GpuMemoryBuffer

Shared memory that can be accessed by the GPU

reveman@chromium.org

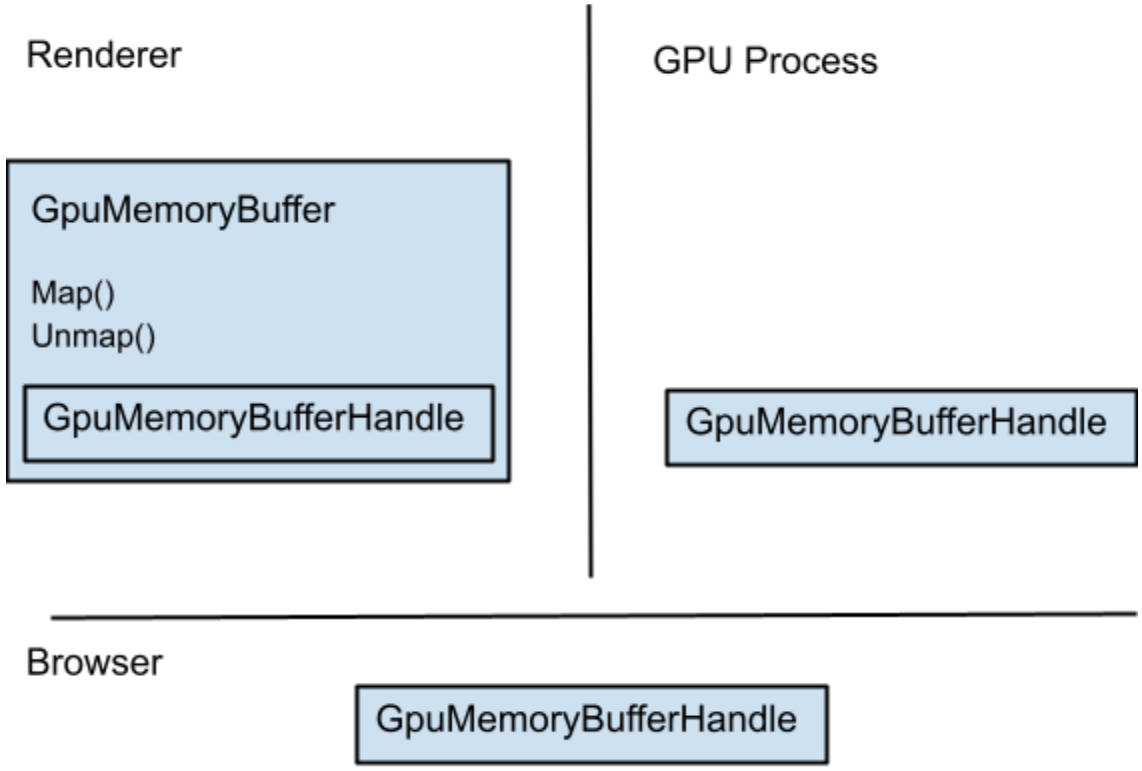
The GpuMemoryBuffer framework provides a system for reading/writing directly to memory that the 3D graphics hardware can use for rendering without any costly copying having to be done on the GPU process side. Initial support is focused on texture data but it could also be used for vertex data and possibly even for writing directly to a command buffer.

Some parts of this system need to be implemented differently depending on the platform and in some cases differently depending on the driver extensions available. The interfaces described here provide the abstractions necessary to allow this and intend to be unchanged across platforms and drivers.

GpuMemoryBuffer
The GpuMemoryBuffer class itself is a client side object. It provides the methods required to map/unmap a buffer. Safe access (not writing to the buffer while the graphics hardware is reading from it) is outside the scope of the APIs described here and will be provided by the CHROMIUM_sync_query[1] extension.

GpuMemoryBufferHandle
A GpuMemoryBufferHandle is a platform specific identifier that can be serialized. A GpuMemoryBuffer instance is created from a GpuMemoryBufferHandle. The procedure required to create a GpuMemoryBuffer on the renderer side is similar to allocating general shared memory. The browser process need to allocate the buffer (possibly through GPU process IPC) and it needs to be shared and registered with the GPU process before the renderer can use it in the command buffer.

[1] https://docs.google.com/document/d/1xwoLmXHPzs5TysV96V3Azml2VReLJK_TnC0Bg3CCAx0/edit



The GpuMemoryBuffer class and GpuMemoryBufferHandle provide what's needed to create and map buffers in the renderer process. However, this is not enough to use these buffers with chromium's GLES implementation. In the case of texture data and zero-copy texture updates this is provided by the GLImage framework.

Texture updates using CHROMIUM_image

GpuMemoryBuffers make it possible to update textures without having to perform a texture upload. Eliminating an unnecessary copy, and maybe even more importantly, makes it easy to keep the full cost of updating a texture in a background process/thread instead of in the GPU process where it affects our ability to effectively produce new frames.

GLImage and CHROMIUM_texture_from_image

GLImage is a service side framework that allow different type of objects to be bound to textures. ie. it allows the browser process to bind IOSurfaces to textures. The CHROMIUM_texture_from_image extension provides a GLES interface for binding a GLImage to a texture once it has been registered on the service side. The ability to create a GLImage from a GpuMemoryBuffer provide what's necessary to use these buffers together with GLES commands.

```
// GL_CHROMIUM_texture_from_image
void bindTexImage2DCHROMIUM(GLenum target, GLuint image_id);
void releaseTexImage2DCHROMIUM(GLenum target, GLuint image_id);
```

CHROMIUM_image

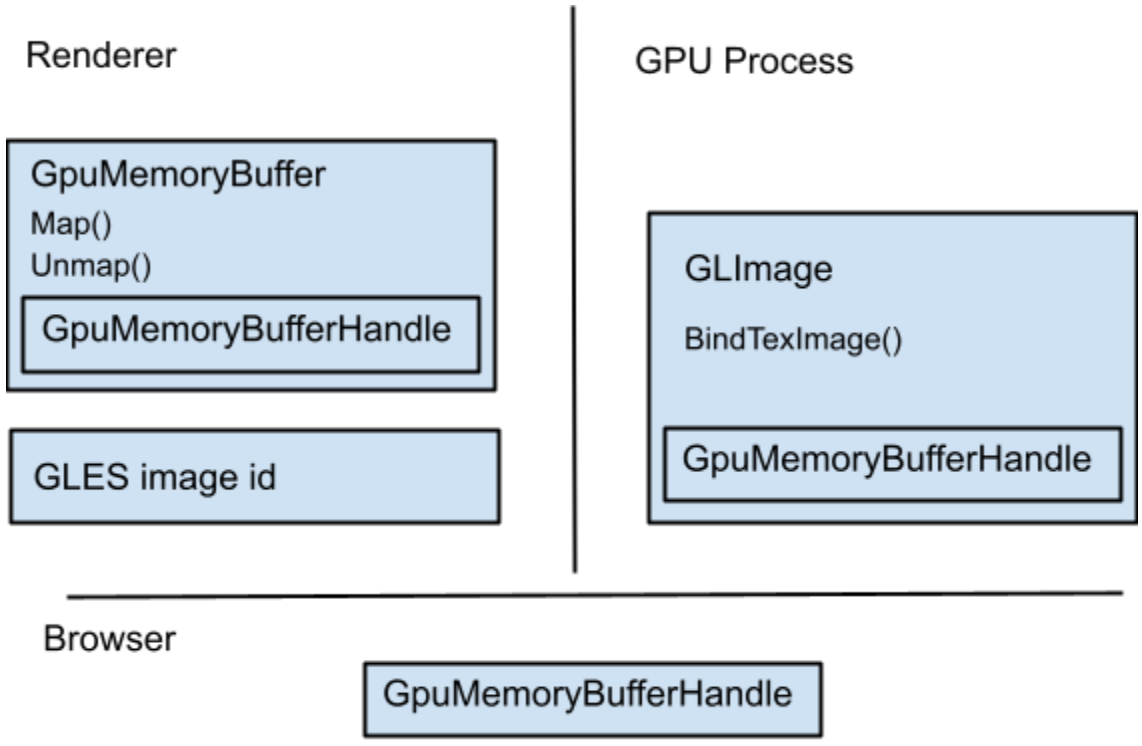
CHROMIUM_image_gpu_memory_buffer extension requires a GLES2Interface to allocate, map and unmap buffers, which makes it complicated to perform these operations on a worker thread. The CHROMIUM_image extension solves this by allowing the renderer to create an image from an existing GpuMemoryBuffer instance.

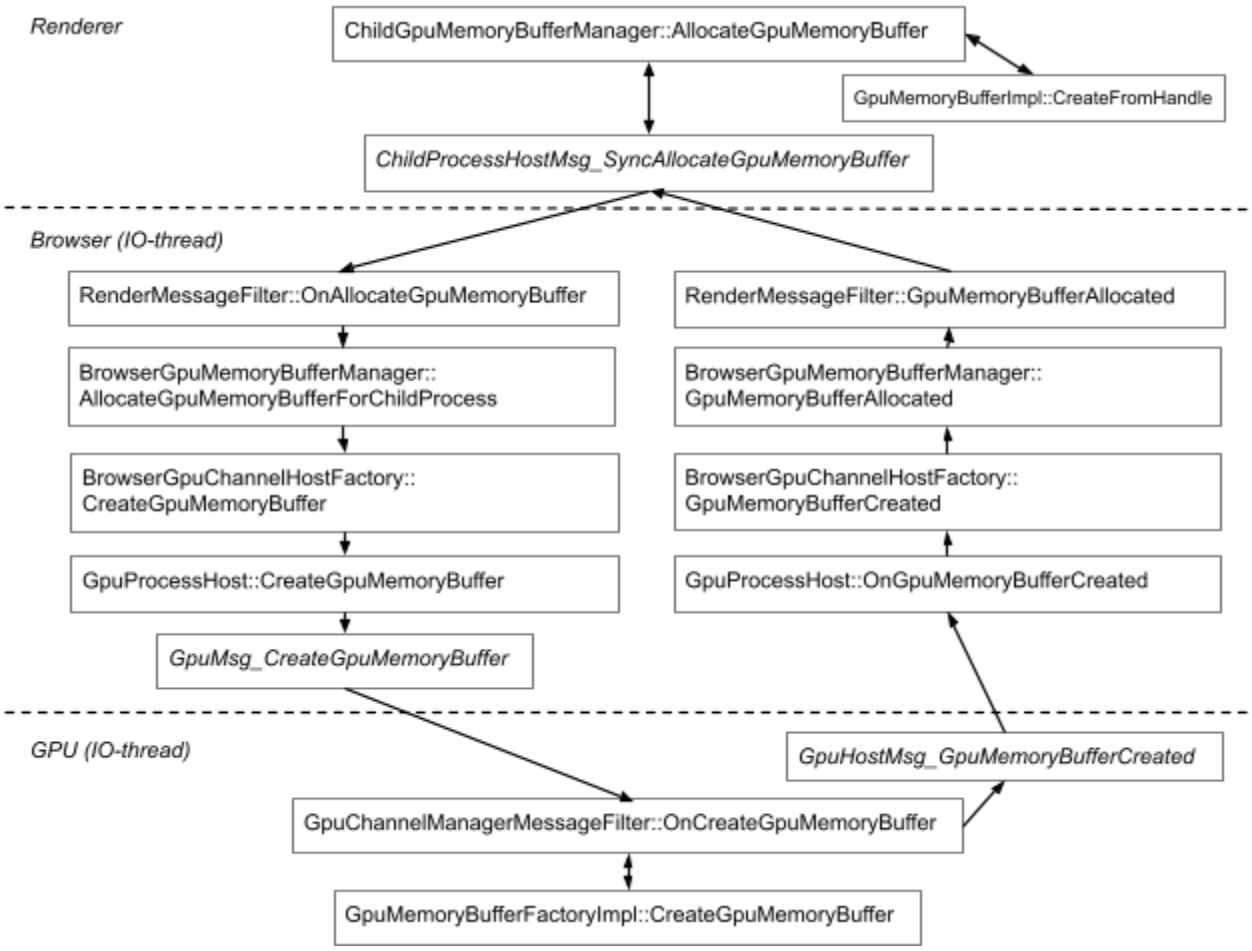
```
// CHROMIUM_image
typedef struct _ClientBuffer* ClientBuffer;

GLuint createImageCHROMIUM(ClientBuffer buffer,
                           GLsizei width,
                           GLsizei height,
                           GLenum internalformat);
void destroyImageCHROMIUM(GLuint image_id);
```

Note that <buffer> is the name (or handle) of a resource to be used as the image source, cast into the type ClientBuffer. Currently, this must always be a pointer to a gfx::GpuMemoryBuffer instance. The renderer can create and use an GpuMemoryBuffer instance on a worker thread where access to the GLESInterface is not available. createImageCHROMIUM can later be used to create an image from this buffer and use it as a texture.

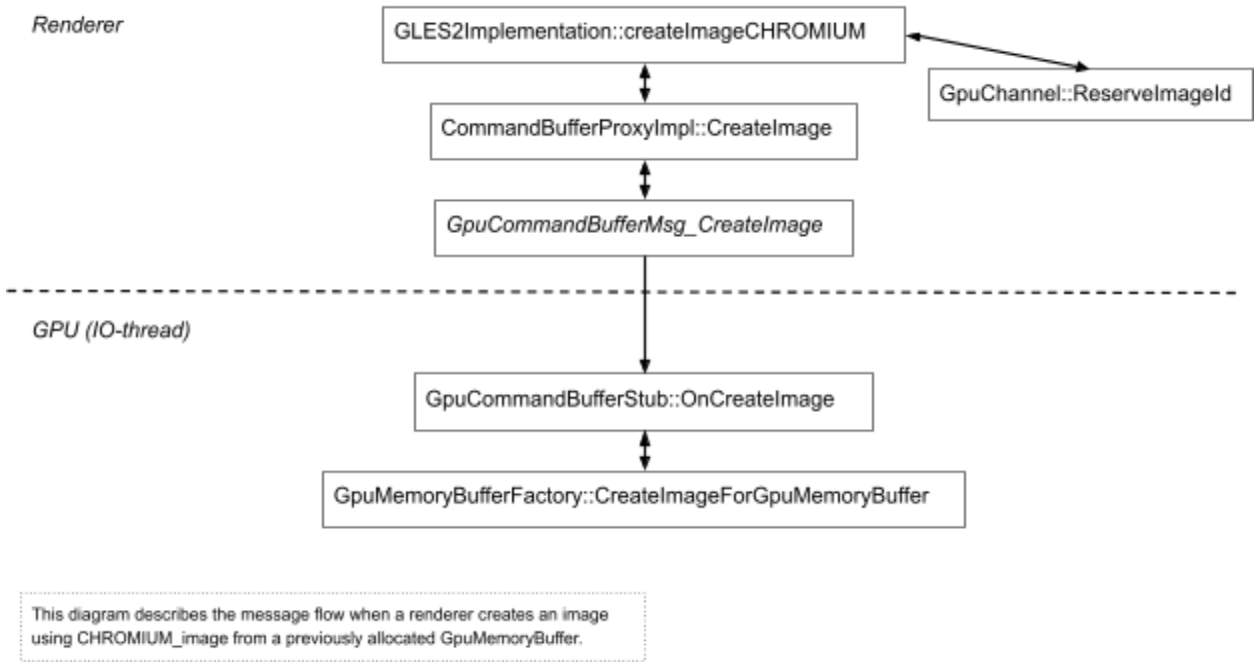
|internalformat| must be GL_RGBA or GL_RGB. createImageCHROMIUM can fail or an additional cost might incur if |internalformat| is not compatible with <buffer>.





The diagram above describes the flow when a renderer allocates a native GpuMemoryBuffer.

Note that a regular shared memory backed GpuMemoryBuffer will be allocated instead of a native buffer when native support is missing. Shared memory buffers are created in the browser process rather than the GPU process. BrowserGpuMemoryBufferManager::AllocateGpuMemoryBufferForChildProcess will allocate and return the buffer synchronously in that case.



CHROMIUM_image doesn't allow a client to create a GpuMemoryBuffer backed image without access to chromium specific GpuMemoryBuffer allocation APIs. CHROMIUM_gpu_memory_buffer_image provides a mechanism to create images without access to these APIs.

```
// CHROMIUM_gpu_memory_buffer_image
GLuint createGpuMemoryBufferImageCHROMIUM(GLsizei width,
                                          GLsizei height,
                                          GLenum internalformat,
                                          GLenum usage);
```

|internalformat| must be GL_RGBA. The actual pixel format for the underlying GpuMemoryBuffer is implementation dependent.

Security

A GpuMemoryBuffer is just a special kind of shared memory and for it to securely be used in chromium's multi-process architecture it needs the same treatment as regular shared memory. That is; the browser process control allocation of buffers and access to a buffer is properly isolated to a specific renderer process.

The risk associated with exposing this type of memory to the renderer will need to be evaluated on a per-platform/driver basis. Ideally it adds no more more risk than the current usage of shared memory. The GpuMemoryBuffer framework described here has been designed to provide the same level of security as standard shared memory when supported by platform and

driver.

Software Fallback

A standard shared memory implementation can be used in cases where native driver support is unavailable. A texture upload is implicitly taking place when a GpuMemoryBuffer backed by standard shared memory is bound to a texture. Each GpuMemoryBuffer/Texture pair will use twice the memory compared a native implementation where the GpuMemoryBuffer storage is directly used as a texture. This can still provide an efficient method for transferring large amounts of texture data to the graphics hardware but the increased memory usage should be taken into account when choosing the mechanism used to perform these transfers.