Design Doc: Enable Video Capture Service for Chromium for ChromeOS

This Document is Public

Authors: chfremer@chromium.org

One-page overview

Summary

Chromium for ChromeOS currently runs a (legacy) video capture stack that is integrated into the Browser process. The goal is to switch to using the *video capture service*, which has already rolled out on Windows, Mac OS, and Linux, (not ChromeOS), on M61 and is in the process of rolling out on Android. As is the case on Windows, Mac OS, and Linux, the goal is to run the service in a dedicated unsandboxed utility process. Sandboxing is a future goal beyond the scope of this rollout.

Platforms

ChromeOS

Teams Involved

WebRTC-MTV team (point of contact: chfremer@chromium.org)
ChromeOS camera capture (point of contact:jcliang@chromium.org)

Bug

Launch bug

Code affected

Video capture library (media/capture/video/*)
Browser-side video capture stack (content/browser/renderer_host/media/*)

Design

For the video capture service in general, see this design doc.
The service has already rolled out on other platforms with M61. On ChromeOS, there are the following differences that need to be addressed:

1. A gpu::GpuMemoryBufferManager* and a MojoJpegDecodeAcceleratorFactoryCB (and, soon, also a MojoJpegEncodeAcceleratorFactoryCB) need to be passed to the media::VideoCaptureDeviceFactory. This is to enable support for hardware-accelerated MJPEG decoding in the CameraHalServer.
2. A media::VideoCaptureJpegDecoderFactoryCB needs to be passed to the VideoCaptureDeviceClient. This is to enable support for hardware-accelerated MJPEG decoding at the level of VideoCaptureDeviceClient.

Proposed solution for 1:
In the (legacy) non-service case, the gpu::GpuMemoryBufferManager* dependency is satisfied by providing a global instance owned by the Browser process. The MojoJpegDecodeAcceleratorFactoryCB dependency is satisfied by delegating to an instance of viz.mojom.GpuService available to the Browser process.

To satisfy both dependencies in the service case, a simple solution would be to have the Browser process inject them into the video capture service right after it connects. To this end, the video capture service would expose a new method in interface DeviceFactoryProvider

```
InjectGpuDependencies(GpuDependencies gpu_dependencies);
```

with

```
interface GpuDependencies {
    CreateJpegDecodeAccelerator(media.mojom.JpegDecodeAccelerator&
accelerator);
    CreateJpegEncodeAccelerator(media.mojom.JpegEncodeAccelerator&
accelerator);
    CreateGpuMemoryBuffer(gfx.mojom.GpuMemoryBufferId id,
                          gfx.mojom.Size size,
                          gfx.mojom.BufferFormat format,
                          gfx.mojom.BufferUsage usage)
        => (gfx.mojom.GpuMemoryBufferHandle buffer_handle);
    DestroyGpuMemoryBuffer(gfx.mojom.GpuMemoryBufferId id,
                           gpu.mojom.SyncToken sync_token);
  };
```

To satisfy the MojoJpegDecodeAcceleratorFactoryCB dependency, the service would then delegate to `gpu_dependencies.CreateJpegDecodeAccelerator`. To satisfy the gpu::GpuMemoryBufferManager* dependency, it could obtain a connection to `viz.mojom.GpuService` via `gpu_dependencies.ConnectToGpuService` and use that to create a viz::ServerGpuMemoryBufferManager.

Proposed solution for 2:
A possible solution would be to use the MojoJpegDecodeAcceleratorFactoryCB from 1. to do the accelerated MJPEG decoding inside VideoCaptureDeviceClient. This would require correspondingly modifying VideoCaptureDeviceClient and VideoCaptureGpuJpegDecoder.

Implementation Plan

CL Title/Link	CL Description	Status
[Video Capture Service] Support accelerated jpeg decoding	[Video Capture Service] Support accelerated jpeg decoding * Move/rename class content::VideoCaptureGpuJpegDecoder to media::VideoCaptureJpegDecoderImp and break dependencies to |content| in order to make it reusable for the video capture service * Inject dependencies on gpu_service from Browser into video capture service. I chose this solution of having the video captuer service "share" the Browser's connection to gpu_service, since I could not see any straightforward way to get a dedicated connection from the video capture service to the gpu_service.	Landed
[Video Capture Service] Operate jpeg decoder IO to gpu process on separate thread	[Video Capture Service] Operate jpeg decoder IO to gpu process on separate thread Adds a dedicated thread to the video capture service for operating the IO between the capture device and the gpu process for acceleraged Mjpeg decoding. This fixes a deadlock during shutdown of video capture in the video capture service after a	Landed

	session that uses the accelerated jpeg decoder. See the crbug for details. The deadlock and the fix for it can currently only be provoked/verified manually by running a ChromeOS device with a camera attached and command-line flag --enable-features=MojoVideoCapture. Getting coverage in automated integration testing is possible but requires modification of the fake video capture device implementation, see crbug/852606	
[Video Capture Service, ChromeOS] Separate startup of CameraHalDelegate from instantiation of VideoCaptureDeviceFactory	Separate startup of CameraHalDelegate from instantiation of VideoCaptureDeviceFactory by extracting the corresponding constructs into a new class CameraHalContext and creating an instance in MediaStreamManager. For ChromeOS builds that us the cros_camer_service for video capture, the CameraHalDelegate must be started as part of Chrome startup. This startup has for now been tied to the instantiation of VideoCaptureDeviceFactoryChromeOS. With the move to the VideoCaptureService, the instantiation of VideoCaptureDeviceFactoryChromeOS no longer happens on Chrome startup, but instead happens on-demand and potentially more than once.	Landed

Metrics

Success metrics

The rollout will be considered successful if it does not significantly change any of the existing tests or UMA stats for video capture.

- Media.VideoCaptureManager.Event
- Media.VideoCaptureService.Event
- Media.VideoCapture.DelayUntilFirstFrame

TODO: Do we already collect metrics showing the usage of accelerated MJPEG decode in ChromeOS devices? Do we collect factors such as CPU usage for that as well?

Regression metrics

Same as above.

Experiments

No experiment planned beyond the standard experiment-controlled rollout.

Rollout plan

Use a standard experiment-controlled rollout targeting M69. This matches what has been done for rolling out the service on other platforms with the rationale being that changes to video capture behavior are heavily device dependent and therefore warrant the extra safety of being able to quickly roll things back if issues emerge.

Core principle considerations

Speed

No change in runtime resources is expected

Security

No change

Privacy considerations

No change

Testing plan

There are three independent variables influencing which code paths are taken when doing video capture on ChromeOS:

- Video Capture Service: enabled vs. disabled (let's call this VCS vs. NonVCS)
- Camera Delivers Mjpeg: yes vs. no (let's call this Mjpeg vs. NonMjpeg)
- Video Capture Implementation: cros_camera_service vs. Linux V4L2 (let's call this CCS vs V4L2). The CCS is a ChromeOS service for video capture on ChromeOS that

enables access to the cameras for both Chrome and the Android Runtime. It is currently enabled for the following 3 boards: soraka, nautilus, dru.

All 8 possible combinations of these variables need to be tested.

Test Coverage in Chromium Commit Queue (CQ)

There are integration tests running in the Chromium CQ the exercise video capture for all 4 combinations of VCS vs. NonVCS and Mjpeg vs. NonMjpeg. These tests use a fake device and a fake jpeg decoder. The tests run on all platforms-specific builds, including for ChromeOS.

Test Coverage in WebRTC Bots

There are integration tests running on WebRTC Bots that exercise video capture using a real webcam. These tests run on Windows, Mac, Linux, and Android. They do not run on ChromeOS. Since accelerated Mjpeg decoding is currently only enables on ChromeOS, these tests only cover the NonMjpeg cases (aside from a bit of factory logic up until the point where it is decided that jpeg decoding is disabled).

Test Coverage in ChromeOS Autotests

- We have a test to make sure we don't fallback to use SW decoder. It is based on the CrOS camera basic functional test. The test CL is here. https://chromium-review.googlesource.com/c/chromiumos/third_party/autotest/+/958742
- We also have another test to just test the MojoChannel and Jpeg Decode function (https://chromium-review.googlesource.com/c/chromiumos/platform/arc-camera/+/721403/33/common/jpeg/jpeg_decode_accelerator_test.cc). There is no autotest for it since the first one covers this case.

TODO: Get more clarity on what test coverage there is on ChromeOS for general video capture via, e.g. Autotests, performance tests.

How to test manually

In general, it is sufficient to test if displaying video from a local camera works. To this end, a suitable test page is https://webrtc.github.io/samples/src/content/getusermedia/resolution/.

To test the Mjpeg cases, navigate to https://webrtc.github.io/samples/src/content/getusermedia/resolution/ and choose a resolution of HD or higher. Webcams typically use Mjpeg for HD or higher resolutions, especially when being connected via USB2. To confirm that Mjpeg is being used, navigate to chrome://histograms/Media.VideoCaptureGpuJpegDecoder.InitDecodeSuccess and check if

an event for InitDecodeSuccess was collected. A new such event should be generated for each video capture session where hardware-accelerated Mjpeg decoding is being used. To test the nonMjpeg cases, use a resolution of QVGA or VGA. To confirm it is NonMjpeg, check that no additional event for InitDecodeSuccess has been generated.

Switching between VCS and NonVCS cannot be done from the browser UI directly, since this is controlled via a Finch flag and/or command-line flag. To manually force VCS or NonVCS, the command-line flag --enable-features=MojoVideoCapture or --disable-features=MojoVideoCapture needs to be passed to the Chrome executable. To confirm that VCS is being used, navigate to chrome://histograms/VideoCaptureService.Event. In the VCS case, events other than event 0 will be generated. In the NonVCS case, an event 0 will be generated (and no other events) once during Chrome startup.

To test the CCS cases, use a ChromeOS device with one of the following boards: soraka, nautilus, dru. To confirm that CCS is being used, navigate to chrome://media-internals and check the Capture API in the Video Capture tab. Note that CCS currently does not yet support external cameras, so testing must be done using the integrated cameras. If CCS is used, it should say “Camera API2 Limited”. If V4L2 is used it will say “V4L2 SPLANE”. To test the V4L2 cases, use a ChromeOS device with any board other than the ones for CCS.

Manual test coverage

Manual tests performed by chfremer@ on 06/21/2018 using local build including the first 3 CLs listed above boards guado (for V4L2) and fizz (for CCS). Results:

Mjpeg vs. NonMjpeg	**VCS vs NonVCS**	**V4L2 vs CCS**	**Test Result**	**Notes**
NonMjpeg	NonVCS	V4L2	Pass	
Mjpeg	NonVCS	V4L2	Pass	
NonMjpeg	VCS	V4L2	Pass	
Mjpeg	VCS	V4L2	Pass	
NonMjpeg	NonVCS	CCS	Pass	
Mjpeg	NonVCS	CCS	Pass	
NonMjpeg	VCS	CCS	Pass	
Mjpeg	VCS	CCS	Pass	

Followup work

Remove legacy code path as soon as the video capture service has been rolled out on all platforms. ChromeOS is the last remaining platform.