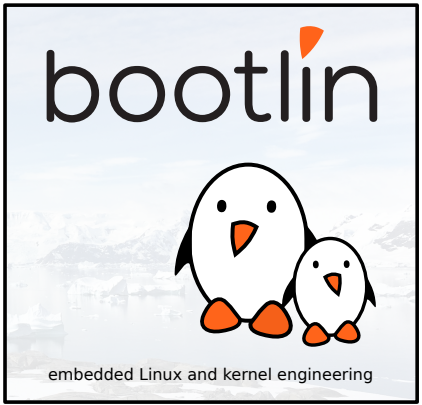
Integrating HW-Accelerated Video Decoding with the Display Stack

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
- Living in **Toulouse**, south-west of France

Outline and Introduction

Purpose of this talk

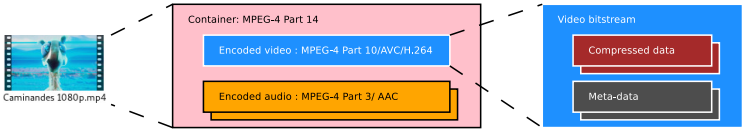
- Present our **specific use case**
  - Some basics about video decoding
  - How Linux supports dedicated hardware for it
  - Our hardware, driver and constraints
- Provide an overview of **video pipeline integration**
  - From source to sink
  - With efficient use of the hardware
  - Using the existing userspace software components
- Detail what went **wrong**
  - Things don't always pan out in the graphics world
  - Sharing the pain points we encountered
  - Constructive criticism, things could be a lot worse
    Always look on the bright side of life

Purpose of this talk

Let's try and build a good pipeline, eh?

You said video decoding?

- Sequences of pictures take a huge load of data to represent...
- So we compress them using a given codec:
  - Color compression: YUV sub-sampling
  - Spatial compression: frequency-space transform (DCT) and filtering
  - Temporal compression: multi-directional interpolation
  - Entropy compression: Huffman coding, Arithmetic coding
- Add some meta-data to the mix to get the bitstream
- Encapsulate that bitstream with other things (audio, ...) in a container
- Then we have a reasonable amount of data for a fair result!



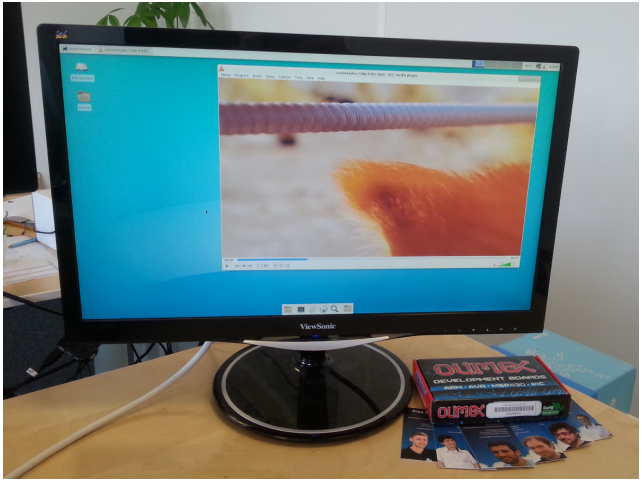
You said hardware video decoding?

- So now we need a significant number of operations to get back our frames
- Embedded systems don't have that much CPU time to spare
- Hardware to the rescue: fixed-function decoder block implementations
  - Digest video bitstream to spit out decoded pictures
  - Implementations are per-codec (or per-generation)
- Two distinct types of hardware implementations:
  - **Stateful**: with a MCU to parse raw meta-data from bitstream, keep track of buffers
  - **Stateless**: that expect parsed metadata and compressed data only

Hardware video decoding in Linux (Media/V4L2)

- In Linux, hardware video decoders (aka VPUs) are supported in V4L2
- Support for stateful VPUs landed with the **V4L2 M2M** framework
  - Adapted to memory-to-memory hardware
  - Source (output) is bitstream, destination (capture) is a decoded picture
- Support for stateless VPUs landed with the **Media Request API**
  - Meta-data is passed in per-codec V4L2 controls
  - Controls are synchronized with buffers under media requests
  - Source (output) is compressed data, destination (capture) is a decoded picture
- Decoded pictures are accessed:
  - By the CPU through `mmap` on the destination buffer
  - By other devices through `dma-buf` import of the destination buffer

The kind of expected result



H.265 hardware video decoding with UI integration

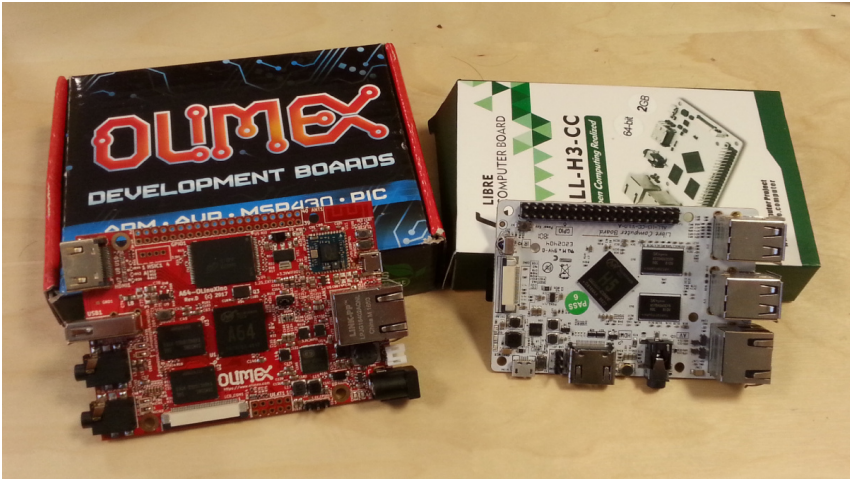
What to do with decoded pictures

Video decoding is just the tip of the iceberg...

- Colorspace conversion (CSC) from YUV is often needed
- Scaling and composition with UI are also required
- These are awfully calculation-intensive
  sometimes more than CPU-based video decoding
- But hey, we have hardware for that too:
  - The display engine usually supports all these operations via overlays/planes
  - Sometimes there are dedicated hardware blocks too
  - The GPU can do anything, so it can do that too (right?)
- Let's avoid copies and share buffers between devices
  full-frame memory copies are just a big no-no for performance

Hardware video decoding on Allwinner platforms and display stack integration

Allwinner platforms

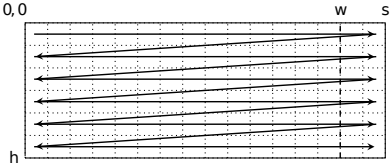


Community Allwinner boards from our friends at Olimex and Libre Computer

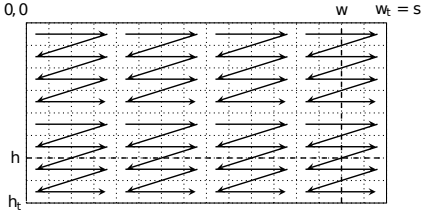
Our situation: the Allwinner side of things

- Relevant **multimedia blocks** on Allwinner hardware:
  - **Video decoder (VPU)**: fixed-function (stateless) implementation, supports **MPEG-2/H.263/Xvid/H.264/VP6/VP8**, **H.265**/VP9 on recent SoCs
  - **Display engines**: support multiple input overlays
  - **GPU**: Mali 400/450 in most cases
- First generation of devices (A10-A33) **comes with constraints**:
  - VPU can only map the lowest 256 MiB of RAM
  - VPU produces pictures in a specific tiled scan order (aka MB32)
  - Display engine supports MB32 tiling for planes/overlay
- Second generation (A33-A64+) **doesn't have these constraints**:
  - VPU still works with tiling internally, but untiling block is in the VPU

Allwinner MB32 tiled video format



Linear (raster) scan order



MB32-tiled scan order

- w: width, s: stride
- h: height

- w_t: tile-aligned width (stride)
- h_t: tile-aligned height

Bootlin's contribution for hardware video decoding support

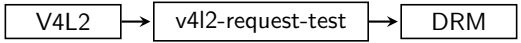
- On the **DRM kernel** side:
  - `DRM_FORMAT_MOD_ALLWINNER_TILED` **modifier** (merged in 5.1)
  - sun4i-drm support for linear/tiled YUV formats in **overlay planes** (merged in 5.1)
- On the **V4L2 kernel** side:
  - Cedrus base driver (merged in 5.1)
  - `V4L2_PIX_FMT_SUNXI_TILED_NV12` pixel format (merged in 5.1)
  - Experimental stateless **MPEG-2** API and cedrus support (merged in 5.1)
  - Experimental stateless **H.264** API and cedrus support (merged in 5.3)
  - Experimental stateless **H.265** API and cedrus support (to be merged in 5.5)
- On the **userspace** side:
  - A test utility: **v4l2-request-test**
    `https://github.com/bootlin/v4l2-request-test`
  - A VAAPI back-end: **libva-v4l2-request**
    `https://github.com/bootlin/libva-v4l2-request`

Investigated and/or implemented setups

Bare-metal pipeline setup

- Test scenario: **standalone dedicated application** (`v4l2-request-test`)
- Talks to the kernel directly (both V4L2 and DRM)
- Uses dma-buf for zero-copy
  - Exported from V4L2 with the `VIDIOC_EXPBUF` ioctl
  - Imported to DRM with the `DRM_IOCTL_PRIME_FD_TO_HANDLE` ioctl
- CSC, scaling and composition offloaded using DRM planes
- Bottomline: **all is well** but very limited use-case (testing)

Pipeline components overview:



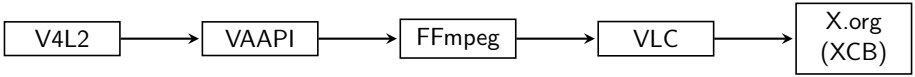
X.org pipeline setup (GPU-less): investigation

- Scenario: usual media players (using VAAPI) under X
- Can we use a similar setup (dma-buf to DRM plane) under X?
  - X initially only knows about RGB formats
  - But extensions exist: Xv, DRI3
- Xv extension allows supporting YUV and scaling, but...
  - Requires writing a hardware-specific DDX (e.g. to use planes)
  - Requires a buffer copy and doesn't support modifiers
  - Has synchronization issues and deprecated anyway (in favor of GL)
- DRI3 supposedly can solve these points:
  - Supports dma-buf import (but no modifier support)
  - Currently apparently only implemented in glamor (GPU-backed)
  - Doesn't give us access to a DRM planes

X.org pipeline setup (GPU-less): bottomline

- Scenario: usual media players (using VAAPI) under X
- What worked:
  - Software untiling (NEON-accelerated) in VAAPI back-end
  - Software-based CSC, scaling and composition
  - Buffer copies through XCB
- As a result, performance sucks
  still surprisingly good without scaling involved

Pipeline components overview:



Improving the X.org pipeline with a GPU in the mix

- Using the GPU shall speed things up
  - Requires using the `xf86-video-armsoc` DDX
  - Only accelerates rendering, not composition using GL (glamor)
- First try: importing YUV with the GPU and untiling
  - Lack of/undocumented blob support for YUV format
  - Zero-copy (dma-buf) import supported by the blob only for RGB formats
- Second try: importing as 8-bit component (luminance) and untiling
  - Wrote an untiling shader that just works on Intel GPUs
  - Zero-copy (dma-buf) not supported for (`GL_LUMINANCE`)
  - Copy import (`glTexImage2D`) for `GL_LUMINANCE` failed
    apparently a weird undocumented issue due to Mali constraints
  - Untiling shader never worked with the Mali (tl;dr)
- Bottomline:
  - GPU didn't help, for reasons we can't fix
  - Perhaps a free driver (Lima) would help?

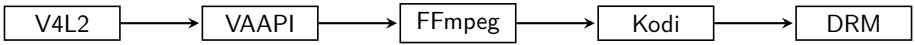
But what about Wayland?

- Didn't investigate/implement at the time of the project
- Wayland's relationship with DRM planes:
  - Planes are not exposed to applications
  - But might be used by the compositor internally
- Zero-copy buffer import from devices:
  - Exposed with the `linux-dmabuf` extension, `zwp_linux_dmabuf_v1` interface
  - Modifiers are supported by the protocol
  - `libweston` implementation calls `EGL_EXT_image_dma_buf_import_modifiers`
  - Requires GPU hardware support for the modifier
- Bottomline: unusable for our (GPU-less) use case

Kodi pipeline

- Kodi (media center) relies on GPU support, compatible with Mali blob
- Kodi supports the GBM EGL back-end
  - Allows using GL with DRM as output surface
  - Used for drawing the UI
  - Video CSC/scaling/composition uses a plane directly
  - Supports dma-buf import from FFmpeg
- Required plumbing to get it to work:
  - FFmpeg hwaccel support to use our V4L2-exposed codec (through VAAPI)
- Bottomline: it works great!

Pipeline components overview:



General takeaway

- Planes support is never exposed to applications
  at best supported and hidden by the compositor
- Modifier support is still very rare in userspace
- Strong incentive all around the userspace stack to use GL
  the unified way to integrate graphics
- But GPU support does not always solve the issue:
  - Life's much harder when it's a proprietary blob
  - Lack of usable dma-buf import support
  - Bugs and limitations
- Some projects try to make use of planes easier:
  - **libliftoff**, **liboutput**
  - Microchip's **Ensemble Graphics Toolkit**: `https://ensemble.graphics/`

Questions? Suggestions? Comments?

Paul Kocialkowski
paul@bootlin.com

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