PyNvVideoCodec API

Programming Guide

vPYNVVIDEOCODEC_PG-08085-001_v07" | May 2025

Table of Contents

Chapter 1. Overview

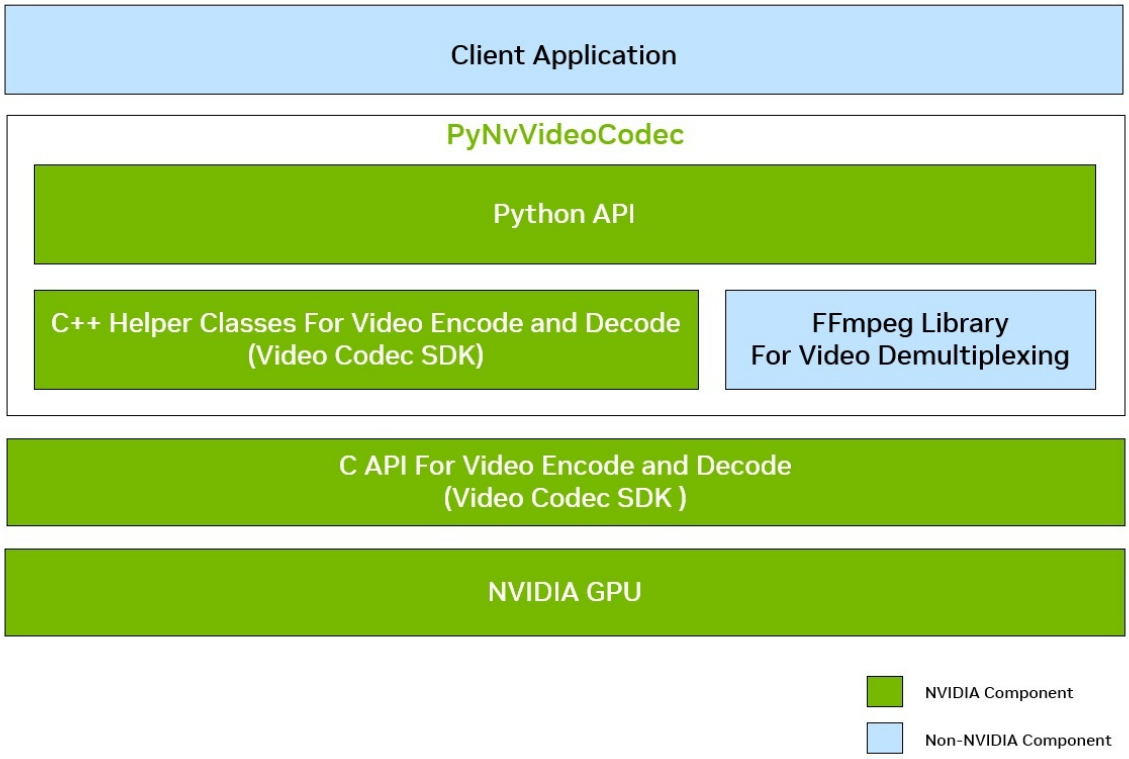
NVIDIA's Video Codec SDK offers hardware-accelerated video encoding and decoding through highly optimized C/C++ APIs. Such encoding and decoding of videos is also useful for a wide range of users, including computer vision experts, researchers and Deep Learning (DL) developers. The objective of PyNvVideoCodec is to provide simple APIs for harnessing such video encoding and decoding capabilities when working with videos in Python.

PyNvVideoCodec is a library that provides Python bindings over C++ APIs for hardware-accelerated video encoding and decoding. Internally, it utilizes core APIs of NVIDIA Video Codec SDK and provides the ease-of-use inherent to Python. It relies on an external FFmpeg library for demuxing media files.

PyNvVideoCodec gives encode and decode performance (FPS) close to Video Codec SDK.

Here is a high level block diagram showing client application, PyNvVideoCodec library and related components.

Figure 1. High Level Architecture Diagram



Chapter 2. Using PyNvVideoCodec API's

The following sections in this chapter explain how to use PyNvVideoCodec APIs for accelerating video decoding and encoding.

All APIs are exposed in python module named `PyNvVideoCodec`.

2.1. Video Demuxing

Demux API

- `CreateDemuxer`

```
CreateDemuxer(filename: str) -> PyNvDemuxer
parameters
    :param _filename: path to media file or encoded bitstream
```

`CreateDemuxer` function accepts files with extension `.mp4`, `.avi`, and `.mkv`.

The `CreateDemuxer` has parameter as follows:

filename

Absolute path to file

Demux API usage

1. Create Demuxer instance as follows. This only argument required is the media file name.

```
import PyNvVideoCodec as nvc
demuxer = nvc.CreateDemuxer(filename=media_file_name)
```

2. `demuxer` object reads media file and splits it into chunks of data (`PacketData`).

   Example below shows how to fetch `PacketData` from `demuxer` object

```
import PyNvVideoCodec as nvc
demuxer = nvc.CreateDemuxer(filename=media_file_name)
for packet in demuxer:
    # process packet
```

3. In addition to file-based demuxing, PyNvVideoCodec also supports buffer-based demuxing which allows processing video data directly from memory buffers.

   This approach is particularly useful for streaming applications or scenarios where video data is already in memory. For more details, see Buffer-Based Demuxing and Decoding.

`PacketData`

This class stores compressed data. It is typically exported by demuxers and then passed as input to decoders. For video, it typically contains one compressed frame.

The class `PacketData` has following attributes:

bsl
Size of the buffer in bytes where the elementary bitstream data is stored.

bsl_data
A pointer to the buffer containing the elementary bitstream data.

dts
The time at which the packet is decompressed.

duration
Duration of this packet in stream's time base.

key
Value of 1 indicates that packet data belongs to key frame.

pos
Byte position in stream.

pts
The time at which the decompressed packet will be presented to the user.

2.2. Buffer-Based Demuxing and Decoding

In addition to file-based demuxing, PyNvVideoCodec supports buffer-based demuxing which allows processing video data directly from memory buffers rather than reading from disk. This approach is particularly useful for streaming applications or scenarios where video data is already in memory.

Buffer-Based Demuxing Overview

Buffer-based demuxing enables applications to:

- Process data that's already in memory without writing to disk first
- Feed video data in chunks to the demuxer via a callback function
- Control the memory flow between application and demuxer
- Implement custom data sources (network streams, encrypted content, etc.)

Applications of Buffer-Based Demuxing

Buffer-based demuxing is particularly useful for:

- Network streaming applications where data arrives in chunks
- Processing encrypted video that needs to be decrypted in memory
- Working with video data from non-file sources (databases, memory-mapped files, etc.)
- Real-time video processing where data is generated on-the-fly
- Multi-source video applications that composite or switch between different streams

Demuxer Callback Function

When creating a demuxer for buffer-based processing, you provide a callback function instead of a filename. This callback function:

- Is called whenever the demuxer needs more data
- Receives a pre-allocated buffer to fill with video data
- Returns the number of bytes copied to the buffer (0 indicates end of stream)

Implementation Examples

1. VideoStreamFeeder Class:

```
class VideoStreamFeeder:
    """
    Class to handle feeding video data in chunks to the demuxer.
    """
    def __init__(self, file_path):
        # Read the entire video file into a memory buffer
        with open(file_path, 'rb') as f:
            self.video_buffer = bytearray(f.read())
        self.current_pos = 0
        self.bytes_remaining = len(self.video_buffer)
        self.chunk_size = 0

    def feed_chunk(self, demuxer_buffer):
        """
        Feed next chunk of video data to demuxer buffer.
        Returns number of bytes copied (0 if no more data)
        """
        buffer_capacity = len(demuxer_buffer)

        if self.bytes_remaining < buffer_capacity:
            self.chunk_size = self.bytes_remaining
        else:
            self.chunk_size = buffer_capacity

        if self.chunk_size == 0:
            return 0

        # Copy data from video buffer to demuxer buffer
        demuxer_buffer[:] = self.video_buffer[
            self.current_pos:self.current_pos + self.chunk_size]

        self.current_pos += self.chunk_size
        self.bytes_remaining -= self.chunk_size
        return self.chunk_size
```

2. Buffer-Based Decoding Pipeline:

```
def demux_from_byte_array(input_file, yuv_file, use_device_memory):
```

```
    # Create the data feeder
    data_feeder = VideoStreamFeeder(input_file)

    # Create buffer-based demuxer with the feed_chunk callback
    buffer_demuxer = nvc.CreateDemuxer(data_feeder.feed_chunk)

    # Create decoder using codec information from demuxer
    buffer_decoder = nvc.CreateDecoder(
        gpuid=0,
        codec=buffer_demuxer.GetNvCodecId(),
        cudacontext=0,
        cudastream=0,
        usedevicememory=use_device_memory
    )

    # Process the packets and frames
    with open(yuv_file, 'wb') as decFile:
        for packet in buffer_demuxer:
            for decoded_frame in buffer_decoder.Decode(packet):
                # Process and save decoded frames
                # ...
```

Best Practices for Buffer-Based Demuxing

- Consider memory usage: Loading large videos entirely into memory may not be efficient; consider streaming or chunking for large files
- Implement proper error handling in your callback function
- Return 0 from the callback when there is no more data to signal end-of-stream
- For streaming scenarios, implement a thread-safe buffer with proper synchronization

2.3. Video Decoding

PyNvVideoCodec provides robust hardware-accelerated video decoding capabilities, leveraging NVIDIA GPUs to efficiently decode various video formats. This section covers the key concepts and API usage for video decoding operations.

Creating a Decoder

The primary method to create a decoder is through the `CreateDecoder` function. This factory function configures and initializes a decoder instance based on your requirements:

```
import PyNvVideoCodec as nvc

# Create CUDA context and stream
device_id = 0
cuda_device = cuda.Device(device_id)
cuda_ctx = cuda_device.retain_primary_context()
cuda_stream = cuda.Stream()

# Create a demuxer for codec detection
nv_dmx = nvc.CreateDemuxer(filename="input.mp4")

# Create a decoder
nv_dec = nvc.CreateDecoder(
    gpuid=0,                        # GPU device ID
    codec=nv_dmx.GetNvCodecId(),    # Get codec from demuxer
```

```
    cudacontext=cuda_ctx.handle,    # CUDA context
    cudastream=cuda_stream.handle, # CUDA stream
    usedevicememory=True,          # Store decoded frames in device memory
    latency=nvc.DisplayDecodeLatencyType.NATIVE  # Latency mode
)
```

Basic Decoding Workflow

A typical video decoding workflow consists of the following steps:

1. Create a demuxer to extract encoded packets from the video container
2. Create a decoder with the appropriate configuration
3. Feed encoded packets to the decoder
4. Retrieve and process decoded frames

```
import PyNvVideoCodec as nvc

# Create demuxer
demuxer = nvc.CreateDemuxer(filename="input.mp4")

# Create decoder
decoder = nvc.CreateDecoder(
    gpuid=0,
    codec=demuxer.GetNvCodecId(),
    usedevicememory=True
)

# Decode frames
for packet in demuxer:
    for frame in decoder.Decode(packet):
        # Process decoded frame
        process_frame(frame)
```

Decoder Parameters

The decoder can be created with various parameters to control its behavior:

Parameter	Description
gpuid	Ordinal of GPU to use
codec	Codec identifier (H.264, HEVC, AV1, etc.)
cudacontext	Optional CUDA context handle
cudastream	Optional CUDA stream handle
usedevicememory	Whether to keep decoded frames in device memory
outputcolortype	The desired color format of the output frames
maxwidth, maxheight	Maximum dimensions for decoded frames
lowlatency	Enable low-latency decoding mode

Color Formats

PyNvVideoCodec supports various color formats for decoded frames. The color format is specified using the `format` parameter when creating a decoder.

Table 1. Supported Color Formats

Format	Description
NV12	Semi-planar YUV format with Y plane followed by interleaved UV plane. This is the most commonly used format for 8-bit content.
P016	16-bit semi-planar YUV format with Y plane followed by interleaved UV plane. Can be used for 10-bit content (6 LSB bits 0) or 12-bit content (4 LSB bits 0).
YUV444	Planar YUV format with separate Y, U, and V planes at full resolution. Used for 8-bit content with no chroma subsampling.
YUV444_16Bit	16-bit planar YUV format with separate Y, U, and V planes at full resolution. Can be used for 10-bit content (6 LSB bits 0) or 12-bit content (4 LSB bits 0) with no chroma subsampling.
NV16	Semi-planar YUV 4:2:2 format with Y plane followed by interleaved UV plane. Used for 8-bit content with horizontal-only chroma subsampling.
P216	16-bit semi-planar YUV 4:2:2 format with Y plane followed by interleaved UV plane. Can be used for 10-bit content (6 LSB bits 0) or 12-bit content (4 LSB bits 0) with horizontal-only chroma subsampling.

Latency Modes

PyNvVideoCodec provides different latency modes for video decoding, which control the timing of when decoded frames are made available to the application. Understanding these modes is crucial for applications that require real-time or low-latency processing.

The `DisplayDecodeLatencyType` enumeration defines three possible latency modes:

- **NATIVE**: For a stream with B-frames, there is at least 1 frame latency between submitting an input packet and getting the decoded frame in display order.
- **LOW**: For All-Intra and IPPP sequences (without B-frames), there is no latency between submitting an input packet and getting the decoded frame in display order. Do not use this flag if the stream contains B-frames. This mode maintains proper display ordering.
- **ZERO**: Enables zero latency for All-Intra / IPPP streams. Do not use this flag if the stream contains B-frames. This mode maintains decode ordering.

Understanding Latency in H.264/HEVC Decoding

In H.264 and HEVC, there is an inherent display latency for video content with frame reordering (typically due to B-frames). Even for All-Intra and IPPP sequences, if `num_reorder_frames` is not explicitly set to 0 in the Video Usability Information (VUI), there can still be display latency. The LOW and ZERO latency modes help eliminate this latency for appropriate content types.

Implementing Low-Latency Decoding

To achieve low-latency decoding, you need to:

1. Set the appropriate `DisplayDecodeLatencyType` when creating the decoder

2. For packets containing exactly one frame or field, set the `ENDOFPICTURE` flag to trigger immediate decode callback

Code Example:

```
import PyNvVideoCodec as nvc

# Create a decoder with low latency mode
nvdec = nvc.CreateDecoder(
    gpuid=0,
    codec=nvc.cudaVideoCodec.H264,
    cudacontext=cuda_ctx.handle,
    cudastream=cuda_stream.handle,
    latency=nvc.DisplayDecodeLatencyType.LOW
)

# When processing packets in low latency mode
for packet in demuxer:
    # If using LOW or ZERO latency mode
    # and packet contains exactly one frame
    if decode_latency == nvc.DisplayDecodeLatencyType.LOW or \
       decode_latency == nvc.DisplayDecodeLatencyType.ZERO:
        # Set flag to trigger decode callback immediately
        # when packet contains exactly one frame
        packet.decode_flag = nvc.VideoPacketFlag.ENDOFPICTURE

    # Decode the packet
    frames = nvdec.Decode(packet)

    for frame in frames:
        # Process frame here
        process_frame(frame)
```

Note: The ENDOFPICTURE flag is only effective for content without B-frames (All-Intra or IPPP sequences). For content with B-frames, some inherent latency will remain due to the nature of bidirectional prediction.

Error Handling

The decoder provides robust error handling mechanisms for dealing with corrupted streams:

```
try:
    for packet in demuxer:
        for frame in decoder.Decode(packet):
            process_frame(frame)
except nvc.PyNvVCExceptionCuda as e:
    print(f"CUDA error: {e}")
except nvc.PyNvVCExceptionDecode as e:
    print(f"Decoding error: {e}")
except nvc.PyNvVCExceptionUnsupported as e:
    print(f"Unsupported feature: {e}")
```

Performance Optimization Tips

- Use device memory (`usedevicememory=True`) to avoid costly host-device transfers
- Reuse decoder instances when processing multiple videos with similar properties
- Provide a CUDA stream to enable parallel processing with other operations
- Choose the appropriate color format for your workflow to minimize conversions

- Consider using the ThreadedDecoder for ML pipelines to hide decoding latency

2.4. SimpleDecoder

SimpleDecoder Overview

The `SimpleDecoder` class provides a high-level, user-friendly interface for video decoding operations. It simplifies video frame access and decoding with a Pythonic interface, handling many of the complex aspects of video processing.

Key features of SimpleDecoder include:

- Random access to frames using Python's indexing syntax
- Support for both individual frames and frame ranges (slices)
- Batch decoding of sequential or arbitrary frames
- Access to comprehensive stream metadata
- Mapping between time and frame indices
- Decoder reuse and reconfiguration

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

Parameter	Description
enc_file_path	Path to the encoded video file
gpu_id	GPU device ID on which to decode (default: 0)
cuda_context	CUDA context under which the source is decoded (default: 0)
cuda_stream	CUDA stream used by the decoder (default: 0)
use_device_memory	If True, decoded frames are stored in GPU memory using CUDeviceptr wrapped in CUDA Array Interface; if False, frames are in host memory (default: True)
max_width	Maximum width that the decoder must support, important for decoder reuse (default: 0, which means no limit)
max_height	Maximum height that the decoder must support, important for decoder reuse (default: 0, which means no limit)
need_scanned_stream_metadata	If True, collects detailed stream metadata by analyzing each packet (runs on a separate thread, processing time depends on stream size) (default: False)
decoder_cache_size	LRU cache size for the number of decoders to cache (default: 4)
output_color_type	Output format for decoded frames: NATIVE (default, returns in format like NV12, YUV444), RGB (interleaved RGB in HWC format), or RGBP (planar RGB in CHW format)

Parameter	Description
bWaitForSessionWarmUp	Whether to wait for decoder session warmup (default: False)

SimpleDecoder API Usage

Here are examples showing how to use SimpleDecoder APIs.

- **Creating a decoder**: Initialize with a video file path and optional parameters for GPU selection, memory management, and output format

```
decoder = SimpleDecoder("video.mp4", use_device_memory=True)
```

- **Accessing individual frames**: Use Python indexing to get frames by position

```
frame = decoder[10]  # Get the 11th frame (zero-based indexing)
```

- **Accessing frame ranges**: Use Python slicing to get multiple frames

```
frames = decoder[10:20]  # Get frames 10 through 19
```

- **Getting sequential batches**: Process frames in batches for efficiency

```
batch = decoder.get_batch_frames(batch_size=10)  # Get next 10 frames
```

- **Getting specific frame batches**: Retrieve arbitrary sets of frames by index

```
frames = decoder.get_batch_frames_by_index([5, 10, 15, 20])  # Get specific frames
```

- **Accessing metadata**: Get information about the video stream

```
metadata = decoder.get_stream_metadata()  # Basic metadata
detailed_metadata = decoder.get_scanned_stream_metadata()  # Detailed metadata (if
 enabled)
```

- **Time-based navigation**: Convert between time and frame indices

```
frame_idx = decoder.get_index_from_time_in_seconds(10.5)  # Get frame at 10.5
 seconds
```

- **Seeking to positions**: Move to specific position in the stream

```
decoder.seek_to_index(100)  # Seek to frame 100
```

- **Decoder reuse**: Reconfigure for a new video source without creating a new decoder

```
decoder.reconfigure_decoder("another_video.mp4")  # Switch to new video
```

- **Advanced initialization**: Configure the decoder with extended parameters for specialized use cases

```
advanced_decoder = SimpleDecoder(
    enc_file_path="input_video.mp4",          # Input filename
   gpu_id=0,                                 # Index of GPU, useful for multi-GPU setups
   use_device_memory=True,                      # Decoded frames reside in device memory
   max_width=1920,                           # Maximum width of buffer for decoder reuse
   max_height=1080,                         # Maximum height of buffer for decoder reuse
    need_scanned_stream_metadata=True,        # Retrieve stream-level metadata
     decoder_cache_size=8,                              # Maximum number of unique
 decoder sessions cached
    output_color_type=nvc.OutputColorType.RGB # Decoded frames available as RGB
 or YUV
)
```

- **Getting video information**: Retrieve video properties like length, resolution, frame rate, and codec

```
# Get total number of frames
total_frames = len(decoder)

# Get basic stream metadata
```

```
metadata = decoder.get_stream_metadata()
print(f"Video dimensions: {metadata.width}x{metadata.height}")
print(f"FPS: {metadata.avg_frame_rate}")
print(f"Codec: {metadata.codec}")
print(f"Duration: {metadata.duration} seconds")
```

- **Sequential frame fetching**: Get frames in sequence using Python indexing

```
# Get the first frame
frame_0 = decoder[0]

# Get a range of frames (frames 0 to 10)
frames_0_10 = decoder[0:10:1]
```

- **Sliced frame fetching**: Get frames at specified intervals using Python's slice notation

```
# Fetch every third frame from index 0 to 9 (frames 0, 3, 6, 9)
sampled_frames = decoder[0:10:3]
```

- **Sequential batch fetching**: Retrieve batches of sequential frames for efficient processing

```
batch_size = 16

# Fetch the first batch of 16 sequential frames (frames 0 to 15)
frame_batch_0_15 = decoder.get_batch_frames(batch_size)

# Fetch the next batch of 16 frames (frames 16 to 31)
frame_batch_16_31 = decoder.get_batch_frames(batch_size)
```

- **Random batch fetching**: Jump to a specific position and retrieve a batch

```
# Seek to Index 50 and get the next 16 frames (frames 50 to 65)
decoder.seek_to_index(50)
frame_batch_50_65 = decoder.get_batch_frames(batch_size)
```

2.5. Decoder Caching

How Decoder Caching Works

The SimpleDecoder class in PyNvVideoCodec manages an internal cache of decoder instances. When a new video is decoded, the class attempts to reuse an existing decoder from cache. A decoder will only be reused if the codec, bit depth, and chroma subsampling of the new video exactly match those of a cached instance.

If no matching decoder is found, a new one is created and added to the cache. If the cache is full, the least recently used (LRU) decoder is evicted to make space for the new instance.

Additional Parameters Affecting Caching

In addition to the basic requirements mentioned above, the ability to reuse a decoder from the cache also depends on the following parameters specified when creating the SimpleDecoder:

- `max_width`: The maximum frame width the decoder must support
- `max_height`: The maximum frame height the decoder must support
- `decoder_cache_size`: The number of decoder instances that can be stored in the cache

For maximum cache reuse, configure these values to accommodate your typical input dimensions without oversizing.

Code Examples

Example 1: Using Decoder Cache

```
import PyNvVideoCodec as nvc

# Create and cache decoder instance
decoder = nvc.SimpleDecoder(enc_file_path="1280x720.mp4",
                            max_width=2048,
                            max_height=2048,
                            decoder_cache_size=6)

# Decode frames from first video
frames = decoder.get_batch_frames(4)

# Reconfigure the same decoder instance for new video
decoder.reconfigure_decoder("1920x1080.mp4")

# Reusing cached decoder instance since the codec, bit depth,
# and chroma subsampling of the new input match those of the cached decoder

# Decode frames from new video
new_frames = decoder.get_batch_frames(4)
```

Example 2: New Decoder Created Due to Resolution Change and Zero Dimension Limits

```
import PyNvVideoCodec as nvc

# Create decoder with max_width and max_height set to 0
decoder = nvc.SimpleDecoder(enc_file_path="1920x1080.mp4")

# Decode frames from first video
frames = decoder.get_batch_frames(4)

# Attempt to reconfigure for a smaller video
decoder.reconfigure_decoder("1280x720.mp4")

# Since max_width and max_height were set to 0, the decoder cannot be reconfigured
# New decoder instance is created

# Decode frames from new video
new_frames = decoder.get_batch_frames(4)
```

Best Practices for Decoder Caching

- **Set appropriate maximum dimensions:** Set `max_width` and `max_height` to the largest resolution you expect to encounter to maximize cache reuse
- **Optimize cache size:** Set `decoder_cache_size` based on how many different types of videos you'll be processing simultaneously
- **Group similar videos:** Process videos with the same codec and similar properties together to improve cache hit rates
- **Monitor performance:** Compare performance with and without caching to determine the optimal strategy for your workflow

Performance Considerations

While decoder caching can significantly improve performance in many scenarios, there are some considerations to keep in mind:

- **Memory usage:** Larger cache sizes will consume more GPU memory
- **Codec compatibility:** Caching is most effective when processing videos with the same codec, bit depth, and chroma subsampling
- **Resolution differences:** If you're processing videos with widely varying resolutions, consider having separate decoders optimized for each resolution range

2.6. ThreadedDecoder

With the increasing demand for real-time and high-performance deep learning (DL) workloads, optimizing video processing pipelines has become crucial. Many inference workloads—such as object detection, action recognition, and video analytics—rely on decoded video frames as input. However, video decoding can introduce latency, potentially becoming a bottleneck in the inference pipeline.

To address this, PyNvVideoCodec provides a ThreadedDecoder feature. This feature enables decoding to run in the background on a dedicated thread, ensuring that a batch of decoded frames is always available for the inference pipeline.

How the Threaded Decoder Works

In a traditional decoding workflow, the inference pipeline must wait for frames to be decoded before processing can begin, resulting in idle GPU cycles and reduced overall performance. The threaded decoder addresses this inefficiency by continuously decoding frames in the background and maintaining a preloaded buffer of ready-to-use frames. This approach effectively hides the decoding latency and ensures that inference becomes the primary bottleneck assuming, as is often the case, that decoding is faster than inference.

Key Benefits

- **Reduced Latency:** The inference process does not have to wait for frame decoding, leading to lower end-to-end processing time.
- **Maximized GPU Utilization:** Ensures that the GPU is consistently engaged in inference without unnecessary stalls due to decoding delays. This helps overlapping SM usage with NVDEC usage thereby increasing utilization of both the engines.
- **Optimized Pipeline Performance:** By keeping a batch of frames ready for processing, the system achieves smoother and more efficient execution.
- **Improved Real-Time Performance:** Particularly useful for real-time applications such as surveillance, autonomous vehicles, and live-streaming analytics.

Threaded Decoder API Usage

The ThreadedDecoder can be easily integrated into existing video processing pipelines. The following code snippet demonstrates how to enable and use the threaded decoder:

```
import PyNvVideoCodec as nvc
```

```
import torch
from torchvision import models

# Initialize the threaded video decoder
decoder = nvc.ThreadedDecoder(
    enc_file_path,
    buffer_size=12,
    cuda_context=0,
    cuda_stream=0,
    use_device_memory=True,
    output_color_type=nvc.OutputColorType.RGBP)

# Loading the pre-trained faster R-CNN model
model = models.fasterrcnn_resnet50_fpn(pretrained=True)
model.to(torch.device('cuda'))
model.eval()

# Main Decoding and Inference loop
while True:

    # Fetch the next batch of frames (pre-fetched by threaded decoder)
    frames = decoder.get_batch_frames(3)

    # Exit the loop if no more frames are available
    if len(frames) == 0:
        break

    src_tensor_list = []

    for frame in frames:
        # Convert PyNvVideoCodec frame to a PyTorch tensor without copying
        # Frame is in planar RGB format: all R pixels, then G, then B
        src_tensor = torch.from_dlpack(frame)

        # Normalize the tensor values to [0, 1] as expected by the model
        src_tensor = src_tensor.float() / 255.0
        src_tensor_list.append(src_tensor)

    # Run inference on batch input
    with torch.no_grad():
        outputs = model(src_tensor_list)
```

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

Parameter	Description
enc_file_path	Path to the encoded video file
buffer_size	Number of frames to prefetch and keep in the buffer
cuda_context	CUDA context handle (default: 0, which uses the primary context)
cuda_stream	CUDA stream handle (default: 0, which creates a new stream)
use_device_memory	Whether to keep decoded frames in GPU memory (default: True)
output_color_type	Format of the decoded frames (e.g., RGB, RGBP, YUV)

Key methods provided by ThreadedDecoder include:

Method	Description
get_batch_frames(batch_size)	Get a batch of prefetched frames
get_stream_metadata()	Get information about the video stream

Method	Description
reconfigure_decoder(new_source)	Switch to a new video source

Performance Considerations

- **Buffer Size:** Adjust the `buffer_size` parameter based on your application's memory constraints and latency requirements
- **Memory Usage:** Using device memory (`use_device_memory=True`) avoids expensive host-device transfers but consumes GPU memory
- **Color Format:** Choose the output color format that best matches your model's input requirements to minimize conversions
- **Thread Synchronization:** The ThreadedDecoder handles thread synchronization internally, so you don't need to worry about race conditions when accessing frames

2.7. Video Encoding

PyNvVideoCodec provides powerful hardware-accelerated video encoding capabilities using NVIDIA GPUs. This section covers the key concepts and API usage for video encoding operations.

Creating an Encoder

The primary method to create an encoder is through the `CreateEncoder` function. This factory function configures and initializes an encoder instance based on your requirements:

```
import PyNvVideoCodec as nvc

# Create an encoder
encoder = nvc.CreateEncoder(
    width=1920,
    height=1080,
    format="NV12",
    usecpuinputbuffer=False,
    **config_params)
```

Encoder Parameters

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

Parameter	Description
gpuid	Ordinal of GPU to use
width	The desired width of the encoded video
height	The desired height of the encoded video
format	Surface format of raw data. See Supported Surface Formats for available options.
usecpuinputbuffer	Value of True indicates that input to encode must be host memory, False indicates device memory

Parameter	Description
**kwargs	Key-value pairs of optional parameters that allow fine-grained control. Please refer to Optional Parameters for more details

Basic Encoding Workflow

A typical video encoding workflow consists of the following steps:

1. Create an encoder with the appropriate configuration
2. Feed raw frames to the encoder
3. Retrieve and process encoded bitstream
4. Flush the encoder when finished

```
import PyNvVideoCodec as nvc
import numpy as np

# Create encoder
encoder = nvc.CreateEncoder(
    width=1920,
    height=1080,
    format="NV12",
    usecpuinputbuffer=True)

frame_size = 1920 * 1080 * 1.5  # Size for NV12 format

# Process input frames
with open("output.h264", "wb") as output_file:
    for i in range(num_frames):
        # Read raw frame data
        chunk = np.fromfile(input_file, np.uint8, count=frame_size)

        if chunk.size == 0:
            break

        # Encode the frame
        bitstream = encoder.Encode(chunk)

        # Write encoded data to file
        output_file.write(bytearray(bitstream))

    # Flush encoder
    bitstream = encoder.EndEncode()
    output_file.write(bytearray(bitstream))
```

Encode API

1. `CreateEncoder`

   This method returns an object of encoder.

   Example below shows how to create encoder object with minimal parameters

   ```
   import PyNvVideoCodec as nvc
   encoder = nvc.CreateEncoder(1920,1080, "NV12", False)
   ```

   The `CreateEncoder` takes following parameters

gpuid
Ordinal of GPU to use

width
The desired width of the encoded video

height
The desired height of the encoded video

format
Surface format of raw data, Can take any of the values from "NV12", "ARBG", "ABGR", "YUV444", "YUV420", "P010" and "YUV444_16bit"

usecpuinputbuffer
Value of 0 indicates that input to encode must be device memory else it must be host memory.

****kwargs**
Key Value pairs of optional parameters that allow fine grained control. Please refer to Optional Parameters for more details.

2. `Encode`

`Encode` method accepts raw data and returns an array of encoded bitstream

Input buffer to `Encode` can be any of as follows

a). **1-D array of bytes**, For e.g. we could read a chunk of bytes from raw YUV and pass it as a parameter as follows

```
import PyNvVideoCodec as nvc
import numpy as np
encoder = nvc.CreateEncoder(
          1920,
          1080,
          "NV12",
          True)
frame_size = 1920 * 1080 * 1.5
chunk = np.fromfile(
        dec_file,
        np.uint8,
        count=frame_size)
if chunk.size != 0:
    bitstream = nvenc.Encode(chunk)  # encode frame one by one
```

b). **Object of any class which implements CUDA Array Interface as follows**

It is important to note that for multi-planar and semi-planar formats such YUV444 or NV12, The Class should have one implementation of CUDA Array Interface per plane

Example below shows how to represent NV12 surface format as class implementing CUDA Array Interface:

```
import PyNvVideoCodec as nvc
import numpy as np
import pycuda.driver as cuda

class AppFrame:
    def __init__(self, width, height, format):
        if format == "NV12":
```

```
            nv12_frame_size = int(width * height * 3 / 2)
            self.gpuAlloc = cuda.mem_alloc(nv12_frame_size)
            self.cai = []
            self.cai.append(AppCAI(
            (height, width, 1),
            (width, 1, 1),
            "|u1", self.gpuAlloc))
            chroma_alloc = int(self.gpuAlloc)
            + width * height
            self.cai.append(AppCAI((int(height / 2),
            int(width / 2), 2),
            (width, 2, 1),
            "|u1",
            chroma_alloc))
            self.frameSize = nv12_frame_size
    def cuda(self):
        return self.cai

encoder = nvc.CreateEncoder(
          1920,
          1080,
         "NV12", False)
input_frame = AppFrame(
          1920,
          1080,
          "NV12")
bitstream = encoder.Encode(input_gpu_frame)
```

> **!** **ATTENTION:** Please note that `AppFrame` implements `cuda` method . `Encode` accepts object of `AppFrame` only if its implements `cuda` method.

c). **NCHW Tensor with batch count as1 (N=1) and channel count as 1 (C=1)**

For a single frame from 1080p YUV, tensor shape shape should be [1,1,1620,1920]

Example below shows how to represent NV12 as NCHW Tensor

```
import PyNvVideoCodec as nvc
import numpy as np
import torch
encoder = nvc.CreateEncoder(1920,1080,
          "NV12", False)
cuda0 = torch.device('cuda:0')
input_tensor = torch.ones(
               [1620, 1920],
               dtype=torch.uint8,
               device=cuda0)
bitstream = encoder.Encode(input_tensor)
```

> **!** **ATTENTION:** Width specified during `CreateEncoder` for NV12 surface format is 1080, but Tensor is created with Width as 1620. This small workaround needed as encode hardware assumes luma and chroma planes are contiguous and Tensor don't work with planar surface formats.

3. `EndEncode`

`EndEncode` method flushes encoder and returns pending bitstream data from encoder queue

Example below shows how to fetch pending bitstream data from encoder queue for 1080p raw YUV after encoding 100 frames

```
import PyNvVideoCodec as nvc
import numpy as np
encoder = nvc.CreateEncoder(
          1920,
          1080, "NV12", True)
frame_size = 1920 * 1080 * 1.5
encoder = nvc.CreateEncoder(
          width,
          height,
          fmt,
          use_cpu_memory,
          **config_params)  # create encoder object
    for i in range(100):
        chunk = np.fromfile(
                dec_file,
                np.uint8,
                count=frame_size)
        if chunk.size != 0:
            bitstream = encoder.Encode(chunk)  # encode frame one by one
        bitstream = encoder.EndEncode()  # flush encoder queue
```

ATTENTION: Call to `EndEncode()` should be done at the last as it signifies that end of input data to encoder

4. `GetEncodeReconfigureParams` and `Reconfigure`

   `Reconfigure` API allows clients to change the encoder initialization parameters 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.

   However, The API currently only supports reconfiguration of parameters listed below:

   - `rateControlMode`.
   - `multiPass`.
   - `averageBitrate`.
   - `vbvBufferSize`.
   - `maxBitRate`.
   - `vbvInitialDelay`.
   - `frameRateNum`.
   - `frameRateDen`.

   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.

Example below shows how to fetch and change reconfigurable parameters:

```
import PyNvVideoCodec as nvc
import numpy as np
encoder = nvc.CreateEncoder(1920,1080, "NV12", True)
t = encoder.GetEncodeReconfigureParams()
t.averageBitrate = int(t.averageBitrate / 2)
t.vbvBufferSize = int(
                  t.averageBitrate * t.frameRateDen
                  / t.frameRateNum)
t.vbvInitialDelay = t.vbvBufferSize
encoder.Reconfigure(t)
```

2.8. Video Encoding Basics

PyNvVideoCodec has been designed for the most simplified possible use of video encoding using appropriate default values and simple functions. However, you can also access the detailed optional parameters and the full flexibility offered by NVIDIA video technology stack through the C++ interface.

If you are familiar with video encoding basic you could directly jump to the video encoding parameters that can be used with video encode API

NVIDIA GPU allows to encode H.264, HEVC, and AV1 content. Depending on your hardware generation, not all Codec will be accessible. Refer to the NVIDIA Hardware Video Encodersection for information about supported Codec for each GPU architecture.

Surface Formats

PyNvVideoCodec supports various input surface formats for encoding. The surface format is specified using the `format` parameter when creating an encoder.

Table 2. Supported Surface Formats

Format	Description
NV12	Semi-Planar YUV [Y plane followed by interleaved UV plane]
YV12	Planar YUV [Y plane followed by V and U planes]
IYUV	Planar YUV [Y plane followed by U and V planes]
YUV444	Planar YUV [Y plane followed by U and V planes]

Format	Description
YUV420_10BIT	10 bit Semi-Planar YUV [Y plane followed by interleaved UV plane]. Each pixel of size 2 bytes. Most Significant 10 bits contain pixel data.
YUV444_10BIT	10 bit Planar YUV444 [Y plane followed by U and V planes]. Each pixel of size 2 bytes. Most Significant 10 bits contain pixel data.
ARGB	8 bit Packed A8R8G8B8. Word-ordered format where a pixel is represented by a 32-bit word with B in the lowest 8 bits, G in the next 8 bits, R in the 8 bits after that and A in the highest 8 bits.
ARGB10	10 bit Packed A2R10G10B10. Word-ordered format where a pixel is represented by a 32-bit word with B in the lowest 10 bits, G in the next 10 bits, R in the 10 bits after that and A in the highest 2 bits.
ABGR	8 bit Packed A8B8G8R8. Word-ordered format where a pixel is represented by a 32-bit word with R in the lowest 8 bits, G in the next 8 bits, B in the 8 bits after that and A in the highest 8 bits.
ABGR10	10 bit Packed A2B10G10R10. Word-ordered format where a pixel is represented by a 32-bit word with R in the lowest 10 bits, G in the next 10 bits, B in the 10 bits after that and A in the highest 2 bits.
NV16	Semi-Planar YUV 422 [Y plane followed by interleaved UV plane]
P210	Semi-Planar 10-bit YUV 422 [Y plane followed by interleaved UV plane]

Notes on Surface Format Usage:

- Both 10-bit and 16-bit input frames result in 10-bit encoding
- The colorspace conversion matrix can be specified using the `colorspace` option during `CreateEncoder`
- Not all formats are supported on all GPU architectures; refer to your GPU's documentation for specific support information

Tuning

The NVIDIA Encoder Interface exposes four different tuning options:

- High quality suited for: - High-quality latency-tolerant transcoding - Video archiving - Encoding for OTT streaming
- Low latency suited for: - Cloud gaming - Streaming - Video conferencing - High bandwidth channel with tolerance for bigger occasional frame sizes
- Ultra-low latency for: - Cloud gaming - Streaming - Video conferencing - In strictly bandwidth-constrained channel
- Lossless for: - Preserving original video footage for later editing - General lossless data archiving (video or non-video)

Presets

For each tuning information, seven presets from P1 (highest performance) to P7 (lowest performance) are available to control performance and quality trade off. Using these presets will automatically set all relevant encoding parameters for the selected tuning information. This is a coarse level of control exposed by the API.

Specific attributes and parameters within the preset can be tuned, if required. This is explained in the next two subsections. For performance references depending on the chosen preset, refer to the NVENC encoding performance in frames/second (fps) table.

Rate Control and Bitrate

NVENC provides control over various parameters related to the rate control algorithm implemented in its firmware, allowing it to adapt the bit rate (or the amount of data necessary to encode your video content per second) depending on your quality, bandwidth, and performance constraints. NVENC supports the following rate control modes:

- Constant bitrate (CBR)
- Variable bitrate (VBR)
- Constant Quantization Parameter (Constant QP)
- Target quality

The bitrate can also be capped to a maximum target value. For more information about rate control, refer to the NVENC Video Encoder API Programming Guide

Building your Optimized Encoder

Refer to the Recommended NVENC Settings section for more information on how to configure NVENC depending on your use case.

2.9. Video Encoding Parameter Details

Table 3. Optional Parameters for `CreateEncoder`

Parameter	Type	Valid Values	Default Parameter	Description
`codec`	String	`h264`, `hevc`, `av1`	`h264`	
`bitrate`	Integer	> 0	10000000U	
`fps`	Integer	> 0	30	Desired Frame Per Second of the video to be encoded, default value is set to 30

Parameter	Type	Valid Values	Default Parameter	Description
initqp	Integer	> 0	unset option	Initial Quantization Parameter (QP)
idrperiod	Integer	> 0	250	Period between Instantaneous Decoder Refresh (IDR) frames
constqp	Integer or list of 3 integers	>=0, <=51		
qmin	Integer or list of 3 integers	>=0, <=51	[30,30,30]	
gop	Integer or list of 3 integers	>0	changes based on other settings	
tuning_info	String	high_quality, low_latency, ultra_low_latency, lossless	high_quality	
preset	String	P1 to P7	P4	
maxbitrate	Integer	>0	10000000U	Maximum bitrate used for Variable BitRate (VBR) encoding, allowing to dynamically adapting bit rate based on video content
vbvinit	Integer	>0	10000000U	
vbvbufsize	Integer	>0	10000000U	Target client Video Buffering Verifier (VBV) buffer size, applicable for vbr.
rc	String	cbr, constqp, vbr	cbr	Type of Rate Control (RC) chosen between Constant BitRate (CBR), Constant QP or Variable BitRate (VBR)
multipass	String	fullres, qres	disabled by default	
bf	Integer	>=0	varies based on tuning_info and preset	Specifies the GOP pattern as follows: bf = 0: I, 1: IPP, 2: IBP, 3: IBBP
max_res	List of 2 integers	>0	4K for H264, 8K for HEVC, AV1	Resolution not greater than maximum

Parameter	Type	Valid Values	Default Parameter	Description
				supported by hardware in order to account for dynamic resolution change. For example: [3840, 2160]
temporalaq	Integer	0 or 1	0	
lookahead	Integer	>0	0 to 255	Number of frames to look ahead.
aq	Integer	0 or 1	0	
ldkfs	Integer	>=0, <255	0	Low Delay Keyframe Scale is useful to avoid channel congestion in case I frame ends up generating high number of bits
colorspace	String	bt601, bt709		Specify this option for ARGB/ ABGR inputs
timingInfo :: num_unit_in_ticks	Integer	>0		Specifies the number of time units of the clock (as defined in Annex E of the ITU-T Specification). HEVC and H264 only
timingInfo :: timescale	Integer	>0		Specifies the frequency of the clock (as defined in Annex E of the ITU-T Specification). HEVC and H264 only
slice::mode	Integer	0 to 3	0	Slice modes for H.264 and HEVC encoding (not available for AV1) which could be 0 (MB based slices), 2 (MB row based slices) or 3 (number of slices)

Parameter	Type	Valid Values	Default Parameter	Description
`slice::data`	Integer	valid range changes based on `slice::mode`	0	Specifies the parameter needed for `sliceMode`. AV1 does not support `slice::data`
`repeatspspps`	Integer	0 or 1	0	Enable writing of `Sequence Parameter Set` (SPS) and `Picture Parameter Set` (PPS) for every IDR frame

2.10. Segment-Based Transcoding

Segment-based transcoding is a critical technique in modern video processing pipelines, particularly in workflows that involve deep learning (DL) and AI model training. This approach focuses on extracting smaller, meaningful segments from long videos, allowing for more targeted and efficient processing.

Traditional transcoding workflows typically process entire videos sequentially, often requiring repeated initialization of decoding and encoding contexts. This introduces significant overhead and slows down processing. In contrast, segment-based transcoding minimizes these inefficiencies by avoiding redundant context creation, resulting in faster performance, better resource utilization, and greater overall efficiency—especially important in AI-driven video analysis.

Challenges of Segment-Based Transcoding with FFmpeg

Although FFmpeg is widely used for video processing, it exhibits significant limitations in segment-based transcoding workflows—particularly when utilizing NVIDIA's NVDEC (decoder) and NVENC (encoder) hardware. Major challenges include:

- **Repeated Context Initialization**: FFmpeg creates a new decoding context and encoding context for each segment, resulting in substantial overhead from repeated memory allocation, and GPU resource setup.
- **Inefficient Hardware Utilization**: Each segment launches new NVDEC and NVENC sessions. This constant setup and teardown reduce GPU utilization and limit overall throughput.
- **Serialization Overhead**: FFmpeg does not support reusing decoder and encoder sessions across segments. Consequently, the pipeline resets frequently, introducing switching delays and serialization bottlenecks.

PyNvVideoCodec addresses these inefficiencies by introducing an optimized approach to segment-based transcoding.

The key optimizations include:

- **Persistent Context Management**: Rather than creating a new decode/encode context for each segment, PyNvVideoCodec maintains a persistent context throughout the transcoding session, significantly reducing overhead.
- **Shared Context Across Segments and Streams**: The same context is reused between segments—eliminating unnecessary reinitialization. This context sharing not only applies within a single bitstream but also across multiple bitstreams, further enhancing performance.
- **Efficient NVDEC and NVENC Utilization**: By keeping GPU resources active and simply switching data buffers, PyNvVideoCodec maximizes throughput and achieves better GPU efficiency compared to traditional FFmpeg-based methods.

Performance Improvement Using Segment-Based Transcoding

Segment-based transcoding with PyNvVideoCodec delivers substantial performance improvements over traditional FFmpeg-based methods. When extracting and encoding segments individually using FFmpeg, performance is significantly hindered by repeated NVDEC and NVENC initialization overheads. PyNvVideoCodec eliminates these inefficiencies, resulting in more than a 2x performance boost.

Example: Using PyNvVideoCodec for Segment-Based Transcoding

```
import PyNvVideoCodec as nvc

# Define transcoding quality settings
config = {
    "preset": "P4",
    "codec": "h264",
    "tuning_info": "high_quality"
}

# Initialize transcoder with input/output paths
transcoder = nvc.Transcoder("input.mp4", "output_muxed.mp4", 0, 0, 0, **config)

# Transcode 3 segments of 2 seconds each
transcoder.segmented_transcode(0.0, 2.0)
transcoder.segmented_transcode(2.0, 4.0)
transcoder.segmented_transcode(4.0, 6.0)
```

The PyNvVideoCodec Transcoder class provides a streamlined API for segment-based transcoding:

- `Transcoder(input_file, output_file, gpu_id, cuda_context, cuda_stream, **config)`: Creates a new transcoder instance with specified input, output, and configuration
- `segmented_transcode(start_time, end_time)`: Transcodes a specific segment defined by start and end times (in seconds)

The Transcoder maintains internal state between segment operations, allowing for efficient processing of multiple segments without reinitializing hardware resources.

Best Practices for Segment-Based Transcoding

- **Segment Length**: For optimal performance, keep individual segments longer than 1-2 seconds to amortize any remaining overhead
- **Similar Format Streams**: The greatest performance benefits come when transcoding segments with similar codec properties
- **Context Reuse**: Reuse the same Transcoder instance for all segments rather than creating new instances
- **Memory Management**: For very long processing sessions, consider monitoring GPU memory usage
- **Order of Segments**: Process segments in temporal order when possible for most efficient seek operations

Summary

In summary, segment-based transcoding using PyNvVideoCodec delivers a high-performance alternative to conventional FFmpeg workflows by reducing redundant operations, improving GPU resource utilization, and eliminating repeated context initialization. These enhancements make it exceptionally well-suited for video processing applications requiring low latency and high throughput—such as AI model training, content curation, and media analytics.

2.11. SEI Message Encoding and Decoding

PyNvVideoCodec provides support for Supplemental Enhancement Information (SEI) messages in video streams. SEI messages are used to carry additional information that is not essential for decoding but can be useful for various applications like HDR metadata, time code information, or custom user data.

SEI Messages Overview

SEI messages are a mechanism defined in video coding standards (H.264/AVC, HEVC, AV1) that allow embedding additional data within a video bitstream. They can be used for various purposes:

- Carrying HDR metadata (mastering display color volume, content light level)
- Including time codes or frame information
- Adding alternative transfer characteristics
- Embedding custom user data for application-specific purposes

PyNvVideoCodec allows both:

- Inserting SEI messages during encoding
- Extracting SEI messages during decoding

Encoding with SEI Messages

The `EncodeSEIMsgInsertion.py` sample demonstrates how to insert SEI messages into a video bitstream during encoding. This can be useful for embedding metadata or custom information that should travel with the video.

Sample Usage:

```
python EncodeSEIMsgInsertion.py -i input.yuv -o output.h264 -s 1920x1080 -if NV12 -c h264
```

Key Components:

1. Creating SEI Messages:

```
# Define sample SEI messages (arrays of bytes)
SEI_MESSAGE_1 = [0xdc, 0x45, 0xe9, 0xbd, 0xe6, 0xd9, 0x48, 0xb7, 0x96, 0x2c, 0xd8, 0x20, 0xd9, 0x23, 0xee, 0xef]
SEI_MESSAGE_2 = [0x12, 0x67, 0x56, 0xda, 0xef, 0x99, 0x00, 0xbb, 0x6a, 0xc4, 0xd8, 0x10, 0xf9, 0xe3, 0x3e, 0x8f]

# Determine SEI type based on codec
if config_params["codec"] in ["hevc", "h264"]:
    sei_info = {"sei_type": 5}  # User data unregistered
elif config_params["codec"] == "av1":
    sei_info = {"sei_type": 6}  # AV1 equivalent
else:
    raise ValueError(f"Unsupported codec: {config_params['codec']}")

# Create SEI messages list
sei_messages = [(sei_info, SEI_MESSAGE_1), (sei_info, SEI_MESSAGE_2)]
```

2. Encoding with SEI Messages:

```
# Create encoder
nvenc = nvc.CreateEncoder(width, height, fmt, False, **config_params)

# Process frames with SEI messages
for input_gpu_frame in FetchGPUFrame(...):
    # Pass the SEI messages to the encoder with the current frame
    bitstream = nvenc.Encode(input_gpu_frame, 0, sei_messages)
    encFile.write(bytearray(bitstream))

# Flush encoder queue
bitstream = nvenc.EndEncode()
encFile.write(bytearray(bitstream))
```

SEI Message Parameters:

The SEI message consists of:

- `sei_info`: A dictionary containing metadata about the SEI message, including:
  - `sei_type`: The type of SEI message (varies by codec)
- SEI payload: An array of bytes containing the actual SEI data

Decoding and Extracting SEI Messages

The `DecodeSEIMsgExtraction.py` sample shows how to extract and process SEI messages during video decoding. This allows applications to access metadata embedded in the video stream.

Sample Usage:

```
python DecodeSEIMsgExtraction.py -i input.h264 -o output.yuv -f sei_messages.bin -d 1
```

Key Components:

1. SEI Message Structures:

The sample defines C-style structures using ctypes to parse different types of SEI messages:

```
class TIMECODE(ctypes.Structure):
    """Structure for time code information."""
    _fields_ = [
        ("time_code_set", TIMECODESET * MAX_CLOCK_TS),
        ("num_clock_ts", ctypes.c_uint8),
    ]

class SEICONTENTLIGHTLEVELINFO(ctypes.Structure):
    """Structure for content light level information."""
    _fields_ = [
        ("max_content_light_level", ctypes.c_uint16),
        ("max_pic_average_light_level", ctypes.c_uint16),
        ("reserved", ctypes.c_uint32),
    ]

class SEIMASTERINGDISPLAYINFO(ctypes.Structure):
    """Structure for mastering display information."""
    _fields_ = [
        ("display_primaries_x", ctypes.c_uint16 * 3),
        ("display_primaries_y", ctypes.c_uint16 * 3),
        ("white_point_x", ctypes.c_uint16),
        ("white_point_y", ctypes.c_uint16),
        ("max_display_mastering_luminance", ctypes.c_uint32),
        ("min_display_mastering_luminance", ctypes.c_uint32),
    ]
```

2. Extracting and Processing SEI Messages:

```
# Process decoded frames
for packet in nv_dmx:
    for decoded_frame in nvdec.Decode(packet):
        # ... frame processing ...

        # Extract SEI messages
        seiMessage = decoded_frame.getSEIMessage()
        if seiMessage:
            for sei_info, sei_message in seiMessage:
                sei_type = sei_info["sei_type"]
                sei_uncompressed = sei_info["sei_uncompressed"]

                if sei_uncompressed == 1:
                    buffer = (ctypes.c_ubyte * len(sei_message))(*sei_message)
                    sei_struct = None

                    # Handle different SEI message types
                if sei_type in (nvc.SEI_TYPE.TIME_CODE_H264, nvc.SEI_TYPE.TIME_CODE):
                        sei_struct = ctypes.cast(
```

```
                            buffer,
                        ctypes.POINTER(TIMECODEMPEG2 if codec == nvc.cudaVideoCodec.MPEG2
else TIMECODE)
                        ).contents
                    elif sei_type == nvc.SEI_TYPE.MASTERING_DISPLAY_COLOR_VOLUME:
                                                sei_struct  =  ctypes.cast(buffer,
ctypes.POINTER(SEIMASTERINGDISPLAYINFO)).contents
                    elif sei_type == nvc.SEI_TYPE.CONTENT_LIGHT_LEVEL_INFO:
                                                sei_struct  =  ctypes.cast(buffer,
ctypes.POINTER(SEICONTENTLIGHTLEVELINFO)).contents
                elif sei_type == nvc.SEI_TYPE.ALTERNATIVE_TRANSFER_CHARACTERISTICS:
                                                sei_struct  =  ctypes.cast(buffer,
ctypes.POINTER(SEIALTERNATIVETRANSFERCHARACTERISTICS)).contents

                    if sei_struct:
                        print(sei_struct)

                # Store raw SEI message data
                file_message.write(bytearray(sei_message))

            # Also save in pickle format for later analysis
            pickle.dump(seiMessage, file_type_message)
```

Supported SEI Message Types

PyNvVideoCodec supports various SEI message types through the `SEI_TYPE` enumeration:

- TIME_CODE: Timing information
- TIME_CODE_H264: H.264-specific timing information
- MASTERING_DISPLAY_COLOR_VOLUME: HDR mastering display information
- CONTENT_LIGHT_LEVEL_INFO: Content light level information for HDR
- ALTERNATIVE_TRANSFER_CHARACTERISTICS: Indicates alternative color transfer functions
- User Data: Various formats for custom application data

The SEI type varies by video codec (H.264, HEVC, AV1) and must be selected appropriately.

Applications of SEI Messages

SEI messages are particularly useful for:

- HDR video workflows (carrying mastering display and content light level information)
- Professional video production (embedding timecode and frame information)
- Custom application data transport (embedding metadata that stays with the video)
- Digital rights management (embedding ownership or licensing information)
- Analytics workflows (embedding processing metadata or detection results)

Best Practices for SEI Messages

- Keep SEI messages compact to minimize overhead in the bitstream
- For custom data, consider using the user data unregistered SEI type
- Ensure SEI message formats are consistent between encoder and decoder

- Be aware that some players may ignore SEI messages
- For HDR content, follow standard formats for mastering display and content light level information
- When working with multiple frames, consider which frames should carry SEI messages (typically IDR frames)

2.12. Interoperability with Deep Learning Frameworks

PyNvVideoCodec provides efficient interoperability with popular deep learning frameworks through DLPack, the open-source memory tensor structure for sharing tensors across frameworks. This allows video frames decoded by PyNvVideoCodec to be directly passed to frameworks like PyTorch, TensorFlow, and others without expensive CPU-GPU memory transfers.

DLPack Overview

DLPack is a standardized memory tensor structure that enables efficient sharing of tensor data between different frameworks with zero-copy. It serves as a common exchange format that allows deep learning libraries to pass tensors to each other without expensive data copies or CPU round-trips.

The key benefits of DLPack include:

- Zero-copy tensor sharing between different libraries
- Standardized memory management protocol
- Support for different device types (CPU, CUDA, etc.)
- Common representation for tensor metadata (shape, strides, data type)
- Proper handling of CUDA stream synchronization

PyNvVideoCodec DLPack Implementation

PyNvVideoCodec implements the Python DLPack protocol through `__dlpack__()` and `__dlpack_device__()` methods on decoded frames. This allows seamless integration with any framework that supports the DLPack protocol.

When a frame is decoded in GPU memory (`use_device_memory=True`), the frame object can be directly converted to a framework-specific tensor using that framework's DLPack import function without any data copying.

The implementation handles important aspects:

- **Memory ownership**: The PyNvVideoCodec frame retains ownership of the underlying memory until the tensor using it is destroyed

- **Stream synchronization**: Proper CUDA stream synchronization is maintained between producer (PyNvVideoCodec) and consumer (e.g., PyTorch)
- **Tensor metadata**: Shape, strides, and data type information are correctly propagated to the DLPack tensor

Integration with PyTorch

PyTorch provides the `torch.from_dlpack()` function to import DLPack tensors directly. This enables zero-copy conversion from PyNvVideoCodec frames to PyTorch tensors:

```
import torch
import PyNvVideoCodec as nvc

# Create decoder with GPU output
decoder = nvc.SimpleDecoder("video.mp4", use_device_memory=True)

# Get a frame (GPU memory)
frame = decoder[0]

# Convert to PyTorch tensor with zero-copy
tensor = torch.from_dlpack(frame)

# tensor is now a regular PyTorch tensor backed by the same GPU memory
# No data copying has occurred - tensor and frame share the same memory
print(f"Tensor shape: {tensor.shape}")
print(f"Tensor device: {tensor.device}")
```

The tensor format follows the video pixel format. For example, with RGB data, the tensor will have dimensions (height, width, 3) for interleaved format or (3, height, width) for planar format.

Batch Processing for Deep Learning

When processing multiple frames for deep learning inference, you can create a batch of tensors and stack them:

```
# Get multiple frames
frames = decoder.get_batch_frames(batch_size)

# Convert all frames to tensors (all zero-copy)
tensors = [torch.from_dlpack(frame) for frame in frames]

# Stack into a batch tensor
batch = torch.stack(tensors)

# Now batch has shape [batch_size, channels, height, width]
# Run inference
with torch.no_grad():
    predictions = model(batch)
```

Integration with Other Frameworks

PyNvVideoCodec's DLPack support works with any framework that supports importing DLPack tensors:

TensorFlow:

```
import tensorflow as tf
```

```
# Convert decoded frame to TensorFlow tensor
tf_tensor = tf.experimental.dlpack.from_dlpack(frame)
```

CuPy:

```
import cupy as cp

# Convert decoded frame to CuPy array
cupy_array = cp.from_dlpack(frame)
```

NumPy (with copy):

```
import numpy as np
import torch

# First convert to PyTorch (zero-copy)
torch_tensor = torch.from_dlpack(frame)

# Then convert to NumPy (will copy from GPU to CPU)
numpy_array = torch_tensor.cpu().numpy()
```

Best Practices for DLPack Interoperability

- Always use `use_device_memory=True` when decoding frames intended for deep learning to enable zero-copy transfer
- Keep the original frame objects alive while using the converted tensors to prevent memory deallocation
- Be aware of tensor memory layout differences between frameworks (channel-first vs. channel-last)
- Consider using `ThreadedDecoder` for better performance in deep learning pipelines
- Match the decoder's output color format to what your model expects (RGB, YUV, etc.)
- For optimal performance, keep processing in GPU memory throughout the entire pipeline
- When processing many videos, consider reusing decoders with `reconfigure_decoder()` rather than creating new instances

Chapter 3. PyNvVideoCodec Performance

PyNvVideoCodec offers video encode and decode performance close to Video Codec SDK. This chapter outlines the performance capabilities enabled by unique APIs and features of PyNvVideoCodec.

Note: The benchmarks presented in this chapter use the BtBN FFmpeg build for comparison purposes.

- Frame Retrieval - Performance of different frame retrieval patterns
- Decoder Reuse - Performance benefits of reusing decoder instances
- Segmented Transcoding - Performance of segment based transcoding

3.1. Frame Retrieval

Performance benchmarks for different frame retrieval patterns using PyNvVideoCodec decoder.

Environment:

- GPU: 1 x L40G (3 NVDECs)
- CPU: AMD EPYC 7313P 16-Core Processor, 2 threads per core
- OS: Ubuntu 22.04

Methodology:

- Script to execute benchmark: `frame_sampling_benchmark.py`
- Dataset generated using FFmpeg with the following default parameters:
  - Resolution: 1920x1080
  - GOP: 30 & 250
  - Duration: 30 seconds
  - Frame Rate: 30
- Multithreaded implementation to fully utilize NVDECs (multiple Python threads)
- Each Python thread independently decodes the same video & reports the FPS

Benchmarks:

- Sequential decode (first 100 frames)

  Decodes frames in sequential order from the start of the video. This approach retrieves a specified number of consecutive frames (e.g., first 100 frames).

Table 4. Sequential Decode Performance

Video Config	Num Threads	FPS
1920x1080 250gop 30s	1	867.8
1920x1080 250gop 30s	3	2556.2
1920x1080 30gop 30s	1	863.3
1920x1080 30gop 30s	3	2543.7

- Random sampling (30 frames)

  Randomly selects frames from across the entire video duration. This method is useful for obtaining a representative sample of frames throughout the video.

Table 5. Random Sampling Performance

Video Config	Num Threads	FPS	Efficiency
1920x1080 250gop 30s	1	38.4	1.05x
1920x1080 250gop 30s	3	109.8	1.01x
1920x1080 30gop 30s	1	60.4	1.62x
1920x1080 30gop 30s	3	172.6	1.61x

- Uniform sampling (30 frames)

  Evenly distributes frame sampling across the entire video duration. For example, when sampling 30 frames from a 30-second video, it fetches one frame every second.

Table 6. Uniform Sampling Performance

Video Config	Num Threads	FPS	Efficiency
1920x1080 250gop 30s	1	38.7	1.05x
1920x1080 250gop 30s	3	114	1.05x
1920x1080 30gop 30s	1	52.9	1.44x
1920x1080 30gop 30s	3	154.5	1.42x

Note on Efficiency: Efficiency represents the performance comparison between two approaches:

1. Direct sampling: Decoding specific frames directly using seek operations
2. Sequential decode + sampling: Decoding all frames sequentially and then sampling the required frames

The efficiency value shows how much faster direct sampling is compared to sequential decoding with sampling. Higher efficiency values indicate better performance of the direct sampling approach.

Key Observations:

- GOP size has significant impact on frame retrieval performance:
  - For random sampling, smaller GOP size (30) increases performance by 57% as compared to bigger GOP size(250)
  - For uniform sampling, smaller GOP size (30) increases performance by 37% as compared to bigger GOP size(250)
  - Sequential decoding performance is largely unaffected by GOP size
- Multi-threading scales efficiently across all sampling methods:
  - Sequential decoding shows near-linear scaling from 1 to 3 threads (approximately 2.95x)
  - Random and uniform sampling show good scaling (approximately 2.85x) with 3 threads
- Efficiency comparison between frame retrieval methods:
  - Direct random sampling with 30 GOP shows the highest efficiency gain (1.62x), making it the most efficient for sparse frame access
  - Uniform sampling with 30 GOP shows good efficiency (1.44x)
  - Both sampling methods with 250 GOP show small efficiency advantage (1.05x)

3.2. Decoder Reuse

Performance benefits of reusing decoder instances when processing multiple videos.

Environment:

- GPU: 1 x L40G (3 NVDECs)
- CPU: AMD EPYC 7313P 16-Core Processor, 2 threads per core
- OS: Ubuntu 22.04

Methodology:

- Script to execute benchmark: `cached_decoder_benchmark.py`
- Dataset generated using FFmpeg with the following parameters:
  - Resolutions: 360p, 480p, 720p, 1080p, 4k
  - Frame Rate: 30 fps
  - GOP Size: 60
  - Duration: 2 seconds
  - Pattern: mandelbrot
- 5 videos created using FFmpeg (1 video per resolution)

- Each video was reused 50 times to create enough decoding workload to fully saturate all available NVDEC hardware instances.
- Videos are distributed across multiple decoder threads
- Example configuration: In a 20-clip/4-thread setup, each thread processes 5 videos

Decoder Types:

- **Simple decoder:**
  - Creates a new decoder instance for each video clip
  - For example, if a thread has to decode 5 videos, a total of 5 decoder instances will be created
- **Cached decoder:**
  - Creates a single decoder instance per thread
  - Reuses the same decoder for subsequent clips through reconfiguration
  - Implementation follows the principles outlined in Decoder Caching
  - For example, for 5 videos per thread, only one decoder instance is created and reused

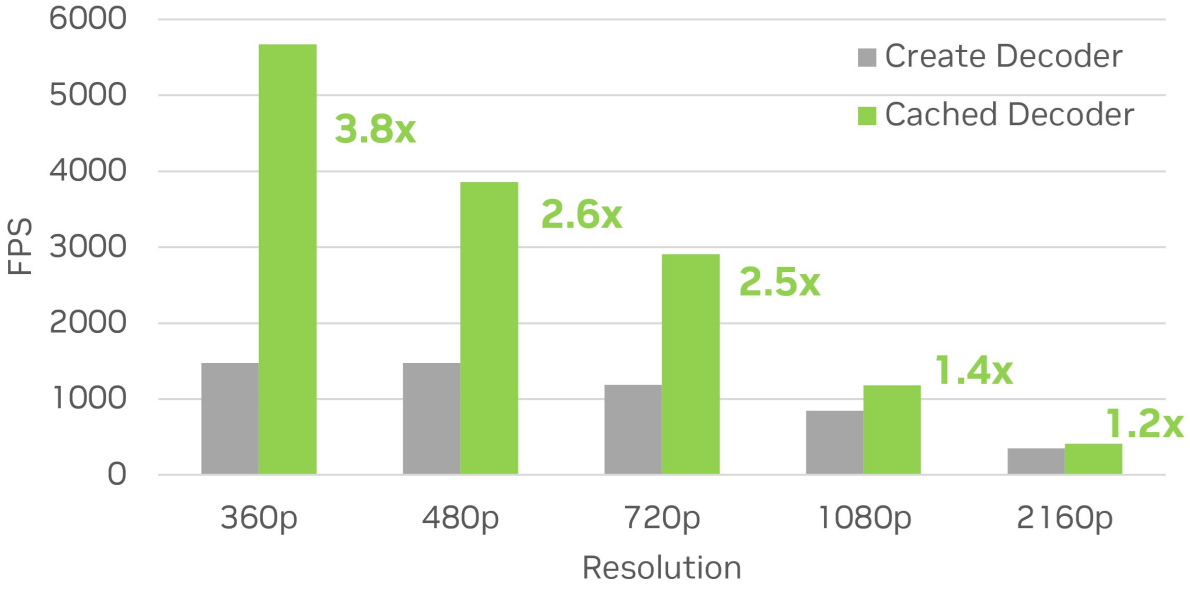
Benchmarks:

Table 7. Decoder Reuse Performance Comparison

Resolution	Decoder Type	FPS
360p	Simple	1475
360p	Cached	5672
480p	Simple	1478
480p	Cached	3855
720p	Simple	1186
720p	Cached	2906
1080p	Simple	849
1080p	Cached	1184
4k	Simple	351
4k	Cached	410

Figure 2. Performance Comparison: Simple vs. Cached Decoders

Bar chart comparing performance of simple decoder creation vs. cached decoder approach across resolutions, showing speedup factors of 3.8x (360p), 2.6x (480p), 2.5x (720p), 1.4x (1080p), and 1.2x (2160p)



Key Observations:

- Cached decoders consistently outperform simple decoders across all resolutions
- Performance improvement is most significant for lower resolutions (360p: 3.8x faster, 480p: 2.6x faster)
- Even at higher resolutions, cached decoders show measurable improvement (4K: 1.2x faster)
- The performance benefit comes from eliminating decoder initialization overhead, which is more significant when processing multiple short videos

3.3. Segmented Transcoding

Performance comparison of PyNvVideoCodec's segmented transcoding approach against traditional FFmpeg-based methods.

Environment:

- GPU: 1 x L40G (3 NVDECs)
- CPU: AMD EPYC 7313P 16-Core Processor, 2 threads per core
- OS: Ubuntu 22.04

Methodology:

- Script to execute benchmark: `segmented_transcode_benchmark.py`
- Dataset details:

  - Total clips: 42
  - Resolutions: 720p (20 clips) and 640p (22 clips)
  - Codec: H.264
  - Input FPS: 29.97
  - Input GOP: 250
  - Total frames: 318,392
  - Transcoded frames: 105,167
- Transcoding parameters:
  - Output FPS: 30
  - Output B Frames: 0
  - Output Preset: P1
- Benchmarks examine performance of different transcoding methods

Transcoding Methods:

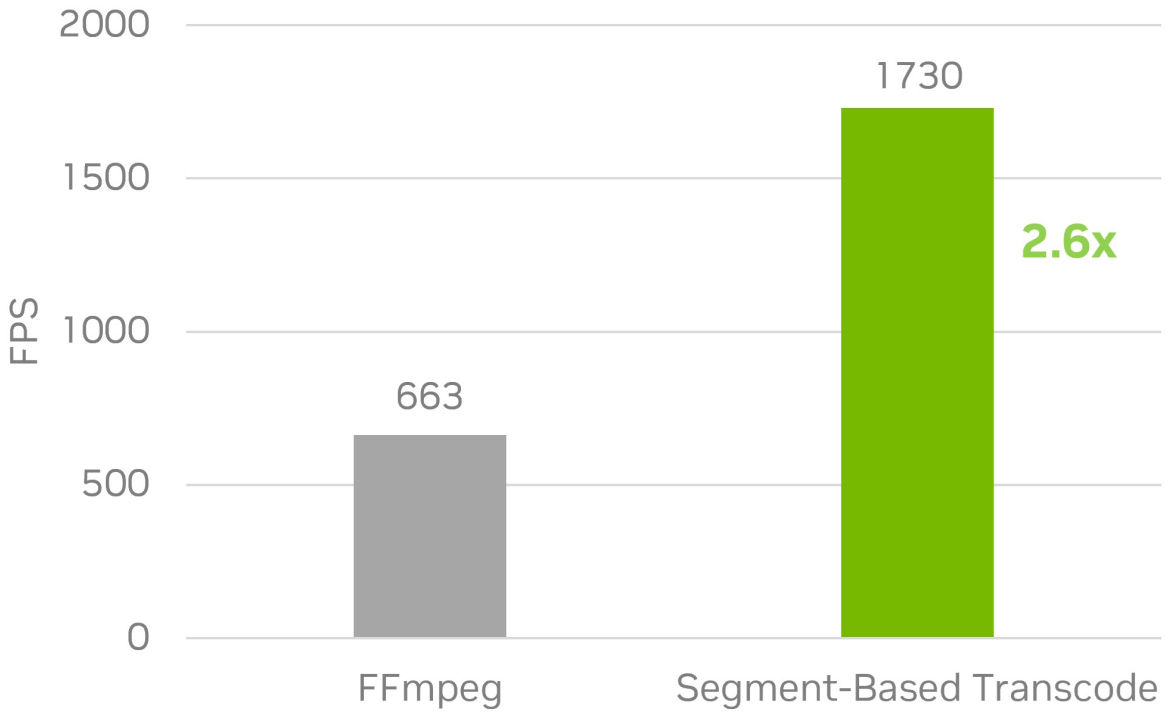
- **PyNVC transcoding**: Uses PyNvVideoCodec with persistent context for segmented transcoding
- **FFmpeg without map**: Uses HW accelerated FFmpeg with simple re-encoding, no mapping or container preservation

Table 8. Segmented Transcoding Performance - H.264 1080p

Method	FPS
PyNVC transcoding	1730
FFmpeg without map	663

Figure 3. Performance Comparison: FFmpeg vs. PyNvVideoCodec Segment-Based Transcoding

Bar chart comparing transcoding performance between standard FFmpeg approach and PyNvVideoCodec's segment-based transcoding for H.264 1080p content, showing a 2.6x performance improvement



Key Observations:

- PyNVC transcoding significantly outperforms FFmpeg's standard transcoding method
- For 1080p content, PyNVC transcoding (1730 FPS) is approximately 2.6x faster than FFmpeg without map (663 FPS)
- The performance advantage comes from persistent context management, avoiding repeated decoder and encoder initialization
- This performance gain is particularly valuable for workflows that process multiple video segments, such as AI training datasets

Chapter 4. Debugging and Logging

Logging Overview

PyNvVideoCodec provides a logging system that helps diagnose issues and understand the library's behavior. The logging system is primarily based on FFmpeg's built-in logging capabilities, which can be controlled using environment variables.

Setting Log Levels

The logging level can be controlled by setting the `LOGGER_LEVEL` environment variable. When set, this environment variable controls the verbosity of FFmpeg logs used by PyNvVideoCodec.

Available log levels (from most verbose to least verbose):

- **TRACE**: Most detailed information (maps to FFmpeg's `AV_LOG_VERBOSE`)
- **DEBUG**: Debugging information (maps to FFmpeg's `AV_LOG_DEBUG`)
- **INFO**: General information messages (maps to FFmpeg's `AV_LOG_INFO`)
- **WARN**: Warning messages (maps to FFmpeg's `AV_LOG_WARNING`)
- **ERROR**: Error messages (maps to FFmpeg's `AV_LOG_ERROR`)
- **FATAL**: Critical error messages (maps to FFmpeg's `AV_LOG_FATAL`)

If the `LOGGER_LEVEL` environment variable is not set, logging defaults to `AV_LOG_QUIET`, which suppresses most messages.

Example Usage

Linux/macOS:

```
# Set log level to DEBUG
export LOGGER_LEVEL=DEBUG

# Run your Python script
python your_script.py
```

Windows (Command Prompt):

```
:: Set log level to DEBUG
set LOGGER_LEVEL=DEBUG

:: Run your Python script
```

```
python your_script.py
```

Windows (PowerShell):

```
# Set log level to DEBUG
$env:LOGGER_LEVEL="DEBUG"

# Run your Python script
python your_script.py
```

Setting in Python code (before importing PyNvVideoCodec):

```
import os
os.environ["LOGGER_LEVEL"] = "DEBUG"

# Now import PyNvVideoCodec
import PyNvVideoCodec as nvc
```

Debugging Recommendations

- **Start with INFO level**: For general troubleshooting, start with `LOGGER_LEVEL=INFO`
- **Use DEBUG for details**: If you need more detailed information about what's happening inside the library, use `LOGGER_LEVEL=DEBUG`
- **TRACE for comprehensive logs**: `LOGGER_LEVEL=TRACE` provides the most detailed logging, but can generate large amounts of output
- **Capture logs to file**: When debugging complex issues, redirect the output to a file for easier analysis:
  ```
  python your_script.py > debug_log.txt 2>&1
  ```
- **Disable logs in production**: For production code, either do not set the environment variable or explicitly set `LOGGER_LEVEL=ERROR` to minimize log output and improve performance

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