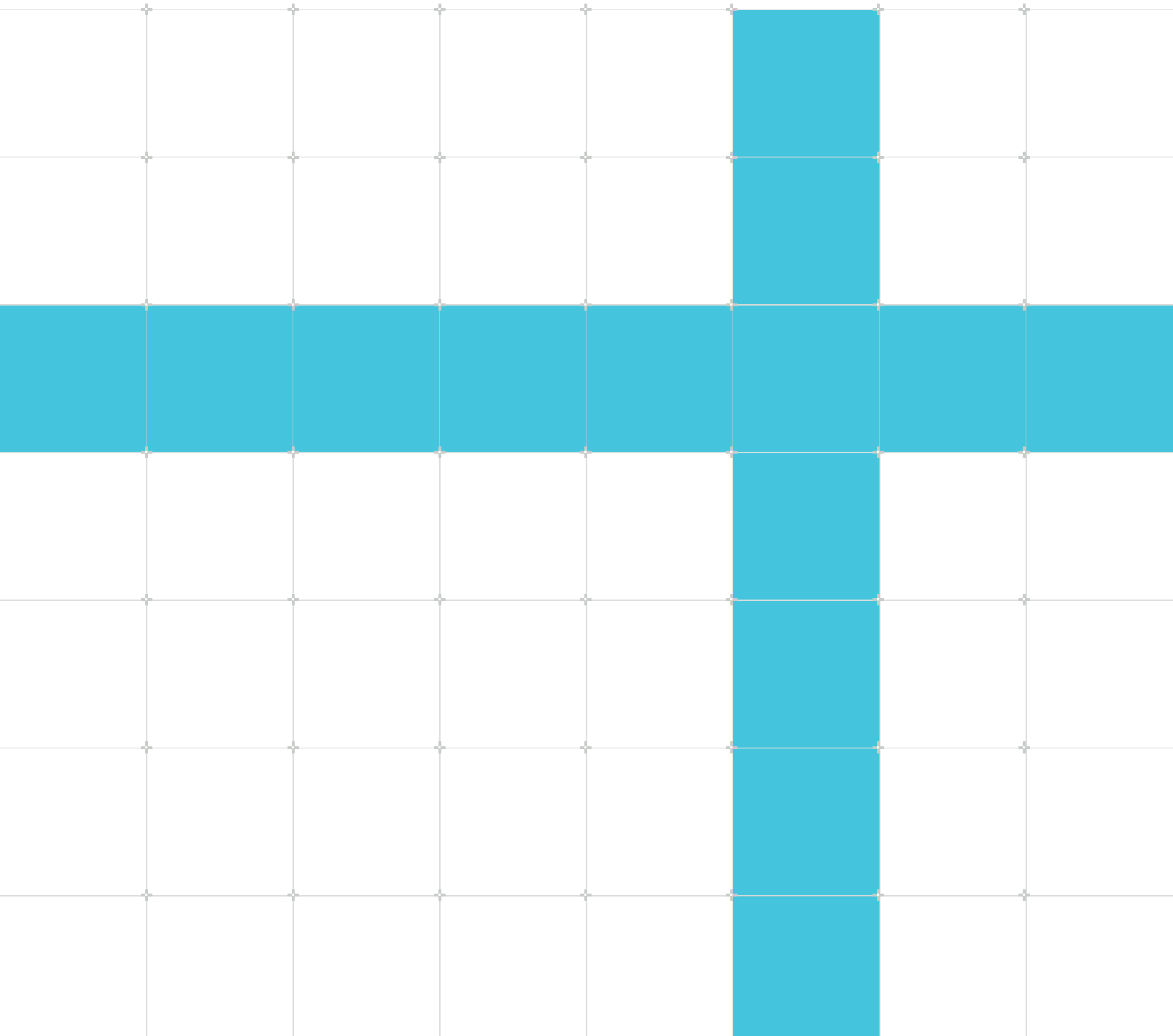
Workload Pipelining

Version 1.0

Issue 02
102521_0100_02_en



Workload Pipelining

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	30 April 2021	Non-Confidential	New document for v1.0

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview....6

2. Efficient parallel processing....7

3. Going parallel....8

4. Pipeline Bottlenecks....9

1. Overview

In this guide, read about how workload pipelining works.

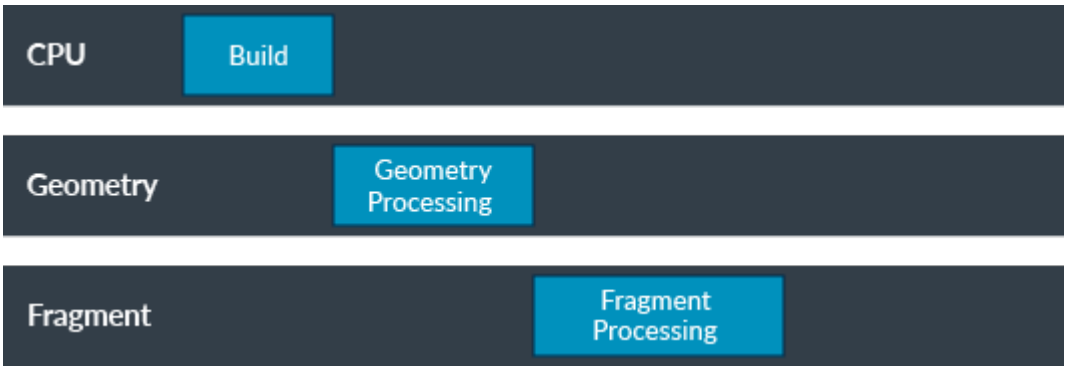
2. Efficient parallel processing

The processing of a render pass on a Mali GPU is split into three distinct phases:

1. The application specifies the render pass using the graphics API.
2. The vertices for the whole render pass are shaded and tiled.
3. The pixels for the whole render pass are shaded tile-by-tile.

These three phases happen serially for each render pass; the application must completely specify a render pass before geometry processing can start, and geometry processing must complete before fragment processing can start. Over time this looks like:

Figure 2-1: This is a diagram of a basic pipline.

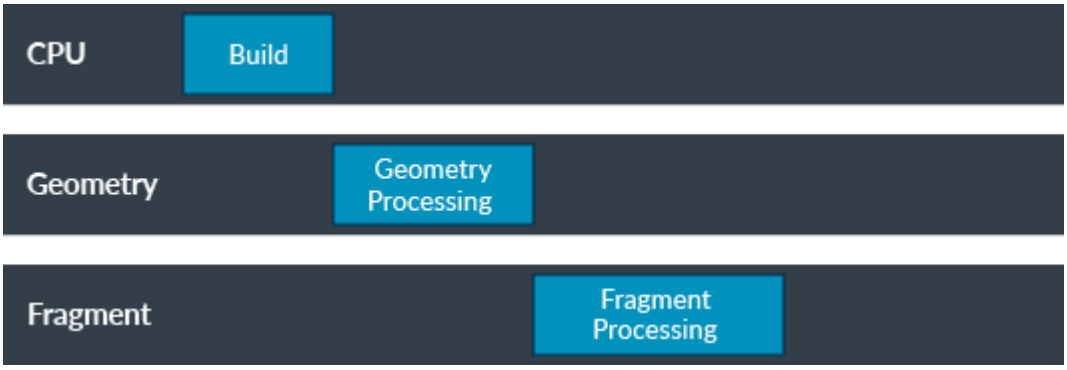


Note that this diagram shows each phase starting immediately after the previous one finishes, but in reality the three processing queues are decoupled. If a queue is busy processing an earlier render pass then there may be a delay in starting its workload.

3. Going parallel

To get best performance the graphics stack aims to process multiple render passes in parallel: one being built on the CPU, one being vertex shaded, and one being fragment shaded. The rendering pipeline is therefore very deep – many milliseconds in length – and can even overlap render passes belonging to two neighboring frames. This overlapping of workloads ensures that the available processing units are kept busy, all of the time.

Figure 3-1: This is a diagram of a basic pipeline multiple.



The processing time of each of the three component workloads of a render pass is not usually identical. A well scheduled workload which is processing limited will see the most heavily loaded pipeline stage running all of the time, and the other two going idle periodically waiting for the slowest stage to catch up. The swim lane diagram below shows the typical pipelining for two frames of content which is bottlenecked by fragment processing performance.

Figure 3-2: This is a diagram of a basic pipeline multiple frag.

4. Pipeline Bottlenecks

One of the first tasks to undertake when optimizing an application is to review how well the system is behaving under load, and how well the graphics workloads are being scheduled on to the GPU hardware.

Processing bound

The most common reason for content to miss its performance goal is that one of the processing stages is overloaded with too much work. In this scenario we would expect to see one of the processing stages running 100% of the time, and this is the area where optimizations should be focused.

Info

All Mali GPUs except the Utgard family GPUs use a unified shader core, so there is a shared resource across the two GPU processing slots. Optimizations to either processing slot can free up these shared resources for use by the other, so optimizing the slot which is not the critical path can have benefits.

Thermal bound

High-end smartphone systems can only dissipate 2 to 3 Watts of heat, depending on packaging, but can generate more than this under high load. Applications which continuously stress the CPU, GPU, and DDR memory will eventually cause the device to throttle component frequencies to avoid overheating.

If a device is overheating the application profile will often look similar to a processing bound device – one pipeline stage will be 100% loaded – but will have either unstable frequencies or lower frequencies than expected. The thermal limit is system wide, so optimizing any pipeline stage will help even if it is not the stage on the critical path.

Pipeline bubbles

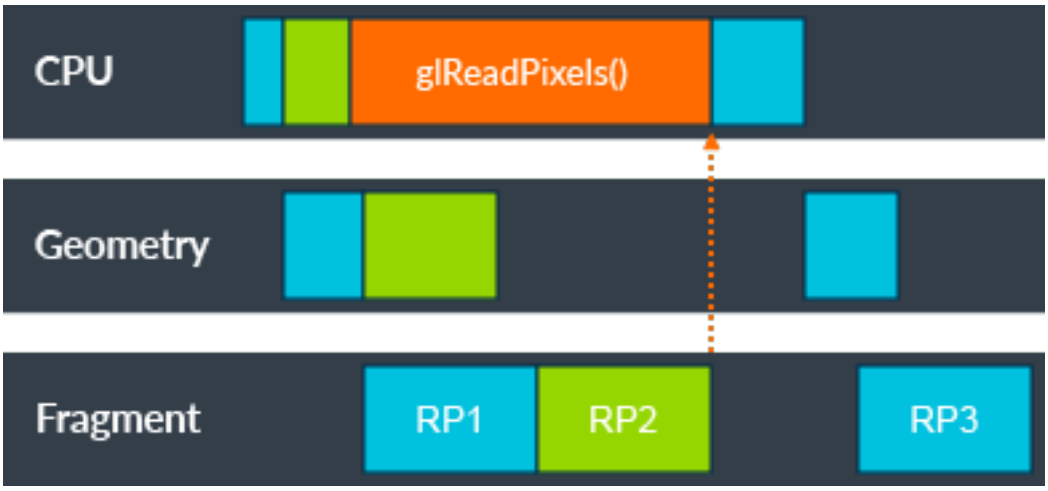
Another reason for poor performance is a lack of overlap in the workload scheduling, meaning that the system fails to keep the hardware busy all of the time due to idle bubbles in all of the queues. In most cases serialization is caused by application API usage which either drains the pipeline, or causes a dependency between render pass workloads in the two queues.

The OpenGL ES API is specified to behave synchronously; later API calls must behave as if all earlier API calls have completed. In reality this is an elaborate illusion maintained by the device driver; rendering is asynchronous and processing for an API call may happen many milliseconds after the API call was made. If an application ever enforces the synchronous requirement – for example by calling `glReadPixels()` immediately after a draw call – the pipeline must be drained to resolve the data needed for the pixel read, and then refilled with new work. During this drain and refill process, the GPU processing queues will run out of work and idle until new work is available.

An example of a synchronous pipeline drain is shown below. The `glReadPixels()` requires the shading of RP2 (shown in green) to complete to provide the necessary data, so the CPU processing will block and idle until the data is available. Once this happens the pipeline will start to refill with

the next render pass – RP3 – but it will take some time to work through the pipeline stages and so we see some idle time bubbles in all of the queues.

Figure 4-1: This is a diagram of bad pipeline readpixels.



The Vulkan API has a different set of behaviors which can cause problems. The API is specified to behave asynchronously, matching how modern GPU hardware works. To ensure that rendering completes successfully the application uses the API to define scheduling dependencies between workloads. For example, dependencies must ensure that the fragment shading of render pass “A” has completed before fragment shading starts for a later render pass “B” which reads the output “A” as an input texture. Vulkan allows dependencies to be specified at the rendering pipeline stage level, not just between entire render passes. In our earlier example vertex shading for “B” could safely overlap with fragment shading for “A”, ensuring the pipeline stays full. Overly conservative dependencies which force the start of one render pass to wait for the end of another will quickly reduce performance due to the bubbles they introduce.

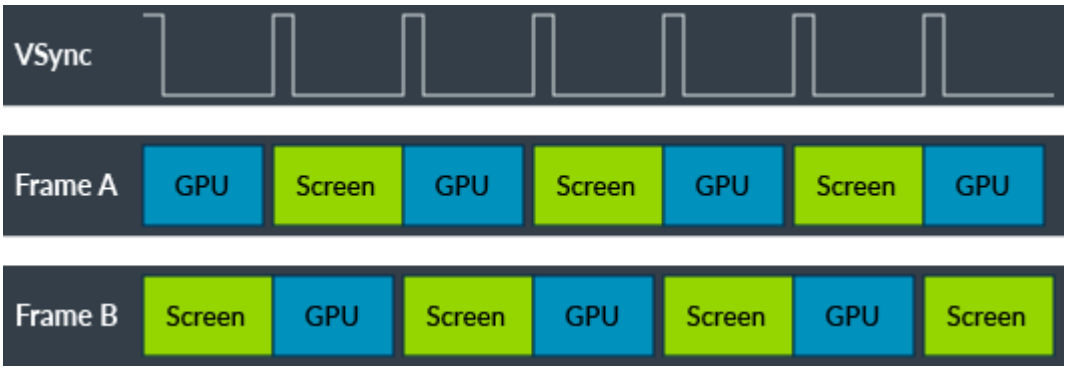
An example of bubbles caused by a serialized pipeline is shown below. The dependencies have been set up in a way which requires each entire render pass to complete before any of the rendering can start for the next render pass, so there is no overlap across the geometry and pixel processing work queues. This can be avoided by not using overly conservative source and destination stage dependencies.

Figure 4-2: This is a diagram of bad pipeline dependencies.

Display vsync

On most consumer devices the display of new frames on screen is locked to the panel's refresh signal, known as the vertical sync or "vsync" signal. Most panels refresh at 60 FPS, so any application running faster than this will be limited by the display update rate and idle waiting for the display panel to swap framebuffers in order to free up a new buffer to render in to. In mobile platforms frequency scaling would trigger at this point with the aim of minimizing the GPU frequency and voltage while still hitting 60 FPS.

Figure 4-3: This is a diagram showing a vsnyc.

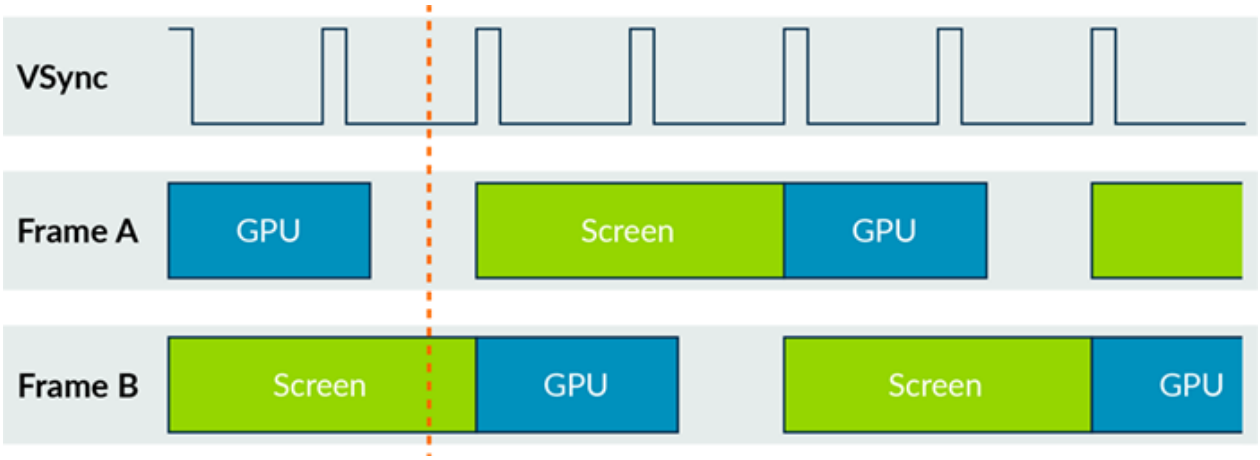


In a system using two buffers for a framebuffer – double-buffering – it is possible to have content which is not hitting 60 FPS but that is still limited by the vsync rate. This occurs because the GPU cannot modify a framebuffer which is still being scanned out to the display, and the GPU runs out of buffers to render in to.

Consider an example where an application is running at 45 FPS in terms of GPU processing workload. It will complete rendering half way though the second vsync period after starting. The back-buffer (A in the diagram below at the orange time marker) has just finished rendering and is queued waiting to be displayed, but the front-buffer (B in the diagram below at the orange time marker) is still locked for scan-out by the display controller. The GPU will therefore go idle until

the next vsync signal occurs, at which point the display buffer swap happens and frees up the old front-buffer for new rendering.

Figure 4-4: This is a diagram showing a vysnc slow.



In this situation the user-visible frame rate will snap to an integer division of the vsync rate – i.e. 30 FPS, 20 FPS, 15 FPS, etc. – despite having a GPU which could render the application faster. If you see this issue it is recommended that you disable vsync locking while doing performing optimization because it will mask improvements and makes measuring progress difficult.

Info

Note that Android systems typically use triple buffering, and so avoid this problem because the GPU has an additional framebuffer available to render in to.