Chrome OS zero-copy video capture (go/cros-vide0Capture)

SHARED EXTERNALLY

Status: Work in progress
Authors: tfiga@google.com (et al.)
Last Updated: 2018-May-8

Objective

The purpose of this project is to eliminate CPU processing (such as memory copy, software format conversion, etc.) of video buffers on various video capture paths used on Chrome OS.

We want to

- Remove memory copies between video buffers,
- Handle pixel formats native for given platform, without intermediate format conversions,
- Reduce CPU load, power consumption and heat dissipation of use cases involving video capture. (See **Improvement estimation** section.)

We might

- Completely replace shmem-backed video buffers with DMA-buf backed ones on Chrome OS. (Even for use cases not involving video capture, e.g. casting.)

We do not want to

- Replace format hard code to I420 with another hard coded format.

For simplicity, only MIPI cameras, which capture into native YUV format are considered in this document.

Background

Chrome OS video capture stack dates back to the times when high resolution (720p+) video capture was not very common. Current design, when coupled with modern hardware and neighboring software stacks (Android HALv3-based camera stack, graphics stack), requires memory copies and software format conversions to work. Moreover, shared memory is used to back video buffers, rather than, more efficient for hardware processing, DMA-buf memory. Even though computing power of new hardware is enough to handle those shortcomings, end users pay for this with reduced battery life and higher heat dissipation of their devices.

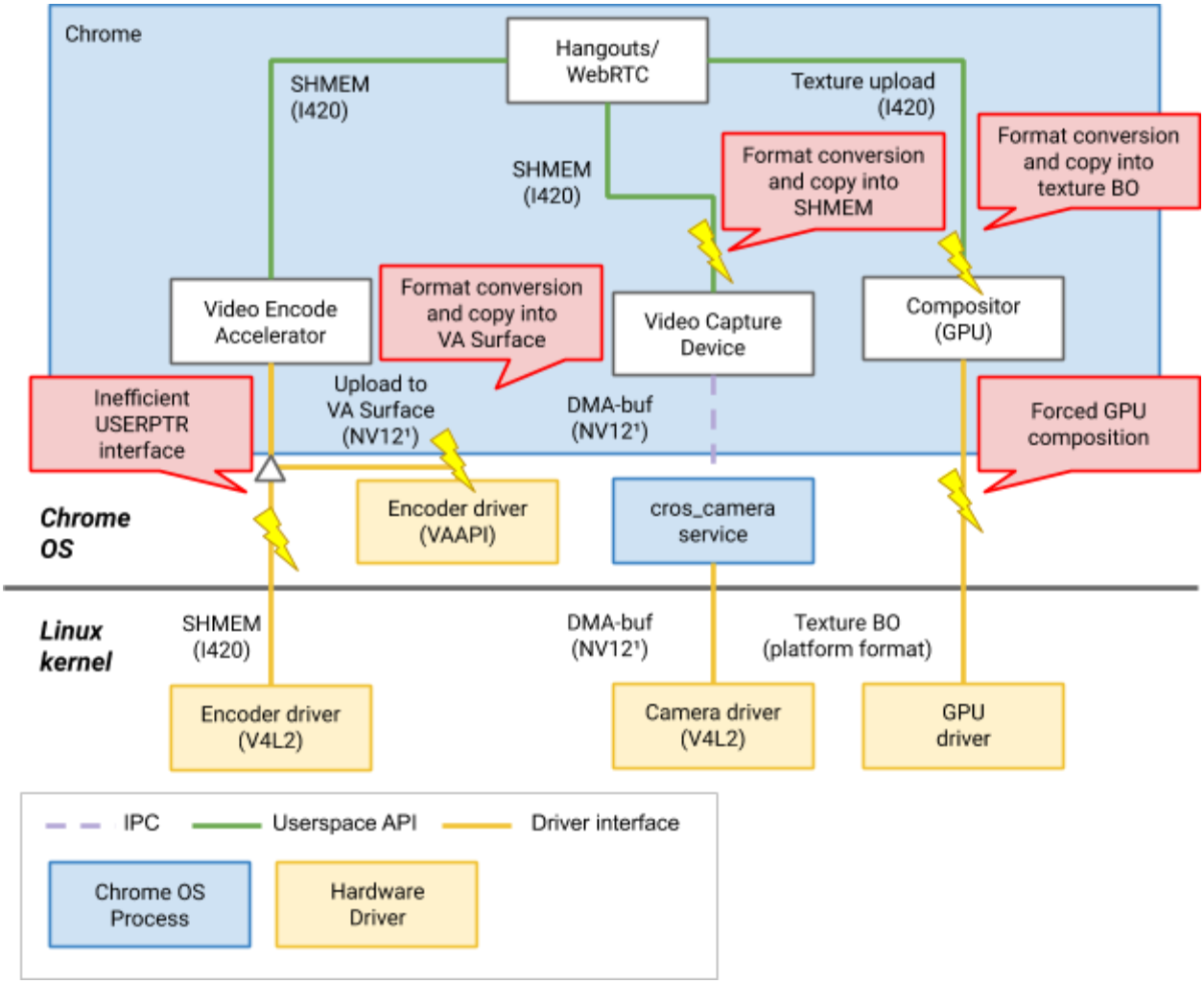
Recently, there is a big movement around Chrome OS team to improve system efficiency and thus reduce heat dissipation and extend battery times. Video capture is a highly exercised use case thanks to Hangouts video calls, which is a good motivation to work on improving the state

of things. Similarly, due to Chrome OS tablets with MIPI cameras surfacing out, video recording will start matter as well.

There are multiple components, which might be involved in video capture process on Chrome OS, depending on the use case. Main pipelines are depicted below.

Video capture in Chrome (Hangouts)

The video capture pipeline during a Hangouts call in Chrome is shown in the diagram below.



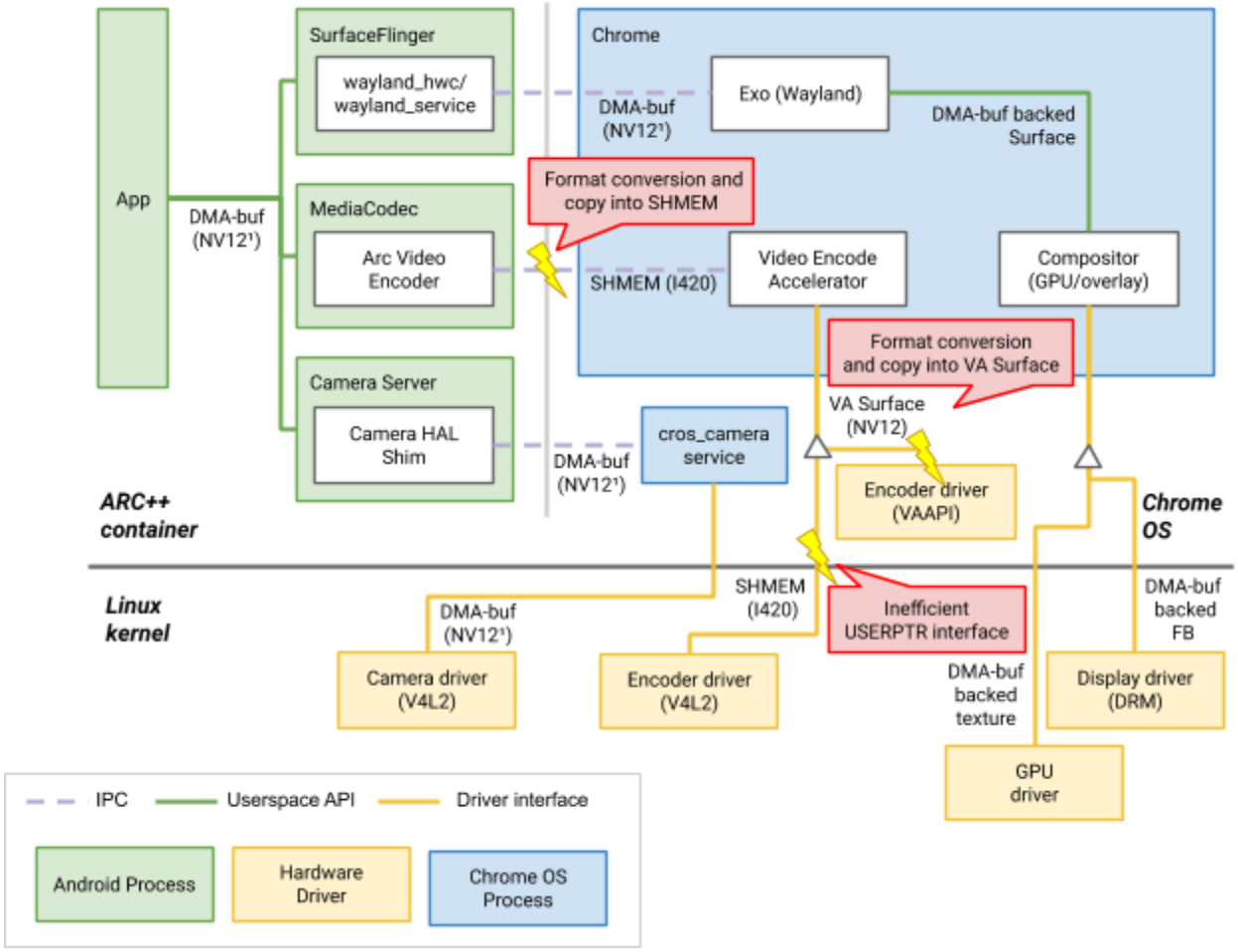
POSIX shared memory (SHMEM) is currently the dominant (and only supported for many interfaces) video frame storage in Chromium, whereas both video and graphics hardware subsystems prefer to deal with DMA-buf memory. Moreover, the only supported pixel format for video capture in Chromium is I420 (3-plane YUV420), which is not necessarily the format supported natively by video capture hardware, video codec, GPU or display controller. Wherever the storage or format is problematic to support in hardware, a memory copy or out-of-place format conversion is required.

On top of that, shared memory is not suitable for backing textures or hardware overlays, which makes the display path fall back to sub-optimal texture upload and GPU composition.

Another aspect of the problem is related to the way the kernel deals with requests to use user space memory (such as shared memory) with hardware (USERPTR mechanism). Mapping and cache maintenance of USERPTR buffers need to be done just before the operation starts and after it ends, further cache maintenance and unmapping needs to happen. The operations carry a processing cost and executing them every frame introduces a significant overhead.

Video capture in Android

The following diagram shows the video capture pipeline used for Android applications running inside ARC++ container.



On Android-side, where DMA-buf is used for all hardware memory, handling of video buffers is already done efficiently. The problem starts on the boundary of the container, where Arc Video Encoder delegates video encoding to Chrome Video Encode Accelerator instance. The latter currently expects SHMEM buffers with I420 pixel format, neither of which is native to Android.

On top of that, the same problems of limiting encoded format support to I420 and forcing the use of inefficient USERPTR, as mentioned for the Chrome pipeline, exist in the ARC++ pipeline as well.

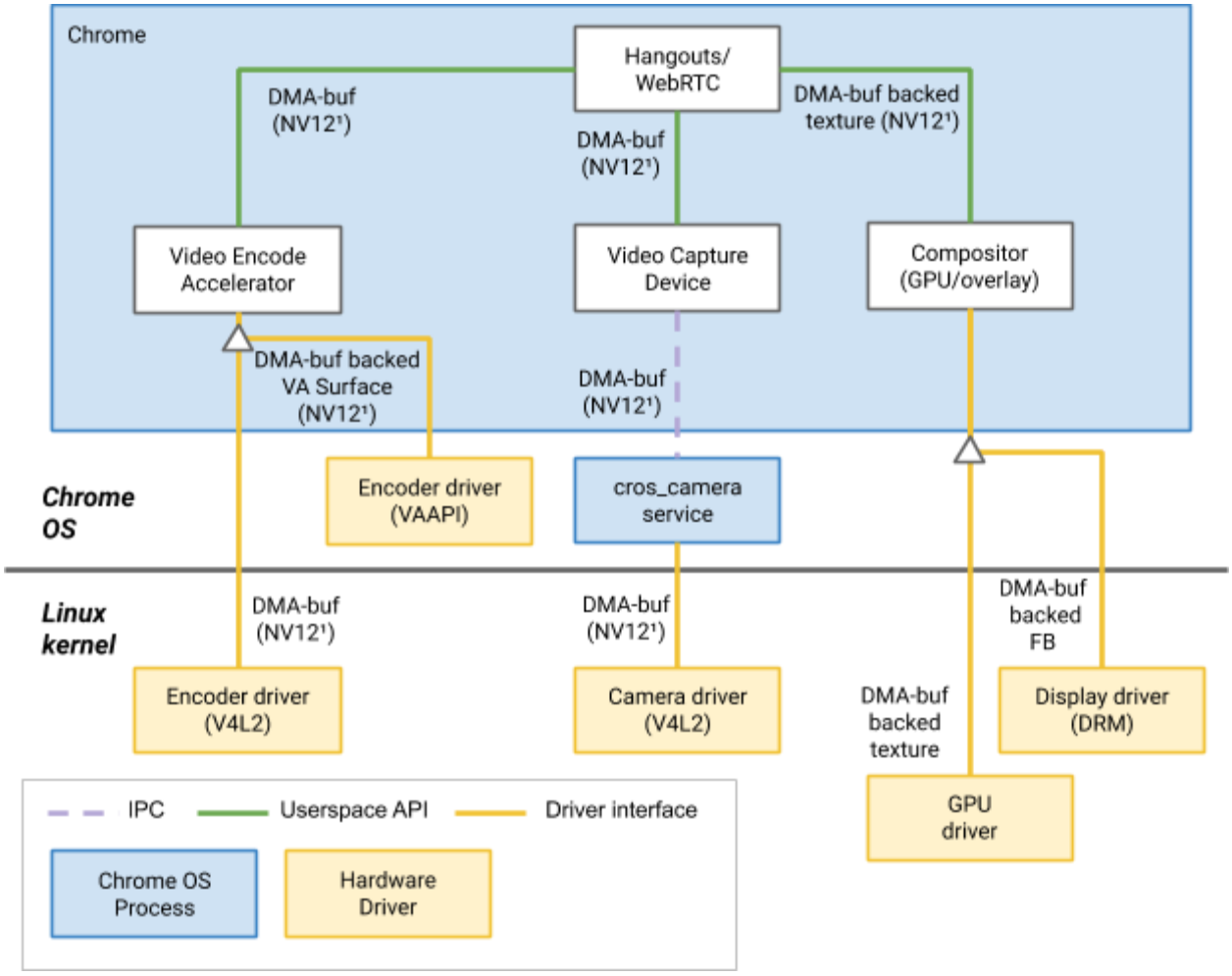
Further reading

- Design Doc: Virtual Video Capture Devices Accepting Textures (public)
- Design Doc: Video Conversion and Distribution (Vicodis) Service (internal)
- Video Capture using GpuMemoryBuffers (shared externally)

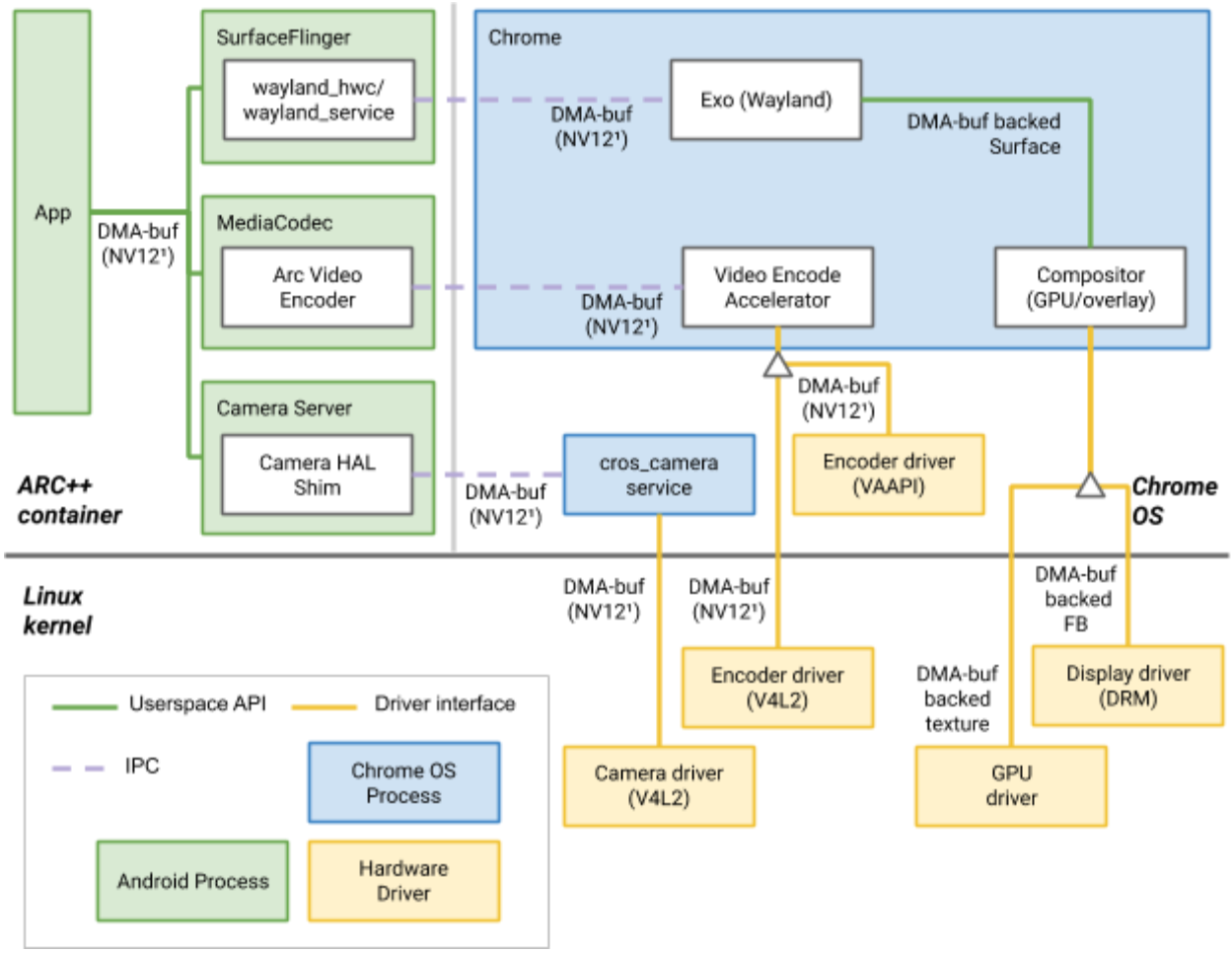
Overview

As depicted by the two diagrams below, with all the required changes, the single camera buffer containing given frame is used directly by all the consumers interested in it (display, encoder, web application). Moreover, for Hangouts in browser case, using the buffer directly makes it possible to display the video preview using hardware overlay, without involving GPU compositor.

Video capture in Chrome (Hangouts)



Video capture in Android



Detailed Design

Native formats in Video Encode Accelerator

V4L2 Video Encode Accelerator [Done]

Should be fine as is, as it doesn't seem to include any assumptions about the format being I420.

[1] The NV12 format is mentioned for simplicity, as it matches most of our hardware. On some platforms, the format might vary between different parts of the pipeline, but typically it would not involve a software format conversion.

VAAPI Video Encode Accelerator [Initial work done for NV12]

Assumes the format is PIXEL_FORMAT_I420 and fails if it isn't [cs]. Includes calculations that assume the format. VaapiWrapper [cs] assumes that Chrome uses I420 and hardware surfaces are backed with NV12 buffers.

ArcVideoEncoder

The arc.mojom.video_encode_accelerator interface [cs] mimics media::VideoEncodeAccelerator and already implements format selection at initialization time.

The complicating factor here is Android side, which typically uses a flexible YUV or implementation-defined format to back native windows and the real format is known only after receiving the first buffer and doing a format resolve.

DMA-buf in Video Encode Accelerator

V4L2 Video Encode Accelerator [Done]

There is already some support for DMA-buf input to encoder for the case when image processor is used for format conversion. It should be possible to separate respective logic and reuse for the case without image processor as well.

There are also some places where USERPTR input is assumed. They will need to be reworked.

A complicating factor is the way current VideoEncodeAccelerator interface specifies buffer type. It doesn't do so at Initialize(), but rather by embedding StorageType into VideoFrame objects directly. We might want to rework it so that it behaves similar to VideoDecodeAccelerator, which specifies the input buffer storage inside Config structure passed to Initialize(). (Some preliminary refactor started in this CL.]

VAAPI Video Encode Accelerator [Done]

Current code always uploads (does an out-of-place format conversion) incoming frames from mapped source buffer to mapped VASurface. It should be possible to add DMA-buf support in a way similar to VAAPI Video Decode Accelerator [cs].

ArcVideoEncoder

The arc.mojom.video_encode_accelerator interface [cs] seems to already include StorageType in Initialize(). The implementation on both Android and Chromium sides currently assumes SHMEM, though.

Native formats and DMA-buf in Chrome Camera Stack

- On the Chrome OS HAL v3 VCD we capture the buffer in GpuMemoryBuffer from the HAL already. The buffer is copied/converted to SharedMemory in VCD currently to pass to Chrome.
- For camera app, we'll need to check if the MediaRecorder API can be made zero-copy. Presumably it should work if the encoder stack for WebRTC works.
- The captured frames may display in a <video> tag on HTML. We should also make it zero-copy if it's not already.
- In case of USB cameras on x86 platforms, VAAPIJpegDecodeAccelerator is used to decode MJPEG frames and it always copies decoded frames from VAImage into VideoFrame (currently shared memory).

Native formats and DMA-buf in WebRTC

WebRTC stack only tries to access the underlying buffer if SW encoders are to be used. Otherwise, we can expect WebRTC to return the buffer we passed from capture back to us for HW encode. As long as we pass the native buffers in Chrome's media::VideoFrame container, we can expect them to be intact.

- In capturer source, we wrap a media::VideoFrame in a webrtc::VideoFrameBuffer and pass into WebRTC video track.
- WebRTC video track sends this frame to 2 sinks in a regular scenario: HW encoder and renderer. Both[1][2] access the underlying media::VideoFrame as it is.
- If WebRTC tries to access the underlying buffer for some reason, it calls webrtc::VideoFrameBuffer::ToI420(). There we can actually mmap() the native buffer.
- DMA-buf should be fine, as long as we don't hit the software fallback on ARM devices (~~Possibly x86 with Kepler too? AI(tfiga): Check how V4L2 memory is mapped on x86.~~)
- **Potential blocker:** Simulcast - requires hardware encoder stack to be able to handle downscaling (ImageProcessor should be helpful?~~ AI(tfiga): Check if our hardware encoders can also scale on the fly.~~).

Improvement estimation

Experiment to estimate CPU load (internal).

tl;dr:	Babymega	58% -> 45%
	Eve	18% -> 15%
	Dru	69% -> 44%

Work plan

See Action Items for Chrome OS Zero-Copy Video Capture (internal).

Alternatives Considered

- Use I420 for the whole pipeline

Does not work, because I420 is not the format natively supported by both camera and codec hardware on all our devices. Moreover, display controllers on our existing devices don't support this format for overlays.

- Use shared memory for the whole pipeline

Not impossible technically, but infeasible. The whole Android side of the capture stack and also Chrome OS graphics stack heavily rely on DMA-buf. Moreover, for security guarantees, user space pointers need more careful handling when mapping into device address space and so performance is affected.

Caveats

- Currently on ARM, whenever Chromium needs to access a DMA-buf using CPU, we end up mapping them uncached, which slows the access down (significantly for complex access patterns). We need to examine what kind of CPU access is still needed, if we do zero-copy and consider mapping DMA-bufs with cache if really needed. (or some shadow buffers, for example)