# arm

# Android performance triage with Streamline

8.8

# Tutorial

# Android performance triage with Streamline

## Tutorial

## Release information

**Document history**

| Issue | Date | Confidentiality | Change |
|-------|------|-----------------|--------|
| 0100-01 | 24 November 2021 | Non-Confidential | Initial release |
| 0808-00 | 6 October 2023 | Non-Confidential | Updates for Streamline 8.8 and general improvements. |

## Proprietary Notice

## 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

Use Arm Streamline to capture and analyze the performance counter activity from a mobile device as it runs your Android application. This topic describes how to interpret the charts in Streamline to identify patterns of CPU and GPU activity that could indicate a performance problem.

## Capture a performance profile

First, follow the get started instructions to learn how to capture a performance profile of your mobile application with Streamline.

## Problem areas to analyze

Here are some problem areas you can analyze with Streamline:

- Analyzing CPU load
- Analyzing GPU load
- Analyzing memory bandwidth
- Primitive culling
- Depth (Z) and stencil (S) testing
- Overdraw
- Analyzing shaders
- Analyzing texture performance

## Resources

Here are some resources to help you interpret the data:

- Arm GPU performance counter documentation
- Arm GPU datasheet
- Arm GPU Training Episode 3-3, Streamline.

# 2. Analyzing CPU load

In Streamline, you can see the overall usage of the CPU by observing the activity on each CPU cluster and core in the system, and how that workload is split across threads.

## About CPU clusters

The CPU in a smartphone is made up of a number of cores, a cluster of large and powerful cores, such as the Cortex-A73, and a cluster of smaller and efficient cores, such as the Cortex-A53. Heavy tasks that require a lot of processing power are handled by the powerful cores, and smaller tasks by the more efficient cores. This enables the device to be both high performing and energy efficient (for more information, see big.LITTLE).

## CPU activity

The CPU Activity charts show the activity of each processor cluster, presented as the percentage of each time slice that the CPU was running. Expand each chart to show the individual cores present inside the cluster. Note that this is the percentage of the time slice at the CPU frequency that is being used, not as a percentage of peak performance.

**Figure 2-1: Streamline CPU activity chart**



If an application is CPU bound, quite often the bottleneck is caused by a single thread that is running all of the time. The thread activity panel below the charts can be used to see when each application thread is running. Select threads in this thread view to filter the CPU charts and the Call Paths and Functions views by those threads.

**Figure 2-2: Using the calipers in Streamline**
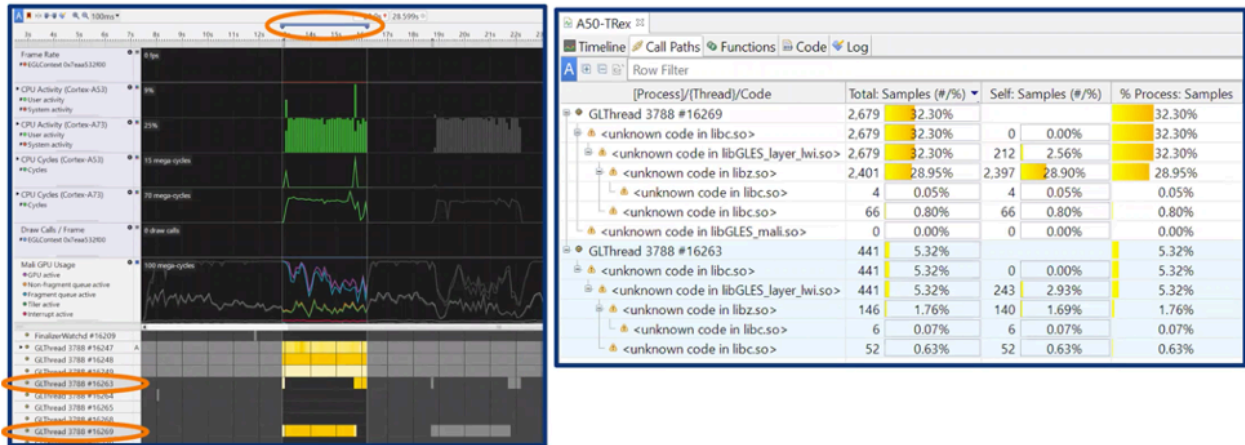


Scheduling bound applications, where neither CPU nor GPU is busy all of the time due to poor synchronization, can been seen in this view as activity oscillating between the impacted CPU thread and the Mali GPU. The CPU thread will block and wait for the GPU to complete, and then the GPU will go idle waiting for the CPU to submit more work to process. Read more information about workload pipelining.

If you identify that the CPU is causing performance issues in your application, use the Call Paths or Functions views to look for function-level hotspots in your code. The Functions view lists all functions that were called during the capture session alongside sample, instance, and usage data. If you used the caliper controls to filter data in the Timeline view, the data in the Functions view reflects this selection.

**Figure 2-3: Data in the functions tab**

# 3. Analyzing GPU load

Analyze the overall usage of the GPU by observing the activity on the GPU processing queues, and the workload split between non-fragment and fragment processing. GPU workloads run asynchronously to the CPU, and the fragment and non-fragment queues can run in parallel to each other, provided that sufficient work is available to process. The charts in Streamline can be used to determine if an application is GPU bound, as they show if the GPU is being kept busy, and the workload distribution across the two main processing queues.

## Mali GPU usage

It is useful to check what frequency the GPU is running at. Highlight a 1 second region of your capture, and look at the Mali GPU usage chart. In this example, the GPU is active for 946 mega-cycles or 946 million cycles per second (946MHz) which is the maximum frequency for the device we tested here. On a higher-end device this number might be much lower, as the GPU may have more shader cores, each running at a lower frequency to achieve the same output more efficiently.

**Figure 3-1: Streamline GPU usage**
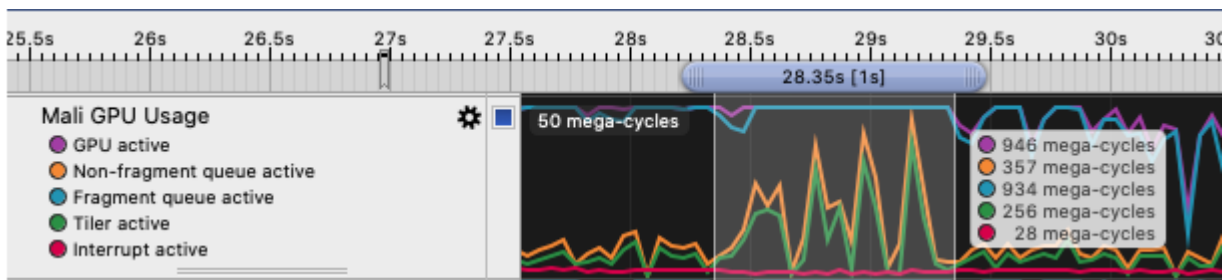


Look for areas where the GPU is active at the maximum frequency. This indicates that the application is GPU bound. These will look like flat lines, where there is no idle time. GPU active cycles will be approximately equal to the dominant work queue (non-fragment or fragment). Fragment work includes all fragment shading work, and non-fragment work includes all vertex shading, tessellation shading, geometry shading, fixed function tiling, and compute shading. For most graphics content there are significantly more fragments than vertices, so the fragment queue will normally have the highest processing load.

Check if drops in GPU activity correlate with spikes in CPU load, and whether those spikes are caused by a particular application thread. Use the calipers in Streamline to select the region where CPU spike occurs, then look at the Call Paths and Functions views to see which threads are active during the spike. If you don't have debug symbols, then you will only see library names in these views, but this can be enough to work out what's going on, because you can see which libraries are being accessed by each thread.

---

**Note**

If you see lots of time spent in `libGLES_mali.so`, this is driver overhead often caused by high draw call counts, bulk data upload, or shader compilation and linking.

---

**Figure 3-2: Filtering with the calipers**
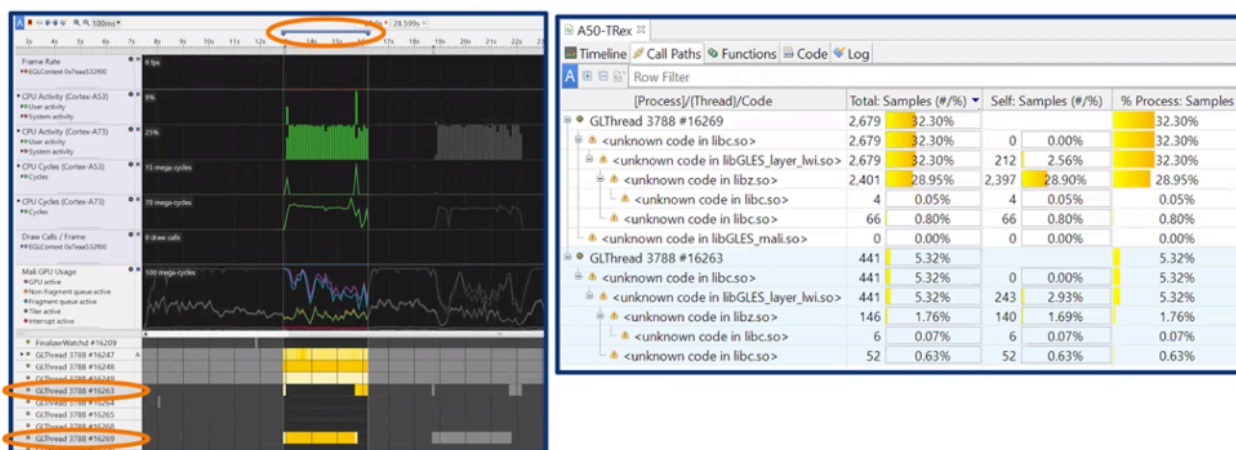


## Workload scheduling

Fragment and non-fragment workloads should overlap. If you see areas where one queue goes idle while the other is active, you could have a serialization problem. This can happen where there are data dependencies, and Vulkan applications can suffer from this. Refer to Workload pipelining and Pipeline bottlenecks for more information.

# 4. Analyzing memory bandwidth

Minimizing GPU memory bandwidth is always a good optimization objective because memory accesses to external DRAM are very power intensive. A good rule of thumb is 100mW per GB/s of bandwidth used.

**Figure 4-1: Memory rule of thumb**
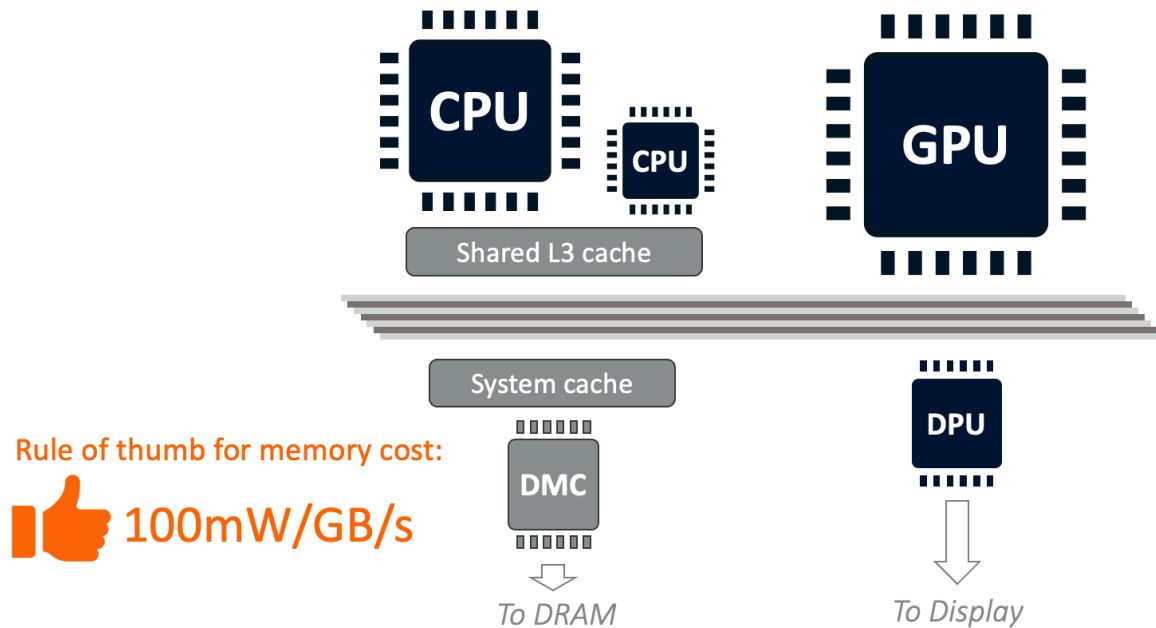


## Mali memory bandwidth

The Mali Memory Bandwidth chart shows the amount of memory traffic between the GPU and the downstream memory system. Depending on the device, these accesses may go directly to external DRAM, or may be sent through additional levels of system cache outside of the GPU.

**Figure 4-2: Mali memory bandwidth chart**
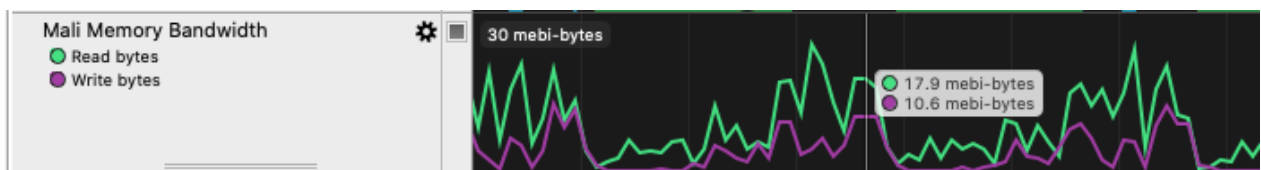


## Mali memory stall rate

The Mali Memory Stall Rate chart shows the memory stall rate seen by the GPU when attempting to make accesses to the downstream memory system.

**Figure 4-3: Mali memory stall rate chart**



Some stalls are expected, particularly for high-end devices that have lots of shader cores all trying to write data in parallel. This is because the GPU can read and write data faster than the memory system can provide it. If you see that the stall rate is constant at around 30% or higher, this is indicative of content which needs optimizing.

## Mali memory read latency

The Mali Memory Read Latency chart shows the memory read latency rate seen by the GPU when making memory system accesses. The chart splits the data so that you can see how many data beats were returned more than a certain number of cycles after the transaction started. A high read latency over 256 cycles could indicate that your content is requesting more data than the memory system in the device can provide.

**Figure 4-4: Mali memory read latency chart**
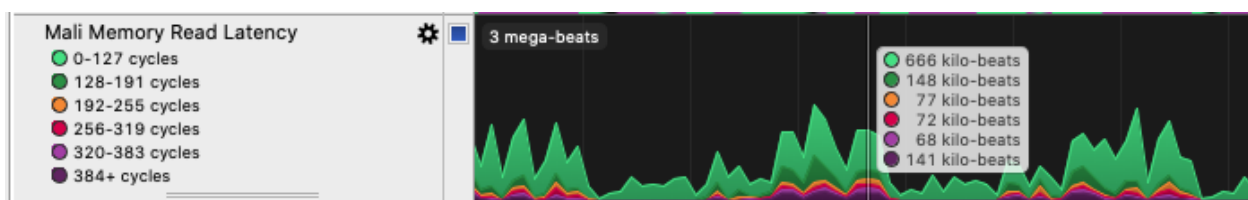


Memory read latency is a property of the device, so the only way to reduce it is to reduce the amount of data being requested.

# 5. Primitive culling

When a GPU processes geometry, it works out which primitives are going to be visible inside the view volume, and discards any that are not. This reduces the amount of pixel processing work required. To decide which primitives can be culled, the GPU performs the following tests:

- Frustum (or Z plane) test. Objects that are outside the camera's near and far clip planes can be discarded because we can guarantee they have no visible impact on screen.

**Figure 5-1: Frustum test showing near and far clip planes**
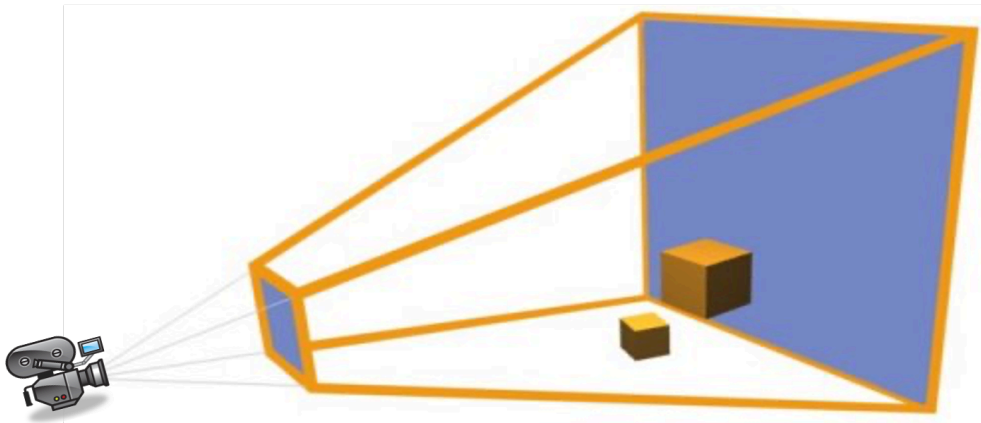


- Facing (or XY plane) test. Triangles that are inside the camera's near and far clip planes, but facing away from the camera are known as 'back facing' triangles. Only one side of each triangle is visible, representing the outer surface of an object. The other side is almost always invisible, because it's inside the object being rendered.

**Figure 5-2: Front and back facing triangles**
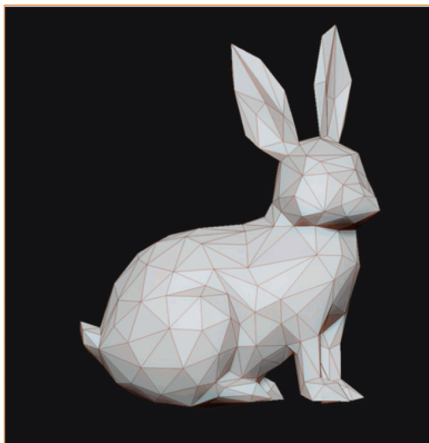
- Sample test. If primitives pass the frustum and facing tests, they are rasterized into 2x2 pixel patches, called quads, prior to fragment shading. Triangles have to hit at least one rasterization sample point to be considered visible. Triangles that don't hit any sample points are too small to be rasterized and are discarded.

**Figure 5-3: Rasterization sample points**
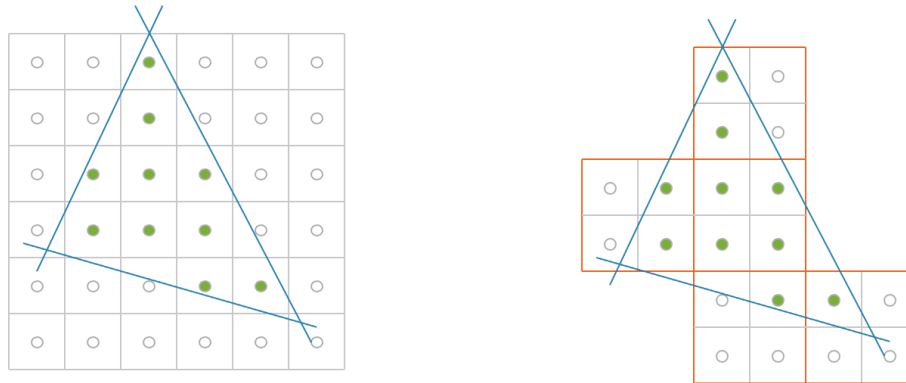


## Mali geometry usage

Use the Mali Geometry Usage chart to check how many primitives were sent to the GPU for processing, how many were visible on-screen, how many were culled.

**Figure 5-4: Mali geometry usage**
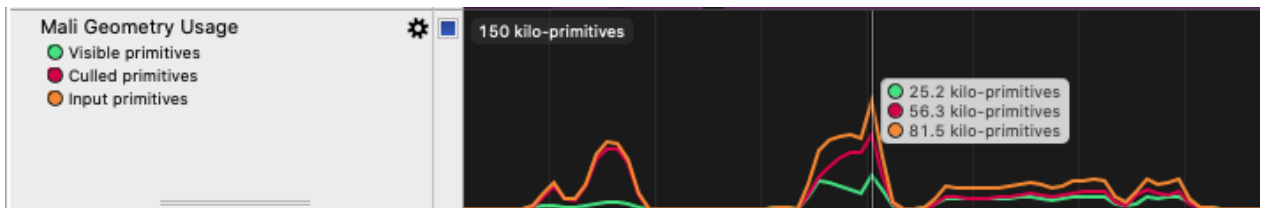


## Mali geometry culling rate

Use the Mali Geometry Culling Rate chart to see how much work was culled by each of the different tests. The values show the percentage of primitives that entered that stage, that were culled by it. These values are not expected to total 100%.

**Figure 5-5: Mali geometry culling rate**
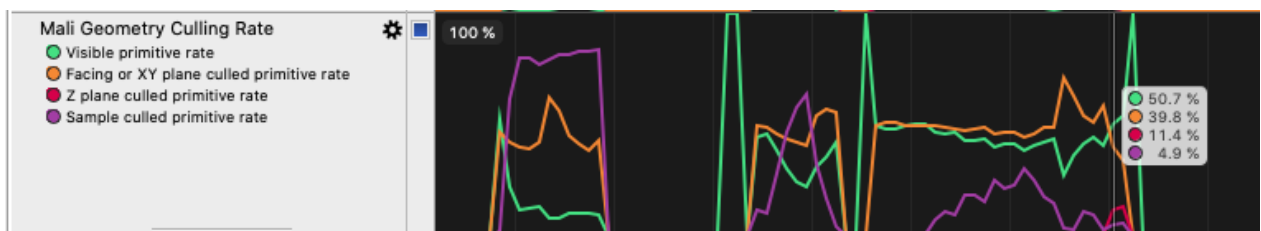
Because every triangle has a back-face, you should expect around 50% of input primitives to be culled by the facing test.

If you see more than 5% of primitives being culled by the sample test, this could indicate that your object meshes could be too complex. Even though the GPU can cull triangles that don't hit any sample points, there may still be many triangles that hit one or two sample points, which then need to be processed using the full quad. These triangles are disproportionately expensive for a GPU to process, as they make shader execution far less efficient. To reduce the number of small triangles, use mesh level-of-detail to force objects to use a simpler mesh when they are further away from the camera.

---

> **Tip**
>
> Use Graphics Analyzer to check the complexity of objects compared with their size on screen.

---

Geometry that passes the frustum, facing and sample tests, is sent through to be fragment shaded, where the GPU colors in the pixels to produce the final on-screen output. However it may still be culled by early or late depth and stencil testing.

# 6. Depth (Z) and stencil (S) testing

Depth (Z) and stencil (S) testing discards pixels that are hidden by other objects or stencil masks. There are two stages in the pixel processing pipeline, where this can happen. Early ZS testing takes place before fragment shading, and late ZS testing takes place after fragment shading.
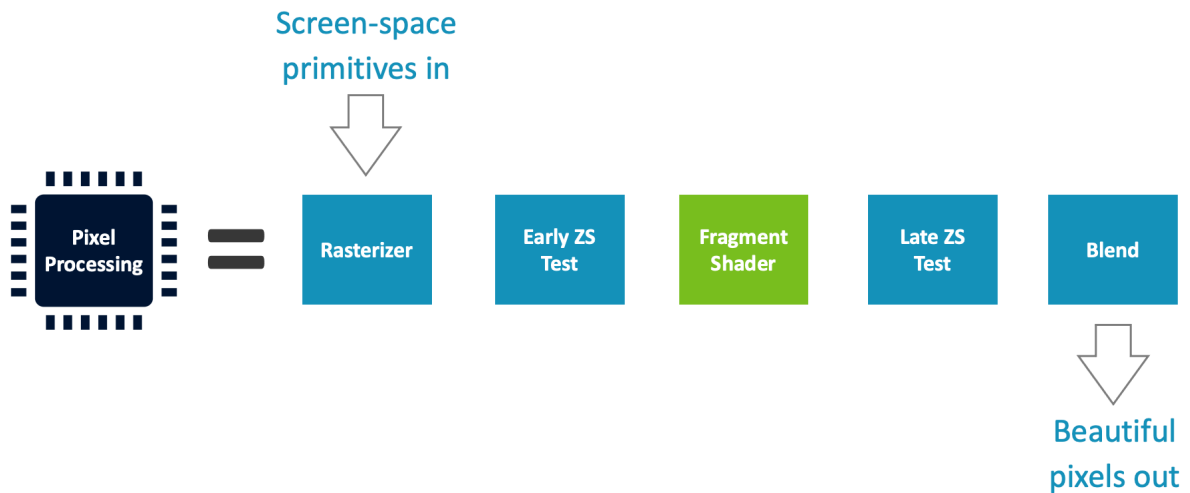


{width=500px}

### Early ZS test

Early ZS testing is relatively inexpensive, because it happens before any pixels are colored in by fragment shading. To get the benefit of early ZS, your application must pass in geometry in a front-to-back render order, starting at the point closest to the camera and moving further away. Use the Mali Early ZS Rate chart in Streamline to check how much work is culled by early ZS testing.

**Figure 6-1: Mali Early ZS Rate**



If your application must pass in geometry in a back-to-front order, early ZS testing can not be used. However, Mali GPUs can use hidden surface removal, also known as forward pixel kill (FPK), to stop rendering of objects that are behind other opaque objects. However, this still has a cost, so try to avoid relying on this optimization if you are able to take advantage of early ZS testing.

### Late ZS test

The late ZS test handles any work that couldn't be processed before fragment shading. For example, if a shader programmatically modifies `gl_FragDepth`, early ZS testing cannot be used, and late ZS testing is forced. Late ZS testing happens after all the fragment shading work has

been done, therefore it is expensive and should be avoided. Use the Mali Late ZS Rate chart in Streamline to check how much work is culled by late ZS testing.

**Figure 6-2: Mali Late ZS Rate**



| | If shader programs modify their coverage mask, for example, by using a `discard` statement, or by using alpha-to-coverage, they must use a late ZS update. Similarly, if fragment shaders write to `gl_FragDepth`, the depth value cannot be known early, and the fragment will be forced to late-ZS test and update. You can check the behavior of your shader programs using Mali Offline Compiler, which reports whether your shader program uses a late ZS test and update. |

# 7. Overdraw

Overdraw occurs in graphics applications where scenes are built using multiple layers of objects that overlap, and are rendered in a back-to-front order. High levels of overdraw can cause poor performance on some devices, where pixels are unnecessarily shaded multiple times.

## Mali overdraw

The Mali Overdraw chart shows the number of fragments shaded per output pixel. Ideally this number should be less than 3.

**Figure 7-1: Mali overdraw chart in Streamline**



This video explains the problem and provides some tips on how to avoid it.

Refer to our optimization advice for reducing overdraw. If the rendering order of objects and blending levels are acceptable, the next step is to check whether you have complex meshes that may result in very small triangles. With bigger triangles, Mali GPUs can use hidden surface removal (FPK) to stop rendering of objects that are behind other opaque objects. However, if triangles are too small to hit any sample points during rasterization, they can not be used as hidden surface occluders during FPK, and this can result in more overdraw. Check the FPK kill rate in the Early ZS chart to see if this number is low.

## Explore overdraw further with Graphics Analyzer

Use Graphics Analyzer to explore your object geometry in more detail. You can capture the level of overdraw in a scene, and explore it to see which objects are causing the problem.

**Figure 7-2: Viewing overdraw in Graphics Analyzer**
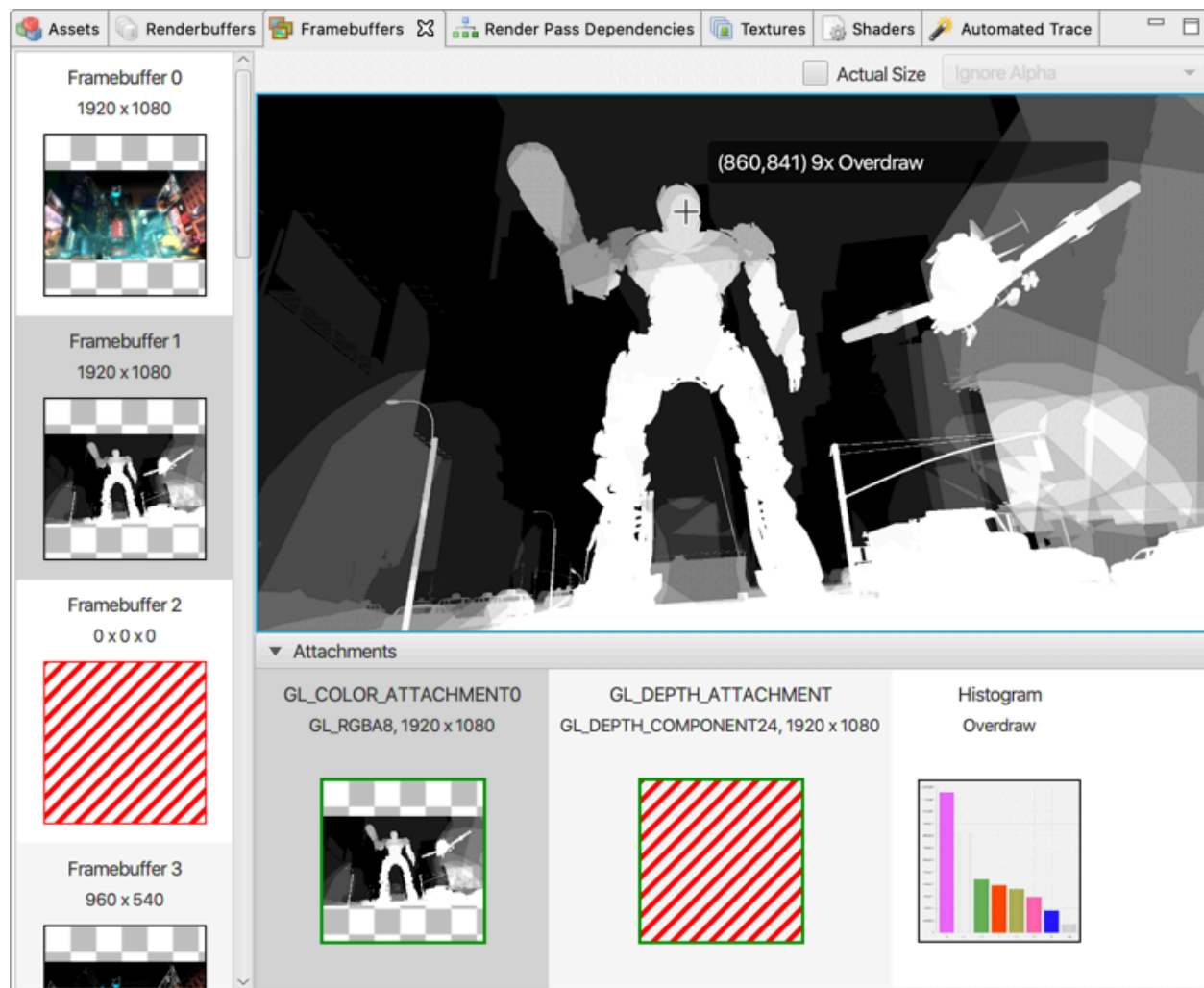
# 8. Analyzing shaders

If shader programs are maths-heavy, texture-heavy or prevent the GPU from applying optimizations, you could see performance problems. Streamline provides a range of charts to help you analyze shader behavior.

In order to be efficient, shader cores within a GPU should:

- Execute calculations at moderate precision - in most cases, `mediump` (16-bit precision) is sufficient.
- Be kept busy with work to process.

## Mali core warps

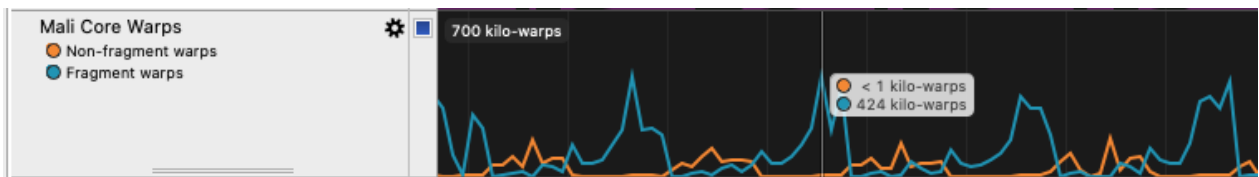The Mali Core Warps chart shows the number of warps created for fragment and non-fragment workloads. Non-fragment workloads include vertex shading, geometry shading, tessellation shading, and compute shading. Each warp represents N threads of shader execution running in lock-step. The value of N - the warp width - varies for different GPUs. Download the Mali GPU datasheet to check this value for different GPUs.

**Figure 8-1: Mali Core Warps**



## Mali core throughput

The Mali Core Throughput chart shows the average number of cycles it takes to get a single thread shaded by the shader core. Note that this chart shows average throughput, not average cost, so includes impacts of processing latency and of resource sharing across the two workload types.

**Figure 8-2: Mali Core Throughput**



## Mali core utilization

The Mali Core Utilization chart shows the activity levels of the major data paths in the shader core. It tells you the number of cycles where there is something running in the shader core, but doesn't tell you how busy it is. This chart can indicate which type of workload could be causing a performance problem, and whether there are any scheduling issues.

**Figure 8-3: Mali Core Utilization**



Fragment FPKB utilization counts the percentage of cycles where the forward pixel kill (FPK) quad buffer, before the execution core, contains at least one quad. Quads in this buffer are queued, waiting to be shaded, and should remain fairly high. If this number drops off significantly, only showing a small percentage of cycles that have queued quads waiting to run, there could be either dependencies on early ZS (you have things that have to be early ZS tested, and can't be pushed to late ZS) or the queue is draining faster than it is filling. This can happen if there are a number of very small triangles, which are being shaded faster than they can be rasterized.

If the execution core utilization is low, this indicates possible lost performance, because there are spare shader core cycles that could be used if they were accessible. In some use cases this is unavoidable, because there are regions in a render pass where there is no shader workload to process. For example, a clear color tile that contains no shaded geometry, or a shadow map that can be resolved entirely using early ZS depth updates. If screen regions contain high volumes of redundant geometry, this can cause the programmable core to run out of work to process because the fragment front-end can not generate warps fast enough. This can happen if a high percentage of triangles are killed by ZS testing or FPK hidden surface removal, or by a very high density of microtriangles which each generate low numbers of threads.

## Mali core unit utilization

The Mali Core Unit Utilization chart shows the percentage utilization of the functional units inside the shader core; the execution engine, the varying unit, the texture unit and the load/store unit. The most heavily utilized functional unit should be the target for optimizations to improve performance, although reducing load on any of the units is good for energy efficiency.

**Figure 8-4: Mali Core Unit Utilization**
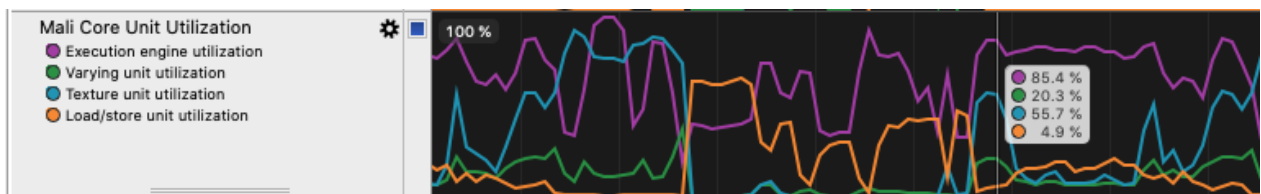


If the texture unit utilization is a bottleneck, check the texturing charts to look for ways to optimize.

## Mali core pipe utilization

For Valhall-based GPUs such as the Mali-G77, an additional Mali Core Pipe Utilization chart shows the breakdown of execution engine activity between the FMA (fused multiply-accumulate), CVT (convert), SFU (special functions unit) and MSG (message) pipes.

**Figure 8-5: Mali Core Pipe Utilization**
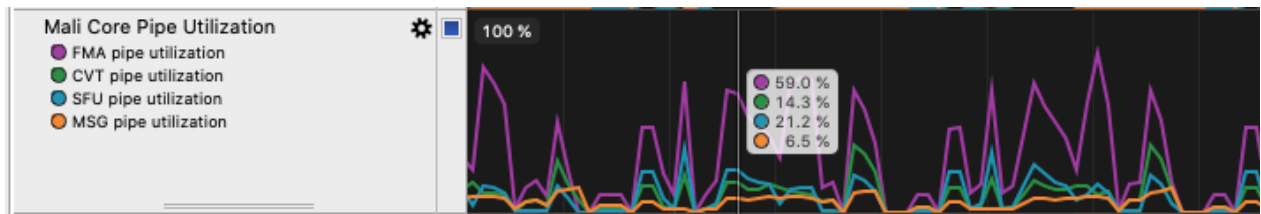


## Mali core workload property rate

The Mali Core Workload Property Rate gives information about shader workload behavior that could be optimized:

- Partial coverage - Warps that contain samples with no coverage. A high number suggests that content has a high density of very small triangles, or microtriangles, which are disproportionately expensive to process.

- Diverged instructions - Instructions that have control flow divergence across the warp.

- Constant tile kill - Tile writes that are killed by the transaction elimination CRC check. A high number indicates that a significant part of the framebuffer is static from frame to frame.

**Figure 8-6: Mali Core Workload Property rate**



## Mali varying usage

The Mali Varying Usage chart shows the amount of interpolation processed by the varying unit, at 16-bit or 32-bit precision.

**Figure 8-7: Mali Varying usage**
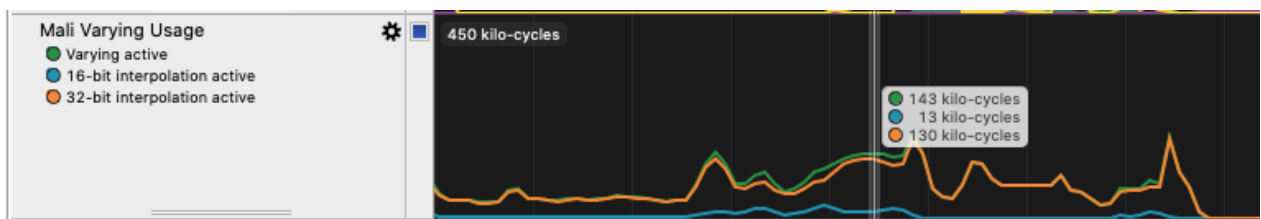


16-bit interpolation is twice as fast as 32-bit interpolation. It is recommended to use `mediump` (16-bit) varying inputs to fragment shaders, rather than `highp` whenever possible.

## Mali Offline Compiler

You can also use Mali Offline Compiler to generate an offline analysis report of how your shader program performs on a Mali GPU. It provides a cycle cost breakdown for the shader's arithmetic, load/store, varying and texture usage. You can check the percentage of arithmetic operations

that are efficiently performed at 16-bit precision or lower, and you can see information about the shader's use of language features that can impact the performance of shader execution.

**Further training**

To learn more about how to optimize shaders, watch the Mali GPU training Episode 2-6, Shaders best practices.

For more information about writing optimal shader code, refer to the Arm GPU best practices guide.

# 9. Analyzing texture performance

The texturing charts in Streamline show you whether texturing is active, and which texture filtering is in use. If your content is making heavy use of the texture unit, consider using simpler texture filters. This video explains the differences between different types of filtering.

---

💡 **Tip**   Check the Arm GPU datasheet for details about the texturing capabilities of the GPU in the device you are testing.

---

## Mali texture usage

The Mali Texture Usage chart shows the number of cycles where texturing was active. Some instructions take more than one cycle due to multi-cycle data access and filtering operations. Mali is optimized for 2D bilinear filtering, trilinear filtering runs at half speed and 3D filtering runs at half speed again. The cycle costs per quad vary for different GPUs, refer to the Mali GPU datasheet for details.

**Figure 9-1: Mali texture usage chart for Bifrost GPUs in Streamline**



For Valhall-based GPUs, you can also see where the filtering unit uses 4x bilinear or 2x trilinear filtering.
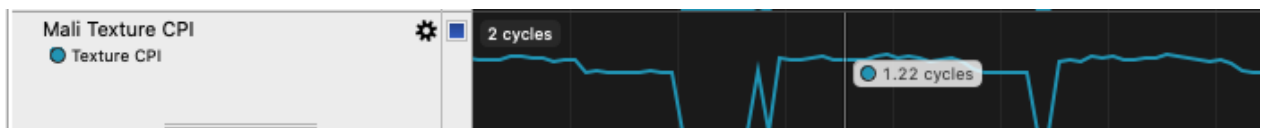
**Figure 9-2: Mali Texture usage chart for Valhall GPUs in Streamline**
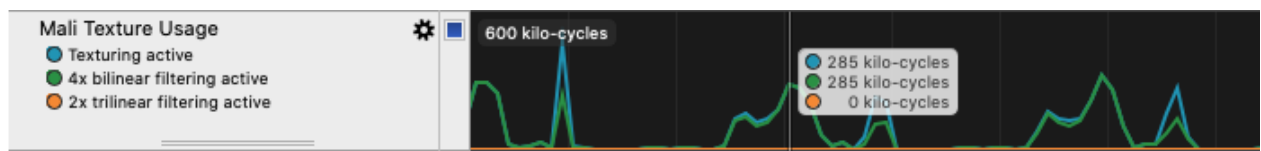


## Mali texture CPI

The Mali Texture CPI (cycles per instruction) chart shows the average number of texture cycles per sample.

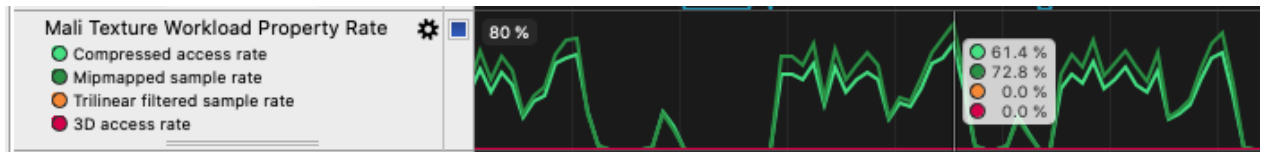**Figure 9-3: Mali Texture CPI chart in Streamline**

## Mali texture workload property rate

The Mali Texture Workload Property Rate chart shows the percentage of texture operations that use texture compression, mipmapping, trilinear filtering or 3D accesses.

**Figure 9-4: Mali texture workload property rate chart in Streamline for Bifrost GPUs**



For Valhall-based GPUs, you can see the percentage of texture filtering cycles that use the 4x data path.

**Figure 9-5: Mali texture workload property rate chart in Streamline for Valhall GPUs**
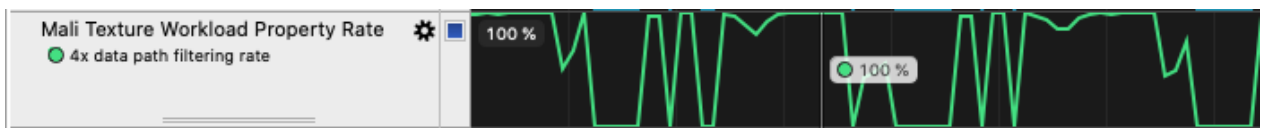


## Mali texture bus utilization

For Valhall-based GPUs, you can see the percentage load on the texture message input bus. Sometime bus rate can be an issue if you're returning `highp` samplers (`fp32` per channel not `fp16`). Use this to check for texture bandwidth issues inside the shader core.

**Figure 9-6: Mali texture bus utilization chart in Streamline for Valhall GPUs**



## Mali texture memory usage

The Mali Texture Memory Usage chart shows the memory usage for the texture pipe normalized by the number of access cycles. For every texturing filter access you make, check how much bandwidth comes from the L2, and how much is coming from external memory. Use this to check cache efficiency.

**Figure 9-7: Mali texture memory usage chart in Streamline**



If a high number of bytes are being requested per access, where high depends on the texture formats you are using, it can be worth reviewing texture settings:

- Enable mipmaps for offline generated textures

- Use ASTC or ETC compression for offline generated textures

- Replace run-time generated framebuffer and texture formats with a narrower format

- Reduce any use of negative LOD bias used for texture sharpening

- Reduce the MAX_ANISOTROPY level for anisotropic filtering

Read more texture optimization advice in the Texture Best Practices guide.

### ASTC Encoder

ASTC Encoder is a command-line tool from Arm that compresses and decompresses images using the Adaptive Scalable Texture Compression standard. It is available to download for free on GitHub.

Read the ASTC Format Overview for a quick introduction to the format, or read the full Khronos Data Format Specification for all the details.

# 10. Arm GPU performance counter documentation

Arm GPUs implement a comprehensive range of performance counters, that enable you to closely monitor GPU activity as your application runs. The charts in Streamline visualize this performance counter activity, to help you identify the cause of heavy rendering loads or workload inefficiencies that cause poor GPU performance.

For detailed descriptions of all the performance counters available for each Arm GPU, refer to the Arm GPU performance counter documentation.

# 11.  Mali GPU datasheet

The Mali family of mobile GPUs has evolved over time, as we've added new features and increased capabilities. Each new generation behaves slightly differently, so download the Arm GPU datasheet, which lists all the key statistics. Remember to interpret the charts in Streamline differently based on the GPU in the device. Different generations of Arm GPU architectures have different shader core capacities, with differences such as the width of warps, number of threads and texture filtering capabilities.

## 12. Streamline user guide

For full instructions on how to use Streamline, and information about all the different views and functionality, refer to the Streamline user guide.

Streamline user guide