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

体系结构相关及性能测试

2013154 段辰睿

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2023 年 3 月 12 日

摘要

针对给定问题，实验测试了 windows-x86 优化算法及平凡算法的性能，进行比较并给出了相应的解释；此外，实验时还探索了不同编译优化力度的不同优化效果、unroll 前后程序性能比较、不同路数的多路链路算法性能、采用模板技术完全消除循环优化效果、不同系统平台对结果的影响等。

关键字：cache 优化、超标量优化、性能测试

目录

一、 n*n 矩阵与向量内积

(一) 实验平台配置

1. **系统环境**
   Windows 10 + TDM-GCC 编译器 + Code::Blocks + vtune

2. **硬件环境**
   lntel(R)Core(TM) i7-9750H CPU @ 2.60GHz

(二) 程序设计

1. 平凡算法

C++ 对于矩阵内元素按行主存储方式进行存储，若采用平凡的按列遍历矩阵的方法，优先计算每一列与给定向量的内积，则不符合 cache 的局部性原理，没有发挥最大的性能。:

2. cache 优化算法

若想尽可能的发挥出 cache 的功能，则对于矩阵每一行的遍历，我们计算出每一个内积结果的一部分，待行遍历完之后即可得到所有的内积结果。

(三) 实验方案

1. **生成实验数据**
   对于矩阵，我们简单采用；对于向量，我们简单采用

2. **进行实验**
   首先初始化结果数组，执行平凡算法并计时，再次初始化结果数组，执行优化算法并计时。为了使结果更有说服力，采用多次测量求平均值的方式来减少或消除部分误差。

3. **利用 vtune 对编译得到的程序进行 profiling**
   分析两种算法在 CPU 的三级 cache 上的命中与缺失次数。

4. **分析实验数据**
   得出实验结论。

(四) 实验结果及分析

实验时，采用 n=4096.count=50 进行实验。首先初始化结果数组，执行平凡算法并计时，再次初始化结果数组，执行优化算法并计时。为了使结果更有说服力，采用多次测量求平均值的方式来减少或消除部分误差

1. 运行结果

平凡算法：99.6486ms

cache 优化算法：11.1517ms

由结果可知，采用 cache 优化算法所需的时间比平凡算法所需的时间减少了近十分之一，性能得到了显著的提升。因为优化算法比起平凡算法的访存模式具有更好的空间局部性，使得 cache 的能力得以发挥。由于数组数据在计算机内部是按照行主序存储的，故以 column-row 形式访问

地址时会大量出现 Cache 丢失的情况，而 row-column 访问形式由于是顺着存储顺序访问，Cache 丢失情况较少，命中率更高。查看平凡算法的反汇编代码也发现，Cache 丢失最多的地方在于寄存器寻址并赋值。而对比优化前后的反汇编代码，可以看出 ldrsw 寻址效率得到了大幅优化。

利用 vtune 对编译得到的程序进行 profiling，分析两种算法在 CPU 的三级 cache 上的命中与缺失次数。得到如下表的实验结果。由图可得到，优化算法的 L1cache 的命中次数是平凡算法

$Cost$		命中率、缺失率					
		$L1Hit$	$L1Miss$	$L2Hit$	$L2Miss$	$L3Hit$	$L3Miss$
算法	$ordinaryCal$	852759656	830808358	3316106	827500148	788011738	26888930
	$optimizedCal$	2437447322	6820128	5423952	1397288	243156	1208472

表 1: vtune 的分析结果

L1cache 的命中次数的三倍且后者缺失次数是前者缺失次数的百余倍，L2cache 亦是如此。上述两结果可以明显看出，优化算法减少了访问 L2、L3 级 cache 的次数，更好的利用了 cache 的空间局部性原理，从而大幅提高了程序的执行速度。

（五） 实验思考（进阶要求）

上述实验均进行了 O2 优化，下面考虑下不优化的情况和采用 O1、O2 优化的情况。实验时，仍采用 n=4096.count=50 进行实验。

$performance/ms$		优化方法			
		–	$O1$	$O2$	$O3$
算法	$ordinaryCal$	104.851	101.658	99.6486	5.0465
	$optimizedCal$	40.3034	12.2315	11.1517	5.22117

表 2: 实验结果

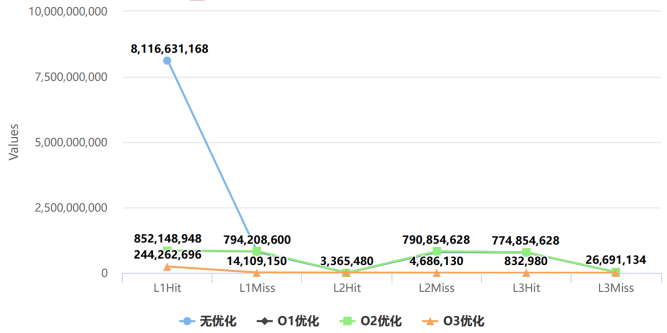

图 1: 平凡算法 vtune 分析结果

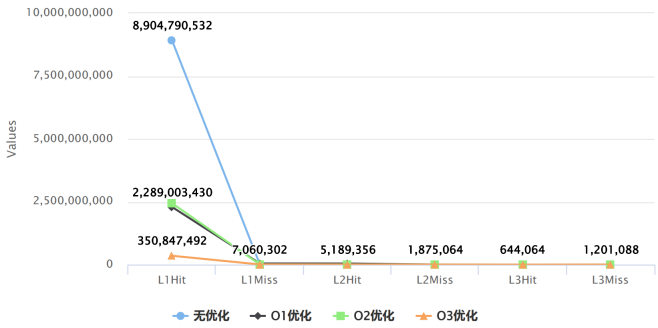

图 2: cache 优化算法 vtune 分析结果

由上图可以看出，在运行时间上，O1、O2 优化对于平凡算法的优化并没有显著提升，但对于优化算法则大幅提升性能，但二者提升幅度并无显著差异。

对于 L1cache 的命中/缺失次数，可以看出 O1、O2 优化对于平凡算法的命中次数都大幅减小，优化算法的命中次数也减小了，幅度略小于平凡算法。而对于 L2cache，值得注意的是，O1 优化下优化算法的命中次数大幅提升。同样对于 L3cache，O2 优化下 L3cache 的命中次数大幅减小。对于 O3 优化，无论是在运行时间上还是 cache 的命中/缺失次数上，性能均得到了大幅度的提升，并且两个算法的时间差别并不大了，多次实验可以发现二者均有可能在时间上领先于对方。

在命中次数上，而这主要在 L1cache 的缺失次数和 L2chche 的命中次数上有较大差异。

对于 O3 优化，我猜测编译器调整了运算的顺序，使得平凡算法在编译器的优化后也可以较好的利用 cache 的空间局部性，从而达到与优化算法近似的性能。

查询 GCC 官方文档得知，无优化情况下，编译器目标是降低编译成本并使调试产生预期的结果，此时语句相互独立，在调试时可以任意修改变量或者 PC 的值并得到对应的正确结果。而启用优化则会使得编译器牺牲编译时间和调试程序的能力来提升性能和代码大小。对于 O1 优化，编译器优化代码的大小和执行的速度，但会花费更多时间，但不会执行任何占用大量变异时间的优化。对于 O2 优化，会消耗更长的编译时间。O2 优化会对内联函数等进行更多优化，提高程序的执行效率，但会对程序执行逻辑有较多修改。O3 优化会开启更多对循环展开的优化，会利用现代 CPU 中的流水线和 Cache 等技术，采取向量化算法提高代码并行执行程度，从而提高代码的执行效率。

由此可以看出之前的猜测是正确的，O3 优化对于平凡算法的优化确实采用了跟优化算法相同的思想，但在此基础上比 O1、O2 优化提升了更多的效率。

二、 n 个数求和

(一) 实验平台配置

1. **系统环境**

   Windows 10 + TDM-GCC 编译器 + Code::Blocks + vtune

2. **硬件环境**

   lntel(R)Core(TM) i7-9750H CPU @ 2.60GHz

(二) 程序设计

1. 平凡算法

若采用逐个累加的平凡算法，则只需采用一个循环语句，循环内每次加和即可。

2. 超标量优化算法

若采取超标量优化，则有多种具体方法可以选择。大致都是减少由于循环语句带来的额外开销，采用循环展开策略，将多个循环步的操作展开到一个循环步。

1. **多链式的累加**
   每次循环加和若干次，最终再把和相加得到所有数的和

2. **递归算法**
   将给定元素两两相加，再将上一步得到的结果两两相加，重复步骤直至得到最终结果。可以使用递归函数来实现，但由于递归函数的调用会带来巨大的开销。也可以采用二重循环实现，外层循环控制轮次，内层循环控制每轮处理的元素个数。

   注意递归算法会产生大量的中间结果，需要把中间结果保存到连续的内存区域，从而利用 cache 的空间局部性来加快运行速度

3. **二重循环**

(三) 实验方案

1. **生成实验数据**
   同上一个实验一样，实验数据由人为生成固定值，实验中简单采用 num[i]=i 。数组个数采用 2 的幂次方便递归。

2. **进行实验**
   首先初始化结果和中间变量数组，再按顺序执行一个算法，记录时间，重复以上操作直至所有算法执行完毕。为了使结果更有说服力，采用多次测量求平均值的方式来减少或消除部分误差。但对于递归算法无法在函数内多次测量，而在 main 函数内多次测量计时又会计入初始化的时间，因此只进行一次测量。

3. **利用 vtune 对编译得到的程序进行 profiling**
   分析两类算法的 IPC，即每个时钟周期执行的指令数，从而可以直观的比较出两类算法的区别。

(四) 实验结果及分析

1. 运行结果

实验时采用 n=4096 ， count=50 ， num[i]=i

分析上述实验结果可以看出，从线性累加到二路/四路累加，超标量技术的应用对于运算速度的提升是较多路链路相当于指令级并发，能更好地利用 CPU 超标量架构，两条求和的链可令两条流水线充分地并发运行指令，这种优化是十分显著，其 IPC 也高于平凡算法。但实验时也发现，虽然 2 路链路的运行时间优于其他两种优化算法，但 IPC 却低于二者。

result		算法					
		serial	*twochain*	*fourchain*	*recursivefunction*	*recursivefunctionunroll*	*doubleloop*
性能	*cost/ms*	0.001316	0.00104	0.000638	0.0033	0.0028	0.0027
	ans	8386560	8386560	8386560	8386560	8386560	8386560

表 3: 实验结果

究其原因，应该是因为此处递归与双重循环虽然运行时较平凡算法与 2 路链路的周期数更大，几乎是后者的 40 倍；但其执行的指令数也更多，均在千亿级。猜测这是因为双重循环与递归的代码运算过程更复杂，取值、译码、执行的操作能够更紧密地填充在一个周期内，故而 IPC 更高。但由于没有充分发挥并发优势，故在运行时间上还是比不过多路链式算法。

正如我们之前分析的那样，递归函数算法会造成较大的函数调用开销，使得运算速度不如线性累加，但在递归函数算法中采用循环展开策略也可以提升一些运算的速度，但提升的速度并不明显，这也再次反映出了递归函数调用所带来的巨大开销。

采用二重循环来进行递归算法也并没有达到理想的预期效果，可能是由于递归算法循环的总次数更多从而在循环上消耗了大量时间。

2. vtune 辅助分析

调整 n=500 ，利用 vtune 进行分析，得到如下图结果。可以看到，若采用线性平凡算法，

算法	serial	two chain	four chain	recursive function	recursive function unroll	double loop
retired	7800000	5200000	52000008	5200000	2600000	2600000
clockticks	2600000	2600000	2600000	2600000	2600000	2600000

表 4: vtune 分析

执行指令数为 7,800,000，而执行的周期数为 2,600,000；而若采用多链式算法，则指令数减少为 5,200,000，周期数未改变。采用多链式算法的 IPC 明显优于线性平凡算法的 IPC，也与我们之前分析的结果一致。

(五) 实验思考（进阶）

1. 采用模板技术完全消除循环展开

从编程范式上来说，C++ 模板元编程是函数式编程，用递归形式实现循环结构的功能，用 C++ 模板的特例化提供了条件判断能力，这两点使得其具有和普通语言一样通用的能力（图灵完备性）。利用模版技术可以在编译阶段完全展开循环，从而减少循环对性能的影响。

Listing:

```
template <int N>
class Loop {
 public:
  static inline void Run() {
    sum += num[n - N - 1];
    Loop<N - 1>::Run();
    return;
```

```
 }
};
template <>
class Loop<0> {
 public:
  static inline void Run() {
    sum += num[n - 1];
    return;
 }
};
```

与之前的结果相比较，采用模版技术进行循环展开得到的程序在时间上是优于采用递归算法的，但仍差于平凡算法与多链式算法。原因在于编译器也会对循环做一定的循环展开，若在平凡算法的循环处使用关键字 volatile 限定，便可以禁止编译器做循环展开，测得时间为 0.005144ms ，说明运用模版技术做循环展开还是可以提升运算速度的

2. 系统平台对性能结果的影响

换用 linux 系统再次运行上面的程序，得到如下的结果 如图3所示

result		算法					
		serial	*twochain*	*fourchain*	*recursivefunction*	*recursivefunctionunroll*	*doubleloop*
性能	*cost/ms*	0.001644	0.001092	0.000566	0.004	0.003	0.002
	ans	8386560	8386560	8386560	8386560	8386560	8386560

表 5: 实验结果

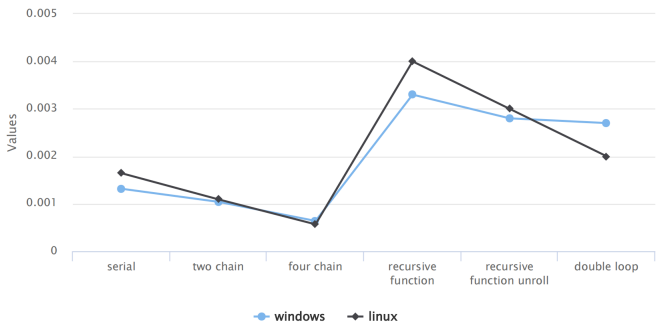

图 3: Owindow 与 linux 系统下的对比

可以看出与在 windows 系统下的性能并无较大差异。

三、 总结

本次实验针对两个问题，均证明了优化算法较平凡算法有更好的性能。并且在一定程度上还对其他相关问题进行了研究与分析。

源码链接：https://github.com/dcr0903/parallel