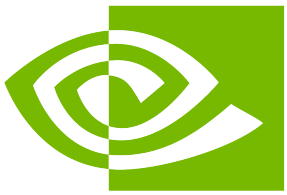

NVIDIA Video Effects (VFX) SDK User Guide

Release 1.3.0

NVIDIA Corporation

Sep 22, 2026

Contents

1 Get Started on Windows **3**
1.1 Hardware Requirements . . . 3
1.2 Software Requirements . . . 3
1.2.1 Install the VFX SDK . . . 4
1.2.2 Sample Applications . . . 6
1.2.3 Unicode Path Support . . . 6
1.2.3.1 Command-Line Arguments in Sample Applications . . . 6
1.2.4 Performance Reference . . . 6

2 Get Started on Linux **9**
2.1 Hardware Requirements . . . 9
2.2 Software Requirements . . . 9
2.2.1 Install the VFX SDK . . . 10
2.2.2 Sample Applications . . . 12
2.2.3 Performance Reference . . . 12
2.2.4 Run SDK in a Container . . . 13
2.2.4.1 Install the Prerequisite Software . . . 13
2.2.4.2 Fetch Sample Apps Repository . . . 14
2.2.4.3 Build Docker Image . . . 14
2.2.4.4 Launch Container . . . 14

3 AI Green Screen **15**

4 Background Blur **17**

5 Upscale **19**

6 Webcam Denoise **21**

7 Video Relighting **23**

8 Video Super Resolution **25**
8.1 Configuration and Requirements . . . 26
8.2 Choosing a VSR Mode . . . 26
8.2.1 Standard Upscaling Modes (0–4) . . . 26
8.2.2 Denoise Modes (8–11) . . . 27
8.2.3 Deblur Modes (12–15) . . . 27
8.2.4 High-Bitrate Modes (16–19) . . . 27
8.2.5 Streaming Modes (21 and 23) . . . 27

9 Video Frame Generation **29**
9.1 Operating Modes . . . 29
9.2 Model Modes . . . 29
9.3 Parameters . . . 30

9.4 Input and Output Requirements . . . 30
9.5 Hardware Requirements . . . 31
9.6 Recommended Use Cases . . . 31

10 TrueHDR **33**
10.1 Recommended Use Cases . . . 34

11 VFX SDK API Architecture **35**
11.1 Use a Video Effect Filter . . . 35
11.1.1 Creating a Video Effect Filter . . . 35
11.1.2 Setting the Path to the Model Folder . . . 35
11.1.3 Setting Up the CUDA Stream . . . 36
11.2 Create and Set State Variables . . . 37
11.2.1 For Webcam Denoising . . . 37
11.2.1.1 For AI Green Screen . . . 37
11.2.2 Setting the Input and Output Image Buffers . . . 38
11.2.3 Setting and Getting Other Parameters of a Video Effect Filter . . . 40
11.2.3.1 Example: Setting the Filter Mode for AI Green Screen . . . 40
11.2.3.2 Example: Enable CUDA Graph Optimization for AI Green Screen . . . 41
11.2.3.3 Example: Optimize Memory Allocations for AI Green Screen . . . 41
11.2.4 Summary of Video Effects SDK Accessor Functions . . . 41
11.2.5 Getting Information About a Filter and Its Parameters . . . 42
11.2.6 Getting a List of All Available Effects . . . 42
11.3 Load a Video Effect Filter . . . 42
11.4 Run a Video Effect Filter . . . 43
11.5 Destroy a Video Effect Filter . . . 43
11.6 Work with Image Frames on GPU or CPU Buffers . . . 44
11.6.1 Converting Image Representations to NvCVImage Objects . . . 44
11.6.1.1 Converting OpenCV Images to NvCVImage Objects . . . 44
11.6.1.2 Converting Image Frames on GPU or CPU Buffers to NvCVImage Objects . . . 45
11.6.1.3 Converting Decoded Frames from the NvDecoder to NvCVImage Objects . . . 45
11.6.1.4 Converting an NvCVImage Object to a Buffer Encodable by NvEncoder . . . 46
11.6.2 Allocating an NvCVImage Object Buffer . . . 47
11.6.2.1 Using the NvCVImage Allocation Constructor to Allocate a Buffer . . . 47
11.6.2.2 Using Image Functions to Allocate a Buffer . . . 48
11.6.3 Transferring Images Between CPU and GPU Buffers . . . 48
11.6.3.1 Transferring Input Images from a CPU Buffer to a GPU Buffer . . . 48
11.6.3.2 Transferring Output Images from a GPU Buffer to a CPU Buffer . . . 49

12 VFX SDK API Reference **51**
12.1 Structures . . . 51
12.1.1 NvVFX_Handle . . . 51
12.1.2 NvVFX_Object . . . 51
12.1.3 NvVFX_StateObjectHandle . . . 51
12.1.4 NvVFX_StateObjectHandleBase . . . 52
12.1.5 NvCVImage . . . 52
12.1.5.1 Members . . . 52
12.1.5.2 Remarks . . . 54
12.2 Enumerations . . . 54
12.3 Type Definitions . . . 54
12.3.1 NvVFX_EffectSelector . . . 54
12.3.2 NvVFX_ParameterSelector . . . 55
12.4 Video Effects Functions . . . 57
12.4.1 NvVFX_CreateEffect . . . 57

12.4.1.1 Parameters . . . 57
12.4.1.2 Return Value . . . 57
12.4.1.3 Remarks . . . 57
12.4.2 NvVFX_CudaStreamCreate . . . 58
12.4.2.1 Parameters . . . 58
12.4.2.2 Return Value . . . 58
12.4.2.3 Remarks . . . 58
12.4.3 NvVFX_CudaStreamDestroy . . . 58
12.4.3.1 Parameters . . . 58
12.4.3.2 Return Value . . . 58
12.4.3.3 Remarks . . . 58
12.4.4 NvVFX_DestroyEffect . . . 59
12.4.4.1 Parameters . . . 59
12.4.4.2 Return Value . . . 59
12.4.4.3 Remarks . . . 59
12.4.5 NvVFX_GetCudaStream . . . 59
12.4.5.1 Parameters . . . 59
12.4.5.2 Return Value . . . 59
12.4.5.3 Remarks . . . 60
12.4.6 NvCV_GetErrorStringFromCode . . . 60
12.4.6.1 Parameters . . . 60
12.4.6.2 Return Value . . . 60
12.4.6.3 Remarks . . . 60
12.4.7 NvVFX_GetF32 . . . 60
12.4.7.1 Parameters . . . 60
12.4.7.2 Return Value . . . 61
12.4.7.3 Remarks . . . 61
12.4.8 NvVFX_GetF64 . . . 61
12.4.8.1 Parameters . . . 61
12.4.8.2 Return Value . . . 61
12.4.8.3 Remarks . . . 62
12.4.9 NvVFX_GetImage . . . 62
12.4.9.1 Parameters . . . 62
12.4.9.2 Return Value . . . 62
12.4.9.3 Remarks . . . 63
12.4.10 NvVFX_GetObject . . . 63
12.4.10.1 Parameters . . . 63
12.4.10.2 Return Value . . . 63
12.4.10.3 Remarks . . . 63
12.4.11 NvVFX_GetS32 . . . 63
12.4.11.1 Parameters . . . 64
12.4.11.2 Return Value . . . 64
12.4.11.3 Remarks . . . 64
12.4.12 NvVFX_GetString . . . 64
12.4.12.1 Parameters . . . 64
12.4.12.2 Return Value . . . 65
12.4.12.3 Remarks . . . 65
12.4.13 NvVFX_GetU32 . . . 65
12.4.13.1 Parameters . . . 65
12.4.13.2 Return Value . . . 66
12.4.13.3 Remarks . . . 66
12.4.14 NvVFX_GetU64 . . . 66
12.4.14.1 Parameters . . . 66
12.4.14.2 Return Value . . . 67

12.4.14.3 Remarks . . . 67
12.4.15 NvVFX_Load . . . 67
12.4.15.1 Parameters . . . 67
12.4.15.2 Return Value . . . 67
12.4.15.3 Remarks . . . 67
12.4.16 NvVFX_Run . . . 67
12.4.16.1 Parameters . . . 67
12.4.16.2 Return Value . . . 68
12.4.16.3 Remarks . . . 68
12.4.17 NvVFX_SetCudaStream . . . 68
12.4.17.1 Parameters . . . 68
12.4.17.2 Return Value . . . 68
12.4.17.3 Remarks . . . 68
12.4.18 NvVFX_AllocateState . . . 69
12.4.18.1 Parameters . . . 69
12.4.18.2 Return Value . . . 69
12.4.18.3 Remarks . . . 69
12.4.19 NvVFX_DeallocateState . . . 69
12.4.19.1 Parameters . . . 69
12.4.19.2 Return Value . . . 70
12.4.19.3 Remarks . . . 70
12.4.20 NvVFX_ResetState . . . 70
12.4.20.1 Parameters . . . 70
12.4.20.2 Return Value . . . 70
12.4.20.3 Remarks . . . 70
12.4.21 NvVFX_SetF32 . . . 70
12.4.21.1 Parameters . . . 71
12.4.21.2 Return Value . . . 71
12.4.21.3 Remarks . . . 71
12.4.22 NvVFX_SetF64 . . . 71
12.4.22.1 Parameters . . . 71
12.4.22.2 Return Value . . . 72
12.4.22.3 Remarks . . . 72
12.4.23 NvVFX_SetImage . . . 72
12.4.23.1 Parameters . . . 72
12.4.23.2 Return Value . . . 73
12.4.23.3 Remarks . . . 73
12.4.24 NvVFX_SetObject . . . 73
12.4.24.1 Parameters . . . 73
12.4.24.2 Return Value . . . 73
12.4.24.3 Remarks . . . 73
12.4.25 NvVFX_SetStateObjectHandleArray . . . 74
12.4.25.1 Parameters . . . 74
12.4.25.2 Return Value . . . 74
12.4.25.3 Remarks . . . 74
12.4.26 NvVFX_SetS32 . . . 74
12.4.26.1 Parameters . . . 74
12.4.26.2 Return Value . . . 75
12.4.26.3 Remarks . . . 75
12.4.27 NvVFX_SetString . . . 75
12.4.27.1 Parameters . . . 75
12.4.27.2 Return Value . . . 76
12.4.27.3 Remarks . . . 76
12.4.28 NvVFX_SetU32 . . . 76

12.4.28.1 Parameters . . . 76
12.4.28.2 Return Value . . . 76
12.4.28.3 Remarks . . . 77
12.4.29 NvVFX_SetU64 . . . 77
12.4.29.1 Parameters . . . 77
12.4.29.2 Return Value . . . 77
12.4.29.3 Remarks . . . 77
12.4.30 NvVFX_ConfigureLogger . . . 77
12.4.30.1 Parameters . . . 78
12.4.30.2 Return Value . . . 79
12.4.30.3 Configuring the Logger Examples . . . 79
12.4.30.4 Advanced: Changing the Logging Level . . . 80
12.5 Return Codes . . . 81

13 Multi-GPU Overview **85**

14 Default Behavior **87**

15 Configure a Single Task **89**

16 Configure Multiple Tasks **91**

17 Using Multi-Instance GPUs **93**

18 Batch Overview **95**
18.1 What Is a Batch? . . . 95

19 Batch Utilities **97**

20 Allocation of Batched Buffers **99**

21 Select a Batch Model **101**

22 Set the Batched Images **103**
22.1 Setting the Batch Size for NvVFX_Run() . . . 104

23 Batching When Using State Variables **105**
23.1 For Webcam Denoising . . . 105
23.1.1 Example Code . . . 106
23.2 For AI Green Screen . . . 106
23.2.1 Example Code . . . 107

Download this guide as a PDF

The NVIDIA Video Effects SDK (VFX SDK) is a comprehensive collection of AI-powered video effects for real-time video enhancement and processing.

The VFX SDK enables developers to build state-of-the-art video processing applications with AI-powered features, such as video relighting, video super resolution, and AI green screen. The SDK is powered by NVIDIA graphics processing units (GPUs) with Tensor Cores, supporting high throughput and low latency processing.

The VFX SDK has the following features:

- **AI Green Screen** (video background segmentation), which segments and masks the background areas in a video or image.
- **Background Blur**, which uses the segmentation mask from the AI Green Screen filter (or other sources) and produces a blur effect in the background of a video or an image.
- **Upscale**, which is a fast and lightweight method to upscale an input video and sharpen the resulting output.
- **Webcam Denoising**, which removes noise from a webcam video while preserving the texture details.
- **Video Relighting**, which reilluminates a person in a video to match the target lighting condition. This effect uses a 360°×180° HDRI image to provide environmental lighting.
- **Video Super Resolution**, which enhances video resolution by upscaling low-resolution content to higher resolutions. This feature uses deep learning to intelligently add detail and improve visual quality beyond traditional upscaling methods.
- **Video Frame Generation**, which synthesizes intermediate frames between consecutive source frames. This feature increases effective frame rates and supports both uniform and application-selected temporal positions.
- **TrueHDR**, which converts SDR video content to HDR10 in real time. This feature uses deep learning to expand dynamic range, extend the color gamut, and remove banding artifacts.

Note

The Windows SDK supports x64 systems and Windows on Arm (ARM64) systems with a supported iGPU such as RTX Spark. The Linux SDK is designed and optimized for server-side (datacenter or cloud) deployments.

Chapter 1. Get Started on Windows

The Video Effects SDK requires specific NVIDIA GPUs and a specific version of the Windows operating system and other associated software on which the SDK depends. An external monitor is also required.

This SDK is designed and optimized for client-side application integration and for local deployment. We *do not* officially support the testing, experimentation, or deployment of this SDK to a datacenter or cloud environment.

1.1. Hardware Requirements

The Video Effects SDK is supported on NVIDIA GPUs with Tensor Cores on the following Windows architectures:

- x64 systems with supported discrete NVIDIA GPUs based on the NVIDIA Turing™, NVIDIA Ampere™, NVIDIA Ada™, or NVIDIA Blackwell™ architecture.
- Windows on Arm (ARM64) systems with a supported iGPU such as RTX Spark.

1.2. Software Requirements

The SDK requires a specific version of Windows and other associated software on which the SDK depends. The NVIDIA CUDA® and inference runtime dependencies are bundled with the SDK: TensorRT™ for x64 and TensorRT-RTX for ARM64.

Table 1-1. Software Requirements for Windows

Software	Required Version
Windows OS	x64: Windows 10 or Windows 11 ARM64: Windows 11 (Windows on Arm, such as RTX Spark)
NVIDIA Graphics Driver for Windows	x64: 570.65 or later For Tesla Compute Cluster (TCC) devices: 595 or later ARM64: 616.41 or later (Windows on Arm, such as RTX Spark)

1.2.1. Install the VFX SDK

The VFX SDK comprises the SDK Core and a set of optional features that can be downloaded and installed individually. The SDK Core and features are distributed through the NVIDIA GPU Cloud (NGC) platform.

The SDK Core includes the API headers, library files, and runtime dependencies. It does not include the libraries and models that are required to run any of the features. After installation, the SDK Core provides a script to fetch and install features from NGC.

To download the VFX SDK Core, navigate to NVIDIA Video Effects SDK Collection on the NVIDIA GPU Cloud (NGC) website, and follow the instructions in the “Getting Started” section.

The SDK is delivered as a compressed `.zip` file.

- To install the SDK Core, extract the contents of the VFX SDK Core package to your desired location.
- To use the `install_feature.ps1` script for feature installation, obtain your NGC API key by using the steps in Generating a Personal API Key in the NGC User Guide. Then use the following template commands to install a feature:

```
# Allow download script to run without any restrictions or warnings:
Set-ExecutionPolicy Bypass -Scope Process

cd /path/to/VFXSDK/features/
$Env:NGC_CLI_API_KEY = "<your-ngc-api-key>"
./install_feature.ps1 -gpu <your_GPU> -features <feature_name> -ngc_org
↪<NGC_ORG> -ngc_team <NGC_TEAM>
```

The following lists the arguments and usage for the `install_feature.ps1` script:

```
=======================================
VFX SDK Feature Installation Script
=======================================
```

(continues on next page)

(continued from previous page)

```
Usage: .\install_feature.ps1 [OPTIONS]

Options:
  -gpu <gpu>            : GPU compute capability (e.g., 75, 86, 89, 120, 121)
                          Optional - will auto-detect if not specified
  -features <features>  : Feature(s) to install (case-insensitive, default:
↪all)
                          Can be a single feature, comma-separated list, or
↪'all'
  -version <version>    : Feature version (default: latest from NGC)
                          applicable only when feature is specified.
  -ngc_org <org>        : NGC organization (default: nvidia)
  -ngc_team <team>      : NGC team (default: maxine)
  -list_features        : Query NGC for complete list of available features
  -help                 : Show this help message

Examples:
  # Install single feature (auto-detect GPU and latest compatible version):
  .\install_feature.ps1 -features nvvfxgreenscreen

  # Install single feature with specific GPU:
  .\install_feature.ps1 -gpu 89 -features nvvfxgreenscreen

  # Install single feature with specific version:
  .\install_feature.ps1 -features nvvfxvideosuperres -version 1.3.0.0

  # Install multiple features (comma-separated):
  .\install_feature.ps1 -gpu 89 -features nvvfxgreenscreen,nvvfxvideosuperres,
↪nvvfxdenoising

  # Install all features:
  .\install_feature.ps1 -gpu 89 -features all

  # Install all features on Windows on Arm (such as RTX Spark):
  .\install_feature.ps1 -gpu 121 -features all

Example VFX SDK features (non-exhaustive, case-insensitive):
  * nvvfxvideosuperres      - Video super resolution
  * nvvfxdenoising          - Video denoising
  * nvvfxupscale            - Video upscaling
  * nvvfxgreenscreen        - AI green screen
  * nvvfxbackgroundblur     - Background blur
  * nvvfxrelighting         - Relighting
  * nvvfxaigsrelighting     - AI green screen relighting

For complete list of features, use: -list_features

Supported GPU compute capabilities: 75, 86, 89, 120, 121 (Windows on Arm)
```

On Windows on Arm (for example, RTX Spark), specify `-gpu 121` (or the `rtx-spark` alias). The script installs the Windows on Arm (`woa`) feature libraries and models. If you omit `-gpu`, the script attempts to auto-detect the GPU on the target machine.

For a full list of NVIDIA GPUs and their corresponding CUDA Compute Capability versions, refer to the CUDA GPU Compute Capability table.

1.2.2. Sample Applications

To build and run the sample applications, fetch the open source repository from https://github.com/NVIDIA-Maxine/VFX-SDK-Samples and follow the instructions in `README.md`.

1.2.3. Unicode Path Support

The Video Effects SDK supports Unicode (non-ASCII) characters in all file and directory paths on Windows, including the following:

- Model directory paths that you set with `NvVFX_SetString` and the `NVVFX_MODEL_DIR` selector.
- Input and output file paths in the sample applications.
- TensorRT engine cache paths under `%LOCALAPPDATA%`, even when the Windows user profile directory contains non-ASCII characters.
- Triton model repository paths.
- Log file paths.

The SDK resolves paths that contain Chinese, Japanese, Korean, accented Latin, or other non-ASCII characters, such as the following path:

```
C:\  \  \NVIDIA_VFX_SDK\models
```

All path string parameters accept strings encoded in UTF-8 on both Windows and Linux.

Note

On Linux, the operating system handles UTF-8 paths natively. The Unicode path improvements apply mainly to Windows, where the SDK previously relied on the system ANSI code page.

1.2.3.1 Command-Line Arguments in Sample Applications

On Windows, the Video Effects SDK sample applications use wide-character entry points (`wmain`, or `GetCommandLineW` with `CommandLineToArgvW`) to receive command-line arguments. The samples convert those arguments to UTF-8 before passing them to the SDK.

The following example runs Video Super Resolution with a Unicode model path:

```
VideoSuperRes.exe --model_dir "C:\  \  \NVIDIA_VFX_SDK\models" --in_file
↪input.mp4 --out_file output.mp4
```

1.2.4. Performance Reference

This table lists latency data for features of the NVIDIA Video Effects SDK on a few supported GPU architectures. Lower values are better.

	Latency (in milliseconds)				
Feature	**RTX 2060**	**RTX 3080**	**RTX 4090**	**RTX 5090**	**RTX Spark**
Video Super Resolution, Streaming Medium (mode 21)	N/A	1.54	0.688	0.642	2.00
Video Super Resolution, Streaming Ultra (mode 23)	N/A	4.88	2.01	1.67	6.36
Video Frame Generation, Medium model, 2× frame multiplier	N/A	N/A	1.32	1.90	3.16
TrueHDR, debanding enabled	1.09	0.443	0.175	0.168	0.669
Webcam Denoising	2.37	1.01	0.472	0.409	2.32
Upscale	0.386	0.145	0.0556	0.0492	0.183
AI Green Screen	3.56	1.83	1.3	1.02	2.8
Video Relighting	40	15.6	6.76	5.69	16.4

Note

- Unless otherwise specified, effects were tested at 1280 × 720 input and output resolution.
- Video Super Resolution used 1280 × 720 input and 2560 × 1440 output (2× scaling in each dimension).
- Video Frame Generation used 1280 × 720 input and output and generated the intermediate frame at `t=0.5`.
- TrueHDR used 1280 × 720 input and output with `DebandingOff=0`, the default, which enables debanding.
- Upscale used 1280 × 720 input and 1920 × 1080 output (1.5× scaling in each dimension).
- The RTX 2060, RTX 3080, RTX 4090, and RTX 5090 columns are measured on Windows x64. The RTX Spark column is measured on Windows on Arm (ARM64).
- N/A indicates an unsupported configuration. The Video Super Resolution streaming modes are unsupported on RTX 2060, and Video Frame Generation is unsupported on RTX 2060 and RTX 3080.

Chapter 2. Get Started on Linux

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

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

2.1. Hardware Requirements

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

Note

For best performance with NVIDIA T4 and other server GPUs, ensure that you use a server that meets the thermal and airflow requirements for these types of products. For the latest list of qualified servers, refer to the Qualified System Catalog.

2.2. Software Requirements

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

Table 2-1: Software Requirements for Linux

Software	Required Version
Linux	Ubuntu 20.04, 22.04 or 24.04 Debian 11, 12 Rocky Linux 8 or 9 RHEL 8 or 9
NVIDIA Graphics Driver for Linux	570.26 or later*

*If using the Video Super Resolution or TrueHDR effect, refer to the *Video Super Resolution* or *TrueHDR* page for driver requirements.

2.2.1. Install the VFX SDK

The VFX SDK comprises the SDK Core and a set of optional features that can be downloaded and installed individually. The SDK Core and features are distributed through the NVIDIA GPU Cloud (NGC) platform.

The SDK Core includes the API headers, library files, and runtime dependencies. It does not include the libraries and models that are required to run any of the features. After installation, the SDK Core provides a script to fetch and install features from NGC.

To download the VFX SDK Core, navigate to NVIDIA Video Effects SDK Collection on the NVIDIA GPU Cloud (NGC) website, and follow the instructions in the "Getting Started" section.

The SDK is delivered as a `.tgz` package, which is a compressed tar format. Before extracting the SDK Core package, install `xz-utils` if it is not already installed. For example, on Ubuntu use the command `sudo apt install xz-utils`.

- To install the SDK Core, extract the contents of the VFX SDK Core package as follows:

```
sudo tar -xvf VFXSDK_linux_<version>.tgz -C /usr/local
```

- To use the `install_feature.sh` script for feature installation, obtain your NGC API key by using the steps in Generating a Personal API Key in the NGC User Guide. Then use the following template commands to install a feature:

```
cd /usr/local/VideoFX/features
export NGC_CLI_API_KEY=<your_API_key>
sudo --preserve-env=NGC_CLI_API_KEY ./install_feature.sh --gpu <your_GPU>
 ↪--feature <feature_name> --ngc-org <NGC_ORG> --ngc-team <NGC_TEAM>
```

Note

When running in privileged mode, omit `sudo --preserve-env=NGC_CLI_API_KEY`; however, the `NGC_CLI_API_KEY` environment variable must still be set.

The following lists the arguments and usage for the `install_feature.sh` script:

```
=====================================
VFX SDK Feature Installation Script
=====================================
Usage: install_feature.sh [OPTIONS]
Options:
  -g, --gpu <gpu>        GPU architecture (e.g., a100, b200)
  -f, --feature <list>   Feature(s) to install (case-insensitive, default:
↪all)
                        Can be a single feature, comma-separated list, or 'all
↪'
  -v, --version <ver>    Feature version (default: latest from NGC)
                        applicable when specific features are specified (not
↪'all')
```

(continues on next page)

(continued from previous page)

```
  --ngc-org <org>        NGC organization (default: nvidia)
  --ngc-team <team>      NGC team (default: maxine)
  --list-features        Query NGC for complete list of available features
  -h, --help             Show this help message

Examples:
  # Install single feature (auto-detect GPU and latest compatible version):
  sudo --preserve-env=NGC_CLI_API_KEY ./install_feature.sh -f nvvfxgreenscreen

  # Install single feature with specific GPU:
  sudo --preserve-env=NGC_CLI_API_KEY ./install_feature.sh -f
↪nvvfxgreenscreen -g a100

  # Install single feature with specific version:
  sudo --preserve-env=NGC_CLI_API_KEY ./install_feature.sh -f
↪nvvfxvideosuperres -v 1.3.0.0

  # Install multiple features (comma-separated):
  sudo --preserve-env=NGC_CLI_API_KEY ./install_feature.sh -f
↪nvvfxgreenscreen,nvvfxvideosuperres,nvvfxdenoising

  # Install all features:
  sudo --preserve-env=NGC_CLI_API_KEY ./install_feature.sh -f all

Example VFX SDK features (non-exhaustive, case-insensitive):
  • nvvfxvideosuperres      - Video super resolution
  • nvvfxdenoising          - Video denoising
  • nvvfxupscale            - Video upscaling
  • nvvfxgreenscreen        - AI green screen
  • nvvfxbackgroundblur     - Background blur

For complete list of features, use: --list-features

Note: Dependencies are installed automatically and won't be reinstalled if
↪already present

Supported GPUs: a40/l40/l4/a30/b200/a2/h100/a10/t4/b100/a16/a100/b40
```

The following table lists supported GPUs and their corresponding CUDA Compute Capability versions:

GPU	CUDA CC
t4	7.5
a100, a30	8.0
a2, a10, a16, a40	8.6
l4, l40	8.9
h100	9.0
b100, b200	10.0
b40	12.0

For a full list of NVIDIA GPUs and their corresponding CUDA Compute Capability versions, see the CUDA GPU Compute Capability table.

2.2.2. Sample Applications

To build and run the sample applications, fetch the open source repository from https://github.com/NVIDIA-Maxine/VFX-SDK-Samples and follow the instructions in `README.md`.

2.2.3. Performance Reference

This table lists latency and throughput data for features of the NVIDIA Video Effects SDK on a few supported GPU architectures. Lower latency and higher throughput are better.

	Latency (in milliseconds)				Throughput (max streams to achieve 30 FPS)			
Feature	**T4**	**A40**	**L40**	**B40**	**T4**	**A40**	**L40**	**B40**
Video Super Resolution, Streaming Medium (mode 21)	N/A	1.52	0.806	0.583	N/A	21	41	57
Video Super Resolution, Streaming Ultra (mode 23)	N/A	4.74	3.14	1.60	N/A	7	10	20
Video Frame Generation, Medium model, 2× frame multiplier	N/A	N/A	1.29	1.15	N/A	N/A	25	28
TrueHDR, debanding enabled	1.16	0.45	0.179	0.167	28	73	186	199
Webcam Denoising	2.51	1.03	0.491	0.367	13	32	67	90
Upscale	0.53	0.132	0.078	0.047	62	252	422	707
AI Green Screen	4.11	1.55	0.762	0.849	8	21	43	39
Video Relighting	50.5	15.6	7.49	5.01	0	2	4	6

Note

- Unless otherwise specified, effects were tested at 1280 × 720 input and output resolution.
- Video Super Resolution used 1280 × 720 input and 2560 × 1440 output (2× scaling in each dimension).
- Video Frame Generation used 1280 × 720 input and output and generated the intermediate frame at `t=0.5`.
- TrueHDR used 1280 × 720 input and output with `DebandingOff=0`, the default, which enables debanding.
- Upscale used 1280 × 720 input and 1920 × 1080 output (1.5× scaling in each dimension).
- Throughput values are calculated from per-call latency as the maximum integer number of streams that can achieve 30 FPS. The calculation uses the source latency before display rounding and is equivalent to `floor(1000 / (latency_ms * 30))`. These values are estimates and are not measured concurrent-stream results.
- N/A indicates an unsupported configuration or a GPU for which no benchmark result is available. The Video Super Resolution streaming modes and Video Frame Generation are unsupported on T4, and Video Frame Generation is also unsupported on A40.

2.2.4. Run SDK in a Container

The VideoFX SDK can run in an isolated container. It is easier to try the sample apps without worrying about dependencies. And the docker image can be shared easily. Two more software packages are required: Docker Engine and NVIDIA Container Toolkit.

2.2.4.1 Install the Prerequisite Software

Two software packages are required: Docker Engine and NVIDIA Container Toolkit.

2.2.4.1.1 Docker Engine

Docker Engine is an open source containerization technology for building and containerizing your applications.

To install Docker Engine on your Linux distribution, go to docs.docker.com/engine/install.

Test Docker Engine on officially supported Linux distributions, such as Ubuntu 20.04.

2.2.4.1.2 NVIDIA Container Toolkit

The NVIDIA Container Toolkit enables users to build and run GPU-accelerated containers. The toolkit includes a container runtime library and utilities to automatically configure containers to leverage NVIDIA GPUs.

To install and configure the NVIDIA Container Toolkit, go to https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html.

2.2.4.2 Fetch Sample Apps Repository

Fetch the open source repository from https://github.com/NVIDIA-Maxine/VFX-SDK-Samples to a local directory `vfx-sdk-samples`, which will be mounted when launching the container.

2.2.4.3 Build Docker Image

Download and install the SDK and all models as described in *Install the VFX SDK*. Then proceed to `vfx-sdk-samples/docker` and execute the `build_docker_image.sh` script. The building script pulls the base image (Ubuntu 20.04) from Docker Hub and installs required third-party libraries for running sample applications.

Once finished, you will see a Docker image `maxine_videofx` created on your system using the `sudo docker image ls` command.

The repository name and the TAG can be customized by modifying the `vfx-sdk-samples/docker/docker_config.sh` and rebuilding the Docker image.

2.2.4.4 Launch Container

Launch the container by executing the script `vfx-sdk-samples/docker/run_docker_image.sh`. You can choose a directory on the host machine mapped to `/host` in the container by specifying `-m <host_dir>` for file sharing purposes.

After the container is launched, you are ready to try the VideoFX SDK sample applications in `/host/vfxsdk-samples`. Only offline mode is supported inside the container. You can dump the output to `/host` directory so that you can evaluate it in the host system for playback.

See Linux-specific sample app instructions in the sample app repo's README.md.

Chapter 3. AI Green Screen

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 with chairs segmented as the foreground (Mode0)**, 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 with chairs segmented as the foreground (Mode1)**, 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.
- **Quality mode with chairs segmented as the background (Mode2)**, 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 with chairs segmented as the background (Mode3)**, 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, 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.

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 8-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. For more information about how to track the input stream, refer to *Create and Set State Variables for AI Green Screen*.

Chapter 4. Background Blur

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 steps:

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 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]`.
- The output is a 24-bit BGR image, and the data type is `UINT8`.

Chapter 5. Upscale

This is a very fast and light-weight method for upscaling an input video.

It also provides a sharpening parameter to sharpen the resulting output. 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, eight-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 needs a fast resolution increase, use Upscale.
- To increase the resolution of a video, use SuperRes with mode 1 for greater enhancement.

The following sample apps are provided for the Upscale filter:

- **VideoEffectsApp:** This app runs the SuperRes and Upscale effects individually. Use it when you want to apply a filter to the input video.
- **UpscalePipelineApp:** This app runs the Upscale effect by itself.

Chapter 6. Webcam Denoise

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.

This feature supports both 8-bit (U8) and 32-bit (F32) input and output formats.

For 32-bit (F32) Input and Output:

- The input should be provided in a GPU buffer in BGR planar format, where each pixel component is a 32-bit float. In this case, the user must provide a normalized input scaled in the [-1, 1] range.
- The output of the filter is a GPU buffer in BGR planar format, where each pixel component is a 32-bit float. The output will also be a scaled value in the [-1, 1] range.

For 8-bit (U8) Input and Output:

- The input should be provided in a GPU buffer in BGR chunky format, where each pixel component is an 8-bit unsigned integer. In this case, the feature will internally scale the input as required.
- The output of the filter is a GPU buffer in BGR chunky format, where each pixel component is an 8-bit unsigned integer.

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 to 1080p resolution range.

Note

To remove temporal noise, webcam denoising uses a state variable to track the input video stream. For more information about how to track the input stream, refer to *Create and Set State Variables for Webcam Denoising*.

Chapter 7. Video Relighting

The video relighting filter reilluminates a person in a video to match the target lighting condition. It accepts an input, matte, and HDRI image, and produces a relit output image of the foreground and an optional reprojected HDRI image for use as a background. The background pixels of the relit output are not valid and must be composited over a chosen background. The input, matte, and outputs should be the same size. The HDRI should have a 2:1 aspect ratio to avoid distortion.

- The input image is chunky BGR or RGB U8 in CUDA memory, from a video stream. The filter adjusts the lighting of the foreground portion of this image.
- The matte image is A or Y U8 in CUDA memory, produced by a grayscale segmenter like the AI Green Screen filter, and selects the foreground. Both the video relighting filter and the AI Green Screen filter receive the same input image, with the video relighting filter additionally using the AI Green Screen Filter's output.
- A High Dynamic Range Image (HDRI), specified as RGB or BGR F32, provides a lighting environment. This is an equirectangular 360°x180° projection of the world onto a sphere. Pixels of light *emitters* will be significantly brighter than those of light *reflectors*. The HDRI can be stored on the CPU or GPU, as it is immediately processed into a form suitable for lighting computations. The HDRI should have a 2:1 aspect ratio; if not, an `NVCV_ERR_WRONGSIZE` error is generated, and the image will be stretched to 2:1 to cover the entire sphere, usable for video relighting.
- If the HDRI is not 2:1 and the optional reprojected output is chosen, distortion will occur due to stretching.
- The primary output image is an RGB or BGR U8 image, typically part of a video stream, showing the relit foreground with invalid background pixels.
- There is an optional second image available, if storage is provided for it. This is the HDR image, reprojected to a plane based on the pan angle and vertical field of view (VFOV) parameters, then tone-mapped from high dynamic range to low.
- The relit foreground image needs to be subsequently composited over a background of your choice, such as:
  - The original input image, effectively relighting the foreground with a nearby light that does not affect the lighting of the faraway background.
  - The reprojected HDRI, effectively placing the foreground into that environment with compatible lighting.
  - An unrelated image.
  - Blurred versions of any of the above.

There are also several parameters that can be adjusted to achieve the desired look:

- The pan angle rotates the environment to change the lighting. It is also used to render the optional reprojected HDRI. The default is 0 radians.

- The VFOV adjusts how near or far the background appears to be when outputting the projected HDR for use as a backdrop. The default is □/3 radians (60°).
- The quality and nature of the lighting are directly controlled by the chosen HDRI. Some sample HDRIs are provided with the Sample App, including a studio light of several color temperatures.

Chapter 8. Video Super Resolution

Video Super Resolution (VSR) is an AI-powered upscaling technology that uses deep learning to enhance video resolution. Unlike traditional bicubic upscaling, VSR can reconstruct fine details and textures. In addition to upscaling, VSR also supports denoise and deblur features.

VSR provides the following modes:

Mode	Name
0	VSR_Bicubic
1	VSR_Low
2	VSR_Medium
3	VSR_High
4	VSR_Ultra
8	Denoise_Low
9	Denoise_Medium
10	Denoise_High
11	Denoise_Ultra
12	Deblur_Low
13	Deblur_Medium
14	Deblur_High
15	Deblur_Ultra
16	HighBitrate_Low
17	HighBitrate_Medium
18	HighBitrate_High
19	HighBitrate_Ultra
21	STREAMING_MEDIUM
23	STREAMING_ULTRA

Modes 5–7, 20, and 22 are reserved and must not be used.

8.1. Configuration and Requirements

VSR supports a floating-point Strength parameter in the interval `[0.0, 1.0]`. Higher values apply stronger enhancement. Set this parameter using the *NvVFX_SetF32()* function with the `NVVFX_STRENGTH` selector string before loading and running the effect.

The input and output of the VSR filter are GPU buffers. For 8-bit processing, VSR accepts BGRA or RGBA interleaved buffers, in which each pixel component is an 8-bit unsigned integer value. For 10-bit processing, VSR accepts RGB10A2 (R10G10B10A2) interleaved buffers, in which the R, G, and B components are 10-bit unsigned integer values packed with a 2-bit alpha component into a 32-bit word.

Upscaling is not supported in Denoise (modes 8–11) and Deblur (modes 12–15). For these modes, the output resolution must be the same as the input resolution.

For Windows GPUs that are Tesla Compute Cluster (TCC) devices, NVIDIA driver R595 or later is required.

On Linux, running the VSR filter requires NVIDIA driver version 570.190+, 580.82+, or 590.44+.

The recommended minimum input resolution for VSR is 360p.

The output frame aspect ratio is not restricted, but for best quality, keep it the same as the input.

8.2. Choosing a VSR Mode

Choose the mode family that matches your source and intended operation. Test representative content because quality and processing speed depend on the source, resolution, and GPU.

8.2.1. Standard Upscaling Modes (0–4)

Use modes 0–4 when the output resolution is larger than the input resolution and the source is typical compressed video. Modes 1–4 use AI to reduce compression artifacts while upscaling.

Mode	Name	When to use
0	`VSR_Bicubic`	Fastest, non-AI resize or comparison baseline.
1	`VSR_Low`	AI upscaling that prioritizes speed.
2	`VSR_Medium`	Balanced speed and output quality.
3	`VSR_High`	AI upscaling that prioritizes output quality.
4	`VSR_Ultra`	Maximum detail preservation when the processing budget allows.

If you are unsure which AI mode to select, start with `VSR_High`. Choose a lower mode for more speed or `VSR_Ultra` for maximum quality.

8.2.2. Denoise Modes (8–11)

Use lower modes to preserve more texture and higher modes for stronger noise removal. Higher modes can soften fine detail.

The following are typical use cases of the denoise mode:

- Low-light footage from consumer devices.
- Archived content with film grain or analog artifacts.
- Pre-encoding optimization to reduce bitrate overhead.

We do *not* recommend using denoise mode in the following cases:

- Extreme noise levels obscuring underlying detail.
- Structured compression artifacts (such as banding or blocking).
- Content with intentional cinematic grain.

8.2.3. Deblur Modes (12–15)

Use lower modes for light sharpening and higher modes for stronger sharpening.

The following are typical use cases of the deblur mode:

- Low-to-moderate QP encoded video with visible softness but intact structure.
- Pre-processing for super-resolution pipelines.
- Consumer camera footage with focus or lens softness issues.
- Digitized archival content with inherent optical blur.

We do *not* recommend using deblur mode in the following cases:

- Severe motion blur or artistic bokeh effects.
- Noisy or heavily compressed inputs in which deblurring amplifies artifacts.

8.2.4. High-Bitrate Modes (16–19)

Within this family, lower modes prioritize speed and higher modes prioritize output quality.

The following are typical use cases of the high-bitrate mode:

- High-bitrate natural and gaming video with minimal compression artifacts.
- Detail restoration after downscaling or quality enhancement in pre-encode workflows.

We do *not* recommend using high-bitrate mode for heavily compressed or severely blurred content in which the underlying information is irrecoverable.

8.2.5. Streaming Modes (21 and 23)

Note

`STREAMING_MEDIUM` and `STREAMING_ULTRA` require an NVIDIA Ampere or later architecture GPU and are not supported on NVIDIA Turing architecture GPUs.

The following are typical use cases of the streaming mode:

- Clean or lightly degraded source content that requires resolution enhancement before re-encoding.
- Live or VOD pipelines targeting higher-resolution delivery, such as SD-to-HD or HD-to-4K ladders.

We do *not* recommend using streaming modes in the following cases:

- Workflows in which upscaling is applied post-encoding, because the model is optimized for pre-transcode use.
- Content with extreme noise or structured blocking artifacts that might be amplified during upscaling.

Expected quality: The streaming modes produce crisper edges and more natural high-frequency detail than bicubic or Lanczos interpolation. They deliver texturally richer output than purely interpolation-based methods while maintaining visual stability without hallucinated or unstable artifacts.

Use `STREAMING_MEDIUM` for balanced quality and performance. Use `STREAMING_ULTRA` for stronger enhancement when the target GPU supports it.

Chapter 9. Video Frame Generation

Video Frame Generation (VFG) is an AI-powered temporal interpolation filter that synthesizes intermediate frames between two consecutive source frames. The previous frame represents time `t=0` and the current frame represents time `t=1`. Each call to `NvVFX_Run()` produces one generated frame at a requested temporal position strictly between the two inputs.

VFG does not perform spatial scaling. The previous frame, current frame, and generated frame must have the same dimensions and use the same encoding family.

9.1. Operating Modes

VFG provides two mutually exclusive ways to select the temporal position of the generated frame:

Mode	Configuration	Generated Position
Multiplier	Set `FrameMultiplier` to `M` in the range 2 through 8, and set `FrameIndex` to `i` in the range 1 through `M-1` before each run.	`t = i / M`
Explicit timestep	Set `FrameMultiplier` to 0, and set `Timestep` to a value in the open interval `(0.0, 1.0)` before each run.	`t = Timestep`

For example, multiplier mode with `M=4` requires three calls to `NvVFX_Run()`, using frame indices 1, 2, and 3. The calls generate frames at `t=0.25`, `t=0.5`, and `t=0.75`. VFG writes each result to the same output buffer, so the application must consume or copy the result before the next run.

Setting `FrameMultiplier`, including setting it to 0, resets `FrameIndex` and `Timestep`. Set a new per-output selector before the next run.

9.2. Model Modes

VFG provides the following model modes:

Value	Mode	Description
0	Low	Selects the lowest-complexity model.
1	Medium	Selects the balanced model. This is the default.
2	High	Selects the highest-complexity model.

Set the model mode before calling `NvVFX_Load()`. Changing it after loading the effect requires another call to `NvVFX_Load()`.

9.3. Parameters

VFG provides the following feature-specific parameters:

Parameter	Type	Default	Description
`NVVFXVIDEOFRAMEGENERATION_MODE`	U32	1	Selects the Low (0), Medium (1), or High (2) model.
`NVVFXVIDEOFRAMEGENERATION_FRAME_MULTIPLIER`	U32	0	Selects explicit timestep mode (0) or multiplier mode (2-8).
`NVVFXVIDEOFRAMEGENERATION_FRAME_INDEX`	U32	Unset	Selects an intermediate frame in the range 1 through `M-1` when multiplier mode is active.
`NVVFXVIDEOFRAMEGENERATION_TIMESTEP`	F32	Unset	Selects a temporal position in `(0.0, 1.0)` when explicit timestep mode is active.
`NVVFXVIDEOFRAMEGENERATION_SHOT_CHANGE`	U32	0	Set to 1 when the two inputs cross a shot change or other discontinuity. VFG bypasses interpolation and copies the current frame to the output. The value persists until the application sets it to 0.
`NVVFXVIDEOFRAMEGENERATION_AUTOMATIC_SHOT_CHANGE_DE`	U32	1	Enables (1) or disables (0) automatic shot-change detection. A manual shot-change value of 1 takes priority.

Set `NVVFX_CUDA_STREAM`, `NVVFX_INPUT_WIDTH`, `NVVFX_INPUT_HEIGHT`, and the model mode before calling `NvVFX_Load()`. Only a batch size of 1 is supported.

9.4. Input and Output Requirements

Bind the previous frame with `NVVFX_INPUT_IMAGE_0`, the current frame with `NVVFX_INPUT_IMAGE_1`, and the generated frame with `NVVFX_OUTPUT_IMAGE`.

VFG supports the following image encodings:

- RGBA or BGRA interleaved images with 8-bit unsigned components.
- RGB10A2 interleaved images packed in a 32-bit component.

All three images must use either the 8-bit encoding family or RGB10A2; mixing the two families is not supported. The images can reside in CUDA device memory (`NVCV_CUDA` or `NVCV_GPU`) or mapped pinned-host memory (`NVCV_CPU_PINNED`).

9.5. Hardware Requirements

VFG supports these platforms and GPUs:

- Windows and Linux x86-64: NVIDIA Ada and Blackwell architecture GPUs.
- Linux x86-64 additionally supports NVIDIA Hopper architecture GPUs.
- Windows ARM64: NVIDIA RTX Spark.

9.6. Recommended Use Cases

The following are typical use cases for VFG:

- Increasing the frame rate of video for smoother playback or streaming.
- Creating slow-motion video by generating intermediate frames while retaining the original playback frame rate.
- Frame-rate conversion and video post-processing.

Chapter 10. TrueHDR

TrueHDR is an AI-powered SDR-to-HDR conversion filter. It accepts an 8-bit-per-channel SDR RGB frame and produces a 10-bit-per-channel HDR frame with expanded dynamic range, wider color gamut, and a deep-learning debanding pass that removes the contouring artifacts inherent to converting 8-bit sources to a higher bit depth. The output frame is packed HDR10—R10G10B10A2 in the BT.2020 color primaries with the SMPTE ST 2084 (PQ) transfer function—which is the same on-the-wire format used by HDR10-capable displays.

TrueHDR provides the following tunables:

Parameter	Min	Max	Default	Description
Contrast	0	200	100	Adjusts the difference between highlights and shadows.
Saturation	0	200	100	Adjusts color intensity.
MiddleGray	10	100	50	Adjusts the average brightness (middle-gray reference).
Luminance	400	2000	650	Sets the target HDR display peak luminance, in nits.
DebandingOff	0	1	0	Set to 1 to skip the deep-learning debanding pass for higher throughput on already clean sources.

The input of the TrueHDR filter is a GPU buffer in RGBA interleaved format, in which each pixel component is an 8-bit unsigned integer value.

The output is a GPU buffer in RGB10A2 interleaved format, in which each of the R, G, and B components is a 10-bit unsigned integer packed together with a 2-bit alpha in a 32-bit word.

For details about allocating and converting `NVCV_RGB10A2` buffers, refer to Working with 10-Bit RGBA (R10G10B10A2) Buffers in the *NvCVImage API Guide*.

The output frame dimensions match the input; TrueHDR does not perform spatial scaling.

For Linux, NVIDIA driver 570 or later is required.

10.1. Recommended Use Cases

The following are typical use cases for TrueHDR:

- Real-time up-conversion of SDR video for playback on HDR10-capable displays.
- Cloud transcoding pipelines that produce HDR10 outputs from SDR sources.
- Streaming services that upgrade legacy SDR content for HDR-capable clients.

Chapter 11. VFX SDK API Architecture

11.1. Use a Video Effect Filter

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

11.1.1. Creating a Video Effect Filter

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

- The *NvVFX_EffectSelector* type.
- 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.

To create an instance of the AI green screen feature type, at the top of your source file, include the feature header file:

```
#include "nvVFXGreenScreen.h"
```

To create the video effect filter instance:

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

Each *NvVFX_EffectSelector* identifier is defined under the respective feature header, such as `nvVFXGreenScreen.h`. Ensure that the requested feature is installed so that the SDK has access to both feature libraries and required model files.

Note that during *NvVFX_CreateEffect()*, the SDK loads the feature library from the location `<vfx-sdk>/features/<feature>/<bin/lib>`. Without a feature installation, the call to this API function returns an error code.

For instructions on how to install features, refer to the installation guide.

11.1.2. 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 = R"(C:\Users\vfxuser\Documents\vfx\models)";

...

vfxErr = NvVFX_SetString(effectHandle, NVVFX_MODEL_DIRECTORY, modelDir);
```

On Windows on Arm (for example, RTX Spark), you can optionally configure the model runtime cache before calling `NvVFX_Load()` for effects that support it (AI Green Screen, Webcam Denoising, and Video Relighting). An empty cache directory selects the default `cache` directory under `ModelDir`:

```
unsigned int modelCacheMode = 0;  // Auto
vfxErr = NvVFX_SetU32(effectHandle, NVVFX_MODEL_CACHE_MODE, modelCacheMode);

const char* modelCacheDir = "";  // default: ModelDir/cache
vfxErr = NvVFX_SetString(effectHandle, NVVFX_MODEL_CACHE_DIRECTORY,
                         modelCacheDir);
```

11.1.3. Setting Up the CUDA Stream

A video effect filter requires an NVIDIA 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()`.
   - 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);
```

11.2. Create and Set State Variables

11.2.1. 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, stateSizeInBytes);
     ```

- 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 or video.

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

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

11.2.1.1 For AI Green Screen

The AI green screen uses a state variable to track the input video stream 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.

  Allocate a state variable by calling the *NvVFX_AllocateState* function, which returns `NvCV_Status`.

The function completes the following steps:

1. Allocates the state variable.
2. Resets the state variable.
3. 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 or video.

  Before you can use it on a new input, reset the state variable by calling the *NvVFX_ResetState* function, which 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

For more information about how to track multiple input streams concurrently, refer to *Batching When Using State Variables for AI Green Screen*.

11.2.2. 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. For more information about NvCVImage, refer to the NvCVImage API Guide. 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
- VideoSuperRes: BGRA/RGBA u8 chunky BGRA/RGBA u8 chunky, or RGB10A2 P32 chunky RGB10A2 P32 chunky
- Video Frame Generation: BGRA/RGBA u8 chunky BGRA/RGBA u8 chunky, or RGB10A2 P32 chunky RGB10A2 P32 chunky
- TrueHDR: RGBA u8 chunky RGB10A2 P32 chunky
- 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 bits. RGB10A2 P32 chunky refers to a packed 32-bit pixel with 10-bit unsigned integer R, G, and B components and a 2-bit alpha component. 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, because 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 dstGpuImage;

...
```

(continues on next page)

(continued from previous page)

```
vfxErr = NvCVImage_Alloc(&dstGpuImage, 960, 540, NVCV_A, NVCV_U8,
NVCV_CHUNKY, NVCV_GPU, 1);

...

vfxErr = NvVFX_SetImage(effectHandle, NVVFX_OUTPUT_IMAGE, &dstGpuImage);
```

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

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

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

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

11.2.3.1 Example: Setting the Filter Mode for AI Green Screen

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

- **Quality mode with chairs segmented as the foreground (Mode0)**, which provides the highest quality result (default).
- **Performance mode with chairs segmented as the foreground (Mode1)**, which provides the fastest performance.
- **Quality mode with chairs segmented as the background (Mode2)**, which provides the highest quality result (default)
- **Performance mode with chairs segmented as the background (Mode3)**, which provides the fastest performance

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*).
- The selector string `NVVFX_MODE`.
- An integer that denotes the mode of operation that you want:
  - `0`: Quality mode with chairs in the foreground
  - 1: Performance mode with chairs in the foreground
  - 2: Quality mode with chairs in the background
  - 3: Performance mode with chairs in the background

This example sets the mode to Performance with chairs in the foreground:

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

11.2.3.2 Example: Enable CUDA Graph Optimization for AI Green Screen

The AI green screen filter supports CUDA graph optimization. It can be enabled by setting the correct value for NVVFX_CUDA_GRAPH.

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_CUDA_GRAPH.
- An integer that denotes the CUDA graph optimization state:
  - 0: OFF
  - 1: ON
- The default state of CUDA Graph optimization is OFF for AI green screen filter:

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

  The set method must be called before the NvVFX_Load() method that initializes the AIGS effect.
- The first call to NvVFX_Run() initializes CUDA Graph if it is enabled. If there is an error during graph initialization, the call returns NVCV_ERR_CUDA.
- If the set method is called after the initialization, the NvVFX_Run() call returns the NVCV_ERR_INITIALIZATION error code.

Enabling the 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 turned **off** and **on**.

11.2.3.3 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 and 1080.

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

- The set method 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.

11.2.4. Summary of Video Effects SDK Accessor Functions

Table 2-1. Summary of NVIDIA Video Effects SDK Accessor Functions

Parameter Type	Data Type	Set and Get Accessor Function
32-bit unsigned integer	unsigned int	NvVFX_SetU32() NvVFX_GetU32()
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()

11.2.5. Getting Information About a Filter and Its Parameters

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

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

11.2.6. Getting a List of All Available Effects

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

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

11.3. Load 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.

11.4. Run 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()
```

11.5. Destroy 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++.

11.6. Work 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 SDK provides functions for converting an image representation to NvCVImage and transferring images between CPU and GPU buffers.

On Windows on Arm iGPU systems (for example, RTX Spark), the CPU and GPU share memory. Allocating input and output `NvCVImage` buffers with `NVCV_CPU_PINNED` (page-locked host memory) can avoid unnecessary host□device copies and improve steady-state transfer performance. Pinned memory is not a VFX SDK configuration parameter; it is chosen when you allocate the image buffer. Sample applications such as `AigsEffectApp` and `RelightingEffectApp` expose this as `--use_pinned_memory`.

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.

11.6.1. 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.

11.6.1.1 Converting OpenCV Images to NvCVImage Objects

Note

Use the wrapper functions that the 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(...);
```

(continues on next page)

(continued from previous page)

```
NvCVImage dstCPUImg(...);

//Declare source and destination OpenCV images
cv::Mat srcCVImg;
cv::Mat dstCVImg;
CVWrapperForNvCVImage (&srcCPUImg, &srcCVImg);
CVWrapperForNvCVImage (&dstCPUImg, &dstCVImg);
```

11.6.1.2 Converting Image Frames on GPU or CPU Buffers to NvCVImage Objects

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

```
NvCVImage src_gpu;

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

NvCVImage src_cpu;

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

11.6.1.3 Converting Decoded Frames from the NvDecoder to NvCVImage Objects

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

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

//Initialize decoder...
//Assuming dec.GetOutputFormat() == cudaVideoSurfaceFormat_NV12
//Initialize memory for decoded frame

NvCVImage_Init(&decoded_frame, dec.GetWidth(), dec.GetHeight(),
dec.GetDeviceFramePitch(), NULL, NVCV_YUV420, NVCV_U8, NVCV_NV12,
NVCV_GPU, 1);

decoded_frame.colorSpace = NVCV_709 \| NVCV_VIDEO_RANGE \| NVCV_CHROMA\_
COSITED;

//Allocate memory for BGRA frame

NvCVImage_Alloc(&BGRA_frame, dec.GetWidth(), dec.GetHeight(), NVCV_BGRA,
NVCV_U8, NVCV_CHUNKY, NVCV_GPU, 1);

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

(continues on next page)

(continued from previous page)

```
//Convert from decoded frame format(NV12) to desired format(BGRA)

NvCVImage_Transfer(&decoded_frame, &BGRA_frame, 1.f, stream, &
stagingBuffer);
```

Note

The preceding sample 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` and `601`. - If they are washed out or blown out, swap `VIDEO` and `FULL`. - If the colors are shifted horizontally, swap `INTSTITIAL` and `COSITED`.

11.6.1.4 Converting an NvCVImage Object to a Buffer Encodable by NvEncoder

To convert the `NvCVImage` object to the pixel format that is used during encoding via NvEncoder, if needed, call the `NvCVImage_Transfer()` function. The following example 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 earlier

NvEncoderCuda::CopyToDeviceFrame(cuContext,
    BGRA_frame.pixels,
    BGRA_frame.pitch,
    (CUdeviceptr)encoderInputFrame->inputPtr,
    encoderInputFrame->pitch,
    pEnc->GetEncodeWidth(),
    pEnc->GetEncodeHeight(),
```

(continues on next page)

(continued from previous page)

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

11.6.2. Allocating an NvCVImage Object Buffer

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

11.6.2.1 Using the NvCVImage Allocation Constructor to Allocate a Buffer

The NvCVImage allocation constructor creates an object to which memory has been allocated and that has been initialized. 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);
  ```

11.6.2.2 Using Image Functions to Allocate a Buffer

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

1. Declare an empty `NvCVImage` object:

```
NvCVImage xfr;
```

2. Allocate or reallocate the buffer for the image.
   - To allocate the buffer, call the NvCVImage_Alloc() function. Allocate a buffer this way when the image is part of a state structure, where you will not know the size 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()`.

11.6.3. Transferring Images Between CPU and GPU Buffers

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

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

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

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

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

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

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

```
NvCVImage stageImg;
```

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

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

```
// Transfer the image from the CPU to the GPU, perhaps with conversion.

NvCVImage_Transfer(&srcCpuImg, &dstGpuImg, 1.0f, stream, &stageImg);
```

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

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

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

```
NvCVImage srcGpuImg(width, height, NVCV_BGR, NVCV_F32, NVCV_PLANAR, NVCV_GPU,
↪1);

NvCVImage dstCpuImg(width, height, NVCV_BGR, NVCV_U8, NVCV_INTERLEAVED, NVCV_
↪CPU, 1);
```

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

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

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

     ```
     NvCVImage stageImg;
     ```

   An appropriately sized buffer is allocated or reallocated if needed. For more information about NvCVImage, refer to 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 12. VFX SDK API Reference

12.1. Structures

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

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

12.1.1. NvVFX_Handle

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

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

Defined in: `nvVideoEffects.h`.

12.1.2. NvVFX_Object

```
struct NvVFX_Object;
```

This is 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`.

12.1.3. NvVFX_StateObjectHandle

```
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`.

12.1.4. NvVFX_StateObjectHandleBase

`struct NvVFX_StateObjectHandleBase;`

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

Defined in: `nvVideoEffects.h`.

12.1.5. NvCVImage

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;
```

12.1.5.1 Members

width

Type: `unsigned int`

The width, in pixels, of the image.

height

Type: `unsigned int`

The height, in pixels, of the image.

pitch

Type: `unsigned int`

The vertical byte stride between pixels.

pixelFormat

Type: `NvCVImage_PixelFormat`

The format of the pixels in the image.

componentType

Type: `NvCVImage_ComponentType`

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

pixelBytes

Type: `unsigned char`

The number of bytes in a chunky pixel.

componentBytes

Type: `unsigned char`

The number of bytes in each pixel component.

numComponents

Type: `unsigned char`

The number of components in each pixel.

planar

Type: `unsigned char`

Specifies the organization of the pixels in the image.

- 0: Chunky
- 1: Planar

gpuMem

Type: `unsigned char`

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

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

colorspace

Type: `unsigned char`

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

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

For more information about the type definitions, refer to YUV Color Spaces.

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.

12.1.5.2 Remarks

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

Defined in: `nvCVImage.h`.

12.2. Enumerations

The enumerations in the 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 Enumerations in the NvCVImage API Guide.

12.3. Type Definitions

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

12.3.1. NvVFX_EffectSelector

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

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

`NVVFX_FX_TRANSFER` “Transfer”

Image transfer effect.

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

`NVVFX_FX_GREEN_SCREEN` “Green Screen”

AI green screen filter.

`NVVFX_FX_BGBLUR` “Background Blur”

Background Blur filter.

`NVVFX_FX_SUPER_RES` “Video Super Res”

AI-based video super resolution.

`NVVFX_FX_SR_UPSCALE` “Upscale”

AI-based fast video upscaler.

`NVVFX_FX_DENOISING` “Denoising”

Webcam Denoising filter.

`NVVFX_FX_VIDEO_FRAME_GENERATION` “VideoFrameGeneration”

AI-based temporal frame interpolation filter. For details about its operating modes and parameters, refer to *Video Frame Generation*.

`NVVFX_FX_TRUE_HDR` “TrueHDR”

AI-based SDR-to-HDR conversion filter. For details of the tunables that this filter accepts, refer to *TrueHDR*.

12.3.2. NvVFX_ParameterSelector

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

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

`NVVFX_INPUT_IMAGE_0` “SrcImage0”

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

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

`NVVFX_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_MODEL_CACHE_DIRECTORY` “ModelCacheDir”

(Windows on Arm only, such as RTX Spark) Optional path to the model runtime cache directory. Set this parameter before calling `NvVFX_Load()`. If it is empty or is not set, the SDK uses a `cache` subdirectory under `ModelDir`. Applicable to AI Green Screen, Webcam Denoising, and Video Relighting.

`NVVFX_MODEL_CACHE_MODE` “ModelCacheMode”

(Windows on Arm only, such as RTX Spark) Optional unsigned integer that controls the model runtime cache. Set this parameter before calling `NvVFX_Load()`:

- `0`: Auto (default). Load a valid cache if present; otherwise run JIT and save a new cache.
- 1: Disabled. Do not read or write cache files.
- 2: Force regenerate. Ignore any existing cache, run JIT, and write a fresh cache.

On the first load of each model without a valid cache, JIT compilation can require on the order of several seconds. Later loads reuse the cache when the mode is Auto. Applicable to AI Green Screen, Webcam Denoising, and Video Relighting.

`NVVFX_CUDA_STREAM` "CudaStream"

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

`NVVFX_CUDA_GRAPH` "CudaGraph"

Enable CUDA Graph optimization to reduce GPU operation submission overhead. Enabling this flag is recommended on Windows on Arm (for example, RTX Spark) for better steady-state performance where the effect supports it.

`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. For Video Super Resolution, use a floating-point value in the interval `[0.0, 1.0]` to control the intensity of the selected mode.

`NVVFX_STRENGTH_LEVELS` "StrengthLevels"

Number of unique strength levels in the interval `[0, 1]`. Currently this applies only 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 selector is used by filters that expose multiple operating modes. For Video Super Resolution mode values, refer to *Video Super Resolution*.

For AI Green Screen, the values are as follows:

- 0: Quality mode with chairs segmented as the foreground
- 1: Performance mode with chairs segmented as the foreground
- 2: Quality mode with chairs segmented as the background
- 3: Performance mode with chairs segmented as the background

`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.

12.4. 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++.

12.4.1. NvVFX_CreateEffect

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

12.4.1.1 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.

12.4.1.2 Return Value

- `NVCV_SUCCESS` on success.

12.4.1.3 Remarks

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

12.4.2. NvVFX_CudaStreamCreate

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

12.4.2.1 Parameters

stream [out]

Type: `CUstream *`

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

12.4.2.2 Return Value

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

12.4.2.3 Remarks

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

12.4.3. NvVFX_CudaStreamDestroy

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

12.4.3.1 Parameters

stream [in]

Type: `CUstream`

The CUDA stream to destroy.

12.4.3.2 Return Value

Does not return a value.

12.4.3.3 Remarks

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

12.4.4. NvVFX_DestroyEffect

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

12.4.4.1 Parameters

obj [in]

Type: `NvVFX_Handle`

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

12.4.4.2 Return Value

Does not return a value.

12.4.4.3 Remarks

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

12.4.5. NvVFX_GetCudaStream

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

12.4.5.1 Parameters

obj

Type: `NvVFX_Handle`

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

paramName

Type: `NvVFX_ParameterSelector`

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

stream

Type: `CUstream *`

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

12.4.5.2 Return Value

- `NVCV_SUCCESS` on success.

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

12.4.5.3 Remarks

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

12.4.6. NvCV_GetErrorStringFromCode

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

12.4.6.1 Parameters

code

Type: NvCV_Status

The NvCV_Status code for which to get an error string.

12.4.6.2 Return Value

The error string that corresponds to the specified error code.

12.4.6.3 Remarks

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

12.4.7. NvVFX_GetF32

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

12.4.7.1 Parameters

obj

Type: NvVFX_Handle

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

paramName

Type: NvVFX_ParameterSelector

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

val

Type: `float *`

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

12.4.7.2 Return Value

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

12.4.7.3 Remarks

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

12.4.8. NvVFX_GetF64

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

12.4.8.1 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.

12.4.8.2 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.

12.4.8.3 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.

12.4.9. NvVFX_GetImage

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

12.4.9.1 Parameters

obj

Type: NvVFX_Handle

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

paramName

Type: NvVFX_ParameterSelector

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

- NVVFX_INPUT_IMAGE_0
- NVVFX_INPUT_IMAGE
- NVVFX_OUTPUT_IMAGE_0
- NVVFX_OUTPUT_IMAGE

Any other selector string returns an error.

im

Type: NvCVImage *

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

12.4.9.2 Return Value

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

12.4.9.3 Remarks

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

12.4.10. NvVFX_GetObject

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

12.4.10.1 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.

12.4.10.2 Return Value

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

12.4.10.3 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.

12.4.11. NvVFX_GetS32

```
NvVFX_Status NvVFX_GetS32(
  NvVFX_Handle obj,
  NvVFX_ParameterSelector paramName,
  int \*val
);
```

12.4.11.1 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.

12.4.11.2 Return Value

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

12.4.11.3 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.

12.4.12. NvVFX_GetString

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

12.4.12.1 Parameters

obj

Type: NvVFX_Handle

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

paramName

Type: NvVFX_ParameterSelector

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

- NVVFX_INFO
- NVVFX_MODEL_DIRECTORY

Any other selector string returns an error.

str

Type: const char **

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

12.4.12.2 Return Value

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

12.4.12.3 Remarks

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

12.4.13. NvVFX_GetU32

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

12.4.13.1 Parameters

obj

Type: NvVFX_Handle

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

paramName

Type: NvVFX_ParameterSelector

The NVVFX_MODE, NVVFX_STRENGTH, NVVFX_TEMPORAL, NVVFX_GPU, 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.

12.4.13.2 Return Value

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

12.4.13.3 Remarks

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

12.4.14. NvVFX_GetU64

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

12.4.14.1 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.

12.4.14.2 Return Value

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

12.4.14.3 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.

12.4.15. NvVFX_Load

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

12.4.15.1 Parameters

obj [in]

Type: NvVFX_Handle

The handle to the video effect filter instance to load.

12.4.15.2 Return Value

NVCV_SUCCESS on success.

12.4.15.3 Remarks

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

12.4.16. NvVFX_Run

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

12.4.16.1 Parameters

obj [in]

Type: NvVFX_Handle

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

async [in]

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

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

12.4.16.2 Return Value

`NVCV_SUCCESS` on success.

12.4.16.3 Remarks

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

12.4.17. NvVFX_SetCudaStream

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

12.4.17.1 Parameters

obj

Type: `NvVFX_Handle`

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

paramName

Type: `NvVFX_ParameterSelector`

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

stream

Type: `CUstream`

The CUDA stream to which the parameter will be set.

12.4.17.2 Return Value

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

12.4.17.3 Remarks

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

12.4.18. NvVFX_AllocateState

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

12.4.18.1 Parameters

obj

Type: NvVFX_Handle

The handle to the video effect filter instance for 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.

12.4.18.2 Return Value

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

12.4.18.3 Remarks

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

12.4.19. NvVFX_DeallocateState

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

12.4.19.1 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 a state variable.

12.4.19.2 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.

12.4.19.3 Remarks

This function deallocates a state variable.

12.4.20. NvVFX_ResetState

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

12.4.20.1 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 a state variable.

12.4.20.2 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.

12.4.20.3 Remarks

This function resets a state variable.

12.4.21. NvVFX_SetF32

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

12.4.21.1 Parameters

obj

Type: NvVFX_Handle

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

paramName

Type: NvVFX_ParameterSelector

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

val

Type: float

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

12.4.21.2 Return Value

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

12.4.21.3 Remarks

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

12.4.22. NvVFX_SetF64

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

12.4.22.1 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.

12.4.22.2 Return Value

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

12.4.22.3 Remarks

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

12.4.23. NvVFX_SetImage

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

12.4.23.1 Parameters

obj

Type: NvVFX_Handle

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

paramName

Type: NvVFX_ParameterSelector

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

- NVVFX_INPUT_IMAGE_0
- NVVFX_INPUT_IMAGE
- NVVFX_OUTPUT_IMAGE_0
- NVVFX_OUTPUT_IMAGE

Any other selector string returns an error.

im

Type: NvCVImage *

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

12.4.23.2 Return Value

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

12.4.23.3 Remarks

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

12.4.24. NvVFX_SetObject

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

12.4.24.1 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.

12.4.24.2 Return Value

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

12.4.24.3 Remarks

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

12.4.25. NvVFX_SetStateObjectHandleArray

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

12.4.25.1 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`.

handle

Type: `NvVFX_StateObjectHandle`

Pointer to the array of state objects.

12.4.25.2 Return Value

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

12.4.25.3 Remarks

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

12.4.26. NvVFX_SetS32

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

12.4.26.1 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.

12.4.26.2 Return Value

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

12.4.26.3 Remarks

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

12.4.27. NvVFX_SetString

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

);

12.4.27.1 Parameters

obj

Type: `NvVFX_Handle`

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

paramName

Type: `NvVFX_ParameterSelector`

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

str

Type: `const char *`

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

12.4.27.2 Return Value

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

12.4.27.3 Remarks

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

12.4.28. NvVFX_SetU32

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

12.4.28.1 Parameters

obj

Type: NvVFX_Handle

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

paramName

Type: NvVFX_ParameterSelector

The NVVFX_MODE, NVVFX_STRENGTH, NVVFX_TEMPORAL, NVVFX_GPU, 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.

12.4.28.2 Return Value

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

12.4.28.3 Remarks

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

12.4.29. NvVFX_SetU64

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

12.4.29.1 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.

12.4.29.2 Return Value

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

12.4.29.3 Remarks

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

12.4.30. NvVFX_ConfigureLogger

```
NvCV_Status NvVFX_ConfigureLogger(int verbosity, const char *file,
            void (*cb)(void*, const char*), void *cb_data);
```

Several specific error codes help to identify the nature of an error. Sometimes more information is needed about specific actions to take to overcome the error during integration into his application. That is the purpose of the logging functions.

But not all applications have an error console, so is it necessary to configure the logger to work in the environment of the application.

All logger instantiations are implemented as callbacks. We have supplied source code examples of five logger instantiations in `nvCVLoggerExamples.c`:

- MemLogger: Collects logs into a string in memory.
- StderrLogger: Log to stderr.
- FileLogger: Log to a file.
- FileThreadLogger: Log to a file, with file writes performed in a separate thread to reduce the latency of the FileLogger.
- Multifile Logger: Log to multiple files, each no larger than a specified size.

These are just examples of logger instantiations. You can of course develop your own logger to better suit the nature of your application.

StderrLogger and FileLogger are compiled into the SDK and can be easily configured:

```
NvVFX_ConfigureLogger(LOG_ERROR, "stderr", NULL, NULL);

NvVFX_ConfigureLogger(LOG_ERROR, "/logs/vfx.log", NULL, NULL);
```

Logging can be turned off by supplying `NULL`:

```
NvVFX_ConfigureLogger(LOG_ERROR, NULL, NULL, NULL);
```

12.4.30.1 Parameters

verbosity

Type: `int`

The maximum level of verbosity to be recorded into the log.

`NVCV_LOG_FATAL` (0)

Message to be printed right before terminating due to an unrecoverable error.

`NVCV_LOG_ERROR` (1)

An operation has failed, but it is not fatal.

`NVCV_LOG_WARNING` (2)

Something was not quite right, but we fixed it up, perhaps at a loss in performance.

`NVCV_LOG_INFO` (3)

Nothing is wrong, but this information might be of interest.

The higher level verbosity also includes the logs from the lower levels of verbosity.

file

Type: `const char*`

This field is used only when logging to stderr or to a file. Otherwise it should be set to `NULL`. Examples:

- "`stderr`": Log to stderr.
- "`/path/to/log.txt`": Log to the file named `/path/to/log.txt`.

cb

Type: `void (*)(void *data, const char *msg)`

Pointer to the callback function. The function has two arguments:

- data: Pointer to data used by the callback function.
- msg: The message to be entered into the log.

cb_data

Type: `void*`

Pointer provided to the callback function as its first argument.

12.4.30.2 Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_FILE` on failure to open the specified log file.

12.4.30.3 Configuring the Logger Examples

Source code for five example loggers is provided in the SDK, and this section shows how to configure them. Though the built-in loggers StderrLogger and FileLogger require one less line of code to configure them, there are cases where the other examples are more appropriate.

- Servers typically run for weeks or months without being restarted. Using a standard file logger, the file would grow without bound. That is where the multi-file logger is useful: to limit the size of the log.
- Coalescing logs from several SDKs cannot be done with the built-in file logger, because each SDK has its own logger. It is necessary, therefore, to instantiate a single logger in the application and then use that instance to configure the logger in each SDK.
- To minimize the performance impact of copious logging (as when specifying `NVCV_LOG_INFO`), the FileThreadLogger is preferred to minimize impact on performance. If the standard FileLogger is used instead, the program will stall while writing to the file.

12.4.30.3.1 MemLogger

```
MemLogger memLog;
NvVFX_ConfigureLogger(LOG_ERROR, NULL, &MemLogger::Callback, &memLog);
```

12.4.30.3.2 StderrLogger

```
StderrLogger stdLog;
NvVFX_ConfigureLogger(LOG_ERROR, NULL, &StderrLogger::Callback, &stdLog);
```

But because the StderrLogger is compiled into the SDK, you can do the following:

```
NvVFX_ConfigureLogger(LOG_ERROR, "stderr", NULL, NULL);
```

12.4.30.3.3 FileLogger

```
FileLogger flog("/path/to/vfx.log", "w");
NvVFX_ConfigureLogger(LOG_ERROR, NULL, &FileLogger::Callback, &flog);
```

This configures the logger to write to `vfx.log`, starting fresh. If `"a"` were supplied instead of `"w"`, it would *append* to the existing file instead.

But since the FileLogger is compiled into the SDK, you can do the following:

```
NvVFX_ConfigureLogger(LOG_ERROR, "/path/to/vfx.log", NULL, NULL);
```

This does not offer the option to append, however; the specified file is overwritten.

12.4.30.3.4 FileThreadLogger

```
FileThreadLogger flog("/path/to/vfx.log", "w");
NvVFX_ConfigureLogger(LOG_ERROR, NULL, &FileThreadLogger::Callback, &flog);
```

Similar to the FileLogger, `"a"` is available as the second argument to *append* to the specified file instead of *overwriting* it.

12.4.30.3.5 MultifileLogger

```
MultifileLogger mflog("/path/to/log%02d.log", 65536, 12, 0);
NvVFX_ConfigureLogger(LOG_ERROR, &MultifileLogger::Callback, &mflog);
```

This configures the logger to write to a set of 12 files, starting with `log00.log`, with no file larger than 65536 bytes. When that maximum size is likely to be exceeded, the `log00.log` file is closed and logging continues in `log01.log`. When `log11.log` is full, logging continues with overwriting `log00.log`, using the same 12 files in a circular manner.

The last argument is used to specify which file is to be written first. This is useful when the service was stopped, then restarted, and it is desired to continue writing in the same sequence as before, without resetting to file 0.

12.4.30.4 Advanced: Changing the Logging Level

Sometimes you might want to change the level of logging within the application code. For example, when debugging a particular section of code, you might want a higher level of verbosity while in that code. You can do so by using the special filename `same`.

```
MultifileLogger mflog("/path/to/log%d.log", 65536, 4, 0);
NvVFX_ConfigureLogger(LOG_ERROR, &MultifileLogger::Callback, &mflog);
… <code logged at the ERROR level>
NvVFX_ConfigureLogger(LOG_INFO, "same", NULL, NULL);
… <code logged at the INFO level>
NvVFX_ConfigureLogger(LOG_ERROR, "same", NULL, NULL);
… <code logged at the ERROR level>
```

What if you don't know the current level of verbosity but want to restore it to that level? Use the `cb_data` parameter to retrieve that data:

```
int saved_verbosity;
NvVFX_ConfigureLogger(LOG_INFO, "same", NULL, &saved_verbosity);
… <code logged at the INFO level>
NvVFX_ConfigureLogger(saved_verbosity, "same", NULL, NULL);
… <code logged at the previous verbosity>
```

12.5. Return Codes

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

`NVCV_SUCCESS = 0`

Successful execution.

`NVCV_ERR_GENERAL`

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

`NVCV_ERR_UNIMPLEMENTED`

The requested feature is not implemented.

`NVCV_ERR_MEMORY`

The requested operation requires more memory than is available.

`NVCV_ERR_EFFECT`

An invalid effect handle has been supplied.

`NVCV_ERR_SELECTOR`

The specified selector is not valid for this effect filter.

`NVCV_ERR_BUFFER`

No image buffer has been specified.

`NVCV_ERR_PARAMETER`

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

`NVCV_ERR_MISMATCH`

Some parameters, such as 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`

No model with dependencies matches this system

`NVCV_ERR_PARSE`

A parsing or syntax error occurred 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`

A problem occurred while creating the inference engine.

`NVCV_ERR_TRT_ENGINE`

A problem occurred while deserializing the inference runtime engine.

`NVCV_ERR_NPP`

An error 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 was encountered.

`NVCV_ERR_NOTHINGRENDERED`

Nothing was rendered in the specified region.

`NVCV_ERR_OPENGL`

An OpenGL error occurred.

`NVCV_ERR_DIRECT3D`

A Direct3D error 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.

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

Chapter 13. Multi-GPU Overview

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 function 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; 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.

Chapter 14. Default Behavior

This section describes the 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 whether the GPU architecture supports the Video Effects SDK:

- If yes, *NvVFX_Load()* uses the GPU.
- If not, *NvVFX_Load()* searches for the most powerful GPU that supports the 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.

Chapter 15. Configure a Single Task

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() ...
...
```

Chapter 16. Configure Multiple 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 whether 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 whether the GPU is the best GPU for each specific task.

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

```
// Loop through the GPUs to get the properties of each GPU and
// determine whether 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 NVIDIA Video Effects SDK API call.

```
// Select the best GPU for each task and perform the task.

while (!done) {
...
cudaSetDevice(gpuGame);
RenderGame();
cudaSetDevice(gpuVFX);
vfxErr = NvCVImage_Transfer(&srcCPU, &srcGPU, 1.0f, stream, &tmpGPU);
vfxErr = NvVFX_Run(eff, 1);
vfxErr = NvCVImage_Transfer(&dstGPU, &dstCPU, 1.0f, stream, &tmpGPU);
...
}
```

Chapter 17. Using Multi-Instance GPUs

Important

This section applies to Linux only.

Applications that are developed with the Video Effects SDK can be deployed on Multi-Instance GPU (MIG) technology on supported devices, such as NVIDIA DGX™ A100. MIG lets you 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 need 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.

For more information about MIG and its usage, refer to the NVIDIA Multi-Instance GPU User Guide.

Chapter 18. Batch Overview

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()*.

18.1. 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 call *NvVFX_SetU32()*:

```
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()*.

Chapter 19. 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 *n*th 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 *n*th image APIs, but the source code illustrates an alternative method of accessing images in a batch.

Chapter 20. Allocation of Batched Buffers

To allocate batched buffers, call the `AllocateBatchBuffer()` function, which allocates 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` object 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 `AllocateBatchBuffer()` yields is used only for bookkeeping and is never used in any of the Video Effects APIs.

Note

The Video Effects APIs require an `NvCVImage` descriptor for only the first image.

Chapter 21. Select a Batch Model

The Video Effects SDK comprises several runtime engines that not only implement an effect but 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 one, two, 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.

Each effect comes 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, first specify the model batch size:

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

Then call:

```
NvVFX_Load(effect);
```

If the model with the batch size you want is available, `NVCV_SUCCESS` is returned. Otherwise, an appropriate substitution is 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

The `NVCV_ERR_MODELSUBSTITUTION` 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.

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

Chapter 22. Set the Batched Images

The API takes the image descriptors for only 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);
```

Because 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 `NvCVImage` objects on the recently initialized `nthImage` view. As previously suggested, `NthImage()` is just a thin wrapper around `NvCVImage_InitView()` and can be used instead. You can use the `Transfer*` functions in `BatchUtilities.cpp` to copy pixels to and from the batch.

22.1. Setting the Batch Size for NvVFX_Run()

By default, the batch size processed by *NvVFX_Run()* is 1. Before you call *NvVFX_Run()* to process a batch of any other size, call the following:

```
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.

Chapter 23. Batching When Using State Variables

23.1. For Webcam Denoising

Webcam denoising filters use state variables to track the input video streams to remove temporal noise. A batched input (`srcBatch`), however, can contain frames from multiple videos in arbitrary number and order.

When you use webcam denoising, you need to complete some additional steps *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.

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

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

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 also contain multiple frames from the same video stream, but the frames must be arranged in the batch in 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 holds the address of the *m*th state variable.

For example, assume that the batch size is six and that the batched input, `srcBatch`, contains the

frames from input stream #N-1, input stream #q, input stream #q, input stream #1, input stream #p, and 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);
```

If the batch size or the order of the source of images in the batch changes, the size and the contents of `batchOfStates` can be changed and set using *NvVFX_SetObject()* before every *NvVFX_Run()*.

23.1.1. Example Code

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.

23.2. 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, you must provide information about the order and the video source of the images in the batch. *Before* calling *NvVFX_Run()*, you need to complete some additional steps. 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, assume that the batch size is five and srcBatch contains the frames from input stream #N-1, input stream #q, input stream #1, input stream #p, and 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], arrayO fStates[p], arrayOfStates[0]};

vfxErr = NvVFX_SetStateObjectHandleArray(effectHandle, NVVFX_STATE,
batchOfStates);
```

If the batch size or the order of the source of images in the batch changes, the size and the contents of batchOfStates can be changed and set by using the *NvVFX_SetStateObjectHandleArray()* function before every *NvVFX_Run()*.

23.2.1. 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.

Copyright

©2021-2026, NVIDIA Corporation