Quiz: Video Processing on Web-Enabled XPU Clients

1. Webcodecs

1.1 熟悉项目并完成 samples

在 linux 系统下，使用 pycharm 中搭建项目。项目结构如下图 1.1。static 目录中存放了 js 文件和相关的 media 媒体文件。Templates 目录下分为 demo1 至 demo5，demo1 至 demo3 为 samples1 至 samples3。

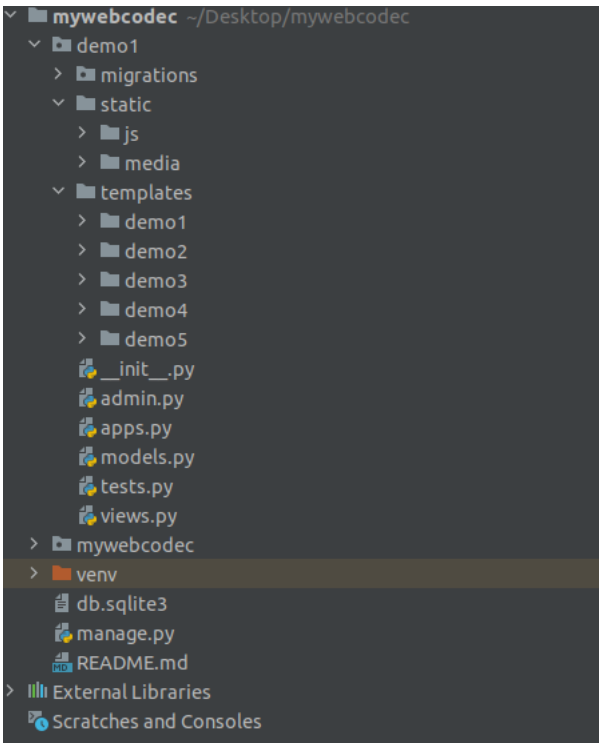


图 1.1 samples 项目结构

（1） demo1

根据资料 Webcodecs 框架下的解码过程如下图 1.2 所示。

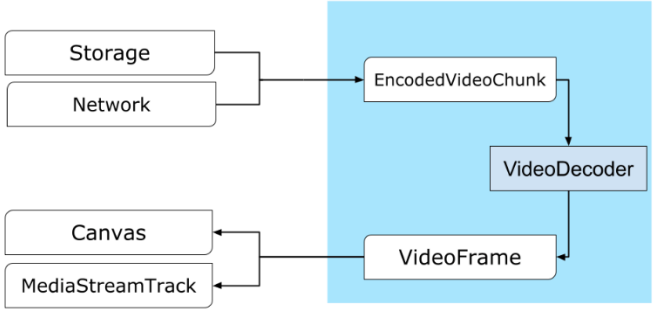


图 1.2 Webcodecs 框架下的解码过程

视频的来源可以是本地存储或者网络，经过处理，将视频解构成一个个EncodedVideoChunk，再经由 videodecoder.decode()就可以将 chunk 转换成videoFrame，videoFrame 可以在 canvas 上渲染出来。

Demo1 的整体流程图见下图 1.3。此处用文字大概说明 demo1 的处理过程。几个较为复杂的函数调用过程我使用不同颜色的箭头在图中做了标记。每个标记箭头上都有相应的数字，代表在当前函数调用过程中的先后顺序。在 index.html 中，将 demux_decode_worker.js 作为参数新建 worker 对象，获取 canvas 对象，通过 transferControlToOffscreen()将控制权交给 worker。

在 worker 中，首先根据路径新建 MP4Demuxer 对象，在新建 MP4Demuxer 对象之时，构造函数构造了 MP4Source 对象，并且绑定了 onready 函数和 onSample 函数。onReady 函数在“moov”框被解析的时候调用，它能够获取视频的 meta data。onSample 函数在视频的一组 sample 准备好时调用，在本 demo 中，onSamples 函数将 sample 组装为可以被解析的 EncodedVideoChunk，然后通过_onChunk 函数调用解析相应的 chunk。回到构建 MP4Source 对象的过程，绑定 onSample 和 onReady 函数之后，MP4Source 对象通过 fetch 方法获得相应的视频。当 fetch 方法获得返回值时，会递归调用 appendBuffer()直到返回 done。整个过程见图绿色线条。

接着，worker 对象新建 videodecoder 对象，并且绑定了回调函数，作用是将 frame 渲染到 canvas 上，并且在当前帧上输出 framestatus。

然后，异步调用 demuxer.getConfig()，这个函数的作用是获取视频的 extradata()，并且返回（整个步骤见图橙色线条）。在具体的函数调用过程中，getconfig()方法调用了 MP4Source.getInfo()方法，在 getInfo()中，尚未获得视频的 info 信息，接着程序通过回调函数在 onReady()函数中将 info 传递给了 getConfig()，getConfig()继续执行，调用 getExtradata()获得了当前视频的 extradata()并且返回。在这执行完之后，运行 getConfig 的 then()函数，在 then()函数中 decoder 将 config 作为参数调用了 configure()。

紧跟着是核心的 decode 过程（整个步骤见紫色线条）。先调用 MP4Demuxer.start()，并且将一个函数（decode frame）作为 onChunk 传入 start 函数中。Start()调用 MP4Source.start()，在该方法中，调用 file.setERxtractionOptions()

设置了要提取的 samples 所在的轨道的序号，然后程序开始提取 samples。当 samples（默认情况下包含 1000 个 sample）被准备好时，程序会自动调用 onSamples()，在 onSamples()中 sample 组装为可以被解析的 EncodedVideoChunk，然后通过_onChunk 函数调用解析相应的 chunk 获得 frame。

在此时，videodecoder 获得了 frame，就会执行之前绑定的回调函数，将当前的 frame 渲染到页面中。

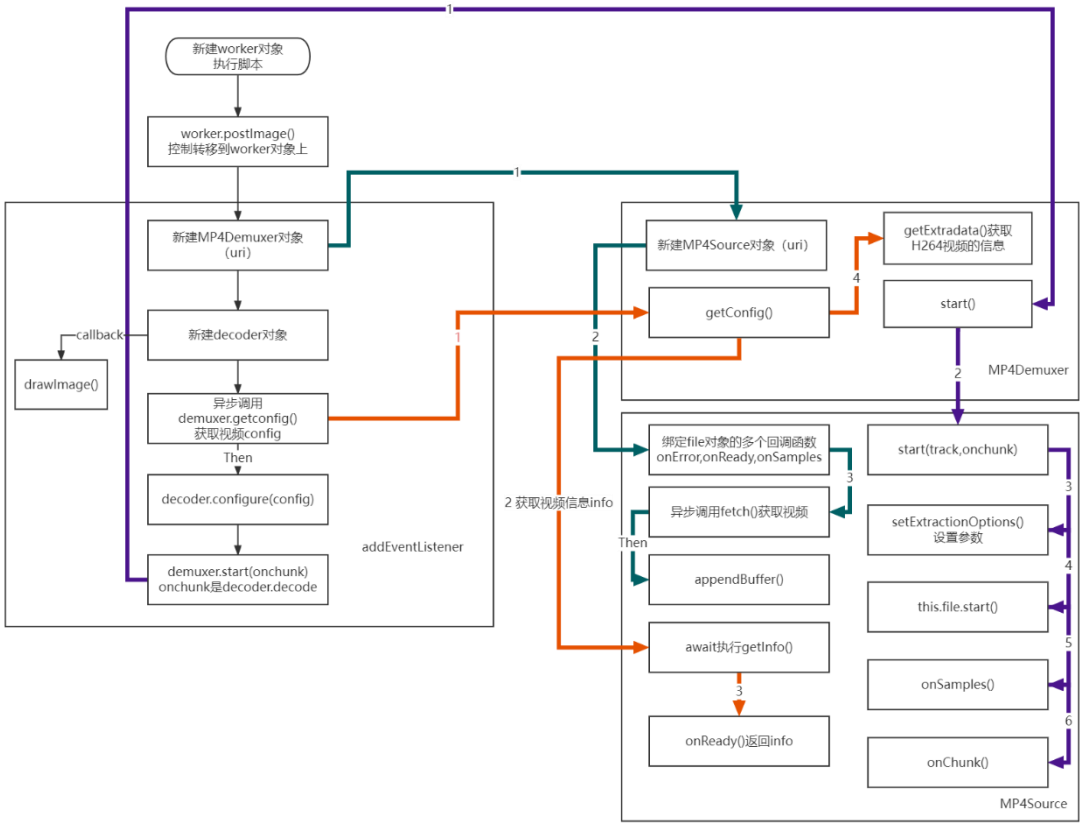


图 1.3 demo1 程序流程图

（2） demo2

整体流程图见下图 1.4。此处用文字大概说明 demo2 的处理过程。首先程序通过 fetch()函数获取 gif 源文件，然后进入 decodeImage()函数。接着，先判断当前浏览器是否支持 imageDecoder，我使用了 chrome 浏览器，在支持的情况下，继续执行程序。接着，程序以 fetch 的返回值为参数新建了 imageDecoder 对象，通过异步调用 imageDecoder.tracks 对象（是一个 tracklist，存储了可用的 tracks 并且提供方法解码 track 对象）的 ready 方法（一旦 ImageTrackList 填充了 track，返回一个 resolve 的 promise 对象），当其满足时，执行 logMetadata()方法，

logMetadata()会根据数据 buffered 状态来决定执行 logTracks()与否，当数据被完全 buffered 时，会再一次执行 logTrack()函数。

在上面的异步函数未决议之前，会执行 imageDecoder.decode(frameIndex)，参数 frameIndex 代表当前解码生成的 frame 的序号。Decode()的返回值时包含了 image 和 complete 两个参数的 promise，它的 then 函数时 renderImage()，renderImage()是一个递归函数，它会根据当前数据的状态，framecount 等信息决定是否返回（这意味着只有 gif 只有一帧），是否要重新循环，以及出现 RangeError 时要做出的响应。

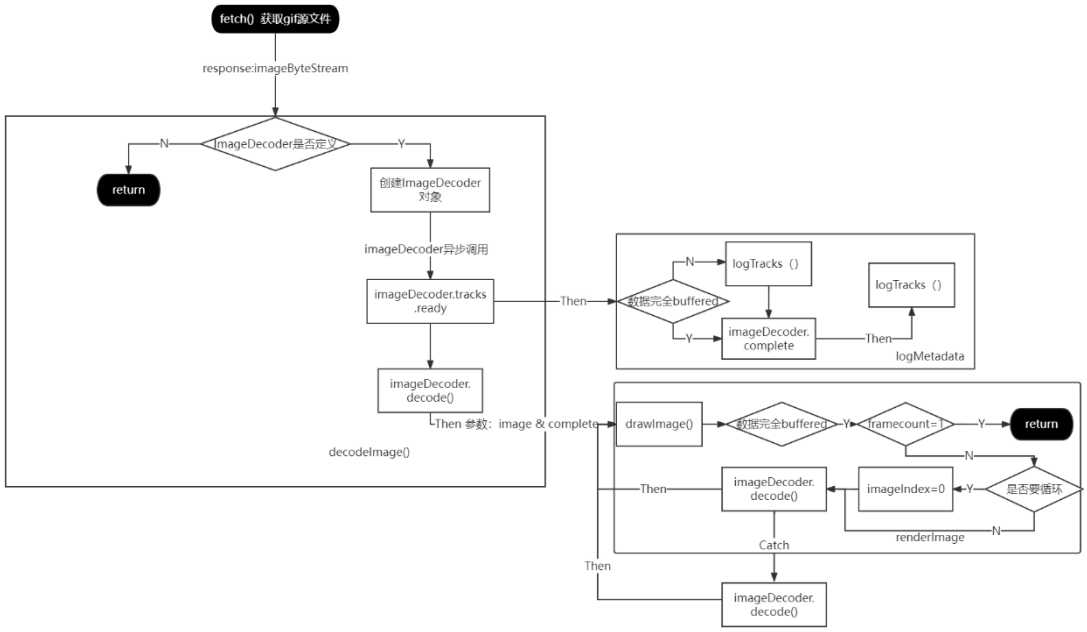


图 1.4 demo2 程序流程图

（3） demo3

根据资料 Webcodecs 框架下的解码过程如下图 1.5 所示。

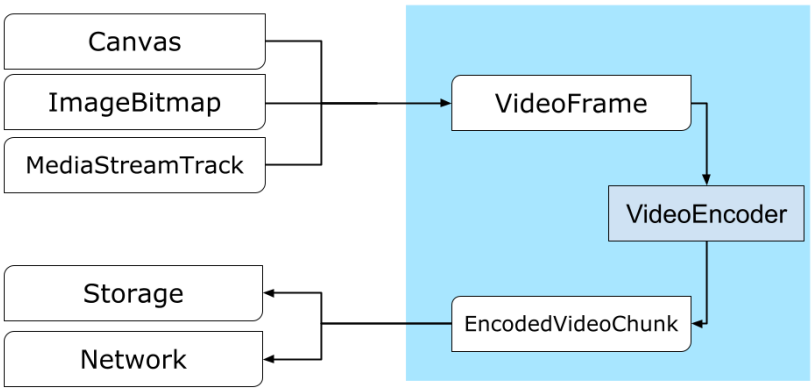


图 1.5 Webcodecs 框架下的解码过程

Raw 格式视频的来源可以是本地存储或者网络，经过处理，将视频分割成一个个 videoframe，再经由 videodecoder.encode()就可以将 chunk 转换成 EncodedVideoChunk，进而可以将其写入本地或者存储到其他地方。

record 过程的流程图见下图 1.6。此处用文字大概说明 demo3 的处理过程。window.navigator.mediaDevices.getUserMedia 提示用户给予使用媒体输入的许可，用户许可后，媒体会产生一个 MediaStream，在当前情况下包含了一个视频轨道。Constrains 是该方法需要的参数，它说明了请求的媒体类型和相对应的参数。程序将 getUserMedia 方法的返回值赋予 stream。

当按下 record 按钮时，程序会进行如下操作：首先，程序获得文件保存器在 js 中的抽象对象 FileSystemFileHandle，程序调用 stream.getTracks()获得首个 media streamtrack 对象。通过该 steamtrack 获得 track 的相关设置 tracksetting。程序新建 MediaStreamProcessor 对象，该对象以 track 为参数，通过调用 readable 使用 MediaStreamTrack 对象的源并生成 frame 流。接着将 encode-worker.js 作为参数新建 worker 对象，将控制权交给 worker。后序操作在 worker 中完成。

在 worker 中，调用 startRecording()方法，该方法接受之前的 filehandle，filestream 以及 tracksetting 作为参数。Worker 创建了一个可用于写入文件的 WritableFileStream 对象，由于要写入一个 webm 文件，worker 将该对象作为一个参数新建 WebMWriter 对象，此对象还规定了其他设置例如编码器，视频的宽度和高度等等。Worker 根据 fileStream 获得了 framereader 对象，framereader 可通过 read()获得视频的具体的一帧。程序以两个回调函数（init，error）为参数新建了 VideoEncoder 对象，init 的作用是将 videoEncoder 编码形成的 chunk 写入 webm 文件。接着，encoder 执行 configure()，将相关属性作为 encoder 的设置。frameReader 对象执行 read 函数并且添加 processFrame 作为 then 函数将获得的 frame 编码生成 chunk（当生成 chunk 时，videoEncoder 的回调函数会执行以将该 chunk 写入文件之中），processFrame 后序反复递归调用 read 函数直至用户按下 stop 按钮时，framestream 被关闭。

当按下 stop 按钮时，worker 先后停止 frameReader，webmWriter 以及 fileWriteStream 多个对象的工作。

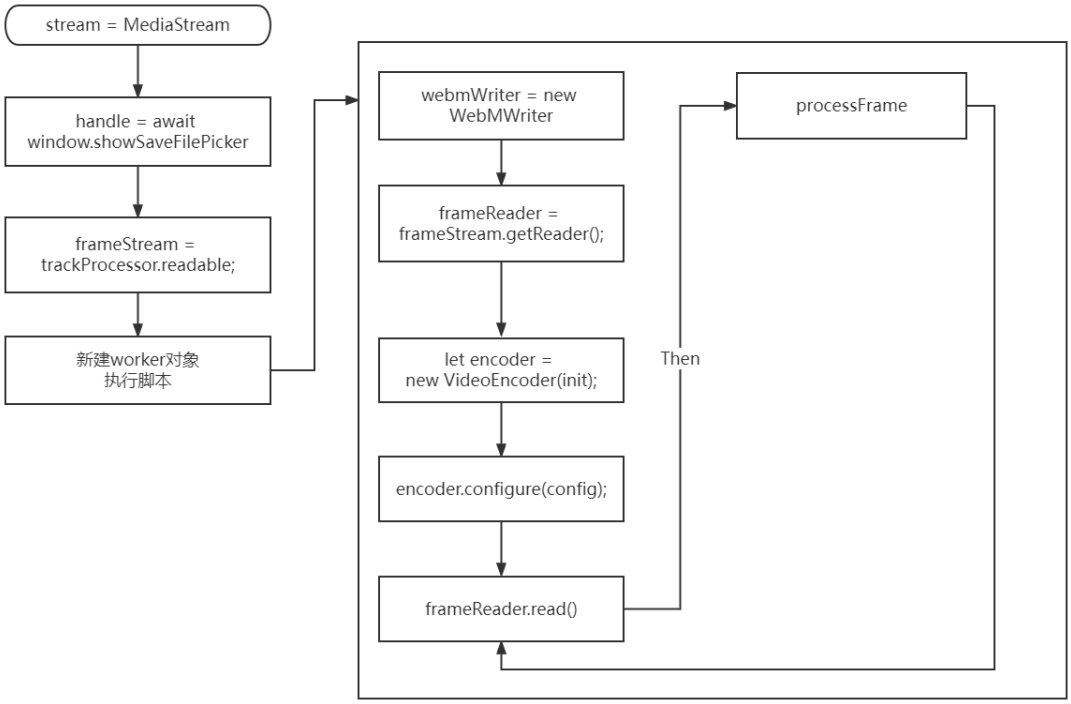


图 1.6 demo3 程序流程图

1.2 自选视频解码与视频格式分析

（1） 自选视频解码

Demo4 为失败的自选解码文件尝试（选用了 HEVC 视频，但是 webcodecs 框架目前不支持.h265 文件的视频解码，因此选用其他视频格式进行实验）。考虑到 av1 格式时一种开放，免费的影片编码格式，且其编码效率相对 HEVC 与 VP9 格式有了进一步提升。Demo5 选择了 av1 格式视频进行实验，解码出的 frame 格式为 YUV420。

Demo5 实现 decoder 解码文件的过程与 Demo1 整体大体相似，但是在新建 decoder 的回调函数中并不将得到的 frame 渲染到画布上，而是统计当前 frame 的大小，进而计算全部 frame 的整体大小，整体流程图见下图 1.7。demo5 程序流程图。解码生成的 frame 格式是 I420 格式。主要的区别在于，更改了 decoder 的 config 中的 extradata，将其适应为 webcodecs 下解码 av1 格式视频的要求。

其次，是在 getFrameStats 函数调用中记录解码的开始时间和具体解码某一帧的时间，以获得整个解码过程的运行时间。此外，在新建 decoder 的回调函数中，通过 frame.allocationSize()获得当前 frame 的大小，并且进行累加。

在实验中，我选用了源文件大小为 2MB 的一个文件，产生 2802 帧图像，解

码时长为 2.074 秒，解码出的文件大小为 8709120000byte，大约是 8.1GB。此外，我选用了源文件大小为 100MB 的一个文件，产生 25592 帧图像，解码时长为 18.115s，解码出的文件大小为 79601356800byte，大约是 74.1GB。

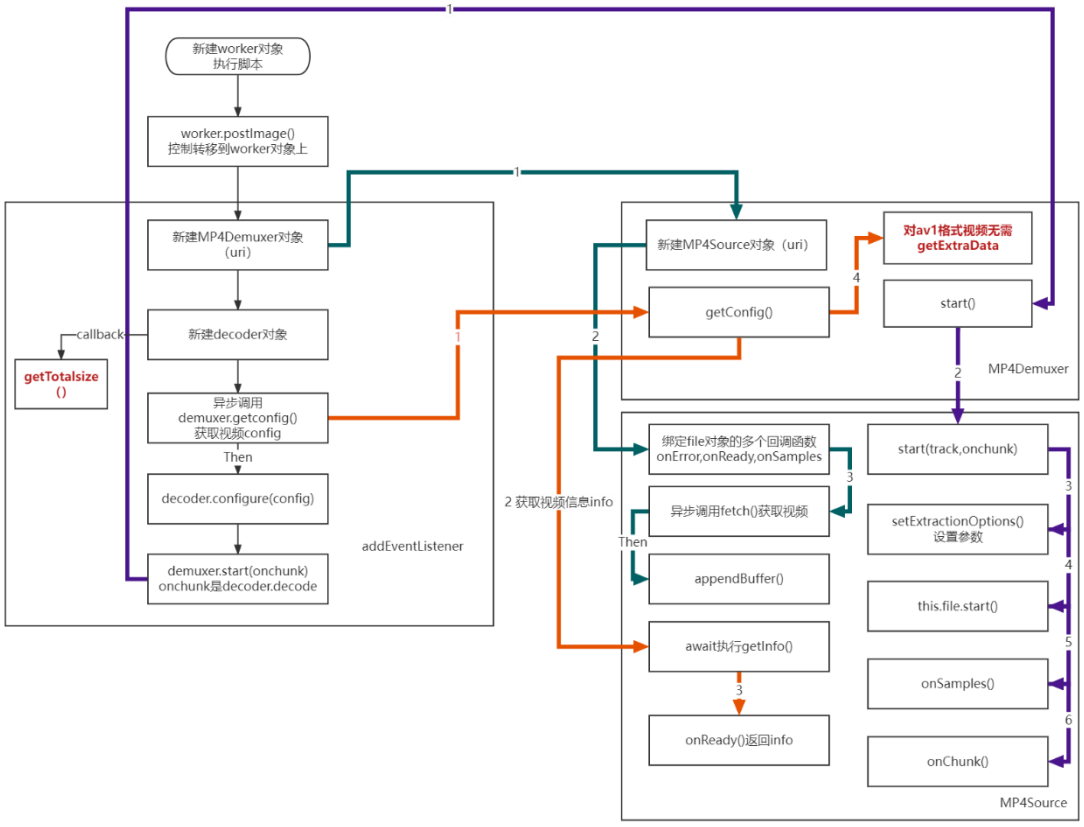


图 1.7 demo5 程序流程图

（2） 视频格式分析

根据资料，视频帧的编码方式主要分为 RGB 和 YUV 两种格式。

RGB 分别表示红（R）、绿（G）、蓝（B），也就是三原色，将它们以不同的比例叠加，可以产生不同的颜色。对于一张 1920*1080（有 1920*1080 个像素点）的图片，在采用 RGB 编码方式下，每个像素点有三个原色（每个原色占用 8 个 bit），每个像素点占用 24 个 bit，一张图片占用 1920*1280*3/1024/1024=7.03MB 存储空间。

YUV 编码采用了明亮度和色度表示每个像素的颜色。其中 Y 表示明亮度，也就是灰阶值。U、V 表示色度，描述的是色调和饱和度。对于 YUV 所表示的图像，Y 和 UV 分量是分离的。如果只有 Y 分量而没有 UV 分离，那么图像表示的就是黑白图像。由于人眼对亮度的分辨要比对颜色的分辨精细一些。可以

把色度信息减少一点，即并不是每个像素点都包含了 YUV 三个分量，根据不同的采样格式，可以每个 Y 分量都对应自己的 UV 分量，也可以几个 Y 分量共用 UV 分量，这样相比 RGB 格式，就能够节省许多的存储空间。

YUV 主流采样方式分为以下三种：YUV4：4：4 采样，YUV 4：2：2 采样以及 YUV4：2：0 采样。YUV4：4：4 采样的情况下，每个像素的三个分量的信息都是完整的，每个像素都占用 24 个 bit，即三个字节。这种的采样方式和 RGB 图像的大小是相同的。见下图 1.8 描绘了 YUV4：4：4 的采样方式。

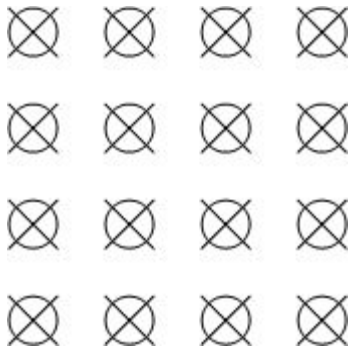


图 1.8 YUV 4：4：4 采样方式

对于 YUV4：2：0 方式的采样，每采样一个像素点，都会采样其 Y 分量，而 U 和 V 分量会间隔采集一个，映射为像素点时，前两个像素点会公用 U 和 V 分量，从而节省了图像空间，对于一个 1920*1280 的图片，采用 YUV4：2：2 采样时的大小为（1920*1280*8＋1920*1280*0.5*8*2）/8/1024/1024=4.68MB，通过对比，可以发现一帧图像的存储比 RGB 和 YUV4：4：4 节省了三分之一的空间。下图 1.9 描绘了 YUV4：2：2 的采样方式。

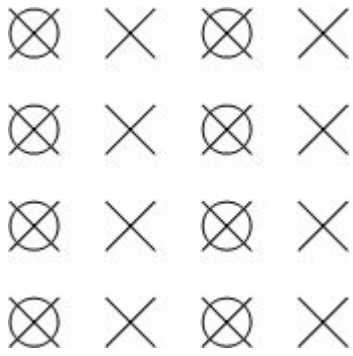


图 1.9 YUV 4：2：2 采样方式

YUV4：2：0 并不意味着不采样 V 分量。它指的是对每条扫描线来说，只有一种色度分量以 2:1 的采样率存储，相邻的扫描行存储不同的色度分量。对于一

个 1920*1280 的图片，采用 YUV4：2：0 采样时的大小为（1920*1280*8 + 1920*1280*0.25*8*2）/8/1024/1024=3.51MB，大约相比 RGB 格式的采样方式节省了一半的空间。

在 demo5 中，经由 videodecoder.decode()解码出来的视频帧的格式为 YUV420 编码的，想对以上多种编码方式来说，属于是最节省空间的格式了。

2. oneVPL

目前为止，基于 oneVPL 实现 decoder 以解码 av1 文件尚未实现。我的思路是仿照 hello-decode（解码 HEVC 的一个 decoder demo），实现 av1 文件的 decoder，但是在改写的时候遇到了 bug，并且用 gdb 调试时发现无法查看库函数的实时运行代码（疑似因库的.so 文件并不是使用-g 链接生成），因此暂时放弃自己实现 decoder，使用 oneVPL 的 sample_decode 先行进行实验。

接下来的内容将首先分析 hello-decode 的代码结构和实现功能的步骤。然后将使用 sample_decode 进行实验并且获得测量结果。

2.1 demo:hellodecode

hellodecode 的调用是通过命令行进行的。示例的调用方式为”hello-decode -sw -i input.h265 ”，-sw 是 decode 的执行形式，代表的是使用 CPU 进行解码，与之相对应的还有-hw，是用 GPU 进行解码。其源代码通过 ParseAndValidate 函数对传入的参数进行检验。如果传入的参数不符合相关要求，那么将直接返回。

在调用 oneVPL 的函数之前，程序必须要创建 oneVPL session。通过调用 MFXload()和 MFXcreateSession()以创建 oneVPL session。由于 oneVPL 的函数具备多样的实现方式，因此 oneVPL dispatcher 会根据给定的实现要求选择相应的实现方式。实现要求包括了编解码器，VPP 过滤器的信息等等。配置 dispatcher 的实现要求一般通过以下代码：

1. Create loader with `MFXLoad()`.
2. Create loader's configuration with `MFXCreateConfig()`.
3. Add configuration properties with `MFXSetConfigFilterProperty()`.
4. Explore available implementations with `MFXEnumImplementations()`.
5. Create a suitable session with `MFXCreateSession()`.

The procedure to terminate an application is as follows:

1. Destroy session with `MFXClose()`.
2. Destroy loader with `MFXUnload()`.

在实际的 hello-decode 函数中，程序反复创建 config，并且通过调用 setConfigFilterProperty 的方法限定了“MFX_IMPL_SOFTWARE”，“MFX_CODEC_HEVC”以及“2.2”三个标准。即实现方式（软件解码），解码器类别（HEVC 解码器），以及 api 的版本要求（2.2 版本及以上）。

接下来，程序在内存中开辟空间，并且让 mfxbitstream 指向空间的首地址，同时规定了该比特流的解码器。程序从源文件（视频文件）读取一部分信息到比特流中，进而创建变量 docodeParams，使用 decodeHeader()从比特流中读取信息并且填充到 decodeparams 中（我的 av1 改写版程序在这里出了问题，前面一直是正常的，猜测问题和 av1 编码方式有关）。

根据先前得到的 decodeparams 以及 session 就可以执行 MFXVideoDECODE_Init()，在该函数中，会对 decode 进行参数配置。然后，在一个 while 循环中，反复执行 readEncodedStream 和 decodeFrameAsync()两个函数，前者是从源文件中读取数据加载到 bitstream 中，后者是对数据进行提取和解码。在解码完成之后，调用 close()即可结束整个 decode 流程。

oneVPL 中，函数的返回值类型大多为 mfxStatus，当其值为 MFX_ERR_NONE 时，代表当前函数运行正常。因此，在许多没有明确返回值的场合，可以通过 mfxStatus 来判断函数是否执行顺利。

纵观下来，hello-decode 的实现并不难懂，但是由于 av1 格式和 hevc 格式存在一定的差别，我还需要一段时间将其改写。

2.2 sample_decode 的运行结果

在本地和 devcloud 上分别运行 sample_decode，对于 2m 大小的视频， cpu 运行结果见下图 2.1。Cpu 运行时间为 5.67 秒。产生的文件大小见下图 2.2，经过换算大约是 8G。对于 100m 大小的 av1 格式视频，由于系统的空间限制，无法

做到完全 decode。见下图 2.3。

```
wudi@wudi-pc:~/Downloads$ python ./gettime.py
CONFIGURE LOADER: required implementation: sw
CONFIGURE LOADER: required implementation mfxAccelerationMode: MFX_ACCEL_MODE_NA
Loaded Library configuration:
    Version: 2.6
    ImplName: oneAPI VPL CPU Implementation
    Adapter number : -1
    DRMRenderNodeNum: 0
Used implementation number: 0
Loaded modules:
   0: /home/wudi/intel/oneapi/vpl/2022.1.0/lib/libvplswref64.so.1

pretending that stream is 30fps one
Decoding Sample Version 8.4.27.0


Input video     AV1
Output format   IYUV
Input:
  Resolution    1920x1080
  Crop X,Y,W,H  0,0,1920,1080
Output:
  Resolution    1920x1080
Frame rate      30.00
Memory type             system
MediaSDK impl           sw
MediaSDK version        2.6

Decoding started
Frame number: 2802, fps: 495.083, fread_fps: 0.000, fwrite_fps: 589.908
Decoding finished
running time: 5.66676020622 Seconds
```

图 2.1 2M av1 源文件 CPUdecode 结果

```
wudi@wudi-pc:~/Downloads$ ll
总用量 36254104
drwxr-xr-x  7 wudi wudi       4096 7月  11 22:31  ./
drwxr-xr-x 34 wudi wudi       4096 7月  11 22:26  ../
-rw-rw-r--  1 wudi wudi  588608845 7月   4 15:06  1988.mp4
drwxrwxr-x  2 wudi wudi       4096 7月   3 19:14  7b099f98f6b373699aa2f54e5d6ccb58-d9b02f5788922b94cf6a40155af1521d3c9ca262/
-rw-rw-r--  1 wudi wudi    4456448 7月   4 13:00  av1ma.mkv
-rw-rw-r--  1 wudi wudi    7923051 7月   4 15:26  cattest.av1
-rw-rw-r--  1 wudi wudi    4254777 7月   6 23:00  cattest.ivf
-rw-rw-r--  1 wudi wudi    9275810 7月   4 15:18  cattest.mp4
-rw-rw-r--  1 wudi wudi 8715340800 7月  11 15:58  cattest.yuv
drwxrwxr-x  5 wudi wudi       4096 4月  28 21:49  clash/
-rw-rw-r--  1 wudi wudi   97340001 4月  28 21:06  Clash.for.Windows-0.19.16-x64-linux.tar.gz
-rw-rw-r--  1 wudi wudi   82686964 5月  10 13:10  code_1.67.0-1651667246_amd64.deb
-rw-rw-r--  1 wudi wudi   12041395 6月  28 14:41  ffmpeg-snapshot.tar.bz2
-rw-rw-r--  1 wudi wudi    2844040 7月   7 12:07 'gdb-oneapi-2022.2 (1).pdf'
-rw-rw-r--  1 wudi wudi    2844040 7月   7 12:07  gdb-oneapi-2022.2.pdf
-rw-rw-r--  1 wudi wudi        187 7月  11 15:58  gettime.py
-rw-rw-r--  1 wudi wudi   85757216 6月  27 19:06  google-chrome-stable_current_amd64.deb
-rw-rw-r--  1 wudi wudi        147 7月  11 17:03  job.sh
-rw-rw-r--  1 wudi wudi 3404046106 6月  27 19:47  l_BaseKit_p_2022.2.0.262_offline.sh
drwxrwxr-x  6 wudi wudi       4096 7月   5 16:44  libva-2.15.0/
-rw-rw-r--  1 wudi wudi     500858 7月   5 16:40  libva-2.15.0.tar.bz2
drwxr-xr-x  7 wudi wudi       4096 3月  29 17:12  l_oneVPL_p_2022.1.0.154/
-rwxrwxr-x  1 wudi wudi   17432298 7月   5 18:01  l_oneVPL_p_2022.1.0.154.sh*
-rw-rw-r--  1 wudi wudi  109548964 7月   4 18:25  mani.av1
-rw-rw-r--  1 wudi wudi   84236474 7月   4 15:45  mani.mp4
-rw-rw-r--  1 wudi wudi  120863980 7月   3 12:48  microsoft-edge-stable_103.0.1264.44-1_amd64.deb
drwxrwxr-x  4 wudi wudi       4096 7月   4 12:07  nv-codec-headers/
-rw-rw-r--  1 wudi wudi     531772 7月   3 19:08  openssl_1.1.0g-2ubuntu4_amd64.deb
-rw-rw-r--  1 wudi wudi 7328102400 7月   7 10:36  out.i420
-rw-rw-r--  1 wudi wudi 7328102400 7月   6 20:43  out.raw
-rw-rw-r--  1 wudi wudi   17369458 7月   5 13:17  rick.mp4
-rw-rw-r--  1 wudi wudi   18421058 7月   5 13:21  roll.h265
-rw-rw-r--  1 wudi wudi       5364 7月   4 19:05  setup-devcloud-access-156846.txt
-rw-rw-r--  1 wudi wudi      12955 7月   3 21:08  streams_av1.webm
-rw-rw-r--  1 wudi wudi    2921786 7月  10 14:00  testav1.ivf
-rw-rw-r--  1 wudi wudi    2907995 7月  10 13:54  testav1.mp4
-rw-rw-r--  1 wudi wudi 8715340800 7月  11 22:31  testav1.yuv
-rw-rw-r--  1 wudi wudi    2907995 7月   3 20:37  test.mp4
-rw-rw-r--  1 wudi wudi     198008 4月  28 21:31  v2rayse_2022_04_28.yaml
-rw-rw-r--  1 wudi wudi    1927488 7月   4 14:39 'VideoConverter_vc142-zx-160@7625873690244933789.exe'
-rw-rw-r--  1 wudi wudi  259614528 7月   4 10:49  videoma.mp4
-rw-rw-r--  1 wudi wudi   94013124 7月   3 18:01  videoplayback.mp4
-rw-rw-r--  1 wudi wudi    1685392 6月  27 19:47  VisualStudioSetup.exe
```

图 2.2 2M av1 源文件 CPUdecode 产生的 yuv 文件大小

图 2.3 100M av1 源文件 CPUdecode 产生报错

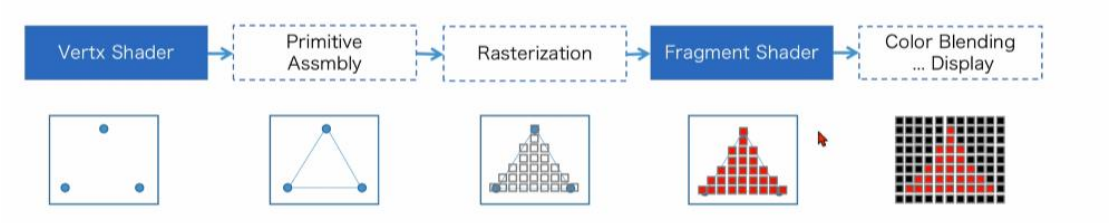
3. WebGPU

目前，我参考教程实现了 rotatingCube 这个 sample，对于在 Webcodecs 框架

下使用 WebGPU 进行加速解码，由于缺少实际的 demo 指引，还在摸索的过程中，并未实现。目前的计划是查询 webgl 加速视频解码或者渲染的 demo 以潜移默化到在 webgpu 下的 demo。

3.1 sample:rotatingCube

渲染 pipeline 的过程详细见下图 3.1。开发人员编写 vertx shader 和 fragment shader，其他复杂的操作由系统内部实现：primitive assembly 能将 vertx shader 输出的顶点数据集合成一个图元，rasterization 将图转化成一个个栅格形成的图像，每个元素对应着其中的一个像素。



在 rotatingCube 这个 sample 中，最后的展现图像是一个旋转的正方体。见下图 3.1。rotatingCube 代码流程图见下图 3.2。

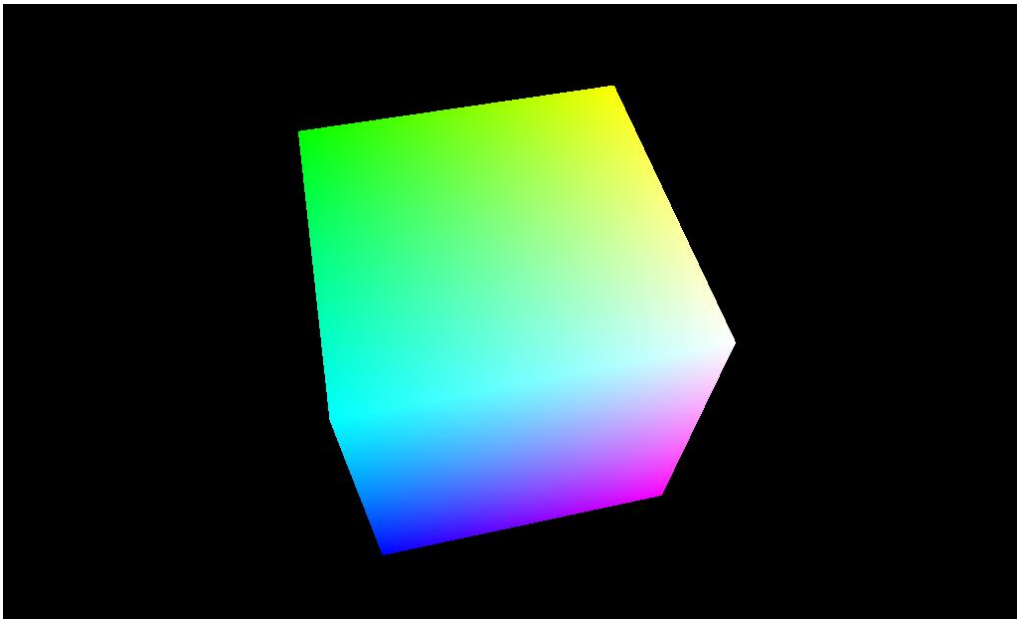


图 3.1 rotatingCube

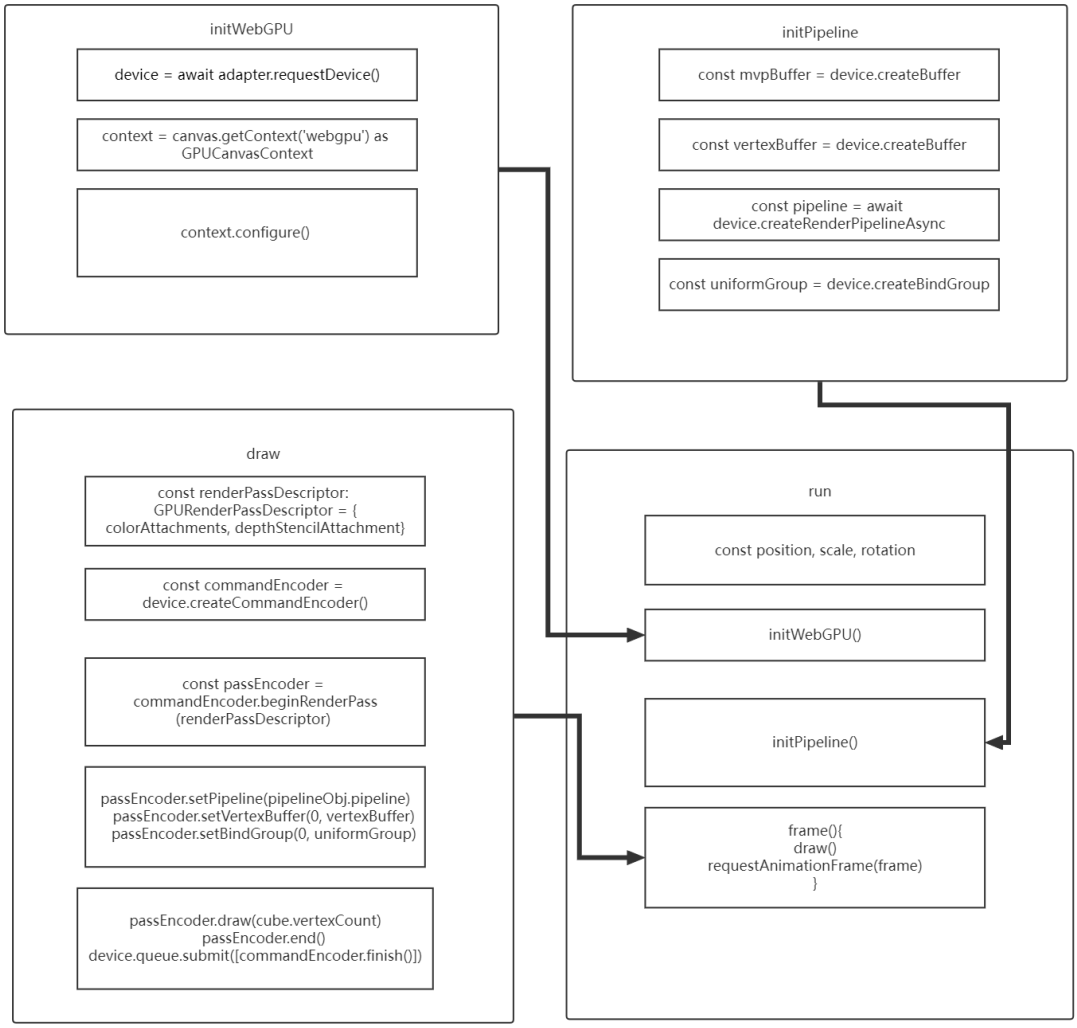


图 3.2 代码流程图

在 initWebGPU()中，程序获得了 adapter（API 和 WebGPU 的中介）与 device（GPU 的逻辑设备）。通过调用 device 的函数，程序可以实现编写 GPU 程序，开辟 GPU 显存空间，创建指令编码器（commandencoder）等等。

在 initPipeline()中，程序创建了 mvpbuffer（用来存储后序计算生成的 mvp 矩阵）以及 vertexBuffer（用来存储将要构建的图形的顶点数据）。同时编写了 vertexshader 文件与 fragmentshader 文件，并且组合 shader 文件创建 pipeline，通过定义 primitive，程序确定了构建图形的方法，渲染重叠图层的方式以及深度检测功能的开启与否，程序创建 uniformgroup 包裹 mvpbuffer 以方便在 GPU 中读取 buffer。

在 draw()中，程序通过 device.createCommandEncoder 创建了指令编码器，进而创建更加具体的 passEncoder，通过在 passEncoder 中传入之前定义的 mvpbuffer

以及 vertexbuffer 以及 pipeline，调用 draw()传入要渲染的 vertex 个数，然后调用 end 与 submit 将所有数据传递给 GPU 进行运行。

实际上，整个绘图过程在 run()函数中执行（图中右下角框）。run()函数中定义了多个矩阵 position，scale 以及 rotation 三个矩阵，分别代表图形的位移，放缩以及旋转步骤的矩阵表示。在 run()函数中的 frame()中，通过 getMvpMatrix()获得了 MVP 矩阵，并且存储进 gpubuffer 中。通过在 frame()中动态更新三个矩阵的值，并且调用 draw()，实现了旋转方块的渲染。

4．实现方案结合

目前的想法是在 webcodecs 和 webgpu 结合的基础上，利用 webgpu 的并行计算效果，在每一个并行运行的程序中使用 oneAPI 加速处理视频。