南開大學
Nankai University

计算机学院

并行程序设计实验报告

体系结构实验报告

姓名：张铭徐

学号：2113615

专业：计算机科学与技术

2023 年 3 月 10 日

目录

1 实验背景相关介绍

新信息时代下，各行各业都在与计算机结合，焕发新的生机，在万物互联的时代之下，算力便显得格外重要，尤其是新 CPU 架构设计，高性能计算等领域焕发了前所未有的生机。而高性能计算和算法性能优化刚需计算机体系结构相关的背景知识，本次实验将尝试初步分析相同算法时间复杂度下计算机执行不同算法时间的差异，借助 VTune 现代分析软件，从底层 CPU 执行的指定数，CPU-clock 等角度入手，试图从底层了解上述问题发生的原因。并对两种问题的平凡算法进行优化，通过数据分析对比性能的差异。

1.1 实验平台及相关配置介绍

本次实验借助 X86 指令集下的 Windows 11 系统，应用 Intel Core 11th 系列芯片做相关实验，芯片的具体型号和规格如下表5所示。电脑型号为联想拯救者，内存为 16GB，编译器选择 Microsoft Visual Studio-2019，编译力度选取无优化（未开启 O2，O3 系列优化），全程在笔记本环境下进行实验。对于有显著性差异的数据采用多次采样取平均值等办法消除误差。

核心指标	相应数据
CPU 主频	2.5GHZ
核心数	8
制造工艺	14nm
架构	Rocket Lake S
GPU 频率	0.35GHZ
GPU 加速频率	1.3GHZ

表 1: I7 11th 系列芯片参数

1.2 实验工具

本次实验采用的主要工具如下表2所示：

编译器	Microsoft Visual Studio-2019
分析软件	VTune
绘图工具	Echarts
写作工具	LaTeX

表 2: 实验主要工具

2 任务一：n*n 矩阵与向量内积

2.1 问题重述与分析

题目为：给定一个 n×n 矩阵，计算每一列与给定向量的内积。首先在数学上，向量内积也叫点积，其运算公式为 $\overrightarrow{a} = (x_1, x_2...x_n)$, $\overrightarrow{b} = (y_1, y_2...y_n)$, $\overrightarrow{a} \cdot \overrightarrow{b} = x_1y_1 + x_2y_2 + ... + x_ny_n$. 题目要求求矩阵每一列与给定向量内积，那么可以有以下方案。

2.2 平凡算法与优化算法设计

2.2.1 平凡算法

通过前面的分析，我们知道如果想计算给定两组向量内积，只需要遍历向量中每一个坐标维度，并做数乘，最后将所得结果累加即可，所以我们对于平凡算法可以采用 for 循环，逐列访问矩阵元素：一步外层循环（内存循环一次完整执行）计算出一个内积结果。时间复杂度为 $O(n^2)$。

2.2.2 优化算法

由于需要求出给定矩阵每一列与给定向量的内积，我们需要遍历矩阵中的每一个元素，所以最优的时间复杂度必然为 $O(n^2)$，并没有比其时间复杂度更加优秀的算法。所以我们优化策略将从计算机体系结构的角度入手进行优化：我们考虑 C++ 内部的存储逻辑，对于一个二维数组 $a[n][m]$，C++ 将会为其分配一段连续的地址空间用于存储这 $n*m$ 个元素，将第一行的 m 个元素存储完毕后，在第 m 个元素后面的地址空间接着存储下一行的元素，以此类推。故我们在考虑对平凡算法进行优化时，可以考虑利用二维数组的存储模式进行加速。

在前面的平凡算法，我们是逐列访问矩阵中的元素，每一步外层循环都计算出一个值，但是很显然，这与二维数组的存储方式并不相符，如果我们考虑按照逐行遍历矩阵中的元素，那么最后在计算机的内部，将会是访问连续的地址空间，很显然这样做具有很好的空间局限性，令我们计算机的高速缓存模块 (cache) 的作用能够得以完全的发挥，故我们在设计算法时，可以考虑将其改为逐行访问矩阵元素：一步外层循环计算不出任何一个内积，只是向每个内积累加一个乘法结果，最后最后一次外层循环才计算出对应的值。根据上述的步骤，我们很轻松能够实现以下代码：

2.3 编程实现

我们在代码中约定：对于数据规模比较小的样例，我们采用多次执行同一段代码的方法以优化可能存在的系统误差。同时，对于代码中出现的几个变量，其含义如注释所述。其中，矩阵和给定的向量可以随机指定，任意生成，为了实验方便起见，我们将其指定为 $a[i]=i, b[i][j]=i+j$ 即可，故平凡算法和优化算法如下表所示：

逐列访问平凡算法

```
inline void func()
{
    double head,tail,freq,head1,tail1,timess=0;
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&head);
    for(int i=1; i<=n; i++) a[i]=i; //a[i]为给定向量
    for(int i=1 ;i<=n; i++)
    for(int j=1; j<=n ;j++) b[i][j]=i+j;//数组b存储给定矩阵
    for (int i=1; i<=n; i++)
        for (int j=1; j<=n; j++)
            sum[i] += (b[j][i]*a[j]); //sum[i]存储第i列与给定向量内积
    QueryPerformanceCounter ((LARGE_INTEGER *)& tail) ;
    cout << "\nordCol:" <<(tail-head)*1000.0 / freq<< "ms" << endl;
}
```

逐行访问优化算法

```
inline void func()
{
    double head,tail,freq,head1,tail1,timess=0;
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&head);
    for(int i=1; i<=n; i++) a[i]=i; //a[i]为给定向量
    for(int i=1 ;i<=n; i++)
    for(int j=1; j<=n ;j++) b[i][j]=i+j;//数组b存储给定矩阵
    for(int j=1; j<=n; j++){
      for(int i=1; i<=n ;i++)
          sum[i] += b[j][i] * a[j];
    }
    QueryPerformanceCounter ((LARGE_INTEGER *)& tail) ;
    cout << "\nordCol:" <<(tail-head)*1000.0 / freq<< "ms" << endl;
}
```

2.4 性能测试与数据对比

我们使用了 windows 中自带的计时函数 QueryPerformance 对两种算法的执行时间进行对比，下面简单记录一下 QueryPerformanceFrequency 的相关用法:QueryPerformanceFrequency 用于获取计数频率，QueryPerformanceCounter 用于获取当前时间，通过执行代码前后的 Counter 值，在除以相应的频率，即可得到程序执行时间。平凡算法和优化算法在不同数据规模和重复次数下的执行时间如下表5所示：

数据规模 (n)	算法重复次数	平凡算法执行时间 (ms)	优化算法执行时间 (ms)
10	294	301	198
20	348	864	814
30	512	3775	3421
40	429	6260	**6799**
50	129	1633	1572
60	279	4681	4470
70	179	3423	3288
80	179	5549	5349
90	60	4528	3595
100	60	5194	4269
200	12	4842	4128
300	9	6895	4973
400	4	4496	4392

表 3: 算法执行时间对比

可以发现，除了数据规模为 40，执行次数为 429 次时，优化算法反而较平凡算法慢，剩下的测试数据优化算法都比平凡算法更快一些，实验数据与我们前文算法设计时的结论恰好吻合。下面我们将借助于 VTune 对结果进行进一步的底层分析，探究到底是什么原因导致了这种现象的发生。

2.5 基于 VTune 对结果进行分析

借助于 VTune 的 HardWare Event-Based Sampling，可以测试出平凡算法和优化算法的指令数，CPU 时间，并行指数，CPI，各级缓存的命中率等参数。在这里，我们采用数据规模为 1000，执行次数为 100 的情况下进行分析，平凡算法和优化算法的各项输入数据均相同。不同算法的各级参数见下表所示：

参数指标	cache 优化算法	平凡算法
指令数	2834329212	2863710778
CPU 时间	0.254	0.321
Parallelism	5.6	1.6
CPI	0.379	0.458
$L1miss$	1200240	96650508
$L2miss$	0	5101077
$L3miss$	0	0
L1 命中率	0.999	0.944
L2 命中率	1	0.95
L3 命中率	None	1

表 4: 两种算法 VTune 底层参数对比

通过上面数据我们可以清楚的看到：优化算法极大程度上发挥了 cache 的高速缓存的作用，L1 缓存速度最快，优化算法也对 L1 的利用率最高，所以用 cache 优化算法执行速度也最快，从 CPU 时间和 CPI 等因素也可以清楚的看到这一点，所以通过上述硬件层面分析，我们可以下定结论：优化算法之所以比平凡算法执行效率更高，是因为对高速缓存 cache 的利用率远高于平凡算法。

3 任务二：n 个数求和

3.1 问题重述与分析

题目为：计算 n 个数的和。这个题目很简单，求和只需要遍历每一个数，然后将其加到我们的统计变量中即可，所以我们可以设计出以下方案：

3.2 平凡算法与优化算法设计

3.3 平凡算法

平凡算法只需要用 for 循环将 n 个变量的值逐渐遍历，然后累加即可，时间复杂度为 $O(n)$。

3.4 优化算法

我们考虑，虽然我们时间复杂度为 $O(n)$，但是我们的 for 循环会出现一个问题，便是在循环中，我们不仅仅需要执行原本加法指令，同时也需要考虑循环变量的判定，迭代这些额外的操作。这些额外的操作也会带来额外的开销。那么我们的优化思路便很清晰：使用循环展开 (unroll) 策略，即减少循环的判定枚举次数，增加单次循环内的加法操作次数，也可以理解为将多个循环的工作合并到了一个循环周期内进行操作，这样我们便可以大幅度降低循环所带来的额外时间开销。

3.5 编程实现

基于以上的讨论，我们可以很轻松的写出平凡算法和超标量优化算法，在算法中，我们采用了递归的方法进行超标量优化。对于数据规模较小数据，我们在本次实验中仍然采取多次执行的方法进行延时操作。为实验方便起见，我们指定 $a[i] = i$ 即可，故平凡算法和优化算法如下所示：

未优化 for 循环平凡算法

```
inline void func()
{
    double head,tail,freq,head1,tail1,timess=0;
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&head);
    for(int i=1; i<=n; i++) a[i]=i; //a[i]为给定数据
    for(int i=1 ;i<=n; i++) sum+=a[i]; //sum存储求出的和
    QueryPerformanceCounter(&tail);
    cout << "\nordCol:" <<(tail-head)*1000.0 / freq<< "ms" << endl;
}
```

Unroll 优化算法

```
inline void recursion(int n)
{
    if(n==1) return ;
    for(int i=1;i<=n/2;i++) a[i]+=a[n-i+1];
    recursion(n/2);
}
int main(){
  double head,tail,freq,head1,tail1,timess=0;
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&head);
    recursion(n);
    QueryPerformanceCounter(&tail);
    cout << "\nordCol:" <<(tail-head)*1000.0 / freq<< "ms" << endl;
}
```

3.6 性能测试与数据对比

数据规模 (n)	算法重复次数	平凡算法执行时间 (ms)	优化算法执行时间 (ms)
2	300000	1188	8564
4	150000	1523	6806
8	75000	1690	4964
16	37500	1736	3539
32	18750	1912	2632
64	9000	2175	2446
128	9000	3175	5051
256	5000	3465	5451
512	4000	4528	3595
1024	2500	5194	4269

表 5: 算法执行时间对比

3.7 基于 VTune 对结果进行分析

借助于 VTune 的 HardWare Event-Based Sampling，可以测试出平凡算法和优化算法的指令数，CPU 时间，并行指数，CPI，各级缓存的命中率等参数。在这里，我们采用数据规模为 8192，执行次数为 1000 的情况下进行分析，平凡算法和优化算法的各项输入数据均相同。不同算法的各级参数见下表所示：

参数指标	Unroll 优化算法	平凡算法
指令数	267997439	2866545932
CPU 时间	0.045	0.321
Parallelism	0.3	0.3
CPI	0.384	0.472
$L1miss$	0	96652683
$L2miss$	0	5101077
$L3miss$	0	0
L1 命中率	1	0.944
L2 命中率	None	0.95
L3 命中率	None	1

表 6: 两种算法 VTune 底层参数对比

从上表我们可以看到：unroll(循环展开) 策略在一定程度上减少了执行该程序所需要的指令数，同时，其 CPI 值从那个 0.472 下降为 0.384，这意味着执行一条指令所需要的时钟周期数显著降低。同时，Unroll 循环展开算法通过递归等方式将问题分解，也使得计算机内部对于高速缓存区域的利用率提升。不过值得注意的是，本次实验中，采取 Unroll 方式为递归，其优点为代码写起来相对简便，而且可读性较高，而缺点就在于需要额外的递归函数堆栈的开销这点在计时器的结果中体现出了较大的差异即优化算法的执行时间大于平凡算法的时间。如果采用不同的 Unroll 方法，消除掉递归函数的影响，实验结果的差异性可能会更加明显。实验所采用的数据规模为 8192 和 1000 次重复执行，在尽可能消除掉系统误差的同时，保证了结果具有一定的可信度。

总之，Unroll 算法通过减少循环周期，增加每个循环周期内的运算量，可以显著提高 CPI 等参数，进而提高程序的性能。与传统方法比，更能够体现出并行体系结构的特点。

4 总结与反思

4.1 实验结论

本次实验的实验一通过遵循计算机内部行主次序存储的二维数组，通过每次访问行中的元素使得计算机内部去高速缓存 L1 能够得到非常充分的利用，进而从底层可以提高程序的性能；而实验二则可以通过减少循环次数，增加单次循环内的指令数来减少循环结构需要额外判定循环变量是否满足条件等操作，通过减少指令数，提高 CPI 来提升性能，不过实验二采取的是递归方法实现 Unroll，有额外的递归函数的开销在，所以在实验结果测试中导致了看起来优化算法的性能还不如平凡算法。而通过 VTune 软件分析可知，Unroll 策略的确可以提高算法的性能。

4.2 实验总结与反思

本次实验的难点在于配置环境，本次实验除了配置了 VTune 实验环境外，额外配置了 wsl2 系统，但由于 wsl2 中硬件采样功能消失，导致不得不用 VTune 分析性能。此外，通过切实操作，VTune 分析得到的数据可以切实的体会到一个程序的性能不仅仅由算法的优劣，时间复杂度决定，还可以从很多细枝末节甚至不起眼的地方进行优化，这些优化力度看起来小，但是累加起来却可以节约出很大一部分性能。

通过本次实验，也增加了对于计算机底层一些评估参数的理解，例如 CPU 时间，CPI，IPC 等参数，以及增加了对计算机内部缓存区域的理解，即分为三级缓存，L1，L2，L3，三级缓存的速度依次递减，容量依次增大，同时树立了设计算法要与底层逻辑相称的理念。

本次实验不足的地方在于对于 VTune 等性能分析软件的掌握程度还不熟练，同时在前期配置环境时浪费时间过多，以及面对陌生的软件不知道从何下手的问题，希望在下次实验中，能够解决或者尽可能的规避这些问题！