NVIDIA Video Effects SDK

Programming Guide

PG-10470-001_v0.7.2 | January 2023

Table of Contents

Chapter 1. Effects in the Video Effects SDK

This section provides information about the effects in the NVIDIA® Video Effects SDK. To learn more about how to get started with Maxine and how to install the SDK, refer to the Video Effects SDK System Guide.

1.1. 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 and at least 512 pixels wide.
  - 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 and at least 512 pixels wide.

For best results, the aspect ratio of the image file must be 16:9. The filter works on 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 format, where each pixel is a 24-bit unsigned char value, and, therefore, 8-bit per pixel component.
- The output of the filter is written to an 8-bit (grayscale) GPU buffer.

Refer to Sample Applications Reference for more information about a sample application that you can use to experience this effect.

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 to which it was assigned.

   - 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 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.

Note: To keep temporal consistency, the AI green screen effect uses a state variable to track the input video stream. Refer to For AI Green Screen for more information about how to track the input stream.

1.2. 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 the AI green screen filter, the segmentation mask output of the AI green screen filter or other sources, and an output buffer.

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

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

Here are the requirements to use the Background Blur filter:

- The input must be a 24-bit BGR input image, and therefore, 8-bit per pixel component.

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

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

1.3. The Artifact Reduction Filter

This filter 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:

- Mode 0 removes lesser artifacts, preserves low gradient information better, and is suited for higher bitrate videos.
- Mode 1 is better suited for lower bitrate videos.

The filter's input/output is as follows:

- The input should be provided in a GPU buffer in BGR planar format, where each pixel component is a 32-bit float value.
- The output of the filter is a GPU buffer of the same size as the input image, in BGR planar format, where each pixel component is a 32-bit float value.

1.4. The Super Resolution Filter

The Super Resolution filter, while preserving the content, improves the resolution of low-resolution videos, enhances the details, and sharpens the output.

This filter enhances the resolution of low-resolution videos and enhances the details and sharpens the output while preserving the content. Mode 0 is suitable for upscaling lossy content that has encoding artifacts, and Mode 1 is suitable for lossless videos:

- Mode 0 enhances less and removes more encoding artifacts and is suited for lower-quality videos.
- Mode 1 enhances more and is suited for higher quality lossless videos.

The filter's input/output is as follows:

- The input should be provided in a GPU buffer in BGR planar format, where each pixel component is a 32-bit float value.
- The output of the filter is a GPU buffer in BGR planar format, where each pixel component is a 32-bit float value.

The following table illustrates the scale and resolution support for input videos to be used with the ArtifactReduction and SuperRes effects.

Table 1. Scale and Resolution Support for Input Videos

Effect	Scale Factor	Input Resolution Change	Output resolution range
Artifact reduction	This effect does not change the resolution, so the input and output range is the same.	[90p, 1080p]	[90p, 1080p]
Super resolution	4/3x	[90p, 2160p]	[120p, 2880p]
	1.5x	[90p, 2160p]	[135p, 3240p]
	2x	[90p, 2160p]	[180p, 4320p]
	3x	[90p, 720p]	[270p, 2160p]
	4x	[90p, 540p]	[360p, 2160p]

Note: A lower end GPU or certain MIG configurations (Linux only) might not have enough memory to support the maximum permitted input resolutions of the Super Resolution feature.

1.5. The Upscale Filter

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.

Upscale filter provides a floating-point strength value that 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.

The filter's input/output is as follows:

- The input should be provided in a GPU buffer in RGBA chunky/interleaved format, where each pixel is 32-bit and therefore, 8-bit per pixel component.
- The output of the filter is a GPU buffer in RGBA chunky/interleaved format, where each pixel is 32-bit, and, therefore, 8-bit per pixel component.

Here are some general recommendations:

- If a video without encoding artifacts needs a fast resolution increase, use Upscale.

- If a video has no encoding artifacts, to increase the resolution, use SuperRes with mode 1 for greater enhancement.
- If a video has fewer encoding artifacts, to remove artifacts, use ArtifactReduction only with mode 0.
- If a video has more encoding artifacts, to remove artifacts, use ArtifactReduction only with mode 1.
- To increase the resolution of a video with encoding artifacts:
  - For light artifacts, use SuperRes with mode 0.
  - Otherwise, use ArtifactReduction followed by SuperRes with mode 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 the *Scale and Resolution Support for Input Videos* table in The Super Resolution Filter.

1.6. 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 filter's input/output is as follows:

- The input should be provided in a GPU buffer in BGR planar format, where each pixel component is a 32-bit float value.
- The output of the filter is a GPU buffer in BGR planar format, where each pixel component is a 32-bit float value.

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

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

Note: To remove temporal noise, webcam denoising uses a state variable to track the input video stream. Refer to Creating and Setting State Variables for more information about how to track the input stream.

Chapter 2. NVIDIA Video Effects SDK API Architecture

This section provides information about the Video Effects API architecture.

2.1. 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 then load, run, and destroy the filter.

2.1.1. Creating a Video Effect Filter

Here is information about how to create a video effect filter.

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

- The `NvVFX_EffectSelector` type Refer to `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 for use in further API calls.

This example creates an AI green screen video effect filter.

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

2.1.2. 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.
- The selector string `NVVFX_MODEL_DIRECTORY`.
- A null-terminated string that indicates the path to the model folder.

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

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

2.1.3. Setting Up the CUDA Stream

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

1. Initialize a CUDA stream by calling one of the following functions.
   - 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.
   - The selector string `NVVFX_CUDA_STREAM`.
   - The CUDA stream that you created in the previous step.

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

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

2.1.4. Creating and Setting State Variables

2.1.4.1. 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.
  1. Query the size of the state variable by calling `NvVFX_GetU32()` with the `NVVFX_STATE_SIZE` selector string.

     ```
     unsigned int stateSizeInBytes;
     vfxErr =  NvVFX_GetU32(effectHandle, NVVFX_STATE_SIZE, &stateSizeInBytes);
     ```
  2. Allocate the necessary space for the state variable in the GPU by using `cudaMalloc()`.

     ```
     void* state[1];
     cudaMalloc(&state[0], stateSizeInBytes);
     ```
  3. 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.

2.1.4.2. For AI Green Screen

The AI green screen filter uses a state variable to track the input video streams. to keep the temporal consistency, and the SDK provides NvVFX_StateObjectHandle to represent a handle to the state variable.

The SDK user is responsible for completing the following tasks:

- Create the state variable.

  1. Query the size of the state variable by calling NvVFX_AllocateState with the NvCV_Status selector string.

     The function completes the following steps:

     a). Allocates the state variable.

     b). Resets the state variable.

     c). Assigns the handle in the second parameter which the SDK user can store for additional processing.

```
NvVFX_StateObjectHandle stateObjectHandle;
vfxErr =  NvVFX_AllocateState(effectHandle, &stateObjectHandle);
```

- Pass the state variable to the SDK.

  To pass the state variable to the SDK, use the NvVFX_SetStateObjectHandleArray function with the NVVFX_STATE selector string.

```
vfxErr =  NvVFX_SetObject(effectHandle, NVVFX_STATE, &stateObjectHandle);
```

- Reuse the state variable for another input video stream.

  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 by calling NvVFX_ResetState function. It returns NvCV_Status.

```
vfxErr = NvVFX_ResetState(effectHandle, stateObjectHandle);
```

- When the state variable is no longer in use, release the state variable by calling the NvVFX_DeallocateState function.

```
vfxErr = NvVFX_DeallocateState(effectHandle, stateObjectHandle);
```

Note: Refer to Batching for AI Green Screen for more information about how to track multiple input streams concurrently.

2.1.5. 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. Refer to the NvCVImage API Guide for more information about NvCVImage. These images are GPU buffers accepted by the filter. The application provides input and output buffers to the filter by setting them as required parameters.

The video effect filter requires the input to be provided in a GPU buffer. 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.

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

Note: BGRu8 chunky refers to a 24-bit pixel, with each B,G, and R pixel component being 8-bit. Similarly, RGBAu8 chunky refers to a 32-bit pixel, with each B,G, R and A pixel component being 8-bit.

In contrast, BGRf32 planar refers to the floating point precision for each pixel component, for example, each of the B, G and R pixel components occupy 32-bits. However, since these are planar images, these are not compact 96-bit pixels, and there are three 32-bit planes where each component of a particular pixel might be separated by megabytes.

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.
- 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);
```

2.1.6. Setting and Getting Other Parameters of a Video Effect Filter

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

Here is a list of the currently used formats:

- The filter handle that was created as explained in Creating a Video Effect Filter.
- 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 the fastest performance.

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

2.1.6.1. Example: Setting the Filter Mode for AI Green Screen

Here is an example of a task for the AI green screen filter.

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.
- 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);
```

2.1.6.2. Example: Enable CUDA Graph Optimization for AI Green Screen

The AI green screen filter supports CUDA Graph Optimization, which can be enabled by setting the correct value for `NVVFX_CUDA_GRAPH`.

Call the `NvVFX_SetU32()` function and specify the following information as parameters:

- The filter handle that was created (refer to Creating a Video Effect Filter for more information).
- The `NVVFX_CUDA_GRAPH` selector string.
- The following integers, which denote the optimization state:
  - 0: `Off`
  - 1: `On`

Note: The default state for CUDA Graph Optimization is `Off` for the AI green screen filter.

Calling the following `set` method **before** calling the `NvVFX_Load()` method initializes the AI green screen effect:

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

- If CUDA Graph Optimization is enabled, the first call to `NVVFX_RUN()` initializes the optimization.

  During graph initialization, if an error occurs, the `NVCV_ERR_CUDA` error code is returned.
- If the `set` method is called **after** the initialization, an `NvVFX_Run()` call returns the `NVCV_ERR_INITIALIZATION` error code.

Enabling CUDA Graph Optimization reduces the kernel launch overhead and might improve overall performance. To verify the improvement, we recommend that developers test their application with CUDA Graph Optimization by switching the state from `Off` and `On`.

2.1.6.3. Example: Optimize Memory Allocations for AI Green Screen

The AI green screen filter works on images of any width and height, but an increase in the resolution will cause the filter to allocate the necessary GPU memory.

The AI green screen filter supports the `NVVFX_MAX_INPUT_WIDTH` and `NVVFX_MAX_INPUT_HEIGHT` parameters, which can optimize memory allocations during `NvVFX_RUN()`.

Call the `NvVFX_SetU32()` function and specify the following information as parameters:

The filter handle that was created as explained in Creating a Video Effect Filter.

The selector string `NVVFX_MAX_INPUT_WIDTH` and `NVVFX_MAX_INPUT_HEIGHT`.

Integer values for the width and height, for example, 1920, 1080.

```
vfxErr = NvVFX_SetU32 (effectHandle, NVVFX_MAX_INPUT_WIDTH , 1920);
vfxErr = NvVFX_SetU32 (effectHandle, NVVFX_MAX_INPUT_HEIGHT , 1080);
```

The set method above must be called before the `NvVFX_Load()` method, which initializes the AIGS effect.

No additional memory will be allocated by the filter up to the values set by using the parameter selector strings above.

2.1.7. Summary of NVIDIA Video Effects SDK Accessor Functions

The following table provides the details about the SDK accessor functions.

Table 2. Video Effects SDK Accessor Functions

Property 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	unsigned long long	`NvVFX_SetU64()`
		`NvVFX_GetU64()`
Image buffer	NvCVImage	`NvVFX_SetImage()`
		`NvVFX_GetImage()`
Object	void	`NvVFX_SetObject()`
		`NvVFX_GetObject()`
Character string	const char*	`NvVFX_SetString()`
		`NvVFX_GetString()`
CUDA stream	CUstream	`NvVFX_SetCudaStream()`
		`NvVFX_GetCudaStream()`

2.1.8. Getting Information About a Filter and its Parameters

Here is some information about how to get 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
);
```

2.1.9. Getting a List of All Available Effects

Here is information about how to get a list of the 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);
```

2.2. Loading a Video Effect Filter

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

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

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

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

Note: If a set accessor function is used to change filter parameters, for the change to take effect, the filter might need to be reloaded before it is run.

2.3. 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.
- 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()
```

2.4. Destroying a Video Effect Filter

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

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

```
NvVFX_DestroyEffect(effectHandle);
```

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

2.5. 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. The Video Effects SDK provides functions for converting an image representation to `NvCVImage` and transferring images between CPU and GPU buffers.

For more information about `NvCVImage`, refer to the NvCVImage API Guide. This section provides a synopsis of the most frequently used functions with the Video Effects SDK.

2.5.1. Converting Image Representations to NvCVImage Objects

The 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.

2.5.1.1. Converting OpenCV Images to NvCVImage Objects

Here is information about how to convert OpenCV images to NvCVImage objects.

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

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

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

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

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

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

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

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

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

2.5.1.2. Converting Image Frames on GPU or CPU Buffers to NvCVImage Objects

Here is information about how to convert image frames on CPU or GPU buffers to `NvCVImage` objects.

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

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

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

2.5.1.3. Converting Decoded Frames from the NvDecoder to NvCVImage Objects

Here is information about 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.

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

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

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

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

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

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

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

Here is some additional information:

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

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

Here is information about how to convert an `NvCVImage` object to a buffer by using the NvEncoder.

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

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

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

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

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

2.5.2. 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.

2.5.2.1. 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. Refer to Allocation Constructor for more information.

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

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

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

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

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

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

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

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

```
NvCVImage gpuSrc(
  srcWidth,
  srcHeight,
```

```
  NVCV_BGR,
  NVCV_U8,
  NVCV_PLANAR,
  NVCV_GPU,
  1
);
```

2.5.2.2. 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 of the image until later.
   - To reallocate a buffer, call `NvCVImage_Realloc()`.

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

2.5.3. 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.

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

Here is information about how to transfer input images from a CPU buffer to a GPU buffer.

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

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

1. Create an `NvCVImage` object to use as a staging GPU buffer in one of the following ways:
   - To avoid allocating memory in a video pipeline, create a GPU buffer during the initialization phase, with the same dimensions and format as the CPU image.

     ```
     NvCVImage stageImg(srcCpuImg.width, srcCpuImg.height,
               srcCpuImg.pixelFormat, srcCpuImg.componentType,
               srcCpuImg.planar, NVCV_GPU);
     ```
   - To simplify your application program code, you can declare an empty staging buffer during the initialization phase.

     ```
     NvCVImage stageImg;
     ```

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

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

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

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

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

Here is information about how to transfer output images from a GPU buffer to a CPU buffer.

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

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

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

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

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

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

     ```
     NvCVImage stageImg;
     ```

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

   For more information about NvCVImage, refer to the NvCVImage API Guide.

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

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

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

Chapter 3. Multi-GPU Usage in the Video Effects SDK

This section provides information about using multiple GPUs, the default behavior, how to select GPUs for your tasks.

3.1. Using Multiple GPUs

Applications developed with the 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 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.

3.2. Default Behavior in Multi-GPU Environments

The `NvVFX_Load()` function internally calls `cudaGetDevice()` to identify the currently selected GPU.

The function then checks the compute capability of the currently selected GPU (default 0) to determine if the GPU architecture supports the Video Effects SDK:

- If yes, `NvVFX_Load()` uses the GPU.
- If no, `NvVFX_Load()` searches for the most powerful GPU that supports the 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.

3.3. 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 a specific GPU in a multi-GPU environment.

In this situation, ensure that the Video Effects SDK does not override your choice of GPU for applying the video effect filter.

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

3.4. Selecting Different GPUs for Different Tasks

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

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

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

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

   This example uses the compute capability to determine if a GPU's properties should be analyzed to determine if the GPU is the best GPU for applying a video effect filter. A GPU's properties are analyzed only if the compute capability is is 7.5, 8.6, or 8.9. This value denotes a GPU based on the NVIDIA Turing, NVIDIA Ampere, or the NVIDIA Ada architecture respectively.

```
// 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;
  if (DeviceIsBestForGame(&deviceProp)) gpuGame = dev;
  ...
  }
  cudaSetDevice(gpuVFX);
  vfxErr = NvVFX_Set...; // set parameters
  vfxErr = NvVFX_Load(eff);
```

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

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

   This example selects the best GPU to render a game and uses custom code to render the game. It then selects the best GPU to apply a video effect filter before calling the `NvCVImage_Transfer()` and `NvVFX_Run()` functions to apply the filter. This step avoids the need to save and restore the GPU for every 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);
  ...
}
```

3.5. Using Multi-Instance GPUs (Linux Only)

Applications that are developed with the Video Effects SDK can be deployed on Multi-Instance GPU (MIG) on supported devices, such as NVIDIA DGX™ A100. MIG allows you to partition a device into multiple GPU instances, up to seven, each with separate streaming multiprocessors, separate slices of the GPU memory, and separate pathways to the memory. This process ensures that heavy resource usage by an application on one partition does not impact the performance of the applications running on other partitions.

To run an application on a MIG partition, you do not have to call any additional SDK API in your application. You can specify which MIG instance to use for execution during the invocation of your application. You can select the MIG instance using one of the following options:

- The bare-metal method of using the `CUDA_VISIBLE_DEVICES` environment variable.
- The container method by using the NVIDIA Container Toolkit.

  MIG is supported only on Linux.

Refer to NVIDIA Multi-instance GPU User Guide for more information about using MIG.

Chapter 4. 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()`.

4.1. 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()`.

4.2. Batch Utilities

Here is some information about the available 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.

4.3. Allocation of 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 of 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 Video Effects APIs only require an `NvCVImage` descriptor for the first image.

4.4. Selecting a Batch Model

The Video Effects SDK comprises several runtime engines that not only implement an effect, are tuned to take the 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 one image, a batch-1 model. Some effects might have other models that can simultaneously and efficiently process large 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.

4.5. Setting the Batched Images

Here is some information about how to set 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.

4.6. Setting the Batch Size for NvVFX_Run()

Here is some information about how to set 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.

4.7. Batching When Using State Variables

This section provides information about batching with state variables.

4.7.1. For Webcam Denoising

Webcam denoising filters use state variables to track the input video streams to remove temporal noise. A batched input, `srcBatch`, can however 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 six 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 a 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()`.

4.7.1.1. Example Code

Here is some information about the example code in the SDK.

The example code is 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.

4.7.2. Batching for AI Green Screen

To keep temporal consistency, the AI Green Screen filter uses state variables to track the input video streams.

A batched input, `srcBatch`, can contain frames from multiple videos in an arbitrary number and order. However, when you create a batched input, the following constraints **must** be met:

- A batched input should not have multiple frames from the same video stream.
- Frames from the same video stream must be passed chronologically in a separate batch.

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

Set the maximum number of concurrent input video streams by calling `NvVFX_SetU32()` with the `NVVFX_MAX_NUMBER_STREAMS` selector string.

```
vfxErr = NvVFX_SetU32(effectHandle, NVVFX_MAX_NUMBER_STREAMS,
 numberOfInputVideoStreams);
```

This set method must be called **before** the `NvVFX_Load()` method, which initializes the effect.

One state variable tracks one video stream, so you need to complete the following steps:

1. In your application, create one state variable per video stream.

   If you have N input video streams, you need to create N state variables.

   ```
   NvVFX_StateObjectHandle stateObjectHandle;
   vfxErr =  NvVFX_AllocateState(effectHandle, &stateObjectHandle);
   ```
2. The SDK user can query the number of active state variables by calling the `NvVFX_GetU32` function with the `NVVFX_STATE_COUNT` parameter selector.

   The number of active state variables will be assigned to the provided parameter.

   ```
   vfxErr =  NvVFX_GetU32(effectHandle, NVVFX_STATE_COUNT,
   &numActiveStateVariables);
   ```
3. 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 should be less than or equal to the number of input video streams, because only one frame per stream can be in each batch. Due to a different number of video streams in each batch, the number of frames in a batch can differ between batches.

A batch can have a frame from different video streams in an arbitrary order.

This array holds the addresses of state variables in the order that corresponds to the video source of the images in 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, if the batch size is five and srcBatch contains the frames from input stream #N-1, 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 by using `NvVFX_SetObject`:

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

Depending on whether there is a 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 by using `NvVFX_SetStateObjectHandleArray` function before every `NvVFX_Run()`.

4.7.2.1. Example Code

The example code is in the `BatchAigsEffectApp.cpp` file, and this file provides the functional sample code that allows you to use batching with AI Green screen.

The code accepts multiple video files as the input and processes the frames from the input videos in batches that are equal to the number of video files.

Chapter 5. Video Effects SDK API Reference

This section provides information about the APIs in the Video Effects SDK.

5.1. Structures

This section provides information about the structure in the Video Effects SDK.

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

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

5.1.1. NvVFX_Handle

Here is detailed information about the `NvVFX_Handle` structure.

```
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`.

5.1.2. NvVFX_Object

Here is detailed information about the `NvVFX_Object` structure.

```
struct NvVFX_Object;
```

This is a pointer to an opaque data structure that is allocated by the application and needs to be disposed of by the application after the effect is destroyed. It is always referenced by its pointer, which is an `NvVFX_Handle`.

Defined in: `nvVideoEffects.h`.

5.1.3. NvVFX_StateObjectHandle

Here is detailed information about the `NvVFX_StateObjectHandle` structure.

```
typedef struct NvVFX_StateObjectHandleBase *NvVFX_StateObjectHandle;
```

This pointer represents the opaque handle that is associated with each instance of a state variable that is used by the SDK effects. The `NvVFX_AllocateState`, `NvVFX_DeallocateState`, `NvVFX_ResetState`, and `NvVFX_SetStateObjectHandleArray` function calls include this handle as a parameter.

Defined in: `nvVideoEffects.h`..

5.1.4. NvVFX_StateObjectHandleBase

Here is detailed information about the `NvVFX_StateObjectHandleBase` structure.

```
struct NvVFX_StateObjectHandleBase;
```

This structure represents a state variable that is used by the SDK.

Defined in: `nvVideoEffects.h`..

5.1.5. NvCVImage

Here is detailed information about the `NvCVImage` structure. For more information, refer to the NvCVImage API Guide.

```
typedef struct NvCVImage {
  unsigned int           width;
  unsigned int           height;
  unsigned int           pitch;
  NvCVImage_PixelFormat     pixelFormat;
  NvCVImage_ComponentType   componentType;
  unsigned char          pixelBytes;
  unsigned char        componentBytes;
  unsigned char          numComponents;
  unsigned char          planar;
  unsigned char          gpuMem;
  unsigned char          colorspace;
  unsigned char          reserved[2];
  void                   *pixels;
  void                   *deletePtr;
  void                   (*deleteProc)(void *p);
  unsigned long long     bufferBytes;

} NvCVImage;
```

Members

width

Type: `unsigned int`

The width, in pixels, of the image.

height

Type: `unsigned int`

The height, in pixels, of the image.

pitch

Type: `unsigned int`

The vertical byte stride between pixels.

pixelFormat

Type: `NvCVImage_PixelFormat`

The format of the pixels in the image.

componentType

Type: `NvCVImage_ComponentType`

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

pixelBytes

Type: `unsigned char`

The number of bytes in a chunky pixel.

componentBytes

Type: `unsigned char`

The number of bytes in each pixel component.

numComponents

Type: `unsigned char`

The number of components in each pixel.

planar

Type: `unsigned char`

Specifies the organization of the pixels in the image.

- `0`: Chunky
- `1`: Planar

gpuMem

Type: `unsigned char`

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

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

colorspace

Type: `unsigned char`

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

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

Refer to YUV Color Spaces for more information about the type definitions.

Always set the colorspace for 420, 422, or 444 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.

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 as described in the NvCVImage API Guide.

Defined in: `nvCVImage.h`.

5.2. Enumerations

The enumerations in the NVIDIA Video Effects SDK, related to pixel organization and image component data types are defined in the `nvCVImage.h` header file . For more information, refer to the NvCVImage API Guide.

5.3. Type Definitions

The Video Effects SDK type definitions provide selector strings for video effect filters and the parameters of a video effect filter.

5.3.1. NvVFX_EffectSelector

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

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

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 "Artifact Reduction"
AI-based artifact reduction

NVVFX_FX_SUPER_RES "Super Res"
AI-based super resolution

NVVFX_FX_SR_UPSCALE "Upscale"
AI-based fast video upscaler

NVVFX_FX_ARTIFACT_REMOVAL "remove_artifacts"
Artifact removal filter.

NVVFX_FX_SUPER_RES "super_res_V0"
Super resolution filter.

NVVFX_FX_DENOISING "Denoising"
Webcam Denoising filter.

5.3.2. NvVFX_ParameterSelector

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

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

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_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_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_CUDA_GRAPH "CudaGraph"
Enables CUDA Graph Optimization.

NVVFX_INFO "Info"
Get information about a video effect filter and its parameters.
NVVFX_MAX_INPUT_WIDTH "MaxInputWidth"
Maximum width of the supported input.
NVVFX_MAX_INPUT_HEIGHT "MaxInputHeight"
Maximum height of the supported input.
NVVFX_MAX_NUMBER_STREAMS "MaxNumberStreams"
Maximum number of concurrent input streams.
NVVFX_SCALE "Scale"
Scale factor. This is used to scale the values of the images during transfer to match formats in a pipeline of effects.
NVVFX_STRENGTH "Strength"
Strength for the filters that use this parameter. Higher strength implies a stronger effect.
NVVFX_STRENGTH_LEVELS "StrengthLevels"
Number of unique strength levels in the interval [0, 1]. Currently this only applies to the Webcam Denoise filter, and is set to 2, which implies that the two strength levels are 0 or 1.
NVVFX_MODE "Mode"
The mode of an AI green screen, Artifact resolution, and Super resolution filter.

- `0`: Quality mode
- `1`: Performance mode

NVVFX_TEMPORAL "Temporal"
Apply temporal filtering.
NVVFX_GPU "GPU"
Preferred GPU to use. This is an optional parameter.
NVVFX_BATCH_SIZE "BatchSize"
Size of a batch of input provided to a filter. The default value is 1.
NVVFX_MODEL_BATCH "ModelBatch"
The preferred batching model to use, which is tuned for a batch size of 1 or 8. This is applicable only to the AI Green Screen filter.
NVVFX_STATE "State"
An array of state variables.
NVVFX_STATE_SIZE "StateSize"
The number of bytes needed to store the state variable.
NVVFX_STATE_COUNT "NumStateObjects"
The number of active state object handles.

5.4. 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++.

5.4.1. NvVFX_CreateEffect

Here is detailed information about the `NvVFX_CreateEffect` function.

```
NvCV_Status NvVFX_CreateEffect(
  NvVFX_EffectSelector code,
```

```
..NvVFX_Handle *obj
);
```

Parameters

code [in]

Type: `NvVFX_EffectSelector`

The selection string for the type of video effect filter to be created. Refer to `NvVFX_EffectSelector` 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.

Return Value

`NVCV_SUCCESS` on success

Remarks

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

5.4.2. NvVFX_CudaStreamCreate

Here is detailed information about the `NvVFX_CudaStreamCreate` function.

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

Parameters

stream [out]

Type: `CUstream *`

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

Return Value

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

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.

5.4.3. NvVFX_CudaStreamDestroy

Here is detailed information about the `NvVFX_CudaStreamDestroy` function.

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

Parameters

stream [in]

Type: `CUstream`

The CUDA stream to destroy.

Return Value

Does not return a value.

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.

5.4.4. NvVFX_DestroyEffect

Here is detailed information about the `NvVFX_DestroyEffect` function.

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

Parameters

obj [in]

Type: `NvVFX_Handle`

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

Return Value

Does not return a value.

Remarks

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

5.4.5. NvVFX_GetCudaStream

Here is detailed information about the `NvVFX_GetCudaStream` function.

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

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.

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.

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.

5.4.6. NvCV_GetErrorStringFromCode

Here is detailed information about the `NvCV_GetErrorStringFromCode` function.

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

Parameters

code

Type: `NvCV_Status`

The `NvCV_Status` code for which to get an error string.

Return Value

The error string that corresponds to the specified error code.

Remarks

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

5.4.7. NvVFX_GetF32

Here is detailed information about the `NvVFX_GetF32` function.

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

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.

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.

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.

5.4.8. NvVFX_GetF64

Here is detailed information about the `NvVFX_GetF64` function.

```
NvVFX_Status NvVFX_GetF64(
NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  double *val
);
```

Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance from which you want to get the specified 64-bit floating-point parameter.

paramName

Type: `NvVFX_ParameterSelector`

The selector string for the 64-bit floating-point parameter that you want to get.

val

Type: `double *`

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

Return Value

- `NVCV_SUCCESS` on success.
- `NVVFX_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.
- `NVVFX_ERR_PARAMETER` when an unexpected NULL pointer was supplied.

Remarks

This function gets the value of the specified double-precision (64-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.

5.4.9. NvVFX_GetImage

Here is detailed information about the `NvVFX_GetImage` function.

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

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.

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.

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.

5.4.10. NvVFX_GetObject

Here is detailed information about the `NvVFX_GetObject` function.

```
NvVFX_Status NvVFX_GetObject(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  void **ptr
);
```

Parameters

obj

Type: `NvVFX_Handle`

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

paramName

Type: `NvVFX_ParameterSelector`

The selector string for the object parameter that you want to get.

ptr

Type: `void **`

Pointer to the address of the object where the retrieved object will be written.

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.

Remarks

This function gets the specified object for the specified video effect filter and writes the retrieved object to the location that was specified by the ptr parameter.

5.4.11. NvVFX_GetS32

Here is detailed information about the `NvVFX_GetS32` function.

```
NvVFX_Status NvVFX_GetS32(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
```

```
  int *val
);
```

Parameters

obj

Type: `NvVFX_Handle`

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

paramName

Type: `NvVFX_ParameterSelector`

The selector string for the 32-bit signed integer parameter that you want to get.

val

Type: `int *`

Pointer to the 32-bit signed integer where the value retrieved will be written.

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.

Remarks

This function gets the value of the specified 32-bit signed 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.

5.4.12. NvVFX_GetString

Here is detailed information about the `NvVFX_GetString` function.

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

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.

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.

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.

5.4.13. NvVFX_GetU32

Here is detailed information about the `NvVFX_GetU32` function.

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

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.

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.

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.

5.4.14. NvVFX_GetU64

Here is detailed information about the `NvVFX_GetU64` function.

```
NvVFX_Status NvVFX_GetU64(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  unsigned long long *val
);
```

Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance from which you want to get the specified 64-bit unsigned integer parameter.

paramName

Type: `NvVFX_ParameterSelector`

The selector string for the 64-bit unsigned integer parameter that you want to get.

val

Type: `unsigned long long *`

Pointer to the 64-bit unsigned integer where the value retrieved is to be written.

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.

Remarks

This function gets the value of the specified 64-bit unsigned integer parameter for the specified video effect filter and writes the value retrieved to the location specified by the `val` parameter.

5.4.15. NvVFX_Load

Here is detailed information about the `NvVFX_Load` function.

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

Parameters

obj [in]

Type: `NvVFX_Handle`

The handle to the video effect filter instance to load.

Return Value

`NVCV_SUCCESS` on success

Remarks

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

5.4.16. NvVFX_Run

Here is detailed information about the `NvVFX_Run` function.

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

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.

Return Value

`NVCV_SUCCESS` on success

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.

5.4.17. NvVFX_SetCudaStream

Here is detailed information about the `NvVFX_SetCudaStream` function.

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

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.

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.

Remarks

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

5.4.18. NvVFX_AllocateState

Here is detailed information about the `NvVFX_AllocateState` function.

```
NvCV_Status NvVFX_AllocateState(
  NvVFX_Handle obj,
  NvVFX_StateObjectHandle *handle
);
```

Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance from which you want to create a state variable.

handle

Type: `NvVFX_StateObjectHandle`

This is a pointer that the SDK uses to return the handle to the state variable.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_PARAMETER` when an unexpected NULL pointer was supplied.

Remarks

This function allocates a state variable and returns the handle for a specified video effect filter.

5.4.19. NvVFX_DeallocateState

Here is detailed information about the `NvVFX_DeallocateState` function.

```
NvCV_Status NvVFX_DeallocateState(
  NvVFX_Handle obj,
  NvVFX_StateObjectHandle handle
```

```
);
```

Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance from which you want to create a state variable.

handle

Type: `NvVFX_StateObjectHandle`

This is a pointer that the SDK uses to return the handle to the state variable.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_PARAMETER` when an unexpected NULL pointer was supplied.
- `NVCV_ERR_OBJECTNOTFOUND` when the supplied handle was not allocated by the filter instance.

Remarks

This function deallocates a state variable.

5.4.20. NvVFX_ResetState

Here is detailed information about the `NvVFX_ResetState` function.

```
NvCV_Status NvVFX_ResetState(
  NvVFX_Handle obj,
  NvVFX_StateObjectHandle handle
);
```

Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance that owns the state variable that is represented by the second parameter.

handle

Type: `NvVFX_StateObjectHandle`

This is a handle to the state variable.

Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_PARAMETER` when an unexpected NULL pointer was supplied.

- `NVCV_ERR_OBJECTNOTFOUND` when the supplied handle was not allocated by the filter instance.

Remarks

This function resets a state variable.

5.4.21. NvVFX_SetF32

Here is detailed information about the `NvVFX_SetF32` function.

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

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.

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.

Remarks

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

5.4.22. NvVFX_SetF64

Here is detailed information about the `NvVFX_SetF64` function.

```
NvVFX_Status NvVFX_SetF64(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  double val
);
```

Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance for which you want to set the specified 64-bit floating-point parameter.

paramName

Type: `NvVFX_ParameterSelector`

The selector string for the 64-bit floating-point parameter that you want to set.

val

Type: `double`

The double-precision floating-point number to which the parameter is to be set.

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.

Remarks

This function sets the specified double-precision (64-bit) floating-point parameter for the specified video effect filter to the `val` parameter.

5.4.23. NvVFX_SetImage

Here is detailed information about the `NvVFX_SetImage` function.

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

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.

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.

Remarks

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

5.4.24. NvVFX_SetObject

Here is the detailed information about the `NvVFX_SetObject` function.

```
NvVFX_Status NvVFX_SetObject(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  void *ptr
);
```

Parameters

obj

Type: `NvVFX_Handle`

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

paramName

Type: `NvVFX_ParameterSelector`

The selector string for the object parameter that you want to set.

ptr

Type: `void *`

Pointer to the object to which the object parameter is to be set.

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.

Remarks

This function sets the specified object for the specified video effect filter to the `ptr` parameter.

5.4.25. NvVFX_SetStateObjectHandleArray

Here is detailed information about the `NvVFX_SetStateObjectHandleArray` function.

```
NvVFX_Status NvVFX_SetStateObjectHandleArray(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  NvVFX_StateObjectHandle* handle
);
```

Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance.

paramname

Type: `NvVFX_ParameterSelector`

The selector string for the object parameter that you want to set: `NVVFX_STATE`.

Type: NvVFX_StateObjectHandle

Pointer to the array of state objects.

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_OBJECTNOTFOUND when the when an unexpected NULL pointer was supplied.

Remarks

This function sets the specified object for the specified video effect filter to the handle parameter.

5.4.26. NvVFX_SetS32

Here is detailed information about the NvVFX_SetS32 function.

```
NvVFX_Status NvVFX_SetS32(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  int val
);
```

Parameters

obj

Type: NvVFX_Handle

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

paramName

Type: NvVFX_ParameterSelector

The selector string for the 32-bit signed integer parameter that you want to set.

val

Type: int

The 32-bit signed integer to which the parameter will be set.

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.

Remarks

This function sets the value of the specified 32-bit signed integer parameter for the specified video effects to the val parameter.

5.4.27. NvVFX_SetString

Here is detailed information about the NvVFX_SetString function.

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

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.

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.

Remarks

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

5.4.28. NvVFX_SetU32

Here is detailed information about the `NvVFX_SetU32` function.

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

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.

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.

Remarks

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

5.4.29. NvVFX_SetU64

Here is detailed information about the `NvVFX_SetU64` function.

```
NvVFX_Status NvVFX_SetU64(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  unsigned long long val
);
```

Parameters

obj

Type: `NvVFX_Handle`

The handle to the video effect filter instance for which you want to set the specified 64-bit unsigned integer parameter.

paramName

Type: `NvVFX_ParameterSelector`

The selector string for the 64-bit unsigned integer parameter that you want to set.

val

Type: `unsigned long long`

The 64-bit unsigned integer to which you want to set the parameter.

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.

Remarks

This function sets the value of the specified 64-bit unsigned integer parameter for the specified video effect filter to the `val` parameter.

5.5. Return Codes

The `NvCV_Status` enumeration defines the following values that the 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_CONFIG
No suitable model exists for the specified parameter configuration.
NVCV_ERR_TOOSMALL
The supplied parameter or buffer is not large enough.
NVCV_ERR_TOOBIG
The supplied parameter is too big.
NVCV_ERR_WRONGSIZE
The supplied parameter is not the expected size.
NVCV_ERR_OBJECTNOTFOUND
The specified object was not found.
NVCV_ERR_SINGULAR
A mathematical singularity has been encountered.
NVCV_ERR_NOTHINGRENDERED
Nothing was rendered in the specified region.
NVCV_ERR_OPENGL
An OpenGL error has occurred.
NVCV_ERR_DIRECT3D
A Direct3D error has occurred.
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 an invalid memory address.
NVCV_ERR_CUDA
An unspecified CUDA error has occurred.

There are many 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.

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

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, CUDA Toolkit, cuDNN, DALI, DIGITS, DGX, DGX-1, DGX-2, DGX Station, DLProf, GPU, JetPack, Jetson, Kepler, Maxwell, NCCL, Nsight Compute, Nsight Systems, NVCaffe, NVIDIA Ampere GPU architecture, NVIDIA Deep Learning SDK, NVIDIA Developer Program, NVIDIA GPU Cloud, NVLink, NVSHMEM, PerfWorks, Pascal, SDK Manager, T4, Tegra, TensorRT, TensorRT Inference Server, Tesla, TF-TRT, Triton Inference Server, Turing, and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the United States 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