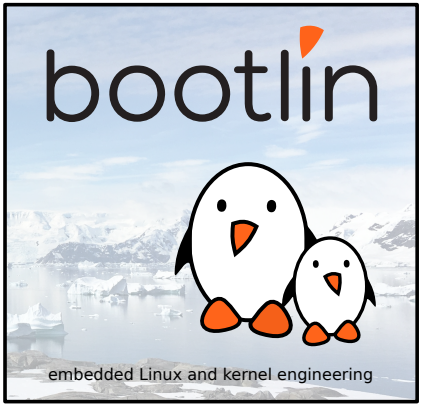
Supporting Hardware-Accelerated Video Encoding with Mainline

Paul Kocialkowski
paul@bootlin.com



Paul Kocialkowski

- Embedded Linux engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
- Open-source contributor
  - Co-maintainer of the **cedrus** VPU driver in V4L2
  - Contributor to the **sun4i-drm** DRM driver
  - Developed the **displaying and rendering graphics with Linux** training
  - Contributed **Allwinner MIPI CSI-2** support
- Living in **Toulouse**, south-west of France

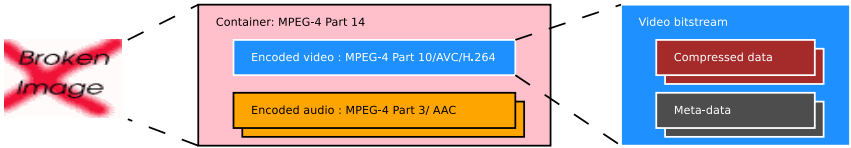
H.264 Encoding

Need for video encoding

- Representing pictures takes **significant memory**
- Example for a 10-minute **1920x1080** (32 bpp) video at 30 fps:
  - $1920 \times 1080 \times 4 = 7.91$ MiB/frame
  - $1920 \times 1080 \times 4 \times 30 = 237.3$ MiB/s
  - $1920 \times 1080 \times 4 \times 30 \times 10 \times 60 =$ **142.4 GiB**
- Significant sizes are an issue for **storage and network transmission**
- Video encoding aims at solving the issue:
  - Applying methods to **reduce the storage/transmission size**
  - Adding encoding and decoding **overhead/latency**
  - Keeping the perceived quality under control: **size/quality trade-off**
- Only the **currently-active frames** are kept in memory when decoding

Codec, Bitstream and Container

- Formats in which video is encoded are called **video codecs**
  - e.g. MJPEG, MPEG-2, MPEG-4 Visual (DivX), H.264/AVC, H.265/HEVC
  - Spanning over 7 generations with enriched features
- Video codecs are **format specifications** for both:
  - **Compressed video data**, that can be decoded into frames
  - **Meta-data** that configures the decoder (to match encoder settings)
- The video **bitstream** packs the data continuously (often with formatting)
- A **container** packs the video bitstream with other sources (audio, subtitles)



H.264 Introduction

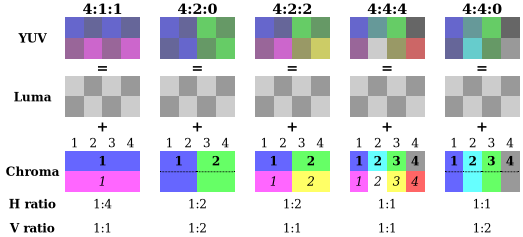
- ITU-T **H.264**, aka ISO **MPEG-4 AVC**, aka ISO MPEG-4 Part 10
- Probably one of the most popular and used codecs nowadays
- Supports both **progressive** and **interlaced** (used for TV broadcast)
- Specific **profiles** support a sub-set of compression features, such as:
  - **Baseline**: Simple profile with few features (low resources)
  - **High**: More features and flexibility
- **Levels** limit the maximum bitrates and dimensions
- Designed for efficient **hardware implementations**
  - Usually limited to specific profiles/levels
- Extended with **annex specifications**:
  - **H.264 SVC**: temporal/spatial/quality scalability
  - **H.264 MVC**: multi-view (stereoscopy)

H.264 Semantics

- H.264 specifies the **semantics and syntax** to store compressed video
- Information is split into **Network Abstraction Layer Units** (NALUs)
- Each NALU has a common header and a specific type:
  - **Sequence Parameter Set** (SPS): meta-data for the sequence
  - **Picture Parameter Set** (PPS): meta-data for the picture
  - **Coded slice data**: slice header and data
  - More for extra information and specific slice coding
- Meta-data is **bit-aligned** and often **conditional**
- NALUs are prefixed with a byte-aligned start code: 00000001 in **Annex-B format**
- Pictures are divided into blocks of 16x16 pixels called **macroblocks**
- Sets of macroblocks are grouped as **slices**
- Slices have a **specific type**, depending on the prediction mode:
  I slices for intra prediction and **P**/**B** slices for inter

H.264 Compression Techniques: Color Sub-sampling

- **Chroma sub-sampling** is used to reduce the bpp
  - The Human visual system is more sensitive to luminance than chrominance
  - Color-model and color-space conversion, e.g. sRGB to YUV Rec. 709
  - Spatial sub-sampling is applied to chrominance
  - YUV 4:2:0 gives 12 bpp, reduces size by 2 without significant quality loss



H.264 Compression Techniques: Quantization

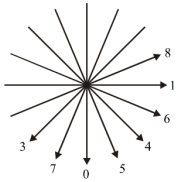
- Macroblocks are converted from spatial to **frequency domain**, using a **discrete cosine transform (DCT)** operation
- A **quantization step** (Q_{step}) parameter divides coefficients before rounding,

$$X_q = round\left(\frac{X}{Q_{step}}\right)$$

- A **quantization parameter** ($QP \in [\![0; 51]\!]$) indexes the quantization step
- **Details in the picture** are lost as QP and Q_{step} increase
- Quantized coefficients are laid out in **zig-zag order** to group zeros, easily compressed with entropy coding

H.264 Compression Techniques: Spatial

- Pictures that can be decoded alone are **intra-coded (I slices)**
- **Redundancy** often exists within a picture
- Pixels can be **deduced from neighbors** with prediction patterns
- H.264 supports many **intra prediction modes** (with specified directions)



Intra prediction directions

H.264 Compression Techniques: Temporal

- In most videos, **subsequent pictures are mostly the same**
- **Temporal differences** can be represented instead of each full picture
- **Motion vectors** between pictures are estimated at encoding
- They are applied to **reference pictures** for inter-picture prediction
- H.264 supports up to **16 references**
- References need to be kept decoded and alive in memory

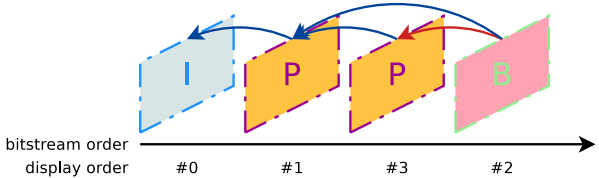
H.264 Compression Techniques: Temporal

Motion vectors visualized using:

```
ffplay -flags2 +export_mvs input.mp4 -vf codecview=mv=pf+bf+bb
```

H.264 Compression Techniques: Temporal

- Types of inter prediction in H.264:
  - **Backwards prediction** (P slices): using previous pictures
  - **Bidirectional prediction** (B slices): using previous and following pictures
- An **intra-coded picture** is necessary for inter prediction
- Following pictures for B slices need to **come first in bitstream order**
- A **group of pictures** (GOP) is sequence starting with an intra picture



- Bidirectional inter prediction introduces **latency** when encoding and decoding

H.264 Compression Techniques: Entropy

- A final **entropic compression pass** is applied to produce the bitstream
- Entropy coding assigns **shorter symbols to frequent occurrences**
- **Lossless compression** method that yields good results for video
- Syntax elements (meta-data) numbers are coded as **Exponential Golomb**
- Quantized DCT coefficients are coded using either:
  - **CAVLC**: Context-Adaptive Variable Length Coding (default)
  - **CABAC**: Context-Adaptive Binary Arithmetic Coding (advanced)

H.264 Encoding Rate Control

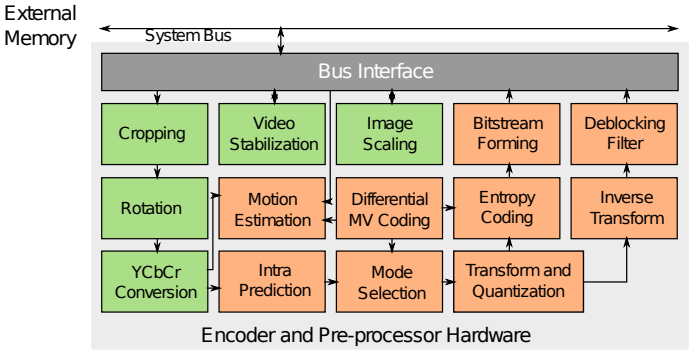
- Encoders apply a trade-off between **quality and bitstream size** the process is called **rate control** in general
- Quality is controlled by the **quantization parameter** QP
- **Rate control** modes:
  - **CQP**: constant $QP \in [\![0; 51]\!]$ parameter for all frames
  - **CRF**: constant rate factor (quality) $CRF \in [\![0; 51]\!]$
  - **CBR**: constant bitrate (kb/s)
  - **ABR**: average bitrate (kb/s) for the whole sequence, works best with two-passs encoding
- The most appropriate mode depends on the **use-case**
- Quality can be evaluated using a **PSNR** metric

Hantro H1 H.264 Encoder

Hantro H1 Outline

- The **Hantro H1** is a common hardware H.264/VP8/JPEG encoder
  - Initially developed by Hantro Oy
  - Acquired by On2 Technologies in 2007
  - Acquired by Google in 2010
  - Distributed as WebM Video Encoder Hardware IP
  - Distributed with H.264 by VeriSilicon since 2015
- Found in some **embedded ARM SoCs**:
  - **Rockchip**: RK3288, RK3328, RK3399, PX30, RK1808
  - **NXP**: i.MX8MM
- Supports encoding H.264 in **1080p at 30 or 60 fps**
- Supports **Baseline, Main and High** H.264 Profiles, also MVC Stereo High

Hantro H1 Block Diagram



Hantro H1 Block Diagram from the i.MX8MM Manual

Hantro H1 Operation

- **Stateless implementation** (no micro-controller/firmware)
- **Pre-processor** with cropping, rotation, scaling, CSC and stabilization
- Produces **slice NALUs** to memory, as Annex-B or direct NALU
- **Meta-data** (PPS, SPS) is generated in software, with **parameter constraints**:
  - SPS `pic_order_cnt_type = 2`
  - SPS `log2_max_frame_num_minus4 = 12`
  - PPS `weighted_bipred_idc = 0`
- Only supports **I and P slices** (no B slices), for embedded recording
- **References** (for P slices) are stored in dedicated **reconstruction buffers**
- **CABAC tables** (for High-Profile) are also stored in a dedicated buffer

Hantro H1 Internal Rate Control Mechanisms

- Base **Quantization** is specified with QP, QP_{min}, QP_{max}
- Advanced **internal mechanisms** exist for Rate Control in Hantro H1
  - Allow **QP adjustments** during the encoding process
  - **No longer used** by reference software nowadays
- **MAD** (mean absolute difference) mechanism:
  - Threshold value ($MAD_{threshold}$) for QP increase/decrease (Δ_{QP})
  - Single threshold and delta for a picture
- **Checkpoints** mechanism:
  - Checkpoints at regular macroblock distance, with up to 10 checkpoints
  - Targets for cumulative non-zero quantization coefficients
  - Error between target and actual count is evaluated at checkpoints
  - A Δ_{QP} is applied depending on the error
- **Feedback data** (from registers) is used for **control loop regulation**

Hantro H1 Feedback Data

- **QP sum**: sum of the QP value for each macroblock:

$$QP_{sum} = \sum_{macroblocks} QP_{macroblock}$$

- **RLC count**: number of non-zero quantization coefficients in the picture
- **Checkpoint values**: number of non-zero quantization coefficients at specified macroblock intervals
- **MAD count**: number of macroblocks under a specified **mean absolute difference** threshold

V4L2 Integration for Stateless Encoding

V4L2 stateful encoding support

- V4L2 already supports **stateful** H.264 encoders:
  - Using the `V4L2_PIX_FMT_H264` pixel format for **H.264 bitstream**
  - Producing both **slice and meta-data** NALUs
  - Using the **V4L2 M2M** framework
  - Drivers: `coda`, `mtk-vcodec`, `venus`, `s5p-mfc`, `hva`
- Various **generic V4L2 controls** allow configuring the encode run:
  - `V4L2_CID_MPEG_VIDEO_H264_PROFILE`, `V4L2_CID_MPEG_VIDEO_H264_LEVEL`
  - `V4L2_CID_MPEG_VIDEO_H264_8X8_TRANSFORM`, `V4L2_CID_MPEG_VIDEO_H264_ENTROPY_MODE`
  - `V4L2_CID_MPEG_VIDEO_BITRATE_MODE`, `V4L2_CID_MPEG_VIDEO_BITRATE`
- Some drivers have **specific V4L2 controls** too:
  - `V4L2_CID_MPEG_MFC51_VIDEO_FORCE_FRAME_TYPE`
- **Rate-control** is implemented by the encoder firmware
- **State and reference** management is also done by the firmware

V4L2 Stateless Encoding Considerations

- With **stateless encoding** on the Hantro H1, many parameters can be set:
  - Most of the PPS/SPS/slice header parameters
  - Some are restricted to specific values
- **State** is tracked by V4L2 and userspace:
  - Buffers and parameters are tied (using the **Media Request API**)
  - Reconstruction buffers need to be kept around
  - They are provided as references when needed
- **Rate control** is left to the V4L2 driver and/or userspace:
  - Feedback data needs to be provided

V4L2 Stateless Encoding: Existing Hantro H1 Support

- Chromium OS supports the **Hantro H1 on Rockchip**
- **Chromium OS kernel** implementation (downstream based on Linux 4.4):
  `https://chromium.googlesource.com/chromiumos/third_party/kernel/+/chromeos-4.4/drivers/media/platform/rockchip-vpu`
- **Chromium OS userspace** `libv4l2plugins` implementation :
  `https://chromium.googlesource.com/chromiumos/third_party/libv4lplugins/`
- **Rockchip's MPP** supports the Hantro H1 (VEPU1/VEPU2):
  `https://github.com/rockchip-linux/mpp`
  `http://opensource.rock-chips.com/wiki_Mpp`

V4L2 Stateless Encoding: First Approach

- The mainline `hantro` driver (in staging) supports **Hantro G1 decoding**
- Bootlin added support for **H.264 Hantro H1 encoding** to the driver
- Inspired by Chromium OS and MPP implementations
- Using the **Media Request API** and V4L2 controls
- Rate control (CBR) is done **fully in userspace** based on feedback
- **Kernel side** (based on Linux 5.4 with backported media patches): `https://github.com/bootlin/linux/tree/hantro/h264-encoding`
- **Userspace side**: `https://github.com/bootlin/v4l2-hantro-h264-encoder`

V4L2 Stateless Encoding: First Approach API

```
struct v4l2_ctrl_h264_encode_params {
  /* Slice parameters */
  __u8 slice_type;
  __u8 pic_parameter_set_id;
  __u16 frame_num;
  __u16 idr_pic_id;
  __u8 cabac_init_idc;
  __u8 disable_deblocking_filter_idc;
  __s8 slice_alpha_c0_offset_div2;
  __s8 slice_beta_offset_div2;
  __s32 slice_size_mb_rows;
  /* PPS parameters */
  __s8 pic_init_qp_minus26;
  __s8 chroma_qp_index_offset;
  __u32 flags; /* V4L2_H264_ENCODE_FLAG_ */
  /* Reference */
  __u64 reference_ts;
};
```

```
struct v4l2_ctrl_h264_encode_rc {
  __u32 qp;
  __u32 qp_min;
  __u32 qp_max;
  __s32 mad_qp_delta;
  __u32 mad_threshold;

  __u32 cp_distance_mbs;
  __u32 cp_target[10];
  __s32 cp_target_error[6];
  __s32 cp_qp_delta[7];
};

struct v4l2_ctrl_h264_encode_feedback {
  __u32 qp_sum;
  __u32 cp[10];
  __u32 mad_count;
  __u32 rlc_count;
};
```

V4L2 Stateless Encoding Approaches: Proposals

- Major downside is the **lack of genericity** for a single API
- Using **generic V4L2 controls** for encode parameters could work:
  - Existing stateful controls (profile/level/features)
  - Additional controls to indicate **references**
- Generic rate control done in **userspace**:
  - Requires generic controls (QP, slice type/GOP can be enough)
  - Requires generic feedback data (RLC and QP sum can be)
  - Cannot support hardware-specific mechanisms
  - Encourages proprietary implementations
- Rate control done in **kernel drivers**:
  - Easier for userspace but no fine control
  - Can reuse existing stateful V4L2 RC controls

V4L2 Stateless Encoding Approaches: Plan

- Bootlin has interest in starting a discussion
- Design decisions are needed to upstream Hantro H1 support
- Also concerns other stateless encoders (e.g. on Allwinner)
  - Little information is available currently
- Feel free to let us know about:
  - Interest in the topic
  - Details of stateless hardware that could be affected
  - Relevant use-cases to support for hardware encoding

Questions? Suggestions? Comments?

Paul Kocialkowski
paul@bootlin.com

https://bootlin.com/pub/conferences/