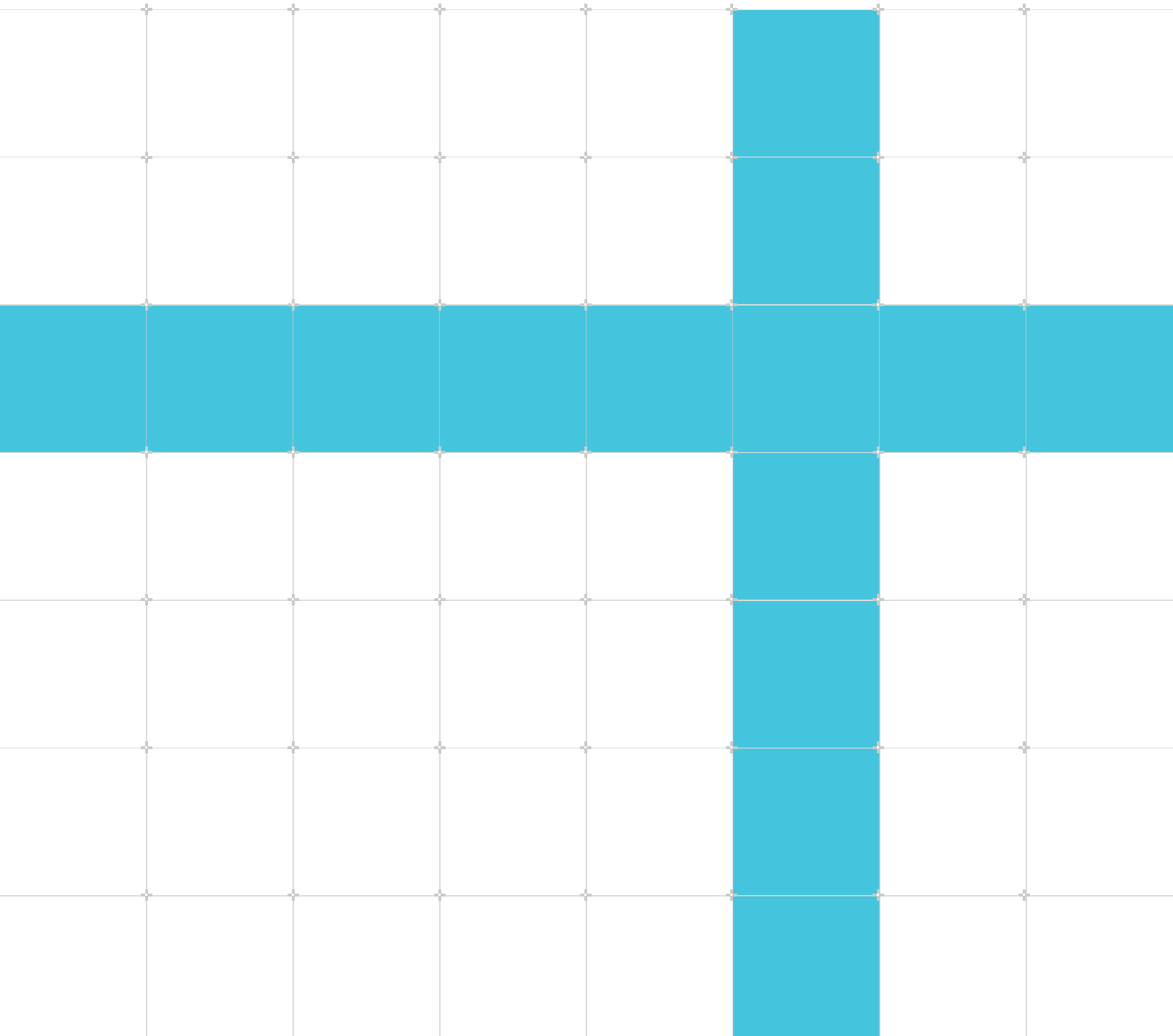
Mali-G610 Performance Counters

1.4

Reference Guide

Issue
102812_0104_en



Mali-G610 Performance Counters

Reference Guide

Release information

Document history

Issue	Date	Confidentiality	Change
1.0	17 February 2022	Non-Confidential	Initial release
1.1	15 July 2022	Non-Confidential	Updated counter guide
1.2	27 October 2022	Non-Confidential	Updated counter guide
1.3	23 February 2023	Non-Confidential	Updated counter guide
1.4	15 June 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. Mali-G610 GPU performance counters

This guide explains the GPU performance counters found in the Arm Streamline profiling template for the Mali-G610 GPU, which is part of the Valhall 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. The Streamline template only shows a subset of the available performance counters. However, it covers the most common types of GPU performance analysis.

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 Mali 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 varying unit: analyze performance of the varying interpolation 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 varying-bound in the shader core functional units section.
- 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.
- Shader core memory traffic: analyze the breakdown of the memory traffic between the shader core and the L2 cache, and the shader core and the external memory system. Use this data to find which type of workload is causing GPU memory accesses, helping you to determine where to apply optimizations.

- 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 Mali 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]
```

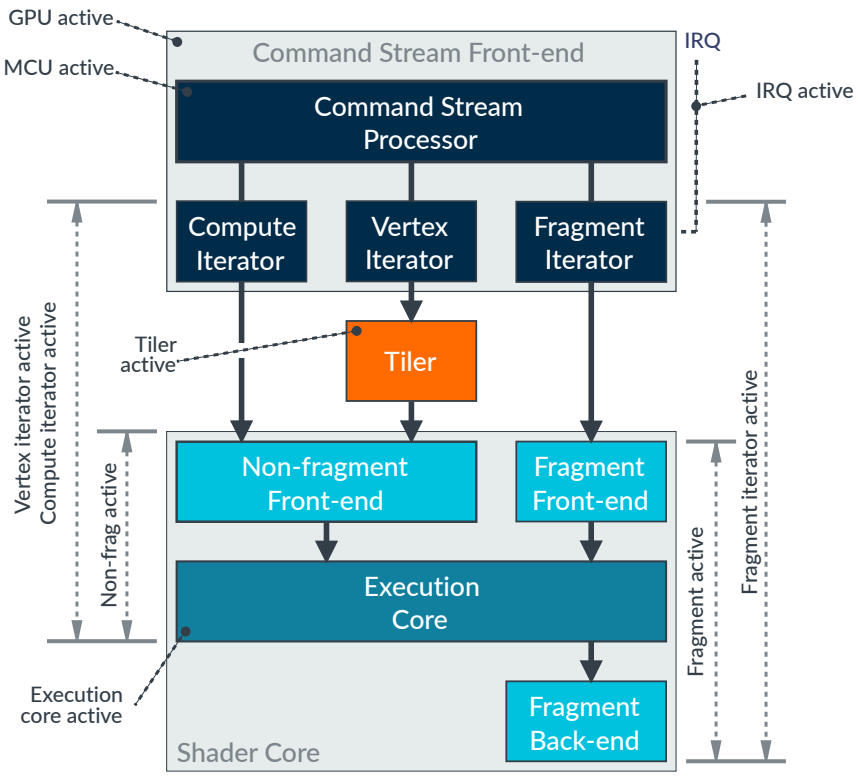
3. GPU activity

The workloads running on this Mali GPU are coordinated by the Command Stream Front-end (CSF). The front-end schedules command streams submitted by the driver on to three hardware work queues, called iterators. Iterators dispatch processing tasks to the shader cores and tiling unit. There are three iterators, one for general-purpose compute shading, one for vertex shading and tiling, and one for fragment shading.

The CSF runs asynchronously to the CPU. If sufficient work is available, the three iterator queues can run in parallel to each other.

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: Valhall CSF GPU top level



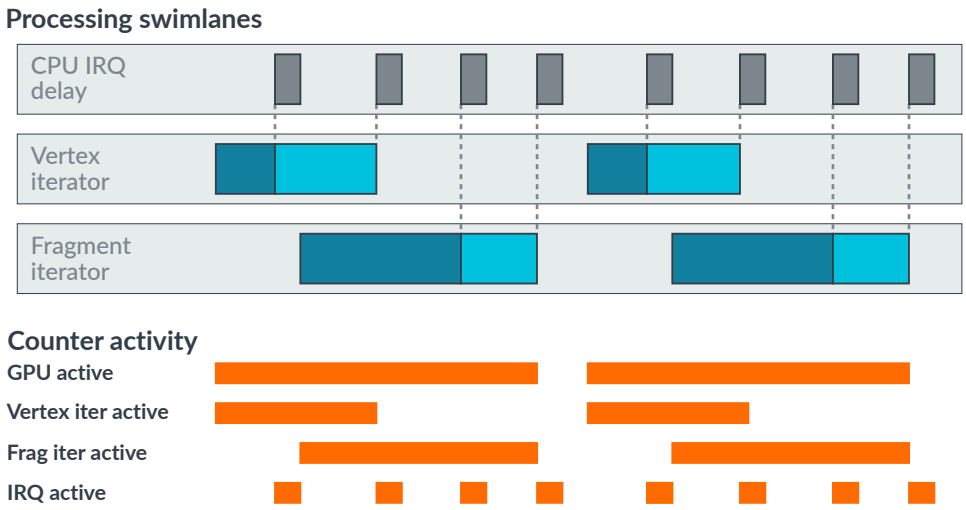
The “active” counters show that a data path or hardware unit processed some workload, but do not necessarily indicate that it was fully utilized. For example, the *Fragment iterator queued* counter increments every cycle where there is any fragment workload queued to run anywhere in the GPU.

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: Valhall CSF 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 front-end 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 MCU active cycles

This counter increments every clock cycle where the GPU command stream microcontroller is executing. Cycles waiting for interrupts or events are not counted.

```
$MaliGPUCyclesMCUActive
```

3.1.3 Vertex iterator issue cycles

This expression increments every clock cycle that the command stream vertex iterator has at least one task issued for processing.

```
$MaliGPUCyclesVertexQueued - $MaliGPUCyclesVertexEndpointStall
```

3.1.4 Fragment iterator issue cycles

This expression increments every clock cycle that the command stream fragment iterator has at least one task issued for processing.

```
$MaliGPUCyclesFragmentQueued - $MaliGPUCyclesFragmentEndpointStall
```

3.1.5 Compute iterator issue cycles

This expression increments every clock cycle that the command stream compute iterator has at least one task issued for processing.

```
$MaliGPUCyclesComputeQueued - $MaliGPUCyclesComputeEndpointStall
```

3.1.6 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

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 Microcontroller utilization

This expression defines the microcontroller utilization compared against the GPU active cycles. High microcontroller load can be indicative of content using many emulated commands, such as API command stream synchronization primitives.

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

3.2.2 Vertex iterator utilization

This expression defines the vertex iterator utilization compared against the GPU active cycles. For GPU bound content, it is expected that the GPU iterators process work in parallel. The dominant iterator must be close to 100% utilized. If no iterator is dominant, but the GPU is fully utilized, then a serialization or dependency problem might be preventing iterator overlap.

```
max(min((($MaliGPUCyclesVertexQueued - $MaliGPUCyclesVertexEndpointStall) /
 $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

3.2.3 Fragment iterator utilization

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

```
max(min((($MaliGPUCyclesFragmentQueued - $MaliGPUCyclesFragmentEndpointStall) /
 $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

3.2.4 Compute iterator utilization

This expression defines the compute iterator utilization compared against the GPU active cycles. For GPU bound content, it is expected that the GPU iterators process work in parallel. The dominant iterator must be close to 100% utilized. If no iterator is dominant, but the GPU is fully utilized, then a serialization or dependency problem might be preventing iterator overlap.

```
max(min((($MaliGPUCyclesComputeQueued - $MaliGPUCyclesComputeEndpointStall) /
 $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

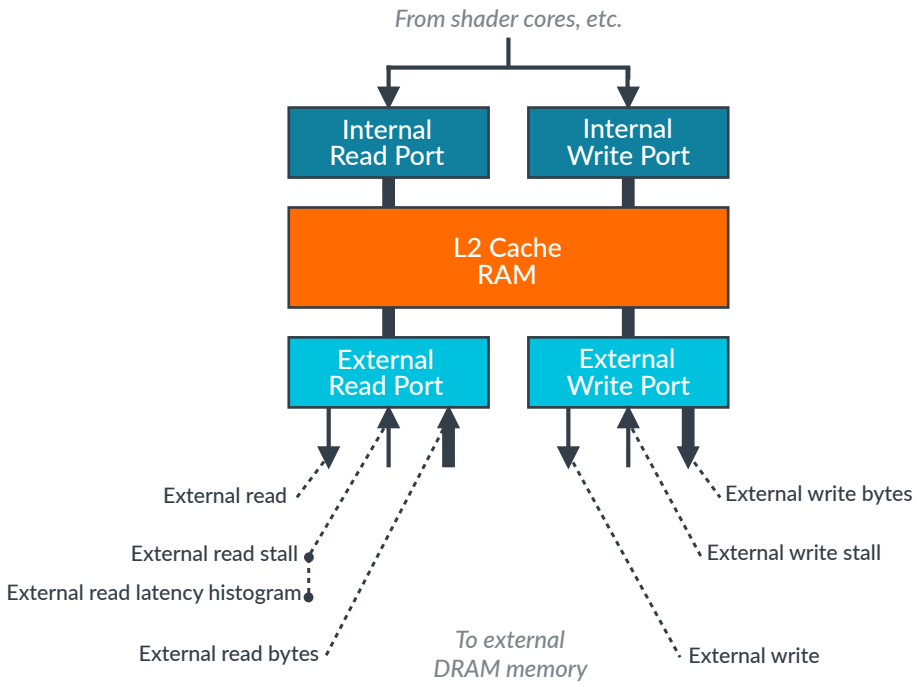
3.3 External memory bandwidth

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: Valhall 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)
```

3.5 External memory read latency

The external memory read latency counters present a histogram of access latencies. This metric shows how many GPU cycles it takes to fetch data from the downstream memory system, which is either system cache or external DRAM.

High latency accesses can reduce performance, and are normally an indication that the application is requesting more data than the memory system can provide. To optimize the workload, try to reduce memory bandwidth.

3.5.1 Output external read latency 0-127 cycles

This counter increments for every data beat that is returned between 0 and 127 cycles after the read transaction started. This latency is considered a fast access response speed.

```
$MaliExternalBusReadLatency0127Cycles
```

3.5.2 Output external read latency 128-191 cycles

This counter increments for every data beat that is returned between 128 and 191 cycles after the read transaction started. This latency is considered a normal access response speed.

```
$MaliExternalBusReadLatency128191Cycles
```

3.5.3 Output external read latency 192-255 cycles

This counter increments for every data beat that is returned between 192 and 255 cycles after the read transaction started. This latency is considered a normal access response speed.

```
$MaliExternalBusReadLatency192255Cycles
```

3.5.4 Output external read latency 256-319 cycles

This counter increments for every data beat that is returned between 256 and 319 cycles after the read transaction started. This latency is considered a slow access response speed.

```
$MaliExternalBusReadLatency256319Cycles
```

3.5.5 Output external read latency 320-383 cycles

This counter increments for every data beat that is returned between 320 and 383 cycles after the read transaction started. This latency is considered a slow access response speed.

```
$MaliExternalBusReadLatency320383Cycles
```

3.5.6 Output external read latency 384+ cycles

This expression increments for every read beat that is returned at least 384 cycles after the transaction started. This latency is considered a very slow access response speed.

```
$MaliExternalBusBeatsReadBeats - $MaliExternalBusReadLatency0127Cycles -
 $MaliExternalBusReadLatency128191Cycles - $MaliExternalBusReadLatency192255Cycles -
 $MaliExternalBusReadLatency256319Cycles - $MaliExternalBusReadLatency320383Cycles
```

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
 + $MaliPrimitiveCullingSampleTestCulledPrimitives +
 $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 because of 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 +
 $MaliPrimitiveCullingSampleTestCulledPrimitives
```

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.

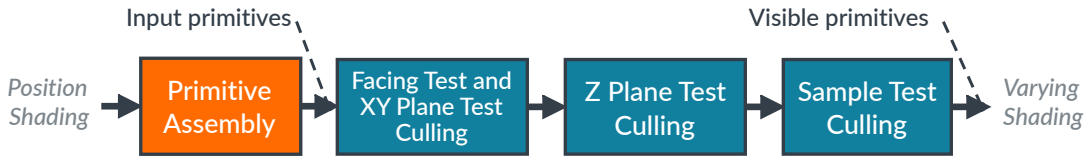
```
$MaliPrimitiveCullingVisiblePrimitives
```

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 runs 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: Valhall 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
 + $MaliPrimitiveCullingSampleTestCulledPrimitives +
 $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
 + $MaliPrimitiveCullingSampleTestCulledPrimitives +
 $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
 + $MaliPrimitiveCullingSampleTestCulledPrimitives +
 $MaliPrimitiveCullingVisiblePrimitives) -
 $MaliPrimitiveCullingFacingAndXYPlaneTestCulledPrimitives)) * 100, 100), 0)
```

4.2.4 Sample test cull rate

This expression defines the percentage of primitives entering the sample coverage test that are culled by it. This stage culls primitives that are so small that they hit no rasterizer sample points.

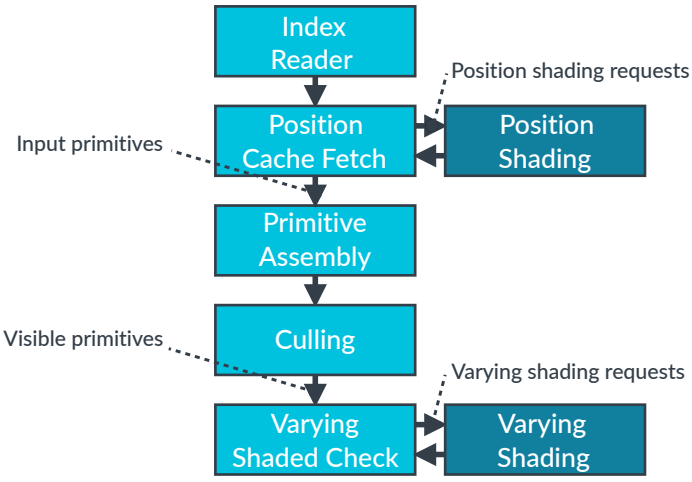
A significant proportion of primitives being culled at this stage indicates that the application is using geometry meshes that are too complex for their screen coverage. Aim to keep triangle screen area above 10 pixels. Use schemes such as dynamic mesh level-of-detail to select simpler meshes as objects move further away from the camera.

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

4.3 Vertex shading

This GPU uses an optimized vertex processing pipeline. In this pipeline, the vertex position is computed before culling. The remaining varyings for any visible vertices are computed after culling. To determine mesh encoding efficiency, use the performance counters to measure the average number of position threads and varying threads per primitive.

Figure 4-2: Valhall GPU tiling pipeline



This pipeline uses a post-transform vertex cache, which contains the positions of recently shaded vertices, to avoid reshading vertices that are common to multiple primitives. Poor temporal locality of index reuse can result in a vertex being shaded multiple times, because it is evicted from the cache before it is reused.

This pipeline submits shading requests in groups of 4 contiguous index values. Unused index locations might be shaded if they are near used index locations. Reduce redundant shading by ensuring that every index between the minimum and maximum index is used.

4.3.1 Position shader thread invocations

This expression defines the number of position shader thread invocations.

```
$MaliTilerShadingRequestsPositionShadingRequests * 4
```

4.3.2 Varying shader thread invocations

This expression defines the number of varying shader thread invocations.

```
$MaliTilerShadingRequestsVaryingShadingRequests * 4
```

Normalized versions of these counters show the amount of shading per primitive, which gives a direct measure of mesh encoding efficiency.

4.3.3 Position threads per input primitive

This expression defines the number of position shader threads per input primitive. Minimize this number by reusing vertices for nearby primitives, improving temporal locality of index reuse, and avoiding unused values in the active index range.

Efficient meshes with a good vertex reuse have average less than 1.5 vertices shaded per triangle, as vertex computation is shared by multiple primitives.

Inefficient meshes with no vertex reuse shade at least 3 vertices per triangle. They can require more if indices are reshaded or if redundant indices are shaded.

```
($MaliTilerShadingRequestsPositionShadingRequests * 4) /
 ($MaliPrimitiveCullingFacingAndXYPlaneTestCulledPrimitives
 + $MaliPrimitiveCullingZPlaneTestCulledPrimitives
 + $MaliPrimitiveCullingSampleTestCulledPrimitives +
 $MaliPrimitiveCullingVisiblePrimitives)
```

4.3.4 Varying threads per input primitive

This expression defines the number of varying shader invocations per visible primitive. Minimize this number by reusing vertices for nearby primitives, improving temporal locality of index reuse, and avoiding unused values in the active index range.

Efficient meshes with a good vertex reuse have average less than 1.5 vertices shaded per triangle, as vertex computation is shared by multiple primitives.

Inefficient meshes with no vertex reuse shade at least 3 vertices per visible triangle. They can require more if indices are reshaded or if redundant indices are shaded.

```
($MaliTilerShadingRequestsVaryingShadingRequests * 4) /
 $MaliPrimitiveCullingVisiblePrimitives
```

4.4 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 Mali 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.

4.4.1 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 32x32 pixel region are shaded. If the rendered region does not align with 32 pixel aligned boundaries, then this metric includes pixels that are not actually shaded.

```
$MaliGPUTasksFragmentTasks * 1024
```

4.4.2 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 * 1024)
```

4.4.3 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.

```
($MaliCoreWarpsFragmentWarps * 16 * $MaliConstantsShaderCoreCount) /
 ($MaliGPUTasksFragmentTasks * 1024)
```

4.5 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.

4.5.1 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)
```

4.5.2 Early ZS updated quad percentage

This expression defines the percentage of rasterized quads that update the framebuffer during early depth and stencil testing.

You achieve the best early update rates by enabling depth testing, and avoiding draw calls with modifiable coverage or shader depth writes.

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

4.5.3 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)
```

4.5.4 FPK killed quad percentage

This expression defines the percentage of rasterized quads that are killed by the Forward Pixel Kill (FPK) hidden surface removal scheme.

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((($MaliCoreQuadsRasterizedFineQuads - $MaliCoreQuadsEarlyZSKilledQuads -
 (($MaliCoreWarpsFragmentWarps * 16) / 4)) / $MaliCoreQuadsRasterizedFineQuads) *
 100, 100), 0)
```

4.5.5 Late ZS tested quad percentage

This expression defines the percentage of rasterized quads 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. Younger fragments cannot complete early ZS until all older fragments at the same coordinate have completed their late ZS operations, which can cause this expression to be high.

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, which is needed if the render pass does not start from a cleared depth value.

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

4.5.6 Late ZS killed quad percentage

This expression defines the percentage of rasterized quads that are killed by late depth and stencil testing. Quads killed by late ZS testing execute at least some of their fragment program before being killed. A significant number of quads being killed at late ZS testing indicates a potential overhead. Aim to minimize the number of quads 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, which is needed if the render pass does not start from a cleared depth value. These

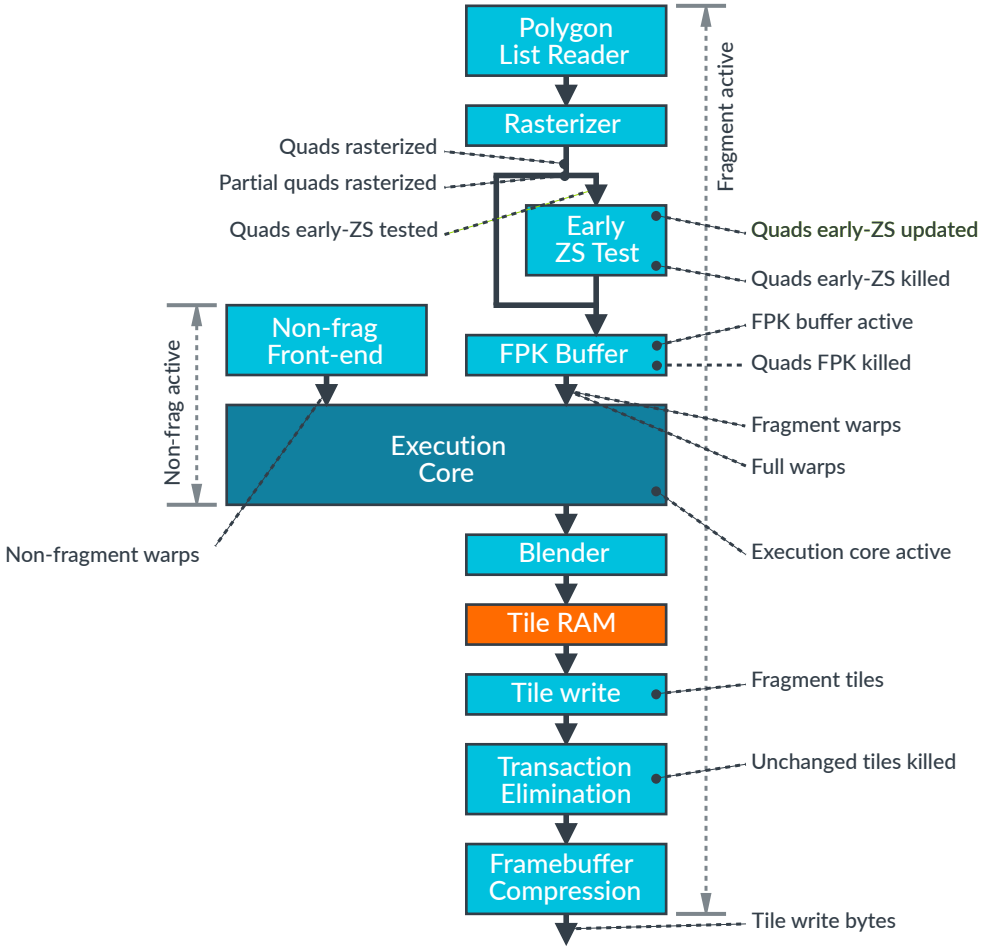
fragments show as a late ZS kill, as no shader execution is needed once the depth or stencil value has been set.

```
max(min(($MaliCoreQuadsLateZSKilledQuads / $MaliCoreQuadsRasterizedFineQuads) * 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: Valhall GPU shader core



5.1 Shader core workload

The warp counters count the number of shader warps issued for the two workload types. Each warp contains 16 threads.

5.1.1 Non-fragment warps

This counter increments for every created non-fragment warp. For this GPU, a warp contains 16 threads.

For compute shaders, to ensure full utilization of the warp capacity any compute work groups must be a multiple of warp size.

```
$MaliCoreWarpsNonFragmentWarps
```

5.1.2 Fragment warps

This counter increments for every created fragment warp. For this GPU, a warp contains 16 threads.

Fragment warps are populated with fragment quads, where each quad corresponds to a 2x2 fragment region from a single triangle. Threads in a quad which correspond to a sample point outside of the triangle still consumes shader resource, which makes small triangles disproportionately expensive.

```
$MaliCoreWarpsFragmentWarps
```

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.

```
$MaliCoreCyclesNonFragmentActive / ($MaliCoreWarpsNonFragmentWarps * 16)
```

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.

```
$MaliCoreCyclesFragmentActive / ($MaliCoreWarpsFragmentWarps * 16)
```

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 Shader core usage

This expression defines the percentage usage of the shader core, relative to the top-level GPU clock. This counter increments every shader core clock cycle when any of the shader core queues contain work.

To improve energy efficiency, some systems clock the shader cores at a lower frequency than the GPU top-level components. In these systems, the maximum achievable usage value is the clock ratio between the GPU top-level clock and the shader clock. For example, a GPU with an 800MHz top-level clock and a 400MHz shader clock can achieve a maximum usage of 50%.

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

5.3.2 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 / $MaliCoreCyclesAnyActive) * 100, 100),
 0)
```

5.3.3 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 / $MaliCoreCyclesAnyActive) * 100, 100), 0)
```

5.3.4 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)
```

5.3.5 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 / $MaliCoreCyclesAnyActive) * 100, 100),
 0)
```

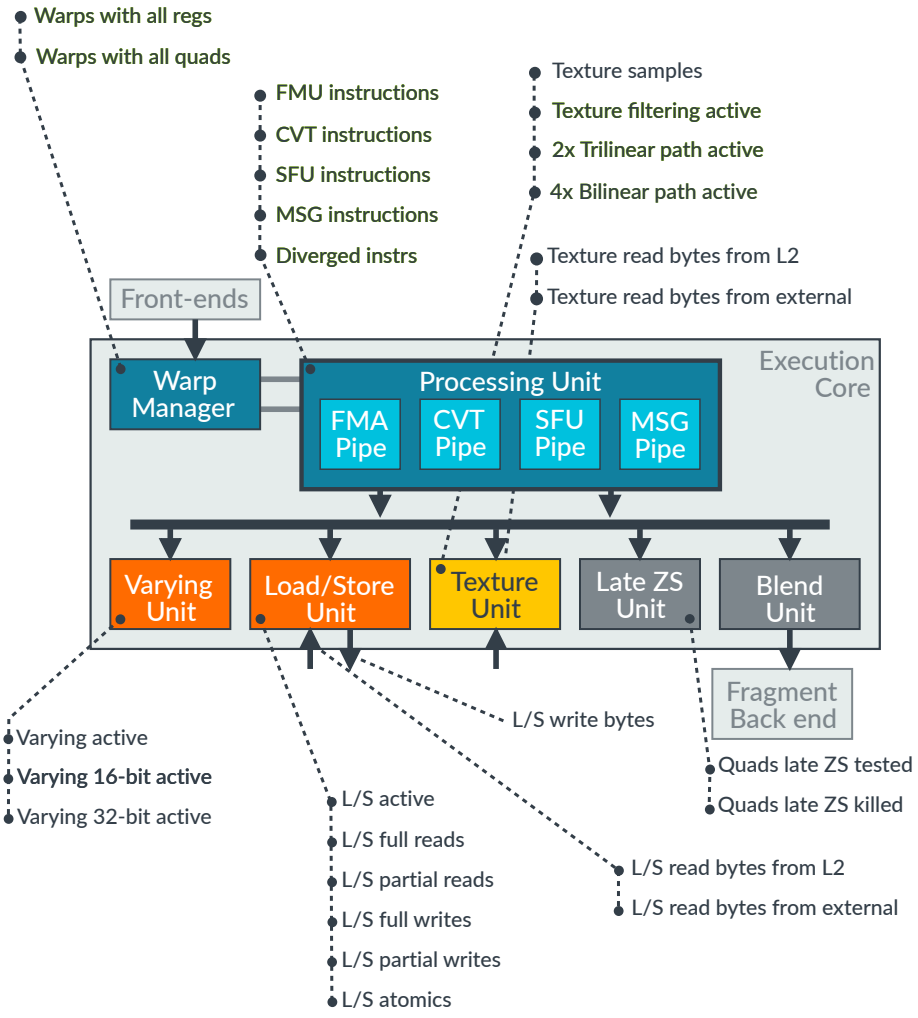
6. Shader core functional units

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 it allows use of a lower operating frequency. However, reducing load on any functional unit helps improve energy efficiency.

Figure 6-1: Valhall 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 estimates 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((max($MaliCoreInstructionsFMAInstructions +
 $MaliCoreInstructionsCVTInstructions + $MaliCoreInstructionsSFUInstructions,
 $MaliCoreInstructionsSFUInstructions * 4) / $MaliCoreCyclesExecutionCoreActive) *
 100, 100), 0)
```

6.1.2 Varying unit utilization

This expression defines the percentage utilization of the varying unit.

The most effective technique for reducing varying load is reducing the number of interpolated values read by the fragment shading. Increasing shader usage of 16-bit (mediump) input variables also helps, as they can be interpolated as twice the speed of 32-bit variables.

```
max(min((($MaliCoreVaryingIssues32BitInterpolationSlots /
 2) + ($MaliCoreVaryingIssues16BitInterpolationSlots / 2)) /
 $MaliCoreCyclesExecutionCoreActive) * 100, 100), 0)
```

6.1.3 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)
```

6.1.4 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((($MaliCoreLoadStoreCyclesFullReadCycles +
 $MaliCoreLoadStoreCyclesPartialReadCycles + $MaliCoreLoadStoreCyclesFullWriteCycles
 + $MaliCoreLoadStoreCyclesPartialWriteCycles +
 $MaliCoreLoadStoreCyclesAtomicAccessCycles) / $MaliCoreCyclesExecutionCoreActive) *
 100, 100), 0)
```

6.2 Shader program properties

Shader program property counters track multiple properties related to the running shader program instruction execution. These can be used to identify sources of program inefficiency.

6.2.1 Warp divergence percentage

This expression defines the percentage of instructions that have control flow divergence across the warp.

```
max(min(($MaliCoreInstructionsDivergedInstructions /
 ($MaliCoreInstructionsFMAInstructions + $MaliCoreInstructionsCVTInstructions +
 $MaliCoreInstructionsSFUInstructions)) * 100, 100), 0)
```

6.2.2 All registers warp rate

This expression defines the percentage of warps that use more than 32 registers, requiring the full register allocation of 64 registers. Warps that require more than 32 registers halve the peak thread occupancy of the shader core, and can make shader performance more sensitive to cache misses and memory stalls.

```
max(min(($MaliCoreWarpsAllRegisterWarps / ($MaliCoreWarpsNonFragmentWarps +
 $MaliCoreWarpsFragmentWarps)) * 100, 100), 0)
```

6.2.3 Shader blend path percentage

This expression defines the percentage of fragments that use shader-based blending, rather than the fixed-function blend path. These fragments are caused by the application using color formats, or advanced blend equations, which the fixed-function blend path does not support.

Vulkan shaders that use software blending do not show up in this data, because the blend is inlined in to the main body of the shader program.

```
max(min((($MaliCoreInstructionsBlendShaderCalls * 4) / $MaliCoreWarpsFragmentWarps)
 * 100, 100), 0)
```

6.3 Shader workload properties

Shader program property counters track multiple properties related to the running shader program instruction execution. These counters are used to identify shader execution behaviors that are sources of program inefficiency.

6.3.1 Partial coverage rate

This expression defines the percentage of fragment quads that contain samples 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(($MaliCoreQuadsPartialRasterizedFineQuads /
 $MaliCoreQuadsRasterizedFineQuads) * 100, 100), 0)
```

6.3.2 Full quad warp rate

This expression defines the percentage of warps that are fully populated with quads. If many warps are not full then performance is reduced. Full warps are more likely if:

- Compute shaders have work groups that are a multiple of warp size.
- Draw calls avoid high numbers of small primitives.

```
max(min(($MaliCoreWarpsFullQuadWarps / ($MaliCoreWarpsNonFragmentWarps +
 $MaliCoreWarpsFragmentWarps)) * 100, 100), 0)
```

6.3.3 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 / (4 * $MaliCoreTilesTiles)) * 100,
 100), 0)
```

7. Shader core varying unit

The varying unit counters monitor the varying interpolation in fragment shaders. If the shader core utilization counters show that this unit is a bottleneck, these counters can indicate optimization opportunities.

The interpolator has one or more 32-bit data paths per thread. Each data path can interpolate a scalar 32-bit value or a vec2 16-bit value in a single cycle. Arm recommends using 16-bit (`mediump`) varying inputs to fragment shaders whenever possible. We also recommend packing 16-bit values into vec2 or vec4 values. For example, a single vec4 interpolates faster than a separate vec3 and scalar float pair.

7.1 Varying unit usage

These counters show the usage of the varying interpolation unit, and the breakdown by data type size.

7.1.1 Varying cycles

This expression defines the total number of cycles where the varying interpolator is active.

```
($MaliCoreVaryingIssues32BitInterpolationSlots / 2) +
 ($MaliCoreVaryingIssues16BitInterpolationSlots / 2)
```

7.1.2 16-bit interpolation cycles

This counter increments for every 16-bit interpolation cycle processed by the varying unit.

```
$MaliCoreVaryingIssues16BitInterpolationSlots / 2
```

7.1.3 32-bit interpolation cycles

This counter increments for every 32-bit interpolation cycle processed by the varying unit. 32-bit interpolation is half the performance of 16-bit interpolation, so if content is varying bound consider reducing precision of varying inputs to fragment shaders.

```
$MaliCoreVaryingIssues32BitInterpolationSlots / 2
```

8. 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.

8.1 Texture unit usage

These counters show the usage of the texturing unit, and the average number of cycles per instruction. The performance of the texture unit in the shader core varies for different GPUs in the Valhall family. For Mali-G610 GPU the maximum performance (bilinear filtered samples) is 0.125 cycles per sample.

8.1.1 Texture filtering cycles

This counter increments for every texture filtering issue cycle. This GPU can do 8 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 sampled quad are:

- 2D bilinear filtering takes half a cycle.
- 2D trilinear filtering takes one cycle.
- 3D bilinear filtering takes one cycle.
- 3D trilinear filtering takes two cycles.

Anisotropic filtering makes multiple filtered subsamples which are combined to make the final output sample color. For a filter with MAX_ANISOTROPY of N, up to N times the cycles of the base filter are required.

```
$MaliCoreTextureCyclesTexturingActive
```

8.1.2 Texture filtering cycles using 8x bilinear

This counter increments for every cycle where the filtering unit uses the 8x path to implement nearest or bilinear filtering. This provides eight filtered samples per clock.

```
$MaliCoreTextureCycles8xBilinearFilteringActive
```

8.1.3 Texture filtering cycles using 4x trilinear

This counter increments for every cycle where the filtering unit uses the 8x path to implement 4x trilinear filtering. This provides four filtered samples per clock.

```
$MaliCoreTextureCycles4xTrilinearFilteringActive
```

8.1.4 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 (8 samples per cycle), consider using simpler texture filters. See *Texture filtering cycles* for details of the expected performance for different types of operation.

```
$MaliCoreTextureCyclesTexturingActive / ($MaliCoreTextureQuadsTextureMessages * 8)
```

8.2 Texture unit memory usage

These counters show the average number of bytes read from the L2 cache or external memory per texture sample. If you are fetching more bytes per access than the size of the texture format, you might be thrashing the cache. Consider using a narrower texture format data type, compression, and mipmaps.

8.2.1 Texture bytes read from L2 per texture cycle

This expression defines the average number of bytes read from the L2 memory system by the texture unit per filtering cycle. This metric indicates how effectively textures are being cached in the L1 texture cache.

If more bytes are being requested per access than you would expect for the format you are using, review your texture settings. Arm recommends:

- Enabling mipmaps for offline generated textures.
- Using ASTC or ETC compression for offline generated textures.
- Replacing runtime generated framebuffer and texture formats with a narrower format.
- Reducing use of imageLoad/Store in OpenGL ES and Vulkan, as they prevent use of framebuffer compression.
- Reducing any use of negative LOD bias used for texture sharpening.
- Reducing the MAX_ANISOTROPY level for anisotropic filtering.

```
($MaliCoreL2ReadsTextureL2ReadBeats * 16) / $MaliCoreTextureCyclesTexturingActive
```

8.2.2 Texture bytes read from external memory per texture cycle

This expression defines the average number of bytes read from the external memory system by the texture unit per filtering cycle. This metric indicates how effectively textures are being cached in the L2 cache.

If more bytes are being requested per access than you would expect for the format you are using, review your texture settings. Arm recommends:

- Enabling mipmaps for offline generated textures.
- Using ASTC or ETC compression for offline generated textures.
- Replacing runtime generated framebuffer and texture formats with a narrower format.
- Reducing use of imageLoad/Store in OpenGL ES and Vulkan, as they prevent use of framebuffer compression.
- Reducing any use of negative LOD bias used for texture sharpening.
- Reducing the MAX_ANISOTROPY level for anisotropic filtering.

```
($MaliCoreExternalReadsTextureExternalReadBeats * 16) /
 $MaliCoreTextureCyclesTexturingActive
```

9. Shader core load/store unit

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

9.1 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, and whether the loads use the full width of the available data path.

Make effective use of the load/store unit by making full-width accesses. We recommend shaders use vector memory accesses, and ensure threads in the same warp access overlapping or sequential addresses inside a single 64-byte range.

9.1.1 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.

```
$MaliCoreLoadStoreCyclesFullReadCycles + $MaliCoreLoadStoreCyclesPartialReadCycles +
 $MaliCoreLoadStoreCyclesFullWriteCycles +
 $MaliCoreLoadStoreCyclesPartialWriteCycles +
 $MaliCoreLoadStoreCyclesAtomicAccessCycles
```

9.1.2 Load/store full read issues

This counter increments for every full-width load/store cache read.

```
$MaliCoreLoadStoreCyclesFullReadCycles
```

9.1.3 Load/store partial read issues

This counter increments for every partial-width load/store cache read. Partial data accesses do not make full use of the load/store cache capability. Merging short accesses together to make fewer larger requests improves efficiency. To do this in shader code:

- Use vector data loads.
- Avoid padding in strided data accesses.

- Write compute shaders so that adjacent threads in a warp access adjacent addresses in memory.

```
$MaliCoreLoadStoreCyclesPartialReadCycles
```

9.1.4 Load/store full write issues

This counter increments for every full-width load/store cache write.

```
$MaliCoreLoadStoreCyclesFullWriteCycles
```

9.1.5 Load/store partial write issues

This counter increments for every partial-width load/store cache write. Partial data accesses do not make full use of the load/store cache capability. Merging short accesses together to make fewer larger requests improves efficiency. To do this in shader code:

- Use vector data loads.
- Avoid padding in strided data accesses.
- Write compute shaders so that adjacent threads in a warp access adjacent addresses in memory.

```
$MaliCoreLoadStoreCyclesPartialWriteCycles
```

9.1.6 Load/store atomic issues

This counter increments for every load/store atomic access. Atomic memory accesses are typically multicycle operations per thread in the warp, so they are exceptionally expensive. Minimize the use of atomics in performance critical code.

```
$MaliCoreLoadStoreCyclesAtomicAccessCycles
```

9.2 Load/store unit memory usage

The memory usage counters show the average number of bytes read or written to the L2 cache per load/store read or write. Use these metrics to see how effectively your workloads are using the L1 and L2 data caches.

9.2.1 Load/store bytes read from L2 per access cycle

This expression defines the average number of bytes read from the L2 memory system by the load/store unit per read cycle. This metric gives some idea how effectively data is being cached in the L1 load/store cache.

Review your access patterns if more bytes are being requested per access than you would expect for the data format you are using.

```
($MaliCoreL2ReadsLoadStoreL2ReadBeats * 16) /
 ($MaliCoreLoadStoreCyclesFullReadCycles +
 $MaliCoreLoadStoreCyclesPartialReadCycles)
```

9.2.2 Load/store bytes read from external memory per access cycle

This expression defines the average number of bytes read from the external memory system by the load/store unit per read cycle. This metric indicates how effectively data is being cached in the L2 cache. If a high number of bytes are being requested per access for the buffer formats you are using, review your data types and access patterns.

```
($MaliCoreExternalReadsLoadStoreExternalReadBeats
 * 16) / ($MaliCoreLoadStoreCyclesFullReadCycles +
 $MaliCoreLoadStoreCyclesPartialReadCycles)
```

9.2.3 Load/store bytes written to L2 per access cycle

This expression defines the average number of bytes written to the L2 memory system by the load/store unit per write cycle.

```
(($MaliCoreWritesLoadStoreWriteBackWriteBeats +
 $MaliCoreWritesLoadStoreOtherWriteBeats) * 16) /
 ($MaliCoreLoadStoreCyclesFullWriteCycles +
 $MaliCoreLoadStoreCyclesPartialWriteCycles)
```

10. Shader core memory traffic

The shader core memory traffic counters show the total amount of memory access a shader core makes to the L2 cache and external memory system. Use the data breakdown to identify the unit making the accesses, and target that unit for optimization.

10.1 Read access from L2 cache

The L2 memory read counters show the shader core memory read traffic that is fetched from the GPU L2 cache.

10.1.1 Front-end read bytes from L2 cache

This expression defines the total number of bytes read from the L2 memory system by the fragment front-end unit.

```
$MaliCoreL2ReadsFragmentL2ReadBeats * 16
```

10.1.2 Load/store read bytes from L2 cache

This expression defines the total number of bytes read from the L2 memory system by the load/store unit.

```
$MaliCoreL2ReadsLoadStoreL2ReadBeats * 16
```

10.1.3 Texture read bytes from L2 cache

This expression defines the total number of bytes read from the L2 memory system by the texture unit.

```
$MaliCoreL2ReadsTextureL2ReadBeats * 16
```

10.2 Read access from external memory

The external memory read counters show the shader core memory read traffic that misses in the GPU cache and that is fetched from the external memory system. This data is either fetched from a layer of system cache external to the GPU, or from the main system DRAM.

10.2.1 Front-end read bytes from external memory

This expression defines the total number of bytes read from the external memory system by the fragment front-end unit.

```
$MaliCoreExternalReadsFragmentExternalReadBeats * 16
```

10.2.2 Load/store read bytes from external memory

This expression defines the total number of bytes read from the external memory system by the load/store unit.

```
$MaliCoreExternalReadsLoadStoreExternalReadBeats * 16
```

10.2.3 Texture read bytes from external memory

This expression defines the total number of bytes read from the external memory system by the texture unit.

```
$MaliCoreExternalReadsTextureExternalReadBeats * 16
```

10.3 Write access

The memory write counters show the shader core memory traffic that is written into the memory system. These writes can be buffered by the GPU L2, or sent to external memory.

10.3.1 Load/store write bytes

This expression defines the total number of bytes written to the L2 memory system by the load/store unit.

```
($MaliCoreWritesLoadStoreWriteBackWriteBeats +
 $MaliCoreWritesLoadStoreOtherWriteBeats) * 16
```

10.3.2 Tile buffer write bytes

This expression defines the total number of bytes written to the L2 memory system by the tile buffer write-back unit.

```
$MaliCoreWritesTileBufferWriteBeats * 16
```

11. 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.

11.1 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.

11.1.1 Shader core count

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

```
$MaliConstantsShaderCoreCount
```

11.1.2 L2 cache slice count

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

```
$MaliConstantsL2SliceCount
```

11.1.3 External bus beat size

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

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