



# The Utgard Shader Core

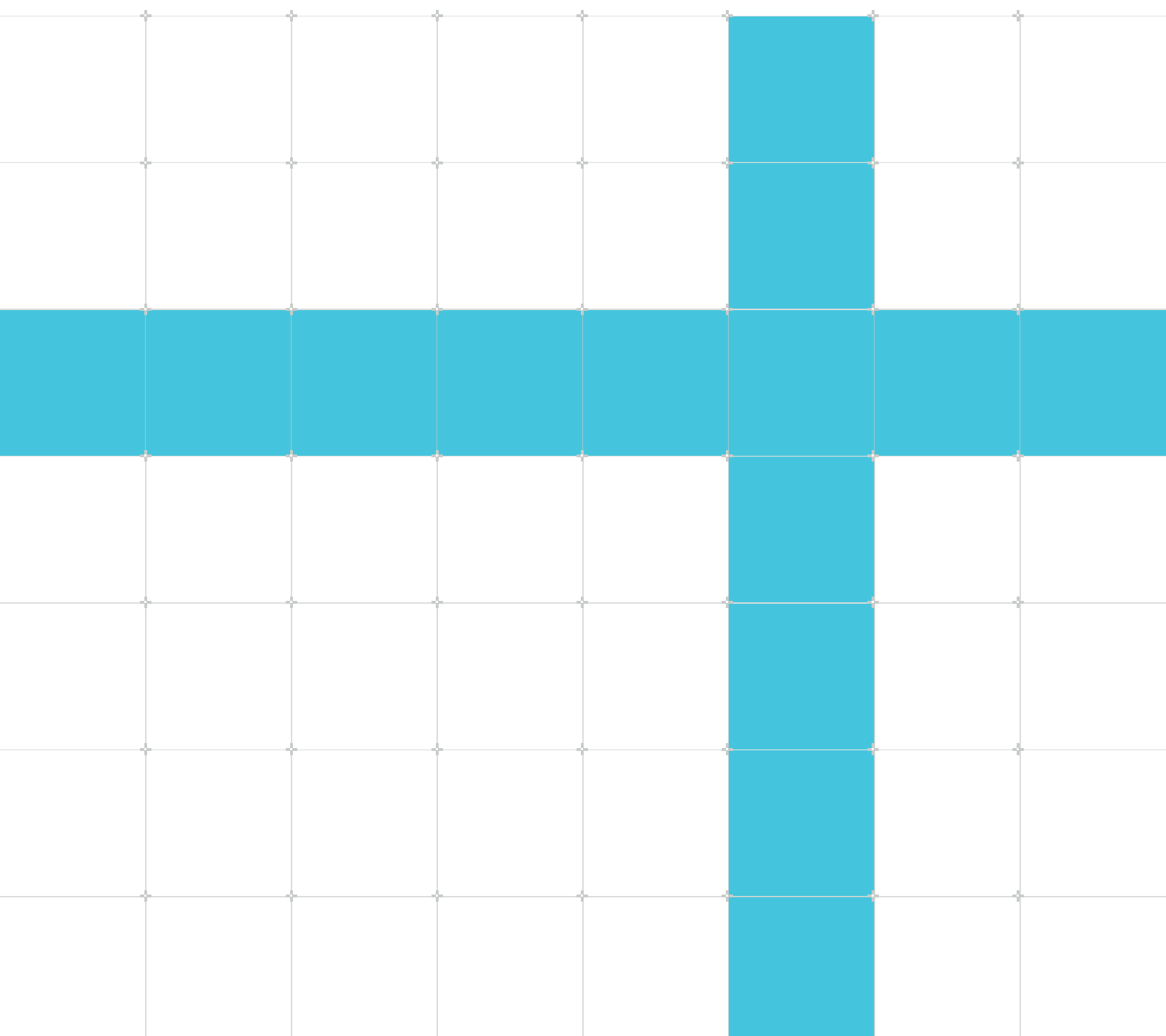
Version 1.0

## Non-Confidential

Copyright © 2020 Arm Limited (or its affiliates).  
All rights reserved.

## Issue 02

102538\_0100\_02\_en



# The Utgard Shader Core

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

## Release information

### Document history

Issue	Date	Confidentiality	Change
0100-02	26 May 2020	Non-Confidential	First release

## 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 © 2020 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](mailto:terms@arm.com).

# Contents

<b>1. Overview.....</b>	<b>6</b>
<b>2. Utgard Shader Core.....</b>	<b>7</b>
<b>3. Vertex shader core.....</b>	<b>8</b>
<b>4. Fragment shader core.....</b>	<b>9</b>
4.1 Load stages.....	11
4.2 Texture stages.....	11
4.3 Arithmetic stages.....	12
4.4 Early and late ZS testing.....	12
<b>5. Performance expectations.....</b>	<b>13</b>
5.1 Platform Dependencies.....	13
<b>6. Next steps.....</b>	<b>14</b>

# 1. Overview

This guide provides an overview of a typical Mali Utgard GPU programmable core. This Mali Utgard GPU programmable core is the first generation of Mali GPUs, and the first to support OpenGL ES 2.0. This core includes the Mali-400, Mali-470, and Mali-450 series products.

To optimize the 2D and 3D performance of your applications, it's important to have a high-level understanding of how the hardware that your application will run on works. For example, understanding the Mali GPU block architecture is important when optimizing using the performance counters of the GPU. This is because this counter data is linked to GPU blocks.

Before reading this guide, it's helpful to have some knowledge of the tile-based rendering approach that the Mali GPUs adopt. You can learn about this in our [Understanding Tile-Based Rendering](#) guide.

By the end of this guide, you will have a better understanding of how the Mali Utgard series of GPUs perform vertex and fragment shader core operations through shared access to the L2 cache. You will also have learned how the three different load stages work and what performance you can expect to get from optimizing for an Utgard GPU.

## 2. Utgard Shader Core

Utgard GPUs use two distinct shader core designs:

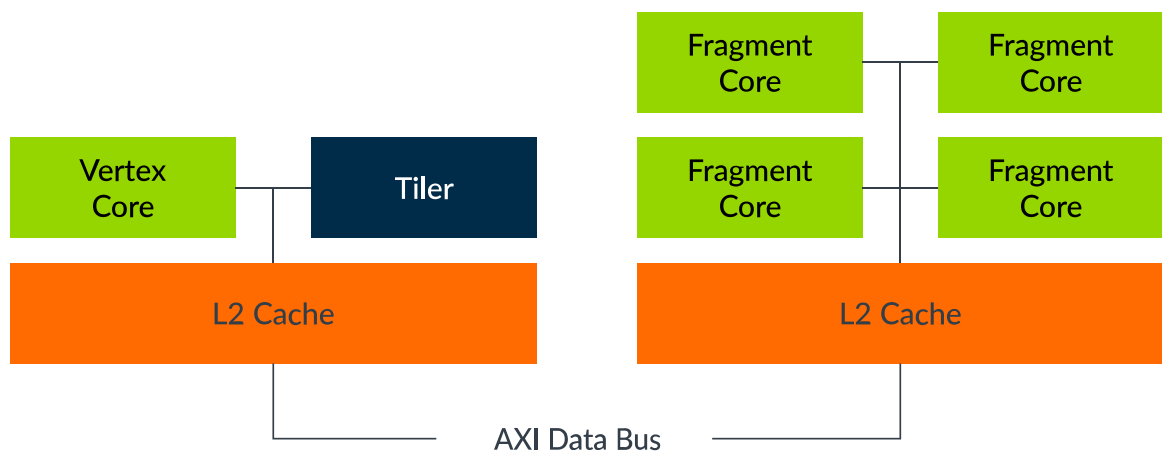
- Vertex shading.
- Fragment shading.

All Utgard GPUs support a single vertex shading core but the exact number of shader cores in a silicon chip varies. Arm licenses configurable designs to our silicon partners, who can then choose how to configure the GPU in their specific chipset, based on their performance needs and silicon area constraints.

Both the Mali-400 and Mali-470 supports up to four fragment shader cores, while the Mali-450 supports up to eight.

This diagram shows the Advanced eXtensible Interface (AXI) Data Bus of a typical Mali Utgard GPU interfacing with the level-2 (L2) Cache:

**Figure 2-1: Utgard top level diagram**



Both the vertex shader core and the tiler share a small L2 cache that acts as a buffer during data exchanges in the shading pipeline. In turn, reducing memory bandwidth usage when data is passed onto the fixed function primitive assembly unit from the vertex shader.

The fragment shader cores share a larger L2 cache to reduce memory bandwidth caused by shared data fetches, such as per-vertex inputs or textures for primitives that span multiple tiles. The size of this L2 is typically 32KB per fragment shader core present, but is configurable upon request by our silicon partners.

### 3. Vertex shader core

The Utgard vertex shader core processes vertices that are shaded in a serial stream, before passing these results to the fixed-function tiling unit for primitive assembly, clipping, culling, and tile list generation.

There is only ever one vertex shader core on an Utgard GPU, and that geometry processing does not increase with multiple fragment shader cores.



The Mali-450's single vertex shader core processes operations at twice the rate of either the Mali-400, or the Mali-470 GPUs.

---

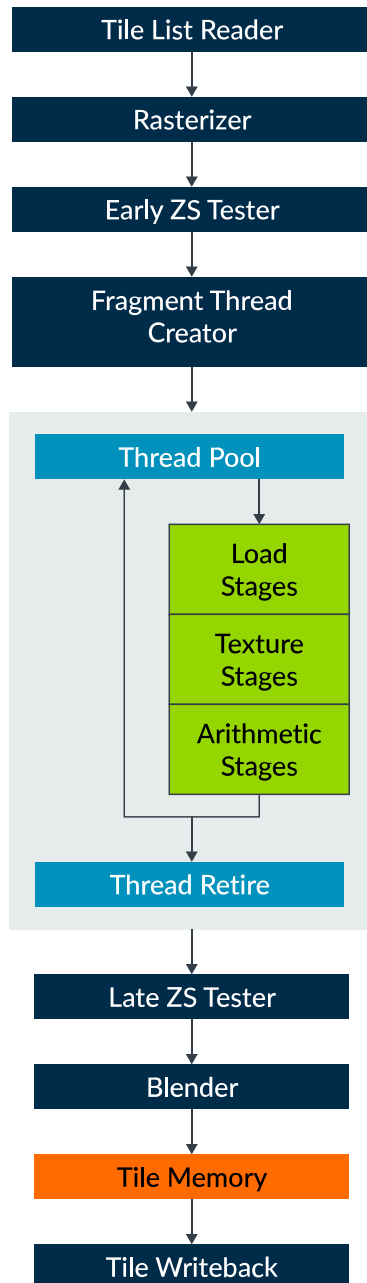


## 4. Fragment shader core

The Utgard fragment shader core consists of a single, programmable, pipeline that is wrapped by fixed function logic for the generation of new fragment threads and the retiring of completed

fragment threads. The fragment shader core data flow for an Utgard GPU is shown in the image below:

**Figure 4-1: Utgard Fragment core diagram**



The programmable fragment shader core is a multi-threaded processing engine that can run up to 128 threads simultaneously, where each running thread equals a single fragment shader program instance.

The large number of threads exists to hide the costs of cache misses and memory fetch latency.

Threads that miss data in the L1 cache can fetch data from L2 cache without any performance penalty. Or, threads that miss data in the L1 cache can fetch data from main memory with only a single cycle of overhead, although this depends on memory latency.

The Utgard programmable pipeline executes all shader programs and contains three types of processing stages, usable by a single instruction issue cycle.

These processing stages are load stages, texture stages, and arithmetic stages.

## 4.1 Load stages

Load stages are responsible for all shader memory accesses that are not related to texture samplers. These accesses include uniform access, varying access and interpolation, and thread stack access.

On a per-clock basis, the load stage can load 64-bits of uniform, interpolate 64-bits of varying data, or load or store 64-bits of stack data.

## 4.2 Texture stages

The texture stages are responsible for all texture memory access. One bilinear filtered texel per-clock can be returned for most texture formats. However, performance varies for some texture formats and filtering modes.

Operation	Performance scaling
Trilinear filter	x2
Depth format	x2
YUV format	x planes

At a cost of one cycle per plane, importing YUV surfaces from camera and video sources can be read directly without needing prior conversion to RGB. However, importing semi-planar Y + UV sources are preferred to fully planar Y + U + V sources.

## 4.3 Arithmetic stages

The arithmetic stages are a Single Instruction Multiple Data (SIMD) vector processing engine, where arithmetic units operate on vec4 fp16 values, and each pipeline can process a total of 14 FP16 FLOPS.



Only `mediump` precision using fp16 data types for fragment shading is supported by Mali-400, `highp` fp32 data types are not supported.

---

## 4.4 Early and late ZS testing

In the OpenGL ES specification, after fragment shading has completed, Fragment operations occurs at the end of the pipeline. Fragment operations includes depth (Z) and stencil (S) testing, Although the OpenGL ES specification is simple to understand, it also implies that fragment shading must occur, even if it must be thrown away afterwards by ZS testing.

Coloring fragments and then discarding them costs a significant amount of performance and wasted energy. So, where possible, early ZS testing is performed before fragment shading occurs. Late ZS testing occurs after fragment shading only when unavoidable. For example, a dependency on a shader fragment that calls `discard` creates an indeterminate depth state until it retires.

## 5. Performance expectations

Setting a realistic performance goal for your application is very important. A Mali Utgard GPU is capable of processing the following tasks in a single clock:

- Issue one new thread per shader core per clock.
- Retire and blend one fragment per shader core per clock.
- Write one pixel per shader core per clock.
- Issue one instruction per shader core per clock.
- Process 14 FP16 operations per clock.
- Read 64 bits of uniform data per clock.
- Interpolate 64 bits of varying data per clock.
- Sample one bilinear filtered texel per clock.

### 5.1 Platform Dependencies

The performance of a Mali GPU in any specific chipset is dependent on both the configuration choices made by the silicon implementation, and the final device form factor the chipset is used in.

Some characteristics, including the number of shader cores and the size of the GPU L2 cache, are visible in terms of the GPU logical configuration that the silicon partner has built.

Other characteristics depend on the memory system's logical configuration, like the memory latency, bandwidth, DDR memory type, and how memory is shared between multiple users.

Some characteristics depend on analogue silicon implementation choices, like which silicon process was used, the target top frequency, the DVFS voltage, and frequency choices available at runtime.

Finally, some characteristics depend on the physical form factor of the device, because this determines the available power budget. Therefore, an identical chipset can have very different peak performance results in different form factor devices.

For example:

- A small smartphone has a sustainable GPU power budget between 1-2 Watts.
- A large smartphone has a sustainable GPU power budget between 2-3 Watts.
- A large tablet has sustainable GPU power budget between 4-5 Watts.
- An embedded device with a heat sink may have a GPU power budget up to 10 Watts.

When combined, it can be difficult to predict the performance of any GPU implementation based on the GPU product name, core count, and top frequency. If you are unsure, write some test scenarios that behave like your own real use cases, and then run them to see how well they work on your target devices.

## 6. Next steps

We recommend that you take the time to have a look at how your application currently interfaces, or hopes to interface, with the target Utgard GPU.

Understanding how to get the most out of the single programmable pipeline, that's wrapped by fixed function logic, as well as how the shader core accesses the L2 cache will prove to be of great use when it comes to learning how to optimize your graphics pipeline for even better compatibility and performance.