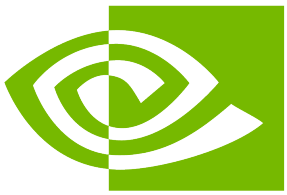

NVIDIA NvCVImage API Guide

Release 1.0.1

NVIDIA Corporation

Sep 10, 2026

Contents

Download this guide as a PDF

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

NvCVImage offers the following features:

- All functions available to C and C++.
- Buffers on CPU or GPU.
- Many pixel formats, such as RGB, BGRA, grayscale, RGB10A2, and YUV420.
- Many component types, such as u8, p32, f32, and f16.
- Chunky (interleaved), planar, and semi-planar arrangements.
- Buffer allocation, reallocation, and deallocation.
- Image transfer.
- Conversion between various 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 1. Images

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.

1.1. Convert 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.

1.1.1. Convert 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);
```

1.1.2. Convert 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);
```

1.1.3. Convert Decoded Frames from 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 example converts a decoded frame from the NV12 format 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();
```

(continues on next page)

(continued from previous page)

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

Note

The preceding example 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:

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

1.1.4. Convert an NvCVImage Object to a Buffer for Encoding by NvEncoder

To convert the NvCVImage object to the pixel format that is used during encoding via NvEncoder, you can call the *NvCVImage_Transfer()* function. The following example encodes a frame 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,
```

(continues on next page)

(continued from previous page)

```
                encoderInputFrame->chromaOffsets,
                encoderInputFrame->numChromaPlanes);
pEnc->EncodeFrame(vPacket);
```

1.2. Allocate 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.

1.2.1. Use 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. For more information, see *NvCVImage Allocation Constructor*.

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 pixel organization, memory type, and byte alignment that are identical 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,
  ```

(continues on next page)

(continued from previous page)

```
  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
);
```

1.2.2. Use 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 the buffer, call the *NvCVImage_Realloc()* function.

     This function checks for an allocated buffer and reshapes the buffer if it is big enough. The function then frees the buffer and calls `NvCVImage_Alloc()`.

1.3. Transfer 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.

1.3.1. Transfer Input Images from a CPU Buffer to a GPU Buffer

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);
```

1. 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 is allocated or reallocated as needed.
2. Call the *NvCVImage_Transfer()* function to copy the contents of the source CPU buffer via the staging GPU buffer into the final GPU buffer.

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

The same staging buffer can be reused in multiple NvCVImage_Transfer calls in various contexts regardless of the image sizes and can avoid buffer allocations if it is persistent.

1.3.2. Transfer Output Images from a GPU Buffer to a CPU Buffer

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);
```

1. 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 is allocated or reallocated as needed.

2. Call the *NvCVImage_Transfer()* function to copy the contents of the source GPU buffer via the staging GPU buffer into the destination CPU 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.

1.4. Working with 10-Bit RGBA (R10G10B10A2) Buffers

The `NVCV_RGB10A2` pixel format packs three 10-bit color components and a 2-bit alpha into a single 32-bit word.

Note

`NVCV_RGB10A2` is always interleaved (`NVCV_INTERLEAVED`). Planar layouts are not supported for this format.

1.4.1. Allocate a 10-Bit RGBA Buffer

Call the *NvCVImage_Alloc()* function with `NVCV_RGB10A2` as the pixel format and `NVCV_P32` as the component type.

```
// Allocate a 10-bit RGBA GPU buffer.
NvCVImage hdrFrame;
NvCVImage_Alloc(&hdrFrame, 1920, 1080, NVCV_RGB10A2, NVCV_P32,
                NVCV_INTERLEAVED, NVCV_GPU, 1);
```

1.4.2. Convert Between 8-Bit and 10-Bit Formats

To convert between 8-bit RGBA and 10-bit RGB10A2 buffers, call the *NvCVImage_Transfer()* function. The function handles the bit-depth expansion or reduction and the repacking.

```
NvCVImage sdrFrame, hdrFrame, stageImg;

// 8-bit RGBA source, for example from a webcam or a decoder.
NvCVImage_Alloc(&sdrFrame, 1920, 1080, NVCV_RGBA, NVCV_U8,
                NVCV_INTERLEAVED, NVCV_GPU, 1);

// 10-bit RGB10A2 destination.
NvCVImage_Alloc(&hdrFrame, 1920, 1080, NVCV_RGB10A2, NVCV_P32,
                NVCV_INTERLEAVED, NVCV_GPU, 1);

// Convert from 8-bit to 10-bit, with repacking.
NvCVImage_Transfer(&sdrFrame, &hdrFrame, 1.0f, stream, &stageImg);
```

Chapter 2. Enumerations

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

2.1. NvCVImage_ComponentType

This enumeration defines the data type that represents 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 (5)
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).

NVCV_P32 (11)
Packed 32-bit value holding three 10-bit color components and a 2-bit alpha. Used with the `NVCV_RGB10A2` pixel format.

2.2. 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_RGB10A2
Packed 10-bit-per-channel RGB with 2-bit alpha. The four components (R10, G10, B10, A2) occupy a single 32-bit word per pixel. The corresponding component type is NVCV_P32.

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).

2.2.1. Pixel Organizations

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 (such as the red components) of all pixels in the image are grouped together.
- *Semi-planar* pixels are a mix between interleaved and planar components. These 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.

For more information about YUV pixel formats, see YUV pixel formats.

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

Type	Value	Description
NVCV_INTERLEAVED NVCV_CHUNKY	0	Each of these types specifies interleaved, or chunky, pixels in which the components of each NVCV_CHUNKY 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.
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.
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 one of the following planar YUV arrangements: • YUV 4:2:2 • YUV 4:2:0 • YUV 4:4:4 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.

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.

2.2.2. YUV Color Spaces

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

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.

2.2.3. Memory Types

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

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 3. Functions

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

3.1. NvCVImage_Alloc

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

3.1.1. 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. For more information, see *Pixel Organizations*.

memSpace [in]
Type: unsigned

The type of memory in which to store the image data buffers. For more information, see *Memory Types*.

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.

3.1.2. Return Values

- 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.

3.1.3. 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.

3.2. NvCVImage_ComponentOffsets

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

3.2.1. Parameters

format [in]
Type: NvCVImage_PixelFormat

The pixel format whose component offsets are 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`.)

3.2.2. Return Values

Does not return a value.

3.2.3. Remarks

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

3.3. NvCVImage_Composite

3.3.1. Parameters

fg [in]
Type: `const NvCVImage *`

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

bg [in]
Type: `const NvCVImage *`

The background source, which is an RGB, BGR, RGBA, or 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.

This can be the same as the `fg` image, if it includes an alpha channel.

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.

3.3.2. Return Values

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

3.3.3. 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.

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

3.4. NvCVImage_CompositeOverConstant

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

3.4.1. 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 *`

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

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

bgColor [in]
Type: `const void *`

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

dst [out]
Type: `NvCVImage *`

The destination RGB, BGR, RGBA, or BGRA 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.

3.4.2. Return Values

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

3.4.3. 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.

3.5. 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
);
```

3.5.1. Parameters

fg [in]
Type: `const NvCVImage *`

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

fgOrg [in]
Type: `const NvCVPoint2i *`

Pointer to the foreground image upper-left origin from which to transfer the image:

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

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

bg [in]
Type: `const NvCVImage *`

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

bgOrg [in]
Type: `const NvCVPoint2i *`

Pointer to the background image upper-left origin from which to transfer the image:

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

If this value 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`

The compositional mode selection options are as follows:

- `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 that is used in other functions that do not explicitly specify a mode.

- 1: premultiplied alpha *over* composition.

  Premultiplied means that the source R, G, and B are premultiplied by alpha; for example, (αR, αG, αB, α). This 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 to which to transfer the image:

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

If this value 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.

3.5.2. Return Values

- `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.

3.5.3. 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);
```

3.6. NvCVImage_Create

```
NvCV_Status NvCVImage_Create(
  unsigned width,
  unsigned height,
  NvCVImage_PixelFormat format,
  NvCVImage_ComponentType type,
  unsigned layout,
  unsigned memSpace,
  unsigned alignment,
```

(continues on next page)

(continued from previous page)

```
  NvCVImage **out
);
```

3.6.1. 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. For more information, see *Pixel Organizations*.

memSpace [in]
Type: unsigned

The type of memory in which to store the image data buffers. For more information, see *Memory Types*.

alignment [in]
Type: unsigned

The row byte alignment, which specifies the alignment of the first pixel in each scan line.

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

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 of the newly allocated image. The image descriptor and the pixel buffer are stored so that they are deallocated when *NvCVImage_Destroy()* is called.

3.6.2. Return Values

- `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.

3.6.3. Remarks

This function creates an image and allocates an image buffer to 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++).

3.7. NvCVImage_Dealloc

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

3.7.1. Parameters

`im [in,out]`
Type: `NvCVImage *`

Pointer to the image whose image buffer is to be freed.

3.7.2. Return Value

Does not return a value.

3.7.3. Remarks

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

3.8. NvCVImage_DeallocAsync

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

3.8.1. Parameters

im [in,out]
Type: `NvCVImage *`

Pointer to the image whose image buffer is to 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.

3.8.2. Return Value

Does not return a value.

3.8.3. 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.

3.9. NvCVImage_Destroy

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

3.9.1. Parameters

im [in,out]
Type: `NvCVImage *`

Pointer to the image to destroy.

3.9.2. Return Value

Does not return a value.

3.9.3. 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++).

3.10. 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
);
```

3.10.1. Parameters

`im [in,out]`
Type: `NvCVImage *`

Pointer to 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.

`pitch [in]`
Type: `unsigned`

The vertical byte stride between pixels.

`pixels [in]`
Type: `void *`

Pointer to the pixel buffer to attach to the `NvCVImage` object.

`format [in]`
Type: `NvCVImage_PixelFormat`

The format of the pixels in the image.

type [in]
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. For more information, see *Pixel Organizations*.

memSpace [in]
Type: unsigned

The type of memory in which to store the image data buffers. For more information, see *Memory Types*.

3.10.2. Return Values

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

3.10.3. Remarks

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

This function is called by functions that initialize the data structure of an NvCVImage object, such as the following:

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

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

3.11. NvCVImage_InitFromD3DTexture

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

3.11.1. Parameters

im [in,out]
Type: NvCVImage *

The image to be initialized.

tx [in]
Type: `struct ID3D11Texture2D *`

The texture to be used for initialization.

3.11.2. Return Value

`NVCV_SUCCESS` on success.

3.11.3. Remarks

You can initialize an `NvCVImage` object 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` object, you need to call *NvCVImage_MapResource()* and then *NvCVImage_UnmapResource()*.

3.12. NvCVImage_InitView

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

3.12.1. Parameters

subImg [in]
Type: `NvCVImage *`

Pointer to the existing image to initialize with the view.

fullImg [in]
Type: `NvCVImage *`

Pointer to the existing image from which to take the view of a specified rectangle in the image.

x [in]
Type: `int`

The x-coordinate of the left edge of the view.

y [in]
Type: `int`

The y-coordinate of the top edge of the view.

width [in]
Type: `unsigned`

The width, in pixels, of the view.

height [in]
Type: `unsigned`

The height, in pixels, of the view.

3.12.2. Return Value

Does not return a value.

3.12.3. 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 function works only for `NVCV_CHUNKY` (`NVCV_INTERLEAVED`) images, not for `NVCV_PLANAR` or any of the YUV planar or semi-planar image formats. However, if you want this view to select a portion of an image to pass to *NvCVImage_Transfer()* or *NvCVImage_Composite()*, you can use the rectangle versions *NvCVImage_TransferRect()* and *NvCVImage_CompositeRect()* instead. These versions work for all formats, including planar or YUV formats.

3.13. NvCVImage_MapResource

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

3.13.1. 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.

3.13.2. Return Value

`NVCV_SUCCESS` on success.

3.13.3. 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()*.

3.14. NvCVImage_Realloc

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

3.14.1. 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. For more information, see *Pixel Organizations*.

memSpace [in]
Type: unsigned

The type of memory in which to store the image data buffers. For more information, see *Memory Types*.

alignment [in]
Type: unsigned

The row byte alignment, which specifies the alignment of the first pixel in each scan line.

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

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.

3.14.2. Return Values

- 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.

3.14.3. 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.

3.15. NvCVImage_Transfer

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

3.15.1. Parameters

src [in]
Type: `const NvCVImage *`

Pointer to the source image to transfer.

dst [out]
Type: `NvCVImage *`

Pointer to the destination image to which to transfer the source image.

scale [in]
Type: `float`

A scale factor to apply 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.

The following are 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 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, such as matching the characteristics of the CPU image. Therefore, for 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 grows 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.

3.15.2. Return Values

- `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.

3.15.3. 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.

The following table provides details about the supported conversions between pixel formats.

Note

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

	U8 → U8	U8 → F32	F32 → U8	F32 → F32
Y → Y	✓	✓	✓	✓
Y → A	✓	✓	✓	✓
Y → RGB	✓	✓	✓	✓
Y → RGBA	✓	✓	✓	✓
A → Y	✓	✓	✓	✓
A → A	✓	✓	✓	✓
A → RGB	✓	✓	✓	✓
A → RGBA	✓			
RGB → Y	✓	✓		
RGB → A	✓	✓		
RGB → RGB	✓	✓	✓	✓
RGB → RGBA	✓	✓	✓	✓
RGBA → Y	✓	✓		
RGBA → A	✓			
RGBA → RGB	✓	✓	✓	✓
RGBA → RGBA	✓	✓	✓	✓
YUV420 → RGB	✓	✓		
YUV420 → RGBA	✓	✓		
YUV422 → RGB	✓	✓		
YUV422 → RGBA	✓	✓		
YUV444 → RGB	✓	✓		
YUV444 → RGBA	✓	✓		
RGB → YUV420	✓		✓	
RGBA → YUV420	✓		✓	
RGB → YUV422	✓		✓	
RGBA → YUV422	✓		✓	
RGB → YUV444	✓		✓	
RGBA → YUV444	✓		✓	

The following kinds of conversions can occur in either direction:

- Conversions between chunky and planar pixel organizations.
- Conversions between CPU and GPU memory types.
- Conversions between different orderings of components, such as BGR → RGB.

Conversions from RGB (or BGR) to RGBA (or BGRA) set the alpha channel to opaque (255 for u8 or 1.0

for f32).

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

```
[cuda]memset(im.pixels, -1, im.pitch * im.height)
[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 be the same in `src` and `dst`.

YUV420, YUV422, and YUV444 sources have several variants. You must set the colorspace manually before the transfer. For more information, see *YUV Color Spaces*.

There are also RGBf16 □ RGBf32 and RGBf32 □ RGBf16 transfers.

CPU □ CPU transfers are synchronous. When the `src` and `dst` formats are the same, all formats are accommodated on CPU and GPU, and you can use this function as a replacement for `cudaMemcpy2DAsync()` (which it uses). If `src` and `dst` have different sizes, the transfer still occurs but is 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; to synchronize, you can call `cudaStreamSynchronize()`.

Transfers to and from CUDA arrays have limited support. These transfers are typically accessed by mapping DirectX (and eventually OpenGL and Vulkan) textures to CUDA. (See *NvCVImage_MapResource()*, *NvCVImage_UnmapResource()*, and *NvCVImage_InitFromD3DTexture()*.) We provide NvCVImage wrappers to map and unmap these textures. There is full capability to copy to and from textures for all basic texture types: chunky (`{RGBA, YA, Y}{u8, u16, s16, f16, u32, s32, f32}`), Au8, and Windows BGRAu8. We also implemented several conversions from RGBAu8 textures to `{RGBA, BGRA, RGB, BGR}{u8, f32}` CUDA buffers.

3.16. 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
);
```

3.16.1. 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 range, chromaticities, and chrominance phase.

yuvMemSpace [in]
Type: `unsigned int`

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

dst [out]
Type: `NvCVImage *`

Pointer to the destination image to which to transfer the source image.

dstRect [in]
Type: `const NvCVRect2i *`

Pointer to the destination image rectangle to which to transfer the image.

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

If this value is `NULL`, the entire destination image is transferred.

scale [in]
Type: `float`

A scale factor to apply 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.

The following are 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 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, such as matching the characteristics of the CPU image. Therefore, for 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 grows 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.

3.16.2. Return Values

- `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.

3.16.3. Remarks

This function is like *NvCVImage_TransferRect()*, which can also copy from YUV images. The difference is that `NvCVImage_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. Because the structure is not known, `NvCVImage_TransferFromYUV()` is also slower than `NvCVImage_TransferRect()` when transferring from CPU to GPU.

3.17. NvCVImage_TransferRect

```
NvCV_Status NvCVImage_TransferRect(
  const NvCVImage *src,
  const NvCVRect2i *srcRect,
  NvCVImage *dst,
```

(continues on next page)

(continued from previous page)

```
  const NvCVPoint2i *dstPt,
  float scale,
  CUstream stream,
  NvCVImage *tmp
);
```

3.17.1. Parameters

src [in]
Type: `const NvCVImage *`

Pointer to the source image to transfer.

srcRect [in]
Type: `const NvCVRect2i *`

Pointer to the source image rectangle to transfer.

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

If this value is NULL, the entire source image rectangle is transferred.

dst [out]
Type: `NvCVImage *`

Pointer to the destination image to which to transfer the source image.

dstPt [in]
Type: `const NvCVPoint2i *`

Pointer to the destination image location to which to transfer the image.

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

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

scale [in]
Type: `float`

A scale factor to apply 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.

The following are 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 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, such as matching the characteristics of the CPU image. Therefore, for 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 grows 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.

3.17.2. Return Values

- 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.

3.17.3. 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 works only for chunky images.

NvCVImage_TransferRect() works on chunky, planar, and semi-planar image layouts with 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 occurs:

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

3.18. 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,
```

(continues on next page)

(continued from previous page)

```
  unsigned yuvColorSpace,          unsigned yuvMemSpace,
  float scale,
  CUstream stream,
  NvCVImage *tmp
);
```

3.18.1. Parameters

src [in]
Type: `const NvCVImage *`

Pointer to the source image to transfer.

srcRect [in]
Type: `const NvCVRect2i *`

Pointer to the source image rectangle to transfer.

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

If this value is NULL, the entire source image rectangle is transferred.

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: `int`

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 range, chromaticities, and chrominance phase.

yuvMemSpace [in]
Type: `unsigned int`

The memory space where the pixel buffers reside.

scale [in]
Type: `float`

A scale factor to apply 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.

The following are 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 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, such as matching the characteristics of the CPU image. Therefore, for 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 grows 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.

3.18.2. Return Values

- `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.

3.18.3. Remarks

This function is like *NvCVImage_TransferRect()*, which can also copy to YUV images. The difference is that `NvCVImage_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.

3.19. NvCVImage_UnmapResource

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

3.19.1. Parameters

im [in,out]
: Type: `NvCVImage *`

  The image to be unmapped.

stream [out]
: Type: `struct CUStream_st *`

  The stream on which the unmapping is to be performed.

3.19.2. Return Value

`NVCV_SUCCESS` on success.

3.19.3. 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()*.

3.20. C++ Helper Functions for Sample Applications

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

3.20.1. CVWrapperForNvCVImage

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

3.20.1.1 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.

3.20.1.2 Return Value

Does not return a value.

3.20.1.3 Remarks

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

3.20.2. NVWrapperForCVMat

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

3.20.2.1 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.

3.20.2.2 Return Value

Does not return a value.

3.20.2.3 Remarks

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

3.21. Image Functions for C++ Only

The image functions are defined in the `nvCVImage.h` header file. The image API provides constructors, a destructor for C++, and some additional functions that are accessible only to C++.

3.21.1. NvCVImage Default Constructor

The default constructor creates an empty image with no buffer.

```
NvCVImage();
```

3.21.2. NvCVImage 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
);
```

`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. For more information, see *Pixel Organizations*.

memSpace [in]
Type: `unsigned`

The type of memory in which the image data buffers are to be stored. For more information, see *Memory Types*.

alignment [in]
Type: `unsigned`

The row byte alignment, which specifies the alignment of the first pixel in each scan line.

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

> **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`.

3.21.3. NvCVImage 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
);
```

fullImg [in]
Type: `NvCVImage *`

Pointer to the existing image from which to take the view of a specified rectangle in the image.

x [in]
Type: `int`

The x coordinate of the left edge of the view.

y [in]
Type: `int`

The y coordinate of the top edge of the view.

width [in]
Type: `unsigned`

The width, in pixels, of the view.

height [in]
Type: `unsigned`

The height, in pixels, of the view.

3.21.4. NvCVImage Destructor

```
~NvCVImage();
```

3.21.5. copyFrom Function

The following 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
);
```

The following 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
);
```

3.21.5.1 Parameters

src [in]
Type: `const NvCVImage *`

Pointer to the existing source image from which to copy the specified rectangle.

srcX [in]
Type: `int`

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

srcY [in]
Type: `int`

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

dstX [in]
Type: `int`

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

dstY [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 copy.

height [in]
Type: `unsigned`

The height, in pixels, of the rectangle to copy.

3.21.5.2 Return Values

- `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.

3.21.5.3 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 various 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.

Copyright