NVIDIA Audio Effects SDK

Programming Guide (Linux)

PG-09731-001_v1.0 | April 2021

Document History

PG-09731-001_v1.0

Version	Date	Description of Change
0.5	January 8, 2021	Early Access for Linux
1.0	April 2021	▶ New effects added to SDK (Dereverb, Noise Suppression + Dereverb). ▶ Added noise profiles to the Noise Suppression effect (see "About the Background Noise Suppression Effect"). ▶ Support added for Ampere GPUs. ▶ Modified name of sample app from `denoise_wav to effects_demo`. ▶ Added information about suppression and cancellation effects. ▶ Modified information about `NvAFX_Run`. ▶ Modified some function and type definitions. Support for input frame size of 5ms for denoiser effect has been deprecated.

Table of Contents

Chapter 1. Introduction to NVIDIA Audio Effects SDK for Linux

NVIDIA® Audio Effects SDK provides the following audio effects for broadcast use cases with real-time audio processing.

- **Denoising**: Recordings of speech made outside of a recording studio can contain a lot of background noise, which causes the speech to be garbled and difficult to understand.

  The audio denoising effect removes such background noise from audio.

- **Dereverb**: Recordings of speech might contain reverberations from the recording environment, affecting speech clarity.

  The dereverb effect helps remove or suppress such reverbs from audio.

- **Denoise and Dereverb:** The effect combines both the above effects to remove/suppress both noise and reverbs from audio. This offers much better performance than applying these effects separately.

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

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

2.1 Hardware and Software Requirements

2.1.1 Hardware Requirements

The SDK is supported on systems with minimum 10 GB RAM and NVIDIA GPUs with Tensor Cores.

Hardware	Required Version
NVIDIA GPU	NVIDIA GPUs with Tensor Cores ▶ Turing: Tesla T4 ▶ Volta: Tesla V100 ▶ Ampere Note: The SDK does not support Multi-Instance GPU (MIG). If this feature is enabled, you might experience issues.

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

2.1.2 Software Requirements

Software	Required Version
Linux distribution	64-bit Linux distribution The supported distros are: • Ubuntu (18.04) • RHEL7 • RHEL8 • CentOS7 • CentOS8 • Debian 10+
NVIDIA Graphics Driver for Linux	455.23 or later 418/440 can be used with CUDA Forward Compatible Upgrade. See "Using Older Drivers (418/440) with CUDA Forward-Compatible Upgrade" on page 3 for more information.
CUDA/TensorRT/CuDNN  Note: All libraries that are required to use the SDK are in the package, under `external/cuda`, do not need to be separately installed.	CUDA: 11.1 update 1 TensorRT: 7.2.2.3 CuDNN: 8.0.5

2.2 Installing NVIDIA Audio Effects SDK

To develop applications with the NVIDIA Audio Effects SDK, extract the files from SDK package and provide the library path to the extracted library during compilation and linking. A sample application is also bundled with the SDK (source/pre-built binaries).

To install the SDK, extract the contents of the NVIDIA Audio Effects SDK archive to the required location on your computer, for example, by using the following command:

```
tar xvf --one-top-level Audio_Effects_SDK.tar.gz
```

2.2.1 Using Older Drivers (418/440) with CUDA Forward-Compatible Upgrade

Applications can use the SDK with older drivers (418/440) by using the CUDA Forward-Compatible upgrade path (refer to CUDA Forward-Compatible Upgrade Path for more information).

To use older supported drivers with the SDK:

- Download the user-mode CUDA libraries (`libcuda.so.*`) and the JIT compiler libraries for PTX files (`libnvidia-ptxjitcompiler.so.*`) from one of the following locations:
  - The CUDA 11.1 Toolkit/datacenter drivers.
  - The CUDA network repositories (`cuda-compat-11.1`).

Before you run the applications by using the SDK, ensure that `LD_LIBRARY_PATH` contains the location that contains these libraries.

For example, to use the CUDA network repository on an Ubuntu 18.04 system with older drivers:

1. Go to CUDA Toolkit 11.1 Downloads to add the CUDA repository to your system:
   a. Under **Operating System**, click **Linux**.
   b. Under **Distribution**, click **Ubuntu**.
   c. Under **Installer Type**, click **deb (network)**.
   d. To add the CUDA repository to the system, follow the steps under *Installation Instructions*.
2. Update repositories:
   ```
   $ sudo apt-get update
   ```
3. Install the compatibility package:
   ```
   $ sudo apt-get install -y cuda-compat-11-1
   ```
4. The commands in step 3 will install libraries in the `/usr/local/cuda-11.1/compat` directory.

   This path must be appended to `LD_LIBRARY_PATH` when the SDK applications are run:
   ```
   # Add path to LD_LIBRARY_PATH
   $ export LD_LIBRARY_PATH=/usr/local/cuda-11.1/compat:$LD_LIBRARY_PATH
   # Run application
   $ ./effects_demo -c turing_denoise48k_1_cfg.txt
   ```
   Refer to CUDA Forward-Compatible Upgrade Path for more information.

2.3 Sample Applications

The SDK provides the following sample applications:

- `effects_demo`
- `effects_delayed_streams_demo`

Note: These applications include the source code (`effects_demo.cpp`/`effects_delayed_streams_demo.cpp`) and the pre-built binaries.

2.3.1 effects_demo Application

This application demonstrates how to use the SDK to apply effects to audio.

2.3.1.1 Building the Application

To build the sample application:

1. Navigate to the `samples/effects_demo` directory.
2. (Optional) To compile the application instead of running pre-built binary, run the `make` command.

```
:/Audio Effects SDK/samples/effects_demo$ make
```

3. Run the application using one of the following scripts.

```
:/Audio Effects SDK/samples/effects_demo$ ./run_effect.sh -a volta -s 16
-b 1 -e denoiser
:/Audio Effects SDK/samples/effects_demo$ ./run_effect.sh -a volta -s 48
-b 1 -e dereverb
:/Audio Effects SDK/samples/effects_demo$ ./run_effect.sh -a volta -s 16
-b 400 -e denoiser
:/Audio Effects SDK/samples/effects_demo$ ./run_effect.sh -a volta -s 48
-b 400 -e dereverb_denoiser
```

or

```
:/Audio Effects SDK/samples/effects_demo$ ./run_effect.sh -a turing -s 16
-b 1 -e denoiser
:/Audio Effects SDK/samples/effects_demo$ ./run_effect.sh -a turing -s 48
-b 1 -e dereverb
:/Audio Effects SDK/samples/effects_demo$ ./run_effect.sh -a turing -s 16
-b 400 -e denoiser
:/Audio Effects SDK/samples/effects_demo$ ./run_effect.sh -a turing -s 48
-b 400 -e dereverb_denoiser
```

Note: The sample app might hit the limit for maximum number of open files that is imposed by default by the Linux kernel, especially for large batch sizes. When this occurs, the sample application will exit with the following error message:

```
[Error] Unable to read wav file:
../input_files/denoiser/48k/Fan_48k.wav.

Open file limit reached.
```

To increase this limit, before you run the sample application, use the `ulimit` command in the same shell to increase number of open files. For example, `ulimit -n 20000` will increase open file limit to 20,000 for that shell. For more information, refer to your distribution's documentation on how to increase open file descriptor limits.

2.3.1.2 Running the Application

Run the sample application by running the following command:

```
./effects_demo -c config-file
```

where `-c config-file`:

Specifies the path of the effect sample config file, for example, `turing_denoise48k_1_cfg.txt`. Sample config files for 16kHz and 48kHz audio are provided with the sample application.

Note: Config files that are used by the sample app can be generated by using the `run_effects.sh` script. `run_effects.sh` accepts a path by using the `-c` or the `--cfg-file` flag, where the script writes a config file that can be reused to run the sample app.

For example, to denoise a 48kHz stream on a Turing GPU for a batch size of 1, run:

```
./effects_demo -c turing_denoise48k_1_cfg.txt -e denoise
```

The configuration files contain pairs of parameters and their values, with one pair per line. Currently, the following parameters are supported:

- `reset` *list-of-stream-ids*

  Specifies the stream identifiers to reset, starting with 1. Multiple identifiers are separated by spaces.

- `effect` *effect-name*

  Specifies the name of the effect to apply. Supported effects are "denoiser", "dereverb", and "dereverb_denoiser".

- `sample_rate` *audio-sample-rate*

  Specifies the sample rate of the audio in Hz. Supported values are `16000` and `48000`.

- `model` *model-file*

  Specifies the path of the model file to be used in the sample application, for example, `models/volta/denoiser_48k_1152.trtpkg`. The model file should match the audio sample rate that was specified in the `sample_rate` parameter and the number

of input wav files specified in `input_wav_list` parameter (see "Setting the Parameters of an Audio Effect" on page 12 for more information).

Note: 16kHz and 48kHz model files are included in the SDK.

- `frame_size` *frame-size-value-in-milliseconds*

  Specifies the input frame size (in milliseconds) to be used in the `NvAFX_Run()` call.

  The supported values are `5`, `10`, and `20`.

- `input_wav_list` *input-audio-file-list*

  Specifies a list of paths to input noisy audio `.wav` files to use. Each file should contain mono channel audio in signed 16-bit or 32-bit float format with basic WAV header. Multiple files are separated by space. The number of input files must match the number of streams/batch size. In a stream, the files that are separated by a semicolon (;) are processed one after another in the same stream. In addition, if the stream ID exists in the `reset` list, `NvAFX_Reset` is called on the stream identifiers when switching between files.

  For example, the following configuration specifies that streams 1, 2 and 4 use `file1.wav`, `file2.wav` and `file6.wav` as the input to the stream, and stream 3 uses multiple files (`file3.wav, file4.wav, file5.wav`) as the input to the stream:

```
input_wav_list file1.wav file2.wav file3.wav;file4.wav;file5.wav
file6.wav
```

Note: Sample input audio files are included with the sample application that in the `samples/input_files/16k` and the `samples/input_files/48k` directory.

- `output_wav_list` *output-audio-file-list*

  Specifies the path to the files to which the applied effect audio output will be written. Output files contain mono audio in 32-bit float format. Multiple files are separated by spaces. In a stream, if multiple files are specified (separated using semicolon), multiple output files will be created with the same name followed by `_1`, `_2`, and so on.

  For example, in the following configuration, the output will be written to `out1.wav` (output of `file1.wav`), `out2.wav` (output of `file2.wav`), `out3.wav` (output of `file3.wav`), `out3_1.wav` (output of `file4.wav`), `out3_2.wav` (output of `file5.wav`), and `out4.wav` (output of `file6.wav`).

```
input_wav_list file1.wav file2.wav file3.wav;file4.wav;file5.wav
file6.wav
output_wav_list out1.wav out2.wav out3.wav out4.wav
```

Note: Only the .wav file format with basic WAV header is supported.

- `real_time` *`enable`*

  Simulates real-time audio input, set to 1 to enable, or 0 to disable (disabled by default). When this option is enabled, each audio frame is passed to the SDK with a delay, like how audio is received from a physical device or stream. For example, if the frame size is 10ms, each frame is passed in every 10ms, like how audio is received from a microphone (10ms audio received from the mic approximately every 10ms).

- intensity_ratio *ratio*

  Specifies the denoising intensity ratio. The value of this parameter ranges from 0.0 to 1.0, where a higher value indicates a stronger suppression of noise/reverb. A value of 0.0 is equivalent to passing out input audio without noise removal/dereverb.

2.3.2 effects_delayed_streams_demo Application

This application demonstrates the use-case for handling delayed streams. In this sample, each of the input streams falls under one of the following categories:

- `one_step_delay_streams`

  These streams have a delay of 1 frame. For example, if the frame size is 5ms, these streams will have a delay of 5ms. This means that these streams will be active every alternate iteration. As a result, when data from these streams arrives, `NvAFX_Run` should be called two times, once with the delayed data and once with the current data.

- `two_step_delay_streams`

  These streams have a delay of 2 frames. For example, if the frame size is 5ms, these streams will have a delay of 10ms. This means that these streams will be active after every two iterations. As a result, when data from these streams arrives, `NvAFX_Run` should be called three times, twice with the delayed data and once with the current data.

- `always_active_streams`

  These streams have no delay and are always active, with one `NvAFX_Run` call per iteration.

`NvAFX_Run()` calls are made based on the description above to generate processed audio output. The configuration files provide a parameter to specify `one_step_delay_streams` and `two_step_delay_streams` (see "Running the Application" on page 5 for more information). These values and the batch size are used to infer the list of `always_active_streams`.

2.3.2.1 Building the Application

To build the sample application:

1. Navigate to the `samples/effects_delayed_streams_demo` directory.
2. To compile the application, run the `make` command.

```
:/Audio Effects SDK/samples/effects_delayed_streams_demo$ make
```

2.3.2.2 Running the Application

To run the application, run the following command:

```
./effects_delayed_streams_demo -c config-file
```

where `-c config-file`:

Specifies the path of the config file, for example, `turing_denoise48k_10_cfg.txt`.

Sample config files for 16kHz and 48kHz audio are provided with the application. For example:

```
./effects_delayed_streams_demo -c turing_denoise48k_10_cfg.txt
```

Like `effects_demo`, the configuration files contain pairs of parameters and their values, with one pair per line. In addition to the configuration parameters used by `effects_demo`, `effects_delayed_streams_demo` requires the following parameters:

- `one_step_delay_streams` *list-of-stream-id*

  Specifies the stream identifiers that belong to the `one_step_delay_streams` category as mentioned in the previous section. If none of the streams are in this category, this value should be set to `none`.

- `two_step_delay_streams` *list-of-stream-id*

  Specifies the stream identifiers that belong to the `two_step_delay_streams` category as mentioned in the previous section. If none of the streams are in this category, this value should be set to `none`.

Chapter 3. Using NVIDIA Audio Effects SDK for Linux in Applications

The NVIDIA Audio Effects API is a C API but can also be used with applications that are built using C++.

3.1 About the Background Noise Suppression Effect

Note: In this guide, the term ***Background Noise Suppression*** is used interchangeably with ***Denoising***.

Recordings of speech made outside of a recording studio contain a lot of background noise. The Audio Denoiser Effect removes the following types of background noise from audio recordings:

- AC noise
- PC noise
- Babble / crowd noise
- Chatter from other people
- Keyboard
- Fan noise
- Sirens
- Clapping
- Tapping
- Sounds of furniture moving
- Sounds of glass breaking
- Traffic noise
- Mouse clicks
- Sounds of a train passing by

- Sounds of a vacuum cleaner
- Washing machine
- Metal sounds
- Baby crying
- Wrappers (plastic / non- plastic rustling)
- Water taps / running water
- Cooking sounds (cutting, cooker, etc)
- Construction site sounds
- Rains
- Pet sounds
- Drums
- Door slamming
- Bird chirping
- Phone ringing

This effect has the following characteristics:

- Supported input/output audio format is 32-bit float audio with a sampling rate of16kHz/48kHz.
- The minimum latency of this effect is 30 ms.
- Maximum batches supported by this effect:

Architecture	Maximum Batch Size for the 16K Effect	Maximum Batch Size for the 48K Effect
Turing	1152	1152
Volta	2688	2699
GA100 (A100)	5248	5248
GA102 (A10)	3200	3200

3.2 About the Room Echo Cancellation Effect (BETA)

Note: In this guide, the term *Room Echo Cancellation* is used interchangeably with *Dereverb*.

Recordings of speech made in a large room/hall contains echoes and reverbs. The Audio Room Echo Cancellation Effect removes/suppresses such echoes and reverbs from audio recordings.

This effect has the following characteristics:

- Supported input/output format is 32-bit float audio with a sampling rate of 16kHz/48kHz.
- The minimum latency of this effect is 30ms.
- Maximum batches supported by this effect:

Architecture	Maximum Batch Size for the 16K Effect	Maximum Batch Size for the 48K Effect
Turing	1152	1152
Volta	2688	2699
GA100 (A100)	4736	4736
GA102	2994	2994

3.3 About the Room Echo Cancellation + Background Noise Suppression Effect (BETA)

Note: In this guide, the term *Room Echo Cancellation + Background Noise Suppression* is used interchangeably with *Dereverb+Denoiser*.

This effect applies both denoising and dereverb effect on the input audio.

This effect has the following characteristics:

- Supported input/output audio is 32-bit float audio with a sampling rate of 16kHz/48kHz.
- The minimum latency of this effect is 30 ms.
- Maximum batches supported by this effect:

Architecture	Maximum Batch Size for the 16K Effect	Maximum Batch Size for the 48K Effect
Turing	1152	384
Volta	2688	1152
GA100 (A100)	5248	2176
GA102	2688	1152

3.4 Creating an Audio Effect

Call the `NvAFX_CreateEffect()` function with the following parameters:

- The `NvAFX_EffectSelector` type `NVAFX_EFFECT_DENOISER`, `NVAFX_EFFECT_DEREVERB`, or `NVAFX_EFFECT_DEREVERB_DENOISER`.

- The pointer to the location that stores the handle to the newly created audio effect.

The `NvAFX_CreateEffect()` function creates a handle to the audio effect instance for use in additional API calls.

This example creates a denoiser audio effect:

```
NvAFX_Status err = NvAFX_CreateEffect(NVAFX_EFFECT_DENOISER, &handle);
```

3.5 Setting the Parameters of an Audio Effect

An audio effect requires a model to transform the input audio. Each model supports a specific audio sample rate. The input audio sample rate and the path to the model file which supports this sample rate must be set in the SDK. The SDK also supports several frame sizes (the number of samples per frame), which can be queried and set in the SDK (see "Getting the Parameters of an Effect" on page 13 for more information).

To set U32 values, call the `NvAFX_SetU32()` function with the following parameters:

- Previously created effect handle.
- The selector string for the parameter to be set:
  - To set the sample rate, specify:
    `NVAFX_PARAM_SAMPLE_RATE`.
  - To set the number of audio streams, specify:
    `NVAFX_PARAM_NUM_STREAMS`.
  - To set the number of samples per frame, specify:
    `NVAFX_PARAM_NUM_SAMPLES_PER_FRAME`.
- An unsigned integer value specifying the value for the selector.

To set the model, call the `NvAFX_SetString()` function with the following parameters:

- Previously created effect handle.
- A null-terminated string specifying the path to the model file.
  - Each model file supports a specific sample rate and a maximum number of audio streams.
  - The specified model should match the sample rate and specified number of audio streams.
  - The model file name follows the format:
    `<effect>_<samplerate>_<max-streams>.trtpkg`
  - For convenience, each folder includes a symlink (for example `denoise_16k.trtpkg` and `denoiser_48k.trtpkg`) which points to the actual model

- `samplerate` can be `16k` or `48k`.
- Number of audio streams should be within the range 1 and `max-streams` (both inclusive).
- The model gives best throughput performance when number of audio streams is set to 64 or a multiple of 256 (256, 512, 768, and so on).

For example, the `denoiser_48k_1152.trtpkg` model can be used for 48kHz and between 1 to 1152 audio streams but will be optimal for 64, 256, 512, 768, and 1024 streams. Code that uses this model can also directly use the symlink `denoiser_48k.trtpkg` in the same folder, which allows the underlying model to be changed without code changes.

For example, the following code sets the sample rate to `sample_rate` and the path to the model specified by the `model_file.c_str()`.

```
NvAFX_Status err;
err = NvAFX_SetU32(handle, NVAFX_PARAM_SAMPLE_RATE, sample_rate);
err = NvAFX_SetString(handle, NVAFX_PARAM_MODEL_PATH, model_file.c_str());
err = NvAFX_SetU32(handle, NVAFX_PARAM_NUM_STREAMS, num_streams);
```

3.6 Getting the Parameters of an Effect

The number of channels in input/output audio are fixed for the Audio Effect and cannot be changed. Before running an audio effect, the number of channels that are supported by the effect must be queried. The SDK also supports several frame sizes (number of samples per frame), which can be queried and set by using the `set` API (see "Setting the Parameters of an Audio Effect" on page 12 for more information). The application can also query and use the default frame size supported by the SDK, as demonstrated in the following sample.

Note: To ensure that the sample rate of the input audio is compatible with the Audio Effect, the sample rate should be queried first.

To query these parameters, call the `NvAFX_GetU32()` function with the following parameters:

- Previously created effect handle.
- The selector string for the parameter to be queried:
  - To get the default number of samples per frame, specify `NVAFX_PARAM_NUM_SAMPLES_PER_FRAME`.
  - To get the number of channels in input/output audio, specify `NVAFX_PARAM_NUM_CHANNELS`.
  - To get the sample rate, specify `NVAFX_PARAM_SAMPLE_RATE`.
- A pointer to the location where the value is to be stored.

To query lists, the user must first query the list size and allocate memory for the output and then pass in the newly allocated memory and size to `NvAFX_GetU32List()`.

To query the list size, call the `NvAFX_GetU32List()` function with the following parameters:

- Previously created effect handle.
- The selector string for the parameter to be queried:

  To get the list of supported number of samples per frame, specify `NVAFX_PARAM_SUPPORTED_NUM_SAMPLES_PER_FRAME`.
- An output pointer, set to `nullptr` (or `NULL`) to query size
- A pointer to the location where the size of the list is to be stored. The size should be initialized to zero, will be updated with the actual size when this function is called.

  The `NvAFX_GetU32List()` call retrieves the size of the list for the corresponding parameter selector with an `NVAFX_STATUS_OUTPUT_BUFFER_TOO_SMALL` error status. To query the list, allocate memory for the list with the returned size and call the `NvAFX_GetU32List()` function with the following parameters:
  - The selector string for the parameter to be queried:

    To get the list of supported number of samples per frame, specify `NVAFX_PARAM_SUPPORTED_NUM_SAMPLES_PER_FRAME`.
  - A pointer to a U32 array of size at least of the list size retrieved from the above call.

    The list values are written to this array.
  - A pointer to a location where the value of the size of the list is stored.

The following example queries an effect for the supported number of samples per frame, the number of channels in input/output audio, the sample rate, and the supported frame sizes.

```
unsigned num_samples_per_frame, num_channels, sample_rate;
NvAFX_Status err;
err = NvAFX_GetU32(handle, NVAFX_PARAM_NUM_SAMPLES_PER_FRAME,
&num_samples_per_frame);
err = NvAFX_GetU32(handle, NVAFX_PARAM_NUM_CHANNELS, &num_channels);
err = NvAFX_GetU32(handle, NVAFX_PARAM_SAMPLE_RATE, &sample_rate);

std::unique_ptr<unsigned int[]> supported_list = nullptr;
int list_size = 0;
err = NvAFX_GetU32List(handle, NVAFX_PARAM_SUPPORTED_NUM_SAMPLES_PER_FRAME,
supported_list.get(), &list_size);
if (err != NVAFX_STATUS_OUTPUT_BUFFER_TOO_SMALL) {
  // This indicates API failure
  return;
}
supported_list.reset(new unsigned int[list_size]);
err = NvAFX_GetU32List(handle, NVAFX_PARAM_SUPPORTED_NUM_SAMPLES_PER_FRAME,
supported_list.get(), &list_size);
```

3.7 Loading an Audio Effect

Loading an effect involves validating the parameters that were set for the effect and loading the specified model into GPU memory.

To load an audio effect, set the parameters for the effect described in the previous section and call `NvAFX_Load()` function with the effect handle.

```
NvAFX_Status err = NvAFX_Load(handle);
```

3.8 Running an Audio Effect

Once the effect is loaded, it can be applied to input audio using the `NvAFX_Run()` function. When the effect is run, the contents of the input memory buffer are read, the audio effect is applied, and the output is written to the output memory buffer.

To run an audio effect, call the `NvAFX_Run()` function with the following parameters:

- Previously created effect handle.
- The input memory buffer.
- The output memory buffer.
- The number of samples per frame per stream of input/output data.
- The number of channels in input/output audio.
  See ¨Getting the Parameters of an Effect¨ on page 13 for more information.

The following example runs an audio effect:

```
NvAFX_Status err = NvAFX_Run(handle, input, output, num_samples,
num_channels);
```

3.9 Running an Audio Effect on Delayed Audio Streams

The SDK supports cases where some streams do not arrive at the expected time. These streams are referred to as delayed streams. To support handling these streams, the SDK allows applications to specify a list that indicates whether the corresponding stream is currently active or delayed/inactive.

The list can be set by calling `NvAFX_SetBoolList()` with the following function parameters:

- Previously created effect handle.
- The selector string, `NVAFX_PARAM_ACTIVE_STREAMS`, for the parameter that will be set.

- An array of `NVAFX_BOOL` datatype where each element represents the status of corresponding audio stream with `NVAFX_TRUE` indicating an active stream and `NVAFX_FALSE` indicating an inactive stream.
- Length of the above array that is equal to the number of audio streams.

For delayed audio streams, the effect can be initially applied on all delayed audio streams by setting them as active and setting the on-time audio streams as inactive. This should be followed by one or more `NvAFX_Run()` calls to apply the effect on the delayed audio streams. After the delayed audio streams are processed, the on-time audio streams are set to active, and `NvAFX_Run()` is executed once to apply the effect.

Here is an example of how to process four streams:

1. Consider that the effect accepts 10ms audio inputs.
2. Audio streams 1 and 3 are delayed by 10ms each and arrive with 20ms worth of data.
3. Audio streams 2 and 4 are on time and arrive with 10ms of data.
4. The process completes the execution with one of the following options :
   - **Option 1:** Process the extra 10ms **only** in the delayed streams and then process on-time 10ms data for **all** streams.

     Initially, by using `NvAFX_SetBoolList`, streams 1 and 3 are set as active, and 2 and 4 are set as inactive.

     a. An `NvAFX_Run` call is executed where 10ms of data from streams 1 and 3 is populated in the input while the rest of input is set to 0.

        This step processes the extra 10ms of data in streams 1 and 3.

     b. A second `NvAFX_SetBoolList` call is executed to set all streams (1, 2, 3, and 4) as active.

     c. An `NvAFX_Run` call is executed with the real-time 10ms data from all four streams.
   - **Option 2:** Process 10ms in **all** streams and then process the extra 10ms data **only** in delayed streams:

     a. An `NvAFX_Run` call is executed with 10ms of data from all streams (stale data from stream 1 and 3 and new data from stream 2 and 4) by calling `NvAFX_Run`.

     b. `NvAFX_SetBoolList` is used to set streams 1 and 3 to active and 2 and 4 to inactive.

     c. An `NvAFX_Run` call is executed with the extra 10ms from stream 1 and 3.

The following example runs an audio effect after setting some of the audio streams as inactive:

```
NvAFX_Status err = NvAFX_SetBoolList(handle, NVAFX_PARAM_ACTIVE_STREAMS,
stream_active_list, num_streams);
```

```
NvAFX_Status err = NvAFX_Run(handle, input, output, num_samples,
num_channels);
```

The internal state of each stream is updated during each `NvAFX_Run` call only for active streams. Setting a stream to inactive will disable updating this state. If required, this state can also be reset using `NvAFX_Reset`, as described in "NvAFX_Reset" on page 31.

3.10 Destroying an Audio Effect

When an audio effect is no longer required, it should be destroyed to free the resources and memory allocated to the effect.

To destroy an audio effect, call the `NvAFX_DestroyEffect()` function specifying the effect handle to the effect to be destroyed.

```
NvAFX_Status err = NvAFX_DestroyEffect(handle);
```

3.11 Using Multiple GPUs

Applications that are developed with the NVIDIA Audio Effects SDK for Linux can be used with multiple GPUs. By default, the SDK assumes that the application will set the GPU. Optionally, the SDK can select the best GPU to run the effect(s).

3.11.1 Selecting GPU for Audio Effects Processing in a Multi-GPU Environment

The GPU to be used to run audio effect(s) in a multi-GPU environment can be controlled by using the `cudaSetDevice()` and `cudaGetDevice()` CUDA functions.
The device should be set **before** `NvAFX_Load`() is called, because `NvAFX_Load()` will succeed only when the currently selected GPU supports the SDK.

```
int chosenGPU = 0; // or whichever GPU you want to use
cudaSetDevice(chosenGPU);
NvAFX_Handle effect;
err = NvAFX_API NvAFX_CreateEffect(code, &effect);
err = NvAFX_Set...; // set parameters
…
err = NvAFX_API NvAFX_Load(effect);
…
err = NvAFX_API NvAFX_Run(effect, ...);
```

3.11.2 Offloading GPU selection to SDK for Audio Effects Processing in a multi-GPU environment

To let the SDK determine which GPU to run the audio effect(s) on, you can use the `NvAFX_SetU32(effect, NVAFX_PARAM_USE_DEFAULT_GPU, 1)` function. This optional call should be called only once before any effect is loaded. If it is called after an audio effect is loaded, it will not have any effect.

If the application sets `NVAFX_PARAM_USE_DEFAULT_GPU` to `1`, the application should not call `cudaSetDevice()`. If the application explicitly calls `cudaSetDevice()` before `NvAFX_Load()`, the SDK will override application's device preference. If the client calls `cudaSetDevice()` to set GPU to a different GPU before calling `NvAFX_Run()`, the call will fail.

If the application sets `NVAFX_PARAM_USE_DEFAULT_GPU` to `0`, the SDK will not explicitly select the GPU to run the effect. The application can set the device on which SDK calls are executed using `cudaSetDevice` to set the device. If this is not set, the SDK will use the default device (device 0).

```
NvAFX_Handle effect;
err = NvAFX_API NvAFX_CreateEffect(code, &effect);
err = NvAFX_API SetU32(effect, NVAFX_PARAM_USE_DEFAULT_GPU, 1);
…
err = NvAFX_API NvAFX_Load(effect);
…
```

3.11.3 Selecting Different GPUs for Different Tasks

The applications that use the SDK might be designed to perform multiple tasks in a multi-GPU environment in addition to applying the audio effect filter. In this situation, the best GPU for each task should be selected before calling `NvAFX_Load()` and be set before each `NvAFX_Run()` call. Switching to the appropriate GPU is the responsibility of the application. If the application does not switch to appropriate GPU before calling `NvAFX_Run()`, the call will fail with error.

To select the best GPU:

1. Call `cudaGetDeviceCount()` to determine the number of available GPUs.

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

2. Determine the best GPU for the task.

   For example, this can be determined by iterating over the available GPUs and selecting the GPU with the highest number of SMs by using `cudaGetDeviceProperties()`.

3. In the loop that completes the application's tasks, select the best GPU for each task before performing the task. Call `cudaSetDevice()` to select the GPU for the task.

```
// Select the best GPU for each task and perform the task.
while (!done) {
  ...
  cudaSetDevice(gpuOtherTask);
  PerformOtherTask();
  cudaSetDevice(gpuAFX);
  err = NvAFX_Run(effect, ...);
```

Chapter 4. NVIDIA Audio Effects SDK API Reference

4.1 Type Definitions

NVIDIA Audio Effects SDK type definitions provide selector strings for the audio effect and the parameters of an audio effect.

4.1.1 NvAFX_EffectSelector

```
typedef const char* NvAFX_EffectSelector;
```

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

Currently supported selectors are:

NVAFX_EFFECT_DENOISER : "denoiser"

Denoiser audio effect.

NVAFX_EFFECT_DEREVERB "dereverb"

De-reverb effect.

NVAFX_EFFECT_DEREVERB_DENOISER "dereverb_denoiser"

Combined De-reverb and Denoiser effects.

4.1.2 NvAFX_ParameterSelector

```
typedef const char* NvAFX_ParameterSelector;
```

This type definition provides selector strings for the parameters of an audio effect.

Currently supported selectors are:

NVAFX_PARAM_MODEL_PATH: "model_path"

A character string that specifies the path to the model file for the Audio effect.

NVAFX_PARAM_SAMPLE_RATE: "sample_rate"

An unsigned integer that specifies the audio sample rate for the Audio effect.

NVAFX_PARAM_NUM_SAMPLES_PER_FRAME: "num_samples_per_frame"

An unsigned integer that specifies the number of samples per frame per audio stream for the Audio effect.

NVAFX_PARAM_NUM_CHANNELS: "num_channels"

An unsigned integer that specifies the number of audio channels for the Audio effect.

`NVAFX_PARAM_NUM_STREAMS: "num_streams"`

An unsigned integer that specifies the number of audio streams to be processed by the audio effect.

`NVAFX_PARAM_ACTIVE_STREAMS: "active_streams"`

A list of NvAFX_Bool values that specify whether the corresponding stream is active.

`NVAFX_PARAM_SUPPORTED_NUM_SAMPLES_PER_FRAME: "supported_num_samples_per_frame"`

A list of U32 values specifying the supported values for number of samples per frame.

4.1.3 NvAFX_Handle

```
typedef void* NvAFX_Handle;
```

This type represents an opaque handle associated with each instance of an audio effect.

4.1.4 NvAFX_Bool

```
typedef char NvAFX_Bool;
```

This type represents a Boolean type and should be set to `NVAFX_TRUE` to represent true, else `NVAFX_FALSE`.

4.1.5 logging_cb_t

```
typedef void(*logging_cb_t)(LoggingSeverity level, const char* log, void*
userdata);
```

This is the callback function type used in `NvAFX_InitializeLogger` API. See ¨NvAFX_InitializeLogger¨ on page 32 for more information.

4.1.6 LoggingSeverity

```
typedef enum LoggingSeverity_t {
  LOG_LEVEL_ERROR,
  LOG_LEVEL_WARNING,
  LOG_LEVEL_INFO,
} LoggingSeverity;
```

This enum provides the levels of LoggingSeverity used in `NvAFX_InitializeLogger` API. See ¨NvAFX_InitializeLogger¨ on page 32 for more information.

4.1.7 LoggingTarget

```
typedef enum LoggingTarget_t
{
    LOG_TARGET_NONE = 0x0,
    LOG_TARGET_STDERR = 0x1,
    LOG_TARGET_FILE = 0x2,
    LOG_TARGET_CALLBACK = 0x4,
} LoggingTarget;
```

This enum provides the logging target used in `NvAFX_InitializeLogger` API. See ¨NvAFX_InitializeLogger¨ on page 32 for more information.

4.2 Functions

4.2.1 NvAFX_GetEffectList

```
NvAFX_Status NvAFX_GetEffectList(
  int* num_effects,
  NvAFX_EffectSelector* effects[]
);
```

4.2.1.1 Parameters

num_effects [out]

Type: `int*`

Pointer to an integer that contains the number of effects returned.

effects [out]

Type: `NvAFX_EffectSelector* []`

Address to a list of effect selection strings supported by the SDK. This list is statically allocated by the SDK, so the caller should not allocate memory for this parameter or free it after use. See "NvAFX_EffectSelector" on page 20 for more information about selection strings.

4.2.1.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.1.3 Remarks

This function retrieves the list of audio effects supported by the SDK. The selection strings for the Audio Effects SDK are populated in the `effects` output parameter. The number of available effects is written to the `num_effects` output parameter.

4.2.2 NvAFX_CreateEffect

```
NvAFX_Status NvAFX_CreateEffect(
  NvAFX_EffectSelector code,
  NvAFX_Handle* effect
);
```

4.2.2.1 Parameters

code [in]

Type: `NvAFX_EffectSelector`

The selection string for the type of audio effect to be created. See "NvAFX_EffectSelector" on page 20 for more information about the allowed selection strings.

effect [out]

Type: `NvAFX_Handle*`

The pointer to the location where the handle to the newly created audio effect instance will be stored.

4.2.2.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.2.3 Remarks

This function creates an instance of the specified type of audio effect and returns the handle to this effect via the `effect` output parameter.

4.2.3 NvAFX_DestroyEffect

```
NvAFX_Status NvAFX_DestroyEffect(
  NvAFX_Handle effect
);
```

4.2.3.1 Parameters

effect [in]

Type: `NvAFX_Handle`

The handle to the audio effect instance to be destroyed.

4.2.3.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.3.3 Remarks

This function destroys the audio effect instance with the specified handle and frees all resources and memory used by that instance.

4.2.4 NvAFX_SetString

```
NvAFX_Status NvAFX_SetString(
  NvAFX_Handle effect,
  NvAFX_ParameterSelector param_name,
  const char* val
);
```

4.2.4.1 Parameters

effect [in]

Type: `NvAFX_Handle`

The handle to the audio effect instance.

param_name [in]

Type: `NvAFX_ParameterSelector`

The selector string `NVAFX_PARAM_MODEL_PATH`.

Any other selector string returns an error.

val [in]

Type: `char*`

Pointer to the character string to be set.

4.2.4.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.4.3 Remarks

This function sets the value of the specified character string parameter for the specified audio effect to `val`.

4.2.5 NvAFX_SetU32

```
NvAFX_Status NvAFX_SetU32(
  NvAFX_Handle effect,
  NvAFX_ParameterSelector param_name,
  unsigned int val
);
```

4.2.5.1 Parameters

effect [in]

Type: `NvAFX_Handle`

The handle to the audio effect.

Param_name [in]

Type: `NvAFX_ParameterSelector`

One of the following options:

- `NVAFX_PARAM_SAMPLE_RATE`
- `NVAFX_PARAM_NUM_STREAMS`.
- `NVAFX_PARAM_NUM_SAMPLES_PER_FRAME`

Any other selector string returns an error.

Note: Valid values for setting `NVAFX_PARAM_NUM_SAMPLES_PER_FRAME` can be queried using `NvAFX_GetU32List()` function with `NVAFX_PARAM_SUPPORTED_NUM_SAMPLES_PER_FRAME` as the selector. Setting any other value would result in an error.

val [in]

Type: `unsigned int`

Value to be set for the parameter.

4.2.5.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.5.3 Remarks

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

4.2.6 NvAFX_GetString

```
NvAFX_Status NvAFX_GetString(
  NvAFX_Handle effect,
  NvAFX_ParameterSelector param_name,
  char* val,
  int max_length
);
```

4.2.6.1 Parameters

effect [in]

Type: `NvAFX_Handle`

The handle to the audio effect instance.

Param_name [in]

Type: `NvAFX_ParameterSelector`

The selector string `NVAFX_PARAM_MODEL_PATH`.

Any other selector string returns an error.

val [out]

Type: `char*`

The address of the buffer where the requested character string would be stored. This buffer must be allocated by and freed by the caller.

max_length [in]

Type: `int`

The length in bytes of the buffer that is specified by the `val` parameter.

4.2.6.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.6.3 Remarks

This function gets the value of the character string parameter for the specified audio effect and writes the retrieved string to the buffer at the location specified by the `val` parameter.

4.2.7 NvAFX_GetU32

```
NvAFX_Status NvAFX_GetU32(
  NvAFX_Handle effect,
  NvAFX_ParameterSelector param_name,
  unsigned int* val
);
```

4.2.7.1 Parameters

effect [in]

Type: `NvAFX_Handle`

The handle to the audio effect instance.

param_name [in]

Type: `NvAFX_ParameterSelector`

Either of

- `NVAFX_PARAM_NUM_SAMPLES_PER_FRAME`
- `NVAFX_PARAM_NUM_CHANNELS`
- `NVAFX_PARAM_SAMPLE_RATE`

Any other selector string returns an error.

Note: `NVAFX_PARAM_NUM_CHANNELS` parameter is preset for all Audio Effect and cannot be changed. If you call `NvAFX_SetU32()` with this parameter, the function call will return an error.

While `NVAFX_PARAM_NUM_SAMPLES_PER_FRAME` can be queried using this API to get the default number of samples per frame, it is advised to use `NvAFX_GetU32List()` with `NVAFX_PARAM_SUPPORTED_NUM_SAMPLES_PER_FRAME` parameter to get the list of supported values. You can then use the `NvAFX_SetU32()` with `NVAFX_PARAM_NUM_SAMPLES_PER_FRAME` parameter to set the value.

val [out]

Type: `unsigned int*`

The address of the buffer where the retrieved 32-bit unsigned integer parameter value will be written.

4.2.7.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.7.3 Remarks

This function gets the value of the specified 32-bit unsigned integer parameter for the specified audio effect and writes the retrieved value to the buffer specified by `val`.

4.2.8 NvAFX_GetU32List

```
NvAFX_Status NvAFX_GetU32List(
  NvAFX_Handle effect,
  NvAFX_ParameterSelector param_name,
  unsigned int* list[],
  int* list_size
);
```

4.2.8.1 Parameters

effect [in]

Type: `NvAFX_Handle`

The handle to the audio effect instance.

param_name [in]

Type: `NvAFX_ParameterSelector`

The following selector:
`NVAFX_PARAM_SUPPORTED_NUM_SAMPLES_PER_FRAME`

Any other selector string returns an error.

Note: Values returned for `NVAFX_PARAM_SUPPORTED_NUM_SAMPLES_PER_FRAME` as the selector depends on the sample rate. It needs to be ensured that `NvAFX_SetU32()` is called with `NVAFX_PARAM_SAMPLE_RATE` selector to set the sample rate before making this call.

list [out]

Type: `unsigned int* []`

The address to a list containing the 32-bit unsigned values for the given selector.

Note: The application needs to call this API with `list_size` initialized to zero, and `list` set to `nullptr` to get the size of list to be allocated. The size will be returned in `list_size` parameter.The application can then allocate an U32 array of at least `list_size` and call the API again with `list` pointing to the array. Refer to section 3.4 for an example.

list_size [out]

Type: `int*`

Pointer to an integer that contains the number of values returned in the list.

4.2.8.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

`NVAFX_STATUS_OUTPUT_BUFFER_TOO_SMALL` when the `list_size` is less than the minimum required size of `list` array.

4.2.8.3 Remarks

This function gets the list of 32-bit unsigned integer values for the specified audio effect and writes the retrieved values to a buffer specified by `list` and writes the size of the returned list in the buffer specified by `list_size`.

4.2.9 NvAFX_Load

```
NvAFX_Status NvAFX_Load(
  NvAFX_Handle effect
);
```

4.2.9.1 Parameters

effect [in]

Type: `NvAFX_Handle`

The handle to the audio effect instance to load.

4.2.9.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.9.3 Remarks

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

4.2.10 NvAFX_Run

```
NvAFX_Status NvAFX_Run(
  NvAFX_Handle effect,
  const float** input,
  float** output,
  unsigned num_samples,
  unsigned num_channels
);
```

4.2.10.1 Parameters

effect [in]

Type: `NvAFX_Handle`

The handle to the audio effect instance to run.

input [in]

Type: `const float**`

Pointer to a user-allocated array of buffers where each buffer holds the audio data for one channel. The size of the array must be equal to the number of I/O channels that were preset for the effect. For example, for the Audio Effect, the number of I/O channels must be equal to the value of the `NVAFX_PARAM_NUM_CHANNELS` parameter that was obtained by the `NvAFX_GetU32()` function.

The sample rate of the audio data must be equal to the sample rate that was preset for the effect. For example, for the Audio Effect, the sample rate must be equal to the value of the `NVAFX_PARAM_SAMPLE_RATE` parameter that was obtained by the `NvAFX_GetU32()` function.

output [out]

Type: `float**`

Pointer to a user-allocated array of buffers to which the output of the effect will be written. After this function returns, each buffer will contain audio data for one channel.

Note: The buffers must be allocated by, and later freed, by the calling program.

`NvAFX_Run` internally copies input/output to/from the GPU, so pinning input/output buffers does not have any effect.

The size of each buffer is same as the size of each buffer that was specified by the `input` parameter.

num_samples [in]

Type: `unsigned`

The number of samples in the input buffer. After this function returns, the buffer that was specified by the `output` parameter will contain the number of samples that were specified in this parameter.

num_channels [in]

Type: `unsigned`

The number of I/O channels.

4.2.10.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.10.3 Remarks

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

4.2.11 NvAFX_Reset

```
NvAFX_Status NvAFX_Reset(
  NvAFX_Handle effect,
  NvAFX_Bool* list,
  int length
);
```

4.2.11.1 Parameters

effect [in]

Type: `NvAFX_Handle`

The handle to the audio effect instance to run.

list [in]

Type: `NvAFX_Bool *`

Pointer to a memory location which indicates the streams to be reset. The `i`-th element in this array should be set to `NVAFX_TRUE` to reset the `i`-th stream, and to `NVAFX_FALSE` otherwise

length [in]

Type: `int`

Number of elements in the array specified. Should be equal to the number of streams (batches).

4.2.11.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.12 NvAFX_SetBoolList

```
NvAFX_Status NvAFX_SetBoolList(
  NvAFX_Handle effect,
  NvAFX_ParameterSelector param_name,
  const NvAFX_Bool* list,
  unsigned int list_size
);
```

4.2.12.1 Parameters

effect [in]

Type: `NvAFX_Handle`

The handle to the audio effect.

Param_name [in]

Type: `NvAFX_ParameterSelector`

The following:

`NVAFX_PARAM_ACTIVE_STREAMS`

Any other selector string returns an error.

list [in]

Type: `NvAFX_Bool*`

Array of Boolean values to be set for the parameter.

list_size [in]

Type: `unsigned int`

Size of the Boolean array passed as input.

4.2.12.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.12.3 Remarks

This function sets the boolean values of the list parameter for the specified audio effect to the values from `list`.

4.2.13 NvAFX_InitializeLogger

```
NvAFX_Status NvAFX_InitializeLogger(
  LoggingSeverity level,
  int target,
  const char *filename,
  logging_cb_t cb,
```

```
  void* userdata
);
```

4.2.13.1 Parameters

level [in]

Type: `LoggingSeverity`

The logging level to enable. Enabling a level is inclusive of the levels preceding it. For example, LOG_LEVEL_INFO also includes LOG_LEVEL_WARNING and LOG_LEVEL_ERROR.

Either of the following are valid values:

- `LOG_LEVEL_ERROR`
- `LOG_LEVEL_WARNING`
- `LOG_LEVEL_INFO`

Target [in]

Type: int

Logging targets to write logs to, `LoggingTarget` can be binary OR'd to enable multiple targets.

The following targets can be used:

- `LOG_TARGET_NONE = 0x0`
- `LOG_TARGET_STDERR = 0x1`
- `LOG_LEVEL_FILE = 0x2`
- `LOG_LEVEL_CALLBACK = 0x4`

filename [in]

Type: const char*

The path of the file where to write logs. Used only when LOG_TARGET_FILE is enabled.

Note: The directory in which the log file resides should exist. For example, if filename is `/foo/bar/log.txt`, the `/foo/bar` directory must already exist. If the `log.text` file exists, it will be overwritten.

cb [in]

Type: const char *

Callback to use if `LOG_TARGET_CALLBACK` is enabled. A null value can be passed when not using callback target.

userdata [in]

Type: void *

Data passed back with log callback. Used only when `LOG_TARGET_CALLBACK` is enabled.

A null value can also be passed.

4.2.13.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.13.3 Remarks

This API enables logging in the SDK. Depending on the flags passed, logs are either redirected to stderr, file, callback. Logging can be disabled using the `NvAFX_UninitializeLogger` API. See "NvAFX_UninitializeLogger" on page 34 for more information.

4.2.14 NvAFX_UninitializeLogger

```
NvAFX_Status NvAFX_UninitializeLogger(void);
```

4.2.14.1 Parameters

`NvAFX_UninitializeLogger` requires no parameters.

4.2.14.2 Return Value

`NVAFX_STATUS_SUCCESS` on success.

4.2.14.3 Remarks

This API disables all logging targets. Logging can be started again using the `NvAFX_InitializeLogger()` API. See "NvAFX_InitializeLogger" on page 32 for more information.

4.3 Return Codes

The `NvAFX_Status` enumeration defines the following values that the NVIDIA Audio Effects functions might return to indicate error or success:

NVAFX_STATUS_SUCCESS

Successful execution.

NVAFX_STATUS_FAILED

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

NVAFX_STATUS_INVALID_HANDLE

An invalid effect handle has been supplied.

NVAFX_STATUS_INVALID_PARAM

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

NVAFX_STATUS_IMMUTABLE_PARAM
: User tried to modify an immutable parameter.

NVAFX_STATUS_INSUFFICIENT_DATA
: There is insufficient data to process.

NVAFX_STATUS_EFFECT_NOT_AVAILABLE
: The specified effect is not supported.

NVAFX_STATUS_OUTPUT_BUFFER_TOO_SMALL
: The output buffer length is too small to hold the requested data.

NVAFX_STATUS_MODEL_LOAD_FAILED
: The specified model file cannot be loaded.

NVAFX_STATUS_MODEL_NOT_LOADED
: Model is not loaded, and it has to be loaded for this operation.

NVAFX_STATUS_INCOMPATIBLE_MODEL
: Selected model is incompatible.

NVAFX_STATUS_GPU_UNSUPPORTED
: The selected GPU is not supported. The SDK requires Turing and above GPU with Tensor cores.

NVAFX_STATUS_NO_SUPPORTED_GPU_FOUND
: No supported GPU found on the system.

NVAFX_STATUS_WRONG_GPU
: Current GPU is not the one selected.

NVAFX_STATUS_CUDA_ERROR
: Cuda operation failure.

NVAFX_STATUS_INVALID_OPERATION
: Invalid operation performed.

Notice

The information provided in this specification is believed to be accurate and reliable as of the date provided. However, NVIDIA Corporation ("NVIDIA") does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This publication supersedes and replaces all other specifications for the product that may have been previously supplied.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and other changes to this specification, at any time and/or to discontinue any product or service without notice. Customer should obtain the latest relevant specification before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer. NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this specification.

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

NVIDIA makes no representation or warranty that products based on these specifications will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this specification. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this specification, or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this specification. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA. Reproduction of information in this specification is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

VESA DisplayPort

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

HDMI

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

OpenCL

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

Trademarks

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

Copyright