NVIDIA Video Effects SDK

Programming Guide

PG-09657-001_0.5 Beta | September 2020

Table of Contents

List of Tables

Chapter 1. Introduction to NVIDIA Video Effects SDK

NVIDIA® Video Effects SDK is an SDK for applying effect filters to videos. The SDK is powered by NVIDIA GPUs with Tensor Cores. With Tensor Cores, algorithm throughput is greatly accelerated, and latency is reduced.

The SDK provides the following filters:

- Encoder Artifact Reduction, which reduces the blocking and noisy artifacts from an encoded video.
- Super resolution, which enhances the scale of the video with artifact reduction and
- Upscale, which is a fast upscale for an input video.

Chapter 2. Getting Started with NVIDIA Video Effects SDK

2.1 Hardware and Software Requirements

NVIDIA Video Effects SDK requires specific NVIDIA GPUs and a specific version of Windows OS and other associated software on which the SDK depends.

2.1.1 Hardware Requirements

NVIDIA Video Effects SDK is compatible with GPUs that are based on the NVIDIA® Turing™ architecture. Although Video Effects can run on Turing™ machines without Tensor Cores, it is optimized for much higher performance on machines with Tensor Cores.

2.1.2 Software Requirements

NVIDIA Video Effects SDK requires a specific version of Windows OS and other associated software on which the SDK depends.

Software	Required Version
Windows OS	64-bit Windows 10
Microsoft Visual Studio	2017 or later
CMake	3.12 or later
NVIDIA Graphics Driver for Windows	455.57 or later

2.2 Installing NVIDIA Video Effects SDK

To develop applications with the NVIDIA Video Effects SDK, you must install it before compilation and linking.

The SDK is distributed in the following parts:

- An open source repository that includes the SDK API headers, the sample applications and their dependency libraries, and a proxy file to enable compilation without the SDK DLLs.
- An Installer that installs the SDK DLLs, the models, and the SDK dependency libraries.

For an application that is built on the SDK, the developers can package the DLLs and the other dependencies that were extracted from the installer into the application or require application users to use the SDK installers.

Note: The source code and sample application are hosted on GitHub at `https://github.com/nvidia/BROADCAST-VFX-SDK`.

To use the SDK, download the source code from GitHub and install the SDK binaries. Your application needs to integrate the API headers and call the SDK APIs, and the sample app source code demonstrates how to complete these tasks.

The sample app also includes a file called `NVVideoEffectsProxy.cpp` that is used to link against the SDK DLL without requiring an import library (.lib) file. Having this file allows the compilation of the open source code independently of the SDK Installer. However, the SDK Installer is still required to load the runtime dependencies, the DLLs and models.

The SDK binaries are installed in the `C:\Program Files\NVIDIA Corporation\NVIDIA Video Effects\` directory.

2.3 NVIDIA Video Effects SDK Sample Application

The NVIDIA Video Effects SDK provides sample applications as source code that you can build and as binary files that you can run without building. You can run the application from the supplied batch file or from the application binary file.

2.3.1 Building the Sample Application

The open source repository includes the source code to build the sample application, and a `NVVideoEffectsProxy.cpp` proxy file to enable compilation without explicitly linking against the SDK DLL.

Note: To download the models and runtime dependencies that are required by the features, run the SDK Installer.

1. In the root folder of the downloaded source code, start the CMake GUI and specify the source folder and a build folder for the binary files.
   a. For the source folder, ensure that the path ends in `OSS`.
   b. For the build folder, ensure that the path ends in `OSS/build`.
   c. Use CMake to configure and generate the Visual Studio solution file.
   d. Click **Configure**.
   e. When prompted to confirm that CMake can create the build folder, click **OK**.
   f. Select **Visual Studio** for the generator and **x64** for the platform.
   g. To complete configuring the Visual Studio solution file, click **Finish**.
   h. To generate the Visual Studio Solution file, click **Generate**.

2. Use Visual Studio to generate the application binary `.exe` file from the solution file that you generated in the previous step.
   a. In CMake, to open Visual Studio, click **Open Project**.
   b. In Visual Studio, select **Build** > **Build Solution**.

2.3.2 Running the AR/SR Application

The AR/SR applications refer to the encoder artifact reduction and super resolution features in the SDK, and the following applications demonstrate this feature:

- VideoEffectsApp

  The `VideoEffectsApp` can select one of the following effects:

  - ArtifactReduction
  - SuperRes
  - Upscale

- UpscalePipelineApp

  The `UpscalePipelineApp` provides an effect that is obtained by pipelining ArtifactReduction with Upscale. See "About the AR/SR Filters" on page 11 for more information about these filters.

The application accepts a video file as the input and produces video output, and the output can be stored in a file and optionally displayed in a window. To display the output in a window, specify the `–show` argument.

You can run this application from the supplied `run.bat` shell script or from the application binary file.

2.3.2.1 Running the VideoEffects Application from the Application Binary File

The `run.sh` script is located in the `samples/VideoEffectApp` folder, which is under the NVIDIA Video Effects SDK the root folder. This batch file sets up `LD_LIBRARY_PATH` and sets the options that are required to run the application. The input has been set as an image in the `samples/input` folder, but you can set it as an image or a video (MP4).

1. In a plain text editor, edit `run.sh` to specify your image file or video file as the argument to the `--in_file` option.
2. Set the corresponding output image file or video file as argument to `-out_file`
3. Open a Command Prompt window.
4. Add the sample build directory to your PATH:
   ```
   cd mysamples/build
   export PATH=`pwd`:$PATH
   ```
5. Navigate to the the `samples/VideoEffectApp` folder:
   ```
   cd mysamples/samples/VideoEffectApp
   ```
6. Execute the `run.bat` file.

2.3.2.2 Running the VideoEffects Application from the Command line

If you do not use the `run.sh` script file, you must provide the application access to the required libraries and set all the options that are required to run the application.

1. Open a Command Prompt window.
2. Navigate to the build folder where you built the SDK:
   ```
   $ cd mysamples/build
   ```
3. Configure `LD_LIBRARY_PATH` to reference required libraries:
   ```
   $ export LD_LIBRARY_PATH=/usr/local/VideoFX/lib:/usr/local/cuda-
   10.2/lib64:/usr/local/TensorRT-6.0.1.8
   ```
4. Use the following arguments to execute the `VideoEffectApp`:
   ```

   VideoEffectApp.exe --in_file=input-file --out_file=output-file –
   effect=(ArtifactReduction  |SuperRes | Upscale) –strength=(0|1|[0.0-1.0]
   for upscale) -show
   ```

`--in_file=`*`input-file`*

The image file or video file for the application to process.

`--effect=`*`ArtifactReduction`* or *`SuperRes or Upscale`*

The effect to be applied.

`--resolution=`*`1080 | 1440 | 2160 (for SuperRes or Upscale)`*

The output resolution.

`--strength=`*`(0|1) for SuperRes and ArtifactReduction, [0.0-1.0] for Upscale`*

The effect to be applied.

`--out_file=`*`output-file`*

The file in which to store the video output.

`--show`

Display the resulting video output in a window.

2.3.2.3 VideoEffects Application Command-Line Reference

```
VideoEffectApp [arguments...]
```

Here are the *`arguments`*:

--in_file=*`path`*

The image file or video file for the application to process.

--out_file=*`path`*

The file in which the video output is to be stored.

--effect=`ArtifactReduction` or `SuperRes` or `Upscale`

Selects the effect that will be applied:

- *`ArtifactReduction`*: removes the encoder artifact without changing the resolution.
- *`SuperRes`*:removes artifact and upscales to given output resolution.
- *`Upscale`*: Fast upscaler that increases the video resolution to the specified output resolution.

--resolution=(*`1080 | 1440`*) for the 720p input and (*`1440|2160`*) for the 1080p input (for the SuperRes or Upscale effect)

The output image/video resolution.

--show[=`(true|false)`]

If `true`, displays the resulting video output in a window.

--model_dir=*`path`*

The path to the folder that contains the model files that will be used for the transformation.

`--codec=`*`fourcc`*

The four-character code (FOURCC) of the video codec of the output video file. The default value is `H264`.

`--strength=`(`0|1`) for SuperRes or ArtifactReduction and [`0.0-1.0`] for Upscale

Selects one of the following strength values to apply:

- `0` selects weak effect
- `1` selects strong effect.

`--progress`

Show progress.

`--verbose [=(true|false)]`

verbose output

`--debug[=(true|false)]`

print extra debugging

2.3.2.4 Running the UpscalePipeline Application from the Application Binary File

The `run.sh` script is located in the `samples/UpscalePipelineApp` folder under the NVIDIA Video Effects SDK root folder . This script configures the `LD_LIBRARY_PATH` and sets the options that are required to run the application. The input is set as an image in the `samples/input` folder. You can set the input as an image or a video (MP4).

1. In a plain-text editor, edit `run.sh` to specify your image file or video file as the argument for the `--in_file` option.
2. Set the corresponding output image file or video file as the argument for the `–out_file` option.
3. Open a Command Prompt window and add the sample build directory to your PATH:
   ```
   cd mysamples/build
   export PATH=`pwd`:$PATH
   ```
4. Navigate to the `samples/UpscalePipelineApp` folder:
   ```
   cd mysamples/samples/UpscalePipelineApp
   ```
5. Execute the `run.bat` file.

2.3.2.5 Running the UpscalePipeline Application from the Command line

If you do not use the `run.sh` script file, you must provide the application access to the required libraries and set the options required to run the application.

1. Open Command Prompt window.
2. Navigate to the build folder where you built the SDK.

   ```
   $ cd mysamples/build
   ```

3. Set up `LD_LIBRARY_PATH` to reference required libraries.

   ```
   $ export LD_LIBRARY_PATH=/usr/local/VideoFX/lib:/usr/local/cuda-
   10.2/lib64:/usr/local/TensorRT-6.0.1.8
   ```

4. Execute the `App` with the following arguments.

   ```

   UpscalePipelineApp.exe --in_file=input-file --out_file=output-file --
   ar_strength=(0|1) --sr_strength=[0.0-1.0] --resolution=(1080|1440 for
   720p, 1440|2160 for 1080p input) --show
   ```

   `--in_file=`*`input-file`*

   The image or video file for the application to process.

   `--resolution=`*`1080`* / *`1440`* / *`2160`*

   The output resolution.

   `--ar_strength=`*`(0|1)`*

   Strength of the AR filter that will be applied.

   `--sr_strength=`*`[0.0-1.0]`*

   Strength of the Upscale filter that will be applied.

   `--out_file=`*`output-file`*

   The file in which to store the video output.

   `--show`

   Display the resulting video output in a window.

2.3.2.6 UpscalePipeline Application Command-Line Reference

```
UpscalePipelineApp [arguments...]
```

Here are the `arguments`:

----in_file=*path*

The image file or video file for the application to process.

--out_file=*path*

The file in which the video output will be stored.

--resolution= *(1080| 1440)* for the 720p inputand *(1440|2160)* for the 1080p input

The output image/video resolution.

--show[=`(true|false)`]

If `true`, displays the resulting video output in a window.

--model_dir=*path*

The path to the folder that contains the model files that will be used for the transformation.

--codec=*fourcc*

The four-character code (FOURCC) of the video codec of the output video file. The default value is `H264`.

--ar_strength=(`0|1`) for ArtifactReduction

Selects the strength of the artifact reduction filter that will be applied:

- `0` selects the weak effect.
- `1` selects the strong effect.

--sr_strength=[`0.0-1.0`] for Upscale

Selects the strength of the upscale filter to be applied.

- `0` selects the weak effect.
- `1` selects the strong effect.
- `0.5` is the suggested optimal value.

--progress

Shows the progress.

`--verbose [=(true|false)]`

Verbose output.

`--debug[=(true|false)]`

Prints extra debugging information.

--show[=(`true|false`)]

If `true`, displays the resulting video output in a window.

--model_dir=*`path`*

The path to the folder that contains the model files that will be used for the transformation.

--codec=*`fourcc`*

The four-character code (FOURCC) of the video codec of the output video file. The default value is `H264`.

Chapter 3. Using NVIDIA Video Effects SDK in Applications

You can use the NVIDIA Video Effects SDK to enable an application to apply effect filters to videos. The NVIDIA Video Effects API is object oriented but is accessible to C in addition to C++.

3.1 About the AR/SR Filters

The AR/SR filter contains the following filters that can be applied to an input video:

- Encoder Artifact reduction
  Reduces encoder artifacts, such as blocking artifacts, ringing, mosquito noise from a low bitrate source. The encoder artifact reduction has been optimized for the H.264 encoder.
- Super resolution
  Enhances the resolution of a low-resolution low bitrate video. In addition to an increase in the resolution, this effect reduces encoder artifacts and enhances details.
- Upscale
  This is a fast method to enhance the resolution of a low-resolution video.

You can use one of the following sample apps for these filters:

- VideoEffectsApp

  This app runs each effect individually and can be used to apply one filter to the input video.

- UpscalePipelineApp

  This app runs a pipeline of Encoder Artifact reduction and Upscale and can be used with a fast application that performs artifact reduction and scale enhancement.

The input resolutions for both the apps should be 720p (720 x 1280) or 1080p (1080 x 1920).

3.1.1 Output Resolutions

Here is some information about output resolutions.

- For 720p input videos, we currently support 1.5x (1080p) or 2x (1440p) outputs.
- For 1080p input videos, we currently support 1.33x (1440p) or 2x (2160p) outputs.

3.1.1.1 Filter Input and Output

The filter's input/output is as follows:

- The input should be provided in a GPU buffer in BGR interleaved, where each pixel is a 24-bit unsigned char value.
- The output of the filter is written to a BGR interleaved GPU buffer.

3.1.1.2 Strength Options

Here is some information about the available strength options:

- The ArtifactReduction and SuperRes filters contain the `0` and `1` strength options, where `0` applies a weak effect, and `1` applies a strong effect.
- ArtifactReduction
  - Option `0` removes lesser artifacts and preserves low gradient information better and is suited for higher bitrate videos.
  - Option `1` is better suited for lower bitrate videos.
- SuperRes
  - Option `0` enhances less, removes more artifacts, and is suited for lower quality videos.
  - Option `1` enhances more, removes lesser artifacts, and is suited for high quality videos.
- Upscale

  This filter provides a floating point strength value, which ranges between 0.0 and 1.0 and signifies an enhancement parameter.
  - Option `0` implies no enhancement, only upscaling.
  - Option `1` implies a maximum enhancement.
  - The optimally suggested value is 0.5.
- The UpscalePipeline app demonstrates the pipelining of the ArtifactReduction and Upscale effects.

3.2 Using a Video Effect Filter

Using a video effect filter involves creating the filter, setting up various properties of the filter, and then loading, running, and destroying the filter.

3.2.1 Creating a Video Effect Filter

Call the `NvVFX_CreateEffect()` function, specifying the following information as parameters:

- The `NvVFX_EffectSelector` type `NVVFX_FX_ARTIFACT_REDUCTION`, `NVVFx_FX_SUPER_RES`, and `NVVFX_FX_SR_UPSCALE`.

- The location in which to store the handle to the newly created video effect filter.

The `NvVFX_CreateEffect()` function creates a handle to the video effect filter instance for use in further API calls.

3.2.2 Setting the Path to the Model Folder

A video effect filter requires a neural network model for transforming the input still or video image. You must set the path to the folder that contains the files that describe the model to be used by the filter.

Call the `NvVFX_SetString()` function, specifying the following information as parameters:

- The filter handle that was created as explained in "Creating a Video Effect Filter" on page 12.
- The selector string `NVVFX_MODEL_DIRECTORY`.
- A null-terminated string that indicates the path to the model folder.

This example sets the path to the model folder to `C:\Users\vfxuser\Documents\vfx\models`.

```
const char* modelDir="C:\Users\vfxuser\Documents\vfx\models";
...
vfxErr = NvVFX_SetString(effectHandle, NVVFX_MODEL_DIRECTORY, modelDir);
```

3.2.3 Setting Up the CUDA Stream

A video effect filter requires a CUDA stream in which to run. For information about CUDA streams, see the NVIDIA CUDA Toolkit Documentation.

1. Initialize a CUDA stream by calling one of the following functions.
   - `NvVFX_CudaStreamCreate()`
   - The CUDA Runtime API function `cudaStreamCreate()`

   Use the `NvVFX_CudaStreamCreate()` function to avoid linking with the NVIDIA CUDA Toolkit libraries.
2. Call the `NvVFX_SetCudaStream()` function, providing the following information as parameters:
   - The filter handle that was created as explained in "Creating a Video Effect Filter" on page 12.
   - The selector string `NVVFX_CUDA_STREAM`.
   - The CUDA stream that you created in the previous step.

This example sets up a CUDA stream that was created by calling the `NvVFX_CudaStreamCreate()` function.

```
CUstream stream;
...
vfxErr = NvVFX_CudaStreamCreate (&stream);
...
vfxErr = NvVFX_SetCudaStream(effectHandle, NVVFX_CUDA_STREAM, stream);
```

3.2.4 Setting the Input and Output Image Buffers

Each filter takes a GPU `NvCVImage` structure as input and produces the result in a GPU `NvCVImage` structure. These images are GPU buffers accepted by the filter. The application provides input and output buffers to the filter by setting them as required parameters.

The video effect filter requires the input to be provided in a GPU buffer in BGR interleaved format, where each pixel is a 24-bit unsigned `char` value. If the original buffer is of type CPU/GPU or is in planar format, it must be converted as explained in "Transferring Images Between CPU and GPU Buffers" on page 20.

For each image buffer, call the `NvVFX_SetImage()` function, specifying the following information as parameters:

- The filter handle that was created as explained in "Creating a Video Effect Filter" on page 12.
- The selector string that denotes the type of buffer that you are creating:
  - For the input image buffer, use `NVVFX_INPUT_IMAGE`.
  - For the output (mask) image buffer, use `NVVFX_OUTPUT_IMAGE`.
- The address of the `NvCVImage` object created for the input or output image

This example creates an input image buffer.

```
NvCVImage srcGpuImage;
...
vfxErr = NvCVImage_Alloc(&srcGpuImage, 960, 540, NVCV_BGR, NVCV_U8,
NVCV_CHUNKY, NVCV_GPU, 1)
...
vfxErr = NvVFX_SetImage(effectHandle, NVVFX_INPUT_IMAGE, &srcGpuImage);
```

This example creates an output image buffer.

```
NvCVImage srcGpuImage;
...
vfxErr = NvCVImage_Alloc(&dstGpuImage, 960, 540, NVCV_A, NVCV_U8,
NVCV_CHUNKY, NVCV_GPU, 1))
...
vfxErr = NvVFX_SetImage(effectHandle, NVVFX_OUTPUT_IMAGE, &dstGpuImage);
```

3.2.5 Setting and Getting Other Parameters of a Video Effect Filter

Before loading and running a video effect filter, set any other parameters that the filter requires. NVIDIA Video Effects SDK provides type-safe set accessor functions for this purpose. If you need the value of a parameter that has by a set accessor function, use the corresponding get accessor function.

In the call to each set and get accessor function, provide the following information as parameters:

- The filter handle that was created as explained in "Creating a Video Effect Filter" on page 12.
- The selector string for the parameter that you want to access.
- The value that you want to set or a pointer to a location in which to store the value that you want to get.

3.2.5.1 Summary of NVIDIA Video Effects SDK Accessor Functions

Parameter Type	Data Type	Set and Get Accessor Function
32-bit unsigned integer	`unsigned int`	`NvVFX_SetU32()`
		`NvVFX_GetU32()`
Single-precision (32-bit) floating-point number	`float`	`NvVFX_SetF32()`
Image buffer	`NvCVImage`	`NvVFX_SetImage()`
Character string	`const char*`	`NvVFX_SetString()`
CUDA stream	`CUstream`	`NvVFX_SetCudaStream()`
		`NvVFX_GetCudaStream()`

3.2.5.2 Getting Information About a Filter and its Parameters

To get information about a filter and its parameters, call the `NvVFX_GetString()` function, specifying the `NVVFX_INFO` type of the `NvVFX_ParameterSelector` type definition.

```
NvCV_Status NvVFX_GetString(
  NvVFX_Handle obj,
  NVVFX_INFO,
  const char **str
);
```

3.2.5.3 Getting a List of All Available Effects

To get a list of the available effects, call the `NvVFX_GetString()` function, specifying `NULL` for the `NvVFX_Handle` object handle.

```
NvCV_Status NvVFX_GetString(NULL, NVVFX_INFO, const char **str);
```

3.2.6 Loading a Video Effect Filter

Loading a filter selects and loads an effect model and validates the parameters that were set for the filter.

Note: Some video effect filters have settings that can be modified only after the filter has been loaded.

To load a video effect filter, call the `NvVFX_Load()` function, specifying the filter handle that was created as explained in "Creating a Video Effect Filter" on page 12.

```
vfxErr = NvVFX_Load(effectHandle);
```

Note: If a set accessor function is used to change filter parameters, the filter must be reloaded before it is run.

3.2.7 Running a Video Effect Filter

After loading a video effect filter, run the filter to apply the desired effect. When a filter is run, the contents of the input GPU buffer are read, the video effect filter is applied, and the output is written to the output GPU buffer.

To run a video effect filter, call the `NvVFX_Run()` function. In the call to the `NvVFX_Run()` function, pass the following information as parameters:

- The filter handle that was created as explained in "Creating a Video Effect Filter" on page 12.
- An integer value to specify whether the filter is to run asynchronously or synchronously:
  - `1`: The filter is to run asynchronously.
  - `0`: The filter is to run synchronously.

This example runs a video effect filter asynchronously and calls the `NvCVImage_Transfer()` function to copy the output into a CPU buffer.

```
vfxErr = NvVFX_Run(effectHandle, 1);
vfxErr = NvCVImage_Transfer()
```

3.2.8 Destroying a Video Effect Filter

When a video effect filter is no longer required, destroy it to free resources and memory that were allocated for the filter.

To destroy a video effect filter, call the `NvVFX_DestroyEffect()` function, specifying the filter handle that was created as explained in "Creating a Video Effect Filter" on page 12.

```
NvVFX_DestroyEffect(effectHandle);
```

3.3 Working with Image Frames on GPU or CPU Buffers

Effect filters accept image buffers as `NvCVImage` objects. The image buffers can be either CPU or GPU buffers, but for performance reasons, the effect filters require GPU buffers . NVIDIA Video Effects SDK provides functions for converting an image representation to `NvCVImage` and transferring images between CPU and GPU buffers.

3.3.1 Converting Image Representations to NvCVImage Objects

NVIDIA Video Effects SDK provides functions for converting 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.

3.3.1.1 Converting OpenCV Images to NVCVImage Objects

Use the wrapper functions that NVIDIA Video Effects SDK provides specifically for RGB OpenCV images.

Note: NVIDIA Video Effects SDK provides 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);
```

3.3.1.2 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);
```

3.3.2 Allocating an NvCVImage Object Buffer

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

3.3.2.1 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. See "Allocation Constructor" on page 66 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
);
```

3.3.2.2 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 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()`.

3.3.3 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.

3.3.3.1 Transferring 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 will be allocated or reallocated as needed, if needed.
2. 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);
   ```

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

3.3.3.2 Transferring Output Images from a GPU Buffer to a CPU Buffer

To transfer an image from the GPU to a CPU buffer with conversion, given

```
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 will be allocated or reallocated as needed, if needed.
2. 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.

3.4 Using Multiple GPUs

Applications developed with the NVIDIA Video Effects SDK can be used with multiple GPUs. By default, the SDK determines which GPU to use based on the capability of the currently selected GPU: If the currently selected GPU supports the NVIDIA Video Effects SDK, the SDK uses it. Otherwise, the SDK chooses the best GPU. You can control which GPU is used in a multi-GPU environment by using the NVIDIA CUDA Toolkit functions `cudaSetDevice(int whichGPU)` and `cudaGetDevice(int *whichGPU)` and the Video Effects Set function `NvVFX_SetS32(NULL, NVVFX_GPU, whichGPU)`. The `Set()` call is intended to be called only once for the Video Effects SDK, before any effects are created. Images that are allocated on one GPU cannot be transparently passed to another GPU, so you must ensure that the same GPU is used for all video effects.

```
NvCV_Status err;
int chosenGPU = 0; // or whatever GPU you want to use
err = NvVFX_SetS32(NULL, NVVFX_GPU, chosenGPU);
if (NVCV_SUCCESS != err) {
    printf("Error choosing GPU %d: %s\n", chosenGPU,
            NvCV_GetErrorStringFromCode(err));
}
```

```
cudaSetDevice(chosenGPU);
NvCVImage *dst = new NvCVImage(…);
NvVFX_Handle eff;
err = NvVFX_API NvVFX_CreateEffect(code, &eff);
err = NvVFX_API NvVFX_SetImage(eff, NVVFX_OUTPUT_IMAGE, dst);
…
err = NvVFX_API NvVFX_Load(eff);
err = NvVFX_API NvVFX_Run(eff, true);
// switch GPU for other task, then switch back for next frame
```

Buffers need to be allocated on the selected GPU, so **before** you allocate images on this GPU, call `cudaSetDevice()`. Neural networks need to be loaded on the selected GPU, so before `NvVFX_Load()` is called, set this GPU as the current device.

To use the buffers and models, **before** you call `NvVFX_Run()`, the GPU device needs to be current. A previous call to `NvVFX_SetS32(NULL, NVVFX_GPU, whichGPU)` helps enforce this requirement.

For performance concerns, switching to the appropriate GPU is the responsibility of the application.

3.4.1 Default Behavior in Multi-GPU Environments

The `NvVFX_Load()` function internally calls `cudaGetDevice()` to identify the currently selected GPU. It then checks the compute capability of the currently selected GPU (default 0) to determine if the GPU architecture supports the NVIDIA Video Effects SDK.

- If so, `NvVFX_Load()` uses the GPU.
- Otherwise, `NvVFX_Load()` searches for the most powerful GPU that supports the NVIDIA Video Effects SDK and calls `cudaSetDevice()` to set that GPU as the current GPU.

If you do not require your application to use a specific GPU in a multi-GPU environment, the default behavior should suffice.

3.4.2 Selecting the GPU for Video Effects Processing in a Multi-GPU Environment

Your application might be designed to perform only the task of applying a video effect filter by using in a specific GPU in a multi-GPU environment. In this situation, ensure that the NVIDIA Video Effects SDK does not override your choice of GPU for applying the video effect filter.

```
// Initialization
cudaGetDevice(&beforeGPU);
vfxErr = NvVFX_Load(eff);
if (NVCV_SUCCESS != vfxErr) { printf("Cannot load VFX: %s\n",
   NvCV_GetErrorStringFromCode(vfxErr)); exit(-1); }
cudaGetDevice(&vfxGPU);
if (beforeGPU != vfxGPU) {
  printf("GPU #%d cannot run VFX, so GPU #%d was chosen instead\n",
    beforeGPU, vfxGPU);
}
vfxErr = NvVFX_SetImage() ...
...
```

3.4.3 Selecting Different GPUs for Different Tasks

Your application might be designed to perform multiple tasks in a multi-GPU environment, for example, rendering a game and applying a video effect filter. In this situation, select the best GPU for each task before calling `NvVFX_Load()`.

1. Call `cudaGetDeviceCount()` to determine the number of GPUs in your environment.

   ```
   // Get the number of GPUs
   cuErr = cudaGetDeviceCount(&deviceCount);
   ```

2. Get the properties of each GPU and determine if it is the best GPU for each task by performing the following operations for each GPU in a loop.
   a. Call `cudaSetDevice()` to set the current GPU.
   b. Call `cudaGetDeviceProperties()` or preferably `cudaDeviceGetAttribute ()` to get the properties of the current GPU.
   c. Use custom code in your application to analyze the properties retrieved by `cudaGetDeviceProperties()` or `cudaDeviceGetAttribute()` to determine if the GPU is the best GPU for each specific task.

This example uses the compute capability to determine if a GPU's properties should be analyzed to determine if the GPU is the best GPU for applying a video effect filter. A GPU's properties are analyzed only if the compute capability is 7.5, which denotes a GPU based on the NVIDIA Turing GPU architecture.

```
// Loop through the GPUs to get the properties of each GPU and
//determine if it is the best GPU for each task based on the
//properties obtained.
for (int dev = 0; dev < deviceCount; ++dev) {
  cudaSetDevice(dev);
  cudaGetDeviceProperties(&deviceProp, dev);
  if (DeviceIsBestForVFX(&deviceProp))  gpuVFX = dev; // 7.5 compute
  if (DeviceIsBestForGame(&deviceProp)) gpuGame = dev;
  ...
}
cudaSetDevice(gpuVFX);
vfxErr = NvVFX_Set...; // set parameters
vfxErr = NvVFX_Load(eff);
```

3. In the loop for performing the application's tasks, select the best GPU for each task before performing the task.
   a. Call `cudaSetDevice()` to select the GPU for the task.
   b. Make all the function calls required to perform the task.

   In this way, you select the best GPU for each task only once without setting the GPU for every function call.

   This example selects the best GPU for rendering a game and uses custom code to render the game. It then selects the best GPU for applying a video effect filter before calling the `NvCVImage_Transfer()` and `NvVFX_Run()` functions to apply the filter, avoiding the need to save and restore the GPU for every NVIDIA Video Effects SDK API call.

```
// Select the best GPU for each task and perform the task.
while (!done) {
  ...
  cudaSetDevice(gpuGame);
  RenderGame();
  cudaSetDevice(gpuVFX);
  vfxErr = NvCVImage_Transfer(&srcCPU, &srcGPU, 1.0f, stream, &tmpGPU);
  vfxErr = NvVFX_Run(eff, 1);
  vfxErr = NvCVImage_Transfer(&dstGPU, &dstCPU, 1.0f, stream, &tmpGPU);
  ...
}
```

Chapter 4. NVIDIA Video Effects SDK API Reference

4.1 Structures

The structures in the NVIDIA Video Effects SDK are defined in the following header files:

- `nvVideoEffects.h`
- `nvCVImage.h`

4.1.1 NvVFX_Handle

```
typedef struct NvVFX_Object NvVFX_Object, *NvVFX_Handle;
```

This structure represents the opaque handle that is associated with each instance of a video effect filter. It is a pointer to an opaque object of type `nvwarpObject`. Most video effect function calls include this handle as the first parameter.

Defined in: `nvVideoEffects.h`.

4.1.2 NvVFX_Object

```
struct NvVFX_Object;
```

This structure represents an opaque video effect filter object that is allocated by the `NvVFX_CreateEffect()` function and deallocated by the `NvFX_DestroyEffect()` function.

Defined in: `nvVideoEffects.h`.

4.1.3 NvCVImage

```
typedef struct NvCVImage {
  unsigned int              width;
  unsigned int              height;
  unsigned int              pitch;
  NvCVImage_PixelFormat   pixelFormat;
  NvCVImage_ComponentType  componentType;
  unsigned char             pixelBytes;
  unsigned char             componentBytes;
  unsigned char             numComponents;
  unsigned char             planar;
  unsigned char             gpuMem;
  unsigned char             colorspace;
  unsigned char             batch;
  void                      *pixels;
  void                      *deletePtr;
  void                      (*deleteProc)(void *p);
  unsigned long long        bufferBytes;

} NvCVImage;
```

4.1.3.1 Members

width

Type: `unsigned int`

The width, in pixels, of the image.

height

Type: `unsigned int`

The height, in pixels, of the image.

pitch

Type: `unsigned int`

The vertical byte stride between pixels.

pixelFormat

Type: `NvCVImage_PixelFormat`

The format of the pixels in the image.

componentType

Type: `NvCVImage_ComponentType`

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

pixelBytes

Type: `unsigned char`

The number of bytes in a chunky pixel.

componentBytes

Type: `unsigned char`

The number of bytes in each pixel component.

numComponents

Type: `unsigned char`

The number of components in each pixel.

planar

Type: `unsigned char`

Specifies the organization of the pixels in the image.

- 0: Chunky
- 1: Planar

gpuMem

Type: `unsigned char`

Specifies the type of memory in which the image data buffer is stored. The different types of memory have different address spaces.

- 0: CPU memory
- 1: CUDA memory
- 2: pinned CPU memory

colorspace

Type: `unsigned char`

Specifies a logical OR group of YUV color space types, for example:

```

my422.colorspace = NVCV_709 | NVCV_VIDEO_RANGE | NVCV_CHROMA_COSITED;
```

See “YUV Color Spaces” on page 34 for more information about the type definitions.

Always set the colorspace for 420 or 422 YUV images. The default colorspace is `NVCV_601 | NVCV_VIDEO_RANGE | NVCV_CHROMA_COSITED`.

batch

Type: `unsigned char`

Set this parameter to 1.

pixels

Type: `void`

Pointer to pixel (0,0) in the image.

deletePtr

Type: `void`

Buffer memory to be deleted (can be `NULL`).

deleteProc

Type: `void`

The function to call instead of `free()` to delete the pixel buffer. To call `free()`, set this parameter to `NULL`. The image allocators use `free()` for CPU buffers and `cudaFree()` for GPU buffers.

bufferBytes

Type: `unsigned long long`

The maximum amount of memory in bytes that is available through pixels.

4.1.3.2 Remarks

This structure defines the properties of an image in an image buffer that is provided as input to an effect filter. The members can be set by using the setter functions in the NVIDIA Video Effects SDK API.

Defined in: `nvCVImage.h`.

4.2 Enumerations

The enumerations in the NVIDIA Video Effects SDK are defined in the header file `nvCVImage.h`.

4.2.1 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`).

4.2.2 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_YUV420

Luminance and subsampled Chrominance (Y, Cb, Cr).

NVCV_YUV422

Luminance and subsampled Chrominance (Y, Cb, Cr).

4.3 Type Definitions

NVIDIA Video Effects SDK type definitions provide selector strings for video effect filters, the parameters of a video effect filter, pixel organizations, and memory types.

4.3.1 NvVFX_EffectSelector

```
typedef const char* NvVFX_EffectSelector;
```

This type definition provides the selector strings for the various types of video effect filters.

NVVFX_FX_TRANSFER "Transfer"

Image transfer effect.

This effect provides the same capability as the `NvCVImage_Transfer()` function in the form of an effect. This effect is especially useful to match formats in a pipeline of effects.

NVVFX_FX_ARTIFACT_REDUCTION "Artifact Reduction"

AI based artifact reduction

NVVFX_FX_SUPER_RES "Super Res"

AI based super resolution

NVVFX_FX_SR_SCALER "Upscale"

AI based fast video upscaler

4.3.2 NvVFX_ParameterSelector

```
typedef const char* NvVFX_ParameterSelector;
```

This type definition provides the selector strings for the parameters of a video effect filter.

NVVFX_INPUT_IMAGE_0 "SrcImage0"

The `NvCVImage` structure that will be used as the input to the effect.

Because no effect takes more than one input image, this selector is equivalent to `NVVFX_INPUT_IMAGE`.

NVVFX_INPUT_IMAGE NVVFX_INPUT_IMAGE_0

The `NvCVImage` structure that will be used as the input to the effect.

NVVFX_OUTPUT_IMAGE_0 "DstImage0"

The `NvCVImage` structure that will be used as the output of the effect.

Because no effect has more than one output image, this selector is equivalent to `NVVFX_OUTPUT_IMAGE`.

NVVFX_OUTPUT_IMAGE NVVFX_OUTPUT_IMAGE_0

The `NvCVImage` structure that will be used as the output of the effect.

NVVFX_MODEL_DIRECTORY "ModelDir"

The path to the folder that contains the model files that will be used for the transformation.

NVVFX_CUDA_STREAM "CudaStream"

The CUDA stream in which to run the video effect filter.

NVVFX_INFO "Info"

Get information about a video effect filter and its parameters.

NVVFX_SCALE "Scale"

Scale factor.

NVVFX_INTENSITY "Intensity"

The intensity with which the video effect will be applied.

NVVFX_MODE "Mode"

The mode of a filter.

- 0: Quality mode
- 1: Performance mode

NVVFX_TEMPORAL "Temporal"

Apply temporal filtering.

NVVFX_SCALE "Scale"

Scale factor for the filters that use this parameter.

NVVFX_STRENGTH "Strength"

Strength for the filters that use this parameter. Higher strength implies stronger effect.

4.3.3 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, for example, the red components, of all pixels in the image are grouped together.

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 NVIDIA Video Effects SDK API defines the following types to specify the pixel organization:

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 6
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_YUV 3
NVCV_I420 3
NVCV_IYUV 3

> Each of these types specifies one of the following planar YUV arrangements:
>
> - YUV 4:2:2
> - YUV 4:2:0

Pixels are arranged in [Y], [U], [V] groups.

NVCV_YVU 5
NVCV_YV12 5

Each of these types specifies YV12 pixels, which are in the planar YUV 4:2:2 format or the planar YUV 4:2:0 format.

Pixels are arranged in [Y], [V], and [U] groups.

NVCV_YCUV 7
NVCV_NV12 7

Each of these types specifies NV12 pixels, which are in the semiplanar YUV 4:2:2 format or the semiplanar YUV 4:2:0 format (default for 4:2:0).

Pixels are arranged in [Y] and [UV] groups.

NVCV_YCVU 9
NVCV_NV21 9

Each of these types specifies NV21 pixels, which are in the semiplanar YUV 4:2:2 format or the semiplanar YUV 4:2:0 format.

Pixels are arranged in [Y] and [VU] 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.

4.3.4 YUV Color Spaces

The NVIDIA Video Effects SDK API defines the following types to specify the YUV color spaces:

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 0
NVCV_CHROMA_MPEG2 0

Each of these types specifies a color space in which the chroma is sampled at the same location as the luma samples horizontally.

NVCV_CHROMA_INTSTITIAL	8
NVCV_CHROMA_MPEG1	8

Each of these types specifies a color space in which the chroma is sampled between luma samples horizontally.

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_INTSTITIAL;
```

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_INTSTITIAL;
```

These are particularly useful and performant for interfacing to the NVDEC video decoder.

4.3.5 Memory Types

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

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.

4.4 Video Effects Functions

The video effects functions are defined in the `nvVideoEffects.h` header file. The video effects API is object oriented but is accessible to C and C++.

4.4.1 NvVFX_CreateEffect

```
NvCV_Status NvVFX_CreateEffect(
  NvVFX_EffectSelector code,
..NvVFX_Handle *obj
);
```

4.4.1.1 Parameters

code [in]

Type: `NvVFX_EffectSelector`

The selection string for the type of video effect filter to be created. See "NvVFX_EffectSelector" on page 31 for more information about the allowed selection strings.

obj [out]

Type: `NvVFX_Handle *`

The location in which to store the handle to the newly created video effect filter instance.

4.4.1.2 Return Value

`NVFVX_SUCCESS` on success

4.4.1.3 Remarks

This function creates an instance of the specified type of video effect filter. The function writes a handle to the video effect filter instance to the out parameter `obj`.

4.4.2 NvVFX_CudaStreamCreate

```
NvCV_Status NvVFX_CudaStreamCreate(
  CUstream *stream
);
```

4.4.2.1 Parameters

stream [out]

Type: `CUstream *`

The location in which to store the newly allocated CUDA stream.

4.4.2.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_CUDA_VALUE` if a CUDA parameter is not within its acceptable range.

4.4.2.3 Remarks

This function creates a CUDA stream. It is a wrapper for the CUDA Runtime API function `cudaStreamCreate()` that you can use to avoid linking with the NVIDIA CUDA Toolkit libraries. This function and `cudaStreamCreate()` are equivalent and interchangeable.

4.4.3 NvVFX_CudaStreamDestroy

```
void NvVFX_CudaStreamDestroy(
  CUstream stream
);
```

4.4.3.1 Parameters

stream [in]

Type: `CUstream`

The CUDA stream to destroy.

4.4.3.2 Return Value

Does not return a value.

4.4.3.3 Remarks

This function destroys a CUDA stream. It is a wrapper for the CUDA Runtime API function `cudaStreamDestroy()` that you can use to avoid linking with the NVIDIA CUDA Toolkit libraries. This function and `cudaStreamDestroy()` are equivalent and interchangeable.

4.4.4 NvVFX_DestroyEffect

```
void NvVFX_DestroyEffect(
  NvVFX_Handle obj
);
```

4.4.4.1 Parameters

obj [in]

Type: `NvVFX_Handle`

The handle to the video effect filter instance to be destroyed.

4.4.4.2 Return Value

Does not return a value.

4.4.4.3 Remarks

This function destroys the video effect filter instance with the specified handle and frees resources and memory that were allocated for it.

4.4.5 NvVFX_GetCudaStream

```
NvCV_Status NvVFX_GetCudaStream(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  CUstream *stream
);
```

4.4.5.1 Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance from which you want to get the CUDA stream.

paramName

Type: `NvVFX_ParameterSelector`

The `NVVFX_CUDA_STREAM` selector string. Any other selector string returns an error.

stream

Type: `CUstream *`

Pointer to the CUDA stream where the CUDA stream retrieved will be written.

4.4.5.2 Return Value

- NVFVX_SUCCESS on success.
- NVCV_ERR_SELECTOR when the obj parameter specifies a video effect filter instance that cannot parse the selector string that was specified by the paramName parameter or the data type of the parameter that the selector string specifies.
- NVCV_ERR_PARAMETER when an unexpected NULL pointer was supplied.

4.4.5.3 Remarks

This function gets the CUDA stream in which the specified video effect filter will run and writes the retrieved CUDA stream to the location that was specified by the parameter stream.

4.4.6 NvCV_GetErrorStringFromCode

```
NvCV_GetErrorStringFromCode(NvCV_Status code);
```

4.4.6.1 Parameters

code

Type: NvCV_Status

The NvCV_Status code for which to get an error string.

4.4.6.2 Return Value

The error string that corresponds to the specified error code.

4.4.6.3 Remarks

This function gets the error string that corresponds to the status code that was specified by the parameter code.

4.4.7 NvVFX_GetF32

```
NvCV_Status NvVFX_GetF32(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  float *val
);
```

4.4.7.1 Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance from which you want to get the specified 32-bit floating-point parameter.

paramName

Type: `NvVFX_ParameterSelector`

The `NVVFX_SCALE` selector string. Any other selector string returns an error.

val

Type: `float *`

Pointer to the floating-point number where the value that was retrieved will be written.

4.4.7.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_SELECTOR` when the `obj` parameter specifies a video effect filter instance that cannot parse the selector string that was specified by the `paramName` parameter or the data type of the parameter that was specified by the selector string.
- `NVCV_ERR_PARAMETER` when an unexpected `NULL` pointer was supplied.

4.4.7.3 Remarks

This function gets the value of the specified single-precision (32-bit) floating-point parameter for the specified video effect filter and writes the value that was retrieved to the location that was specified by the `val` parameter.

4.4.8 NvVFX_GetImage

```
NvCV_Status NvVFX_GetImage(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  NvCVImage *im
);
```

4.4.8.1 Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance from which you want to get the specified image buffer.

paramName

Type: `NvVFX_ParameterSelector`

One of the following selector strings for the image buffer that you want to get:

- `NVVFX_INPUT_IMAGE_0`
- `NVVFX_INPUT_IMAGE`
- `NVVFX_OUTPUT_IMAGE_0`
- `NVVFX_OUTPUT_IMAGE`

Any other selector string returns an error.

im

Type: `NvCVImage *`

Pointer to an empty `NvCVImage` structure where a view of the requested image is to be written.

4.4.8.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_SELECTOR` when the `obj` parameter specifies a video effect filter instance that cannot parse the selector string that was specified by the `paramName` parameter or the data type of the parameter that was specified by the selector string.
- `NVCV_ERR_PARAMETER` when an unexpected `NULL` pointer was supplied.

4.4.8.3 Remarks

This function gets the specified input or output image descriptor for the specified video effect filter and writes it to the location that was specified by the `im` parameter. The retrieved image descriptor is a copy of the descriptor that was supplied in an earlier call to `NvVFX_SetImage()`. If `NvVFX_SetImage()` has not been called previously with the same selector, the location that was specified by `im` is filled with zeros. The buffer is not deallocated when the supplied `NvCVImage` object goes out of scope.

4.4.9 NvVFX_GetString

```
NvCV_Status NvVFX_GetString(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  const char **str
);
```

4.4.9.1 Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance from which you want to get the specified character string parameter. To get a list of the available effects, set the `obj` parameter to `NULL` and the `paramName` parameter to `NVVFX_INFO`.

paramName

Type: `NvVFX_ParameterSelector`

One of the following selector strings for the character string parameter that you want to get:

- `NVVFX_INFO`
- `NVVFX_MODEL_DIRECTORY`

Any other selector string returns an error.

str

Type: `const char **`

The address where the requested character string pointer will be stored.

4.4.9.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_SELECTOR` when the `obj` parameter specifies a video effect filter instance that cannot parse the selector string that was specified by the `paramName` parameter or the data type of the parameter that was specified by the selector string.
- `NVCV_ERR_PARAMETER` when an unexpected `NULL` pointer was supplied.

4.4.9.3 Remarks

This function gets the value of the specified character string parameter for, or information about, the specified video effect filter and writes the string that was retrieved to the location that was specified by the `str` parameter.

4.4.10 NvVFX_GetU32

```
NvCV_Status NvVFX_GetU32(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector
  paramName,
  unsigned int *val
);
```

4.4.10.1 Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance from which you want to get the specified 32-bit unsigned integer parameter.

paramName

Type: `NvVFX_ParameterSelector`

The `NVVFX_MODE` selector string. Any other selector string returns an error.

val

Type: `unsigned int *`

Pointer to the 32-bit unsigned integer where the value retrieved is to be written.

4.4.10.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_SELECTOR` when the `obj` parameter specifies a video effect filter instance that cannot parse the selector string that was specified by the `paramName` parameter or the data type of the parameter that was specified by the selector string.
- `NVCV_ERR_PARAMETER` when an unexpected `NULL` pointer was supplied.

4.4.10.3 Remarks

This function gets the value of the specified 32-bit unsigned integer parameter for the specified video effect filter and writes the value that was retrieved to the location that was specified by the `val` parameter.

4.4.11 NvVFX_Load

```
NvCV_Status NvVFX_Load(
  NvVFX_Handle obj
);
```

4.4.11.1 Parameters

obj [in]

Type: `NvVFX_Handle`

The handle to the video effect filter instance to load.

4.4.11.2 Return Value

`NVFVX_SUCCESS` on success

4.4.11.3 Remarks

This function loads the specified video effect filter and validates the parameters that are set for the filter.

4.4.12 NvVFX_Run

```
NvCV_Status NvVFX_Run(
  NvVFX_Handle obj,
  int async
);
```

4.4.12.1 Parameters

obj [in]

Type: `NvVFX_Handle`

The handle to the video effect filter instance that will be run.

async [in]

An integer value that specifies whether the filter will run asynchronously or synchronously.

Here are the values:

- `1`: The filter runs asynchronously.
- `0`: The filter runs synchronously.

4.4.12.2 Return Value

`NVFVX_SUCCESS` on success

4.4.12.3 Remarks

This function runs the specified video effect filter by reading the contents of the input GPU buffer, applying the video effect filter, and writing the output to the output GPU buffer.

4.4.13 NvVFX_SetCudaStream

```
NvCV_Status NvVFX_SetCudaStream(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  CUstream stream
);
```

4.4.13.1 Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance for which you want to set the CUDA stream.

paramName

Type: `NvVFX_ParameterSelector`

The `NVVFX_CUDA_STREAM` selector string. Any other selector string returns an error.

stream

Type: `CUstream`

The CUDA stream to which the parameter will be set.

4.4.13.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_SELECTOR` when the `obj` parameter specifies a video effect filter instance that cannot parse the selector string that was specified by the `paramName` parameter or the data type of the parameter that was specified by the selector string.
- `NVCV_ERR_PARAMETER` when an unexpected `NULL` pointer was supplied.

4.4.13.3 Remarks

This function sets the CUDA stream in which the specified video effect filter will run to the parameter `stream`.

4.4.14 NvVFX_SetF32

```
NvCV_Status NvVFX_SetF32(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  float val
);
```

4.4.14.1 Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance for which you want to set the specified 32-bit floating-point parameter.

paramName

Type: `NvVFX_ParameterSelector`

The `NVVFX_SCALE` selector string. Any other selector string returns an error.

val

Type: `float`

The floating-point number to which the parameter will be set.

4.4.14.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_SELECTOR` when the `obj` parameter specifies a video effect filter instance that cannot parse the selector string that was specified by the `paramName` parameter or the data type of the parameter that was specified by the selector string.
- `NVCV_ERR_PARAMETER` when an unexpected `NULL` pointer was supplied.

4.4.14.3 Remarks

This function sets the specified single-precision (32-bit) floating-point parameter for the specified video effect filter to the `val` parameter.

4.4.15 NvVFX_SetImage

```
NvCV_Status NvVFX_SetImage(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  NvCVImage *im
);
```

4.4.15.1 Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance for which you want to set the specified image buffer.

paramName

Type: `NvVFX_ParameterSelector`

One of the following selector strings for the image buffer that you want to set:

- `NVVFX_INPUT_IMAGE_0`

- `NVVFX_INPUT_IMAGE`
- `NVVFX_OUTPUT_IMAGE_0`
- `NVVFX_OUTPUT_IMAGE`

Any other selector string returns an error.

im

Type: `NvCVImage *`

Pointer to the `NvCVImage` object to which the parameter will be set.

4.4.15.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_SELECTOR` when the `obj` parameter specifies a video effect filter instance that cannot parse the selector string that was specified by the `paramName` parameter or the data type of the parameter that was specified by the selector string.
- `NVCV_ERR_PARAMETER` when an unexpected `NULL` pointer was supplied.

4.4.15.3 Remarks

This function sets the specified input or output image buffer for the specified video effect filter to the parameter `im`.

4.4.16 NvVFX_SetString

```
NvCV_Status NvVFX_SetString(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  const char *str
);
```

4.4.16.1 Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance for which you want to set the specified character string parameter.

paramName

Type: `NvVFX_ParameterSelector`

The `NVVFX_MODEL_DIRECTORY` selector string. Any other selector string returns an error.

str

Type: `const char *`

Pointer to the character string to which you want to set the parameter.

4.4.16.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_SELECTOR` when the `obj` parameter specifies a video effect filter instance that cannot parse the selector string that was specified by the `paramName` parameter or the data type of the parameter that was specified by the selector string.
- `NVCV_ERR_PARAMETER` when an unexpected `NULL` pointer was supplied.

4.4.16.3 Remarks

This function sets the value of the specified character string parameter for the specified video effect filter to the parameter `str`.

4.4.17 NvVFX_SetU32

```
NvCV_Status NvVFX_SetU32(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  unsigned int val
);
```

4.4.17.1 Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance for which you want to set the specified 32-bit unsigned integer parameter.

paramName

Type: `NvVFX_ParameterSelector`

The `NVVFX_MODE` selector string. Any other selector string returns an error.

val

Type: `unsigned int`

The 32-bit unsigned integer to which you want to set the parameter.

4.4.17.2 Return Value

- `NVFVX_SUCCESS` on success.

- `NVCV_ERR_SELECTOR` when the `obj` parameter specifies a video effect filter instance that cannot parse the selector string that was specified by the `paramName` parameter or the data type of the parameter that was specified by the selector string.
- `NVCV_ERR_PARAMETER` when an unexpected `NULL` pointer was supplied.

4.4.17.3 Remarks

This function sets the value of the specified 32-bit unsigned integer parameter for the specified video effect filter to the `val` parameter.

4.5 Image Functions for C and C++

The image functions are defined in the `nvCVImage.h` header file. The image API is object oriented but is accessible to C **and** C++.

4.5.1 CVWrapperForNvCVImage

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

4.5.1.1 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 the `cv::Mat` constructor.

4.5.1.2 Return Value

Does not return a value.

4.5.1.3 Remarks

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

4.5.2 NvCVImage_Alloc

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

4.5.2.1 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. See “Pixel Organizations” on page 32 for more information.

memSpace [in]

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See “Memory Types” on page 35 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, irrespective of the value of `alignment`.

4.5.2.2 Return Value

- `NVFVX_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.

4.5.2.3 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.

4.5.3 NvCVImage_ComponentOffsets

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

4.5.3.1 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`).

4.5.3.2 Return Values

Does not return a value.

4.5.3.3 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.

4.5.4 NvCVImage_Composite

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

4.5.4.1 Parameters

fg [in]

Type: const `NvCVImage *`

The foreground source, which is a BGRu8 or an RGBu8 image.

bg [in]

Type: const `NvCVImage *`

The background source, which is a BGRu8 or an RGBu8 image.

mat [in]

Type: `const NvCVImage *`

The matte Yu8 or Au8 image, which indicates where the source image should come through.

dst [out]

Type: `NvCVImage *`

The destination BGRu8 or RGBu8 image, which can be the same as the `fg` foreground or `bg` background image, or a totally unrelated image.

4.5.4.2 Return Value

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

4.5.4.3 Remarks

This function uses the specified matte image to composite a foreground BGRu8 or RGBu8 image over a background image. The `fg`, `bg`, and `dst` images must be of the same type.

4.5.5 NvCVImage_CompositeOverConstant

```
NvCV_Status NvCVImage_CompositeOverConstant(
  const NvCVImage *src,
  const NvCVImage *mat,
  const unsigned char bgColor[3],
  NvCVImage *dst
);
```

4.5.5.1 Parameters

src [in]

Type: `const NvCVImage *`

The source BGRu8 or RGBu8 image.

mat [in]

Type: `const NvCVImage *`

The matte Yu8 or Au8 image, which indicates where the source image should come through.

[in] bgColor

Type: `const unsigned char`

A three-element array of characters that defines the color field over which the source image will be composited. This color field must have the same component ordering as the source and destination images.

dst [out]

Type: `NvCVImage *`

The destination BGRu8 or RGBu8 image. The destination image might be the same image as the source image.

4.5.5.2 Return Value

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

4.5.5.3 Remarks

This function uses the specified matte image to composite a BGRu8 or RGNU8 image over a constant color field.

4.5.6 NvCVImage_Create

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

4.5.6.1 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. See "Pixel Organizations" on page 32 for more information.

memSpace [in]

Type: `unsigned`

The type of memory in which the image data buffers will be stored. See "Memory Types" on page 35 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, irrespective 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.

4.5.6.2 Return Value

- `NVFVX_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.

4.5.6.3 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++).

4.5.7 NvCVImage_Dealloc

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

4.5.7.1 Parameters

im [in,out]

Type: `NvCVImage *`

Pointer to the image whose image buffer will be freed.

4.5.7.2 Return Value

Does not return a value.

4.5.7.3 Remarks

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

4.5.8 NvCVImage_Destroy

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

4.5.8.1 Parameters

im

Type: `NvCVImage *`

Pointer to the image that will be destroyed.

4.5.8.2 Return Value

Does not return a value.

4.5.8.3 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++).

4.5.9 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
);
```

4.5.9.1 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. See "Pixel Organizations" on page 32 for more information.

memSpace [in]

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See "Memory Types" on page 35 for more information.

4.5.9.2 Return Value

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

4.5.9.3 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.

4.5.10 NvCVImage_InitView

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

4.5.10.1 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.

4.5.10.2 Return Value

Does not return a value.

4.5.10.3 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.

4.5.11 NvCVImage_Realloc

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

4.5.11.1 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. See "Pixel Organizations" on page 32 for more information.

memSpace [in]

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See "Memory Types" on page 35 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`.

4.5.11.2 Return Value

- `NVFVX_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.

4.5.11.3 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 the frees the memory for the existing buffer and allocates the memory for a new buffer.

4.5.12 NvCVImage_Transfer

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

4.5.12.1 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 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 the component type of all images is the same (all integer or all floating-point).

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.

4.5.12.2 Return Value

- `NVFVX_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.

4.5.12.3 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.

Table 4-1 provides details about the supported conversions between pixel formats.

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

Table 4-1. Pixel Conversions

	u8→u8	u8→f32	f32→u8	f32→f32
Y→Y	X		X	
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	X		
RGB→RGB	X	X	X	X
RGB→RGBA	X	X	X	X
RGBA→Y	X	X		
RBBA→A		X		
RGBA→RGB	X	X	X	X
RGBA→RGBA	X			
YUV420→RGB	X			
YUV422→RGB	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.
- Other than pitch, if no conversion is necessary, all pixel format transfers are implemented, with `cudaMemcpy2DAsync()`.
- YUV420 and YUV422 sources have several variants.

  See "YUV Color Spaces" on page 34 for more information.
- CPU→CPU transfers are synchronous.

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.

4.5.13 NVWrapperForCVMat

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

4.5.13.1 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.

4.5.13.2 Return Value

Does not return a value.

4.5.13.3 Remarks

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

4.6 Image Functions for C++ Only

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

4.6.1 NvCVImage Constructors

4.6.1.1 Default Constructor

```
NvCVImage();
```

The default constructor creates an empty image with no buffer.

4.6.1.2 Allocation Constructor

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

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

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. See “Pixel Organizations” on page 32 for more information.

memSpace [in]

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See “Memory Types” on page 35 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`.

4.6.1.3 Subimage Constructor

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

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

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.

4.6.2 NvCVImage Destructor

```
~NvCVImage();
```

4.6.3 copyFrom

This version copies an entire image to another image. This version 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
);
```

4.6.3.1 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.

4.6.3.2 Return Value

- `NVFVX_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.

4.6.3.3 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 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.

4.7 Return Codes

The `NvCV_Status` enumeration defines the following values that the NVIDIA Video Effects functions might return to indicate error or success.

NVCV_SUCCESS = 0

Successful execution.

NVCV_ERR_GENERAL

Generic error code, which indicates that the function failed to execute for an unspecified reason.

NVCV_ERR_UNIMPLEMENTED

The requested feature is not implemented.

NVCV_ERR_MEMORY

The requested operation requires more memory than is available.

NVCV_ERR_EFFECT

An invalid effect handle has been supplied.

NVCV_ERR_SELECTOR

The specified selector is not valid for this effect filter.

NVCV_ERR_BUFFER

No image buffer has been specified.

NVCV_ERR_PARAMETER

An invalid parameter value has been supplied for this combination of effect and selector string.

NVCV_ERR_MISMATCH

Some parameters, for example, image formats or image dimensions, are not correctly matched.

NVCV_ERR_PIXELFORMAT

The specified pixel format is not supported.

NVCV_ERR_MODEL

An error occurred while the TRT model was being loaded.

NVCV_ERR_LIBRARY

An error while the dynamic library was being loaded.

NVCV_ERR_INITIALIZATION

The effect has not been properly initialized.

NVCV_ERR_FILE

The specified file could not be found.

NVCV_ERR_FEATURENOTFOUND

The requested feature was not found.

NVCV_ERR_MISSINGINPUT

A required parameter was not set.

NVCV_ERR_RESOLUTION

The specified image resolution is not supported.

NVCV_ERR_UNSUPPORTEDGPU

The GPU is not supported.

NVCV_ERR_WRONGGPU

The current GPU is not the one selected.

NVCV_ERR_UNSUPPORTEDDRIVER

The currently installed graphics driver is not supported.

NVCV_ERR_CUDA_MEMORY

The requested operation requires more CUDA memory than is available.

NVCV_ERR_CUDA_VALUE

A CUDA parameter is not within its acceptable range.

NVCV_ERR_CUDA_PITCH

A CUDA pitch is not within its acceptable range.

NVCV_ERR_CUDA_INIT

The CUDA driver and runtime could not be initialized.

NVCV_ERR_CUDA_LAUNCH

The CUDA kernel failed to launch.

NVCV_ERR_CUDA_KERNEL

No suitable kernel image is available for the device.

NVCV_ERR_CUDA_DRIVER

The installed NVIDIA CUDA driver is older than the CUDA runtime library.

NVCV_ERR_CUDA_UNSUPPORTED

The CUDA operation is not supported on the current system or device.

NVCV_ERR_CUDA_ILLEGAL_ADDRESS

CUDA attempted to load or store on an invalid memory address.

NVCV_ERR_CUDA

An unspecified CUDA error has occurred.

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