VIAA Preservation Reformatting

Statistics and notes on reformatting of jpeg2000/mxf files to other preservation and access formats

Authors: Jérôme Martinez, Dave Rice, Emanuel Lorrain, Matthias Priem

Introduction

VIAA is in the midst of a large preservation effort. A large number of these files is currently converted to JPEG2000 in MXF. While this effort is underway, VIAA has been examining the progress of FFV1 in Matroska as a developing preservation format. VIAA is also closely following the Preforma project where standardisation and conformance checking of FFV1 is being further developed. Presently JPEG2000 in MXF does present some accessibility challenges as many of the most accessible media tools do not support the method of interlaced field-encoding storage used by the MXF container. VIAA collaborated with MediaArea to extend MediaConch to facilitate reformatting JPEG2000 in MXF files to FFV1 in Matroska as well as several other files as access copies. The transcoding was performed by FFmpeg and MediaConch was extended in order to facilitate workflow management of reformatting, transcoding, and conformance checks.

The report summarizes the findings of this effort including lessons learned, transcoding statistics, and pending issues for exploration. For a initial workflow test, VIAA selected 70 hours of content, about 3 terabytes of MXF files, SD content (720x576@50i 10-bit) with JP2k compression + 4 AES audio (24-bit stereo) without compression, for processing. These files were all created by OpenCube MXFTk Advanced from videotape sources.

Integrity of the transcoding

Our first concern was whether the resulting FFv1 encoded files would indeed be lossless compared to the original JPEG2000 files, so they would decode to the same result as the original masters. To measure this, we tested the “framemd5” of all video frames, both after JPEG2000 decoding and FFV1 decoding. The process decodes the frames of both the source JPEG2000 file and the output FFV1 file, frame by frame, to produce a checksum of the decoded data. This method allows a lossless transcoding to be verified by demonstrating that two different data streams both decode to exactly the same data. Using a framemd5 comparison of the original and output can find input/output errors, transmission errors, or instances where the process is not lossless. All transcoding in this sample set was evaluated with framemd5 and no difference or irregularity was detected.

Speed

When transcoding large amounts of archive material, the speed of transcoding and creating derivatives is crucial for forecasting and resource planning. To get insight in the processing time of such a transcode, we tested the decoding performance of 3 decoders on a E5-2698V3 (16 cores+HT, 2.3-3.6 GHz) and compute of framemd5, configured with 32 parallel jobs:

- Using FFmpeg JPEG2000 decoder: 32 hours (~2x real time)
- Using FFmpeg+OpenJPEG decoder: 72 hours (~real time)
- Using FFmpeg FFV1 decoder: 10 hours (~7x real time)

When tested with a single thread (in order to compare decoders without benching their threading capability) the first 2.5 minutes of each file, we have the following results:

- From 0.4x to 2.4x (average 0.7x) real time/thread (encoding and decoding have same speed)
- From 0.7x to 16x (average 3x) the speed of JPEG2000 (FFmpeg decoding)

We tested the decoding performance of 3 decoders on a E5-2698V3 (16 cores+HT, 2.3-3.6 GHz) then compress to derivative files (MPEG-4 Visual, ProRes, WebM), configured with 32 parallel jobs:

- Using FFmpeg JP2k decoder: 38 hours (~2x real time)
- Using FFmpeg+OpenJPEG decoder: 78 hours (~real time)
- Using FFmpeg FFV1 decoder: 18 hours (~4x real time)

Comments:

- Decoding of FFV1 takes in 3x less time than decoding of JP2k by FFmpeg
- Using FFmpeg compiled with OpenJPEG library for JP2k decoding is not better (actually 2x worse)
- We did not test other JP2k decoders (e.g. HW accelerated ones)

Compression ratios

Next, we were interested in the compression ratio of the resulting files. We tested 3 different configurations of FFV1:

- version 1 (no slice, no slicecrc)
- version 3 with 4 slices and slicecrcs
- version 3 with 24 slices and slicecrcs

All of them used FLAC audio coding. Given that many audio tracks contained silence the use of FLAC could significantly reduce storage requirements for audio. Audio with sound recordings could be reduced in size by at least half and audio tracks of silence could be reduced much further. Additionally FLAC adds CRC values per audio frame so damage may be pinpointed far more easily than with PCM.

Through experimentation, the FFV1 encoding options eventually settled upon were:

- “-level 3”: to force the use of FFV1 version 3
- “-g 1”: to encode FFV1 as i-frame only which increases resiliency
- “-slices 24”: to encode each frame with 24 slices. A high slice count can increase data size but also increase resiliency.
- “-slicecrc 1”: to ensure that each slice contains a crc which aids in error concealment.

We compare the compression ratio of FFV1 in Matroska to JPEG2000 in MXF, we looked at three different settings over a large sample set of files. First we compared the two using the same level of features (i.e. no slices, no extra slicecrcs, or compression per field). Here the average file size difference is -9% (-2% due to JP2k to FFV1, -7% due to PCM to FLAC). Note that these percentages represent the reduction of the entire file and not the tracks specifically.

Using 4 slices per frame, the average file size difference is -7% (0% due to JP2k to FFV1, -7% due to PCM to FLAC).

Using the recommended encoding options listed above reduced the efficiency of the compression a bit but added several features focused on fixity and data resilience. With 24 slices per video frame and slicecrc, the average file size difference was -5% (+2% due to JP2k to FFV1 and -7% due to PCM to FLAC).

These numbers are expected. Generally lossless codecs working with an analog source can reduce the data rate to about one third the size of the original data source. Encoders may vary a bit with options that prioritize speed or size, but generally data sizes from one lossless encoding to another are not anticipated to be dramatically different.

Also note that these numbers reflect an average across a large file set. Depending on the content, compression ratio change per file results varied between +5% and -30% (when 24 slices per video frame is used). Although the difference in audio visual encoding is the most notable there is also a minor difference in the overhead of the container. Remuxing content from MXF to Matroska brought a -0.1% difference in file size as Matroska uses 10x less bytes than MXF. This is mostly because Matroska uses an efficient variable-size integer to represent structural names and sizes, whereas MXF uses fixed-size values throughout. Still the percentage of bytes used for the container compared to audiovisual encodings is quite small, so the impact in data size by the container change is small. On the other hand, Matroska brings some advantages by adding CRCs values to its top level elements so digital damage may be detected with greater precision than may be possible in MXF.

About audio compression: in the VIAA case, the source selection contains 3 different kinds of audio content: real content (first audio track), analog silence (second audio track) and digital silence (third and fourth audio tracks). FLAC compression has an average content size difference of -50% for tracks containing real content, -80% for analog silence, -99% for tracks with digital silence.

Some earlier tests gave the impression of a larger reduction in storage requirements with FFV1 as compared to JPEG2000, but these tests used FFmpeg's tinterlace filter to merge the field-based JPEG2000 encoding into a frame-based FFV1 encoding. When verifying the losslessness of this strategy we found that the tinterlace filter transforms a 10 bit input to an 8 bit output, and then a JPEG2000 to FFV1 transcoding through the tinterlace filter significantly reduces the size of the FFV1 encoding compared to the JPEG2000 encoding; however this was not a lossless transformation as the bit depth was reduced as well. To fix this issue we moved from the tinterlace filter to the similar weave filter which accomplishes the same field merging but can do so losslessly with 10 bit YCbCr video. Once the filter was replaced the compression rates of JPEG2000 used by OpenCube were more similar to FFV1 used by FFmpeg.

Findings

(this section was added May 15, 2017)
During our testing we discovered that our script produces FFV1 files with the same output framemd5 as the jpeg2000 file, but when a lossy output (such as mpeg4, webm, or prores) was created from the FFV1 file, it differed from the same lossy output generated from the original jpeg2000 file. This was not expected because we use the same configuration and intend for the jpeg2000 and ffv1 files to produce identical decoded data. Eventually we found that the handling of the jpeg2000 file differs from FFV1 because it requires its field-based encodings to be weaved together into frames; for this process we used the ffmpeg option -filter_complex with weave=first_field=bottom in the filterchain. Through testing we discovered that when -filter_complex is used, that ffmpeg presumes a different method for handling scaling. This was discussed on the ffmpeg-user listserv at http://ffmpeg.org/pipermail/ffmpeg-user/2017-May/036078.html. In order to ensure that the lossy outputs are consistent, we resolved the issue by adding specificity to our process by adding "scale=flags=bicubic" to all filterchains so that the scaling method we'd prefer is declared if it needs to be used.

Future work

- Working with current files poses some challenges: FFmpeg can not natively handle MXF files provided by VIAA because MXF has one field per packet so FFmpeg considers the 720x576@25fps stream as a 720x288@50fps stream, we had to add a field merge filter. FFmpeg may be updated as to support the current format natively.
- Our ordonnancer is not optimal, e.g. we use 1 thread per file (in order to avoid issues with decoders not supporting multithreading) and the script does not optimize the

order depending of the size of the file so the last file may be the longest one and a single core is used during some time at the end instead of using all cores.
- We may add a first pass for checking silent tracks, for lossy derivatives only or for lossy+lossless derivates.
- Standardisation of the FFV1 / MKV format, see ongoing work of the CELLAR workgroup. See https://datatracker.ietf.org/wg/cellar/documents/ for current status.

Source files

- VIAA derive script used for transcoding: https://github.com/viaacode/viaaarea
- MediaConch script used for benching: https://github.com/MediaArea/MediaConch-Bench

Numbers

VIAA_Compression_Performance.csv is in annex, columns:

- orig_file: file name of the source MXF/JP2k/PCM file
- orig_size: size in bytes of the source file
- orig_duration: duration in seconds of the source file
- created_file_size_4slices: size in byte of the transcoded MKV/JP2k/FLAC file when a specific profile with 4 CRC per FFV1 frame and FLAC audio coding is used
- created_file_size_24slices: size in byte of the transcoded MKV/JP2k/FLAC file when a specific profile with 24 CRC per FFV1 frame and FLAC audio coding is used.
- jp2ktolossy_150_transcode_time: duration of the transcoding from the first 2.5 minutes of JP2k source content to lossy derivatives
- jp2ktoffv1_150_transcode_time: duration of the transcoding from the first 2.5 minutes of JP2k source content to MKV/FFV1/FLAC
- ffv1tolossy_150_transcode_time: duration of the transcoding from the first 2.5 minutes of MKV/FFV1/FLAC transcoded content to lossy derivatives
- mediaconch_150_analysis_time: duration of the analysis from the first 2.5 minutes of MKV/FFV1/FLAC transcoded content to lossy derivatives