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: x_s,y_s (这是第一个 block 输出源的 prefetch)
prefetch: x_n,y_n (这是第二个 block 输出的 prefetch)
prefetch: x_{n-1},y_{n-1} (这是第三个 block 输出的 prefetch)
prefetch: x_{n-2},y_{n-2} (这是第四个 block 输出的 prefetch)
……
prefetch: x_1,y_1 (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：

$x_1,y_1 = 16,8$
$x_2,y_2 = 26,18$
$x_3,y_3 = 36,28$
……
$x_n,y_n = x_{n-1}+10,x_{n-1}+2$

$x_s,y_s = x_n+32,x_n+16$ (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）