NVIDIA VIDEO DECODER INTERFACE

NVDECODEAPI_PG-08085-001_v06 | Aug 2019

Programming Guide

DOCUMENT CHANGE HISTORY

NVDECODEAPI_PG-08085-001_v06

Version	Date	Authors	Description of Change
1.0	2016/6/10	VU/CC	Initial release
2.0	2017/2/15	SM	Update for SDK 8.0
3.0	2018/1/10	SM	Update for SDK 8.1
4.0	2018/4/10	SM	Update for SDK 8.2
5.0	2019/1/10	SM	Update for SDK 9.0
6.0	2019/8/10	SM	Update for SDK 9.1

TABLE OF CONTENTS

LIST OF TABLES

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: 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 other engines on the GPU.

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 can be passed directly to a dedicated hardware encoder (NVENC) for high-performance video transcoding or can also be used for GPU accelerated inferencing.

1.1 SUPPORTED CODECS

The codecs supported by NVDECODE API are:

- MPEG-1,
- MPEG-2,
- MPEG4,
- VC-1,
- H.264 (AVCHD) (8 bit),
- H.265 (HEVC) (8bit, 10 bit and 12 bit),
- VP8,
- VP9(8bit, 10 bit and 12 bit).
- Hybrid (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
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/GV100/ Turing	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 and Turing
[2] VP9 10-bit and 12-bit decoding is supported on select GP10x GPUs

Chapter 3. VIDEO DECODER PIPELINE

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).
4. De-Mux the content (like .mp4). This can be done using third party software like FFMPEG.
5. Parse the video bitstream using third party parser like FFMPEG.
6. Kick off the Decoding using NVDECODE API.
7. Obtain the decoded YUV for further processing.
8. Query the status of the decoded frame.
9. Depending on the decoding status, use the decoded output for further processing like rendering, inferencing, postprocessing etc.
10. If the application needs to display the output,
    a. Convert decoded YUV surface to RGBA.
    b. Map RGBA surface to DirectX or OpenGL texture.
    c. 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)

All NVDECODE APIs are exposed in two header-files: `cuviddec.h` and `nvcuvid.h`. These headers can be found under `..\Samples\NvCodec\NvDecoder` 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 QUERYING DECODE CAPABILITIES

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, 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.

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

- `eCodecType`: Codec type (H.264, HEVC, VP9, JPEG 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 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);

rResult = cuvidCreateDecoder(&hDecoder, &stDecodeCreateInfo);
```

4.2 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`: 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 `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 decoding output:

- Output surface format (User needs to specify `cudaVideoSurfaceFormat_NV12` or `cudaVideoSurfaceFormat_P016` for 8-bit or 10/12-bit contents respectively).
- Scaling dimension
- Cropping dimension
- Dimension if the user wants to change the aspect ratio
- `ulNumDecodeSurfaces`: This is the number of surfaces that the client will use for storing the decoded frames. Using a higher number ensures better pipelining but

increases GPU memory consumption. The driver internally allocates the corresponding number of surfaces. The NVDEC engine outputs decoded data to one of these surfaces.

- `ulNumOutputSurfaces`: This is the maximum number of surfaces that the client will simultaneously *map* for further processing. The driver internally allocates the corresponding number of surfaces. Refer to section 4.4 to understand the definition of *map*.
- Flags to be used during decoder creation (`ulCreationFlags`)
- Control for memory optimization for I/IDR frame only decode (`ulIntraDecodeOnly`)

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 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);
...
```

4.3 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:

1. 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.

2. Call `cuvidDecodePicture()` and pass the decoder handle and the pointer to `CUVIDPICPARAMS`. `cuvidDecodePicture()` kicks off the decoding on NVDEC.

4.4 PREPARING THE DECODED FRAME FOR FURTHER PROCESSING

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

Please note, `cuvidDecodePicture()` instructs the NVDEC engine to kick off the decoding of the frame/field. However, successful completion of `cuvidMapVideoFrame()` ensures that the decoding process is completed; the decoded YUV frame is converted from the format generated by NVDEC to the YUV format exposed in NVDECODE API and made available for further processing.

The `nPicIdx` passed in `cuvidMapVideoFrame`() specifies the surface index where the decoded output will be saved for further processing.

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

After the user is done with the processing on the frame, `cuvidUnmapVideoFrame()` needs to be called to make the surface corresponding to `nPicIdx` availabe for storing other decoded frames.

If the user continuously fails to call the corresponding `cuvidUnmapVideoFrame()` after `cuvidMapVideoFrame`(), then `cuvidMapVideoFrame()` will eventually fail.

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 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);
    }
    ...
}
```

4.5 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. The support of this API is currently available only for HEVC, H264 and JPEG.

4.6 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 should 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.

4.7 DESTROYING THE DECODER

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

4.8 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. It is important to call `cuvidDecodePicture` () and `cuvidMapVideoFrame`() in separate threads to obtain maximum throughput.

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 configured can 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.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2011-2019 NVIDIA Corporation. All rights reserved.