Linux GPU
开发指南

版本号: 1.6
发布日期: 2023.11.15

版本历史

版本号	日期	制/修订人	内容描述
1.0	2022.07.14	AWA1831	初始版本
1.1	2022.10.29	AWA1831	添加 mali-utgard 系列 debugfs 功能介绍
1.2	2023.03.07	AWA1831	添加 mali-valhall 系列 GPU 的介绍
1.3	2023.04.20	AWA1831	支持 MR527 系列产品
1.4	2023.05.29	AWA1831	支持 AI985 系列产品
1.5	2023.10.12	AWA1831	更新 vf_table 策略 & 添加 sysfs 调试阶段，用于替代 debugfs
1.6	2023.11.15	AWA1831	支持 T527 系列产品

目　录

1 概述 **1**
- 1.1 编写目的 . . . 1
- 1.2 相关人员 . . . 1
- 1.3 适用范围 . . . 1

2 概念阐释 **2**
- 2.1 GPU 简介 . . . 2
- 2.2 标准接口 . . . 2
- 2.3 窗口系统 . . . 2
- 2.4 软件术语 . . . 2

3 GPU 支持的图像格式 **4**
- 3.1 input . . . 4
- 3.2 ouput . . . 5

4 开发指引 **6**
- 4.1 配置 GPU 驱动 . . . 6
  - 4.1.1 选择 gpu 型号 . . . 6
  - 4.1.2 设备树配置 . . . 7
  - 4.1.3 加载 ko . . . 10
- 4.2 在屏渲染 . . . 11
  - 4.2.1 EGL . . . 11
  - 4.2.2 步骤解析 . . . 12
- 4.3 离屏渲染 . . . 13
  - 4.3.1 前提条件 . . . 14
  - 4.3.2 图像数据排列格式 . . . 14
  - 4.3.3 步骤解析 . . . 17
- 4.4 OpenCL 开发 . . . 29
- 4.5 调试方法 . . . 29
  - 4.5.1 debug 节点（sys/kernel/debug） . . . 29
    - 4.5.1.1 Mali-bifrost 架构 . . . 29
    - 4.5.1.2 Mali-utgard 架构 . . . 30
  - 4.5.2 调频相关 . . . 31
  - 4.5.3 sysfs 节点 . . . 31
    - 4.5.3.1 mali-valhall 架构 . . . 31
  - 4.5.4 调试工具 . . . 33

1 概述

1.1 编写目的

介绍 Sunxi 平台上 GPU 驱动模块的一般使用方法及调试接口，为开发与调试提供参考。

1.2 相关人员

- GPU 驱动开发人员。
- GPU 应用开发（二次开发）和维护人员。

1.3 适用范围

Sunxi 平台上所有的 GPU 模块。

2 概念阐释

2.1 GPU 简介

GPU：Graphics Processing Unit，图形处理单元，主要用于 2D 和 3D 加速。GPU 能够绘制普通 UI 界面、渲染游戏动画，能够做缩放、全景拼接、畸变矫正等，GPU 还能提供并行运算算力。

在有硬件做支撑的前提下，软件的兼容性显得特别重要。目前使用最为广泛的图形 API 是 Khronos 组织提出来的 OpenGL，在移动端产品中对应地叫 OpenGL ES。OpenGL ES 是在 OpenGL 的基础上做一些裁剪，以更好地适应移动端产品对功耗和成本限制的较高要求。除了 OpenGL 之外，Khronos 还定义了 Vulkan 这一套新的图形渲染 API，旨在替代 OpenGL/OpenGL ES。此外，Khronos 组织还定义了一套并行运算 API，即 OpenCL，开发者可以通过 OpenCL 接口来使用 GPU 硬件进行算法加速。

2.2 标准接口

- OpenGL：OpenGL 是一种跨平台的图形 API，用于为 3D 图形处理硬件指定标准的软件接口 (API 驱动 GPU 来绘制 2D 或 3D 图形)。
- OpenGL ES：OpenGL ES 是 3D 图形 API OpenGL 的子集，针对手机、PDA 和游戏主机等嵌入式设备而设计，该 API 由 Khronos 组织定义和推广。
- EGL：embedded Graphic Interface，是 OpenGL ES 和底层 Native 平台视窗系统之间的接口，让 OpenGLES 绘制的内容显示到屏幕上，常见的窗口系统有 Android、fbdev、X11、Wayland 等。
- OpenCL：Open Computing Language 是一个为异构平台编写程序的框架，此异构平台可由 CPU、GPU、DSP、FPGA 或其他类型的处理器与硬件加速器所组成，运算任务通过并行的方式提升运算效率。

2.3 窗口系统

在 2.2 小节中，提到 EGL 接口主要的作用是实现 OpenGL ES 和 Native 平台窗口系统之间的联系。常见的窗口系统一共有 fbdev(Linux 系统)、Android、wayland 几种。

2.4 软件术语

表 2-1: 模块软硬件相关术语介绍

术语	解释说明
DVFS	一种动态电源管理技术，根据设备的工作负载，动态调节工作频率及电压
GPU idle	根据 GPU 工作状态，是否自动开关电源及时钟

说明

GPU idle 只有 GPU 模块为单独供电时才起作用，GPU 是否为单独供电，可以通过 cat /sys/kernel/debug/sunxi_dump/dump 节点查看。

3 GPU 支持的图像格式

3.1 input

表 3-1: 支持 YUV 的输入格式

Format（格式）	Planes（平面数）
8-bit YUV 4:4:4	1
8-bit YUVA 4:4:4	1
8-bit YUYV 4:2:2	1
8-bit VYUY 4:2:2	1
8-bit Y-UV 4:2:2	2
8-bit Y-UV 4:2:2, 16x16 block-linear	2
8-bit Y-U-V 4:2:2	3
8-bit Y-UV 4:2:0	2
8-bit Y-UV 4:2:0, 16x16 block-linear	2
8-bit Y-U-V 4:2:0	3
10-bit YUVA 4:4:4	1
10-bit AYUV 4:4:4	1
10-bit YUVA 4:2:0	1
10-bit YUYV 4:2:2	1
10-bit VYUY 4:2:2	1
10-bit Y-UV 4:2:2	2
10-bit Y-UV 4:2:0	2
16-bit YUV 4:4:4	1
16-bit YUVA 4:4:4	1
16-bit YUYV 4:2:2	1
16-bit VYUY 4:2:2	1
10-bit Y-UV 4:2:2, 16-bit container	2
16-bit Y-UV 4:2:2	2
10-bit Y-UV 4:2:0, 16-bit container	2
16-bit Y-UV 4:2:0	2
8-bit AFBC Y-UV 4:2:2	1
10-bit AFBC Y-UV 4:2:2	1
8-bit AFBC Y-UV 4:2:0	1
10-bit AFBC Y-UV 4:2:0	1

3.2 ouput

表 3-2: 支持 YUV 的输出格式

Format（格式）	Planes（平面数）
8-bit YUV 4:4:4	1
8-bit YUVA 4:4:4	1
8-bit YUYV 4:2:2	1
8-bit VYUY 4:2:2	1
8-bit Y-UV 4:2:2	2
8-bit Y-U-V 4:2:2	3
8-bit Y-UV 4:2:0	2
8-bit Y-U-V 4:2:0	3
10-bit YUVA 4:4:4	1
10-bit AYUV 4:4:4	1
10-bit YUVA 4:2:0	1
10-bit YUV 4:4:4, 16-bit container	1
10-bit YUVA 4:4:4, 16-bit container	1
10-bit YUYV 4:2:2, 16-bit container	1
10-bit VYUY 4:2:2, 16-bit container	1
10-bit Y-UV 4:2:2, 16-bit container	2
10-bit Y-UV 4:2:0, 16-bit container	2
8-bit AFBC Y-UV 4:2:0	1
10-bit AFBC Y-UV 4:2:0	1

说明

planes 指出了图像数据在内存中由几个 plane 组成。在 GPU 处理图像数据时，需要知道图像数据在内存中的排列方式，这样 GPU 才能正确地处理数据。在 4.3.2 章节介绍如何创建 pbuffer（离屏渲染）时，会详细介绍 plane 存储方式。
目前，全志平台中，只有型号为 mali-g31、mali-g57 的 GPU 才支持输入输出 AFBC 格式数据。

4 开发指引

GPU 模块涉及的几个常见应用场景如下：

- 配置 GPU 驱动
- 基于 OpenGL ES，在屏渲染
- 基于 OpenGL ES，离屏渲染
- OpenCL 开发
- 常见的调试方法

4.1 配置 GPU 驱动

4.1.1 选择 gpu 型号

在内核目录 (例如 T507 环境下，路径为根目录) 下执行./build.sh menuconfig，查找 gpu_type 项，配置为 mali-g31。

图 4-1: menuconfig

具体的 GPU 型号需要查看芯片规格获取，以下表格列出 Sunxi 不同平台中，GPU 的型号以及 gpu_type 的值。

表 4-1: gpu_type 表

平台	GPU 型号	GPU 架构	gpu_type
T3/T3-C/T3PRO	Mali400	mali-utgard	mali400
A40I-H/A40I-C	Mali400	mali-utgard	mali400
T507/T507-H	Mali-g31	mali-bifrost	mali-g31
T517/T517-H	Mali-g31	mali-bifrost	mali-g31
A523	Mali-G57	mali-valhall	mali-g57
MR527	Mali-G57	mali-valhall	mali-g57-r32p1
AI985	Mali-G57	mali-valhall	mali-g57
T527	Mali-G57	mali-valhall	mali-g57

4.1.2 设备树配置

Device Tree 主要用来配置模块相关的参数，例如中断、寄存器、时钟信息、vf 表等。

Device Tree 文件的路径为：

- bsp/configs/linux-5.10/boot/dts/{CHIP}.dtsi(CHIP 为研发代号，如 T507 的研发代号为 sun50iw9p1 等)；
- device/config/chips/{IC}/configs/{BOARD}/board.dts(IC 为产品型号，如 T507，BOARD 有 demo.2、demo.3 等)；

其中，前者用于配置中断、寄存器、时钟信息等固定参数，后者则用于配置 vf 表 (如果 dtsi 中 GPU 实现了 operating-points-v2，则会优先使用 operating-points-v2)、各种功能使能开关等可变参数。

表 4-2: board.dts 配置说明

参数名	说明
independent_power	GPU 是否独立供电，0 表示非独立供电，1 表示独立供电
gpu_idle	是否打开 GPU idle，0 表示否，1 表示是
dvfs_status	是否打开 DVFS，0 表示关闭，1 表示打开
operating-points	GPU 的 vf 表
gpu-supply	GPU 独立供电时使用的 regulator 的 id

```
gpu: gpu@0x01800000 {
    device_type = "gpu";
    compatible = "arm,mali-midgard";
    reg = <0x0 0x01800000 0x0 0x10000>;//寄存器地址
    interrupts = <GIC_SPI 117 IRQ_TYPE_LEVEL_HIGH>,//中断
```

```
        <GIC_SPI 118 IRQ_TYPE_LEVEL_HIGH>,
        <GIC_SPI 76 IRQ_TYPE_LEVEL_HIGH>;
    interrupt-names = "JOB", "MMU", "GPU";
    clocks = <&ccu CLK_PLL_GPU>, <&ccu CLK_GPU>, <&ccu CLK_BUS_GPU>;//时钟
    clock-names = "clk_parent", "clk_mali", "clk_bus";
    resets = <&ccu RST_BUS_GPU>;
    #cooling-cells = <2>;
    power-domains = <&pd TV303_PD_GPU>;
    ipa_dvfs:ipa_dvfs {//温控ipa相关参数
      compatible = "arm,mali-simple-power-model";
      static-coefficient = <5800>;
      dynamic-coefficient = <860>;
      ts = <1000000 0xffffc310 0xffffffb0 9>;
      thermal-zone = "gpu_thermal_zone";
      /*ss-coefficient = <36>;*/
      /*ff-coefficient = <291>;*/
    };
  };
```

```
  gpu: gpu@0x01800000 {
      gpu_idle = <1>;
      dvfs_status = <1>;
      operating-points = <
        s /* KHz   uV */
        600000 950000
        300000 950000
        200000 950000
        150000 950000
       >;
  };
```

- 使用 operating-points-v2

目前，全志平台中，只有 mali-valhall 系列的 GPU 使用operating-points-v2, 则产品方案 board.dts 中的operating-points将不再起作用。以下是 mali-valhall 平台中，operating-points-v2的实现。

```
    gpu:gpu@1800000 {
      device_type = "gpu";
      compatible = "arm,mali-valhall";
      reg = <0x0 0x01800000 0x0 0x10000>;
      interrupts = <GIC_SPI 117 IRQ_TYPE_LEVEL_HIGH>,
            <GIC_SPI 118 IRQ_TYPE_LEVEL_HIGH>,
            <GIC_SPI 119 IRQ_TYPE_LEVEL_HIGH>;
      interrupt-names = "JOB", "MMU", "GPU";
      clocks = <&ccu CLK_PLL_GPU>, <&ccu CLK_GPU>, <&ccu CLK_BUS_GPU>;
      clock-names = "clk_parent", "clk_mali", "clk_bus";
      resets = <&ccu RST_BUS_GPU>;
      operating-points-v2 = <&gpu_opp_table>; //不同点在这里，实现了opp—v2，驱动中使用opp框架初始化opp_table
     时，会优先使用opp-v2
      #cooling-cells = <2>;
      ipa_dvfs:ipa_dvfs {
        compatible = "arm,mali-simple-power-model";
        static-coefficient = <636>;
        dynamic-coefficient = <1434>;
        /* ts0 -> ts3 */
        ts = <0xcc77c0 217000 0xffffd508 200>;
        thermal-zone = "gpu_thermal_zone";
        /*ss-coefficient = <36>;*/
        /*ff-coefficient = <291>;*/
```

```
    };
     /*power-domains = <&pd1 A523_PCK_GPU>;*/
  };

  gpu_opp_table: gpu-opp-table {
    compatible = "allwinner, mali-valhall-operating-points";
    opp@150000000 {
      opp-hz = /bits/ 64 <150000000>;
      opp-microvolt = <900000>;
    };

    opp@200000000 {
      opp-hz = /bits/ 64 <200000000>;
      opp-microvolt = <900000>;
    };

    opp@300000000 {
      opp-hz = /bits/ 64 <300000000>;
      opp-microvolt = <900000>;
    };

    opp@400000000 {
      opp-hz = /bits/ 64 <400000000>;
      opp-microvolt = <900000>;
    };

    opp@600000000 {
      opp-hz = /bits/ 64 <600000000>;
      opp-microvolt = <900000>;
    };

    opp@648000000 {
      opp-hz = /bits/ 64 <648000000>;
      opp-microvolt = <0>;
      opp-microvolt-vf0900 = <900000>;
    };

    opp@696000000 {
      opp-hz = /bits/ 64 <696000000>;
      opp-microvolt = <0>;
      opp-microvolt-vf1920 = <900000>;
      opp-microvolt-vf2920 = <900000>;
      opp-microvolt-vf3920 = <900000>;
      opp-microvolt-vf21920 = <900000>;
      opp-microvolt-vf31920 = <900000>;
      opp-microvolt-vf5920 = <900000>;
    };

    opp@744000000 {
      opp-hz = /bits/ 64 <744000000>;
      opp-microvolt = <0>;
      opp-microvolt-vf4920 = <900000>;
      /* Not use: only for performance test
        opp-microvolt-vf2920 = <900000>;
        opp-microvolt-vf3920 = <900000>;
        opp-microvolt-vf21920 = <900000>;
        opp-microvolt-vf31920 = <900000>;
      */
    };
```

```
        /* Not use: only for performance test */
        opp@792000000 {
            opp-hz = /bits/ 64 <792000000>;
            opp-microvolt = <0>;
            /* opp-microvolt-vf2950 = <900000>;
              opp-microvolt-vf3950 = <900000>;
              opp-microvolt-vf21950 = <900000>;
              opp-microvolt-vf31950 = <900000>;
            */
        };
    };
```

说明

Sunxi 不同的平台使用的 GPU 硬件型号一般是有差异的，型号不同，GPU 使用的驱动也是不同的，但同一型号的 GPU 的配置方法一般都是相同的（例如：寄存器地址、中断、时钟、温控参数等）；部分配置项是所有 GPU 都支持的（如：是否独立供电、dvfs 是否开启、opp 表等），只是这些配置项的内容会有所差异。
对于部分老旧平台（GPU 型号是 Mali-400/Mali-450），部分配置信息在 sys_config.fex 中配置，sys_config.fex 文件的位置一般在 device/config/chips/{IC}/configs/{BOARD}/sys_config.fex(IC 为产品型号，如 T509，BOARD 为板型如 perf1)。

4.1.3 加载 ko

内核编译完成后，会生成 mali_kbase.ko、dma-buf-test-exporter.ko 模块，bsp 固件会打包放在lib/modules/${kernel_version}路径下，我们需要手动加载。

```
#加载GPU模块、dma模块
#以下是内核打印，可以用来确认模块是否成功加载(使用命令'dmesg -c' 查看)
/lib/modules/5.10.80+ # insmod mali_kbase.ko
[  70.808909] mali_kbase: loading out-of-tree module taints kernel.
[  70.825123] mali 1800000.gpu: Regulators probed: 0
[  70.830521] mali 1800000.gpu: Clocks probed: 2
[  70.836386] mali 1800000.gpu: [950mv-600MHz] inde_power:0 idle:1 dvfs:1
[  70.844231] mali 1800000.gpu: GPU identified as 0x3 arch 7.0.9 r0p0 status 0
[  70.852346] mali 1800000.gpu: No memory group manager is configured
[  70.861659] mali 1800000.gpu: Using configured power model mali-simple-power-model, and fallback mali-simple-
    power-model


/lib/modules/5.10.80+ # insmod dma-buf-test-exporter.ko
[  97.565562] misc dma_buf_te: dma_buf_te ready


#模块加载成功的同时，会在dev/目录下分别生成mali0、dma_buf_te两个节点，这也是判断模块是否加载的有效方法。
```

说明

Sunxi 不同的平台，GPU 模块的型号可能不同，所编译出来的 ko 模块的命名也会有所差异 (mali_kbase.ko 或 mali.ko)，此处只是提醒开发人员需要注意模块的加载，保证正确的开发环境。此外，Android 系统不需要手动加载 ko 模块。

4.2 在屏渲染

4.2.1 EGL

前面章节已提到，EGL 是 OpenGL ES 渲染 API 和本地窗口系统 (native platform window system) 之间的一个中间接口层，它主要由系统制造商实现。EGL 提供主要功能如下。

1. 与设备的原生窗口系统通信

   - eglGetDisplay – 创建图形 API 与本地窗口系统的连接。
   - eglInitialize – 初始化 EGL 连接。
   - eglTerminate – 终止 EGL。

2. 查询、配置 framebuffer 的属性

   - eglGetConfigs – 获取底层窗口系统支持的所有 EGl framebuffer（在屏渲染就是 fb0）配置。
   - eglChooseConfig – 获取与需求列表匹配的 framebuffer 配置列表，用于配置与 display 绑定的 framebuffer。
   - eglGetConfigAttrib – 获取 EGL framebuffer 指定的配置信息。

3. 创建、销毁 surface

   - eglCreateWindowSurface – 创建一个在屏渲染的 surface（window surface）。
   - eglCreatePbufferSurface – 创建一个离屏渲染的 surface（pbuffer surface）。
   - eglQuerySurface – 查询 EGL surface 的信息。
   - eglDestroySurface – 销毁 EGL surface。

4. 管理工作上下文

   - eglCreateContext – 创建一个新的 EGL rendering context。
   - eglDestroyContext – 销毁一个 EGL rendering context。
   - eglMakeCurrent – 将 context 附着到 surface 上。

5. 管理纹理贴图等渲染资源

   - glGenTextures – 生成纹理（图片）的名字。
   - glActiveTexture – 激活纹理单元。
   - glBindTexture – 将纹理对象与纹理名绑定在一起。
   - glDeleteTextures – 删除纹理对象。
   - glTexImage2D – 加载纹理单元，生成 2D 图像。
   - eglCreateImageKHR – 创建外部纹理（图片），这是 EGL 的一种扩展，这个扩展接口实现了 CPU 与 GPU 共享纹理内存。
   - eglDestroyImageKHR – 销毁外部纹理。

6. 送显相关

- eglSwapInterval – 指定每个缓冲区交换的最小视频帧周期数。
- eglSwapBuffers – 将 EGL surface 送显到本地窗口。
- glFinish – 将所有绘制指令发送给 GPU，直到 GPU 硬件执行完毕后返回。
- glFlush – 将绘制指令发送给 GPU，直接返回。

⚠ 注意

如果想要使用 KHR 结尾的 egl 函数，需要添加相关的扩展函数。

4.2.2 步骤解析

3.2.1 中，已经介绍了 EGL 常见的使用方法，以下是 OpenGL ES 实现在屏渲染（window surface）的 demo。

```
#include <stdio.h>
#include <EGL/egl.h>
#include <GLES2/gl2.h>
#include <GLES2/gl2ext.h>

int main(int argc, char **argv)
{
    EGLBoolean egl_ret;
    EGLContext context;
    EGLSurface window_surface;
    EGLConfig config;
    EGLint num_config;
    //（1）获取本地默认的display
    EGLDisplay dpy = eglGetDisplay(EGL_DEFAULT_DISPLAY);
    EGLint context_attribs[] = {
        EGL_CONTEXT_CLIENT_VERSION, 2,
        EGL_NONE
    };

    EGLint attrib_list[] = {
        EGL_RED_SIZE, 8,
        EGL_GREEN_SIZE, 8,
        EGL_BLUE_SIZE, 8,
        EGL_ALPHA_SIZE, 8,
        EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
        EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
        EGL_NONE
    };

    if (dpy == EGL_NO_DISPLAY) {
        printf("eglGetDisplay returned EGL_NO_DISPLAY\n");
        return -1;
    }
    //（2）初始化获取到的display，建立API与本地窗口的连接
    egl_ret = eglInitialize(dpy, NULL, NULL);
    if (egl_ret != EGL_TRUE) {
        printf("eglInitialize failed\n");
        return -1;
```

```
    }
    //（3）按照attrib_list 来选择display相应的配置，生成的配置表写入到 config中（第三个参数）
    egl_ret = eglChooseConfig(dpy, attrib_list, &config, 1, &num_config);
    if (egl_ret != EGL_TRUE) {
        printf("eglChooseConfig failed\n");
        return 0;
    }
    //（4）创建window surface
    window_surface = eglCreateWindowSurface(dpy, config, (NativeWindowType)NULL, NULL);
    if (window_surface == EGL_NO_SURFACE) {
        printf("gelCreateWindowSurface failed\n");
        return 0;
    }
    //（5）创建工作上下文
    context = eglCreateContext(dpy, config, EGL_NO_CONTEXT, context_attribs);
    if (context == EGL_NO_CONTEXT) {
        printf("eglCreateContext failed\n");
        return 0;
    }
    //（6）指定当前工作的上下文
    egl_ret = eglMakeCurrent(dpy, window_surface, window_surface, context);
    if (egl_ret != EGL_TRUE) {
        printf("eglMakeCurrent failed\n");
        return 0;
    }

    glClearColor(1.0f, 0.0f, 0.0f, 1.0f);

    glClear(GL_COLOR_BUFFER_BIT);
    //（7）送显
    eglSwapBuffers(dpy, window_surface);
    //（8）销毁surface
    eglDestroySurface(dpy, window_surface);
    //（9）销毁工作上下文
    eglDestroyContext(dpy, context);
    //（10）销毁display连接
    eglTerminate(dpy);
}
```

说明

本文中，只介绍 EGL 最基本的使用流程，更多细节请参考官方连接：https://www.khronos.org/egl。
OpenGL ES 相关语法请参考官方连接：https://www.khronos.org/opengles。

4.3 离屏渲染

这里所说的离屏渲染，指的是 pbuffer surface。使用 pbuffer surface，一方面实现了 GPU 与 CPU 共享纹理（surface/图片）内存，提高渲染效率，节省 CPU 与 GPU 之间纹理拷贝的时间和带宽；另一方面，GPU 硬件可以直接处理 yuv 格式数据（OpenGL 2.0 及以后版本，支持输入 yuv 数据，但 OpenGL 3.0 及之后才支持直接输出 yuv 数据）。

创建 pbuffer surface 所需的物理内存，由用户自己创建与释放。我们可以借助 dma-buf、ion 来管理申请的物理内存，将申请好的物理内存交给 GPU 处理。

4.3.1 前提条件

- 内核驱动支持 ion、dma-buf（Sunxi 平台，linux4.9 & linux5.4 是支持的）。
- EGL 支持创建外部纹理对象的扩展函数（EGLKHR）。
- 以 GPU 型号为 G31 为例，支持 EGL1.4、OpenGL ES 1.1、OpenGL ES 2.0、OpenGL ES 3.2、Vulkan 1.0。如果想要渲染输出 yuv 格式数据，需要 OpenGL ES3.0 及以上版本，且添加 GL_EXT_YUV_target 扩展。

4.3.2 图像数据排列格式

在介绍 GPU 支持的 YUV 格式数据时，有提到过 plane 这个名词。plane 的值表征了图像数据在内存中的排布方式，在创建外部纹理时，这个信息是至关重要的。在这里，以 HAL_PIXEL_FORMAT_AW_NV12(8-bit Y-UV 4:2:0) 数据为例，介绍图像数据在内存中的存储排列方式。

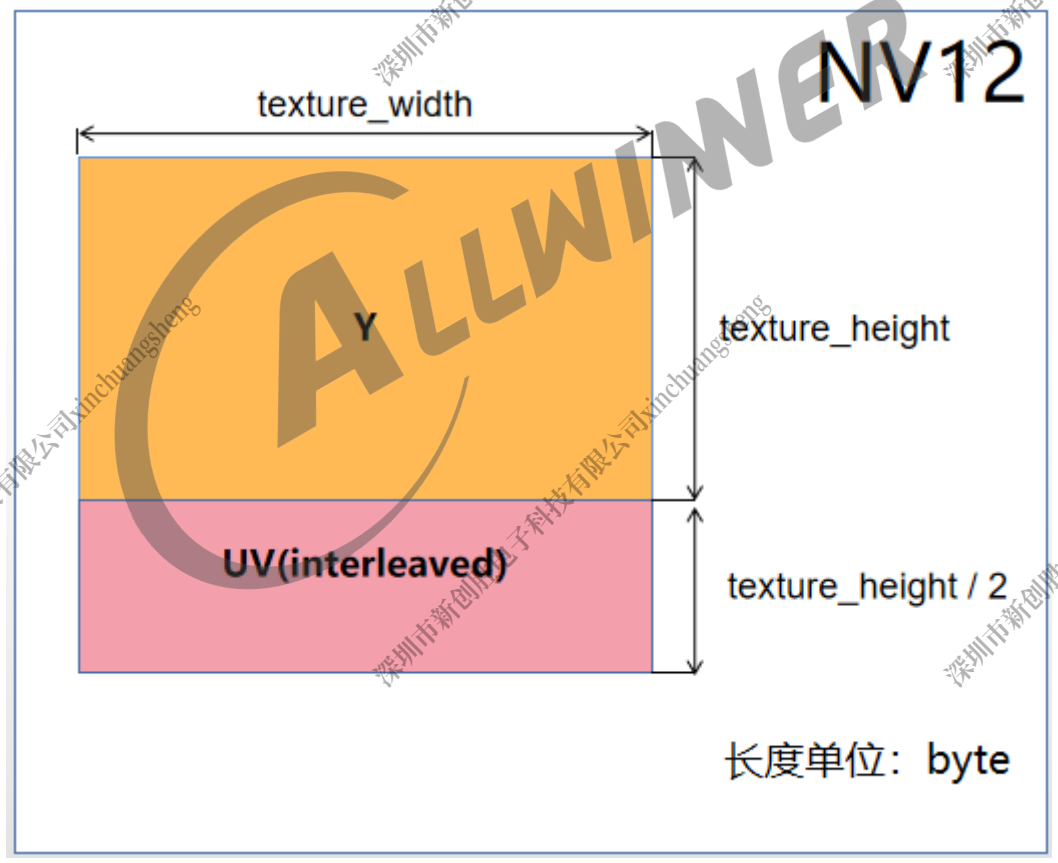

图 4-2: NV12_ 数据排列方式

由上图可知，图像数据分为了 2 个 plane(黄色表示第一个 plane，存放图像的所有 Y 分量，粉红色表示第二个 plane，存放图像所有的 UV 分量)。第一个 plane 存放所有像素的 Y 值，第二个 plane 存放 UV 值（UV 交替存放，如果数据格式是 NV21，则第二个 plane 中，UV 分量的排列顺序为 VU

交替存放）。GPU 在处理图像数据时，需要得到每个像素的 YUV 值，所以在创建外部纹理时，我们需要把纹理在内存中的起始地址、几个 plane、每个 plane 的偏移量等信息都传递给 GPU。

再举一个分为 3 个 plane 存储的例子，图像数据格式为 YU16(8-bit Y-UV 4:2:2，4:2:2 采样，分为 3 个 plane 存储，Y、U、V 分量各一 plane)。

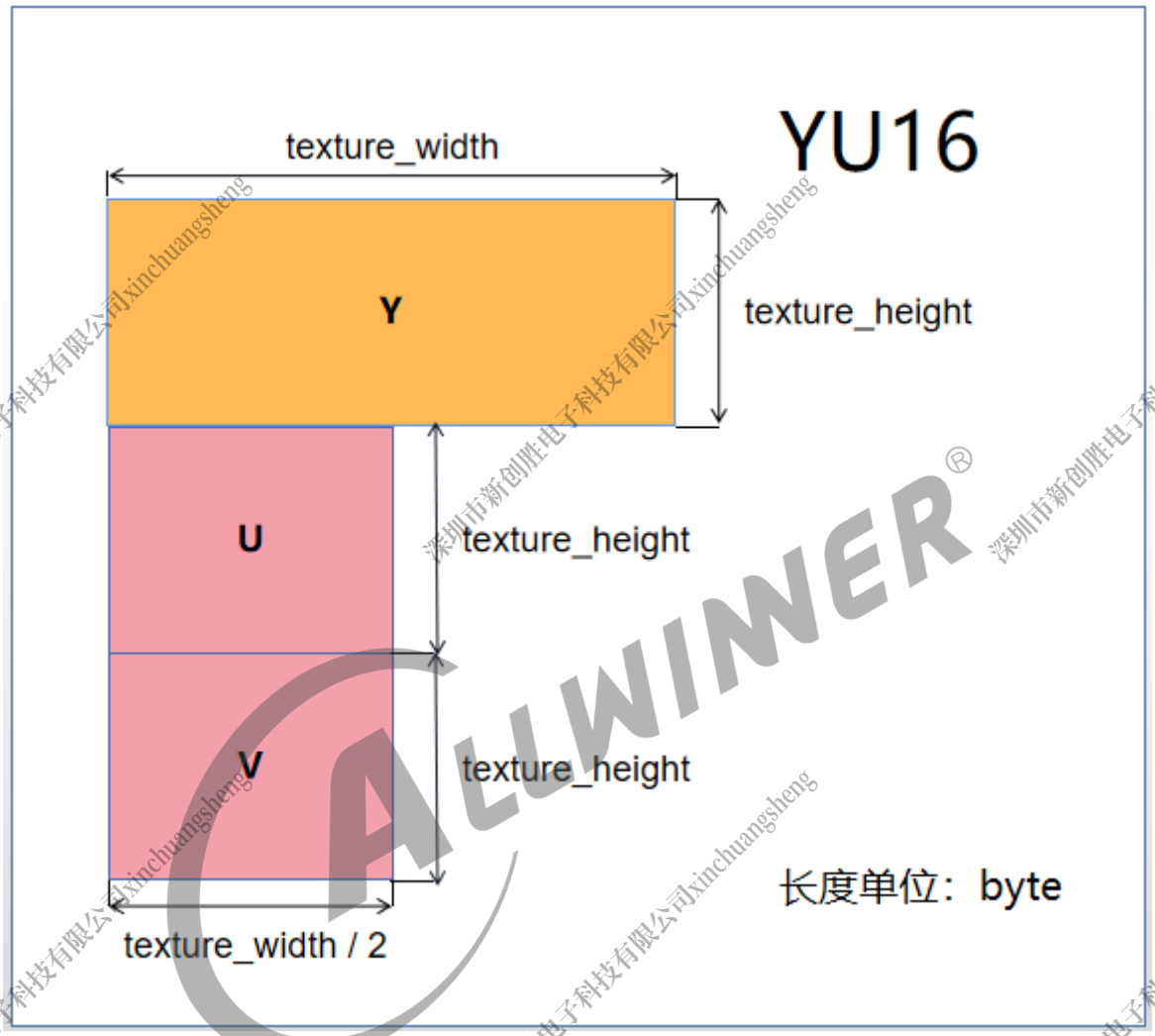

图 4-3: YU16_ 数据排列方式

综上所述，无论是哪一种图像格式，只需要关注两点，即采样方式和数据分为几个 plane 存储。知道这两点信息后，我们就知道图像数据的各分量在内存中的存储方式了，就能计算好各 plane 相对于起始地址的偏移量，进而正确的编写渲染程序。

```
static int setupTexture(EGLDisplay dpy, int dmafd, EGLImageKHR *imgTex,
      unsigned int width, unsigned int height)
{
   int dma_fd;
   int atti = 0;
   EGLint attribs0[30];
   EGLImageKHR img0;
   GLuint Tex;
   //已经分配好的dma_fd,由之前的dma_alloc得到
   dma_fd = dmafd;
```

```
glActiveTexture(GL_TEXTURE0);
//设置宽高尺寸，纹理/图片的宽高
attribs0[atti++] = EGL_WIDTH;
attribs0[atti++] = width;
attribs0[atti++] = EGL_HEIGHT;
attribs0[atti++] = height;

//设置像素格式
attribs0[atti++] = EGL_LINUX_DRM_FOURCC_EXT;
attribs0[atti++] = DRM_FORMAT_NV12;

//设置Y分量的buffer的fd、offset、pitch
attribs0[atti++] = EGL_DMA_BUF_PLANE0_FD_EXT;
attribs0[atti++] = dma_fd;
//y分量地址的偏移量
attribs0[atti++] = EGL_DMA_BUF_PLANE0_OFFSET_EXT;
attribs0[atti++] = 0;
//y分量的长度，一个像素点包含一个y分量，plan0的宽度是每行像素个数，也就是每行y分量的个数
attribs0[atti++] = EGL_DMA_BUF_PLANE0_PITCH_EXT;
attribs0[atti++] = width;

//设置uv分量的buffer的fd、offset、pitch
attribs0[atti++] = EGL_DMA_BUF_PLANE1_FD_EXT;
attribs0[atti++] = dma_fd;
//uv分量地址的偏移量，因为uv在y分量之后存储，y分量的个数是像素的个数，一个y分量的大小正好是一个字节
//如果图像数据分为3个plane存储，就需要配置3个plane的信息, DRM_FORMAT_NV12分为两个plane存储，只需要配置两
    个plane
attribs0[atti++] = EGL_DMA_BUF_PLANE1_OFFSET_EXT;
attribs0[atti++] = width * height;
attribs0[atti++] = EGL_DMA_BUF_PLANE1_PITCH_EXT;
attribs0[atti++] = width;

//设置color space和color range
attribs0[atti++] = EGL_YUV_COLOR_SPACE_HINT_EXT;
attribs0[atti++] = EGL_ITU_REC709_EXT;
attribs0[atti++] = EGL_SAMPLE_RANGE_HINT_EXT;
attribs0[atti++] = EGL_YUV_FULL_RANGE_EXT;

attribs0[atti++] = EGL_NONE;

//共享内存的target设置为: EGL_LINUX_DMA_BUF_EXT；GPU会根据以上配置信息，创建外部image
img0 = eglCreateImageKHR(dpy, EGL_NO_CONTEXT, EGL_LINUX_DMA_BUF_EXT, 0,
        attribs0);
if (img0 == EGL_NO_IMAGE_KHR) {
  checkEglErrorState();
  return -1;
}

//创建纹理对象
glGenTextures(1, &Tex);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_EXTERNAL_OES, Tex);

glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
```

```
    glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_WRAP_T,
        GL_CLAMP_TO_EDGE);
    //将纹理与Image相关联，此时纹理对象和外部buffer关联在一起
    glEGLImageTargetTexture2DOES(GL_TEXTURE_EXTERNAL_OES,
            (GLeglImageOES)img0);
    if (checkEglErrorState())
      return -1;
    if (checkGlErrorState())
      return -1;

    *imgTex = img0;
    return Tex;
}
```

4.3.3 步骤解析

```
#include <stdlib.h>
#include <stdbool.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <time.h>
#include <sys/fcntl.h> // include O_RDWR
#include <sys/ioctl.h> // include ioctl()
#include <sys/mman.h> // include mmap()
#include <string.h> // include memcpy()
#include <linux/fb.h>

#include <EGL/egl.h>
#include <GLES2/gl2.h>
#include <EGL/eglext.h>
#include <GLES2/gl2ext.h>
#include <EGL/mali_fbdev_types.h>

#include "ion.h"
#include "drm_fourcc.h" // include DRM_FORMAT_NV21
#include "sunxi_display2.h" //include dev/disp

#define WIDTH 1920
#define HEIGHT 1080

#define TEXTURE_WIDTH 1920
#define TEXTURE_HEIGHT 1080

struct Shader {
  GLuint id;
};
//申请dma_buf相关的结构体
struct ion_mem {
  unsigned int phy;  /* physical address */
  unsigned char *virt;   /* visual address */
  unsigned long size; /* memory size */
  int dmafd;    /* ion's dmabuf fd */
  int handle;   /* ion's handle */
};

struct Renderer {
```

```
    //EGL References
    EGLint egl_major, egl_minor;
    EGLDisplay egl_display;
    EGLContext egl_context;
    EGLSurface egl_surface;
    EGLImageKHR imgTex;
    EGLImageKHR imgFboTex;

    //Shader references
    struct Shader vs;
    struct Shader fs;
    GLuint shader_program;

    //ion dma references
    struct ion_mem TexMem;
    struct ion_mem FboMem;

    /*Texure references*/
    GLuint Tex;
    GLuint FboColorBufferTex;

    GLuint fbo;

};

struct TestCase {
    struct Renderer renderer;
    unsigned int screen_width, screen_height;
    unsigned int tex_width, tex_height;
    int ion_fd;
};

struct fbdev_window native_window = {
    .width = WIDTH,
    .height = HEIGHT,
};

struct fbdev_window native_window;

/*PFNEGLCREATESYNCKHRPROC eglCreateSyncKHR;
PFNEGLDESTROYSYNCKHRPROC eglDestroySyncKHR;
PFNEGLCLIENTWAITSYNCKHRPROC eglClientWaitSyncKHR;
PFNEGLCREATEIMAGEKHRPROC eglCreateImageKHR;
PFNGLEGLIMAGETARGETTEXTURE2DOESPROC glEGLImageTargetTexture2DOES;
*/

static const char *vertex_shader_source =
    "#version 300 es                          \n"
    "in vec4 aPosition;                       \n"
    "in vec2 aTexCoord;                       \n"
    "uniform float aScale;                    \n"
    "out vec2 vTexCoord;                      \n"
    "void main()                              \n"
    "{                                        \n"
    "   mat4 sTemp_mat4 = mat4(aScale);       \n"
    "   vTexCoord = vec2(aTexCoord.x, 1.0 - aTexCoord.y);   \n"
    "   vec4 temp = sTemp_mat4 * aPosition;   \n"
    "   gl_Position = vec4(temp.xyz, 1.0);    \n"
    "}                                        \n";
static const char *fragment_shader_source =
```

```
    "#version 300 es                                \n"
    "#extension GL_OES_EGL_image_external_essl3 : enable    \n"
    "#extension GL_EXT_YUV_target : enable              \n"
    "precision mediump float;                       \n"
    "uniform __samplerExternal2DY2YEXT uTexSampler;       \n"
    "in vec2 vTexCoord;                             \n"
    "layout (yuv) out vec4 color;                   \n"
    "void main()                                    \n"
    "{                                              \n"
    "   color = texture(uTexSampler, vTexCoord);          \n"
    "}                                              \n";

static GLfloat vVertices[] = {
    -1.0f, -1.0f, 0.0f,
    1.0f,  -1.0f, 0.0f,
    1.0f,  1.0f,  0.0f,
    -1.0f, 1.0f,  0.0f,
};

static GLfloat vTexVertices[] = {
    0.0f, 1.0f,
    1.0f, 1.0f,
    1.0f, 0.0f,
    0.0f, 0.0f,
};

void Redraw(struct TestCase *test)
{
    glViewport(0, 0, test->screen_width, test->screen_height);
    eglSwapInterval(test->renderer.egl_display, 1);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

    //Render in blocking mode
    glFinish();
}

int ion_open(void)
{
    int ion_fd;

    ion_fd = open("/dev/ion", O_RDWR, 0);
    if (ion_fd <= 0) {
        printf("open ion failed!\n");
        return -1;
    }

    return ion_fd;
}

void ion_close(int ion_fd)
{
    if (ion_fd > 0)
        close(ion_fd);
}
//申请dma_buf,将物理地址映射为虚拟地址
static int dma_alloc(int ion_fd, struct ion_mem *mem, int size)
{
    int ret;
    void *addrmap = NULL;
```

```
    //alloc dma buffer,and reture an ion handle
    struct ion_allocation_data sAllocInfo = {
        .len = size,
        .heap_id_mask = ION_HEAP_SYSTEM,
        .flags = ION_FLAG_CACHED,
    };

    ret = ioctl(ion_fd, ION_IOC_ALLOC, &sAllocInfo);
    if (ret < 0) {
        printf("alloc ion err.\n");
        close(ion_fd);
        return -1;
    }

    if (sAllocInfo.fd < 0) {
        printf("ION_IOC_MAP failed!\n");
        close(ion_fd);
        return -1;
    }

    /* mmap to user */
    addrmap = mmap(NULL, sAllocInfo.len, PROT_READ | PROT_WRITE, MAP_SHARED,
            sAllocInfo.fd, 0);
    if (MAP_FAILED == addrmap) {
        printf("mmap err!\n");
        addrmap = NULL;
        return -1;
    }

    memset(addrmap, 0, sAllocInfo.len);
    mem->phy = 0;
    mem->virt = (unsigned char *)addrmap;
    mem->size = size;
    mem->dmafd = sAllocInfo.fd;
    mem->handle = 0;

    return mem->dmafd;
}

void ion_free(int ion_fd, struct ion_mem *mem)
{

    if (mem->virt) {
        munmap((void *)mem->virt, mem->size);
        mem->virt = 0;
    }

    if (mem->dmafd > 0) {
        close(mem->dmafd);
        mem->dmafd = -1;
    }

}

int load_texture(int ion_fd, struct ion_mem *mem, const char *path, int size)
{
    void *addr_file;
    int dma_fd;
    int file_fd;
```

```
    dma_fd = dma_alloc(ion_fd, mem, size);
    if (dma_fd < 0) {
        printf("init_dma_buf for texture failed!\n");
        return -1;
    }

    //load textue
    file_fd = open(path, O_RDWR);
    if (file_fd < 0) {
        printf("open %s err.\n", path);
        return -1;
    }
    addr_file = (void *)mmap((void *)0, size, PROT_READ | PROT_WRITE,
            MAP_SHARED, file_fd, 0);

    //copy texture into dma buffer
    memcpy(mem->virt, addr_file, size);

    munmap(addr_file, size);
    close(file_fd);
    //printf("succeed to load texuture:%s\n", path);

    return dma_fd;
}
//创建输出的surface，使用共享内存，将GPU的渲染结果直接写入GPU外部内存中。实现方式和setupTexture类似
int createFBO(EGLDisplay dpy, int ion_fd, struct ion_mem *mem, GLuint *Fbo,
        GLuint *tex, EGLImageKHR *imgFbo, unsigned int width, unsigned int height)
{
    int dma_fd;
    int atti = 0;
    EGLint attribs0[30];
    EGLImageKHR img0;
    GLuint Tex;

    dma_fd = dma_alloc(ion_fd, mem, width * height * 3 / 2);
    if (dma_fd < 0) {
        printf("init dma buf of nv12 for fbo failed!\n");
        return -1;
    }

    //设置宽高尺寸
    attribs0[atti++] = EGL_WIDTH;
    attribs0[atti++] = width;
    attribs0[atti++] = EGL_HEIGHT;
    attribs0[atti++] = height;

    //设置像素格式
    attribs0[atti++] = EGL_LINUX_DRM_FOURCC_EXT;
    attribs0[atti++] = DRM_FORMAT_NV21;
    //attribs0[atti++] = DRM_FORMAT_ARGB8888;

    //设置Y分量的buffer的fd、offset、pitch
    attribs0[atti++] = EGL_DMA_BUF_PLANE0_FD_EXT;
    attribs0[atti++] = dma_fd;
    attribs0[atti++] = EGL_DMA_BUF_PLANE0_OFFSET_EXT;
    attribs0[atti++] = 0;
    attribs0[atti++] = EGL_DMA_BUF_PLANE0_PITCH_EXT;
    attribs0[atti++] = width;
    //attribs0[atti++] = width * 4;
```

```
//设置uv分量的buffer的fd、offset、pitch
attribs0[atti++] = EGL_DMA_BUF_PLANE1_FD_EXT;
attribs0[atti++] = dma_fd;
attribs0[atti++] = EGL_DMA_BUF_PLANE1_OFFSET_EXT;
attribs0[atti++] = width * height;
attribs0[atti++] = EGL_DMA_BUF_PLANE1_PITCH_EXT;
attribs0[atti++] = width;

//设置color space和color range
attribs0[atti++] = EGL_YUV_COLOR_SPACE_HINT_EXT;
attribs0[atti++] = EGL_ITU_REC709_EXT;
attribs0[atti++] = EGL_SAMPLE_RANGE_HINT_EXT;
attribs0[atti++] = EGL_YUV_FULL_RANGE_EXT;

attribs0[atti++] = EGL_NONE;

//共享内存的target设置为: EGL_LINUX_DMA_BUF_EXT
img0 = eglCreateImageKHR(dpy, EGL_NO_CONTEXT, EGL_LINUX_DMA_BUF_EXT, 0,
        attribs0);
if (img0 == EGL_NO_IMAGE_KHR) {
  checkEglErrorState();
  return -1;
}

glGenTextures(1, &Tex);
glBindTexture(GL_TEXTURE_EXTERNAL_OES, Tex);

glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_MIN_FILTER,
     GL_LINEAR);
glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_MAG_FILTER,
     GL_LINEAR);
glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_WRAP_S,
     GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_EXTERNAL_OES, GL_TEXTURE_WRAP_T,
     GL_CLAMP_TO_EDGE);

glEGLImageTargetTexture2DOES(GL_TEXTURE_EXTERNAL_OES,
        (GLeglImageOES)img0);
if (checkEglErrorState())
  return -1;
if (checkGlErrorState())
  return -1;

GLuint fbo;
glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);

//glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, Tex, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
        GL_TEXTURE_EXTERNAL_OES, Tex, 0);
uint32_t glStatus = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if (glStatus != GL_FRAMEBUFFER_COMPLETE) {
  printf("glCheckFramebufferStatusOES error 0x%x", glStatus);
  return -1;
}

if (checkGlErrorState())
  return -1;

//Setting the background color
```

```
    glClearColor(0.5, 0.5, 0.5, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);

    *Fbo = fbo;
    *tex = Tex;
    *imgFbo = img0;
    return dma_fd;
}

void outputRenderResult(int ion_fd, struct ion_mem *mem, char *path,
        unsigned int size)
{
    FILE *fp;

    fp = fopen(path, "ab");
    fwrite(mem->virt, 1, size, fp);

    fclose(fp);
}

static void deleteRenderTexture(struct Renderer *renderer)
{  //渲染结束后，把输入纹理相关的framebuffer、image等资源释放
    if (renderer->Tex)
        glDeleteTextures(1, &renderer->Tex);

    if (renderer->imgTex &&
        (eglDestroyImageKHR(renderer->egl_display, renderer->imgTex) == EGL_FALSE)) {
        printf("destroy input tex egl image failed\n");
        return;
    }
    return;
}

static void deleteOutputTexture(struct Renderer *renderer)
{  //渲染结束后，把输出纹理相关的framebuffer、image等资源释放
    if (renderer->FboColorBufferTex) {
        glDeleteTextures(1, &renderer->FboColorBufferTex);
        glDeleteFramebuffers(1, &renderer->fbo);
    }

    if (renderer->imgFboTex &&
        (eglDestroyImageKHR(renderer->egl_display, renderer->imgFboTex) == EGL_FALSE)) {
        printf("destroy tex egl image failed\n");
        return;
    }

    return;
}

static EGLint const config_attribute_list[] = {
    EGL_RED_SIZE,     8,
    EGL_GREEN_SIZE,   8,
    EGL_BLUE_SIZE,    8,
    EGL_ALPHA_SIZE,   8,
    EGL_BUFFER_SIZE,  32,
    EGL_STENCIL_SIZE, 0,
    EGL_DEPTH_SIZE,   0,
    EGL_SAMPLES,      4,

    EGL_RENDERABLE_TYPE,EGL_OPENGL_ES2_BIT,
```

```
    //off_screen
    EGL_SURFACE_TYPE,EGL_PBUFFER_BIT,
    EGL_NONE
};

static const EGLint context_attribute_list[] = { EGL_CONTEXT_CLIENT_VERSION, 2,
                EGL_NONE };
int initEGL(struct TestCase *test)
{
    EGLConfig config;
    EGLint num_config;

    struct Renderer *renderer = &test->renderer;
    //a. 获取display连接
    renderer->egl_display = eglGetDisplay(EGL_DEFAULT_DISPLAY);
    if (renderer->egl_display == EGL_NO_DISPLAY) {
        fprintf(stderr, "Error: No display found!\n");
        return -1;
    }
    //b. 初始化获取的display连接
    if (!eglInitialize(renderer->egl_display, &renderer->egl_major,
            &renderer->egl_minor)) {
        fprintf(stderr, "Error: eglInitialise failed!\n");
        return -1;
    }

    printf("EGL Version: \"%s\"\n",
        eglQueryString(renderer->egl_display, EGL_VERSION));
    printf("EGL Vendor: \"%s\"\n",
        eglQueryString(renderer->egl_display, EGL_VENDOR));
    printf("EGL Extensions: \"%s\"\n",
        eglQueryString(renderer->egl_display, EGL_EXTENSIONS));
    printf("\n\n\n");

    if (!eglBindAPI(EGL_OPENGL_ES_API)) {
        printf("failed to bind api EGL_OPENGL_ES_API\n");
        return -1;
    }
    //c. 按照attrib_list 来选择display相应的配置，生成的配置表写入到 config中（第三个参数
    eglChooseConfig(renderer->egl_display, config_attribute_list, &config,
            1, &num_config);
    //d. 根据config创建contex
    renderer->egl_context =
        eglCreateContext(renderer->egl_display, config, EGL_NO_CONTEXT,
                context_attribute_list);
    if (renderer->egl_context == EGL_NO_CONTEXT) {
        checkEglErrorState();
        return -1;
    }

    //f. 创建pbuffer surface
    renderer->egl_surface =
        eglCreatePbufferSurface(renderer->egl_display, config, NULL);
    if (renderer->egl_surface == EGL_NO_SURFACE) {
        checkEglErrorState();
        return -1;
    }
    //g. 将刚创建的contex设置为当前工作上下文
    if (!eglMakeCurrent(renderer->egl_display, renderer->egl_surface,
            renderer->egl_surface, renderer->egl_context)) {
```

```
    checkEglErrorState();
    return -1;
  }

  eglSwapInterval(renderer->egl_display, 1);

  //获取EGL extension的相关接口
  /*  eglCreateImageKHR = (PFNEGLCREATEIMAGEKHRPROC)
        eglGetProcAddress("eglCreateImageKHR");
  glEGLImageTargetTexture2DOES = (PFNGLEGLIMAGETARGETTEXTURE2DOESPROC)
        eglGetProcAddress("glEGLImageTargetTexture2DOES");
  eglCreateSyncKHR = (PFNEGLCREATESYNCKHRPROC)
        eglGetProcAddress("eglCreateSyncKHR");
  eglDestroySyncKHR = (PFNEGLDESTROYSYNCKHRPROC)
        eglGetProcAddress("eglDestroySyncKHR");
  eglClientWaitSyncKHR = (PFNEGLCLIENTWAITSYNCKHRPROC)
        eglGetProcAddress("eglClientWaitSyncKHR");
*/

  return 0;
}

int createShader(struct Shader *shader, int type, const char *source)
{
  GLint ret;

  shader->id = glCreateShader(type);
  if (!shader->id) {
    checkGlErrorState();
    return -1;
  }

  glShaderSource(shader->id, 1, &source, NULL);
  glCompileShader(shader->id);

  glGetShaderiv(shader->id, GL_COMPILE_STATUS, &ret);
  if (!ret) {
    char *log;

    fprintf(stderr, "Error: shader compilation failed, type:%d!\n",
      type);
    glGetShaderiv(shader->id, GL_INFO_LOG_LENGTH, &ret);

    if (ret > 1) {
      log = malloc(ret);
      glGetShaderInfoLog(shader->id, ret, NULL, log);
      fprintf(stderr, "%s", log);
    }
    return -1;
  }

  return 0;
}

int initShader(struct Renderer *renderer, const char *vertex,
      const char *fragment)
{
  GLint ret;

  if (createShader(&renderer->vs, GL_VERTEX_SHADER, vertex) < 0) {
```

```
    printf("createShader vertex shader failed!\n");
    return 0;
  }

  if (createShader(&renderer->fs, GL_FRAGMENT_SHADER, fragment) < 0) {
    printf("createShader fragment shader failed!\n");
    return 0;
  }

  renderer->shader_program = glCreateProgram();
  if (!renderer->shader_program) {
    fprintf(stderr, "Error: failed to create shader_program!\n");
    return -1;
  }

  glAttachShader(renderer->shader_program, renderer->vs.id);
  glAttachShader(renderer->shader_program, renderer->fs.id);

  glBindAttribLocation(renderer->shader_program, 0, "aPosition");
  glBindAttribLocation(renderer->shader_program, 1, "aTexCoord");

  glLinkProgram(renderer->shader_program);

  glGetProgramiv(renderer->shader_program, GL_LINK_STATUS, &ret);
  if (!ret) {
    char *log;

    fprintf(stderr, "Error: shader_program linking failed!\n");
    glGetProgramiv(renderer->shader_program, GL_INFO_LOG_LENGTH,
          &ret);

    if (ret > 1) {
      log = malloc(ret);
      glGetProgramInfoLog(renderer->shader_program, ret, NULL,
            log);
      fprintf(stderr, "%s", log);
    }
    return -1;
  }
  glUseProgram(renderer->shader_program);

  return 0;
}

int main(int argc, char *argv[])
{
  int ret = 0;
  bool countine_test = 0;
  static GLfloat uScale = 1.0;
  struct TestCase test;
  struct Renderer *renderer = &test.renderer;
  struct timeval tv;
  uint32_t last_time = 0;
  uint32_t current_time = 0;

  static GLint vTexSamplerHandler;

  test.screen_width = WIDTH;
  test.screen_height = HEIGHT;
  test.tex_width = TEXTURE_WIDTH;
```

```
test.tex_height = TEXTURE_HEIGHT;

test.ion_fd = ion_open();
if (test.ion_fd < 0)
  return -1;
//（1）初始化EGL，创建与本地窗口的连接
if (initEGL(&test) < 0) {
  printf("eglInitial failed!\n");
  return -1;
}

printf("GL Vendor: \"%s\"\n", glGetString(GL_VENDOR));
printf("GL Renderer: \"%s\"\n", glGetString(GL_RENDERER));
printf("GL Version: \"%s\"\n", glGetString(GL_VERSION));
printf("GL Extensions: \"%s\"\n", glGetString(GL_EXTENSIONS));
printf("\n\n\n");
//（2）创建vertex shader和fragment shader
if (initShader(renderer, vertex_shader_source, fragment_shader_source) <
  0) {
  printf("initShader failed\n");
  return -1;
}
glUseProgram(renderer->shader_program);

//（3）Setting the vertex coordinates texture coordinates
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vVertices);
glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, vTexVertices);
glEnableVertexAttribArray(1);

glUniform1f(glGetUniformLocation(renderer->shader_program, "aScale"),
    uScale);

//（4）off_screen,creat FBO for staging rendering result
ret = createFBO(renderer->egl_display, test.ion_fd,
    &renderer->FboMem, &renderer->fbo,
    &renderer->FboColorBufferTex,
    &renderer->imgFboTex,
    test.screen_width, test.screen_height);
if (ret < 0) {
  printf("create FBO failed\n");
  return -1;
}

while (1) {
  //（5）Import the texture in the way of dma,read the file here
  //"./media/pic_bin/1920x1080.nv21"是GPU读取的纹理数据
  ret = load_texture(test.ion_fd, &renderer->TexMem,
      "./media/pic_bin/1920x1080.nv21",
      TEXTURE_WIDTH * TEXTURE_HEIGHT * 3 / 2);
  if (ret < 0) {
    printf("load texture failed!\n");
    return -1;
  }

  gettimeofday(&tv,NULL);
  last_time = tv.tv_sec * 1000 + tv.tv_usec / 1000;

  //(6)Set up input texture
```

```
    renderer->Tex = setupTexture(renderer->egl_display,
            renderer->TexMem.dmafd,
            &renderer->imgTex,
            test.tex_width, test.tex_height);

    glUniform1f(glGetUniformLocation(renderer->shader_program,
                "aScale"), uScale);

    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0,
             vTexVertices);
    glEnableVertexAttribArray(1);
    vTexSamplerHandler = glGetUniformLocation(
      renderer->shader_program, "uTexSampler");
    glUniform1i(vTexSamplerHandler, 0);
     （7）渲染
    glBindTexture(GL_TEXTURE_EXTERNAL_OES, renderer->Tex);
    //If do not clear color buffer,the subsequent pictures will be
    //cover it;but the uncovered part of picture will be remained.
    glClear(GL_COLOR_BUFFER_BIT);
    Redraw(&test);

    gettimeofday(&tv,NULL);
    current_time = tv.tv_sec * 1000 + tv.tv_usec / 1000;
    printf(" %f fps\n", (float) 1000 / (current_time - last_time));

    //sendForDisplay2(0, 0, 0, 15, test.screen_width,
    //    test.screen_height, DRM_FORMAT_NV21,
    //    &renderer->FboMem);
    //(8)渲染结果写入本地文件
    outputRenderResult(test.ion_fd, &renderer->FboMem,
        "/mnt/render_result.yuv",
        test.screen_width * test.screen_height * 3 /
          2);
    //(9) 删除输入纹理对象
    deleteRenderTexture(renderer);
    ion_free(test.ion_fd, &renderer->TexMem);
    //printf("Render finish!\n");

    if (!countine_test) {
      char c = getchar();
      if (c == 'b')
        break;
      else if (c == 'n')
        countine_test = true;
    }

    if (uScale >= 0.0)
      uScale -= 0.01;
    else
      uScale = 1.0;
  }
  //（10）程序退出前，删除输出纹理对象、释放display、释放申请的buffer
  deleteOutputTexture(renderer);
  eglTerminate(renderer->egl_display);
  ion_free(test.ion_fd, &renderer->FboMem);
  close(test.ion_fd);

  printf("Render exit!\n");
  return 0;
}
```

⚠注意

由于代码篇幅过长，demo 启指导作用，以上只截取关键步骤进行说明，工程并不完全。重点在于setupTexture、createFBO两函数中，如何根据不同的图像数据格式配置 plane 信息。

4.4 OpenCL 开发

Mali-G31 支持标准的 OpenCL 2.0 Full profile。关于 OpenCL 开发及优化，请查看 Arm 官方文档《Arm Mali Bifrost and Valhall OpenCL Developer Guide Version 4.8》https://developer.arm.com/documentation/101574/0408?lang=en。

4.5 调试方法

4.5.1 debug 节点（sys/kernel/debug）

4.5.1.1 Mali-bifrost 架构

- 前置条件

加载 GPU 驱动，mali_kbase.ko(不同的平台，GPU 驱动名字可能不同，Mali-utgard 架构编译出的 GPU 模块名为 mali.ko)。

手动挂载 debugfs(mount -t debugfs none ./sys/kernel/debug) – 将 debugfs 挂载在./sys/kernel/debug/ 之下，none 表示挂载是非物理内存，是虚拟的文件系统。

- 查看 gpu 运行信息

```
/sys/kernel/debug/sunxi_gpu # cat dump
idle:on;                //不启作用，只是一个标志，标志着内部可自己idle功能是否开启；
scenectrl:off;          //场景控制，根据不同的场景（跑分模式、游戏模式），会更改电压、频率等；
dvfs:on;                //动态调频调压，一种降低功耗的策略；
independent_power:no;       //指示GPU是否只用单独的电源供电，如：TV303的GPU挂载在SYS路，和其余模块共用电源，
    此时为no；
Frequency:600MHz;           //当前工作频率；
Utilisation from last show:0%; //GPU的负载/利用率；

// GPU寄存器，可以反映GPU的工作状态；
Register state:  GPU_IRQ_RAWSTAT=0x00000200 GPU_STATUS=0x00000100
 JOB_IRQ_RAWSTAT=0x00000000 JOB_IRQ_JS_STATE=0x00000000
 JS0_STATUS=0x00000000    JS0_HEAD_LO=0xe70fad40
 JS1_STATUS=0x00000000    JS1_HEAD_LO=0xe70fa8c0
 JS2_STATUS=0x00000000    JS2_HEAD_LO=0x00000000
 MMU_IRQ_RAWSTAT=0x00000000 GPU_FAULTSTATUS=0x00000000
 GPU_IRQ_MASK=0x00010581   JOB_IRQ_MASK=0x00070007   MMU_IRQ_MASK=0x00ff00ff
```

```
PWR_OVERRIDE0=0x00000000  PWR_OVERRIDE1=0x00000000
SHADER_CONFIG=0x00000000  L2_MMU_CONFIG=0x00000000
SHADER_READY_HI=0x00000000  SHADER_READY_LO=0x00000001 //GPU内部，SHADER开启
L2_READY_HI=0x00000000  L2_READY_LO=0x00000001      //GPU内部，L2_cache子系统开启
SHADER_PWRTRANS_HI=0x00000000  SHADER_PWRTRANS_LO=0x00000000
L2_PWRTRANS_HI=0x00000000  L2_PWRTRANS_LO=0x00000000  //indicates that L2 switch between power-on and
    power-off
TILER_PWRTRANS_HI=0x00000000  TILER_PWRTRANS_LO=0x00000000
```

- 查看 gpu 工作状态

```
//如果没有此节点，则说明该ic不支持此功能
/sys/kernel/debug/pm_genpd/pd_gpu # ls
active_time(活跃时间)    devices       sub_domains
current_state(当前状态)   idle_states    total_idle_time(总的空闲时间)

/sys/kernel/debug # cat pm_genpd_summary                                    <
domain                  status       slaves
   /device                           runtime status
----------------------------------------------------------------------
pd_av1                  off-0
pd_ve                   on
   /devices/platform/soc@2900000/1c0e000.ve          unsupported
pd_tvcap                 on
   /devices/genpd:0:5700000.tvtop                 active
pd_tvfe                 on
   /devices/genpd:1:5700000.tvtop                 active
pd_gpu                  on
   /devices/platform/1800000.gpu                  active    //GPU模块状态，只有再支持power-domains的IC上，才有此节点
```

4.5.1.2 Mali-utgard 架构

```
echo "enable:1;" > /sys/kernel/debug/mali/write;   //debug的总开关。如果不设置，dump将不会看到任何GPU相关的信
    息；

echo "level:1;" > /sys/kernel/debug/mali/write;    //会将vf表及当前频率打印出来


echo "frequency:1;" > /sys/kernel/debug/mali/write; // 0/1 :关闭/开启GPU频率的打印，否则将会设置GPU的频率，例如："
    frequency:600000000;" GPU会将频率设定为600MHz

echo "voltage:1;" > /sys/kernel/debug/mali/write; // 0/1 :关闭/开启GPU当前电压的打印，否则将会更改GPU的电压，例
    如："voltage:1100000;" GPU电压将会设定为1.1v，更改GPU电压，需要注意GPU是不是独立供电，A40i GPU与其他模
    块共用sys路电压，不能更改GPU的电压。

echo "tempctrl:x;" > /sys/kernel/debug/mali/write;
x：0/1 关闭/开启 查看温控状态的开关，打开后，才能通过dump查看到温控相关的信息
  2/3 关闭/开启 GPU温控开关

echo "dvfs:x;" > /sys/kernel/debug/mali/write;
x：0/1 关闭/开启 查看dvfs状态的开关，打开后，才能通过dump查看到dvfs相关的信息
  2/3 关闭/开启 GPU dvfs开关

cat /sys/kernel/debug/mali/dump //打印GPU运行时，相关的状态信息。 上面提到的开关是前提，只有打开相应的开关后，
    才能通过cat dump查看到对应的信息
```

4.5.2 调频相关

- 手动模式–调频

```
Mali系列的GPU是支持手动调频的，手动调频开启后，dvfs将不在启作用，即GPU将维持在手动设定的频率工作。开启&关闭
      手动模式的步骤如下。

* 开启手动调频
 * echo "frequency:1;" >write; echo "voltage:1;" >write; //任选其中之一
 * echo "voltage:900;" >write;  //设定电压为900mv
 * echo "frequency:600;" >write; //设定GPU的工作频率为600Mhz，注意，这里设定的频率&电压要和vf表对应；
* 关闭手动调频
 * echo "freqency:0" >write; echo "voltage:0" >write;  //任选其中之一，关闭后，dvfs会恢复作用
```

- 场景控制

Mali 系列 GPU，有两种工作模式，分别是 normal、performance 两种。sys 节点下有关于场景控制的节点。

```
./sys/devices/platform/soc@3000000/1800000.gpu/scenectrl/
echo 1 > command //开启场景控制，0 关闭；
```

开启场景控制后，会进行一次调频，让 GPU 工作在最高频率。同样，开启场景控制后，dvfs 不再起作用，GPU 将保持最高频率工作。如果此时，手动更改频率，GPU 将维持在手动更改后的频率工作 (这不是一种合理的使用场景，即想要 GPU 工作在高性能模式，就不需要手动修改频率了)。

说明

sys/kernel/debug/sunxi_gpu/ 节点下，只能查看 GPU 的工作模式，不能更改 GPU 的工作模式；如果想要更改，需要去./sys/devices/platform/soc@3000000/1800000.gpu/scenectrl/路径

- 失能 dvfs（以下方式任选其一）

1. echo “dvfs:0” > write; //GPU 将会工作在 dvfs 失能那一刻的频率；
2. 开启手动调频模式；//GPU 会工作在手动设置的频率；
3. 开启场景控制；//GPU 会工作在 vf 表中设置的最高频率；

4.5.3 sysfs 节点

4.5.3.1 mali-valhall 架构

由于部分 Android 平台 GKI 的要求，默认不开启CONFIG_DEBUG_FS, 导致 cpu_monitor 工具无法获取 gpu 的频率信息，于是 mali-valhall dirver 将 debugfs 相应的功能移动到 sysfs。具体的路径位于：/sys/devices/platform/soc@3000000/1800000.gpu/sunxi_gpu。接下来介绍下 sysfs 调试节点的使用方法。

- scene_ctrl

可读写，GPU 是否运行在性能模式。

```
echo 1 >scene_ctrl; GPU将会关闭dvfs，运行在最高频。

0：关闭
1：开启
```

- sunxi_gpu_freq

可读写，用来查看、改变 GPU 的频率。

```
/sys/devices/platform/soc@3000000/1800000.gpu/sunxi_gpu # cat sunxi_gpu_freq
Frequency:696MHz; //GPU当前的频率
Utilisation from last show:0%; //GPU当前的负载率，这两个信息是用户最常用的

// 更改GPU的频率，单位是MHz，将GPU频率设为600MHz；同时会将dvfs关闭
/sys/devices/platform/soc@3000000/1800000.gpu/sunxi_gpu # echo 600 >sunxi_gpu_freq
/sys/devices/platform/soc@3000000/1800000.gpu/sunxi_gpu # cat sunxi_gpu_freq
Frequency:600MHz;
Utilisation from last show:0%;

如果更改频率失败，会有以下提示：
/sys/devices/platform/soc@3000000/1800000.gpu/sunxi_gpu # echo 1 >sunxi_gpu_freq
[ 1212.444659] mali 1800000.gpu: sunxi:mali_kbase:set gpu core clock to 1000000 err -22!
注意：虽然更改频率失败，但仍会影响dvfs_ctrl的值，将dvfs_ctrl设为0，即dvfs不再起作用。读写频率的单位MHz。
```

- sunxi_gpu_volt

可读写，用来查看、改变 GPU 的电压 (GPU 独立供电时才能更改电压)。

```
/sys/devices/platform/soc@3000000/1800000.gpu/sunxi_gpu # cat sunxi_gpu_volt
920mv

//非独立供电时，尝试更改电压会报错
/sys/devices/platform/soc@3000000/1800000.gpu/sunxi_gpu # echo 900 >sunxi_gpu_volt
[ 1315.910428] mali 1800000.gpu: sunxi:mali_kbase:[ERR]: GPU not support change voltage!!!
ash: write error: Operation not permitted
注意：读写电压的单位是mv
```

- sunxi_gpu_dvfs

可读写，用来查看、配置 dvfs 对 GPU 的控制。

```
echo 0 >sunxi_gpu_dvfs；//dvfs将不再影响GPU的频率，这个配置，只有内核开启PM_DEVFREQ
并且GPU驱动开启CONFIG_MALI_DEVFREQ选项时才有效。
```

- sunxi_gpu_info

只读，查看 GPU 详细的运行信息和几组重要寄存器的值。

```
/sys/devices/platform/soc@3000000/1800000.gpu/sunxi_gpu # cat sunxi_gpu_info
voltage:920mV;
idle:on;
scenectrl:off;
dvfs:off;
independent_power:no;
Frequency:600MHz;
Utilisation from last show:0%;

Register state:
GPU_IRQ_RAWSTAT=0x00000000    GPU_STATUS=0x00000100
JOB_IRQ_RAWSTAT=0x90910010    JOB_IRQ_JS_STATE=0x00000001
JS0_STATUS=0x00000000  JS0_HEAD_LO=0x00000000
JS1_STATUS=0x00000000  JS1_HEAD_LO=0x00000000
JS2_STATUS=0x00000000  JS2_HEAD_LO=0x00000000
MMU_IRQ_RAWSTAT=0x00000000    GPU_FAULTSTATUS=0x00000000
GPU_IRQ_MASK=0x00000000 JOB_IRQ_MASK=0x00000000 MMU_IRQ_MASK=0x00000000
PWR_OVERRIDE0=0x00000000      PWR_OVERRIDE1=0x00000000
SHADER_CONFIG=0x20000000      L2_MMU_CONFIG=0x00000000
TILER_CONFIG=0x00000000 JM_CONFIG=0x000f0000
```

说明

只有 mali-valhall 架构的 GPU 才支持此功能。

4.5.4 调试工具

- Systrace

Systrace 是 Android4.1 版本之后推出的，对系统性能分析的工具，实现原理是在系统的一些关键路径（比如 System Service, 虚拟机，Binder 驱动）插入一些信息收集，从而获取系统关键路径的运行时间信息，进而得到整个系统的运行性能信息。Systrace 的功能包括跟踪系统的 I/O 操作、内核工作队列、CPU 负载以及 Android 各个子系统的运行状况等。借助该工具能有效提升对显示绘制通路的分析调试效率，具体的使用说明可参照以下链接：
https://developer.android.google.cn/studio/profile/systrace?hl=zh_cn。

- PVRTrace

PVRTrace 是 PowerVR 提供的用于性能分析、鉴定瓶颈、修改应用程序等功能的工具集 PowerVR Graphics Tools 中的一部分。PVRTrace 一般用于完成记录与分析操作，具体而言，PVRTrace 可以捕获 OpenGL ES 应用程序的 API 调用，方便溯源分析代码流程。通过 PVRTrace，可以抓取每个 OpenGL ES 调用接口及每一帧绘制的图像，并可将抓取的调用进行回放操作。具体的使用说明可参考以下链接：
https://blog.imaginationtech.com/powervr-graphics-sdk-v3-2-brings-advanced-features/。

著作权声明

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明


、 
、全志科技、  （不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。