NVENC - NVIDIA VIDEO ENCODER INTERFACE

NVENC_VideoEncoder_API_PG-06155-001_v04 | July 2014

Programming Guide

REVISION HISTORY

Revision	Date	Author	Description
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

TABLE OF CONTENTS

NVIDIA VIDEO ENCODER INTERFACE

INTRODUCTION

NVIDIA's generation of GPUs based on the Kepler and Maxwell architectures contain a hardware-based H.264 video encoder (referred to as NVENC). The NVENC hardware takes YUV as input, and generates a H.264 video bitstream. NVENC hardware's encoding capabilities are accessed using the NVENC API. This document provides information on how to program the NVENC using the APIs exposed in the SDK.

Developers should have a basic understanding of the H.264 Video Codec, and be familiar with either Windows or Linux development.

1. NVIDIA VIDEO ENCODER INTERFACE

Developers can create a client application that calls NVENC API functions within `nvEncodeAPI.dll` for Windows OSes or `libnvidia-encode.so` for Linux OSes. These libraries are installed as part of the NVIDIA driver. The client application can either link at run-time using `LoadLibrary()` on Windows OS or `dlopen()` for Linux OS. The binary is copied to the system (.\System32) during the driver installation process.

The client's first interaction with the NVIDIA Video Encoder Interface is to call `NvEncodeAPICreateInstance`. This populates the input / output buffer passed to `nvEncodeAPICreateInstance` with pointers to functions that implement the functionality provided in the interface.

2. SETTING UP THE HARDWARE FOR ENCODING

2.1 Opening an Encode Session

After loading the NVENC Interface, the client should first call `NvEncOpenEncodeSessionEx API`to open an encoding session. This function provides an encode session handle to the client, and that handle must be used for all further API calls in the current session.

2.1.1 Initializing the Encode Device

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

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 the 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 OS.

2) **DirectX 10:**

   The client should pass a pointer to the 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 OS.

3) **DirectX 11:**

   The client should pass a pointer to the 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 OS.

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 Windows XP and Linux, in addition to Windows 7 and later, from NVIDIA Video Encoder Interface version 2.0 onwards.

2.2 Selecting an Encoder GUID

The client calls the NVIDIA Encoder Interface to select an Encoding GUID that represents the desired codec for encoding the video sequence in the following manner:

i) The client should call `NvEncGetEncodeGUIDCount` to get the number of supported Encoder GUIDs from the NVIDIA Video Encoder Interface.

ii) The client should use this count to allocate a buffer of sufficient size to hold the supported Encoder GUIDS.

iii) 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.

2.3 Querying Capability Values

In case the client needs to explicitly to query the features supported by the Encoder, the following should be done

i) The client is required to specify the capability attribute it wants to query through the `NV_ENC_CAPS_PARAM::capsToQuery` parameter. This should be a member of the `NV_ENC_CAPS` enum.

ii) The client should call `NvEncGetEncoderCaps` to determine support for the required attribute.

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

2.4 Encoder Preset Configurations

The NVIDIA Encoder Interface exposes various presets to cater to different use cases which can be used by the client. Using presets through the Encoder Interface will automatically set all of the relevant encoding parameters. This is a coarse level of control exposed by the NVIDIA Encoder Interface to the client.

2.4.1 Enumerating Preset GUIDs

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

i) The client should call `NvEncGetEncodePresetCount` to get the number of supported Encoder GUIDs from the NVIDIA Video Encoder Interface.

ii) The client should use this count to allocate a buffer of sufficient size to hold the supported Preset GUIDS.

iii) The client should then call `NvEncGetEncodePresetGUIDs` to populate this list.

2.4.2 Fetching Preset Encoder Configuration

The client can either use the preset GUID for configuring the encode session or it can fine-tune the encoder configuration corresponding to a preset GUID. This mechanism provides a client the ability to fine tune parameter values by overriding the preset defaults.

The client should follow these steps to fetch a Preset Encode configuration:

i) The client should enumerate the supported presets as described above, in the section "2.4.1 Enumerating Preset GUIDs"

ii) The client should select a Preset GUID for which the encode configuration is to be fetched.

iii) The client should call `NvEncGetPresetConfig` with the selected EncodeGUID and PresetGUID as inputs

iv) The required preset encoder configuration can be retrieved through `NV_ENC_PRESET_CONFIG::presetCfg`.

v) This gives the client the flexibility to over-ride the encoder parameters that defaulted by the specific preset.

2.5 Selecting an Encoder Profile

The client may specify a profile to encode for a specific encoder profile.

(Eg: encoding video for playback on iPod, encoding video for BD authoring, etc.)

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

i) The client should call `NvEncGetEncodeProfileGUIDCount` to get the number of supported Encoder GUIDs from the NVIDIA Video Encoder Interface.

ii) The client should use this count to allocate a buffer of sufficient size to hold the supported Encode Profile GUIDS.

iii) The client should then call `NvEncGetEncodeProfileGUIDs` to populate this list.

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

2.6 Getting Supported Input Format list

The client should follow these steps to retrieve the list of supported input formats:

i) The client should call `NvEncGetInputFormatCount`.

ii) The Client should use the count retrieved from `NvEncGetInputFormatCount` to allocate a buffer to hold the list of supported input buffer formats [list elements of type `NV_ENC_BUFFER_FORMAT`].

iii) The client should then populate this list by calling `NvEncGetInputFormats`.

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

2.7 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 on successful opening of encode session)

2.7.1 Configuring Encode Session Attributes

Encode session Configuration is divided into three parts:

2.7.1.1 Session Parameters

Common parameters such as output dimensions, input format, 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` struct for the Encode session to be successfully initialized:

i) `NV_ENC_INITALIZE_PARAMS::encodeGUID` : The client must select a suitable codec GUID as described in section "2.2 Selecting an Encoder GUID"

ii) `NV_ENC_INITIALIZE_PARAMS::encodeWidth` : The client must specify the desired width of the encoded video.

iii) `NV_ENC_INITIALIZE_PARAMS::encodeHeight`: The client must specify the desired height of the encoded video.

2.7.1.2 Advanced Codec-level parameters

Parameters dealing with the encoded bitstream such as GOP length, encoder profile, Rate Control mode, etc are exposed through the `NV_ENC_CONFIG` structure. The client can pass codec level parameters through `NV_ENC_INITIALIZE_PARAMS::codecConfig`.

2.7.1.3 Advanced Codec-specific parameters

Advanced codec-specific parameters are available in `NV_ENC_CONFIG_XXXX` structures.

Eg: H.264-specific parameters are available in `NV_ENC_CONFIG_H264`.

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

2.7.2 Finalizing the Codec Configuration for Encoding

2.7.2.1 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:

i) The client should specify the session parameters as described in section "2.7.1.1 Session Parameters"

ii) Optionally, the client can enumerate and select Preset GUID that best suites the current use case, as described in section "2.4.1 Enumerating Preset GUIDs" 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. Hence, this step is recommended.

iii) The client should set the advanced codec-level parameter pointer `NV_ENC_INITIALIZE_PARAMS::codecParams` to NULL

2.7.2.2 Coarse Control by overriding Preset Parameters

The client can choose to edit some encoding parameters, but may not want to populate the complete encode configuration from scratch. In this case, the client is advised to follow these steps:

i) The client should specify the session parameters as described in section "2.7.1.1 Session Parameters"

ii) The client should enumerate and select a Preset GUID that best suites the current use case, as described in section "2.4.1 Enumerating Preset GUIDs" The client should retrieve a Preset encode configuration as described in section "2.4.2 Fetching Preset Encoder Configuration"

iii) 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:

iv) The client is required to specify the capability attribute it wants to query through the NV_ENC_CAPS_PARAM::capsToQuery parameter. This should be a member of the NV_ENC_CAPS enum.

v) The client should call NvEncGetEncoderCaps to determine support for the required attribute. Refer to NV_ENC_CAPS enum definition in the API reference for interpretation of individual capability attributes.

vi) The client should then select a desired Preset GUID and fetch the corresponding Preset Encode Configuration as described in: "2.4 Getting Preset Encoder Configurations".

vii) The client can 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::codecConfig` pointer.

viii) 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. Passing the preset GUID will not override the fine-tuned parameters.

2.7.3 Setting Encode Session Attributes

Once all Encoder settings have been finalized, the client should populate a `NV_ENC_CONFIG` struct, 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, can be changed on-the-fly.

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

2.7.3.1 Mode of Operation

The client should set `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync` to 1 if it wants to operate in Asynchronous mode. Set it to 0 for operating in Synchronous mode. Asynchronous mode encoding is only supported on Windows 7 and later Windows OS. Refer to section 5 for more detailed explanation.

2.7.3.2 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_H26::.displayPOCSyntax
- NV_ENC_PIC_PARAMS_H264 ::refPicFlag

Note: `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync` == 0 and `NV_ENC_INITIALIZE_PARAMS:: enablePTD` == 1 are not a compatible combination.

If the client needs to use synchronous mode of operation, the client should also take care of picture-type decision; i.e. if `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync` == 0 then `NV_ENC_INITIALIZE_PARAMS::enablePTD` should also be 0.

2.8 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 the NVIDIA Video Encoder Interface, by calling `NvEncCreateInputBuffer` API. The input buffer width and height should be 32-aligned. 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 bitstream using the `NvCreateBitstreamBuffer` 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 "3.1.2 Input Buffers Allocated Externally"

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 2.0 only supports CUdevicePtr as input. Support for CUarray inputs will be added future version.

Note: The client should allocate at least [1 + No. of B-Frames] Input/Output buffers.

2.9 Retrieving Sequence Parameters

After configuring the Encode Session, the client can retrieve the Sequence parameter information at any time by calling `NvEncGetSequenceParams`. The client must allocate a buffer of size `NV_MAX_SEQ_HDR_LEN` to hold the Sequence parameters. It is the client's responsibility to manage this memory.

By default, SPS PPS data will be attached to every IDR frame. However, the client can request the NVIDIA Video Encoder Interface to generate SPS PPS data on the fly as well. For this, the client must `set NV_ENC_PIC_PARAMS::encodeFlags` to `NV_ENC_FLAGS_OUTPUT_SPSPPS`. The output frame generated for the current input will then contain SPS PPS data attached to it.

The client can call `NvEncGetSequenceParams` at any time during the encoding session, after it has called `NvEncInitializeEncoder`.

3. 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 bitstream buffer to the NVIDIA Video Encoder Interface for encoding an input picture.

3.1 Preparing Input Buffer for Encoding:

3.1.1 Input Buffers Allocated through the 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 client should use the input buffer for encode only after unlocking it. The client must also take care to unlock any locked input buffer before destroying it.

3.1.2 Input Buffers Allocated Externally

If the client is using externally allocated buffers as input, the client is required to call `NVEncRegisterResource` before use with the NVIDIA Video Encoder Interface. The client should also explicitly call `NvEncUnregisterResource` with this handle before destroying the resource. The client should call `NvEncMapInputResource` to retrieve a handle to the resource that is understandable by the NVIDIA Video Encoder Interface. Note that the client is required to pass the registered handle as `NV_ENC_MAP_INPUT_RESOURCE::registeredResource.` The mapped handle will be made available in `NV_ENC_MAP_INPUT_RESOURCE::mappedResource`. The client should use this mapped handle as the input buffer handle in `NV_ENC_PIC_PARAM`. The client should call `NvEncUnmapInputResource` after it has finished using the resource as an input to the NVIDIA Video Encoder Interface. The resource 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.

3.2 Configuring Per-Frame Encode Parameters

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

3.2.1 Forcing the Current Frame to be used as Intra-Predicted Reference Frame

The client should set `NV_ENC_PIC_PARAMS::encodeFlags` to `NV_ENC_FLAGS_FORCEINTRA`.

3.2.2 Forcing the Current Frame to be used as a Reference Frame

The client should set `NV_ENC_PIC_PARAMS_H264::refPicFlag` to 1

3.2.3 Forcing current frame to be used as an IDR frame

The client should set `NV_ENC_PIC_PARAMS_H264::forceIDR` to 1.

3.2.4 Requesting Generation of Sequence parameters

The client should set `NV_ENC_PIC_PARAMS::encodeFlags` to `NV_ENC_FLAGS_OUTPUT_SPSPPS`.

3.2.6 Enforcing a Constant QP

The client should set the flag `NV_ENC_PIC_PARAMS::userForcedConstQP` to 1 and specify the required RateControl QP value in `NV_ENC_RC_PARAMS:: constQP`. This request will be honored only if the Encode Session is already running with rate control mode `NV_ENC_PARAMS_RC_CONSTQP`, or the rate control mode is being set to `NV_ENC_PARAMS_RC_CONSTQP` in the current operation.

3.3 Submitting Input 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 bitstream will be available in the specified bitstream buffer once the encoding process completes.

Common parameters such as timestamp, duration, input buffer pointer, etc are available in `NV_ENC_PIC_PARAMS` while Codec-specific parameters are available in `NV_ENC_PIC_PARAMS_XXXX` structures.

For example, H.264-specific parameters are available in `NV_ENC_PIC_PARAMS_H264`.

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

3.4 Generating 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 choose to make a local copy of the encoded data or pass on the CPU pointer 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 them (while closing an encode session) or even before reusing it again as an output buffer.

4. END OF ENCODING

4.1 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::encodeFlags` set to `NV_ENC_FLAGS_EOS`. This must be done before closing the Encode Session.

When notifying End of Stream, the client should set all other members of `NV_ENC_PIC_PARAMS` to 0. No input buffer is required in case of EOS notification.

EOS notification effectively flushes the encoder. This can be called multiple times in a single encode session.

4.2 Releasing the Created 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 be sure not to destroy a buffer while it is locked.

The client should call `NvEncDestroyBitStreamBuffer` to destroy each bitstream buffer it had allocated. The client must be sure not to destroy a bitstream buffer while it is locked.

4.3 Closing the Encode Session

The client should call `NvEncDestroyEncodeSession` to close the encoding session. The client should ensure that all resources tied to the encode session being closed have been destroyed before calling `NvEncDestroyEncodeSession`. These include Input buffers, bitstream buffers, SPSPPS buffer, etc.

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

5. MODES OF OPERATION

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

5.1 Asynchronous Mode

This mode of operation is used for asynchronous output buffer processing. It is also 0 event driven. 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 bitstream output produced by the HW encoder. Note: For NVIDIA Video Encoder Interface versions 1.0, 2.0 and 3.0 can support asynchronous mode of operation for Windows only. In Linux, only synchronous mode is supported (refer to Section "5.2 Synchronous Mode")

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 corresponds to each bitstream buffer it creates], the client needs to register them with the NVIDIA Video Encoder Interface through the `NvEncRegisterAsyncEvent` API. The client is required to pass a bitstream buffer handle and an event handle as input to `NvEncEncodePicture`. The NVIDIA Video Encoder Interface will signal this event when the HW 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. This is the preferred mode of operation.

A step-by-step control flow would be:

i) 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).

ii) 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.

iii) The events are windows event handles allocated using win32 CreateEvent API and registered with NvEncodeAPI calling 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.

iv) 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 2 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.

v) Client must send both the event and output buffer in `NV_ENC_PIC_PARAMS::outputBitstream` and `NV_ENC_PIC_PARAMS::completionEvent`
fields as part of `NvEncEncodePicture` API call.

vi) Client should then wait on the event on the secondary thread in the same order in which they have 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.

vii) When the event gets signalled then client must send down the output buffer of the output sample on whose event it was waiting on in `NV_ENC_LOCK_BITSTREAM::outputBitstream` field as part of `NvEncLockBitstream`.

viii) The NVIDIA Encoder Interface returns a CPU pointer and bitstream size in bytes as part of the `NV_ENC_LOCK_BITSTREAM`.

ix) After copying the bitstream data, client must call `NvEncUnlockBitstream` for the locked output bitstream buffer.

Note:

- The client will receive the events signal and output buffer in the same order in which they have been queued.
- The LockBitstream parameter struct has a picture type field which will notify the output picture type to the clients.
- 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.

5.2 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 HW Encoder finishes writing the output bitstream. The client can then operate on the generated bitstream data and call `NvEncUnlockBitstream`.

It is important to note that if the client wants to use Synchronous mode for encoding with B-frames, the client must also set `NV_ENC_CONFIG::enablePTD` to 0, i.e. the client must also handle the picture-type decision, and send a valid picture input to the NVIDIA Encoder Interface.

6. 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 `NvEncLockBitstream`/`NvEncUnlockBitstream` blocking API in synchronous mode, should be done on the secondary thread. This ensures the main encoder thread is never blocked except when the encoder client runs out of resources.

7. USING ADDITIONAL FEATURES

7.1 Low Latency Encoding

The following section describes the NVIDIA recommended settings for H.264 encoder for low-latency applications such as cloud gaming, real-time streaming etc.

7.1.1 Low Latency Settings

The VBV buffer size determines the maximum size of a coded picture. Typical VBV buffer size values are in the range of 0.5*maxBitrate to 3.0*maxBitrate, which corresponds to a maximum transmission delay of 500ms to 3s over a constant bitrate channel. We would recommend the clients to use 2 Pass Rate control modes for low latency scenarios.

For low-latency applications, the lowest transmission delay can be achieved by setting the VBV buffer size to the average frame size. In this case, the GOP length should be set to NVENC_INFINITE_GOPLENGTH, as such a small VBV buffer size severely restricts the quality of I-frames. When the rate control mode is set to NV_ENC_PARAMS_RC_2_PASS_QUALITY, the encoder is allowed to exceed the VBV buffer size at scene changes (and I frames) in order to minimize the coding artifacts resulting from the limited maximum frame size (When the rate control mode is NV_ENC_RC_2_PASS_FRAMESIZE_CAP, the coded picture size is never allowed to exceed the VBV buffer size).

```
uint32_t maxFrameSize = B/N;
NV_ENC_RC_PARAMS::vbvBufferSize= maxFrameSize;
NV_ENC_RC_PARAMS::vbvInitialDelay= maxFrameSize;
NV_ENC_RC_PARAMS::maxBitRate= NV_ENC_CONFIG::vbvBufferSize *N; // where
N is the encoding frame rate.
NV_ENC_RC_PARAMS::averageBitRate=
 NV_ENC_RC_PARAMS::vbvBufferSize *N; // where N is the encoding frame
rate.
NV_ENC_RC_PARAMS::rateControlMode= NV_ENC_RC_2_PASS_FRAMESIZE_CAP;
```

The client should also not enable B frames for low latency scenarios.

To maintain uniform quality at all times, both I and P-frames must be encoded with the same quality. However, encoding I frames with quality comparable to P frame requires significantly more bits. With single-frame VBV buffer size, this is not possible, and results in poor quality I-frames.

For low-latency applications, NVIDIA recommends using infinite GOP length while encoding. This is achieved by setting `NV_ENC_CONFIG::gopLength` to `NVENC_INFINITE_GOPLENGTH`. Infinite GOP length disables automatic insertion of I frames. The client is recommended to avoid sending I frames unless it is necessary for error recovery.

Additionally, clients can also use two pass rate control (NV_ENC_PARAMS_RC_2_PASS_QUALITY), which will allow encoder to detect and perform additional processing to encode scene-cuts at better quality.

```
NV_ENC_CONFIG::idrPeriod = 0xffffffff;
NV_ENC_CONFIG::gopLength = NVENC_INFINITE_GOPLENGTH;
```

The time-stamps passed must be same as `NV_ENC_PIC_PARAMS::inputTimeStamp` value sent as part of the `NvEncEncodePicture` API while encoding those frames.

7.1.2. Two-Pass Rate Control

For better quality, it is recommended to use two-pass rate control (e.g. `NV_ENC_PARAMS_RC_TWOPASS_CBR)`, which allows better-quality encoding compared to single pass rate control (e.g. `NV_ENC_PARAMS_RC_CBR`) with a drop in performance.

If client sets NV_ENC_PARAMS_RC_2_PASS_QUALITY, encoder might violate the VBV settings to improve the quality of I frames.

Applications using strict single frame VBV and single pass rate control will result in poor quality scene-cuts and lower quality for frames with high complexity. Using two-pass rate control helps to mitigate these issues. It should be set as follows:

```
NV_ENC_RC_PARAMS::rateControlMode= NV_ENC_RC_2_PASS_FRAMESIZE_CAP;
```

7.1.3. Low latency Presets

The client application can use either of the following three encoding presets provided by NvENC interface which can be specified by `NV_ENC_CONFIG` parameter. This controls various encoding parameters for motion estimation (ME) and mode decision in the H.264 encoder.

`NV_ENC_PRESET_LOW_LATENCY_HQ_GUID`: This preset is desgined for to give the highest quality but at the least speed among all the low latency presets.

`NV_ENC_PRESET_DEFAULT_GUID`: This is the default low latency preset and the quality and speed is midway of the two other presets.

`NV_ENC_PRESET_LOW_LATENCY_HP_GUID`: This is the preset designed to give the maximum speed but the qaulity is expected to be the lowest among the three low latency presets.

7.2 QoS Features

7.2.1. Invalidate Reference Frame

This is the recommended error resiliency feature for low latency applications. When a client decoder detects a lost packet or corrupt frame, it can signal the server side encoder to invalidate that frame and all the frames, which have been constructed using the corrupt frame. If no frames are left for motion estimation, then the current frame will be encoded as an I frame. The client must set a large DPB size (`NV_ENC_CONFIG_H264::maxNumRefFrames`) to allow encoder to store a large number of backup reference frames, in case encoder has to fallback to use older reference frames for motion estimation when recent reference frames have been invalidated. Note that using a large DPB size doesn't increase the actual number of frames used for motion estimation; so there is no drop in performance. The recommended DPB size is 16 frames. This achieved by setting the following parameter while calling `NvEncInitializeEncoder` :

```
NV_ENC_CONFIG_H264::maxNumRefFrames = 16;
```

Let's say server receives a request from the decoder to invalidate *N* reference frames with capture time-stamps of *a*, *b*, *c*, … Then, the `NvEncInvalidateRefFrames` API must be called once with each time-stamp that needs to be invalidated.

7.2.1. Intra Refresh

This is also a recommended Error resiliency feature where after waves of Intra Macro blocks are forced over the frames to help recover from error gradually.

7.2.2. Dynamic Bit-Rate and Resolution change

Due to fluctuations in channel bandwidth the encoder may need to dynamically adjust resolutions and bitrate on the fly without destroying and recreating a new encoder session. This can be achieved by using Reconfigure API described in the next session.

7.2.3. Forcing Intra

The client can force individual frame to be encoded as intra-coded frame by setting `NV_ENC_PIC_PARAMS::encodePicFlags` to include `NV_ENC_PIC_FLAG_FORCEINTRA` while calling `NvEncEncodePicture` API.

7.3 Reconfigure API

`NvEncReconfigureEncoder` API allows changing of encoder initialization parameters in `NV_ENC_INITIALIZE_PARAMS` without closing existing encoder session. The reconfigured parameters are passed via `NV_ENC_RECONFIGURE_PARAMS::reInitEncodeParams.` It is much more efficient than recreating a new encode session.

Features like dynamic resolution change, bitrate change, and resetting the encoder are now supported using this API. This API was introduced in NV Encode API version 3.0.

Currently reconfiguration of following parameters is not supported via this method:

1. Change in GOP structure (`NV_ENC_CONFIG_H264::idrPeriod, NV_ENC_CONFIG::gopLength, NV_ENC_CONFIG:: frameIntervalP`)
2. Change in sync-async mode (`NV_ENC_INITIALIZE_PARAMS:: enableEncodeAsync)`
3. Change in MaxWidth & MaxHeight (`NV_ENC_INITIALIZE_PARAMS:: maxEncodeWidth, NV_ENC_INITIALIZE_PARAMS::maxEncodeHeight`)
4. Change in PTDmode (`NV_ENC_INITIALIZE_PARAMS::enablePTD`)

Support for changing these parameters *may be* added in the future versions of the API.

Resolution change is possible only if `maxEncodeWidth` & `maxEncodeHeight` of `NV_ENC_INITIALIZE_PARAMS` is set while creating encoder session.

If the client wishes to change the resolution using this API, it is advisable to force the next frame following the reconfiguration as an IDR frame by setting `NV_ENC_RECONFIGURE_PARAMS:: forceIDR` to 1.

If the client wishes to reset the internal rate control state, set `NV_ENC_RECONFIGURE_PARAMS::resetEncoder` to 1.