arm

# The Valhall shader core

Version 1.0

**Non-Confidential**

**Issue 02**

102203_0100_02_en

# The Valhall shader core

## Release information

### Document history

| Issue | Date | Confidentiality | Change |
|---|---|---|---|
| 0100-02 | 19 August 2020 | Non-Confidential | First release |

## Proprietary Notice

## Confidentiality Status

## Product Status

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

## Feedback

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

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

## Inclusive language commitment

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

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

# Contents

1. Overview........................................6

2. Fourth-generation Mali GPU architecture........................................7

3. Valhall shader core........................................9

4. Texture unit........................................13

5. Index-driven geometry pipeline........................................14

6. Check your knowledge........................................15

7. Related information........................................16

8. Next steps........................................17

# 1. Overview

This guide describes the top-level layout and the benefits of and shader core functionality of a typical Mali Valhall GPU programmable core. Valhall is the fourth generation of Mali GPUs. The Valhall family includes the Mali-G5x and Mali-G7x series of products. These products were released from 2018 onwards.

When optimizing applications using a GPU, it is useful to have at least a high-level mental model for how the underlying hardware works. It is also useful to understand the expected performance of the hardware, and the data rates for the different types of operations that it might perform. Understanding the block architecture is particularly important when optimizing using the Mali performance counters. This is because understanding what the counters are telling you implies a need to understand the blocks that the counters are tied to.

By the end of this guide, you will understand how the Mali Valhall series of GPUs perform shader core operations using shared access to the L2 cache. You will also learn the benefits of the warp manager and the processing engines within the execution core, and the benefits of the Index-Driven Vertex Shading (IDVS) geometry pipeline.

## Before you begin

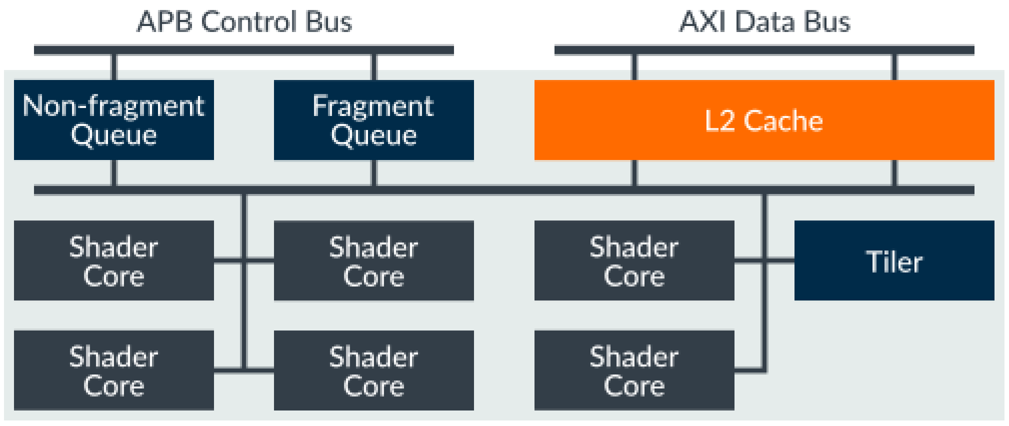
Before you work through this guide, we assume that you have read our introductory-level guide on tile-based rendering. This guide builds on concepts that are introduced in that guide.

# 2. Fourth-generation Mali GPU architecture

The Valhall family of Mali GPUs uses the same top-level architecture as the earlier Bifrost GPUs. The Valhall family uses a unified shader core architecture. This means that only a single type of hardware shader processor exists in the design. This single processor type can execute all types of shader code, including vertex shaders, fragment shaders, and compute kernels.

The exact number of shader cores that are present in a silicon chip can vary. At Arm we license configurable designs to our silicon partners. Partners choose how to configure the GPU in their chipset based on their performance needs and silicon area constraints.

The following diagram provides a top-level overview of the control bus and data bus of a typical Mali Valhall GPU:



To improve performance, and to reduce memory bandwidth wastage that is caused by repeated data fetches, the shader cores in the system all share access to a level 2 cache. The size of the L2 cache is configurable by our silicon partners, is typically in the range of 64-128KB for each shader core in the GPU. However, the size of the L2 cache depends on how much silicon area is available.

Also, our silicon partners can configure the number, and bus width, of the memory ports that the L2 cache has to external memory.

The Valhall architecture aims to write two 32-bit pixel per core per clock. Therefore, it is reasonable to expect a 4-core design to have a total of 256 bits of memory bandwidth, for both read and write, per clock cycle. This can vary between chipset implementations.

## Work issue

When the application has completed defining the render pass, the Mali driver submits a pair of independent workloads for each render pass.

One section deals with all geometry-related and compute-related workloads in the pass, and the other section is for the fragment workload. Mali GPUs are tile-based renderers, all geometry processing for a render pass must be complete before the fragment shading can start. This is because we need a finalized tile list to provide fragment processing with the primitive coverage information that it needs.

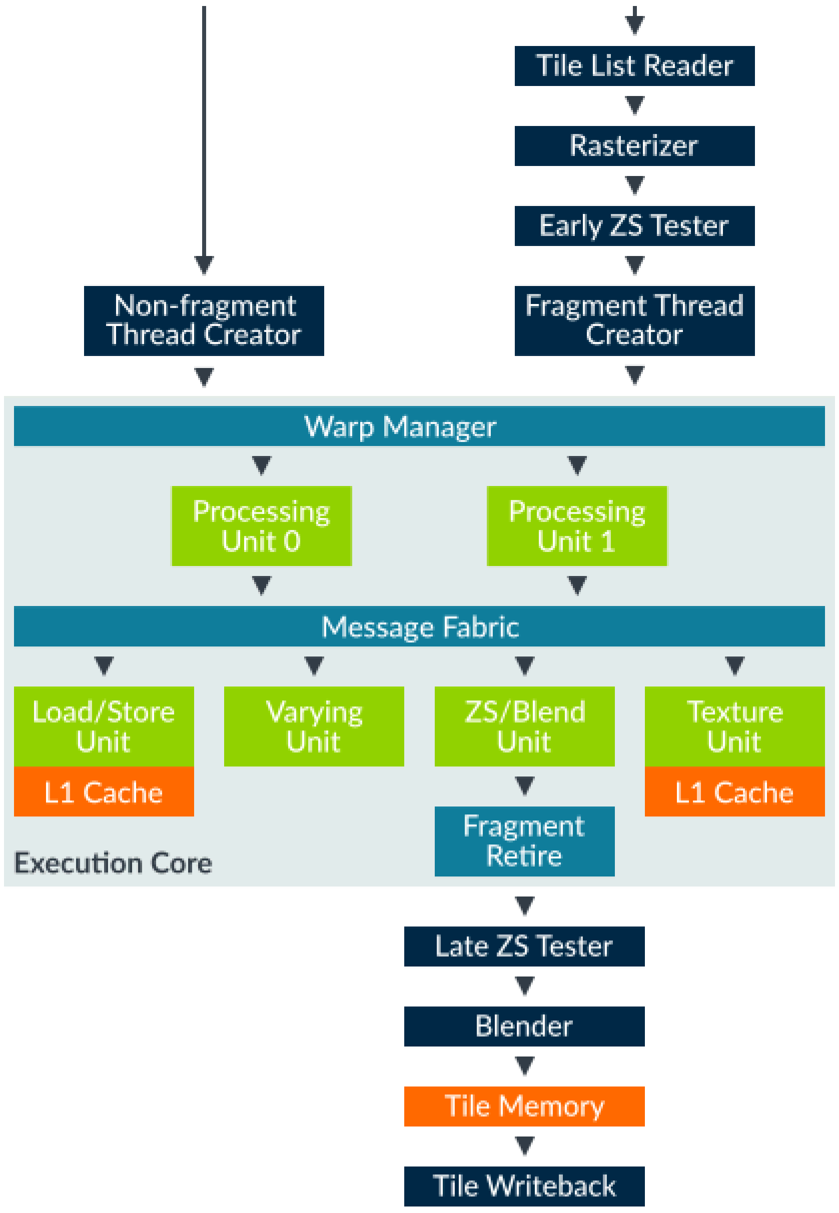
The hardware supports two parallel issue queues which the driver can use, one for each workload type. Workloads from both queues can be processed by the GPU at the same time. This means that geometry processing and fragment processing for different render passes can be running in parallel.

The workload for a single render pass is nearly always large and highly parallelizable. This means that the GPU hardware will break the workload into smaller pieces and distribute it across all of the shader cores that are available in the GPU.

# 3. Valhall shader core

All Mali shader cores are structured as a number of fixed-function hardware blocks that are wrapped around a programmable core. The programmable core is the largest area of change in the Valhall GPU family, with some significant changes from the earlier Bifrost designs.

The following image shows a single execution core implementation and the fixed-function units that surround it:



The Valhall programmable Execution Core (EC) consists of one or more Processing Engines (PEs). The Mali-G57 and Mali-G77 have two PEs, and several shared data processing units, all of which are linked by a messaging fabric.

A Valhall core can perform 32 FP32 FMAs, read four bilinear filtered texture samples, blend two fragments, and write two pixels per clock.

## The processing engines

Each PE executes the programmable shader instructions.

Each PE includes three arithmetic processing pipelines:

- FMA pipeline with is used for complex maths operations
- CVT pipeline which is used for simple maths operations
- SFU pipeline which is used for special functions

The FMA and SVT pipelines are 16-wide, the SFU pipeline is 4-wide and runs at one quarter of the throughput of the other two.

## Thread state

Programs can use up to 32x32-bit registers while maintaining full thread occupancy. More complex programs can use to up 64 at the expense of reduced thread count.

## Arithmetic processing

The arithmetic units in the Valhall PE implement a warp-based vectorization scheme to improve functional unit utilization. To improve processing speed, multiple threads are grouped into a set of bundle, which is called a warp, before being executed in parallel by the GPU. Valhall uses 16-wide warps.

For of a single thread, this architecture looks like a stream of scalar 32-bit operations. This means that achieving high utilization of the hardware is a relatively straight forward task for the shader compiler. However, the underlying hardware keeps the efficiency benefit of being a vector unit with a single set of control logic. This unit can be amortized over all of the threads in the warp.

The two following diagrams illustrate the advantages of keeping the hardware units busy, regardless of the vector length in the program.

In this example, a vec3 arithmetic operation maps onto a pure Single Instruction Multiple Data (SIMD) unit, where a pipeline executes one thread per clock:

| | | | | |
|---|---|---|---|---|
| Cycle 1 | T1.x | T1.y | T1.z | Idle |
| Cycle 2 | T2.x | T2.y | T2.z | Idle |
| Cycle 3 | T3.x | T3.y | T3.z | Idle |
| Cycle 4 | T4.x | T4.y | T4.z | Idle |

By contrast, in this example the pipeline in a 4-wide warp unit, executes one lane per thread for four threads per clock:

| | | | | |
|---|---|---|---|---|
| Cycle 1 | T1.x | T2.x | T3.x | T4.x |
| Cycle 2 | T1.y | T2.y | T3.y | T4.y |
| Cycle 3 | T1.z | T2.z | T3.z | T4.z |

Valhall maintains native support for int8, int16, and fp16 data types. These data types can be packed using SIMD instructions to fill each 32-bit data processing lane. This arrangement maintains the power efficiency and performance that is provided by the types that are narrower than 32-bits.

A single 16-wide warp maths unit can therefore perform 32x fp16/int16 operations per clock cycle, or 64x int8 operations per clock cycle.

### Load/store unit

The load/store unit is responsible for all shader memory accesses which are not related to texture samplers. This includes generic pointer-based memory access, buffer access, atomics, and imageLoad() and imageStore() accesses for mutable image data.

Data that is stored in a single 64-byte cache line per clock cycle can be accessed. Warp unit accesses are optimized to reduce unique cache access requests. For example, data can be returned in a single cycle if all threads access data inside the same cache line.

We recommend that you use shader algorithms that exploit the wide data access, and the cross-thread merging functionalities, of the load/store unit.

For example:

- Use vector loads and stores in each thread.
- Access sequential address ranges across the threads in a warp

The load/store unit includes 16KB L1 data cache per core, which is backed up by the shared L2 cache.

## Varying unit

The varying unit is a dedicated fixed-function varying interpolator. The varying unit ensures good functional unit utilization by using warp vectorization.

For every clock, the varying unit can interpolate 32 bits for every thread in a warp. For example, interpolating a mediump - fp16 - vec4 takes two cycles. Therefore, interpolation of a mediump fp16 is both faster, and more energy efficient, than interpolation of a highp fp32.

# 4. Texture unit

The texture unit implements all texture memory accesses.

The baseline performance is four bilinear filtered texels per clock. This means that a 2x2 pixel quad is textured in one cycle. This is true for most texture formats, but performance can vary for some texture formats and filtering modes.

The following table shows the operations available, and their performance scaling values:

| Operation | Performance scaling |
|---|---|
| N x anisotropic filter | Up to x N |
| Trilinear filter | x2 |
| 3D format | x2 |
| More than 32-bit per texel | x2 |
| Filtered fp32 | x2 |

For example, the worst-case cost of a 4x anisotropic filter, using trilinear samples applied to a RGBA16F texture, is 16 times more expensive than a basic bilinear filter (4×2×2=16).

## ZS and blend unit

The ZS and color blend units are both responsible for handling all OpenGL ES and programmatic accesses to the tile buffer for functionality like:

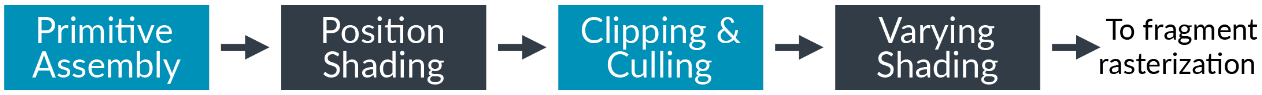
- EXT_shader_pixel_local_storage
- ARM_shader_framebuffer_fetch
- ARM_shader_framebuffer_fetch_depth_stencil
- Merged subpass functionality in Vulkan.

The blender can write either of two fragments per clock to the tile memory. All Mali GPUs are designed to support fast Multi-Sample Anti-Aliasing (MSAA). This means that both full rate fragment blending, and the resolving of pixels when using 4xMSAA, are supported.

# 5. Index-driven geometry pipeline

Valhall supports an Index-Driven Vertex Shading (IDVS) geometry processing pipeline.

The following diagram shows how the IDVS pipeline starts. The pipeline builds primitives and submits shading in primitive order, which splits the shader in half. Next, position shading occurs before clipping and culling, then varying shading runs for all, non-culled, visible vertices:



The IDVS pipeline flow provides the following key optimizations:

- Position shading is only submitted for small batches of vertices where at least one vertex in each batch is referenced by the index buffer. This allows vertex shading to jump spatial gaps in the index buffer which are never referenced.
- Varying shading removes some redundant computation and bandwidth. This is because varying shading is only submitted for primitives that survive the clip-and-cull phase.

Also, partially deinterleaving packed vertex buffers ensure that you maximize the benefit from the IDVS geometry flow. Using two packed buffers, place attributes contributing to position in one, and attributes contributing to non-position varyings in the other. Placing the attributes in this way ensures that non-position varyings are not pulled into the cache for vertices that are culled.

# 6. Check your knowledge

The following questions helps you test yout knowledge:

**How many shader cores are present in a Valhall GPU?**

Arm licenses designs to partners who can then choose how to configure the GPU in their specific chipset, based on their performance needs and silicon area constraints. There is no set number.

**Which kind of rendering does a Mali GPU perform?**

Tile-based rendering

**What is a load/store unit?**

A load/store unit is responsible for all shader memory accesses which are not related to texture samplers.

# 7. Related information

Pre reading

- Introduction to tile-based rendering

Khronos resources

- EXT_shader_pixel_local_storage
- ARM_shader_framebuffer_fetchg
- ARM_shader_framebuffer_fetch_depth_stencil

# 8. Next steps

This guide introduced the fundamental principles of the Valhall Shader Core. If you are interested in previous Mali architecture generations, you can read our guides:

- The Bifrost shader core
- The Midgard shader core
- The Utgard shader core

To learn more about the architecture and the products in each architecture, visit the GPU architecture page.