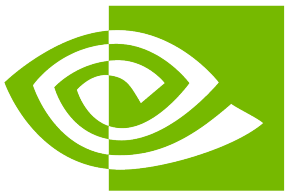

NVIDIA PyNvVideoCodec API Programming Guide

Release 2.2

NVIDIA Corporation

Jul 29, 2026

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 and muxing 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.

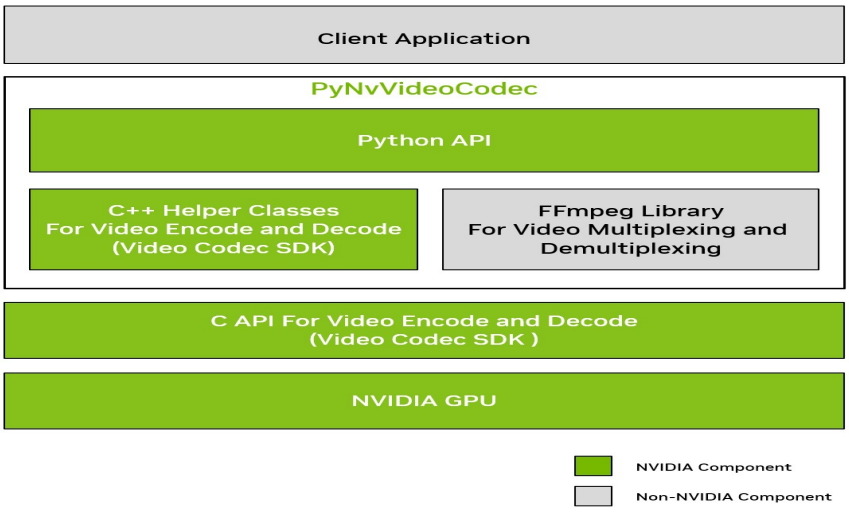

Figure 1.1: High Level Architecture Diagram

Chapter 2. Using PyNvVideoCodec APIs

This chapter explains how to use the PyNvVideoCodec APIs for video decode, encode, and transcode workflows. The chapter also covers how PyNvVideoCodec can exchange video data with popular deep learning frameworks, enabling smooth integration of PyNvVideoCodec into AI and computer-vision pipelines.

2.1. What You Will Learn

This chapter covers the following workflows:

- **Video Decoding**: Learn to use various decoder interfaces (SimpleDecoder, ThreadedDecoder, Core Decoder) for different use cases, from simple frame sampling to high-throughput pipelines.
- **Video Encoding**: Understand encoding workflows, parameter configuration, runtime reconfiguration, and SEI message insertion.
- **Video Transcoding**: Implement complete file transcoding and segment-based operations for adaptive streaming.
- **Interoperability**: Integrate PyNvVideoCodec with PyTorch, TensorFlow, and other deep learning frameworks using efficient zero-copy data exchange.

2.2. Chapter Organization

For each workflow, this chapter:

- Explains the code flow and which APIs to use
- Describes important parameters and enumerations
- Starts with basic use cases, then covers advanced concepts
- Provides practical code examples from sample applications
- Highlights real-world use cases and best practices

2.3. Prerequisites

Before working through this chapter, ensure you have:

- Installed PyNvVideoCodec and its dependencies

- An NVIDIA GPU with hardware video codec support
- Basic familiarity with Python and video concepts (codecs, containers, frame rates)

2.4. Video Demuxing

Extract encoded video packets from container formats using PyNvVideoCodec's demuxing capabilities.

In this section, we'll learn how to extract encoded video packets from container formats like MP4, MKV, and AVI using PyNvVideoCodec's demuxing APIs. Demuxing is the first step when working with the low-level decoder APIs.

2.4.1. What is Demuxing?

Demuxing (demultiplexing) is the process of extracting encoded video packets from container formats. A container format (like MP4 or MKV) wraps the actual video bitstream along with metadata, audio streams, and other data. The demuxer parses this container and provides individual encoded video packets that can be fed to a decoder.

2.4.2. When Do You Need Demuxing?

Demuxing is required when using the low-level `CreateDecoder` API. If you're using `SimpleDecoder` or `ThreadedDecoder`, demuxing is handled automatically for you.

Use explicit demuxing when you need:

- Fine-grained control over packet processing
- Access to packet-level metadata (PTS, DTS, flags)
- Custom streaming or network-based video sources
- SEI message extraction during decoding

2.4.3. Two Demuxing Approaches

PyNvVideoCodec provides two ways to demux video data:

File-based demuxing reads directly from video files on disk. This is the simplest approach for processing local files and supports seeking.

In-memory input reads video from memory buffers or seekable streams. For most in-memory decode workflows, prefer `SimpleDecoder` with `bytes`, `bytearray`, `memoryview`, or `BinaryIO` input. Use callback-based demuxing only for non-seekable streaming sources or low-level packet control.

2.4.4. Next Steps

Choose the demuxing approach that fits your use case:

- *Demuxing from File* - For processing local video files

- *Demuxing from Memory* - For in-memory sources, with `SimpleDecoder` recommended for seekable input

2.4.5. Demuxing from File

Extract encoded video packets from local video files using file-based demuxing.

2.4.5.1 Example

The following example demonstrates the complete decode pipeline:

Video File → Demuxer → Packets → Decoder → Raw Frames

Step 1: Create the Demuxer

Import PyNvVideoCodec and create a demuxer by passing the path to your video file:

```
import PyNvVideoCodec as nvc

# Create demuxer to read video file
nv_dmx = nvc.CreateDemuxer(filename="input.mp4")
```

Step 2: Query Stream Properties

The demuxer exposes stream metadata that you can use to configure the decoder or for display purposes:

```
# Query stream properties for decoder setup
print("FPS:", nv_dmx.FrameRate())
print("Resolution:", nv_dmx.Width(), "x", nv_dmx.Height())
```

Step 3: Create the Decoder

Create a hardware decoder using the codec information from the demuxer. The `GetNvCodecId()` method returns the codec type detected in the video stream:

```
# Create decoder using demuxer's codec information
nv_dec = nvc.CreateDecoder(
    gpuid=0,
    codec=nv_dmx.GetNvCodecId(),
    usedevicememory=True
)
```

Step 4: Iterate and Decode

The demuxer is iterable. Loop over it to retrieve packets, then pass each packet to the decoder. The decoder may return zero, one, or multiple frames per packet (due to B-frame reordering):

```
# Iterate over packets and decode
for packet in nv_dmx:
    # Decode returns a list of frames (0 to N depending on B-frame reordering)
    for decoded_frame in nv_dec.Decode(packet):
        # Process frame - access via CUDA Array Interface
        frame_ptr = decoded_frame.cuda()
        # ... process frame data ...
```

2.4.5.2 Note

- The demuxer uses FFmpeg internally for container parsing.
- Seeking accuracy depends on keyframe placement in the video. The demuxer seeks to the nearest keyframe *before* the requested timestamp.
- The decoder may buffer frames internally for B-frame reordering. After processing all packets, call `Flush()` on the decoder to retrieve remaining buffered frames.
- For buffer-based demuxing (streaming, network sources), see *Demuxing from Memory*.

2.4.5.3 APIs Used

The following APIs are used in this example:

- CreateDemuxer() - Create a demuxer from a video file
- Demuxer.FrameRate() - Get the video frame rate
- Demuxer.Width() / Height() - Get the video dimensions
- Demuxer.GetNvCodecId() - Get the codec identifier for decoder creation
- CreateDecoder() - Create a hardware decoder
- Demuxer Iterator - Iterate over video packets
- Decoder.Decode() - Decode a packet and return frames

2.4.5.4 Sample Applications

See these sample applications in the `samples/advanced/` directory:

- decode.py - Basic video decoding using demuxer and native decoder. Demonstrates the complete pipeline from file to raw YUV frames.
- decode_with_cuda_control.py - Decoding with explicit CUDA context and stream management for advanced GPU control.
- decode_with_low_latency.py - Low-latency decoding modes for real-time applications.

2.4.6. Demuxing from Memory

Process video data directly from memory buffers and streams.

Important

New in this release: for most seekable in-memory decode workflows, use `SimpleDecoder` directly with `bytes`, `bytearray`, `memoryview`, or a seekable `BinaryIO` stream. `SimpleDecoder` handles demuxing internally and supports random frame access, slicing, batch retrieval, and metadata queries.

Use callback-based `CreateDemuxer` only when the source is non-seekable, data arrives in chunks, or you need full access to the low-level demuxer and decoder pipeline.

2.4.6.1 Recommended: SimpleDecoder with In-Memory Input

If your use case does not require packet-level control, `SimpleDecoder` provides the preferred path for in-memory decoding. It accepts `bytes`, `bytearray`, `memoryview`, or any seekable `BinaryIO` stream directly as its source argument:

```
import io
import PyNvVideoCodec as nvc

# From a bytes buffer already in memory
with open("input.mp4", "rb") as f:
    raw = f.read()

decoder = nvc.SimpleDecoder(raw, gpu_id=0)              # bytes
decoder = nvc.SimpleDecoder(io.BytesIO(raw), gpu_id=0)  # BinaryIO (no full-
↪copy)

# All seekable API methods work with any input type
frame = decoder[42]                                     # random access
frames = decoder[10:20]                                 # slice
decoder.get_batch_frames_by_index([0, 50, 100])         # batch by index
```

Choose `SimpleDecoder` in-memory input when:

- You need random frame access, slicing, or out-of-order batch retrieval from an in-memory source.
- Your source is a seekable stream such as `io.BytesIO`, an open file handle, a tar archive member, or an S3/HTTP range reader.
- You want to avoid implementing a custom callback and managing chunk boundaries.

2.4.6.2 Advanced: Callback-Based Demuxing

The following example demonstrates the lower-level callback-based demuxing path where video data is read from memory instead of directly from a file:

Memory Buffer → Data Feeder → Demuxer → Packets → Decoder → Raw Frames

Step 1: Create a Data Feeder Class

Create a class that reads video data into memory and provides a callback method to feed chunks to the demuxer:

```
class VideoStreamFeeder:
    """Class to handle feeding video data in chunks to the demuxer."""

    def __init__(self, file_path):
        # Read entire file into 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)

    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 (EOF)
```

(continues on next page)

(continued from previous page)

```
        """
        buffer_capacity = len(demuxer_buffer)
        chunk_size = min(self.bytes_remaining, buffer_capacity)

        if chunk_size == 0:
            return 0  # Signal end of stream

        # Copy data to demuxer buffer
        demuxer_buffer[:] = self.video_buffer[self.current_pos:self.current_
↪pos + chunk_size]

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

Step 2: Create the Buffer-Based Demuxer

Pass the callback function to `CreateDemuxer()` instead of a filename. The demuxer will call this function whenever it needs more data:

```
import PyNvVideoCodec as nvc

# Create data feeder with video file loaded into memory
data_feeder = VideoStreamFeeder("input.mp4")

# Create demuxer using the callback function
buffer_demuxer = nvc.CreateDemuxer(data_feeder.feed_chunk)
```

Step 3: Create the Decoder

Create a hardware decoder using the codec information from the demuxer, the same as file-based demuxing:

```
# Create decoder using demuxer's codec information
buffer_decoder = nvc.CreateDecoder(
    gpuid=0,
    codec=buffer_demuxer.GetNvCodecId(),
    cudacontext=0,
    cudastream=0,
    usedevicememory=True
)
```

Step 4: Iterate and Decode

The demuxer is iterable. Loop over it to retrieve packets, then pass each packet to the decoder:

```
# Iterate over packets and decode
for packet in buffer_demuxer:
    for decoded_frame in buffer_decoder.Decode(packet):
        # Process frame - access via CUDA Array Interface
        frame_ptr = decoded_frame.cuda()
        # ... process frame data ...
```

2.4.6.3 Note

- The callback function receives a pre-allocated buffer from the demuxer and must return the number of bytes copied.
- Return 0 from the callback to signal end of stream.
- This approach is useful for non-seekable network streaming, encrypted content, or video data from databases.
- The decode pipeline after demuxer creation is identical to file-based demuxing.
- This callback-based demuxer is **non-seekable**: it supports only sequential decode and does not support random frame access, slicing, or index-based retrieval.

2.4.6.4 APIs Used

The following APIs are used in these examples:

- SimpleDecoder() - Decode directly from file paths, bytes-like objects, and seekable BinaryIO streams
- CreateDemuxer(callback) - Create a demuxer using a callback function for buffer-based input
- Demuxer.GetNvCodecId() - Get the codec identifier for decoder creation
- CreateDecoder() - Create a hardware decoder
- Demuxer Iterator - Iterate over video packets
- Decoder.Decode() - Decode a packet and return frames

2.4.6.5 Sample Applications

See these sample applications for complete implementations:

- simple_decode_from_memory.py - SimpleDecoder decoding from bytes, bytearray, memoryview, and BinaryIO streams (fully seekable)
- decode_from_memory_buffer.py - Callback-based buffer demuxing with a VideoStreamFeeder class (non-seekable, sequential decode)

2.4.7. Stream Metadata

Query video stream metadata using PyNvVideoCodec's demuxer and decoder APIs.

PyNvVideoCodec provides APIs to query video stream metadata including resolution, codec, frame rate, duration, and more. This metadata is useful for configuring processing pipelines and understanding video properties.

2.4.7.1 APIs

The following APIs are available for querying stream metadata:

SimpleDecoder

- get_stream_metadata() - Get basic stream metadata (codec, resolution, frame rate, duration)
- get_scanned_stream_metadata() - Get accurate metadata by scanning the entire video file

ThreadedDecoder

- get_stream_metadata() - Get basic stream metadata
- get_scanned_stream_metadata() - Get accurate metadata by scanning

Demuxer

- FrameRate() - Get video frame rate
- Width() / Height() - Get video dimensions
- GetNvCodecId() - Get codec identifier
- ChromaFormat() - Get chroma subsampling format
- BitDepth() - Get bit depth

2.4.7.2 Related Topics

- *Demuxing from File* - File-based demuxing workflow
- *Frame Sampling with SimpleDecoder* - SimpleDecoder usage
- *ThreadedDecoder* - ThreadedDecoder usage

2.5. Video Decoding

PyNvVideoCodec provides robust hardware-accelerated video decoding capabilities, leveraging NVIDIA GPUs to efficiently decode various video formats. This section introduces three decoder interfaces, each optimized for specific use cases, and explains how to use them for frame sampling and decoding.

2.5.1. Overview of Decoder Interfaces and Selecting the Right One

Understand the different decoder interfaces available in PyNvVideoCodec and how to choose the right one for your use case.

PyNvVideoCodec provides two high-level decoder interfaces optimized for common use cases. For advanced scenarios requiring fine-grained control, a low-level decoding API is also available.

2.5.1.1 Available Decoder Interfaces

SimpleDecoder

The SimpleDecoder is a high-level interface designed for ease of use. It provides built-in demuxing, frame indexing, and random access capabilities. It accepts file paths and, new in this release, seekable in-memory inputs such as `bytes`, `bytearray`, `memoryview`, and `BinaryIO` streams.

ThreadedDecoder

The ThreadedDecoder is optimized for maximum throughput in batch processing scenarios. It uses internal threading to overlap decoding with frame processing.

2.5.1.2 Low-Level Decoding API

For advanced scenarios requiring fine-grained control, use `CreateDecoder()` to create a native decoder. This requires explicit demuxing but offers control over packet processing, SEI message extraction, low-latency modes, and resolution reconfiguration. See *Core Decoder for Low-Level Control* for details.

2.5.2. Video Decoding and Frame Sampling Using SimpleDecoder

Learn how to efficiently sample frames from videos for deep learning training and inference using PyNvVideoCodec's SimpleDecoder.

The SimpleDecoder provides a powerful and flexible interface for frame sampling from video datasets. It supports multiple access patterns optimized for different deep learning workflows, from training data preparation to real-time inference.

2.5.2.1 SimpleDecoder Input Sources

`SimpleDecoder` can be constructed from a file path or from seekable in-memory video data. Use file paths for videos that are already on disk. Use in-memory input when video content is already in memory, comes from object storage, or is read from an archive where writing a temporary file is undesirable.

Important

New in this release: `SimpleDecoder` accepts `bytes`, `bytearray`, `memoryview`, and seekable `BinaryIO` streams. For seekable in-memory sources, prefer this `SimpleDecoder` input support instead of the older callback-based `CreateDemuxer` + `CreateDecoder` pipeline. Use callback-based demuxing only for non-seekable streaming sources or packet-level control.

All in-memory input types are fully seekable and support the complete `SimpleDecoder` API - random frame access, slicing, batch retrieval, and metadata queries work exactly as with file-based input.

Supported source types include:

- **File path** - the simplest option when the video is available on local disk.
- **bytes / bytearray / memoryview** - the buffer is copied once into native memory; FFmpeg reads from it with no per-read Python overhead. Best when the whole clip is already in RAM.
- **BinaryIO stream** - any seekable file-like object (`io.BytesIO`, an open file handle, a tar archive member, an S3/HTTP range reader, etc.). FFmpeg pulls only the bytes it needs on demand - no full-file copy. Best for large or remote sources.

```
import io
import PyNvVideoCodec as nvc

# File path - simplest for local files
decoder = nvc.SimpleDecoder("input.mp4", gpu_id=0)

# Load file once; reuse as different in-memory input types
with open("input.mp4", "rb") as f:
```

(continues on next page)

(continued from previous page)

```
    raw = f.read()

# bytes - full buffer copied into native memory
decoder = nvc.SimpleDecoder(raw, gpu_id=0)

# io.BytesIO - seekable stream, FFmpeg pulls bytes on demand
decoder = nvc.SimpleDecoder(io.BytesIO(raw), gpu_id=0)

# All seekable API methods work identically with any input type
total = len(decoder)
frame = decoder[0]                                  # random access
frames = decoder[10:20]                             # slice
batch = decoder.get_batch_frames_by_index([0, 50, 100])
```

For non-seekable streaming sources or packet-level demuxing control, see *Demuxing from Memory*.

2.5.2.2 Example

The following example uses file-path input and demonstrates multi-file video decoding with frame sampling and PyTorch tensor conversion:

Video Files → SimpleDecoder → Frame Sampling → PyTorch Tensors

Step 1: Create the SimpleDecoder

Create a SimpleDecoder with RGB output format for deep learning workflows.

```
import PyNvVideoCodec as nvc

decoder = nvc.SimpleDecoder(
    video_path,
    gpu_id=0,
    use_device_memory=True,
    output_color_type=nvc.OutputColorType.RGB  # RGB format for DL
)
```

Step 2: Get Total Frame Count

Use `len()` to get the total number of frames in the video:

```
# Get total frames in the video
total_frames = len(decoder)
print(f"Video has {total_frames} frames")
```

Step 3: Calculate Sample Indices

Create evenly spaced frame indices across the video duration for balanced sampling:

```
import numpy as np

# Sample frames evenly across the video
num_frames = 16  # Number of frames to sample
frame_indices = np.linspace(0, total_frames-1, num_frames, dtype=int).tolist()
print(f"Sampling frames at indices: {frame_indices}")
```

Step 4: Get Batch Frames by Index

Use `get_batch_frames_by_index()` to retrieve specific frames in one operation:

```
# Get batch of frames by indices
decoded_frames = decoder.get_batch_frames_by_index(frame_indices)
```

Step 5: Convert to PyTorch Tensors

Convert decoded frames to PyTorch tensors using DLPack for zero-copy transfer:

```
import torch

# Convert frames to torch tensors
frames_tensor = torch.stack([
    torch.from_dlpack(frame) for frame in decoded_frames
])
print(f"Tensor shape: {frames_tensor.shape}")  # [N, H, W, C]
```

Step 6: Reconfigure Decoder for Multiple Videos

Reuse the decoder for subsequent videos using `reconfigure_decoder()`:

```
# Process multiple video files efficiently
video_files = ['video1.mp4', 'video2.mp4', 'video3.mp4']

for i, video_file in enumerate(video_files):
    if i == 0:
        # First video - decoder already created
        pass
    else:
        # Reconfigure decoder for subsequent files
        torch.cuda.current_stream().synchronize()
        decoder.reconfigure_decoder(video_file)

    # Process frames from current video
    total_frames = len(decoder)
    frame_indices = np.linspace(0, total_frames-1, num_frames, dtype=int).
↪tolist()
    decoded_frames = decoder.get_batch_frames_by_index(frame_indices)
    # ... process frames ...
```

2.5.2.3 Frame Access Patterns

SimpleDecoder supports multiple frame fetching patterns:

- **Single Frame:** `decoder[10]` - Access frame at index 10
- **Slice:** `decoder[0:100:5]` - Get every 5th frame from 0 to 100
- **Sequential Batch:** `decoder.get_batch_frames(16)` - Get 16 consecutive frames
- **Indexed Batch:** `decoder.get_batch_frames_by_index([0, 10, 20])` - Get specific frames

2.5.2.4 Note

- SimpleDecoder requires seekable container formats (MP4, MKV, AVI). Elementary streams are not supported.
- `reconfigure_decoder()` accepts file paths only. To process successive in-memory clips, construct a new `SimpleDecoder` for each clip.
- Use `output_color_type=nvc.OutputColorType.RGBP` for planar CHW format (common in PyTorch models).
- Call `torch.cuda.current_stream().synchronize()` before reconfiguring file-path decoders to ensure all GPU operations complete.

2.5.2.5 APIs Used

The following APIs are used in this section:

- SimpleDecoder() - Constructor with all parameters
- len(decoder) - Get total frame count
- decoder[index] - Single frame and slice access
- get_batch_frames() - Get sequential batch of frames
- get_batch_frames_by_index() - Get frames by specific indices
- seek_to_index() - Seek to specific frame position
- get_index_from_time_in_seconds() - Convert time to frame index
- get_stream_metadata() - Get basic stream metadata
- get_scanned_stream_metadata() - Get accurate metadata by scanning
- reconfigure_decoder() - Reconfigure for different video

2.5.2.6 Sample Applications

PyNvVideoCodec includes sample applications demonstrating SimpleDecoder usage:

- simple_decode_from_memory.py - Decoding from bytes, bytearray, memoryview, and BinaryIO streams
- simple_decode_sampling.py - Multi-file video decoding with frame sampling and PyTorch tensor conversion
- simple_decode_tutorial.ipynb - Interactive Jupyter notebook tutorial covering multiple frame access methods

These samples can be found in the `samples/` directory.

2.5.3. Decoder Caching

Efficiently process multiple video files by reusing decoder instances with SimpleDecoder's built-in caching mechanism.

When processing multiple video files, creating a new decoder for each video introduces significant overhead. SimpleDecoder addresses this with decoder caching - an LRU (Least Recently Used) cache that stores and reuses decoder instances based on video properties.

2.5.3.1 Example

The following example demonstrates efficient processing of multiple video files using decoder caching:

Video Files → SimpleDecoder (with cache) → Reconfigure → Process Next Video

Step 1: Create SimpleDecoder with Caching Parameters

Configure the decoder with `max_width`, `max_height`, and `decoder_cache_size` to enable caching across multiple videos:

```
import PyNvVideoCodec as nvc

# Create decoder with caching enabled
decoder = nvc.SimpleDecoder(
    "video1.mp4",
    gpu_id=0,
    use_device_memory=True,
    max_width=2048,
    max_height=2048,
    decoder_cache_size=4  # Cache up to 4 decoder instances
)
```

Step 2: Process First Video

Decode frames from the first video using any of SimpleDecoder's access methods:

```
# Get total frames and process
total_frames = len(decoder)
print(f"Video 1 has {total_frames} frames")

# Access frames using indexing
frames = decoder[0:10]  # Get first 10 frames
```

Step 3: Reconfigure for Next Video

Use `reconfigure_decoder()` to switch to a new video source. If the new video's properties match a cached decoder, it will be reused:

```
# Reconfigure decoder for next video
decoder.reconfigure_decoder("video2.mp4")

# Process the new video
total_frames = len(decoder)
print(f"Video 2 has {total_frames} frames")

frames = decoder[0:10]  # Get first 10 frames
```

Step 4: Process Multiple Videos in a Loop

Efficiently process a batch of video files:

```
video_files = ["video1.mp4", "video2.mp4", "video3.mp4"]

for i, video_file in enumerate(video_files):
    if i == 0:
        # First video - decoder already created
```

(continues on next page)

(continued from previous page)

```
        pass
    else:
        # Reconfigure for subsequent videos
        decoder.reconfigure_decoder(video_file)

    # Process frames from current video
    total_frames = len(decoder)
    frames = decoder[0:16]  # Sample first 16 frames
    print(f"Processed {len(frames)} frames from {video_file}")
```

2.5.3.2 Cache Behavior

The decoder cache uses an LRU (Least Recently Used) eviction policy:

1. **Lookup**: When reconfiguring, SimpleDecoder checks the cache for a decoder matching the new video's properties
2. **Reuse**: If a matching decoder is found (cache hit), it's reused immediately
3. **Create**: If no match is found (cache miss), a new decoder is created
4. **Eviction**: If the cache is full, the least recently used decoder is removed

Cache Key Properties:

Decoders are matched based on:

- Video codec (H.264, HEVC, VP9, AV1)
- Bit depth (8-bit, 10-bit, 12-bit)
- Chroma format (4:2:0, 4:2:2, 4:4:4)
- Resolution within `max_width` and `max_height`

2.5.3.3 Note

- Set `max_width` and `max_height` to the largest resolution you expect to process for maximum cache reuse.
- Increase `decoder_cache_size` if processing videos with different codecs or bit depths.
- Videos with the same codec, bit depth, and chroma format will share cached decoders.
- Cache is automatically managed - no manual cleanup required.

2.5.3.4 APIs Used

The following APIs are used in this example:

- SimpleDecoder() - Constructor with caching parameters
- len(decoder) - Get total frame count
- decoder[index] - Frame access using indexing
- reconfigure_decoder() - Switch to a different video source

2.5.3.5 Sample Applications

See this sample application for a complete implementation:

- simple_decode_sampling.py - Multi-file video decoding with decoder reconfiguration

2.5.4. High-Throughput Pipelines Using ThreadedDecoder

ThreadedDecoder enables background frame decoding on a dedicated thread, ensuring a continuous supply of ready-to-process frames for inference pipelines.

ThreadedDecoder continuously decodes frames in the background and maintains a preloaded buffer of ready-to-use frames. With this approach the decoder latency could be hidden behind inference.

2.5.4.1 Example

The following example demonstrates ThreadedDecoder usage for video analytics pipelines:

Video File → ThreadedDecoder (Background Prefetch) → Batched Frames → PyTorch Tensors

Step 1: Import Required Modules

Import ThreadedDecoder and OutputColorType from PyNvVideoCodec, along with PyCUDA for GPU context management:

```
from PyNvVideoCodec import ThreadedDecoder, OutputColorType
import pycuda.driver as cuda
from pycuda.autoinit import context

import torch
```

Output Color Formats

Choose the output format based on your model requirements:

- `OutputColorType.RGBP` - Planar RGB (CHW format). Preferred for most PyTorch/TensorFlow models.
- `OutputColorType.RGB` - Interleaved RGB (HWC format). Use when your pipeline expects HWC layout.
- `OutputColorType.NV12` - Native decoder output. Most efficient if your pipeline can handle YUV.

Note

- ThreadedDecoder prefetches frames in the background, so `get_batch_frames()` returns immediately with already-decoded frames.
- An empty list from `get_batch_frames()` indicates end of video.
- Use `torch.cuda.current_stream().synchronize()` before reconfiguring to ensure all GPU operations complete.
- For random access patterns, consider *SimpleDecoder* instead.

Step 2: Create the ThreadedDecoder

Initialize ThreadedDecoder with the video path, buffer size, and output color format. Use `OutputColorType.RGBP` (planar RGB in CHW format) for deep learning models:

```
# Configure decoder parameters
color_format = OutputColorType.RGBP  # Planar RGB (CHW) for DL models
batch_size = 3   # Process 3 frames at a time

# Initialize ThreadedDecoder
decoder = ThreadedDecoder(
    enc_file_path="input.mp4",    # Input video path
    buffer_size=12,               # Number of frames to prefetch
    gpu_id=0,                     # GPU device ID
    use_device_memory=True,       # Keep frames in GPU memory
    output_color_type=color_format
)
```

Step 3: Get Stream Metadata

Query the video stream metadata to get the total number of frames and other properties:

```
# Get video information
metadata = decoder.get_stream_metadata()
num_frames = metadata.num_frames
print(f"Video has {num_frames} frames")
```

Step 4: Process Frames in Batches

Use `get_batch_frames()` to retrieve prefetched frames. Convert to PyTorch tensors using DLPack for zero-copy transfer:

```
# Process video frames in batches
frame_count = 0
while frame_count < num_frames:
    # Get batch of prefetched frames (returns immediately)
    frames = decoder.get_batch_frames(batch_size)
    if len(frames) == 0:
        break

    # Convert frames to PyTorch tensors
    for frame in frames:
        tensor = torch.from_dlpack(frame)
        # tensor shape: [C, H, W] for RGBP, [H, W, C] for RGB

        # Normalize for model input
        normalized = tensor.float() / 255.0

        # ... run inference with your model ...

    frame_count += len(frames)
```

Step 5: Reconfigure for Multiple Videos

Reuse the decoder for subsequent videos using `reconfigure_decoder()`:

```
# Process multiple video files efficiently
video_files = ['video1.mp4', 'video2.mp4', 'video3.mp4']

for i, video_file in enumerate(video_files):
```

(continues on next page)

(continued from previous page)

```
    if i == 0:
        # First video - decoder already created
        pass
    else:
        # Reconfigure decoder for subsequent files
        torch.cuda.current_stream().synchronize()
        decoder.reconfigure_decoder(video_file)

    # Process frames from current video
    metadata = decoder.get_stream_metadata()
    while True:
        frames = decoder.get_batch_frames(batch_size)
        if len(frames) == 0:
            break
        # ... process frames ...
```

2.5.4.2 Buffer Size Selection

The `buffer_size` parameter controls how many frames are prefetched in the background:

- Recommended: 2-3x your batch size (e.g., for `batch_size=4`, use `buffer_size=8-12`)
- Larger buffers provide more cushion for variable inference times but consume more GPU memory
- Smaller buffers reduce memory usage but may cause stalls if inference is slower than decoding

2.5.4.3 APIs Used

The following APIs are used in this example:

- ThreadedDecoder() - Constructor with all parameters
- get_stream_metadata() - Get video stream metadata
- get_batch_frames() - Get batch of prefetched frames
- reconfigure_decoder() - Reconfigure for different video

2.5.4.4 Sample Applications

PyNvVideoCodec includes sample applications demonstrating ThreadedDecoder usage:

- object_detection_tutorial.ipynb - Interactive Jupyter notebook demonstrating ThreadedDecoder integration with Faster R-CNN model for real-time object detection

These samples are located in the `samples/jupyter/` directory.

2.5.5. Core Decoder for Low-Level Control

The Core Decoder provides direct access to NVDEC hardware for fine-grained control over video decoding operations.

The Core Decoder (also known as the native decoder) is the low-level decoding interface that gives you complete control over the decode pipeline. Unlike SimpleDecoder and ThreadedDecoder which handle demuxing internally, the Core Decoder requires explicit demuxing and packet management.

2.5.5.1 When to Use Core Decoder

Use the Core Decoder when you need:

- ***SEI message extraction*:** Access to Supplemental Enhancement Information embedded in the video stream
- ***Low-latency decoding*:** Control over decode latency modes for real-time applications
- ***Resolution reconfiguration*:** Switch between videos with different resolutions without recreating the decoder
- ***Packet-level control*:** Fine-grained control over individual packet processing
- ***Custom streaming sources*:** Decode from network streams or memory buffers
- ***Decode statistics*:** Extract QP values, coding-unit types, and motion vectors

2.5.5.2 Decode Pipeline

The Core Decoder pipeline requires explicit management of each stage:

Video File → **Demuxer** → **Packets** → **Core Decoder** → **Raw Frames**

You must create a demuxer to extract packets from the container format, then feed those packets to the decoder. This separation provides flexibility but requires more code than the high-level interfaces.

2.5.5.3 Example

The following example demonstrates the complete Core Decoder workflow:

```
import PyNvVideoCodec as nvc

# Step 1: Create demuxer to read video file
nv_dmx = nvc.CreateDemuxer(filename="input.mp4")

# Step 2: Query stream properties
print(f"Resolution: {nv_dmx.Width()}x{nv_dmx.Height()}")
print(f"Codec: {nv_dmx.GetNvCodecId()}")
print(f"FPS: {nv_dmx.FrameRate()}")

# Step 3: Create Core Decoder using demuxer's codec information
nv_dec = nvc.CreateDecoder(
    gpuid=0,
    codec=nv_dmx.GetNvCodecId(),
    usedevicememory=True
)

# Step 4: Iterate over packets and decode
frame_count = 0
for packet in nv_dmx:
    # Decode returns a list of frames (0 to N due to B-frame reordering)
    for decoded_frame in nv_dec.Decode(packet):
        # Access frame via CUDA Array Interface
        frame_ptr = decoded_frame.cuda()
        frame_count += 1
        # ... process frame data ...
```

(continues on next page)

(continued from previous page)

```
# Step 5: Flush remaining frames from decoder buffer
for decoded_frame in nv_dec.Flush():
    frame_count += 1

print(f"Decoded {frame_count} frames")
```

2.5.5.4 Resolution Reconfiguration

The Core Decoder supports dynamic resolution changes using `setReconfigParams()`. This allows you to decode multiple videos with different dimensions using a single decoder instance:

```
# Create decoder with max dimensions to accommodate all streams
nv_dec = nvc.CreateDecoder(
    gpuid=0,
    codec=codec_id,
    usedevicememory=True,
    maxwidth=3840,    # Maximum width across all videos
    maxheight=2160    # Maximum height across all videos
)

# Decode first video...

# Reconfigure for second video with different dimensions
nv_dec.setReconfigParams(new_width, new_height)

# Continue decoding second video...
```

2.5.5.5 APIs Used

The following APIs are used with the Core Decoder:

- CreateDemuxer() - Create a demuxer to extract packets
- CreateDecoder() - Create the Core Decoder
- Decoder.Decode() - Decode a packet and return frames
- Decoder.Flush() - Flush remaining buffered frames
- Decoder.setReconfigParams() - Reconfigure decoder for new resolution

2.5.5.6 Sample Applications

See these sample applications demonstrating Core Decoder usage:

- decode.py - Basic decoding with Core Decoder
- decode_with_cuda_control.py - Explicit CUDA context and stream management
- decode_with_low_latency.py - Low-latency decoding modes
- decode_reconfigure.py - Dynamic resolution reconfiguration
- decode_sei_msg.py - SEI message extraction

2.5.6. Latency Modes

Configure decode latency modes for real-time and low-latency video processing applications.

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.

2.5.6.1 DisplayDecodeLatencyType Enumeration

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.

2.5.6.2 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.

2.5.6.3 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 \
```

(continues on next page)

(continued from previous page)

```
        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.

2.5.6.4 Sample Applications

See the following sample application for a complete low-latency decoding implementation:

- decode_with_low_latency.py - Demonstrates all three latency modes with proper packet flag handling

2.5.7. SEI Message Decoding

Extract and process Supplemental Enhancement Information (SEI) messages from video streams.

SEI (Supplemental Enhancement Information) messages are metadata embedded in video bitstreams that provide additional information such as HDR metadata, timecode data, and custom application-specific data.

2.5.7.1 Example

The following example demonstrates SEI message extraction from a video file:

Video File → Demuxer → Decoder (SEI enabled) → Decoded Frames → SEI Messages

Step 1: Initialize CUDA Context

Initialize PyCUDA and create a CUDA context for GPU operations:

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

cuda.init()
cuda_device = cuda.Device(0)
cuda_ctx = cuda_device.retain_primary_context()
cuda_ctx.push()
cuda_stream = cuda.Stream()
```

Step 2: Create Demuxer

Create a demuxer to read the video file and extract encoded packets:

```
# Create demuxer to read video file
nv_dmx = nvc.CreateDemuxer(filename="input.mp4")

print(f"FPS = {nv_dmx.FrameRate()}")
```

Step 3: Create Decoder with SEI Enabled

Create a decoder with `enableSEIMessage=1` to enable SEI message extraction:

```
# Create decoder with SEI extraction enabled
nv_dec = nvc.CreateDecoder(
    gpuid=0,
    codec=nv_dmx.GetNvCodecId(),
    cudacontext=cuda_ctx.handle,
    cudastream=cuda_stream.handle,
    usedevicememory=True,
    enableSEIMessage=1  # Enable SEI message extraction
)
```

Step 4: Decode and Extract SEI Messages

Iterate over packets, decode frames, and extract SEI messages using `getSEIMessage()`:

```
import ctypes

# Decode and extract SEI messages
for packet in nv_dmx:
    for decoded_frame in nv_dec.Decode(packet):
        # Get SEI messages from decoded frame
        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"]

                print(f"SEI Type: {sei_type}, Size: {len(sei_message)} bytes")
```

Step 5: Parse SEI Message Types

Parse different SEI message types using ctypes structures. Common types include timecode, HDR metadata (mastering display, content light level), and alternative transfer characteristics:

```
# Parse SEI based on type (when sei_uncompressed == 1)
if sei_uncompressed == 1:
    buffer = (ctypes.c_ubyte * len(sei_message))(*sei_message)

    # Handle different SEI message types
    if sei_type in (nvc.SEI_TYPE.TIME_CODE_H264, nvc.SEI_TYPE.TIME_CODE):
        # Parse timecode structure
        pass
    elif sei_type == nvc.SEI_TYPE.MASTERING_DISPLAY_COLOR_VOLUME:
```

(continues on next page)

(continued from previous page)

```
        # Parse HDR mastering display info
        pass
    elif sei_type == nvc.SEI_TYPE.CONTENT_LIGHT_LEVEL_INFO:
        # Parse content light level info
        pass
    elif sei_type == nvc.SEI_TYPE.ALTERNATIVE_TRANSFER_CHARACTERISTICS:
        # Parse alternative transfer characteristics
        pass
```

2.5.7.2 Common SEI Types

PyNvVideoCodec provides constants for common SEI message types via `nvc.SEI_TYPE`:

- `TIME_CODE` / `TIME_CODE_H264` - Frame timing and sequence information
- `MASTERING_DISPLAY_COLOR_VOLUME` - HDR color space and primaries
- `CONTENT_LIGHT_LEVEL_INFO` - HDR brightness metadata
- `ALTERNATIVE_TRANSFER_CHARACTERISTICS` - Transfer function characteristics

2.5.7.3 Note

- SEI extraction requires using `CreateDecoder` with `enableSEIMessage=1`.
- Not all videos contain SEI messages.
- The `sei_uncompressed` flag indicates if the message can be parsed as a structured type.
- For SEI message encoding, see *SEI Message Encoding*.

2.5.7.4 APIs Used

The following APIs are used in this example:

- CreateDemuxer() - Create a demuxer from a video file
- Demuxer.FrameRate() - Get the video frame rate
- Demuxer.GetNvCodecId() - Get the codec identifier
- CreateDecoder() - Create a hardware decoder with SEI enabled
- Decoder.Decode() - Decode a packet and return frames
- DecodedFrame.getSEIMessage() - Get SEI messages from decoded frame

2.5.7.5 Sample Applications

See this sample application for a complete implementation:

- decode_sei_msg.py - Demonstrates SEI message extraction and parsing for various SEI types including timecode and HDR metadata

2.5.8. Decoder Statistics Extraction

Extract low-level decoding statistics including QP values, coding unit types, and motion vectors for video analysis.

PyNvVideoCodec provides access to detailed decoding statistics. These statistics include QP (Quantization Parameter) values, CU (Coding Unit) types, and motion vectors for each macroblock.

2.5.8.1 Example

The following example demonstrates decode statistics extraction using SimpleDecoder:

Video File → SimpleDecoder (stats enabled) → Decoded Frames → Statistics

Step 1: Create SimpleDecoder with Statistics Enabled

Create a SimpleDecoder with `enableDecodeStats=True` to enable statistics collection:

```
import PyNvVideoCodec as nvc

# Create decoder with statistics collection enabled
simple_decoder = nvc.SimpleDecoder(
    "input.mp4",
    need_scanned_stream_metadata=False,
    use_device_memory=True,
    gpu_id=0,
    enableDecodeStats=True  # Enable statistics collection
)
```

Step 2: Get Stream Metadata

Query stream metadata for video information:

```
# Get video metadata
metadata = simple_decoder.get_stream_metadata()
print(f"Video: {metadata.width}x{metadata.height}")
```

Step 3: Iterate and Extract Statistics

Iterate over decoded frames and check for available statistics using `decode_stats_size`:

```
# Process frames and extract statistics
for frame_idx, decoded_frame in enumerate(simple_decoder):
    # Check if statistics are available for this frame
    if hasattr(decoded_frame, 'decode_stats_size') and decoded_frame.decode_
↪stats_size > 0:
        # Parse the statistics
        parsed_stats = decoded_frame.ParseDecodeStats()

        # Access statistics fields
        qp_values = parsed_stats.get("qp_luma", [])
        cu_types = parsed_stats.get("cu_type", [])

        if len(qp_values) > 0:
            avg_qp = sum(qp_values) / len(qp_values)
            print(f"Frame {frame_idx}: Avg QP = {avg_qp:.2f}")
```

Step 4: Analyze Statistics

The `ParseDecodeStats()` method returns a dictionary with the following fields:

```
# Available statistics fields
parsed_stats = decoded_frame.ParseDecodeStats()

# QP Analysis - compression level per macroblock
qp_luma = parsed_stats["qp_luma"]      # List of QP values (higher = more-
↪compression)

# CU Type Distribution - prediction mode per macroblock
# 0=INTRA, 1=INTER, 2=SKIP, 3=PCM, 7=INVALID
cu_type = parsed_stats["cu_type"]

# Motion Vectors - temporal prediction info
mv0_x = parsed_stats["mv0_x"]  # L0 reference X component
mv0_y = parsed_stats["mv0_y"]  # L0 reference Y component
mv1_x = parsed_stats["mv1_x"]  # L1 reference X component (B-frames)
mv1_y = parsed_stats["mv1_y"]  # L1 reference Y component (B-frames)
```

2.5.8.2 Note

- Statistics collection must be enabled at decoder creation time with `enableDecodeStats=True`.
- Enabling statistics incurs a small performance overhead.
- Supported codecs: H.264 (AVC) and H.265 (HEVC).
- Check `decode_stats_size > 0` before calling `ParseDecodeStats()`.
- CU types: 0=INTRA (spatial prediction), 1=INTER (temporal prediction), 2=SKIP (copy from reference), 3=PCM (uncompressed).

2.5.8.3 APIs Used

The following APIs are used in this example:

- SimpleDecoder() - Constructor with `enableDecodeStats` parameter
- get_stream_metadata() - Get video stream metadata
- `decode_stats_size` - Property indicating statistics data size (>0 if available)
- `ParseDecodeStats()` - Parse statistics into a dictionary

2.5.8.4 Sample Applications

See this sample application for a complete implementation:

- simple_decode_stats.py - SimpleDecoder-based statistics extraction with formatted output including QP analysis, CU type distribution, and motion vector statistics

2.6. Video Encoding

2.6.1. Overview

This section provides an overview of the key workflows and features for video encoding, from basic frame encoding to advanced runtime configuration and metadata handling.

The encoder accepts raw frames from either CPU memory (numpy arrays) or GPU memory (CUDA buffers) and produces encoded bitstream data that can be written to files or streamed.

2.6.2. Topics

- *Basic Encoding Workflow* - Step-by-step guide to encode raw frames to compressed video
- *Encoder Settings* - Configure codec, bitrate, presets, and quality options
- *Encoder Reconfiguration* - Change encoder parameters at runtime without recreating the session
- *SEI Message Encoding* - Embed metadata and custom data in the bitstream

2.6.3. Basic Encoding Workflow

PyNvVideoCodec provides hardware-accelerated video encoding using NVIDIA GPUs. The encoder supports both CPU (host memory) and GPU (device memory) buffer modes.

2.6.3.1 Basic Encoding Workflow

The following steps demonstrate the complete encoding workflow:

Raw Frames → Buffer Preparation → Encoder → Encoded Bitstream

Step 1: Prepare Buffer for Encoding

Prepare input buffers based on your buffer mode. For CPU buffers, read raw YUV data into a numpy array. For GPU buffers, use CUDA device memory objects.

CPU Buffer Mode:

```
import numpy as np

# Calculate frame size based on format (NV12 = height * 1.5)
frame_size = int(width * height * 1.5)

# Read raw YUV frame into numpy array
with open("input.yuv", "rb") as dec_file:
    chunk = np.fromfile(dec_file, np.uint8, count=frame_size)
```

GPU Buffer Mode:

```
# For GPU buffers, use objects implementing CUDA Array Interface
# The object must expose a cuda() method returning device pointers
class AppFrame:
    def __init__(self, width, height, fmt):
        self.frameSize = int(width * height * 1.5)  # NV12
        # Allocate CUDA device memory

    def cuda(self):
```

(continues on next page)

(continued from previous page)

```
        # Return CUDA Array Interface for each plane
        return [self.luma_cuda_interface, self.chroma_cuda_interface]

input_frame = AppFrame(width, height, "NV12")
```

Step 2: Configure and Create Encoder

Create an encoder with `CreateEncoder()` specifying resolution, format, buffer mode, and encoding parameters. See CreateEncoder API Reference for all available parameters.

```
import PyNvVideoCodec as nvc

# Encoder configuration parameters
config_params = {
    "gpu_id": 0,
    "codec": "h264",
    # Additional optional parameters (bitrate, preset, etc.)
}

# Create encoder: usecpuinputbuffer=True for CPU, False for GPU
nvenc = nvc.CreateEncoder(
    width=1920,
    height=1080,
    format="NV12",
    usecpuinputbuffer=True,  # True=CPU buffers, False=GPU buffers
    **config_params
)
```

Step 3: Encode Frames and Flush

Pass frames to `Encode()` to get encoded bitstream. After processing all frames, call `EndEncode()` to flush remaining data from the encoder queue. See Encode API Reference and EndEncode API Reference.

```
with open("output.h264", "wb") as enc_file:
    # Encode each frame
    for i in range(num_frames):
        chunk = np.fromfile(dec_file, np.uint8, count=frame_size)
        if chunk.size == 0:
            break

        # Encode frame - returns bitstream data
        bitstream = nvenc.Encode(chunk)
        enc_file.write(bytearray(bitstream))

    # Flush encoder queue - REQUIRED to get remaining frames
    bitstream = nvenc.EndEncode()
    enc_file.write(bytearray(bitstream))
```

Step 4: Runtime Reconfiguration (Optional)

Change encoder parameters at runtime without recreating the encoder session using `Reconfigure()`. This is useful for adaptive bitrate streaming or handling network conditions. See Reconfigure API Reference for supported parameters.

```
# Get current encoder parameters
reconfig_params = nvenc.GetEncodeReconfigureParams()

# Modify parameters (e.g., change bitrate)
reconfig_params["averageBitrate"] = 5000000  # 5 Mbps

# Apply new configuration
nvenc.Reconfigure(reconfig_params)
```

2.6.3.2 Note

- Supported formats: NV12, ARGB, ABGR, YUV444, YUV420, P010, YUV444_16bit
- Supported codecs: H264, HEVC, AV1
- For GPU buffer mode, input objects must implement the `cuda()` method exposing CUDA Array Interface
- Always call `EndEncode()` at the end to flush remaining encoded data
- Reconfigurable parameters: rateControlMode, averageBitrate, maxBitRate, vbvBufferSize, frameRateNum, frameRateDen

2.6.3.3 Sample Applications

See these sample applications for complete implementations:

- encode.py - Unified encoding supporting both CPU and GPU buffer modes with configurable codec and format options

2.6.3.4 API Reference

For complete API specifications, see:

- CreateEncoder() - Create an encoder instance
- Encode() - Encode a raw frame
- EndEncode() - Flush encoder and get remaining data
- Reconfigure() - Change encoder parameters at runtime

2.6.4. Video Encoder Settings

Detailed explanation of video encoder parameters and configuration options for optimizing encoding quality, performance, and output characteristics.

2.6.4.1 Overview

PyNvVideoCodec provides hardware-accelerated video encoding with extensive configurability. This section explains the important parameters and values they can take, helping you optimize your encoder for specific use cases.

PyNvVideoCodec has been designed for simplified video encoding with appropriate default values. However, you can also access detailed optional parameters and the full flexibility offered by NVIDIA video technology stack.

2.6.4.2 Supported Codecs

NVIDIA GPUs support encoding for H.264, HEVC (H.265), and AV1 codecs. Depending on your hardware generation, not all codecs will be accessible. Refer to the NVIDIA Hardware Video Encoder section for information about supported codecs for each GPU architecture.

Codec Selection Guidelines:

- **H.264:** Best compatibility across all devices and platforms. Suitable for streaming, video conferencing, and general use
- **HEVC:** Better compression efficiency (approximately 50% better than H.264) but requires more powerful decode hardware. Ideal for 4K content, archival, and OTT streaming
- **AV1:** Next-generation codec with superior compression. Best for web streaming and modern devices

2.6.4.3 Presets

Encoder presets control the quality and performance tradeoff. NVENC offers seven presets from P1 (highest performance) to P7 (highest quality). Using these presets will automatically configure all relevant encoding parameters for the selected tuning information.

Preset	Speed	Best For
P1	Fastest	Real-time streaming, live broadcasts, cloud gaming
P2-P3	Fast	Video conferencing, game streaming, screen capture
P4	Balanced (Default)	General-purpose encoding, transcoding workflows
P5-P6	Slow	High-quality archival, OTT streaming, VOD content
P7	Slowest	Maximum quality archival, master copies, premium content

Higher presets produce better quality but encode slower. Specific attributes within a preset can be further tuned if required.

2.6.4.4 Tuning Information

The NVIDIA Encoder Interface exposes different tuning options to optimize the encoder for specific scenarios:

- **High Quality:** Tune presets for latency-tolerant encoding. Suited for high-quality transcoding, video archiving, and encoding for OTT streaming
- **Low Latency:** Tune presets for low latency streaming. Suited for cloud gaming, streaming, video conferencing, and high bandwidth channels with tolerance for bigger occasional frame sizes
- **Ultra-Low Latency:** Tune presets for ultra low latency streaming. Suited for cloud gaming, streaming, and video conferencing in strictly bandwidth-constrained channels
- **Lossless:** Tune presets for lossless encoding. Suited for preserving original video footage for later editing and general lossless data archiving (video or non-video)

- **Ultra High Quality:** Tune presets for latency-tolerant encoding with higher quality. Suited for premium content creation and high-end video production. Only supported for HEVC and AV1 on Turing+ architectures

For low latency use cases (video conferencing), combine `LOW_LATENCY` tuning with `P1` preset and `IPP` GOP pattern (no B-frames). For high quality archival, use `HIGH_QUALITY` tuning with `P6` preset and `IBBBP` GOP pattern.

2.6.4.5 Rate Control and Bitrate

NVENC provides control over various parameters related to the rate control algorithm, allowing it to adapt the bitrate depending on your quality, bandwidth, and performance constraints. NVENC supports the following rate control modes:

Mode	Description	Best For
CBR	Constant Bitrate - Maintains steady bitrate throughout the video	Streaming, broadcasting
VBR	Variable Bitrate - Adjusts bitrate based on content complexity	File storage, VOD
CQP	Constant Quantization Parameter - Fixed quality level regardless of bitrate	Quality testing, research
Target Quality	Targets a specific quality level, varying bitrate as needed	Quality-focused encoding

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

Rate Control Guidelines:

- **CBR for streaming:** Set `codecconfig=CBR` with `bitrate` and `maxbitrate` equal for strict constant bitrate
- **VBR for file storage:** Set `codecconfig=VBR` with `bitrate` as target and `maxbitrate` higher for peaks
- **CQP for constant quality:** Set `codecconfig=CQP` with `qp` parameter (lower = higher quality, typical range: 18-28)

2.6.4.6 Surface Formats

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

Table 2.1: Supported Surface Formats

Format	Description
NV12	Semi-Planar YUV [Y plane followed by interleaved UV plane] - Most efficient format
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]
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`
- NV12 format is most efficient and recommended when possible
- Not all formats are supported on all GPU architectures; refer to your GPU's documentation for specific support information

2.6.4.7 GOP Structure

Group of Pictures (GOP) structure defines the pattern of I-frames (Intra-coded), P-frames (Predictive), and B-frames (Bidirectional predictive):

- **I (Intra):** All-I frames. Largest size but best seek-ability and lowest latency
- **IPP:** I and P frames only. Good for low latency, no B-frames
- **IBP:** I, B, and P frames with one B-frame between references
- **IBBBP:** Multiple B-frames between references. Best compression efficiency

Longer GOPs improve compression efficiency but reduce seek-ability. Typical GOP sizes: 30-250 frames.

2.6.4.8 Common Encoding Scenarios

Recommended settings for common use cases:

Use Case	Codec	Recommended Settings
Live streaming	H264	Preset P1, CBR, LOW_LATENCY, GOP=60
Video archival	HEVC	Preset P6, VBR, HIGH_QUALITY, GOP=250
OTT/VOD content	HEVC or AV1	Preset P4-P5, VBR, HIGH_QUALITY
Video conferencing	H264	Preset P1-P2, CBR, ULTRA_LOW_LATENCY, IPP
Screen recording	H264	Preset P3, VBR or LOSSLESS

2.6.4.9 Building Your Optimized Encoder

To configure NVENC for your specific use case, refer to the Recommended NVENC Settings section in the NVENC Programming Guide.

For advanced parameter tuning and performance optimization, see *Advanced Encoding Parameters*.

2.6.4.10 API Reference

For complete parameter documentation, refer to:

- Encoder API Reference - Complete list of encoder parameters and their valid values

2.6.5. Video Encoding Parameter Details

Table 2.2: 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
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

2.6.6. Encoder Reconfiguration

Dynamic reconfiguration of encoder parameters during encoding sessions for adaptive encoding workflows.

2.6.6.1 Overview

PyNvVideoCodec supports runtime reconfiguration of certain encoder parameters without recreating the encoder instance. This capability is essential for adaptive encoding scenarios where encoding parameters need to change dynamically based on content characteristics, network conditions, or application requirements.

Encoder reconfiguration offers significant performance benefits by avoiding the overhead of encoder creation and destruction. It allows seamless parameter changes during an active encoding session, maintaining encoder state and reducing initialization latency.

2.6.6.2 When to Use Encoder Reconfiguration

Encoder reconfiguration is particularly useful in the following scenarios:

- **Adaptive Bitrate Streaming:** Adjust bitrate dynamically based on available network bandwidth to maintain smooth streaming
- **Dynamic Quality Adjustment:** Change quality settings in response to content complexity or system resource availability
- **Processing Multiple Videos:** Encode multiple videos with different settings without recreating encoder instances, improving efficiency for batch processing
- **Scene-Based Encoding:** Apply different encoding parameters for different scenes within the same video (e.g., higher quality for complex scenes)
- **Real-Time Encoding:** Respond to changing conditions in live streaming or video conferencing applications

2.6.6.3 Reconfigurable Parameters

The following encoder parameters can be reconfigured during an active encoding session:

- **Bitrate:** Target bitrate and maximum bitrate for rate control
- **Frame Rate:** Output frame rate
- **GOP Structure:** I-frame interval and B-frame configuration
- **Quality Parameters:** QP values, VBV buffer size
- **Intra Refresh:** Periodic intra refresh settings

Note: Some parameters cannot be changed once the encoder is created, including codec type, resolution, and profile. For changes to these parameters, a new encoder instance must be created.

2.6.6.4 Reconfiguration Workflow

To reconfigure an encoder during encoding, call the `Reconfigure()` method with the new parameter values. The method accepts parameters like `bitrate`, `framerate`, `maxbitrate`, and other reconfigurable settings.

The typical workflow is:

1. Create encoder with initial settings

2. Encode frames with initial configuration
3. Call `Reconfigure()` with new parameters when needed
4. Continue encoding with the new settings

2.6.6.5 Adaptive Bitrate Encoding

Adaptive bitrate encoding adjusts encoder parameters based on network conditions. The application periodically checks available bandwidth and calls `Reconfigure()` to update bitrate and maxbitrate parameters when significant changes are detected.

Key considerations for adaptive encoding:

- Use LOW_LATENCY tuning mode for streaming scenarios
- Set an appropriate check interval (e.g., every 30 frames)
- Include a buffer margin (e.g., 20%) when setting maxbitrate
- Avoid reconfiguring on every frame to minimize overhead

2.6.6.6 Batch Processing with Reconfiguration

Reconfiguration improves efficiency when processing multiple videos with different encoding requirements. Instead of creating new encoder instances for each video, use `Reconfigure()` to change parameters between videos.

When planning for batch processing with varying resolutions, specify `max_width` and `max_height` during encoder creation to allow reconfiguration up to those limits.

2.6.6.7 Important Considerations

- **Flush Before Reconfiguration:** In some cases, it may be necessary to flush the encoder before reconfiguring to ensure all pending frames are encoded with previous settings
- **Parameter Compatibility:** Not all parameter combinations can be changed at runtime. Refer to the API documentation for limitations
- **Performance Impact:** While reconfiguration is faster than recreating an encoder, there is still a small performance cost. Avoid reconfiguring on every frame
- **Resolution Limits:** When reconfiguring resolution (if supported), the new resolution must not exceed the `max_width` and `max_height` specified during encoder creation

2.6.6.8 Sample Applications

PyNvVideoCodec includes sample applications demonstrating encoder reconfiguration:

- encode_reconfigure.py: Demonstrates dynamic bitrate and frame rate changes during encoding

These samples are located in the `samples/` directory.

2.6.6.9 API Reference

For complete documentation of reconfigurable parameters and method signatures, refer to:

- Encoder API Reference - Reconfigure() method documentation
- CreateEncoder API Reference - Parameters and their valid ranges

2.6.7. Encoding SEI Messages

Insert Supplemental Enhancement Information (SEI) messages into encoded video streams for embedding metadata.

SEI messages are metadata containers that can be embedded in H.264/HEVC/AV1 bitstreams. Common uses include HDR metadata, timecodes, closed captions, and custom application data.

2.6.7.1 Example

The following example demonstrates SEI message insertion during encoding:

Raw Frames + SEI Data → Encoder → Encoded Bitstream with SEI

Step 1: Define SEI Message Data

Create SEI message payloads as byte arrays. For User Data Unregistered (type 5), the payload typically starts with a 16-byte UUID:

```
# Define SEI message payloads (16-byte UUID for User Data Unregistered)
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]
```

Step 2: Create SEI Info Dictionary

Specify the SEI type based on codec. Use type 5 (User Data Unregistered) for H.264/HEVC, or type 6 for AV1:

```
import PyNvVideoCodec as nvc

# Determine SEI type based on codec
codec = "h264"  # or "hevc", "av1"

if codec in ["hevc", "h264"]:
    sei_info = {"sei_type": 5}  # User Data Unregistered
elif codec == "av1":
    sei_info = {"sei_type": 6}  # Metadata OBU for AV1
```

Step 3: Create SEI Messages List

Combine SEI info and payload into a list of tuples. Multiple SEI messages can be inserted per frame:

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

Step 4: Create Encoder and Encode with SEI

Pass the SEI messages list as the third argument to `Encode()`:

```
# Create encoder
config_params = {"gpu_id": 0, "codec": codec}
nvenc = nvc.CreateEncoder(1920, 1080, "NV12", False, **config_params)
```

(continues on next page)

(continued from previous page)

```
# Encode frame with SEI messages
# Encode(frame, pic_flags, sei_messages)
bitstream = nvenc.Encode(input_frame, 0, sei_messages)
enc_file.write(bytearray(bitstream))

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

2.6.7.2 Common SEI Types

- **Type 5** (H.264/HEVC) - User Data Unregistered: Custom metadata with 16-byte UUID
- **Type 4** (H.264/HEVC) - User Data Registered: Closed captions (CEA-608/708)
- **Type 137** (HEVC) - Mastering Display Color Volume: HDR display metadata
- **Type 144** (HEVC) - Content Light Level: HDR luminance levels
- **Type 6** (AV1) - Metadata OBU: Custom metadata for AV1

2.6.7.3 Note

- SEI messages are passed as the third argument to `Encode()`.
- Each SEI message is a tuple of `(sei_info_dict, payload_bytes)`.
- Multiple SEI messages can be inserted per frame.
- To verify SEI insertion, decode the output and extract SEI using *SEI Message Decoding*.

2.6.7.4 Sample Applications

See this sample application for a complete implementation:

- encode_sei_msg.py - Demonstrates SEI message insertion during encoding with custom user data

2.6.7.5 API Reference

- Encode() - Encode frame with optional SEI messages
- CreateEncoder() - Create encoder instance

2.7. Segment-Based Transcoding

Extract smaller, meaningful segments from long videos with optimized context management for efficient processing.

2.7.1. Overview

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.

2.7.2. Optimized Segment-Based Transcoding with PyNvVideoCodec

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

- **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.

2.7.3. Creating Video Segments

Extract video segments using PyNvVideoCodec's Transcoder with persistent context management.

PyNvVideoCodec provides the `Transcoder` class for efficient segment-based transcoding. The transcoder maintains persistent decode/encode contexts across segments, eliminating the overhead of repeated initialization.

2.7.3.1 Example

The following example demonstrates segment extraction from a video file:

Input Video → Transcoder → Video Segments

Step 1: Get Video Duration

Use `SimpleDecoder` to get the video metadata for validating segment timestamps:

```
import PyNvVideoCodec as nvc

# Get video duration for validation
decoder = nvc.SimpleDecoder(input_file_path, gpu_id=0)
duration = decoder.get_stream_metadata().duration
print(f"Video duration: {duration:.2f} seconds")
```

Step 2: Load Transcoder Configuration

Define encoding parameters such as codec, preset, tuning, and bitrate:

```
import json

# Load transcoder configuration from JSON file
with open(config_file_path) as json_file:
    config = json.load(json_file)

# Example config structure:
# {
#     "codec": "h264",
#     "preset": "P4",
#     "tuning_info": "high_quality",
#     "bitrate": 5000000
# }
```

Step 3: Create Transcoder and Extract Segment

Create a `Transcoder` instance with input/output paths and configuration, then call `segmented_transcode()` with start and end times:

```
# Define segment boundaries (in seconds)
start_time = 10.0
end_time = 25.0

# Create transcoder and extract segment
transcoder = nvc.Transcoder(
    input_file_path,
    output_file_path,
    gpu_id,
    0,  # cuda_context (0 for default)
    0,  # cuda_stream (0 for default)
    **config
)

# Extract the segment
transcoder.segmented_transcode(start_time, end_time)
print(f"Created segment: {start_time}s - {end_time}s")
```

Step 4: Process Multiple Segments

For multiple segments, create a new transcoder for each output file:

```
# Define multiple segments as (start, end) tuples
segments = [
    (0.0, 10.5),
    (15.0, 30.0),
    (45.5, 60.0)
]

for start_time, end_time in segments:
    # Validate against video duration
    if end_time > duration:
```

(continues on next page)

(continued from previous page)

```
        end_time = duration

    # Generate output path with timestamps
    output_path = f"segment_{start_time}_{end_time}.mp4"

    # Create transcoder and extract segment
    transcoder = nvc.Transcoder(input_file_path, output_path, gpu_id, 0, 0,-
↪**config)
    transcoder.segmented_transcode(start_time, end_time)
    print(f"Created: {output_path}")
```

2.7.3.2 Note

- Segment times are specified in seconds (float values).
- The transcoder automatically seeks to the nearest keyframe before the start time.
- Output files are named with timestamps appended by the API.
- For concatenating segments into a single file, use the same transcoder instance with multiple `segmented_transcode()` calls.

2.7.3.3 APIs Used

The following APIs are used in this example:

- SimpleDecoder() - Get video metadata for duration validation
- get_stream_metadata() - Get video duration and properties
- Transcoder() - Create transcoder with encoding configuration
- segmented_transcode() - Extract a segment by start/end times

2.7.3.4 Sample Applications

See this sample application for a complete implementation:

- create_video_segments.py - Demonstrates extracting multiple segments from a video file with configurable start/end times from a segments file

2.8. 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.

2.8.1. 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

2.8.2. 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.

```
from PyNvVideoCodec import SimpleDecoder, OutputColorType

# Decode with GPU memory enabled
decoder = SimpleDecoder(
    "video.mp4",
    use_device_memory=True,
    output_color_type=OutputColorType.RGBP
)

frame = decoder[0]

# DLPack protocol methods are available on the frame object
device_type, device_id = frame.__dlpack_device__()
print(f"Device: {device_type}, ID: {device_id}")  # Device: 2 (CUDA), ID: 0

# The __dlpack__() method is called automatically by from_dlpack()
# You typically don't call it directly - just use:
# tensor = torch.from_dlpack(frame)
```

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

2.8.3. Integration with PyTorch

PyTorch provides the `torch.from_dlpack()` function to import DLPack tensors directly. The resulting tensor shares the same GPU memory with no data copying.

```
import torch
from PyNvVideoCodec import SimpleDecoder, OutputColorType

# Create decoder with GPU memory and planar RGB output
decoder = SimpleDecoder(
    "video.mp4",
    use_device_memory=True,
    output_color_type=OutputColorType.RGBP  # Planar RGB (CHW format)
)

# Get a decoded frame
frame = decoder[0]

# Convert to PyTorch tensor - zero-copy!
tensor = torch.from_dlpack(frame)
print(f"Tensor shape: {tensor.shape}")  # Output: torch.Size([3, 1080, 1920])
print(f"Tensor device: {tensor.device}")  # Output: cuda:0

# Normalize for model input
normalized = tensor.float() / 255.0
```

The tensor format follows the video pixel format:

- **RGBP (Planar):** Shape is (3, height, width) - preferred for most deep learning models
- **RGB (Interleaved):** Shape is (height, width, 3)
- **NV12 (Native):** Shape depends on the native decoder output format

2.8.4. Batch Processing for Deep Learning

When processing multiple frames for deep learning inference, convert frames to tensors and stack them into a batch:

```
import torch
from PyNvVideoCodec import SimpleDecoder, OutputColorType

# Create decoder with planar RGB output for CNN models
decoder = SimpleDecoder(
    "video.mp4",
    use_device_memory=True,
    output_color_type=OutputColorType.RGBP
)

batch_size = 4

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

(continues on next page)

(continued from previous page)

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

# Stack into batch tensor for inference
batch = torch.stack(tensors)  # Shape: [batch_size, 3, height, width]

# Normalize and prepare for model
batch = batch.float() / 255.0

# Run inference with your model
# output = model(batch)
```

Using ThreadedDecoder for High-Throughput Inference:

```
from PyNvVideoCodec import ThreadedDecoder, OutputColorType

# ThreadedDecoder prefetches frames in background
decoder = ThreadedDecoder(
    enc_file_path="video.mp4",
    buffer_size=12,
    use_device_memory=True,
    output_color_type=OutputColorType.RGBP
)

metadata = decoder.get_stream_metadata()
batch_size = 4

while True:
    # get_batch_frames() returns immediately with prefetched frames
    frames = decoder.get_batch_frames(batch_size)
    if len(frames) == 0:
        break

    # Convert and stack
    batch = torch.stack([torch.from_dlpack(f) for f in frames])
    batch = batch.float() / 255.0

    # Run inference - decoding happens in parallel!
    # output = model(batch)
```

2.8.5. Integration with Other Frameworks

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

TensorFlow Integration:

Use `tf.experimental.dlpack.from_dlpack(frame)` to convert decoded frames to TensorFlow tensors. Refer to the TensorFlow DLPack documentation for details and compatibility information.

CuPy Integration:

```
import cupy as cp
from PyNvVideoCodec import SimpleDecoder, OutputColorType

decoder = SimpleDecoder(
    "video.mp4",
    use_device_memory=True,
    output_color_type=OutputColorType.RGBP
)

frame = decoder[0]

# Convert to CuPy array - zero-copy!
cupy_array = cp.from_dlpack(frame)
print(f"CuPy array shape: {cupy_array.shape}")

# Perform GPU-accelerated operations with CuPy
normalized = cupy_array.astype(cp.float32) / 255.0
```

NumPy Integration (requires copy):

```
import torch
import numpy as np
from PyNvVideoCodec import SimpleDecoder, OutputColorType

decoder = SimpleDecoder(
    "video.mp4",
    use_device_memory=True,
    output_color_type=OutputColorType.RGBP
)

frame = decoder[0]

# First convert to PyTorch, then to NumPy (copies GPU → CPU)
tensor = torch.from_dlpack(frame)
numpy_array = tensor.cpu().numpy()
print(f"NumPy array shape: {numpy_array.shape}")
```

Note

Converting to NumPy requires copying data from GPU to CPU memory, which is slower than zero-copy GPU-to-GPU transfers. For best performance, keep data on the GPU whenever possible.

Chapter 3. Debugging and Logging

3.1. 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.

3.2. 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.

3.3. Example Usage

Linux/macOS: Set with `export LOGGER_LEVEL=DEBUG` before running your script.

Windows (Command Prompt): Set with `set LOGGER_LEVEL=DEBUG` before running your script.

Windows (PowerShell): Set with `$env:LOGGER_LEVEL="DEBUG"` before running your script.

Setting in Python code: Set `os.environ["LOGGER_LEVEL"] = "DEBUG"` before importing PyNvVideoCodec.

Chapter 4. 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.

4.1. Benchmark Overview

The benchmark scripts provided with PyNvVideoCodec measure performance across different use cases. Each benchmark automatically generates test videos using FFmpeg on the first run, and subsequent runs will reuse these videos for consistent testing.

Important Considerations Before Running Benchmarks:

- **Initial run time:** The first execution of any benchmark script takes significantly longer because it generates sample videos using FFmpeg. Subsequent runs are much faster as they reuse the generated videos.
- **Disk space:** The generated test videos are stored locally. Ensure sufficient disk space is available.
- **GPU requirements:** A CUDA-capable NVIDIA GPU with NVDEC hardware decoder support is required.

4.2. Understanding the NVDEC Parameter

Benchmark scripts require an `--nvdecs` parameter, which specifies the number of hardware NVDEC (NVIDIA Video Decoder) instances available on your GPU. This parameter is critical for achieving optimal performance.

How to determine your NVDEC count:

1. Visit the NVIDIA Video Encode and Decode GPU Support Matrix
2. Find your GPU model in the list
3. Look for the “NVDEC” column to see the number of decoder instances

Common NVDEC counts by GPU:

- NVIDIA L40G: 3 NVDECs
- NVIDIA A100: 5 NVDECs
- NVIDIA RTX 4090: 2 NVDECs
- NVIDIA RTX 3090: 1 NVDEC
- NVIDIA T4: 2 NVDECs

Setting the correct NVDEC count allows the benchmark to spawn the appropriate number of threads to fully saturate the available hardware decoders, maximizing throughput.

4.3. Benchmark Dependencies

Before running the benchmark scripts, ensure you have all required Python packages installed. `requirements.txt` file is provided in the benchmark scripts directory.

Install the dependencies using:

```
pip install -r requirements.txt
```

Additional requirements:

- **FFmpeg:** Must be installed and accessible in your system PATH. The benchmarks use FFmpeg (with NVENC support) to generate test videos. We recommend using the BtBN FFmpeg builds which include NVIDIA hardware acceleration support.
- **CUDA Toolkit:** A compatible CUDA toolkit must be installed for PyCUDA.

4.4. Expected Execution Time

The following table provides approximate execution times for each benchmark script. These times were measured on an NVIDIA L40G GPU with 3 threads (matching the 3 NVDECs available).

Table 4.1: Benchmark Execution Time Estimates (L40G, 3 Threads)

Benchmark Script	Execution Time	Notes
`frame_sampling_benchmark.py`	~1 minute	Tests 1080p videos with different GOP sizes
`cached_decoder_benchmark.py`	~42 minutes	Tests multiple resolutions (360p to 4K) with 500 iterations each
`segmented_transcode_benchı py`	~6 minutes	Generates and processes video segments

Actual execution times will vary depending on your GPU model, CPU, storage speed, and the number of threads used.

4.5. Available Benchmarks

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

4.6. Frame Retrieval

Performance benchmarks for different frame retrieval patterns using PyNvVideoCodec decoder.

4.6.1. Objective

This benchmark measures the sampling performance of PyNvVideoCodec when retrieving frames using different access patterns. It evaluates how efficiently frames can be extracted from a video depending on whether you need sequential, uniformly distributed, or randomly selected frames.

What this benchmark measures:

- Frame retrieval throughput (Frames Per Second) for three sampling patterns
- Impact of GOP (Group of Pictures) size on seek performance
- Efficiency of direct frame sampling versus sequential decoding
- Multi-threaded scaling performance across available NVDECs

Sampling Patterns Tested:

- **Sequential Decoding:** Retrieves frames in order from the start of the video (e.g., first 100 frames). This is the fastest pattern as it requires minimal seeking.
- **Uniform Sampling:** Retrieves frames at regular intervals across the entire video duration. For example, sampling 30 frames from a 30-second video fetches one frame every second.
- **Random Sampling:** Retrieves frames at randomly selected positions throughout the video. This pattern represents the most challenging access pattern due to unpredictable seek locations. The script uses `torch.randperm()` to generate unique random frame indices, ensuring no duplicate frames are sampled.

Key Performance Indicators (KPI):

- **FPS (Frames Per Second):** The number of frames retrieved per second. Higher is better.
- **Efficiency:** Ratio comparing direct sampling performance to sequential decode-then-sample approach. Values greater than 1.0x indicate direct sampling is faster than decoding all frames and then selecting the needed ones.

4.6.2. How the Benchmark Works

The benchmark follows these steps:

1. **Video Generation (first run only):** Creates test videos using FFmpeg with the mandelbrot pattern at 1080p resolution. Multiple videos with different GOP sizes (default: 30 and 250) are generated to test the impact of GOP on seek performance.

2. **Thread Setup:** Creates multiple decoder threads (1 thread for single-threaded test, N threads to match NVDEC count).
3. **Sequential Decode Test:** Each thread decodes the first N frames (default: 100) sequentially and measures FPS.
4. **Uniform Sampling Test:** Each thread samples M frames (default: 30) at regular intervals and measures FPS. The efficiency is calculated by comparing against the time needed to sequentially decode up to the last sampled frame.
5. **Random Sampling Test:** Each thread samples M frames at random positions and measures FPS, also calculating efficiency.
6. **Results Aggregation:** FPS and efficiency metrics are calculated and displayed for all configurations.

4.6.3. Running the Benchmark

Basic Usage:

```
python frame_sampling_benchmark.py --nvdecs 3
```

Replace 3 with the number of NVDEC instances on your GPU. See the *NVDEC Parameter* section to determine your GPU's NVDEC count.

Command Line Options:

Option	Default	Description
`--nvdecs`	(required)	Number of NVDEC instances on your GPU. Determines the number of parallel decoder threads.
`--resolution, -res`	1920x1080	Video resolution for generated test videos
`--gop, -g`	30 250	GOP sizes to test (space-separated list)
`--duration, -d`	30	Video duration in seconds
`--fps, -f`	30	Video frames per second
`--num-seq-frames, -seq`	100	Number of frames to decode for sequential test
`--num-samp-frames, -samp`	30	Number of frames to sample for uniform/random tests
`--verbose, -v`	False	Show detailed per-thread performance information

Example Commands:

```
# Run benchmark with default settings on a GPU with 3 NVDECs
python frame_sampling_benchmark.py --nvdecs 3

# Run with 720p resolution and specific GOP sizes
python frame_sampling_benchmark.py --nvdecs 3 --resolution 1280x720 --gop 30-
↪60 120
```

(continues on next page)

(continued from previous page)

```
# Run with verbose output showing per-thread details
python frame_sampling_benchmark.py --nvdecs 3 --verbose

# Run with custom sampling parameters
python frame_sampling_benchmark.py --nvdecs 2 --num-seq-frames 200 --num-samp-
↪frames 50
```

Output Files:

- `benchmark_results.json` - Detailed results including system info and per-test metrics
- `benchmark_videos/` - Generated test videos (reused in subsequent runs)

Expected Execution Time: Approximately 1 minute on an L40G GPU with 3 threads.

4.6.4. Benchmark Environment

Environment:

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

4.6.5. 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

4.6.6. Benchmark Results

The following results were obtained on the representative platform described in the Benchmark Environment section above. You can run the benchmark script in your own environment to obtain results specific to your hardware configuration.

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 4.2: Sequential Decode Performance

Video Config	Num Threads	FPS
1920x1080 250gop 30s	1	886
1920x1080 250gop 30s	3	2615.4
1920x1080 30gop 30s	1	881.1
1920x1080 30gop 30s	3	2609.4

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 4.3: Random Sampling Performance

Video Config	Num Threads	FPS	Efficiency
1920x1080 250gop 30s	1	37.3	1.02x
1920x1080 250gop 30s	3	110.8	1.03x
1920x1080 30gop 30s	1	78.4	2.14x
1920x1080 30gop 30s	3	218	1.98x

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 4.4: Uniform Sampling Performance

Video Config	Num Threads	FPS	Efficiency
1920x1080 250gop 30s	1	39.6	1.05x
1920x1080 250gop 30s	3	117.6	1.05x
1920x1080 30gop 30s	1	54.2	1.44x
1920x1080 30gop 30s	3	158.4	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 up to the last required frame, then extracting the needed 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.

Important: Efficiency should only be compared within the same thread configuration. Do not compare efficiency values across different thread counts. For example, while 1-thread random sampling shows 2.14x efficiency and 3-thread shows 1.98x efficiency, this does not mean single-threaded is better. The

3-thread configuration achieves 218 FPS compared to 78.4 FPS for single-thread-a 2.8x improvement in absolute throughput. The efficiency metric only indicates how much faster direct sampling is versus sequential decoding within that same thread configuration.

4.6.7. Key Observations

- GOP size has significant impact on frame retrieval performance:
  - For random sampling, smaller GOP size (30) increases performance by 110% 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 provides significant absolute performance gains:
  - Sequential decoding: 881 FPS (1 thread) → 2609 FPS (3 threads) = 2.96x speedup
  - Random sampling (30 GOP): 78.4 FPS (1 thread) → 218 FPS (3 threads) = 2.78x speedup
  - Uniform sampling (30 GOP): 54.2 FPS (1 thread) → 158.4 FPS (3 threads) = 2.92x speedup
- Efficiency comparison (within same thread configuration):
  - Smaller GOP (30) provides higher efficiency for both sampling methods because less data needs to be decoded to reach each target frame
  - Random sampling with 30 GOP: 2.14x efficiency (1 thread), 1.98x efficiency (3 threads)
  - Uniform sampling with 30 GOP: 1.44x efficiency (1 thread), 1.42x efficiency (3 threads)
  - Larger GOP (250) shows minimal efficiency advantage (1.02x-1.05x) because more frames must be decoded to reach seek points

4.7. Decoder Reuse

Performance benefits of reusing decoder instances when processing multiple videos.

4.7.1. Objective

This benchmark measures and compares the performance of NVIDIA's video decoder in two operational modes:

1. **Simple Decoder:** Creates a new decoder instance for each video file
2. **Cached Decoder:** Reuses the same decoder instance across multiple video files through reconfiguration

What this benchmark measures:

- Decoding throughput (Frames Per Second) for both decoder modes
- Total time taken to decode a batch of video clips
- Performance comparison across different video resolutions (360p, 480p, 720p, 1080p, 4K)
- Impact of decoder initialization overhead on overall performance

Key Performance Indicator (KPI): The primary metric is FPS (Frames Per Second). Higher FPS indicates better decoder efficiency. The speedup ratio (Cached FPS / Simple FPS) shows the benefit of decoder caching.

4.7.2. How the Benchmark Works

The benchmark follows these steps:

1. **Video Generation (first run only):** Creates test videos using FFmpeg with the mandelbrot test pattern at various resolutions (360p, 480p, 720p, 1080p, 4K). Each video is 2 seconds long at 30 fps.
2. **Workload Creation:** Each generated video is queued 500 times to create sufficient workload to saturate the GPU's NVDEC hardware.
3. **Thread Distribution:** Videos are distributed across multiple decoder threads (1 thread for single-threaded test, N threads to match NVDEC count).
4. **Simple Decoder Test:** Each thread creates a new decoder instance for every video clip and measures total decoding time.
5. **Cached Decoder Test:** Each thread creates a single decoder instance with caching enabled and reconfigures it for each subsequent video, measuring total decoding time.
6. **Results Comparison:** FPS is calculated for both modes and compared across all resolutions.

4.7.3. Running the Benchmark

Basic Usage:

```
python cached_decoder_benchmark.py --nvdecs 3
```

Replace 3 with the number of NVDEC instances on your GPU. See the *NVDEC Parameter* section to determine your GPU's NVDEC count.

Command Line Options:

Option	Default	Description
`--nvdecs`	(required)	Number of NVDEC instances on your GPU. This determines the number of parallel decoder threads.
`--codec`	h264	Video codec to use: h264, hevc, or av1
`--fps`	30	Frame rate for generated test videos
`--gop`	60	GOP (Group of Pictures) size for generated videos
`--plot-only`	False	Skip benchmark and only generate plots from existing JSON results. The JSON files from the existing runs are stored in the same directory as the benchmark script.

Example Commands:

```
# Run benchmark with H.264 codec on a GPU with 3 NVDECs
python cached_decoder_benchmark.py --nvdecs 3

# Run benchmark with HEVC codec on a GPU with 2 NVDECs
python cached_decoder_benchmark.py --nvdecs 2 --codec hevc

# Run benchmark with custom video settings
python cached_decoder_benchmark.py --nvdecs 4 --codec av1 --fps 60 --gop 120

# Only generate plots from existing results
python cached_decoder_benchmark.py --nvdecs 3 --plot-only
```

Output Files:

- `cached_decoder_performance_{codec}_{threads}_threads.json` - Detailed results in JSON format
- `cached_decoder_performance_{codec}_{threads}_threads.png` - Performance comparison bar graphs
- `test_videos_{codec}/` - Generated test videos (reused in subsequent runs)

Expected Execution Time: Approximately 42 minutes on an L40G GPU with 3 threads. This benchmark takes longer because it tests multiple resolutions (360p to 4K) with 500 iterations each to ensure statistically significant results.

4.7.4. Benchmark Environment

Environment:

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

4.7.5. 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 (short) and 30 seconds (long)
  - Pattern: mandelbrot
- 5 videos created using FFmpeg (1 video per resolution)
- Each video was reused 500 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

4.7.6. Benchmark Results

The following results were obtained on the representative platform described in the Benchmark Environment section above. You can run the benchmark script in your own environment to obtain results specific to your hardware configuration.

Short Duration Videos (2 seconds)

Performance comparison when decoding many short video clips, where decoder initialization overhead is most significant.

Table 4.5: Decoder Reuse Performance - Short Videos (2 sec)

Resolution	Decoder Type	Time Taken (s)	FPS
360p	Simple	17.05	1760
360p	Cached	2.37	12679
480p	Simple	17.27	1737
480p	Cached	3.28	9151
720p	Simple	18.55	1617
720p	Cached	5.78	5190
1080p	Simple	20.53	1461
1080p	Cached	11.53	2602
4k	Simple	53.28	563
4k	Cached	42.78	701

Long Duration Videos (30 seconds)

Performance comparison when decoding longer video clips, where actual decoding time dominates over initialization overhead.

Table 4.6: Decoder Reuse Performance - Long Videos (30 sec)

Resolution	Decoder Type	Time Taken (s)	FPS
360p	Simple	39.28	11456
360p	Cached	30.78	14621
480p	Simple	54.28	8291
480p	Cached	44.78	10049
720p	Simple	94.03	4786
720p	Cached	84.28	5339
1080p	Simple	188.28	2390
1080p	Cached	177.78	2531
4k	Simple	709.78	634
4k	Cached	695.53	647

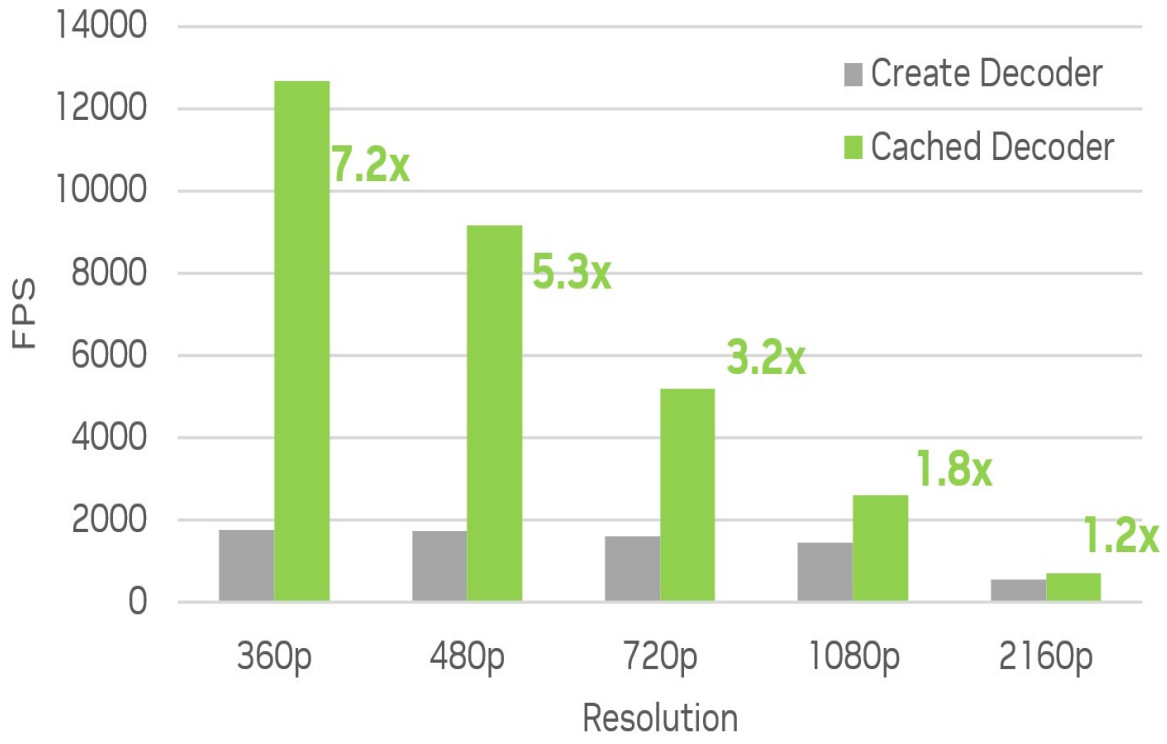

Figure 4.1: Performance Comparison: Simple vs. Cached Decoders Bar chart comparing performance of simple decoder creation vs. cached decoder approach across resolutions for short duration videos

Key Observations:

- Cached decoders consistently outperform simple decoders across all resolutions and video durations

- For short videos (2 sec), performance improvement is dramatic at lower resolutions:
  - 360p: 7.2x faster (12679 vs 1760 FPS)
  - 480p: 5.3x faster (9151 vs 1737 FPS)
  - 720p: 3.2x faster (5190 vs 1617 FPS)
  - 1080p: 1.8x faster (2602 vs 1461 FPS)
  - 4K: 1.2x faster (701 vs 563 FPS)
- For long videos (30 sec), the improvement is more modest as decoding time dominates:
  - 360p: 1.3x faster
  - 480p: 1.2x faster
  - 720p-4K: 1.02x-1.1x faster
- The performance benefit comes from eliminating decoder initialization overhead, which is most significant when processing many short video clips

4.8. Segmented Transcoding

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

4.8.1. Objective

This benchmark compares the performance of different approaches for transcoding video segments. It measures how efficiently PyNvVideoCodec's Transcoder class handles segmented video transcoding compared to traditional FFmpeg-based methods.

What this benchmark measures:

- Transcoding throughput (Frames Per Second) for each method
- Total processing time for a batch of video segments
- Performance difference between PyNvVideoCodec and FFmpeg approaches
- Impact of different FFmpeg configurations (with/without filter_complex, audio handling)

Transcoding Methods Compared:

- **Mode 0 - PyNvVideoCodec Transcoding:** Uses PyNvVideoCodec's Transcoder class with `segmented_transcode()` method. Maintains persistent GPU context and avoids repeated encoder/decoder initialization.
- **Mode 1 - FFmpeg Without Map:** Uses separate FFmpeg commands for each segment with `-ss`/`-to` for time ranges. Simple approach but spawns multiple processes.
- **Mode 2 - FFmpeg With Map (No Audio):** Uses FFmpeg's filter_complex to process multiple segments in one command. Video only processing.
- **Mode 3 - FFmpeg With Map (With Audio):** Same as Mode 2 but includes audio stream processing.

Key Performance Indicator (KPI): The primary metric is FPS (Frames Per Second) representing transcoding throughput. Higher FPS indicates faster processing. The speedup ratio (PyNvVideoCodec FPS / FFmpeg FPS) shows the performance advantage of using PyNvVideoCodec.

4.8.2. How the Benchmark Works

The benchmark follows these steps:

1. **Video Generation (first run only):** Creates a test video using FFmpeg with the mandelbrot pattern and audio. The video includes both H.264 video and AAC audio tracks. A short base clip is generated and then looped to reach the target duration.
2. **Segment Creation:** Generates random non-overlapping segments within the video. Each segment has a configurable minimum duration (default: 5 seconds).
3. **PyNvVideoCodec Transcoding Test:** Uses PyNvVideoCodec's Transcoder class to transcode each segment. The decoder and encoder contexts are maintained across segments, avoiding repeated initialization.
4. **FFmpeg Transcoding Tests:** Runs the same segments through different FFmpeg configurations (Modes 1-3) for comparison.
5. **Results Comparison:** Calculates FPS for each method and generates a comparison report.
6. **Logging:** Saves detailed execution logs in JSON format for reproducibility and replay.

4.8.3. Running the Benchmark

Basic Usage:

```
python segmented_transcode_benchmark.py
```

This runs the benchmark with default settings (1920x1080, 5 seconds, 10 segments, all 4 transcoding modes).

Command Line Options:

Option	Default	Description
`-W, --width`	1920	Video width in pixels
`-H, --height`	1080	Video height in pixels
`-d, --duration`	5400	Video duration in seconds
`-fps, --fps`	30	Frames per second
`-s, --segments`	10	Number of random segments to transcode
`--segment-duration`	5	Segment duration in seconds
`-u, --usage`	0 1 2 3	Transcoding modes to benchmark (space-separated list)
`-ic, --input-codec`	h264	Input codec: h264, hevc, or av1
`-c, --codec`	h264	Output codec: h264, hevc, or av1
`-p, --preset`	P1	Encoder preset (P1-P7)
`-n, --numthreads`	1	Number of concurrent threads
`--gop-size`	250	GOP size for encoding
`-g, --gpuid`	0	GPU device ID
`-i, --input`	(none)	Use existing video file instead of generating
`--log`	(auto)	Path to save execution log
`--replay`	(none)	Replay transcoding from a previous log file

Example Commands:

```
# Run full benchmark with default settings (all 4 modes)
python segmented_transcode_benchmark.py

# Compare only PyNvVideoCodec vs basic FFmpeg
python segmented_transcode_benchmark.py -u 0 1

# Test only PyNvVideoCodec transcoding
python segmented_transcode_benchmark.py -u 0

# Custom video parameters with 10 segments
python segmented_transcode_benchmark.py -W 1920 -H 1080 -d 30 -s 10

# Use HEVC codec with 2 B-frames
python segmented_transcode_benchmark.py -ic hevc -c hevc -bf 2

# Use an existing video file
python segmented_transcode_benchmark.py -i /path/to/video.mp4

# Replay a previous benchmark run
python segmented_transcode_benchmark.py --replay logs/run_20240615_123045.json
```

Output Files:

- `logs/run_{timestamp}.json` - Detailed execution log
- `pynvc_out/` - Transcoded segments from PyNvVideoCodec
- `ffmpeg_out/` - Transcoded segments from FFmpeg Mode 1
- `ffmpeg_fc_out/` - Transcoded segments from FFmpeg Modes 2 and 3
- `source_videos/` - Generated source videos (reused in subsequent runs)

Expected Execution Time: Approximately 6 minutes on an L40G GPU with 3 threads.

4.8.4. Benchmark Environment

Environment:

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

4.8.5. Methodology

- Script to execute benchmark: `segmented_transcode_benchmark.py`
- Dataset details:
  - Resolution: 1920x1080
  - Codec: H.264
  - Duration: 5400 seconds
  - Number of segments: 10
  - GOP Size: 250
  - Segment duration: 5 seconds
- Transcoding parameters:
  - Output FPS: 30
  - Output B Frames: 0
  - Output Preset: P1
- Benchmarks examine performance of different transcoding methods

Transcoding Methods:

- **PyNvVideoCodec 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

4.8.6. Benchmark Results

The following results were obtained on the representative platform described in the Benchmark Environment section above. You can run the benchmark script in your own environment to obtain results specific to your hardware configuration.

Table 4.7: Segmented Transcoding Performance - H.264 1080p

Method	Time (s)	Throughput (FPS)
PyNvVideoCodec transcoding	2.31	1072.34
FFmpeg without map	6.36	389.17

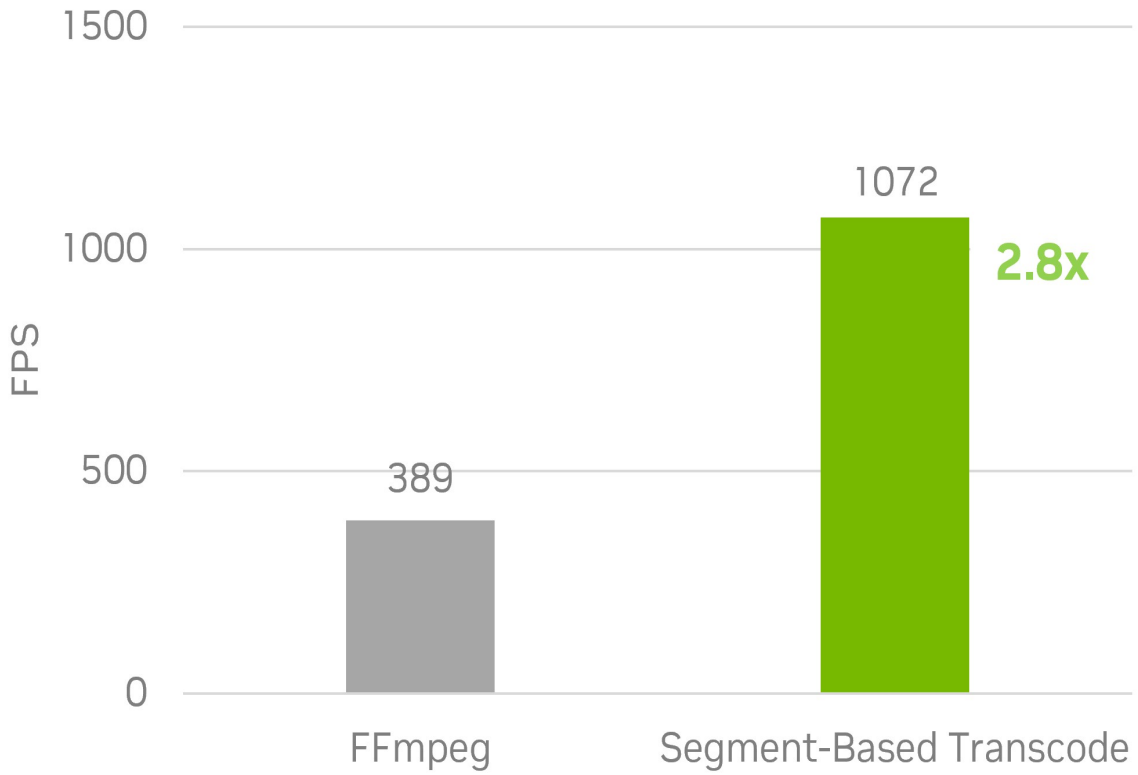

Figure 4.2: Performance Comparison: FFmpeg vs. PyNvVideoCodec Segment-Based Transcoding Bar chart comparing transcoding performance between standard FFmpeg approach and PyNvVideoCodec's segment-based transcoding for H.264 1080p content, showing a 2.8x performance improvement

4.8.7. Key Observations

- PyNvVideoCodec transcoding significantly outperforms FFmpeg's standard transcoding method
- For 1080p content, PyNvVideoCodec transcoding (1072 FPS) is approximately 2.8x faster than FFmpeg without map (389 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

Copyright