Issue 25, 2014-07-21

Solving Advanced Encoding Problems with FFMPEG

Previous articles in the Code4Lib Journal touch on the capabilities of FFMPEG in great detail, and given these excellent introductions, the purpose of this article is to tackle some of the common problems users might face, dissecting more complicated commands and suggesting their possible uses.

by Josh Romphf

Introduction

FFMPEG has been an important encoding tool for technologists, preservationists, and hobbyists for well over a decade. It is a powerful, multi-purpose open-source library that operates from the command-line, and while a basic knowledge of the command-line environment is useful, FFMPEG can be effectively utilized with little to no programming experience. The source files are easily compiled on all major platforms, and static builds are available for those who prefer to install from an executable[1]. Two articles in Code4Lib Issue 23 touch on the capabilities of FFMPEG in great detail[2], and given these excellent introductions, the purpose of this article is to tackle some of the common problems users might face, dissecting more complicated commands and suggesting their possible uses. To refer to the FFMPEG community as an active one is a vast understatement, and it not surprising how quickly one can fall down an encoding rabbit hole when trying to search forums for solutions to their challenges. This article serves to provide some useful suggestions for transcoding techniques within a specific library or archival context.

Note: the following commands rely heavily on VideoLan's high performance x264 Library, which is licensed under the GNU GPL and can be compiled with FFMPEG, but is not added to the standard build. The x264 Library is used for encoding H.264/MPEG-4 AVC, and undergirds some of the most high profile streaming operations on the web, including YouTube, Vimeo, and Hulu. When compiled with FFMPEG, it is capable of high quality compression at relatively high speeds.

Understanding Your Project's Needs

Before embarking on an encoding project, consider whether or not FFMPEG is the right tool to be used. For instance, FFMPEG is excellent for batch processing large collections, but is not ideal for detail oriented digital restoration work. Similarly, it may prove to be overkill for simple transcoding projects, such as on-demand delivery of a single file for a patron. It may be more efficient to use a transcoding program such as Handbrake (which is built on FFMPEG and x264), which has a GUI and significantly less of a learning curve, instead of a command-line application that can give you highly customizable results.

One of the key steps in helping to make this call is defining the desired output. While this may sound obvious, it is important to tailor your tools to the task at hand: what is the method of delivery? Are you producing preservation quality files? Are you streaming? What type of player will be used? Does file size matter? If so, is there an optimal file size? For those who opt for FFMPEG, what follows are some scenarios accompanied by possible solutions.

Preservation and Access Basics: Input, Output, Preservation, and Streaming

Arguably the most common use of H.264/MPEG-4 AVC encoding is for streaming and access solutions, but it can also be used to transcode lossless preservation masters and high quality mezzanine files[3]. When the ultimate goal is preservation, the x264 codec supports lossless encoding, yet it should be noted that a number of streaming services, with the exception of YouTube, are capable of decoding lossless H.264. In addition, a new container format may be desired for posterity; one such format is Matroska (**MKV**), an open source container designed with the intention of becoming the defacto standard for multimedia containers. It has become known for its support of "all known video and audio compression formats" and has been adopted by a number of archival projects [5]. As libraries and archives begin to digitize their moving image collections and provide online access to them, the quality of streamed files becomes more and more important. Depending on whether or not the file size of the output is an issue, there are a few paths one can take with x264.

1) Transcoding for Quality:

```
ffmpeg -i input -c:v libx264 -preset slow -crf 18 -vf yadif -strict -2 output.mp4
```

When file size is not an issue, this is a quick command that will produce a visually lossless H.264 encoding with an mp4 container. **C:V** is the video codec of choice, **preset** is the compression preset (in this case **slow** for higher quality compression), and **CRF** is the **Constant Rate Factor,** which preserves an overall level of quality throughout the file by adjusting each frame's bitrate based on the given quality level. Consequently, the higher the CRF, the lower the overall quality level. The video filter flag (**-vf**) is used to call FFMPEG's pre-bundled video filters, while **yadif** (Yet Another Deinterlacing Filter) deinterlaces an interlaced input, as progressive video is not only easier to compress and most current computer monitors and televisions are progressive scan. Finally, when encoding with H.264/MPEG-4 AVC, the audio format used is AAC (Advanced Audio Coding); in order to enable FFMPEG's experimental, native AAC encoder, **-strict -2** needs to be added to the command. An external library such as libfaac [4] can also be used, and –strict -2 can be omitted.

A preservation quality master can be output using the same principles, setting the –crf to 0, the preset to veryslow, and the output container to .mkv:

```
ffmpeg -i input -c:v libx264 -preset veryslow -crf 0 -vf yadif -strict -2 output.mkv
```

A preservation quality master can be output using the same principles, setting the –crf to 0, the preset to **veryslow**, and the output container to .mkv:

```
ffmpeg -i input -c:v libx264 -preset veryslow -crf 0 -vf  yadif -strict -2 output.mkv
```

2) **Streaming for File Size:**

In a scenario where server space or bandwidth is limited, a desired file size may be required. To calculate the output file's bitrate based on a target size, the following equation can be used:

Bitrate = output file size in MB * 8192 (convert MB to KB) / Duration in seconds

From there, two-pass, variable bit rate (VBR) compression can be used to yield an approximate file size:

```
ffmpeg -i input -c:v libx264 -preset slow -b:v (calculated bitrate) -strict -2
-pass 1 -f mp4 /dev/null && ffmpeg -i input -c:v libx264 -preset slow -b:v (calculated bitrate) –strict -2 -pass 2 output.mp4
```

In the above example, the first command is passed to a null device, the **&&** operator then executes the second command, which yields the mp4 file.

3) **Streaming within a Browser**

If a streaming service such as YouTube or Vimeo is not being used, an option is to add **–movflags faststart** to the command, which shifts the *moov atom* (the data unit that defines the file's timescale, duration, and information for each track)[7] to the beginning of the mp4 container. This allows for playback and seeking to be started before the video has fully downloaded – a feature that is particularly useful when streaming content. Although some delivery mechanisms do not require the position of the moov atom to be located at the beginning of the container, it is required by both Flash video (**.flv**) and **html5 video**, which are two of the most common web-based video options available. Most other web-based players will be able to function regardless of the moov atom's position.

Concatenating Files

Another essential use of FFMPEG lies in its ability to concatenate (join) multiple video files. The need to concatenate may arise when content is ingested in the form of multiple files, when a patron requests a sequence of clips, or when an image sequence needs to be animated. The use of the **concat** filter is outlined in *Automated Processing of Massive Audio/Video Content Using FFmpeg*, and is suitable when concatenating at the file level, yet it can be lossy and uncooperative with some containers, especially mp4.

The most straightforward means of concatenating files produced with the same codec is by using the **concat demuxer**, which calls on an external .txt file for orders on which files to join and in what order. The text file should appear as follows:

```
#List of Files to Join (Comment)
file ‘/usr/path/to/file1’
file ‘/usr/path/to/file2’
```

The command itself is quite simple, and can be used as a stream copy for efficiency. **–c copy** moves all of the streams over to a new container without decoding and encoding the input:

```
ffmpeg -f concat -i /path/to/list/concatList.txt -c copy out.mp4
```

The Unix **cat** command can also be used for concatenating files, and is particularly helpful when pulling .VOB files together from a DVD. A Handbrake alternative, this method is ideal for transcoding to containers not offered by the program, such as .mov and .mp4:

```
cat /Volumes/YOURDVDTITLE/VIDEO_TS/VTS_01_[1234].VOB | ffmpeg -y -i - -c:v libx264 -preset slow -crf 18 -strict -2 output.mp4
```

The .VOB files are concatenated in order, where [1234] corresponds to the filename; for instance, to concatenate VTS_01_1.VOB and VTS_01_2.VOB, the command is cat /Volumes/YOURDVDTITLE/VIDEO_TS/VTS_01_[12]. This is then piped to FFMPEG, setting the .VOB files as the input. The user can then customize the remainder of the command in any way.

Another means of concatenating files with the same codec is the concat protocol in ffmpeg, which works at the file level as opposed to the stream level. Unlike the Unix cat command, the concat protocol uses pipes "|" to delineate input files:

```
ffmpeg –i “concat:input1|input2|input3” -c copy out.mp4
```

Be aware that some codec and container combinations may not work properly. For instance, H.264 encoded files cannot be losslessly concatenated in ffmpeg. There is, however, a workaround for this which is achieved by first copying them to mpeg2 transport streams (.ts). Using –c copy will yield an error that can be solved by the use of a bitstream filter (–bsf), which performs modifications at the bitstream level, foregoing any decoding and producing the same results. The transport streams can then be written to a temporary directory, concatenated, and removed:

```
ffmpeg -i input1.mp4 -bsf:v h264_mp4toannexb -f mpegts -c:v mpeg2video -b:v 2500k /path/to/temp/directory/intermediate1.ts &&

ffmpeg -i input2.mp4 -bsf:v h264_mp4toannexb -f mpegts -c:v mpeg2video -b:v 2500k /path/to/temp/directory/intermediate2.ts &&

ffmpeg -probesize 100M -analyzeduration 250M -i
"concat:/path/to/temp/directory/intermediate1.ts|/path/to/temp/directory/intermediate2.ts" -c:v libx264 -crf 18 -preset slow -
strict -2 -movflags faststart output.mp4 &&
rm -rf /path/to/temp/directory
```

The use of **–probesize** and **–analyzeduration** help FFMPEG to recognize the audio and video stream parameters of a file, and while they are by no means necessary, their use can help reduce input processing errors, especially when moving across codecs.

Finally, libx264's powerful animation capabilities can prove incredibly beneficial for preservation projects. For instance, film scans that produce individual frames (for instance, Cineon, JPEG, DPX or TIFF sequences) can be recombined and compressed for access. Another use could be for pulling JPEG2000 images from a DCP (Digital Cinema Package) and recombining them for viewing. To join image files, use the command:

```
ffmpeg -r 24/1 -i filename%04d.jpg -c:v libx264 -r 24 -pix_fmt yuv420p output.mp4
```

Where **–r 24/1** refers to the duration of each image, in this case 1/24th of a second, **filename%04d.jpg** finds all jpeg files with filename followed by a 4 number sequence (i.e. filename0001.jpg), and **–r 24** refers to the framerate. **–pix_fmt** is also important, as the pixel format must be set; in this case, the YUV color space with 4:2:0 chroma subsampling is used to maximize playback compatibility. FFMPEG also supports globbing for image sequences, where –i is supplemented with **–pattern_type glob –i**, whereby wildcards (i.e. *.jpg) can be used.

Scripting and Batch Processing

An even greater level of automation can be achieved by scripting FFMPEG and Unix commands, freeing up time for other tasks. All of the above commands can be scripted and combined using a number of scripting languages. For example, a script can be written to batch transcode a preservation MKV and streaming optimized MP4 from any type of source file. Two of the more popular methods are to use a conventional shell script or a Python script with subprocess calls. For instance, a simple shell script with a for loop can be written as such:

```
#!/bin/bash
#Convert all MOV files to MP4 in a directory
for x in /path/to/files/*.mov;
do ffmpeg -i "${x}" -c:v libx264 -crf 18 -preset slow –strict -2 –pix_fmt yuv420p "$x.mp4";
#Insert Other Commands, such as moving files or creating a lower quality copy
done
```

The file is then saved with the .sh extension and can be opened in a bash shell. This is by far the simplest method of the two, as it is language that FFMPEG users are already familiar with. Python can be used for more complex scripting, such as reading files through FFMPEG, processing them with Python, and piping them back out to FFMPEG for an output. With the inclusion of the OS and Subprocess modules, one can execute shell commands with all the functionality of a shell script. Subprocess calls can also be couched within a for loop or stored in variables for more efficient coding. A simple Python script with multiple commands would be written as follows:

```
#!/usr/bin/env python

import os
import subprocess

#Script to compile image sequence and add audio

#If ffmpeg is not in $PATH, use os.chdir('path/to/ffmpeg/binary/directory/')

#Compile image sequence

subprocess.call([
'ffmpeg',
'-r', '24/1',
'-i', 'filename%04d.jpg',
'-r', '24',
'-c:v', 'libx264',
'-preset', 'slow',
'-crf', '18',
'-pix_fmt', 'yuv420p'
'-s', '1920x1080',
' output.mp4'])

#Add audio track to output video

subprocess.call([
'ffmpeg',
'-i', 'output.mp4',
'-i', 'audioTrack.wav',
'-c:v', 'libx264',
'-c:a', 'aac',
'-strict', '-2'
'-b:a', '-192k',
'composite.mp4'])
```

The script is saved with the extension .py, and can be run in the same manner as any other python script:

```
python /path/to/script/videoscript.py
```

Conclusion

One can imagine the possibilities of FFMPEG, especially when combined with a powerful, yet accessible programming language like Python. This article merely scratches the surface of what can be done. An indispensable tool for libraries and archives – especially those with limited technical staffing – FFMPEG can be utilized to solve nearly all the needs of a digital video project. Again, the learning curve can be daunting, and the end result is largely what you make of it. It is hoped that this article served as a gentle introduction to both video encoding principals, as well as a companion for some of FFMPEG and x264's idiosyncrasies. Please consult the references section for more specific, in-depth commands and explanations.

Notes

[1] http://www.ffmpeg.org/download.html features downloads for nearly all operating systems, and https://trac.ffmpeg.org/wiki/CompilationGuide provides easy to follow compilation guides.

[2] Automated Processing of Massive Audio/Video Content Using FFMPEG (KIA Siang Hock and LI Lingxia, January 2014) contains several important, common commands and provides explanations, while Unix Commands and Batch Processing for the Reluctant Librarian or Archivist (Anthony Cocciolo, January 2014) explains how to batch transcode files using FFMPEG commands with a bash for loop.

[3] Mezzanine files are intended to be used as your input when transcoding access files to lessen the risk of damaging the master.

[4] Libfaac (Freeware Advanced Audio Encoder) –enable-libfaac –enable-nonfree

[5] The Matroska (http://matroska.org/technical/specs/index.html) container format has gained a great deal of momentum over the last few years, and has been adopted as the preservation container within the infrastructres of open source digital asset management systems such as Archivematica (https://www.archivematica.org/wiki/Main_Page) and Islandora (https://discoverygarden.ca/).

[6] Visit http://trac.ffmpeg.org/wiki/How%20to%20quickly%20compile%20libx264 for instructions on how to easily compile x264 with FFMPEG. As a side note for Mac users, the x264 Library (and a number of other dependencies) are included in the Homebrew installation.

[7] For an excellent article on the moov atom, see Understanding the MPEG-4 movie atom (Maxim Levkov, Adobe Developer Connection, October 2010) http://www.adobe.com/devnet/video/articles/mp4_movie_atom.html.

References:

Several FFMPEG tutorials are available through the official bug tracker and wiki: https://trac.ffmpeg.org/.

Another helpful resource is video engineer Fabio Sonnati's excellent FFmpeg – The Swiss Army Knife of Internet Streaming (http://sonnati.wordpress.com/2011/07/11/ffmpeg-the-swiss-army-knife-of-internet-streaming-part-i/).

VideoLan also maintains excellent documentation on the x264 library: http://www.videolan.org/developers/x264.html.

As always, Stack Overflow, albeit somewhat difficult to navigate, offers a great deal of tips and tricks http://stackoverflow.com/questions/tagged/ffmpeg. After Dawn http://www.afterdawn.com/ also maintains a forum for video encoding related questions and issues.

About the Author

Josh Romphf is a Programmer in the Digital Humanities Center at Rush Rhees Library (University of Rochester), specializing in web development, 3-D modeling, video encoding, and preservation solutions.