Multimedia User Guide

发行版本 (HEAD detached from eb5ee6b)

SOPHGO

2025 年 06 月 19 日

目录

发布记录

版本	发布日期	说明
V2.0.2	2019.11.15	第一版增加 OpenCV 私有 API、FFMPEG API，增强 libyuv API 接口介绍
V2.1.0	2020.07.06	第二版优化 FFMPEG 部分的章节结构，增加视频编码部分的内容介绍
V2.2.0	2020.08.26	第三版调整字体，优化版式，增加 bmx264 视频编码器的内容
V2.2.1	2021.02.09	更新 Opencv 新增接口的说明: bmcv::/av::/Mat:: 下新增接口

CHAPTER 1

声明

法律声明

注意

您购买的产品、服务或特性等应受算能商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，算能对本文档内容不做任何明示或默示的声明或保证。由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

技术支持

地址
北京市海淀区丰豪东路 9 号院中关村集成电路设计园（ICPARK）1 号楼

邮编
100094

网址
https://www.sophgo.com/

邮箱
sales@sophgo.com

电话
+86-10-57590723 +86-10-57590724

SDK 发布记录

版本	发布日期	说明
v1.5.1	2024.05.01	v1.5.1 版本发布。
v1.6.0	2024.06.14	v1.6.0 版本发布。
v1.7.0	2024.07.26	v1.7.0 版本发布。
v1.8.0	2024.10.12	v1.8.0 版本发布。

1.8.0 更新内容

- 开放 bmapi 接口，支持视频编解码，图片编解码
- 支持 gstreamer，视频编解码，图片编解码…..
- ffmpeg 编码支持 yuvj420p 格式
-

1.7.0 更新内容

- opencv 的转码例子 videotranscode 支持多线程演示
- jpu 支持在任何 heap 上操作
-

1.6.0 更新内容

- base64 支持
- 2DE 升级
-

1.5.1 更新内容

- 第一次正式公开发布
- opencv 支持 vpu 编码/解码
- ffmpeg 支持 vpu 编码/解码
- opencv 支持 jpu 编码/解码
- ffmpeg 支持 jpu 编码/解码

CHAPTER 2

多媒体用户手册

2.1 SOPHGO 多媒体框架介绍

2.1.1 简介

本文档所述多媒体框架的描述对象为算能的算丰 BM1688 产品系列，目前该产品系列仅包括 BM1688。其中 1）本文中所有关于视频硬件编码的内容均只针对 BM1688 而言；2）本文中提到的 Opencv 中的 bmcv 名字空间下的函数，仅针对 BM1688 版本产品而言。

本文档所述多媒体框架的覆盖范围包括 BM1688 产品系列中的视频解码 VPU 模块、视频编码 VPU 模块、图像编码 JPU 模块、图像解码 JPU 模块、图像处理模块 VPSS。这些模块的功能封装到 FFMPEG 和 OPENCV 开源框架中，客户可以根据自己的开发习惯，选择 FFMPEG API 或者 OPENCV API。其中图像处理模块，我们还单独提供了算能自有的 BMCV API 底层接口，这部分接口有专门的文档介绍，可以参考《BMCV User Guide》，本文档不再详细介绍，仅介绍这三套 API 之间的层级关系及如何互相转换。

OPENCV，FFMPEG 和 BMCV 这三套 API 在功能上是子集的关系，但有少部分不能全部包含，下面的括号中进行了特别标注。

1）BMCV API 包含了所有能用硬件加速的图像处理加速接口（这里图像处理硬件加速，包括硬件图像处理 VPSS 模块加速，以及借用其他硬件模块实现的图像处理功能）

2）FFMPEG API 包含了所有硬件加速的视频/图像编解码接口，所有软件支持的视频/图像编解码接口（即所有 FFMPEG 开源支持的格式），通过 bm_scale filter 支持的部分硬件加速的图像处理接口（这部分图像处理接口，仅包括用硬件图像处理 VPSS 模块加速的缩放、crop、padding、色彩转换功能）

3）OPENCV API 包含了所有 FFMPEG 支持的硬件及软件视频编解码接口（视频底层通过 FFMPEG 支持，这部分功能完全覆盖），硬件加速的 JPEG 编解码接口，软件支持的其他所有图像编解码接口（即所有 OPENCV 开源支持的图像格式），部分硬件加速的图像处理接

口（指用图像处理 VPSS 模块加速的缩放、crop、padding、色彩转换功能），所有软件支持的 OPENCV 图像处理功能。

这三个框架中，BMCV 专注于图像处理功能，且能用 BM1688 硬件加速的部分；FFMPEG 框架强于图像和视频的编解码，几乎所有格式都可以支持，只是是否能用硬件加速的区别；OPENCV 框架强于图像处理，各种图像处理算法最初都先集成到 OPENCV 框架中，而视频编解码通过底层调用 FFMPEG 来实现。

因为 BMCV 仅提供了图像处理接口，因此 FFMPEG 或者 OPENCV 框架中，客户一般会选择其中一个作为主框架进行开发。这两个框架，从功能抽象上来说，OPENCV 的接口要更加简洁，一个接口就可以实现一次视频编解码操作；从性能上说，这两个的性能是完全一致的，几乎没有差别，在视频编解码上，OPENCV 只是对 FFMPEG 接口的一层封装；从灵活性上说，FFMPEG 的接口更加分离，可插入的操作粒度更细。最重要的，客户还是要根据自己对于某个框架的熟悉程度来做选择，只有深入了解，才能把框架用好。

这三个框架层级关系如图所示

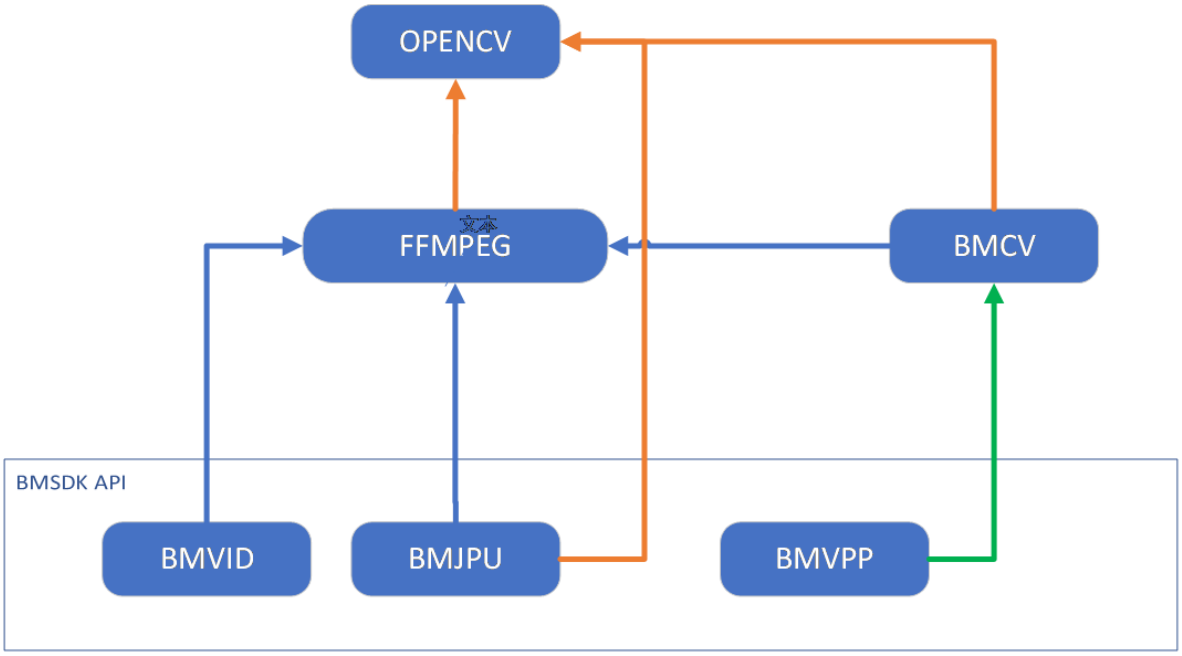


图 1 OPENCV/FFMPEG/BMCV 与 BMSDK 之间的层级调用关系

在很多应用场景下，需要用到某个框架下的特殊功能，因此在第 4 节中给出了三个框架之间灵活转换的方案。这种转换是不需要发生大量数据拷贝的，对性能几乎没有损失。

2.1.2 BM1688 硬件加速功能

本节给出了多媒体框架中硬件加速模块能支持的功能。其中硬件加速模块包括视频解码 VPU 模块，视频编码 VPU 模块，图像编解码 JPU 模块，图像处理 VPSS 模块。

需要特别注意，这里只列出能够用硬件加速的能力，以及典型场景下的性能估计值。更详细的性能指标参考 BM1688 产品规格书。

视频编解码

BM1688 产品支持 H264（AVC），HEVC 视频格式的硬件解码加速，最高支持到 4K 视频的实时解码。支持 H264(AVC), HEVC 视频格式的硬件编码，最高支持到 HD(1080p) 视频的实时编码。

视频解码的速度与输入视频码流的格式有很大关系，不同复杂度的码流的解码速度有比较大的波动，比如码率、GOP 结构，分辨率等，都会影响到具体的测试结果。一般来说，针对视频监控应用场景，BM1688 产品单芯片可以支持到 16 路 1080p30 高清实时解码。

视频编码的速度与编码的配置参数有很大关系，不同的编码配置下，即使相同的视频内容，编码速度也不是完全相同的。一般来说，BM1688 产品单芯片最高可以支持到 10 路 1080p30 高清实时编码。

图像编解码

BM1688 产品支持 JPEG baseline 格式的硬件编/解码加速。注意，仅支持 JPEG baseline 档次的硬件编解码加速，对于其他图片格式，包括 JPEG2000, BMP, PNG 以及 JPEG 标准的 progressive, lossless 等档次均自动采用软解支持。在 opencv 框架中，这种兼容支持对于客户是透明的，客户应用开发时无需特别处理。

图像硬件编解码的处理速度和图像的分辨率、图像色彩空间（YUV420/422/444）有比较大的关系，一般而言，对于 1920x1080 分辨率的图片，色彩空间为 YUV420 的，单芯片 1080p 图像硬件编解码可以达到 480fps 左右。

图像处理

BM1688 产品有专门的视频处理 VPSS 单元对图像进行硬件加速处理。支持的图像操作有色彩转换、图像缩放、图像切割 crop、图像拼接 stitch 功能。最大支持到 8k 图像输入。对于 VPSS 不支持的一些常用复杂图像处理功能，如线性变换 ax+b，直方图等, 我们在 BMCV API 接口中，利用其他硬件单元做了特殊的加速处理。

2.1.3 硬件内存分类

在后续的讨论中，内存同步问题是应用调试中经常会遇到的，比较隐蔽的问题。我们通常统一用设备内存和系统内存来称呼这两类内存间的同步。

SOC 模式，是指用 BM1688 芯片中的处理器作为主控 CPU，BM1688 产品独立运行应用程序。典型的产品有 SE9、SM9 模组。在这类模式下，采用 Linux 系统下的 ION 内存对设备内存进行管理。在 SOC 模式下，设备内存指 ION 分配的物理内存，系统内存其实是 cache，这里的命名只是为了和 PCIE 模式保持一致。从系统内存（cache）到设备内存，称为 Upload 上传（实质是 cache flush）；从设备内存到系统内存（cache），称为 Download 下载（实质是 cache invalidation）。在 SOC 模式下，设备内存和系统内存最终操作的其实是同一块物理内存，大部分时间，操作系统会自动对其进行同步，这也导致内存没有及时同步时的现象更加隐蔽和难以复现。

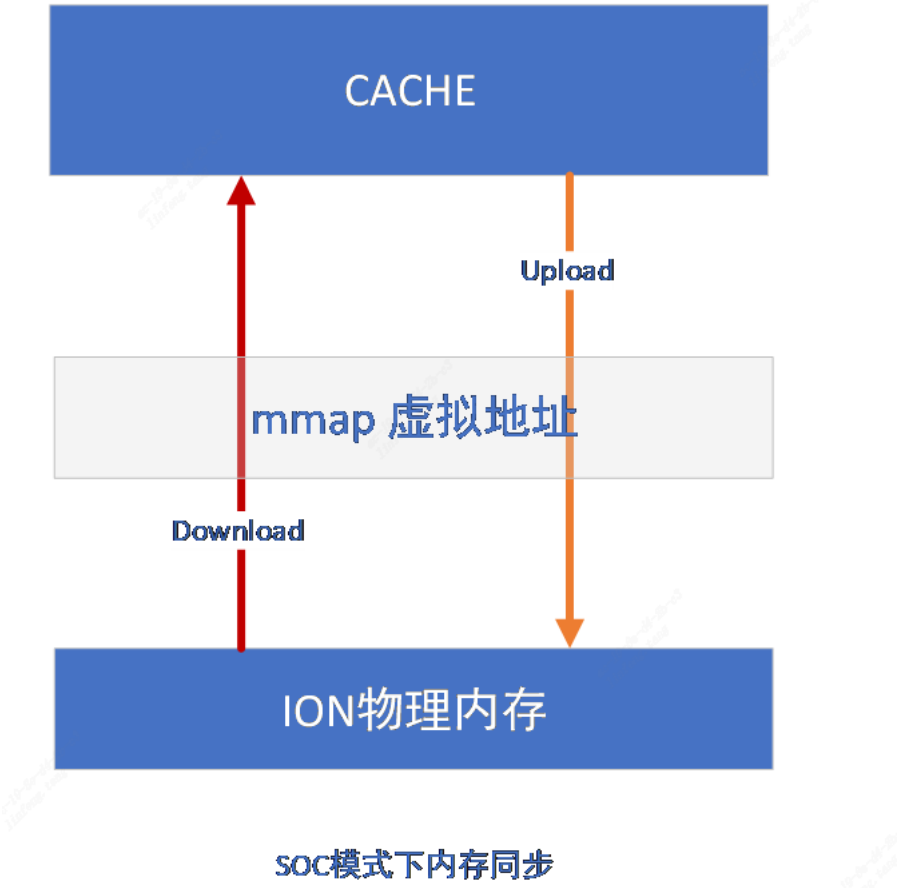


图 2 内存同步模型

FFMPEG 和 OPENCV 两个框架都提供了内存同步操作的函数。而 BMCV API 只面向设备内存操作，因此不存在内存同步的问题，在调用 BMCV API 的时候，需要将数据在设备内存准备好。

在 OPENCV 框架中，部分函数的形参中就提供了 update 的标志位，当标志位设置 true 的时候，函数内部会自动进行内存同步操作。这部分可以参考后续的第二章第 3 节的 API 介绍。用户也可以通过 bmcv::downloadMat() 和 bmcv::uploadMat() 两个函数，主动控制内存同步。

同步的基本原则是：a) opencv 原生函数中，yuv Mat 格式下设备内存中的数据永远是最新的，RGB Mat 格式下系统内存中的数据永远是最新的 b) 当 opencv 函数向 BMCV API 切换的时候，根据上一个原则，将最新数据同步到设备内存中；反之，从 BMCV API 向 opencv 函数切换的时候，在 RGB Mat 下将最新数据同步到系统内存中。c) 在不发生框架切换的时候，要尽量减少内存同步的操作。频繁的内存同步操作会明显降低性能。

在常规 FFMPEG 框架中，有两类称之为软（常规）和硬（hwaccel）的 codec API 和 filter API。这两套 API 的框架都可以支持 BM1688 的硬件视频编解码和硬件图像 filter，从这个角度上说，所谓的软解码和硬解码在底层性能上是完全一样的，只是在使用习惯上的区别。软 codec/filter API 的使用方式和通常 ffmpeg 内置 codec 完全一致，硬 codec/filter API 要用-hwaccel 来指定使能 bmcodec 专用硬件设备。当在软 codec API 和 filter API 中，通过 av_dict_set 传入标志参数 “is_dma_buffer” 或者” zero_copy”，来控制内部 codec 或 filter 是否将设备内存数据同步到系统内存中，具体参数使用可以用 ffmpeg -h 来查看。当后续直接衔接硬件处理的时候，通常不需要将设备内存数据同步到系统内存中。

在 hwaccel codec API 和 filter API 中，内存默认只有设备内存，没有分配系统内存。如果需要内存同步，则要通过 hwupload 和 hwdownload filter 来完成。

综上所述，OPENCV 和 FFMPEG 框架都对内存同步提供了支持，应用可以根据自己的使用习惯选择相应的框架，对数据同步进行精准控制。BMCV API 则始终工作在设备内存上。

2.1.4 框架之间转换

在应用开发中，总会碰到一些情况下，某个框架无法完全满足设计需求。这时就需要在各种框架之间快速切换。BM1688 多媒体框架对其提供了支持，可以满足这种需求，并且这种切换是不发生数据拷贝的，对于性能几乎没有影响。

FFMPEG 和 OPENCV 转换

FFMPEG 和 OPENCV 之间的转换，主要是数据格式 AVFrame 和 cv::Mat 之间的格式转换。

当需要 FFMPEG 和 OPENCV 配合解决的时候，推荐使用常规非 HWAccel API 的通路，目前 OPENCV 内部采用是这种方式，验证比较完备。

FFMPEG AVFrame 转到 OPENCV Mat 格式如下。

1. AVFrame * picture;

2. 中间经过 ffmpeg API 的一系列处理，比如 avcodec_decode_video2 或者 avcodec_receive_frame，然后将得到的结果转成 Mat

4. card_id 为进行 ffmpeg 硬件加速解码的设备序号，在常规 codec API 中，通过 av_dict_set 的 sophon_idx 指定，或者 hwaccel API 中，在 hwaccel 设备初始化的时候指定， soc 模式下默认为 0

5. cv::Mat ocv_frame(picture, card_id);

6. 或可以通过分步方式进行格式转换

7. cv::Mat ocv_frame;

8. ocv_frame.create(picture, card_id);

9. 然后可以用 ocv_frame 进行 opencv 的操作，此时 ocv_frame 格式为 BM1688 扩展的 yuv mat 类型，如果后续想转成 opencv 标准的 bgr mat 格式，可以按下列操作。

10. 注意：这里就有内存同步的操作， 如果没有设置，ffmpeg 默认是在设备内存中的，如果 update=false, 那么转成 bgr 的数据也一直在设备内存中，系统内存中为无效数据，如果 update=true，则设备内存同步到系统内存中。如果后续还是硬件加速处理的话，可以 update=false, 这样可以提高效率，当需要用到系统内存数据的时候，显式调用 bmcv::downloadMat() 来同步即可。

11. cv::Mat bgr_mat;

12. cv::bmcv::toMAT(ocv_frame, bgr_mat, update);

13. 最后 AVFrame *picture 会被 Mat ocv_frame 释放，因此不需要对 picture 进行 av_frame_free() 操作。如果希望外部调用 av_frame_free 来释放 picture，则可以加上 card_id = card_id | BM_MAKEFLAG(UMatData::AVFRAME_ATTACHED,0,0), 该标准表明 AVFrame 的创建和释放由外部管理

14. ocv_frame.release();

15. picture = nullptr;

OPENCV Mat 转成 FFMPEG AVFrame 的情况比较少见，因为几乎所有需要的 FFMPEG 操作都在 opencv 中有对应的封装接口。比如：ffmpeg 解码在 opencv 有 videoCapture 类，ffmpeg 编码在 opencv 中有 videoWriter 类，ffmpeg 的 filter 操作对应的图像处理在 opencv 中有 bmcv 名字空间下的接口以及丰富的原生图像处理函数。

一般来说，opencv Mat 转成 FFMPEG AVFrame，指的是 yuv Mat。在这种情况下，可以按下进行转换。

1. 创建 yuv Mat，如果 yuv Mat 已经存在，可以忽略此步.card_id 为 BM1688 设备序号，soc 模式下默认为 0

2. AVFrame * f = cv::av::create(height, width, AV_PIX_FMT_YUV420P, NULL, 0, -1, NULL, NULL, AVCOL_SPC_BT709, AVCOL_RANGE_MPEG, card_id);

3. cv::Mat image(f, card_id);

4. do something in opencv

5. AVFrame * frame = image.u->frame;

6. call FFMPEG API

7. 注意：在 ffmpeg 调用完成前，必须保证 Mat image 没有被释放，否则 AVFrame 会和 Mat image 一起释放。如果需要将两个的声明周期分离开来，则上面的 image 声明要改成如下格式。

8. cv::Mat image(f, card_id | BM_MAKEFLAG(UMatData::AVFRAME_ATTACHED, 0, 0));

9. 这样 Mat 就不会接管 AVFrame 的内存释放工作

FFMPEG 和 BMCV API 转换

FFMPEG 经常需要和 BMCV API 搭配使用，因此 FFMPEG 和 BMCV 之间的转换是比较频繁的。为此我们专门给了一个例子 ff_bmcv_transcode，该例子可以在 SDK 发布包里找到。

ff_bmcv_transcode 例子演示了用 ffmpeg 解码，将解码结果转换到 BMCV 下进行处理，然后再转换回到 ffmpeg 进行编码的过程。FFMPEG 和 BMCV 之间的互相转换可以参考 ff_avframe_convert.cpp 文件中的 avframe_to_bm_image() 和 bm_image_ to_avframe() 函数。

OPENCV 和 BMCV API 转换

OPENCV 和 BMCV API 之间的转换，专门在 opencv 扩展的 bmcv 名字空间下提供了专门的转换函数。

OPENCV Mat 转换到 BMCV bm_image 格式。

1. cv::Mat m(height, width, CV_8UC3, card_id);

2. opencv 操作

3. bm_image bmcv_image;

4. 这里 update 用来控制内存同步，是否需要内存同步取决于前面的 opencv 操作，如果前面的操作都是用硬件加速完成，设备内存中就是最新数据，就没必要进行内存同步，如果前面的操作调用了 opencv 函数，没有使用硬件加速（后续 opencv 章节 6.2 中提到了哪些函数采用了硬件加速），对于 bgr mat 格式就需要做内存同步。

5. 也可以在调用下面函数之前，显式的调用 cv::bmcv::uploadMat(m) 来实现内存同步

6. cv::bmcv::toBMI(m, &bmcv_image, update);

7. 使用 bmcv_image 就可以进行 bmcv api 调用，调用期间注意保证 Mat m 不能被释放，因为 bmcv_image 使用的是 Mat m 中分配的内存空间. handle 可以通过 bm_image_get_handle() 获得

8. 释放：必须调用此函数，因为在 toBMI 中 create 了 bm_image, 否则会有内存泄漏

9. bm_image_destroy(bmcv_image);

10. m.release();

由 BMCV bm_image 格式转换到 OPENCV Mat 有两种方式，一种是会发生数据拷贝，这样 bm_image 和 Mat 之间相互独立，可以分别释放，但是有性能损失；一种是直接引用 bm_image 内存，性能没有任何损失。

1. bm_image bmcv_image;

2. 调用 bmcv API 给 bmcv_image 分配内存空间，并进行操作

3. Mat m_copy, m_nocopy;

4. 下面接口将发生内存数据拷贝，转换成标准 bgr mat 格式.

5. update 控制内存同步，也可以在调用完这个函数后用 bmcv::downloadMat() 来控制内存同步

6. csc_type 是控制颜色转换系数矩阵，控制不同 yuv 色彩空间转换到 bgr

7. cv::bmcv::toMAT(&bmcv_image, m_copy, update, csc_type);

8. 下面接口接口将直接引用 bm_image 内存 (nocopy 标志位 true), update 仍然按照之前的描述，

9. 选择是否同步内存。 在后续 opencv 操作中，必须保证 bmcv_image 没有释放，因为 mat 的内存

10. 直接引用自 bm_image cv::bmcv::toMAT(&bmcv_image, &m_nocopy, AVCOL_SPC_BT709, AVCOL_RANGE_MPEG, NULL, -1, update, **true**);

11. 进行 opencv

2.2 SOPHGO OpenCV 使用指南

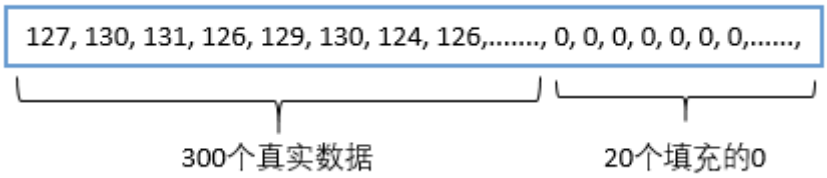
2.2.1 OpenCV 简介

BM1688 系列芯片中的多媒体、BMCV 模块可以加速对图片和视频的处理：

1) 多媒体模块：硬件加速 JPEG 编码解码和 Video 编解码操作。
2) BMCV 模块：包含 TPU、VPSS、IVE、GDC 等硬件模块。对 resize、color conversion、crop、split、linear transform、nms、sort、rgb2gray、mean、scale、int8tofloat32 等操作实现硬件加速。

为了方便客户使用芯片上的硬件模块加速图片和视频的处理，提升应用 OpenCV 软件性能，算能修改了 OpenCV 库，在其内部调用硬件模块进行 Image 和 Video 相关的处理。

算能当前 OpenCV 的版本为 4.1.0，除了以下几个算能自有的 API 以外，其它的所有 API 与 OpenCV API 都是一致的。



在 SOC 模式下，由于填充了多余的 0 值，Mat 对象中存储数据的 data 变量不能直接传递给 BMRuntime 库的 API 做推理，否则会降低模型的准确率。请在最后一次 BMCV 做变换的时候，将 stride 设置为非对齐方式，多余的 0 会被自动清除掉。

2.2.2 数据结构扩展说明

OpenCV 内置标准处理的色彩空间为 BGR 格式，但是很多情况下，对于视频、图片源，直接在 YUV 色彩空间上处理，可以节省带宽和避免不必要的 YUV 和 RGB 之间的互相转换。因此 SOPHGO Opencv 对于 Mat 类进行了扩展。

1) 在 Mat.UMatData 中，引入了 AVFrame 成员，扩展支持各种 YUV 格式。其中 AVFrame 的格式定义与 FFMPEG 中的定义兼容
2) 在 Mat.UMatData 中增加了 fd，addr（soc 模式下）的定义，分别表示对应的内存管理句柄和物理内存地址
3) 在 Mat 类中增加了 fromhardware 变量，标识当前的视频、图片解码是由硬件或是软件计算完成的，开发者在程序开发过程中无需考虑该变量的值。

2.2.3 API 扩展说明

bool VideoCapture::get_resampler(int *den, int *num)

函数原型	bool VideoCapture::get_resampler(int *den, int *num)
功能	取视频的采样速率。如 den=5, num=3 表示每 5 帧中有 2 帧被丢弃
输入参数	int *den –采样速率的分母
	int *num –采样速率的分子
输出参数	无
返回值	true –执行成功 false - 执行失败
说明	此接口将废弃。推荐用 double VideoCapture::get(CAP_PROP_OUTPUT_SRC) 接口。

bool VideoCapture::set_resampler(int den, int num)

函数原型	bool VideoCapture::set_resampler(int den, int num)
功能	置视频的采样速率。如 den=5, num=3，表示每 5 帧中有 2 帧被丢弃。
输入参数	int den –采样速率的分母
	int num –采样速率的分子
输出参数	无
返回值	true –执行成功 false - 执行失败
说明	此接口将废弃。推荐用 bool VideoCapture::set(CAP_PROP_OUTPUT_SRC, double resampler) 接口。

double VideoCapture::get(CAP_PROP_TIMESTAMP)

函数原型	double VideoCapture::get(CAP_PROP_TIMESTAMP)
功能	提供当前图片的时间戳，时间基数取决于在流中给出的时间基数
输入参数	CAP_PROP_TIMESTAMP –特定的枚举类型指示获取时间戳，此类型由 Sophgo 定义
输出参数	无
返回值	在使用前先将返回值转成 unsigned int64 数据类型
	0x8000000000000000L-No AV PTS value
	other-AV PTS value

double VideoCapture::get(CAP_PROP_STATUS)

函数原型	double VideoCapture::get(CAP_PROP_STATUS)
功能	该函数提供了一个接口，用于检查视频抓取的内部运行状态
输入参数	CAP_PROP_STATUS –枚举类型，此类型由 Sophgo 定义
输出参数	无
返回值	在使用返回值前请将转换成 int 类型
	0 视频抓取停止，暂停或者其他无法运行的状态
	1 视频抓取正在进行
	2 视频抓取结束

bool VideoCapture::set(CAP_PROP_OUTPUT_SRC, double resampler)

函数原型	double VideoCapture::get(CAP_PROP_OUTPUT_SRC, double resampler)
功能	设置 YUV 视频的采样速率。如 resampler 为 0.4，表示每 5 帧中保留 2 帧，有 3 帧被丢弃
输入参数	CAP_PROP_OUTPUT_SRC–枚举类型，此类型由 Sophgo 定义
	double resampler –采样速率
输出参数	无
返回值	true 执行成功
	false 执行失败

double VideoCapture::get(CAP_PROP_OUTPUT_SRC)

函数原型	double VideoCapture::get(CAP_PROP_OUTPUT_SRC)
功能	取视频的采样速率。
输入参数	CAP_PROP_OUTPUT_SRC - 特定的枚举类型，指视频输出，此类型由 SOPHGO 定义
输出参数	无
返回值	采样率数值

bool VideoCapture::set(CAP_PROP_OUTPUT_YUV, double enable)

函数原型	bool VideoCapture::set(CAP_PROP_OUTPUT_YUV, double enable)
功能	开或者关闭 YUV 格式的 frame 输出。BM1688 系列下 YUV 格式为 I420
输入参数	CAP_PROP_OUTPUT_YUV - 特定的枚举类型，指 YUV 格式的视频 frame 输出，此类型由 SOPHGO 定义; double enable - 操作码，1 表示打开，0 表示关闭
输出参数	无
返回值	true：执行成功 false：执行失败

double VideoCapture::get(CAP_PROP_OUTPUT_YUV)

函数原型	double VideoCapture::get(CAP_PROP_OUTPUT_YUV)
功能	取 YUV 视频 frame 输出的状态。
输入参数	CAP_PROP_OUTPUT_YUV - 特定的枚举类型，指 YUV 格式的视频 frame 输出，此类型由 SOPHGO 定义。
输出参数	无
返回值	YUV 视频 frame 输出的状态。1 表示打开，0 表示关闭。

bm_status_t bmcv::toBMI(Mat &m, bm_image *image, bool update = true)

函数原型	bm_status_t bmcv::toBMI(Mat &m, bm_image *image, bool date = true)
功能	OpenCV Mat 对象转换成 BMCV 接口中对应格式的 bm_image 数据对象，本接口直接引用 Mat 的数据指针，不会发生 copy 操作。本接口仅支持 1N 模式
输入参数	Mat& m –Mat 对象，可以为扩展 YUV 格式或者标准 OpenCV BGR 格式; bool update –是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	bm_image *image –对应格式的 BMCV bm_image 数据对象
返回值	BM_SUCCESS(0)：执行成功其他：执行失败
说明	目前支持压缩格式、Gray、NV12、NV16，YUV444P、YUV422P、YUV420P、BGR separate、BGR packed、CV_8UC1 的格式转换

bm_status_t bmcv::toBMI(Mat &m, Mat &m1, Mat &m2, Mat &m3, bm_image *image, bool update = true)

函数原型	bm_status_t bmcv::toBMI(Mat &m, Mat &m1, Mat &m2, Mat &m3, bm_image *image, bool update = true)
功能	OpenCV Mat 对象转换成 BMCV 接口中对应格式的 bm_image 数据对象，本接口直接引用 Mat 的数据指针，不发生 copy 操作。本接口针对 BMCV 的 4N 模式。要求所有 Mat 的输入图像格式一致, 仅对 BM1688 有效
输入参数	Mat &m - 4N 中的第 1 幅图像，扩展 YUV 格式或者标准 OpenCV BGR 格式。 Mat &m1 - 4N 中的第 2 幅图像，扩展 YUV 格式或者标准 OpenCV BGR 格式。 Mat &m2 - 4N 中的第 3 幅图像，扩展 YUV 格式或者标准 OpenCV BGR 格式。 Mat &m3 - 4N 中的第 4 幅图像，扩展 YUV 格式或者标准 OpenCV BGR 格式。 bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	bm_image *image - 对应格式的 BMCV bm_image 数据对象，其中包含 4 个图像数据
返回值	BM_SUCCESS(0)：执行成功其他：执行失败
说明	目前支持压缩格式、Gray、NV12、NV16，YUV444P、YUV422P、YUV420P、BGR separate、BGR packed、CV_8UC1 的格式转换

bm_status_t bmcv::toMAT(Mat &in, Mat &m0, bool update=true)

函数原型	bm_status_t bmcv::toMAT(Mat &in, Mat &m0, bool update = true)
功能	输入的 MAT 对象, 可以为各种 YUV 或 BGR 格式, 转换成 BGR packet 格式的 MAT 对象输出
输入参数	Mat &in - 输入的 MAT 对象，可以为各种 YUV 格式或者 BGR 格式; bool update –是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	Mat &m0 - 输出的 MAT 对象，转成标准 OpenCV 的 BGR 格式
返回值	BM_SUCCESS(0)：执行成功其他：执行失败
说明	目前支持压缩格式、Gray、NV12、NV16，YUV444P、YUV422P、YUV420P、BGR separate、BGR packed、CV_8UC1 到 BGR packed 格式转换。在 YUV 格式下，会自动根据 AVFrame 结构体中 colorspace,color_range 信息选择正确的色彩转换矩阵。

bm_status_t toMAT(bm_image *image, Mat &m, int color_space, int color_range, void* vaddr = NULL, int fd0 = -1, bool update = true, bool nocopy = true)

函数原型	bm_status_t bmcv::toMAT(bm_image *image, Mat &m, int color_space, int color_range, void* vaddr=NULL, int fd0=-1, bool update=true, bool nocopy=true)
功能	输入的 bm_image 对象，当 nocopy 为 true 时，直接复用设备内存转成 Mat 格式，当 nocopy 为 false 时，行为类似 3.13toMAT 接口，1N 模式。
输入参数	bm_image *image - 输入的 bm_image 对象，可以为各种 YUV 格式或者 BGR 格式; Int color_space –输入 image 的色彩空间，可以为 AVCOL_SPC_BT709 或 AVCOL_SPC_BT470，详见 FFMPEG pixfmt.h 定义; Int color_range –输入 image 的色彩动态范围，可以为 AVCOL_RANGE_MPEG 或 AVCOL_RANGE_JPEG，详见 FFMPEG pixfmt.h 定义; Void* vaddr –输出 Mat 的系统虚拟内存指针，如果已分配，输出 Mat 直接使用该内存作为 Mat 的系统内存。如果为 NULL，则 Mat 内部自动分配; Int fd0 –输出 Mat 的物理内存句柄，如果为负，则使用 bm_image 内的设备内存句柄，否则使用 fd0 给定的句柄来 mmap 设备内存; bool update -是否需要同步 cache 或内存。如果为 true，则转换完成后同步 cache bool nocopy –如果是 true，则直接引用 bm_image 的设备内存，如果为 false，则转换成标准 BGR Mat 格式
输出参数	Mat &m - 输出的 MAT 对象，当 nocopy 为 true 时，输出标准 BGR 格式或扩展的 YUV 格式的 Mat；当 nocopy 为 false 时，转成标准 OpenCV 的 BGR 格式。
返回值	BM_SUCCESS(0)：执行成功其他：执行失败
说明	1.no copy 方式只支持 1N 模式，4N 模式因为内存排列方式，不能支持引用 2. 在 nocopy 为 false 的情况下，会自动根据参数 colorspace,color_range 信息选择正确的色彩转换矩阵进行色彩转换。 3. 如果系统内存 vaddr 来自于外部，那么外部需要来管理这个内存的释放，Mat 释放的时候不会释放该内存

bm_status_t bmcv::toMAT(bm_image *image, Mat &m0, bool update = true, csc_type_t csc = CSC_MAX_ENUM)

函数原型	bm_status_t bmcv::toMAT(bm_image *image, Mat &m0, bool update=true, csc_type_t csc=CSC_MAX_ENUM)
功能	输入的 bm_image 对象，可以为各种 YUV 或 BGR 格式，转换成 BGR 格式的 MAT 对象输出，1N 模式
输入参数	bm_image *image - 输入的 bm_image 对象，可以为各种 YUV 格式或者 BGR 格式; bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache csc_type_t csc –色彩转换类型，仅当输入 bm_image 为 YUV 格式时需要 csc 转换，默认 type 为 YCbCr2RGB_BT601
输出参数	Mat &m0 - 输出的 MAT 对象，转成标准 OpenCV 的 BGR 格式
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::toMAT(bm_image *image, Mat &m0, Mat &m1, Mat &m2, Mat &m3, bool update=true, csc_type_t csc=CSC_MAX_ENUM)

函数原型	bm_status_t bmcv::toMAT(bm_image *image, Mat &m0, Mat &m1, Mat &m2, Mat &m3, bool update=true, csc_type_t csc=CSC_MAX_ENUM)
功能	输入的 bm_image 对象，可以为各种 YUV 或 BGR 格式，转换成 BGR 格式的 MAT 对象输出，4N 模式，仅在 BM1684 下有效
输入参数	bm_image *image - 输入的 4N 模式下的 bm_image 对象，可以为各种 YUV 格式或者 BGR 格式; bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache csc_type_t csc –色彩转换类型，仅当输入 bm_image 为 YUV 格式时需要 csc 转换，默认 type 为 YCbCr2RGB_BT601
输出参数	Mat &m0 - 输出的第一个 MAT 对象，转成标准 OpenCV 的 BGR 格式; Mat &m1 - 输出的第二个 MAT 对象，转成标准 OpenCV 的 BGR 格式; Mat &m2 - 输出的第三个 MAT 对象，转成标准 OpenCV 的 BGR 格式; Mat &m3 - 输出的第四个 MAT 对象，转成标准 OpenCV 的 BGR 格式
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::resize(Mat &m, Mat &out, bool update = true, int interpolation= BMCV_INTER_NEAREST)

函数原型	bm_status_t bmcv::resize(Mat &m, Mat &out, bool update = true, int interpolation = BMCV_INTER_NEAREST)
功能	输入的 MAT 对象，缩放到输出 Mat 给定的大小，输出格式为输出 Mat 指定的色彩空间，因为 MAT 支持扩展的 YUV 格式，因此本接口支持的色彩空间并不仅局限于 BGR packed。
输入参数	Mat &m - 输入的 Mat 对象，可以为标准 BGR packed 格式或者扩展 YUV 格式; bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache int interpolation –缩放算法，可为 NEAREST 或者 LINEAR 算法
输出参数	Mat &out - 输出的缩放后的 Mat 对象
返回值	BM_SUCCESS(0)：执行成功其他：执行失败
说明	支持 Gray、YUV444P、YUV420P、BGR/RGB separate、BGR/RGB packed、ARGB packed 格式缩放

bm_status_t bmcv::convert(Mat &m, Mat &out, bool update=true)

函数原型	bm_status_t bmcv::convert(Mat &m, Mat &out, bool update = true)
功能	实现两个 mat 之间的色彩转换，它与 toMat 接口的区别在于 toMat 只能实现各种色彩格式到 BGR packed 的色彩转换，而本接口可以支持 BGR packed 或者 YUV 格式到 BGR packed 或 YUV 之间的转换。
输入参数	Mat &m - 输入的 Mat 对象，可以为扩展的 YUV 格式或者标准 BGR packed 格式; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	Mat &out - 输出的色彩转换后的 Mat 对象，可以为 BGR packed 或者 YUV 格式。
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::convert(Mat &m, std::vector<Rect> &vrt, std::vector<Size> &vsz, std::vector<Mat> &out, bool update= true, csc_type_t csc=CSC_YCbCr2RGB_BT601, csc_matrix_t *matrix = nullptr, bmcv_resize_algorithm algorithm= BMCV_INTER_LINEAR)

函数原型	bm_status_t bmcv::convert(Mat &m, std::vector<Rect> &vrt, std::vector<Size> &vsz, std::vector<Mat> &out, bool update = true, csc_type_t csc=CSC_YCbCr2RGB_BT601, csc_matrix_t *matrix=nullptr, bmcv_resize_algorithm algorithm = BMCV_INTER_LINEAR)
功能	接口采用内置的 VPSS 硬件加速单元，集 crop,resize 和 csc 于一体。按给定的多个 rect 框，给定的多个缩放 size，将输入的 Mat 对象，输出到多个 Mat 对象中，输出为 OpenCV 标准的 BGR pack 格式或扩展 YUV 格式
输入参数	Mat &m - 输入的 Mat 对象，可以为扩展的 YUV 格式或者标准 BGR packed 格式; std::vector<Rect> &vrt - 多个 rect 框，输入 Mat 中的 ROI 区域。矩形框个数和 resize 个数应该相同; std::vector<Size> &vsz - 多个 resize 大小，与 vrt 的矩形框一一对应; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache csc_type_t csc –色彩转换矩阵，可以根据颜色空间指定合适的色彩转换矩阵; csc_matrix_t *matrix –当色彩转换矩阵不在列表中时，可以给出外置的用户自定义的转换矩阵; bmcv_resize_algorithm algorithm –缩放算法，可以为 Nearest 或者 Linear 算法
输出参数	std::vector<Mat> &out - 输出的缩放、crop 以及色彩转换后的标准 BGR pack 格式或 YUV 格式的 Mat 对象。
返回值	BM_SUCCESS(0)：执行成功其他：执行失败
说明	接口可以将 resize,crop,csc 三种操作在一步之内完成，效率最高。在可能的情况下，要尽可能的使用该接口提高效率

bm_status_t bmcv::convert(Mat &m, std::vector<Rect> &vrt, bm_image *out, bool update= true)

函数原型	bm_status_t bmcv::convert(Mat &m, std::vector<Rect> &vrt, bm_image *out, bool update= true)
功能	接口采用内置的 VPSS 硬件加速单元，集 crop,resize 和 csc 于一体。按给定的多个 rect 框，按照多个 bm_image 中指定的 size，将输入的 Mat 对象，输出到多个 bm_image 对象中，输出格式由 bm_image 初始化值决定。注意，bm_image 必须由调用者初始化好，并且个数和 vrt 一一对应。
输入参数	Mat &m - 输入的 Mat 对象，可以为扩展的 YUV 格式或者标准 BGR packed 格式; std::vector<Rect> &vrt - 多个 rect 框，输入 Mat 中的 ROI 区域。矩形框个数和 resize 个数应该相同; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	bm_image *out - 输出的缩放、crop 以及色彩转换后的 bm_image 对象，输出色彩格式由 bm_image 初始化值决定。同时该 bmimage 参数包含的初始化的 size、色彩信息也作为输入信息，用于处理。
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::warpAffine(InputArray src, OutputArray dst, InputArray M0, Size dsize, int flags = 1, int borderMode = 0, bool update = true)

函数原型	bm_status_t bmcv::warpAffine(InputArray src, OutputArray dst, InputArray M0, Size dsize, int flags = 1, int borderMode = 0, bool update = true)
功能	接口采用内置的 TPU 硬件加速单元, 将输入的 Mat 对象, 按给定的坐标变换矩阵进行旋转, 平移, 缩放等实现图像的仿射变换, 结果输出到 Mat 对象中。
输入参数	InputArray src - 仅支持输入为 Mat 的对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; InputArray M0 - 仅支持输入为 Mat 的对象, 坐标变换矩阵是一个 6 点的 2x3 矩阵; Size dsize - 输出图的大小 int flags - 插值方法, 当 flag = 0 时, 使用 nearest, 当 flag = 1 时, 使用 bilinear; int borderMode - 像素外推方法, 当 borderMode = 0 时, 边缘像素设置为 0, 当 borderMode = 1 时, 复制边缘像素 bool update - 是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	OutputArray dst - 输出的仿射变换后的 Mat 对象, 输出色彩格式为 BGR_PACKED。
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::rectangle(InputOutputArray _img, Point pt1, Point pt2, const Scalar& color, int thickness, bool update)

函数原型	bm_status_t bmcv::rectangle(InputOutputArray _img, Point pt1, Point pt2, const Scalar& color, int thickness, bool update)
功能	接口采用内置的 VPP 硬件加速单元, 在输入的 Mat 对象上按 pt1 和 pt2 定义的矩形大小和位置画矩形框, 该图像会被直接修改。
输入参数	InputOutputArray _img - 绘制矩形的图像, 输入只支持 Mat 对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; Point pt1 - 矩形左上角的坐标, 以 (x, y) 表示, 该点定义了矩形的一个顶点; Point pt2 - 矩形右下的坐标, 以 (x, y) 表示, 与 pt1 一起定义了矩形的大小和位置; const Scalar& color - 矩形的颜色, 由 BGR 值构成, 这里的 Scalar 类只接受 1 到 3 个参数, 对应图像的通道数 (3); int thickness(可选) - 矩形边框的厚度, 单位为像素, 默认值为 1, 若设置为负数 (通常为-1), 则绘制实心矩阵, 填充矩阵内部; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::rectangle(InputOutputArray img, Rect rec, const Scalar& color, int thickness, bool update)

函数原型	bm_status_t bmcv::rectangle(InputOutputArray img, Rect rec, const Scalar& color, int thickness, bool update)
功能	接口采用内置的 VPP 硬件加速单元, 在输入的 Mat 对象上按 rec 定义的矩形大小和位置画矩形框, 该图像会被直接修改。
输入参数	InputOutputArray _img - 绘制矩形的图像, 输入只支持 Mat 对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; Rect rec - 指定矩形左上角的坐标 x, y 和矩形的宽高, 指定矩形规格; const Scalar& color - 矩形的颜色, 由 BGR 值构成, 这里的 Scalar 类只接受 1 到 3 个参数, 对应图像的通道数 (3); int thickness(可选) - 矩形边框的厚度, 单位为像素, 默认值为 1, 若设置为负数 (通常为-1), 则绘制实心矩阵, 填充矩阵内部; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::rectangle(Mat &m, std::vector<Rect> &vrt, const Scalar& color, int thickness, bool update)

函数原型	bm_status_t bmcv::rectangle(Mat &m, std::vector<Rect> &vrt, const Scalar& color, int thickness, bool update)
功能	接口采用内置的 VPP 硬件加速单元, 在输入的 Mat 对象上按 vrt 定义的矩形大小和位置画矩形框, 该图像会被直接修改。
输入参数	Mat &m - 绘制矩形的 Mat 对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; std::vector<Rect> &vrt - 多个 rect 框; const Scalar& color - 矩形的颜色, 由 BGR 值构成, 这里的 Scalar 类只接受 1 到 3 个参数, 对应图像的通道数 (3); int thickness(可选) - 矩形边框的厚度, 单位为像素, 默认值为 1, 若设置为负数 (通常为-1), 则绘制实心矩阵, 填充矩阵内部; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::bitwise_and(InputArray a, InputArray b, OutputArray c, bool update)

函数原型	bm_status_t bmcv::bitwise_and(InputArray a, InputArray b, OutputArray c, bool update)
功能	接口采用内置的 TPU 硬件加速单元, 对输入的两个大小相同的 Mat 对象对应像素值进行按位与操作。
输入参数	InputArray a - 第一张图像的 Mat 对象 (只支持 Mat 输入), 可以为扩展的 YUV 格式或者标准 BGR packed 格式; InputArray b - 第二张图像的 Mat 对象 (只支持 Mat 输入), 可以为扩展的 YUV 格式或者标准 BGR packed 格式; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	OutputArray c - Mat a 和 Mat b 按位与的输出对象, 只支持 Mat 对象
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::bitwise_or(InputArray a, InputArray b, OutputArray c, bool update)

函数原型	bm_status_t bmcv::bitwise_or(InputArray a, InputArray b, OutputArray c, bool update)
功能	接口采用内置的 TPU 硬件加速单元, 对输入的两个大小相同的 Mat 对象对应像素值进行按位或操作。
输入参数	InputArray a - 第一张图像的 Mat 对象 (只支持 Mat 输入), 可以为扩展的 YUV 格式或者标准 BGR packed 格式; InputArray b - 第二张图像的 Mat 对象 (只支持 Mat 输入), 可以为扩展的 YUV 格式或者标准 BGR packed 格式; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	OutputArray c - Mat a 和 Mat b 按位或的输出对象, 只支持 Mat 对象
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::bitwise_xor(InputArray a, InputArray b, OutputArray c, bool update)

函数原型	bm_status_t bmcv::bitwise_xor(InputArray a, InputArray b, OutputArray c, bool update)
功能	接口采用内置的 TPU 硬件加速单元, 对输入的两个大小相同的 Mat 对象对应像素值进行按位与操作。
输入参数	InputArray a - 第一张图像的 Mat 对象 (只支持 Mat 输入), 可以为扩展的 YUV 格式或者标准 BGR packed 格式; InputArray b - 第二张图像的 Mat 对象 (只支持 Mat 输入), 可以为扩展的 YUV 格式或者标准 BGR packed 格式; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	OutputArray c - Mat a 和 Mat b 按位异或的输出对象, 只支持 Mat 对象
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::absdiff(InputArray src1, InputArray src2, OutputArray dst, bool update)

函数原型	bm_status_t bmcv::absdiff(InputArray src1, InputArray src2, OutputArray dst, bool update)
功能	接口采用内置的 TPU 硬件加速单元, 对输入的两个大小相同的 Mat 对象对应像素值相减并取绝对值。
输入参数	InputArray src1 - 第一张图像的 Mat 对象 (只支持 Mat 输入), 可以为扩展的 YUV 格式或者标准 BGR packed 格式; InputArray src2 - 第二张图像的 Mat 对象 (只支持 Mat 输入), 可以为扩展的 YUV 格式或者标准 BGR packed 格式; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	OutputArray dst - Mat src1 和 Mat src2 按位与的输出对象, 只支持 Mat 对象
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::rotate(InputArray _src, OutputArray _dst, int rotateMode, bool update)

函数原型	bm_status_t bmcv::rotate(InputArray _src, OutputArray _dst, int rotateMode, bool update)
功能	接口采用内置的 VPP 和 TPU 硬件加速单元, 实现图像顺时针旋转 90 度, 180 度和 270 度。
输入参数	InputArray _src - 进行旋转的图像, 输入只支持 Mat 对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; int rotateMode - rotateMode = 0, 1, 2 分别对应顺时针旋转 90 度, 180 度, 270 度; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	OutputArray _dst - 旋转后的图像, 只支持 Mat 对象
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::threshold(InputArray _src, OutputArray _dst, unsigned char thresh, unsigned char max_value, int type, bool update)

函数原型	bm_status_t bmcv::threshold(InputArray _src, OutputArray _dst, unsigned char thresh, unsigned char max_value, int type, bool update)
功能	接口采用内置的 TPU 硬件加速单元, 实现图像阈值化处理。
输入参数	InputArray _src - 进行阈值化处理的图像, 输入只支持 Mat 对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; unsigned char thresh - 像素阈值, 取值范围为 0-255; unsigned char max_value - 阈值化操作后的像素最大值, 取值范围为 0-255; int type - 阈值化类型, 取值范围为 0-4, 分别对应 BM_THRESH_BINARY = 0, BM_THRESH_BINARY_INV, BM_THRESH_TRUNC, BM_THRESH_TOZERO, BM_THRESH_TOZERO_INV; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	OutputArray _dst - 阈值化后的图像, 只支持 Mat 对象
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::convertTo(InputArray _srcs, OutputArray _dsts, int _type, std::array<float, 3> alpha, std::array<float, 3> beta, bool update)

函数原型	bm_status_t bmcv::convertTo(InputArray _srcs, OutputArray _dsts, int _type, std::array<float, 3> alpha, std::array<float, 3> beta, bool update)
功能	接口采用内置的 TPU 硬件加速单元, 实现一个或多个图像像素线性变化, 具体数据关系为: y=kx+b。
输入参数	InputArray _src - 进行像素线性变换的图像, 输入支持 Mat 或 std::vector<Mat> 对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; int _type - 需要输出的矩阵类型, 如果是负值 (常用-1), 输出矩阵和输入矩阵类型相同; std::array<float, 3> alpha - 比例因子, 对应图像的 3 通道; std::array<float, 3> beta - 将输入数组元素按比例缩放后添加的值, 对应图像的 3 通道; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	OutputArray _dst - 像素线性变化后的图像, 支持 Mat 或 std::vector<Mat> 对象
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::convertTo(InputArray _srcs, OutputArray _dsts, int _type, float alpha, float beta, bool update)

函数原型	bm_status_t bmcv::convertTo(InputArray _srcs, OutputArray _dsts, int _type, float alpha, float beta, bool update)
功能	接口采用内置的 TPU 硬件加速单元, 实现一个图像像素线性变化, 具体数据关系为: y=kx+b。
输入参数	InputArray _src - 进行像素线性变换的图像, 输入只支持 Mat 对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; int _type - 需要输出的矩阵类型, 如果是负值 (常用-1), 输出矩阵和输入矩阵类型相同; float alpha - 比例因子, 图像所有通道的比例因子相同; float beta - 将输入数组元素按比例缩放后添加的值, 图像所有通道的值相同; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	OutputArray _dst - 像素线性变化后的图像, 只支持 Mat 对象
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::mosaic(Mat &m, std::vector<Rect> &vrt, int is_expand, bool update)

函数原型	bm_status_t bmcv::mosaic(Mat &m, std::vector<Rect> &vrt, int is_expand, bool update)
功能	接口采用内置的 VPP 硬件加速单元, 按给定的一个或多个 rect 框, 在输入的 Mat 对象上打一个或多个马赛克, 输出到原 Mat 对象上。
输入参数	Mat &m - 输入的 Mat 对象, 输出为加马赛克后的原始 Mat 对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; std::vector<Rect> &vrt - 多个 rect 框, 输入 Mat 中的 ROI 区域; int is_expand - 是否扩列。值为 0 时表示不扩列, 值为 1 时表示在原马赛克周围扩列一个宏块 (8 个像素); bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

bm_status_t bmcv::quantify(Mat &m, Mat &output, bool update)

函数原型	bm_status_t bmcv::quantify(Mat &m, Mat &output, bool update)
功能	接口采用内置的 TPU 硬件加速单元, 将输入的 Mat 对象的 float 类型数据转化成 int 类型 (舍入模式为小数点后直接截断), 并将小于 0 的数变为 0, 大于 255 的数变为 255。
输入参数	Mat &m - 输入的 Mat 对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache
输出参数	Mat &output - 数据类型转化后的图像, 只支持 Mat 对象
返回值	BM_SUCCESS(0)：执行成功其他：执行失败

void bmcv::uploadMat(Mat &mat)

函数原型	void bmcv::uploadMat(Mat &mat)
功能	cache 同步或者设备内存同步接口。当执行此函数时，cache 中内容会 flush 到实际内存中（SOC 模式）
输入参数	Mat &mat - 输入的需要内存同步的 mat 对象
输出参数	无
返回值	无
说明	合理调用本接口，可以有效控制内存同步的次数，仅在需要的时候调用。

void bmcv::downloadMat(Mat &mat)

函数原型	void bmcv::downloadMat(Mat &mat)
功能	cache 同步或者设备内存同步接口。当执行此函数时，cache 中内容会 invalidate（SOC 模式）
输入参数	Mat &mat - 输入的需要内存同步的 mat 对象
输出参数	无
返回值	无
说明	合理调用本接口，可以有效控制内存同步的次数，仅在需要的时候调用。

bm_status_t bmcv::stitch(std::vector<Mat> &in, std::vector<Rect>& srt, std::vector<Rect>& drt, Mat &out, bool update = true, bmcv_resize_algorithm algorithm = BMCV_INTER_LINEAR)

函数原型	bm_status_t bmcv::stitch(std::vector<Mat> &in, std::vector<Rect> &src, std::vector<Rect> &drt, Mat &out, bool update=true, bmcv_resize_alogrithm algorithm=BMCV_INTER_LINEAR)
功能	图像拼接，将输入的多个 Mat 按照按照给定的位置缩放并拼接到一个 Mat 中
输入参数	std::vector<Mat> &in –多个输入的 Mat 对象，可以为扩展的 YUV 格式或者标准 BGR pack 格式; std::vector<Rect> &src –对应每个 Mat 对象的显示内容框; std::vector<Rect> &drt –对应每个显示内容在目标 Mat 中的显示位置; bool update -是否需要同步 cache 或内存。如果为 true, 则转换完成后同步 cache bmcv_resize_algorithm algorithm –缩放算法, 可以为 Nearest 或者 Linear 算法
输出参数	Mat &out –输出拼接后的 Mat 对象，可以为 BGR packed 或者 YUV 格式
返回值	BM_SUCCESS(0)：执行成功其他：执行失败
说明	

void bmcv::print(Mat &m, bool dump = false)

函数原型	void bmcv::print(Mat &m, bool dump = false)
功能	调试接口，打印输入 Mat 对象的色彩空间，宽高以及数据。
输入参数	Mat &m - 输入的 Mat 对象，可以为扩展的 YUV 格式或者标准 BGR-packed 格式; bool dump - true 的时候打印 Mat 中的数据值，默认不打印。如果为 true，则会在当前目录下生成 mat_dump.bin 文件
输出参数	无
返回值	无
说明	当前支持 dump OpenCV 标准 BGRpacked 或者 CV_8UC1 数据,以及扩展的 NV12,NV16,YUV420P,YUV422P,GRAY,YUV444P 和 BGRSeparate 格式数据

void bmcv::print(bm_image *image, bool dump)

函数原型	void bmcv::print(bm_image *image, bool dump)
功能	调试接口，打印输入 bm_image 对象的色彩空间，宽高以及数据。
输入参数	bm_image *image - 输入的 bm_image 对象; bool dump - true 的时候打印 Mat 中的数据值，默认不打印，如果为 true，则会在当前目录下生成 BMI-“宽”x”高”.bin 文件
输出参数	无
返回值	无
说明	当前支持 dump BGR packed,NV12,NV16,YUV420P,YUV422P,GRAY,YUV444P 和 BGR Separate 格式的 bm_image 数据

void bmcv::dumpMat(Mat &image, const String &fname)

函数原型	void bmcv::dumpMat(Mat &image, const String &fname)
功能	调试接口，专门 dumpMat 的数据到指定命名的文件。功能同 3.23 的 dump 为 true 时的功能。
输入参数	Mat &image - 输入的 Mat 对象, 可以为扩展的 YUV 格式或者标准 BGR packed 格式; const String &fname –dump 的输出文件名
输出参数	无
返回值	无
说明	当前支持 dump OpenCV 标准 BGR packed 或者 CV_8UC1 数据，以及扩展的 NV12,NV16,YUV420P,YUV422P,GRAY,YUV444P 和 BGR Separate 格式数据

void bmcv::dumpBMImage(bm_image *image, const String &fname)

函数原型	void bmcv::dumpBMImage(bm_image *image, const String &fname)
功能	调试接口，专门 dump bm_image 的数据到指定命名的文件。功能同 3.25 的 dump 为 true 时的功能。
输入参数	bm_image *image - 输入的 bm_image 对象; const String &fname –dump 的输出文件名
输出参数	无
返回值	无
说明	当前支持 dump BGR packed, NV12,NV16,YUV420P,YUV422P,GRAY,YUV444P 和 BGR Separate 格式的 bm_image 数据

bool Mat::avOK()

函数原型	bool Mat::avOK()
功能	判断当前 Mat 是否为扩展的 YUV 格式
输入参数	无
输出参数	无
返回值	true –表示当前 Mat 为扩展的 YUV 格式 false –表示当前 Mat 为标准 OpenCV 格式
说明	接口和接口 3.21 3.22 downloadMat、uploadMat 结合起来，可以有效地管理内存同步。 一般 avOK 为 true 的 Mat，物理内存是最新的，而 avOK 为 false 的 Mat，其 cache 或者 host 内存中的数据是最新的。可以根据这个信息，决定是调用 uploadMat 还是 downloadMat。 如果一直在设备内存中通过硬件加速单元工作，则可以省略内存同步，仅在需要交换到系统内存中时再调用 downloadMat。

int Mat::avCols()

函数原型	int Mat::avCols()
功能	获取 YUV 扩展格式的 Y 的宽
输入参数	无
输出参数	无
返回值	返回扩展的 YUV 格式的 Y 的宽，如果为标准 OpenCV Mat 格式，返回 0

int Mat::avRows()

函数原型	int Mat::avRows()
功能	获取 YUV 扩展格式的 Y 的高
输入参数	无
输出参数	无
返回值	返回扩展的 YUV 格式的 Y 的高，如果为标准 OpenCV Mat 格式，返回 0

int Mat::avFormat()

函数原型	int Mat::avFormat()
功能	获取 YUV 格式信息
输入参数	无
输出参数	无
返回值	返回扩展的 YUV 格式信息，如果为标准 OpenCV Mat 格式，返回 0

int Mat::avAddr(int idx)

函数原型	int Mat::avAddr(int idx)
功能	获取 YUV 各分量的物理地址
输入参数	int idx –指定 YUV plane 的序号
输出参数	无
返回值	返回指定的 plane 的物理首地址，如果为标准 OpenCV Mat 格式，返回 0

int Mat::avStep(int idx)

函数原型	int Mat::avStep(int idx)
功能	获取 YUV 格式中指定 plane 的 line size
输入参数	int idx –指定 YUV plane 的序号
输出参数	无
返回值	指定的 plane 的 line size，如果为标准 OpenCV Mat 格式，返回 0

AVFrame* av::create(int height, int width, int color_format, void *data, long addr, int fd, int* plane_stride, int* plane_size, int color_space = AVCOL_SPC_BT709, int color_range = AVCOL_RANGE_MPEG, int id = 0)

函数原型	AVFrame* av::create(int height, int width, int clor_format, void *data, long addr, int fd, int* plane_stride, int* plane_size, int color_space = AVCOL_SPC_BT709, int color_range = AVCOL_RANGE_MPEG, int id = 0)
功能	AVFrame 的创建接口，允许外部创建系统内存和物理内存，创建的格式与 FFMPEG 下的 AVFrame 定义兼容
输入参数	int height –创建图像数据的高; int width –创建图像数据的宽; int color_format –创建图像数据的格式，详见 FFMPEG pixfmt.h 定义; void *data –系统内存地址，当为 null 时，表示该接口内部自己创建管理; long addr –设备内存地址; int fd –设备内存地址的句柄。如果为-1，表示设备内存由内部分配，反之则由 addr 参数给出。 int* plane_stride –图像数据每层的每行 stride 数组; int* plane_size –图像数据每层的大小; int color_space –输入 image 的色彩空间，可以为 AVCOL_SPC_BT709 或 AVCOL_SPC_BT470，详见 FFMPEG pixfmt.h 定义, 默认为 AVCOL_SPC_BT709; int color_range –输入 image 的色彩动态范围，可以为 AVCOL_RANGE_MPEG 或 AVCOL_RANGE_JPEG，详见 FFMPEG pixfmt.h 定义，默认为 AVCOL_RANGE_MPEG; int id –指定设备卡号以及 HEAP 位置的标志，详见 5.1，该参数默认为 0
输出参数	无
返回值	AVFrame 结构体指针
说明	1. 本接口支持创建以下图像格式的 AVFrame 数据结构: AV_PIX_FMT_GRAY8, AV_PIX_FMT_GBRP, AV_PIX_FMT_YUV420P, AV_PIX_FMT_NV12, AV_PIX_FMT_YUV422P horizontal, AV_PIX_FMT_YUV444P, AV_PIX_FMT_NV16 2. 当设备内存和系统内存均有外部给出时，在 soc 模式下外部要保证两者地址的匹配，即系统内存是设备内存映射出来的虚拟地址；当设备内存由外部给出，系统内存为 null 时，该接口内部会自动创建系统内存；当设备内存没有给出，系统内存也为 null 时，本接口内部会自动创建；当设备内存没有给出，系统内存由外部给出时，本接口创建失败

AVFrame* av::create(int height, int width, int id = 0)

函数原型	AVFrame* av::create(int height, int width, int id = 0)
功能	AVFrame 的简易创建接口，所有内存均由内部创建管理，仅支持 YUV420P 格式
输入参数	int height –创建图像数据的高; int width –创建图像数据的宽; int id –指定设备卡号以及 HEAP 位置的标志，详见 5.1，该参数默认为 0
输出参数	无
返回值	AVFrame 结构体指针
说明	本接口仅支持创建 YUV420P 格式的 AVFrame 数据结构

int av::copy(AVFrame *src, AVFrame *dst, int id)

函数原型	int av::copy(AVFrame *src, AVFrame *dst, int id)
功能	AVFrame 的深度 copy 函数，将 src 的有效图像数据拷贝到 dst 中
输入参数	AVFrame *src –输入的 AVFrame 原始数据指针; int id –指定设备卡号，详见 5.1
输出参数	AVFrame *dst –输出的 AVFrame 目标数据指针
返回值	返回 copy 的有效图像数据个数，为 0 则没有发生拷贝
说明	1. 本接口仅支持同设备卡号内的图像数据拷贝，即 id 相同 2. 函数中的 id 仅需要指定设备卡号，不需要其他标志位

int av::get_scale_and_plane(int color_format, int wscale[], int hscale[])

函数原型	int av::get_scale_and_plane(int color_format, int wcale[], int hscale[])
功能	获取指定图像格式相对于 YUV444P 的宽高比例系数
输入参数	int color_format –指定图像格式，详见 FFMPEG pixfmt.h 定义
输出参数	int wscale[] –对应格式相对于 YUV444P 每一层的宽度比例; int hscale[] - 对应格式相对于 YUV444P 每一层的高度比例
返回值	返回给定图像格式的 plane 层数
说明	

cv::Mat(int height, int width, int total, int _type, const size_t* _steps, void* _data, unsigned long addr, int fd, SophonDevice device=SophonDevice())

函数原型	cv::Mat(int height, int width, int total, int _type, const size_t* _steps, void* _data, unsigned long addr, int fd, SophonDevice device=SophonDevice())
功能	新增的 Mat 构造接口。可以创建 opencv 标准格式或扩展的 YUV Mat 格式，并且系统内存和设备内存都允许通过外部分配给定
输入参数	int height –输入图像数据的高; int width –输入图像数据的宽; int total –内存大小，该内存可以为内部待分配的内存，或外部已分配内存的大小; int _type –Mat 类型，本接口只支持 CV_8UC1 或 CV_8UC3，扩展的 YUV Mat 的格式 _type 类型一律为 CV_8UC1; const size_t *steps –所创建的图像数据的 step 信息，如果该指针为 null，则为 AUTO_STEP; void *_data –系统内存指针，如果为 null，则内部分配该内存; unsigned long addr –设备物理内存地址，任意值均被认为有效的物理地址; int fd –设备物理内存对应的句柄。如果为负，则设备物理内存在内部分配管理; SophonDevice device –指定设备卡号以及 HEAP 位置的标志，详见 5.1，该参数默认为 0
输出参数	构造的标准 BGR 或扩展 YUV 的 Mat 数据类型
返回值	无
说明	1.SophonDevice 是为了避免 C++ 隐含类型匹配造成函数匹配失误而引入的类型，可以用 SophonDevice(int id) 直接从 5.1 节的 ID 转换过来 2. 当设备内存和系统内存均有外部给出时，在 soc 模式下外部要保证两者地址的匹配，即系统内存是设备内存映射出来的虚拟地址；当设备内存由外部给出，系统内存为 null 时，该接口内部会自动创建系统内存; 当设备内存没有给出，系统内存也为 null 时，本接口内部会自动创建; 当设备内存没有给出，系统内存由外部给出时，本接口创建的 Mat 在 soc 模式下只有系统内存

Mat::Mat(SophonDevice device)

函数原型	Mat::Mat(SophonDevice device)
功能	新增的 Mat 构造接口，指定该 Mat 的后续操作在给定的 device 设备上
输入参数	SophonDevice device - 指定设备卡号以及 HEAP 位置的标志，详见 5.1
输出参数	声明 Mat 数据类型
返回值	无
说明	1. 本构造函数仅初始化 Mat 内部的设备 index，并不实际创建内存 2. 本构造函数的最大作用是对于某些内部 create 内存的函数，可以通过这个构造函数，提前指定创建内存的设备号和 HEAP 位置，从而避免将大量的内存分配在默认的设备号 0 上

void Mat::create(int height, int width, int total, int _type, const size_t* _steps, void* _data, unsigned long addr, int fd, int id = 0)

函数原型	void Mat::create(int height, int width, int total, int type, const size_t* _steps, void* _data, unsigned long addr, int fd, int id = 0)
功能	Mat 分配内存接口，该接口系统内存和设备内存都允许通过外部分配给定，也可内部分配。
输入参数	int height –输入图像数据的高; int width –输入图像数据的宽; int total –内存大小，该内存可以为内部待分配的内存，或外部已分配内存的大小; int _type –Mat 类型，本接口只支持 CV_8UC1 或 CV_8UC3，扩展的 YUV Mat 的格式 _type 类型一律为 CV_8UC1; const size_t *steps –所创建的图像数据的 step 信息，如果该指针为 null，则为 AUTO_STEP; void *_data –系统内存指针，如果为 null，则内部分配该内存; unsigned long addr –设备物理内存地址，任意值均被认为有效的物理地址; int fd –设备物理内存对应的句柄。如果为负，则设备物理内存在内部分配管理; int id –指定设备卡号以及 HEAP 位置的标志，详见 5.1，该参数默认为 0
输出参数	无
返回值	无
说明	1. 扩展的内存分配接口，主要改进目的是允许外置指定设备物理内存，当设备或者系统内存由外部创建的时候，则外部必须负责该内存的释放，否则会造成内存泄漏 2. 当设备内存和系统内存均有外部给出时，在 soc 模式下外部要保证两者地址的匹配，即系统内存是设备内存映射出来的虚拟地址；当设备内存由外部给出，系统内存为 null 时，该接口内部会自动创建系统内存; 当设备内存没有给出，系统内存也为 null 时，本接口内部会自动创建; 当设备内存没有给出，系统内存由外部给出时，本接口创建的 Mat 在 soc 模式下只有系统内存

void VideoWriter::write(InputArray image)

函数原型	void VideoWriter::write(InputArray image)
功能	与 OpenCV 标准 VideoWriter::write 接口用法和功能一致 **在调用 write 将全部 frame 传入之后，需要额外调用一次 write，传入空的 Mat 进行 flush 操作**
输入参数	InputArray image –输入的图像数据 Mat 结构
返回值	无
使用示例	

```
// General use, frame is the actual data you get from VideoCapture or somewhere
write(frame);
// Flush, empty is an empty data need not to be initialized
Mat emptyMat;
write(emptyMat);   // 仅需要调用一次
```

void VideoWriter::write(InputArray image, char *data, int *len, cv::CV_RoiInfo *roiinfo)

函数原型	void VideoWriter::write(InputArray image, char *data, int len)
功能	新增的视频编码接口。与 OpenCV 标准 VideoWriter::write 接口不同，他提供了将编码视频数据输出到 buffer 的功能，便于后续处理 **在调用 write 将全部 frame 传入之后，需要额外调用多次 write，传入空的 Mat 进行 flush 操作，直到接收不到新数据 (len=0) 停止 flush**
输入参数	InputArray image –输入的图像数据 Mat 结构 cv::CV_RoiInfo *roiinfo(可选) - 输入的 roi 信息
输出参数	char *data –输出的编码数据缓存 int *len –输出的编码数据长度
返回值	无
使用示例	

```
// General use, frame is the actual data you get from VideoCapture or somewhere
write(frame, data ,&data_len);
fwrite(data, 1, data_len, fp_out);  //fp_out is the file to be written
// Flush, empty is an empty data need not to be initialized
Mat emptyMat;
while(1){   //需要多次调用
    writer.write(emptyMat, data, &data_len);
    if(len > 0){
        fwrite(data, 1, data_len, fp_out);
        len = 0; //reset data_len
    }else
        break;  //直到拿不出数据结束flush
}
```

virtual bool VideoCapture::grab(char *buf, unsigned int len_in, unsigned int *len_out);

函数原型	bool VideoCapture::grab(char *buf, unsigned int len_in, usigned int *len_out);
功能	新增的收流解码接口。与 OpenCV 标准 VideoWriter::grab 接口不同，他提供了将解码前的视频数据输出到 buf 的功能。
输入参数	char *buf –外部负责分配释放内存。 unsigned int len_in –buf 空间的大小。
输出参数	char *buf –输出解码前的视频数据。 int *len_out –输出的 buf 的实际大小。
返回值	true 表示收流解码成功； false 表示收流解码失败。

virtual bool VideoCapture::read_record(OutputArray image, char *buf, unsigned int len_in, unsigned int *len_out);

函数原型	bool VideoCapture::read_record(OutputArray image, char *buf, unsigned int len_in, unsigned int *len_out);
功能	新增的读取码流视频接口。他提供了将解码前的视频数据输出到 buf 的功能，将解码后的数据输出到 image。
输入参数	char *buf –外部负责分配释放内存。 unsigned int len_in –buf 空间的大小。
输出参数	OutputArray image –输出解码后的视频数据。 char *buf –输出解码前的视频数据。 int *len_out –输出的 buf 的实际大小。
返回值	true 表示收流解码成功； false 表示收流解码失败。

2.2.4 硬件 JPEG 解码器的 OpenCV 扩展

在 BM1688 系列芯片中，提供 JPEG 硬件编码解码模块。为使用这些硬件模块，SDK 软件包中，扩展了 OpenCV 中与 JPEG 图片处理相关的 API 函数，如：cv::imread()、cv::imwrite()、cv::imdecode()、cv::imencode() 等。您在使用这些函数做 JPEG 编解码的时候，函数内部会自动调用底层的硬件加速资源，从而大幅度提高了编解码的效率。如果您想保持这些函数原始的 OpenCV API 使用习惯，可以略过本节介绍；但如果你还想了解一下我们提供的简单易用的扩展功能，这节可能对您非常有帮助。

输出 yuv 格式的图像数据

OpenCV 原生的 cv::imread()、cv::imdecode() API 函数执行 JPEG 图片的解码操作，返回一个M at 结构体，该 Mat 结构体中保存有 BGR packed 格式的图片数据，算能扩展的 API 函数功能可以返回 JPEG 图片解码后的原始的 YUV 格式数据。用法如下：

当这两个函数的第二个参数 flags 被设置成 cv::IMREAD_AVFRAME 时，表示解码后返回的 Mat 结构体 out 中保存着 YUV 格式的数据。具体是什么格式的 YUV 数据要根据 JPEG 文件的 image 格式而定。当 flags 被设置成其它值或者省略不设置时，表示解码输出 OpenCV 原生的 BGR packed 格式的 Mat 数据。解码器输入输出扩展数据格式说明如下表所示：

输入 Image 格式	输入 YUV 格式	FFMPEG 对应格式
I400	I400	AV_PIX_FMT_GRAY8
I420	NV12	AV_PIX_FMT_NV12
I422	NV16	AV_PIX_FMT_NV16
I444	I444 planar	AV_PIX_FMT_YUV444P

可以通过 Mat::avFormat() 扩展函数，得到当前数据所对应的具体的 FFmpeg 格式。可以通过 Mat::avOK() 扩展函数，得知 cv::imdecode(buf, cv::IMREAD_AVFRAME, &out) 解码返回的 out，是否是算能扩展的 Mat 数据格式。

另外在这两个接口中的 flags 增加 cv::IMREAD_RETRY_SOFTDEC 标志时会在硬件解码失败的情况下尝试切换软件解码，也可以通过设置环境变量 OPENCV_RETRY_SOFTDEC=1 实现此功能。

支持 YUV 格式的函数列表

目前算能 Opencv 已经支持 YUV Mat 扩展格式的函数接口列表如下：

- 视频解码类接口
  - VideoCapture 类的成员函数

  这类成员函数如 read, grab，对于常用的 HEVC, H264 视频格式都使用了 BM1688 系列的硬件加速，并支持 YUV Mat 扩展格式。
- 视频编码类接口
  - VideoWriter 类的成员函数

  这类成员函数如 write，对于常用的 HEVC,H264 视频格式已经使用了 BM1688 系列的硬件加速，并支持 YUV Mat 扩展格式。
- JPEG 编码类接口
- JPEG 解码类接口
  - Imread
  - Imwrite

  - Imdecode
  - Imencode

  以上接口在处理 JPEG 格式的时候，已经使用了 BM1688 系列的硬件加速功能，并支持 YUV Mat 扩展格式。

- 图像处理类接口
  - cvtColor
  - resize

    这两个接口在 BM1688 系列 SOC 模式下支持 YUV Mat 扩展格式，并使用硬件加速进行了优化。

    尤其需要注意的是 cvtColor 接口，只在 YUV 转换成 BGR 或者 GRAY 输出的时候支持硬件加速和 YUV Mat 的格式，即只支持输入为 YUV Mat 格式，并进行了硬件加速，输出不支持 YUV Mat 格式。

- line
- rectangle
- circle
- putText

  以上四个接口均支持 YUV 扩展格式。注意，这四个接口并没有采用硬件加速，而是使用 CPU 对 YUV Mat 扩展格式进行的支持。

- 基本操作类接口
  - Mat 类部分接口
    * 创建释放接口：create，release，Mat 声明接口
    * 内存赋值接口：clone，copyTo，cloneAll，copyAllTo，assignTo, operator =,
    * 扩展 AV 接口：avOK, avComp, avRows, avCols, avFormat, avStep, avAddr

  以上接口均支持 YUV 扩展格式，尤其是 copyTo, clone 接口都采用硬件进行了加速。

- 扩展类接口
  - Bmcv 接口: 详见 opencv2/core/bmcv.hpp
  - AvFrame 接口: 详见 opencv2/core/av.hpp

  以上算能扩展类接口，均支持 YUV Mat 扩展格式，并均针对硬件加速处理进行了优化。

注意：支持 YUV Mat 扩展格式的接口并不等价于使用了硬件加速，部分接口是通过 CPU 处理来实现。这点需要特别注意。

2.2.5 OpenCV 与 BMCV API 的调用原则

BMCV API 充分发挥了 BM1688 系列芯片中的硬件单元的加速能力，能提高数据处理的效率。而 OpenCV 软件提供了非常丰富的图像图形处理能力，将两者有机的结合起来，使客户开发既能利用 OpenCV 丰富的函数库，又能在硬件支持的功能上获得加速，是本节的主要目的。

在 BMCV API 和 OpenCV 函数以及数据类型的切换过程中，最关键是要尽量避免数据拷贝，使得切换代价最小。因此在调用流程中要遵循以下原则。

1) 由 OpenCV Mat 到 BMCV API 的切换，可以利用 toBMI() 函数，该函数以零拷贝的方式，将 Mat 中的数据转换成了 BMCV API 调用所需的 bm_image 类型。

2) 当 BMCV API 需要切换到 OpenCV Mat 的时候，要将最后一步的操作通过 OpenCV 中的 bmcv 函数来实现。这样既完成所需的图像处理操作，同时也为后续 OpenCV 操作完成了数据类型准备。因为一般 OpenCV 都要求 BGR Pack 的色彩空间，所以一般用 toMat() 函数作为切换前的最后一步操作。

3) 一般神经网络处理的数据为不带 padding 的 RGB planar 数据，并且对于输入尺寸有特定的要求。因此建议将 resize() 函数作为调用神经网络 NPU 接口前的最后一步操作。

4) 当 crop、resize、color conversion 三个操作是连续的时候，强烈建议客户使用 convert() 函数，这可以在带宽优化和速度优化方面都获得理想的收益。即使后续可能还需要做一次拷贝，但因为拷贝发生在缩放之后的图像上，这种代价也是值得的。

2.2.6 OpenCV 中 GB28181 国标接口介绍

SOPHGO 复用 OpenCV 原生的 Cap 接口，通过对于 url 定义进行扩展，提供 GB28181 国标的播放支持。因此客户并不需要重新熟悉接口，只要对扩展的 url 定义进行理解，即可像播放 rtsp 视频一样，无缝的播放 GB28181 视频。

注意：国标中的 SIP 代理注册步骤，需要客户自己管理。当获取到前端设备列表后，可以直接用 url 的方式进行播放。

国标 GB28181 支持的一般步骤

- 启动 SIP 代理（一般客户自己部署或者平台方提供）
- 客户的下级应用平台注册到 SIP 代理
- 客户应用获取前端设备列表，如下所示。其中，34010000001310000009 等为设备 20 位编码。

  { “devidelist” :

  [{ “id” : “34010000001310000009” }

  { “id” : “34010000001310000010” }

  { “id” : “34020000001310101202” }]}
- 组织 GB28181 url 直接调用 OpenCV Cap 接口进行播放

GB28181 url 格式定义

UDP 实时流地址定义

gb28181://34020000002019000001:123456@35.26.240.99:5666?deviceid
=34010000001310000009#localid=1247879287116362497 9#localip=172.
10.18.201#localmediaport=20108:

gb28181://34020000002019000001:123456@35.26.240.99:5666?deviceid
=34010000001310000009#localid=1247879287116362497 9#localip=172.
10.18.201#localmediaport=20108:

注释

34020000002019000001:123456@35.26.240.99:5666:

sip 服务器国标编码:sip 服务器的密码 @sip 服务器的 ip 地址:sip 服务器的 port

deviceid:

前端设备 20 位编码

localid:

本地二十位编码，可选项

localip:

本地 ip，可选项

localmediaport:

媒体接收端的视频流端口，需要做端口映射，映射两个端口 (rtp:11801, rtcp:11802)，两个端口映射的 in 和 out 要相同。同一个核心板端口不可重复。

UDP 回放流地址定义

gb28181_playback://34020000002019000001:123456@35.26.240.99:5666?deviceid
=35018284001310090010#devicetype=3#localid=1247879287116362497 9#localip=
172.10.18.201#localmediaport=20108#begtime=20191018160000#endtime
=20191026163713:

gb28181_playback://34020000002019000001:123456@35.26.240.99:5666?deviceid
=35018284001310090010#devicetype=3#localid=1247879287116362497 9#localip=
172.10.18.201#localmediaport=20108#begtime=20191018160000#endtime
=20191026163713:

注释

34020000002019000001:123456@35.26.240.99:5666:

sip 服务器国标编码:sip 服务器的密码 @sip 服务器的 ip 地址:sip 服务器的 port

deviceid:

前端设备 20 位编码

devicetype:

录像存储类型

localid:

本地二十位编码，可选项

localip:

本地 ip，可选项

localmediaport:

媒体接收端的视频流端口，需要做端口映射，映射两个端口 (rtp:11801, rtcp:11802)，两个端口映射的 in 和 out 要相同。同一个核心板端口不可重复。

begtime:

录像起始时间

endtime:

录像结束时间

TCP 实时流地址定义

gb28181://34020000002019000001:123456@35.26.240.99:5666?deviceid
=35018284001310090010#localid=124787928711636249 79#localip=172.10.18.201:

gb28181://34020000002019000001:123456@35.26.240.99:5666?deviceid
=35018284001310090010#localid=124787928711636249 79#localip=172.10.18.201:

注释

34020000002019000001:123456@35.26.240.99:5666:

sip 服务器国标编码:sip 服务器的密码 @sip 服务器的 ip 地址:sip 服务器的 port

deviceid:

前端设备 20 位编码

localid:

本地二十位编码，可选项

localip:

本地 ip，可选项

TCP 回放流地址定义

gb28181_playback://34020000002019000001:123456@35.26.240.99:5666?deviceid=35018284001310090010#devicetype=3#localid=12478792871163624979#localip=172.10.18.201#begtime=20191018160000#endtime=20191026163713:

gb28181_playback://34020000002019000001:123456@35.26.240.99:5666?deviceid=35018284001310090010#devicetype=3#localid=12478792871163624979#localip=172.10.18.201#begtime=20191018160000#endtime=20191026163713:

注释

34020000002019000001:123456@35.26.240.99:5666:

sip 服务器国标编码:sip 服务器的密码 @sip 服务器的 ip 地址:sip 服务器的 port

deviceid:

前端设备 20 位编码

devicetype:

录像存储类型

localid:

本地二十位编码，可选项

localip:

本地 ip，可选项

begtime:

录像起始时间

endtime:

录像结束时间

2.3 SOPHGO FFMPEG 使用指南

2.3.1 前言

BM1688 系列芯片中，有一个 8 核的 A53 处理器，同时还内置有视频、图像相关硬件加速模块。在 SOPHGO 提供的 FFMPEG SDK 开发包中，提供了对这些硬件模块的接口。其中，通过这些硬件接口，提供了如下模块：硬件视频解码器、硬件视频编码器、硬件 JPEG 解码器、硬件 JPEG 编码器、硬件 scale filter、硬件 overlay filter、hwupload filter、hwdownload filter。

FFMPEG SDK 开发包符合 FFMPEG hwaccel 编写规范，实现了视频转码硬件加速框架，实现了硬件内存管理、各个硬件处理模块流程的组织等功能。同时 FFMPEG SDK 也提供了与通常 CPU 解码器兼容的接口，以匹配部分客户的使用习惯。这两套接口我们称之为 HWAccel 接口和常规接口，他们底层共享 BM1688 硬件加速模块，在性能上是相同的。区别仅在于 1）HWAccel 需要初始化硬件设备; 2）HWAccel 接口只面向设备内存，而常规接口同时分配了设备内存和系统内存; 3）他们的参数配置和接口调用上有轻微差别。

下面描述中，如非特殊说明，对常规接口和 HWAccel 接口都适用。

2.3.2 硬件视频解码器

BM1688 系列支持 H.264 和 H.265 硬件解码。硬件解码器性能详情如下表所述。

Standard	Profile	Level	Max R eso-lution	Min R esolu-tion	Bit rate
H.264/AVC	BP/CBP/MP/HP	4.1	8192x8192	16x16	50Mbps
H.265/HEV(	Main/Main10	L5.1	8192x8192	16x16	N/A

在 SophGo 的 FFMPEG 发布包中，H.264 硬件视频解码器的名字为 h264_bm，H.265 硬件视频解码器的名字为 hevc_bm。可通过如下命令, 来查询 FFMPEG 支持的编码器。

```
$ ffmpeg -decoders | grep _bm
```

硬件视频解码器支持的选项

FFMPEG 中，BM1688 系列的硬件解码器提供了一些额外选项，可以通过如下命令查询。

```
$ ffmpeg -h decoder=h264_bm
```

```
$ ffmpeg -h decoder=hevc_bm
```

这些选项可以使用 av_dict_set API 来设置。在设置之前，需要对对这些选项有正确的理解。下面详细解释一下这些选项。

output_format:

- 输出数据的格式。

- 设为 0，则输出线性排列的未压缩数据；设为 101，则输出压缩数据。
- 缺省值为 0。
- 推荐设置为 101，输出压缩数据。可以节省内存、节省带宽。输出的压缩数据，可以调用后面介绍的 scale_bm filter 解压缩成正常的 YUV 数据。具体可参考应用示例中的示例 1。

cbcr_interleave:

- 硬件视频解码器解码输出的帧色度数据是否是交织格式。
- 设为 1，则输出为 semi-planar yuv 图像，譬如 nv12；设为 0，则输出 planar yuv 图像，譬如 yuv420p。
- 缺省值为 1。

extra_frame_buffer_num:

- 硬件视频解码器额外提供硬件帧缓存数量。
- 缺省值为 7。最小值为 1。

skip_non_idr:

- 跳帧模式。0，关闭；1，跳过 Non-RAP 帧；2，跳过非参考帧。
- 缺省值为 0。

handle_packet_loss

- 出错时，对 H.264, H.265 解码器使能丢包处理。0, 不做丢包处理；1，进行丢包处理。
- 缺省值为 0。

zero_copy:

- 将设备上的帧数据直接拷贝到 AVFrame 的 data[0]-data[3] 所自动申请的系统内存里。1，关闭拷贝；0，使能拷贝。
- 缺省值为 1。

dec_cmd_queue:

- 设置 cmmond queue 的深度，可选值为 1~4。cmmond queue 可以提高解码器并行处理效率，但是会占用额外的内存资源。
- 缺省值为 4。

2.3.3 硬件视频编码器

BM1688 硬件视频编码器，支持 H.264/AVC 和 H.265/HEVC 视频编码。

BM1688 硬件编码器设计的能力为: 能够实时编码 10 路 1080P30 的视频。具体指标如下:

H.265 编码器:

- Capable of encoding HEVC Main/Main10/MSP(Main Still Picture) Profile @ L5.1 High-tier

H.264 编码器:

- Capable of encoding Baseline/Constrained Baseline/Main/High/High 10 Profiles Level @ L5.2

通用指标

- 最大分辨率: 8192x8192
- 最小分辨率: 256x128
- 编码图像宽度须为 8 的倍数
- 编码图像高度宽度须为 8 的倍数

在 SophGo 的 FFMPEG 发布包中，H.264 硬件视频编码器的名字为 h264_bm，H.265 硬件视频编码器的名字为 h265_bm 或 hevc_bm。可通过如下命令, 来查询 FFMPEG 支持的编码器。

```
$ ffmpeg -encoders
```

硬件视频编码器支持的选项

FFMPEG 中，硬件视频编码器提供了一些额外选项，可以通过如下命令查询。

```
$ ffmpeg -h encoder=h264_bm
```

```
$ ffmpeg -h encoder=hevc_bm
```

BM1688 硬件视频编码器支持如下选项:

preset: 预设编码模式。推荐通过 enc-params 设置。

- 0 - fast, 1 - medium, 2 - slow。
- 缺省值为 2。

gop_preset: gop 预设索引值。推荐通过 enc-params 设置。

- 1: all I, gopsize 1
- 2: IPP, cyclic gopsize 1
- 3: IBB, cyclic gopsize 1
- 4: IBPBP, cyclic gopsize 2

- 5: IBBBP, cyclic gopsize 4
- 6: IPPPP, cyclic gopsize 4
- 7: IBBBB, cyclic gopsize 4
- 8: random access, IBBBBBBBB, cyclic gopsize 8

qp:

- 恒定量化参数的码率控制方法
- 取值范围为 0 至 51

perf:

- 用于指示是否需要测试编码器性能
- 取值范围为 0 或 1。

enc-params:

- 用于设置视频编码器内部参数。
- 支持的编码参数：preset, gop_preset, qp, bitrate, mb_rc, delta_qp, min_qp, max_qp, bg，nr，deblock，weightp
- 编码参数 preset：取值范围为 fast, medium, slow 或者是 0，1，2
- 编码参数 gop_preset：gop 预设索引值。参考上面已有详细解释。

  1: all I, gopsize 1

  2: IPP, cyclic gopsize 1

  3: IBB, cyclic gopsize 1

  4: IBPBP, cyclic gopsize 2

  5: IBBBP, cyclic gopsize 4

  6: IPPPP, cyclic gopsize 4

  7: IBBBB, cyclic gopsize 4

  8: random access, IBBBBBBBB, cyclic gopsize 8
- 编码参数 qp：恒定量化参数，取值范围为 [0, 51]。当该值有效时，关闭码率控制算法，用固定的量化参数编码。
- 编码参数 bitrate：用于编码所指定的码率。单位是 Kbps，1Kbps=1000bps。当指定改参数时，请不要设置编码参数 qp。
- 编码参数 mb_rc：取值范围 0 或 1。当设为 1 时，开启宏块级码率控制算法；当设为 0 时，开启帧级码率控制算法。
- 编码参数 delta_qp：用于码率控制算法的 QP 最大差值。该值太大影响视频主观质量。太小影响码率调整的速度。

- 编码参数 min_qp 和 max_qp：码率控制算法中用于控制码率和视频质量的最小量化参数和最大量化参数。取值范围 [0, 51]。
- 编码参数 bg：是否开启背景检测。取值范围 0 或 1。
- 编码参数 nr：是否开启降噪算法。取值范围 0 或 1。
- 编码参数 deblock：是否开启环状滤波器。有如下几种用法：
  - 关闭环状滤波器“deblock=0”或“no-deblock”。
  - 简单开启环状滤波器，使用缺省环状滤波器参数”deblock=1”。
  - 开启环状滤波器并设置参数，譬如”deblock=6,6”。
- 编码参数 weightp：是否开启 P 帧、B 帧加权预测。取值范围 0 或 1。

is_dma_buffer:

- 用于提示编码器，输入的帧缓存是否为设备上的连续物理内存地址。
- 在 SoC 模式，值 0 表示输入的是设备内存的虚拟地址。值 1 表示，输入的是设备上的连续物理地址。
- 缺省值为 1。
- 仅适用于常规接口。

roi_enable

- 使能 roi。
- 缺省值为 0

enc_cmd_queue

- 设置 cmmond queue 的深度，可选值为 1~4。cmmond queue 可以提高编码器并行处理效率，但是会占用额外的内存资源。
- 缺省值为 4

2.3.4 硬件 JPEG 解码器

在 BM1688 系列芯片中，硬件 JPEG 解码器提供硬件 JPEG 图像解码输入能力。这里介绍一下，如何通过 FFMPEG 来实现硬件 JPEG 解码。

在 FFMPEG 中, 硬件 JPEG 解码器的名称为 jpeg_bm。可以通过如下命令, 来查看 FFMPEG 中是否有 jpeg_bm 解码器。

```
$ ffmpeg -decoders | grep jpeg_bm
```

硬件 JPEG 解码器支持的选项

FFMPEG 中，可以通过如下命令, 来查看 jpeg_bm 解码器支持的选项

```
$ ffmpeg -h decoder=jpeg_bm
```

解码选项的说明如下。硬件 JPEG 解码器中这些选项, 可以使用 av_dict_set() API 函数对其进行重置。

bs_buffer_size: 用于设置硬件 JPEG 解码器中输入比特流的缓存大小 (KBytes)。

- 取值范围 (0 到 INT_MAX)
- 缺省值 5120

cbcr_interleave: 用于指示 JPEG 解码器输出的帧数据中色度数据是否为交织的格式。

- 0: 输出的帧数据中色度数据为 plannar 的格式
- 1: 输出的帧数据中色度数据为 interleave 的格式
- 缺省值为 0

num_extra_framebuffers: JPEG 解码器需要的额外帧缓存数量

- 对于 Still JPEG 的输入, 建议该值设为 0
- 对于 Motion JPEG 的输入, 建议该值至少为 2
- 取值范围 (0 到 INT_MAX)
- 缺省值为 2

2.3.5 硬件 JPEG 编码器

在 BM1688 系列芯片中，硬件 JPEG 编码器提供硬件 JPEG 图像编码输出能力。这里介绍一下，如何通过 FFMPEG 来实现硬件 JPEG 编码。

在 FFMPEG 中, 硬件 JPEG 编码器的名称为 jpeg_bm。可以通过如下命令, 来查看 FFMPEG 中是否有 jpeg_bm 编码器。

```
$ ffmpeg -encoders | grep jpeg_bm
```

硬件 JPEG 编码器支持的选项

FFMPEG 中，可以通过如下命令, 来查看 jpeg_bm 编码器支持的选项

```
$ ffmpeg -h encoder=jpeg_bm
```

编码选项的说明如下。硬件 JPEG 编码器中这些选项, 可以使用 av_dict_set() API 函数对其进行重置。

is_dma_buffer:

- 用于提示编码器，输入的帧缓存是否为设备上的连续物理内存地址。

- 在 SoC 模式，值 0 表示输入的是设备内存的虚拟地址。值 1 表示，输入的是设备上的连续物理地址。
- 缺省值为 1。
- 仅适用于常规接口。

2.3.6 硬件 scale filter

BM1688 系列硬件 scale filter 用于将输入的图像进行”缩放/裁剪/补边”操作。譬如，转码应用。在将 1080p 的视频解码后，使用硬件 scale 缩放成 720p 的，再进行压缩输出。

内容	最大分辨率	最小分辨率	放大倍数
硬件限制	4096 * 4096	8*8	32

在 FFMPEG 中，硬件 scale filter 的名称为 scale_bm。

```
$ ffmpeg -filters | grep bm
```

硬件 scale filter 支持的选项

FFMPEG 中，可以通过如下命令, 来查看 scaler_bm 编码器支持的选项

```
$ ffmpeg -h filter=scale_bm
```

scale_bm 选项的说明如下:

w:

- 缩放输出视频的宽度。请参考 ffmpeg scale filter 的用法。

h:

- 缩放输出视频的高度。请参考 ffmpeg scale filter 的用法。

format:

- 缩放输出视频的像素格式。请参考 ffmpeg scale filter 的用法。
- 输入输出支持的格式详见附表 7.1。
- 缺省值”none”。即输出像素格式为系统自动。输入为 yuv420p，输出为 yuv420p; 输入为 yuvj420p，输出为 yuvj420p。输入为 nv12 时，缺省输出为 yuv420p。
- 在 HWAccel 框架下：支持 nv12 到 yuv420p、nv12 到 yuvj420p、yuv420p 到 yuvj420p、yuvj422p 到 yuvj420p、yuvj422p 到 yuv420p 的格式转换。在不启用 HWAccel 框架的正常模式下支持情况见附表 7.1。

输入	输出	是否持缩放	是否支持颜色转换
GRAY8	GRAY8	是	是
NV12（压缩）	YUV420P	是	是
	YUV422P	否	是
	YUV444P	是	是
	BGR	是	是
	RGB	是	是
	RGBP	是	是
	BGRP	是	是
NV12（非压缩）	YUV420P	是	是
	YUV422P	否	是
	YUV444P	是	是
	BGR	是	是
	RGB	是	是
	RGBP	是	是
	BGRP	是	是
YUV420P	YUV420P	是	是
	YUV422P	否	是
	YUV444P	是	是
	BGR	是	是
	RGB	是	是
	RGBP	是	是
	BGRP	是	是
YUV422P	YUV420P	是	是
	YUV422P	否	否
	YUV444P	否	否
	BGR	是	是
	RGB	是	是
	RGBP	是	是
	BGRP	是	是
YUV444P	YUV420P	是	是
	YUV422P	否	是
	YUV444P	是	是
	BGR	是	是
	RGB	是	是
	RGBP	是	是
	BGRP	是	是
BGR、RGB	YUV420P	是	
	YUV422P	否	是
	YUV444P	是	是
	BGR	是	是
	RGB	是	是
	RGBP	是	是
	BGRP	是	是
RGBP、BGRP	YUV420P	是	

续下页

表 2.1 – 接上页

输入	输出	是否持缩放	是否支持颜色转换
	YUV422P	否	是
	YUV444P	是	是
	BGR	是	是
	RGB	是	是
	RGBP	是	是
	BGRP	是	是

表 7.1 scale_bm 像素格式支持列表

opt:

- 缩放操作 (from 0 to 2) (default 0)
- 值 0 - 仅支持缩放操作。缺省值。
- 值 1 - 支持缩放 + 自动裁剪操作。命令行参数中可用 crop 来表示。
- 值 2 - 支持缩放 + 自动补黑边操作。命令行参数中可用 pad 来表示。

flags:

- 缩放方法 (from 0 to 2) (default 1)
- 值 0 - nearest 滤波器。命令行参数中，可用 nearest 来表示。
- 值 1 - bilinear 滤波器。命令行参数中，可用 bilinear 来表示。
- 值 2 - bicubic 滤波器。命令行参数中，可用 bicubic 来表示。

sophon_idx:

- 设备 ID , 从 0 开始。

zero_copy:

- 值 0 - 表示 scale_bm 的输出 AVFrame 将同时包含设备内存和主机内存指针，兼容性最好，性能稍有下降。缺省为 0
- 值 1 - 表示 scale_bm 的输出到下一级的 AVFrame 中将只包含有效设备地址，不会对数据进行从设备内存到系统内存的同步。建议对于下一级接使用 SOPHGO 的编码/filter 的情况，可以选择设置为 1，其他建议设置为 0。
- 缺省为 0

2.3.7 硬件 overlay filter

BM1688 系列硬件 overlay filter 用于将一个视频流叠加在另一个视频流上，可以用于实现水印/字幕/画中画等效果。

例如，在一个 1080p 的主输入视频上使用硬件 overlay 叠加一个 352x288 大小的滤镜视频。具体地，将两个视频解码后，使用硬件 overlay 把 352x288 大小的滤镜视频叠加到 1080p 的视频上，再将处理后的主输入视频进行压缩输出。

在 FFMPEG 中，硬件 overlay filter 的名称为 overlay_bm。

```
$ ffmpeg -filters | grep bm
```

硬件 overlay filter 支持的选项

FFMPEG 中，可以通过如下命令，来查看 overlay_bm filter 支持的选项

```
$ ffmpeg -h filter=overlay_bm
```

overlay_bm 选项的说明如下:

sophon_idx:

- 设备 ID，缺省值为 0。

zero_copy:

- 将设备上的帧数据直接拷贝到 AVFrame 的 data[0]-data[3] 所自动申请的系统内存里，缺省值为 1。
- 值 1，关闭拷贝。
- 值 0，使能拷贝。

x:

- 设置叠加滤镜的水平位置

y:

- 设置叠加滤镜的垂直位置

eof_action:

- 当遇到叠加输入的 EOF（文件结束）状态时，overlay 接下来的操作，缺省值为 0。
- 值 0 - repeat，重复叠加输入的上一帧
- 值 1 - endall，立即结束叠加
- 值 2 - pass，仅输入主输入流

eval:

- 用于指定在视频处理阶段的何时评估表达式，缺省值为 1。
- 值 0 - init，在初始化时评估表达式一次

· 值 1 - frame，每帧都评估表达式，可以根据输入的变化动态调整 overlay 的属性。

shortest:

· 如果设置为 1，则当较短的输入流结束时，整个处理将结束，缺省值为 0。

repeatlast:

· 如果设置为 1，当叠加内容结束后，将重复最后一帧的内容，缺省值为 1。

ts_sync_mode:

· 控制流之间的时间同步策略，缺省值为 0。

· 值 0，从叠加流中选择时间戳最接近且小于等于主输入流当前帧的帧。

· 值 1，从叠加流中选择与主输入流当前帧时间戳绝对最接近的帧。

2.3.8 AVFrame 特殊定义说明

遵从 FFMPEG 的规范，硬件解码器是通过 AVFrame 来提供输出的，硬件编码器是通过 AVFrame 来提供输入的。因此，在通过 API 方式，调用 FFMPEG SDK、进行硬件编解码处理时，需要注意到 AVFrame 的如下特殊规定。AVFrame 是线性 YUV 输出。在 AVFrame 中，data 为数据指针，用于保存物理地址，linesize 为每个平面的线跨度。

硬件解码器输出的 avframe 接口定义

常规接口

data 数组的定义

下标	说明
0	Y 的虚拟地址
1	cbcr_interleave=1 时 CbCr 的虚拟地址; cbcr_interleave=0 时 Cb 的虚拟地址
2	cbcr_interleave=0 时 Cr 的虚拟地址
3	未使用
4	Y 的物理地址
5	cbcr_interleave=1 时 CbCr 的物理地址; cbcr_interleave=0 时 Cb 的物理地址
6	cbcr_interleave=0 时 Cr 的物理地址
7	未使用

linesize 数组的定义

下标	说明
0	Y 的虚拟地址的跨度
1	cbcr_interleave=1 时 CbCr 的虚拟地址的跨度; cbcr_interleave=0 时 Cb 的虚拟地址的跨度
2	cbcr_interleave=0 时 Cr 的虚拟地址的跨度
3	未使用
4	Y 的物理地址的跨度
5	cbcr_interleave=1 时 CbCr 的物理地址的跨度; cbcr_interleave=0 时 Cb 的物理地址的跨度
6	cbcr_interleave=0 时 Cr 的物理地址的跨度
7	未使用

HWAccel 接口

data 数组的定义

下标	未压缩格式说明	压缩格式明
0	Y 的物理地址	压缩的亮度数据的物理地址
1	cbcr_interleave=1 时 CbCr 的物理地址; cbcr_interleave=0 时 Cb 的物理地址	压缩的色度数据的物理地址
2	cbcr_interleave=0 时 Cr 的物理地址	亮度数据的偏移量表的物理地址
3	保留	色度数据的偏移量表的物理地址
4	保留	保留

linesize 数组的定义

下标	未压缩格式说明	压缩格式说明
0	Y 的物理地址的跨度	亮度数据的跨度
1	cbcr_interleave=1 时 CbCr 的物理地址的跨度; cbcr_int erleave=0 时 Cb 的物理地址的跨度	色度数据的跨度
2	cbcr_interleave=0 时 Cr 的物理地址的跨度	亮度偏移量表的大小
3	未使用色	偏移量表的大小

硬件编码码器输入的 avframe 接口定义

常规接口

data 数组的定义

下标	说明
0	Y 的虚拟地址
1	Cb 的虚拟地址
2	Cr 的虚拟地址
3	保留
4	Y 的物理地址
5	Cb 的物理地址
6	Cr 的物理地址
7	未使用

linesize 数组的定义

下标	说明
0	Y 的虚拟地址的跨度
1	Cb 的虚拟地址的跨度
2	Cr 的虚拟地址的跨度
3	未使用
4	Y 的物理地址的跨度
5	Cb 的物理地址的跨度
6	Cr 的物理地址的跨度
7	未使用

HWAccel 接口

data 数组的定义

下标	说明
0	Y 的物理地址
1	Cb 的物理地址
2	Cr 的物理地址
3	保留
4	保留

linesize 数组的定义

下标	说明
0	Y 的物理地址的跨度
1	Cb 的物理地址的跨度
2	Cr 的物理地址的跨度
3	未使用

硬件 filter 输入输出的 AVFrame 接口定义

1. 在不启用 HWAccel 加速功能时，AVFrame 接口定义采用常规接口的内存布局。

data 数组的定义

下标	说明
0	Y 的虚拟地址
1	Cb 的虚拟地址
2	Cr 的虚拟地址
3	保留
4	Y 的物理地址
5	Cb 的物理地址
6	Cr 的物理地址
7	未使用

linesize 数组的定义

下标	说明
0	Y 的虚拟地址的跨度
1	Cb 的虚拟地址的跨度
2	Cr 的虚拟地址的跨度
3	未使用
4	Y 的物理地址的跨度
5	Cb 的物理地址的跨度
6	Cr 的物理地址的跨度
7	未使用

2.HWAccel 接口下 AVFrame 接口定义

data 数组的定义

下标	说明	压缩格式的输入接口
0	Y 的物理地址	压缩的亮度数据的物理地址
1	Cb 物理地址	压缩的色度数据的物理地址
2	Cr 物理地址	亮度数据的偏移量表的物理地址
3	保留	色度数据的偏移量表的物理地址
4	保留	保留

linesize 数组的定义

下标	说明	缩格式的输入接口
0	Y 物理地址的跨度	亮度数据的跨度
1	Cb 物理地址的跨度	色度数据的跨度
2	Cr 物理地址的跨度	亮度偏移量表的大小
3	未使用	色度偏移量表的大小

2.3.9 硬件加速在 FFMPEG 命令中的应用示例

下面同时给出常规模式和 HWAccel 模式对应的 FFMPEG 命令行参数。

为便于理解，这里汇总说明：

- 常规模式下，bm 解码器的输出内存是否同步到系统内存上，用 zero_copy 控制，默认为 1。
- 常规模式下，bm 编码器的输入内存在系统内容还是设备内存上，用 is_dma_buffer 控制，默认值为 1。
- 常规模式下，bm 滤波器会自动判断输入内存的同步，输出内存是否同步到系统内存，用 zero_copy 控制，默认值为 0。
- HWAccel 模式下，设备内存和系统内存的同步用 hwupload 和 hwdownload 来控制。
- 常规模式下，用 sophon_idx 来指定设备，默认为 0；HWAccel 模式下用 hwaccel_device 来指定。

示例 1

使用设备 0。解码 H.265 视频，输出 compressed frame buffer，scale_bm 解压缩 compressed frame buffer 并缩放成 CIF，然后编码成 H.264 码流。

常规模式：

```
ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:zero_copy=1" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale.264
```

HWAccel 模式：

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 \
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale.264
```

示例 2

使用设备 0。解码 H.265 视频，按比例缩放并自动裁剪成 CIF，然后编码成 H.264 码流。

常规模式：

```
ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:opt=crop:zero_copy=1" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale_crop.264
```

HWAccel 模式：

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 \
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:opt=crop" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale_crop.264
```

示例 3

使用设备 0。解码 H.265 视频，按比例缩放并自动补黑边成 CIF，然后编码成 H.264 码流。

常规模式：

```
ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:opt=pad:zero_copy=1" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale_pad.264
```

HWAccel 模式:

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 \
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:opt=pad" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale_pad.264
```

示例 4

演示视频截图功能。使用设备 0。解码 H.265 视频，按比例缩放并自动补黑边成 CIF，然后编码成 jpeg 图片。

常规模式:

```
ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:opt=pad:format=yuvj420p:zero_copy=1" \
-c:v jpeg_bm -vframes 1 \
-y wkc_100_cif_scale.jpeg
```

HWAccel 模式:

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 -c:v hevc_bm -output_format 101 \
-i src/wkc_100.265 -vf "scale_bm=352:288:opt=pad:format=yuvj420p" \
-c:v jpeg_bm -vframes \
1 -y wkc_100_cif_scale.jpeg
```

示例 5

演示视频转码 + 视频截图功能。使用设备 0。硬件解码 H.265 视频，缩放成 CIF，然后编码成 H.264 码流；同时缩放成 720p，然后编码成 JPEG 图片。

常规模式:

```
ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 -filter_complex \
"[0:v]scale_bm=352:288:zero_copy=1[img1];[0:v]scale_bm=1280:720:format=yuvj420p:zero_
↪copy=1[img2]" \
-map '[img1]' -c:v h264_bm -b:v 256K -y img1.264 \
-map '[img2]' -c:v jpeg_bm -vframes 1 -y img2.jpeg
```

HWAccel 模式:

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 -c:v hevc_bm -output_format 101 \
-i src/wkc_100.265 -filter_complex \
"[0:v]scale_bm=352:288[img1];[0:v]scale_bm=1280:720:format=yuvj420p[img2]" \
-map '[img1]' -c:v h264_bm -b:v 256K -y img1.264 \
-map '[img2]' -c:v jpeg_bm -vframes 1 -y img2.jpeg
```

示例 6

演示 hwdownload 功能。硬件解码 H.265 视频，然后下载存储成 YUV 文件。

Filter hwdownload 专门为 HWAccel 接口服务，用于设备内存和系统内存的同步。在常规模式中，这步可以通过 codec 中指定 zero_copy 选项来实现，因此不需要 hwdownload 滤波器。

常规模式：

```
ffmpeg -c:v hevc_bm -cbcr_interleave 0 -zero_copy 0 \
-i src/wkc_100.265 -y test_transfer.yuv
```

HWAccel 模式

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 -c:v hevc_bm -cbcr_interleave 0 \
-i src/wkc_100.265 -vf "hwdownload,format=yuv420p|bmcodec" -y test_transfer.yuv
```

示例 7

演示 hwdownload 功能。硬件解码 H.265 视频，缩放成 CIF 格式，然后下载存储成 YUV 文件。

在常规模式中，scale_bm 会自动根据 filter 的链条判定是否同步内存，因此不需要 hwdownload。

常规模式：

```
ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288,format=yuv420p" \
-y test_transfer_cif.yuv
```

HWAccel 模式：

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 \
-c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288,hwdownload,format=yuv420p|bmcodec" \
-y test_transfer_cif.yuv
```

示例 8

演示 hwupload 功能。使用设备 0。上传 YUV 视频，然后编码 H.264 视频。

Filter hwupload 专门为 HWAccel 接口服务，用于设备内存和系统内存的同步。在常规模式中，这步可以通过编码器中指定 is_dma_buffer 选项来实现，因此不需要 hwupload 滤波器。

常规模式：

```
ffmpeg -s 1920x1080 -pix_fmt yuv420p -i test_transfer.yuv \
-c:v h264_bm -b:v 3M -is_dma_buffer 0 -y test_transfer.264
```

HWAccel 模式:

```
ffmpeg -init_hw_device bmcodec=foo:0 \
-s 1920x1080 -i test_transfer.yuv \
-filter_hw_device foo -vf "format=yuv420p|bmcodec,hwupload" \
-c:v h264_bm -b:v 3M -y test_transfer.264
```

这里 foo 为设备 0 的别名。

示例 9

演示 hwupload 功能。使用设备 1。上传 YUV 视频，并缩放成 CIF，然后编码 H.264 视频。

常规模式:

```
ffmpeg -s 1920x1080 -i test_transfer.yuv \
-vf "scale_bm=352:288:sophon_idx=0:zero_copy=1" \
-c:v h264_bm -b:v 256K -sophon_idx 0 \
-y test_transfer_cif.264
```

说明：1）这里不指定-pix_fmt yuv420p 是因为默认输入为 yuv420p 格式

2）常规模式下,bm_scale filter, bm_overlay filter, decoder，encoder 通过参数 sophon_idx 来指定使用哪个设备

HWAccel 模式:

```
ffmpeg -init_hw_device bmcodec=foo:1 \
-s 1920x1080 -i test_transfer.yuv -filter_hw_device foo \
-vf "format=yuv420p|bmcodec,hwupload,scale_bm=352:288" \
-c:v h264_bm -b:v 256K -y test_transfer_cif.264
```

说明：这里 foo 为设备 1 的别名，HWAccel 模式下通过 init_hw_device 来指定使用具体的硬件设备。

示例 10

演示 hwdownload 功能。硬件解码 YUVJ444P 的 JPEG 视频，然后下载存储成 YUV 文件。

常规模式:

```
ffmpeg -c:v jpeg_bm -zero_copy 0 -i src/car/1920x1080_yuvj444.jpg \
-y car_1080p_yuvj444_dec.yuv
```

HWAccel 模式:

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 \
-c:v jpeg_bm -i src/car/1920x1080_yuvj444.jpg \
-vf "hwdownload,format=yuvj444p|bmcodec" \
-y car_1080p_yuvj444_dec.yuv
```


示例 11

演示 hwupload 功能。使用设备 1。上传 YUVJ444P 图像数据，然后编码 JPEG 图片。

常规模式：

```
ffmpeg -s 1920x1080 -pix_fmt yuvj444p -i car_1080p_yuvj444.yuv \
-frames:v 1 -c:v jpeg_bm -sophon_idx 1 -is_dma_buffer 0 \
-y car_1080p_yuvj444_enc.jpg
```

HWAccel 模式：

```
ffmpeg -init_hw_device bmcodec=foo:1 -filter_hw_device foo \
-s 1920x1080 -pix_fmt yuvj444p -i car_1080p_yuvj444.yuv \
-frames:v 1 -vf 'format=yuvj444p|bmcodec,hwupload' \
-c:v jpeg_bm -y car_1080p_yuvj444_enc.jpg
```

这里 foo 为设备 1 的别名。

示例 12

演示软解码和硬编码混合使用的方法。使用设备 2。使用 ffmpeg 自带的 h264 软解码器，解码 H.264 视频，上传解码后数据到芯片 2，然后编码 H.265 视频。

常规模式：

```
ffmpeg -c:v h264 -i src/1920_18MG.mp4 \
-c:v h265_bm -is_dma_buffer 0 -sophon_idx 2 -g 256 -b:v 5M \
-y test265.mp4
```

HWAccel 模式：

```
ffmpeg -init_hw_device bmcodec=foo:2 -c:v h264 -i src/1920_18MG.mp4 \
-filter_hw_device foo -vf 'format=yuv420p|bmcodec,hwupload' \
-c:v h265_bm -g 256 -b:v 5M -y test265.mp4
```

这里 foo 为设备 2 的别名。

示例 13

演示使用 enc-params 设置视频编码器的方法。使用设备 0。解码 H.265 视频，缩放成 CIF，然后编码成 H.264 码流。

常规模式：

```
ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:zero_copy=1" -c:v h264_bm -g 50 -b:v 32K \
-enc-params "gop_preset=2:mb_rc=1:delta_qp=3:min_qp=20:max_qp=40" \
-y wkc_100_cif_scale_ipp_32Kbps.264
```

HWAccel 模式:

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 -c:v hevc_bm -output_format 101 \
-i src/wkc_100.265 -vf "scale_bm=352:288" -c:v h264_bm -g 50 -b:v 32K \
-enc-params "gop_preset=2:mb_rc=1:delta_qp=3:min_qp=20:max_qp=40" \
-y wkc_100_cif_scale_ipp_32Kbps.264
```

示例 14

使用设备 0。解码 H.265 视频，使用 bilinear 滤波器，按比例缩放成 CIF，并自动补黑边，然后编码成 H.264 码流。

常规模式:

```
ffmpeg -c:v hevc_bm -output_format 101 -i src/wkc_100.265 \
-vf "scale_bm=352:288:opt=pad:flags=bilinear:zero_copy=1" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale_pad.264
```

HWAccel 模式:

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 -c:v hevc_bm -output_format 101 \
-i src/wkc_100.265 -vf "scale_bm=352:288:opt=pad:flags=bilinear" \
-c:v h264_bm -g 256 -b:v 256K \
-y wkc_100_cif_scale_pad.264
```

示例 15

演示 overlay 在图片上叠加图片滤镜功能。上传两个 JPEG 图片，overlay 处理后，保存为 JPEG 文件。

常规模式:

```
ffmpeg -zero_copy 1 -c:v jpeg_bm -i JPEG_1920x1088_yuv420_planar.jpg \
-zero_copy 1 -c:v jpeg_bm -i JPEG_1280x720_yuv420_planar.jpg \
-filter_complex "overlay_bm=x=99:y=99:eof_action=1:zero_copy=1" \
-is_dma_buffer 1 -c:v jpeg_bm overlay_jpeg_to_jpeg.jpg
```

HWAccel 模式:

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 -zero_copy 1 -c:v jpeg_bm \
-i JPEG_1920x1088_yuv420_planar.jpg \
-hwaccel bmcodec -hwaccel_device 0 -zero_copy 1 -c:v jpeg_bm \
-i JPEG_1280x720_yuv420_planar.jpg \
-filter_complex "overlay_bm=x=0:y=0:eof_action=1:zero_copy=1" \
-is_dma_buffer 1 -c:v jpeg_bm overlay_jpeg_to_jpeg.jpg
```

示例 16

演示 overlay 在视频上叠加图片滤镜功能。上传一个 264 视频和一个 JPEG 图片，overlay 处理后，保存为 264 文件。

常规模式：

```
ffmpeg -zero_copy 0 -extra_frame_buffer_num 5 -c:v h264_bm -i 1920x1080.264 \
-zero_copy 0 -c:v jpeg_bm -i JPEG_1280x720_yuv420_planar.jpg \
-filter_complex "scale_bm=1920:1088,overlay_bm=x=0:y=0:eof_action=0:zero_copy=0" \
-pix_fmt yuv420p -is_dma_buffer 0 -qp 30 -c:v h264_bm -vframes 100 overlay_jpeg_to_video.264
```

HWAccel 模式：

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 -zero_copy 0 -extra_frame_buffer_num 5 \
-c:v h264_bm -i 1920x1080.264 \
-hwaccel bmcodec -hwaccel_device 0 -zero_copy 0 -c:v jpeg_bm \
-i JPEG_1280x720_yuv420_planar.jpg \
-filter_complex "scale_bm=1920:1088,overlay_bm=x=0:y=0:eof_action=0:zero_copy=0,
↪hwdownload,format=yuv420p|bmcodec" \
-pix_fmt yuv420p -is_dma_buffer 0 -qp 30 -c:v h264_bm -vframes 100 overlay_jpeg_to_video.264
```

示例 17

演示 overlay 在视频上叠加视频滤镜功能。上传两个 264 视频，overlay 处理后，保存为 264 文件。

常规模式：

```
ffmpeg -zero_copy 1 -extra_frame_buffer_num 5 -c:v h264_bm -i 1920x1080.264 \
-c:v h264_bm -extra_frame_buffer_num 2 -zero_copy 1 -i test_352x288.264 \
-filter_complex "overlay_bm=x=300:y=100:eof_action=1:zero_copy=1" \
-is_dma_buffer 1 -qp 30 -c:v h264_bm -vframes 6 -an overlay_video_to_video.264
```

HWAccel 模式：

```
ffmpeg -hwaccel bmcodec -hwaccel_device 0 -zero_copy 1 -extra_frame_buffer_num 10 \
-c:v h264_bm -i 1920x1080.264 \
-hwaccel bmcodec -hwaccel_device 0 -c:v h264_bm -extra_frame_buffer_num 2 \
-zero_copy 1 -i test_352x288.264 \
-filter_complex "overlay_bm=x=10:y=10:eof_action=0:zero_copy=1" \
-is_dma_buffer 0 -qp 30 -c:v h264_bm -vframes 6 -an overlay_video_to_video.264
```

2.3.10 通过调用 API 方式来使用硬件加速功能

examples/multimedia/ff_bmcv_transcode/例子演示了使用 ffmpeg 做编解码，用 bmcv 做图像处理的整个流程。

examples/multimedia/ff_video_decode/例子演示了使用 ffmpeg 做解码的流程。

examples/multimedia/ff_video_encode/例子演示了使用 ffmpeg 做编码的流程。

2.3.11 硬件编码支持 roi 编码

参考 examples/multimedia/ff_video_encode/例子。设置 roi_enable 既可启用 roi 编码。

Roi 编码数据通过 av_frame side data 传递。

Roi 数据结构定义为

```
609 typedef union {
610     struct {
611         int mb_force_mode;
612         int mb_qp;
613     }H264;
614
615     struct {
616         int ctu_force_mode;
617         int ctu_coeff_drop;
618
619         int sub_ctu_qp_0;
620         int sub_ctu_qp_1;
621         int sub_ctu_qp_2;
622         int sub_ctu_qp_3;
623
624         int lambda_sad_0;
625         int lambda_sad_1;
626         int lambda_sad_2;
627         int lambda_sad_3;
628     }HEVC;
629 } RoiField;
630
631 typedef struct AVBMRoiInfo {
632     // int numbers;
633     /* Enable ROI map. */
634     int customRoiMapEnable;
635     /* Enable custom lambda map. */
636     int customLambdaMapEnable;
637     /* Force CTU to be encoded with intra or to be skipped.  */
638     int customModeMapEnable;
639     /* Force all coefficients to be zero after TQ or not for each CTU (to be dropped).*/
640     int customCoefDropEnable;
641
642     RoiField field[0x40000];
643 } AVBMRoiInfo;
644
```

字段说明:

- QP Map

  H264 下 QP 以宏块 16x16 为单位给出。HEVC 下 QP 以 sub-ctu（32x32）为单位给出。QP 对应的就是 video 编码中的 Qstep，取值范围为 0-51.

- Lamda Map

lamda 是用来控制和调节 IP 内部的 RC 计算公式

cost = distortion + lamda * rate

这个调节参数仅在 HEVC 下有效，允许以 32x32 sub-CTU 模块为单位控制。

- Mode Map

  这个参数用来指定模式选择。0 –不适用 1 –skip mode 2- intra mode。H264 下以宏块 16x16 为单位控制，HEVC 下以 CTU 64x64 为单位控制。

- Zero-cut Flag

  仅在 HEVC 下有效。将当前 CTU 64x64 残差系数全部置为 0，从而节省出更多的比特给其他更重要的部分。

2.4 SOPHGO LIBYUV 使用指南

2.4.1 简介

BM1688 系列芯片中的各种硬件模块，可以加速对图片和视频的处理。颜色转换方面，采用专用硬件来加速速度很快。

但在有些场合，也会存在一些专用硬件覆盖不到的特殊情况。此时采用经过 SIMD 加速优化的软件实现, 成为专用硬件有力的补充。

SOPHGO 增强版 **libyuv**，是随同 SDK 一同发布的一个组件。目的是充分利用 BM1688 系列芯片提供的 8 核 A53 处理器，通过软件手段为硬件的局限性提供补充。

除了 libyuv 提供的标准函数之外，针对 AI 的需求，在 SOPHGO 增强版 libyuv 中，补充了 27 个扩展函数。

注意：这里说的是运行在 BM1688 系列的 A53 处理器上，而不是 host 的处理器。这从设备加速的角度是可以理解的。这样可以避免占用 host 的 CPU。

2.4.2 libyuv 扩展说明

新增了如下增强 AI 应用方面的 API。

fast_memcpy

void* fast_memcpy(void *dst, const void *src, size_t n)

功能	CPU SIMD 指令实现 memcpy 功能。从内存区域 src 拷贝 n 个字节到内存区域 dst	
参数	src	源内存区域
	n	需要拷贝的字节数
	dst	目的内存区域
返回值	返回一个指向 dst 的指针	

RGB24ToI400

int RGB24ToI400(const uint8_t* src_rgb24, int src_stride_rgb24, uint8_t* dst_y, int dst_stride_y, int width, int height);

功能	将一帧 BGR 数据转换成 BT.601 灰度数据	
参数	src_rgb24	packed BGR 图像数据所在的内存虚地址
	src_stride_rgb24	内存中每行 BGR 图像实际跨度
	dst_y	灰度图像虚拟地址
	dst_stride_y	内存中每行灰度图像实际跨度
	width	每行 BGR 图像数据中 packed BGR 的数量
	height	BGR 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

RAWToI400

int RAWToI400(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y, int dst_stride_y, int width, int height);

功能	将一帧 RGB 数据转换成 BT.601 灰度数据	
参数	src_row	packed BGR 图像数据所在的内存虚地址
	src_stride_row	内存中每行 BGR 图像实际跨度
	dst_y	灰度图像虚拟地址
	dst_stride_y	内存中每行灰度图像实际跨度
	width	每行 BGR 图像数据中 packed BGR 的数量
	height	BGR 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

I400ToRGB24

int I400ToRGB24(const uint8_t* src_y, int src_stride_y, uint8_t* dst_rgb24, int dst_stride_rgb24, int width, int height);

功能	将一帧 BT.601 灰度数据转换成 BGR 数据	
参数	src_y	灰度图像虚拟地址
	src_stride_y	内存中每行灰度图像实际跨度
	dst_rgb24	packed BGR 图像数据所在的内存虚地址
	dst_stride_rgb24	内存中每行 BGR 图像实际跨度
	width	每行 BGR 图像数据中 packed BGR 的数量
	height	BGR 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

I400ToRAW

int I400ToRAW(const uint8_t* src_y, int src_stride_y, uint8_t* dst_raw, int dst_stride_raw, int width, int height);

功能	将一帧 BT.601 灰度数据转换成 RGB 数据	
参数	src_y	灰度图像虚拟地址
	src_stride_y	内存中每行灰度图像实际跨度
	dst_rgb24	packed RGB 图像数据所在的内存虚地址
	dst_stride_rgb24	内存中每行 RGB 图像实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

J400ToRGB24

int J400ToRGB24(const uint8_t* src_y, int src_stride_y, uint8_t* dst_rgb24, int dst_stride_rgb24, int width, int height);

功能	将一帧 BT.601 full range 灰度数据转换成 BGR 数据	
参数	src_y	灰度图像虚拟地址
	src_stride_y	内存中每行灰度图像实际跨度
	dst_rgb24	packed BGR 图像数据所在的内存虚地址
	dst_stride_rgb24	内存中每行 BGR 图像实际跨度
	width	每行 BGR 图像数据中 packed BGR 的数量
	height	BGR 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

RAWToJ400

int RAWToJ400(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y, int dst_stride_y, int width, int height);

功能	将一帧 RGB 数据转换成 BT.601 full range 灰度数据	
参数	src_raw	packed RGB 图像数据所在的内存虚地址
	src_stride_raw	内存中每行 RGB 图像实际跨度
	dst_y	灰度图像虚拟地址
	dst_stride_y	内存中每行灰度图像实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

J400ToRAW

int J400ToRAW(const uint8_t* src_y, int src_stride_y, uint8_t* dst_raw, int dst_stride_raw, int width, int height);

功能	将一帧 BT.601 full range 灰度数据转换成 RGB 数据	
参数	src_y	灰度图像虚拟地址
	src_stride_y	内存中每行灰度图像实际跨度
	dst_rgb24	packed RGB 图像数据所在的内存虚地址
	dst_stride_rgb24	内存中每行 RGB 图像实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

RAWToNV12

int RAWToNV12(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_uv, int dst_stride_uv, int width, int height);

功能	将一帧 RGB 数据转换成 BT.601 limited range 的 semi-planar YCbCr 420 数据	
参数	src_raw	packed RGB 图像数据所在的内存虚地址
	src_stride_raw	内存中每行 RGB 图像实际跨度
	dst_y	Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_uv	CbCr 分量的虚拟地址
	dst_stride_uv	内存中每行 CbCr 分量数据的实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

RGB24ToNV12

int RGB24ToNV12(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_uv, int dst_stride_uv, int width, int height);

功能	将一帧 BGR 数据转换成 BT.601 limited range 的 semi-planar YCbCr 420 数据	
参数	src_raw	packed BGR 图像数据所在的内存虚地址
	src_stride_raw	内存中每行 BGR 图像实际跨度
	dst_y	Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_uv	CbCr 分量的虚拟地址
	dst_stride_uv	内存中每行 CbCr 分量数据的实际跨度
	width	每行 BGR 图像数据中 packed BGR 的数量
	height	BGR 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

RAWToJ420

int RAWToJ420(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int dst_stride_v, int width, int height);

功能	将一帧 RGB 数据转换成 BT.601 full range 的 semi-planar YCbCr 420 数据	
参数	src_raw	packed RGB 图像数据所在的内存虚地址
	src_stride_raw	内存中每行 RGB 图像实际跨度
	dst_y	Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_u	Cb 分量的虚拟地址
	dst_stride_u	内存中每行 Cb 分量数据的实际跨度
	dst_v	Cr 分量的虚拟地址
	dst_stride_v	内存中每行 Cr 分量数据的实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

J420ToRAW

int J420ToRAW(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u, int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_raw, int dst_stride_raw, int width, int height);

功能	将一帧 BT.601 full range 的 YCbCr 420 数据转换成 RGB 数据	
参数	src_y	Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	src_u	Cb 分量的虚拟地址
	src_stride_u	内存中每行 Cb 分量数据的实际跨度
	src_v	Cr 分量的虚拟地址
	src_stride_v	内存中每行 Cr 分量数据的实际跨度
	dst_raw	packed RGB 图像数据所在的内存虚地址
	dst_stride_raw	内存中每行 RGB 图像数据的实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

RAWToJ422

int RAWToJ422(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int dst_stride_v, int width, int height);

功能	将一帧 RGB 数据转换成 BT.601 full range 的 YCbCr 422 数据	
参数	src_raw	packed RGB 图像数据所在的内存虚地址
	src_stride_raw	内存中每行 RGB 图像实际跨度
	dst_y	Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_u	Cb 分量的虚拟地址
	dst_stride_u	内存中每行 Cb 分量数据的实际跨度
	dst_v	Cr 分量的虚拟地址
	dst_stride_v	内存中每行 Cr 分量数据的实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

J422ToRAW

int J422ToRAW(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u, int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_raw, int dst_stride_raw, int width, int height);

功能	将一帧 BT.601 full range 的 YCbCr 422 数据转换成 RGB 数据	
参数	src_y	Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	src_u	Cb 分量的虚拟地址
	src_stride_u	内存中每行 Cb 分量数据的实际跨度
	src_v	Cr 分量的虚拟地址
	src_stride_v	内存中每行 Cr 分量数据的实际跨度
	dst_raw	packed RGB 图像数据所在的内存虚地址
	dst_stride_raw	内存中每行 RGB 图像数据的实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

RGB24ToJ422

int RGB24ToJ422(const uint8_t* src_rgb24, int src_stride_rgb24, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int dst_stride_v, int width, int height);

功能	将一帧 BGR 数据转换成 BT.601 full range 的 YCbCr 422 数据	
参数	src_rgb24	packed BGR 图像数据所在的内存虚地址
	src_stride_rgb24	内存中每行 BGR 图像实际跨度
	dst_y	Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_u	Cb 分量的虚拟地址
	dst_stride_u	内存中每行 Cb 分量数据的实际跨度
	dst_v	Cr 分量的虚拟地址
	dst_stride_v	内存中每行 Cr 分量数据的实际跨度
	width	每行 BGR 图像数据中 packed BGR 的数量
	height	BGR 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

J422ToRGB24

int J422ToRGB24(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u, int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_rgb24, int dst_stride_rgb24, int width, int height);

功能	将一帧 BT.601 full range 的 YCbCr 422 数据转换成 BGR 数据	
参数	src_y	Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	src_u	Cb 分量的虚拟地址
	src_stride_u	内存中每行 Cb 分量数据的实际跨度
	src_v	Cr 分量的虚拟地址
	src_stride_v	内存中每行 Cr 分量数据的实际跨度
	dst_rgb24	packed BGR 图像数据所在的内存虚地址
	dst_stride_rgb24	内存中每行 BGR 图像数据的实际跨度
	width	每行 RGB 图像数据中 packed BGR 的数量
	height	BGR 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

RAWToJ444

int RAWToJ444(const uint8_t* src_raw, int src_stride_raw, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int dst_stride_v, int width, int height);

功能	将一帧 RGB 数据转换成 BT.601 full range 的 YCbCr 444 数据	
参数	src_row	packed RGB 图像数据所在的内存虚地址
	src_stride_row	内存中每行 RGB 图像实际跨度
	dst_y	Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_u	Cb 分量的虚拟地址
	dst_stride_u	内存中每行 Cb 分量数据的实际跨度
	dst_v	Cr 分量的虚拟地址
	dst_stride_v	内存中每行 Cr 分量数据的实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

J444ToRAW

int J444ToRAW(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u, int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_raw, int dst_stride_raw, int width, int height);

功能	将一帧 BT.601 full range 的 YCbCr 444 数据转换成 RGB 数据	
参数	src_y	Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	src_u	Cb 分量的虚拟地址
	src_stride_u	内存中每行 Cb 分量数据的实际跨度
	src_v	Cr 分量的虚拟地址
	src_stride_v	内存中每行 Cr 分量数据的实际跨度
	dst_raw	packed RGB 图像数据所在的内存虚地址
	dst_stride_raw	内存中每行 RGB 图像数据的实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

RGB24ToJ444

int RGB24ToJ444(const uint8_t* src_rgb24, int src_stride_rgb24, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int dst_stride_v, int width, int height);

功能	将一帧 BGR 数据转换成 BT.601 full range 的 YCbCr 444 数据	
参数	src_rgb24	packed BGR 图像数据所在的内存虚地址
	src_stride_rgb24	内存中每行 BGR 图像实际跨度
	dst_y	Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_u	Cb 分量的虚拟地址
	dst_stride_u	内存中每行 Cb 分量数据的实际跨度
	dst_v	Cr 分量的虚拟地址
	dst_stride_v	内存中每行 Cr 分量数据的实际跨度
	width	每行 BGR 图像数据中 packed BGR 的数量
	height	BGR 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

J444ToRGB24

int J444ToRGB24(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u, int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_rgb24, int dst_stride_rgb24, int width, int height);

功能	将一帧 BT.601 full range 的 YCbCr 444 数据转换成 BGR 数据	
参数	src_y	Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	src_u	Cb 分量的虚拟地址
	src_stride_u	内存中每行 Cb 分量数据的实际跨度
	src_v	Cr 分量的虚拟地址
	src_stride_v	内存中每行 Cr 分量数据的实际跨度
	dst_rgb24	packed BGR 图像数据所在的内存虚地址
	dst_stride_rgb24	内存中每行 BGR 图像数据的实际跨度
	width	每行 RGB 图像数据中 packed BGR 的数量
	height	BGR 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

H420ToJ420

int H420ToJ420(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u, int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int dst_stride_v, int width, int height);

功能	将一帧 BT.709 limited range 的 YCbCr 420 数据转换成 BT.601 full range 的数据。可以在 jpeg 编码之前，作预处理函数使用	
参数	src_y	Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	src_u	Cb 分量的虚拟地址
	src_stride_u	内存中每行 Cb 分量数据的实际跨度
	src_v	Cr 分量的虚拟地址
	src_stride_v	内存中每行 Cr 分量数据的实际跨度
	dst_y	Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_u	Cb 分量的虚拟地址
	dst_stride_u	内存中每行 Cb 分量数据的实际跨度
	dst_v	Cr 分量的虚拟地址
	dst_stride_v	内存中每行 Cr 分量数据的实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

I420ToJ420

int I420ToJ420(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u, int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int dst_stride_v, int width, int height);

功能	将一帧 BT.601 limited range 的 YCbCr 420 数据转换成 BT.601 full range 的数据。可以在 jpeg 编码之前，作预处理函数使用	
参数	src_y	Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	src_u	Cb 分量的虚拟地址
	src_stride_u	内存中每行 Cb 分量数据的实际跨度
	src_v	Cr 分量的虚拟地址
	src_stride_v	内存中每行 Cr 分量数据的实际跨度
	dst_y	Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_u	Cb 分量的虚拟地址
	dst_stride_u	内存中每行 Cb 分量数据的实际跨度
	dst_v	Cr 分量的虚拟地址
	dst_stride_v	内存中每行 Cr 分量数据的实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

NV12ToJ420

int NV12ToJ420(const uint8_t* src_y, int src_stride_y, const uint8_t* src_uv, int src_stride_uv, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int dst_stride_v, int width, int height);

功能	将一帧 BT.601 limited range 的 semi-plannar YCbCr 420 数据转换成 BT.601 full range 的数据。可以在 jpeg 编码之前，作预处理函数使用	
参数	src_y	Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	src_uv	CbCr 分量的虚拟地址
	src_stride_uv	内存中每行 CbCr 分量数据的实际跨度
	dst_y	Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_u	Cb 分量的虚拟地址
	dst_stride_u	内存中每行 Cb 分量数据的实际跨度
	dst_v	Cr 分量的虚拟地址
	dst_stride_v	内存中每行 Cr 分量数据的实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

NV21ToJ420

int NV21ToJ420(const uint8_t* src_y, int src_stride_y, const uint8_t* src_vu, int src_stride_vu, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_u, int dst_stride_u, uint8_t* dst_v, int dst_stride_v, int width, int height);

功能	将一帧 BT.601 limited range 的 semi-plannar YCbCr 420 数据转换成 BT.601 full range 的数据。可以在 jpeg 编码之前，作预处理函数使用	
参数	src_y	Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	src_vu	CrCb 分量的虚拟地址
	src_stride_vu	内存中每行 CrCb 分量数据的实际跨度
	dst_y	Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_u	Cb 分量的虚拟地址
	dst_stride_u	内存中每行 Cb 分量数据的实际跨度
	dst_v	Cr 分量的虚拟地址
	dst_stride_v	内存中每行 Cr 分量数据的实际跨度
	width	每行 RGB 图像数据中 packed RGB 的数量
	height	RGB 图像数据的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

I444ToNV12

int I444ToNV12(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u, int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_uv, int dst_stride_uv, int width, int height);

功能	将一帧 YCbCr 444 数据转换成 semi-plannar YCbCr 420 数据。可用于 full range，也可以用于 limited range 的数据。不涉及颜色空间转换，可灵活使用	
参数	src_y	源图像 Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	src_u	源图像 Cb 分量的虚拟地址
	src_stride_u	内存中每行 Cb 分量数据的实际跨度
	src_v	源图像 Cr 分量的虚拟地址
	src_stride_v	内存中每行 Cr 分量数据的实际跨度
	dst_y	目的图像 Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_uv	目的图像 CbCr 分量的虚拟地址
	dst_stride_uv	内存中每行 CbCr 分量数据的实际跨度
	width	每行图像数据中像素的数量
	height	图像数据像素的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

I422ToNV12

int I422ToNV12(const uint8_t* src_y, int src_stride_y, const uint8_t* src_u, int src_stride_u, const uint8_t* src_v, int src_stride_v, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_uv, int dst_stride_uv, int width, int height);

功能	将一帧 YCbCr 422 数据转换成 semi-plannar YCbCr 420 数据。可用于 full range，也可以用于 limited range 的数据。不涉及颜色空间转换，可灵活使用	
参数	src_y	源图像 Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	src_u	源图像 Cb 分量的虚拟地址
	src_stride_u	内存中每行 Cb 分量数据的实际跨度
	src_v	源图像 Cr 分量的虚拟地址
	src_stride_v	内存中每行 Cr 分量数据的实际跨度
	dst_y	目的图像 Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_uv	目的图像 CbCr 分量的虚拟地址
	dst_stride_uv	内存中每行 CbCr 分量数据的实际跨度
	width	每行图像数据中像素的数量
	height	图像数据像素的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

I400ToNV12

int I400ToNV12(const uint8_t* src_y, int src_stride_y, uint8_t* dst_y, int dst_stride_y, uint8_t* dst_uv, int dst_stride_uv, int width, int height);

功能	将一帧 YCbCr 400 数据转换成 semi-plannar YCbCr 420 数据。可用于 full range，也可以用于 limited range 的数据。不涉及颜色空间转换，可灵活使用	
参数	src_y	源图像 Y 分量的虚拟地址
	src_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_y	目的图像 Y 分量的虚拟地址
	dst_stride_y	内存中每行 Y 分量数据的实际跨度
	dst_uv	目的图像 CbCr 分量的虚拟地址
	dst_stride_uv	内存中每行 CbCr 分量数据的实际跨度
	width	每行图像数据中像素的数量
	height	图像数据像素的有效行数
返回值	0, 正常结束; 非 0, 参数异常。	

2.5 SOPHGO JPEG 使用指南

2.5.1 简介

JPEG 在 BM1688 产品系列中是一个可以实现多种图片格式编解码的功能模块。该模块包含编码、解码、镜像、旋转及 ROI 等功能，支持各种 YUV 格式。可以支持的分辨率范围为 16x16~32768x32768，通过硬件加速，编解码性能也十分出色。

2.5.2 JPEG 数据结构说明

BmJpuLogLevel

该类型表示 jpeg 动态库内日志等级。

BmJpuLogLevel 类型定义如下：

```
typedef enum
{
    BM_JPU_LOG_LEVEL_ERROR   = 0,
    BM_JPU_LOG_LEVEL_WARNING = 1,
    BM_JPU_LOG_LEVEL_INFO    = 2,
    BM_JPU_LOG_LEVEL_DEBUG   = 3,
    BM_JPU_LOG_LEVEL_LOG     = 4,
    BM_JPU_LOG_LEVEL_TRACE   = 5
} BmJpuLogLevel;
```

BmJpuImageFormat

该类型表示图像的格式。

BmJpuImageFormat 类型定义如下：

```
typedef enum
{
    BM_JPU_IMAGE_FORMAT_YUV420P = 0,
    BM_JPU_IMAGE_FORMAT_YUV422P,
    BM_JPU_IMAGE_FORMAT_YUV444P,
    BM_JPU_IMAGE_FORMAT_NV12,
    BM_JPU_IMAGE_FORMAT_NV21,
    BM_JPU_IMAGE_FORMAT_NV16,
    BM_JPU_IMAGE_FORMAT_NV61,
    BM_JPU_IMAGE_FORMAT_GRAY,
    BM_JPU_IMAGE_FORMAT_RGB  /* for opencv */
} BmJpuImageFormat;
```

BmJpuColorFormat

该类型表示图像的像素格式。

BmJpuColorFormat 类型定义如下：

```
typedef enum
{
    /* planar 4:2:0; if the chroma_interleave parameter is 1, the corresponding format
↪is NV12, otherwise it is I420 */
    BM_JPU_COLOR_FORMAT_YUV420          = 0,
    /* planar 4:2:2; if the chroma_interleave parameter is 1, the corresponding format
↪is NV16 */
    BM_JPU_COLOR_FORMAT_YUV422_HORIZONTAL = 1,
    /* 4:2:2 vertical, actually 2:2:4 (according to the JPU docs); no corresponding
↪format known for the chroma_interleave=1 case */
    /* NOTE: this format is rarely used, and has only been seen in a few JPEG files */
    BM_JPU_COLOR_FORMAT_YUV422_VERTICAL  = 2,
    /* planar 4:4:4; if the chroma_interleave parameter is 1, the corresponding format
↪is NV24 */
    BM_JPU_COLOR_FORMAT_YUV444          = 3,
    /* 8-bit greayscale */
    BM_JPU_COLOR_FORMAT_YUV400          = 4,
    /* RGBP */
    BM_JPU_COLOR_FORMAT_RGB             = 5,
    /* BUTT */
    BM_JPU_COLOR_FORMAT_BUTT
} BmJpuColorFormat;
```

- **BM_JPU_COLOR_FORMAT_YUV420**

  YUV420 格式。在 cbcr_interleave=1 的时候对应 NV12 格式。

- **BM_JPU_COLOR_FORMAT_YUV422_HORIZONTAL**

  YUV422 格式。在 cbcr_interleave=1 的时候对应 NV16 格式。

- **BM_JPU_COLOR_FORMAT_YUV422_VERTICAL**

  YUV422 格式（实际按 2：2：4 排列），JPU 定义的一种格式，不常用。

- **BM_JPU_COLOR_FORMAT_YUV444**

  YUV444 格式。在 cbcr_interleave=1 的时候对应 NV24 格式。

- **BM_JPU_COLOR_FORMAT_YUV400**

  YUV400 格式。对应灰度图，只有 Y 分量。

- **BM_JPU_COLOR_FORMAT_RGB**

  RGB 格式，暂未使用。

- **BM_JPU_COLOR_FORMAT_BUTT**

  未定义格式，用于设置默认值和异常判断。

BmJpuChromaFormat

该类型表示色度格式。

BmJpuChromaFormat 类型定义如下：

```
typedef enum
{
    BM_JPU_CHROMA_FORMAT_CBCR_SEPARATED = 0,
    BM_JPU_CHROMA_FORMAT_CBCR_INTERLEAVE = 1,
    BM_JPU_CHROMA_FORMAT_CRCB_INTERLEAVE = 2
} BmJpuChromaFormat;
```

- **BM_JPU_CHROMA_FORMAT_CBCR_SEPARATED**

  cb 与 cr 分开，不交织。

- **BM_JPU_CHROMA_FORMAT_CBCR_INTERLEAVE**

  cb 与 cr 交织。

- **BM_JPU_CHROMA_FORMAT_CRCB_INTERLEAVE**

  cr 与 cb 交织。

BmJpuRotateAngle

旋转角度。

BmJpuRotateAngle 类型定义如下：

```
typedef enum
{
    BM_JPU_ROTATE_NONE = 0,
    BM_JPU_ROTATE_90 = 90,
    BM_JPU_ROTATE_180 = 180,
    BM_JPU_ROTATE_270 = 270
} BmJpuRotateAngle;
```

- **BM_JPU_ROTATE_NONE**

  不旋转。

- **BM_JPU_ROTATE_90**

  逆时针旋转 90 度。

- **BM_JPU_ROTATE_180**

  逆时针旋转 180 度。

- **BM_JPU_ROTATE_270**

  逆时针旋转 270 度。

BmJpuMirrorDirection

镜像方向。

BmJpuMirrorDirection 类型定义如下：

```
typedef enum
{
    BM_JPU_MIRROR_NONE = 0,
    BM_JPU_MIRROR_VER = 1,
    BM_JPU_MIRROR_HOR = 2,
    BM_JPU_MIRROR_HOR_VER = 3
} BmJpuMirrorDirection;
```

- **BM_JPU_MIRROR_NONE**

  不旋转。

- **BM_JPU_MIRROR_VER**

  竖直镜像。

- **BM_JPU_MIRROR_HOR**

  水平镜像。

- **BM_JPU_MIRROR_HOR_VER**

  竖直水平镜像。

BmJpuFramebuffer

该结构体用于在 BMAPI 中描述一帧 YUV 数据。

BmJpuFramebuffer 结构体定义如下：

```
typedef struct {
    unsigned int y_stride;
    unsigned int cbcr_stride;
    bm_device_mem_t *dma_buffer;
    size_t y_offset;
    size_t cb_offset;
    size_t cr_offset;

    void *context;
    int already_marked;
    void *internal;
} BmJpuFramebuffer;
```

- **y_stride**

  亮度（Y）分量的步长。

- **cbcr_stride**

  色度（Cb、Cr）分量的步长。

- **dma_buffer**

  用于存放 YUV 数据的一块设备内存，由 bmlib 分配。

- **y_offset**

  Y 分量起始地址相对于 dma_buffer 中物理地址的偏移量。

- **cb_offset**

  Cb 分量起始地址相对于 dma_buffer 中物理地址的偏移量。

- **cr_offset**

  Cr 分量起始地址相对于 dma_buffer 中物理地址的偏移量。

- **context**

  用于保存解码上下文信息。

- **already_marked**

  标记当前帧是否可以输出。

- **internal**

  内部数据，由用户决定如何使用。

BmJpuFramebufferSizes

该结构体用于记录对齐之后的 framebuffer 信息

BmJpuFramebufferSizes 结构体定义如下：

```
typedef struct{
    unsigned int aligned_frame_width, aligned_frame_height;
    unsigned int y_stride, cbcr_stride;
    unsigned int y_size, cbcr_size;
    unsigned int total_size;
    BmJpuImageFormat image_format;
} BmJpuFramebufferSizes;
```

- **aligned_frame_width**

  对齐之后的宽度。

- **aligned_frame_height**

  对齐之后的高度。

- **y_stride**

  亮度（Y）分量的步长。

- **cbcr_stride**

  色度（Cb、Cr）分量的步长。

- **y_size**

  Y 分量的总数据量，单位：byte。

- **cbcr_size**

  Cb、Cr 分量的总数据量，单位：byte。

- **total_size**

  framebuffer 的总数据量，单位：byte。

- **image_format**

  图像格式，参考 BmJpuImageFormat。

BmJpuEncodedFrame

编码出的 jpeg 的信息。（未使用）

BmJpuEncodedFrame 结构体定义如下：

```
typedef struct{
    uint8_t *data;
    size_t data_size;
    void *acquired_handle;
    void *context;
    uint64_t pts, dts;
} BmJpuEncodedFrame;
```

- **data**

  编码出的 jpeg data。

- **data_size**

  编码出的 jpeg data 的长度。

- **acquired_handle**

  在 bm_jpu_jpeg_enc_encode 时，用户定义的 acquire_output_buffer 函数指针。

- **context**

  用户定义的 context，动态库内不会用到。

- **pts, dts**

  用户定义的时间戳。

BmJpuRawFrame

用于编码的原始的 yuv 图像的信息。

BmJpuRawFrame 结构体定义如下：

```
typedef struct{
    BmJpuFramebuffer *framebuffer;
    void *context;
    uint64_t pts, dts;
} BmJpuRawFrame;
```

- **BmJpuFramebuffer**

  用于编码的 yuv 帧，参考 BmJpuFramebuffer。

- **context**

  用户定义的 context，动态库内不会用到。

- **pts, dts**

  用户定义的时间戳。

BmJpuDecOpenParams

该结构体用于定义打开解码器的参数，换句话说是设置解码器的属性（接收图片的大小，解码缓存区大小等属性）。

BmJpuDecOpenParams 结构体定义如下：

```
typedef struct
{
    /* These are necessary with some formats which do not store the width
    * and height in the bitstream. If the format does store them, these
    * values can be set to zero. */
    unsigned int min_frame_width;
    unsigned int min_frame_height;
    unsigned int max_frame_width;
    unsigned int max_frame_height;

    BmJpuColorFormat color_format;
    /* If this is 1, then Cb and Cr are interleaved in one shared chroma
    * plane, otherwise they are separated in their own planes.
    * See the BmJpuColorFormat documentation for the consequences of this. */
    int chroma_interleave;

    /* 0: no scaling; n(1-3): scale by 2^n; */
    unsigned int scale_ratio;

    /* The DMA buffer size for bitstream */
    int bs_buffer_size;
```

(续下页)

(接上页)

```
#ifdef _WIN32
    uint8_t *buffer;
#else
    uint8_t *buffer __attribute__((deprecated));
#endif

    int device_index;

    int rotationEnable;
    BmJpuRotateAngle rotationAngle;
    int mirrorEnable;
    BmJpuMirrorDirection mirrorDirection;

    int roiEnable;
    int roiWidth;
    int roiHeight;
    int roiOffsetX;
    int roiOffsetY;

    bool framebuffer_recycle;
    size_t framebuffer_size;

    bool bitstream_from_user;
    bm_jpu_phys_addr_t bs_buffer_phys_addr;
    bool framebuffer_from_user;
    int framebuffer_num;
    bm_jpu_phys_addr_t *framebuffer_phys_addrs;

    int timeout;
    int timeout_count;
} BmJpuDecOpenParams;
```

- **min_frame_width**

  解码器支持的最小 width。

- **min_frame_height**

  解码器支持的最小 height。

- **max_frame_width**

  解码器支持的最大 width。

- **max_frame_height**

  解码器支持的最大 height。

- **color_format**

  输入图像的编码格式。(BM1688 弃用)

- **chroma_interleave**

  色度分量的存储方式的标识选项，可以是交错存储也可以是分开存储。(BM1688 弃用)

- **scale_ratio**

  用于指定视频解码时的缩放比例。它决定了图像在解码过程中是否进行大小调整（即缩放）。如果值为 0，则表示不进行任何缩放；如果值在 1 到 3 之间（包括 1 和 3），则表示将图像按照 2 的 n 次幂进行缩放，其中 n 为 scale_ratio 的值。

- **bs_buffer_size**

  表示码流的缓冲区的大小，这里记录了存储输入图片需要的字节大小。当用户外部申请 bitstream 的设备内存时，buffer size 需要 1024 对齐。

- **buffer**

  用于存储输入图片的具体内容。(BM1688 弃用)

- **device_index**

  解码设备 ID。

- **rotationEnable**

  是否做旋转操作的标识，0 表示不旋转，1 表示旋转。

- **rotationAngle**

  参考 BmJpuRotateAngle。

- **mirrorEnable**

  是否做镜像操作的标识，0 表示不镜像，1 表示镜像。

- **mirrorDirection**

  参考 BmJpuMirrorDirection。

- **roiEnable**

  是否设置 ROI（感兴趣区域）。

- **roiWidth**

  ROI 的宽度。

- **roiHeight**

  ROI 的高度。

- **roiOffsetX**

  ROI 相对图像左上角的水平偏移量。

- **roiOffsetY**

  ROI 相对图像左上角的垂直偏移量。

- **framebuffer_recycle**

  是否复用 framebuffer。(BM1688 弃用)

- **framebuffer_size**

  设置 framebuffer 的大小，单位：byte。

- **bitstream_from_user**

  jpeg data 的设备内存是否由用户设置。

- **bs_buffer_phys_addr**

  用户设置的 jpeg data 的设备内存地址。

- **framebuffer_from_user**

  解码得到的 yuv 图像的设备内存是否由用户设置。

- **framebuffer_num**

  framebuffer_num 个数。(BM1688 弃用)

- **framebuffer_phys_addrs**

  用户设置的解码得到的 yuv 图像的设备内存地址的指针。

- **timeout**

  用户设置解码超时时间（毫秒），不设置默认为 20000ms。

- **timeout_count**

  用户设置解码超时重试次数。(BM1688 弃用)

BmJpuDecoder

该结构体定义了 BMAPI 内部的解码器，包含解码器句柄、设备 ID、解码分配的 framebuffer 等信息。

BmJpuDecoder 结构体定义如下:

```
struct _BmJpuDecoder
{
    unsigned int device_index;
    DecHandle handle;
    bm_device_mem_t *bs_dma_buffer;
    uint8_t *bs_virt_addr;
    uint64_t bs_phys_addr;
    int chroma_interleave;
    int scale_ratio;
    unsigned int old_jpeg_width;
    unsigned int old_jpeg_height;
    BmJpuColorFormat old_jpeg_color_format;
    unsigned int num_framebuffers, num_used_framebuffers;
    FrameBuffer *internal_framebuffers;
    BmJpuFramebuffer *framebuffers;
    BmJpuDecFrameEntry *frame_entries;
```

(续下页)

(接上页)

```
    DecInitialInfo initial_info;
    int initial_info_available;
    DecOutputInfo dec_output_info;
    int available_decoded_frame_idx;
    bm_jpu_dec_new_initial_info_callback initial_info_callback;
    void *callback_user_data;
    int framebuffer_recycle;
    int channel_id;
    FramebufferList *fb_list_head;
    FramebufferList *fb_list_curr;
    int timeout;
};
typedef struct _BmJpuDecoder BmJpuDecoder;
```

- **device_index**

  解码设备 ID。

- **handle**

  解码器句柄。

- **bs_dma_buffer**

  用于存放待解码码流的一块设备内存，由 bmlib 分配。

- **bs_virt_addr**

  存放待解码码流的虚拟地址。

- **bs_phys_addr**

  存放待解码码流的物理地址。

- **chroma_interleave**

  表示色度分量是否交叉排列。

- **scale_ratio**

  表示图像缩放比例。该参数同时控制图像在水平和垂直方向上的缩放级别。

- **old_jpeg_width**

  表示上一帧解码图像的宽度。

- **old_jpeg_height**

  表示上一帧解码图像的高度。

- **old_jpeg_color_format**

  表示上一帧解码图像的编码格式。

- **num_framebuffers**

  表示已分配的解码 buffer 数量。

- **num_used_framebuffers**

  表示已使用的解码 buffer 数量。

- **internal_framebuffers**

  解码器内部注册到 JPU 的 framebuffer，用于和 JPU 交互。

- **framebuffers**

  解码器内部分配的全部 framebuffer。

- **frame_entries**

  解码器内部解码 buffer 控制结构，指定了当前帧的 PTS（显示时间）、DTS（解码时间）及使用状态。

- **initial_info**

  用于记录解码器的一些初始配置。

- **initial_info_available**

  用来标识解码器是否完成初始化。

- **dec_output_info**

  用于记录解码器的输出信息。

- **available_decoded_frame_idx**

  表示当前使用的解码帧序号。

- **initial_info_callback**

  一个用于初始化 framebuffer 的回调函数，当解码器输入码流变化时会重新分配 framebuffer。

- **callback_user_data**

  用于存储上述回调函数中传递的用户自定义数据。

- **framebuffer_recycle**

  用于表示 framebuffer 是否复用的标识。在复用的情况下，可用同一个解码器实例解码不同分辨率、格式的码流。

- **channel_id**

  表示解码器通道 ID。

- **fb_list_head**

  表示 framebuffer 链表中的头节点。解码器中使用链表存储当前使用中的 framebuffer，会在执行 flush 或 close 操作时全部释放，也可以使用 **bm_jpu_jpeg_dec_frame_finished** 接口释放某一帧。

- **fb_list_curr**

  表示 framebuffer 链表中的当前节点。用于将新生成的 framebuffer 加入链表。

- **timeout**

  用户设置的超时时间。

BmJpuJPEGDecoder

该结构体定义了对外提供的解码器，包含解码器句柄、设备 ID、解码分配的 framebuffer 等信息。

BmJpuJPEGDecoder 结构体定义如下：

```
typedef struct _BMJpuJPEGDecoder
{
    BmJpuDecoder *decoder;

    bm_jpu_phys_addr_t bitstream_buffer_addr;
    size_t bitstream_buffer_size;
    unsigned int bitstream_buffer_alignment;

    BmJpuDecInitialInfo initial_info;

    BmJpuFramebuffer *framebuffers;
    bm_jpu_phys_addr_t *framebuffer_addrs;
    unsigned int num_framebuffers;
    unsigned int num_extra_framebuffers;
    BmJpuFramebufferSizes calculated_sizes;

    BmJpuRawFrame raw_frame;
    int device_index;

    void *opaque;

    int rotationEnable;
    BmJpuRotateAngle rotationAngle;
    int mirrorEnable;
    BmJpuMirrorDirection mirrorDirection;

    bool framebuffer_recycle;
    bool bitstream_from_user;
    bool framebuffer_from_user;

}BmJpuJPEGDecoder;
```

- **decoder**

  BMAPI 内部定义的解码器。

- **bitstream_buffer_addr**

  用户外部申请，存放待解码码流的一块设备内存。（在 BmJpuDecOpenParams 里生效，BM1688 弃用）

- **bitstream_buffer_size**

表示上述码流的内存大小，单位：byte。(在 BmJpuDecOpenParams 里生效，BM1688 弃用)

- **bitstream_buffer_alignment**

  表示上述码流的对齐要求，单位：byte。(BM1688 弃用)

- **initial_info**

  用于记录解码器的一些初始配置。

- **framebuffers**

  用于记录解码器中 framebuffer 的地址及大小。(BM1688 弃用)

- **framebuffer_addrs**

  用户申请的用于存储解码器中 framebuffer 的设备内存。(在 BmJpuDecOpenParams 里生效，BM1688 弃用)

- **num_framebuffers**

  解码需要的 framebuffer 总帧数。

- **num_extra_framebuffers**

  解码需要的 framebuffer 额外帧数，通常为 0。

- **calculated_sizes**

  记录对齐后的 framebuffer 大小信息。

- **raw_frame**

  表示原始帧数据，用于存储图像的原始数据和时间戳。(BM1688 弃用)

- **device_index**

  表示解码设备 ID。

- **opaque**

  未定义数据，由用户决定如何使用。

- **rotationEnable**

  是否做旋转操作的标识，0 表示不旋转，1 表示旋转。

- **rotationAngle**

  参考 BmJpuRotateAngle。

- **mirrorEnable**

  是否做镜像操作的标识，0 表示不镜像，1 表示镜像。

- **mirrorDirection**

  参考 BmJpuMirrorDirection。

- **framebuffer_recycle**

  用于表示 framebuffer 是否复用的标识。在复用的情况下，可用同一个解码器实例解码不同分辨率、格式的码流。

- **bitstream_from_user**

  用户申请 jpeg data 的设备内存。(在 BmJpuDecOpenParams 里生效，BM1688 弃用)

- **framebuffer_from_user**

  用户申请解码得到的 yuv 图像的设备内存。(在 BmJpuDecOpenParams 里生效，BM1688 弃用)

BmJpuJPEGDecInfo

该结构体记录了解码后的 YUV 数据信息，用于用户获取解码数据。

BmJpuJPEGDecInfo 结构体定义如下：

```
typedef struct
{
    /* Width and height of JPU framebuffers are aligned to internal boundaries.
    * The frame consists of the actual image pixels and extra padding pixels.
    * aligned_frame_width / aligned_frame_height specify the full width/height
    * including the padding pixels, and actual_frame_width / actual_frame_height
    * specify the width/height without padding pixels. */
    unsigned int aligned_frame_width, aligned_frame_height;
    unsigned int actual_frame_width, actual_frame_height;

    /* Stride and size of the Y, Cr, and Cb planes. The Cr and Cb planes always
    * have the same stride and size. */
    unsigned int y_stride, cbcr_stride;
    unsigned int y_size, cbcr_size;

    /* Offset from the start of a framebuffer's memory, in bytes. Note that the
    * Cb and Cr offset values are *not* the same, unlike the stride and size ones. */
    unsigned int y_offset, cb_offset, cr_offset;

    /* Framebuffer containing the pixels of the decoded frame. */
    BmJpuFramebuffer *framebuffer;

    BmJpuImageFormat image_format;

    bool framebuffer_recycle;
    size_t framebuffer_size;
} BmJpuJPEGDecInfo;
```

- **aligned_frame_width**

  对齐之后该帧数据的宽度。

- **aligned_frame_height**

对齐之后该帧数据的高度。

- **actual_frame_width**

  原始图像的宽度。

- **actual_frame_height**

  原始图像的高度。

- **y_stride**

  Y 分量的步长。

- **cbcr_stride**

  Cb、Cr 分量的步长。

- **y_size**

  Y 分量的总数据量，单位：byte。

- **cbcr_size**

  Cb、Cr 分量的总数据量，单位：byte。

- **y_offset**

  Y 分量起始地址相对于 framebuffer 中物理地址的偏移量。

- **cb_offset**

  Cb 分量起始地址相对于 framebuffer 中物理地址的偏移量。

- **cr_offset**

  Cr 分量起始地址相对于 framebuffer 中物理地址的偏移量。

- **framebuffer**

  用来记录解码后的 YUV 数据相关信息。

- **image_format**

  图像格式。

- **framebuffer_recycle**

  用于表示 framebuffer 是否复用的标识。在复用的情况下，可用同一个解码器实例解码不同分辨率、格式的码流。

- **framebuffer_size**

  framebuffer_recycle 模式下需要申请的 framebuffer 内存大小，单位：byte。

BmJpuEncoder

该结构体定义了 BMAPI 内部的编码器，包含编码器句柄、设备 ID、输出码流等信息。

BmJpuEncoder 结构体定义如下:

```
typedef struct _BmJpuEncoder
{
    unsigned int device_index;
    EncHandle handle;

    bm_device_mem_t *bs_dma_buffer;
    uint8_t *bs_virt_addr;

    BmJpuColorFormat color_format;
    unsigned int frame_width, frame_height;

    BmJpuFramebuffer *framebuffers;

    int channel_id;
    int timeout;
} BmJpuEncoder;
```

- **device_index**

  编码设备 ID。

- **handle**

  编码器句柄。

- **bs_dma_buffer**

  用于存放输出码流的一块设备内存，由 bmlib 分配。

- **bs_virt_addr**

  存放输出码流的虚拟地址。

- **color_format**

  输出码流的编码格式。

- **frame_width**

  输出码流的宽度。

- **frame_height**

  输出码流的高度。

- **framebuffers**

  编码器内部使用的 framebuffer。

- **channel_id**

  内部 channel id。

- **timeout**

  编码器超时时间。

BmJpuJPEGEncoder

该结构体定义了对外提供的编码器，包含编码器句柄、设备 ID、输出码流等信息。

BmJpuJPEGEncoder 结构体定义如下:

```
typedef struct _BmJpuJPEGEncoder
{
    BmJpuEncoder *encoder;

    bm_jpu_phys_addr_t bitstream_buffer_addr;
    size_t bitstream_buffer_size;
    unsigned int bitstream_buffer_alignment;

    BmJpuEncInitialInfo initial_info;

    unsigned int frame_width, frame_height;

    BmJpuFramebufferSizes calculated_sizes;

    unsigned int quality_factor;

    BmJpuImageFormat image_format;

    int device_index;

    int rotationEnable;
    BmJpuRotateAngle rotationAngle;
    int mirrorEnable;
    BmJpuMirrorDirection mirrorDirection;

    bool bitstream_from_user;
} BmJpuJPEGEncoder;
```

- **encoder**

  BMAPI 内部定义的编码器。

- **bitstream_buffer_addr**

  用户外部申请的 jpeg data 的设备内存地址。

- **bitstream_buffer_size**

  用户外部申请的 jpeg data 的大小。

- **bitstream_buffer_alignment**

  表示上述码流的对齐要求，单位：byte。

- **initial_info**

  用于记录编码器的 framebuffer 初始配置。

- **frame_width**

  输入图像的宽度。

- **frame_height**

  输入图像的高度。

- **calculated_sizes**

  记录对齐后的 framebuffer 大小信息。

- **quality_factor**

  编码质量。

- **image_format**

  输入图像的编码格式。

- **device_index**

  表示编码设备 ID。

- **rotationEnable**

  是否做旋转操作的标识，0 表示不旋转，1 表示旋转。

- **rotationAngle**

  参考 BmJpuRotateAngle。

- **mirrorEnable**

  是否做镜像操作的标识，0 表示不镜像，1 表示镜像。

- **mirrorDirection**

  参考 BmJpuMirrorDirection。

- **bitstream_from_user**

  jpeg data 的设备内存是否由用户外部申请。(BM1688 弃用)

BmJpuJPEGEncParams

该结构体定义了编码配置参数及可配置的获取输出数据的接口函数。

BmJpuJPEGEncParams 结构体定义如下:

```
typedef struct
{
    /* Frame width and height of the input frame. These are the actual sizes;
    * they will be aligned internally if necessary. These sizes must not be
```

(续下页)

(接上页)

```
    * zero. */
    unsigned int frame_width, frame_height;

    /* Quality factor for JPEG encoding. 1 = best compression, 100 = best quality.
    * This is the exact same quality factor as used by libjpeg. */
    unsigned int quality_factor;

    /* Image format of the input frame. */
    BmJpuImageFormat image_format;

    /* Functions for acquiring and finishing output buffers. See the
    * typedef documentations in bmjpuapi.h for details about how
    * these functions should behave. */
    BmJpuEncAcquireOutputBuffer acquire_output_buffer;
    BmJpuEncFinishOutputBuffer finish_output_buffer;

    /* Function for directly passing the output data to the user
    * without copying it first.
    * Using this function will inhibit calls to acquire_output_buffer
    * and finish_output_buffer. */
    BmJpuWriteOutputData write_output_data;

    /* User supplied value that will be passed to the functions:
    * acquire_output_buffer, finish_output_buffer, write_output_data */
    void *output_buffer_context;

    int rotationEnable;
    BmJpuRotateAngle rotationAngle;
    int mirrorEnable;
    BmJpuMirrorDirection mirrorDirection;

    /* Identify the output data is in device memory or system memory */
    int bs_in_device;

    int timeout;
    int timeout_count;

    /* Optional: User supplied device memory for following encode,
     * will replace the bitstream buffer in encoder */
    bm_jpu_phys_addr_t bs_buffer_phys_addr;
    int bs_buffer_size;
} BmJpuJPEGEncParams;
```

- **frame_width**

  输出码流的宽度。

- **frame_height**

  输出码流的高度。

- **quality_factor**

编码质量，可选 1（压缩率最高）~100（图像质量最好）。

- **image_format**

  输出图片的编码格式。

- **acquire_output_buffer**

  用来获取编码码流输出 buffer 的回调函数。

- **finish_output_buffer**

  用来释放上述 buffer 的回调函数。

- **write_output_data**

  用来指定编码码流输出方式的回调函数，如：写入文件或写入指定的内存地址，与上述两个接口互斥。

- **output_buffer_context**

  用来保存输出数据的上下文。

- **rotationEnable**

  是否做旋转操作的标识，0 表示不旋转，1 表示旋转。

- **rotationAngle**

  参考 BmJpuRotateAngle。

- **mirrorEnable**

  是否做镜像操作的标识，0 表示不镜像，1 表示镜像。

- **mirrorDirection**

  参考 BmJpuMirrorDirection。

- **bs_in_device**

  编码得到的 jpeg data 输出到设备内存。(BM1688 弃用)

- **timeout**

  用户设置编码器超时时间（ms），默认 20000ms。

- **timeout_count**

  用户设置超时重试次数。(BM1688 弃用)

- **bs_buffer_phys_addr**

  用户设置 jpeg data 的设备内存地址。

- **bs_buffer_size**

  用户设置 jpeg data 的 size。

2.5.3 JPEG 接口说明

bm_jpu_dec_load

该接口根据传入的 ID 打开指定的解码设备节点，可以通过 bmlib 管理内存分配。

接口形式：

```
BmJpuDecReturnCodes bm_jpu_dec_load(int device_index);
```

参数说明：

- **device_index**

  解码设备 ID。

返回值说明：

- BM_JPU_DEC_RETURN_CODE_OK: 成功
- 其他: 失败

bm_jpu_dec_unload

该接口释放指定的解码设备节点。

接口形式：

```
BmJpuDecReturnCodes bm_jpu_dec_unload(int device_index);
```

参数说明：

- **device_index**

  解码设备 ID。

返回值说明：

- BM_JPU_DEC_RETURN_CODE_OK: 成功
- 其他: 失败

bm_jpu_jpeg_dec_open

该接口打开一个解码器，根据传入的参数配置解码通道。

接口形式：

```
BmJpuDecReturnCodes bm_jpu_jpeg_dec_open(BmJpuJPEGDecoder **jpeg_
↪decoder,
                                         BmJpuDecOpenParams *open_params,
                                         unsigned int num_extra_framebuffers)
```

参数说明：

- **jpeg_decoder**

  指向解码器的二级指针，在接口内部完成初始化。

- **open_params**

  打开解码器时的配置参数。

- **num_extra_framebuffers**

  需要额外申请的 framebuffer 个数。（BM1688 弃用）

返回值说明：

- BM_JPU_DEC_RETURN_CODE_OK：成功
- 其他：失败

bm_jpu_jpeg_dec_close

该接口用来关闭解码器，释放资源。

接口形式：

```
BmJpuDecReturnCodes bm_jpu_jpeg_dec_close(BmJpuJPEGDecoder *jpeg_
↪decoder);
```

参数说明：

- **jpeg_decoder**

  一个已经打开的解码器。

返回值说明：

- BM_JPU_DEC_RETURN_CODE_OK：成功
- 其他：失败

bm_jpu_jpeg_dec_decode

该接口执行解码操作。

接口形式：

```
BmJpuDecReturnCodes bm_jpu_jpeg_dec_decode(
    BmJpuJPEGDecoder *jpeg_decoder,
    uint8_t const *jpeg_data,
    size_t const jpeg_data_size
    int timeout,
    int timeout_count);
```

参数说明：

- **jpeg_decoder**

  一个已经打开的解码器。

- **jpeg_data**

  待解码的图像数据。

- **jpeg_data_size**

  待解码的图像数据大小。

- **timeout**

  解码器超时时间。

- **timeout_count**

  解码器超时重试次数。(BM1688 弃用)

返回值说明:

- BM_JPU_DEC_RETURN_CODE_OK: 成功
- 其他: 失败

bm_jpu_jpeg_dec_get_info

该接口从解码器获取解码信息。

接口形式:

```
void bm_jpu_jpeg_dec_get_info(
    BmJpuJPEGDecoder *jpeg_decoder,
    BmJpuJPEGDecInfo *info);
```

参数说明:

- **jpeg_decoder**

  一个已经打开的解码器。

- **info**

  用于存储解码信息的数据结构。

返回值说明:

- 无

bm_jpu_jpeg_dec_frame_finished

该接口释放一帧解码完成的 framebuffer。

接口形式：

```
BmJpuDecReturnCodes bm_jpu_jpeg_dec_frame_finished(
    BmJpuJPEGDecoder *jpeg_decoder,
    BmJpuFramebuffer *framebuffer);
```

参数说明：

- **jpeg_decoder**

  一个已经打开的解码器。

- **framebuffer**

  一帧解码完成的 framebuffer。

返回值说明：

- BM_JPU_DEC_RETURN_CODE_OK：成功
- 其他：失败

bm_jpu_jpeg_dec_flush

该接口用来刷新解码器，释放所有解码完成的 framebuffer。

接口形式：

```
BmJpuDecReturnCodes bm_jpu_jpeg_dec_flush(BmJpuJPEGDecoder *jpeg_
↪decoder);
```

参数说明：

- **jpeg_decoder**

  一个已经打开的解码器。

返回值说明：

- BM_JPU_DEC_RETURN_CODE_OK：成功
- 其他：失败

bm_jpu_enc_load

该接口根据传入的 ID 打开指定的编码设备节点，可以通过 bmlib 管理内存分配。

接口形式：

```
BmJpuEncReturnCodes bm_jpu_enc_load(int device_index);
```

参数说明：

- **device_index**

  编码设备 ID。

返回值说明：

- BM_JPU_ENC_RETURN_CODE_OK: 成功
- 其他: 失败

bm_jpu_enc_unload

该接口释放指定的编码设备节点。

接口形式：

```
BmJpuEncReturnCodes bm_jpu_enc_unload(int device_index);
```

参数说明：

- **device_index**

  编码设备 ID。

返回值说明：

- BM_JPU_ENC_RETURN_CODE_OK: 成功
- 其他: 失败

bm_jpu_jpeg_enc_open

该接口打开一个编码器，并申请指定大小的 bitstream buffer。

接口形式：

```
BmJpuEncReturnCodes bm_jpu_jpeg_enc_open(
        BmJpuJPEGEncoder **jpeg_encoder,
        bm_jpu_phys_addr_t bs_buffer_phys_addr,
        int bs_buffer_size,
        int device_index
        );
```

参数说明：

- **jpeg_encoder**

  指向编码器的二级指针，在接口内部完成初始化。

- **bs_buffer_phys_addr**

  用户外部申请的 bitstream buffer 设备内存地址。

- **bs_buffer_size**

  bistream buffer 的大小，输入为 0 则默认申请 5MB。

- **device_index**

  编码设备 ID。

返回值说明:

- BM_JPU_ENC_RETURN_CODE_OK: 成功
- 其他: 失败

bm_jpu_jpeg_enc_close

该接口用来关闭编码器，释放资源。

接口形式:

```
BmJpuEncReturnCodes bm_jpu_jpeg_enc_close(BmJpuJPEGEncoder *jpeg_
↪encoder);
```

参数说明:

- **jpeg_encoder**

  一个已经打开的编码器

返回值说明:

- BM_JPU_ENC_RETURN_CODE_OK: 成功
- 其他: 失败

bm_jpu_jpeg_enc_encode

该接口执行编码操作。

接口形式:

```
BmJpuEncReturnCodes bm_jpu_jpeg_enc_encode(
        BmJpuJPEGEncoder *jpeg_encoder,
        BmJpuFramebuffer const *framebuffer,
        BmJpuJPEGEncParams const *params,
        void **acquired_handle,
```

(续下页)

(接上页)

```
    size_t *output_buffer_size
    );
```

参数说明：

- **jpeg_encoder**

  一个已经打开的编码器。

- **framebuffer**

  输入的 frame 数据。

- **params**

  编码相关参数。

- **acquired_handle**

  用于存放编码数据的位置，由用户指定。如果为 NULL，则通过 write_output_data 接口输出。

- **output_buffer_size**

  输出数据 buffer 的大小，单位：byte。

返回值说明：

- BM_JPU_ENC_RETURN_CODE_OK: 成功
- 其他: 失败

acquire_output_buffer

该接口用于获取 buffer 接收编码数据。

接口形式：

```
void* acquire_output_buffer(void *context, unsigned int size, void **acquired_
↪handle);
```

参数说明：

- **context**

  输出 buffer 上下文。

- **size**

  输出 buffer 大小，单位：byte。

- **acquired_handle**

  输出 buffer 的起始地址。

finish_output_buffer

该接口用于释放上述接口获取的 buffer。

接口形式：

```
void finish_output_buffer(void *context, void *acquired_handle);
```

参数说明：

- **context**

  输出 buffer 上下文。

- **acquired_handle**

  输出 buffer 的起始地址。

2.5.4 JPEG 测试用例说明

bmjpegdec

代码请参考 example/jpeg_dec_test.c

参数说明

```
usage:  bmjpegdec [option]
option:
        -i input_file
        -o output_file
        -n loop_num
        -c crop function(0:disable  1:enable crop)
        -g rotate (default 0) [rotate mode[1:0]  0:No rotate  1:90  2:180  3:270] [rotator
↪mode[2]:vertical flip] [rotator mode[3]:horizontal flip]
        -s scale (default 0) -> 0 to 3
        -r roi_x,roi_y,roi_w,roi_h
```

单路解码

```
bmjpegdec -i JPEG_1920x1088_yuv420_planar.jpg -o out_1920x1088_yuv420_
↪planar.yuv -n 1
```

单路循环解码

```
bmjpegdec -i JPEG_1920x1088_yuv420_planar.jpg -o out_1920x1088_yuv420_
↪planar.yuv -n 10
```

bmjpegenc

代码请参考 example/jpeg_enc_test.c

参数说明

```
usage: bmjpegenc [option]
option:
        -f pixel format: 0.YUV420(default); 1.YUV422; 2.YUV444; 3.YUV400. (optional)
        -w actual width
        -h actual height
        -y luma stride (optional)
        -c chroma stride (optional)
        -v aligned height (optional)
        -i input file
        -o output file
        -n loop num
        -g rotate (default 0) [rotate mode[1:0] 0:No rotate 1:90 2:180 3:270] [rotator F
↪mode[2]:vertical flip] [rotator mode[3]:horizontal flip]
```

单路编码

```
bmjpegenc -f 0 -w 1920 -h 1088 -i JPEG_1920x1088_yuv420_planar.yuv -o JPEG_
↪1920x1088_yuv420_planar.jpg -n 1
```

单路循环编码

```
bmjpegenc -f 0 -w 1920 -h 1088 -i JPEG_1920x1088_yuv420_planar.yuv -o JPEG_
↪1920x1088_yuv420_planar.jpg -n 10000
```

旋转

```
bmjpegenc -f 0 -w 1920 -h 1088 -i JPEG_1920x1088_yuv420_planar.yuv -o JPEG_
↪1920x1088_yuv420_planar.jpg -n 1 -g 1
```

镜像

```
bmjpegenc -f 0 -w 1920 -h 1088 -i JPEG_1920x1088_yuv420_planar.yuv -o JPEG_
↪1920x1088_yuv420_planar.jpg -n 1 -g 4
```

bmjpegmulti

代码请参考 example/jpeg_multi_test.c

32 路解码

```
首先编写配置文件 multi.
↪lst，内容如下：(第一行表示路数和每路的循环次数，之后每一行表示每一路的配置)

 32 100
```

(续下页)

(接上页)

```
 JPEG_352x288_yuv420_planar.jpg 32 1 0 0
 JPEG_352x288_yuv420_planar.jpg 32 1 0 0
 ...(重复到32行)

然后执行 bmjpegmulti -f multi.lst
选项输入 30
```

32 路编码

```
首先编写配置文件 enc.cfg，内容如下:

 YUV_SRC_IMG JPEG_352x288_yuv420_planar.yuv
 FRAME_FORMAT 0
 PICTURE_WIDTH 352
 PICTURE_HEIGHT 288
 IMG_FORMAT 0

然后编写配置文件 multi.
↪lst，内容如下:（第一行表示路数和每路的循环次数，之后每一行表示每一路的配置）

 32 1000000
 enc.cfg 32 0 0 0
 enc.cfg 32 0 0 0
 ... (重复到32行)

然后执行 bmjpegmulti -f multi.lst
选项输入 30
```

单路 YUV400 编码

```
bmjpegmulti -t 1 -i JPEG_1920x1088_yuv400_planar.yuv -o OUT400.jpg -w 1920 -h
↪1088 -s 4 -f 0 -n 4
```

单路 YUV420P 编码

```
bmjpegmulti -t 1 -i JPEG_1920x1088_yuv420_planar.yuv -o OUT420_planar.jpg -w
↪1920 -h 1088 -s 0 -f 0 -n 4
```

单路 YUV422P 编码

```
bmjpegmulti -t 1 -i JPEG_1920x1088_yuv422_planar.yuv -o OUT422_planar.jpg -w
↪1920 -h 1088 -s 1 -f 0 -n 4
```

2.6 SOPHGO Video Decoder 使用指南

2.6.1 简介

概述

VDEC 模块提供驱动视频解码硬件工作的对应接口，实现视频解码功能。

BM1688 VDEC 模块支持 H.264 和 H.265 解码，支持对 16 路 1080P 视频同时以 30fps 的性能进行解码。

定义及缩写

缩写	含义
VPU	视频处理单元
VDEC	视频解码器
core	核心
BitStream	输入码流数据
Frame	帧
Buffer	缓冲区
channel	通道

2.6.2 VDEC 数据类型介绍

BMVidDecRetStatus

定义了解码器错误返回值类型。

枚举定义

```
typedef enum
{
    BM_ERR_VDEC_INVALID_CHNID = -27,
    BM_ERR_VDEC_ILLEGAL_PARAM,
    BM_ERR_VDEC_EXIST,
    BM_ERR_VDEC_UNEXIST,
    BM_ERR_VDEC_NULL_PTR,
    BM_ERR_VDEC_NOT_CONFIG,
    BM_ERR_VDEC_NOT_SUPPORT,
    BM_ERR_VDEC_NOT_PERM,
    BM_ERR_VDEC_INVALID_PIPEID,
    BM_ERR_VDEC_INVALID_GRPID,
    BM_ERR_VDEC_NOMEM,
    BM_ERR_VDEC_NOBUF,
    BM_ERR_VDEC_BUF_EMPTY,
```

(续下页)

(接上页)

```
  BM_ERR_VDEC_BUF_FULL,
  BM_ERR_VDEC_SYS_NOTREADY,
  BM_ERR_VDEC_BADADDR,
  BM_ERR_VDEC_BUSY,
  BM_ERR_VDEC_SIZE_NOT_ENOUGH,
  BM_ERR_VDEC_INVALID_VB,
  BM_ERR_VDEC_ERR_INIT,
  BM_ERR_VDEC_ERR_INVALID_RET,
  BM_ERR_VDEC_ERR_SEQ_OPER,
  BM_ERR_VDEC_ERR_VDEC_MUTEX,
  BM_ERR_VDEC_ERR_SEND_FAILED,
  BM_ERR_VDEC_ERR_GET_FAILED,
  BM_ERR_VDEC_BUTT,
  BM_ERR_VDEC_FAILURE
}BMVidDecRetStatus;
```

参数含义

- **BM_ERR_VDEC_INVALID_CHNID**: 无效的 channel id。
- **BM_ERR_VDEC_ILLEGAL_PARAM**: 非法参数。
- **BM_ERR_VDEC_NULL_PTR**: 空指针。
- **BM_ERR_VDEC_NOBUF**: 无效的缓冲区。
- **BM_ERR_VDEC_BUF_FULL**: 缓冲区空。
- **BM_ERR_VDEC_BUF_FULL**: 缓冲区满。
- **BM_ERR_VDEC_BUSY**: 解码器忙。
- **BM_ERR_VDEC_FAILURE**: 解码器一般错误。

BmVpuDecStreamFormat

码流格式。

枚举定义

```
typedef enum{
  BMDEC_AVC     = 0,
  BMDEC_HEVC    = 12,
}BmVpuDecStreamFormat;
```

参数含义

- **BMDEC_AVC**: AVC 码流。
- **BMDEC_HEVC**: HEVC 码流。

BmVpuDecSkipMode

设置跳帧模式

枚举定义

```
typedef enum {
    BMDEC_FRAME_SKIP_MODE          = 0,
    BMDEC_SKIP_NON_REF_NON_I    = 1,
    BMDEC_SKIP_NON_I            = 2,
}BmVpuDecSkipMode;
```

参数含义

- **BMDEC_FRAME_SKIP_MODE**: 禁用跳帧模式。
- **BMDEC_SKIP_NON_REF_NON_I**: 开启跳帧模式，跳过除参考帧和 I 帧外的视频帧
- **BMDEC_SKIP_NON_I**: 开启跳帧模式，跳过除 I 帧外的视频帧

BmVpuDecDMABuffer

存储 VPU 缓冲区的信息

结构体定义

```
typedef struct {
    unsigned int  size;
    u64           phys_addr;
    u64           virt_addr;
} BmVpuDecDMABuffer;
```

参数含义

- **size**: 缓冲区的大小。
- **phys_addr**: 缓冲区的物理地址。
- **phys_addr**: 缓冲区的虚拟地址。

BmVpuDecOutputMapType

设置输出数据的类型。

枚举定义

```
typedef enum {
    BMDEC_OUTPUT_UNMAP,
    BMDEC_OUTPUT_TILED = 100,
    BMDEC_OUTPUT_COMPRESSED,
} BmVpuDecOutputMapType;
```

参数含义

- **BMDEC_OUTPUT_UNMAP**: 输出 yuv 数据。
- **BMDEC_OUTPUT_COMPRESSED**: 输出压缩模式数据。

BmVpuDecBitStreamMode

设置 VPU 解码方式

枚举定义

```
typedef enum {
    BMDEC_BS_MODE_INTERRUPT = 0,
    BMDEC_BS_MODE_RESERVED,
    BMDEC_BS_MODE_PIC_END = 2,
} BmVpuDecBitStreamMode;
```

参数含义

- **BMDEC_BS_MODE_INTERRUPT**: 采用流模式解码，当输入缓冲区填满后送入解码器。
- **BMDEC_BS_MODE_PIC_END**: 采用帧模式解码，获取到一帧数据就送入解码器，需要提前解析码流。

BmVpuDecPixFormat

设置输出数据的格式

枚举定义

```
typedef enum
{
    BM_VPU_DEC_PIX_FORMAT_YUV420P = 0,
    BM_VPU_DEC_PIX_FORMAT_YUV422P = 1,
    BM_VPU_DEC_PIX_FORMAT_YUV444P = 3,
    BM_VPU_DEC_PIX_FORMAT_YUV400  = 4,
    BM_VPU_DEC_PIX_FORMAT_NV12    = 5,
    BM_VPU_DEC_PIX_FORMAT_NV21    = 6,
    BM_VPU_DEC_PIX_FORMAT_NV16    = 7,
    BM_VPU_DEC_PIX_FORMAT_NV24    = 8,
    BM_VPU_DEC_PIX_FORMAT_COMPRESSED = 9,
    BM_VPU_DEC_PIX_FORMAT_COMPRESSED_10BITS = 10,
} BmVpuDecPixFormat;
```

参数含义

- **BM_VPU_DEC_PIX_FORMAT_YUV420P**: 输出 YUV420P 数据
- **BM_VPU_DEC_PIX_FORMAT_YUV422P**: 输出 YUV422P 数据，BM1688 不支持
- **BM_VPU_DEC_PIX_FORMAT_YUV444P**: 输出 YUV444P 数据，BM1688 不支持

- **BM_VPU_DEC_PIX_FORMAT_YUV400**: 输出 YUV400 数据，BM1688 不支持
- **BM_VPU_DEC_PIX_FORMAT_NV12**: 输出 NV12 数据
- **BM_VPU_DEC_PIX_FORMAT_NV21**: 输出 NV21 数据，BM1688 不支持
- **BM_VPU_DEC_PIX_FORMAT_NV16**: 输出 NV16 数据，BM1688 不支持
- **BM_VPU_DEC_PIX_FORMAT_NV24**: 输出 NV24 数据，BM1688 不支持
- **BM_VPU_DEC_PIX_FORMAT_COMPRESSED**: 输出压缩格式数据
- **BM_VPU_DEC_PIX_FORMAT_COMPRESSED_10BITS**: 输出 10bits 压缩格式数据，BM1688 不支持

BMDecStatus

用于指示解码器的状态。

枚举定义

```
typedef enum {
    BMDEC_UNCREATE,
    BMDEC_UNLOADING,
    BMDEC_UNINIT,
    BMDEC_INITING,
    BMDEC_WRONG_RESOLUTION,
    BMDEC_FRAMEBUFFER_NOTENOUGH,
    BMDEC_DECODING,
    BMDEC_FRAME_BUF_FULL,
    BMDEC_ENDOF,
    BMDEC_STOP,
    BMDEC_HUNG,
    BMDEC_CLOSE,
    BMDEC_CLOSED,
} BMDecStatus;
```

参数含义

目前只有以下几个状态有效：

- **BMDEC_UNINIT**: 解码器未进行初始化
- **BMDEC_INITING**: 解码器正在进行初始化
- **BMDEC_WRONG_RESOLUTION**: 设置的分辨率不匹配
- **BMDEC_FRAMEBUFFER_NOTENOUGH**: 分配的 Frame Buffer 不足
- **BMDEC_DECODING**: 解码器正在解码
- **BMDEC_STOP**: 解码器解码结束

BMVidDecParam

BMVidDecParam 用于设置解码器的初始化参数，在调用接口 bmvpu_dec_create 前需要创建 BMVidDecParam 对象，并对其进行初始化。

结构体定义

```
typedef struct {
    BmVpuDecStreamFormat        streamFormat;
    BmVpuDecOutputMapType       wtlFormat;
    BmVpuDecSkipMode            skip_mode;
    BmVpuDecBitStreamMode       bsMode;
    int                         enableCrop;
    BmVpuDecPixFormat           pixel_format;
    int                         secondaryAXI;
    int                         mp4class;
    int                         frameDelay;
    int                         pcie_board_id;
    int                         pcie_no_copyback;
    int                         enable_cache;
    int                         perf;
    int                         core_idx;
    int                         cmd_queue_depth;
    int                         decode_order;
    int                         picWidth;
    int                         picHeight;
    int                         timeout;
    int                         timeout_count;
    int                         extraFrameBufferNum;
    int                         min_framebuf_cnt;
    int                         framebuf_delay;
    int                         streamBufferSize;
    BmVpuDecDMABuffer*          bitstream_buffer;
    BmVpuDecDMABuffer*          frame_buffer;
    BmVpuDecDMABuffer*          Ytable_buffer;
    BmVpuDecDMABuffer*          Ctable_buffer;
    int                         reserved[13];
} BMVidDecParam;
```

参数含义

- **streamFormat:**
  设置输入码流类型，0 为 H.264(AVC)，12 为 H.265(HEVC)。
- **wtlFormat:**
  设置输出数据类型。

  设置为 0，则根据 yuv 类型输出对应的 yuv 数据；

  设置为 101，则输出压缩的 fbc 数据。
- **skip_mode:**
  设置跳帧模式。

- **bsMode:**
  设置解码器工作方式。
- **pixel_format:**
  输出图像格式。
- **secondaryAXI:**
  BM1688 中不在需要设置此参数，

  SDK 中会根据码流类型，自动选择是否开启 secondary AXI 功能。
- **decode_order**
  设置解码器出帧顺序。0，以显示序出帧；1，以解码序出帧。
- **timeout:**
  解码超时时间。
- **timeout_count:**
  解码超时重试次数。
- **extraFrameBufferNum:**
  设置 Frame Buffer 的数量。
- **min_framebuf_cnt:**
  输入码流所需要的最小的 Frame Buffer 的数量。
- **framebuf_delay:**
  解码延迟出帧所需要的 Frame Buffer 的数量。
- **streamBufferSize:**
  设置输入码流的缓冲区大小。

  若设置为 0，则默认缓冲区大小为 0x700000。
- **bitstream_buffer:**
  输入码流缓冲区信息。
- **frame_buffer:**
  Frame Buffer 缓冲区信息。
- **Ytable_buffer:**
  压缩模式 Y table 缓冲区信息。
- **Ctable_buffer:**
  压缩模式 C table 缓冲区信息。

BMVidStream

通过 BMVidStream 对象，向解码器传递码流数据。

结构体定义

```
typedef struct BMVidStream {
    unsigned char*  buf;
    unsigned int    length;
    unsigned char*  header_buf;
    unsigned int    header_size;
    unsigned char*  extradata;
    unsigned int    extradata_size;
    unsigned char   end_of_stream;
    u64             pts;
    u64             dts;
} BMVidStream;
```

参数含义

- **buf、length:**
  BitStream Buffer 的地址和大小。
- **end_of_stream:**
  码流结束标志。当码流读取完成后，需要将这个标志位置 1。
- **pts、dts:**
  时间戳
- BM1688 中，不再接受 header 和 extradata 的数据。为了和前序产品保持一致，以上变量仍然存在。

BMVidFrame

BMVidFrame 用于接收解码器输出的帧信息。

结构体定义

```
typedef struct BMVidFrame {
    BmVpuDecPicType     picType;
    unsigned char*      buf[8];
    int                 stride[8];
    unsigned int        width;
    unsigned int        height;
    BmVpuDecLaceFrame   interlacedFrame;
    int                 lumaBitDepth;
    int                 chromaBitDepth;
    BmVpuDecPixFormat   pixel_format;
    int                 endian;
    int                 sequenceNo;
    int                 frameIdx;
    u64                 pts;
```

(续下页)

(接上页)

```
    u64             dts;
    int             colorPrimaries;
    int             colorTransferCharacteristic;
    int             colorSpace;
    int             colorRange;
    int             chromaLocation;
    int             size;
    unsigned int      coded_width;
    unsigned int      coded_height;
} BMVidFrame;
```

参数含义

- **picType:**
  表示当前 Frame 的类型，对应关系如下：

表 2.2: picType 对应关系

picType	类型
0	I picture
1	P picture
2	B picture
4	IDR picture

- **buf:**
  存放输出数据的地址。各通道对应的含义如下表：

表 2.3: buf 对应关系

channel	YUV420P	NV12/NV21	FBC
0	Y 分量虚拟地址	Y 分量虚拟地址	/
1	Cb 分量虚拟地址	CbCr 分量虚拟地址	/
2	Cr 分量虚拟地址	/	/
3	/	/	/
4	Y 分量物理地址	Y 分量物理地址	Y 分量物理地址
5	Cb 分量物理地址	CbCr 分量物理地址	Cb 分量物理地址
6	Cr 分量物理地址	/	Y table 分量物理地址
7	/	/	Cb table 分量物理地址

- **stride:**
  和 buf 对应，存放对应通道的宽度，该宽度是进行对齐后的宽度。

  对于 FBC 数据，stride 存放的数据稍有不同。

  channel 0 和 4，存放 Y 分量的宽度；

  channel 1 和 5，存放 Cb 分量的宽度；

  channel 2 和 6，存放 Y table 的长度；

channel 3 和 7，存放 Cb table 的长度。

- **width:**
  存放 Frame 的宽度。
- **height:**
  存放 Frame 的高度。
- **frameFormat:**
  存放图像的格式。
- **interlacedFrame:**
  图像扫描方式。
- **lumaBitDepth:**
  亮度数据的深度。
- **chromaBitDepth:**
  色度数据的深度。
- **cbcrInterleave:**
  表示色度分量的存储方式。

  cbcrInterleave 为 0，Cb 和 Cr 分量存储在不同的内存空间中；

  cbcrInterleave 为 1，Cb 和 Cr 分量存储在同一个内存空间中。
- **nv21:**
  表示 YUV 数据的存储格式，仅当 cbcrInterleave=1 时有效。

  nv21=0，以 Nv12 格式存储；

  nv21=1，以 NV21 格式存储。
- **endian:**
  表示帧缓冲区的段序。

  endian=0，以小端模式存储；

  endian=1，以大端模式存储；

  endian=2，以 32 位小端模式存储；

  endian=3，以 32 位大端模式存储。
- **sequenceNo:**
  表示码流序列的状态。当码流序列改变时，sequenceNo 的值会进行累加。
- **frameIdx:**
  图像帧缓冲区的索引。用于表示该帧缓冲区在解码器中位置。
- **pts:**
  显示时间戳。
- **dts:**
  解码时间戳。

- **size:**
  帧缓冲区的大小。
- **coded_width:**
  用于显示的图片宽度。
- **coded_height:**
  用于显示的图片高度。
- **compressed_mode:**
  表示解码器输出数据的格式。

CropRect

CropRect 用于保存图像的剪裁信息。

结构体定义

```
typedef struct {
    unsigned int left;
    unsigned int top;
    unsigned int right;
    unsigned int bottom;
} CropRect;
```

参数含义

- **left:**
  剪裁框左上角相对于像素原点的水平偏移量。
- **top:**
  剪裁框左上角相对于像素原点的垂直偏移量。
- **right:**
  剪裁框右下角相对于像素原点的水平偏移量。
- **bottom:**
  剪裁框右下角相对于像素原点的垂直偏移量。

BMVidStreamInfo

BMVidStreamInfo 用于接收输入码流的信息。

结构体定义

```
typedef struct BMVidStreamInfo {
    int         picWidth;
    int         picHeight;
    int         fRateNumerator;
    int         fRateDenominator;
    CropRect    picCropRect;
```

(续下页)

(接上页)

```
    int             mp4DataPartitionEnable;
    int             mp4ReversibleVlcEnable;
    int             mp4ShortVideoHeader;
    int             h263AnnexJEnable;
    int             minFrameBufferCount;
    int             frameBufDelay;
    int             normalSliceSize;
    int             worstSliceSize;
    int             maxSubLayers;
    int             profile;
    int             level;
    int             tier;
    int             interlace;
    int             constraint_set_flag[4];
    int             direct8x8Flag;
    int             vc1Psf;
    int             isExtSAR;
    int             maxNumRefFrmFlag;
    int             maxNumRefFrm;
    int             aspectRateInfo;
    int             bitRate;
    int             mp2LowDelay;
    int             mp2DispVerSize;
    int             mp2DispHorSize;
    unsigned int    userDataHeader;
    int             userDataNum;
    int             userDataSize;
    int             userDataBufFull;
    int             chromaFormatIDC;
    int             lumaBitdepth;
    int             chromaBitdepth;
    int             seqInitErrReason;
    int             warnInfo;
    unsigned int    sequenceNo;
} BMVidStreamInfo;
```

参数含义

- **picWidth:**
  图片宽度。
- **picHeight:**
  图片高度。
- **fRateNumerator:**
  帧率分数的分子。
- **fRateDenominator:**
  帧率分数的分母。
- **picCropRect:**
  剪裁信息，详细信息可以参考 CropRect。

- **minFrameBufferCount:**
  解码器所需要的帧缓冲区的最小数量。
- **frameBufDelay:**
  用于缓冲解码图像重排序的最大显示帧缓冲延迟。
- **profile:**
  H.264/H.265 的配置文件索引。
- **level:**
  H.264/H.265 的级别索引。
- **interlace:**
  interlace=1 时，码流被解码为逐行帧或隔行帧；否则解码为逐行帧。
- **bitRate:**
  BitStream 写入时的比特率。
- **lumaBitdepth:**
  亮度数据的深度。
- **chromaBitdepth:**
  色度数据的深度。
- **sequenceNo:**
  表示码流序列的状态。当码流序列改变时，sequenceNo 的值会进行累加。

2.6.3 VDEC API 介绍

bmvpu_dec_create

bmvpu_dec_create 用来创建一个解码器通道。

接口形式

```
BMVidDecRetStatus bmvpu_dec_create (
  BMVidCodHandle* pVidCodHandle,
  BMVidDecParam decParam );
```

参数说明

- **pVidCodHandle:**
  输出参数。解码器句柄，当解码器创建成功后将会返回一个句柄，用句柄可以对解码器进行后续的操作。
- **decParam:**
  输入参数。解码器初始化参数。

返回值

当解码器创建成功，会返回 0，否则将返回对应的错误码。

bmvpu_dec_decode

利用 bmvpu_dec_decode 将 BitStream 送入 VDEC 进行解码。BM1688 支持两种解码模式，分别是 INTERRUPT 和 PIC_END 模式。

> **INTERRUPT 模式**是按照预先设置的 BitStream Buffer 大小，每次送入固定大小的码流数据，并不存在帧的概念。
>
> **PIC_END 模式**是根据 H.264/H.265 协议，预先解析出一帧数据的位置，每次只向解码器传输一帧数据对应的 BitStream，因此在创建解码器时，需要合理考虑 streamBufferSize 的值。

接口形式

```
BMVidDecRetStatus bmvpu_dec_decode (
    BMVidCodHandle vidCodHandle,
    BMVidStream vidStream );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。
- **vidStream:**
  输入参数。BitStream 的地址和大小。

返回值

当码流成功送入解码器，会返回 0，否则将返回对应的错误码。返回错误并不代表解码器工作异常。若返回 BM_ERR_VDEC_BUF_FULL，需要检查解码器状态，并再次尝试送 BitStream。

bmvpu_dec_get_output

bmvpu_dec_get_output 这个接口用于来获取解码器的输出数据。

接口形式

```
BMVidDecRetStatus bmvpu_dec_get_output (
    BMVidCodHandle vidCodHandle,
    BMVidFrame *bmFrame );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。
- **bmFrame:**
  输出参数。用于接收解码器的输出数据。

返回值

当成功获得到解码数据，会返回 0，否则返回对应的错误码。返回错误码并不代表解码器工作异常，以下情况都有可能造成返回错误码：

1. 没有输入 BitStream 数据；
2. 解码尚未完成，输出数据未准备好；
3. 解码器关闭；
4. 解码器异常。

bmvpu_dec_clear_output

当 Frame Buffer 不再使用时，需要调用 bmvpu_dec_clear_output 对其进行释放。否则解码器可能因为没有足够的 Frame Buffer 而工作异常。

接口形式

```
BMVidDecRetStatus bmvpu_dec_clear_output (
    BMVidCodHandle vidCodHandle,
    BMVidFrame *frame );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。
- **frame:**
  输入参数。需要释放的 bmFrame 对象的地址。

返回值

当 frame 释放成功，会返回 0，否则返回对应的错误码。

bmvpu_dec_flush

当 BitStream 全部送入解码器后，并不代表解码已经完成，还需要等待解码器将全部的 Frame 输出。用 bmvpu_dec_flush 来刷出剩余的 Frame。

接口形式

```
BMVidDecRetStatus bmvpu_dec_flush (
    BMVidCodHandle vidCodHandle );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。

返回值

当 Frame 全部取出后，将会返回 0。

bmvpu_dec_get_all_frame_in_buffer

同样用于在码流传送完毕后，获取解码器中的剩余的 Frame 数据。和 bmvpu_dec_flush 不同之处在于，bmvpu_dec_get_all_frame_in_buffer 不会阻塞。

接口形式

```
BMVidDecRetStatus bmvpu_dec_get_all_frame_in_buffer (
    BMVidCodHandle vidCodHandle );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。

返回值

当 Frame 全部拿出解码器后，将会返回 0。

bmvpu_dec_delete

当解码任务完成后，调用 bmvpu_dec_delete 销毁解码器，释放解码器占用的资源

接口形式

```
BMVidDecRetStatus bmvpu_dec_delete (
    BMVidCodHandle vidCodHandle );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。

返回值

解码器销毁成功后，会返回 0，否则返回对应的错误码。

bmvpu_dec_get_caps

bmvpu_dec_get_caps 用于获取解码器信息，主要是送入解码器的码流信息。能够获取到的信息可以参考 BMVidStreamInfo 结构体的定义。

接口形式

```
BMVidDecRetStatus bmvpu_dec_get_caps (
    BMVidCodHandle vidCodHandle,
    BMVidStreamInfo *streamInfo );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。

- **streamInfo:**
  输出参数。接收解码器信息。

返回值

成功获取解码器信息，返回 0，否则返回对应的错误码。

bmvpu_dec_get_status

bmvpu_dec_get_status 用于获取解码器的工作状态。目前所支持的解码器状态可以参考 BMDecStatus 的定义。

接口形式

```
BMDecStatus bmvpu_dec_get_status (
    BMVidCodHandle vidCodHandle );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。

返回值

若成功获取到解码器状态，将会返回对应的状态，反则将返回错误码。

bmvpu_dec_get_all_empty_input_buf_cnt

用于获取空闲的 BitStream Buffer 的数量。

接口形式

```
int bmvpu_dec_get_all_empty_input_buf_cnt (
    BMVidCodHandle vidCodHandle );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。

返回值

若成功获取到 BitStream 的数量，则返回空闲 Buffer 的数量，否则返回错误码。

bmvpu_dec_get_stream_buffer_empty_size

用于获取 BitStream Buffer 未使用的空间大小。

接口形式

```
int bmvpu_dec_get_stream_buffer_empty_size (
    BMVidCodHandle vidCodHandle );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。

返回值

若成功则返回 BitStream Buffer 未使用的空间大小，否则返回错误码。

bmvpu_dec_get_pkt_in_buf_cnt

用于获取被占用的 BitStream Buffer 的数量。

接口形式

```
int bmvpu_dec_get_pkt_in_buf_cnt (
    BMVidCodHandle vidCodHandle );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。

返回值

若成功获取到 BitStream 的数量，则返回被占用 Buffer 的数量，否则返回错误码。

bmvpu_dec_dump_stream

用于保存输入码流数据，目前仅支持保存 INTERRUPT 模式的数据；若为 PIC_END 模式，保存的数据可能出现无法播放的情况（缺少一些非显示帧的信息）。该接口已经写入 bmvpu_dec_decode 中，通过设置环境变量 BMVPU_DEC_DUMP_NUM 可以使能该功能。

接口形式

```
BMVidDecRetStatus bmvpu_dec_dump_stream (
    BMVidCodHandle vidCodHandle,
    BMVidStream vidStream );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。
- **vidStream:**
  输入参数。BitStream 的地址和大小。

返回值

码流保存成功，则返回 0，否则返回错误码。

bmvpu_dec_get_core_idx

用于获取 VPU 的 core id。对于 BM1688，有两个解码 core，分别是 core 0 和 core 1。

接口形式

```
int bmvpu_dec_get_core_idx (
    BMVidCodHandle handle );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。

返回值

成功则返回 VPU 的 core id，否则返回错误码。

bmvpu_dec_get_inst_idx

用于获取 VDEC 的 channel id。对于 BM1688 来说，每个 core 最多可以申请 32 个 channel。

接口形式

```
int bmvpu_dec_get_inst_idx (
    BMVidCodHandle vidCodHandle );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。

返回值

成功则返回 VDEC 的 channel id，否则返回错误码。

bmvpu_dec_get_device_fd

用于获取 VDEC channel 对应的设备节点 id。

接口形式

```
int bmvpu_dec_get_device_fd (
    BMVidCodHandle vidCodHandle );
```

参数说明

- **vidCodHandle:**
  输入参数。解码器句柄，在使用该接口前需要先创建解码器。

返回值

成功则返回设备节点的 id，否则返回错误码。

2.6.4 VDEC API 测试例程

针对 VDEC 提供了专用的测试例程 bm_test。其中调用了 VDEC API 来实现对本地 H.264/H.265 码流文件进行解码。

bm_test 提供了以下可配置参数

- **-h** 查看提示信息，可以看到 bm_test 的可选参数以及对应的参数说明。
- **-v** 设置 LOG 等级。0：none；1：error；2：warning；3：info；4：trace。缺省值为 1。
- **-c** 设置输出校验方式。0：不进行输出校验；1：直接对比 yuv 数据；2：对比输出 yuv 的 md5 值。

  开启输出校验前需提前准备好参考文件，通过–comp-skip 设置对比的频率。缺省值为 0。
- **-n** 设置解码的轮数。缺省值为 1。
- **-m** 设置解码模式。0：设置为 INTERRUPT 模式；2：设置为 PIC_END 模式。
- –**input** 设置输入文件。
- **–cbcr_interleave ** 设置 Cb 和 Cr 是否交错，0 表示不交错，1 表示交错.
- –**nv21** 仅当–cbcr_interleave 设置为 1 时有效。设置为 0，以 NV12 格式输出；设置为 1，以 NV21 格式输出。
- –**stream-type** 设置输入码流类型。0：H.264；12：H.265。
- –**ref-yuv** 设置参考文件路径。当开启输出校验后，必须设置此参数，否则不会进行输出校验。
- –**instance** 设置解码线程数。
- –**write_yuv** 设置输出帧数。这个参数只用于设置需要写文件的帧数，并不是解码帧数。
- –**wtl-format** 设置输出模式。0：输出 YUV；1：输出 FBC。
- –**read-block-len** 设置输入缓冲区大小。缺省值为 0x80000。

示例 1

采用 INTERRUPT 模式对 H.264 码流进行解码，采用 yuv420p 格式进行输出。

```
bm_test -c 0 -n 1 -m 0 --input wkc_h264_100.264 --cbcr_interleave 0 --instance 1
```

示例 2

采用 PIC_END 模式对 H.265 码流进行解码，采用 yuv420p 格式进行输出。

```
bm_test -c 0 -n 1 -m 2 --stream-type 12 --input wkc_h265_100.265 --cbcr_interleave 0 --instance 1
```

示例 3

采用 INTERRUPT 模式对 H.264 码流进行解码，采用 nv12 格式进行输出，并且开启 4 个解码线程。

```
bm_test -c 0 -n 1 -m 0 --input wkc_h264_100.264 --cbcr_interleave 1 --nv21 0 --instance 4
```

示例 4

采用 INTERRUPT 模式对 H.264 码流进行解码，采用 nv12 格式进行输出，并且开启 yuv 对比。

```
bm_test -c 1 -n 1 -m 0 --input wkc_h264_100.264 --cbcr_interleave 1 --nv21 0 \
--ref-yuv wkc_h264_100.yuv --instance 1
```

示例 5

采用 PIC_END 模式对 H.265 码流进行解码，采用 yuv420p 格式进行输出，并且开启 yuv 对比，将前 10 帧写入文件。

```
bm_test -c 1 -n 1 -m 2 --stream-type 12 --input wkc_h265_100.265 --cbcr_interleave 0 \
--ref-yuv wkc_h265_100.yuv --instance 1 --write_yuv 10
```

2.7 SOPHGO Video Encoder 使用指南

2.7.1 简介

Encoder 模块主要包括了视频的编码, 该模块囊括了所以对用户开放的接口以及其参数详解

2.7.2 数据结构说明

1. BmVpuEncReturnCodes

```
typedef enum
{
  BM_VPU_ENC_RETURN_CODE_OK = 0,
  BM_VPU_ENC_RETURN_CODE_ERROR,
  BM_VPU_ENC_RETURN_CODE_INVALID_PARAMS,
  BM_VPU_ENC_RETURN_CODE_INVALID_HANDLE,
  BM_VPU_ENC_RETURN_CODE_INVALID_FRAMEBUFFER,
  BM_VPU_ENC_RETURN_CODE_INSUFFICIENT_FRAMEBUFFERS,
  BM_VPU_ENC_RETURN_CODE_INVALID_STRIDE,
  BM_VPU_ENC_RETURN_CODE_WRONG_CALL_SEQUENCE,
  BM_VPU_ENC_RETURN_CODE_TIMEOUT,
  BM_VPU_ENC_RETURN_CODE_RESEND_FRAME,
  BM_VPU_ENC_RETURN_CODE_ENC_END,
  BM_VPU_ENC_RETURN_CODE_END
} BmVpuEncInitialInfo;
```

参数说明

- BM_VPU_ENC_RETURN_CODE_OK

  操作成功完成

- BM_VPU_ENC_RETURN_CODE_ERROR

  通用错误代码，用作其他错误返回代码不匹配时的通用错

- BM_VPU_ENC_RETURN_CODE_INVALID_PARAMS

  输入参数无效

- BM_VPU_ENC_RETURN_CODE_INVALID_HANDLE

  VPU 编码器句柄无效，内部错误，可能是库中的错误，请报告此类错误

- BM_VPU_ENC_RETURN_CODE_INVALID_FRAMEBUFFER

  帧缓冲区信息无效，通常发生在将包含无效值的 BmVpuFramebuffer 结构传递给 bmvpu_enc_register_framebuffers() 函数时

- BM_VPU_ENC_RETURN_CODE_INSUFFICIENT_FRAMEBUFFERS

  注册用于编码的帧缓冲区失败，因为未提供足够的帧缓冲区给 bmvpu_enc_register_framebuffers() 函数

- BM_VPU_ENC_RETURN_CODE_INVALID_STRIDE

  步幅值无效，例如帧缓冲区的一个步幅值无效

- BM_VPU_ENC_RETURN_CODE_WRONG_CALL_SEQUENCE

  在不适当的时间调用函数

- BM_VPU_ENC_RETURN_CODE_TIMEOUT

操作超时

- BM_VPU_ENC_RETURN_CODE_RESEND_FRAME

  重复送帧

- BM_VPU_ENC_RETURN_CODE_ENC_END

  编码结束

- BM_VPU_ENC_RETURN_CODE_END

  编码结束

2. BmVpuEncOutputCodes

```
typedef enum
{
  BM_VPU_ENC_OUTPUT_CODE_INPUT_USED = 1 << 0,
  BM_VPU_ENC_OUTPUT_CODE_ENCODED_FRAME_AVAILABLE = 1 <
↪< 1,
  BM_VPU_ENC_OUTPUT_CODE_CONTAINS_HEADER = 1 << 2
} BmVpuEncOutputCodes;
```

参数说明

- BM_VPU_ENC_OUTPUT_CODE_INPUT_USED

  表示输入数据已被使用。如果未设置该标志位，则编码器尚未使用输入数据，因此请将其再次输入给编码器，直到此标志位被设置或返回错误

- BM_VPU_ENC_OUTPUT_CODE_ENCODED_FRAME_AVAILABLE

  表示现在有一个完全编码的帧可用。传递给 bmvpu_enc_encode() 的 encoded_frame 参数包含有关此帧的信息

- BM_VPU_ENC_OUTPUT_CODE_CONTAINS_HEADER

  表示编码帧中的数据还包含头信息，如 h.264 的 SPS/PSS。头信息始终放置在编码数据的开头，如果未设置 BM_VPU_ENC_OUTPUT_CODE_ENCODED_FRAME_AVAILABLE，则该标志位不会被设置

3. BmVpuEncHeaderDataTypes

```
typedef enum
{
  BM_VPU_ENC_HEADER_DATA_TYPE_VPS_RBSP = 0,
  BM_VPU_ENC_HEADER_DATA_TYPE_SPS_RBSP,
  BM_VPU_ENC_HEADER_DATA_TYPE_PPS_RBSP
} BmVpuEncHeaderDataTypes;
```

参数说明

- BM_VPU_ENC_HEADER_DATA_TYPE_VPS_RBSP

  视频参数集 (VPS) 的 RBSP（Raw Byte Sequence Payload）数据类型
- BM_VPU_ENC_HEADER_DATA_TYPE_SPS_RBSP

  序列参数集 (SPS) 的 RBSP 数据类型
- BM_VPU_ENC_HEADER_DATA_TYPE_PPS_RBSP

  图像参数集 (PPS) 的 RBSP 数据类型

4. BmVpuCodecFormat

```
typedef enum
{
   BM_VPU_CODEC_FORMAT_H264 = 0,
   BM_VPU_CODEC_FORMAT_H265
} BmVpuCodecFormat;
```

参数说明

- BM_VPU_CODEC_FORMAT_H264

  编码类型 h264
- BM_VPU_CODEC_FORMAT_H265

  编码类型 h265

5. BmVpuEncPixFormat

```
typedef enum
{
   BM_VPU_ENC_PIX_FORMAT_YUV420P = 0,
   BM_VPU_ENC_PIX_FORMAT_YUV422P  = 1,
   BM_VPU_ENC_PIX_FORMAT_YUV444P  = 3,
   BM_VPU_ENC_PIX_FORMAT_YUV400 = 4,
   BM_VPU_ENC_PIX_FORMAT_NV12  = 5,
   BM_VPU_ENC_PIX_FORMAT_NV16 = 6,
   BM_VPU_ENC_PIX_FORMAT_NV24 = 7,
   BM_VPU_ENC_PIX_FORMAT_NV21 = 8
} BmVpuEncPixFormat;
```

参数说明

- 编码器输入 yuv 格式

  目前仅支持 nv12，nv21，yuv420p

6. BMVpuEncGopPreset

```
typedef enum
{
  BM_VPU_ENC_GOP_PRESET_ALL_I = 1,
  BM_VPU_ENC_GOP_PRESET_IPP   = 2,
  BM_VPU_ENC_GOP_PRESET_IBBB  = 3,
  BM_VPU_ENC_GOP_PRESET_IBPBP = 4,
  BM_VPU_ENC_GOP_PRESET_IBBBP  = 5,
  BM_VPU_ENC_GOP_PRESET_IPPPP = 6,
  BM_VPU_ENC_GOP_PRESET_IBBBB = 7,
  BM_VPU_ENC_GOP_PRESET_RA_IB = 8
} BMVpuEncGopPreset;
```

参数说明

- BM_VPU_ENC_GOP_PRESET_ALL_I

  全 I 帧模式 gopsize=1

  - BM_VPU_ENC_GOP_PRESET_IPP

    全 IP 帧模式 gopsize=1

  - BM_VPU_ENC_GOP_PRESET_IBBB

    全 IB 帧模式 gopsize=1

  - BM_VPU_ENC_GOP_PRESET_IBPBP

    全 IBP 帧模式 gopsize=2

  - BM_VPU_ENC_GOP_PRESET_IBBBP

    全 IBP 帧模式 gopsize=4

  - BM_VPU_ENC_GOP_PRESET_IPPPP

    全 IP 帧模式 gopsize=4

  - BM_VPU_ENC_GOP_PRESET_IBBBB

    全 IB 帧模式 gopsize=4

  - BM_VPU_ENC_GOP_PRESET_RA_IB

    Random IB 帧模式 gopsize=8

7. BMVpuEncMode

```
typedef enum
{
  BM_VPU_ENC_CUSTOM_MODE = 0,
  BM_VPU_ENC_RECOMMENDED_MODE  = 1,
  BM_VPU_ENC_BOOST_MODE  = 2,
  BM_VPU_ENC_FAST_MODE = 3
} BMVpuEncMode;
```

参数说明

- BM_VPU_ENC_CUSTOM_MODE

  自定义模式

  - BM_VPU_ENC_RECOMMENDED_MODE

    推荐模式（慢编码速度，最高画质）

  - BM_VPU_ENC_BOOST_MODE

    提升模式（正常编码速度，正常画质）

  - BM_VPU_ENC_FAST_MODE

    快速模式（高编码速度，低画质）

8. BmVpuMappingFlags

```
typedef enum
{
  BM_VPU_MAPPING_FLAG_WRITE = 1 << 0,
  BM_VPU_MAPPING_FLAG_READ  = 1 << 1
} BmVpuMappingFlags;
```

参数说明

- BM_VPU_MAPPING_FLAG_WRITE

  可写权限标志

  - BM_VPU_MAPPING_FLAG_READ

    可读权限标志

9. BmVpuEncH264Params

```
typedef struct
{
   int enable_transform8x8;
} BmVpuEncH264Params;
```

参数说明

- enable_transform8x8

  启用 8x8 帧内预测和 8x8 变换。默认值为 1

10. BmVpuEncH265Params

```
typedef struct
{
   int enable_tmvp;
   int enable_wpp;
   int enable_sao;
   int enable_strong_intra_smoothing;
   int enable_intra_trans_skip;
   int enable_intraNxN;
} BmVpuEncH265Params;
```

参数说明

- enable_tmvp

  启用时域运动矢量预测。默认值为 1

- enable_wpp

  启用线性缓冲区模式的波前并行处理。默认值为 0

- enable_sao

  如果设置为 1，则启用 SAO；如果设置为 0，则禁用。默认值为 1

- enable_strong_intra_smoothing

  启用对带有少量 AC 系数的区域进行强烈的帧内平滑，以防止伪影。默认值为 1

- enable_intra_trans_skip

  启用帧内 CU 的变换跳过。默认值为 0

- enable_intraNxN

  启用帧内 NxN PUs。默认值为 1

11. BmVpuEncOpenParams

```
typedef struct
{
    BmVpuCodecFormat codec_format;
    BmVpuEncPixFormat color_format;
    uint32_t frame_width;
    uint32_t frame_height;
    uint32_t timebase_num;
    uint32_t timebase_den;
    uint32_t fps_num;
    uint32_t fps_den;
    int64_t  bitrate;
    uint64_t vbv_buffer_size;
    int cqp;
    BMVpuEncMode enc_mode;
    int max_num_merge;
    int enable_constrained_intra_prediction;
    int enable_wp;
    int disable_deblocking;
    int offset_tc;
    int offset_beta;
    int enable_cross_slice_boundary;
    int enable_nr;
    union
    {
        BmVpuEncH264Params h264_params;
        BmVpuEncH265Params h265_params;
    };
    int soc_idx;
    BMVpuEncGopPreset gop_preset;
    int intra_period;
    int bg_detection;
    int mb_rc;
    int delta_qp;

    /* minimum QP for rate control
    * Default value is 8 */
    int min_qp;
    /* maximum QP for rate control
    * Default value is 51 */
    int max_qp;

    /* roi encoding flag
    * Default value is 0 */
    int roi_enable;
    /* set cmd queue depath
      * Default value is 4
      * the value must 1 <= value <= 4*/
    int cmd_queue_depth;

    int timeout;
```

(续下页)

(接上页)

```
    int timeout_count;

    BmVpuEncBufferAllocFunc buffer_alloc_func;
    BmVpuEncBufferFreeFunc buffer_free_func;
    void *buffer_context;
} BmVpuEncOpenParams;
```

参数说明

- BmVpuCodecFormat codec_format

  指定要生成的编码数据的编码格式

- BmVpuEncPixFormat color_format

  指定传入帧使用的图像格式

- uint32_t frame_width

  传入帧的宽度（以像素为单位），无需对齐

- uint32_t frame_height

  传入帧的高度（以像素为单位），无需对齐

- uint32_t timebase_num

  时间基数，以分数形式给出

- uint32_t timebase_den

  时间分母，以分数形式给出

- uint32_t fps_num

  帧率，以分数形式给出

- uint32_t timebase_den

  帧率分母，以分数形式给出

- int64_t bitrate

  比特率（以 bps 为单位）。如果设置为 0，则禁用码率控制，而使用常量质量模式。默认值为 100000

- int cqp

  常量质量模式的量化参数

- int enc_mode

  0: 自定义模式

  1: 推荐的编码器参数（慢编码速度，最高画质）

  2: 提升模式（正常编码速度，正常画质）

  3: 快速模式（高编码速度，低画质）

- int max_num_merge

  RDO 中的合并候选数（1 或 2）。1：提高编码性能，2：提高编码图像的质量。默认值为 2

- int enable_constrained_intra_prediction

  启用受限帧内预测。如果设置为 1，则启用；如果设置为 0，则禁用。默认值为 0

- int enable_wp

  启用加权预测。默认值为 1

- int disable_deblocking

  如果设置为 1，则禁用去块滤波器。如果设置为 0，则保持启用。默认值为 0

- int offset_tc

  deblocking 滤波器的 Alpha/Tc 偏移。默认值为 0

- int offset_beta

  deblocking 滤波器的 Beta 偏移。默认值为 0

- int enable_cross_slice_boundary

  启用帧内循环滤波中的跨切片边界滤波。默认值为 0

- int enable_nr

  启用帧内循环滤波中的跨切片边界滤波。默认值为 0

- int h264_params

  H.264 编码器参数。(union，从 h264_params 和 h265_params 中选择一个)

- int h265_params

  H.265 编码器参数。(union，从 h264_params 和 h265_params 中选择一个)

- int soc_idx

  仅用于 PCIe 模式。对于 SOC 模式，此值必须为 0。默认值为 0。

- int gop_preset

  1: all I, all Intra, gopsize = 1

  2: I-P-P, consecutive P, cyclic gopsize = 1

  3: I-B-B-B, consecutive B, cyclic gopsize = 1

  4: I-B-P-B-P, gopsize = 2

  5: I-B-B-B-P, gopsize = 4

  6: I-P-P-P-P, consecutive P, cyclic gopsize = 4

  7: I-B-B-B-B, consecutive B, cyclic gopsize = 4

  8: Random Access, I-B-B-B-B-B-B-B-B, cyclic gopsize = 8

低延迟情况为 1、2、3、6、7。默认值为 5

- int intra_period

  GOP 大小内的帧内图片周期。默认值为 28

- int bg_detection

  启用背景检测。默认值为 0

- int mb_rc

  启用 MB 级/CU 级码率控制。默认值为 1

- int delta_qp

  码率控制的最大 delta QP。默认值为 5

- int min_qp

  码率控制的最小 QP。默认值为 8

- int max_qp

  码率控制的最大 QP。默认值为 51

- int roi_enable

  ROI 编码标志。默认值为 0

- int cmd_queue_depth

  编码队列深度，可提升编码器性能，同时需要消耗一定的物理内存

- int timeout

  编码超时时间，默认为 1000ms（即 VPU_WAIT_TIMEOUT）(bm1688 为异步接口，无需设置 timeout)

- int timeout_count

  编码超时重试次数，默认为 40（即 VPU_MAX_TIMEOUT_COUNTS）(bm1688 为异步接口，无需设置 timeout_count)

- BmVpuEncBufferAllocFunc buffer_alloc_func

  缓冲区内存分配函数接口

- BmVpuEncBufferFreeFunc buffer_free_func

  缓冲区内存释放函数接口

- void* buffer_context

  缓冲区上下文信息

12. BmVpuEncInitialInfo

```
typedef struct
{
    uint32_t min_num_rec_fb;
    uint32_t min_num_src_fb;
    uint32_t framebuffer_alignment;
    BmVpuFbInfo rec_fb;
    BmVpuFbInfo src_fb;
} BmVpuEncInitialInfo;
```

参数说明

- min_num_src_fb

  最小推荐帧缓冲区数量，分配少于此数量可能会影响编码质量

- rec_fb

  输入 YUV 数据帧的最小数量，分配少于此数量可能会影响编码

- framebuffer_alignment

  物理帧缓冲区地址的对齐要求

- rec_fb

  用于重建的帧缓冲区大小信息。包括宽度、高度等信息

- src_fb

  输入 YUV 数据的宽高信息

13. BmCustomMapOpt

```
typedef struct
{
    int roiAvgQp;
    int customRoiMapEnable;
    int customLambdaMapEnable;
    int customModeMapEnable;
    int customCoefDropEnable;
    bmvpu_phys_addr_t addrCustomMap;
} BmCustomMapOpt;
```

参数说明

- roiAvgQp

  ROI 映射的平均 QP

- customRoiMapEnable

  是否开启 ROI 映射

- customLambdaMapEnable

  是否开启 Lambda 映射

- customModeMapEnable

  是否指定 CTU 使用帧间编码，否则跳过

- customCoefDropEnable

  对于每个 CTU，是否设置 TQ 系数为全 0，系数全 0 的 CTU 将被丢弃

- addrCustomMap

  自定义映射缓冲区的起始地址

14. BmVpuEncParams

```
typedef struct
{
    int skip_frame;
    int forcePicTypeEnable;
    int forcePicType;
    BmVpuEncAcquireOutputBuffer acquire_output_buffer;
    BmVpuEncFinishOutputBuffer finish_output_buffer;
    void* output_buffer_context;
    BmCustomMapOpt* customMapOpt;
} BmVpuEncParams;
```

参数说明

- skip_frame

  默认值为 0, 禁用跳帧生成。如果设置为 1，则 VPU 忽略给定的原始帧，而生成一个“跳帧”，它是前一帧的复制。这个跳帧被编码为 P 帧

- forcePicTypeEnable

  是否强制指定编码帧类

- forcePicType

  强制指定的编码帧类型（I 帧、P 帧、B 帧、IDR 帧、CRA 帧），只有当 forcePicTypeEnable 为 1 时有效

- acquire_output_buffer

  用于获取输出缓冲区的函数

- finish_output_buffer

  用于释放输出缓冲区的函数

- output_buffer_context

  传递给上述函数的用户提供的值

- customMapOpt

  指向自定义映射选项的指针

15. BmVpuEncoder

```
/* BM VPU Encoder structure. */
typedef struct
{
    void* handle;

    int soc_idx; /* The index of Sophon SoC.
                  * For PCIE mode, please refer to the number at /dev/bm-sophonxx.
                  * For SOC mode, set it to zero. */
    int core_idx; /* unified index for vpu encoder cores at all Sophon SoCs */

    BmVpuCodecFormat codec_format;
    BmVpuEncPixFormat pix_format;

    uint32_t frame_width;
    uint32_t frame_height;

    uint32_t fps_n;
    uint32_t fps_d;

    int first_frame;

    int rc_enable;
    /* constant qp when rc is disabled */
    int cqp;

    /* DMA buffer for working */
    BmVpuEncDMABuffer*   work_dmabuffer;

    /* DMA buffer for bitstream */
    BmVpuEncDMABuffer* bs_dmabuffer;

    unsigned long long bs_virt_addr;
    bmvpu_phys_addr_t bs_phys_addr;

    /* DMA buffer for frame data */
    uint32_t      num_framebuffers;
    void * /*VpuFrameBuffer**/   internal_framebuffers;
    BmVpuFramebuffer* framebuffers;

    /* TODO change all as the parameters of bmvpu_enc_register_framebuffers() */
    /* DMA buffer for colMV */
    BmVpuEncDMABuffer*   buffer_mv;

    /* DMA buffer for FBC luma table */
    BmVpuEncDMABuffer*   buffer_fbc_y_tbl;
```

(续下页)

(接上页)

```
    /* DMA buffer for FBC chroma table */
    BmVpuEncDMABuffer*   buffer_fbc_c_tbl;

    /* Sum-sampled DMA buffer for ME */
    BmVpuEncDMABuffer*   buffer_sub_sam;

    uint8_t* headers_rbsp;
    size_t   headers_rbsp_size;

    BmVpuEncInitialInfo initial_info;

    int timeout;
    int timeout_count;

    /* internal use */
    void *video_enc_ctx;
} BmVpuEncoder;
```

参数说明

- handle

  编码器句柄

- soc_idx

  Sophon SoC 的索引。对于 PCIE 模式，请参考 /dev/bm-sophonxx 中的编号。对于 SOC 模式，请将其设置为零

- core_idx

  所有 Sophon SoC 中 VPU 编码器 core 的统一索引

- codec_format

  编码器使用的视频编解码格式

- color_format

  传入帧使用的图像格式

- frame_width

  传入帧的宽度（以像素为单位）

- frame_height

  传入帧的高度（以像素为单位）

- fps_n

  帧率的分子。

- fps_d

  帧率的分母

- first_frame

  是否为第一帧

- rc_enable

  是否启用码率控制

- cqp

  在禁用码率控制时，使用恒定的量化参数 QP

- work_dmabuffer

  用于编码器工作的 DMA 缓冲区

- bs_dmabuffer

  用于存储码流的 DMA 缓冲区

- bs_virt_addr

  码流的虚拟地址

- bs_phys_addr

  码流的物理地址

- num_framebuffers

  帧缓冲区的数量

- internal_framebuffers

  编码器内部的帧缓冲区

- framebuffers

  帧缓冲区

- buffer_mv

  用于存储运动矢量的 DMA 缓冲区

- buffer_fbc_y_tbl

  用于存储 FBC 亮度表的 DMA 缓冲区

- buffer_fbc_c_tbl

  用于存储 FBC 色度表的 DMA 缓冲区

- buffer_sub_sam

  用于 ME 的子采样 DMA 缓冲区

- headers_rbsp

  帧头 RBSP 数据

- headers_rbsp_size

  帧头 RBSP 数据的大小

- initial_info

  编码器的初始信息

- timeout

  编码超时时间，默认为 1000ms（即 VPU_WAIT_TIMEOUT）

- timeout_count

  编码超时重试次数，默认为 40（即 VPU_MAX_TIMEOUT_COUNTS）

- video_enc_ctx

  编码上下文信息，内部使用

16. BmVpuFbInfo

参数说明

- width

  帧的宽度，按照 VPU 要求的 16 像素边界对齐

- height

  帧的高度，按照 VPU 要求的 16 像素边界对齐

- y_stride

  对齐后的 Y 分量的跨距大小，以字节为单位

- c_stride

  对齐后的 Cb 和 Cr 分量的跨距大小，以字节为单位（可选）

- y_size

  Y 分量的 DMA 缓冲区大小，以字节为单位

- c_size

  Cb 和 Cr 分量的 DMA 缓冲区大小，以字节为单位

- size

  帧缓冲区 DMA 缓冲区的总大小，以字节为单位。这个值包括所有通道的大小

17. BmVpuEncodedFrame

```
typedef struct
{
    uint8_t *data;
    size_t data_size;
    BmVpuFrameType frame_type;
    void *acquired_handle;
    void *context;
    uint64_t pts;
    uint64_t dts;
    int src_idx;
    bmvpu_phys_addr_t u64CustomMapPhyAddr;
    int avg_ctu_qp;
} BmVpuEncodedFrame;
```

参数说明

- uint8_t *data

  在解码时，data 必须指向包含码流数据的内存块，编码器不使用

- size_t data_size

  编码数据的大小。在编码时，由编码器设置，表示获取的输出块的大小，以字节为单位

- BmVpuFrameType frame_type

  编码帧的帧类型（I、P、B 等）。由编码器填充。仅由编码器使用

- void* acquired_handle

  在编码时由用户定义的 **acquire_output_buffer** 函数生成的句柄。仅由编码器使用

- void* context

  用户定义的指针。编码器不会更改此值。这个指针和相应原始帧的指针将具有相同的值，在编码器中传递

- uint64_t pts

  用户定义的显示时间戳（Presentation Timestamp）。与 context 指针一样，编码器只是将其传递到关联的原始帧，并不实际更改其值

- uint64_t dts

  用户定义的解码时间戳（Decoding Timestamp）。与 pts 指针一样，编码器只是将其传递到关联的原始帧，并不实际更改其值

- int src_idx

  原始帧的索引

- bmvpu_phys_addr_t u64CustomMapPhyAddr

  用户自定义映射选项的起始地址

- int avg_ctu_qp

  平均 CTU QP（Quantization Parameter）

18. BmVpuEncDMABuffer

```
typedef struct
{
    unsigned int  size;
    uint64_t      phys_addr;
    uint64_t      virt_addr;
    int           enable_cache;
    int           dmabuf_fd;
} BmVpuEncDMABuffer;
```

参数说明

- size

  物理内存的大小

- phys_addr

  物理内存的地址

- virt_addr

  物理内存 mmap 后的虚拟地址

- enable_cache

  是否开启 cache

- dmabuf_fd

  文件句柄，用户不可以修改直接透传即可

19. BmVpuRawFrame

```
typedef struct
{
    BmVpuFramebuffer *framebuffer;
    void *context;
    uint64_t pts;
    uint64_t dts;
} BmVpuRawFrame;
```

参数说明

- BmVpuFramebuffer *framebuffer

  原始帧的帧缓冲区

- void* context

  用户定义的指针。编码器不会更改此值。这个指针和相应编码帧的指针将具有相同的值，在编码器中传递

- uint64_t pts

  用户定义的显示时间戳（Presentation Timestamp）。与 context 指针一样，编码器只是将其传递到关联的编码帧，并不实际更改其值

- uint64_t dts

  用户定义的解码时间戳（Decoding Timestamp）。与 pts 指针一样，编码器只是将其传递到关联的编码帧，并不实际更改其值

20. BmVpuFramebuffer

```
typedef struct
{
    BmVpuEncDMABuffer *dma_buffer;
    int          myIndex;
    unsigned int y_stride;
    unsigned int cbcr_stride;
    unsigned int width;
    unsigned int height;
    size_t y_offset;
    size_t cb_offset;
    size_t cr_offset;
    int already_marked;
    void *internal;
    void *context;
} BmVpuFramebuffer;
```

参数说明

- BmVpuEncDMABuffer *dma_buffer

  保存 YUV 数据的物理内存

- int myIndex

  YUV 索引，用户设置，用于释放 YUV 数据

- unsigned int y_stride

  Y 通道对齐后的大小

- unsigned int cbcr_stride

  UV 通道对齐后的大小

- unsigned int width

  编码 YUV 图像的宽

- unsigned int height

  编码 YUV 图像的高

- size_t y_offset

  Y 通道 offset。相对于缓冲区起始位置，指定每个分量的起始偏移量。以字节为单位指定

- size_t cb_offset

  U 通道 offset

- size_t cr_offset

  V 通道 offset

- int already_marked

  如果帧缓冲区已在编码器中标记为已使用，则设置为 1。仅供内部使用。不要从外部读取或写入

- void* internal

  内部实现定义的数据。不要修改

- void* context

  用户定义的指针，编码器不会修改此值。用法由用户决定，例如，可以用于标识在编码器中包含该帧的帧缓冲区的序号

2.7.3 API 扩展说明

1. char const * bmvpu_enc_error_string(BmVpuEncReturnCodes code)

功能	返回编码错误码的具体描述
输入参数	BmVpuEncReturnCodes code 编码错误码

2. int bmvpu_enc_get_core_idx(int soc_idx)

功能	在指定的 Sophon SoC 上，获取 VPU 编码器 core 的唯一索引
输入参数	int soc_idx 设备索引号

3. int bmvpu_enc_load(int soc_idx)

功能	加载 Sophon 设备上的视频处理单元（VPU）的编码模块。
输入参数	int soc_idx 设备索引号
函数说明	unload() 和 load() 的调用次数要一致。 在对编码器执行任何其他操作之前，必须先加载 (load) 编码器。 同样，在完成所有编码器活动之前，不得卸载 (unload) 编码器，包括打开编码器实例。

4. int bmvpu_enc_unload(int soc_idx)

功能	卸载 (unload) 编码器
输入参数	int soc_idx 设备索引号

5. void bmvpu_enc_get_bitstream_buffer_info(size_t *size, uint32_t *alignment)

功能	该函数得到编码器所需的 bitstream buffer 的大小 (size) 和所需要的 alignment 值
输入参数	size_t *size - 码流缓冲区所需的物理内存块的大小 uint32_t *alignment - 码流缓冲区所需的物理内存块的对齐方式
返回值	返回编码器的码流缓冲区所需的物理内存块的对齐方式和大小
函数说明	需要在 bmvpu_enc_open 之前调用 用户必须分配至少此大小的 DMA 缓冲区，并且必须根据对齐值对其物理地址进行对齐

6. void bmvpu_enc_set_default_open_params(BmVpuEncOpenParams *open_params, BmVpuCodecFormat codec_format)

功能	设置编码器的默认变量，用于编码器初始化时传递参数
输入参数	BmVpuEncOpenParams *open_params - 用于返回编码器的参数 BmVpuCodecFormat codec_format - 编码器和解码器的选择,h264 和 h265
返回值	无
函数说明	在 bmvpu_enc_open 之前调用 如果调用方只想修改几个成员变量（或者不做修改），可以调用此函数

7. int bmvpu_fill_framebuffer_params(BmVpuFramebuffer *fb, BmVpuFbInfo *info, bm_device_mem_t *fb_dma_buffer, int fb_id, void *context)

功能	根据 bmvpufbinfo 获取的数据填充缓冲区结构体 BmVpuFramebuffer 结构体的字段，同时也设置了指定的 DMA 缓冲区和上下文指针
输入参数	BmVpuFramebuffer *fb - 一个实际的缓冲区，通过接收 info 中的信息来设置缓冲区，例如缓冲区索引、指针和帧的宽高以及 Y、U、V 三分量在帧缓冲区中的偏移等 BmVpuFbInfo *info - 存放缓冲区设置的相关信息 bm_device_mem_t *fb_dma_buffer - 指针指向了实际的 DMA 缓冲区的起始地址，通过这个指针，程序可以直接访问和操作 DMA 缓冲区中的数据，而无需通过中央处理器进行复制或处理。 int fb_id - 缓冲区索引 void *context - 上下文信息
返回值	BM_SUCESS - 分配成功 else - 分配失败
函数说明	无

8. int bmvpu_enc_open(BmVpuEncoder **encoder, BmVpuEncOpenParams *open_params, BmVpuDMABuffer *bs_dmabuffer, BmVpuEncInitialInfo *initial_info)

功能	打开一个新的编码器实例，设置编码器参数并开始接收视频帧
输入参数	BmVpuEncOpenParams *open_params - 编码器各项参数 BmVpuEncoder **encoder - 指向编码器实例的二级指针，接收编码器的属性和视频帧的部分设置，例如设备 id、缓冲区设置和帧率、宽高等 BmVpuDMABuffer *bs_dmabuffer - 指向码流缓冲区的指针，使用之前已经分配的码流缓冲区 BmVpuEncInitialInfo *initial_info - 编码器的初始化信息，返回给用户编码器需要的帧缓冲区最小个数和大小
返回值	BM_SUCESS - 打开成功 else - 打开失败
函数说明	调用前需要确保 BmVpuEncOpenParams 和 BmVpuDMABuffer 不为空

9. int bmvpu_enc_close(BmVpuEncoder *encoder)

功能	关闭编码器实例
输入参数	BmVpuEncoder *encoder - 视频编码器实例
返回值	BM_SUCESS - 关闭成功 else - 关闭失败
函数说明	多次尝试关闭同一实例会导致未定义的行为

10. int bmvpu_enc_encode(BmVpuEncoder *encoder, BmVpuRawFrame const *raw_frame, BmVpuEncodedFrame *encoded_frame, BmVpuEncParams *encoding_params, uint32_t *output_code)

功能	使用给定的编码参数对给定的原始输入帧进行编码。encoded_frame 填充有关于所得到的编码输出帧的信息 bm1688 编码使用异步接口，不支持此接口
输入参数	BmVpuEncoder *encoder - 视频编码器实例 BmVpuRawFrame const *raw_frame - 原始视频帧，包括帧数据、时间戳等 BmVpuEncodedFrame *encoded_frame - 编码后的视频帧，包括帧数据、帧类型、时间戳等 BmVpuEncParams *encoding_params - 用于编码的参数 uint32_t *output_code - 返回输出状态代码
返回值	BM_SUCESS - 编码成功 else - 编码失败
函数说明	编码的帧数据本身被存储在由用户提供的函数（在 encoding_params 中被设置为 acquire_output_buffer 和 finish_output_buffer 函数指针）分配的缓冲区中

11. int bmvpu_enc_send_frame(BmVpuEncoder *encoder, BmVpuRawFrame const *raw_frame, BmVpuEncParams *encoding_params)

功能	使用给定的编码参数对给定的原始输入帧进行编码。bm1688 新增接口
输入参数	BmVpuEncoder *encoder - 视频编码器实例 BmVpuRawFrame const *raw_frame - 原始视频帧，包括帧数据、时间戳等 BmVpuEncParams *encoding_params - 用于编码的参数
返回值	BM_SUCESS - 送数据成功 else - 送数据失败

10. int bmvpu_enc_get_stream(BmVpuEncoder *encoder, BmVpuEncodedFrame *encoded_frame, BmVpuEncParams *encoding_params)

功能	获取编码后的码流数据，数据存在 encoded_frame 中 bm1688 新增接口
输入参数	BmVpuEncoder *encoder - 视频编码器实例 BmVpuEncodedFrame *encoded_frame - 编码后的视频帧，包括帧数据、帧类型、时间戳等 BmVpuEncParams *encoding_params - 用于编码的参数
返回值	BM_SUCESS - 获取成功 else - 获取失败
函数说明	编码的帧数据本身被存储在由用户提供的函数（在 encoding_params 中被设置为 acquire_output_buffer 和 finish_output_buffer 函数指针）分配的缓冲区中

11. int bmvpu_enc_dma_buffer_allocate(int vpu_core_idx, BmVpuEncDMABuffer *buf, unsigned int size)

功能	根据用户指定的 size 分配设备内存
输入参数	int vpu_core_idx - 输入参数，指定编码器所在 core 的索引 BmVpuEncDMABuffer *buf - 输出参数，函数执行后，将会填充该结构体的 phys_addr、size、enable_cache 成员变量 unsigned int size- 输入参数，以字节为单位，指定需要的缓冲区大小
返回值	BM_SUCESS - 关闭成功 else - 关闭失败
函数说明	无

12. int bmvpu_enc_dma_buffer_deallocate(int vpu_core_idx, BmVpuEncDMABuffer *buf)

功能	释放由 bmvpu_enc_dma_buffer_allocate 函数分配的设备内存
输入参数	int vpu_core_idx - 输入参数，指定编码器所在 core 的索引 BmVpuEncDMABuffer *buf - 输出参数，函数执行后，将会填充该结构体的 phys_addr、size、enable_cache 成员变量
返回值	BM_SUCESS - 关闭成功 else - 关闭失败
函数说明	无

13. int bmvpu_enc_dma_buffer_attach(int vpu_core_idx, uint64_t paddr, unsigned int size)

功能	将用户通过 bmvpu_enc_dma_buffer_allocate 函数以外的其它方式申请的设备内存地址绑定至编码器
输入参数	int vpu_core_idx - 输入参数，指定编码器所在 core 的索引 uint64_t paddr - 输入参数，由用户通过 **bmvpu_enc_dma_buffer_allocate** 函数以外的其它方式申请的设备内存地址 unsigned int size 输入参数，该块设备内存大小 (byte)
返回值	BM_SUCESS - 关闭成功 else - 关闭失败
函数说明	无

14. int bmvpu_enc_dma_buffer_deattach(int vpu_core_idx, uint64_t paddr, unsigned int size)

功能	将用户通过 bmvpu_enc_dma_buffer_attach 函数绑定的设备内存解绑
输入参数	int vpu_core_idx - 输入参数，指定编码器所在 core 的索引 uint64_t paddr - 输入参数，由用户通过 **bmvpu_enc_dma_buffer_allocate** 函数以外的其它方式申请的设备内存地址 unsigned int size 输入参数，该块设备内存大小 (byte)
返回值	BM_SUCESS - 关闭成功 else - 关闭失败
函数说明	无

15. int bmvpu_dma_buffer_map(int vpu_core_idx, BmVpuEncDMABuffer *buf, int port_flag)

功能	将对应 core 上申请的设备内存映射到系统内存
输入参数	int vpu_core_idx - 输入参数，指定编码器所在 core 的索引 BmVpuEncDMABuffer *buf - 输入参数，指定设备内存的地址、大小等信息 int port_flag 输入参数，配置 **mmap** 内存可读 (BM_VPU_MAPPING_FLAG_READ) 或可写 (BM_VPU_MAPPING_FLAG_WRITE)
返回值	BM_SUCESS - 关闭成功 else - 关闭失败
函数说明	无

16. int bmvpu_dma_buffer_unmap(int vpu_core_idx, BmVpuEncDMABuffer *buf)

功能	对某个 core 上映射过的设备内存解除映射
输入参数	int vpu_core_idx - 输入参数，指定编码器所在 core 的索引 BmVpuEncDMABuffer *buf - 输入参数，指定设备内存的地址、大小等信息
返回值	BM_SUCESS - 关闭成功 else - 关闭失败
函数说明	无

17. int bmvpu_enc_dma_buffer_flush(int vpu_core_idx, BmVpuEncDMABuffer *buf)

功能	对已分配的设备内存进行 flush 操作
输入参数	int vpu_core_idx - 输入参数，指定编码器所在 core 的索引 BmVpuEncDMABuffer *buf - 输入参数，调用前用户至少要填充该结构体的 phys_addr、size 成员变量
返回值	BM_SUCESS - 关闭成功 else - 关闭失败
函数说明	无

18. int bmvpu_enc_dma_buffer_invalidate(int vpu_core_idx, BmVpuEncDMABuffer *buf)

功能	对已分配的设备内存进行 invalid 操作
输入参数	int vpu_core_idx - 输入参数，指定编码器所在 core 的索引 BmVpuEncDMABuffer *buf - 输入参数，调用前用户至少要填充该结构体的 phys_addr、size 成员变量
返回值	BM_SUCESS - 关闭成功 else - 关闭失败
函数说明	无

19. uint64_t bmvpu_enc_dma_buffer_get_physical_address(BmVpuEncDMABuffer *buf)

功能	返回已分配的设备内存的地址
输入参数	BmVpuEncDMABuffer *buf - 输入参数，已分配的设备内存
返回值	已分配的设备内存的物理地址
函数说明	无

20. unsigned int bmvpu_enc_dma_buffer_get_size(BmVpuEncDMABuffer *buf)

功能	返回已分配的设备内存的大小
输入参数	BmVpuEncDMABuffer *buf - 输入参数
返回值	已分配的设备内存的大小
函数说明	无

21. int bmvpu_enc_upload_data(int vpu_core_idx, const uint8_t* host_va, int host_stride, uint64_t vpu_pa, int vpu_stride, int width, int height)

功能	向使用 bmvpu_enc_dma_buffer_allocate() 分配的设备内存地址传输数据
输入参数	int vpu_core_idx - 输入参数，指定编码器所在 core 的索引 const uint8_t *host_va - 输入参数, 待传输数据的 host 端虚拟地址 int host_stride - 输入参数，host 端的数据跨距 uint64_t vpu_pa - 输入参数，传输数据的目标物理地址 int vpu_stride - 输入参数，device 端的数据跨距 int width - 输入参数，数据宽度 int height - 输入参数，数据高度
返回值	BM_SUCESS - 成功 else - 失败
函数说明	仅支持 PCie 模式

22. int bmvpu_enc_download_data(int vpu_core_idx, uint8_t* host_va, int host_stride, uint64_t vpu_pa, int vpu_stride, int width, int height)

功能	从 bmvpu_enc_dma_buffer_allocate() 分配的设备内存地址向 host 端传输数据
输入参数	int vpu_core_idx - 输入参数，指定编码器所在 core 的索引 const uint8_t *host_va - 输入参数, 待传输数据的 host 端虚拟地址 int host_stride - 输入参数，host 端的数据跨距 uint64_t vpu_pa - 输入参数，传输数据的目标物理地址 int vpu_stride - 输入参数，device 端的数据跨距 int width - 输入参数，数据宽度 int height - 输入参数，数据高度
返回值	BM_SUCESS - 成功 else - 失败
函数说明	仅支持 PCie 模式