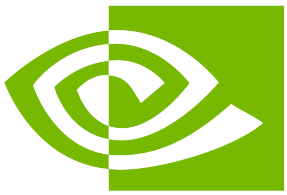

# NVIDIA PyNvVideoCodec API Reference

***Release 2.2***

**NVIDIA Corporation**

**Jul 29, 2026**

# Contents

# Chapter 1. Overview

The PyNvVideoCodec API provides Python bindings for NVIDIA's hardware-accelerated video encoding and decoding capabilities. This reference documents all public APIs organized by functional area.

## 1.1. API Summary

The following table lists all PyNvVideoCodec APIs organized by category.

| Category | API | Description |
|---|---|---|
| Decoder | *SimpleDecoder* | Straightforward decoder that reads video files and outputs decoded frames. Recommended for most users. |
| | *ThreadedDecoder* | Performance-optimized decoder with background thread prefetching, ideal for AI/ML pipelines. |
| CoreDecoder | *Demuxer* | Container parser that separates compressed video packets from container formats. |
| | *Decoder* | Core decoder class for elementary video bitstreams. Provides maximum control. |
| Encoder | *Encoder* | Video encoder class for encoding raw frames to compressed packets. |
| Transcoder | *Transcoder* | Combines decoding, encoding, and muxing for video format conversion. Supports full and segmented transcoding. |

## 1.2. Module Information

Module-level attributes and version information.

| Attribute | Description |
|---|---|
| **version** | PyNvVideoCodec version string |
| **cuda_version** | CUDA Toolkit version used to build the module |
| **video_codec_sdk_version** | NVIDIA Video Codec SDK version |

Access version information via `nvc.__version__`, `nvc.__cuda_version__`, and `nvc.__video_codec_sdk_version__`.

# Chapter 2. Decoder

## 2.1. SimpleDecoder

### 2.1.1. SimpleDecoder Class

High-level decoder class providing random access to video frames with indexing, slicing, and batch operations.

#### 2.1.1.1 Description

The `SimpleDecoder` class provides a high-level, user-friendly interface for video decoding with random frame access capabilities. It abstracts the complexities of video demuxing, seeking, and decoding behind an intuitive Python API that supports indexing, slicing, and flexible frame retrieval patterns.

Unlike `ThreadedDecoder` which is optimized for sequential processing, and the low-level `Decoder` which requires manual packet management, `SimpleDecoder` provides file-like random access to video frames, making it ideal for non-linear access patterns and exploratory workflows.

**Important Note:**

SimpleDecoder requires seekable video sources (container formats with proper index). Elementary streams (raw H.264/HEVC bitstreams without container) are not supported. Use `Decoder` or `ThreadedDecoder` if you are dealing with elementary video streams.

#### 2.1.1.2 Syntax

```
import PyNvVideoCodec as nvc

decoder = nvc.SimpleDecoder(
    enc_file_path,
    gpu_id=0,
    cuda_context=0,
    cuda_stream=0,
    use_device_memory=True,
    max_width=0,
    max_height=0,
    need_scanned_stream_metadata=False,
    decoder_cache_size=4,
    output_color_type=nvc.OutputColorType.NATIVE,
    bWaitForSessionWarmUp=False,
```

(continues on next page)

(continued from previous page)

```
    enableDecodeStats=False
)
```

#### 2.1.1.3 Constructor Parameters

| Parameter | Type | Default | Description |
|---|---|---|---|
| enc_file_path | str | Required | Path to encoded video file. Must be a seekable container format (MP4, MKV, AVI, MOV, etc.). Elementary streams are not supported |
| gpu_id | int | 0 | GPU device ID to use for decoding. Useful for multi-GPU systems |
| cuda_context | size_t | 0 | CUDA context handle. 0 uses the primary context for the specified GPU |
| cuda_stream | size_t | 0 | CUDA stream handle. 0 creates a new stream internally |
| use_device_memory | bool | True | If True, decoded frames remain in GPU memory (CUdeviceptr via CUDA Array Interface). If False, frames are copied to host memory |
| max_width | int | 0 | Maximum frame width for decoder allocation. Important for decoder reuse with multiple sources. 0 uses actual stream width |
| max_height | int | 0 | Maximum frame height for decoder allocation. Important for decoder reuse. 0 uses actual stream height |
| need_scanned_stream_metadata | bool | False | If True, performs complete stream scan on background thread to collect accurate frame count and detailed metadata. Required for precise len() on certain formats |
| decoder_cache_size | int | 4 | LRU cache size for decoder instances when using reconfigure_decoder(). Caches decoders for frequently accessed sources |
| output_color_type | OutputColorType | NATIVE | Output color format: NATIVE (NV12/YUV444/P016 depending on source), RGB (interleaved HWC), or RGBP (planar CHW) |
| bWaitForSessionWarmUp | bool | False | Wait for decoder session initialization to complete. Useful for synchronized multi-threaded scenarios |
| enableDecodeStats | bool | False | Enable decode statistics collection (motion vectors, QP values, CU types). Only available on supported hardware |

#### 2.1.1.4 Methods

**getitem (Indexing and Slicing)**

`decoder[index: int] -> DecodedFrame`

`decoder[start:stop:step] -> List[DecodedFrame]`

Provides Python-style indexing and slicing for random frame access.

See *__getitem__ (Indexing and Slicing)* for detailed documentation.

**len**

`__len__() -> int`

Returns total number of frames in the video.

See *__len__* for detailed documentation.

**get_batch_frames**

`get_batch_frames(batch_size: int) -> List[DecodedFrame]`

Retrieves a sequential batch of frames from the current decoder position.

See *get_batch_frames* for detailed documentation.

**get_batch_frames_by_index**

`get_batch_frames_by_index(indices: List[int]) -> List[DecodedFrame]`

Retrieves specific frames by their indices in arbitrary order.

See *get_batch_frames_by_index* for detailed documentation.

**get_stream_metadata**

`get_stream_metadata() -> StreamMetadata`

Returns fast stream metadata extracted from container header.

See *get_stream_metadata* for detailed documentation.

**get_scanned_stream_metadata**

`get_scanned_stream_metadata() -> ScannedStreamMetadata`

Returns accurate stream metadata by scanning entire video.

See *get_scanned_stream_metadata* for detailed documentation.

**seek_to_index**

`seek_to_index(index: int) -> None`

Moves the internal decoder position to specified frame index.

See *seek_to_index* for detailed documentation.

**get_index_from_time_in_seconds**

`get_index_from_time_in_seconds(time_in_seconds: float) -> int`

Converts time position to frame index.

See *get_index_from_time_in_seconds* for detailed documentation.

**reconfigure_decoder**

`reconfigure_decoder(new_source: str) -> None`

Reconfigures decoder to process a different video source.

See *reconfigure_decoder* for detailed documentation.

#### 2.1.1.5 getitem (Indexing and Slicing)

Python-style indexing and slicing operator for random frame access in SimpleDecoder.

##### 2.1.1.5.1 Description

The `__getitem__` method enables Python-style indexing and slicing for random frame access in the SimpleDecoder. This allows you to access frames using familiar Python syntax like `decoder[10]` for single frames or `decoder[10:20:2]` for frame ranges.

##### 2.1.1.5.2 Syntax

**Single Frame Access:**

```
frame = decoder[index]
```

**Slice Access:**

```
frames = decoder[start:stop:step]
```

##### 2.1.1.5.3 Method Signatures

```
decoder[index: int] -> DecodedFrame
```

```
decoder[start:stop:step] -> List[DecodedFrame]
```

##### 2.1.1.5.4 Parameters

**index (int)**
: Zero-based frame index for single frame access. Must be in range [0, num_frames-1]

**start:stop:step (slice)**
: Python slice notation for multiple frames. All standard Python slice semantics apply

##### 2.1.1.5.5 Returns

- **Single index:** DecodedFrame object containing the decoded frame data
- **Slice:** List of DecodedFrame objects for all frames in the slice range

##### 2.1.1.5.6 Exceptions

- `IndexError`: Raised if index is out of range [0, num_frames-1]
- `TypeError`: Raised if key type is neither int nor slice
- `ValueError`: Raised if slice produces empty range

#### 2.1.1.5.7 See Also

- *SimpleDecoder Class*
- *get_batch_frames() Method*

### 2.1.1.6 len

Returns the total number of frames in the video.

#### 2.1.1.6.1 Description

The `__len__` method enables Python's built-in `len()` function to work with SimpleDecoder objects, returning the total number of frames in the video.

#### 2.1.1.6.2 Syntax

```
total_frames = len(decoder)
```

#### 2.1.1.6.3 Method Signature

```
__len__() -> int
```

#### 2.1.1.6.4 Returns

Integer frame count. Internally it uses scanned metadata if `need_scanned_stream_metadata=True` was specified during decoder creation, otherwise uses container metadata (may be approximate or 0 for some formats).

#### 2.1.1.6.5 See Also

- *SimpleDecoder Class*
- *__getitem__ Method*

### 2.1.1.7 get_batch_frames

Retrieves a sequential batch of frames from the current decoder position.

#### 2.1.1.7.1 Description

The `get_batch_frames` method retrieves a sequential batch of frames from the current decoder position. This is the most efficient way to process video frames sequentially.

#### 2.1.1.7.2 Syntax

```
frames = decoder.get_batch_frames(batch_size)
```

#### 2.1.1.7.3 Method Signature

```
get_batch_frames(batch_size: int) -> List[DecodedFrame]
```

#### 2.1.1.7.4 Parameters

**batch_size (int)**
Number of sequential frames to retrieve

#### 2.1.1.7.5 Returns

List of DecodedFrame objects. May return fewer frames than requested if end of stream is reached. Returns empty list when no more frames are available.

#### 2.1.1.7.6 See Also

- *SimpleDecoder Class*
- *seek_to_index Method*
- *get_batch_frames_by_index Method*

### 2.1.1.8 get_batch_frames_by_index

Retrieves specific frames by their indices in arbitrary order.

#### 2.1.1.8.1 Description

The `get_batch_frames_by_index` method retrieves specific frames by their indices in arbitrary order, making it ideal for non-sequential frame access patterns.

#### 2.1.1.8.2 Syntax

```
frames = decoder.get_batch_frames_by_index(indices)
```

#### 2.1.1.8.3 Method Signature

```
get_batch_frames_by_index(indices: List[int]) -> List[DecodedFrame]
```

#### 2.1.1.8.4 Parameters

**indices (List[int])**
List of frame indices to retrieve

#### 2.1.1.8.5 Returns

List of DecodedFrame objects in the same order as requested indices.

#### 2.1.1.8.6 See Also

- *SimpleDecoder Class*
- *get_batch_frames Method*
- *__getitem__ Method*

### 2.1.1.9 get_stream_metadata

Returns fast stream metadata extracted from container header.

#### 2.1.1.9.1 Description

The `get_stream_metadata` method provides fast access to video stream metadata extracted from the container header without scanning the entire file.

#### 2.1.1.9.2 Syntax

```
metadata = decoder.get_stream_metadata()
```

#### 2.1.1.9.3 Method Signature

```
get_stream_metadata() -> StreamMetadata
```

#### 2.1.1.9.4 Returns

StreamMetadata object containing:

- `codec`: Video codec (cudaVideoCodec enum)
- `width`, `height`: Frame dimensions in pixels
- `num_frames`: Approximate frame count (may be 0 or inaccurate)
- `avg_frame_rate`: Average frame rate
- `duration`: Video duration in seconds
- `bit_rate`: Bitrate in bits per second
- `chroma_format`: Chroma subsampling format
- `bit_depth`: Bit depth per color channel

#### 2.1.1.9.5 See Also

- *SimpleDecoder Class*
- *get_scanned_stream_metadata Method*

### 2.1.1.10 get_scanned_stream_metadata

Returns accurate stream metadata by scanning entire video.

#### 2.1.1.10.1 Description

The `get_scanned_stream_metadata` method returns accurate stream metadata by scanning the entire video file. This provides precise frame counts and keyframe locations.

#### 2.1.1.10.2 Syntax

```
scanned_metadata = decoder.get_scanned_stream_metadata()
```

#### 2.1.1.10.3 Method Signature

```
get_scanned_stream_metadata() -> ScannedStreamMetadata
```

#### 2.1.1.10.4 Returns

ScannedStreamMetadata object with accurate frame count and keyframe locations.

#### 2.1.1.10.5 Exceptions

- Raises `Exception` if decoder was created with `need_scanned_stream_metadata=False`

#### 2.1.1.10.6 See Also

- *SimpleDecoder Class*
- *get_stream_metadata Method*

### 2.1.1.11 seek_to_index

Moves the internal decoder position to specified frame index.

#### 2.1.1.11.1 Description

The `seek_to_index` method moves the internal decoder position to a specified frame index, affecting subsequent calls to `get_batch_frames()`.

#### 2.1.1.11.2 Syntax

```
decoder.seek_to_index(index)
```

#### 2.1.1.11.3 Method Signature

```
seek_to_index(index: int) -> None
```

#### 2.1.1.11.4 Parameters

**index (int)**
Target frame index (zero-based)

#### 2.1.1.11.5 Exceptions

- `IndexError`: If index is out of valid range

#### 2.1.1.11.6 See Also

- *SimpleDecoder Class*
- *get_batch_frames Method*
- *get_index_from_time_in_seconds Method*

### 2.1.1.12 get_index_from_time_in_seconds

Converts time position to frame index.

#### 2.1.1.12.1 Description

The `get_index_from_time_in_seconds` method converts a time position (in seconds) to the corresponding frame index, enabling time-based video navigation.

#### 2.1.1.12.2 Syntax

```
frame_index = decoder.get_index_from_time_in_seconds(time_in_seconds)
```

#### 2.1.1.12.3 Method Signature

```
get_index_from_time_in_seconds(time_in_seconds: float) -> int
```

#### 2.1.1.12.4 Parameters

**time_in_seconds (float)**
: Time position in seconds

#### 2.1.1.12.5 Returns

Integer frame index corresponding to the time position.

#### 2.1.1.12.6 See Also

- *SimpleDecoder Class*
- *seek_to_index Method*
- *__getitem__ Method*

### 2.1.1.13 reconfigure_decoder

Reconfigures decoder to process a different video source.

#### 2.1.1.13.1 Description

The `reconfigure_decoder` method reconfigures the decoder to process a different video source, reusing the decoder instance and benefiting from internal decoder caching.

#### 2.1.1.13.2 Syntax

```
decoder.reconfigure_decoder(new_source)
```

#### 2.1.1.13.3 Method Signature

```
reconfigure_decoder(new_source: str) -> None
```

#### 2.1.1.13.4 Parameters

**new_source (str)**
Path to new video file

#### 2.1.1.13.5 See Also

- *SimpleDecoder Class*

## 2.2. ThreadedDecoder

### 2.2.1. ThreadedDecoder Class

High-performance threaded decoder class for background video decoding with automatic frame prefetching suitable for inference pipelines and high throughput pipelines.

#### 2.2.1.1 Description

The `ThreadedDecoder` class provides hardware-accelerated video decoding on a background thread, enabling near-zero frame fetch latency for CPU-bound inference pipelines. This class is specifically designed for real-time and high-performance deep learning workloads where video decoding should not become a bottleneck.

Unlike the standard `Decoder` class which decodes frames synchronously, the `ThreadedDecoder` continuously decodes frames in the background and maintains a preloaded buffer of ready-to-use frames. This approach effectively hides decoding latency by overlapping decode operations with inference processing.

#### 2.2.1.2 Syntax

```
import PyNvVideoCodec as nvc

decoder = nvc.ThreadedDecoder(
    enc_file_path,
    buffer_size,
    gpu_id=0,
    cuda_context=0,
```

(continues on next page)

(continued from previous page)

```
    cuda_stream=0,
    use_device_memory=True,
    max_width=0,
    max_height=0,
    need_scanned_stream_metadata=False,
    decoder_cache_size=4,
    output_color_type=nvc.OutputColorType.NATIVE,
    start_frame=0,
    enableDecodeStats=False
)
```

#### 2.2.1.3 Constructor Parameters

| Parameter | Type | Default | Description |
|---|---|---|---|
| enc_file_path | str | Required | Path to the encoded video file. Supports various container formats (MP4, MKV, AVI, etc.) |
| buffer_size | int | Required | Number of decoded frames to prefetch and keep in the buffer. Larger values increase memory usage but provide more buffering against pipeline stalls. Recommended: 8-16 for typical workloads |
| gpu_id | int | 0 | GPU device ID to use for decoding |
| cuda_context | size_t | 0 | CUDA context handle. 0 uses the primary context for the specified GPU |
| cuda_stream | size_t | 0 | CUDA stream handle. 0 creates a new stream internally |
| use_device_memory | bool | True | If True, decoded frames remain in GPU memory (CUdeviceptr). If False, frames are copied to host memory |
| max_width | int | 0 | Maximum frame width the decoder must support. 0 uses stream width |
| max_height | int | 0 | Maximum frame height the decoder must support. 0 uses stream height |
| need_scanned_stream_metadata | bool | False | If True, performs complete stream scan on background thread to collect accurate frame count and metadata. This may take time for large files |
| decoder_cache_size | int | 4 | LRU cache size for number of decoders to retain when using reconfigure_decoder(). Useful for switching between multiple video sources |
| output_color_type | OutputColorType | NATIVE | Output format: NATIVE (NV12/YUV444), RGB (interleaved HWC), or RGBP (planar CHW) |
| start_frame | int | 0 | Frame index to start decoding from. Decoder seeks to nearest keyframe and skips frames until reaching this index. Useful for processing video segments |
| enableDecodeStats | bool | False | If True, enables decode statistics collection (motion vectors, QP values, etc.) for each frame |

#### 2.2.1.4 Methods

**get_batch_frames**

`get_batch_frames(batch_size: int) -> List[DecodedFrame]`

Retrieves a batch of prefetched decoded frames from the internal buffer.

See *get_batch_frames* for detailed documentation.

**get_stream_metadata**

`get_stream_metadata() -> StreamMetadata`

Returns fast stream metadata extracted from container header.

See *get_stream_metadata* for detailed documentation.

**get_scanned_stream_metadata**

`get_scanned_stream_metadata() -> ScannedStreamMetadata`

Returns accurate stream metadata by scanning the entire video stream.

See *get_scanned_stream_metadata* for detailed documentation.

**reconfigure_decoder**

`reconfigure_decoder(new_source: str) -> None`

Reconfigures the decoder to process a new video source.

See *reconfigure_decoder* for detailed documentation.

**len**

`__len__() -> int`

Returns the total number of frames in the video stream.

See *__len__* for detailed documentation.

#### 2.2.1.5 How It Works

In a traditional synchronous decoding workflow, the inference pipeline must wait for each frame to be decoded before processing can begin:

The threaded decoder eliminates this inefficiency by decoding frames continuously in the background:

The decoder maintains a producer-consumer pattern using a lock-free SPSC (Single Producer Single Consumer) buffer:

- **Producer Thread:** Continuously decodes frames and pushes to buffer
- **Consumer Thread:** Application calls get_batch_frames() to pop frames
- **Synchronization:** Automatic blocking when buffer is full/empty
- **Frame Locking:** Frames remain valid until next get_batch_frames() call

#### 2.2.1.6 get_batch_frames

Retrieves a batch of prefetched decoded frames from the internal buffer.

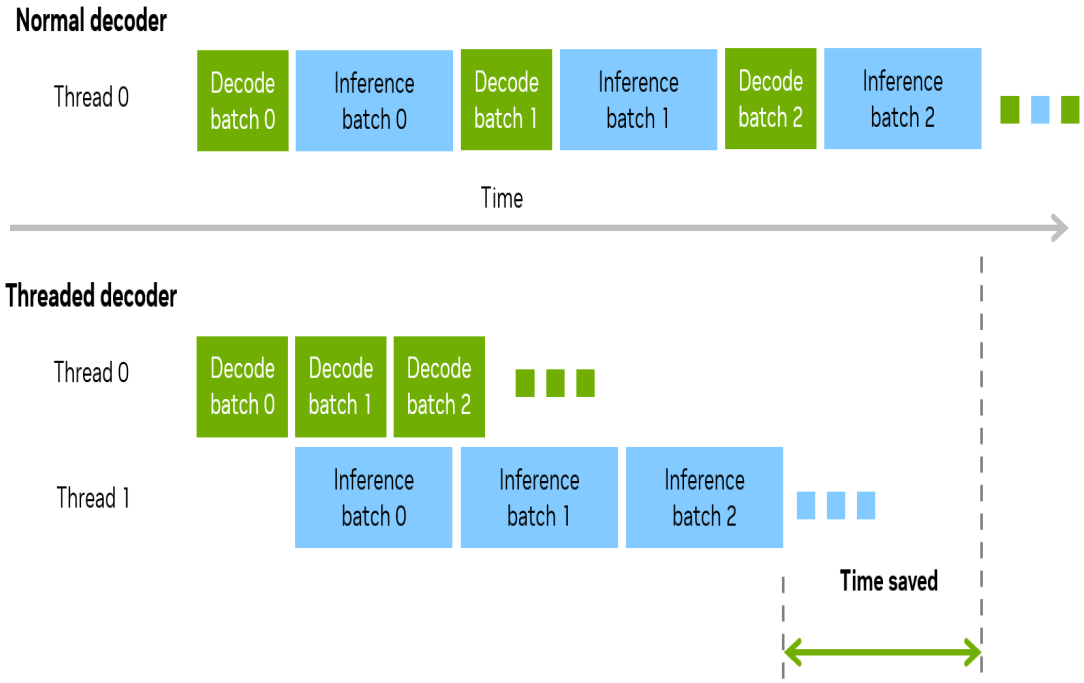


Figure 2.1: Normal decoder(e.g. Core Decoder) vs. Threaded Decoder Overlapping inference workload with decode

#### 2.2.1.6.1 Description

The `get_batch_frames` method retrieves a batch of prefetched decoded frames from the internal buffer. The frames are decoded continuously in the background, providing near-zero latency access.

#### 2.2.1.6.2 Syntax

```
frames = decoder.get_batch_frames(batch_size)
```

#### 2.2.1.6.3 Method Signature

```
get_batch_frames(batch_size: int) -> List[DecodedFrame]
```

#### 2.2.1.6.4 Parameters

**batch_size (int)**
: Number of frames to retrieve. Must be <= buffer_size specified in constructor. Use 0 to drain all remaining frames

#### 2.2.1.6.5 Returns

List of DecodedFrame objects. Empty list indicates end of stream or decoder stopped.

#### 2.2.1.6.6 Exceptions

- Raises `Exception` if batch_size exceeds buffer_size

#### 2.2.1.6.7 See Also

- *ThreadedDecoder Class*

### 2.2.1.7 get_stream_metadata

Returns fast stream metadata extracted from container header.

#### 2.2.1.7.1 Description

The `get_stream_metadata` method provides fast access to video stream metadata extracted from the container header without scanning the entire file.

#### 2.2.1.7.2 Syntax

```
metadata = decoder.get_stream_metadata()
```

#### 2.2.1.7.3 Method Signature

```
get_stream_metadata() -> StreamMetadata
```

#### 2.2.1.7.4 Returns

StreamMetadata object containing:

- `codec`: Video codec type
- `width`, `height`: Frame dimensions
- `numFrames`: Approximate frame count from container (may be 0 or inaccurate)
- `frameRate`: Frame rate as fraction (numerator/denominator)
- `bitRate`: Bitrate in bits per second
- `chromaFormat`: Chroma subsampling format
- `bitDepth`: Bit depth per channel

#### 2.2.1.7.5 See Also

- *ThreadedDecoder Class*
- *get_scanned_stream_metadata Method*

### 2.2.1.8 get_scanned_stream_metadata

Returns accurate stream metadata by scanning the entire video stream.

#### 2.2.1.8.1 Description

The `get_scanned_stream_metadata` method returns accurate stream metadata by scanning the entire video file, providing precise frame counts and additional metadata.

#### 2.2.1.8.2 Syntax

```
scanned_metadata = decoder.get_scanned_stream_metadata()
```

#### 2.2.1.8.3 Method Signature

```
get_scanned_stream_metadata() -> ScannedStreamMetadata
```

#### 2.2.1.8.4 Returns

ScannedStreamMetadata object with accurate frame count and additional metadata collected during stream scan.

#### 2.2.1.8.5 Exceptions

- Raises `Exception` if decoder was created with `need_scanned_stream_metadata=False`

#### 2.2.1.8.6 See Also

- *ThreadedDecoder Class*
- *get_stream_metadata Method*

#### 2.2.1.9 reconfigure_decoder

Reconfigures the decoder to process a new video source.

##### 2.2.1.9.1 Description

The `reconfigure_decoder` method reconfigures the decoder to process a new video source, efficiently reusing the decoder instance and benefiting from internal decoder caching.

##### 2.2.1.9.2 Syntax

```
decoder.reconfigure_decoder(new_source)
```

##### 2.2.1.9.3 Method Signature

```
reconfigure_decoder(new_source: str) -> None
```

##### 2.2.1.9.4 Parameters

**new_source (str)**
Path to new encoded video file

##### 2.2.1.9.5 See Also

- *ThreadedDecoder Class*

#### 2.2.1.10 len

Returns the total number of frames in the video stream.

##### 2.2.1.10.1 Description

The `__len__` method enables Python's built-in `len()` function to work with ThreadedDecoder objects, returning the total number of frames in the video.

##### 2.2.1.10.2 Syntax

```
total_frames = len(decoder)
```

##### 2.2.1.10.3 Method Signature

```
__len__() -> int
```

##### 2.2.1.10.4 Returns

Integer frame count. Uses scanned metadata if available (`need_scanned_stream_metadata=True`), otherwise uses container metadata (may be 0 or inaccurate).

#### 2.2.1.10.5 See Also

- *ThreadedDecoder Class*
- *get_scanned_stream_metadata Method*

# 2.3. Core Decoder

This section describes the core APIs for low-level demuxing and decoding of videos.

## 2.3.1. Demuxer

### 2.3.1.1 CreateDemuxer Function

Function for creating a file-based demuxer.

#### 2.3.1.1.1 Syntax

```
CreateDemuxer(filename: str) -> Demuxer
```

#### 2.3.1.1.2 Description

Creates a Demuxer instance for parsing video container files and extracting encoded packets.

#### 2.3.1.1.3 Parameters

**filename**
: Path to the video file to demux.

#### 2.3.1.1.4 Returns

Demuxer object for the specified file.

### 2.3.1.2 Demuxer Class

Video demuxer for extracting encoded packets from container files.

#### 2.3.1.2.1 Overview

The `Demuxer` class parses video container formats (MP4, MKV, AVI) and extracts encoded video packets for decoding. Create a demuxer using the *CreateDemuxer* function.

#### 2.3.1.2.2 Methods

| Method | Description |
|---|---|
| *GetNvCodecId()* | Returns the NVIDIA codec identifier for the video stream |
| *ChromaFormat()* | Returns the chroma subsampling format (e.g., YUV420) |
| *BitDepth()* | Returns the bit depth per color component (8, 10, or 12) |
| *FrameRate()* | Returns the frame rate in frames per second |
| Width() | Returns the video width in pixels |
| Height() | Returns the video height in pixels |
| ColorSpace() | Returns the color space (BT_601, BT_709, UNSPEC) |
| ColorRange() | Returns the color range (MPEG, JPEG, UDEF) |
| Seek(timestamp) | Seeks to the nearest keyframe before the specified timestamp |

#### 2.3.1.2.3 Iterator Protocol

The Demuxer implements the Python iterator protocol. Use a for loop to iterate over packets:

```
for packet in demuxer:
    # packet is a PacketData object
    for frame in decoder.Decode(packet):
        process_frame(frame)
```

See *Demuxer Iterator* for details.

#### 2.3.1.2.4 GetNvCodecId

Returns the NVIDIA codec identifier for the video stream.

##### 2.3.1.2.4.1 Syntax

```
demuxer.GetNvCodecId() -> cudaVideoCodec
```

##### 2.3.1.2.4.2 Description

Returns the codec identifier corresponding to the NVDEC hardware decoder. This value is used when creating a decoder with `CreateDecoder()`.

##### 2.3.1.2.4.3 Returns

`cudaVideoCodec` - The NVIDIA codec identifier (e.g., H264, HEVC, VP9, AV1).

#### 2.3.1.2.5 ChromaFormat

Returns the chroma subsampling format of the video stream.

##### 2.3.1.2.5.1 Syntax

```
demuxer.ChromaFormat() -> cudaVideoChromaFormat
```

##### 2.3.1.2.5.2 Description

Returns the chroma subsampling format of the video stream. Common formats include YUV420 (4:2:0), YUV422 (4:2:2), and YUV444 (4:4:4).

This information can be used to query decoder capabilities with `GetDecoderCaps()`.

##### 2.3.1.2.5.3 Returns

`cudaVideoChromaFormat` - The chroma format of the video stream.

#### 2.3.1.2.6 Demuxer.BitDepth

Returns the bit depth per color component of the video stream.

##### 2.3.1.2.6.1 Syntax

```
BitDepth() -> int
```

##### 2.3.1.2.6.2 Description

Returns the bit depth per color component. Common values are 8 (SDR content) and 10 (HDR content).

This information can be used to query decoder capabilities with `GetDecoderCaps()`.

##### 2.3.1.2.6.3 Returns

`int` - Bit depth per color component (typically 8, 10, or 12).

#### 2.3.1.2.7 FrameRate

Returns the frame rate of the video stream.

##### 2.3.1.2.7.1 Syntax

```
demuxer.FrameRate() -> float
```

##### 2.3.1.2.7.2 Description

Returns the average frame rate of the video stream as a floating-point value. Common values include 23.976, 24.0, 25.0, 29.97, 30.0, 50.0, 59.94, and 60.0 fps.

#### 2.3.1.2.7.3 Returns

`float` - Frame rate in frames per second.

#### 2.3.1.2.8 Iterator

Iterate over the demuxer to retrieve video packets.

#### 2.3.1.2.8.1 Syntax

```
for packet in demuxer:
    # Process packet
```

#### 2.3.1.2.8.2 Description

The Demuxer object implements the Python iterator protocol (`__iter__` and `__next__`). Each iteration extracts a single compressed video packet from the container and yields a `PacketData` object.

The iteration continues until all packets in the video stream have been extracted.

#### 2.3.1.2.8.3 Yields

`PacketData` - A packet containing compressed video data with the following properties:

- `pts` - Presentation timestamp
- `dts` - Decode timestamp
- `duration` - Packet duration
- `key` - True if this is a keyframe (I-frame)
- `bsl_data` - Bitstream data

#### 2.3.1.2.8.4 Note

- The decoder may return zero, one, or multiple frames per packet due to B-frame reordering.
- After iterating through all packets, call `decoder.Flush()` to retrieve any buffered frames.

## 2.3.2. Decoder

### 2.3.2.1 Decoder Class

Low-level decoder class for advanced video decoding with full parameter control and hardware acceleration.

#### 2.3.2.1.1 Description

The `Decoder` class (implemented as `PyNvDecoder`) provides low-level access to NVIDIA hardware-accelerated video decoding using the NVDEC engine. It is created via the `CreateDecoder` function and provides maximum flexibility for advanced applications requiring fine-grained control over the decoding process.

This class supports multiple video codecs (H.264, HEVC, AV1, VP9, etc.), various output formats (NV12, P016, YUV444, RGB), and advanced features including SEI message extraction, decode statistics, and configurable latency modes.

#### 2.3.2.1.2 Syntax

```
decoder = CreateDecoder(
    gpuid=0,
    codec=cudaVideoCodec.H264,
    cudacontext=0,
    cudastream=0,
    usedevicememory=True,
    maxwidth=0,
    maxheight=0,
    outputColorType=OutputColorType.NATIVE,
    enableSEIMessage=False,
    bWaitForSessionWarmUp=False,
    latency=DisplayDecodeLatency.DISPLAYDECODELATENCY_NATIVE,
    enableDecodeStats=False
)
```

#### 2.3.2.1.3 Methods

**Decode**

`Decode(packetData: PacketData) -> List[DecodedFrame]`

Decodes bitstream data in the packet into uncompressed frames.

See *Decode* for detailed documentation.

**GetNumDecodedFrame**

`GetNumDecodedFrame(packetData: PacketData) -> int`

Decodes bitstream data and returns the count of decoded frames without retrieving frame data.

**Parameters**

- `packetData` (PacketData): Structure containing bitstream data, size, PTS, and decode flags

**Returns**

Integer count of decoded frames available for retrieval

**GetFrame**

`GetFrame() -> DecodedFrame`

Retrieves a single decoded frame from the internal decoder buffer.

**Returns**

DecodedFrame object containing the decoded frame data, timestamp, SEI message, CUDA event, and decode statistics

**Remarks**

Call this method in a loop to fetch all available decoded frames after calling Decode().

**GetLockedFrame**

`GetLockedFrame() -> CUdeviceptr`

Returns a locked frame buffer that remains valid until explicitly unlocked.

**Returns**
CUDA device pointer to the locked frame buffer

**Remarks**
Locked frames are protected from being overwritten by subsequent decode calls. Use UnlockFrame() to release the frame buffer when processing is complete.

**UnlockFrame**

```
UnlockFrame(pFrame: CUdeviceptr) -> None
```

Unlocks a previously locked frame buffer, making it available for reuse.

**Parameters**

- `pFrame` (CUdeviceptr): Device pointer to the frame buffer to unlock

**SetSeekPTS**

```
SetSeekPTS(targetPTS: int64) -> None
```

Sets the presentation timestamp of the target frame for seeking operations.

**Parameters**

- `targetPTS` (int64): Target presentation timestamp for seeking

**Remarks**
The decoder can skip decoding frames with PTS less than the seek target, improving seek performance.

**setReconfigParams**

```
setReconfigParams(width: int = 0, height: int = 0) -> None
```

Dynamically reconfigures decoder output resolution.

See *setReconfigParams* for detailed documentation.

**GetWidth**

```
GetWidth() -> int
```

Returns the width of the decoded frame output.

**Returns**
Integer width in pixels (2-byte aligned for NV12/P016/NV16/P216 formats)

**GetHeight**

```
GetHeight() -> int
```

Returns the luma height of the decoded frame output.

**Returns**
Integer height in pixels

**GetFrameSize**

```
GetFrameSize() -> int
```

Returns the total size of the decoded frame in bytes.

**Returns**
Integer frame size in bytes, calculated based on the pixel format

**GetPixelFormat**

`GetPixelFormat() -> str`

Returns the pixel format of the decoded output.

**Returns**
String representation of the pixel format (e.g., “NV12”, “P016”, “YUV444”)

**SyncOnCUStream**

`SyncOnCUStream() -> None`

Synchronizes the decoder stream, forcing all operations to complete.

**Remarks**
Blocks until all decoder post-processing kernels and memory copies finish. Use for explicit synchronization when needed.

**GetSessionInitTime**

`GetSessionInitTime() -> int64`

Returns the decoder session initialization time in milliseconds.

**Returns**
Integer initialization time in milliseconds

**setDecoderSessionID**

`setDecoderSessionID(sessionID: int) -> None`

Sets the decoder session identifier for performance tracking.

**Parameters**

- `sessionID` (int): Session identifier

#### 2.3.2.1.4 Static Methods

**SetSessionCount**

`PyNvDecoder.SetSessionCount(numThreads: int) -> None`

Sets the expected number of concurrent decoder sessions for performance optimization.

**Parameters**

- `numThreads` (int): Number of concurrent decoder sessions

**getDecoderSessionOverHead**

`PyNvDecoder.getDecoderSessionOverHead(sessionID: int) -> int64`

Returns the cumulative session overhead (initialization + deinitialization time).

**Parameters**

- `sessionID` (int): Session identifier

**Returns**
Integer overhead time in milliseconds

#### 2.3.2.1.5 DecodedFrame Properties

The `DecodedFrame` class represents a decoded video frame with the following properties and methods:

| Property/Method | Type | Description |
|---|---|---|
| timestamp | int64 | Presentation timestamp (PTS) of the decoded frame |
| format | Pixel_Format | Pixel format enumeration (NV12, P016, YUV444, RGB, etc.) |
| decoder_stream_event | CUevent | CUDA event for synchronization with decoder operations |
| decode_stats_ptr | uint8_t* | Pointer to decode statistics buffer (if enabled) |
| decode_stats_size | size_t | Size of decode statistics buffer in bytes |
| getPTS() | int64 | Returns the presentation timestamp |
| getSEIMessage() | SEI_MESSAGE | Returns SEI message data (if enabled) |
| getRawDecodeStats() | List[uint8] | Returns raw decode statistics buffer as byte array |
| ParseDecodeStats() | Dict | Returns parsed decode statistics: qp_luma, cu_type, motion vectors |
| framesize() | int | Returns total frame size in bytes |
| cuda() | List[CAIMemoryVie | Returns CUDA Array Interface memory views |
| GetPtrToPlane(planeIdx) | CUdeviceptr | Returns device pointer to specified plane index |
| shape | List[int] | Shape of the frame buffer as array |
| strides | List[int] | Strides of the frame buffer |
| dtype | str | Data type of the buffer |
| **dlpack**(stream) | DLPackTensor | Exports frame as DLPack tensor for zero-copy interoperability |
| **dlpack_device**() | Tuple[int, int] | Returns device type and device ID for DLPack |

#### 2.3.2.1.6 CreateDecoder Function

Function for creating a low-level video decoder with full parameter control.

##### 2.3.2.1.6.1 Syntax

```
CreateDecoder(
    gpuid: int,
    codec: CudaVideoCodec,
```

(continues on next page)

(continued from previous page)

```
    cudacontext: int = 0,
    cudastream: int = 0,
    usedevicememory: bool = True,
    outputcolortype: ColorFormat = ColorFormat.NV12,
    latency: DisplayDecodeLatencyType = DisplayDecodeLatencyType.NATIVE,
    maxwidth: int = 0,
    maxheight: int = 0,
    lowlatency: bool = False
) -> Decoder
```

#### 2.3.2.1.6.2 Description

Creates a Decoder instance for hardware-accelerated video decoding with full control over decoding parameters. This is the low-level API for advanced applications requiring fine-grained control.

#### 2.3.2.1.6.3 Returns

Decoder object configured with the specified parameters.

### 2.3.2.1.7 Decode

Decodes bitstream data in a packet into uncompressed video frames.

#### 2.3.2.1.7.1 Description

The `Decode` method is the core low-level decoding operation that processes compressed bitstream data and returns decoded video frames. This method provides fine-grained control over the decoding pipeline and is suitable for custom video processing workflows.

#### 2.3.2.1.7.2 Syntax

```
frames = decoder.Decode(packet_data)
```

#### 2.3.2.1.7.3 Method Signature

```
Decode(packetData: PacketData) -> List[DecodedFrame]
```

#### 2.3.2.1.7.4 Parameters

**packetData (PacketData)**
Structure containing:

- `bitstream`: Compressed bitstream data (bytes)
- `size`: Size of bitstream in bytes
- `pts`: Presentation timestamp
- `flags`: Decode flags (e.g., end-of-stream)

#### 2.3.2.1.7.5 Returns

List of `DecodedFrame` objects containing decoded video frames with associated metadata. May return multiple frames from a single packet due to decoder buffering, or an empty list if frames are still buffered.

#### 2.3.2.1.7.6 See Also

- *Decoder Class*
- *Demuxer Class*
- *SimpleDecoder Class*

### 2.3.2.1.8 Decoder.Flush

Flushes remaining frames from the decoder's internal buffer after all input packets have been processed.

#### 2.3.2.1.8.1 Description

The `Flush` method retrieves any frames that remain in the decoder's internal buffer after all compressed bitstream packets have been submitted via `Decode()`. Because the decoder may hold multiple frames internally due to B-frame reordering, calling `Flush` after the main decode loop is required to ensure all frames are returned. It should be the final step in any Core Decoder pipeline.

#### 2.3.2.1.8.2 Syntax

```
frames = decoder.Flush()
```

#### 2.3.2.1.8.3 Method Signature

```
Flush() -> List[DecodedFrame]
```

#### 2.3.2.1.8.4 Parameters

None

#### 2.3.2.1.8.5 Returns

List of `DecodedFrame` objects containing the remaining buffered frames. Returns an empty list when the decoder buffer is fully drained.

#### 2.3.2.1.8.6 See Also

- *Decoder Class*
- *Decode Method*

### 2.3.2.1.9 Decoder.setReconfigParams

Dynamically reconfigures decoder output resolution without recreating the decoder instance.

#### 2.3.2.1.9.1 Description

The `setReconfigParams` method allows dynamic reconfiguration of the decoder's output resolution. This is particularly useful for adaptive bitrate streaming (ABR) scenarios where resolution may change mid-stream without needing to destroy and recreate the decoder.

#### 2.3.2.1.9.2 Syntax

```
decoder.setReconfigParams(width=1920, height=1080)
```

#### 2.3.2.1.9.3 Method Signature

```
setReconfigParams(width: int = 0, height: int = 0) -> None
```

#### 2.3.2.1.9.4 Parameters

| Parameter | Type | Description |
|---|---|---|
| width | int | Updated width for decoded output. Default: 0 (no change) |
| height | int | Updated height for decoded output. Default: 0 (no change) |

#### 2.3.2.1.9.5 See Also

- *Decoder Class*
- *Decode Method*

## 2.4. Decode Statistics API

Python API for extracting low-level decoding statistics including quantization parameters, coding unit types, and motion vectors.

### 2.4.1. Description

The decode statistics API provides access to detailed per-frame and per-macroblock decoding information. This functionality is based on the `cuvidGetDecodeStatus()` API from the NVDEC Video Decoder API.

Statistics collection must be explicitly enabled when creating a decoder by setting `enableDecodeStats=True`. Once enabled, each `DecodedFrame` object provides access to statistics through the `ParseDecodeStats()` method.

**Supported Codecs:** H.264 (AVC) and H.265 (HEVC)

**Supported Decoders:**

- `SimpleDecoder` - Use `enableDecodeStats=True` parameter

- `CreateDecoder` (Core Decoder) - Use `enableDecodeStats=1` parameter

## 2.4.2. Enabling Statistics Collection

**SimpleDecoder:** Use `enableDecodeStats=True` parameter when creating the decoder.

**Core Decoder (CreateDecoder):** Use `enableDecodeStats=1` parameter when creating the decoder.

## 2.4.3. DecodedFrame Statistics Properties

When statistics collection is enabled, `DecodedFrame` objects expose the following properties:

| Property | Type | Description |
|---|---|---|
| decode_stats_ptr | uint8_t* | Pointer to raw decode statistics buffer |
| decode_stats_size | size_t | Size of decode statistics buffer in bytes. Check if > 0 before parsing |

## 2.4.4. ParseDecodeStats Method

**Signature:** `ParseDecodeStats() -> Dict[str, List]`

**Description:**

Parses the raw decode statistics buffer and returns a dictionary containing per-macroblock statistics. This method processes the NVDEC decode status data and extracts quantization parameters, coding unit types, and motion vectors.

**Returns:**

A dictionary with the following keys:

| Key | Type | Description |
|---|---|---|
| qp_luma | List[int] | Luma quantization parameter for each macroblock. Range: 0-51 for H.264/HEVC. Higher values = more compression, potentially lower quality |
| cu_type | List[int] | Coding unit type for each macroblock. Values: 0=INTRA, 1=INTER, 2=SKIP, 3=PCM, 7=INVALID |
| mv0_x | List[int] | X component of primary motion vector (L0 reference list) in quarter-pixel units |
| mv0_y | List[int] | Y component of primary motion vector (L0 reference list) in quarter-pixel units |
| mv1_x | List[int] | X component of secondary motion vector (L1 reference list, B-frames only) |
| mv1_y | List[int] | Y component of secondary motion vector (L1 reference list, B-frames only) |

**Exceptions:**

- Raises `Exception` if statistics collection was not enabled
- Raises `Exception` if parse operation fails

## 2.4.5. getRawDecodeStats Method

**Signature:** `getRawDecodeStats() -> List[uint8]`

**Description:**

Returns the raw decode statistics buffer as a byte array. Use this for custom parsing or binary export of statistics data.

**Returns:**

List of unsigned 8-bit integers containing the raw statistics buffer.

## 2.4.6. Coding Unit Type Constants

The `cu_type` field contains values indicating the prediction mode for each macroblock:

| Value | Name | Description |
|---|---|---|
| 0 | INTRA | Spatial prediction from current frame only. High in I-frames and at scene changes |
| 1 | INTER | Temporal prediction using motion compensation. Most common in P/B frames |
| 2 | SKIP | Block copied from reference without residual. Efficient for static regions |
| 3 | PCM | Pulse Code Modulation - raw uncompressed values. Rare |
| 7 | INVALID | Invalid or undefined block type |

# Chapter 3. Encoder

## 3.1. Encoder Class

### 3.1.1. Encoder Overview

The `Encoder` class provides hardware-accelerated video encoding using NVIDIA's NVENC API. It encodes raw video frames into compressed bitstreams with support for multiple codecs, formats, and encoding configurations.

The Encoder is created using the `CreateEncoder` function with encoder configuration parameters.

| Parameter | Type | Description |
|---|---|---|
| width | int | Width of input frames in pixels |
| height | int | Height of input frames in pixels |
| format | str | Input surface format (e.g., "NV12", "YUV444", "P010", "ARGB") |
| use_cpu_buffer | bool | True for CPU/host memory input, False for GPU/device memory input |
| gpu_id | int | GPU device ID for encoding (default: 0) |
| cuda_context | int | CUDA context pointer (default: 0 for automatic) |
| cuda_stream | int | CUDA stream pointer (default: 0 for automatic) |
| codec | str | Output codec: "h264", "hevc", or "av1" (default: "h264") |
| preset | str | Encoding preset: "p1" through "p7" (higher values favor quality over speed) |
| bitrate | str | Target bitrate (e.g., "5M" for 5 Mbps) |
| fps | str | Frame rate (e.g., "30", "60") |
| rc | str | Rate control mode: "cbr" (Constant Bitrate), "vbr" (Variable Bitrate), "cqp" (Constant QP) |

### 3.1.2. Supported Input Formats

The encoder validates format support based on GPU capabilities:

- **Always Supported:** NV12, YUV420 (IYUV), ARGB, ABGR
- **YUV444:** Requires 4:4:4 encode support
- **P010:** Requires 10-bit encode support
- **YUV444_10BIT, YUV444_16BIT:** Requires both 4:4:4 and 10-bit encode support
- **NV16:** Requires 4:2:2 encode support (Video Codec SDK 13.0+)
- **P210:** Requires both 4:2:2 and 10-bit encode support (Video Codec SDK 13.0+)

Use `GetEncoderCaps()` to query format support for specific codecs and GPUs.

### 3.1.3. Methods

**Encode(frame)**

Encodes a single frame and returns the compressed bitstream.

See *Encode* for detailed documentation.

**EndEncode()**

Flushes the encoder pipeline and retrieves any buffered frames.

See *EndEncode* for detailed documentation.

**GetEncodeReconfigureParams()**

Retrieves the current encoder reconfiguration parameters including bitrate, rate control mode, frame rate, and VBV buffer settings.

```
params = encoder.GetEncodeReconfigureParams()
```

**Returns:** structEncodeReconfigureParams object with properties:

- **rateControlMode:** Current rate control mode
- **multiPass:** Multi-pass encoding mode
- **averageBitrate:** Average bitrate in bits per second
- **maxBitRate:** Maximum bitrate for VBR mode
- **vbvBufferSize:** VBV (Video Buffering Verifier) buffer size
- **vbvInitialDelay:** VBV initial delay
- **frameRateNum:** Frame rate numerator
- **frameRateDen:** Frame rate denominator

**Reconfigure(params)**

Dynamically reconfigures the encoder parameters during encoding.

See *Reconfigure* for detailed documentation.

## 3.1.4. Encoder Capabilities

**GetEncoderCaps(codec, gpu_id=0)**

Static method to query encoder hardware capabilities for a specific codec and GPU. Returns a dictionary of capability flags.

```
caps = nvc.GetEncoderCaps(codec="hevc", gpu_id=0)
print(f"Supports 4:4:4: {caps['support_yuv444_encode']}")
print(f"Supports 10-bit: {caps['support_10bit_encode']}")
print(f"Max width: {caps['width_max']}")
print(f"Max height: {caps['height_max']}")
```

**Key Capability Flags:**

- **support_yuv444_encode:** YUV 4:4:4 format encoding support
- **support_10bit_encode:** 10-bit encoding support
- **support_yuv422_encode:** YUV 4:2:2 format encoding support (SDK 13.0+)
- **width_max, height_max:** Maximum resolution support
- **width_min, height_min:** Minimum resolution support
- **support_dyn_bitrate_change:** Dynamic bitrate change support
- **support_dyn_res_change:** Dynamic resolution change support
- **support_lossless_encode:** Lossless encoding mode support
- **num_max_bframes:** Maximum number of B-frames supported
- **support_lookahead:** Lookahead support for better rate control
- **support_temporal_aq:** Temporal adaptive quantization support

## 3.1.5. Encode

Encodes a single frame and returns the compressed bitstream.

### 3.1.5.1 Description

The Encode method is the core encoding operation that takes an uncompressed video frame and returns compressed bitstream data. It accepts frames from CPU memory (numpy arrays) or GPU memory (CUDA Array Interface/DLPack objects) and can optionally apply picture flags and SEI messages.

### 3.1.5.2 Syntax

```
bitstream = encoder.Encode(frame)
bitstream = encoder.Encode(frame, pic_flags)
bitstream = encoder.Encode(frame, pic_flags, sei_messages)
```

### 3.1.5.3 Method Signatures

```
Encode(frame) -> bytes
```

```
Encode(frame, pic_flags: int) -> bytes
```

```
Encode(frame, pic_flags: int, sei_messages: list) -> bytes
```

#### 3.1.5.4 Parameters

| Parameter | Type | Description |
|---|---|---|
| frame | numpy.ndarray or GPU buffer | Input frame data (numpy array for CPU, CUDA Array Interface object for GPU) |
| pic_flags | int (optional) | NV_ENC_PIC_FLAGS enumeration value(s) to control encoding behavior |
| sei_messages | list (optional) | List of SEI (Supplemental Enhancement Information) messages to insert |

**Available Picture Flags:**

- **FORCEINTRA:** Force this frame to be encoded as an intra frame
- **FORCEIDR:** Force this frame to be encoded as an IDR (Instantaneous Decoder Refresh) frame
- **OUTPUT_SPSPPS:** Include SPS/PPS/VPS headers with this frame
- **EOS:** Signal end of stream

#### 3.1.5.5 Returns

`bytes` - Encoded bitstream packet(s). May return multiple packets in a single call (e.g., B-frames).

#### 3.1.5.6 See Also

- *Encoder Class*
- *EndEncode Method*

## 3.1.6. CreateEncoder

Function for creating a hardware-accelerated video encoder.

#### 3.1.6.1 Syntax

```
CreateEncoder(
    width: int,
    height: int,
    fmt: str,
    usecpuinputbuffer: bool,
    **kwargs
) -> Encoder
```

#### 3.1.6.2 Parameters

**Required Parameters:**

- `width` (int) - Width of the input frames in pixels.
- `height` (int) - Height of the input frames in pixels.
- `fmt` (str) - Input surface format: `"NV12"`, `"YUV444"`, `"ARGB"`, `"ABGR"`, etc.
- `usecpuinputbuffer` (bool) - If `True`, accepts NumPy arrays (host memory). If `False`, accepts CUDA buffers (device memory).

**Optional Keyword Arguments (kwargs):**

- `cudacontext` (int) - CUDA context handle. Default: 0 (use current context).
- `cudastream` (int) - CUDA stream handle. Default: 0 (use default stream).
- `gpu_id` (int) - GPU device ordinal. Default: 0.
- `codec` (str) - Output codec: `"h264"`, `"hevc"`, or `"av1"`. Default: `"h264"`.
- `preset` (str) - Quality/speed tradeoff: `"p1"` (fastest) to `"p7"` (best quality). Default: `"p4"`.
- `tuning_info` (str) - Optimization target: `"high_quality"`, `"low_latency"`, `"ultra_low_latency"`, `"lossless"`.
- `rc` (str) - Rate control mode: `"cbr"`, `"vbr"`, `"constqp"`.
- `bitrate` (int/str) - Target bitrate in bits per second.
- `maxbitrate` (int/str) - Maximum bitrate (for VBR mode).
- `vbvbufsize` (int/str) - VBV buffer size in bits.
- `fps` (int) - Target frame rate. Default: 30.
- `gop` (int) - GOP (Group of Pictures) length. Default: 30.
- `bf` (int) - Number of B-frames. Default: 0.
- `profile` (str) - Codec profile (e.g., `"main"`, `"high"`).
- `slice_mode` (int) - Slice mode configuration.
- `slice_data` (int) - Slice data configuration.
- `num_unit_in_ticks` (int) - Timing info: number of units in ticks.
- `timescale` (int) - Timing info: timescale value.

#### 3.1.6.3 Description

Creates an Encoder instance for hardware-accelerated video encoding using NVIDIA NVENC. The encoder can compress raw frames into various video codecs (H.264, HEVC, AV1).

The function accepts required positional parameters for frame dimensions and format, followed by optional keyword arguments for encoder configuration. Nested parameters like `slice` and `timinginfo` are automatically flattened.

#### 3.1.6.4 Example

```
import PyNvVideoCodec as nvc

# Basic encoder with CPU input buffers
encoder = nvc.CreateEncoder(
    width=1920,
    height=1080,
    fmt="NV12",
    usecpuinputbuffer=True,
    codec="h264",
    preset="p4",
    bitrate=5000000
)

# Encoder with GPU input buffers and custom settings
encoder = nvc.CreateEncoder(
    width=3840,
    height=2160,
    fmt="NV12",
    usecpuinputbuffer=False,
    codec="hevc",
    preset="p6",
    tuning_info="high_quality",
    rc="vbr",
    bitrate=25000000,
    maxbitrate=30000000,
    fps=60,
    gop=120
)
```

#### 3.1.6.5 Returns

Encoder object configured with the specified parameters.

#### 3.1.6.6 See Also

- *Encoder Class*
- *Encode Method*
- *Reconfigure Method*

## 3.1.7. EndEncode

Flushes the encoder pipeline and retrieves any buffered frames.

#### 3.1.7.1 Description

The EndEncode method signals the end of the encoding session and flushes any frames remaining in the encoder's internal buffer. This method must be called at the end of encoding to ensure all frames are properly encoded and output.

#### 3.1.7.2 Syntax

```
bitstream = encoder.EndEncode()
```

#### 3.1.7.3 Method Signature

```
EndEncode() -> bytes
```

#### 3.1.7.4 Returns

`bytes` - Remaining encoded bitstream packets from the encoder buffer.

#### 3.1.7.5 See Also

- *CreateEncoder Method*
- *Encode Method*

## 3.1.8. Reconfigure

Dynamically reconfigures encoder parameters such as bitrate, rate control mode, and frame rate without recreating the encoder.

#### 3.1.8.1 Description

The `Reconfigure` method allows dynamic adjustment of encoder parameters during an encoding session. This is useful for adaptive bitrate streaming, changing quality targets, or adjusting frame rates based on system conditions.

#### 3.1.8.2 Syntax

```
encoder.Reconfigure(reconfig_params)
```

#### 3.1.8.3 Method Signature

```
Reconfigure(params: structEncodeReconfigureParams) -> None
```

#### 3.1.8.4 Parameters

**params (structEncodeReconfigureParams)**
Reconfiguration parameters object with properties:

- `rateControlMode`: Rate control mode (CBR, VBR, CQP)
- `averageBitrate`: Average bitrate in bits per second
- `maxBitrate`: Maximum bitrate for VBR mode
- `vbvBufferSize`: VBV buffer size in bits
- `vbvInitialDelay`: VBV initial delay in bits
- `frameRateNum`: Frame rate numerator
- `frameRateDen`: Frame rate denominator

#### 3.1.8.5 See Also

- *Encoder Class*
- *Encode Method*

# Chapter 4. Transcoder

## 4.1. Transcoder

### 4.1.1. Transcoder Overview

The `Transcoder` class provides a simple interface for transcoding video streams. It combines decoding, encoding, and muxing operations to convert video files from one format to another while preserving audio streams.

The Transcoder can be configured with various parameters to control its behavior:

| Parameter | Type | Description |
|---|---|---|
| enc_file_path | str | Path to the input container file (source video to transcode) |
| muxed_file_path | str | Path to the output container file after transcoding |
| gpu_id | int | GPU device ID on which to perform decoding and encoding |
| cuda_context | int | CUDA context under which the transcoding operations are performed |
| cuda_stream | int | CUDA stream used by the decoder and encoder |
| **kwargs | dict | Encode configuration settings (codec, bitrate, preset, etc.) |

### 4.1.2. Methods

**segmented_transcode(start, end)**

Transcodes a specific segment of the video defined by start and end timestamps.

See *segmented_transcode* for detailed documentation.

## 4.1.3. segmented_transcode

Transcodes a specific segment of the video defined by start and end timestamps.

#### 4.1.3.1 Description

The `segmented_transcode` method extracts and transcodes a specific time range from the input video. It seeks to the start time, forces an IDR frame at the beginning of the segment for independent playback, re-encodes video frames, and copies corresponding audio packets.

#### 4.1.3.2 Syntax

```
transcoder.segmented_transcode(start, end)
```

#### 4.1.3.3 Method Signature

```
segmented_transcode(start: float, end: float) -> None
```

#### 4.1.3.4 Parameters

| Parameter | Type | Description |
|---|---|---|
| start | float | Start timestamp in seconds (rounded to 2 decimal places) |
| end | float | End timestamp in seconds (rounded to 2 decimal places) |

#### 4.1.3.5 See Also

- *Transcoder Class*

## Copyright