VCB-Studio教程19 AVS的多线程优化-MPP的使用

作为一个十几年前设计的程序，avs的多线程和内存管理可谓是一团糟。avs原生没有任何的多线程优化：

source = LWLVS(“”) #输入片源(#后面的表示注释，不影响avs的运行)

denoise = source.RemoveGrain(20) #对片源降噪

repaired = denoise.Repair(source, 1) #将降噪后的视频，对比片源修复一些细节

这个简单的avs脚本用了3个滤镜。假设你的CPU有4个核心（假设没有超线程），3个滤镜中只有RemoveGrain在滤镜设计的时候有多线程优化：

source = LWLVS(“”)只会在核心1里面跑；

denoise = source.RemoveGrain(20)会在4个核心跑；

repaired = denoise.Repair(source, 1)也只会在核心1里面跑，跟LWLVS共用一个核心。

这三个滤镜，后面的依赖前面的。最后脚本的效率由跑的最慢的滤镜决定。可想而知，瓶颈容易出现在source和repaired两个上面。这两个滤镜被挤在一个线程里面，效率堪忧。如果三个滤镜都只能单线程跑，整个脚本等于只能用到cpu的一个线程。这是avs的一个弱点：缺乏原生多线程优化机制。

另外，滤镜之间的工作，需要帧的缓存：

1. source = LWLVS(“”)读入片源第0帧，记录在内存里，记为A
2. denoise = source.RemoveGrain(20) 拿过A帧，算出第0帧降噪后的内容，记为B
3. repaired = denoise.Repair(source, 1)拿过A和B，算出修复后的内容。

可见，要让这个脚本正常运转，A和B两个缓存必不可少。事实上，除了这种滤镜间需要缓存，滤镜本身计算也要缓存。对于时域滤镜，经常需要读取当前帧前后的内容，就需要缓存更多的帧。

问题是，如果在第二步结束，准备计算第三步，缓存A就被扔了呢？avs就不得不这么做：

先把B存好了；然后重新执行步骤1，得出A。

这时候，LWLVS这个滤镜就被重复执行了。重复计算的帧对于效率的浪费是很严重的。

avs的缓存是自动管理的。默认情况下，avs会开销1GB的内存（你可以用setMemoryMax()修改，但是原生32bit的avs最多只能4GB(如果在32bit系统下，只能用到2GB/3GB)。历史遗留问题导致我们现在没办法安全用64bit的avs）。4GB对于高级脚本来说捉襟见肘，导致的后果就是缓存不够，大量帧被迫要被重复计算。(并且avs本身需要占用一些内存，实际可用滤镜内存一般不建议超过3GB)

于是我们寻求一个解决方案，能让我们利用更多的缓存，以及手动给一些中间步骤足够多的缓存，让下游滤镜在调用的时候不必重复计算。本教程讲述如何用MP\_Pipeline实现三个目的：

1. avs滤镜间的多线程优化
2. 更多内存控制和开销
3. 手动管理部分重要的缓存

1. MPP原理解释——流水线运行多个avs

对于多线程和内存，一个可行的想法是多开：

第一个avs运行LWLVS，输出source

第二个avs可以接过source，运行RemoveGrain，输出denoise，并传递source

第三个avs可以接过source和denoise，运行Repair

三个avs模块，每个都在独立的进程里面运行，都可以有高达4GB的可用缓存。这样就优化了一些多线程的占用，也增加了内存的可用度。下面只需要规定缓存了：上游的模块丢给下游足够的预存。在每个avs结束的时候，规定输出的clip缓存数量。

MPP干的就是这些事情。

2. MPP语法演示

把上面的avs用MPP写出来:

MP\_Pipeline(“””

### inherit start ###

RamUsage = 1.0

### inherit end ###

SetMemoryMax(min(3000,500\*RamUsage))

Source = LWLibavVideoSource(“xxx.mp4”,threads=1)

### export clip: source

### prefetch : 48,32

### ###

SetMemoryMax(min(3000,3000\*RamUsage))

denoise = source.RemoveGrain(20)

### export clip: denoise

### pass clip: source

### prefetch : 16,8

### ###

SetMemoryMax(min(3000,3000\*RamUsage))

denoise.Repair(source,1)

“””)

用MPP优化的avs，所有的内容一定是包裹在

MP\_Pipeline(“””

“””)

里面的。注意引号是英文格式下的引号。

### ###(中间有个空格)代表分隔符。如上文所示，两个分隔符将整个avs分成了3个模块（block）。

### inherit start ### 和 ### inherit end ###中间的代码，是每个block通用的变量或者指令。比如这次我们设置了一个RamUsage的变量，来指定全局内存分配量。

### export clip: clip1, clip2, clip3 代表这个模块向下游输出哪些clip。这些clip一定是当前模块最新生成的。比如上文中，第一个模块输出了source，第二个模块输出了denoise。当前模块生成的，需要向下游传递的，都需要用export clip来传递。每个block的last 是自动export的；适用下文的prefetch。

### pass clip: clip1, clip2, clip3 代表这个模块向下游传递哪些clip。这些clip是更上游的模块丢来的，当前模块不做改动，直接丢下去。比如上文中，第二个模块原封不动的传递了source

### prefetch: x,y表示当前block输出的clip，设置怎样的强制缓存(x>y)。具体x,y如何设置，我们后文再说。

3. MPP中内存设置

每个模块，你可以用SetMemoryMax()来指定使用的内存数量。单位是MB

比如SetMemoryMax(3000)表示允许使用3000MB的内存

内存的设置，一般是视模块的复杂度而定。越是复杂的模块，含有的组合滤镜越多（比如SMDegrain, QTGMC等大型组合滤镜），就适合给越多的内存。如果是本身开销比较小的（比如f3dkb, RemoveGrain）滤镜居多，那就可以少给一点。在64bit的系统上，使用32bit的avs，最大可以开到3500左右。

同时，MPP本身需要一些额外的开销。所以avs部分占据内存，差不多是你设置的内存数量总和的110%。

对于16GB的内存，avs的开销不建议超过10GB。因为本身x264和系统还需要占据相当数量的内存。这导致复杂的avs很容易爆内存。一个简单的调节方式是，加一个宏观的控制变量RamUsage：

.这个变量默认是100%，可以调大也可以调小

.每个block的内存占用，是一个 你觉得合适的预设值\*RamUsage

.每个block的内存占用不超过3000

把以上自然语言用数学语言表示，就是：

每个block的内存分配 = 预设值\*RamUsage & 3000MB 中较小的一个

所以上文的avs我们采用的写法是SetMemoryMax(min(3000,500\*RamUsage))。这种写法的好处在于，如果你觉得内存总体分配的太多或者太少，你可以通过调节RamUsage来一键调整内存分配。

4. MPP中缓存设置

prefetch: x,y是MPP中设置缓存的。因为avs的滤镜并非是一帧一帧的顺序处理，而是可能需要用到前后多帧，所以缓存的设置就格外重要。如果请求的帧不在缓存内，意味着这一帧需要被重新计算。

对于在avs中出现的每一个clip，它都是会被请求的。要么它作为输出的clip，被输出请求，要么它参与滤镜运算，被下游滤镜请求。MPP的缓存机制是，如果一个clip在某个block被export，并且block设置了prefetch: x,y，那么对这个clip被请求的位置，前后共缓存x帧（加上被请求的位置就是x+1帧），其中，向后（这里向后意味着时间较早的帧，也是在编码过程中更早被处理的帧）缓存y帧。

越是在上游的block，输出的帧，向后缓存的就需要越多，因为下游的处理是滞后的。如果下游需要请求上游的帧，就需要保留较大的缓存。所以y的设置，一般就是需要覆盖下游所有的请求。也就是说，上游的y需要比下游的x来的大。这样上游的向后缓存就可以照顾到下游所有缓存的需求。

既然总共缓存x帧，向前缓存的数量就是x-y。这个值通常取值8左右比较合理；可以照顾绝大多数的时域滤镜，和block间的性能缓存。

如何设置一套优秀的缓存结构却是有经验公式的。假设从上到下，Block的输出分别是：

prefetch: xs,ys (这是第一个block输出源的prefetch)

prefetch: xn,yn (这是第二个block输出的prefetch)

prefetch: xn-1,yn-1 (这是第三个block输出的prefetch)

prefetch: xn-2,yn-2 (这是第四个block输出的prefetch)

……

prefetch: x1,y1 (这是倒数第二个block输出的prefetch)

**最后一个block无需prefetch，因为它是直接交给avs主进程运行；所有MPP的指令(prefetch, export clip, pass clip)无效。**

怎么设置比较科学呢：

x1,y1 = 16,8

x2,y2 = 26,18

x3,y3 = 36,28

……

xn,yn = xn-1+10,xn-1+2

xs,ys = xn+32,xn+16 (源block的输出最好给非常大的缓存。你还可以给的更大一点)

5. MPP使用时候的一些调试经验

1、将开头结尾的MP\_Pipeline(“”” 和 “””)用#注释掉，就可以完全取消MPP的添加。剩下的avs脚本就相当于是一个没有MPP的脚本（那些prefetch之类的因为是###开头，也会被认为是注释）。

所以在调试的时候可以先取消MPP架构调试；否则MPP的缓存机制会让在大量随机取样的调试变得十分缓慢且吃内存。

2、如果一个clip是由某个速度黑洞滤镜产生的，那么这个clip一旦被重新计算，代价就很高。这时候最好将滤镜单独放一个block，并且设置export clip和prefetch，保证下游请求的时候，一定是在MPP的缓存中，不用担心重复计算。

3、第一个block只干一件事情：读取source并且export足够多的帧给下游。不要做多余的事情；否则很容易造成在第一个block内，源被重复的定位读取，很容易触发滤镜bug导致花屏

4、MPP输出clip的时候是会抹掉音频信息的。所以不要用MPP去处理音频；也不要尝试去输出将信息记录在音频里面的clip（比如MVtools滤镜输出的相关clip，会把有效信息记录在音频里面）