并行程序设计报告二

CPU 架构编程

2313911 黄尚扬 2025.3.21

Git 链接：https://gitee.com/hsy23333/parallel-jobs

一．实验一：n*n 矩阵与向量内积

（一）算法设计

1. 平凡算法设计思路

暴力地，我们采用模拟手算的过程，对于结果向量的每一位，遍历矩阵的每一列，分别累加，算完每个结果后跳到下一个要计算的位置。

2. Cache 优化算法设计思路

考虑到行优先的 cache 存取方式，在遍历时优先按照行来读取；每次遍历矩阵一行，叠加一部分到结果向量的对应位置上；最后遍历完整个矩阵得到结果。

（二）编程实现

1. 平凡算法

```
//朴素算法
for(int i=1;i<=n;++i){
    sum[i]=0.0;
    for(int j=1;j<=n;++j){
        sum[i]+=a[j]*b[j][i];
    }
}
```

实现如图，非常暴力，不做解释。

2. Cache 优化算法

```
//优化算法
for(int i=1;i<=n;++i)   sum[i]=0.0;
for(int j=1;j<=n;++j){
    for(int i=1;i<=n;++i){
        sum[i]+=a[j]*b[j][i];
    }
}
```

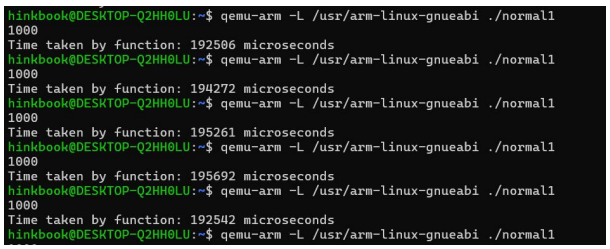
如图，改为了行优先读取，结果很显然不变；而同时利用了 cache 的存储特点。

（三）性能测试

在输入 n 后，采取随机生成的方式模拟相应规模的执行，如下：

```
for(int i=1;i<=n;++i)   a[i]=rand()%10000;
for(int i=1;i<=n;++i)   for(int j=1;j<=n;++j)   b[i][j]=rand()%10000;
```

笔者测试的 n 规模有：50，100，200，300，500，1000。每一算法每一规模重复测试 5 次，例子如下：



如图所示，这是优化算法 n=1000 的测试结果，随后取平均值即可得到最终结果。**注意，由于是使用 rand()函数简单取随机数，所以可以保证相同规模每次实验数据相同。**

1. 平凡算法

表 1：测试结果

n	时间 1（ms）	时间 2（ms）	时间 3（ms）	时间 4（ms）	时间 5（ms）	平均（ms）
50	1.292	1.323	1.156	1.140	1.186	1.2194
100	2.613	2.539	2.772	2.521	2.483	2.5856
200	8.320	8.290	8.134	9.886	8.113	8.5486
300	17.908	23.007	16.950	17.762	17.222	18.5698
500	48.817	47.713	48.064	48.010	49.563	48.4334
1000	197.932	200.063	195.827	197.168	187.095	195.617

2. Cache 优化算法

表 2：测试结果

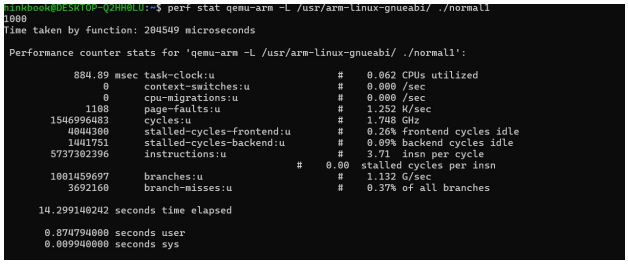
n	时间 1（ms）	时间 2（ms）	时间 3（ms）	时间 4（ms）	时间 5（ms）	平均（ms）

50	1.147	1.332	1.227	1.150	1.789	1.329
100	2.524	2.526	2.501	2.670	2.484	2.5410
200	8.840	7.946	8.457	7.917	7.912	8.2144
300	17.216	17.853	17.024	17.137	17.028	17.2516
500	46.693	46.733	47.008	46.469	47.786	46.7978
1000	192.506	194.272	195.261	195.682	192.542	194.0526

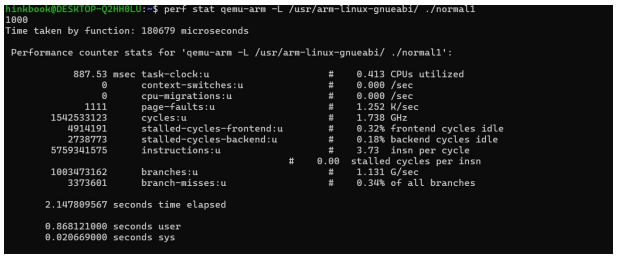
（四）Profiling

这里我们皆以 n=1000 的情况为例进行性能分析。本模块基于 perf，仅放出实验结果截图；而具体的分析在下一模块。**注：所有编译出来的文件名皆为 normal1，此处是进行了重新编译后的实验结果，并不代表二者是同一个程序。**

1. 平凡算法



2. Cache 优化算法



（五）结果分析

总体而言，cache 优化确实有帮助，但效果有限，优化效率基本在 1%量级，如下图。推测原因是编译器采用了较为先进的优化，使得朴素算法能够获得比预

期更高的效率。

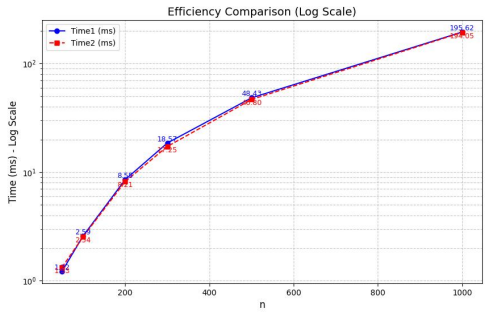


表 3 性能对比

1. 平凡算法

基于 perf，我们可以看到，在这次测试中耗时约 204 微秒，与表内数据接近。分支预测 3292160，占比 0.37%。

2. Cache 优化算法

基于 perf，本次测试耗时约 181 微秒，稍稍快于先前测试结果。推测原因是在进行大量计算之后，cache 命中率和分支预测带来的性能提升显著变强。分支 3373601，占比 0.34%，可以发现确实效率更高。

二．实验二：n 个数求和

（一）算法设计

1. 平凡算法设计思路

根据下标顺序遍历数组，逐个累加到 sum 中，十分暴力，不再赘述。

2. Cache 优化算法设计思路

基于类似归并或分治的思维，每次执行当前规模/2 大小的累加；再利用递归逐层加和。共执行 logn 级别层数。累加后仍然是 O（n）的算法。

（二）编程实现

1. 平凡算法

```
//朴素算法
for(int i=1;i<=n;++i)   sum+=a[i];
```

2. Cache 优化算法

```
for(int i=n;i>1;i>>=1)  for(int j=0;j<(i>>1);++j){
    a[j]=a[j*2]+a[j*2+1];
}
```

（三）性能测试

和上一个实验相似，我们采取多次测量取平均值的方式测试。进一步地，由于上一个是 n^2 复杂度，本次为 n 复杂度，于是为凸显差别，本次 n 的规模扩大为：10000，100000，1000000，2000000，5000000，10000000。

1. 平凡算法

表 4：测试结果

n	时间 1（ms）	时间 2（ms）	时间 3（ms）	时间 4（ms）	时间 5（ms）	平均（ms）
10000	2.874	2.913	2.777	2.900	2.790	2.8508
100000	3.268	3.206	3.152	3.290	3.966	3.3764
1000000	7.879	7.590	7.449	7.685	7.475	7.6156
2000000	10.586	10.270	11.123	11.526	11.267	10.9544
5000000	22.523	21.353	21.869	22.998	21.349	22.0184
10000000	39.948	41.863	41.982	41.024	39.741	40.9116

2. Cache 优化算法

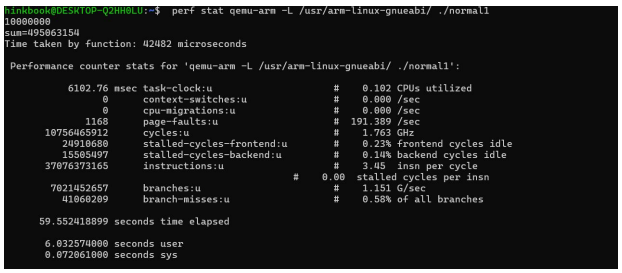
表 5：测试结果

n	时间 1（ms）	时间 2（ms）	时间 3（ms）	时间 4（ms）	时间 5（ms）	平均（ms）
10000	2.888	2.963	2.874	2.910	2.835	2.894
100000	3.627	3.542	3.534	3.474	3.643	3.564
1000000	9.131	8.660	8.339	8.938	8.903	8.7942
2000000	15.052	15.026	14.527	14.387	14.808	14.76
5000000	33.511	32.269	32.909	32.445	32.791	32.785
10000000	63.373	61.134	61.602	61.294	62.476	61.9758

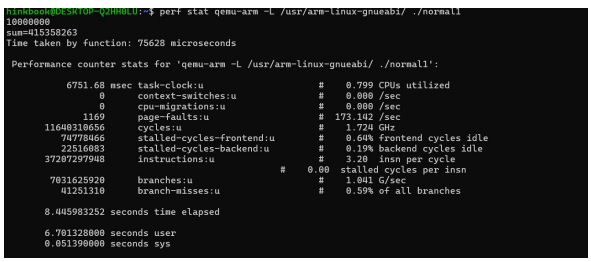
（四）Profiling

和上一个实验相同，这里仅采用 1e7 规模进行性能测试，以凸显差距。

1. 平凡算法



2. Cache 优化算法



（五）结果分析

在这里，我们惊奇地发现：Cache 优化居然比原先慢出了快一倍！笔者尝试了使用递归和循环两种方式，结果都差不多。于是，我们可以认为：要么耗时的计算方式有歧义，要么这种优化在一定程度上是成问题的。笔者将在下方试图解释。

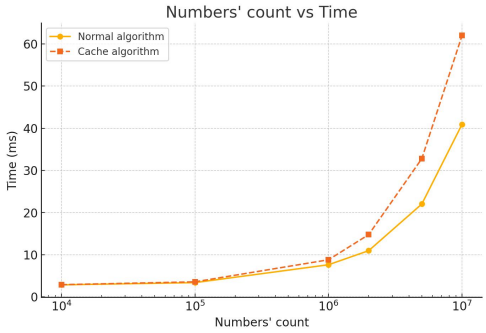


表 6 耗时分析

1. 平凡算法

基于 perf，本次主函数耗时 42 微秒，而总执行耗时 59.55 秒，出现显著差异。这可能是由于 CPU 使用效率过低导致。而实际的用户时间为 6.03 秒，问题不大。

2. Cache 优化算法

主函数耗时 76 微秒，总执行 8.44 秒，用户时间 6.70 秒，和前文结果基本一致：略慢于朴素算法。值得一提的是，该算法分支 miss 率为 0.59%，略高于前者的 0.58%。可能是这个原因。也有可能是随机数据恰巧使得该算法较劣。

三．实验总结与思考

Cache 优化往往可以利用存取特性小规模地加速程序运行；但有时并非进行 Cache 优化就会提升效率，还需要考虑到实际数据的特性。