GPU Rasterization Triggering

nduca, ernstm, alokp, ajuma, vangelis

Summary

We want to make gpu rasterization available to mobile content as soon as possible. To do this, we will use an page-wide trigger based on the presence of:

```
<meta name="viewport"
 content="width=device-width, minimum-scale=1.0,
 initial-scale=1.0, user-scalable=yes">
```

to use, and has much better future-compatibility. It has a simpler implementation in cc. However, it does mean we have to support a much larger set of pages at launch. Nonetheless, recent data plus veto plans give us reasonably high confidence in our chances of success.

This alone will not force content into GPU rasterization: a whitelist will limit GPU rasterization to a limited set of Android devices, and we will also apply a content-based vetoing mechanism to eliminate tricky corner cases to keep focused on high impact content. Post launch, more GPUs and content will be made to trigger Ganesh. Importantly, no corrective action or regression risk will be faced by early adopters that use this initial viewport tag.

Compared to hybrid plans, this approach is easier for content authors

Getting involved

Launch Plan

There are three big parts to our short term triggering strategy for GPU rasterization:

- GPU Blacklist Check: *currently* only allow GPU rasterization on a limited set of Android devices. Other devices and OS' get software rasterization. See content authoring section below for more information.

- Meta tag: authors put
  ```
  <meta name="viewport"
   content="width=device-width, minimum-scale=1.0,
   initial-scale=1.0, user-scalable=yes">
  ```
  onto their page, in any order. All items must be present. This is the primary trigger for GPU rasterization. This is also the trigger for many other Chrome fast paths: double tap delay a, pinch zoom control, so we wish to attract people toward this flag, anyway.

- Per-SkPicture veto: during recording, Skia looks for problematic ops or sequences of ops that would be *currently* bad for performance on the GPU. If any picture gets vetoed by Skia, then GPU rasterization for that LayerTreeHostImpl is disabled for that instance's remaining lifetime.

During our initial launch, our primary mechanism to mitigate negative impact on content/stability is adding veto steps to Skia for problematic content. We can also narrow the GPU blacklist, though it is already quite narrow.

- Tooling to help triage slow pages and quickly identify what content is problematic would be very helpful.

Content Authoring with GPU Raster in mind

GPU rasterization is invisible to the platform, intentionally. This means that making content that only works with GPU raster isn't going to be directly possible. There are two ways to approach the new GPU rasterization abilities in chrome, as a content author:

1. UA-detect Chrome Android and add the meta tag. If GPU raster is available, then your content will go faster or as-fast-as-before. You will have to accept the risk that content may run radically slower on lower-end Android devices, at least for now while we have GPU raster deployed on a limited set of devices.

2. Sit tight and do nothing. If you need this meta tag for other reasons, then GPU raster will either be faster, or file crbug.com/new to let us know and we'll try to fix the issue. Eventually, GPU raster will be available on all devices.

The Role of `will-change`

will-change's primary application is to control layer creation and prepainting. The following improvements to layer managment are LGTM to ship in M36:

- `will-change: top/left` are a stable replacement for -webkit-backface-visibility: hidden, which is used to create a layer without forcing a stacking context and change of initial containing block. This is useful for having divs that have child popup menus and position:sticky emulation via rAF.

- `will-change: transform` is a stable fix for content that tries to force layer creation with transform: translateZ(0) but then moves content around with transform: translateX(). This anti-pattern causes the layer to go away while moving and is a common mistake.

- `will-change: opacity` forces a stacking context on a div. This allows content to animate opacity to or away from one without having to cause a re-layout due to becoming or unbecoming a stacking context.

Additionally, at some point in the future:

- `will-change: contents` will be a mechanism to disable prepainting and decide whether to use direct rasterization. If you have an element that is offscreen, it is currently ambiguous to the browser whether that div is going to change next frame, or whether it is likely to be the same for a while to come. If it is unlikely to change, then it is often better for performance to cache it in a GPU texture, and prepaint it if there are available system resources, for example when the user is paused between scrolls of the screen. But, if the content is changing, then it is better to only temporarily render the content using direct rasterization or to a temporary texture, and only when it is visible.

Post-Launch Comments

There will always be some horribly broken GPUs that require fallback to software rasterization. We expect in the asymptote, these same GPUs will be using software compositing as well. For the remainder of "baseline capable GPUs," we expect Ganesh to mature to the point that it is internally handling all web content. If a software fallback needed, we expect Ganesh to handle this internally.

Post-launch, this implies a few parallel goals that we need to balance:

1. Keep making GPU rasterization faster for the initial launch content. Mobile first focus will endlessly demand better baseline performance.

2. Eliminate veto paths and make GPU rasterization passably fast for previously-excluded content.
3. Relax the blacklist so more OS' and GPUs trigger GPU rasterization.

As this unfolds, the minimal viewport meta tag may be less critical for getting GPU rasterization. We can relax the viewport criteria over time to include more pages, e.g. by dropping the scale requirement, or even go to the point that we don't consider it at all. But, will remain a good signal for things like double-tap, and a useful signal if we need to make difficult tradeoffs within Ganesh or Blink between mobile- and non-mobile content.

The relative priority of "more GPUs/OS" and "more content rendered via Ganesh" depends on our organizational priorities. The "mobile first" focus will push for faster GPU rasterization on basic content and focus implies focus on #2, but the desire to double down on a single rendering path and reduce code complexity implies focus on #1. In great Chrome tradition, we will probably do a mix of both.

Command Line Flags and their Meaning LP

Default behavior: `--enable-gpu-rasterization` but with blacklist check

`--enable-bleeding-edge-rendering-fast-paths`: no effect.

`--disable-gpu-rasterization`: Kill switch for all GPU rasterization

`--force-gpu-rasterization`: Bypasses blacklist, veto and meta-tag (allows us to test the performance of content that is currently veto-ed and make veto/non-veto comparisons)

`--enable-gpu-rasterization`: Bypasses blacklist but respects veto and meta-tag (allows us to test our default mode on a currently blacklisted device). This should in theory be achievable with --ignore-gpu-blacklist but that flag brings also disables some valuable driver workarounds so it's not ideal.

Prevalence of meta viewport tag

Of ~ 45,000 common domains, approximately 145 have the viewport tag specified above (rowser to set the viewport to the width of the device with an initial scale of 1. This example also allows zooming, something that ma) when accessed with a mobile user agent. This data was gathered by taking a list of common domains, fetching their index pages and looking at <head>. It won't count anything included by JavaScript since JS wasn't run, and it won't include domains with non-index documents that have the viewport tag, but should still provide a good approximation.

Initial Performance Measurements

Measurements on Nexus 5 running Android Chrome r266009 indicate a manageable amount of risk for this plan.

smoothness key_mobile_sites: All but three pages run at ~60fps already using CPU rasterization. Improvements in mean_frame_time are therefore not expected. Only a single page regresses with GPU rasterization (reddit.com). This is a known issue (dotted lines) that will soon be vetoed and eventually be optimized in Ganesh.

repaint key_mobile_sites: This benchmark invrepaints the viewport. GPU rasterization brings dalidates the entire page every frame and then own average mean_frame_time from 76 ms to 46 ms (1.65x faster). The best improvement is 4.47x on cuteoverload.com. Only two pages regress: sports.yahoo.com (0.8x) and reddit.com (0.28x, see smoothness for details). Several pages do full repaints at 60Hz with GPU rasterization.

Why the change?

During the launch process of hybrid mode, two sets of problems were identified.

First, we realized that in the future, `will-change: contents`' long term application is as a mechanism to control prepainting and decide whether to retain an element as a texture after rasterization:

- An element that is constantly changing shouldn't be rasterized when offscreen.
- Relatedly, the SkPicture for an element that is constantly changing should be recorded only for the visible part of the SkPicture.
- If the platform supports rasterization of the SkPicture direct to the framebuffer, then a changing element should be drawn right to the framebuffer to save an extra copy [and therefore memory bandwidth and memory footprint]

By using `will-change: contents` to temporarily trigger GPU rasterization, we create an incorrect incentive to authors put this tag on anything that benefits from GPU rasterization. When we then try to make GPU rasterization on by default in the future, this then means that us using this flag for prepainting control would be difficult if not impossible. For this reason alone, we knew we needed to seek an alternative triggering mechanism.

Second, hybrid mode turns out to be a hassle without a huge amount of payoff:

- Initial data show that nearly all mobile sites get better with GPU raster turned on, and that the remaining can be solved with a blacklist.
- Have to wait for ack from sw raster worker pool to activate, even if no sw rasterization has occurred
- Have to maintain a virtual interface with two subclasses in the raster worker pool, with associated test complexity
- GPU rasterized content benefits from larger tiles than s/w raster. Mixing the two modes reduces our ability to recycle textures effectively.

Additionally, we also could not find examples of content for which hybrid mode was a defensible performance strategy. In the cases we considered, it was always better to either (a) veto and fall back to all software rasterization, or (b) just fix Ganesh to work passably well.

Risks

- Although the smoothness benchmark reports no regressions on the new triggering sites, a few of them degrade visually. We need to figure out if missing tiles or slower image uploads are to blame.
- Our confidence in having caught all the sites with min-viewport is better than nothing but we aren't sure we've caught everything.
- We haven't yet done exhaustive performance runs on the devices for our initial launch. N5 has the best coverage, with 2nd gen N7 also getting some recent study.
- Our tooling for understanding why a page is slow is immature. It takes a lot of time and energy to triage a single page.

Alternatives Considered

At the core of the GPU rasterization triggering debate is the fact that Chrome ships on a huge number of OSs, OS versions, and GPUs. Other browser vendors have to support fewer OS' and fewer GPUs, simplifying the content problem considerably.

As part of developing this plan, we considered the following other options:

- OS-specific or device-specific trigger: we are already proposing a launch restricted to a narrow set of devices only on Android. The problem is that this alone still requires we have meets-or-exceeds performance relative to software rasterization. Data indicate that

content rendered via "request desktop site", e.g. desktop content, is still a problem

- Dev-, Beta-, Canary-only triggering, off on stable: GPU has long had an issue that people with these builds often have higher end devices. A large amount of our ship risk lies in the medium and lower-end devices.

- about:flags entry: Similar to a channel-specific flag, risks self-selecting in higher end users and websites more favored by developers than our mainstream customers. Also does not create an active, reliable user population, which prevents content authors from creating content that assumes GPU rasterizationas a baseline.

- %-of-users/pageviews experiment: Would get us TON of data, but would likely be very diverse and shift priority toward non-mobile-oriented content. Our first order goal is mobile-first content.

- Waiting until Ganesh is good enough to use one of the above strategies: although there are acceptable mitigations for the near term, the overall story that the mobile web is viable is considerably better with GPU rasterization launched and working.

- Custom meta tag: although absolutely useful for our goals, it is not very web friendly. No other vendors have needed this, complicating things, and our need for it would be relatively short lived.

- New property on viewport meta tag, e.g. gpu_raster=yes. Same concerns as custom meta tag: only for Chrome, and our need is short lived.

Next Steps

We've got lots to do:

- @ajuma, urgently: switch tip-of-tree over to the new triggering policy. crbug.com/367202
- @bsalomon/@robertphillips: veto dotted lines to dry run veto mechanism
- @alokp: switch cc from hybrid to "veto forces off GPU rasterization on all layers". crbug.com/367198
- @ernstm, @wiltzius, @nduca: start triaging topsites that trigger with this content for performance roadblocks

Getting involved

- Find sites that are slower with GPU rasterization turned on: set GPU rasterization to "enabled" in your about:flags.
- Help us figure out why those sites are slower! And fix them.

- Help us build tooling to tell which part of a page is making GPU rasterization slow
- Contribute to the PictureStream format so we can send problematic use cases in a 100% reproducible way