $2^n$，当数据规模较小时，运行时间很小，所以我们直接从 $2^8$ 开始记录，得到表 4，通过表中数据，我们绘制折线图如下。由折线图可知，无论是哪种算法，随着问题规模的增大，




运行时间也会增大，并且图像类似于指数函数，这是由于无论是哪种算法，时间复杂度都为 O(n)，但是我们可以看到，当数据规模比较大的时候，优化算法的时间效益会优于平凡算法一些，这是由于优化算法把问题分成了两个不相关的问题，把 n 个数求和分解为两条线路，根据超标量的原理，这样计算机在执行程序时就会同时调用两条流水线，起到了优化作用。

### 3.3.2 超标量分析

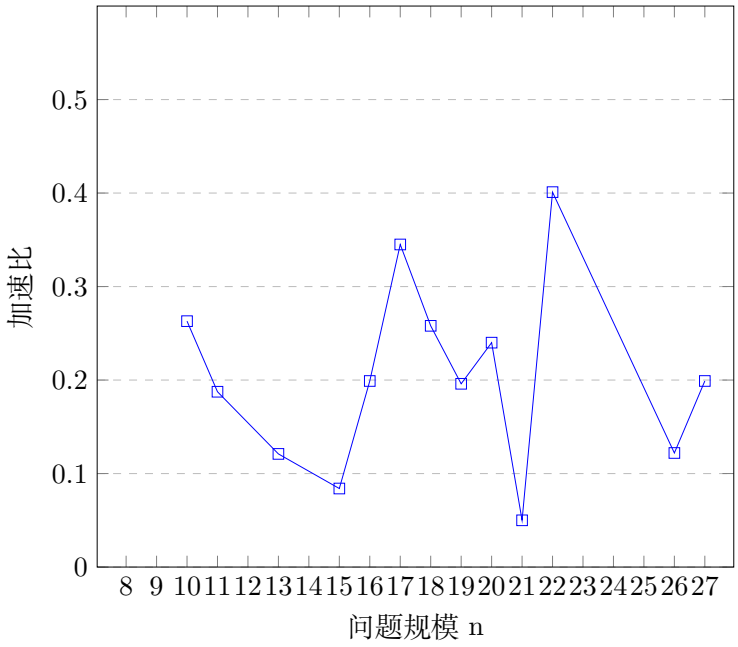
通过使用 vtune 监测两种算法的 CPI，我们得到如下的数据 (表 5)，可以看到，在数据规模比较大时，优化算法的 CPI 明显小于平凡算法，每条指令的时钟周期更小，所以对于相同指令数，CPU time

更小，效率更高。

| n | normal | optimize |
|---|---|---|
| 21 | 0.545 | 0.444 |
| 23 | 0.500 | 0.361 |
| 25 | 0.558 | 0.347 |

表 5: 不同规模下两种算法 CPI Rate

进一步分析，通过计算不同问题规模下优化算法的加速比，扣除可能由于计算机性能不稳定造成的负优化现象，我们得到下面的折线图。由于计算机性能不稳定，造成折线图数据部分不准，但是我们




可以知道，在数据规模比较小的时候性能会对运行时间造成比较大的影响所以我们不考虑，而 n=22、23 时明显是噪声数据，所以按图的发展趋势来看，在 n-17 时，加速比应为最大。而后加速比会有下降，也就是说在 n=17 时，优化程度达到最大，这之后优化减少，可能是由于大量访问了内存，使得超标量优化比受到限制。

# 4 一些发现：编译器优化

在查阅相关资料后，我发现，在编译器中，例如 visual studio c++，在项目属性->c++-> 优化-> 优化，可以开启编译器对程序执行的最大优化 O2，在开启了这个选项之后，我们发现，对于我们的优化算法和循环展开算法来说，运行时间大大减少，在查阅了相关资料后，我了解了这是由于现代编译器可以自动进行循环展开和指令调度，使得算法效率大大提高。而这是由于现代编译器将不同的代码生成相同的汇编语言，对于这样统一的语言，编译器可以进行优化，会去分析一些数据流和控制流的信息，确定哪部分可以优化，去评估优化的收益，这在一定程度上会优于手动循环展开，因为编译器可以根据 CPU 的硬件特性去决定是否对循环进行展开或确定展开的路数。

# 5 反思总结

**· 学习收获**

通过矩阵向量相乘的实验深刻理解行连续访问对性能的提升，特别是大规模数据下优化算法效率显著提升，了解了 cache 这种存储结构，能够分析优化算法快是由于利用了 cache 的缓存行结构，平凡算法慢是由于没利用这种机制，导致访问次数增多；通过两路累加算法，减少数据之间的依赖，验证了超标量架构的并行能力；学会了使用 vtune 分析缓存命中率与 CPI，并且学会了数据的可视化。另外，我还了解到了编译器可以对程序进行性能优化，并采用 O2 优化证明了这一发现。

**· 遇到的问题**

当数据规模较小时优化效果不是很明显，需要对数据进行一些舍取；递归求和算法处理大规模问题时，因栈空间占用，导致性能较低；计算机性能不稳定，可能是硬件预取、温度降频等现象导致实验结果波动。

**· 改进方向** 学习如何利用并行原理实现递归求和优于链式求和；探索多线程与 GPU 加速的可能性。

# 6 代码链接

点击访问 GitHub 主页