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

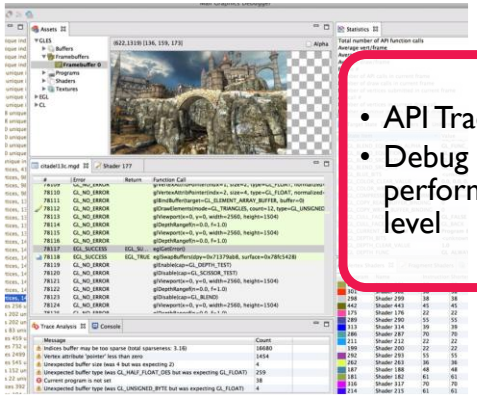
- Solution
  - ARM® Cortex® CPUs and Mali™ GPUs are designed for low power whilst providing innovative features to keep up performance
  - Software developers can be “smart” when developing apps
  - Good tools can do the heavy lifting

ARM

Mali GPU Software Tools

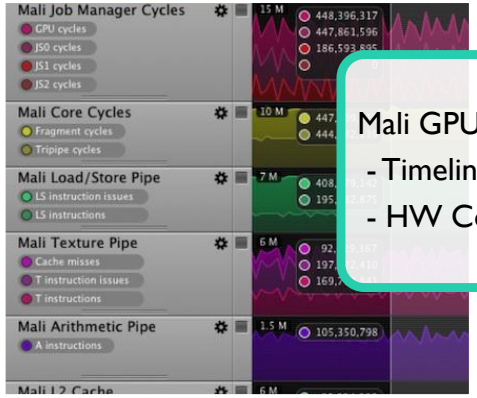
Performance Analysis, Debug, and Software Development

Mali Graphics Debugger



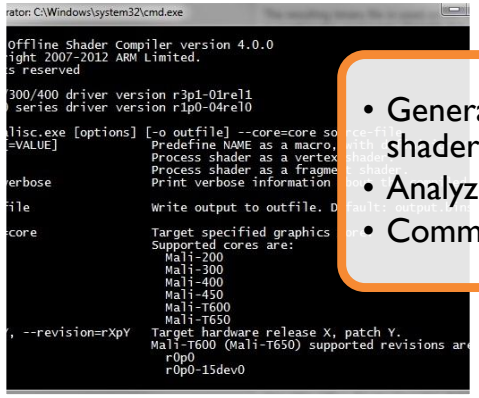
- API Trace & Debug
- Debug and improve performance at frame level

ARM DS-5 Streamline



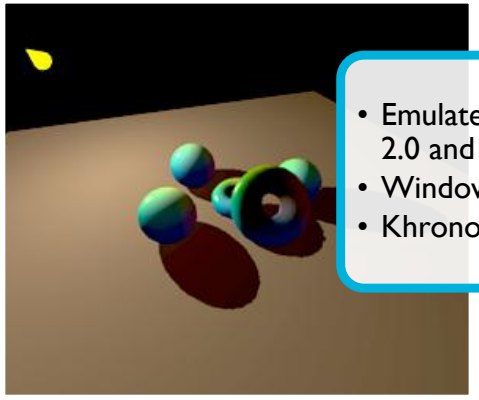
Mali GPU
- Timeline
- HW Counters

Mali Offline Compiler



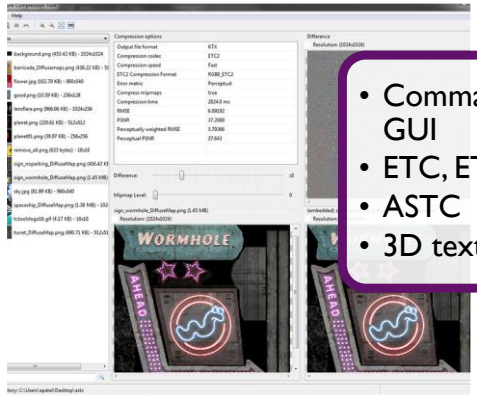
- Generate binary shaders
- Analyze shaders
- Command line tool

OpenGL ES Emulator



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

Texture Compression Tool

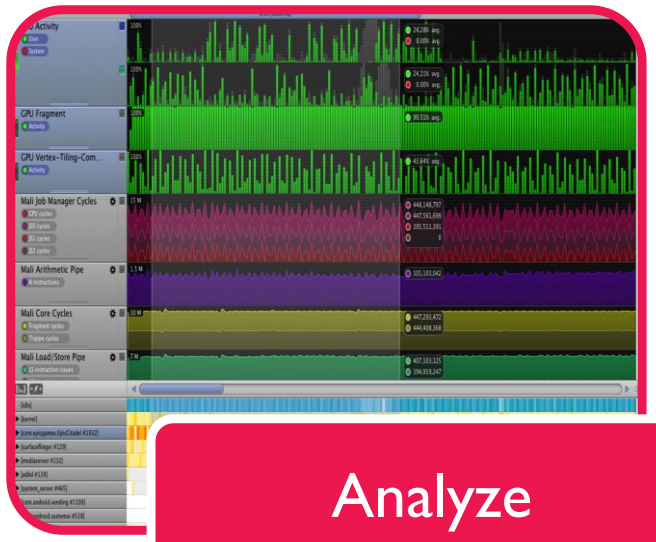


- Command line and GUI
- ETC, ETC2
- ASTC
- 3D textures

Third party tools

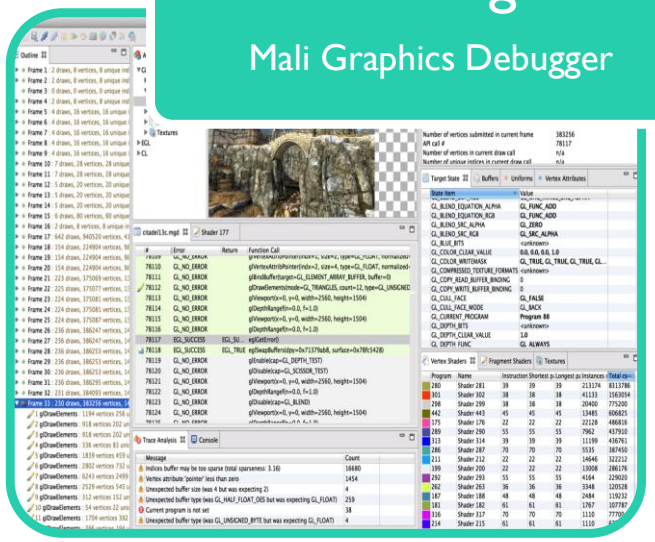
Integration with partners' tools

Performance Analysis and Debug with Mali GPUs



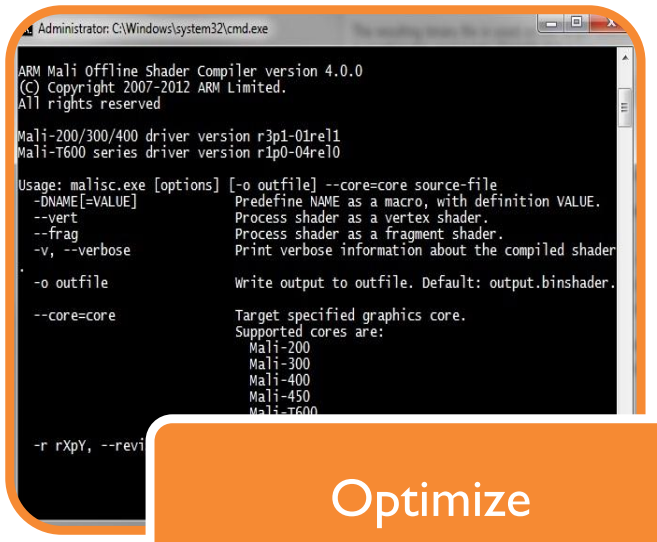
Analyze

DS-5 Streamline



Debug

Mali Graphics Debugger

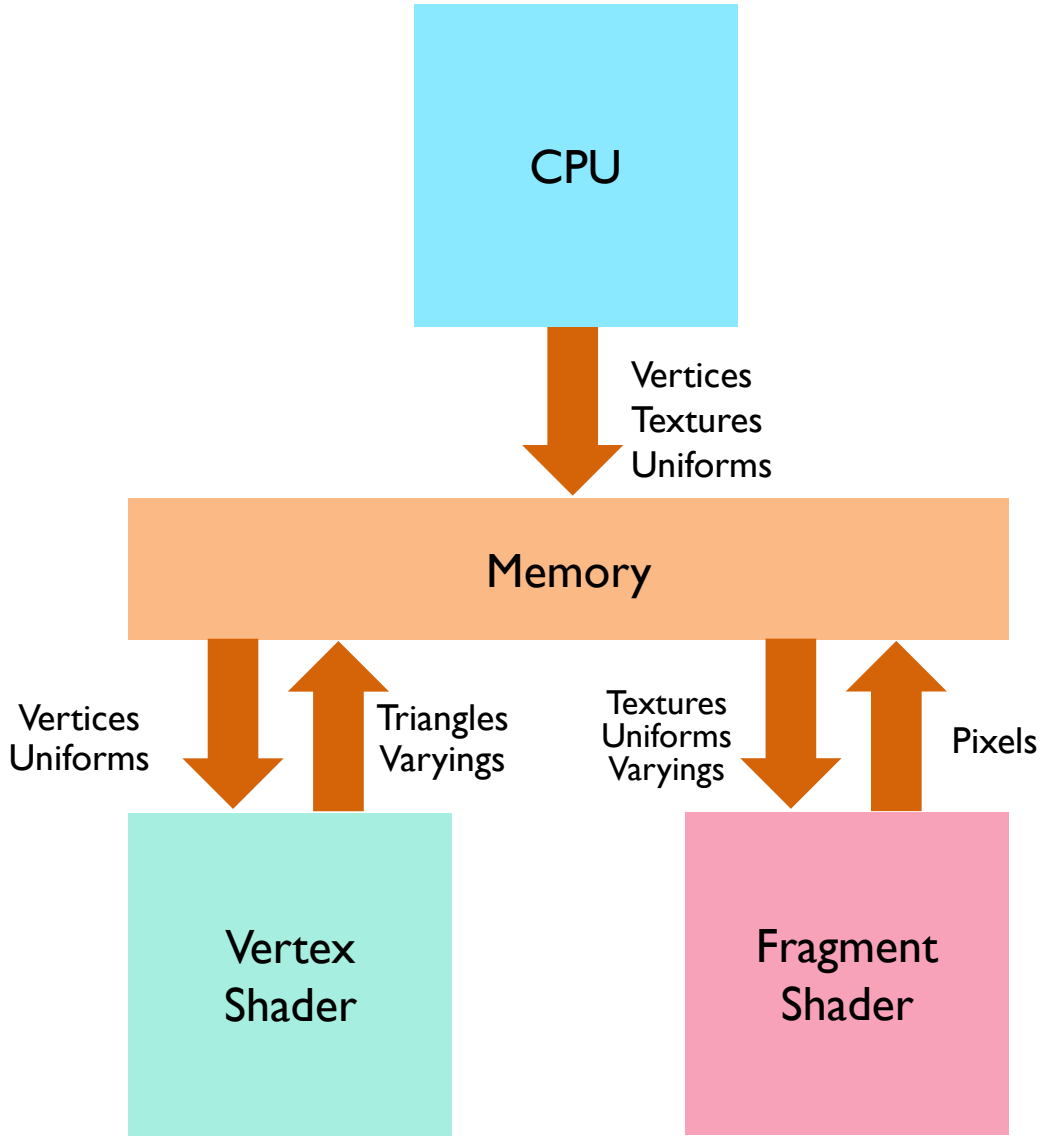


Optimize

Mali Offline Compiler

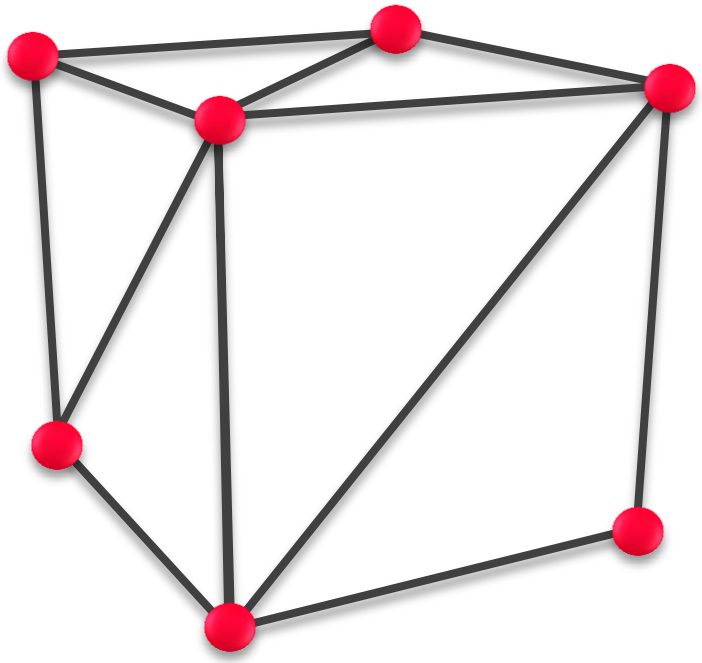
Main Bottlenecks

- **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

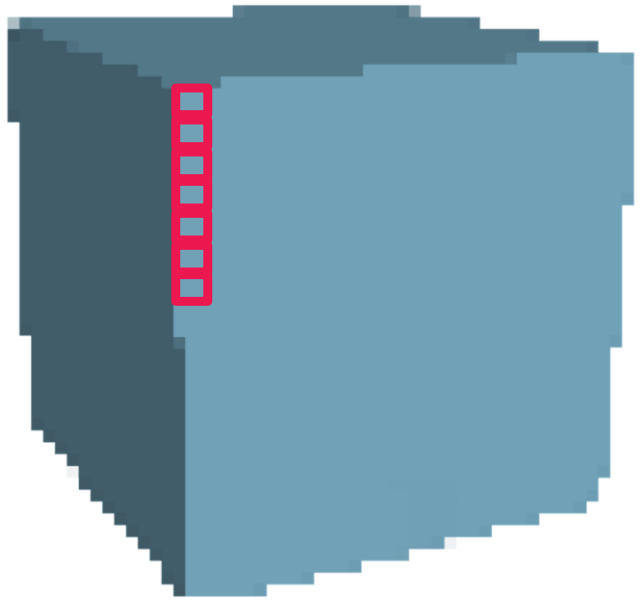


ARM

Shaders



Vertex Shader



Fragment Shader

ARM

Analyzing and Debugging on Device

8

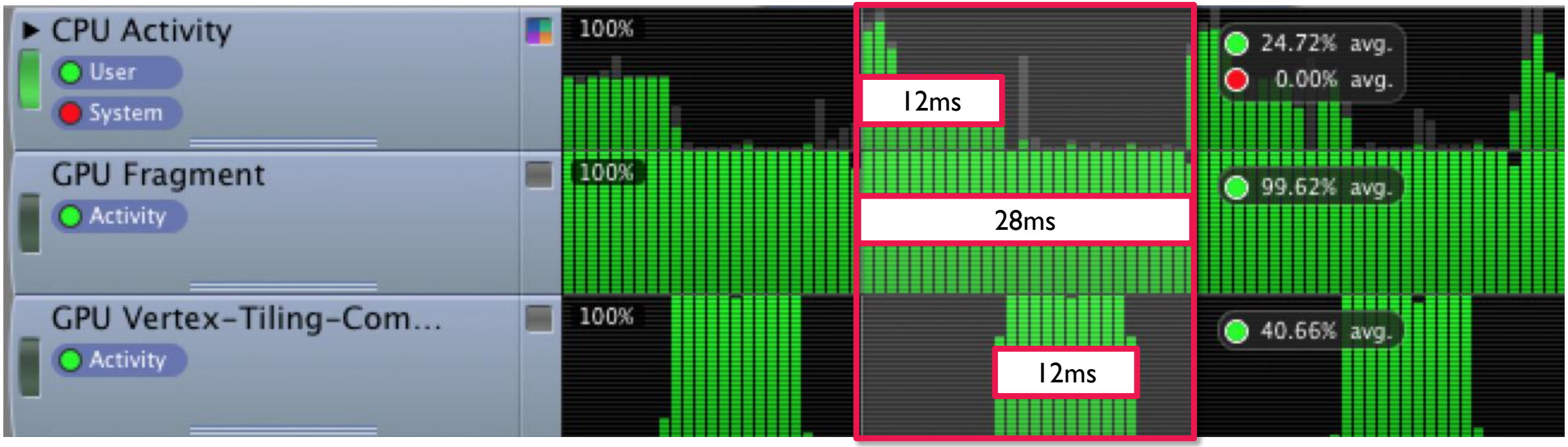
“Epic Citadel” – A Case Study

ARM


0.1s [1000ms]
CPU Activity
User
System
100%
24.28% avg.
0.00% avg.
100%
24.21% avg.
0.00% avg.
CPU Activity → User 24%
GPU Fragment
Activity
100%
99.51% avg.
GPU Fragment → Activity 99%
GPU Vertex-Tiling-Com...
Activity
100%
43.64% avg.
GPU Vertex → Activity 44%
Mali Job Manager Cycles
GPU cycles
JS0 cycles
JS1 cycles
JS2 cycles
15 M
448,148,797
447,561,699
195,511,391
0
Mali Arithmetic Pipe
A instructions
1.5 M
105,103,042
Mali Core Cycles
Fragment cycles
Tripipe cycles
10 M
447,293,472
444,408,368
Mali Load/Store Pipe
LS instruction issues
7 M
407,103,125
194,919,247
[idle]
[kernel]
[com.epicgames.EpicCitadel #1932]
[surfaceflinger #129]
[mediaserver #132]
[adbd #139]
[system_server #465]
[com.android.vending #1109]
[com.android.systemui #538]

The Application is GPU bound

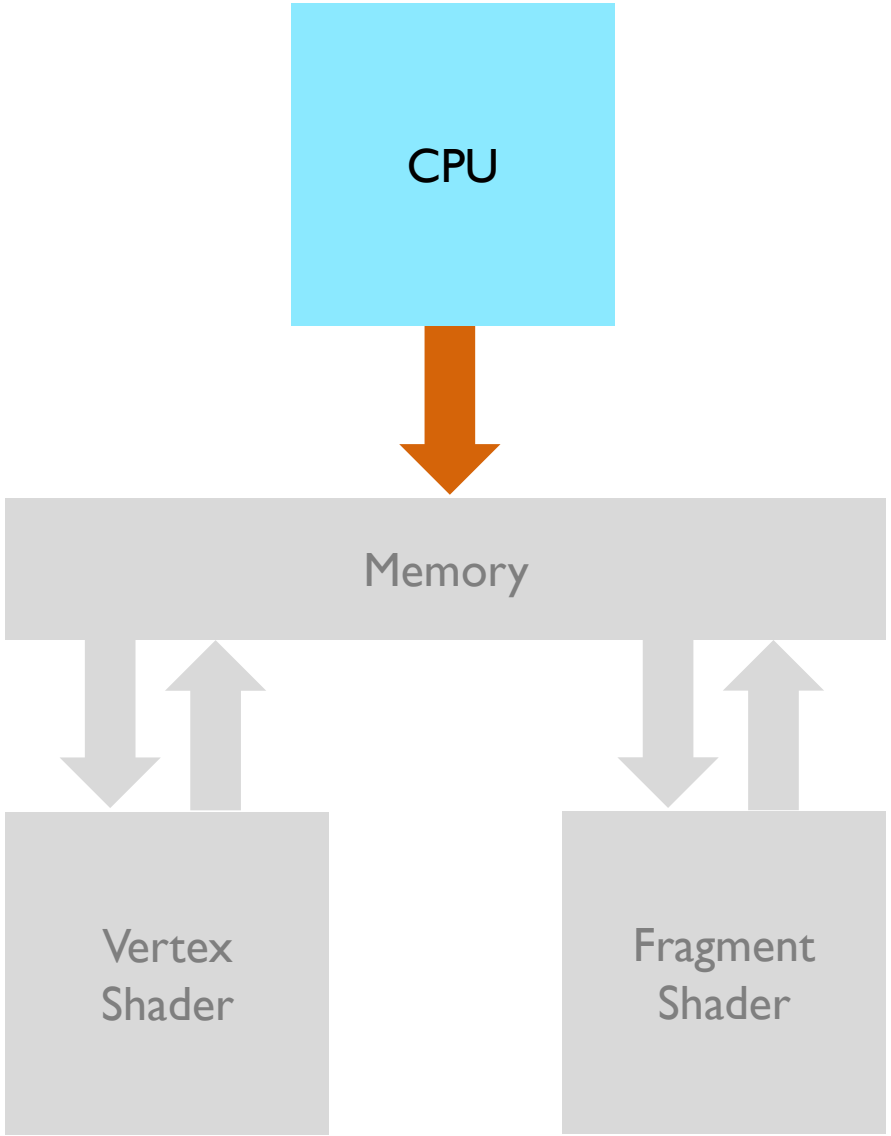
The CPU has to wait until the fragment processing has finished



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

ARM

CPU Bound

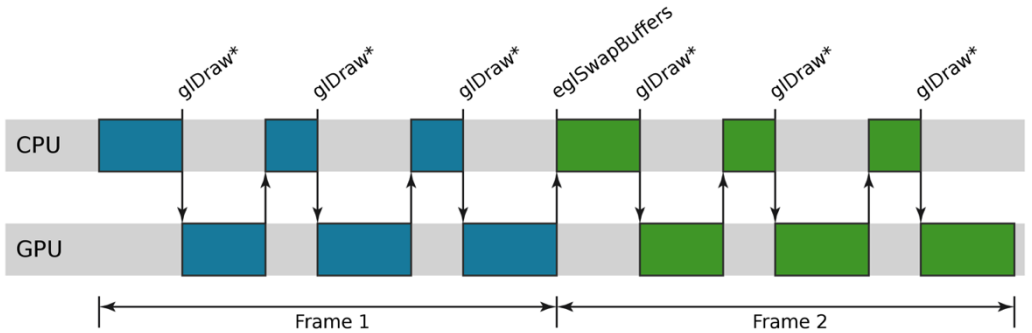


ARM

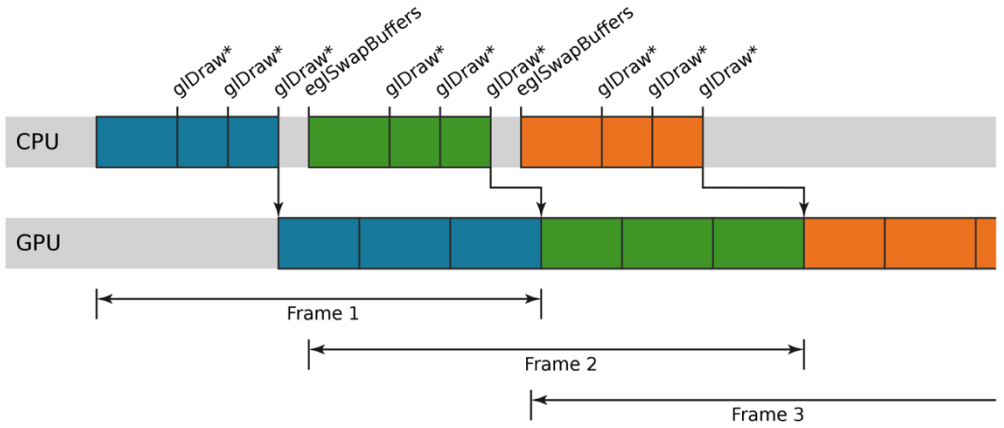
CPU Bound

- 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.)

Synchronous Rendering



Deferred Rendering



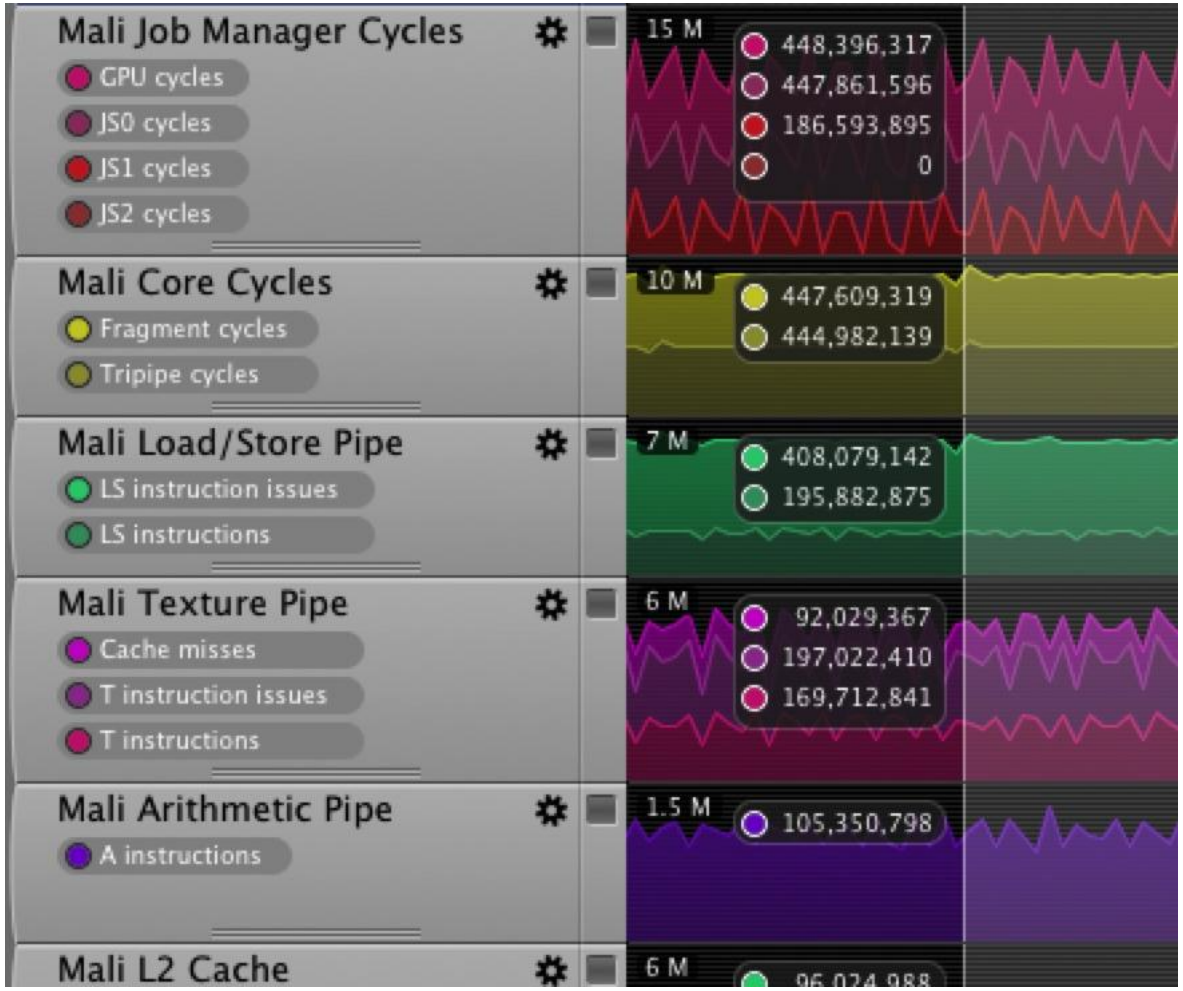
ARM

GPU Activity Analysis

ARM

ARM® Mali™ GPU Hardware Counters

- Over the highlighted time of one second the GPU was active for 448m cycles (*Mali Job Manager Cycles → GPU cycles*)
- With this hardware, the maximum number of cycles is 450m
- 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



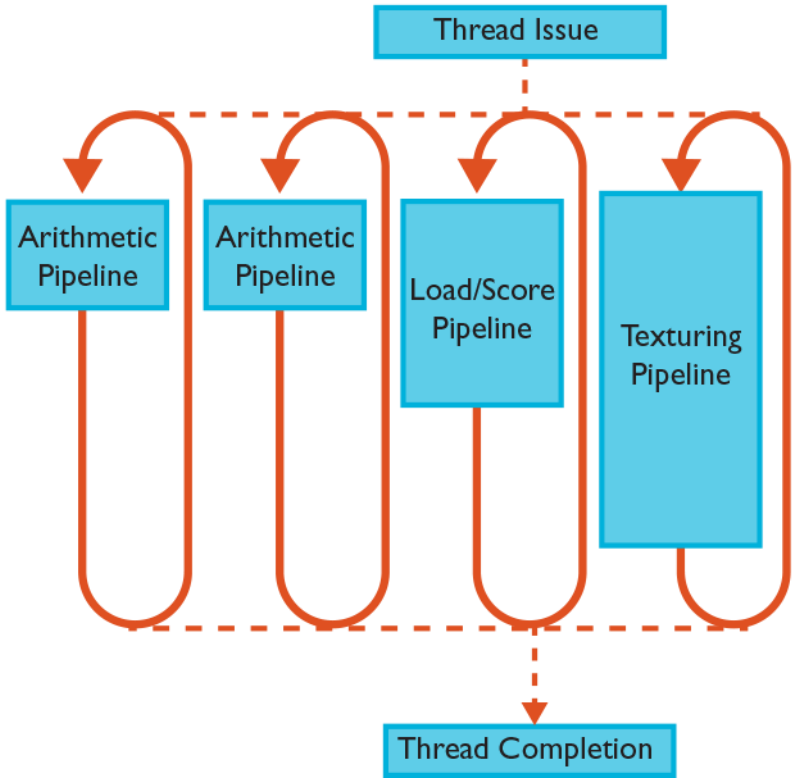
ARM

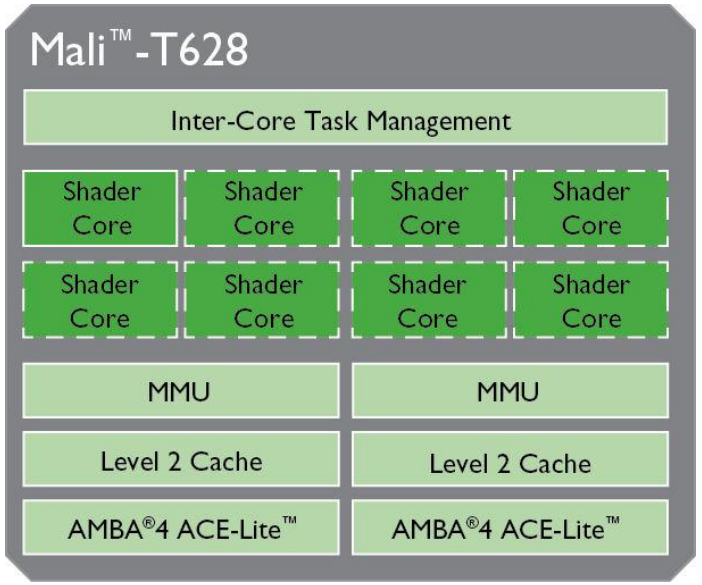
Vertex and Fragment Processing

- 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

ARM

Inspect the Tripipe Counters

Reduce the load on the L/S pipeline

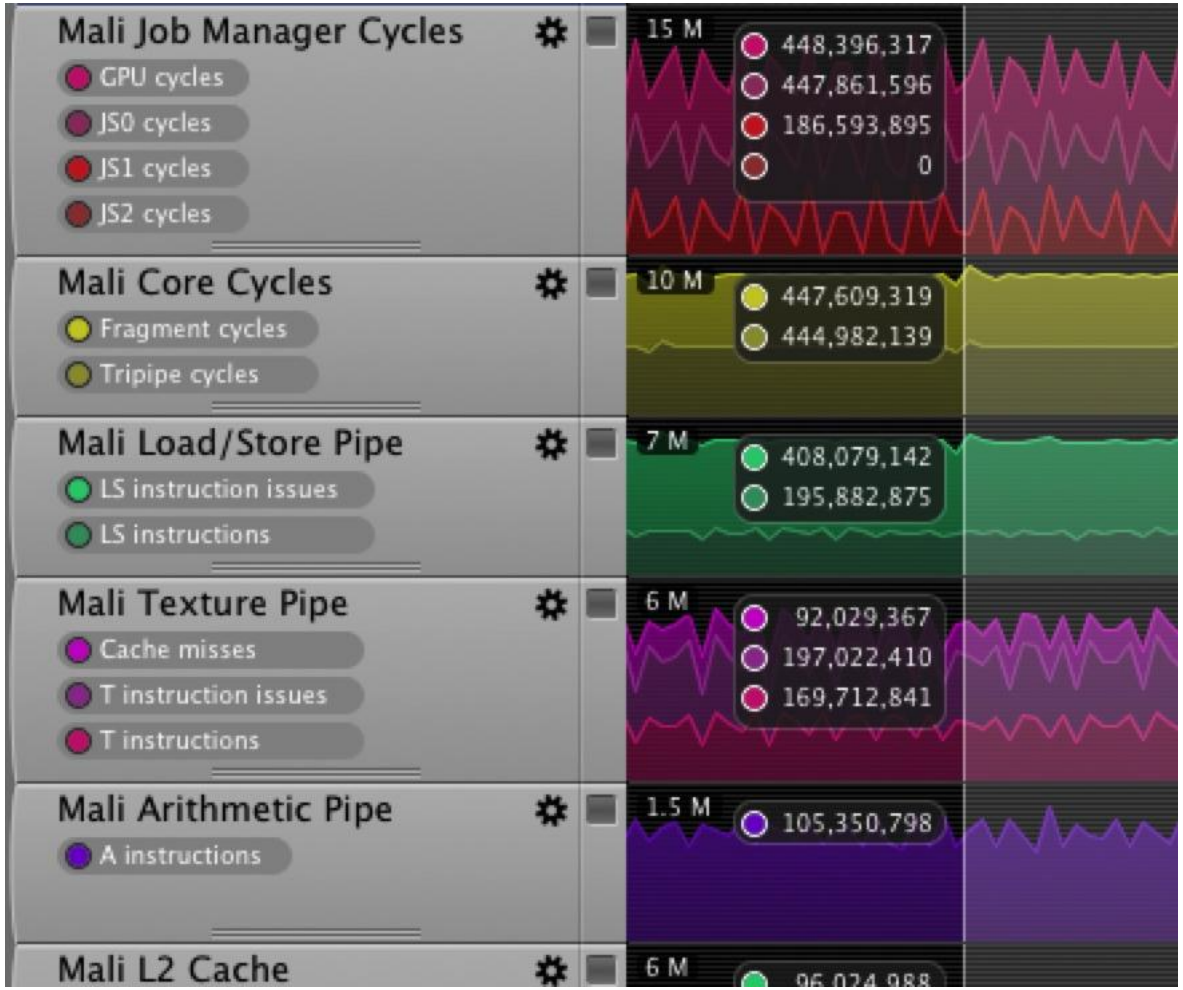
GPU Cycles 448m

Tripipe Cycles 444m

Load & Store 408m

Texture 197m

Arithmetic 105m



ARM

Tripipe Counters

Cycles per instruction metrics

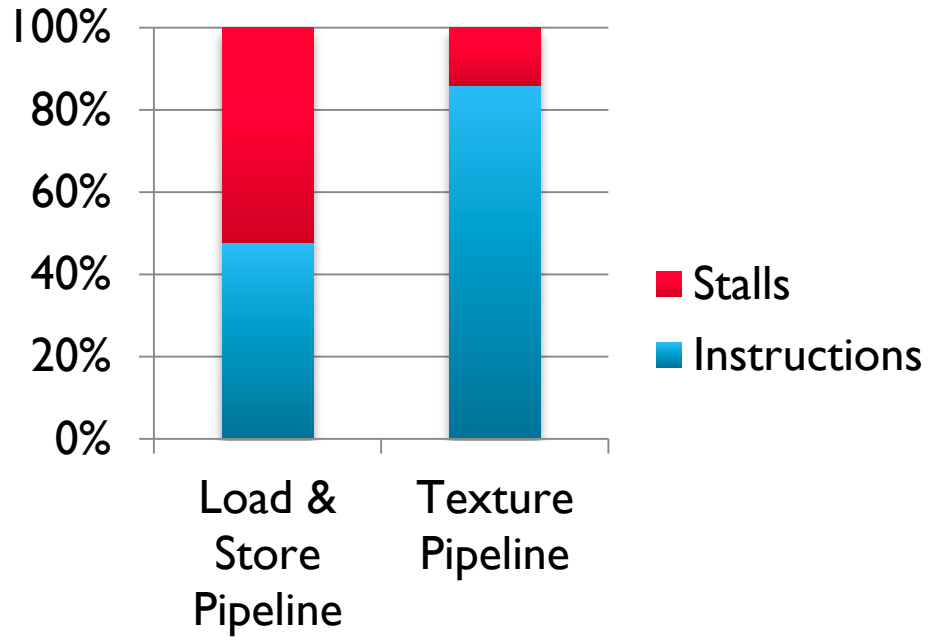
- It's easy to calculate a couple of CPI (cycles per instruction) metrics:

- For the load/store pipeline we have:

  408m (*Mali Load/Store Pipe → LS instruction issues*)
  / **195m** (*Mali Load/Store Pipe → LS instructions*)
  = 2.09 cycles/instruction

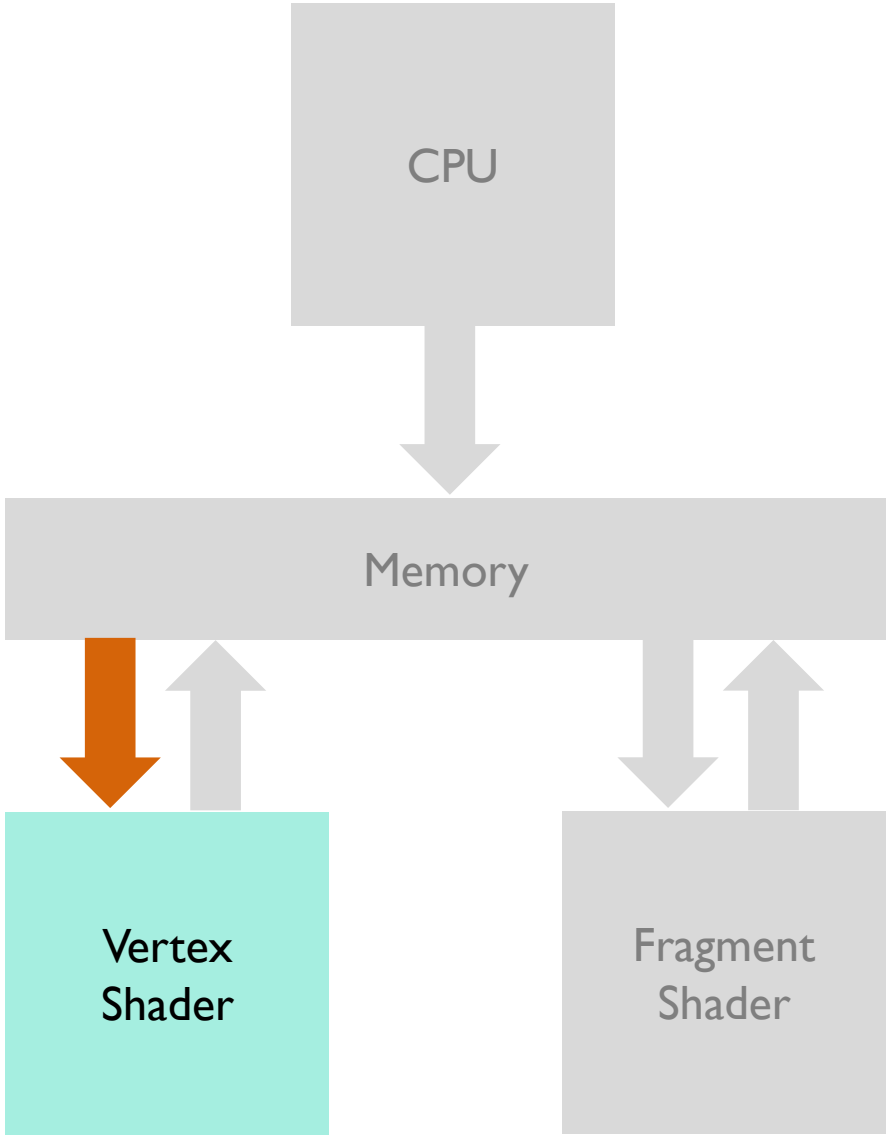
- For the texture pipeline we have:

  197m (*Mali Texture Pipe → T instruction issues*)
  / **169m** (*Mali Texture Pipe → T instructions*)
  = 1.16 cycles/instruction



ARM

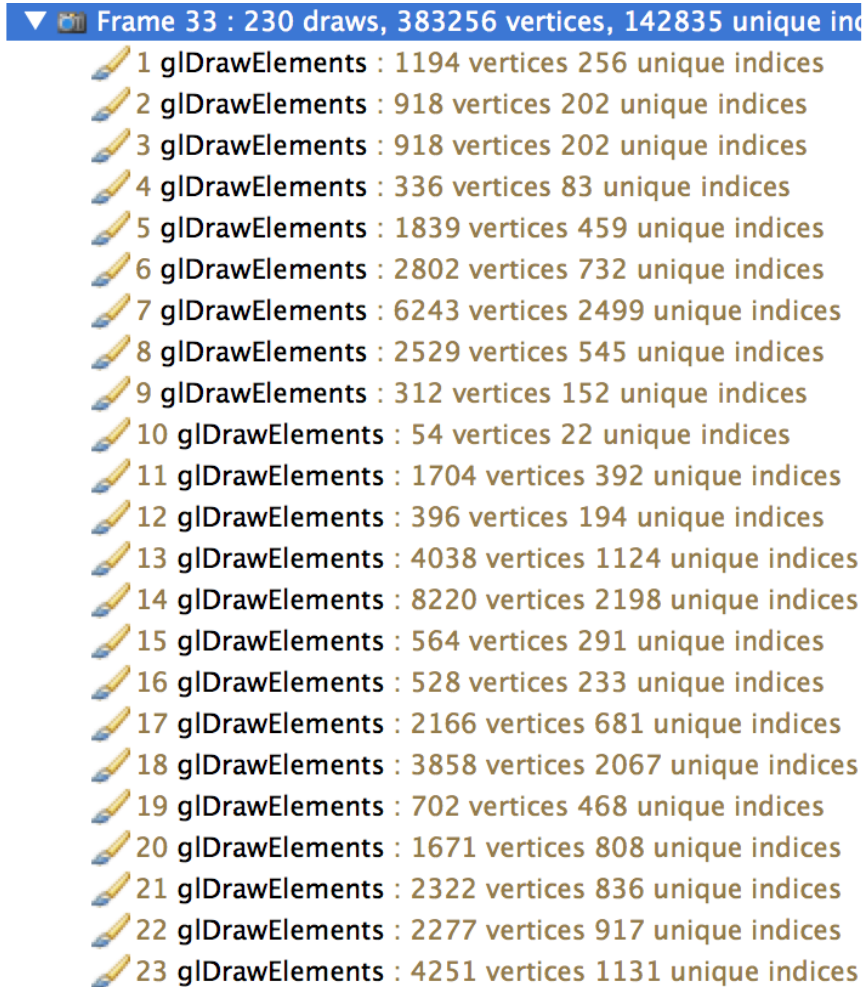
Vertex Bound



ARM

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



ARM

Vertex Count and Shader Optimizations

Identify the top heavyweight vertex shaders

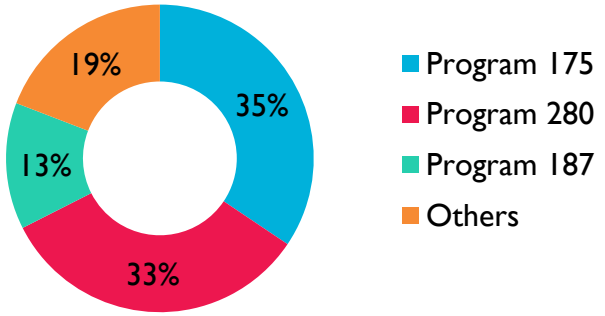
citadel13c.mgd | Shader 176

```
precision highp float;
float Square(float A)
{
return A * A;
}
vec2  Square( vec2  A)
{
return A * A;
}
vec3  Square( vec3  A)
{
return A * A;
}
vec4  Square( vec4  A)
{
return A * A;
}
float EncodeLightAttenuation(float InColor)
{
return sqrt(InColor);
}
vec4  EncodeLightAttenuation( vec4  InColor)
{
return sqrt(InColor);
}
uniform mat3 TextureTransform ;
void DummyPreprocessorFixFunction();
uniform mat4 LocalToWorld ;
uniform mat3 LocalToWorldRotation ;
uniform mat4 ViewProjection ;
uniform mat4 LocalToProjection ;
uniform vec4 FadeColorAndAmount ;
```

Assets | Vertex Shaders | Fragment Shaders | Textures

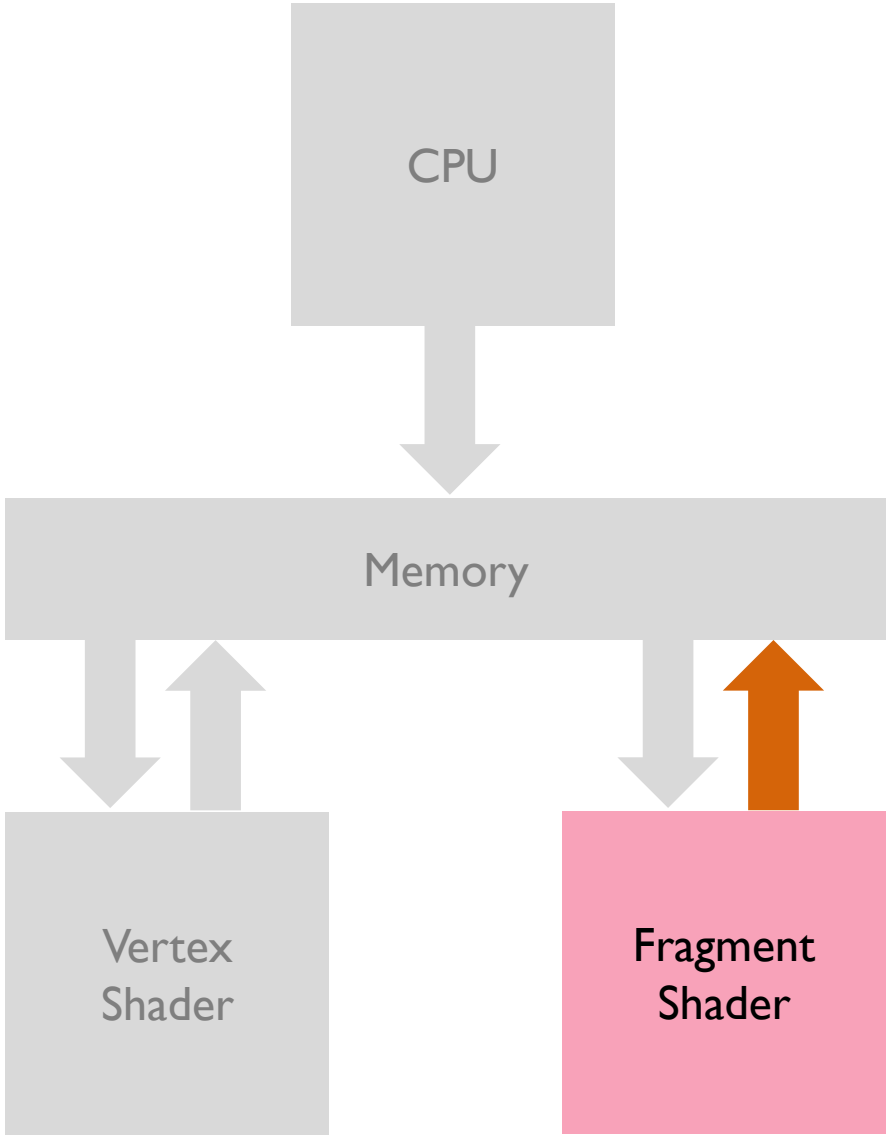
Program	Name	Instruction	Shortest pa	Longest pa	Instances r	Total cy
175	Shader 176	22	22	22	148500	3267000
280	Shader 281	39	39	39	80595	3143205
187	Shader 188	48	48	48	26328	1263744
181	Shader 182	61	61	61	11487	700707
211	Shader 212	22	22	22	14646	322212
289	Shader 290	55	55	55	5484	301620
298	Shader 299	38	38	38	5259	199842
208	Shader 209	22	22	22	4257	93654
214	Shader 215	61	61	61	1110	67710
73	Shader 74	20	20	20	2880	57600
262	Shader 263	36	36	36	1152	41472

Vertex Cycles Per Program



ARM

Fragment Bound



ARM

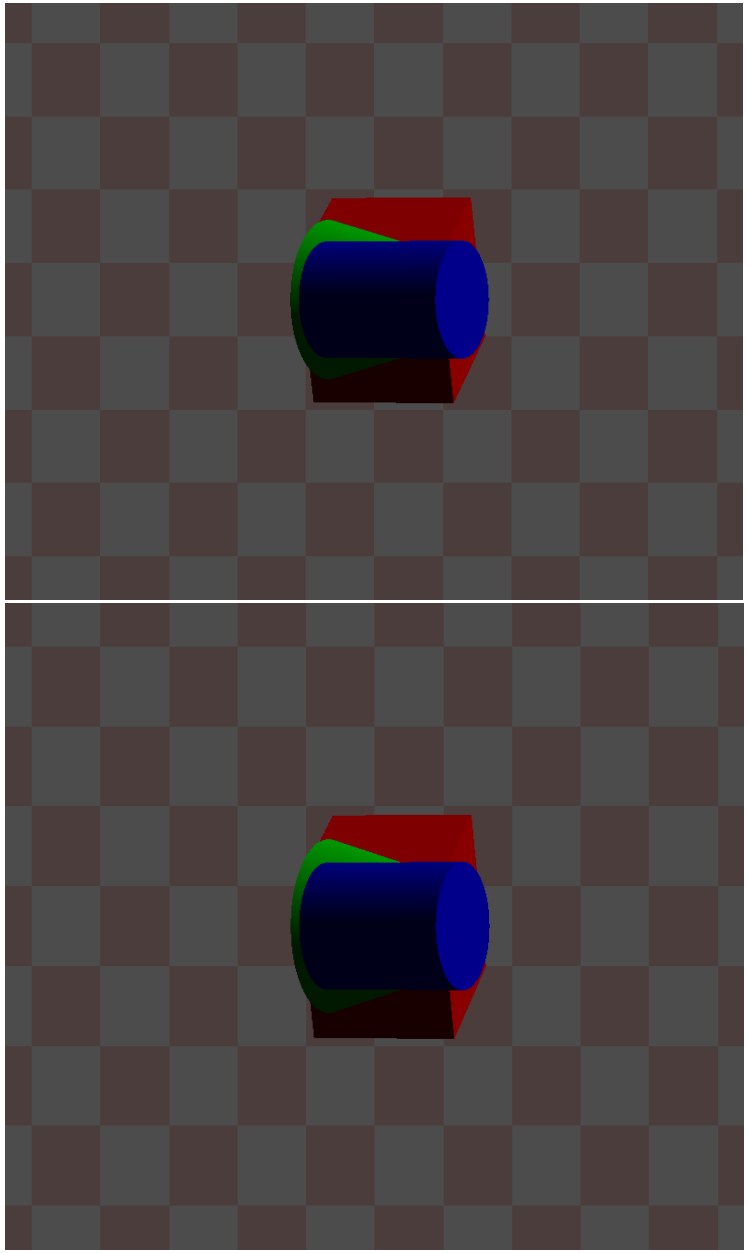
Fragment Bound

- 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

ARM

Overdraw

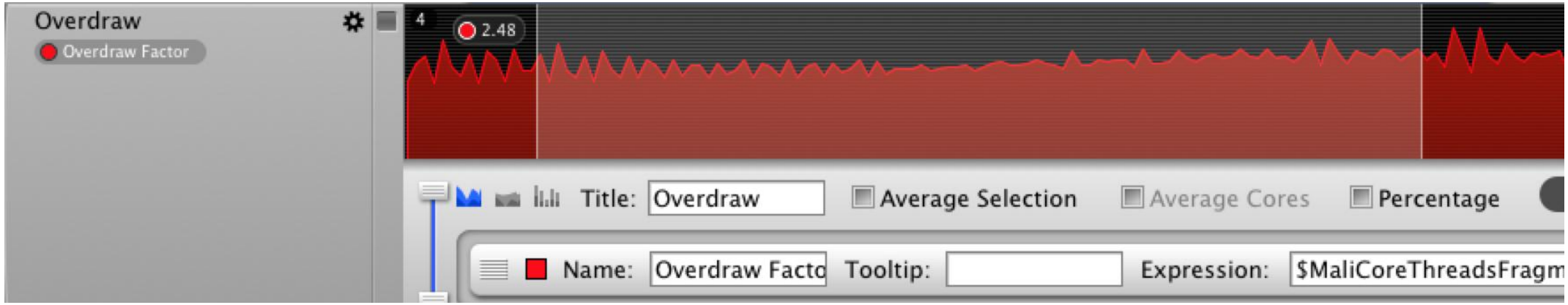
- This is when you draw to each pixel on the screen more than once
  - Drawing your objects front to back instead of back to front reduces overdraw
- The transparent objects must be drawn back to front for a correct ordering
  - Limiting the amount of transparency in the scene can help
- For OpenGL® 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:

90.7m (*Mali Core Threads → Fragment threads*)
/ 143K (*Mali Fragment Tasks → Tiles rendered*) x 256
= 2.48 threads/pixel



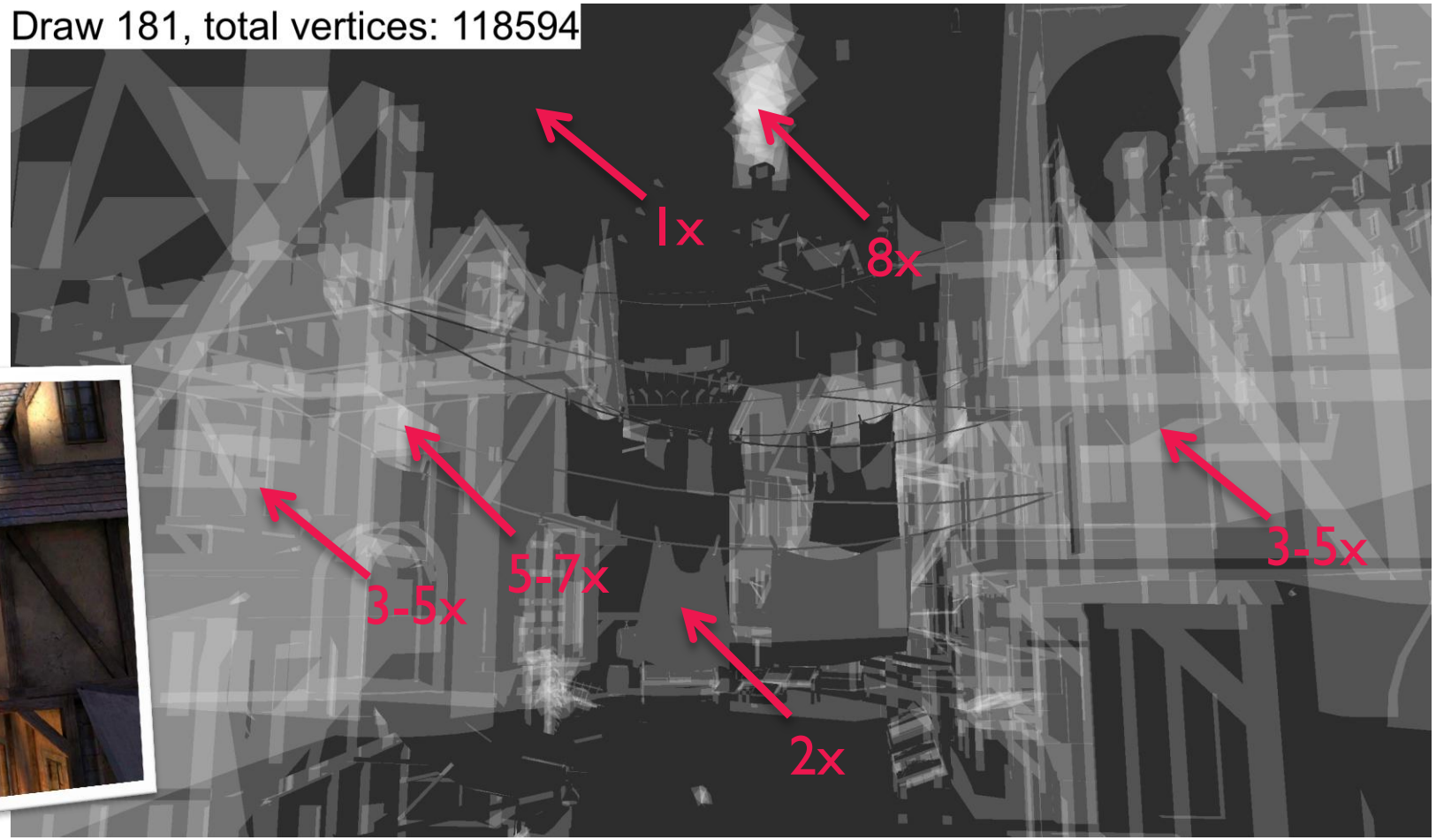
ARM

Frame Capture

ARM

Frame Analysis

Check the overdraw factor



ARM

Shader Map and Fragment Count

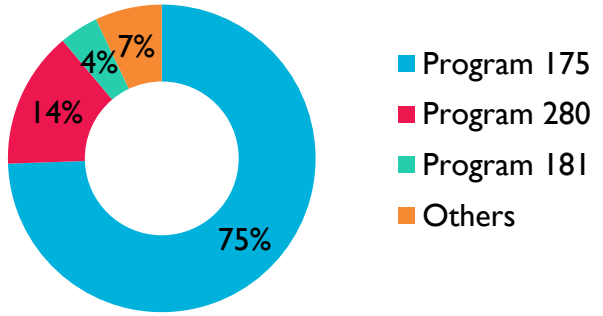
Identify the top heavyweight fragment shaders

Assets | Vertex Shaders | Fragment Shaders | Textures

Program	Name	Instructions	Shortest	Longest	Instances	Total cycles
175	Shader 177	5	5	5	7537773	37688865
280	Shader 282	5	5	5	1459254	7296270
181	Shader 183	5	5	5	415710	2078550
187	Shader 189	6	6	6	197329	1183974
73	Shader 75	4	4	4	279555	1118220
382	Shader 384	8	8	8	129913	1039304
289	Shader 291	6	6	6	16856	101136
208	Shader 210	7	3	6	7975	39875
262	Shader 264	5	5	5	6025	30125
400	Shader 402	5	5	5	914	4570

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

Fragment Count Per Program



ARM

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!

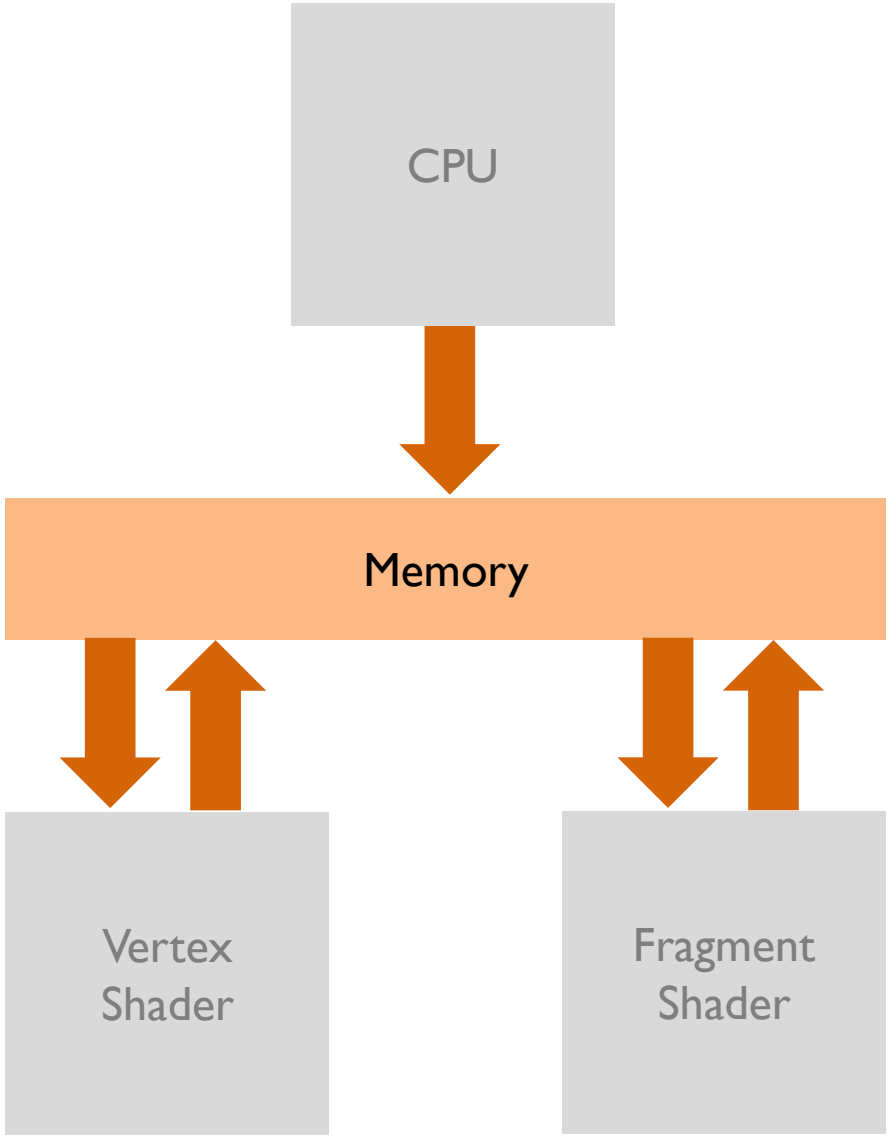
  http://malideveloper.arm.com/develop-for-mali/tools/analysis-debug/mali-gpu-offline-shader-compiler/

citadel13c.mgd | Shader 177

```
uniform sampler2D TextureBase ;
varying highp vec4 UVBase ;
uniform sampler2D TextureLightmap ;
varying highp vec2 UVLightmap ;
varying lowp vec4 GlobalEffectColorAndAmount ;
void main()
{
lowp vec3 DebugColor;
highp vec2 FinalBaseUV = UVBase.xy;
highp vec2 TransformedFinalBaseUV = UVBase.zw;
highp vec2 BaseTextureCoord;
BaseTextureCoord = FinalBaseUV;
lowp vec4 BaseColor =  texture2D(TextureBase, BaseTextureCoord, -0.50 ) ;
lowp float AlphaVal = BaseColor.a;
{
}
ALPHAKILL( AlphaVal )
BaseColor.xyz =  BaseColor.xyz ;
lowp vec4 PolyColor = vec4(BaseColor.xyz, AlphaVal);
lowp  vec3  EnvironmentSpecular =  vec3 (0, 0, 0);
lowp vec3 TotalDiffuseLight = vec3( 0.0, 0.0, 0.0 );
PolyColor.rgb += EnvironmentSpecular;
lowp vec3 PreSpecularPolyColor = PolyColor.rgb;
{
lowp vec3 LightmapColor = texture2D( TextureLightmap, UVLightmap ).rgb;
LightmapColor =  LightmapColor ;
PolyColor.rgb = PolyColor.rgb * LightmapColor;
PolyColor.rgb += PolyColor.rgb;
}
PolyColor.xyz = (PolyColor.xyz * GlobalEffectColorAndAmount.w) + GlobalEffectColorAndAmou
;
PolyColor.xyz =  PolyColor.xyz ;
gl_FragColor =  PolyColor ;
```


ARM

Bandwidth Bound



ARM

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

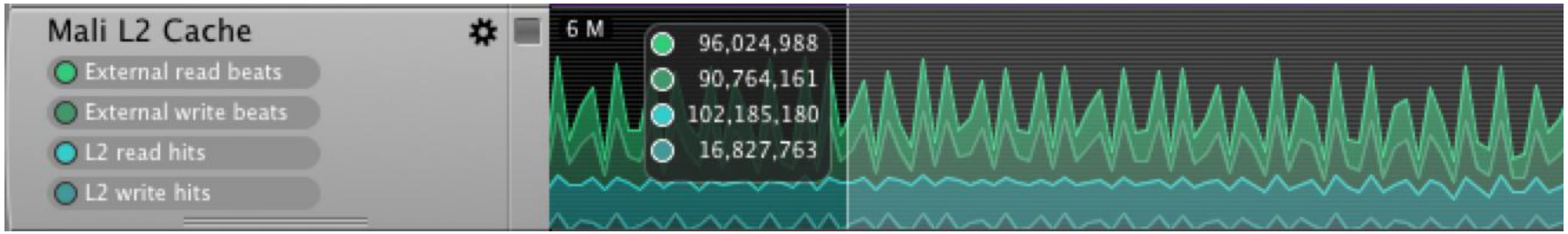
- The application is not bandwidth bound as it performs, over a period of one second:

(96m (*Mali L2 Cache → External read beats*) +

90.7m (*Mali L2 Cache → External write beats*)) x 16

~= 2.9 GB/s

- Since bandwidth usage is related to energy consumption it's always worth optimizing it



ARM

Bandwidth Bound

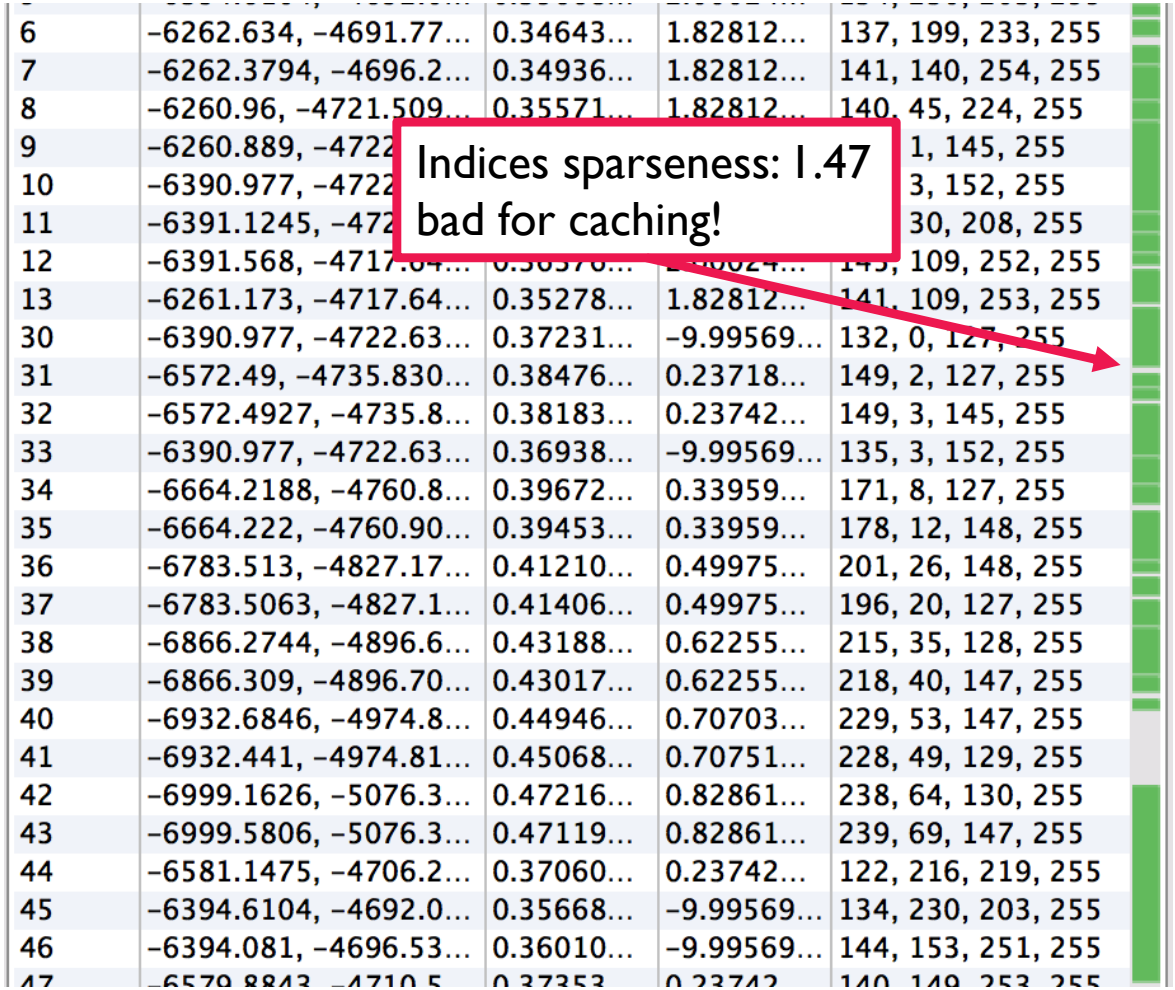
Vertices

- 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.



ARM

Texture Compression Formats

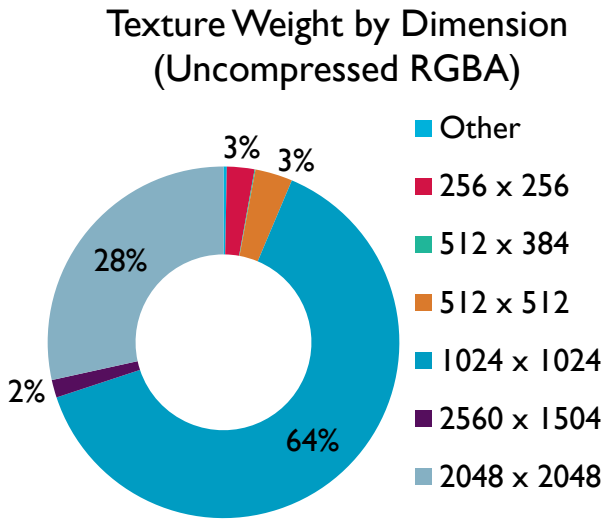
- ETC – ARM Mali-400 GPU
  - 4bpp
  - RGB No alpha channel
- ETC2 – ARM Mali-T604 GPU
  - 4bpp
  - Backward compatible
  - RGB also handles alpha & punch through
- ASTC – ARM Mali-T624 GPU and beyond
  - 0.8bpp to 8bpp
  - Supports RGB, RGB alpha, luminance, luminance alpha, normal maps
  - Also supports HDR
  - Also supports 3D textures

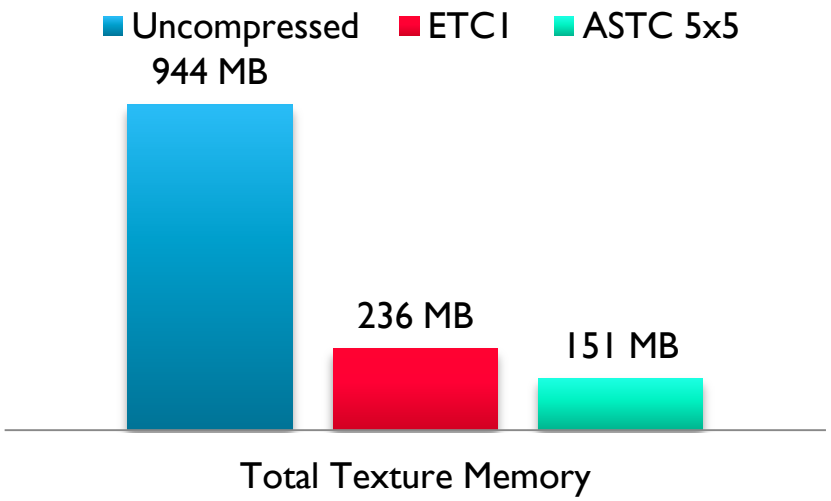
ARM

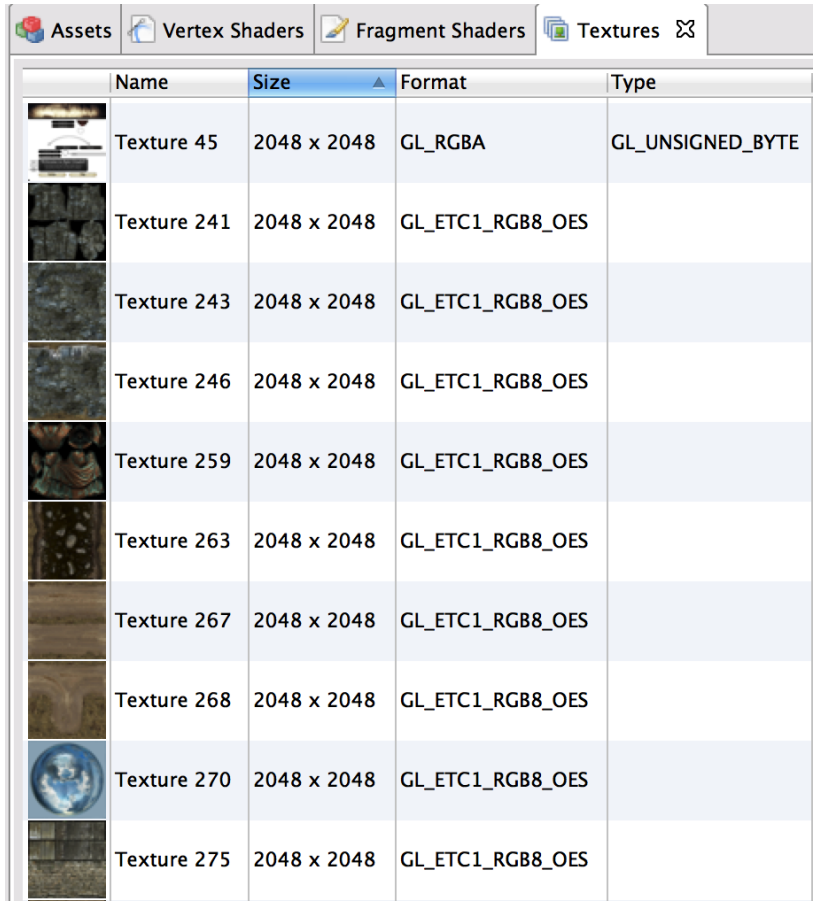
Textures

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







ARM

- 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

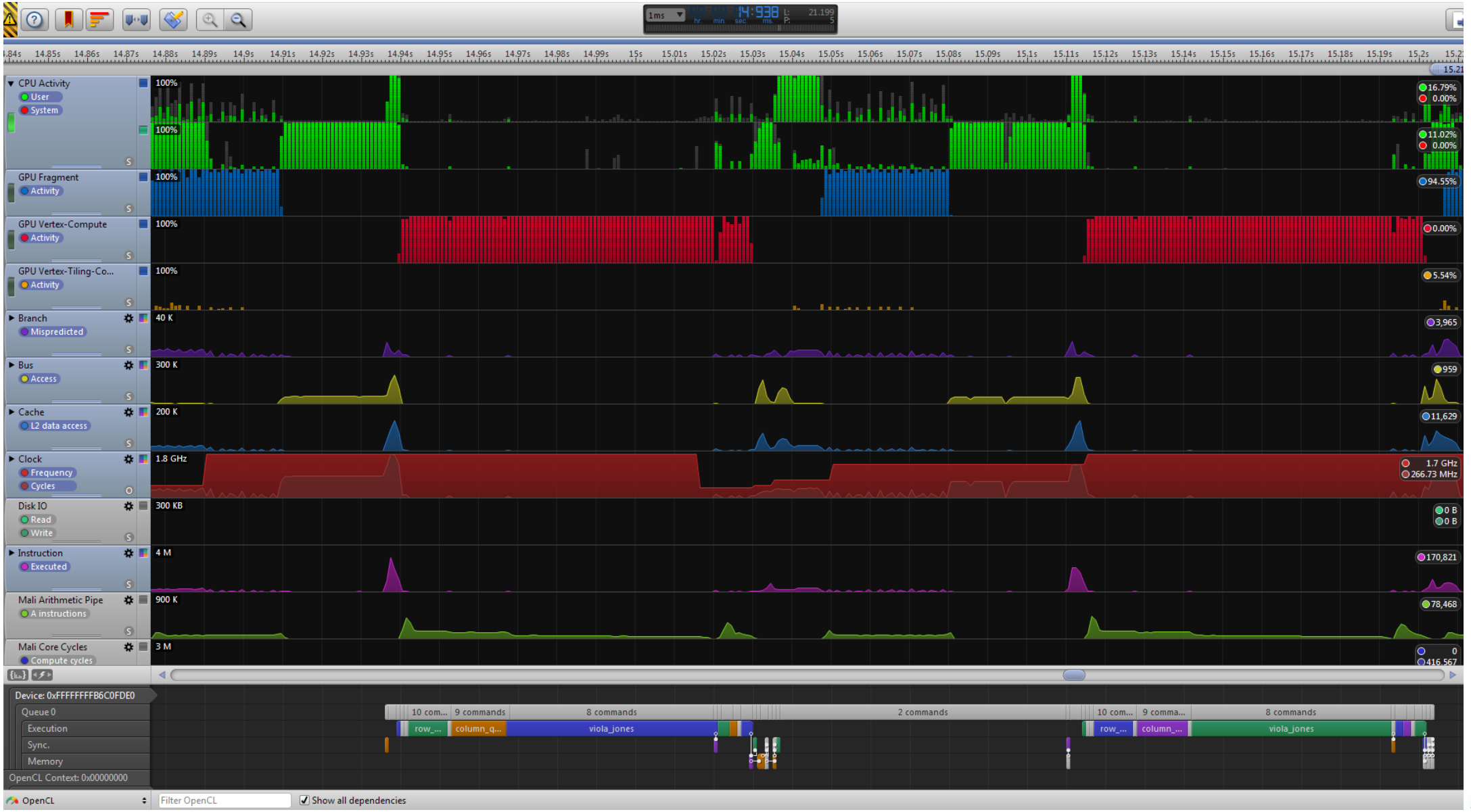
malidevelopers@arm.com

ARM

Mali tools update

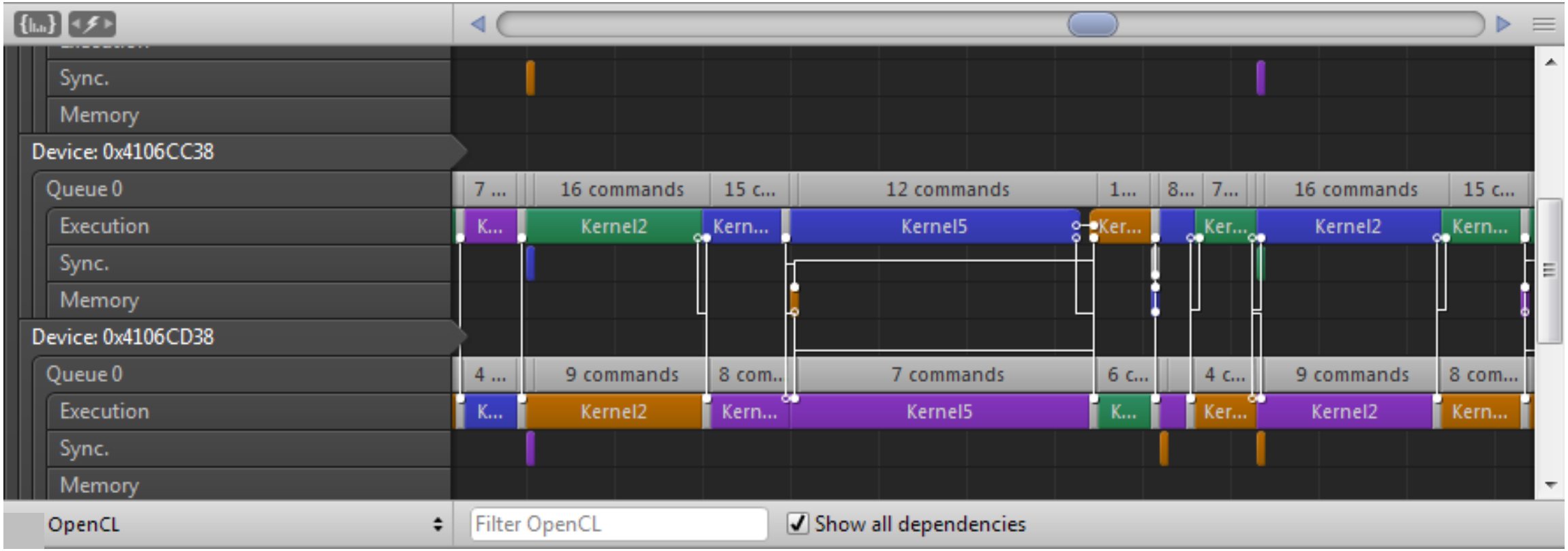
ARM

Streamline: OpenCL Timeline



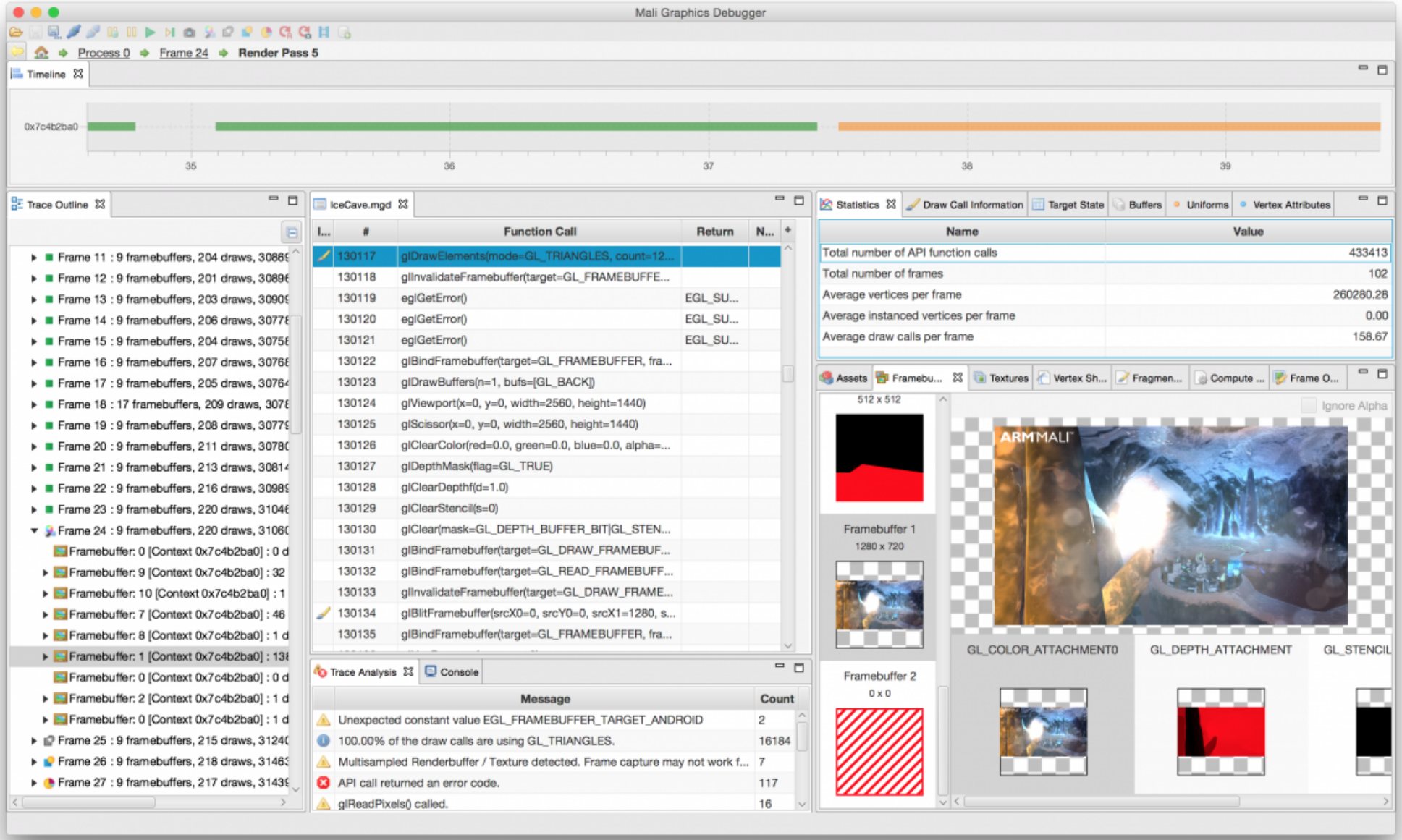
Streamline: OpenCL Timeli

Streamline: OpenCL Timeline

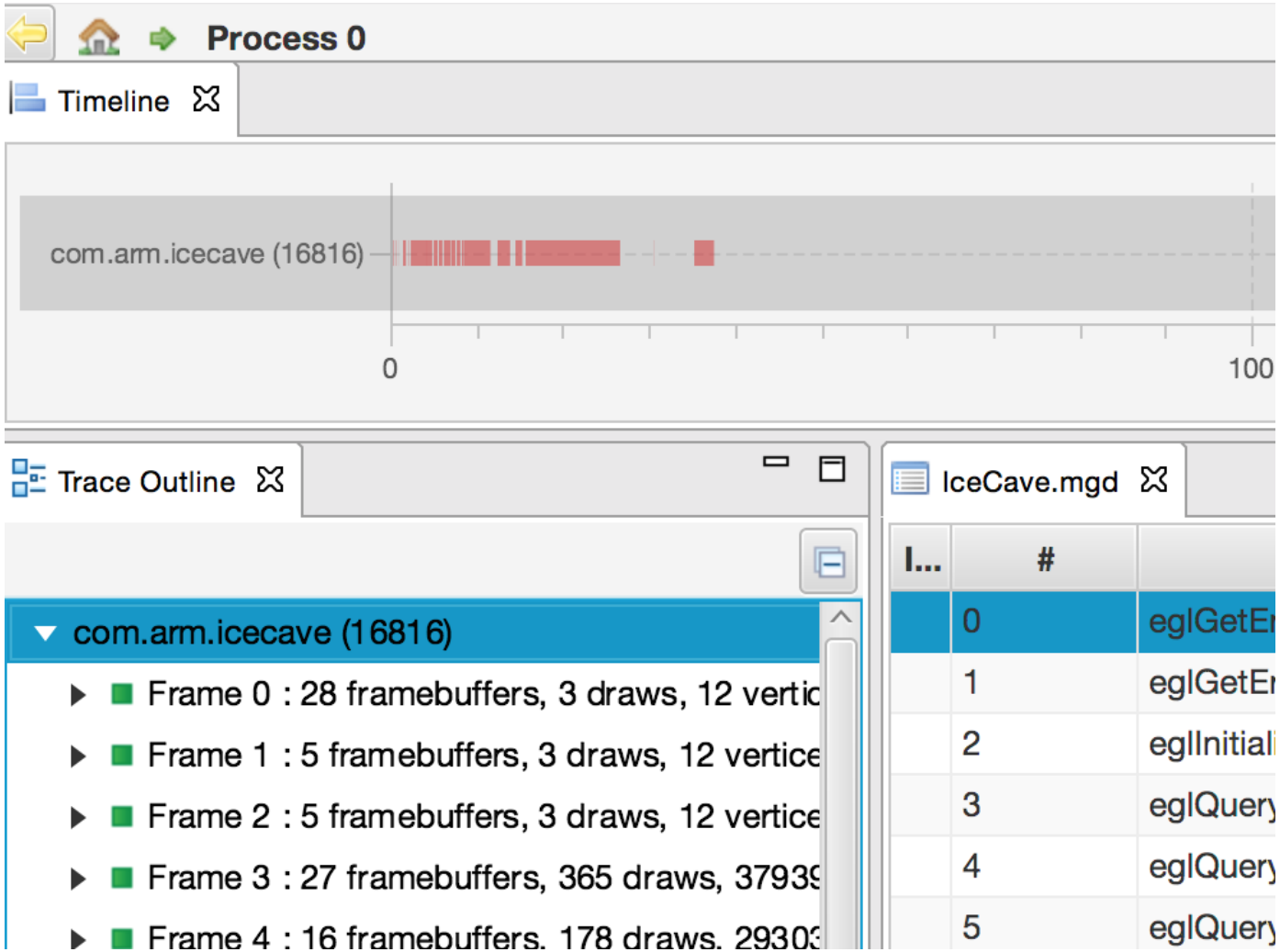


ARM

Mali Graphics Debugger 3.0: Overview

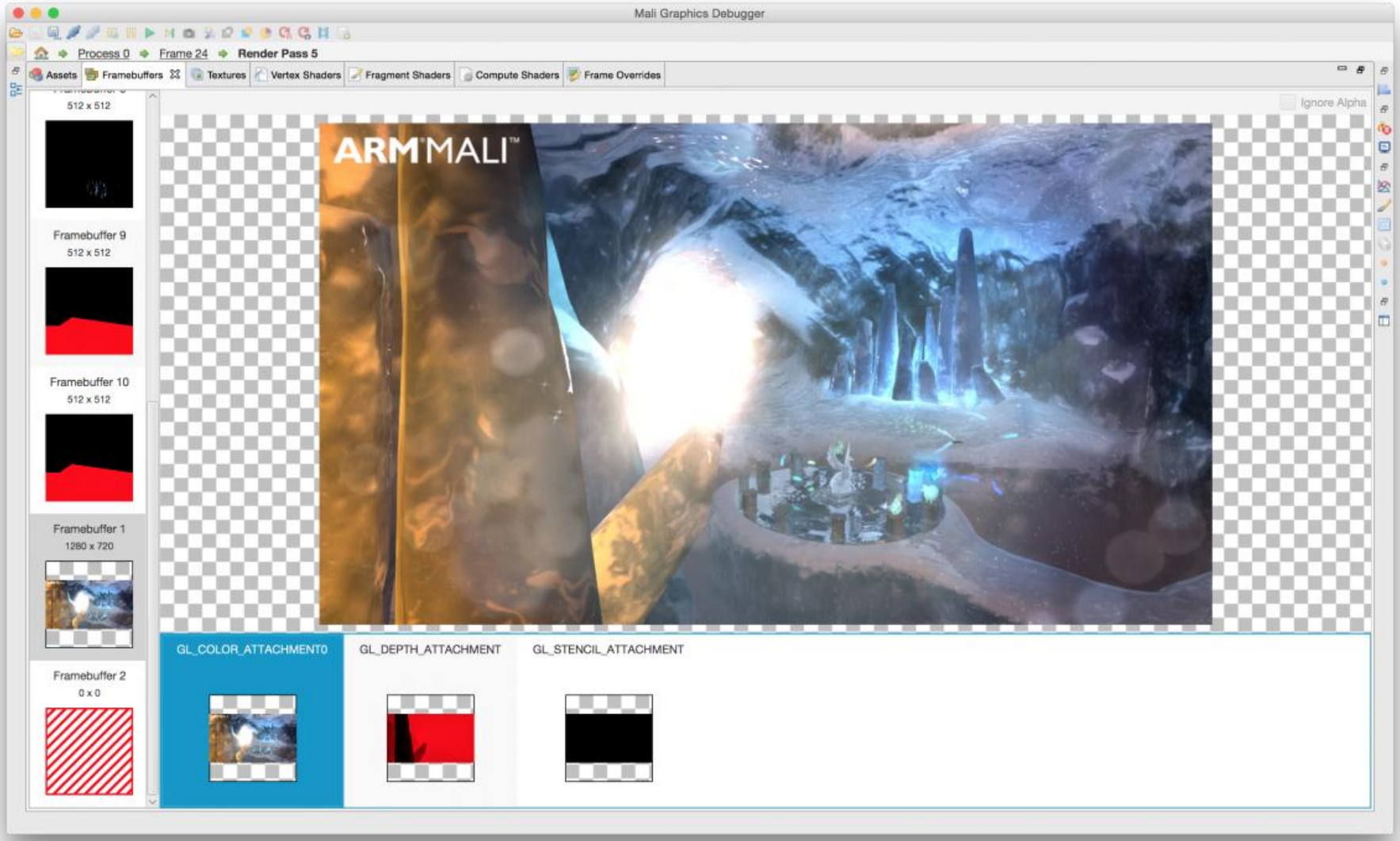


Mali Graphics Debugger 3.0: Timeline



ARM

Mali Graphics Debugger 3.0: Frame Capture



ARM