Asynchronous GPU Rasterization

vmiura@, sunnyps@, dyen@, piman@

Tracking issue: 525259

Motivation

GPU Rasterization (using Skia Ganesh) is Chromium's GPU accelerated rasterizer. Utilizing the GPU has promise for faster rasterization, support for dynamic app-like Web content on high DPI displays, and lower power consumption.

While GPU Rasterization can be significantly faster, current lack of scheduling on the GPU Service and GPU devices make the system more prone to jank than pure CPU Software Rasterization. This design aims to comprehensively solve scheduling of work on the GPU Service.

Background

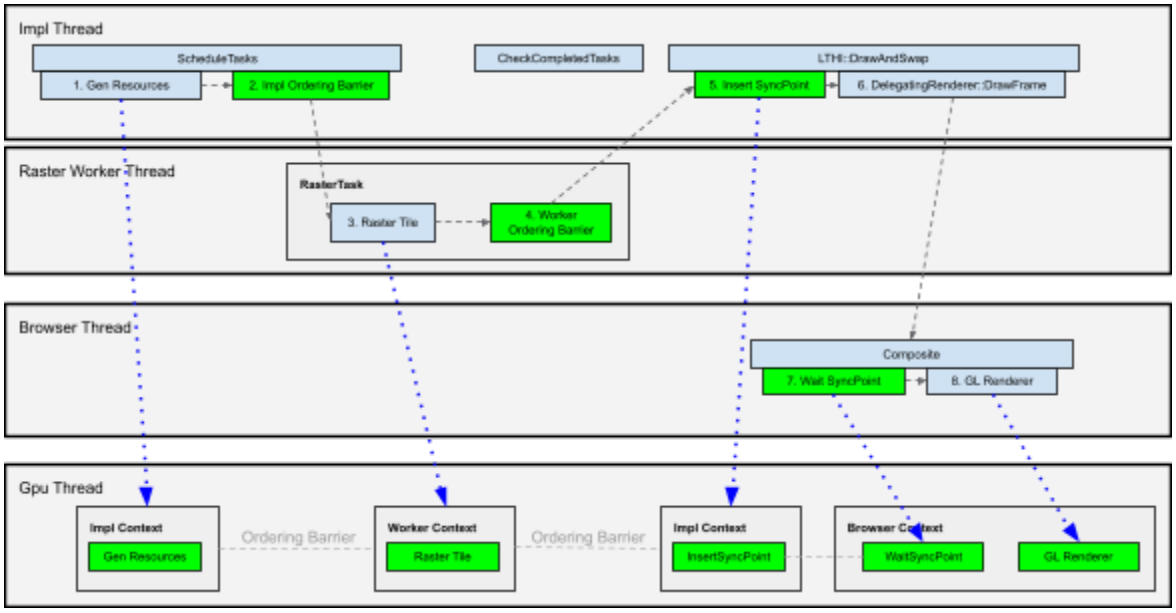
There are many cases on the Web today where content is more complex than can be fully rendered in real-time within the capabilities of the host device. Content can have many layers of complexity, modern displays can reach to 5k resolutions, and there is a great deal of variation in CPU & GPU performance among devices running Chromium. For a great browsing experience, UI and input must remain smooth and responsive even when rendering the content itself is not smooth or responsive.

Chromium's Multithreaded Rasterization system decouples content updates from compositing, and utilizes CPU worker threads to make rasterization asynchronous. Within Multithreaded Rasterization, Chromium has two raster paths, Software and GPU:

- Software Raster: Simple OS thread scheduling enables us to perform rasterization in parallel with compositing. Rasterization can scale from using part of one CPU core, to using many cores if available.
- GPU Raster: Was updated in M-42 to use worker threads for rasterization Threaded Gpu Rasterization. This allows the GPU Raster frontend to run in parallel with compositing. GPU Raster however, also creates a lot of new work for the GPU Service.

  The GPU Service is currently single-threaded, and has no ability to schedule and control priority between raster and compositing. In fact due to the existing execution ordering rules, the GPU Service doesn't have room to make useful scheduling decisions without breaking correctness.
  This design covers changes to Threaded GPU Rasterization to enable the GPU Scheduler to make scheduling decisions.

Current GPU Raster Threading & Synchronization



OrderingBarrier API ensures context order on a shared GPU channel in the Renderer Process. SyncPoint API enables synchronization across GPU channels in different processes.

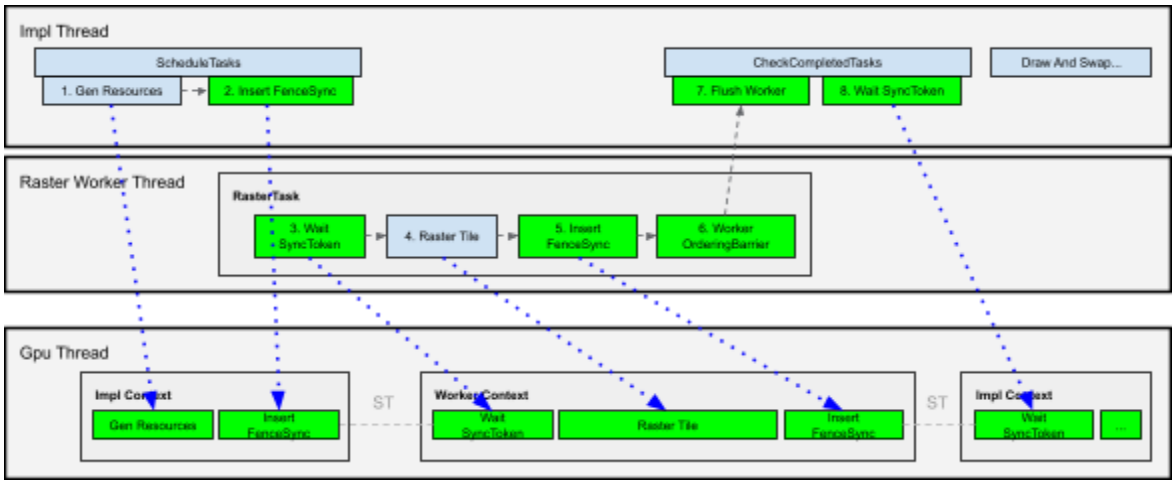
1. **Gen Resources**: Generates GL texture handles for tiles.
2. **Impl OrderingBarrier:** Sets up GPU Thread side ordering between steps **1** and **3**, ensuring resources are generated before rasterizing the tile.
3. **Raster Tile:** Runs Skia GPU rasterization. Generates GL commands on Worker Context.
4. **Worker OrderingBarrier:** Sets up GPU Thread side ordering between steps **3** and **5**, ensuring dependent tile rasterization is done before the following Insert SyncPoint on Impl Context.
5. **Insert SyncPoint:** After gathering all completed RasterTasks in CheckCompletedTasks, tiles available for drawing are known. Inserts SyncPoint on Imp Context to set up synchronization with out-of-process Browser Context.
6. **DelegatingRenderer DrawFrame:** IPCs frame data to Browser process: {Compositing Quads, Resources, SyncPoints}.
7. **Wait SyncPoint:** Sets up GPU Thread side ordering between steps **5** and **8**, ensuring dependent tiles are rasterized before use by GL Renderer.
8. **GL Renderer:** Finally composites frame to output surface.

Enabling Asynchronous GPU Rasterization

Contexts on a shared GPU Channel have an implicit execution order: Contexts on a Channel must strictly execute in FIFO order based on order of calls to Flush or OrderingBarrier APIs.

The following steps enable asynchronous execution of the Raster Worker and Impl Contexts:

- A new GPU Channel Stream concept enables independent execution of contexts on a channel. Worker and Impl Contexts move to separate GPU Streams.
- With the Worker Context in a separate Stream, the new GPU Scheduler can pre-empt GPU Raster execution and give priority to other clients.
- Textures are shared between Worker and Impl Streams with Mailboxes.
- SyncPoint APIs are used for synchronization across Streams when required. There's a new Lightweight SyncPoint design (named FenceSync/SyncToken) to make this practical and fast to use on a per-Tile basis.



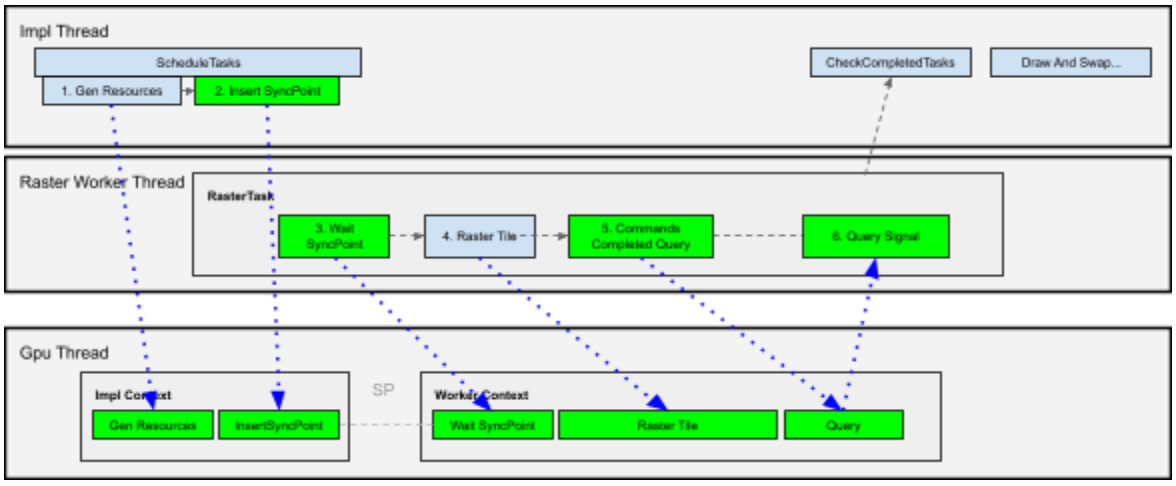
1. **Gen Resources**: Generates GL texture handles and Mailboxes for Tiles.
2. **Insert FenceSync:** Sets up FenceSync/SyncToken for synching **1** and **3**, ensuring resources are generated before rasterizing the tile.
3. **Wait SyncToken:** Waits for step **2** on GPU Service.
4. **Raster Tile:** Runs Skia GPU rasterization. Generates GL commands on Worker Context.
5. **Insert FenceSync:** Sets up FenceSync/SyncToken for synching **4** and **8**, ensuring raster completes before DrawAndSwap.
6. **Worker OrderingBarrier:** Sets up to be flushed by **7**.
7. **Flush Worker:** Flushes Worker context up to **6**.
8. **Wait SyncToken:** Waits for step **5** on the GPU Service.

Asynchronous “nice” GPU RasterTasks

In SMOOTHNESS_TAKES_PRIORITY compositing mode, GPU Raster work should play nice and give priority to the Compositor.

Nice GPU RasterTasks don’t throw work over to the GPU Service and immediately send blocking SyncPoints to the Browser. Instead, work is thrown to the GPU Service to be scheduled at a low priority by the GPU Scheduler. The RasterTask waits for asynchronous completion with GL Query GL_COMMANDS_COMPLETED_CHROMIUM before marking the rastered Tile as done.

At this point, additional SyncPoint synchronization is optional, since the RasterTasks’ commands have completed.



1. **Gen Resources**: Generates GL texture handles and Mailboxes for Tiles.
2. **Insert SyncPoint:** Sets up SyncPoint for synching **1** and **3**, ensuring resources are generated before rasterizing the tile.
3. **Wait SyncPoint:** Waits for step **2** on GPU Service.
4. **Raster Tile:** Runs Skia GPU rasterization. Generates GL commands on Worker Context.
5. **Commands Completed Query:** Set up query for GPU Service command completion.
6. **Query Signal:** RasterTask completes when commands completed.