Mali移动游戏优化方略

Frank Lei
DevRel Shanghai
Agenda

1. Analyse the performance of Epic Citadel
   - Software Profiling
   - GPU Profiling
   - Using the ARM® Mali™ GPU hardware counters to find the bottleneck

2. Debugging with Mali Graphics Debugger
   - Overdraw and frame analysis

3. Q & A
Importance of Analysis & Debug

- **Mobile Platforms**
  - Expectation of amazing console-like graphics and playing experience
  - Screen resolution beyond HD
  - Limited power budget

- **Solution**
  - ARM® Cortex® CPUs and Mali™ GPUs are designed for low power whilst providing innovative features to keep up performance
  - Software developers can be “smart” when developing apps
  - Good tools can do the heavy lifting
Mali GPU Software Tools
Performance Analysis, Debug, and Software Development

Mali Graphics Debugger
- API Trace & Debug
- Debug and improve performance at frame level

ARM DS-5 Streamline
- Mali GPU
  - Timeline
  - HW Counters

Mali Offline Compiler
- Generate binary shaders
- Analyze shaders
- Command line tool

OpenGL ES Emulator
- Emulate OpenGL ES 2.0 and 3.0 on desktop
- Windows and Linux
- Khronos Conformant

Texture Compression Tool
- Command line and GUI
- ETC, ETC2
- ASTC
- 3D textures

Integration with partners’ tools

Third party tools
Performance Analysis and Debug with Mali GPUs

Analyze
DS-5 Streamline

Debug
Mali Graphics Debugger

Optimize
Mali Offline Compiler
Main Bottlenecks

- **CPU**
  - Too many draw calls
  - Complex physics

- **Vertex processing**
  - Too many vertices
  - Too much computation per vertex

- **Fragment processing**
  - Too many fragments, overdraw
  - Too much computation per fragment

- **Bandwidth**
  - Big and uncompressed textures
  - High resolution framebuffer

- **Battery life**
  - Energy consumption strongly affects User Experience
Shaders

Vertex Shader

Fragment Shader
Analyzing and Debugging on Device
“Epic Citadel” – A Case Study
CPU Activity ➞ User 24%
GPU Fragment ➞ Activity 99%
GPU Vertex ➞ Activity 44%
The Application is GPU bound
The CPU has to wait until the fragment processing has finished

While rendering the most complicated scene, the application is capable of 36 fps (28ms/frame)
CPU Bound

![Diagram showing CPU, Memory, Vertex Shader, and Fragment Shader connections.](image-url)
CPU Bound

- Mali GPU is a deferred architecture
  - Do not force a pipeline flush by reading back data (glReadPixels, glFinish, etc.)
  - Reduce the amount of draw calls
  - Try to combine your draw calls together

- Offload some of the work to the GPU
  - Move physics from CPU to GPU

- Avoid unnecessary OpenGL® ES calls (glGetError, redundant stage changes, etc.)
GPU Activity Analysis
ARM® Mali™ GPU Hardware Counters

- Over the highlighted time of one second, the GPU was active for \(448\)m cycles (Mali Job Manager Cycles \(\rightarrow\) GPU cycles)

- With this hardware, the maximum number of cycles is \(450\)m

- A first pass of optimization would lead to a higher frame rate

- After reaching V-SYNC, optimization can lead to saving energy and to a longer play time
Vertex and Fragment Processing

- GPU is spending:
  - **186m** (41%) on vertex processing
    *(ARM® Mali™ Job Manager Cycles ➞ JS1 cycles) / GPU frequency*
  - **448m** (99%) on fragment processing
    *(Mali Job Manager Cycles ➞ JS0 cycles) / GPU frequency*
ARM® Mali™-T628 GPU Tripipe Cycles

- Arithmetic instructions
  - Math in the shaders
- Load & Store instructions
  - Uniforms, attributes and varyings
- Texture instructions
  - Texture sampling and filtering

- Instructions can run in parallel
- Each one can be a bottleneck
- There are two arithmetic pipelines so we should aim to increase the arithmetic workload
Inspect the Tripipe Counters
Reduce the load on the L/S pipeline

- GPU Cycles 448m
- Tripipe Cycles 444m
- Load & Store 408m
- Texture 197m
- Arithmetic 105m
It’s easy to calculate a couple of CPI (cycles per instruction) metrics:

- For the load/store pipeline we have:
  \[\frac{408m}{195m} = 2.09 \text{ cycles/instruction}\]

- For the texture pipeline we have:
  \[\frac{197m}{169m} = 1.16 \text{ cycles/instruction}\]
Vertex Bound
Vertex Bound

- Get your artist to remove unnecessary vertices
- LOD switching
  - Only objects near the camera need to be in high detail
- Use culling
- Too many cycles in the vertex shader
Vertex Count and Shader Optimizations

Identify the top heavyweight vertex shaders
Fragment Bound

- CPU
- Memory
- Vertex Shader
- Fragment Shader
Fragment Bound

- Render to a smaller framebuffer
- Move computation from the fragment to the vertex shader (use HW interpolation)
- Drawing your objects front to back instead of back to front reduces overdraw
- Reduce the amount of transparency in the scene
Overdraw

- This is when you draw to each pixel on the screen more than once
  - Drawing your objects front to back instead of back to front reduces overdraw
- The transparent objects must be drawn back to front for a correct ordering
  - Limiting the amount of transparency in the scene can help
- For OpenGL® ES 3.0, avoid modifying the depth buffer from the fragment shader or you will not benefit from the early Z testing, making the front to back sorting useless
Overdraw Factor

- We divide the number of output pixels by the number of fragments, each rendered fragment corresponds to one fragment thread and each tile is 16x16 pixels, thus in our case:

\[
\frac{90.7m \text{ (Mali Core Threads)} \rightarrow \text{Fragment threads}}{143K \text{ (Mali Fragment Tasks)} \rightarrow \text{Tiles rendered}} \times 256 = 2.48 \text{ threads/pixel}
\]
Frame Capture

Draw 188, total vertices: 129024
Frame Analysis

Check the overdraw factor

Draw 181, total vertices: 118594
**Shader Map and Fragment Count**

Identify the top heavyweight fragment shaders

---

**Fragment Count Per Program**

<table>
<thead>
<tr>
<th>Program</th>
<th>Name</th>
<th>Instructions</th>
<th>Shortest</th>
<th>Longest</th>
<th>Instances</th>
<th>Total cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>175</td>
<td>Shader 177</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>7537773</td>
<td>3768865</td>
</tr>
<tr>
<td>280</td>
<td>Shader 282</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1459254</td>
<td>7296270</td>
</tr>
<tr>
<td>281</td>
<td>Shader 183</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>415710</td>
<td>2078550</td>
</tr>
<tr>
<td>187</td>
<td>Shader 189</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>197329</td>
<td>1183974</td>
</tr>
<tr>
<td>73</td>
<td>Shader 75</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>279555</td>
<td>1118220</td>
</tr>
<tr>
<td>382</td>
<td>Shader 384</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>129913</td>
<td>1039304</td>
</tr>
<tr>
<td>289</td>
<td>Shader 291</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>16856</td>
<td>101136</td>
</tr>
<tr>
<td>208</td>
<td>Shader 210</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>7975</td>
<td>39875</td>
</tr>
<tr>
<td>262</td>
<td>Shader 264</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6025</td>
<td>30125</td>
</tr>
<tr>
<td>400</td>
<td>Shader 402</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>914</td>
<td>4570</td>
</tr>
</tbody>
</table>

~10m instances / (2560×1600) pixel = 2.44
Shader Optimization

- Since the arithmetic workload is not very big, we should **reduce the number of uniform and varyings** and calculate them **on-the-fly**
- **Reduce their size**
- **Reduce their precision**: is `highp` always necessary?
- **Use the Mali Offline Shader Compiler!**
  
Bandwidth Bound

- CPU
- Memory
- Vertex Shader
- Fragment Shader
Bandwidth

- When creating embedded graphics applications bandwidth is a scarce resource
  - A typical embedded device can handle 5.0 Gigabytes a second of bandwidth
  - A typical desktop GPU can do in excess of 100 Gigabytes a second

- The application is not bandwidth bound as it performs, over a period of one second:
  
  \[
  \text{(96m (Mali L2 Cache $\rightarrow$ External read beats) + 90.7m (Mali L2 Cache $\rightarrow$ External write beats)) } \times 16 \\
  \approx 2.9 \text{ GB/s}
  \]

- Since bandwidth usage is related to energy consumption it’s always worth optimizing it.
Bandwidth Bound

Vertices
- Reduce the number of vertices and varyings
- Interleave vertices, normals, texture coordinates
- Use Vertex Buffer Objects

Fragments
- Use texture compression
- Enable texture mipmapping

This will also cause a better cache utilization.
Texture Compression Formats

- **ETC – ARM Mali-400 GPU**
  - 4bpp
  - RGB No alpha channel

- **ETC2 – ARM Mali-T604 GPU**
  - 4bpp
  - Backward compatible
  - RGB also handles alpha & punch through

- **ASTC – ARM Mali-T624 GPU and beyond**
  - 0.8bpp to 8bpp
  - Supports RGB, RGB alpha, luminance, luminance alpha, normal maps
  - Also supports HDR
  - Also supports 3D textures
Textures

Save memory and bandwidth with texture compression

- The current most popular format is ETC Texture Compression
- But ASTC (Adaptive Scalable Texture Compression) can deliver < 1 bit/pixel
Covered today:

- Performance analysis with ARM DS-5 Streamline
- Software Profiling
- GPU Profiling
- Debugging with the ARM® Mali™ Graphics Debugger

Thank You

Any Questions?
malideveloper.arm.com

ds.arm.com

community.arm.com/community/arm-cc-cn

malidevelopers@arm.com
Mali tools update
Streamline: OpenCL Timeline
Streamline: OpenCL Timeline
Streamline: OpenCL Timeline
Mali Graphics Debugger 3.0: Overview
Mali Graphics Debugger 3.0: Timeline
Mali Graphics Debugger 3.0: Frame Capture