南开大学
Nankai University

南　开　大　学

计 算 机 学 院

并行程序设计实验报告

体系结构与性能测试实验报告

周辰霏 1712991

年级：2017 级

专业：计算机科学与技术

2020 年 3 月 12 日

目录

一、 概述

(一) 实验内容

针对以下两个问题:

1. 计算给定 n*n 矩阵的每一列与给定向量的内积, 分别实现逐列访问元素的平凡算法和 cache 优化算法两类算法设计思路
2. 计算 n 个数的和, 实现逐个累加的平凡算法 (链式) 以及超标量优化算法两类算法设计思路

完成以下实验内容:

1. 编程实现两种算法设计思路
2. 使用高精度计时工具测试程序执行时间, 比较平凡算法与优化算法的性能
3. 使用 vtune 进行更细致的 profiling, 分析结果与性能表现间的关系
4. 撰写实验报告

(二) 实验环境

- 系统环境: Windows10 家庭中文版 (64 位)
- 编译环境: gcc 8.1.0 x86_64-posit-seh, Codeblocks17.12
- 实验测试工具: Intel Vtune profiler 2020

二、 矩阵与向量内积问题

(一) 程序设计思路

由于该问题的侧重点是 cache 空间局部性的优化, 所以首先使用 init 函数每次重置矩阵和向量内容, 尽量保证 cache 初始对两种算法公平。

cache-init

```
1  void init(int n) // generate a N*N matrix && N vector
2  {
3      for(int i=0;i<N;i++)
4      {
5      for(int j=0;j<N;j++)
6          b[i][j]=i+j;
7      a[i]=i;
8      }
9  }
```

1. 逐列访问平凡算法

平凡算法保证是按列访问且每次初始化 cache 状态以及使用 Windows 平台上的 QueryPerformance 计时即可。

逐列访问平凡算法

```
void ord()
{
    double head,tail,freq,head1,tail1,timess=0; // timers
    init(N);
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq );
    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    for (int i=0; i<NN; i++)
        for (int j=0; j<NN; j++)
            col_sum[i] += (b[j][i]*a[j]);
    QueryPerformanceCounter ((LARGE_INTEGER *)& tail) ;
    cout << "\nordCol:" <<(tail-head)*1000.0 / freq<< "ms" << endl;
}
```

2. cache 优化算法

cache 优化算法实际上是根据 cache 的时空局部性原理将原先的按列访问改为按行访问，而每一次的外循环都不会产生一个最终的内积，而是为最终的内积结果加上一个过程中的乘法结果。

cache 优化逐行访问算法

```
        void ord()
        {
            double head,tail,freq,head1,tail1,timess=0; // timers
            init(N);
            QueryPerformanceFrequency((LARGE_INTEGER *)&freq );
            QueryPerformanceCounter((LARGE_INTEGER *)&head);
            for (int i=0; i<NN; i++)
                for(int j=0; j<NN; j++)
                    col_sum[j] += (b[i][j]*a[i]);
            QueryPerformanceCounter ((LARGE_INTEGER *)& tail) ;
            cout << "\noptCol:" <<(tail-head)*1000.0 / freq<< "ms" << endl;
        }
```

(二) 高精度计时性能比较

由于实验中可能出现种种误差，为尽量保证实验的公平性，作以下假设：

- Cache 对两种算法都是公平的，每次进行矩阵向量内积之前通过 init 函数刷一遍 cache
- 两种算法使用的矩阵向量数值是设定的固定值，且最终结果一致
- 由于 N 较小的时候测出的计算时间也较小，误差较大，故采取多次测量取均值的方法确定较合理的性能测试结果，同时保证两种算法重复次数一致，减少误差

N	重复测试次数	平凡算法单次时间均值 (ms)	cache 优化算法单次时间均值 (ms)
0	10000000	4.14933×10^{-5}	4.10364×10^{-5}
10	10000000	1.09826×10^{-4}	1.09363×10^{-4}
20	3000	5.344×10^{-4}	4.3243×10^{-4}
...	...	...	...
100	150	1.8068×10^{-2}	6.81×10^{-3}
200	150	6.54193×10^{-2}	2.3474×10^{-2}
...	...	...	...
1000	30	3.8185	0.6887
2000	30	60.5527	2.51124
5000	30	617.357	16.9051
10240	30	2670.4067	69.0433

表 1: 性能测试结果 (部分)

根据全部测试数据绘制如下图, 可以看出在 N 较小的时候, 平凡算法与优化算法的差别维持在较小的数量级, 而当 N>1000 时, 平凡算法在速度上开始明显慢于 cache 优化算法, 特别地, 当 N 取 10240 时, 优化算法相对于平凡算法的加速比达到了 52。

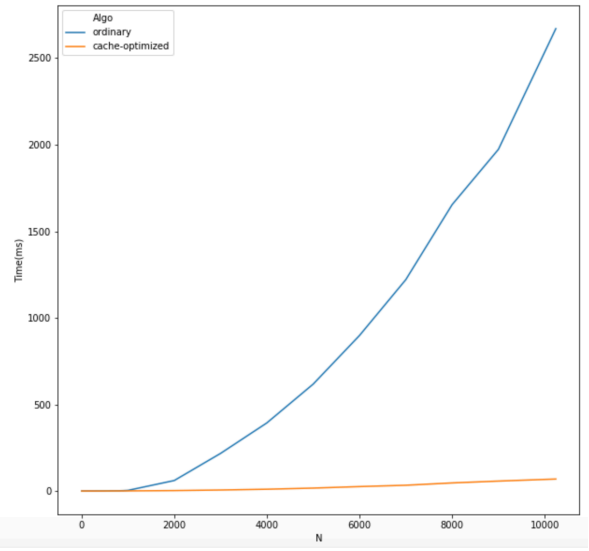

图 1: 性能测试对比图

通过上述的测试比较, 可以看出 cache 优化后的算法确实在性能上优于平凡算法, 这里的差距并不是因为计算次数的减少, 而是因为优化后的算法可以很好的发挥出 cache 的空间局部性, 使得访存时间大大减少, 因此 N 越大, 提升效果越显著。下图是借助 CPU-Z 软件得到的测试机自身的 cache 大小。按 int 型占 4 个字节粗略估算,32KB 的一级 cache 可以容纳 $N \approx 90$ 时矩阵大小的数据, 加上 256KB 的二级 cache 可以容纳 $N \approx 346$ 时矩阵大小的数据, 再加上 6MB 的三级 cache 则最终可以容纳 $N \approx 1254$ 时矩阵大小的数据。这与我们实验测试结果相符。确实是在一千左右优化算法明显和平凡算法拉开了差距。

图 2: 测试机 Cache 大小

为了验证 cache 优化算法确实在对 cache 的访存时间上做了性能提升, 做出细化的时间性能对比图如下:

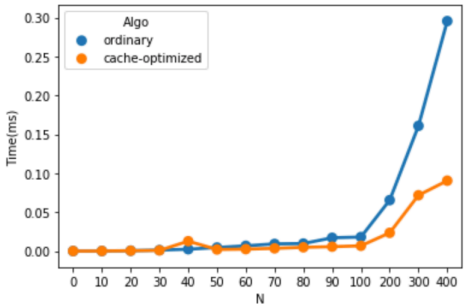

图 3: 一二级 cache

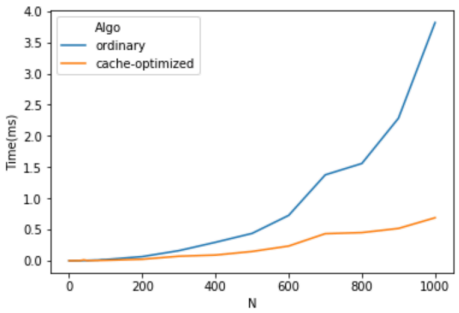

图 4: 一二三级 cache

确实看出拐点出现的 N 值在估算范围附近, 而根据体系结构计算机组成原理上学习到的知识, 访存速度的对比为: L1 cache 为 3 cycles,L2 cache 为 11 cycles,L3 cache 为 25 cycles, 主存则要 100 cycles。根据这种速度上的对比就可以理解为什么 cache 优化算法仅仅只是改变了访问数据的逻辑顺序就可以取得如此大的性能提升, 毕竟当 N 取一万的时候, 大多数时间都浪费在访问主存上了, 实际的运算次数二者并没有巨大的差别。但是这个访问主存的时间浪费仅仅只是基于理论和实验结果的合情推理, 所以接下来就要使用 profiling 工具来更细致的剖析结果与性能表现间的关系。

(三) 基于 Vtune 的程序性能剖析

按照实验指导书配置测试三级 cache 的命中缺失是否与理论预测相符。在 N=10000 的时候可以明显看出 cache 优化算法在命中率上远远高于平凡算法, 几乎都是一级 cache 命中, 而平凡算法三级 cache 缺失的数据接近三分之一, 可想而知会带来多大的性能损失。N=1000 的时候情况好了许多, 只有少部分一级 cache 缺失并且三级 cache 缺失也很少。而 N 取 500 的时候, 平凡算法最差也可以保证二级 cache 命中, 这也与预期相符合。

Elapsed Time: 3.060s

CPU Time: 2.971s
CPI Rate: 1.971
Total Thread Count: 4
Paused Time: 0s

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample	Precise
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	73,429,540	582	25003	
CPU_CLK_UNHALTED.REF_TSC	5,348,546,875	3,303	1600000	
CPU_CLK_UNHALTED.REF_XCLK	74,833,715	593	25003	
CPU_CLK_UNHALTED.THREAD	4,468,723,980	1,792	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	180,158,995	19	2000003	
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	2000003	
INST_RETIRED.ANY	2,267,507,821	912	1600000	
MEM_LOAD_RETIRED.L1_HIT	230,142,720	24	2000003	
MEM_LOAD_RETIRED.L1_MISS	102,628,625	206	100003	
MEM_LOAD_RETIRED.L2_HIT	500,145	2	100003	
MEM_LOAD_RETIRED.L2_MISS	102,001,995	408	50021	
MEM_LOAD_RETIRED.L3_HIT	3,758,665	16	50021	
MEM_LOAD_RETIRED.L3_MISS	95,509,025	192	100007	

图 5: 平凡 N=10000

Elapsed Time: 0.363s

CPU Time: 0.337s
CPI Rate: 0.569
Total Thread Count: 4
Paused Time: 0s

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample	Precise
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	8,788,605	71	25003	
CPU_CLK_UNHALTED.REF_TSC	606,273,675	383	1600000	
CPU_CLK_UNHALTED.REF_XCLK	10,425,315	84	25003	
CPU_CLK_UNHALTED.THREAD	1,245,846,937	431	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	610,012,815	62	2000003	
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	2000003	
INST_RETIRED.ANY	2,189,157,238	655	1600000	
MEM_LOAD_RETIRED.L1_HIT	360,188,675	37	2000003	
MEM_LOAD_RETIRED.L1_MISS	0	1	100003	
MEM_LOAD_RETIRED.L2_HIT	0	1	100003	
MEM_LOAD_RETIRED.L2_MISS	0	1	50021	
MEM_LOAD_RETIRED.L3_HIT	0	0	50021	
MEM_LOAD_RETIRED.L3_MISS	0	0	100007	
UOPS_EXECUTED.THREAD	3,200,817,095	319	2000003	
UOPS_EXECUTED.X87	0	0	2000003	

图 6: 优化 N=10000

Elapsed Time: 0.389s

CPU Time: 0.336s
CPI Rate: 0.800
Total Thread Count: 4
Paused Time: 0.001s

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample	Precise
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	7,516,525	574	2500	
CPU_CLK_UNHALTED.REF_TSC	604,429,600	3,564	160000	
CPU_CLK_UNHALTED.REF_XCLK	8,654,495	657	2500	
CPU_CLK_UNHALTED.THREAD	1,192,292,068	3,828	200000	
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	200000	
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	200000	
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	200000	
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	0	0	200000	
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	0	0	200000	
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	200000	
INST_RETIRED.ANY	1,489,484,935	4,490	160000	
MEM_LOAD_RETIRED.L1_HIT	142,508,870	142	200000	
MEM_LOAD_RETIRED.L1_MISS	1,468,530	30	10000	
MEM_LOAD_RETIRED.L2_HIT	757,855	16	10000	
MEM_LOAD_RETIRED.L2_MISS	710,055	29	5002	
MEM_LOAD_RETIRED.L3_HIT	377,340	16	5002	
MEM_LOAD_RETIRED.L3_MISS	250,015	6	10000	
UOPS_EXECUTED.THREAD	1,626,855,315	1,514	200000	

图 7: 平凡 N=1000

Elapsed Time: 0.295s

CPU Time: 0.276s
CPI Rate: 0.684
Total Thread Count: 4
Paused Time: 0s

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample	Precise
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	9,668,180	78	25003	
CPU_CLK_UNHALTED.REF_TSC	497,122,275	316	1600000	
CPU_CLK_UNHALTED.REF_XCLK	10,437,645	84	25003	
CPU_CLK_UNHALTED.THREAD	997,611,656	337	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	0	1	2000003	
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	2000003	
INST_RETIRED.ANY	1,458,729,217	446	1600000	
MEM_LOAD_RETIRED.L1_HIT	100,064,860	11	2000003	
MEM_LOAD_RETIRED.L1_MISS	0	1	100003	
MEM_LOAD_RETIRED.L2_HIT	0	0	100003	
MEM_LOAD_RETIRED.L2_MISS	0	1	50021	
MEM_LOAD_RETIRED.L3_HIT	0	0	50021	

图 8: 优化 N=1000

Elapsed Time: 0.372s

CPU Time: 0.346s
CPI Rate: 0.793
Total Thread Count: 4
Paused Time: 0s

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample	Precise
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	5,274,465	43	25003	
CPU_CLK_UNHALTED.REF_TSC	623,571,075	393	1600000	
CPU_CLK_UNHALTED.REF_XCLK	10,437,965	85	25003	
CPU_CLK_UNHALTED.THREAD	1,168,437,092	415	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	2000003	
INST_RETIRED.ANY	1,473,979,624	544	1600000	
MEM_LOAD_RETIRED.L1_HIT	170,109,185	18	2000003	
MEM_LOAD_RETIRED.L1_MISS	1,000,820	3	100003	
MEM_LOAD_RETIRED.L2_HIT	500,265	2	100003	
MEM_LOAD_RETIRED.L2_MISS	0	1	50021	
MEM_LOAD_RETIRED.L3_HIT	0	0	50021	
MEM_LOAD_RETIRED.L3_MISS	0	0	100007	
UOPS_EXECUTED.THREAD	2,013,988,915	201	2000003	

图 9: 平凡 N=500

Elapsed Time: 0.296s

CPU Time: 0.277s
CPI Rate: 0.701
Total Thread Count: 4
Paused Time: 0s

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample	Precise
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	8,790,880	71	25003	
CPU_CLK_UNHALTED.REF_TSC	498,777,900	315	1600000	
CPU_CLK_UNHALTED.REF_XCLK	10,306,660	83	25003	
CPU_CLK_UNHALTED.THREAD	1,026,038,691	338	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	0	0	2000003	
FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	0	0	2000003	
FP_ARITH_INST_RETIRED.SCALAR_SINGLE	0	0	2000003	
INST_RETIRED.ANY	1,463,216,400	451	1600000	
MEM_LOAD_RETIRED.L1_HIT	110,076,410	12	2000003	
MEM_LOAD_RETIRED.L1_MISS	0	1	100003	
MEM_LOAD_RETIRED.L2_HIT	0	0	100003	
MEM_LOAD_RETIRED.L2_MISS	0	1	50021	
MEM_LOAD_RETIRED.L3_HIT	0	0	50021	
MEM_LOAD_RETIRED.L3_MISS	0	0	100007	
UOPS_EXECUTED.THREAD	1,319,039,370	132	2000003	
UOPS_EXECUTED.X87	0	0	2000003	

图 10: 优化 N=500

再打开 Event Count 查看详细界面, 统计计数表明绝大多数的数据访问是在 main 函数非 init 部分的 (平凡算法、优化算法的主体位置), 这符合实际情况, 同时观察优化算法图可以发现数据访问在一段时间内是非常密集的, 而平凡算法图则可以看出, 基本上只要以及 cache 发生了缺失, 二级 cache 同时也会出现缺失, 同时由于平凡算法的按列访问,cache 的缺失是分时间段的多次缺失。

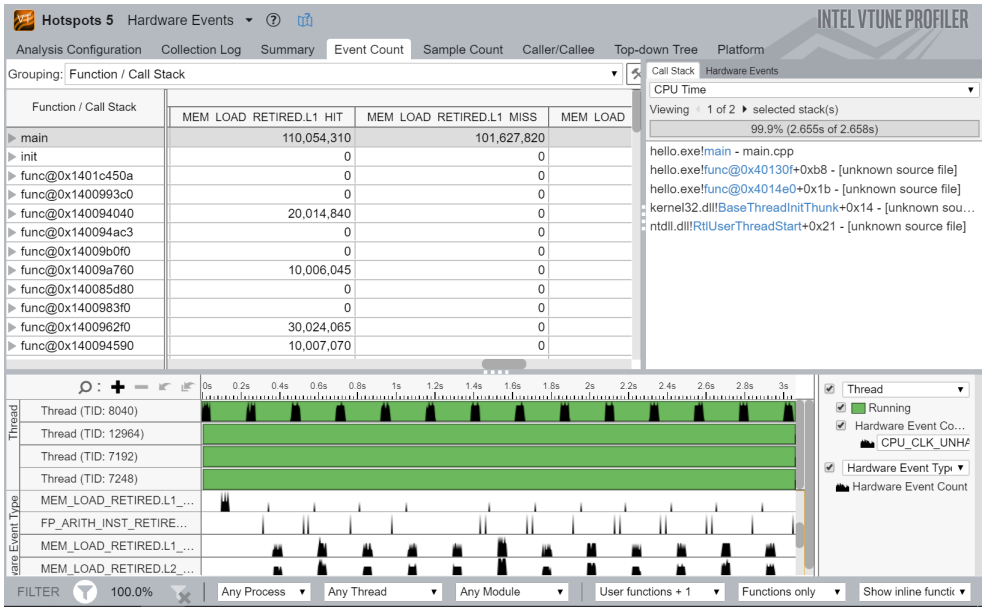

图 11: 平凡 10240

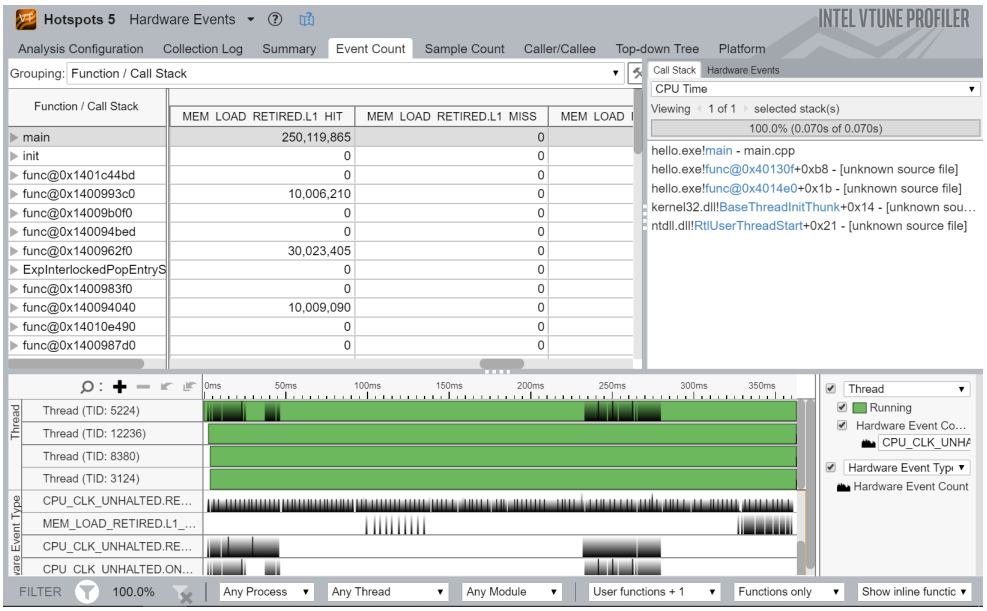

图 12: 优化 10240

三、 求和超标量优化问题

(一) 程序设计思路

根据实验指导书, 针对超标量优化求和实现了以下算法:

- 逐个累加链式串行平凡算法
- 多路链式累加算法
- 递归算法

- 循环展开算法
- 模板展开循环
- 宏展开循环

1. 串行链式平凡算法

serial

```
int serial_cal(int num[])      //ordinary
{
    int sum = 0;
    for(int i = 0;i<N;i++)
        sum += num[i];
    return sum;
}
```

2. 多路链式累加算法

多路是相对于串行单链而言的,测试了 2、4 路与单链对比,为了应用 CPU 的超标量超流水特性,所以将原先需要 n 次循环才能计算出的值分解为 n/2(二路)、n/4(四路) 次循环计算,减少了循环次数,同时多条求和的链可令多条流水线充分地并发运行指令

link

```
        int two_link_cal(int a[], int n)
        {
            int sum1=0,sum2=0;
            for(int i=0;i<n;i+=2){
                sum1+=a[i];sum2+=a[i+1];
            }
            return sum1+sum2;
        }
        int four_link_cal(int a[], int n)
        {
            int sum1=0,sum2=0,sum3=0,sum4=0;
            for(int i=0;i<n;i+=4){
                sum1+=a[i];sum2+=a[i+1];sum3+=a[i+2];sum4+=a[i+3];
            }
            return sum1+sum2+sum3+sum4;
        }
```

3. 递归算法

分成二路和四路的递归,递归的思路其实和链路的一致,都是为了降低循环对性能的影响,将循环进行了拆解

recursive

```
int recursive_cal(int num[], int starts, int ends)    //ord recursive
{
    if(starts == ends)  return num[ends];
    else{
        int mid = (starts+ends)/2;
        return recursive_cal(num, starts, mid)+
        recursive_cal(num, mid + 1, ends);
    }
}

int recursive_cal_unroll(int num[], int starts, int ends)    //opt recursive
{
    if(ends<starts) return 0;
    if(starts==ends) return num[starts];
    if(starts == ends-1)
        return num[starts]+num[ends];
    else
    {
        int mid = (ends+starts)/2;
        int qu=(starts+mid)/2;
        int qua=(mid+ends)/2;
        return recursive_cal_unroll(num,starts,qu)+
        recursive_cal_unroll(num,qu+1,mid)+
        recursive_cal_unroll(num,mid+1,qua)+
        recursive_cal_unroll(num,qua+1,ends);
    }
}
```

4. 循环展开优化算法

这个算法其实是对多链算法的另一种改写方式,即将计算结果直接保存在数组中,减少访存,应用的前提是必须要初始化数组至初始状态,虽然我本以为和问题一一样减少访存可以提升速率,但是由于这个访存的优化是借助指令的增多实现的,所以实际上并没有达到理想的好的效果。

opt-link

```
    int opt2_cal(int a[], int n)
    {
        while(n >= 2)
        {
            n /= 2;
            int j = 0;
            for(int i = 0; j<n;i += 2)
                a[j++] = a[i] + a[i+1];
        }
        return a[0] + a[1] ;
    }
    int opt4_cal(int a[], int n)
```

```
    {
        while(n >= 8)
        {
            n /= 2;
            int j = 0;
            for(int i = 0; j<n;i += 8)
            {
                a[j++] = a[i] + a[i+1];
                a[j++] = a[i+2] + a[i+3];
                a[j++] = a[i+4] + a[i+5];
                a[j++] = a[i+6] + a[i+7];
            }
        }
        return a[0] + a[1] + a[2] + a[3];
    }
```

5. 模板展开算法

模板技术实际上是对循环展开以达到性能提升的目的,但是由于迭代次数过多可能在 N 较大的时候编译器可能无法正常处理,故模板展开在数据较大时的意义不太大

template

```
int sumt=0;        //template
template<int N>
class Loop{
public:
    static inline int Run(){
        int v=Loop <N-1>::Run();
        sumt+=v;
        return v+1;
    }
};
template<>
class Loop<0>{
public:
    static inline int Run(){return 0;}
};
```

6. 宏展开算法

宏展开实际上是借助宏本身在预处理阶段的直接替换进行展开处理,实际上是将循环彻底消除了。

macro

```
#define DO_THING x;
#define DO_THING_2 DO_THING; DO_THING
#define DO_THING_4 DO_THING_2; DO_THING_2
```

```
#define DO_THING_8 DO_THING_4; DO_THING_4
#define DO_THING_16 DO_THING_8; DO_THING_8
// 调用时手动叠加更改数字即可如N=256
DO_THING_16(DO_THING_16(sumd+num[i]; i++))
```

7. main 函数运行部分

基本与问题一一致, 以宏调用为例

main

```
int main()
{
    double freq,head,tail;
    int num[N];
    for(int i = 0;i<N;i++)
    {
        num[i] = i;
    }
    memset(num,0,N);
    for(int i = 0;i<N;i++)
    {
        num[i] = i;
    }
     QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
     QueryPerformanceCounter((LARGE_INTEGER *)&head);
     int sumd=0,i=0;
     DO_THING_16(DO_THING_16(sumd+num[i]; i++))
     QueryPerformanceCounter((LARGE_INTEGER *)&tail);
     cout << "Macro calculation results: " << sumd <<endl;
     cout << "Macro calculation costs: "
     << (tail - head) * 1000.0 / freq << "ms" << endl;
}
```

(二) 高精度计时性能比较

同问题一, 由于实验中可能出现种种误差, 为尽量保证实验的公平性, 作以下假设:

- Cache 对两种算法都是公平的, 每次进行求和之前通过 memset 重置数组 (求和数据来源), 保证 cache 的公平以及消除个别算法会改变数组值的影响
- 每种算法都使用设定固定值相加, 且最终结果一致, 确保算法正确性
- 由于 N 较小的时候测出的计算时间也较小, 误差较大, 故采取多次测量取均值的方法确定较合理的性能测试结果, 同时保证算法重复次数一致, 减少误差

根据全部测试数据绘制如下图, 可以看出在 N 较小 (N<1000) 的时候, 模板就基本失效无法正常编译成功了, 而且按照模板是展开循环来看模板应该和宏一样处于一个较快的地位才对, 但是并没有, 由于模板在数据量较大的时候失效, 且其展开时间并没有比循环快, 所以姑且认为模板的

N	重复测试次数	平凡算法 (ms)	二路递归 (ms)	四路递归 (ms)	二路链式 (ms)
2	30	0.00001	0.00001	0.00001	0.00001
4	30	0.0001	0.0001	0.0001	0.00001
8	30	0.0001	0.0001	0.0001	0.00001
16	30	0.0001	0.00015	0.00015	0.00001
32	30	0.0001	0.0003	0.00025	0.0001
64	30	0.0001	0.00053	0.0004	0.0001
128	30	0.0003	0.0015	0.0007	0.0002
256	30	0.0005	0.0023	0.0013	0.0003
512	30	0.0011	0.0097	0.0023	0.0008
1024	30	0.0019	0.0071	0.0052	0.0014
2048	30	0.0041	0.0143	0.0064	0.0026
4096	30	0.0082	0.024	0.0133	0.0052
8192	30	0.0183	0.041	0.023	0.0102
16384	30	0.0368	0.097	0.0862	0.0206

表 2: 性能测试结果 (I)

N	重复测试次数	四路链式 (ms)	二链改进 (ms)	四链改进 (ms)	模板 (ms)	宏 (ms)
2	30	0.00001	0.00001	0.00001	0.00001	0.00001
4	30	0.00001	0.00001	0.00001	0.00001	0.00001
8	30	0.00001	0.00001	0.00001	0.00001	0.00001
16	30	0.00001	0.00001	0.00001	0.0001	0.00001
32	30	0.0001	0.00014	0.0001	0.0001	0.00001
64	30	0.0001	0.0001	0.0001	0.0001	0.00001
128	30	0.0001	0.00023	0.00014	0.0001	0.00001
256	30	0.0003	0.0007	0.0006	0.0001	0.00001
512	30	0.0005	0.0013	0.001	0.0001	0.0001
1024	30	0.0009	0.0023	0.0017	\	0.002
2048	30	0.0017	0.0047	0.0037	\	0.0037
4096	30	0.0034	0.0092	0.0075	\	0.0067
8192	30	0.0071	0.018	0.0137	\	0.0133
16384	30	0.0135	0.0366	0.0324	\	0.0288

表 3: 性能测试结果 (II)

消除循环没有特别大的意义。此外，递归的解决方法，即使是四路的递归也从一开始就比平凡串行算法要慢。只有链路一直表现良好，达到了很好的与串行平凡算法相比的性能提升效果，但是做了空间改进的多链算法并不如原来的表现好，原因可能是计算更耗时，或者其实这个程序的测试数据都在 cache 中不需要多此一举的优化。宏也是一直表现较串行好，但是宏是在预编译阶段处理，所以一旦 N 变得较大的时候，编译时间也会变长，所以整体下来还是最简单思路的多路算法表现最令人满意。

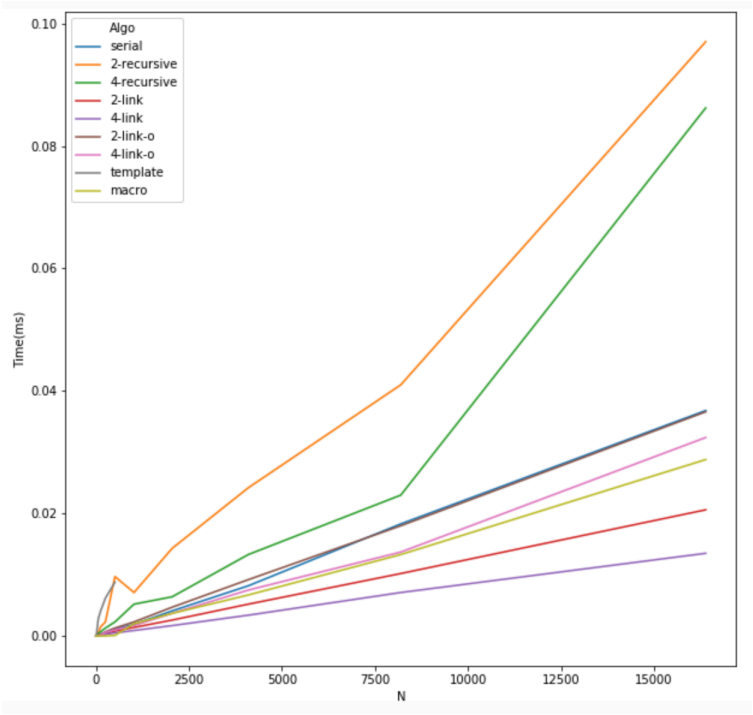

图 13: 性能测试对比图

（三） 基于 Vtune 的程序性能剖析

由于这个求和程序即便是最慢的递归在 N=16384 的时候时间也不大，所以 Vtune 默认的 CPU interval 的 1ms 显然有点大，容易测不出具体的每一个函数，所以需要事先将间隔调整为 10^{-2} 数量级。选取各算法趋势比较明了的 N=4096 进行剖析。

Microarchitecture Exploration 5 Microarchitecture Exploration

Analysis Configuration | Collection Log | Summary | Bottom-up | Top-down Tree | Event Count | Platform

Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiri
main	0.167ms	533,599	366,406	1.456	
two_link_cal	0.013ms	47,072	53,236	0.884	
opt2_cal	0.014ms	83,565	108,988	0.767	
opt4_cal	0.014ms	83,550	100,824	0.829	
recursive_cal_unroll	0.053ms	196,426	230,606	0.852	
recursive_cal	0.037ms	136,881	234,730	0.583	
serial_cal	0.012ms	46,825	37,622	1.245	

图 14: Vtune-CPI 比对

如上图所示, 根据 Vtune 的测试结果, 单看 CPI 结果, 所有算法的 CPI 都是优于串行链式算法的, 可见超标量优化有一定效果, 但是如递归算法在第二步高精度计时时效果却远远差于串行平凡算法, 比照执行指令数可以发现, 除了链路算法以外尤其是递归算法, 执行的指令数都远远高于串行平凡算法, 而时钟周期也是同样的趋势。如果单从 CPI 每条指令执行的周期数来看, 每种算法对于 CPU 的利用性能上都有着或多或少的提升, 但是从宏观的时间的角度来看, 整个实验过程实际上只有多链路算法达到了时间性能提升的要求。但是由于课程重点在并行, 且实验内容要求是超标量优化, 就姑且认为以上几种算法都可以达成要求。

四、 总结

这次实验属于一开始上手很容易, 但是中间会遇到各式各样的小问题。就比如问题一直到开始 Vtune 的时候才发现要注意 cache 的公平; 问题二的模板在同一段代码不同时间的执行结果不一致导致测出时间不可信、没有统一使用数据作为求和数据源而是有的使用立即数导致结果不准确、串行比链路算法以外的都快很多, 等等一系列或是无心之失或是不明所以的棘手问题, 好在最后还是基本都解决并算圆满完成了实验, 对并行的认知也从一开始的多核、多线程进展到了微体系结构与流水线的设计、cache 的设计相关的层次。