Performance Analysis and Debug Tools for Mobile Games

Lorenzo Dal Col — *Senior Software Engineer, ARM*
Ronan Synnott — *Select Field Applications Engineer, ARM*

Agenda

1. Introduction to performance analysis with ARM® DS-5 and Streamline Performance Analyzer
2. Software Profiling
   - Find hotspots, system glitches, critical conditions at a glance
   - Power measurements
3. GPU Profiling
   - Using the ARM® Mali™ GPU hardware counters to find the bottleneck
4. Debugging with Mali Graphics Debugger
   - Overdraw and frame analysis
5. Q & A

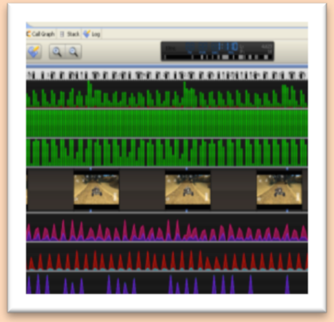
Importance of Analysis & Debug

- 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

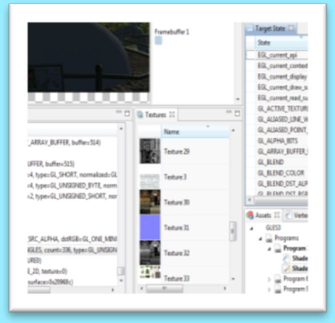
ARM

Performance Analysis & Debug



ARM® DS-5 Streamline Performance Analyzer

- System-wide performance analysis
- Combined ARM Cortex® Processors and Mali™ GPU visibility
- Optimize for performance & power across the system



ARM Mali Graphics Debugger

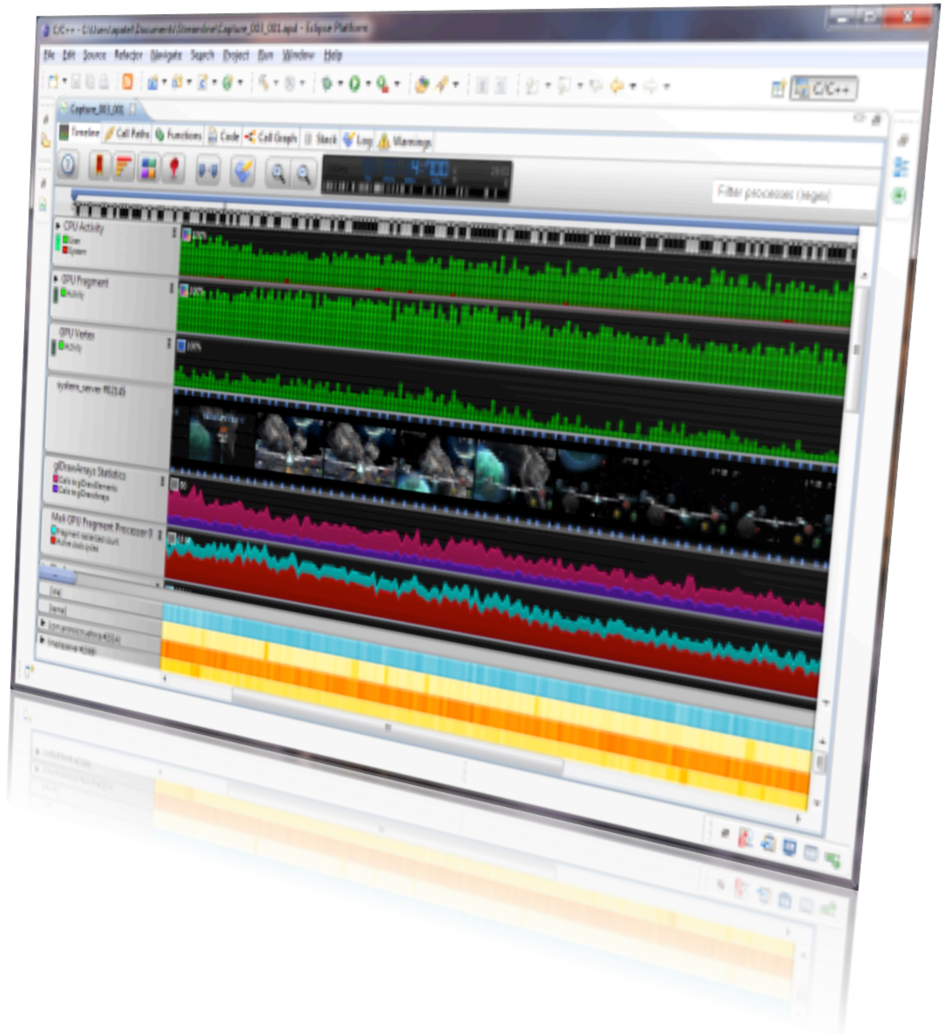
- API Trace & Debug Tool
- Understand graphics and compute issues at the API level
- Debug and improve performance at frame level
- Support for OpenGL® ES 1,1, 2.0, 3.0 and OpenCL™ 1.1

Offline Compilers

- Understand complexity of GLSL shaders and CL kernels
- Support for ARM Mali-4xx and Mali-T6xx GPU families

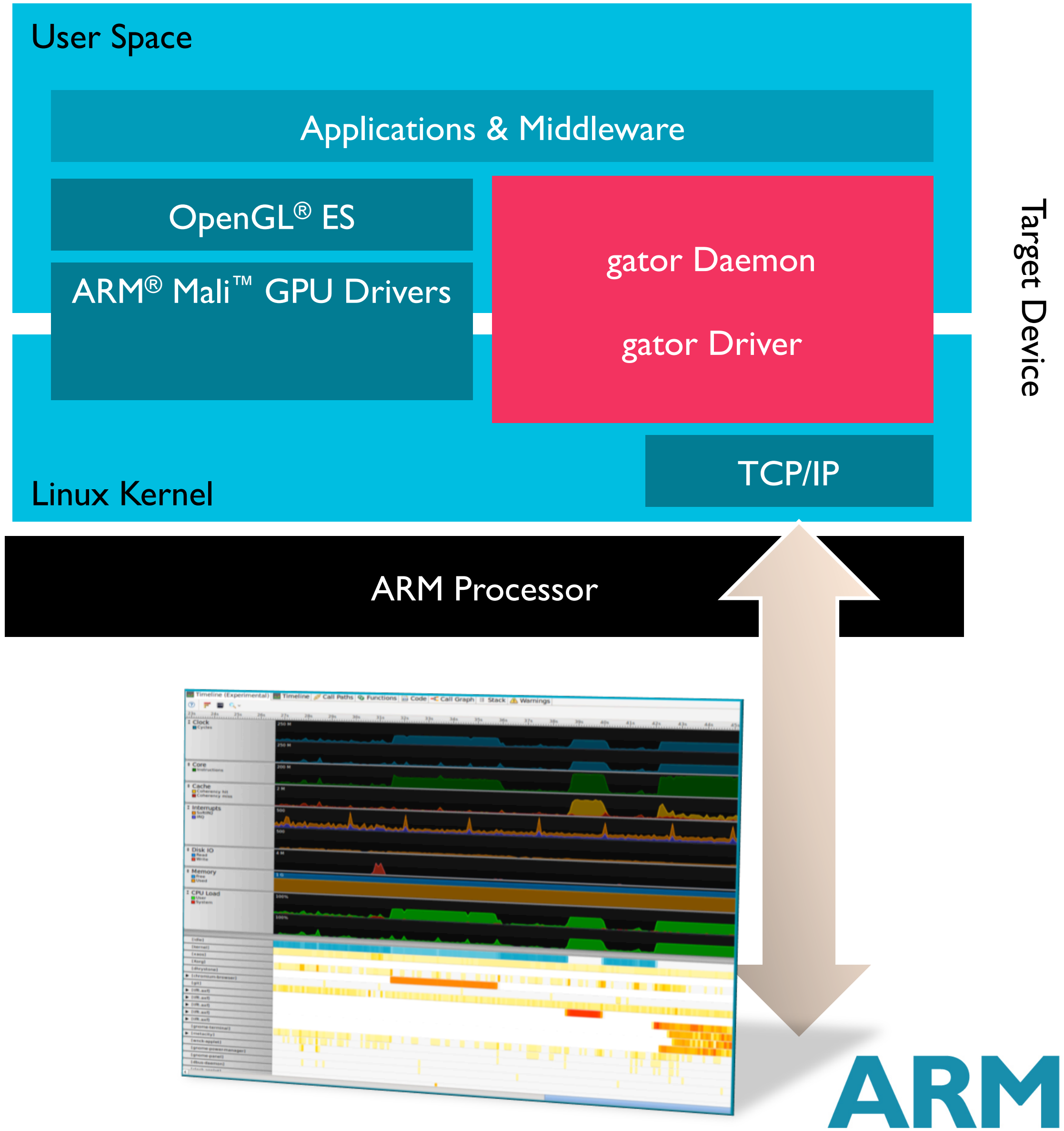
ARM® DS-5 Streamline Performance Analyzer

- System Wide Performance Analysis
  - Simultaneous visibility across ARM Cortex® processors & Mali™ GPUs
  - Support for graphics and GPU Compute performance analysis on Mali-T600 series
  - Timeline profiling of hardware counters for detailed analysis
  - Custom counters
  - Per-core/thread/process granularity
  - Frame buffer capture and display
- Optimize
  - Performance
  - Energy efficiency
  - Across the system



The Basics

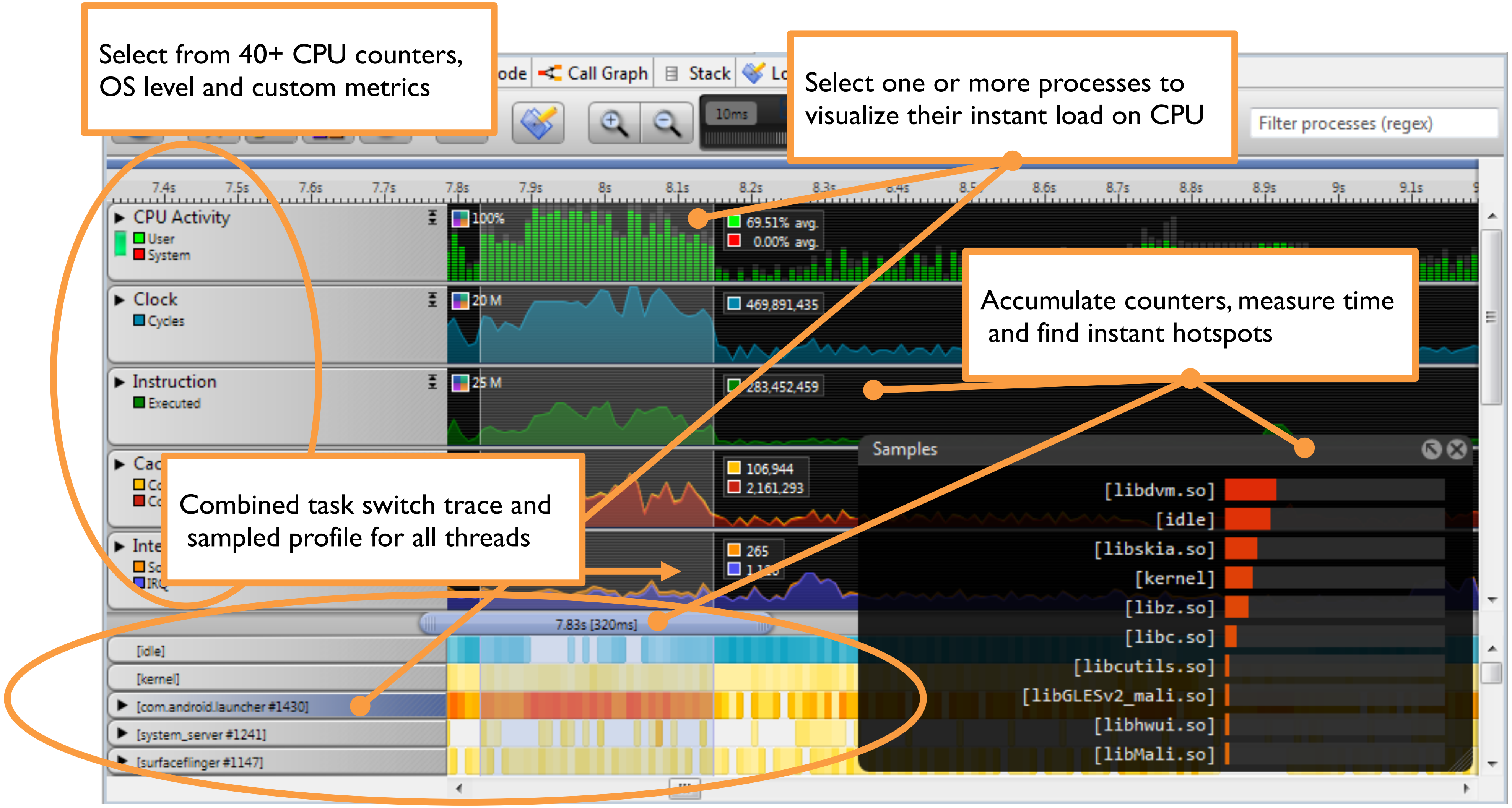
- Software based solution
  - ICE/trace units not required
  - Support for Linux kernel 2.6.32+ on target
  - Eclipse plug-in or command line
- Lightweight sample profiling
  - Time- or event*-based sampling
  - Process to C/C++ source code profiler
  - Low probe effect; <5% typically
- Multiple data sources
  - CPU, GPU and Interconnect hardware counters
  - Software counters and kernel tracepoints
  - User defined counters and instrumented code
  - Power/energy measurements



* Event-based sampling is available on kernels 3.0 or later

Timeline: The Big Picture

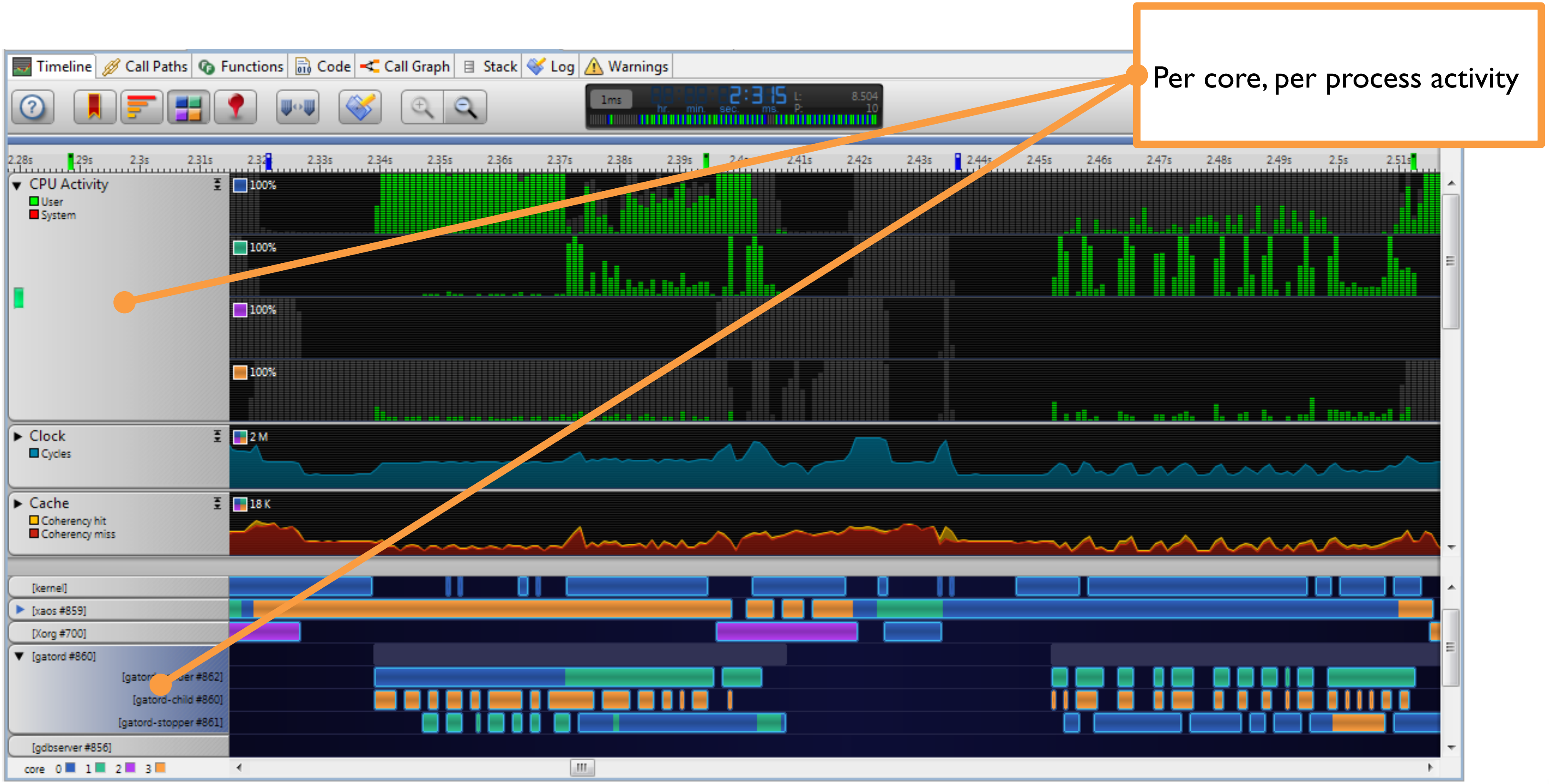
Find hotspots, system glitches, critical conditions at a glance



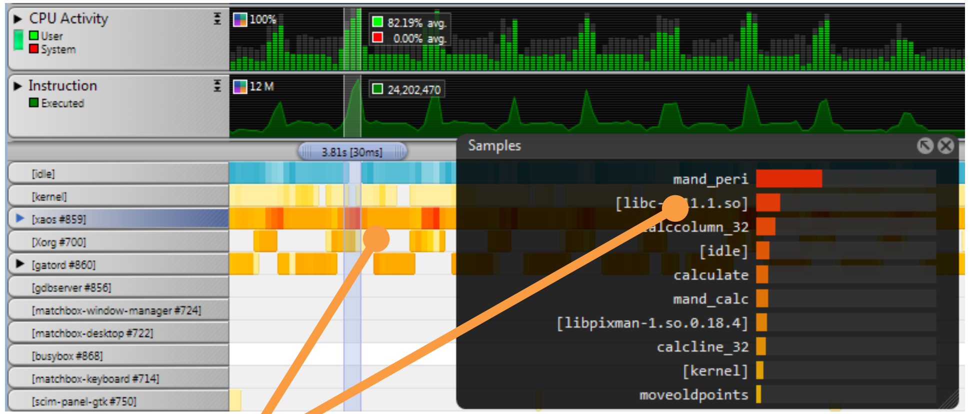
SMP Analysis

Take advantage of multicore SMP platforms

- Visually trace core migration and per-core statistics
- Spot non-optimal thread synchronization and improve parallelism

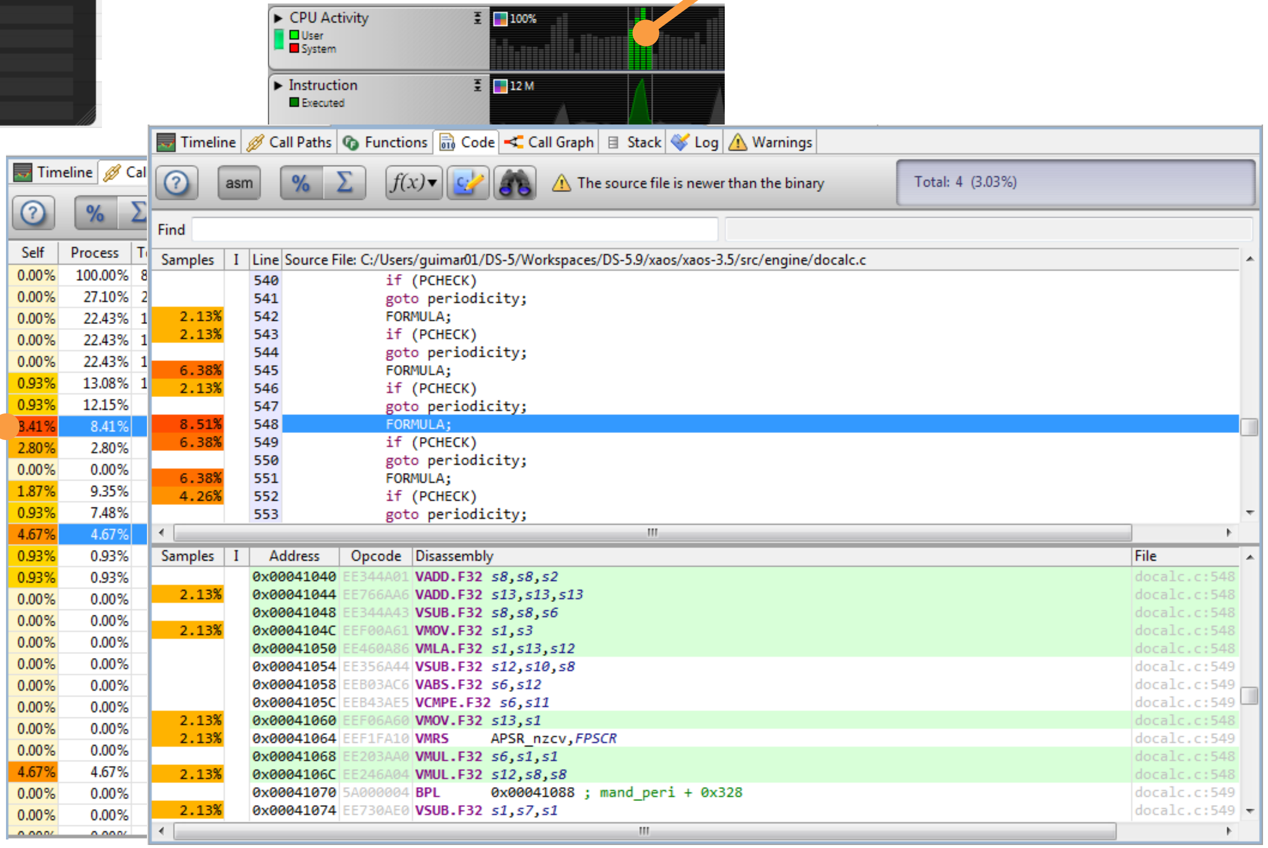


Drilldown Software Profiling



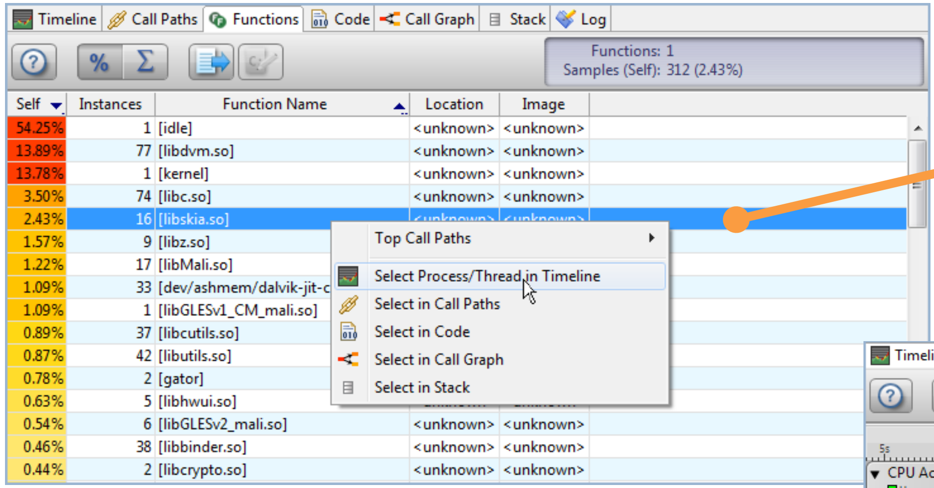
Filter timeline data to generate focused software profile reports

Quickly identify instant hotspots



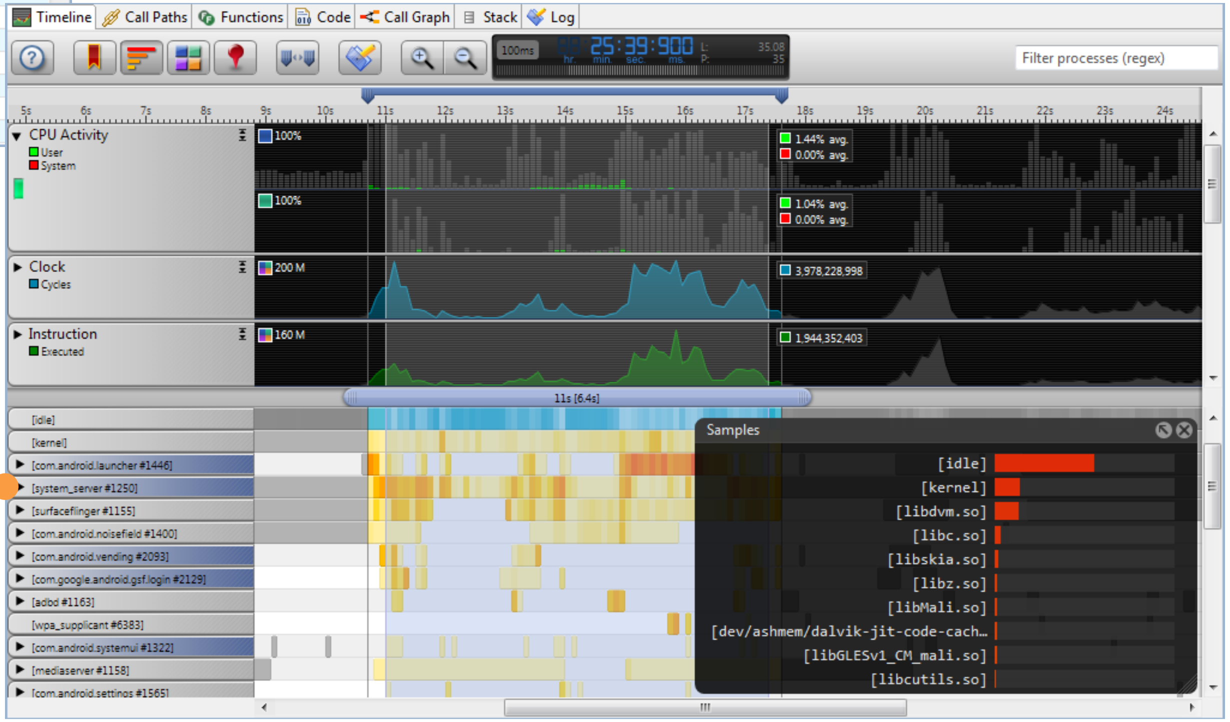
Click on the function name to go to source code level profile

Bottom-Up Shared Library Analysis



Select the library or function to look into, then navigate to Call Paths or Timeline

Processes or call paths using it will be automatically highlighted



Power Measurement Interfaces

Data Acquisition

ARM® Energy Probe

- 3-channel
- System-level analysis
- Easy to deploy
- Affordable

Good for trend spotting and application optimization

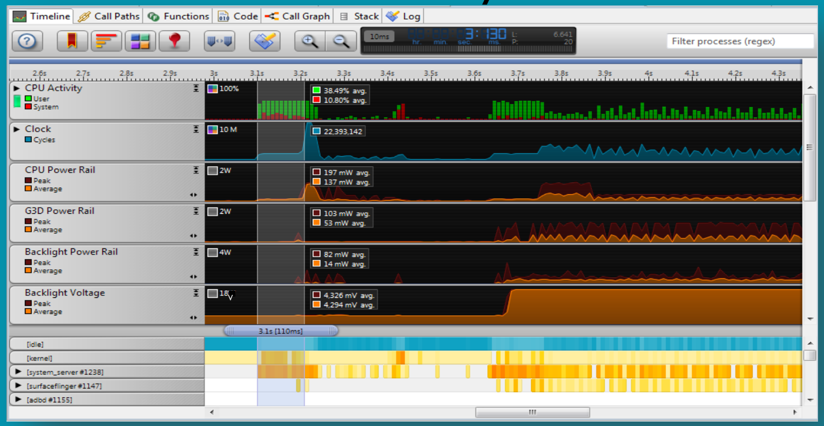
NI DAQ USB-62xx

- 40+ analog inputs
- Subcomponent sensitivity
- High fidelity
- Higher cost

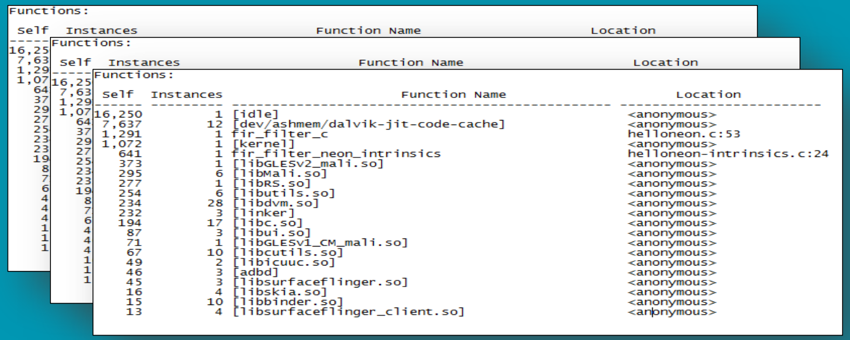
Good for OS power management tuning and benchmarking

ARM DS-5 Streamline Performance Analyzer

Visual Analysis



Automated Tests

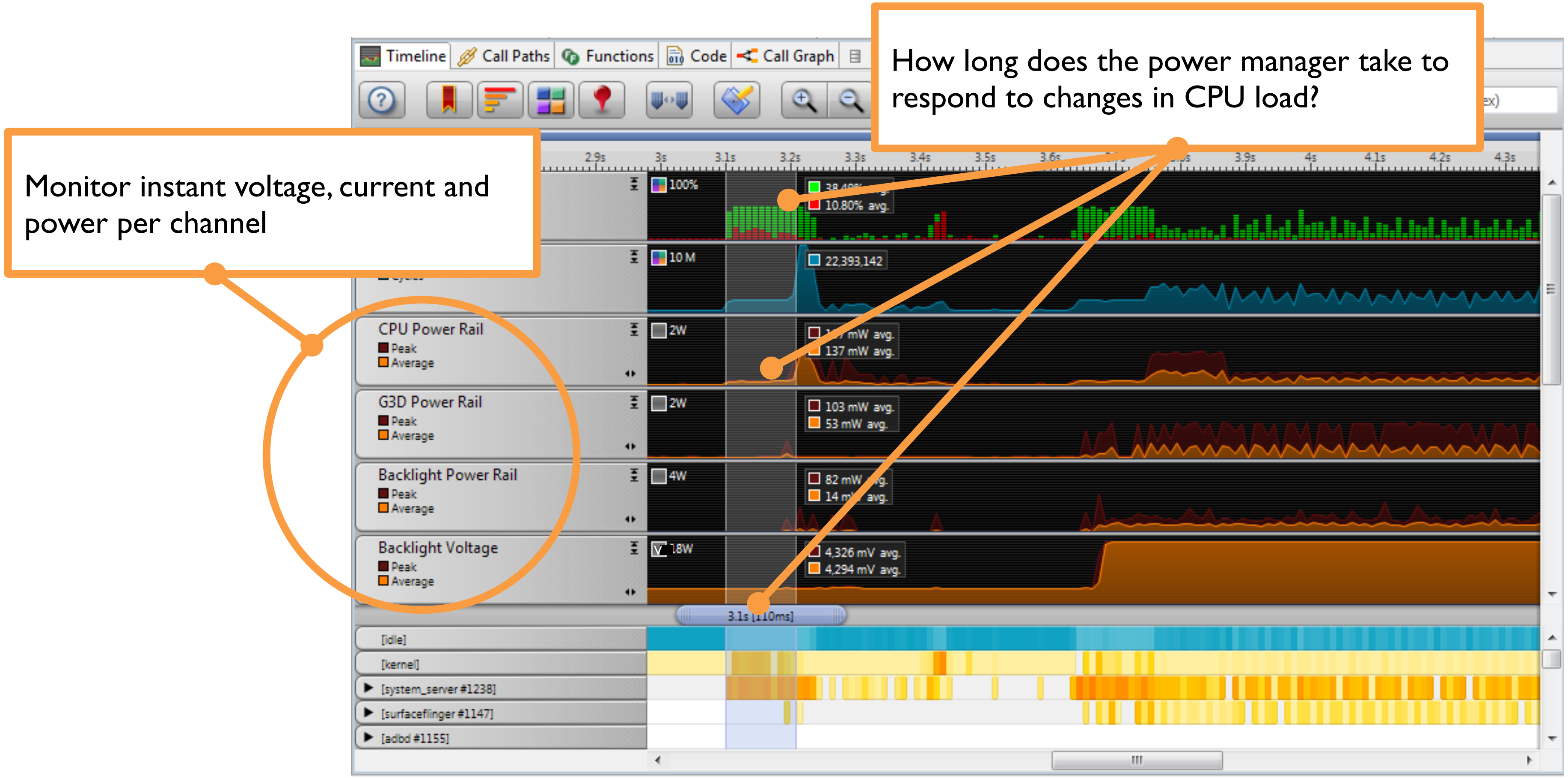


Functions:

Self	Instances	Function Name	Location
16,250	1	[idle]	<anonymous>
7,637	12	[dev/ashmem/dalvik-jit-code-cache]	<anonymous>
1,291	1	fir_filter_c	helloneon.c:53
1,072	1	[kernel]	<anonymous>
641	1	fir_filter_neon_intrinsics	helloneon-intrinsics.c:24
373	1	[libGLESv2_mali.so]	<anonymous>
295	6	[libMali.so]	<anonymous>
277	1	[libRS.so]	<anonymous>
254	6	[libutils.so]	<anonymous>
234	28	[libdvm.so]	<anonymous>
232	3	[linker]	<anonymous>
194	17	[libc.so]	<anonymous>
87	3	[libui.so]	<anonymous>
71	1	[libGLESv1_CM_mali.so]	<anonymous>
67	10	[libcutils.so]	<anonymous>
49	2	[libicuuc.so]	<anonymous>
46	3	[adbd]	<anonymous>
45	3	[libsurfaceflinger.so]	<anonymous>
16	4	[libskia.so]	<anonymous>
15	10	[libbinder.so]	<anonymous>
13	4	[libsurfaceflinger_client.so]	<anonymous>

The *Power* of Having It All in One Place

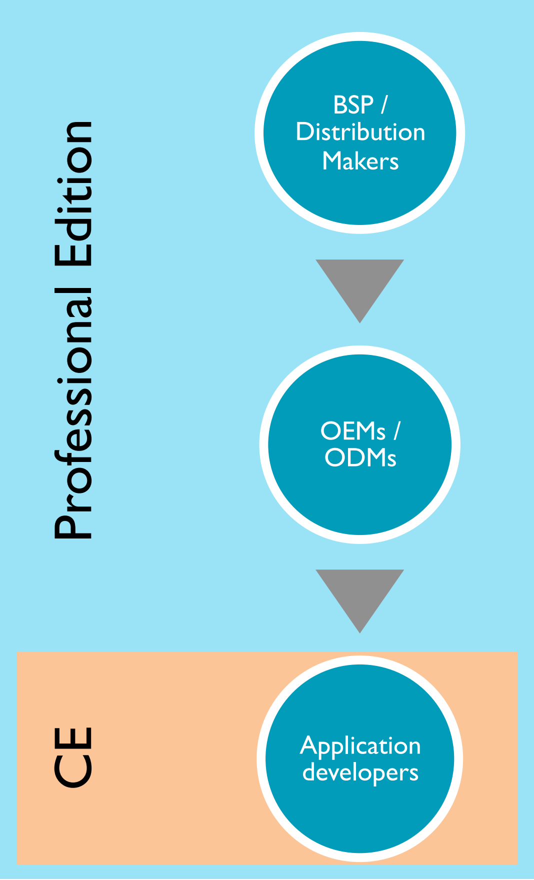
How effective are you at managing your energy budget?



ARM

Community Edition vs. Professional Edition

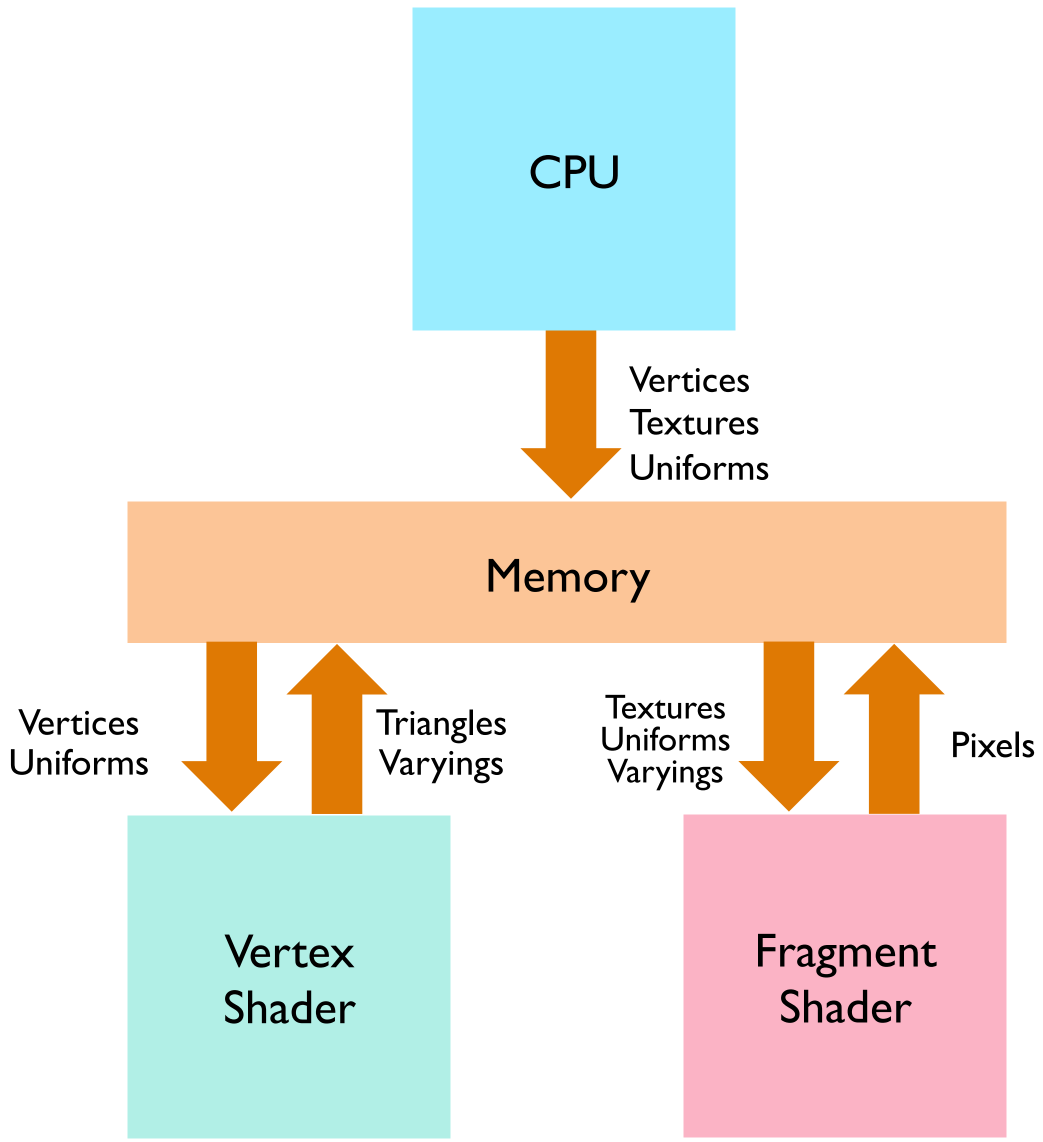
Which is the right version for you?



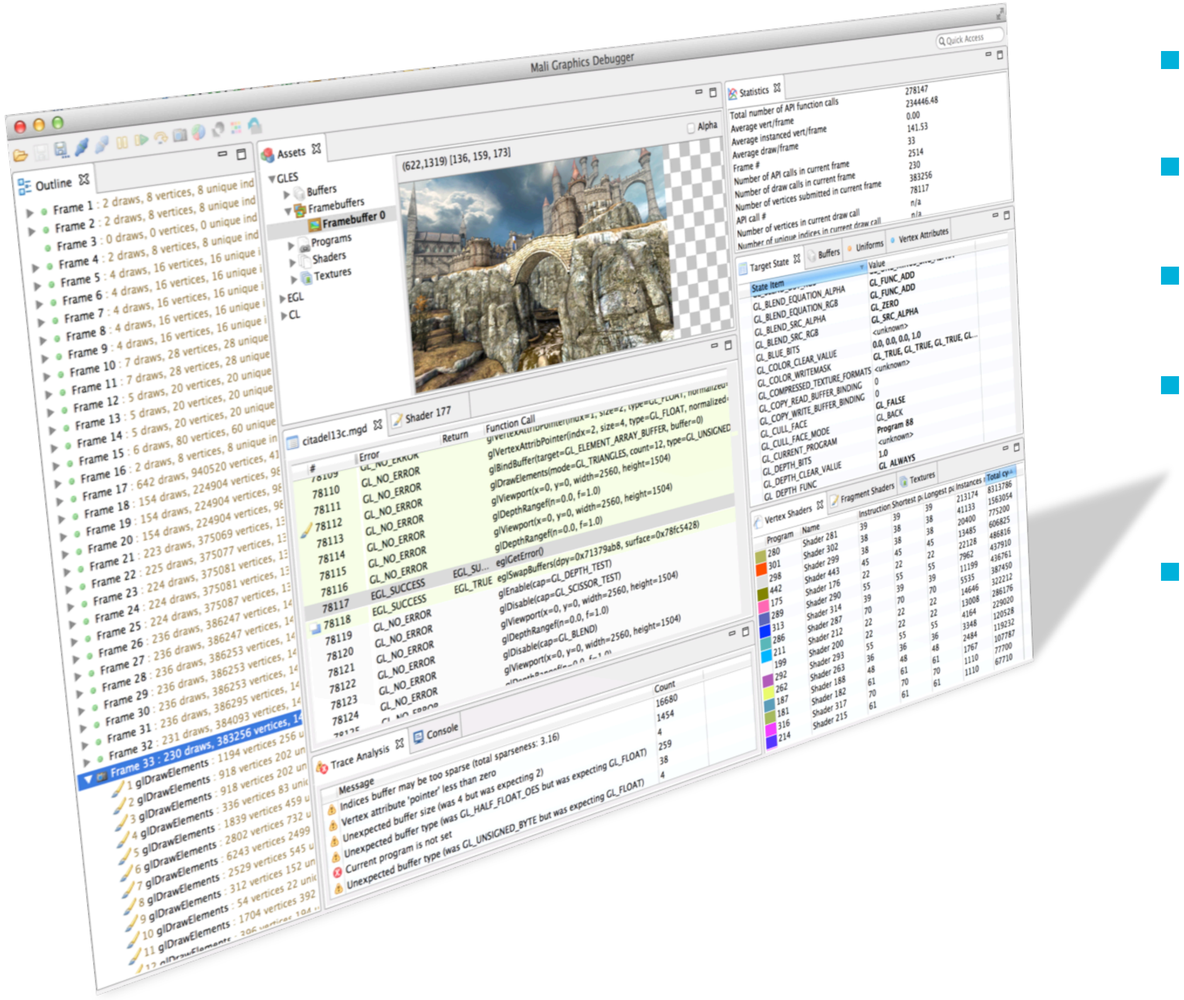
	Community	Professional
Typical Use Case	Simple application profiling	System-wide, SMP analysis
Program Images	1	Limited to host memory
Timeline View		
*** Performance Charts**	✓	✓
*** Process Bars**	✓	✓
*** ARM® Mali™ GPU Analysis**	✓	✓
*** Quick Profile Summary**		✓
*** Core Affinity Mode**		✓
*** Energy Probe data capture**		✓
*** Time Filtering**		✓
*** Annotation**	✓	✓
Call Paths View		✓
Functions View	✓	✓
Code View		✓
Call Graph		✓
Stack View		✓
Log View		✓
Command Line		✓
Event Based Sampling		✓

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® Mali™ Graphics Debugger

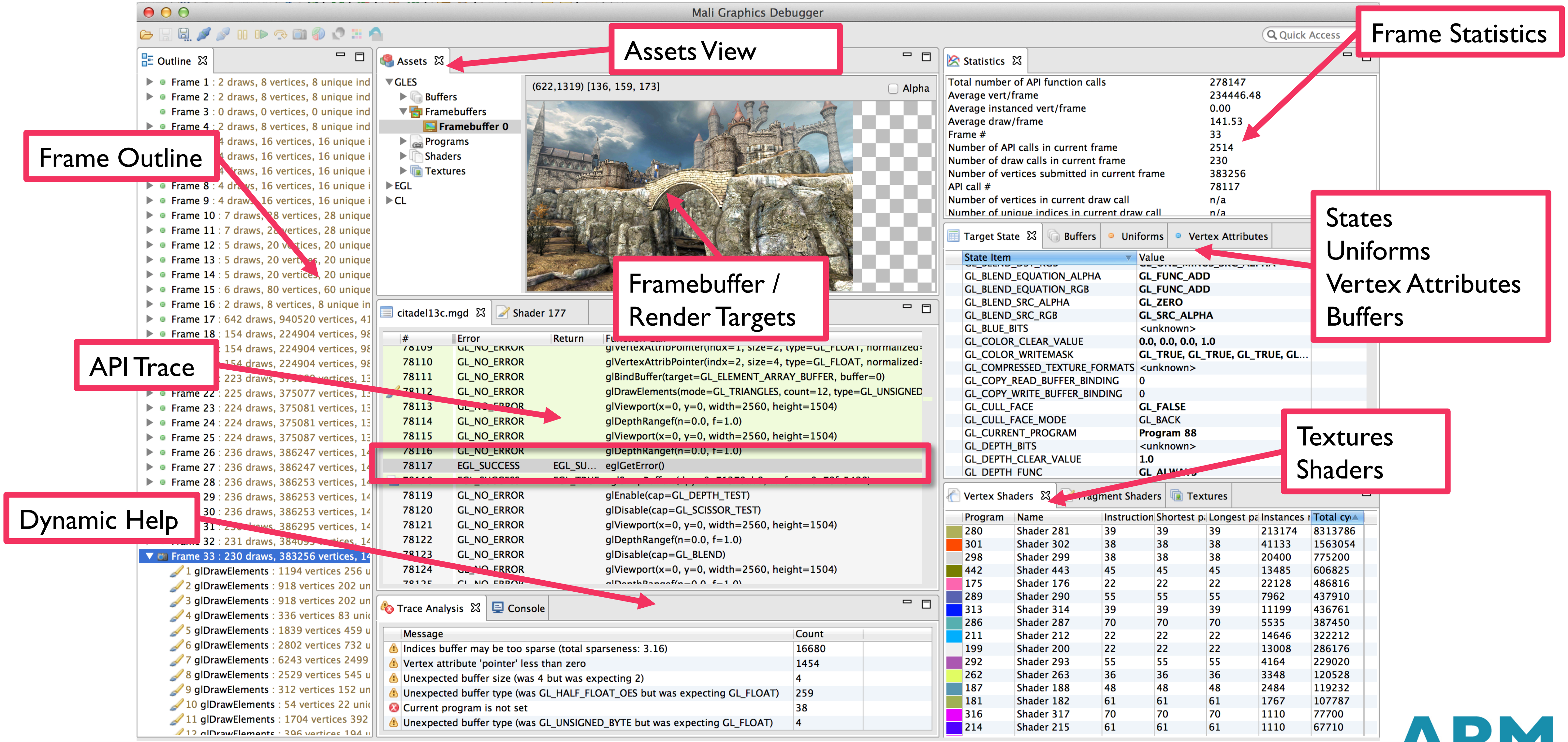


- Graphics debugging for content developers
- API level tracing
- Understand issues and causes at frame level
- Support for OpenGL® ES 2.0, 3.0, EGL™ & OpenCL™ 1.1
- Complimentary to DS-5 Streamline

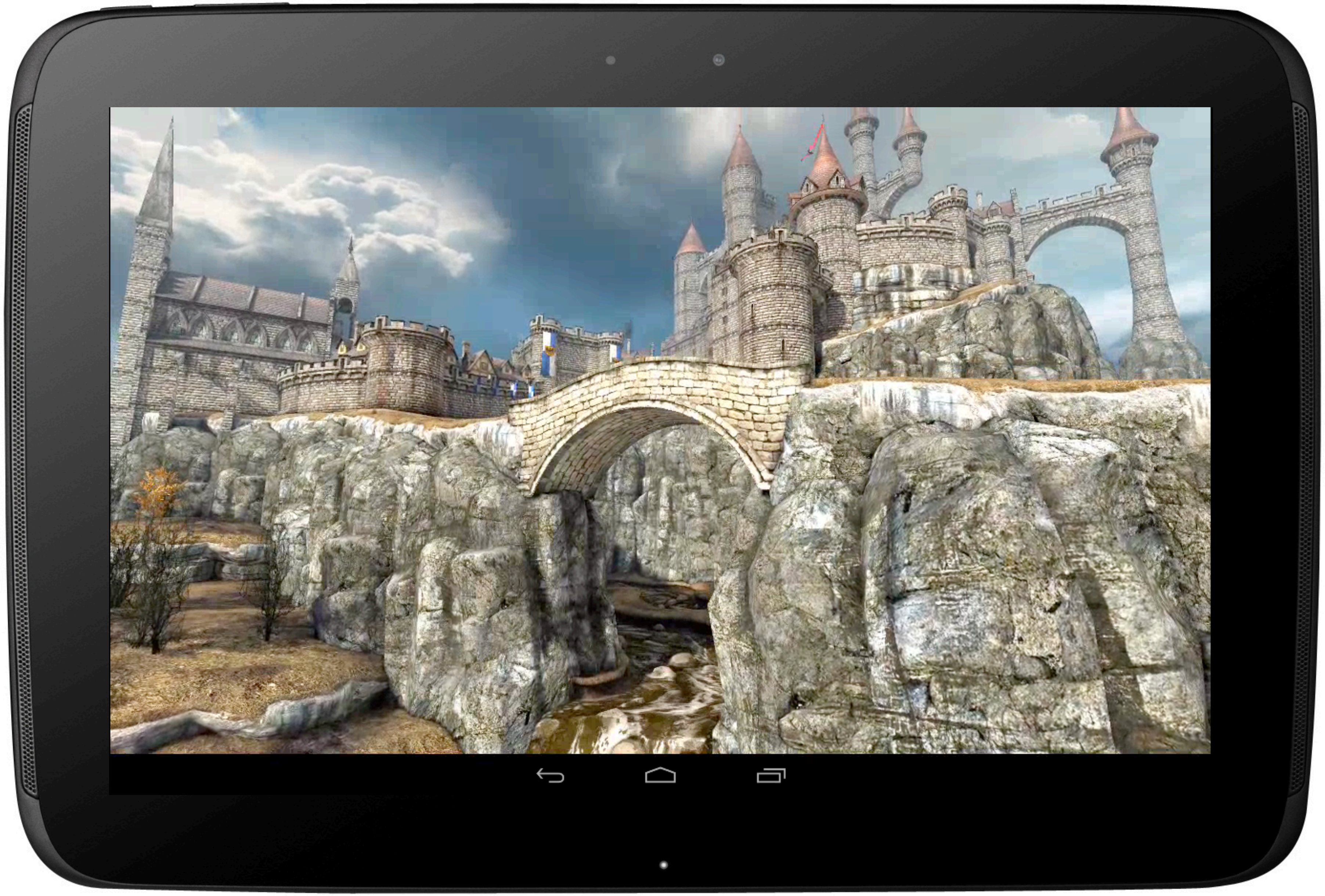
v1.2.2 released in February

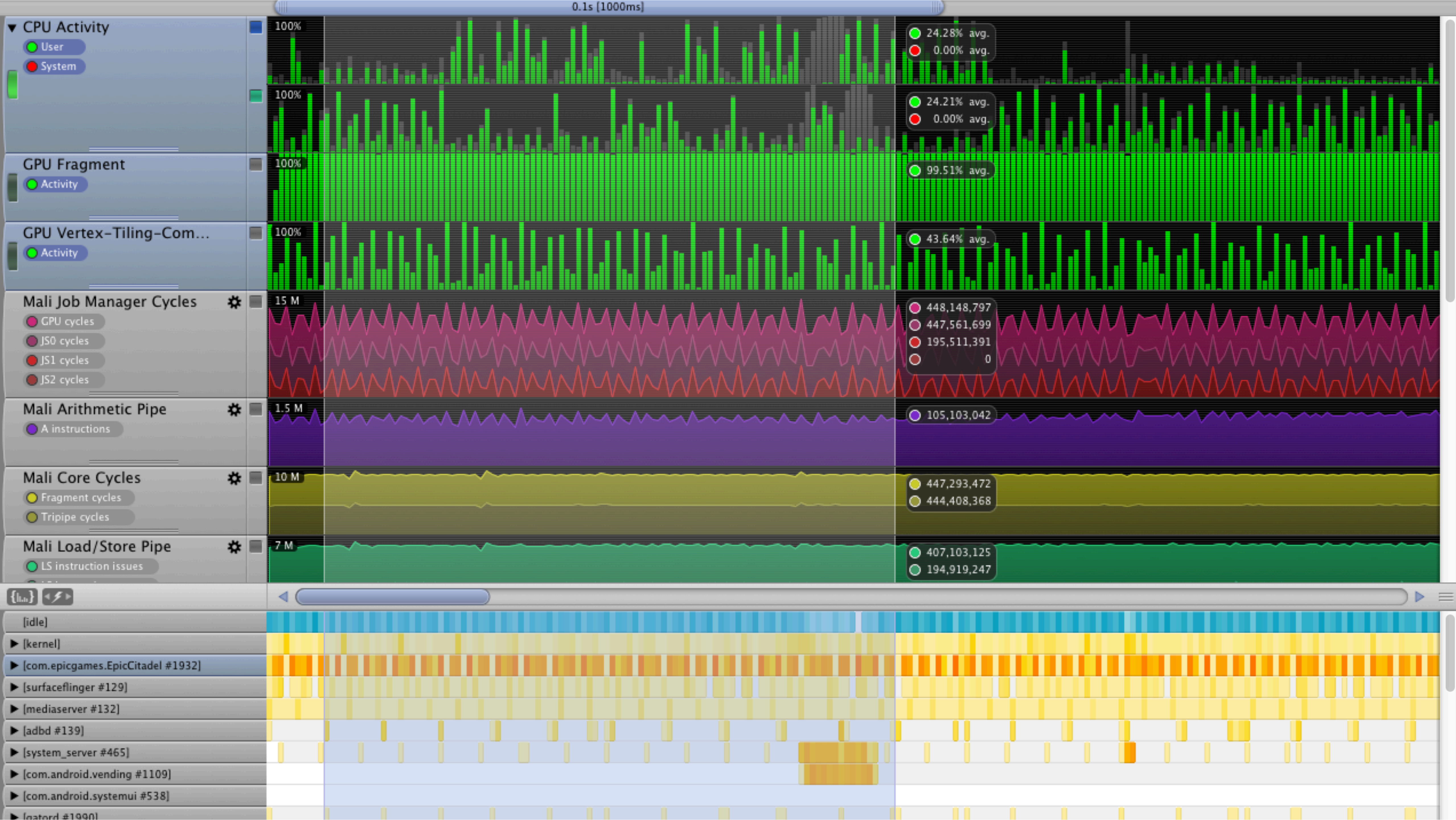
v1.3 will be available soon

Investigation with the ARM® Mali™ Graphics Debugger



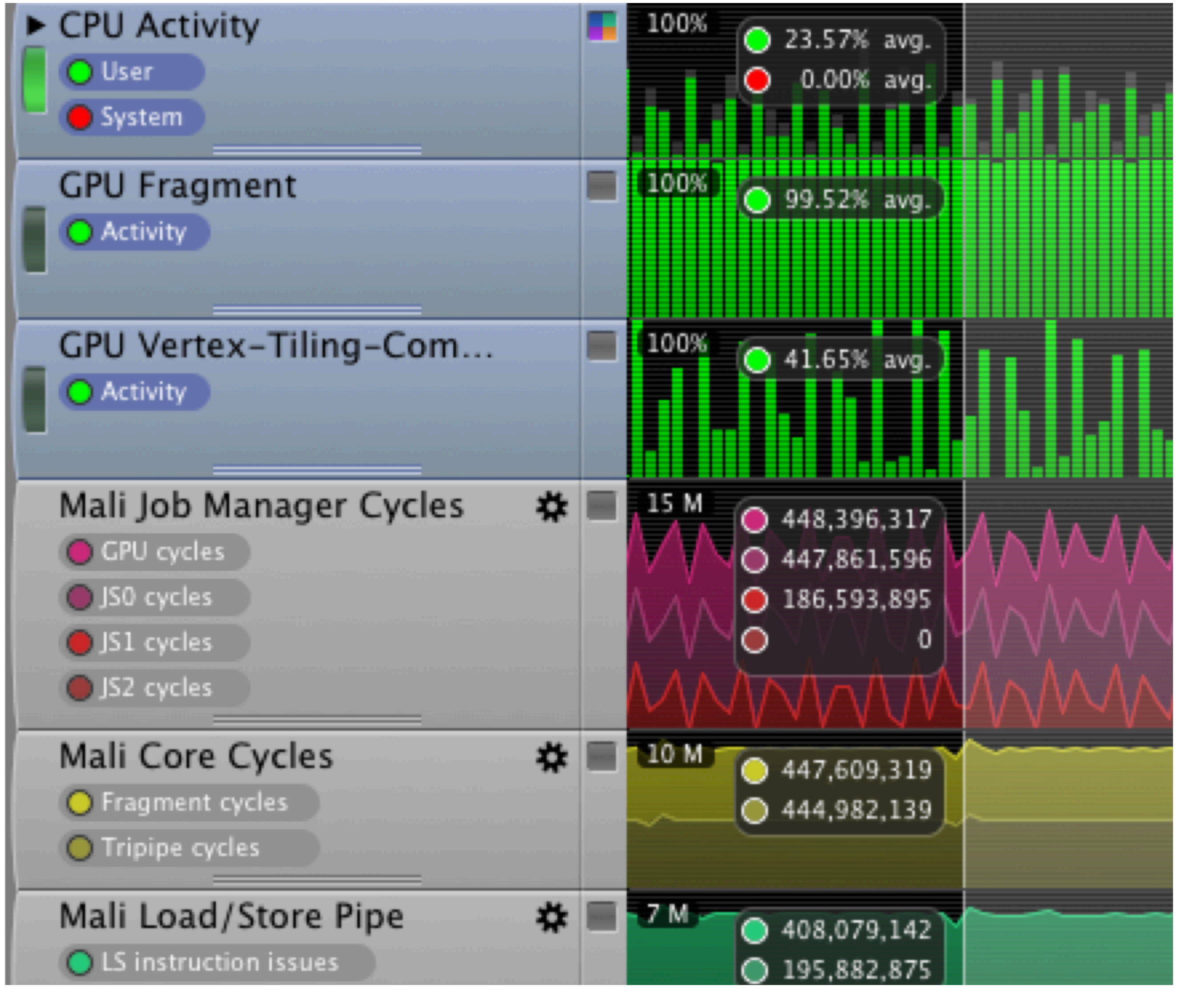
Epic Citadel




0.1s [1000ms]
CPU Activity
User
System
100%
24.28% avg.
0.00% avg.
24.21% avg.
0.00% avg.
GPU Fragment
Activity
99.51% avg.
GPU Vertex-Tiling-Com...
43.64% avg.
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]
[gatord #1990]

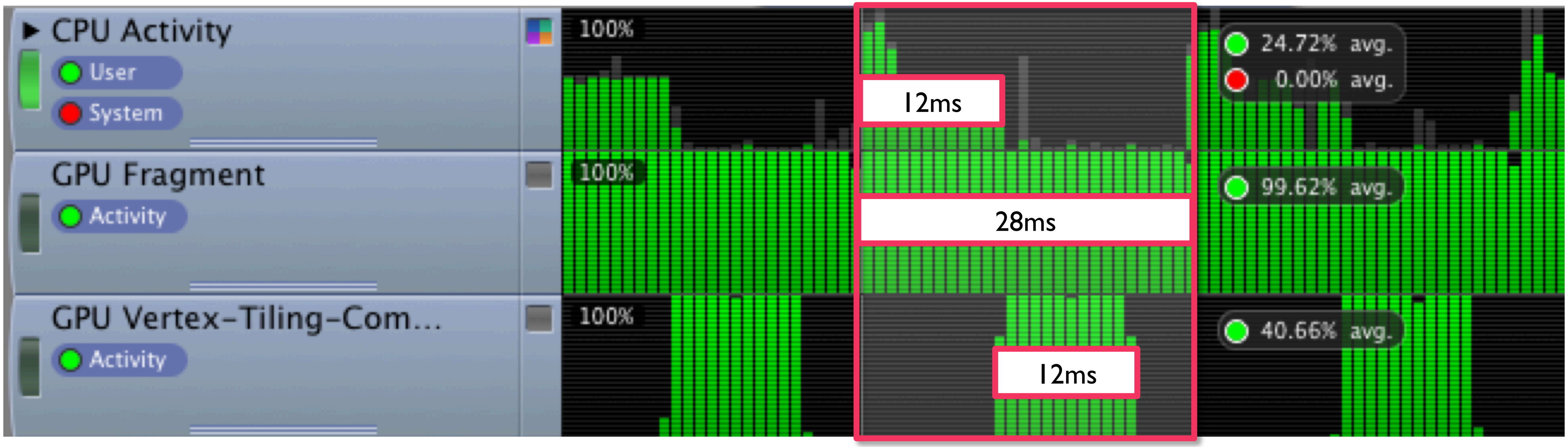
Profiling via ARM® DS-5 Development Studio

- DS-5 Streamline to capture data
  - Google Nexus 10, Android™ 4.4
  - Dual core ARM Cortex®-A15, Mali™-T604
- Low CPU activity (*CPU Activity -> User*) that averages to 24% over one second
- Burst in GPU activity: 99% utilization (*GPU Fragment → Activity*)
- While rendering the most complicated scene, the application is capable of 36 fps (29ms/frame)



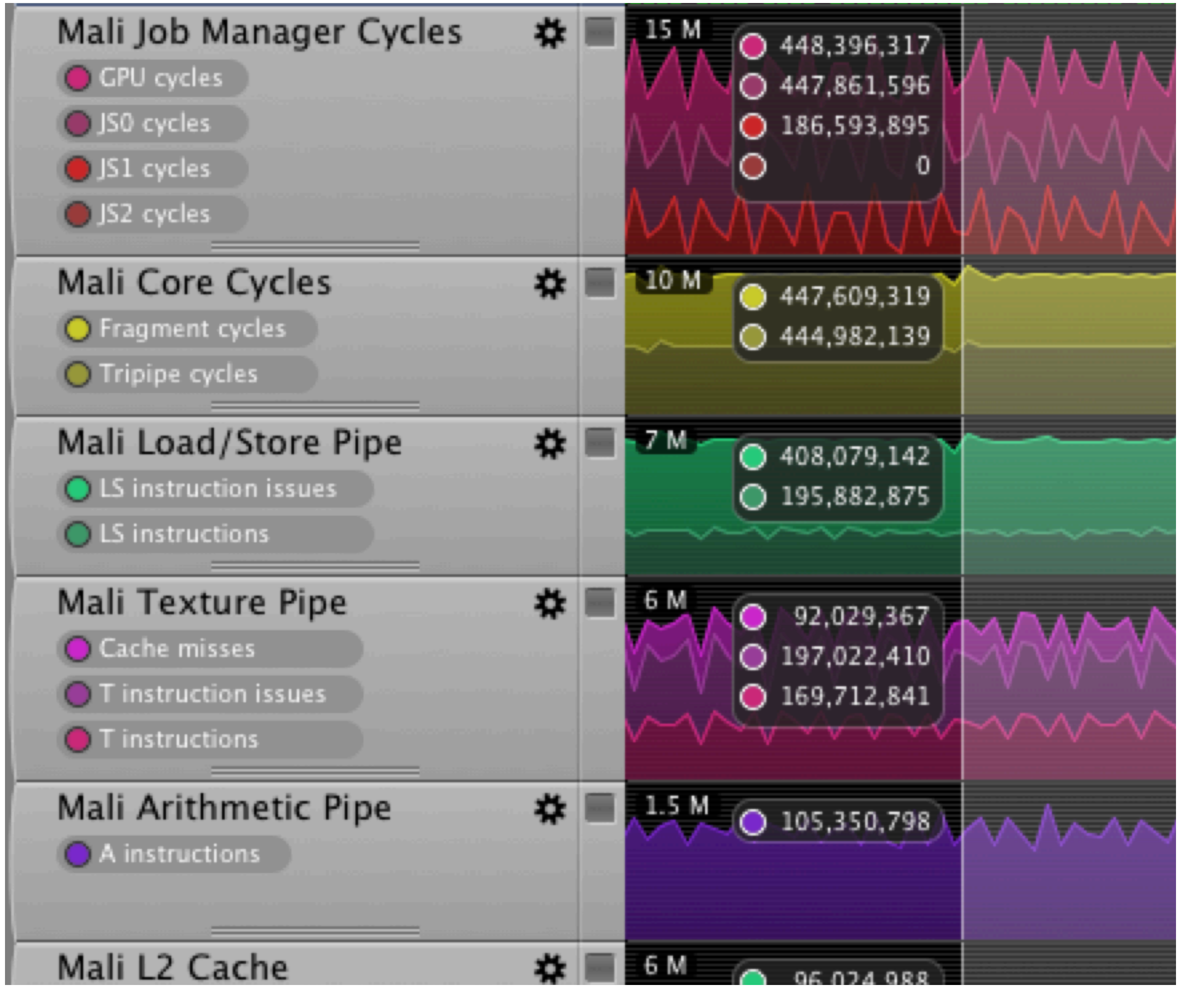
The Application is GPU bound

The CPU has to wait until the fragment processing has finished



ARM® Mali™ GPU Hardware Counters

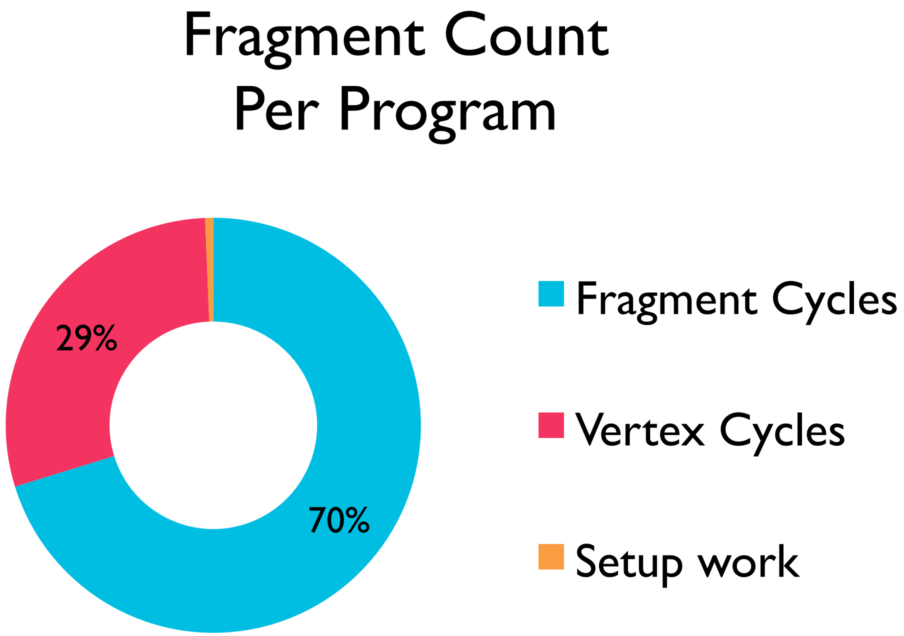
- Over the highlighted time of one second the GPU was active for 448m cycles (*Mali Job Manager Cycles* → *GPU cycles*)
- With this hardware, the maximum number of cycles is 450m
- A first pass of optimization would lead to a higher frame rate
- After reaching V-SYNC, optimization can leads to saving energy and to a longer play time



ARM

Vertex and Fragment Processing

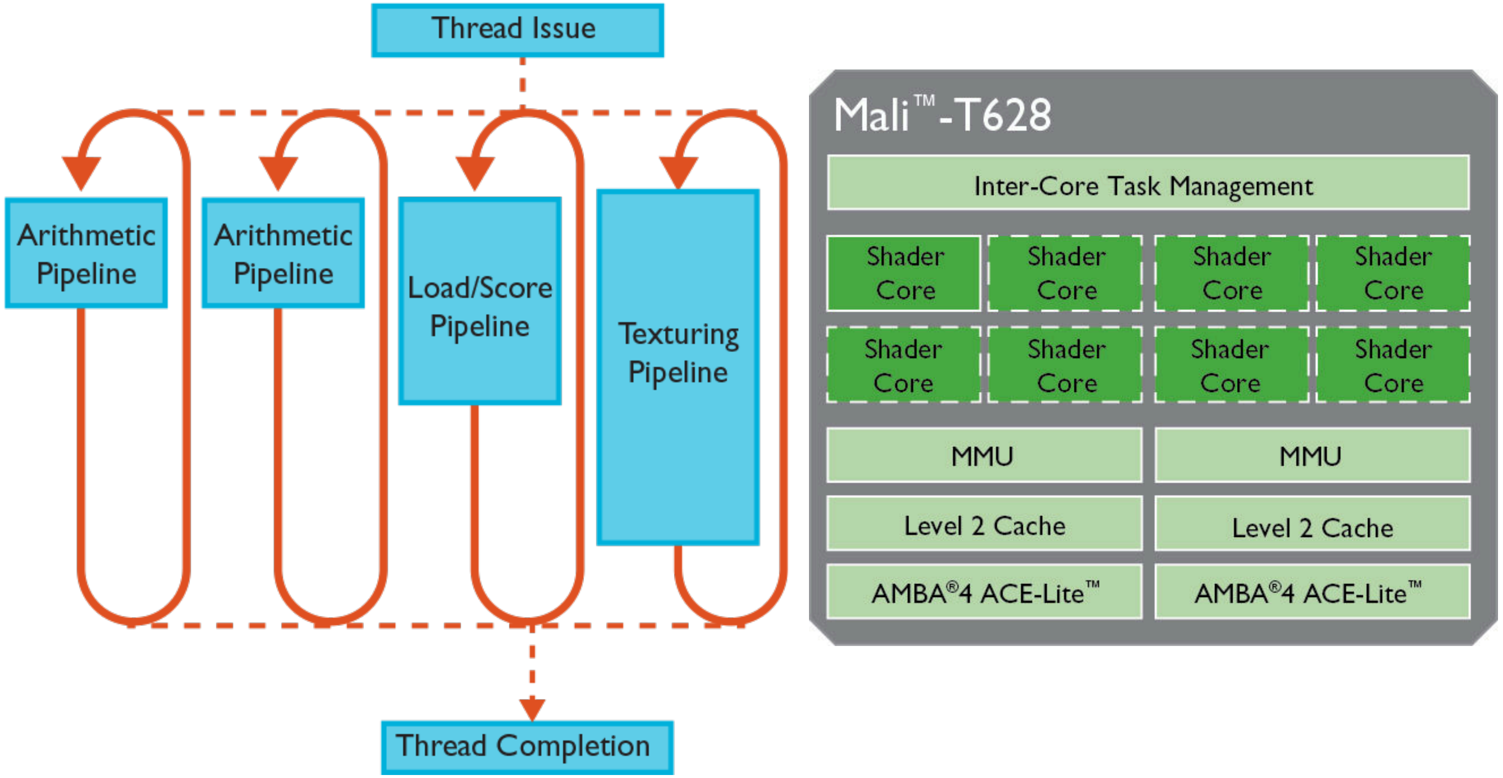
- GPU is spending:
  - **186m** (29%) on vertex processing
    (*ARM® Mali™ Job Manager Cycles* → *JS1 cycles*)
  - **448m** (70%) on fragment processing
    (*Mali Job Manager Cycles* → *JS0 cycles*)



There might be an overhead in the job manager trying to optimize vertex list packing into jobs.

ARM® Mali™-T628 GPU Tripipe Cycles

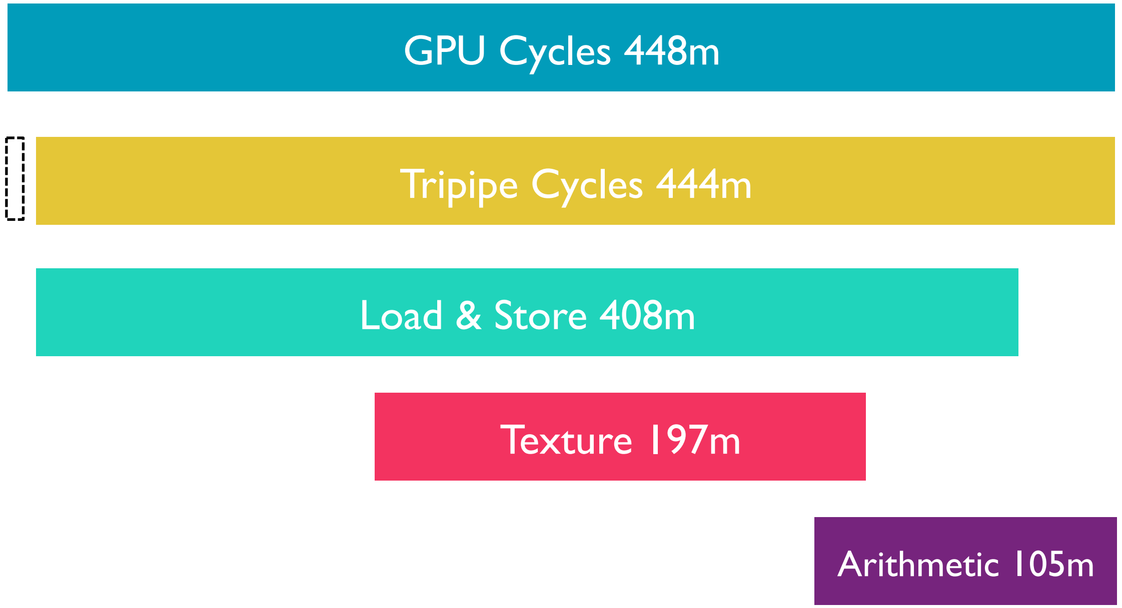
- Arithmetic instructions
  - Math in the shaders
- Load & Store instructions
  - Uniforms, attributes and varyings
- Texture instructions
  - Texture sampling and filtering

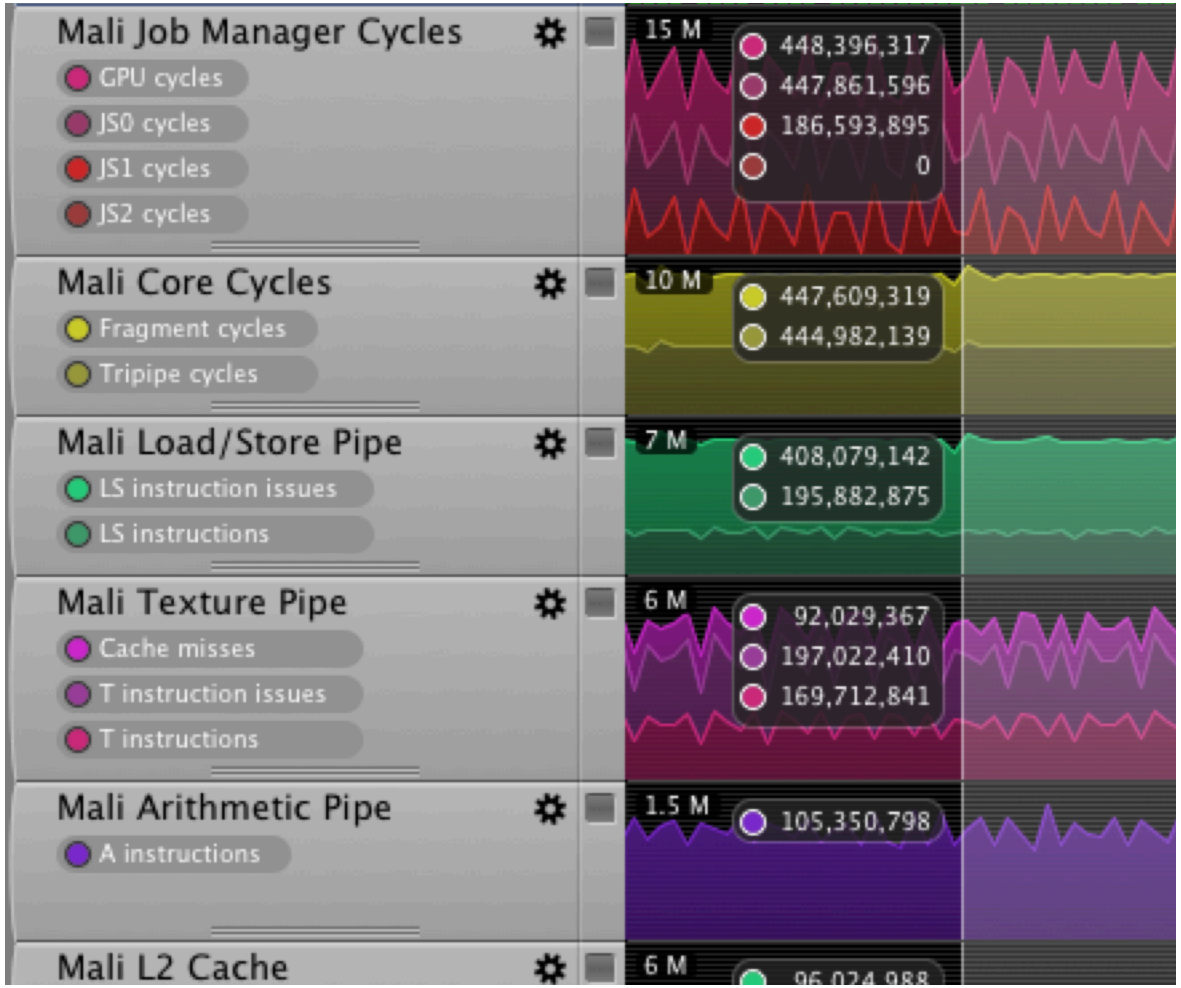


- Instructions can run in parallel
- Each one can be a bottleneck
- There are two arithmetic pipelines so we should aim to increase the arithmetic workload

Inspect the Tripipe Counters

Reduce the load on the L/S pipeline





Tripipe Counters

Cycles per instruction metrics

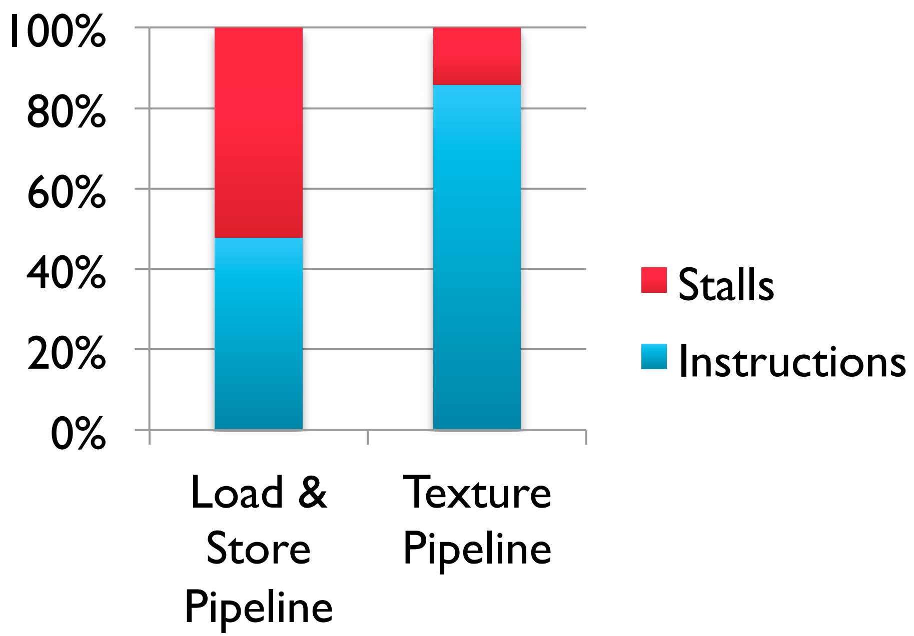
- It's easy to calculate a couple of CPI (cycles per instruction) metrics:

- For the load/store pipeline we have:

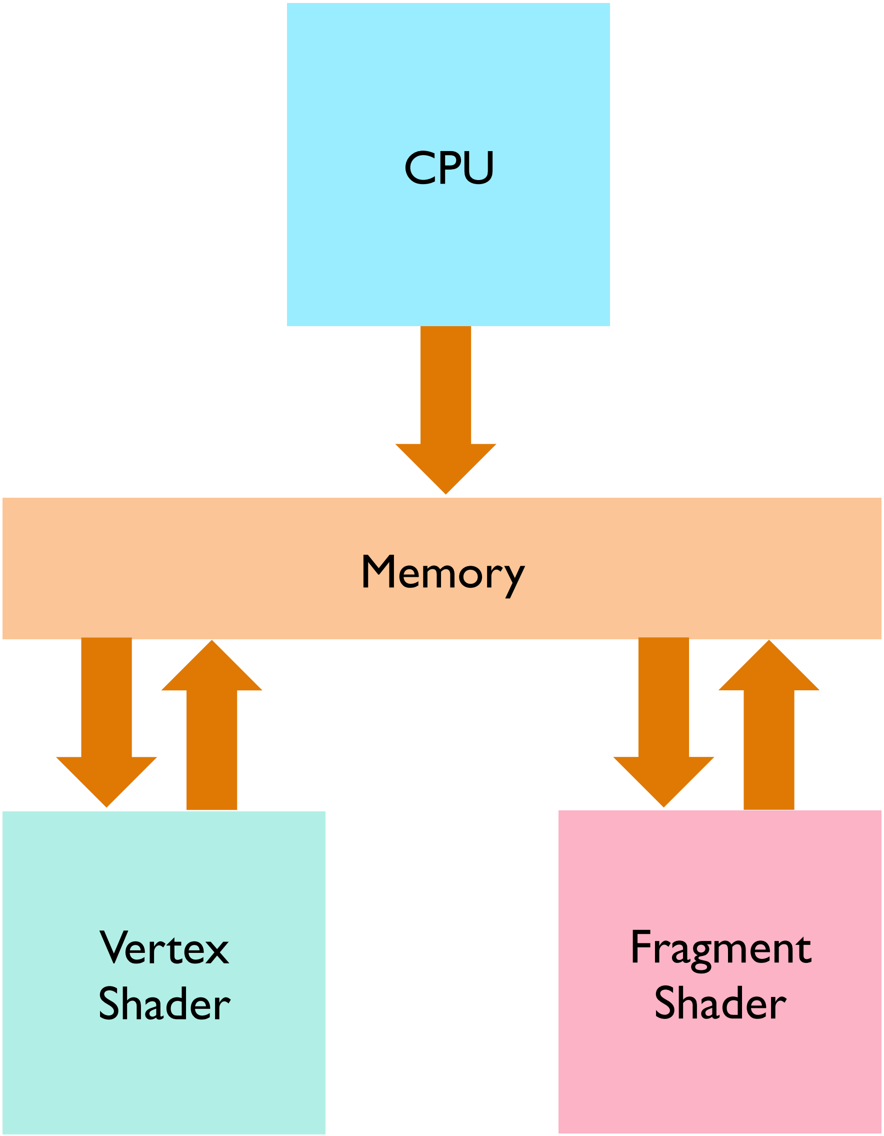
  408m (*Mali Load/Store Pipe → LS instruction issues*)
  / **195m** (*Mali Load/Store Pipe → LS instructions*)
  = 2.09 cycles/instruction

- For the texture pipeline we have:

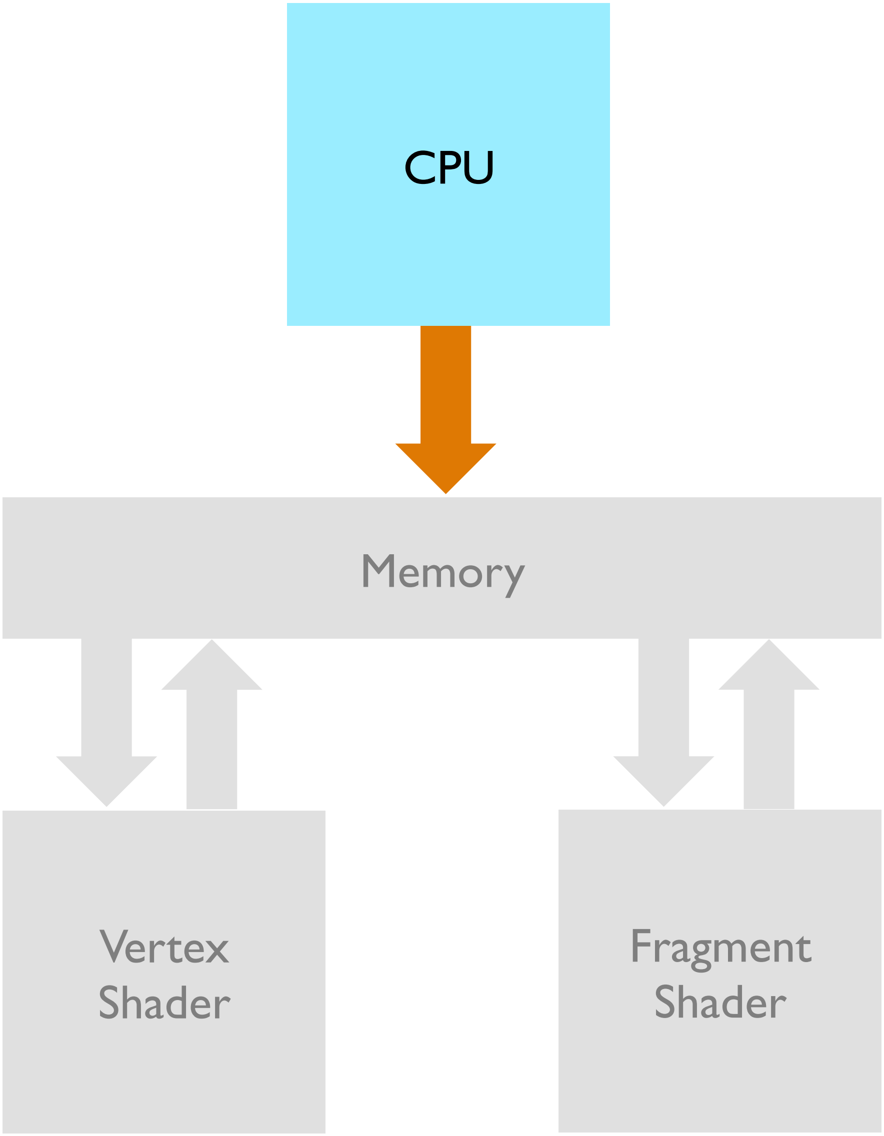
  197m (*Mali Texture Pipe → T instruction issues*)
  / **169m** (*Mali Texture Pipe → T instructions*)
  = 1.16 cycles/instruction



ARM


CPU
Memory
Vertex Shader
Fragment Shader

CPU Bound

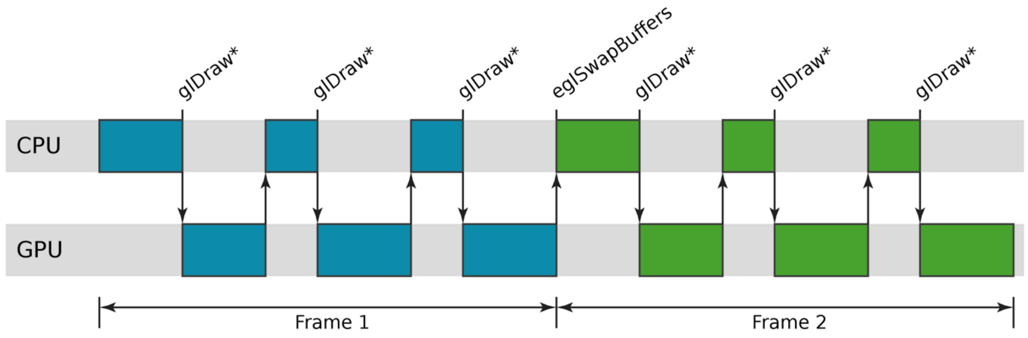


ARM

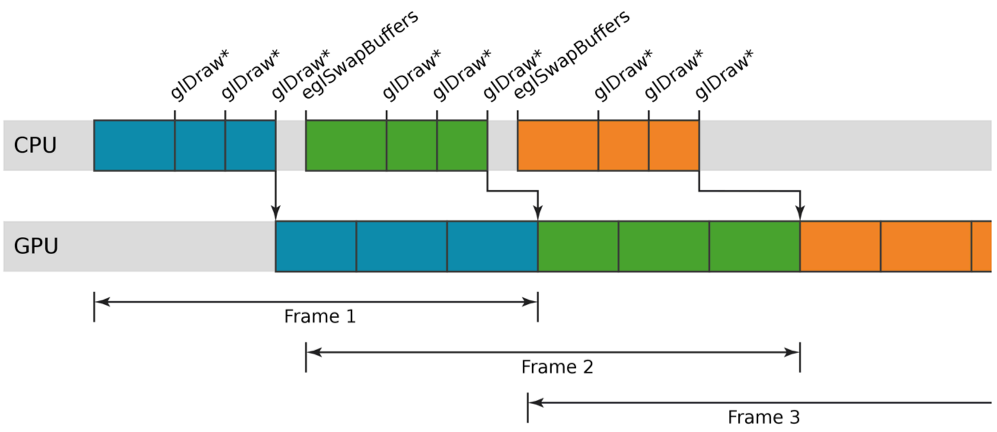
CPU Bound

- 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.)

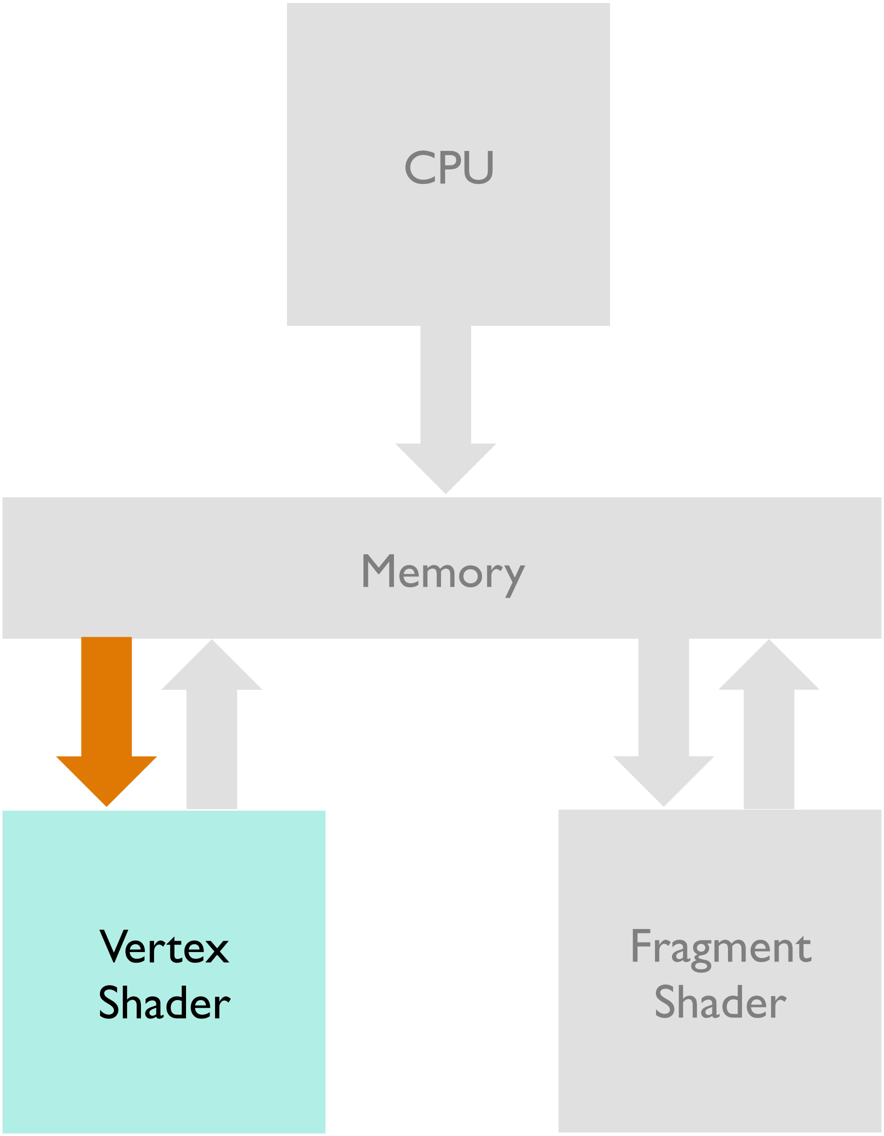
Synchronous Rendering



Deferred Rendering



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

Frame 33 : 230 draws, 383256 vertices, 142835 unique ind

1 glDrawElements : 1194 vertices 256 unique indices
2 glDrawElements : 918 vertices 202 unique indices
3 glDrawElements : 918 vertices 202 unique indices
4 glDrawElements : 336 vertices 83 unique indices
5 glDrawElements : 1839 vertices 459 unique indices
6 glDrawElements : 2802 vertices 732 unique indices
7 glDrawElements : 6243 vertices 2499 unique indices
8 glDrawElements : 2529 vertices 545 unique indices
9 glDrawElements : 312 vertices 152 unique indices
10 glDrawElements : 54 vertices 22 unique indices
11 glDrawElements : 1704 vertices 392 unique indices
12 glDrawElements : 396 vertices 194 unique indices
13 glDrawElements : 4038 vertices 1124 unique indices
14 glDrawElements : 8220 vertices 2198 unique indices
15 glDrawElements : 564 vertices 291 unique indices
16 glDrawElements : 528 vertices 233 unique indices
17 glDrawElements : 2166 vertices 681 unique indices
18 glDrawElements : 3858 vertices 2067 unique indices
19 glDrawElements : 702 vertices 468 unique indices
20 glDrawElements : 1671 vertices 808 unique indices
21 glDrawElements : 2322 vertices 836 unique indices
22 glDrawElements : 2277 vertices 917 unique indices
23 glDrawElements : 4251 vertices 1131 unique indices

ARM

Vertex Count and Shader Optimizations

Identify the top heavyweight vertex shaders

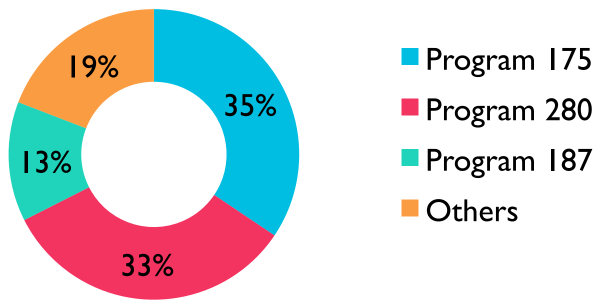
citadel13c.mgd | Shader 176

```
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

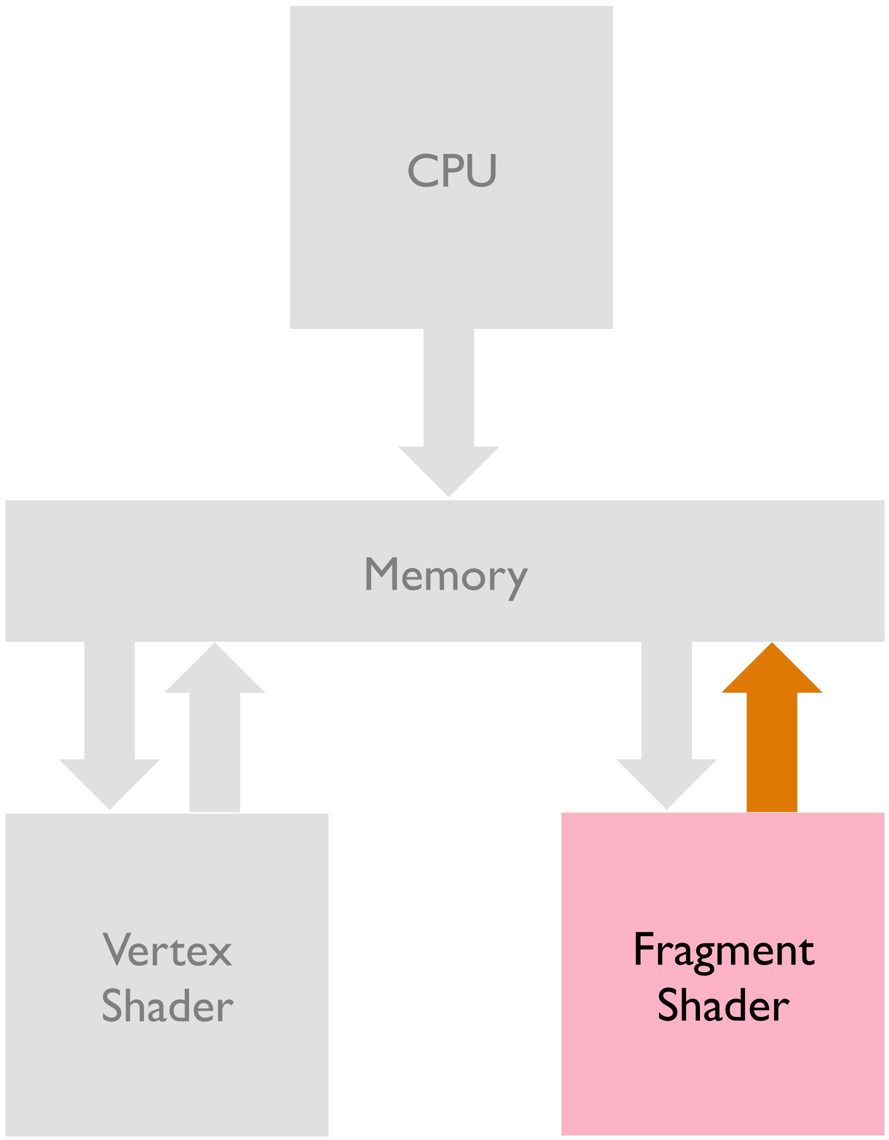
Program	Name	Instruction	Shortest pa	Longest pa	Instances r	Total cy
175	Shader 176	22	22	22	148500	3267000
280	Shader 281	39	39	39	80595	3143205
187	Shader 188	48	48	48	26328	1263744
181	Shader 182	61	61	61	11487	700707
211	Shader 212	22	22	22	14646	322212
289	Shader 290	55	55	55	5484	301620
298	Shader 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



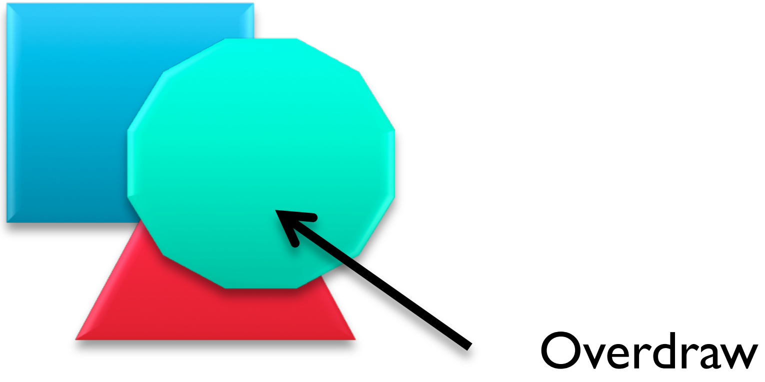
ARM

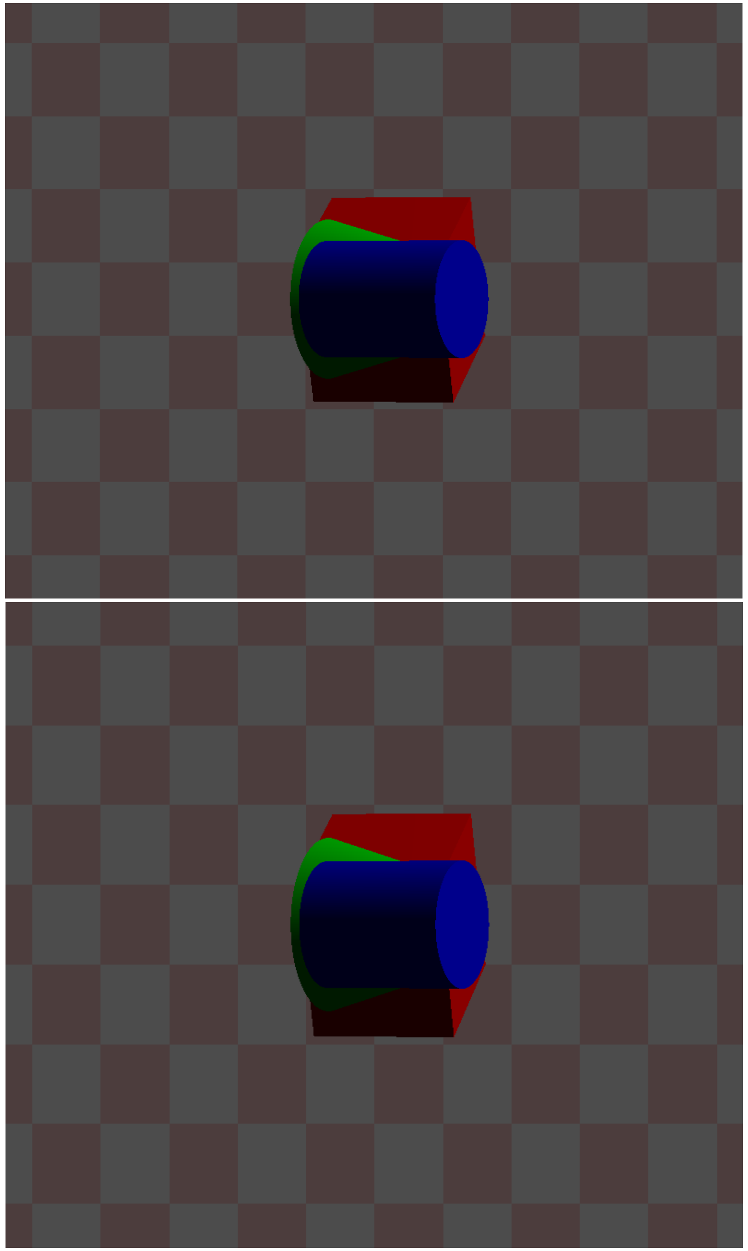
Fragment Bound

- 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

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
- Limiting the amount of transparency in the scene can help

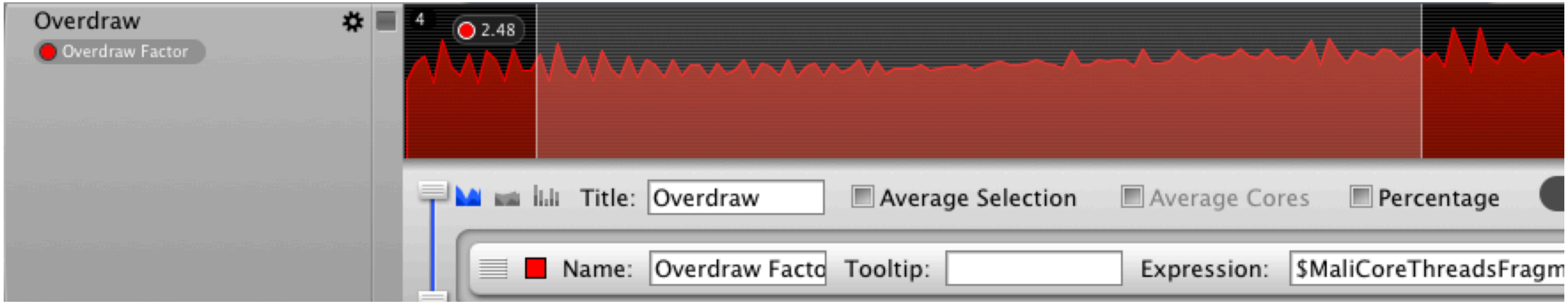




Overdraw Factor

- 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

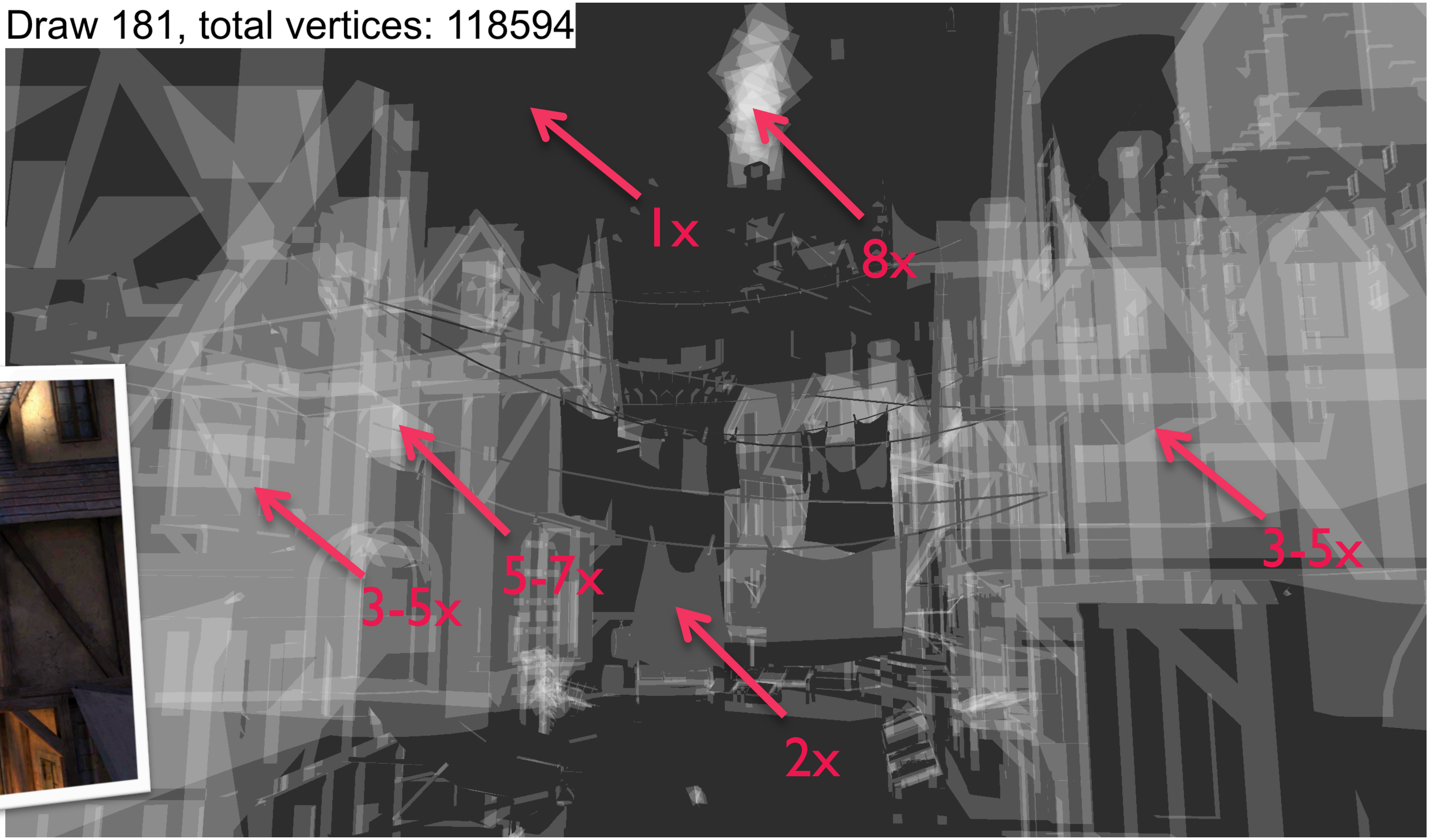


ARM

Frame Capture

Frame Analysis

Check the overdraw factor



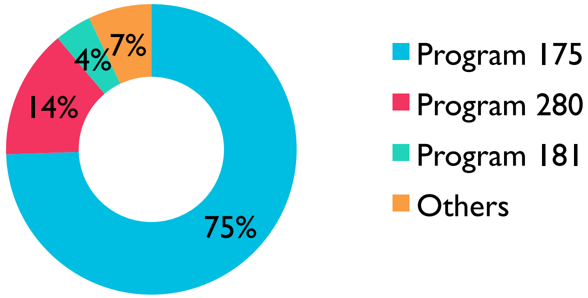
Shader Map and Fragment Count

Identify the top heavyweight fragment shaders

	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

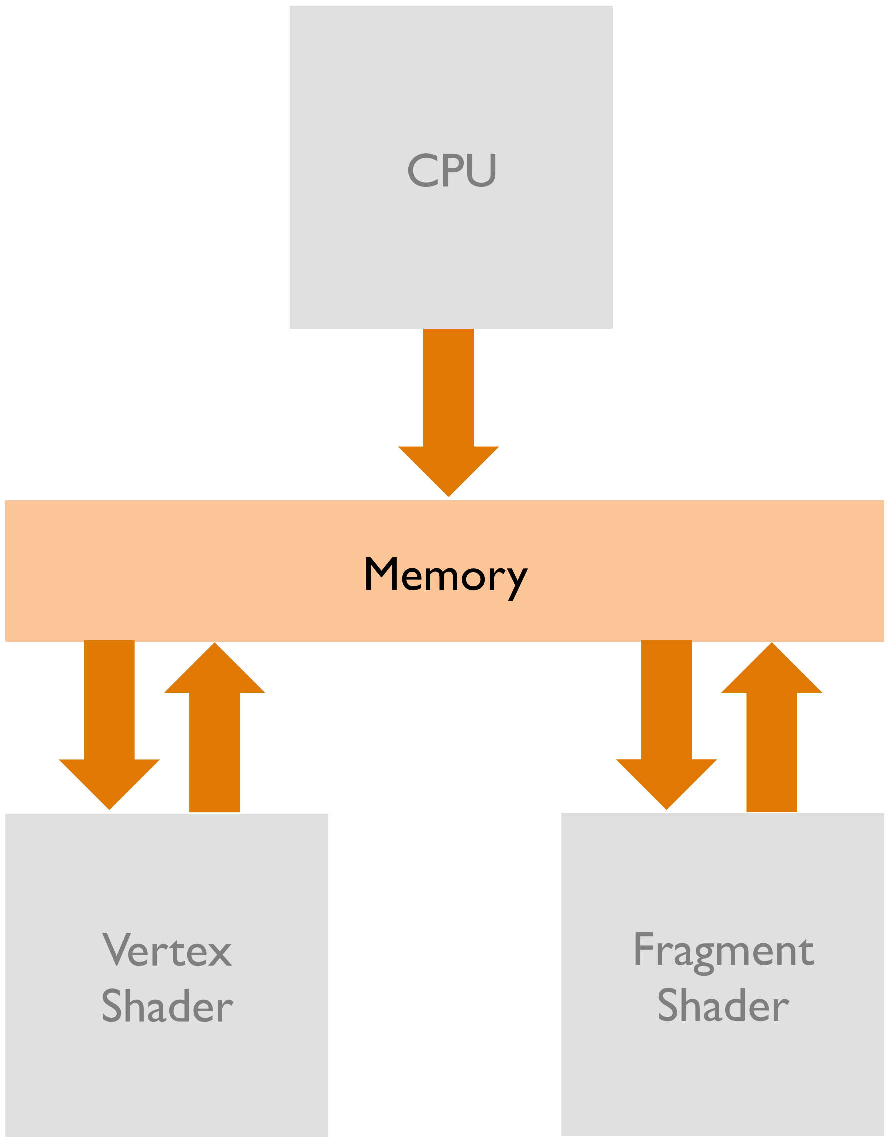
Shader Optimization

- 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 ;
```

Bandwidth Bound



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

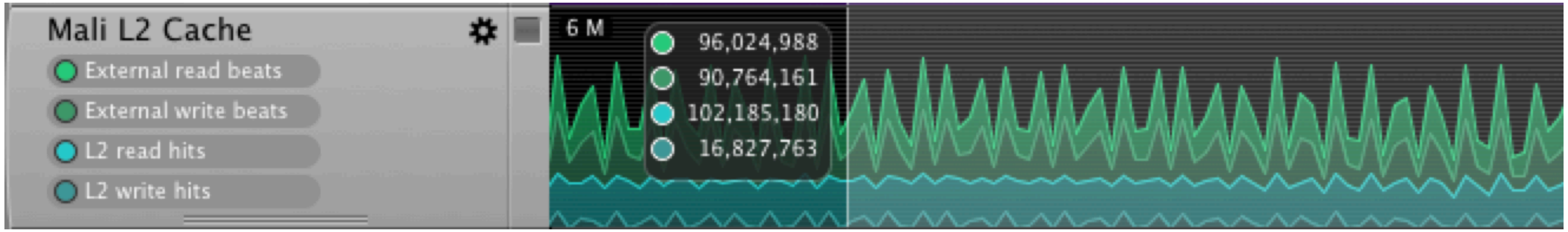
- The application is not bandwidth bound as it performs, over a period of one second:

(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



Bandwidth Bound

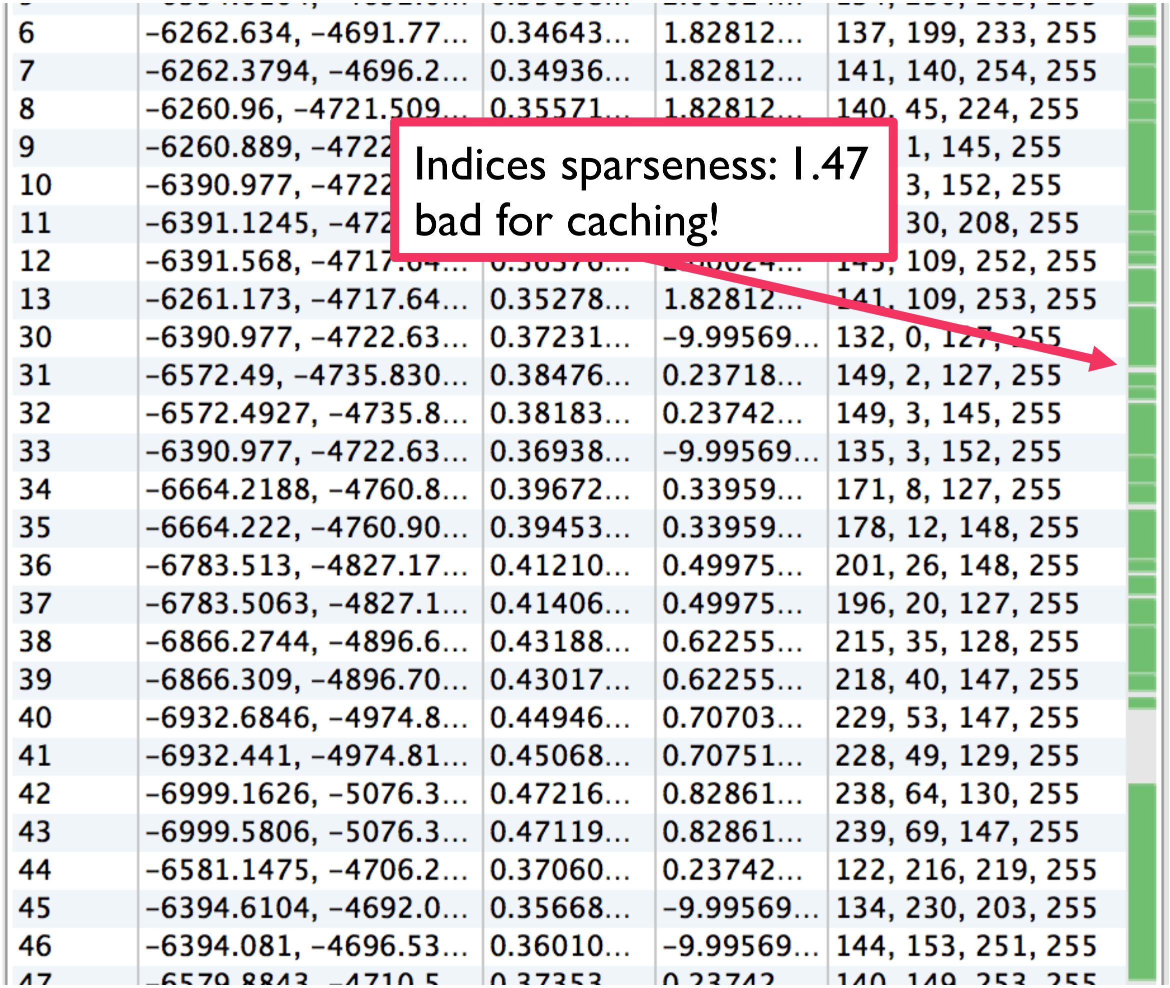
Vertices

- 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.

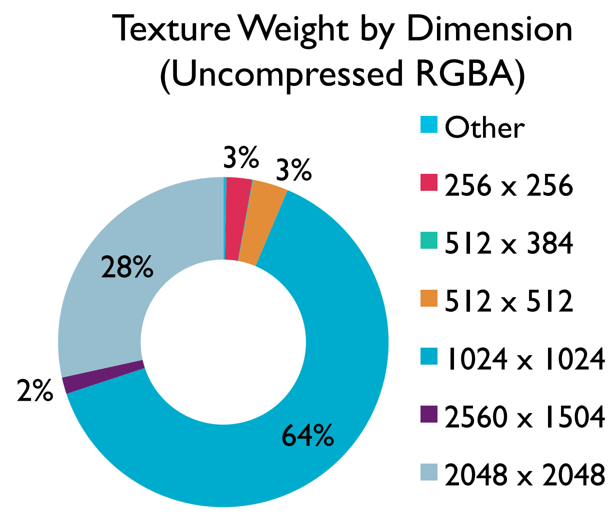


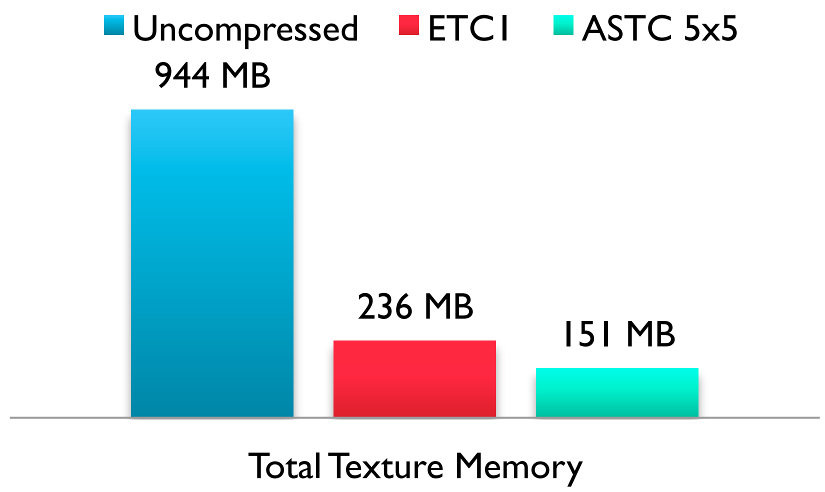
6	-6262.634, -4691.77...	0.34643...	1.82812...	137, 199, 233, 255
7	-6262.3794, -4696.2...	0.34936...	1.82812...	141, 140, 254, 255
8	-6260.96, -4721.509...	0.35571...	1.82812...	140, 45, 224, 255
9	-6260.889, -4722	[illegible]	[illegible]	1, 145, 255
10	-6390.977, -4722	[illegible]	[illegible]	3, 152, 255
11	-6391.1245, -472	[illegible]	[illegible]	30, 208, 255
12	-6391.568, -4717	[illegible]	[illegible]	109, 252, 255
13	-6261.173, -4717.64...	0.35278...	1.82812...	141, 109, 253, 255
30	-6390.977, -4722.63...	0.37231...	-9.99569...	132, 0, 127, 255
31	-6572.49, -4735.830...	0.38476...	0.23718...	149, 2, 127, 255
32	-6572.4927, -4735.8...	0.38183...	0.23742...	149, 3, 145, 255
33	-6390.977, -4722.63...	0.36938...	-9.99569...	135, 3, 152, 255
34	-6664.2188, -4760.8...	0.39672...	0.33959...	171, 8, 127, 255
35	-6664.222, -4760.90...	0.39453...	0.33959...	178, 12, 148, 255
36	-6783.513, -4827.17...	0.41210...	0.49975...	201, 26, 148, 255
37	-6783.5063, -4827.1...	0.41406...	0.49975...	196, 20, 127, 255
38	-6866.2744, -4896.6...	0.43188...	0.62255...	215, 35, 128, 255
39	-6866.309, -4896.70...	0.43017...	0.62255...	218, 40, 147, 255
40	-6932.6846, -4974.8...	0.44946...	0.70703...	229, 53, 147, 255
41	-6932.441, -4974.81...	0.45068...	0.70751...	228, 49, 129, 255
42	-6999.1626, -5076.3...	0.47216...	0.82861...	238, 64, 130, 255
43	-6999.5806, -5076.3...	0.47119...	0.82861...	239, 69, 147, 255
44	-6581.1475, -4706.2...	0.37060...	0.23742...	122, 216, 219, 255
45	-6394.6104, -4692.0...	0.35668...	-9.99569...	134, 230, 203, 255
46	-6394.081, -4696.53...	0.36010...	-9.99569...	144, 153, 251, 255

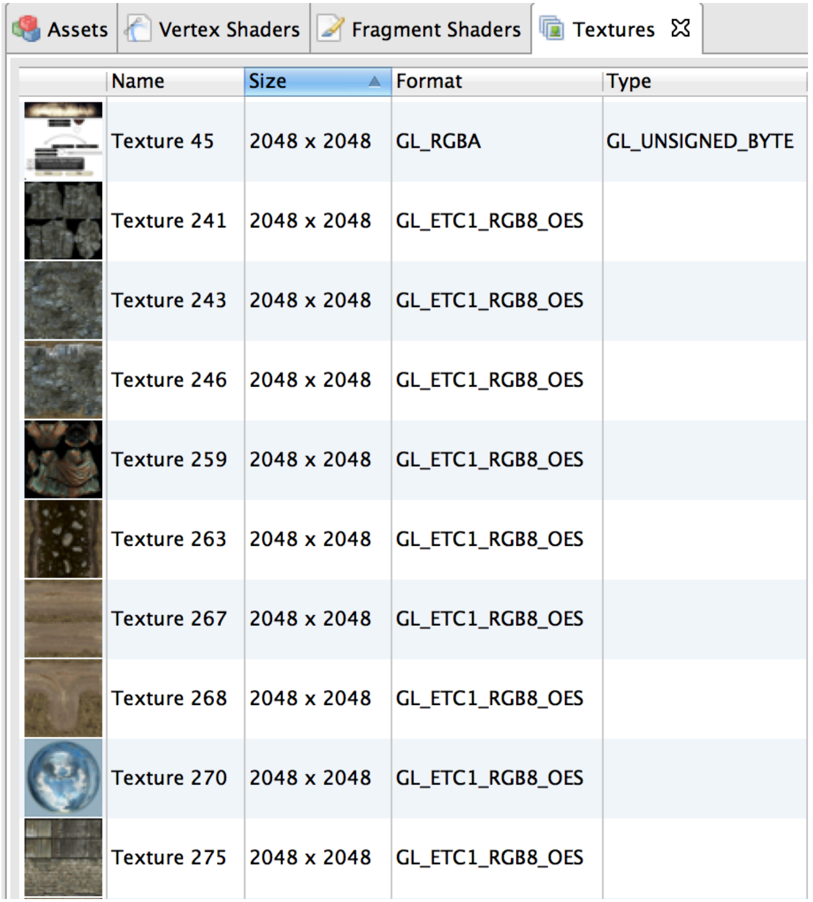
Textures

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







Name	Size	Format	Type
Texture 45	2048 x 2048	GL_RGBA	GL_UNSIGNED_BYTE
Texture 241	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 243	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 246	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 259	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 263	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 267	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 268	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 270	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 275	2048 x 2048	GL_ETC1_RGB8_OES	

ARM

Transaction Elimination

Helps reduce bandwidth consumption

This technology prevents the game from wasting bandwidth while still utilizing GPU resources to render tiles that haven’t changed from previous frames.

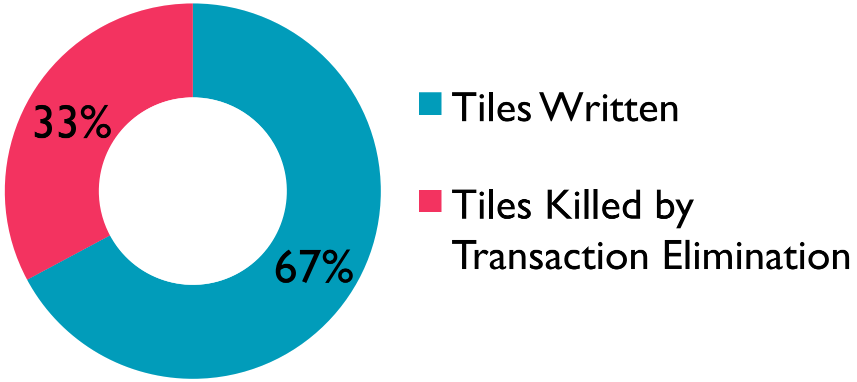
- Every time the GPU resolves a tile-full of color samples, it computes a signature
- Each signature is written into a list associated with the output color buffer
- The next time it renders to that buffer, if the signature hasn't changed, it skips writing out the tile

More about Transaction Elimination here:
http://community.arm.com/groups/arm-mali-graphics/blog/2012/08/17/how-low-can-you-go-building-low-power-low-bandwidth-arm-mali-gpus

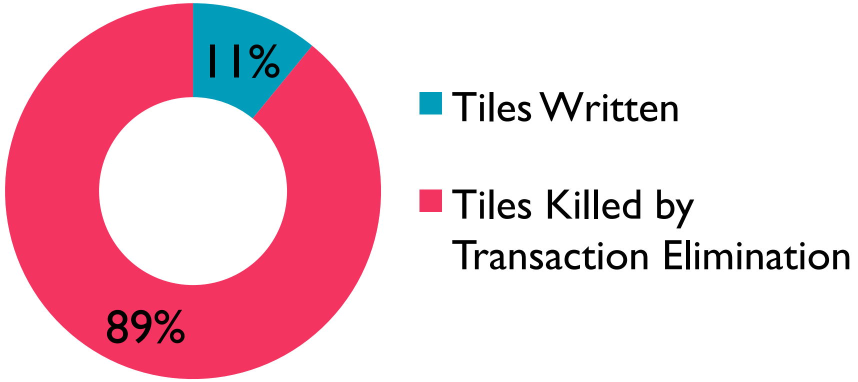
ARM

Transaction Elimination

Camera moving in the scene

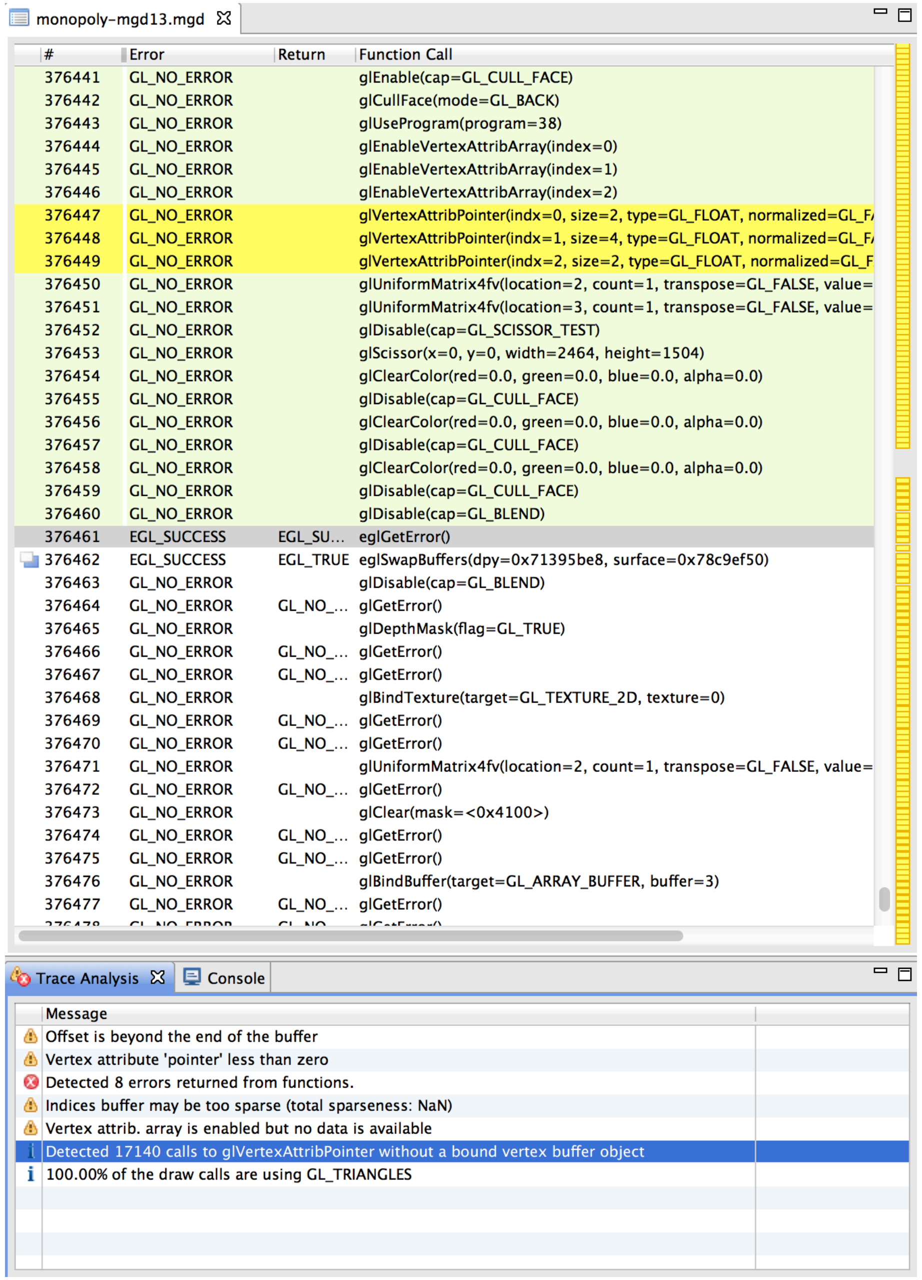


Loading screen



Vertex Buffer Objects

- Using Vertex Buffer Objects (VBOs) can save you a lot of time in overhead
- Every frame in your application, all of your vertices and colour information will get sent to the GPU
- A lot of the time these won't change. So there is no need to keep sending them
- Would be a much better idea to cache the data in graphics memory



ARM

Summary

- Covered today:
  - Introduction to performance analysis
  - Software Profiling
  - GPU Profiling
  - Debugging with the ARM® Mali™ Graphics Debugger
- For more information:
  - www.malideveloper.arm.com
  - www.ds.arm.com
  - www.community.arm.com

ARM

Thank You

Any Questions?