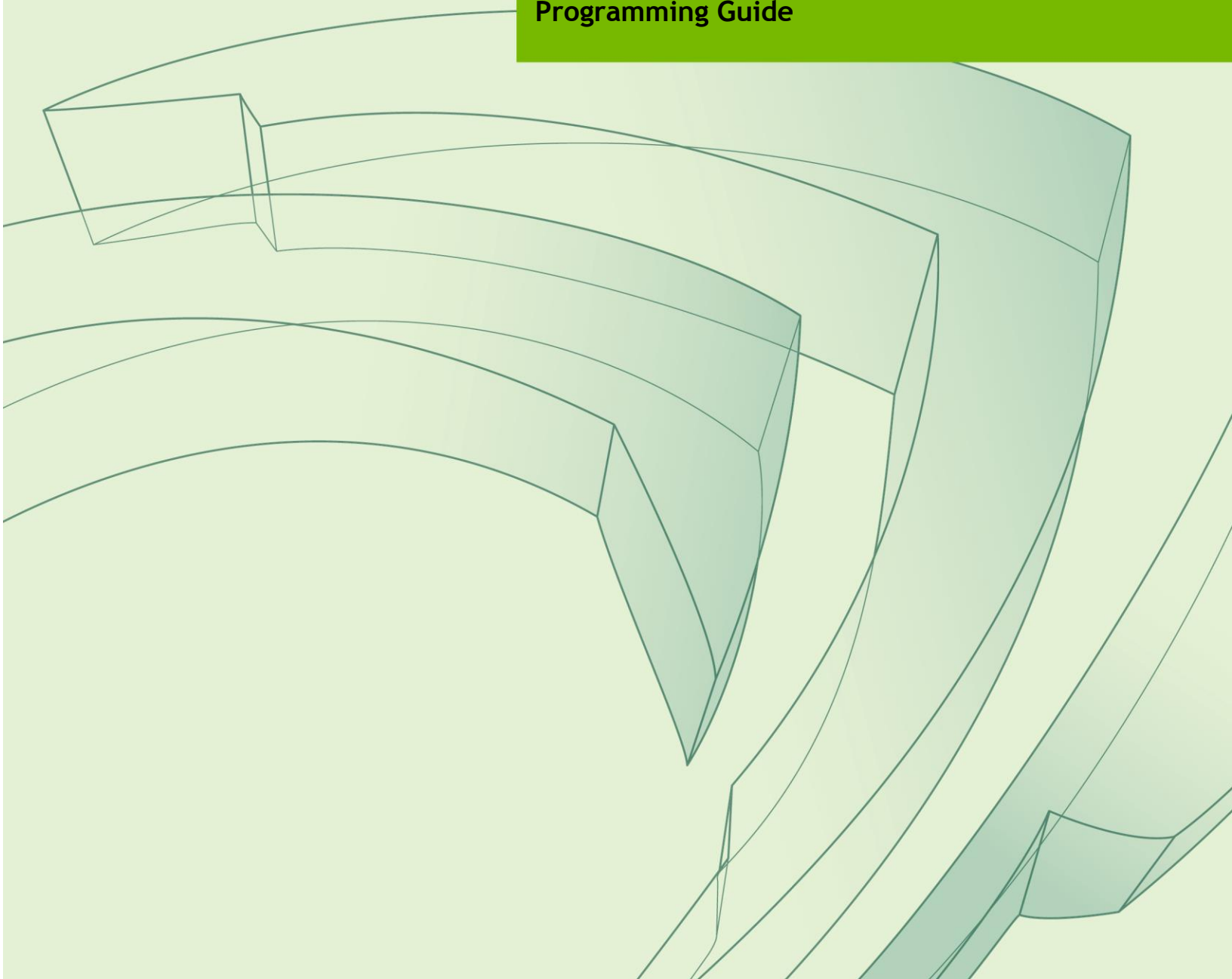
NVIDIA VIDEO ENCODER (NVENC) INTERFACE

NVENC_VideoEncoder_API_PG-06155-001_v07 | June 2016

Programming Guide



DOCUMENT CHANGE HISTORY

NVENC_VideoEncoder_API_PG-06155-001_v07

Version	Date	Authors	Description of Change
1.0	2011/12/29	SD/CC	Initial release.
1.1	2012/05/04	SD	Update for Version 1.1
2.0	2012/10/12	SD	Update for Version 2.0
3.0	2013/07/25	AG	Update for Version 3.0
4.0	2014/07/01	SM	Update for Version 4.0
5.0	2014/11/30	MV	Update for Version 5.0
6.0	2015/10/15	VP	Update for Version 6.0
7.0	2016/6/10	SM	Update for Version 7.0

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

NVIDIA GPUs based on Kepler, Maxwell and the latest Pascal architectures contain a hardware-based H.264/HEVC video encoder (hereafter referred to as NVENC). The NVENC hardware takes YUV/RGB as input, and generates an H.264/HEVC compliant video bit stream. NVENC hardware's encoding capabilities can be accessed using the NVENCODE APIs, available in the NVIDIA Video Codec SDK.

This document provides information on how to program the NVENC using the NVENCODE APIs exposed in the SDK. The NVENCODE APIs expose encoding capabilities on Windows (Windows 7 and above) and Linux.

It is expected that the developers should have understanding of H.264/HEVC video codecs and familiarity with Windows and/or Linux development environment.

NVENCODE API *guarantees* backward compatibility. This means that applications compiled with older versions of released API will continue to work on future driver versions released by NVIDIA.

Chapter 2. BASIC ENCODING FLOW

Developers can create a client application that calls NVENCODE API functions exposed by `nvEncodeAPI.dll` for Windows or `libnvidia-encode.so` for Linux. These libraries are installed as part of the NVIDIA display driver. The client application can either link to these libraries at run-time using `LoadLibrary()` on Windows or `dlopen()` on Linux.

The NVIDIA video encoder API is designed to accept raw video frames (in YUV or RGB format) and output the H.264 or HEVC bitstream. Broadly, the encoding flow consists of the following steps:

1. Initialize the encoder
2. Set up the desired encoding parameters
3. Allocate input/output buffers
4. Copy frames to input buffers and read bitstream from the output buffers. This can be done synchronously (Windows & Linux) or asynchronously (Windows 7 and above only).
5. Close the encoding session
6. Clean-up; release all allocated input/output buffers

These steps are explained in the rest of the document and demonstrated in the sample application included in the Video Codec SDK package.

Chapter 3. SETTING UP HARDWARE FOR ENCODING

3.1 OPENING AN ENCODE SESSION

After loading the DLL or shared object library, the client's first interaction with the API is to call `NvEncodeAPICreateInstance`. This populates the input/output buffer passed to `NvEncodeAPICreateInstance` with pointers to functions which implement the functionality provided in the interface.

After loading the NVENC Interface, the client should first call `NvEncOpenEncodeSessionEx` API to open an encoding session. This function returns an encode session handle which must be used for all subsequent calls to the API functions in the current session.

3.1.1 Initializing encode device

The NVIDIA Encoder supports use of the following types of devices:

3.1.1.1 DirectX 9

The client should create a DirectX 9 device with behavior flags including:
D3DCREATE_FPU_PRESERVE
D3DCREATE_MULTITHREADED
D3DCREATE_HARDWARE_VERTEXPROCESSING
The client should pass a pointer to `IUnknown` interface of the created device (typecast to `void *`) as `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::device`, and set `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::deviceType` to `NV_ENC_DEVICE_TYPE_DIRECTX`. Use of DirectX devices is supported only on Windows 7 and later versions of the Windows OS.

3.1.1.2 DirectX 10

The client should pass a pointer to `IUnknown` interface of the created device typecast to `void *)` as `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::device`, and set `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::deviceType` to `NV_ENC_DEVICE_TYPE_DIRECTX`. Use of DirectX devices is supported only on Windows 7 and later versions of Windows OS.

3.1.1.3 DirectX 11

The client should pass a pointer to `IUnknown` interface of the created device (typecast to `void *)` as `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::device`, and set `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::deviceType` to `NV_ENC_DEVICE_TYPE_DIRECTX`. Use of DirectX devices is supported only on Windows 7 and later versions of Windows OS.

3.1.1.4 CUDA

The client should create a floating CUDA context, and pass the CUDA context handle as `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::device`, and set `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::deviceType` to `NV_ENC_DEVICE_TYPE_CUDA`. Use of CUDA device for Encoding is supported on Linux and Windows 7 and later OS's.

3.2 SELECTING ENCODER CODEC GUID

The client should select an Encoding GUID that represents the desired codec for encoding the video sequence in the following manner:

1. The client should call `NvEncGetEncodeGUIDCount` to get the number of supported Encoder GUIDs from the NVIDIA Video Encoder Interface.
2. The client should use this count to allocate a buffer of sufficient size to hold the supported Encoder GUIDS.
3. The client should then call `NvEncGetEncodeGUIDs` to populate this list.

The client should select a GUID that matches its requirement from this list and use that as the `encodeGUID` for the remainder of the encoding session.

3.3 ENCODER PRESET CONFIGURATIONS

The NVIDIA Encoder Interface exposes various presets to cater to different video encoding use cases. Using these presets will automatically set all relevant encoding parameters. This is a coarse level of control exposed by the API. Specific

attributes/parameters within the preset can be tuned, if required. This is explained in next two subsections.

3.3.1 Enumerating preset GUIDs

The client can enumerate supported Preset GUIDs for the selected `encodeGUID` as follows:

1. The client should call `NvEncGetEncodePresetCount` to get the number of supported Encoder GUIDs.
2. The client should use this count to allocate a buffer of sufficient size to hold the supported Preset GUIDs.
3. The client should then call `NvEncGetEncodePresetGUIDs` to populate this list.

3.3.2 Selecting encoder preset configuration

As mentioned above, the client can use the `presetGUID` for configuring the encode session directly. This will automatically set the hardware encoder with appropriate parameters for particular use-case implied by the preset. If required, the client has the option to fine-tune the encoder configuration parameters in the preset, and override the preset defaults. This approach is often-times more convenient from programming point of view as the programmer only needs to change the configuration parameters which he/she is interested in, leaving everything else pre-configured as per the preset definition.

Here are the steps to fetch a preset encode configuration and optionally change select configuration parameters:

1. Enumerate the supported presets as described above, in Section 3.3.1.
2. Select the preset GUID for which the encode configuration is to be fetched.
3. The client should call `NvEncGetEncodePresetConfig` with the selected `encodeGUID` and `presetGUID` as inputs
4. The required preset encoder configuration can be retrieved through `NV_ENC_PRESET_CONFIG::presetCfg`.
5. Over-ride the default encoder parameters, if required, using the corresponding configuration APIs.

3.4 SELECTING AN ENCODER PROFILE

The client may specify a profile to encode for specific encoding scenario. For example, certain profiles are required for encoding video for playback on iPhone/iPod, encoding video for blue-ray disc authoring, etc.

The client should do the following to retrieve a list of supported encoder profiles:

1. The client should call `NvEncGetEncodeProfileGUIDCount` to get the number of supported Encoder GUIDs from the NVIDIA Video Encoder Interface.
2. The client should use this count to allocate a buffer of sufficient size to hold the supported Encode Profile GUIDS.
3. The client should then call `NvEncGetEncodeProfileGUIDs` to populate this list.

The client should select the profile GUID that best matches the requirement.

3.5 GETTING SUPPORTED LIST OF INPUT FORMATS

NVENCODE API accepts input frames in several different formats, such as YUV and RGB in specific formats, as enumerated in `NV_ENC_BUFFER_FORMAT`.

List of supported input formats can be retrieved as follows:

1. Call `NvEncGetInputFormatCount` and use the count retrieved from `NvEncGetInputFormatCount` to allocate a buffer to hold the list of supported input buffer formats (which are list elements of type `NV_ENC_BUFFER_FORMAT`).
2. Retrieve the supported input buffer formats by calling `NvEncGetInputFormats`.

The client should select a format enumerated in this list for creating input buffers.

3.6 QUERYING CAPABILITY VALUES

NVIDIA video encoder hardware has evolved over multiple generations, with many features being added in each new generation of the GPU. To facilitate application to dynamically figure out the capabilities of the underlying hardware encoder on the system, NVENCODE API provides a dedicated API to query these capabilities. It is a good programming practice to query for support of the desired encoder feature before making use of the feature.

Querying the encoder capabilities can be accomplished as follows:

1. Specify the capability attribute to be queried in `NV_ENC_CAPS_PARAM::capsToQuery` parameter. This should be a member of the `NV_ENC_CAPS` enum.
2. Call `NvEncGetEncodeCaps` to determine support for the required attribute.

Refer to the API reference `NV_ENC_CAPS` enum definition for interpretation of individual capability attributes.

3.7 INITIALIZING THE HARDWARE ENCODER SESSION

The client needs to call `NvEncInitializeEncoder` with a valid encoder configuration specified through `NV_ENC_INITIALIZE_PARAMS` and encoder handle (returned upon successful opening of encode session)

3.8 ENCODE SESSION ATTRIBUTES

3.8.1 Configuring encode session attributes

Encode session configuration is divided into three parts:

3.8.1.1 Session parameters

Common parameters such as input format, output dimensions, display aspect ratio, frame rate, average bitrate, etc. are available in `NV_ENC_INITIALIZE_PARAMS` structure. The client should use an instance of this structure as input to `NvEncInitalizeEncoder`.

The Client must populate the following members of the `NV_ENC_INITIALIZE_PARAMS` structure for the encode session to be successfully initialized:

- `NV_ENC_INITALIZE_PARAMS::encodeGUID`: The client must select a suitable codec GUID as described in Section 3.2.
- `NV_ENC_INITIALIZE_PARAMS::encodeWidth`: The client must specify the desired width of the encoded video.
- `NV_ENC_INITIALIZE_PARAMS::encodeHeight`: The client must specify the desired height of the encoded video.

`NV_ENC_INITALIZE_PARAMS::reportSliceOffsets` can be used to enable reporting of slice offsets. This feature requires `NV_ENC_INITALIZE_PARAMS::enableEncodeAsync` to be set to 0, and does not work with MB-based and byte-based slicing on Kepler GPUs.

3.8.1.2 Advanced codec-level parameters

Parameters dealing with the encoded bit stream such as GOP length, encoder profile, rate control mode, etc. are exposed through the structure `NV_ENC_CONFIG`. The client can pass codec level parameters through `NV_ENC_INITIALIZE_PARAMS::encodeConfig::encodeCodecConfig as` explained below.

3.8.1.3 Advanced codec-specific parameters

Advanced H.264 and HEVC specific parameters are available in structures `NV_ENC_CONFIG_H264` and `NV_ENC_CONFIG_HEVC` respectively.

The client can pass codec-specific parameters through the structure `NV_ENC_CONFIG::encodeCodecConfig`.

3.8.2 Finalizing codec configuration for encoding

3.8.2.1 High-level control using presets

This is the simplest method of configuring the NVIDIA Video Encoder Interface, and involves minimal setup steps to be performed by the client. This is intended for use cases where the client does not need to fine-tune any codec level parameters.

In this case, the client should follow these steps:

1. The client should specify the session parameters as described in Section 3.8.1.1.
2. Optionally, the client can enumerate and select preset GUID that best suits the current use case, as described in Section 3.3.1. The client should then pass the selected preset GUID using `NV_ENC_INITIALIZE_PARAMS::presetGUID`. This helps the NVIDIA Video Encoder interface to correctly configure the encoder session based on the `encodeGUID` and `presetGUID` provided.
3. The client should set the advanced codec-level parameter pointer `NV_ENC_INITIALIZE_PARAMS::encodeConfig::encodeCodecConfig` to NULL.

3.8.2.2 Finer control by overriding preset parameters

The client can choose to edit some encoding parameters on top of the parameters set by the individual preset, as follows:

1. The client should specify the session parameters as described in Section 3.8.1.1.
2. The client should enumerate and select a preset GUID that best suites the current use case, as described in Section 3.3.1. The client should retrieve a preset encode configuration as described in Section 3.3.2.

3. The client may need to explicitly query the capability of the encoder to support certain features or certain encoding configuration parameters. For this, the client should do the following:
4. Specify the capability desired attribute through `NV_ENC_CAPS_PARAM::capsToQuery` parameter. This should be a member of the `NV_ENC_CAPS` enum.
5. Call `NvEncGetEncodeCaps` to determine support for the required attribute. Refer to `NV_ENC_CAPS` enum definition in the API reference for interpretation of individual capability attributes.
6. Select a desired preset GUID and fetch the corresponding Preset Encode Configuration as described in Section 3.3.
7. The client can then override any parameters from the preset `NV_ENC_CONFIG` according to its requirements. The client should pass the fine-tuned `NV_ENC_CONFIG` structure using `NV_ENC_INITIALIZE_PARAMS::encodeConfig::encodeCodecConfig` pointer.
8. Additionally, the client should also pass the selected preset GUID through `NV_ENC_INITIALIZE_PARAMS::presetGUID`. This is to allow the NVIDIA Video Encoder interface to program internal parameters associated with the encoding session to ensure that the encoded output conforms to the client's request. Note that passing the preset GUID here will not override the fine-tuned parameters.

3.8.3 Setting encode session attributes

Once all Encoder settings have been finalized, the client should populate a `NV_ENC_CONFIG` structure, and use it as an input to `NvEncInitializeEncoder` in order to freeze the Encode settings for the current encodes session. Some settings such as rate control mode, average bitrate, resolution etc. can be changed on-the-fly.

The client is required to explicitly specify the following while initializing the Encode Session:

3.8.3.1 Mode of operation

The client should set `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync` to 1 if it wants to operate in asynchronous mode and 0 for operating in synchronous mode.

Asynchronous mode encoding is only supported on Windows 7 and later. Refer to Chapter 6 for more detailed explanation.

3.8.3.2 Picture-type decision

If the client wants to send the input buffers in display order, it must set `enablePTD = 1`.

If the client wants to send the input buffers in encode order, it must set `enablePTD = 0`, and must specify

- `NV_ENC_PIC_PARAMS::pictureType`
- `NV_ENC_PIC_PARAMS_H264/NV_ENC_PIC_PARAMS_HEVC::displayPOCSyntax`
- `NV_ENC_PIC_PARAMS_H264/NV_ENC_PIC_PARAMS_HEVC::refPicFlag`

3.9 CREATING RESOURCES REQUIRED TO HOLD INPUT/OUTPUT DATA

Once the encode session is initialized, the client should allocate buffers to hold the input/output data.

The client may choose to allocate input buffers through NVIDIA Video Encoder Interface by calling `NvEncCreateInputBuffer` API. In this case, the client is responsible to destroy the allocated input buffers before closing the encode session. It is also the client's responsibility to fill the input buffer with valid input data according to the chosen input buffer format.

The client should allocate buffers to hold the output encoded bit stream using the `NvEncCreateBitstreamBuffer` API. It is the client's responsibility to destroy these buffers before closing the encode session.

Alternatively, in scenarios where the client cannot or does not want to allocate input buffers through the NVIDIA Video Encoder Interface, it can use any externally allocated DirectX resource as an input buffer. However, the client has to perform some simple processing to map these resources to resource handles that are recognized by the NVIDIA Video Encoder Interface before use. The translation procedure is explained in Section 4.1.2.

If the client has used a CUDA device to initialize the encoder session, and wishes to use input buffers NOT allocated through the NVIDIA Video Encoder Interface, the client is required to use buffers allocated using the `cuMemAlloc` family of APIs. The NVIDIA Video Encoder Interface version 7.0 only supports `CUdevicePtr` as a supported input format. Support for `CUarray` inputs will be added in future versions.

Note: The client should allocate at least (1 + N_B) input and output buffers, where N_B is the number of B frames between successive P frames.

3.10 RETRIEVING SEQUENCE PARAMETERS

After configuring the encode session, the client can retrieve the sequence parameter information (SPS) at any time by calling `NvEncGetSequenceParams`. It is the client's responsibility to allocate and eventually de-allocate a buffer of size `MAX_SEQ_HDR_LEN` to hold the sequence parameter information.

By default, SPS/PPS data will be attached to every IDR frame. However, the client can request the encoder to generate SPS/PPS data on demand as well. To accomplish this, set `NV_ENC_PIC_PARAMS::encodePicFlags = NV_ENC_PIC_FLAG_OUTPUT_SPSPPS`. The output frame generated for the current input will then include SPS/PPS.

The client can call `NvEncGetSequenceParams` at any time, after the encoder has been initialized (`NvEncInitializeEncoder`) and the session is active.

Chapter 4. ENCODING THE VIDEO STREAM

Once the encode session is configured and input/output buffers are allocated, the client can start streaming the input data for encoding. The client is required to pass a handle to a valid input buffer and a valid bit stream (output) buffer to the NVIDIA Video Encoder Interface for encoding an input picture.

4.1 PREPARING INPUT BUFFERS FOR ENCODING

There are two methods to allocate and pass input buffers to the video encoder.

4.1.1 Input buffers allocated through NVIDIA Video Encoder Interface

If the client has allocated input buffers through `NvEncCreateInputBuffer`, the client needs to fill valid input data before using the buffer as input for encoding. For this, the client should call `NvEncLockInputBuffer` to get a CPU pointer to the input buffer. Once the client has filled input data, it should call `NvUnlockInputBuffer`. The input buffer should be passed to the encoder only after unlocking it. Any input buffers should be unlocked by calling `NvUnlockInputBuffer` before destroying/reallocating them.

4.1.2 Input buffers allocated externally

In order to pass externally allocated buffers to the encoder, follow these steps:

1. Populate `NV_ENC_REGISTER_RESOURCE` with attributes of the externally allocated buffer.
2. Call `NvEncRegisterResource` with the `NV_ENC_REGISTER_RESOURCE` populated in the above step.
3. NvEncRegisterResource returns an opaque handle in NV_ENC_REGISTER_RESOURCE::registeredResource which should be saved.
4. Call `NvEncMapInputResource` with the handle returned above.
5. The mapped handle will then be available in `NV_ENC_MAP_INPUT_RESOURCE::mappedResource.`
6. The client should use this mapped handle (`NV_ENC_MAP_INPUT_RESOURCE::mappedResource`) as the input buffer handle in `NV_ENC_PIC_PARAMS`.
7. After the client has finished using the resource `NvEncUnmapInputResource` must be called.
8. The client must also call `NvEncUnregisterResource` with the handle returned by `NvEncRegisterResource` before destroying the registered resource.

   The mapped resource handle (`NV_ENC_MAP_INPUT_RESOURCE::mappedResource`) should not be used for any other purpose outside the NVIDIA Video Encoder Interface while it is in mapped state. Such usage is not supported and may lead to undefined behavior.

4.2 CONFIGURING PER-FRAME ENCODE PARAMETERS

The client should populate `NV_ENC_PIC_PARAMS` with the parameters to be applied to the current input picture. The client can do the following on a per-frame basis.

4.2.1 Forcing current frame to be encoded as intra frame

To force the current frame as intra (I) frame, set

```
NV_ENC_PIC_PARAMS::encodePicFlags = NV_ENC_PIC_FLAG_FORCEINTRA
```

4.2.2 Forcing current frame to be used as a reference frame

To force the current frame to be used as a reference frame, set

```
NV_ENC_PIC_PARAMS_H264/NV_ENC_PIC_PARAMS_HEVC::refPicFlag = 1
```

4.2.3 Forcing current frame to be used as an IDR frame

To force the current frame to be encoded as IDR frame, set

```
NV_ENC_PIC_PARAMS::encodePicFlags = NV_ENC_PIC_FLAG_FORCEIDR
```

4.2.4 Requesting generation of sequence parameters

To include SPS/PPS along with the currently encoded frame, set

```
NV_ENC_PIC_PARAMS::encodePicFlags = NV_ENC_PIC_FLAG_OUTPUT_SPSPPS
```

4.3 SUBMITTING INPUT FRAME FOR ENCODING

The client should call `NvEncEncodePicture` to perform encoding.

The input picture data will be taken from the specified input buffer, and the encoded bit stream will be available in the specified bit stream (output) buffer once the encoding process completes.

Codec-agnostic parameters such as timestamp, duration, input buffer pointer, etc. are passed via the structure `NV_ENC_PIC_PARAMS` while codec-specific parameters are passed via the structure `NV_ENC_PIC_PARAMS_H264`/`NV_ENC_PIC_PARAMS_HEVC` depending upon the codec in use.

The client should specify the codec-specific structure in `NV_ENC_PIC_PARAMS` using the `NV_ENC_PIC_PARAMS::codecPicParams` member.

4.4 RETRIEVING ENCODED OUTPUT

Upon completion of the encoding process for an input picture, the client is required to call `NvEncLockBitstream` to get a CPU pointer to the encoded bit stream. The client can make a local copy of the encoded data or pass the CPU pointer for further processing (e.g. to a media file writer).

The CPU pointer will remain valid until the client calls `NvUnlockBitstreamBuffer`. The client should call `NvUnlockBitstreamBuffer` after it completes processing the output data.

The client must ensure that all bit stream buffers are unlocked before destroying/de-allocating them (e.g. while closing an encode session) or even before reusing them again as output buffers for subsequent frames.

Chapter 5. END OF ENCODING

5.1 NOTIFYING THE END OF INPUT STREAM

To notify the end of input stream, the client must call `NvEncEncodePicture` with the flag `NV_ENC_PIC_PARAMS:: encodePicFlags` set to `NV_ENC_FLAGS_EOS` and all other members of `NV_ENC_PIC_PARAMS` set to 0. No input buffer is required while calling `NvEncEncodePicture` for EOS notification.

EOS notification effectively flushes the encoder. This can be called multiple times in a single encode session. This operation however must be done before closing the encode session.

5.2 RELEASING RESOURCES

Once encoding completes, the client should destroy all allocated resources.

The client should call `NvEncDestroyInputBuffer` if it had allocated input buffers through the NVIDIA Video Encoder Interface. The client must ensure that input buffer is first *unlocked* by calling `NvUnlockInputBuffer` before destroying it.

The client should call `NvEncDestroyBitStreamBuffer` to destroy each bitstream buffer it had allocated. The client must ensure that the bitstream buffer is first *unlocked* by calling `NvEncUnlockBitstream` before destroying it.

5.3 CLOSING ENCODE SESSION

The client should call `NvEncDestroyEncoder` to close the encoding session. The client should ensure that all resources tied to the encode session being closed have been destroyed before calling `NvEncDestroyEncoder`. These include input buffers, bit stream buffers, SPS/PPS buffer, etc.

It must also ensure that all registered events are unregistered, and all mapped input buffer handles are unmapped.

Chapter 6. MODES OF OPERATION

The NVIDIA Video Encoder Interface supports the following two modes of operation.

6.1 ASYNCHRONOUS MODE (WINDOWS 7 AND ABOVE)

This mode of operation is used for asynchronous output buffer processing. For this mode, the client allocates an event object and associates the event with an allocated output buffer. This event object is passed to the NVIDIA Encoder Interface as part of the `NvEncEncodePicture` API. The client can wait on the event in a separate thread. When the event is signaled, the client calls the NVIDIA Video Encoder Interface to copy output bitstream produced by the encoder. Note that the encoder support asynchronous mode of operation only for Windows 7 and above. In Linux, ONLY synchronous mode is supported (refer to Section 6.2.)

The client should set the flag `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync` to 1 to indicate that it wants to operate in asynchronous mode. After creating the event objects (one object for each output bitstream buffer allocated), the client needs to register them with the NVIDIA Video Encoder Interface using the `NvEncRegisterAsyncEvent`. The client is required to pass a bitstream buffer handle and the corresponding event handle as input to `NvEncEncodePicture`. The NVIDIA Video Encoder Interface will signal this event when the hardware encoder finishes encoding the current input data. The client can then call `NvEncLockBitstream` in non-blocking mode `NV_ENC_LOCK_BITSTREAM::doNotWait` flag set to 1 to fetch the output data.

The client should call `NvEncUnregisterAsyncEvent` to unregister the Event handles before destroying the event objects. Whenever possible, NVIDIA recommends using the asynchronous mode of operation instead of synchronous mode.

A step-by-step control flow for asynchronous mode is as follows:

1. When working in asynchronous mode, the output sample must consist of an event + output buffer and clients must work in multi-threaded manner (D3D9 device should be created with MULTITHREADED flag).
2. The output buffers are allocated using `NvEncCreateBitstream` API. The NVIDIA Video Encoder Interface will return an opaque pointer to the output memory in `NV_ENC_CREATE_BITSTREAM_BUFFER::bitstreambuffer`. This opaque output pointer should be used in `NvEncEncodePicture` and `NvEncLockBitsteam`/ `NvEncUnlockBitsteam` calls. For accessing the output memory using CPU, client must call `NvEncLockBitsteam` API. The number of IO buffers should be at least 4 + number of B frames.
3. The events are windows event handles allocated using Windows' `CreateEvent` API and registered using the function `NvEncRegisterAsyncEvent` before encoding. The registering of events is required only once per encoding session. Clients must unregister the events using `NvEncUnregisterAsyncEvent` before destroying the event handles. The number of event handles must be same as number of output buffers as each output buffer is associated with an event.
4. Client must create a secondary thread in which it can wait on the completion event and copy the bitstream data from the output sample. Client will have two threads: one is the main application thread which submits encoding work to NVIDIA Encoder while secondary thread waits on the completion events and copies the compressed bitstream data from the output buffer.
5. Client must send the event and output buffer in `NV_ENC_PIC_PARAMS::outputBitstream` and `NV_ENC_PIC_PARAMS::completionEvent` Fields, respectively, as part of `NvEncEncodePicture` API call.
6. Client should then wait on the event on the secondary thread in the same order in which it has called `NvEncEncodePicture` calls irrespective of input buffer re-ordering (encode order != display order). NVIDIA Encoder takes care of the reordering in case of B frames and should be transparent to the encoder clients.
7. When the event gets signalled client must send down the output buffer of sample event it was waiting on in `NV_ENC_LOCK_BITSTREAM::outputBitstream` field as part of `NvEncLockBitstream` call.
8. The NVIDIA Encoder Interface returns a CPU pointer and bitstream size in bytes as part of the `NV_ENC_LOCK_BITSTREAM`.
9. After copying the bitstream data, client must call `NvEncUnlockBitstream` for the locked output bitstream buffer.

Note:

1. The client will receive the event's signal and output buffer in the same order in which they were queued.
2. The `NV_ENC_LOCK_BITSTREAM::pictureType` notifies the output picture type to the clients.
3. Both, the input and output sample (output buffer and the output completion event) are free to be reused once the NVIDIA Video Encoder Interface has signalled the event and the client has copied the data from the output buffer.

6.2 SYNCHRONOUS MODE

This mode of operation is used for synchronous output buffer processing. In this mode the client makes a blocking call to the NVIDIA Video Encoder Interface to retrieve the output bitstream data from the encoder. The client sets the flag `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync` to 0 for operation in synchronous mode. The client then must call `NvEncEncodePicture` without setting a completion event handle. The client must call `NvEncLockBitstream` with flag `NV_ENC_LOCK_BITSTREAM::doNotWait` set to 0, so that the lock call blocks until the hardware encoder finishes writing the output bitstream. The client can then operate on the generated bitstream data and call `NvEncUnlockBitstream`. This is the only mode supported on Linux.

6.3 THREADING MODEL

In order to get maximum performance for encoding, the encoder client should create a separate thread to wait on events or when making any blocking calls to the encoder interface.

The client should avoid making any blocking calls from the main encoder processing thread. The main encoder thread should be used only for encoder initialization and to submit work to the HW Encoder using `NvEncEncodePicture` API, which is non-blocking.

Output buffer processing, such as waiting on the completion event in asynchronous mode or calling the blocking API's such as `NvEncLockBitstream`/`NvEncUnlockBitstream` in synchronous mode, should be done in the secondary thread. This ensures that the main encoder thread is never blocked except when the encoder client runs out of resources.

It is also recommended to allocate a large number of input and output buffers in order to avoid resource hazards and improve overall encoder throughput.

Chapter 7. MOTION-ESTIMATION-ONLY MODE

NVENC can be used as a hardware accelerator to perform motion search and generate motion vectors and mode information only. The resulting motion vectors or mode decisions can used, for example, in motion compensated filtering or for supporting other codecs not fully supported by NVENC or simply as motion vector hints for a custom encoder. The procedure to use the feature is explained below.

7.1 QUERY MOTION-ESTIMATION ONLY MODE CAPABILITY

Before using the motion-estimation (ME) only mode, the client should explicitly query the capability of the encoder to support ME only mode. For this, the client should do the following:

1. Specify the capability attribute as `NV_ENC_CAPS_SUPPORT_MEONLY_MODE` to query through the `NV_ENC_CAPS_PARAM::capsToQuery` parameter.
2. The client should call `NvEncGetEncoderCaps` to determine support for the required attribute.

`NV_ENC_CAPS_SUPPORT_MEONLY_MODE` indicates support of ME only mode in hardware.

0: ME only mode not supported.

1: ME only mode supported.

7.2 CREATE RESOURCES FOR INPUT/OUTPUT DATA

The client should allocate at least one buffer for the input picture by calling `NvEncCreateInputBuffer` API, and should also allocate one buffer for the reference frame by using `NvEncCreateInputBuffer` API. The client can refer to Section 4.1.2 of the programming guide if they want to use any externally allocated DirectX resource as input buffer. The client is responsible for filling in valid input data.

After input resources are created, client needs to allocate resources for the output data by using `NvEncCreateMVBuffer` API.

7.3 RUN MOTION ESTIMATION

The Client should create an instance of `NV_ENC_MEONLY_PARAMS`.

The pointers of the input picture buffer and the reference frame buffer need to be fed to `NV_ENC_MEONLY_PARAMS::inputBuffer` and `NV_ENC_MEONLY_PARAMS::referenceFrame` respectively.

The pointer returned by `NvEncCreateMVBuffer` API in the `NV_ENC_CREATE_MV_BUFFER::mvBuffer` field needs to be fed to `NV_ENC_MEONLY_PARAMS::mvBuffer`.

In order to operate in asynchronous mode, the client should create an event and pass this event in `NV_ENC_MEONLY_PARAMS::completionEvent`. This event will be signaled upon completion of motion estimation. Each output buffer should be associated with a distinct event pointer.

Client should call `NvEncRunMotionEstimationOnly` to run the motion estimation on hardware encoder.

For asynchronous mode client should wait for motion estimation completion signal before reusing output buffer and application termination.

Client must lock `NV_ENC_CREATE_MV_BUFFER::mvBuffer` using `NvEncLockBitstream` to get the motion vector data.

Finally, `NV_ENC_LOCK_BITSTREAM::bitstreamBufferPtr` which contains the output motion vectors should be typecast to `NV_ENC_H264_MV_DATA*/NV_ENC_HEVC_MV_DATA*` for H.264/HEVC respectively. Client should then unlock `NV_ENC_CREATE_MV_BUFFER::mvBuffer` by calling `NvEncUnlockBitstream`.

7.4 RELEASE THE CREATED RESOURCES

Once the usage of motion estimation is done, the client should call `NvEncDestroyInputBuffer` to destroy the input picture buffer and the reference frame buffer and should call `NvEncDestroyMVBuffer` to destroy the motion vector data buffer.

Chapter 8. ADVANCED FEATURES AND SETTINGS

8.1 LOOK-AHEAD

Look-ahead improves the video encoder's rate control accuracy by enabling the encoder to buffer the specified number of frames, estimate their complexity and allocate the bits appropriately among these frames proportional to their complexity.

To use this feature, the client must follow these steps:

1. The availability of the feature in the current hardware can be queried using `NvEncGetEncodeCaps` and checking for `NV_ENC_CAPS_SUPPORT_LOOKAHEAD`.
2. Look-ahead needs to be enabled during initialization by setting `NV_ENC_INITIALIZE_PARAMS::encodeconfig->rcParams.enableLookahead` = 1.
3. The number of frames to be looked ahead should be set in `NV_ENC_INITIALIZE_PARAMS::encodeconfig->rcParams.lookaheadDepth` which can be up to 32.
4. By default, look-ahead enables adaptive insertion of intra frames and B frames. They can however be disabled by setting

   `NV_ENC_INITIALIZE_PARAMS::encodeconfig->rcParams.disableIadapt` and/or `NV_ENC_INITIALIZE_PARAMS::encodeconfig->rcParams.disableBadapt` to 1.
5. When the feature is enabled, frames are queued up in the encoder and hence `NvEncEncodePicture` will return `NV_ENC_ERR_NEED_MORE_INPUT` until the encoder has sufficient number of input frames to satisfy the look-ahead requirement. Frames should be continuously fed in until `NvEncEncodePicture` returns `NV_ENC_SUCCESS`.

8.2 TEMPORAL ADAPTIVE QUANTIZATION (T-AQ)

This feature tries to adjust encoding QP based on temporal characteristics of the sequence. Temporal AQ improves the quality of encoded frames by adjusting QP for regions which are constant or have low motion across frames but have high spatial detail, such that they become better reference for future frames. Allocating extra bits to such regions in reference frames is better than allocating them to the residuals in referred frames because it helps improve the overall encoded video quality. If majority of the region within a frame has little or no motion, but has high spatial details (e.g. high-detail non-moving background) enabling temporal AQ will benefit the most.

One of the potential disadvantages of temporal AQ is that enabling temporal AQ *may* result in high fluctuation of bits consumed per frame within a GOP. I/P-frames will consume more bits than average P-frame size and B-frames will consume lesser bits. Although target bitrate will be maintained at the GOP level, the frame size will fluctuate from one frame to next within a GOP more than it would without temporal AQ. If a strict CBR profile is required for every frame size within a GOP, it is not recommended to enable temporal AQ. Additionally, since some of the complexity estimation is performed in CUDA, there may be some performance impact when temporal AQ is enabled.

To use temporal AQ, follow these steps in your application.

- Query the availability of temporal AQ for the current hardware by calling the API `NvEncGetEncodeCaps` and checking for `NV_ENC_CAPS_SUPPORT_TEMPORAL_AQ`.
- If supported, temporal AQ can be enabled during initialization by setting `NV_ENC_INITIALIZE_PARAMS::encodeconfig->rcParams.enableTemporalAQ = 1`.

8.3 HIGH BIT DEPTH ENCODING

All NVIDIA GPUs support 8-bit encoding (RGB/YUV input with 8-bit precision). Some of the NVIDIA GPUs support high-bit-depth HEVC encoding (HEVC main-10 profile with 10-bit input precision). To encode 10-bit content the following steps are to be followed.

1. The availability of the feature can be queried using `NvEncGetEncodeCaps` and checking for `NV_ENC_CAPS_SUPPORT_10BIT_ENCODE`.
2. Create the encoder session with `NV_ENC_HEVC_PROFILE_MAIN10_GUID`.
3. During encoder initialization, set `encodeConfig->encodeCodecConfig.hevcConfig.pixelBitDepthMinus8 = 2`

4. The input surface format needs be set to `NV_ENC_BUFFER_FORMAT_YUV420_10BIT` OR `NV_ENC_BUFFER_FORMAT_ABGR10` OR `NV_ENC_BUFFER_FORMAT_ARGB10` or `NV_ENC_BUFFER_FORMAT_YUV444_10BIT`, depending upon nature of input.
5. Other encoding parameters such as preset, rate control mode etc. can be set as desired.

8.4 ENCODER FEATURES USING CUDA

Although the core video encoder hardware on GPU is completely independent of CUDA cores or graphics engine on the GPU, following encoder features internally use CUDA for hardware acceleration. Note that the impact of enabling these features on overall CUDA or graphics performance is minimal and this list is provided purely for information purposes.

- Two-pass rate control modes for high quality presets.
- Look-ahead
- All adaptive quantization modes.
- Encoding with inputs in RGB formats.

Chapter 9. RECOMMENDED NVENC SETTINGS

NVIDIA hardware video encoder is used for several purposes in various applications. Some of the common applications include: Video-recording (archiving), game-casting (broadcasting/multicasting video gameplay online), transcoding (live and video-on-demand) and streaming (games or live content). Each of these use-cases has its unique requirements for quality, bitrate, latency tolerance, performance constraints etc. Although NVIDIA Encoder Interface provides flexibility to control the settings with a large number of API's, below table can be used as a general guideline for recommended settings for some of the popular use-cases to deliver the best encoded bit stream quality. These recommendations are particularly applicable to GPUs based on second generation Maxwell architecture beyond. For earlier GPUs (Kepler and first generation Maxwell), it is recommended that clients use the Table 1 as a starting point and adjust the settings to achieve appropriate performance-quality tradeoff.

Table 1. Recommended NVENC settings for various use-cases

Use-case	Recommended settings for optimal quality and performance
Recording/Archiving	• High Quality preset • Rate control mode = VBR • Very large VBV buffer size (4 seconds) • B Frames[1] • Finite GOP length (2 seconds) • Adaptive quantization[2] (AQ) enabled
Game-casting & cloud transcoding	• High Quality preset • Rate control mode = Two-pass CBR • Look-ahead (with dynamically inserted I and B frames) • Medium VBV buffer size (1 second) • B Frames[1] • Finite GOP length (2 seconds) • Adaptive quantization[2] (AQ) enabled
Low-latency use cases like game-streaming, video conferencing etc.	• Low-Latency High Quality preset • Rate control mode = Two-pass CBR • Very low VBV buffer size (Single frame) • No B Frames • Infinite GOP length • Adaptive quantization[2] (AQ) enabled

[1] Recommended for low motion games and natural video. It is observed that 3 B frames results in most optimal quality

[2] Available only in second generation Maxwell GPUs and above. Temporal AQ in general gives better quality than Spatial AQ but is computationally complex.

Notice

HDMI

OpenCL

Trademarks

Copyright