VCB-Studio 教程 12 16bit YUV 的处理

本教程旨在讲述 avs/vs 处理中，YUV 不同 bitdepth 的转换，以及一些简单的高精度处理用法。

0. YUV 的量化（以下部分主要来自 mawen1250 的讲解）

RGB 模型下，三个平面都属于强度，而转换为 YUV 之后，只有亮度 Y 属于强度，UV 则属于色差。在模拟信号表示（或者计算机中浮点数表示）中，用 0-1 来表示强度的强弱。R、G、B 的范围都是 0-1 前提下，Y 的来源是 R、G、B 的加权平均，权重和为 1，所以 Y 也是 0-1。在使用无符号整数的时候，8bit 下常常用[16,235]来对应 0-1，这就是 tvrange(limited range)；也可以用[0,255]来对应，这就是 pcrange(full range)。注意，一般 RGB 永远使用的是 pcrange，只有 YUV 会出现 pcrange 和 tvrange 的设定。

U、V 的来源是对 R、G、B、Y 作差。比如 Cb 是类似 B-Y，Cr 是类似 R-Y。那么不做任何其他处理的话，其范围是一个奇怪的值： 现在假设 Y=(R+G+B)/3，那么 Cb 最小的值就是当 B=0、R=1、G=1，也就是-2/3；Cb 最大的值是当 B=1、R=0、G=0，这时候 Cb 就是 2/3。于是 Cb 的取值范围就变成了[-2/3, 2/3]。

但我们显然不希望 UV 的取值范围是这么一个奇怪的数值，而且是和 R、G、B 在 Y 中的权重有关。所以我们就给它乘上一个修正的 scale，标准化到[-0.5, 0.5]的范围，动态范围和 Y 保持一致，都是 1。而 UV 之所以可以取负值是因为它是作差的产物，所以当 B 和 Y 相同的时候，代表没有色差，Cb 就是 0，而 0 就是色差通道的中点（neutral point）。

在用无符号整型表示数据的时候，因为没有负数，所以我们人为给 UV 规定了一个中点，在 8bit 下就是 128。然后[0-0.5, 0+0.5]就被对应到了[128-112, 128+112]，也就是[16, 240]，这是 tvrange 下 UV 的取值范围；对于 UV 来说，limited range 下 8bit 动态范围就是 224。

然而 full range 下，8bit 的动态范围是 255，所以定义成了[128-255/2, 128+255/2]，得到了[0.5, 255.5]，不在[0,255]范围内。所以 full range 的 chroma 其实定义并非完美；好在实际操作中极难碰到临界的 chroma 取值，所以实际影响不大。

总结一下：8bit 下，tvrange 时候，Y 的范围是[16,235]，UV 的范围是[16,240]，pcrange 下则都是[0,255]。UV 以 128 为 neutral point。

当扩展到更高 bitdepth 的整数时候，比如转换为 10bit，tvrange 下的操作是 YUV 都 * 4，原来是(160, 60,140)，变成(640,240,560)。pcrange 下，原则上是 Y 转换到[0,1]的浮点，UV 对齐中点，再进行对应的转换：

Y：160/255*1023 ≈ 642
U：512 + (60-128)/255*1023 ≈ 239
V: 512 + (140-128)/255*1023 ≈ 560

注意，UV 不等于 8bit 的数值直接*1023/255。虽然一个简单的近似公式是全都 * 4，这在 tvrange 下是精确计算，pcrange 下是近似计算。16bit 下计算方法类似，只不过倍数从 4 变成了 256，动态范围从 1023 变为 65535。

1. avs 中高精度 YUV 的显示方法——stacked/interleaved 16bit

avs 当中，默认的精度是每个像素，每个通道 8bit. 如果想使用 10/16bit 的精度，就必须通过 8bit 来伪装，更精确的说，让两个像素的位置，来记录一个像素的信息，从而提供 16bit 的储存空间。

考虑如下 2x2 的 16bit 数值：
5762　18329
52543　 345

每个数字的前 8 位，我们称为 MSB(Most Significant Bits,最重要的位数。好比一个数字的十位比个位重要)，后 8 位称为 LSB(Least Significant Bits,不重要的位数)

将每个数字转换成(MSB, LSB)的格式，我们得到(MSB*256 + LSB=16 位原数)：

(22,130) (71,153)

(205,63) (1,89)

其中，每一位的范围是 0~255， 128 为中值。所以如果将这四个 16bit 的数值，通过四舍五入(rounding)到 8bit，结果应该是：

23, 72

205,1

换言之，和这组 16bit 的显示的效果，最接近的 8bit 矩阵就是上文所示。

如果采用截位取整，结果和四舍五入相差并不大：

22, 71

205,1

一个简单的想法是，通过两个像素来并排显示一个像素，这样显示的后果是：

22,130,71,153

205,63,1,89

这种显示方式，叫做 interleaved 16bit。interleaved 是交织的意思。它的特点是，图像宽度增倍，每个数据的 MSB 和 LSB 交织显示。其中，奇数列的数值，正好是截位取整后的数值。所以从 Interleaved 16bit 中，还是可以稍微看出点片源痕迹的：

原图：

Interleaved 16bit:

除了 Interleaved 16bit，还有一种存储方式，叫做 stacked 16bit(stacked 意思是层叠)。它的规则是：上下层叠两个矩阵，第一个矩阵放 MSB，第二个矩阵放 LSB。

比如之前提到的

(22,130) (71,153)

(205,63) (1,89)

如果用 stacked 16bit 来显示：

22 71

205 1

130 153

63 89

它的特点是，图像高度增倍，而且上半部分完全是 16bit 通过截尾后转为 8bit。所以 stacked 16bit 在 avs 中，表现是上半部分正常，下半部分很乱：

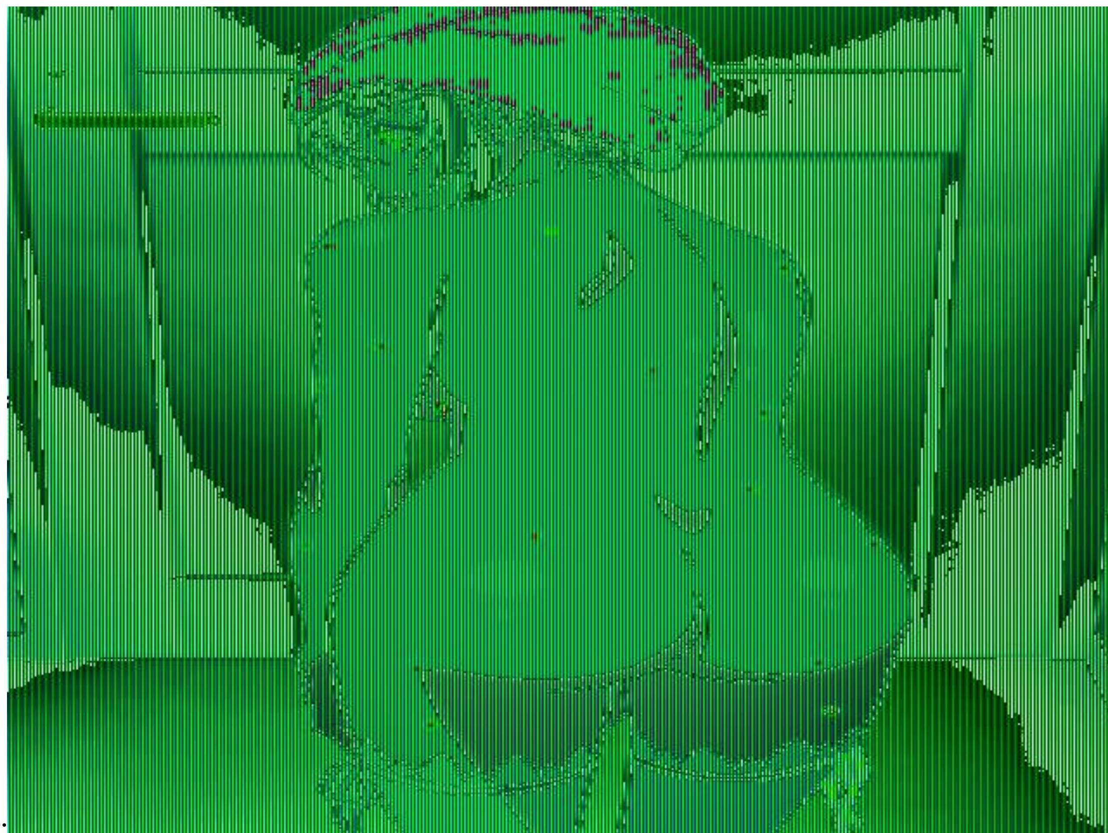
原图：

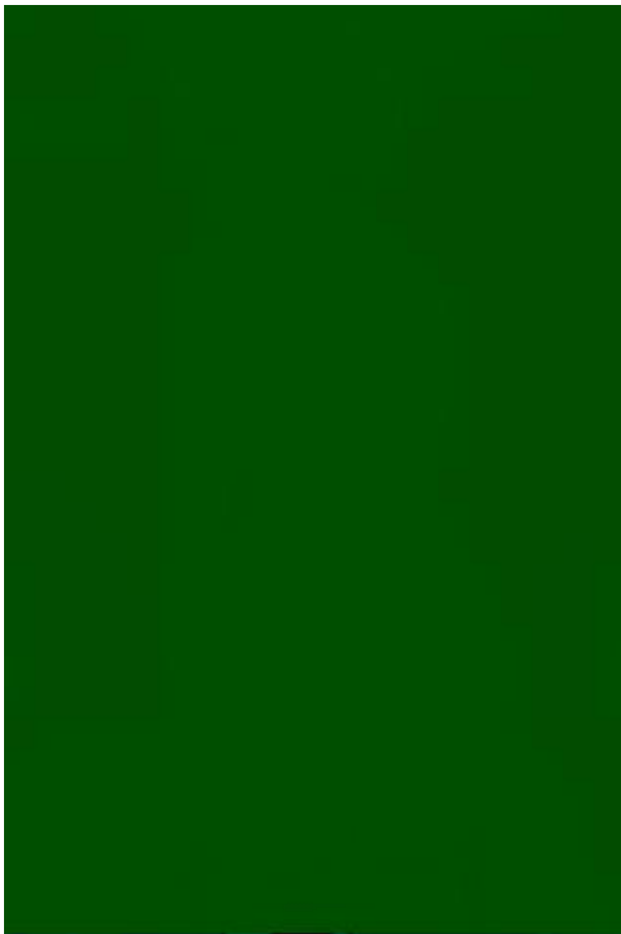
转 stacked 16bit：

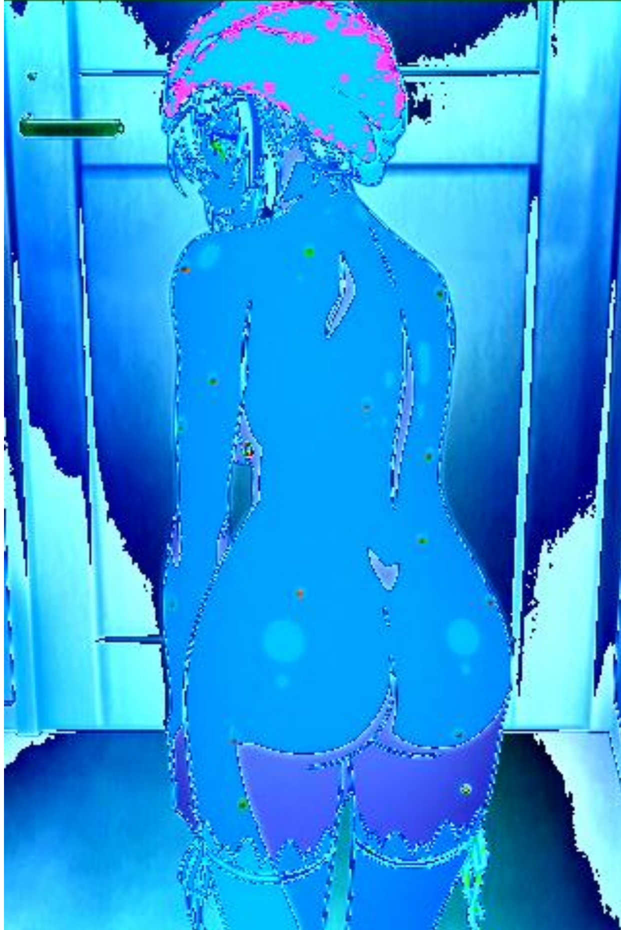
2. 10bit YUV 的显示方法——stacked/interleaved 10bit

10bit 的理解方式和 16bit 是完全一样的，除了 10bit 的 MSB 其实只有两位，最多是 0~4 的取值。加上 8bit 的 LSB，构成 10bit 的精度。

因为 10bit 的 MSB 很接近 0，所以只看 MSB 部分，效果很接近 YUV=(0,0,0)的效果（深绿色），但是在 stacked 16bit 中，你还是略微可以在 MSB 部分看出轮廓：

interleaved 10bit: 



stacked 10bit：

3. 日常处理中高精度的使用原则

x264/x265 可以接受原生 8bit 输入，也可以支持 interleaved 9~16bit 的输入，输入的时候通过--input-depth 指定。一般来说，编码 10bit AVC/HEVC，都是直接将处理好的 interleaved 10 数据喂给编码器。你也可以输出 interleaved 16bit，然后让编码器自己抖动到 10bit（x265 需要加入--dither 指令）。也就是说：

avs 中使用 interleaved 10bit 输出，同时编码指令中加入--input-depth 10，或者 avs 中使用 interleaved 16bit 输出，同时编码指令中加入--input-depth 16，如果是 x265，再加入--dither

avs 脚本，很多滤镜支持 16bit 的处理，是通过 stacked 16bit 实现的。16bit 处理对平面高精度过渡非常有利，因此很多涉及到平面部分的处理，比如 resize，降噪，去色带等，都推荐在 16bit 下做处理，并且全程保持 16bit 的精度。

处理完毕后，如果需要压制 8bit，就降低到 8bit，否则转为 interleaved 10/16bit 输出。

如果需要预览 avs，则转换为 RGB24。

4. avs 中不同精度之间相互转换的方法

如何在不同精度间相互转换，avs 的 Dither 工具提供了很多好用的滤镜：

8bit->stacked 16bit: U16()
比如说：
LWLVS(“00001.m2ts”) # yuv420p8
U16() # yuv420p16 stacked

stacked 16bit->8bit: ditherpost()
LWLVS(“00001.mkv”,format=”yuv420p16”,stacked=true) #将一个 mkv 读入为 stacked 16bit, 这个写法适用于读入 10bit 视频
ditherpost() #yuv420p8

ditherpost 默认会做抖动处理，效果比直接截尾或者四舍五入来的好。

stacked 16bit->interleaved 16bit: Dither_out()
LWLVS(“00001.m2ts”) # yuv420p8
U16() # yuv420p16 stacked
Dither_out() # yuv420p16 interleaved

interleaved 16bit->stacked 16bit: C16()
LWLVS(“00001.mkv”,format=”yuv420p16”,stacked=false) # interleaved 16bit
C16() # stacked 16bit

stacked 16bit->interleaved/stacked 10bit: Down10(stack=false/true)
LWLVS(“00001.m2ts”) # yuv420p8
U16() # yuv420p16 stacked
Down10(stack=false) # yuv420p10 interleaved

5. avs 中一些简单的 16bit 滤镜使用方法

很多滤镜就是原生工作在 stacked 16bit 下面的。比如说 Dither_resize16，就是 16bit 下的 resizer。Dither_RemoveGrain16()可以用于降噪。还有些滤镜通过参数来控制输入输出是否为 16bit；比如 f3kdb 的 input_mode 和 output_mode，设置为 1 的时候表示使用 stacked 16bit。

举一些简单的例子：

1、读入一个 m2ts，在 16bit 下做降噪和去色带，然后转为 720p 输出，准备做 10bit 压制：

```
LWLVS() #yuv420p8
U16() #yuv420p16 stacked
Dither_RemoveGrain16() #降噪
f3kdb(input_mode=1,output_mode=1) #去色带，输入输出指定为 stacked 16bit
Dither_resize16(1280,720) #降低为 720p
dither_out() #yuv420p16 interleaved
```

2、读入一个 10bit MKV，转为 720p，在 16bit 精度下加字幕，准备做 8bit 压制：

```
LWLVS(“mkv”,format=”yuv420p16”,stacked=true) #yuv420p16 stacked
Dither_resize16(1280,720) #降低为 720p
Textsub16(“subtitle.ass”) #加字幕
ditherpost() #输出为 yuv420p8
```

3、读入一个 30i 的演唱会原盘，对它进行反交错，然后用 SMDegrain 做高精度降噪，准备给 10bit 压制：

```
LWLVS(“m2ts”)
QTGMC(preset=”slow”,border=true)   #反交错。注意 QTGMC 不支持 16bit 处理
pre_nr16 = last.U16()   #反交错后，降噪之前，记录一个 16bit 的备份
SMDegrain(tr=2,thSAD=500,thSADC=200,refinemotion=true,pel=2,truemotion=false,hpad=16,vpad=16
,lsb_out=true)   #注意输入依旧是 QTGMC 后的结果，为 yuv420p8
#输出通过 lsb_out=true 来指定为 yuv420p16. SMDegrain 的内部运算精度为 32bit 浮点数。
dither_repair16(last, pre_nr16, 3, 3) #在 16bit 下对降噪的结果，对比 pre_nr16 做一些修复补偿。
dither_out() #转为 interleaved 16bit 输出。
```

对于 avs 结尾，可以加上这么一段开关：

```
output_depth = 10
output_depth == 10?Down10(stack=false):dither_convert_yuv_to_rgb(lsb_in=true,a1=0,a2=0.5)
```

注意，第一句的=是赋值，表示将 output_depth 赋值为 10；
第二句的==是判断是否相等，是的话，执行 dither_out(),否则执行 dither_conver_yuv_to_rgb(lsb_in=true)，做 stacked 16bit yuv->RGB 的转换。

这段开关的作用是：

1、你可以通过修改第一句，表示你想输出 10bit 与否
2、如果你指定输出 10bit，avs 会做 stacked 16bit->interleaved 16bit
3、否则，avs 会做 yuv420p16->RGB24，供你预览和调试 avs。

调试的时候可以将 output_depth 赋值为 8；编码前记得改为 10。如果是输出 16bit 如法炮制；只不过 Down10 改为 dither_out()

6. vs 中不同精度之间相互转换的方法

vs 原生支持高精度的 yuv 信息，但是这不意味着 vs 中处理精度和输入输出都自动用最高精度进行。vs 中，我们也需要用滤镜来进行处理。典型的包括 fmtc 和 zimg 所提供的滤镜。

fmtc 的使用比较简单，就一个 bitdepth 函数:

```
src16 = core.fmtc.bitdepth(src8,bits=16)

down10 = core.fmtc.bitdepth(res16,bits=10)
```

其中，你可以用 fulls 和 fulld 表示输入输出是否是 full。默认都是 false，表示进行 tvrange 的转换。只不过 fullrange 下在处理边界值有一点小问题，所以强迫症可以用 zimg 处理涉及 fullrange 的转换。

zimg 则可以通过 mvf 中的 depth 来实现：

```
src16 = mvf.Depth(src8, depth=16)

down10 = mvf.Depth(res16,depth=10)
```

同时它也是通过 fulls 和 fulld 来指定输入输出的 range。比如说我们把一个 fullrange yuv8bit 转为 limited 16bit:

```
src16 = mvf.Depth(src8, depth=16, fulls=True, fulld=False)
```

vs 中，多数情况下，滤镜的处理精度和输出精度就是输入的精度。所以 vs 的高精度处理无需指定 bitdepth，滤镜会自动读取精度信息，并自动选择输出精度。比如说系统自带的 resize，就是输入啥精度，输出啥精度。也有强制精度的，比如说 SangNomMod 就只能支持 8bit 输入输出，16bit 的 clip。使用的时候需要先降低到 8bit，处理完了再升回去。也有类似 f3kdb.Deband 这种，可以通过 output_depth 来指定输出精度的。

vs 的脚本末端可以加上这个指令：

```
Debug = False
if Debug:
    res=core.std.Interleave([src16,res])
    res=mvf.ToRGB(res,full=False,depth=8)
else: res = core.fmtc.bitdepth(res,bits=10)
```

其作用是，当 Debug 是 1/True 的时候，表示开启调试模式，先将 src16 和 res(一般处理完毕也是 16bit)交织每一帧，然后转为 RGB24 显示，这样可以绕开 vsedit 自带的转 RGB。否则当 debug=0/False 时候，表示关闭调试模式，将处理结果转为 10bit 输出，准备喂给编码器。