BMCV User Guide

BITMAIN

Dec 22, 2021

CONTENTS

注意

您购买的产品、服务或特性等应受算能公司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，算能公司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

北京算能科技有限公司

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 6F 算能

邮编：100094

网址：www.sophon.ai

邮箱：info@sophgo.com

发布记录

版本	发布日期	说明
V2.0.0	2019/09/20	创建
V2.0.1	2019/11/10	新增接口
V2.1.0	2020/05/01	优化
V2.2.0	2020/11/01	优化/新增接口
V2.3.0	2020/03/06	优化/新增接口
V2.4.0	2021/06/19	优化/新增接口
V2.5.0	2021/09/26	新增接口

CHAPTER

ONE

BMCV 介绍

1.1 BMCV 介绍

BMCV 提供了一套基于 Sophon AI 芯片优化的机器视觉库，通过利用芯片的 TPU 和 VPP 模块，可以完成色彩空间转换、尺度变换、仿射变换、透射变换、线性变换、画框、JPEG 编解码、BASE64 编解码、NMS、排序、特征匹配等操作。

CHAPTER

TWO

BM_IMAGE 介绍

2.1 bm_image 结构体

bmcv api 均是围绕 bm_image 来进行的，一个 bm_image 对象对应于一张图片。用户通过 bm_image_create 来构建 bm_image 对象，然后供各个 bmcv 的功能函数使用，使用完需要调用 bm_image_destroy 销毁。

2.1.1 bm_image

bm_image 结构体定义如下:

```
struct bm_image {
    int width;
    int height;
    bm_image_format_ext image_format;
    bm_data_format_ext data_type;
    bm_image_private* image_private;
};
```

bm_image 结构成员包括图片的宽高（width、height），图片格式 image_format，图片数据格式 data_type，以及该结构的私有数据。

2.1.2 bm_image_format_ext image_format

其中 image_format 有以下枚举类型

```
typedef enum bm_image_format_ext_{
    FORMAT_YUV420P,
    FORMAT_YUV422P,
    FORMAT_YUV444P,
    FORMAT_NV12,
    FORMAT_NV21,
    FORMAT_NV16,
    FORMAT_NV61,
    FORMAT_NV24,
    FORMAT_RGB_PLANAR,
```

(continues on next page)

(continued from previous page)

```
    FORMAT_BGR_PLANAR,
    FORMAT_RGB_PACKED,
    FORMAT_BGR_PACKED,
    FORMAT_RGBP_SEPARATE,
    FORMAT_BGRP_SEPARATE,
    FORMAT_GRAY,
    FORMAT_COMPRESSED
} bm_image_format_ext;
```

各个格式说明:

- FORMAT_YUV420P

  表示预创建一个 YUV420 格式的图片, 有三个 plane

- FORMAT_YUV422P

  表示预创建一个 YUV422 格式的图片, 有三个 plane

- FORMAT_YUV444P

  表示预创建一个 YUV444 格式的图片, 有三个 plane

- FORMAT_NV12

  表示预创建一个 NV12 格式的图片, 有两个 plane

- FORMAT_NV21

  表示预创建一个 NV21 格式的图片, 有两个 plane

- FORMAT_NV16

  表示预创建一个 NV16 格式的图片, 有两个 plane

- FORMAT_NV61

  表示预创建一个 NV61 格式的图片, 有两个 plane

- FORMAT_RGB_PLANAR

  表示预创建一个 RGB 格式的图片,RGB 分开排列, 有一个 plane

- FORMAT_BGR_PLANAR

  表示预创建一个 BGR 格式的图片,BGR 分开排列, 有一个 plane

- FORMAT_RGB_PACKED

  表示预创建一个 RGB 格式的图片,RGB 交错排列, 有一个 plane

- FORMAT_BGR_PACKED

  表示预创建一个 BGR 格式的图片,BGR 交错排列, 有一个 plane

- FORMAT_RGBP_SEPARATE

  表示预创建一个 RGB planar 格式的图片，RGB 分开排列并各占一个 plane，共有 3 个 plane

- FORMAT_BGRP_SEPARATE

表示预创建一个 BGR planar 格式的图片，BGR 分开排列并各占一个 plane，共有 3 个 plane

- FORMAT_GRAY

表示预创建一个灰度图格式的图片, 有一个 plane

- FORMAT_COMPRESSED

表示预创建一个 VPU 内部压缩格式的图片，共有四个 plane，分别存放内容如下：

plane0: Y compressed table

plane1: Y compressed data

plane2: CbCr compressed table

plane3: CbCr compressed data

2.1.3 bm_data_format_ext data_type

data_type 有以下枚举类型

```
typedef enum bm_image_data_format_ext_{
    DATA_TYPE_EXT_FLOAT32,
    DATA_TYPE_EXT_1N_BYTE,
    DATA_TYPE_EXT_4N_BYTE,
    DATA_TYPE_EXT_1N_BYTE_SIGNED,
    DATA_TYPE_EXT_4N_BYTE_SIGNED,
}bm_image_data_format_ext;
```

各个格式说明:

- DATA_TYPE_EXT_FLOAT32

表示所创建的图片数据格式为单精度浮点数

- DATA_TYPE_EXT_1N_BYTE

表示所创建图片数据格式为普通无符号 1N UINT8

- DATA_TYPE_EXT_4N_BYTE

表示所创建图片数据格式为 4N UINT8, 即四张无符号 INT8 图片数据交错排列，一个 bm_image 对象其实含有四张属性相同的图片

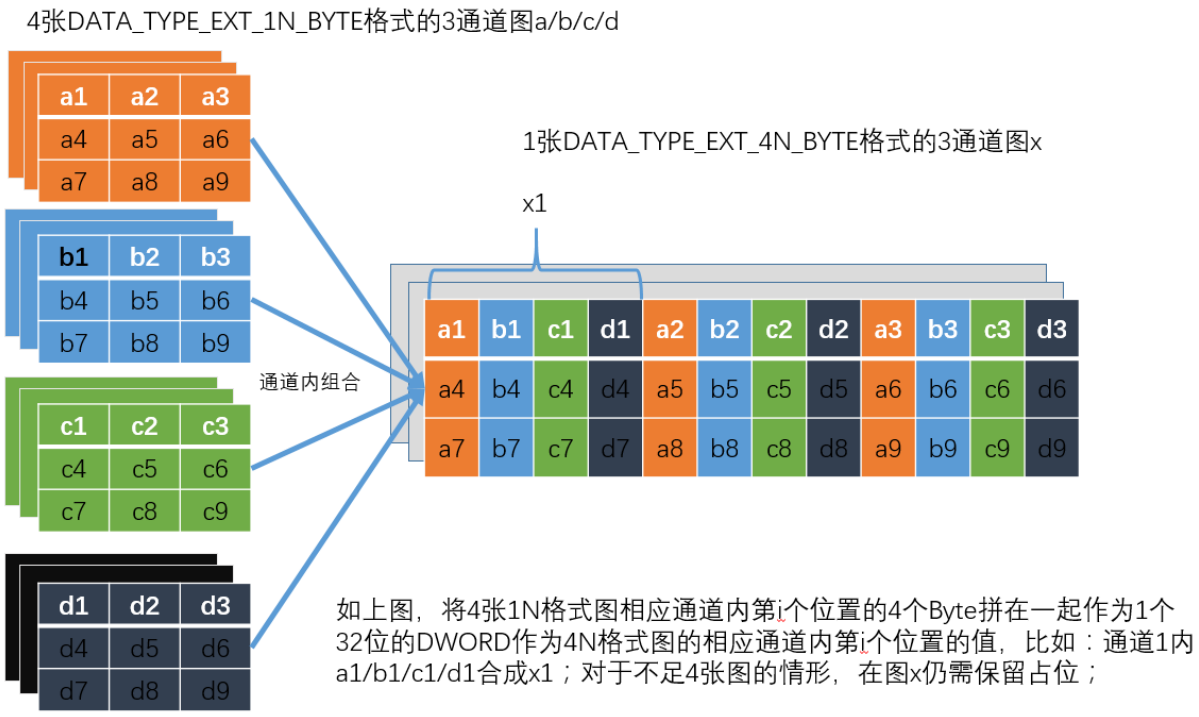
- DATA_TYPE_EXT_1N_BYTE_SIGNED

表示所创建图片数据格式为普通有符号 1N INT8

- DATA_TYPE_EXT_4N_BYTE_SIGNED

表示所创建图片数据格式为 4N INT8, 即四张有符号 INT8 图片数据交错排列

- 其中，对于 4N 排列方式可参考下图：



4N 仅支持 RGB 相关格式，不支持 YUV 相关格式及 FORMAT_COMPRESSED。

2.2 bm_image_create

我们不建议用户直接填充 bm_image 结构，而是通过以下 API 来创建/销毁一个 bm_image 结构。

接口形式:

```
bm_status_t  bm_image_create(
    bm_handle_t handle,
    int img_h,
    int img_w,
    bmcv_image_format_ext image_format,
    bmcv_data_format_ext data_type,
    bm_image *image,
    int* stride);
```

传入参数说明:

- bm_handle_t handle

  输入参数。设备环境句柄，通过调用 bm_dev_request 获取

- int img_h

  输入参数。图片高度

- int img_w

  输入参数。图片宽度

- bmcv_image_format_ext image_format

输入参数。所需创建 bm_image 图片格式，所支持图片格式在 bm_image_format_ext 中介绍

- bm_image_format_ext data_type

输入参数。所需创建 bm_image 数据格式，所支持数据格式在 bm_image_data_format_ext 中介绍

- bm_image *image

输出参数。输出填充的 bm_image 结构指针

- int* stride

输入参数。stride 描述了所创建 bm_image 将要关联的 device memory 内存布局。在每个 plane 的 width stride 值, 以 byte 计数。

返回值说明：

bmcv_image_create 成功调用将返回 BM_SUCCESS，并填充输出的 image 指针结构。这个结构中记录了图片的大小，以及相关格式。但此时并没有与任何 device memory 关联，也没有申请数据对应的 device memory。

注意事项:

1) 以下几种图片格式仅支持 DATA_TYPE_EXT_1N_BYTE

   - FORMAT_YUV420P
   - FORMAT_YUV422P
   - FORMAT_YUV444P
   - FORMAT_NV12
   - FORMAT_NV21
   - FORMAT_NV16
   - FORMAT_NV61
   - FORMAT_GRAY
   - FORMAT_COMPRESSED

   如果试图以上图片格式的其他数据格式 bm_image，则返回失败。

   对于如下 data_type 为 4N 的情形：

   - DATA_TYPE_EXT_4N_BYTE
   - DATA_TYPE_EXT_4N_BYTE_SIGNED

   每调用一次 bm_image_create()，实际是同时为 4 张 image 配置同样的属性。

2) 以下图片格式的宽和高可以是奇数，接口内部会调整到偶数再完成相应功能。但建议尽量使用偶数的宽和高，这样可以发挥最大的效率。

   - FORMAT_YUV420P
   - FORMAT_NV12
   - FORMAT_NV21
   - FORMAT_NV16

- FORMAT_NV61

3) FORMAT_COMPRESSED 图片格式的图片宽度或者 stride 必须 64 对齐，否则返回失败。

4) stride 参数默认值为 NULL，此时默认各个 plane 的数据是 compact 排列，没有 stride。

5) 如果 stride 非 NULL，则会检测 stride 中的 width stride 值是否合法。所谓的合法，即 image_format 对应的所有 plane 的 stride 大于默认 stride。默认 stride 值的计算方法如下：

```
int data_size = 1;
switch (data_type) {
    case DATA_TYPE_EXT_FLOAT32:
        data_size = 4;
        break;
    case DATA_TYPE_EXT_4N_BYTE:
    case DATA_TYPE_EXT_4N_BYTE_SIGNED:
        data_size = 4;
        break;
    default:
        data_size = 1;
        break;
}
int default_stride[3] = {0};
switch (image_format) {
    case FORMAT_YUV420P: {
        image_private->plane_num = 3;
        default_stride[0] = width * data_size;
        default_stride[1] = (ALIGN(width, 2) >> 1) * data_size;
        default_stride[2] = default_stride[1];
        break;
    }
    case FORMAT_YUV422P: {
        default_stride[0] = res->width * data_size;
        default_stride[1] = (ALIGN(res->width, 2) >> 1) * data_size;
        default_stride[2] = default_stride[1];
        break;
    }
    case FORMAT_YUV444P: {
        default_stride[0] = res->width * data_size;
        default_stride[1] = res->width * data_size;
        default_stride[2] = default_stride[1];
        break;
    }
    case FORMAT_NV12:
    case FORMAT_NV21: {
        image_private->plane_num = 2;
        default_stride[0] = width * data_size;
        default_stride[1] = ALIGN(res->width, 2) * data_size;
        break;
    }
```

(continues on next page)

(continued from previous page)

```
    case FORMAT_NV16:
    case FORMAT_NV61: {
        image_private->plane_num = 2;
        default_stride[0] = res->width * data_size;
        default_stride[1] = ALIGN(res->width, 2) * data_size;
        break;
    }
    case FORMAT_GRAY: {
        image_private->plane_num = 1;
        default_stride[0] = res->width * data_size;
        break;
    }
    case FORMAT_COMPRESSED: {
        image_private->plane_num = 4;
        break;
    }
    case FORMAT_BGR_PACKED:
    case FORMAT_RGB_PACKED: {
        image_private->plane_num = 1;
        default_stride[0] = res->width * 3 * data_size;
        break;
    }
    case FORMAT_BGR_PLANAR:
    case FORMAT_RGB_PLANAR: {
        image_private->plane_num = 1;
        default_stride[0] = res->width * data_size;
        break;
    }
    case FORMAT_BGRP_SEPARATE:
    case FORMAT_RGBP_SEPARATE: {
        image_private->plane_num = 3;
        default_stride[0] = res->width * data_size;
        default_stride[1] = res->width * data_size;
        default_stride[2] = res->width * data_size;
        break;
    }
}
```

2.3 bm_image_destroy

销毁 bm_image 对象，与 bm_image_create 成对使用，建议在哪里创建的 bm_image 对象，就在哪里销毁，避免不必要的内存泄漏。

接口形式:

```
bm_status_t bm_image_destroy(
        bm_image image
);
```

传入参数说明:

- bm_image image

输入参数。为待销毁的 bm_image 对象。

返回参数说明:

成功返回将销毁该 bm_image 对象，如果该对象的 device memory 是使用 bm_image_alloc_dev_mem 申请的则释放该空间，否则该对象的 device memory 不会被释放由用户自己管理。

2.4 bm_image_copy_host_to_device

接口形式:

```
bm_status_t bm_image_copy_host_to_device(
        bm_image image,
        void* buffers[]
);
```

该 API 将 host 端数据拷贝到 bm_image 结构对应的 device memory 中。

传入参数说明:

- bm_image image

输入参数。待填充 device memory 数据的 bm_image 对象。

- void* buffers[]

输入参数。host 端指针，buffers 为指向不同 plane 数据的指针，数量应由创建 bm_image 结构时 image_format 对应的 plane 数所决定。每个 plane 的数据量会由创建 bm_image 时的图片宽高、stride、image_format、data_type 决定。具体的计算方法如下：

```
switch (res->image_format) {
    case FORMAT_YUV420P: {
        width[0]  = res->width;
        width[1]  = ALIGN(res->width, 2) / 2;
        width[2]  = width[1];
        height[0] = res->height;
```

(continues on next page)

(continued from previous page)

```
        height[1] = ALIGN(res->height, 2) / 2;
        height[2] = height[1];
        break;
    }
    case FORMAT_YUV422P: {
        width[0]  = res->width;
        width[1]  = ALIGN(res->width, 2) / 2;
        width[2]  = width[1];
        height[0] = res->height;
        height[1] = height[0];
        height[2] = height[1];
        break;
    }
    case FORMAT_YUV444P: {
        width[0]  = res->width;
        width[1]  = width[0];
        width[2]  = width[1];
        height[0] = res->height;
        height[1] = height[0];
        height[2] = height[1];
        break;
    }
    case FORMAT_NV12:
    case FORMAT_NV21: {
        width[0]  = res->width;
        width[1]  = ALIGN(res->width, 2);
        height[0] = res->height;
        height[1] = ALIGN(res->height, 2) / 2;
        break;
    }
    case FORMAT_NV16:
    case FORMAT_NV61: {
        width[0]  = res->width;
        width[1]  = ALIGN(res->width, 2);
        height[0] = res->height;
        height[1] = res->height;
        break;
    }
    case FORMAT_GRAY: {
        width[0]  = res->width;
        height[0] = res->height;
        break;
    }
    case FORMAT_COMPRESSED: {
        width[0]  = res->width;
        height[0] = res->height;
        break;
```

(continues on next page)

(continued from previous page)

```
    }
    case FORMAT_BGR_PACKED:
    case FORMAT_RGB_PACKED: {
        width[0]  = res->width * 3;
        height[0] = res->height;
        break;
    }
    case FORMAT_BGR_PLANAR:
    case FORMAT_RGB_PLANAR: {
        width[0]  = res->width;
        height[0] = res->height * 3;
        break;
    }
    case FORMAT_RGBP_SEPARATE:
    case FORMAT_BGRP_SEPARATE: {
        width[0]  = res->width;
        width[1]  = width[0];
        width[2]  = width[1];
        height[0] = res->height;
        height[1] = height[0];
        height[2] = height[1];
        break;
    }
}
```

因此，对应的 host 端指针所指向的每个 plane 的 buffers 所对应的数据量都应为以上计算出来的 plane_byte_size 值。

返回值说明

该函数成功调用时, 返回 BM_SUCCESS。

Note:

1. 如果 bm_image 未由 bm_image_create 创建，则返回失败。
2. 如果所传入的 bm_image 对象还没有与 device memory 相关联的话，会自动为每个 plane 申请对应 image_private->plane_byte_size 大小的 device memory，并将 host 端数据拷贝到申请的 device memory 中。如果申请 device memory 失败，则该 API 调用失败。
3. 如果所传入的 bm_image 对象图片格式为 FORMAT_COMPRESSED 时，直接返回失败，FORMAT_COMPRESSED 不支持由 host 端指针拷贝输入。
4. 如果拷贝失败, 则该 API 调用失败。

2.5 bm_image_copy_device_to_host

接口形式:

```
bm_status_t bm_image_copy_device_to_host(
        bm_image image,
        void* buffers[]
);
```

传入参数说明:

- bm_image image

输入参数。待传输数据的 bm_image 对象。

- void* buffers[]

输出参数。host 端指针，buffers 为指向不同 plane 数据的指针，数量每个 plane 需要传输的数据量可以通过 bm_image_get_byte_size API 获取。

Note:

1. 如果 bm_image 未由 bm_image_create 创建，则返回失败。
2. 如果 bm_image 没有与 device memory 相关联的话，将返回失败。
3. 数据传输失败的话，API 将调用失败。
4. 如果该函数成功返回, 会将所关联的 device memory 中的数据拷贝到 host 端 buffers 中。

2.6 bm_image_attach

如果用户希望自己管理 device memory，或者 device memory 由外部组件 (VPU/VPP 等) 产生，则可以调用以下 API 将这块 device memory 与 bm_image 相关联。

接口形式:

```
bm_status_t bm_image_attach(
        bm_image image,
        bm_device_mem_t* device_memory
);
```

传入参数说明:

- bm_image image

输入参数。待关联的 bm_image 对象。

- bm_device_mem_t* device_memory

输入参数。填充 bm_image 所需的 device_memory，数量应由创建 bm_image 结构时 image_format 对应的 plane 数所决定。

Note:

1. 如果 bm_image 未由 bm_image_create 创建，则返回失败。
2. 该函数成功调用时，bm_image 对象将与传入的 device_memory 对象相关联。
3. bm_image 不会对通过这种方式关联的 device_memory 进行管理，即在销毁的时候并不会释放对应的 device_memory，需要用户自行管理这块 device_memory。

2.7 bm_image_detach

接口形式:

```
bm_status_t bm_image_detach(
        bm_image
);
```

传入参数说明:

- bm_image image

输入参数。待解关联的 bm_image 对象。

注意事项:

1. 如果传入的 bm_image 对象未被创建，则返回失败。
2. 该函数成功返回时，会对 bm_image 对象关联的 device_memory 进行解关联，bm_image 对象将不再关联 device_memory。
3. 如果解关联的 device_memory 是内部自动申请的话，则会释放这块 device memory。
4. 如果对象未关联任何 device memory，则直接返回成功。

2.8 bm_image_alloc_dev_mem

接口形式:

```
bm_status_t bm_image_alloc_dev_mem(
        bm_image       image
);
```

该 API 为 bm_image 对象申请内部管理内存，所申请 device memory 大小为各个 plane 所需 device memory 大小之和。plane_byte_size 计算方法在 bm_image_copy_host_to_device 中已经介绍，或者通过调用 bm_image_get_byte_size API 来确认。

传入参数说明:

- bm_image image

  输入参数。待申请 device memory 的 bm_image 对象。

注意事项:

1. 如果 bm_image 对象未创建，则返回失败。
2. 如果 image format 为 FORMAT_COMPRESSED，则返回失败。
3. 如果 bm_image 对象已关联 device memory，则会直接成功返回。
4. 所申请 device memory 由内部管理。在 destroy 或者不再使用时释放。

2.9 bm_image_alloc_dev_mem_heap_mask

接口形式:

```
bm_status_t bm_image_alloc_dev_mem_heap_mask(
        bm_image      image,
        int           heap_mask
);
```

该 API 为 bm_image 对象申请内部管理内存，所申请 device memory 大小为各个 plane 所需 device memory 大小之和。plane_byte_size 计算方法在 bm_image_copy_host_to_device 中已经介绍，或者通过调用 bm_image_get_byte_size API 来确认。

传入参数说明:

- bm_image image

  输入参数。待申请 device memory 的 bm_image 对象。

- int heap_mask

输入参数。选择一个或多个 heap id 的掩码，每一个 bit 表示一个 heap id 的是否有效, 1 表示可以在该 heap 上分配，0 则表示不可以在该 heap 上分配，最低位表示 heap0，以此类推。比如 heap_mask=2 则表示指定在 heap1 上分配空间，heap_mask=5 则表示指定在 heap0 或者 heap2 上分配空间

注意事项:

1. 如果 bm_image 对象未创建，则返回失败。
2. 如果 image format 为 FORMAT_COMPRESSED，则返回失败。
3. 如果 bm_image 对象已关联 device memory，则会直接成功返回。
4. 所申请 device memory 由内部管理。在 destroy 或者不再使用时释放。

2.10 bm_image_get_byte_size

获取 bm_image 对象各个 plane 字节大小。

接口形式:

```
bm_status_t bm_image_get_byte_size(
        bm_image image,
        int* size
);
```

传入参数说明:

- bm_image image

输入参数。待获取 device memory 大小的 bm_image 对象。

- int* size

输出参数。返回的各个 plane 字节数结果。

注意事项

1. 如果 bm_image 对象未创建, 则返回失败。
2. 如果 image format 为 FORMAT_COMPRESSED 并且未关联外部 device memory，则返回失败。
3. 该函数成功调用时将在 size 指针中填充各个 plane 所需的 device memory 字节大小。size 大小的计算方法在 bm_image_copy_host_to_device 中已介绍。
4. 如果 bm_image 对象未关联 device memory，除了 FORMAT_COMPRESSED 格式外，其他格式仍能够成功返回并填充 size。

2.11 bm_image_get_device_mem

接口形式:

```
bm_status_t bm_image_get_device_mem(
        bm_image image,
        bm_device_mem_t* mem
);
```

传入参数说明:

- bm_image image

输入参数。待获取 device memory 的 bm_image 对象。

- bm_device_mem_t* mem

输出参数。返回的各个 plane 的 bm_device_mem_t 结构。

注意事项:

1. 该函数将成功返回时，将在 mem 指针中填充 bm_image 对象各个 plane 关联的 bm_device_mem_t 结构。
2. 如果 bm_image 对象未关联 device memory 的话，将返回失败。
3. 如果 bm_image 对象未创建，则返回失败。
4. 如果是 bm_image 内部申请的 device memory 结构，请不要将其释放，以免发生 double free。

2.12 bm_image_alloc_contiguous_mem

为多个 image 分配连续的内存。

接口形式:

```
bm_status_t bm_image_alloc_contiguous_mem(
        int            image_num,
        bm_image       *images
);
```

传入参数说明:

- int image_num

  输入参数。待分配内存的 image 个数

- bm_image *images

  输入参数。待分配内存的 image 的指针

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项:

1、image_num 应该大于 0, 否则将返回错误。

2、如传入的 image 已分配或者 attach 过内存，应先 dettach 已有内存，否则将返回失败。

3、所有待分配的 image 应该尺寸相同，否则将返回错误。

4、当希望 destory 的 image 是通过调用本 api 所分配的内存时，应先调用 bm_image_free_contiguous_mem 将分配内存释放，再用 bm_image_destroy 来实现 destory image

5、bm_image_alloc_contiguous_mem 与 bm_image_free_contiguous_mem 应成对使用。

2.13 bm_image_alloc_contiguous_mem_heap_mask

为多个 image 在指定的 heap 上分配连续的内存。

接口形式:

```
bm_status_t bm_image_alloc_contiguous_mem_heap_mask(
        int            image_num,
        bm_image       *images,
        int            heap_mask
);
```

传入参数说明:

- int image_num

输入参数。待分配内存的 image 个数

- bm_image *images

输入参数。待分配内存的 image 的指针

- int heap_mask

输入参数。选择一个或多个 heap id 的掩码，每一个 bit 表示一个 heap id 的是否有效, 1 表示可以在该 heap 上分配，0 则表示不可以在该 heap 上分配，最低位表示 heap0，以此类推。比如 heap_mask=2 则表示指定在 heap1 上分配空间，heap_mask=5 则表示指定在 heap0 或者 heap2 上分配空间

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项:

1、image_num 应该大于 0, 否则将返回错误。

2、如传入的 image 已分配或者 attach 过内存，应先 dettach 已有内存，否则将返回失败。

3、所有待分配的 image 应该尺寸相同，否则将返回错误。

4、当希望 destory 的 image 是通过调用本 api 所分配的内存时，应先调用 bm_image_free_contiguous_mem 将分配内存释放，再用 bm_image_destroy 来实现 destory image

5、bm_image_alloc_contiguous_mem 与 bm_image_free_contiguous_mem 应成对使用。

2.14 bm_image_free_contiguous_mem

释放通过 bm_image_alloc_contiguous_mem 所分配的多个 image 中的连续内存。

接口形式:

```
bm_status_t bm_image_free_contiguous_mem(
        int image_num,
        bm_image *images
);
```

传入参数说明:

- int image_num

输入参数。待释放内存的 image 个数

- bm_image *images

输入参数。待释放内存的 image 的指针

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项:

1、image_num 应该大于 0，否则将返回错误。

2、所有待释放的 image 应该尺寸相同。

3、bm_image_alloc_contiguous_mem 与 bm_image_free_contiguous_mem 应成对使用。bm_image_free_contiguous_mem 所要释放的内存必须是通过 bm_image_alloc_contiguous_mem 所分配的。

4、应先调用 bm_image_free_contiguous_mem, 将 image 中内存释放，再调 bm_image_destroy 去 destory image。

2.15 bm_image_attach_contiguous_mem

将一块连续内存 attach 到多个 image 中。

接口形式:

```
bm_status_t bm_image_attach_contiguous_mem(
        int image_num,
        bm_image * images,
        bm_device_mem_t dmem
);
```

传入参数说明:

- int image_num

  输入参数。待 attach 内存的 image 个数。

- bm_image *images

  输入参数。待 attach 内存的 image 的指针。

- bm_device_mem_t dmem

  输入参数。已分配好的 device memory 信息。

  返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

  注意事项:

  1、image_num 应该大于 0，否则将返回错误。

  2、所有待 attach 的 image 应该尺寸相同，否则将返回错误。

2.16 bm_image_dettach_contiguous_mem

将一块连续内存从多个 image 中 dettach。

接口形式:

```
bm_status_t bm_image_dettach_contiguous_mem(
        int image_num,
        bm_image * images
);
```

传入参数说明:

- int image_num

  输入参数。待 dettach 内存的 image 个数。

- bm_image *images

  输入参数。待 dettach 内存的 image 的指针。

  返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

  注意事项:

  1、image_num 应该大于 0，否则将返回错误。

  2、所有待 dettach 的 image 应该尺寸相同，否则将返回错误。

3、bm_image_attach_contiguous_mem 与 bm_image_dettach_contiguous_mem 应成对使用。bm_image_dettach_contiguous_mem 所要 dettach 的 device memory 必须是通过 bm_image_attach_contiguous_mem attach 到 image 中的。

2.17 bm_image_get_contiguous_device_mem

从多个内存连续的 image 中得到连续的内存的 device memory 信息。

接口形式:

```
bm_status_t bm_image_get_contiguous_device_mem(
        int image_num,
        bm_image *images,
        bm_device_mem_t * mem
);
```

传入参数说明:

- int image_num

输入参数。待获取信息的 image 个数。

- bm_image *images

输入参数。待获取信息的 image 指针。

- bm_device_mem_t * mem

输出参数。得到的连续内存的 device memory 信息。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项:

1、image_num 应该大于 0，否则将返回错误。

2、所填入的 image 应该尺寸相同，否则将返回错误。

3、所填入的 image 必须是内存连续的，否则返回错误。

4、所填入的 image 内存必须是通过 bm_image_alloc_contiguous_mem 或者 bm_image_attach_contiguous_mem 获得。

2.18 bm_image_get_format_info

该接口用于获取 bm_image 的一些信息。

接口形式:

```
bm_status_t bm_image_get_format_info(
        bm_image *              src,
        bm_image_format_info_t *info
);
```

输入参数说明:

- bm_image* src

  输入参数。所要获取信息的目标 bm_image。

- bm_image_foramt_info_t *info

  输出参数。保存所需信息的数据结构，返回给用户，具体内容见下面的数据结构说明。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

数据结构说明:

```
typedef struct bm_image_format_info {
        int                         plane_nb;
        bm_device_mem_t             plane_data[8];
        int                         stride[8];
        int                         width;
        int                         height;
        bm_image_format_ext         image_format;
        bm_image_data_format_ext    data_type;
        bool                        default_stride;
} bm_image_format_info_t;
```

- int plane_nb

  该 image 的 plane 数量

- bm_device_mem_t plane_data[8]

  各个 plane 的 device memory

- int stride[8];

  各个 plane 的 stride 值

- int width;

  图片的宽度

- int height;

图片的高度

- bm_image_format_ext image_format;

  图片的格式

- bm_image_data_format_ext data_type;

  图片的存储数据类型

- bool default_stride;

  是否使用了默认的 stride

2.19 bm_image_get_stride

该接口用于获取目标 bm_image 的 stride 信息。

接口形式:

```
bm_status_t bm_image_get_stride(
        bm_image image,
        int *stride
);
```

输入参数说明:

- bm_image image

  输入参数。所要获取 stride 信息的目标 bm_image

- int *stride

  输出参数。存放各个 plane 的 stride 的指针

返回值说明

- BM_SUCCESS: 成功
- 其他: 失败

2.20 bm_image_get_plane_num

该接口用于获取目标 bm_image 的 plane 数量。

接口形式:

```
int bm_image_get_plane_num(bm_image image);
```

输入参数说明:

- bm_image image

输入参数。所要获取 plane 数量的目标 bm_image

返回值说明：

返回值即为目标 bm_image 的 plane 数量

2.21 bm_image_is_attached

该接口用于判断目标是否已经 attach 存储空间。

接口形式：

```
bool bm_image_is_attached(bm_image image);
```

输入参数说明：

- bm_image image

输入参数。所要判断是否 attach 存储空间的目标 bm_image。

返回值说明：

若目标 bm_image 已经 attach 存储空间则返回 true，否则返回为 false。

2.22 bm_image_get_handle

该接口用于通过 bm_image 获取句柄 handle。

接口形式：

```
handle_t bm_image_get_handle(bm_image* image);
```

输入参数说明：

- bm_image image

输入参数。所要获取 handle 的目标 bm_image。

返回值说明：

返回值即为目标 bm_image 的句柄 handle。

CHAPTER

THREE

BM_IMAGE DEVICE MEMORY 管理

3.1 bm_image device memory 管理

bm_image 结构需要关联相关 device memory，并且 device memory 中有你所需要的数据时，才能够调用之后的 bmcv API。无论是调用 bm_image_alloc_dev_mem 内部申请，还是调用 bm_image_attach 关联外部内存，均能够使得 bm_image 对象关联 device memory。

判断 bm_image 对象是否已经关联了，可以调用以下 API：

```
bool bm_image_is_attached(
        bm_image image
);
```

传入参数说明：

- bm_image image

输入参数。待判断的 bm_image 对象

返回值说明：

1. 如果 bm_image 对象未创建，则返回 false;
2. 该函数返回 bm_image 对象是否关联了一块 device memory，如果已关联，则返回 true，否则返回 false

注意事项：

1. 一般情况而言，调用 bmcv api 要求输入 bm_image 对象关联 device memory，否则返回失败。而输出 bm_image 对象如果未关联 device memory，则会在在内部调用 bm_image_alloc_dev_mem 函数，内部申请内存。
2. bm_image 调用 bm_image_alloc_dev_mem 所申请的内存都由内部自动管理，在调用 bm_image_destroy、bm_image_detach 或者 bm_image_attach 其他 device memory 时自动释放，无需调用者管理。相反，如果 bm_image_attach 一块 device memory 时，表示这块 memory 将由调用者自己管理。无论是 bm_image_destroy、bm_image_detach，或者再调用 bm_image_attach 其他 device memory，均不会释放，需要调用者手动释放。
3. 目前 device memory 分为三块内存空间：heap0、heap1 和 heap2，三者的区别在于芯片的硬件 VPP 模块是否有读取权限，其他完全相同，因此如果某一 API 需要指定使用硬件 VPP 模块来实现，则必须保证该 API 的输入 bm_image 保存在 heap1 或者 heap2 空间上。

heap id	VPP 是否可读
heap0	否
heap1	是
heap2	是

CHAPTER

FOUR

BMCV API

4.1 bmcv_image_yuv2bgr_ext

该接口实现 YUV 格式到 RGB 格式的转换。

接口形式:

```
bm_status_t bmcv_image_yuv2bgr_ext(
        bm_handle_t handle,
        int image_num,
        bm_image* input,
        bm_image* output
);
```

传入参数说明:

- bm_handle_t handle

  输入参数。设备环境句柄，通过调用 bm_dev_request 获取。

- int image_num

  输入参数。输入/输出 image 数量。

- bm_image* input

  输入参数。输入 bm_image 对象指针。

- bm_image* output

  输出参数。输出 bm_image 对象指针。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

代码示例

```
#include <iostream>
#include <vector>
#include "bmcv_api_ext.h"
```

(continues on next page)

(continued from previous page)

```
#include "bmlib_utils.h"
#include "common.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include <memory>

int main(int argc, char *argv[]) {
    bm_handle_t handle;
    bm_dev_request(&handle, 0);

    int image_n = 1;
    int image_h = 1080;
    int image_w = 1920;
    bm_image src, dst;
    bm_image_create(handle, image_h, image_w, FORMAT_NV12,
            DATA_TYPE_EXT_1N_BYTE, &src);
    bm_image_create(handle, image_h, image_w, FORMAT_BGR_PLANAR,
            DATA_TYPE_EXT_1N_BYTE, &dst);
    std::shared_ptr<u8*> y_ptr = std::make_shared<u8*>(
            new u8[image_h * image_w]);
    std::shared_ptr<u8*> uv_ptr = std::make_shared<u8*>(
            new u8[image_h * image_w / 2]);
    memset((void *)(*y_ptr.get()), 148, image_h * image_w);
    memset((void *)(*uv_ptr.get()), 158, image_h * image_w / 2);
    u8 *host_ptr[] = {*y_ptr.get(), *uv_ptr.get()};
    bm_image_copy_host_to_device(src, (void **)host_ptr);
    bmcv_image_yuv2bgr_ext(handle, image_n, &src, &dst);
    bm_image_destroy(src);
    bm_image_destroy(dst);
    bm_dev_free(handle);
    return 0;
}
```

注意事项:

1. 该 API 输入 NV12/NV21/NV16/NV61/YUV420P 格式的 image 对象，并将转化后的 RGB 数据结果填充到 output image 对象所关联的 device memory 中。
2. 目前该 API 仅支持输入 bm_image 图像格式为:
   - FORMAT_NV12
   - FORMAT_NV21
   - FORMAT_NV16
   - FORMAT_NV61
   - FORMAT_YUV420P

- FORMAT_YUV422P

支持输入 bm_image 数据格式为：

- DATA_TYPE_EXT_1N_BYTE

支持输出 bm_image 图像格式为:

- FORMAT_RGB_PLANAR
- FORMAT_BGR_PLANAR

支持输出 bm_image 数据格式为:

- DATA_TYPE_EXT_FLOAT32,
- DATA_TYPE_EXT_1N_BYTE,
- DATA_TYPE_EXT_4N_BYTE,

如果不满足输入输出格式要求，则返回失败。

3. 输入输出所有 bm_image 结构必须提前创建，否则返回失败。
4. 所有输入 bm_image 对象的 image_format、data_type、width、height 必须相等，所有输出 bm_image 对象的 image_format、data_type、width、height 必须相等，所有输入输出 bm_image 对象的 width、height 必须相等，否则返回失败。
5. image_num 表示输入对象的个数，如果输出 bm_image 数据格式为 DATA_TYPE_EXT_4N_BYTE，则仅输出 1 个 bm_image 4N 对象，反之则输出 image_num 个对象。
6. image_num 必须大于等于 1，小于等于 4，否则返回失败。
7. 所有输入对象必须 attach device memory，否则返回失败
8. 如果输出对象未 attach device memory，则会内部调用 bm_image_alloc_dev_mem 申请内部管理的 device memory，并将转化后的 RGB 数据填充到 device memory 中。

4.2 bmcv_image_warp_affine

该接口实现图像的仿射变换，可实现旋转、平移、缩放等操作。仿射变换是一种二维坐标 (x_0 , y_0) 到二维坐标 (x , y) 的线性变换，该接口的实现是针对输出图像的每一个像素点找到在输入图像中对应的坐标，从而构成一幅新的图像，其数学表达式形式如下：

$$\begin{cases} x_0 = a_1 x + b_1 y + c_1 \\ y_0 = a_2 x + b_2 y + c_2 \end{cases}$$

对应的齐次坐标矩阵表示形式为：

$$\begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

坐标变换矩阵是一个 6 点的矩阵，该矩阵是从输出图像坐标推导输入图像坐标的系数矩阵，可以通过输入输出图像上对应的 3 个点坐标来获取。在人脸检测中，通过获取人脸定位点来获取变换矩阵。

bmcv_affine_matrix 定义了一个坐标变换矩阵，其顺序为 float m[6] = {a1, b1, c1, a2, b2, c2}。而 bmcv_affine_image_matrix 定义了一张图片里面有几个变换矩阵，通常来说一张图片有多个人脸时，会对应多个变换矩阵。

```
typedef struct bmcv_affine_matrix_s{
        float m[6];
} bmcv_warp_matrix;

typedef struct bmcv_affine_image_matrix_s{
        bmcv_affine_matrix *matrix;
        int matrix_num;
} bmcv_affine_image_matrix;
```

接口形式:

```
bm_status_t bmcv_image_warp_affine(
        bm_handle_t handle,
        int image_num,
        bmcv_affine_image_matrix matrix[4],
        bm_image* input,
        bm_image* output,
        int use_bilinear = 0
);
```

输入参数说明

- bm_handle_t handle

输入参数。输入的 bm_handle 句柄。

- int image_num

输入参数。输入图片数，最多支持 4。

- bmcv_affine_image_matrix matrix[4]

输入参数。每张图片对应的变换矩阵数据结构，最多支持 4 张图片。

- bm_image* input

输入参数。输入 bm_image，对于 1N 模式，最多 4 个 bm_image，对于 4N 模式，最多一个 bm_image。

- bm_image* output

输出参数。输出 bm_image，外部需要调用 bmcv_image_create 创建，建议用户调用 bmcv_image_attach 来分配 device memory。如果用户不调用 attach，则内部分配 device memory。对于输出 bm_image，其数据类型和输入一致，即输入是 4N 模式，则输出也是 4N 模式, 输入 1N 模式，输出也是 1N 模式。所需要的 bm_image 大小是所有图片的变换矩阵之和。比如输入 1 个 4N 模式的 bm_image，4 张图片的变换矩阵数目为【3,0,13,5】，则共有变换矩阵 3+0+13+5=21，由于输出是 4N 模式，则需要 (21+4-1)/4=6 个 bm_image 的输出。

- int use_bilinear

输入参数。是否使用 bilinear 进行插值，若为 0 则使用 nearest 插值，若为 1 则使用 bilinear 插值，默认使用 nearest 插值。选择 nearest 插值的性能会优于 bilinear，因此建议首选 nearest 插值，除非对精度有要求时可选择使用 bilinear 插值。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项

1. 该接口所支持的 image_format 包括：

- FORMAT_BGR_PLANAR
- FORMAT_RGB_PLANAR

2. 该接口所支持的 data_type 包括：

- DATA_TYPE_EXT_1N_BYTE
- DATA_TYPE_EXT_4N_BYTE

3. 该接口的输入以及输出 bm_image 均支持带有 stride。
4. 要求该接口输入 bm_image 的 width、height、image_format 以及 data_type 必须保持一致。
5. 要求该接口输出 bm_image 的 width、height、image_format、data_type 以及 stride 必须保持一致。

代码示例

```
#inculde "common.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include <memory>#include <iostream>
#include "bmcv_api_ext.h"
#include "bmlib_utils.h"

int main(int argc, char *argv[]) {
    bm_handle_t handle;

    int image_h = 1080;
    int image_w = 1920;

    int dst_h = 256;
    int dst_w = 256;
    bm_dev_request(&handle, 0);
    bmcv_affine_image_matrix matrix_image;
    matrix_image.matrix_num = 1;
    std::shared_ptr<bmcv_affine_matrix> matrix_data
            = std::make_shared<bmcv_affine_matrix>();
    matrix_image.matrix = matrix_data.get();
```

(continues on next page)

(continued from previous page)

```
    matrix_image.matrix->m[0] = 3.848430;
    matrix_image.matrix->m[1] = -0.02484;
    matrix_image.matrix->m[2] = 916.7;
    matrix_image.matrix->m[3] = 0.02;
    matrix_image.matrix->m[4] = 3.8484;
    matrix_image.matrix->m[5] = 56.4748;

    bm_image src, dst;
    bm_image_create(handle, image_h, image_w, FORMAT_BGR_PLANAR,
            DATA_TYPE_EXT_1N_BYTE, &src);
    bm_image_create(handle, dst_h, dst_w, FORMAT_BGR_PLANAR,
            DATA_TYPE_EXT_1N_BYTE, &dst);

    std::shared_ptr<u8*> src_ptr = std::make_shared<u8*>(
            new u8[image_h * image_w * 3]);
    memset((void *)(*src_ptr.get()), 148, image_h * image_w * 3);
    u8 *host_ptr[] = {*src_ptr.get()};
    bm_image_copy_host_to_device(src, (void **)host_ptr);

    bmcv_image_warp_affine(handle, 1, &matrix_image, &src, &dst);

    bm_image_destroy(src);
    bm_image_destroy(dst);
    bm_dev_free(handle);

    return 0;
}
```

4.3 bmcv_image_warp_perspective

该接口实现图像的透射变换，又称投影变换或透视变换。透射变换将图片投影到一个新的视平面，是一种二维坐标 (x_0 , y_0) 到二维坐标 (x , y) 的非线性变换，该接口的实现是针对输出图像的每一个像素点坐标得到对应输入图像的坐标，然后构成一幅新的图像，其数学表达式形式如下：

$$\begin{cases} x' = a_1 x + b_1 y + c_1 \\ y' = a_2 x + b_2 y + c_2 \\ w' = a_3 x + b_3 y + c_3 \\ x_0 = x'/w' \\ y_0 = y'/w' \end{cases}$$

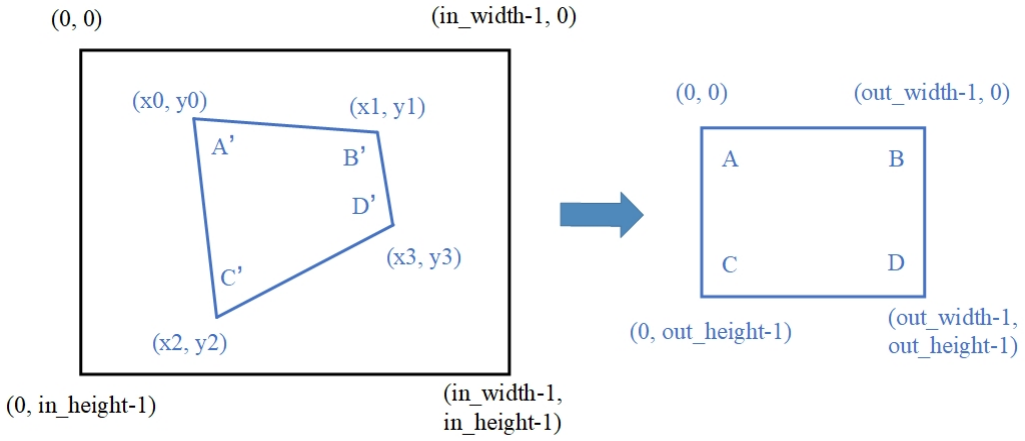
对应的齐次坐标矩阵表示形式为：

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{cases} x_0 = x'/w' \\ y_0 = y'/w' \end{cases}$$

坐标变换矩阵是一个 9 点的矩阵（通常 c3 = 1），利用该变换矩阵可以从输出图像坐标推导出对应的输入原图坐标，该变换矩阵可以通过输入输出图像对应的 4 个点的坐标来获取。

为了更方便地完成透射变换，该库提供了两种形式的接口供用户使用：一种是用户提供变换矩阵给接口作为输入；另一种接口是提供输入图像中 4 个点的坐标作为输入，适用于将一个不规则的四边形透射为一个与输出大小相同的矩形，如下图所示，可以将输入图像 A’ B’ C’ D’ 映射为输出图像 ABCD，用户只需要提供输入图像中 A’、B’、C’、D’ 四个点的坐标即可，该接口内部会根据这四个的坐标和输出图像四个顶点的坐标自动计算出变换矩阵，从而完成该功能。



接口形式一：

```
bm_status_t bmcv_image_warp_perspective(
        bm_handle_t handle,
        int image_num,
        bmcv_perspective_image_matrix matrix[4],
        bm_image* input,
        bm_image* output,
        int use_bilinear = 0
);
```

其中，bmcv_perspective_matrix 定义了一个坐标变换矩阵，其顺序为 float m[9] = {a1, b1, c1, a2, b2, c2, a3, b3, c3}。而 bmcv_perspective_image_matrix 定义了一张图片里面有几个变换矩阵，可以实现对一张图片里的多个小图进行透射变换。

```
typedef struct bmcv_perspective_matrix_s{
        float m[9];
} bmcv_perspective_matrix;

typedef struct bmcv_perspective_image_matrix_s{
        bmcv_perspective_matrix *matrix;
        int matrix_num;
} bmcv_perspective_image_matrix;
```

接口形式二：

```
bm_status_t bmcv_image_warp_perspective_with_coordinate(
        bm_handle_t handle,
        int image_num,
        bmcv_perspective_image_coordinate coord[4],
        bm_image* input,
        bm_image* output,
        int use_bilinear = 0
);
```

其中，bmcv_perspective_coordinate 定义了四边形四个顶点的坐标，按照左上、右上、左下、右下的顺序存储。而 bmcv_perspective_image_coordinate 定义了一张图片里面有几组四边形的坐标，可以实现对一张图片里的多个小图进行透射变换。

```
typedef struct bmcv_perspective_coordinate_s{
        int x[4];
        int y[4];
} bmcv_perspective_coordinate;

typedef struct bmcv_perspective_image_coordinate_s{
        bmcv_perspective_coordinate *coordinate;
        int coordinate_num;
} bmcv_perspective_image_coordinate;
```

输入参数说明

- bm_handle_t handle

  输入参数。输入的 bm_handle 句柄。

- int image_num

  输入参数。输入图片数，最多支持 4。

- bmcv_perspective_image_matrix matrix[4]

  输入参数。每张图片对应的变换矩阵数据结构，最多支持 4 张图片。

- bmcv_perspective_image_coordinate coord[4]

  输入参数。每张图片对应的四边形坐标信息，最多支持 4 张图片。

- bm_image* input

  输入参数。输入 bm_image，对于 1N 模式，最多 4 个 bm_image，对于 4N 模式，最多一个 bm_image。

- bm_image* output

输出参数。输出 bm_image，外部需要调用 bmcv_image_create 创建，建议用户调用 bmcv_image_attach 来分配 device memory。如果用户不调用 attach，则内部分配 device memory。对于输出 bm_image，其数据类型和输入一致，即输入是 4N 模式，则输出也是 4N 模式, 输入 1N 模式，输出也是 1N 模式。所需要的 bm_image 大小是所有图片的变换矩阵之和。比如输入 1 个 4N 模式的 bm_image，4 张图片的变换矩阵数目为【3,0,13,5】，则共有变换矩阵 3+0+13+5=21，由于输出是 4N 模式，则需要 (21+4-1)/4=6 个 bm_image 的输出。

- int use_bilinear

输入参数。是否使用 bilinear 进行插值，若为 0 则使用 nearest 插值，若为 1 则使用 bilinear 插值，默认使用 nearest 插值。选择 nearest 插值的性能会优于 bilinear，因此建议首选 nearest 插值，除非对精度有要求时可选择使用 bilinear 插值。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项

1. 该接口要求输出图像的所有坐标点都能在输入的原图中找到对应的坐标点，不能超出原图大小，建议优先使用接口二，可以自动满足该条件。
2. 该接口所支持的 image_format 包括：

- FORMAT_BGR_PLANAR
- FORMAT_RGB_PLANAR

3. 该接口所支持的 data_type 包括：

- DATA_TYPE_EXT_1N_BYTE
- DATA_TYPE_EXT_4N_BYTE

4. 该接口的输入以及输出 bm_image 均支持带有 stride。
5. 要求该接口输入 bm_image 的 width、height、image_format 以及 data_type 必须保持一致。
6. 要求该接口输出 bm_image 的 width、height、image_format、data_type 以及 stride 必须保持一致。

代码示例

```
#inculde "common.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include <memory>
#include <iostream>
#include "bmcv_api_ext.h"
#include "bmlib_utils.h"

int main(int argc, char *argv[]) {
    bm_handle_t handle;

    int image_h = 1080;
    int image_w = 1920;

    int dst_h = 1080;
    int dst_w = 1920;
    bm_dev_request(&handle, 0);
    bmcv_perspective_image_matrix matrix_image;
```

(continues on next page)

(continued from previous page)

```
    matrix_image.matrix_num = 1;
    std::shared_ptr<bmcv_perspective_matrix> matrix_data
            = std::make_shared<bmcv_perspective_matrix>();
    matrix_image.matrix = matrix_data.get();

    matrix_image.matrix->m[0] = 0.529813;
    matrix_image.matrix->m[1] = -0.806194;
    matrix_image.matrix->m[2] = 1000.000;
    matrix_image.matrix->m[3] = 0.193966;
    matrix_image.matrix->m[4] = -0.019157;
    matrix_image.matrix->m[5] = 300.000;
    matrix_image.matrix->m[6] = 0.000180;
    matrix_image.matrix->m[7] = -0.000686;
    matrix_image.matrix->m[8] = 1.000000;

    bm_image src, dst;
    bm_image_create(handle, image_h, image_w, FORMAT_BGR_PLANAR,
            DATA_TYPE_EXT_1N_BYTE, &src);
    bm_image_create(handle, dst_h, dst_w, FORMAT_BGR_PLANAR,
            DATA_TYPE_EXT_1N_BYTE, &dst);

    std::shared_ptr<u8*> src_ptr = std::make_shared<u8*>(
            new u8[image_h * image_w * 3]);
    memset((void *)(*src_ptr.get()), 148, image_h * image_w * 3);
    u8 *host_ptr[] = {*src_ptr.get()};
    bm_image_copy_host_to_device(src, (void **)host_ptr);

    bmcv_image_warp_perspective(handle, 1, &matrix_image, &src, &dst);

    bm_image_destroy(src);
    bm_image_destroy(dst);
    bm_dev_free(handle);

    return 0;
}
```

4.4 bmcv_image_crop

该接口实现从一幅原图中 crop 出若干个小图。

接口形式：

```
bm_status_t bmcv_image_crop(
        bm_handle_t         handle,
        int                 crop_num,
        bmcv_rect_t*        rects,
```

(continues on next page)

(continued from previous page)

```
        bm_image            input,
        bm_image*           output
);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- int crop_num

输入参数。需要 crop 小图的数量，既是指针 rects 所指向内容的长度，也是输出 bm_image 的数量。

- bmcv_rect_t* rects

输入参数。表示 crop 相关的信息，包括起始坐标、crop 宽高等，具体内容参考下边的数据类型说明。该指针指向了若干个 crop 框的信息，框的个数由 crop_num 决定。

- bm_image input

输入参数。输入的 bm_image，bm_image 需要外部调用 bmcv_image _create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image* output

输出参数。输出 bm_image 的指针，其数量即为 crop_num。bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

数据类型说明：

```
typedef struct bmcv_rect {
    int start_x;
    int start_y;
    int crop_w;
    int crop_h;
} bmcv_rect_t;
```

- start_x 描述了 crop 图像在原图中所在的起始横坐标，自左而右从 0 开始，取值范围 [0, width)。
- start_y 描述了 crop 图像在原图中所在的起始纵坐标。自上而下从 0 开始，取值范围 [0, height)。
- crop_w 描述的 crop 图像的宽度，也就是对应输出图像的宽度。
- crop_h 描述的 crop 图像的高度，也就是对应输出图像的高度。

格式支持：

crop 目前支持以下 image_format:

num	image_format
1	FORMAT_BGR_PACKED
2	FORMAT_BGR_PLANAR
3	FORMAT_RGB_PACKED
4	FORMAT_RGB_PLANAR
5	FORMAT_GRAY

crop 目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_FLOAT32
2	DATA_TYPE_EXT_1N_BYTE
3	DATA_TYPE_EXT_1N_BYTE_SIGNED

注意事项:

1、在调用 bmcv_image_crop() 之前必须确保输入的 image 内存已经申请。

2、input output 的 data_type，image_format 必须相同。

3、为了避免内存越界, start_x + crop_w 必须小于等于输入图像的 width，start_y + crop_h 必须小于等于输入图像的 height。

代码示例:

```
int channel   = 3;
int in_w      = 400;
int in_h      = 400;
int out_w     = 800;
int out_h     = 800;
int    dev_id = 0;
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<unsigned char> src_ptr(
        new unsigned char[channel * in_w * in_h],
        std::default_delete<unsigned char[]>());
std::shared_ptr<unsigned char> res_ptr(
        new unsigned char[channel * out_w * out_h],
        std::default_delete<unsigned char[]>());
unsigned char * src_data = src_ptr.get();
unsigned char * res_data = res_ptr.get();
for (int i = 0; i < channel * in_w * in_h; i++) {
    src_data[i] = rand() % 255;
}
// calculate res
bmcv_rect_t crop_attr;
crop_attr.start_x   = 0;
crop_attr.start_y   = 0;
```

(continues on next page)

(continued from previous page)

```
crop_attr.crop_w    = 50;
crop_attr.crop_h    = 50;
bm_image input, output;
bm_image_create(handle,
        in_h,
        in_w,
        FORMAT_RGB_PLANAR,
        DATA_TYPE_EXT_1N_BYTE,
        &input);
bm_image_alloc_dev_mem(input);
bm_image_copy_host_to_device(input, (void **)&src_data);
bm_image_create(handle,
        out_h,
        out_w,
        FORMAT_RGB_PLANAR,
        DATA_TYPE_EXT_1N_BYTE,
        &output);
bm_image_alloc_dev_mem(output);
if (BM_SUCCESS != bmcv_image_crop(handle, 1, &crop_attr, input, &output))
↪{
    std::cout << "bmcv_copy_to error !!!" << std::endl;
    bm_image_destroy(input);
    bm_image_destroy(output);
    bm_dev_free(handle);
    exit(-1);
}
bm_image_copy_device_to_host(output, (void **)&res_data);
bm_image_destroy(input);
bm_image_destroy(output);
bm_dev_free(handle);
```

4.5 bmcv_image_resize

该接口用于实现图像尺寸的变化, 如放大、缩小、抠图等功能。

接口形式:

```
bm_status_t bmcv_image_resize(
        bm_handle_t handle,
        int input_num,
        bmcv_resize_image resize_attr[4],
        bm_image* input,
        bm_image* output
);
```

参数说明:

- bm_handle_t handle

输入参数。bm_handle 句柄。

- int input_num

输入参数。输入图片数，最多支持 4 ，如果 input_num > 1, 那么多个输入图像必须是连续存储的（可以使用 bm_image_alloc_contiguous_mem 给多张图申请连续空间）。

- bmcv_resize_image resize_attr [4]

输入参数。每张图片对应的 resize 参数, 最多支持 4 张图片。

- bm_image* input

输入参数。输入 bm_image。每个 bm_image 需要外部调用 bmcv_image _create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image* output

输出参数。输出 bm_image。每个 bm_image 需要外部调用 bmcv_image_create 创建，image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存，如果不主动分配将在 api 内部进行自行分配。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

数据类型说明:

```
typedef struct bmcv_resize_s{
        int start_x;
        int start_y;
        int in_width;
        int in_height;
        int out_width;
        int out_height;
}bmcv_resize_t;

typedef struct bmcv_resize_image_s{
        bmcv_resize_t *resize_img_attr;
        int roi_num;
        unsigned char stretch_fit;
        unsigned char padding_b;
        unsigned char padding_g;
        unsigned char padding_r;
        unsigned int interpolation;
}bmcv_resize_image;
```

- bmcv_resize_image 描述了一张图中 resize 配置信息。
- roi_num 描述了一副图中需要进行 resize 的子图总个数。

- stretch_fit 表示是否按照原图比例对图片进行缩放，1 表示无需按照原图比例进行缩放，0 表示按照原图比例进行缩放，当采用这种方式的时候，结果图片中为进行缩放的地方将会被填充成特定值。
- padding_b 表示当 stretch_fit 设成 0 的情况下，b 通道上被填充的值。
- padding_r 表示当 stretch_fit 设成 0 的情况下，r 通道上被填充的值。
- padding_g 表示当 stretch_fit 设成 0 的情况下，g 通道上被填充的值。
- interpolation 表示缩图所使用的算法。BMCV_INTER_NEAREST 表示最近邻算法，BMCV_INTER_LINEAR 表示线性插值算法。
- start_x 描述了 resize 起始横坐标 (相对于原图)，常用于抠图功能。
- start_y 描述了 resize 起始纵坐标 (相对于原图)，常用于抠图功能。
- in_width 描述了 crop 图像的宽。
- in_height 描述了 crop 图像的高。
- out_width 描述了输出图像的宽。
- out_height 描述了输出图像的高。

代码示例:

```
int image_num = 4;
int crop_w = 711, crop_h = 400, resize_w = 711, resize_h = 400;
int image_w = 1920, image_h = 1080;
int img_size_i = image_w * image_h * 3;
int img_size_o = resize_w * resize_h * 3;
std::unique_ptr<unsigned char[]> img_data(
        new unsigned char[img_size_i * image_num]);
std::unique_ptr<unsigned char[]> res_data(
        new unsigned char[img_size_o * image_num]);
memset(img_data.get(), 0x11, img_size_i * image_num);
memset(res_data.get(), 0, img_size_o * image_num);
bmcv_resize_image resize_attr[image_num];
bmcv_resize_t resize_img_attr[image_num];
for (int img_idx = 0; img_idx < image_num; img_idx++) {
  resize_img_attr[img_idx].start_x = 0;
  resize_img_attr[img_idx].start_y = 0;
  resize_img_attr[img_idx].in_width = crop_w;
  resize_img_attr[img_idx].in_height = crop_h;
  resize_img_attr[img_idx].out_width = resize_w;
  resize_img_attr[img_idx].out_height = resize_h;
}
for (int img_idx = 0; img_idx < image_num; img_idx++) {
  resize_attr[img_idx].resize_img_attr = &resize_img_attr[img_idx];
  resize_attr[img_idx].roi_num = 1;
  resize_attr[img_idx].stretch_fit = 1;
  resize_attr[img_idx].interpolation = BMCV_INTER_NEAREST;
}
```

(continues on next page)

(continued from previous page)

```
bm_image input[image_num];
bm_image output[image_num];
for (int img_idx = 0; img_idx < image_num; img_idx++) {
  int input_data_type = DATA_TYPE_EXT_1N_BYTE;
  bm_image_create(handle,
       image_h,
       image_w,
       FORMAT_BGR_PLANAR,
       (bm_image_data_format_ext)input_data_type,
       &input[img_idx]);
}
bm_image_alloc_contiguous_mem(image_num, input, 1);
for (int img_idx = 0; img_idx < image_num; img_idx++) {
  unsigned char * input_img_data = img_data.get() + img_size_i * img_idx;
  bm_image_copy_host_to_device(input[img_idx],
  (void **)&input_img_data);
}
for (int img_idx = 0; img_idx < image_num; img_idx++) {
  int output_data_type = DATA_TYPE_EXT_1N_BYTE;
  bm_image_create(handle,
       resize_h,
       resize_w,
       FORMAT_BGR_PLANAR,
       (bm_image_data_format_ext)output_data_type,
       &output[img_idx]);
}
bm_image_alloc_contiguous_mem(image_num, output, 1);
bmcv_image_resize(handle, image_num, resize_attr, input, output);
for (int img_idx = 0; img_idx < image_num; img_idx++) {
  unsigned char *res_img_data = res_data.get() + img_size_o * img_idx;
  bm_image_copy_device_to_host(output[img_idx],
                               (void **)&res_img_data);
}
bm_image_free_contiguous_mem(image_num, input);
bm_image_free_contiguous_mem(image_num, output);
for(int i = 0; i < image_num; i++) {
  bm_image_destroy(input[i]);
  bm_image_destroy(output[i]);
}
```

格式支持:

1. resize 支持下列 image_format 的转化：

- FORMAT_BGR_PLANAR -> FORMAT_BGR_PLANAR
- FORMAT_RGB_PLANAR -> FORMAT_RGB_PLANAR
- FORMAT_BGR_PACKED -> FORMAT_BGR_PLANAR

- FORMAT_RGB_PACKED -> FORMAT_RGB_PLANAR

2. resize 支持下列情形 data type 之间的转换：

- DATA_TYPE_EXT_1N_BYTE -> DATA_TYPE_EXT_1N_BYTE (1 幅图像 resize (crop) 一幅图像的情形)
- DATA_TYPE_EXT_FLOAT32 -> DATA_TYPE_EXT_FLOAT32 (1 幅图像 resize (crop) 一幅图像的情形)
- DATA_TYPE_EXT_4N_BYTE -> DATA_TYPE_EXT_4N_BYTE (1 幅图像 resize (crop) 一幅图像的情形)
- DATA_TYPE_EXT_4N_BYTE -> DATA_TYPE_EXT_1N_BYTE (1 幅图像 resize (crop) 一幅图像的情形)
- DATA_TYPE_EXT_4N_BYTE -> DATA_TYPE_EXT_1N_BYTE (1 幅图像 resize (crop) 多幅图像的情形)
- DATA_TYPE_EXT_1N_BYTE -> DATA_TYPE_EXT_1N_BYTE (1 幅图像 resize (crop) 多幅图像的情形)
- DATA_TYPE_EXT_FLOAT32 -> DATA_TYPE_EXT_FLOAT32 (1 幅图像 resize (crop) 多幅图像的情形)

注意事项:

1. 在调用 bmcv_image_resize() 之前必须确保输入的 image 内存已经申请。
2. 支持最大尺寸为 2048*2048，最小尺寸为 16*16，最大缩放比为 32。

4.6 bmcv_image_convert_to

该接口用于实现图像像素线性变化，具体数据关系可用如下公式表示：

$$y = kx + b$$

接口形式:

```
bm_status_t bmcv_image_convert_to (
        bm_handle_t handle,
        int input_num,
        bmcv_convert_to_attr convert_to_attr,
        bm_image* input,
        bm_image* output
);
```

输入参数说明:

- bm_handle_t handle

  输入参数。bm_handle 句柄。

- int input_num

输入参数。输入图片数，如果 input_num > 1，那么多个输入图像必须是连续存储的（可以使用 bm_image_alloc_contiguous_mem 给多张图申请连续空间）。

- bmcv_convert_to_attr convert_to_attr

输入参数。每张图片对应的配置参数。

- bm_image* input

输入参数。输入 bm_image。每个 bm_image 外部需要调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image* output

输出参数。输出 bm_image。每个 bm_image 外部需要调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

数据类型说明:

```
typedef struct bmcv_convert_to_attr_s{
        float alpha_0;
        float beta_0;
        float alpha_1;
        float beta_1;
        float alpha_2;
        float beta_2;
} bmcv_convert_to_attr;
```

- alpha_0 描述了第 0 个 channel 进行线性变换的系数
- beta_0 描述了第 0 个 channel 进行线性变换的偏移
- alpha_1 描述了第 1 个 channel 进行线性变换的系数
- beta_1 描述了第 1 个 channel 进行线性变换的偏移
- alpha_2 描述了第 2 个 channel 进行线性变换的系数
- beta_2 描述了第 2 个 channel 进行线性变换的偏移

代码示例:

```
int image_num = 4, image_channel = 3;
int image_w = 1920, image_h = 1080;
bm_image input_images[4], output_images[4];
bmcv_convert_to_attr convert_to_attr;
convert_to_attr.alpha_0 = 1;
convert_to_attr.beta_0 = 0;
```

(continues on next page)

(continued from previous page)

```
convert_to_attr.alpha_1 = 1;
convert_to_attr.beta_1 = 0;
convert_to_attr.alpha_2 = 1;
convert_to_attr.beta_2 = 0;
int img_size = image_w * image_h * image_channel;
std::unique_ptr<unsigned char[]> img_data(
        new unsigned char[img_size * image_num]);
std::unique_ptr<unsigned char[]> res_data(
        new unsigned char[img_size * image_num]);
memset(img_data.get(), 0x11, img_size * image_num);
for (int img_idx = 0; img_idx < image_num; img_idx++) {
  bm_image_create(handle,
        image_h,
        image_w,
        FORMAT_BGR_PLANAR,
        DATA_TYPE_EXT_1N_BYTE,
        &input_images[img_idx]);
}
bm_image_alloc_contiguous_mem(image_num, input_images, 0);
for (int img_idx = 0; img_idx < image_num; img_idx++) {
  unsigned char *input_img_data = img_data.get() + img_size * img_idx;
  bm_image_copy_host_to_device(input_images[img_idx],
        (void **)&input_img_data);
}

for (int img_idx = 0; img_idx < image_num; img_idx++) {
  bm_image_create(handle,
        image_h,
        image_w,
        FORMAT_BGR_PLANAR,
        DATA_TYPE_EXT_1N_BYTE,
        &output_images[img_idx]);
}
bm_image_alloc_contiguous_mem(image_num, output_images, 1);
bmcv_image_convert_to(handle, image_num, convert_to_attr, input_images,
        output_images);
for (int img_idx = 0; img_idx < image_num; img_idx++) {
  unsigned char *res_img_data = res_data.get() + img_size * img_idx;
  bm_image_copy_device_to_host(output_images[img_idx],
        (void **)&res_img_data);
}
bm_image_free_contiguous_mem(image_num, input_images);
bm_image_free_contiguous_mem(image_num, output_images);
for(int i = 0; i < image_num; i++) {
  bm_image_destroy(input_images[i]);
  bm_image_destroy(output_images[i]);
}
```

格式支持:

1. 该接口支持下列 image_format 的转化:

- FORMAT_BGR_PLANAR ——> FORMAT_BGR_PLANAR
- FORMAT_RGB_PLANAR ——> FORMAT_RGB_PLANAR
- FORMAT_GRAY ——> FORMAT_GRAY

2. 该接口支持下列情形 data type 之间的转换:

- DATA_TYPE_EXT_1N_BYTE ——> DATA_TYPE_EXT_FLOAT32
- DATA_TYPE_EXT_1N_BYTE ——> DATA_TYPE_EXT_1N_BYTE
- DATA_TYPE_EXT_1N_BYTE_SIGNED ——> DATA_TYPE_EXT_1N_BYTE_SIGNED
- DATA_TYPE_EXT_1N_BYTE ——> DATA_TYPE_EXT_1N_BYTE_SIGNED
- DATA_TYPE_EXT_FLOAT32 ——> DATA_TYPE_EXT_FLOAT32
- DATA_TYPE_EXT_4N_BYTE ——> DATA_TYPE_EXT_FLOAT32

注意事项:

1. 在调用 bmcv_image_convert_to() 之前必须确保输入的 image 内存已经申请。
2. 输入的各个 image 的宽、高以及 data_type、image_format 必须相同。
3. 输出的各个 image 的宽、高以及 data_type、image_format 必须相同。
4. 输入 image 宽高必须等于输出 image 宽高。
5. image_num 必须大于 0。
6. 输入以及输出的 image_format 只允许为 FORMAT_BGR_PLANAR 或者 FORMAT_RGB_PLANAR。
7. 输出 image 的 stride 必须等于 width。
8. 输入 image 的 stride 必须大于等于 width。
9. 支持最大尺寸为 2048*2048，最小尺寸为 16*16，当 image format 为 DATA_TYPE_EXT_4N_BYTE 时，w * h 不应大于 1024 * 1024。

4.7 bmcv_image_storage_convert

该接口将源图像格式的对应的数据转换为目的图像的格式数据，并填充在目的图像关联的 device memory 中。

接口形式:

```
bm_status_t bmcv_image_storage_convert(
        bm_handle_t handle,
        int image_num,
        bm_image* input_image,
        bm_image* output_image
);
```

传入参数说明:

- bm_handle_t handle

输入参数。设备环境句柄，通过调用 bm_dev_request 获取。

- int image_num

输入参数。输入/输出 image 数量。

- bm_image* input

输入参数。输入 bm_image 对象指针。

- bm_image* output

输出参数。输出 bm_image 对象指针。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项

1. 该 API 支持以下所有格式的两两相互转换:

- (FORMAT_NV12, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_NV21, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_NV16, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_NV61, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_YUV420P, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_YUV444P, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_RGB_PLANAR, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_RGB_PLANAR, DATA_TYPE_EXT_4N_BYTE)
- (FORMAT_RGB_PLANAR, DATA_TYPE_EXT_FLOAT32)
- (FORMAT_BGR_PLANAR, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_BGR_PLANAR, DATA_TYPE_EXT_4N_BYTE)
- (FORMAT_BGR_PLANAR, DATA_TYPE_EXT_FLOAT32)
- (FORMAT_RGB_PACKED, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_RGB_PACKED, DATA_TYPE_EXT_4N_BYTE)
- (FORMAT_RGB_PACKED, DATA_TYPE_EXT_FLOAT32)
- (FORMAT_BGR_PACKED, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_BGR_PACKED, DATA_TYPE_EXT_4N_BYTE)
- (FORMAT_BGR_PACKED, DATA_TYPE_EXT_FLOAT32)

- (FORMAT_RGBP_SEPARATE, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_RGBP_SEPARATE, DATA_TYPE_EXT_4N_BYTE)
- (FORMAT_RGBP_SEPARATE, DATA_TYPE_EXT_FLOAT32)
- (FORMAT_BGRP_SEPARATE, DATA_TYPE_EXT_1N_BYTE)
- (FORMAT_BGRP_SEPARATE, DATA_TYPE_EXT_4N_BYTE)
- (FORMAT_BGRP_SEPARATE, DATA_TYPE_EXT_FLOAT32)。

如果输入输出 image 对象不在以上格式中，则返回失败。

2. 输入输出所有 bm_image 结构必须提前创建，否则返回失败。
3. 所有输入 bm_image 对象的 image_format，data_type，width，height 必须相等，所有输出 bm_image 对象的 image_format，data_type，width，height 必须相等，所有输入输出 bm_image 对象的 width，height 必须相等，否则返回失败。
4. image_num 表示输入图像个数，如果输入图像数据格式为 DATA_TYPE_EXT_4N_BYTE，则输入 bm_image 对象为 1 个，在 4N 中有 image_num 个有效图片。如果输入图像数据格式不是 DATA_TYPE_EXT_4N_BYTE，则输入 image_num 个 bm_image 对象。如果输出 bm_image 数据格式为 DATA_TYPE_EXT_4N_BYTE，则输出 1 个 bm_image 4N 对象，对象中有 bm_image 个有效图片。反之如果输出图像数据格式不是 DATA_TYPE_EXT_4N_BYTE，则输出 image_num 个对象。
5. image_num 必须大于等于 1，小于等于 4，否则返回失败。
6. 所有输入对象必须 attach device memory，否则返回失败。
7. 如果输出对象未 attach device memory，则会内部调用 bm_image_alloc_dev_mem 申请内部管理的 device memory, 并将转化后的数据填充到 device memory 中。
8. 如果输入图像和输出图像格式相同，则直接返回成功，且不会将原数据拷贝到输出图像中。
9. 暂不支持 image_w > 8192 时的图像格式转换，如果 image_w > 8192 则返回失败。

代码示例:

```
#include <iostream>
#include <vector>
#include "bmcv_api_ext.h"
#include "bmlib_utils.h"
#include "common.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include <memory>

int main(int argc, char *argv[]) {
    bm_handle_t handle;
    bm_dev_request(&handle, 0);

    int image_n = 1;
    int image_h = 1080;
```

(continues on next page)

(continued from previous page)

```
    int image_w = 1920;
    bm_image src, dst;
    bm_image_create(handle, image_h, image_w, FORMAT_NV12,
            DATA_TYPE_EXT_1N_BYTE, &src);
    bm_image_create(handle, image_h, image_w, FORMAT_BGR_PLANAR,
            DATA_TYPE_EXT_1N_BYTE, &dst);
    std::shared_ptr<u8*> y_ptr = std::make_shared<u8*>(
            new u8[image_h * image_w]);
    std::shared_ptr<u8*> uv_ptr = std::make_shared<u8*>(
            new u8[image_h * image_w / 2]);
    memset((void *)(*y_ptr.get()), 148, image_h * image_w);
    memset((void *)(*uv_ptr.get()), 158, image_h * image_w / 2);
    u8 *host_ptr[] = {*y_ptr.get(), *uv_ptr.get()};
    bm_image_copy_host_to_device(src, (void **)host_ptr);
    bmcv_image_storage_convert(handle, image_n, &src, &dst);
    bm_image_destroy(src);
    bm_image_destroy(dst);
    bm_dev_free(handle);
    return 0;
}
```

4.8 bmcv_image_vpp_basic

bm1684 上有专门的视频后处理模块 VPP，在满足一定条件下可以一次实现 crop、color-space-convert、resize 以及 padding 功能，速度比 TPU 更快。该 API 可以实现对多张图片的 crop、color-space-convert、resize、padding 及其任意若干个功能的组合。

```
bm_status_t bmcv_image_vpp_basic(
    bm_handle_t           handle,
    int                   in_img_num,
    bm_image*             input,
    bm_image*             output,
    int*                  crop_num_vec = NULL,
    bmcv_rect_t*          crop_rect = NULL,
    bmcv_padding_atrr_t*  padding_attr = NULL,
    bmcv_resize_algorithm algorithm = BMCV_INTER_LINEAR,
    csc_type_t            csc_type = CSC_MAX_ENUM,
    csc_matrix_t*         matrix = NULL);
```

传入参数说明:

- bm_handle_t handle

  输入参数。设备环境句柄，通过调用 bm_dev_request 获取。

- int in_img_num

  输入参数。输入 bm_image 数量。

- bm_image* input

输入参数。输入 bm_image 对象指针，其指向空间的长度由 in_img_num 决定。

- bm_image* output

输出参数。输出 bm_image 对象指针，其指向空间的长度由 in_img_num 和 crop_num_vec 共同决定，即所有输入图片 crop 数量之和。

- int* crop_num_vec = NULL

输入参数。该指针指向对每张输入图片进行 crop 的数量，其指向空间的长度由 in_img_num 决定，如果不使用 crop 功能可填 NULL。

- bmcv_rect_t * crop_rect = NULL

输入参数。具体格式定义如下：

```
typedef struct bmcv_rect {
    int start_x;
    int start_y;
    int crop_w;
    int crop_h;
} bmcv_rect_t;
```

每个输出 bm_image 对象所对应的在输入图像上 crop 的参数，包括起始点 x 坐标、起始点 y 坐标、crop 图像的宽度以及 crop 图像的高度。图像左上顶点作为坐标原点。如果不使用 crop 功能可填 NULL。

- bmcv_padding_atrr_t* padding_attr = NULL

输入参数。所有 crop 的目标小图在 dst image 中的位置信息以及要 pdding 的各通道像素值，若不使用 padding 功能则设置为 NULL。

```
typedef struct bmcv_padding_atrr_s {
    unsigned int  dst_crop_stx;
    unsigned int  dst_crop_sty;
    unsigned int  dst_crop_w;
    unsigned int  dst_crop_h;
    unsigned char padding_r;
    unsigned char padding_g;
    unsigned char padding_b;
    int           if_memset;
} bmcv_padding_atrr_t;
```

1. 目标小图的左上角顶点相对于 dst image 原点（左上角）的 offset 信息：dst_crop_stx 和 dst_crop_sty；
2. 目标小图经 resize 后的宽高：dst_crop_w 和 dst_crop_h；
3. dst image 如果是 RGB 格式，各通道需要 padding 的像素值信息：padding_r、padding_g、padding_b，当 if_memset=1 时有效，如果是 GRAY 图像可以将三个值均设置为同一个值；
4. if_memset 表示要不要在该 api 内部对 dst image 按照各个通道的 padding 值做 memset，仅支持 RGB 和 GRAY 格式的图像。如果设置为 0 则用户需要在调用该 api 前，根据需要 padding 的像素值信息，

调用 bmlib 中的 api 直接对 device memory 进行 memset 操作，如果用户对 padding 的值不关心，可以设置为 0 忽略该步骤。

- bmcv_resize_algorithm algorithm = BMCV_INTER_LINEAR

输入参数。resize 算法选择，包括 BMCV_INTER_NEAREST 和 BMCV_INTER_LINEAR 两种，默认情况下是双线性差值。

- csc_type_t csc_type = CSC_MAX_ENUM

输入参数。color space convert 参数类型选择，填 CSC_MAX_ENUM 则使用默认值，默认为 CSC_YCbCr2RGB_BT601 或者 CSC_RGB2YCbCr_BT601，支持的类型包括：

CSC_YCbCr2RGB_BT601
CSC_YPbPr2RGB_BT601
CSC_RGB2YCbCr_BT601
CSC_YCbCr2RGB_BT709
CSC_RGB2YCbCr_BT709
CSC_RGB2YPbPr_BT601
CSC_YPbPr2RGB_BT709
CSC_RGB2YPbPr_BT709
CSC_USER_DEFINED_MATRIX
CSC_MAX_ENUM

- csc_matrix_t* matrix = NULL

输入参数。如果 csc_type 选择 CSC_USER_DEFINED_MATRIX，则需要传入系数矩阵，格式如下：

```
typedef struct {
    int csc_coe00;
    int csc_coe01;
    int csc_coe02;
    int csc_add0;
    int csc_coe10;
    int csc_coe11;
    int csc_coe12;
    int csc_add1;
    int csc_coe20;
    int csc_coe21;
    int csc_coe22;
    int csc_add2;
} __attribute__((packed)) csc_matrix_t;
```

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项:

1. 该 API 所需要满足的格式以及部分要求, 如下表格所示：

src format	dst format	其他限制
RGB_PACKED	RGB_PACKED	条件 1
	RGB_PLANAR	条件 1
	BGR_PLANAR	条件 1
	BGR_PACKED	条件 1
	RGBP_SEPARATE	条件 1
	BGRP_SEPARATE	条件 1
BGR_PACKED	RGB_PACKED	条件 1
	RGB_PLANAR	条件 1
	BGR_PACKED	条件 1
	BGR_PLANAR	条件 1
	RGBP_SEPARATE	条件 1
	BGRP_SEPARATE	条件 1
RGB_PLANAR	RGB_PACKED	条件 1
	RGB_PLANAR	条件 1
	BGR_PACKED	条件 1
	BGR_PLANAR	条件 1
	RGBP_SEPARATE	条件 1
	BGRP_SEPARATE	条件 1
BGR_PLANAR	RGB_PACKED	条件 1
	RGB_PLANAR	条件 1
	BGR_PACKED	条件 1
	BGR_PLANAR	条件 1
	RGBP_SEPARATE	条件 1
	BGRP_SEPARATE	条件 1
RGBP_SEPARATE	RGB_PACKED	条件 1
	RGB_PLANAR	条件 1
	BGR_PACKED	条件 1
	BGR_PLANAR	条件 1
	RGBP_SEPARATE	条件 1
	BGRP_SEPARATE	条件 1
BGRP_SEPARATE	RGB_PACKED	条件 1
	RGB_PLANAR	条件 1
	BGR_PACKED	条件 1
	BGR_PLANAR	条件 1
	RGBP_SEPARATE	条件 1
	BGRP_SEPARATE	条件 1
GRAY	GRAY	条件 1
YUV420P	YUV420P	条件 2
COMPRESSED	YUV420P	条件 2
RGB_PACKED	YUV420P	条件 3
RGB_PLANAR		条件 3
BGR_PACKED		条件 3

continues on next page

Table 1 – continued from previous page

src format	dst format	其他限制
BGR_PLANAR		条件 3
RGBP_SEPARATE		条件 3
BGRP_SEPARATE		条件 3
YUV420P	RGB_PACKED	条件 4
	RGB_PLANAR	条件 4
	BGR_PACKED	条件 4
	BGR_PLANAR	条件 4
	RGBP_SEPARATE	条件 4
	BGRP_SEPARATE	条件 4
NV12	RGB_PACKED	条件 4
	RGB_PLANAR	条件 4
	BGR_PACKED	条件 4
	BGR_PLANAR	条件 4
	RGBP_SEPARATE	条件 4
	BGRP_SEPARATE	条件 4
COMPRESSED	RGB_PACKED	条件 4
	RGB_PLANAR	条件 4
	BGR_PACKED	条件 4
	BGR_PLANAR	条件 4
	RGBP_SEPARATE	条件 4
	BGRP_SEPARATE	条件 4

其中：

- 条件 1：src.width >= crop.x + crop.width，src.height >= crop.y + crop.height
- 条件 2：src.width, src.height, dst.widht，dst.height 必须是 2 的整数倍，src.width >= crop.x + crop.width，src.height >= crop.y + crop.height
- 条件 3：dst.widht，dst.height 必须是 2 的整数倍，src.width == dst.width，src.height == dst.height，crop.x == 0，crop.y == 0,src.width >= crop.x + crop.width，src.height >= crop.y + crop.height
- 条件 4：src.width，src.height 必须是 2 的整数倍，src.width >= crop.x + crop.width，src.height >= crop.y + crop.height

2. 输入 bm_image 的 device mem 不能在 heap0 上。
3. 所有输入输出 image 的 stride 必须 64 对齐。
4. 所有输入输出 image 的地址必须 32 byte 对齐。
5. 图片缩放倍数（(crop.width / output.width) 以及 (crop.height / output.height)）限制在 1/32 ~ 32 之间。
6. 输入输出的宽高（src.width, src.height, dst.widht, dst.height）限制在 16 ~ 4096 之间。
7. 输入必须关联 device memory，否则返回失败。
8. FORMAT_COMPRESSED 是 VPU 解码后内置的一种压缩格式，它包括 4 个部分：Y compressed table、Y compressed data、CbCr compressed table 以及 CbCr compressed data。请注意 bm_image 中这四部分

存储的顺序与 FFMPEG 中 AVFrame 稍有不同，如果需要 attach AVFrame 中 device memory 数据到 bm_image 中时，对应关系如下，关于 AVFrame 详细内容请参考 VPU 的用户手册。

```
bm_device_mem_t src_plane_device[4];
src_plane_device[0] = bm_mem_from_device((u64)avframe->data[6],
        avframe->linesize[6]);
src_plane_device[1] = bm_mem_from_device((u64)avframe->data[4],
        avframe->linesize[4] * avframe->h);
src_plane_device[2] = bm_mem_from_device((u64)avframe->data[7],
        avframe->linesize[7]);
src_plane_device[3] = bm_mem_from_device((u64)avframe->data[5],
        avframe->linesize[4] * avframe->h / 2);

bm_image_attach(*compressed_image, src_plane_device);
```

4.9 bmcv_image_vpp_convert

该 API 将输入图像格式转化为输出图像格式, 并支持 crop + resize 功能, 支持从 1 张输入中 crop 多张输出并 resize 到输出图片大小。

```
bm_status_t bmcv_image_vpp_convert(
    bm_handle_t  handle,
    int output_num,
    bm_image input,
    bm_image *output,
    bmcv_rect_t *crop_rect,
    bmcv_resize_algorithm algorithm = BMCV_INTER_LINEAR
);
```

传入参数说明:

- bm_handle_t handle

  输入参数。设备环境句柄, 通过调用 bm_dev_request 获取

- int output_num

  输出参数。输出 bm_image 数量，和 src image 的 crop 数量相等, 一个 src crop 输出一个 dst bm_image

- bm_image input

  输入参数。输入 bm_image 对象

- bm_image* output

  输出参数。输出 bm_image 对象指针

- bmcv_rect_t * crop_rect

  输入参数。具体格式定义如下:

```
typedef struct bmcv_rect {
    int start_x;
    int start_y;
    int crop_w;
    int crop_h;
} bmcv_rect_t;
```

每个输出 bm_image 对象所对应的在输入图像上 crop 的参数，包括起始点 x 坐标、起始点 y 坐标、crop 图像的宽度以及 crop 图像的高度。

- bmcv_resize_algorithm algorithm = BMCV_INTER_LINEAR

输入参数。resize 算法选择，包括 BMCV_INTER_NEAREST 和 BMCV_INTER_LINEAR 两种，默认情况下是双线性差值。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项:

1. 该 API 所需要满足的格式以及部分要求与 bmcv_image_vpp_basic 中的表格相同。
2. 输入输出的宽高（src.width, src.height, dst.widht, dst.height）限制在 16 ~ 4096 之间。
3. 输入必须关联 device memory，否则返回失败。
4. FORMAT_COMPRESSED 是 VPU 解码后内置的一种压缩格式，它包括 4 个部分：Y compressed table、Y compressed data、CbCr compressed table 以及 CbCr compressed data。请注意 bm_image 中这四部分存储的顺序与 FFMPEG 中 AVFrame 稍有不同，如果需要 attach AVFrame 中 device memory 数据到 bm_image 中时，对应关系如下，关于 AVFrame 详细内容请参考 VPU 的用户手册。

```
bm_device_mem_t src_plane_device[4];
src_plane_device[0] = bm_mem_from_device((u64)avframe->data[6],
        avframe->linesize[6]);
src_plane_device[1] = bm_mem_from_device((u64)avframe->data[4],
        avframe->linesize[4] * avframe->h);
src_plane_device[2] = bm_mem_from_device((u64)avframe->data[7],
        avframe->linesize[7]);
src_plane_device[3] = bm_mem_from_device((u64)avframe->data[5],
        avframe->linesize[4] * avframe->h / 2);

bm_image_attach(*compressed_image, src_plane_device);
```

代码示例:

```
#include <iostream>
#include <vector>
#include "bmcv_api_ext.h"
#include "bmlib_utils.h"
```

(continues on next page)

(continued from previous page)

```
#include "common.h"
#include <memory>
#include "stdio.h"
#include "stdlib.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    bm_handle_t handle;
    int             image_h     = 1080;
    int             image_w     = 1920;
    bm_image        src, dst[4];
    bm_dev_request(&handle, 0);
    bm_image_create(handle, image_h, image_w, FORMAT_NV12,
            DATA_TYPE_EXT_1N_BYTE, &src);
    bm_image_alloc_dev_mem(src, 1);
    for (int i = 0; i < 4; i++) {
        bm_image_create(handle,
            image_h / 2,
            image_w / 2,
            FORMAT_BGR_PACKED,
            DATA_TYPE_EXT_1N_BYTE,
            dst + i);
        bm_image_alloc_dev_mem(dst[i]);
    }
    std::unique_ptr<u8 []> y_ptr(new u8[image_h * image_w]);
    std::unique_ptr<u8 []> uv_ptr(new u8[image_h * image_w / 2]);
    memset((void *)(y_ptr.get()), 148, image_h * image_w);
    memset((void *)(uv_ptr.get()), 158, image_h * image_w / 2);
    u8 *host_ptr[] = {y_ptr.get(), uv_ptr.get()};
    bm_image_copy_host_to_device(src, (void **)host_ptr);

    bmcv_rect_t rect[] = {{0, 0, image_w / 2, image_h / 2},
            {0, image_h / 2, image_w / 2, image_h / 2},
            {image_w / 2, 0, image_w / 2, image_h / 2},
            {image_w / 2, image_h / 2, image_w / 2, image_h / 2}};

    bmcv_image_vpp_convert(handle, 4, src, dst, rect);

    for (int i = 0; i < 4; i++) {
        bm_image_destroy(dst[i]);
    }

    bm_image_destroy(src);
    bm_dev_free(handle);
    return 0;
}
```

4.10 bmcv_image_vpp_convert_padding

使用 vpp 硬件资源，结合对 dst image 做 memset 操作，实现图像 padding 的效果。这个效果的实现是利用了 vpp 的 dst crop 的功能，通俗的讲是将一张小图填充到大图中。可以从一张 src image 上 crop 多个目标图像，对于每一个目标小图，可以一次性完成 csc+resize 操作，然后根据其在大图中的 offset 信息，填充到大图中。一次 crop 的数量不能超过 256。

```
bm_status_t bmcv_image_vpp_convert_padding(
    bm_handle_t           handle,
    int                   output_num,
    bm_image              input,
    bm_image *            output,
    bmcv_padding_atrr_t * padding_attr,
    bmcv_rect_t *         crop_rect = NULL,
    bmcv_resize_algorithm algorithm = BMCV_INTER_LINEAR);
```

传入参数说明:

- bm_handle_t handle

  输入参数。设备环境句柄, 通过调用 bm_dev_request 获取

- int output_num

  输出参数。输出 bm_image 数量，和 src image 的 crop 数量相等, 一个 src crop 输出一个 dst bm_image

- bm_image input

  输入参数。输入 bm_image 对象

- bm_image* output

  输出参数。输出 bm_image 对象指针

- bmcv_padding_atrr_t * padding_attr

  输入参数。src crop 的目标小图在 dst image 中的位置信息以及要 pdding 的各通道像素值

```
typedef struct bmcv_padding_atrr_s {
    unsigned int    dst_crop_stx;
    unsigned int    dst_crop_sty;
    unsigned int    dst_crop_w;
    unsigned int    dst_crop_h;
    unsigned char padding_r;
    unsigned char padding_g;
    unsigned char padding_b;
    int           if_memset;
} bmcv_padding_atrr_t;
```

1. 目标小图的左上角顶点相对于 dst image 原点（左上角）的 offset 信息：dst_crop_stx 和 dst_crop_sty；
2. 目标小图经 resize 后的宽高：dst_crop_w 和 dst_crop_h；

3. dst image 如果是 RGB 格式，各通道需要 padding 的像素值信息：padding_r、padding_g、padding_b，当 if_memset=1 时有效，如果是 GRAY 图像可以将三个值均设置为同一个值；

4. if_memset 表示要不要在该 api 内部对 dst image 按照各个通道的 padding 值做 memset，仅支持 RGB 和 GRAY 格式的图像。如果设置为 0 则用户需要在调用该 api 前，根据需要 padding 的像素值信息，调用 bmlib 中的 api 直接对 device memory 进行 memset 操作，如果用户对 padding 的值不关心，可以设置为 0 忽略该步骤。

- bmcv_rect_t * crop_rect

输入参数。在 src image 上的各个目标小图的坐标和宽高信息

具体格式定义如下：

```
typedef struct bmcv_rect {
    int start_x;
    int start_y;
    int crop_w;
    int crop_h;
} bmcv_rect_t;
```

- bmcv_resize_algorithm algorithm

输入参数。resize 算法选择，包括 BMCV_INTER_NEAREST 和 BMCV_INTER_LINEAR 两种，默认情况下是双线性差值

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项:

1. 该 API 的 dst image 的格式仅支持:FORMAT_RGB_PLANAR、FORMAT_BGR_PLANAR、FORMAT_RGBP_SEPARATE、FORMAT_BGRP_SEPARATE、FORMAT_RGB_PACKED 和 FORMAT_BGR_PACKED。

2. 在调用该 api 前，需要根据各通道 padding 的像素值对 dst image 的 device memory 进行 memset 操作，并将 if_memset 置 0，否则，padding 的部分是内存中残留数据。

3. 该 API 所需要满足的格式以及部分要求与 bmcv_image_vpp_basic 一致。

4.11 bmcv_image_vpp_stitch

使用 vpp 硬件资源的 crop 功能，实现图像拼接的效果，对输入 image 可以一次完成 src crop + csc + resize + dst crop 操作。dst image 中拼接的小图像数量不能超过 256。

```
bm_status_t bmcv_image_vpp_stitch(
    bm_handle_t           handle,
    int                   input_num,
    bm_image*             input,
```

(continues on next page)

(continued from previous page)

```
    bm_image                output,
    bmcv_rect_t*            dst_crop_rect,
    bmcv_rect_t*            src_crop_rect = NULL,
    bmcv_resize_algorithm algorithm = BMCV_INTER_LINEAR);
```

传入参数说明:

- bm_handle_t handle

  输入参数。设备环境句柄, 通过调用 bm_dev_request 获取

- int input_num

  输入参数。输入 bm_image 数量

- bm_imagei* input

  输入参数。输入 bm_image 对象指针

- bm_image output

  输出参数。输出 bm_image 对象

- bmcv_rect_t * dst_crop_rect

  输入参数。在 dst images 上，各个目标小图的坐标和宽高信息

- bmcv_rect_t * src_crop_rect

  输入参数。在 src image 上，各个目标小图的坐标和宽高信息

  具体格式定义如下:

```
typedef struct bmcv_rect {
     int start_x;
     int start_y;
     int crop_w;
     int crop_h;
 } bmcv_rect_t;
```

- bmcv_resize_algorithm algorithm

  输入参数。resize 算法选择，包括 BMCV_INTER_NEAREST 和 BMCV_INTER_LINEAR 两种，默认情况下是双线性差值。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项:

1. 该 API 的 src image 不支持压缩格式的数据。
2. 该 API 所需要满足的格式以及部分要求与 bmcv_image_vpp_basic 一致。
3. 如果对 src image 做 crop 操作，一张 src image 只 crop 一个目标。

4.12 bmcv_image_vpp_csc_matrix_convert

默认情况下，bmcv_image_vpp_convert 使用的是 BT_609 标准进行色域转换。有些情况下需要使用其他标准，或者用户自定义 csc 参数。

```
bm_status_t bmcv_image_vpp_csc_matrix_convert(
    bm_handle_t  handle,
    int output_num,
    bm_image input,
    bm_image *output,
    csc_type_t csc,
    csc_matrix_t * matrix = nullptr,
    bmcv_resize_algorithm algorithm = BMCV_INTER_LINEAR);
```

传入参数说明:

- bm_handle_t handle

  输入参数。设备环境句柄, 通过调用 bm_dev_request 获取

- int image_num

  输入参数。输入 bm_image 数量

- bm_image input

  输入参数。输入 bm_image 对象

- bm_image* output

  输出参数。输出 bm_image 对象指针

- csc_type_t csc

  输入参数。色域转换枚举类型，目前可选:

```
typedef enum csc_type {
    CSC_YCbCr2RGB_BT601 = 0,
    CSC_YPbPr2RGB_BT601,
    CSC_RGB2YCbCr_BT601,
    CSC_YCbCr2RGB_BT709,
    CSC_RGB2YCbCr_BT709,
    CSC_USER_DEFINED_MATRIX = 1000,
    CSC_MAX_ENUM
} csc_type_t;
```

- csc_matrix_t * matrix

  输入参数。色域转换自定义矩阵，当且仅当 csc 为 CSC_USER_DEFINED_MATRIX 时这个值才生效。

  具体格式定义如下:

```
typedef struct {
    int csc_coe00;
```

(continues on next page)

(continued from previous page)

```
    int csc_coe01;
    int csc_coe02;
    int csc_add0;
    int csc_coe10;
    int csc_coe11;
    int csc_coe12;
    int csc_add1;
    int csc_coe20;
    int csc_coe21;
    int csc_coe22;
    int csc_add2;
} __attribute__((packed)) csc_matrix_t;
```

$$\begin{cases} dst_0 = (csc\_coe_{00} * src_0 + csc\_coe_{01} * src_1 + csc\_coe_{02} * src_2 + csc\_add_0) >> 10 \\ dst_1 = (csc\_coe_{10} * src_0 + csc\_coe_{11} * src_1 + csc\_coe_{12} * src_2 + csc\_add_1) >> 10 \\ dst_2 = (csc\_coe_{20} * src_0 + csc\_coe_{21} * src_1 + csc\_coe_{22} * src_2 + csc\_add_2) >> 10 \end{cases}$$

- bmcv_resize_algorithm algorithm

输入参数。resize 算法选择，包括 BMCV_INTER_NEAREST 和 BMCV_INTER_LINEAR 两种，默认情况下是双线性差值。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项:

1. 该 API 所需要满足的格式以及部分要求与 vpp_convert 一致
2. 如果色域转换枚举类型与 input 和 output 格式不对应，如 csc == CSC_YCbCr2RGB_BT601, 而 input image_format 为 RGB 格式，则返回失败。
3. 如果 csc == CSC_USER_DEFINED_MATRIX 而 matrix 为 nullptr，则返回失败。

代码示例:

```
#include <iostream>
#include <vector>
#include "bmcv_api_ext.h"
#include "bmlib_utils.h"
#include "common.h"
#include <memory>
#include "stdio.h"
#include "stdlib.h"
#include <stdio.h>
#include <stdlib.h>
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char *argv[]) {
    bm_handle_t handle;
    int             image_h     = 1080;
    int             image_w     = 1920;
    bm_image        src, dst[4];
    bm_dev_request(&handle, 0);
    bm_image_create(handle, image_h, image_w, FORMAT_NV12,
            DATA_TYPE_EXT_1N_BYTE, &src);
    bm_image_alloc_dev_mem(src, 1);
    for (int i = 0; i < 4; i++) {
        bm_image_create(handle,
            image_h / 2,
            image_w / 2,
            FORMAT_BGR_PACKED,
            DATA_TYPE_EXT_1N_BYTE,
            dst + i);
        bm_image_alloc_dev_mem(dst[i]);
    }
    std::unique_ptr<u8 []> y_ptr(new u8[image_h * image_w]);
    std::unique_ptr<u8 []> uv_ptr(new u8[image_h * image_w / 2]);
    memset((void *)(y_ptr.get()), 148, image_h * image_w);
    memset((void *)(uv_ptr.get()), 158, image_h * image_w / 2);
    u8 *host_ptr[] = {y_ptr.get(), uv_ptr.get()};
    bm_image_copy_host_to_device(src, (void **)host_ptr);

    bmcv_rect_t rect[] = {{0, 0, image_w / 2, image_h / 2},
            {0, image_h / 2, image_w / 2, image_h / 2},
            {image_w / 2, 0, image_w / 2, image_h / 2},
            {image_w / 2, image_h / 2, image_w / 2, image_h / 2}};

    bmcv_image_vpp_csc_matrix_convert(handle, 4, src, dst, CSC_YCbCr2RGB_
↪BT601);

    for (int i = 0; i < 4; i++) {
        bm_image_destroy(dst[i]);
    }

    bm_image_destroy(src);
    bm_dev_free(handle);
    return 0;
}
```

4.13 bmcv_image_jpeg_enc

该接口可以实现对多张 bm_image 的 JPEG 编码过程。

接口形式:

```
bm_status_t bmcv_image_jpeg_enc(
        bm_handle_t handle,
        int          image_num,
        bm_image *   src,
        void *       p_jpeg_data[],
        size_t *     out_size,
        int          quality_factor = 85
);
```

输入参数说明:

- bm_handle_t handle

输入参数。bm_handle 句柄。

- int image_num

输入参数。输入图片数量，最多支持 4。

- bm_image* src

输入参数。输入 bm_image 的指针。每个 bm_image 需要外部调用 bmcv_image_create 创建，image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存, 或者使用 bmcv_image_attach 来 attach 已有的内存。

- void * p_jpeg_data,

输出参数。编码后图片的数据指针，由于该接口支持对多张图片的编码，因此为指针数组，数组的大小即为 image_num。用户可以选择不为其申请空间（即数组每个元素均为 NULL），在 api 内部会根据编码后数据的大小自动分配空间，但当不再使用时需要用户手动释放该空间。当然用户也可以选择自己申请足够的空间。

- size_t *out_size,

输出参数。完成编码后各张图片的大小（以 byte 为单位）存放在该指针中。

- int quality_factor = 85

输入参数。编码后图片的质量因子。取值 0 ~ 100 之间，值越大表示图片质量越高，但数据量也就越大，反之值越小图片质量越低，数据量也就越少。该参数为可选参数，默认值为 85。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

Note:

目前编码支持的图片格式包括以下几种：

FORMAT_YUV420P

FORMAT_YUV422P

FORMAT_YUV444P

FORMAT_NV12

FORMAT_NV21

FORMAT_NV16

FORMAT_NV61

FORMAT_GRAY

示例代码

```
int image_h     = 1080;
int image_w     = 1920;
int size        = image_h * image_w;
int format      = FORMAT_YUV420P;
bm_image src;
bm_image_create(handle, image_h, image_w, (bm_image_format_ext)format,
        DATA_TYPE_EXT_1N_BYTE, &src);
std::unique_ptr<unsigned char[]> buf1(new unsigned char[size]);
memset(buf1.get(), 0x11, size);

std::unique_ptr<unsigned char[]> buf2(new unsigned char[size / 4]);
memset(buf2.get(), 0x22, size / 4);

std::unique_ptr<unsigned char[]> buf3(new unsigned char[size / 4]);
memset(buf3.get(), 0x33, size / 4);

unsigned char *buf[] = {buf1.get(), buf2.get(), buf3.get()};
bm_image_copy_host_to_device(src, (void **)buf);

void* jpeg_data = NULL;
size_t out_size = 0;
int ret = bmcv_image_jpeg_enc(handle, 1, &src, &jpeg_data, &out_size);
if (ret == BM_SUCCESS) {
    FILE *fp = fopen("test.jpg", "wb");
    fwrite(jpeg_data, out_size, 1, fp);
    fclose(fp);
}
free(jpeg_data);
bm_image_destroy(src);
```

4.14 bmcv_image_jpeg_dec

该接口可以实现对多张图片的 JPEG 解码过程。

接口形式：

```
bm_status_t bmcv_image_jpeg_dec(
        bm_handle_t handle,
        void *      p_jpeg_data[],
        size_t *    in_size,
        int         image_num,
        bm_image *  dst
);
```

输入参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- void * p_jpeg_data[]

输入参数。待解码的图片数据指针，由于该接口支持对多张图片的解码，因此为指针数组。

- size_t *in_size

输入参数。待解码各张图片的大小（以 byte 为单位）存放在该指针中，也就是上述 p_jpeg_data 每一维指针所指向空间的大小。

- int image_num

输入参数。输入图片数量，最多支持 4

- bm_image* dst

输出参数。输出 bm_image 的指针。每个 dst bm_image 用户可以选择自行调用 bm_image_create 创建，也可以选择不创建。如果用户只声明而不创建则由接口内部根据待解码图片信息自动创建，默认的 format 如下表所示，当不再需要时仍然需要用户调用 bm_image_destory 来销毁。

码流	默认输出 format
YUV420	FORMAT_YUV420P
YUV422	FORMAT_YUV422P
YUV444	FORMAT_YUV444P
YUV400	FORMAT_GRAY

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

注意事项：

1. 如果用户没有使用 bmcv_image_create 创建 dst 的 bm_image，那么需要将参数传入指针所指向的空间置 0。
2. 目前解码支持的图片格式及其输出格式对应如下，如果用户需要指定以下某一种输出格式，可通过使用 bmcv_image_create 自行创建 dst bm_image，从而实现将图片解码到以下对应的某一格式。

码流	输出 format
YUV420	FORMAT_YUV420P
	FORMAT_NV12
	FORMAT_NV21
YUV422	FORMAT_YUV422P
	FORMAT_NV16
	FORMAT_NV61
YUV444	FORMAT_YUV444P
YUV400	FORMAT_GRAY

示例代码

```
size_t size = 0;
// read input from picture
FILE *fp = fopen(filename, "rb+");
assert(fp != NULL);
fseek(fp, 0, SEEK_END);
*size = ftell(fp);
u8* jpeg_data = (u8*)malloc(*size);
fseek(fp, 0, SEEK_SET);
fread(jpeg_data, *size, 1, fp);
fclose(fp);

// create bm_image used to save output
bm_image dst;
memset((char*)&dst, 0, sizeof(bm_image));
// if you not create dst bm_image it will create automatically inside.
// you can also create dst bm_image here, like this:
// bm_image_create(handle, IMAGE_H, IMAGE_W, FORMAT_YUV420P,
//         DATA_TYPE_EXT_1N_BYTE, &dst);

// decode input
int ret = bmcv_image_jpeg_dec(handle, (void**)&jpeg_data, &size, 1, &dst);
free(jpeg_data);
bm_image_destory(dst);
```

4.15 bmcv_image_copy_to

该接口实现将一幅图像拷贝到目的图像的对应内存区域。

接口形式：

```
bm_status_t bmcv_image_copy_to(
        bm_handle_t handle,
        bmcv_copy_to_atrr_t copy_to_attr,
        bm_image            input,
        bm_image            output
);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bmcv_copy_to_atrr_t copy_to_attr

输入参数。api 所对应的属性配置。

- bm_image input

输入参数。输入 bm_image，bm_image 需要外部调用 bmcv_image _create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image output

输出参数。输出 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

数据类型说明：

```
typedef struct bmcv_copy_to_atrr_s {
    int           start_x;
    int           start_y;
    unsigned char padding_r;
    unsigned char padding_g;
    unsigned char padding_b;
    int if_padding;
} bmcv_copy_to_atrr_t;
```

- padding_b 表示当 input 的图像要小于输出图像的情况下，多出来的图像 b 通道上被填充的值。
- padding_r 表示当 input 的图像要小于输出图像的情况下，多出来的图像 r 通道上被填充的值。

- padding_g 表示当 input 的图像要小于输出图像的情况下，多出来的图像 g 通道上被填充的值。
- start_x 描述了 copy_to 拷贝到输出图像所在的起始横坐标。
- start_y 描述了 copy_to 拷贝到输出图像所在的起始纵坐标。
- if_padding 表示当 input 的图像要小于输出图像的情况下，是否需要对多余的图像区域填充特定颜色，0 表示不需要，1 表示需要。当该值填 0 时，padding_r，padding_g，padding_b 的设置将无效

代码示例：

```
int channel   = 3;
int in_w      = 400;
int in_h      = 400;
int out_w     = 800;
int out_h     = 800;
int    dev_id = 0;
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<unsigned char> src_ptr(
        new unsigned char[channel * in_w * in_h],
        std::default_delete<unsigned char[]>());
std::shared_ptr<unsigned char> res_ptr(
        new unsigned char[channel * out_w * out_h],
        std::default_delete<unsigned char[]>());
unsigned char * src_data = src_ptr.get();
unsigned char * res_data = res_ptr.get();
for (int i = 0; i < channel * in_w * in_h; i++) {
    src_data[i] = rand() % 255;
}
// calculate res
bmcv_copy_to_atrr_t copy_to_attr;
copy_to_attr.start_x   = 0;
copy_to_attr.start_y   = 0;
copy_to_attr.padding_r = 0;
copy_to_attr.padding_g = 0;
copy_to_attr.padding_b = 0;
bm_image input, output;
bm_image_create(handle,
        in_h,
        in_w,
        FORMAT_RGB_PLANAR,
        DATA_TYPE_EXT_1N_BYTE,
        &input);
bm_image_alloc_dev_mem(input);
bm_image_copy_host_to_device(input, (void **)&src_data);
bm_image_create(handle,
        out_h,
        out_w,
        FORMAT_RGB_PLANAR,
```

(continues on next page)

(continued from previous page)

```
        DATA_TYPE_EXT_1N_BYTE,
        &output);
bm_image_alloc_dev_mem(output);
if (BM_SUCCESS != bmcv_image_copy_to(handle, copy_to_attr, input,␣
↪output)) {
    std::cout << "bmcv_copy_to error !!!" << std::endl;
    bm_image_destroy(input);
    bm_image_destroy(output);
    bm_dev_free(handle);

    exit(-1);
}
bm_image_copy_device_to_host(output, (void **)&res_data);
bm_image_destroy(input);
bm_image_destroy(output);
bm_dev_free(handle);
```

格式支持：

copyTo 目前支持以下 image_format 和 data_type 的组合:

num	image_format	data_type
1	FORMAT_BGR_PACKED	DATA_TYPE_EXT_FLOAT32
2	FORMAT_BGR_PLANAR	DATA_TYPE_EXT_FLOAT32
3	FORMAT_BGR_PACKED	DATA_TYPE_EXT_1N_BYTE
4	FORMAT_BGR_PLANAR	DATA_TYPE_EXT_1N_BYTE
5	FORMAT_BGR_PLANAR	DATA_TYPE_EXT_4N_BYTE
6	FORMAT_GRAY	DATA_TYPE_EXT_1N_BYTE

注意事项：

1、在调用 bmcv_image_copy_to() 之前必须确保输入的 image 内存已经申请。

2、input output 的 data_type，image_format 必须相同。

3、为了避免内存越界，输入图像 width + start_x 必须小于等于输出图像 width stride。

4.16 bmcv_image_draw_lines

可以实现在一张图像上画一条或多条线段，从而可以实现画多边形的功能，并支持指定线的颜色和线的宽度。

接口形式：

```
typedef struct {
    int x;
    int y;
```

(continues on next page)

(continued from previous page)

```
} bmcv_point_t;

typedef struct {
    unsigned char r;
    unsigned char g;
    unsigned char b;
} bmcv_color_t;

bm_status_t bmcv_image_draw_lines(
        bm_handle_t handle,
        bm_image img,
        const bmcv_point_t* start,
        const bmcv_point_t* end,
        int line_num,
        bmcv_color_t color,
        int thickness);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_image img

输入/输出参数。需处理图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- const bmcv_point_t* start

输入参数。线段起始点的坐标指针，指向的数据长度由 line_num 参数决定。图像左上角为原点，向右延伸为 x 方向，向下延伸为 y 方向。

- const bmcv_point_t* end

输入参数。线段结束点的坐标指针，指向的数据长度由 line_num 参数决定。图像左上角为原点，向右延伸为 x 方向，向下延伸为 y 方向。

- int line_num

输入参数。需要画线的数量。

- bmcv_color_t color

输入参数。画线的颜色，分别为 RGB 三个通道的值。

- int thickness

输入参数。画线的宽度，对于 YUV 格式的图像建议设置为偶数。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

格式支持：

该接口目前支持以下 image_format:

num	image_format
1	FORMAT_GRAY
2	FORMAT_YUV420P
3	FORMAT_YUV422P
4	FORMAT_YUV444P
5	FORMAT_NV12
6	FORMAT_NV21
7	FORMAT_NV16
8	FORMAT_NV61

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

代码示例：

```
int channel   = 1;
int width     = 1920;
int height    = 1080;
int dev_id    = 0;
int thickness = 4
bmcv_point_t start = {0, 0};
bmcv_point_t end = {100, 100};
bmcv_color_t color = {255, 0, 0};
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<unsigned char> data_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
for (int i = 0; i < channel * width * height; i++) {
    data_ptr.get()[i] = rand() % 255;
}
// calculate res
bm_image img;
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &img);
bm_image_alloc_dev_mem(img);
bm_image_copy_host_to_device(img, (void **)&(data_ptr.get()));
```

(continues on next page)

(continued from previous page)

```
if (BM_SUCCESS != bmcv_image_draw_lines(handle, img, &start, &end, 1,␣
↪color, thickness)) {
    std::cout << "bmcv draw lines error !!!" << std::endl;
    bm_image_destroy(img);
    bm_dev_free(handle);
    return;
}
bm_image_copy_device_to_host(img, (void **)&(data_ptr.get()));
bm_image_destroy(img);
bm_dev_free(handle);
```

4.17 bmcv_image_draw_rectangle

该接口用于在图像上画一个或多个矩形框。

接口形式：

```
bm_status_t bmcv_image_draw_rectangle(
        bm_handle_t   handle,
        bm_image      image,
        int           rect_num,
        bmcv_rect_t * rects,
        int           line_width,
        unsigned char r,
        unsigned char g,
        unsigned char b)
```

传入参数说明：

- bm_handle_t handle

  输入参数。设备环境句柄，通过调用 bm_dev_request 获取。

- bm_image image

  输入参数。需要在其上画矩形框的 bm_image 对象。

- int rect_num

  输入参数。矩形框数量，指 rects 指针中所包含的 bmcv_rect_t 对象个数。

- bmcv_rect_t* rect

  输入参数。矩形框对象指针，包含矩形起始点和宽高。具体内容参考下面的数据类型说明。

- int line_width

  输入参数。矩形框线宽。

- unsigned char r

  输入参数。矩形框颜色的 r 分量。

- unsigned char g

输入参数。矩形框颜色的 g 分量。

- unsigned char b

输入参数。矩形框颜色的 g 分量。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

数据类型说明:

```
typedef struct bmcv_rect {
    int start_x;
    int start_y;
    int crop_w;
    int crop_h;
} bmcv_rect_t;
```

- start_x 描述了 crop 图像在原图中所在的起始横坐标，自左而右从 0 开始，取值范围 [0, width)。
- start_y 描述了 crop 图像在原图中所在的起始纵坐标。自上而下从 0 开始，取值范围 [0, height)。
- crop_w 描述的 crop 图像的宽度，也就是对应输出图像的宽度。
- crop_h 描述的 crop 图像的高度，也就是对应输出图像的高度。

注意事项:

1. 该 API 输入 NV12 / NV21 / NV16 / NV61 / YUV420P / RGB_PLANAR / RGB_PACKED / BGR_PLANAR / BGR_PACKED 格式的 image 对象，并在对应的 device memory 上直接画框，没有额外的内存申请和 copy。
2. 目前该 API 支持输入 bm_image 图像格式为
    - FORMAT_NV12
    - FORMAT_NV21
    - FORMAT_NV16
    - FORMAT_NV61
    - FORMAT_YUV420P
    - RGB_PLANAR
    - RGB_PACKED
    - BGR_PLANAR
    - BGR_PACKED

支持输入 bm_image 数据格式为

- DATA_TYPE_EXT_1N_BYTE

如果不满足输入输出格式要求，则返回失败。

3. 输入输出所有 bm_image 结构必须提前创建，否则返回失败。
4. 如果 image 为 NV12/NV21/NV16/NV61/YUV420P 格式，则线宽 line_width 会自动偶数对齐。
5. 如果 rect_num 为 0，则自动返回成功。
6. 如果 line_width 小于零，则返回失败。
7. 所有输入矩形对象部分在 image 之外，则只会画出在 image 之内的线条，并返回成功。

代码示例

```
#include <iostream>
#include <vector>
#include "bmcv_api_ext.h"
#include "bmlib_utils.h"
#include "common.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include <memory>

int main(int argc, char *argv[]) {
    bm_handle_t handle;
    bm_dev_request(&handle, 0);

    int image_h = 1080;
    int image_w = 1920;
    bm_image src;
    bm_image_create(handle, image_h, image_w, FORMAT_NV12,
            DATA_TYPE_EXT_1N_BYTE, &src);
    std::shared_ptr<u8*> y_ptr = std::make_shared<u8*>(
            new u8[image_h * image_w]);
    memset((void *)(*y_ptr.get()), 148, image_h * image_w);
    memset((void *)(*uv_ptr.get()), 158, image_h * image_w / 2);
    u8 *host_ptr[] = {*y_ptr.get(), *uv_ptr.get()};
    bm_image_copy_host_to_device(src, (void **)host_ptr);
    bmcv_rect_t rect;
    rect.start_x = 100;
    rect.start_y = 100;
    rect.crop_w = 200;
    rect.crop_h = 300;
    bmcv_image_draw_rectangle(handle, src, 1, &rect, 3, 255, 0, 0);
    bm_image_destroy(src);
    bm_dev_free(handle);
    return 0;
}
```

4.18 bmcv_image_put_text

可以实现在一张图像上写字的功能（英文），并支持指定字的颜色、大小和宽度。

接口形式：

```
typedef struct {
    int x;
    int y;
} bmcv_point_t;

typedef struct {
    unsigned char r;
    unsigned char g;
    unsigned char b;
} bmcv_color_t;

bm_status_t bmcv_image_put_text(
        bm_handle_t handle,
        bm_image image,
        const char* text,
        bmcv_point_t org,
        bmcv_color_t color,
        float fontScale,
        int thickness);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_image image

输入/输出参数。需处理图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- const char* text

输入参数。待写入的文本内容，目前仅支持英文。

- bmcv_point_t org

输入参数。第一个字符左下角的坐标位置。图像左上角为原点，向右延伸为 x 方向，向下延伸为 y 方向。

- bmcv_color_t color

输入参数。画线的颜色，分别为 RGB 三个通道的值。

- float fontScale

输入参数。字体大小。

- int thickness

输入参数。画线的宽度，对于 YUV 格式的图像建议设置为偶数。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

格式支持：

该接口目前支持以下 image_format:

num	image_format
1	FORMAT_GRAY
2	FORMAT_YUV420P
3	FORMAT_YUV422P
4	FORMAT_YUV444P
5	FORMAT_NV12
6	FORMAT_NV21
7	FORMAT_NV16
8	FORMAT_NV61

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

代码示例：

```
int channel   = 1;
int width     = 1920;
int height    = 1080;
int dev_id    = 0;
int thickness = 4
float fontScale = 4;
char text[20] = "hello world";
bmcv_point_t org = {100, 100};
bmcv_color_t color = {255, 0, 0};
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<unsigned char> data_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
for (int i = 0; i < channel * width * height; i++) {
    data_ptr.get()[i] = rand() % 255;
}
// calculate res
```

(continues on next page)

(continued from previous page)

```
bm_image img;
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &img);
bm_image_alloc_dev_mem(img);
bm_image_copy_host_to_device(img, (void **)&(data_ptr.get()));
if (BM_SUCCESS != bmcv_image_put_text(handle, img, text, org, color,␣
↪fontScale, thickness)) {
    std::cout << "bmcv put text error !!!" << std::endl;
    bm_image_destroy(img);
    bm_dev_free(handle);
    return;
}
bm_image_copy_device_to_host(img, (void **)&(data_ptr.get()));
bm_image_destroy(img);
bm_dev_free(handle);
```

4.19 bmcv_image_fill_rectangle

该接口用于在图像上填充一个或者多个矩形。

接口形式：

```
bm_status_t bmcv_image_fill_rectangle(
        bm_handle_t   handle,
        bm_image      image,
        int           rect_num,
        bmcv_rect_t * rects,
        unsigned char r,
        unsigned char g,
        unsigned char b)
```

传入参数说明：

- bm_handle_t handle

  输入参数。设备环境句柄，通过调用 bm_dev_request 获取。

- bm_image image

  输入参数。需要在其上填充矩形的 bm_image 对象。

- int rect_num

  输入参数。需填充矩形的数量，指 rects 指针中所包含的 bmcv_rect_t 对象个数。

- bmcv_rect_t* rect

输入参数。矩形对象指针，包含矩形起始点和宽高。具体内容参考下面的数据类型说明。

- unsigned char r

输入参数。矩形填充颜色的 r 分量。

- unsigned char g

输入参数。矩形填充颜色的 g 分量。

- unsigned char b

输入参数。矩形填充颜色的 b 分量。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

数据类型说明:

```
typedef struct bmcv_rect {
    int start_x;
    int start_y;
    int crop_w;
    int crop_h;
} bmcv_rect_t;
```

- start_x 描述了 crop 图像在原图中所在的起始横坐标，自左而右从 0 开始，取值范围 [0, width)。
- start_y 描述了 crop 图像在原图中所在的起始纵坐标。自上而下从 0 开始，取值范围 [0, height)。
- crop_w 描述的 crop 图像的宽度，也就是对应输出图像的宽度。
- crop_h 描述的 crop 图像的高度，也就是对应输出图像的高度。

注意事项:

1. 该 API 输入 NV12 / NV21 / NV16 / NV61 / YUV420P / RGB_PLANAR / RGB_PACKED / BGR_PLANAR / BGR_PACKED 格式的 image 对象，并在对应的 device memory 上直接填充，没有额外的内存申请和 copy。
2. 目前该 API 支持输入 bm_image 图像格式为
    - FORMAT_NV12
    - FORMAT_NV21
    - FORMAT_NV16
    - FORMAT_NV61
    - FORMAT_YUV420P
    - RGB_PLANAR
    - RGB_PACKED
    - BGR_PLANAR

- BGR_PACKED

支持输入 bm_image 数据格式为

- DATA_TYPE_EXT_1N_BYTE

如果不满足输入输出格式要求，则返回失败。

3. 输入输出所有 bm_image 结构必须提前创建，否则返回失败。
4. 如果 rect_num 为 0，则自动返回成功。
5. 所有输入矩形对象部分在 image 之外，则只会填充在 image 之内的部分，并返回成功。

代码示例

```
#include <iostream>
#include <vector>
#include "bmcv_api_ext.h"
#include "bmlib_utils.h"
#include "common.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include <memory>

int main(int argc, char *argv[]) {
    bm_handle_t handle;
    bm_dev_request(&handle, 0);

    int image_h = 1080;
    int image_w = 1920;
    bm_image src;
    bm_image_create(handle, image_h, image_w, FORMAT_NV12,
            DATA_TYPE_EXT_1N_BYTE, &src);
    std::shared_ptr<u8*> y_ptr = std::make_shared<u8*>(
            new u8[image_h * image_w]);
    memset((void *)(*y_ptr.get()), 148, image_h * image_w);
    memset((void *)(*uv_ptr.get()), 158, image_h * image_w / 2);
    u8 *host_ptr[] = {*y_ptr.get(), *uv_ptr.get()};
    bm_image_copy_host_to_device(src, (void **)host_ptr);
    bmcv_rect_t rect;
    rect.start_x = 100;
    rect.start_y = 100;
    rect.crop_w = 200;
    rect.crop_h = 300;
    bmcv_image_fill_rectangle(handle, src, 1, &rect, 255, 0, 0);
    bm_image_destroy(src);
    bm_dev_free(handle);
    return 0;
}
```

4.20 bmcv_image_absdiff

两张大小相同的图片对应像素值相减并取绝对值。

接口形式：

```
bm_status_t bmcv_image_absdiff(
        bm_handle_t handle,
        bm_image input1,
        bm_image input2,
        bm_image output);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_image input1

输入参数。输入第一张图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image input2

输入参数。输入第二张图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image output

输出参数。输出 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

格式支持：

该接口目前支持以下 image_format:

num	image_format
1	FORMAT_BGR_PACKED
2	FORMAT_BGR_PLANAR
3	FORMAT_RGB_PACKED
4	FORMAT_RGB_PLANAR
5	FORMAT_RGBP_SEPARATE
6	FORMAT_BGRP_SEPARATE
7	FORMAT_GRAY
8	FORMAT_YUV420P
9	FORMAT_YUV422P
10	FORMAT_YUV444P
11	FORMAT_NV12
12	FORMAT_NV21
13	FORMAT_NV16
14	FORMAT_NV61
15	FORMAT_NV24

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

注意事项：

1、在调用 bmcv_image_absdiff() 之前必须确保输入的 image 内存已经申请。

2、input output 的 data_type，image_format 必须相同。

代码示例：

```
int channel   = 3;
int width     = 1920;
int height    = 1080;
int dev_id    = 0;
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<unsigned char> src1_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
std::shared_ptr<unsigned char> src2_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
std::shared_ptr<unsigned char> res_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
unsigned char * src1_data = src1_ptr.get();
```

(continues on next page)

(continued from previous page)

```
unsigned char * src2_data = src2_ptr.get();
unsigned char * res_data = res_ptr.get();
for (int i = 0; i < channel * width * height; i++) {
    src1_data[i] = rand() % 255;
    src2_data[i] = rand() % 255;
}
// calculate res
bm_image input1, input2, output;
bm_image_create(handle,
                height,
                width,
                FORMAT_RGB_PLANAR,
                DATA_TYPE_EXT_1N_BYTE,
                &input1);
bm_image_alloc_dev_mem(input1);
bm_image_copy_host_to_device(input1, (void **)&src1_data);
bm_image_create(handle,
                height,
                width,
                FORMAT_RGB_PLANAR,
                DATA_TYPE_EXT_1N_BYTE,
                &input2);
bm_image_alloc_dev_mem(input2);
bm_image_copy_host_to_device(input2, (void **)&src2_data);
bm_image_create(handle,
                height,
                width,
                FORMAT_RGB_PLANAR,
                DATA_TYPE_EXT_1N_BYTE,
                &output);
bm_image_alloc_dev_mem(output);
if (BM_SUCCESS != bmcv_image_absdiff(handle, input1, input2, output)) {
    std::cout << "bmcv absdiff error !!!" << std::endl;
    bm_image_destroy(input1);
    bm_image_destroy(input2);
    bm_image_destroy(output);
    bm_dev_free(handle);
    exit(-1);
}
bm_image_copy_device_to_host(output, (void **)&res_data);
bm_image_destroy(input1);
bm_image_destroy(input2);
bm_image_destroy(output);
bm_dev_free(handle);
```

4.21 bmcv_image_add_weighted

实现两张相同大小图像的加权融合，具体如下：

$$output = alpha * input1 + beta * input2 + gamma$$

接口形式：

```
bm_status_t bmcv_image_add_weighted(
        bm_handle_t handle,
        bm_image input1,
        float alpha,
        bm_image input2,
        float beta,
        float gamma,
        bm_image output);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_image input1

输入参数。输入第一张图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- float alpha

第一张图像的权重。

- bm_image input2

输入参数。输入第二张图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- float beta

第二张图像的权重。

- float gamma

融合之后的偏移量。

- bm_image output

输出参数。输出 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

返回值说明：

- BM_SUCCESS: 成功

- 其他: 失败

格式支持：

该接口目前支持以下 image_format:

num	image_format
1	FORMAT_BGR_PACKED
2	FORMAT_BGR_PLANAR
3	FORMAT_RGB_PACKED
4	FORMAT_RGB_PLANAR
5	FORMAT_RGBP_SEPARATE
6	FORMAT_BGRP_SEPARATE
7	FORMAT_GRAY
8	FORMAT_YUV420P
9	FORMAT_YUV422P
10	FORMAT_YUV444P
11	FORMAT_NV12
12	FORMAT_NV21
13	FORMAT_NV16
14	FORMAT_NV61
15	FORMAT_NV24

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

注意事项：

1、在调用该接口之前必须确保输入的 image 内存已经申请。

2、input output 的 data_type，image_format 必须相同。

代码示例：

```
int channel   = 3;
int width     = 1920;
int height    = 1080;
int dev_id    = 0;
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<unsigned char> src1_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
std::shared_ptr<unsigned char> src2_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
```

(continues on next page)

(continued from previous page)

```
std::shared_ptr<unsigned char> res_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
unsigned char * src1_data = src1_ptr.get();
unsigned char * src2_data = src2_ptr.get();
unsigned char * res_data = res_ptr.get();
for (int i = 0; i < channel * width * height; i++) {
    src1_data[i] = rand() % 255;
    src2_data[i] = rand() % 255;
}
// calculate res
bm_image input1, input2, output;
bm_image_create(handle,
                height,
                width,
                FORMAT_RGB_PLANAR,
                DATA_TYPE_EXT_1N_BYTE,
                &input1);
bm_image_alloc_dev_mem(input1);
bm_image_copy_host_to_device(input1, (void **)&src1_data);
bm_image_create(handle,
                height,
                width,
                FORMAT_RGB_PLANAR,
                DATA_TYPE_EXT_1N_BYTE,
                &input2);
bm_image_alloc_dev_mem(input2);
bm_image_copy_host_to_device(input2, (void **)&src2_data);
bm_image_create(handle,
                height,
                width,
                FORMAT_RGB_PLANAR,
                DATA_TYPE_EXT_1N_BYTE,
                &output);
bm_image_alloc_dev_mem(output);
if (BM_SUCCESS != bmcv_image_add_weighted(handle, input1, 0.5, input2, 0.
↪5, 0, output)) {
    std::cout << "bmcv add_weighted error !!!" << std::endl;
    bm_image_destroy(input1);
    bm_image_destroy(input2);
    bm_image_destroy(output);
    bm_dev_free(handle);
    exit(-1);
}
bm_image_copy_device_to_host(output, (void **)&res_data);
bm_image_destroy(input1);
bm_image_destroy(input2);
```

(continues on next page)

(continued from previous page)

```
bm_image_destroy(output);
bm_dev_free(handle);
```

4.22 bmcv_image_threshold

图像阈值化操作。

接口形式：

```
bm_status_t bmcv_image_threshold(
        bm_handle_t handle,
        bm_image input,
        bm_image output,
        unsigned char thresh,
        unsigned char max_value,
        bm_thresh_type_t type);
```

其中 thresh 类型如下：

```
typedef enum {
    BM_THRESH_BINARY = 0,
    BM_THRESH_BINARY_INV,
    BM_THRESH_TRUNC,
    BM_THRESH_TOZERO,
    BM_THRESH_TOZERO_INV,
    BM_THRESH_TYPE_MAX
} bm_thresh_type_t;
```

各个类型对应的具体公式如下：

Enumerator	
THRESH_BINARY	$\texttt{dst}(x,y) = \begin{cases} \texttt{maxval} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$
THRESH_BINARY_INV	$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxval} & \text{otherwise} \end{cases}$
THRESH_TRUNC	$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$
THRESH_TOZERO	$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$
THRESH_TOZERO_INV	$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_image input

输入参数。输入图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image output

输出参数。输出 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

- unsigned char thresh

阈值。

- max_value

最大值。

- bm_thresh_type_t type

阈值化类型。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

格式支持：

该接口目前支持以下 image_format:

num	input image_format	output image_format
1	FORMAT_BGR_PACKED	FORMAT_BGR_PACKED
2	FORMAT_BGR_PLANAR	FORMAT_BGR_PLANAR
3	FORMAT_RGB_PACKED	FORMAT_RGB_PACKED
4	FORMAT_RGB_PLANAR	FORMAT_RGB_PLANAR
5	FORMAT_RGBP_SEPARATE	FORMAT_RGBP_SEPARATE
6	FORMAT_BGRP_SEPARATE	FORMAT_BGRP_SEPARATE
7	FORMAT_GRAY	FORMAT_GRAY
8	FORMAT_YUV420P	FORMAT_YUV420P
9	FORMAT_YUV422P	FORMAT_YUV422P
10	FORMAT_YUV444P	FORMAT_YUV444P
11	FORMAT_NV12	FORMAT_NV12
12	FORMAT_NV21	FORMAT_NV21
13	FORMAT_NV16	FORMAT_NV16
14	FORMAT_NV61	FORMAT_NV61
15	FORMAT_NV24	FORMAT_NV24

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

注意事项：

1、在调用该接口之前必须确保输入的 image 内存已经申请。

2、input output 的 image_format 以及 data_type 必须相同。

代码示例：

```
int channel   = 1;
int width     = 1920;
int height    = 1080;
int dev_id    = 0;
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<unsigned char> src_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
std::shared_ptr<unsigned char> res_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
unsigned char * src_data = src_ptr.get();
unsigned char * res_data = res_ptr.get();
for (int i = 0; i < channel * width * height; i++) {
    src_data[i] = rand() % 255;
}
// calculate res
bm_image input, output;
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &input);
bm_image_alloc_dev_mem(input);
bm_image_copy_host_to_device(input, (void **)&src_data);
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &output);
bm_image_alloc_dev_mem(output);
if (BM_SUCCESS != bmcv_image_threshold(handle, input, output, 200, 200,␣
↪BM_THRESH_BINARY)) {
    std::cout << "bmcv thresh error !!!" << std::endl;
    bm_image_destroy(input);
```

(continues on next page)

(continued from previous page)

```
    bm_image_destroy(output);
    bm_dev_free(handle);
    exit(-1);
}
bm_image_copy_device_to_host(output, (void **)&res_data);
bm_image_destroy(input);
bm_image_destroy(output);
bm_dev_free(handle);
```

4.23 bmcv_image_dct

对图像进行 DCT 变换。

接口的格式如下：

```
bm_status_t bmcv_image_dct(
        bm_handle_t handle,
        bm_image input,
        bm_image output,
        bool is_inversed);
```

输入参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄

- bm_image input

输入参数。输入 bm_image，bm_image 需要外部调用 bmcv_image _create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image output

输出参数。输出 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

- bool is_inversed

输入参数。是否为逆变换。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

由于 DCT 变换的系数仅与图像的 width 和 height 相关，而上述接口每次调用都需要重新计算变换系数，对于相同大小的图像，为了避免重复计算变换系数的过程，可以将上述接口拆分成两步完成：

1. 首先算特定大小的变换系数;
2. 然后可以重复利用改组系数对相同大小的图像做 DCT 变换。

计算系数的接口形式如下:

```
bm_status_t bmcv_dct_coeff(
        bm_handle_t handle,
        int H,
        int W,
        bm_device_mem_t hcoeff_output,
        bm_device_mem_t wcoeff_output,
        bool is_inversed);
```

输入参数说明:

- bm_handle_t handle

输入参数。bm_handle 句柄

- int H

输入参数。图像的高度。

- int W

输入参数。图像的宽度。

- bm_device_mem_t hcoeff_output

输出参数。该 device memory 空间存储着 h 维度的 DCT 变换系数，对于 H*W 大小的图像，该空间的大小为 H*H*sizeof(float)。

- bm_device_mem_t wcoeff_output

输出参数。该 device memory 空间存储着 w 维度的 DCT 变换系数，对于 H*W 大小的图像，该空间的大小为 W*W*sizeof(float)。

- bool is_inversed

输入参数。是否为逆变换。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

得到系数之后，将其传给下列接口开始计算过程:

```
bm_status_t bmcv_image_dct_with_coeff(
        bm_handle_t handle,
        bm_image input,
        bm_device_mem_t hcoeff,
        bm_device_mem_t wcoeff,
        bm_image output);
```

输入参数说明:

- bm_handle_t handle

输入参数。bm_handle 句柄

- bm_image input

输入参数。输入 bm_image，bm_image 需要外部调用 bmcv_image _create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_device_mem_t hcoeff

输入参数。该 device memory 空间存储着 h 维度的 DCT 变换系数，对于 H*W 大小的图像，该空间的大小为 H*H*sizeof(float)。

- bm_device_mem_t wcoeff

输入参数。该 device memory 空间存储着 w 维度的 DCT 变换系数，对于 H*W 大小的图像，该空间的大小为 W*W*sizeof(float)。

- bm_image output

输出参数。输出 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

格式支持:

该接口目前支持以下 image_format:

num	input image_format	output image_format
1	FORMAT_GRAY	FORMAT_GRAY

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_FLOAT32

注意事项:

1、在调用该接口之前必须确保输入的 image 内存已经申请。

2、input output 的 data_type 必须相同。

示例代码

```
int channel   = 1;
int width     = 1920;
int height    = 1080;
```

(continues on next page)

(continued from previous page)

```
int dev_id    = 0;
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<float> src_ptr(
        new float[channel * width * height],
        std::default_delete<float[]>());
std::shared_ptr<float> res_ptr(
        new float[channel * width * height],
        std::default_delete<float[]>());
float * src_data = src_ptr.get();
float * res_data = res_ptr.get();
for (int i = 0; i < channel * width * height; i++) {
    src_data[i] = rand() % 255;
}
bm_image bm_input, bm_output;
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_FLOAT32,
                &bm_input);
bm_image_alloc_dev_mem(bm_input);
bm_image_copy_host_to_device(bm_input, (void **)&src_data);
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_FLOAT32,
                &bm_output);
bm_image_alloc_dev_mem(bm_output);
bm_device_mem_t hcoeff_mem;
bm_device_mem_t wcoeff_mem;
bm_malloc_device_byte(handle, &hcoeff_mem, height*height*sizeof(float));
bm_malloc_device_byte(handle, &wcoeff_mem, width*width*sizeof(float));
bmcv_dct_coeff(handle, bm_input.height, bm_input.width, hcoeff_mem,␣
↪wcoeff_mem, is_inversed);
bmcv_image_dct_with_coeff(handle, bm_input, hcoeff_mem, wcoeff_mem, bm_
↪output);
bm_image_copy_device_to_host(bm_output, (void **)&res_data);
bm_image_destroy(bm_input);
bm_image_destroy(bm_output);
bm_free_device(handle, hcoeff_mem);
bm_free_device(handle, wcoeff_mem);
bm_dev_free(handle);
```

4.24 bmcv_image_sobel

边缘检测 Sobel 算子。

接口形式：

```
bm_status_t bmcv_image_sobel(
        bm_handle_t handle,
        bm_image input,
        bm_image output,
        int dx,
        int dy,
        int ksize = 3,
        float scale = 1,
        float delta = 0);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_image input

输入参数。输入图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image output

输出参数。输出 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

- int dx

x 方向上的差分阶数。

- int dy

y 方向上的差分阶数。

- int ksize = 3

Sobel 核的大小，必须是-1,1,3,5 或 7。其中特殊地，如果是-1 则使用 3×3 Scharr 滤波器，如果是 1 则使用 3×1 或者 1×3 的核。默认值为 3。

- float scale = 1

对求出的差分结果乘以该系数，默认值为 1。

- float delta = 0

在输出最终结果之前加上该偏移量，默认值为 0。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

格式支持：

该接口目前支持以下 image_format:

num	input image_format	output image_format
1	FORMAT_BGR_PACKED	FORMAT_BGR_PACKED
2	FORMAT_BGR_PLANAR	FORMAT_BGR_PLANAR
3	FORMAT_RGB_PACKED	FORMAT_RGB_PACKED
4	FORMAT_RGB_PLANAR	FORMAT_RGB_PLANAR
5	FORMAT_RGBP_SEPARATE	FORMAT_RGBP_SEPARATE
6	FORMAT_BGRP_SEPARATE	FORMAT_BGRP_SEPARATE
7	FORMAT_GRAY	FORMAT_GRAY
8	FORMAT_YUV420P	FORMAT_GRAY
9	FORMAT_YUV422P	FORMAT_GRAY
10	FORMAT_YUV444P	FORMAT_GRAY
11	FORMAT_NV12	FORMAT_GRAY
12	FORMAT_NV21	FORMAT_GRAY
13	FORMAT_NV16	FORMAT_GRAY
14	FORMAT_NV61	FORMAT_GRAY
15	FORMAT_NV24	FORMAT_GRAY

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

注意事项：

1、在调用该接口之前必须确保输入的 image 内存已经申请。

2、input output 的 data_type 必须相同。

3、目前支持图像的最大 width 为 2048。

代码示例：

```
int channel   = 1;
int width     = 1920;
int height    = 1080;
int dev_id    = 0;
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<unsigned char> src_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
```

(continues on next page)

(continued from previous page)

```
std::shared_ptr<unsigned char> res_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
unsigned char * src_data = src_ptr.get();
unsigned char * res_data = res_ptr.get();
for (int i = 0; i < channel * width * height; i++) {
    src_data[i] = rand() % 255;
}
// calculate res
bm_image input, output;
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &input);
bm_image_alloc_dev_mem(input);
bm_image_copy_host_to_device(input, (void **)&src_data);
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &output);
bm_image_alloc_dev_mem(output);
if (BM_SUCCESS != bmcv_image_sobel(handle, input, output, 0, 1)) {
    std::cout << "bmcv sobel error !!!" << std::endl;
    bm_image_destroy(input);
    bm_image_destroy(output);
    bm_dev_free(handle);
    exit(-1);
}
bm_image_copy_device_to_host(output, (void **)&res_data);
bm_image_destroy(input);
bm_image_destroy(output);
bm_dev_free(handle);
```

4.25 bmcv_image_canny

边缘检测 Canny 算子。

接口形式：

```
bm_status_t bmcv_image_canny(
        bm_handle_t handle,
        bm_image input,
```

(continues on next page)

(continued from previous page)

```
        bm_image output,
        float threshold1,
        float threshold2,
        int aperture_size = 3,
        bool l2gradient = false);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_image input

输入参数。输入图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image output

输出参数。输出 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

- float threshold1

滞后处理过程中的第一个阈值。

- float threshold2

滞后处理过程中的第二个阈值。

- int aperture_size = 3

Sobel 核的大小，目前仅支持 3。

- bool l2gradient = false

是否使用 L2 范数来求图像梯度, 默认值为 false。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

格式支持：

该接口目前支持以下 image_format:

num	input image_format	output image_format
1	FORMAT_GRAY	FORMAT_GRAY

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

注意事项：

1、在调用该接口之前必须确保输入的 image 内存已经申请。

2、input output 的 data_type，image_format 必须相同。

3、目前支持图像的最大 width 为 2048。

4、输入图像的 stride 必须和 width 一致。

代码示例：

```
int channel   = 1;
int width     = 1920;
int height    = 1080;
int dev_id    = 0;
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<unsigned char> src_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
std::shared_ptr<unsigned char> res_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
unsigned char * src_data = src_ptr.get();
unsigned char * res_data = res_ptr.get();
for (int i = 0; i < channel * width * height; i++) {
    src_data[i] = rand() % 255;
}
// calculate res
bm_image input, output;
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &input);
bm_image_alloc_dev_mem(input);
bm_image_copy_host_to_device(input, (void **)&src_data);
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &output);
bm_image_alloc_dev_mem(output);
if (BM_SUCCESS != bmcv_image_canny(handle, input, output, 0, 200)) {
```

(continues on next page)

(continued from previous page)

```
    std::cout << "bmcv canny error !!!" << std::endl;
    bm_image_destroy(input);
    bm_image_destroy(output);
    bm_dev_free(handle);
    exit(-1);
}
bm_image_copy_device_to_host(output, (void **)&res_data);
bm_image_destroy(input);
bm_image_destroy(output);
bm_dev_free(handle);
```

4.26 bmcv_image_yuv2hsv

对 YUV 图像的指定区域转为 HSV 格式。

接口形式：

```
bm_status_t bmcv_image_yuv2hsv(
        bm_handle_t handle,
        bmcv_rect_t rect,
        bm_image    input,
        bm_image    output);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bmcv_rect_t rect

描述了原图中待转换区域的启始坐标以及大小。具体参数可参见 bmcv_image_crop 接口中的描述。

- bm_image input

输入参数。输入图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image output

输出参数。输出 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

格式支持：

该接口目前支持以下 image_format:

num	input image_format	output image_format
1	FORMAT_YUV420P	FORMAT_HSV_PLANAR
2	FORMAT_NV12	FORMAT_HSV_PLANAR
3	FORMAT_NV21	FORMAT_HSV_PLANAR

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

注意事项：

1、在调用该接口之前必须确保输入的 image 内存已经申请。

代码示例：

```
int channel   = 2;
int width     = 1920;
int height    = 1080;
int dev_id    = 0;
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<unsigned char> src_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
std::shared_ptr<unsigned char> res_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
unsigned char * src_data = src_ptr.get();
unsigned char * res_data = res_ptr.get();
for (int i = 0; i < channel * width * height; i++) {
    src_data[i] = rand() % 255;
}
// calculate res
bmcv_rect_t rect;
rect.start_x   = 0;
rect.start_y   = 0;
rect.crop_w    = width;
rect.crop_h    = height;
bm_image input, output;
bm_image_create(handle,
                height,
                width,
                FORMAT_NV12,
```

(continues on next page)

(continued from previous page)

```
                DATA_TYPE_EXT_1N_BYTE,
                &input);
bm_image_alloc_dev_mem(input);
bm_image_copy_host_to_device(input, (void **)&src_data);
bm_image_create(handle,
                height,
                width,
                FORMAT_HSV_PLANAR,
                DATA_TYPE_EXT_1N_BYTE,
                &output);
bm_image_alloc_dev_mem(output);
if (BM_SUCCESS != bmcv_image_yuv2hsv(handle, rect, input, output)) {
    std::cout << "bmcv yuv2hsv error !!!" << std::endl;
    bm_image_destroy(input);
    bm_image_destroy(output);
    bm_dev_free(handle);
    exit(-1);
}
bm_image_copy_device_to_host(output, (void **)&res_data);
bm_image_destroy(input);
bm_image_destroy(output);
bm_dev_free(handle);
```

4.27 bmcv_image_gaussian_blur

图像的高斯滤波。

接口形式：

```
bm_status_t bmcv_image_gaussian_blur(
        bm_handle_t handle,
        bm_image input,
        bm_image output,
        int kw,
        int kh,
        float sigmaX,
        float sigmaY = 0);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_image input

输入参数。输入图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用

bmcv_image_attach 来 attach 已有的内存。

- bm_image output

输出参数。输出 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

- int kw

kernel 在 width 方向上的大小。

- int kh

kernel 在 height 方向上的大小。

- float sigmaX

X 方向上的高斯核标准差。

- float sigmaY = 0

Y 方向上的高斯核标准差。如果为 0 则表示与 X 方向上的高斯核标准差相同。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

格式支持：

该接口目前支持以下 image_format:

num	input image_format	output image_format
1	FORMAT_BGR_PACKED	FORMAT_BGR_PACKED
2	FORMAT_BGR_PLANAR	FORMAT_BGR_PLANAR
3	FORMAT_RGB_PACKED	FORMAT_RGB_PACKED
4	FORMAT_RGB_PLANAR	FORMAT_RGB_PLANAR
5	FORMAT_RGBP_SEPARATE	FORMAT_RGBP_SEPARATE
6	FORMAT_BGRP_SEPARATE	FORMAT_BGRP_SEPARATE
7	FORMAT_GRAY	FORMAT_GRAY
8	FORMAT_YUV420P	FORMAT_YUV420P
9	FORMAT_YUV422P	FORMAT_YUV422P
10	FORMAT_YUV444P	FORMAT_YUV444P

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

注意事项：

1、在调用该接口之前必须确保输入的 image 内存已经申请。

2、input output 的 data_type，image_format 必须相同。

3、目前支持图像的最大 width 为 2048。

代码示例：

```
int channel   = 1;
int width     = 1920;
int height    = 1080;
int dev_id    = 0;
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
std::shared_ptr<unsigned char> src_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
std::shared_ptr<unsigned char> res_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
unsigned char * src_data = src_ptr.get();
unsigned char * res_data = res_ptr.get();
for (int i = 0; i < channel * width * height; i++) {
    src_data[i] = rand() % 255;
}
// calculate res
bm_image input, output;
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &input);
bm_image_alloc_dev_mem(input);
bm_image_copy_host_to_device(input, (void **)&src_data);
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &output);
bm_image_alloc_dev_mem(output);
if (BM_SUCCESS != bmcv_image_gaussian_blur(handle, input, output, 3, 3, 0.
↪1)) {
    std::cout << "bmcv gaussian blur error !!!" << std::endl;
    bm_image_destroy(input);
    bm_image_destroy(output);
    bm_dev_free(handle);
    exit(-1);
}
bm_image_copy_device_to_host(output, (void **)&res_data);
```

(continues on next page)

(continued from previous page)

```
bm_image_destroy(input);
bm_image_destroy(output);
bm_dev_free(handle);
```

4.28 bmcv_image_transpose

该接口可以实现图片宽和高的转置。

接口形式:

```
bm_status_t bmcv_image_transpose(
        bm_handle_t handle,
        bm_image input,
        bm_image output
);
```

传入参数说明:

- bm_handle_t handle

输入参数。设备环境句柄，通过调用 bm_dev_request 获取。

- bm_image input

输入参数。输入图像的 bm_image 结构体。

- bm_image output

输出参数。输出图像的 bm_image 结构体。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

代码示例

```
#include <iostream>
#include <vector>
#include "bmcv_api_ext.h"
#include "bmlib_utils.h"
#include "common.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include <memory>

int main(int argc, char *argv[]) {
    bm_handle_t handle;
    bm_dev_request(&handle, 0);
```

(continues on next page)

(continued from previous page)

```
    int image_n = 1;
    int image_h = 1080;
    int image_w = 1920;
    bm_image src, dst;
    bm_image_create(handle, image_h, image_w, FORMAT_RGB_PLANAR,
            DATA_TYPE_EXT_1N_BYTE, &src);
    bm_image_create(handle, image_w, image_h, FORMAT_RGB_PLANAR,
            DATA_TYPE_EXT_1N_BYTE, &dst);
    std::shared_ptr<u8*> src_ptr = std::make_shared<u8*>(
            new u8[image_h * image_w * 3]);
    memset((void *)(*src_ptr.get()), 148, image_h * image_w * 3);
    u8 *host_ptr[] = {*src_ptr.get()};
    bm_image_copy_host_to_device(src, (void **)host_ptr);
    bmcv_image_transpose(handle, src, dst);
    bm_image_destroy(src);
    bm_image_destroy(dst);
    bm_dev_free(handle);
    return 0;
}
```

注意事项:

1. 该 API 要求输入和输出的 bm_image 图像格式相同，支持以下格式：
   - FORMAT_RGB_PLANAR
   - FORMAT_BGR_PLANAR
   - FORMAT_GRAY
2. 该 API 要求输入和输出的 bm_image 数据类型相同，支持以下类型：
   - DATA_TYPE_EXT_FLOAT32,
   - DATA_TYPE_EXT_1N_BYTE,
   - DATA_TYPE_EXT_4N_BYTE,
   - DATA_TYPE_EXT_1N_BYTE_SIGNED,
   - DATA_TYPE_EXT_4N_BYTE_SIGNED,
3. 输出图像的 width 必须等于输入图像的 height，输出图像的 height 必须等于输入图像的 width ;
4. 输入图像支持带有 stride;
5. 输入输出 bm_image 结构必须提前创建，否则返回失败。
6. 输入 bm_image 必须 attach device memory，否则返回失败
7. 如果输出对象未 attach device memory，则会内部调用 bm_image_alloc_dev_mem 申请内部管理的 device memory，并将转置后的数据填充到 device memory 中。

4.29 bmcv_image_morph

可以实现对图像的基本形态学运算，包括膨胀 (Dilation) 和腐蚀 (Erosion)。

用户可以分为以下两步使用该功能：

4.29.1 获取 Kernel 的 Device Memory

可以在初始化时使用以下接口获取存储 Kernel 的 Device Memory，当然用户也可以自定义 Kernel 直接忽略该步骤。

函数通过传入所需 Kernel 的大小和形状，返回对应的 Device Memory 给后面的形态学运算接口使用，用户应用程序的最后需要用户手动释放该空间。

接口形式：

```
typedef enum {
    BM_MORPH_RECT,
    BM_MORPH_CROSS,
    BM_MORPH_ELLIPSE
} bmcv_morph_shape_t;

bm_device_mem_t bmcv_get_structuring_element(
        bm_handle_t handle,
        bmcv_morph_shape_t shape,
        int kw,
        int kh
        );
```

参数说明：

- bm_handle_t handle

  输入参数。bm_handle 句柄。

- bmcv_morph_shape_t shape

  输入参数。表示 Kernel 的形状，目前支持矩形、十字、椭圆。

- int kw

  输入参数。Kernel 的宽度。

- int kh

  输入参数。Kernel 的高度。

返回值说明：

返回 Kernel 对应的 Device Memory 空间。

4.29.2 形态学运算

目前支持腐蚀和膨胀操作，用户也可以通过这两个基本操作的组合实现以下功能：

- 开运算 (Opening)
- 闭运算 (Closing)
- 形态梯度 (Morphological Gradient)
- 顶帽 (Top Hat)
- 黑帽 (Black Hat)

接口形式：

```
bm_status_t bmcv_image_erode(
        bm_handle_t handle,
        bm_image src,
        bm_image dst,
        int kw,
        int kh,
        bm_device_mem_t kmem
        );

bm_status_t bmcv_image_dilate(
        bm_handle_t handle,
        bm_image src,
        bm_image dst,
        int kw,
        int kh,
        bm_device_mem_t kmem
        );
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_image src

输入参数。需处理图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image dst

输出参数。处理后图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存，如用户不申请内部会自动申请。

- int kw

输入参数。Kernel 的宽度。

- int kh

输入参数。Kernel 的高度。

- bm_device_mem_t kmem

输入参数。存储 Kernel 的 Device Memory 空间，可以通过接口 bmcv_get_structuring_element 获取，用户也可以自定义，其中值为 1 表示选中该像素，值为 0 表示忽略该像素。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

格式支持：

该接口目前支持以下 image_format:

num	image_format
1	FORMAT_GRAY
2	FORMAT_RGB_PLANAR
3	FORMAT_BGR_PLANAR
4	FORMAT_RGB_PACKED
5	FORMAT_BGR_PACKED

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

代码示例：

```
int channel    = 1;
int width      = 1920;
int height     = 1080;
int kw         = 3;
int kh         = 3;
int dev_id     = 0;
bmcv_morph_shape_t shape = BM_MORPH_RECT;
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
bm_device_mem_t kmem = bmcv_get_structuring_element(
        handle,
        shape,
        kw,
        kh);
std::shared_ptr<unsigned char> data_ptr(
        new unsigned char[channel * width * height],
        std::default_delete<unsigned char[]>());
```

(continues on next page)

(continued from previous page)

```
for (int i = 0; i < channel * width * height; i++) {
    data_ptr.get()[i] = rand() % 255;
}
// calculate res
bm_image src, dst;
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &src);
bm_image_create(handle,
                height,
                width,
                FORMAT_GRAY,
                DATA_TYPE_EXT_1N_BYTE,
                &dst);
bm_image_alloc_dev_mem(src);
bm_image_alloc_dev_mem(dst);
bm_image_copy_host_to_device(src, (void **)&(data_ptr.get()));
if (BM_SUCCESS != bmcv_image_erode(handle, src, dst, kw, kh, kmem)) {
    std::cout << "bmcv erode error !!!" << std::endl;
    bm_image_destroy(src);
    bm_image_destroy(dst);
    bm_free_device(handle, kmem);
    bm_dev_free(handle);
    return;
}
bm_image_copy_device_to_host(dst, (void **)&(data_ptr.get()));
bm_image_destroy(src);
bm_image_destroy(dst);
bm_free_device(handle, kmem);
bm_dev_free(handle);
```

4.30 bmcv_image_laplacian

梯度计算 laplacian 算子。

接口形式：

```
bm_status_t  bmcv_image_laplacian(
    bm_handle_t handle,
    bm_image input,
    bm_image output,
    unsigned int ksize);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_image input

输入参数。输入图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- bm_image output

输出参数。输出 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以通过 bm_image_alloc_dev_mem 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。如果不主动分配将在 api 内部进行自行分配。

- int ksize = 3

Laplacian 核的大小，必须是 1 或 3。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

格式支持：

该接口目前支持以下 image_format:

num	input image_format	output image_format
1	FORMAT_GRAY	FORMAT_GRAY

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

注意事项：

1、在调用该接口之前必须确保输入的 image 内存已经申请。

2、input output 的 data_type 必须相同。

3、目前支持图像的最大 width 为 2048。

代码示例：

```
int loop =1;
int ih = 1080;
int iw = 1920;
unsigned int ksize = 3;
bm_image_format_ext fmt = FORMAT_GRAY;
```

(continues on next page)

(continued from previous page)

```
fmt = argc > 1 ? (bm_image_format_ext)atoi(argv[1]) : fmt;
ih = argc > 2 ? atoi(argv[2]) : ih;
iw = argc > 3 ? atoi(argv[3]) : iw;
loop = argc > 4 ? atoi(argv[4]) : loop;
ksize = argc >5 ? atoi(argv[5]) : ksize;

bm_status_t ret = BM_SUCCESS;
bm_handle_t handle;
ret = bm_dev_request(&handle, 0);
if (ret != BM_SUCCESS)
    throw("bm_dev_request failed");

bm_image_data_format_ext data_type = DATA_TYPE_EXT_1N_BYTE;
bm_image input;
bm_image output;

bm_image_create(handle, ih, iw, fmt, data_type, &input);
bm_image_alloc_dev_mem(input);

bm_image_create(handle,ih, iw, fmt, data_type, &output);
bm_image_alloc_dev_mem(output);

std::shared_ptr<unsigned char*> ch0_ptr = std::make_shared<unsigned char*>
↪(new unsigned char[ih * iw]);
std::shared_ptr<unsigned char*> tpu_res_ptr = std::make_shared<unsigned␣
↪char *>(new unsigned char[ih * iw]);
std::shared_ptr<unsigned char*> cpu_res_ptr = std::make_shared<unsigned␣
↪char *>(new unsigned char[ih*iw]);

for (int i = 0; i < loop; i++) {
    for (int j = 0; j < ih * iw; j++) {
        (*ch0_ptr.get())[j] = j % 256;
    }

    unsigned char *host_ptr[] = {*ch0_ptr.get()};
    bm_image_copy_host_to_device(input, (void **)host_ptr);

    ret = bmcv_image_laplacian(handle, input, output, ksize);
    if (ret) {
        cout << "test laplacian failed" << endl;
        bm_image_destroy(input);
        bm_image_destroy(output);
        bm_dev_free(handle);
        return ret;
    } else {
        host_ptr[0] = *tpu_res_ptr.get();
        bm_image_copy_device_to_host(output, (void **)host_ptr);
```

(continues on next page)

(continued from previous page)

```
    }
}

bm_image_destroy(input);
bm_image_destroy(output);
bm_dev_free(handle);
```

4.31 bmcv_image_lkpyramid

LK 金字塔光流算法。完整的使用步骤包括创建、执行、销毁三步。该算法前半部分使用 TPU，而后半部分为串行运算需要使用 CPU，因此对于 PCIe 模式，建议使能 CPU 进行加速，具体步骤参考第 5 章节。

4.31.1 创建

由于该算法的内部实现需要一些缓存空间，为了避免重复申请释放空间，将一些准备工作封装在该创建接口中，只需要在启动前调用一次便可以多次调用 execute 接口（创建函数参数不变的情况下），接口形式如下：

```
bm_status_t bmcv_image_lkpyramid_create_plan(
        bm_handle_t handle,
        void*& plan,
        int width,
        int height,
        int winW = 21,
        int winH = 21,
        int maxLevel = 3);
```

输入参数说明：

- bm_handle_t handle

  输入参数。bm_handle 句柄

- void*& plan

  输出参数。执行阶段所需要的句柄。

- int width

  输入参数。待处理图像的宽度。

- int height

  输入参数，待处理图像的高度。

- int winW

  输入参数，算法处理窗口的宽度，默认值为 21。

- int winH

输入参数，算法处理窗口的高度，默认值为 21。

- int maxLevel

输入参数，金字塔处理的高度，默认值为 3, 目前支持的最大值为 5。该参数值越大，算法执行时间越长，建议根据实际效果选择可接受的最小值。

返回值说明:

- BM_SUCCESS: 成功
- 其他：失败

4.31.2 执行

使用上述接口创建后的 plan 就可以开始真正的执行阶段了，接口格式如下：

```
typedef struct {
    float x;
    float y;
} bmcv_point2f_t;

typedef struct {
    int type;    // 1: maxCount   2: eps   3: both
    int max_count;
    double epsilon;
} bmcv_term_criteria_t;

bm_status_t bmcv_image_lkpyramid_execute(
        bm_handle_t handle,
        void* plan,
        bm_image prevImg,
        bm_image nextImg,
        int ptsNum,
        bmcv_point2f_t* prevPts,
        bmcv_point2f_t* nextPts,
        bool* status,
        bmcv_term_criteria_t criteria = {3, 30, 0.01});
```

输入参数说明:

- bm_handle_t handle

输入参数。bm_handle 句柄

- const void *plan

输入参数。创建阶段所得到的句柄。

- bm_image prevImg

输入参数。前一幅图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用

bmcv_image_attach 来 attach 已有的内存。

- bm_image nextImg

输入参数。后一幅图像的 bm_image，bm_image 需要外部调用 bmcv_image_create 创建。image 内存可以使用 bm_image_alloc_dev_mem 或者 bm_image_copy_host_to_device 来开辟新的内存，或者使用 bmcv_image_attach 来 attach 已有的内存。

- int ptsNum

输入参数。需要追踪点的数量。

- bmcv_point2f_t* prevPts

输入参数。需要追踪点在前一幅图中的坐标指针，其指向的长度为 ptsNum。

- bmcv_point2f_t* nextPts

输出参数。计算得到的追踪点在后一张图像中坐标指针，其指向的长度为 ptsNum。

- bool* status

输出参数。nextPts 中的各个追踪点是否有效，其指向的长度为 ptsNum，与 nextPts 中的坐标一一对应，如果有效则为 true，否则为 false（表示没有在后一张图像中找到对应的跟踪点，可能超出图像范围）。

- bmcv_term_criteria_t criteria

输入参数。迭代结束标准，type 表示以哪个参数作为结束判断条件：若为 1 则以迭代次数 max_count 为结束判断参数，若为 2 则以误差 epsilon 为结束判断参数，若为 3 则两者均需满足。该参数会影响执行时间，建议根据实际效果选择最优的停止迭代标准。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

4.31.3 销毁

当执行完成后需要销毁所创建的句柄。该接口必须和创建接口 bmcv_image_lkpyramid_create_plan 成对使用。

```
void bmcv_image_lkpyramid_destroy_plan(bm_handle_t handle, void *plan);
```

格式支持:

该接口目前支持以下 image_format:

num	image_format
1	FORMAT_GRAY

目前支持以下 data_type:

num	data_type
1	DATA_TYPE_EXT_1N_BYTE

4.31.4 示例代码

```
bm_handle_t handle;
bm_status_t ret = bm_dev_request(&handle, 0);
if (ret != BM_SUCCESS) {
    printf("Create bm handle failed. ret = %d\n", ret);
    return -1;
}
ret = bmcv_open_cpu_process(handle);
if (ret != BM_SUCCESS) {
    printf("BMCV enable CPU failed. ret = %d\n", ret);
    bm_dev_free(handle);
    return -1;
}
bm_image_format_ext fmt = FORMAT_GRAY;
bm_image prevImg;
bm_image nextImg;
bm_image_create(handle, height, width, fmt, DATA_TYPE_EXT_1N_BYTE, &
↪prevImg);
bm_image_create(handle, height, width, fmt, DATA_TYPE_EXT_1N_BYTE, &
↪nextImg);
bm_image_alloc_dev_mem(prevImg);
bm_image_alloc_dev_mem(nextImg);
bm_image_copy_host_to_device(prevImg, (void **)(&prevPtr));
bm_image_copy_host_to_device(nextImg, (void **)(&nextPtr));
void *plan = nullptr;
bmcv_image_lkpyramid_create_plan(
        handle,
        plan,
        width,
        height,
        kw,
        kh,
        maxLevel);
bmcv_image_lkpyramid_execute(
        handle,
        plan,
        prevImg,
        nextImg,
        ptsNum,
        prevPts,
        nextPts,
        status,
        criteria);
bmcv_image_lkpyramid_destroy_plan(handle, plan);
bm_image_destroy(prevImg);
bm_image_destroy(nextImg);
ret = bmcv_close_cpu_process(handle);
```

(continues on next page)

(continued from previous page)

```
if (ret != BM_SUCCESS) {
    printf("BMCV disable CPU failed. ret = %d\n", ret);
    bm_dev_free(handle);
    return -1;
}
bm_dev_free(handle);
```

4.32 bmcv_debug_savedata

该接口用于将 bm_image 对象输出至内部定义的二进制文件方便 debug，二进制文件格式以及解析方式在示例代码中给出。

接口形式:

```
bm_status_t bmcv_debug_savedata(
        bm_image input,
        const char *name
);
```

参数说明:

- bm_image input

  输入参数。输入 bm_image。

- const char* name

  输入参数。保存的二进制文件路径以及文件名称。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

代码示例以及二进制文件解析方法:

```
bm_image input;
bm_image_create(handle,
    1080,
    1920,
    FORMAT_BGR_PLANAR,
    DATA_TYPE_EXT_1N_BYTE,
    &input);
bm_image_alloc_dev_mem(input);
// ... your own function
bmcv_debug_savedata(input, "input.bin");
// now a file named "input.bin" is generated in current folder

// the following code shows how to parse the binary file
```

(continues on next page)

(continued from previous page)

```
FILE *   fp           = fopen("input.bin", "rb");
uint32_t data_offset  = 0;
uint32_t width        = 0;
uint32_t height       = 0;
uint32_t image_format = 0;
uint32_t data_type    = 0;
uint32_t plane_num    = 0;

uint32_t stride[4] = {0};
uint64_t size[4]         = {0};

fread(&data_offset, sizeof(uint32_t), 1, fp);
fread(&width, sizeof(uint32_t), 1, fp);
fread(&height, sizeof(uint32_t), 1, fp);
fread(&image_format, sizeof(uint32_t), 1, fp);
fread(&data_type, sizeof(uint32_t), 1, fp);
fread(&plane_num, sizeof(uint32_t), 1, fp);

fread(size, sizeof(size), 1, fp);
fread(stride, sizeof(stride), 1, fp);

uint32_t channel_stride[4] = {0};
uint32_t batch_stride[4]   = {0};
uint32_t meta_data_size[4] = {0};

uint32_t N[4] = {0};
uint32_t C[4] = {0};
uint32_t H[4] = {0};
uint32_t W[4] = {0};

fread(channel_stride, sizeof(channel_stride), 1, fp);
fread(batch_stride, sizeof(batch_stride), 1, fp);
fread(meta_data_size, sizeof(meta_data_size), 1, fp);

fread(N, sizeof(N), 1, fp);
fread(C, sizeof(C), 1, fp);
fread(H, sizeof(H), 1, fp);
fread(W, sizeof(W), 1, fp);

fseek(fp, data_offset, SEEK_SET);
std::vector<std::unique_ptr<unsigned char[]>> host_ptr;
host_ptr.resize(plane_num);
void* void_ptr[4] = {0};
for (uint32_t i = 0; i < plane_num; i++) {
    host_ptr[i] =
        std::unique_ptr<unsigned char[]>(new unsigned char[size[i]]);
    void_ptr[i] = host_ptr[i].get();
```

(continues on next page)

(continued from previous page)

```
    fread(host_ptr[i].get(), 1, size[i], fp);
}
fclose(fp);
std::cout << "image width " << width << " image height " << height
        << " image format " << image_format << " data type " << data_type
        << " plane num " << plane_num << std::endl;
for (uint32_t i = 0; i < plane_num; i++) {
    std::cout << "plane" << i << " size " << size[i] << " C " << C[i]
            << " H " << H[i] << " W " << W[i] << " stride "
            << stride[i] << std::endl;
}
// The following shows how to recover the image
bm_image recover;
bm_image_create(handle,
                height,
                width,
                (bm_image_format_ext)image_format,
                (bm_image_data_format_ext)data_type,
                &recover,
                (int *)stride);
bm_image_copy_host_to_device(recover, (void **)&void_ptr);
bm_image_write_to_bmp(recover, "recover.bmp");
bm_image_destroy(recover);
```

注意事项:

1. 在调用 bmcv_debug_savedata() 之前必须确保输入的 image 已被正确创建并保证 is_attached，否则该函数将返回失败。

4.33 bmcv_sort

该接口可以实现浮点数据的排序（升序/降序），并且支持排序后可以得到原数据所对应的 index。

接口形式:

```
bm_status_t bmcv_sort(bm_handle_t     handle,
                      bm_device_mem_t src_index_addr,
                      bm_device_mem_t src_data_addr,
                      int             data_cnt,
                      bm_device_mem_t dst_index_addr,
                      bm_device_mem_t dst_data_addr,
                      int             sort_cnt,
                      int             order,
                      bool            index_enable,
                      bool            auto_index);
```

输入参数说明:

- bm_handle_t handle

输入参数。输入的 bm_handle 句柄。

- bm_device_mem_t src_index_addr

输入参数。每个输入数据所对应 index 的地址。如果使能 index_enable 并且不使用 auto_index 时，则该参数有效。bm_device_mem_t 为内置表示地址的数据类型，可以使用函数 bm_mem_from_system(addr) 将普通用户使用的指针或地址转为该类型，用户可参考示例代码中的使用方式。

- bm_device_mem_t src_data_addr

输入参数。待排序的输入数据所对应的地址。bm_device_mem_t 为内置表示地址的数据类型，可以使用函数 bm_mem_from_system(addr) 将普通用户使用的指针或地址转为该类型，用户可参考示例代码中的使用方式。

- int data_cnt

输入参数。待排序的输入数据的数量。

- bm_device_mem_t dst_index_addr

输出参数。排序后输出数据所对应 index 的地址，如果使能 index_enable 并且不使用 auto_index 时，则该参数有效。bm_device_mem_t 为内置表示地址的数据类型，可以使用函数 bm_mem_from_system(addr) 将普通用户使用的指针或地址转为该类型，用户可参考示例代码中的使用方式。

- bm_device_mem_t dst_data_addr

输出参数。排序后的输出数据所对应的地址。bm_device_mem_t 为内置表示地址的数据类型，可以使用函数 bm_mem_from_system(addr) 将普通用户使用的指针或地址转为该类型，用户可参考示例代码中的使用方式。

- int sort_cnt

输入参数。需要排序的数量，也就是输出结果的个数，包括排好序的数据和对应 index 。比如降序排列，如果只需要输出前 3 大的数据，则该参数设置为 3 即可。

- int order

输入参数。升序还是降序，0 表示升序，1 表示降序。

- bool index_enable

输入参数。是否使能 index。如果使能即可输出排序后数据所对应的 index ，否则 src_index_addr 和 dst_index_addr 这两个参数无效。

- bool auto_index

输入参数。是否使能自动生成 index 功能。使用该功能的前提是 index_enable 参数为 true，如果该参数也为 true 则表示按照输入数据的存储顺序从 0 开始计数作为 index，参数 src_index_addr 便无效，输出结果中排好序数据所对应的 index 即存放于 dst_index_addr 地址中。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项：

1、要求 sort_cnt <= data_cnt。

2、若需要使用 auto index 功能，前提是参数 index_enable 为 true。

3、该 api 至多可支持 1MB 数据的全排序。

示例代码

```
int data_cnt = 100;
int sort_cnt = 50;
float src_data_p[100];
int src_index_p[100];
float dst_data_p[50];
int dst_index_p[50];
for (int i = 0; i < 100; i++) {
    src_data_p[i] = rand() % 1000;
    src_index_p[i] = 100 - i;
}
int order = 0;
bmcv_sort(handle,
          bm_mem_from_system(src_index_p),
          bm_mem_from_system(src_data_p),
          data_cnt,
          bm_mem_from_system(dst_index_p),
          bm_mem_from_system(dst_data_p),
          sort_cnt,
          order,
          true,
          false);
```

4.34 bmcv_base64_enc(dec)

base64 网络传输中常用的编码方式，利用 64 个常用字符来对 6 位二进制数编码。

接口形式：

```
bm_status_t bmcv_base64_enc(bm_handle_t handle,
        bm_device_mem_t src,
        bm_device_mem_t dst,
        unsigned long len[2])

bm_status_t bmcv_base64_dec(bm_handle_t handle,
        bm_device_mem_t src,
        bm_device_mem_t dst,
        unsigned long len[2])
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_device_mem_t src

输入参数。输入字符串所在地址，类型为 bm_device_mem_t。需要调用 bm_mem_from_system() 将数据地址转化成转化为 bm_device_mem_t 所对应的结构。

- bm_device_mem_t dst

输入参数。输出字符串所在地址，类型为 bm_device_mem_t。需要调用 bm_mem_from_system() 将数据地址转化成转化为 bm_device_mem_t 所对应的结构。

- unsigned long len[2]

输入参数。进行 base64 编码或解码的长度，单位是字节。其中 len[0] 代表输入长度，需要调用者给出。而 len[1] 为输出长度，由 api 计算后给出。

返回值：

- BM_SUCCESS: 成功
- 其他: 失败

代码示例：

```
int original_len[2];
int encoded_len[2];
int original_len[0] = (rand() % 134217728) + 1;
int encoded_len[0] = (original_len + 2) / 3 * 4;
char *src = (char *)malloc((original_len + 3) * sizeof(char));
char *dst = (char *)malloc((encoded_len + 3) * sizeof(char));
for (j = 0; j < original_len; j++)
    a[j] = (char)((rand() % 100) + 1);

bm_handle_t handle;
ret = bm_dev_request(&handle, 0);
if (ret != BM_SUCCESS) {
    printf("Create bm handle failed. ret = %d\n", ret);
    exit(-1);
}
bmcv_base64_enc(
    handle,
    bm_mem_from_system(src),
    bm_mem_from_system(dst),
    original_len);

bmcv_base64_dec(
    handle,
    bm_mem_from_system(dst),
    bm_mem_from_system(src),
    original_len);
```

(continues on next page)

(continued from previous page)

```
bm_dev_free(handle);
free(src);
free(dst);
```

注意事项：

1、该 api 一次最多可对 128MB 的数据进行编解码，即参数 len 不可超过 128MB。

2、同时支持传入地址类型为 system 或 device。

3、encoded_len[1] 在会给出输出长度，尤其是解码时根据输入的末尾计算需要去掉的位数

4.35 bmcv_feature_match

该接口用于将网络得到特征点（int8 格式）与数据库中特征点（int8 格式）进行比对，输出最佳匹配的 top-k。

接口形式：

```
bm_status_t bmcv_feature_match(
    bm_handle_t     handle,
    bm_device_mem_t input_data_global_addr,
    bm_device_mem_t db_data_global_addr,
    bm_device_mem_t output_sorted_similarity_global_addr,
    bm_device_mem_t output_sorted_index_global_addr,
    int             batch_size,
    int             feature_size,
    int             db_size,
    int             sort_cnt = 1,
    int             rshiftbits = 0);
```

输入参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_device_mem_t input_data_global_addr

输入参数。所要比对的特征点数据存储的地址。该数据按照 batch_size * feature_size 的数据格式进行排列。batch_size，feature _size 具体含义将在下面进行介绍。bm_device_mem_t 为内置表示地址的数据类型，可以使用函数 bm_mem_from_system(addr) 将普通用户使用的指针或地址转为该类型，用户可参考示例代码中的使用方式。

- bm_device_mem_t db_data_global_addr

输入参数。数据库的特征点数据存储的地址。该数据按照 feature_size * db_size 的数据格式进行排列。feature_size，db_size 具体含义将在下面进行介绍。bm_device_mem_t 为内置表示地址的数据类型，可以使用函数 bm_mem_from_system(addr) 将普通用户使用的指针或地址转为该类型，用户可参考示例代码中的使用方式。

- bm_device_mem_t output_sorted_similarity_global_addr

输出参数。每个 batch 得到的比对结果的值中最大几个值（降序排列）存储地址，具体取多少个值由 sort_cnt 决定。该数据按照 batch_size * sort_cnt 的数据格式进行排列。batch_size 具体含义将在下面进行介绍。bm_device_mem_t 为内置表示地址的数据类型，可以使用函数 bm_mem_from_system(addr) 将普通用户使用的指针或地址转为该类型，用户可参考示例代码中的使用方式。

- bm_device_mem_t output_sorted_index_global_addr

输出参数。每个 batch 得到的比对结果的在数据库中的序号的存储地址。如对于 batch 0，如果 output_sorted_similarity_global_addr 中 bacth 0 的数据是由输入数据与数据库的第 800 组特征点进行比对得到的，那么 output_sorted_index_global_addr 所在地址对应 batch 0 的数据为 800. output_sorted_similarity_global_addr 中的数据按照 batch_size * sort_cnt 的数据格式进行排列。batch_size 具体含义将在下面进行介绍。bm_device_mem_t 为内置表示地址的数据类型，可以使用函数 bm_mem_from_system(addr) 将普通用户使用的指针或地址转为该类型，用户可参考示例代码中的使用方式。

- int batch_size

输入参数。待输入数据的 batch 个数，如输入数据有 4 组特征点，则该数据的 batch_size 为 4。batch_size 最大值不应超过 8。

- int feature_size

输入参数。每组数据的特征点个数。feature_size 最大值不应该超过 4096。

- int db_size

输入参数。数据库中数据特征点的组数。db_size 最大值不应该超过 500000。

- int sort_cnt

输入参数。每个 batch 对比结果中所要排序个数，也就是输出结果个数，如需要最大的 3 个比对结果，则 sort_cnt 设置为 3。该值默认为 1。sort_cnt 最大值不应该超过 30。

- int rshiftbits

输入参数。对结果进行右移处理的位数，右移采用 round 对小数进行取整处理。该参数默认为 0。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

注意事项:

1、输入数据和数据库中数据的数据类型为 char。

2、输出的比对结果数据类型为 short，输出的序号类型为 int。

3、数据库中的数据在内存的排布为 feature_size * db_size，因此需要将一组特征点进行转置之后再放入数据库中。

4、sort_cnt 的取值范围为 1 ~ 30。

示例代码

```
int batch_size = 4;
int feature_size = 512;
int db_size = 1000;
int sort_cnt = 1;
unsigned char src_data_p[4 * 512];
unsigned char db_data_p[512 * 1000];
short output_val[4];
int output_index[4];
for (int i = 0; i < 4 * 512; i++) {
    src_data_p[i] = rand() % 1000;
}
for (int i = 0; i < 512 * 1000; i++) {
    db_data_p[i] = rand() % 1000;
}
bmcv_feature_match(handle,
    bm_mem_from_system(src_data_p),
    bm_mem_from_system(db_data_p),
    bm_mem_from_system(output_val),
    bm_mem_from_system(output_index),
    batch_size,
    feature_size,
    db_size,
    sort_cnt, 8);
```

4.36 bmcv_gemm

该接口可以实现 float32 类型矩阵的通用乘法计算，如下公式：

$$C = \alpha \times A \times B + \beta \times C$$

其中，A、B、C 均为矩阵，α 和 β 均为常系数

接口的格式如下：

```
bm_status_t bmcv_gemm(bm_handle_t     handle,
                      bool            is_A_trans,
                      bool            is_B_trans,
                      int             M,
                      int             N,
                      int             K,
                      float           alpha,
                      bm_device_mem_t A,
                      int             lda,
                      bm_device_mem_t B,
                      int             ldb,
```

(continues on next page)

(continued from previous page)

```
                    float           beta,
                    bm_device_mem_t C,
                    int             ldc);
```

输入参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄

- bool is_A_trans

输入参数。设定矩阵 A 是否转置

- bool is_B_trans

输入参数。设定矩阵 B 是否转置

- int M

输入参数。矩阵 A 和矩阵 C 的行数

- int N

输入参数。矩阵 B 和矩阵 C 的列数

- int K

输入参数。矩阵 A 的列数和矩阵 B 的行数

- float alpha

输入参数。数乘系数

- bm_device_mem_t A

输入参数。根据数据存放位置保存左矩阵 A 数据的 device 地址或者 host 地址。如果数据存放于 host 空间则内部会自动完成 s2d 的搬运

- int lda

输入参数。矩阵 A 的 leading dimension, 即第一维度的大小，在行与行之间没有 stride 的情况下即为 A 的列数（不做转置）或行数（做转置）

- bm_device_mem_t B

输入参数。根据数据存放位置保存右矩阵 B 数据的 device 地址或者 host 地址。如果数据存放于 host 空间则内部会自动完成 s2d 的搬运。

- int ldb

输入参数。矩阵 C 的 leading dimension, 即第一维度的大小，在行与行之间没有 stride 的情况下即为 B 的列数（不做转置）或行数（做转置。

- float beta

输入参数。数乘系数。

- bm_device_mem_t C

输出参数。根据数据存放位置保存矩阵 C 数据的 device 地址或者 host 地址。如果是 host 地址，则当 beta 不为 0 时，计算前内部会自动完成 s2d 的搬运，计算后再自动完成 d2s 的搬运。

- int ldc

输入参数。矩阵 C 的 leading dimension, 即第一维度的大小，在行与行之间没有 stride 的情况下即为 C 的列数。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

示例代码

```
int M = 3, N = 4, K = 5;
float alpha = 0.4, beta = 0.6;
bool is_A_trans = false;
bool is_B_trans = false;
float *A      = new float[M * K];
float *B      = new float[N * K];
float *C      = new float[M * N];
memset(A, 0x11, M * K * sizeof(float));
memset(B, 0x22, N * K * sizeof(float));
memset(C, 0x33, M * N * sizeof(float));

bmcv_gemm(handle,
          is_A_trans,
          is_B_trans,
          M,
          N,
          K,
          alpha,
          bm_mem_from_system((void *)A),
          is_A_trans ? M : K,
          bm_mem_from_system((void *)B),
          is_B_trans ? K : N,
          beta,
          bm_mem_from_system((void *)C),
          N);
delete A;
delete B;
delete C;
```

4.37 bmcv_matmul

该接口可以实现 8-bit 数据类型矩阵的乘法计算，如下公式：

$$C = (A \times B) >> rshift\_bit \tag{4.1}$$

或者

$$C = alpha \times (A \times B) + beta \tag{4.2}$$

其中，

- A 是输入的左矩阵，其数据类型可以是 unsigned char 或者 signed char 类型的 8-bit 数据，大小为（M，K）；
- B 是输入的右矩阵，其数据类型可以是 unsigned char 或者 signed char 类型的 8-bit 数据，大小为（K，N）；
- C 是输出的结果矩阵，其数据类型长度可以是 int8、int16 或者 float32，用户配置决定。

  当 C 是 int8 或者 int16 时，执行上述公式 (4.1) 的功能，而其符号取决于 A 和 B，当 A 和 B 均为无符号时 C 才为无符号数，否则为有符号；

  当 C 是 float32 时，执行上述公式 (4.2) 的功能。
- rshift_bit 是矩阵乘积的右移数，当 C 是 int8 或者 int16 时才有效，由于矩阵的乘积有可能会超出 8-bit 或者 16-bit 的范围，所以用户可以配置一定的右移数，通过舍弃部分精度来防止溢出。
- alpha 和 beta 是 float32 的常系数，当 C 是 float32 时才有效。

接口的格式如下：

```
bm_status_t bmcv_matmul(bm_handle_t      handle,
                        int              M,
                        int              N,
                        int              K,
                        bm_device_mem_t  A,
                        bm_device_mem_t  B,
                        bm_device_mem_t  C,
                        int              A_sign,
                        int              B_sign,
                        int              rshift_bit,
                        int              result_type,
                        bool             is_B_trans,
                        float            alpha = 1,
                        float            beta = 0);
```

输入参数说明：

- bm_handle_t handle

  输入参数。bm_handle 句柄

- int M

  输入参数。矩阵 A 和矩阵 C 的行数

- int N

  输入参数。矩阵 B 和矩阵 C 的列数

- int K

  输入参数。矩阵 A 的列数和矩阵 B 的行数

- bm_device_mem_t A

  输入参数。根据左矩阵 A 数据存放位置保存其 device 地址或者 host 地址。如果数据存放于 host 空间则内部会自动完成 s2d 的搬运

- bm_device_mem_t B

  输入参数。根据右矩阵 B 数据存放位置保存其 device 地址或者 host 地址。如果数据存放于 host 空间则内部会自动完成 s2d 的搬运。

- bm_device_mem_t C

  输出参数。根据矩阵 C 数据存放位置保存其 device 地址或者 host 地址。如果是 host 地址，则当 beta 不为 0 时，计算前内部会自动完成 s2d 的搬运，计算后再自动完成 d2s 的搬运。

- int A_sign

  输入参数。左矩阵 A 的符号，1 表示有符号，0 表示无符号。

- int B_sign

  输入参数。右矩阵 B 的符号，1 表示有符号，0 表示无符号。

- int rshift_bit

  输入参数。矩阵乘积的右移数，为非负数。只有当 result_type 等于 0 或者 1 时才有效。

- int result_type

  输入参数。输出的结果矩阵数据类型，0 表示是 int8，1 表示 int16, 2 表示 float32。

- bool is_B_trans

  输入参数。输入右矩阵 B 是否需要计算前做转置。

- float alpha

  常系数，输入矩阵 A 和 B 相乘之后再乘上该系数，只有当 result_type 等于 2 时才有效，默认值为 1。

- float beta

  常系数，在输出结果矩阵 C 之前，加上该偏移量，只有当 result_type 等于 2 时才有效，默认值为 0。

返回值说明:

- BM_SUCCESS: 成功

- 其他: 失败

示例代码

```
int M = 3, N = 4, K = 5;
int result_type = 1;
bool is_B_trans = false;
int rshift_bit = 0;
char *A      = new char[M * K];
char *B      = new char[N * K];
short *C      = new short[M * N];
memset(A, 0x11, M * K * sizeof(char));
memset(B, 0x22, N * K * sizeof(char));

bmcv_matmul(handle,
            M,
            N,
            K,
            bm_mem_from_system((void *)A),
            bm_mem_from_system((void *)B),
            bm_mem_from_system((void *)C),
            1,
            1,
            rshift_bit,
            result_type,
            is_B_trans);

delete A;
delete B;
delete C;
```

4.38 bmcv_distance

计算多维空间下多个点与特定一个点的欧式距离，前者坐标存放在连续的 device memory 中，而特定一个点的坐标通过参数传入。坐标值为 float 类型。

接口的格式如下：

```
bm_status_t bmcv_distance(
        bm_handle_t handle,
        bm_device_mem_t input,
        bm_device_mem_t output,
        int dim,
        const float *pnt,
        int len);
```

输入参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄

- bm_device_mem_t input

输入参数。存放 len 个点坐标的 device 空间。其大小为 len*dim*sizeof(float)。

- bm_device_mem_t output

输出参数。存放 len 个距离的 device 空间。其大小为 len*sizeof(float)。

- int dim

输入参数。空间维度大小。

- const float *pnt

输入参数。特定一个点的坐标，长度为 dim。

- int len

输入参数。待求坐标的数量。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

示例代码

```
int L = 1024 * 1024;
int dim = 3;
float pnt[8] = {0};
for (int i = 0; i < dim; ++i)
    pnt[i] = (rand() % 2 ? 1.f : -1.f) * (rand() % 100 + (rand() % 100) *␣
↪0.01);
float *XHost = new float[L * dim];
for (int i = 0; i < L * dim; ++i)
    XHost[i] = (rand() % 2 ? 1.f : -1.f) * (rand() % 100 + (rand() % 100)␣
↪* 0.01);
float *YHost = new float[L];
bm_handle_t handle = nullptr;
bm_dev_request(&handle, 0);
bm_device_mem_t XDev, YDev;
bm_malloc_device_byte(handle, &XDev, L * dim * 4);
bm_malloc_device_byte(handle, &YDev, L * 4);
bm_memcpy_s2d(handle, XDev, XHost);
bmcv_distance(handle,
              XDev,
              YDev,
              dim,
              pnt,
              L));
bm_memcpy_d2s(handle, YHost, YDev));
delete [] XHost;
```

(continues on next page)

(continued from previous page)

```
delete [] YHost;
bm_free_device(handle, XDev);
bm_free_device(handle, YDev);
bm_dev_free(handle);
```

4.39 bmcv_min_max

对于存储于 device memory 中连续空间的一组数据，该接口可以获取这组数据中的最大值和最小值。

接口的格式如下：

```
bm_status_t bmcv_min_max(bm_handle_t handle,
                         bm_device_mem_t input,
                         float *minVal,
                         float *maxVal,
                         int len);
```

输入参数说明：

- bm_handle_t handle

  输入参数。bm_handle 句柄

- bm_device_mem_t input

  输入参数。输入数据的 device 地址。

- float *minVal

  输出参数。运算后得到的最小值结果, 如果为 NULL 则不计算最小值。

- float *maxVal

  输出参数。运算后得到的最大值结果，如果为 NULL 则不计算最大值。

- int len

  输入参数。输入数据的长度。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

示例代码

```
bm_handle_t handle;
bm_status_t dev_ret = bm_dev_request(&handle, dev_id);
int len = 1000;
float max = 0;
float min = 0;
float *input = new float[len];
```

(continues on next page)

(continued from previous page)

```
for (int i = 0; i < 1000; i++) {
    input[i] = (float)(rand() % 1000) / 10.0;
}
bm_device_mem_t input_mem;
bm_malloc_device_byte(handle, input_mem, len * sizeof(float))
bm_memcpy_s2d(handle, input_mem, input);
bmcv_min_max(handle,
             input_mem,
             &min,
             &max,
             len,
             2);
bm_free_device(handle, input_mem);
bm_dev_free(handle);
delete [] input;
```

4.40 bmcv_fft

FFT 运算。完整的使用步骤包括创建、执行、销毁三步。

4.40.1 创建

支持一维或者两维的 FFT 计算，其区别在于创建过程中，后面的执行和销毁使用相同的接口。

对于一维的 FFT，支持多 batch 的运算，接口形式如下：

```
bm_status_t bmcv_fft_1d_create_plan(
        bm_handle_t handle,
        int batch,
        int len,
        bool forward,
        void *&plan);
```

输入参数说明：

- bm_handle_t handle

  输入参数。bm_handle 句柄

- int batch

  输入参数。batch 的数量。

- int int

  输入参数。每个 batch 的长度。

- bool forward

  输入参数。是否为正向变换，false 表示逆向变换。

- void *&plan

输出参数。执行阶段需要使用的句柄。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

对于两维 M*N 的 FFT 运算，接口形式如下:

```
bm_status_t bmcv_fft_2d_create_plan(
        bm_handle_t handle,
        int M,
        int N,
        bool forward,
        void *&plan);
```

输入参数说明:

- bm_handle_t handle

输入参数。bm_handle 句柄

- int M

输入参数。第一个维度的大小。

- int N

输入参数。第二个维度的大小。

- bool forward

输入参数。是否为正向变换，false 表示逆向变换。

- void *&plan

输出参数。执行阶段需要使用的句柄。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

4.40.2 执行

使用上述创建后的 plan 就可以开始真正的执行阶段了，支持复数输入和实数输入两种接口，其格式分别如下:

```
bm_status_t bmcv_fft_execute(
        bm_handle_t handle,
        bm_device_mem_t inputReal,
        bm_device_mem_t inputImag,
```

(continues on next page)

(continued from previous page)

```
        bm_device_mem_t outputReal,
        bm_device_mem_t outputImag,
        const void *plan);

bm_status_t bmcv_fft_execute_real_input(
        bm_handle_t handle,
        bm_device_mem_t inputReal,
        bm_device_mem_t outputReal,
        bm_device_mem_t outputImag,
        const void *plan);
```

输入参数说明:

- bm_handle_t handle

输入参数。bm_handle 句柄

- bm_device_mem_t inputReal

输入参数。存放输入数据实数部分的 device memory 空间，对于一维的 FFT，其大小为 batch*len*sizeof(float)，对于两维 FFT，其大小为 M*N*sizeof(float)。

- bm_device_mem_t inputImag

输入参数。存放输入数据虚数部分的 device memory 空间，对于一维的 FFT，其大小为 batch*len*sizeof(float)，对于两维 FFT，其大小为 M*N*sizeof(float)。

- bm_device_mem_t outputReal

输出参数。存放输出结果实数部分的 device memory 空间，对于一维的 FFT，其大小为 batch*len*sizeof(float)，对于两维 FFT，其大小为 M*N*sizeof(float)。

- bm_device_mem_t outputImag

输出参数。存放输出结果虚数部分的 device memory 空间，对于一维的 FFT，其大小为 batch*len*sizeof(float)，对于两维 FFT，其大小为 M*N*sizeof(float)。

- const void *plan

输入参数。创建阶段所得到的句柄。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

4.40.3 销毁

当执行完成后需要销毁所创建的句柄。

```
void bmcv_fft_destroy_plan(bm_handle_t handle, void *plan);
```

4.40.4 示例代码

```
bool realInput = false;
float *XRHost = new float[M * N];
float *XIHost = new float[M * N];
float *YRHost = new float[M * N];
float *YIHost = new float[M * N];
for (int i = 0; i < M * N; ++i) {
    XRHost[i] = rand() % 5 - 2;
    XIHost[i] = realInput ? 0 : rand() % 5 - 2;
}
bm_handle_t handle = nullptr;
bm_dev_request(&handle, 0);
bm_device_mem_t XRDev, XIDev, YRDev, YIDev;
bm_malloc_device_byte(handle, &XRDev, M * N * 4);
bm_malloc_device_byte(handle, &XIDev, M * N * 4);
bm_malloc_device_byte(handle, &YRDev, M * N * 4);
bm_malloc_device_byte(handle, &YIDev, M * N * 4);
bm_memcpy_s2d(handle, XRDev, XRHost);
bm_memcpy_s2d(handle, XIDev, XIHost);
void *plan = nullptr;
bmcv_fft_2d_create_plan(handle, M, N, forward, plan);
if (realInput)
    bmcv_fft_execute_real_input(handle, XRDev, YRDev, YIDev, plan);
else
    bmcv_fft_execute(handle, XRDev, XIDev, YRDev, YIDev, plan);
bmcv_fft_destroy_plan(handle, plan);
bm_memcpy_d2s(handle, YRHost, YRDev);
bm_memcpy_d2s(handle, YIHost, YIDev);
bm_free_device(handle, XRDev);
bm_free_device(handle, XIDev);
bm_free_device(handle, YRDev);
bm_free_device(handle, YIDev);
bm_dev_free(handle);
```

4.41 bmcv_calc_hist

4.41.1 直方图

接口形式：

```
bm_status_t bmcv_calc_hist(
        bm_handle_t handle,
        bm_device_mem_t input,
        bm_device_mem_t output,
        int C,
        int H,
        int W,
        const int *channels,
        int dims,
        const int *histSizes,
        const float *ranges,
        int inputDtype);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_device_mem_t input

输入参数。该 device memory 空间存储了输入数据，类型可以是 float32 或者 uint8，由参数 inputDtype 决定。其大小为 C*H*W*sizeof(Dtype)。

- bm_device_mem_t output

输出参数。该 device memory 空间存储了输出结果，类型为 float，其大小为 histSizes[0]*histSizes[1]*……*histSizes[n]*sizeof(float)。

- int C

输入参数。输入数据的通道数量。

- int H

输入参数。输入数据每个通道的高度。

- int W

输入参数。输入数据每个通道的宽度。

- const int *channels

输入参数。需要计算直方图的 channel 列表，其长度为 dims，每个元素的值必须小于 C。

- int dims

输入参数。输出的直方图维度，要求不大于 3。

- const int *histSizes

输入参数。对应每个 channel 统计直方图的份数，其长度为 dims。

- const float *ranges

输入参数。每个通道参与统计的范围，其长度为 2*dims。

- int inputDtype

输入参数。输入数据的类型：0 表示 float，1 表示 uint8。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

代码示例：

```
int H = 1024;
int W = 1024;
int C = 3;
int dim = 3;
int channels[3] = {0, 1, 2};
int histSizes[] = {15000, 32, 32};
float ranges[] = {0, 1000000, 0, 256, 0, 256};
int totalHists = 1;
for (int i = 0; i < dim; ++i)
    totalHists *= histSizes[i];
bm_handle_t handle = nullptr;
bm_status_t ret = bm_dev_request(&handle, 0);
float *inputHost = new float[C * H * W];
float *outputHost = new float[totalHists];
for (int i = 0; i < C; ++i)
    for (int j = 0; j < H * W; ++j)
        inputHost[i * H * W + j] = static_cast<float>(rand() % 1000000);
if (ret != BM_SUCCESS) {
    printf("bm_dev_request failed. ret = %d\n", ret);
    exit(-1);
}
bm_device_mem_t input, output;
ret = bm_malloc_device_byte(handle, &input, C * H * W * 4);
if (ret != BM_SUCCESS) {
    printf("bm_malloc_device_byte failed. ret = %d\n", ret);
    exit(-1);
}
ret = bm_memcpy_s2d(handle, input, inputHost);
if (ret != BM_SUCCESS) {
    printf("bm_memcpy_s2d failed. ret = %d\n", ret);
    exit(-1);
}
ret = bm_malloc_device_byte(handle, &output, totalHists * 4);
if (ret != BM_SUCCESS) {
```

(continues on next page)

(continued from previous page)

```
    printf("bm_malloc_device_byte failed. ret = %d\n", ret);
    exit(-1);
}
ret = bmcv_calc_hist(handle,
                     input,
                     output,
                     C,
                     H,
                     W,
                     channels,
                     dim,
                     histSizes,
                     ranges,
                     0);
if (ret != BM_SUCCESS) {
    printf("bmcv_calc_hist failed. ret = %d\n", ret);
    exit(-1);
}
ret = bm_memcpy_d2s(handle, outputHost, output);
if (ret != BM_SUCCESS) {
    printf("bm_memcpy_d2s failed. ret = %d\n", ret);
    exit(-1);
}
bm_free_device(handle, input);
bm_free_device(handle, output);
bm_dev_free(handle);
delete [] inputHost;
delete [] outputHost;
```

4.41.2 带权重的直方图

接口形式：

```
bm_status_t bmcv_calc_hist_with_weight(
        bm_handle_t handle,
        bm_device_mem_t input,
        bm_device_mem_t output,
        const float *weight,
        int C,
        int H,
        int W,
        const int *channels,
        int dims,
        const int *histSizes,
        const float *ranges,
        int inputDtype);
```

参数说明：

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_device_mem_t input

输入参数。该 device memory 空间存储了输入数据，其大小为 C*H*W*sizeof(Dtype)。

- bm_device_mem_t output

输出参数。该 device memory 空间存储了输出结果，类型为 float，其大小为 histSizes[0]*histSizes[1]*……*histSizes[n]*sizeof(float)。

- const float *weight

输入参数。channel 内部每个元素在统计直方图时的权重，其大小为 H*W*sizeof(float)，如果所有值全为 1 则与普通直方图功能相同。

- int C

输入参数。输入数据的通道数量。

- int H

输入参数。输入数据每个通道的高度。

- int W

输入参数。输入数据每个通道的宽度。

- const int *channels

输入参数。需要计算直方图的 channel 列表，其长度为 dims，每个元素的值必须小于 C。

- int dims

输入参数。输出的直方图维度，要求不大于 3。

- const int *histSizes

输入参数。对应每个 channel 统计直方图的份数，其长度为 dims。

- const float *ranges

输入参数。每个通道参与统计的范围，其长度为 2*dims。

- int inputDtype

输入参数。输入数据的类型：0 表示 float，1 表示 uint8。

返回值说明：

- BM_SUCCESS: 成功
- 其他: 失败

4.42 bmcv_nms

该接口用于消除网络计算得到过多的物体框，并找到最佳物体框。

接口形式:

```
bm_status_t bmcv_nms(bm_handle_t handle,
                     bm_device_mem_t input_proposal_addr,
                     int proposal_size,
                     float nms_threshold,
                     bm_device_mem_t output_proposal_addr)
```

参数说明:

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_device_mem_t input_proposal_addr

输入参数。输入物体框数据所在地址，输入物体框数据结构为 face_rect_t, 详见下面数据结构说明。需要调用 bm_mem_from_system() 将数据地址转化成转化为 bm_device_mem_t 所对应的结构。

- int proposal_size

输入参数。物体框个数。

- float nms_threshold

输入参数。过滤物体框的阈值，分数小于该阈值的物体框将会被过滤掉。

- bm_device_mem_t output_proposal_addr

输出参数。输出物体框数据所在地址，输出物体框数据结构为 nms_proposal_t，详见下面数据结构说明。需要调用 bm_mem_from_system() 将数据地址转化成转化为 bm_device_mem_t 所对应的结构。

返回值:

- BM_SUCCESS: 成功
- 其他: 失败

数据类型说明:

```
typedef struct
{
    float x1;
    float y1;
    float x2;
    float y2;
    float score;
}face_rect_t;
```

face_rect_t 描述了一个物体框坐标位置以及对应的分数。

- x1 描述了物体框左边缘的横坐标

- y1 描述了物体框上边缘的纵坐标
- x2 描述了物体框右边缘的横坐标
- y2 描述了物体框下边缘的纵坐标
- score 描述了物体框对应的分数

```
typedef struct
{
    face_rect_t face_rect[MAX_PROPOSAL_NUM];
    int size;
    int capacity;
    face_rect_t *begin;
    face_rect_t *end;
} nms_proposal_t;
```

nms_proposal_t 描述了输出物体框的信息。

- face_rect 描述了经过过滤后的物体框信息
- size 描述了过滤后得到的物体框个数
- capacity 描述了过滤后物体框最大个数
- begin 暂不使用
- end 暂不使用

代码示例:

```
face_rect_t *proposal_rand = new face_rect_t[MAX_PROPOSAL_NUM];
nms_proposal_t *output_proposal = new nms_proposal_t;
int proposal_size =32;
float nms_threshold = 0.2;
for (int i = 0; i < proposal_size; i++)
{
    proposal_rand[i].x1 = 200;
    proposal_rand[i].x2 = 210 ;
    proposal_rand[i].y1 = 200;
    proposal_rand[i].y2 = 210;
    proposal_rand[i].score = 0.23;
}
bmcv_nms(handle,
         bm_mem_from_system(proposal_rand),
         proposal_size,
         nms_threshold,
         bm_mem_from_system(output_proposal));
delete[] proposal_rand;
delete output_proposal;
```

注意事项:

该 api 可输入的最大 proposal 数为 56000。

4.43 bmcv_nms_ext

该接口是 bmcv_nms 接口的广义形式，支持 Hard_NMS/Soft_NMS/Adaptive_NMS/SSD_NMS，用于消除网络计算得到过多的物体框，并找到最佳物体框。

接口形式:

```
bm_status_t bmcv_nms_ext(bm_handle_t     handle,
                 bm_device_mem_t input_proposal_addr,
                 int             proposal_size,
                 float           nms_threshold,
                 bm_device_mem_t output_proposal_addr,
                 int             topk,
                 float           score_threshold,
                 int             nms_alg,
                 float           sigma,
                 int             weighting_method,
                 float *         densities,
                 float           eta)
```

参数说明:

- bm_handle_t handle

输入参数。bm_handle 句柄。

- bm_device_mem_t input_proposal_addr

输入参数。输入物体框数据所在地址，输入物体框数据结构为 face_rect_t, 详见下面数据结构说明。需要调用 bm_mem_from_system() 将数据地址转化成转化为 bm_device_mem_t 所对应的结构。

- int proposal_size

输入参数。物体框个数。

- float nms_threshold

输入参数。过滤物体框的阈值，分数小于该阈值的物体框将会被过滤掉。

- bm_device_mem_t output_proposal_addr

输出参数。输出物体框数据所在地址，输出物体框数据结构为 nms_proposal_t，详见下面数据结构说明。需要调用 bm_mem_from_system() 将数据地址转化成转化为 bm_device_mem_t 所对应的结构。

- int topk

输入参数。当前未使用，为后续可能的的扩展预留的接口。

- float score_threshold

输入参数。当使用 Soft_NMS 或者 Adaptive_NMS 时，最低的 score threshold。当 score 低于该值时，score 所对应的框将被过滤掉。

- int nms_alg

输入参数。不同的 NMS 算法的选择，包括 Hard_NMS/Soft_NMS/Adaptive_NMS/SSD_NMS。

- float sigma

  输入参数。当使用 Soft_NMS 或者 Adaptive_NMS 时，Gaussian re-score 函数的参数。

- int weighting_method

  输入参数。当使用 Soft_NMS 或者 Adaptive_NMS 时，re-score 函数选项：包括线性权值和 Gaussian 权值。可选参数：

```
typedef enum {
    LINEAR_WEIGHTING = 0,
    GAUSSIAN_WEIGHTING,
    MAX_WEIGHTING_TYPE
} weighting_method_e;
```

线性权值表达式如下：

$$s_i = \begin{cases} s_i, & iou(\mathcal{M}, b_i) < N_t \\ s_i \times (1 - iou(\mathcal{M}, b_i)), & iou(\mathcal{M}, b_i) \geq N_t \end{cases}$$

Gaussian 权值表达式如下：

$$s_i = s_i \times e^{-iou(\mathcal{M},\, b_i)^2/\sigma}$$

上面两个表达式中，$\mathcal{M}$ 表示当前 score 最大的物体框，b_i 表示其他 score 比 $\mathcal{M}$ 低的物体框，s_i 表示其他 score 比 $\mathcal{M}$ 低的物体框的 score 值，N_t 表示 NMS 门限，σ 对应本接口的参数 float sigma。

- float* densities

  输入参数。Adaptive-NMS 密度值。

- float eta

  输入参数。SSD-NMS 系数，用于调整 iou 阈值。

  返回值:

- BM_SUCCESS: 成功
- 其他: 失败

  代码示例:

```
unsigned int seed1 = 100;
bm_nms_data_type_t  nms_threshold = 0.22;
bm_nms_data_type_t  nms_score_threshold = 0.22;
bm_nms_data_type_t  sigma               = 0.4;
int  proposal_size       = 0;
int  rand_loop_num       = 10;
int  weighting_method    = GAUSSIAN_WEIGHTING;
std::function<float(float, float)> weighting_func;
int  nms_type = SOFT_NMS;
const int soft_nms_total_types = MAX_NMS_TYPE - HARD_NMS - 1;
for (int rand_loop_idx = 0;
```

(continues on next page)

(continued from previous page)

```
    rand_loop_idx < (rand_loop_num * soft_nms_total_types);
    rand_loop_idx++) {
   for (int rand_mode = 0; rand_mode < MAX_RAND_MODE; rand_mode++) {
       soft_nms_gen_test_size(nms_score_threshold,
           nms_threshold,
           sigma,
           proposal_size,
           weighting_method,
           rand_mode);
       nms_type = rand_loop_idx % soft_nms_total_types + HARD_NMS + 1;
       if (nms_type == ADAPTIVE_NMS) {
           std::cout << "[ADAPTIVE NMS] rand_mode : " << rand_mode
                     << std::endl;
       } else if (nms_type == SOFT_NMS) {
           std::cout << "[SOFT NMS] rand_mode : " << rand_mode
                     << std::endl;
       } else if (nms_type == SSD_NMS) {
           std::cout << "[SSD NMS] rand_mode : " << rand_mode <<␣
↪std::endl;
       } else {
           std::cout << "nms type error" << std::endl;
           exit(-1);
       }
       std::cout << "nms_threshold: " << nms_threshold << std::endl;
       std::cout << "nms_score_threshold: " << nms_score_threshold
                 << std::endl;
       std::cout << "proposal size: " << proposal_size << std::endl;
       std::cout << "sigma: " << sigma << std::endl;
       std::shared_ptr<Blob<face_rect_t>> proposal_rand =
           std::make_shared<Blob<face_rect_t>>(MAX_PROPOSAL_NUM);
       std::shared_ptr<nms_proposal_t> output_proposal =
           std::make_shared<nms_proposal_t>();

       std::vector<face_rect_t>        proposals_ref;
       std::vector<face_rect_t>        nms_proposal;
       std::vector<bm_nms_data_type_t> score_random_buf;
       std::vector<bm_nms_data_type_t> density_vec;
       std::shared_ptr<Blob<float>>    densities =
           std::make_shared<Blob<float>>(proposal_size);
       generate_random_buf<bm_nms_data_type_t>(
           score_random_buf, 0, 1, 10000);
       face_rect_t *proposal_rand_ptr = proposal_rand.get()->data;
       float eta = ((float)(rand() % 10)) / 10;
       for (int32_t i = 0; i < proposal_size; i++) {
           proposal_rand_ptr[i].x1 =
               ((bm_nms_data_type_t)(rand() % 100)) / 10;
           proposal_rand_ptr[i].x2 = proposal_rand_ptr[i].x1
```

(continues on next page)

(continued from previous page)

```
                + ((bm_nms_data_type_t)(rand() % 100)) / 10;
            proposal_rand_ptr[i].y1 =
                ((bm_nms_data_type_t)(rand() % 100)) / 10;
            proposal_rand_ptr[i].y2 = proposal_rand_ptr[i].y1
                + ((bm_nms_data_type_t)(rand() % 100)) / 10;
            proposal_rand_ptr[i].score = score_random_buf[i];
            proposals_ref.push_back(proposal_rand_ptr[i]);
            densities.get()->data[i] = ((float)(rand() % 100)) / 100;
        }
        assert(proposal_size <= MAX_PROPOSAL_NUM);
        if (weighting_method == LINEAR_WEIGHTING) {
            weighting_func = linear_weighting;
        } else if (weighting_method == GAUSSIAN_WEIGHTING) {
            weighting_func = gaussian_weighting;
        } else {
            std::cout << "weighting_method error: " << weighting_method
                      << std::endl;
        }
        bmcv_nms_ext(handle,
                     bm_mem_from_system(proposal_rand.get()->data),
                     proposal_size,
                     nms_threshold,
                     bm_mem_from_system(output_proposal.get()),
                     1,
                     nms_score_threshold,
                     nms_type,
                     sigma,
                     weighting_method,
                     densities.get()->data,
                     eta);
    }
}

return GEN_PROPOSAL_SUCCESS;
```

注意事项:

该 api 可输入的最大 proposal 数为 1024。

CHAPTER

FIVE

PCIE CPU

5.1 PCIe CPU

对于不方便使用 TPU 加速的操作，需要 CPU 配合来完成。

如果是 SoC 模式，host 端即为片上的 ARM A53 处理器，由它来完成 CPU 操作。

如果是 PCIe 模式，host 端为用户的主机，CPU 操作可以选择在 host 端完成，也可以使用片上的 ARM A53 处理器来完成。两种实现方式各有优缺点：前者需要在 device 和 host 之间搬运输入输出数据，但运算性能可能优于 ARM，所以用户可以根据自身 host 处理器性能、负载等实际情况选择最优的方式。默认情况下为前者，如果需要使用片上处理器可按照以下方式开启。

5.1.1 准备工作

如果要使能片上处理器，那么需要以下两个文件：

- ramboot_rootfs.itb
- fip.bin

需要将这两个文件所在的路径设置到程序运行的环境变量 BMCV_CPU_KERNEL_PATH 中，如下：

$ export BMCV_CPU_KERNEL_PATH=/path/to/kernel_fils/

BMCV 所有需要 CPU 操作的实现均在库 libbmcv_cpu_func.so 中，需要将该文件所在路径添加到程序运行的环境变量 BMCV_CPU_LIB_PATH 中，如下：

$ export BMCV_CPU_LIB_PATH=/path/to/lib/

目前需要 CPU 参与实现的 API 如下所示，如果没有使用以下 API 可忽略该功能。

num	API
1	bmcv_image_draw_lines
2	bmcv_image_erode
3	bmcv_image_dilate
4	bmcv_image_lkpyramid_execute

5.1.2 开启和关闭

用户可以在程序的开始结束处分别使用以下两个接口，即可分别实现该功能的开启和关闭。

```
bm_status_t bmcv_open_cpu_process(bm_handle_t handle);

bm_status_t bmcv_close_cpu_process(bm_handle_t handle);
```

传入参数说明:

- bm_handle_t handle

  输入参数。bm_handle 句柄。

返回值说明:

- BM_SUCCESS: 成功
- 其他: 失败

CHAPTER

SIX

LEGACY API

6.1 bmcv_image

以下数据结构以及接口仅为了兼容上一代产品，并且其功能在前边章节中均可实现，在 BM1684 系列产品中不建议使用。

BM1682 的 BMCV 库对常用的图像相关的数据结构进行了封装，主要定义了 image 相关的数据结构。

定义以下:

```
#define MAX_BMCV_IMAGE_CHANNEL 4
typedef enum bmcv_data_format_{
    DATA_TYPE_FLOAT = 0,  // data type is 32 bit float
    DATA_TYPE_BYTE = 4  // data type is 8 bit
}bmcv_data_format;

typedef enum bmcv_color_space_{
    COLOR_YUV,  // color space is yuv
    COLOR_YCbCr, // color space is YCbCr
    COLOR_RGB  // color space is rgb
}bmcv_color_space;

typedef enum bmcv_image_format_{
    YUV420P,  // data is in YUV420 packed format
    NV12,  // data is in NV12 format
    NV21,  // data is in NV21 format
    RGB,  // data is in RGB planar format
    BGR,  // data is in BGR planar format
    RGB_PACKED, // data is in RGB packed format
    BGR_PACKED, // data is in BGR packed format
    BGR4N  // data is in BGR 4N mode, for future use
}bmcv_image_format;

typedef struct bmcv_image_t{
    bmcv_color_space color_space;
    bmcv_data_format data_format;
    bmcv_image_format image_format;
```

(continues on next page)

(continued from previous page)

```
    int               image_width;
    int               image_height;
    bm_device_mem_t   data[MAX_BMCV_IMAGE_CHANNEL];
    int               stride[MAX_BMCV_IMAGE_CHANNEL];
}bmcv_image;
```

各个参数说明（bm_image 中有类似的参数定义）：

- bmcv_color_space color_space

  表示图像颜色空间

- bmcv_data_format data_format

  表示图像数据类型

- bmcv_image_format image_format

  表示图像格式

- int image_width

  表示图像宽

- int image_height

  表示图像高

- bm_device_mem_t data[MAX_BMCV_IMAGE_CHANNEL]

  表示图像各通道数据地址

- int stride[MAX_BMCV_IMAGE_CHANNEL]

  表示图像各通道 stride

6.2 bmcv_img_bgrsplit

此接口用于 BM1682 对 bgr 数据的从 interleave 排列重排为 plannar 排列。可以使用 bmcv_image_vpp_convert 接口实现该功能。

```
bm_status_t bmcv_img_bgrsplit(
    bm_handle_t     handle,
    bmcv_image          input,
    bmcv_image          output
);
```

传入参数说明:

- bm_handle_t handle

  设备环境句柄, 通过调用 bm_dev_request 获取

- bmcv_image input

输入待处理的图像数据，bgr 数据，packed 排列方式

bmcv_image output

输出待处理的图像数据，bgr 数据，plannar 排列方式

6.3 bmcv_img_crop

此接口用于 BM1682 从 bgr 图像中剪切一部分形成新的图像

```
bm_status_t bmcv_img_crop(
    bm_handle_t       handle,
    bmcv_image        input,
    int               input_c,
    int               top,
    int               left,
    bmcv_image        output
);
```

传入参数说明:

- bm_handle_t handle
- 设备环境句柄, 通过调用 bm_dev_request 获取
- bmcv_image input

输入图像，只支持 BGR planar 或者 RGB planar 格式，输入输出的 BGR 或者 RGB 顺序一致不做交换。输入输出可以选择数据类型为 float 或者 byte

- int input_c

  输入数据颜色数，可以为 1 或者 3，相应输出颜色数也会是 1 或者 3

- int top

  剪切开始点纵坐标，图片左上角为原点，单位为像素

- int left

  剪切开始点横坐标，图片左上角为原点，单位为像素

- bmcv_image output

  输出的图像的描述结构

6.4 bmcv_img_scale

此接口用于 BM1682 对图像进行缩放，并支持从原图中抠图缩放, 以及对输出图像进行归一化操作

```
bm_status_t bmcv_img_scale(
    bm_handle_t handle,
    bmcv_image input,
    int n, int do_crop, int top, int left, int height, int width,
    unsigned char stretch, unsigned char padding_b, unsigned char padding_
↪g, unsigned char padding_r,
    int pixel_weight_bias,
    float weight_b, float bias_b,
    float weight_g, float bias_g,
    float weight_r, float bias_r,
    bmcv_image output
);
```

传入参数说明:

- bm_handle_t handle

  bmcv 初始化获取的 handle

- bmcv_image input

  输入图像数据地址, 只支持 BGR planar 或者 RGB planar 格式数据，支持 float 或者 byte 类型数据

- int n

  处理图像的数量，目前只支持 1

- int do_crop

  是否进行抠图缩放。0：不抠图，下面四个参数无用。1：抠图，下面四个参数用于确定抠图的尺寸

- int top

  抠图开始点纵坐标，图片左上角为原点，单位为像素

- int left

  抠图开始点横坐标，图片左上角为原点，单位为像素

- int height

  抠图高度，单位为像素

- int width

  抠图宽度，单位为像素

- unsigned char stretch

  scale 模式。0：fit 模式，等比例缩放，按照下面的 padding 参数填充颜色；1：拉伸模式，填满目标大小，下面的 padding 参数无用

- unsigned char padding_b

fit 模式下蓝色填充色值，为 0-255 之 byte 类型，如果输出类型为 byte 或 float 但未指定归一化计算，则此值为填充颜色分量，如果输出为 float 类型且需要归一化，则此分量也相应进行归一化计算。

- unsigned char padding_g

fit 模式下绿色填充色值，为 0-255 之 byte 类型，如果输出类型为 byte 或 float 但未指定归一化计算，则此值为填充颜色分量，如果输出为 float 类型且需要归一化，则此分量也相应进行归一化计算。

- unsigned char padding_r

fit 模式下红色填充色值，为 0-255 之 byte 类型，如果输出类型为 byte 或 float 但未指定归一化计算，则此值为填充颜色分量，如果输出为 float 类型且需要归一化，则此分量也相应进行归一化计算。

- int pixel_weight_bias

  是否对输出图像进行归一化计算。0：不做归一化计算，下面 weight 和 bias 参数无用；1：进行归一化计算

- float weight_b

  b 通道的系数

- float bias_b

  b 通道偏移量

- float weight_g

  g 通道的系数

- float bias_g

  g 通道的偏移量

- float weight_r

  r 通道的系数

- float bias_r

  r 通道的偏移量

- bmcv_image ouput

  输出的图像的描述结构，支持 BGR planar 或者 RGB planar 格式，但是必须与输入相同。

6.5 bmcv_img_transpose

此接口用于 BM1682 对 bgr 格式图像的转置，将图像按照矩阵转置的方式转置

```
bm_status_t bmcv_img_transpose(
    bm_handle_t     handle,
        bmcv_image              input,
        bmcv_image              output
);
```

传入参数说明:

- bm_handle_t handle
- 设备环境句柄, 通过调用 bm_dev_request 获取
- bmcv_image input

  输入图像，支持 BGR planar 或者 RGB planar 类型的 float 或 byte 类型

- bmcv_image output

  输出图像，支持 BGR planar 或者 RGB planar 类型的 float 或 byte 类型，BGR 摆放顺序与输入相同

6.6 bmcv_img_yuv2bgr

此函数用于 BM1682 将 NV12 格式的图像数据转换为 BGR 格式的数据

```
bm_status_t bm_img_yuv2bgr(
    bm_handle_t      handle,
    bmcv_image       input,
    bmcv_image       output
);
```

传入参数说明:

- bm_handle_t handle
- 设备环境句柄, 通过调用 bm_dev_request 获取
- bmcv_image input

  输入的图像描述结构, 当前只支持 NV12 格式的输入图像，颜色空间支持 YUV 和 YCbCr

- bmcv_image output

  输出的图像的描述结构，输出支持 BGR planar/packed 或者 RGB planar/packed 格式摆放。输出可选 float 类型或者 byte 类型