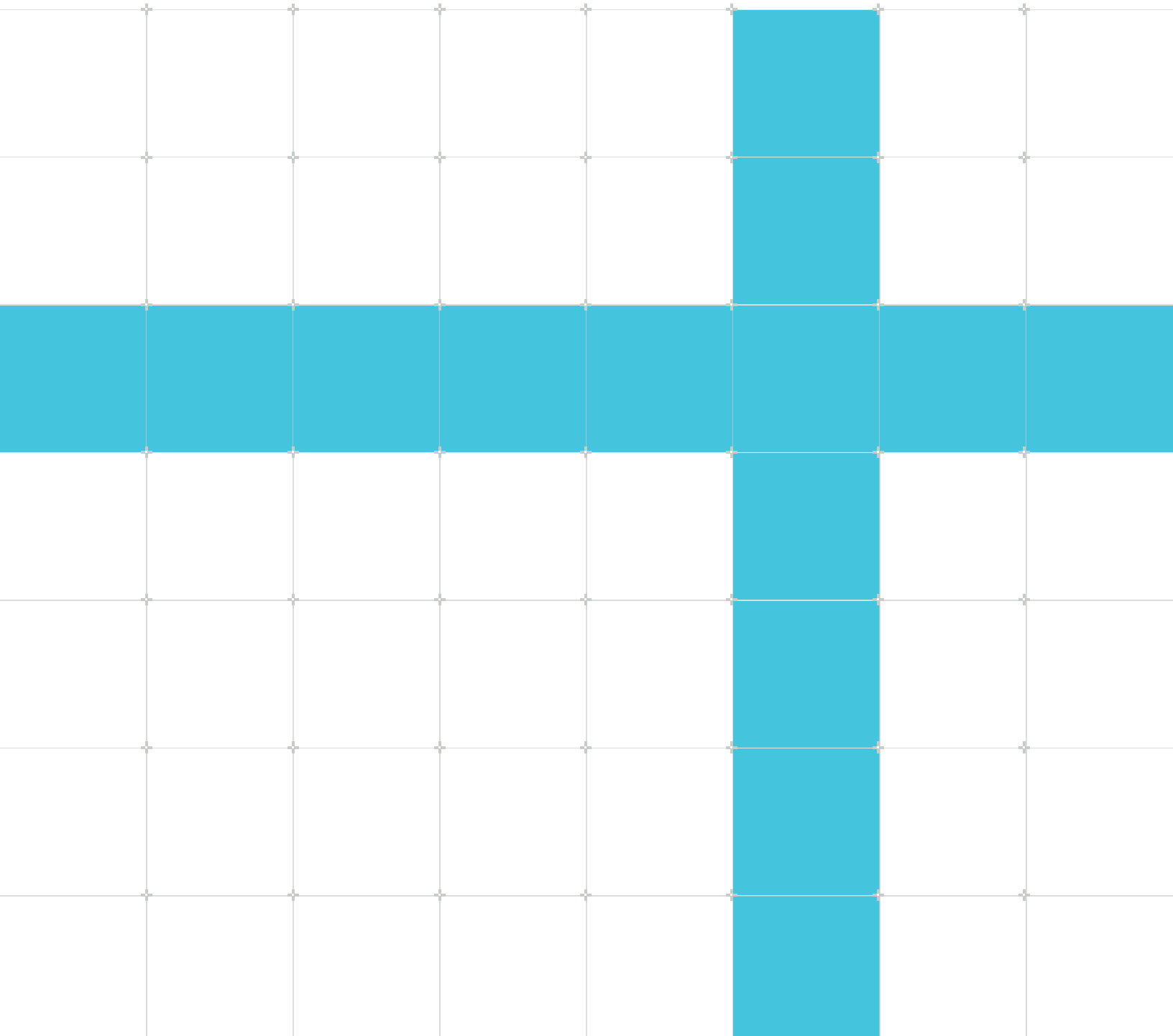
Arm® Mali™ Offline Compiler

Version 8.2

User Guide

Issue 00
101863_0802_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
0704-00	26 August 2021	Non-Confidential	New document for v7.4
0705-00	22 February 2022	Non-Confidential	New document for v7.5
0706-00	26 May 2022	Non-Confidential	New document for v7.6
0707-00	26 August 2022	Non-Confidential	New document for v7.7
0708-00	25 November 2022	Non-Confidential	New document for v7.8
0708-01	1 December 2022	Non-Confidential	Updated document for v7.8
0709-00	15 April 2023	Non-Confidential	New document for v7.9
0800-00	11 June 2023	Non-Confidential	New document for v8.0
0801-00	5 August 2023	Non-Confidential	New document for v8.1
0802-00	22 November 2023	Non-Confidential	New document for v8.2

Proprietary Notice

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.

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

Learn how to use Arm® Mali™ Offline Compiler to analyze performance of graphics shaders and compute kernels targeting Arm Immortalis™ and Mali GPUs.

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
italic	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 underline`	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: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

Recommendations. Not following these recommendations might lead to system failure or damage.

Requirements for the system. Not following these requirements might result in system failure or damage.

Requirements for the system. Not following these requirements will result in system failure or damage.

An important piece of information that needs your attention.

A useful tip that might make it easier, better or faster to perform a task.

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

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 is used to:

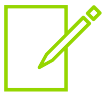
- Validate the syntax of shaders
- Identify performance bottlenecks
- Measure the performance impact of any changes

2.1 API support

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

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

- 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

Mali™ Offline Compiler supports the following Arm GPU products:

Arm 5th Generation architecture

- Immortalis-G720
- Mali-G720
- Mali-G620

Valhall architecture

- Immortalis-G715

- Mali-G715
- Mali-G710
- Mali-G615
- Mali-G610
- Mali-G510
- Mali-G310
- Mali-G78AE
- Mali-G78
- Mali-G77
- Mali-G68
- Mali-G57

Bifrost architecture

- Mali-G76
- Mali-G72
- Mali-G71
- Mali-G52
- Mali-G51
- Mali-G31

Midgard architecture

- Mali-T880
- Mali-T860
- Mali-T830
- Mali-T820
- Mali-T760
- Mali-T720 (No Vulkan support)

Compiler versions

The compiler backends used are taken from the following driver releases:

- Bifrost, Valhall, and 5th Generation architecture GPUs use the r45p0 compiler
- Midgard architecture GPUs use the r23p0 compiler

2.3 Binary generation support

Mali™ Offline Compiler does not provide the ability to generate precompiled binaries for graphics shaders or compute kernels. However, you can create your own precompiled binaries using the GPU driver on your target device.

To create a precompiled binary, you must compile and link your shader program using the production driver on your target device. You must then retrieve the binary using API calls such as `glGetProgramBinary()`.

Compiled program binaries are specific to the pairing of the GPU hardware and driver version used to create them. Reliance on binary distribution is not recommended.

3. Using Mali Offline Compiler

Learn how to install and use Mali™ Offline Compiler to query the capabilities of an Arm GPU, and analyze the performance of shaders and compute kernels compiled for it.

3.1 Install Mali Offline Compiler

This topic describes how to install 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 Mali Offline Compiler:
   - 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 on-screen instructions. For easy access, the directory tree is copied 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. Use Mali Offline Compiler:
   - On Windows:

     Mali Offline Compiler is automatically added to your `PATH` by the Arm Mobile Studio installer. You can invoke `malioc` directly from any working directory.
   - On macOS:

If you use the Mali Offline Compiler terminal launcher provided in the tool installation, Mali Offline Compiler is automatically added to your `PATH`. You can invoke `malioc` directly from any working directory.

If you do not use the launcher you must either:

- Use an absolute path to invoke `malioc`.
- Invoke `malioc` from the `malioc` installation directory.
- Manually add the `malioc` installation directory to your `PATH` before use.

- On Linux:

  Mali Offline Compiler is not automatically added to your `PATH`. You must either:

  - Use an absolute path to invoke `malioc`.
  - Invoke `malioc` from the `malioc` installation directory.
  - Manually add the `malioc` installation directory to your `PATH` before use.

Next steps

Check your compiler configuration, see Querying compiler capabilities.

3.2 Querying compiler capabilities

You can query information about the supported Arm GPUs, and the compiler configuration and extension support of each Arm GPU, from the command line.

- The `--list` option lists all the supported Arm GPUs, and the API support provided for each by this release of Mali™ Offline Compiler.
- The `--info -c <target_gpu>` option shows detailed information for a specific GPU. This includes the compiler version used, and the language versions and extensions that are supported when using this compiler version.

3.3 Compiling OpenGL ES shaders

This topic describes how to use Mali™ Offline Compiler to compile OpenGL ES shaders.

Use the following command-line syntax to compile your shader:

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

The `target_gpu` must be one of the GPUs that are listed in GPU support. If `target_gpu` is not specified the latest supported GPU is used.

The `shader_type` is one of the following options:

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

If you do not specify an explicit shader type, the input file name is used to determine 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.

The `-D<def>` option is used to define a macro on the command line for use in shader compilation. For example:

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

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

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 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.

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 if missing.

3.4 Compiling Vulkan shaders

This topic describes how to use Mali™ Offline Compiler to compile Vulkan shaders.

Use the following command-line syntax to compile your shader:

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

The `target_gpu` must be one of the GPUs that are listed in GPU support. If `target_gpu` is not specified the latest supported GPU is used.

The `shader_type` is one of the following options:

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

If you do not specify an explicit shader type, the input file name is used to determine the shader type:

`.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.

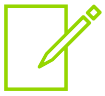
`.rint`
: OpenGL or OpenGL ES syntax ray intersection shader.

`.rmiss`
: OpenGL or OpenGL ES syntax ray miss shader.

`.rcall`
: OpenGL or OpenGL ES syntax ray callable shader.

For SPIR-V binary modules, the supported file extensions are the same as the list above with an appended `.spv`, for example `.vert.spv`. You can force interpretation of a file as a SPIR-V binary module by passing in the `--spirv` option.

Use the `-n <name>` option to specify the name of the SPIR-V entry point for binary module inputs.



For SPIR-V binary modules that contain a single shader stage and entry point, the shader stage and entry point name are automatically determined. This automatic detection is used, even if the file uses a non-standard file extension or is loaded from `stdin`.

Use the `-D<def>` option to define a macro on the command line for use in shader compilation. Macros are only used when compiling from source code, and are ignored when compiling a SPIR-V binary module. For example:

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

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

The input files are either:

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

If you specify multiple input source 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.

To read input from `stdin`, instead of a file on disk, insert a single – character.

Source shaders are converted into a SPIR-V binary module using the version of `glslangValidator` that is provided in the installation.

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 if missing.

3.5 Compiling OpenCL kernels

This topic describes how to use Mali™ Offline Compiler to compile OpenCL kernels.

Use the following command-line syntax to compile your kernel:

```
malioc [-c <target_gpu>] [--opencl <version>] [--kernel] [--spirv] [-n <name>] [-
D<def>] \

 <file1> [<file2> ...] [-o <file>]
```

The `target_gpu` must be one of the GPUs that are listed in GPU support. If `target_gpu` is not specified the latest supported GPU is used.

The `--opencl` option specifies 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.

The `--kernel` option indicates that this is an OpenCL kernel shader type. If you do not use the explicit `--kernel` option, the file extension is used to identify the type as an OpenCL kernel:

`.cl`
: OpenCL C compute kernel.

`.cl.spv`
: OpenCL SPIR-V binary compute kernel.

You can force interpretation of an input file as a SPIR-V binary module by passing in the `--spirv` option, although this is not usually required.

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

For SPIR-V binary modules that contain a single shader stage and entry point, the shader stage and entry point name are automatically determined. This automatic detection is used, even if the file uses a non-standard file extension or is loaded from `stdin`.

Use the `-D<def>` option to define a macro on the command line for use in kernel compilation. Macros are only used when compiling from source code, and are ignored when compiling a SPIR-V binary module. For example:

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

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

The input files are either:

- One or more OpenCL C source files.
- A single SPIR-V binary module that has been compiled using OpenCL semantics.

If you specify multiple input source files:

- They are concatenated in the order in which they are specified, before compilation.
- If you do not explicitly specify `--kernel`, they must all use the `.cl` extension.

To read input from `stdin`, instead of a file on disk, insert a single – character.

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 if missing.

3.5.1 Header includes

The OpenCL C language allows you to use header files to share common implementation code across multiple kernels. To use a header file, use the `#include` preprocessor directive with a correctly set file path.

Relative include file paths use the current working directory as the root of the search path:

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

You can also use absolute file paths to avoid a dependency on the working directory:

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

3.6 Syntax error reporting

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

Error message line numbers are the line number after all input source files have been concatenated.

3.7 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 (100% occupancy)
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
```

3.7.1 Resource usage

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

Work register

Work registers are general purpose read-write registers that are allocated to each running thread. The number of threads that can run in the shader core concurrently is dependent on the number of work registers used by each thread. Reducing the work register usage can increase the number of threads that can run concurrently, which is often beneficial.

To reduce work register usage, use 16-bit data types, or simplify your shader program.

See Arm GPU pipelines for more information about work registers in each Arm® GPU architecture.

Uniform registers

Uniform registers are read-only registers that are allocated to each running program. Uniform registers are used to store 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 additional values.

To reduce uniform register usage, use 16-bit data types, or reduce the number of uniforms and constants in your shader program.

Shared storage

Shared storage is a memory pool that allows threads in a single compute work group to exchange data. Arm GPUs implement shared memory using cached system RAM, so it has the same performance as any other buffer access. Use shared storage only where you need algorithmic data sharing across threads.

To reduce shared memory usage, consider using subgroup operations as an alternative method of sharing data.

Stack spilling

Shaders that run out of work registers must spill additional values to the per-thread stack stored in memory. Stack spills are very expensive for a GPU to process because of the high number of running threads.

You can reduce register pressure and stack spilling in the following ways:

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

16-bit arithmetic

The arithmetic units in Arm GPUs can process either a 32-bit operation or a vec2 16-bit operation per per clock. Using 16-bit data types reduces register pressure, and increases performance and energy efficiency.

You can maximize the percentage of 16-bit arithmetic in the following ways:

- By using the the `lowp` and `mediump` variable qualifiers in GLSL or ESSL source shaders.
- By using the `RelaxedPrecision` variable qualifier in binary SPIR-V modules.

3.7.2 Performance table

The performance table gives an indication of the potential performance of the shader program. All data is shown as a cycle cost, normalized to the performance of a single shader core of the target GPU.

It contains the following rows:

Total Instruction Cycles

The cumulative number of 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.

Longest Path Cycles

An estimate of the number of cycles for the longest control flow path though the shader program. 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. In these cases the row will show 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, allowing 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 Arm 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.

3.7.3 Shader properties

The shader properties section of the report provides information about the behaviors of the shader program that can influence run-time performance.

The following entries describe the properties that can be reported, if relevant to the shader being compiled:

Has uniform computation

This 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. Where possible, move it from the shader into application logic on the CPU.

Has side-effects

This shows if this shader has side-effects that are visible in memory, outside of the fixed graphics pipeline. Side-effects are caused by writes in to:

- Storage buffers
- Images
- Atomics

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

Has slow ray traversal

This 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()`. You must call both `rayQueryProceed()` and `rayQueryInitialize()` 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, Arm recommends 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, Arm 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);
}
```

Has slow shading rate

On some Arm GPUs, Variable Rate Shading (VRS) can run slower than expected if the shading rate block size could be larger than 2x2 in either axis. This check shows as true if the compiler cannot statically prove that a value assigned to `gl_PrimitiveShadingRateEXT` is less than or equal to 2 in both axes.

To avoid slow VRS performance on this hardware generation, Arm recommends using a `min()` combiner to ensure that the shading rate is smaller than 2. Use the combiner to merge a primitive shading rate, or a shading rate image, with a constant per-draw shading rate of 2. The per-draw shading rate provides a driver-visible guarantee on the upper bound, which ensures the fast path behavior is used.

Modifies coverage

This shows if a fragment shader has a coverage mask that the shader program can change, 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.

API-side behavior, such as enabling alpha-to-coverage, can impact coverage masks but is not considered here.

Uses late ZS test

This 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.

You can force use of early ZS testing by setting `layout(early_fragment_tests)` on content where you know it is safe to do so.

API-side behavior, such as disabling depth writes, can impact use of late ZS testing but is not considered here.

Uses late ZS update

This shows if a fragment shader contains logic that forces a late ZS update, for example by using `discard` to dynamically modify the fragment coverage mask. This can cause stalls for later fragments at the same coordinate, because the correct ZS values are not known until later in the pipeline.

You can force use of early ZS testing by setting `layout(early_fragment_tests)` on content where you know it is safe to do so.

Note

API-side behavior, such as enabling alpha-to-coverage, can impact use of late ZS update but is not considered here.

Reads color buffer

This 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 programmatically are treated as transparent, and cannot be used as hidden surface removal occluders.

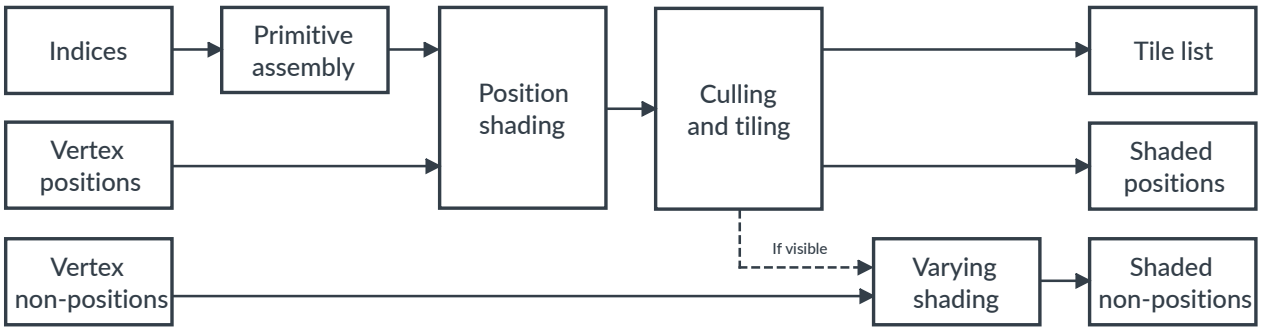
3.7.4 Vertex shader variants

On Arm GPUs since the Bifrost family, vertex shaders use 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 the position output.
- A varying shader, which computes the remaining non-position outputs.

Figure 3-1: IDVS pipeline



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

3.7.5 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® 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 run 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. This memory layout ensures that the position shading phase only loads useful position-related data from DRAM, minimizing 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.


Note

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. If present, the static binding location set by a `layout` qualifier is also shown.

For Vulkan, each attribute entry contains the `OpVariable` index and the static binding location set in the SPIR-V module. If present, the symbol name from an associated `OpDecorate` annotation is also displayed.

```
Position attributes
  - OpVariable %17 'offset' (location=2)
  - OpVariable %64 (location=0)
```


Note

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

3.8 Performance considerations

This topic describes some important considerations you need to be aware of when analyzing data in the performance table.

The cycle measurements are based on the optimal cost of the instructions in the program. For some instructions, the actual performance is also dependent on inputs that are not visible in the

instruction sequence. For example, texturing performance is influenced by the configuration of the bound texture sampler and associated image format. A sampler using a trilinear filter runs at half of the rate of a sampler using a bilinear filter. Trilinear filtering doubles 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 an abstract graph walk through the shader source code. This graph walk is not based on the 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 specific use of the shader.

Offline compilation only processes and optimizes a single shader 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. The lack of whole program optimization in the offline compiler can result in the reported performance being different to the performance that would be seen in production.

You can directly measure pipeline activity on the target platform using the Streamline profiling tools. Profiling with Arm® 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. Arm GPU pipelines

The internal microarchitecture of the shader core can influence both the register usage and the performance 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® GPU architecture.

4.1 Arm Mali Midgard architecture

Mali™ Midgard GPU shader cores have three parallel pipeline classes, comprising an arithmetic pipeline class and two fixed-function support pipeline classes.

The number of arithmetic pipelines is variable, and depends on the target GPU. Data presented in the tool is normalized based on the number of pipelines present, giving an overall cost for the targeted shader core.

The three parallel pipeline classes are:

Arithmetic unit (A)

The arithmetic pipelines run all types of shader arithmetic instruction.

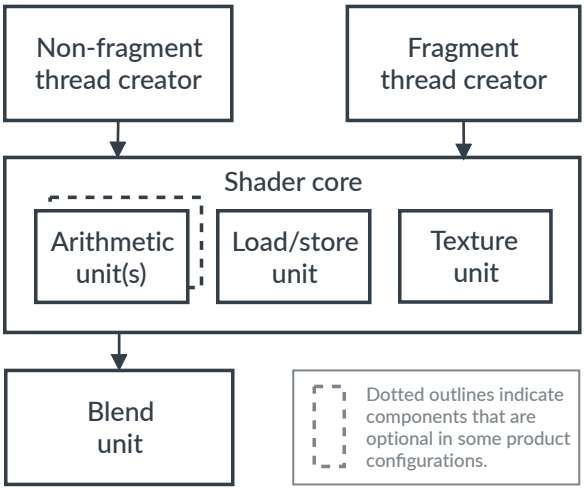
Load/store unit (LS)

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

Texture unit (T)

The texture pipeline handles all texture sampling and filtering operations.

Figure 4-1: Midgard shader core



4.1.1 Arm Mali Midgard work register breakpoints

Mali™ Midgard GPU shader cores can run variable numbers of threads, depending on the number of work registers 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 registers for other shader types.

The most effective way to reduce register pressure is to minimize the precision of stored variables. Use `lowp` or `mediump` precision, or SPIR-V `RelaxedPrecision`, whenever possible.

4.2 Arm Mali Bifrost architecture

Mali™ Bifrost GPU shader cores have four parallel pipeline classes, comprising an arithmetic pipeline class and three fixed-function support pipeline classes.

The number of arithmetic pipelines is variable, and depends on the target GPU. Data presented in the tool is normalized based on the number of pipelines present, giving an overall cost for the targeted shader core.

The four parallel pipeline classes are:

Arithmetic unit (A)

The arithmetic pipelines run all types of shader instruction.

Load/store unit (LS)

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

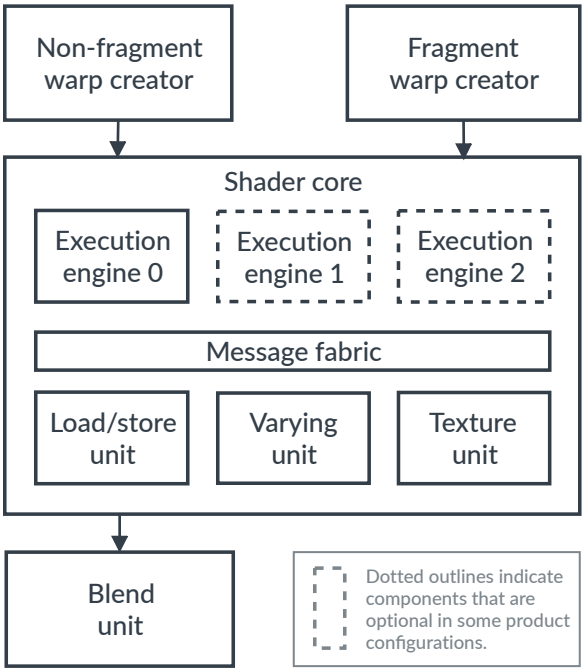
Varying unit (V)

The varying pipeline implements the varying interpolator used to access vertex data during fragment shading.

Texture unit (T)

The texture pipeline handles all texture sampling and filtering operations.

Figure 4-2: Bifrost shader core



4.2.1 Arm Mali Bifrost work register breakpoints

Mali™ Bifrost GPU shader cores can run variable numbers of threads, depending on the number of work registers 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 `lowp` or `mediump` precision, or SPIR-V `RelaxedPrecision`, whenever possible.

4.3 Arm Valhall and 5th Generation architectures

Arm Valhall and 5th Generation GPU shader cores have six parallel pipeline classes, comprising three arithmetic pipeline classes and three fixed-function support pipeline classes.

The number of arithmetic pipelines is variable, and depends on the target GPU. Data presented in the tool is normalized based on the number of pipelines present, giving an overall cost for the targeted shader core.

The six parallel pipeline classes are:

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

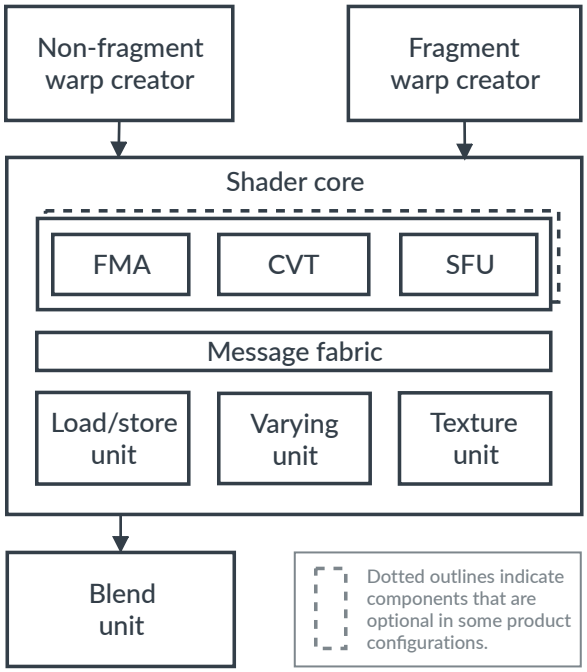
The Varying (V) pipeline is a dedicated pipeline which implements the varying interpolator.

Texture unit

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

For these Arm 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 Arm Valhall and 5th Generation work register breakpoints

Arm Valhall and 5th Generation GPU shader cores can run variable numbers of threads, depending on the number of work registers 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.

4.4 Shader core configurations

Data is presented in the performance table as number of shader core cycles, normalized to the expected performance of a single shader core.

For some products the Arm® GPU does not have a fixed configuration, and can be configured by the silicon implementor. The following Arm® Mali™ product configurations are used:

Mali-G31
: 8 FMA/cycle, 2 texture ops/cycle, 2 pixels/cycle

Mali-G51
: 12 FMA/cycle, 2 texture ops/cycle, 2 pixels/cycle

Mali-G52
: 24 FMA/cycle, 2 texture ops/cycle, 2 pixels/cycle

Mali-G310
: 32 FMA/cycle, 4 texture ops/cycle, 4 pixels/cycle

Mali-G510
: 48 FMA/cycle, 8 texture ops/cycle, 4 pixels/cycle