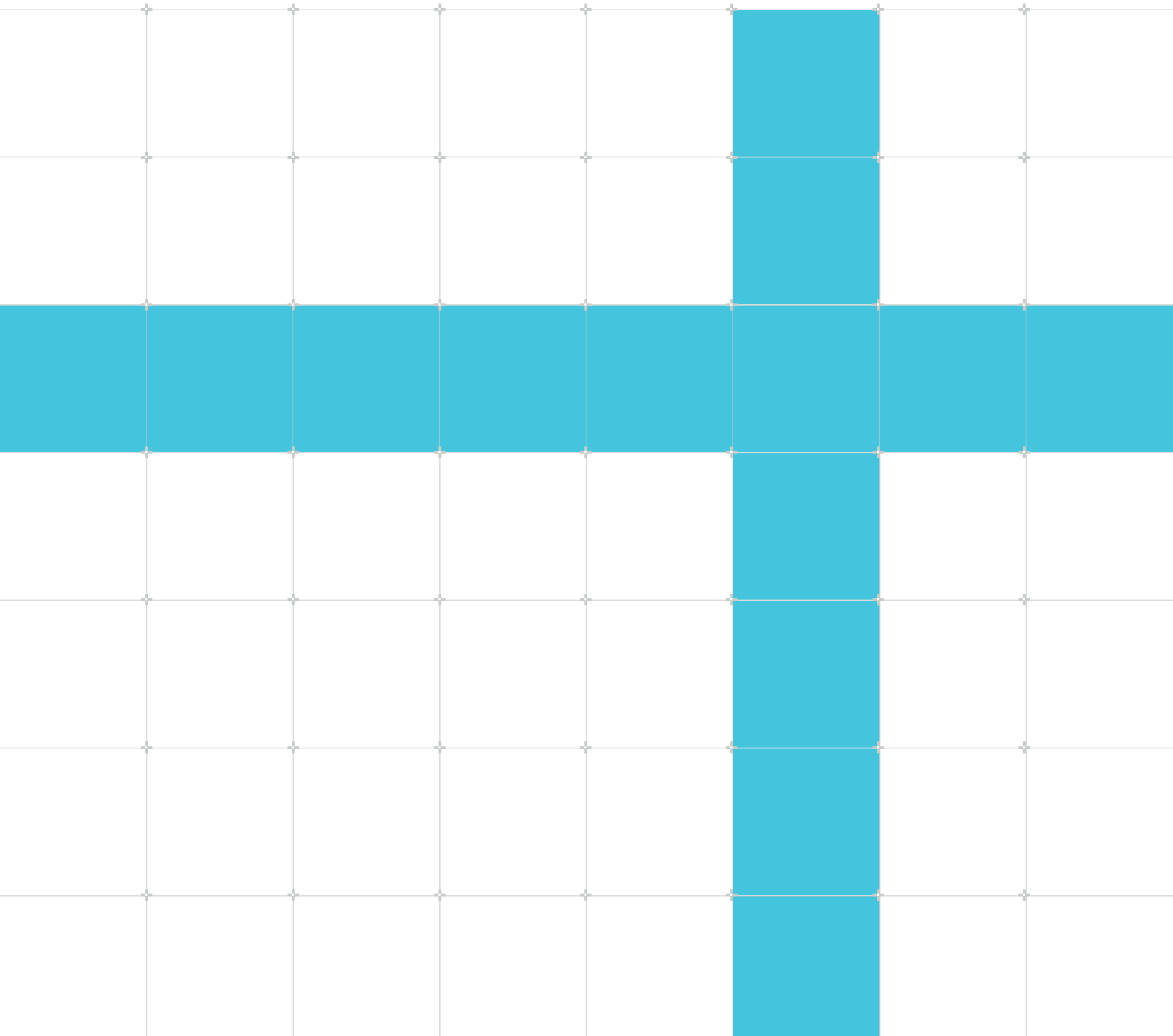
Mali-G76 Performance Counters

1.1

Reference Guide

Issue
102697_0101_en



Mali-G76 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

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

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-G76 performance counter reference

This guide explains the Mali performance counters found in the Arm Streamline profiling template for the Mali-G76 GPU, which is part of the Bifrost 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, and which application threads cause the workload.
- 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 by observing 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. This can be used to find optimization opportunities for varying-bound content that has been identified 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. This can be used to find optimization opportunities for texture-bound content that has been identified 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. This can be used to find optimization opportunities for memory-bound content that has been identified 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. This can be used to identify which type of workload is causing GPU memory accesses, helping to narrow down where optimizations should be targeted.

- GPU configuration: these utility counters expose the GPU configuration of the platform, allowing Streamline expressions to be created based on the specific platform configuration in the target device.

1.1 CPU performance

Many graphics performance issues are caused by high CPU load or poor scheduling of workloads across the CPU and GPU. The first part of the analysis template looks at the CPU workloads allowing you to identify regions where application performance was bound by CPU 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.

1.1.1 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 allows an assessment of how busy the CPUs were, although 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 of 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.

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

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

1.1.2 CPU cycles

The CPU cycle charts show the activity of each processor cluster, presented as the number of processor clock cycles used. Using this in conjunction with the CPU activity information can give an indication of the CPU operating frequency.

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

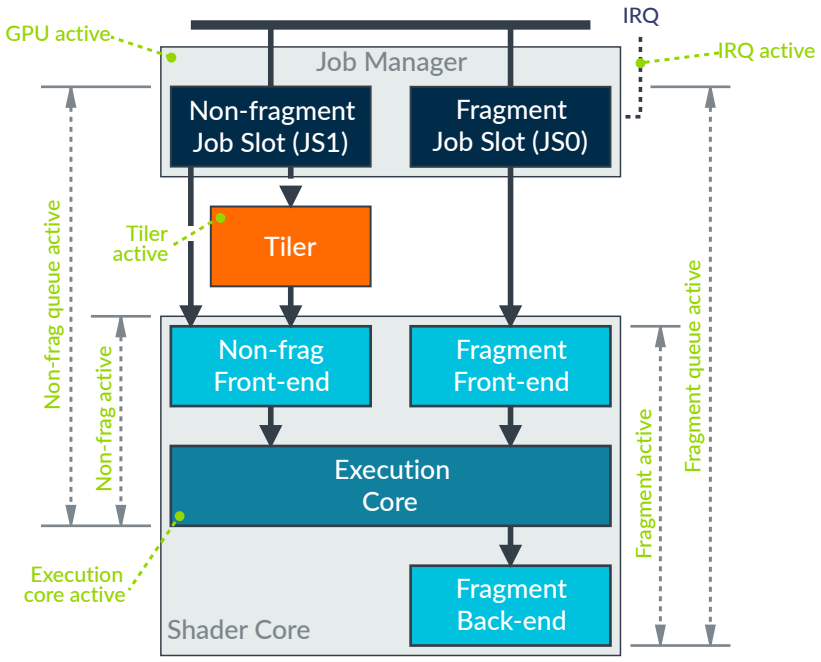
1.2 GPU activity

The Mali workloads running on Mali-G76 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. There is one queue for non-fragment workloads, which include compute shading and vertex shading, and one slot 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 Mali GPUs.

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

Figure 1-1: Bifrost GPU top level

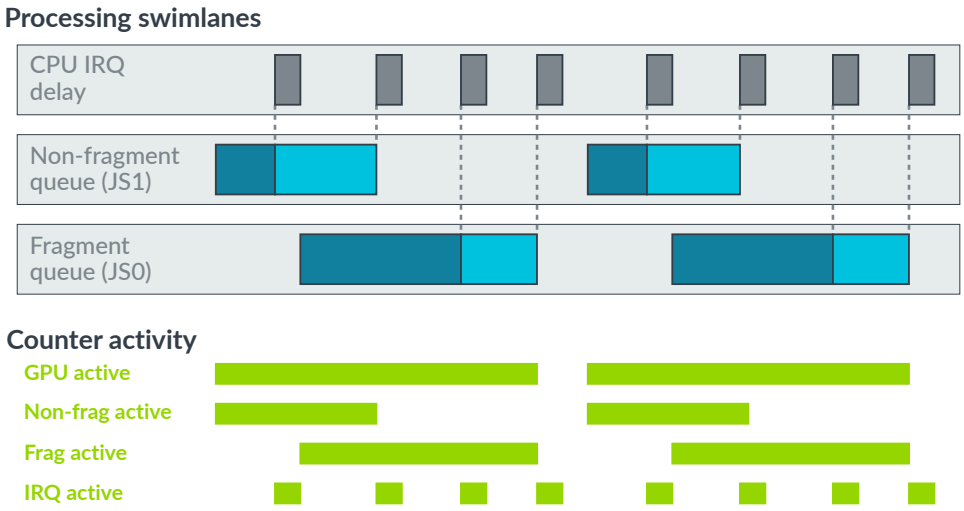


The “active” counters show that a data path or hardware unit was processing some workload, but do not necessarily indicate that that data path or unit was fully utilized. For example, the *Fragment queue active cycles* counter increments every cycle that there is any fragment workload queued to run anywhere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

Some counters are common to multiple data paths. For example, both non-fragment and fragment shader programs will run on the same unified shader core. If these different workload types are overlapping in the same counter sample, then shader core counter data will include contributions from both types of workload and cannot distinguish them individually.

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

Figure 1-2: Bifrost GPU top level timeline



This diagram shows two render passes per frame, shown in different shades of blue, each consisting of a single piece of non-fragment work that is executed before a single 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 will increment whenever any queue contains work.

1.2.1 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 be used to determine the dominant workload type submitted by the application. They can also be used to determine the effectiveness of workload scheduling at keeping the hardware queues running in parallel.

1.2.1.1 GPU active cycles

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

This counter will increment every clock cycle where any workload is present in a processing queue, even if the GPU is stalled waiting for external memory to return data; this is still counted as active time even though no forward progress is being made.

```
$MaliGPUCyclesGPUActive
```

1.2.1.2 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 can be 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 should be similar to the GPU active counter.

This counter will increment any clock cycle where a workload is loaded into a queue even if the GPU is stalled waiting for external memory to return data; this is still counted as active time even though no forward progress is being made.

```
$MaliGPUCyclesNonFragmentQueueActive
```

1.2.1.3 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 should be similar to the GPU active counter.

This counter will increment any clock cycle where a workload is loaded into a queue even if the GPU is stalled waiting for external memory to return data; this is still counted as active time even though no forward progress is being made.

```
$MaliGPUCyclesFragmentQueueActive
```

1.2.1.4 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 as well as providing the fixed-function tiling needed for Mali's 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* counters in the shader cores are very low relative to this.

```
$MaliGPUCyclesTilerActive
```

1.2.1.5 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* is a high percentage of *GPU active cycles*, there could be an underlying problem that is preventing the CPU from handling interrupts efficiently. This is normally a system integration issue, which cannot be worked around by an application developer.

```
$MaliGPUCyclesGPUInterruptActive
```

1.2.2 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 can 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 should be close to 100% utilization, with the other running in parallel to it only some of the time. The most heavily loaded queue should be the highest priority target for content optimization, as it is the critical path workload.

For GPU-bound content where the GPU is always busy, but the queues are running serially for all or part of the frame, application API usage may prevent parallel processing and reduce the achievable performance. This can be caused by:

- The application blocking and waiting for GPU activity to complete, for example by waiting on a query object result which is not yet available. This may cause one or more of the work queues to run out of new work to process.
- The application using conservative Vulkan pipeline barriers that prevent vertex workloads from a later render passes from overlapping with the fragment processing of an earlier render pass.
- The application submitting rendering workloads that have data dependencies across the queues which prevent parallel execution. For example, a fragment-compute-fragment data flow may mean that no processing can be executed in the fragment queue while the compute shader is running, if no non-dependent work is available.

Mobile systems use Dynamic Voltage and Frequency Scaling (DVFS), reducing voltage and clock frequency for light workloads, to improve energy efficiency. When seeing a workload with high percentage utilization, always check the *GPU active cycles* counter because the GPU might be highly utilized but running at a low clock frequency.

1.2.2.1 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, so the dominant queue should be close to 100% utilized. If no queue is dominant, but the GPU is close to 100% utilized, then there could be a serialization or dependency problem preventing better overlap across the queues.

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

1.2.2.2 Fragment queue utilization

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

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

1.2.2.3 Tiler utilization

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

Note that this measures the overall processing time for index-driven vertex shading (IDVS) workloads, in addition to the fixed function tiling process. It is not necessarily indicative of the runtime of the fixed-function tiling process itself.

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

1.2.2.4 Interrupt pending utilization

This expression defines the IRQ pending utilization compared against the GPU active cycles. In a well-functioning system this expression should be less than approximately 2% of the total cycles. If the value is much higher than this then there may be a system issue preventing the CPU from efficiently handling interrupts.

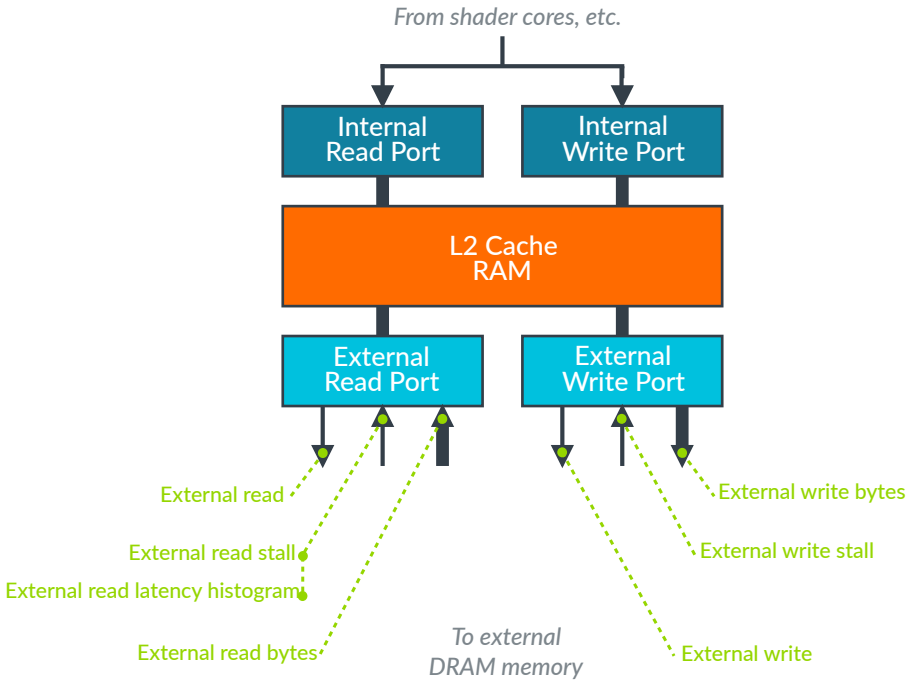
```
max(min(($MaliGPUCyclesGPUInterruptActive / $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

1.2.3 External memory bandwidth

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.

The Mali performance counters measure the memory accesses that are external to the GPU, but note that they may not be external to the system-on-a-chip if there are additional layers of system cache between the GPU and external DRAM.

Figure 1-3: Bifrost GPU memory system



Memory accesses to external DRAM are very power intensive. A good rule of thumb 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 consumption are a high priority for mobile application development.

1.2.3.1 Output external read bytes

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

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

1.2.3.2 Output external write bytes

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

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

1.2.4 External memory stalls

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 memory system can provide, so optimizations to reduce memory bandwidth should be attempted.

1.2.4.1 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)
```

1.2.4.2 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)
```

1.2.5 External memory read latency

The external memory read latency counters present a histogram of access latencies. High latency accesses can reduce performance, and are normally an indication that the application is requesting more data than the memory system can provide. Optimizations that reduce memory bandwidth usage should be attempted.

1.2.5.1 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 is considered a fast access response speed.

```
$MaliExternalBusReadLatency0127Cycles
```

1.2.5.2 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 is considered a normal access response speed.

```
$MaliExternalBusReadLatency128191Cycles
```

1.2.5.3 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 is considered a normal access response speed.

```
$MaliExternalBusReadLatency192255Cycles
```

1.2.5.4 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 is considered a slow access response speed.

```
$MaliExternalBusReadLatency256319Cycles
```

1.2.5.5 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 is considered a slow access response speed.

```
$MaliExternalBusReadLatency320383Cycles
```

1.3 Content behavior

Rendering performance is strongly influenced by the content being rendered by the application. Optimizing rendering requires both efficient content to be provided, and efficient handling of that

content by the GPU. The content behavior metrics help developers 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, which means it takes longer to render than it should.
- 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 bullets, looking at the size and efficiency of the workload that has been submitted.

1.3.1 Geometry usage

The vertex stream is the first application input processed by the GPU rendering pipeline. This set of 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. It is important that high detail geometry is used only when needed. Pseudo-geometry techniques, such as normal mapping, should be preferred to using vertex-based geometry whenever possible. Dynamic mesh level-of-detail, using simpler meshes when objects are further from the camera, should be used to avoid micro-triangle geometry ending up on-screen.

1.3.1.1 Total input primitives

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

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

1.3.1.2 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 are 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 for overly large batch sizes.

```
$MaliPrimitiveCullingFacingAndXYPlaneTestCulledPrimitives
 + $MaliPrimitiveCullingZPlaneTestCulledPrimitives +
 $MaliPrimitiveCullingSampleTestCulledPrimitives
```

1.3.1.3 Visible primitives

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

Note that visible means only that the primitive is front-facing and inside the visible clip volume. It still may produce no visible output on the screen if it is occluded by other primitives closer to the camera.

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

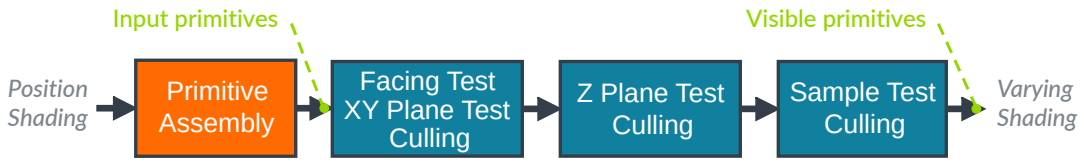
```
$MaliPrimitiveCullingVisiblePrimitives
```

1.3.2 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. This set of counters helps to identify the reasons why triangles are being culled, allowing you to correctly target optimizations at the area causing problems.

The Mali culling pipeline executes in the order shown below, and the counters in this section show the percentage of the primitives that enter each pipeline stage that are killed by it. Note that the percentages are relative to the per-stage input, not the total geometry input, so do not add up to 100%.

Figure 1-4: Bifrost GPU culling pipeline



1.3.2.1 Visible primitives rate

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

For 3D content it is typically expected that 50% of primitives are visible, due to the use of back-face culling. Significantly lower visibility rates may indicate missing optimizations.

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

1.3.2.2 Facing or XY plane test cull rate

This expression defines the percentage of primitives entering the facing and XY plane test that are killed by it. This is triggered by primitives that are outside of the view frustum in the X or Y axis, or triangles which are inside the frustum and facing away from the camera (back-facing).

It is expected that approximately 50% of primitives are killed at this stage; these are triangles that are in-frustum, but back-facing. Seeing significantly more than 50% are killed can be indicative of insufficient software culling, resulting in out-of-frustum meshes being sent to the GPU.

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

1.3.2.3 Z plane test cull rate

This expression defines the percentage of primitives entering the Z plane culling test that are killed by it. This is triggered by primitives that are closer than the frustum near clip plane, or further away than the frustum far clip plane.

Seeing a significant proportion of triangles are killed at this stage can be indicative of insufficient application software culling, resulting in out-of- frustum meshes being sent to the GPU.

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

1.3.2.4 Sample test cull rate

This expression defines the percentage of primitives entering the sample coverage test that are killed by it. This is triggered by triangles that are so small that they hit no rasterizer sample points.

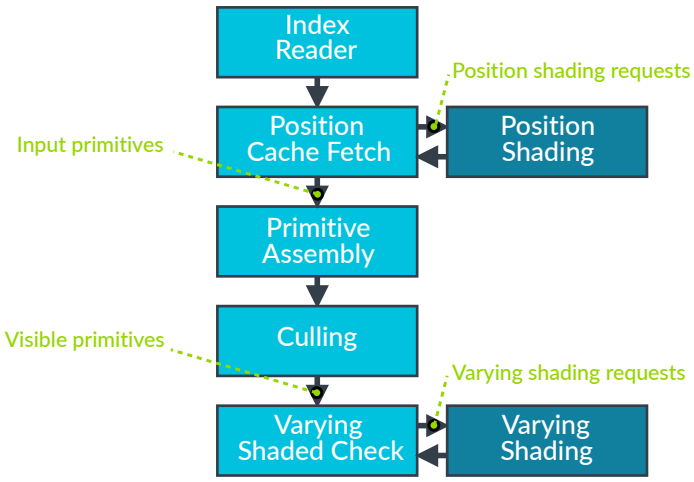
If a significant proportion of triangles are killed at this stage, it is indicative that the application is using geometry meshes that are too complex for their screen coverage. Aim to keep triangle screen area above 10 pixels, using schemes such as dynamic mesh level-of-detail to select simpler meshes as objects move further away from the camera.

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

1.3.3 IDVS shading

This GPU uses an optimized Index-Driven Vertex Shading (IDVS) processing pipeline. In this pipeline the vertex position is computed before culling, and the remaining varyings are computed after culling for any visible vertices. Counters monitoring the number of position and varying threads spawned per primitive can be used to measure mesh encoding efficiency.

Figure 1-5: Bifrost 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 in the index buffer can result in a vertex being shaded multiple times, because it can be evicted from the cache before it is reused.

This pipeline submits shading requests in groups of 4 contiguous index values. Unused index locations may be shaded if they are adjacent to used index locations. Reduce redundant shading by ensuring meshes use every index between the min and max index, without any holes.

1.3.3.1 Position shader thread invocations

This expression defines the number of position shader thread invocations.

```
$MaliTilerShadingRequestsPositionShadingRequests * 4
```

1.3.3.2 Varying shader thread invocations

This expression defines the number of varying shader thread invocations.

```
$MaliTilerShadingRequestsVaryingShadingRequests * 4
```

We can usefully normalize these counters to show the amount of shading per primitive, which gives a direct measure of mesh encoding efficiency.

1.3.3.3 Position threads per input primitive

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

Efficient meshes with a high degree of vertex reuse tend to have average less than 1.5 vertices shaded per triangle, as the vertex computation can be shared by multiple adjacent primitives.

Inefficient meshes with no vertex reuse will shade at 3 vertices per triangle, but may require more than 3 if positions are reshaded or if redundant index locations are shaded.

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

1.3.3.4 Varying threads per input primitive

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

Efficient meshes with a high degree of vertex reuse tend to have average less than 1.5 vertices shaded per visible triangle, as the vertex computation can be shared by multiple adjacent primitives.

Inefficient meshes with no vertex reuse will shade at 3 vertices per visible triangle, but may require more than 3 if positions are reshaded or if redundant index locations are shaded.

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

1.3.4 Fragment overview

Fragment overview counters look at the GPU processing workload being requested in terms of the total number of output pixels shaded, the average number of GPU cycles spent per pixel, and the average numbers of fragments shaded per output pixel.

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

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

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

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

This can help set 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 will have a cycle budget of just 10 cycles per pixel, which must cover all processing costs, including vertex shading and fragment shading. It is definitely possible to write content inside this budget, but ensure you spend each cycle wisely if you want to achieve the best graphics fidelity.

1.3.4.1 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 the underlying hardware counter rounds the width and height values of the rendered surface to be 32-pixel aligned, even if those pixels are not actually processed during shading because they are out of the active viewport and/or scissor region.

```
$MaliGPUTasksFragmentTasks * 1024
```

1.3.4.2 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 the target resolution and frame rate you want to achieve. 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, especially if you have multiple render passes per frame, so those cycles must be used wisely.

```
$MaliGPUCyclesGPUActive / ($MaliGPUTasksFragmentTasks * 1024)
```

1.3.4.3 Fragments per pixel

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

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

```
($MaliCoreWarpsFragmentWarps * 8 * $MaliConstantsShaderCoreCount) /
 ($MaliGPUTasksFragmentTasks * 1024)
```

1.3.5 Fragment depth and stencil testing

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

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

1.3.5.1 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 draw calls that can modify their own depth value by writing to the fragment depth.

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

1.3.5.2 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 ensuring depth testing is enabled, and avoiding draw calls that can modify their own coverage by using shader discard or alpha-to-coverage, or that modify their own depth value by writing to the fragment depth.

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

1.3.5.3 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 may be indicative of 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)
```

1.3.5.4 FPK killed quad percentage

This expression defines the percentage of rasterized quads that are killed by Mali's 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 may be indicative of 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 * 8) / 4)) / $MaliCoreQuadsRasterizedFineQuads) *
 100, 100), 0)
```

1.3.5.5 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 hitting late-ZS can cause slow performance, even

if fragments are not killed, as younger fragments at a coordinate cannot complete early-ZS until all older fragments at that coordinate have completed any pending late ZS operations.

For application shaders, use of late-ZS testing can be caused by shaders with mutable coverage, mutable depth, or side-effects on shared resources in memory. The driver will also generate late-ZS updates to load a depth or stencil attachment from memory at the start of a render pass, which is needed if the pass does not start from a cleared depth value.

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

1.3.5.6 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 will execute at least some of their fragment program before being killed, so a significant number of quads being killed at late ZS testing can indicates a significant overhead. Aim to minimize the number of quads using and being killed by late ZS testing.

For application shaders, use of late-ZS testing can be caused by shaders with mutable coverage, mutable depth, or side-effects on shared resources in memory.

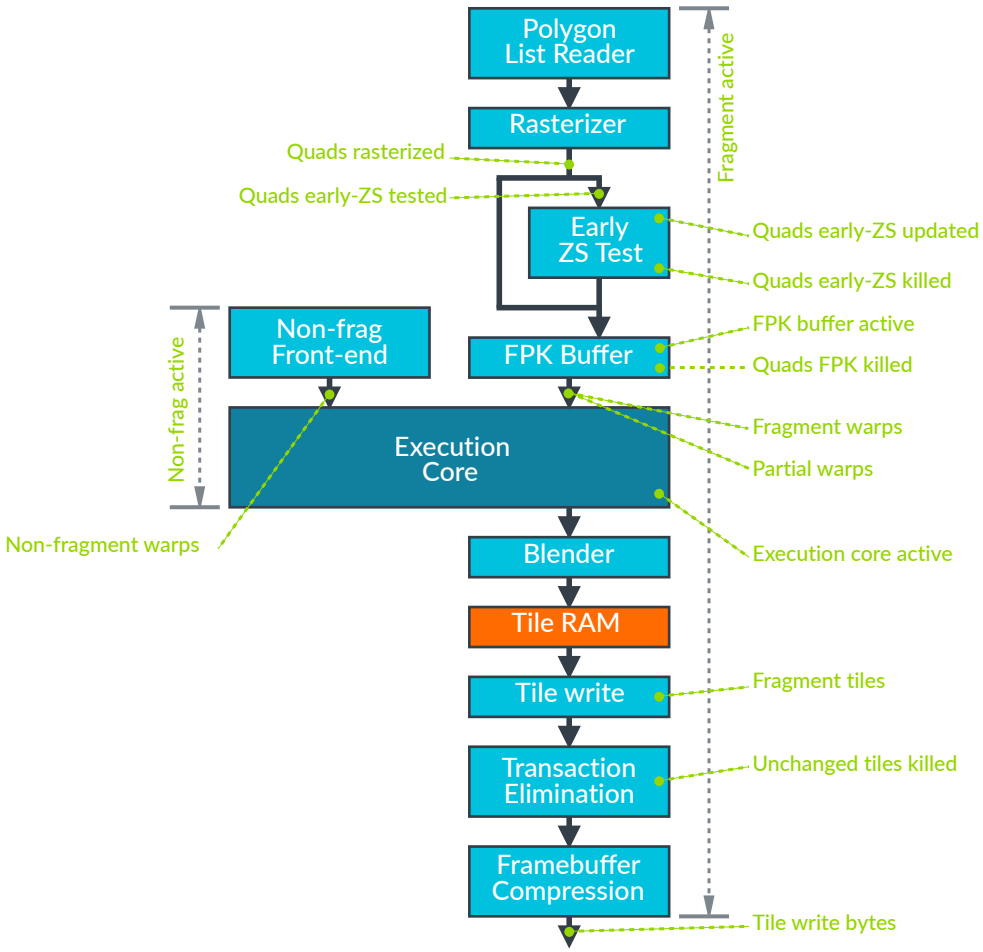
The driver will also generate late-ZS updates to load a depth or stencil attachment from memory at the start of a render pass, which is needed if the pass does not start from a cleared depth value. These fragments will show as a late-ZS kill, as no shader execution is needed once the depth or stencil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

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

1.4 Shader core data path

Each Mali 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 1-6: Bifrost GPU shader core



1.4.1 Shader core workload

The warp counters count the number of shader warps issued for the two workload types. For Mali-G76 each warp contains 8 threads.

1.4.1.1 Non-fragment warps

This counter increments for every created non-fragment warp. For this GPU warps contain 8 threads.

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

```
$MaliCoreWarpsNonFragmentWarps
```

1.4.1.2 Fragment warps

This counter increments for every created fragment warp. For this GPU warps contain 8 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 will still consume shader resource, which makes small triangles disproportionally expensive.

```
$MaliCoreWarpsFragmentWarps
```

1.4.2 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 and of any resource sharing inside the shader core.

1.4.2.1 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 may not be a direct measure of processing cost for content that is sensitive to memory access latency. In addition there will be some crosstalk 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 * 8)
```

1.4.2.2 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 may not be a direct measure of processing cost for content which is sensitive to memory access latency. In addition there will be

some crosstalk 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.

```
$MaliCoreCyclesFragmentActive / ($MaliCoreWarpsFragmentWarps * 8)
```

1.4.3 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 can help guide optimizations, and identifying lack of parallelism can confirm presence of scheduling problems.

1.4.3.1 Non-fragment utilization

This expression defines the percentage utilization of the shader core non- fragment path. This counter will monitor 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 / $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

1.4.3.2 Fragment utilization

This expression defines the percentage utilization of the shader core fragment path. This counter will monitor 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 / $MaliGPUCyclesGPUActive) * 100, 100), 0)
```

1.4.3.3 Fragment FPK buffer utilization

This expression defines the percentage of cycles where the forward pixel kill (FPK) quad buffer, which is located after early-ZS but before the execution core, contains at least one fragment quad.

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

- The running workload has many empty tiles with no geometry to render. This can be 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 causes the shader core to complete fragments faster than they can be rasterized, so the quad buffer will start to drain,

- The application consists of layers which stall at early-ZS due to a dependency on an earlier fragment layer which is still in flight. This prevents new fragments entering the quad buffer, so the quad buffer will start to drain.

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

1.4.3.4 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 could be used.

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 can be 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 / $MaliGPUCyclesGPUActive) * 100, 100),
 0)
```

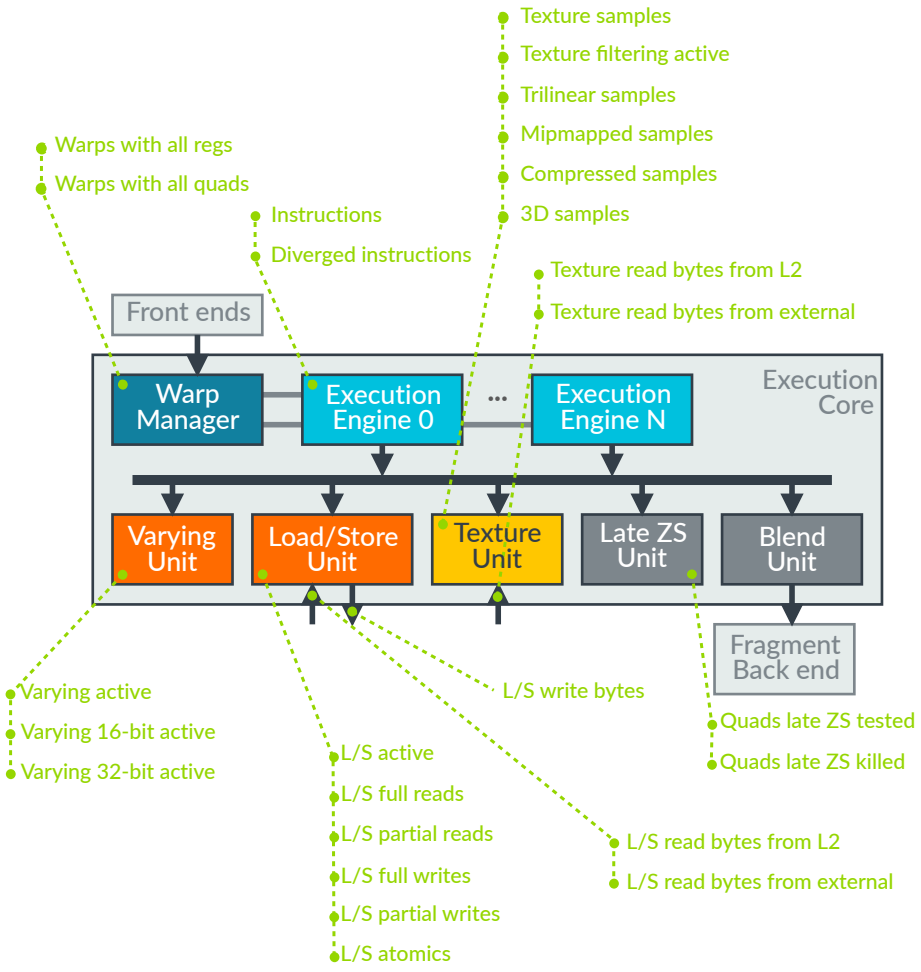
1.5 Shader core functional units

A shader core consists of multiple parallel processing units, providing both programmable and fixed-function operations. Mali counters can track utilization and workload type for all of 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, reducing precision of the operations to use 16-bit types may allow multiple operations to be performed in parallel.

For thermally-bound content reducing the critical path load will give the biggest gain as it may allow lower operating frequency to be used. However, reducing load on any functional unit will help improve energy efficiency.

Figure 1-7: Bifrost GPU execution core



1.5.1 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, and the most heavily utilized functional unit should be the target for optimizations to improve performance, although reducing the load of any unit will be good for energy efficiency.

1.5.1.1 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)
```

1.5.1.2 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
 + $MaliCoreVaryingIssues16BitInterpolationSlots) /
 $MaliCoreCyclesExecutionCoreActive) * 100, 100), 0)
```

1.5.1.3 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)
```

1.5.1.4 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, such as vertex attribute access, buffer access, work group shared memory access, and stack access. This unit also implements imageLoad/Store and atomic access functionality.

For most traditional vertex and fragment content the most significant contributor to load/store usage is vertex data, so simplifying mesh complexity (fewer triangles, fewer vertices, and/or fewer bytes per vertex) is recommended. However, this is dependent on the application workload.

```
max(min((($MaliCoreLoadStoreCyclesFullReadCycles +
 $MaliCoreLoadStoreCyclesPartialReadCycles + $MaliCoreLoadStoreCyclesFullWriteCycles
```

```
 + $MaliCoreLoadStoreCyclesPartialWriteCycles +
 $MaliCoreLoadStoreCyclesAtomicAccessCycles) / $MaliCoreCyclesExecutionCoreActive) *
 100, 100), 0)
```

1.5.2 Shader program properties

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

1.5.2.1 Diverged instruction issue rate

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

Control flow divergence can reduce performance, because only some lanes of the warp are active while diverged. Minimize divergence by using warp-uniform branch decisions for conditional checks and loop limits.

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

1.5.2.2 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, which can make shader performance more sensitive to cache misses and memory stalls.

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

1.5.3 Shader workload properties

Shader workload property counters track a variety of 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.

1.5.3.1 Partial coverage rate

This expression defines the percentage of warps quads that contain fragment 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(($MaliCoreWarpsPartialFragmentWarps / $MaliCoreWarpsFragmentWarps) * 100,
 100), 0)
```

1.5.3.2 Full quad warp rate

This expression defines the percentage of warps that are fully populated with quads. If there are many warps that are not full then performance may be lower, because thread slots in the warp are unused. 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)
```

1.5.3.3 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 / $MaliCoreTilesTiles) * 100, 100), 0)
```

1.6 Shader core varying unit

Varying unit counters monitor use of the varying interpolation in fragment shaders. If the shader core utilization counters show that this unit is a bottleneck, these counters may provide some indication of optimization opportunities.

The interpolator has a 32-bit data path per thread, which can be used to interpolate a vec2 16-bit value in a single cycle, so 16-bit interpolation is twice as fast as 32-bit interpolation. We recommend 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 will interpolate faster than a separate vec3 + float pair.

1.6.1 Varying unit usage

These counters show the usage of the varying unit, which is used for all vertex attribute interpolation in fragment shaders.

1.6.1.1 Varying cycles

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

```
$MaliCoreVaryingIssues32BitInterpolationSlots +
 $MaliCoreVaryingIssues16BitInterpolationSlots
```

1.6.1.2 16-bit interpolation cycles

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

```
$MaliCoreVaryingIssues16BitInterpolationSlots
```

1.6.1.3 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
```

1.7 Shader core texture unit

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 might provide some indication of optimization opportunities.

1.7.1 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 Bifrost family. For Mali-G76 the best case performance (bilinear filtered samples) is 0.5 cycle(s) per sample.

1.7.1.1 Texture filtering cycles

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

- 2D bilinear filtering takes two cycles.
- 2D trilinear filtering takes four cycles.
- 3D bilinear filtering takes four cycles.
- 3D trilinear filtering takes eight cycles.

Using anisotropic filtering will make multiple filtered sub-samples 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
```

1.7.1.2 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 (2 sample(s) per cycle), consider using simpler texture filters. See *Texture filtering cycles* for details of the expected performance for different types of operation.

```
$MaliCoreTextureCyclesTexturingActive / ($MaliCoreTextureQuadsTextureRequests * 4)
```

1.7.2 Texture unit workload properties

These counters show the content behavior in the texture unit, in terms of the number of accesses using texture compression, mipmapping, or trilinear filtering probes.

1.7.2.1 Texture data fetches from compressed lines

This expression defines the percentage of texture line fetches that are from textures using API block compression, such as ASTC or ETC2.

Texture compression significantly reduces memory bandwidth consumption, so aim to maximize this percentage by compressing all offline-authored texture assets. However, note that this

counter does not include texture data fetches from framebuffers compressed with AFBC lossless compression.

```
max(min(($MaliCoreTextureLineFetchesCompressedLineFetches /
 $MaliCoreTextureLineFetchesLineFetches) * 100, 100), 0)
```

1.7.2.2 Texture accesses using trilinear filter percentage

This expression defines the percentage of texture operations using trilinear filtering. Trilinear samples are twice the cost of simple bilinear samples.

```
max(min(($MaliCoreTextureQuadsTrilinearFilteredIssues /
 $MaliCoreTextureQuadsTextureRequests) * 100, 100), 0)
```

1.7.2.3 Texture accesses using mipmapping percentage

This expression defines the percentage of texture operations accessing mipmapped textures.

Mipmapping significantly improves image quality and memory bandwidth in 3D scenes with variable object view distance. Aim to enable mipmapping for all offline-authored texture assets in 3D content.

```
max(min(($MaliCoreTextureQuadsMipmappedTextureIssues /
 $MaliCoreTextureQuadsTextureIssues) * 100, 100), 0)
```

1.7.3 Texture unit memory usage

These counters show the average number of bytes read from the L2 cache or external memory per texture sample.

1.7.3.1 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 well textures are being cached in the L1 texture cache. If a high number of bytes are being requested per access, where high depends on the size of the texture formats you are using, it can be worth reviewing texture settings:

- Enable mipmaps for offline generated textures
- Use ASTC or ETC compression for offline generated textures
- Replace run-time generated framebuffer and texture formats with a narrower format
- Reduce use of imageLoad/Store in OpenGL ES and Vulkan, as this prevents use of framebuffer compression.
- Reduce any use of negative LOD bias used for texture sharpening

- Reduce the MAX_ANISOTROPY level for anisotropic filtering

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

1.7.3.2 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 well textures are being cached in the L2 cache. If a high number of bytes are being requested per access, where high depends on the size of the texture formats you are using, it can be worth reviewing texture settings:

- Enable mipmaps for offline generated textures
- Use ASTC or ETC compression for offline generated textures
- Replace run-time generated framebuffer and texture formats with a narrower format
- Reduce use of imageLoad/Store in OpenGL ES and Vulkan, as this prevents use of framebuffer compression.
- Reduce any use of negative LOD bias used for texture sharpening
- Reduce the MAX_ANISOTROPY level for anisotropic filtering

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

1.8 Shader core load/store unit

Load/store unit counters show the use of the general purpose load/store cache. This unit is used for all shader memory accesses except texturing and frame buffer write-back. This includes buffer, image and stack access.

1.8.1 Load/store unit usage

The unit usage counters show the content behavior in the load/store unit, in terms of the number of reads and writes being made, and whether those loads use the full width of the available data path.

An important memory access optimization for compute shaders is to make effective use of the data width the load/store interface provides. We recommend shaders make vector memory accesses in each thread, and to ensure that threads in the same warp access overlapping or sequential addresses inside a single 64 byte address range.

1.8.1.1 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 best case cycle usage.

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

1.8.1.2 Load/store full read issues

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

```
$MaliCoreLoadStoreCyclesFullReadCycles
```

1.8.1.3 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, so efficiency can be improved by merging short accesses together to make fewer larger access requests. 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
```

1.8.1.4 Load/store full write issues

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

```
$MaliCoreLoadStoreCyclesFullWriteCycles
```

1.8.1.5 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, so efficiency can be improved by merging short accesses together to make fewer larger access requests. 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
```

1.8.1.6 Load/store atomic issues

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

```
$MaliCoreLoadStoreCyclesAtomicAccessCycles
```

1.8.2 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. This can be used to see how well workloads are using the GPU L1 and L2 caches, although knowing what “good” looks like requires knowledge of the algorithm being executed.

1.8.2.1 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 well data is being cached in the L1 load/store cache. If a high number of bytes are being requested per access, where high depends on the buffer formats you are using, it can be worth reviewing data formats and access patterns.

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

1.8.2.2 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 well data is being cached in the L2 cache. If a high number of bytes are being requested per access, where high depends on the texture formats you are using, it can be worth reviewing data formats and access patterns.

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

1.8.2.3 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)
```

1.9 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. This data is broken down by unit making the access, allowing the sources of memory bandwidth to be identified and targeted for optimization.

1.9.1 Read access from L2 cache

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

1.9.1.1 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
```

1.9.1.2 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
```

1.9.1.3 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
```

1.9.2 Read access from external memory

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 may be fetched from a layer of system cache, external to the GPU, or from the main system DRAM.

1.9.2.1 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
```

1.9.2.2 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
```

1.9.2.3 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
```

1.9.3 Write access

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

1.9.3.1 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
```

1.9.3.2 Tile buffer write bytes

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

```
$MaliCoreWritesTileBufferWriteBeats * 16
```

1.10 GPU configuration

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

1.10.1 GPU configuration counters

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.

1.10.1.1 Shader core count

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

```
$MaliConstantsShaderCoreCount
```

1.10.1.2 L2 cache slice count

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

```
$MaliConstantsL2SliceCount
```

1.10.1.3 External bus beat size

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

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