



Arm[®] Mali[™] Offline Compiler

Version 8.1

User Guide

Non-Confidential

Copyright © 2019–2023 Arm Limited (or its affiliates). All rights reserved.

Issue 00

101863_0801_00_en



Arm® Mali™ Offline Compiler

User Guide

Copyright © 2019–2023 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.
0708-01	1 December 2022	Non-Confidential	Documentation update 1 for Mali Offline Compiler version 7.8.
0709-00	15 April 2023	Non-Confidential	Update for Mali Offline Compiler version 7.9.
0800-00	11 June 2023	Non-Confidential	Update for Mali Offline Compiler version 8.0.
0801-00	5 August 2023	Non-Confidential	Update for Mali Offline Compiler version 8.1.

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 © 2019–2023 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.....	7
1.1 Conventions.....	7
1.2 Useful resources.....	8
1.3 Other information.....	8
2. Platform support.....	9
2.1 API support.....	9
2.2 GPU support.....	9
2.3 Binary generation support.....	10
3. Using Mali Offline Compiler.....	12
3.1 Install Mali Offline Compiler.....	12
3.2 Querying compiler capabilities.....	13
3.3 Compiling OpenGL ES shaders.....	13
3.4 Compiling Vulkan shaders.....	14
3.5 Compiling OpenCL kernels.....	16
3.5.1 Header includes.....	17
3.6 Syntax error reporting.....	18
3.7 Performance analysis.....	18
3.7.1 IDVS shader variants.....	18
3.7.2 Resource usage.....	19
3.7.3 Performance table.....	20
3.7.4 Shader properties.....	21
3.7.5 Recommended vertex attribute streams.....	23
3.8 Performance considerations.....	24
3.9 Generating JSON reports.....	24
4. Mali GPU pipelines.....	26
4.1 Mali Midgard architecture.....	26
4.1.1 Midgard work register breakpoints.....	27
4.2 Mali Bifrost architecture.....	27
4.2.1 Bifrost work register breakpoints.....	28
4.2.2 Bifrost shader core size.....	29

4.3 Mali Valhall and 5th Generation architectures.....	29
4.3.1 Valhall and 5th Generation architecture work register breakpoints.....	30
4.3.2 Valhall and 5th Generation architecture shader core size.....	31

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 graphics shaders that are written in OpenGL ES Shading Language (ESSL) or Vulkan SPIR-V intermediate representation. Compute kernels that are written in OpenCL C or OpenCL SPIR-V intermediate representation are also supported.

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:

Arm 5th Generation architecture

- Immortalis-G720 (OpenGL ES, Vulkan, OpenCL)
- Mali-G720 (OpenGL ES, Vulkan, OpenCL)
- Mali-G620 (OpenGL ES, Vulkan, OpenCL)

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, Valhall, and Arm 5th Generation GPUs use r44p0
- 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

1. 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
```

2. 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 kernels

Use the following command-line syntax to compile OpenCL 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 `--kernel`.

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`.

3.5.1 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"
```

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:  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
```

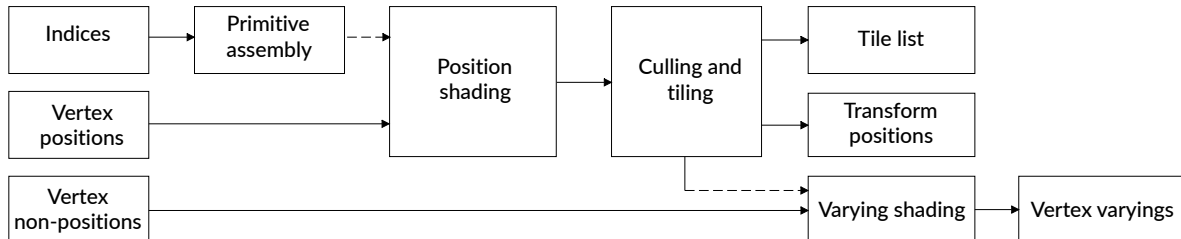
3.7.1 IDVS shader variants

On Arm® 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 3-1: 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.

3.7.2 Resource usage

The resource usage section of the report shows how resources are managed by the shader program. You can see the usage of registers, stack memory, shared memory, ray traversal, and the 16-bit data path in the arithmetic unit.

Work register

Demand on the work register 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 [Mali GPU pipelines](#) for more details about work register usage for each Arm® Mali™ GPU architecture.

Uniform registers

Uniform registers are used as read-only storage for uniform and literal constant values. Programs that run out of uniform storage, as indicated by a '100% used' metric in the report, need to fall back to per-thread memory loads for constant values. To reduce uniform register pressure, use 16-bit data types, or reduce the number of uniforms and constants in your shaders.

Shared storage

Shared storage allows threads in a single compute work group to share data. Mali GPUs use cached system RAM to back shared memory, so shared memory has the same performance as any other buffer access. Use shared storage only where you need algorithmic data sharing across threads.

Stack spilling

Shaders that spill to stack are expensive for a GPU to process. Reduce register pressure to help stop the shaders from spilling. You can reduce register pressure in one of the following ways:

- By reducing variable precision.
- By reducing the live ranges of variables.
- By simplifying the shader program.

16-bit arithmetic

16-bit arithmetic is more energy efficient, and higher performance, than 32-bit arithmetic. For most operations, Mali can either submit a vec2 SIMD 16-bit operation or a scalar fp32 operation, so in the best case using 16-bit operations are twice as fast. Even in cases where overall performance does not increase, a higher percentage of 16-bit operations improves energy efficiency.

3.7.3 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 through 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 through 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 column 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. Optimize the shader by reducing the number of, or the precision of, the mathematical operations that it performs. The **Bound** column lists the functional units with the highest cycle count, which allows you to quickly identify the units that are a bottleneck in your shader code.

The functional unit columns that are displayed depend on the architecture of the GPU being targeted. See [Mali GPU pipelines](#) for more details. In addition, there are some important

considerations to be aware of when reviewing the performance data. See [Performance considerations](#) for more details.

3.7.4 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.

Has slow ray traversal

Shows if the shader is using at least one ray traversal which forces the compiler to fallback to a slower traversal behavior.

To avoid the slow traversal behavior for ray query usage, Arm recommends having a single `rayQueryProceed()` per `rayQueryInitialize()`. Both `rayQueryProceed()` and `rayQueryInitialize()` must be called unconditionally. The following examples demonstrate possible causes of a slower traversal:

```
// Slow due to divergent initialization
if (cond)
    rayQueryInitialize(rq, params_1);
else
    rayQueryInitialize(rq, params_2);
```

```
// Slow due to multiple proceeds for a single initialize
rayQueryInitialize(rq, params);
if (cond)
    rayQueryProceed(rq);
rayQueryProceed(rq);
```

```
// Slow due to multiple proceeds for a single initialize
rayQueryInitialize(rq, params);
rayQueryProceed(rq);
```

```
rayQueryProceed(rq);
```

For cases where multiple proceeds are required, we recommend placing a single non-conditional `rayQueryProceed()` inside a while loop.

```
// Fast due to single initialize and single proceed call site
rayQueryInitialize(rq, params);
while (cond) {
    rayQueryProceed(rq);
}
```

For cases where a conditional proceed is required, we recommend placing the `rayQueryInitialize()` inside the same conditional block as the `rayQueryProceed()`.

```
// Fast due to single initialize and proceed under the same condition
if (cond) {
    rayQuery rq;
    rayQueryInitialize(rq, params);
    rayQueryProceed(rq);
}
```

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.

3.7.5 Recommended vertex attribute streams

To achieve optimal performance and memory bandwidth, you must tune the memory layout of your application for vertex attributes to match the phased processing approach Arm® Mali™ GPUs use for vertex shading.

The first processing phase computes only the vertex position, which is needed for primitive culling. The second phase computes any non-position outputs, and is only executed for vertices that contribute to a visible primitive.

Arm recommends storing all position-related input attributes interleaved in one memory range, and all non-position input attributes interleaved in a separate memory range. Storing attribute data interleaved but in separate memory ranges, ensures that the position shading phase only loads useful position-related data from DRAM, and minimizes cache pollution caused by fetching unnecessary non-position data. The non-position data is only fetched during phase two, and therefore only for the vertices that survive culling.

Vertex shader reports contain a **Recommended attribute stream** section which defines the mapping of attributes to in-memory streams that you must use to get the optimal geometry memory bandwidth.



The **Recommended attribute stream** report section is not supported on Midgard family GPUs.

```
Recommended attribute streams
=====

Position attributes
- inPos (location=dynamic)

Non-position attributes
- inTexCoord (location=1)
```

For OpenGL ES, each attribute entry contains the symbol name from the source and, if set, the static binding location set by a `layout` qualifier.

For Vulkan, each attribute entry contains the `opVariable` index and static binding location set in the SPIR-V module. If present, a symbol name from an associated `opDecorate` annotation also displays.

```
Position attributes
- OpVariable %17 'offset' (location=2)
```

```
- OpVariable %64 (location=0)
```



The Vulkan variable decoration name presented here is the value after the name has been converted into a legal Mali Offline Compiler symbol name. This might not be exactly the same as the value stored in the input SPIR-V module.

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.

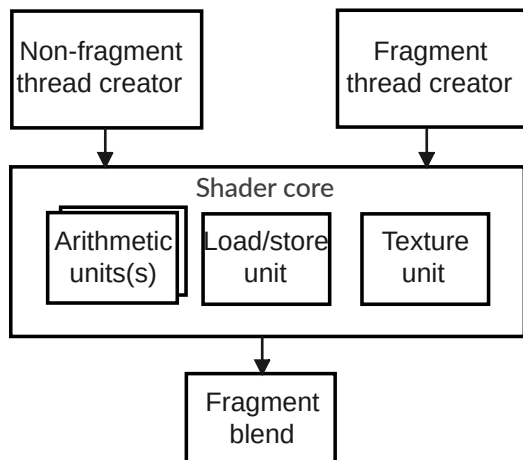
Load/store unit (LS)

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



4.1.1 Midgard work register breakpoints

Arm® 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.

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)

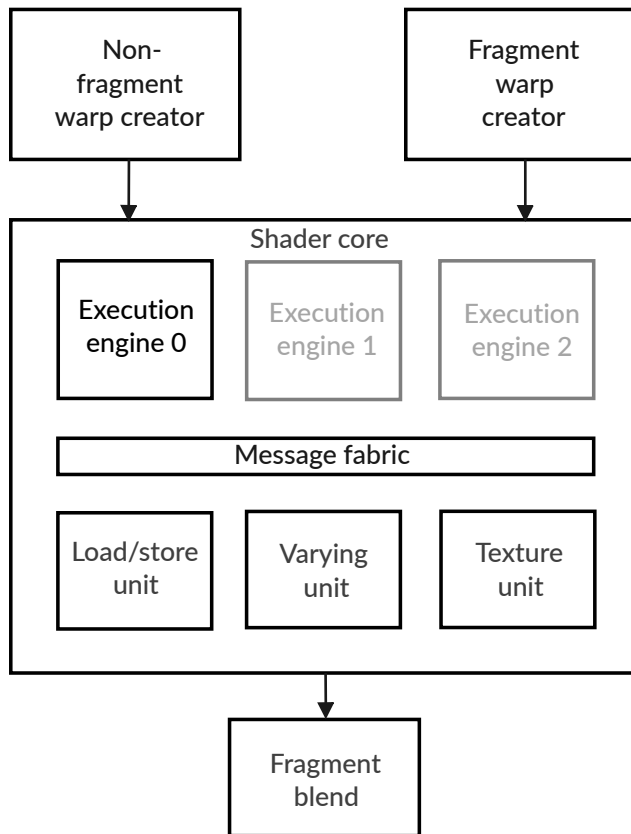
The load/store pipeline handles all non-texture memory access, including buffer access, image access, and atomic operations.

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

4.2.1 Bifrost work register breakpoints

Arm® Mali™ Bifrost GPU shader cores allow you to create variable numbers of threads, 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 work effectively. Aim to use 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.

4.2.2 Bifrost shader core size

The early-generation Bifrost shader cores, Arm® 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.

4.3 Mali Valhall and 5th Generation architectures

Arm® Mali™ Valhall and 5th Generation 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

The Fused Multiply Accumulate (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

The ConVerT (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

The Special Functions Unit (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

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

Varying unit

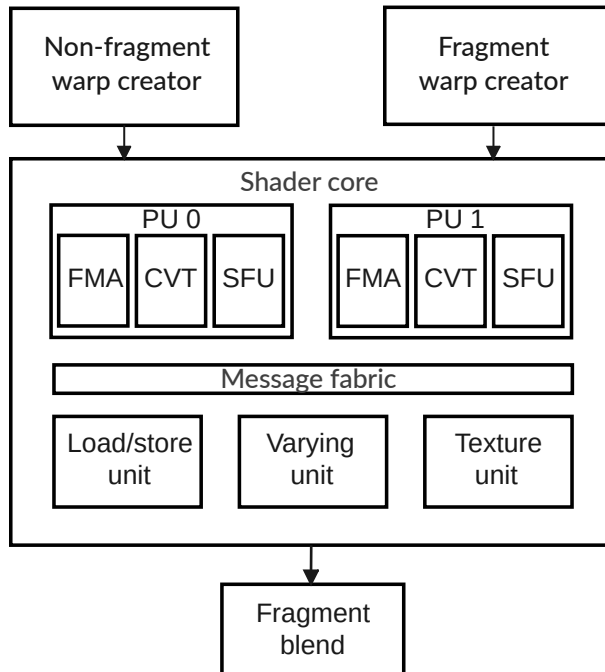
The Varying (V) pipeline is a dedicated pipeline which implements the varying interpolator.

Texture unit

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

For these Mali GPUs the text performance report shows a single combined Arithmetic (A) cycle cost. 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 and 5th Generation shader core



4.3.1 Valhall and 5th Generation architecture work register breakpoints

Arm® Mali™ Valhall and 5th Generation 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 work effectively. Aim to use 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.

4.3.2 Valhall and 5th Generation architecture shader core size

The Arm® Mali™-G57, Mali-G68, Mali-G77, and Mali-G78 shader cores implement the Valhall architecture. These cores implement a four texel-per-clock and two pixel-per-clock shader core, with a variable amount of arithmetic performance depending on GPU model.

The Mali-G610, and Mali-G710 shader cores implement the Valhall architecture. This generation of hardware doubled the shader core throughput to eight texels per clock and four pixels per clock. The per-core cycle counts reported by Mali Offline Compiler are expected to halve compared to the earlier generation. However, these designs often use fewer shader cores to offset the increase in shader core size.

The Mali-G510 and Mali-G310 shader cores implement the Valhall architecture. These cores support configurable amounts of arithmetic, texturing, and pixel throughput. This allows a silicon design to optimize the shader core for the expected workload, which is ideal for cost-sensitive markets. However, the performance per core is not consistent across configurations. The performance reports for Mali Offline Compiler assume the following configurations:

Mali-G310

32 FMA/cycle, 4 texture ops/cycle, 4 pixels/cycle

Mali-G510

48 FMA/cycle, 8 texture ops/cycle, 4 pixels/cycle

You must rescale the reported performance in the reports if your target device uses a different configuration. Check your chipset documentation for the correct configuration.

The Arm Immortalis™-G715, Mali-G715, and Mali-G615 cores implement the Valhall architecture. These cores doubled the FMA arithmetic capability of the core compared to the Mali-G710. Arithmetic cycles reported by Mali Offline Compiler reduces significantly for shaders that are limited by FMA performance.

The Immortalis-G720, Mali-G720, and Mali-G620 cores implement the 5th Generation architecture. These designs use the same arithmetic pipeline architecture as the Immortalis-G715 generation of Valhall shader cores.