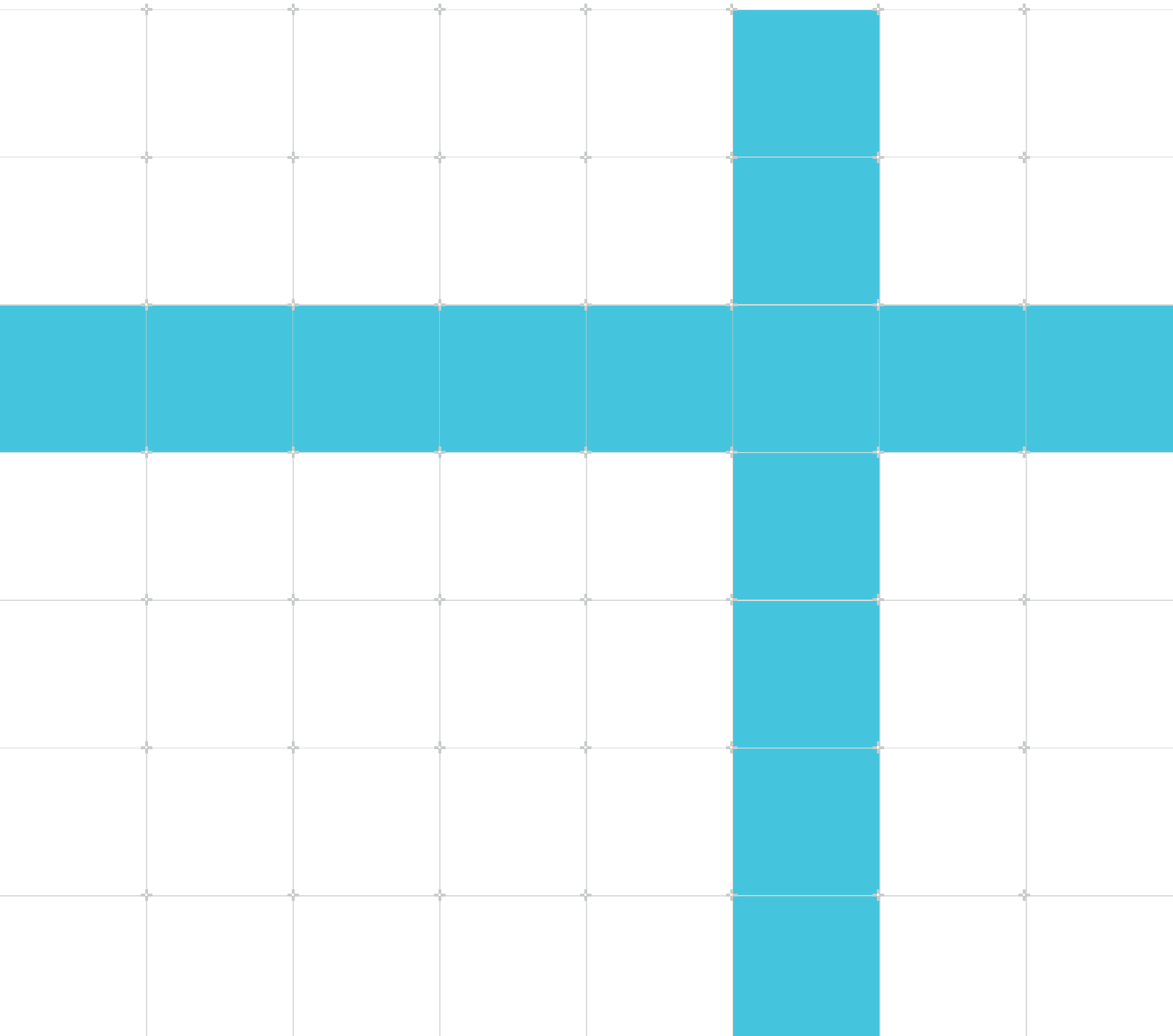
arm

# Optimization advice for graphics content on mobile devices

Version 1.0



**Issue 00**
102643_0100_00_en



# Optimization advice for graphics content on mobile devices



## Release information

### Document history

| Issue | Date | Confidentiality | Change |
|---|---|---|---|
| 0100-00 | 10 August 2021 | Non-Confidential | Initial release |

## Proprietary Notice

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

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



Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

## Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

## Product Status

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

## Feedback

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

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

## Inclusive language commitment

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

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

# Contents

# 1. Overview

These topics provide advice on how to optimize your Android application by avoiding common graphics problems that might cause your application to run slow, or cause the device to overheat.

Arm Mobile Studio can help you to identify performance bottlenecks, and determine where your application is CPU, vertex or fragment bound. Once you know this, use the advice here to increase your application's performance:

- Avoid redundant work
- Reduce precision
- Issue only visible draws, in the right order
- Optimize shader programs
- Use texture compression, mesh level-of detail and mipmapping techniques.

For further best practises for Mali-based devices, you can also refer to the Mali GPU best practises guide.

# 2. Improving CPU bound content

Content that is CPU bound fails to hit its target performance due to high software processing demands. There are two causes of CPU boundness: Critical path load and high aggregate load.

## Critical path load

High critical path load is caused when a single thread, or a synchronized sequence of multiple threads, creates a bottleneck. It is the most common type of CPU bottleneck found in applications running on multi-core mobile devices. To improve performance, you must reduce the CPU load on this critical path, either by optimizing content to reduce cost or by moving work into threads that can execute in parallel.

## High aggregate load

High aggregate load occurs when the total cumulative load of the executing processes and application threads is too high. In this scenario, poor performance is not due to a single thread or critical path, so you must reduce any CPU workload across all threads to improve performance.

## Improving render thread performance

High loads can occur when the application rendering thread makes poor use of the native graphics API. Here are some recommended best practices for reducing the CPU cost of rendering operations:

1. Perform expensive software operations at level load time, not during gameplay. This could include operations such as shader compilation, program linking, large buffer upload, and large texture data upload.
2. Do not modify buffers or textures that are still referenced by in-flight draw calls, unless you are using `MAP_UNSYNCHRONIZED` to disable resource dependency tracking.
3. Large numbers of draw calls are expensive for the CPU to process, so you should aim to reduce the number of draw calls. Refer to the advice page for high draw calls for tips on how to do this.

# 3. Improving CPU to GPU scheduling bound content

Content that is CPU-to-GPU scheduling bound fails to hit its target frame rate, but neither CPU nor GPU is kept busy because of workload serialization across the CPU-GPU interface.

The rendering process is designed to be asynchronously pipelined. The CPU puts new rendering work into a queue, to be processed by the GPU some time later. For content that is not hitting target performance, we want the CPU to keep some work in this queue. If the queue ever empties, then the GPU will go idle and performance is wasted.

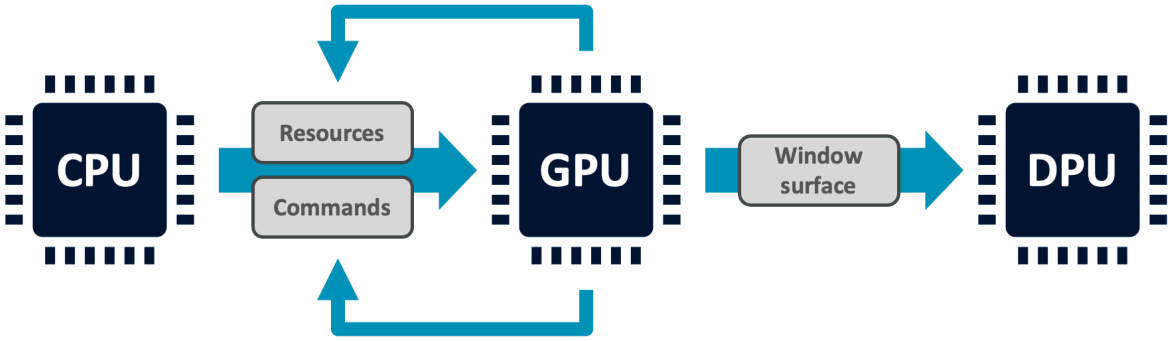
The main cause for this queue to empty is where the CPU is blocked and is waiting for some of the queued work to complete. When it is blocked, the CPU stops adding more rendering to the queue, so it is possible for the queue to drain. Here are some recommendations to avoid this issue:

1. Avoid using API calls that force a pipeline drain, such as `glFinish()` or a synchronous `glReadPixels()`.
2. Avoid using `glMapBuffer()` on a buffer that is still referenced by an in-flight draw call or compute dispatch, unless you are using `MAP_UNSYNCHRONIZED`.
3. Use query objects and client-side fences in a pipelined way, waiting for the result at least one, and ideally two, frames after the query or fence was submitted to the command stream.

# 4. Data resource issues

The CPU controls the rendering process. It provides new data, such as transforms and light positions for every frame. GPU processing is asynchronous. This means data resources may be referenced by a queued command, and sit in the command stream for some time. OpenGL ES needs rendering to reflect the state of the resources at the time the draw call was made, so resources can't be modified until the GPU workload referencing them completes.

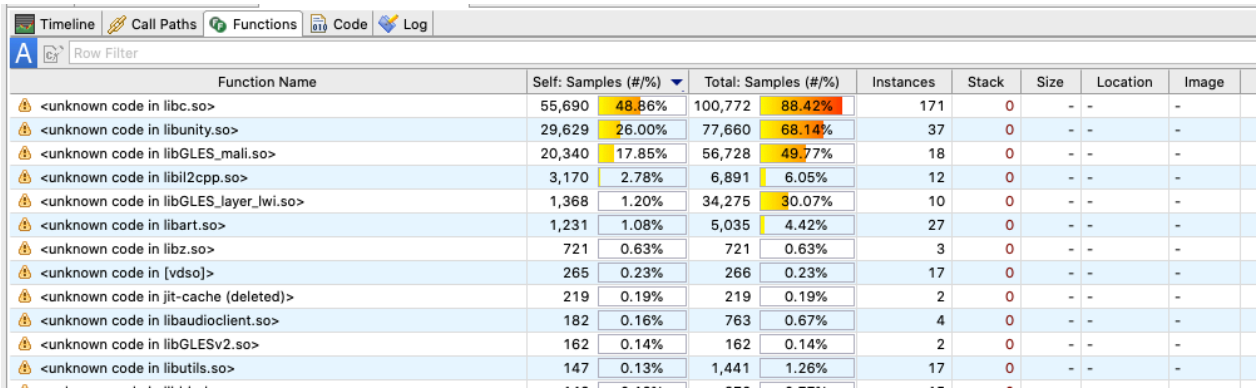
**Figure 4-1: Rendering pipeline**




Attempting to modify a resource that's still referenced will usually trigger the creation of a new copy of that resource. This will avoid stalling the pipeline, but it creates significant CPU overhead. In effect, we need to make a new memory allocation, then copy data to populate the parts which were not replaced. This is often referred to as resource ghosting.

Streamline lets you pinpoint instances of high CPU load. Where this is being caused by resource ghost creation, you might see high-time inside the graphics driver `libGLES_Mali.so`, in the Call Paths or Functions views.

**Figure 4-2: The Call Paths and Function views in Streamline**




| Function Name | Self: Samples (#/%) | | Total: Samples (#/%) | | Instances | Stack | Size | Location | Image |
|---|---|---|---|---|---|---|---|---|---|
| <unknown code in libc.so> | 55,690 | 48.86% | 100,772 | 88.42% | 171 | 0 | - | - | - |
| <unknown code in libunity.so> | 29,629 | 26.00% | 77,660 | 68.14% | 37 | 0 | - | - | - |
| <unknown code in libGLES_mali.so> | 20,340 | 17.85% | 56,728 | 49.77% | 18 | 0 | - | - | - |
| <unknown code in libil2cpp.so> | 3,170 | 2.78% | 6,891 | 6.05% | 12 | 0 | - | - | - |
| <unknown code in libGLES_layer_lwi.so> | 1,368 | 1.20% | 34,275 | 30.07% | 10 | 0 | - | - | - |
| <unknown code in libart.so> | 1,231 | 1.08% | 5,035 | 4.42% | 27 | 0 | - | - | - |
| <unknown code in libz.so> | 721 | 0.63% | 721 | 0.63% | 3 | 0 | - | - | - |
| <unknown code in [vdso]> | 265 | 0.23% | 266 | 0.23% | 17 | 0 | - | - | - |
| <unknown code in jit-cache (deleted)> | 219 | 0.19% | 219 | 0.19% | 2 | 0 | - | - | - |
| <unknown code in libaudioclient.so> | 182 | 0.16% | 763 | 0.67% | 4 | 0 | - | - | - |
| <unknown code in libGLESv2.so> | 162 | 0.14% | 162 | 0.14% | 2 | 0 | - | - | - |
| <unknown code in libutils.so> | 147 | 0.13% | 1,441 | 1.26% | 17 | 0 | - | - | - |

There are two ways to resolve this problem:

- The application creates multiple buffers (or textures) and uses them in a round-robin fashion. The aim here is to have enough buffers in the rotation that all pending references have dropped by the time they are reused in a later frame.
- Use `glMapBufferRange` with `GL_MAP_UNSYNCHRONIZED`. You can then build the rotation using subregions inside a single buffer. This avoids the need for multiple buffers, however, you will need to manage the subregion dependencies in application logic. Avoid specifying the `MAP_INVALIDATE` flag. Arm is a unified memory architecture, so the flag can be safely omitted. In fact on some older Mali driver releases, specifying `INVALIDATE` would override the unsynchronized behavior and re-trigger ghosting.

# 5. Improving fragment bound content

Content that is fragment bound fails to hit its target performance due to high fragment processing demands. There are three main causes of slow fragment performance.

- Too many fragments to shade
- Too many shader cycles per fragment

Content that causes poor fragment processing efficiency

One of the most common problems is with content that tries to shade too many fragments, or fragments that are too expensive, given the performance capabilities of the target GPU. This is a particularly common problem in mass-market devices, which have smaller GPU configurations than high-end smartphones, but have a similar screen resolution. For these devices it is a useful exercise to set a per-pixel performance budget to help guide design choices.

## Set a performance budget

Consider a device with a Mali-G72 MP2, a two core and two pixel-per-clock GPU, running at 600MHz. The best-case cycle budget for this device when targeting 1080p 60FPS is:

```
pixelsPerSecond = 1920 * 1080 * 60 = 124,416,000
 cyclesPerSecond = 2 * 600,000,000 = 1,200,000,000
 cyclesPerPixel = cyclesPerSecond / pixelsPerSecond = 9.6
```

This budget assumes 100% shader core utilization and must include all frame costs, including vertex shading. This is a usable budget for a 2D game or a simple 3D game, but it's impossible to run a high-end rendering pipeline inside this budget. The first set of choices that you should review for mass market devices are therefore the target resolution and frame rate, as these are easy to change and have the biggest impact on the overall pipeline cost. Dropping the target configuration to 720p 30FPS frees up a lot of processing capacity, increasing the cycle budget to over 40 cycles per pixel.

## Minimize the number of fragments

Once those coarse settings have been decided, it is important to minimize the number of fragments that must be shaded for each frame, as rendering multiple layers of fragment per pixel can rapidly consume valuable cycles.

1. Render opaque objects from front-to-back. Objects closes:
    a. Disable blending
    b. Disable alpha-to-coverage
    c. Reduce the number of shaders that use discard statements.

       This will maximize the number of fragments killed by early depth testing and hidden surface removal.

2. Review menus and user interfaces for efficient use of transparent layers; layers of 2D interface components can quickly accumulate into a high layer count, which is expensive to process even if the layers themselves are simple.

### Minimize processing cost per fragment

For fragments that are required, you should reduce the processing cost per fragment. Exactly what is required depends on the dominant shader pipeline, but here are some best practises:

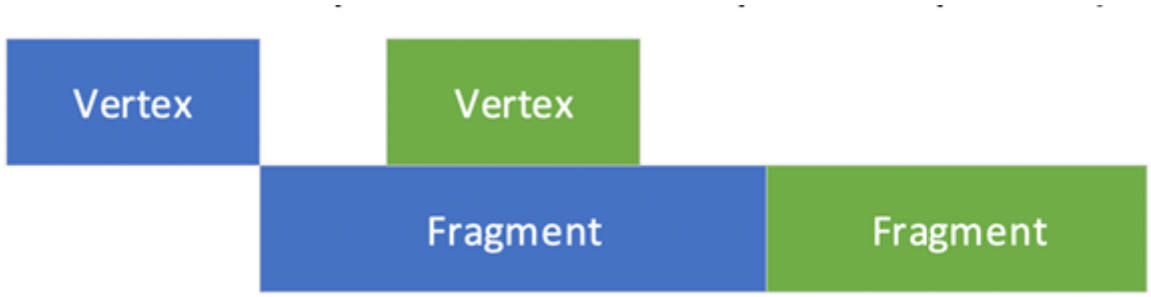
1. Reduce the precision of computation - mediump arithmetic is faster than highp arithmetic.
2. Reduce the precision of per-vertex inputs - mediump varying values use less memory and interpolate faster than highp varying values.
3. Reduce texture filtering complexity - bilinear (LINEAR_MIP_NEAREST) filtering is faster than trilinear (LINEAR_MIP_LINEAR) filtering, and you should only use anisotropic filtering sparingly.

# 6. Improving GPU queue scheduling bound content

Content that is GPU queue scheduling bound fails to hit its target frame rate, and the GPU is busy all of the time, but neither GPU queue is kept busy because of workload serialization across them.

The Mali tile-based rendering process is designed to parallelize across the two GPU queues. One render pass is being vertex shaded in one queue, while an earlier pass is being fragment shaded in the other. This parallel processing ensures the most efficient use of the available processing resources.

**Figure 6-1: Vertex and fragment tile-based rendering**



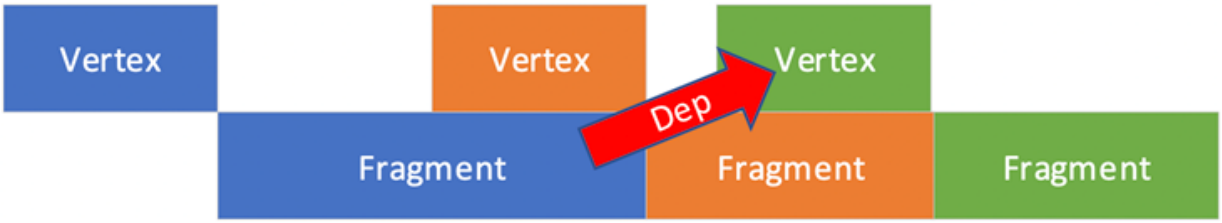

## OpenGL ES

For OpenGL ES, serialization across queues commonly occurs because of data dependencies between render passes. For example, if a vertex shader in render pass N reads a texture written to by the fragment stage in render pass N-1 then it can not start until render pass N-1 has completed.

**Figure 6-2: Vertex shader in render pass**


Aim to minimize these dependencies by inserting non-dependent work between the two dependent processing stages, allowing other work to fill the bubble.

**Figure 6-3: Vertex dependency in the fragment bubble**




## Vulkan

For Vulkan, workload dependencies are explicitly stated by the application, so the most common cause for queue-to-queue scheduling issues is where the application specifies overly conservative dependencies that force serialization when it is not required. For example, specifying `rcStage=BOTTOM_OF_PIPE` and `dstStage=TOP_OF_PIPE` makes all render passes run serially with no parallel processing.

Your application should specify the most relaxed dependencies possible while still maintaining correctness. This means that `srcStage` should be as early in the pipeline as possible, and `dstStage` should be as late in the pipeline as possible. For cases where the minimal valid dependencies still cause slot serialization, follow the advice stated for OpenGL ES above, and insert non-dependent work between the dependent passes to fill the bubble.

# 7. High arithmetic load

For applications that are GPU limited, and have high shader loads dominated by arithmetic processing, the solution is to reduce arithmetic complexity in the shaders.

To reduce arithmetic load:

1. Reduce precision - mediump computation can be twice as fast as highp computation.
2. Avoid branch divergence - divergent branches within the threads of a warp reduce arithmetic efficiency as not all threads are active when executing divergent code paths.
3. Vectorize operations - Mali Midgard GPUs use SIMD arithmetic logic, so matrix and vector operations in the source code are more likely to vectorize well into SIMD operations than scalar operations.
4. Move processing from per-fragment to per-vertex, to lower evaluation frequency.

# 8. High culling percentage

Triangles are expensive inputs to a GPU, so it is critical to make sure that they are used wisely. Triangles that are thrown away during primitive culling have no visual benefit to the scene, so you should aim to optimize draw call dispatch on the GPU to minimize the number of primitives that are culled.

For well-performing 3D content, it is expected that half of all triangles are culled due to the facing test, because they make up the side of a model that is facing away from the camera. If significantly fewer triangles than this are being killed by the facing test, check that back-face culling is enabled for as many draw calls as possible.

The frustum test runs after the facing test and kills triangles that are outside of the view frustum. Aim to minimize the number of primitives killed by this test by filtering out as much as possible on the CPU, discarding a draw call when all objects that it contains are outside of the view frustum.

The sample test runs last, and kills triangles that are so small that they hit no sample points. Any measurable level of culling detected at this stage is indicative of an application using very dense geometry, which is very expensive for the GPU to process.

Avoid tiny triangles by:

1. Using dynamic mesh level-of-detail to select simpler meshes as objects get further from the camera.
2. Using texture pseudo-geometry techniques such as normal mapping to replace fine detail physical geometry.

# 9. High draw calls

Large numbers of draw calls are expensive for the CPU to process, so it is important to reduce the number of draw calls, particularly those that have no visible impact on the scene.

To reduce the number of draw calls:

1. Use software culling techniques to discard objects that are not visible to the camera, reducing the number of API draw calls. Techniques could include using algorithms such as bounding box intersection with the camera frustum (test if a spatial volume is visible), and portal visibility checks (test if a room is visible).
2. Use object batching techniques to merge multiple objects into a single draw, reducing the number of API draw calls.

## Related information

For more information about best practises for handling draw calls, refer to the following topics in the Mali GPU best practises guide:

- Draw call batching
- Draw call culling
- Optimizing the draw call render order

# 10. High load store

To improve the performance of applications that are GPU-limited, and that have high shader loads dominated by load/store operations, you should improve memory access efficiency and vectorization in your shader programs.

To reduce a high load/store load:

1. Improve access density, by using vector loads in compute shaders, and access patterns that touch adjacent data from adjacent threads in each warp. This will enable a single cache line access to return data for multiple threads.
2. Reduce cache pressure, by reducing precision and improving spatial locality of accesses.
3. Avoid using `imageLoad()` calls for read-only texture accesses. Use `texture()` calls instead.
4. Avoid using atomic calls, because they have a high per-thread cost.

# 11. High overdraw

Content that has a high degree of overdraw - multiple fragments shaded per output pixel - can suffer from poor performance because of the cumulative cost of shading all of the layers. This can occur even if the layers are individually simple, especially for devices running at high resolutions and frame rates.

Reduce overdraw in your application by:

1. Ensuring draw calls are opaque:
    a. Disable blending
    b. Disable alpha-to-coverage
    c. Avoid shaders that use discard statements.
2. Splitting large UI elements with a mixture of opaque and transparent parts into two draw calls; one for the opaque parts and one for the transparent parts.
3. Changing the content to reducing the number of layers, pre-baking layers of transparency into a single flattened texture where possible.

# 12. High texture load

To improve the performance of applications that are GPU-limited, and that have high shader load dominated by texture processing, you should reduce texture filtering complexity.

To reduce a high texture load:

1. Reduce filtering quality
    - Bilinear (LINEAR_MIP_NEAREST) filters are twice as fast as trilinear (LINEAR_MIP_LINAR) filters.
    - A lower level of MAX_ANISOTROPY will limit how many texture samples are made per shader operation.
2. Reduce sampler precision - mediump sampler types need less data than highp samplers.
3. Reduce data size:
    - Use texture compression
    - Use mipmaps
    - Use ASTC_decode_mode for ASTC textures
    - Use narrower uncompressed formats.

# 13. High varying load

To improve the performance of applications that are GPU-limited, and that have high shader loads dominated by varying interpolation processing, you should reduce interpolation complexity in your shader programs.

To reduce a high varying load:

1. Reduce precision - mediump interpolation is twice as fast as highp interpolation.
2. Pack mediump vectors into multiples of 32-bits.


Note

Vectors that are a multiple of 32 bits in length are more efficient than non-packed vectors. For example, a mediump vec2 + vec2 pair is faster than a mediump vec3 + float pair.

# 14. Improving thermally bound applications

When under a high processing load, high-end smartphones and tablets can produce more heat than they can dissipate. If such a load is prolonged, the device starts to throttle its performance to avoid overheating, reducing CPU and GPU performance until a thermally-sustainable level is reached. To maintain smooth gameplay and a consistent frame rate, you should aim to make efficient use of the available CPU, GPU, and memory bandwidth to avoid pushing the device into a thermally unsustainable performance requirement.

When a device is thermally bound, optimizing any workload, even if it is not the highest load, will improve the thermal situation. You should review the following areas:

1. DRAM access is very power intensive, costing approximately 100pJ per byte for LPDDR4. Aim to reduce the number of memory accesses per frame. To do this:
    - Reduce the resolution of render passes, and use narrower attachment color formats.
    - Use simpler meshes with reduced vertex count and attribute precision
    - Use texture compression and mipmapping.
2. For CPU processing, Arm-based devices often use multiple different CPU designs, using big.LITTLE or DynamIQ technology. These designs mix “big” cores, which have high performance, and “LITTLE” cores, which are slower but much more energy efficient. When thermally bound you should aim to get as much work as possible running on the “LITTLE” cores, by using a mixture of optimization and multi-threading.
3. For GPU processing, reduce the precision of computation in shaders; `mediump` uses half the energy per operation of a `highp` operation.

# 15. Improving non-fragment bound content

If you have established that your content that is not fragment bound, it might be failing to hit its target performance due to high vertex processing or compute demands, such as:

- High vertex count
- Large vertex data size
- Vertex shader complexity
- Expensive compute shaders

Vertices are one of the most expensive inputs into a render, so it is important that they are used efficiently. Typically, each vertex requires between 32 and 64 bytes of input attribute data and high-precision shader processing to accurately compute their position. You should aim to keep triangles as large as possible to amortize the high per-vertex shader processing and memory bandwidth cost.

Here are some recommended best practices for reducing the CPU cost of rendering operations:

1. Use dynamic mesh level-of-detail to dynamically select a suitable mesh triangle density based on the distance between the object and the camera.
2. Use pseudo-geometry techniques, such as normal mapping, to replace mesh geometry with textures and shader computation.
3. Use higher densities of triangles only to enhance areas that normal maps do not, such as silhouette edges of objects.
4. Use smaller data types such as half-float, and minimize padding, to reduce the number of bytes of data per vertex.
5. Separate position-related attributes from those related to non-position calculations and store them tightly-packed in separate buffer regions. This maximizes the bandwidth savings from the Mali index-driven vertex shading scheme.

### Caveats

In addition to the direct cost of vertex shading, complex meshes can also reduce fragment shading efficiency. This is because the cost incurred per triangle - such as rasterization and vertex data fetch - is not amortized over very many fragments. In addition, small triangles are more likely to only partially cover each 2x2 pixel quad used by fragment shading, meaning that more quads must be shaded to achieve the same screen coverage.

# 16. Related information

Useful information and shortcuts to next steps.

- Graphics developer guides
- Graphics and gaming blogs
- Ask a question on the Arm graphics and gaming forum