计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2017,26(10):95−101 [doi: 10.15888/j.cnki.csa.006012]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

SurfaceFlinger 在 X Window 系统环境下的运行方案①

江　帆, 贺也平, 周启明

(中国科学院 软件研究所 基础软件国家工程研究中心, 北京 100190)

摘　要: 本文给出一种将 Android 图形系统 SurfaceFlinger 移植到桌面 Linux 发行版的 X Window 系统环境下运行的方案. 在 X Window 系统环境下运行的 SurfaceFlinger 可使 Android 运行环境中以本地进程形式的 Android 应用进程的 UI 界面显示到 X Window 的窗口中. 使用 Mesa 作为 OpenGL ES 实现并使 Mesa EGL 兼容 Android 的本地窗口 ANativeWindow, 同时借助 Androidx86 的 gralloc.drm.so 模块, 实现了 Android 应用程序的 UI 渲染过程 SurfaceFlinger 的图像合成过程能够使用 GPU 进行硬件加速. 另外, 用 X11 的 DRI2 扩展协调 SurfaceFlinger 的窗口和 X Server 的 DDX 驱动, 使合成后的图像能高效地更新到窗口中, 避免了 SurfaceFlinger 的图像缓存由独立显存到系统内存的拷贝过程. 经实验, 在本移植方案下, 第三方 3D 基准测试软件 San-Angeles 能达到 60FPS 的帧率. 相比于已有方案, 本方案的架构更加简洁高效, 且支持硬件加速.

关键词: Android 图形系统; SurfaceFlinger; X Window 系统; 移植; 兼容环境

引用格式: 江帆,贺也平,周启明.SurfaceFlinger 在 X Window 系统环境下的运行方案.计算机系统应用,2017,26(10):95–101. http://www.c-s-a.org.cn/1003-3254/6012.html

Method to Run SurfaceFlinger on X Window System

JIANG Fan, HE Ye-Ping, ZHOU Qi-Ming

(National Engineering Research Center of Fundamental Software, Institue of Sofware, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: This paper presents a method to transplant Android's graphic system, SurfaceFlinger, to run on desktop Linux distribution's X Window System. SurfaceFlinger running on X can make UI of the Android applications, running as native processes in the desktop Android runtime, show in a window of X. Using Mesa as the OpenGL ES implementation and making Mesa EGL compatible with Android's native window, together with gralloc.drm.so module, the Android UI rendering process and SurfaceFlinger's graphic compositing process can be implemented by using the GPU for hardware acceleration. In addition, using X11's DRI2 extension to coordinate window of SurfaceFlinger with X Server's DDX driver, can avoid graphic buffer coping from GPU's dedicated memory to the system memory. In our experiment, the third-party 3D benchmark software San-Angeles can achieve 60FPS on variety GPUs. Compared with the existing method, the architecture of our method is more concise, efficient, and supportive of hardware acceleration.

Key words: Android graphic system; SurfaceFlinger; X Window system; transplant; compatibility environment

Android 运行环境可以使数量众多的 Android 应用程序不经修改即可在桌面 Linux 系统上以本地进程的形式运行, 这对于原本应用资源匮乏的桌面 Linux 系统来说是一个很好的补充[1]. 而使 Android 的图形系统 SurfaceFlinger 在桌面 Linux 系统上运行是构建 Android 运行环境的重要环节. SurfaceFlinger 的主要功能是将应用窗口和系统窗口合成一帧完整画面并提交到屏幕显示. 张超等[2]成功地实现 SurfaceFlinger 在桌

① 基金项目: 国家科技重大专项 (2012ZX01039-004)
收稿时间: 2017-01-17; 采用时间: 2017-02-23

面 Linux 的 X Window 环境下运行. 其主要方法是利用软件渲染的方法方式将 SurfaceFlinger 合成的图像写入共享内存, 再由一个桌面图形程序输出画面. 该方法的不足是使用软件渲染而没有调用 GPU 实现硬件加速, 造成占用大 CPU 时间, 并且画面输出过程需要不断调用 IPC, 在处理 3D 图形时只有个位数的帧数, 性能十分低下.

提高图形系统性能的思路是使用 GPU 进行硬件渲染而非使用 CPU 进行软件渲染. arm 架构和 x86 架构在图形系统方面存在较大差异. 具体表现为 arm 架构下的 GPU 没有独立显存, 所有程序的图像缓存都位于系统内存, 而在 x86 的 Linux 内核之上, 需要使用 DRI 框架来调用 GPU, 并且 GPU 有自己的独立显存[3]. 而面向 arm 架构设计的 SurfaceFlinger 的源码即使能面向 x86 架构编译, 也不能直接在 x86 架构的 Linux 内核之上运行. 另一方面, SurfaceFlinger 在自己的图像缓存上合成好的画面帧需要提交到 X Window 的窗口中显示. 因此, 要实现 SurfaceFlinger 在 X Window 系统环境下运行, 需要解决的问题主要有两个: 1) 使 Android 应用程序的 UI 渲染过程和 SurfaceFlinger 的图像合成过程能够在 x86 架构的 Linux 内核之上进行硬件加速; 2) 使 SurfaceFlinger 的图像缓存能够显示在 X Window 的窗口中.

从 OpenGL ES 的角度来看 Android 应用程序的 UI 渲染过程和 SurfaceFlinger 的图像合成过程, Android 应用程序和 SurfaceFlinger 都可以看作是 OpenGL ES 的应用程序. 因此首先需要一种能在 x86 架构下使用的 OpenGL ES 的实现作为 GPU 驱动. Androidx86 项目在让 Android 系统在 x86 架构的 PC 上运行做出了重要贡献[4]. 该项目使用 Mesa[5]作为 OpenGL ES 实现, 并基 DRM[6]和 GEM[7]实现了 gralloc.drm.so 模块, 成功地使 SurfaceFlinger 在 x86 架构下运行并实现硬件加速. 但是该项目的目标是在 x86 架构下运行完整的 Android 系统, 因此其运行时会占用整个显示屏, 并不能实现在桌面 Linux 的图形系统 X Window 的环境下独立运行 SurfaceFlinger 程序. 本文以 Mesa 作为 OpenGL ES 实现, 借用 Androidx86 的扩展的 Mesa EGL 的 DRI 驱动实现对 Android 本地窗口 ANativeWindow 的支持, 并使用 gralloc.drm.so 进行图像缓存的申请, 成功实现了基于硬件加速的 Android 应用程序 UI 渲染. 对于 SurfaceFlinger 将画面提交到窗口显示的过程, 首先 SurfaceFlinger 需要有自己的窗口. 其次使用 gralloc.drm.so 直接通过内核 DRM 模块申请的显存无法和 SurfaceFlinger 的窗口进行内在绑定. 本文使用 X11 的 DRI2 扩展协调 SurfaceFlinger 的图像缓存和窗口, 有效避免了 SurfaceFlinger 图像缓存由 GPU 独立显存到系统内存的拷贝, 使得 SurfaceFlinger 的窗口画面更新过程更加流畅.

本文余下章节的结构如下. 第 1 节, 以 Android 应用程序 UI 渲染到 SurfaceFlinger 合成的画面在窗口中显示的过程为线索, 分析整体架构的运作流程. 第 2 节, 为更好地理解系统架构的原理, 分析 Android 应用程序 UI 渲染过程中涉及到的关键组件. 第 3 节, 在不考虑 X Window 系统的情况下, 讨论 Android 应用程序和 SurfaceFlinger 如何在 x86 的架构下运行并支持硬件加速. 第 4 节, 给出 SurfaceFlinger 和 X Window 系统兼容的关键技术和进一步的优化方案. 第 5 节, 给出并分析移植方案在各型 GPU 上运行的实验结果. 第 6 节, 结语.

1 总体架构

图 1 描述了 SurfaceFlinger 移植到 X Window 环境下的架构. 序号 1 至 19 描述了从 Actitity 的 UI 渲染到画面最终提交到窗口的完整过程. 整个系统运行于开启了 Binder 驱动的 x86 架构的 Linux 内核之上. /dev/binder 和/dev/dri/card0 分别为 Binder IPC 和 DRM 内核子系统的设备文件.

App 的 UI 使用硬件渲染路径 (1~6). 软件渲染调用 CPU 在系统内存上进行计算, 这样会占用大量 CPU 时间, 性能低下. 因此从 OpenGL ES 的角度来看, 所有 App 都是 OpenGL ES 的应用程序. App 的图像缓存通过 EGL 向 Surface 申请. Android 的本地窗口是 BufferQueue(由 producer 和 consumer 组成) 的 wrapper, 因此 Surface 拿到的图像缓存是从 BufferQueue 申请的 GraphicBuffer. GraphicBuffer 包含的图像缓存的实际句柄则是通过 gralloc.drm.so 调用 libdrm 向内核的 GEM 模块申请的.

SurfaceFlinger 同样拥有自己的 Surface, 但是它的图像缓存则是通过 xwindow.default.so 调用 X11 的 DRI2 扩展向 X Server 的 DDX 驱动申请的, 而非直接调用 libdrm 向内核申请. 这样可使 SurfaceFlinger 图像缓存的 GEM 句柄和 SurfaceFlinger 自己的窗口在 X Server 端产生内在绑定.

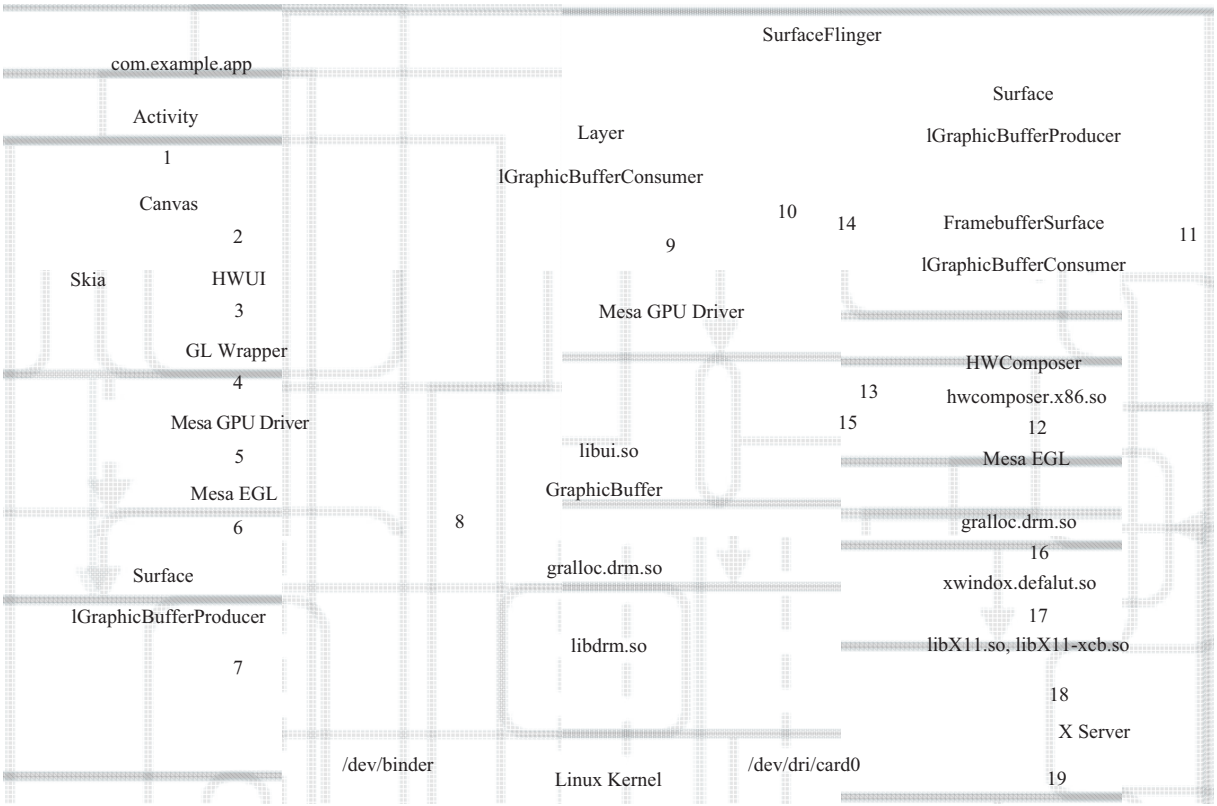

图 1 系统架构

当一个 Activity 的一帧画面渲染好以后, 就通过 BufferQueue 经 Binder IPC 机制将 GraphicBuffer 提交到 SurfaceFlinger 中 (7~8). GraphicBuffer 支持序列化, 因此可以通过 Binder 传递. 每个 Surface 对应 SurfaceFlinger 中的一个 Layer. 当一个由 EventThread 发处的 VSync 同步信号到来时, SurfaceFlinger 就会调用 OpenGL ES 将这些 Layer 合成为一帧画面 (9~10). SurfaceFlinger 同样需要有自己的本地窗口 Surface 来承载这一帧合成好的图像. 在 INVALIDATE_ON_VSYNC 宏置为 1 时, 在 Surface 缓存更新的同时发起一个画面更新.

因为桌面上没有硬件 HWComposer, 因此 SurfaceFlinger 的画面合成过程是基于 OpenGL ES 完成的, 而 hwcomposer.x86.so 只是一个符合 HWComposer 1.0 接口标准的软件实现, 并不是硬件 HWComposer 的驱动程序. 它的 set() 接口仅调用了 EGL 层的 eglSwapBuffers() 函数进行前后台图像缓存更新, eglSwapBuffers() 再调用 Surface 的 queueBuffer() 将一个 GraphicBuffer 通过 BufferQueuer 传给了同位于 SurfaceFlinger 进程中的作为 consumer 的 FrameBufferSurface 对象 (11~14). FrameBufferSurface 接下来通过 gralloc.drm.so 将这个渲染好的 GraphicBuffer 提交到窗口.

在 HAL 层中基于 Xlib 和 X11 的 DRI2 扩展实现了 xwindow.default.so 模块. SurfaceFlinger 通过该模块和 X Server 通信, 并创建属于 SurfaceFlinger 自己的窗口. gralloc.drm.so 将这个渲染好的 GraphicBuffer 提交到窗口的过程由 xwindow.default.so 调用 X11 的 DRI2 扩展的 pageflip 功能完成 (15~19).

本项目基于源码分支 android-5.1.1_r9 开发. 后续章节中关于 Android 图形系统的分析以该版本为基准.

2 Android 应用程序的 UI 渲染路径

为理解移植后的系统架构如何运作, 我们以 Android 应用程序 UI 渲染到图像呈现在屏幕的过程为线索, 分析该过程中涉及的一些 Android 图形系统的关键组件.

要让 SurfaceFlinger 在 X Window 环境下运行, 我们首先需要了解在 arm 架构下 Android 系统中 App 的 UI 是如何渲染和显示到屏幕的以及 SurfaceFlinger 在这个过程起到的作用. App 的 UI 界面的渲染方式分为 2D 和 3D 两种. 进行 2D 渲染的方式包括: 1) 调用

Canvas 的 drawRect(), drawText() 等方法将像素点绘制到一个 Bitmap; 2) View 的子类绘制 Button、List 等空间; 3) 或一个定制的 View 类实现开发者自己定义的行为等. 2D 渲染有两条路径, 一是使用 hwui 的基于 OpenGL ES2.0 的硬件渲染, 一种是使用 Skia 软件渲染引擎的软件渲染. Android 的 UI 绘制也支持硬件加速, 这时 Activity 会创建一个 GLSurfaceView 并且使用 OpenGL ES 的 Java 绑定接口 (android.opengl.*) 进行画面渲染. AOSP 中提供的 libEGL.so、libGLESv1_CM.so、libGLESv2.so、libGLESv3.so 只是 EGL 和 OpenGL ES 实现的 wrapper, 底下真正调用的是由供应商提供的闭源专属 GPU 驱动或 Android 自己实现的软件 GPU 模拟 PixelFlinger. 但是不论是 2D 渲染还是 3D 渲染, 最终的渲染结果的所有像素点都位于该 Activity 的 Surface 之上.

2.1 SurfaceFlinger

SurfaceFlinger 是 Android 的窗口合成器. 它和 App 的 Surface 构成了 C/S 架构. 一个完整的屏幕界面实际上是由多个窗口 (包括导航条、状态条和应用 UI) 合成后的结果. 每个窗口分别属于不同的进程, 由这些进程独立渲染在自己的 Surface 之上. 每个 Surface 对应 SurfaceFlinger 中的一个 Layer, 这些 Layer 通过 z 轴顺序排列起来. SurfaceFlinger 基于 OpenGL ES 的 API 将这些 Layer 合成一帧完整的画面.

2.2 BufferQueue 和 gralloc

Surface 和 SurfaceFlinger 构成 C/S 架构. BufferQueue 是 Surface 和 SurfaceFlinger 之间进行 GraphicBuffer 交换的通道. App 从 Surface 获取新的 GraphicBuffer 并将绘制好的 GraphicBuffer 通过 BufferQueue 传递给 SurfaceFlinger, 而 SurfaceFlinger 从 BufferQueue 接收渲染好的 GraphicBuffer 用于下一帧的画面合成, 同时将新的 GraphicBuffer 放入 BufferQueue 传递给 App 的 Surface.

gralloc 是 Android HAL 层定义的图形缓存分配器. 在 BufferQueue 中传递的 GraphicBuffer 对象通过 gralloc 获取实际的图像缓存描述句柄. gralloc 的图形缓存申请接口 alloc() 接收 width, height, pixel format 以及 usage flags 等参数. App 和 SurfaceFlinger 使用图像缓存的目的不同, App 是为了渲染 UI 画面并传递给 SurfaceFlinger 进行图形合成, 而 SurfaceFlinger 要在自己的图像缓存上合成最终画面并提交到屏幕显示. 因此它们申请图像缓存的类型也不一样, 具体表现为 usage flags 参数的区别. SurfaceFlinger 是 GRALLOC_USAGE_HW_FB, 而 App 的则是其他类型. 因此, 需要在 App 和 SurfaceFlinger 之间传递的 App 的图像缓存是从 Ashmem 申请的共享内存, 而 SurfaceFlinger 的图像缓存则是 Framebuffer. 但在 arm 架构下, 两者实际上都位于系统内存之上. App 的画面渲染完成时会通过 Ashmem 共享内存换地给 SurfaceFlinger, 而 SurfaceFlinger 在画面合成完毕时会调用 Framebuffer 的 pageflip 机制将新生成的画面更新到屏幕显示.

2.3 Android 的本地窗口 Surface

本地窗口是 EGL 中的概念, EGL 为了保证 OpenGL ES 的平台独立性, 需要适应各种操作系统下的本地窗口. ANativeWindow 是 Android 系统定义的本地窗口, 也是 EGL 的 Android 平台适配窗口, 其中定义了获取和释放图像缓存的接口. 图像缓存用 ANativeWindowBuffer 其定义, 它描述了窗口的基本信息如宽度、高度、像素格式等基本信息. 实际图像缓存的地址和句柄 buffer_handle_t 结构体描述. Surface 是应用层次的本地窗口, 不论是 App 还是 SurfaceFlinger, Android 上的所有图形程序都需要有自己的 Surface, 而 Surface 的作用就是给 EGL 层提供 ANativeWindow 的接口. 上层应用即可通过这些接口获取和释放图像缓存.

3 在 x86 架构下运行 SurfaceFlinger

要让 SurfaceFlinger 能在 X Window 的环境下运行, 首先要解决的问题是如何让 SurfaceFlinger 在 x86 架构的 Linux 内核之上运行. Androidx86 项目很好地解决了 SurfaceFlinger 在 x86 架构下运行的问题. Android 系统主要是为 arm 架构设计的, arm 架构下的 GPU 没有自己的独立显存, SurfaceFlinger 通过 Surface 申请的的 Framebuffer 硬件缓存仍然是系统内存的一部分. 而 x86 架构下, GPU 有自己的专属独立图显存. SurfaceFlinger 本身基于 OpenGL ES 实现, 要使用 x86 架构的 GPU 进行硬件加速, 就需要有可以在 x86 架构下运行的 OpenGL ES 实现. 因此 Androidx86 项目在图形系统方面的主要工作有两点: 1) 使用 Mesa 作为 OpenGLES 实现; 2) 实现了 gralloc.drm.so.

3.1 Mesa 和 EGL

DRI 是一个使应用程序在 X Window 系统的环境

下直接发访问图形硬件的框架. 其实现分散在 X Server 和它的客户端共享库 Xlib、Mesa 以及 DRM 内核子系统中[8]. Mesa 在 DRI 框架中扮演了 GPU 驱动的角色, 因此 Androidx86 选择 Mesa 作为 OpenGL ES 的实现. 但是 Mesa 默认只支持 X Window 系统的本地窗口, 而不支持 Android 的本地窗口 ANativeWindow. 在 Mesa 的架构中, EGL 是 OpenGL ES 和本地窗口之间的一个中间层, 其作用之一是保证 OpenGL ES 的平台独立性. 因此 Androidx86 在 Mesa EGL 中添加了对 ANativeWindow 的支持 (platform_android.c), 使得 Mesa 可以通过 Surface 获取 ANativeWindow. 因此本文中借用 Android 对 Mesa EGL 的扩展实现对 Android 本地窗口的支持.

3.2 gralloc.drm.so

在 X Window 环境下, 一个桌面 OpenGL ES 程序的图像缓存是由 Mesa EGL 的 DRI 驱动 (platform_x11.c) 借用 X11 的 DRI2 扩展通过 X Server 的 DDX 驱动向内核申请的. 而在 Android 系统中, 实际图像缓存申请操作是由 GraphicBuffer 类调用 HAL 层的 gralloc 完成的. 在 arm 架构下, gralloc 为应用申请的图像缓存来自 Ashmem 共享内存, 而为 SurfaceFlinger 申请的图像缓存来自 Framebuffer, 这些图像缓存本身都位于系统内存, 而 Mesa 无法基于系统内存调用 GPU 进行硬件加速. 这一点也从侧面说明文献[2]中的移植方法不支持硬件加速. gralloc.drm.so 基于 libdrm, 实现了从 x86 架构的独立显存为应用程序和 SurfaceFlinger 申请图像缓存. libdrm 是内核 DRM 子系统的用户态共享库, 应用程序可以通过 libdrm 调用内核的 GEM 模块来创建图像缓存对象, 在用户态使用一个 GEM 句柄 (本质是一个整数) 作为图像缓存的引用. 因此在 arm 架构下, buffer_handle_t 包含的是图像缓存的首地址,而在 x86 架构下包含的则是 GEM 句柄.这时,在 Androidx86 系统中, 不论是 App 还是 SurfaceFlinger, 通过 graloc.drm.so 申请图像缓存的过程是一样的, 并不会因为 alloc() 接口的 usage flags 参数不同而不同. 另一方面, 在 Androidx86 环境下, SurfaceFlinger 会占用整个屏幕作为其画面输出的目标. 这样无法和桌面 Linux 发行版的 X Window 系统兼容. 因此, 要让 SurfaceFlinger 的合成画面在 X Window 系统的窗口中显示, 就需要将 SurfaceFlinger 的角色由 Android 窗口合成器降级为 X Window 系统的应用程序.

4 在 X Window 的环境下运行 SurfaceFlinger

X Window 系统是桌面 Linux 系统的标准图形系统[9]. 它采用 C/S 架构, 客户端程序基于 Xlib 和 X Server 通过 TCP 连接通信, 如果客户端程序和 X Server 位于统一系统上, 则使用 IPC 通信. 因为 X Window 系统为分布式环境而设计, 客户端程序通过 Xlib 向 X Server 申请的资源 (例如一个窗口) 位于 X Server 中, 客户端程序只能拿到该资源的一个整数 ID 作为引用.

在 Androidx86 中, SurfaceFlinger 在初始化时直接作为 DRM Master, 进行 KMS 设置, 这样会与 X Server 发生冲突. 因为在一个桌面 Linux 系统中只允许有一个 DRM Master. 在画面需要更新到屏幕是直接调用 drm 的 pageflip 机制进行屏幕画面刷新, 并且 SurfaceFlinger 合成的画面会显示到整个屏幕上. 现在我们要让 SurfaceFlinger 在 X Window 系统的环境下运行, 具体而言是让 SurfaceFlinger 将更新后的图像缓存提交到一个窗口中显示.

4.1 移植方案

首先 SurfaceFlinger 需要有自己的窗口, 此时 SurfaceFlinger 需要调用 X Window 的客户端共享库, 而采用 NDK 编译的 SurfaceFlinger 的 ELF 文件无法使用桌面 Linux 的默认动态链接器和已安装的 Xlib 共享库[1]. 因此需要将 Xlib 的源码转换为 Android 项目并使用 NDK 进行编译, 并基于移植的 Xlib 在 HAL 实现了 xwindow.default.so 模块. SurfaceFlinger 即可使用 xwindow.default.so 模块创建自己的窗口.

当 SurfaceFlinger 需要将自己的 Surface 中的最新的 GraphicBuffer 提交窗口时, 在 SurfaceFlinger 进程中已有两个资源, 一是通过 GraphicBuffer 的 buffer_handle_t 中获取的图像缓存标识 GEM 句柄, 一是标识窗口的整数 ID. 一个直观的方法是利用 libdrm 将 GEM 句柄描述的图像缓存映射到系统内存空间, 将这段空间封装成 XImage 最后通过 Xlib 提交到 X Server 显示.

XImage 和 Pixmap 的区别是前者位于客户端进程而后者位于 X Server 进程. XImage 是一个结构体, 包含了位图的各种格式、大小等信息, 同时还有一个指针指向像素数据所在的内存地址. 所以我们可以根据图像缓存在系统内存空间的映射来构造这个 XImage, 然后让这个 XImage 显示在 SurfaceFlinger 的窗口中.

这个 XImage 最终要通过 Xlib 传给 X Server, 由 X Server 显示出来. 但不论是 TCP 连接还是 IPC 机制, 传递大块内存的效率较低. 所以我们使用了 X11 的共享内存扩展 MIT-SHM[10]. 这样, XImage 就可以通过共享内存传递给 X Server. 相比于文献[2]的架构, 本架构下画面更新频率由 HWComposer 中用于模拟硬件 VSync 信号的 VSyncThread 的信号频率决定 (一般为 60FPS, 和显示屏刷新率保持一致), 而不需要额外的同步信号控制, 也不需要额外的图像显示进程, 因此总体架构更为简洁.

4.2 优化方案

在各种 GPU 上测试时发现, 只有在 Intel 集成显卡上画面刷新频率能达到 60FPS, 而 Nvidia 和 AMD 独立显卡上画面十分卡顿. 在上述移植方案中, App 和 SurfaceFlinger 从 Surface 获取的图像缓存都是通过 libdrm 向内核的 GEM 模块申请的, 因此基于 Mesa 的 App 的 UI 渲染过程和 SurfaceFlinger 的画面合成过程是使用 GPU 进行硬件加速的. 问题在于将 GEM 句柄标识的图像缓存映射到系统内存空间并封装到 XImage 的过程需要将图像缓存从独立显存读取到系统内存, 这个过程的延迟较大. 在该方法下, 每次画面更新都需要拷贝一次, 因此拖慢了画面刷新频率.

优化的思路来自于对桌面上基于 Mesa 的图形程序架构的思考. 基于 X Window 系统的图形程序申请图像缓存是由 X Server 端的 DDX 驱动完成的, 本质仍然是通过 libdrm 调用 GEM 内核模块创建显存对象. 而桌面上基于 Mesa 的图形程序调用 eglSwapBuffers() 进行画面更新时, EGL 的 DRI 驱动并没有将图像缓存拷贝到系统内存, 但是画面仍然能在一个由 X Window 系统创建的窗口中更新. 因此桌面 Linux 上基于 Mesa 的图形程序的图像缓存的申请和更新是由 Mesa 和 X Window 协同完成的. 其原理在于, 在 DRI 架构中, X Server 提供 X11 协议的 DRI2 扩展, 用于 DRI 客户端进行窗口系统和 DDX 驱动的协调[8,11]. 因此, 在上文移植方案的基础上需要做两点改进: 1) 在 gralloc.drm.so 中区分 App 和 SurfaceFlinger 图像缓存申请方式, App 直接使用 libdrm 向内核 GEM 模块申请, 而 SurfaceFlinger 使用 DRI2 扩展接口通过 DDX 驱动申请; 2) SurfaceFlinger 进行画面更新时不再直接使用 DRM 的 pageflip 接口, 而使用 DRI2 扩展的缓存交换接口. 这样就不需要将 SurfaceFlinger 的图像缓存拷贝到系统内存了, 因此是一种完全支持硬件加速的架构.

5 性能测试

系统性能测试方法是运行一个以 Surface 为本地窗口的 Android OpenGL ES 图形程序, 测量其在一段时间内的平均帧率 (FPS). 同时测量该程序的 CPU 负载. 采用 San Angeles Observation 作为基准测试软件, 后文简称 Angeles. Angeles 基于 OpenGL ES1.0 实现, 每次运行约 109 秒, 结束后会自动给出该时间内的总帧数和帧率. 系统部署平台选取了三台 GPU 分别由 Intel、Nvidia 和 AMD 制造的 PC. 三台 PC 均运行 Linux Mint 17.3 操作系统, 并使用开启了 Binder 驱动的版本号为 4.2.6 的 Linux 内核. 各 PC 的关键硬件参数如表 1 所示. 实验方案是以 800*480、1280*720 两种分辨率大小的窗口, 分别在各 PC 以优化前和优化两种架构运行 Angeles 程序, 并采集运行期间 SurfaceFlinger 和 Angeles 的 CPU 占用情况.

表 1 测试主机关键参数

主机编号	CPU	GPU
PC1	Intel Core i5 4200U	Intel GMA HD 4400
PC2	Intel Core i5 650	Nvidia Gefore 405
PC3	Intel Core i5 M460	Radeon HD 5650

从表 2 和表 3 可以看出, 不论是在哪种分辨率下, 优化前只有 PC1 的帧率能到达 60FPS 的速度, PC2 和 PC3 的帧率均小于 30FPS. 帧率低于 30FPS 时人眼便能觉察到画面的卡顿感. 原因在于 Intel 集显没有独立显存, 而是共享系统内存, 因此每次将显存拷贝是由系统内存的一个地址拷贝到另一个地址, 该过程的速度很快, 不会拖慢图像更新频率. 而在 PC2 和 PC3 上, 显存的拷贝需要从 GPU 独显拷贝到系统内存, 此过程耗时较长, 拖慢了 SurfaceFlinger 的画面更新速率. 另一方面, 观察两种分辨率下的优化前的 PC2 和 PC3 的帧率变化, 发现分辨率越大, 帧率越小, 其原因在于分辨率越大, 需要拷贝的显存越大, 拷贝时间越长.

表 2 图形性能测试结果 (800*480)

	主机	时间(s)	帧数	帧率(FPS)	CPU(S)(%)	CPU(A)(%)
优化前	PC1	108.87	6535	60.03	4.83	5.58
	PC2	109.80	212	1.93	25.16	0.50
	PC3	108.89	1952	17.92	24.88	4.85
优化后	PC1	108.86	6533	60.01	1.77	5.89
	PC2	108.87	6516	59.85	1.65	16.55
	PC3	108.86	6527	59.96	1.88	14.76

表 3 图形性能测试结果 (1280*720)

	主机	时间(s)	帧数	帧率(FPS)	CPU(S)(%)	CPU(A)(%)
优化前	PC1	108.87	6535	60.02	6.20	4.82
	PC2	109.69	175	1.60	25.06	0.75
	PC3	108.05	833	7.64	24.75	1.76
优化后	PC1	108.86	6533	60.01	1.88	6.20
	PC2	108.88	5219	47.93	1.39	14.65
	PC3	108.86	6529	59.97	1.89	14.48

优化后, 在 800*480 分辨率时, 三台 PC 均能达到 60FPS 帧率, 画面十分流畅. 只有在 1280*720 时, PC2 的帧率低于 60FPS, 其原因是 PC2 的 GPU 性能较弱. 优化后的运行帧率和 CPU 负载均优于张超等人[2]的实验结果.

从 SurfaceFlinger(S) 和 Angeles(A) 的 CPU 负载的变化情况分析, 不论在哪种分辨率下, 优化后相较于优化前, SurfaceFlinger 的 CPU 负载降低且均低于 2%, 而 Angeles 的 CPU 负载升高. 原因在于, 优化前, SurfaceFlinger 的显存拷贝过程需要占用大量 CPU 时间, 拖慢了画面更新速率. 在 BufferQueue 另一端的 Angeles 程序需要等待 SurfaceFlinger 交出空闲的 GraphicBuffer 才能进行下一帧的渲染, CPU 负责也较低. 优化后, SurfaceFlinger 的画面更新时没有了图像缓存的拷贝过程, SurfaceFlinger 本身不再是瓶颈, 而 Angeles 在渲染完一帧画面后可以立即从 BufferQueue 获取一个空闲的 GraphicBuffer. 因此造成这种变化.

6 结语

使 Android 的图形系统 SurfaceFlinger 能够在桌面 Linux 发行版的 X Window 系统的环境下运行, 是构建 Android 桌面运行环境的重要环节. 本文给出了一种简洁高效的移植方案. 使用 Mesa 作为 OpenGL ES 实现, 并借助 gralloc.drm.so 模块, 实现了 Android 应用程序的 UI 渲染过程和 SurfaceFlinger 的图像合成过程能够使用 GPU 进行硬件加速. 同时, 使用 X11 的 DRI2 扩展协调 SurfaceFlinger 的窗口和 X Server 的 DDX 驱动, 实现了窗口画面的快速更新, 且避免了 SurfaceFlinger 图像缓存由独立显存到系统内存的拷贝过程. 经实验, 本移植方案能基于各型 GPU 进行硬件加速, 图形性能相较于已有方案有显著提升.

参考文献

1 张超. 面向桌面 Linux 的 Android 运行环境构建[硕士学位论文]. 长沙: 国防科学技术大学, 2012.

2 张超, 易晓东, 戴华东. Android 图形系统向桌面 Linux 的移植. 2012 全国计算机体系结构学术年会论文集. 西安, 中国. 2012. 209–213.

3 Inki Dae Software Platform Lab. DRM driver development for embedded systems. http://elinux.org/images/7/71/Elce11_dae.pdf. [2011].

4 Android-x86-porting android to x86. http://www.android-x86.org/.

5 The mesa 3D graphics library. http://www.mesa3d.org/.

6 Direct rendering manager (DRM). https://dri.freedesktop.org/wiki/DRM.

7 Packard K. Gem-the graphics execution manager. 2008. https://lwn.net/Articles/283798/.

8 Direct rendering infrastructure. https://en.wikipedia.org/wiki/Direct_Rendering_Infrastructure.

9 Scheifler RW, Gettys J. The X Window system. ACM Trans. on Graphics, 1986, 5(2): 79–109.

10 Corbet J. MIT-SHM-the MIT shared memory extension. https://www.x.org/Releases/X11R7.7/doc/xextproto/shm.html. [1991].

11 Høgsberg K. The DRI2 extension. https://www.x.org/releases/X11R7.7/doc/dri2proto/dri2proto.txt. [2008].