NVIDIA VIDEO CODEC SDK - DECODER

Programming Guide

vNVDECODEAPI_PG-08085-001_v07 | January 2025

Table of Contents

Chapter 1. Overview

NVIDIA GPUs - beginning with the NVIDIA® Fermi™ generation - contain a video decoder engine (referred to as NVDEC in this document) which provides fully-accelerated hardware video decoding capability. NVDEC can be used for decoding bitstreams of various formats: AV1, H.264, HEVC (H.265), VP8, VP9, MPEG-1, MPEG-2, MPEG-4 and VC-1. NVDEC runs completely independent of compute/graphics engine. NVDEC 的运行完全独立于计算 / 图形引擎，不会占用其资源。

NVIDIA provides software API and libraries for programming NVDEC. The software API, hereafter referred to as NVDECODE API lets developers access the video decoding features of NVDEC and interoperate NVDEC with other engines on the GPU. 开发者可通过它调用 NVDEC 的视频解码功能，并实现 NVDEC 与显卡上其他引擎的协同工作。

NVDEC 对压缩视频流进行解码后，会将生成的 YUV 帧复制到显存中。由于帧数据存储在显存内，可通过 CUDA 对视频进行后处理。此外，NVDECODE API 还提供经 CUDA 优化的常用后处理操作实现，包括缩放、裁剪、宽高比转换、去隔行以及向多种主流输出视频格式的色彩空间转换。开发者可选择使用 NVDECODE API 提供的这些 CUDA 优化实现，也可自行对解码后的输出帧实现后处理逻辑。

NVDEC decodes the compressed video streams and copies the resulting YUV frames to video memory. With frames in video memory, video post processing can be done using CUDA. The NVDECODE API also provides CUDA-optimized implementation of commonly used post-processing operations such as scaling, cropping, aspect ratio conversion, de-interlacing and color space conversion to many popular output video formats. The client can choose to use the CUDA-optimized implementations provided by the NVDECODE API for these post-processing steps or choose to implement their own post-processing on the decoded output frames.

Decoded video frames can be presented to the display with graphics interoperability for video playback, passed directly to a dedicated hardware encoder (NVENC) for high-performance video transcoding, used for GPU accelerated inferencing or consumed further by CUDA or CPU-based processing. 解码后的视频帧可通过图形协同功能输出到显示器进行视频播放，也可直接传递至专用硬件编码器（NVENC）以实现高性能视频转码，还可用于 GPU 加速推理，或供 CUDA 及基于 CPU 的处理流程进一步使用。

1.1. Supported Codecs

The codecs supported by NVDECODE API are:

- MPEG-1,
- MPEG-2,
- MPEG4,
- VC-1 (Simple/Main/Advanced Profile),
- H.264 (AVCHD) Baseline/Main/High/High10 (exclude MBAFF)/High422 (exclude MBAFF) Profile,
- H.265 (HEVC) (Main/Main10/Main12 Profile, Main 4:2:2/4:4:4 10/12 profile (exclude YUV400)),
- VP8,

- VP9(8bit, 10 bit and 12 bit),
- AV1 Main profile,
- Hybrid (CUDA + CPU) JPEG 混合（CUDA + CPU）JPEG

Refer to Chapter 2 for complete details about the video capabilities for various GPUs.

Chapter 2. Video Decoder Capabilities

Table 1 shows the codec support and capabilities of the hardware video decoder for each GPU architecture.

Table 1. Hardware Video Decoder Capabilities

GPU Architecture	MPEG-1 & MPEG-2	VC-1 & MPEG-4	H.264/AVCHD	H.265/HEVC	VP8	VP9	AV1
Fermi (GF1xx)	**Maximum Resolution:** 4080x4080	**Maximum Resolution:** 2048x1024 & 1024x2048	**Maximum Resolution:** 4096x4096 **Profile:** Baseline, Main, High profile up to Level 4.1	Unsupported	Unsupported	Unsupported	Unsupported
Kepler (GK1xx)	**Maximum Resolution:** 4080x4080	**Maximum Resolution:** 2048x1024 & 1024x2048	**Maximum Resolution:** 4096x4096 **Profile:** Main, Highprofile up to Level4.1	Unsupported	Unsupported	Unsupported	Unsupported
First generation Maxwell (GM10x)	**Maximum Resolution:** 4080x4080	**Maximum Resolution:** 2048x1024 & 1024x2048	**Maximum Resolution:** 4096x4096 **Profile:** Baseline, Main, High profile up to Level5.1	Unsupported	Unsupported	Unsupported	Unsupported
Second generation Maxwell (GM20x, except GM206)	**Maximum Resolution:** 4080x4080	**Maximum Resolution:** 2048x1024 & 1024x2048 Max bitrate: 60 Mbps	**Maximum Resolution:** 4096x4096 **Profile:** Baseline, Main, High profile up to Level5.1	Unsupported	**Maximum Resolution:** 4096x4096	Unsupported	Unsupported

GPU Architecture	MPEG-1 & MPEG-2	VC-1 & MPEG-4	H.264/AVCHD	H.265/HEVC	VP8	VP9	AV1
GM206	**Maximum Resolution:** 4080x4080	**Maximum Resolution:** 2048x1024 & 1024x2048	**Maximum Resolution:** 4096x4096 **Profile:** Baseline, Main, High profile up to Level5.1	**Maximum Resolution:** 4096x2304 **Profile:** Main profile up to Level5.1 and main10 profile	**Maximum Resolution:** 4096x4096	**Maximum Resolution:** 4096x2304 **Profile:** Profile 0	Unsupported
GP100	**Maximum Resolution:** 4080x4080	**Maximum Resolution:** 2048x1024 & 1024x2048	**Maximum Resolution:** 4096x4096 **Profile:** Baseline, Main, High profile up to Level 5.1	**Maximum Resolution:** 4096x4096 **Profile:** Main profile up to Level 5.1, main10 and main12 profile	**Maximum Resolution:** 4096x4096	**Maximum Resolution:** 4096x4096 **Profile:** Profile 0	Unsupported
GP10x/ GV100/ Turing/GA100	**Maximum Resolution:** 4080x4080	**Maximum Resolution:** 2048x1024 & 1024x2048	**Maximum Resolution:** 4096x4096 **Profile:** Baseline, Main, High profile up to Level 5.1	**Maximum Resolution:** 8192x8192 **Profile:** Main profile up to Level 6.0, main10 and main12 profile	**Maximum Resolution:** 4096x4096[1]	**Maximum Resolution:** 8192x8192[2] **Profile:** Profile 0, 10-bit and 12-bit decoding	Unsupported
Hopper	**Maximum Resolution:** 4080x4080	**Maximum Resolution:** 2048x1024 & 1024x2048	**Maximum Resolution:** 4096x4096 **Profile:** Baseline, Main, High profile up to Level 5.1	**Maximum Resolution:** 8192x8192 **Profile:** Main, main10 and main12 profile up to Level 6.0, 444 chroma format support	**Maximum Resolution:** 4096x4096	**Maximum Resolution:** 8192x8192 **Profile:** Profile 0, 10-bit and 12-bit decoding	Unsupported
GA10x/AD10x	**Maximum Resolution:** 4080x4080	**Maximum Resolution:** 2048x1024 & 1024x2048	**Maximum Resolution:** 4096x4096 **Profile:** Baseline, Main, High profile up to Level 5.1	**Maximum Resolution:** 8192x8192 **Profile:** Main, main10 and main12 profile up to Level 6.0, 444 chroma format support	**Maximum Resolution:** 4096x4096	**Maximum Resolution:** 8192x8192 **Profile:** Profile 0, 10-bit and 12-bit decoding	**Maximum Resolution:** 8192x8192 **Profile:** Profile 0 upto level 6.0
Blackwell	**Maximum Resolution:** 4080x4080	**Maximum Resolution:** 2048x1024 & 1024x2048	**Maximum Resolution:** 8192x8192 **Profile:**	**Maximum Resolution:** 8192x8192 **Profile:**	**Maximum Resolution:** 4096x4096	**Maximum Resolution:** 8192x8192 **Profile:**	**Maximum Resolution:** 8192x8192 **Profile:**

GPU Architecture	MPEG-1 & MPEG-2	VC-1 & MPEG-4	H.264/AVCHD	H.265/HEVC	VP8	VP9	AV1
			Baseline, Main, High up to Level 6.2 High 10 and High 422 (exclude MBAFF) up to Level 6.2	Main, main10, main12, Main 4:2:2/4:4:4 10/12 profile (exclude YUV400) Level up to 6.0		Profile 0, 10-bit and 12-bit decoding	Profile 0 upto level 6.0

[1] Supported only on select GP10x GPUs, all Turing GPUs and GA100

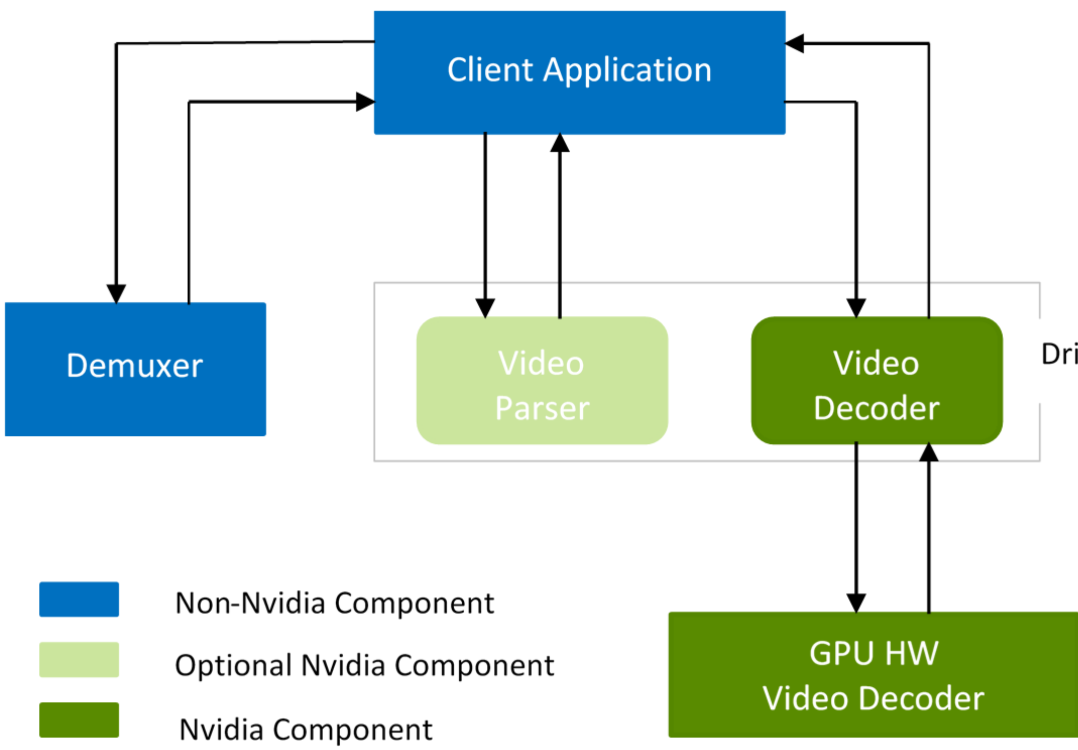
[2] VP9 10-bit and 12-bit decoding is supported on select GP10x GPUs, all Turing GPUs and GA100

Chapter 3. Video Decoder Pipeline

解码器流水线包含三个核心组件：解复用器（Demuxer）、视频解析器（Video Parser）和视频解码器（Video Decoder）。这三个组件相互独立，可单独使用。NVDECODE API 提供了用于调用 NVIDIA 视频解析器和 NVIDIA 视频解码器的接口，其中 NVIDIA 视频解析器纯为软件组件，若有需求，开发者也可自行实现解析器以替代它。

Decoder pipeline consists of three major components - Demuxer, Video Parser, and Video Decoder. The components are not dependent on each other and hence can be used independently. NVDECODE API provide API's for NVIDIA video parser and NVIDIA video decoder. Of these, NVIDIA video parser is purely a software component and users can implement their own parser in place of NVIDIA video parser, if required.

Figure 1. Video decoder pipeline using NVDECODE API



At a high level the following steps should be followed for decoding any video content using NVDECODEAPI:

1. Create a CUDA context.
2. Query the decode capabilities of the hardware decoder.

3. Create the decoder instance(s). 创建解码器实例
4. De-Mux the content (like .mp4). This can be done using third party software like FFMPEG.
5. Parse the video bitstream using parser provided by NVDECODE API or third-party parser such as FFmpeg.
6. Kick off the Decoding using NVDECODE API. 调用 NVDECODE API 启动解码流程。
7. Obtain the decoded YUV for further processing.
8. Query the status of the decoded frame. 查询解码帧的状态。
9. Depending on the decoding status, use the decoded output for further processing like rendering, inferencing, postprocessing etc. 根据解码状态，将解码输出用于渲染、推理、后处理等后续操作。
10. If the application needs to display the output,
    - Convert decoded YUV surface to RGBA.
    - Map RGBA surface to DirectX or OpenGL texture.
    - Draw texture to screen.
11. Destroy the decoder instance(s) after the completion of decoding process.
12. Destroy the CUDA context.

The above steps are explained in the rest of the document and demonstrated in the sample application(s) included in the Video Codec SDK package.

Chapter 4. Using NVIDIA Video Decoder (NVDECODE API)

所有 NVDECODE API 均通过两个头文件暴露：cuviddec.h 和 nvcuvid.h，这两个头文件位于视频编解码 SDK 软件包的 Interface 文件夹下。
NVIDIA 视频编解码 SDK 中的示例程序会静态加载库（Windows 版 SDK 软件包附带该库）函数，并在源文件中包含 cuviddec.h 和 nvcuvid.h。其中，Windows 系统的 nvcuvid.dll 随 NVIDIA 显示驱动一同提供，Linux 系统的 libnvcuvid.so 也包含在 NVIDIA 显示驱动中。

All NVDECODE APIs are exposed in two header-files: `cuviddec.h` and `nvcuvid.h`. These headers can be found under `Interface` folder in the Video Codec SDK package. The samples in NVIDIA Video Codec SDK statically load the library (which ships as a part of the SDK package for windows) functions and include `cuviddec.h` and `nvcuvid.h` in the source files. The Windows DLL `nvcuvid.dll` is included in the NVIDIA display driver for Windows. The Linux library `libnvcuvid.so` is included with NVIDIA display driver for Linux.

The following sections in this chapter explain the flow that should be followed to accelerate decoding using NVDECODE API.

4.1. Video Parser

4.1.1. Creating a parser

Parser object can be created by calling `cuvidCreateVideoParser()` after filling the structure `CUVIDPARSERPARAMS`. The structure should be filled up with following information about the stream to be decoded:

ulMaxNumDecodeSurfaces：表示解析器解码图像缓冲区（DPB）中的数量。初始化解析器时，该值可能未知，可先设为 1 等临时值以创建解析器对象。应用程序必须向驱动注册 pfnSequenceCallback 回调函数 —— 当解析器遇到第一个序列头或序列发生变化时，会调用该回调函数。回调函数会通过 CUVIDEOFORMAT::min_num_decode_surfaces 告知解析器 DPB 正确解码所需的最小数量。

- `CodecType`: must be from `enum cudaVideoCodec`, indicating codec type of content like H.264, HEVC, VP9 etc.
- `ulMaxNumDecodeSurfaces`: This is number of surfaces in parser's DPB (decode picture buffer). This value may not be known at the parser initialization time and can be set to a dummy number like 1 to create parser object. Application must register a callback `pfnSequenceCallback` with the driver, which is called by the parser when the parser encounters the first sequence header or any changes in the sequence. This callback reports the minimum number of surfaces needed by parser's DPB for correct decoding in `CUVIDEOFORMAT::min_num_decode_surfaces`. The sequence callback may return this value to the parser if wants to update `CUVIDPARSERPARAMS::ulMaxNumDecodeSurfaces`. The parser then overwrites `CUVIDPARSERPARAMS::ulMaxNumDecodeSurfaces` with the value returned by the sequence callback, if return value of the sequence callback is greater than 1 (see description about `pfnSequenceCallback` below). Therefore, for optimum memory allocation, decoder object creation should be deferred until `CUVIDPARSERPARAMS::ulMaxNumDecodeSurfaces` is known, so that the decoder object can be created with required

若序列回调函数的返回值大于 1，解析器会用该返回值覆盖 CUVIDPARSERPARAMS::ulMaxNumDecodeSurfaces（详见下文 pfnSequenceCallback 说明）。因此，为实现内存优化分配，建议延迟创建解码器对象，直至获取 CUVIDPARSERPARAMS::ulMaxNumDecodeSurfaces 的准确值，确保解码器对象创建时分配的缓冲区数量满足 CUVIDDECODECREATEINFO::ulNumDecodeSurfaces = CUVIDPARSERPARAMS::ulMaxNumDecodeSurfaces。

number of buffers, such that `CUVIDDECODECREATEINFO::ulNumDecodeSurfaces` = `CUVIDPARSERPARAMS::ulMaxNumDecodeSurfaces`.

- `ulClockRate`: is timestamp units in Hz (0=default=10000000Hz)
- `ulErrorThreshold`: controls non-compliance bitstream checks in parser. Its valid range is 0 to 100. 0 means strict check and parser will return error if found any non-compliance or error and 100 means ignore all non-compliance bitstream checks in parser.

  控制解析器对非合规码流的检查严格程度，取值范围为 0~100。0 表示严格检查，若发现非合规或错误，解析器会返回错误；100 表示忽略所有非合规码流检查。
- `ulMaxDisplayDelay`: Max display callback delay. 0 = no delay
- `bAnnexb`: must be set to 1 for AV1 annexB streams
- `pfnSequenceCallback`: Application must register a function to handle any sequence change. Parser triggers this callback for initial sequence header or when it encounters a video format change. Return value from sequence callback is interpreted by the driver as follows: 应用程序必须注册此函数以处理序列变化。当解析器遇到初始序列头或视频格式变化时，会触发该回调。
  - 0: fail
  - 1: succeeded, but driver should not override `CUVIDPARSERPARAMS::ulMaxNumDecodeSurfaces`
  - >1: succeeded, and driver should override `CUVIDPARSERPARAMS::ulMaxNumDecodeSurfaces` with this return value
- `pfnDecodePicture`: Parser triggers this callback when bitstream data for one frame is ready. In case of field pictures, there may be two decode calls per one display call since two fields make up one frame. Return value from this callback is interpreted as: 当一帧的码流数据准备就绪时，解析器会触发该回调。
  - 0: fail
  - ≥1: succeeded
- `pfnDisplayPicture`: Parser triggers this callback when a frame in display order is ready. Return value from this callback is interpreted as: 当一帧按显示顺序准备就绪时，解析器会触发该回调。
  - 0: fail
  - ≥1: succeeded
- `pfnGetOperatingPoint`: Parser triggers this callback to get operating point of an AV1 scalable stream. Parser picks default operating point as 0 and outputAllLayers flag as 0 if `pfnGetOperatingPoint` is not set or return value is -1 or invalid operating point. Return value from this callback is interpreted as: 解析器触发该回调以获取 AV1 可伸缩码流的工作点。若未设置 pfnGetOperatingPoint，或返回值为 -1 及无效工作点，解析器会默认选择工作点 0，且 outputAllLayers 标志设为 0。
  - < 0: fail
  - ≥0: succeeded (bit 0-9: currOperatingPoint, bit 10-10: bOutputAllLayer)
- `pfnGetSEIMsg`: Parser triggers this callback in decode order when all the unregistered user SEI messages or Metadata OBUs are parsed for a frame. Currently this callback is supported for H264, HEVC and AV1 codecs. Return value from this callback is interpreted as:

  当解析器解析完一帧中所有未注册的用户补充增强信息（SEI）或元数据 OBU（按解码顺序）时，会触发该回调。目前，该回调支持 H.264、HEVC 及 AV1 编码格式。
  - 0: fail
  - ≥1: succeeded

4.1.2. Parsing the packets

从demuxer提取的码流需连同其长度、时间戳、标志等辅助信息一同封装到 CUVIDSOURCEDATAPACKET 结构体（简称 “数据包”）中，再通过 cuvidParseVideoData() 将该数据包传入解析器。

Bitstream extracted from demultiplexer along with its length and some other auxiliary info like timestamp, flags is packed into struct `CUVIDSOURCEDATAPACKET` , called as packet. This packet is fed into parser using `cuvidParseVideoData()`. This packet is initialized as:

- `flags`: These flags are set by application and interpreted by parser as below:
  - `CUVID_PKT_ENDOFSTREAM`: MUST be set with last packet for this stream. Parser will trigger display callback for all pending buffers in the display queue.

    必须与该码流的最后一个数据包一同设置，解析器会触发显示回调以处理显示队列中所有未处理的缓冲区。
  - `CUVID_PKT_TIMESTAMP`: indicate that timestamp in packet is valid.
  - `CUVID_PKT_DISCONTINUITY`: should be set if there is any discontinuity like packet after seek. 若存在码流不连续（如跳转后的数据包），需设置该标志

    当数据包恰好包含一帧或一个场的数据时，必须设置该标志。基于NALU的编码格式，其解码回调存在一帧延迟（因解析器需接收下一帧的部分非 VCL NALU 才能检测当前帧边界）。设置该标志可强制解析器跳过边界检查，立即触发解码回调。若数据包数据不完整，解码回调会接收到部分帧数
  - `CUVID_PKT_ENDOFPICTURE`: MUST be set when packet contains exactly one frame or one field data. NALU based codecs have one frame latency for decode callback as parser detects frame boundary when some non-VCL NALU are received (that belong to next frame). This flag will force parser to skip this boundary check and trigger decode callback immediately. If packet has incomplete data, decode callback will get triggered with partial frame data. If packet has more than one frame data, parser will trigger decode callback for first frame data. Rest of the NALU will get dropped.

    若与 CUVID_PKT_ENDOFSTREAM 同时设置，解析器会额外触发一次显示回调，且 CUVIDPARSERDISPINFO 取值为 NULL，以此表示码流结束
  - `CUVID_PKT_NOTIFY_EOS`: If this flag is set along with CUVID_PKT_ENDOFSTREAM, an additional (dummy) display callback will be invoked with null value of CUVIDPARSERDISPINFO which should be interpreted as end of the stream.
- `payload_size`: represents number of bytes in payload
- `payload`: points to bitstream memory buffer
- `timestamp`: Presentation time stamp (10MHz clock), only valid if `CUVID_PKT_TIMESTAMP` flag is set

在 cuvidParseVideoData() 函数内部，只要满足相应条件（如序列参数变化时触发 pfnSequenceCallback、帧准备好解码时触发 pfnDecodePicture），解析器就会同步触发创建解析器对象时注册的回调函数。若回调函数返回失败，cuvidParseVideoData() 会将该失败状态传递给应用程序。

Parser triggers callbacks registered while creating parser object synchronously from within `cuvidParseVideoData()`, whenever there is corresponding condition is hit like `pfnSequenceCallback` when there is change in sequence parameters or `pfnDecodePicturepicture` when frame is ready to be decoded. If the callback returns failure, it will be propagated by `cuvidParseVideoData()` to the application.

The decoded result gets associated with a picture-index value in the `CUVIDPICPARAMS` structure, which is also provided by the parser. This picture index is later used to map the decoded frames to CUDA memory. 解码结果会与 CUVIDPICPARAMS 结构体中的图像索引（picture-index）关联，该结构体也由解析器提供。后续可通过此图像索引将解码帧映射到 CUDA 内存。

4.1.3. Destroying parser

The user needs to call `cuvidDestroyVideoParser()` to destroy the parser object and free up all the allocated resources.

4.2. Video Decoder

4.2.1. Querying decode capabilities

The API `cuvidGetDecoderCaps()` lets users query the capabilities of underlying hardware video decoder. cuvidGetDecoderCaps() API 可用于查询底层硬件视频解码器的功能特性。

As illustrated in Table 1, different GPUs have hardware decoders with different capabilities. Therefore, to ensure your application works on all generations of GPU hardware, it is highly recommended that the application queries the hardware capabilities and makes appropriate decision based on presence/absence of the desired capability/functionality.

调用 cuvidGetDecoderCaps() 前，调用线程需关联有效的 CUDA 上下文。用户需先填充 CUVIDDECODECAPS 结构体的以下字段:

The API `cuvidGetDecoderCaps()` lets users query the capabilities of underlying hardware video decoder. Calling thread should have a valid CUDA context associated.

The client needs to fill in the following fields of `CUVIDDECODECAPS` before calling `cuvidGetDecoderCaps()`.

- `eCodecType`: Codec type (AV1, H.264, HEVC, VP9, JPEG etc.)
- `eChromaFormat`: enum cudaVideoChromaFormat (Monochrome, 420, 422, 444)
- `nBitDepthMinus8`: 0 for 8-bit, 2 for 10-bit, 4 for 12-bit

调用 cuvidGetDecoderCaps() 后，底层驱动会填充 CUVIDDECODECAPS 的其余字段，包括目标功能的支持情况、支持的输出格式以及硬件支持的最大 / 最小分辨率。
以下伪代码演示了如何查询 NVDEC 的功能特性：

When `cuvidGetDecoderCaps()` is called , the underlying driver fills up the remaining fields of `CUVIDDECODECAPS`, indicating the support for the queried capabilities, supported output formats and the maximum and minimum resolutions the hardware supports.

The following pseudo-code illustrates how to query the capabilities of NVDEC.

```
CUVIDDECODECAPS decodeCaps = {};
// set IN params for decodeCaps
decodeCaps.eCodecType = cudaVideoCodec_HEVC;//HEVC
decodeCaps.eChromaFormat = cudaVideoChromaFormat_420;//YUV 4:2:0
decodeCaps.nBitDepthMinus8 = 2;// 10 bit
result = cuvidGetDecoderCaps(&decodeCaps);
```

Returned parameters from API can be interpreted as below to validate if content can be decoded on underlying hardware:

```
// Check if content is supported
if (!decodecaps.bIsSupported){
       NVDEC_THROW_ERROR(Codec not supported on this GPU", CUDA_ERROR_NOT_SUPPORTED);
}
// validate the content resolution supported on underlying hardware
if ((coded_width > decodecaps.nMaxWidth) ||
        (coded_height > decodecaps.nMaxHeight)){
                    NVDEC_THROW_ERROR(Resolution  not  supported  on  this  GPU",
 CUDA_ERROR_NOT_SUPPORTED);
}
// Max supported macroblock count CodedWidth*CodedHeight/256 must be <= nMaxMBCount
if ((coded_width>>4)*(coded_height>>4) > decodecaps.nMaxMBCount){
       NVDEC_THROW_ERROR(MBCount not supported on this GPU",
CUDA_ERROR_NOT_SUPPORTED);
}
```

In most situations, bit-depth and chroma subsampling to be used at the decoder output is same as that at the decoder input (i.e. in the content). In certain cases, however, it may be necessary to have the decoder produce output with bit-depth and chroma subsampling different from that used in the input bitstream. In general, it's always a good idea to first check if the desired output bit-depth and chroma subsampling format is supported before creating the decoder. This can be done in the following way:

```
// Check supported output format
if (decodecaps.nOutputFormatMask & (1<<cudaVideoSurfaceFormat_NV12)){
        // Decoder supports output surface format NV12
}
if (decodecaps.nOutputFormatMask & (1<<cudaVideoSurfaceFormat_P010){
        // Decoder supports output surface format P010
}
……
```

The API `cuvidGetDecoderCaps()` also returns histogram related capabilities of underlying GPU. Histogram data is collected by NVDEC during the decoding process resulting in zero performance penalty. NVDEC computes the histogram data for only the luma component of decoded output, not on post-processed frame(i.e. when scaling, cropping, etc. applied). In case of AV1 when film gain is enabled, histogram data is collected on the decoded frame prior to the application of the flim grain.

```
// Check if histogram is supported
if (decodecaps.bIsHistogramSupported){
       nCounterBitDepth = decodecaps.nCounterBitDepth; // histogram counter bit depth
     nMaxHistogramBins = decodecaps.nMaxHistogramBins; // Max number of histogram bins
}
……
```

Histogram data is calculated as : `Histogram_Bin[pixel_value >> (pixel_bitDepth - log2(nMaxHistogramBins))]++;`

4.2.2. Creating a Decoder

Before creating the decoder instance, user needs to have a valid CUDA context which will be used in the entire decoding process.

The decoder instance can be created by calling `cuvidCreateDecoder()` after filling the structure `CUVIDDECODECREATEINFO`. The structure `CUVIDDECODECREATEINFO` should be filled up with the following information about the stream to be decoded:

- `CodecType:` must be from `enum cudaVideoCodec`. It represents codec type of content likeH.264, HEVC, VP9 etc.
- `ulWidth, ulHeight`: coded width and coded height in pixels.
- `ulMaxWidth, ulMaxHeight`: max width and max height that decoder support in case of resolution change. When there is resolution change (new resolution <= ulMaxWidth, ulMaxHeight) in video stream, app can reconfigure decoder using cuvidReconfigureDecoder() API instead of destroy and recreate the decoder. If ulMaxWidth or ulMaxHeight is set to 0, ulMaxWidth and ulMaxHeight are set to ulWidth and ulHeight respectively.
- `ChromaFormat:` must be from `enum cudaVideoChromaFormat.` It represents chroma format of content like 4:2:0, 4:4:4, etc.

- `bitDepthMinus8`: bit-depth minus 8 of video stream to be decoded like 0 for 8-bit, 2 for 10-bit, 4 for 12-bit.
- `ulNumDecodeSurfaces`: Referred to as decode surfaces elsewhere in this document, this is the number of surfaces that the driver will internally allocate for storing the decoded frames. Using a higher number ensures better pipelining but increases GPU memory consumption. For correct operation, minimum value is defined in `CUVIDEOFORMAT::min_num_decode_surfaces` and can be obtained from first sequence callback from Nvidia parser. The NVDEC engine writes decoded data to one of these surfaces. These surfaces are not accessible by the user of NVDECODE API, but the *mapping* stage, which includes decoder output format conversion, scaling, cropping etc.) use these surfaces as input surfaces.
- `ulNumOutputSurfaces`: This is the maximum number of output surfaces that the client will simultaneously *map* to decode surfaces for further processing using `cuvidMapVideoFrame()`. These surfaces have postprocessed decoded output to be used by client. The driver internally allocates the corresponding number of surfaces (referred as output surfaces in this document). Client will have access to output surfaces. Refer to section Preparing the decoded frame for further processing to understand the definition of *map*.
- `OutputFormat`: Output surface format defined as `enum cudaVideoSurfaceFormat`. This output format must be one of supported format obtained in `decodecaps.nOutputFormatMask` in `cuvidGetDecoderCaps()`. If an unsupported output format is passed, API will fail with error `CUDA_ERROR_NOT_SUPPORTED`.
- `ulTargetWidth, ulTargetHeight`: This is resolution of output surfaces. For use-case which involve no scaling, these should be set to `ulWidth, ulHeight`, respectively.
- `DeinterlaceMode`: This should be set to `cudaVideoDeinterlaceMode_Weave` or `cudaVideoDeinterlaceMode_Bob` for progressive content and `cudaVideoDeinterlaceMode_Adaptive` for interlaced content. `cudaVideoDeinterlaceMode_Adaptive`yields better quality but increases memory consumption.
- `ulCreationFlags`: It is defined as `enum cudaVideoCreateFlags`. It is optional to explicitly define this flag. Driver will pick appropriate mode if not defined.
- `ulIntraDecodeOnly`: Set this flag to 1 to instruct the driver that the content being decoded contains only I/IDR frames. This helps the driver optimize memory consumption. Do not set this flag if content has non-intra frames.
- `enableHistogram`: Set this flag to 1 to enable histogram data collection.

The `cuvidCreateDecoder()` call fills `CUvideodecoder` with the decoder handle which should be retained till the decode session is active. The handle needs to be passed along with other NVDECODE API calls.

The user can also specify the following parameters in the `CUVIDDECODECREATEINFO` to control the final output:

- Scaling dimension
- Cropping dimension
- Dimension if the user wants to change the aspect ratio

The following code demonstrates the setup of decoder in case of scaling, cropping, or aspect ratio conversion.

```
// Scaling. Source size is 1280x960. Scale to 1920x1080.
CUresult rResult;
unsigned int uScaleW, uScaleH;
uScaleW = 1920;
uScaleH = 1080;
...
CUVIDDECODECREATEINFO stDecodeCreateInfo;
memset(&stDecodeCreateInfo, 0, sizeof(CUVIDDECODECREATEINFO));
... // Setup the remaining structure members
stDecodeCreateInfo.ulTargetWidth  = uScaleWidth;
stDecodeCreateInfo.ulTargetHeight = uScaleHeight;
rResult = cuvidCreateDecoder(&hDecoder, &stDecodeCreateInfo);
...
```

```
// Cropping. Source size is 1280x960
CUresult rResult;
unsigned int uCropL, uCropR, uCropT, uCropB;
uCropL = 30;
uCropR = 700;
uCropT = 20;
uCropB = 500;
...
CUVIDDECODECREATEINFO stDecodeCreateInfo;
memset(&stDecodeCreateInfo, 0, sizeof(CUVIDDECODECREATEINFO));
// Setup the remaining structure members
...
stDecodeCreateInfo.display_area.left  = uCropL;
stDecodeCreateInfo.display_area.right = uCropR;
stDecodeCreateInfo.display_area.top   = uCropT;
stDecodeCreateInfo.display_are.bottom = uCropB;
rResult = cuvidCreateDecoder(&hDecoder, &stDecodeCreateInfo);
...
```

```
// Aspect Ratio Conversion. Source size is 1280x960(4:3). Convert to
// 16:9
CUresult rResult;
unsigned int uCropL, uCropR, uCropT, uCropB;
uDispAR_L = 0;
uDispAR_R = 1280;
uDispAR_T = 70;
uDispAR_B = 790;
...
CUVIDDECODECREATEINFO stDecodeCreateInfo;
memset(&stDecodeCreateInfo, 0, sizeof(CUVIDDECODECREATEINFO));
... // setup structure members
stDecodeCreateInfo.target_rect.left   = uDispAR_L;
stDecodeCreateInfo.target_rect.right  = uDispAR_R;
stDecodeCreateInfo.target_rect.top    = uDispAR_T;
stDecodeCreateInfo.target_rect.bottom = uDispAR_B;
reResult = cuvidCreateDecoder(&hDecoder, &stDecodeCreateInfo);
...
```

4.2.3. Decoding the frame/field

After de-muxing and parsing, the client can submit the bitstream which contains a frame or field of data to hardware for decoding. To accomplish this the following steps, need to be followed:

- Fill up the `CUVIDPICPARAMS` structure.

  - The client needs to fill up the structure with parameters derived during the parsing process. `CUVIDPICPARAMS` contains a structure specific to every supported codec which should also be filled up.
- Call `cuvidDecodePicture()` and pass the decoder handle and the pointer to CUVIDPICPARAMS. `cuvidDecodePicture()` kicks off the decoding on NVDEC.

4.2.4. Preparing the decoded frame for further processing

The user needs to call `cuvidMapVideoFrame()` to get the CUDA device pointer and pitch of the output surface that holds the decoded and post-processed frame.

Please note that `cuvidDecodePicture()` instructs the NVDEC hardware engine to kick off the decoding of the frame/field. However, successful completion of `cuvidMapVideoFrame()` indicates that the decoding process is completed and that the decoded YUV frame is converted from the format generated by NVDEC to the YUV format specified in `CUVIDDECODECREATEINFO::OutputFormat`.

`cuvidMapVideoFrame()` API takes decode surface index (`nPicIdx`) as input and *maps* it to one of available output surfaces, post-processes the decoded frame and copy to output surface and returns CUDA device pointer and associated pitch of the output surfaces.

The above operation performed by `cuvidMapVideoFrame()` is referred to as *mapping* in this document.

After the user is done with the processing on the frame, `cuvidUnmapVideoFrame()` must be called to make the output surface available for storing other decoded and post-processed frames.

If the user continuously fails to call the corresponding `cuvidUnmapVideoFrame()` after `cuvidMapVideoFrame()`, then `cuvidMapVideoFrame()` will eventually fail. At most `CUVIDDECODECREATEINFO::ulNumOutputSurfaces` frames can be mapped at a time.

`cuvidMapVideoFrame()` is a blocking call as it waits for decoding to complete. If `cuvidMapVideoFrame()` is called on same CPU thread as `cuvidDecodePicture()`, it will block `cuvidDecodePicture()` as well. In this case, the application will not be able to submit decode packets to NVDEC until mapping is complete. It can be avoided by performing the mapping operation on a CPU thread (referred as mapping thread) different from the one calling `cuvidDecodePicture()` (referred as decoding thread).

When using NVIDIA parser from NVDECODE API, the application can implement a producer-consumer queue between decoding thread (as producer) and mapping thread (as consumer). The queue can contain picture indexes (or other unique identifiers) for frames being decoded. Parser can run on decoding thread. Decoding thread can add the picture index to the queue in display callback and return immediately from callback to continue decoding subsequent frames as they become available. On the other side, mapping thread will monitor the queue. If it sees the queue has non-zero length, it will dequeue the entry and call `cuvidMapVideoFrame(…)` with `nPicIdx` as the picture index. Decoding thread must ensure to not reuse the corresponding decode picture buffer for storing the decoded output until its entry is consumed and freed by mapping thread.

The following code demonstrates how to use `cuvidMapVideoFrame()` and `cuvidUnmapVideoFrame()`.

```
// MapFrame: Call cuvidMapVideoFrame and get the devptr and associated
// pitch. Copy this surface (in device memory) to host memory using
// CUDA device to host memcpy.
bool MapFrame()
{
    CUVIDPARSEDISPINFO stDispInfo;
    CUVIDPROCPARAMS stProcParams;
    CUresult rResult;
    unsigned long long cuDevPtr = 0;
    int nPitch, nPicIdx, frameSize;
    unsigned char* pHostPtr = nullptr;
    memset(&stDispInfo, 0, sizeof(CUVIDPARSEDISPINFO));
    memset(&stProcParams, 0, sizeof(CUVIDPROCPARAMS));
    /*************************************************
    *   setup stProcParams
    **************************************************/
    // retrieve the frames from the Frame Display Queue. This Queue is
    // is populated in HandlePictureDisplay.
    if (g_pFrameQueue->dequeue(&stDispInfo))
    {
        nPicIdx = stDispInfo.picture_index;
        rResult = cuvidMapVideoFrame(&hDecoder, nPicIdx, &cuDevPtr,
                                     &nPitch, &stProcParams);
      frameSize = (ChromaFormat == cudaVideoChromaFormat_444) ? nPitch * (3*nheight) :
                     nPitch * (nheight + (nheight + 1) / 2);
        // use CUDA based Device to Host memcpy
        rResult = cuMemAllocHost((void** )&pHostPtr, frameSize);
        if (pHostPtr)
        {
            rResult = cuMemcpyDtoH(pHostPtr, cuDevPtr, frameSize);
        }
        rResult = cuvidUnmapVideoFrame(&hDecoder, cuDevPtr);
    }
    ... // Dump YUV to a file
    if (pHostPtr)
    {
        cuMemFreeHost(pHostPtr);
    }
    ...
}
```

In multi-instance decoding use-case, NVDEC could be bottleneck so there wouldn't be significant benefit of calling `cuvidMapVideoFrame()` and `cuvidDecodePicture()` on different CPU threads. `cuvidDecodePicture()` will stall if wait queue on NVDEC inside driver is full. Sample applications in Video Codec SDK are using *mapping* and *decode* calls on same CPU thread, for simplicity.

4.2.5. Getting histogram data buffer

Histogram data is collected by NVDEC during the decoding process resulting in zero performance penalty. NVDEC computes the histogram data for only the luma component of decoded output, not on post-processed frame(i.e. when scaling, cropping, etc. applied). In case of AV1 when film gain is enabled, histogram data is collected on the decoded frame prior to the application of the flim grain.

`cuvidMapVideoFrame()` API returns the CUDA device pointer of histogram data buffer along with output surface if `CUVIDDECODECREATEINFO::enableHistogram` flag is set while creating decoder (using API `cuvidCreateDecoder()`). CUDA device pointer of histogram buffer can be obtained from `CUVIDPROCPARAMS::histogram_dptr`.

Histogram buffer is mapped to output buffer in driver so `cuvidUnmapVideoFrame()` does unmap of histogram buffer also along with output surface.

The following code demonstrates how to use `cuvidMapVideoFrame()` and `cuvidUnmapVideoFrame()` for accessing histogram buffer.

```
// MapFrame: Call cuvidMapVideoFrame and get the output frame and associated
// histogram buffer CUDA device pointer

    CUVIDPROCPARAMS stProcParams;
    CUresult rResult;
    unsigned long long cuOutputFramePtr = 0, cuHistogramPtr = 0;
    int nPitch;
    int histogram_size = (decodecaps.nCounterBitDepth / 8) *
                            decodecaps.nMaxHistogramBins;
    unsigned char *pHistogramPtr = nullptr;

    memset(&stProcParams, 0, sizeof(CUVIDPROCPARAMS));
    /**************************************************
    *   setup stProcParams
    **************************************************/
    stProcParams.histogram_dptr = &cuHistogramPtr;

    rResult = cuvidMapVideoFrame(&hDecoder, nPicIdx, &cuOutputFramePtr,
                                 &nPitch, &stProcParams);
    // allocate histogram buffer for cuMemcpy
    rResult = cuMemAllocHost((void** )&pHistogramPtr, histogram_size);
    if (pHistogramPtr)
    {
        rResult = cuMemcpyDtoH(pHistogramPtr, cuHistogramPtr, histogram_size);
    }
    // unmap output frame
    rResult = cuvidUnmapVideoFrame(&hDecoder, cuOutputFramePtr);
    ...
}
```

4.2.6. Querying the decoding status

After the decoding is kicked off, `cuvidGetDecodeStatus()` can be called at any time to query the status of decoding of that frame. The underlying driver fills the status of decoding in `CUVIDGETDECODESTATUS::*pDecodeStatus`.

The NVDECODEAPI currently reports the following statuses:

- Decoding is in progress.
- Decoding of the frame completed successfully.
- The bitstream for the frame was corrupted and concealed by NVDEC.
- The bitstream for the frame was corrupted, however could not be concealed by NVDEC.

The API is expected to help in the scenarios where the client needs to take a further decision based on the decoding status of the frame, for e.g. whether to carry out inferencing on the frame or not.

Please note that the NVDEC can detect a limited number of errors depending on the codec. This API is supported for HEVC, H264 and JPEG on Maxwell and above generation GPUs.

4.2.7. Reconfiguring the decoder

Using `cuvidReconfigureDecoder()` the user can reconfigure the decoder if there is a change in the resolution and/or post processing parameters of the bitstream without having to destroy the ongoing decoder instance, and create a new one thereby saving time (and latency) in the process.

In the earlier SDKs the user had to destroy the existing decoder instance and create a new decoder instance for handling any change in decoder resolution or post processing parameters (like scaling ratio, cropping dimensions etc.).

The API can be used in scenarios where the bitstream undergoes changes in resolution, for e.g. when the encoder (on server side) changes image resolution frequently to adhere to Quality of Service(QoS) constraints.

The following steps need to be followed for using the `cuvidReconfigureDecoder()`.

1. The user needs to specify `CUVIDDECODECREATEINFO::ulMaxWidth` and `CUVIDDECODECREATEINFO::ulMaxHeight` while calling `cuvidCreateDecoder()`. The user should choose the values of `CUVIDDECODECREATEINFO::ulMaxWidth` and `CUVIDDECODECREATEINFO::ulMaxHeight` which to ensure that the resolution of the bitstream is never exceeded during the entire decoding process. Please note that the values of `CUVIDDECODECREATEINFO::ulMaxWidth` and `CUVIDDECODECREATEINFO::ulMaxHeight` cannot be changed within a session and if user wants to change the values, the decoding session should be destroyed and recreated.
2. During the process of decoding, when the user needs to change the bitstream or change postprocessing parameters, the user needs to call `cuvidReconfigureDecoder()`. This call should be ideally made from `CUVIDPARSERPARAMS::pfnSequenceCallback` when the bitstream changes. The parameters the user wants to reconfigure should be filled up in `::CUVIDRECONFIGUREDECODERINFO`. Please note, `CUVIDRECONFIGUREDECODERINFO::ulWidth` and `CUVIDRECONFIGUREDECODERINFO::ulHeight` must be equal to or smaller than `CUVIDDECODECREATEINFO::ulMaxWidth` and `CUVIDDECODECREATEINFO::ulMaxHeight` respectively or else the `cuvidReconfigureDecoder()` would fail.

The API is supported for all codecs supported by NVDECODEAPI.

4.2.8. Destroying the decoder

The user needs to call `cuvidDestroyDecoder()` to destroy the decoder session and free up all the allocated decoder resources.

4.3. Run-time dynamic linking of Nvidia libraries

Video Codec SDK sample applications are using two main Nvidia libraries: nvcuvid and cuda. Both libraries can be used as either load-time dynamic linking or run-time dynamic linking. Video Codec SDK sample applications are using load-time dynamic linking. User can use run-time dynamic linking of these libraries if needed. Below code snippets can help understand the changes needed in programming style:

4.3.1. Run-time dynamic linking

In case of run-time dynamic linking, library is loaded to memory at run-time. Below is code snippet to dynamically load nvcuvid library at run-time on Windows and Linux systems:

```
#if defined(WIN32) || defined(_WIN32) || defined(WIN64) || defined(_WIN64)
#include <Windows.h>

#ifdef UNICODE
   static LPCWSTR __DriverLibName = L"nvcuvid.dll";
#else
   static LPCSTR __DriverLibName = "nvcuvid.dll";
#endif

typedef HMODULE DLLDRIVER;

static CUresult LOAD_LIBRARY(DLLDRIVER *pInstance)
{
    *pInstance = LoadLibrary(__DriverLibName);

    if (*pInstance == NULL)
    {
        printf("LoadLibrary \"%s\" failed!\n", __DriverLibName);
        return CUDA_ERROR_UNKNOWN;
    }

    return CUDA_SUCCESS;
}

#elif defined(__unix__) || defined(__APPLE__) || defined(__MACOSX)

#include <dlfcn.h>

static char __DriverLibName[] = "libnvcuvid.so";

typedef void *DLLDRIVER;

static CUresult LOAD_LIBRARY(DLLDRIVER *pInstance)
{
    *pInstance = dlopen(__DriverLibName, RTLD_NOW);

    if (*pInstance == NULL)
    {
        printf("dlopen \"%s\" failed!\n", __DriverLibName);
        return CUDA_ERROR_UNKNOWN;
    }

    return CUDA_SUCCESS;
}
#endif
```

4.3.2. Getting function pointers

Function pointers can be fetched using `GetProcAddress()` on Windows and `dlsym()` on Linux:

```
typedef     CUresult     CUDAAPI     tcuvidCreateVideoParser(CUvideoparser     *pObj,
 CUVIDPARSERPARAMS *pParams);
```

```
typedef CUresult CUDAAPI tcuvidParseVideoData(CUvideoparser obj, CUVIDSOURCEDATAPACKET
 *pPacket);
typedef CUresult CUDAAPI tcuvidDestroyVideoParser(CUvideoparser obj);

typedef CUresult CUDAAPI tcuvidGetDecoderCaps(CUVIDDECODECAPS *pdc);
typedef    CUresult    CUDAAPI    tcuvidCreateDecoder(CUvideodecoder    *phDecoder,
 CUVIDDECODECREATEINFO *pdci);
typedef CUresult CUDAAPI tcuvidDestroyDecoder(CUvideodecoder hDecoder);
typedef CUresult CUDAAPI tcuvidDecodePicture(CUvideodecoder hDecoder, CUVIDPICPARAMS
 *pPicParams);

tcuvidCreateVideoParser              *cuvidCreateVideoParser;
tcuvidParseVideoData                 *cuvidParseVideoData;
tcuvidDestroyVideoParser             *cuvidDestroyVideoParser;

tcuvidGetDecoderCaps                 *cuvidGetDecoderCaps;
tcuvidCreateDecoder                  *cuvidCreateDecoder;
tcuvidDestroyDecoder                 *cuvidDestroyDecoder;
tcuvidDecodePicture                  *cuvidDecodePicture;

#if defined(WIN32) || defined(_WIN32) || defined(WIN64) || defined(_WIN64)
#define GET_PROC_EX(name, alias, required)                     \
    alias = (t##name *)GetProcAddress(DriverLib, #name);               \
    if (alias == NULL && required) {                                    \
        printf("Failed to find required function \"%s\" in %s\n",       \
               #name, __DriverLibName);                                  \
        return CUDA_ERROR_UNKNOWN;                                      \
    }

#elif defined(__unix__) || defined(__APPLE__) || defined(__MACOSX)
#define GET_PROC_EX(name, alias, required)                              \
    alias = (t##name *)dlsym(DriverLib, #name);                        \
    if (alias == NULL && required) {                                    \
        printf("Failed to find required function \"%s\" in %s\n",       \
               #name, __DriverLibName);                                  \
        return CUDA_ERROR_UNKNOWN;                                      \
    }
#endif


#define GET_PROC_REQUIRED(name) GET_PROC_EX(name,name,1)
#define GET_PROC_OPTIONAL(name) GET_PROC_EX(name,name,0)
#define GET_PROC(name)          GET_PROC_REQUIRED(name)

#define CHECKED_CALL(call)              \
    do {                                \
        CUresult result = (call);       \
        if (CUDA_SUCCESS != result) {   \
            return result;              \
        }                               \
    } while(0)

CUresult CUDAAPI cuvidInit(unsigned int Flags)
{
    DLLDRIVER DriverLib;

    CHECKED_CALL(LOAD_LIBRARY(&DriverLib));

    // fetch all function pointers
    GET_PROC(cuvidCreateVideoParser);
    GET_PROC(cuvidParseVideoData);
    GET_PROC(cuvidDestroyVideoParser);

    GET_PROC(cuvidGetDecoderCaps);
    GET_PROC(cuvidCreateDecoder);
    GET_PROC(cuvidDestroyDecoder);
    GET_PROC(cuvidDecodePicture);
```

```
    // fetch other functions pointers

    return CUDA_SUCCESS;
}
```

4.4. Writing an Efficient Decode Application

The NVDEC engine on NVIDIA GPUs is a dedicated hardware block, which decodes the input video bitstream in supported formats. A typical video decode application broadly consists of the following stages:

1. De-Muxing
2. Video bitstream parsing and decoding
3. Preparing the frame for further processing

Of these, de-muxing and parsing are not hardware accelerated and therefore outside the scope of this document. The de-muxing can be performed using third party components such as FFmpeg, which provides support for many multiplexed video formats. The sample applications included in the SDK demonstrate de-muxing using FFmpeg.

Similarly, post-decode or video post-processing (such as scaling, color space conversion, noise reduction, color enhancement etc.) can be effectively performed using user-defined CUDA kernels.

The post-processed frames can then be sent to the display engine for displaying on the screen, if required. Note that this operation is outside the scope of NVDECODE APIs.

An optimized implementation should use independent threads for de-muxing, parsing, bitstream decode and processing etc. as explained below:

1. De-muxing: This thread demultiplexes the media file and makes the raw bit-stream available for parser to consume.
2. Parsing and decoding: This thread does the parsing of the bitstream and kicks off decoding by calling `cuvidDecodePicture()`.
3. Mapping and making the frame available for further processing: This thread checks if there are any decoded frames available. If yes, then it should call `cuvidMapVideoFrame()` to get the CUDA device pointer and pitch of the frame. The frame can then be used for further processing.

The NVDEC driver internally maintains a queue of 4 frames for efficient pipelining of operations. Please note that this pipeline does not imply any decoding delay for decoding. The decoding starts as soon as the first frame is queued, but the application can continue queuing up input frames so long as space is available without stalling. Typically, by the time application has queued 2-3 frames, decoding of the first frame is complete and the pipeline continues. This pipeline ensures that the hardware decoder is utilized to the maximum extent possible.

For performance intensive and low latency video codec applications, ensure the PCIE link width is set to the maximum available value. PCIE link width currently configuredcan be obtained by running command 'nvidia-smi -q'. PCIE link width can be configured in the system's BIOS settings.

In the use cases where there is frequent change of decode resolution and/or post processing parameters, it is recommended to use `cuvidReconfigureDecoder()` instead of destroying the existing decoder instance and recreating a new one.

The following steps should be followed for optimizing video memory usage:

1. Make `CUVIDDECODECREATEINFO::ulNumDecodeSurfaces = CUVIDEOFORMAT::min_num_decode_surfaces`. This will ensure that the underlying driver allocates minimum number of decode surfaces to correctly decode the sequence. In case there is reduction in decoder performance, clients can slightly increase `CUVIDDECODECREATEINFO::ulNumDecodeSurfaces`. It is therefore recommended to choose the optimal value of `CUVIDDECODECREATEINFO::ulNumDecodeSurfaces` to ensure right balance between decoder throughput and memory consumption.
2. `CUVIDDECODECREATEINFO::ulNumOutputSurfaces` should be decided optimally after due experimentation for balancing decoder throughput and memory consumption.
3. `CUVIDDECODECREATEINFO::DeinterlaceMode` should be set "`cudaVideoDeinterlaceMode::cudaVideoDeinterlaceMode_Weave`" or "`cudaVideoDeinterlaceMode::cudaVideoDeinterlaceMode_Bob`". For interlaced contents, choosing `cudaVideoDeinterlaceMode::cudaVideoDeinterlaceMode_Adaptive` results to higher quality but increases memory consumption. Using `cudaVideoDeinterlaceMode::cudaVideoDeinterlaceMode_Weave` or `cudaVideoDeinterlaceMode::cudaVideoDeinterlaceMode_Bob` results to minimum memory consumption though it may result in lesser video quality. In case "`CUVIDDECODECREATEINFO::DeinterlaceMode`" is not specified by the client, the underlying display driver sets it to "`cudaVideoDeinterlaceMode::cudaVideoDeinterlaceMode_Adaptive`" which results to higher memory consumption. Hence it is strongly recommended to choose the right value of `CUVIDDECODECREATEINFO::DeinterlaceMode` depending on the requirement.
4. While decoding multiple streams it is recommended to allocate minimum number of CUDA contexts and share it across sessions. This saves the memory overhead associated with the CUDA context creation.
5. `CUVIDDECODECREATEINFO::ulIntraDecodeOnly` should be set to 1 if it is known beforehand that the sequence contains Intra frames only. This feature is supported only for HEVC, H.264 and VP9. However, decoding might fail if the flag is enabled in case of supported codecs for regular bit streams having P and/or B frames.

The sample applications included with the Video Codec SDK are written to demonstrate the functionality of various APIs, but they may not be fully optimized. Hence programmers are strongly encouraged to ensure that their application is well-designed, with various stages in the decode-postprocess-display pipeline structured in an efficient manner to achieve desired performance and memory consumption.

4.5. Allocating decode surfaces dynamically

The bitstream sequence (SPS) often has more DPB size than necessary. In most instances, not all of these surfaces are required. Therefore, the following approach can be used to reduce the number of decode surfaces used during a decode session. This will help lower the memory footprint of each session in most instances, allowing for an increase in the number of simultaneous decode sessions. However, the decode throughput of individual sessions may be slightly reduced, as the surfaces will be reused more frequently.

The following steps should be followed for further optimizing video memory usage:

1. Set `CUVIDPARSERPARAMS::bMemoryOptimize = 1` before invoking `cuvidCreateVideoParser()`. This will ensure that the parser sends lowest possible `CUVIDPICPARAMS::CurrPicIdx`.
2. `pfnSequenceCallback()` has `min_num_decode_surfaces` information. Return `min_num_decode_surfaces` value from `pfnSequenceCallback()`. This will overwrite DPB size to `min_num_decode_surfaces`.
3. Assign a value to `CUVIDDECODECREATEINFO::ulNumDecodeSurfaces` that is less than `min_num_decode_surfaces`, with the minimum value being 1. Next, create a decoder using `cuvidCreateDecoder()`. This will create a decoder with a reduced number of decode surfaces.
4. In `pfnDecodeCallback()`, check the value of `CUVIDPICPARAMS::CurrPicIdx` sent by the parser.

    a). If `CUVIDPICPARAMS::CurrPicIdx` is less than `ulNumDecodeSurfaces`, continue with the decode using `cuvidDecodePicture()` API.

    b). If `CUVIDPICPARAMS::CurrPicIdx` is greater than or equal to `ulNumDecodeSurfaces`, then invoke `cuvidReconfigureDecoder()` with increased `CUVIDRECONFIGUREDECODERINFO::ulNumDecodeSurfaces` to increase the number of decode surfaces. Then, proceed with decoding the frame using `cuvidCreateDecoder()`. Note: If `cuvidReconfigureDecoder()` fails with `CUDA_ERROR_OUT_OF_MEMORY` error, application can retry for some time, if `cuvidReconfigureDecoder()` continues to fail, the application needs to handle the failure.

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