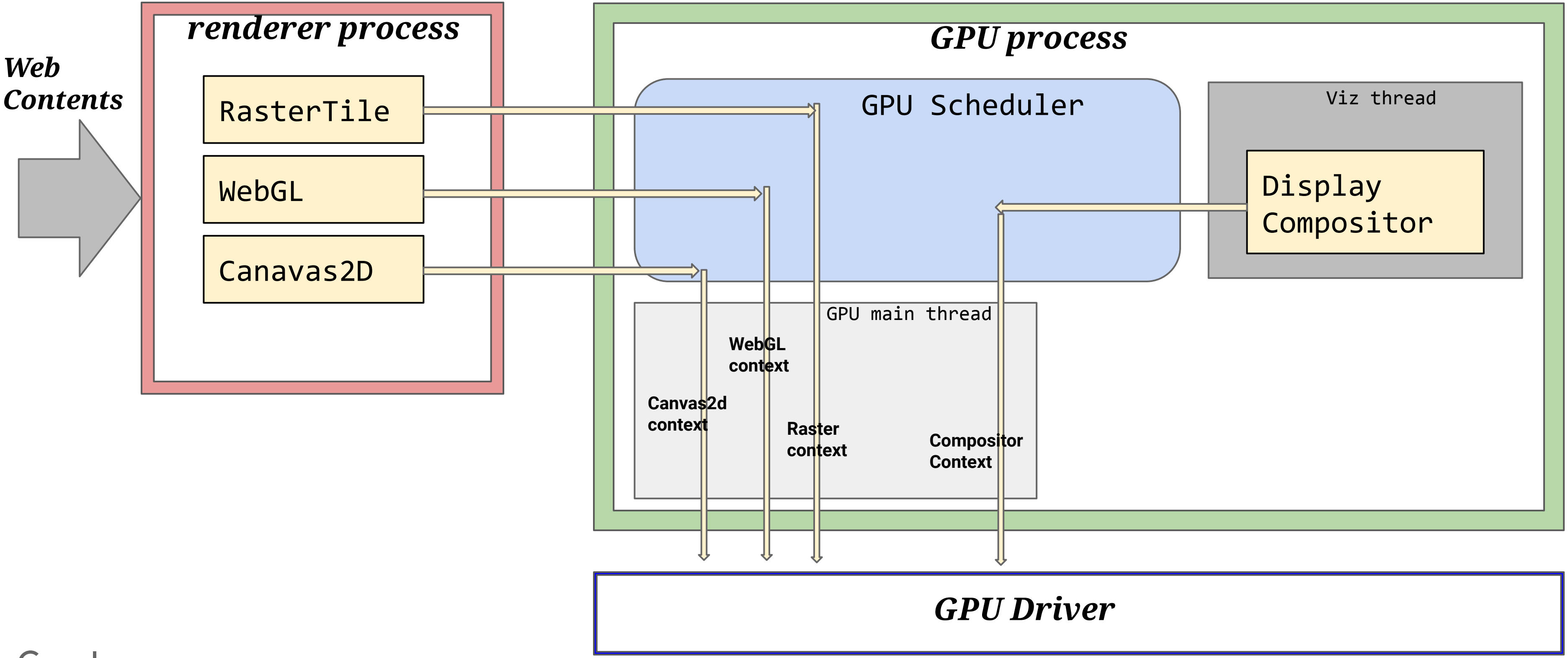
Direct **R**endering **D**isplay **C**ompositor

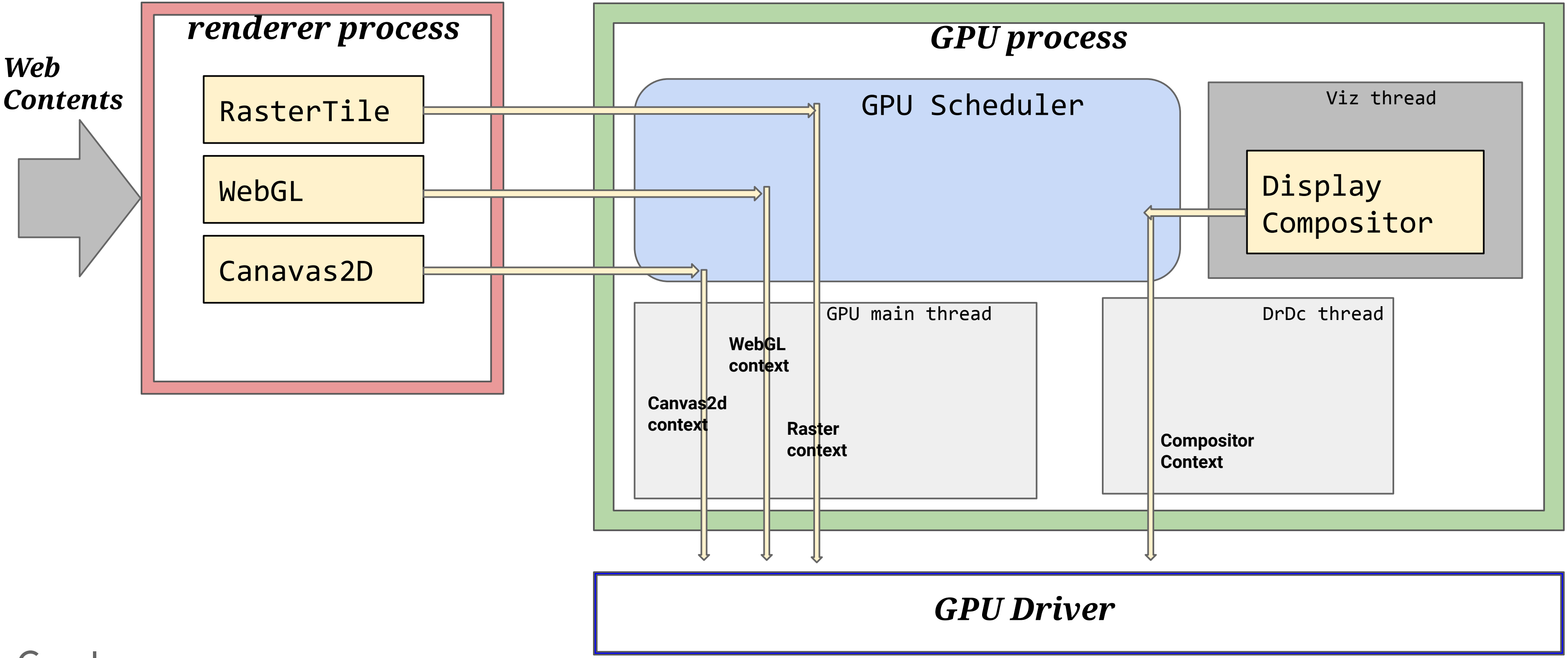
vikassoni@

Recap..



Google

What is DrDc..



Google

Why DrDc..

A. Less context switch overhead.

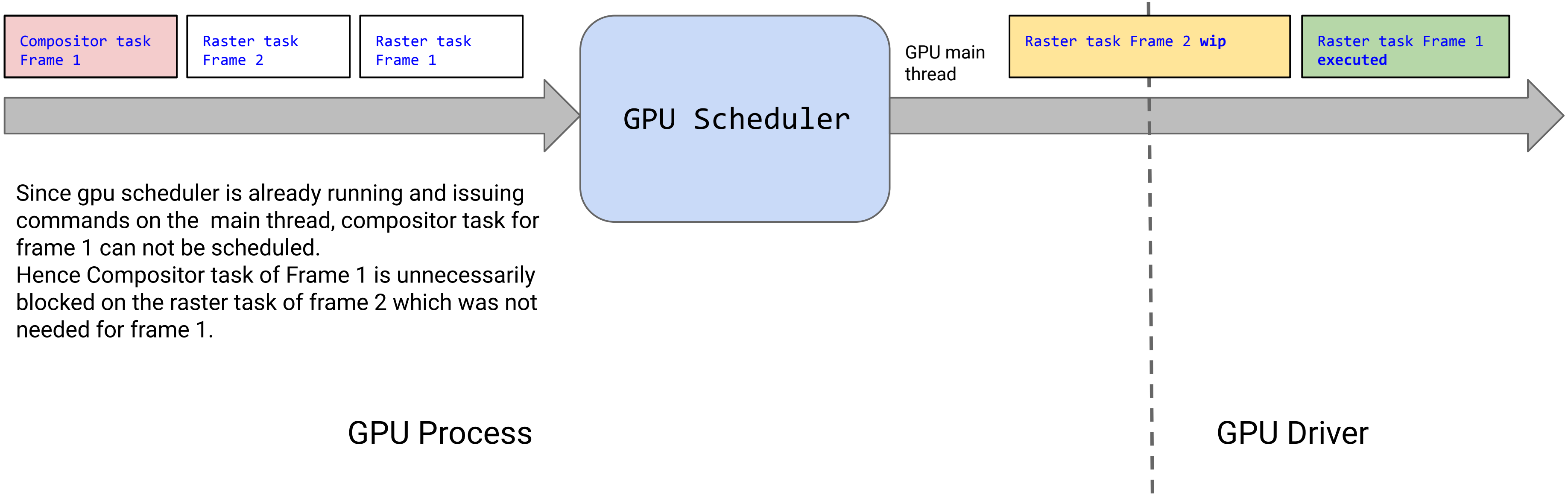
   a. For the most common case of raster tile, raster tasks will use raster context to generate a texture and display compositor will use compositor context to display that texture on screen. This causes context switches since only one context can be current on a thread at any given time.

   b. Switching display compositor to use a dedicated gpu thread results in eliminating this context switches.

Why DrDc..

B. More efficient scheduling.

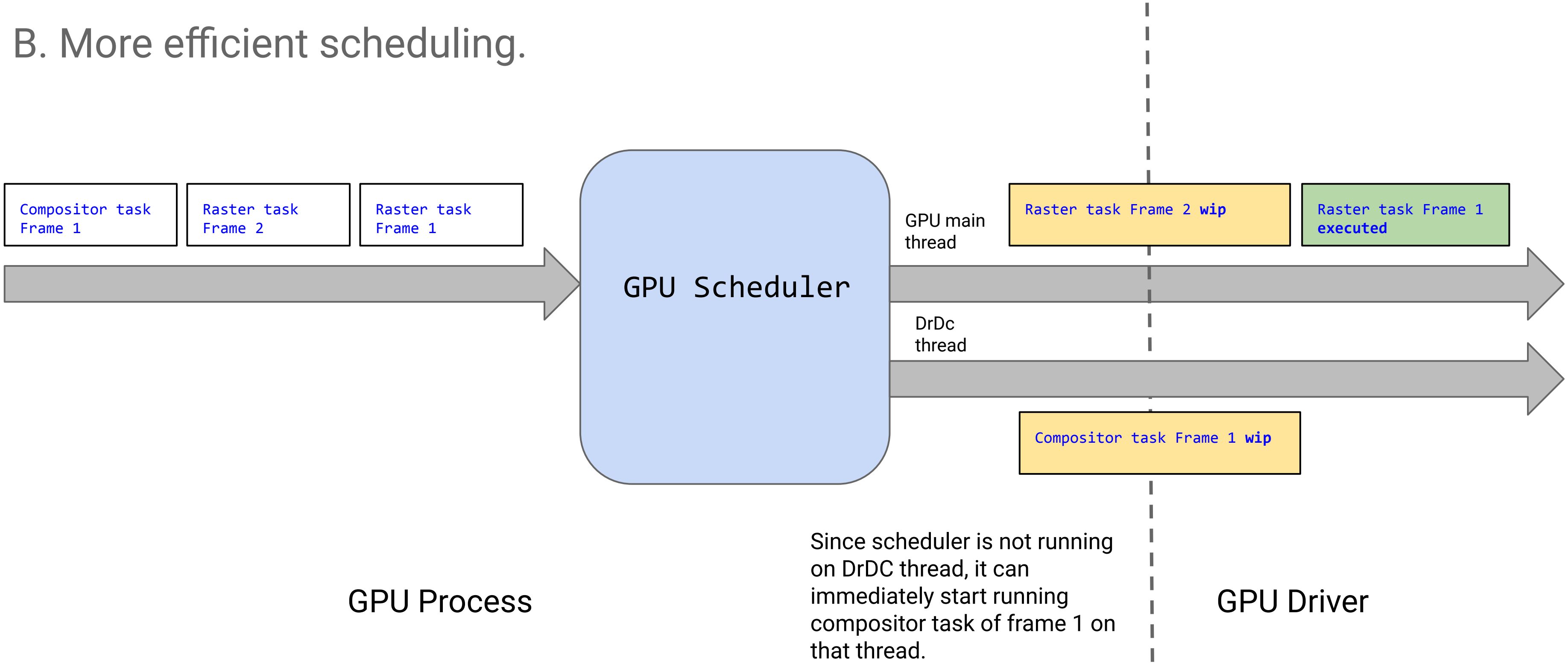
Scheduler can preempt the raster task of Frame 2 after about 20 commands but with OOPR this problem becomes worse since we can only schedule a task for whole tile and it can not be broken into smaller tasks.



Since gpu scheduler is already running and issuing commands on the main thread, compositor task for frame 1 can not be scheduled.
Hence Compositor task of Frame 1 is unnecessarily blocked on the raster task of frame 2 which was not needed for frame 1.

Why DrDc..

B. More efficient scheduling.



Compositor task Frame 1

Raster task Frame 2

Raster task Frame 1

GPU Scheduler

GPU main thread

Raster task Frame 2 **wip**

Raster task Frame 1 **executed**

DrDc thread

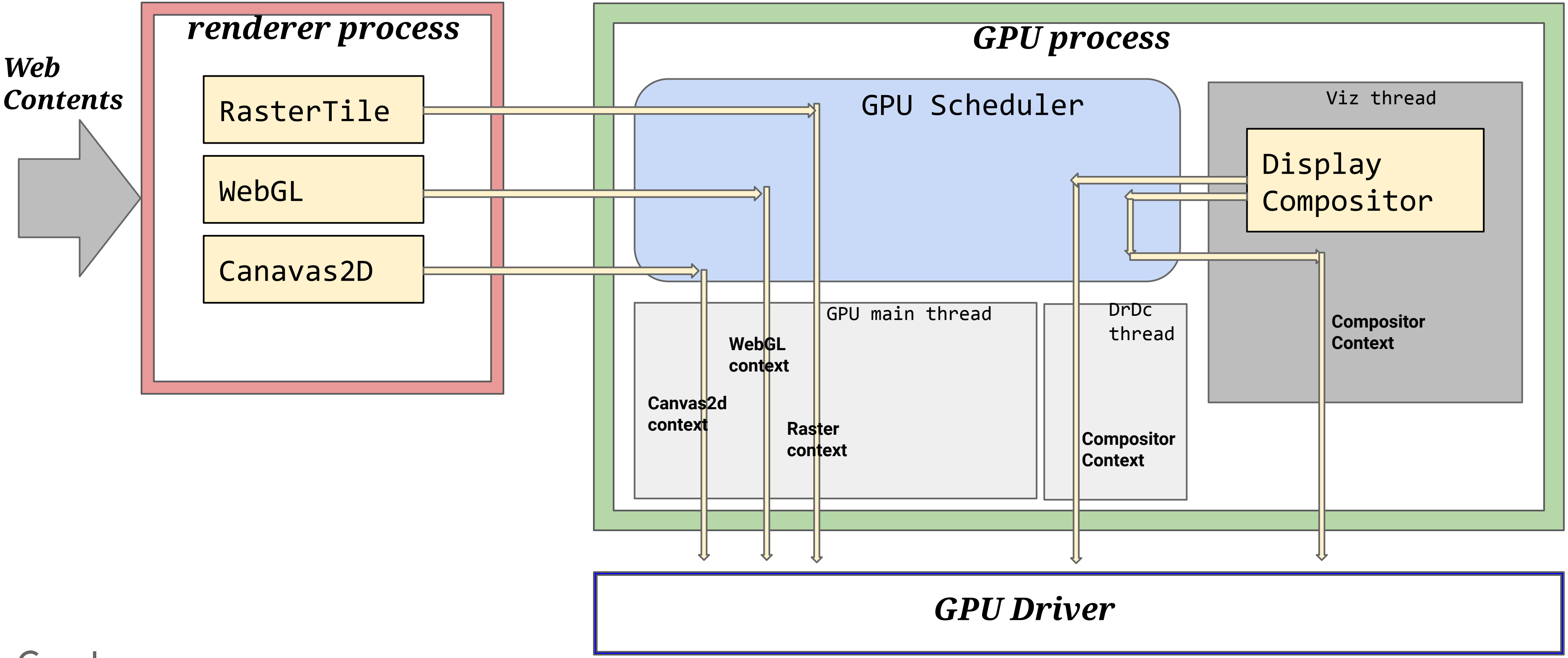
Compositor task Frame 1 **wip**

Since scheduler is not running on DrDC thread, it can immediately start running compositor task of frame 1 on that thread.

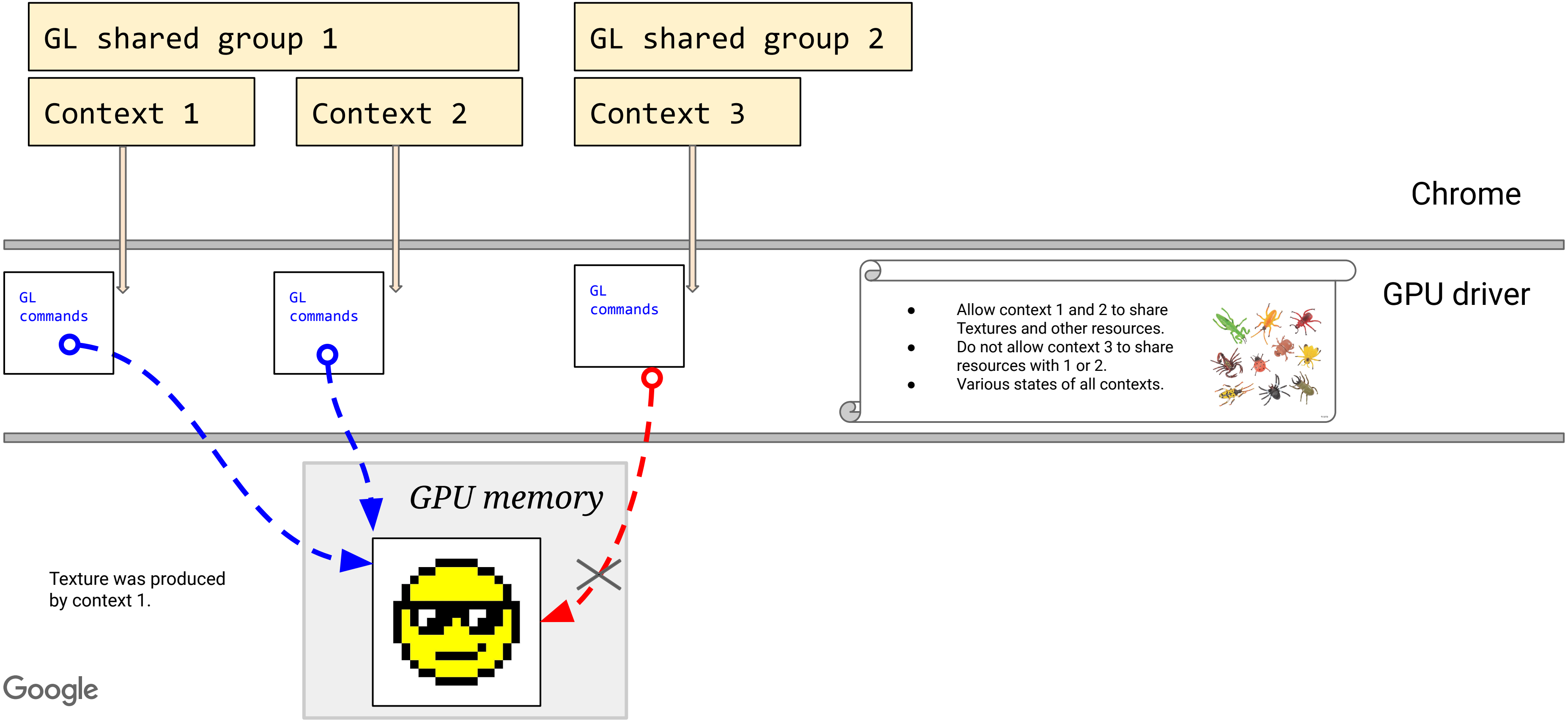
GPU Process

GPU Driver

Threading model…



GL context model… Resource sharing across contexts



GL shared group 1

GL shared group 2

Context 1

Context 2

Context 3

Chrome

GPU driver

GL commands

GL commands

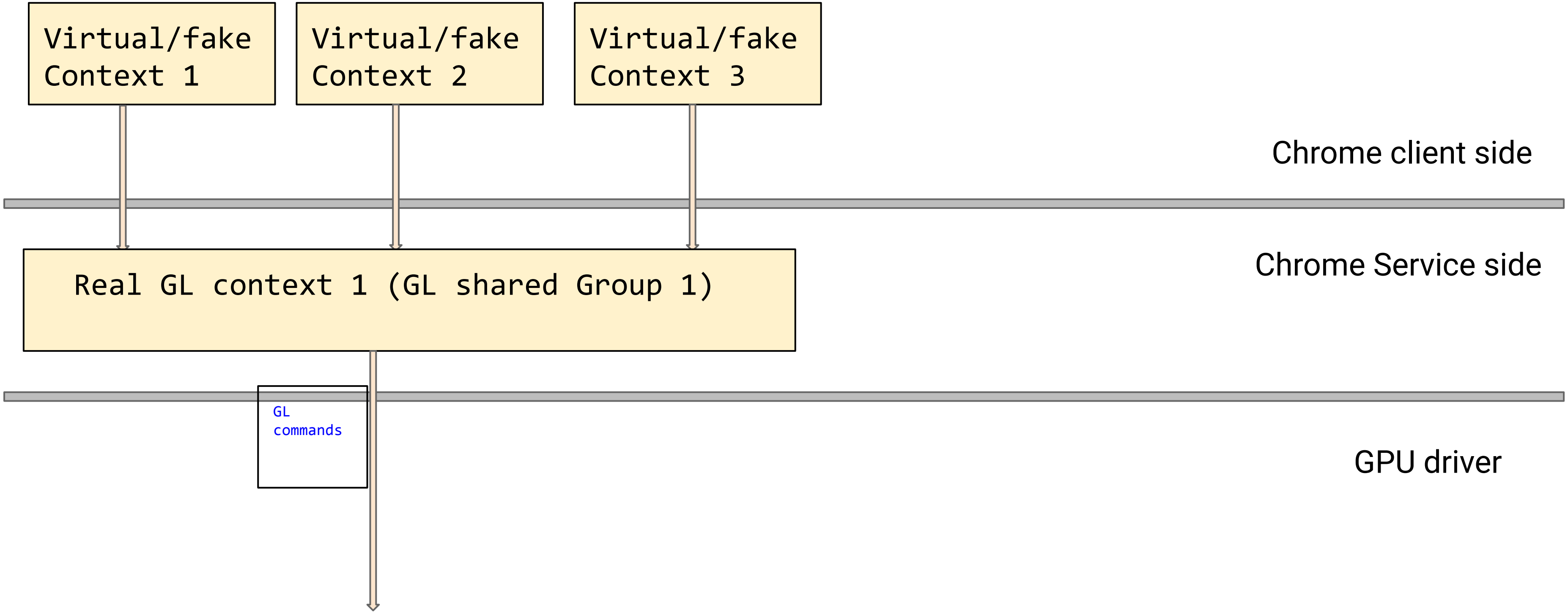
GL commands

- Allow context 1 and 2 to share Textures and other resources.
- Do not allow context 3 to share resources with 1 or 2.
- Various states of all contexts.

GPU memory

Texture was produced by context 1.

GL context model… Virtual contexts



Virtual/fake Context 1

Virtual/fake Context 2

Virtual/fake Context 3

Chrome client side

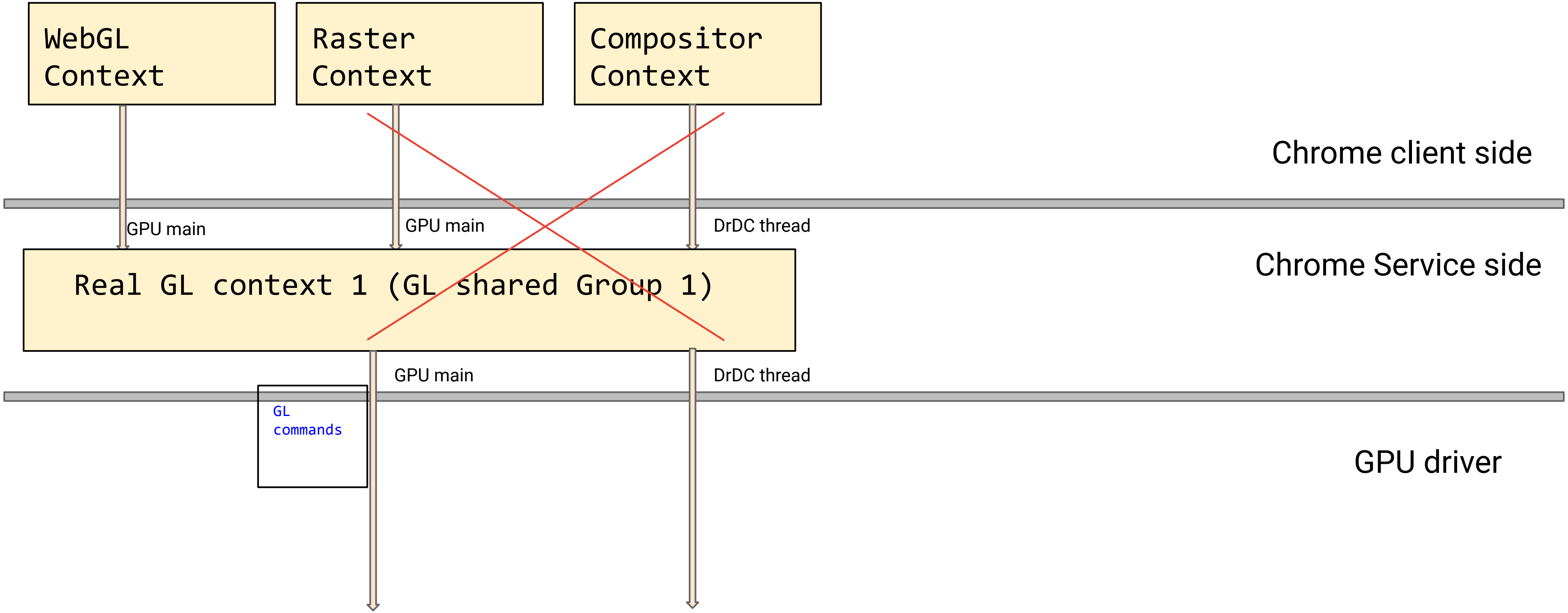
Real GL context 1 (GL shared Group 1)

Chrome Service side

GL commands

GPU driver

GL context model… Virtual contexts



WebGL Context

Raster Context

Compositor Context

Chrome client side

GPU main

GPU main

DrDC thread

Real GL context 1 (GL shared Group 1)

Chrome Service side

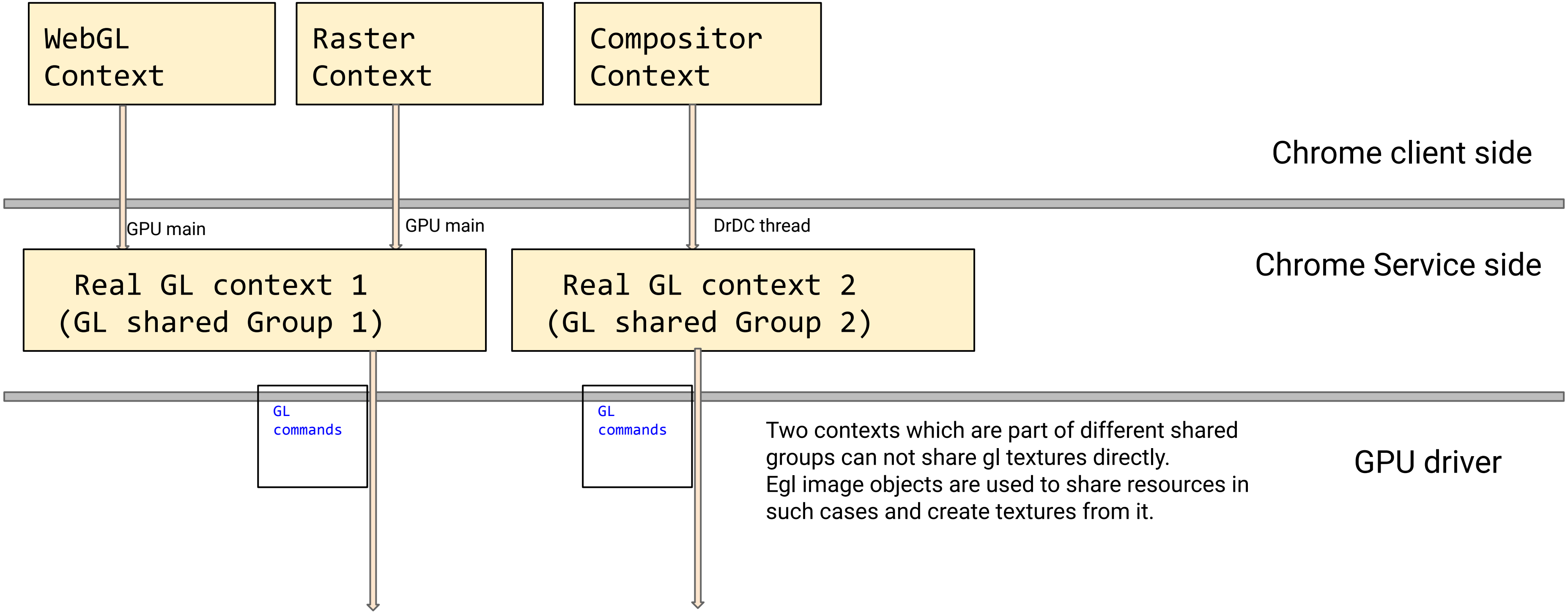
GPU main

DrDC thread

GL commands

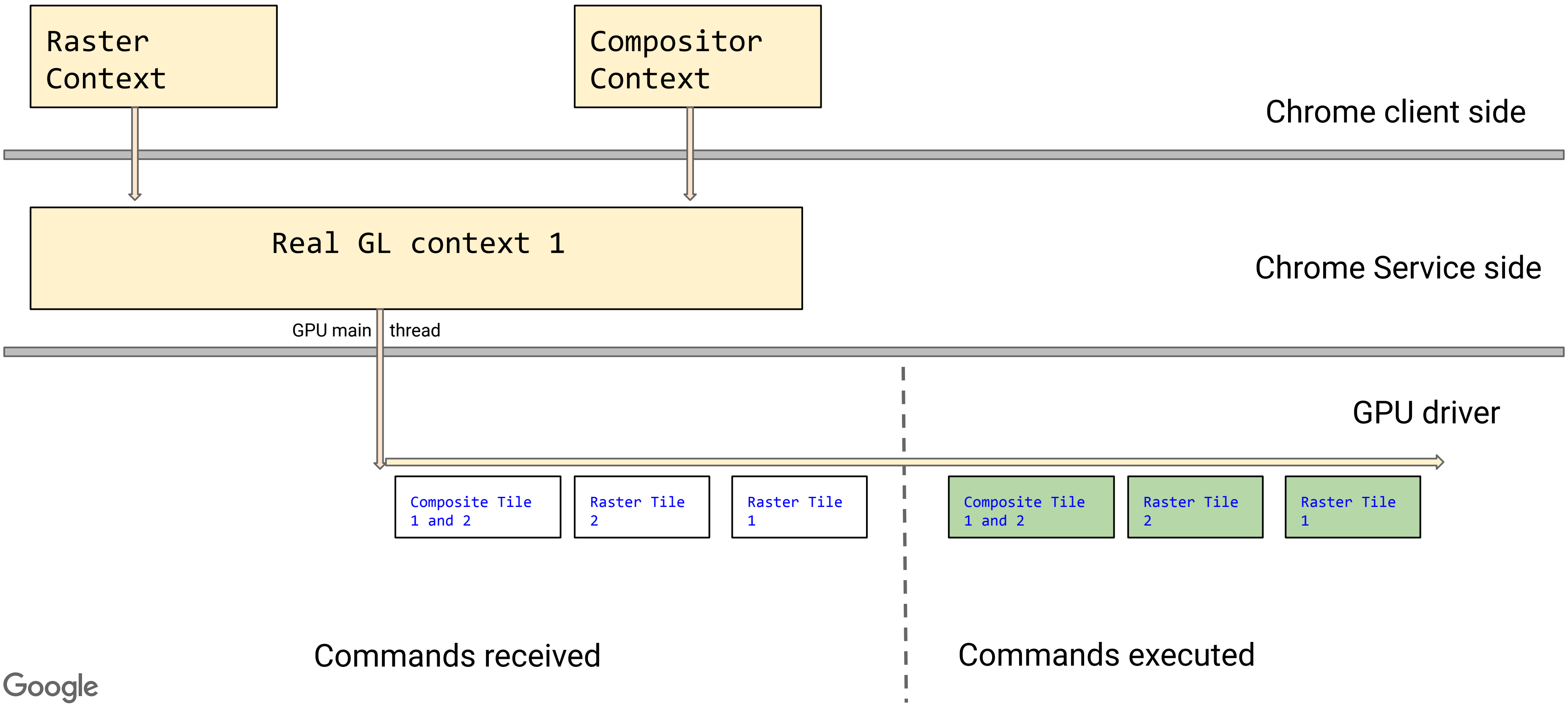
GPU driver

GL context model… DrDc context



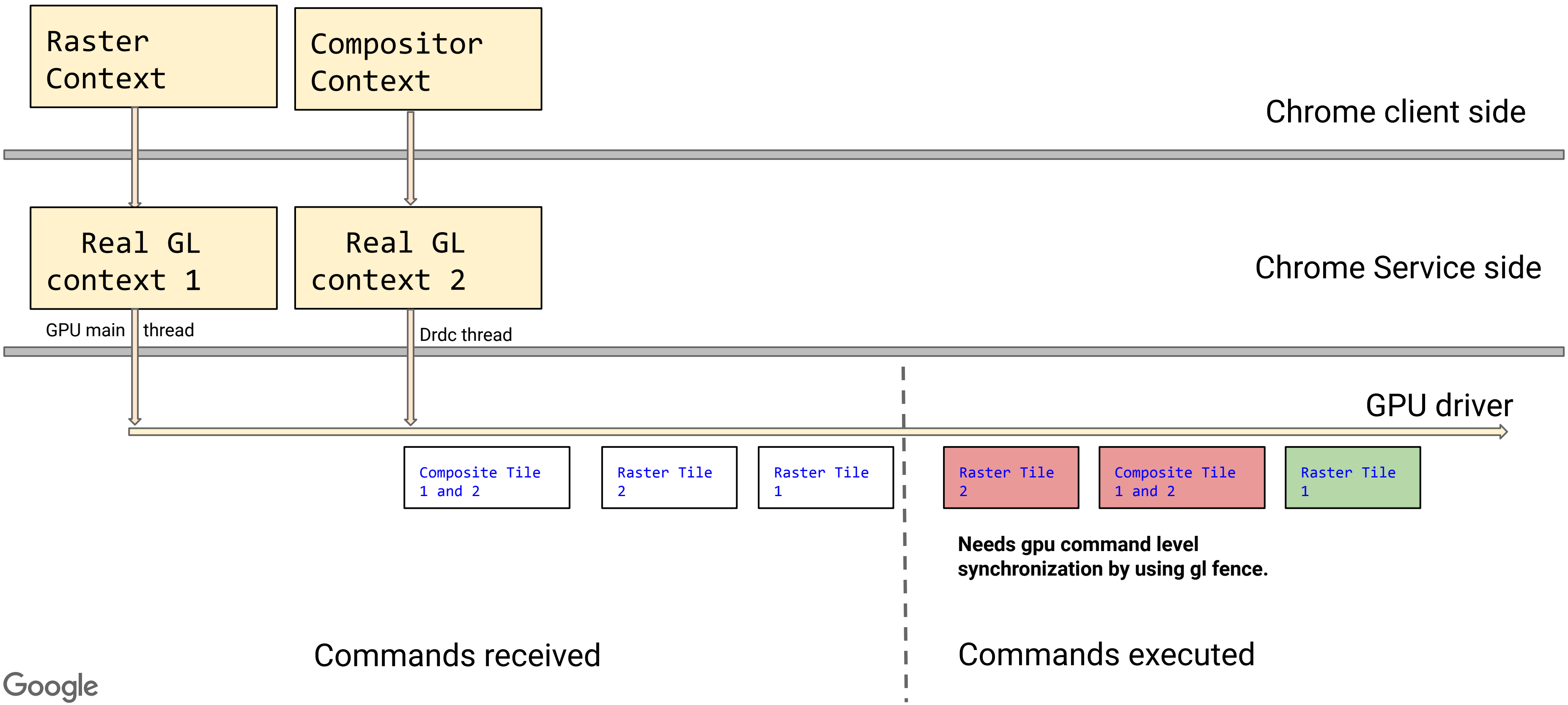
Two contexts which are part of different shared groups can not share gl textures directly.
Egl image objects are used to share resources in such cases and create textures from it.

Synchronizing resources..



Google

Synchronizing resources..



Raster Context

Compositor Context

Chrome client side

Real GL context 1

Real GL context 2

Chrome Service side

GPU main thread

Drdc thread

GPU driver

Composite Tile 1 and 2

Raster Tile 2

Raster Tile 1

Raster Tile 2

Composite Tile 1 and 2

Raster Tile 1

Needs gpu command level synchronization by using gl fence.

Commands received

Commands executed

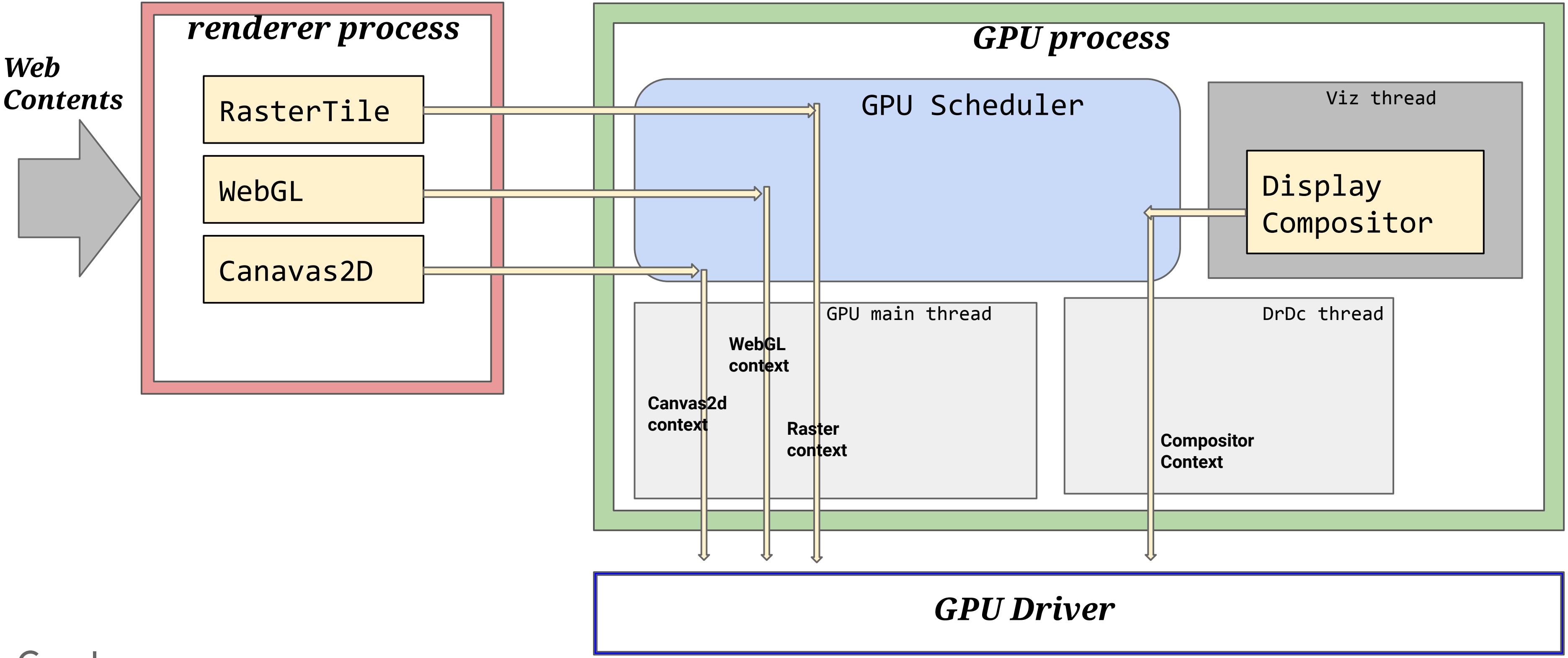
SharedImages changes…

- In chrome, textures are produced and consumed by different apis or subsystems. For eg:, when we raster a tile, it could be rasterized in opengl and composited in vulkan.
- We need to handle this requirement by allocating buffers for these textures in such a way that they are compatible with both producer and consumer.
- To handle this requirement, we have the concept of shared images. It allows producer of shared image to specify the size and usage requirements via shared image apis and a compatible shared image backing is chosen based on that requirement.

SharedImages changes…

- For DrDc, the underlying resource or buffer needs to be egl image through which we create and share textures across different contexts part of different shared group. We also need to synchronize read/write access from different thread and contexts.
- SharedImageBackingEGLImage is implemented and used for this purpose. It is thread safe, uses egl objects and has the added gpu fence level synchronization to ensure correctness.

GPU Scheduler changes…



Google

Vulkan Changes

- No concept of GL context.
- Multiple options to submit commands from different thread :
  - Use a single vKQueue to submit commands from both gpu threads and synchronize all access to the queue.
  - Use 2 different queues from same vk devices, but not all gpus supports more than 1 queue.
  - Use 2 different vk device in each thread but needs more explicit synchronization or interops using semaphores.
- WIP.

Current Status

- Enabled 50% canary/dev on android P.
- Android Q+ work is in progress which needs vulkan changes.

Benchmark Results...

- Prototype showed ~7% improvements in graphics smoothness metrics (touchscroll) for Motionmark and tough scrolling on Pixel 2.
- Average fps got 7% better on MotionMark.
- Motionmark results
- “tough_Scrolling” story tag results.
- Analyzing benchmarks on production code is wip.

Future Work

- Add all optimizations for android.
- Enable it on all other platforms.

Q and A

1. Can we start scheduling some commands to composite a frame before waiting for the raster to be done completely. Probably use some finer grain synchronization than sync tokens.
2. Vulkans is implemented and working on webview. So android can take free implementations from it. Also it would be better to have same/similar implementation in both webview and dr-dc for simplicity of maintenance which is basically continue using new dr-dc thread instead of the viz thread.
3. From skiarenderer point of view, using same viz thread as dr-dc could results in better performance since now commands does not need to be send through commands buffer. Performance is greater priority than simplicity of code.
4. In some platforms overlay processing could be better with using viz as dr-dc thread.
5. How does dr-dc works with passthrogh decoder on android. And how it works on window today with angle being the backend.