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

姓名：苏耀磊

学号：2311727

专业：计算机科学与技术

2025 年 3 月 27 日

目录

1 实验环境

架构	x86
CPU 型号	13th Gen Intel® Core™ i5-13500H
CPU 主频	2.6GHz
CPU 内核数	12
一级缓存 L1	1.1MB
二级缓存 L2	9.0MB
三级缓存 L3	18.0MB

实验相关代码已上传至Github。

2 实验一：n*n 矩阵与向量内积

2.1 问题描述

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：逐列访问元素的平凡算法和 cache 优化算法。

2.2 算法设计

实验数据均由自己生成，矩阵元素由该位置的行标和列标相加得到，即 a[i][j] = i + j；相乘向量对应位置元素等于其下标，即 b[i] = i。

2.2.1 平凡算法设计思路

平凡算法逐列访问矩阵元素，锁定某一列，每一次循环都读取该列的所有元素，计算内积结果，以此循环往复，直到计算出所有结果。

2.2.2 cache 算法设计思路

cache 优化算法则改为逐行访问矩阵元素，将读取到的元素先与向量中对应位置元素相乘，矩阵第一行的所有元素与向量中的第一个元素相乘，每一个结果先加到 sum 中对应位置，矩阵第二行的所有元素与向量中的第二个元素相乘，每一个结果再加到 sum 对应位置，以此类推，直到所有行都计算完成，得到最终结果。这样的访存模式具有很好空间局部性，令 cache 的作用得以发挥。

2.3 实验分析

矩阵在内存中的存储是按照行主存储，但是实验要求得到每一列与向量的内积，平凡算法按照逐列访问，相当于每一次读取矩阵元素，一并得到的与该元素行相邻的元素没有得到有效利用，因此要想得到矩阵中的元素，必须将所有元素全部访问一次，而访问内存的时间成本很大，这是我们无法接受的。

经过 cache 优化后的算法逐行访问矩阵元素，每一次访问某元素同时得到与其行相邻的元素，都能在下一次循环中被利用，从而不需要再额外去访问内存，虽然每一步无法直接得到对应结果，但是大大降低时间开销。

2.4 编程实现

2.4.1 x86 平台下的平凡算法

逐列访问平凡算法

```
  //m为程序执行次数
void gettime_normal(int m) {
  initial();
  long long head, tail, freq;
  QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
  QueryPerformanceCounter((LARGE_INTEGER*)&head);
  //设置执行次数
  for (int f = 0; f < m; f++) {

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

  }
  QueryPerformanceCounter((LARGE_INTEGER*)&tail);
  double time_normal = (tail - head) * 1000.0 / freq;
  cout<< "执行"<<m<<"次normal时间: " << time_normal << " ms" << endl;
}
```

2.4.2 x86 平台下的 cache 优化算法

cache 优化算法

```
  //m为程序执行次数
void gettime_cache(int m) {
  initial();
  long long head, tail, freq;
  QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
  QueryPerformanceCounter((LARGE_INTEGER*)&head);
  //设置执行次数
  for (int f = 0; f < m; f++) {

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

```
18
19      }
20      QueryPerformanceCounter((LARGE_INTEGER*)&tail);
21      double time_cache = (tail - head) * 1000.0 / freq;
22      cout << "执行" << m << "次cache时间: " << time_cache << " ms" << endl;
23  }
```

2.5 性能测试

对于解决该方法涉及到的两种算法，我在个人电脑（x86 平台）上测试了程序运行时间，由于本机的三级缓存分别为 1.1MB、9.0MB、18.0MB，能容纳的矩阵维数分别约为 366 维，1024 维以及 1443 维，于是我在设置矩阵的大小时插入了这三个数据，如表1所示。由表上数据可知，在矩阵大小不超过 400*400 时，逐行逐列两种访问方式的效率相差不大，但是 cache 优化算法依旧略优于平凡算法；而当矩阵规模更大时，二者的运行速度差距明显增大，cache 优化算法的的效率明显优于平凡算法；直到矩阵维数超过 2000 之后，cache 优化算法的效率远远超过平凡算法，这意味着缓存优化发挥了巨大的作用。

N	Iteration	Normal	Cache
50	10000	8.9255	8.7837
100	1000	4.6053	3.7162
200	1000	16.9203	11.946
366	1000	57.8378	40.5322
400	1000	71.4125	46.5207
800	100	51.335	27.6151
1024	10	24.7612	3.3297
1443	10	26.0892	8.516
1600	10	34.5648	12.2383
2000	1	10.2098	2.0231
3200	1	49.5158	5.2252
5000	1	98.7943	14.1023
6400	1	311.3	21.0029
8000	1	354.158	35.1879
10000	1	520.459	52.3899

表 1: x86 架构下的两种算法性能测试结果 (单位:ms)

2.6 profiling

对于平凡算法和 cache 优化算法，我们使用 VTnue 采集了数据规模在 50 和 5000 的两组实验各级缓存的访问次数，由于当规模为 50 时，数据规模较小，可能得到结果并不准确，于是我设置规模为 50 时运行 10000 次，规模为 5000 时运行 1 次，测试结果如表2所示。

由记录表可知，在矩阵规模为 50 时，一级缓存 L1 在平凡算法和 cache 改进算法的访存次数相差不大，从而导致在这个时候，两种方法的实现时间相差不大，性能相差无几；而当矩阵的规模增大到 5000 时，平凡算法的 L1 cache 命中率为 94.1%左右，cache 改进算法为 99.8%，明显看到 cache 优化方法一级缓存 L1 的命中率更高，使得二级缓存需要命中次数更少，以此类推，三级缓存 L3 访存次数也减少，因为一二级缓存的访问速度要明显快于三级缓存，这意味着 cache 优化算法的运行速度将会远快于平凡算法，从而性能更加出色，这也解释了上一张表两种算法的性能测试结果。

	L1 HIT	L1 MISS	L2 HIT	L2 MISS	L3 HIT	L3 MISS
			normal			
N=50	102095714	20042	20042	0	0	0
N=5000	434001302	27200408	24000360	3600756	2418976	275942
			cache			
N=50	100092480	0	0	0	0	0
N=5000	438001314	800012	800012	200042	300126	230540

表 2: 两种算法的各级缓存访问次数记录

3 实验二：n 个数求和

3.1 问题描述

计算 n 个数的和，考虑两种算法设计思路：逐个累加的平凡算法（链式）；适合超标量架构的指令级并行算法（相邻指令无依赖），如使用两路链式累加。

3.2 算法设计

实验数据均由人为生成，数组赋值方法为 a[i] = i。

3.2.1 平凡算法设计思路

将数组的单个数进行累加，得到最终结果（链式求和）。

3.2.2 超标量架构的指令级并行算法

定义两个临时变量部分和，分别计算奇数项和偶数项的和，最后将两个部分和相加（两路链式累加）。

3.3 实验分析

平凡算法每次都在同一个变量上进行累加，有多少个元素就要执行多少个周期，只能调用 CPU 的一条流水线进行处理，因此需要改进算法。

超标量改进后的算法设置两个临时变量，在一个循环周期内采用多条流水线对不同位置的元素进行累加，使得所需的循环周期数减少，降低了循环的开销。

3.4 编程实现

3.4.1 x86 平台下的平凡算法

单个数累加平凡算法

```
1 void gettime_normal(int m) {
2     long long head, tail, freq;
3     QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
4     QueryPerformanceCounter((LARGE_INTEGER*)&head);
```

```
    //设置执行次数
    for (int f = 0; f < m; f++) {

      result = 0;
      for (int i = 0; i < n; i++) {
          result += a[i];
        }

    }

    QueryPerformanceCounter((LARGE_INTEGER*)&tail);
    double time_normal = (tail - head) * 1000.0 / freq;
    cout <<"执行"<< m <<"次normal时间: " << time_normal << " ms" << endl;
}
```

3.4.2 x86 平台下的超标量优化算法（指令级并行）

两路链式累加优化算法

```
void gettime_pro(int m) {
    long long head, tail, freq;
    QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
    QueryPerformanceCounter((LARGE_INTEGER*)&head);
    //设置执行次数
    for (int f = 0; f < m; f++) {

      result = 0;
      long long int result1 = 0, result2 = 0;
      for (int i = 0; i < n; i += 2) {
       result1 += a[i];
       result2 += a[i + 1];
      }
      result = result1 + result2;

    }

    QueryPerformanceCounter((LARGE_INTEGER*)&tail);
    double time_pro = (tail - head) * 1000.0 / freq;
    cout <<"执行"<< m <<"次pro时间: " << time_pro << " ms" << endl;
}
```

3.5 性能测试

通过修改参与运算的数组大小，得到的程序运行时间如表3所示 (m 表示数组的规模为 2 的 m 次幂，下同)。

因为我们并未修改访问内存读取元素具体数值的方法，仅仅在原条件下新增加一条链路进行运算，因此，由表上数据可知，超标量算法的运行时间大致等于平凡算法运行时间的一半，而每一次扩大数

2 的 m 次幂	平凡算法消耗时间	超标量算法消耗时间
16	0.2715	0.1374
17	0.5427	0.3684
18	1.09	0.7654
19	2.3866	1.1134
20	4.3673	2.1815
21	9.1341	4.6382
22	17.7804	9.5529
23	37.289	18.3792
24	70.3709	36.9807
25	139.657	72.0601
26	288.75	149.739
27	576.308	289.969

表 3: x86 架构下的两种算法性能测试结果 (单位:ms)

组长度到原来的二倍，所消耗的时间也约等于之前的二倍。得到结果符合该程序的线性时间复杂度。

3.6 profiling

我们采集规模为 2 的 21 次幂和 2 的 27 次幂的两个程序数据，记录它们运行时的各级缓存的 Hit 次数和 Miss 次数, 以及各级缓存的命中率，如表4，表5所示。

	L1 HIT	L1 MISS	L2 HIT	L2 MISS	L3 HIT	L3 MISS
			单链路			
m=21	10760032	28001	41032	4000	3981	1421
m=27	668422005	11120166	10524157	686144	20004	75942
			两条链路			
m=21	8560025	44001	44001	2001	1681	241
m=27	526321578	8204123	7648114	494103	22004	14572

表 4: 两种算法的各级缓存访问次数记录

	L1 cache 命中率	L2 cache 命中率	L3 cache 命中率
		m = 21	
单链路	>99%	>91.11%	>73.69%
两条链路	>99%	>95.65%	>87.46%
		m = 27	
单链路	> 98.36%	>93.87%	>20.84%
两条链路	>98.46%	>93.93%	>60.15%

表 5: 不同问题规模下各级缓存命中率

由表中数据可知，在 m = 21 时，两种算法的 L1 cache 命中率基本无差别，但是由于单链路算法在之后的 L2、L3 cache 命中率都与两条链路实现的算法有一定差距，因此需要多次访问内存，增大了时间开销，运行的时间结果因而有巨大差别；而当 m 增大到 27 时，两种算法在 L1、L2 的 cache 命

中率都相差不大，但是就 L3 的 cache 命中率来说，两种算法都有所下降，尤其单链路的命中率较低，这表明二者都对内存进行大量访问，因此时间开销都增大，而单链路的访问次数更多，程序运行相对更慢。

4 总结

本实验涉及到的矩阵乘法和数组求和两个问题，串行算法都是相对更符合人的思维逻辑，但是并不能同计算机的硬件结构很好的集合起来，在数据规模较大时性能表现很差。于是为了利用 CPU 的多级缓存以及多条流水线进行工作，我们分别设计了行优先读取矩阵元素以及两条链路并行计算求和的改进算法。在读取矩阵元素时，通过提高 cache 命中率来减少对内存的访问次数，极大减少了程序时间开销；数组求和时，我们不再使用单一变量逐个累加，而是添加两个临时变量分别求和，充分利用 CPU 的多条流水线同时作业，提升程序性能。

除此之外，我也掌握了性能分析工具 VTune 的使用方法，对每个程序进行了细致的 profiling，能够更直观地感受到每个算法的性能差距在哪里，并且能够根据性能分析对最后得到的结果进行解释。比如在当我们提升 cache 的命中率和采用多条流水线工作后，可以明显观察到程序的性能得到极大提升，这是我们所期望的。