浅谈程序优化

2015/03/27 • 开发 • 553 阅读 • 优化, 软件开发

原文出处： 过客冲冲的博客 欢迎分享原创到伯乐头条

当初在学校实验室的时候，常常写一个算法，让程序跑着四处去晃荡一下回来，结果也就出来了。可工作后，算法效率似乎重要多了，毕竟得真枪实弹放到产品中，卖给客户的；很多时候，还要搞到嵌入式设备里实时地跑，这么一来真是压力山大了~~~。这期间，对于程序优化也算略知皮毛，下面就针对这个问题讲讲。

首先说明一下，这里说的程序优化是指程序效率的优化。一般来说，程序优化主要是以下三个步骤：

1. 算法优化

2. 代码优化

3. 指令优化

算法优化

算法上的优化是必须首要考虑的，也是最重要的一步。一般我们需要分析算法的时间复杂度，即处理时间与输入数据规模的一个量级关系，一个优秀的算法可以将算法复杂度降低若干量级，那么同样的实现，其平均耗时一般会比其他复杂度高的算法少（这里不代表任意输入都更

快）。

比如说排序算法，快速排序的时间复杂度为O(nlogn)，而插入排序的时间复杂度为O(n*n)，那么在统计意义下，快速排序会比插入排序快，而且随着输入序列长度n的增加，两者耗时相差会越来越大。但是，假如输入数据本身就已经是升序(或降序)，那么实际运行下来，快速排序会更慢。

因此，实现同样的功能，优先选择时间复杂度低的算法。比如对图像进行二维可分的高斯卷积，图像尺寸为MxN，卷积核尺寸为PxQ，那么

直接按卷积的定义计算，时间复杂度为O(MNPQ)

如果使用2个一维卷积计算，则时间复杂度为O(MN(P+Q))

使用2个一位卷积+FFT来实现，时间复杂度为O(MNlogMN)

如果采用高斯滤波的递归实现，时间复杂度为O(MN)（参见paper：Recursive implementation of the Gaussian filter，源码在GIMP中有）

很显然，上面4种算法的效率是逐步提高的。一般情况下，自然会选择最后一种来实现。

还有一种情况，算法本身比较复杂，其时间复杂度难以降低，而其效率又不满足要求。这个时候就需要自己好好地理解算法，做些修改了。一种是保持算法效果来提升效率，另一种是舍弃部分效果来换取一定的效率，具体做法得根据实际情况操作。

代码优化

代码优化一般需要与算法优化同步进行，代码优化主要是涉及到具体的编码技巧。同样的算法与功能，不同的写法也可能让程序效率差异巨大。一般而言，代码优化主要是针对循环结构进行分析处理，目前想到的几条原则是：

a.避免循环内部的乘(除)法以及冗余计算

这一原则是能把运算放在循环外的尽量提出去放在外部，循环内部不必要的乘除法可使用加法来替代等。如下面的例子，灰度图像数据存在BYTE Img[MxN]的一个数组中，对其子块 (R1至R2行，C1到C2列)像素灰度求和，简单粗暴的写法是：

```
int sum = 0;
for(int i = R1; i < R2; i++)
{
    for(int j = C1; j < C2; j++)
    {
        sum += Image[i * N + j];
    }
}
```

但另一种写法：

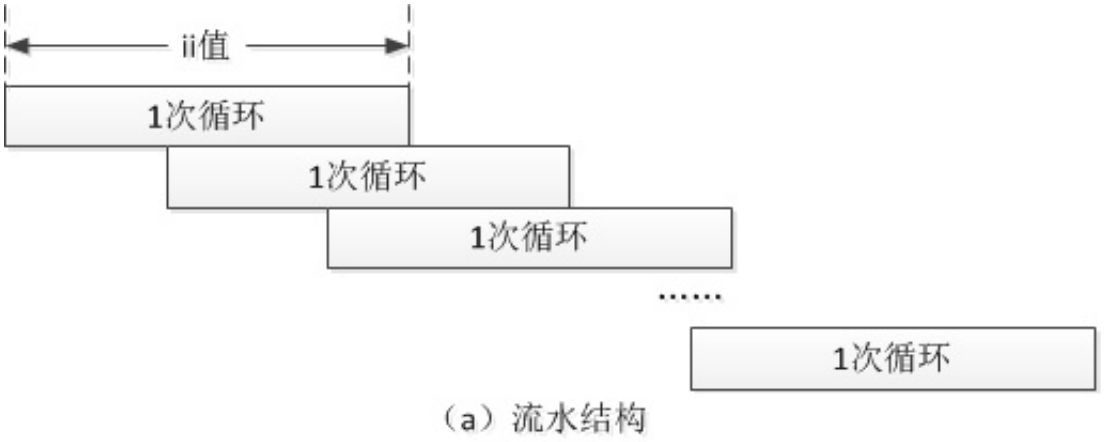
```
int sum = 0;
BYTE *pTemp = Image + R1 * N;
for(int i = R1; i < R2; i++, pTemp += N)
{
    for(int j = C1; j < C2; j++)
    {
        sum += pTemp[j];
    }
}
```

可以分析一下两种写法的运算次数，假设R=R2-R1，C=C2-C1，前面一种写法i++执行了R次，j++和sum+=…这句执行了RC次，则总执行次数为3RC+R次加法，RC次乘法；同　　样地可以分析后面一种写法执行了2RC+2R+1次加法，1次乘法。性能孰好孰坏显然可知。

b.避免循环内部有过多依赖和跳转，使cpu能流水起来

关于CPU流水线技术可google/baidu，循环结构内部计算或逻辑过于复杂，将导致cpu不能流水，那这个循环就相当于拆成了n段重复代码的效率。

另外ii值是衡量循环结构的一个重要指标，ii值是指执行完1次循环所需的指令数，ii值越小，程序执行耗时越短。下图是关于cpu流水的简单示意图：



（a）流水结构

（b）顺序执行结构

简单而不严谨地说，cpu流水技术可以使得循环在一定程度上并行，即上次循环未完成时即可处理本次循环，这样总耗时自然也会降低。

先看下面一段代码：

```
for(int i = 0; i < N; i++)
{
    if(i < 100) a[i] += 5;
    else if(i < 200) a[i] += 10;
    else a[i] += 20;
}
```

这段代码实现的功能很简单，对数组a的不同元素累加一个不同的值，但是在循环内部有3个分支需要每次判断，效率太低，有可能不能流水；可以改写为3个循环，这样循环内部就不用进行判断，这样虽然代码量增多了，但当数组规模很大(N很大)时，其效率能有相当的优势。改写的代码为：

```
for(int i = 0; i < 100; i++)
{
    a[i] += 5;
}
for(int i = 100; i < 200; i++)
{
    a[i] += 10;
}
for(int i = 200; i < N; i++)
{
    a[i] += 20;
```

```
}
```

关于循环内部的依赖，见如下一段程序：

```
for(int i = 0; i < N; i++)
{
    int x = f(a[i]);
    int y = g(x);
    int z = h(x,y);
}
```

其中f，g，h都是一个函数，可以看到这段代码中x依赖于a[i]，y依赖于x，z依赖于xy，每一步计算都需要等前面的都计算完成才能进行，这样对cpu的流水结构也是相当不利的，尽量避免此类写法。另外C语言中的restrict关键字可以修饰指针变量，即告诉编译器该指针指向的内存只有其自己会修改，这样编译器优化时就可以无所顾忌，但目前VC的编译器似乎不支　　持该关键字，而在DSP上，当初使用restrict后，某些循环的效率可提升90%。

c. 定点化

定点化的思想是将浮点运算转换为整型运算，目前在PC上我个人感觉差别还不算大，但在很多性能一般的DSP上，其作用也不可小觑。定点化的做法是将数据乘上一个很大的数后，将所有运算转换为整数计算。例如某个乘法我只关心小数点后3位，那把数据都乘上10000后，进行整型运算的结果也就满足所需的精度了。

d. 以空间换时间

空间换时间最经典的就是查表法了，某些计算相当耗时，但其自变量的值域是比较有限的，这样的情况可以预先计算好每个自变量对应的函数值，存在一个表格中，每次根据自变量的值去索引对应的函数值即可。如下例：

```
//直接计算
for(int i = 0 ; i < N; i++)
{
    double z = sin(a[i]);
}

//查表计算
double aSinTable[360] = {0, ..., 1,...,0,...,-1,...,0};
for(int i = 0 ; i < N; i++)
{
    double z = aSinTable[a[i]];
}
```

后面的查表法需要额外耗一个数组double aSinTable[360]的空间，但其运行效率却快了很多很多。

e. 预分配内存

预分配内存主要是针对需要循环处理数据的情况的。比如视频处理，每帧图像的处理都需要一定的缓存，如果每帧申请释放，则势必会降低算法效率，如下所示：

```
//处理一帧
void Process(BYTE *pimg)
{
    malloc
    ...
```

```
    free
}

//循环处理一个视频
for(int i = 0; i < N; i++)
{
    BYTE *pimg = readimage();
    Process(pimg);
}
```

```
//处理一帧
void Process(BYTE *pimg,  BYTE *pBuffer)
{
    ...
}

//循环处理一个视频
malloc pBuffer
for(int i = 0; i < N; i++)
{
    BYTE *pimg = readimage();
    Process(pimg, pBuffer);
}
free
```

前一段代码在每帧处理都malloc和free，而后一段代码则是有上层传入缓存，这样内部就不需每次申请和释放了。当然上面只是一个简单说明，实际情况会比这复杂得多，但整体思想是一致的。

指令优化

对于经过前面算法和代码优化的程序，一般其效率已经比较不错了。对于某些特殊要求，还需要进一步降低程序耗时，那么指令优化就该上场了。指令优化一般是使用特定的指令集，可快速实现某些运算，同时指令优化的另一个核心思想是打包运算。目前PC上intel指令集有MMX，SSE和SSE2/3/4等，DSP则需要跟具体的型号相关，不同型号支持不同的指令集。intel指令集需要intel编译器才能编译，安装icc后，其中有帮助文档，有所有指令的详细说明。

例如MMX里的指令 __m64 _mm_add_pi8(__m64 m1, __m64 m2)，是将m1和m2中8个8bit的数对应相加，结果就存在返回值对应的比特段中。假设2个N数组相加，一般需要执行N个加法指令，但使用上述指令只需执行N/8个指令，因为其1个指令能处理8个数据。

实现求2个BYTE数组的均值，即z[i]=(x[i]+y[i])/2，直接求均值和使用MMX指令实现2种方法如下程序所示：

```
#define N 800
BYTE x[N],Y[N], Z[N];
inital x,y;...
//直接求均值
for(int i = 0; i < N; i++)
{
    z[i] = (x[i] + y[i]) >> 1;
}

//使用MMX指令求均值，这里N为8的整数倍，不考虑剩余数据处理
__m64 m64X, m64Y, m64Z;
for(int i = 0; i < N; i+=8)
{
    m64X = *(__m64 *)(x + i);
    m64Y = *(__m64 *)(y + i);
    m64Z = _mm_avg_pu8(m64X, m64Y);
    *(__m64 *)(x + i) = m64Z;
}
```

使用指令优化需要注意的问题有：

a.关于值域，比如2个8bit数相加，其值可能会溢出；若能保证其不溢出，则可使用一次处理8个数据，否则，必须降低性能，使用其他指令一次处理4个数据了；

b.剩余数据，使用打包处理的数据一般都是4、8或16的整数倍，若待处理数据长度不是其单次处理数据个数的整数倍，剩余数据需单独处理；

补充——如何定位程序热点

程序热点是指程序中最耗时的部分，一般程序优化工作都是优先去优化热点部分，那么如何来定位程序热点呢？

一般而言，主要有2种方法，一种是通过观察与分析，通过分析算法，自然能知道程序热点；另一方面，观察代码结构，一般具有最大循环的地方就是热点，这也是前面那些优化手段都针对循环结构的原因。

另一种方法就是利用工具来找程序热点。x86下可以使用vtune来定位热点，DSP下可使用ccs的profile功能定位出耗时的函数，更近一步地，通过查看编译保留的asm文件，可具体分析每个循环结构情况，了解到该循环是否能流水，循环ii值，以及制约循环ii值是由于变量的依赖还是运算量等详细信息，从而进行有针对性的优化。由于Vtune刚给卸掉，没法截图；下图是CCS编译生成的一个asm文件中一个循环的截图：

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 256
;*      Loop opening brace source line   : 257
;*      Loop closing brace source line   : 347
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 25
;*      Unpartitioned Resource Bound     : 16
;*      Partitioned Resource Bound(*)    : 17
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     6        7
;*      .S units                     3        5
;*      .D units                     3       15
;*      .M units                     0        1
;*      .X cross paths              13        4
;*      .T address paths             6       10
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           4        0     (.L or .S unit)
;*      Addition ops (.LSD)         30       22     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             7        6
;*      Bound(.L .S .D .LS .LSD)    16       17*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 25 Did not find schedule
;*         ii = 26 Did not find schedule
;*         ii = 27 Schedule found with 2 iterations in parallel
;*      Done
;*
;*      Collapsed epilog stages       : 1
;*      Prolog not removed
;*      Collapsed prolog stages       : 0
;*
;*      Minimum required memory pad   : 0 bytes
;*
;*      For further improvement on this loop, try option -mh510
;*
;*      Minimum safe trip count       : 1
;*----------------------------------------------------------------------------*
```

最后提一点，某些代码使用Intel编译器编译可以比vc编译器编译出的程序快很多，我遇到过最快的可相差10倍。对于gcc编译后的效率，目前还没测试过。

相关文章

- 通过正确的权衡来获得最便捷有效的故障排除及最快速可行的优化
- 为什么转置512×512矩阵，会比513×513矩阵慢很多？
- 从 JavaScript 数组去重谈性能优化
- 陈皓：性能调优攻略
- 再谈Yahoo关于性能优化的N条规则
- 给网页设计师和前端开发者看的前端性能优化
- 程序员要懂得“大道至简”