RGA FAQ

文件标识：RK-PC-YF-404

发布版本：V1.1.2

日期：2023-06-28

文件密级：□绝密 □秘密 □内部资料 ■公开

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址： 福建省福州市铜盘路软件园A区18号

网址： www.rock-chips.com

客户服务电话： +86-4007-700-590

客户服务传真： +86-591-83951833

客户服务邮箱： fae@rock-chips.com

读者对象

本文档（本指南）主要适用于以下工程师：

- 技术支持工程师
- 软件开发工程师

修订记录

日期	版本	作者	修改说明
2021/06/28	1.0.0	余乔伟	初始版本
2022/12/21	1.1.0	余乔伟	增加针对multi_rga驱动的异常案例
2023/02/09	1.1.1	余乔伟	更正文档格式
2023/06/28	1.1.2	余乔伟	补充Q&A

目录

1. 概述

本文针对于RGA驱动以及用户态接口librga，总结RK平台上调用RGA硬件实现OSD（On Screen Display）和 GUI（Graphics User Interface）图形绘制加速功能时遇到的一些常见问题。

2. 版本说明

2.1 硬件版本

RGA硬件主要分为三个版本版本：RGA1、RGA2、RGA3。具体平台搭载信息、支持功能以及限制条件可以查看 Rockchip_Developer_Guide_RGA_CN ——概述 章节。

2.2 软件版本

以下仅提供常用的版本查询方式，详细的可以查阅 Rockchip_Developer_Guide_RGA_CN —— 版本说明 章节。

2.2.1 librga

API版本号分为主版本号、次版本号、修订版本号、编译版本号，四个等级版本号对应不同程度的功能更新。

- 版本号查询

  比较通用的查询方法如下：

  ```
  strings librga.so |grep rga_api |grep version
  ```

- 更新版本方式

  当发现版本不满足要求时，可以通过以下方式获取源码或预编译的库文件。

  - github预编译仓库：

    https://github.com/airockchip/librga

  - 联想网盘链接：

    https://console.zbox.filez.com/l/fuGojC （提取码：rkrga）

2.2.2 RGA driver

驱动版本号分为主版本号、次版本号、修订版本号、编译版本号，四个等级版本号对应不同程度的功能更新，通常发布的SDK中HAL库与驱动是匹配的，librga内部会进行校验版本，开发者无需关心该版本。当出现单独更新librga时出现以下报错，则须要更新驱动至对应版本即可。

- 版本号查询

  不同芯片平台debug节点开启路径不同，通常有以下两个路径。

```
cat /sys/kernel/debug/rkrga/driver_version
cat /proc/rkrga/driver_version
```

- 更新版本方式

  当发现版本不满足要求时，可以通过以下方式获取源码更新kernel。

  - 联想网盘链接：

    https://console.zbox.filez.com/l/7oOrKO （提取码：rkrga）

2.2.3 版本对应关系

使用RGA时需要确认保证当前的运行环境是可以正常工作的，下表为常用的librga与驱动版本对应关系。

librga版本	对应驱动	硬件支持
无版本号	对应SDK内驱动	RGA1、RGA2
1.0.0 ~ 1.3.2	RGA Device Driver（kernel - 4.4及以上） RGA2 Device Driver（无版本号或v2.1.0）	RGA1、RGA2
> 1.4.0	RGA multicore Device Driver（v1.2.0及以上）	RGA2、RGA3
> 1.9.0	RGA Device Driver（kernel-4.4及以上） RGA2 Device Driver（无版本号和v2.1.0） RGA multicore Device Driver（v1.2.0及以上）	RGA1、RGA2、RGA3

通常发布的SDK中是版本是匹配的，但是出于一些应用对高版本librga.so的依赖，可以使用以下百度网盘链接获取RGA模块代码更新包：

https://console.zbox.filez.com/l/mu2SOR （提取码：rkrga）

- update-to-MULTI_RGA

  原有驱动为RGA Device Driver、RGA2 Device Driver时，使用该更新包更新驱动到RGA multicore Device Driver，并更新匹配版本的librga。

- MUTIL_RGA

  原有驱动为RGA multicore Device Driver时，使用该更新包更新驱动版本，并更新匹配版本的librga。

- RGA2

  原有驱动为RGA2 Device Driver时，使用该更新包更新驱动版本，并更新匹配版本的librga。

- RGA1

  原有驱动为RGA Device Driver时，使用该更新包更新驱动版本，并更新匹配版本的librga。

3. 调试说明

3.1 HAL层运行日志

3.1.1 日志开关

- Android平台

  Android平台支持使用属性配置librga是否开启HAL层日志打印：

  - 开启日志打印：

```
setprop vendor.rga.log 1
logcat -s librga
```

  - 设置日志等级：

    日志等级分为全打印（0）、DEFAULT（1）、DEBUG（3）、INFO（4）、WRANING（5）、ERROR（6）。

```
setprop vendor.rga.log_level 6
```

- Linux平台

  Linux平台支持通过设置环境变量的方式（librga 1.9.0版本以上），开启/关闭HAL层日志打印：

  - 开启日志打印：

```
export ROCKCHIP_RGA_LOG=1
```

  - 设置日志等级：

    日志等级分为全打印（0）、DEFAULT（1）、DEBUG（3）、INFO（4）、WRANING（5）、ERROR（6）。

```
export ROCKCHIP_RGA_LOG_LEVEL=6
```

3.1.2 日志说明

- 初始化日志

当每个进程首次调用librga时，会初始化librga的单例，并打印当前的API版本号等信息

```
E rockchiprga: rga_api version 1.9.0_[0]
```

当出现驱动版本与librga版本不适配时，会打印对应的报错。

> 当驱动版本较低时，会启动兼容模式，并在单例初始化时打印如下日志，这时可以考虑更新驱动到日志提示的版本，也可以继续使用兼容模式运行。

```
librga fail to get driver version! Compatibility mode will be enabled.

29 im2d_rga_impl rga_version_below_minimun_range_user_driver(310): The driver may be compatible, but it is best to update the driver to version 1.2.4. You can try to update the SDK or update the <SDK>/kernel/drivers/video/rockchip/rga3 directory individually. current version: librga 1.8.5, driver .
```

> 当librga版本较低时，通过imStrError()会返回一些Invalid parameters相关的报错，这表明当前的librga版本过低，需要更新librga版本。

```
Invalid parameters: invaild GraphicBuffer, can not get fd and virtual address,
```

- 运行日志

```
D librga  : <<<<-------- print rgaLog -------->>>>
//以下部分为传入librga的参数打印。
D librga  : src->hnd = 0x0 , dst->hnd = 0x0 , src1->hnd = 0x0
//三个通道（src、src1、dst）传入的内存句柄的值
D librga  : src: Fd = 00 , phyAddr = 0x0 , virAddr = 0xb400007431ed6040
//src通道传入的内存类型对应的值，对应为DMA_FD、物理地址、虚拟地址。
D librga  : dst: Fd = 00 , phyAddr = 0x0 , virAddr = 0xb400007431b4f040
//dst通道传入的内存类型对应的值，对应为DMA_FD、物理地址、虚拟地址。
D librga  : src: Fd = -01 , buf = 0xb400007431ed6040, mmuFlag = 1, mmuType = 0
//src通道将配置传递的内存类型对应的值以及是否使能MMU，这里HAL层选择虚拟地址传入驱动。
D librga  : dst: Fd = -01 , buf = 0xb400007431b4f040, mmuFlag = 1, mmuType = 0
//dst通道将配置传递的内存类型对应的值以及是否使能MMU，这里HAL层选择虚拟地址传入驱动。
E librga  : blend = 0 , perpixelAlpha = 1
//混合模式以及图像格式是否本身存在Alpha值
D librga  : scaleMode = 0 , stretch = 0;
//缩放模式（RGA1）。
E librga  : rgaVersion = 3.200000  , ditherEn =0
//硬件版本号，16阶灰度图（Y4）dither使能。
D librga  : srcMmuFlag = 1 , dstMmuFlag = 1 , rotateMode = 0
//MMU使能标志位，旋转模式。
D librga  : <<<<-------- rgaReg -------->>>>
//以下为配置入驱动的参数打印。
E librga  : render_mode=0 rotate_mode=0
//RGA运行模式，旋转模式。
E librga  : src:[0,b400007431ed6040,b400007431fb7040],x-y[0,0],w-h[1280,720],vw-vh[1280,720],f=0    //src通道的内存、图像参数、格式信息。
E librga  : dst:[0,b400007431b4f040,b400007431c30040],x-y[0,0],w-h[1280,720],vw-vh[1280,720],f=0    //dst通道的内存、图像参数、格式信息。
E librga  : pat:[0,0,0],x-y[0,0],w-h[0,0],vw-vh[0,0],f=0
//pat/src1通道的内存、图像参数、格式信息，由于当前模式没有使用到该通道，所以参数均为0。
//以下部分开发者通常不用关心，为librga配置入驱动的不同模式的相关参数。
E librga  : ROP:[0,0,0],LUT[0]
//ROP模式配置，LUT表配置
E librga  : color:[0,0,0,0,0]
//colorkey配置（max color, min color），填充颜色配置(前景色配置，背景色配置，颜色填充配置)
E librga  : MMU:[1,0,80000521]
//MMU配置
```

```
E librga  : mode[0,0,0,0]
//palette、csc、colorkey配置
E librga  : Full CSC : en[0]
//full csc使能标志
E librga  : gr_color_x [0, 0, 0]
//填充颜色配置，对应R、G、B的颜色值
```

3.2 驱动调试节点

3.2.1 调试节点路径

不同的SDK kernel的配置不同，通常RGA的调试节点存在在以下两个目录其中一个或者均存在：

- 使用默认使能CONFIG_ROCKCHIP_RGA_DEBUG_FS编译选项的kernel。

```
/sys/kernel/debug
```

- 使能ROCKCHIP_RGA_PROC_FS编译选项的kernel。

```
/proc
```

除了默认的开启外，也可以根据自己的项目需求修改kernel的编译选项实现自定义RGA调试节点路径。

3.2.2 调试节点名称

不同的驱动上调试节点的名称是不相同的，后续更新的驱动中会统一为rkrga，rgax_debug的名称目前已经弃用。

驱动名称	调试节点路径
RGA Device Driver	rga_debug
RGA2 Device Driver（无版本号）	rga2_debug
RGA2 Device Driver（v2.1.0）	rkrga
RGA multicore Device Driver	rkrga

3.2.3 调试节点功能

3.2.3.1 概述

- rga_debug/rga2_debug

rga_debug/rga2_debug节点仅支持运行日志开关功能。

- rkrga

该版本调试节点支持运行日志开关、负载查询、版本查询、硬件信息查询、内存/任务管理器状态查询等功能。

3.2.3.2 运行日志

3.2.3.2.1 日志开关

- 运行日志开关节点名称

驱动名称	调试节点路径
RGA Device Driver	rga_debug/rga
RGA2 Device Driver（无版本号）	rga2_debug/rga2
RGA2 Device Driver（v2.1.0）	rkrga/debug
RGA multicore Device Driver	rkrga/debug

- 调试功能说明

不同的驱动版本调试日志的开关方式是相同的，都是对rga/rga2/debug节点进行操作。

以RGA multicore Device Driver为例，在对应的目录下可以通过cat节点，获取对应功能说明：

```
/# cd /sys/kerne/debug/rkrga/
/# cat debug
REG [DIS]
MSG [DIS]
TIME [DIS]
INT [DIS]
CHECK [DIS]
STOP [DIS]

help:
 'echo reg > debug' to enable/disable register log printing.
 'echo msg > debug' to enable/disable message log printing.
 'echo time > debug' to enable/disable time log printing.
 'echo int > debug' to enable/disable interruppt log printing.
 'echo check > debug' to enable/disable check mode.
 'echo stop > debug' to enable/disable stop using hardware
```

> echo reg > debug：该命令开关 RGA 寄存器配置信息的打印。打开该打印时，将会打印每次 rga 工作寄存器的配置值
>
> echo msg> debug：该命令开关 RGA 上层配置参数信息的打印。打开该打印时，上层调用 rga 驱动传递的参数将被打印出来。

> echo time> debug：该命令开关 RGA 工作耗时信息的打印。打开该打印时，将会打印每一次的调用 rga 工作的耗时
>
> echo check> debug：该命令开关 RGA 内部的测试 case。打开该打印时，将会在 RGA 每次工作的时候检查相关的参数，主要是内存的检查，和对齐是否满足要求。若输出如下 log 表示通过检查。若内存存在越界的情况，将会导致内核 crash。可以通过 cash 之前的打印 log 确认是 src 数据的问题还是 dst 数据的问题。
>
> echo stop> debug：该命令开关 RGA 的工作状态。开启时，rga 将不工作直接返回。用于一些特殊情况下的调式。
>
> echo int> debug：该命令开关 RGA 寄存器中断信息的打印。打开该打印时，将会在 RGA 进入中断后打印中断寄存器和状态基础器的当前值。
>
> echo slt> debug：该命令让 rga 驱动执行内部 SLT case 测试 rga 硬件是否正常。 若输出日志“rga slt success !!”则表示功能正常。

- 开关调试节点

日志打印的开启与关闭命令是相同的，每次输入命令进行切换状态（开启/关闭），可以通过cat debug节点或者输入命令后打印的日志信息（“open xxx”或者“close xxx”）确认日志打印功能是否如预期般开启或者关闭。

```
echo <cmd> > <节点名>
```

以RGA multicore Device Driver为例，开启运行日志 ‘msg’

```
/# cd /sys/kernel/debug/rkrga/
/# cat debug
REG [DIS]
MSG [DIS]
TIME [DIS]
INT [DIS]
CHECK [DIS]
STOP [DIS]

help:
 'echo reg > debug' to enable/disable register log printing.
 'echo msg > debug' to enable/disable message log printing.
 'echo time > debug' to enable/disable time log printing.
 'echo int > debug' to enable/disable interruppt log printing.
 'echo check > debug' to enable/disable check mode.
 'echo stop > debug' to enable/disable stop using hardware
/# echo msg > debug
/# cat debug
REG [DIS]
MSG [EN]
TIME [DIS]
INT [DIS]
CHECK [DIS]
STOP [DIS]

help:
 'echo reg > debug' to enable/disable register log printing.
 'echo msg > debug' to enable/disable message log printing.
 'echo time > debug' to enable/disable time log printing.
 'echo int > debug' to enable/disable interruppt log printing.
```

```
 'echo check > debug' to enable/disable check mode.
 'echo stop > debug' to enable/disable stop using hardware
/# echo msg > debug
/# cat debug
REG [DIS]
MSG [DIS]
TIME [DIS]
INT [DIS]
CHECK [DIS]
STOP [DIS]

help:
 'echo reg > debug' to enable/disable register log printing.
 'echo msg > debug' to enable/disable message log printing.
 'echo time > debug' to enable/disable time log printing.
 'echo int > debug' to enable/disable interruppt log printing.
 'echo check > debug' to enable/disable check mode.
 'echo stop > debug' to enable/disable stop using hardware
```

开启/关闭运行日志时，内核日志会有对应的日志。

```
/# echo reg > /sys/kerne/debug/rkrga/debug
/# dmesg -c                    //For logs opened through nodes, the printing
level is KERNEL_DEBUG. You need to run the dmesg command to view the
corresponding logs on the serial port or adb.
[ 4802.344683] rga2: open rga2 reg!
/# echo reg > /sys/kernel/debug/rga2_debug/rga2
/# dmesg -c
[ 5096.412419] rga2: close rga2 reg!
```

3.2.3.2.2 日志说明

对于RGA的问题调试需要借助日志来确认RGA硬件最终执行的工作，当HAL层的参数传入驱动后，以下日志将描述着对应的参数。通常我们调试常用到msg、reg和time三种模式。

- msg模式
  - RGA Device Driver、RGA2 Device Driver

```
rga2: open rga2 test MSG!                                   //msg日志开启打印。
rga2: cmd is RGA2_GET_VERSION                               //获取版本号功能，每个进程第一次调用librga时会查询硬件版本。
rga2: cmd is RGA_BLIT_SYNC                                  //显示当前传入的工作模式。
rga2: render_mode:bitblt,bitblit_mode=0,rotate_mode:0   //render_mode显示调用接口，bitblit_mode为当前混合模式（0：双通道模式——A+B->B， 1：三通道模式A+B->C），rotate_mode为旋转角度。
rga2: src : y=0 uv=b4000072cc8bc040 v=b4000072cc99d040 aw=1280 ah=720 vw=1280 vh=720 xoff=0 yoff=0 format=RGBA8888  //src通道的图像数据参数：y：如有则为fd的值，uv：如有则为虚拟地址的值， v：vw * vh + uv， aw、ah：实宽实高，即实际操作图像区域，vw、vh：虚宽虚高，即图像本身大小，xoff、yoff：x、y方向的偏移量，format：传入的图像数据格式。
rga2: dst : y=0 uv=b4000072cc535040 v=b4000072cc616040 aw=1280 ah=720 vw=1280 vh=720 xoff=0 yoff=0 format=RGBA8888  //dst通道的图像数据参数。
rga2: mmu : src=01 src1=00 dst=01 els=00                    //MMU使能标志，0为关闭，1为开启。
rga2: alpha : flag 0 mode0=0 mode1=0                        //blend相关配置
rga2: blend mode is no blend                                //blend混合模式
rga2: yuv2rgb mode is 0                                     //csc模式
rga2: *** rga2_blit_sync proc ***
```

- RGA multicore Device Driver
  - 内存管理器日志

```
rga: import buffer info:
rga_common: external: memory = 0xb400007458406000, type = virt_addr
                                                            //memory：内存的数值，type：内存类型
rga_common: memory param: w = 1280, h = 720, f = RGBA8888(0x0), size = 0
                                                            //w/h/f：以图像画布的形式描述内存大小，size：内存大小
rga_dma_buf: iova_align size = 3686400                      //iova对齐后的大小
```

  - 任务请求日志

```
rga: Blit mode: request id = 192732                         //运行模式以及request id
rga_debugger: render_mode = 0, bitblit_mode=0, rotate_mode = 0
                                                            //render_mode显示调用接口，bitblit_mode为当前混合模式（0：双通道模式——A+B->B， 1：三通道模式A+B->C），rotate_mode为旋转角度。
rga_debugger: src: y = 19 uv = 0 v = e1000 aw = 1280 ah = 720 vw = 1280 vh = 720
                                                            //src通道的图像数据参数：y：如有则为fd的值， uv：如有则为虚拟地址的值， v：vw * vh + uv， aw、ah：实宽实高，即实际操作图像区域，vw、vh：虚宽虚高，即图像本身大小。
rga_debugger: src: xoff = 0, yoff = 0, format = 0x0, rd_mode = 1
                                                            //xoff、yoff：x、y方向的偏移量，format：传入的图像数据格式，rd_mode：当前通道读/写数据模式（1：raster，2：FBC，3：tile 16*16）
rga_debugger: dst: y=1a uv=0 v=e1000 aw=1280 ah=720 vw=1280 vh=720
                                                            //dst通道的图像数据参数
rga_debugger: dst: xoff = 0, yoff = 0, format = 0x0, rd_mode = 1
rga_debugger: mmu: mmu_flag=0 en=0                          //MMU使能标志，0为关闭，1为开启。使用rga_buffer_handle_t调用时禁用该配置，由驱动抉择最优配置。
```

```
rga_debugger: alpha: rop_mode = 0                    //alpha/ROP模式使能
rga_debugger: yuv2rgb mode is 0                      //CSC模式
rga_debugger: set core = 0, priority = 0, in_fence_fd = -1
                                                     //set_core：用户态指定
的核心，priority：用户态指定的优先级，in_fence_fd：用户态传递的acquire_fence fd
```

- 硬件匹配日志

```
rga_policy: start policy on core = 1
rga_policy: start policy on core = 2
rga_policy: start policy on core = 4                 //遍历所有的核心支持情况
rga_policy: RGA2 only support under 4G memory!       //对应核心不支持的原因日
志
rga_policy: optional_cores = 3                       //当前请求可匹配的硬件核
心合集
rga_policy: assign core: 1                           //匹配后绑定的硬件核心标
识
```

- 对应硬件参数日志

```
rga3_reg: render_mode:bitblt, bitblit_mode=0, rotate_mode:0
rga3_reg: win0: y = ffc70000 uv = ffd51000 v = ffd89400 src_w = 1280
src_h = 720
rga3_reg: win0: vw = 1280 vh = 720 xoff = 0 yoff = 0 format = RGBA8888
rga3_reg: win0: dst_w = 1280, dst_h = 720, rd_mode = 0
rga3_reg: win0: rot_mode = 1, en = 1, compact = 1, endian = 0
rga3_reg: wr: y = ff8e0000 uv = ff9c1000 v = ff9f9400 vw = 1280 vh = 720
rga3_reg: wr: ovlp_xoff = 0 ovlp_yoff = 0 format = RGBA8888 rdmode = 0
rga3_reg: mmu: win0 = 00 win1 = 00 wr = 00
rga3_reg: alpha: flag 0 mode0=0 mode1=a0a
rga3_reg: blend mode is no blend
rga3_reg: yuv2rgb mode is 0
```

- reg模式

```
rga2: open rga2 reg!                                 //reg日志开启打印。
rga2: CMD_REG                                        //功能寄存器配置
rga2: 00000000 00000000 00000040 000e1040
rga2: 00119440 00000000 00000500 02cf04ff
rga2: 00000000 00000000 00000000 00000000
rga2: 00000000 00000000 00000000 00000040
rga2: 000e1040 00119440 00000500 02cf04ff
rga2: 00000000 00000000 0000ff00 ffffffff
rga2: 00000007 00000000 00000000 00000101
rga2: 07a80000 00000000 07a800e4 00000000
rga2: CSC_REG                                        //full csc寄存器配置
rga2: 00000000 00000000 00000000 00000000
rga2: 00000000 00000000 00000000 00000000
rga2: 00000000 00000000 00000000 00000000
rga2: CMD_READ_BACK_REG                              //功能寄存器回读值
rga2: 00000000 00000000 00000040 000e1040
rga2: 00119440 00000000 00000500 02cf04ff
rga2: 00000000 00000000 00000000 00000000
rga2: 00000000 00000000 00000000 00000040
rga2: 000e1040 00119440 00000500 02cf04ff
rga2: 00000000 00000000 0000ff00 ffffffff
```

```
rga2: 00000007 00000000 00000000 00000101
rga2: 07a80000 00000000 07a800e4 00000000
rga2: CSC_READ_BACK_REG                                    //full csc寄存器回读值
rga2: 00000000 00000000 00000000 00000000
rga2: 00000000 00000000 00000000 00000000
rga2: 00000000 00000000 00000000 00000000
```

- time模式
  - rga2

```
rga2: sync one cmd end time 2414                           //打印本次工作RGA硬件的耗时，单位为us
```

  - multi-rga

> 1.3.0以下版本

```
rga3_reg: set cmd use time = 196                           //开始处理请求到配置寄存器的耗时，单位为us
rga_job: hw use time = 554                                 //硬件启动到硬件中断返回耗时，单位为us
rga_job: (pid:3197) job done use time = 751                //开始处理请求到请求完成的耗时，单位为us
rga_job: (pid:3197) job clean use time = 933               //开始处理请求到请求资源处理完毕的耗时，单位为us
```

> 1.3.0及以上版本

```
rga_mm: request[3300], get buffer_handle info cost 188 us        //获取当前buffer_handle信息耗时（虚拟地址则包含cache同步的耗时）
rga3_reg: request[3300], generate register cost time 2 us        //生成寄存器配置耗时
rga3_reg: request[3300], set register cost time 301 us           //配置寄存器耗时
rga_job: request[3300], hardware[RGA3_core0] cost time 539 us    //对应的硬件核心完成任务耗时
rga_mm: request[3300], put buffer_handle info cost 153 us        //释放当前buffer_handle信息耗时（虚拟地址则包含cache同步的耗时）
rga_job: request[3300], job done total cost time 1023 us         //当前job从提交到完成返回用户态的全部耗时
rga_job: request[3300], job cleanup total cost time 1030 us      //当前job从提交到资源释放完毕的全部耗时
```

3.2.3.3 版本信息查询

通过以下命令查询当前驱动名称以及驱动版本：

```
/# cat driver_version
RGA multicore Device Driver: v1.2.23
```

3.2.3.4 负载查询

通过以下命令查询RGA负载情况：

```
/# cat load
num of scheduler = 3                                          //当前搭载硬件核心数
================ load ==================
scheduler[0]: rga3_core0
         load = 0%                                            //对应核心负载占比
----------------------------------
scheduler[1]: rga3_core1
         load = 0%
----------------------------------
scheduler[2]: rga2
         load = 0%
----------------------------------
```

3.2.3.5 内存管理器查询

通过以下命令查询内存管理器内内存状态：

```
/# cat mm_session
rga_mm dump:
buffer count = 3                                              //内存管理器内保存的buffer数
量
=================================================================
handle = 34 refcount = 1 mm_flag = 0x2  tgid = 3210      //内存句柄、引用计数、内存标
识、进程号打印
virtual address:
         va = 0xb400007286e1c000, pages = 0x00000000ae081f65, size = 3686400
         iova = 0xffc70000, offset = 0x0, sgt = 0x00000000cc976f9e, size =
3686400, map_core = 0x1
                                                              //内存信息
-----------------------------------------------------------------
handle = 35 refcount = 1 mm_flag = 0x2  tgid = 3210
virtual address:
         va = 0xb400007286a95000, pages = 0x000000002f083efc, size = 3686400
         iova = 0xff8e0000, offset = 0x0, sgt = 0x0000000062bb1297, size =
3686400, map_core = 0x1
-----------------------------------------------------------------
handle = 36 refcount = 1 mm_flag = 0x2  tgid = 3210
virtual address:
         va = 0xb40000728670e000, pages = 0x00000000785fef63, size = 3686400
         iova = 0xff550000, offset = 0x0, sgt = 0x00000000cdd7688d, size =
3686400, map_core = 0x1
-----------------------------------------------------------------
```

3.2.3.6 任务请求查询

通过以下命令任务管理器内任务请求状态：

```
/# cat request_manager
rga internal request dump:
request count = 1                                          //任务管理器内任务请求数量
================================================================
------------------ request: 200073 ------------------
         set cmd num: 1, finish job: 0, failed job: 0, flags = 0x0, ref = 2
                                                           //任务请求完成情况
         cmd dump:                                         //任务请求参数
                 rotate_mode = 0
                 src: y = 25 uv = 0 v = e1000 aw = 1280 ah = 720 vw = 1280 vh =
720
                 src: xoff = 0, yoff = 0, format = 0x0, rd_mode = 1
                 dst: y=26 uv=0 v=e1000 aw=1280 ah=720 vw=1280 vh=720
                 dst: xoff = 0, yoff = 0, format = 0x0, rd_mode = 1
                 mmu: mmu_flag=0 en=0
                 alpha: rop_mode = 0
                 yuv2rgb mode is 0
                 set core = 0, priority = 0, in_fence_fd = -1
```

3.2.3.7 硬件信息查询

通过以下命令查询当前搭载硬件信息：

```
/# cat hardware
==================================
rga3_core0, core 1: version: 3.0.76831                     //搭载核心的硬件版本、支持的
功能选项等参数
input range: 68x2 ~ 8176x8176
output range: 68x2 ~ 8128x8128
scale limit: 1/8 ~ 8
byte_stride_align: 16
max_byte_stride: 32768
csc: RGB2YUV 0xf YUV2RGB 0xf
feature: 0x4
mmu: RK_IOMMU
----------------------------------
rga3_core1, core 2: version: 3.0.76831
input range: 68x2 ~ 8176x8176
output range: 68x2 ~ 8128x8128
scale limit: 1/8 ~ 8
byte_stride_align: 16
max_byte_stride: 32768
csc: RGB2YUV 0xf YUV2RGB 0xf
feature: 0x4
mmu: RK_IOMMU
----------------------------------
rga2, core 4: version: 3.2.63318
input range: 2x2 ~ 8192x8192
```

```
output range: 2x2 ~ 4096x4096
scale limit: 1/16 ~ 16
byte_stride_align: 4
max_byte_stride: 32768
csc: RGB2YUV 0x7 YUV2RGB 0x7
feature: 0x5f
mmu: RGA_MMU
-----------------------------------
```

3.2.3.8 dump运行数据

通过以下命令dump运行数据用于调试，可以通过调试节点配置实现将RGA接下来几帧数据写到指定目录下。没有该节点说明当前kernel不支持内核写入写出数据。

- 设置dump数据路径，使能dump运行数据时将输出到该文件夹下。

```
/# echo /data/rga_image > dump_path
/# dmesg -c
rga_debugger: dump path change to: /data/rga_image
```

- 设置dump数据帧数。

```
/# echo 1 > dump_image
/# dmesg -c
rga_debugger: dump image 1

.... RGA运行 ....

/# dmesg -c
rga_debugger: dump image to:
/data/rga_image/1_core1_src_plane0_virt_addr_w1280_h720_RGBA8888.bin
rga_debugger: dump image to:
/data/rga_image/1_core1_dst_plane0_virt_addr_w1280_h720_RGBA8888.bin

/# ls /data/rga_image/
1_core1_dst_plane0_virt_addr_w1280_h720_RGBA8888.bin
1_core1_src_plane0_virt_addr_w1280_h720_RGBA8888.bin
                                                    //输入（src）、输出（dst）运
行图像数据
```

4. Q & A

本节将较为常见的RGA相关问题以Q&A的形式进行分类介绍，如不在本节内的问题请整理相关日志和初步分析的信息提交至redmine平台交由维护RGA模块的工程师处理。

4.1 性能咨询

Q1.1：RGA效率如何评估？

A1.1： RGA在执行拷贝时，可以通过以下公式进行计算理论耗时（该功能仅支持数据的拷贝评估）：

单次拷贝图像耗时 = 图像宽 × 图像高 / RGA每秒能处理的像素数量

= 图像宽 × 图像高 /（RGA每个时钟周期能够处理的像素数量 × RGA频率）

例如： 一幅1920 × 1080大小的图像用RGA（频率设定为300M）做拷贝的理论耗时是：

RGA1 ： 1920 × 1080 /（1 × 300000000） = 0.006912s

RGA2 ： 1920 × 1080 /（2 × 300000000） = 0.003456s

RGA3 ： 1920 × 1080 /（3 × 300000000） = 0.002304s

而实际的耗时与使用的内存类型是相关的，不同的传入内存类型效率从高到低是：物理地址 > dma_fd > 虚拟地址。

在系统空载时，物理地址的实际耗时约为理论耗时的1.1-1.2倍，使用dma_fd的实际耗时约为理论耗时的1.3-1.5倍，而使用虚拟地址的实际耗时约为理论耗时的1.8-2.1倍，并且受CPU影响较大。通常我们比较建议开发者使用dma_fd作为传入的内存类型，在易获取和效率上得到了较好的平衡，虚拟地址仅用于学习阶段了解RGA时，作为简单易上手的内存类型来使用。

下表为在RK3566上系统空载时不同的RGA频率的实际测试数据。

测试环境：

芯片平台	**RK3566**
RGA硬件版本	RGA2-EHANCE
系统平台	Android 11
RGA频率	300 M
CPU频率	1.8 Ghz
GPU频率	800 M
DDR频率	1056 M

测试数据：

分辨率	内存类型	理论耗时（us）	实际耗时（us）
1280 × 720	GraphicBuffer（cache）	1,536	2,620
1280 × 720	GraphicBuffer（no cache）	1,536	2,050
1280 × 720	Drm buffer（cache）	1,536	2,190
1280 × 720	Physical address（Drm）	1,536	2,000
1920 × 1080	GraphicBuffer（cache）	3,456	5,500
1920 × 1080	GraphicBuffer（no cache）	3,456	4,180
1920 × 1080	Drm buffer（cache）	3,456	4,420
1920 × 1080	Physical address（Drm）	3,456	4,100
3840 × 2160	GraphicBuffer（cache）	13,824	21,500
3840 × 2160	GraphicBuffer（no cache）	13,824	15,850
3840 × 2160	Drm buffer（cache）	13,824	16,800
3840 × 2160	Physical address（Drm）	13,824	15,600

Q1.2：理论公式仅提供拷贝的评估方法，那么其他模式如何评估？

A1.2：目前仅有拷贝的公式可供评估使用，其他模式比如缩放、裁剪，可以使用两张图像较大的分辨率带入拷贝公式进行计算得到的耗时进行评估，通常会根据缩放、裁剪的大小有一定的上下浮动，混合等分辨率没有变化的模式耗时约为拷贝模式耗时的1.1-1.2倍。具体实际场景中由于受到DDR带宽影响，建议实际评估时以在目标场景中的实际测试数据为准。

Q1.3：为什么RGA在一些场景中性能表现很差，与跑demo时耗时最大能到2倍？

A1.3：因为RGA在目前RK平台中的总线优先级为最低档，当带宽资源较为紧张时，例如ISP运行多路的场景中，RGA由于带宽资源紧张，没有办法及时的读写DDR内的数据，产生了较大的延迟，从而表现为RGA的性能下降。

Q1.4：RGA的效率不能满足我们产品的需求，有什么办法可以提升么？

A1.4：部分芯片早期（2021年之前）的出厂固件的RGA频率并不是最高频率，例如3399、1126等芯片RGA的频率最高可以到400M，可以通过以下两种方式实现RGA提频：

- 通过命令设置（临时修改，设备重启则恢复频率）

> 查询RGA频率

```
cat /sys/kernel/debug/clk/clk_summary | grep rga                    //查询rga频率，其中的aclk的频率
```

> 修改RGA频率

```
echo 400000000 > /sys/kernel/debug/clk/aclk_rga/clk_rate          //400000000修改为想要修改的频率
```

- 修改dts实现修改RGA频率（重启后依旧为设置的频率）

> 以下示例为RK3288上修改dts中RGA频率的修改方法，其他平台可以在对应的dts中进行修改

```
diff --git a/arch/arm/boot/dts/rk3288-android.dtsi b/arch/arm/boot/dts/rk3288-android.dtsi
index 02938b0..10a1dc4 100644
--- a/arch/arm/boot/dts/rk3288-android.dtsi
+++ b/arch/arm/boot/dts/rk3288-android.dtsi
@@ -450,6 +450,8 @@
         compatible = "rockchip,rga2";
         clocks = <&cru ACLK_RGA>, <&cru HCLK_RGA>, <&cru SCLK_RGA>;
         clock-names = "aclk_rga", "hclk_rga", "clk_rga";
+        assigned-clocks = <&cru ACLK_RGA>, <&cru SCLK_RGA>;
+        assigned-clock-rates = <300000000>, <300000000>;
         dma-coherent;
  };
```

Q1.5：RGA是否支持通过命令或接口查询当前的RGA硬件利用率（负载）？

A1.5：RGA multicore Device Driver支持查看硬件负载，详情可以参考 **调试说明——驱动调试节点——调试节点功能——负载查询**。

Q1.6：为什么一些场景使用异步模式调用RGA耗时比同步模式还要慢？

A1.6：RGA Device Driver、RGA2 Device Driver 由于目前librga的异步模式的标识符为打开的设备节点，而单例模式的librga一个进程只会打开一个fd，所以imsync()是等待该进程所有的异步模式均运行结束后才会返回。而RGA multicore Device Driver引入了fence机制，所以是针对单次请求的实时处理，不会存在这种问题。

Q1.7：有些场景使用虚拟地址调用RGA做拷贝耗时比memcpy还要高，可有办法优化？

A1.7：通常我们不建议使用虚拟地址调用RGA，因为在CPU负载较高的场景下使用虚拟地址调用RGA的效率会大大下降，这是因为RGA驱动中虚拟地址转换为物理地址页表这一部分是由CPU来计算的，并且本身虚拟地址转换为物理地址页表这个过程本身就很耗时；加之虚拟地址通常没有用户态的接口同步cache，因此驱动内部针对虚拟地址是每一帧都会强制同步cache的。所以通常我们建议使用物理地址或dma_fd来调用librga。

Q1.8：调用RGA时为什么会有较高的CPU负载？

A1.8：调用RGA时除了基础必要的CPU负载外，有以下几种情况会导致增加较高的额外CPU负载：

1). 当使用虚拟地址调用RGA时，虚拟地址本身是CPU的访问地址，要通过当前进程的映射表转换为硬件可识别的离散的物理地址表是通过CPU查询、计算的，因此会引入额外的CPU负载。通常不建议在实际的产品场景中使用虚拟地址调用RGA，更建议使用dma-buf fd来调用RGA，除非业务逻辑上仅存在虚拟地址并不在意这部分CPU负载损耗。

2). 当使用的虚拟地址是cacheable的，由于使能了cache，RGA驱动会在硬件访问内存前后强制同步cache数据，因此会增加CPU同步cache和内存的负载。由于常见的虚拟地址分配器并不是设计用于给其他硬件访问的，并存在同步cache的接口，因此驱动针对虚拟地址强制同步cache也是必要的。

3). 当使用dma-buf fd调用RGA时，有些分配器默认分配的是cacheable的buffer，并且kernel中dma-buf的处理会强制同步cache的情况，这是也会存在每次调用RGA时会有较大的CPU负载，也是因为CPU同步cache和内存引入的负载。该种情况下建议分配禁用cache的dma-buf。

Q1.9：为什么当搭载8G DDR时，RGA效率较于4G时性能下降严重？

A1.9：由于部分RGA1/RGA2的IOMMU仅支持最大32位的物理地址，而RGA Device Driver、RGA2 Device Driver中对于不满足硬件内存要求的调用申请，默认是通过swiotlb机制进行访问访问受限制的内存（原理上相当于通过CPU将高位内存拷贝至复合硬件要求的低位内存中，再交由硬件进行处理，处理完毕后再通过CPU将低位内存搬运回目标的高位内存上。）因此效率十分低下，通常在正常耗时的3-4倍之间浮动，并且引入受CPU负载影响。

RGA Multicore Device Driver中针对访问受限制的内存会禁用swiotlb机制，直接通过调用失败的方式显示的通知调用者申请符合要求的内存再调用，来保证RGA的高效。通常伴随着以下日志：

> HAL层日志：

```
RgaBlit(1483) RGA_BLIT fail: Invalid argument
Failed to call RockChipRga interface, please use 'dmesg' command to view driver
error log.
```

> 驱动日志：

```
rga_policy: invalid function policy                          //标识存在无效的参数，这
里是指没有硬件能够访问当前请求配置的内存。
rga_job: job assign failed                                   //匹配硬件核心失败
rga_job: failed to get scheduler, rga_job_commit(403)
rga_job: (pid:3524) job clean use time = 19
rga_job: request[282567] task[0] job_commit failed.
rga_job: rga request commit failed!
rga: request[282567] submit failed!
```

> 驱动运行日志：

```
rga_policy: start policy on core = 4
[82116.782252] rga_policy: RGA2 only support under 4G memory!
                                                             //标识当前搭载的RGA2核心
仅支持4G以内的内存。
[82116.782256] rga_policy: optional_cores = 0
[82116.782258] rga_policy: invalid function policy
[82116.782260] rga_policy: assign core: -1
[82116.782262] rga_job: job assign failed
```

因此，针对这种场景建议申请4G以内的内存调用librga，常见的分配4G内存方式可以查看以下示例代码：

<librga_souce_path>/samples/allocator_demo/src/rga_allocator_dma32_demo.cpp

<librga_souce_path>/samples/allocator_demo/src/rga_allocator_graphicbuffer_demo.cpp

Q1.10：为什么调用RGA API时发现API返回耗时远高于驱动打印硬件耗时？

Q1.10.1：通过“TIME”运行日志发现map/unmap buffer耗时过大。
Q1.10.2：对比kernel日志时间戳发现打印参数日志到寄存器打印之间存在较大的空白时间。
Q1.10.3：相同的参数配置，仅使用不同的内存分配器得到的运行耗时差异较大。

A1.10：这里的耗时异常的原因均为外部buffer的内存映射行为（map/unmap）导致。所有的外部buffer都需要映射、绑定到RGA驱动中才能保证硬件最终能够访问指定的buffer。而不同的分配器对应的底层实现差异会导致驱动映射、绑定内存时耗时不一，从而导致看起来好像API耗时会比硬件实际耗时高很多的情况。常见的会存在较高额外耗时的dma-buf分配器有ION、V4L2等，通常这些差异与cache的同步有关，针对这类型问题可以通过横向对比不同分配器进行确认。

这类问题通常可以通过以下几种方式进行优化：

1). 使用map/unmap耗时合理的内存分配器，常见的有dma_heap、DRM以及对应的封装内存分配器，以下是对应内存分配器分配内存调用RGA的示例代码：

<librga_souce_path>/samples/allocator_demo/src/rga_allocator_dma_demo.cpp

<librga_souce_path>/samples/allocator_demo/src/rga_allocator_drm_demo.cpp

2). 该问题对应的调用场景为通过wrapbuffer_fd()封装rga_buffer_t或者使用importbuffer_fd后仅运行一帧就立即releasebuffer_handle，这对于临时的测试或者每一帧buffer都是变化的场景是正常的，但本身在实际产品中buffer反复的重新分配这个行为就是性能较差且不合理的，建议整体性的进行优化buffer流程。

通常我们建议整体流程按照以下方式进行设计：

> 1. 构造buffer_pool，分配n个buffer用于作为轮转buffer，n的大小视实际场景进行配置。
> 2. 将这部分buffer 通过importbuffer_fd()导入RGA，获取到RGA的buffer_handle。
> 3. 使用轮转到的buffer_handle调用RGA执行图像操作，反复轮转、循环。
> 4. 当不再需要这个buffer_pool内的buffer时，调用releasebuffer_handle()释放这部分buffer在RGA内部的引用，以保证后续该buffer能够被释放、销毁。
> 5. 释放buffer_pool内不需要的buffer。

按照上述流程设计，那么即使分配器的map/unmap行为会导致异常耗时也被收敛到importbuffer_fd()/releasebuffer_handle()的调用上，对于实际运行时每一帧调用将不再会有影响，这是一种很好的规避由于内存分配器实现差异引入性能差异的方案。

3). 对于无法更改内存分配器以及业务流程的场景，将只能通过修改使用的内存分配器map/unmap流程进行优化耗时，这是十分危险的行为，需要确保自己知晓全部使用该内存分配器的模块的应用行为后，提交redmine咨询对应内存分配器维护者来获取技术支持。

Q1.11：为什么importbuffer_fd()/importbuffer_virtualaddr()调用耗时很高，为什么要调用该API？

A1.11：该接口相关用法以及说明可以查看源码目录下docs文件夹内的《Rockchip_Developer_Guide_RGA_CN》中 “概述” 章节——“图像缓冲区预处理” 了解用法说明。importbuffer_xx()的作用是将外部的buffer导入到RGA驱动内，使后续每一帧RGA调用都可以通过buffer_handle快速的访问该buffer，而导入外部buffer是比较耗时的操作，需要将外部的buffer映射到RGA驱动内，并保存对应的物理地址以及buffer信息，这对于调用RGA来说是不可缺少的行为。

Q1.12：RGA支持并行的操作么？为什么多线程调用RGA时会出现个别帧耗时增多、翻倍的情况？

A1.12：RGA API是可以支持多线程/进程并行调用的，但实际硬件上是否并行执行图像操作取决于当前使用芯片搭载的RGA核心数量，即搭载的核心数量则为最大支持的并行任务数量，超过核心数量的任务则会进入等待状态，直到有核心进入空闲状态。因此当并行调用的数量超过了硬件最大支持的并行数量后，那么个别帧的调用将会增加等待硬件空闲的耗时。具体可以通过以下调试节点（具体说明可以查看“驱动调试节点”小节中“硬件信息查询”部分）获取当前芯片搭载的核心数量以及支持的功能：

```
/# cat hardware
```

4.2 功能咨询

Q2.1：如何知道我当前的芯片平台搭载的RGA版本以及可以实现的功能？

A2.1：可以查看源码目录下docs文件夹内的《Rockchip_Developer_Guide_RGA_CN》中 “概述” 章节了解RGA的版本以及支持信息。

不同系统的源码路径会有所差异，librga源码目录路径在不同SDK的路径如下：

Android 7.0即以上SDK ：

hardware/rockchip/librga

Android 7.0以下SDK：

hardware/rk29/librga

Linux SDK：

external/linux-rga

Q2.2：如何调用RGA实现硬件加速？可有demo可供参考？

A2.2：1). API调用接口可以查询docs目录下《Rockchip_Developer_Guide_RGA_CN》中 “应用接口说明” 章节。

2). 演示demo位于samples目录下rga_im2d_demo，该演示demo内部实现了RGA大部分的接口，通过命令配置实现对应的RGA功能，亦可作为一些场景下测试RGA是否正常的工具。建议初次了解RGA的开发者初期可以直接运行demo并查看结果，从而了解RGA的实际功能，再根据自己的需求在demo中修改参数实现对应功能，最终再尝试单独在自己的工程中调用RGA API。

3). 常见应用常见的示例代码在samples目录下：

├── **allocator_demo**：内存分配器相关示例代码

├── **alpha_demo**：alpha混合、叠加相关示例代码

├── **async_demo**：异步模式相关示例代码

├── **config_demo**：线程全局配置相关示例代码

├── **copy_demo**：图像搬运、拷贝相关示例代码

├── **crop_demo**：图像裁剪、拼接相关示例代码

├── **cvtcolor_demo**：图像格式转换、色域转换相关示例代码

├── **fill_demo**：图像填充、画框相关示例代码

├── **mosaic_demo**：马赛克遮盖相关示例代码

├── **padding_demo**：padding相关示例代码

├── **resize_demo**：图像缩放相关示例代码

├── **rop_demo**：ROP运算相关示例代码

└── **transform_demo**：图像变换相关示例代码

Q2.3：RGA的支持信息？

Q2.3.1：RGA支持哪些格式？

A2.3.1：具体支持情况可以查看《Rockchip_Developer_Guide_RGA_CN》中“概述”——“图像格式支持”小节中查询对应的芯片版本搭载的RGA的格式支持情况，也可以在代码中调用 **querystring(RGA_INPUT_FORMAT | RGA_OUTPUT_FORMAT);** 接口查询当前硬件的输入输出格式支持情况。

Q2.3.2：RGA支持的缩放倍率是多少？

A2.3.2：具体支持情况可以查看《Rockchip_Developer_Guide_RGA_CN》中“概述”——“设计指标”小节中查询对应的芯片版本搭载的RGA支持的缩放倍率，也可以在代码中调用 **querystring(RGA_SCALE_LIMIT);** 接口查询当前硬件的支持的缩放倍率。

Q2.3.3：RGA支持的最大分辨率是多少？

A2.3.3：具体支持情况可以查看《Rockchip_Developer_Guide_RGA_CN》中“概述”——“设计指标”小节中查询对应的芯片版本搭载的RGA支持的最大输入输出分辨率，也可以在代码中调用 **querystring(RGA_MAX_INPUT | RGA_MAX_OUTPUT);** 接口查询当前硬件的支持的最大输入输出分辨率。

Q2.3.4：RGA对不同的格式对齐要求是什么？

A2.3.4：具体支持情况可以查看《Rockchip_Developer_Guide_RGA_CN》中“概述”——“图像格式对齐说明”小节中查询对应的芯片版本搭载的RGA对不同格式的对齐要求。

A2.3：总体来说，对于RGA的支持有疑问可以查看《Rockchip_Developer_Guide_RGA_CN》，其中对于RGA的支持信息会有较详细的介绍。

Q2.4：多个版本的librga有何差异？又该如何分辨？

A2.4：目前的RK平台所有发布SDK中，主要分配无法获取版本号的旧版本librga，支持查询版本号的新版本librga。

无法获取版本的旧版本librga目前已经停止支持与维护，主要的表征点为2020年11月前发布的SDK中，搭载的均为旧版本librga，部分芯片平台例如RK3399 Linux SDK 2021年6月前发布的SDK（V2.5即以下）亦为旧版本librga，该版本librga无法完美契合较新的驱动，可能会出现颜色偏差、格式异常等问题，不建议混合使用，如果有需要使用到较新内核时建议更新新版本librga，反之使用到新版本librga亦然，需要更新内核至匹配。

支持查询版本号新版本librga是目前主要支持与维护的版本，主要表征点为源码目录下增加**im2d_api**目录，该版本集成与旧版本librga，并推出简单易用的IM2D API，亦可称呼为IM2D版librga。新版本librga不仅支持新的IM2D API，旧版本的RockchipRga接口和C_XXX接口也是支持的。具体的API调用说明可以查看《Rockchip_Developer_Guide_RGA_CN》了解。

通常对于一些新旧版本librga功能支持情况一般优先建议更新整体SDK避免出现依赖问题，强烈不建议新版本librga搭配旧驱动或者旧版本librga搭配新内核使用，部分场景会有较明显的错误。

Q2.5：RGA是否有对齐限制？

A2.5：不同的格式对齐要求不同，RGA硬件本身是对图像每行的数据是按照字（world）对齐的方式进行取数的，即4个字节32个bit。例如RGBA格式本身单个像素存储大小为32（4 × 8）bit，所以没有对齐要求；RGB565格式存储大小为16（5 + 6 +5）bit，所以需要2对齐；RGB888格式存储大小为24（8 × 3）bit，所以该格式需要4对齐才能满足RGA硬件的32bit取数要求；YUV格式存储相对较为特殊，本身排列要求需要2对齐，Y通道单像素存储大小为8bit，UV通道根据420/422决定每四个像素的存储大小，所以YUV格式Y通道需要4对齐才能满足RGA的硬件取数要求，则YUV格式需要4对齐；其他的未提及的格式对齐要求原理相通。注意，该题中对齐均指width stride的对齐要求，YUV格式本身实际宽高、偏移量由于格式本身特性也是要求2对齐的。具体对齐限制可以查看《Rockchip_Developer_Guide_RGA_CN》中“概述”——“图像格式对齐说明”小节。

Q2.6：RGA能否支持一次绘制多个矩形区域，或执行多次操作？RGA的工作原理？

A2.6：RGA 在硬件上只能顺序工作即配置的一个任务工作结束和进行下一个配置的工作。因此不能一次绘制多个矩形区域，可以通过 async 模式把需要 RGA 做的工作往底层驱动配置，RGA 会将工作存储在驱动自己管理的一个工作队列中按顺序完成。当上层需要处理这块 buffer 时再调用 **imsync()** 来确定 RGA 硬件是否已经完成工作。

在librga 1.9.0版本后，增加尾缀为array的接口，支持配置多个矩形区域进行划线、画框、填充矩形等操作，例如imfillArray、imrectangleArray，详细可以查看《Rockchip_Developer_Guide_RGA_CN》中“应用接口”——“图像颜色填充、边框绘制”小节。

Q2.7：RGA的fill功能可否支持YUV格式？

A2.7：旧版本的librga是不支持的，只有新版本的librga在包含以下提交以后的librga版本是支持的。如若没有该提交请尝试更新SDK至最新版。

```
commit 8c526a6bb9d0e43b293b885245bb53a3fa8ed7f9
Author: Yu Qiaowei <cerf.yu@rock-chips.com>
Date:   Wed Dec 23 10:57:28 2020 +0800

    Color fill supports YUV format as input source.

    Signed-off-by: Yu Qiaowei <cerf.yu@rock-chips.com>
    Change-Id: I0073c31d770da513f81b9b64e4c27fee2650f30b
```

该功能与RGB颜色填充调用一致，通过配置需要填充色彩的RGB值填充色彩，不同的是输出结果可以设置为YUV格式。

Q2.8：RGA支持YUYV格式么？

A2.8：旧版本的librga（此处指2020年10月份前发布的SDK中的librga）是不支持的，只有新版本的librga（源码目录下有 **im2d_api** 目录的版本）在包含以下提交以后的librga版本是支持的。如若没有该提交请尝试更新SDK至最新版。

```
commit db278db815d147c0ff7a80faae0ea795ceffd341
Author: Yu Qiaowei <cerf.yu@rock-chips.com>
Date:   Tue Nov 24 19:50:17 2020 +0800

    Add support for Y4/YUV400/YUYV in imcheck().

    Signed-off-by: Yu Qiaowei <cerf.yu@rock-chips.com>
    Change-Id: I3cfea7c8bb331b65b5bc741956da47924eeda6e1
```

Q2.9：RGA支持灰度图输入输出做缩放么？

A2.9：旧版本的librga（此处指2020年10月份前发布的SDK中的librga）是不支持的，只有新版本的librga（源码目录下有 **im2d_api** 目录的版本）1.2.2版本才支持灰度图输入。如若librga版本低于该版本请尝试更新SDK至最新版。由于RGA硬件本身不支持灰度图格式，这里灰度图使用的格式是 **RK_FORMAT_Y400** ，表征为没有UV通道的YUV格式，仅有Y通道的YUV便是256阶的灰度图。

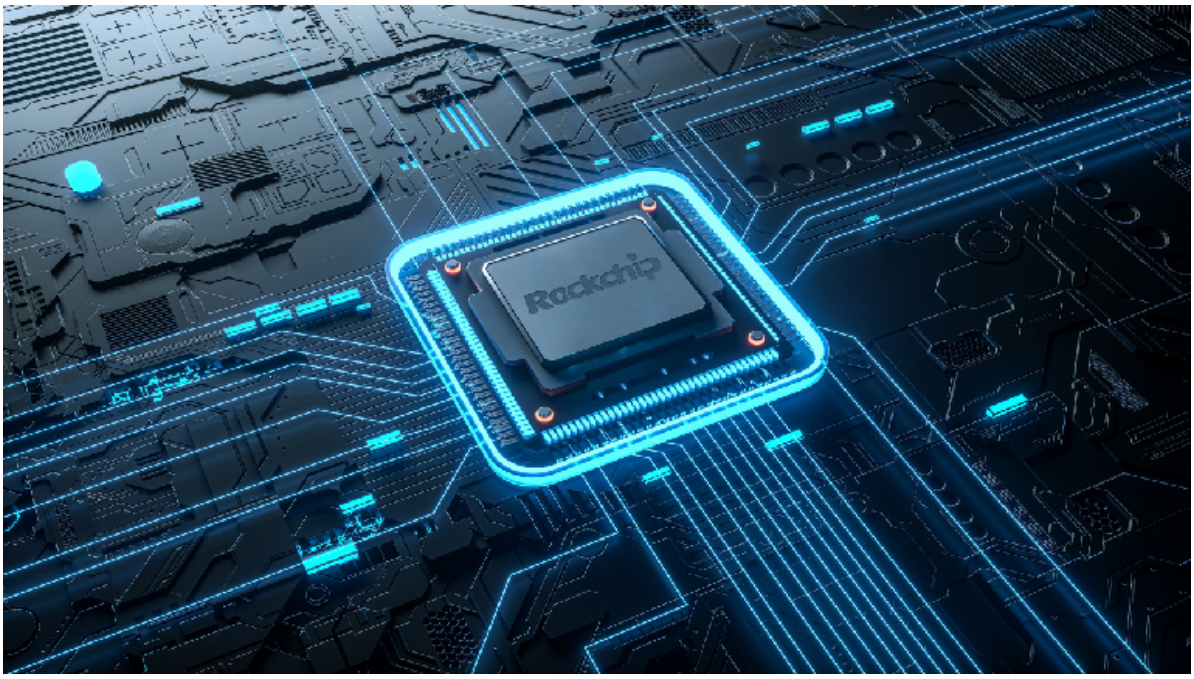
由于是YUV格式，这里需要注意色域空间的问题，librga中CSC 转YUV格式时默认为BT.601 limit range，而limit range的Y通道并不是0~255，涉及到CSC转换（RGB转YUV）输出为Y400格式时，需要注意色域空间的转换时配置full range的标识。

Q2.10：为什么RK3399上ROP的代码放到RV1126上执行却没有对应的效果？

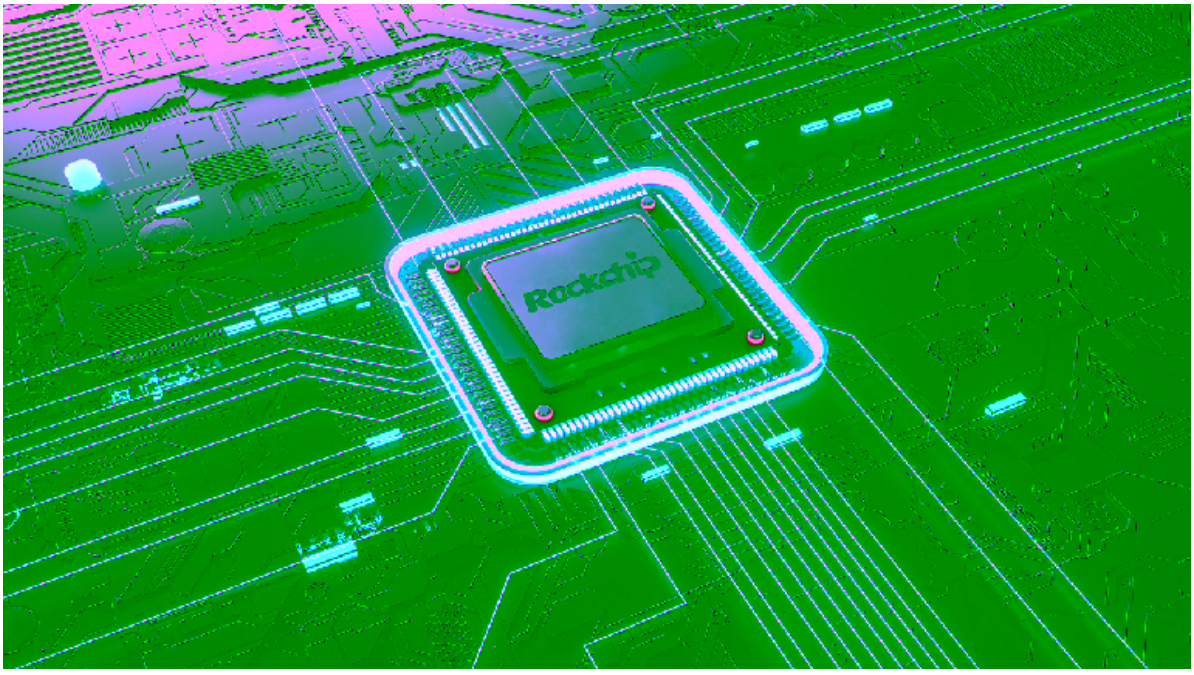
A2.10：虽然RK3399和RV1126上搭载的RGA均为RGA2-ENHANCE，但是他们的小版本是不同的，ROP功能在RV1126上被裁剪掉了，具体功能支持情况可以查看《Rockchip_Developer_Guide_RGA_CN》或者在代码中调用 **querystring(RGA_FEATURE)** 接口实现查询支持功能。

Q2.11：使用RGA其他功能正常，仅在RGB与YUV格式转换时出现严重色差（偏粉偏绿）是什么原因？

预期：



结果：

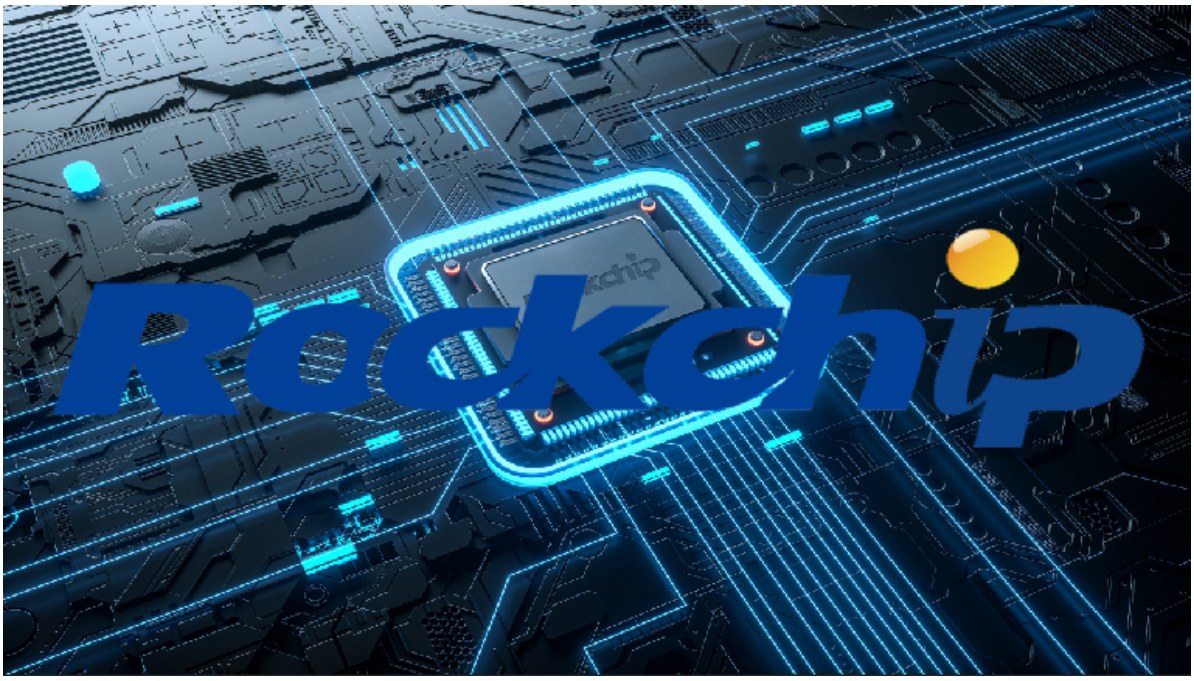


A2.11：该现象通常是由于librga与内核不匹配导致，详细版本说明可以查看 **A2.4** 。该问题通常是在一些2020年11月前发布的SDK中使用了github上获取的librga之后出现该现象。github上更新librga为新版本librga，与较旧版本的RGA驱动是不匹配的，这里一些关于色域空间的配置有发生改变，所以会出现较明显的色偏现象。

该问题的解决方案有两种，一为更新SDK或RGA驱动，保持librga与驱动是匹配的即可，第二种则是如若无需新版本librga才有的功能，可以使用SDK自带的librga即可。

Q2.12：RGA如何实现OSD叠加字幕？

预期：



A2.12：如果输出结果为RGB格式，可以通过 **imblend()** 接口实现，通常选择src over模式，将src通道的图像叠加在dst通道的图像上；如果输出结果为YUV格式，可以通过 **imcomposite()** 接口实现，通常选择dst over‘模式，将src1通道的图像叠加在src通道的图像上，再输出到dst通道。

该功能的叠加原理为 **Porter-Duff混合模型** ，详细可以查看《Rockchip_Developer_Guide_RGA_CN》中 “应用接口说明” —— “图像合成” 小节。

RGA针对不同输出格式，需要不同的配置的原因是，RGA2拥有3个图像通道——src、src1/pat、dst。其中src通道支持YUV2RGB转换，src1/pat和dst通道只支持RGB2YUV转换，而RGA内部的叠加均需要在RGB格式下进行，所以为了保证RGB图像叠加在YUV图像上，必须src作为叠加的背景图像YUV，src1作为叠加的前景图像RGB，最终由dst通道将混合后的RGB图像转换为YUV格式输出。

可以查看示例代码：

<librga_souce_path>/samples/alpha_demo/src/rga_alpha_osd_demo.cpp

<librga_souce_path>/samples/alpha_demo/src/rga_alpha_yuv_demo.cpp

Q2.13：为什么调用RGA实现YUV格式与RGB格式相互转换输出有亮度或者数值差异？

A2.13：该现象原因大致可分为两种：

1). YUV与RGB互转配置相同时，部分像素数值会有轻微差异（通常相差为1），这是由于RGA硬件实现CSC功能时公式的精度问题导致，RGA1和RGA2的CSC公式的小数位精度均为8bit，RGA3的CSC公式的小数位精度为10bit。这里由于精度会导致一些运算结果四舍五入后会有±1的误差。

2). 当RGB2YUV和YUV2RGB转换时配置的CSC模式不同导致，新版本librga中默认的RGB2YUV、YUV2RGB的CSC模式为BT.601-limit _range，当错误的配置了对应的 **color_space_mode** 成员变量时，色域空间的配置不同，便会导致相互转换时产生较大的变化。而旧版本librga中RGB2YUV默认为BT.601-full_range,YUV2RGB默认为BT.709-limit_range，由于两种转换的色域空间配置不同，所以互转会存在较大的变化。

Q2.14：librga中如何配置格式转换时的色域空间呢？

A2.14：两个版本的librga都是支持配置格式转换时的色域空间的。

1). 新版本librga中，可以参考《Rockchip_Developer_Guide_RGA_CN》中 “应用接口说明” —— “图像格式转换” 小节中介绍，重点配置mode参数即可。

2). 旧版本librga中，需要修改librga源码，Normal/NormaRga.cpp中yuvToRgbMode的值，对应的参数如下：

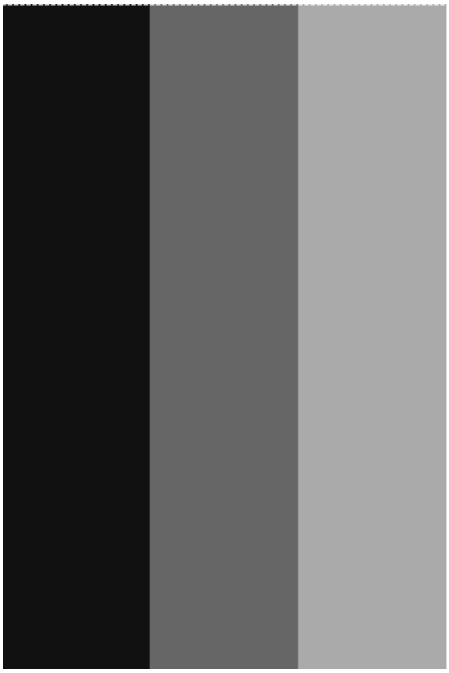
转换格式	色域空间	参数
YUV2RGB	BT.601-limit_range	yuvToRgbMode = 0x1 << 0;
YUV2RGB	BT.601-full_range	yuvToRgbMode = 0x2 << 0;
YUV2RGB	BT.709-limit_range	yuvToRgbMode = 0x3 << 0;
RGB2YUV	BT.601-limit_range	yuvToRgbMode = 0x2 << 4;
RGB2YUV	BT.601-full_range	yuvToRgbMode = 0x1 << 4;
RGB2YUV	BT.709-limit_range	yuvToRgbMode = 0x3 << 4;

Q2.15：调用RGA执行alpha叠加，为什么没有效果？

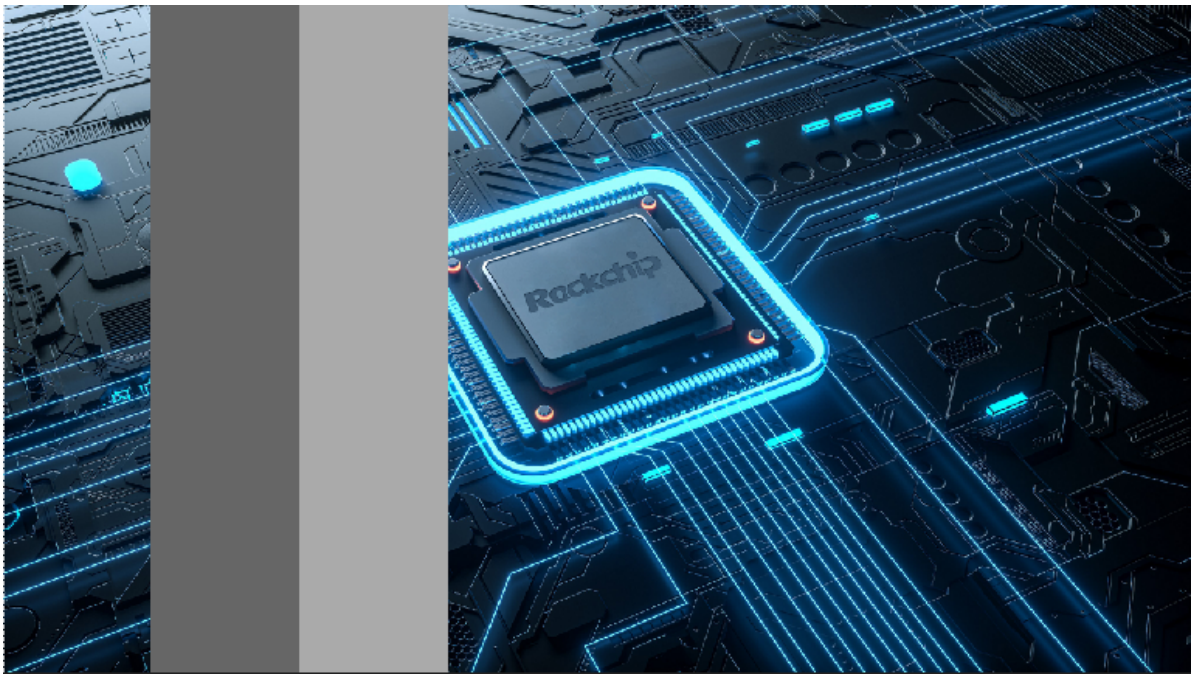
A2.15：检查输入的两张图像的alpha值是否皆为0xFF，当叠加中的前景图像的alpha值为0xFF时，其结果便是前景图像直接覆盖在背景图像上，看起来的结果看着像是没有效果一般，实际上是正常的结果。

Q2.16：调用RGA执行alpha叠加，前景图像的alpha值为0x0，为什么结果不是全透？

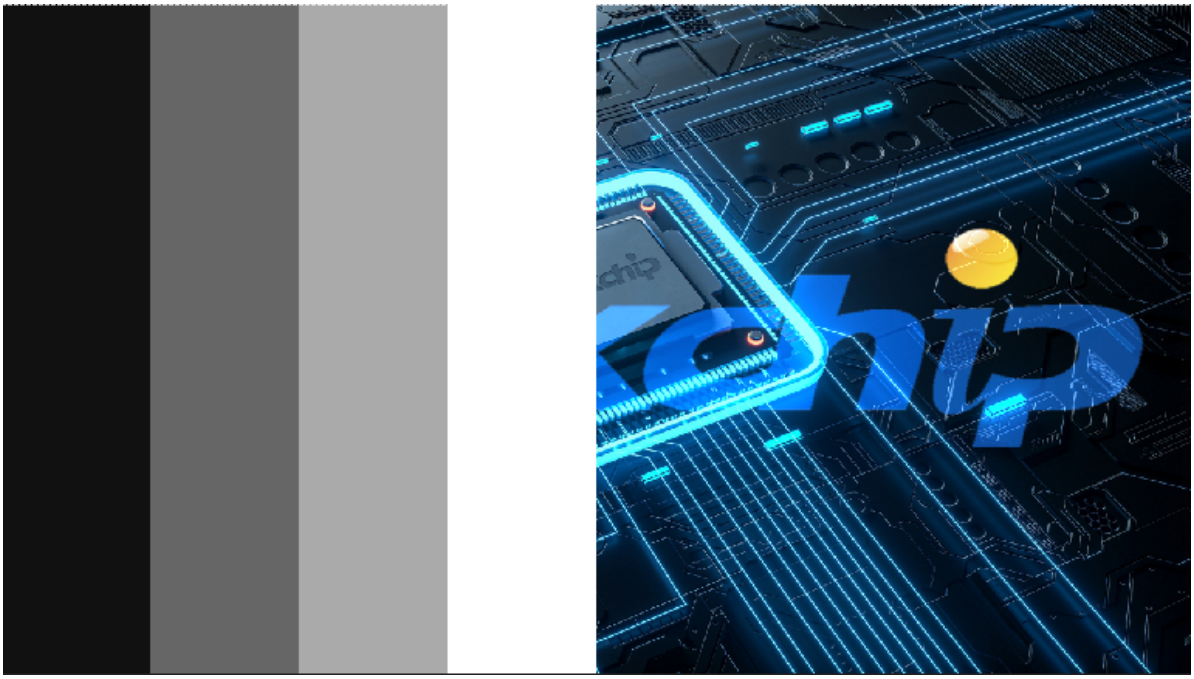
前景图：（黑白和rockchip alpha为0x00）



预期：



结果：



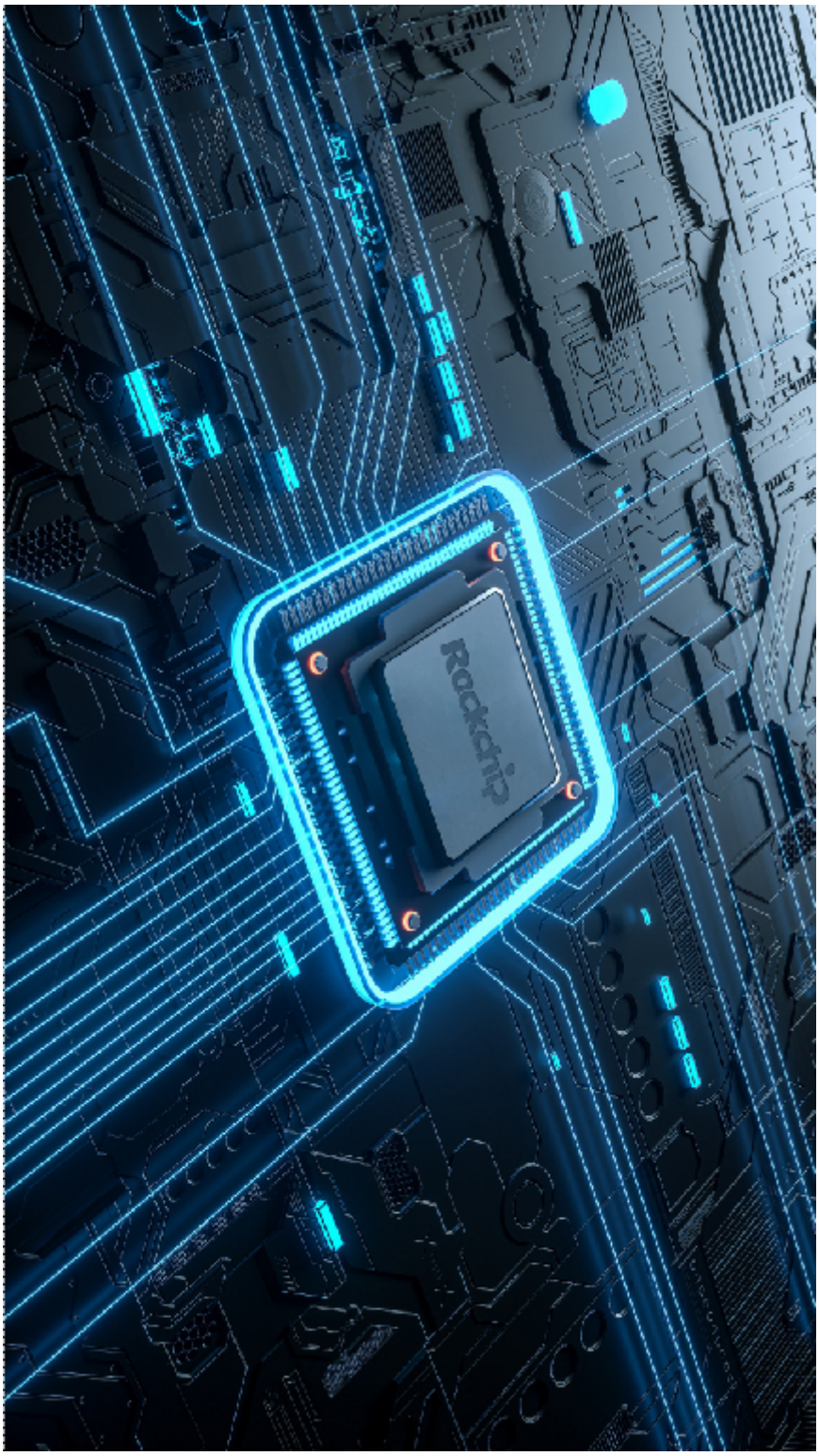
A2.16：我们正常配置的模式是默认颜色值已经预乘过对应的alpha值的结果，而直接读取的原始图片的颜色值并没有预乘过alpha值，所以需要在调用imblend时额外的增加标志位来标识本次处理中的图像颜色值没有需要预乘alpha值。具体调用方式可以查看《Rockchip_Developer_Guide_RGA_CN》中 “应用接口说明” —— “图像合成“ 小节。

Q2.17：IM2D API可以一次RGA调用实现多种功能么？

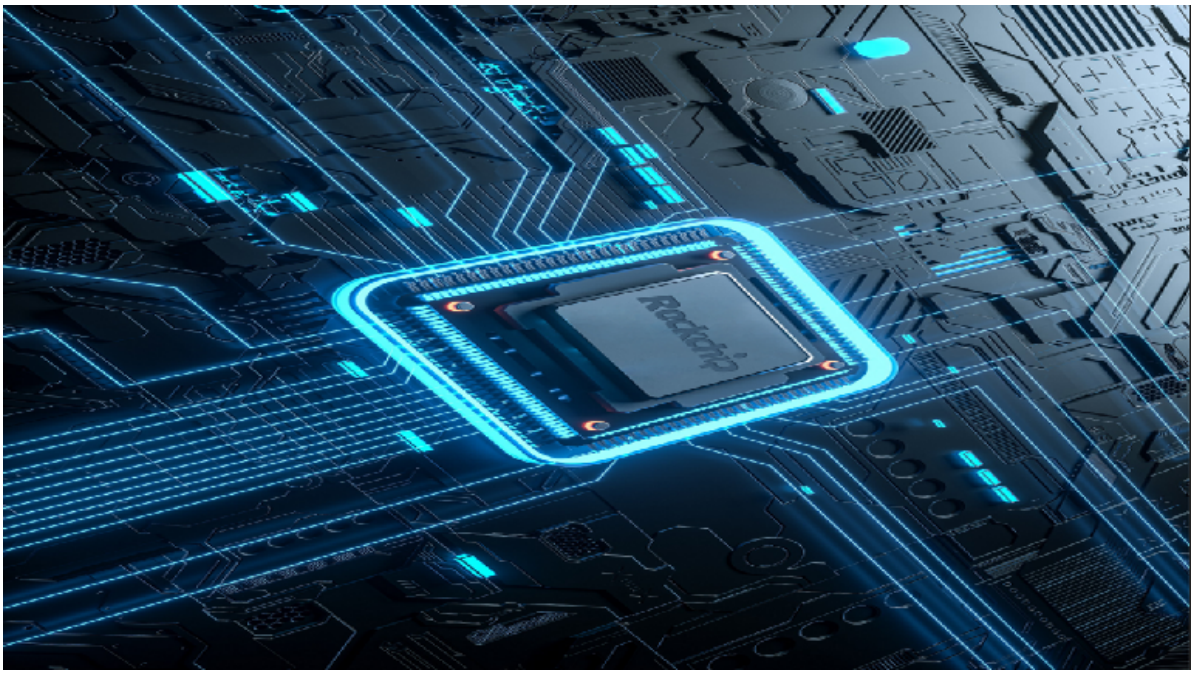
A2.17：可以的，详细可以查看《Rockchip_Developer_Guide_RGA_CN》中 “应用接口说明” —— “图像处理” 小节，并参考IM2D API其他接口的实现，了解 **improcess()** 的用法。

Q2.18：调用RGA执行图像旋转时，结果图像被拉伸？

预期：



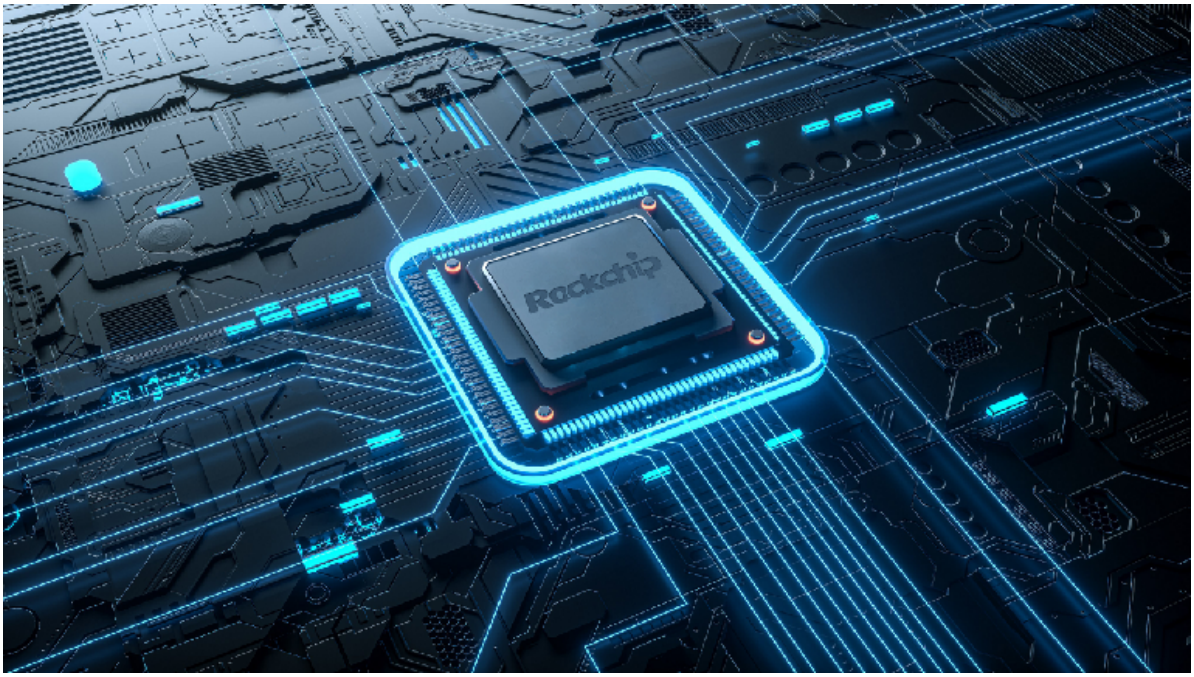
结果：



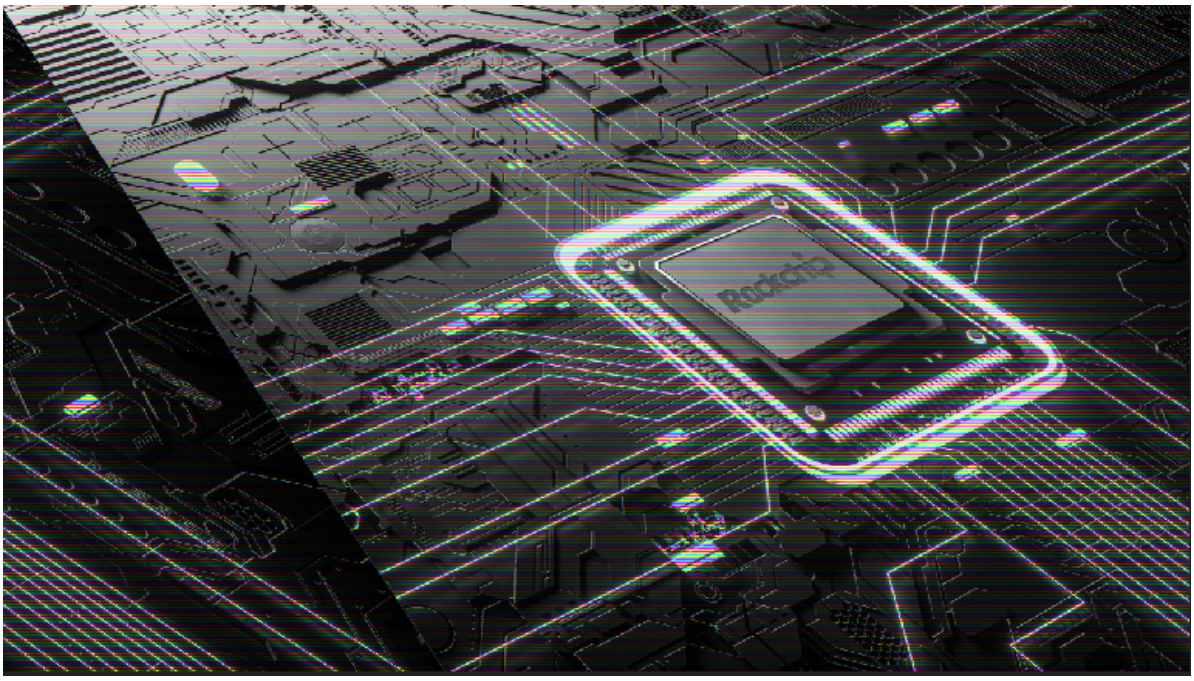
A2.18：在旋转90°、270°时，如果不希望RGA执行缩放，应将图像的宽、高交换，否则RGA驱动默认该行为为旋转＋缩放的行为去执行工作，结果表现便是拉伸的效果了。

Q2.19：RGB888输出缩放后结果显示图像是斜的，并且有黑线？

原图（1920 × 1080）：



结果（1282 × 720）：

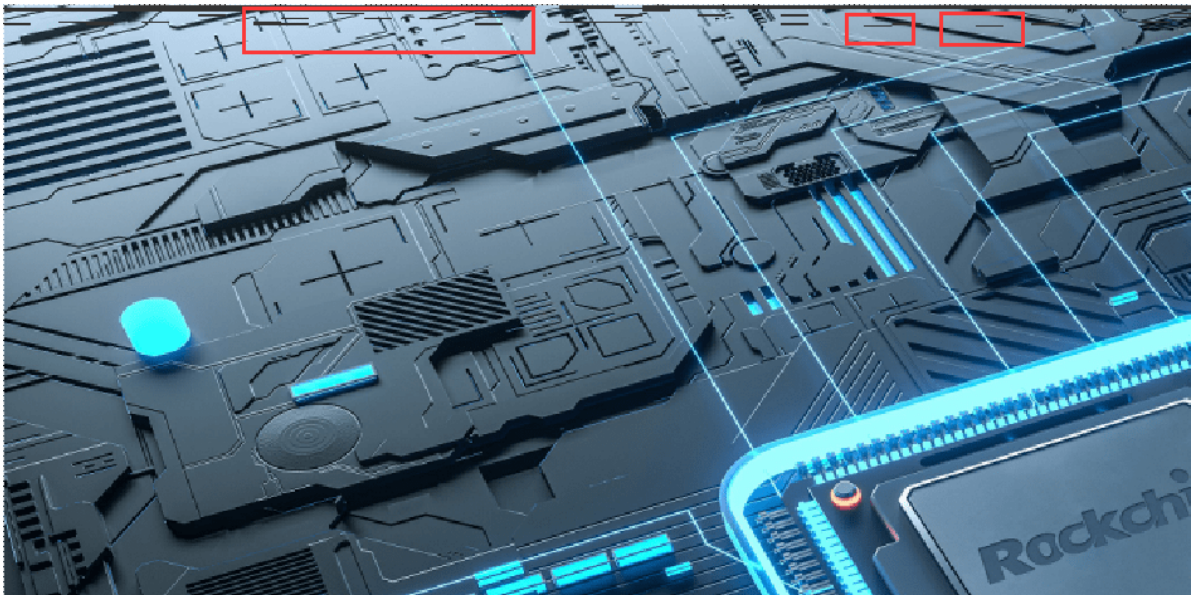


A2.19：该问题是对齐限制导致的，RGB888格式的虚宽需要4对齐，请检查配置的图像参数，对齐限制可以参考 **Q2.5** 的回答。

Q2.20：在一些系统流程中调用RGA输出的结果是花的，这是什么原因导致的？

A2.20：通常RGA的异常不会出现图像花掉的现象，一般遇到这种问题需要先定位问题是否是RGA出现的问题，在一些系统流程中需要先确认输入RGA的源数据是否已经是异常的，可以通过在调用RGA前将内存里的数据调用 **fwrite()** 写文件出来，查看源数据是否正常。写文件的方法如果不太熟悉，可以参考源码目录下 **core/RgaUtils.cpp** 中的 **output_buf_data_to_file()** 函数的实现部分。

Q2.21：调用RGA处理图像后出现黑色或绿色的小条纹，这是什么原因？



A2.21：这是使用非虚拟地址调用时，buffer使能了cache，并且在CPU操作前后没有同步cache导致的。如果不了解如何同步cache可以参考samples/allocator_demo/src/rga_allocator_dma_cache_demo.cpp中的用法。

Q2.22：在RK3588上出现同一显示区域使用RGA缩放后画面抖动，这是什么原因导致的？

A2.22：由于RK3588比较特殊，搭载有两种RGA核心（一颗RGA2，两颗RGA3），它们在缩放算法上存在一些取数行为的差异导致结果会出现整体左上移/右下移的现象，因为在这种对显示效果有要求的场景上，建议指定核心的方式来避免出现算法差异引入的抖动问题。

指定核心可以参考以下示例代码：

<librga_souce_path>/samples/config_demo/src/rga_config_single_core_demo.cpp

<librga_souce_path>/samples/config_demo/src/rga_config_thread_core_demo.cpp

4.3 HAL层报错

4.3.1 IM2D_API报错

Q3.1.1：imcheck()返回报错，该如何处理？

```
check error! Invalid parameters: dst, Error yuv not align to 2, rect[x,y,w,h] =
[0, 0, 1281, 720], wstride = 1281, hstride = 720, format = 0xa00(nv12)
output support format : RGBA_8888 RGB_888 RGB_565 RGBA_4444 RGBA_5551
YUV420/YUV422 YUV420_10bit/YUV422_10bit YUYV420 YUYV422 YUV400/Y4
```

A3.1.1：imcheck()接口作为调用librga的校验接口，它将判断即将传递到librga内部的数据结构的参数是否正确、功能是否支持、是否触发硬件限制等，可以将imcheck()的返回报错值作为传参传入 **IMStrError()** 返回的字符串则为详细的报错信息，可以根据错误提示确认哪些条件限制被触发，或是参数错误。

如问题中报错，则为YUV格式对齐的限制问题，这里图像的宽1281不是2对齐的，所以校验失败。

Q3.1.2：imstrError()错误提示没有具体参数打印说明是什么问题？

```
Fatal error: Failed to call RockChipRga interface, please use 'dmesg' command to
view driver error log.
```

A3.1.2：说明配置在im2d api校验已经通过并配置到后级驱动上，可以通过dmesg的方式查看驱动的报错。

4.3.2 RockchipRga接口报错

Q3.2.1：“Try to use uninit rgaCtx=(nil)”报错如何处理？

A3.2.1：1). 该报错为调用到的接口发现librga模块并没有得到初始化，所返回报错。目前版本中该报错通常是由于一些较旧的调用RGA的代码中依旧使用RgaInit/RgaDeInit/c_RkRgaInit/c_RkRgaDeInit接口自行管理RGA模块的初始化，而目前的版本接口使用的单例模式，当被异常DeInit后，便会出现该报错，只需要移除调用代码中的Init/DeInit相关的调用即可。

2). 当驱动没有probe成功，或者驱动设备节点（/dev/rga）访问受限制时也会产生这样的报错。

Q3.2.2：“RgaBlit(1027) RGA_BLIT fail: ”、“RGA_COLORFILL(1027) RGA_BLIT fail: ”标头的报错是什么原因？

A3.2.2：出现该标头报错说明当前RGA任务在驱动运行失败返回，具体原因需要通过dmesg查看驱动日志。

Q3.2.2.1：“RgaBlit(1027) RGA_BLIT fail: Not a typewriter”

A3.2.2.1：该报错通常为参数错误导致，建议检查一下缩放倍数、虚宽是否小于实宽与对应方向的偏移的和、对齐是否符合要求。建议新开发项目使用IM2D API，拥有更全面的检测报错机制，方便开发者节省大量的调试时间。

Q3.2.2.2：“RgaBlit(1349) RGA_BLIT fail: Bad file descriptor”

A3.2.2.2：该报错为ioctl报错，标识当前传入的设备节点的fd无效，请尝试更新librga或确认RGA的初始化流程是否有被修改。

Q3.2.2.3：“RgaBlit(1360) RGA_BLIT fail: Bad address”

A3.2.2.4：该报错通常为传入内核的src/src1/dst通道的内存地址存在问题导致（常见为越界），可以参照本文档 “日志获取与说明” —— “驱动调试节点” 小节，开启驱动日志，并定位出错的内存。

Q3.2.2.4：“RgaBlit(1466) RGA BIIT fail: Invalid argument”

A3.2.2.4：该报错为传入参数不满足当前芯片搭载核心功能、限制要求时上报的无效参数报错，建议检查当前配置的任务参数是否满足当前芯片搭载RGA核心的要求。

Q3.2.3：日志报错“err ws[100,1280,1280]”、”Error srcRect“ 是什么错误？

A3.2.3：该报错为明显的参数报错，“err ws” 即虚宽（width stride）参数异常，其后“[]”内的参数分别为[x_offeset, width, width_stride]，这里由于X方向的偏移与实际操作区域的宽的和大于了虚宽，所以librga认为虚宽存在问题而返回的报错。这里只要将虚宽改为1380或将实宽（width）改为1180，即可。

通常该类型报错后logcat中会打印对应的一些参数：

```
E librga  : err ws[100,1280,1280]                                          //标识单签虚宽存在问题
E librga  : [RgaBlit,731]Error srcRect                                     //标识是src通道报错
E rockchiprga: fd-vir-phy-hnd-format[0, 0xb400006eb6ea9040, 0x0, 0x0, 0]   //对应src通道的输入地址（fd、虚拟地址、物理地址、handle）。
E rockchiprga: rect[100, 0, 1280, 720, 1280, 720, 1, 0]                    //对应src通道的图像参数依次为：x方向偏移、y方向偏移、实际操作区域的宽、实际操作区域的高、图像宽（虚高）、图像高（虚高）、图像格式、size（目前没有使用到的参数）。
E rockchiprga: f-blend-size-rotation-col-log-mmu[0, 0, 0, 0, 0, 0, 1]      //标识着本次调用中的模式配置。
E rockchiprga: fd-vir-phy-hnd-format[0, 0xb400006eb2ea6040, 0x0, 0x0, 0]   //对应dst通道的参数
E rockchiprga: rect[0, 0, 1920, 1080, 1920, 1080, 1, 0]
E rockchiprga: f-blend-size-rotation-col-log-mmu[0, 0, 0, 0, 0, 0, 1]
E rockchiprga: This output the user parameters when rga call blit fail    //报错信息
```

4.4 kernel层报错

Q4.1：“RGA2 failed to get pte, result = -14, pageCount = 112”、“RGA2 failed to get vma, result = 32769, pageCount = 65537”报错是什么导致的？

A4.1：该报错通常为使用虚拟地址调用RGA时，虚拟地址的实际内存小于实际需要的内存大小（即根据图像参数计算出当前通道的图像需要多大的内存），只需检查buffer的大小即可，在一些申请和调用不是在同一处的场景下，可以在调用RGA前执行一遍memset对应图像的大小，确认是否为内存大小不足导致的问题。

改报错后，通常便随着 “rga2 map src0 memory failed” 可以确认是哪一个通道的内存出现问题，如该例中所示，src通道由于实际申请的buffer大小仅为图像所需大小的一半，所以触发了这个报错。

Q4.2：”rga2_reg_init, [868] set mmu info error“ MMU报错是什么原因？

A4.2：该报错表征为fd/虚拟地址转换为物理地址页表出错，通常是申请的内存大小的问题，与Q4.1相同。

Q4.3：“rga：dma_buf_get fail fd[328]” 报这种错误，一般是指buffer出现了什么异常？

Q4.3：该报错为fd在内核经过dma的接口时的报错，建议检查一下申请fd的流程，并在librga外部验证fd可用后再用于调用RGA。

Q4.4：“RGA2 failed to get pte, result = -14, pageCount = 112”、”rga2_reg_init, [868] set mmu info error“ 按照 **Q4.1** 、**Q4.2** 方式检查后，还是一样的报错，这里使用的是DRM分配的物理地址，通过mmap映射的虚拟地址传入RGA的，memset均正常，这是什么原因导致的？

A4.4：该问题为分配器DRM本身的问题，DRM本身认为当用户态获取到物理地址后，正常来讲内核态是不需要虚拟地址的了，所以在分配buffer时就会将对应的kmap释放，仅释放kmap也不会影响到用户态中映射虚拟地址和使用，但是当这块buffer用户态的虚拟地址传入RGA驱动，驱动进行物理地址页表的转换查询时，由于该buffer的kmap已经被释放，或是无法查询到对应的页表项，或是直接访问到错误的地址导致内核crash。

针对这种场景，DRM提供了一个接口标志位，用户判断用户态是否希望DRM释放kmap，即是否考虑讲映射的虚拟地址传入内核使用：

```
(1) drm buffer申请选项增加ROCKCHIP_BO_ALLOC_KMAP定义。
+   /* keep kmap for cma buffer or alloc kmap for other type memory */
+   ROCKCHIP_BO_ALLOC_KMAP  = 1 << 4,
(2) 申请drm内存时，增加新增的drm buffer选项ROCKCHIP_BO_ALLOC_KMAP。
    struct drm_mode_create_dumb arg;
    ...
-   arg.flags = ROCKCHIP_BO_CONTIG;
+   arg.flags = ROCKCHIP_BO_CONTIG | ROCKCHIP_BO_ALLOC_KMAP;
//ROCKCHIP_BO_ALLOC_KMAP仅与ROCKCHIP_BO_CONTIG共同使用时有效。
    ret = drmIoctl(drm_fd, DRM_IOCTL_MODE_CREATE_DUMB, &arg);
```

并确认kernel是否包含以下提交，如若没有请更新SDK：

```
commit 1a81ee3e2d3726b9382ff2c48d08f4d837bc0143
Author: Sandy Huang <hjc@rock-chips.com>
Date:   Mon May 10 16:52:04 2021 +0800

    drm/rockchip: gem: add flag ROCKCHIP_BO_ALLOC_KMAP to assign kmap

    RGA need to access CMA buffer at kernel space, so add this flag to keep
kernel
    line mapping for RGA.

    Change-Id: Ia59acee3c904a495792229a80c42f74ae34200e3
    Signed-off-by: Sandy Huang <hjc@rock-chips.com>
```

Q4.5：“RGA_MMU unsupported Memory larger than 4G!”报错该如何解决？

A4.5：该报错通常对应HAL层报错：

```
RgaBlit(1483) RGA_BLIT fail: Invalid argument
Failed to call RockChipRga interface, please use 'dmesg' command to view driver
error log.
```

该报错标识当前配置的图像任务配置的内存无法满足当前匹配到的硬件核心对内存的要求，由于不同的硬件版本的RGA的IOMMU对内存位数的要求不同，当分配的内存超过对应硬件的限制时，则会出现该该报错，详细的不同硬件版本RGA的限制可见《Rockchip_Developer_Guide_RGA_CN》中的概述——设计指标小节。

当出现该报错时，通常有以下几种场景以及对应的解决方案：

1. 在搭载多种RGA的芯片平台（例如RK3588搭载有2颗RGA3核心、1颗RGA2核心）上，没有使用importbuffer_xx接口获取handle，而是直接使用wrapbuffer_xx接口调用im2d api时：

   由于没有使用importbuffer_xx来提前映射外部内存到RGA驱动内存，因此在实际任务匹配中无法提前获知内存是否不满足对应核心的限制，因此在高负载场景下可能会出现该报错，建议使用importbuffer_xx提前将外部内存导入到RGA驱动内部，避免该问题。

2. 在搭载多种RGA的芯片平台（例如RK3588搭载有2颗RGA3核心、1颗RGA2核心）上，使用了importbuffer_xx接口获取handle，但是依旧存在该问题：

   可以检查一下配置的图像任务的参数，确认是否配置了仅有RGA2核心（内存访问受限制的核心）支持的功能或格式，以RK3588为例，color fill功能和YUV422/420 planar格式均是RGA2核心特有的功能和格式，因此该场景下必须分配4G以内内存空间的内存调用RGA。

   常见的分配4G内存方式可以查看以下示例代码：

   <librga_souce_path>/samples/allocator_demo/src/rga_allocator_dma32_demo.cpp

   <librga_souce_path>/samples/allocator_demo/src/rga_allocator_graphicbuffer_demo.cpp

   如果使用的其他分配器，例如mpp_buffer、v4l2_buffer、drm_buffer等，请查询对应分配器是否支持限制分配4G以内内存空间内存，并按照对应方式申请复合RGA硬件要求的内存。

3. 仅搭载一种RGA的芯片平台（例如仅搭载RGA2的RK3399、RK3568、Rk3566）上：

   当芯片平台上仅搭载内存访问受限制的核心时，则调用RGA时必须申请符合搭载核心对内存要求的内存，解决方案同上场景2。

4. 当使用DRM、malloc、new等不支持指定分配4G以内内存空间的内存的内存分配器时，也可以通过修改uboot的内存映射范围来解决。

uboot相关修改可以参考SDK文档中 **uboot开发文档->Chapter-8 调试手段->修改DDR容量** ，将内存映射范围全局限制在0~4G内存空间以内即可。

Q4.6：“rga_policy: invalid function policy”、“rga_job: job assign failed”字样报错是什么导致的？

A4.6：可以开启驱动运行日志查看，具体错误原因

例如：

```
rga_policy: start policy on core = 4
rga_policy: RGA2 only support under 4G memory!     //标识当前搭载的RGA2核心仅支持4G以内的内存。
rga_policy: optional_cores = 0
rga_policy: invalid function policy
rga_policy: assign core: -1
rga_job: job assign failed
```

```
rga_policy: start policy on core = 1
rga_policy: core = 1, break on rga_check_dst       //对应核心不支持的原因日志，这里是dst通道的图像参数不满足当前核心要求（可以查阅文档确认该核心支持情况，这里core 0x1、0x2为RGA3核心，0x4为RGA2核心）
rga_policy: start policy on core = 2
rga_policy: core = 2, break on rga_check_dst       //对应核心不支持的原因日志，同上。
rga_policy: start policy on core = 4
rga_policy: RGA2 only support under 4G memory!     //对应核心不支持的原因日志，标识当前不匹配原因为该核心不支持4G内存空间以外的内存。
 rga_policy: optional_cores = 0
rga_policy: invalid function policy
rga_policy: assign core: -1                        //遍历全部核心后，无可匹配核心，则上报匹配失败错误。
rga_job: job assign failed
```

以上两种情况可以根据对应的日志去确认配置的参数信息，并针对性的进行修改。

Q4.7：“rga：Rga err irq! INT[701],STATS[1]” 调用RGA出现中断报错是什么导致的？

A4.7：该问题通常发生在RGA硬件执行过程中遇到问题异常返回，异常原因很多，常见的有内存越界、异常配置。建议遇到该问题优先检查传入的内存是否会发生越界。

Q4.8：“rga: Rga sync pid 1001 wait 1 task done timeout” 硬件超时报错一般是什么导致的？

A4.8：硬件超时报错原因有很多种，可以按照以下情形依次排查：

1). 检查整体流程，确认没有其他模块或应用对该块buffer持锁或异常占用中，当同一块buffer被其他模块异常占用时，RGA无法正常读写数据，超过了驱动设计的200ms的阈值后，便会异常返回并打印报错。

2). 检查当前系统的DDR带宽与利用率，由于RGA的总线优先级较低，当DDR负载跑满时，如果RGA在200ms内没有执行完毕，驱动便会异常返回并打印该报错。

3). 确认RGA超时报错前是否已经有其他IP模块的报错，例如ISP、vpu等，当在同一条总线上的硬件出现问题的情况，可能会导致RGA无法正常工作，驱动等待超过200ms后，便异常返回并打印报错。

4). 确认当前RGA频率（可以参考 **Q1.4** 中RGA频率相关操作），某些场景可能会出现同一条总线上的模块降频后影响到RGA的频率，RGA频率下降从而导致整体的性能下降，无法在200ms内完成工作，驱动便会异常返回并打印报错。

5). 部分芯片RGA被超频到一个较高的频率，此时RGA频率上升但是电压没有提升，会导致RGA整体性能显著下降，导致无法在规定阈值内完成工作，从而驱动异常返回并打印报错。该场景建议开发者将RGA频率修改至正常频率，超频对整体芯片的稳定性与使用寿命均有影响，强烈不建议该种行为。

6). 以上场景均没有发现问题，可以尝试在RGA超时报错返回后，将目标内存中的数据写到文件中，查看RGA是否有写入部分数据，如有写入部分数据，请重新确认1-5场景，该现象明显为RGA性能表现不足导致；如果目标内存没有被RGA写入数据，收集对应的日志信息以及相关实验过程，联系维护RGA模块的工程师。

Q4.9：当出现timeout报错时，同时伴随着“rga_job: hardware has finished, but the software has timeout!”日志，是什么原因？

A4.9：当出现该日志则说明当前系统环境负责中断的CPU核心被抢占，导致RGA驱动在上半部的硬件中断结束后，等不到下半部的软中断，超过驱动设置的超时阈值后，驱动上报的超时错误。

这种情况常见于应用层存在实时进程抢占了CPU，导致驱动设备无法正常工作，不建议使用实时进程强制抢占CPU资源，出现该问题只能从CPU侧进行优化，避免负责中断的CPU核心被抢占无法执行其他设备驱动的软中断。