Mali Tools and 64 bit CPU/GPU Synergy

Nathan Li – DevRel, Ecosystem, Shanghai
ARM Multi-media Seminars
June 2014
1. Introduction to performance analysis with ARM® DS-5 Streamline Analyser

2. Software Profiling
   - Find hotspots, system glitches, critical conditions at a glance

3. GPU Profiling
   - Using the ARM® Mali™ GPU hardware counters to find the bottlenecks
   - CPU bound, Vertex bound, Fragment bound, and bandwidth bound cases

4. Debugging with Mali Graphics Debugger
   - Overdraw and Frame Analysis

5. 64 bit CPU & GPU synergy

6. 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
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

Select from 40+ CPU counters, OS level and custom metrics

Select one or more processes to visualize their instant load on CPU

Accumulate counters, measure time and find instant hotspots

Combined task switch trace and sampled profile for all threads
SMP Analysis
Take advantage of multicore SMP platforms

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

Per core, per process activity
Drilldown Software Profiling

Quickly identify instant hotspots

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

Filter timeline data to generate focused software profile reports
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.
The Power of Having It All in One Place
How effective are you at managing your energy budget?

Monitor instant voltage, current and power per channel

How long does the power manager take to respond to changes in CPU load?
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

Frame Outline
Assets View
Framebuffer / Render Targets
API Trace
Dynamic Help
Frame Statistics
Textures
Shaders
States
Uniforms
Buffers
Epic Citadel
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
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.
- 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:
  \[ \frac{197m}{169m} = 1.16 \text{ cycles/instruction} \]
CPU Bound

CPU

Memory

Vertex Shader

Fragment Shader
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.)
Vertex Bound

The diagram illustrates the flow of data from the CPU to the Memory, then to the Vertex Shader and Fragment Shader.
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
Fragment Bound
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
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

- 1x
- 8x
- 3-5x
- 5-7x
- 2x
- 3-5x
Shader Map and Fragment Count

Identify the top heavyweight fragment shaders

Draw 185, total vertices: 120380

Fragment Count Per Program

~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:
  \[(96m \text{ (Mali L2 Cache} \rightarrow \text{External read beats)} + 90.7m \text{ (Mali L2 Cache} \rightarrow \text{External write beats)}) \times 16 \approx 2.9 \text{ GB/s}\]
- 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**.

Indices sparseness: 1.47 bad for caching!
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
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

Camera moving in the scene

Loading screen
ARM CPU Architecture Evolution

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

ARMv4
- ARM7TDMI

ARMv5
- ARM926EJ

ARMv6
- ARM1176
- Thumb®-2
- TrustZone®
- SIMD

ARMv7-A/R
- Cortex-A9
- A32+T32 ISAs
  - Scalar FP (SP and DP)
  - Adv SIMD (SP Float)

ARMv8-A
- Cortex-A50 series
  - A64 ISA
  - Scalar FP (SP and DP)
  - Adv SIMD (SP+DP Float)

Virtualization

NEON™ Adv SIMD

1995

2005

2015
**ARMv8-A Architecture Designed for Efficiency**

<table>
<thead>
<tr>
<th>Enhancement</th>
<th>Why it Matters</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit architecture</td>
<td>Efficient access to large datasets</td>
</tr>
<tr>
<td>Increased number and size of general purpose registers</td>
<td>Gains in performance and code efficiency</td>
</tr>
<tr>
<td>Large Virtual Address Space</td>
<td>Applications not limited to 4GB memory</td>
</tr>
<tr>
<td></td>
<td>Large memory mapped files handled efficiently</td>
</tr>
<tr>
<td>Efficient 32-bit/64-bit architecture</td>
<td>Common software architecture (phone, tablet, clamshell)</td>
</tr>
<tr>
<td></td>
<td>A single software model across the entire portfolio</td>
</tr>
<tr>
<td>Double the number and size of NEON™ registers</td>
<td>Enhanced capacity of SIMD multimedia engine</td>
</tr>
<tr>
<td>Cryptography support</td>
<td>Over 10x software encryption performance</td>
</tr>
<tr>
<td></td>
<td>New security models for consumer and enterprise</td>
</tr>
</tbody>
</table>
ARM Mali GPU Families

- **Mali 400/450 MP (massive market share)**
  - Mid-range Leading GPU (32 bit HW Arch)
  - OpenGL ES 2.0 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)
Summary

- **Covered today:**
  - Introduction to performance analysis
  - Software Profiling
  - GPU Profiling
  - Debugging with the ARM® Mali™ Graphics Debugger
  - 64 bit CPU and GPU Synergy

- **For more information:**
  - [www.malideveloper.arm.com](http://www.malideveloper.arm.com)
  - [www.ds.arm.com](http://www.ds.arm.com)
  - [www.community.arm.com](http://www.community.arm.com)
Thank You

Any Questions?

The trademarks featured in this presentation are registered and/or unregistered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. Any other marks featured may be trademarks of their respective owners.