PyNvVideoCodec API

Programming Guide

vPYNVVIDEOCODEC_PG-08085-001_v07" | April 2024

Table of Contents

Chapter 1. Overview....1
Chapter 2. Using PyNvVideoCodec API's....3
2.1. Video Demuxing....3
2.2. Video Decoding....4
2.3. Video Encoding....8
2.4. Video Encoding Basics....12
2.5. Video Encoding Parameter Details....13
2.6. Interoperability with DL/ML Frameworks....16

Chapter 1. Overview

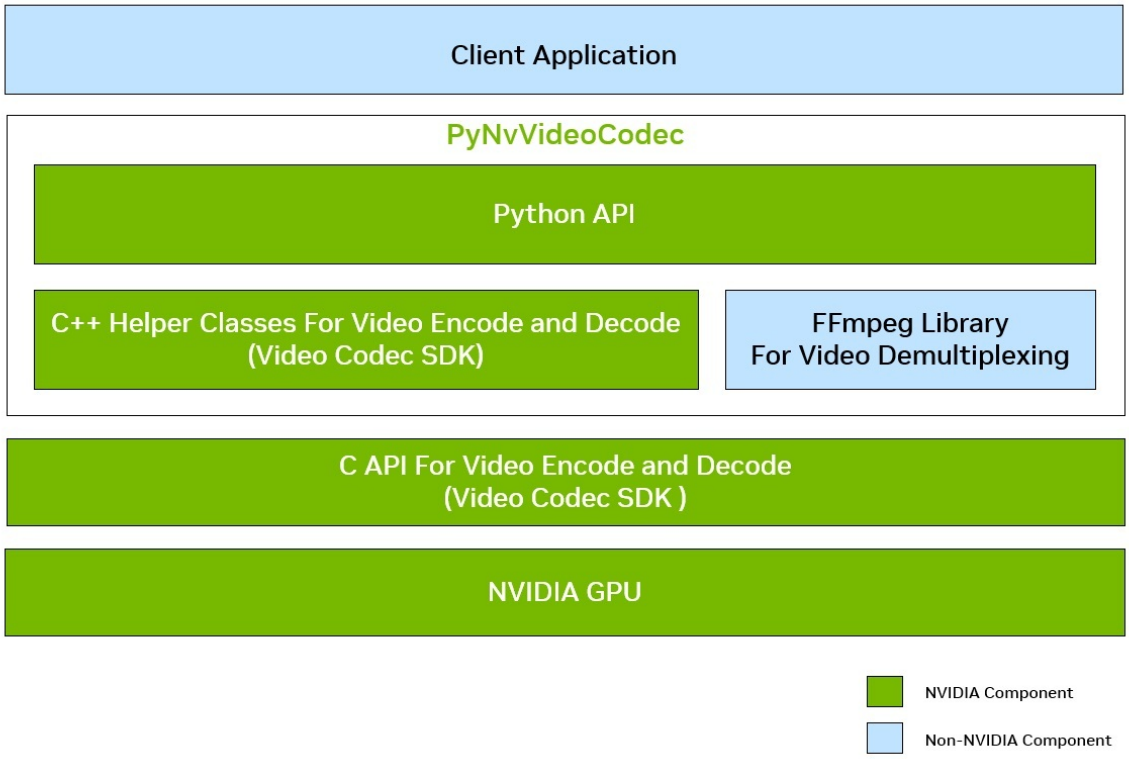
NVIDIA's Video Codec SDK offers hardware-accelerated video encoding and decoding through highly optimized C/C++ APIs. Such encoding and decoding of videos is also useful for a wide range of users, including computer vision experts, researchers and Deep Learning (DL) developers. The objective of PyNvVideoCodec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

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

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

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

Figure 1. High Level Architecture Diagram



Chapter 2. Using PyNvVideoCodec API's

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

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

2.1. Video Demuxing

Demux API

- `CreateDemuxer`

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

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

The `CreateDemuxer` has parameter as follows:

filename

Absolute path to file

Demux API usage

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

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

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

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

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

PacketData

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

The class `PacketData` has following attributes:

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

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

dts
The time at which the packet is decompressed.

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

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

pos
Byte position in stream.

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

2.2. Video Decoding

Decode API

1. `CreateDecoder`

```
import PyNvVideoCodec as nvc
decoder                                                                    =

 enableasyncallocations=False)
```

Here is the CreateDecoder API showing the defaolt parameters. In this case, the decoder internally manages allocation and deallocation of decode buffers.

```
CreateDecoder(
    gpuid: int = 0,
    codec: PyNvVideoCodec.
    _PyNvVideoCodec.cudaVideoCodec
    = <cudaVideoCodec.H264: 4>,
    cudacontext: int = 0,
    cudastream: int = 0,
    usedevicememory: bool = 0) -> PyNvDecoder
```

The `CreateDecoder` has named parameter as follows:

gpuid
Parameter not in use, please ignore

codec

code is inferred from Demuxer, can take any one of the values from list below:

- PyNvVideoCodec._PyNvVideoCodec.cudaVideoCodec.H264
- PyNvVideoCodec._PyNvVideoCodec.cudaVideoCodec.HEVC
- PyNvVideoCodec._PyNvVideoCodec.cudaVideoCodec.AV1

cudacontext

Handle to the CUDA Context created by application.

cudastream

Handle to CUDA Stream created by application

usedevicememory

Value of 1 indicates the surface allocation within library is in device memory and value of 0 indicates that its in Host memory

`CreatedDecoder` API returns an object that can be used to decode packets containing elementary bitstream to raw video frames. Please refer Demux API usage. to split the media file into `PacketData`

2. `decoder.Decode()` takes `PacketData` as input.

   Please refer to PacketData for more details

```
import PyNvVideoCodec as nvc
decoder = nvc.CreateDecoder(
          gpuid=0,
          codec=nvc.cudaVideoCodec.H264,
          cudacontext=0,
          cudastream=0,
          usedevicememory=True)
for decodedframe in decoder.Decode(packet):
        # process decodedframe
```

Video Decoding Details

Python sample Decoder.py shows how to decode video files.

1. Following examples show how to create decoder object and provide raw compressed data(`PacketData`) to `Decode()`.

   - Create a decoder object with cuda context created within library, default cuda stream and output surface in device memory.

     In this case, decoder creates and manages its own cuda context and stream.

     Output surface after call to `Decode()` resides in host memory

     Example below demonstrates how to create decoder object and fetch decoded frames as device memory buffer

```
import PyNvVideoCodec as nvc
demuxer = nvc.CreateDemuxer(
          filename=enc_file_path)
decoder = nvc.CreateDecoder(gpuid=0,
```

```
          codec=GetNvCodecId(),
          cudacontext=0,
          cudastream=0,
          usedevicememory=True)
for packet in demuxer:
    for decoded_frame in decoder.Decode(packet):
        new_array = cast_address_to_1d_bytearray(
        base_address=luma_base_addr,
        size=decoded_frame.framesize())
        #refer to Utils class for this implementation
```

- Create a decoder object with cuda context created within library, default cuda stream and output surface in host memory.

  In this case, decoder creates and manages its own cuda context and stream.

  Example below demonstrates how to create decoder object and fetch decoded frames as host memory buffer

```
import PyNvVideoCodec as nvc
import pycuda.driver as cuda
demuxer = nvc.CreateDemuxer(
          filename=enc_file_path)
decoder = nvc.CreateDecoder(
          gpuid=0,
          codec=GetNvCodecId(),
          cudacontext=0,
          cudastream=0,
          usedevicememory=False)
seq_triggered = False
for packet in demuxer:
    for decoded_frame in decoder.Decode(packet):
        if not seq_triggered:
            decoded_frame_size
            = nv_dec.GetFrameSize()
            raw_frame
            = np.ndarray(
            shape=decoded_frame_size,
            dtype=np.uint8)
            seq_triggered = True
        cuda.memcpy_dtoh(
            raw_frame,
            luma_base_addr)
```

- Create a decoder object with externally manaager cuda context, stream and output surface from decoder is in device memory.

  In this case, decoder uses externally created cuda context and stream.

  Example below demonstrates how to create decoder object and fetch decoded frames from device memory buffer.

```
import PyNvVideoCodec as nvc
import pycuda.driver as cuda
cuda.init()
cuda_device = cuda.Device(0)
cuda_ctx = cuda_device.retain_primary_context()
cuda_ctx.push()
cuda_stream_decoder = cuda.Stream()
seq_triggered = False
demuxer = nvc.CreateDemuxer(
          filename=enc_file_path)
decoder = nvc.CreateDecoder(
```

```
          gpuid=0,codec=nvc.cudaVideoCodec.H264,
          cudacontext=cuda_ctx.handle,
          cudastream=cuda_stream_decoder.handle,
          usedevicememory=True)
for packet in demuxer:
    for decoded_frame in decoder.Decode(packet):
        if not seq_triggered:
            decoded_frame_size = nv_dec.GetFrameSize()
            raw_frame = np.ndarray(
            shape=decoded_frame_size,
            dtype=np.uint8)
            seq_triggered = True
        cuda.memcpy_dtoh(
        raw_frame,
        luma_base_addr)
```

- Create a decoder object with asynchronous allocations enabled.

  In this case, decoder allocates device memory on externally provided cuda stream and context instead of creating its own.

  Example below demonstrates how to create decoder object and fetch decoded frames to device memory buffer allocated on external cuda stream.

```
import PyNvVideoCodec as nvc
import pycuda.driver as cuda
cuda.init()
cuda_device = cuda.Device(0)
cuda_ctx = cuda_device.retain_primary_context()
cuda_ctx.push()
cuda_stream_decoder = cuda.Stream()
cuda_stream_app = cuda.Stream()
decoder = nvc.CreateDecoder(
          gpuid=0,
          codec=nvc.cudaVideoCodec.H264,
          cudacontext=cuda_ctx.handle,
          cudastream=cuda_stream_decoder.handle,
          usedevicememory=True,
          enableasyncallocations=True)
raw_frame = None
seq_triggered = False
for packet in demuxer:
    for decoded_frame in decoder.Decode(packet):
        if not seq_triggered:
            decoded_frame_size = decoder.GetFrameSize()
            raw_frame = cuda.pagelocked_empty(
            shape=decoded_frame_size,
            dtype=np.uint8,
            order='C',
              mem_flags=0)  # for stream aware allocations, we need to create
 page locked host
            # memory
            seq_triggered = True
        luma_base_addr = decoded_frame.GetPtrToPlane(0)
        decoder.WaitOnCUStream(cuda_stream_app.handle)
        cuda.memcpy_dtoh_async(
        raw_frame,
        luma_base_addr,
        cuda_stream_app)
        cuda_stream_app.synchronize()
```

> **!** **ATTENTION:** Please note the `WaitOnCUStream` call after decoded frames are received, since allocation is done on a stream different than stream on which memory copy is scheduled. application needs to wait till allocation is complete only then it can schedule the memory copy.

2. Client needs to check the pitch of the output surface before calling the interoperability API, pitch of the decoded surface is aligned by 16 bytes.
3. To Decode SVC(Scalable Video Coding) streams or having Dynamic Resolution Change, users should enable dumping output in host memory
4. After decoding, ownership of buffers remains with PyNvVideoCodec library only, Client application needs to deep copy the the decoded surface for usage.
5. Output buffers in NvCUVID are size of DPB, for H264 codec its 16.

2.3. Video Encoding

Encode API

1. `CreateEncoder`

   This method returns an object of encoder.

   Example below shows how to create encoder object with minimal parameters

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

   The `CreateEncoder` takes following parameters

   gpuid
   Parameter not in use, please ignore

   width
   The desired width of the encoded video

   height
   The desired height of the encoded video

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

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

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

2. `Encode`

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

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

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

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

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

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

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

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

class AppFrame:
    def __init__(self, width, height, format):
        if format == "NV12":
            nv12_frame_size = int(width * height * 3 / 2)
            self.gpuAlloc = cuda.mem_alloc(nv12_frame_size)
            self.cai = []
            self.cai.append(AppCAI(
            (height, width, 1),
            (width, 1, 1),
            "|u1", self.gpuAlloc))
            chroma_alloc = int(self.gpuAlloc)
            + width * height
            self.cai.append(AppCAI((int(height / 2),
            int(width / 2), 2),
            (width, 2, 1),
            "|u1",
            chroma_alloc))
            self.frameSize = nv12_frame_size
    def cuda(self):
        return self.cai

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

```
          1080,
          "NV12")
bitstream = encoder.Encode(input_gpu_frame)
```

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

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

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

Example below shows how to represent NV12 as NCHW Tensor

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

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

3. `EndEncode`

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

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

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

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

4. `GetEncodeReconfigureParams` and `Reconfigure`

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

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

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

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

   Resolution change is possible only if `NV_ENC_INITIALIZE_PARAMS::maxEncodeWidth` and `NV_ENC_INITIALIZE_PARAMS::maxEncodeHeight` are set while creating encoder session.

   If the client wishes to change the resolution using this API, it is advisable to force the next frame following the reconfiguration as an IDR frame by setting `NV_ENC_RECONFIGURE_PARAMS::forceIDR` to 1.

   If the client wishes to reset the internal rate control states, set `NV_ENC_RECONFIGURE_PARAMS::resetEncoder` to 1.

   Example below shows how to fetch and change reconfigurable parameters:

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

2.4. Video Encoding Basics

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

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

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

Surface Format Support

Currently supported input formats are

- NV12(8 bit)
- YUV 4:2:0(10 bit)
- YUV 4:4:4(8 bit and 10 bit)

Both 10 bit and 16 bit input frames result in 10 bit encoding. The colorspace conversion matrix can be specified by the client using the colorspace option during `CreateEncoder`.

Tuning

The NVIDIA Encoder Interface exposes four different tuning options:

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

Presets

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

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

Rate Control and Bitrate

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

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

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

Building your Optimized Encoder

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

2.5. Video Encoding Parameter Details

Table 1. 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

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

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

Parameter	Type	Valid Values	Default Parameter	Description
				does not support `slice::data`
`repeatspspps`	Integer	0 or 1	0	Enable writing of `Sequence Parameter Set` (SPS) and `Picture Parameter Set` (PPS) for every IDR frame

2.6. Interoperability with DL/ML Frameworks

Example below shows how `DecodedFrame` can be consumed by `PyTorch` without the need of explicit memory copy

```
for packet in demuxer:
    for decoded_frame in decoder.Decode(packet):
     src_tensor = torch.from_dlpack(decoded_frame)
```

"PyNvVideoCodec APIs can seamlessly (zero-copy) exchange data with popular DL frameworks like PyTorch and TensorRT. Video frame decoded by PyNvVideoCodec decode API can be directly consumed by DL framework. The decoded surface supports DLpack and CUDA Arrary Inteface for enabling this. Similarly encode API can consume the video frame produced by DL frameworks.

Example below shows a `DecodedFrame` class for NV12 1080p Surface. The `DecodedFrame` instance contains list of `CAIMemoryView`.

For NV12 list of `CAIMemoryView` would have 2 entries one for luma component and other for chroma component.

```
import PyNvVideoCodec as nvc
print(nvc.DecodedFrame)
<DecodedFrame [timestamp=0, format=Pixel_Format.NV12, [<CAIMemoryView [1080, 1920,
 1]>, <CAIMemoryView [540, 960, 2]>]]>
```

`DecodedFrame` implements methods as below:

1. Access the underlying list of `CAIMemoryView` where each view implements `__cuda_array_interface__`.
   ```
   decodedFrame.cuda()
   ```
2. Convert `DecodedFrame` in semi-planar NV12 and YUV444 format to 1-D single channel tensor.
   ```
   decodedFrame.nvcv_image()
   ```
3. Access the DLPack methods. DLPack is an intermediate in-memory representation standard for tensor data structures that allows exchange between major frameworks.

- Shape of Tensor - (tuple of ints describing axes length)
  ```
  decodedFrame.shape()
  ```
- Stride of Tensor - (tuple of ints describing strides of data in memory)
  ```
  decodedFrame.shape()
  ```
- `dtype` of Tensor - (data type)
  ```
  decodedFrame.dtype()
  ```

4. Access the opaque pointer to the underlying GPU buffer.
   ```
   decodedFrame.__dlpack_device___
   ```

ATTENTION: In order to create custom DataLoader for media files, please refer NVVL

Notice

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

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

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

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

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

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

Trademarks

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

Copyright

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