Arm® Mali™ GPU Best Practices

Version 2.0

Developer Guide

arm

101897_0200_00_en

Arm® Mali™ GPU Best Practices

Developer Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	27 October 2017	Non-Confidential	First release of version 1.0
0100-00	30 June 2019	Non-Confidential	First release of version 1.1
0200-00	30 October 2019	Non-Confidential	First release of version 2.0

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

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.

Web Address

www.arm.com

Contents
Arm® Mali™ GPU Best Practices Developer Guide

Preface

This preface introduces the *Arm® Mali™ GPU Best Practices Developer Guide*.

It contains the following:

About this book

This guide provides recommendations for efficient API usage on Mali GPUs.

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r*1*p*2*, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This guide is for experienced software engineers who want to optimize the performance of the graphics in their application.

Using this book

This book is organized into the following chapters:

Chapter 1 Overview

This quick-reference guide describes best practices for Mali GPUs. The topics in this guide are relevant for developers who are getting started with Mali GPUs.

Chapter 2 Optimizing application logic

It is important to optimize application logic to minimize the CPU load that is generated by the graphics driver.

Chapter 3 CPU Overheads

There are various ways to reduce CPU overheads to increase efficiency and reduce software processing costs. To reduce CPU overheads, consider using: shader compilation, pipeline creation, memory allocation, memory mapping, command pools, command buffers, descriptor sets, and layouts.

Chapter 4 Vertex shading

The efficiency of vertex processing, in terms of both buffer packing and vertex shading, is important when rendering a scene. Avoid poor mesh selection and inefficient vertex data encoding on mobile devices, as it can significantly increase DRAM bandwidth.

Chapter 5 Tessellation, geometry shading, and tiling

This chapter covers ways on how to optimize tessellation, geometry shading, and tiling instances in your application.

Chapter 6 Fragment shading

This chapter provides multiple areas of guidance on how to optimize the fragment shading elements of your application on Mali GPUs.

Chapter 7 Buffers and textures

This chapter contains information on how to optimize the performance of your in-game textures on a Mali GPU.

Chapter 8 Compute

This chapter covers how to optimize workgroup sizes, how to correctly use shared memory on a Mali GPU, and optimized ways to process images.

Chapter 9 Shader code

Writing optimal shader code through correct precision, vectorizing, uniforms and other techniques is key to optimizing the graphics performance of your application.

Chapter 10 System integration

This chapter covers how to optimizer system integration, for the Vulkan swapchain and other measures.

Appendix A Revisions

This appendix provides additional information that is related to this guide.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic

Introduces special terminology, denotes cross-references, and citations.

bold

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

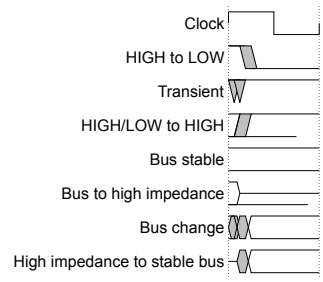

Figure 1 Key to timing diagram conventions

Signals

The signal conventions are:

Signal level

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n

At the start or end of a signal name, n denotes an active-LOW signal.

Additional reading

This document contains information that is specific to this product. See the following documents for other relevant information. See *https://developer.arm.com* for additional Arm documents.

Arm publications

- *https://community.arm.com/developer/tools-software/graphics/b/blog/posts/killing-pixels---a-new-optimization-for-shading-on-arm-mali-gpus*
- *https://www.arm.com/products/development-tools/graphics/arm-mobile-studio*
- *https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-offline-compiler*

Other publications

None.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Arm Mali GPU Best Practices Developer Guide*.
- The number 101897_0200_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Overview

This quick-reference guide describes best practices for Mali GPUs. The topics in this guide are relevant for developers who are getting started with Mali GPUs.

It contains the following sections:

1.1 Before you begin

This guide provides best practices for a Mali GPU. However, real-world applications can be complicated and there are always exceptions to this generalized advice. We recommend that you make measurements of any applied optimizations to check that they are performing as intended on your target devices.

Note

This guide is aimed at established developers. We therefore assume that you have experience as a graphics developer, and that you have a general familiarity with the underlying APIs.

1.2 The graphics rendering pipeline

Graphics processing can be represented as a pipeline of processing stages that includes the application, the graphics driver, and the various hardware stages inside the GPU.

Most stages follow a strict architectural pipeline, with outputs from one stage becoming the inputs into the next stage.

The following figure shows the graphics pipeline beginning at the application, and ending at the depth, color, and stencil buffers:

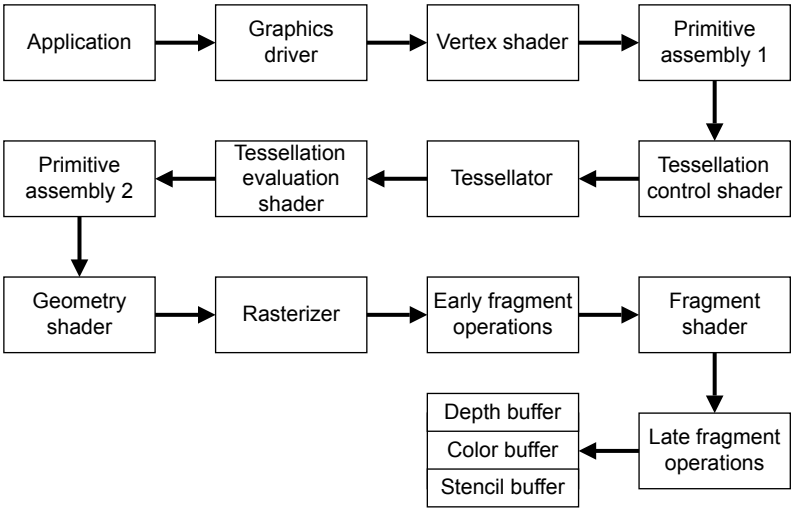

Figure 1-1 The stages of the graphics pipeline

Compute shaders are the exception to this strict pipeline due to writing shader results back to system memory. Any stage of the pipeline that can consume a buffer or texture resource can use the compute shader outputs.

Guide structure

This guide has been structured to follow the graphics rendering pipeline. Each topic gives recommendations that can be applied to workloads that are running in that pipeline stage.

There are some generic topics, such as shader program authoring advice, that apply to multiple pipeline stages. These sections can be found at the end of our guide.

For each best practice topic we provide an explanation of the recommendation, along with actionable technical points that can be considered during development. Where possible, we document the likely impact of not following the advice, and techniques to use when debugging.

We provide relatively terse advice in this guide. For example, we tell you not to use `discard` in fragment shaders. It is inevitable that there are cases where you must use a feature that we do not recommend using, because the feature is required for some algorithms. Therefore, do not feel restricted by our advice to never use a feature. However, always keep in mind that there could be an underlying performance risk.

Chapter 2
Optimizing application logic

It is important to optimize application logic to minimize the CPU load that is generated by the graphics driver.

The CPU is used to process the application logic, drive the graphics stack, and run the code inside the graphics driver. This software is the first potential source of performance issues that your application can run into.

It contains the following sections:

2.1 Draw call batching best practices

Committing draw calls to the command stream is an expensive operation in terms of CPU driver overheads. Draw-call related run-time costs are higher on OpenGL ES than on Vulkan.

Prerequisites

You must understand the following concepts:

- Draw calls.
- Instancing.

Batch rendering and command buffers to reduce draw calls

Draw calls that contain few vertices and fragments take less time to process on the GPU than the CPU time that is required to dispatch the workload. This means that the performance of the application is CPU-limited because the CPU is unable to keep the GPU busy.

These issues encourage draw call batching. Draw call batching merges rendering for multiple objects that use the same render state into a single draw call. This reduces the total number of draw calls that must render each frame, which lowers the CPU processing cost and energy consumption.

How to optimize draw call batching

Try using the following optimization steps:

- Batch objects to reduce the draw call count.
- Use batches, even if not CPU-limited, to reduce system power consumption.
- Use instanced draw calls when drawing multiple copies of a mesh. This instancing allows the application to cull instances that are, for example, outside the view frustum.
- In OpenGL ES, aim for fewer than 500 draw calls per frame.
- In Vulkan, aim for fewer than 1000 draw calls per frame.

——— **Note** ———

Because CPU performance can vary widely between chipsets, these draw call count recommendations are approximate guidelines for current Mali hardware.

———————————

Behaviors to avoid when draw call batching

Arm recommends that you:

- Do not make batches so large that frustum culling, and sorting for front-to-back rendering order, become compromised and inefficient.
- Do not render many small draw calls, for example single points or quads, without batching.

Negative impacts of unoptimized draw call batches

The different types of impact you can see are:

- Higher application CPU load, because of a high draw call count.
- A reduction in overall performance if the application is CPU bound.

Debugging when draw call batching

Try the following debugging tips:

1. Profiling the application CPU load.
2. Counting the number of draw calls per frame.

2.2 Draw call culling best practices

The fastest draw calls that an application can process are the ones that are discarded before they reach the graphics API.

Prerequisites

You must understand the following concepts:

- Culling.
- Primitives.
- Vertex shading.

Minimizing the number of draw calls

Application culling of entire meshes is always more efficient than per-primitive culling in the GPU, as it can exploit scene knowledge. GPU culling can only be performed after the primitive clip-space coordinates are known. Therefore, the vertex shading must be executed even if the primitive is eventually culled.

How to optimize culling draw calls

Try using the following optimization steps:

- Cull objects that are out of frustum on the CPU. For example, by using bounding box frustum checks.
- Cull objects that are known to be occluded on the CPU. For example, by using portal culling.
- Experiment to find a balance between batch size and culling efficiency.

Outcome to avoid when draw call culling

Do not send every object to the graphics API.

Negative impacts on the CPU and GPU

Unoptimized draw call culling can lead to the following problems:

- Higher application CPU load, because of unnecessary draw calls.
- High GPU vertex shading and bandwidth, because of redundant vertex shading.

Debugging draw call culling problems

Try the following debugging tips:

- Profile the application CPU load.
- Use GPU performance counters to verify tiler primitive culling rates. Expect around 50% of the triangles to be culled because they are back-facing inside the frustum. Higher culling rates can indicate that the application needs an improved draw call culling approach.

2.3 Optimizing the draw call render order

When rendering draw calls, the GPU can reject occluded fragments efficiently using the early-zs test. Efficient draw call order can maximize the benefit of the early-zs test by removing as many occluded fragments as possible.

Prerequisites

You must understand the following concepts:

- Command buffers.
- Draw calls.
- Early and late zs-testing.

Increasing culling rates using early-zs testing and *Forward Pixel Kill* (FPK)

To get the highest fragment culling rate from the early-zs unit, first render all opaque meshes in a front-to-back render order. To ensure blending works correctly, render all transparent meshes in a back-to-front render order, over the top of the opaque geometry.

All Mali GPUs since the Mali-T620 GPU includes the FPK optimization. FPK provides automatic hidden surface removal of fragments that are occluded, but early-zs testing does not kill. This is due to the use of a back-to-front render order for opaque geometry. However, do not rely on the FPK optimization alone. An early-zs test is always more energy-efficient, consistent, and works on older Mali GPUs that do not include hidden surface removal.

How to optimize draw call render ordering

Try using the following optimization steps:

- Render opaque objects in a front-to-back order.
- Render opaque objects with blending disabled.

Behaviors to avoid when optimizing draw call render ordering

Arm recommends that you:

- Do not use `discard` in the fragment shader, as it forces the late-zs test.
- Do not use alpha-to-coverage, as it forces the late-zs test.
- Do not write to fragment depth in the fragment shader, as it forces the late-zs test.

The negative impact of an unoptimized draw call render order

Suboptimal draw call render ordering, and the use of late-zs testing, incurs a higher fragment shading load, because of fragments that are visually occluded, but were not killed before fragment shading.

Debugging draw call render order problems

Try these steps when debuging:

- Render your scene without your transparent elements, and then use GPU performance counters to check the number of fragments that is being rendered per output pixel. If the number of fragments is higher than 1, you have opaque fragment overdraw that the early-zs test can remove.
- Use the GPU performance counters to check the number of fragments that require late-zs testing; check the number of fragments that late-zs testing kills.

2.4 Avoid using depth prepasses

Depth prepasses are a common technique in PC and console games development. It is used in scenarios where there is potentially many overdraw and expensive fragment shaders. It is also used where you cannot reliably get front-to-back sorted opaque geometry.

Prerequisites

You must understand the following concepts:

- Early zs-testing.
- Render passes.

An optimization that reduces performance

The aim of the depth prepass is to quickly set depth for all geometry, without incurring fragment shading cost. The color shading pass, which follows the prepass, only shades the fragments where the depth exactly matches. In turn, giving the ideal of one fragment shader invocation per pixel. Mali GPUs include hidden surface removal which does not need a strict front-to-back order, so for most use cases a prepass is a net loss in performance due to the need to render each mesh twice.

During the depth prepass algorithm, the opaque geometry is drawn twice. First, it is drawn as a depth-only update, and then it is drawn as a color render that uses an EQUALS depth test.

Because the full color shader only executes for the visible fragments, it minimizes the amount of redundant fragment processing that occurs. However, depth prepasses double the draw call count and the processed vertex count.

Mali GPUs already include optimizations, such as *Forward Pixel Kill* (FPK), to reduce redundant fragment processing automatically. Therefore, the performance cost of the additional draw calls, vertex shading, and memory bandwidth usually outweighs the benefits.

Behaviors to avoid when draw call batching

Do not use depth prepass algorithms to remove any fragment overdraw.

Negative impacts of using depth prepasses

The impact on performance when using depth prepasses are:

- The CPU incurs a higher load due to duplicated draw calls.
- There is a higher vertex shading and memory bandwidth cost due to geometry that is duplicated.

2.5 OpenGL ES GPU pipelining

OpenGL ES exposes a synchronous rendering model to users of the API, despite using asynchronous execution on the GPU. In comparison, Vulkan exposes this asynchronous nature directly at the API.

Prerequisites

You must understand the following concepts:

- The differences between synchronous and asynchronous execution.
- Pipeline draining.

Keeping the GPU busy, fences, and queries

Whether execution on the GPU is synchronous or asynchronous, the application must keep the GPU busy with work. Therefore, unless the intended target frame rate has been reached, avoid using operations that drain the GPU pipeline and starve the GPU of work. Keeping the GPU busy means that you get the best rendering performance from the platform.

How to optimize OpenGL ES GPU pipelining on Mali GPUs

Try using the following optimization steps:

- Avoid using the API in a manner that causes the GPU to go idle, unless the target performance has been reached.
- Pipeline any use of fences and query objects. Do not wait on the result of the query too early.
- Use `GL_MAP_UNSYNCHRONIZED` to allow the use of `glMapBufferRange()` on a buffer that is referenced by an in-flight draw call without stalling.
- Use asynchronous `glReadPixels()` calls to read data into a pixel buffer object.

Behaviors to avoid when optimizing OpenGL ES GPU pipelining on Mali GPUs

Arm recommends that you:

- Avoid the following operations that enforce the synchronous behavior of OpenGL ES:
  - `glFinish()`
  - Synchronous `glReadPixels()`
  - `glMapBufferRange()` without `GL_MAP_UNSYNCHRONIZED`, while the target buffer is still referenced by an inflight draw call.
- Avoid using `glMapBufferRange()` with either `GL_MAP_INVALIDATE_RANGE` or `GL_MAP_INVALIDATE_BUFFER`. These flags can trigger the creation of a resource ghost on some Mali driver versions.
- Avoid using `glFlush()` to split render passes since the driver automatically flushes as needed.

Negative impacts of inefficient OpenGL ES GPU pipelining on Mali GPUs

The different types of impact you can see are:

- If the pipeline is drained, then the GPU is partially idle during the bubble, resulting in a loss of performance.
- Depending on the interaction with the system DVFS power management logic, there is the potential for performance instability.

Debugging OpenGL ES issues for Mali GPUs

Try the following debugging tips:

- CPU and GPU activity can be monitored using system profilers such as the Arm Streamline profiler. Pipeline drains are visible as periods of busy time oscillating between the CPU and GPU, without the CPU or the GPU being fully utilized.

2.6 Vulkan GPU pipelining

Mali GPUs can run compute or vertex work at the same time as fragment processing from another render pass is running. To ensure high performance, applications must not unnecessarily create bubbles in this pipeline.

Prerequisites

You must understand the following concepts:

- Using command buffers.
- The different shader stages.

Pipeline Bubbles

Note

For tile-based renderers, it is important to overlap the vertex or compute work from one render pass with the fragment work from earlier render passes.

When using Vulkan, the following reasons can lead to pipeline bubbles in applications:

1. Command buffers not submitted often enough

Not submitting command buffers often enough reduces the amount of work in the GPU processing queue. Restricting the possible scheduling opportunities.

2. Data dependency

For example, consider two render passes N and M. Render pass M occurs at a later stage in the pipeline. Data dependency results when N is consumed earlier in the pipeline by M. Data dependency causes a delay during which enough work must be done to hide the latency in the result generation.

In the following image, which is taken from our Streamline profiler, you can see that we are not hitting the targeted 60FPS. Instead, the CPU and the vertex and fragment activity in the GPU all have idle time.

This pattern of work and idle time oscillating between the processors is a good indicator of the presence of API calls or data dependencies that are limiting GPU scheduling opportunities.

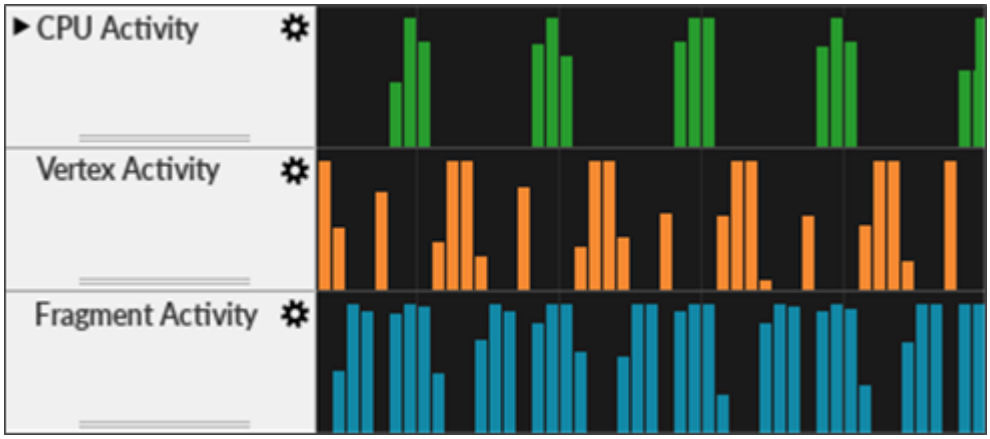

Figure 2-1 Pipeline bubble example

How to prevent pipeline bubbles

To prevent the occurrence of pipeline bubbles:

- Submit command buffers for processing frequently, for example, for each major render pass in a frame.
- If you have a case that causes a bubble, try to see how you can fill that bubble. For example, by inserting independent workloads between the two render passes.

- Consider generating dependent data in an earlier pipeline stage than the stage that consumes it. For example, the compute stage is suited to generate input data for the vertex shading stage. The fragment stage is poorly suited because it occurs later than the vertex shading stage in the pipeline.
- Consider processing dependent data later in the pipeline. For example, fragment shading consuming output from other fragment shading works better than compute shading consuming fragment shading.
- Use fences to asynchronously read back data to the CPU. Do not block synchronously and cause the pipeline to drain.

Actions to avoid while optimizing the GPU pipeline

Arm recommends that you:

- Do not unnecessarily wait for GPU data anywhere in the pipeline.
- Do not wait until the end of the frame to submit all of the render passes.
- Do not create any backwards data dependencies in the pipeline without sufficient intervening work to hide the result generation latency.
- Do not use either `vkQueueWaitIdle()` or `vkDeviceWaitIdle()`.

Debugging your application

The Arm Streamline system profiler visualizes the Arm CPU and GPU activity on both GPU queues. The system profiler quickly shows bubbles in scheduling that occur in two ways:

- Locally to the GPU queues. This type of bubble is indicative of a stage dependency issue.
- Globally across both the CPU and GPU. This type of bubble is indicative of a blocking CPU call that is being used.

2.7 Vulkan pipeline synchronization

Mali GPUs expose two hardware processing slots. Each slot implements a subset of the rendering pipeline stages and runs each slot in parallel with the other slot.

Prerequisites

You must understand the following concepts:

- Command synchronization barriers.
- Shader stages.

Using parallel processing with Vulkan

Note

Tile-based GPUs like Mali differ from desktop intermediate-mode renderers as Tile-based GPUs have two independently scheduled hardware slots present for different types of workload. Tune your pipeline to work well on a tile-based GPU when porting content from a desktop GPU.

The GPU allows the maximum amount of parallel processing across the two hardware slots. The following list shows the mappings of Vulkan stages to the Mali GPU processing slots:

Vertex or compute hardware slot

- `VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT`
- `VK_PIPELINE_STAGE_VERTEX_*_BIT`
- `VK_PIPELINE_STAGE_TESSELLATION_*_BIT`
- `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- `VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT`
- `VK_PIPELINE_STAGE_TRANSFER_BIT`

Fragment hardware slot

- `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`
- `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT`
- `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`
- `VK_PIPELINE_STAGE_TRANSFER_BIT`

Vulkan places the application in control of how dependencies between commands are expressed. An application must make sure that a pipeline stage for one command has produced its results before a later command can consume the results.

The API includes multiple primitives that are available for command synchronization, for example:

- Subpass dependencies, pipeline barriers, and events, which are used to express fine-grained synchronization within a single queue.
- Semaphores, which are used to express heavier weight dependencies across queues.

All fine-grained dependency tools allow the application to specify a restricted scope for their synchronization. The `srcStage` mask indicates which pipeline stages must be waited for. The `dstStage` mask indicates which pipeline stages must wait for synchronization before processing starts.

To get best parallel processing across the two Mali hardware processing slots begin by minimizing the scope for synchronization. Set `srcStage` as early as possible in the pipeline, and set `dstStage` as late as possible

Semaphores allow control when dependent commands run using `pWaitDstStages`. However, semaphores assume that the `srcStage` is the worst case `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`. Therefore, only use semaphores when no fine-grained alternative is available.

Synchronization at low levels

Two kinds of synchronization are needed at the low level:

- Synchronization within a single hardware processing slot.
- Synchronization across the two hardware processing slots.

As fragment shading always comes after vertex or compute in the Mali rendering pipeline, synchronization from a `srcStage` running in the vertex or compute processing slot to a `dstStage` running in the fragment processing slot is low cost.

Synchronization from a `srcStage` in the fragment hardware slot to a `dstStage` in the vertex or compute hardware slot is expensive. The synchronization creates a pipeline bubble unless the extra latency for `srcStage` result generation is accounted for. For example, by having sufficient non-dependent work to fill the bubble

The `TRANSFER` stage is an overloaded term in the Vulkan pipeline. The driver can implement the transfer operations in either hardware processing slot. This means that the direction of a dependency through the pipeline, either backwards or forwards, is not obvious.

Transfers from buffer-to-buffer are implemented in the vertex or compute processing slot. Other transfers can be implemented in either processing slot and determining which hardware processing slot is used depends on the state of the data resource that is being written at the time.

The processing slot that is used materially impacts the rendering workload of an application pipeline. Therefore, always review the performance of transfer operations.

How to optimize your Vulkan pipeline synchronizations

Try using the following optimization steps:

- Set up `srcStageMask` as early as possible in the pipeline.
- Set up `dstStageMask` as late as possible in the pipeline.
- Check whether dependencies point forwards, so source is vertex or compute and destination is fragment, or backwards, so source is fragment and destination is vertex or compute, through the pipeline. Minimize the use of backwards-facing dependencies.
- If backwards-facing dependency is needed, then add sufficient latency between the generation and the consumption of the resource that is used to hide the added scheduling bubble.
- Use `srcStageMask = ALL_GRAPHICS_BIT` and `dstStageMask = FRAGMENT_SHADING_BIT` to synchronize render passes with each other.
- Zero-copy algorithms are the most efficient, so minimize the use of `TRANSFER` copy operations. Always review how `TRANSFER` copies are impacting the hardware pipelining.
- Only use intra-queue barriers when they are needed and put as much work as possible between barriers.

Vulkan pipelining steps to avoid

Arm recommends that you:

- Do not starve the hardware of work.
- Do not forget to overlap vertex or compute with fragment processing.
- Do not use the following `srcStageMask` to `dstStageMask` synchronization pairings because they completely drain the pipeline:
  - `BOTTOM_OF_PIPE_BIT` to `TOP_OF_PIPE_BIT`
  - `ALL_GRAPHICS_BIT` to `ALL_GRAPHICS_BIT`
  - `ALL_COMMANDS_BIT` to `ALL_COMMANDS_BIT`
- Do not use `VkEvent` to signal and wait for that event right away. Use `vkCmdPipelineBarrier()` instead.
- Do not use `VkSemaphore` for dependency management within a single queue.

The negative impact of inefficient Vulkan pipelining

The wrong pipeline barrier can either starve the GPU of work with too much synchronization, or cause rendering corruption with too little synchronization.

Debugging your Vulkan pipeline to identify bubbles

The Arm Streamline system profiler tool quickly shows bubbles in scheduling either locally to the GPU hardware, or globally across both the CPU and GPU. GPU local bubbles are indicative of a stage dependency issue. Global bubbles suggest that a blocking CPU call is being used.

2.8 Pipelined resource updates

OpenGL ES must make sure that resources are in the correct state when the draw call is executed on the GPU. Doing so correctly prevents the modification of resources that are still referenced by in flight draw call.

Prerequisites

You must understand the following concepts:

- Resource ghosting.
- N-buffering.

Referencing resources

Note

Attempting to modify a resource that is still referenced can cause either pipeline bubbles or increased CPU load, depending how the driver handles the conflict.

OpenGL ES presents a synchronous rendering model to the application developer, even though the underlying execution is asynchronous. Rendering must reflect the state of the data resources at the point that the draw call was made. If an application modifies a resource while a pending draw call is still referencing it, then the driver must take evasive action to ensure correctness.

The Mali driver avoids blocking and waiting for the resource reference count to hit zero as doing so drains the pipeline and leads to poor performance. To reflect the new state, Mali GPUs instead create a new version of the resource. The old, or ghost, version of the resource is kept until the pending draw calls have completed and its reference count drops to zero.

This process is expensive and requires at least a memory allocation for the new resource and cleaning up of the ghost resource when it is no longer needed. If the update is not a total replacement, then a copy from the old resource buffer into the new one is also needed.

How to optimize your pipelined resources

Try using the following optimization steps:

- To prevent modifying resources that are referenced by the queued draw calls, use N-buffered resources, and pipeline your dynamic resource updates.
- Use `GL_MAP_UNSYNCHRONIZED` to allow the use of `glMapBufferRange()` to patch an unreferenced region of a buffer that is still referenced by in-flight draw calls.

Approaches to avoid when updating pipelined resources

Arm recommends that you:

- Do not modify resources that are still being referenced by in-flight draw calls.
- Do not use `glMapBufferRange()` with either `GL_MAP_INVALIDATE_RANGE` or `GL_MAP_INVALIDATE_BUFFER` on some older Mali driver versions, as these flags trigger the creation of an unnecessary resource ghost.

Negative impacts of unoptimized pipelined resource updates

Suboptimal pipelined resource updates can result in the following negative impacts:

- Resource ghosting increases CPU load due to memory allocation overheads and the need for copies to build new versions of resources.
- The constant allocation and freeing of the ghosts can lead to an instability in the memory footprint.

Debugging resource updates

The Arm Streamline system profiler visualizes the Arm CPU and GPU activity. Failing to pipeline resource updates normally show as elevated CPU activity.

Chapter 3
CPU Overheads

There are various ways to reduce CPU overheads to increase efficiency and reduce software processing costs. To reduce CPU overheads, consider using: shader compilation, pipeline creation, memory allocation, memory mapping, command pools, command buffers, descriptor sets, and layouts.

It contains the following sections:

3.1 Compiling shaders in OpenGL ES

Both shader compilation and program linkage are expensive operations. Generating new programs while trying to render at high frame rates, can cause dropped frames. Therefore, avoid shader compilation and program linkage in the interactive part of your application.

Prerequisites

You must understand the following concepts:

- Compiling shaders.
- Linking programs.

How to optimize compiled shaders

When either starting an activity, or when loading a level of a game, compile all shaders and link all programs.

Behaviors to avoid when compiling shaders

Arm recommends that you:

- Do not attempt to compile shaders during interactive gameplay.
- Do not rely on the Android blob cache for interactive compile and link performance as first use is poor as shaders are not in the cache. It is also often too small to contain all shader programs for a complex application and is unable to cache all the shaders.

The negative impact of not compiling shaders correctly

The cost of trying to compile, and link, shaders during the interactive portions of the application is a high CPU load and dropped frames.

3.2 Pipeline creation in Vulkan

Vulkan pipelines have similar compile requirements to OpenGL ES shaders. In addition, Vulkan provides no automatic caching that is equivalent to the Android blob cache that is available for OpenGL ES programs. You are responsible for providing persistent storage of compiled programs for use across program invocations.

Prerequisites

You must understand the following concepts:

- Vulkan pipelines.

How to optimize pipeline creation

Try using the following optimization steps:

- Create pipelines when starting an activity or loading a game level.
- Speed up pipeline creation by using a pipeline cache.
- Serialize the pipeline cache to disk and then reload it the next time the application is used, providing end users with a faster load time.

Avoiding suboptimal use

Do not create derivative pipelines using `VK_PIPELINE_CREATE_DERIVATIVE_BIT`.

Negative impacts on the application

Keep in mind the following impacts on your application:

- Attempting to create pipelines during the interactive portions of your application increases the CPU load and can cause skipped frames.
- Failure to serialize, and then reload, a pipeline cache increases load times on later application runs.

3.3 Allocating memory in Vulkan

For frequent allocations, do not use the `vkAllocateMemory()` allocator. All allocations by `vkAllocateMemory()` are expensive kernel calls.

Prerequisites

You must understand the following concepts:

- Memory allocation.

How to optimize memory allocation

When allocating memory, use your own allocator to manage block allocations.

Something to avoid when allocating memory

Do not use `vkAllocateMemory()` as a general-purpose allocator.

Negative impacts of suboptimal memory allocation

Using `vkAllocateMemory()` as a general-purpose allocator increases the load on the application CPU load.

Debugging memory allocation issues

Monitor the frequency and allocation size of all calls to `vkAllocateMemory()` at runtime.

3.4 OpenGL ES CPU memory mapping

By using `glMapBufferRange`, OpenGL ES provides direct access to buffer objects that are mapped into the application address space. It is efficient to stream writes, but reads are expensive for mapped buffers because they are uncached on the CPU.

Prerequisites

You must understand the following concepts:

- Buffers.
- Memory mapping.

How to optimize CPU memory mapping

To benefit from store merging in the CPU write buffer, make write-only buffer updates to sequential addresses.

Outcome to avoid when using CPU memory mapping

Arm recommends that you do not read values from mapped buffers.

Negative impacts of not using CPU memory mapping correctly

Reading from uncached buffers shows up as increased CPU load in the application functions that read them.

3.5 Vulkan CPU memory-mapping

Vulkan gives applications support for multiple buffer memory types.

Prerequisites

You must understand the following concepts:

- CPU and GPU memory mapping.
- Cached versus uncached memory.

Memory Types

On Midgard architecture GPUs, the Mali GPU driver exposes three memory types:

1. `DEVICE_LOCAL_BIT` | `HOST_VISIBLE_BIT` | `HOST_COHERENT_BIT`
2. `DEVICE_LOCAL_BIT` | `HOST_VISIBLE_BIT` | `HOST_CACHED_BIT`
3. `DEVICE_LOCAL_BIT` | `LAZILY_ALLOCATED_BIT`

On Bifrost and later architecture GPUs, the Mali GPU driver exposes four memory types:

1. `DEVICE_LOCAL_BIT` | `HOST_VISIBLE_BIT` | `HOST_COHERENT_BIT`
2. `DEVICE_LOCAL_BIT` | `HOST_VISIBLE_BIT` | `HOST_CACHED_BIT`
3. `DEVICE_LOCAL_BIT` | `HOST_VISIBLE_BIT` | `HOST_COHERENT_BIT` | `HOST_CACHED_BIT`
4. `DEVICE_LOCAL_BIT` | `LAZILY_ALLOCATED_BIT`

The four memory types and their purposes

The purposes that these memory types are useful for are:

Not cached, coherent

The `HOST_VISIBLE | HOST_COHERENT` memory type:

- Is guaranteed to be supported.
- Provides uncached storage on the CPU.
- Avoids polluting the CPU caches with unnecessary data.
- Efficiently merges small writes that are then sent on to the external memory device by using the CPU write buffer hardware.
- It is the optimal memory type for CPU write-only resources.

Cached, incoherent

The `HOST_VISIBLE | HOST_CACHED` memory type:

- Provides cached storage on the CPU but does not guarantee that the CPU and the GPU get coherent views of the underlying memory. The CPU view of the memory is not coherent with the GPU view of the memory, therefore you must call:
  - `vkFlushMappedRanges()` when the CPU has finished writing data for the GPU.
  - `vkInvalidateMappedRanges()` when reading back the data that the GPU has written.
  - However, both calls are expensive to use, so must be used sparingly.
- Must only be used for resources that are both mapped and read by the application software on the CPU.

Cached, coherent

The `HOST_VISIBLE | HOST_COHERENT | HOST_CACHED` memory type:

- Provides cached storage on the CPU, which is also coherent with the GPU view of the memory without needing manual synchronization.
- Supported by Mali Bifrost and later GPUs if the chipset supports the hardware coherency protocol between the CPU and GPU.
- Due to hardware coherency, it avoids the overheads of manual synchronization operations. Cached, coherent memory is preferred over the `Cached, incoherent` memory type when available.
- Must be used for resources that are both mapped and read by the application software on the CPU.
- Hardware coherency has a small power cost, so must not be used for resources that are write-only on the CPU. For write-only resources, bypass the CPU cache by using the `Cached, incoherent` memory type.

Lazily allocated

The `LAZILY_ALLOCATED` memory type:

- It is a special memory type that is initially only backed by GPU virtual address space and not physical memory pages. Physical backing pages are allocated on demand if the memory is accessed.
- It must be used alongside transient image attachments created using `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT`. Transient images are intended for use as framebuffer attachments which only exist during a single render pass. Doing so avoids backing the attachment with physical memory. You can do this by backing the attachment image with a lazy allocation, and a `VK_ATTACHMENT_STORE_OP_DONT_CARE` storage operation. Common use cases for this include depth or stencil buffers for simple renders. Another use case and G-buffer attachments, that are used for deferred lighting, that are then consumed by a later subpass and are not written back to memory.
- Must not be used to provide storage for resources that are expected to be written back to memory.

How to optimize Vulkan CPU memory mapping

Try using the following optimization steps:

- Use `HOST_VISIBLE | HOST_COHERENT` for immutable resources.
- Use `HOST_VISIBLE | HOST_COHERENT` for resources which are write-only on the CPU.
- Write updates to `HOST_VISIBLE | HOST_COHERENT` memory use `memcpy()` or write sequentially to get best efficiency from the CPU write-combine unit.
- Use `HOST_VISIBLE | HOST_COHERENT | HOST_CACHED` for resources which are read back on to the CPU. Use `HOST_VISIBLE | HOST_CACHED` if it is not available.
- Use `LAZILY_ALLOCATED` for transient framebuffer attachments which only exist during a single render pass.
- Only use `LAZILY_ALLOCATED` memory for `TRANSIENT_ATTACHMENT` framebuffer attachments.
- Mapping and unmapping buffers have a CPU cost. Therefore, persistently map buffers which are accessed often. Example: uniform buffers, data buffers, or dynamic vertex data buffers.

Vulkan CPU memory-mapping steps to avoid

Arm recommends that you:

- Do not read back data from uncached memory on the CPU.
- Do not store suballocator metadata, which must be read on the CPU, inside an uncached memory buffer.

Negative impacts of inefficient Vulkan CPU memory mapping

Uncached readbacks can be much slower than cached reads due to increased CPU processing costs.

Debugging Vulkan CPU memory-mapping performance problems

There are a couple of steps you can take:

- Check that all CPU-read buffers are using cached memory.
- Design interfaces for buffers to encourage implicit flushes or invalidates as needed. It is difficult and time consuming to debug coherency failures due to missing maintenance operations without this infrastructure already in place.

3.6 Command pools for Vulkan

Command pools do not automatically recycle memory from deleted command buffers unless created with the `RESET_COMMAND_BUFFER_BIT` flag. Pools without this flag do not recycle their memory until the application resets the pool.

Prerequisites

You must understand the following concepts:

- Command pools.

How to optimize command pools

Periodically call `vkResetCommandPool()` to release the memory.

Command pool step to avoid

Read *3.7 Optimizing command buffers for Vulkan* on page 3-36 for why using `RESET_COMMAND_BUFFER_BIT` is not recommended. However, using `RESET_COMMAND_BUFFER_BIT` is better than not releasing memory.

The negative impact of inefficient Vulkan command pools

An increase in memory usage until a manual command pool reset is triggered.

3.7 Optimizing command buffers for Vulkan

Command buffer usage flags affect performance.

Prerequisites

You must understand the following concepts:

- Command buffers.
- Command pools.

How to optimize command buffers

Try using the following optimization steps:

- For best performance set the `ONE_TIME_SUBMIT_BIT` flag.
- Build per-frame command buffers instead of using simultaneous command buffers.
- If the alternative is to replay the same command sequence every time in application logic, then use `SIMULTANEOUS_USE_BIT`. It is more efficient than an application replaying commands manually, but less efficient than a one-time submit buffer.

Command buffer steps to avoid

Arm recommends that you:

- Do not set `SIMULTANEOUS_USE_BIT`, unless required to do so.
- Do not use command pools with `RESET_COMMAND_BUFFER_BIT` set. Doing so increases the memory management overhead, as it prohibits the driver from using a single large allocator for all command buffers in a pool.

Negative impacts of inefficient Vulkan command buffers

Keep the following points in mind:

- There is a risk of an increase in CPU load if inappropriate flags are used, or if command buffer resets are too frequent.
- Avoid calling `vkResetCommandBuffer()` on a high frequency call path.

Debugging your Vulkan command buffer performance problems

There are a couple of steps you can take, to speed up your debugging process. These two steps are:

- Review and evaluate the performance impact every use of any command buffer flag other than `ONE_TIME_SUBMIT_BIT`.
- Evaluate every use of `vkResetCommandBuffer()` and assess if it could be replaced with `vkResetCommandPool()` instead.

3.8 Secondary command buffers

The current Mali hardware does not have native support for invoking commands in a secondary command buffer. Therefore, there is extra overhead that is incurred when using secondary command buffers.

Prerequisites

You must understand the following concepts:

- Command buffers.
- Command pools.

Secondary command buffers

It is expected that applications must use secondary command buffers, typically to allow multi-threaded command buffer construction. However, the total number of secondary command buffer invocations must be minimized. As with primary command buffers, we recommend avoiding creating command buffers with the `SIMULTANEOUS_USE_BIT` due to increased overheads.

How to optimize secondary command buffers

Try using the following optimization steps:

- Use secondary command buffers to allow multi-threaded render pass construction.
- Minimize the number of secondary command buffer invocations that are used per frame.

Secondary command buffer step to avoid

Do not set `SIMULTANEOUS_USE_BIT` on secondary command buffers.

The negative impact of inefficient secondary command buffers

Keep in mind that an increased CPU load is incurred.

3.9 Optimizing descriptor sets and layouts for Vulkan

Midgard and Bifrost family Mali GPUs support four simultaneous bound descriptor sets at the API level. However, they require a single physical descriptor table per draw call internally.

Prerequisites

You must understand the following concepts:

- Descriptor sets.
- Binding spaces.

Descriptor sets and layouts

If any of the four source descriptor sets have changed, then the driver rebuilds the internal table for a draw call.

The first draw call, after a descriptor changes, has a higher CPU overhead than following draw calls that reuse the same descriptor set. Larger descriptor sets cause a more expensive rebuild.

With current drivers, the descriptor set pool allocations are not pooled. Do not call `vkAllocateDescriptorSets()` on a performance critical code path.

How to optimize descriptor sets and layouts

Try using the following optimization steps:

- Pack the descriptor set binding space as much as possible.
- Instead of resetting descriptor pools and reallocating new descriptor sets, update descriptor sets that are already allocated, but no longer referenced.
- Reuse pre-allocated descriptor sets and do not update them with the same information every time.
- Use `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` or `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` to bind the same UBO or SSBO with different offsets. The alternative is building more descriptor sets.

Descriptor set and layout steps to avoid

Arm recommends that you:

- Do not leave holes in the descriptor set.
- Do not leave unused entries as copying and merging still has a computational cost.
- Do not allocate descriptor sets from descriptor pools on performance critical code paths.
- Do not use `DYNAMIC_OFFSET UBOs/SSBOs` if you never plan on changing the binding offset, as there is a small, extra, cost for handling the dynamic offset.

The negative impact of inefficient descriptor sets and layout

Unoptimized Vulkan descriptor sets and layout leads to a risk of increased CPU load for draw calls.

Debugging your descriptor sets and layout performance problems

Ways to speed up your debugging process:

- Monitor the pipeline layout for unused entries.
- Monitor for contention-related performance problems on `vkAllocateDescriptorSets()`.

Chapter 4
Vertex shading

The efficiency of vertex processing, in terms of both buffer packing and vertex shading, is important when rendering a scene. Avoid poor mesh selection and inefficient vertex data encoding on mobile devices, as it can significantly increase DRAM bandwidth.

It contains the following sections:

4.1 Index draw calls

Indexed draw calls allow for reuse of vertices and are more efficient than non-indexed draw calls.

Prerequisites

You must understand the following concepts:

- Draw calls.
- Index buffers.

How to optimize index draw calls

Try using the following optimization steps:

- Use indexed draw calls whenever vertex reuse is possible.
- Optimize index locality for a post-transform cache.
- To minimize redundant processing and data fetch, ensure that the index buffer references the entire range of index values.
- Create tightly packed vertex ranges for lower level of detail meshes.
- Use `glDrawRangeElements()` for volatile resources.
- To implement geometry *Level of Detail* (LOD), create a contiguous set of vertices for each detail level.

Index draw call steps to avoid

Arm recommends that you:

- Do not use client-side index buffers.
- Do not modify the contents of index buffers. Doing so causes the driver to rescan the index buffer to determine the active index range.
- Do not use indices that sparsely access vertices in the vertex buffer.
- Do not use indexed draws for simple geometry which has no vertex reuse, such as simple quads or point lists.
- Do not create a low detail mesh by sparsely sampling vertices from the full detail mesh.

Negative impacts of inefficient index draw calls

Unoptimized index draw calls can lead to the following problems:

- The use of client-side index buffers increases the CPU load. Client-side index buffers first allocate server-side buffer storage, then copies the data, and finally, scans the contents to determine the active index ranges.
- The use of index buffers with frequently modified contents shows up as increased load on the CPU. The increased load is due to the need to rescan the buffer to determine the active index ranges.
- Inefficient mesh indexing that is due to index sparseness, or poor spatial locality. This shows up as extra GPU vertex processing time or extra memory bandwidth that is being used. The severity of the impact depends on the complexity of the mesh and the layout of the index buffer that is in memory.

Debugging index draw calls issues

Scan through all index buffers before submission. Optimize any buffers that include any unused indices.

4.2 Index buffer encoding

The index buffer data is one of the primary data sources into the primitive assembly and tiling process in Mali GPUs. To minimize the cost of tiling, efficiently pack the index buffer.

Prerequisites

You must understand the following concepts:

- Index buffers.
- Strip formats and simple list formats.

How to optimize index buffer encodes

Try using the following optimization steps:

- Use the lowest precision index data type possible to reduce index list size.
- To reduce index list size, use strip formats over simple list formats.
- To reduce index list size, use primitive restart instead of degenerate triangles.
- For a post-transform cache, optimize the index locally.

Index buffer encoding steps to avoid

Arm recommends that you:

- Do not use 32-bit index values.
- Do not use index buffers low spatial coherency because they hurt caching.

The negative impact of inefficient index buffer encoding

Inefficient index buffer encoding does not usually have a significant impact, but can be part of small improvements that add up to improve application performance. A significant negative impact is possible if draw calls are small, and vertex shading and tiling do not pipeline cleanly.

4.3 Attribute precision

Full FP32 `highp` precision of vertex attributes is unnecessary for many uses of attribute data. For example, color computation. Asset pipelines must keep the data at the minimum precision that is required. Doing so reduces bandwidth and improves performance.

Prerequisites

You must understand the following concepts:

- Attribute data.
- Asset pipelines.
- FP32 and FP16.

Attribute precision

OpenGL ES and Vulkan provide different attribute data types that fit every precision that is needed. Lower precision attributes are available in 8-bit, 16-bit, and packed formats such as `RGB10_A2`. Mali GPU hardware can convert attributes to FP16 and FP32 for free on data load. Therefore, there is no shader processing overhead from using narrower data types.

How to optimize attribute precision

Try using the following optimization steps:

- Stable geometry positioning needs extra precision, so use FP32 for computing vertex positions.
- Use low precision for other attributes, increasing to high precision only when it is needed.

Attribute precision steps to avoid

Arm recommends that you:

- Do not use FP32 for everything.
- Do not upload FP32 data into a buffer and then read it as a `mediump` attribute. Doing so wastes both memory storage and bandwidth as the extra precision is discarded.

Negative impacts of inefficient attribute precision

Higher memory bandwidth, large memory footprint, energy inefficiency, and reduced vertex shader performance.

4.4 Attribute layout

For Bifrost and above architecture, vertices are shaded using an *Index Driven Vertex Shading* (IDVS) flow.

Prerequisites

You must understand the following concepts:

- Vertices.
- Vertex buffers.

IDVS vertex shading order

The order in which vertices are shaded using the IDVS are as follows:

1. Positions.
2. Varyings of vertices of primitives which have survived culling.

Good buffer layout maximizes the benefit of this geometry pipeline.

How to optimize the attribute layout

Try using the following optimization steps:

- Use a dedicated, tightly packed, vertex buffer for position data.
- Use a second dedicated vertex buffer for non-position data, if any exists.
- Remove unused attributes for specific uses to optimize the meshes. For example: Generating a tightly packed buffer consisting of only position-related attributes for shadow mapping.

Attribute layout steps to avoid

Arm recommends that you:

- Do not use one buffer per attribute as this wastes bandwidth by fetching data for culled vertices when they share a cache line with visible vertices.
- Do not pad interleaved vertex buffer strides up to the power-of-two. Mali hardware does not require it, and doing so increases memory bandwidth.

The negative impacts of an inefficient attribute layout

Here are two potential problems that your application can experience:

- Increased memory bandwidth due to redundant data fetches.
- Increased pressure on caches which cause loss of performance.

4.5 Varying precision

Vertex shader outputs are commonly called varying outputs. They are written back to main memory before fragment shading commences.

Prerequisites

You must understand the following concepts:

- mediump.
- highp.

Vertex shader outputs

It is important to minimize the precision of varying outputs. Doing so reduces the amount of memory storage and bandwidth that is needed for the varying output data.

Typical uses of varying data that usually have sufficient precision in `mediump` include:

- Normals.
- Vertex color.
- Texture coordinates for non-tiled textures that go up to 512x512 pixels.

Typical uses of varying data that need `highp` precision are:

- World-space positions.
- Texture coordinates for large textures, or textures with high levels of UV coordinate wrapping.

How to optimize varying outputs

Use the `mediump` qualifier on varying outputs if the precision is acceptable.

Varying output precision steps to avoid

Arm recommends that you:

- Do not use varyings with more components and precision than is needed.
- Do not use vertex shaders with outputs that are left unused in the fragment shader.

The negative impacts of inefficient varying outputs

The different types of impact you can see are:

- Increased GPU memory bandwidth.
- Reduced vertex shader and fragment shader performance.

4.6 Triangle density

A vertex requires more bandwidth and processing power to process than a fragment. Ensure that there are many pixels worth of fragment work for each primitive that is rendered. Doing so spreads the processing expense of the vertices over multiple pixels of output.

Prerequisites

You must understand the following concepts:

- Bandwidth and processing vertex costs.
- Mesh *Levels of Detail* (LOD).

How to optimize triangle density

Try using the following optimization steps:

- Use models that create at least 10-20 fragments per primitive.
- Use dynamic mesh level-of-detail, using simpler meshes when objects are further away from the camera.
- Use techniques such as normal mapping to bake the required complex geometry during asset creation into a simpler run-time mesh with a supplemental texture for per-pixel lighting.
- To improve final image quality, favor improved lighting effects and textures instead of increased geometry.

Vertex processing steps to avoid

Do not generate micro triangles, as they increase bandwidth and processing power costs for little visual benefit.

The negative impacts of inefficient vertex processing

If the triangle density of a mesh is not properly optimized, then high volumes of geometry cause many problems for a tile-based renderer. For example: poor shading performance, low memory bandwidth, and high system energy consumption due to the memory traffic.

4.7 Instanced vertex buffers

Both OpenGL ES and Vulkan have support for instanced drawing. Instanced drawing uses attribute instance divisors to divide a buffer to locate the data for each instance. There are some hardware limitations which determine the optimal usage of instanced vertex buffers.

Prerequisites

You must understand the following concepts:

- Instance divisors.
- Instanced vertex buffers.

How to optimize the use of vertex buffers

Try using the following optimization steps:

- Use a single interleaved vertex buffer for all instance data.
- Use instanced attributes to work around limitation of 16KB uniform buffers.
- Use a number of vertices per instance that are a power-of-two.
- Prefer indexed lookups using `gl_InstanceID` into uniform buffers or shader storage buffers. Rather than per-instance attribute data.

Instanced vertex buffer steps to avoid

Do not use more than one instanced vertex buffer per draw call.

The negative impacts of an inefficient instanced vertex buffer

If the vertex buffer is not properly optimized, then you can expect a reduced performance on all impacted instanced draw calls.

Chapter 5
Tessellation, geometry shading, and tiling

This chapter covers ways on how to optimize tessellation, geometry shading, and tiling instances in your application.

It contains the following sections:

5.1 Tessellation

Tessellation is a powerful brute-force technique that creates higher-quality meshes, at the cost of GPU memory bandwidth and power consumption. Therefore, tessellation can become expensive to run on tile-based GPUs, such as Mali, that writes the output of the geometry processing back to system memory.

Prerequisites

You must understand the following concepts:

- Tessellation.
- Using and optimizing mesh *Levels of Details* (LODs).
- How to monitor GPU memory bandwidth and power consumption.

How to optimize tessellation

Try using the following optimization steps:

- Add more static mesh levels-of-detail to character meshes.
- Use the geo-mipmap morphing technique for terrain meshes.

However, if you must use tessellation:

- Only add geometry where it benefits most. For example, on the silhouette edges of your meshes.
- Tessellation can also be used to reduce geometry complexity.
- Cull patches to avoid redundant evaluation shader invocations. In the control shader, use either the `PrimBB` extension, or frustum cull patches.
- To ensure that there is a sensible upper-bound on complexity, clamp your maximum tessellation factor.
- Pre-tessellate a new static mesh instead of using fixed tessellation factors.

Steps to avoid when using tessellation

Arm recommends that you:

- Do not use tessellation until you have evaluated other options.
- Do not use tessellation together with geometry shaders in the same draw call.
- Do not use transform feedback with tessellation.
- Do not use microtriangulation. There is little perceptual quality benefit from triangle densities of less than ten fragments per primitive.

Negative impacts of unnecessarily implementing tessellation

Using tessellation to significantly increase triangle density leads to poor performance, high memory bandwidth, and increased system power consumption.

Debugging tessellation issues

Try the following debugging tips:

- It is easy to overlook that tessellation is generating millions of triangles because it is procedurally generated and is hidden from the application. Therefore, always check the GPU performance counters to monitor the number of triangles that are being generated and the number of useless microtriangles that are being generated.
- To perform a controlled exploration of the performance versus visual benefit trade-offs, apply different levels of `min()` to the tessellation factors in the control shader.

5.2 Geometry shading

Before using geometry shading, keep in mind that tile-based GPU architecture is sensitive to geometry bandwidth levels.

Prerequisites

You must understand the following concepts:

- Geometry shading.

How to optimize geometry shading

Use compute shaders instead of geometry shading, as compute shaders are more flexible and avoid the unnecessary duplication of vertices that most geometry shaders trigger.

Steps to avoid when using geometry shading

Arm recommends that you:

- Do not use geometry shaders to filter primitives, such as triangles, or to pass down more attributes to the fragment stage.
- Do not use geometry shaders to expand points to quads. Instance a quad instead.
- Do not use transform feedback with geometry shaders.
- Do not use primitive restart with geometry shading.
- Do not use geometry shaders.

Negative impacts of unnecessarily implementing geometry shading

Using geometry shading leads to increased memory bandwidth. Therefore, creating an indirect negative effect of reduced performance and energy efficiency.

5.3 Tiling and effective triangulation

Tiling and rasterization both work on fragment patches that are larger than a single pixel. For Mali GPUs, the tiling uses bins that are at least 16x16 pixels. Fragment rasterization emits 2x2 pixel quads for fragment shading.

Prerequisites

You must understand the following concepts:

- Tiling and rasterization techniques.
- Fragment quads.

Minimizing number of triangles needed

Optimal performance is achieved when mesh triangulation uses the minimum number of triangles to cover the necessary pixels.

How to optimize tiling and effective triangulation

Use triangles that are almost equilateral. Using equilateral triangles maximizes the ratio of the area to edge length. Doing so reduces the number of generated partial fragment quads.

Tiling and rasterization steps to avoid

The center point of a triangle fan has a high triangle density for low pixel coverage per triangle loaded. Therefore, avoid the use of fans or similar geometry layouts.

Negative impacts of inefficient tiling and rasterization

There is a risk of increasing the fragment shading overhead due to the triangle fetch and partial sample coverage of the 2x2 pixel fragment quads.

Debugging triangulation issues

Use the Arm Graphics Analyzer. The Graphics Analyzer contains mesh visualization tools that allow the outline of the mesh, submitted to a draw call, to be visualized in object-space. *https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/components/graphics-analyzer*

Chapter 6
Fragment shading

This chapter provides multiple areas of guidance on how to optimize the fragment shading elements of your application on Mali GPUs.

It contains the following sections:

6.1 Efficient render passes with OpenGL ES

Tile-based rendering operates on the concept of render passes. Each render pass has an explicit start and end and produces an output in memory only at the end of the pass.

Prerequisites

You must understand the following concepts:

- OpenGL ES rendering APIs.
- Render passes.
- Tile-based rendering.

Render pass handling

At the start of the pass, the tile memory is initialized inside the GPU. At the end of the pass, the required outputs are written back to system memory. The intermediate framebuffer working state lives entirely inside the tile memory.

Efficient render passes

The driver infers the OpenGL ES render passes based on framebuffer binding calls, as they are not explicit in the API. A render pass for the framebuffer starts when it is bound as the `GL_DRAW_FRAMEBUFFER` target and normally ends when the draw framebuffer binding changes to another framebuffer.

How to optimize render passes

Try using the following optimization steps:

- When starting a render pass, clear or invalidate every attachment. This does not apply if the content of a render target is used as the starting point for rendering.
- To clear the tile memory quickly, clear the entire content of the attachment, ensuring that the color, depth, or stencil writes are not masked.
- At the end of the render pass, invalidate any attachments that are not needed outside of the pass, before changing the framebuffer binding to the next FBO.
- For rendering to a subregion of framebuffer, use a scissor box to restrict the area of clearing and rendering required.

Render pass steps to avoid

Arm recommends that you:

- Do not switch back and render to the same FBO multiple times in a frame. Complete each of your render passes in a single `glBindFramebuffer()` call before moving on to the next.
- Do not split a render pass by using either `glFlush()` or `glFinish()`.
- Do not create a packed depth-stencil texture, `D24_S8` or `D32F_S8`, and only attach one of the two components as an attachment.

Negative impacts of inefficient render passes

Incorrect handling of render passes causes worse fragment shading performance and increased memory bandwidth. Therefore, avoid lowering fragment shading performance and increasing memory bandwidth. At the start of rendering, read non-cleared attachments into tile memory, and then write out non-invalidated attachments at the end of rendering.

Debugging render pass issues

Review your API usage of framebuffer binding, clears, draws, and invalidates.

6.2 Efficient render passes with Vulkan

Tile-based rendering operates on the concept of render passes. Each render pass has an explicit start and end and produces an output in memory only at the end of the pass.

Prerequisites

You must understand the following concepts:

- Vulkan rendering APIs.
- Render passes.
- Tile-based rendering.

Render pass handling

At the start of the pass, the tile memory is initialized inside the GPU. At the end of the pass, the required outputs are written back to system memory. The intermediate framebuffer working state lives entirely inside the tile memory.

Efficient render passes

Unlike with OpenGL ES, Vulkan render passes are explicit in the API. There are defined `loadOp` and `storeOp` operations. `loadOp` defines how GPUs initialize the tile memory at the start of the pass. `storeOp` defines what is written back at the end of a pass.

Vulkan introduces lazily allocated memory, meaning that transient attachments existent during a single render pass do not need physical storage.

How to optimize render passes

Try using the following optimization steps:

- Clear or invalidate each attachment at the start of a render pass using `loadOp = LOAD_OP_CLEAR` or `loadOp = LOAD_OP_DONT_CARE`.
- Set up any attachment that is only live during a single render pass as a TRANSIENT_ATTACHMENT that is backed by LAZILY_ALLOCATED memory.
- Ensure that the contents are invalidated at the end of the render pass using `storeOp = STORE_OP_DONT_CARE`.

Render pass steps to avoid

Arm recommends that you:

- Do not clear an attachment inside a render pass using `vkCmdClearAttachments()`. This is not free, unlike a clear or invalidate `loadOp` operation.
- Do not write a constant color using a shader program to manually clear a render pass.
- Do not use `loadOp = LOAD_OP_LOAD` unless your algorithm relies on the initial framebuffer state.
- Do not set `loadOp` or `storeOp` for attachments that are not needed in the render pass to avoid generating a needless round trip through the tile-memory for that attachment.
- Do not use `vkCmdBlitImage` as a way of upscaling a low-resolution game frame to native resolution. Especially if you render the UI or HUD directly on top of the frame with `loadOp = LOAD_OP_LOAD`, as this is an unnecessary round trip to memory.

Negative impacts of inefficient render passes

Incorrect handling of render passes causes worse fragment shading performance and increased memory bandwidth.

Debugging render pass issues

Review the API usage of render pass creation and any use of `vkCmdClearColorImage()`, `vkCmdClearDepthStencilImage()`, and `vkCmdClearAttachments()`.

6.3 Multisampling for OpenGL ES

For most multisampling, all of the data for the additional samples are kept in the tile memory, which is inside the GPU. This data is resolved to a single pixel color as part of the tile writeback. This is efficient because the bandwidth for the additional samples never enters the external main memory.

Prerequisites

You must understand the following concepts:

- Anti-aliasing.
- OpenGL ES APIs.
- The `EXT_multisampled_render_to_texture` extension - *https://www.khronos.org/registry/gles/extensions/EXT/EXT_multisampled_render_to_texture.txt*

Optimal multisampling performance

To get optimal render-to-texture multisampling performance, use the `EXT_multisampled_render_to_texture` extension. This extension can render multisampled data directly into a single-sampled image in memory, without needing a second pass.

How to optimize the use of MSAA

Try using the following optimization steps:

- Use 4x MSAA because it is not expensive and provides good image quality improvements.
- Use the `EXT_multisampled_render_to_texture` extension render-to-texture multisampling.

MSAA steps to avoid

Arm recommends that you:

- Do not use `glBlitFramebuffer()` to implement a multisample resolve.
- Do not use more than 4x MSAA without checking performance.

Negative impacts of implementing MSAA incorrectly

Failing to resolve multisampling inline results in higher memory bandwidth and reduced performance. For example, manually writing and resolving a 4xMSAA 1080p surface at 60 FPS requires 3.9GB/s of memory bandwidth. This is compared to 500MB/s when using the extension.

Debugging MSAA issues more effectively

Review any use of `glBlitFramebuffer()`.

6.4 Multisampling for Vulkan

For most multisampling, all of the data for the additional samples are kept in the tile memory, which is inside the GPU. This data is resolved to a value of a single pixel color as part of the tile writeback. This is efficient because the bandwidth for the additional samples never enters the external main memory.

Prerequisites

You must understand the following concepts:

- Anti-aliasing.
- Vulkan APIs.

Optimal multisampling performance for Vulkan

Multisampling is fully integrated with Vulkan render passes, which allows the multisample resolve to be explicitly specified at the end of the subpass using the end of pass `resolveOp`.

How to optimize the use of MSAA with Vulkan

Try using the following optimization steps:

- Use 4x MSAA as it is not expensive and provides good image quality improvements.
- Use `loadOp = LOAD_OP_CLEAR` or `loadOp = LOAD_OP_DONT_CARE` for the multisampled image.
- Use `pResolveAttachments` in a subpass to automatically resolve a multisampled color buffer into a single-sampled color buffer.
- Use `storeOp = STORE_OP_DONT_CARE` for the multisampled image.
- Use LAZILY_ALLOCATED memory to back the allocated multisampled images. No physical backing storage is required as they do not need to be stored in the main memory.

Vulkan MSAA steps to avoid

Arm recommends that you:

- Do not use `vkCmdResolveImage()`. Bandwidth and performance is negatively impacted.
- Do not use `storeOp = STORE_OP_STORE` for multisampled image attachments.
- Do not use `storeOp = LOAD_OP_LOAD` for multisampled image attachments.
- Do not use more than 4x MSAA without checking performance.

The negative impact of implementing MSAA with Vulkan incorrectly

Failing to resolve multisampling inline results in higher memory bandwidth and reduced performance. For example, manually writing and resolving a 4xMSAA 1080p surface at 60 FPS requires 3.9GB/s of memory bandwidth. This is compared to 500MB/s when using an inline resolve.

6.5 Multipass rendering

Multipass rendering is an important feature of Vulkan which enables applications to exploit the full power of tile-based architectures using the standard API.

Prerequisites

You must understand the following concepts:

- Anti-aliasing.
- Vulkan APIs.
- Using late-zs testing.
- Render passes and subpasses.

Enabling powerful algorithms

Mali GPUs can take color attachments and depth attachments from one subpass, and use them as input attachments in a later subpass without going via main memory. This enables powerful algorithms, such as deferred shading or programmable blending, to be used efficiently. However, a few things must be set up correctly.

Per-pixel storage requirements

Most of the Mali GPUs are designed for rendering 16x16 pixel tiles with 128 bit per pixel of tile buffer color storage. Some GPUs, such as Mali-G72, increase this count to up to 256 bits per pixel.

G-buffers, which require more color storage, can be used at the expense of requiring smaller tiles during fragment shading, which can reduce performance.

For example, a sensible G-buffer layout that fits neatly into a 128-bit budget could be:

- Light: `B10G11R11_UFLOAT`
- Albedo: `RGBA8_UNORM`
- Normal: `RGB10A2_UNORM`
- PBR material parameters/misc: `RGBA8_UNORM`

Image layouts

Multipass rendering is one of the few cases where image layout matters because it impacts the optimizations which the driver enables.

Here is a sample multipass layout which hits all the good paths:

Initial layouts

- Light: `UNDEFINED`
- Albedo: `UNDEFINED`
- Normal: `UNDEFINED`
- PBR: `UNDEFINED`
- Depth: `UNDEFINED`

G-buffer pass (subpass #0) output attachments

- Light: `COLOR_ATTACHMENT_OPTIMAL`
- Albedo: `COLOR_ATTACHMENT_OPTIMAL`
- Normal: `COLOR_ATTACHMENT_OPTIMAL`
- PBR: `COLOR_ATTACHMENT_OPTIMAL`
- Depth: `DEPTH_STENCIL_ATTACHMENT_OPTIMAL`

To boost performance make the eventual output, in this case the light attachment, occupy the first render target slot in the hardware. To do this, light attachment must be attachment #0 in the `VkRenderPass`.

To enable the render to write out emissive parameters from the opaque material, light is included as an output from the G-buffer in this example. There is no extra bandwidth to write out an extra render target because the subpasses are merged.

Unlike a desktop GPU, there is no need to invent schemes to forward emissive light contributions through the other G-buffer attachments.

Lighting pass (subpass #1) input attachments

- Albedo: `SHADER_READ_ONLY_OPTIMAL`
- Normal: `SHADER_READ_ONLY_OPTIMAL`
- PBR: `SHADER_READ_ONLY_OPTIMAL`
- Depth: `DEPTH_STENCIL_READ_ONLY`

From the point that any pass starts to read from the tile buffer, optimize multipass performance by marking every depth or stencil attachment as read-only.

`DEPTH_STENCIL_READ_ONLY` is designed for this read-only depth or stencil testing. It can be used concurrently as an input attachment to the shader program for programmatic access to depth values.

Lighting pass (subpass #1) output attachments

- Light: `COLOR_ATTACHMENT_OPTIMAL`

Lighting that is computed during subpass #1, is blended on top of the pre-computed emissive data from subpass #0. If needed, the application also blends transparent objects after the lighting passes have completed.

Subpass dependencies

Dependencies between the subpasses use `VkSubpassDependency` which sets the `DEPENDENCY_BY_REGION_BIT` flag. This dependency tells the driver that each subpass depends on the previous subpasses at that pixel coordinate.

For the example above, the subpass dependency setup would look like:

```
VkSubpassDependency subpassDependency = {};
subpassDependency.srcSubpass = 0;
subpassDependency.dstSubpass = 1;

subpassDependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT |
                                 VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |
                                 VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;

subpassDependency.dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT |
                                 VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT |
                                 VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT |
                                 VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;

subpassDependency.srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                                  VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT

subpassDependency.dstAccessMask = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT |
                                  VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
                                  VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
                                  VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
                                  VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;

subpassDependency.dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
```

Subpass merge considerations

The driver merges subpasses if the following conditions are met:

- The color attachment data formats can be merged.
- Merging can save a write-out or read-back. Two unrelated subpasses which do not share any data do not benefit from multipass and are not merged.
- The number of unique `VkAttachments` used for input and color attachments in all considered subpasses is less than nine. However, keep in mind that depth or stencil does not count towards this limit.
- The depth or stencil attachment does not change between subpasses.
- Multisample counts are the same for all attachments.

How to optimize the use of multipass rendering with Vulkan

Try using the following optimization steps:

- Use multipass.
- Use a 128-bit G-buffer budget for color.
- Use by-region dependencies between subpasses.
- Use `DEPTH_STENCIL_READ_ONLY` image layout for depth after the G-buffer pass is done.
- Use `LAZILY_ALLOCATED` memory to back images for every attachment except for the light buffer, which is the only texture that is written out to memory.
- Follow the basic render pass best practices, with `LOAD_OP_CLEAR or LOAD_OP_DONT_CARE` for attachment loads and `STORE_OP_DONT_CARE` for transient stores.

Multipass rendering steps to avoid

It is important that you do not store G-buffer data to memory, only to write the final color output.

The negative impact of implementing multipass rendering incorrectly

Not using multipass correctly forces the driver to use multiple physical passes, sending intermediate image data back to the main memory between passes. In turn, losing all of the benefits of multipass rendering.

How to debug multipass rendering issues more effectively

Here is a couple of steps you can try to debug issues you can encounter:

- To determine if passes are being merged, refer to the GPU performance counters for information about the number of physical tiles that are rendered.
- The GPU performance counters also provide information about the number of fragment threads using late-zs testing. A high value in the late-zs test can be indicative of your application not using `DEPTH_STENCIL_READ_ONLY` correctly.

6.6 HDR rendering

For mobile content, the relevant formats for rendering HDR images are `RGB10_A2`, for unorm data, and `B10R11G11` and `RGBA16F`, for floating-point data.

Prerequisites

You must understand the following concepts:

- HDR rendering.

How to optimize the use of HDR rendering

Try using the following optimization steps:

- Use `RGB10_A2 UNORM` formats for rendering where small increases in dynamic range are required.
- Use `B10G11R11` for floating-point rendering as it is only 32bpp, compared to 64bpp with full fp16 float.

HDR rendering steps to avoid

Arm recommends that you:

- Do not use `RGBA16F` unless:
  - `B10G11R11` is not providing suitable image quality.
  - Alpha is a requirement in the frame buffer.

Negative impacts of implementing HDR rendering incorrectly

The different types of impact you can see are:

- Increased bandwidth usage and reduced application performance.
- Fitting into 128bpp is difficult for multipass rendering to achieve efficiently.

6.7 Stencil updates

Many stencil masking algorithms toggle the stencil between a few values when creating and using a mask. When designing a stencil mask algorithm that uses multiple draw calls, the aim is to minimize the number of stencil buffer updates that occur.

Prerequisites

You must understand the following concepts:

- Stencil masking algorithms.

How to minimize stencil buffer updates

Try using the following optimization steps:

- If the values are the same, then use `KEEP` rather than `REPLACE`.
- Some algorithms use pairs of draws: one to create the stencil mask, and one to color the unmasked fragments. In this case, use the second draw to reset the stencil value so it is ready for the next pair. Doing so avoids the need for a separate clear operation.

HDR rendering steps to avoid

Do not waste performance by writing a new stencil value unnecessarily.

The negative impact of implementing stencil updates incorrectly

A fragment that writes a stencil value cannot be rapidly discarded. In turn, introducing an extra fragment processing cost that affects the performance of your application.

6.8 Blending

Blending is efficient on Mali GPUs because the `dstColor` is available on-chip, inside the tile buffer. However, it is important to remember that blending is more efficient for some formats than others.

Prerequisites

You must have a foundational knowledge of the following key graphics development concepts:

- Tile buffers.
- Blending.
- *Forward Pixel Kill* (FPK) - *https://community.arm.com/developer/tools-software/graphics/b/blog/posts/killing-pixels---a-new-optimization-for-shading-on-arm-mali-gpus*.
- Early-zs testing.

How to optimize blending

Here are several optimizations that you can try to optimize your application on Mali GPUs when using blending:

- Prefer blending on unorm formats, rather than floating-point values.
- Always disable blending and alpha-to-coverage if an object is opaque.
- Monitor the number of blended layers that are being generated on a per-pixel basis. Even if the shaders are simple, high layer counts quickly consume cycles due to the number of fragments that must be processed.
- Consider splitting large UI elements into opaque and transparent portions. The opaque and transparent portions can then be drawn separately, allowing either early-zs, or FPK, to remove the overdraw beneath the opaque parts.

Avoiding suboptimal blending

Arm recommends that you:

- Do not use blending on floating-point frame buffers.
- Do not use blending on multisampled floating-point frame buffers.
- Do not generalize the user interface rendering code so that blending is always enabled.
- Do not just set alpha to 1.0 in the fragment shader to disable blending.

The negative impact of inefficient blending

Blending has a significant impact on performance since blending disables many of the important optimizations that remove fragment overdraw, such as early-zs testing and FPK. The negative impact is especially noticeable for user interfaces and 2D games that use multiple layers of sprites.

Debugging OpenGL ES issues for Mali GPUs

When the time comes to debugging any blending issues, use the Graphics Analyzer tool to step through the construction of a frame. You must also monitor which draws calls are being blended and the amount of overdraw that the blends create.

6.9 Transaction elimination

Transaction elimination is a Mali technology that is used to avoid frame buffer write bandwidth for static regions of the framebuffer. It is more beneficial for games that contain many static opaque overlays.

Prerequisites

You must understand the following concepts:

- Vulkan APIs.
- Image layouts.

Avoiding frame buffer write bandwidth in Vulkan

Transaction elimination is used for an image if:

- The sample count is 1.
- The mipmap level is 1.
- The image uses `COLOR_ATTACHMENT_BIT`.
- The image does not use `TRANSIENT_ATTACHMENT_BIT`.
- A single color attachment is being used. Does not apply to the Mali-G51 GPU, or later.
- The effective tile size is 16x16 pixels. Pixel data storage determines the effective tile size.

The difference between image layouts that are defined as either safe or unsafe

Transaction elimination is a rare case where the driver uses the image layout. Whenever the image transitions from a safe to an unsafe image layout, the driver invalidates the transaction elimination signature buffer.

Safe image layouts are defined as image layouts that are either read-only, or images where only fragment shading writes to. These layouts are:

- `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`.
- `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`.
- `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`.
- `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`.

All other layouts are considered unsafe as they allow writes to an image that is outside of the write path of the tile.

If the color attachment reference layout is different from the final layout, then the signature buffer invalidation can happen as part of a `VkImageMemoryBarrier`, `vkCmdPipelineBarrier()`, `vkCmdWaitEvents()`, or as part of a `VkRenderPass`.

Transitioning from an `UNDEFINED` layout preserves the memory content of an image. The signature buffer is not invalidated unless another triggering effect requires it.

How to optimize transaction elimination on Mali GPUs

Try using the following optimization steps:

- For color attachments, use the `COLOR_ATTACHMENT_OPTIMAL` image layout.
- To avoid unnecessary signature invalidation, use the safe image layouts for color attachments.

Transaction elimination steps to avoid

Do not transition color attachments from safe to unsafe, unless the algorithm requires it.

The negative impact of not using transaction elimination

The loss of transaction elimination increases external memory bandwidth for scenes with static regions across frames. In turn, reducing the performance on systems that have limited memory bandwidth.

Debugging transaction elimination

To determine if transaction elimination is triggered, make the GPU performance counters count the number of tile writers that have been killed.

Chapter 7
Buffers and textures

This chapter contains information on how to optimize the performance of your in-game textures on a Mali GPU.

It contains the following sections:

7.1 Buffer update for OpenGL ES

OpenGL ES implements a synchronous programming model with each API call behaving as if the rendering triggered by earlier API calls has completed. In reality, this is an illusion as rendering is handled asynchronously, often completing milliseconds after the triggering API call was made.

Prerequisites

You must understand the following concepts:

- Buffers.
- Pipeline draining.
- Resource ghosting.
- OpenGL ES APIs.

Buffer update

To maintain the illusion, the driver must track resources that are referenced by pending render commands. The driver locks them to prevent modification until those rendering commands have been completed.

If the application attempts to modify a locked resource, then the driver must take some evasive action. Either draining the pipeline until the lock is released, or creating a new ghost copy of the resource to contain the modifications. Neither of these are free, and both incur an overhead that the application can avoid.

How to optimize buffer updates

Try using the following optimization steps:

- Perform modification of buffers using `glMapBufferRange()` and `MAP_UNSYNCHRONIZED` to explicitly bypass the synchronous rendering and resource lock semantics.
- Design buffer use to cycle though N buffers. Where the value of N is high enough to ensure that any resource locks have been released before a buffer is used again.
- Prefer complete buffer replacement using `glBufferData()` over partial replacement of buffers that are still referenced by pending commands.

Buffer update steps to avoid

Do not use `glBufferSubData()` to modify buffers that are still referenced by pending commands.

The negative impacts of inefficient buffer updates

Keep the following points in mind:

- Pipeline draining stalls the application until the resource lock is released. While this does not increase CPU load, GPU processing efficiency is reduced.
- CPU loads are increased when resource ghosting requires a new buffer to be allocated and any parts that are not overwritten are to be copied from the original.

7.2 Robust buffer access

Vulkan devices support bound checking to the GPU memory accesses by using `robustBufferAccess`.

Prerequisites

You must understand the following concepts:

- Bounds checking.
- Vulkan APIs.
- Push constants.
- Uniform buffers.

Adding bounds checking

Bounds checking ensures that accesses cannot fall outside of the buffer, preventing hard failure modes such as a GPU segmentation fault. However, do note that out-of-bounds access behavior, with bounds checking, is defined in the implementation. In turn, creating a render that cannot be predicted.

Caution

Enabling bounds checking causes loss in performance for accesses to uniform buffers and shader storage buffers.

How to optimize robust buffer access on Vulkan

Try using the following optimization steps:

- Use `robustBufferAccess` as a debugging tool during development.
- Disable `robustBufferAccess` in release builds. Only leave it enabled if the application use-case requires the additional level of reliability due to the use of unverified user-supplied draw parameters.
- If `robustBufferAccess` is required, use push constants in preference to uniform buffers. Doing so reduces the number of buffer accesses that are required per frame, meaning that fewer bound checks are needed.

Robust buffer access steps to avoid

Arm recommends that you:

- Do not enable `robustBufferAccess` unless it is needed.
- Do not enable `robustBufferAccess` without reviewing the performance impact that it can have on your application.

The negative impact of using robust buffer access

It is important to remember that even though `robustBufferAccess` has advantages, the use of `robustBufferAccess` causes measurable performance loss in both uniform buffers and shader storage buffers.

Debugging robust buffer access

Try the following debugging tips:

- To verify performance input, compare two test runs. One with `robustBufferAccess` enabled, and one without.
- The `robustBufferAccess` feature is a useful debug tool during development. If the application has problems with crashes or `DEVICE_LOST` errors being returned, then enable the robust access feature and see if the problem stops. If the problem does stop, there is either a draw call, or compute dispatch, that is making an out-of-bounds access.

7.3 Texture sampling performance

Depending on texture format and filtering mode, the Mali GPU texture unit can spend variable amounts of cycles sampling a texture. The texture unit is designed to give full-speed performance for both nearest sample and bilinear filtered, `LINEAR_MIPMAP_NEAREST`, texel sampling.

Prerequisites

You must understand the following concepts:

- Texture sampling.
- Texture formats.

Cases where extra texture sampling cycles are required

Ignoring data cache effects, the cases that require extra cycles are:

- Trilinear, `LINEAR_MIPMAP_LINEAR`, filtering has a 2x cost.
- 3D formats have a 2x cost.
- FP32 formats have a 2x cost.
- Depth formats have a 2x cost for Utgard or Midgard Mali GPUs, but only a 1x cost on Mali Bifrost GPUs.
- Cubemap formats have a 1x cost per cube face that is accessed.
- YUV formats have an *N*x cost, where *N* is the number of texture planes that are required for Mali GPUs that are older than Utgard. For the second-generation Bifrost architecture cores, that is Mali-G51 onwards, YUV has a 1x cost, irrespective of the plane count.

For example, a trilinear filtered RGBA8 3D texture access takes four times longer to filter than a bilinear 2D RGBA texture access.

How to optimize texture sampling performance

Try using the following optimization steps:

- Use the lowest texture resolution that you can.
- Use the narrowest precision that still retains a level of texture quality that is acceptable to you.
- For static resources, use offline texture compression, such as *Ericsson texture Compression* (ETC), *Ericsson texture Compression 2* (ETC2), or *Adaptive Scale Texture Compression* (ASTC).
- To improve both texture caching and image quality, always use mipmaps for textures that are in 3D scenes.
- Be careful to use trilinear filtering selectively if your content is texture-rate limited. Trilinear filtering gives the most benefits to textures with fine detail, such as text rendering.
- Use `mediump` samplers. `highp` samplers can be slower due to their wider data path requirements.
- If you need higher dynamic range, then consider using packed 32-bit formats. Such 32-bit formats include `RGB10_A2` or `RGB9_E5`, as an alternative to FP16 or FP32 textures.
- To lower the intermediate precision of the ASTC texture filtering, which can further improve performance and energy efficiency.
  - Use the `EXT_texture_compression_astc_decode_mode` extension for OpenGL ES. *https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_texture_compression_astc_decode_mode.txt*
  - Or the `VK_EXT_astc_decode_mode` extension for Vulkan: *https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#VK_EXT_astc_decode_mode*

Texture sampling steps to avoid

Arm recommends that you:

- Do not use wider data types unless essential.
- Do not use trilinear filtering for everything. Use it on a case-by-case basis only.

Negative impacts of unoptimized texture sampling

The different types of impact you can see are:

- Applications can experience problems due to either texture cache pressure, or general external memory bandwidth issues when content is doing any of the following:

  - Loading too much texture data.
  - Is using sparse sampling due to missing mipmaps.
  - Is using wide data types.
- Content that makes effective use of texture compression and mipmapping are typically limited by filtering performance rather than external memory bandwidth. Such content only sees a measurable impact if the texture unit is the critical path unit for the shaders. However, if a shader is arithmetically expensive, then the texture filtering cost can be hidden beneath.

Debugging texture sampling

Try the following debugging tips:

- The GPU performance counters can show you the utilization of the texture unit to determine if your application is texture filtering limited. Also, external bandwidth counters can monitor traffic to the external system memory.
- Try disabling trilinear filtering to see if it improves the performance.
- Try clamping the texture resolution to see if the performance improves.
- Try narrowing the texture formats that you use to see if performance improves.

One final note

The Mali Offline Compiler can help you identify if your important shaders are texture-rate limited through the statistics about functional unit usage that it presents. However, the texture cycle counts from the tool assumes one cycle per texel performance. The compiler does not have visibility of the precise sampler or format that you plan on using. You must manually de-rate the texture performance and base the performance on your own texture and sampler usage.

7.4 Anisotropic sampling performance

When determining the sample pattern to use, anisotropic filtering enables the texture sampling unit to account for the orientation of the triangle. Doing so improves image quality, in particular for primitives that are viewed at a steep angle regarding the view plan. However, anisotropic filtering comes at the cost of needing extra samples for a texture operation.

Prerequisites

You must understand the following concepts:

- Texture sampling.
- Bilinear filtering.
- Trilinear filtering.

Anisotropic Filtering (AF)

The following figure shows a cube that has been textured as a wooden crate. The image on the left uses traditional trilinear filtering, and the image on the right uses a bilinear filter with 2x AF. You can see the improved fidelity of the right-hand face of the cube when AF is used.

Figure 7-1 Trilinear vs Bilinear AF

From a performance perspective, the worst-case cost of AF is one sample per level of maximum anisotropy, which is controlled by the application. Samples can be bilinear and use `LINEAR_MIPMAP_NEAREST`, or trilinear and use `LINEAR_MIPMAP_LINEAR`.

Therefore, a 2x bilinear AF makes up to two bilinear samples. One significant advantage of anisotropic filtering is that actual number of samples that are made can be dynamically reduced based on the orientation of the primitive under the current sample position.

How to optimize anisotropic filtering

Try using the following optimization steps:

- Start using a max anisotropy of two, and then assess if it provides sufficient quality. Higher numbers of samples can improve quality, but they also give diminishing returns which are often not worth the performance cost.
- Consider using 2x bilinear AF in preference to isotropic trilinear filtering. 2x bilinear is faster, and has better image quality in regions of high anisotropy. Note, that by switching to bilinear filtering, you can see some visible seams at the decision point between mipmap levels.
- Only use AF and trilinear filtering for objects that benefit from it the most.

Anisotropic filtering steps to avoid

Arm recommends that you:

- Do not use higher levels of max anisotropy without reviewing performance. 8x bilinear AF costs eight times more GPU computational power than a simple bilinear filter.
- Do not use trilinear AF without reviewing performance. 8x trilinear AF costs 16 times more than a simple bilinear filter.

Negative impacts of using incorrect anisotropic filtering

The different types of impact you can see are:

- Using 2x bilinear AF, instead of trilinear filtering, increases image quality and can also improve performance.
- Using high levels of max anisotropy can improve image quality, but at the cost of performance.

Debugging anisotropic filtering performance

Arm recommends that to debug a performance problem with texture filtering, first try disabling AF completely and measuring any improvement. Then, incrementally increase the amount of max anisotropy that is allowed, and assess whether the quality is worth the additional cost to performance.

7.5 Texture and sampler descriptors

Mali GPUs cache texture and sampler descriptors in a control structure cache that can store a variable number of descriptors, depending on their content.

Prerequisites

You must understand the following concepts:

- Texture caching.
- Sampler descriptors.

Directions for maximizing the cache capacity of descriptor entries

Arm recommends that you use the following descriptor settings. Doing so ensures that you reach the maximum cache capacity in terms of descriptor entries, and therefore, the best performance:

For OpenGL ES only

- Set `GL_TEXTURE_WRAP_(S|T|R)` to identical values

  — **Note**

  The OpenGL ES driver can specialize the sampler state that is based on the current texture, unlike with Vulkan. So there is no need to set `GL_TEXTURE_WRAP_R` for 2D textures.

- Do not use `GL_CLAMP_TO_BORDER`.
- Set `GL_TEXTURE_MIN_LOD` to the default of -1000.0
- Set `GL_TEXTURE_MAX_LOD` to the default of +1000.0
- Set `GL_TEXTURE_BASE_LEVEL` to the default of 0
- Set `GL_TEXTURE_SWIZZLE_R` to the default of `GL_RED`.
- Set `GL_TEXTURE_SWIZZLE_G` to the default of `GL_GREEN`.
- Set `GL_TEXTURE_SWIZZLE_B` to the default of `GL_BLUE`.
- Set `GL_TEXTURE_SWIZZLE_A` to the default of `GL_ALPHA`.
- If the `EXT_texture_filter_anisotropic` filtering extension is available, then set `GL_TEXTURE_MAX_ANISOTROPY_EXT` to 1.0.

For Vulkan only

For Vulkan, when populating the `VkSamplerCreateInfo` structure:

- Set sampler `addressMode(U|V|W)` so they are all the same.

  — **Note**

  `addressModeW` must be set to be the same as U and V, even when sampling a 2D texture.

- Set sampler `mipLodBias` to 0.0
- Set sampler `minLod` to 0.0
- Set sampler `maxLod` to `VK_LOD_CLAMP_NONE`.
- Set sampler `anisotropyEnable` to `VK_FALSE`.
- Set sampler `maxAnisotropy` to 1.0
- Set sampler `borderColor` to `VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK`
- Set sampler `unnormalizedCoordinates` to `VK_FALSE`.

When populating the `VkImageViewCreateInfo` structure:

- Set every field that is in a view component to either `VK_COMPONENT_SWIZZLE_IDENTITY` or to the explicit per-channel identity-mapping equivalent.
- Set view `subresourceRange.baseMipLevel` to 0.

Using the Nearest and Linear filters in OpenGL ES and Vulkan

For emulating `GL_NEAREST` and `GL_LINEAR` sampling for mipmapped textures, the requirements for maximizing descriptor storage conflicts with the Vulkan recommended specification approach.

The Vulkan specification states that there are no Vulkan filter modes that directly correspond to OpenGL minification filters of `GL_LINEAR` or `GL_NEAREST`. However, required filters can be emulated using `VK_SAMPLER_MIPMAP_MODE_NEAREST`, `minLod = 0`, `maxLod = 0.25`, and using `minFilter = VK_FILTER_LINEAR` or `minFilter = VK_FILTER_NEAREST`, respectively.

To emulate these two texture filtering modes for a texture with multiple mipmaps levels, while also being compatible with the requirements for compact samplers, the recommended application behavior is to create a unique `VkImageView` instance. The `VkImageView` instance references only the level 0 mipmap and uses a `VkSampler` with `pCreateInfo.maxLod` setting to `VK_LOD_CLAMP_NONE` in accordance with the compact sampler restrictions.

Direct access to textures through `imageLoad()` and `imageStore()` in shader programs, or the equivalent in SPIR-V, are not impacted by this issue.

Behaviors to avoid when optimizing texture and sampler descriptors

Do not set `maxLod` to the maximum mipmap level in the texture chain. Instead, use `VK_LOD_CLAMP_NONE`. Otherwise, you can experience a reduced texture filtering throughput.

The negative impact of unoptimized texture and sampler descriptors

Reduced throughput of texture filtering.

7.6 sRGB textures

sRGB textures are natively supported in Mali GPU hardware. Both sampling from and rendering or blending to an sRGB surface comes at no performance cost. sRGB textures have a better perceptual color resolution than non-gamma corrected formats at the same bit depth. Therefore, Arm encourages the use of sRGB textures.

Prerequisites

You must understand the following concepts:

- sRGB textures.
- Color formats.

How to optimze sRGB texture performance

Try using the following optimization steps:

- Use sRGB textures for improved color quality.
- Use sRGB framebuffers for improved color quality.
- Remember that *Adaptive Scalable Texture Compression* (ASTC) supports sRGB compression modes for offline compressed textures.

Something to avoid when using sRGB textures

Do not use 16-bit linear formats to gain perceptual color resolution when 8-bit sRGB can suffice.

The negative impact of using an unoptimized texture format

The different types of impact you can see are:

- Not using sRGB textures, where appropriate, can reduce the quality of the image that is being rendered.
- Using wider float formats in place of sRGB textures increases bandwidth and reduces performance.

7.7 AFBC textures

Compatible with the Mali-T760 GPU onwards, *Arm FrameBuffer Compression* (AFBC) is a lossless image compression format. AFBC can be used for compressing framebuffer outputs from the GPU.

Prerequisites

You must understand the following concepts:

- Texture compression techniques.
- Framebuffer attachments.

Using AFBC

When enabled, AFBC is automatic and functionally transparent to the application. However, it is useful to be aware of some areas where AFBC cannot be used, and that can require the driver to insert run-time decompression from AFBC back to an uncompressed pixel format.

How to optimize the use of AFBC textures

Try using the following optimization steps:

- Use the `texture()` functions in shaders to access textures and images that were previously rendered by the GPU as framebuffer attachments.
- When packing data into color channels, to get the best compression rates, store the most volatile bits in the least significant bits of the channel.

Something to avoid when using AFBC textures

Do not use `imageLoad()` or `imageStore()` to read or write into a texture or image that was previously rendered by the GPU as a framebuffer attachment. Doing so triggers decompression.

The negative impact of not using AFBC textures correctly

Arm recommends that you keep in mind that the incorrect use of AFBC can trigger decompression.

Chapter 8
Compute

This chapter covers how to optimize workgroup sizes, how to correctly use shared memory on a Mali GPU, and optimized ways to process images.

It contains the following sections:

8.1 Workgroup sizes

Mali Midgard and Bifrost GPUs have a fixed number of registers available in each shader core. These GPUs can split those registers across a variable number of threads depending on the register usage requirements of the shader program.

Prerequisites

You must understand the following concepts:

- Shader core resource scheduling.
- Workgroups.
- Stack memory.

Using workgroups

The GPU hardware can split up, and then merge, workgroups during shader core resource scheduling. If barriers or shared memory are used, then GPUs cannot do this with workgroups. In such a case, all work items in the workgroup must be executed concurrently in the shader core.

Large workgroup sizes restrict the number of registers that are available to each work item in this scenario. In turn, forcing shader programs to use stack memory if insufficient registers are available.

How to optimize the use of workgroup sizes

Try using the following optimization steps:

- Use 64 as a baseline workgroup size.
- Use a multiple of 4 as a workgroup size.
- Try smaller workgroup sizes before larger ones, especially if using barriers or shared memory.
- When working with images or textures, use a square execution dimension, for example 8x8, to exploit optimal 2D cache locality.
- If a workgroup has per-workgroup work to be done, consider splitting the work into two passes. Doing so avoids barriers and kernels that contain portions where most threads are idle.
- Compute shader performance is not always intuitive, so keep measuring the performance levels.

Things to avoid when optimizing workgroup sizes

Arm recommends that you:

- Do not use more than 64 threads per workgroup.
- Do not assume that barriers with small workgroups are free from performance costs.

Negative impacts of not using workgroup sizes correctly

The different types of impact you can see are:

- Be careful with large workgroups. If a high percentage of work items are waiting on a barrier, then the shader core can be starved of work.
- Shaders that spill to the stack incur a higher load and store unit utilization, along with a higher cost to external memory bandwidth.

8.2 Shared memory

Mali GPUs do not implement dedicated on-chip shared memory for compute shaders. The shared memory that is available to use is system RAM that is backed up by the load-store cache.

Prerequisites

You must understand the following concepts:

- Cache memory.
- Memory allocation.
- Reduction shaders.

How to optimize the use of shared memory on Mali GPUs

Try using the following optimization steps:

- Use shared memory to share significant computation between threads in a workgroup.
- Keep your shared memory as small as possible, as it reduces the chance of thrashing the data cache.
- To reduce the size of the shared memory that is needed, reduce the precision and data widths.
- You need barriers to synchronize access to shared data. Shader code that has been ported from desktop development can sometimes omit barriers due to GPU-specific assumptions on warp width. Such an approach is not safe to use on mobile GPUs.
- It can be computationally cheaper splitting an algorithm over multiple shaders when compared to inserting barriers.
- For barriers, smaller workgroups are less expensive.

Things to avoid when optimizing shared memory use on Mali GPUs

Arm recommends that you:

- Do not copy data from global memory to shared memory on Mali GPUs. Doing so pollutes the caches.
- Do not use shared memory to implement code. For example:

```
if  (localInvocationID == 0) {
    common_setup();
}

barrier();

// Per-thread workload here

barrier();

if  (localInvocationID == 0) {
    result_reduction();
}
```

Splitting the example problem into three shaders would be an improvement. The setup and reduction shaders would need fewer threads.

The negative impact of not using shared memory correctly on Mali GPUs

The negative impact of using shared memory incorrectly is specific to your application, as it depends on the algorithmic design of the compute shader that you use.

8.3 Image processing

One common use case for compute shaders is for image post-processing effects. Remember however, that fragment shaders have access to many fixed-function features in the hardware. Such features can speed up things, reduce power, and even reduce bandwidth.

Prerequisites

You must understand the following concepts:

- Fragment shading.
- Compute.
- Pipeline bubbles.

Image processing

Here are some advantages to using fragment shading for image processing:

- Texture coordinates are interpolated using fixed function hardware when using varying interpolation. In turn, freeing up shader cycles for more useful workloads.
- Write out to memory can be done using the tile-writeback hardware in parallel to shader code.
- There is no need to range check `imageStore()` coordinates. Doing so can be a problem when you are using workgroups that do not subdivide a frame completely.
- It is possible to do framebuffer compression and transaction elimination.

Here are some advantages to using compute for image processing:

- It can be possible to exploit shared data sets between neighboring pixels. Doing so avoids extra passes for some algorithms.
- It is easier to work with larger working sets per thread, avoiding extra passes for some algorithms.
- For complicated algorithms such as FFTs, which need multiple fragment render passes, it is often possible to merge into a single compute dispatch.

How to optimize the use of image processing

Try using the following optimization steps:

- Use fragment shaders for simple image processing.
- For more complicated scenarios try using, and monitoring the performance of, compute shaders.
- Use `texture()` instead of `imageLoad()` for reading read-only texture data. `texture()` works with *Arm Frame Buffer Compression* (AFBC) textures that have been rendered by previous fragment passes. Using `texture()` also load balances the GPU pipelines better because `texture()` operations use the texture unit and both `imageLoad()` and `imageStore()` use the load or store unit. The load-store units are often already being used in compute shaders for generic memory accesses.

Things to avoid when optimizing your image processing implementation

Arm recommends that you:

- Do not use `imageLoad()` in compute unless you must use coherent read and writes within the dispatch.
- Do not use compute to process images that were produced by fragment shading. Doing so creates a backwards dependency that can cause a bubble. If fragment shader outputs are consumed by fragment shaders of later render passes, then render passes go through the pipeline more cleanly.

The negative impact of not using the correct image processing method

Compute shaders can be slower and less energy-efficient than fragment shaders for simple post-processing workloads. Examples of such workloads include: downscaling, upscaling, and blurs.

Chapter 9
Shader code

Writing optimal shader code through correct precision, vectorizing, uniforms and other techniques is key to optimizing the graphics performance of your application.

It contains the following sections:

9.1 Minimize precision

Mali GPUs have full support for reduced precision in the shader core register file and arithmetic units. Also, reducing precision on inputs and outputs saves data bandwidth. Using 16-bit precision is normally sufficient for computer graphics, especially for fragment shading when computing an output color.

Prerequisites

You must understand the following concepts:

- Different precision types, including `lowp` and `mediump`.
- Maintaining correct values through calculations involving reduced precision variables and temporaries.

Marking variables and temporaries

Both ESSL and Vulkan GLSL support marking variables and temporaries to use reduced precision level with `mediump`. There is no benefit to using `lowp` for Mali GPUs as it is functionally identical to `mediump`.

How to optimize the use of minimized precision on Mali GPUs

Try using the following optimization steps:

- Use `mediump` when the resulting precision is acceptable.
- Use `mediump` for inputs, outputs, variables, and samplers where possible.
- For angles use a range of -PI to +PI, rather than 0 to 2PI. Doing so gains extra precision for `mediump` values as you make use of the floating-point sign bit.

Things to avoid when optimizing your use of minimized precision on Mali GPUs

Do not test the correctness of `mediump` precision on desktop GPUs. Desktop GPUs ignore `mediump` and process it as `highp`. There is no difference in function or performance, so the test is worthless.

The negative impact of not using minimized precision correctly on Mali GPUs

If you choose to use full FP32 precision, performance and power efficiency can be negatively impacted.

How to debug precision-related performance issues on Mali GPUs

Try forcing `mediump` for everything except for the contributors to `gl_Position`, and then compare the performance difference afterwards.

9.2 Vectorized arithmetic code

The Mali Utgard and Midgard GPU architectures implement *Single Instruction Multiple Data* (SIMD) maths units, exposing vector instructions to each thread of execution. The Mali Bifrost GPU architecture switches to scalar arithmetic instructions, but still implements vector access to memory.

Prerequisites

You must understand the following concepts:

- Vector and scalar arithmetic instructions.
- Vector processing units.

How to optimize the use of vectorized arithmetic code on Mali GPUs

Try using the following optimization steps:

- Write vector arithmetic code in your shaders. While doing so is less critical with the introduction of Bifrost architecture, there are still large numbers of devices that are using Utgard and Midgard architecture.
- Write compute shaders so that work items contain enough work to fill the vector processing units.

Something to avoid when optimizing your use of vectorized arithmetic code on Mali GPUs

Do not write scalar code and hope that the compiler optimizes it. The compiler can, but it is more reliably vectorized if the input code starts out in vector form.

9.3 Vectorize memory access

The Mali GPU shader core load-store data cache has a wide data path capable of returning multiple values in a single clock cycle. It is important to make vector data accesses to get the highest access bandwidth from the data caches. Shader programs expose direct access to the underlying memory.

Prerequisites

You must understand the following concepts:

- The load-store data cache.
- Vector data accesses.
- Thread quads.

How to optimize the use of vectorized memory accesses on Mali GPUs

Try using the following optimizations:

- For memory accesses with a single thread, use vector data types.
- Bifrost GPUs can run four neighboring threads in lockstep, which is known as a quad. You must access overlapping or sequential memory ranges across the four threads to allow load merging.

Things to avoid when optimizing your use of vectorized memory accesses on Mali GPUs

Arm recommends that you:

- Do not use scalar loads if vector loads are possible.
- Do not access divergent addresses across a thread quad where possible.

The negative impact of not using vectorized memory accesses correctly on Mali GPUs

Many types of common compute programs perform relatively light arithmetic on large data sets. Getting memory access correct for them can have a significant impact on performance.

9.4 Manual source code optimization

There is no guarantee that the shader compiler can safely perform code transforms. Rendering errors can result from a floating-point infinity or *Not-a-Number* (NaN) that would not have occurred in the original program order.

Prerequisites

You must understand the following concepts:

- Code transforms.
- Matrix mathematics.
- Floating-point limitations.

Reducing the number of computations needed

Where possible, refactor your source code to reduce the number of computations that are needed, rather than relying on the compiler to apply the optimizations.

How to optimize your source code

Try using the following optimizations:

- Based on your knowledge of values, refactor your source code to optimize the code as much as possible by hand.
- Graphics are not bit exact. Therefore, you must be willing to approximate when it helps refactoring. For example, simplify `(A * 0.5) + (B * 0.45)` by using `(A + B) * 0.5` instead, saving a multiply.
- Use the built-in function library where possible. Often, a hardware implementation that is faster or, lower power than the equivalent hand-written shader code backs up the function library.

Things to avoid when optimizing your use of your source code

Do not reinvent the built-in function library in your custom shader code.

The negative impact of not using minimized precision correctly

Less efficient shader programs cause reduced application performance.

How to debug source-code related issues

Use the Mali Offline Compiler to measure the impact of your shader code changes, including analysis of shortest and longest path through the programs. *https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-offline-compiler*

9.5 Instruction caches

The shader core instruction cache is a performance-impacting area that is often overlooked. Due to the number of threads running concurrently, it is a critically important part to be aware of.

Prerequisites

You must understand the following concepts:

- Instruction caches.
- Early-zs testing.

How to optimize the use of instruction caches

Try using the following optimizations:

- Use shorter shaders with many threads over long shaders with few threads. A shorter program is more likely to be hot in the cache.
- Use shaders that do not have control-flow divergence. Divergence can reduce temporal locality and increase cache pressure.

Things to avoid when optimizing your use of instruction caches

Arm recommends that you:

- Do not unroll loops too aggressively, although some unrolling can help.
- Do not generate duplicate shader programs or pipeline binaries from identical source code.
- Beware of fragment shading with many visible layers in a tile. The shaders for all layers that are not killed by early-zs or *Forward Pixel Killing* (FPK), must be loaded and executed, increasing cache pressure. *https://community.arm.com/developer/tools-software/graphics/b/blog/posts/killing-pixels---a-new-optimization-for-shading-on-arm-mali-gpus*

How to debug instruction cache-related performance issues

Try the following debugging steps:

- Use the Mali Offline Compiler to statically determine the sizes of the programs being generated for any given Mali GPU. *https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-offline-compiler*
- The Arm Mobile Studio tool suite can be used to step through draw calls and visualize how many transparent layers are building up in your render passes. *https://www.arm.com/products/development-tools/graphics/arm-mobile-studio*

9.6 Uniforms

Mali GPUs can promote data from API-set uniforms and uniform buffers into shader core registers. The data is then loaded on a per draw basis, instead of on every shader thread. In turn, removing many load operations from the shader programs.

Prerequisites

You must understand the following concepts:

- Uniforms.

Using uniforms

Not all uniforms can be promoted into registers. Uniforms that are dynamically accessed cannot always be promoted to register-mapped uniforms, unless the compiler is able to make the uniforms constant expressions. For example, expression array indices that are made constant by loop unrolling a fixed iteration for-loop.

How to optimize the use of uniforms on Mali GPUs

Try using the following optimizations:

- Keep your uniform data small. 128 bytes is a good general rule for how much data can be promoted to registers in any given shader.
- Promote uniforms to compile-time constants with `#defines` for OpenGL ES, specialization constants for Vulkan, or literals in the shader source if they are static.
- Avoid uniform vectors or matrices that are padded with constant elements that are used in computation. For example, elements that are always zero or one.
- Prefer uniforms set by `glUniform*()` on OpenGL ES, or push constants on Vulkan, rather than uniforms loaded from buffers.

Things to avoid when optimizing your use of uniforms on Mali GPUs

Arm recommends that you:

- Do not dynamically index into uniform arrays.
- Do not over use instancing. Instanced uniforms that are indexed using `gl_InstanceID` count as being dynamically indexed and cannot use register mapped uniforms.

The negative impact of not using uniforms correctly on Mali GPUs

Register mapped uniforms cost little to use computationally. Any spilling to buffers in memory increases the load-store cache accesses to the per-thread uniform fetches.

How to debug uniform-related performance issues on Mali GPUs

The Mali Offline Compiler provides statistics about both the number of uniform registers that are being used, and the number of load and store instructions that are being generated. *https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-offline-compiler*

9.7 Uniform sub-expressions

One common source of inefficiency is the presence of uniform sub-expressions in the shader source. Uniform sub-expressions are pieces of code that only depend on the value of literals or other uniforms. Therefore, the results are always the same.

Prerequisites

You must understand the following concepts:

- Uniforms.

How to optimize the use of uniform sub-expressions on Mali GPUs

Minimize the number of uniform-on-uniform or uniform-on-literal computations. Compute the result of the uniform sub-expression on the CPU and then upload that as your uniform.

The negative impact of not using uniform sub-expressions correctly on Mali GPUs

The Mali GPU drivers can optimize the cost of most uniform sub-expressions so that they are only computed a single time per draw. While it can seem like the benefit is not as large as it appears, the optimization pass still incurs a small cost for every draw call. The draw call cost can be avoided by removing the redundancy.

How to debug uniform sub-expression-related performance issues on Mali GPUs

Use the Mali Offline Compiler to measure the impact of your shader code changes, including analysis of shortest and longest path through the programs. *https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-offline-compiler*

9.8 Uniform control-flow

One common source of inefficiency is the presence of conditional control-flow, such as `if` blocks and `for` loops, which are parameterized by uniform expressions.

Prerequisites

You must understand the following concepts:

- `#defines` in OpenGL ES.
- Specialization constants in Vulkan.

How to optimize the use of a uniform control-flow

Use `#defines` at compile time in OpenGL ES, and specialization constants in Vulkan for all control flow. Doing so allows the compilation to completely remove unused code blocks and statically unroll loops.

Things to avoid when optimizing your use of a uniform control-flow

Do not use uniform values that parametrize control-flows. Instead, specialize shaders for each control path that is needed.

The negative impact of not using contol-flow correctly

You can expect a reduced performance in your application due to less efficient shader programs.

How to debug uniform- control-flow-related performance issues

Use the Mali Offline Compiler to measure the impact of your shader code changes. Include an analysis of the shortest and longest path through the programs. *https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-offline-compiler*

9.9 Branches

Branching is expensive on a GPU. Branching either restricts how the compiler can pack groups of instructions within a thread. Or, when there is divergence across multiple threads, introduces cross-thread scheduling restrictions.

Prerequisites

You must understand the following concepts:

- Branching.
- Thread scheduling.

How to optimize the use of branches

Try using the following optimizations:

- Minimize the use of complex branches in shader programs.
- Minimize the amount of control-flow divergence in spatially adjacent shader threads.
- Use `min()`, `max()`, `clamp()`, and `mix()` functions to avoid small branches.
- Check the benefits of branching over computation. For example, skipping lights that are above a threshold distance from the camera. Often, it is faster just doing the computation.

Things to avoid when optimizing your use of branches

Do not implement multiple expensive data paths that are selected from using a `mix()`. Branching is usually the best solution for minimizing the overall cost in this particular scenario.

The negative impact of not using branches correctly

You can expect to experience a reduced performance in your application due to less efficient shader programs.

How to debug branch-related performance issues

Use the Mali Offline Compiler to measure the impact of your shader code changes. Include an analysis of shortest and longest path through the programs. *https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-offline-compiler*

9.10 Discards

Using a discard in a fragment shader, or when using alpha-to-coverage, are commonly used techniques. For example, alpha-testing complex shapes for foliage and trees.

Prerequisites

You must understand the following concepts:

- Fragment shaders.
- Alpha-to-coverage.
- Late-zs updates.
- Depth and stencil writing.

Using discards

These techniques force fragments to use late-zs updates. It is not known beforehand if the fragments survive the fragment operations stage of the pipeline until after shading. You must run the shader to determine the discard state of each sample. Doing so can cause redundant shading or pipeline starvation due to pixel dependencies.

How to optimize the use of discards and alpha-to-coverage

Try using the following optimizations:

- Minimize your use of shader discard and alpha-to-coverage.
- Computing lights in deferred lighting commonly uses fragment discard to cull fragments that are too far from the light source. To minimize execution bubbles, Arm recommends that you disable zs writes for these lighting passes.
- Render alpha-tested geometry front-to-back with depth-testing enabled. Doing so causes as many fragments as possible to fail early-zs testing. In turn, minimizing the number of late-zs updates needed.

The negative impact of not using discards and alpha-to-coverage correctly

Keep the following in mind:

- Extra fragment shading costs can cause a performance loss or bandwidth increase.
- Pipeline starvation waiting for pixel dependencies to resolve can also cause a loss in performance.

9.11 Atomics

Atomic operations are common to many compute algorithms and some fragment algorithms. With some slight modifications, atomic operations allow many algorithms to be implemented on highly parallel GPUs that would otherwise be serial.

Prerequisites

You must understand the following concepts:

- Atomics.
- Contention.
- L1 and L2 caches.

Atomics and contention

The key performance problem with atomics is contention. Atomic operations from different shader cores, hitting the same cache line, requires data coherency snooping through L2 cache. Which is computationally expensive.

Optimized compute applications that use atomics must aim to spread out the contention by keeping the atomic operations local to a single shader core. Atomics are efficient when a shader core controls the necessary cache line in its L1.

How to optimize atomics

Try using the following optimization steps:

- Consider how to avoid contention when using atomics in algorithm design.
- Consider spacing atomics 64 bytes apart to avoid multiple atomics contending on the same cache line.
- Consider whether it is possible to amortize the contention by accumulating into a shared memory atomic. Then, have one thread push the global atomic operation at the end of the workgroup.
- For OpenCL, consider the use of the `cl_arm_get_core_id` extension to allow explicit management of per-shader-core atomic variables.

Things to avoid when optimizing atomics

If better solutions that use multiple passes are available, then do not use atomics.

The negative impact of not using atomics correctly

Heavy contention on a single atomic cache entry significantly reduces overall throughput. It also impacts how well problems scale up when running on a GPU implementation with more shader cores.

How to debug atomics-related performance issues

The GPU performance counters include counters for monitoring the frequency of L1 cache snoops from the other shader cores in the system.

Chapter 10
System integration

This chapter covers how to optimizer system integration, for the Vulkan swapchain and other measures.

It contains the following sections:

10.1 Using EGL buffer preservation in OpenGL ES

Creating window surfaces that are configured with `EGL_BUFFER_PRESERVED` allows applications to logically update only part of the screen. With `EGL_BUFFER_PRESERVED`, applications always start rendering with the framebuffer state from the previous frame, rather than rendering from scratch.

Prerequisites

You must understand the following concepts:

- Framebuffers.
- How to use the `PartUpd` extension, *https://www.khronos.org/registry/EGL/extensions/KHR/EGL_KHR_partial_update.txt*
- How to use the `SwapDam` extension, *https://www.khronos.org/registry/EGL/extensions/KHR/EGL_KHR_swap_buffers_with_damage.txt*

Preserving the EGL buffer

If your application only wants to update a subregion of the screen, then using the EGL buffer can appear like an efficiency boost. However, in real systems, the use of double, or triple buffering means that rendering starts with a full screen blit from the window surface for frame *n*, into the window surface for frame *n+1*.

How to optimize the EGL buffer

Try using the following optimization steps:

- To minimize client rendering use the `PartUpd` extension, instead of `EGL_BUFFER_PRESERVED`. `PartUpd` allows incremental updates and true partial rendering of a frame.
- To minimize server-side composition overheads, use the `SwapDam` extension instead of `eglSwapBuffers()`.
- If the previous two extensions are not available, review the cost of using `EGL_BUFFER_PRESERVED` against the cost of just re-rendering a whole frame. It can be more efficient to re-render the entire frame from the original source material when using simple UI content that uses compressed texture inputs and procedural fills.
- If you know that an entire frame is being overdrawn when using `EGL_BUFFER_PRESERVED`, then insert a full-screen `glClear()` at the start of the render pass. Doing so removes the readback of the previous frame into the GPU tile memory.

Outcomes to avoid when using the EGL buffer

Arm recommends that you:

- Do not use `EGL_BUFFER_PRESERVED` surfaces without considering if it makes sense for the content involved. Always try using `EGL_BUFFER_DESTROYED` and then measure the benefits.
- Do not use either `EGL_KHR_partial_update` or `EGL_KHR_swap_buffers_with_damage` for applications that always re-render the whole frame. There is an extra cost to the software in doing so.

The negative impact of not using the EGL buffer correctly

Unnecessary readbacks can reduce performance and increase memory bandwidth.

10.2 The Android blob cache size in OpenGL ES

The Mali OpenGL ES drivers use the `BlobCache` extension to persistently cache the results of shader compilation and linkage.

Prerequisites

You must understand the following concepts:

- `BlobCache` extension, *https://www.khronos.org/registry/EGL/extensions/ANDROID/EGL_ANDROID_blob_cache.txt*

Reconsider the size of the Android blob cache

For many applications, shaders are only compiled and linked when the application is first used. Subsequent application runs use binaries from the cache, benefiting from a faster startup and level load times.

The Android blob cache defaults to a relatively small size, 64KB per application. 64KB per application is insufficient for many modern games and applications that can use hundreds of shaders.

Therefore, to increase the number of applications which benefit from program caching, we recommend that system integrators significantly increase the maximum size of the blob cache for their platforms.

How to optimize the size of the Android blob cache

Try increasing the size of the Android blob cache for each application. For example, up to 512KB or even 1MB.

The negative impact of not setting the Android blob cache size correctly

As shader programs must be compiled and linked at runtime, any games and applications that exceed the size of the blob cache begin more slowly.

10.3 Optimizing the swapchain surface count for Vulkan

The application has control over the window surface swapchain when using Vulkan. In particular, the application can decide how many surfaces to use.

Prerequisites

You must understand the following concepts:

- Vulkan windows surface swapchain.

The swapchain and vsync

Most mobile platforms use the Vsync signal of the display to prevent screen tearing on buffer swap. If the GPU is rendering more slowly than the vsync period, then a swapchain that contains only two buffers is prone to stalling the GPU.

How to optimize the swapchain surface count

Try using the following optimization steps:

- If your application always runs faster than vsync, then use two surfaces in the swapchain. Doing so reduces your memory consumption.
- If your application sometimes runs more slowly than vsync, then use three surfaces in the swapchain. Doing so gives you the best performance for your application.

Outcomes to avoid when using the window surface swapchain

If your application runs more slowly than vsync, then do not use two surfaces in the swapchain.

The negative impact of not using the window surface swapchain correctly

Using double buffering and vsync locks rendering to an integer fraction of the vsync rate. In turn, reducing the performance of the application if rendering is slower than vsync.

For example, on a device that uses a 60FPS panel refresh rate, an application that is otherwise capable of running at 50FPS, drops down to 30FPS.

How to debug swapchain-related performance issues

System profilers, such as Streamline, can show when the GPU is going idle. If the GPU is going idle, and the frame period is a multiple of the vsync period, then it can indicate rendering blocking and that the GPU is waiting for a buffer to be released by the vsync signal.

10.4 Optimizing the swapchain surface rotation for Vulkan

For Vulkan, the application has control over the window surface orientation. The application is responsible for handling the differences between the logical and the physical orientation of the window when a mobile device is rotated.

Prerequisites

You must understand the following concepts:

- How swapchain surface rotation works.

Swapchain surface rotation

It is more efficient for the presentation subsystem if the application renders into a window surface whose orientation matches the physical orientation of the display panel.

How to optimize swapchain surface rotation

Try using the following optimization steps:

- To avoid presentation engine transformation passes ensure that swapchain `preTransform` value matches the `currentTransform` value that is returned by `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`.
- If a swapchain image acquisition returns `VK_SUBOPTIMAL_KHR` or `VK_ERROR_OUT_OF_DATE_KHR`, then recreate the swapchain. When doing so, consider any updated surface properties, including potential orientation updates reported using `currentTransform`.

Outcomes to avoid when using swapchain surface rotation

Do not assume that supported presentation engines transforms, other than `currentTransform`, are free. Many presentation engines can handle rotation or mirroring. However, it can come with extra processing cost.

The negative impact of not using swapchain surface rotation correctly

Non-native orientation can require extra transformation passes in the presentation engine. Therefore, some systems must use the GPU as part of the presentation engine to handle cases that the display controller cannot handle natively.

How to debug swapchain surface rotation performance issues

System profilers, such as the kernel-integrated version of Streamline, can track the use of the Mali GPU to specific processes. In turn, allowing the attribution of extra GPU workload to the compositor process. However, the previous step assumes that the GPU is being used by the compositor to apply the presentation transformation.

10.5 Optimizing swapchain semaphores for Vulkan

In Vulkan, semaphores allow synchronization and safe access of swapchain data.

Prerequisites

You must understand the following concepts:

- Semaphores.
- Swapchains.
- *Windows System Integration* (WSI).
- Pipeline bubbles.

Swapchain semaphores

The following is a typical example of what a Vulkan frame looks like:

1. Create a `VkSemaphore`, *`#1`*, for the start of frame acquire.
2. Create a separate `VkSemaphore`, *`#2`*, for the end of frame release.
3. Calling `vkAcquireNextImage()` gives you the swapchain index *`#N`* and then associates with semaphore *`#1`*.
4. Wait for all fences that are associated with swapchain index *`#N`*.
5. Build the command buffers.
6. Submit the command buffers rendering to the window surface to `VkQueue`. Tell the command buffers to wait for semaphore *`#1`* before rendering can begin, and to then signal semaphore *`#2`* when the command buffers have completed.
7. Call `vkQueuePresent()`, configuring it to wait for semaphore `#2`.

The critical part of the previous example occurs when setting up the wait for any command buffers on semaphore *`#1`* . We must also specify which pipeline stages must wait for the WSI semaphore.

Along with `pWaitSemaphores[i]`, there is also `pWaitDstStageMask[i]`. The `pWaitDstStageMask[i]` mask specifies which pipeline stages must wait for the WSI semaphore. You must wait for the final color buffer using `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`.

When rendering, you must transition the WSI image from either an `UNDEFINED`, or `PRESENT_SRC_KHR` layout to a different layout. The layout transition must wait for `COLOR_ATTACHMENT_OUTPUT_BIT`, creating a dependency chain to the semaphore.

How to optimize swapchain semaphores in WSI

When the application is waiting for a semaphore from WSI, use `pWaitDstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`.

The negative impact of not using swapchain semaphores correctly

Large pipeline bubbles are created when the vertex, or compute, stalls.

10.6 Window buffer alignment

If rows are insufficiently aligned, then linear format framebuffers can suffer from dramatically reduced write performance. To ensure an efficient write performance, align all memory system allocated surfaces that are imported into Mali drivers.

Prerequisites

You must understand the following concepts:

- Framebuffer formats.
- Row alignments.

How to optimize the window buffer alignment

Try using the following optimization steps:

- Align rows in linear format framebuffers to the smallest possible alignment of either a multiple of 16 pixels, or 64 bytes. For example, for `RGB565` framebuffers you can use a 32-byte alignment. You can use a 64-byte alignment for `RGBA8` and `RGBA fp16` framebuffers.
- When an alpha channel is not required, then use power-of-two formats as they have better memory alignment properties. Use a format such as `RGBX8` with a dummy channel, instead of using `RGB8`.

Outcomes to avoid when using the window buffer alignment

Arm recommends that you:

- Do not use any row alignment other than either a multiple of 16 pixels, or 64 bytes.
- Do not use framebuffer formats that are not power-of-two. For example, do not use true 24-bit `RGB8`.

The negative impact of not using the window buffer alignment correctly

AXI bursts must be aligned to the burst size. Unaligned rows must be broken into multiple smaller AXI bursts to meet this requirement. Doing so makes less efficient use of the memory bus and, often causes the GPU to stall. The stalling occurs because the GPU exhausts the pool of available transaction IDs.

Appendix A
Revisions

This appendix provides additional information that is related to this guide.

It contains the following section:

- *A.1 Revisions* on page Appx-A-98.

A.1 Revisions

This appendix describes the technical changes between released issues of this guide.

Table A-1 Issue 0100-00

Change	Location	Affects
First release.	-	-

Table A-2 Issue 0101-00

Change	Location	Affects
Added information about command pools.	*3.6 Command pools for Vulkan* on page 3-35	-
Added information about secondary command buffers.	*3.8 Secondary command buffers* on page 3-37	-
Added information about buffer update.	*7.1 Buffer update for OpenGL ES* on page 7-64	-

Table A-3 Issue 0200-00

Change	Location	Affects
No changes.	-	-