NVIDIA VIDEO CODEC SDK - ENCODER

Programming Guide

vNVENCODEAPI_PG-06155-001_v11 | March 2024

Table of Contents

Chapter 1. Introduction

NVIDIA® GPUs based on NVIDIA Kepler™ and later GPU architectures contain a hardware-based H.264/HEVC/AV1 video encoder (hereafter referred to as NVENC). The NVENC hardware takes YUV/RGB as input and generates an H.264/HEVC/AV1 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 10 and above) and Linux.

It is expected that developers should understand H.264/HEVC/AV1 video codecs and be familiar with Windows and/or Linux development environments.

NVENCODE API *guarantees* binary backward compatibility (and will make explicit reference whenever backward compatibility is broken). 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 NVENCODE API functions, structures and other parameters are exposed in `nvEncodeAPI.h`, which is included in the SDK.

NVENCODE API is a C-API, and uses a design pattern like C++ interfaces, wherein the application creates an instance of the API and retrieves a function pointer table to further interact with the encoder. For programmers preferring more high-level API with ready-to-use code, SDK includes sample C++ classes expose important API functions.

Rest of this document focuses on the C-API exposed in `nvEncodeAPI.h`. NVENCODE API is designed to accept raw video frames (in YUV or RGB format) and output the H.264, HEVC or AV1 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 10 and above only).
5. Clean-up - release all allocated input/output buffers
6. Close the encoding session

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` 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` and `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 10 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 10 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 10 and later versions of Windows OS.

3.1.1.4. DirectX 12

- 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 12 devices is supported only on Windows 10 20H1 and later versions of Windows OS.

3.1.1.5. 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 10 and later versions of Windows OS.

3.1.1.6. OpenGL

- The client should create an OpenGL context and make it current (in order to associate the context with the thread/process that is making calls to NVENCODE API) to the thread calling into NVENCODE API. `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::device` must be `NULL` and `NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS::deviceType` must be set to `NV_ENC_DEVICE_TYPE_OPENGL`. Use of the OpenGL device type for encoding is supported only on Linux.

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 large-enough buffer 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 TUNING INFO AND Preset Configurations

The NVIDIA Encoder Interface exposes four different tuning info enums (high quality, low latency, ultra-low latency and lossless) to cater to different video encoding use-cases. Table 1 shows the recommended tuning info applicable to some popular use-cases.

Table 1. Tuning info for popular video encoding use-cases

Use-case	Recommended value for tuning info parameter
1. **High-quality latency-tolerant transcoding** 2. **Video archiving** 3. **Encoding for OTT streaming**	Ultra High Quality / High quality
1. **Cloud gaming** 2. **Streaming** 3. **Video conferencing** **In high bandwidth channel with tolerance for bigger occasional frame sizes**	Low latency, with CBR
1. **Cloud gaming** 2. **Streaming** 3. **Video conferencing** **In strictly bandwidth-constrained channel**	Ultra-low latency, with CBR
1. **Preserving original video footage for later editing** 2. **General lossless data archiving (video or non-video)**	Lossless

For each tuning info, seven presets from P1 (highest performance) to P7 (lowest performance) have been provided to control performance/quality trade off. Using these presets will automatically set all relevant encoding parameters for the selected tuning info. 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 large-enough buffer 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 the use-case implied by the tuning info/preset combination. 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 Enumerating preset GUIDs.
2. Select the preset GUID for which the encode configuration is to be fetched.
3. The client should call `NvEncGetEncodePresetConfigEx` with the selected `encodeGUID`, `tuningInfo` 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. The client should call `NvEncGetInputFormatCount` to get the number of supported input formats.
2. The client should use this count to allocate a buffer to hold the list of supported input buffer formats (which are list elements of type `NV_ENC_BUFFER_FORMAT`).
3. 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 encoder Capabilities

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 Selecting Encoder Codec GUID.
- `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 as` explained below.

3.8.1.3. Advanced codec-specific parameters

Advanced H.264, HEVC and AV1 specific parameters are available in structures `NV_ENC_CONFIG_H264`, `NV_ENC_CONFIG_HEVC` and `NV_ENC_CONFIG_AV1` 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:

- The client should specify the session parameters as described in Section Session parameters.
- Optionally, the client can enumerate and select preset GUID that best suits the current use case, as described in Section Selecting Encoder Codec GUID. 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`, `tuning info` and `presetGUID` provided.
- 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 Session parameters.
2. The client should enumerate and select a preset GUID that best suites the current use case, as described in Section Selecting Encoder Codec GUID. The client should retrieve a preset encode configuration as described in Section Selecting encoder preset configuration.
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 tuning info and preset GUID and fetch the corresponding Preset Encode Configuration as described in Section Encoder TUNING INFO AND Preset Configurations.
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. Rate control

NVENC supports several rate control modes and provides control over various parameters related to the rate control algorithm via structure `NV_ENC_INITIALIZE_PARAMS::encodeConfig::rcParams`. The rate control algorithm is implemented in NVENC firmware.

NVENC supports the following rate control modes:

Constant bitrate (CBR): Constant bitrate is specified by setting `rateControlMode` to `NV_ENC_PARAMS_RC_CBR`. In this mode, only `averageBitRate` is required and used as the target output bitrate by the rate control algorithm. Clients can control the ratio of I to P frames using `NV_ENC_RC_PARAMS::lowDelayKeyFrameScale` which is useful to avoid channel congestion in case I frame ends up generating high number of bits. Set `NV_ENC_CONFIG_H264/ NV_ENC_CONFIG_HEVC::enableFillerDataInsertion = 1` or `NV_ENC_CONFIG_AV1::enableBitstreamPadding = 1` incase the bitrate needs to be strictly adhered to.

Variable bitrate (VBR): Variable bitrate is specified by setting `rateControlMode` to `NV_ENC_PARAMS_RC_VBR`. The encoder tries to conform to average bitrate of `averageBitRate` over the long term while not exceeding `maxBitRate` any time during the encoding. In this mode, `averageBitRate` must be specified. If `maxBitRate` isn't specified, NVENC will set it to an internally determined default value. It is recommended that the client specify both parameters `maxBitRate` and `averageBitRate` for better control.

Constant QP: This mode is specified by setting `rateControlMode` to `NV_ENC_PARAMS_RC_CONSTQP`. In this mode, the entire frame is encoded using QP specified in `NV_ENC_RC_PARAMS::constQP`.

Target quality: This mode is specified by setting `rateControlMode` to VBR and desired target quality in `targetQuality`. The range of this target quality is 0 to 51(fractional values are also supported in Video Codec SDK 8.0 and above). In this mode, the encoder tries to maintain constant quality for each frame, by allowing the bitrate to vary subject to the bitrate parameter specified in `maxBitRate`. The resulting average bitrate can, therefore, vary significantly depending on the video content being encoded. In this mode, if `maxBitRate` is set, it will form an upper bound on the actual bitrate. Therefore, if `maxBitRate` is set too low, the bitrate may become constrained, resulting in the desired target quality possibly not being achieved.

3.8.4. Multi pass frame encoding

When determining the QP to use for encoding a frame, it is beneficial if NVENC knows the overall complexity of the frame to distribute the available bit budget in the most optimal manner. In some situations, multi-pass encoding may also help catch larger motion between frames. For this purpose, NVENC supports the following types of multi-pass frame encoding modes:

- 1-pass per frame encoding (`NV_ENC_MULTI_PASS_DISABLED`)
- 2-passes per frame, with first pass in quarter resolution and second pass in full resolution (`NV_ENC_TWO_PASS_QUARTER_RESOLUTION`)
- 2-passes per frame, with both passes in full resolution (`NV_ENC_TWO_PASS_FULL_RESOLUION`).

In 1-pass rate control modes, NVENC estimates the required QP for the macroblock and immediately encodes the macroblock. In 2-pass rate control modes, NVENC estimates the complexity of the frame to be encoded and determines bit distribution across the frame in the first pass. In the second pass, NVENC encodes macroblocks in the frame using the distribution determined in the first pass. As a result, with 2-pass rate control modes, NVENC can distribute the bits more optimally within the frame and can reach closer to the target bitrate, especially for CBR encoding. Note, however, that everything else being the

same, performance of 2-pass rate control mode is lower than that of 1-pass rate control mode. The client application should choose an appropriate multi-pass rate control mode after evaluating various modes, as each of the modes has its own advantages and disadvantages. `NV_ENC_TWO_PASS_FULL_RESOLUION` generates better statistics for the second pass, whereas `NV_ENC_TWO_PASS_QUARTER_RESOLUTION` results in larger motion vectors being caught and fed as hints to second pass.

3.8.5. 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` 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:

3.8.5.1. 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 supported only on Windows 10 and later. Refer to Chapter 6 for more detailed explanation.

3.8.5.2. Picture-type decision

If the client wants to send the input buffers in display order, it must set `enablePTD = 1`. If `enablePTD` is set to 1 the decision of determining the picture type will be taken by NVENCODE API.

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/ NV_ENC_PIC_PARAMS_AV1::displayPOCSyntax`

`NV_ENC_PIC_PARAMS_H264/NV_ENC_PIC_PARAMS_HEVC/ NV_ENC_PIC_PARAMS_AV1::refPicFlag`

`NV_ENC_PIC_PARAMS_AV1::goldenFrameFlag/arfFrameFlag/arf2FrameFlag/ bwdFrameFlag/overlayFrameFlag`

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 for destroying 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 must 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 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. NVIDIA Video Encoder Interface supports `CUdeviceptr` and `CUarray` input formats.

If the client has used the OpenGL device type to initialize the encoder session and wishes to use input buffers NOT allocated through the NVIDIA Video Encoder Interface, the client is required to provide the textures allocated earlier.

The client may generate textures using `glGenTextures()`, bind it to either the `NV_ENC_INPUT_RESOURCE_OPENGL_TEX::GL_TEXTURE_RECTANGLE` or `NV_ENC_INPUT_RESOURCE_OPENGL_TEX::GL_TEXTURE_2D` target, allocate storage for it using `glTexImage2D()` and copy data to it.

Note that the OpenGL interface for NVENCODE API is only supported on Linux.

If the client has used a DirectX 12 device to initialize encoder session, then client must allocate input and output buffers using `ID3D12Device::CreateCommittedResource()` API. The client must perform some simple processing to map these input and output resources to resource handles that are recognized by the NVIDIA Video Encoder Interface before use. The translation procedure is explained in Section Input output buffer allocation for DirectX 12.

Note: The client should allocate at least (1 + *NB*) input and output buffers, where *NB* 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 for H.264/HEVC and Sequence Header OBU for AV1) 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 and Sequence Header OBU data will be attached to every IDR frame and Key frame for H.264/HEVC and AV1 respectively. However, the client can request the encoder to generate SPS/PPS and Sequence Header OBU data on demand as well. To accomplish this, set `NV_ENC_PIC_PARAMS::encodePicFlags = NV_ENC_PIC_FLAG_OUTPUT_SPSPPS`. The

output bitstream generated for the current input will then include SPS/PPS for H.264/HEVC or Sequence Header OBU for AV1.

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 `NvEncUnlockInputBuffer`. The input buffer should be passed to the encoder only after unlocking it. Any input buffers should be unlocked by calling `NvEncUnlockInputBuffer` before destroying/reallocating them.

4.1.2. Input buffers allocated externally

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.

4.1.3. Input output buffer allocation for DirectX 12

Allocation of input and output buffers should be done in following manner:

1. Input buffer should be created using DirectX 12 `ID3D12Device::CreateCommittedResource()` API, by specifying `D3D12_HEAP_PROPERTIES::Type = D3D12_HEAP_TYPE_DEFAULT` and `D3D12_RESOURCE_DESC::Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D`.
2. Output buffer should be created using DirectX 12 `ID3D12Device::CreateCommittedResource()` API, by specifying `D3D12_HEAP_PROPERTIES::Type = D3D12_HEAP_TYPE_READBACK` and `D3D12_RESOURCE_DESC::Dimension = D3D12_RESOURCE_DIMENSION_BUFFER`.
3. For HEVC, H.264 or AV1 encoding, the recommended size for output buffer is:

   ```
   Output buffer size = 2 * Input YUV buffer size
   ```

   in bytes.

To pass these externally allocated input and output buffers to the encoder, follow these steps:

1. Populate `NV_ENC_REGISTER_RESOURCE` with attributes of the externally allocated buffer.
2. To enable explicit synchronization in DirectX 12, the API `NvEncRegisterResource` accepts two `NV_ENC_FENCE_POINT_D3D12` pointer type objects (A fence point is a pair of `ID3D12Fence` pointer and a `value`), `NV_ENC_REGISTER_RESOURCE_PARAMS_D3D12::pInputFencePoint` and `NV_ENC_REGISTER_RESOURCE_PARAMS_D3D12::pOutputFencePoint`, for registering input buffer. NVENC engine waits until `pInputFencePoint` is reached before processing the `NV_ENC_REGISTER_RESOURCE::resourceToRegister`. NVENC engine signals the `pOutputFencePoint` when processing of the resource is completed so that other engines which need to use this resource can start processing.
3. Call `NvEncRegisterResource` with the `NV_ENC_REGISTER_RESOURCE` populated in the above step.
4. `NvEncRegisterResource` returns an opaque handle in `NV_ENC_REGISTER_RESOURCE::registeredResource` which should be saved.
5. The client should use this registered handle (`NV_ENC_REGISTER_RESOURCE::registeredResource`) as the input and output buffer handle in `NV_ENC_INPUT_RESOURCE_D3D12::pInputBuffer` and `NV_ENC_INPUT_RESOURCE_D3D12::pOutputBuffer` respectively.
6. The client must also call `NvEncUnregisterResource` with the handle returned by `NvEncRegisterResource` before destroying the registered resource.

The registered resource handle (`NV_ENC_REGISTER_RESOURCE::registeredResource`) should not be used for any other purpose outside the NVIDIA Video Encoder Interface while it is in registered 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/
NV_ENC_PIC_PARAMS_AV1::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 (H.264 and HEVC) or Sequence Header OBU (AV1) 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/NV_ENC_PIC_PARAMS_AV1 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.

If the client has used a DirectX 12 device to initialize encoder session, client must pass pointer to NV_ENC_INPUT_RESOURCE_D3D12 in NV_ENC_PIC_PARAMS:: inputBuffer containing the registered resource handle and the corresponding input NV_ENC_FENCE_POINT_D3D12 for NVENC to wait before starting encode. Client must pass pointer to NV_ENC_OUTPUT_RESOURCE_D3D12 in NV_ENC_PIC_PARAMS::outputBuffer containing the registered resource handle and the corresponding output NV_ENC_FENCE_POINT_D3D12. NVENC engine waits until the NV_ENC_INPUT_RESOURCE_D3D12::inputFencePoint is reached before starting processing of input buffer. NVENC engine signal the NV_ENC_OUTPUT_RESOURCE_D3D12::outputFencePoint when processing of the resource is completed so that other engines which need to use these input and output resources can start processing.

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 NvEncUnlockBitstream. The client should call NvEncUnlockBitstream after it completes processing the output data.

If the client has used a DirectX 12 device to initialize encoder session, client must pass the same NV_ENC_OUTPUT_RESOURCE_D3D12 pointer in NV_ENC_LOCK_BITSTREAM::outputBitstream for retrieving the output, which it had sent in NV_ENC_PIC_PARAMS::outputBuffer during encode.

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 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 `NvEncUnlockInputBuffer` 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

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 supports asynchronous mode of operation only for Windows 10 and above, with driver running in WDDM mode. In Linux and Windows with TCC mode (TCC mode is available on Tesla boards[1]), ONLY synchronous mode is supported (refer to Section 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 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 `NvEncCreateBitstreamBuffer` API. The NVIDIA Video Encoder Interface will return an opaque pointer to the output memory in `NV_ENC_CREATE_BITSTREAM_BUFFER::bitstreambuffer`. This opaque

[1] To check the mode in which your board is running, run the command-line utility nvidia-smi (nvidia-smi.exe on Windows) included with the driver.

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 output buffer and event 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). When `enablePTD = 1`, NVIDIA Encoder takes care of the reordering in case of B frames in a way that is transparent to the encoder clients. For AV1, NVIDIA encoder also transparently performs frame bitstream packing, meaning it always concatenates into a single output buffer the bitstream corresponding to leading no-show frames with the bitstream of the first show frame that follows. Each output buffer therefore always contains a single frame to display along with all the preceding non-display frames in encode order since the previous frame to display.
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:

- The client will receive the event's signal and output buffer in the same order in which they were queued.
- The `NV_ENC_LOCK_BITSTREAM::pictureType` notifies 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.

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

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 many input and output buffers in order to avoid resource hazards and improve overall encoder throughput.

On Windows, when encode device type is DirectX, calling DXGI APIs like `IDXGIOutputDuplication::AcquireNextFrame` from the primary thread and `NvEncLockBitstream`/`NvEncUnlockBitstream` from secondary thread, can lead to suboptimal or undefined behavior. This is because `NvEncLockBitstream` can internally use the application's DirectX device.

For optimal performance in such applications, the following encoder settings should be used:

- `NV_ENC_INITIALIZE_PARAMS::enableEncodeAsync = 1`
- `NV_ENC_LOCK_BITSTREAM::doNotWait = 0`
- `NV_ENC_INITIALIZE_PARAMS::enableOutputInVidmem = 0`

6.4. 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: 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
- Weighted prediction
- Encoding of RGB contents
- Temporal Filtering

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. The resulting motion vectors or mode decisions can be 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.

For use-cases involving computer vision, AI and frame interpolation, Turing and later GPUs contain another hardware accelerator for computing optical flow vectors between frames, which provide better visual matching than the motion vectors.

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.

Motion-estimation (ME) only mode is not supported if DirectX 12 device is used.

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 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. Populate ME only mode settings

The structure `NV_ENC_CODEC_CONFIG::NV_ENC_CONFIG_H264_MEONLY` provides the ability to control the partition types of motion vectors and modes returned by NVENC hardware. Specifically, the client can disable intra mode and/or specific MV partition sizes by setting the following flags:

```
NV_ENC_CONFIG_H264_MEONLY::disableIntraSearch
NV_ENC_CONFIG_H264_MEONLY::disablePartition16x16
NV_ENC_CONFIG_H264_MEONLY::disablePartition8x16
NV_ENC_CONFIG_H264_MEONLY::disablePartition16x8
NV_ENC_CONFIG_H264_MEONLY::disablePartition8x8
```

The API also exposes a parameter `NV_ENC_CONFIG::NV_ENC_MV_PRECISION` to control the precision of motion vectors returned by the hardware. For full-pel precision, the client must ignore two LSBs of the motion vector. For sub-pel precision, the two LSBs of the motion vector represent fractional part of the motion vector. To get motion vectors for each macroblock, it is recommended to disable intra modes by setting `NV_ENC_CONFIG_H264_MEONLY::disableIntraSearch = 1` and let NVENC decide the optimal partition sizes for motion vectors.

7.4. 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`.

7.5. Enabling Motion estimation for stereo usecases

For stereo use cases where in two views need to be processed, we suggest the following approach for better performance and quality of motion vectors:

- Client should create single encode session.
- The client should kick-off the processing of left and right views on separate threads.
- The client should set `NV_ENC_MEONLY_PARAMS::viewID` to 0 and 1 for left and right views.
- The main thread should wait for completion of the threads which have been kicked off NVENC for left and right views.

7.6. 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. This also dynamically allocates B and P frames.

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. B-Frames As Reference

Using B frame as a reference improves subjective and objective encoded quality with no performance impact. Hence the users enabling multiple B frames are strongly recommended to enable this feature.

To use the feature, follow these steps:

- Query availability of the feature using `NvEncGetEncodeCaps` API and checking for `NV_ENC_CAPS_SUPPORT_BFRAME_REF_MODE` in the return value.
- During encoder initialization, set `NV_ENC_CONFIG_H264/NV_ENC_CONFIG_HEVC/NV_ENC_CONFIG_AV1::useBFramesAsRef` = `NV_ENC_BFRAME_REF_MODE_MIDDLE`:
  - For H.264 and HEVC, this will set the (N/2)th B frame as reference where N = number of B frames. In case N is odd, then (N-1)/2th frame will be picked up as reference.
  - For AV1, this will set every other B frame as an Altref2 reference but for the last B frame in the Altref interval.

8.3. Reconfigure API

`NvEncReconfigureEncoder` allows clients to change the encoder initialization parameters in `NV_ENC_INITIALIZE_PARAMS` without closing existing encoder session and re-creating a new encoding session. This helps clients avoid the latency introduced due to destruction and re-creation of the encoding session. This API is useful in scenarios which are prone to instabilities in transmission mediums during video conferencing, game streaming etc.

Using this API clients can change parameters like bit-rate, frame-rate, resolution dynamically using the same encode session. The reconfigured parameters are passed via `NV_ENC_RECONFIGURE_PARAMS::reInitEncodeParams`.

However, The API currently doesn't support reconfiguration of all parameters, some of which are listed below:

- Changing the GOP structure (`NV_ENC_CONFIG_H264::idrPeriod`, `NV_ENC_CONFIG::gopLength`, `NV_ENC_CONFIG::frameIntervalP`)
- Changing from synchronous mode of encoding to asynchronous mode and vice-versa.
- Changing `NV_ENC_INITIALIZE_PARAMS::maxEncodeWidth` and `NV_ENC_INITIALIZE_PARAMS::maxEncodeHeight`.
- Changing picture type decision in `NV_ENC_INITIALIZE_PARAMS::enablePTD`.
- Changing bit-depth.
- Changing chroma format.
- Changing `NV_ENC_CONFIG_HEVC::maxCUSize`.
- Changing `NV_ENC_CONFIG::frameFieldMode`.

The API would fail if any attempt is made to reconfigure the parameters which is not supported.

Resolution change is possible only if `NV_ENC_INITIALIZE_PARAMS::maxEncodeWidth` and `NV_ENC_INITIALIZE_PARAMS::maxEncodeHeight` are 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 states, set `NV_ENC_RECONFIGURE_PARAMS::resetEncoder` to 1.

8.4. Adaptive Quantization (AQ)

This feature improves visual quality by adjusting encoding QP (on top of QP evaluated by the Rate Control Algorithm) based on spatial and temporal characteristics of the sequence. The current SDK support two flavors of AQ which are explained as follows:

8.4.1. Spatial AQ

Spatial AQ mode adjusts the QP values based on spatial characteristics of the frame. Since the low complexity flat regions are visually more perceptible to quality differences than high complexity detailed regions, extra bits are allocated to flat regions of the frame at the cost of the regions having high spatial detail. Although spatial AQ improves the perceptible visual quality of the encoded video, the required bit redistribution results in PSNR drop in most of the cases. Therefore, during PSNR-based evaluation, this feature should be turned off.

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

- Spatial AQ can be enabled during initialization by setting `NV_ENC_INITIALIZE_PARAMS::encodeconfig->rcParams. enableAQ = 1`.
- The intensity of QP adjustment can be controlled by setting `NV_ENC_INITIALIZE_PARAMS::encodeconfig->rcParams.aqStrength` which ranges from 1 (least aggressive) to 15 (most aggressive). If not set, strength is auto selected by driver.

8.4.2. Temporal AQ

Temporal AQ tries to adjust encoding QP (on top of QP evaluated by the rate control algorithm) 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.

1. Query the availability of temporal AQ for the current hardware by calling the API NvEncGetEncodeCaps and checking for NV_ENC_CAPS_SUPPORT_TEMPORAL_AQ.

2. If supported, temporal AQ can be enabled during initialization by setting NV_ENC_INITIALIZE_PARAMS::encodeconfig->rcParams.enableTemporalAQ = 1.

Temporal AQ uses CUDA pre-processing and hence requires CUDA processing power, depending upon resolution and content.

Enabling temporal AQ may result in very minor degradation in encoder performance.

8.5. High Bit Depth Encoding

All NVIDIA GPUs support 8-bit encoding (RGB/YUV input with 8-bit precision). Starting from Pascal generation, NVIDIA GPUs support high-bit-depth HEVC encoding (HEVC main-10 profile with 10-bit input precision). Starting from Ada generation, NVIDIA GPUs support high-bit-depth AV1 encoding (AV1 main profile with 8 or 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` for HEVC or `NV_ENC_AV1_PROFILE_MAIN_GUID` for AV1.
3. During encoder initialization,
   - For HEVC, set `encodeConfig->encodeCodecConfig.hevcConfig.outputBitDepth = 10` and `encodeConfig->encodeCodecConfig.av1Config.inputBitDepth = 8` for 8-bit input content or `encodeConfig->encodeCodecConfig.av1Config.inputBitDepth = 10` for 10-bit input content. In case of 8-bit input content, NVENC performs an internal CUDA 8 to 10-bit conversion of the input prior to encoding.
   - For AV1, set `encodeConfig->encodeCodecConfig.av1Config.outputBitDepth = 10` and `encodeConfig->encodeCodecConfig.av1Config.inputBitDepth = 8` for 8-bit input content or `encodeConfig->encodeCodecConfig.av1Config.inputBitDepth = 10` for 10-bit input content. In case of 8-bit input content, NVENC performs an internal HW 8 to 10-bit conversion of the input prior to encoding.
4. Other encoding parameters such as preset, rate control mode, etc. can be set as desired.

8.6. Weighted Prediction

Weighted prediction involves calculation of a multiplicative weighting factor and an additive offset to the motion compensated prediction. Weighted prediction provides significant quality gain for contents having illumination changes. NVENCODE API supports weighed prediction for HEVC and H.264 starting from Pascal generation GPUs.

The following steps need to be followed for enabling weighted prediction.

1. The availability of the feature can be queried using `NvEncGetEncodeCaps` and checking for `NV_ENC_CAPS_SUPPORT_WEIGHTED_PREDICTION`.
2. During encoder initialization, set `NV_ENC_INITIALIZE_PARAMS::enableWeightedPrediction = 1`.

Weighted prediction is not supported if the encode session is configured with B frames.

Weighted prediction is not supported if DirectX 12 device is used.

Weighted prediction uses CUDA pre-processing and hence requires CUDA processing power, depending upon resolution and content.

Enabling weighted prediction may also result in very minor degradation in encoder performance.

8.7. Long-Term Reference in H.264 and HEVC

NVENCODE API provides the functionality to mark and use specific frames as long-term reference (LTR) frames, which can later be used as reference for encoding the current picture. This helps in error concealment where in the client decoders can predict from the long-term reference frame in case an intermediate frame loses data. The feature is useful in video streaming applications to recover from frame losses at the receiver.

Following steps are to be followed to enable the feature.

1. During encoder initialization,
   - For H.264, set `NV_ENC_CONFIG_H264:enableLTR` = 1
   - For HEVC, set `NV_ENC_CONFIG_HEVC:enableLTR` = 1
2. The maximum number of long-term reference pictures supported in the current hardware can be queried using `NvEncGetEncoderCaps` and checking for `NV_ENC_CAPS_NUM_MAX_LTR_FRAMES`.

During normal encoding operation, following steps need to be followed to mark specific frame(s) as LTR frame(s).

1. Configure the number of LTR frames:
   - For H.264, set `NV_ENC_CONFIG_H264:ltrNumFrames`
   - For HEVC, set `NV_ENC_CONFIG_HEVC:ltrNumFrames`
2. The client can mark any frame as LTR by setting `NV_ENC_PIC_PARAMS_H264::ltrMarkFrame = 1` OR `NV_ENC_PIC_PARAMS_HEVC::ltrMarkFrame = 1` for H.264 and HEVC respectively. Each LTR frame needs to be assigned an LTR frame index. This value should be between 0 and `ltrNumFrames - 1`.
3. The LTR frame index can be assigned by setting `NV_ENC_PIC_PARAMS_H264::ltrMarkFrameIdx` OR `NV_ENC_PIC_PARAMS_HEVC::ltrMarkFrameIdx` for H264 and HEVC respectively.

The frames previously marked as long-term reference frames can be used for prediction of the current frame in the following manner:

1. The LTR frames that are to be used for reference have to be specified using `NV_ENC_PIC_PARAMS_H264::ltrUseFrameBitmap` OR

`NV_ENC_PIC_PARAMS_HEVC::ltrUseFrameBitmap` for H.264 and HEVC respectively. The bit location specifies the LTR frame index of the frame that will be used as reference.

The current SDK does not support LTR when the encoding session is configured with B frames.

8.8. Emphasis MAP

The emphasis map feature in NVENCODE API provides a way to specify regions in the frame to be encoded at varying levels of quality, at macroblock-level granularity. Depending upon the actual emphasis level for each macroblock, the encoder applies an adjustment to the quantization parameter used to encode that macroblock. The value of this adjustment depends on the following factors:

- Absolute value of the QP as decided by the rate control algorithm, depending upon the rate control constraints. In general, for a given emphasis level, higher the QP determined by the rate control, higher the (negative) adjustment.
- Emphasis level value for the macroblock.

Note: The QP adjustment is performed after the rate control algorithm has run. Therefore, there is a possibility of VBV/rate violations when using this feature.

Emphasis level map is useful when the client has prior knowledge of the image complexity (e.g. NVFBC's Classification Map feature) and encoding those high-complexity areas at higher quality (lower QP) is important, even at the possible cost of violating bitrate/VBV buffer size constraints. This feature is not supported when AQ (Spatial/Temporal) is enabled.

Follow these steps to enable the feature.

1. Query availability of the feature using `NvEncGetEncodeCaps` API and checking for `NV_ENC_CAPS_SUPPORT_EMPHASIS_LEVEL_MAP`.
2. Set `NV_ENC_RC_PARAMS::qpMapMode = NV_ENC_QP_MAP_EMPHASIS`.
3. Fill up the `NV_ENC_PIC_PARAMS::qpDeltaMap` (which is a signed byte array containing value per macroblock in raster scan order for the current picture) with a value from `enum NV_ENC_EMPHASIS_MAP_LEVEL`.

As explained above, higher values of `NV_ENC_EMPHASIS_MAP_LEVEL` imply higher (negative) adjustment made to the QP to *emphasize* quality of that macroblock. The user can choose higher emphasis level for the regions (s)he wants to encode with a higher quality.

8.9. NVENC Output in Video Memory

Starting SDK 9.0, NVENCODE API supports bitstream and H.264 ME-only mode output in video memory. This is helpful in use-cases in which the operation on the output of NVENC is to be performed using CUDA or DirectX shaders. Leaving the output of NVENC in video memory avoids unnecessary PCIe transfers of the buffers. The video memory should be allocated by the client application, as 1-dimensional buffer. This feature is currently supported for H.264, HEVC and AV1 encode, and H.264 ME-only mode. The feature is supported for DirectX 11 and CUDA interfaces.

Follow these steps for the output to be available in video memory.

1. Set `NV_ENC_INITIALIZE_PARAMS::enableOutputInVidmem = 1` when calling `nvEncInitializeEncoder()`.
2. Allocate 1-dimensional buffer in video memory for NVENC to write the output.
   - For AV1, HEVC or H.264 encoding, the recommended size for this buffer is:

     ```
     Output buffer size = 2 * Input YUV buffer size + sizeof(NV_ENC_ENCODE_OUT_PARAMS)
     ```

     First `sizeof(NV_ENC_ENCODE_OUT_PARAMS)` bytes of the output buffer contain `NV_ENC_ENCODE_OUT_PARAMS` structure, followed by encoded bitstream data.
   - For H.264 ME-only output, the recommended size of output buffer is:

     ```
     Output buffer size = HeightInMbs * WidthInMbs * sizeof(NV_ENC_H264_MV_DATA)
     ```

     where `HeightInMbs` and `WidthInMbs` are picture height and width in number of 16x16 macroblocks, respectively.
   - For DirectX 11 interface, this buffer can be created using DirectX 11 `CreateBuffer()` API, by specifying `usage = D3D11_USAGE_DEFAULT; BindFlags = (D3D11_BIND_VIDEO_ENCODER | D3D11_BIND_SHADER_RESOURCE); and CPUAccessFlags = 0;`
   - For CUDA interface, this buffer can be created using `cuMemAlloc()`.
3. Register this buffer using `nvEncRegisterResource()`, by specifying:
   - `NV_ENC_REGISTER_RESOURCE::bufferUsage = NV_ENC_OUTPUT_BITSTREAM` if output is encoded bitstream,
   - and as `NV_ENC_REGISTER_RESOURCE::bufferUsage= NV_ENC_OUTPUT_MOTION_VECTOR` if output is motion vectors in case of H.264 ME only mode.
   - Set `NV_ENC_REGISTER_RESOURCE::bufferFormat= NV_ENC_BUFFER_FORMAT_U8`. `NvEncRegisterResource()` will return a registered handle in `NV_ENC_REGISTER_RESOURCE::registeredResource`.
4. Set`NV_ENC_MAP_INPUT_RESOURCE::registeredResource = NV_ENC_REGISTER_RESOURCE::registeredResource` obtained in the previous step.
5. Call `nvEncMapInputResource()`, which will return a mapped resource handle in `NV_ENC_MAP_INPUT_RESOURCE::mappedResource`.
6. For AV1/HEVC/H.264 encoding mode, call `nvEncEncodePicture()` by setting `NV_ENC_PIC_PARAMS::outputBitstream` to `NV_ENC_MAP_INPUT_RESOURCE::mappedResource`.
7. For H.264 ME-only mode, call nvEncRunMotionEstimationOnly() by setting `NV_ENC_MEONLY_PARAMS::mvBuffer` to `NV_ENC_MAP_INPUT_RESOURCE::mappedResource`.

When reading the output buffer, observe the following:

After calling `nvEncEncodePicture()` or `nvEncRunMotionEstimationOnly()`, client can use the output buffer for further processing only after un-mapping this output buffer. `NvEncLockBitstream()` should not be called.

When operating in asynchronous mode, client application should wait on event before reading the output. In synchronous mode no event is triggered, and the synchronization is handled internally by NVIDIA driver.

To access the output, follow these steps:

1. Client must un-map the input buffer by calling `nvEncUnmapInputResource()` with mapped resource handle `NV_ENC_MAP_INPUT_RESOURCE::mappedResource` returned by `nvEncMapInputResource()`. After this, the output buffer can be used for further processing/ reading etc.
2. In case of encode, the first `sizeof(NV_ENC_ENCODE_OUT_PARAMS)` bytes of this buffer should be interpreted as `NV_ENC_ENCODE_OUT_PARAMS` structure followed by encode bitstream data.The size of encoded bitstream is given by `NV_ENC_ENCODE_OUT_PARAMS::bitstreamSizeInBytes`.
3. If CUDA mode is specified, all CUDA operations on this buffer must use the default stream. To get the output in system memory, output buffer can be read by calling any CUDA API (e.g. `cuMemcpyDtoH()`) with default stream. The driver ensures that the output buffer is read only after NVENC has finished writing the output in it.
4. For DX11 mode, any DirectX 11 API can be used to read the output. The driver ensures that the output buffer is read only after NVENC has finished writing the output in it. To get the output in system memory, `CopyResource()` (which is a DirectX 11 API) can be used to copy the data in a CPU readable staging buffer. This staging buffer then can be read after calling `Map()` which is a DirectX 11 API.

8.10. Alpha Layer Encoding support in HEVC

NVENCODE API implements support for encoding alpha layer in HEVC. The feature allows an application to encode a base layer which contains YUV data and an auxilary layer with alpha channel data.

Following steps are to be followed to enable the feature:

1. The availability of the feature can be queried using `nvEncGetEncodeCaps()` and checking for `NV_ENC_CAPS_SUPPORT_ALPHA_LAYER_ENCODING`. Note that only `NV_ENC_BUFFER_FORMAT_NV12`, `NV_ENC_BUFFER_FORMAT_ARGB` and `NV_ENC_BUFFER_FORMAT_ABGR` input formats are supported with alpha layer encoding.
2. During encoder initialization, set `NV_ENC_CONFIG_HEVC:: enableAlphaLayerEncoding = 1`. Clients can also specify the ratio in which the bitrate is to be split between YUV and the auxiliary alpha layer by setting `NV_ENC_RC_PARAMS::alphaLayerBitrateRatio`. For example, if `NV_ENC_RC_PARAMS::alphaLayerBitrateRatio = 3` then 75% of the bits will be spent on base layer encoding whereas the other 25% will be spent on alpha layer.

During normal encoding operation, following steps need to be followed to for alpha layer encoding:

1. Passing the alpha input in `nvEncEncodePicture() :`
   - For input format `NV_ENC_BUFFER_FORMAT_NV12`, the YUV data should be passed in `NV_ENC_PIC_PARAMS::inputBuffer` whereas the alpha input data needs to be passed separately using `NV_ENC_PIC_PARAMS::alphaBuffer`. The format of

`NV_ENC_PIC_PARAMS::alphaBuffer` should be `NV_ENC_BUFFER_FORMAT_NV12`. The luma plane should contain the alpha data whereas the chroma component should to be memset to 0x80.

- For input format `NV_ENC_BUFFER_FORMAT_ABGR` or `NV_ENC_BUFFER_FORMAT_ARGB`, the input data should be passed in `NV_ENC_PIC_PARAMS::inputBuffer`. The field `NV_ENC_PIC_PARAMS::alphaBuffer` should be set as NULL in this case.

2. The encoded output for YUV as well as the alpha layer is fetched using a call to `NvEncLockBitstream`. The field `NV_ENC_LOCK_BITSTREAM::bitstreamSizeInBytes` will contain the total encoded size i.e it is the size of YUV layer bitstream data, alpha bitstream data and any other header data. Cients can get the size of alpha layer separately using the field `NV_ENC_LOCK_BITSTREAM::alphaLayerSizeInBytes`.

Alpha encoding is not supported in the following scenarios:

1. When subframe more is enabled.
2. Input image is YUV 444.
3. The bit-depth of input image is 10 bit.
4. The bit stream output is specified to be in video memory.
5. Weighted prediction is enabled.

8.11. Temporal Scalable Video Coding (SVC) in H.264

NVENCODE API supports temporal scalable video coding(SVC) as specified in Annex G of the H.264/AVC video compression standard. Temporal SVC results in an heirarchical structure with a base layer and multiple auxiliary layers.

To use temporal SVC, follow these steps:

1. Query the availability of temporal SVC for the current hardware by calling the API NvEncGetEncodeCaps and checking for NV_ENC_CAPS_SUPPORT_TEMPORAL_SVC.
2. If supported, query the maximum number of temporal layers supported in SVC using `nvEncGetEncodeCaps()` and check the value of `NV_ENC_CAPS_NUM_MAX_TEMPORAL_LAYERS`
3. During encoder initialization, set `NV_ENC_CONFIG_H264::enableTemporalSVC = 1`. Specify the number of temporal layer and maximum number of temporal layers using `NV_ENC_CONFIG_H264::numTemporalLayers` and `NV_ENC_CONFIG_H264::maxTemporalLayers` respectively.
4. If maximum number of temporal layer is greater than 2, then the minimum DPB size for frame reordering needs to be `(maxTemporalLayers - 2) * 2`. Therefore, set the `NV_ENC_CONFIG_H264::maxNumRefFrames` to be greater than or equal to this value. Note that the default value of `NV_ENC_CONFIG_H264::maxNumRefFrames` is `NV_ENC_CAPS::NV_ENC_CAPS_NUM_MAX_TEMPORAL_LAYERS`.
5. By default, SVC prefix NALU is added when temporal SVC is enabled. To disable this, set `NV_ENC_CONFIG_H264::disableSVCPrefixNalu = 0`.
6. NVENCODE API supports addition of scalability information SEI message in the bitstream. To enable this SEI, set `NV_ENC_CONFIG_H264::enableScalabilityInfoSEI = 1`. This SEI

will be added with every IDR frame in the encoded bitstream. Note that only a subset of fields related to temporal scalability is currently supported in this SEI.

When temporal SVC is enabled, only base layer frames can be marked as long term references.

Temporal SVC is currently not supported with B-frames. The field `NV_ENC_CONFIG::frameIntervalP` will be ignored when temporal SVC is enabled.

8.12. Error Resiliency features

In a typical scenario involving video streaming, it is common to have bit errors at the client decoder. To minimize the impact of these errors and to recover from such errors, NVENCODE API provides a few error resiliency features which are explained in this section.

Reference Picture Invalidation

NVENCODE API provides a mechanism for invalidating certain pictures when a picture decoded by a client is found to be corrupt by the decoder (client side). Such invalidation is achieved by using the API `NvEncInvalidateRefFrames`. The client can prevent further corruption by asking the encoder on the streaming server to invalidate this frame, which will prevent all the subsequent frames from using the current frame as reference frame for motion estimation. The server then uses older short term and long term frames for reference based on whatever is available for reference. In case no frame is available for reference the current frame will be encoded as intra frame. The parameter `NV_ENC_CONFIG_H264::maxNumRefFrames`, `NV_ENC_CONFIG_HEVC::maxNumRefFramesInDPB` or `NV_ENC_CONFIG_AV1::maxNumRefFramesInDPB` determines the number of frames in DPB and setting this to a large value will allow older frames to be available in the DPB even when some frames have been invalidated, and allow for better picture quality as compared to an intra frame that will be coded in the absence of no reference frames.

The specific frame to be invalidated via API `NvEncInvalidateRefFrames` is identified using a unique number for each frame, referred to as *timestamp*. This is the timestamp sent to the encoder via field NV_ENC_PIC_PARAMS::inputTimeStamp when encoding the picture. This can be any monotonically increasing unique number. In its most common incarnation, the unique number can be the presentation timestamp for the picture. The encoder stores the frames in its DPB using `inputTimeStamp` as the unique identifier and uses that identifier to invalidate the corresponding frame when requested via API `NvEncInvalidateRefFrames`.

Intra Refresh

Reference picture invalidation technique described in Section Reference Picture Invalidation depends upon availability of an out-of-band upstream channel to report bitstream errors at the decoder (client side). When such an upstream channel is not available, or in situations where bitstream is more likely to suffer from more frequent errors, intra-refresh mechanism can be used as an error recovery mechanism. Also, when using infinite GOP length, no intra frames are transmitted and intra refresh may be a useful mechanism for recovery from transmission errors.

NVENCODE API provides a mechanism to implement intra refresh. The `enableIntraRefresh` flag should be set to 1 in order to enable intra refresh. `intraRefreshPeriod` determines the period after which intra refresh would happen again and `intraRefreshCnt` sets the number of frames over which intra refresh would happen.

Intra Refresh causes consecutive sections of the frames to be encoded using intra macroblocks, over `intraRefreshCnt` consecutive frames. Then the whole cycle repeats after `intraRefreshPeriod` frames from the first intra-refresh frame. It is essential to set `intraRefreshPeriod` and `intraRefreshCnt` appropriately based on the probability of errors that may occur during transmission. For example, `intraRefreshPeriod` may be small like 30 for a highly error prone network thus enabling recovery every second for a 30 FPS video stream. For networks that have lesser chances of error, the value may be set higher. Lower value of `intraRefreshPeriod` comes with a slightly lower quality as a larger portion of the overall macroblocks in an intra refresh *period* are forced to be intra coded, but provides faster recovery from network errors.

`intraRefreshCnt` determines the number of frames over which the intra refresh will happen within an intra refresh period. A smaller value of `intraRefreshCnt` will refresh the entire frame quickly (instead of refreshing it slowly in *bands*) and hence enable a faster error recovery. However, a lower `intraRefreshCnt` also means sending a larger number of intra macroblocks per frame and hence slightly lower quality.

The default NVENCODE API Intra Refresh behavior is slice based for H.264/HEVC and tile based for AV1 i.e frames in an intra refresh wave will have multiple slices/tiles with one slice/tile containing only intra coded MBs / CTUs / SBs.

- For AV1, the number of tiles used during the intra refresh wave is automatically determined by the driver based on the value of `intraRefreshCnt` and `intraRefreshPeriod`. Any custom tiles configuration specified by the application will be ignored for the duration of the intra refresh wave.
- If the application does not explicitly specify the number of slices or if the specified number of slices are less than 3, during the intra refresh wave, the driver will set 3 slices per frame.
- For `NV_ENC_CONFIG_H264::sliceMode` = 0 (MB based slices), 2 (MB row based slices) and 3 (number of slices), the driver will maintain slice count, equal to minimum of: the slice count calculated from slice mode setting and `intraRefreshCnt` number of slices during intra refresh period.
- For `NV_ENC_CONFIG_H264::sliceMode` = 1 (byte based slices), the number of slices during an intra refresh wave is always 3.

For certain usecases, clients may want to avoid multiple slices in a frame. In such scenarios, clients can enable single slice intra refresh.

- Query the support for single slice intra refresh for the current driver by calling the API `NvEncGetEncodeCaps` and checking for `NV_ENC_CAPS_SINGLE_SLICE_INTRA_REFRESH`.

- If supported, single slice intra refresh can be enabled by setting `NV_ENC_CONFIG_H264::singleSliceIntraRefresh` / `NV_ENC_CONFIG_HEVC::singleSliceIntraRefresh`.

In case there in a resolution reconfiguration in the middle of an intra refresh wave, the ongoing wave will be terminated immediately. The next wave will start after `NV_ENC_CONFIG_H264::intraRefreshPeriod` number of frames.

Intra refresh is applied in encode order and only on frames which can be used as reference.

8.13. Multi NVENC Split Frame Encoding in HEVC and AV1

When Split frame encoding is enabled, each input frame is partitioned into horizontal strips which are encoded independently and simultaneously by separate NVENCs, usually resulting in increased encoding speed compared to single NVENC encoding.

Please note the following:

1. Though the feature improves the encoding speed it degrades quality.
2. The overall encode throughput (total number of frames encoded in a certain time interval when all NVENCs are fully utilized) will remain the same.
3. The feature is available only for HEVC and AV1.

This feature should therefore be used to achieve higher encoding speeds in a single encode session which would not have been possible on a single NVENC because horizontal strips of particular input stream are encoded simultaneously across multiple NVENCs. As mentioned above, this feature does not impact the overall throughput when multiple encode sessions are created to fully utilize all the NVENCs.

There are two modes of enabling the feature listed below.

1. Auto Mode: The conditions that automatically trigger this feature are:

- Number of NVENCs on GPU: 2 or more.
- Frame height: must be 2112 pixels or more for HEVC and 2048 pixels or more for AV1.
- Preset and Tuning Info configuration: Table 2 summarizes the preset and tuning info combinations that enable split frame encoding

Table 2. Preset configurations enabling Split Frame Encoding

Tuning Info	Preset						
	P1	P2	P3	P4	P5	P6	P7
High Quality	Yes	Yes	No	No	No	No	No
Low Latency	Yes	Yes	Yes	Yes	No	No	No

Tuning Info	Preset						
	P1	P2	P3	P4	P5	P6	P7
Ultra Low Latency	Yes	Yes	Yes	Yes	No	No	No

2. User controlled mode:

The following modes are supported for split encoding in case of HEVC and AV1:

- NV_ENC_SPLIT_ENCODE_MODE::NV_ENC_SPLIT_AUTO_MODE: In this mode split encoding will be automatically enabled only in configurations described above. It will be disabled in all other configurations.
- NV_ENC_SPLIT_ENCODE_MODE::NV_ENC_SPLIT_AUTO_FORCED_MODE: Split encoding will be enabled for all configurations with number of horizontal strips automatically selected by driver for optimal performance.
- NV_ENC_SPLIT_ENCODE_MODE::NV_ENC_SPLIT_TWO_FORCED_MODE: Split encoding will be enabled for all configurations with number of horizontal strips forced to 2 when number of NVENCs > 1.
- NV_ENC_SPLIT_ENCODE_MODE::NV_ENC_SPLIT_THREE_FORCED_MODE: Split encoding will be enabled for all configurations with number of horizontal strips forced to 3 when number of NVENCs > 2, NVENC number of strips otherwise.
- NV_ENC_SPLIT_ENCODE_MODE::NV_ENC_SPLIT_DISABLE: Split encoding will be disabled for all configurations.

Note that a few encoding features are incompatible with the use of split frame encoding. Split frame encoding is always disabled when any of the following features is in use:

1. Weighted Prediction (HEVC).
2. Alpha Layer Encoding (HEVC).
3. Bitstream Subframe Readback Mode (HEVC)
4. Bitstream Output in Video Memory (HEVC/AV1)

8.14. NVENC Reconstructed Frame Output

Starting SDK 12.1, NVENCODE API supports reconstructed frame output for H.264, HEVC and AV1 encode for Turing and later GPUs. This is helpful in use-cases which require reconstructed frame output to access the encode quality and therefore eliminates the need for decoding the stream leading to overall improvement in performance. The reconstructed frame buffer should be allocated by the client application as a 2-dimensional buffer. The availability of this feature in the current hardware can also be queried using NvEncGetEncodeCaps() and checking for NV_ENC_CAPS_OUTPUT_RECON_SURFACE. Supported buffer formats are: NV_ENC_BUFFER_FORMAT_NV12 and NV_ENC_BUFFER_FORMAT_YUV420_10BIT. For CUDA interface NV_ENC_BUFFER_FORMAT_YUV444 and NV_ENC_BUFFER_FORMAT_YUV444_10BIT are also supported.

Follow these steps for the reconstructed frame output:

1. Set `NV_ENC_INITIALIZE_PARAMS::enableReconFrameOutput = 1` when calling `nvEncInitializeEncoder()`.
2. Allocate 2-dimensional buffer for NVENC to write the reconstructed frame output.
   - For CUDA interface, this buffer can be created using `cuMemAllocPitch()`.
   - For DirectX 9 interface, this buffer can be created using `CreateOffscreenPlainSurface()` or `CreateSurface()` APIs.
   - For DirectX 11 interface, this buffer can be created using DirectX 11 `CreateTexture2D()` API, by specifying `usage = D3D11_USAGE_DEFAULT`; `BindFlags = (D3D11_BIND_SHADER_RESOURCE)`; and `CPUAccessFlags = 0`;
3. Register this buffer using `nvEncRegisterResource()`, by specifying:
   - `NV_ENC_REGISTER_RESOURCE::bufferUsage = NV_ENC_OUTPUT_RECON`
   - Set `NV_ENC_REGISTER_RESOURCE::bufferFormat` to desired value. `NvEncRegisterResource()` will return a registered handle in `NV_ENC_REGISTER_RESOURCE::registeredResource`.
4. Set `NV_ENC_MAP_INPUT_RESOURCE::registeredResource` to `NV_ENC_REGISTER_RESOURCE::registeredResource`, which was obtained in the previous step.
5. Call `nvEncMapInputResource()`. It will return a mapped resource handle in `NV_ENC_MAP_INPUT_RESOURCE::mappedResource`.
6. Call `nvEncEncodePicture()` by setting `NV_ENC_PIC_PARAMS::outputReconBuffer` to `NV_ENC_MAP_INPUT_RESOURCE::mappedResource` and `NV_ENC_PIC_PARAMS::encodePicFlags` to `NV_ENC_PIC_FLAG_OUTPUT_RECON_FRAME`.

When reading the reconstructed output, observe the following:

After calling `nvEncEncodePicture()`, client can use the output buffer for further processing only after un-mapping this output buffer.

When operating in asynchronous mode, client application should wait on event before reading the output. In synchronous mode no event is triggered, and the synchronization is handled internally by NVIDIA driver.

To access the reconstructed frame output, follow these steps:

1. Client must un-map the reconstructed frame buffer by calling `nvEncUnmapInputResource()` with mapped resource handle `NV_ENC_MAP_INPUT_RESOURCE::mappedResource` returned by `nvEncMapInputResource()`. After this, the output reconstructed buffer can be used for further processing/reading etc.
2. If CUDA mode is specified, to get the output in system memory, reconstructed output buffer can be read by calling any CUDA API (e.g. `cuMemcpyDtoH()`). The driver ensures that the output buffer is read only after NVENC has finished writing the output in it.
3. For DX11 mode, any DirectX 11 API can be used to read the output. The driver ensures that the output buffer is read only after NVENC has finished writing the output in it. To get the output in system memory, `CopyResource()` (which is a DirectX 11 API) can be used to copy the data in a CPU readable staging buffer. This staging buffer then can be read after calling `Map()` which is a DirectX 11 API.

8.15. Encoded Frame Stats

Starting SDK 12.1, NVENCODE API supports encoded frame stats output for H.264, HEVC and AV1 encode for Turing and later GPUs. The availability of this feature in the current hardware can be queried using `NvEncGetEncodeCaps()` and checking for `NV_ENC_CAPS_OUTPUT_ROW_STATS` or `NV_ENC_CAPS_OUTPUT_BLOCK_STATS`. This is helpful in use-cases which require encoded frame stats: QP and bitcount, at row or block level.

Follow these steps for the encoded frame output stats:

1. Set `NV_ENC_INITIALIZE_PARAMS::enableOutputStats = 1` and `NV_ENC_INITIALIZE_PARAMS::outputStatsLevel` to `NV_ENC_OUTPUT_STATS_ROW_LEVEL` or `NV_ENC_OUTPUT_STATS_BLOCK_LEVEL` when calling `nvEncInitializeEncoder()`. `NV_ENC_OUTPUT_STATS_ROW_LEVEL` is supported for Turing and Ampere GPUs. `NV_ENC_OUTPUT_STATS_BLOCK_LEVEL` is supported for ADA and later architectures.
2. Set the following parameters to get encoded frame stats for every row:
   a). `NV_ENC_LOCK_BITSTREAM::outputStatsPtrSize=sizeof(NV_ENC_OUTPUT_STATS_ROW) x Number of rows.`
   b). `Number of rows for H.264 = (PicHeight + 15 ) >> 4`
   c). `Number of rows for HEVC = (PicHeight + 31 ) >> 5`
3. Set the following parameters to get encoded frame stats for every block:
   a). `NV_ENC_LOCK_BITSTREAM::outputStatsPtrSize=sizeof(NV_ENC_OUTPUT_STATS_BLOCK) x Number of blocks.`
   b). `Number of blocks for H.264 = (PicWidth + 15 ) >> 4 * (PicHeight + 15 ) >> 4`
   c). `Number of blocks for HEVC = (PicWidth + 31 ) >> 5 * (PicHeight + 31 ) >> 5`
   d). `Number of blocks for AV1 = (PicWidth + 63 ) >> 6 * (PicHeight + 63 ) >> 6`
4. Allocate system memory buffer of size `NV_ENC_LOCK_BITSTREAM::outputStatsPtrSize` and assign it to `NV_ENC_LOCK_BITSTREAM::outputStatsPtr` and then call `NvEncLockBitstream()` API.
5. Read the encoded frame stats in `NV_ENC_OUTPUT_STATS_BLOCK` or `NV_ENC_OUTPUT_STATS_ROW` format from `NV_ENC_LOCK_BITSTREAM::outputStatsPtr`.
6. Call `NvEncUnlockBitstream()` API.

8.16. Iterative encoding

Starting SDK 12.1, NVENCODE API supports iterative encoding for H.264, HEVC and AV1 encoders for Turing and later GPUs. The availability of this feature in the current hardware can be queried using `NvEncGetEncodeCaps()` and checking for `NV_ENC_CAPS_DISABLE_ENC_STATE_ADVANCE`. Using this feature the same frame can be encoded multiple times, for e.g., each time with a different QP or delta-QP. NVENC stores all

new states, corresponding to each of these iterations in its internal state buffers. NVENC state can be then advanced to any one of the iterations using `NvEncRestoreEncoderState()` API.

Steps to enable iterative encoding:

1. Set `NV_ENC_INITIALIZE_PARAMS::numStateBuffers` to desired value when calling `NvEncInitializeEncoder()` API. Maximum number of state buffers which can be allocated is 16 for H.264 and HEVC and 32 for AV1.

Application can call `NvEncReconfigureEncoder()` API to set desired encoding parameters, for e.g. different QP values, before every iteration of the frame. Alternatively, it can also set `NV_ENC_PIC_PARAMS::qpDeltaMap` array to desired value for encoding current iteration of the frame.

Iterative encoding when picture type decision(PTD) is taken by application:

Follow these steps to encode the same frame multiple times:

1. Set `NV_ENC_PIC_PARAMS::encodePicFlags` to `NV_ENC_PIC_FLAG_DISABLE_ENC_STATE_ADVANCE`.
2. Set `NV_ENC_PIC_PARAMS::frameIdx` to a valid value. It must be the same for all iterations of the frame.
3. Set `NV_ENC_PIC_PARAMS::stateBufferIdx` to the desired index to save the encoder state in internal state buffer. Different state buffer index should be specified for each iteration, so that it can be used later to advance the encoder state.
4. Call `NvEncEncodePicture()` API. It must return `NV_ENC_SUCCESS`.
5. Repeat above steps 1-4 as many times as needed to encode multiple iterations of the same frame with different encoding parameters. Since encoder states are saved in internal state buffers, the maximum number of iterations for the frame will depend on the number of state buffers which are available to save the state.
6. Call `NvEncLockBitstream()` API for all iterations to get the encoded output. Application can also call `NvEncLockBitstream()` API immediately after every iteration.
7. Select internal state buffer index for advancing the encoder state and assign it to `NV_ENC_RESTORE_ENCODER_STATE_PARAMS::bufferIdx` for the chosen iteration. Also, select the type of state i.e. NV_ENC_STATE_RESTORE_TYPE to update and assign it to `NV_ENC_RESTORE_ENCODER_STATE_PARAMS::state`. Call `NvEncRestoreEncoderState()` API.
8. If application choses a state type other than `NV_ENC_STATE_RESTORE_FULL`, then the application must call `NvEncRestoreEncoderState()` API twice, once with `NV_ENC_RESTORE_ENCODER_STATE_PARAMS::state` set to `NV_ENC_STATE_RESTORE_ENCODE` and then again with `NV_ENC_RESTORE_ENCODER_STATE_PARAMS::state` set to `NV_ENC_STATE_RESTORE_RATE_CONTROL`, each time with desired state buffer index and not necessarily in same order.
9. To encode the next frame, increment `NV_ENC_PIC_PARAMS::frameIdx` and repeat steps 1-8.

Iterative encoding for H.264 and HEVC when picture type decision(PTD) is taken by NVENCODE API

Follow these steps to encode the same frame multiple times:

1. Set `NV_ENC_PIC_PARAMS::encodePicFlags`, `NV_ENC_PIC_PARAMS::frameIdx` and `NV_ENC_PIC_PARAMS::stateBufferIdx` to valid values, as mentioned in above section and call `NvEncEncodePicture()` API. This API will return either `NV_ENC_SUCCESS` or `NV_ENC_ERR_NEED_MORE_INPUT` status.
2. If it returns `NV_ENC_SUCCESS`, the application can do iterative encoding on this frame now.
3. If it returns `NV_ENC_ERR_NEED_MORE_INPUT`, the application can not do the iterative encoding on this frame right now. Application must send the next frames for encoding, until it returns `NV_ENC_SUCCESS`. Application can now do iterative encoding on this frame for which `NV_ENC_SUCCESS` is returned.
4. Call `NvEncLockBitstream()` API to get the encoded output for the first iteration.
   a). If `NV_ENC_LOCK_BITSTREAM::frameIdxDisplay` is same as `NV_ENC_PIC_PARAMS::frameIdx` in step 1 or 3, `NvEncLockBitstream()` API must be called for all remaining iterations.
   b). In some cases, `NV_ENC_LOCK_BITSTREAM::frameIdxDisplay` may be different than `NV_ENC_PIC_PARAMS::frameIdx` indicating different frame is received than the one on which iterative encoding was done in step 1 or 3. In this case, application must do iterative encoding on the frame corresponding to `NV_ENC_LOCK_BITSTREAM::frameIdxDisplay`, if needed. `NvEncLockBitstream()` API must be called for all iterations of this frame to get the encoded outputs, followed by `NvEncRestoreEncoderState()` API, to restore the state. NVENCODE API will save encoding parameters for all iterations of the frame corresponding to `NV_ENC_PIC_PARAMS::frameIdx` in step 1 or 3 and will send them for encoding after all previous frames (for which `NV_ENC_ERR_NEED_MORE_INPUT` status was returned) are encoded and the encoder state is restored for all of them.
5. Call `NvEncRestoreEncoderState()` API for the chosen frame iteration.
   a). If `NV_ENC_ERR_NEED_MORE_INPUT` was returned for any of the frames, NVENCODE API will now send one of those frames for encoding.
   b). If `NV_ENC_PIC_PARAMS::encodePicFlags` is not set to `NV_ENC_PIC_FLAG_DISABLE_ENC_STATE_ADVANCE` for the frame which is sent for encoding in step a), subsequent frame will also be sent for encoding.
6. If there are frames for which `NV_ENC_ERR_NEED_MORE_INPUT` was returned, application must call `NvEncLockBitstream()` API. `NV_ENC_LOCK_BITSTREAM::frameIdxDisplay` will indicate the frame on which iterative encoding can be done now.
7. Repeat above step for all frames for which `NV_ENC_ERR_NEED_MORE_INPUT` status was returned.

Buffer reordering:

1. Driver does the buffer reordering when B frames are present. Buffer reordering is done for output bitstream buffer, completion event and reconstructed frame buffer.
2. This reordering is done so that the application can get output in decode order and does not have to take care of picture types.
3. For state buffer index, there is no reordering.

Following table describes the API calls and the buffers which will have the encoded output, reconstructed frame output and internal states:

Table 3. API calls for iterative encoding for H.264 and HEVC

S. no.	API calls	Return parameters	Comments
1	NvEncEncodePicture (I1, N1=1, O1, E1, R1, F1=0)	NV_ENC_SUCCESS	
2	NvEncLockBitstream(O1)	frameIdxDisplay=1, picType: NV_ENC_PIC_TYPE_I	
3	NvEncEncodePicture (I2, N2=2, O2, E2, R2, S1, F2=1)	NV_ENC_ERR _NEED_MORE_INPUT	
4	NvEncEncodePicture (I3, N3=3, O3, E3, R3, S2, F3=1)	NV_ENC_SUCCESS	
5	NvEncEncodePicture (I3, N3=3, O4, E4, R4, S3, F4=1)	NV_ENC_SUCCESS	
6	NvEncLockBitstream(O2)	frameIdxDisplay=3, picType: NV_ENC_PIC_TYPE_P	Output for iteration 1 of frame 3, Recon output in R2, Internal state saved in S2
7	NvEncLockBitstream(O3)	frameIdxDisplay=3, picType: NV_ENC_PIC_TYPE_P	Output for iteration 2 of frame 3, Recon output in R3, Internal state saved in S3
8	NvEncRestoreEncoderState(S2 or S3)	NV_ENC_SUCCESS	Frame 2 will be sent for encoding
9	NvEncLockBitstream (O4)	frameIdxDisplay=2, picType: NV_ENC_PIC_TYPE_B	Output for iteration 1 of frame 2, Recon output in R4, Internal state saved in S1
10	NvEncEncodePicture (I2, N2=2, O5, E5, R5, S4, F5=1)	NV_ENC_SUCCESS	

S. no.	API calls	Return parameters	Comments
11	NvEncLockBitstream(O5)	frameIdxDisplay=2, picType: NV_ENC_PIC_TYPE_B	Output for iteration 2 of frame 2, Recon output in R5, Internal state saved in S4
12	NvEncRestoreEncoderState(S1 or S4)	NV_ENC_SUCCESS	

Note: I1, N1, O1, E1, R1, S1 represent input buffer, frame index, output buffer, completion event, reconstructed buffer and state buffer index for the first iteration of the first frame. F1=0 represents NV_ENC_PIC_PARAMS::encodePicFlags. is not set to NV_ENC_PIC_FLAG_DISABLE_ENC_STATE_ADVANCE. F1=1 represents NV_ENC_PIC_PARAMS::encodePicFlags is set to NV_ENC_PIC_FLAG_DISABLE_ENC_STATE_ADVANCE.

Iterative encoding for AV1 when picture type decision(PTD) is taken by NVENCODE API:

Due to the presence of non-displayable frames, iterative encoding for AV1 has few differences when compared with H.264 and HEVC encode. Non-displayable frames are enabled when NV_ENC_CONFIG::frameIntervalP is set to greater than 1. This section describes those differences in detail.

Follow these steps to encode the same frame multiple times:

1. Set NV_ENC_PIC_PARAMS::encodePicFlags, NV_ENC_PIC_PARAMS::frameIdx and NV_ENC_PIC_PARAMS::stateBufferIdx to valid values, as mentioned in above section and call NvEncEncodePicture() API. This API will return NV_ENC_SUCCESS or NV_ENC_ERR_NEED_MORE_INPUT status.
2. If it returns NV_ENC_SUCCESS, application can do iterative encoding on this frame now.
3. If it returns NV_ENC_ERR_NEED_MORE_INPUT, the application can not do the iterative encoding on this frame right now. Application must send the next frames for encoding, until it returns NV_ENC_SUCCESS. Application can now do iterative encoding on this frame for which NV_ENC_SUCCESS is returned.
4. Call NvEncLockBitstream() API to get the encoded output for the first iteration.

   a). If NV_ENC_LOCK_BITSTREAM::frameIdxDisplay is same as NV_ENC_PIC_PARAMS::frameIdx in step 1 or 3, NvEncLockBitstream() API must be called for all remaining iterations.

   b). In some cases, NV_ENC_LOCK_BITSTREAM::frameIdxDisplay may be different than NV_ENC_PIC_PARAMS::frameIdx indicating different frame is received than the one on which iterative encoding was done in step 1 or 3. In this case, application must do iterative encoding on the frame corresponding to NV_ENC_LOCK_BITSTREAM::frameIdxDisplay, if needed. NvEncLockBitstream() API

must be called for all iterations of this frame to get the encoded outputs, followed by `NvEncRestoreEncoderState()` API, to restore the state. NVENCODE API will save encoding parameters for all iterations for the frame corresponding to `NV_ENC_PIC_PARAMS::frameIdx` in step 1 or 3 and will send them for encoding after all previous frames (for which `NV_ENC_ERR_NEED_MORE_INPUT` status was returned) are encoded and the encoder state is restored for all of them.

c). If `NV_ENC_PIC_PARAMS::encodePicFlags` was set to `NV_ENC_PIC_FLAG_DISABLE_ENC_STATE_ADVANCE` for `NV_ENC_LOCK_BITSTREAM::frameIdxDisplay` frame and there were frames prior to it for which `NV_ENC_ERR_NEED_MORE_INPUT` status was returned, then the received encoded frame will be non-displayable frame.

d). For any non-displayable frame, corresponding OVERLAY frame would be encoded just once, after all frames prior to this frame for which `NV_ENC_ERR_NEED_MORE_INPUT` was returned are encoded, regardless of the number of iterations for this frame.

5. Application must call `NvEncRestoreEncoderState()` API to restore this frame. It may return `NV_ENC_ERR_NEED_MORE_OUTPUT` or `NV_ENC_SUCCESS` status.

   a). If it returns `NV_ENC_ERR_NEED_MORE_OUTPUT`, application must call `NvEncRestoreEncoderState()` API again with an output buffer as input in `NV_ENC_RESTORE_ENCODER_STATE_PARAMS::outputBitstream`. Application must send completion event as input in `NV_ENC_RESTORE_ENCODER_STATE_PARAMS::completionEvent`, if asynchronous mode of encoding is enabled.

   b). If `NvEncRestoreEncoderState()` API returns `NV_ENC_SUCCESS`, NVENCODE API will now send one of the frames for which `NV_ENC_ERR_NEED_MORE_INPUT` was returned for encoding.

   c). If `NV_ENC_PIC_PARAMS::encodePicFlags` is not set to `NV_ENC_PIC_FLAG_DISABLE_ENC_STATE_ADVANCE` for the frame which is sent for encoding in step b), subsequent frame will also be sent for encoding.

6. Call `NvEncLockBitstream()` API if there are frames for which `NV_ENC_ERR_NEED_MORE_OUTPUT` was returned.

   a). Application should expect the encoded output for these frames in same order in which they were sent for encoding.

   b). If `NV_ENC_LOCK_BITSTREAM::frameIdxDisplay` is not in same order, it indicates that non-displayable frame is received.

   c). Application can now do iterative encoding on the frame corresponding to `NV_ENC_LOCK_BITSTREAM::frameIdxDisplay`, if needed.

   d). For any non-displayable frame, corresponding OVERLAY frame would be encoded just once, after all frames prior to this frame for which `NV_ENC_ERR_NEED_MORE_INPUT` was returned are encoded.

7. Repeat steps 5) and 6) for all frames for which `NV_ENC_ERR_NEED_MORE_OUTPUT` was returned.

Following table describes the API calls and the buffers which will have the encoded output, reconstructed frame output and internal states:

Table 4. API calls for iterative encoding for AV1.

S. no.	API calls	Return parameters	Comments
1	`NvEncEncodePicture (I1, N1=1, O1, E1, R1, F1=0)`	`NV_ENC_SUCCESS`	
2	`NvEncLockBitstream(O1)`	`frameIdxDisplay=1, picType: NV_ENC_PIC_TYPE_I`	Recon output in R1
3	`NvEncEncodePicture (I2, N2=2, O2, E2, R2, S1, F2=1)`	`NV_ENC_ERR _NEED_MORE_INPUT`	
4	`NvEncEncodePicture (I3, N3=3, O3, E3, R3, S2, F3=1)`	`NV_ENC_SUCCESS`	
5	`NvEncEncodePicture (I3, N3=3, O4, E4, R4, S3, F4=1)`	`NV_ENC_SUCCESS`	
6	`NvEncLockBitstream(O2)`	`frameIdxDisplay=3, picType: NV_ENC_PIC_TYPE_P`	Output for iteration 1 of frame 3, Recon output in R3, Internal state saved in S2 Non-displayable frame
7	`NvEncLockBitstream(O3)`	`frameIdxDisplay=3, picType: NV_ENC_PIC_TYPE_P`	Output for iteration 2 of frame 3, Recon output in R4, Internal state saved in S3, Non-displayable frame
8	`NvEncRestoreEncoderState(S2 or S3)`	`NV_ENC_ERR_NEED _MORE_OUTPUT`	Application must call this API again
9	`NvEncRestoreEncoderState(O5, E5, S2 or S3)`	`NV_ENC_SUCCESS`	Frame 2 will be sent for encoding
10	`NvEncLockBitstream (O4)`	`frameIdxDisplay=2, picType: NV_ENC_PIC_TYPE_B`	Output for iteration 1 of frame 2, Recon output in R2, Internal state saved in S1
11	`NvEncEncodePicture (I2, N2=2, O6, E6, R5, S4, F5=1)`	`NV_ENC_SUCCESS`	

S. no.	API calls	Return parameters	Comments
12	NvEncLockBitstream(O5)	frameIdxDisplay=2, picType: NV_ENC_PIC_TYPE_B	Output for iteration 2 of frame 2, Recon output in R5, Internal state saved in S4
13	NvEncRestoreEncoderState(S1 or S4)	NV_ENC_SUCCESS	OVERLAY frame corresponding to frame 3 will be sent for encoding
14	NvEncLockBitstream(O6)	frameIdxDisplay=3, picType: NV_ENC_PIC_TYPE_P	OVERLAY frame corresponding to frame 3

Note: I1, N1, O1, E1, R1, S1 represent input buffer, frame index, output buffer, completion event, reconstructed buffer, state buffer index for the first iteration of the first frame. F1=0 represents NV_ENC_PIC_PARAMS::encodePicFlags. is not set to NV_ENC_PIC_FLAG_DISABLE_ENC_STATE_ADVANCE. F1=1 represents NV_ENC_PIC_PARAMS::encodePicFlags is set to NV_ENC_PIC_FLAG_DISABLE_ENC_STATE_ADVANCE. When picture type decision is taken by application, there is no reordering for reconstructed buffer. Application must use NV_ENC_LOCK_BITSTREAM::frameIdxDisplay to track this buffer.

8.17. External lookahead

Starting SDK 12.1, NVENCODE API supports external lookahead for H.264, HEVC and AV1 encoders for Turing and later GPUs. External lookahead gives same result as internal lookahead, which is enabled by just setting NV_ENC_RC_PARAMS:: lookaheadDepth. Internal lookahead is not supported for iterative encoding. So, the advantage external lookahead feature has over internal lookahead is that it can be used along with iterative encoding.

Follow these steps for using external lookahead:

1. Set NV_ENC_RC_PARAMS::enableExtLookahead to 1 and Set NV_ENC_RC_PARAMS::lookaheadDepth to the desired value when calling nvEncInitializeEncoder() API.

Application needs to do the following for every frame:

1. Call NvEncLookaheadPicture() with NV_ENC_LOOKAHEAD_PIC_PARAMS::inputBuffer set to the pointer obtained from ::NvEncCreateInputBuffer() or ::NvEncMapInputResource() APIs.
2. For lookahead depth N, application must call NvEncLookaheadPicture () API N+1 times, before calling NvEncEncodePicture() API for the first frame.

Eg. For lookahead depth equal to 4:

1. Call NvEncLookaheadPicture() API for frame 0

2. Call `NvEncLookaheadPicture()` API for frame `1`
3. Call `NvEncLookaheadPicture()` API for frame `2`
4. Call `NvEncLookaheadPicture()` API for frame `3`
5. Call `NvEncLookaheadPicture()` API for frame `4`
6. Call `NvEncEncodePicture()` API for frame `0`
7. Call `NvEncLookaheadPicture()` API for frame `5`
8. Call `NvEncEncodePicture()` API for frame `1`
9. Call `NvEncLookaheadPicture()` API for frame `6`
10. So on...

8.18. Unidirectional B Frames

Unidirectional B frames use past frames only for both L0 and L1 reference list and avoid the latency issues observed with conventional B frames. Therefore they can be used in place of P frames especially in low latency encoding. Unidirectional B frames improve video encoding quality and has little performance impact.

To use the feature, follow these steps:

- Query availability of the feature using `NvEncGetEncodeCaps` API and checking for `NV_ENC_CAPS_SUPPORT_UNIDIRECTIONAL_B` in the return value.
- During encoder initialization, set `enableUniDirectionalB = 1`:
  - This feature is currently supported for HEVC.

8.19. Lookahead Level

Lookahead level improves the video encoding quality 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. It determines the propgation of CTBs and assigns lower QP values to CTBs that propagate the maximum and also improves the video encoder's rate control accuracy. There are 4 different lookahead levels that provide for different quality and performance trade offs. NV_ENC_LOOKAHEAD_LEVEL_0 has the highest performance while NV_ENC_LOOKAHEAD_LEVEL_3 has the highest quality. Users can select the appropriate lookahead Level based on their quality/performance needs

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_LEVEL`.
2. Lookahead needs to be enabled during initialization by setting `NV_ENC_INITIALIZE_PARAMS::encodeconfig->rcParams.enableLookahead` = 1.
3. Lookahead level needs to be set during initialization by setting `NV_ENC_INITIALIZE_PARAMS::encodeconfig->rcParams.lookaheadLevel` = NV_ENC_LOOKAHEAD_LEVEL_0...3.

4. 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.
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`.

8.20. Temporal Filter

Temporal Filter tries to filter a frame based on neighbouring past and future frames and is very for natural video content captured using a camera, that would likely have sensor/other noise. Temporal Filter improves the objective quality of the video and is very useful in case of latency tolerant encoding.

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

1. Query the availability of temporal filter for the current hardware by calling the API NvEncGetEncodeCaps and checking for NV_ENC_CAPS_SUPPORT_TEMPORAL_FILTER.
2. If supported, temporal filter can be enabled during initialization by setting NV_ENC_CONFIG_HEVC::tfLevel = NV_ENC_TEMPORAL_FILTER_LEVEL_4.

Temporal Filter uses CUDA pre-processing and hence requires CUDA processing power, depending upon resolution and content.

Enabling temporal filter may result in minor degradation in encoder performance.

Chapter 9. Recommended NVENC Settings

The 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 many API's, the table below can be used as a general guideline for recommended settings for some of the popular use-cases to deliver the best encoded bitstream 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 information in Table 5 as a starting point and adjust the settings to achieve appropriate performance-quality tradeoff.

Table 5. Recommended NVENC settings for various use-cases

Use-case	Recommended settings for optimal quality and performance
Recording/Archiving	▶ High Quality Tuning Info / Ultra High Quality Tuning Info ▶ Rate control mode = VBR ▶ Very large VBV buffer size (4 seconds) ▶ B Frames* ▶ Look-ahead ▶ B frame as reference ▶ Finite GOP length (2 seconds) ▶ Adaptive quantization (AQ) enabled**
Game-casting & cloud transcoding	▶ High Quality Tuning Info / Ultra High Quality Tuning Info ▶ Rate control mode = CBR ▶ Medium VBV buffer size (1 second) ▶ B Frames* ▶ Look-ahead

Use-case	Recommended settings for optimal quality and performance
	▶ B frame as reference ▶ Finite GOP length (2 seconds) ▶ Adaptive quantization (AQ) enabled**
Low-latency use cases like game-streaming, video conferencing etc.	▶ Ultra-low latency or low latency Tuning Info ▶ Rate control mode = CBR ▶ Multi Pass – Quarter/Full (evaluate and decide) ▶ Very low VBV buffer size (e.g. single frame = bitrate/framerate) ▶ Unidirectional B Frames ▶ Infinite GOP length ▶ Adaptive quantization (AQ) enabled** ▶ Long term reference pictures*** ▶ Intra refresh*** ▶ Non-reference P frames*** ▶ Force IDR***
Lossless Encoding	▶ Lossless Tuning Info

*: Recommended for low motion games and natural video.

**: Recommended on second generation Maxwell GPUs and above.

***: These features are useful for error recovery during transmission across noisy mediums.

For usecases where the client requires reduced video memory footprint, following guidelines should be followed.

- **Avoid using B-frames.** B-frames requires additional buffers for reordering, hence avoiding B-frames would result to savings in video memory usage.
- **Reduce maximum number of reference frames.** Reducing number of maximum reference frames results in NVIDIA display driver allocating lesser number of buffers internally thereby reducing video memory footprint.
- **Use single pass rate control modes.** Two pass rate control consume additional video memory in comparison to single pass due to additional allocations for first pass encoding. Two pass rate control mode with first pass with full resolution consumes more than first pass with quarter resolution.
- **Avoid Adaptive Quantization / Weighted Prediction.** Features such as Adaptive Quantization / Weighted Prediction allocate additional buffers in video memory. These allocations can be avoided if these features are not used.
- **Avoid Lookahead.** Lookahead allocates additional buffers in video memory for frames that are buffered in the lookahead queue.
- **Avoid Temporal Filter.** Temporal filter requires neighbouring frames and and allocates additional buffers in the video memory.

- **Avoid UHQ Tuning Info.** UHQ Tuning Info enables lookahead and temporal Filter, that have higher memory requirements.

Note, however, that the above guidelines may result in some loss in encode quality. Clients are, therefore, recommended to do a proper evaluation to achieve right balance between encoded quality, speed and memory consumption.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgment, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

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

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, CUDA Toolkit, cuDNN, DALI, DIGITS, DGX, DGX-1, DGX-2, DGX Station, DLProf, GPU, Jetson, Kepler, Maxwell, NCCL, Nsight Compute, Nsight Systems, NVCaffe, NVIDIA Deep Learning SDK, NVIDIA Developer Program, NVIDIA GPU Cloud, NVLink, NVSHMEM, PerfWorks, Pascal, SDK Manager, Tegra, TensorRT, TensorRT Inference Server, Tesla, TF-TRT, Triton Inference Server, Turing, and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

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