NVIDIA CUDA VIDEO DECODER

DA-05614-001_v6.5 | August 2014

Reference Guide

TABLE OF CONTENTS

Chapter 1. OVERVIEW

The CUDA Video Decoder API gives developers access to hardware video decoding capabilities on NVIDIA GPU. The actual hardware decode can run on either Video Processor (*VP*) or CUDA hardware, depending on the hardware capabilities and the codecs. This API supports the following video stream formats for Linux and Windows platforms: MPEG-2, VC-1, and H.264 (*AVCHD*).

Applications can decode video streams on the GPU into Video Memory, and also apply a post processing stage on the uncompressed video surfaces with CUDA. CUDA supports I/O transfers of video surfaces to system memory using CUDA's fast asynchronous uploads and read-backs. For display, CUDA supports native graphics interoperability, enabling the application to render the video generated surfaces using a 3D API such as OpenGL (Windows and Linux) or DirectX (Windows platforms).

Chapter 2. VIDEO DECODE

2.1. MPEG-2/VC-1 Support

- Decode Acceleration for G8x, G9x (Requires Compute 1.1 or higher)
- Full Bitstream Decode for MCP79, MCP89, G98, GT2xx, GF1xx, GK1xx
- MPEG-2 CUDA accelerated decode with a GPU with 8+ SMs (64 CUDA cores) (Windows)
- Supports HD (1080i/p) playback including Blu-ray Disc™ content
- R185+ (Windows), R260+ (Linux)

2.2. H.264/AVCHD Support

- Baseline, Main, and High Profile, up to Level 4.1
- Full Bitstream Decoding in hardware including HD (1080i/p) Bluray content
- Supports B-Frames, bitrates up to 45 mbps
- Available on NVIDIA GPUs: G8x, G9x, MCP79, MCP89, G98, GT2xx, GF1xx, GK1xx
- R185+ (Windows), R260+ (Linux)

Chapter 3. INTRODUCTION

This CUDA Video Decoder API allows developers access the video decoding features of NVIDIA graphics hardware. This OS platform independent API is an extension to NVIDIA's CUDA technology.

The CUDA Video Decoder API is based on CUDA, it inherits all of CUDA's interoperability capabilities with OpenGL, Direct3D, and the CUDA support for fast memory copies between video memory and system memory. It is now possible to implement a video playback pipeline from video decode to image post-processing with CUDA all running entirely on the GPU. With transcode applications, this API allows the video bitstream decode to be fully offloaded to the GPU's video processor. The decoded frames can be passed to a CUDA accelerated video encoding stage through your GPU accelerated encoder or the NVIDIA CUDA encoder.

The NVIDIA CUDA Samples application (windows only) implements the following playback pipeline:

1. Parse the Video input Source (using CUDA Video Decoder API)
2. Decode Video on GPU using *NVCUVID* API.
3. Convert decoded surface (*NV12* format or *YUV 4:2:0* format) to *RGBA*.
4. Map RGBA surface to DirectX 9.0 or OpenGL surface.
5. Draw texture to screen.

This document will focus on the use of the CUDA Video Decoder API 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, as we believe most developers will already have code for parsing video streams down to the slice-level. Note: The low level decode APIs are supported on both Linux and Windows platforms. The NVCUVID APIs for *Parsing* and *Source Stream* input are available only on Windows platforms.

Chapter 4. CUDA VIDEO DECODER

The CUDA Video Decode API consists of a header-file: **cuviddec.h** and **nvcuvid.h** lib-file: **nvcuvid.lib** located in CUDA toolkit include files location. 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,
                            CUdeviceptr *pDevPtr, unsigned int *pPitch,
                            CUVIDPROCPARAMS *pVPP);

// Unmap a previously mapped video frame
CUresult cuvidUnmapVideoFrame(CUvideodecoder hDecoder, CUdeviceptr 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 (Windows only), which spawns its own thread. The source calls a callback on the *VideoParser* class (Windows only) to parse the stream, the VideoParser in turn calls back into two callbacks that handle the decode and display of the frames. For Linux platforms, you will need to write your own video source and parsing functions that connect to the Video Decoding functions.

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 CUDA 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 *NV12* decoded frame. This then gets passed to a CUDA kernel to convert NV12 surface to a *RGBA* surface. The final RGBA surface is then copied directly into a DirectX texture and then drawn to the screen.

4.4. Performance Optimizations for Video Decoding

The CUDA Samples (**`cudaDecodeGL`** and **`cudaDecodeD3D9`**) are intended for simplicity and understanding of how to use this API case. It is by no means a fully optimized application. This CUDA Video Decoder library makes use two different engines on the GPU, the Video Processor and the Graphics hardware (CUDA and 3D). This allows the Video Processor and the Graphics hardware to run asynchronously. The display thread for this sample is this:

1. **`cuvidMapVideoFrame`** – gets a CUDA device pointer from decoded frame of a Video Decoder (using map)
2. **`cuD3D9ResourceGetMappedPointer`** – gets a CUDA device pointer from a D3D texture
3. **`cudaPostProcessFrame`** – calls all subsequent CUDA post-process functions on that frame, and writes the result directly to the Mapped D3D texture.
4. **`cuD3D9UnmapResources`** – driver will release pointer back to *D3D9*. This tells the *Direct3D* driver that CUDA is done modifying it, and that it is safe to use for D3D9.
5. **`cuvidUnmapVideoFrame`** (Decoded Frame)

To improve performance, having 2 or more D3D9 or OpenGL surfaces to ping/pong can improve performance. This enables the driver to schedule workload without blocking the display thread.