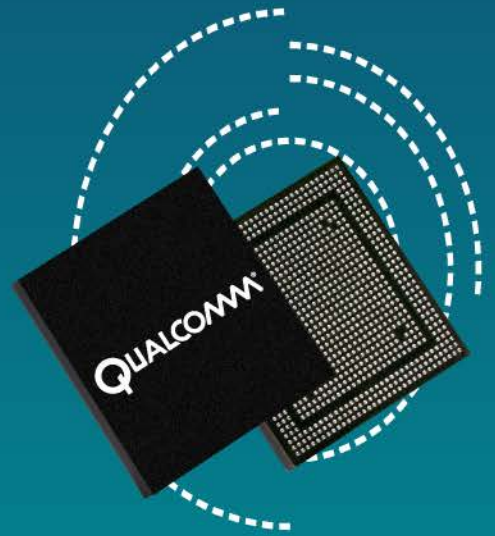
Qualcomm
TECHNOLOGIES, INC



Adreno™ Debugging Overview

80-NR299-1 A

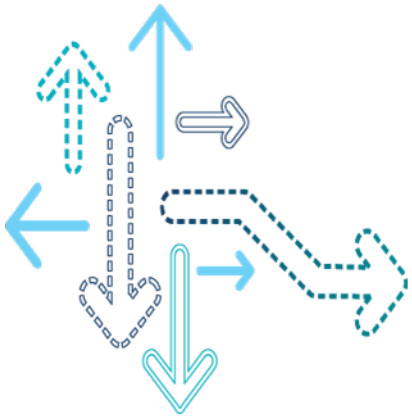
Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision History

Revision	Date	Description
A	Jul 2014	Initial release

Contents

Introduction



Scope

- This document focuses on Adreno™ driver debugging.
  - OpenGL® ES is part of this topic.
  - OpenCL is not part of this topic.
  - Some Google tools are discussed.
  - The Google framework is out of scope.

What is a GPU Use Case?

- The GPU is one of the most actively used parts in mobile devices.
  - Game rendering
  - Rendering by HWUI
  - Layer composition by SurfaceFlinger
  - Computing (OpenGL/Renderscript)
- What uses the GPU?
  - Apps that call OpenGL ES API directly
  - Apps that are based on HWUI or Renderscript
  - Google framework, which calls the OpenGL ES API directly
  - App that calls OpenCL

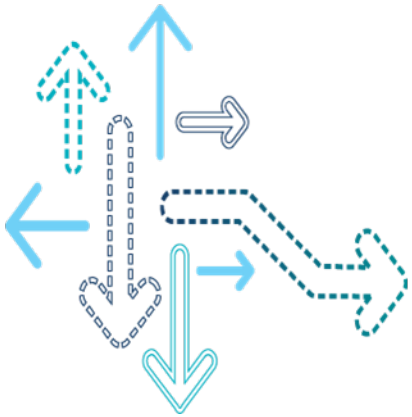
What Causes a GPU-Related Issue?

- Rendering corruption
  - OEM is primarily interested in this issue
  - Issue is not necessarily a driver issue
- GSL out of memory
  - If the client does not release the GPU resource, this issue can occur
  - There may also be an internal issue with the driver
- Driver stability, power, and performance issues are possibly driver issues

Primary Debugging Step

- Check if it is a known issue
  - Whenever you find a new issue, always refer to the latest Qualcomm Technologies, Inc. (QTI) build release notes to ensure that the issue is not already a known issue.
- Test on the latest build
  - Whenever you find a new issue related to graphics, one of the key methods to getting the quickest resolution is to test it on the latest build.
  - Frequently, the graphics module is independent of other system-level components; migrate only the graphics component, i.e., the KGSL, Adreno User Mode Driver (UMD), or HAL to the latest build to see if the issue is resolved there.
  - This is particularly important for issues before and after the CS release. A large percentage of issues are resolved by using newer versions of the build.

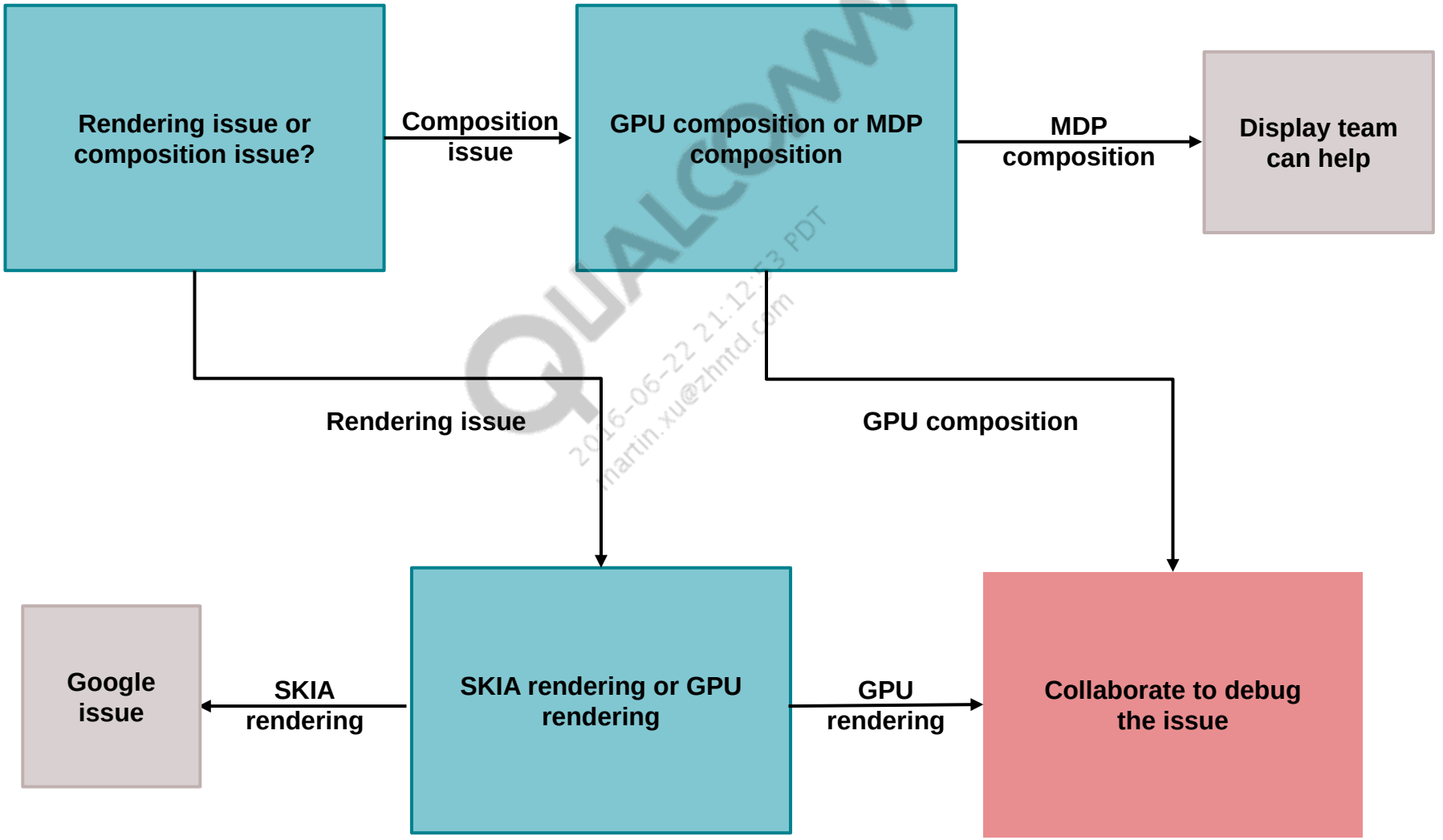
Rendering Corruption



How to Approach Rendering Corruption Issues

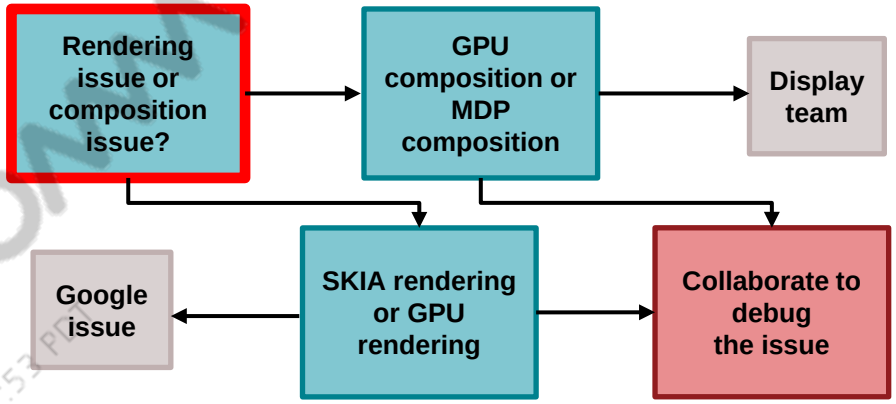
- Reproducible corruption issues – Solid approaches are:
  - Adreno debugging configuration (adreno_config.txt)
  - Enable GL API logs to understand GL APIs used for the use case
  - GI Tracer (Google tool) to capture the specific screen and analyze GL calls
  - Add extra logs in the Adreno library to analyze details of some suspicious variables
  - Adreno Profiler to observe shader, texture, and GL calls
- Nonreproducible UI corruption issues
  - Close collaboration between the OEM and QTI; it is not reasonable to add a random debug code into the Adreno graphics driver without good solutions to fix the issue
  - Suggested approaches to move forward
    - OEM
      - Find reproducible steps
      - Provide detailed information, i.e., test condition, app information, behavior, and OEM customization for framework
    - QTI
      - Provide debugging code/SBA based on detailed information provided by the OEM
      - Identify internal known issues that may help OEM issues

Basic Approaches



Dump Layers

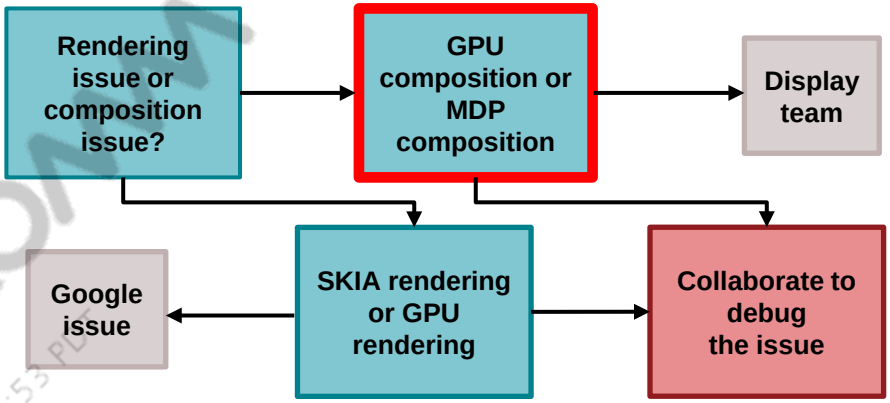
- Easy way to confirm if it is a rendering issue
- It is a composition issue if it has no layer corrupted, but the composited scene is corrupted
- /system/build.prop – Permission 644
  - debug.sf.dump.enable=true – Set this at boot time
- Do “adb shell setprop” on runtime to dump each layer
  - debug.sf.dump.primary true – dump primary.(default)
  - debug.sf.dump.external true – dump external
  - adb shell setprop debug.sf.dump <no. of frames>
    - **/data/sfdump.raw**<*YYYY*><*MM*><*DD*>**.**<*HH*><*MM*><*SS*>/**sfdump**<*dump frame no.*>_**layer**<*layer no.*>_<*buffer width*>**x**<*buffer height*>_<*format*>**.raw**
  - adb shell setprop debug.sf.dump.png <no. of frames>
    - format: **/data/sfdump.png**<*YYYY*><*MM*><*DD*>**.**<*HH*><*MM*><*SS*>**/sfdump**<*dump frame no.*>_**layer**<*layer no.*>**.png**
    - Slower than raw layer dump

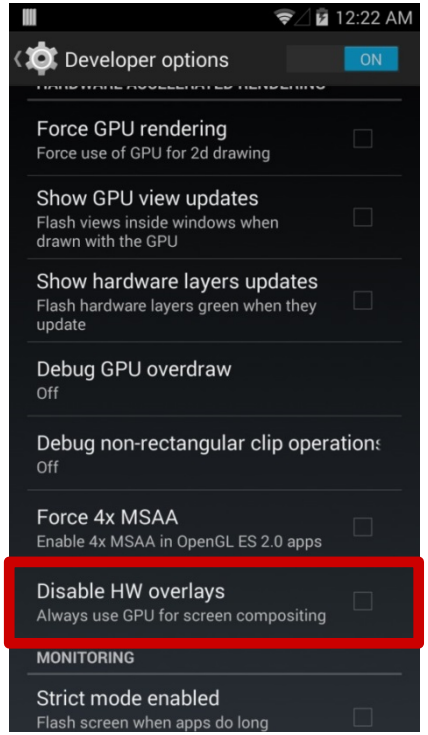


Dump Layers (cont.)

Enable GPU Composition

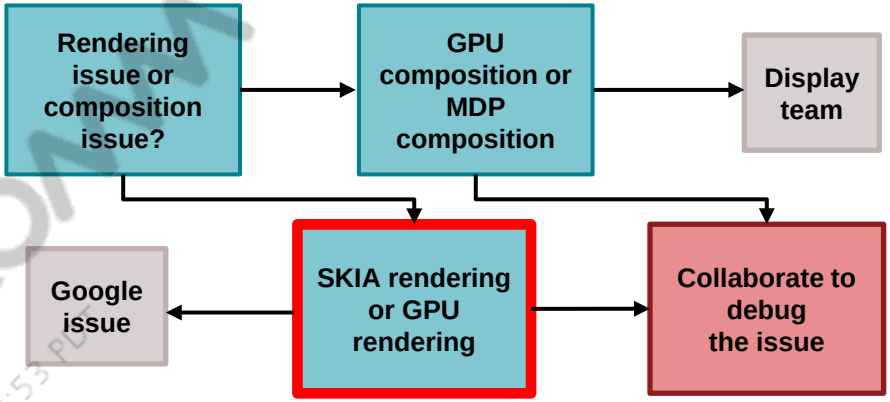
- If it is a composition issue, check if it is GPU composition.
- To use only GPU composition:
  1. Disable MDP composition.
  2. Go to **Settings→About phone**.
  3. Touch the build number five times repeatedly. The developer options appear in Settings.
  4. Go to **Settings→Developer options→Disable HW overlays**.

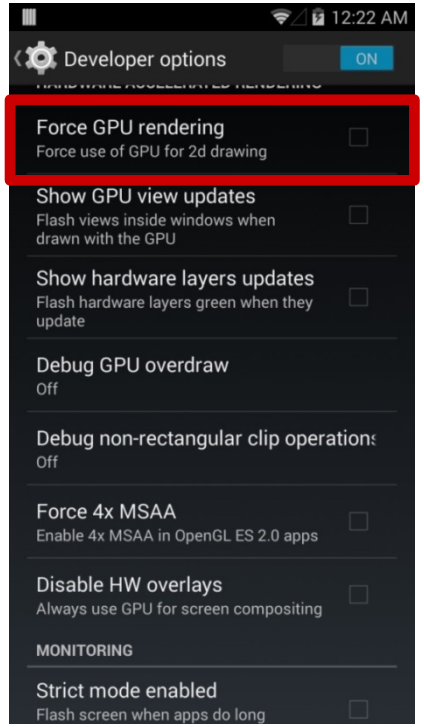




Enable HWUI Rendering

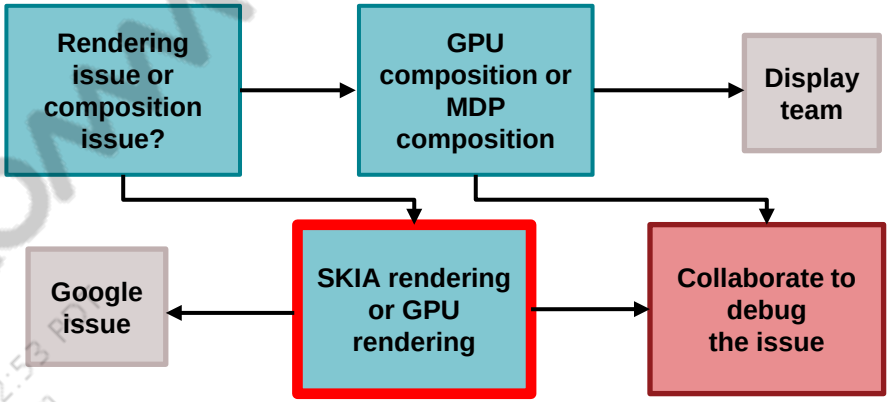
- If rendering is based on SKIA, try HWUI rendering.
- Enable HWUI for every SKIA rendering an app by selecting **Force GPU rendering** in Developer options.





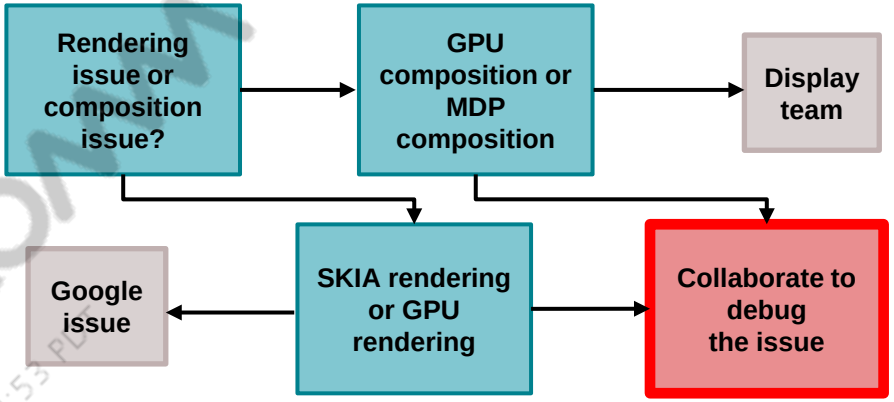
Disable HWUI Rendering

- Eliminate Google hardware acceleration using the following method.
- To disable hardware acceleration for certain processes:
  - Confirm if an app is based on HWUI.
    - adb shell dumpsys gfxinfo <pid>
    - If there is a display list, the app is using the hardware acceleration path.
  - If the app source code is accessible, open the AndroidManifest.xml file and locate the following attribute under <application/> tag:
    - Android:hardwareAccelerated="true"
  - If the above attribute is True, the hardware acceleration is set to enable. Set the attribute to False and rebuild the app to determine if it fixes the issue.
- To disable hardware acceleration:
  - vendor/qcom/proprietary/common/{Chipset}/BoardConfigVendor.mk
  - USE_OPENGL_RENDERER=false



Getting Started on GPU Rendering Issues

- If Mixed mode composition and SKIA rendering issues are eliminated, it is a GPU rendering issue
  - It is not always a GPU driver issue; 38.8% out of rendering corruption was a driver issue in MSM8974/MSM8974Pro
- Possible causes of rendering corruption issues
  - In the GLES client
  - Adreno driver issue
- Confirm if it is a GLES client (app) issue
  - If it is not an app issue, consider the Adreno driver



Check Common Errors

- Check if there is any page fault in the kernel log.
  - A page fault error indicates the GPU failed to read and/or write data from/to the CPU; this occasionally causes UI artifacts.
  - If there is a page fault error, it needs to be solved.
- Check if the Adreno driver issues any GL errors.
  - The Adreno driver prints out whenever a GL_ERROR is set.
  - The GL driver sets GL_ERROR code when it detects some error case in the operation; this error code usually indicates something is incorrect in the current rendering and might be related to the UI corruptions.
  - GL errors can be detected in the logcat log.

Use Adreno Profiler to Capture Rendering Corruption

- Use the Adreno Profiler to capture the corrupted rendering. It captures all the GL commands, texture, shader program, VBO, and FBO of this particular frame and uses the PC GPU card to render the result.
- Adreno Profiler is very helpful in:
  - Capturing the API call of this frame; this helps determine if there is any app bug in this flow
  - Determining which draw call causes these UI artifacts
  - Locating the area that causes this issue by toggling EGL and GL state settings
  - The Adreno Profiler has two tools, scrubber and grapher; of these, grapher is suited to analyze performance problems; scrubber is used to capture the entire GL calls, textures, and shaders that are used to render a scene and suited to analyze UI corruptions.
  - The Adreno profiler is located at https://developer.qualcomm.com/download.

Best Practice of Adreno Profiler Debugging

- A game issue
- A fragment shader is not working properly

```
#define ECCENTRICITY      0.3
#define BASE_SHINNESS     32.0
#define SPEUCLAR_ROLL_OFF  0.5
#define RIM_LIGHTINDENSITY 0.1
#define RIM_FACTOR        0.0
uniform sampler2D                colorTex;
uniform sampler2D                normalTex;
uniform sampler2D                specularTex;
uniform lowp vec4                ambientColor ;
varying mediump vec2      TexCoord0;
varying mediump vec3      vEyeDirTan;
varying mediump vec3 LightDirTan;

mediump float HalfLambertLighting( mediump vec3 Normal, mediump vec3 LightVec )
{
        // Scale from [-1, 1] to [0, 1]
        return 0.5 * ( dot( Normal, LightVec ) + 1.0 );
}

// Rescale value from [0, 1] to [interval_min, interval_max]
mediump float IntervalRescale( mediump float val, mediump float interval_min, mediump float interval_max )
{
        return interval_min + clamp( val, 0.0, 1.0 ) * ( interval_max - interval_min );
}

void main()
{
  lowp vec4      diffuse    = texture2D ( colorTex, TexCoord0 ) ;
  mediump vec3 normalTan  = normalize ( texture2D ( normalTex, TexCoord0 ).xyz * 2.0 - 1.0 ) ;
  lowp vec4 color       = vec4 ( 1.0 , 1.0 , 1.0 , 1.0 ) ;
  mediump float diffDot   = HalfLambertLighting( normalTan, LightDirTan ) ;
  diffDot = IntervalRescale( diffDot, 0.5, 1.0 );
  color   = clamp ( vec4 ( diffDot , diffDot , diffDot , 1.0 ) , 0.0 , 1.0 ) ;
  lowp vec4 specularColor = texture2D ( specularTex , TexCoord0 ) ;
  mediump vec3 reflection = normalize( 2.0 * color.xyz * normalTan - LightDirTan );
  lowp vec4 spec        = specularColor * SPEUCLAR_ROLL_OFF * pow ( clamp ( dot ( reflection , vEyeDirTan ) , 0.0 , 1.0 ) , BASE_SHINNESS * ECCENTRICITY ) ;
  mediump vec4 rimColor   = vec4 ( 0.8 , 0.8 , 0.8 , 1.0 ) ;
  rimColor          *= RIM_LIGHTINDENSITY * pow ( 1.0 - dot ( normalTan , vEyeDirTan ) , 1.6 ) ;
  rimColor          = mix ( rimColor , vec4 ( 1.0 , 1.0 , 1.0 , 1.0 ) , RIM_FACTOR ) ;

  gl_FragColor = diffuse * color + spec + rimColor;
  gl_FragColor.a = gl_FragCoord.z;
}
```

Best Practice of Adreno Profiler Debugging (cont.)

- If the value of “dot (normalTan , vEyeDirTan)” is larger than 2 or smaller than 0, gl_FragColor is over 1.0.
- Therefore, the final color is shown in black.
- Modifed shader
  - gl_FragColor = **min**(diffuse * color + spec + rimColor, 1.0); (Left image)
  - rimColor*= RIM_LIGHTINDENSITY*pow(1.0-dot(normalTan, vEyeDirTan),**1.0**); (Right Image)
- The app needs to fix the shader code.

Texture Issues

- Many of the rendering issues originate from the texture side
- If the input texture is invalid, it needs to dump textures
  - Invalid input texture – Look at the GLES client first
  - Valid input texture – Make sure if the GLES client bind right texture
- To dump input texture, put the following configuration in the adreno_config.txt file:
  - enableTextureDumping=1
  - textureDumpingSkipDraws={value}
  - textureDumpingNumDraws={value}
  - Texture dumps to be stored in /data/local/tmp
  - Limitation – Cannot dump textures of the EGL image; QTI CE team can provide the debugging library to dump the EGL image

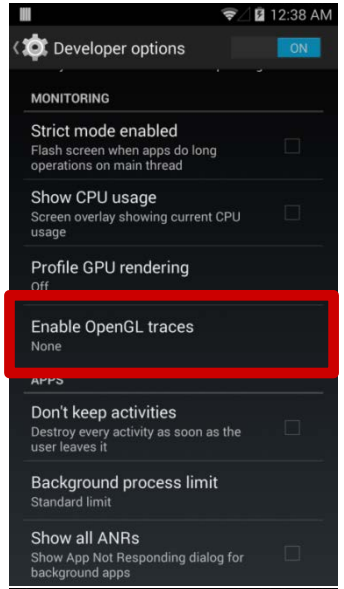
Enable GLES API Logs

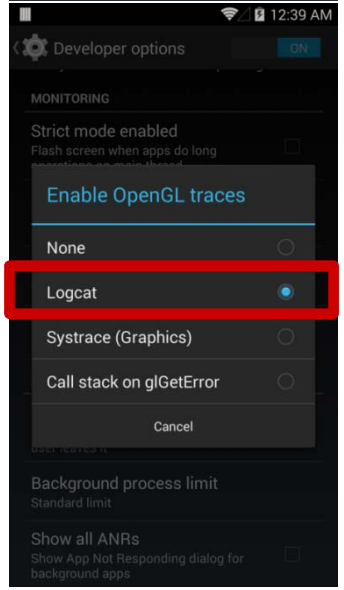
- Use GLES API logs to understand the app behavior
- Two ways to enable GLES API logs
  - GL Trace by Android™
  - GL API logging by QTI
- GL Trace log by Android
  - Pros – Easy to enable; GL API to is shown in logcat
  - Cons – GL API logs only; no resource data; hard to check per context
- GL API logging by Adreno driver
  - Adreno debugging feature
  - Pros – GL API logs are saved as a separate file, per context; corresponding resource to be saved
  - Cons – Hard to match corresponding logcat log

Enable GLES API Logs (cont.)

- To enable a GL Trace log:
  - Enable **OpenGL traces** in the Developer options.
  - Select **Logcat**.
  - GLES API log to be printed in logcat

```
D/libEGL  ( 3986): glUniform4f(5, 1, value);
D/libEGL  ( 3986): glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 4);
D/libEGL  ( 3986): glDrawElements(GL_TRIANGLES, 1080, GL_UNSIGNED_SHORT, (const GLvoid *) 0x00000000);
D/libEGL  ( 3986): glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
D/libEGL  ( 3986): glBindBuffer(GL_ARRAY_BUFFER, 3);
D/libEGL  ( 3986):     -0.00781383, -0.999919, 0.0100778, 0,
D/libEGL  ( 3986):     -0.0188642, -1.35788, -0.0105946, -0.0103362,
D/libEGL  ( 3986):     0.967147, -0.00499606, 0.254169,
D/libEGL  ( 3986):     0.377193, 0.32166, -3.74772, 1
D/libEGL  ( 3986): glUniformMatrix4fv(1, 1, GL_FALSE, value);
D/libEGL  ( 3986): const GLfloat value[] = {
D/libEGL  ( 3986): const GLfloat value[] = {
D/libEGL  ( 3986): glDisableVertexAttribArray(1);
D/libEGL  ( 3986): glBindBuffer(GL_ARRAY_BUFFER, 0);
D/libEGL  ( 3986):     0.101887, 0.94488, 0.311159, 0,
D/libEGL  ( 3986):     0.98, 0.75, 0.03
D/libEGL  ( 3986): glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 4);
D/libEGL  ( 3986): glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 24, (const GLvoid*) 0x00000000);
D/libEGL  ( 3986): const GLfloat value[] = {
D/libEGL  ( 3986): const GLfloat value[] = {
D/libEGL  ( 3986):     -0.936234, -0.217562, 0.27592,
D/libEGL  ( 3986): glUniform3f(6, 1, value);
D/libEGL  ( 3986): };
```





Enable GLES API Logs (cont.)

- To enable GL API logging in the Adreno debugging feature:
  - log.apicalls=1
  - Make sure /data/local/tmp folder has write permissions set
  - GL API is saved in /data/local/tmp device path with the resource files
  - Files are stored per context
  - QTI can replay the app with API logs and resource files
    - However, if a resource is allocated by non-Adreno, e.g., EGL image, it might not make a complete replay app due to a missing resource
    - Once creation of replay app has succeeded, QTI can debug the issue on a QTI reference board (MTP)

Enable GLES API Logs (cont.)

- A case study with GLES API logging
- In a VT call, remote view shows preview contexts for a short time
- Enabled GL logging and looked at the API logging to understand the app behavior
- Found remote view bound preview texture when the issue occurs

GL Tracer

- GL Tracer is a tool from Google that analyzes UI problems, corruptions, etc.
  - http://developer.android.com/tools/help/gltracer.html
- To take a glTrace:
  1. Connect the device to a PC and start capturing logcat.
  2. Start the app.
  3. Search the logcat log to identify the app, e.g.:
     - Find the glTrace for the main screen of the Gallery app.
     - Launch the email app and open logcat; the activity name is:
       - Line 13715: I/ActivityManager(656): START {act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=com.google.android.gallery3d/com.android.gallery3d.app.Gallery u=0}
  4. Close the Gallery app.
  5. Invoke gltrace app by starting “monitor” from the shell.
  6. Select **Tracer for OpenGLES Perspective**.
  7. Fill the Activity Package name – com.google.android.gallery3d.
  8. Fill the Activity to launch – com.google.android.gallery3d/com.android.gallery3d.app.Gallery.

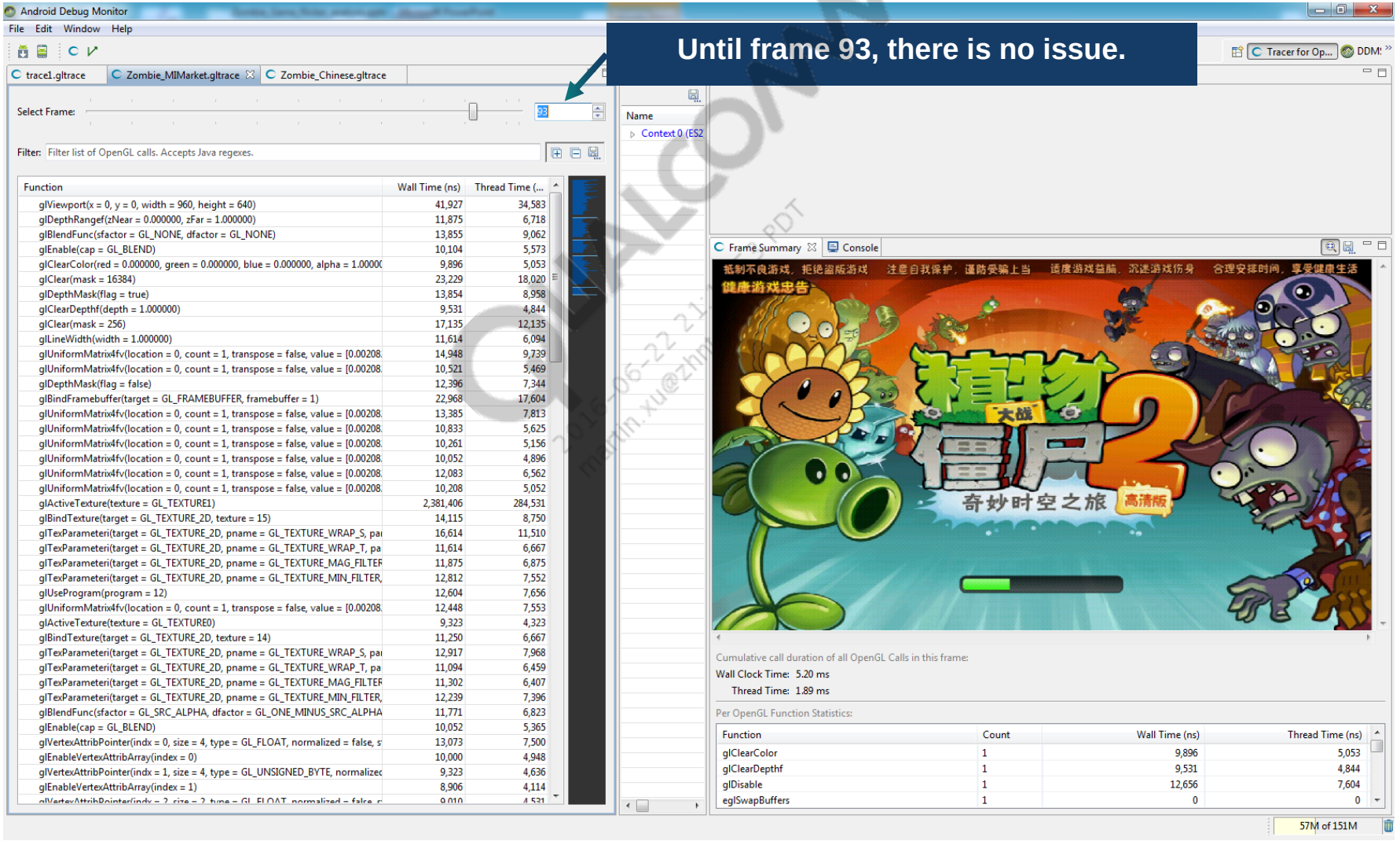
GL Tracer (cont.)

- To take a glTrace (cont.):
  9. Select **Activity name is fully qualified. Do not prefix with package name**.
  10. Select **Read back framebuffer 0 on eglSwapBuffers** and ignore the other data collection options.
  11. Enter the filename onto which the trace has to be captured, e.g., androidGallery.gltrace.
  12. Click **Start** and the glTrace invokes the activity and starts capturing the glCalls.
  13. Reproduce the use case and stop the capturing glCalls.
  14. Share the saved .gltrace file for further analysis with QTI.

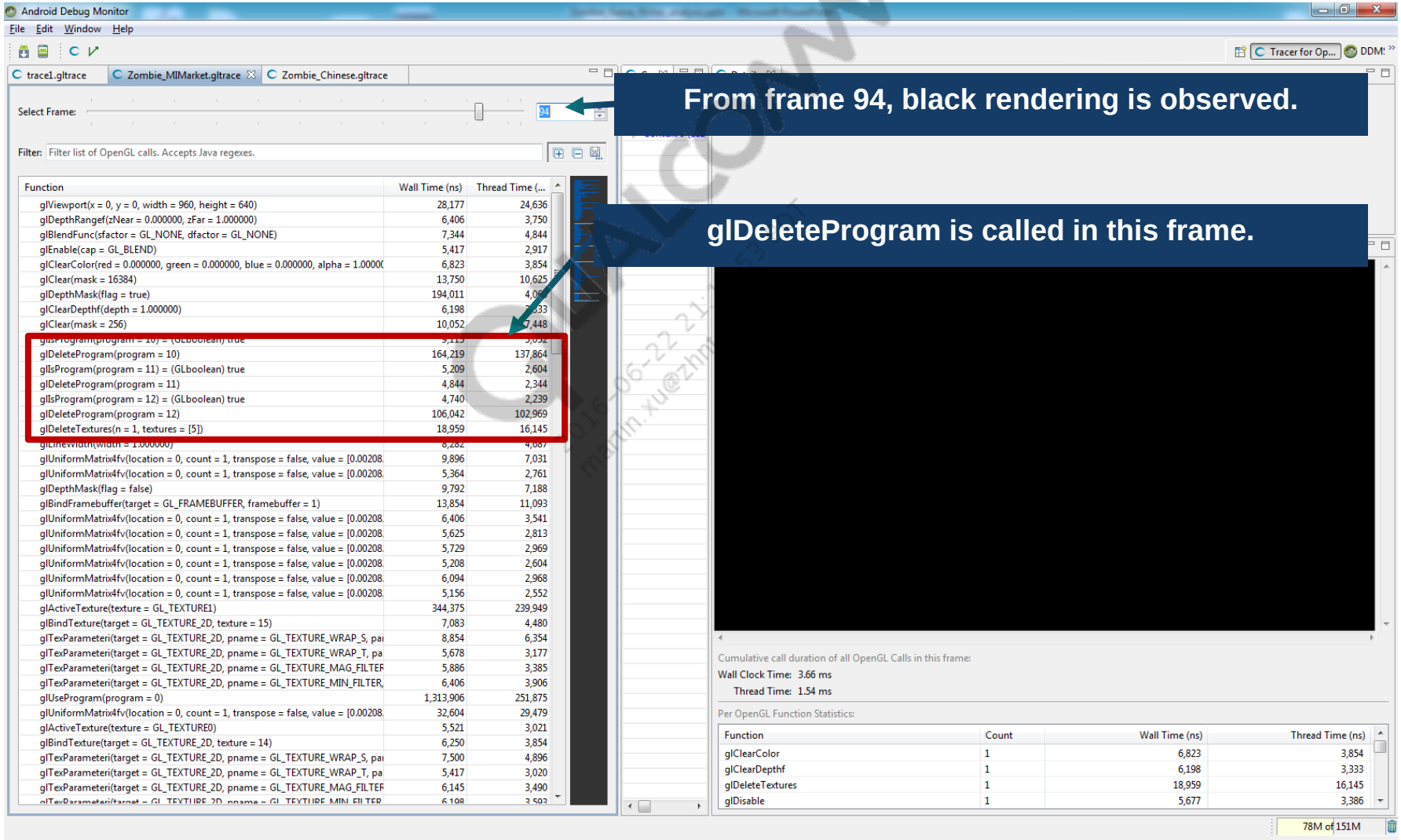
GL Tracer – Debugging Example

- Issue – A flickering is observed in a game
- Debug step
  - Dump SurfaceFlinger layers – Some game layers show a black scene
  - Connect GL Tracer to get a glTrace
  - Observation
    - When the issue occurs, the first frame calls glDeleteProgram
    - After that time, the game was calling only eglSwapBuffer without any GL calls (no rendering)
    - Since Android is using triple buffers, the black scene appears per every three frames→Appears as flickering

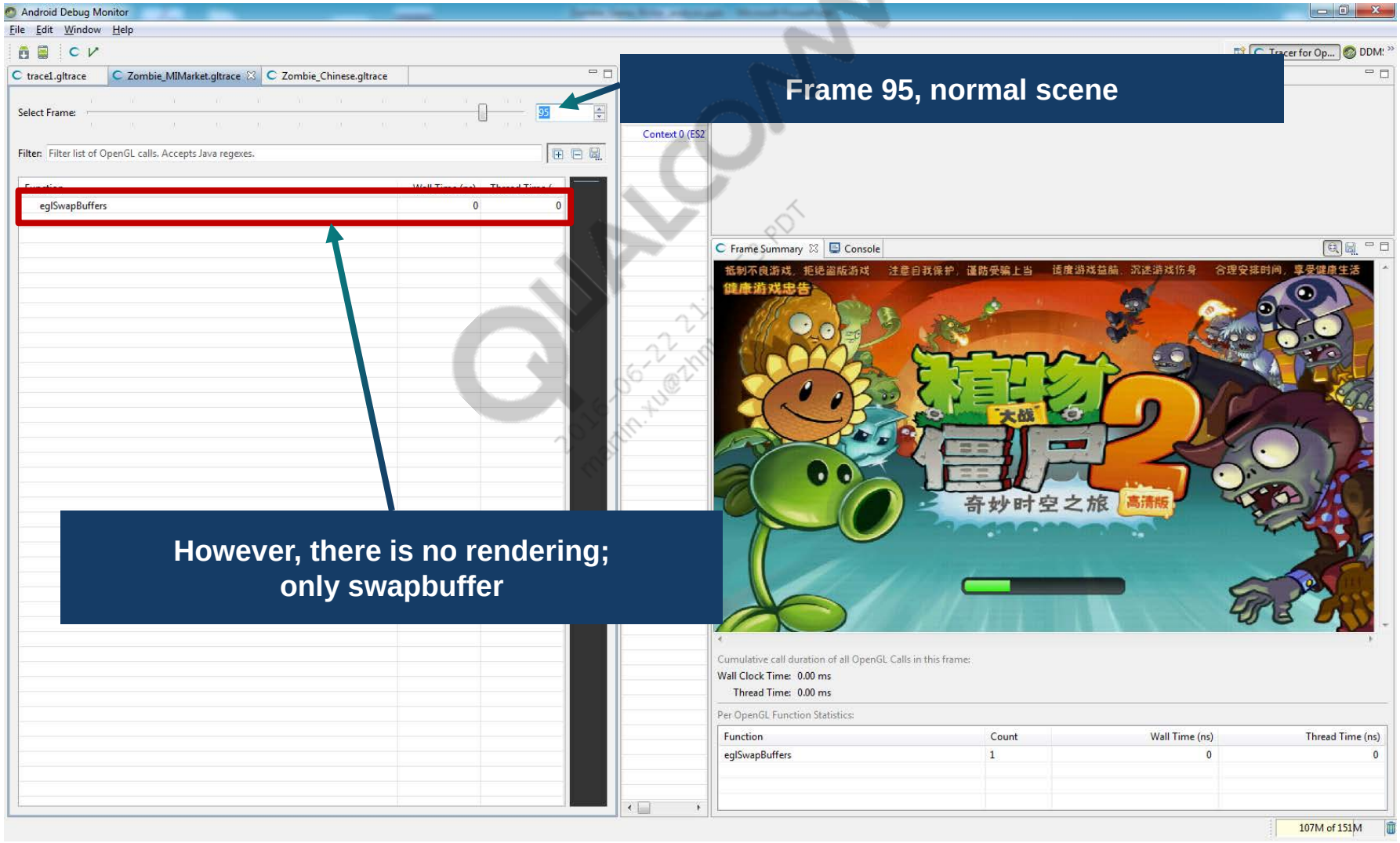
GL Tracer – Debugging Example (cont.)



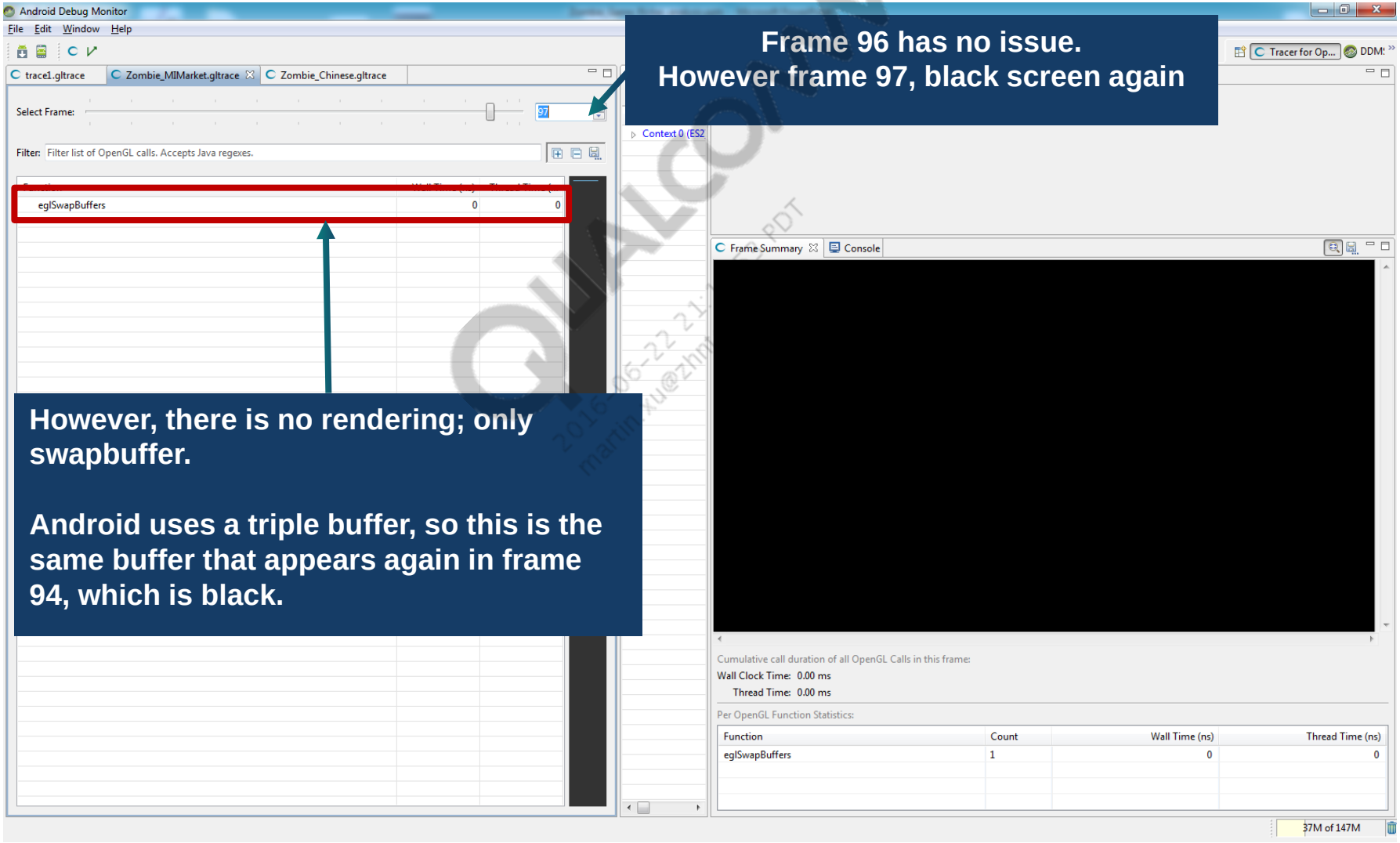
GL Tracer – Debugging Example (cont.)



GL Tracer – Debugging Example (cont.)



GL Tracer – Debugging Example (cont.)



Adreno Driver Issues

- If the rendering corruption does not originate from the applicaton, look at the Adreno driver.
- QTI has to investigate the issue.
- Provide as much information as possible.
- There are many Adreno debugging features, which can turned on/off by adreno_config.txt.
  - If any of the features help the issue, it is a good starting point.

Prerotation

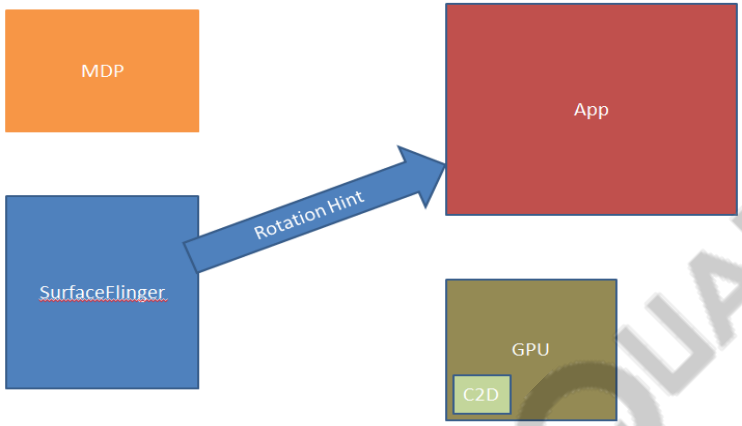
- If a rendering corruption occurs with rotation, it may be a prerotation issue.
- Prerotation skips the rotation stage from SurfaceFlinger, which allows a rotated layer bypass to the MDP.
  - Power/performance benefits
    - Helps SurfaceFlinger to be in MDP Composition mode
    - Significant power savings when the device is rotated
- To disable prerotation:
  - enableRotationShaderPatching=0 in adreno_config.txt
- If issue disappears with prerotation disabled, it is a driver issue.

Prerotation (cont.)

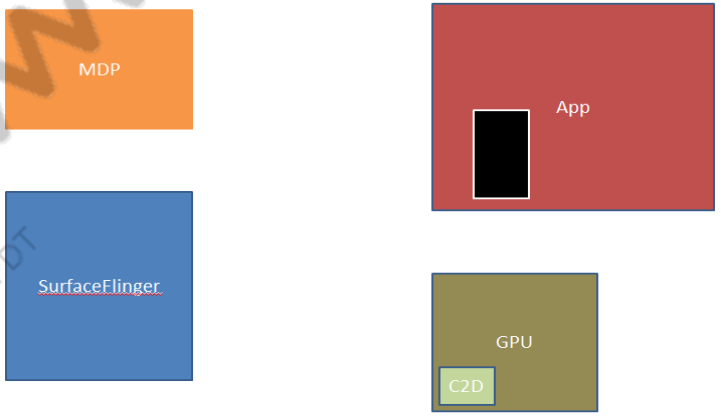
- Without prerotation, draw landscape buffer on the portrait device.

Prerotation (cont.)

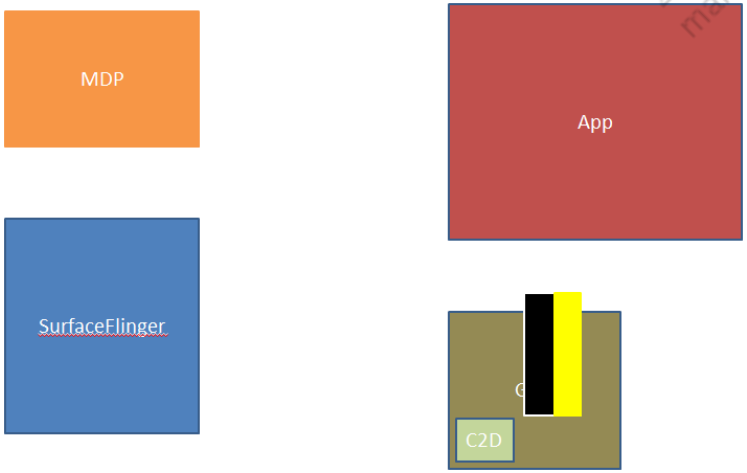
- With prerotation, draw landscape buffer on the portrait device.



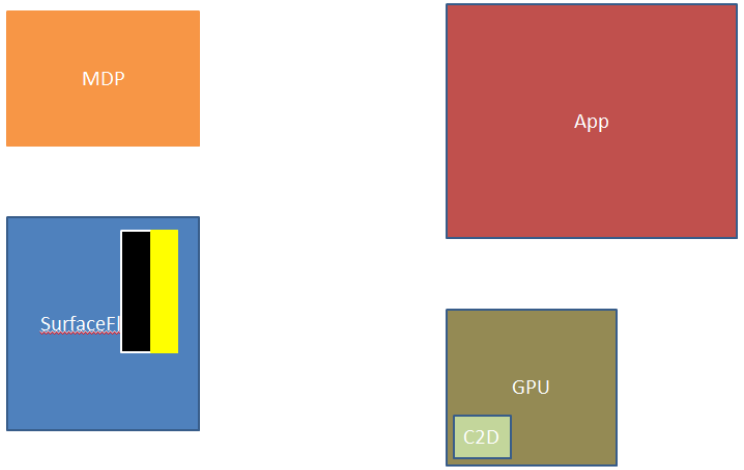
App can get a hint from SurfaceFlinger to rotate before the surface is allocated



Allocate a surface that looks portrait even though the user is holding it landscape



GLES renders the yellow and sends it over to SurfaceFlinger



Since SurfaceFlinger does not need to rotate, it can potentially put that layer in bypass and pass it over to MDP

Prerotation (cont.)

- Confirm if prerotation is working
  - Layer transform value is 0 even if the screen is rotated
  - If prerotation is not working, every layer should go to GPU composition even if there are small number of layers
- Captured “dumpsys SurfaceFlinger” running a portrait game
  - The game layer goes to HWC with tr 0, which means SurfaceFlinger does not rotate the layer with prerotation enabled.

type	handle	hints	flags	tr	blend	format	source crop	frame	name
HWC	b73bff78	00000002	00000000	00	00100	00000002	[0.0, 0.0, 720.0, 1280.0]	[0, 0, 720, 1280]	SurfaceView
FB TARGET	b734b470	00000000	00000000	00	00105	00000001	[0.0, 0.0, 720.0, 1280.0]	[0, 0, 720, 1280]	HWC_FRAMEBUFFER_TARGET

  - The game layer goes to GPU composition, since SurfaceFlinger needs to rotate the layer with prerotation disabled (tr==4).

type	handle	hints	flags	tr	blend	format	source crop	frame	name
GLES	b87b9dc8	00000000	0000000	04	00100	00000002	[0.0, 0.0, 1280.0, 720.0]	[0, 0, 720, 1280]	SurfaceView
FB TARGET	b876bab0	00000000	0000000	00	00105	00000001	[0.0, 0.0, 720.0, 1280.0]	[0, 0, 720, 1280]	HWC_FRAMEBUFFER_TARGET

- **Note:** Prerotation is only applicable for GPU rendering; if an app does not use GPU rendering, prerotation is not working.

Tiled Rendering

- Tiled rendering
  - Use QCOM_tiled_rendering.
  - This extension allows the app to specify a rectangular tile rendering area and have full control over the resolves for that area.
  - The information given to the driver through this API can be used to perform various optimizations in the driver and hardware.
    - An example of optimization is being able to reduce the size or number of the resolves.

Tiled Rendering (cont.)

- To disable tiled rendering:
  - disableTiledRendering=1 in adreno_config.txt
  - glStartTilingQCOM() and glEndTilingQCOM() will be silently ignored when set
- If disabling tiled rendering helps, verify if the issue is related to a single bin or multiple bins.
  - forceGmemSize=1
  - gmemSize=64 // this limits GMEM size Adreno driver uses to 64 KB.
- If the issue is related to tile rendering, resolve logs are useful to debug the issue. Use the method on the next slide to enable the resolve log.

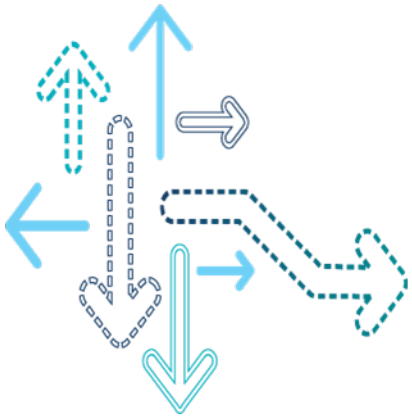
Tiled Rendering (cont.)

- To enable the resolve log:
  - log.resolves=1 in adreno_config.txt
  - Resolve logs are stored in /data/local/tmp; verify that the path has write permissions set
  - Resolve refers to the rendered content from the GPU memory (GMEM) being saved into the actual framebuffer that is part of the slow memory accessible by the CPU
  - Unresolve is the opposite, copying system memory into the GMEM

Resolve Log

- The resolve log captures the overall resolve activity, including:
  - Unresolve
  - Dirty region tracking (with tiled rendering)
  - Binning
  - The resolve operation itself
- To enable resolve logs, add the following to the Adreno configuration file:
  - Enabling resolve logs
    - write log.resolves=1 to adreno_config.txt
      - adb shell “echo ‘log.resolves=1’ >> /data/local/tmp/adreno_config.txt”
  - Ensure your output directory is writeable
  - Run the process to be monitored
  - Resolve process is generated for each process and context
    - resolve_log_<pid>_<context>.txt
    - For example, resolve_log_2178_5b77d008.txt

GSL Out of Memory



GSL Memory

- The KGSL driver uses vmalloc-mapped memory for all of its internal memory needs when the GPU IOMMU is turned on, including command buffers and texture/FBO memory.
  - Without mapped to GPU IOMMU, the GPU cannot access those memory chunks.
  - Types of memory mapped to GPU IOMMU:
    - Texture, FBO, VBO, command buffer, EGL surface, EGL image, and every resources accessed by GPU
- Every resource is allocated and freed by the Adreno driver except EGL resources.
  - EGL resources are allocated and freed outside of the Adreno driver.
  - Adreno driver maps and unmaps EGL memory for the GPU to access them.

How to Confirm GSL Memory Leakage

- The following command shows mapped memory size in a process:
  - cat /sys/class/kgsl/kgsl/pagetables/{pid}/mapped
    - This command shows a single number of mapped size in a byte.
    - GPU IOMMU is per-process based by default.
  - If this size of memory keeps increasing, GSL memory leakage can be suspected.
    - The problematic process keeps mapping memory chunks to GPU IOMMU.

Debug GSL Memory Leakage

- Need to understand which memory chunk type causes memory leakage
- Easier way to check this:
  - cat /sys/kernel/debug/kgsl/proc/{PID}/mem
  - This command shows GSL memory chuck list with memory types.
  - Quickly understand which type of memory causes leakage.
  - However, it may not be necessary to go to the code level directly with this information.

gpuaddr	useraddr	size	id	flags	type	usage	glen
00000000	00000000	204800	29	--l-p	gpumem	texture	5
00000000	00000000	4096	48	----p	gpumem	any(0)	1
00000000	00000000	16384	49	----p	gpumem	texture	4
00000000	00000000	65536	53	-r--p	gpumem	command	16
00000000	00000000	8192	54	----p	gpumem	texture	2
00000000	00000000	65536	69	-r--p	gpumem	command	16
00000000	00000000	65536	70	-r--p	gpumem	command	16
00000000	00000000	65536	71	-r--p	gpumem	command	16
00000000	00000000	4096	1	----p	gpumem	arraybuffer	1
00000000	00000000	16384	75	----p	gpumem	texture	4
00000000	00000000	16384	76	----p	gpumem	texture	4
00000000	00000000	196608	89	--l-p	gpumem	texture	3
00000000	00000000	4096	90	----p	gpumem	any(0)	1
00000000	00000000	2359296	24	--L--	ion	egl_image	5
00000000	00000000	188416	33	-----	ion	egl_surface	16
00000000	00000000	188416	52	-----	ion	egl_surface	16
00000000	00000000	188416	34	-----	ion	egl_surface	16

Debug GSL Memory Leakage (cont.)

- Extensive logging for GSL memory
  - Debugging version of Adreno library is needed
  - log.vmem=1
  - GSL memory log (vmem log) to be stored in /data/local/tmp per process base
    - Need write permission in /data/local/tmp before trying this
    - vmem_{pid}.txt to be generated
  - This log has code line number of alloc/free/map/unmap for GSL memory operation
    - The OEM cannot look at or narrow down further with vmem log, since the Adreno driver is not Open Source.
    - The QTI team can narrow down this issue with the vmem log.

```
Success::malloc, addr=0x47e1d000, gpuaddr=0x101bc000, size=0x00001000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_program.c, 3992
Success::malloc, addr=0x47f34000, gpuaddr=0x101be000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x48182000, gpuaddr=0x10200000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x481c2000, gpuaddr=0x10242000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x48483000, gpuaddr=0x10284000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x489a2000, gpuaddr=0x102c6000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x48c05000, gpuaddr=0x10308000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x48f46000, gpuaddr=0x1034a000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x48f86000, gpuaddr=0x1038c000, size=0x00010000, vendor/qcom/proprietary/gles/adreno200/rb/src/rb_cmdbuffer.c, 368
Success::malloc, addr=0x48f96000, gpuaddr=0x1039e000, size=0x00001000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_program.c, 3992
Success::malloc, addr=0x4a5b3000, gpuaddr=0x103a0000, size=0x00acc000, vendor/qcom/proprietary/gles/adreno200/rb/src/rb_textureformat.c, 2327
Success::malloc, addr=0x4b07f000, gpuaddr=0x10e6e000, size=0x00001000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_program.c, 3992
Success::malloc, addr=0x4b080000, gpuaddr=0x10e70000, size=0x00020000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_primitive.c, 463
Success::mmap(ION), hostptr=0x4b0a0000, len=0x007f8000, offset=0x00000000, gpuaddr=0x10f00000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2503
Success::malloc, addr=0x477f2000, gpuaddr=0x10e92000, size=0x00010000, vendor/qcom/proprietary/gles/adreno200/rb/src/rb_cmdbuffer.c, 368
Success::malloc, addr=0x55277000, gpuaddr=0x11700000, size=0x002fd000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1761
Success::malloc, addr=0x55574000, gpuaddr=0x11a00000, size=0x002fd000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1761
Success::mmap(ION), hostptr=0x55871000, len=0x007f8000, offset=0x00000000, gpuaddr=0x11d00000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2503
Success::malloc, addr=0x56069000, gpuaddr=0x12500000, size=0x002fd000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1761
Success::malloc, addr=0x56366000, gpuaddr=0x12800000, size=0x002fd000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1761
Success::mmap(ION), hostptr=0x56663000, len=0x007f8000, offset=0x00000000, gpuaddr=0x12b00000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2503
```

Issue Example

- OEMs are unable to perform all of the following debugging steps because the Adreno driver is not Open Source
  - However, it is useful for the OEM to understand how to approach GSL memory leakage
- Determine the issue:
  - Keep running the test; the kernel log shows the following error:
    - [7701.922907] kgsl: kgsl_mmu_get_gpuaddr: gen_pool_alloc(16388096) failed, pool: general_pool
    - [7701.922912] kgsl: kgsl_mmu_get_gpuaddr: [1855] allocated=913301504, entries=172
  - KGSL tried to get 16388096 size of memory, but it already allocated 913301504
- Monitor the GSL memory size for the process
  - adb shell cat /sys/kernel/debug/kgsl/proc/{PID}/mem
  - Ion memory kept increasing
  - Monitor the Ion memory size
  - adb shell cat sys/class/kgsl/kgsl/proc/<pid>/ion

Issue Example (cont.)

- Enable vmem log
- Search **ION** in the log and confirm Ion memory never unmapped in the Adreno driver

```
Line 34: Success=mmap(ION), hostptr=0x7c522000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc0100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 44: Success=mmap(ION), hostptr=0x7de6d000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc1100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2949
Line 51: Success=mmap(ION), hostptr=0x7f192000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc2100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2949
Line 54: Success=mmap(ION), hostptr=0x80132000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc3100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2949
Line 57: Success=mmap(ION), hostptr=0x82f47000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc4100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 105: Success=mmap(ION), hostptr=0x7f145000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc1100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 108: Success=mmap(ION), hostptr=0x800e5000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc2100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2949
Line 114: Success=mmap(ION), hostptr=0x81fa7000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc3100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2949
Line 116: Success=mmap(ION), hostptr=0x82f47000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc4100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2949
Line 117: Success=mmap(ION), hostptr=0x7db50000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc5100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 120: Success=mmap(ION), hostptr=0x85c12000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc6100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 123: Success=mmap(ION), hostptr=0x86bb2000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc7100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 125: Success=mmap(ION), hostptr=0x87b52000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc8100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 127: Success=mmap(ION), hostptr=0x88af2000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xc9100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 130: Success=mmap(ION), hostptr=0x89a92000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xca100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 132: Success=mmap(ION), hostptr=0x8aa32000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xcb100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 134: Success=mmap(ION), hostptr=0x8b9d2000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xcc100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 137: Success=mmap(ION), hostptr=0x8c972000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xcd100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 139: Success=mmap(ION), hostptr=0x8d912000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xce100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
Line 141: Success=mmap(ION), hostptr=0x84c72000, len=0x00fa0000, offset=0x00000000, gpuaddr=0xcf100000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1785
```

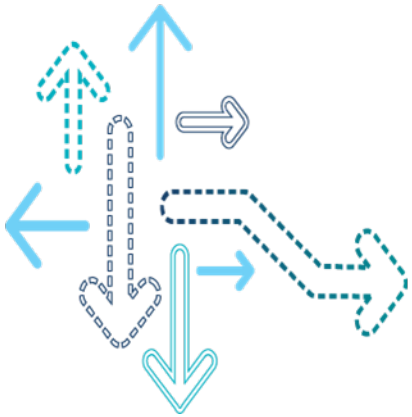
Issue Example (cont.)

- Review the Adreno source code to confirm when Ion is mapped and unmapped
  - Ion memory mapped to GPU IOMMU
    - eglCreateImageKHR eventually maps Ion memory to GPU IOMMU
  - Ion memory unmapped from GPU IOMMU
    - eglDestroyImageKHR eventually unmaps Ion memory from GPU IOMMU
- Theory – eglDestroyImageKHR was not called by the process
  - Add a log to confirm
  - However, it was called normally by the process
  - Are there any if-condition to skip unmap inside eglDestroyImageKHR?

Issue Example (cont.)

- Add more logs to understand the code flow to unmap Ion
  - Found the eglImage reference count was not 0
  - If the reference count is not 0, the eglImage is still being used
- Add logs GL texture functions
  - glBindTexture, glDeleteTextures, glEGLImageTargetTexture2DOES
  - To make sure matching textureID is not deleted
- Conclusion for this case
  - glDeleteTextures is not called by the process
  - The make eglImage reference count keep is nonzero
  - eglImage could not be destroyed
  - Ion memory kept increasing

GPU IOMMU Page Fault

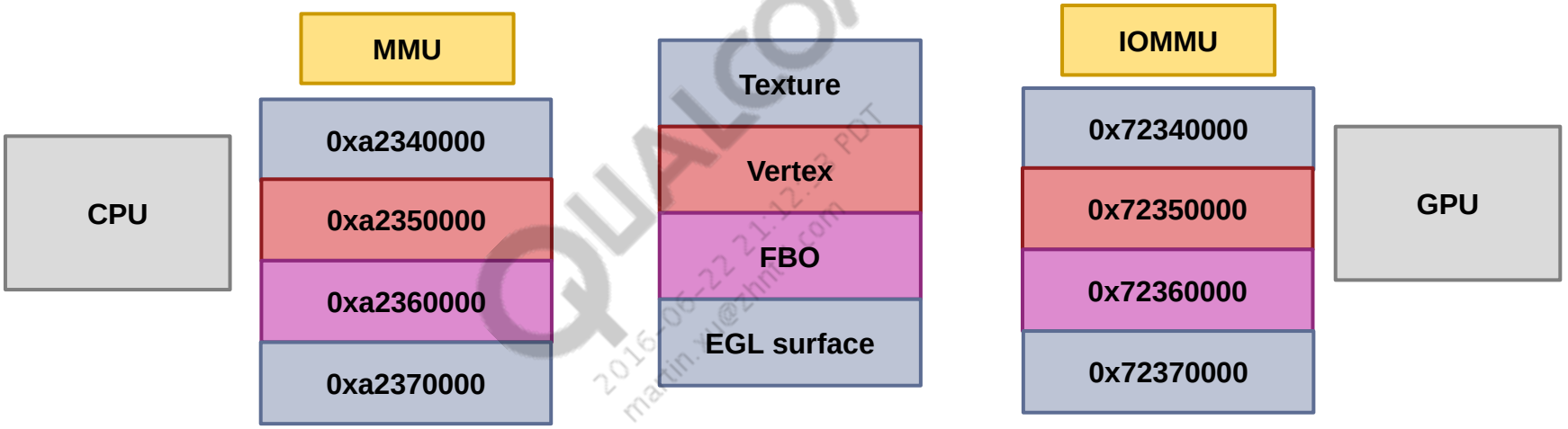


GPU Page Fault

- GPU memory is managed by the GPU IOMMU
- The GPU IOMMU could cause a page fault by accessing the incorrect virtual address
- Symptoms of a GPU page fault
  - UI corruptions
  - GPU hangs
  - Other unpredictable behavior
- IOMMU
  - A dedicated hardware block that allows virtually contiguous memory to be backed by physically noncontiguous pages
  - The virtual-to-physical memory translation logic in the IOMMU is the same as the logic in the CPU MMU
    - An important difference from the GPU MMU is that the IOMMU is a dedicated hardware block outside of the GPU; whereas, the GPU MMU is an inbuilt hardware block within the GPU

CPU MMU and GPU IOMMU

- Even though CPU and CPU can access the same physical memory, the CPU address and GPU address does not need to be the same.



How to Read GPU Page Fault Log

- The following is a typical GPU page fault log:

```
<2>[ 989.537402 / 07-25 11:07:34.173] kgsl kgsl-3d0: |kgsl_iommu_fault_handler| GPU PAGE FAULT: addr = 6A223000 pid = 7213
<2>[ 989.547111 / 07-25 11:07:34.183] kgsl kgsl-3d0: |kgsl_iommu_fault_handler| context = 0 FSR = 2 FSYNR0 = 582 FSYNR1 = 60030008(read fault)
<3>[ 989.575793 / 07-25 11:07:34.213] kgsl kgsl-3d0: ---- nearby memory ----
<3>[ 989.582563 / 07-25 11:07:34.223] kgsl kgsl-3d0: [6A219000 - 6A21A000] (+guard) (pid = 7213) (elementarraybuffer)
<3>[ 989.592558 / 07-25 11:07:34.233] kgsl kgsl-3d0:  <- fault @ 6A223000
<3>[ 989.598834 / 07-25 11:07:34.233] kgsl kgsl-3d0: [6A229000 - 6A22A000] (+guard) (pid = 7213) (texture)
```

  - Page fault occurred during accessing 0x6A223000(GPU address)
  - Page fault occurred in PID=7213
  - This is a read fault
  - Nearby memories were arraybuffer and texture
    - However, the memory 0x6A223000 is unknown
- If PID = 0 is observed, verify that the device works on a per-process page table
  - \kernel\arch\arm\configs\msm{Chipset}_defconfig
  - CONFIG_KGSL_PER_PROCESS_PAGE_TABLE=y
  - If PPPP is disabled, enable it and test it again

Debug Page Fault

- Determine if the page fault address is valid.
  - From the kernel log, get the page fault address.
  - If the page fault address is above or equal to 0x10000000, this is a valid GPU address.
  - If the page fault address is below 0x10000000, this is a invalid GPU address.

Debug Page Fault (cont.)

- If the page fault address is a valid GPU address:
  - Major effort should be made by QTI, since the Adreno driver is not Open Source.
  - This type of page fault is mostly caused by a timing issue.
  - The only valid debugging approach is using vmem log.
    - Debugging version of the Adreno lib is needed.
    - log.vmem=1
    - Determine which function allocates the memory; analyze why this memory is deallocated while the GPU is still using it.

```
Success::malloc, addr=0x47e1d000, gpuaddr=0x101bc000, size=0x00001000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_program.c, 3992
Success::malloc, addr=0x47f34000, gpuaddr=0x101be000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x48182000, gpuaddr=0x10200000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x481c2000, gpuaddr=0x10242000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x48483000, gpuaddr=0x10284000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x489a2000, gpuaddr=0x102c6000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x48c05000, gpuaddr=0x10308000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x48f46000, gpuaddr=0x1034a000, size=0x00040000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_binning.c, 261
Success::malloc, addr=0x48f86000, gpuaddr=0x1038c000, size=0x00010000, vendor/qcom/proprietary/gles/adreno200/rb/src/rb_cmdbuffer.c, 368
Success::malloc, addr=0x48f96000, gpuaddr=0x1039e000, size=0x00001000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_program.c, 3992
Success::malloc, addr=0x4a5b3000, gpuaddr=0x103a0000, size=0x00acc000, vendor/qcom/proprietary/gles/adreno200/rb/src/rb_textureformat.c, 2327
Success::malloc, addr=0x4b07f000, gpuaddr=0x10e6e000, size=0x00001000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_program.c, 3992
Success::malloc, addr=0x4b080000, gpuaddr=0x10e70000, size=0x00020000, vendor/qcom/proprietary/gles/adreno200/rb/src/hwl/oxili/oxili_primitive.c, 463
Success::mmap(ION), hostptr=0x4b0a0000, len=0x007f8000, offset=0x00000000, gpuaddr=0x10f00000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2503
Success::malloc, addr=0x477f2000, gpuaddr=0x10e92000, size=0x00010000, vendor/qcom/proprietary/gles/adreno200/rb/src/rb_cmdbuffer.c, 368
Success::malloc, addr=0x55277000, gpuaddr=0x11700000, size=0x002fd000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1761
Success::malloc, addr=0x55574000, gpuaddr=0x11a00000, size=0x002fd000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1761
Success::mmap(ION), hostptr=0x55871000, len=0x007f8000, offset=0x00000000, gpuaddr=0x11d00000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2503
Success::malloc, addr=0x56069000, gpuaddr=0x12500000, size=0x002fd000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1761
Success::malloc, addr=0x56366000, gpuaddr=0x12800000, size=0x002fd000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 1761
Success::mmap(ION), hostptr=0x56663000, len=0x007f8000, offset=0x00000000, gpuaddr=0x12b00000, vendor/qcom/proprietary/gles/adreno200/egl14/src/linux/android/eglSubDriverAndroid.c, 2503
```

Debug Page Fault (cont.)

- If the page fault address is a valid GPU address (cont.)
  - Two major causes of this type of page fault:
    - Timing page faults (premature free)
      - Timing page faults are usually caused by timestamp issues.
      - Each block of memory is assigned a timestamp, which determines the lifetime of that block of memory.
      - Once the timestamp has expired, the memory is freed and reassigned to the memory pool for reuse.
      - If the timestamp is not updated appropriately, the memory may be freed before being used causing the GPU to access an invalid address and page fault.
      - This is called a premature free. These types of page faults can be tracked down using VMEM logging.
      - VMEM logging records all calls to malloc, free, mmap, and unmap at the GSL layer.
      - This can be used to help determine where in the code the memory block was allocated and freed.
    - Invalid indexing/offset/padding page faults
      - These types of page faults are usually caused by programming errors, i.e., math calculations and off by 1 errors.
        - The issue is reproduced with excessive logging – Indicates not a timing issue.
        - The page fault address does not fall within a previously allocated/freed block of memory.
        - The page fault address occurs just after (within ~4096 bytes) the end of the nearest block of memory.

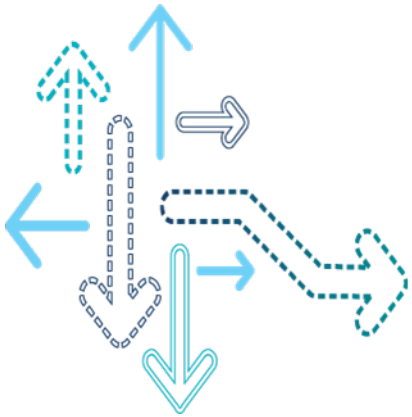
Debug Page Fault (cont.)

- If the page fault address is an invalid GPU address:
  - GPU snapshot is needed
    - Look at the memory addresses in GPU command packets
- How to make force the GPU hang after a page fault

```
diff --git a/drivers/gpu/msm/adreno.h b/drivers/gpu/msm/adreno.h
index b3a4fb5..72adb9d 100644
--- a/drivers/gpu/msm/adreno.h
+++ b/drivers/gpu/msm/adreno.h
@@ -488,7 +488,7 @@ struct log_field {
#define KGSL_FT_PAGEFAULT_GPUHALT_ENABLE     BIT(1)
#define KGSL_FT_PAGEFAULT_LOG_ONE_PER_PAGE   BIT(2)
#define KGSL_FT_PAGEFAULT_LOG_ONE_PER_INT    BIT(3)
-#define KGSL_FT_PAGEFAULT_DEFAULT_POLICY     KGSL_FT_PAGEFAULT_INT_ENABLE
+#define KGSL_FT_PAGEFAULT_DEFAULT_POLICY     KGSL_FT_PAGEFAULT_GPUHALT_ENABLE
#define ADRENO_FT_TYPES \
  { BIT(KGSL_FT_OFF), "off" }, \
```

- How to get a GPU snapshot to be handled in a GPU hang session

GPU Hang



What is GPU Hang

- A GPU hang occurs when the GPU pipeline is stuck and cannot recover without a reset.
- Before GPU Fault Tolerance (FT), most GPU hangs impacted the user, such as UI corruption or system-wide stability issue. However, after GPU FT, many GPU hangs can get recovered without user impact.

GPU FT

- Algorithm
  - Reset the GPU if a hang is detected; replay IB.
  - If a GPU hang is not detected again, GPU FT is done.
    - Most timing-related GPU hang issues can be avoided.
  - Reset the GPU and skip IB causing the GPU hang if a hang is detected again.
  - If a GPU hang is not detected again, GPU FT is done.
    - A GPU hang with IB corruption can be avoided.
  - Reset the GPU and mark context bad, with which Android can kill the process if a hang occurs again.
    - The user will observe an app crash, but a system-wide stability issue can be avoided.

How Driver Detects GPU Hang

- GPU hang interrupt
  - If GPU hang interrupts are raised, the KGSL detects a GPU hang.
- GPU hang timer in the KGSL
  - Hang timer checks **timeout** if none of the essential registers have changed.
  - If they did not change, the kernel declares that the GPU is hung.
  - The registers checked by fast hang detection have the following information:
    - Overall GPU status (RBBM status register)
    - Command buffer status for ring buffer and IB1/IB2 consumption (buffer base address/buffer size remained)
    - Number of cycles when ALU is working
    - Number of L1 instructions cache misses in the shader pipe
    - Number of fragment shader flow instructions
  - If none of them is changed for 200 ms, the KGSL declares GPU hang.
- Once KGSL detects a GPU hang, it dumps a GPU snapshot and simple postmortem dump.

Kernel Log Analysis for GPU Hang

- Kernel log has very limited information for a GPU hang.
  - However, we can be aware of GPU hang with the following types of kernel log.

```
kgsl kgsl-3d0: |adreno_ft_detect| Proc system_server, ctxt_id 4 ts 9847 triggered fault tolerance on global ts 13204
kgsl kgsl-3d0: STATUS E54F4003 | IB1:668A0284/000005AE | IB2: 6B99C000/0000018E | RPTR: 04C0 | WPTR: 05AB
kgsl kgsl-3d0: |adreno_snapshot| GPU snapshot froze 3060Kb of GPU buffers
kgsl kgsl-3d0: |kgsl_device_snapshot| snapshot created at pa 37a00000 size 156192
kgsl kgsl-3d0: |adreno_ft_detect| Proc system_server, ctxt_id 4 ts 9847 triggered fault tolerance on global ts 13204
kgsl kgsl-3d0: |adreno_idle| spun too long waiting for RB to idle
kgsl kgsl-3d0: |_adreno_ft| Replay status: 1
kgsl kgsl-3d0: |adreno_ft| policy 0x6 status 0x0
```

    - STATUS E54F4003 – RBBM status register, which we are using as GPU hang signature. Even though the same signature does not mean the same cause, it is useful way to categorize a GPU hang since it indicated the overall GPU status, i.e., which internal GPU block is busy.
    - IB1– 668A0284/000005AE | IB2: 6B99C000/0000018E – IB1 and IB2 buffer status; the first number shows the IB1/IB2 base address and the second number shows the remaining buffer size that was supposed to be consumed by GPU.
    - |kgsl_device_snapshot| snapshot created at pa 37a00000 size 156192 – This is the GPU snapshot created at 0x37a00000 with size of 156192.
    - Debugging a GPU hang with only a kernel log is insufficient. A GPU snapshot is required.

GPU Snapshot

- The kernel log has very limited information for a GPU hang.
- A GPU snapshot has important GPU information when a GPU hang occurs.
  - All the important GPU register values
  - Command ringbuffer contents
  - Current IB command, which GPU is executing
  - Shader memory
- A single binary file
  - QTI will parse it to get relevant information
- Whenever a GPU hang occurs, a GPU snapshot is needed.
  - **Cannot investigate a GPU hang without a GPU snapshot**

How to Get a GPU Snapshot

- If a GPU hang device is alive, a GPU snapshot can be pulled out easily.
  - Check the last timestamp to determine if a snapshot was taken; it should be nonzero.
    - adb cat sys/class/kgsl/kgsl-3d0/snapshot/timestamp
  - After the snapshot is taken, the binary output can be copied to the host machine.
    - adb pull /sys/class/kgsl/kgsl-3d0/snapshot/dump GPUsnapshot.bin
  - This file contains only the first GPU snapshot.
    - If there are consecutive GPU hangs, they are not saved.
  - This file disappears if the device gets reset.
- Adreno driver supports to set the GPU snapshot saving path.
  - gpuSnapshotPath={a permanent path where GPU snapshot is saved}, e.g.:
    - gpuSnapshotPath=/data/local/tmp; write permission is needed
  - GPU snapshot is saved in /data/local/tmp.
  - All of GPU snapshots are stored in the path specified by the user.

How to Get a GPU Snapshot (cont.)

- If a GPU hang causes a stability issue and RAM dump is available, get a GPU snapshot from RAM dump.
  - Load RAM dump to Trace32 (T32).
  - Save the binary with GPU snapshot physical address and size in the kernel log.
    - data.save.binary <filename> a:0x**<Physical address>**++**<range>**, e.g.:
      - data.save.binary GPUsnapshot.bin a:0x37a00000++26220

```
kgsl kgsl-3d0: |adreno_ft_detect| Proc system_server, ctxt_id 4 ts 9847 triggered fault tolerance on global ts 13204
kgsl kgsl-3d0: STATUS E54F4003 | IB1:668A0284/000005AE | IB2: 6B99C000/0000018E | RPTR: 04C0 | WPTR: 05AB
kgsl kgsl-3d0: |adreno_snapshot| GPU snapshot froze 3060Kb of GPU buffers
kgsl kgsl-3d0: |kgsl_device_snapshot| snapshot created at pa 37a00000 size 156192
kgsl kgsl-3d0: |adreno_ft_detect| Proc system_server, ctxt_id 4 ts 9847 triggered fault tolerance on global ts 13204
kgsl kgsl-3d0: |adreno_idle| spun too long waiting for RB to idle
kgsl kgsl-3d0: |_adreno_ft| Replay status: 1
kgsl kgsl-3d0: |adreno_ft| policy 0x6 status 0x0
```

GPU Hang Examples

- Given a GPU snapshot, QTI can narrow down where a GPU hang occurs
- Shader corruption case
  - Four GPU hang occurs with the same pattern of shader corruption
  - Beginning of the vertex shader, the binary shows 0xff000000 values, which caused a hang
  - It was a memory corruption issue rather than a driver issue

00000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000008	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000010	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000018	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000020	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000028	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000030	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000038	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000040	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000048	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000050	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000058	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000060	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000068	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000	FF000000
00000070	00840847	14480C29	01021158	79F08003	D03000A4	C050196E	981F0915	47000008
00000078	040457D8	718E82C3	00895BEC	20870220	17001A46	02947301	A021790C	0A071001
00000080	00784D65	91030438	30580039	0818321A	10020825	C8020042	14102CBA	50780913
00000088	002C9188	4340829D	1082389E	10000465	32008859	0114481B	2C00040A	804670CA
00000090	06009400	C8803C20	02021841	28230634	04A42047	2A10249A	009004AA	00044250
00000098	40660D03	62980185	50B05404	002C0028	09443507	4262C515	320158C2	C1062400
000000A0	010754D2	63025A05	40000046	02842A00	18678804	40E20317	00028020	0480220C
000000A8	3E69C4A4	600A023E	04A8E11D	01B00478	D000541C	00E80258	008431A0	92C80800
000000B0	007D0095	6049625B	00004822	20A42115	80850245	0234040B	0010042D	262E2258
000000B8	2940405B	20920100	004D4850	441802F4	20121A00	8A41A412	02222430	48200000
000000C0	22001010	C0001182	80C50081	62968A02	30604443	C20C2520	50C90022	861B0029

GPU Hang Examples (cont.)

- Depth buffer size issue
  - A GPU hang occurs Z test enabled, but the depth buffer base was 0, which was invalid.
  - There was an Adreno driver bug, so QTI could narrow down the hang issue and fixed it.

SHADER_Z_ENABLE (0:0)	0x0
Z_TEST_ENABLE (1:1)	0x1
Z_WRITE_ENABLE (2:2)	0x0
LATE_Z_ENABLE (3:3)	0x0
Z_TEST_FUNC (6:4)	0x3
Z_CLAMP_ENABLE (7:7)	0x0
Z_READ_ENABLE (31:31)	0x1

DATA

RB_DEPTH_BUF_INFO	0x00000000
DEPTH_FORMAT (1:0)	0x0
DEPTH_BUF_BASE (30:4)	0x0

Long IB Detection (QoS)

- Long IB detection is a situation where a GPU command is processed by the GPU for too long.
  - Timeout is 2 sec currently
    - kernel/drivers/gpu/msm/adreno_dispatch.c

      /* Command batch timeout (in milliseconds) */

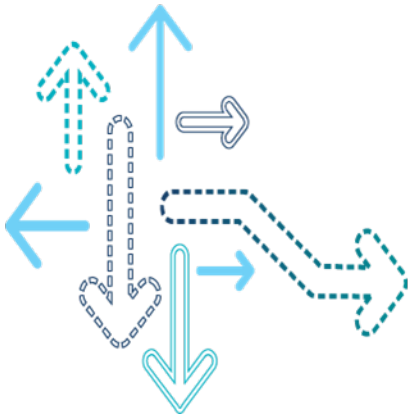
      static unsigned int _cmdbatch_timeout = 2000;
  - Though it is not a GPU hang and the GPU is running to process the GPU command, the KGSL is waiting too long time to finish the job.
  - If the KGSL does not define a timeout, the user may experience system-wide UI stuck with a long IB, since the GPU may be occupied for a very long time by context with a long IB.
  - If a long IB is detected, the KGSL marks the context bad and the process will be killed eventually.
- Distinguishing a long IB detection
  - Kernel log shows a different log than a GPU hang

```
<3>[40879.596442] c0  2200 kgsl kgsl-3d0: ogle.android.gm[21367]: gpu timeout ctx 8 ts 1220
<3>[40879.596479] c0  2200 kgsl kgsl-3d0: ogle.android.gm[21367]: gpu failed ctx 8 ts 1220
```

Long IB Detection (QoS) (cont.)

- Typical causes of a long IB
  - App bug – One of typical long IB generation scenarios is infinite or heavy for-loop in the shader. If the app implemented an invalid shader code, it may cause long IB.
  - WebGL site targeting desktop environment – Some WebGL sites cause a long IB since the site targets high performance desktop GPU. Those may make even a desktop PC slower, since it generates a too heavy operation.
- Confirming long IB detection
  - Disable long IB detection – This may cause other issues such as a fence timeout.
  - adb shell "echo 0 > > /sys/class/kgsl/kgsl-3d0/ft_long_ib_detect"

GPU Power and Performance



KGSL Power Event Trace (F-Trace)

- GPU power event logs through an F-Trace enabled environment can be analyzed for GPU power-related issues.
- OEMs are required to provide KGSL power event trace logs using the following method:
  - Mount debugfs
    - adb shell mount -t debugfs none /sys/kernel/debug
  - See available events
    - adb shell cat /sys/kernel/debug/tracing/available_events
    - adb shell cat /sys/kernel/debug/tracing/available_events | grep kgsl
  - Make a text file with the desired events
    - kgsl:kgsl_<adreno_chip>_irq_status
    - kgsl:kgsl_clk
    - kgsl:kgsl_irq
    - kgsl:kgsl_rail
    - kgsl:kgsl_bus
    - kgsl:kgsl_pwrlevel
    - kgsl:kgsl_pwr_set_state
    - kgsl:kgsl_pwr_request_state

For MSM8974, adreno_chip is a3xx; therefore, kgsl_a3xx_irq_status should be enabled. For chipsets using Adreno 4xx, like Adreno 420 on MSM8084, adreno_chip is a4xx; therefore, kgsl_a4xx_irq_status should be used.

KGSL Power Event Trace (F-Trace) (cont.)

- Provide KGSL power event trace logs using the following method: (cont.)
  - Push it
    - adb push events.txt /sys/kernel/debug/tracing/set_event
  - Run the usecase showing the power issue
    - No need to reset device
  - Pull the log
    - adb pull /sys/kernel/debug/tracing/trace

KGSL Power Kernel Logs

- On targets that do not support power trace, power issues could be analyzed by enabling maximum logging levels for the kgsl_power modules.
- To enable the maximum power logs in the KGSL:
  - adb shell mkdir /data/debug
  - adb shell mount -t debugfs debugfs /data/local/tmp/debug
  - echo 7 > /data/local/tmp/debug/kgsl/log_level_pwr
- KGSL power kernel logs are not supported in newer chipsets having kernel version higher than 3.1x.

GPU Frequency

- The KGSL power control subsystem exposes sysfs entries that could be used to peek the current GPU frequency and also set the maximum frequency.
- To read GPU frequency:
  - adb shell mount –t debugfs none /sys/kernel/debug
  - adb shell cat /sys/kernel/debug/clk/gfx3d_clk/measure
- **Note:** ‘measure’ is the exact clock frequency as supplied by the clock driver. ‘rate’ is the clock value as requested by the software.

GPU DCVS and GPU Bus DCVS (Post Kernel 3.2)

- GPU DCVS is based on the Linux Devfreq framework, whose changing clock levels based on the governor setting is exposed through a new sysfs node.
- Depending on the granularity of the available system bus frequencies and GPU clock frequencies, some chips optimize bus bandwidth request from the GPU through the GPU bus DCVS, while the others (with less number of available clocks) have GPU DCVS controlling both GPU clocks and bus bandwidth request votes.

GPU BUS DCVS

- To determine whether your target supports GPU bus DCVS, check the following sysfs node:
  - adb shell cat /sys/class/kgsl/kgsl-3d0/bus_split
  - 1 – GPU bus DCVS is enabled
  - 0 – GPU bus DCVS is disabled
- GPU bus DCVS, for targets that support it, can be disabled, and doing so will statically map the GPU clock to its default bus bandwidth voting. For targets that do not support GPU bus DCVS, enabling it does not have any effect.

Devfreq Framework and GPU DCVS Governor

- Devfreq framework is a governor model-based device frequency controlling framework (a kind of DVFS) that is much similar to CpuFreq framework.
- For more information about Devfreq framework and its overview, any internet search with “Devfreq, Linux” keywords will provide a vast resource from its implementation design and codes.

Devfreq Framework and GPU DCVS Governor (cont.)

- For KGSL, GPU DCVS governor is exposed through the following sysfs node:
  - /sys/class/kgsl/kgsl-3d0/devfreq

Sub nodes from devfreq	Usage and description
available_frequencies	adb shell cat /sys/class/kgsl/kgsl-3d0/devfreq/available_frequencies will list supported GPU clocks that will be used by GPU DCVS
available_govnerors	adb shell cat /sys/class/kgsl/kgsl-3d0/devfreq/available_governors will list supported governors that can be used by GPU DCVS*
Governor	adb shell cat /sys/class/kgsl/kgsl-3d0/devfreq/governor will show the currently set governor for GPU DCVS. Default governor is msm-adreno-tz
	adb shell echo governor_name > /sys/class/kgsl/kgsl-3d0/devfreq/governor will set the governor named governor_name to be used by GPU DCVS. The names of possible govenors can be acquired by catting available_governors as shown earlier.

    - Devfreq is commonly used by other subsystems, including GPU; currently, GPU DCVS only supports the following governors – msm-adreno-tz (default), performance, and powersave.
- To set GPU DCVS to be at a maximum performance level:
  - adb shell “echo performance > /sys/class/kgsl/kgsl-3d0/devfreq/governor”
  - This sets the GPU to run at maximum clock all the time (no clock changes).
- To set GPU DCVS to be at a minimum performance level (maximum power saving level):
  - adb shell “echo powersave > /sys/class/kgsl/kgsl-3d0/devfreq/governor”
  - This sets the GPU to run at minimum clock all the time (no clock change).

Debugging Performance Issues

- GPU performance issues arise from UI use case and graphics benchmarks apps.
- Typically, performance issues such as frame drops, low fps, and janky UI are caused by not only GPU performance, but also CPU performance and other subsystem performance issues. It is very critical to determine whether a given performance issue is caused by a GPU performance or other non-GPU (typically caused by CPU and/or bus).
- The starting point for performance issue analysis is, therefore, to check whether the bottleneck is at tge CPU, GPU, or CPU and GPU.

Debugging Performance Issues (cont.)

- Provide the following data for further issue analysis:
  - Systrace capture with all options turned on
    - For Systrace capturing instruction, visit:
      - http://developer.android.com/tools/help/systrace.html
      - http://developer.android.com/tools/debugging/systrace.html
  - CPU performance mode/GPU performance mode results

Checkpoint	What to check	How to set Performance mode
CPU bottleneck	Check the CPU clock frequencies when the issue occurs. If CPU clocks are running at low freqs, put CPUs in performance mode and check the issue.	>adb shell stop mpdecision >adb shell stop thermal-engine >sleep 1 >adb shell "echo 1 > /sys/devices/system/cpu/cpu1/online" >adb shell "echo 1 > /sys/devices/system/cpu/cpu2/online" >adb shell "echo 1 > /sys/devices/system/cpu/cpu3/online" >sleep 1 >adb shell "echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor" >adb shell "echo performance > /sys/devices/system/cpu/cpu1/cpufreq/scaling_governor" >adb shell "echo performance > /sys/devices/system/cpu/cpu2/cpufreq/scaling_governor" >adb shell "echo performance > /sys/devices/system/cpu/cpu3/cpufreq/scaling_governor" >sleep 1
GPU bottleneck	Check the GPU clock frequency when the issue occurs If GPU clock is running at low freq, put GPU in performance mode.	>adb shell "echo 1 > /sys/class/kgsl/kgsl-3d0/force_clk_on" >adb shell "echo 10000000 > /sys/class/kgsl/kgsl-3d0/idle_timer" >adb shell "echo performance > /sys/class/kgsl/kgsl-3d0/devfreq/governor"
CPU/GPU bottleneck	Put both CPU and GPU in performance mode.	Use both CPU/GPU performance mode settings above.

Debugging Thermal Issues

- For some use cases and OEM settings, GPU DCVS may be capped because of thermal mitigation. QTI's default thermal mitigation deamon called thermal-engine can be modified by OEMs to throttle maximum GPU clk to reduce temperature on the GPU.
- /sys/class/kgsl/kgsl-3d0/thermal_pwrlevel can be read to check whether thermal mitigation occurred on the GPU and capped the GPU to run at nonmaximum performance level.
- To check the impact of thermal mitigation, do the following:
  - $adb shell "cat > /sys/class/kgsl/kgsl-3d0/thermal_pwrlevel"
    - 0 – There is *no* thermal migitation on the GPU
    - Nonzero values – Indicates thermal migration kicked in to cap the GPU's maximum frequency

Debugging Thermal Issues (cont.)

- When thermal mitigation is observed, disable thermal mitigation (only for testing/verification purpose) temporarily and check the use case again.
- To disable thermal mitigation:
  - adb shell stop thermal-engine
- To check whether thermal-engine is running:
  - adb shell “ps thermal-engine”
  - If there is no process returned in the list, it indicates thermal mitigation off.

```
USER      PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
```

  - If there is a process returned in the list, it indicates thermal mitigation on.

```
USER      PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root      223   1     47004  1324    ffffffff 00000000 S /system/bin/thermal-eng
```

References

Ref.	Document	
Qualcomm Technologies		
Q1	*Application Note: Software Glossary for Customers*	CL93-V3077-1

Questions?

https://support.cdmatech.com