NVIDIA VIDEO DECODER INTERFACE

PG-08085-001_v08 | Feb 2017

Programming Guide

DOCUMENT CHANGE HISTORY

PG-08085-001_v08

Version	Date	Authors	Description of Change
1.0	2016/6/10	VU/CC	Initial release
2.0	2017/2/15	SM	Update for SDK8.0

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. OVERVIEW

NVIDIA GPUs - beginning with the 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: H.264, HEVC (H.265), VP8, VP9, MPEG-1, MPEG-2, MPEG-4 and VC-1. NVDEC runs completely independent of compute/graphics engine.

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 compute and graphics.

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. Decoded video frames can either be presented to the display with graphics interoperability for video playback, or frames can be passed directly to a dedicated hardware encoder (NVENC) for high-performance video transcoding.

1.1 CODECS SUPPORT

The API is supported on Windows and Linux and works in conjunction with NVIDIA's CUDA, graphics, and encoder capabilities. The codecs supported by NVDECODE API are:

- MPEG-1,
- MPEG-2,
- MPEG4,
- VC-1,
- H.264 (AVCHD),
- H.265 (HEVC)
- VP8,
- VP9.

Please 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
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
Kepler (GK1xx)	Maximum Resolution: 4080x4080	Maximum Resolution: 2048x1024 1024x2048	Maximum Resolution: 4096x4096 Profile: Main, High profile up to Level 4.1	Unsupported	Unsupported	Unsupported
Maxwell Gen 1 (GM10x)	Maximum Resolution: 4080x4080	Maximum Resolution: 2048x1024 1024x2048	Maximum Resolution: 4096x4096 Profile: Baseline, Main, High profile up to Level 5.1	Unsupported	Unsupported	Unsupported

GPU Architecture	MPEG-1 & MPEG-2	VC-1 & MPEG-4	H.264/AVCHD	H.265/HEVC	VP8	VP9
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 Level 5.1	Unsupported	Maximum Resolution: 4096x4096	Unsupported
GM206	Maximum Resolution: 4080x4080	Maximum Resolution: 2048x1024 1024x2048	Maximum Resolution: 4096x4096 Profile: Baseline, Main, High profile up to Level 5.1	Maximum Resolution: 4096x2304 Profile: Main profile up to Level 5.1 and main10 profile	Maximum Resolution: 4096x4096	Maximum Resolution: 4096x2304 Profile: Profile 0
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
GP10x	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 5.1, main10 and main12 profile	Maximum Resolution: 4096x4096[1]	Maximum[2] Resolution: 8192x8192 Profile: Profile 0, 10-bit and 12-bit decoding

[1] Supported only on GP104

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

Chapter 3. VIDEO PLAYBACK AND DECODER PIPELINE

Sample applications[3] `NvDecodeD3D9` (DirectX 9), `NvDecodeD3D11` (DirectX 11) and `NvDecodeGL` (OpenGL on Windows and Linux), included in the SDK package, demonstrate the following functions in a typical video playback application:

1. Parsing the video input source.
2. Querying the decode capabilities.
3. Decoding video bitstream on GPU using NVDECODE API.
4. Converting decoded YUV surface NV12/P016 format to RGBA.
5. Mapping RGBA surface to DirectX 9.0 or OpenGL texture.
6. Drawing texture to screen.

Sample application `NvTranscoder` included in the SDK package demonstrates how to set up an end-to-end video transcode pipeline using NVDECODE and NVENCODE APIs, with following functions:

1. Parsing the video input source.
2. Decoding video bitstream on GPU using NVDECODE API.
3. Sending YUV video frame to hardware video encoder using the NVENCODE API.
4. Getting back compressed video bitstream in system memory for further processing.

[3] Location: ./Samples in the Video Codec SDK package

3.1 DECODER PIPELINE

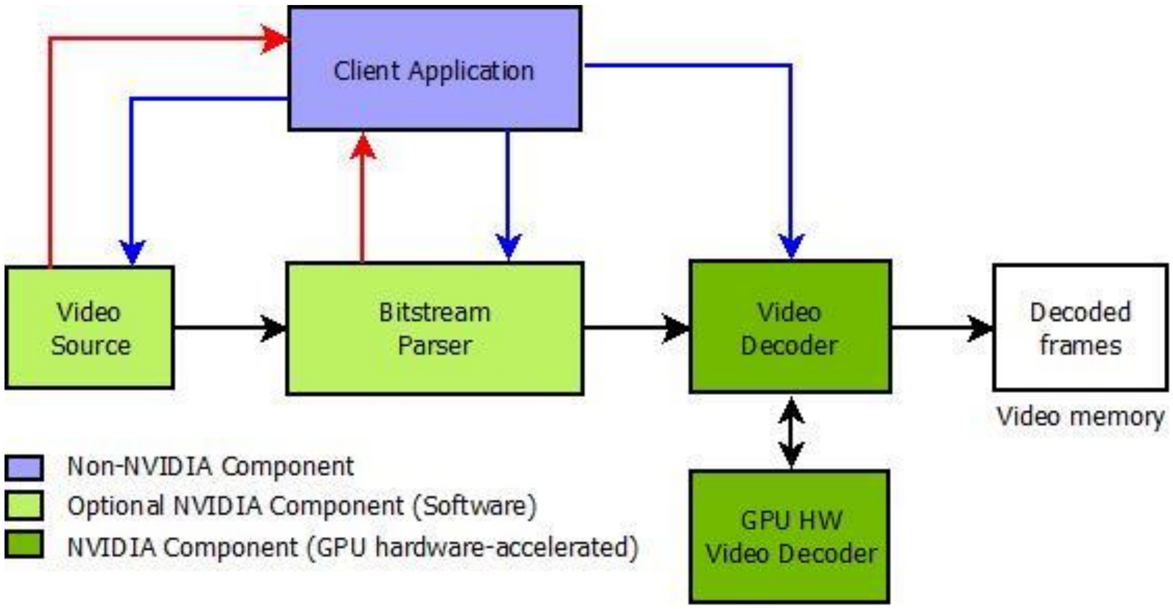

Figure 1. Video decoder pipeline using NVDECODE API

Figure 1 shows the decoder pipeline using NVDECODE API. The solid black lines show the data flow between modules. The solid colored lines represent process flow.

1. The application thread (referred to as the primary thread) calls `cuvidCreateVideoSource()`, which spawns a de-multiplexer thread (referred to as the secondary thread).
2. The primary thread (colored in blue above) calls `cuvidCreateVideoParser()` to create the parser. It also creates the decoder by calling `cuvidCreateDecoder()`.
3. The secondary thread (colored in red above) makes the following callbacks given that the function pointers are not NULL. The callbacks are serial:
   a) Handle video data: The callback implementation calls `cuvidParseVideoData()` to parse the video data.
   b) Handle video sequence: The callback is made when there is sequence change.
   c) Handle picture decode: The callback implementation calls `cuvidDecodePicture()` to decode the frame.
   d) Handle Picture Display: The callback implementation signals the primary thread to display the picture.
4. The primary thread calls `cuvidMapVideoFrame()` to get the pitch and CUDA device pointer to the surface which contains the decoded/post-processed frame. Thereafter it calls `cuvidUnmapVideoFrame()` as the complimentary operation.

5. The primary thread destroys the resources by calling `cuvidDestroyDecoder()`, `cuvidDestroyVideoParser()` and `cuvidDestroyVideoSource()`.

The sample applications use all three components - Video Source, Video Parser, and Video Decoder. The components are not dependent on each other and hence can be used independently. The user can unplug the Video Source and Video Parser and plug-in an implementation of his own. In this document we will be concentrating particularly on the Video Decoder (colored dark green in Figure 1) and the stages following decode (format conversion and display using OpenGL or DirectX). It is highly recommended that the user use his/her own implementation for Video Source and Video Parser. These two components are neither hardware-accelerated nor software-optimized and users may want to have their own customized parsers which have better performance.

Chapter 4. NVIDIA VIDEO DECODER (NVDECODE) API

The NVDECODE API consists of two main header-files: `dynlink_cuviddec.h` and `dynlink_nvcuvid.h`. The samples in NVIDIA Video Codec SDK dynamically load the library functions and only include `dynlink_cuviddec.h` and `dynlink_nvcuvid.h` in the source files. These headers can be found under `./Samples/common/inc` folder in the Video Codec SDK package. 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.

4.1 VIDEO DECODER APIS

The Video Decoder API consists of the following main functions:

```
// Queries the functionalities exposed through NVDECODE API
CUresult cuvidGetDecoderCaps(CUVIDDECODECAPS *pdc);

// Create the Decoder Object
CUresult cuvidCreateDecoder(CUvideodecoder *phDecoder,
                            CUVIDDECODECREATEINFO *pdci);
// Destroy the Decoder Object
CUresult cuvidDestroyDecoder(CUvideodecoder hDecoder);

// Decode a single picture (field or frame)
CUresult cuvidDecodePicture(CUvideodecoder hDecoder,
                            CUVIDPICPARAMS *pPicParams);

// Post-Process and map a video frame for use in CUDA
CUresult cuvidMapVideoFrame(CUvideodecoder hDecoder, int PicIdx,
                            unsigned int* pDevPtr. unsigned int*
```

```
                             pPitch, CUVIDPROCPARAMS* pVPP);

// Unmap the previously mapped video frame
CUresult cuvidUnmapVideoFrame(CUvideodecoder hDecoder, unsigned int
DevPtr
```

4.2 QUERYING DECODE CAPABAILITIES

The API `cuvidGetDecoderCaps()` lets users query the capabilities of underlying hardware video decoder. As illustrated in Table 1, different GPUs have hardware decoders with different capabilities. Therefore, in order to ensure your application works on all generations of GPU hardware, it is highly recommended that the application is written to query the hardware capabilities and take appropriate decision based on presence/absence of the desired capability/functionality.

The sample applications demonstrate how to use the API `cuvidGetDecoderCaps`() to query the functionalities exposed by the NVDECODE API.

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

- `eCodecType`: Codec type (H.264, HEVC, VP9 etc.)
- `eChromaFormat`: 4:2:0, 4:4:4, etc.
- `nBitDepthMinus8`: 0 for 8-bit, 2 for 10-bit, 4 for 12-bit

When `cuvidGetDecoderCaps`() is called, the driver fills up the remaining fields of `CUVIDDECODECAPS`, indicating the support for the queried codec profile, supported resolutions etc. Please refer to the structure definition of `CUVIDDECODECAPS` structure for more information.

4.3 CREATING A DECODER

The sample applications use the API `cuvidCreateDecoder()` through a C++ wrapper class `VideoDecoder` defined in `VideoDecoder.h`. The class's constructor is a good starting point to see how to set up the structure `CUVIDDECODECREATEINFO` for `cuvidCreateDecoder()` method. Most importantly, the structure `CUVIDDECODECREATEINFO` contains the following information about the stream to be decoded:

- `CodecType:` H.264, HEVC, VP9 etc.
- Frame size: Values of `ulWidth` and `ulHeight`
- `ChromaFormat:` 4:2:0, 4:4:4, etc.

- Bit depth: 0 for 8-bit, 2 for 10-bit, 4 for 12-bit

The user also specifies various properties of the output that the decoder should generate:

- Output surface format (User needs to specify `cudaVideoSurfaceFormat_NV12` or `cudaVideoSurfaceFormat_P016` for 8-bit or 10/12 bit contents respectively).
- Output frame size
- Maximum number of output surfaces: This is the maximum number of surfaces that the client code will simultaneously map for display.
- Maximum number of surfaces the decoder may allocate for decoding.

The following pseudo-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 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 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);
...
```

4.4 DECODING SURFACES

The classes `VideoSource` and `VideoParser` wrap the calls to Video Source and Video Parser components of Figure 1. The `VideoParser` class implements three callback functions, two of which are explained below:

```
// called by the video parser to decode a single picture. Since the
// parser will deliver data as fast as it can, we need to make sure
// that the picture index we're attempting to use for decode is no
// longer used for display.
static int CUDAAPI HandlePictureDecode(void *pUserData,
                                       CUVIDPICPARAMS *pPicParams);

// called by the video parser to display a video frame (in case of
```

```
// field pictures, there may be two decode calls per one display call,
// since two fields make up one frame)
static int CUDAAPI HandlePictureDisplay(void *pUserData,
                                       CUVIDPARSERDISPINFO *pPicParams);
```

`VideoParser` passes a `CUVIDPICPARAMS` structure to the callback which can be passed without any modifications to the function `cuvidDecodePicture()`. The `CUVIDPICPARAMS` structure contains all the information necessary for the decoder to decode a frame or field. In particular, it contains pointers to the video bitstream, information about frame size, flags denoting whether it's a field or a frame, bottom or top field, etc.

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.

The implementation of `HandlePictureDecode()` in the sample application waits if the output queue is full. When a slot in the queue becomes available, it simply invokes the `cuvidDecodePicture()` function, passing the `pPicParams` as received from the parser.

The `HandlePictureDisplay()` method is passed a `CUVIDPARSERDISPINFO` structure which contains the necessary data for displaying a frame; i.e. frame index of the decoded frame (as given to the decoder), and some information relevant for display such as frame time, field information, etc. The parser calls this method for frames in the order that they should be displayed.

The implementation of `HandlePictureDisplay()` method in the sample application simply enqueues the `pPicParams` passed by the parser into the `FrameQueue` object.

The `FrameQueue` is used to implement a producer-consumer pattern for passing frames (or better, references to decoded frames) between the VideoSource's decoding thread and the application's main thread, which is responsible for displaying them on the screen.

4.5 PROCESSING AND DISPLAYING FRAMES

The user needs to call `cuvidmapVideoFrame()` to get the CUDA device pointer and pitch of the surface that has the decoded frame. The following is a pseudo-code that demonstrates using `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 int cuDevPtr; int nPitch, nPicIdx;
    unsigned char* pHostPtr;

    memset(&stDispInfo, 0, sizeof(CUVIDPARSEDISPINFO));
    memset(&stProcParams, 0, sizeof(CUVIDPROCPARAMS));

    ... // setup stProcParams if required

    // 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);

        // use CUDA based Device to Host memcpy
        pHostPtr = cuMemAllocHost((void** )&pHostPtr, nPitch);
        if (pHostPtr)
        {
            rResult = cuMemcpyDtoH(pHostPtr, cuDevPtr, nPitch);
        }
        rResult = cuvidUnmapVideoFrame(&hDecoder, cuDevPtr);
    }

    ... // Dump YUV to a file

    if (pHostPtr)
    {
        cuMemFreeHost(pHostPtr);
```

```
    }
    ...
}
```

The function `copyDecodedFrameToTexture()` in `videoDecode.cpp` does something more than the above pseudo-code. It retrieves the frame (decoded surface) from `FrameQueue` as above. Uses `cuvidMapVideoFrame()` to get the CUDA device pointer and the associated pitch of the decoded surface. It maps a D3D/OGL texture to be used by CUDA (interop surface). It then calls `cudaPostProcessFrame()` to do the color space conversion from NV12 to RGBA. The texture holds the RGBA surface. This texture can now be drawn to the screen.

The following list summaries the function calls involved (refer sample apps) in the display and post-process pipeline:

1. `cuvidMapVideoFrame` – gets a CUDA device pointer from decoded frame of a Video Decoder (using map).
2. `cuD3D9ResourceGetMappedPointer` – For cudaDecodeD3D9, this function retrieves a CUDA device pointer from a D3D9 texture.
3. `cuGLMapBufferObject` – For cudaDecodeGL, this function retrieves a CUDA device pointer from an OpenGL PBO (Pixel Buffer Object).
4. `cudaPostProcessFrame` – calls all subsequent CUDA post-process functions on that frame, and writes the result directly to the Mapped D3D texture.
5. `cuD3D9UnmapResources` – For `NvDecodeD3D9`, the CUDA driver will release the pointer back to the D3D9 driver. This tells the Direct3D driver that CUDA is finished modifying the resource, and that it is safe to use it with D3D9.
6. `cuGLUnmapBufferObject` – For `NvDecodeGL`, the CUDA driver will release the pointer back to the OpenGL driver. This tells the OpenGL driver that CUDA is finished modifying the resource, and that it is safe to use it with OpenGL.
7. `cuvidUnmapVideoFrame` – Unmap the previously mapped frame.

4.6 WRITING AN EFFICIENT DECODE-DISPLAY 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 and display application consists of the following stages:

1. Video bitstream parser
2. Video decoder
3. Post-processor
4. Screen display

Of these, 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.

The sample applications included with the Video Codec SDK are written to demonstrate the functionality of various APIs but they are by no means fully optimized applications. In fact, 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.

As a starting point, an optimized implementation may make use of independent threads for bitstream decode and display as follows:

1. **Decode Thread**: This thread calls `cuvidDecodePicture()` and pushes the decoded frame to the display queue. This continues as long as there are frames to decode.
2. **Display thread**: This thread reads the display queue and checks if there are any decoded frames. If yes, then it calls `cuvidMapVideoFrame()` to get the CUDA device pointer and pitch of the frame. The resulting CUDA device pointer can be used for CUDA post-processing of the decoded video frames using user-defined CUDA kernels. Finally, it is necessary to call `cuvidUnMapVideoFrame()` so that the decoded frame buffer is unmapped by the driver. This continues as long as there are decoded frames in the display queue and end of decode has not been reached. The display thread presents the contents of the post processed video frame to an OpenGL or Direct3D surface, using CUDA interoperability.

To ensure that the video frames will playback without stuttering or hitching, it is necessary to ensure that decode and display threads do not get blocked. Two or more D3D9/D3D11 or OpenGL surfaces allows double or triple buffered playback. This allows both decode and display to run on different surfaces without being blocked.