Tile-Based Rendering

Previous Section

Next Section

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.

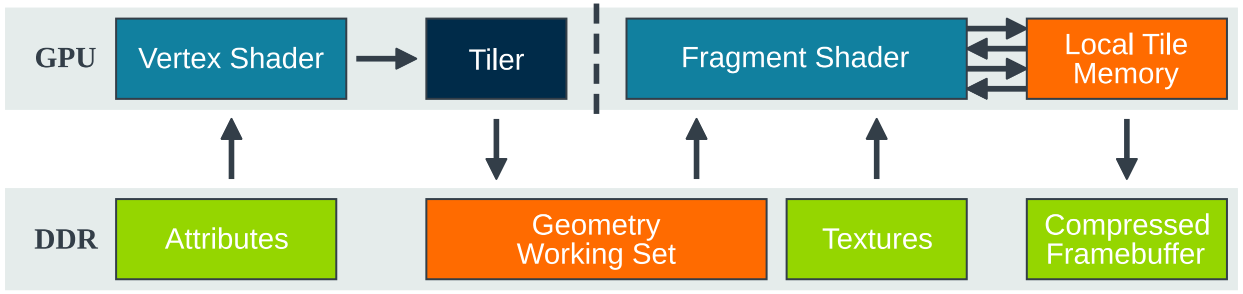
Here is an example of the rendering algorithm for tile-based architectures:

~~~~
```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:</>



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.

Note that 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.

← Previous Section

Next Section →

Development

SoC Design

Embedded Software

Graphics and Multimedia

High Performance Computing

Linux and Open Source

Research and Education

Architecture

CPU Architecture

System Architectures

Security Architectures

Instruction Sets

Platform Design

Products

CPU Processors

Graphics and Multimedia

Physical IP

System IP

System Design Tools

Software Development Tools

Support

Design Reviews

Training

Documentation

Licensing

Downloads

Contact Support

Arm Security Updates

Community

Communities

About Arm

Leadership