

NVIDIA PyNvVideoCodec Read Me

Release 2.2.3

NVIDIA Corporation

Sep 15, 2026

Contents

1 Release Notes **1**
- 1.1 Key Features and Enhancements . . . 1
  - 1.1.1 v 2.2.3 (Latest) . . . 1
  - 1.1.2 v 2.2 . . . 1
- 1.2 Deprecation Notices . . . 2
- 1.3 Limitations and Known Issues . . . 2
- 1.4 Package Contents . . . 3

2 System Requirements **5**
- 2.1 Windows (x64 and ARM64), Linux, Windows Subsystem for Linux (WSL), Jetson Thor, and DGX Spark . . . 5
- 2.2 Additional Configuration for WSL . . . 7
- 2.3 Additional Configuration for Jetson Thor . . . 7
- 2.4 Additional Configuration for DGX Spark . . . 7

3 Installing PyNvVideoCodec Python Module **8**
- 3.1 Installing from PyPI . . . 8
- 3.2 Building and Installing from Source on NVIDIA NGC . . . 8

4 Running Sample Applications **10**
- 4.1 Prerequisites . . . 10
- 4.2 Sample Applications Overview . . . 11
- 4.3 Basic Sample Applications . . . 13
  - 4.3.1 simple_decode_sampling.py . . . 13
  - 4.3.2 encode.py . . . 13
  - 4.3.3 create_video_segments.py . . . 14
- 4.4 Jupyter Notebooks . . . 15
  - 4.4.1 Setting Up Jupyter Notebooks . . . 15
  - 4.4.2 simple_decode_tutorial.ipynb . . . 15
  - 4.4.3 object_detection_tutorial.ipynb . . . 16
  - 4.4.4 face_protect_tutorial.ipynb . . . 16
- 4.5 Advanced Sample Applications . . . 17
  - 4.5.1 decode.py . . . 17
  - 4.5.2 decode_with_cuda_control.py . . . 17
  - 4.5.3 decode_from_memory_buffer.py . . . 18
  - 4.5.4 decode_with_low_latency.py . . . 18
  - 4.5.5 decode_perf.py . . . 19
  - 4.5.6 decode_sei_msg.py . . . 20
  - 4.5.7 simple_decode_stats.py . . . 20
  - 4.5.8 simple_decode_from_memory.py . . . 21
  - 4.5.9 decode_reconfigure.py . . . 22
  - 4.5.10 encode_reconfigure.py . . . 22
  - 4.5.11 encode_sei_msg.py . . . 23

4.5.12 encode_perf.py 23
4.5.13 transcode.py 24

Chapter 1. Release Notes

1.1. Key Features and Enhancements

1.1.1. v 2.2.3 (Latest)

- **Windows on ARM64 support (NVIDIA RTX Spark):** Brings native, hardware-accelerated Python video APIs to Windows on ARM64, enabling high-performance video and AI pipelines on NVIDIA RTX Spark.

1.1.2. v 2.2

- **NVIDIA Jetson support:** Brings easy-to-use, GPU-accelerated Python video APIs to Jetson, making it easy to build video and AI pipelines for robotics and other edge devices.
- **NVIDIA DGX Spark support:** Unlocks high-throughput, Python-based video decode and encode on DGX Spark, speeding up video curation, experimentation, prototyping, training, and deployment of video and AI pipelines
- **SimpleDecoder in-memory input support:** `SimpleDecoder` now accepts video data directly from `bytes`, `bytearray`, `memoryview`, or any seekable `BinaryIO` stream, such as `io.BytesIO`, an open file handle, a tar archive member, or an S3/HTTP range reader. This enables decoding from network buffers or object-storage streams without writing to a temporary file, and all input types support the full seekable API including random frame access and batch retrieval.
- **New samples:**
  - **Transcode:** The new `transcode.py` sample demonstrates end-to-end GPU transcoding with demuxing, NVDEC decode, NVENC encode, muxing, and audio passthrough. The pipeline preserves source A/V interleave and bit-exact PTS, making it a reference implementation for high-throughput format conversion and codec migration workflows.
  - **Encode with muxing:** The `encode.py` sample now supports muxing encoded video directly into container formats (MP4, MOV). Muxing is triggered automatically by the output file extension, so no separate muxing step is required. Supported combinations are H.264/HEVC/AV1 into MP4 and H.264/HEVC into MOV.
  - **SimpleDecoder from memory:** The new `simple_decode_from_memory.py` sample demonstrates decoding from `bytes`, `bytearray`, `memoryview`, and seekable `BinaryIO`.
  - **Face protection:** The `face_protect_tutorial.ipynb` notebook demonstrates face detection with a TensorRT-optimized RetinaFace model and applies GPU-accelerated Gaussian blur to protect detected face regions.

1.2. Deprecation Notices

The following features and methods are deprecated in this release and will be removed in future versions:

- **nvcv_image() method:** The `nvcv_image()` method in the `DecodedFrame` class is deprecated and will be removed in a future version. This method was originally designed as a workaround for CV-CUDA tensor representation but is no longer the recommended approach.

Users are encouraged to migrate to alternative methods for CV-CUDA tensor conversion. The deprecated method will continue to function in this release but will emit deprecation warnings.

Important

Deprecated features may be removed without further notice in major version updates. Please update your code to use supported alternatives.

1.3. Limitations and Known Issues

- PyNvVideoCodec uses the FFmpeg binaries for demuxing and muxing of audio and video content.

  NVIDIA will not update the FFmpeg binaries included in our release package as these binaries are available, maintained and updated by the FFmpeg open-source community.

  Attention

  NVIDIA does not provide support for FFMPEG; therefore, it is the responsibility of end users and developers, to stay informed about any vulnerabilities or quality bugs reported against FFMPEG. Users are encouraged to refer to the official FFmpeg website and community forums for the latest updates, patches, and support related to FFmpeg binaries and act as they deem necessary.

- **WebM Container Seeking Limitations:** WebM containers may experience reduced seek accuracy due to codec-specific behavior of VP8/VP9 streams.

  During decode, some frames in VP8/VP9 (commonly used in WebM containers) are marked as non-displayable, causing discrepancies between the reported total frame count from container metadata and the actual displayable frame count. This can result in frame count mismatches and potential seeking issues near the end of video streams.

  PyNvVideoCodec implements workarounds to handle these discrepancies, including special packet filtering and frame count adjustments for WebM containers. However, users should be aware that seek operations may be less precise compared to other container formats like MP4.

  Note

  Similar limitations may also affect FLV container that use VP8/VP9 codecs.

- **SimpleDecoder in-memory input limitations:**

- **reconfigure_decoder() accepts file paths only.** `reconfigure_decoder()` cannot retarget a decoder at a `bytes` or `BinaryIO` source. Callers that need to process successive in-memory clips must construct a new `SimpleDecoder` per clip rather than reusing a pooled instance.
- **Scanned stream metadata unsupported for BinaryIO streams.** Passing `need_scanned_stream_metadata=True` and calling `get_scanned_stream_metadata()` raises an error when the source is a `BinaryIO` stream, because the background scan thread cannot safely share a Python stream object. This option works normally for file-path and `bytes` inputs.
- **Containers without a header frame count rejected for BinaryIO streams.** Formats such as FLV and WebM that do not store a frame count in their headers are not supported as `BinaryIO` inputs, because determining the frame count requires a full sequential scan. File-path and `bytes` inputs are not affected.

1.4. Package Contents

This package contains the following:

1. Sample applications demonstrating usage of PyNvVideoCodec APIs for encoding, decoding and transcoding use cases.
   - [.\samples\basic] - Basic sample applications
   - [.\samples\advanced] - Advanced sample applications
2. Jupyter notebooks demonstrating usage of PyNvVideoCodec APIs.
   - [.\samples\jupyter]
3. Requirements files and platform setup scripts specifying dependencies.
   - [.\samples\requirements.txt] - Required libraries to run the sample applications and Jupyter notebooks
   - [.\benchmarks\requirements.txt] - Required libraries to run the benchmark scripts
   - [.\samples\setup_pycuda_win_arm64.bat] - Setup script for PyCUDA on Windows on ARM64 (NVIDIA RTX Spark) sample workflows
   - [.\benchmarks\setup_pycuda_win_arm64.bat] - Same PyCUDA setup script for Windows on ARM64 (NVIDIA RTX Spark) benchmark workflows
4. Python Bindings
   - [.src\PyNvVideoCodec]
5. Video codec helper classes and utilities
   - [.src\VideoCodecSDKUtils]
6. FFmpeg libraries and source code
   - [.external\ffmpeg]
7. Documents
   - [.docs]
8. Benchmarks contains performance benchmarking scripts for testing various PyNvVideoCodec features including segmented transcoding, decoder caching, and frame sampling capabilities.

- [.\benchmarks\]

The sample applications provided in the package are for demonstration purposes only and may not be fully tuned for quality and performance. Hence the users are advised to do their independent evaluation for quality and/or performance.

Chapter 2. System Requirements

2.1. Windows (x64 and ARM64), Linux, Windows Subsystem for Linux (WSL), Jetson Thor, and DGX Spark

PyNvVideoCodec is supported on Windows x64, Windows on ARM64 (NVIDIA RTX Spark), Linux, Windows Subsystem for Linux (WSL), NVIDIA Jetson Thor platforms, and NVIDIA DGX Spark. The following requirements apply to all supported platforms:

Table 2.1: System Requirements

Operating System	Windows 10 or higher (x64) Windows on ARM64 (NVIDIA RTX Spark) Ubuntu 20.04 or higher (Linux x86_64 and aarch64) Ubuntu 20.04 or higher for WSL Jetson Linux 39.2 or higher (Jetson Thor, Linux arm64) NVIDIA DGX OS (DGX Spark, Linux arm64)
GPU	**Discrete GPU platforms:** Turing, Ampere, Ada, Hopper, Blackwell **Supported GPU Platforms:** NVIDIA Jetson Thor, NVIDIA DGX Spark, NVIDIA RTX Spark
Drivers	NVIDIA Windows display driver 531.61 or newer; Get most recent NVIDIA Display Driver **Windows on ARM64 (NVIDIA RTX Spark):** RTX Spark Developer Preview **Jetson Thor:** Jetson Linux 39.2 or higher **DGX Spark:** NVIDIA DGX OS with the NVIDIA driver and CUDA stack supplied by the NVIDIA DGX Spark system image
Python	**Windows x64 wheels:** Python 3.10 to 3.14 **Windows ARM64 wheels:** Python 3.11 to 3.14 **Linux x86_64 wheels:** Python 3.10 to 3.14, built on a manylinux base image with glibc 2.28 and GCC 11.x; requires glibc 2.28 or newer at runtime **Linux arm64/aarch64 wheels:** Python 3.10–3.12 (glibc 2.28, GCC 11.x; requires glibc 2.28 or newer at runtime); Python 3.13–3.14 (glibc 2.34; requires glibc 2.34 or newer at runtime) **Building from source on Ubuntu:** Python development headers for the selected version are required, for example `python3.10-dev`, `python3.11-dev`, `python3.12-dev`, `python3.13-dev`, or `python3.14-dev`.
CMake	3.21 and onwards
Visual Studio (Windows only)	Visual Studio
CUDA Toolkit	**Discrete GPU platforms:** Latest CUDA Toolkit; **Windows on ARM64 (NVIDIA RTX Spark):** CUDA Toolkit 13.4 Developer Preview; **Jetson Thor and DGX Spark:** CUDA 13.0 or higher from the platform software stack
Python modules to run Sample applications	PyCUDA and PyTorch. **Windows ARM64:** PyCUDA must be set up explicitly by running `samples\setup_pycuda_win_arm64.bat` or the identical `benchmarks\setup_pycuda_win_arm64.bat`.

2.2. Additional Configuration for WSL

In addition to all the requirements listed above, WSL users need the following configuration:

- Add the directory `/usr/lib/wsl/lib` to the PATH environment variable if it is not added by default. This is required to ensure the WSL libraries are included in the system path.

2.3. Additional Configuration for Jetson Thor

In addition to all the common Linux requirements listed above, Jetson Thor users need the following configuration:

- Use a Jetson Linux release that is compatible with the PyNvVideoCodec package and the NVIDIA Video Codec SDK release used by the package.
- Install CUDA from the Jetson Linux or JetPack package repositories. If CUDA headers and development libraries are required, install the Jetson platform CUDA development packages, such as `nvidia-cuda-dev`.
- Use Linux arm64 Python wheels and Python dependencies that match the Jetson Linux CUDA stack.

2.4. Additional Configuration for DGX Spark

In addition to all the common Linux requirements listed above, DGX Spark users need the following configuration:

- Use NVIDIA DGX OS and keep the system image updated to receive the recommended NVIDIA driver, CUDA, and library stack for DGX Spark.
- Use Linux arm64 Python wheels and Python dependencies that match the DGX OS CUDA stack.

Chapter 3. Installing PyNvVideoCodec Python Module

Attention

PyNvVideoCodec will download and install additional third-party open source software projects - DLPack. Review the license terms of these open source projects before use.

The Python module can be installed using following ways.

3.1. Installing from PyPI

1. The ready-to-use Python WHL's (Wheel) of the PyNvVideoCodec for supported Windows x64, Windows on ARM64 (NVIDIA RTX Spark), Linux, WSL (Windows Subsystem for Linux), Jetson Thor, and DGX Spark platforms are hosted on PyPI.
2. Open the bash/shell prompt and run:

```
pip install PyNvVideoCodec
```

This is the recommended way to install PyNvVideoCodec.

Upon installation of the wheel, the sample applications and benchmark scripts are placed in the Python site-packages directory. The specific location of site-packages may vary depending on the operating system and Python environment. The path can be identified by running:

```
python -c "import site; print(site.getsitepackages())"
```

3.2. Building and Installing from Source on NVIDIA NGC

The package containing PyNvVideCodec Python module's source code, all dependencies, Python sample applications, and documents is hosted on NVIDIA NGC.

Follow these steps:

1. Download the zip file of the latest package from NVIDIA NGC .

2. Open the bash/shell prompt from the same directory where zip was downloaded and run the following command, replacing “PyNvVideoCodec.zip” with the actual name of the downloaded zip file:

```
pip install "PyNvVideoCodec.zip"
```

3. You can access documents and Python sample applications from the package.

Use this method if you need any customization on PyNvVideoCodec Python module e.g. enabling NVTX markers for profiling.

Follow these steps to build customized version:

1. Unzip the source package to a directory.
2. Do the necessary modifications to the source.
3. On the same directory where `setup.py` is located, run the following commands:

```
pip install .
```

Chapter 4. Running Sample Applications

PyNvVideoCodec includes a comprehensive set of sample applications and Jupyter notebooks that demonstrate the API usage across various use cases. These samples are organized into three categories:

- **Basic Sample Applications** [.\samples\basic]: Demonstrate fundamental encoding, decoding, and transcoding workflows suitable for getting started with PyNvVideoCodec.
- **Advanced Sample Applications** [.\samples\advanced]: Showcase advanced features such as low-latency decoding, SEI message handling, performance measurement, and CUDA resource management.
- **Jupyter Notebooks** [.\samples\jupyter]: Provide interactive tutorials that guide users through the API with step-by-step explanations and live code execution.

4.1. Prerequisites

Ensure you have the following prerequisites before running the samples:

- NVIDIA GPU with appropriate drivers installed
- PyNvVideoCodec module is installed
- Required dependencies installed by running the following command on shell prompt:

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

- For Windows on ARM64 (NVIDIA RTX Spark), PyCUDA must be set up explicitly before running samples that use PyCUDA because PyCUDA is not yet supported through the standard dependency install on Windows on ARM64. Run the provided batch script from a native 64-bit CPython ARM64 environment.

```
samples\setup_pycuda_win_arm64.bat
```

  The same script is also provided at `benchmarks\setup_pycuda_win_arm64.bat` for benchmark workflows.

4.2. Sample Applications Overview

The following table provides an overview of all sample applications and their objectives:

Type	Application Name	Objective
Basic	simple_decode_sampling.py	Decode multiple video files and sample frames at regular intervals. Converts decoded frames to PyTorch tensors for use in machine learning workflows.
	encode.py	Encode raw video frames into a compressed video file. Supports reading input from both CPU memory and GPU memory, with optional muxing to container formats (MP4, MOV) triggered automatically by the output file extension.
	create_video_segments.py	Extract specific portions of a video based on start and end timestamps. Useful for creating training clips or trimming videos.
Jupyter	simple_decode_tutorial.ipynb	Step-by-step tutorial for video decoding. Learn how to access frames by index, retrieve video metadata, and reuse the decoder for multiple files.
	object_detection_tutorial.ipynb	Run object detection on video frames using a pre-trained deep learning model. Uses threaded decoding to run video decoding and AI inference in parallel.
	face_protect_tutorial.ipynb	Protect faces in video by detecting them with a TensorRT-optimized RetinaFace model and applying Gaussian blur. Uses threaded decoding to run video decoding and AI inference in parallel.
Advanced	decode.py	Reads a video file using low level core decoder interface and outputs raw decoded frame
	decode_with_cuda_control.py	Decode video with explicit control over CUDA resources. Useful when integrating with other CUDA-based libraries that require specific context or stream management.
	decode_from_memory_buffer.py	Decode video data directly from memory instead of reading from a file. Useful for processing video received over a network or stored in memory.
	decode_with_low_latency.py	Decode video with minimal delay between input and output. Useful for real-time applications like video conferencing or live streaming.
	decode_perf.py	Measure video decoding speed using multiple decoder instances in parallel. Helps evaluate system performance and throughput.
	decode_sei_msg.py	Extract metadata embedded in the video stream (SEI messages). Used to retrieve information like HDR settings, timecodes, or custom data.
	simple_decode_stats.py	Extracts decode stats QP values, coding-unit types, and motion vectors from video stream
	simple_decode_from_memory.py	Decode video directly from in-memory sources — bytes, bytearray, memoryview, or any seekable BinaryIO stream — without writing a temporary file. Supports the full SimpleDecoder API including random frame access and batch retrieval.
	decode_reconfigure.py	Dynamically reconfigure the core decoder to handle resolution changes. Switch between input streams with different dimensions without recreating the decoder.

4.3. Basic Sample Applications

The basic sample applications demonstrate fundamental video decode, encode and transcode workflows. These applications are ideal for getting started with PyNvVideoCodec.

4.3.1. simple_decode_sampling.py

This sample demonstrates multi-file video decoding with frame sampling and tensor conversion, which is a common use case in machine learning pipelines. The application accepts one or more video files as input and samples frames evenly across each video's duration. Using the SimpleDecoder API, frames are decoded directly to GPU memory in RGB format and converted to PyTorch tensors for downstream processing. The sample showcases decoder reconfiguration, allowing a single decoder instance to process multiple video files of different resolutions efficiently. Frames are sampled evenly across the video duration based on the specified frame count.

```
python simple_decode_sampling.py video1.mp4 video2.mp4 -g 0 -f 16
```

Parameter	Type	Description
video_files	string	One or more video files to process (positional arguments)
-g, –gpu-id	int	GPU device ID to use (default: 0)
-f, –frames	int	Number of frames to sample per video (default: 8)

4.3.2. encode.py

This sample demonstrates video encoding with support for both CPU and GPU buffer modes, with optional muxing to container formats (MP4, MOV). The application reads raw YUV frames from an input file and encodes them using PyNvVideoCodec's encode API. In CPU buffer mode, frame data is read into host memory and copied to CUDA buffers for encoding. In GPU buffer mode, frame data is loaded directly into CUDA device buffers, reducing memory transfer overhead. The sample supports multiple codecs (H.264, HEVC, AV1) and various input formats (NV12, YUV444, P010, etc.). Muxing is automatically enabled when the output file has a container extension such as `.mp4` or `.mov`; using a codec-specific extension (such as `.h264` or `.hevc`) writes a raw elementary bitstream instead. Encoder configuration can be customized via a JSON configuration file for fine-grained control over encoding parameters such as bitrate, GOP size, and quality presets.

```
# Encode to MP4 container (muxing enabled automatically)
python encode.py -i input.yuv -s 1920x1080 -m gpu -o output.mp4

# Encode to raw H.264 bitstream (no muxing)
python encode.py -i input.yuv -s 1920x1080 -m gpu -o output.h264
```

Parameter	Type	Description
-i, –input	string	Path to raw input video file (YUV format)
-o, –output	string	Path to output file. Use a container extension (.mp4, .mov) to enable muxing; use a codec extension (.h264, .hevc, .av1) for a raw elementary bitstream.
-s, –size	string	Frame size in WxH format (e.g., 1920x1080)
-m, –mode	string	Buffer mode: cpu or gpu (default: gpu)
-if, –format	string	Input pixel format: NV12, YUV444, P010, ARGB, ABGR, etc. (default: NV12)
-f, –frames	int	Number of frames to encode (default: all frames)
-g, –gpu-id	int	GPU device ID to use (default: 0)
-json	string	Path to JSON encoder configuration file (controls codec, fps, GOP size, preset, bitrate, etc.)

4.3.3. create_video_segments.py

This sample demonstrates creation of smaller video segments from bigger video file. This approach is useful for creating small meaning video clips for training of AI models. The application extracts segments from a video file based on timestamp ranges specified in a text file. The sample uses the SimpleDecoder to retrieve video metadata for duration validation, then uses the Transcoder API to extract each segment as a separate output file. Output files are automatically named with the timestamp range appended to the base filename. Timestamps that exceed the video duration are automatically clipped, and invalid ranges are reported with error messages.

```
python create_video_segments.py -i input.mp4 -s segments.txt -c transcode_
↪config.json
```

Parameter	Type	Description
-i, –input	string	Path to input video file
-s, –segments	string	Path to segments file containing timestamp ranges (default: segments.txt)
-c, –config	string	Path to transcoder configuration JSON file (default: transcode_config.json)
-o, –output	string	Output filename template (timestamp is appended automatically)
-g, –gpu-id	int	GPU device ID to use (default: 0)

The segments file should contain one timestamp range per line in the format: `start_time end_time`

Example segments.txt:

```
0.0 10.5
15.0 30.0
45.5 60.0
```

4.4. Jupyter Notebooks

The Jupyter notebooks provide interactive tutorials that guide users through the PyNvVideoCodec APIs with step-by-step explanations and live code execution. These notebooks are ideal for learning and experimentation.

4.4.1. Setting Up Jupyter Notebooks

To run the Jupyter notebooks, follow these steps:

1. Ensure PyNvVideoCodec is installed in your Python environment.
2. Install the required dependencies:

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

3. Navigate to the notebooks directory and launch Jupyter:

   ```
   cd samples/jupyter
   jupyter notebook
   ```

4. Open the desired notebook from the Jupyter interface in your browser.

Note

Ensure that the CUDA toolkit is installed and your NVIDIA GPU drivers are up to date before running the notebooks.

4.4.2. simple_decode_tutorial.ipynb

This notebook provides a comprehensive tutorial on using the SimpleDecoder API for GPU-accelerated video decoding. The tutorial demonstrates how to decode videos efficiently and access frames using various methods. The notebook covers multiple frame access methods: single frame access by index, seeking to specific frames, batch frame retrieval, accessing frames by specific indices, slice-based access, and time-based frame retrieval. Another key feature demonstrated is decoder reconfiguration, which allows reusing a single decoder instance to process multiple video files of different resolutions without the overhead of creating new decoder instances. This is particularly useful in deep learning pipelines where processing many short video clips efficiently is critical. The sample also displays decoded output frames.

Key Features Demonstrated:

- Basic decoder setup and initialization with SimpleDecoder
- Retrieving video stream metadata (resolution, frame count, FPS, duration)
- Advanced stream information including key frame locations and GOP structure

- Multiple frame access methods: index, batch, slice, and time-based
- Decoder reconfiguration for processing multiple videos efficiently
- Converting decoded frames to PyTorch tensors

4.4.3. object_detection_tutorial.ipynb

This notebook demonstrates a practical deep learning use case: real-time object detection in video using ThreadedDecoder is used to decode video in parallel. Unlike a normal decoder where decode and inference are executed sequentially (causing the inference to stall while waiting for frames), the ThreadedDecoder runs decoding on a separate background thread and keeps next batch of pre-decoded frames ready for inference. This ensures that decoded frames are always ready when the inference model needs them, eliminating pipeline stalls and maximizing GPU utilization. The tutorial downloads a sample video, sets up a ThreadedDecoder with configurable buffer size, and uses a pre-trained Faster R-CNN model with ResNet-50 backbone to detect objects in video frames. The model is trained on the COCO dataset and can identify 91 different object classes. Detection results are filtered by confidence threshold and displayed with bounding boxes and class labels overlaid on each frame. This notebook serves as a template for building efficient video analytics applications.

Key Features Demonstrated:

- Understanding the difference between normal decoder and ThreadedDecoder
- Setting up ThreadedDecoder with buffer size configuration
- Integrating video decoding with PyTorch deep learning models
- Running Faster R-CNN object detection on decoded frames
- Filtering detection results by confidence threshold
- Visualizing object detection results with bounding boxes and labels
- Building efficient parallel decode-inference pipelines

4.4.4. face_protect_tutorial.ipynb

This notebook demonstrates a practical deep learning application: real-time face protection in video, where ThreadedDecoder is used to decode video frames in parallel. The threaded decoder runs in background and ensure that decoded frames are always ready when the inference model needs them. The tutorial downloads a sample video, sets up a ThreadedDecoder with configurable buffer size, and uses a pre-trained RetinaFace-ResNet34 model compiled to a TensorRT FP16 engine to detect faces in video frames. Detection results are filtered by confidence threshold and non-maximum suppression (NMS), then each detected face region is protected with GPU-accelerated Gaussian blur. Results are displayed inline with Matplotlib. This notebook serves as a template for building efficient, privacy-compliant video analytics applications.

Key Features Demonstrated:

- Setting up ThreadedDecoder with buffer size configuration
- Integrating video decoding with TensorRT deep learning models
- Downloading an ONNX model and building a TensorRT FP16 engine for the local GPU
- Running RetinaFace face detection on decoded frames
- Letterbox preprocessing and reverse coordinate mapping for variable aspect ratios

- Filtering detection results by confidence threshold and NMS
- Applying Gaussian blur to protect detected face regions
- Visualizing Face Protect results with inline frame display
- Building efficient parallel decode-inference pipelines

4.5. Advanced Sample Applications

This section demonstrates advanced features of PyNvVideoCodec including low-latency decoding, SEI message handling, performance measurement, CUDA resource management, and decode statistics extraction. These samples are designed for users who need fine-grained control over video processing workflows.

4.5.1. decode.py

This sample demonstrates basic video decoding using the core decoder API with a demuxer. The application creates a demuxer to extract encoded packets from a video file and sets up a core decoder for hardware-accelerated decoding.

The decode pipeline in this sample is as follows:

video file -> demuxer -> packets -> decoder -> raw YUV frames.

The sample shows how to work with both device and host memory for frame data, query hardware capabilities like the number of NVDEC engines, limit the number of frames decoded using the frame count parameter. Output is written as raw YUV frames to a file.

```
python decode.py -i input.mp4 -o output.yuv -d 1
```

Parameter	Type	Description
-i, –input	string	Path to input video file
-o, –output	string	Path to output raw YUV file (default: <input_name>.yuv)
-d	int	Use device memory (1) or host memory (0) (default: 1)
-g, –gpu-id	int	GPU device ID to use (default: 0)
-f, –frames	int	Number of frames to decode (default: all frames)

4.5.2. decode_with_cuda_control.py

This sample demonstrates advanced video decoding with explicit CUDA context and stream management. This application gives full control over CUDA resources by manually initializing CUDA contexts and streams before creating the decoder. This is essential for applications that need to share CUDA resources across multiple components or require precise control over GPU memory and synchronization. The sample shows proper CUDA resource cleanup order (decoder, demuxer, stream, context), switching between device memory (zero-copy) and host memory modes, and querying hardware capabilities.

```
python decode_with_cuda_control.py -i input.mp4 -o output.yuv -d 1
```

Parameter	Type	Description
-i, –input	string	Path to input video file
-o, –output	string	Path to output raw YUV file (default: <input_name>.yuv)
-d	int	Use device memory (1) or host memory (0) (default: 1)
-g, –gpu-id	int	GPU device ID to use (default: 0)
-f, –frames	int	Number of frames to decode (default: all frames)

4.5.3. decode_from_memory_buffer.py

This sample demonstrates decoding video directly from memory. This approach is useful when video data comes from network streams, memory-mapped files, or any source where data is available in memory rather than as a file on disk. The application implements a custom VideoStreamFeeder class that reads video data into memory and feeds chunks to the demuxer through a callback mechanism. This saves file I/O overhead and enables streaming applications. The sample shows how to create a callback-based demuxer, manage video data buffers and chunk sizes efficiently, implement proper buffer position tracking and EOF handling, and work with both device and host memory for decoded frames.

```
python decode_from_memory_buffer.py -i input.mp4 -o output.yuv -d 1
```

Parameter	Type	Description
-i, –input	string	Path to input video file
-o, –output	string	Path to output raw YUV file (default: <input_name>.yuv)
-d	int	Use device memory (1) or host memory (0) (default: 1)
-g, –gpu-id	int	GPU device ID to use (default: 0)
-f, –frames	int	Number of frames to decode (default: all frames)

4.5.4. decode_with_low_latency.py

This sample demonstrates low-latency video decoding using different latency modes. The decoder supports three latency modes:

- **NATIVE (0):** Has at least 1 frame latency for streams with B-frames and outputs in display order.
- **LOW (1):** Has zero latency for All-Intra and IPPP sequences (without B-frames) while maintaining display order.
- **ZERO (2):** Has zero latency for All-Intra/IPPP streams and outputs in decode order.

Low and zero latency modes should not be used with streams containing B-frames. For low latency modes, the sample sets the ENDOFPICTURE flag on packets to trigger immediate decode. This is useful for real-time video applications such as live streaming, video conferencing, and interactive media where minimizing decode latency is critical.

```
python decode_with_low_latency.py -i input.mp4 -o output.yuv -dl 1
```

Parameter	Type	Description
-i, –input	string	Path to input video file
-o, –output	string	Path to output raw YUV file (default: <input_name>.yuv)
-d	int	Use device memory (1) or host memory (0) (default: 1)
-g, –gpu-id	int	GPU device ID to use (default: 0)
-f, –frames	int	Number of frames to decode (default: all frames)
-dl	int	Decode latency mode: 0=NATIVE, 1=LOW, 2=ZERO

4.5.5. decode_perf.py

This sample demonstrates how to measure video decoding performance. The application supports two execution modes:

- **Thread mode:** Offers higher performance with lower overhead, shared memory access, and shared CUDA context between decoder instances. Python GIL is not a bottleneck as decoder operations are GIL-free.
- **Process mode:** Provides complete isolation between decoder instances with independent GPU memory.

The sample reports detailed performance metrics including frames per second (FPS), total frames decoded, and wall time. This is a performance testing application that does not write output files.

```
python decode_perf.py -i input.mp4 -n 4 -m thread
```

Parameter	Type	Description
-i, –input	string	Path to input video file
-n	int	Number of parallel instances (default: 1)
-m, –mode	string	Execution mode: thread or process (default: thread)
-d	int	Use device memory (1) or host memory (0) (default: 1)
-g, –gpu-id	int	GPU device ID to use (default: 0)
-f, –frames	int	Number of frames to decode per instance (default: all frames)

4.5.6. decode_sei_msg.py

This sample demonstrates how to extract and parse Supplemental Enhancement Information (SEI) messages from video streams during decoding. SEI messages are additional data embedded in video streams that provide supplementary information such as HDR/display metadata (color volume, light levels, transfer characteristics), timecode data for frame timing and sequence information, and custom metadata for application-specific needs. Common use cases include HDR display configuration for video playback, frame-accurate editing in content creation, timing synchronization in broadcast, and embedding application-specific metadata. The sample outputs raw binary SEI messages to one file and pickled SEI type information to another file, enabling further analysis or processing.

```
python decode_sei_msg.py -i input.mp4 -s sei_message.bin -st sei_type_message.
↪bin
```

Parameter	Type	Description
-i, –input	string	Path to input video file
-s	string	Output SEI message file (default: sei_message.bin)
-st	string	Output SEI type message file (default: sei_type_message.bin)
-d	int	Use device memory (1) or host memory (0) (default: 1)
-g, –gpu-id	int	GPU device ID to use (default: 0)

4.5.7. simple_decode_stats.py

This sample demonstrates how to extract and analyze decode statistics from H.264 and H.265 video streams using the SimpleDecoder API. The statistics collected include:

- **QP (Quantization Parameter):** Analysis with average, min, and max values per frame indicating compression levels.
- **CU (Coding Unit) type distribution:** Shows INTRA (spatial prediction), INTER (temporal prediction), SKIP (copy from reference), and PCM (uncompressed) blocks.
- **Motion vector statistics:** For L0 and L1 references, enabling temporal complexity assessment.
- **Macroblock details:** Provides per-block encoding decisions and parameters.

These statistics are valuable for video quality analysis, encoder behavior understanding, performance optimization, and debugging encoding/decoding issues. The output is written as formatted text to a statistics file.

```
python simple_decode_stats.py -i input.mp4 -p output_stats.txt -d 1
```

Parameter	Type	Description
-i, –input	string	Path to input video file
-p	string	Output file for statistics (default: <input_name>_stats.txt)
-d	int	Use device memory (1) or host memory (0) (default: 1)
-g, –gpu-id	int	GPU device ID to use (default: 0)

Note

The decode statistics feature requires NVIDIA display driver version 590 or newer.

4.5.8. simple_decode_from_memory.py

This sample demonstrates that `SimpleDecoder` can decode video directly from in-memory sources without requiring a temporary file on disk. It exercises every supported in-memory input type in one place:

- **bytes:** An immutable in-memory buffer. The entire buffer is copied once into native memory; FFmpeg reads from it with no per-read Python overhead.
- **bytearray:** A mutable in-memory buffer, handled the same way as `bytes`.
- **memoryview:** A zero-copy view over an existing buffer, also treated as a bytestream input.
- **BinaryIO stream:** Any seekable file-like object (file handle, `io.BytesIO`, tar member, S3/HTTP range reader, etc.). FFmpeg pulls only the bytes it needs on demand by calling the object's `read()` and `seek()` methods — no full-file copy required.

All input types are fully seekable and support the complete `SimpleDecoder` API: `len()`, random access (`decoder[i]`), slicing, out-of-order batch access (`get_batch_frames_by_index`), `seek_to_index`, and stream metadata. This is the key difference from `decode_from_memory_buffer.py`, which uses the older streaming (non-seekable) callback demuxer and only supports sequential decode.

```
python simple_decode_from_memory.py -i input.mp4
```

Parameter	Type	Description
-i, –input	string	Path to input video file (container format such as MP4 or MKV)
-g, –gpu-id	int	GPU device ID to use (default: 0)

4.5.9. decode_reconfigure.py

This sample demonstrates dynamic core decoder reconfiguration for handling resolution changes during playback. The application shows how to switch between input streams with different dimensions without recreating the decoder, which is useful for adaptive streaming scenarios or processing multiple video files with varying resolutions. The sample creates a decoder with maximum dimensions to accommodate both streams, decodes frames from the first stream, uses `setReconfigParams()` to reconfigure for the second stream's dimensions, and continues decoding the second stream.

```
python decode_reconfigure.py -i1 video1.mp4 -i2 video2.mp4 -o1 output1.yuv -
↪o2 output2.yuv
```

Parameter	Type	Description
-i1	string	Path to first input video file
-i2	string	Path to second input video file (can have different resolution)
-o1	string	Path to first output raw YUV file (default: <input1_name>.yuv)
-o2	string	Path to second output raw YUV file (default: <input2_name>.yuv)
-d	int	Use device memory (1) or host memory (0) (default: 1)
-g, –gpu-id	int	GPU device ID to use (default: 0)
-f, –frames	int	Number of frames to decode per stream (default: all frames)

4.5.10. encode_reconfigure.py

This sample demonstrates dynamic encoder reconfiguration for bitrate control at runtime. The application shows how to modify encoder parameters without resetting the encoder session, which is useful for adaptive bitrate streaming and dynamic quality adjustment. The sample changes the bitrate every 100 frames: at frame 0 it uses the original bitrate, at frame 100 it reduces to half the bitrate, at frame 200 it restores the original bitrate, and so on. The reconfiguration also handles VBV (Video Buffer Verifier) parameters including buffer size and initial delay. This capability is essential for live streaming applications that need to adapt to changing network conditions or for video conferencing systems that adjust quality based on bandwidth availability.

```
python encode_reconfigure.py -i input.yuv -s 1920x1080 -c h264
```

Parameter	Type	Description
-i, –input	string	Path to raw input video file (YUV format)
-o, –output	string	Path to output encoded video file
-s, –size	string	Frame size in WxH format (e.g., 1920x1080)
-if, –format	string	Input pixel format (default: NV12)
-c, –codec	string	Output codec: h264, hevc, av1
-f, –frames	int	Number of frames to encode (default: all frames)
-g, –gpu-id	int	GPU device ID to use (default: 0)
-json	string	Path to JSON encoder configuration file

4.5.11. encode_sei_msg.py

This sample demonstrates SEI (Supplemental Enhancement Information) message insertion during encoding. The application shows how to embed custom metadata into the encoded bitstream, which can be extracted during playback or transcoding. SEI messages support various types including HDR/display metadata (color volume, light levels, transfer characteristics), timecode data for frame timing, and custom user-defined data. The sample handles codec-specific SEI types: for H.264 and HEVC it uses SEI type 5 (user data unregistered), and for AV1 it uses type 6. Common use cases include embedding HDR metadata for proper display configuration, inserting timecodes for broadcast synchronization, and adding application-specific data for content management systems.

```
python encode_sei_msg.py -i input.yuv -s 1920x1080 -c hevc
```

Parameter	Type	Description
-i, –input	string	Path to raw input video file (YUV format)
-o, –output	string	Path to output encoded video file (auto-generated as .)
-s, –size	string	Frame size in WxH format (e.g., 1920x1080)
-if, –format	string	Input pixel format (default: NV12)
-c, –codec	string	Output codec: h264, hevc, av1
-f, –frames	int	Number of frames to encode (default: all frames)
-g, –gpu-id	int	GPU device ID to use (default: 0)
-json	string	Path to JSON encoder configuration file

4.5.12. encode_perf.py

This sample demonstrates advanced parallel video encoding using multiple threads or processes to achieve higher encoding throughput. Similar to decode_perf.py, it supports two execution modes:

- **Thread mode:** Runs multiple encoders in the same process with shared memory access, lower overhead, and simpler synchronization.
- **Process mode:** Runs multiple encoders in separate processes with IPC memory sharing, providing better isolation and stability.

The sample reports detailed performance metrics including total frames encoded, combined FPS across all workers, and average FPS per instance. This is a performance testing application that does not write output files.

```
python encode_perf.py -m thread -i input.yuv -s 1920x1080 -n 4
```

Parameter	Type	Description
-m, –mode	string	Execution mode: thread or process
-i, –input	string	Path to raw input video file (YUV format)
-s, –size	string	Frame size in WxH format (e.g., 1920x1080)
-if, –format	string	Input pixel format (default: NV12)
-c, –codec	string	Output codec: h264, hevc, av1
-n	int	Number of worker threads/processes (default: 1)
-f, –frames	int	Frames per worker (default: all frames)
-g, –gpu-id	int	GPU device ID to use (default: 0)
-json	string	Path to JSON encoder configuration file

4.5.13. transcode.py

This sample demonstrates a full GPU zero-copy transcode pipeline: demux → NVDEC → NVENC → mux, with audio passthrough. The entire pipeline runs on the GPU without any host round-trips, making it suitable for high-throughput, low-latency transcode workflows. The application shows how to create and share a CUDA context and stream across the decoder and encoder, decode an input video file to GPU memory, re-encode the decoded frames in-place to a different codec or container, and demux video and audio in lockstep using `DemuxNoSkipAudio` while muxing both streams to the output container, preserving the source's A/V interleave and bit-exact PTS. Encoder parameters such as bitrate, preset, and B-frames can be supplied via a JSON configuration file.

```
# Transcode H.264 input to HEVC and mux to MP4 (video + audio)
python transcode.py -i input.mp4 -o output.mp4 -c HEVC

# Same, with diagnostic YUV and raw bitstream dump
python transcode.py -i input.mp4 -o output.mp4 -c HEVC --dump

# Override encoder settings via JSON
python transcode.py -i input.mp4 -o output.mp4 -c H264 -json my_config.json
```

Parameter	Type	Description
-i, –input	string	Path to input video file (any container or elementary bitstream supported by FFmpeg)
-o, –output	string	Path to output video file. A container extension (.mp4, .mov, .mkv) enables muxing of video and audio.
-c, –codec	string	Output video codec: H264, HEVC, AV1 (default: same as input)
-g, –gpu-id	int	GPU device ID to use (default: 0)
–fps	float	Override output frame rate (default: taken from input)
–bframes	int	Number of B-frames (default: from JSON config)
–dump	flag	Dump decoded YUV and encoded bitstream to files alongside the output for offline inspection
-json	string	Path to JSON encoder configuration file (controls preset, bitrate, B-frames, etc.)

Copyright

©2010-2026, NVIDIA Corporation