Arm® Mali™-T860 and Arm® Mali™-T880 Performance Counters

1.1

Reference Guide

Non-Confidential

Issue

108061_0101_en

Arm® Mali™-T860 and Arm® Mali™-T880 Performance Counters

Reference Guide

Release information

Document history

Issue	Date	Confidentiality	Change
1.0	15 June 2023	Non-Confidential	Initial release
1.1	10 August 2023	Non-Confidential	Updated counter guide

Proprietary Notice

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. Arm® Mali™-T860 and Mali-T880 GPU performance counters

This guide explains the GPU performance counters found in the Arm® Streamline profiling template for the Mali™-T860 and Mali-T880 GPUs, which are part of the Midgard architecture family.

The counter template in Streamline follows a step-by-step analysis workflow. Analysis starts with high-level workload triage, measuring the CPU, GPU, and memory bandwidth usage. A detailed analysis of the application rendering workload then reviews how efficiently the available hardware resources are used by the application.

For each counter in the template, this guide documents the meaning of the counter and provides the Streamline variable name or expression associated with it.

Note

The Streamline template only shows a subset of the available performance counters. However, it covers the most common types of GPU performance analysis.

1.1 Counter handling

Arm GPU hardware emits unique counters per shader core and per cache slice. The data presented in Streamline, and the expression equations defined in this document, use the summed value of all of the counter instances.

Note

This behavior changed in Streamline 8.7. In earlier releases, Streamline showed the averaged value for shader core counters and the summed value for cache slice counters.

1.2 Guide content

This guide contains the following sections:

- CPU performance: analyze the overall usage of the CPU by observing the activity on the CPU clusters and cores in the system.
- GPU activity: 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.
- Content behavior: analyze content efficiency by observing the number of vertices being processed, the number of primitives being culled, and the number of pixels being processed.

- Shader core data path: analyze the shader core workload scheduling, and data path throughput.
- Shader core functional units: analyze the overall usage of the shader core. Observe the effectiveness of fragment depth and stencil testing, the number of threads spawned for shading, and the relative loading of the programmable core processing pipelines.
- Shader core texture unit: analyze performance of the texture filtering unit, and how the unit is being used by the shader programs that are running. Use this data to find optimization opportunities for content identified as texture-bound in the shader core functional units section.
- Shader core load/store unit: analyze performance of the load/store unit, and how the unit is being used by the shader programs that are running. Use this data to find optimization opportunities for content identified as load-store-bound in the shader core functional units section.
- GPU configuration: these utility counters expose the GPU configuration of the platform, allowing Streamline to create expressions based on the specific configuration of the connected device.

2. CPU performance

High CPU load or poor scheduling of workloads can cause many graphics performance issues. The first part of the analysis template looks at the CPU workloads, allowing you to identify regions where CPU performance impacts the overall application performance.

The default view for the CPU charts shows the activity of each cluster of CPUs. To see individual CPUs, expand the chart group to show individual cores present inside each cluster.

2.1 CPU activity

CPU activity charts show the usage of each processor cluster, displaying the percentage of each time slice that the CPUs in the cluster were running. This percentage allows you to assess how busy the CPUs were. Note that this metric is only a time-based measure and does not factor in the CPU frequency that was used.

For CPU-bound applications, it is common for a single thread to run all the time and become the bottleneck for overall application performance. The thread activity panel below the counter charts shows when each application thread was running. Selecting one or more threads in this view filters the CPU activity and counter charts to show the load attributed to the selected threads.

For scheduling-bound applications, it is common for both CPU and GPU to go idle due to poor synchronization. The CPU goes idle when it is waiting for the GPU to complete work. The GPU goes idle when waiting for the CPU to submit new work. To identify scheduled bound applications in this view, look for activity that is oscillating between the impacted CPU thread and the GPU.

```
$CPUActivityUser.Cluster[0..N]
```

2.2 CPU cycles

The CPU cycle charts show the activity of each processor cluster, presented as the number of processor clock cycles used. Using this data with the CPU activity information can indicate the CPU operating frequency.

```
$CyclesCPUCycles.Cluster[0..N]
```

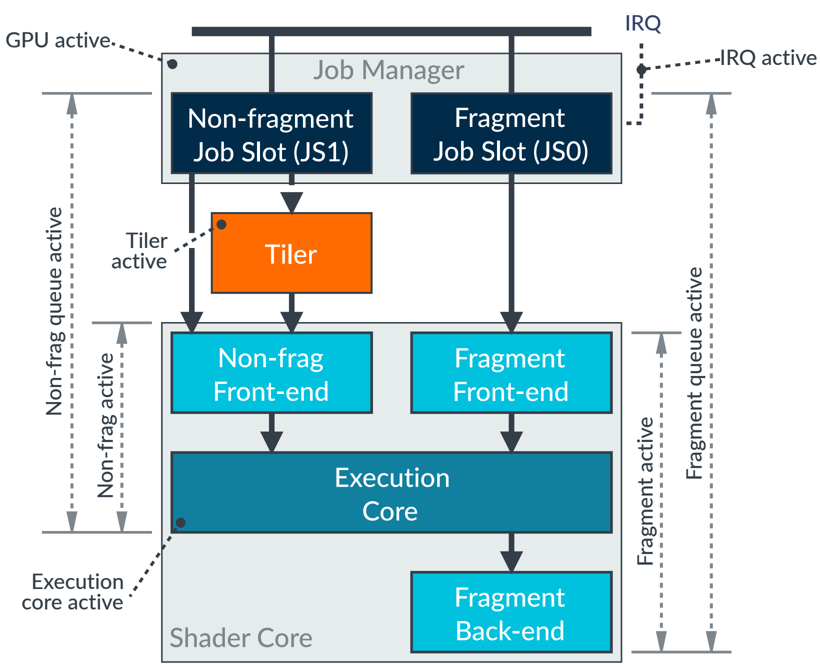
3. GPU activity

The workloads running on this GPU are coordinated by the Job Manager, the hardware unit that is responsible for scheduling workloads onto the GPU. The Job Manager exposes two FIFO work queues to the graphics driver. One queue is used for non-fragment workloads, which include compute shading and vertex shading, and one queue is used for fragment workloads.

These two queues run asynchronously to the CPU and can run in parallel to each other, provided that sufficient non-dependent work is available to run. Keeping the CPU and GPU processing in parallel, and the two queues processing in parallel, is an important goal when optimizing content for Arm GPUs.

The following diagram shows the processing pipeline data paths through the GPU for different kinds of workload. It also shows the performance counters available for each data path or major block in the hierarchy.

Figure 3-1: Midgard GPU top-level

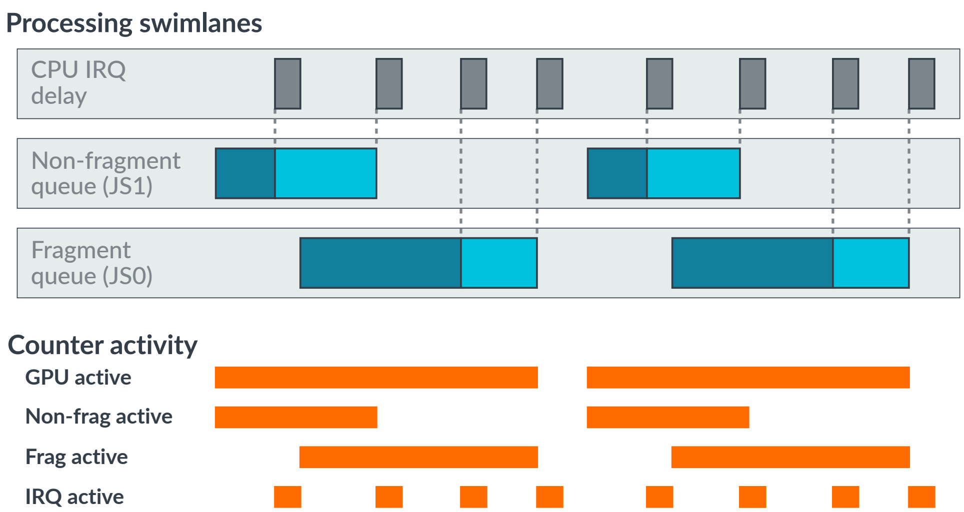


The “active” counters show that a data path or hardware unit processed some workload, but do not necessarily indicate that it was fully utilized. For example, the *Fragment queue active cycles* counter increments every cycle where there is any fragment workload queued to run anywhere in the GPU.

Some counters are common to multiple data paths. For example, both non-fragment and fragment shader programs run on the same unified shader core. If these different workload types are overlapping in the same counter sample, then shader core counter data includes contributions from both.

The following swim lane diagram shows how the top-level GPU counters increment for overlapping render passes.

Figure 3-2: Midgard GPU top-level timeline



This diagram shows two render passes per frame, shown in different shades of blue. Each render pass consists of a single piece of non-fragment work that must be executed before its fragment shading can start. An interrupt is raised back to the CPU at the end of each piece of work on each queue. The *GPU active cycles* counter increments whenever any queue contains work.

3.1 GPU usage

GPU usage counters monitor the overall load on the GPU by measuring the workload submitted to the Job Manager queues. These counters can indicate the dominant workload type submitted by the application, which is a good target for optimization. They can also indicate the effectiveness of workload scheduling at keeping the hardware queues running in parallel.

3.1.1 GPU active cycles

This counter increments every clock cycle where the GPU has any pending workload present in one of its processing queues. It shows the overall GPU processing load requested by the application.

This counter increments when any workload is present in any processing queue, even if the GPU is stalled waiting for external memory. These cycles are counted as active time even though no progress is being made.

```
$MaliGPUCyclesGPUActive
```

3.1.2 Non-fragment queue active cycles

This counter increments every clock cycle where the GPU has any workload present in the non-fragment queue. This queue is used for vertex shaders, tessellation shaders, geometry shaders, fixed-function tiling, and compute shaders. This counter can not disambiguate between these workloads.

In content achieving good parallelism, which is important for overall efficiency of rendering, the highest queue active cycle counter must be similar to the GPU active counter.

This counter increments when any workload is present in the non-fragment processing queue, even if the GPU is stalled waiting for external memory. These cycles are counted as active time even though no progress is being made.

```
$MaliGPUCyclesNonFragmentQueueActive
```

3.1.3 Fragment queue active cycles

This counter increments every clock cycle where the GPU has any workload present in the fragment queue.

In content achieving good parallelism, which is important for overall efficiency of rendering, the highest queue active cycle counter must be similar to the GPU active counter.

This counter increments when any workload is present in the fragment queue, even if the GPU is stalled waiting for external memory. These cycles are counted as active time even though no progress is being made.

```
$MaliGPUCyclesFragmentQueueActive
```

3.1.4 Tiler active cycles

This counter increments every cycle the tiler has a workload in its processing queue. The tiler is responsible for coordinating geometry processing and providing the fixed-function tiling needed for the Mali tile-based rendering pipeline. It can run in parallel to vertex shading and fragment shading.

A high cycle count here does not necessarily imply a bottleneck, unless the *Non-fragment active cycles* counter in the shader core is comparatively low.

```
$MaliGPUCyclesTilerActive
```

3.1.5 GPU interrupt pending cycles

This counter increments every cycle that the GPU has an interrupt pending and is waiting for the CPU to process it.

Cycles with a pending interrupt do not necessarily indicate lost performance because the GPU can process other queued work in parallel. However, if *GPU interrupt pending cycles* are a high percentage of *GPU active cycles*, an underlying problem might be preventing the CPU from efficiently handling interrupts. This problem is normally a system integration issue, which an application developer cannot work around.

```
$MaliGPUCyclesGPUInterruptActive
```

3.2 GPU utilization

The GPU utilization counters provide an alternative view of the data path activity cycles, normalizing the queue usage against the total GPU active cycle count. These metrics provide a clearer view of breakdown by workload type, and the effectiveness of queue scheduling.

For GPU-bound content that is achieving good parallelism, one of the queues is close to 100% utilization, with the other running in parallel to it. Prioritize the most heavily loaded queue for content optimization, as it is the critical path workload.

If the GPU is always busy, but the queues are running serially for all or part of the frame, application API usage might prevent parallel processing. Serial processing reduces the achievable performance. The following actions can cause serial processing:

- The application blocking and waiting for GPU activity to complete, for example, by waiting on a query object result which is not yet available. Waiting on an unavailable query object result can cause one or more of the hardware queues to drain and run out of work to process.
- The application using conservative Vulkan pipeline barriers. For example, submitting using a `STAGE_TOP_OF_PIPE` destination when a `STAGE_FRAGMENT_SHADER` destination would have been sufficient.
- The application submitting rendering workloads that have data dependencies across the queues which prevent parallel execution. For example, if no non-dependent work is available, a fragment-compute-fragment data flow might mean no processing occurs in the fragment queue while the compute shader is running.

Mobile systems improve energy efficiency by using Dynamic Voltage and Frequency Scaling (DVFS) to reduce voltage and clock frequency for light workloads. When seeing a workload with high percentage utilization, check the *GPU active cycles* counter to confirm the frequency. If the workload is light, a highly utilized GPU can run at a low clock frequency.

3.2.1 Non-fragment queue utilization

This expression defines the non-fragment queue utilization compared against the GPU active cycles. For GPU bound content, it is expected that the GPU queues process work in parallel. The dominant queue must be close to 100% utilized. If no queue is dominant, but the GPU is close to 100% utilized, then there might be a serialization or dependency problem preventing better overlap across the queues.

```
max(min(($MaliGPUCyclesNonFragmentQueueActive / $MaliGPUCyclesGPUActive) * 100,
 100), 0)
```

3.2.2 Fragment queue utilization

This expression defines the fragment queue utilization compared against the GPU active cycles. For GPU bound content, the GPU queues are expected to process work in parallel. Aim to keep the dominant queue close to 100% utilized. If no queue is dominant, but the GPU is close to 100% utilized, then there might be a serialization or dependency problem preventing better queue overlap.

```
max(min(($MaliGPUCyclesFragmentQueueActive / $MaliGPUCyclesGPUActive) * 100, 100),
 0)
```

3.2.3 Tiler utilization

This expression defines the tiler utilization compared to the total GPU active cycles.

Note that this metric measures the overall processing time for the tiler geometry pipeline. The metric includes aspects of vertex shading, in addition to the fixed-function tiling process.

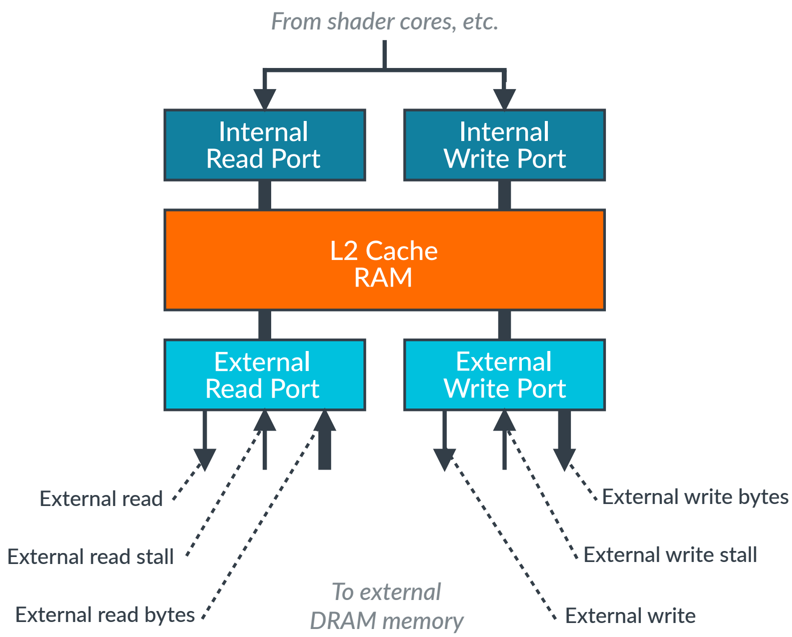
```
max(min(($MaliGPUCyclesTilerActive / $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

3.3 External memory bandwidth

The external memory bandwidth counters show the total memory bandwidth between the GPU and the downstream memory system. Accessing external DRAM is one of the most energy-intensive operations that the GPU can perform, so reducing memory bandwidth is a key optimization goal.

These performance counters measure the memory accesses that are external to the GPU. If there are layers of system cache between the GPU and external DRAM, these accesses might not be external to the system-on-a-chip.

Figure 3-3: Midgard GPU memory system



Memory accesses to external DRAM are very power intensive. A good guideline is that external DRAM access costs between 80mW and 100mW per GB/s of bandwidth used. Assuming a typical 650mW power budget for DRAM access, an application can only sustainably use a total of 100MB per frame at 60FPS. Optimizations that help to minimize GPU memory bandwidth are a high priority for mobile application development.

3.3.1 Output external read bytes

This expression defines the total output read bandwidth for the GPU.

```
$MaliExternalBusBeatsReadBeats * ($MaliConstantsBusWidthBits / 8)
```

3.3.2 Output external write bytes

This expression defines the total output write bandwidth for the GPU.

```
$MaliExternalBusBeatsWriteBeats * ($MaliConstantsBusWidthBits / 8)
```

3.4 External memory stalls

The external memory stall rate counters measure the back-pressure seen by the GPU when it is attempting to make external memory accesses.

A high stall rate is indicative of content which is requesting more data than the downstream memory system can provide. To optimize the workload, try to reduce memory bandwidth.

3.4.1 Output external read stall rate

This expression defines the percentage of GPU cycles with a memory stall on an external read transaction.

Stall rates can be reduced by reducing the size of data resources, such as textures or models.

```
max(min(($MaliExternalBusStallsReadStallCycles / $MaliConstantsL2SliceCount /
 $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

3.4.2 Output external write stall rate

This expression defines the percentage of GPU cycles with a memory stall on an external write transaction.

Stall rates can be reduced by reducing geometry complexity, or the size of framebuffers in memory.

```
max(min(($MaliExternalBusStallsWriteStallCycles / $MaliConstantsL2SliceCount /
 $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

4. Content behavior

Optimal rendering performance requires both efficient content, and efficient handling of that content by the GPU. The content behavior metrics help you to supply the GPU with efficiently structured content.

Slow rendering performance has three common causes:

- Content which is efficiently written, but doing too much processing given the capabilities of the target device.
- Content which is inefficiently written, with redundancy in the workload submitted for rendering.
- Application API usage which triggers high workload, or causes idle bubbles, due to GPU-specific or driver-specific behaviors.

This section of the Streamline template aims to focus on the first two of these causes. It looks at the size and efficiency of the submitted workload.

4.1 Geometry usage

The vertex stream is the first application input processed by the GPU rendering pipeline. These counters monitor the amount of geometry being processed, and how much is discarded due to culling.

Geometry is one of the most expensive inputs to the GPU, as vertices typically need 32-64 bytes of input data and data access is expensive. To avoid dense geometry appearing on-screen, use simpler meshes when objects are further away from the camera. You can use compressed normal maps as an efficient alternative to high geometric detail.

4.1.1 Total input primitives

This expression defines the total number of input primitives to the rendering process.

```
$MaliPrimitiveCullingFacingAndXYPlaneTestCulledPrimitives
 + $MaliPrimitiveCullingZPlaneTestCulledPrimitives +
 $MaliPrimitiveCullingVisiblePrimitives
```

4.1.2 Culled primitives

This expression defines the number of primitives that were culled during the rendering process, for any reason.

For 3D content, it is expected that approximately 50% of the primitives are culled due to the facing test. If a significantly higher percentage is culled, then the GPU performance is being lost shading objects which are not visible. In this scenario, review the efficiency of CPU-side culling techniques and check for overly large batch sizes.

```
$MaliPrimitiveCullingFacingAndXYPlaneTestCulledPrimitives +
 $MaliPrimitiveCullingZPlaneTestCulledPrimitives
```

4.1.3 Visible primitives

This counter increments for every visible primitive that survives all culling stages.

Visible means only that the primitive is front-facing and inside the visible clip volume. If the primitive is occluded by other primitives closer to the camera, the primitive might produce no visible output.

Application software techniques, such as portal culling, can often be used to efficiently cull occluded objects inside the frustum. Culling these occluded objects can reduce the amount of redundant vertex processing that the GPU has to do.

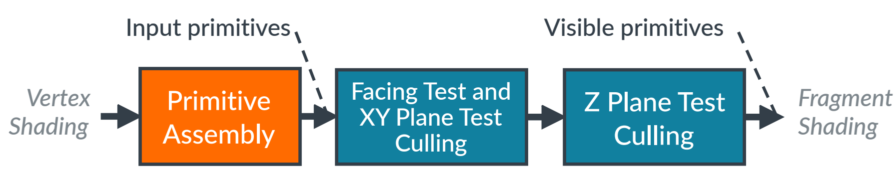
```
$MaliPrimitiveCullingVisiblePrimitives
```

4.2 Geometry culling

The GPU must compute positions of primitives before they can enter the culling stages. Culled geometry can have a significant processing and bandwidth cost, even though it contributes no useful visual output. These counters help to identify the reasons why primitives are culled, allowing you to correctly target optimizations at the area causing problems.

The culling pipeline for this GPU executes in the order shown in the following diagram. The counters for this pipeline show the percentage of the primitives entering a stage that the stage culls. Because these percentages are relative to the per-stage input, not the total geometry input, they do not add up to 100%.

Figure 4-1: Midgard GPU culling pipeline



4.2.1 Visible primitives rate

This expression defines the percentage of primitives that are visible after culling.

For 3D content, it is typical that 50% of primitives are visible, because of the use of back-face culling. Significantly lower visibility rates can indicate missing optimizations.

```
max(min(($MaliPrimitiveCullingVisiblePrimitives /
 ($MaliPrimitiveCullingFacingAndXYPlaneTestCulledPrimitives
 + $MaliPrimitiveCullingZPlaneTestCulledPrimitives +
 $MaliPrimitiveCullingVisiblePrimitives)) * 100, 100), 0)
```

4.2.2 Facing or XY plane test cull rate

This expression defines the percentage of primitives entering the facing and XY plane test that are culled by it. Primitives that are outside of the view frustum in the XY axis, or that are back-facing inside the frustum, are culled by this stage.

It is expected that approximately 50% of primitives are culled at this stage. These triangles are in-frustum, but back-facing. If more than 50% of input primitives are culled because of being out-of-frustum, then there might be opportunity to improve software culling or batching granularity.

```
max(min(($MaliPrimitiveCullingFacingAndXYPlaneTestCulledPrimitives /
 ($MaliPrimitiveCullingFacingAndXYPlaneTestCulledPrimitives
 + $MaliPrimitiveCullingZPlaneTestCulledPrimitives +
 $MaliPrimitiveCullingVisiblePrimitives)) * 100, 100), 0)
```

4.2.3 Z plane test cull rate

This expression defines the percentage of primitives entering the Z plane culling test that are culled by it. Primitives that are closer than the frustum near clip plane, or further away than the frustum far clip plane, are culled by this stage.

Seeing a significant proportion of triangles culled at this stage can be indicative of insufficient application software culling.

```
max(min(($MaliPrimitiveCullingZPlaneTestCulledPrimitives /
 (($MaliPrimitiveCullingFacingAndXYPlaneTestCulledPrimitives
 + $MaliPrimitiveCullingZPlaneTestCulledPrimitives +
 $MaliPrimitiveCullingVisiblePrimitives) -
 $MaliPrimitiveCullingFacingAndXYPlaneTestCulledPrimitives)) * 100, 100), 0)
```

4.3 Fragment overview

Fragment overview counters look at the requested pixel processing workload. These counters can show the total number of output pixels shaded, the average number of cycles spent per pixel, and the average overdraw factor.

It is a useful exercise to set a cycle budget for an application, measured in terms of cycles per pixel. Compute the maximum cycle budget using this equation:

```
shaderCyclesPerSecond = MaliCoreCount MaliFrequency
pixelsPerSecond = Screen_Resolution * Target_FPS

// Maximum cycle budget assuming perfect execution
maxBudget = shaderCyclesPerSecond / pixelsPerSecond

// Real-world cycle budget assuming 85% utilization
realBudget = 0.85 * maxBudget
```

Setting a cycle budget helps manage expectations of what is possible. For example, consider a mass-market device with a 3 core GPU running at 500MHz. At 1080p60 this device has a cycle budget of just 10 cycles per pixel. This budget must cover all processing costs, including vertex shading and fragment shading. If you want to achieve the best graphics fidelity, you must ensure you spend each cycle wisely.

4.3.1 Pixels

This expression defines the total number of pixels that are shaded by the GPU, including on-screen and off-screen render passes.

This measure can be a slight overestimate because it assumes all pixels in each active 16x16 pixel region are shaded. If the rendered region does not align with 16 pixel aligned boundaries, then this metric includes pixels that are not actually shaded.

```
$MaliGPUTasksFragmentTasks * 256
```

4.3.2 Cycles per pixel

This expression defines the average number of GPU cycles being spent per pixel rendered, including any vertex shading cost.

It can be a useful exercise to set a cycle budget for each render pass in your game, based on your target resolution and frame rate. Rendering 1080p at 60 FPS is possible in a mass-market device, but the number of cycles per pixel you have to work with can be small. Those cycles must be used wisely to achieve a 60 FPS performance target.

```
$MaliGPUCyclesGPUActive / ($MaliGPUTasksFragmentTasks * 256)
```

4.3.3 Fragments per pixel

This expression computes the number of fragments shaded per output pixel.

GPU processing cost per pixel accumulates with the layer count. High overdraw can build up to a significant processing cost, especially when rendering to a high-resolution framebuffer. Minimize overdraw by rendering opaque objects front-to-back and minimizing use of blended transparent layers.

```
$MaliCoreThreadsFragmentThreads / ($MaliGPUTasksFragmentTasks * 256)
```

4.4 Fragment depth and stencil testing

It is important that as many fragments as possible are early ZS (depth and stencil) tested before shading. Removing redundant work at this stage is more efficient than testing and killing fragments later using late ZS. These counters monitor the number of early and late test and kill operations performed.

To maximize the efficiency of early ZS testing, Arm recommends drawing opaque objects starting with the objects closest to camera and then working further away. Render transparent objects from back-to-front in a second pass.

4.4.1 Early ZS tested quad percentage

This expression defines the percentage of rasterized quads that were subjected to early depth and stencil testing.

You achieve the best early test rates by ensuring depth testing is enabled, and avoiding fragment shaders that write their depth value.

```
max(min(($MaliCoreQuadsEarlyZSTestedQuads / $MaliCoreQuadsRasterizedFineQuads) *
 100, 100), 0)
```

4.4.2 Early ZS killed quad percentage

This expression defines the percentage of rasterized quads that are killed by early depth and stencil testing.

Quads killed at this stage are killed before shading, so a high percentage here is not generally a performance problem. However, it can indicate an opportunity to use software culling techniques such a portal culling to avoid sending occluded draw calls to the CPU.

```
max(min(($MaliCoreQuadsEarlyZSKilledQuads / $MaliCoreQuadsRasterizedFineQuads) *
 100, 100), 0)
```

4.4.3 Late ZS tested thread percentage

This expression defines the percentage of rasterized threads that are tested by late depth and stencil testing.

A high percentage of fragments performing a late ZS update can cause slow performance, even if fragments are not killed. This occurs because younger fragments cannot complete early ZS until all older fragments at the same coordinate have completed their late ZS operations.

Shaders with mutable coverage, mutable depth, or side-effects on shared resources in memory, use late ZS testing.

The driver also generates late ZS updates to preload a depth or stencil attachment at the start of a render pass. This is needed if the render pass does not start from a cleared depth value.

```
max(min(($MaliCoreThreadsLateZSTestedThreads / ($MaliCoreQuadsRasterizedFineQuads *
 4)) * 100, 100), 0)
```

4.4.4 Late ZS killed thread percentage

This expression defines the percentage of rasterized threads that are killed by late depth and stencil testing. Threads killed by late ZS testing execute at least some of their fragment program before being killed. A significant number of threads being killed at late ZS testing indicates a potential overhead. Aim to minimize the number of threads using and being killed by late ZS testing.

Shaders with mutable coverage, mutable depth, or side-effects on shared resources in memory, use late ZS testing.

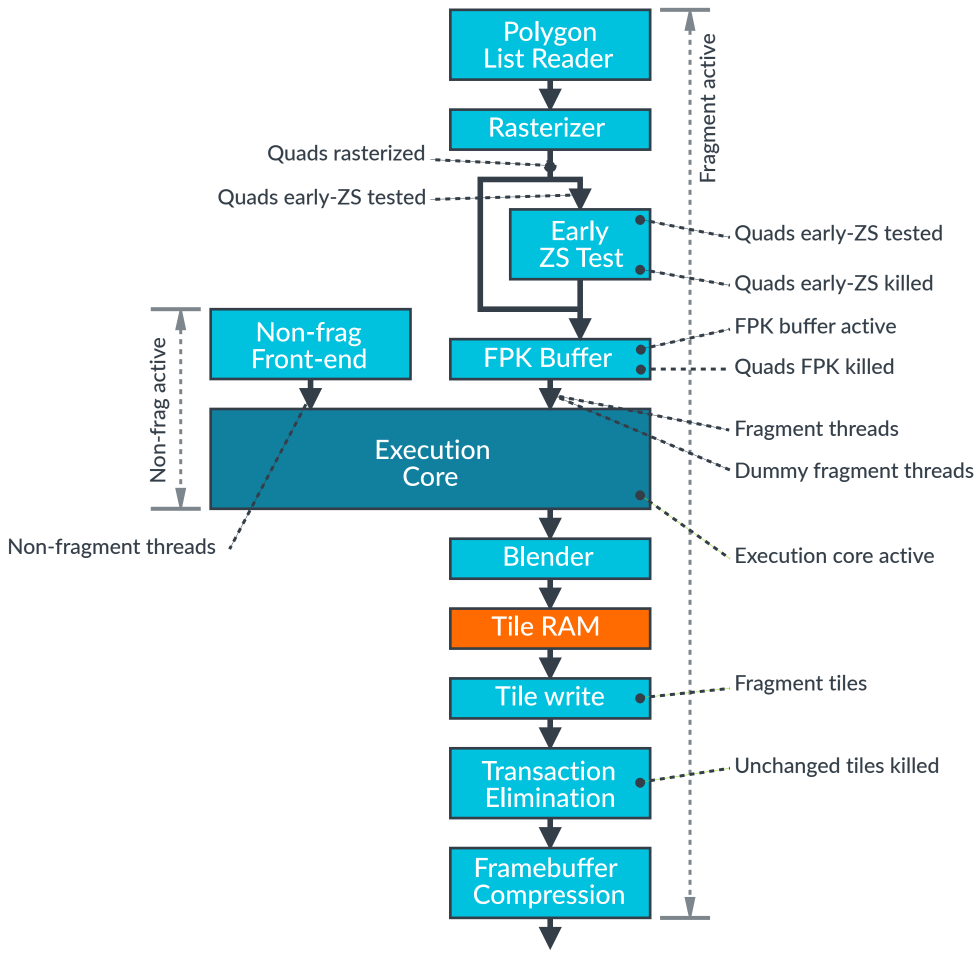
The driver also generates late ZS updates to preload a depth or stencil attachment at the start of a render pass. This is needed if the render pass does not start from a cleared depth value. These fragments show as a late ZS kill, as no shader execution is needed once the depth or stencil value has been set.

```
max(min(($MaliCoreThreadsLateZSKilledThreads / ($MaliCoreQuadsRasterizedFineQuads *
 4)) * 100, 100), 0)
```

5. Shader core data path

Each shader core has two parallel data paths for issuing threads to the core, one for non-fragment workloads and one for fragment workloads. These counters track the thread issue for each path, and their relative scheduling.

Figure 5-1: Midgard GPU shader core



5.1 Shader core workload

The thread counters count the number of shader threads issued for the two workload types.

5.1.1 Non-fragment threads

This expression defines the number of non-fragment threads started.

```
$MaliCoreThreadsNonFragmentThreads
```

5.1.2 Fragment threads

This expression defines the number of fragment threads started.

```
$MaliCoreThreadsFragmentThreads
```

5.2 Shader core throughput

The throughput metrics show the average number of cycles it takes to get a single thread shaded by the shader core. Note that these metrics show average throughput, not average cost, so include the impact of processing latency, memory latency, and any resource sharing inside the shader core.

5.2.1 Non-fragment cycles per thread

This expression defines the average number of shader core cycles per non-fragment thread.

Note that this measurement captures the average throughput, which might not be a direct measure of processing cost for content that is sensitive to memory access latency. In addition, there is some interference caused by non-fragment and fragment workloads running concurrently on the same hardware. This expression is therefore indicative of cost, but does not reflect precise costing.

```
$MaliCoreCyclesNonFragmentActive / $MaliCoreThreadsNonFragmentThreads
```

5.2.2 Fragment cycles per thread

This expression defines the average number of shader core cycles per fragment thread. Note that this measurement captures the average throughput, which might not be a direct measure of processing cost for content which is sensitive to memory access latency. In addition, there is some interference caused by different workload types running concurrently on the same hardware. This expression is therefore indicative of cost, but does not reflect precise costing.

```
$MaliCoreCyclesFragmentActive / $MaliCoreThreadsFragmentThreads
```

5.3 Shader core data path utilization

The data path utilization counters show the total activity level of the major data paths in the shader core. Identifying the dominant workload type helps to target optimizations. Identifying lack of parallelism can confirm that there are scheduling problems.

5.3.1 Non-fragment utilization

This expression defines the percentage utilization of the shader core non-fragment path. This counter measures any cycle where a non-fragment workload is active in either the non-fragment shader core front-end, or in the programmable core itself.

```
max(min(($MaliCoreCyclesNonFragmentActive / $MaliConstantsShaderCoreCount /
 $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

5.3.2 Fragment utilization

This expression defines the percentage utilization of the shader core fragment path. This counter measures any cycle where a fragment workload is active in either the fragment shader core front-end, or in the programmable core itself.

```
max(min(($MaliCoreCyclesFragmentActive / $MaliConstantsShaderCoreCount /
 $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

5.3.3 Fragment FPK buffer utilization

This expression defines the percentage of cycles where the Forward Pixel Kill (FPK) quad buffer contains at least one fragment quad. This buffer is located after early ZS but before the execution core.

During fragment shading this counter must be close to 100%. This indicates that the fragment front-end is able to keep up with the shader core shading rate. This counter commonly drops below 100% for three reasons:

- The running workload has many empty tiles with no geometry to render. Empty tiles are common in shadow maps, for any screen region with no shadow casters.
- The application consists of simple shaders but a high percentage of microtriangles. This combination causes the shader core to complete fragments faster than they are rasterized, so the quad buffer starts to drain.

- The application consists of layers which stall at early ZS because of a dependency on an earlier fragment layer which is still in flight. Stalled layers prevent new fragments entering the quad buffer, so the quad buffer starts to drain.

```
max(min(($MaliCoreCyclesFragmentFPKBActive / $MaliCoreCyclesFragmentActive) * 100,
 100), 0)
```

5.3.4 Execution core utilization

This expression defines the percentage utilization of the programmable execution core, monitoring any cycle where the shader core contains at least one warp. A low utilization here indicates lost performance, because there are spare shader core cycles that are unused.

In some use cases an idle core is unavoidable. For example, a clear color tile that contains no shaded geometry, or a shadow map that is resolved entirely using early ZS depth updates.

Improve execution core utilization by parallel processing of the non-fragment and fragment queues, running overlapping workloads from multiple render passes. Also aim to keep the FPK buffer utilization as high as possible, ensuring constant forward-pressure on fragment shading.

```
max(min(($MaliCoreCyclesExecutionCoreActive / $MaliConstantsShaderCoreCount /
 $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

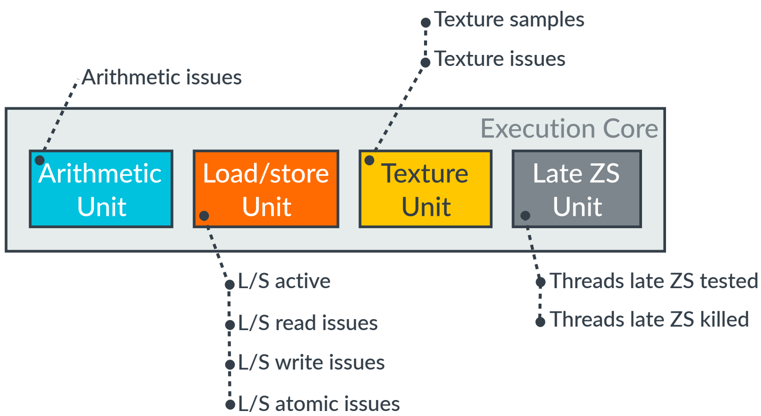
6. Shader core functional units

A shader core consists of multiple parallel processing units that provide both programmable and fixed-function operations. Performance counters can track utilization and workload type for all the major processing units, allowing developers to find both bottlenecks and content inefficiencies to optimize.

For shader-bound content, the functional unit with the highest loading is likely to be the bottleneck. To improve performance, you can reduce the number of operations of that type in the shader. Alternatively, reduce the precision of the operations to use 16-bit types so that multiple operations can be performed in parallel.

For thermally bound content, reducing the critical path load gives the biggest gain as allows use of a lower operating frequency. However, reducing load on any functional unit helps improve energy efficiency.

Figure 6-1: Midgard GPU execution core



6.1 Functional unit utilization

Functional unit utilization counters provide normalized views of the functional unit activity inside the shader core. The functional units run in parallel. To improve performance, target the most heavily utilized functional unit for optimization. Although it might not help performance, reducing the load of any unit improves energy efficiency.

6.1.1 Arithmetic unit utilization

This expression defines the percentage utilization of the arithmetic unit in the execution engine.

The most effective technique for reducing arithmetic load is reducing the complexity of your shader programs. Increasing shader usage of 16-bit (mediump) variables can also help.

```
max(min(($MaliCoreInstructionsExecutedInstructions /
 $MaliCoreCyclesExecutionCoreActive) * 100, 100), 0)
```

6.1.2 Texture unit utilization

This expression defines the percentage utilization of the texturing unit.

The most effective technique for reducing texturing unit load is reducing the number of texture samples read by the fragment shader. Using simpler texture filters can reduce filtering cost. Using 32bpp color formats, and the ASTC decode mode extensions can reduce data access cost.

```
max(min(($MaliCoreTextureCyclesTexturingActive / $MaliCoreCyclesExecutionCoreActive)
 * 100, 100), 0)
```

6.1.3 Load/store unit utilization

This expression defines the percentage utilization of the load/store unit. The load/store unit is used for general-purpose memory accesses, and includes vertex attribute access, buffer access, work group shared memory access, and stack access. This unit also implements imageLoad/Store and atomic access functionality.

For traditional graphics content the most significant contributor to load/store usage is vertex data. Arm recommends simplifying mesh complexity, using fewer triangles, fewer vertices, and fewer bytes per vertex.

Shaders that spill to stack are also expensive, as any spilling is multiplied by the large number of parallel threads that are running. You can use the Mali Offline Compiler to check your shaders for spilling.

```
max(min(($MaliCoreLoadStoreCyclesLoadStoreActive /
 $MaliCoreCyclesExecutionCoreActive) * 100, 100), 0)
```

6.2 Shader workload properties

Shader workload property counters track multiple properties of the running workload that can impact efficiency. These can be used to identify sources of inefficiency that are not related to the shader program code itself.

6.2.1 Helper thread rate

This expression defines the percentage of fragment threads that are helper threads with no coverage. A high percentage can indicate that the content has a high density of small triangles, which are expensive to process. To avoid this, use mesh level-of-detail algorithms to select simpler meshes as objects move further from the camera.

```
max(min(($MaliCoreThreadsFragmentThreads / $MaliCoreThreadsFragmentHelperThreads) *
 100, 100), 0)
```

6.2.2 Unchanged tile kill rate

This expression defines the percentage of tiles that are killed by the transaction elimination CRC check because the content of a tile matches the content already stored in memory.

A high percentage of tile writes being killed indicates that a significant part of the framebuffer is static from frame to frame. Consider using scissor rectangles to reduce the area that is redrawn. To help manage the partial frame updates for window surfaces consider using the EGL extensions such as:

- EGL_KHR_partial_update
- EGL_EXT_swap_buffers_with_damage

```
max(min(($MaliCoreTilesUnchangedTilesKilled / $MaliCoreTilesTiles) * 100, 100), 0)
```

7. Shader core texture unit

The texture unit counters show use of all texture sampling and filtering in shaders. If the shader core utilization counters show that this unit is a bottleneck, these counters can indicate optimization opportunities.

7.1 Texture unit usage

These counters show the usage of the texturing unit, and the average number of cycles per instruction. For Midgard GPUs the maximum performance, using bilinear filtered samples, is 1 cycle per sample.

7.1.1 Texture filtering cycles

This counter increments for every texture filtering issue cycle. This GPU can do 1 2D bilinear texture samples per clock. More complex filtering operations are composed of multiple 2D bilinear samples, and take proportionally more filtering time to complete. The costs per sample are:

- 2D bilinear filtering takes one cycle.
- 2D trilinear filtering takes two cycles.
- 3D bilinear filtering takes two cycles.
- 3D trilinear filtering takes four cycles.
- Sampling from multi-plane YUV takes one cycle per plane.

```
$MaliCoreTextureCyclesTexturingActive
```

7.1.2 Texture filtering cycles per instruction

This expression defines the average number of texture filtering cycles per instruction. For texture-limited content that has a CPI higher than the optimal throughout of this core (1 samples per cycle), consider using simpler texture filters. See *Texture filtering cycles* for details of the expected performance for different types of operation.

```
$MaliCoreTextureCyclesTexturingActive / $MaliCoreTextureRequestsTextureRequests
```

8. Shader core load/store unit

The load/store unit counters show the use of the general-purpose L1 data cache, including any varying interpolation. This unit is used for all shader data accesses except texturing and framebuffer write-back, including: buffer, image, shared storage, and stack access.

8.1 Load/store unit usage

The unit usage counters show the content behavior in the load/store unit. These counters show the number of reads and writes being made.

8.1.1 Load/store total issues

This expression defines the total number of load/store cache access cycles. This counter ignores secondary effects such as cache misses, so provides the minimum possible cycle usage.

```
$MaliCoreLoadStoreCyclesLoadStoreActive
```

8.1.2 Load/store read issues

This expression defines the total number of load/store read cycles.

```
$MaliCoreLoadStoreCyclesReadCycles
```

8.1.3 Load/store write issues

This expression defines the total number of load/store write cycles.

```
$MaliCoreLoadStoreCyclesWriteCycles
```

9. GPU configuration

The GPU configuration counters show the hardware product configuration in the target device. For example, showing the number of shader cores present in the design.

9.1 GPU configuration counters

The configuration counters are virtual counters that you can use to scale performance results and create alternative data visualizations. For example, multiplying the per shader core workload counter series by $MaliConstantsShaderCoreCount would give a GPU-wide total.

9.1.1 Shader core count

This configuration constant defines the number of shader cores in the design.

```
$MaliConstantsShaderCoreCount
```

9.1.2 L2 cache slice count

This configuration constant defines the number of L2 cache slices in the design.

```
$MaliConstantsL2SliceCount
```

9.1.3 External bus beat size

This configuration constant defines the number of bytes transferred per external bus beat.

```
($MaliConstantsBusWidthBits / 8)
```