

MPSoC VCU Ctrl-SW 2020.2 输出NV12的YUV文件

付汉杰 hankf@xilinx.com

- 1. 介绍
 - 1.1. VCU输入和输出格式
 - 1.2. VCU内存的高度和宽度要求
 - 1.3. VCU内存的pitch
- 2. 显示YUV文件
 - 2.1. 工具RAW yuv player
 - 2.1.1. 技巧
 - 2.2. hexdump
- 3. 输出NV12/NV16格式文件
 - 3.1. 选项
- 4. 代码
- 5. 测试
 - 5.1. 1080分辨率
 - 5.1.1. 分辨率显示格式错误的现象
 - 5.2. 3840x2160分辨率
 - 5.2.1. x2160分辨率显示格式错误的现象
- 6. 未来工作
- 7. 参考文档

1. 介绍

Xilinx提供超低延时编解码方案，并提供了全套软件。[MPSoC Video Codec Unit](#)提供了详细说明。其中的底层应用是[VCU Control-Software\(Ctrl-SW\)](#)。

本文主要说明为Ctrl-SW增加输出NV12视频的功能。

1.1. VCU输入和输出格式

Video Codec Unit(VCU) 输入和输出都是是NV12/NV16格式的视频，Y分量存放在一块连续内存区，UV分量交替存放在Y分量后面的连续内存。具体信息，可以参考[VCU Product Guide](#)中的“Source Frame

Format”和“Memory Format”。

Ctrl-SW的输入文件最好是NV12/NV16格式的视频文件，由于不需要做格式转换，帧率(FPS)最高。但是Ctrl-SW的输出文件缺省是图像真实分辨率的I420/I422的文件，其中的Y、U、V分量，各自存在一块连续内存，UV分量没有像NV12/NV16格式的视频交替在一起。可以使用FFMPeg等工具，将I420的文件，转换成NV12/NV16格式的文件。

1.2. VCU内存的高度和宽度要求

对于视频的输入内存区，VCU要求高和宽都按32向上对齐。对于1920x1080分辨率,输入的buffer大小至少是1920x1088字节；对于3840x2160分辨率,输入的buffer大小至少是3840x2176字节。

对于视频的输出内存区，VCU要求宽以256地址对齐，高以64地址对齐。对于1920x1080分辨率,输出的buffer大小是2048x1088字节；对于3840x2160分辨率,输出的buffer大小是3840x2176字节。

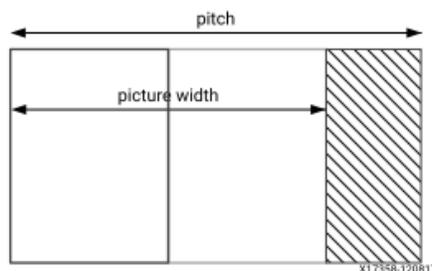
1.3. VCU内存的pitch

视频数据在内存区中存放时，两行之间的数据可以有间隔。对于每个像素的Y分量用8-bit表示的图像，每个像素的Y分量对应内存的一个字节，图像Y分量的每一行对应的内存大小就是其宽度代表的字节数。比如1920x1080,每一行图像的Y分量需要1920字节内存。如果以2048字节来存储一行1920x1080的图像数据，则在前面存放图像数据，后面的数据被VCU忽略。也可以参考PG252的“Figure 7: Frame Buffer Pitch”。

The encoder buffer must be one contiguous memory region and should be aligned to a 32-byte boundary.

The frame buffer width (pitch) may be larger than the frame width. When the pitch is greater than the frame width, pixels in each line beyond the picture width are ignored, as illustrated in the following figure.

Figure 7: Frame Buffer Pitch



Note: Encoder input buffers width and height should be in multiple of 32. Decoder output buffers width is in multiple of 256-byte. Height is in multiples of 64. For example, for 1920x1080 resolution, the decoder output is 2048x1088.

2. 显示YUV文件

2.1. 工具RAW yuv player

Github上的[RAW yuv player](#)能显示YUV文件，它的旧版本[Sourceforge RAW yuv player](#)在Sourceforge上。RAW yuv player的菜单“Colour”下，有各种颜色格式，菜单“Size”下有各种分辨率；菜单“Zoom”下可以选择图像缩放比例。

RAW yuv player的YUV420(YV12)格式，就是I420格式，可以显示Ctrl-SW缺省输出的YUV文件。RAW yuv player的NV12格式，也是Ctrl-SW的NV12格式，可以显示修改后的Ctrl-SW输出的YUV NV12文件。

2.1.1. 技巧

各种YUV文件，第一片数据一般都是分量Y。如果发现YUV文件的显示有问题，可以设置好分辨率，在菜单“Colour”下选择“Y”，只看其中的分量Y，当成黑白图片看。如果黑白图片是正常的，说明分量Y是对的。

2.2. hexdump

如果图片内容不对，可以使用二进制比较工具比较错误图片和正确图片，比如Beyond Compare。比较的时候，注意取消对齐设置。

如果没有二进制比较工具，可以使用hexdump把YUV文件按HEX格式转换为文本文件，再用文本比较工具，比如kdiff3、meld进行比较。hexdump输出时，会输出星号“*”代替一样的行；多个重复行，也只输出一个星号。为hexdump加上“-v”选项，则会输出所有数据。

```
xilinx@XSZHANKF$ hexdump -x test_1080p_h264.264.1f.i420.1920x1080.yuv > test_1080p_h264.264.1f.i420.1920x1080.yuv
xilinx@XSZHANKF$ hexdump -v -x test_1080p_h264.264.1f.i420.1920x1080.yuv > test_1080p_h264.264.1f.i420.1920x1080.yuv
```

3. 输出NV12/NV16格式文件

如果Ctrl-SW能输出NV12/NV16格式的文件，Ctrl-SW就能直接对自己的文件进行编码，测试时更加方便。

经过研究，在Ctrl-SW 2020.2里，实现了输出NV12/NV16格式文件的功能。

3.1. 选项

Ctrl-SW里有三种分辨率，分别是图像的真实分辨率，Meta数据分辨率，内存块（buffer）分辨率。

图像的真实分辨率，是真实显示的分辨率。

在Ctrl-SW里，为YUV数据分配内存时，根据图像分辨率，并按对齐要求像是对齐图像分辨率后，得到YUV数据的内存块大小，这就是对应的内存块（buffer）分辨率。对于解码，1920x1080分辨率的内存块是2048x1088字节；3840x2160分辨率的buffer是3840x2176字节。

另外分配内存后，每个内存块有一个对应的Meta数据，保存YUV数据的分辨率。Meta数据分辨率可能比内存块分辨率低。1920x1080分辨率的Meta数据分辨率是2048x1088；而3840x2160分辨率的Meta数据分辨率却是3840x2160，在高度上并没有像1080p时向上对齐。

所以输出NV12/NV16的视频时，也有多种组合。为了测试方便，实现了输出各种组合的NV12/NV16的视频。

如果使用选项“-yuv-nvx”，按得到的Meta数据的分辨率信息，从Y和UV分量的地址，逐行写入到文件。

如果使用选项“-yuv-nvx-1buffer”，行依照pitch长度，高使用分辨率的高度并向上按64字节对齐，计算出YUV整个的内存区大小，相当于内存块（buffer）分辨率，一次性写入到文件。

如果使用选项“-yuv-nvx-stride”，行依照pitch长度，高使用分辨率的高度，从Y和UV分量的地址，逐行输出。相当于宽按内存块（buffer）分辨率，高按Meta数据分辨率输出。

如果使用选项“-yuv-nvx-dispay”，按得到的显示分辨率信息，从Y和UV分量的地址，逐行输出。

另外，还附带增加了输出YUV文件时跳帧、抽帧的功能。如果使用选项“--yuv-skip-num”，则前面的指定数量的帧不会被输出；比如指定5，前面的5帧不会被写入到YUV文件。如果使用选项“--yuv-skip-interval”，则指定数字的倍数序号的帧，才会被输出；比如指定数字3，则只有序号是3的倍数的帧才会被写入到YUV文件。

4. 代码

在Ctrl-SW 2020.2里，添加如下代码后，可以直接输出NV12/NV16格式的文件。

下面是增加的全局变量定义。

```

int gi_yuv_output_skip_frame_num=0;
int gi_yuv_output_skip_frame_interval=0;
int gi_yuv_output_nv_x_flag=0;
int gi_yuv_output_nv_x_1buffer_flag=0;
int gi_yuv_output_nv_x_stride_flag=0;
int gi_yuv_output_nv_x_dispay_flag=0;
int gi_yuv_output_dispay_width=0;
int gi_yuv_output_dispay_height=0;

```

下面是增加的ctrlsw_decoder的命令行选项。

```

opt.addInt("--yuv-skip-num", &gi_yuv_output_skip_frame_num, "Skip frame number before writing YUV file.");
opt.addInt("--yuv-skip-interval", &gi_yuv_output_skip_frame_interval, "Interval frame number when writing YUV file.");
opt.addFlag("-yuv-nv-x", &gi_yuv_output_nv_x_flag, "Output NV12/NV16 YUV file.", 1);
opt.addFlag("-yuv-nv-x-1buffer", &gi_yuv_output_nv_x_1buffer_flag, "Output one continuous NV12/NV16 buffer to YUV file.");
opt.addFlag("-yuv-nv-x-stride", &gi_yuv_output_nv_x_stride_flag, "Output NV12/NV16 YUV file with VCU round-up padding.");
opt.addFlag("-yuv-nv-x-dispay", &gi_yuv_output_nv_x_dispay_flag, "Output NV12/NV16 YUV file with display width.", 1);

```

下面是在UncompressedOutputWriter::ProcessFrame()内部增加的判断是否输出NV12/NV16格式的视
频文件的判断代码。

```

if( gi_yuv_output_skip_frame_num_local < gi_yuv_output_skip_frame_num )
{
    return;
}

if( 0 == (gi_yuv_output_skip_frame_num_local%gi_yuv_output_skip_frame_interval) )
{
    return;
}

if( ( 1 == gi_yuv_output_nv_x_flag ) || ( 1 == gi_yuv_output_nv_x_1buffer_flag )
    || ( 1 == gi_yuv_output_nv_x_stride_flag ) || ( 1 == gi_yuv_output_nv_x_dispay_flag ) )
{
    ProcessFrameNVx( tRecBuf, info, iBdOut);
    return;
}

```

下面是显示视频参数的代码，用于调试，可以被屏蔽掉。

```

void UncompressedOutputWriter::ShowVideoInfo(AL_TBuffer& tRecBuf, AL_TInfoDecode info, int iBdOut)
{

    // only print one time.
    static int i_call_time=0;
    if( 0 != i_call_time )
    {
        return;
    }
    i_call_time++;

    iBdOut = convertBitDepthToEven(iBdOut);

    auto const iSizePix = (iBdOut + 7) >> 3;

    TFourCC tRecFourCC = AL_PixMapBuffer_GetFourCC(&tRecBuf);
    printFourCC( tRecFourCC, "Recorded frame buffer", __func__, __LINE__ );

    int sx = 1, sy = 1;
    AL_GetSubsampling(tRecFourCC, &sx, &sy);

    int iPitchSrcY = AL_PixMapBuffer_GetPlanePitch(&tRecBuf, AL_PLANE_Y);

    int iPitchSrcUV = AL_PixMapBuffer_GetPlanePitch(&tRecBuf, AL_PLANE_UV);
    if( iPitchSrcY != iPitchSrcUV )
    {
        LogInfo("YUV Y Plane pitch: %d does not equal to UV Plane pitch:%d at %s:%d.\n",
                iPitchSrcY, iPitchSrcUV, __func__, __LINE__ );
    }
    LogVerbose("YUV Y Plane pitch:%d, UV Plane pitch:%d at %s:%d.\n", iPitchSrcY, iPitchSrcUV, __func__, __LINE__

    AL_TDimension tYuvDim = AL_PixMapBuffer_GetDimension(&tRecBuf);
    int const iRoundUpYWidth = RoundUp(tYuvDim.iWidth, 256);
    if( iPitchSrcY != iRoundUpYWidth )
    {
        LogInfo("YUV Y Plane pitch %d does not equal to Y Round-up Width:%d at %s:%d.\n",
                iPitchSrcY, iRoundUpYWidth, __func__, __LINE__ );
    }

    // For 1920x1080, YuvDim.iHeight is always rounded up, it is 1920x1088.
    // For 3840x2160, YuvDim.iHeight is not rounded up, it is 3840x2160.
    int const iRoundUpHeight = RoundUp(tYuvDim.iHeight, 64);
    if( tYuvDim.iHeight != iRoundUpHeight )
    {
        // For 4K video, tYuvDim.iHeight(2160) does not equal to Round-up Height(2176)
        LogInfo("YUV Height: %d does not equal to Round-up Height:%d at %s:%d.\n",
                tYuvDim.iHeight, iRoundUpHeight, __func__, __LINE__ );
    }

    //int const iNumPix = tYuvDim.iHeight * tYuvDim.iWidth; // For I420 without extra padding bytes.
    const AL_EChromaMode stRecChromaMode = AL_GetChromaMode(tRecFourCC);

```

```

//int const iNumPixC = AL_GetChromaMode(tRecFourCC) == AL_CHROMA_MONO ? 0 : ((tYuvDim.iWidth + sx - 1) / sx) *
int const iLineNumPixC = (stRecChromaMode == AL_CHROMA_MONO) ? 0 : ((tYuvDim.iWidth + sx - 1) / sx);
int const iRoundUpLineNumPixC = (stRecChromaMode == AL_CHROMA_MONO) ? 0 : ((iPitchSrcUV + sx - 1) / sx);

    // Get display in sResolutionFound()
int const iNumPix = tYuvDim.iHeight * tYuvDim.iWidth;
int const iRoundUpNumPix = iRoundUpHeight * iPitchSrcY;

int const iNumPixC = iLineNumPixC * ((tYuvDim.iHeight + sy - 1) / sy);
int const iRoundUpNumPixC = iRoundUpLineNumPixC * ((iRoundUpHeight + sy - 1) / sy);
int i_y_buffer_size = iRoundUpNumPix * iSizePix;
int i_uv_buffer_size = 2 * iRoundUpNumPixC * iSizePix;

int i_yuv_buffer_size = i_y_buffer_size + i_uv_buffer_size;
LogVerbose("Subsampling sx:%d, sy:%d at %s:%d.\n", sx, sy, __func__, __LINE__ );
LogVerbose("Height:%d, Width:%d at %s:%d.\n", tYuvDim.iHeight, tYuvDim.iWidth, __func__, __LINE__ );
LogVerbose("Roundup Height:%d, Width:%d at %s:%d.\n", iRoundUpHeight, iPitchSrcY, __func__, __LINE__ );
LogVerbose("iNumPix:%d, iLineNumPixC:%d, iNumPixC:%d, iSizePix:%d at %s:%d.\n", iNumPix, iLineNumPixC, iNumPixC, iSizePix, __func__, __LINE__ );
LogVerbose("iRoundUpNumPix:%d, iRoundUpLineNumPixC:%d, iRoundUpNumPixC:%d at %s:%d.\n", iRoundUpNumPix, iRoundUpLineNumPixC, iRoundUpNumPixC, __func__, __LINE__ );

LogVerbose("NV12/NV16 YUV Plane Y: %d, UV: %d YUV: %d bytes at %s:%d.\n",
            i_y_buffer_size, i_uv_buffer_size, i_yuv_buffer_size, __func__, __LINE__ );

uint8_t* p_buff_y_plane = AL_PixMapBuffer_GetPlaneAddress(&tRecBuf, AL_PLANE_Y);
uint8_t* p_buff_uv_plane = AL_PixMapBuffer_GetPlaneAddress(&tRecBuf, AL_PLANE_UV);
LogVerbose("NV12/NV16 YUV Y Plane address: %p, UV Plane address: %p at %s:%d.\n",
            p_buff_y_plane, p_buff_uv_plane, __func__, __LINE__ );

int const iOffsetY_UV_Plane = ( (unsigned long long)p_buff_uv_plane - (unsigned long long)p_buff_y_plane);
if( i_y_buffer_size != iOffsetY_UV_Plane )
{
    LogInfo("YUV Y Plane size: %d does not equal to offset: %d between Y/UV plane at %s:%d.\n",
            i_y_buffer_size, iOffsetY_UV_Plane, __func__, __LINE__ );
}
}
}

```

下面是增加的输出NV12/NV16格式的视频文件的主体代码。

```

void UncompressedOutputWriter::ProcessFrameNVx(AL_TBuffer& tRecBuf, AL_TInfoDecode info, int iBdOut)
{

    if(!(YuvFile.is_open() || CertCrcFile.is_open()))
        return;

    static int i_call_time=0;
    i_call_time++;

    iBdOut = convertBitDepthToEven(iBdOut);

    auto const iSizePix = (iBdOut + 7) >> 3;

    TFourCC tRecFourCC = AL_PixMapBuffer_GetFourCC(&tRecBuf);

#ifdef 1
    ShowVideoInfo( tRecBuf, info, iBdOut);
#endif

    int sx = 1, sy = 1;
    AL_GetSubsampling(tRecFourCC, &sx, &sy);

    int iPitchSrcY = AL_PixMapBuffer_GetPlanePitch(&tRecBuf, AL_PLANE_Y);
    int iPitchSrcUV = AL_PixMapBuffer_GetPlanePitch(&tRecBuf, AL_PLANE_UV);

    AL_TDimension tYuvDim = AL_PixMapBuffer_GetDimension(&tRecBuf);

    const AL_EChromaMode stRecChromaMode = AL_GetChromaMode(tRecFourCC);
    int const iLineNumPixC = (stRecChromaMode == AL_CHROMA_MONO) ? 0 : ((tYuvDim.iWidth + sx - 1) / sx) ;

    uint8_t* p_buff_y_plane = AL_PixMapBuffer_GetPlaneAddress(&tRecBuf, AL_PLANE_Y);
    LogVerbose("NV12/NV16 YUV data Y Plane address: %p at %s:%d.\n", p_buff_y_plane, __func__, __LINE__ );

    LogVerbose("YUV data with VCU padding Height:%d, Y Width:%d, UV Width:%d at %s:%d.\n",
        tYuvDim.iHeight, iPitchSrcY, iPitchSrcUV, __func__, __LINE__ );

    if( 1 == gi_yuv_output_nv_x_1buffer_flag )
    {

        // For 1920x1080, YuvDim.iHeight is always rounded up, it is 1920x1088.
        // For 3840x2160, YuvDim.iHeight is not rounded up, it is 3840x2160.
        int const iWriteHeight = RoundUp(tYuvDim.iHeight, 64);

        //int const iNumPix = tYuvDim.iHeight * tYuvDim.iWidth; // For I420 without extra padding bytes.
        int const iRoundUpNumPix = iWriteHeight * iPitchSrcY; // For NV12/NV16 with extra padding bytes.
        //int const iNumPixC = AL_GetChromaMode(tRecFourCC) == AL_CHROMA_MONO ? 0 : ((tYuvDim.iWidth + sx - 1)
        int const iRoundUpLineNumPixC = (stRecChromaMode == AL_CHROMA_MONO) ? 0 : ((iPitchSrcUV + sx - 1) / sx);
        int const iRoundUpNumPixC = iRoundUpLineNumPixC * ((iWriteHeight + sy - 1) / sy);
        int i_y_buffer_size = iRoundUpNumPix * iSizePix;
        int i_uv_buffer_size = 2 * iRoundUpNumPixC * iSizePix;
        int i_yuv_buffer_size = i_y_buffer_size + i_uv_buffer_size;
    }
}

```

```

// Display YUV file: 1920x1080: 2048x1088; 3840x2160: 3840 x 2176
if( 1 == i_call_time )
{
    LogInfo("Diplay NV12/NV16 YUV file of one continuous buffer with Height:%d, Y Width:%d, UV Width:%d at %s:%d",
            iWriteHeight, iPitchSrcY, iPitchSrcUV, __func__, __LINE__ );
}
YuvFile.write((const char*)p_buff_y_plane, i_yuv_buffer_size);
}
else
{
    int iWriteHeight;
    int iWriteYWidth;
    int iWriteUVWidth;

    // For 1920x1080, YuvDim.iHeight is always rounded up, it is 1920x1088.
    // For 3840x2160, YuvDim.iHeight is not rounded up, it is 3840x2160.
    if( 1 == gi_yuv_output_nv_x_stride_flag )
    {
        iWriteHeight = tYuvDim.iHeight;
        iWriteYWidth = iPitchSrcY;
        iWriteUVWidth = 2 * ((iPitchSrcUV + sx - 1) / sx);
        LogVerbose("Diplay NV12/NV16 YUV file of stride with Height:%d, Y Width:%d, UV Width:%d at %s:%d",
                  tYuvDim.iHeight, iPitchSrcY, iPitchSrcUV, __func__, __LINE__ );
    }
    else
    {
        if( 0 != gi_yuv_output_dispay_height )
        {
            // Display YUV file: 1920x1080: 1920x1080.
            iWriteHeight = gi_yuv_output_dispay_height;
        }
        else
        {
            // Display YUV file: 1920x1080: 1920x1088.
            iWriteHeight =tYuvDim.iHeight;
        }

        if( 0 != gi_yuv_output_dispay_width )
        {
            iWriteYWidth = gi_yuv_output_dispay_width;
            iWriteUVWidth = 2 * ((gi_yuv_output_dispay_width + sx - 1) / sx);
        }
        else
        {
            iWriteYWidth =tYuvDim.iWidth;
            iWriteUVWidth = 2 * iLineNumPixC;
        }
    }
}
if( 1 == i_call_time )

```

```

{
    LogInfo("Display NV12/NV16 YUV file with Height: %d, Y Width: %d, UV Width: %d at %s:%d.\n",
           iWriteHeight, iWriteYWidth, iWriteUVWidth, __func__, __LINE__ );
}

// two writes, skip padding bytes between Y plane and UV plane.
uint8_t* p_buff_y=p_buff_y_plane;
int iYBytes=0;
for( int i=0; i<iWriteHeight; i++ )
{
    // Write each line of Y plane, and skip padding bytes between each line.
    YuvFile.write((const char*)p_buff_y, iWriteYWidth);
    p_buff_y = p_buff_y + iPitchSrcY;
    iYBytes+=iWriteYWidth;
}

uint8_t* p_buff_uv_plane = AL_PixMapBuffer_GetPlaneAddress(&tRecBuf, AL_PLANE_UV);
LogVerbose("NV12/NV16 YUV data UV Plane address: %p at %s:%d.\n", p_buff_uv_plane, __func__, __LINE__);
uint8_t* p_buff_uv=p_buff_uv_plane;
int iUVBytes=0;
for( int i=0; i<(iWriteHeight/sy); i++ )
{
    // Write each line of UV plane, and skip padding bytes between each line.
    YuvFile.write((const char*)p_buff_uv, iWriteUVWidth);
    p_buff_uv = p_buff_uv + iPitchSrcUV;
    iUVBytes+=iWriteUVWidth;
}
LogVerbose("write NV12/NV16 YUV data Y: %d bytes, UV: %d bytes at %s:%d.\n",
           iYBytes, iUVBytes, __func__, __LINE__ );
}
}

```

5. 测试

在开发过程中，测试了1920x1080、3840x2160分辨率的NV12图像。

5.1. 1080分辨率

1920x1080分辨率时，以内存块分辨率输出，分辨率是1920x1088；以pitch长度和分辨率高度向上对齐后输出，分辨率是2048x1088；以pitch长度和Meta分辨率的高度输出，分辨率是2048x1088；以显示分辨率输出，分辨率是1920x1080。如果没有特殊说明，图像都是NV12格式的，也是以NV12格式显示。

查看YUV文件时，必须设置正确的分辨率和格式，否则数据显示会混乱。

以分辨率1920x1080显示选项“-yuv-nvx-dispay”输出的图片，结果正常。

以分辨率2048x1088显示选项“-yuv-nvx-1buffer”输出的图片，结果正常。右边有一块红色图像，是因为对应的内存区没有真实图像数据。

以分辨率2048x1088显示选项“-yuv-nvx-stride”输出的图片，结果正常。右边也有一块红色图像。

5.1.1. 分辨率显示格式错误的现象

以分辨率1920x1080显示分辨率2048x1088图片，图像混乱了。因为实际图像数据每行2048字节，显示时每行1920字节，所以读出的数据混乱了，上面的连续红色块变成了小块，分布到了图像各处。

以分辨率2048x1080显示分辨率2048x1088图片，最上面有绿条，因为把8行的Y分量数据当成了UV分量数据。这8行的实际图像是黑色的。

5.2. 3840x2160分辨率

3840x2160分辨率时，以pitch长度和分辨率高度向上对齐后的分辨率输出，分辨率是3840x2176；其它模式输出，分辨率都是3840x2160。

以分辨率3840x2160显示选项“-yuv-nvx-dispay”输出的图片，结果正常。

以分辨率3840x2176显示选项“-yuv-nvx-1buffer”输出的图片，结果正常。下面有一条绿色，也是因为对应的内存区没有有效的图像数据。

5.2.1. x2160分辨率显示格式错误的现象

以分辨率3840x2176格式显示分辨率3840x2160的图片，最下面有绿条，是因为把部分UV分量数据当成了Y分量数据，导致最下面部分图像缺少UV分量数据。

以分辨率3840x2160、I420格式显示选项“-yuv-nvx-stride”输出的NV12分辨率3840x2176图片，轮廓正常，色彩异常。轮廓正常，是因为对Y分量数据的解析是对的；色彩异常是因为对UV分量数据的解析是错的。

6. 未来工作

未来可以继续测试NV16的图像，也可以测试其它分辨率的图像。

7. 参考文档

