NVDIA Video Effects SDK

Programming Guide (Linux)

PG-09657-001_v0.6 Beta | April 2021

Table of Contents

Chapter 1. Introduction to NVIDIA Video Effects SDK 1

Chapter 2. Getting Started with NVIDIA Video Effects SDK for Linux 2

2.1 Hardware and Software Requirements 2

2.1.1 Hardware Requirements 2

2.1.2 Software Requirements 3

2.2 Installing the NVIDIA Video Effects SDK 3

2.3 NVIDIA Video Effects SDK Sample Applications 4

2.3.1 Building the Sample Applications 4

2.3.2 The AI Green Screen Application 4

2.3.2.1 Running the AI Green Screen Application 4

2.3.2.2 Running the AI Green Screen Application from the Shell Script 5

2.3.2.3 AI Green Screen Application Command-Line Reference 5

2.3.2.4 Keyboard Controls 7

2.3.3 The VideoEffects Application 7

2.3.3.1 Running the VideoEffects Application from the Shell Script 7

2.3.3.2 VideoEffects Application Command-Line Reference 8

2.3.4 The Upscale Pipleline Application 9

2.3.4.1 Running the UpscalePipeline Application from the Shell Script 9

2.3.4.2 UpscalePipeline Application Command-Line Reference 9

2.3.5 The DenoiseEffect Application 10

2.3.5.1 Running the DenoiseEffect Application from the Application Binary File 10

2.3.5.2 DenoiseEffect Application Command-Line Reference 11

2.3.6 The Batch Effect Application 12

2.3.6.1 Running the Batch Effect Application 12

2.3.6.2 Running the Batch Effect Application from the Shell Script 12

2.3.6.3 BatchEffectApp Command-Line Reference 12

2.3.7 The Batch Denoise Effect Application 13

2.3.7.1 Running the Batch Denoise Effect Application 13

2.3.7.2 Running the Batch Effect Application from the Shell Script 13

2.3.7.3 BatchDenoiseEffectApp Command-Line Reference 14

Chapter 3. Using NVIDIA Video Effects SDK in Applications 15

3.1 About the AI Green Screen Filter 15

3.2 About the Background Blur Filter 16

3.3 About the Artifact Reduction, Super Resolution, and Upscale Filters 17

3.4 About the Webcam Denoise Filter 19

3.5 Using a Video Effect Filter 20
3.5.1 Creating a Video Effect Filter 20
3.5.2 Setting the Path to the Model Folder 20
3.5.3 Creating and Setting State Variables (Only for Webcam Denoising) 21
3.5.4 Setting Up the CUDA Stream 22
3.5.5 Setting the Input and Output Image Buffers 22
3.5.6 Setting and Getting Other Parameters of a Video Effect Filter 24
3.5.6.1 Summary of NVIDIA Video Effects SDK Accessor Functions 24
3.5.6.2 Example: Setting the Filter Mode for AI Green Screen 25
3.5.6.3 Getting Information About a Filter and its Parameters 25
3.5.6.4 Getting a List of All Available Effects 26
3.5.7 Loading a Video Effect Filter 26
3.5.8 Running a Video Effect Filter 26
3.5.9 Destroying a Video Effect Filter 27
3.6 Working with Image Frames on GPU or CPU Buffers 27
3.6.1 Converting Image Representations to NvCVImage Objects 27
3.6.1.1 Converting OpenCV Images to NvCVImage Objects 27
3.6.1.2 Converting Other Image Representations to NvCVImage Objects 28
3.6.1.3 Converting Decoded Frames from the NvDecoder to NvCVImage Objects 28
3.6.1.4 Converting an NvCVImage Object to a Buffer that can be Encoded by NvEncoder 29
3.6.2 Allocating an NvCVImage Object Buffer 30
3.6.2.1 Using the NvCVImage Allocation Constructor to Allocate a Buffer 30
3.6.2.2 Using Image Functions to Allocate a Buffer 32
3.6.3 Transferring Images Between CPU and GPU Buffers 32
3.6.3.1 Transferring Input Images from a CPU Buffer to a GPU Buffer 32
3.6.3.2 Transferring Output Images from a GPU Buffer to a CPU Buffer 33
3.7 Using Multiple GPUs 33
3.7.1 Default Behavior in Multi-GPU Environments 34
3.7.2 Default Behavior in Multi-GPU Environments 35
3.7.3 Selecting the GPU for Video Effects Processing in a Multi-GPU Environment 35
3.7.4 Selecting Different GPUs for Different Tasks 35
3.8 Batch Processing 37
3.8.1 What is a Batch? 37
3.8.2 Batch Utilities 38
3.8.3 Allocating Batched Buffers 38
3.8.4 Selecting a Batch Model 38
3.8.5 Setting the Batched Images 40
3.8.6 Setting the Batch Size for NvVFX_Run() 41

3.8.7 Batching for Webcam Denoising 41
3.8.8 Example Code 42
Chapter 4. NVIDIA Video Effects SDK API Reference 43
4.1 Structures 43
4.1.1 NvVFX_Handle 43
4.1.2 NvVFX_Object 43
4.1.3 NvCVImage 43
4.1.3.1 Members 44
4.1.3.2 Remarks 46
4.2 Enumerations 46
4.2.1 NvCVImage_ComponentType 46
4.2.2 NvCVImage_PixelFormat 47
4.3 Type Definitions 48
4.3.1 NvVFX_EffectSelector 48
4.3.2 NvVFX_ParameterSelector 49
4.3.3 Pixel Organizations 50
4.3.4 YUV Color Spaces 53
4.3.5 Memory Types 54
4.4 Video Effects Functions 54
4.4.1 NvVFX_CreateEffect 54
4.4.1.1 Parameters 54
4.4.1.2 Return Value 54
4.4.1.3 Remarks 55
4.4.2 NvVFX_CudaStreamCreate 55
4.4.2.1 Parameters 55
4.4.2.2 Return Value 55
4.4.2.3 Remarks 55
4.4.3 NvVFX_CudaStreamDestroy 55
4.4.3.1 Parameters 55
4.4.3.2 Return Value 56
4.4.3.3 Remarks 56
4.4.4 NvVFX_DestroyEffect 56
4.4.4.1 Parameters 56
4.4.4.2 Return Value 56
4.4.4.3 Remarks 56
4.4.5 NvVFX_GetCudaStream 56
4.4.5.1 Parameters 57
4.4.5.2 Return Value 57
4.4.5.3 Remarks 57

4.4.6 NvCV_GetErrorStringFromCode....57
4.4.6.1 Parameters....57
4.4.6.2 Return Value....57
4.4.6.3 Remarks....58
4.4.7 NvVFX_GetF32....58
4.4.7.1 Parameters....58
4.4.7.2 Return Value....58
4.4.7.3 Remarks....58
4.4.8 NvVFX_GetImage....59
4.4.8.1 Parameters....59
4.4.8.2 Return Value....59
4.4.8.3 Remarks....60
4.4.9 NvVFX_GetString....60
4.4.9.1 Parameters....60
4.4.9.2 Return Value....61
4.4.9.3 Remarks....61
4.4.10 NvVFX_GetU32....61
4.4.10.1 Parameters....61
4.4.10.2 Return Value....62
4.4.10.3 Remarks....62
4.4.11 NvVFX_Load....62
4.4.11.1 Parameters....62
4.4.11.2 Return Value....62
4.4.11.3 Remarks....62
4.4.12 NvVFX_Run....62
4.4.13 Parameters....63
4.4.13.1 Return Value....63
4.4.13.2 Remarks....63
4.4.14 NvVFX_SetCudaStream....63
4.4.14.1 Parameters....63
4.4.14.2 Return Value....64
4.4.14.3 Remarks....64
4.4.15 NvVFX_SetF32....64
4.4.15.1 Parameters....64
4.4.15.2 Return Value....65
4.4.15.3 Remarks....65
4.4.16 NvVFX_SetImage....65
4.4.16.1 Parameters....65
4.4.16.2 Return Value....66

4.4.16.3 Remarks 66
4.4.17 NvVFX_SetString 66
4.4.17.1 Parameters 66
4.4.17.2 Return Value 67
4.4.17.3 Remarks 67
4.4.18 NvVFX_SetU32 67
4.4.18.1 Parameters 67
4.4.18.2 Return Value 68
4.4.18.3 Remarks 68
4.5 Image Functions for C and C++ 68
4.5.1 CVWrapperForNvCVImage 68
4.5.1.1 Parameters 68
4.5.1.2 Return Value 68
4.5.1.3 Remarks 69
4.5.2 NvCVImage_Alloc 69
4.5.2.1 Parameters 69
4.5.2.2 Return Value 70
4.5.2.3 Remarks 70
4.5.3 NvCVImage_ComponentOffsets 71
4.5.3.1 Parameters 71
4.5.3.2 Return Values 71
4.5.3.3 Remarks 72
4.5.4 NvCVImage_Composite 72
4.5.4.1 Parameters 72
4.5.4.2 Return Value 73
4.5.4.3 Remarks 73
4.5.5 NvCVImage_CompositeRect 73
4.5.5.1 Parameters 73
4.5.5.2 Return Value 74
4.5.5.3 Remarks 75
4.5.6 NvCVImage_CompositeOverConstant 75
4.5.6.1 Parameters 75
4.5.6.2 Return Value 76
4.5.6.3 Remarks 76
4.5.7 NvCVImage_Create 76
4.5.7.1 Parameters 76
4.5.7.2 Return Value 77
4.5.7.3 Remarks 78
4.5.8 NvCVImage_Dealloc 78

4.5.8.1 Parameters 78
4.5.8.2 Return Value 78
4.5.8.3 Remarks 78
4.5.9 NvCVImage_Destroy 78
4.5.9.1 Parameters 78
4.5.9.2 Return Value 78
4.5.9.3 Remarks 79
4.5.10 NvCVImage_Init 79
4.5.10.1 Parameters 79
4.5.10.2 Return Value 80
4.5.10.3 Remarks 80
4.5.11 NvCVImage_InitView 81
4.5.11.1 Parameters 81
4.5.11.2 Return Value 81
4.5.11.3 Remarks 82
4.5.12 NvCVImage_Realloc 82
4.5.12.1 Parameters 82
4.5.12.2 Return Value 83
4.5.12.3 Remarks 84
4.5.13 NvCVImage_Transfer 84
4.5.13.1 Parameters 84
4.5.13.2 Return Value 85
4.5.13.3 Remarks 85
4.5.14 NvCVImage_TransferRect 87
4.5.14.1 Parameters 87
4.5.14.2 Return Value 89
4.5.14.3 Remarks 89
4.5.15 NvCVImage_TransferFromYUV 89
4.5.15.1 Parameters 89
4.5.15.2 Return Value 91
4.5.15.3 Remarks 92
4.5.16 NvCVImage_TransferToYUV 92
4.5.16.1 Parameters 92
4.5.16.2 Return Value 94
4.5.16.3 Remarks 94
4.5.17 NvCVImage_MapResource 94
4.5.17.1 Parameters 95
4.5.17.2 Return Value 95
4.5.17.3 Remarks 95

4.5.18 NvCVImage_UnmapResource 95
4.5.18.1 Parameters 95
4.5.18.2 Return Value 96
4.5.18.3 Remarks 96
4.5.19 NVWrapperForCVMat 96
4.5.19.1 Parameters 96
4.5.19.2 Return Value 96
4.5.19.3 Remarks 96
4.6 Image Functions for C++ Only 97
4.6.1 NvCVImage Constructors 97
4.6.1.1 Default Constructor 97
4.6.1.2 Allocation Constructor 97
4.6.1.3 Subimage Constructor 98
4.6.2 NvCVImage Destructor 99
4.6.3 copyFrom 99
4.6.3.1 Parameters 100
4.6.3.2 Return Value 100
4.6.3.3 Remarks 100
4.7 Return Codes 101

List of Tables

Table 2-1: Software Requirements....3
Table 2-2: Downloading the Prerequisites....3
Table 3-1. Scale and Resolution Support for Input Videos....18
Table 3-2: Video Effects Accessor Functions....24
Table 4-1. Pixel Conversions....86

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:

- AI green screen (video background segmentation) (Beta), which segments and masks the background areas in a video or image.
- Background Blur (Beta), which uses the segmentation mask from the AI green screen filter, or other sources, and produces a blur effect in the background, in a video, or in an image.
- Encoder Artifact Reduction (Beta), which reduces the blocking and noisy artifacts that are produced from encoding while preserving the details of the original video.

  The ArtifactReduction effect has the following modes:

  - Strength 0, which applies a weak effect.
  - Strength 1, which applies a strong effect.
- Super resolution (Beta), which upscales a video and reduces encoding artifacts.

  This filter enhances the details, sharpens the output, and preserves the content. The SuperRes effect has two modes:

  - Strength 1, which applies strong enhancements.
  - Strength 0, which applies weaker enhancements while reducing encoding artifacts.
- Upscale (Beta), which is a fast and light-weight method to upscale for an input video and sharpen the resulting output.

  This filter can optionally be pipelined with encoder artifact reduction to enhance the scale while reducing the video artifacts.

- **Webcam Denoising (Beta)**, which removes noise from a webcam video while preserving the texture details.

  This effect has the following modes:

  - Strength 0, which applies a weak effect.
  - Strength 1, which applies a strong effect.

Chapter 2. Getting Started with NVIDIA Video Effects SDK for Linux

2.1 Hardware and Software Requirements

NVIDIA Video Effects SDK requires specific NVIDIA GPUs. specific versions of the Linux OS, and other associated software on which the SDK depends.

This SDK is designed and optimized for server-side (datacenter/cloud) deployment. We do not officially support the testing, experimentation and production deployment of this SDK to client-side application integration and local deployment.

2.1.1 Hardware Requirements

NVIDIA Video Effects SDK is compatible with GPUs that are based on NVIDIA Turing™, NVIDIA Volta™, or the NVIDIA Ampere™ architecture.

Note: The SDK does not support Multi-Instance GPU (MIG). If this feature is enabled, you might experience issues.

Note: For best performance with NVIDIA T4 and other server GPUs, make sure that you use a server that meets the thermal and airflow requirements for these types of products. Refer to https://www.nvidia.com/en-us/data-center/tesla/tesla-qualified-servers-catalog/ for the latest list of qualified servers.

2.1.2 Software Requirements

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

Table 2-1: Software Requirements

Software	Required Version
Linux	Ubuntu 18.04 or CentOS 7
CUDA	11.1
TensorRT	7.2.2.3 (for NVIDIA CUDA® 11.1)
cudnn	8.0.5 (for the appropriate NVIDIA TensorRT™/CUDA versions)
CMake	3.10
opencv	3.2+ or 4.x (for sample apps only)
NVIDIA Graphics Driver for Linux	455.23+

2.2 Installing the NVIDIA Video Effects SDK

The SDK is delivered as a `.tar.gz` package, which is a compressed tar format.

The package contains the video effects library and header files, which need to be extracted to the `/usr/local/VideoFX` directory.

You can download the prerequisites from https://developer.nvidia.com:

Table 2-2: Downloading the Prerequisites

Prerequisite	Download location
CUDA	https://developer.nvidia.com/cuda
TensorRT	https://developer.nvidia.com/tensorrt
cudnn	https://developer.nvidia.com/cudnn

See "Software Requirements" on page 2 for more information about the version numbers for *<version>* below).

- To install CUDA (include graphics driver):

```
$ sudo sh cuda_<version>_linux.run
```

- To install TensorRT:

```
$ sudo tar -xvf TensorRT-<version>.Ubuntu-18.04.x86_64-gnu.cuda-
<version>.cudnn<version>.tar.gz -C /usr/local
```

- To install cudnn:
  ```
  $ sudo tar -xvf cudnn-<version>-linux-x64-v<version>.tgz -C /usr/local
  ```
- To install the SDK:
  ```
  $ sudo tar -xvf VideoFX-<version>.tar.gz -C /usr/local
  ```

To install CUDA 11.1 and preserve older (supported) graphics drivers, such as versions 418 and 440, you need to carefully install the CUDA toolkit and the CUDA compatibility package. Refer to https://docs.nvidia.com/deploy/cuda-compatibility/ and the README_quickstart.md file for more information.

Note: If you plan to use an older (supported) graphics driver, you must set `LD_LIBRARY_PATH` to include the compatibility package by running the following command:

```
export LD_LIBRARY_PATH=/usr/local/cuda/compat:$LD_LIBRARY_PATH
```

2.3 NVIDIA Video Effects SDK Sample Applications

To demonstrate the features of the NVIDIA Video Effects SDK, the SDK provides sample applications as source code that you can build and as binary files that you can run without building. You can run each application from the supplied shell script or from the application binary file.

2.3.1 Building the Sample Applications

To build the sample applications, run the following command:

```
/usr/local/VideoFX-<version>/share/build_samples.sh
```

Follow the prompts to provide an install location and build the sample apps. The script might prompt you to install the necessary prerequisites.

2.3.2 The AI Green Screen Application

2.3.2.1 Running the AI Green Screen Application

The AI green screen application, `AigsEffectApp`, demonstrates the AI green screen (video background segmentation) feature of the SDK. The application accepts an image a video file, or the output from a webcam as input and produces video output, which can be stored in a file or displayed in a window.

The application produces the output initially with foreground pixels highlighted. When the `--show` argument is specified, the keys described in “Keyboard Controls” on page 7 toggle the behavior of the application.

You can run this application from the supplied shell script (`run_aigs_webcam.sh` or `run_aigs_image.sh`) or from the application binary file.

When the appropriate `--comp_mode` value is passed via command line, the AigsEffectApp sample app also demonstrates the background blur effect,

2.3.2.2 Running the AI Green Screen Application from the Shell Script

The `~/samples` folder contains the following shell scripts:

- `run_aigs_webcam.sh`
- `run_aigs_image.sh`

To run the AI green screen sample application on a sample image that is included in the SDK, run the following command:

```
$ cd ~/mysamples
$ ./run_aigs_image.sh
```

This step creates an output image called `aigs_image_out.jpg`.

To run the AI green screen sample application by using a connected webcam that displays on the screen, run the following command:

```
$ cd ~/mysamples
$ ./run_aigs_webcam.sh
```

Note: To run the application directly from the command line, review the contents of the shell script.

2.3.2.3 AI Green Screen Application Command-Line Reference

`AigsEffectApp` [*arguments*...]

Here are the arguments:

--in_file=*path*

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

--webcam[=(true|false)]

If `true`, use a webcam as input instead of a file.

--cam_res=*[widthx]height*

If --webcam is `true`, specify the resolution of the webcam. *width* is optional. If omitted, *width* is computed from height to give an aspect ratio of 16:9. For example:

`--cam_res=1280x720` or `--cam_res=720`

If --webcam is `false`, this argument is ignored.

--out_file=*path*

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

--show[=(true|false)]

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

--model_dir=*path*

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

--codec=*fourcc*

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

--help

Display help information for the command.

--mode=(0|1)

Selects the mode in which to run the application:

0 selects best quality.

1 selects fastest performance.

--comp_mode=(0|1|2|3|4|5|6)

Where *mode* selects which composition mode to use:

- 0: Displays the segmentation mask (`compMatte`).
- 1: Overlays the mask on top of the image (`compLight`).
- 2: Provides a composition with a BGR={0,255,0} background image (`compGreen`).
- 3: Provides a composition with a BGR={255,255,255} background image (`compWhite`).
- 4: No composition but displays the input image (`compNone`).
- 5: Overlays the mask on the image (`compBG`).
- 6: Applies a background blur filter on the input image by using the segmentation mask (`compBlur`).

2.3.2.4 Keyboard Controls

The sample application provides keyboard controls for changing the run-time behavior of the application.

- `C`: Toggles between the following ways of rendering the image:
  - Displaying a semitransparent white overlay on the subject
  - Replacing the background with a solid green color
  - Displaying the resultant alpha matte
- `F`: Toggles the frame rate display on and off.
- `Q` or `Escape`: exits the app, and cleanly finishes writing any output file.

2.3.3 The VideoEffects Application

2.3.3.1 Running the VideoEffects Application from the Shell Script

The `~/mysamples` folder contains the `run_videoeffects.sh` shell script.

To run the VideoEffects sample application on the sample images in the SDK, run the following command:

```
./run_videoeffects.sh
```

Note: To run the application directly from the command line, review the contents of the shell script.

2.3.3.2 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.

--effect=ArtifactReduction, SuperRes, or Upscale

This argument selects the effect that will be applied:

- *`ArtifactReduction`*: removes the encoder artifact without changing the resolution.
- *`SuperRes`*: removes artifact (strength 0 mode) and upscales to the specified output resolution.
- *`Upscale`*: Fast upscaler that increases the video resolution to the specified output resolution.

Note: You can also select any of the effects that are listed when you run the VideoEffectsApp with the `--help` flag.

--resolution=*`NNN`*

The desired output vertical resolution from Upscale and SuperRes, scaled 1.3333x, 1.5x, 2x, 3x or 4x times the input.

--out_file=*path*

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

--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 to 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 the strength of the filter to be applied.

- `0` selects a weak effect.
- `1` selects a strong effect.
- Continuously variable values [0.0-1.0] in Upscale to adjust *sharpness*.

--verbose [=(true|false)]

Verbose output.

--debug

Print extra debugging.

--help

Display help information for the command.

2.3.4 The Upscale Pipleline Application

2.3.4.1 Running the UpscalePipeline Application from the Shell Script

To run the UpscalePipeline sample application on the sample images in the SDK, run the following command:

```
$ cd ~/mysamples
$ ./run_upscalepipeline.sh
```

Note: To run the application directly from the command line, review the contents of the shell script.

2.3.4.2 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 is to be stored.

--resolution=*NNN*

The output image/video vertical resolution, which is scaled from the input vertical resolution by 1.3333, 1.5, 2, 3, or 4.

--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 is H264.

--ar_strength=[0|1] for ArtifactReduction

Selects the strength of the filter to be applied.

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

--sr_strength=[0.0-1.0] for Upscale

Selects the strength of the upscale filter to be applied.

- 0 selects no enhancement.
- 1 selects the maximum crispness.
- The default value is 0.4.

--progress

Show the progress.

--verbose [=(true|false)]

Verbose output.

--debug [=(true|false)]

Prints extra debugging information.

--help

Displays help information for the command.

2.3.5 The DenoiseEffect Application

2.3.5.1 Running the DenoiseEffect Application from the Application Binary File

The ~/mysamples folder contains the run_denoiseeffect.sh shell script.

To run the DenoiseEffect sample application on the sample images in the SDK, run the following command:

```
./run_denoiseeffect.sh
```

The input is currently set to a webcam stream and the output to a video file, but you can set them as images or as videos. When the sample app is running, you can toggle the effect on and off by pressing the E key.

Note: To run the application directly from the command line, review the contents of the shell script.

2.3.5.2 DenoiseEffect Application Command-Line Reference

```
DenoiseEffectApp [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 image or the video output is to be stored.

--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`)

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

--progress

Shows the progress.

--webcam

Uses the webcam as input.

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

Verbose output

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

Prints extra debugging.

--help

Displays help information for the command.

2.3.6 The Batch Effect Application

2.3.6.1 Running the Batch Effect Application

Some of the effects can take advantage of batching to achieve higher performance. The `BatchEffectApp` contains code that illustrates the extra steps that are needed to simultaneously process a batch of images. Unlike the other sample applications, this application accepts multiple images to be used as input.

2.3.6.2 Running the Batch Effect Application from the Shell Script

Note: The `~/mysamples` folder contains the `run_batcheffect.sh` shell script.

To run the `BatchEffectApp` on a batch of sample images, run:

```
$ cd ~/mysamples
$ ./run_batcheffect.sh
```

Note: To run the application directly from the command line, review the content of the shell script.

2.3.6.3 BatchEffectApp Command-Line Reference

```
BatchEffectApp [flags ...] inFile1 [ inFileN ...]
```

Here are the flags:

`--out_file=<path>`
Output video files to be written, which is a pattern with a `%u` or `%d` that defaults to `"BatchOut_%02u.mp4"`.

`--effect=<effect>`
One of the following effects will be applied:

- Transfer
- ArtifactReduction
- SuperRes
- GreenScreen
- Upscale

`--strength=<value>`
The strength of an effect:
- 0 or 1 for super res and artifact reduction.
- [0.0, 1.0] range for upscaling.

`--scale=<scale>`
The scale factor that will be applied, which are 1.3333333, 1.5, 2, 3, or 4.

`--mode=<mode>`
The values for mode are 0 or 1.

`--model_dir=<path>`
The path to the directory that contains the models.

`--verbose`
Verbose output.

`--help`
A help message.

2.3.7 The Batch Denoise Effect Application

2.3.7.1 Running the Batch Denoise Effect Application

The Batch Denoise Effect Application can be used to denoise frames from multiple video stream in user-specified batches. `BatchDenoiseEffectApp` contains code that illustrates the extra steps that are needed to simultaneously process multiple video stream. The app assumes that the input videos have the same resolution and length. Unlike the other sample applications, this application accepts multiple videos to be used as input.

2.3.7.2 Running the Batch Effect Application from the Shell Script

Note: The `~/mysamples` folder contains the `run_batchdenoiseeffect.sh` shell script.

To run the `BatchDenoiseEffectApp` on a batch of sample images, run:

```
$ cd ~/mysamples
$ ./run_batcheffect.sh inFile1 [ ... inFileN ]
```

This process denoises the videos in the N input files, inFile1 to inFileN and writes the output to the N corresponding output files.

Note: Currently sample input videos for batch denoising are not supplied with the SDK.

2.3.7.3 BatchDenoiseEffectApp Command-Line Reference

```
BatchDenoiseEffectApp [flags ...] inFile1 [ inFileN ...]
```

Here are the flags:

`--out_file=<path>`
Output video files to be written, which is a pattern with a `%u` or `%d` that defaults to `"BatchOut_%02u.mp4"`.

`--strength=<value>`
The values for Strength are 0 or 1.

`--batchsize=<value>`
The size of the batch. Default: 8

`--model_dir=<path>`
The path to the directory that contains the models.

`--verbose`
Verbose output.

`--help`
A help message.

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 AI Green Screen Filter

The AI green screen filter segments a video or still image into foreground and background regions. The following modes of operation are supported:

- Quality mode, which gives the highest quality result.
  - Images must be at least 288 pixels high.
  - This mode is the default.
- Performance mode, which gives the fastest performance.
  - Some degradation in quality may be observed.
  - Images must be at least 288 pixels high.

  For best results, the aspect ratio of the image file must be 16:9. The filter letterboxes images with other aspect ratios.

  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 grayscale GPU buffer.

The AI green screen filter processes an input image by performing the following sequence of operations:

1. The filter classifies the pixels in the video based on the filter's confidence:
   - Pixels that are part of a human are classified as foreground pixels.
   - All other pixels are classified as background pixels.
2. After classifying the pixels, the filter generates an eight-bit alpha mask with the same resolution as the input image.

   The alpha values are determined from the filter's confidence that the pixel belongs to the class it was assigned to.
   - Pixels with a high foreground confidence are assigned alpha values close to 255.
   - Pixels with a high background confidence are assigned alpha values close to zero.

   Downstream processes can combine the alpha mask generated by the AI green screen filter with the original image to generate effects. For example, a 24-bit BGR input image can be combined with the mask to create a 32-bit image with an alpha channel.

   The alpha values can be determined in one of the following ways:
   - Thresholds can be applied during the process to force the alpha channel of background pixels to be 0 and of foreground pixels to be 255.
   - The original alpha values can be maintained to create semi-transparency based on confidence.

   The 32-bit image can then be composited onto a tertiary image to generate an image in which the background pixels of the original image are replaced with the background pixels of the tertiary image.

Note: The AI green screen effect achieves its best results on videos that are recorded by one person sitting in front of a camera. The feature will not perform well on full-body videos, multiple persons in the scene, or camera angles that deviate too much from a front facing camera.

3.2 About the Background Blur Filter

The Background Blur filter uses the segmentation mask and an input image to produce a blur effect in the background region of the input image.

The Strength value, which allows you to change the strength of the applied blur filter by selecting a value in the [0-1] range, and the default is 0.5. The Strength value can be set by using the `NVVFX_SetF32` function. The value uses the same input as AI green screen filter, the segmentation mask output of AI green screen filter or other sources, and an output buffer.

The Background Blur filter processes an input image by completing the following steps:

1. The filter uses the segmentation mask to the find the region of interest to apply blur filter.
2. The input is composited with the blurred image to produce the background blur effect.

Here are requirements to use the Background Blur filter:

- The input must be 24-bit BGR input image.

  The data type is UINT8,and the range of values is [0, 255].
- The segmentation mask must be 8-bit segmentation mask.

  The data type is UINT8, and the range of values is [0, 255].
- Output is a 24-bit output image, and the data type is UINT8.

3.3 About the Artifact Reduction, Super Resolution, and Upscale Filters

The AR/SR filter contains the following effects 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 video while preserving the details of the original video. The encoder artifact reduction has been optimized for the H.264 encoder.
  Here are the two modes of operation:
  - Strength 0 removes lesser artifacts, preserves low gradient information better, and is suited for higher bitrate videos.
  - Strength 1 is better suited for lower bitrate videos.
- Super resolution

  This filter enhances the resolution of low-resolution videos and enhances the details and sharpens the output while preserving the content. Strength 0 mode is suitable for upscaling lossy content that has encoding artifacts, and Strength 1 mode is suitable for lossless videos.
  - Strength 0 enhances less and removes more encoding artifacts and is suited for lower quality videos.
  - Strength 1 enhances more and is suited for higher quality lossless videos.

Table 3-1 illustrates the scale and resolution support for input videos to be used with the `ArtifactReduction` and `SuperRes` effects.

Table 3-1. Scale and Resolution Support for Input Videos

Effect	Input resolution range	Output Resolution Range	Comment
Artifact reduction	[90p, 1080p]	[90p, 1080p]	Input and output range is the same as AR does not change resolution
Super resolution 1.33x	[90p, 1080p]	[120p, 1440p]	
Super resolution 1.5x	[90p, 1080p]	[135p, 1620p]	
Super resolution 2x	[90p, 1080p]	[180p, 4k]	
Super resolution 3x	[90p, 720p]	[270p, 4k]	Exception: lower end GPU may not have enough memory to support up to 720p input.
Super resolution 4x	[90p, 540p]	[360p, 4k]	Exception: lower end GPU may not have enough memory to support up to 540p input.

- Upscale

  This is a very fast and light-weight method for upscaling an input video. It also provides a sharpening parameter to sharpen the resulting output. This feature can be optionally pipelined with the encoder Artifact reduction feature to enhance the scale while reducing the video artifacts. Upscale supports any input resolution and can be upscaled 4/3x, 1.5x, 2x, 3x, or 4x. The output resolution values must be integers, and the ratio of widths must exactly equal the ratio of heights.

  The Upscale filter provides a floating-point strength value which ranges between 0.0 and 1.0. This signifies an enhancement parameter.

  - Strength 0 implies no enhancement, only upscaling.
  - Strength 1 implies the maximum enhancement.
  - The default value is `0.4`.

Here are some general recommendations:

- If a video without encoding artifacts needs fast resolution increase, use `Upscale`.
- If a video has no encoding artifacts, to increase the resolution, use `SuperRes` with strength 1 for greater enhancement.
- If a video has fewer encoding artifacts, to remove artifacts, use `ArtifactReduction` only with strength 0.
- If a video has more encoding artifacts, to remove artifacts, use `ArtifactReduction` only with strength 1.
- To increase the resolution of a video with encoding artifacts
  - For light artifacts, use `SuperRes` with strength 0.
  - Otherwise, use `ArtifactReduction` followed by `SuperRes` with strength 1.

The following sample apps are provided for these filters:

- `VideoEffectsApp`: This app runs each of the `ArtifactReduction`, `SuperRes`, `Upscale` effects individually and should be used when you want to apply a filter to the input video.
- `UpscalePipelineApp`: This app runs a pipeline of `ArtifactReduction` followed by `Upscale`.

  You can use this app when you want a fast application that performs Artifact reduction and scale enhancement.

  Both apps support input videos within a resolution range as specified in Table 3-1.

3.4 About the Webcam Denoise Filter

The webcam denoise filter removes the noise from the webcam video while preserving details. The Strength value allows you to change the strength of the applied denoise filter by selecting a value of 0 or 1, and the default is 0.

Here is some additional information about the values:

- The Strength of value 0 corresponds to a weak effect, which places a higher emphasis on texture preservation.
- The Strength of value 1 corresponds to a strong effect, which places a higher emphasis on noise removal.

You can set the Strength value by using the `NVVFX_SetF32` function.

The webcam denoising filter can be applied on videos and images, but for better denoising results, we recommend that you apply this filter to videos.

This filter supports input images/videos in the 80p-1080p resolution range.

3.5 Using a Video Effect Filter

To use a video effect filter, you need to create the filter; set up various properties of the filter; and load, run, and destroy the filter.

3.5.1 Creating a Video Effect Filter

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

- The `NvVFX_EffectSelector` type See NvVFX_EffectSelector for more information.
- 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 that can be used in additional API calls.

This example creates an AI green screen video effect filter.

```
NvCV_Status vfxErr = NvVFX_CreateEffect(NVVFX_FX_GREEN_SCREEN,
&effectHandle);
```

3.5.2 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 20.
- 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 `/usr/local/VideoFX/bin/models`.

```
const char* modelDir="/usr/local/VideoFX/bin/models";
...
vfxErr = NvVFX_SetString(effectHandle, NVVFX_MODEL_DIRECTORY, modelDir);
```

3.5.3 Creating and Setting State Variables (Only for Webcam Denoising)

Webcam denoising uses a state variable to track the input video stream to remove temporal noise. The SDK user is responsible for completing the following tasks:

- Create the state variable.

  To create the state variable:

  a). Query the size of state variable by calling `NvVFX_GetU32()` with the `NVVFX_STATE_SIZE` selector string.

  ```
  unsigned int stateSizeInBytes;
  vfxErr =  NvVFX_GetU32(effectHandle, NVVFX_STATE_SIZE,
  &stateSizeInBytes);
  ```

  b). Allocate the necessary space for the state variable in the GPU by using `cudaMalloc()`.

  ```
  void* state[1];
  cudaMalloc(&state[0], stateSizeInBytes);
  ```

  c). Initialize the state variable to 0 by using `cudaMemset()`.

  ```
  cudaMemset(state[0], 0, stateSizeInByte);
  ```

- Pass the state variable to the SDK.

  To pass the state variable to the SDK, use `NvVFX_SetObject()` with the `NVVFX_STATE` selector string.

  ```
  vfxErr =  NvVFX_SetObject(effectHandle, NVVFX_STATE, (void*)state);
  ```

- Release the state variable memory.

  After the state variable has been initialized and set, the filter can be run on an image or a video. After the state variable has completed the processing of the original input, you can reuse the variable with another image/video.

  However, **before** you can use it on a new input, reset the state variable to 0 by using `cudaMemset()`. When the state variable is no longer in use, release the memory that was allocated for the state variable by using `cudaFree()`.

  ```
  cudaFree(state[0]);
  ```

3.5.4 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 20.
   - The NVVFX_CUDA_STREAM selector string.
   - 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);
```

3.5.5 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 AI green screen filter requires 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 32.

Here is a list of the currently used formats:

- AI green screen: BGRu8 chunky → Au8
- Background Blur: BGRu8 chunky + Au8 chunky → BGRu8 chunky
- Upscale: RGBAu8 chunky → RGBAu8 chunky
- ArtifactReduction: BGRf32 planar normalized → BGRf32 planar normalized
- SuperRes: BGRf32 planar normalized → BGRf32 planar normalized
- Transfer: anything → anything
- Denoise: BGRf32 planar normalized → BGRf32 planar normalized

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

- The filter handle that was created as explained in "Creating a Video Effect Filter" on page 20.
- 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.
- For Background blur, use `NVVFX_INPUT_IMAGE_1` for passing the second input which is the segmentation mask.

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);
```

3.5.6 Setting and Getting Other Parameters of a Video Effect Filter

Before you load and run a video effect filter, set the 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 20.
- 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.

This example sets the mode for an AI green screen filter to fastest performance.

```
vfxErr = NvVFX_SetU32(effectHandle, NVVFX_MODE, 1);
```

3.5.6.1 Summary of NVIDIA Video Effects SDK Accessor Functions

Table 3-2: Video Effects Accessor Functions

Parameter Type	Data Type	Set and Get Accessor Function
32-bit unsigned integer	`unsigned int`	`NvVFX_SetU32()`
		`NvVFX_GetU32()`
32-bit signed integer	`int`	`NvVFX_SetS32()`
		`NvVFX_GetS32()`
Single-precision (32-bit) floating-point number	`float`	`NvVFX_SetF32()`
		`NvVFX_GetF32()`
Double-precision (64-bit) floating point number	`double`	`NvVFX_SetF64()`
		`NvVFX_GetF64()`
64-bit unsigned integer	`long long`	`NvVFX_SetU64()`
		`NvVFX_GetU64()`
Image buffer	`NvCVImage`	`NvVFX_SetImage()`
		`NvVFX_GetImage()`
	`void`	`NvVFX_SetObject()`

Parameter Type	Data Type	Set and Get Accessor Function
Object		`NvVFX_GetObject()`
Character string	`const char*`	`NvVFX_SetString()`
		`NvVFX_GetString()`
CUDA stream	`CUstream`	`NvVFX_SetCudaStream()`
		`NvVFX_GetCudaStream()`

3.5.6.2 Example: Setting the Filter Mode for AI Green Screen

The AI green screen filter supports the following modes of operation:

- Quality mode, which provides the highest quality result (default).
- Performance mode, which provides the fastest performance.

Call the `NvVFX_SetU32()` function, specifying the following information as parameters:

- The filter handle that was created as explained in "Creating a Video Effect Filter" on page 20.
- The selector string NVVFX_MODE
- An integer that denotes the mode of operation that you want:
  - 0: Quality mode
  - 1: Performance mode

This example sets the mode to Performance.

```
vfxErr = NvVFX_SetU32 (effectHandle, NVVFX_MODE, 1);
```

3.5.6.3 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
);
```

3.5.6.4 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);
```

3.5.7 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: To select the correct model, some video effect filters have settings that need to be set before you call `Load()`. For example, `GreenScreen` needs the mode, `SuperRes` and `Upscale` need the input and output images, and `ArtifactReduction` needs the input image and the strength. The images or some proxies are needed in `Load` to select the correct model.

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 20.

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

Note: If a set accessor function is used to change a filter parameter that is used to select a model, the filter must be reloaded so that the change can take effect before it is run. This includes the mode for `GreenScreen` and the strength for `ArtifactReduction`. The input and output images can be changed before any `Run()` call without subsequently calling `Load()`, as can the scale and the Upscale strength.

3.5.8 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 20.
- 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()
```

3.5.9 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, and specify the filter handle that was created as explained in "Creating a Video Effect Filter" on page 20.

```
NvVFX_DestroyEffect(effectHandle);
```

3.6 Working with Image Frames on GPU or CPU Buffers

Effect filters accept image buffers as `NvCVImage` objects. The image buffers can be 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.

3.6.1 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.

3.6.1.1 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);
```

3.6.1.2 Converting Other Image Representations 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);
```

3.6.1.3 Converting Decoded Frames from the NvDecoder to NvCVImage Objects

Call the `NvCVImage_Transfer()` function to convert the decoded frame that is provided by the `NvDecoder` from the decoded pixel format to the format that is required by a feature of the Video Effects SDK. The following sample shows a decoded frame that was converted from the NV12 to the BGRA pixel format.

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

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

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

//Allocate memory for BGRA frame, and set alpha opaque
NvCVImage_Alloc(&BGRA_frame, dec.GetWidth(), dec.GetHeight(), NVCV_BGRA,
NVCV_U8, NVCV_CHUNKY, NVCV_GPU, 1);
cudaMemset(BGRA_frame.pixels, -1, BGRA_frame.pitch * BGRA_frame.height);

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

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

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

Here is some additional information:

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

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

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

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

//Initialize encoder with a BGRA output pixel format
```

```
using NvEncCudaPtr = std::unique_ptr<NvEncoderCuda,
std::function<void(NvEncoderCuda*)>>;
NvEncCudaPtr pEnc(new NvEncoderCuda(cuContext, dec.GetWidth(),
dec.GetHeight(), NV_ENC_BUFFER_FORMAT_ARGB));
pEnc->CreateEncoder(&initializeParams);
//...

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


//Copy the pixel data from BGRA_frame into the input frame address obtained
above
NvEncoderCuda::CopyToDeviceFrame(cuContext,
                        BGRA_frame.pixels,
                        BGRA_frame.pitch,
                        (CUdeviceptr)encoderInputFrame->inputPtr,
                        encoderInputFrame->pitch,
                        pEnc->GetEncodeWidth(),
                        pEnc->GetEncodeHeight(),
                        CU_MEMORYTYPE_DEVICE,
                        encoderInputFrame->bufferFormat,
                        encoderInputFrame->chromaOffsets,
                        encoderInputFrame->numChromaPlanes);
pEnc->EncodeFrame(vPacket);
```

3.6.2 Allocating an NvCVImage Object Buffer

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

3.6.2.1 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 97 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
);
```

3.6.2.2 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 will not 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()`.

3.6.3 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.

3.6.3.1 Transferring Input Images from a CPU Buffer to a GPU Buffer

1. Create an `NvCVImage` object to use as a staging GPU buffer that has the same dimensions and format as the source CPU buffer.

```
NvCVImage srcGpuPlanar(inWidth, inHeight, NVCV_BGR, NVCV_F32,
NVCV_PLANAR, NVCV_GPU,1)
```

2. Create a staging buffer in one of the following ways:
   - To avoid allocating memory in a video pipeline, create a GPU buffer that has the same dimensions and format as required for input to the video effect filter.

```
NvCVImage srcGpuStaging(inWidth, inHeight, srcCPUImg.pixelFormat,
srcCPUImg.componentType, srcCPUImg.planar, NVCV_GPU)
```

   - To simplify your application program code, declare an empty staging buffer.

```
NvCVImage srcGpuStaging;
```

     An appropriate buffer will be allocated or reallocated as needed.

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

```
//Read the image into srcCPUImg
NvCVImage_Transfer(&srcCPUImg, &srcGPUPlanar, 1.0f, stream,
&srcGPUStaging)
```

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

1. Create an `NvCVImage` object to use as a staging GPU buffer that has the same dimensions and format as the destination CPU buffer.

```
NvCVImage dstGpuPlanar(outWidth, outHeight, NVCV_BGR, NVCV_F32,
NVCV_PLANAR, NVCV_GPU, 1)
```

2. Create a staging buffer in one of the following ways:
   - To avoid allocating memory in a video pipeline, create a GPU buffer that has the same dimensions and format as the output of the video effect filter.

     ```
     NvCVImage dstGpuStaging(outWidth, outHeight, dstCPUImg.pixelFormat,
     dstCPUImg.componentType, dstCPUImg.planar, NVCV_GPU)
     ```

   - To simplify your application program code, declare an empty staging buffer,

     ```
     NvCVImage dstGpuStaging;
     ```

     An appropriately sized buffer will be allocated as needed.

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

```
//Retrieve the image from the GPU to CPU, perhaps with conversion.
NvCVImage_Transfer(&dstGpuPlanar, &dstCPUImg, 1.0f, stream,
&dstGpuStaging);
```

3.7 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.

3.7.1 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.

3.7.2 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.

3.7.3 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 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() ...
...
```

3.7.4 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()` to get the properties of the current GPU.

   c). Use custom code in your application to analyze the properties retrieved by `cudaGetDeviceProperties()` to determine whether the GPU is the best GPU for each 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 7.0, 7.5, 8.0, and 8.6, which denotes a GPU based on the NVIDIA Turing™ GPU, NVIDIA Ampere™ GPU architectures or NVIDIA Volta V100.

```
// 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; //>= 7.0 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). To select the GPU for the task, call `cudaSetDevice()`.

   b). Make all the function calls required to perform the task.

   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);
  …
```

3.8 Batch Processing

Some video effects have higher performance when multiple images are submitted in a contiguous batch. All video effects can process batches, regardless of whether they have a specifically tuned model.

Submitting a batch differs from submitting an image in the following ways:

- Allocating batched buffers
- Selecting a batched model
- Setting the batched images
- Setting the batch size for `NvVFX_Run()`.

3.8.1 What is a Batch?

In the Video Effects SDK, a batch is a contiguous buffer that contains multiple images with the same structure. For some effects, this process can yield a higher throughput than submitting the images individually.

The batch is represented by an `NvCVImage` descriptor for the first image and a separate batch size parameter that is set when you run the following command:

```
NvVFX_SetU32(effect, NVVFX_BATCH_SIZE, batchSize);
```

This value is sampled each time `NvVFX_Run()` is called, which allows the batch size to change with each call to `NvVFX_Run()`.

3.8.2 Batch Utilities

The following utility functions in `BatchUtilities.cpp` help you to work with image batches:

- `AllocateBatchBuffer()` can be used to allocate a buffer for a batch of images.
- `NthImage()` can be used to set a view into the nth image in a batched buffer.
- `ComputeImageBytes()` can be used to determine the number of bytes for each image to advance the pixel pointer from one image to the next.
- `TransferToNthImage()` makes it easy to call `NvCVImage_Transfer()` to set one of the images in a batch.
- `TransferFromNthImage()` makes it easy to call `NvCVImage_Transfer()` to copy one of the images in a batch to a regular image.
- `TransferToBatchImage()` transfers multiple images from different locations to a batched image.
- `TransferFromBatchImage()` can be used to retrieve the images in a batch to different images in different locations.
- `TransferBatchImage()` transfers all images in a batch to another compatible batch of images.

The last three functions can also be accomplished by repeatedly calling the Nth image APIs, but the source code illustrates an alternative method of accessing images in a batch.

3.8.3 Allocating Batched Buffers

To allocate batched buffers, call the `AllocateBatchBuffer()` function, which will allocate an image that is N times taller than the prototypical image.

Note: The allocation cannot always be interpreted this way, especially if the pixels are planar.

The purpose of this function is mainly to provide storage and then dispose this storage when the `NvCVImage` goes out of scope, or its destructor is called.

You can use your own method to allocate the storage for the batched images. The image that the `AllocateBatchBuffer()` yields is only used for bookkeeping and is never used in any of the Video Effects APIs.

Note: The `VideoEffects` APIs only require an `NvCVImage` descriptor for the first image.

3.8.4 Selecting a Batch Model

The Video Effects comprise several runtime engines that not only implement an effect, are tuned to take best advantage of a particular GPU architecture, and are optimized to simultaneously process multiple inputs.

These multiple inputs, also known as a batch, and the engine that is optimized to process N images simultaneously is called a *Batch-N model*. An effect on a GPU architecture might have 1, 2, or more batch models.

Other than for efficiency, the batch size of images that are submitted to Video Effects is unrelated to the [maximum] batch size of a batch model. The maximum efficiency is achieved for image batch sizes that are integral multiples of the model batch size. For example, a batch-4 model will be most efficient when passed image batches of size 4, 8, 12, and so on, although you can also feed the models in batches of 3, 5, 10, or even 1.

All effects come with a model that is optimized for 1 image, a batch-1 model. Some effects might have other models that can simultaneously and efficiently process larger batches of images. If your application can take advantage of the higher efficiency of these larger batches, you can specify the batch model you want before loading it.

By default, a batch-1 model is loaded when `NvVFX_Load()` is called. To select another model, call:

```
NvVFX_SetU32(effect, NVVFX_MODEL_BATCH, modelBatch);
```

where you specify the model batch size and then call:

```
NvVFX_Load(effect);
```

If the model with the batch size you want is available, `NVCV_SUCCESS` is returned. Otherwise, an appropriate substitution will be made, and the `NVCV_ERR_MODELSUBSTITUTION` status is returned. Although it might not be the result you wanted, the most efficient batch model for your specified batch size is loaded.

Note: This status is a notification and not an error.

The batch size of the loaded model can be subsequently queried by running the following command:

```
NvVFX_GetU32(effect, NVVFX_MODEL_BATCH, &modelBatch);
```

To find the available model batch sizes, query the `INFO` string.

```
NvVFX_GetString(effect, NVVFX_INFO, &infoStr);
```

Note: The `INFO` string is designed to be humanly parsable.

Programmatically, you can repeatedly call the following with increasing sizes, until the returned size is smaller than the requested size:

```
NvVFX_SetU32(effect, NVVFX_MODEL_BATCH, modelBatch);
NvVFX_Load(effect);
NvVFX_GetU32(effect, NVVFX_MODEL_BATCH, &modelBatch);
```

The values that are returned by `NvVFX_GetU32()` are identical to the available model batch sizes. This information might be useful at runtime startup to maximize throughput and minimize latency.

`NvCV_Load()` is a heavyweight operation (on the order of seconds), so we recommend that you avoid repeated calls when the service is online. The code example above is just a suggestion for offline querying purposes, during startup or during quarterly tune-ups.

3.8.5 Setting the Batched Images

The API only takes the image descriptors for the first image in a batch. The following sample allocates the `src` and `dst` batch buffers and sets the input and outputs via the views of the first image in each batch buffer.

```
NvCVImage srcBatch, dstBatch, nthSrc, nthDst;
AllocateBatchBuffer(&srcBatch, batchSize, srcWidth, srcHeight,
                    ...);
AllocateBatchBuffer(&dstBatch, batchSize, dstWidth, dstHeight,
                     ...);
NthImage(0, srcHeight, &srcBatch, &nthSrc);
NthImage(0, dstHeight, &dstBatch, &nthDst);
NvVFX_SetImage(effect, NVVFX_INPUT_IMAGE,  &nthSrc);
NvVFX_SetImage(effect, NVVFX_OUTPUT_IMAGE, &nthDst);
```

Since the image descriptors are copied into the Video Effects SDK, this can be simplified to the following:

```
  NvCVImage srcBatch, dstBatch, nth;
  AllocateBatchBuffer(&srcBatch, batchSize, srcWidth, srcHeight,
                      ...);
  AllocateBatchBuffer(&dstBatch, batchSize, dstWidth, dstHeight,
                      ...);
  NvVFX_SetImage(effect, NVVFX_INPUT_IMAGE,
                 NthImage(0, srcHeight, &srcBatch, &nth));
  NvVFX_SetImage(effect, NVVFX_OUTPUT_IMAGE,
                 NthImage(0, dstHeight, &dstBatch, &nth));
```

The other images in the batch are computed by advancing the pixel pointer by the size of each image.

The other aspect of setting the batched images is determining how to set the pixel values. Each image in the batch is accessible by calling the `NthImage()` function:

```
NthImage(n, imageHeight, &batchImage, &nthImage);
```

You can then use the same techniques that were used for other NvCVImages on the recently initialized nthImage view. As previously suggested, NthImage() is just a thin wrapper around NvCVImage_InitView() and can be used instead. The NvVFX_Transfer() functions in BatchUtilities.cpp can be used to copy pixels to and from the batch.

3.8.6 Setting the Batch Size for NvVFX_Run()

By default, the batch size processed by NvVFX_ Run() is 1. Before you call NvVFX_Run() to process a batch of any other size, call this before calling NvVFX_Run():

```
NvVFX_SetU32(effect, NVVFX_BATCH_SIZE, batchSize);
```

As previously noted, there is no connection between the MODEL_BATCH and the BATCH_SIZE, except what the highest performance will be when BATCH_SIZE is an integral multiple of MODEL_BATCH. All images in the submitted batch will be processed.

3.8.7 Batching for Webcam Denoising

Webcam denoising filter uses state variables to track the input video streams to remove temporal noise. However, a batched input, srcBatch , can contain frames from multiple videos in arbitrary number and order.

When you use webcam denoising, there are some additional steps you need to complete **before** NvVFX_Run() to provide information about the ordering and the video source of the images in the batch. This process allows each input video stream to be properly tracked.

Since one state variable tracks one video stream, you need to:

1. Create one state variable per video stream in your application. If you have N input video streams, you need to create N state variables:

   ```
   void* arrayOfStates[N] = {nullptr};

   cudaMalloc(&arrayOfStates[0], stateSizeInBytes);
   cudaMemset(arrayOfStates[0], 0, stateSizeInByte);
   . . .
   . . .
   cudaMalloc(&arrayOfStates[N-1], stateSizeInBytes);
   cudaMemset(arrayOfStates[N-1], 0, stateSizeInByte);
   ```

2. Create an array , batchOfStates, to hold the memory addresses of a batch of state variables.

   The size of this array will be equal to the batch size.

Note: The size of the batch need not be equal to the number of input video streams and the batch can contain frames from different video streams in arbitrary order. A batch can contain multiple

frames from the same video stream as well, but they must be arranged in the batch in a chronological order.

This array holds the addresses of state variables in the order that corresponds to the video source of the images in the batched input, `srcBatch`. If the *n-th* image in the batch arises from the *m-th* video stream, the *n-th* element in this array will hold the address of the *m-th* state variable.

For example, assuming the batch size is 6 and the batched input, `srcBatch`, contains the frames from input stream #N-1, input stream #q, input stream #q, input stream #1, input stream #p, input stream #0 in this order. You need to copy the address of the corresponding state variable in `batchOfStates` and pass this array to the SDK using `NvVFX_SetObject`:

```
void* batchOfStates[] = {arrayOfStates[N-1], arrayOfStates[q],
                         arrayOfStates[q], arrayOfStates[1],
                         arrayOfStates[p], arrayOfStates[0]};
vfxErr =  NvVFX_SetObject(effectHandle, NVVFX_STATE, (void*)batchOfStates);
```

Depending on whether there is change in the batch size and/or ordering of the source of images in the batch, the size and the contents of `batchOfStates` can be changed and set using `NvVFX_SetObject` before every `NvVFX_Run()`.

3.8.8 Example Code

Example code is provided in the following files:

- `BatchEffectApp.cpp`, which provides the functional example code and includes a list of image files and the code to process these files as a batch.
- `BatchDenoiseEffectApp.cpp`, which provides the functional example code to use batching with Webcam Denoising.

  The code accepts multiple video files as the input and processes the frames from the input videos in batches of the user-specified size.

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;
  signed 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             reserved[2];
  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
- 2: UYVY
- 3: YUV

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 53 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`.

reserved

Type: `unsigned char[2]`

Reserved for padding and future capabilities. Set this parameter to 0.

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_YUV444

Luminance and full bandwidth 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_GREEN_SCREEN "Green Screen"

AI green screen filter.

NVVFX_FX_BGBLUR "BackgroundBlur"

Background Blur filter.

NVVFX_FX_ARTIFACT_REDUCTION "ArtifactReduction"

Artifact reduction filter.

NVVFX_FX_SUPER_RES "SuperRes"

Super-resolution filter.

NVVFX_FX_SR_UPSCALE " Upscale"

Upscale filter.

NVVFX_FX_DENOISING "Denoising"
Webcam Denoising filter

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_STRENGTH "Strength"

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

NVVFX_MODE "Mode"

The mode of an AI green screen filter.

- 0: Quality mode
- 1: Performance mode

NVVFX_TEMPORAL "Temporal"

Apply temporal filtering.

NVVFX_GPU "GPU"

Preferred GPU (optional).

NVVFX_BATCH_SIZE "BatchSize"

The number of images submitted in a batch.

NVVFX_MODEL_BATCH "ModelBatch"

The batch size of a video effects model.

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.
- **Semi-planar** pixels are a mix between **interleaved** and **planar** components.

  These types of pixels are found in the video world, where the [Y] component sits in one plane, and the [UV] components are interleaved in another plane.

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

Video pixels (YUV) have a richer set of pixel organizations. In the descriptions of these 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_CYUV 10

This type specifies the interleaved (chunky) YUV 4:4:4 pixels.

Pixels are arranged in [YUV] groups.

NVCV_CYVU 12

This type specifies interleaved (chunky) YVU 4:4:4 pixels.

Pixels are arranged in [YVU] groups.

NVCV_YUV 3

NVCV_I420 3

NVCV_IYUV 3

NVCV_I420 3 (used with NVCV_YUV420)
NVCV_IYUV 3 (used with NVCV_YUV420)

NVCV_I444 3 (used with NVCV_YUV444)

NVCV_YM24 3 (used with NVCV_YUV444)

Each of these types specifies one of the following planar YUV arrangements:

- YUV 4:2:2
- YUV 4:2:0
- YUV 4:4:4

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

NVCV_YVU 5

NVCV_YVU 5
NVCV_YV12 5 (used with NVCV_YUV420)

NVCV_YM42 5 (used with NVCV_YUV444)

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

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

NVCV_YCUV 7
NVCV_NV12 7 (used with NVCV_YUV420)

NVCV_NV24 7 (used with NVCV_YUV444)

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

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

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 (used with NVCV_YUV420)

NVCV_NV42 9 (used with NVCV_YUV444)

Each of these types specifies NV21 pixels, which are in the semiplanar YUV 4:2:2 format, the semiplanar YUV 4:2:0 format, or the semiplanar YUV 4:4:4 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. Most video formats use this sampling scheme.

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. Most video formats use this sampling scheme.

Example: Creating an HD NV12 CUDA buffer

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

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

```
NvCVImage img;
NvCVImage_Init(&img, 1920, 1080, 1920, existingBuffer, NVCV_YUV420, NVCV_U8,
               NVCV_NV12, NVCV_CUDA);
img.colorSpace = NVCV_709 | NVCV_VIDEO_RANGE | NVCV_CHROMA_COSITED;
```

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

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 page48 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

`NVCV_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 `obj` out parameter.

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

- `NVCV_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

- `NVCV_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

```
const char* 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` or `NVVFX_STRENGTH` selector strings. Any irrelevant selector strings return 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

- `NVCV_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

- `NVCV_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

- `NVCV_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` or `NVVFX_STRENGTH` or `NVVFX_TEMPORAL` or `NVVFX_GPU` or `NVVFX_BATCH_SIZE` or `NVVFX_MODEL_BATCH` selector strings. Any irrelevant selector strings return 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

- `NVCV_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

`NVCV_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
);
```

4.4.13 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.

4.4.13.1 Return Value

`NVCV_SUCCESS` on success

4.4.13.2 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.

4.4.14 NvVFX_SetCudaStream

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

4.4.14.1 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.

4.4.14.2 Return Value

- NVCV_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 CUDA stream in which the specified video effect filter will run to the parameter stream.

4.4.15 NvVFX_SetF32

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

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 32-bit floating-point parameter.

paramName

Type: NvVFX_ParameterSelector

The NVVFX_SCALE or NVVFX_STRENGTH selector strings. Any irrelevant selector strings return an error.

val

Type: float

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

4.4.15.2 Return Value

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

4.4.16 NvVFX_SetImage

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

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 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.

4.4.16.2 Return Value

- `NVCV_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 specified input or output image buffer for the specified video effect filter to the `im` parameter.

4.4.17 NvVFX_SetString

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

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 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.

4.4.17.2 Return Value

- `NVCV_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 character string parameter for the specified video effect filter to the parameter str.

4.4.18 NvVFX_SetU32

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

4.4.18.1 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` or `NVVFX_STRENGTH` or `NVVFX_TEMPORAL` or `NVVFX_GPU` or `NVVFX_BATCH_SIZE` or `NVVFX_MODEL_BATCH` selector strings. Any irrelevant selector strings return an error.

val

Type: `unsigned int`

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

4.4.18.2 Return Value

- `NVCV_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.

4.4.18.3 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 50 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 54 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

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

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,
  CUstream stream
);
```

4.5.4.1 Parameters

fg [in]

Type: `const NvCVImage *`

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

bg [in]

Type: `const NvCVImage *`

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

.

mat [in]

Type: `const NvCVImage *`

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

dst [out]

Type: `NvCVImage *`

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

stream [out]

Type: `CUstream`

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

4.5.4.2 Return Value

- `NVCV_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 image over a background image. The `fg`, `bg`, `mat`, and `dst` images must be of the same component type.

4.5.5 NvCVImage_CompositeRect

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

4.5.5.1 Parameters

fg [in]

Type: const `NvCVImage *`

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

fgOrg [in]

Type: `const NvCVPoint2i *`

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

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

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

bg [in]

Type: const `NvCVImage *`

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

bgOrg [in]

Type: `const NvCVPoint2i *`

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

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

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

mat [in]

Type: `const NvCVImage *`

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

mode [in]

Type: `unsigned int`

The compositional mode selection. Currently only mode 0 (normal, over) is implemented, and other values return a parameter error.

dst [out]

Type: `NvCVImage *`

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

dstOrg [in]

Type: `const NvCVPoint2i *`

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

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

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

stream [out]

Type: `CUstream`

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

4.5.5.2 Return Value

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

4.5.5.3 Remarks

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

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

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

4.5.6 NvCVImage_CompositeOverConstant

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

4.5.6.1 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.

4.5.6.2 Return Value

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

4.5.6.3 Remarks

This function uses the specified matte image to composite a BGRu8 or RGNU8 image over a constant color field. This function is primarily used for debugging on the CPU and may be discontinued in the future.

4.5.7 NvCVImage_Create

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

4.5.7.1 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 50 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 54 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.

4.5.7.2 Return Value

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

4.5.7.3 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++).

4.5.8 NvCVImage_Dealloc

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

4.5.8.1 Parameters

im [in,out]

Type: `NvCVImage *`

Pointer to the image whose image buffer will be freed.

4.5.8.2 Return Value

Does not return a value.

4.5.8.3 Remarks

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

4.5.9 NvCVImage_Destroy

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

4.5.9.1 Parameters

im

Type: `NvCVImage *`

Pointer to the image that will be destroyed.

4.5.9.2 Return Value

Does not return a value.

4.5.9.3 Remarks

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

4.5.10 NvCVImage_Init

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

4.5.10.1 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 50 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 54 for more information.

4.5.10.2 Return Value

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

4.5.10.3 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.

4.5.11 NvCVImage_InitView

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

4.5.11.1 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.

4.5.11.2 Return Value

Does not return a value.

4.5.11.3 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.

4.5.12 NvCVImage_Realloc

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

4.5.12.1 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 50 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 54 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.

4.5.12.2 Return Value

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

4.5.12.3 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.

4.5.13 NvCVImage_Transfer

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

4.5.13.1 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 none of the component types is 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.

4.5.13.2 Return Value

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

4.5.13.3 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, for example, RGB or BGR.

Table 4-1. Pixel Conversions

	u8→u8	u8→f32	f32→u8
Y→Y	X		X
Y→A	X		X
Y→RGB	X	X	X
Y→RGBA	X	X	X
A→Y	X		X
A→A	X		X
A→RGB	X	X	X
A→RGBA	X		
RGB→Y	X	X	
RGB→A	X	X	
RGB→RGB	X	X	X
RGB→RGBA	X	X	X
YUV420→RGB	X	X	
YUV422→RGB	X	X	
YUV444→RGB	X	X	
RGB→YUV420	X		X
RGB→YUV422	X		X
RGB→YUV444	X		X

Here is some additional information about these conversions:

- Conversions between chunky and planar pixel organizations occur in either direction.
- Conversions between CPU and GPU memory types can occur in either direction.
- Conversions between different orderings of components occur in either direction, for example, BGR → RGB.
- For RGBA (or BGRA) destinations, most implementations do not change the alpha channel, so we recommend that you set it at the initialization time with
  `[cuda]memset(im.pixels, -1, im.pitch * im.height)` or
  `[cuda]memset(im.pixels, -1, im.pitch * im.height* im.numComponents)`
  or chunky and planar u8 images, respectively.
- Other than pitch, if no conversion is necessary, all pixel format transfers are implemented, with `cudaMemcpy2DAsync()`.

  Another restriction in YUV→YUV transfers is that the formats, layouts and colorspaces must match between `src` and `dst`.

- YUV420 and YUV422 and YUV444 sources have several variants. The colorspace must be set manually prior to Transfer.

  See "YUV Color Spaces" on page 53 for more information.
- There are also RGBf16→RGBf32 and RGBf32→RGBf16 transfers.
- CPU→CPU transfers are synchronous.
- Additionally, when the `src` and `dst` formats are the same, all formats are accommodated on CPU and GPU, and this can be used as a replacement for `cudaMemcpy2DAsync()` (which it utilizes).
- If the `src` and `dst` have different sizes, the transfer still occurs, but it will be clipped to the smaller size.

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

4.5.14 NvCVImage_TransferRect

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

4.5.14.1 Parameters

src [in]

Type: `const NvCVImage *`

Pointer to the source image that will be transferred.

srcRect [in]

Type: `const NvCVRect2i *`

Pointer to the source image rectangle that will be transferred.

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

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

dst [out]

Type: `NvCVImage *`

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

dstPt [in]

Type: `const NvCVPoint2i *`

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

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

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

scale [in]

Type: `float`

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

Here are the typical values:

1.0f

255.0f

1.0f/255.0f

This parameter is ignored if 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.

4.5.14.2 Return Value

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

4.5.14.3 Remarks

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

In addition to chunky, `NvCVImage_TransferRect` works on planar and semi-planar, andthere is no difference in performance.

When you copy YUV rectangles, if you are not careful, unexpected clipping will occur:

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

4.5.15 NvCVImage_TransferFromYUV

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

4.5.15.1 Parameters

y [in]

Type: `const void *`

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

yPixBytes [in]

Type: `int`

The byte stride between y pixels horizontally.

yPitch [in]

Type: `int`

The byte stride between y pixels vertically.

u [in]

Type: `const void *`

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

v [in]

Type: `const void *`

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

uvPixBytes [in]

Type: `int`

The byte stride between u or v pixels horizontally.

uvPitch [in]

Type: `int`

The byte stride between u or v pixels vertically.

yuvColorSpace [in]

Type: unsigned `int`

The yuv colorspace, specifying range, chromaticities, and chrominance phase.

yuvMemSpace [in]

Type: unsigned `int`

The byte stride between y pixels horizontally.

dst [out]

Type: `NvCVImage *`

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

dstRect [in]

Type: `const NvCVRect2i *`

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

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

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

scale [in]

Type: `float`

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

Here are the typical values:

1.0f

255.0f

1.0f/255.0f

When the component type of all images is the same (all integer or all floating-point), this parameter is ignored.

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.

4.5.15.2 Return Value

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

4.5.15.3 Remarks

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

4.5.16 NvCVImage_TransferToYUV

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

4.5.16.1 Parameters

src [in]

Type: `const NvCVImage *`

Pointer to the source image that will be transferred.

srcRect [in]

Type: `const NvCVRect2i *`

Pointer to the source image rectangle that will be transferred.

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

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

y [out]

Type: `NvCVImage *`

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

yPixBytes [in]

Type: `int`

The byte stride between y pixels horizontally.

yPitch [in]

Type: `in`

The byte stride between y pixels vertically.

u [out]

Type: `NvCVImage *`

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

v [out]

Type: `NvCVImage *`

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

uvPixBytes [in]

Type: `int`

The byte stride between u or v pixels horizontally.

uvPitch [in]

Type: `int`

The byte stride between u or v pixels vertically.

yuvColorSpace [in]

Type: unsigned `int`

The yuv colorspace, specifying range, chromaticities, and chrominance phase.

yuvMemSpace [in]

Type: unsigned `int`

The memory space where the pixel buffers reside.

scale [in]

Type: `float`

A scale factor 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.

4.5.16.2 Return Value

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

4.5.16.3 Remarks

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

4.5.17 NvCVImage_MapResource

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

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

4.5.17.1 Parameters

im [in,out]

Type: `NvCVImage *`

The image to be mapped.

stream [out]

Type: `struct CUStream_st *`

The stream on which the mapping is to be performed.

4.5.17.2 Return Value

- `NVCV_SUCCESS` on success.

4.5.17.3 Remarks

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

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

4.5.18 NvCVImage_UnmapResource

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

4.5.18.1 Parameters

im [in,out]

Type: `NvCVImage *`

The image to be mapped.

stream [out]

Type: `struct CUStream_st *`

The stream on which the mapping is to be performed.

4.5.18.2 Return Value

- `NVCV_SUCCESS` on success.

4.5.18.3 Remarks

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

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

4.5.19 NVWrapperForCVMat

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

4.5.19.1 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.

4.5.19.2 Return Value

Does not return a value.

4.5.19.3 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 gpuMem,
  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 50 for more information.

gpuMem [in]

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See "Memory Types" on page 54 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

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

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_MODELDEPENDENCIES

There is no model with dependencies that match this system

NVCV_ERR_PARSE

There has been a parsing or syntax error while reading a file

NVCV_ERR_MODELSUBSTITUTION

The specified model does not exist and has been substituted.

NVCV_ERR_READ

An error occurred while reading a file.

NVCV_ERR_WRITE

An error occurred while writing a file.

NVCV_ERR_PARAMREADONLY

The selected parameter is read-only.

NVCV_ERR_TRT_ENQUEUE

TensorRT enqueue failed.

NVCV_ERR_TRT_BINDINGS

Unexpected TensorRT bindings.

NVCV_ERR_TRT_CONTEXT

An error occurred while creating a TensorRT context.

NVCV_ERR_TRT_INFER

There was a problem creating the inference engine.

NVCV_ERR_TRT_ENGINE

There was a problem deserializing the inference runtime engine.

NVCV_ERR_NPP

An error has occurred in the NPP library.

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.

There are a some other CUDA-related errors that are not listed here. However, the function `NvCV_GetErrorStringFromCode()` will turn the error code into a string to help you debug.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

ARM

ARM, AMBA and ARM Powered are registered trademarks of ARM Limited. Cortex, MPCore and Mali are trademarks of ARM Limited. All other brands or product names are the property of their respective holders. "ARM" is used to represent ARM Holdings plc; its operating company ARM Limited; and the regional subsidiaries ARM Inc.; ARM KK; ARM Korea Limited.; ARM Taiwan Limited; ARM France SAS; ARM Consulting (Shanghai) Co. Ltd.; ARM Germany GmbH; ARM Embedded Technologies Pvt. Ltd.; ARM Norway, AS and ARM Sweden AB.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA, the NVIDIA logo, TensorRT™, CUDA®, Turing™ and Tesla® are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

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