



Arm[®] Mali[™] Offline Compiler

Version 7.8

User Guide

Non-Confidential

Copyright © 2019–2022 Arm Limited (or its affiliates).
All rights reserved.

Issue 00

101863_0708_00_en



Arm® Mali™ Offline Compiler

User Guide

Copyright © 2019–2022 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0700-00	30 October 2019	Non-Confidential	Document release for Mali Offline Compiler version 7.0.
0701-00	28 February 2020	Non-Confidential	Update for Mali Offline Compiler version 7.1.
0702-00	26 August 2020	Non-Confidential	Update for Mali Offline Compiler version 7.2.
0703-00	27 November 2020	Non-Confidential	Update for Mali Offline Compiler version 7.3.
0704-00	26 August 2021	Non-Confidential	Update for Mali Offline Compiler version 7.4.
0705-00	22 February 2022	Non-Confidential	Update for Mali Offline Compiler version 7.5.
0706-00	26 May 2022	Non-Confidential	Update for Mali Offline Compiler version 7.6.
0707-00	26 August 2022	Non-Confidential	Update for Mali Offline Compiler version 7.7.
0708-00	25 November 2022	Non-Confidential	Update for Mali Offline Compiler version 7.8.

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2019–2022 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

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

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

Inclusive language commitment

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

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

Contents

1. Introduction.....	6
1.1 Conventions.....	6
1.2 Useful resources.....	7
1.3 Other information.....	7
2. Platform support.....	8
2.1 API support.....	8
2.2 GPU support.....	8
2.3 Binary generation support.....	9
3. Using Mali Offline Compiler.....	10
3.1 Install Mali Offline Compiler.....	10
3.2 Querying compiler capabilities.....	11
3.3 Compiling OpenGL ES shaders.....	11
3.4 Compiling Vulkan shaders.....	12
3.5 Compiling OpenCL C kernels.....	14
3.6 Syntax error reporting.....	15
3.7 Performance analysis.....	16
3.8 Performance considerations.....	16
3.9 Generating JSON reports.....	17
4. Mali GPU pipelines.....	18
4.1 Mali Midgard architecture.....	18
4.2 Mali Bifrost architecture.....	19
4.3 Mali Valhall architecture.....	20

1. Introduction

Describes how to install, get started, and use Mali™ Offline Compiler to capture performance measurements from your Mali and Immortalis devices.

1.1 Conventions

The following subsections describe conventions used in Arm documents.




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: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
<i>italic</i>	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
monospace	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
monospace <u>underline</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: <pre>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></pre>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the <i>Arm® Glossary</i> . For example, IMPLEMENTATION DEFINED , IMPLEMENTATION SPECIFIC , UNKNOWN , and UNPREDICTABLE .
 Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
 Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
 Danger	Requirements for the system. Not following these requirements will result in system failure or damage.

Convention	Use
 Note	An important piece of information that needs your attention.
 Tip	A useful tip that might make it easier, better or faster to perform a task.
 Remember	A reminder of something important that relates to the information you are reading.

1.2 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Non-Arm resources	Document ID	Organization
Annotate and validate JSON documents	-	JSON Schema

1.3 Other information

See the Arm website for other relevant information.

- [Arm® Developer](#).
- [Arm® Documentation](#).
- [Technical Support](#).
- [Arm® Glossary](#).

2. Platform support

Arm® 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.

Mali Offline Compiler can be used to:

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

2.1 API support

Arm® Mali™ Offline Compiler supports compiling shaders for the OpenGL ES and Vulkan graphics APIs, and compiling kernels for the OpenCL compute API.

The following API versions are supported, subject to support being available for the targeted GPU core:

- OpenGL ES 2.0 and 3.0-3.2
- Vulkan 1.0-1.3
- OpenCL 1.0-1.2, 2.0, and 3.0

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

2.2 GPU support

Arm® Mali™ Offline Compiler supports the following Arm Immortalis™ and Arm Mali GPU products:

Valhall architecture

- Immortalis-G715 (OpenGL ES, Vulkan, OpenCL)
- Mali-G715 (OpenGL ES, Vulkan, OpenCL)
- Mali-G710 (OpenGL ES, Vulkan, OpenCL)
- Mali-G615 (OpenGL ES, Vulkan, OpenCL)
- Mali-G610 (OpenGL ES, Vulkan, OpenCL)
- Mali-G510 (OpenGL ES, Vulkan, OpenCL)
- Mali-G310 (OpenGL ES, Vulkan, OpenCL)
- Mali-G78AE (OpenGL ES, Vulkan, OpenCL)
- Mali-G78 (OpenGL ES, Vulkan, OpenCL)

- Mali-G77 (OpenGL ES, Vulkan, OpenCL)
- Mali-G68 (OpenGL ES, Vulkan, OpenCL)
- Mali-G57 (OpenGL ES, Vulkan, OpenCL)

Bifrost architecture

- Mali-G76 (OpenGL ES, Vulkan, OpenCL)
- Mali-G72 (OpenGL ES, Vulkan, OpenCL)
- Mali-G71 (OpenGL ES, Vulkan, OpenCL)
- Mali-G52 (OpenGL ES, Vulkan, OpenCL)
- Mali-G51 (OpenGL ES, Vulkan, OpenCL)
- Mali-G31 (OpenGL ES, Vulkan, OpenCL)

Midgard architecture

- Mali-T880 (OpenGL ES, Vulkan, OpenCL)
- Mali-T860 (OpenGL ES, Vulkan, OpenCL)
- Mali-T830 (OpenGL ES, Vulkan, OpenCL)
- Mali-T820 (OpenGL ES, Vulkan, OpenCL)
- Mali-T760 (OpenGL ES, Vulkan, OpenCL)
- Mali-T720 (OpenGL ES, OpenCL)

Mali Offline Compiler targets the following driver versions for the supported GPUs:

- Bifrost and Valhall architecture GPUs use r41p0
- Midgard architecture GPUs use r23p0

2.3 Binary generation support

Arm® 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.



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.

3. 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.

3.1 Install Mali Offline Compiler

This topic describes how to install Arm® Mali™ Offline Compiler as part of Arm Mobile Studio.

About this task

If you have already installed Arm Mobile Studio, you do not need to do anything further to install Mali Offline Compiler.

Before you begin

1. Log in to your Arm Account. If you don't have one, register at [Downloads](#).
2. Download the Arm Mobile Studio install package for your platform.

Procedure

- Install:
 - On 64-bit Windows:

Arm Mobile Studio is provided with an installer executable. Double-click the `.exe` file and follow the instructions in the **Setup Wizard**.
 - On macOS:

Arm Mobile Studio is provided as a dmg package. To mount the package, double-click the dmg package and follow the instructions. For easy access, the directory tree copies to the **Applications** folder on your local file system.
 - On Linux:

Arm Mobile Studio is provided as a gzipped tar archive. Use a recent version (1.13 or later) of GNU tar to extract the tar archive to your preferred location:

```
tar xvzf Arm_Mobile_Studio_<version>_linux.tgz
```

1. Add the path to the installation directory to your `PATH` environment variable.

If you do not update your `PATH`, you must manually invoke the compiler from the installation directory.

Next steps

Check your compiler configuration, see [Querying compiler capabilities](#).

3.2 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.

3.3 Compiling OpenGL ES shaders

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

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

`target_gpu` is one of the GPUs that are listed in [GPU support](#). If `target_gpu` is not specified the latest supported GPU is used.

`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 because it is not 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`.

3.4 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 [GPU support](#). If `target_gpu` is not specified the latest supported GPU is used.

`shader_type` is one of the following:

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

The input files are either:

- One or more GLSL, or 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 or OpenGL ES syntax vertex shader.

.tesc

OpenGL or OpenGL ES syntax tessellation control shader.

.tese

OpenGL or OpenGL ES syntax tessellation evaluation shader.

.geom

OpenGL or OpenGL ES syntax geometry shader.

.frag

OpenGL or OpenGL ES syntax fragment shader.

.comp

OpenGL or OpenGL ES syntax compute shader.

.rgen

OpenGL or OpenGL ES syntax ray generation shader.

.rahit

OpenGL or OpenGL ES syntax ray any hit shader.

.rchit

OpenGL or OpenGL ES syntax ray closest hit shader.

.rint

OpenGL or OpenGL ES syntax ray intersection shader.

.rmiss

OpenGL or OpenGL ES syntax ray miss shader.

.rcall

OpenGL or OpenGL ES syntax ray callable shader.



For binary modules containing a single shader stage, `maliloc` 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 the desired entry point 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 Arm® Mali™ Offline Compiler backend.

Use the `-n <name>` option to specify a custom SPIR-V entry point for binary module inputs. You do not need to specify `-n <name>` for SPIR-V modules which contain only a single entry point because the entry point will be automatically detected.

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 because it is not 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`.

3.5 Compiling OpenCL C kernels

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

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

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

1.1

Targets OpenCL 1.1.

1.2

Targets OpenCL 1.2.

2.0

Targets OpenCL 2.0.

3.0

Targets OpenCL 3.0.

If you do not explicitly specify `--opencl` the compiler defaults to targeting OpenCL 1.2. OpenCL 3.0 is required to support SPIR-V binary modules.

`target_gpu` is one of the GPUs that are listed in [GPU support](#). If `target_gpu` is not specified the latest supported GPU is used.

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:

.cl

OpenCL C compute kernel.

cl.spv

OpenCL SPIR-V binary compute kernel.



For binary inputs `maliloc` 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 the desired entry point manually. You can provide shader type information either by using an auto-detected file extension, or a manually specified shader type flag. You can force interpretation of a file as SPIR-V by using the `--spirv` option.

Use the `-n <name>` option to specify the entry point of the kernel to be compiled. You do not need to specify `-n <name>` for SPIR-V modules which contain only a single entry point because the entry point will be automatically detected.

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 `--kernel1`.

By default, `maliloc` 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 because it is not 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`.

3.6 Syntax error reporting

If Arm® 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.

3.7 Performance analysis

If compilation is successful, Arm® 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 (100% occupancy)
Uniform registers: 2
Stack spilling: false

Total Instruction Cycles:      A   LS   T   Bound
Shortest Path Cycles:         6.0  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
```

3.8 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.
- Arm® Mali™ Offline Compiler only processes single shaders at a time. The on-device 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.



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

3.9 Generating JSON reports

By default, Arm® 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 returns a `list`, `info`, or `performance` (compilation) JSON report.

1

Compilation failed because of a shader syntax error. This utility returns an `error` JSON report.

2

The tool failed because of a configuration error, such as a bad command line option. This utility always emits human-readable text output, not a JSON report.

4. 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 Arm® Mali™ GPU architecture.

4.1 Mali Midgard architecture

Arm® 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.

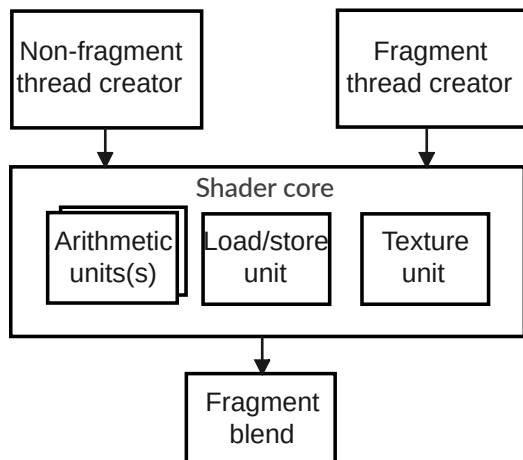
Load/store unit (LS)

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 4-1: Midgard shader core



4.2 Mali Bifrost architecture

Arm® 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.

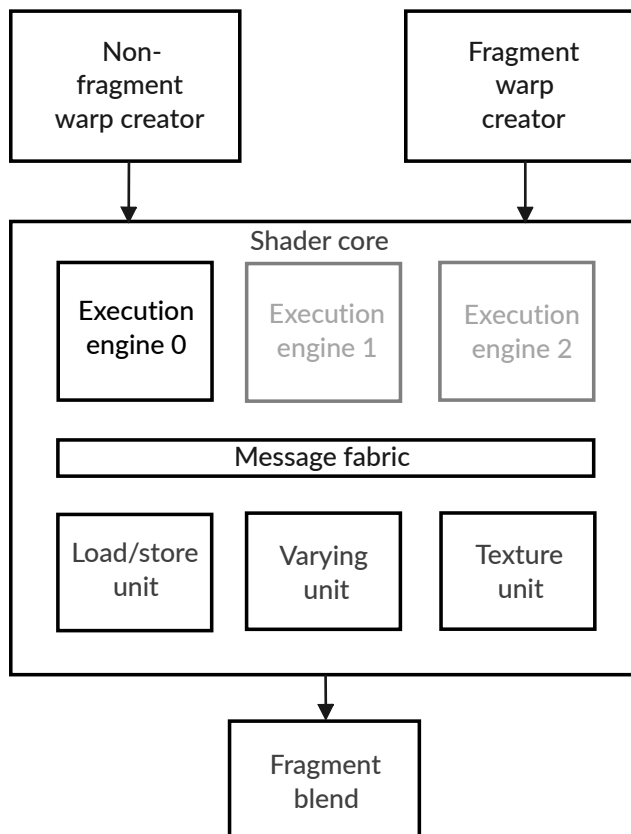
Varying unit (V)

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 4-2: Bifrost shader core



4.3 Mali Valhall architecture

Arm® 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.

For Mali Valhall GPUs the text performance report shows a single combined arithmetic cycle cost (A). This cycle cost is estimated for the target GPU based on the identified FMA, CVT, and SFU workload. To enable the full report, and show the individual arithmetic pipelines, use the `--detailed` command line option.

Figure 4-3: Valhall shader core