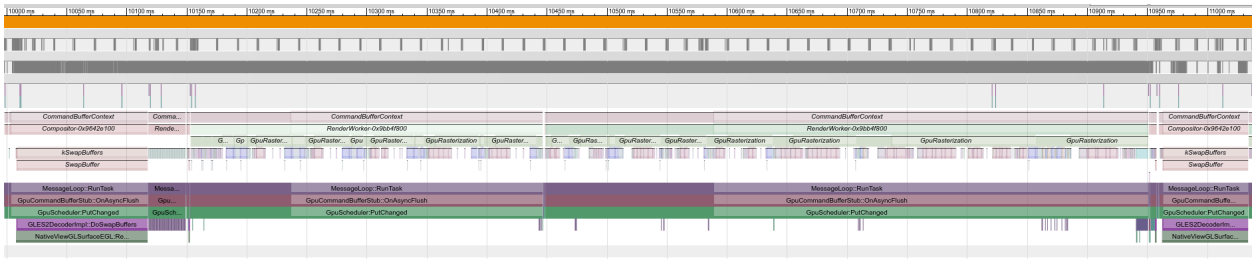
Chrome GPU Service Scheduler

sunnyps@, piman@, vmiura@, dyen@
Tracking bug: crbug.com/514813

Problem

With the introduction of GPU rasterization in Chrome, the ability to schedule raster work on multiple background threads was lost. Instead all raster and compositing work is submitted to the GPU from a single thread in Chrome's GPU process. The order in which raster and compositing work happen depends on the order in which the work was issued by Chrome's renderer process. It is common to emit a lot of raster work that runs for a long time on the GPU process work thread before compositing or presentation.

Trace captured for theverge.com on Nexus 5. There is 800ms of raster work in between two SwapBuffers. This includes both state changes and driver overhead due to submitting too much work.



Background

Chrome has a multi-process architecture where webpages live in individual processes which we refer to as renderer processes. Besides renderers we also have the browser process, the GPU process and processes for plugins e.g. PPAPI. The GPU process uses a client-service architecture whereby the GPU service runs in the GPU process and the clients are the browser, renderers or plugins.

The GPU channel is the fundamental abstraction over the OS pipe between the GPU service and it's client that's used for IPC. IPC messages are sent/received on a dedicated thread called the IO thread and are queued up so that they can be executed later on the GPU process main thread in order. There can be multiple GPU channels, typically one for each renderer, which all want to execute their messages on the main thread.

GL commands are transmitted in command buffers which are shared between the GPU service and client. The offset at which commands have been emitted by the client is passed as the put offset in an AsyncFlush IPC message. The service tries to submit work to the GPU when it gets a flush message. A consequence of having a single queue of IPC messages for each channel on the service is that command buffer flushes from different client contexts on each channel have to run serially with respect to one another.

This serial behavior implies that if too much raster work is emitted, compositing and display gets delayed leading to dropped frames. The cost of raster work even CPU side depends on a variety of factors including the behavior of the GL driver and is hard to predict.

Overview

We propose to extend GPU channels to support multiple streams of execution and implement a co-operative GPU service scheduler for them. These streams will enable asynchronous GPU rasterization so that compositing and presentation is not blocked for raster work.

Detailed Design

Command Buffers And Streams

Streams each have a queue of IPC messages which are run in order. There is no ordering guarantee between different streams just as there is none between channels. The primitives used for synchronization and resource sharing between different streams will be the same as those used for sharing between different renderers i.e. sync tokens and mailboxes.

Command buffers are associated with a stream on creation and stay in that stream throughout their lifetime. Any synchronization between command buffers on different streams will be the responsibility of the client. As a result command buffers in a share group must be in the same stream. A special stream for control messages is always created on channels.

Each stream corresponds to an instance of GpuChannelMessageQueue. The message queue can operate both with existing scheduling features and with the new scheduler. The message queue also keeps track of sync token order numbers so that we can synchronize different streams using sync tokens.

Stream Priorities

Streams can have one of the following priority levels:

- Real-time: Used by the browser's compositor.
- High: Used for control messages such as those which create contexts.
- Normal: Used for renderer compositor context.
- Low: Used for the renderer GPU raster (Ganesh) context.

Stream Synchronization

To synchronize between streams on the same process the existing sync point implementation will be too expensive. In particular since there are no trust issues between streams in this case, there's no need for expensive synchronous IPC to guarantee ordering. These issues and a performant implementation for sync points are discussed in the Lightweight GPU Sync Points proposal.

Async GPU Raster

The scheduler proposed is meant to be used for enabling an asynchronous mode for GPU rasterization. The worker thread on which Ganesh issues commands uses a context that's on its own low priority stream. Each raster task has a sync token associated with it. The compositor adjusts the priority of the rasterization by synchronizing with the raster task it chooses and relies on the GPU service scheduler to ensure that rasterization does not supersede other compositing. Details can be found in this proposal.

Scheduling Algorithm

(Out of date - see https://codereview.chromium.org/2814843002/ for implementation details)

The existing GPU scheduling logic is based on using PostTasks to run messages on a channel. This makes it essentially a FIFO system with the caveat that there is a way to prevent execution on low priority channels by means of a shared flag. There is a complicated state machine which controls when this flag is set by a single high-priority channel.

We propose to implement a global GPU service scheduler based on a multi-level round-robin scheduling algorithm. A brief description of the scheduling algorithm follows:

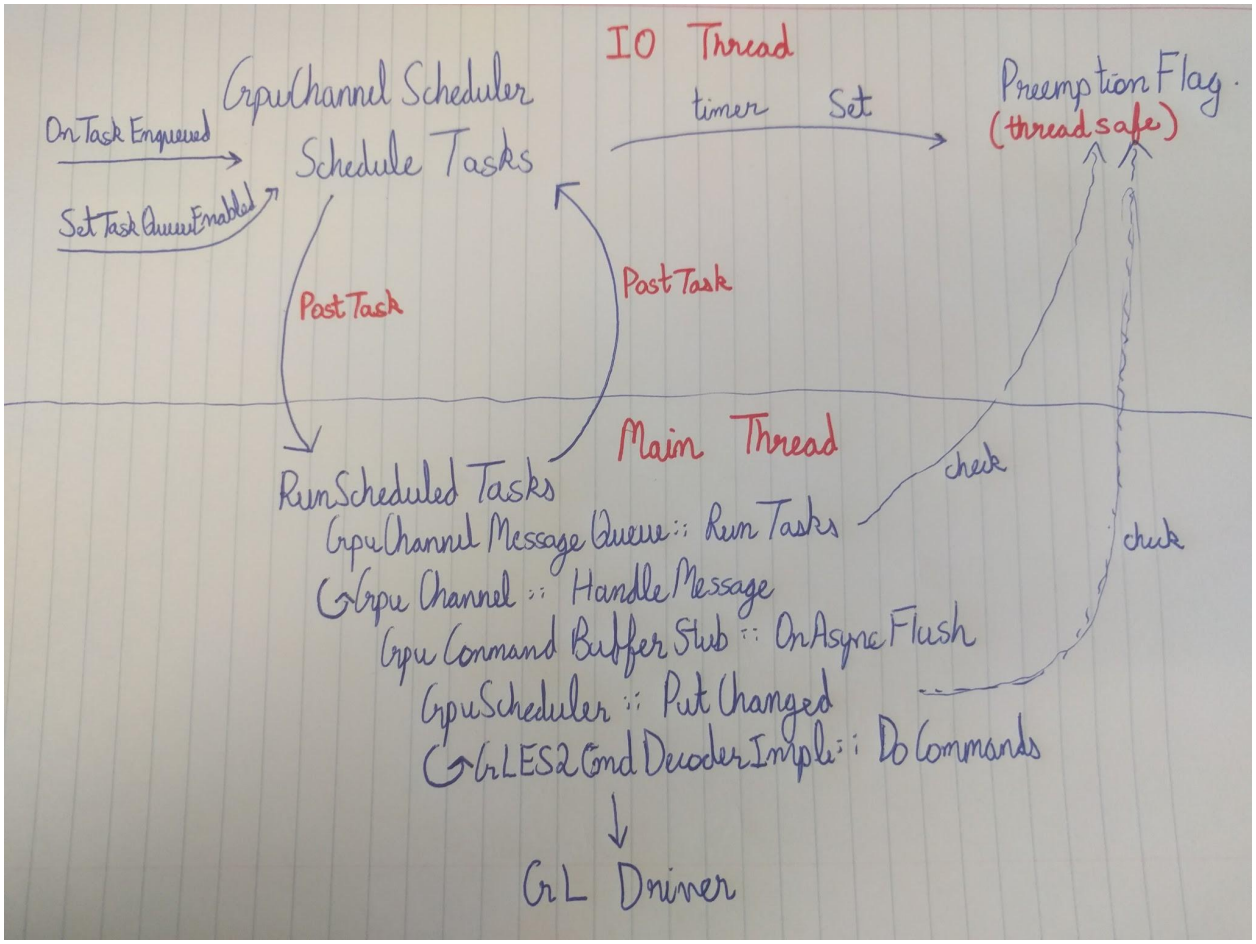
- When a stream becomes active it is enqueued at the back of a FIFO queue based on its priority level.
- When choosing a stream the front of the queue with the highest priority level is chosen as the next stream to run. The stream only runs as many messages as have been received at the time it's asked to run.
- If a stream runs beyond its time slice, it is downgraded in priority unless it's priority is real-time. Otherwise the stream is reset to its original priority. In either case the stream is enqueued at the back of the appropriate queue.
- If a stream with higher priority than the current stream becomes active we reduce the time slice of the current stream subject to a certain granularity.
- When a stream waits on another stream using sync tokens, the latter stream's priority is increased to the level of the waiting stream. This applies transitively so that if the browser waits for a renderer which in turn waits for async raster, the async raster runs at the browser's priority.

The scheduler uses a shared preemption flag for requesting the current stream to yield. The preemption flag is checked after every N GL commands or after particularly expensive commands. This is similar to how the existing scheduling logic works.

Scheduler API

```
class CONTENT_EXPORT GpuChannelScheduler
    : public base::RefCountedThreadSafe<GpuChannelScheduler> {
 public:
  static scoped_refptr<GpuChannelScheduler> Create(
      scoped_refptr<base::SingleThreadTaskRunner> main_task_runner,
      scoped_refptr<base::SingleThreadTaskRunner> io_task_runner,
      scoped_refptr<gpu::PreemptionFlag> preemption_flag);

  void Destroy();

  void AddTaskQueue(GpuChannelTaskQueue* queue, GpuStreamPriority priority);
  void RemoveTaskQueue(GpuChannelTaskQueue* queue);

  // This returns the dynamic priority of the queue.
  GpuStreamPriority GetScheduledPriority(GpuChannelTaskQueue* queue) const;

  // These methods do not have any effect until the next time the queue is
  // scheduled.
  void SetTaskQueueEnabled(GpuChannelTaskQueue* queue, bool enabled);
  void SetTaskQueuePriority(GpuChannelTaskQueue* queue,
                            GpuStreamPriority priority);

  void OnTaskEnqueued(GpuChannelTaskQueue* queue);
}

class GpuChannelTaskQueue {
 public:
  // Returns true if there are more tasks to run.
  virtual bool RunTasks() = 0;
};
```

Scheduler Operation Diagram



Caveats

A scheduler based on the CFS algorithm was prototyped but it wasn't a good fit because the browser's context can spend a lot of time inside a swap and that would cause the scheduler to allocate more time to other contexts in order to ensure fairness. It might be possible to work around this but the workaround would be unnecessarily complex. In any case the behavior of the CFS scheduler was found to be unpredictable.

We considered using the Blink scheduler implementation as it has very similar goals to this scheduler. It was rejected because it uses instances of base::Task which wraps a closure as a basic unit of work and is also more complex. Our basic unit of work is an IPC::Message and using a closure would have been unnecessary overhead. Furthermore the above system can be extended to support other kinds of tasks e.g. we could have a mix of task runners, Mojo and IPC task queues which all have different representations for tasks.

Future Work

The first version of async GPU raster will use a single context on its own stream but we could experiment with multiple contexts and streams. The scheduler should prioritize streams based on what higher priority streams wait on. This could help in reprioritizing raster work based on tile priorities.

On most platforms we use virtual contexts that are emulated on a single real GL context. Preempting a context is costly especially when it happens inside a render pass. The state set up for the render pass is lost and has to be reset when the context runs later. We should check the preemption flag before beginning a new render pass.

The scheduler proposed here only deals with CPU side work although it also has the side-effect of reducing GPU side work. Extending the scheduler to throttle streams based on GPU side work will be challenging but important. A lot of mobile websites make heavy use of CSS features such as blurs and filters. While it's important to implement these features when a page is animating, it's not as important while flinging, scrolling or during a pinch-zoom.

Appendix: Related Links

State of GPU Scheduling
Asynchronous GPU Rasterization
Lightweight GPU Sync Points
CFS Scheduler Prototype