Using FFmpeg with NVIDIA GPU Hardware Acceleration

User Guide

vDA-08430-001_v02 | October 2021

Table of Contents

Chapter 1. Introduction

All NVIDIA® GPUs starting with Kepler generation support fully-accelerated hardware video encoding and decoding. The hardware encoder and hardware decoder are referred to as NVENC and NVDEC, respectively, in the rest of the document.

The hardware capabilities of NVENC and NVDEC are exposed in the NVIDIA Video Codec SDK through APIs (herein referred to as NVENCODE API and NVDECODE API), by which the user can access the hardware acceleration abilities of NVENC and NVDEC.

FFmpeg is the most popular multimedia transcoding software and is used extensively for video and audio transcoding. NVENC and NVDEC can be effectively used with FFmpeg to significantly speed up video decoding, encoding, and end-to-end transcoding.

This document explains ways to accelerate video encoding, decoding and end-to-end transcoding on NVIDIA GPUs through FFmpeg which uses APIs exposed in the NVIDIA Video Codec SDK.

Chapter 2. Setup

2.1. Hardware Setup

FFmpeg with NVIDIA GPU acceleration requires a system with Linux or Windows operating system and a supported NVIDIA GPU.

For a list of supported GPUs, refer to https://developer.nvidia.com/nvidia-video-codec-sdk .

For the rest of this document, it is assumed that the system being used has a GPU which has both NVENC and NVDEC.

2.2. Software Setup

2.2.1. Prerequisites

FFmpeg supports both Windows and Linux. FFmpeg has been compiled and tested with Microsoft Visual Studio 2013 SP2 and above (Windows), MinGW (msys2-x86_64-20161025) (Windows) and gcc 4.8 and above (Linux) compilers.

FFmpeg requires separate git repository nvcodec-headers for NV-accelerated ffmpeg build.

To compile FFmpeg, the CUDA toolkit *must* be installed on the system, though the CUDA toolkit is not needed to *run* the FFmpeg compiled binary.

Before using FFmpeg, it is recommended to refer to the FFmpeg documentation, note the version of the Video Codec SDK it uses, and ensure that the minimum driver required for that version of the Video Codec SDK is installed.

2.2.2. Compiling FFmpeg

FFmpeg is an open-source project. Download the FFmpeg source code repository and compile it using an appropriate compiler.

More Information on building FFmpeg can be found at: https://trac.ffmpeg.org/wiki/CompilationGuide

2.2.2.1. Compiling for Linux

FFmpeg with NVIDIA GPU acceleration is supported on all Linux platforms.

To compile FFmpeg on Linux, do the following:

- Clone ffnvcodec

```
git clone https://git.videolan.org/git/ffmpeg/nv-codec-headers.git
```

- Install ffnvcodec

```
cd nv-codec-headers && sudo make install && cd –
```

- Clone FFmpeg's public GIT repository.

```
git clone https://git.ffmpeg.org/ffmpeg.git ffmpeg/
```

- Install necessary packages.

```
sudo apt-get install build-essential yasm cmake libtool libc6 libc6-dev unzip wget
 libnuma1 libnuma-dev
```

- Configure

```
./configure --enable-nonfree --enable-cuda-nvcc --enable-libnpp --extra-cflags=-I/usr/
local/cuda/include --extra-ldflags=-L/usr/local/cuda/lib64 --disable-static --enable-
shared
```

- Compile

```
make -j 8
```

- Install the libraries.

```
sudo make install
```

2.2.2.2. Compiling for Windows

FFmpeg with NVIDIA GPU acceleration is supported on all Windows platforms, with compilation through Microsoft Visual Studio 2013 SP2 and above, and MinGW. Depending upon the Visual Studio Version and CUDA SDK version used, the paths specified may have to be changed accordingly.

To compile FFmpeg on Windows, do the following:

- Install msys2 from www.msys2.org.
- Clone ffnvcodec

```
git clone https://git.videolan.org/git/ffmpeg/nv-codec-headers.git
```

- Clone FFmpeg's public GIT repository.

```
git clone https://git.ffmpeg.org/ffmpeg.git
```

- Create a folder named nv_sdk in the parent directory of FFmpeg and copy all the header files from C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v8.0\include and library files from C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v8.0\lib\x64 to nv_sdk folder.
- Launch the Visual Studio x64 Native Tools Command Prompt.

- From the Visual Studio x64 Native Tools Command Prompt, launch the MinGW64 environment by running `mingw64.exe` from the msys2 installation folder.
- In the MinGW64 environment, install the necessary packages.

```
pacman -S diffutils make pkg-config yasm
```

- Add the following paths by running the commands.

```
export PATH="/c/Program Files (x86)/Microsoft Visual Studio 12.0/VC/BIN/amd64/":$PATH
export PATH="/c/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v8.0/bin/":$PATH
```

- Goto nv-codec-headers directory and install ffnvcodec

```
make install PREFIX=/usr
```

- Go to the FFmpeg installation folder and run the following command.

```
./configure --enable-nonfree -disable-shared --enable-cuda-nvcc --enable-libnpp --
toolchain=msvc --extra-cflags=-I../nv_sdk --extra-ldflags=-libpath:../nv_sdk
```

- Compile the code by executing the following command.

```
make -j 8
```

2.2.2.3. Compiling for Windows Subsystem for Linux (WSL)

FFmpeg with NVIDIA GPU acceleration is supported on WSL. To compile FFmpeg on WSL, follow the steps as specified in Compiling for Linux.

2.2.2.4. Commonly faced issues and tips to resolve them

- Common compilation issues
- 1. FFmpeg TOT may be broken at times. Please check out a release version if it is broken, or use an older snapshot.
  2. Make sure you are using mingw64 for a 64-bit system. Using mingw32 would result in errors such as - "`Relocation truncated to fit - R_X86_64_32`".
  3. Msys (and not Msys2) cannot launch the mingw64 command shell. It can only launch the mingw32 shell.
  4. Make sure cuda.h is available in /usr/local/cuda/include along with SDK header files. It is required for enabling NVCUVID, otherwise the configuration will lead to an error - "`CUDA Not found`".
  5. Not specifying `--extra-ldflags` in the correct format will lead to error - argument not recognized.
  6. nvcc from CUDA toolkit version 11.0 or higher does not support compiling for 'compute_30'. This will cause the `configure` script to fail with the message `'ERROR: failed checking for nvcc.'`. To work around this error, use `--nvccflags="-gencode arch=compute_52,code=sm_52 -O2"` which overrides the default nvcc flags.
- Common run-time issues
- 1. Use `-vsync 0` option with decode to prevent FFmpeg from creating output YUV with duplicate and extra frames.

2. Msys2 gives errors such as - "`Libbz2-1.dll missing from your computer`" while running FFmpeg. Workaround for this error - Copy all DLLs under C:\msys64\mingw64\bin in the folder where ffmpeg.exe is present.

Chapter 3. Basic Testing

Once the FFmpeg binary with NVIDIA hardware acceleration support is compiled, hardware-accelerated video transcode should be tested to ensure everything works well. To automatically detect NV-accelerated video codec and keep video frames in GPU memory for transcoding, the ffmpeg cli option "-hwaccel cuda -hwaccel_output_format cude" is used in further code snippets.

3.1. 1:1 HWACCEL Transcode without Scaling

The following command reads file *input.mp4* and transcodes it to *output.mp4* with H.264 video at the same resolution and with the same audio codec.

```
ffmpeg -y -vsync 0 -hwaccel cuda -hwaccel_output_format cuda -i input.mp4 -c:a copy
 -c:v h264_nvenc -b:v 5M output.mp4
```

3.2. 1:1 HWACCEL Transcode with Scaling

The following command reads file *input.mp4* and transcodes it to *output.mp4* with H.264 video at 720p resolution and with the same audio codec. The following command uses the built in resizer in cuvid decoder.

```
ffmpeg -y -vsync 0 -hwaccel cuda -hwaccel_output_format cuda –resize 1280x720 -i
 input.mp4 -c:a copy -c:v h264_nvenc -b:v 5M output.mp4
```

There is a built-in cropper in cuvid decoder as well. The following command illustrates the use of cropping. (-crop (top)x(bottom)x(left)x(right))

```
ffmpeg -y -vsync 0 -hwaccel cuda -hwaccel_output_format cuda –crop 16x16x32x32 -i
 input.mp4 -c:a copy -c:v h264_nvenc -b:v 5M output.mp4
```

Alternately scale_cuda or scale_npp resize filters could be used as shown below

```
ffmpeg -y -vsync 0 -hwaccel cuda -hwaccel_output_format cuda -i input.mp4 -vf
 scale_cuda=1280:720 -c:a copy -c:v h264_nvenc -b:v 5M output.mp4
ffmpeg -y -vsync 0 -hwaccel cuda -hwaccel_output_format cuda -i input.mp4 -vf
 scale_npp=1280:720 -c:a copy -c:v h264_nvenc -b:v 5M output.mp4
```

3.3. 1:N HWACCEL Transcode with Scaling

The following command reads file *input.mp4* and transcodes it to two different H.264 videos at various output resolutions and bit rates. Note that while using the GPU video encoder and decoder, this command also uses the scaling filter (*scale_npp*) in FFmpeg for scaling the decoded video output into multiple desired resolutions. Doing this ensures that the memory transfers (system memory to video memory and vice versa) are eliminated, and that transcoding is performed with the highest possible performance on the GPU hardware.

Input: *input.mp4*

Outputs: *1080p, 720p (audio same as input)*

```
ffmpeg -y -vsync 0 -hwaccel cuda -hwaccel_output_format cuda -i input.mp4
-vf scale_npp=1920:1080 -c:a copy -c:v h264_nvenc -b:v 5M output1.mp4
-vf scale_npp=1280:720 -c:a copy -c:v h264_nvenc -b:v 8M output2.mp4
```

3.4. 1:N HWACCEL encode from YUV or RAW Data

Encode from YUV or RAW Files can result in disk I/O being bottleneck and it is advised to do such encodes from an SSD to get maximum performance. The following command reads file *input.yuv* and encodes it to four different H.264 videos at various output bit rates. Note that this command results in a single YUV load only for all encode operations, resulting in more efficient disk I/O to improve the overall encode performance.

Input: *input.yuv (420p, 1080p)*

Outputs: *1080p (8M), 1080p (10M), 1080p (12M), 1080p (14M)*

```
ffmpeg  -y  -vsync  0  -pix_fmt  yuv420p  -s  1920x1080  -i  input.yuv  -filter_complex
 "[0:v]hwupload_cuda,split=4[o1][o2][o3][o4]"  -map  "[o1]"  -c:v  h264_nvenc  -b:v  8M
 output1.mp4  -map  "[o2]"  -c:v  h264_nvenc  -b:v  10M  output2.mp4  -map  "[o3]"  -c:v
 h264_nvenc -b:v 12M output3.mp4 -map "[o4]" -c:v h264_nvenc -b:v 14M output4.mp4
```

The pixel format (pix_fmt) should be changed to yuv444p/p010/yuv444p16 for encoding YUV 444, 420-10 and 444-10 files respectively.

3.5. Multiple 1:N HWACCEL Transcode with Scaling

This method should be used to realize the full potential of GPU hardware-accelerated transcoding. One of the typical workloads for transcoding consists of videos being transcoded and archived at different resolutions and bitrates so that they can be served to different clients later. The following command reads file *input1.mp4* as the input, decodes it in GPU hardware, scales the input in hardware, and re-encodes as H.264 videos to *output11.mp4* at 480p and *output12.mp4* at 240p using the GPU hardware encoder. Simultaneously it reads file *input2.mp4* and transcodes it to *output21.mp4* at 720p and *output22.mp4* at 480p as H.264 videos. These are achieved using a single command line.

Input: *input1.mp4, input2.mp4*

Output: *480p 240p (from input1.mp4), 720p. 480p (from input2.mp4) (audio same as input)*

```
ffmpeg -y -hwaccel cuda -hwaccel_output_format cuda -i input1.mp4
-hwaccel cuda -hwaccel_output_format cuda -i input2.mp4
-map 0:0 -vf scale_npp=640:480 -c:v h264_nvenc -b:v 1M output11.mp4
-map 0:0 -vf scale_npp=320:240 -c:v h264_nvenc -b:v 500k output12.mp4
-map 1:0 -vf scale_npp=1280:720 -c:v h264_nvenc -b:v 3M output21.mp4
-map 1:0 -vf scale_npp=640:480 -c:v h264_nvenc -b:v 2M output22.mp4
```

3.6. Multiple 1:N Transcode with Scaling (SW Decode->HW Scaling->HW Encode)

In some situations, it is necessary to perform video decoding in software. For example, consider the situation in which the hardware encoder has more capacity than the decoder. To realize the full potential of the encoder hardware in such cases, it is beneficial to run part of the decode workload in hardware (until the hardware decoder saturates), and the rest in software.

The following command reads file *input1.mp4*, decodes it in software, scales the input in hardware, and transcodes it to *output11.mp4* at 480p and *output12.mp4* at 240p as H.264 videos and simultaneously reads file *input2.mp4* and transcodes it to *output21.mp4* at 720p and *output22.mp4* at 480p as H.264 videos.

Input: *input1.mp4, input2.mp4*

Output: *480p 240p (from input1.mp4), 720p. 480p (from input2.mp4) (audio same as input)*

```
ffmpeg -y -init_hw_device cuda=foo:bar -filter_hw_device foo \
 -i input1.mp4 -i input2.mp4 \
-map 0:0 -vf hwupload,scale_npp=640:480 -c:v h264_nvenc -b:v 1M \
    output11.mp4 \
-map 0:0 -vf hwupload,scale_npp=320:240 -c:v h264_nvenc -b:v 500k \
    output12.mp4 \
-map 1:0 -vf hwupload,scale_npp=1280:720 -c:v h264_nvenc -b:v 2M \
    output21.mp4 \
-map 1:0 -vf hwupload,scale_npp=640:480 -c:v h264_nvenc -b:v 1M \
    output22.mp4
```

Chapter 4. Quality Testing

Once basic FFmpeg setup is confirmed to be working properly, other options provided on the FFmpeg command line can be used to test encoding, decoding, and transcoding.

This chapter lists FFmpeg commands for accelerating video encoding, decoding, and transcoding using NVENC and NVDEC.

4.1. Video Encoding

The quality of encoded video depends on various features in use by the encoder. To encode a 720p YUV, use the following command.

```
ffmpeg -y -vsync 0 –s 1280x720 –i input.yuv -c:v h264_nvenc output.mp4
```

This generates the output file in MP4 format (output.mp4) with H264 encoded video.

Video encoding can be broadly classified into two types of use cases:

- **Latency tolerant high quality**: In these kind of use cases latency is permitted. Encoder features such as B-frames, look-ahead, reference B frames, variable bitrate (VBR) and higher VBV buffer sizes can be used. Typical use cases include cloud transcoding, recording and archiving, etc.
- **Low latency**: In these kind of use cases latency should be low and can be as low as 16 ms. In this mode, B-frames are disabled, constant bitrate modes are used, and VBV-buffer sizes are kept very low. Typical use cases include real-time gaming, live streaming and video conferencing etc. This encoding mode results in a lower encoding quality due to the above constraints.

NVENCODEAPI supports several features for adjusting quality, performance, and latency which are exposed through the FFmpeg command line. It is recommended to enable the feature(s) and command line option(s) depending on the use case.

4.2. Video Decoding

The FFmpeg video decoder is straightforward to use. To decode an input bitstream from *input.mp4*, use the following command.

```
ffmpeg -y -vsync 0 -c:v h264_cuvid -i input.mp4 output.yuv
```

This generates the output file in NV12 format (*output.yuv*).

To decode multiple input bitstreams concurrently within a single FFmpeg process, use the following command.

```
ffmpeg -y -vsync
            0 -hwaccel cuda -hwaccel_output_format cuda -i input1.264 -hwaccel cuda
 -hwaccel_output_format cuda -i
              input2.264 -hwaccel cuda -hwaccel_output_format cuda -i input3.264 -
filter_complex
                    "[0:v]hwdownload,format=nv12[o0];[1:v]hwdownload,format=nv12[o1];
[2:v]hwdownload,format=nv12[o2]"
            -map "[o0]" -f rawvideo output1.yuv -map "[o1]" -f rawvideo output2.yuv
 -map "[o2]"
          -f rawvideo output3.yuv
```

This uses a separate thread per decode operation, a single Cuda context shared among all threads and generates the output files in NV12 format (outputN.yuv).

4.3. Command Line for Latency-Tolerant High-Quality Transcoding

Input: *input.mp4*

Output: *same resolution as input, bitrate = 5M (audio same as input)*

- Slow Preset

```
ffmpeg -y -vsync 0 -hwaccel cuda -hwaccel_output_format cuda -i input.mp4 -c:a copy
 -c:v h264_nvenc -preset p6 -tune hq -b:v 5M -bufsize 5M -maxrate 10M -qmin 0 -g 250
 -bf 3 -b_ref_mode middle -temporal-aq 1 -rc-lookahead 20 -i_qfactor 0.75 -b_qfactor
 1.1 output.mp4
```

- Medium Preset

Use -preset p4 instead of -preset p6 in the above command line.

- Fast Preset

Use -preset p2 instead of -preset p6 in the above command line.

4.4. Command Line for Low Latency Transcoding

Input: *input.mp4 (30fps)*

Output: *same resolution as input, bitrate = 5M (audio same as input)*

- Low Latency High Quality

```
ffmpeg -y -vsync 0 -hwaccel cuda -hwaccel_output_format cuda -i input.mp4 -c:a copy
 -c:v h264_nvenc -preset p6 -tune ll -b:v 5M -bufsize 5M -maxrate 10M -qmin 0 -g 250
 -bf 3 -b_ref_mode middle -temporal-aq 1 -rc-lookahead 20 -i_qfactor 0.75 -b_qfactor
 1.1 output.mp4
```

- Low Latency High performance

Use -preset p2 instead of -preset p6 in above command line.

Chapter 5. Advanced Quality Settings

5.1. Lookahead

Lookahead improves the video encoder's rate-control accuracy by enabling the encoder to buffer the specified number of frames, estimate their complexity, and allocate the bits appropriately among these frames proportional to their complexity. This typically results in better quality because the encoder can distribute the bits proportional to the complexity over a larger number of frames. The number of lookahead frames should be at least the number of B frames + 1 to avoid CPU stalls. A lookahead of 10-20 frames is suggested for optimal quality benefits.

To enable lookahead, use the `-rc-lookahead N` (`N` = number of frames) option on FFmpeg command line.

5.2. Adaptive Quantization (AQ)

This feature improves visual quality by adjusting the encoding quantization parameter (QP) (on top of the QP evaluated by the rate control algorithm) based on spatial and temporal characteristics of the sequence. NVENC supports two flavors of AQ which are explained below. AQ internally uses CUDA for complexity estimation which may have a slight impact on the performance and graphics engine utilization.

Spatial AQ

Spatial AQ mode adjusts QP values based on spatial characteristics of the frame. Since the low complexity flat regions are visually more perceptible to quality differences than high complexity detailed regions, extra bits are allocated to flat regions of the frame at the cost of the regions having high spatial detail. Although Spatial AQ improves the perceptible visual quality of the encoded video, the required bit redistribution results in peak signal-to-noise ratio (PSNR) drop in most cases. Therefore, during PSNR-based evaluation, this feature should be turned off. The spatial AQ algorithm can be controlled by specifying the aq-strength parameter that controls the variations in QP values, with larger values bringing more QP variations. AQ strength ranges from 1-15.

To enable spatial AQ, use -spatial-aq 1 option on FFmpeg command line, and -aq-strength 8 (can range from 1 to 15). If no value is specified, the strength is auto selected by the driver.

Temporal AQ

Temporal AQ attempts to adjust the encoding quantization parameter (QP) (on top of QP evaluated by the rate control algorithm) based on temporal characteristics of the sequence. Temporal AQ improves the quality of encoded frames by adjusting QP for regions which are constant or have low motion across frames but have high spatial detail, such that they become a better reference for future frames. Allocating extra bits to such regions in reference frames is better than allocating them to the residuals in referred frames because it helps improve the overall encoded video quality. If most of the region within a frame has little or no motion but has high spatial details (e.g. high-detail non-moving background), enabling temporal AQ will benefit the most.

One of the potential disadvantages of temporal AQ is that enabling temporal AQ may result in high fluctuation of bits consumed per frame within a GOP. I/P-frames will consume more bits than average P-frame size, and B-frames will consume fewer bits. Although the target bitrate will be maintained at the GOP level, the frame size will fluctuate from one frame to the next within a GOP more than it would without temporal AQ. Enabling temporal AQ is not recommended if a strict CBR profile is required for every frame size within a GOP. To enable temporal AQ, use the -temporal_aq 1 option on the FFmpeg command line.

Chapter 6. Performance Evaluation and Optimization

Various factors affect the performance of hardware accelerated transcoding on the GPU. Getting the highest performance for your workload requires some tuning. This section provides some tips for measuring and optimizing end-to-end transcode performance.

NVIDIA Video Codec SDK documentation publishes performance of GPU hardware accelerated encoder and decoder as stand-alone numbers, measured using high-performance encode or decode application included in the SDK. Although FFmpeg software is highly optimized, its performance is slightly lower than the performance reported in the SDK documentation, mainly due to software overheads and additional setup/initialization time within FFmpeg code. Therefore, to get high transcoding throughput using FFmpeg, it is essential to saturate the hardware encoder and decoder engines such that the initialization time overhead for one session gets hidden behind the transcoding time of other sessions. This can be achieved by running multiple parallel encode/decode sessions on the hardware (see Section 1:N HWACCEL encode from YUV or RAW Data). In such a case, the *aggregate* transcode performance with FFmpeg matches closely with the theoretically expected hardware performance.

6.1. Measuring Aggregate Performance

To measure GPU hardware accelerated aggregate performance, follow the steps below:

1. Run multiple simultaneous sessions (say 4 FFmpeg sessions) in parallel, each performing transcoding.
2. Ensure the inputs have large number of frames (more than 15 seconds of video is recommended) so that initialization time overhead can be ignored.
3. Measure the time required by each transcode.
4. Derive the aggregate performance in terms of frames per second (FPS).

6.2. Settings for Reduced Initialization Time

To prepare longer videos for streamed distribution, they are typically split into smaller chunks and each chunk is encoded separately. Such chunk-based encoding avoids error propagation, provides clean boundaries for streaming bandwidth adaptation and helps parallelizing transcoding workloads on the servers. Transcoding smaller video chunks using GPU-hardware-accelerated transcoding, however, poses a challenge because the initialization time overhead of each FFmpeg process becomes significant.

To minimize the overhead when transcoding *M* input files into *MN* output files (i.e. when each of the *M* inputs is transcoded into *N* outputs), it is better to minimize the number of FFmpeg processes launched (see Section 1:N HWACCEL encode from YUV or RAW Data for example command lines).

Additionally, follow these tips to reduce the FFmpeg initialization time overhead:

- Set the following environment variables:

```
export CUDA_VISIBLE_DEVICES=0  // (Use ID for the GPU device which you plan to use
 for transcode)
export CUDA_DEVICE_MAX_CONNECTIONS=2
```

- Use FFmpeg command lines such as those in Sections 1:N HWACCEL Transcode with Scaling and 1:N HWACCEL encode from YUV or RAW Data. These command lines share the CUDA context across multiple transcode sessions, thereby reducing the CUDA context initialization time overhead significantly.