ARM® Mali™ Development Tools, GPU, and 64-bits

Nathan Li – SGL of DevRel, Ecosystem, ARM Shanghai

CGDC 2014

Agenda

1. Introduction of ARM® Mali™ Development Tools and Performance Profiling
   - DS-5 Streamline™ Performance Profiler and Mali Graphics Debugger
   - Using the ARM Mali GPU hardware counters to find the bottlenecks
   - CPU bound, Vertex bound, Fragment bound, and Bandwidth bound cases
   - Using MGD - an Overdraw and Frame Analysis case study (Epic Citadel)
2. Mali GPU Technologies for Game Developers
   - Compressions - ASTC and AFBC
   - Transaction Elimination
   - Smart Composition
   - Pixel Local Storage
3. 64-bit CPU & GPU synergy
4. Q & A

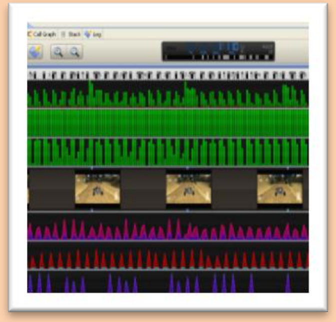
Importance of Analysis & Debug

- 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 Tools

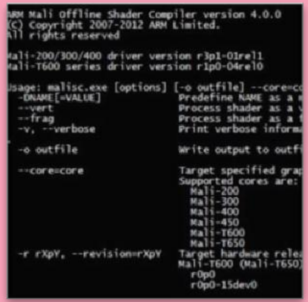


ARM® DS-5 Streamline™ Performance Analyzer

- 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

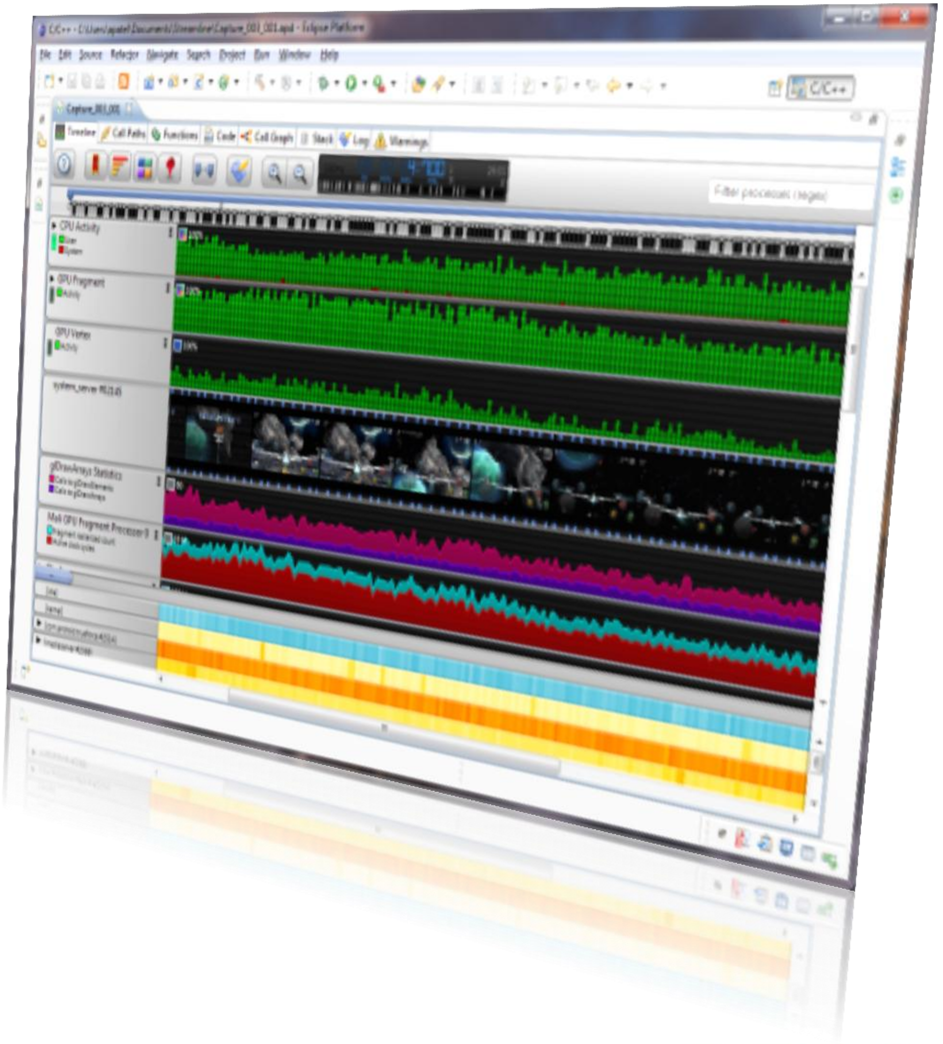


ARM Mali GPU Offline Compiler

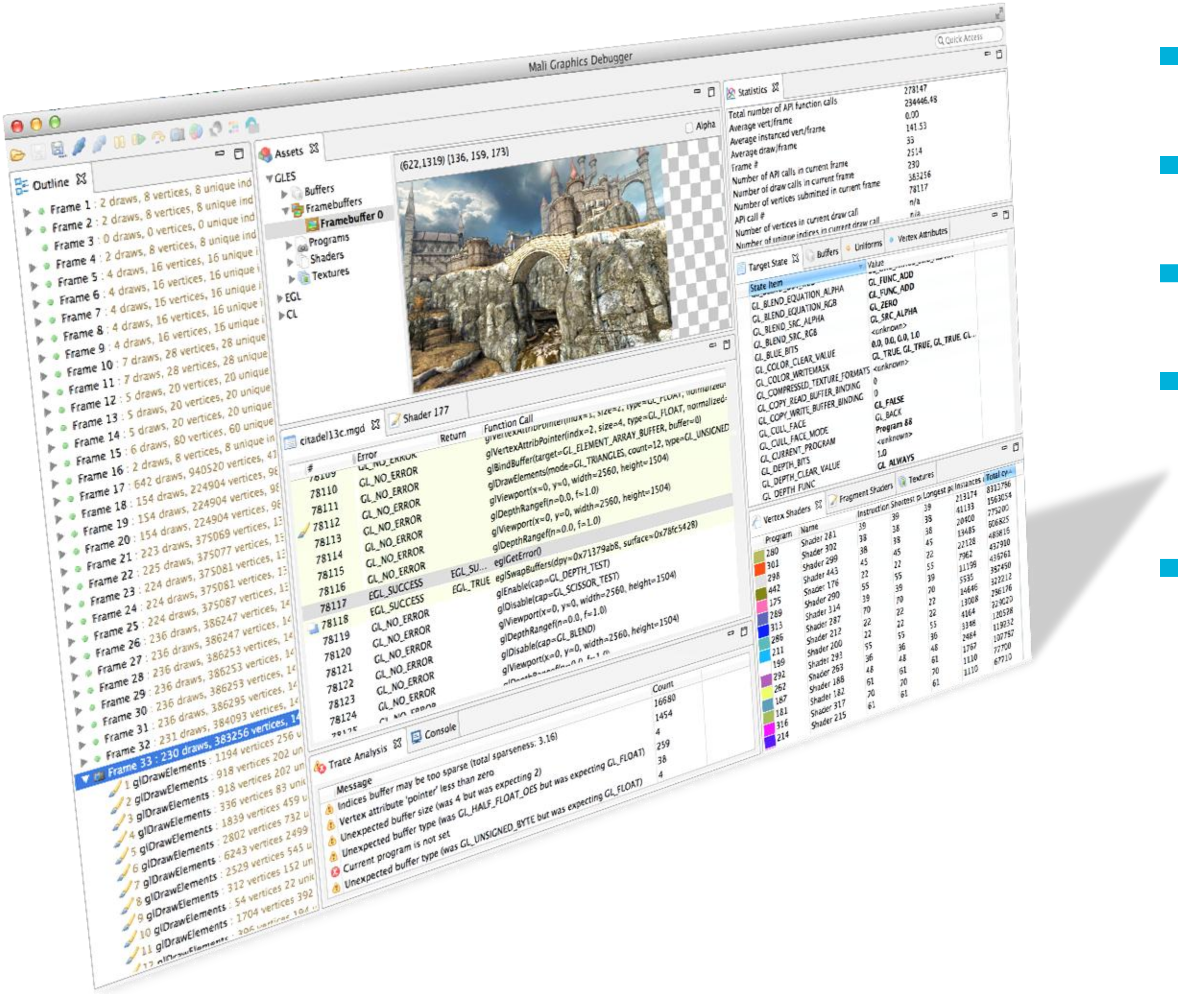
- Understand complexity of GLSL shaders and CL kernels
- Support for ARM Mali-4xx and Mali-T6xx GPU families

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
- What's New in 5.19
  - Early access for Mali-V500 support
  - User Space Gator now supports kernels 3.4 and later



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 released in July, 2014

ARM

Mali™ Graphics Debugger v1.3 Update

New features:

- Frame replay
  - Replay the same frame in different modes: overdraw, fragment count, etc.
- New binary data format
  - Faster tracing and smaller files
  - 5-10x speed improvement
- Memory performance improvements
  - We can capture 20M+ calls
- Better support for OpenGL® ES 3.0
  - We can now trace GFXBench 3.0
- Bug fixes

Latest release (July 2014):
Mali Graphics Debugger v1.3

ARM

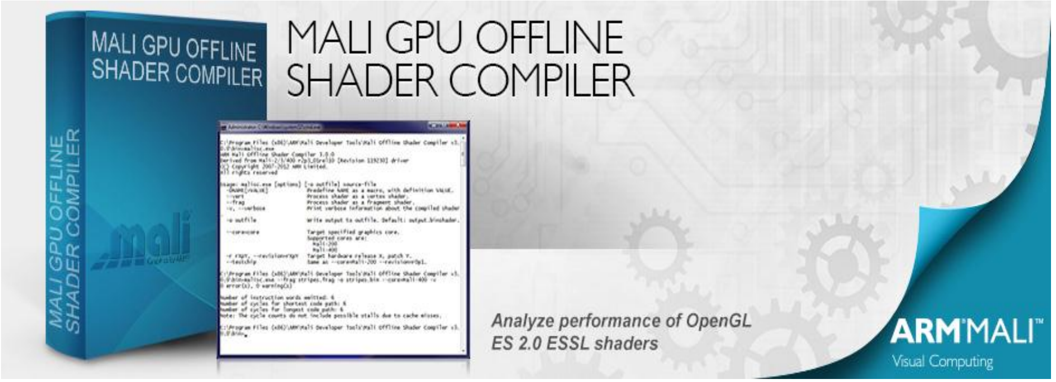
Mali™ Offline Shader Compiler v4.3 Update

Last release:

- We have added support for the Mali compiler version r4p0 of the 'Midgard' series
- This version introduces bug fixes and performance optimization
- We are now supporting Mac OS X again, additionally to Windows and Linux

Next releases:

- Mali Compiler r4p1 with Mali-T700 support
- Mali Compiler r4p0-rel01 for Mali-450



Last release (July 2014):
Mali Offline Shader Compiler v4.3

Mali™ OpenGL® ES Emulator v1.4 Update

The OpenGL ES Emulator is a library that maps OpenGL ES API calls to the OpenGL API. It supports OpenGL ES 2.0 and 3.0, plus additional extensions.

In this release we have implemented:

- Single library with EGL/OpenGL ES
- Improved textures support
- Providing Mali-cube executable for installation verification
- Debian Software Package (.deb) now available for Ubuntu

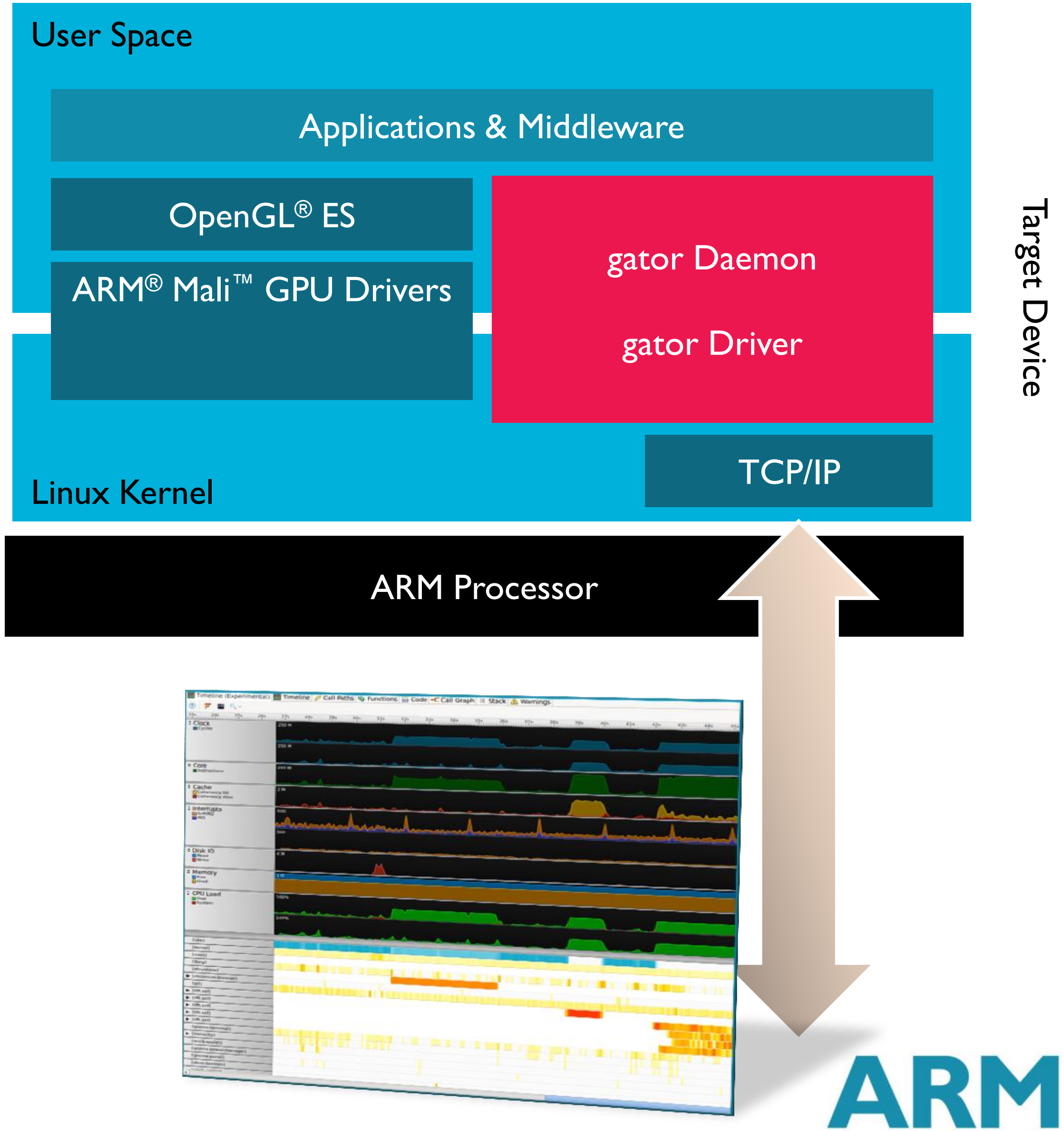
The source code of the emulator has also been completely refactored by our engineers.

Last release (July 2014):
Mali OpenGL ES Emulator v1.4

ARM

The Basics

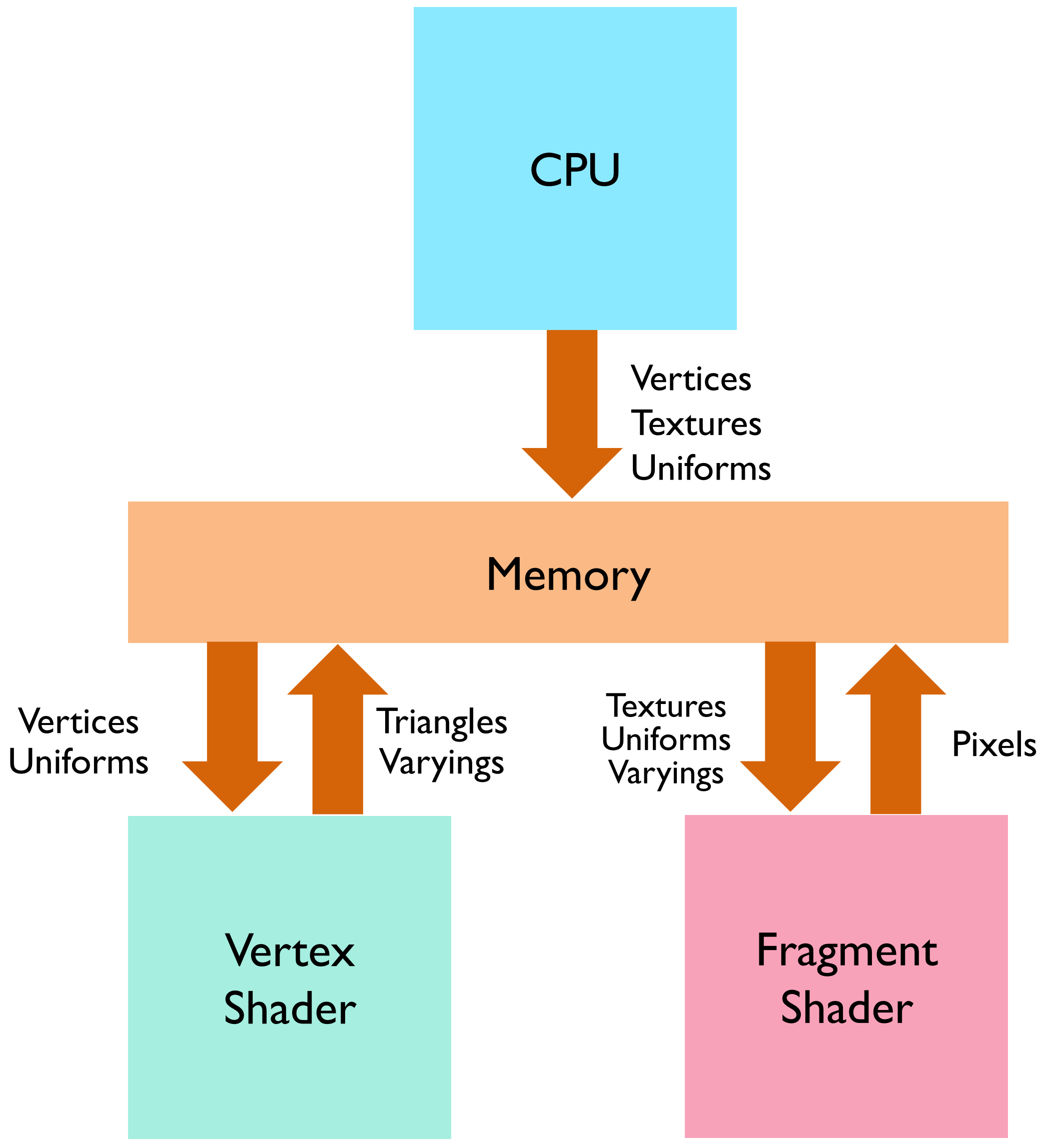
- Software based solution
  - ICE/trace units not required
  - Support for Linux kernel 2.6.32+ on target
  - Eclipse plug-in or command line
- Lightweight sample profiling
  - Time- or event*-based sampling
  - Process to C/C++ source code profiler
  - Low probe effect; <5% typically
- Multiple data 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

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



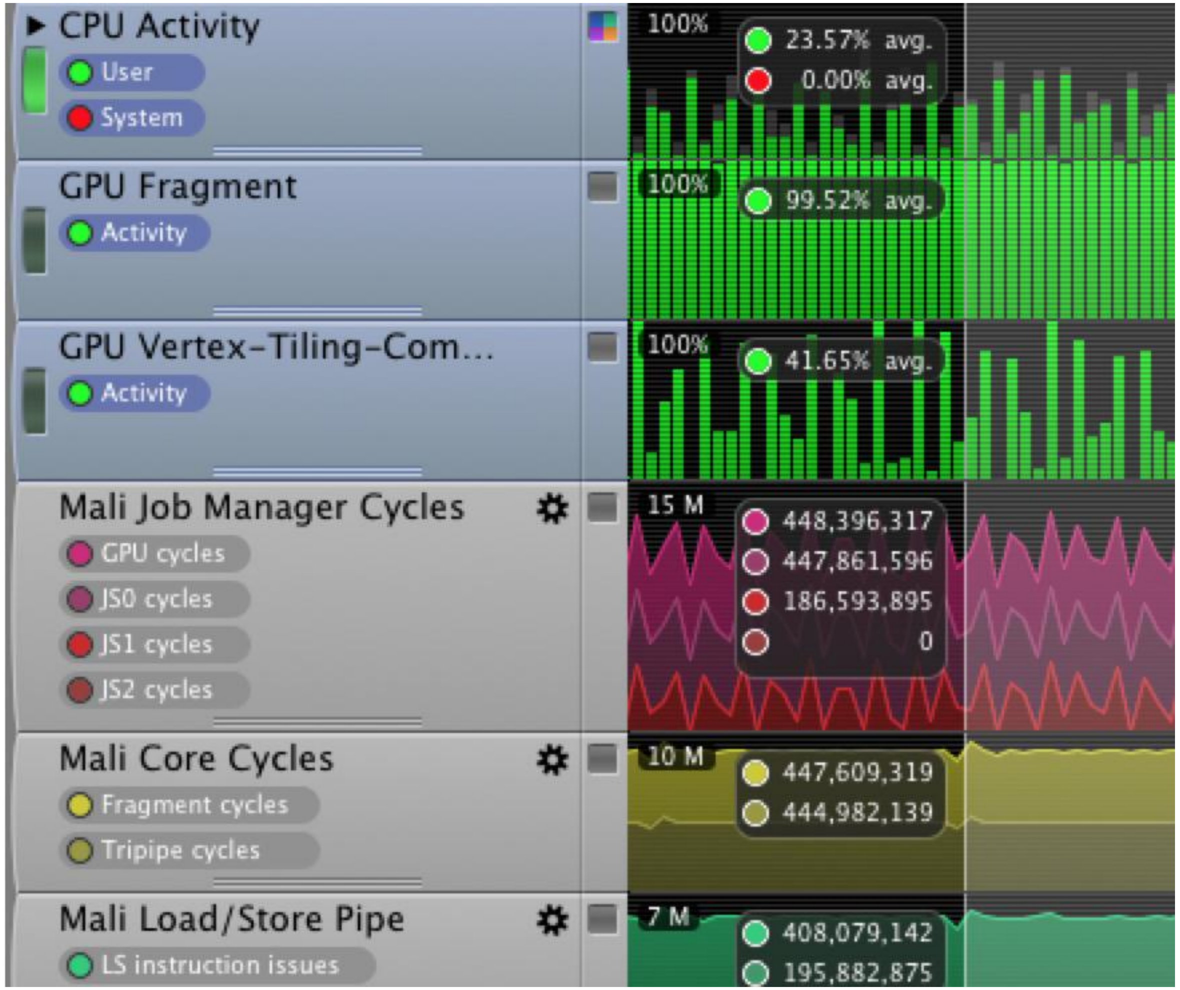
ARM

“Epic Citadel” – A Case Study

ARM

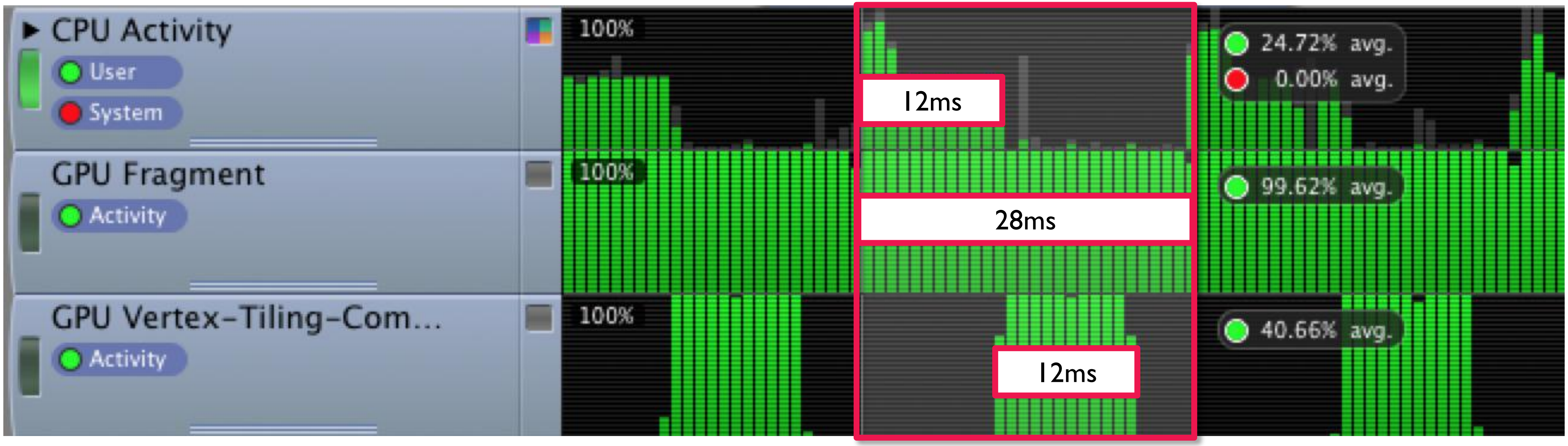
Profiling via ARM® DS-5 Streamline™

- 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

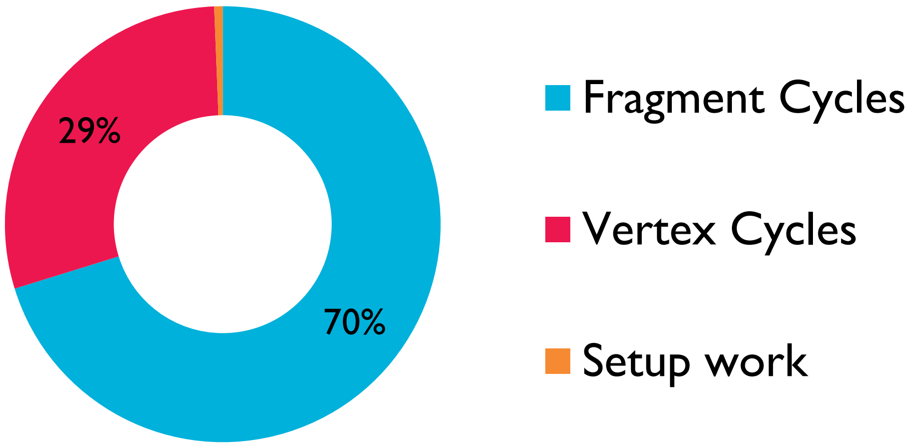
The CPU has to wait until the fragment processing has finished



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*)

Fragment Count Per Program

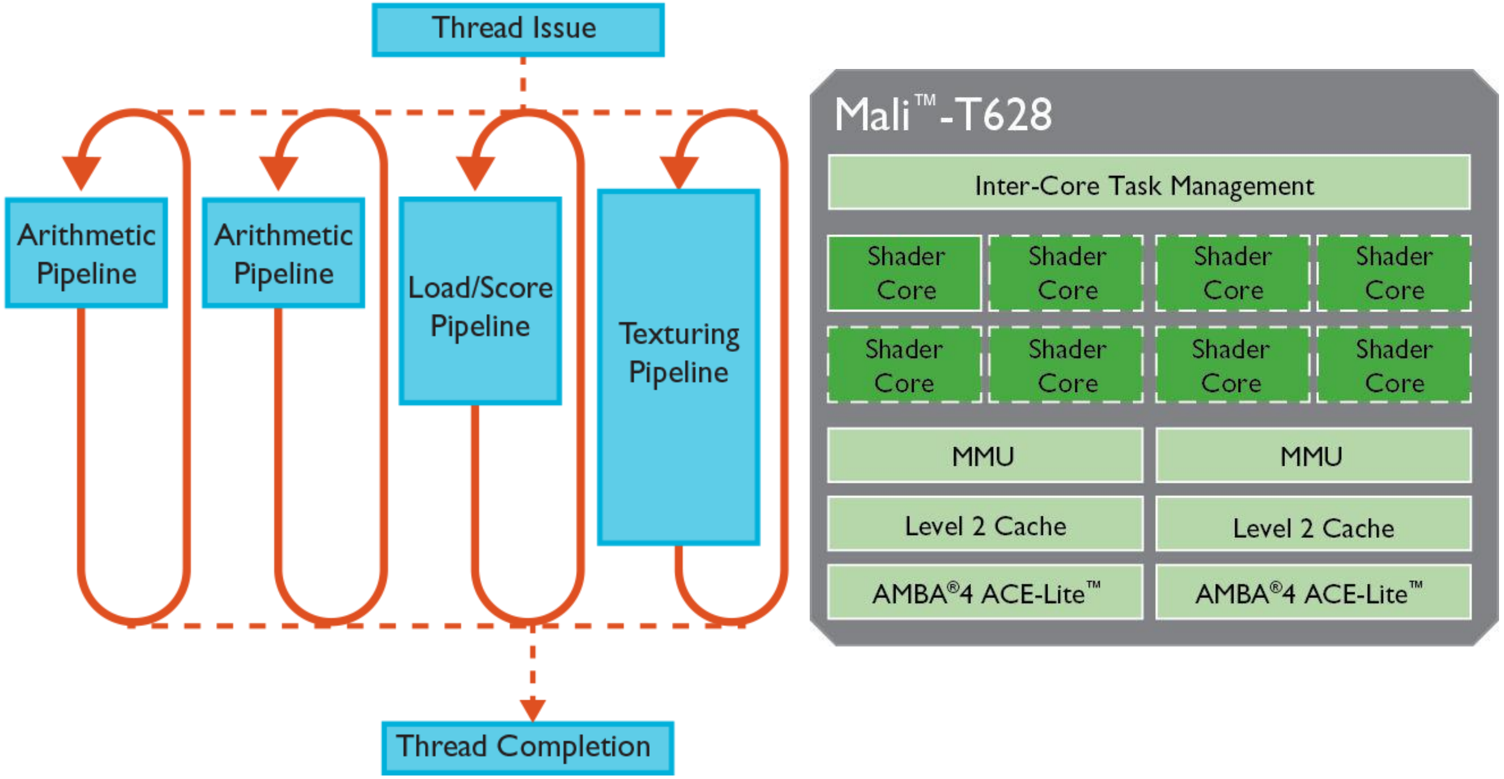


There might be an overhead in the job manager trying to optimize vertex list packing into jobs.

ARM

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

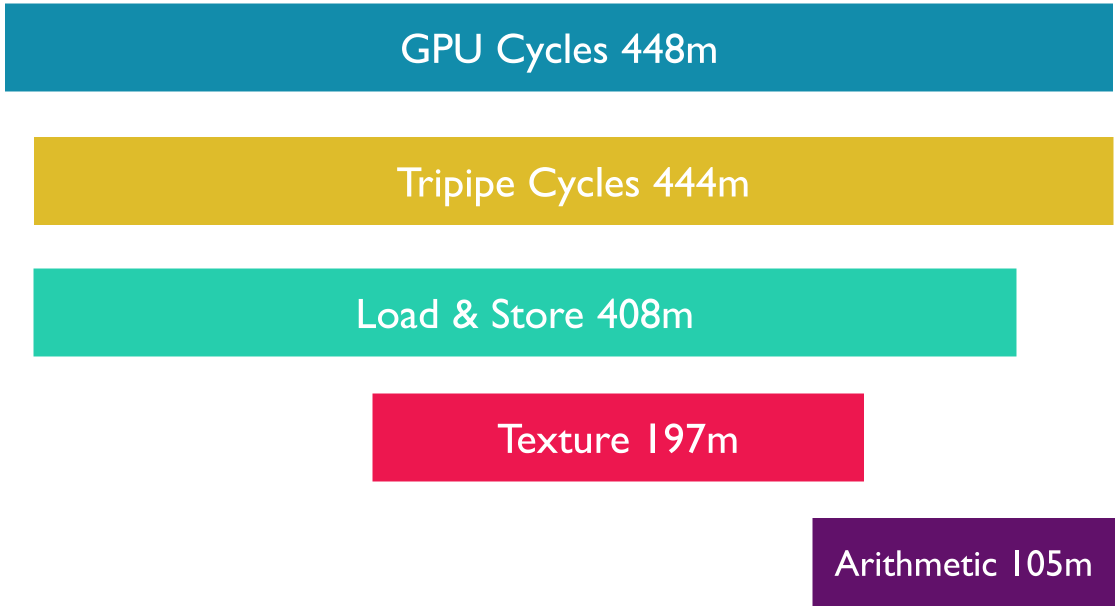


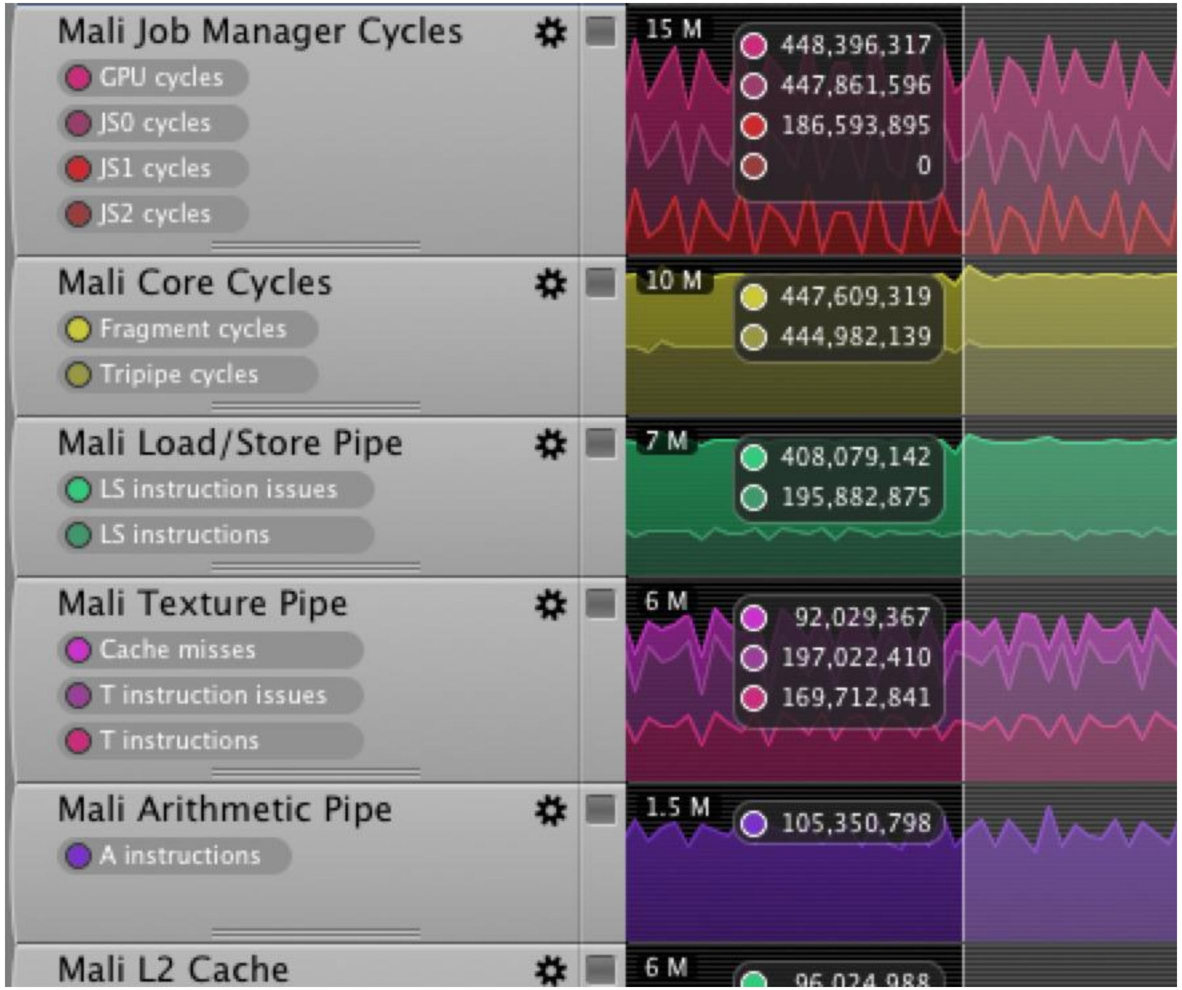
- Instructions can run in parallel
- Each one can be a bottleneck
- There are two arithmetic pipelines so we should aim to increase the arithmetic workload

ARM

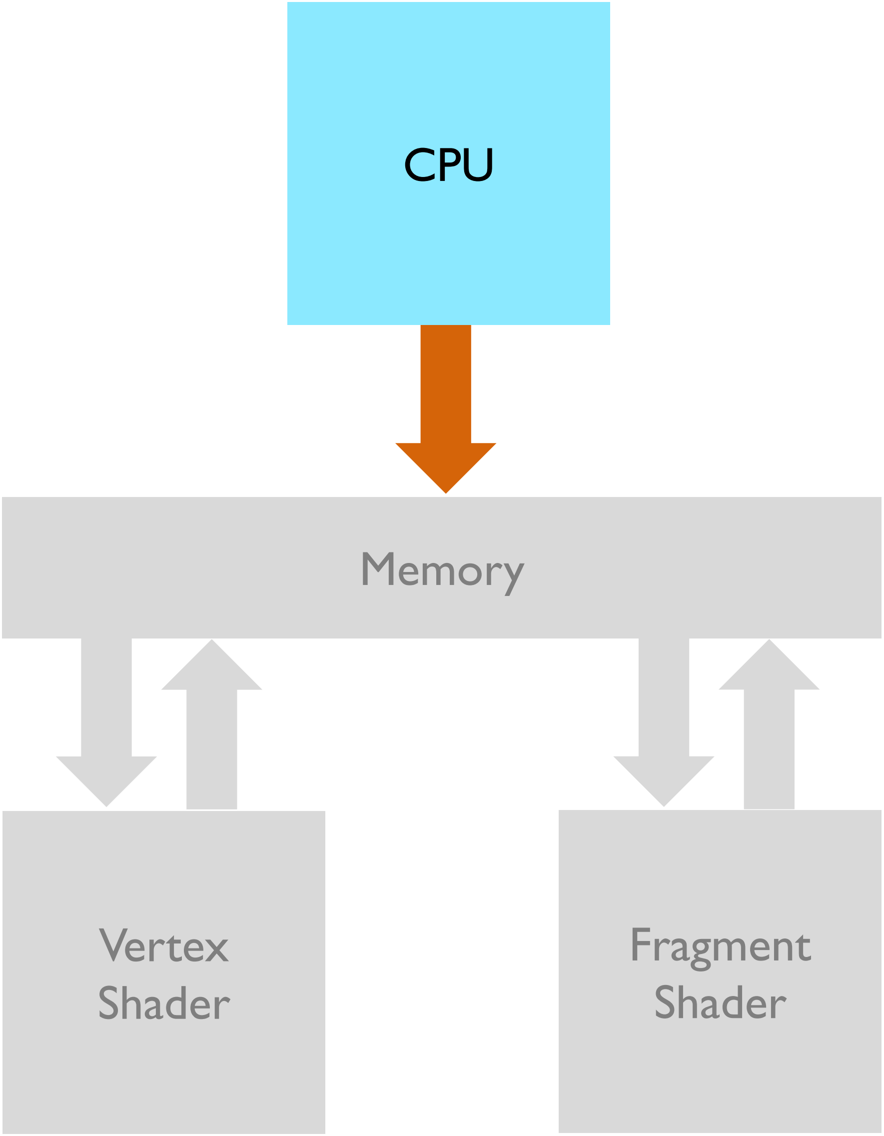
Inspect the Tripipe Counters

Reduce the load on the L/S pipeline





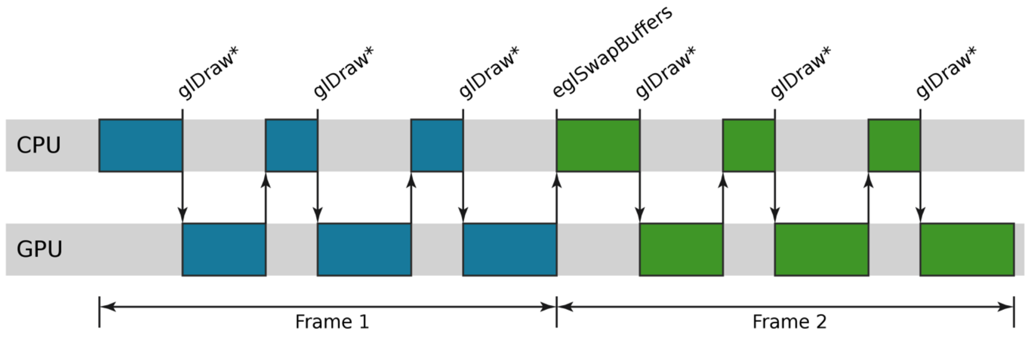
CPU Bound



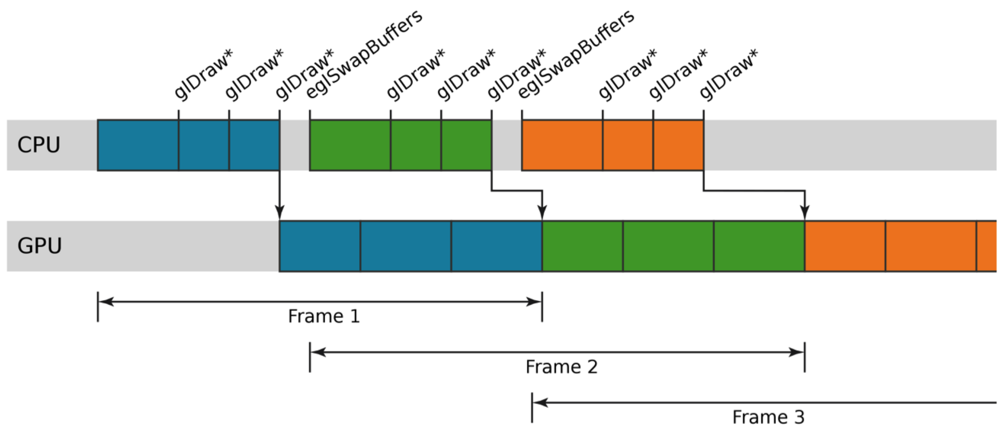
CPU Bound

- 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

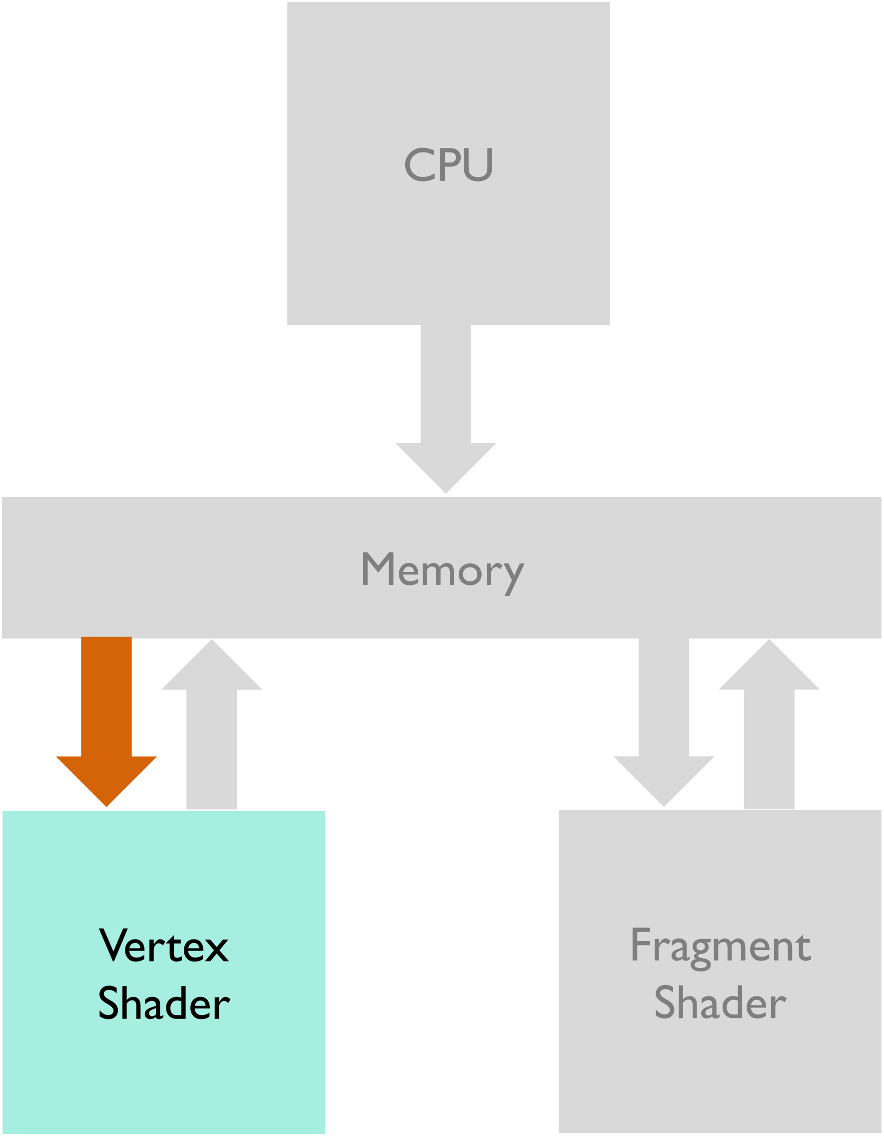


Deferred Rendering



ARM

Vertex Bound



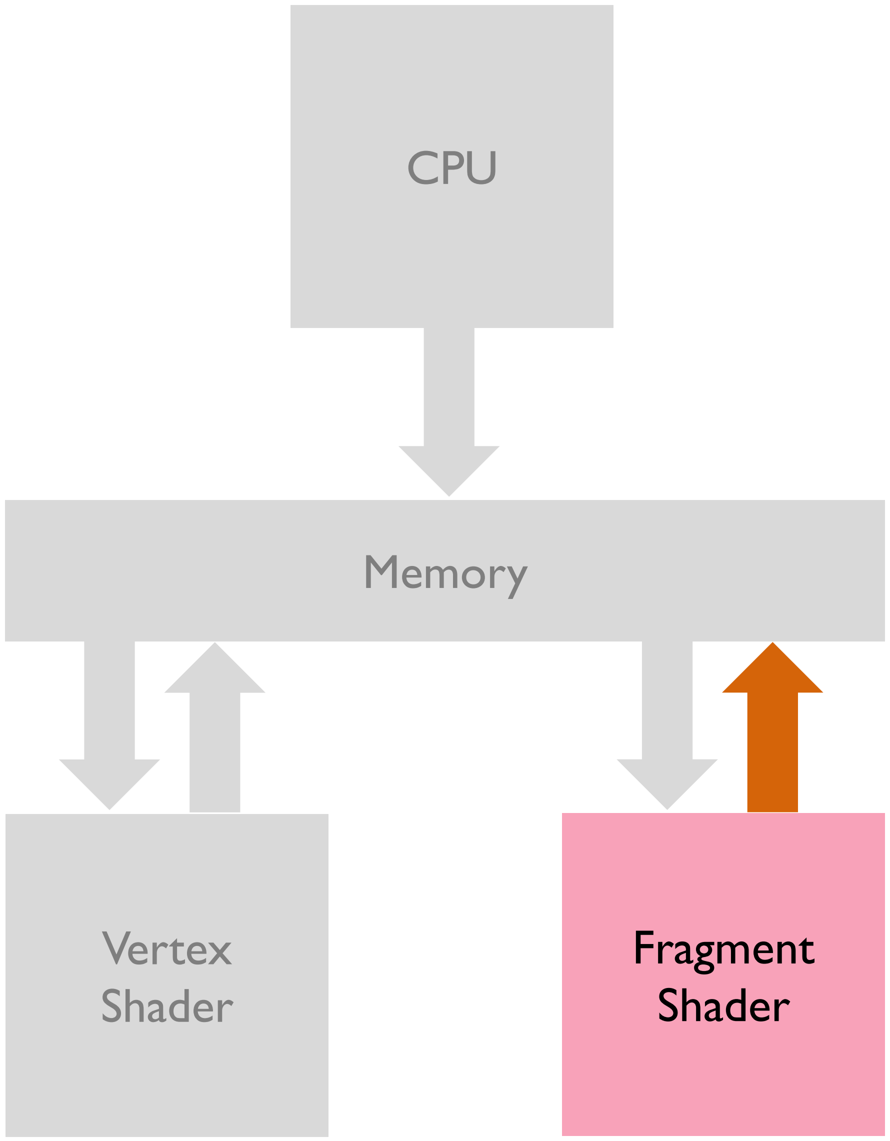
Vertex Bound

- Get your artist to remove unnecessary vertices
- LOD switching
  - Only objects near the camera need to be in high detail
  - Tessellation is still not popular on Mobile, and there are alternatives
- Use culling
  - The earlier in the pipeline, the better
- 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

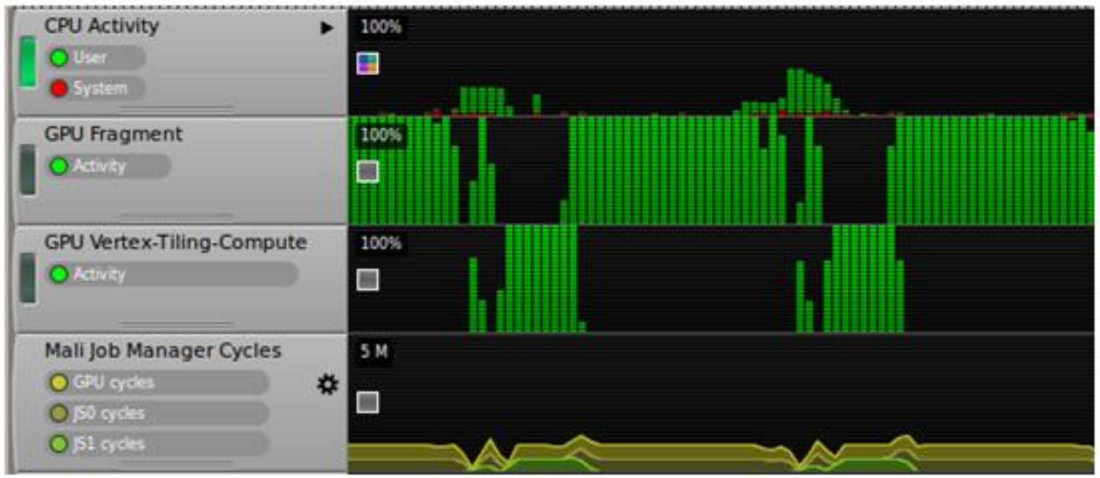
Fragment Bound



ARM

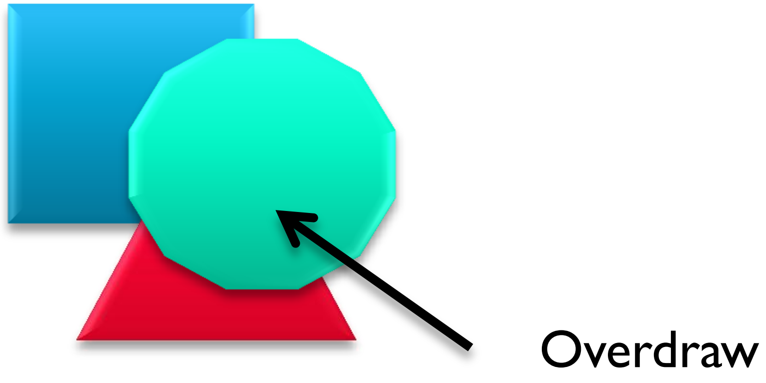
Fragment Bound

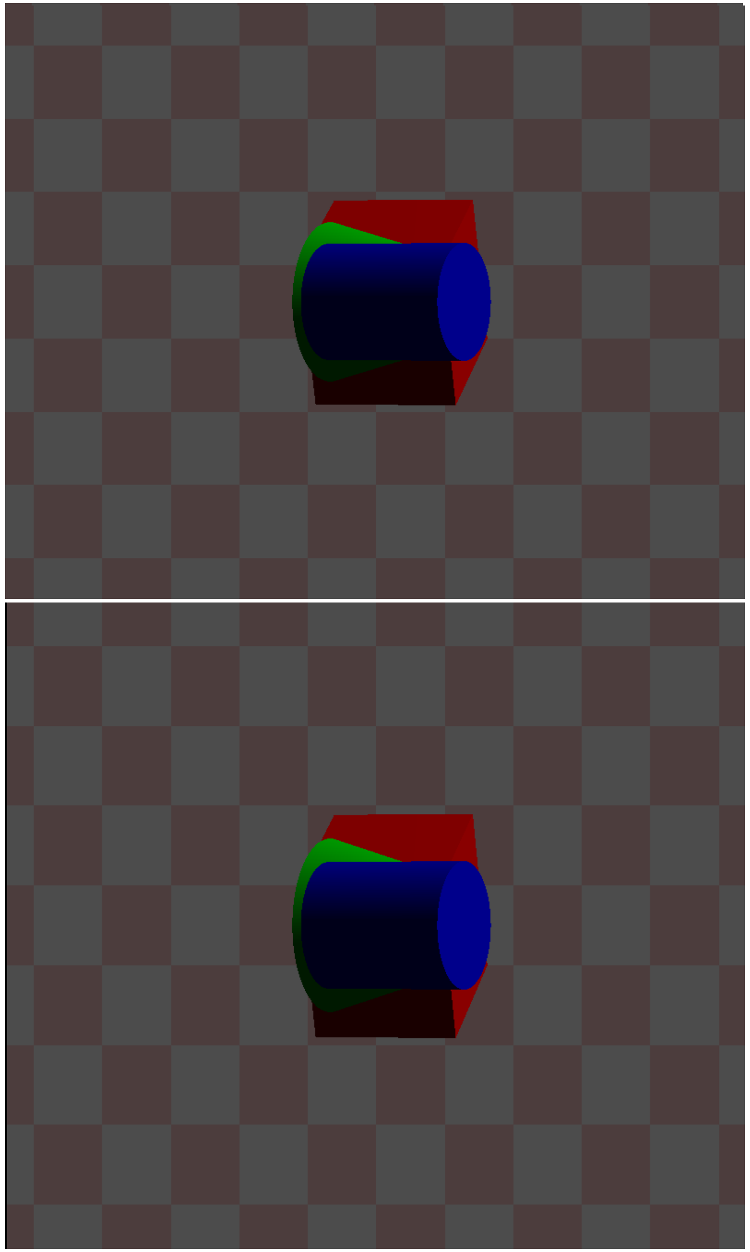
- Render to a smaller framebuffer
- Move computation from the fragment to the vertex shader (use HW interpolation)
- Drawing your objects front to back instead of back to front reduces overdraw
- Reduce the amount of transparency in the scene



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

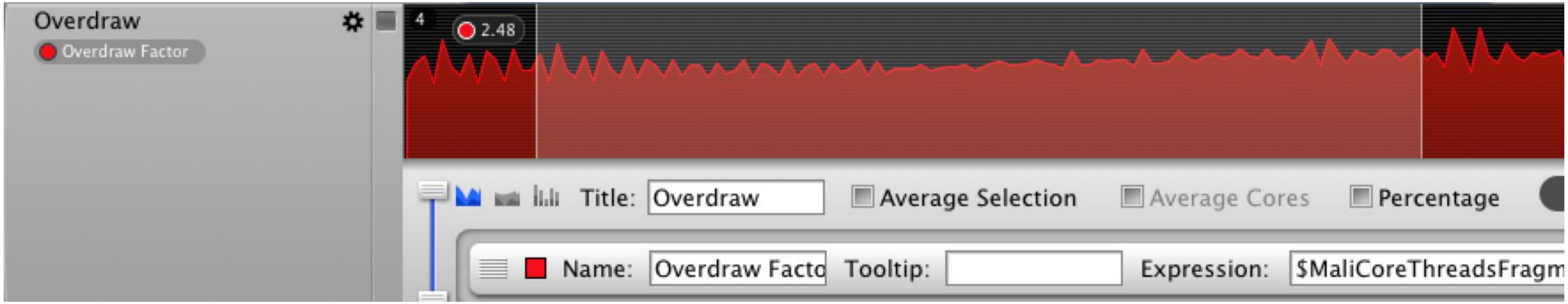




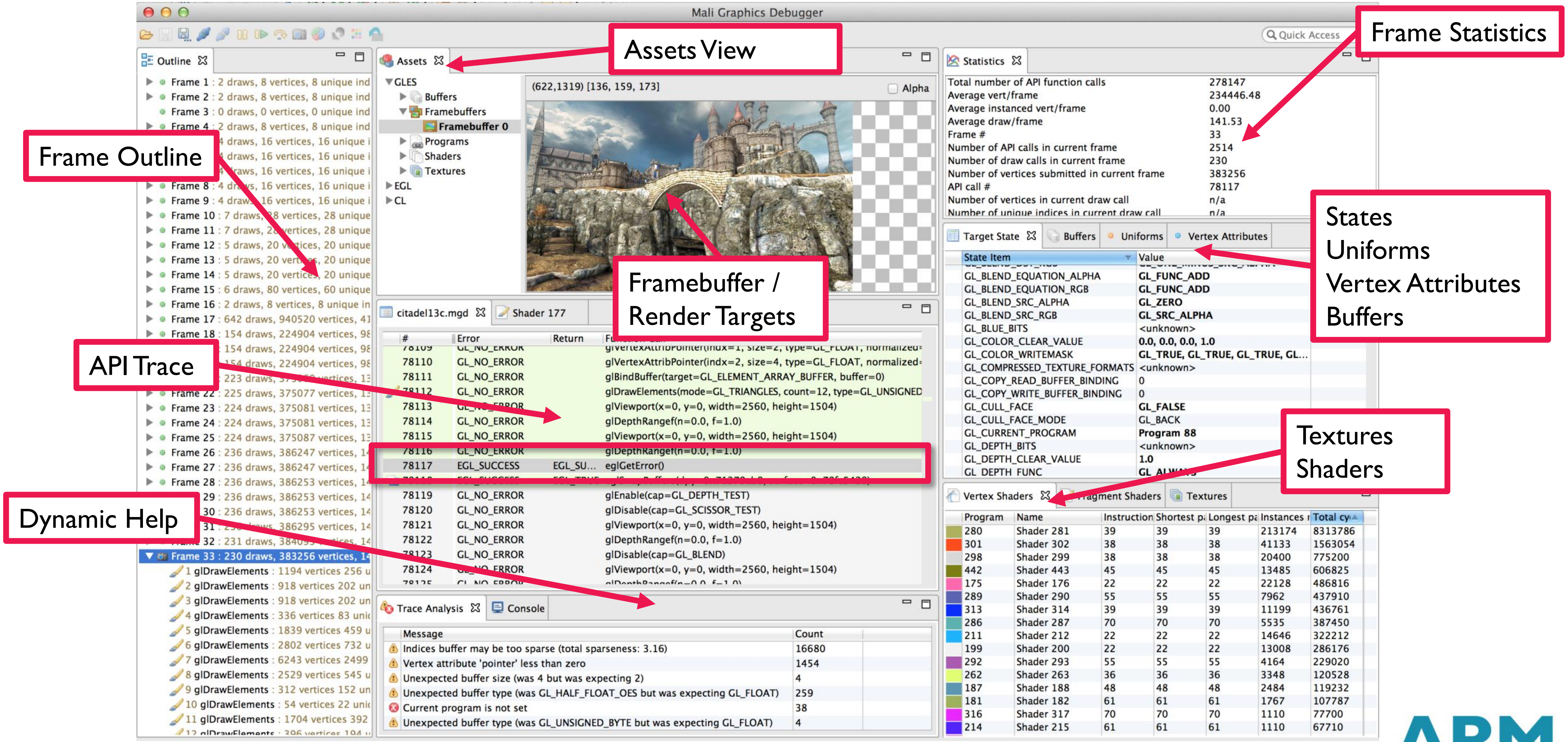
Overdraw Factor

- 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



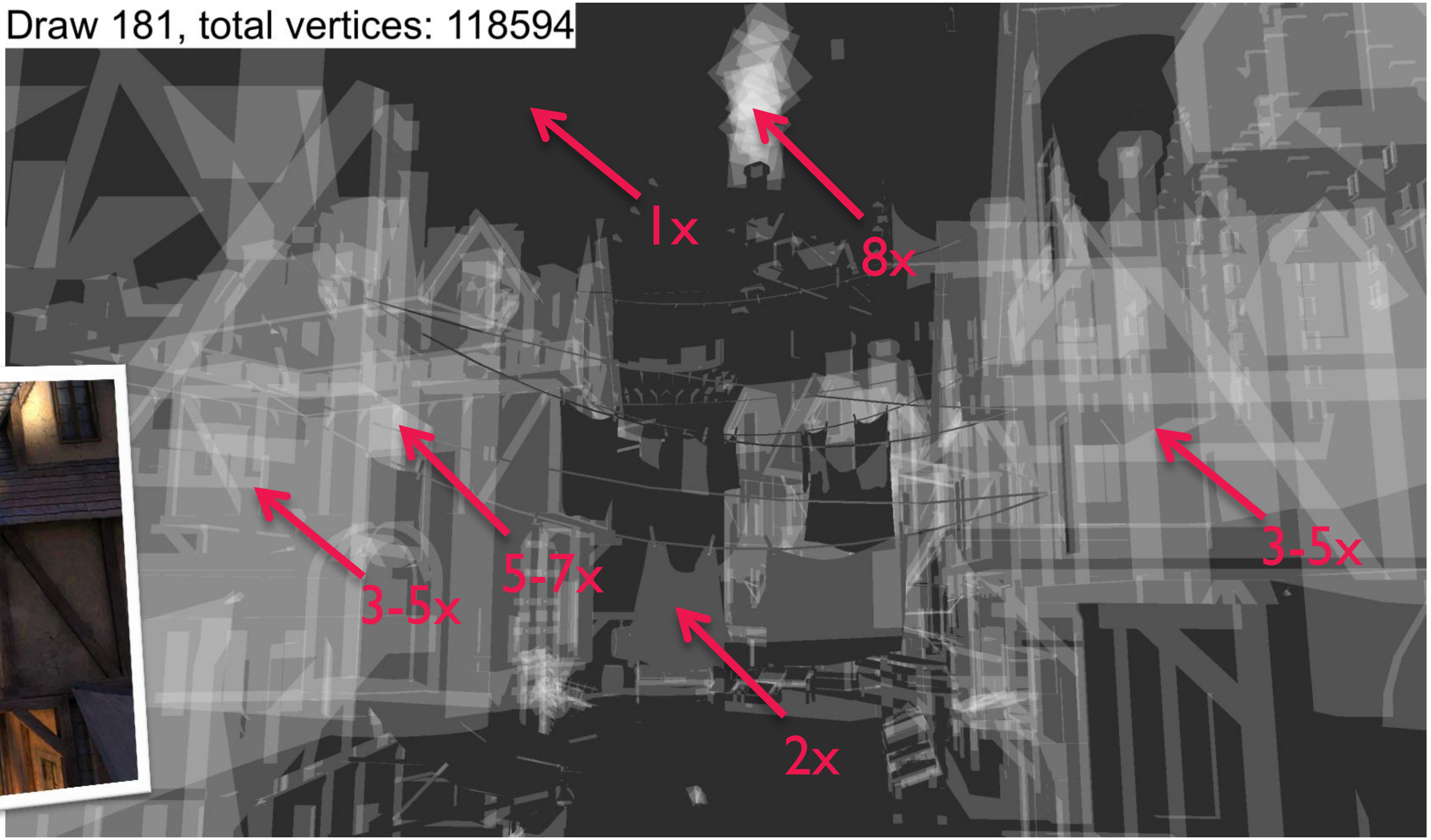
Investigation with the ARM® Mali™ Graphics Debugger



ARM

Frame Analysis

Check the overdraw factor



Shader Map and Fragment Count

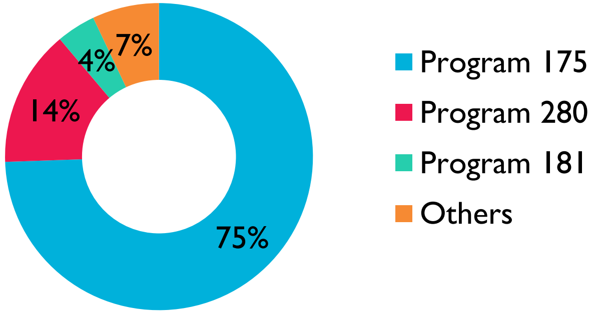
Identify the top heavyweight fragment shaders

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?
- Considering the Differences between GPUs (i.e., Mali™-450 or Mali-T764)
- Use the Mali Offline Shader Compiler!

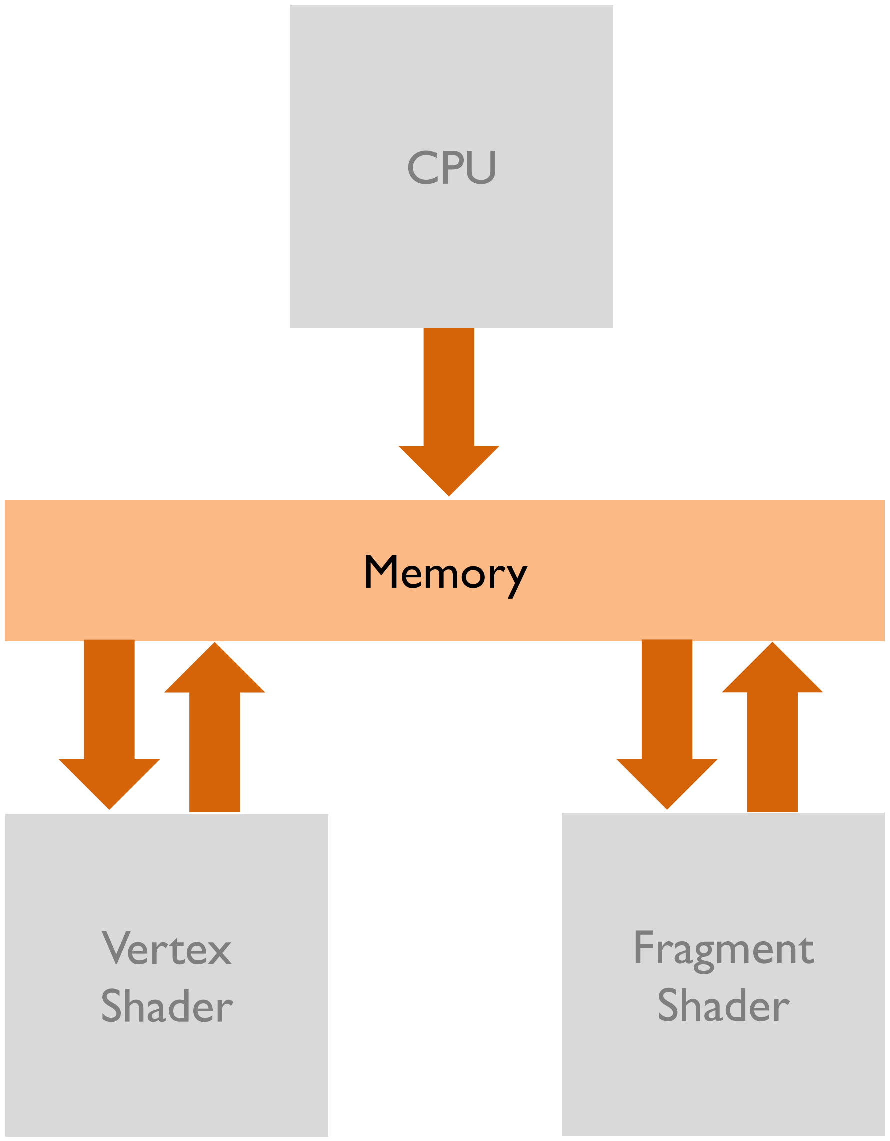
  http://malideveloper.arm.com/develop-for-mali/tools/analysis-debug/mali-gpu-offline-shader-compiler/

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

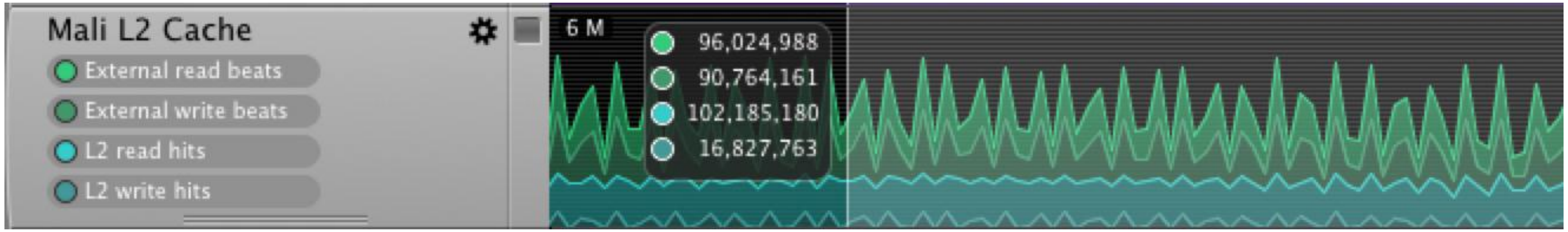
Bandwidth

- 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

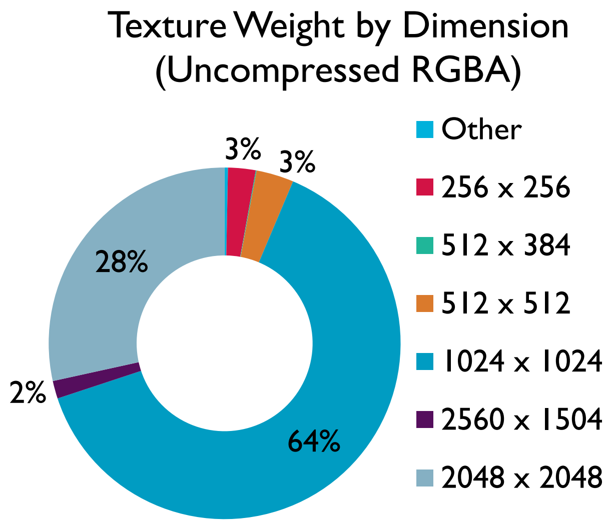


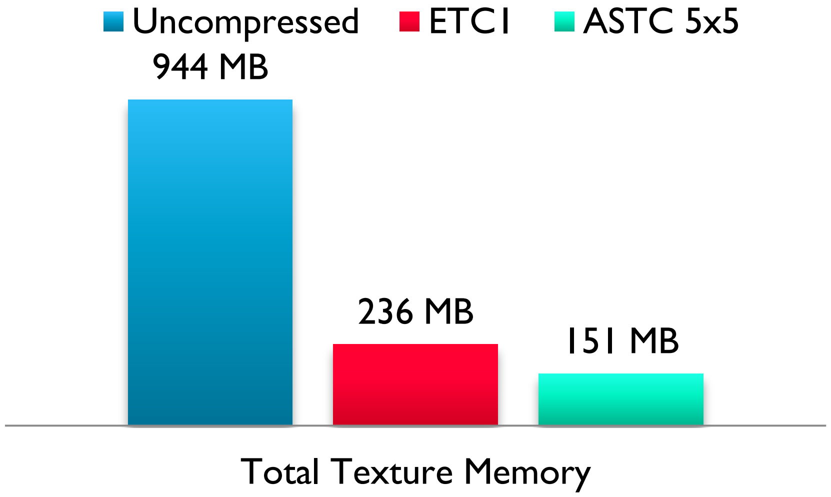
ARM

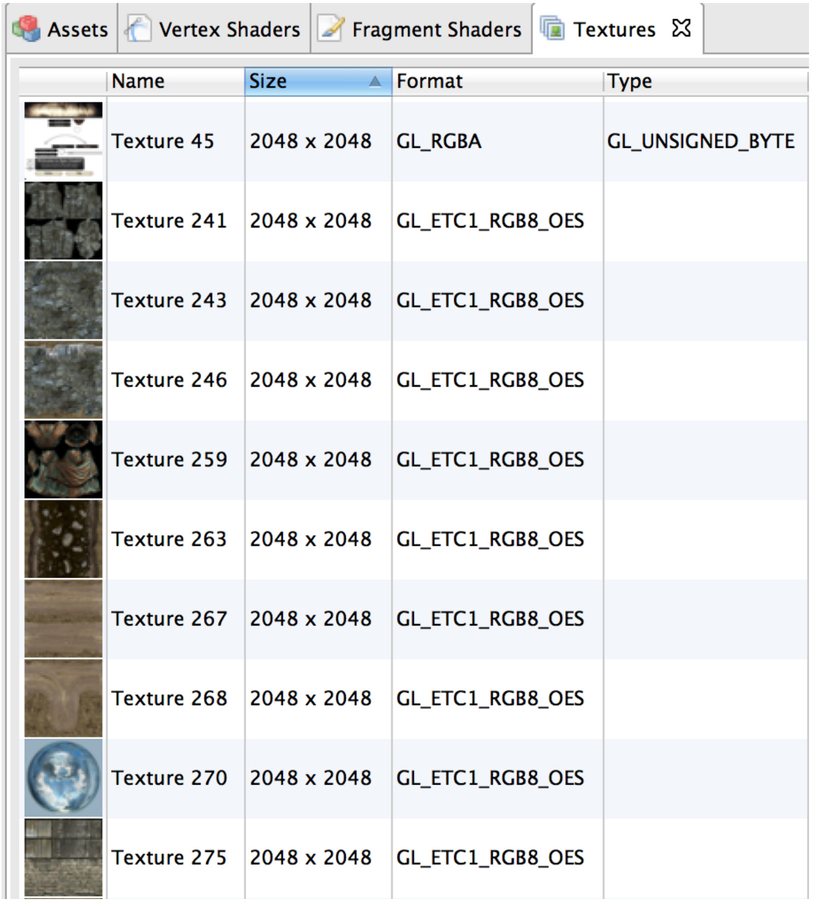
Textures

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





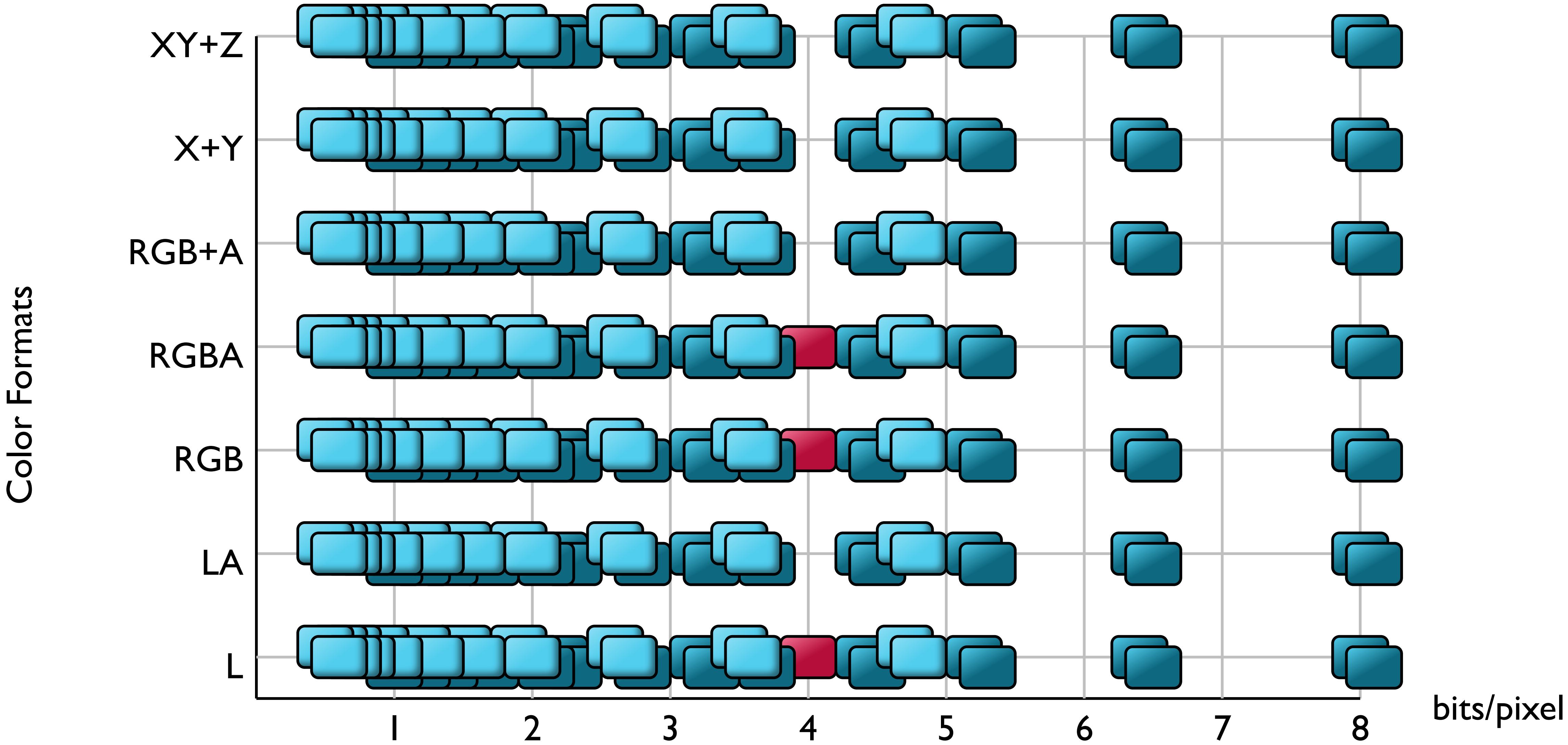


Name	Size	Format	Type
Texture 45	2048 x 2048	GL_RGBA	GL_UNSIGNED_BYTE
Texture 241	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 243	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 246	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 259	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 263	2048 x 2048	GL_ETC1_RGB8_OES	
Texture 267	2048 x 2048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

Adaptive Scalable Texture Compression (ASTC)

- Now it is an official extension to both the OpenGL® and OpenGL ES graphics APIs.
- A major step forward in terms of image quality, reducing memory bandwidth and thus energy use.
- ASTC offers a number of advantages over existing texture compression schemes:
  - Flexibility, with bit rates from 8 bits per pixel (bpp) down to less than 1 bpp. This allows content developers to fine-tune the tradeoff of space against quality.
  - Support for 1 to 4 color channels, together with modes for uncorrelated channels for use in mask textures and normal maps.
  - Support for both low dynamic range (LDR) and high dynamic range (HDR) images.
  - Support for both 2D and 3D images.
  - Interoperability. Developers can choose any combination of features that suits their needs.
- ASTC specification includes two profiles: LDR and Full.

ASTC – LDR + HDR + 3D



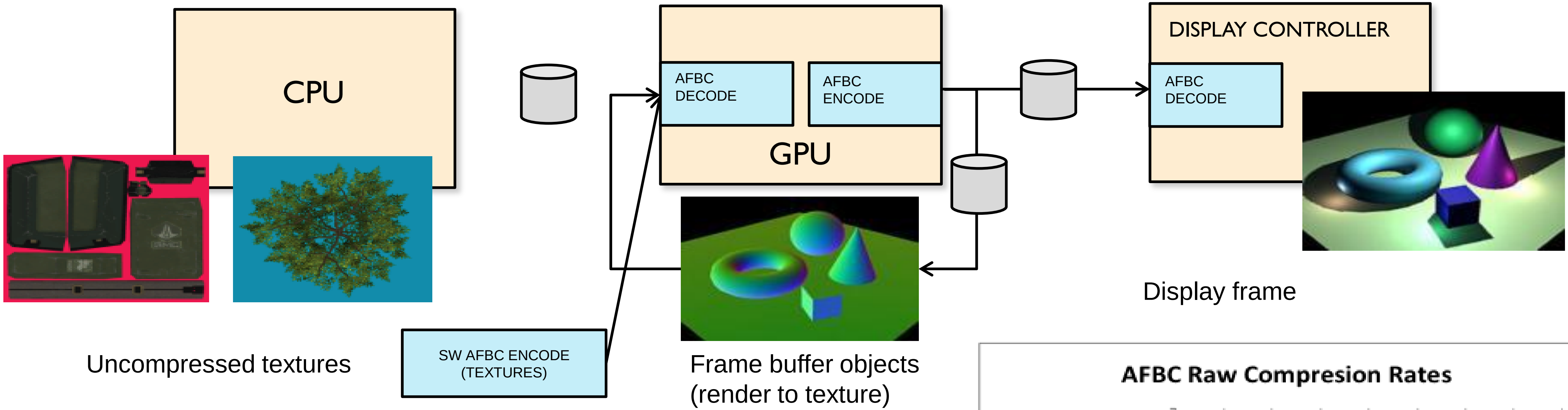
ARM

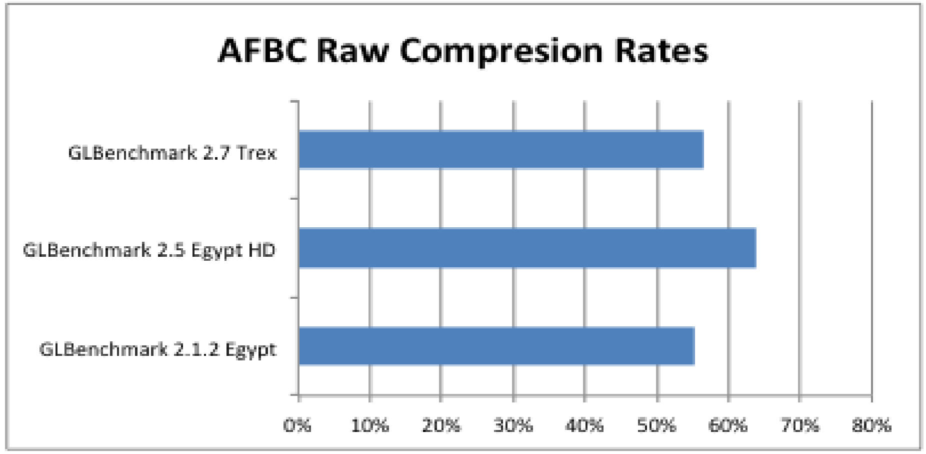
ARM® Frame Buffer Compression

- The ARM® Frame Buffer Compression (AFBC) protocol reduces the overall system level bandwidth and power cost of transferring spatially coordinated image data throughout the system by up to 50%.
- A lossless image compression protocol and format, AFBC minimizes the amount of data transferred between IP blocks within an SoC.
- ARM Frame Buffer Compression has the following properties:
  - Lossless data compression
  - Random access down to 4x4 block level
  - Bounded worst-case compression ratios
  - Support for both YUV and RGB formats
  - Compression ratios comparable to other lossless compression standards
  - YUV compression ratio of typically 50%

AFBC Application In SoC

- Employing AFBC throughout SoC saves significant system bandwidth and power





Transaction Elimination

Helps reduce bandwidth consumption

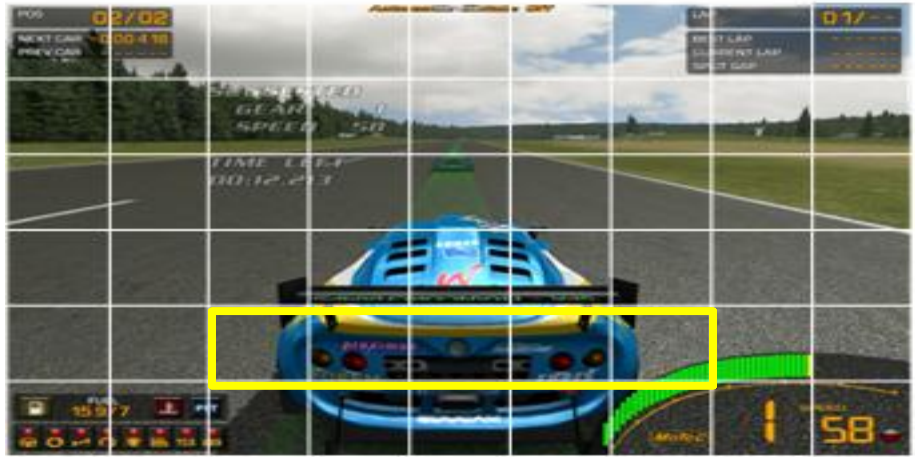
This technology prevents the game from wasting bandwidth while still utilizing GPU resources to render tiles that haven’t changed from previous frames.

- 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

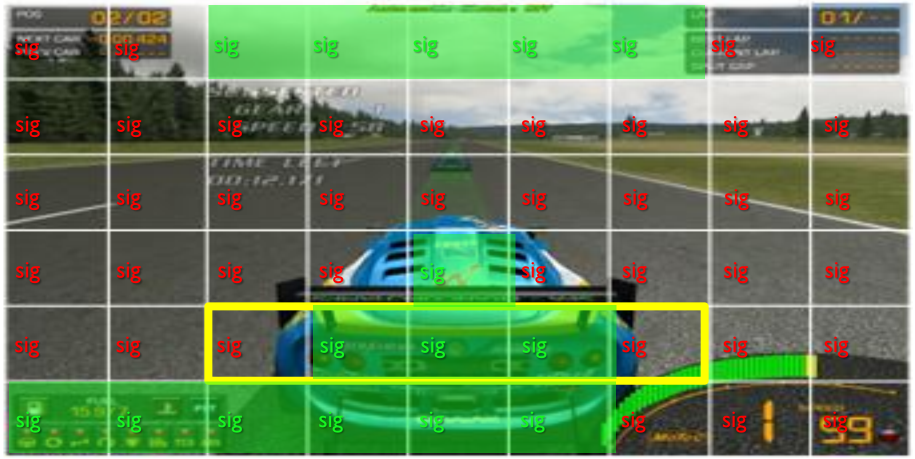
More about Transaction Elimination here:
http://community.arm.com/groups/arm-mali-graphics/blog/2012/08/17/how-low-can-you-go-building-low-power-low-bandwidth-arm-mali-gpus

Transaction Elimination

Frame N



Frame N+1



Maintain a list of signatures for each tile

…


…

Compare to sigs calculated for frame N+1

…

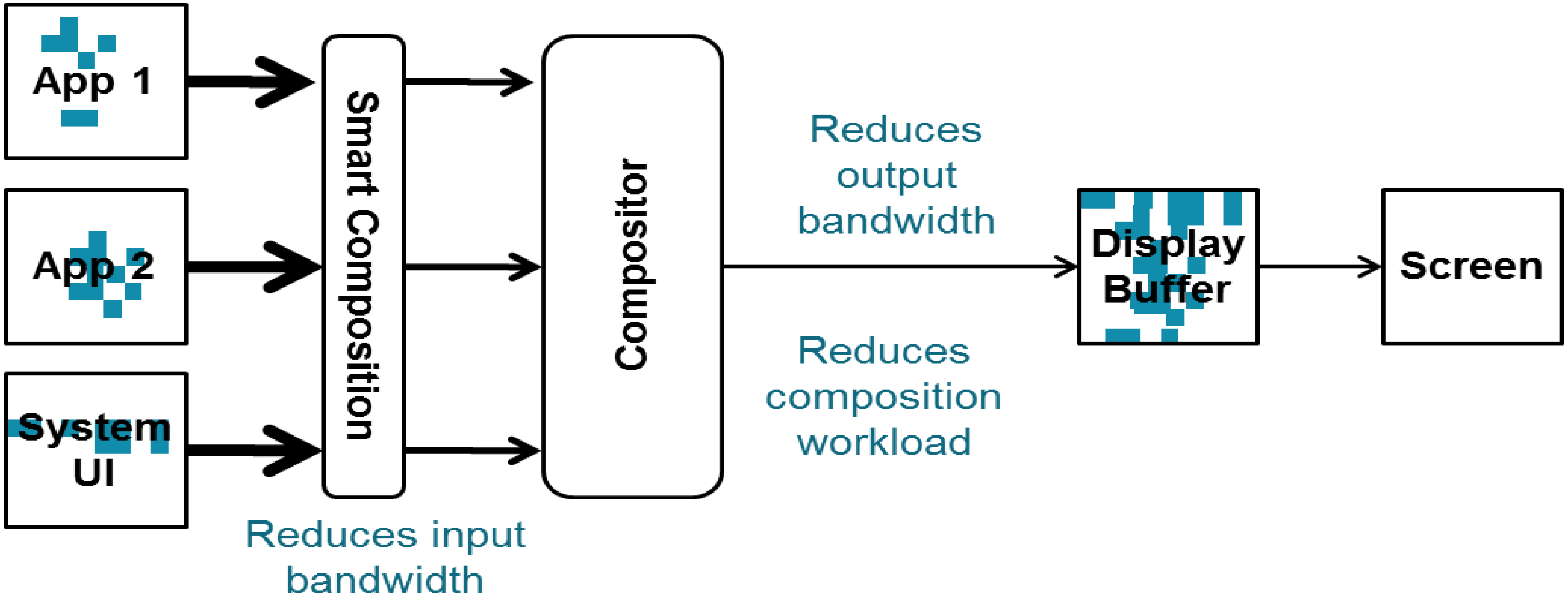

…

Where signatures match, don't write the tile

- Surprisingly effective, even on FPS games and video

Smart Composition

- Reduce standard Android™ User Interface texture read bandwidth by better than 50%.
- Significantly reducing read bandwidth and composition work by ignoring repetitive tile data.

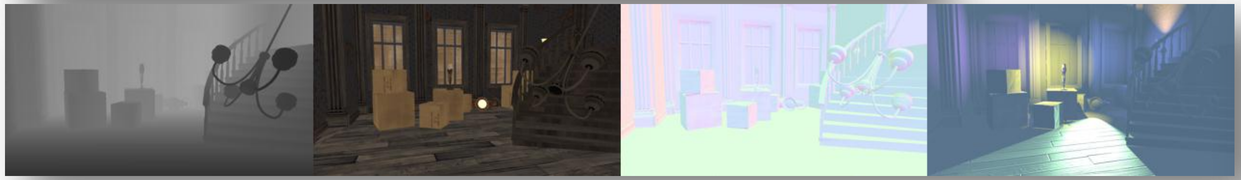


ARM

Pixel Local Storage on Mali™ GPUs

- Multi-Pass Rendering in OpenGL® ES:
  - Pass 1:

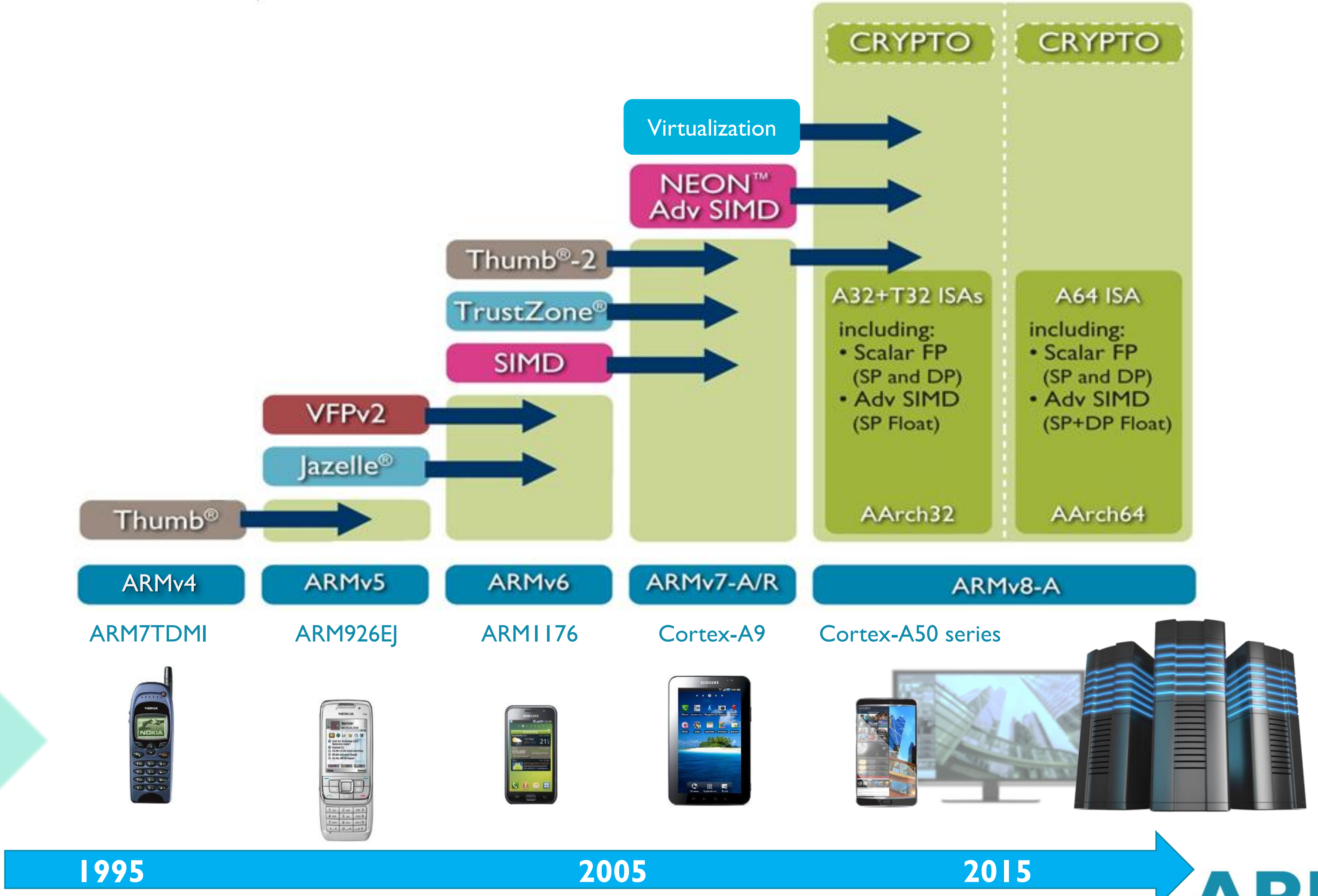
  - Pass 2:



  - Pass 3: the final rendering result is shown at the right side →
- EXT_shader_pixel_local_storage
  - enables applications to store custom data per pixel
- ARM_shader_framebuffer_fetch & ARM_shader_framebuffer_fetch_depth_stencil
  - return the current color, depth, and stencil values of a pixel to the fragment shader

More about Pixel Local Storage on Mali GPU:
http://community.arm.com/groups/arm-mali-graphics/blog/2014/04/01/pixel-local-storage-on-arm-mali-gpus

ARM® CPU Architecture Evolution



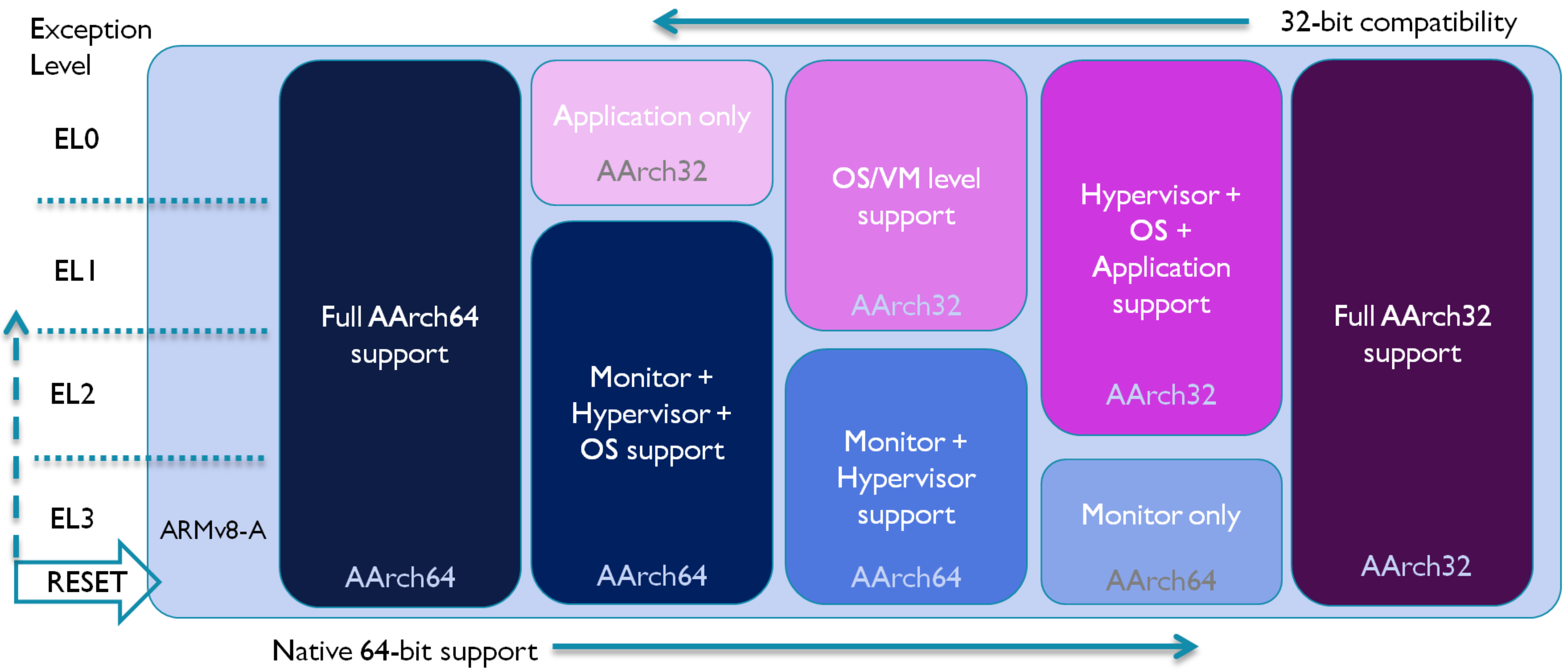
Increasing SoC complexity
Increasing OS complexity
Increasing choice of HW and SW

ARM

A Little Taxonomy

- ARMv{Version/Extension/Class} – Generic Architecture Name
  - ARMv8-A – ARM® architecture version 8, application class
- AArch64 – 64-bit execution state
  - A64 – ARM instruction set
  - LP64 – 64-bit data model
  - ILP32 – 32-bit data model
- AArch32 – 32-bit execution state
  - A32 – ARM instruction set
  - T32 – Thumb instruction set
  - ILP32 – 32-bit data model
- Interprocessing – Interaction of execution environments

Exception Levels & Interprocessing

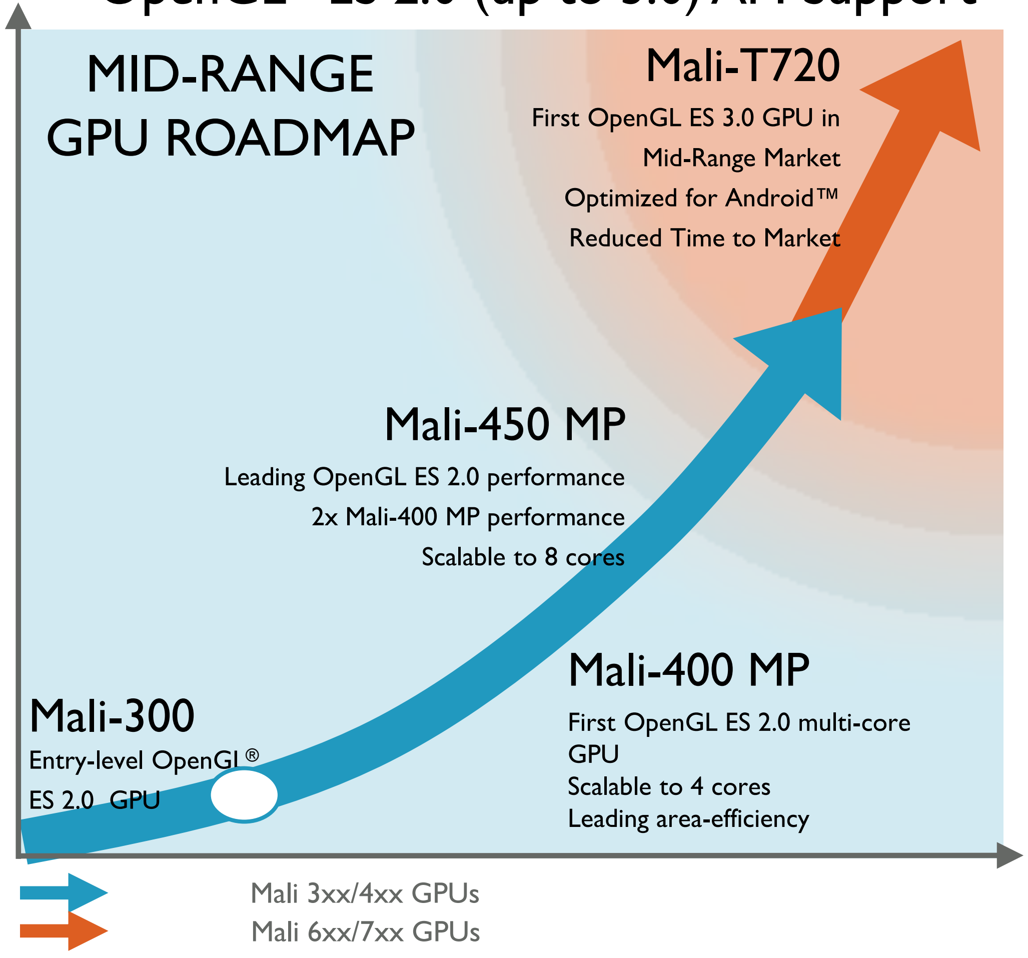


ARMv8-A Architecture Designed for Efficiency

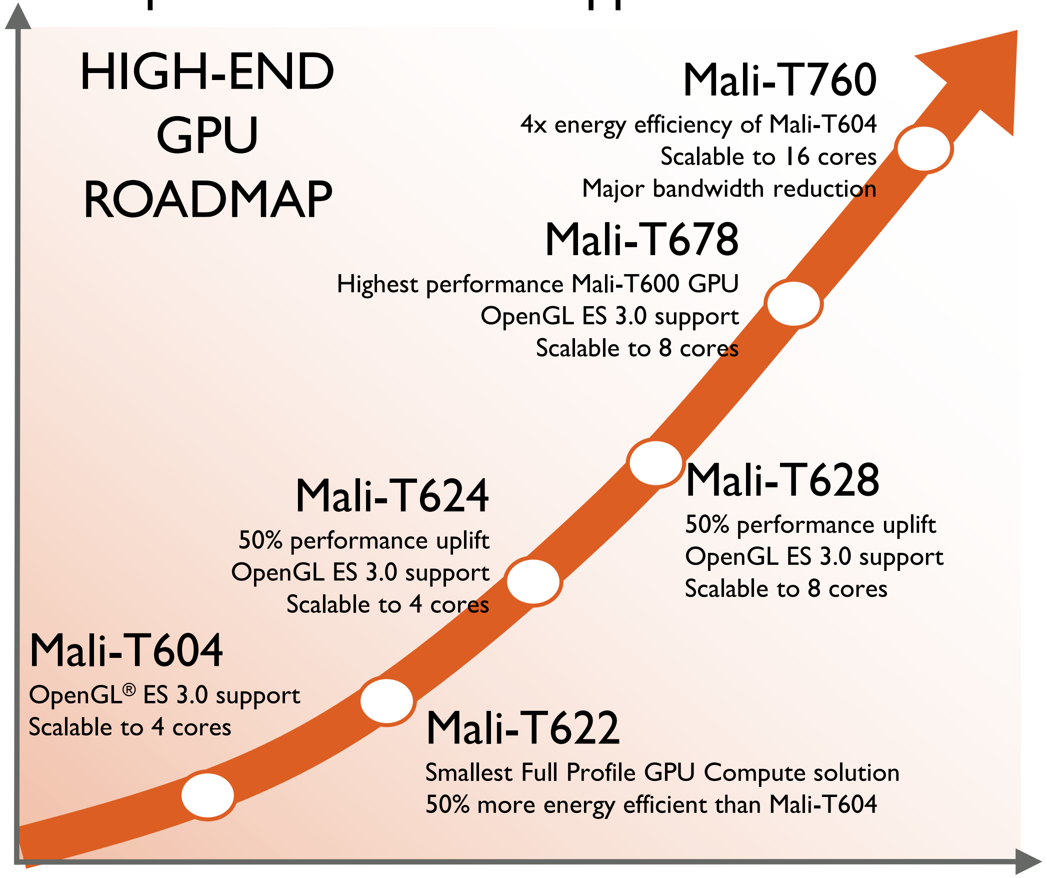
Enhancement	Why it Matters
64-bit architecture	Efficient access to large datasets
Increased number and size of general purpose registers	Gains in performance and code efficiency
Large Virtual Address Space	Applications not limited to 4GB memory Large memory mapped files handled efficiently
Efficient 32-bit/64-bit architecture	Common software architecture (phone, tablet, clamshell) A single software model across the entire portfolio
Double the number and size of NEON™ registers	Enhanced capacity of SIMD multimedia engine
Cryptography support	Over10x software encryption performance New security models for consumer and enterprise

ARM® Mali™ GPU Families

- Mali-400/450 MP (massive market share)
  - Mid-range Leading GPU (32 bit HW Arch)
  - OpenGL® ES 2.0 (up to 3.0) API Support



- Mali 6xx/7xx MP (high-end and mid-range)
  - 64-bit Architecture, ARMv8, IEEE-754-2008
  - OpenGL ES 3.x API Support



64-bit Support on Mali™ GPUs

- Mali-4xx 64-bit Support
  - DDK allocates memory from a proper range - SoC venders map Mali-4xx memory space (4 GB) into the system memory properly
  - User Mode/Kernel Mode compatibilities (32 and 64-bit UMDs both on a 64-bit KMD)
  - UMD driver generates the exactly same GPU commands on 32 and 64-bit CPUs
  - Pointer/Long/INT Type Castings in DDK
  - The New AArch64 NEON™ Optimization
  - Extended Life-cycles of Current Designs

- Mali-6xx/7xx 64 bit Support
  - Native 64-bit GPU architecture (>4 GB memory address space) – quite compatible with the CPU
  - Unified Memory Management Backend in DDK
    - Supports both the 32 bit and 64 bit systems
    - Now ALL GPU Virtual Address = CPU Virtual Address (up to 48 bit, only 4GB VA on 32-bit)
    - Enhanced Debugging Experience (32 & 64-bit)
  - GPU commands have been upgraded to utilize the full 64-bit capability of the system, with the access to much more data, than on 32-bit CPUs
  - Extended Capabilities of GPU Compute (64-bit)
  - Global Illumination

Summary

- Covered today:
  - Introduction of ARM® Mali™ Development Tools and Performance Profiling
  - Mali GPU Technologies for Game Developers
  - 64-bit CPU & GPU synergy
  - Q & A
- For more information:
  - malideveloper.arm.com
  - www.ds.arm.com
  - www.community.arm.com
  - malidevelopers@arm.com

ARM

Thank You

Any Questions?

malidevelopers@arm.com
Nathan.Li@arm.com