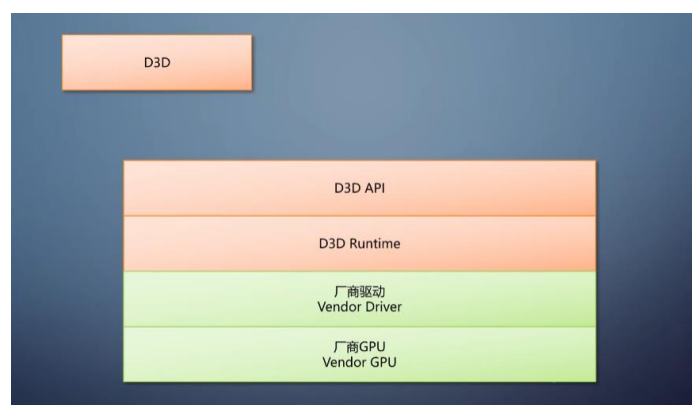


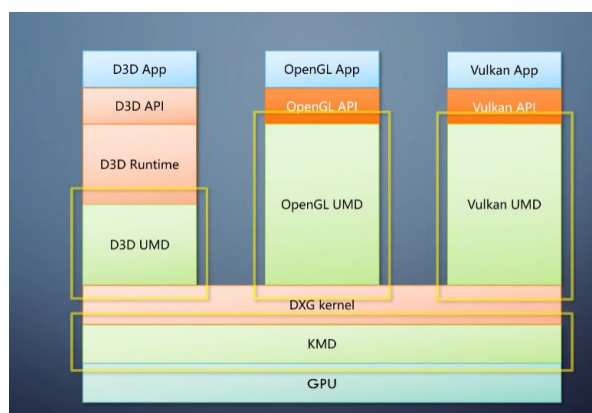
在查询 webcodecs 的 issue 时，发现一个问题 “Converting VideoFrame to canvas images or WebGL textures is very slow”，其中一个回复是 “I suspect that the VideoFrame you got from decoder is backed by CPU resources(Which means, the GPU decoder not works, because of profiles). One simple way to check this with chrome is that you could use HTMLVideoElement to play your video and use chrome://media-internals to see the decoder it uses.”

kTotalBytes	"0x2c5f5b"
kVideoDecoderName	"D3D11VideoDecoder"
kVideoTracks	[{"alpha mode":"is_opaque","codec":"avi","coded size":"1920x1080","color space":" primaries:BT709, transfer:BT709, matrix:BT709, range:LIMITED","encryption scheme":"Unencrypted","has extra data":true,"hdr metadata":"unset","natural size":"1920x1080","orientation":"0","profile":"avi profile main","visible rect":"0,0 1920x1080"}]
origin_url	"http://localhost:3000/"

在 vscode 中，通过<video>元素播放准备的视频，发现 kVideoDecoderName 的值是 D3D11VideoDecoder。经过查询资料，了解到 D3D11 是图形学 API 接口，该接口由微软实现，往下翻译上层应用的请求以形成对硬件的操作。D3D11 不跨平台，但是跨厂商，即通过该 API 在 windows 上可以实现调用不同厂商的 GPU 进行工作。D3D11 的模式是微软拥有接口和上层实现，硬件厂商拥有底层实现。



从分层架构可以了解到，GPU 执行的是驱动发来的操作，并不知道这个操作来自于哪一个 API。GPU 支持什么 API，实际上是指 GPU 厂商提供了哪个 API 的驱动来访问 GPU 的功能。

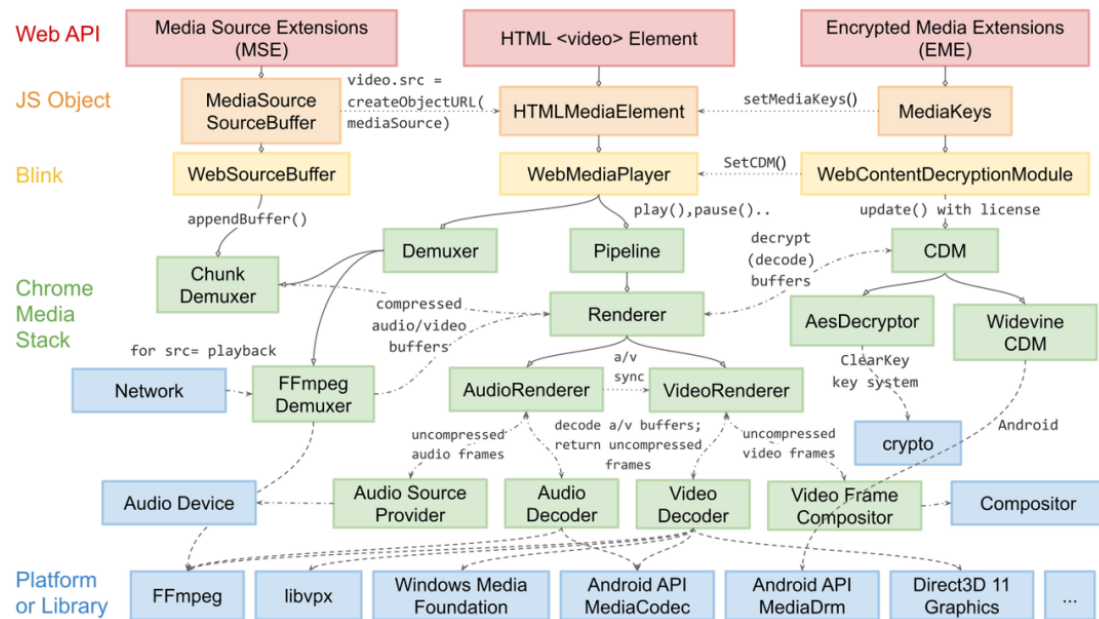


硬解框架五花八门，不同的显卡厂商和设备有各自的专用解码框架，操作系统也有定义好的通用解码框架，由于显卡厂商众多，因此大部分播放器一般均基于通用框架实现硬解，

少部分播放器在人力充裕的情况可能会为了更好的性能额外对专用框架二次实现。

其中 Windows 平台通用的解码框架有 Media Foundation, Direct3D 11, DXVA2, 以及 OpenCL。macOS 平台通用的解码框架只有一个, 也就是苹果自己的 VideoToolbox。Linux 平台的通用解码框架有 VA-API 和 OpenCL。

至于在浏览器上的解码过程，根据 Chromium Media 的模块介绍，浏览器将音视频播放一共抽象成三种类型，比较常见的有：Video Element 标签，MSE API。此外还有支持加密视频播放的 EME API，这三种在底层又存在多种复用关系。



Chromium 的解码流程

在解码一个视频时，底层的解码模块的逻辑大概为：

- 1) 浏览器会从列表中依次按照顺序查找 Decoder, 通常来说优先级最高的是硬解 Decoder, 然后会尝试软解 Decoder。
- 2) 如有命中其中的某个 Decoder 则执行后续解码逻辑。
- 3) 如没有命中的 Decoder, 则解码失败, 中止。

因此，Windows 平台上，在 chrome 中播放 av1 视频时，chrome 会调用通用的解码框架 D3D11VideoDecoder（在大于 Windows8 且支持 D3D11 的系统默认被使用）以及 VDAAudioDecoder（前者不被支持时使用）。

之前的实验 demo 在 linux（虚拟机/双系统）上进行，运行截图如下图，即产出的 frame 格式为 I420。

I420
4843
I420
4846.700000000186
I420
4846.799999999814
I420
4851.5999999996275
I420
4851.700000000186
I420
4854.4000000003725
I420
4856.5
I420
4857.9000000003725

如图，webcodecs 在虚拟机上运行时解析出来的每一帧 frame 都是 I420 格式。通过查阅 chrome://gpu，发现虚拟机上的 video decode 与 video encode 加速均不支持。

- Accelerated video encode has been disabled, either via blocklist, about:flags or the command line.
Disabled Features: *video_encode*
- Accelerated video decode has been disabled, either via blocklist, about:flags or the command line.
Disabled Features: *video_decode*

在 windows 平台上，能够发现解析出来的帧是 NV12 格式，经过查询，了解到 NV12 是 NVIDIA GPU 解码的特定输出，这也就代表了使用 GPU 硬解码。通过查询 chrome://gpu，查询“av1”关键字，也进一步确认了 chrome 在 windows 平台上对 av1 视频有硬件解码的支持。

Video Acceleration Information

Decoding	
Decode h264 baseline	64x64 to 4096x4096 pixels
Decode h264 main	64x64 to 4096x4096 pixels
Decode h264 high	64x64 to 4096x4096 pixels
Decode vp9 profile0	64x64 to 8192x8192 pixels
Decode vp9 profile2	64x64 to 8192x8192 pixels
Decode av1 profile main	64x64 to 8192x8192 pixels
Encoding	
Encode h264 baseline	0x0 to 1920x1088 pixels, and/or 30.000 fps
Encode h264 main	0x0 to 1920x1088 pixels, and/or 30.000 fps
Encode h264 high	0x0 to 1920x1088 pixels, and/or 30.000 fps

之前一直在思考，为什么要把 videoframe 存储下来，为什么要尽量的转码快而且占空间少。一直在想，单纯的做一个视频播放应该用不到这些，但是突然意识到了 webcodecs 和 webgpu 的用途(也是在一个会议记录上受到的启发)，他可能是要实时加载视频并且转出去。因为 webcodecs 允许对视频的每一帧进行处理，在这个具体的处理过程中，应用将每一帧进行背景虚化处理，(然后重新生成 frame?)

在实际的处理过程中，流程如图：

首先，也是我目前比较关注的部分，导入 videoframe 的过程：

VideoFraome - (createImageBitmap) -> ImageBitmap - (copyExternalImageToTexture) -> GPUTexture

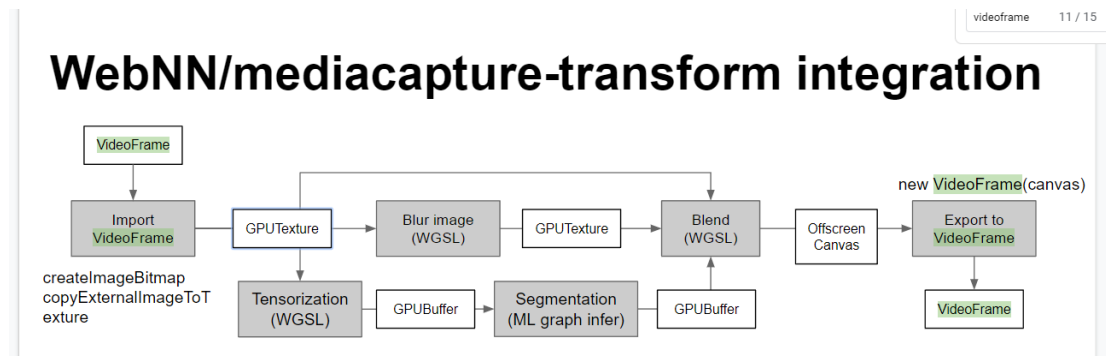
Video Processing in Workers (cont'd)

videoframe 11 / 15

- Details of the WebGPU/WebNN processing pipeline (webgpu-background-blur.js):
 1. **VideoFrame** import: VideoFrame - (createImageBitmap) -> ImageBitmap - (GPUQueue::copyExternalImageToTexture) -> GPUTexture
 2. Image blur: the shader implementation is based on @austinEng's WebGPU samples project (thanks!).
 3. Input tensor preprocessing: it is implemented in a WebGPU compute shader. Its input is a GPUTexture and the output is a GPUBuffer. The GPUBuffer will feed to WebNN graph compute as the input.
 4. Background segmentation: it is implemented by a WebNN graph (webnn_deeplabv3.js). The weights come from the TFLite DeepLabV3 model. This TFLite model and TF.js DeepLabV3 model (used by WebGL pipeline) are based on the same TF model (tensorflow/deeplabv3/1).
 5. Image blend: WebNN graph puts the segmentation results (segmap) into the output GPUBuffer. Another WebGPU compute shader is used to blend the original input and the blurred one based on the segmap. The final output is drawn into an offscreen canvas.
 6. **VideoFrame** export: create **VideoFrame** from the offscreen canvas

14

整体的流程如图：



在经过一系列复杂的操作后，将结果重新绘制在 canvas 上，然后用这个 canvas 导出成 videoframe。

这个项目目前的考虑优化点：

Opens:

- Reduce the CPU usage, current sample spends 35% on createImageBitmap and 20% on GC
 - More efficient video import?
 - More static pipeline?
- Flow control?
 - The WebGL-based processing pipeline would hang UI on entry level GPU

- 1) 高居不下的 CPU，35%的 createImageBitMap 和 20%的垃圾销毁
- 2) 更加方便的视频导入方式？
- 3) 更加静态的 pipeline
- 4) 流控制？基于 webgl 的 pipeline 处理会将 UIhang 在入门级 GPU

看到了一个问题，关于如何通过 copyTo 函数缓存解码后的 videoframe？回答中提及 createImageBitmap()不会回读，但是它产生的空间会更大，目前我还没有想通这具体是指什

么。 <https://github.com/w3c/webcodecs/issues/500>

如果使用 copyTo 的话，看看这个文章 <https://github.com/w3c/webcodecs/issues/501>

oneVPL

为什么要使用硬件 codec?


许广新：“其中一个好处是，低延时和高吞吐。硬件 codec 还有一个好处是，CPU 的使用率会降低，会大量节省电源的消耗。但是它也有不好的地方。一个是，它很难跟软件 encoder 的最高质量模式竞争。因为客户对一般硬件编码器不是很熟悉，有很多厂商从下到上都是闭环的，很难去做定制化。但是，在 Intel 的平台上，我们从上到下，从 driver 到中间件、FFmpeg 都是开源的，能在一定程度上帮助大家。大家需要对 Intel 硬件有一定的了解，才能去做定制化。所以，硬件 codec 大概的用途有 video play，现在市面上看到的播放器都会用硬件来播放，很少用软件，否则 CPU 就会全部占用了。还有流媒体、云游戏的用途，以及对时延要求非常高的 video encoder。”


下面来自 ffmpeg 的官方网站（<https://trac.ffmpeg.org/wiki/HWAccelIntro>）

Ffmpeg 本身是一个多媒体框架，能够实现解码，编码，复用，解复用，转码的工具。Ffmpeg 具备高度可移植性。Ffmpeg 包含了可以供程序使用的 libavcodec，libvutil 和 libavformat、libavfilter、libavdevice、libswscale 和 libswresample 等多种库与 ffmpeg、ffplay 和 ffprobe 这些可以供用户进行转码和播放的小工具。

许多硬件解码器的一个共同特点是能够在适合其他组件使用的硬件表面中生成输出（对于独立显卡，这意味着表面在卡上的内存中，而不是在系统内存中）——这通常对播放很有用，因为在渲染输出之前不需要进一步复制，并且在某些情况下，它还可以与支持硬件表面输入的编码器一起使用，以避免在转码情况下进行任何复制。

硬件编码器生成的输出质量通常比 x264 等优质软件编码器低得多，但通常速度更快且不使用太多 CPU 资源。（也就是说，它们需要更高的比特率才能以相同的感知质量进行输出，或者它们以相同的比特率以较低的感知质量进行输出。）

 2021 上海

 Intel Video Stack
— 智能媒体技术 —


Intel Linux Media API Introduction

- VAAPI is a low-level open-source API that allows applications to use hardware video acceleration capabilities, usually provided by the GPU. It is implemented by the libva, combined with a hardware-specific driver, usually provided together with the GPU driver. VAAPI is available everywhere in Linux. It is also used by Android and Chrome
- The oneAPI Video Processing Library (oneVPL) is a high-level API for video decoding, encoding, and processing to build portable media pipelines on CPUs, GPUs, and other accelerators. It provides device discovery and selection in media centric and video analytics workloads and API primitives for zero-copy buffer sharing. oneVPL has great encoder features/quality.

oneVPL/MediaSDK

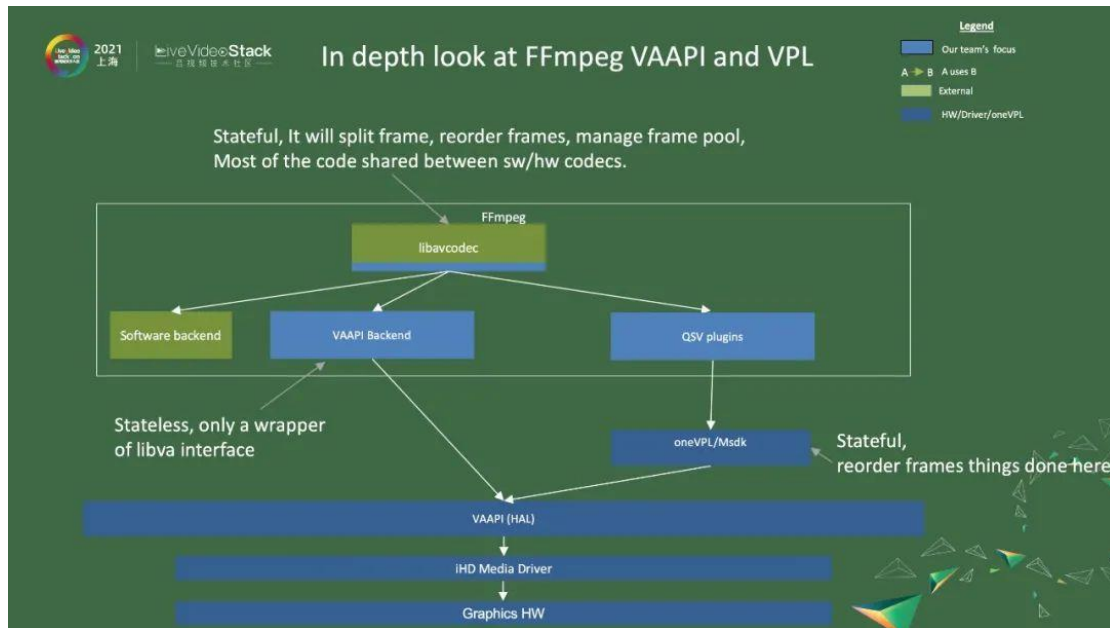
VAAPI(HAL)

Video Driver



图中是对 Intel 在 Linux 上的媒体 API 的简单介绍，在介绍后面内容之前需要对此有个基本的了解。最下面是软件层，是 video driver。在此之上，我们提供了一个叫 VAAPI 的接口，这就是 Intel 在 Linux 上面的硬件抽象层。基本上 Linux 系统都会有这个硬件抽象层，比如 Android、Chrome 上都有。但是，这个硬件抽象层，一般人会觉得非常复杂，因为需要做 surface 管理，最后要 decode 到 slice 层，才能使用到这个 API。所以，Intel 又提供了另外一套 API，就是 oneVPL，你可以认为它就是以前的 MediaSDK，它是一个相对高级的 api，能够通过 CPU，GPU 以及其他的加速器实现解码，编码等操作，它提供了设备发现和设备选择功能（it provides device discovery and selection in media centric and video analytics workloads and api primitives for zero-copy buffer sharing.）。这一层主要的接口就是，一帧的码流进去，然后一帧解码后的数据出来，因为它是基于 VAAPI 的二次开发，所以它能够提供更强大的编码参数，并做了一些跟硬件相关的质量调整，能够提供更高质量的编码码流。

Intel 媒体处理框架的整体架构：



简单介绍 FFmpeg VAAPI 和 VPL, 图中是架构图。最底下是 Graphic HW, 上面是 driver, 以及硬件抽象层。在硬件抽象层的基础上, FFmpeg 会提供 VAAPI 的 backend, 直接调用 VAAPI。然后, 我们也会做一些硬件相关优化, 就提供了一个 oneVPL 或 media SDK, 然后会被 QSV plugin 调用。它们都会被 libavcodec 接口调用, 最后 FFmpeg 提供统一的接口给外部使用。

Platform API Availability

	Linux			Windows			Android	Apple		Other
	AMD	Intel	NVIDIA	AMD	Intel	NVIDIA		macOS	iOS	Raspberry Pi
AMF	N	N	N	Y	N	N	N	N	N	N
NVENC/NVDEC/CUVID	N	N	Y	N	N	Y	N	N	N	N
Direct3D 11	N	N	N	Y	Y	Y	N	N	N	N
Direct3D 9 (DXVA2)	N	N	N	Y	Y	Y	N	N	N	N
libmfx	N	Y	N	N	Y	N	N	N	N	N
MediaCodec	N	N	N	N	N	N	Y	N	N	N
Media Foundation	N	N	N	Y	Y	Y	N	N	N	N
MMAL	N	N	N	N	N	N	N	N	N	Y
OpenCL	Y	Y	Y	Y	Y	Y	P	Y	N	N
OpenMAX	P	N	N	N	N	N	P	N	N	Y
V4L2 M2M	N	N	N	N	N	N	P	N	N	N
VAAPI	P	Y	P	N	N	N	N	N	N	N
VDPAU	P	N	Y	N	N	N	N	N	N	N
VideoToolbox	N	N	N	N	N	N	N	Y	Y	N

Key:

- Y Fully usable.
- P Partial support (some devices / some features).
- N Not possible.

图为各个系统平台上的相关 api

透过上图, 可以发现 libmfx 支持 linux 和 window 上的 intel 设备, 有点类似于 nVidia 的 nvenc 一套工具。Vaapi 则只支持 linux 上的 intel 设备。

Intel Quick Sync Video(QSV)是 Intel GPU 上跟视频处理有关的一系列硬件特性的称呼。

在不同平台上可通过不同 API 使用 Intel GPU 的硬件加速能力。目前主要由两套 API: VAAPI 以及 libmfx。

VAAPI (视频加速 API, Video Acceleration API)包含一套开源的库(LibVA) 以及 API 规范, 用于硬件加速下的视频编解码以及处理, 只有 Linux 上的驱动提供支持。

Libmfx 是 Intel Media SDK 中的 API 规范, 支持视频编解码以及媒体处理。支持在 Windows 以及 Linux 平台上实现硬件处理。

除了 Intel 自己的 API, 在 Windows 系统上还有其他 API 可使用 Intel GPU 的硬件加速能力, 这些 API 属于 Windows 标准, 由 Intel 显卡驱动实现。

DXVA2 / D3D11VA, 他们是标准 Windows API, 支持通过 Intel 显卡驱动进行视频编解码, FFmpeg 有对应实现。

Media Foundation, 也是标准 Windows API, 支持通过 Intel 显卡驱动进行视频编解码, FFmpeg 不支持该 API。

Intel qsv:

Qsv 是 intel GPU 芯片中一组硬件功能的名称。

API 支持:

可以通过许多不同的 API 设备访问硬件:

Libvpl: 这是一个来自英特尔的库, 可以作为英特尔 oneAPI 视频处理库 (OneVPL) 的一部分进行安装。 OneVPL 是 MSDK 的继承者, 将支持更多的编解码案例。

DXVA2 / D3D11VA: 这些是标准的 Windows API, 由英特尔图形驱动程序实现以支持视频解码。

libmfx on Linux / Windows: 这是一个来自英特尔的库, 可以作为 intel media SDK 的一部分安装, 并支持编码和解码案例的子集。