



Tile-Based Rendering

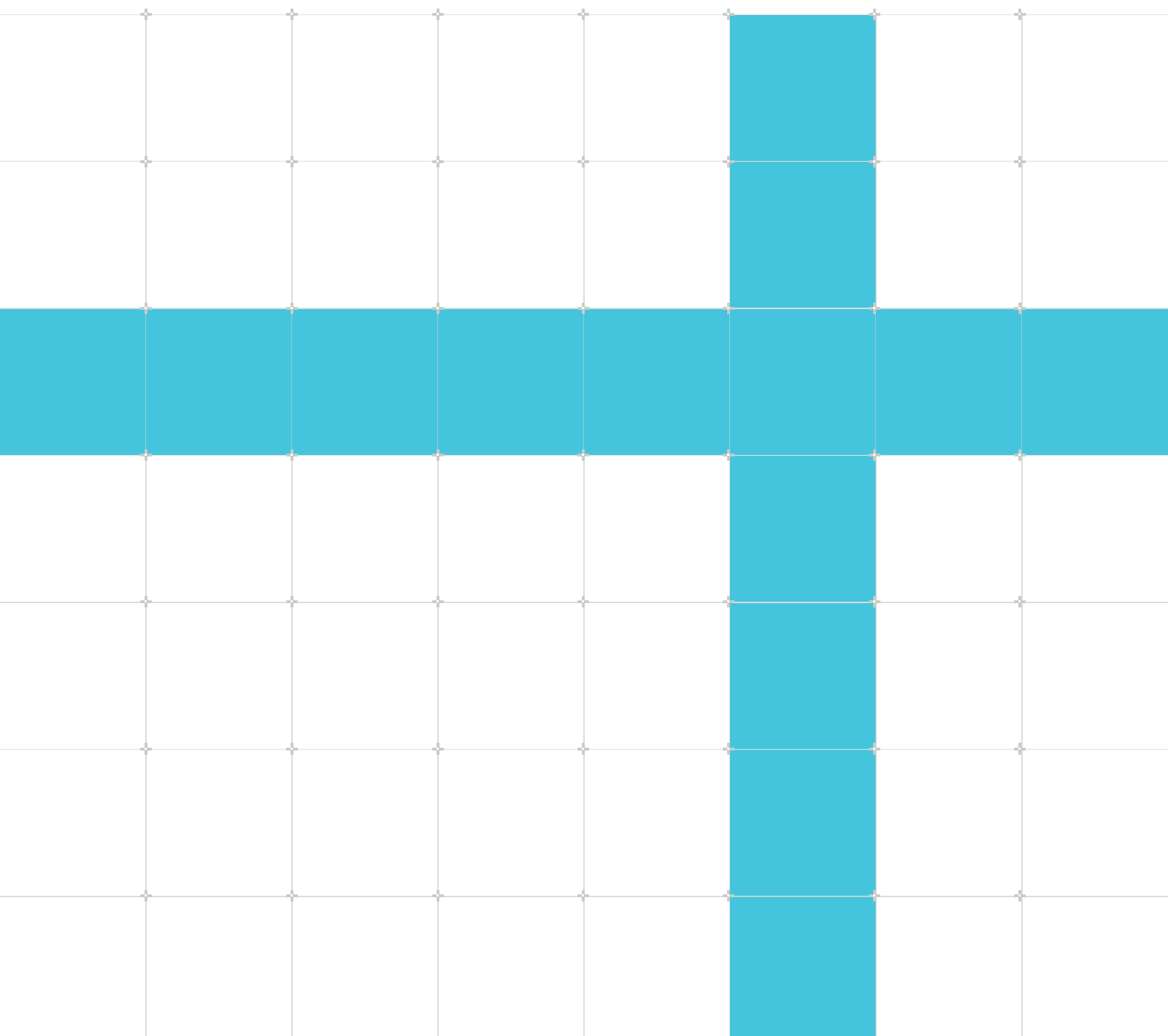
Version 1.0

Non-Confidential

Copyright © 2021 Arm Limited (or its affiliates).
All rights reserved.

Issue 02

102662_0100_02_en



Tile-Based Rendering

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	25 August 2021	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 © 2021 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. Overview.....	6
2. Immediate Mode GPUs.....	7
3. Tile-based GPUs.....	9
4. Lowering Bandwidth Use.....	12
5. Next steps.....	13

1. Overview

This guide introduces the advantages and disadvantages of tile-based GPU architectures. It also compares Arm Mali's tile-based GPU architecture design against the, more traditional, immediate mode GPU that you typically find in a desktop PC or console.

Mali GPUs use a tile-based rendering architecture. This means that the GPU renders the output framebuffer as several distinct smaller sub-regions called tiles. Then it writes each tile out to memory as it is completed. With Mali GPUs, these tiles are small, spanning just 16x16 pixels each.

By the end of this guide you'll understand the key benefits and challenges of immediate mode GPUs and tile-based GPUs.

2. Immediate Mode GPUs

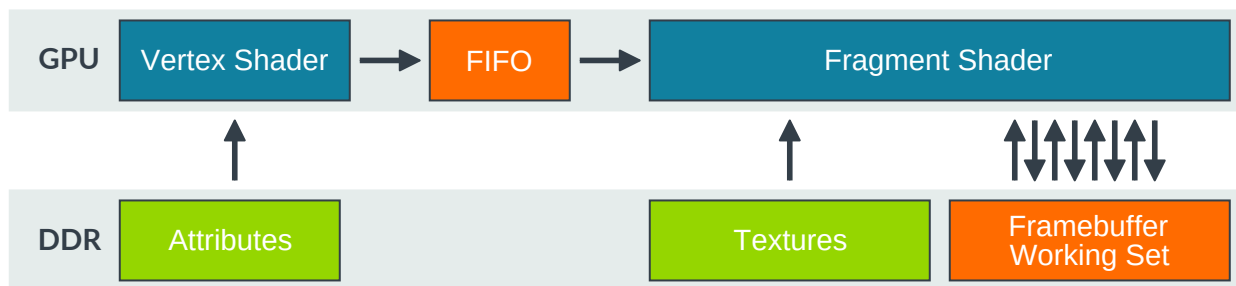
Traditional desktop GPU architecture is commonly known as immediate mode architecture. Immediate mode GPUs process rendering as a strict command stream, executing the vertex and fragment shaders in sequence on each primitive in every draw call.

Ignoring parallel processing and pipelining, here is a high-level pseudo-code example of this approach:

```
python
for draw in renderPass:
    for primitive in draw:
        for vertex in primitive:
            execute_vertex_shader(vertex)
        if primitive not culled:
            for fragment in primitive:
                execute_fragment_shader(fragment)
```

The diagram below shows the hardware data flow and memory interactions:

Figure 2-1: Hardware data flow and memory interactions



Advantages

The output of the vertex shader, and other geometry related shaders, can remain on-chip inside the GPU. The output of these shaders can be stored in a FIFO buffer until the next stage in the pipeline is ready to use the data. This means that the GPU uses little external memory bandwidth storing and retrieving intermediate geometry results.

Disadvantages

The fragment shading jumps around the screen depending on the locations of the triangles in each draw. This happens because any triangle in the stream may cover any part of the screen and triangles are processed in draw order.

The effect of this means that the active working set is the size of the entire framebuffer. For example, consider a device with 1440p resolution, it uses 32 Bits-Per-Pixel (BPP) for color, and 32 BPP for packed depth/stencil. This gives a total working set of 30MB, which is far too large to keep on chip and therefore must be stored off-chip in DRAM.

The GPU must fetch from this working set the current value of the data for the pixel coordinate of the current fragment for every blending, depth testing, and stencil testing operation.

Typically, all shaded fragments access this working set. Therefore, at high resolutions the bandwidth load placed on this memory can be very high because of multiple read-modify-write operations for each fragment. However, caching can mitigate high bandwidth load by keeping recently accessed parts of the framebuffer close to the GPU.

3. Tile-based GPUs

Mali GPUs take a different approach to processing render passes, and this is called tile-based rendering. This approach is designed to minimize the amount of external memory accesses the GPU needs during fragment shading.

Tile-based renders split the screen into small pieces and fragment shade each small tile to completion before writing it out to memory. To make this work, the GPU must know upfront which geometry contributes to each tile. Therefore, tile-based renderers split each render pass into two processing passes:

1. The first pass executes all the geometry related processing, and generates a tile list data structure that indicates what primitives contribute to each screen tile.
2. The second pass executes all the fragment processing, tile by tile, and writes tiles back to memory as they have been completed. Note that Mali GPUs render 16x16 tiles.

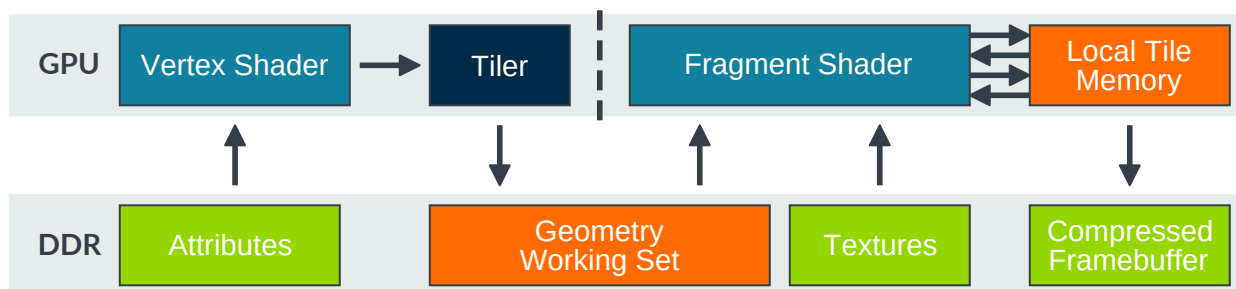
Here is an example of the rendering algorithm for tile-based architectures:

```
python
# Pass one
for draw in renderPass:
    for primitive in draw:
        for vertex in primitive:
            execute_vertex_shader(vertex)
        if primitive not culled:
            append_tile_list(primitive)

# Pass two
for tile in renderPass:
    for primitive in tile:
        for fragment in primitive:
            execute_fragment_shader(fragment)
```

This image shows the hardware data flow and interactions with memory: </>

Figure 3-1: Hardware data flow and interactions



Advantage: Bandwidth

The main advantage of tile-based rendering is that a tile is only a small fraction of the total framebuffer. Therefore, it is possible to store the entire working set of color, depth, and stencil on fast on-chip RAM, that is tightly coupled to the GPU shader core.

The framebuffer data that the GPU needs for depth testing and for blending transparent fragments is therefore available without requiring external memory access. Fragment-heavy content can be made significantly more energy efficient by reducing the number of external memory accesses the GPU needs for common framebuffer operations.

Also, a significant proportion of content has a depth and stencil buffer that is transient and only needs to exist for the duration of the shading process. If you tell the Mali drivers that an attachment does not need to be preserved, then the drivers will not write them back to main memory.

You can achieve this with a call to `glDiscardFramebufferEXT` in OpenGL ES 2.0, `glInvalidateFramebuffer` in OpenGL ES 3.0, or using the appropriate render pass `storeOp` settings in Vulkan.

More bandwidth optimizations are possible because Mali GPUs only have to write the color data for a tile back to memory once rendering is complete, and at this point you know its final state. You can compare the content of a tile with the current data already in main memory with a Cyclic Redundancy Check (CRC) check. This runs a process called Transaction Elimination. This process skips writing the tile to external memory if there is no change in color.

In many situations, Transaction Elimination does not help performance as the fragment shaders must still build the tile content. However, the process greatly reduces the external memory bandwidth in many common use cases, such as UI rendering and casual gaming. As a result, it also reduces system power consumption.

Mali GPUs can also compress the color data for the tiles that they write out using a lossless compression scheme called Arm Frame Buffer Compression (AFBC), which lowers the bandwidth and power consumed even further.



AFBC works for render-to-texture workloads. However, compression of the window surface requires an AFBC enabled display controller. Framebuffer compression therefore saves bandwidth multiple times; once on write out from the GPU and once each time that framebuffer is read.

Advantage: Algorithms

Tile-based renderers enable some algorithms that would otherwise be too computationally expensive or too bandwidth heavy.

A tile is small enough that a Mali GPU can store enough samples locally in memory to allow Multi-Sample Anti-Aliasing (MSAA). As a result, the hardware can resolve multiple samples to a single pixel color during tile writeback to external memory without needing a separate resolve pass. The Mali architecture allows for very low performance overhead and bandwidth costs when performing anti-aliasing.

Some advanced techniques, such as deferred lighting, can benefit from fragment shaders programmatically accessing values that are stored in the framebuffer by previous fragments.

Traditional algorithms would implement deferred lighting using Multiple Render Target (MRT) rendering, writing back multiple intermediate values per pixel back to main memory, and then re-reading them in a second pass.

4. Lowering Bandwidth Use

Tile-based renderers can enable lower bandwidth approaches where intermediate per-pixel data is shared directly from the tile-memory, and the GPU only writes the final lit pixels back to memory.

For a deferred shading G-Buffer, that can use four 1080p 32bpp intermediate textures, this approach can save up to 4GB/s of bandwidth at 60 FPS.

The following extensions expose this functionality in OpenGL ES:

- `ARM_shader_framebuffer_fetch`
- `ARM_shader_framebuffer_fetch_depth_stencil`
- `EXT_shader_pixel_local_storage`

In Vulkan, using mergeable subpasses allows access to this functionality.

Disadvantages

Tile-based rendering carries a number of advantages, in particular it gives significant reductions in the bandwidth associated with framebuffer data and provides low-cost anti-aliasing. However, there is an important downside to consider.

The principal additional overhead of any tile-based rendering scheme applies at the point of hand-over from the geometry pass to the fragment pass.

The GPU must store the output of the geometry pass – the per-vertex varying data and tiler intermediate state – to main memory, which the fragment pass will subsequently read. There is therefore a balance to be struck between the extra bandwidth costs related to geometry, and the bandwidth savings for the framebuffer data.

It is also important to consider that some rendering operations, such as tessellation, are disproportionately expensive for a tile-based architecture. These operations are designed to suit the strengths of the immediate mode architecture where the explosion in geometry data can be buffered inside the on-chip FIFO buffer, rather than being written back to main memory.

5. Next steps

Understanding the fundamentals of tile-based GPUs will help you as you begin to work with Arm Mali GPUs. In addition, it's important to consider both the advantages and disadvantages of tile-based renderers as you start to consider how to optimize your graphics workflow to get the best performance from the Mali architecture.