Arm® Mali™ Offline Compiler

Version 7.3

User Guide

101863_0703_00_en

Arm® Mali™ Offline Compiler

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0700-00	30 October 2019	Non-Confidential	New document for v7.0.
0701-00	28 February 2020	Non-Confidential	New document for v7.1.
0702-00	26 August 2020	Non-Confidential	New document for v7.2.
0703-00	27 November 2020	Non-Confidential	New document for v7.3.

(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.

Web Address

developer.arm.com

Progressive terminology commitment

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

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact *terms@arm.com*.

Contents

Arm® Mali™ Offline Compiler User Guide

Preface

This preface introduces the *Arm® Mali™ Offline Compiler User Guide*.

It contains the following:

About this book

This document explains how to use the Mali Offline Compiler for static analysis of GPU graphics shaders and compute kernels.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
Mali Offline Compiler is a command-line tool that provides static analysis of GPU shaders that are written in OpenGL® ES Shading Language (ESSL), Vulkan SPIR-V intermediate representation, or OpenCL™ C.

Chapter 2 Using Mali Offline Compiler
To query the capabilities of the compiler, or of a specific GPU, and to compile the shader, invoke `malioc` with different command-line options. If compilation is successful, analyze the output performance report.

Chapter 3 Mali GPU pipelines
The internal microarchitecture of the shader core can influence both the register usage and the processing pipelines that are reported in the performance analysis report.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace` (underlined: mono)
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<and>`
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Arm Mali Offline Compiler User Guide*.
- The number 101863_0703_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- *Arm® Developer*.
- *Arm® Documentation*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Introduction

Mali Offline Compiler is a command-line tool that provides static analysis of GPU shaders that are written in OpenGL® ES Shading Language (ESSL), Vulkan SPIR-V intermediate representation, or OpenCL™ C.

It can be used to:

- Validate the syntax of shaders.
- Identify performance bottlenecks.
- Measure the performance impact of any changes.

It contains the following sections:

1.1 API support

Mali Offline Compiler supports the following API versions:

- OpenGL ES 2.0 and 3.0-3.2
- Vulkan 1.0-1.1
- OpenCL 1.0-1.2 and 2.0

OpenCL support is only available on Linux and macOS host installations.

1.2 GPU support

Mali Offline Compiler supports the following Mali™ GPU products:

- Mali-T700 series (Midgard architecture)
- Mali-T800 series (Midgard architecture)
- Mali-G31 (Bifrost architecture)
- Mali-G51 and Mali-G52 (Bifrost architecture)
- Mali-G71, Mali-G72, and Mali-G76 (Bifrost architecture)
- Mali-G57, Mali-G68, Mali-G77, and Mali-G78 (Valhall architecture)

Mali Offline Compiler always targets the latest hardware version and driver version for each supported GPU.

1.3 Binary generation support

Mali Offline Compiler no longer provides the ability to generate binaries for graphics shaders or compute kernels.

Compile and link entire shader programs using the production driver on the target device, and then retrieve the binary using API calls such as `glGetProgramBinary()`. These whole-program binaries are often more efficient than the single shader stage binaries produced by legacy Mali Offline Compiler releases, as extra program-level optimizations can be applied.

———— **Note** ————

Most compiled shader binaries are specific to a single pairing of GPU hardware version and driver version, so reliance on binary-only shader distribution is not recommended.

————————————

Chapter 2
Using Mali Offline Compiler

To query the capabilities of the compiler, or of a specific GPU, and to compile the shader, invoke `malioc` with different command-line options. If compilation is successful, analyze the output performance report.

It contains the following sections:

2.1 Installation

Mali Offline Compiler is installed as part of Arm Mobile Studio.

See *Install Arm Mobile Studio* for instructions on how to download and install this package.

Before using Mali Offline Compiler, we recommend that you add the installation directory to your `PATH` environment variable. Otherwise, you must manually invoke the compiler from the installation directory.

2.2 Querying compiler capabilities

You can query information about the compiler configuration from the command line.

- The `--list` option lists all the valid combinations of supported driver versions, GPUs, and hardware revisions. The listing shows the full capabilities of the compiler, but a specific GPU might not support all the language versions and extensions that the compiler supports.
- The `--info <gpu>` option shows detailed capability information for a specific GPU. For example:

  ```
  malioc --info -c Mali-G72
  ```

  It only shows the language versions and extensions that the GPU supports.

2.3 Compiling OpenGL ES shaders

Use the following command-line syntax to compile OpenGL ES shader programs:

```
malioc -c <target_gpu> [<shader_type>] <file1> [<file2> …] \
[-o <file>]
```

`target_gpu` is one of the GPUs that are listed in *1.2 GPU support* on page 1-10.

`shader_type` is one of the following:

- `--vertex`
- `--tessellation_control`
- `--tessellation_evaluation`
- `--geometry`
- `--fragment`
- `--compute`

You must specify one or more input files that contain the ESSL source code to compile. To read input from stdin, instead of a file on disk, insert a single `-` character. If the input files use one of the following default file extensions, you do not need to explicitly specify the shader type:

`.vert`	OpenGL ES vertex shader.
`.tesc`	OpenGL ES tessellation control shader.
`.tese`	OpenGL ES tessellation evaluation shader.
`.geom`	OpenGL ES geometry shader.
`.frag`	OpenGL ES fragment shader.
`.comp`	OpenGL ES compute shader.

If you specify multiple input files:
- They are concatenated in the order in which they are specified, before compilation
- They must all use the same extension if you do not explicitly specify the shader type

By default, `malioc` emits reports to the `stdout` output stream. You can write directly to a file by specifying the `-o <file>` option. The destination directory must exist; it will not be created.

Use the `-D` option to define a macro on the command line for use in shader source code. For example:

-Dfoo
Defines `foo` with a default value of 1.

-Dfoo=bar
Defines `foo` with the value `bar`.

2.4 Compiling Vulkan shaders

Use the following command-line syntax to compile Vulkan shaders:

```
malioc --vulkan -c <target_gpu> [<shader_type>] [--spirv] [-n <name>] \
<file1> [<file2> …] [-o <file>]
```

`target_gpu` is one of the GPUs that are listed in *1.2 GPU support* on page 1-10.

`shader_type` is one of the following:

- `--vertex`
- `--tessellation_control`
- `--tessellation_evaluation`
- `--geometry`
- `--fragment`
- `--compute`

The input files are either:

- One or more ESSL source shaders.
- A single SPIR-V binary module that has been compiled using Vulkan semantics.

To read input from `stdin`, instead of a file on disk, insert a single `-` character. You do not need to explicitly specify the source shader type if the input files use one of the supported file extensions:

`.vert`	OpenGL ES vertex shader.
`.tesc`	OpenGL ES tessellation control shader.
`.tese`	OpenGL ES tessellation evaluation shader.
`.geom`	OpenGL ES geometry shader.
`.frag`	OpenGL ES fragment shader.
`.comp`	OpenGL ES compute shader.

Note

For binary modules containing a single shader stage, `malioc` automatically detects that they are SPIR-V binary modules, and attempts to deduce the shader type and entry point name. For target binary modules containing multiple entry points, you must specify them manually. You can provide shader type information either by using an auto-detected file extension, or a manually specified shader type flag. The supported file extensions are appended with `.spv`, for example `.vert.spv`. You can force interpretation of a file as SPIR-V by passing in the `--spirv` option.

If you specify multiple input files:
- They are concatenated in the order in which they are specified, before compilation.
- If you do not explicitly specify the shader type, they must all use the same extension.

If you pass an ESSL source file, it is automatically converted into a SPIR-V binary module using the version of glslang that is provided in the installation. The resulting SPIR-V module is passed to the Mali Offline Compiler backend.

Use the `-n <name>` option to specify a custom SPIR-V entry point for binary module inputs. The default entry point is called `main`.

By default, `malioc` emits reports to the stdout output stream. You can write directly to a file by specifying the `-o <file>` option. The destination directory must exist; it will not be created.

Use the `-D` option to define a macro on the command line for use in shader source code. For example:

`-Dfoo`

Defines `foo` with a default value of 1.

`-Dfoo=bar`

Defines `foo` with the value `bar`.

2.5 Compiling OpenCL C kernels

Use the following command-line syntax to compile OpenCL C kernels:

```
malioc -c <target_gpu> [--opencl <version>] [--kernel] [-n <name>] \
<file1> [<file2> …] [-o <file>]
```

`target_gpu` is one of the GPUs that are listed in *1.2 GPU support* on page 1-10.

Use the `--opencl` option to specify the version of OpenCL to be targeted, either:

1.x	Targets the latest minor version of OpenCL 1 supported by the target GPU.
2.x	Targets the latest minor version of OpenCL 2 supported by the target GPU.

To read input from stdin, instead of a file on disk, insert a single `-` character. If the input filename has a `.cl` extension, which is the default for an OpenCL kernel, you do not need to explicitly specify the type as `--kernel`.

Use the `-n <name>` option to specify the entry point of the kernel to be compiled.

If you specify multiple input files:
- They are concatenated in the order in which they are specified, before compilation
- They must all have a `.cl` extension if you do not explicitly specify `--kernel`

By default, `malioc` emits reports to the stdout output stream. You can write directly to a file by specifying the `-o <file>` option. The destination directory must exist; it will not be created.

Use the `-D` option to define a macro on the command line for use in kernel source code. For example:

-Dfoo

Defines `foo` with a default value of 1.

-Dfoo=bar

Defines `foo` with the value `bar`.

This section contains the following subsection:
- *2.5.1 Header includes* on page 2-18.

2.5.1 Header includes

The OpenCL C language allows you to use header files in your source code, with the `#include` preprocessor directive.

Relative path header inclusions use the current working directory as the root of the search path:

```
#include "my_header.h"
```

You can also use absolute path header inclusions:

```
#include "/work/my_header.h"
```

2.6 Syntax error reporting

If Mali Offline Compiler fails to compile a shader program due to an error in the code, it produces a compilation error and emits an error message to the console.

Error messages only give a line number, which is the line number after all input source files have been concatenated.

2.7 Performance analysis

If compilation is successful, Mali Offline Compiler emits a static analysis report outlining the shader performance on the target GPU.

For example:

```
Configuration
=============

Hardware: Mali-T880 r2p0
Driver: Midgard r23p0-00rel0
Shader type: OpenGL ES Fragment

Main shader
===========

Work registers: 2
Uniform registers: 2
Stack spilling: false

                              A    LS    T   Bound
Total Instruction Cycles:   6.0  1.0  0.0      A
Shortest Path Cycles:       1.7  1.0  0.0      A
Longest Path Cycles:        1.7  1.0  0.0      A

A = Arithmetic, LS = Load/Store, T = Texture

Shader properties
=================

Has uniform computation: true
```

This section contains the following subsections:

- *2.7.1 IDVS shader variants* on page 2-20.
- *2.7.2 Register usage* on page 2-21.
- *2.7.3 Performance table* on page 2-21.
- *2.7.4 Shader properties* on page 2-21.

2.7.1 IDVS shader variants

On Mali GPUs in the Bifrost and Valhall families, vertex shaders are executed using an optimized shading flow called Index-Driven Vertex Shading (IDVS).

In the IDVS pipeline, vertex shaders are compiled into two binaries:

- A position shader, which computes only position.
- A varying shader, which computes the remaining non-position vertex attribute outputs.

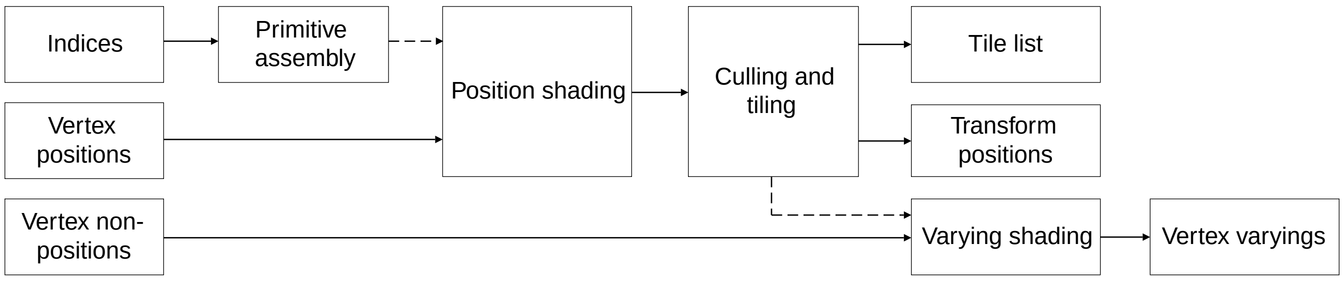

Figure 2-1 IDVS pipeline

The position shader is executed for every index vertex, but the varying shader is only executed for vertices that are part of a visible primitive that survives culling. Mali Offline Compiler reports separate performance tables for each of these variants.

2.7.2 Register usage

The report outlines the register usage by the shader program, including whether it is spilling to stack memory.

Work register usage can impact the number of threads that the shader core can execute simultaneously. This impact is because the available physical register pool is divided among the shader threads that are executing. Reducing the work register usage per thread can increase the number of threads that can be executed, which is often beneficial. See *Chapter 3 Mali GPU pipelines* on page 3-25 for more details about work register usage for each Mali GPU architecture.

Shaders that spill to stack are expensive for a GPU to process, so try to eliminate spilling from shaders by reducing register pressure. You can reduce register pressure in the following ways:
- By reducing variable precision.
- By reducing the live ranges of variables.
- By simplifying the shader program.

2.7.3 Performance table

The performance table gives an indication of the potential performance of the shader program for a single shader core.

It contains the following rows:

`Total Instruction Cycles`
The cumulative number of execution cycles for all instructions that are generated for the program, irrespective of program control flow.

`Shortest Path Cycles`
An estimate of the number of cycles for the shortest control flow path though the shader program. This row normalizes the cycle cost based on the number of functional units present in the design.

`Longest Path Cycles`
An estimate of the number of cycles for the longest control flow path though the shader program. This row normalizes the cycle cost based on the number of functional units present in the design. It is not always possible to determine the longest path based on static analysis, for example if a uniform variable controls a loop iteration limit. So this row might indicate an unknown cycle count ("N/A").

The reported statistics are broken down by functional unit. The unit with the highest cycle cost in either or both of the `Shortest Path Cycles` and `Longest Path Cycles` rows is a good candidate to optimize. For example, a shader whose highest values are in the `A` (Arithmetic) column, is arithmetic bound. It would be best to optimize it by reducing the number of, or the precision of, the mathematical operations that it performs.

The functional unit columns that are displayed depend on the architecture of the GPU being targeted. See *Chapter 3 Mali GPU pipelines* on page 3-25 for more details. In addition, there are some important considerations to be aware of when reviewing the performance data. See *2.8 Performance considerations* on page 2-23 for more details.

2.7.4 Shader properties

The `Shader properties` section provides information about behavioral properties of the shader program.

It can contain the following entries:

`Has uniform computation`

Shows if there was any optimized uniform computation. This is computation that depends only on literal constants or uniform values, and therefore produces the same result for every thread in a draw call or compute dispatch. While the drivers can optimize this, it still has a cost, so where possible, move it from the shader into application logic on the CPU.

`Has side-effects`

Shows if this shader has side-effects that are visible in memory, outside of the fixed graphics pipeline. They can be caused by:

- Writes into shader storage buffers
- Stores into images
- Uses of atomics

Side-effecting shaders cannot be optimized away by techniques such as hidden surface removal, so their use should be minimized.

`Modifies coverage`

Shows if a fragment shader has a coverage mask that can be changed by shader execution, for example by using a `discard` statement. Shaders with modifiable coverage must use a late-ZS update, which reduces efficiency of early ZS testing for later fragments at the same coordinate.

Note

Other API-side behaviors, such as setting of alpha-to-coverage, can also impact coverage masks and are not considered here.

`Uses late ZS test`

Shows if a fragment shader contains logic that forces a late ZS test, for example by writing to `gl_FragDepth`. This disables use of early-ZS testing and hidden surface removal, which can be a significant efficiency loss.

Note

Other API-side behaviors, such as disabling depth testing, can override this behavior.

`Uses late ZS update`

Shows if a fragment shader contains logic that forces a late ZS update, for example by reading the old depth value in the shader by using `gl_LastFragDepthARM`. This can reduce efficiency of early ZS testing for later fragments at the same coordinate.

`Reads color buffer`

Shows if a fragment shader contains logic that programmatically reads from the color buffer, for example by reading from `gl_LastFragColorARM`. Shaders that read from the color buffer in this manner are treated as transparent, and cannot be used as hidden-surface removal occluders.

2.8 Performance considerations

There are several important considerations to be aware of when analyzing the data in the performance table:

- The cycle measurements are purely based on the execution cost of the instructions in the program. The actual performance is also dependent on inputs that are not visible in the instruction sequence, such as texture sampler configuration and texture format.

  For example, using trilinear filtering for all texture samples halves the filtering rate. Therefore it would double the texture cycle count compared to the value that is reported in the `T` (Texture) column in the performance table.
- The shortest and longest control flow measurements are based on what is possible in the shader source code. They are not based on the real run-time inputs, such as uniform values, that are used for a specific draw call. These costings therefore define the flight-envelope of performance possibilities but are not accurate for any single specific use of the shader.
- Mali Offline Compiler only processes single shaders at a time. The on-device Mali driver compilation process optimizes whole programs and pipelines, including use of pipeline state information in the case of Vulkan. This optimization can result in the reported performance being different to the performance that would be seen in a production device, although it should be indicative.

——— **Note** ———

You can directly measure pipeline activity on the target platform using the Streamline profiling tools. Profiling with Streamline can provide a useful comparison with the static analysis that Mali Offline Compiler provides.

———————

2.9 Generating JSON reports

By default, Mali Offline Compiler generates reports in a human readable text format. To allow easier integration into other tooling or scripted workflows, it also supports generating machine-readable JSON reports. These reports are enabled by adding the `--format json` command-line option to any of the operations.

There are four types of JSON output report that Mali Offline Compiler can generate, identified by a schema identifier field in the root JSON object:

`list`
For `--list` operations.

`info`
For `--info` operations.

`error`
For compile operations that fail with a compilation error.

`performance`
For compile operations that succeed.

To aid writing parsers, sample reports and *JSON Schema* definitions are provided for all four of the supported output reports. These files are in `<install_directory>/samples/json_reports` and `<install_directory>/samples/json_schemas` respectively.

To help with JSON parsing, the command line utility can return three possible process return codes:

0 The operation was successful and will return a `list`, `info`, or `performance` (compilation) JSON report.

1 Compilation failed due to a shader syntax error. This will return an `error` JSON report.

2 The tool failed due to a configuration error, such as a bad command line option. This will always emit human-readable text output, not a JSON report.

Chapter 3
Mali GPU pipelines

The internal microarchitecture of the shader core can influence both the register usage and the processing pipelines that are reported in the performance analysis report.

Correct identification of the shader pipeline with the highest load is critical in performance analysis. Optimizing that pipeline is more likely to give a performance benefit. This section provides a brief summary of the register thresholds and processing pipelines for each supported Mali GPU architecture.

It contains the following sections:

3.1 Mali Midgard architecture

Mali Midgard GPU shader cores have three parallel pipeline classes:

Arithmetic unit (A)
The arithmetic pipeline executes all types of shader arithmetic instructions. There can be multiple parallel arithmetic pipelines, the number present depends on the Mali GPU being targeted. Data presented in the tool is normalized based on the number of pipelines in the design.

Load/store unit (LS)
The load/store pipeline handles all non-texture memory access, including buffer access, image access, and atomic operations. In addition, this pipeline implements the Midgard varying interpolator.

Texture unit (T)
The texture pipeline handles all texture sampling and filtering operations.

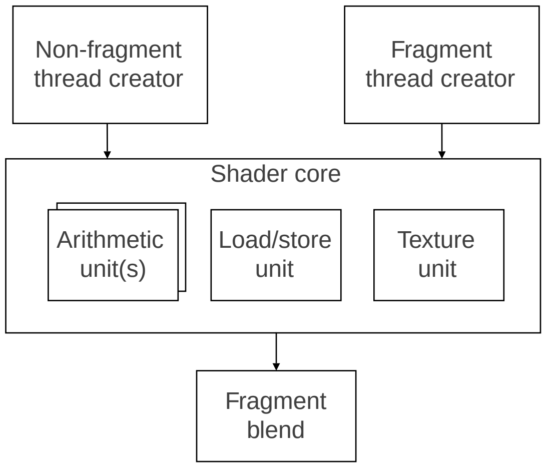

Figure 3-1 Midgard shader core

This section contains the following subsection:
- *3.1.1 Midgard work register breakpoints* on page 3-26.

3.1.1 Midgard work register breakpoints

Mali Midgard GPU shader cores allow variable numbers of threads to be created, depending on the number of work registers that are used by the in-flight shader programs.

0-4 registers
Maximum thread capacity

5-8 registers
Half thread capacity

8-16 registers
Quarter thread capacity

Usually, running more threads simultaneously helps a GPU to keep busy. A good objective is to stay at 0-4 registers for fragment shaders and 0-8 threads for other shader types.

The most effective way to reduce register pressure is to minimize the precision of stored variables. Use `mediump` precision in preference to `highp` whenever possible.

3.2 Mali Bifrost architecture

Mali Bifrost GPU shader cores have four parallel pipeline classes:

Arithmetic unit (A)
The arithmetic pipeline, also known as the execution engine, executes all types of shader instructions. There can be multiple parallel arithmetic pipelines, the number present depends on the Mali GPU being targeted. To give an overall cost for the targeted shader core, data presented in the tool is normalized based on the number of engines in the design.

Load/store unit (LS)
The load/store pipeline handles all non-texture memory access, including buffer access, image access, and atomic operations.

Varying unit (V)
The varying pipeline is a dedicated pipeline which implements the varying interpolator.

Texture unit (T)
The texture pipeline handles all texture sampling and filtering operations.

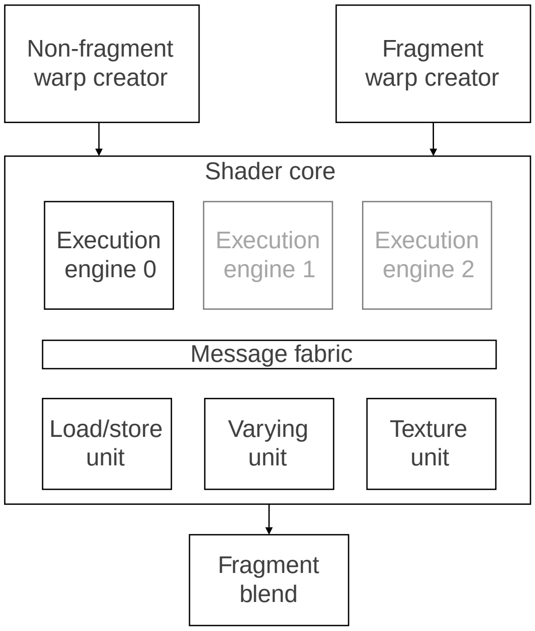

Figure 3-2 Bifrost shader core

This section contains the following subsections:
- *3.2.1 Bifrost work register breakpoints* on page 3-27.
- *3.2.2 Shader core size* on page 3-28.

3.2.1 Bifrost work register breakpoints

Mali Bifrost GPU shader cores allow variable numbers of threads to be created, depending on the number of work registers that are used by the in-flight shader programs.

0-32 registers
Maximum thread capacity

33-64 registers
Half thread capacity

Usually, running more threads simultaneously helps a GPU to keep busy. A good objective is to stay at 0-32 registers for fragment shaders.

The most effective way to reduce register pressure is to minimize the precision of stored variables. Use `mediump` precision in preference to `highp` whenever possible.

3.2.2 Shader core size

The early-generation Bifrost shader cores, Mali-G71 and Mali-G72, implement a single texel-per-clock and single pixel-per-clock shader core. Later shader cores in the Bifrost family implement a two texel-per-clock and two pixel-per-clock shader core, with an increase in arithmetic performance to compensate. Not every GPU doubled the available performance though.

Mali Offline Compiler reports results per shader core. It is expected, for example, that performance results for a Mali-G76 have approximately half the cycle count of the results for a Mali-G72. Silicon implementations using a Mali-G76 generally implement fewer shader cores than an equivalent Mali-G72 design. Remember therefore that the results must be scaled by the shader core count in your target device.

3.3 Mali Valhall architecture

Mali Valhall GPU shader cores have six parallel pipeline classes, comprising three arithmetic pipelines and three fixed-function support pipelines.

All Valhall GPUs implement two parallel processing engines, each containing their own set of arithmetic pipelines. Data presented in the tool is normalized based on the number of engines in the design, to give an overall cost for the targeted shader core, not just for a single engine.

Arithmetic fused multiply accumulate unit (FMA)

The FMA pipelines are the main arithmetic pipelines, implementing the floating-point multipliers that are widely used in shader code. Each FMA pipeline implements a 16-wide warp, and can issue a single 32-bit operation or two 16-bit operations per thread and per clock cycle.

Most programs that are arithmetic-limited are limited by the performance of the FMA pipeline.

Arithmetic convert unit (CVT)

The CVT pipelines implement simple operations, such as format conversion and integer addition. Each CVT pipeline implements a 16-wide warp, and can issue a single 32-bit operation or two 16-bit operations per thread and per clock cycle.

Arithmetic special functions unit (SFU)

The SFU pipelines implement a special functions unit for computation of complex functions such as reciprocals and transcendental functions. Each SFU pipeline implements a 4-wide issue path, executing a 16-wide warp over 4 clock cycles.

Load/store unit (LS)

The load/store pipeline handles all non-texture memory access, including buffer access, image access, and atomic operations.

Varying unit (V)

The varying pipeline is a dedicated pipeline which implements the varying interpolator.

Texture unit (T)

The texture pipeline handles all texture sampling and filtering operations.

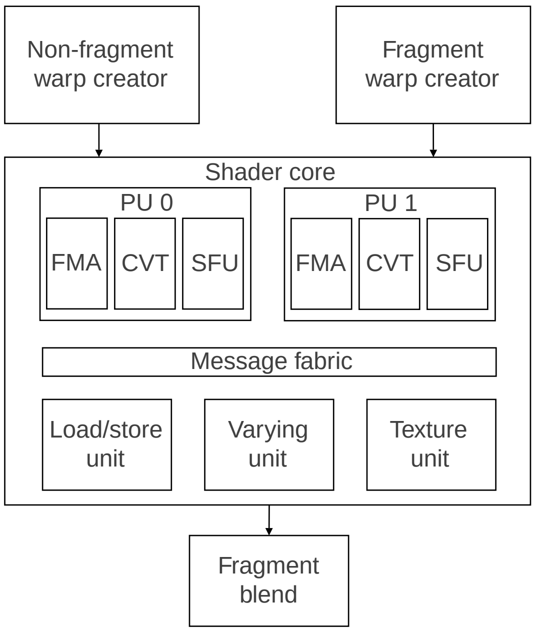

Figure 3-3 Valhall shader core

This section contains the following subsections:
- *3.3.1 Valhall work register breakpoints* on page 3-30.
- *3.3.2 Shader core size* on page 3-30.

3.3.1 Valhall work register breakpoints

Mali Valhall GPU shader cores allow variable numbers of threads to be created, depending on the number of work registers that are used by the in-flight shader programs.

0-32 registers
Maximum thread capacity

33-64 registers
Half thread capacity

Usually, running more threads simultaneously helps a GPU to keep busy. A good objective is to stay at 0-32 registers for fragment shaders.

The most effective way to reduce register pressure is to minimize the precision of stored variables. Use `mediump` precision in preference to `highp` whenever possible.

3.3.2 Shader core size

All Valhall GPU cores implement a four texel-per-clock and two pixel-per-clock shader core.