# NVIDIA VIDEO DECODER (NVCUVID) INTERFACE

DA-05614-001_v8.0 | November 2015

**Programming Guide**

# TABLE OF CONTENTS

# Chapter 1.
# OVERVIEW

The NVIDIA Video Decoder (NVCUVID) API enables developers to use the hardware video decode capabilities with the ability to interoperate video with compute and graphics. The NVCUVID API supports the following video codec formats: MPEG-2, VC-1, H.264 (*AVCHD*), and H.265 (*HEVC*). Refer to Chapter 2 for the complete details about video capabilities for each GPU architecture.

Applications that use this API can decode compressed video streams directly to video memory. With frames in video memory, post processing can be done with CUDA kernels. Additionally, the NVIDIA CUDA driver can allow the dedicated copy engines to be used for asychronous I/O transfers between video memory and system memory. Decoded video frames can either be presented to the display with graphics interop for video playback, or frames can be passed directly to a dedicated hardware encoder (NVENC) for video transcoding.

# Chapter 2.
# VIDEO DECODER CAPABILITIES

## 2.1. Hardware Video Decoder Capabilities

| GPU Architecture | MPEG-2 | VC-1 | H.264/AVCHD | H.265/HEVC |
|---|---|---|---|---|
| Fermi (GF1xx) | Max Resolution: 4080x4080 | Max Resolution: 2048x1024 1024x2048 | Max Resolution: 4096x4096 Profile: Baseline, Main, High profile up to Level 4.1, Max Bitrate: 45mbps | Unsupported |
| Kepler (GK1xx) | Max Resolution: 4080x4080 | Max Resolution: 2048x1024 1024x2048 | Max Resolution: 4096x4096 Profile: Main, High profile up to Level 4.1 | Unsupported |
| Maxwell Gen 1 - (GM10x) | Max Resolution: 4080x4080<br>Performance: 8HD | Max Resolution: 2048x1024 1024x2048<br>Performance: 7.5HD, 6.5HD (Interlaced), Max bitrate: 60Mbps | Max Resolution: 4096x4096 Profile: Baseline, Main, High profile up to Level 5.1, Performance: 8HD for 15Mbps, 6HD when 40Mbps | Unsupported |
| Maxwell Gen 2 - (GM20x) | Max Resolution: 4080x4080<br>Performance: 8HD | Max Resolution: 2048x1024 1024x2048<br>Performance: 7.5HD, 6.5HD (Interlaced), Max bitrate: 60Mbps | Max Resolution: 4096x4096 Profile: Baseline, Main, High profile up to Level 5.1, Performance: 8HD for 15Mbps, 6HD when 40Mbps | Unsupported |
| Maxwell Gen 2 - (GM206) | Max Resolution: 4080x4080<br>Performance: 8HD | Max Resolution: 2048x1024 1024x2048<br>Performance: 7.5HD, 6.5HD | Max Resolution: 4096x4096 Profile: Baseline, Main, High profile up to Level 5.1, | Max Resolution: 4096*2304 Profile: Main profile up to Level 5.1, Performance: |

| GPU Architecture | MPEG-2 | VC-1 | H.264/AVCHD | H.265/HEVC |
|---|---|---|---|---|
| | | (Interlaced), Max bitrate: 60Mbps | Performance: 8HD, 4kx2k 60fps @ 120Mbps | 8HD 4kx2k 60fps @160Mbps, 5.1 high tier |

# Chapter 3.
# INTRODUCTION

This NVIDIA Video Decoder (NVCUVID) API allows developers access the video decoding features of NVIDIA graphics hardware. This API is supported with multiple OS platforms and works in conjunction with NVIDIA's CUDA, Graphics, and Encoder capabilities.

The NVIDIA Video Decoder API interoperates with CUDA, OpenGL, and Direct3D. It supports CUDA fast memory copies between video memory and system memory. Now it is possible to implement a fully hardware accelerated video pipeline where all video stages are running directly on GPU. For video playback applications, video frames can be processed with compute or graphics playback. For video transcoding applications, frames can be sent directly to a dedicated hardware encoder (NVENC) for video encoding.

The two decode samples **`NvDecodeD3D9`** (DirectX) and **`NvDecodeGL`** (OpenGL on windows and linux) demonstrate a hardware accelerated video playback pipeline:

1. Parse a video input source (using NVIDIA Video Decoder API)
2. Decode video on GPU using *NVCUVID* API.
3. Convert decoded surface *NV12* format to *RGBA*.
4. Map RGBA surface to DirectX 9.0 or OpenGL surface.
5. Draw texture to screen.

A single transcode example **`NvTranscoder`** for windows and linux demonstrates a pipeline for video transcode:

1. Parse a video input source (using NVIDIA Video Decoder API)
2. Decode video on GPU using *NVCUVID* API.
3. Send YUV video frame to encoding using the *NVENC* API.
4. Receive a compressed video bitstream back to the host.

This document will focus on the use of the NVIDIA Video Decoder API (NVCUVID) and the stages following decode, (i.e. format conversion and display using DirectX or OpenGL). Parsing of the video source using the NVCUVID API is secondary to the sample. While NVCUVID has a built in video parser, most developers will already have code to parse video streams down to the slice-level. The NVCUVID API can be used hardware picture level decoding.

# Chapter 4.
# NVIDIA VIDEO DECODER (NVCUVID)

The NVIDIA Video Decode (NVCUVID) API consists of a header-file: **`cuviddec.h`** and **`nvcuvid.h`** lib-file: **`nvcuvid.lib`** located in CUDA toolkit include files location. For the samples in the NVIDIA Video SDK, the samples dynamically load the library functions and only require that you include **`dynlink_cuviddec.h`** and **`dynlink_nvcuvid.h`** in your source projects. These headers can be found at **`./common/inc`** folder.
The Windows DLLs **`nvcuvid.dll`** ship with NVIDIA display drivers. The Linux **`libnvcuvid.so`** is included with Linux drivers (R260+).

This API defines five function entry points for decoder creation and use:

```
// Create/Destroy the decoder object
CUresult cuvidCreateDecoder(CUvideodecoder *phDecoder,
                            CUVIDDECODECREATEINFO *pdci);

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 nPicIdx,
                            unsigned int *pDevPtr, unsigned int *pPitch,
                            CUVIDPROCPARAMS *pVPP);

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

## 4.1. Decoder Creation

The sample application uses this API through a C++ Wrapper class **`VideoDecoder`** defined in **`VideoDecoder.h`**. The class's constructor is a good starting point to see how to setup the **`CUVIDDECODECREATEINFO`** for the **`cuvidCreateDecoder()`** method. Most importantly, the create-info contains the following information about the stream that's going to be decoded:

1. codec-type
2. the frame-size
3. chroma format

The user also determines various properties of the output that the decoder is to generate:

1. Output surface format (currently only *NV12* supported)
2. Output frame size
3. Maximum number of output surfaces. This is the maximum number of surfaces that the client code will simultaneously map for display.

The user also needs to specify the maximum number of surfaces the decoder may allocate for decoding.

## 4.2. Decoding Surfaces

The decode sample application is driven by the *VideoSource* class, which spawns its own thread. The source calls a callback on the *VideoParser* class to parse a stream. The VideoParser in turn calls back into two callbacks that handle the decode and display of the frames.

The parser thread calls two callbacks to decode and display frames:

```
// 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 the 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);
```

The NVIDIA VideoParser passes a **`CUVIDPICPARAMS`** struct to the callback which can be passed straight on to the **`cuvidDecodePicture()`** function. The **`CUVIDPICPARAMS`** struct contains all the information necessary for the decoder to decode a frame or field; in particular pointers to the video bitstream, information about frame size, flags if field or frame, bottom or top field, etc.

The decoded result gets associated with a picture-index value in the **`CUVIDPICPARAMS`** struct, 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`** struct which contains the necessary data for displaying a frame; i.e. the frame-index of the decoded frame (as given to the decoder), and some information relevant for display like frame-time, field, etc. The parser calls this method for frames in the order as 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 passing frames (or better, references to decoded frames) between the VideoSource's decoding thread and the application's main thread, which is responsible for their screen display.

## 4.3. Processing and Displaying Frames

The application's main loop retrieves images from the *FrameQueue* (**`copyDecodedFrameToTexture()`** in **`videoDecode.cpp`**) and renders the texture to the screen. The *DirectX* device is set up to block on monitor *vsync*, throttling rendering to 60Hz for the typical flat-screen display. To handle frame rate conversion of 3:2 pulldown content, we also render the frame multiple-times, according to the repeat information passed from the parser.

**`copyDecodedFrameToTexture()`** is the method where the CUDA Decoder API is used to map a decoded frame (based on its *Picture-Index*) into CUDA device memory.

Post processing on a frame is done by mapping the frame through **`cudaPostProcessFrame()`**. This returns a pointer to a frame decoded as a *NV12* surface. This then gets passed to a CUDA kernel which converts the NV12 surface to a *RGBA* surface. The final RGBA surface is then copied directly to either a DirectX or OpenGL texture and then drawn to the screen.

## 4.4. Performance Optimizations for NVIDIA Video Decoding

The two Samples (**`NvDecodeGL`** and **`NvDecodeD3D9`**) are intended for simplicity and understanding of how to use this API. It is by no means a fully optimized application. This NVCUVID library makes use two different engines on the GPU, the Video Processor for video decode, and the Compute Cores for colorspace conversion and display. The Video Processor and the Compute Cores can be run concurrently. The display thread for this sample is as follows:

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 done 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 done modifying the resource, and that it is safe to use it with OpenGL.
7. **`cuvidUnmapVideoFrame`** (Decoded Frame)

For optimal performance, use two or more D3D9 or OpenGL surfaces to ping/pong between them. This allows the driver to work on both decode and display without waiting.