T-Sync

A Tile-Based GPU to Display Interface

Status: Public. Work in progress. Seeking feedback.

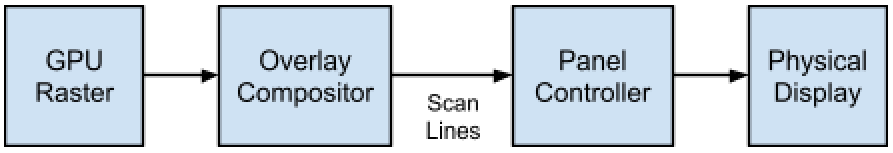
Summary

This document describes a display interface and potential API that allows a GPU to stream its output directly to display, which will:

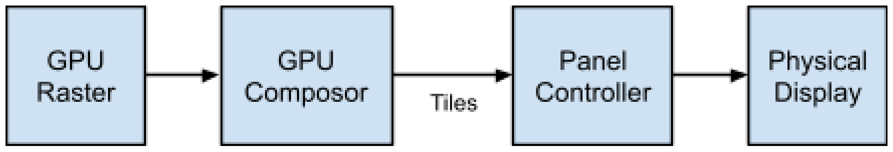
1. Save one write to and one read from DRAM per pixel per frame.
2. Reduce the effective display latency, especially when combined with features like Asynchronous Shader Input from the CPU, Sync Point Shader Arguments, and Trustable Future Sync Points.
3. Remove the need for a feature-constrained overlay compositor.

The following diagrams summarize the difference between how the display pipeline looks today versus how it would look with a tile-based interface. Some vocabulary:

- **Overlay compositor**: can generally composite 3 to 5 screen-aligned rects
- **Panel controller:** drives the physical panel and may contain a copy of the framebuffer currently displayed to save bandwidth during self-refresh.



Normal flow



TSync based flow

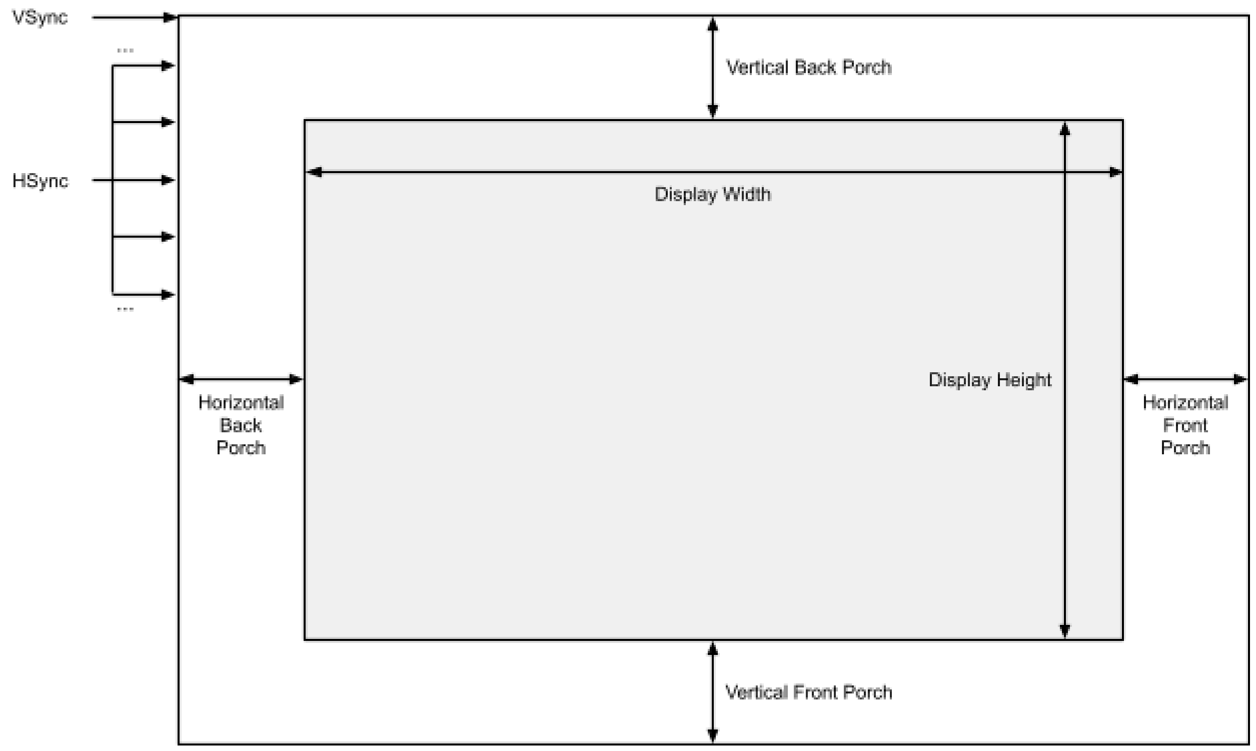
There are drawbacks to a T-Sync system, so the pros/cons must be weighed. Although power is saved by reducing DRAM bandwidth, a GPU compositor is going to be more power hungry than an overlay compositor. And although a GPU compositor could support many more compositing use cases than a simple overlay compositor, it becomes more difficult to guarantee throughput and latency, requiring more complicated flow controls to prevent underflow (i.e. displaying pixels that aren't ready).

To prevent underflow, this document proposes two flow-control approaches that can both adjust the VSYNC *and* HSYNC frequencies and intervals dynamically:

1. Conservatively displaying the frame *after* the GPU Compositor complete.
2. Aggressively displaying the frame *as* the GPU Compositor completes.

Background

Here is a diagram of important display concepts:



Some more vocabulary:

- **VSync**: The start of the entire display's scan out.
- **HSync**: The start of each scan line.
- **Vertical/Horizontal Front/Back Porch**: An invisible/imaginary buffer of pixels around the visible display. Historically, these helped hide the time it takes to transition the magnetic field within a CRT display; but current display technologies still use the porches to hide memory and compositing latency to avoid underflow and to target a specific refresh rate given a fixed clock that ticks once per pixel.

Most displays refresh their physical pixels at a fixed VSYNC interval. While providing a consistent interval works well for content that can output at the given rate, it is not ideal for content that cannot reach that rate since the next consistent rate for the content is half the display's refresh rate. Even for content that is fast enough, the phase of the VSYNC will add unnecessary latency since a framebuffer may have to wait for presentation after it's completion.

GSync

GSync addresses many of the issues resulting from a fixed VSYNC interval by allowing the application to trigger the VSYNC more directly. The display sits in an idle state while it waits for the application to complete it's drawing and swap. The swap triggers the VSYNC immediately, removing any delay and allowing for refresh rates that don't have to be multiples of some fixed rate. *Note: There is a min and max frequency at which a display must operate, so the display's waiting behavior is slightly different at those extremes.*

VESA

- VESA has also added Adaptive Sync to the DisplayPort standard, which appears to be very similar to G-Sync.
- VESA has also added Display Stream Compression that compresses the communication of pixel data.

Panel Controller Buffers

Some panel controllers have an internal buffer that stores the last frame sent to the display. For example, Renesas' UPD60802A. This helps save power in certain cases by allowing the panel controller to refresh the display independently since panels start to exhibit corruption when the refresh gets too low.

Potential hardware mechanisms for sending tile data

1. The status quo
    a. Composite to DRAM and simultaneously scan out to the panel's frame buffer.
    b. Simple and cheap, but not an efficient use of bandwidth.
2. Mipi DSI
    a. Existing chips seem to support 1 Gbps per data line.
    b. Would be like pushing tiles through a straw, making it hard for the GPU to win any races.
    c. Would also result in underutilization of a GPU block if it is perpetually bandwidth limited.
3. UniPro - A high bandwidth potential successor to Mipi DSI
    a. Press release indicating UniPro bandwidth will be 6 Gbps per lane.
    b. Better, but lanes would sit idle if the GPU races as fast as this proposal wants it to be.
    c. For a balanced system, might require a separate GPU tuned for lower bandwidth.
4. Bring back VRAM!
    a. Can we have it sit on the same bus as the system RAM?
        i. Would remove the need for a dedicated display data bus.
        ii. Likely cheaper per MB that putting memory in the panel controller.
        iii. Would not remove pixel writes through the system's memory controller.
    b. VRAM supports two I/O ports. The second port could be used for the display to avoid reading through the system's memory controller.
5. GDDR RAM
    a. Simulates the dual port nature of VRAM, but without physical ports by supporting two active DRAM rows, instead of just one.

   b. Reads and writes would still have to go through the system's memory controller, but the bandwidth efficiency might improve.

High-Level Proposal

Unless we can calculate with certainty that a given stream of GPU Compositing commands will execute fast enough to beat the scan out, there will be a risk of underflow if we make them race. Therefore a conservative scan out approach is proposed for cases where we are not certain and a separate, more aggressive, approach is proposed for cases where we are certain.

Being able to switch between the conservative and aggressive approach will require double buffering in the panel controller. This will increase hardware costs, but will also increase DRAM space and bandwidth available to the rest of the system.

Conservative Approach: Scan out fast on frame completion

1. Have the GPU Compositor draw to the panel controller's back buffer.
2. Once the GPU Compositor completes, swap the panel controller buffers and start the vsync.
3. If GPU Compositing is too slow, the panel controller will have to scan out it's front buffer one or more times.
   a. Ideally this wouldn't be the case so that we don't need two frame buffers. Are there any panel technologies that don't require refresh at all, whether self refresh or "host" refresh, if the content hasn't changed?
4. Make the vertical and horizontal porches as small as possible to reduce latency and wobbliness.
   a. There is no GPU Compositing latency to hide and on-chip memory access should be fast within the panel controller.

Aggressive Approach: Race a slower scan out

1. Make the vertical back porch very large to hide any GPU Compositor variability and latency.
2. Allow the horizontal porches to vary frame-to-frame to control the speed of the scan out.
   a. Slowing down the scan out will make it easier for the compositor to win the race.
   b. Speeding up the scan out will make the display less wobbly.
   c. Changing the speed too much or too often could result in "jumpy wobbliness" and should be avoided.
   d. Might it help to slow down the scan out mid frame?
      i. Could allow us to be more aggressive while still avoiding underflow.
      ii. Might result in a "squiggly wobbliness" though.
3. To improve scheduling, could we use a higher-than-necessary self refresh rate that we can interrupt mid scan out to scan out the new frame instead?
   a. Would interrupting a scan out result in any kind of corruption?

Transition from Conservative to Aggressive
TBD

Transition from Aggressive to Conservative
TBD

Low-Level Details: Wire Interface
TBD

The API
All of the details regarding how T-Sync works could be hidden from the application, but the results would not be optimal for three reasons:

1. The latency characteristics would be different in the conservative vs. aggressive approach, so it would be nice for there to be feedback into the shaders and back to the application regarding the current approach used.
2. A driver working with low-level information of the frame being drawn will have a difficult time deciding whether or not it can use the aggressive approach. Whatever heuristics it uses would have to be more conservative than if it were provided with extra high-level restrictions of the scene.
3. If the driver provides feedback regarding what's possible in the aggressive approach, then the application could change its compositing decisions. For example, Chrome's Ubercompositor could decide to composite everything except the foreground tab to an off-screen buffer first to reduce the final frame's complexity.

API Details
TBD

Discussion
1. **Hybrid Approach**: Should we allow a hybrid approach that includes a conservative phase plus aggressive phase? For example, a compositor could decide to composite a desktop and background tabs conservatively first to the back buffer in the panel controller. Then the foreground content could be composited using the aggressive approach over the background content during scan out.
    a. To avoid having to read from the buffer in the panel controller, the aggressive phase would have to be able to mask off the area of the screen it's drawing to, which may limit the aggressive phase to a screen-aligned rect for simplicity and bandwidth considerations.
    b. Two phases would use more bandwidth than just compositing everything to the back buffer using just the conservative approach.