

# 计算机学院 并行程序设计实验报告

# CPU 架构相关编程

姓名:熊诚义

学号:2313310

专业:计算机科学与技术

# 目录

1	实验	一: n*n 矩阵与向量内积	2
	1.1	算法设计	2
		1.1.1 平凡算法设计思路	2
		1.1.2 cache 优化算法设计思路	2
	1.2	编程实现	2
		1.2.1 平凡算法	2
		1.2.2 cache 优化算法	3
	1.3	性能测试	3
		1.3.1 平凡算法	3
		1.3.2 cache 优化算法	4
	1.4	结果分析	4
2	实验	二: n 个数求和	5
	2.1	算法设计	5
		2.1.1 平凡算法设计思路	5
		2.1.2 cache 优化算法设计思路	5
	2.2	编程实现	6
		2.2.1 平凡算法	6
		2.2.2 cache 优化算法	6
	2.3	性能测试	7
		2.3.1 平凡算法	7
		2.3.2 cache 优化算法	8
	2.4	结果分析	8
n	心心	. ¼ ሬተ- <b>ሰ</b> ባ ዘነ ታል	•
3		总结和思考	9
	3.1	对比 2 个实验的异同	9
		3.1.1 相同点	9
	2.2	3.1.2 不同点	9
	3.2	总结	10
4	其他		10
	4.1	源码项目链接	10
	4.2	实验平台配置	10

## 1 实验一: n\*n 矩阵与向量内积

#### 1.1 算法设计

对于本问题,测试数据均为人为设定固定值,如 matrix[i][j]=i+j,方便程序正确性检查。

#### 1.1.1 平凡算法设计思路

首先,先回忆一下矩阵在内存中的存储方式。通常,在 C 或者 C++ 中,二维数组是优先存储的,也就是同一行的元素在内存中是连续的。

直接按照问题的直观逻辑设计算法,逐列遍历矩阵,计算每一列与给定向量的内积。先对外层循环遍历,从第 0 列开始,依次处理每一列 (j 从 0 到 n-1),再对当前列 j,遍历该列的所有行元素 (i 从 0 到 n-1),逐元素计算内积,得到该列的内积结果 res[j]。

平凡算法按列访问,每次访问 matrix[i][j] 后,下一个访问的 matrix[i][j+1] 位于下一行的同一列,内存地址间隔 n\*sizeof(element) 字节。

#### 1.1.2 cache 优化算法设计思路

cache 优化算法利用内存访问的连续性,通过调整计算顺序,使矩阵元素的访问模式与内存存储模式 (行优先) 一致,从而减少缓存缺失 (Cache Miss),提高计算效率。

步骤是先进行外层循环遍历行,从第 0 行开始,依次处理每一行 (i 从 0 到 n-1),接着进行内存循环遍历列,对当前行 i,遍历改行的所有列元素 (j 从 0 到 n-1),然后批量更新列内积,将当前行元素 matrix[i][j] 与向量元素 v[i] 相乘,并累加到结果数组 res[i] 中。

矩阵按行优先存储时,相邻内存地址对应同一行的连续列元素 (matrix[i][j] 与 matrix[i][j+1])。访问 matrix[i][j] 时,后续元素 matrix[i][j+1]、matrix[i][j+2] 等已被预加载到同一缓存行 (Cache Line),无需重复加载。外层循环每处理一行 i 时,向量元素 v[i] 被重复使用 n 次 (对应该行所有列的计算)。 v[i] 可被缓存在寄存器或 L1 缓存中,避免多次访问内存。结果数组 res 的访问模式为按列顺序 (res[j]、res[j+1]、…),虽然是非连续的,但现代 CPU 的写缓存机制可缓解写入开销。

#### 1.2 编程实现

#### 1.2.1 平凡算法

```
for(int i = 0; i < n; i++){
    res[i] = 0.0;
    for(int j = 0; j < n; j++){
    res[i] += martix[j][i] * v[j];
}</pre>
```

外层循环中,变量 i 表示当前处理的列索引 (从 0 到 n-1),初始化 res[i]=0.0,表示从零开始累加第 i 列的内积结果。

内层循环中,变量 j 表示当前处理的行索引 (从 0 到 n-1),访问矩阵元素 matrix[j][i],即第 j 行第 i 列的值,将其与向量元素 v[j] 相乘,并累加到 res[i] 中。

#### 1.2.2 cache 优化算法

```
for(int i = 0; i < n; i++){
    res[i] = 0.0;
}

for(int j = 0; j < n; j++){
    for(int i = 0; i < n; i++){
        res[i] += matrix[j][i] * v[j];
    }
}</pre>
```

初始化阶段将结果数组 res 的所有元素初始化为 0.0, 为后续累加做准备。

计算阶段,外层循环遍历行索引  $j(M \ 0 \ 20 \ n-1)$ ,每次处理矩阵的一行。内层循环遍历列索引  $i(M \ 0 \ 20 \ n-1)$ ,将当前行 j 的第 i 列元素 matrix[j][i] 与向量元素 v[j] 相乘,结果累加到 res[i] 中。

#### 1.3 性能测试

对程序性能进行测试,我们采用 VTune 进行剖析。同时,为了令结果更有说服力,我们可以测试不同问题规模(测试 n=1e3, 1e4),分析其与系统参数相对关系对性能的影响等。

为分析此程序, 我们希望看到程序执行时间等参数。可能需要额外假如额外的 Event, 在 Hotspots 窗口中选择"Hardware Event-Based Sampling", 然后编辑希望采样的时间。

为进一步比较分析程序性能,我们采用了两种类型对程序进行测试,分别是 Hotspots 和 Microarchitecture Exploration 类型。在它们的 Summary 数据下我们可以看到总体运行时间、CPU 时间、总体执行的周期数 (Clokticks)、执行指令数 (Instructions Retired) 以及 CPI(IPC 的倒数,每条指令执行的周期数)。

#### 1.3.1 平凡算法

对不同类型下不同问题规模下程序的性能进行测试。

#### Hotspots

```
© Elapsed Time ®: 3.081s ≿

③ CPU Time ®: 0.120s

Instructions Retired: 549,113,000

③ Microarchitecture Usage ®: N/A* of Pipeline Slot.

Total Thread Count: 4

Paused Time ®: 0s

*WA is applied to metrics with undefined value. There is no data to calculate the metric.
```

图 1.1: 问题规模 1e3 下 hs 平凡算法

#### 

图 1.2: 问题规模 1e4 下 hs 平凡算法

#### Microarchitecture Exploration

 ③ Clockticks:
 447,515,000

 ⑤ Instructions Retired:
 537,018,000

 ⑤ CPI Rate ⑨:
 0.833

 MUX Reliability ⑨:
 0.416 №

图 1.3: 问题规模 1e3 下 ue 平凡算法

 ③ Clockticks:
 13,340,785,000

 ⑤ Instructions Retired:
 34,978,740,000

 ⑤ CPI Rate ⑨:
 0.381

 MUX Reliability ⑨:
 0.987

Performance-core (P-core):

图 1.4: 问题规模 1e4 下 ue 平凡算法

#### 1.3.2 cache 优化算法

对不同类型下不同问题规模下程序的性能进行测试。

#### Hotspots

OPU Time ◎: 0.126s
Instructions Retired: 580,560,000
 Microarchitecture Usage ◎: N/A\* of Pipeline Slot.
Total Thread Count: 4
Paused Time ◎: 0s
 \*N/A is applied to metrics with undefined value. There is no data to calculate the

metric.

图 1.5: 问题规模 1e3 下 hs cache 算法

#### 

○ CPU Time ②: 2.110s
 Instructions Retired: 32,816,154,000

 ○ Microarchitecture Usage ②: N/A\*
 Total Thread Count: 4
 Paused Time ②: 0s

\*N/A is applied to metrics with undefined value. There is no data to calculate the metric

e 算法 图 1.6: 问题规模 1e4 下 hs cache 优化算法

#### Microarchitecture Exploration

O Clockticks: 125,788,000
 O Instructions Retired: 365,269,000
 O CPI Rate <sup>⊙</sup>: 0.344
 MUX Reliability <sup>⊙</sup>: N/A\*

Performance-core (P-core):

图 1.7: 问题规模 1e3 下 ue cache 优化算法

#### 

☼ Clockticks: 9,499,413,000
 ☼ Instructions Retired: 32,854,858,000
 ☼ CPI Rate ۞: 0.289
 MUX Reliability ۞: 0.947
 ❖ Performance-core (P-core):

图 1.8: 问题规模 1e4 下 ue cache 优化算法

#### 1.4 结果分析

我们比较在不同问题规模情况下两种算法的性能差异。

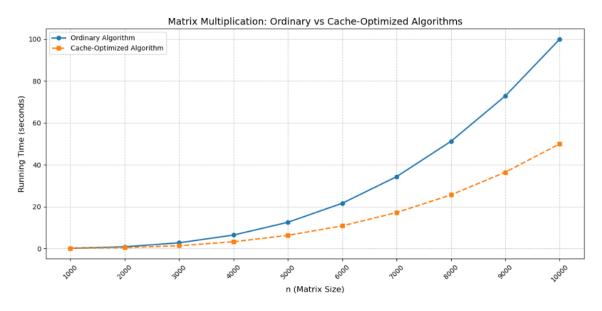


图 1.9: 两种算法不同规模运行时间比较

运行时间 可以直观的发现在小问题规模下平凡算法的运行时间不一定就比 cache 优化算法要长甚至有时要优于 cache 优化算法 (如在 Hotspots 类型下,如图1.1和图1.5)。而当问题规模变大时,这时平凡算法就明显不如 cache 优化算法。但总体来看,cache 优化算法在运行时间上要优于平凡算法。

**CPI** 通过比较在 Microarchitecture Exploration 类型下两种算法的 CPI Rate 可以发现,相同问题规模下,均是 cache 优化算法测试所得的 CPI 率更接近 0.25,即更加优秀。

# 2 实验二: n 个数求和

#### 2.1 算法设计

对于本问题,测试数据均为人为设定固定值,如 a[i]=i,方便程序正确性检查。

#### 2.1.1 平凡算法设计思路

核心思想: 顺序遍历数组,逐个元素累加到总和中。这是最直观的算法。

#### 2.1.2 cache 优化算法设计思路

通过拆分计算任务,减少数据依赖,利用 CPU 的多执行单元并行计算。

1) **两路链式累加 (循环展开)** 将数组分为两个部分,同时累加两个独立的和,最后合并结果,减少指令依赖。

**2) 递归分治算法 (两两相加)** 递归将给定元素两两相加,得到 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 n/4 个中间结果。依此类推,log(n) 个步骤后得到一个值即为最终结果。

实现方式 1: 递归函数 实现方式 2: 二重循环

#### 2.2 编程实现

#### 2.2.1 平凡算法

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

将给定元素依次累加到结果变量即可

#### 2.2.2 cache 优化算法

```
// 多链路式
   int sum1 = 0;
   int sum2 = 0;
   for(int i = 0; i < n; i += 2){
       sum1 += a[i];
       sum2 += a[i+1];
6
   }
   sum = sum1 + sum2;
q
   // 递归
10
11
   // 实现方法 1:
12
   function recursion(n){
13
       if(n == 1) return;
       else{
15
            for(int i = 0; i < n/2; i++){
16
                a[i] += a[n-i-1];
18
           n = n/2;
19
           recursion(n);
```

```
}
21
   }
22
23
   // 实现方法 2:
24
   for(m = n; m > 1; m /= 2){
                                     //log(n) 个步骤
25
       for(int i = 0; i < m/2; i++){
26
           a[i] = a[i * 2] + a[i * 2 + 1]; //相邻元素相加连续存储到数组最前面
       }
28
   } //a[0] 为最终结果
```

- 1 多路链式累加算法 将数组分为奇偶两部分,分别累加到 sum1 和 sum2. 最终结果为两部 分的合并。若 n 为奇数, 需单独处理最后一个元素 (代码中未体现)。循环中需确保 i+1 < n, 否则会越界(代码中未体现)。
- 2 递归分治算法 1) 递归函数实现:每次递归将数组后半部分累加到前半部分。递归深度 为 log(n), 最终结果存储在 a[0] 中。
- 2) 二重循环实现: 外层循环控制层级数 (log(n) 次)。内层循环将相邻元素相加, 结果 存储在数组前半部分。

#### 2.3 性能测试

这里我们对平凡算法和多路链式优化算法进行测试。

#### 2.3.1 平凡算法

对不同类型下不同问题规模下程序的性能进行测试。

#### **Hotspots**



图 2.10: 问题规模 1e3 下 hs 平凡算法

#### (2) CPU Time (2): 0.011s Instructions Retired: 38,704,000 N/A\* of Pipeline Slot CPI Rate 19: 1.062 Total Thread Count: 4 Paused Time ①: 0s \*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

图 2.11: 问题规模 1e4 下 hs 平凡算法

#### Microarchitecture Exploration

2 实验二: N 个数求和 并行程序设计实验报告

### 

Olockticks: 41,123,000
 Instructions Retired: 36,285,000
 CPI Rate ②: 1.133 ►
 Performance-core (P-core) ②: 1.133 ►
 Efficient-core (E-core) ③: 0.000 ►
 MUX Reliability ③: N/A\*

Performance-core (P-core):

图 2.12: 问题规模 1e3 下 ue 平凡算法

#### 

O Clockticks: 38,704,000
 O Instructions Retired: 38,704,000
 O CPI Rate ②: 1.000
 MUX Reliability ③: N/A\*
 O Performance-core (P-core):

图 2.13: 问题规模 1e4 下 ue 平凡算法

#### 2.3.2 cache 优化算法

对不同类型下不同问题规模下程序的性能进行测试。

#### **Hotspots**

### 

○ CPU Time ②: 0.012s
 Instructions Retired: 38,704,000

 ○ Microarchitecture Usage ②: N/A\* of Pipeline Slot
 Total Thread Count: 4
 Paused Time ③: 0s

\*N/A is applied to metrics with undefined value. There is no data to calculate the metric

图 2.14: 问题规模 1e3 下 hs cache 优化算法

#### 

\*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

图 2.15: 问题规模 1e4 下 hs cache 优化算法

0s

Microarchitecture Exploration

# 

 ③ Clockticks:
 38,704,000

 ③ Instructions Retired:
 43,542,000

 ⑤ CPI Rate ⑩:
 0.889

 MUX Reliability ⑩:
 N/A\*

图 2.16: 问题规模 1e3 下 ue cache 优化算法

### ⊙ Elapsed Time<sup>®</sup>: 1.547s ≽ 💺

Paused Time 19:

○ Clockticks: 38,704,000
 ○ Instructions Retired: 36,285,000
 ○ CPI Rate ②: 1.067 №
 Performance-core (P-core) ②: 1.067 №
 Efficient-core (E-core) ③: 0.000 №
 MUX Reliability ③: N/A\*

 ○ Performance-core (P-core):

图 2.17: 问题规模 1e4 下 ue cache 优化算法

#### 2.4 结果分析

我们比较在不同问题规模情况下两种算法的性能差异。

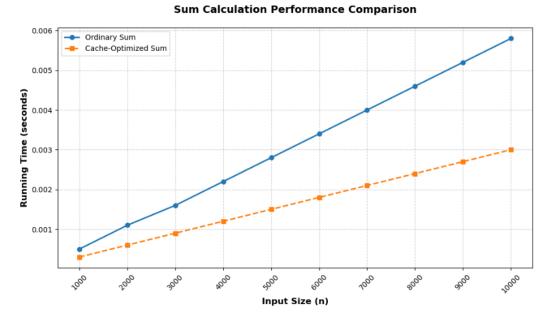


图 2.18: 两种算法不同规模运行时间比较

运行时间 由测试数据可以看出,当问题规模较小时,cache 优化算法的运行时间明显小于平凡算法,而当问题规模增大时,其结果却具有不定性。但总体来看,cache 优化算法在运行时间上要优于平凡算法。

**CPI** 由测试数据可以看出,虽然在小问题规模下,cache 优化算法的 CPI 略优于平凡算法,但在问题规模较大时两种算法的 CPI 却相差并不大,而且两种算法的 CPI 都并不是很优秀。

# 3 实验总结和思考

#### 3.1 对比 2 个实验的异同

#### 3.1.1 相同点

两个实验均旨在通过优化算法设计提升计算效率,解决平凡算法在特定场景下的性能瓶颈。

两种问题的平凡算法与优化算法的时间复杂度相同 (矩阵内积为  $O(n^2)$ , 求和为 O(n)),但优化算法通过减少常数因子提升实际性能。

#### 3.1.2 不同点

**实验**一 该实验优化方向为缓存优化,具有内存访问连续性,平凡跨列访问导致缓存未命中率高。通过调整遍历顺序,即从列优先转变为行优先,提高缓存行利用率,使得缓存未命中率大大降低。

**实验**二 该实验优化方向为超标量优化,具有指令级并行性,平凡链式累加导致指令依赖,限制并行性。通过循环展开、分治递归等技术提高指令流水线效率,使得程序执行时间减少。

#### 3.2 总结

算法的性能优化本质上是针对硬件特性的适配。不同问题的主要瓶颈可能截然不同。

- 1 实验一优化最大化内存局部性,通过连续内存访问(行优先)减少缓存未命中,适合处理大规模矩阵运算。实验二最小化数据依赖,通过拆分计算任务(多路累加、分治递归)提高指令级并行性,适合单核 CPU 优化或 SIMD 加速。
- **2** 优化算法常需要引入复杂逻辑,可能降低可维护性,需要根据场景进行权衡。在复杂任务中,也可能需同时优化内存访问和指令并行。在极端性能场景又需从算法设计阶段考虑硬件特性。
- **3** 两个实验虽问题不同,但共同揭示了一个核心规律: 高效算法 = 正确逻辑 + 硬件适配。优化不是魔法,而是对硬件行为的精准把控。

## 4 其他

#### 4.1 源码项目链接

并行 GitHub 源码链接

#### 4.2 实验平台配置

该实验均使用本地 C++ 编译器执行。