南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计课程作业

体系结构相关编程

姓名：姚知言

年级：2022 级

专业：计算机科学与技术

指导教师：王刚

2024 年 3 月 24 日

摘要

在本次实验中，通过对 n*n 矩阵与向量内积和 n 个数求和两个实验，学习算法设计，编程实现，性能测试，profiling 的学习与应用，使用多种算法，多种实验环境进行实验，掌握了使用 vtune 和 godblot 进行 profiling 分析。根据实验结果和 profiling 结果展开对比分析。

关键字：编程；优化；cache；超标量；

目录

一、 实验一：n*n 矩阵与向量内积

(一) 算法设计

1. 整体思想

选用 chrono 库函数进行时间测量。由于该问题中运算量较小，对于矩阵大小较小的测试组，在单次实验中采取多次实验取平均值的思想，在操作中得到选取 1e8/(n*n) 的循环次数能够较好的平衡运算时间和准确性。将时间结果输出到 txt 中，以便后续使用。

2. 平凡算法

平凡算法采用逐列访问矩阵元素，一步外层循环（内层循环一次完整执行）计算出一个内积结果的方式。

3. cache 优化算法

cache 优化算法选用逐行访问矩阵元素，一步外层循环向每个内积累加一个乘法结果的方式。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，减少读取数据的次数。

4. unroll 循环展开优化

在 cache 优化算法的架构下，保证不改变 cache 优化算法的优势的前提下，进行循环展开优化，从而减少循环语句汇编指令的执行，提升运行速度。

(二) 编程实现

1. 核心框架

主函数

```
int main(){
    long long n,t;
    ofstream outfile("out.txt",ios::app);
    for(n=1;n<=5e4;n*=2){//通过控制n修改测试规模
        t=max(double(1),round(double(1e8)/(n*n)));
        cout<<"n="<<n<<"␣t="<<t<<'\n';
        pre(n);//为各数组分配内存并为矩阵和向量赋初值
        outfile<<n<<'␣'<<op(n,t)<<'\n';//算法函数
        fin(n);//释放各数组内存
    }
    outfile.close();
    return 0;
}
```

矩阵向量初始化

```
for(int num=0;num<n;num++){
    a[num]=num%100;
    for(int nnum=0;nnum<n;nnum++)b[num][nnum]=nnum%100+num%100;
}
```

注：矩阵存储于二维指针 b 指向的存储空间中，向量存储于指针 a 指向的存储空间中，内积结果存储于指针 sum 指向的存储空间中，均为 double 类型。其余分配释放内存等非核心代码不在此列出，可于 github 仓库中查看。

2. 平凡算法

平凡算法

```
1  double op(int n,int t){
2      int i,j;
3      auto start = chrono::steady_clock::now();
4      for(int k=0;k<t;k++){
5          for(i=0; i<n;i++){
6              sum[i]= 0.0;
7              for(j=0;j<n;j++)sum[i]+=b[j][i]*a[j];
8          }
9      }
10     auto end = chrono::steady_clock::now();
11     auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
12     double le=duration.count()*1.0/t;
13     cout << le<< "ms" << endl;
14     return le;
15 }
```

3. cache 优化算法

该算法对于时间计算函数无任何修改，仅对平凡算法中外层 for 循环内部代码算法（上图 line4-9）进行替换如下：

cache 优化算法

```
1  for(int k=0;k<t;k++){
2      for(i = 0; i < n; i++)sum[i] = 0.0;
3      for(j = 0; j < n; j++)
4          for(i = 0; i < n; i++)sum[i] += b[j][i]*a[j];
5  }
```

4. unroll 循环展开优化

该优化算法将上图代码替换如下，其他部分无变化。

循环展开优化

```
1  for(int k=0;k<t;k++){
2      for(i = 0; i < n; i+=2){sum[i] = 0.0;sum[i+1]=0.0;}
3      for(j = 0; j < n; j+=2){
4          for(i = 0; i < n; i+=2){
5              sum[i] +=b[j][i]*a[j];
6              sum[i+1]+=b[j][i+1]*a[j];
7          }
```

```
8           for(i = 0;i < n; i+=2){
9               sum[i]+=b[j+1][i]*a[j+1];
10              sum[i+1]+=b[j+1][i+1]*a[j+1];
11          }
12      }
13  }
```

(三) 性能测试

本实验采用本人物理机：x86-64 平台，windows 11 系统，TDM-GCC 编译器，Code::Blocks 集成开发环境，C++ 17 GNU，Optimize even more[-O2] 优化。

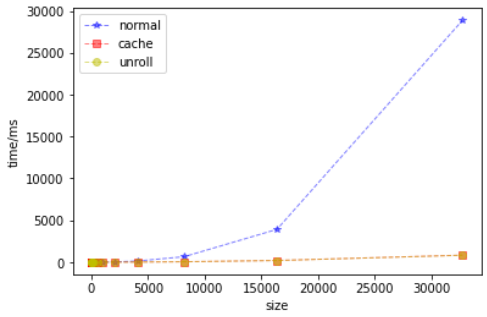
各组实验均在测试前已通过输出部分结果验证算法正确性。

由于调用的时间函数精度为 1ms，为降低性能测试误差，对每种算法进行 10 次独立重复实验，取平均值如下：

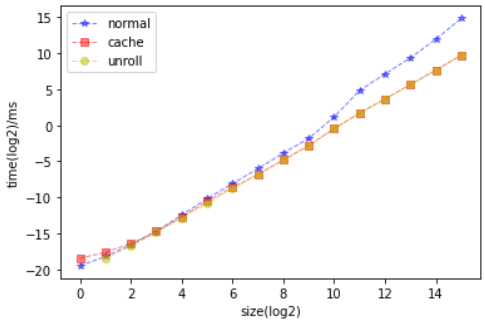
数据规模 (n)	1	2	4	8	16	32	64	128
单次测试次数	1.00e8	2.50e7	6.25e6	1.56e6	390625	97656	24414	6104
AVG 平凡/ms	1.38e-6	3.36e-6	1.03e-5	3.70e-5	1.80e-4	8.59e-4	3.66e-3	0.0155
AVGcache/ms	2.86e-6	5.01e-6	1.15e-5	3.72e-5	1.44e-4	6.78e-4	2.56e-3	8.88e-3
AVGunroll/ms		2.80e-6	9.02e-6	3.35e-5	1.40e-4	5.54e-4	2.30e-3	8.94e-3
数据规模 (n)	256	512	1024	2048	4096	8192	16384	32768
单次测试次数	1526	381	95	24	6	1	1	1
AVG 平凡/ms	0.0694	0.282	2.23	28.39	139.03	651.4	3916	28876
AVGcache/ms	0.0358	0.142	0.726	3.241	12.63	49.8	202.2	838.5
AVGunroll/ms	0.0357	0.143	0.729	3.15	12.18	48.7	197.5	823.8

表 1: 性能测试结果

注：在 unroll 优化中，由于是否考虑 n 为奇数的特殊判定对总体运行时间影响不大，故放弃考虑 n 为奇数的情况，n=1 数据缺省。



坐标图



对数坐标图

图 1: 性能测试坐标图

（四） profiling

1. VTune cache 命中率分析

在本部分，对以上三种算法分别在 n=32768 的情况下运行 1 次，使用 VTune Profiler 分析结果如下：

	平凡算法	cache 算法	unroll 算法
MEM_LOAD_RETIRED.L1_HIT	4,358,674,205	4,736,336,375	4,904,878,345
MEM_LOAD_RETIRED.L1_HIT	1,095,927,505	91,591,495	79,454,090
MEM_LOAD_RETIRED.L2_HIT	35,230,325	60,670,190	49,461,855
MEM_LOAD_RETIRED.L2_MISS	1,060,776,795	30,946,870	30,019,370
MEM_LOAD_RETIRED.L3_HIT	325,478,310	21,189,025	20,648,565
MEM_LOAD_RETIRED.L3_MISS	740,581,070	8,380,475	7,858,550
cache 命中率	61.96%	97.35%	97.70%

表 2: cache 命中率分析

2. 基于 Godblot 的汇编程序分析

考虑普通 cache 算法和 unroll 循环展开后优化算法的二重循环累加部分代码：

内层循环判定代码

```
1 add     DWORD PTR [rbp-4], 1
2 mov     eax, DWORD PTR [rbp-4]
3 cdqe
4 cmp     QWORD PTR [rbp-72], rax
5 jg      .L47
```

运算代码长度过长，故不在此列出。对于 cache 算法，内层循环判定代码（5 行）和运算代码（31 行）需要执行 n^2 次。对于 unroll 循环展开优化后的算法，内层循环判定代码（5 行）需要执行 $n^2/2$ 次，运算代码（31+32+34+35 行）需要执行 $n^2/4$ 次。

总的汇编语言数量从 $36n^2$ 优化为 $35.5n^2$。这是因为要保证 cache 优化算法的基本结构不变造成的，若使用此方法优化平凡算法，将获得更理想的优化比。

（五） 结果分析

性能测试的结果表明，在数据规模较小时，平凡算法和 unroll 算法稍优于 cache 算法，但差距并不大；在数据规模较大时，平凡算法的性能远低于 cache 算法，unroll 优化后的 cache 算法比原始 cache 算法有小幅度的优化，随数据集的增大而更加显著。

从 cache 命中率来看，cache 算法的 cache 命中率极高，接近 100%，与 unroll 循环展开优化差别不大。而平凡算法的 cache 命中率仅有不到 62%，这也能够很好的与二者性能测试结果相差极大对应。

从汇编语言分析中，我们可以看出 unroll 优化后的 cache 算法可以执行更少的汇编语言数量，但二者差距并不大，这也与性能测试中优化比并不是很大相对应。也印证了使用更少的汇编语言能够加快程序性能。

二、 实验二：n 个数求和

(一) 算法设计

1. 平凡算法

将元素依次累加到 sum 中。

2. 多链路式算法

将元素分组求和，分别累加，最后将部分和累加得到结果。
该方法将分别通过默认 2 链路和为更好适配 8 核处理器的 8 链路实现。

3. 两两合并 logn 算法

将元素两两相加，再将上一步的结果两两相加，以此类推 logn 步骤后得到结果。
该方法将分别通过递归和循环实现。

(二) 编程实现

1. 数据初始化及变量说明

整体架构与实验一类似，可前往 github 查看，以下对数据初始化方式和变量名称进行说明：

数据初始化

```
for(int i=0;i<n;i++)//a,b数组,sum使用float类型
    a[i]=1/float(i+1);//后续提到b数组是a数组的拷贝，结果为sum变量
```

2. 平凡算法

平凡算法

```
sum=0.0;
for(int i=0;i<n;i++)sum+=a[i];
```

3. 多链路式算法（2 链路）

多链路式算法（2 链路）

```
sum=0.0;
float sum1=0,sum2=0;
for(int i=0;i<n;i+=2){
    sum1+=a[i];sum2+=a[i+1];
}
sum=sum1+sum2;
```

4. 多链路式算法（8 链路）

多链路式算法（8 链路）

```
sum=0.0;float sump[8]={0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
for(int i=0;i<n;i+=8){
    sump[0]+=a[i];sump[1]+=a[i+1];sump[2]+=a[i+2];sump[3]+=a[i+3];
    sump[4]+=a[i+4];sump[5]+=a[i+5];sump[6]+=a[i+6];sump[7]+=a[i+7];
}
sum=sump[0]+sump[1]+sump[2]+sump[3]+sump[4]+sump[5]+sump[6]+sump[7];
```

5. 两两合并 logn 算法（递归）

两两合并 logn 算法（递归）

```
//主函数片段
sum=0.0;
for(int u=0;u<n;u++)b[u]=a[u];//为保证多次实验的重赋值操作，与算法无关，在后续性能测试中去除此部分消耗
recursion(n);
sum=b[0];
//递归函数
void recursion(int n){
    if(n==1) return;
    for(int i=0;i<n/2;i++) b[i]+=b[n-i-1];
    n=n/2+n%2;
    recursion(n);
}
```

6. 两两合并 logn 算法（循环）

两两合并 logn 算法（循环）

```
sum=0.0;
for(int u=0;u<n;u++)b[u]=a[u];//为保证多次实验的重赋值操作，与算法无关，在后续性能测试中去除此部分消耗
for(int m=n;m>1;m=m/2+m%2){
    for (int i=0;i<m/2;i++)b[i]=b[i*2]+b[i*2+1];
    if(m%2)b[m/2]=b[m-1];
}
sum=b[0];
```

(三) 性能测试

在本实验中，测试 n 的范围为从 10000 开始，每次增加 1000，直到 100000，共 91 个。对每个 n，采取运算 10000 次取平均值的方式计算。

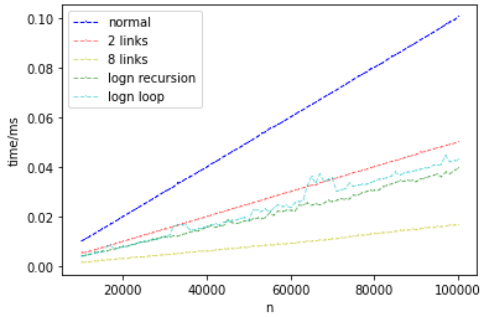
根据各算法之间互证结果，我们可以认为算法的正确性是可靠的。

1. 实验环境一：windows 物理机

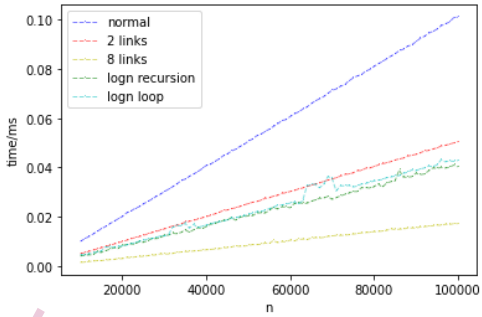
x86-64 平台，windows 11 系统，TDM-GCC 编译器，Code::Blocks 集成开发环境，C++ 17 GNU，Optimize even more[-O2] 优化。

2. 实验环境二：ubuntu 虚拟机

Ubuntu 虚拟机，g++ 编译器，O2 优化。



实验环境一



实验环境二

图 2: 性能测试坐标图

对于计算结果误差情况，即五种累加方式结果是否有差异，统计如下表。

n	[1e4,2e4)	[2e4,3e4)	[3e4,4e4)	[4e4,5e4)	[5e4,6e4)
测试样本数	10	10	10	10	10
误差样本数	4	3	2	8	9
误差率	40%	30%	20%	80%	90%
n	[6e4,7e4)	[7e4,8e4)	[8e4,9e4)	[9e4,1e5]	ALL
测试样本数	10	10	10	11	91
误差样本数	10	10	10	11	67
误差率	100%	100%	100%	100%	73.6%

表 3: 计算结果误差统计

(四) profiling

采用 VTune Profiler 进行超标量分析，结果如表4所示。(求 100000 个数之和，进行 10000 次重复实验)

(五) 结果分析

从性能测试结果中可以看出，各种超标量优化算法相比于平凡算法都有着较大的优化力度，且优化力度最大的是多链路式算法（8 链路），两两合并 logn 算法的循环实现和递归实现相差不大，但递归实现稍好于循环实现。

对于物理机和 ubuntu 虚拟机中的对比，我们可以看出二者的运行时间差距并不大。

算法名称	核心代码 CPI rate
平凡算法	0.893
多链路式算法（2 链路）	0.889
多链路式算法（8 链路）	0.559
两两合并 logn 算法（递归）	0.312
两两合并 logn 算法（循环）	0.398

表 4: 超标量分析结果

对于 logn 算法的循环算法在中间产生的运行时间波动，估计是由于其中连续多次二分不能整除，多执行了一些计算语句造成相较于附近数值结果偏大。

根据计算结果误差统计表，我们可以看出在浮点数精度较大时，计算次序确实会在一定程度上影响最终累加的结果，这一结果随着数据规模的增大和被加数和加数的精细程度差距而更容易产生误差。由于本题采用 float 存储，在后续计算规模足够大时，误差几乎已经成为了必然事件。

结合超标量分析的测量结果，我们可以看出相比于平凡算法，各种超标量优化算法的 CPI rate 均较低，其中 logn 级别的算法在超标量计算中展现出了极低的 CPI rate，至于为什么 logn 算法在性能测试中未能取得最好的成绩，我猜测可能是电脑线程数量有限，随着线程数量的增加，logn 算法的优势就将会快速体现出来。

三、 总结

在本次体系结构相关编程实验中，随着以上两个实验的进行，我熟悉了的“x86 平台、Windows 系统、TDM-GCC 编译器 +Code::Blocks 集成开发环境”，和“x86 平台，Windows 系统 +Ubuntu 虚拟机、GCC 编译器”两种开发环境的使用，熟悉了算法性能测试的基础知识和时间函数的调用。

完整实现了实验手册中的各种算法，并创新了 8 链路的多链路式方法，学习 unroll 循环展开相关知识并完成了相关优化算法的建立。

学习了利用 Vtune Profiler 进行 cache 命中率分析和超标量分析的实验方法，以及基于 Godblot 的汇编语言分析方法，对 profiling 有了进一步的认识。

对各类实验结果进行了自己的总结与分析，加强了对计算机体系结构的了解。

已将性能测试代码及结果上传至github 仓库中，供查看。