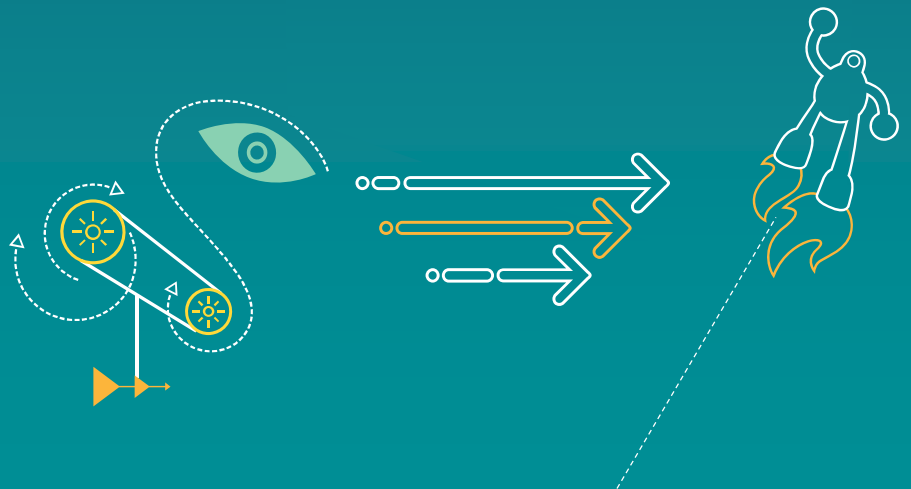
高通多媒体技术期刊　20140903

QUALCOMM®

Qualcomm Technologies, Inc.



Revision History

Revision	Date	Description
A	Sep 2014	Initial release

Contents

- Display
- Graphics
- Video

Display

Frame Buffer 分配管理

- 目前，在 Android 系统上，Frame Buffer Target (FBT) 被应用，这也意味着 利用GPU 合成的layer 被绘制到 FBT上，故 预留的物理连续 Framebuffer 内存可以被删除，这样对系统的memory优化有帮助。
  - 在正常的Android 应用场景，Frame Buffer Target(FBT) 被直接使用，且 FBT 所需内存 动态地从系统的ION memory heap中进行分配。
    当 apk 的layer 使用GPU 合成，那么 合成后的内容 是在FBT 上。
    当 apk 的layer 使用MDP 合成，那么 会使用overlay API，直接把显示的内容输出到LCD 屏幕上。
  - 在recovery mode 下，Qualcomm默认使用 overlay api。但在其它某些场景，如果 有的客户 需要 使用 FrameBuffer的 pan display API，那么，此时mmap 函数会被调用，从ION memory heap中动态分配内存， 直到这块 内存不再被使用，系统会自动释放。
- 比如，8974，8916，8939，8994，之前预留的8M的静态物理连续内存被删除，因为当前的显示驱动 overlay API 不再 使用 这块物理连续的内存。
- 为了对遗留平台的兼容支持，如果 用户 仍然要使用 FrameBuffer (pan display API)，那么这个块 memory 会通过mmap 从系统的ION memory 中分配得到， 一直到 FB 的参考计数的count到0，这块内存被释放。
- 参考下面代码：
  - 在 dtsi 文件中，例如，8974 平台，参考下面的 CAF link。

    https://www.codeaurora.org/cgit/quic/la/kernel/msm-3.10/tree/arch/arm/boot/dts/qcom/msm8974-mdss.dtsi?h=msm-3.10

    mdss_fb0: qcom,mdss_fb_primary {
    cell-index = <0>;
    compatible = "qcom,mdss-fb";
    - qcom,memory-reservation-type = "EBI1";
    - qcom,memory-reservation-size = <0x800000>; // remove this 8M memory
    qcom,memblock-reserve = <0x03200000 0x01E00000>;
    };
- 对于8916，可以参看下面link：
  - https://www.codeaurora.org/cgit/quic/la/kernel/msm-3.10/tree/arch/arm/boot/dts/qcom/msm8916-mdss.dtsi?h=msm-3.10

Frame Buffer 分配管理 - 续一

- 对于 连续显示功能，即continuous splash screen, 此时 显示内容 延续 LK display content，以 8916 为例，从下面的link，我们能看到：
  - https://www.codeaurora.org/cgit/quic/la/kernel/msm-3.10/tree/arch/arm/boot/dts/qcom/msm8916-mdss.dtsi?h=msm-3.10

    mdss_fb0: qcom,mdss_fb_primary {
    cell-index = <0>;
    compatible = “qcom,mdss-fb”;
    qcom,memblock-reserve = <0x83200000 0xfa0000>;

- 针对 “qcom,memblock-reserve”，这块特殊的memory 主要目的为 连续显示使用， 在开机过程中，从 LK 过渡到 kernel 时，需要设置 qcom,cont-splash-enabled，一旦出现 Android Boot animation，此block 内存就会被释放。

- 具体CAF links
  - Remove FB memory allocation (***)
    https://www.codeaurora.org/cgit/quic/la/kernel/msm-3.10/commit/?h=msm-3.10&id=f1d88ed5d465d5ad3fc3fffe4b7bdbaf95cb84f2
  - Allocate FB memory during mmap call (***)
    https://www.codeaurora.org/cgit/quic/la/kernel/msm-3.10/commit/?h=msm-3.10&id=cd3b75f365dafcb2e3977b80890e0754a882069b
  - Add backward compatibility to FB memory (***)
    https://www.codeaurora.org/cgit/quic/la/kernel/msm-3.10/commit/?h=msm-3.10&id=24589b39babf0f0d9967c244f78047c76a01f2da

- 相关的 Fix changes：
  - mdss: mdp: update fix screen variable in mmap call
    https://www.codeaurora.org/cgit/quic/la/kernel/msm-3.10/commit/?h=msm-3.10&id=d5d7a9694bced89703b1340a238f0180bbdc63ba
  - mdss: fb: ensure that shared memory is available during probe
    https://www.codeaurora.org/cgit/quic/la/kernel/msm-3.10/commit/?h=msm-3.10&id=1e45a92c6dcbb79f5f56395489afb187535aa5ce
  - mdss: fb: update smem_len when user updates parameters
    https://www.codeaurora.org/cgit/quic/la/kernel/msm-3.10/commit/?h=msm-3.10&id=b90090ee8dad738aae19fc4e60ccc01d7e2fdb78

- 参考Solution：00029644

UI Flickering 问题的调试

- 在Android系统中，会遇到 闪屏的问题，主要分为下面几大类：
  - 与 MDP 合成相关的 flickering
  - 与 GPU 相关的flickering
  - 与 framework/apk 本身相关的flickering

- 下面对其调试方法分别进行介绍

- 1. Disable HW overlay on settings apk, 此时，让所有的layers 使用GPU 进行合成。
  - 如果闪屏依然存在，需要查看GPU，framework, apk 相关领域。反之，需要查看MDP合成相关内容。

- 2. 让所有的layers 使用MDP 合成，请参看下面代码
  - 在 hardware/qcom/display/libhwcomposer/hwc_mdpcomp.cpp，MDPComp::tryFullFrame 函数

```
/* 注释掉下面代码
if(sIdleFallBack && !ctx->listStats[mDpy].secureUI) {
ALOGD_IF(isDebug(), "%s: Idle fallback dpy %d",__FUNCTION__, mDpy);
return false;
}
*/
```

- 3. 关闭混合 合成模式，即 MDP + GPU
  - 方法一：

```
adb shell setprop debug.mdpcomp.mixedmode.disable 1
adb shell stop
adb shell start
```

  - 方法二：

在 system/build.prop 文件中修改对应的flag，然后push到手机上

UI Flickering 问题的调试 - 续一

- 4. Disable PTOR(Peripheral Tiny Overlap Removal) 功能
  - 去查看flickering 问题 是否 与ptor 相关

    adb shell setprop persist.hwc.ptor.enable false
    adb shell stop
    adb shell start

- 5. Disable dirty-rect optimization for GPU
  - 去查看flickering 问题 是否 与 GPU 的dirty region 的优化 相关

    adb shell setprop debug.sf.gpu_comp_tiling 0
    adb shell stop
    adb shell start

- 6. Disable CABL 功能
  - 去查看flickering 问题 是否 与CABL 相关

    adb shell setprop ro.qualcomm.cabl 0
    adb shell stop
    adb shell start

- 7. 对于command mode panel来说，disable partial update function
  - 在panel driver dtsi 文件中， 删除下面flag，去查看 是否 与此功能相关

    qcom,partial-update-enabled

UI Flickering 问题的调试 - 续二

- 8. Dump apk 的layers，分类如下：
  - 分为 primary LCD display内容 和 external display内容
  - 分为 png 格式 和 raw 格式

- 对于LCD 主屏显示的layers dump 命令如下：

  adb shell setprop debug.sf.dump.enable true
  adb shell setprop debug.sf.dump.primary true
  adb shell stop
  adb shell start
  adb shell setprop debug.sf.dump 0 // Raw 格式，可以把video layer dump 下来，如果是png格式 (debug.sf.dump.png)，只能dump graphics layer.
  adb shell setprop debug.sf.dump 20 // 这里的是需要dump的帧数，根据需求，自行调整
  // 去复现问题
  adb shell setprop debug.sf.dump 0 // 停止dump
  adb pull /data/sf*

  然后把dump layers 的数据pull到电脑上去查看。

- 对于 external display layers dump 命令如下：

  需要把上面命令中 setprop debug.sf.dump.primary 改为 debug.sf.dump.external 即可。

- 参考solution：00028590

Graphics

KGSL Page Fault FAQ (Solution: 0028669)

Q1) 什么是GPU page fault? 怎么确认是不是GPU page fault?
A1) GPU 有自己独有的MMU (GPU IOMMU)去访问CPU-GPU 共享的内存。当GPU试图去访问一个非法的地址时，GPU page fault就发生了。
你可以在kernel log中很容易地找到GPU page fault，下面就是一个GPU page fault 的例子，包含有“GPU PAGE FAULT"字符串。

<2>[122867.712606] c1 81 kgsl kgsl-3d0: |kgsl_iommu_fault_handler| GPU PAGE FAULT: addr = 773E9B00 pid = 2770
<2>[122867.712617] c1 81 kgsl kgsl-3d0: |kgsl_iommu_fault_handler| context = 0 FSR = 80000002 FSYNR0 = 582 FSYNR1 = 3C030008(read fault)
<3>[122867.712633] c1 81 kgsl kgsl-3d0: ---- premature free ----
<3>[122867.712640] c1 81 kgsl kgsl-3d0: [773DB000-773FB000] (vertexarraybuffer) was already freed by pid 2770
<3>[122867.712649] c1 81 kgsl kgsl-3d0: ---- nearby memory ----
<3>[122867.712679] c1 81 kgsl kgsl-3d0: [773D6000 - 773DA000] (+guard) (pid = 2770) (texture)
<3>[122867.712686] c1 81 kgsl kgsl-3d0: <- fault @ 773E9B00
<3>[122867.712692] c1 81 kgsl kgsl-3d0: [773FC000 - 77400000] (+guard) (pid = 2770) (texture)

Q2) GPU page fault log的含义是什么?
A2) 你可以在log中找到产生GPU page fault 的进程PID 和GPU试图要去访问的地址。
NOTE: 这个地址是GPU 地址，不同于普通的CPU 虚拟地址。
<2>[122867.712606] c1 81 kgsl kgsl-3d0: |kgsl_iommu_fault_handler| GPU PAGE FAULT: addr = 773E9B00 pid = 2770
第二行包括IOMMU 寄存器的信息和page fault 类型 - read fault还是write fault
<2>[122867.712617] c1 81 kgsl kgsl-3d0: |kgsl_iommu_fault_handler| context = 0 FSR = 80000002 FSYNR0 = 582 FSYNR1 = 3C030008(read fault)
同时在fault地址附近的内存信息。
我们可以从这些信息中推测引起GPU page fault 的内存类型。
然而这并不意味着我们可以确定引起GPU page fault 的内心类型，因为那块内心实际上并没有影射到GPU page table。
为了确定内存类型，我们还需要更多的信息。这只是一个Hint而已。
<3>[122867.712679] c1 81 kgsl kgsl-3d0: [773D6000 - 773DA000] (+guard) (pid = 2770) (texture)
<3>[122867.712686] c1 81 kgsl kgsl-3d0: <- fault @ 773E9B00
<3>[122867.712692] c1 81 kgsl kgsl-3d0: [773FC000 - 77400000] (+guard) (pid = 2770) (texture)

KGSL Page Fault FAQ (Solution: 0028669) - 续一

Q3) 对GPU page fault 问题，我需要提供什么信息给QCOM?
A3) 一般地，我们需要知道QCOM build ID, 这个可以帮助我们检查QCOM内部记录，找到任何在你的build上面缺少的修复。
如果没有任何已知问题， 我们需要找到哪部分的代码引起这个问题。
GPU驱动分为两个部分，KGSL (内核部分)和 UMD (用户态驱动)
对KGSL 你可以直接查看code，这个在Linux kernel 目录下。然而QCOM并不共享UMD 驱动的源代码，而是作为共享库提供 (*.so 文件)。

(1) KGSL的一个功能就是管理context和page table。
如果你在kernel log里看到一个无效的PID 而且/或者一个无效的地址， 这很有可能是KGSL没有正确的处理page table 和context 引起的问题。
一个典型的无效PID 是0, 地址是FFFFxxxx
kgsl kgsl-3d0: |kgsl_iommu_fault_handler| GPU PAGE FAULT: addr = FFFF0000 pid = 0

如果你看到一个GPU pagefault有无效的PID 和地址，请先提供kernel log 和logcat log。

(2) 如果你在GPU page fault log里面看到有效的地址和PID， 这可能是UMD 驱动的问题，有可能GPU 内存被提前释放了。
UMD驱动的一个功能就是管理GPU使用的内存类型。GPU 内存有很多类型，比如texture, command, VBO, FBO, egl image 等。
使用不同内存的代码路径完全不同，因此第一步先要缩小是哪种类型的内存引起GPU page fault。
举例，如果你看到下面的log, GPU 在Process PID 2770里面试图去访问 773E9B00
<2>[122867.712606] c1 81 kgsl kgsl-3d0: |kgsl_iommu_fault_handler| GPU PAGE FAULT: addr = 773E9B00 pid = 2770
因此我们需要知道内存地址773E9B00 是什么类型。为了理解这个，我们需要用使能GPU Memory Logging的UMD 驱动库。
NOTE: 这个使能GPU Memory Logging的UMD 驱动库需要QCOM 为你单独重新编译提供，正式发布的库不包含这个功能。

KGSL Page Fault FAQ (Solution: 0028669) - 续二

你可以用下面方法动态打开GPU memory Logging的功能。
>vi adreno_config.txt
log.vmem=1
>adb push adreno_config.txt /data/local/tmp
>adb shell sync
>adb reboot
>adb wait-for-device
>adb shell chmod 777 /data/local/tmp
然后你可以在设备的/data/local/tmp路径下找到这些 vmem_{pid}.txt，这些Logging允许用户了解哪种类型的内存分配在哪个地址上。
如果你碰到这种类型的GPU page fault, 请提供这些vmem_{pid}.txt 和kernel log。

(3) 如果是两种类型的GPU page fault (无效的PID/地址和有效的PID/地址) 都有， 最好能提供这个page fault 对应的GPU snapshot。
KGSL 有一个办法可以在GPU page fault的时候触发一个GPU snapshot，你可以用下面方法在GPU page fault 时dump GPU snapshot。
>adb shell "echo 0x3 > > /sys/class/kgsl/kgsl-3d0/ft_pagefault_policy"

请提供GPU snapshot和其他信息给QCOM检查。

Video

Video问题各种Log的获取和用途

在各种Video问题的分析和解决中, 我们经常需要各种详细的Log来查看, 而为了减少Log量的需要, 这些Log往往并没有在版本中默认打开. 这就需要在提出case前, 大家针对不同问题的不同需要, 把相应的开关打开, 以便为初始的分析提供详细的依据.

以下是一个针对不同类型Video问题如何打开详细Log的指南.

视频编解码类Log(包括User space和Kernel Driver)

1. User space

>adb root
>adb shell setprop vide.debug.level 7
这个命令会打开User space中视频编解码相关详细Log.

2. Kernel Driver

>adb root
>adb shell
#cd /d/msm_vidc
#echo 0x101f > debug_level //打开video kernel driver详细Log
#echo 0x3f > fw_level //打开video firmware Log

另外Log抓取时需要在启动Video实例前, 以免遗漏重要信息.

视频编解码Input和Output buffer logs

在涉及到与流媒体或Camera sensor相关的解码/编码问题时, 我们需要查看视频编解码器的输入和输出以确定问题所在的模块, 这时就需要提供以下Log(针对解码和编码而有不同).

1. 视频解码Input和Output buffer logs

>adb root

>adb shell setprop vidc.dec.log.out 1 //enable output buffer log

>adb shell setprop vidc.dec.log.in 1 //enable input buffer log

文件缺省地址在/data/misc/media. 文件名如下:

input_dec_<width>_<height>_<inst_ptr>.<codec>

output_dec_<width>_<height>_<inst_ptr>.yuv

为工作正常, 可能需要给目录/data/msic/media读写权限777.

2. 视频编码Input和Output buffer logs

>adb root

>adb shell setprop vidc.dec.log.out 1 //enable output buffer log

>adb shell setprop vidc.dec.log.in 1 //enable input buffer log

文件缺省地址在/data/misc/media. 文件名如下:

input_dec_<width>_<height>_<inst_ptr>.yuv

output_dec_<width>_<height>_<inst_ptr>.<codec>

为工作正常, 可能需要给目录/data/msic/media读写权限777.

视频Android Framework的Log

根据测试场景的不同, 相应的Framework log开关需要打开. 下面分别说明.

1. 本地视频文件播放

mediaplayer.cpp

AwesomePlayer.cpp

OMXCodec.cpp

MPEG4Extractor.cpp

MediaPlayerService.cpp

AudioPlayer.cpp

2. 录像

StagefrightRecorder.cpp

OmxCodec.cpp

CameraSource.cpp

MPEG4Writer.cpp

视频Android Framework的Log(续)

3. HTTP流媒体

mediaplayer.cpp,

Awesomeplayer.cpp

OMXCodec.cpp

MPEG4Extractor.cpp

NuCachedSource2.cpp

chromium_http/ChromiumHTTPDataSource.cpp

chromium_http/support.cpp

4. RTSP流媒体

mediaplayer.cpp

NuPlayer.cpp

NuPlayerRenderer.cpp

NuPlayerDriver.cpp

ACodec.cpp

MediaCodec.cpp

RTSPSource.cpp

MyHandler.h

ARTPConnection.cpp

ARTSPConnection.cpp

视频Android Framework的Log(续)

5. HTTP Live Streaming(HLS)

NuPlayer.cpp

NuPlayerRenderer.cpp

ACodec.cpp

LiveSession.cpp

HTTPLiveSource.cpp

NuPlayerDriver.cpp

MediaCodec.cpp

打开Framework Log方法:

在要打开Log的文件头部增加

#define LOG_NDEBUG 0

#define LOG_NIDEBUG 0

#define LOG_NDDEBUG 0

Questions?

You may also submit questions to:
https://support.cdmatech.com