NVIDIA

NvCVImage

API Guide

DU-10464-001_v1.1 | January 2023

Table of Contents

Chapter 1. About the NvCVImage APIs

NvCVImage provides a rich descriptor and optimized functionality for a wide variety of images.

- All function available to C and C++.
- Buffers on CPU or GPU.
- Many pixel formats, for example, RGB, BGRA, grayscale, YUV420, and so on.
- Many component types, for example, u8, f32, f16, and so on.
- Chunky (interleaved), planar, and semi-planar arrangements.
- Buffer allocation, reallocation and deallocation.
- Image transfer.
- Conversion between different image formats, types and arrangements.
- Composition of one image with another, controlled by a matte.
- Operations on sub-rectangles of images.
- Image wrappers for zero-copy conversion between NvCVImage and other image representations.

Chapter 2. Images

This section provides information about how to work with images.

2.1. Working with Image Frames on GPU or CPU Buffers

AI filters in the NVIDIA® AR and Video Effects SDKs accept image buffers as `NvCVImage` objects. The image buffers can be CPU or GPU buffers, but for performance reasons, the AI filters require GPU buffers.

The SDKs provide the functions to convert an image representation to `NvCVImage` and transfer images between CPU and GPU buffers.

2.2. Converting Image Representations to NvCVImage Objects

The AR and Video Effects SDKs provide functions to convert OpenCV images and other image representations to `NvCVImage` objects.

Each function places a wrapper around an existing buffer. The wrapper prevents the buffer from being freed when the destructor of the wrapper is called.

2.2.1. Converting OpenCV Images to NvCVImage Objects

You can use the wrapper functions that the SDKs provide specifically for RGB OpenCV images.

> **Note:** The AR and Video Effects SDKs provide wrapper functions only for RGB images. No wrapper functions are provided for YUV images.

- To create an `NvCVImage` object wrapper for an OpenCV image, use the `NVWrapperForCVMat()` function.

```
//Allocate source and destination OpenCV images
cv::Mat srcCVImg(   );
cv::Mat dstCVImg(...);
```

```
// Declare source and destination NvCVImage objects
NvCVImage srcCPUImg;
NvCVImage dstCPUImg;

NVWrapperForCVMat(&srcCVImg, &srcCPUImg);
NVWrapperForCVMat(&dstCVImg, &dstCPUImg);
```

- To create an OpenCV image wrapper for an `NvCVImage` object, use the `CVWrapperForNvCVImage()` function.

```
// Allocate source and destination NvCVImage objects
NvCVImage srcCPUImg(...);
NvCVImage dstCPUImg(...);

//Declare source and destination OpenCV images
cv::Mat srcCVImg;
cv::Mat dstCVImg;

CVWrapperForNvCVImage (&srcCPUImg, &srcCVImg);
CVWrapperForNvCVImage (&dstCPUImg, &dstCVImg);
```

2.2.2. Converting Image Frames on GPU or CPU buffers to NvCVImage Objects

Call the `NvCVImage_Init()` function to place a wrapper around an existing buffer (`srcPixelBuffer`).

```
NvCVImage src_gpu;
vfxErr = NvCVImage_Init(&src_gpu, 640, 480, 1920, srcPixelBuffer, NVCV_BGR, NVCV_U8,
 NVCV_INTERLEAVED, NVCV_GPU);

NvCVImage src_cpu;
vfxErr = NvCVImage_Init(&src_cpu, 640, 480, 1920, srcPixelBuffer, NVCV_BGR, NVCV_U8,
 NVCV_INTERLEAVED, NVCV_CPU);
```

2.2.3. Converting an NvCVImage Object to a Buffer that can be Encoded by NvEncoder

To convert the `NvCVImage` to the pixel format that is used during encoding via NvEncoder, you can call the `NvCVImage_Transfer()` function. The following sample shows a frame that is encoded in the BGRA pixel format.

```
//BGRA frame is 4-channel, u8 buffer residing on the GPU
NvCVImage BGRA_frame;
NvCVImage_Alloc(&BGRA_frame, dec.GetWidth(), dec.GetHeight(), NVCV_BGRA, NVCV_U8,
 NVCV_CHUNKY, NVCV_GPU, 1);

//Initialize encoder with a BGRA output pixel format
using NvEncCudaPtr = std::unique_ptr<NvEncoderCuda,
 std::function<void(NvEncoderCuda*)>>;
NvEncCudaPtr pEnc(new NvEncoderCuda(cuContext, dec.GetWidth(), dec.GetHeight(),
 NV_ENC_BUFFER_FORMAT_ARGB));
pEnc->CreateEncoder(&initializeParams);
//...

std::vector<std::vector<uint8_t>> vPacket;
//Get the address of the next input frame from the encoder
const NvEncInputFrame* encoderInputFrame = pEnc->GetNextInputFrame();

//Copy the pixel data from BGRA_frame into the input frame address obtained above
NvEncoderCuda::CopyToDeviceFrame(cuContext,
```

```
                        BGRA_frame.pixels,
                     BGRA_frame.pitch,
                        (CUdeviceptr)encoderInputFrame->inputPtr,
                        encoderInputFrame->pitch,
                     pEnc->GetEncodeWidth(),
                        pEnc->GetEncodeHeight(),
                        CU_MEMORYTYPE_DEVICE,
                        encoderInputFrame->bufferFormat,
                        encoderInputFrame->chromaOffsets,
                     encoderInputFrame->numChromaPlanes);
pEnc->EncodeFrame(vPacket);
```

2.2.4. Converting Decoded Frames from the NvDecoder to NvCVImage Objects

Call the `NvCVImage_Transfer()` function to convert the decoded frame that is provided by the NvDecoder from the decoded pixel format to the format that is required by a feature of the SDKs.

The following sample shows a decoded frame that was converted from the NV12 to the BGRA pixel format.

```
NvCVImage decoded_frame, BGRA_frame, stagingBuffer;
NvDecoder dec;

//Initialize decoder...
//Assuming dec.GetOutputFormat() == cudaVideoSurfaceFormat_NV12

//Initialize memory for decoded frame
NvCVImage_Init(&decoded_frame, dec.GetWidth(), dec.GetHeight(),
 dec.GetDeviceFramePitch(), NULL, NVCV_YUV420, NVCV_U8, NVCV_NV12, NVCV_GPU, 1);
decoded_frame.colorSpace = NVCV_709 | NVCV_VIDEO_RANGE | NVCV_CHROMA_ COSITED;

//Allocate memory for BGRA frame
NvCVImage_Alloc(&BGRA_frame, dec.GetWidth(), dec.GetHeight(), NVCV_BGRA, NVCV_U8,
 NVCV_CHUNKY, NVCV_GPU, 1);

decoded_frame.pixels = (void*)dec.GetFrame();

//Convert from decoded frame format(NV12) to desired format(BGRA)
NvCVImage_Transfer(&decoded_frame, &BGRA_frame, 1.f, stream, & stagingBuffer);
```

Note:

The sample above assumes the typical colorspace specification for HD content. SD typically uses NVCV_601. There are 8 possible combinations, and you should use the one that matches your video as described in the video header or proceed by trial and error.

Here is some additional information:

- If the colors are incorrect, swap 709<->601.
- If they are washed out or blown out, swap VIDEO<->FULL.
- If the colors are shifted horizontally, swap INTSTITIAL<->COSITED.

2.3. Allocating an NvCVImage Object Buffer

You can allocate the buffer for an `NvCVImage` object by using the `NvCVImage` allocation constructor or image functions. In both options, the buffer is automatically freed by the destructor when the images go out of scope.

2.3.1. Using the NvCVImage Allocation Constructor to Allocate a Buffer

The NvCVImage allocation constructor creates an object to which memory has been allocated and that has been initialized. Refer Allocation Constructor for more information.

The final three optional parameters of the allocation constructor determine the properties of the resulting NvCVImage object:

- The pixel organization determines whether blue, green, and red are in separate planes or interleaved.
- The memory type determines whether the buffer resides on the GPU or the CPU.
- The byte alignment determines the gap between consecutive scanlines.

The following examples show how to use the final three optional parameters of the allocation constructor to determine the properties of the NvCVImage object.

- This example creates an object without setting the final three optional parameters of the allocation constructor. In this object, the blue, green, and red components interleaved in each pixel, the buffer resides on the CPU, and the byte alignment is the default alignment.

```
NvCVImage cpuSrc(
  srcWidth,
  srcHeight,
  NVCV_BGR,
  NVCV_U8
);
```

- This example creates an object with identical pixel organization, memory type, and byte alignment to the previous example by setting the final three optional parameters explicitly. As in the previous example, the blue, green, and red components are interleaved in each pixel, the buffer resides on the CPU, and the byte alignment is the default, that is, optimized for maximum performance.

```
NvCVImage src(
  srcWidth,
  srcHeight,
  NVCV_BGR,
  NVCV_U8,
  NVCV_INTERLEAVED,
  NVCV_CPU,
  0
);
```

- This example creates an object in which the blue, green, and red components are in separate planes, the buffer resides on the GPU, and the byte alignment ensures that no gap exists between one scanline and the next scanline.

```
NvCVImage gpuSrc(
  srcWidth,
  srcHeight,
  NVCV_BGR,
  NVCV_U8,
  NVCV_PLANAR,
  NVCV_GPU,
  1
);
```

2.3.2. Using Image Functions to Allocate a Buffer

By declaring an empty image, you can defer buffer allocation.

1. Declare an empty `NvCVImage` object.

```
NvCVImage xfr;
```

2. Allocate or reallocate the buffer for the image.

    - To allocate the buffer, call the `NvCVImage_Alloc()` function.

      Allocate a buffer this way when the image is part of a state structure, where you won't know the size of the image until later.

    - To reallocate a buffer, call `NvCVImage_Realloc()`.

      This function checks for an allocated buffer and reshapes the buffer if it is big enough, before freeing the buffer and calling `NvCVImage_Alloc()`.

2.4. Transferring Images Between CPU and GPU Buffers

If the memory types of the input and output image buffers are different, an application can transfer images between CPU and GPU buffers.

2.4.1. Transferring Input Images from a CPU Buffer to a GPU Buffer

Here is some information about how to transfer input images.

1. To transfer an image from the CPU to a GPU buffer with conversion, given the following code:

```
NvCVImage srcCpuImg(width, height, NVCV_RGB, NVCV_U8, NVCV_INTERLEAVED,
                    NVCV_CPU, 1);
NvCVImage dstGpuImg(width, height, NVCV_BGR, NVCV_F32, NVCV_PLANAR,
                    NVCV_GPU, 1);
```

2. Create an `NvCVImage` object to use as a staging GPU buffer in one of the following ways:

    - To avoid allocating memory in a video pipeline, create a GPU buffer during the initialization phase, with the same dimensions and format as the CPU image.

```
NvCVImage stageImg(srcCpuImg.width, srcCpuImg.height,
       srcCpuImg.pixelFormat, srcCpuImg.componentType,
       srcCpuImg.planar, NVCV_GPU);
```

- To simplify your application program code, you can declare an empty staging buffer during the initialization phase.

  ```
  NvCVImage stageImg;
  ```

  An appropriately sized buffer will be allocated or reallocated as needed, if needed.

3. Call the `NvCVImage_Transfer()` function to copy the source CPU buffer contents into the final GPU buffer via the staging GPU buffer.

```
// Transfer the image from the CPU to the GPU, perhaps with conversion.
NvCVImage_Transfer(&srcCpuImg, &dstGpuImg, 1.0f, stream, &stageImg);
```

2.4.2. Transferring Output Images from a GPU Buffer to a CPU Buffer

Here are the steps to transfer output images from a GPU to a CPU buffer.

1. To transfer an image from the GPU to a CPU buffer with conversion, given the following code:

```
NvCVImage srcGpuImg(width, height, NVCV_BGR, NVCV_F32, NVCV_PLANAR,
                 NVCV_GPU, 1);
NvCVImage dstCpuImg(width, height, NVCV_BGR, NVCV_U8, NVCV_INTERLEAVED,
                 NVCV_CPU, 1);
```

2. Create an `NvCVImage` object to use as a staging GPU buffer in one of the following ways:

- To avoid allocating memory in a video pipeline, create a GPU buffer during the initialization phase with the same dimensions and format as the CPU image.

  ```
  NvCVImage stageImg(dstCpuImg.width, dstCpuImg.height,
  dstCPUImg.pixelFormat, dstCPUImg.componentType,
  dstCPUImg.planar, NVCV_GPU);
  ```

- To simplify your application program code, you can declare an empty staging buffer during the initialization phase.

  ```
  NvCVImage stageImg;
  ```

  An appropriately sized buffer will be allocated or reallocated as needed, if needed.

3. Call the `NvCVImage_Transfer()` function to copy the GPU buffer contents into the destination CPU buffer via the staging GPU buffer.

```
// Retrieve the image from the GPU to CPU, perhaps with conversion.
NvCVImage_Transfer(&srcGpuImg, &dstCpuImg, 1.0f, stream, &stageImg);
```

The same staging buffer can be used repeatedly without reallocations in `NvCVImage_Transfer` if it is persistent.

Chapter 3. Enumerations

This section provides information about the enumerations in the NvCVImage APIs.

3.1. NvCVImage_ComponentType

Here is detailed information about `NvCVImage_ComponentType`.

This enumeration defines the data type that is used to represent one component of a pixel.

`NVCV_TYPE_UNKNOWN = 0`
Unknown component data type.

`NVCV_U8 = 1`
Unsigned 8-bit integer.

`NVCV_U16 = 2`
Unsigned 16-bit integer.

`NVCV_S16 = 3`
Signed 16-bit integer.

`NVCV_F16 = 4`
16-bit floating-point number.

`NVCV_U32`
Unsigned 32-bit integer.

`NVCV_S32 = 6`
Signed 32-bit integer.

`NVCV_F32 = 7`
32-bit floating-point number (float).

`NVCV_U64 =8`
Unsigned 64-bit integer.

`NVCV_S64 = 9`
Signed 64-bit integer.

`NVCV_F64 = 10`
64-bit floating-point (double).

3.2. NvCVImage_PixelFormat

This enumeration defines the order of the components in a pixel.

NVCV_FORMAT_UNKNOWN
Unknown pixel format.

NVCV_Y
Luminance (gray).
NVCV_A
Alpha (opaque).
NVCV_YA
Luminance, alpha.
NVCV_RGB
Red, green, blue.
NVCV_BGR
Blue, green, red.
NVCV_RGBA
Red, green, blue, alpha.
NVCV_BGRA
Blue, green, red, alpha.
NVCV_YUV420
Luminance and subsampled Chrominance (Y, Cb, Cr).
NVCV_YUV444
Luminance and full bandwidth Chrominance { Y, Cb, Cr }
NVCV_YUV422
Luminance and subsampled Chrominance (Y, Cb, Cr).

3.2.1. Pixel Organizations

Here is information about how pixels are organized.

The components of the pixels in an image can be organized in the following ways:

- Interleaved pixels (also known as chunky pixels) are compact and are arranged so that the components of each pixel in the image are contiguous.
- Planar pixels are arranged so that the individual components, for example, the red components, of all pixels in the image are grouped together.
- Semi-planar pixels are a mix between interleaved and planar components.
- These types of pixels are found in the video world, where the [Y] component sits in one plane, and the [UV] components are interleaved in another plane.

Typically, pixels are interleaved. However, many neural networks perform better with planar pixels.

In the descriptions of the pixel organizations, square brackets ([]) are used to indicate how groups of pixel components are arranged. For example:

- [VYUY] indicates that groups of V, Y, U and Y components are interleaved.
- [Y][U][V] indicates that the Y, U, and V components of all pixels are grouped.
- [Y][UV] indicates that groups of Y components and groups of U and V components are interleaved.

Refer to YUV pixel formats for more information about YUV pixel formats.

The AR and Video Effects APIs define the following types to specify the pixel organization:

Table 1. Types for Pixel Organization

Type	Value	Description
NVCV_INTERLEAVED	0	
NVCV_CHUNKY	0	Each of these types specifies interleaved, or chunky, pixels in which the components of each pixel in the image are adjacent.
NVCV_PLANAR	1	This type specifies planar pixels in which the individual components of all pixels in the image are grouped.
NVCV_UYVY	2	This type specifies UYVY pixels, which are in the interleaved YUV 4:2:2 format (default for 4:2:2 and default for non-YUV). Pixels are arranged in [UYVY] groups.
NVCV_VYUY	4	This type specifies VYUY pixels, which are in the interleaved YUV 4:2:2 format. Pixels are arranged in [VYUY] groups.
NVCV_YUYV NVCV_YUY2	6	Each of these types specifies YUYV pixels, which are in the interleaved YUV 4:2:2 format. Pixels are arranged in [YUYV] groups.
NVCV_YVYU	8	This type specifies YVYU pixels, which are in the interleaved YUV 4:2:2 format. Pixels are arranged in [YVYU] groups.
NVCV_CYUV	10	This type specifies the interleaved (chunky) YUV 4:4:4 pixels. Pixels are arranged in [YUV] groups.
NVCV_CYVU	12	This type specifies interleaved (chunky) YVU 4:4:4 pixels. Pixels are arranged in [YVU] groups.

Type	Value	Description
NVCV_YUV NVCV_I420 (used with NVCV_YUV420) NVCV_IYUV (used with NVCV_YUV420) NVCV_I444 (used with NVCV_YUV444) NVCV_YM24 (used with NVCV_YUV444)	3	Each of these types specifies planar YUV pixels, with 4:2:0, 4:2:2 or 4:4:4 sampling. Pixels are arranged in [Y], [U], and [V] groups.
NVCV_YVU NVCV_YV12 (used with NVCV_YUV420) NVCV_YM42 (used with NVCV_YUV444)	5	Each of these types specifies YV12 pixels, which are in the planar YUV 4:2:0, YUV 4:2:2 or YUV 4:4:4 formats. Pixels are arranged in [Y], [V], and [U] groups.
NVCV_YCUV NVCV_NV12 (used with NVCV_YUV420) NVCV_NV24 (used with NVCV_YUV444)	7	Each of these types specifies semiplanar YUV pixels, with 4:2:0, 4:2:2 or 4:4:4 sampling. Pixels are arranged in [Y] and [UV] groups, for example, Y in the first plane and U and V interleaved in the second. NV12 is used to refer to semiplanar pixels with the 4:2:0 sampling, and is the most popular YUV format on GPUs.
NVCV_YCVU NVCV_NV21 (used with NVCV_YUV420) NVCV_NV42 (used with NVCV_YUV444)	9	Each of these types specifies semiplanar YVU pixels, with 4:2:0, 4:2:2 or 4:4:4 sampling. Pixels are arranged in [Y] and [VU] groups, for example, Y in the first plane and V and U interleaved in the second. This is similar to the previous layout, except that U and V are swapped.

Note: FlipY is supported only with the planar 4:2:2 formats (UYVY, VYUY, YUYV, and YVYU) and not with other planar or semiplanar formats.

3.2.2. YUV Color Spaces

Here is information about YUV color spaces.

The AR and Video Effects APIs defines the following types to specify the YUV color spaces:

Table 2. Types to Specify the YUV Color Space

Type	Value	Description
NVCV_601	0	This type specifies the Rec.601 YUV color space, which is typically used for standard definition (SD) video.
NVCV_709	1	This type specifies the Rec.709 YUV colorspace, which is typically used for high definition (HD) video.
NVCV_VIDEO_RANGE	0	This type specifies the video range [16, 235].
NVCV_FULL_RANGE	4	This type specifies the video range [0, 255].
NVCV_CHROMA_COSITED NVCV_CHROMA_MPEG2	0	Each of these types specifies a color space in which the chroma is sampled horizontally in the same location as the luma samples. Most video formats use this sampling scheme.
NVCV_CHROMA_INTSTITIAL NVCV_CHROMA_MPEG1	8	Each of these types specifies a color space in which the chroma is sampled horizontally midway between luma samples. This sampling scheme is used for JPEG.

Example: Creating an HD NV12 CUDA buffer

```
NvCVImage *imp = new NvCVImage(1920, 1080, NVCV_YUV420, NVCV_U8,
                            NVCV_NV12, NVCV_CUDA, 0);
imp->colorSpace = NVCV_709 | NVCV_VIDEO_RANGE | NVCV_CHROMA_COSITED;
```

Example: Wrapping an NvCVImage descriptor around an existing HD NV12 CUDA buffer

```
NvCVImage img;
NvCVImage_Init(&img, 1920, 1080, 1920, existingBuffer, NVCV_YUV420, NVCV_U8,
            NVCV_NV12, NVCV_CUDA);
img.colorSpace = NVCV_709 | NVCV_VIDEO_RANGE | NVCV_CHROMA_COSITED;
These are particularly useful and performant for interfacing to the NVDEC video
 decoder.
```

3.2.3. Memory Types

Here is information about the memory types that are available in the AR and Video Effects SDKs.

Image data buffers can be stored in different types of memory, which have different address spaces.

Table 3. Memory Types to Store Image Data Buffers

Type	Description
NVCV_CPU	The buffer is stored in normal CPU memory.
NVCV_CPU_PINNED	The buffer is stored in pinned CPU memory; this can yield higher transfer rates (115%-200%) between the CPU and GPU but should be used sparingly.
NVCV GPU NVCV CUDA	The buffer is stored in CUDA memory.

Chapter 4. APIs

This section provides details about the NvCVImage APIs in the AR and Video Effects SDKs.

4.1. NvCVImage_Alloc

Here is detailed information about `NcCVImage_Alloc`.

```
NvCV_Status NvCVImage_Alloc(
  NvCVImage *im
  unsigned width,
  unsigned height,
  NvCVImage_PixelFormat format,
  NvCVImage_ComponentType type,
  unsigned layout,
  unsigned memSpace,
  unsigned alignment
);
```

Parameters

im [in,out]

Type: `NvCVImage *`

The image to initialize.

width [in]

Type: `unsigned`

The width, in pixels, of the image.

height [in]

Type: `unsigned`

The height, in pixels, of the image.

format [in]

Type: `NvCVImage_PixelFormat`

The format of the pixels.

type [in]

Type: `NvCVImage_ComponentType`

The type of the components of the pixels.

layout [in]

Type: `unsigned`

The organization of the components of the pixels in the image. Refer to Pixel Organizations for more information.

memSpace [in]

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. Refer to Memory Types for more information.

alignment [in]

Type: unsigned

The row byte alignment, which specifies the alignment of the first pixel in each scan line. Set this parameter to 0 or a power of 2.

- 1: Specifies no gap between scan lines.

  A byte alignment of 1 is required by all GPU buffers that are used by many of the video effect filters.
- 0: Specifies the default alignment, which depends on the type of memory in which the image data buffers are stored:
  - CPU memory: Specifies an alignment of 4 bytes.
  - GPU memory: Specifies the alignment set by cudaMallocPitch.
- 2 or greater: Specifies any other alignment, such as a cache line size of 16 or 32 bytes.

  For example, the Upscale effect requires an alignment of 32.

> **Note:** If the product of `width` and the `pixelBytes` member of `NvCVImage` is a whole-number multiple of `alignment`, the gap between scan lines is 0 bytes, regardless of the value of `alignment`.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT` when the pixel format is not supported.
- `NVCV_ERR_MEMORY` when the buffer requires more memory than is available.

Remarks

This function allocates memory for, and initializes, an image. This function assumes that the image data structure has nothing meaningful in it.

This function is called by the C++ NvCVImage constructors. You can call this function from C code to allocate memory for, and to initialize, an empty image.

4.2. NvCVImage_ComponentOffsets

Here is detailed information about `NvCVImage_ComponentOffsets`.

```
void NvCVImage_ComponentOffsets(
  NvCVImage_PixelFormat format,
  int *rOff,
  int *gOff,
  int *bOff,
  int *aOff,
  int *yOff
);
```

Parameters

format [in]

Type: `NvCVImage_PixelFormat`

The pixel format whose component offsets will be retrieved.

rOff [out]

Type: `int *`

The location in which to store the offset for the red channel (can be NULL).

gOff [out]

Type: `int *`

The location in which to store the offset for the green channel (can be NULL).

bOff [out]

Type: `int *`

The location in which to store the offset for the blue channel (can be NULL).

aOff [out]

Type: `int *`

The location in which to store the offset for the alpha channel (can be NULL).

yOff [out]

Type: `int *`

The location in which to store the offset for the luminance channel (can be NULL).

Return Value

Does not return a value.

Remarks

This function gets offsets for the components of a pixel format. These offsets are component, and not byte, offsets. For interleaved pixels, a component offset must be multiplied by the `componentBytes` member of `NvCVImage` to obtain the byte offset.

4.3. NvCVImage_Composite

Here is detailed information about `NvCVImage_Composite`.

```
NvCV_Status NvCVImage_Composite(
  const NvCVImage *fg,
  const NvCVImage *bg,
  const NvCVImage *mat,
  NvCVImage *dst,
  CUstream stream
);
```

Parameters

fg [in]

Type: `const NvCVImage *`

The foreground source, which is an RGB, a BGR, an RGBA, or a BGRA image with u8 or f32 components.

bg [in]

Type: `const NvCVImage *`

The background source, which is is an RGB, a BGR, an RGBA, or a BGRA image with u8 or f32 components.

mat [in]

Type: `const NvCVImage *`

The matte Y or A image with u8 or f32 components, which indicates where the source image should come through.

dst [out]

Type: `NvCVImage *`

The destination image, which can be the same as the `fg` foreground or `bg` background image, or a totally unrelated image.

> **Note:** If the destination image format contains alpha, the alpha channel is not updated in this implementation.

stream [out]

Type: `CUstream`

The CUDA stream on which to transfer the image. If the memory type of both the source and destination images is CPU, this parameter is ignored.

Return Value

- NVCV_SUCCESS on success
- NVCV_ERR_PIXELFORMAT if the pixel format is not supported.

Remarks

This function uses the specified matte image to composite a foreground image over a background image. The fg, bg, mat, and dst images must be of the same component type, but the images do not need to be the same pixel format. RGBA or BGRA images might be used, but the A channel of the destination is not updated.

This function assumes that the source is not premultiplied by alpha. If the value is pre-multiplied, use mode 1 of NvCVImage_CompositeRect() instead.

4.4. NvCVImage_CompositeRect

Here is detailed information about NvCVImage_CompositeRect.

```
NvCV_Status NvCVImage_CompositeRect(
  const NvCVImage *fg,  const NvCVPoint2i *fgOrg,
  const NvCVImage *bg,  const NvCVPoint2i *bgOrg,
  const NvCVImage *mat, unsigned int mode,
  NvCVImage *dst, const NvCVPoint2i *dstOrg,
  CUstream stream
);
```

Parameters

fg [in]

Type: const NvCVImage *

The foreground source, which is an RGB, a BGR, an RGBA, or a BGRA image with u8 or f32 components.

fgOrg [in]

Type: const NvCVPoint2i *

Pointer to the foreground image upper-left origin from which the image will be transferred.

```
typedef struct NvCVPoint2i { int x, y; } NvCVPoint2i;
```

If this is NULL, the image is transferred from (0,0).

bg [in]

Type: const NvCVImage *

The background source, which is an RGB, a BGR, an RGBA, or a BGRA image with u8 or f32 components.

bgOrg [in]

Type: const `NvCVPoint2i *`

Pointer to the background image upper-left origin

```
typedef struct NvCVPoint2i { int x, y; } NvCVPoint2i;
```

from which the image will be transferred. If this is NULL, the image is transferred from (0,0).

mat [in]

Type: `const NvCVImage *`

The matte Y or A image with u8 or f32 components, which indicates where the source image should come through. The dimensions of the matte determine the size of the area to be composited.

If this parameter includes an alpha channel , the parameter can be the same as the `fg` image.

mode [in]

Type: `unsigned int`

Here are the compositional mode selection options:

- `0`: normal, straight-alpha over composition.

  "Over" is the most common compositing operation, and applies the foreground over the background, as guided by the matte. . This is the compositional mode utilized that is used in other functions that do not explicitly specify a mode.
- `1`: pre-multiplied alpha *over* composition.

  Pre-multiplied means that the source R, G, and B are pre-multiplied by alpha, for example *(#R, #G, #B, #)*, and occurs naturally in a rendering scenario.
- Other values return a parameter error.

dst [out]

Type: `NvCVImage *`

The destination image, which can be the same as the `fg` (foreground) or `bg` (background) image, or a totally unrelated image.

> **Note:** If the destination image format contains alpha, the alpha channel is not updated in this implementation.

dstOrg [in]

Type: `const NvCVPoint2i *`

Pointer to the destination image upper-left origin

```
typedef struct NvCVPoint2i { int x, y; } NvCVPoint2i;
```

to which the image will be transferred. If this is NULL, the image is transferred to (0,0).

stream [out]

Type: `CUstream`

The CUDA stream on which to transfer the image. If the memory type of both the source and destination images is CPU, this parameter is ignored.

Return Value

- `NVCV_SUCCESS` on success
- `NVCV_ERR_PIXELFORMAT` if the pixel format is not supported.
- `NVCV_ERR_PARAMETER` if a mode other than 0 is selected.

Remarks

This function uses the specified matte image to composite a foreground image over a background image. The `fg`, `bg`, `mat`, and `dst` images must be of the same component type, but the images do not need to be the same pixel format. RGBA or BGRA images might also be used, but the A channel of the destination is not updated.

```
NvCVImage_Composite(fg, bg, mat, dst, str);
```

is equal to

```
NvCVImage_CompositeRect(fg, 0, bg, 0, mat, 0, dst, 0, str);
```

4.5. NvCVImage_CompositeOverConstant

Here is detailed information about `NvCVImage_Composite`.

```
NvCV_Status NvCVImage_CompositeOverConstant(
  const NvCVImage *src,
  const NvCVImage *mat,
  const void *bgColor,
  NvCVImage *dst,
  CUstream stream
);
```

Parameters

src [in]

Type: `const NvCVImage *`

The source RGB, BGR, RGBA, or BGRA image with u8 or f32 components. The pixel colors should not be pre-multiplied by alpha.

mat [in]

Type: `const NvCVImage *`

If this parameter includes an `alpha` channel, the parameter can be the same as the `src` image.

[in] bgColor

Type: `const void*`

A pointer to the background color over which the source image will be composited. This color must have the same type (u8, f32) and format (RGB, BGR) as the destination image and must remain valid until the composition completes.

dst [out]

Type: `NvCVImage *`

The destination BGR or RGB u8 or f32 image. The destination image might be the same image as the source image.

> **Note:** If the destination image format contains `alpha`, the alpha channel is not updated in this implementation.

stream [out]

Type: `CUstream`

The CUDA stream on which to transfer the image. If the memory type of both the source and destination images is CPU, this parameter is ignored.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT` when the pixel format is not supported.

Remarks

This function uses the specified matte image to composite a BGR or RGB image on a constant color field. RGBA and BGRA images might also be used, but the A channel of the destination is not updated.

4.6. NvCVImage_Create

Here is detailed information about `NvCVImage_Create`.

```
NvCV_Status NvCVImage_Create(
  unsigned width,
  unsigned height,
  NvCVImage_PixelFormat format,
  NvCVImage_ComponentType type,
  unsigned layout,
  unsigned memSpace,
  unsigned alignment,
  NvCVImage **out
);
```

Parameters

width [in]

Type: `unsigned`

The width, in pixels, of the image.

height [in]

Type: `unsigned`

The height, in pixels, of the image.

format [in]

Type: `NvCVImage_PixelFormat`

The format of the pixels.

type [in]

Type: `NvCVImage_ComponentType`

The type of the components of the pixels.

layout [in]

Type: `unsigned`

The organization of the components of the pixels in the image. Refer to Pixel Organizations for more information.

memSpace [in]

Type: `unsigned`

The type of memory in which the image data buffers will be stored. Refer to Memory Types for more information.

alignment [in]

Type: unsigned

The row byte alignment, which specifies the alignment of the first pixel in each scan line. Set this parameter to 0 or a power of 2.

- 1: Specifies no gap between scan lines.
- A byte alignment of 1 is required by all GPU buffers that are used by the video effect filters.
- 0: Specifies the default alignment, which depends on the type of memory in which the image data buffers are stored:
  - CPU memory: Specifies an alignment of 4 bytes,
  - GPU memory: Specifies the alignment set by cudaMallocPitch.

- 2 or greater: Specifies any other alignment, such as a cache line size of 16 or 32 bytes.

> **Note:** If the product of `width` and the `pixelBytes` member of `NvCVImage` is a whole-number multiple of `alignment`, the gap between scan lines is 0 bytes, regardless of the value of `alignment`.

out [out]

Type: `NvCVImage **`

Pointer to the location where the newly allocated image will be stored. The image descriptor and the pixel buffer are stored so that they are deallocated when `NvCVImage_Destroy()` is called.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT` when the pixel format is not supported.
- `NVCV_ERR_MEMORY` when the buffer requires more memory than is available.

Remarks

This function creates an image and allocates an image buffer that will be provided as input to an effect filter and allocates storage for the new image. This function is a C-style constructor for an instance of the `NvCVImage` structure (equivalent to new `NvCVImage` in C++).

4.7. NvCVImage_Dealloc

Here is detailed information about `NvCVImage_Dealloc`.

```
void NvCVImage_Dealloc(
  NvCVImage *im
);
```

Parameters

im [in,out]

Type: `NvCVImage *`

Pointer to the image whose image buffer will be freed.

Return Value

Does not return a value.

Remarks

This function frees the image buffer from the specified `NvCVImage` structure and sets the contents of the `NvCVImage` structure to 0.

4.8. NvCVImage_DeallocAsync

Here is detailed information about `NvCVImage_DeallocAsync`.

```
void NvCVImage_DeallocAsync(
  NvCVImage *im,
  CUstream stream
);
```

Parameters

im [in,out]

Type: `NvCVImage *`

Pointer to the image whose image buffer will be freed.

stream [out]

Type: `CUstream`

The CUDA stream on which to transfer the image. If the memory type of the source and destination images is CPU, this parameter is ignored.

Return Value

Does not return a value.

Remarks

This function asynchronously frees the image buffer on the specified CUDA stream and sets the contents of the NvCVImage structure to 0. If there is no buffer, or it is located on the CPU, it is freed immediately.

4.9. NvCVImage_Destroy

Here is detailed information about `NvCVImage_Destroy`.

```
void NvCVImage_Destroy(
  NvCVImage *im
);
```

Parameters

im

Type: `NvCVImage *`

Pointer to the image that will be destroyed.

Return Value

Does not return a value.

Remarks

This function destroys an image that was created with the `NvCVImage_Create()` function and frees resources and memory that were allocated for this image. This function is a C-style destructor for an instance of the `NvCVImage` structure (equivalent to delete im in C++).

4.10. NvCVImage_Init

Here is detailed information about `NvCVImage_Init`.

```
NvCV_Status NvCVImage_Init(
  NvCVImage *im,
  unsigned width,
  unsigned height,
  unsigned pitch,
  void *pixels,
  NvCVImage_PixelFormat format,
  NvCVImage_ComponentType type,
  unsigned layout,
  unsigned memSpace
);
```

Parameters

im [in,out]

Type: `NvCVImage *`

Pointer to the image that will be initialized.

width [in]

Type: `unsigned`

The width, in pixels, of the image.

height [in]

Type: `unsigned`

The height, in pixels, of the image.

pitch [in]

Type: `unsigned`

The vertical byte stride between pixels.

pixels [in]

Type: `void`

Pointer to the pixel buffer that will be attached to the `NvCVImage` object.

format

Type: `NvCVImage_PixelFormat`

The format of the pixels in the image.

type

Type: `NvCVImage_ComponentType`

The data type used to represent each component of the image.

layout [in]

Type: `unsigned`

The organization of the components of the pixels in the image. Refer to Pixel Organizations for more information.

memSpace [in]

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. Refer to Memory Types for more information.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT` when the pixel format is not supported.

Remarks

This function initializes an NvCVImage structure from a raw buffer pointer. Initializing an NvCVImage object from a raw buffer pointer is useful when you wrap an existing pixel buffer in an NvCVImage image descriptor.

This function is called by functions that initialize an NvCVImage object's data structure, for example:

- C++ constructors
- `NvCVImage_Alloc()`
- `NvCVImage_Realloc()`
- `NvCVImage_InitView()`

Call this function to initialize an NvCVImage object instead of directly setting the fields.

4.11. NvCVImage_InitView

Here is detailed information about `NvCVImage_InitView`.

```
void NvCVImage_InitView(
  NvCVImage *subImg,
  NvCVImage *fullImg,
  int x,
  int y,
  unsigned width,
  unsigned height
);
```

Parameters

subImg [in]

Type: `NvCVImage *`

Pointer to the existing image that will be initialized with the view.

fullImg [in]

Type: `NvCVImage *`

Pointer to the existing image from which the view of a specified rectangle in the image will be taken.

x [in]

Type: `int`

The `x` coordinate of the left edge of the view to be taken.

y [in]

Type: `int`

The `y` coordinate of the top edge of the view to be taken.

width [in]

Type: `unsigned`

The width, in pixels, of the view to be taken.

height [in]

Type: `unsigned`

The height, in pixels, of the view to be taken.

Return Value

Does not return a value.

Remarks

This function takes a view of the specified rectangle in an image and initializes another existing image descriptor with the view. No memory is allocated because the buffer of the image that is being initialized with the view (specified by the parameter fullImg) is used instead.

> **Note:** This only works for `NVCV_CHUNKY (NVCV_INTERLEAVED)` images, not for `NVCV_PLANAR` or any of the YUV planar or semi-planar image formats. However, if this view was intended to select a portion of an image to call `NvCVImage_Transfer()` or `NvCVImage_Composite()`, the rectangle versions `NvCVImage_TransferRect()` and `NvCVImage_CompositeRect()` can be used instead. These versions work for all formats,including planar or YUV formats.

4.12. NvCVImage_Realloc

Here is detailed information about `NvCVImage_Realloc`.

```
NvCV_Status NvCVImage_Realloc(
  NvCVImage *im,
  unsigned width,
  unsigned height,
  NvCVImage_PixelFormat format,
  NvCVImage_ComponentType type,
  unsigned layout,
  unsigned memSpace,
  unsigned alignment
);
```

Parameters

im [in,out]

Type: `NvCVImage *`

The image to initialize.

width [in]

Type: `unsigned`

The width, in pixels, of the image.

height [in]

The height, in pixels, of the image.

format [in]

Type: `NvCVImage_PixelFormat`

The format of the pixels.

type [in]

Type: `NvCVImage_ComponentType`

The type of the components of the pixels.

layout [in]

Type: `unsigned`

The organization of the components of the pixels in the image. Refer to Pixel Organizations for more information.

memSpace [in]

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. Refer to Memory Types for more information.

alignment [in]

Type: `unsigned`

The row byte alignment, which specifies the alignment of the first pixel in each scan line. Set this parameter to 0 or a power of 2.

- 1: Specifies no gap between scan lines.

  A byte alignment of 1 is required by all GPU buffers that are used by the video effect filters.
- 0: Specifies the default alignment, which depends on the type of memory in which the image data buffers are stored:
  - CPU memory: Specifies an alignment of 4 bytes.
  - GPU memory: Specifies the alignment set by cudaMallocPitch.
- 2 or greater: Specifies any other alignment, such as a cache line size of 16 or 32 bytes.

> **Note:** If the product of `width` and the `pixelBytes` member of NvCVImage is a whole-number multiple of `alignment`, the gap between scan lines is 0 bytes, regardless of the value of `alignment`.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT` when the pixel format is not supported.
- `NVCV_ERR_MEMORY` when the buffer requires more memory than is available.

Remarks

This function reallocates memory for, and initializes, an image.

> **Note:** This function assumes that the image is valid.

The function checks the bufferBytes member of NvCVImage to determine whether enough memory is available:

- If enough memory is available, the function reshapes, instead of reallocating, the memory.
- If enough memory is not available, the function frees the memory for the existing buffer and allocates the memory for a new buffer.

4.13. NvCVImage_Transfer

Here is detailed information about `NvCVImage_Transfer`.

```
NvCV_Status NvCVImage_Transfer(
  const NvCVImage *src,
  NvCVImage *dst,
  float scale,
  CUstream stream,
  NvCVImage *tmp
);
```

Parameters

src [in]

Type: `const NvCVImage *`

Pointer to the source image that will be transferred.

dst [out]

Type: `NvCVImage *`

Pointer to the destination image to which the source image will be transferred.

scale [in]

Type: `float`

A scale factor that can be applied when the component type of the source or destination image is floating-point. The scale factor scales the value of each component (not the size of the image), and has an effect **only** when the component type of the source or destination image is floating-point.

Here are the typical values:

- 1.0f
- 255.0f
- 1.0f/255.0f

This parameter is ignored if neither image has a floating-point component type.

stream [in]

Type: `CUstream`

The CUDA stream on which to transfer the image. If the memory type of both the source and destination images is CPU, this parameter is ignored.

tmp [in,out]

Type: `NvCVImage *`

Pointer to a temporary buffer in GPU memory that is required only if the source image is being converted and if the memory types of the source and destination images are different. The buffer has the same characteristics as the CPU image but resides on the GPU.

If necessary, the temporary GPU buffer is reshaped to suit the needs of the transfer, for example, to match the characteristics of the CPU image. Therefore, for the best performance, you can supply an empty image as the temporary GPU buffer. If necessary, `NvCVImage_Transfer()` allocates an appropriately sized buffer. The same temporary GPU buffer can be used in subsequent calls to `NvCVImage_Transfer()`, regardless of the shape, format, or component type, because the buffer will grow as needed to accommodate the largest memory requirement.

If a temporary GPU buffer is not needed, no buffer is allocated. If a temporary GPU buffer is not required, `tmp` can be `NULL`. However, if `tmp` is `NULL`, and a temporary GPU buffer is required, an ephemeral buffer is allocated with a resultant degradation in performance for image sequences.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_CUDA` when a CUDA error occurs.
- `NVCV_ERR_PIXELFORMAT` when the pixel format of the source or destination image is not supported.
- `NVCV_ERR_GENERAL` when an unspecified error occurs.

Remarks

This function transfers one image to another image and can perform some conversions on the image. The function uses the GPU to perform the conversions when an image resides on the GPU.

Table 4 provides details about the supported conversions between pixel formats.

Note: In each conversion type, the RGB can be in any order.

Table 4. Pixel Conversions

	U8 --> U8	U8 --> F32	F32 --> U8	F32 --> F32
Y--> Y	X		X	

	U8 --> U8	U8 --> F32	F32 --> U8	F32 --> F32
Y --> A	X		X	
Y --> RGB	X	X	X	X
Y --> RGBA	X	X	X	X
A --> Y	X		X	
A --> A	X		X	
A --> RGB	X	X	X	X
A --> RGBA	X			
RGB --> Y	X	X		
RGB --> A	X			
RGB --> RGB	X	X	X	X
RGB --> RGBA	X	X	X	X
RGBA --> Y	X	X		
RGBA --> A	X			
RGBA --> RGB	X	X	X	X
RGBA --> RGBA	X		X	
YUV420 --> RGB	X	X		
YUV422 --> RGB	X	X		
YUV444 --> RGB	X	X		
RGB --> YUV420	X		X	
RGB --> YUV422	X		X	
RGB --> YUV444	X		X	

Here is some additional information about these conversions:

- Conversions between chunky and planar pixel organizations occur in either direction.
- Conversions between CPU and GPU memory types can occur in either direction.
- Conversions between different orderings of components occur in either direction, for example, BGR --> RGB.
- Conversions from RGB (or BGR) to RGBA (or BGRA) will set the alpha channel to opaque (255 for u8 or 1.0 for f32).
- For RGBA (or BGRA) destinations, most implementations do not change the alpha channel.

  At initialization time, for chunky and planar u8 images, we recommend that you set it with one of the following:

  ```
  [cuda]memset(im.pixels, -1, im.pitch * im.height)
  ```

  or

```
[cuda]memset(im.pixels, -1, im.pitch * im.height*
                                im.numComponents)
```

- Other than pitch, if no conversion is necessary, all pixel format transfers are implemented, with `cudaMemcpy2DAsync()`.

  Another restriction in YUV --> YUV transfers is that the formats, layouts and colorspaces must match between `src` and dst.
- YUV420 and YUV422 and YUV444 sources have several variants.

  The colorspace must be set manually before the transfer. See YUV Color Spaces for more information.
- There are also RGBf16 --> RGBf32 and RGBf32 --> RGBf16 transfers.
- CPU --> CPU transfers are synchronous.
  - Additionally, when the `src` and `dst` formats are the same, all formats are accommodated on CPU and GPU, and this can be used as a replacement for `cudaMemcpy2DAsync()` (which it utilizes).
  - If the `src` and `dst` have different sizes, the transfer still occurs, but it will be clipped to the smaller size.

If both images reside on the CPU, the transfer occurs synchronously. However, if either image resides on the GPU, the transfer might occur asynchronously. A chain of asynchronous calls on the same CUDA stream is automatically sequenced as expected, but to synchronize, the `cudaStreamSynchronize()` function can be called.

4.14. NvCVImage_TransferRect

Here is detailed information about `NvCVImage_TransferRect`.

```
NvCV_Status NvCVImage_TransferRect(
  const NvCVImage *src,
  const NvCVRect2i *srcRect,
  NvCVImage *dst,
  const NvCVPoint2i *dstPt,
  float scale,
  CUstream stream,
  NvCVImage *tmp
);
```

Parameters

src [in]

Type: `const NvCVImage *`

Pointer to the source image that will be transferred.

srcRect [in]

Type: const `NvCVRect2i *`

Pointer to the source image rectangle that will be transferred.

```
typedef struct NvCVRect2i { int x, y, width, height; } NvCVRect2i;
```

If this is NULL, the entire src image rectangle is used.

dst [out]

Type: `NvCVImage *`

Pointer to the destination image to which the source image will be transferred.

dstPt [in]

Type: const NvCVPoint2i *

Pointer to the destination image location to which the image will be transferred.

```
typedef struct NvCVPoint2i { int x, y; } NvCVPoint2i;
```

If this is NULL, the image is transferred to (0,0).

scale [in]

Type: `float`

A scale factor that can be applied when the component type of the source or destination image is floating-point. The scale has an effect only when the component type of the source or destination image is floating-point.

Here are the typical values:

- 1.0f
- 255.0f
- 1.0f/255.0f

This parameter is ignored if neither image has a floating-point component type.

stream [in]

Type: `CUstream`

The CUDA stream on which to transfer the image. If the memory type of both the source and destination images is CPU, this parameter is ignored.

tmp [in,out]

Type: `NvCVImage *`

Pointer to a temporary buffer in GPU memory that is required only if the source image is being converted **and** if the memory types of the source and destination images are different. The buffer has the same characteristics as the CPU image but resides on the GPU.

If necessary, the temporary GPU buffer is reshaped to suit the needs of the transfer, for example, to match the characteristics of the CPU image. Therefore, for the best performance, you can supply an empty image as the temporary GPU buffer. If necessary, `NvCVImage_Transfer()` allocates an appropriately sized buffer. The same temporary GPU buffer can be used in subsequent calls to `NvCVImage_Transfer()`, regardless of the shape, format, or component type, because the buffer will grow as needed to accommodate the largest memory requirement.

If a temporary GPU buffer is not needed, no buffer is allocated. If a temporary GPU buffer is not required, tmp can be NULL. However, if tmp is NULL, and a temporary GPU buffer is required, an ephemeral buffer is allocated with a resultant degradation in performance for image sequences.

Return Value

- NVCV_SUCCESS on success.
- NVCV_ERR_CUDA when a CUDA error occurs.
- NVCV_ERR_PIXELFORMAT when the pixel format of the source or destination image is not supported.
- NVCV_ERR_GENERAL when an unspecified error occurs.

Remarks

This function is like NvCVImage_Transfer(), because they share the same code. A rectangle can be copied by combining NvCVImage_InitView() with NvCVImage_Transfer(), but this only works for chunky images.

NvCVImage_TransferRect works on the chunky, planar, and semi-planar image layouts, and there is no difference in performance.

NvCVImage_Transfer(src, dst, scale, stream, tmp) is equivalent to NvCVImage_TransferRect(src, 0, dst, 0, scale, stream, tmp).

If you are not careful when you copy YUV rectangles, unexpected clipping will occur:

- YUV420 must have even x, y, width and height.
- YUV422 must have even x and width.

4.15. NvCVImage_TransferFromYUV

Here is detailed information about NvCVImage_TransferFromYUV.

```
NvCV_Status NvCVImage_TransferFromYUV(
  const void *y,              int yPixBytes,   int yPitch,
  const void *u, const void *v, int uvPixBytes, int uvPitch,
  NvCVImage_PixelFormat yuvFormat, NvCVImage_ComponentType yuvType,
  unsigned yuvColorSpace, unsigned yuvMemSpace,
  NvCVImage *dst, const NvCVRect2i *dstRect,
  float scale, struct CUstream_st *stream, NvCVImage *tmp
);
```

Parameters

y [in]

Type: const void *

Pointer to pixel(0,0) of the luminance channel.

yPixBytes [in]

Type: `int`

The byte stride between y pixels horizontally.

yPitch [in]

Type: `int`

The byte stride between y pixels vertically.

u [in]

Type: `const void *`

Pointer to pixel(0,0) of the u (Cb) chrominance channel.

v [in]

Type: `const void *`

Pointer to pixel(0,0) of the v (Cr) chrominance channel.

uvPixBytes [in]

Type: `int`

The byte stride between u or v pixels horizontally.

uvPitch [in]

Type: `int`

The byte stride between u or v pixels vertically.

yuvColorSpace [in]

Type: `unsigned int`

The yuv colorspace, which specifies the range, the chromaticities, and the chrominance phase.

yuvMemSpace [in]

Type: `unsigned int`

The memory space in which the YUV buffer resides (for example, CPU, CUDA, and so on).

dst [out]

Type: `NvCVImage *`

Pointer to the destination image to which the source image will be transferred.

dstRect [in]

Type: `const NvCVRect2i *`

Pointer to the destination image rectangle

```
typedef struct NvCVRect2i { int x, y, width, height; } NvCVRect2i;
```

to which the image will be transferred. This can be NULL, in which case the entire dst image is transferred.

scale [in]

Type: `float`

A scale factor that can be applied when the component type of the source or destination image is floating-point. The scale has an effect only when the component type of the source or destination image is floating-point.

Here are the typical values:

- 1.0f
- 255.0f
- 1.0f/255.0f

This parameter is ignored if neither image has a floating-point component type.

stream [in]

Type: `CUstream`

The CUDA stream on which to transfer the image. If the memory type of both the source and destination images is CPU, this parameter is ignored.

tmp [in,out]

Type: `NvCVImage *`

Pointer to a temporary buffer in GPU memory that is required only if the source image is being converted and if the memory types of the source and destination images are different. The buffer has the same characteristics as the CPU image but resides on the GPU.

If necessary, the temporary GPU buffer is reshaped to suit the needs of the transfer, for example, to match the characteristics of the CPU image. Therefore, for the best performance, you can supply an empty image as the temporary GPU buffer. If necessary, `NvCVImage_Transfer()` allocates an appropriately sized buffer. The same temporary GPU buffer can be used in subsequent calls to `NvCVImage_Transfer()`, regardless of the shape, format, or component type, because the buffer will grow as needed to accommodate the largest memory requirement.

If a temporary GPU buffer is not needed, no buffer is allocated. If a temporary GPU buffer is not required, `tmp` can be NULL. However, if `tmp` is NULL, and a temporary GPU buffer is required, an ephemeral buffer is allocated with a resultant degradation in performance for image sequences.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_CUDA` when a CUDA error occurs.

- NVCV_ERR_PIXELFORMAT when the pixel format of the source or destination image is not supported.
- NVCV_ERR_GENERAL when an unspecified error occurs.

Remarks

This function is like NvCVImage_TransferRect(), which can also copy from YUV images. The difference is that TransferRect works with images that have a structure, as described in the layout (or planar) parameter, and NvCVImage_TransferFromYUV works with images that have no structure that is represented in the taxonomy of the layout parameter. Since the structure is not known, TransferFromYUV is also slower than TransferRect when transferring from CPU --> GPU.

4.16. NvCVImage_TransferToYUV

Here is detailed information about NvCVImage_TransferToYUV.

```
NvCV_Status NvCVImage_TransferToYUV(
  const NvCVImage *src,             const NvCVRect2i *srcRect,
  const void *y,                    int yPixBytes,  int yPitch,
  const void *u, const void *v,     int uvPixBytes, int uvPitch,
  NvCVImage_PixelFormat yuvFormat, NvCVImage_ComponentType yuvType,
  unsigned yuvColorSpace,           unsigned yuvMemSpace,
  float scale,
  CUstream stream,
  NvCVImage *tmp
);
```

Parameters

src [in]

Type: const NvCVImage *

Pointer to the source image that will be transferred.

srcRect [in]

Type: const NvCVRect2i *

Pointer to the source image rectangle that will be transferred.

```
typedef struct NvCVRect2i { int x, y, width, height; } NvCVRect2i;
```

If this is NULL, the entire src image rectangle is used.

y [out]

Type: NvCVImage *

Pointer to pixel(0,0) of the luminance channel.

yPixBytes [in]

Type: int

The byte stride between y pixels horizontally.

yPitch [in]

Type: `in`

The byte stride between y pixels vertically.

u [out]

Type: `NvCVImage *`

Pointer to pixel(0,0) of the u (Cb) chrominance channel.

v [out]

Type: `NvCVImage *`

Pointer to pixel(0,0) of the v (Cr) chrominance channel.

uvPixBytes [in]

Type: `int`

The byte stride between u or v pixels horizontally.

uvPitch [in]

Type: `int`

The byte stride between u or v pixels vertically.

yuvColorSpace [in]

Type: `unsigned int`

The yuv colorspace, which specifies the range, the chromaticities, and the chrominance phase.

yuvMemSpace [in]

Type: `unsigned int`

The memory space where the pixel buffers reside.

scale [in]

Type: `float`

A scale factor that can be applied when the component type of the source or destination image is floating-point. The scale has an effect only when the component type of the source or destination image is floating-point.

Here are the typical values:

- 1.0f
- 255.0f
- 1.0f/255.0f

This parameter is ignored if neither image has a floating-point component type.

stream [in]

Type: `CUstream`

The CUDA stream on which to transfer the image. If the memory type of both the source and destination images is CPU, this parameter is ignored.

tmp [in,out]

Type: `NvCVImage *`

Pointer to a temporary buffer in GPU memory that is required only if the source image is being converted and if the memory types of the source and destination images are different. The buffer has the same characteristics as the CPU image but resides on the GPU.

If necessary, the temporary GPU buffer is reshaped to suit the needs of the transfer, for example, to match the characteristics of the CPU image. Therefore, for the best performance, you can supply an empty image as the temporary GPU buffer. If necessary, `NvCVImage_Transfer()` allocates an appropriately sized buffer. The same temporary GPU buffer can be used in subsequent calls to `NvCVImage_Transfer()`, regardless of the shape, format, or component type, because the buffer will grow as needed to accommodate the largest memory requirement.

If a temporary GPU buffer is not needed, no buffer is allocated. If a temporary GPU buffer is not required, `tmp` can be NULL. However, if `tmp` is NULL, and a temporary GPU buffer is required, an ephemeral buffer is allocated with a resultant degradation in performance for image sequences.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_CUDA` when a CUDA error occurs.
- `NVCV_ERR_PIXELFORMAT` when the pixel format of the source or destination image is not supported.
- `NVCV_ERR_GENERAL` when an unspecified error occurs.

Remarks

This function is like `NvCVImage_TransferRect()`, which can also copy to YUV images. The difference is that `TransferRect` works with images that have a structure, as described in the layout (or planar) parameter, and `NvCVImage_TransferToYUV` works with images that have no structure that is represented in the taxonomy of the layout parameter.

4.17. NvCVImage_MapResource

Here is detailed information about `NvCVImage_MapResource`.

```
NvCV_Status NvCVImage_MapResource(
  NvCVImage *im,
```

```
  struct CUstream_st *stream
);
```

Parameters

im [in,out]

Type: NvCVImage *

The image to be mapped.

stream [out]

Type: struct CUStream_st *

The stream on which the mapping is to be performed.

Return Value

NVCV_SUCCESS on success.

Remarks

Between rendering by a graphics system and Transfer by CUDA, you also need to map the texture resource. This process involves quite a bit of overhead, so its use should be minimized. Every call to NvCVImage_MapResource() should be matched by a subsequent call to NvCVImage_UnmapResource().

One way to create an image-wrapped resource on Windows is to call NvCVImage_InitFromD3DTexture().

4.18. NvCVImage_UnmapResource

Here is detailed information about NvCVImage_UnmapResource.

```
NvCV_Status NvCVImage_UnmapResource(
  NvCVImage *im,
  struct CUstream_st *stream
);
```

Parameters

im [in,out]

Type: NvCVImage *

The image to be mapped.

stream [out]

Type: struct CUStream_st *

The stream on which the mapping is to be performed.

Return Value

`NVCV_SUCCESS` on success

Remarks

Between rendering by a graphics system and Transfer by CUDA, you also need to map the texture resource. This process involves quite a bit of overhead, so its use should be minimized. Every call to `NvCVImage_MapResource()` should be matched by a subsequent call to NvCVImage_UnmapResource().

One way to create an image-wrapped resource on Windows is to call `NvCVImage_InitFromD3DTexture()`.

4.19. NvCVImage_InitFromD3DTexture

Here is detailed information about `NvCVImage_InitFromD3DTexture`.

```
NvCV_Status NvCVImage_InitFromD3DTexture(
  NvCVImage *im,
  struct ID3D11Texture2D *tx
);
```

Parameters

im [in,out]

Type: `NvCVImage *`

The image to be initialized.

tx [in]

Type: `struct ID3D11Texture2D *`

The texture to be used for initialization.

Return Value

`NVCV_SUCCESS` on success.

Remarks

You can initialize an `NvCVImage` from a D3D11 texture. The `pixelFormat` and component types are transferred, and a `cudaGraphicsResource` is registered. The `NvCVImage` destructor unregisters the resource.

Note: This is designed to work with `NvCVImage_Transfer()`.

Before you allow the D3D texture to render into the NvCVImage, you need to first call `NvCVImage_MapResource()` and `NvCVImage_UnmapResource()`.

4.20. C++ Helper Functions for Sample Applications

The helper functions in this section are provided in the `nvCVOpenCV.h` file to help the NvCVImage interface with OpenCV's image representation, for example, `cv::Mat`.

4.20.1. CVWrapperForNvCVImage

Here is detailed information about `CVWrapperForNvCVImage`.

```
void CVWrapperForNvCVImage(
  const NvCVImage *vfxIm,
  cv::Mat *cvIm
);
```

Parameters

vfxIm [in]

Type: `const NvCVImage *`

Pointer to an allocated `NvCVImage` object.

cvIm [out]

Type: `cv::Mat *`

Pointer to an empty `OpenCV` image, appropriately initialized to access the buffer of the `NvCVImage` object. An empty `OpenCV` image is created by the default `cv::Mat` constructor.

Return Value

Does not return a value.

Remarks

This function creates an `OpenCV` image wrapper for an `NvCVImage` object.

4.20.2. NVWrapperForCVMat

Here is detailed information about `NVWrapperForCVMat`.

```
void NVWrapperForCVMat(
  const cv::Mat *cvIm,
  NvCVImage *vIm
);
```

Parameters

cvIm [in]

Type: `const cv::Mat *`

Pointer to an allocated `OpenCV` image.

vfxIm [out]

Type: `NvCVImage *`

Pointer to an empty `NvCVImage` object, appropriately initialized by this function to access the buffer of the `OpenCV` image. An empty `NvCVImage` object is created by the default (no-argument) `NvCVImage()` constructor.

Return Value

Does not return a value.

Remarks

This function creates an `NvCVImage` object wrapper for an `OpenCV` image.

4.21. Image Functions for C++ Only

The image API provides constructors, a destructor for C++, and some additional functions that are accessible only to C++.

4.21.1. NvCVImage Constructors

This section provides a list of the `NvCVImage` Constructors in the AR and Video Effects SDKs.

4.21.1.1. Default Constructor

The default constructor creates an empty image with no buffer.

```
NvCVImage();
```

4.21.1.2. Allocation Constructor

The allocation constructor creates an image to which memory has been allocated and that has been initialized.

```
NvCVImage(
  unsigned width,
  unsigned height,
  NvCVImage_PixelFormat format,
  NvCVImage_ComponentType type,
  unsigned layout,
  unsigned memSpace,
  unsigned alignment
);
```

Here are the parameters:

format [in]

Type: `NvCVImage_PixelFormat`

The format of the pixels.

type [in]

Type: `NvCVImage_ComponentType`

The type of the components of the pixels.

layout [in]

Type: `unsigned`

The organization of the components of the pixels in the image. Refer to Pixel Organizations for more information.

memSpace [in]

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. Refer to Memory Types for more information.

alignment [in]

Type: unsigned

The row byte alignment, which specifies the alignment of the first pixel in each scan line. Set this parameter to 0 or a power of 2.

- 1: Specifies no gap between scan lines.

  A byte alignment of 1 is required by all GPU buffers used by the video effect filters.
- 0: Specifies the default alignment, which depends on the type of memory in which the image data buffers are stored:
  - CPU memory: Specifies an alignment of 4 bytes.
  - GPU memory: Specifies the alignment set by `cudaMallocPitch`.
- 2 or greater: Specifies any other alignment, such as a cache line size of 16 or 32 bytes.

> **Note:** If the product of width and the pixelBytes member of `NvCVImage` is a whole-number multiple of alignment, the gap between scan lines is 0 bytes, regardless of the value of alignment.

4.21.1.3. Subimage Constructor

The subimage constructor creates an image that is initialized with a view of the specified rectangle in another image. No additional memory is allocated.

```
NvCVImage(
  NvCVImage *fullImg,
  int x,
```

```
  int y,
  unsigned width,
  unsigned height
);
```

Here are the parameters:

fullImg [in]

Type: `NvCVImage *`

Pointer to the existing image from which the view of a specified rectangle in the image will be taken.

x [in]

The x coordinate of the left edge of the view to be taken.

y [in]

The y coordinate of the top edge of the view to be taken.

width [in]

Type: `unsigned`

The width, in pixels, of the view to be taken.

height [in]

Type: `unsigned`

The height, in pixels, of the view to be taken.

4.21.2. NvCVImage Destructor

Here is the code for this destructor.

```
~NvCVImage();
```

4.21.3. copyFrom

Here is some information about the `copyFrom` function.

This version copies an entire image to another image and is functionally identical to `NvCVImage_Transfer(src, this, 1.0f, 0, NULL);`.

```
NvCV_Status copyFrom(
  const NvCVImage *src
);
```

This version copies the specified rectangle in the source image to the destination image.

```
NvCV_Status copyFrom(
  const NvCVImage *src,
  int srcX,
  int srcY,
  int dstX,
  int dstY,
  unsigned width,
  unsigned height
);
```

Parameters

src [in]

Type: `const NvCVImage *`

Pointer to the existing source image from which the specified rectangle will be copied.

srcX [in]

Type: `int`

The x coordinate in the source image of the left edge of the rectangle will be copied.

srcY [in]

Type: `int`

The y coordinate in the source image of the top edge of the rectangle to be copied.

dstX [in]

Type: `int`

The x coordinate in the destination image of the left edge of the copied rectangle.

srcY [in]

Type: `int`

The y coordinate in the destination image of the top edge of the copied rectangle.

width [in]

Type: `unsigned`

The width, in pixels, of the rectangle to be copied.

height [in]

Type: `unsigned`

The height, in pixels, of the rectangle to be copied.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT` when the pixel format is not supported.
- `NVCV_ERR_MISMATCH` when the formats of the source and destination images are different.
- `NVCV_ERR_CUDA` if a CUDA error occurs.

Remarks

This overloaded function copies an entire image to another image or copies the specified rectangle in an image to another image.

This function can copy image data buffers that are stored in different memory types as follows:

- From CPU to CPU
- From CPU to GPU
- From GPU to GPU
- From GPU to CPU

Note: For additional use cases, use the `NvCVImage_Transfer()` function.

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com