基于子系统建模的 GPU 验证系统方法

刘江宁 (jiangning.liu@arm.com)、秦凯文 (kevin.qin@arm.com)、张帕特里克

摘要 - ARM Fast Model 是一种软件建模系统，用于仿真集成于子系统中的多个系统知识产权核。Fast Model **允许客**

关键词——GPU 验证、OpenGL ES、安卓、建模、仿真

I. 背景介绍

ARM Fast Model [1] 是一个便于创建虚拟平台模型的环境，这些模型能够以极高的仿真速度运行。ARM硬件知识产权核的接口可以通过 LISA+ 代码定义，并可轻松地进行可视化和组合，从而生成完整的子系统。现代 SoC厂商通常会在首款芯片准备流片时，提供完整的软件栈支持。在 ARM Fast Model 生成的平台上，我们不仅可以运行裸机基准测试，还可以运行安卓等主流移动操作系统，以及运行在 CPU 上的其他应用软件。此外，近年来移动生态系统不断发展，使得验证 GPU知识产权核的能力更具意义。

GPU架构发展迅速，通常采用大量统一标量核心来解决高度数据并行的流处理问题。与 GPU 相比，CPU 通常核心数量较少，但单个 CPU 核心比单个 GPU 核心更复杂，例如拥有更深的流水线和更大的缓存。由于历史原因，在以 CPU 为中心的计算机架构中，GPU 始终被视为外设。在全系统仿真中，我们通常使用 CPU 仿真所有设备，包括使用 GPU 进行 3D 渲染；但 CPU 与 GPU 在架构层面的巨大差异，使得快速 GPU 模拟难以实现。当前移动系统中不断提高的显示分辨率，对仿真环境提出了更高的计算资源需求。

此外，工业界的趋势是将 GPU 与 CPU 融合，以解决 CPU 和 GPU 之间内存访问能力不足及带宽受限这一棘手问题。无论如何，使用有限的 CPU资源来仿真和验证 GPU知识产权核都将是一项巨大挑战。整个系统的仿真和验证速度可能非常慢，运行一个简单的工作负载就需要数天甚至数月。

为了有效开展 CPU架构研究，一些研究工作（例如 Gem5 [4] NoMali模型 [9],）会有意忽略真实的 GPU工作负载，使使用 GPU 进行 3D 渲染的应用程序仍能运行。这样一来，GPU内存系统交互便无法得到完整仿真，因此无法呈现具有丰富图形特性的应用程序，也给在实际环境中启用真正的安卓软件栈带来了困难。为了呈现屏幕上的 2D GUI 等简单图形，Google安卓模拟器使用 QEMU [6] 进行软件渲染，本质上是利用 CPU资源来模拟简单的图形工作。采用这一方案，无法真正支持 GLES 2.0 [3] 及更高版本定义的复杂 3D 渲染，因此最终用户无法运行安兔兔、GFX 和 Unity 等 3D 基准测试。

传统上，图形学研究会使用跟踪技术捕获所有运行时 GL/GLES API 调用及其参数，然后使用周期模型 [8] 或 C模型离线重放这些调用。人们使用周期模型研究 GPU架构，但由于速度极慢，它只能运行几帧。C 模型速度稍快一些，但它只关注 GPU 研究，在跟踪重放期间完全忽略了 CPU 侧的计算。这无法帮助分析实际基准测试中 CPU 和 GPU 之间交错执行的逻辑。此外，有时 C模型的速度仍不足以运行商业基准测试。

然而，通过使用主机中的 GPU硬件资源进行 GPU模拟，可以实现目标端的 GPU/CPU模拟。在工业界，使用 GPU 进行 3D 渲染的软件编程模型已经发展为采用一致的 API，例如

Linux 上的 OpenGL 和 Windows 上的 DirectX，尽管不同代 GPU 的硬件架构可能存在显著差异。将目标 GPU 中的 3D 渲染任务直接交由主机 GPU 运行，可以更轻松地实现加速。

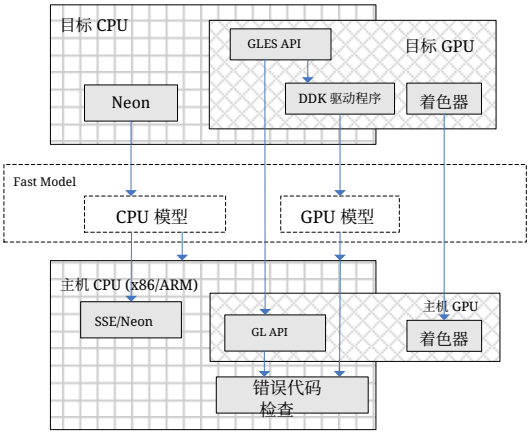


图 1. Fast Model 3D 渲染加速

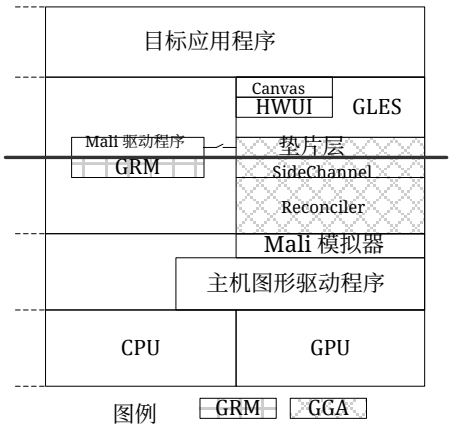


图2. 安卓图形软件栈

图 1 展示了 Fast Model 如何在完整的运行时工作环境中支持 3D 渲染加速。Fast Model 将目标端与主机完全隔离。目标端的 CPU 计算可以直接转换为主机上的指令，这通常涉及跨架构转换。目标端用于支持 GPU 的驱动程序也运行在 CPU 上，而目标端的着色器代码则运行在 GPU 上。EGL/GLES API 会直接分派到主机，由主机上的 GPU 相应地执行实际的 3D 渲染工作。同时，在主机上的验证阶段，我们可以比较目标端和主机返回的错误代码，这对于在真实 GPU 硬件上运行之前进行 GPU 驱动正确性验证至关重要。本文提出了一个系统化框架，使完整的图形软件栈能够运行在一个子系统上

ARM Fast Model。借助该框架，用户级应用程序、用户空间和内核空间中的原生图形驱动程序、安卓硬件用户界面、Surface Flinger、Gralloc 硬件抽象层 [12] 以及其他与图形相关的组件，都可以在单体式 Fast Model 虚拟环境中运行。采用这种方法，我们不仅能够轻松检测所有系统集成问题，还能揭示 CPU 与 GPU 之间的内存流量。这使我们有机会捕获图形驱动程序缺陷，以及 GLES API 引入的内存流量相关性能瓶颈。我们还提出了一种增强方案，以支持远程 3D 渲染加速。该方案最终有助于在真实生产环境中通过大规模集群机器开展真实 GPU 验证工作。

与 Youngsub 等人提出的 GPU-HIL 仿真技术 [7] 相比，我们提出的方案有两大优势：(a) 支持图形寄存器模型，因此可以直接执行并验证目标端的图形驱动程序。(b) 流片时无需修改 Android 镜像，因此能够在软件开发的最早阶段发现大多数集成问题。

本文其余部分安排如下。第二节概述使用主机 GPU 加速 Android 操作系统 3D 渲染的方法。第三节介绍 GPU 模型所支持的 GPU 驱动程序验证潜在能力。第四节提出面向集群的远程 3D 渲染方案。第五节展示图形软件栈仿真性能的实验结果。第六节总结全文并探讨未来工作。

II. 使用主机 GPU 加速渲染

为了加速 ARM Fast Model 中的 GPU IP 建模，我们探索了一种新方法：将计算任务从目标 GPU 分派到主机 GPU。图形软件栈与 CPU 侧的软件生态系统截然不同。传统上，图形领域的最终用户会使用 DirectX 或 Open GL/GLES 等高级 API 来辅助 3D 渲染开发。实际工作负载会被分派到 GPU 微型核心，作为所谓的着色器或内核执行。（图 1）将着色器二进制代码从目标 GPU 指令集架构转换为主机 GPU 的指令集架构极具挑战，而且也非常耗时。因此，我们转而分派着色器源代码或中间语言，并让主机侧的 GPU 执行实际渲染工作，然后将帧缓冲数据传回在 Fast Model 中运行的目标 Android 操作系统。通过这种方式，我们可以在 Fast Model 平台上全面支持 3D 渲染，最终用户也可以直接在虚拟化显示设备上查看 3D 渲染结果。对于片上系统软件栈开发人员而言，这是一项极具吸引力且切实可行的功能，可确保硬件和软件开发真正并行开展。

2.1 安卓中的 OpenGL ES

图2展示了 Android 操作系统中的图形软件栈。ARM Fast Model 将 Android 操作系统目标端与宿主操作系统隔离开来。在 Android 目标端，OpenGL ES API 可由多个不同来源调用，包括终端用户应用程序、Android 主屏幕启动器、Canvas 调用的 HWUI 以及 Surface Flinger 等。除了非常薄的 EGL/GLES 封装层之外，Android 中的库还可以拦截 OpenGL ES API。在我们的方法中，我们使用 vendor 文件夹中的一个名为 Shim Layer（libShim.so）的库来拦截所有 API，并将其同时分派给目标端的图形驱动程序和主机中的 Reconciler。一方面，目标端的原生图形驱动程序运行在 NoMali模型上，该模型与 Gem5 中的实现共享。与 Gem5 方案不同的是，对于每个 API，我们都需要在目标端等待，直到该 API 真正被分派到主机。另一方面，该 API 会被传递给主机图形驱动程序，以完成实际的 3D 渲染工作。

垫片层会在初始化时为每个进程注册一个软件中断，并通过该软件中断隐式发送宿主端 API 调用消息。这样，我们完全不需要更改安卓构建，只需实现一个名为 SideChannel 的 Fast Model 插件，在 Fast Model 中捕获此软件中断即可。所有参数以及指针参数所指向的数据都会打包成一条消息，并通过 SideChannel 传递给主机中的 Reconciler。Reconciler 会完成以下工作：(a) 在主机中维护 GLES 状态，并准备在主机中调用 API 所需的参数；(b) 通过 Mali 模拟器 [12] 或原生图形驱动程序在主机中调用 API；(c) 将主机中的帧缓冲区转换为目标端格式并写回。

对于某些需要在主机中公开 OpenGL 状态的 API，例如 TransformFeedbackBuffer 和 ShaderStorageBuffer，我们必须在主机中获取缓冲区数据，并将其写回目标端图形驱动程序返回的缓冲区内存。这样可以确保目标端的所有 API 都具有完整语义，因此图形驱动程序中在 CPU 上运行的代码始终能够正确工作。

Mali 模拟器是 ARM 的一款商业产品，它通过将 OpenGL ES 转换为 OpenGL 来构建一个虚拟的 Mali 设备驱动程序，因此会接收来自 Reconciler 的 API，并将其分发给真正的原生图形驱动程序。如果主机上的原生图形驱动程序直接支持 GLES，则不需要这一层，我们可以直接将 GLES API 传递给主机驱动程序。某些 API（如 glDraw 和 glClear）会触发主机驱动程序执行实际的渲染工作。当主机端发出 eglSwapBuffer API 时，Reconciler 会从主机上的帧缓冲读取像素，并通过 SideChannel 插件将其写入目标端的帧缓冲区地址。

本文将使用硬件 GPU 资源进行 3D 渲染加速的方案称为 GGA（通用图形加速），将通过目标端的图形驱动程序在 Fast Model 上运行 API 称为 GRM（图形寄存器模型）执行。我们可以选择是否启用 GRM；不执行 GRM 时，速度可能会快得多。通常，片上系统上的图形软件验证可以先从使用不含 GRM 的 GGA 开始。如果 GGA 模式运行良好，则可以在 ARM Fast Model 上启用 GRM 执行，以进一步验证目标端驱动程序集成的正确性。

2.2 API 延迟分派：隔离的目标端

主机地址空间给在主机上实现某些 OpenGL ES API、以模拟目标端相同行为带来了困难。这一挑战本质上源于 OpenGL ES API 参数属性的不完整性。例如，在向 API 传递内存指针时，有时并未指定传递给 OpenGL 状态机的内存大小，直到该内存即将被其他 API 使用时才能确定，因此我们必须将内存复制延迟到该指针真正开始被使用的时刻。为了支持延迟内存复制，我们必须在主机端的 GLES 状态机之上引入一个中间状态，并在必要时延迟分发 GLES API。

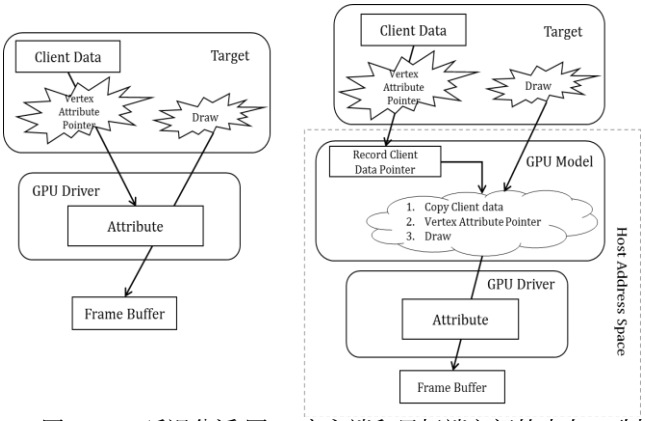


图 3. API 延迟分派

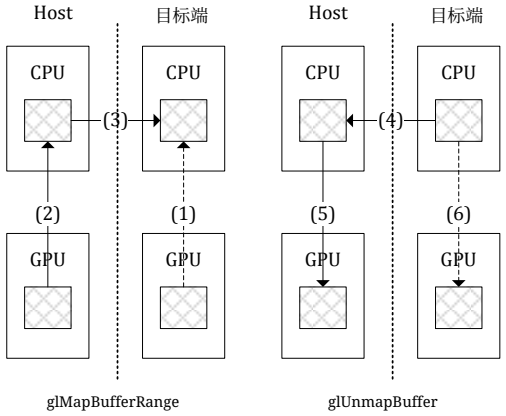


图 4. 宿主端和目标端之间的内存一致性

在图 3 中，glVertexAtrribPointer 指定了用于渲染的通用顶点属性数组的位置和数据格式。当它在目标端发出时，如果我们也想让它在主机端发出，我们就必须

以便将参数指定的所有 CPU 可访问内存数据从目标端复制到主机。不幸的是，OpenGL ES 并未设计带有明确数据大小、可供使用的此类 API，而只是通知 GPU 将访问 CPU 中的内存地址，并且要到 glDraw 时我们才能确切知道数据大小。这为最终用户提供了一定的使用灵活性，但它并不是一个自包含的解决方案。在这种情况下，我们必须在主机端采用批量分发方案，将 glVertexAtrribPointer 的分发延迟到 glDraw 时执行。这意味着我们必须在主机端记录并维护一些 OpenGL ES 状态，以映射目标端对应的状态，并在必要时使用这些状态。

2.3 目标端与主机之间的内存一致性 CPU 和 GPU 之间的内存访问可

由操作系统和内核空间中的驱动程序支持自然保证。然而，当我们希望将目标 API 分派到位于不同地址空间的主机时，情况就不同了。两个特殊的 API glMapBufferRange 和 glUnmapBuffer 尤其值得关注，它们用于将指定缓冲区对象的数据存储区全部或部分从 GPU 映射到 CPU，或取消映射。图 4 说明了如何正确处理这两个 API。对于 glMapBufferRange，我们应先在目标端调用，然后在主机上调用，接着将映射的缓冲区数据从主机复制到目标端，即 (1)→(2)→(3)。对于 glUnmapBuffer，我们应按相反的顺序操作：先将缓冲区复制回主机，再在主机上调用该 API，以取消映射主机 GPU 的缓冲区对象，最后也在目标端调用该 API，即 (4)→(5)→(6)。步骤 (3) 和 (6) 确保目标应用程序中的语义执行正确，并且是维护目标驱动程序中 OpenGL ES 缓冲区对象状态的必要步骤。

这样可以确保目标端与主机之间的内存一致性。尽管这两个 API 可能会因 CPU 与 GPU 之间的同步而产生 GPU 气泡，但在某些场景中，仍会大量使用它们来定位 GPU 中的一些反馈缓冲区。监控此类内存访问工作负载所造成的

这两个 API 对 GPU 性能调优可能非常有用。

2.4 使用提议的 GPU 模型技术进行目标库注入

在本文所述的技术中，需要在 Android 目标端安装两个库，才能实现目标端与主机之间的通信。其中一个是我们之前在图2中讨论过的垫片层库，它用于拦截应用程序或安卓本身调用的所有 EGL/GLES API，并将目标端的所有 API 调用分派到主机。该方案完全遵循标准的安卓图形库加载机制。另一个库用于支持原生窗口处理程序（NWH）抽象层（libnwhal.so）。我们需要这一层公开原生窗口处理程序对象的内部信息，因为最终用户可能会直接将安卓原生窗口缓冲区指针传递给 glEGLImageTargetTexture2DOES 等 API。图5展示了注入 Android 目标端的两个库，它们使 GGA 和 GRM 能够正常工作。

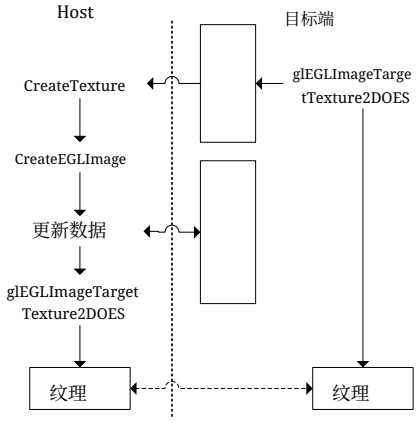


图5. 目标库注入

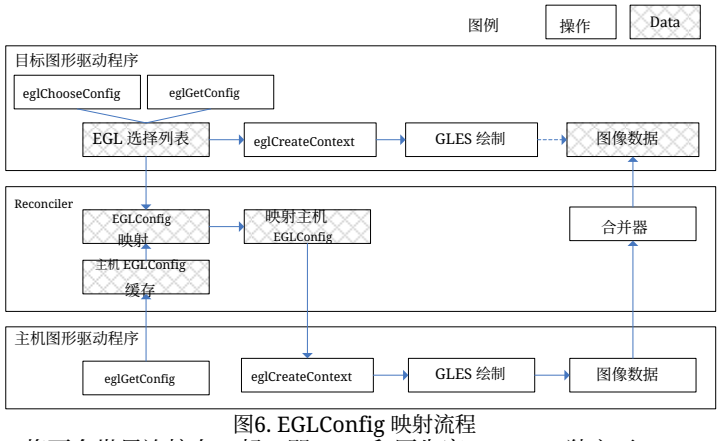


图6. EGLConfig 映射流程

名为 eglCreateWindowSurface 的 EGL API 将两个世界连接在一起，即 EGL 和原生窗口。EGL 独立于 GLES，是 OpenGL ES 与底层原生平台窗口系统之间的接口。通常，最终用户无需了解 EGL 世界中原生窗口处理程序的内部机制，而且 EGL驱动程序能够正确访问原生窗口缓冲区，因为它是原生图形驱动程序的一部分，能够了解安卓系统。然而，在我们提出的使用主机端API加速 3D 渲染的方案中，主机端必须了解目标端原生窗口缓冲区的内部结构，才能支持三个特殊 API：eglCreateWindowSurface、eglCreateImageKHR 和 glEGLImageTargetTexture2DOES。例如，API glEGLImageTargetTexture2DOES 支持从 EGLImage 为 EGL_NATIVE_BUFFER_ANDROID 类型的缓冲区定义整个二维纹理数组，这意味着主机端需要了解目标端当前原生窗口处理程序的布局。我们必须将原生窗口缓冲区数据从目标端复制到主机端，并创建一个纹理，以生成一个模拟的 EGLImage 来容纳

主机端中的相同内容，并最终使主机端的 glEGLImageTargetTexture2DOES 能够工作，从而获得与目标端相同的纹理映射。

III. 目标端图形驱动程序验证

借助 GGA 提供的目标库注入能力，可以将验证功能自由插入各种应用程序级调用中；而 GRM 则能够将验证逻辑注入内核、驱动程序，甚至硬件层。重要的是，这不会对渲染内容的视觉呈现造成明显影响，而且可以有效控制不断增加验证逻辑所带来的性能损耗。为全面验证图形驱动程序及系统集成，需要特别处理 EGL 接口，因为宿主端与目标端之间存在不一致。

3.1 错误码检查：从 O 的角度来看

penGL ES API 中，eglGetError 和 glGetError 是用于在运行时检查应用程序级问题，并据此决定后续代码路径的通用接口。不过，这些接口会对 3D 状态机造成一定干扰，并带来少许性能损耗；除非确有必要，大多数应用程序不会频繁执行此类检查。在 GGA 和 GRM 的执行过程中，每当 GLES 或 EGL API 调用完成后，我们都会立即在宿主端和目标端分别调用 egl/glGetError API。通过这种方法，我们能够获取所有 API 的错误代码，并将其缓存在两端；同时，egl/glGetError API 本身的实现也稍显复杂，因为它会改为使用缓存中的值。GRM 可配置开关可以启用或关闭此功能。启用后，工具集支持的白名单中所定义的 API 将在运行时接受错误代码比对检查。不过，这可能会大幅降低运行时速度，因此在完成正确性验证后可以将其关闭。这样一来，应用程序和游戏引擎开发者便可以随时自由地进行全部或部分错误检查，而无需更改应用程序源代码。当 3D 应用程序极其复杂时，这也可以成为 GPU 驱动调试的合适工具。如图 1 所示，此错误代码检查功能由验证模块提供。

3.2 渲染缓冲区目标分析 不同的安卓图形组件

通过 BufferQueue 连接起来，其中一些缓冲区由 Android gralloc 模块分配。在 Android 目标端，每个窗口表面在原生图形驱动程序内部依次执行以下流程：(a) 在目标驱动程序中 glDraw 开始时，从 BufferQueue 中 Dequeue 一块缓冲区。(b) 执行 glDraw，使所有 3D 渲染操作生效，并将渲染结果写入刚刚 Dequeue 的缓冲区。(c) 通过 eglSwapBuffer 将渲染器缓冲区 Enqueue 回 BufferQueue。要将渲染结果写回目标端的缓冲区，我们需要知道 glDraw 在步骤 (a) 中 Dequeue 的缓冲区地址。遗憾的是，该地址通常隐藏在图形驱动程序中，不会向最终用户公开。在不知道该地址的情况下，我们必须使用 DequeueBuffer 和 CancelBuffer 来获取缓冲区地址，但前提是，被取消的缓冲区能够很快通过后续的 glDraw 从 BufferQueue 中 Dequeue 出来。

不过，在 Fast Model 中，我们仍有机会通过分析图形驱动程序发送给图形硬件的 RenderTargetDescriptor 来获知此地址。这项特殊的分析操作在 GRM 内部实现；图形驱动程序发送给 GPU 的所有指令在 GRM 中都可见，因此我们可以拦截对图形设备的寄存器写入，并以主设备的身份通过 MMU 访问 CPU 中的内存地址，从而确定渲染目标缓冲区的地址。

在其他一些情况下，如果我们无法解析 RenderTargetDescriptor，另一种解决方案是使用名为 EGL_KHR_lock_surface3 的 EGL 扩展。借助该扩展，EGL 表面的颜色缓冲区可以映射到用户空间，随后我们便可将渲染结果写入这一公开的位置。不过，这可能不是理想的选择，因为在图形驱动程序中实现该扩展需要打断 GPU 渲染管线，因此性能可能会受到相应影响。

3.3 目标端扩展支持 为了实现良好的

衡 3D 资源管理，在移动系统上兼顾质量与性能，GPU 驱动程序通常会提供许多专用的 3D API 扩展。例如，EGL_ANDROID_image_crop 要求驱动程序对 3D 图像或纹理资源进行裁剪，而不是使用纹理变换矩阵；在某些情况下，这可以减轻计算负担。对于纯软件仿真，实现这些专用接口可能会带来额外开销；如果找不到特定扩展的支持，应用程序也可能会自适应地改变内部逻辑。而 GGA 可以捕获这些扩展并将其转发给主机端实现，后者能够提供一组统一的输入和结果，使目标端应用程序能够沿用相同的代码路径运行，与在真实设备上运行时保持一致。

3.4 EGLConfig 映射

如果启用了 GRM 执行，目标端和主机的图形驱动程序可能支持不同的功能。这意味着，双方无法同时支持的某些功能应予以舍弃，或转换为其他实现方式。不过，为了验证安卓中的软件栈，我们需要尽可能保留目标图形驱动程序的功能。EGLConfig 映射是渲染中最关键的问题之一。目标图形驱动程序可以支持各种 EGLConfig，例如 RGB565 以及许多其他功能，但有些主机图形驱动程序完全不支持这些功能。将目标端的 EGLConfig 映射到主机端的 EGLConfig 时，必须确保主机端的渲染结果与目标端具有相同的能力。否则，就需要进行仿真。例如，如果目标端的应用程序请求 RGB565 格式，而主机端不支持该格式，Reconciler 模块中的合并器可以为目标端调整帧缓冲数据，例如将主机端 4 字节的 RGBA8888 图像数据转换为目标端 2 字节的 RGB565 图像数据。

图6描述了 EGLConfig 映射流程。首先，调用 eglInitialize 时会缓存主机端的所有 EGLConfig。其次，目标图形驱动程序会为 eglChooseConfig 和 eglGetConfig 生成 EGLConfig 列表，同时将每个 EGLConfig 映射到最能支持目标端的主机端 EGLConfig。第三，映射后的主机 EGLConfig 用于主机端 GLES 绘制，最终的图像数据由合并器重新组织，然后写回目标端帧缓冲。

IV. 远程渲染加速

传统上，使用软件进行片上系统仿真速度非常慢，尽管使用 ARM Fast Model 生成的平台在典型工作站上可达到 100–500 MIPS 的仿真速度。许多片上系统开发厂商都在使用由数千台高性能机器组成的集群来加快验证工作。我们在第 3 章提出的解决方案要求在所有运行 Fast Model 的机器上安装显卡。要在集群中大量机器上实际启用如此多的显卡，是一项巨大的挑战。
因此，我们探索了一种新方案，通过网络远程加速集群中与 3D 渲染相关的图形计算。

初步实验显示，ARM Mali SDK 示例 Cube 在 GGA 上的渲染速度为 ~4帧率，该速度可能因不同的硬件配置而有所变化，因此这意味着，在分辨率为 1024x768 的显示器上，宿主端和目标端之间的内存缓冲区通信速度至少为 ~100Mb/s。此外，在宿主端针对目标端的每个窗口表面使用无损压缩和渲染缓冲区缓存，可以大幅减少内存数据传输量。对于千兆以太网而言，这完全不是问题，因此最终可以使用单个远程 GPU 资源，在集群中加速多个 3D 渲染验证实例。

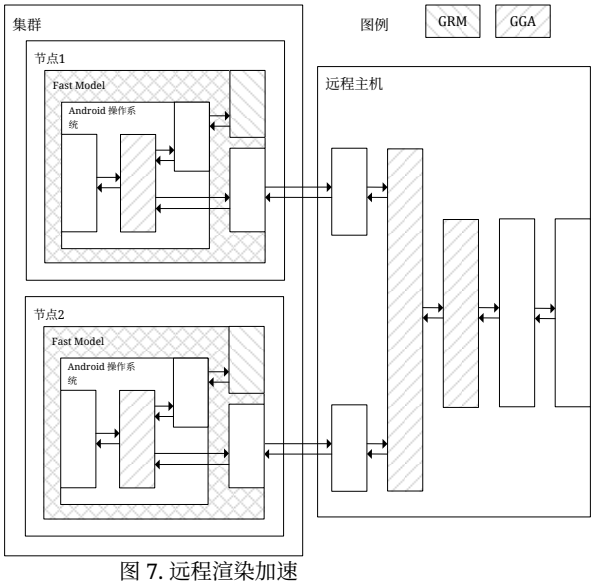


图 7. 远程渲染加速

图 7 说明了在 ARM Fast Model 环境中远程渲染加速的工作方式。首先，在 Android 目标端运行的应用程序所发出的 API 会直接进入垫片层。其次，这些 API 可以分发到目标图形驱动程序并进入 GRM，也可以通过 Fast Model SideChannel 插件传递给远程 GGA 协调器，该插件支持远程进程间通信。第三，在主机端，API 最终会通过 Mali 模拟器传递给主机图形驱动程序。渲染结果将在适当时机沿相反方向传回目标端，例如

在调用 eglSwapBuffer 时，或者由 GRM 在显示设备发出的 VSYNC 信号触发的某个时刻获取。我们将在第4.2章进一步讨论 eglSwapBuffer。

4.1 批处理模式

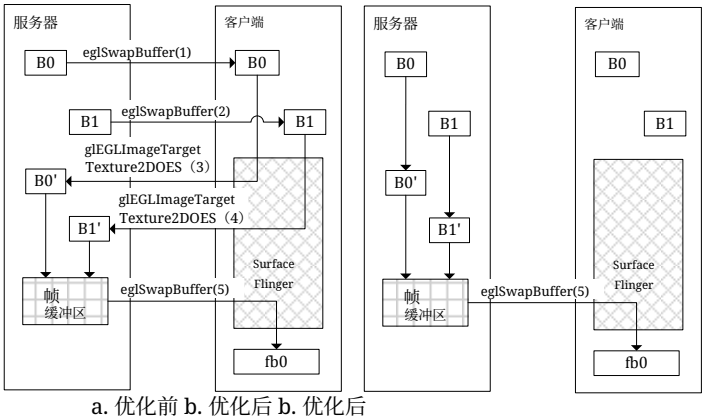
为进一步减少加速器与集群之间的网络流量，我们为 OpenGL ES API 引入了批处理模式。GLES API 可以先累积起来，直到遇到以下两种情况之一时再一次性发出：(a) 通过参数传递的指针返回内存；(b) 调用 eglGetError 或 glGetError，且需要将结果返回给应用程序。在实际运行基准测试时，情况 (b) 的出现频率非常高。大多数情况下，诸如商业基准测试之类的常规应用程序不会返回异常错误代码，因此我们可以始终返回无错误，以此绕过这一问题。采用这种方案后，主机端与目标端之间传输的 API 流量大幅减少。

4.2 帧缓冲区识别分析 如前所述，主机端的 G

LES API 执行必须依赖从目标端复制而来的内存缓冲区。实际上，主机端与目标端之间消耗内存带宽最多的是原生渲染缓冲区：对于每个可见窗口，都要从主机端读取并写入目标端。图 8.a 展示了 Surface Flinger 执行硬件合成时渲染缓冲区的读写方式。客户端指的是在 Fast Model 上运行的目标端，而服务器指的是执行实际 3D 渲染工作的远程主机。在客户端调用 eglCreateSurface 时，会创建一个原生缓冲区。与此同时，我们通知服务器为服务器端的 GPU 创建对应的缓冲区。服务器端完成渲染后，将调用 eglSwapBuffer，把内存复制回客户端，如 (1) 和 (2) 所示。图中以两个缓冲区 B0 和 B1 为例。当 Surface Flinger 收到来自显示设备的 VSYNC 信号而被触发时，软件合成操作会调用 GLES API glEGLImageTargetTexture2DOES，将客户端中的缓冲区 B0 和 B1 复制到服务器端的 B0’ 和 B1’。此时，就其中的数据而言，B0’ 等同于 B0，B1’ 等同于 B1。服务器端完成实际合成操作后，一帧

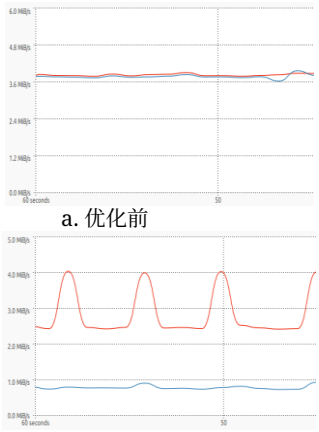
将生成符合显示尺寸的缓冲区，并调用 eglSwapBuffer 将其复制回客户端 Android 操作系统中的帧缓冲设备 fb0。如果我们在不进行优化的情况下，将客户端的所有 GLES API 调用都分发到服务器，这就是标准行为。

实际上，可以省去（1）（2）（3）（4）所述的内存复制，并通过减少不必要的内存复制，相应地降低巨大的内存带宽消耗。图 8.b 展示了这项优化。为了在服务器端识别 B0’ 与 B0 相同，且 B1’ 与 B1 相同，我们会以客户端的身份检查目标端的显示尺寸。如果渲染缓冲区的尺寸恰好与显示器所用的尺寸一致，我们就认为该渲染缓冲区是用于硬件合成目标的，也就是设备 fb0。采用这一方案，可减少约 3⁄4 的帧缓冲区复制。图 9.a 展示了优化前的网络带宽，因此上行和下行的数据量几乎相同。优化后如图 9.b 所示，上行数据减少了 75%。该方案适用于大多数实际情况，但在某些边界情况下仍有缺陷。我们将在第 6.2 章中进一步讨论。



a. 优化前 b. 优化后 b. 优化后

图8。渲染缓冲区访问



a. 优化前

图9。远程建模网络流量

V. 实验与性能

本节介绍了一组实验结果，旨在展示在 Fast Model 上运行的图形应用程序的 3D 渲染加速效果。所有实验均在 Intel(R) Xeon(R) CPU E5- 2690 v4 @ 2.60GHz 上运行。实验评估采用了商用 GPU 性能基准测试

安兔兔、GFX 和 Unity。我们还测量了安卓启动过程，以比较 Google 提供的软件渲染的效果。对于所有这些基准测试，我们都可以在安卓图形用户界面中查看运行时执行结果。

5.1 性能比较 图10给出了性能

比较软件渲染、GGA 和 GRM 执行情况。软件渲染无法支持真正的 OpenGL ES API 来运行大多数商业基准测试，只能使安卓成功启动。

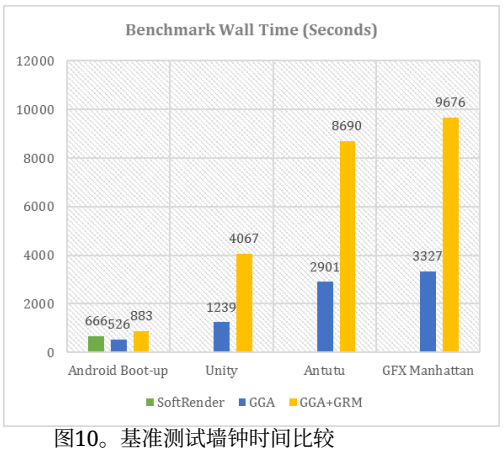


图10。基准测试墙钟时间比较

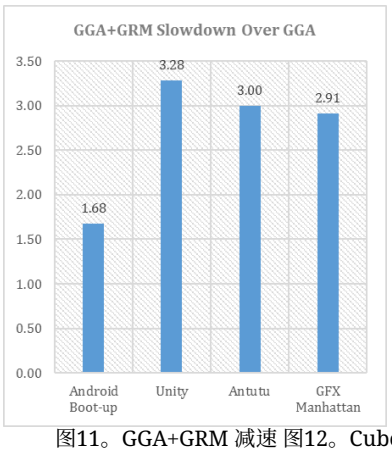


图11。GGA+GRM 减速

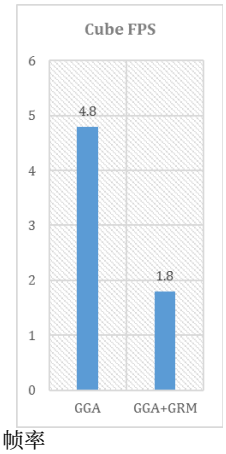
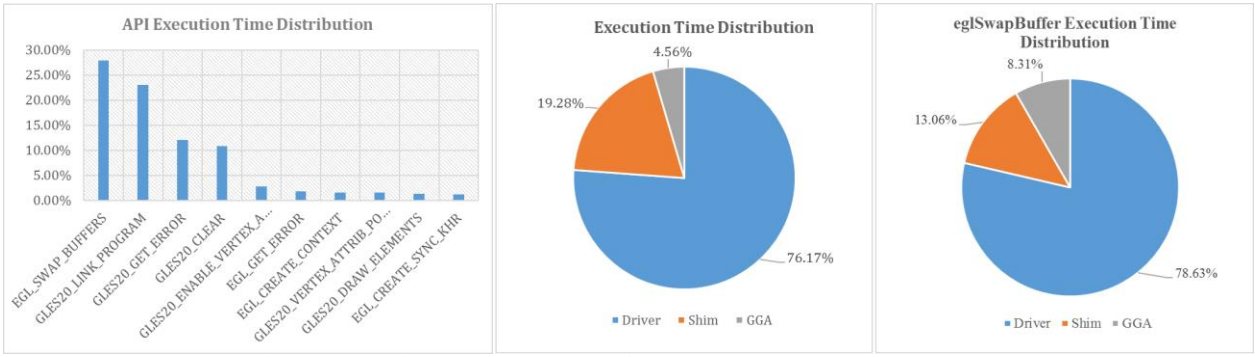


图12。Cube 帧率

图11展示了运行真实基准测试时，GRM 执行使 Fast Model 变慢的程度。这 4 个典型基准测试的平均值为 2.72。GRM 执行不会改变 GGA 在 3D 渲染中的行为，也不会为 GRM 增加新的开销，因此减速主要源于目标端图形驱动程序的执行，以及安卓 Android SurfaceFlinger 中相关行为的变化。图12给出了运行 Mali Cube示例时的帧率，该示例来自 Mali图形SDK。帧率减速比为 2.67，与墙钟时间减速比非常接近。

5.2 GRM 的性能瓶颈

图13展示了在 Mali Cube 上的执行时间细分。图13.a 列出了耗时占比为>1%的所有 API，其中 EGL API eglSwapBuffers 的执行时间最长；从图13.c 可以看出，目标端驱动程序执行此 API 的耗时最长。这主要是因为 eglSwapBuffers 会隐式刷新上下文，并给 Fast Model 带来巨大的内存复制压力。图13.b 显示，大部分执行时间都花在目标端的图形驱动程序 API 执行上。



a. API 细分 b. 按模块细分 c. eglSwapBuffer 细分

图13. API 执行时间分布

VI. 结论与未来工作

本文所述工作基于 ARM Fast Model 产品的图形模型开发。利用主机资源进行 3D 渲染加速，可以显著提高图形软件栈验证速度，无论该资源位于本地还是远程。借助本文提出的技术，我们可以运行安兔兔、GFX 和 Unity 等真实基准测试，在不对原生图形驱动程序和 Android 操作系统构建进行任何更改的情况下，验证软件栈完整性。因此，在 Fast Model 上完成验证的映像无需任何更改，即可直接在真实芯片上进一步验证。与 Android开源代码库提供的软件渲染相比，ARM Fast Model 的图形模型凭借高易用性、高性能和高效率，打破了片上系统图形技术验证的僵局

此外，所提出的技术还可以通过展示 GPU 与 CPU 之间的内存流量，帮助开展图形学研究。除了这些优势之外，我们也期待在未来解决一些潜在问题。

6.1 直接内存写入 最棘手的问题

在我们当前的解决方案中，最棘手的问题是目标端与主机之间需要进行大量内存复制。这是因为目标端和主机处于完全不同的地址空间。通常，传输内存的方式是使用 Fast Model 侧信道插件，它本质上是目标端与主机之间通信的一条后门。在写入目标端之前，我们会先访问待写入的内存，以确保该页确实位于目标端的内存中。完成这一步后，由主机传递的内存写入操作必须经过 Fast Model 模拟的直接内存访问总线，最终将渲染缓冲区内存写入目标端的物理帧缓冲区。实际执行时，这会形成一条很长的执行轨迹，从而拖慢 Fast Model 的执行速度。由于 Fast Model 子系统能够识别所有受支持的设备，因此可以通过主机端的一条特殊后门公开物理帧缓冲区地址。但问题在于，该地址指向的内存大小不能超过单个页，也就是说，我们只能保证与主机共享的目标端内存在一个页内连续。如果能够始终保证跨页连续，就可以将渲染结果直接写入目标端的共享内存，避免内存写入绕行，从而大幅提升性能。

6.2 帧缓冲区身份检测：第4章所述的解决方案。

2 并非总是有效。在某些边界情况下，如果目标端的渲染缓冲区尺寸与显示尺寸相同，但该缓冲区并非表面硬件合成的目标，则此优化将不再有效。我们如何才能在服务器端始终正确判断 B0’ 是 B0、B1’ 是 B1？ 如图8所示

。我们无法在服务器端看到 B0/B1 与 B0’/B1’ 之间的显式关联，但它们通过客户端的原生缓冲区 B0/B1 隐式关联。因此，我们需要一种方案来告知服务器这种关联，或者利用帧缓冲区的特征推断这种关联。遗憾的是，在用户空间中，没有完美的方案能够确定 B0/B1 是否与 B0’/B1’ 是同一个缓冲区。为了解决这个问题，我们必须在 Android 操作系统中构建并安装一个小型文件描述符查询驱动，以检查 B0/B1 与 B0’/B1’ 是否可以相互映射。在这个驱动中，我们可以通过用户空间提供的文件描述符找到驱动程序中的隐藏数据结构，从而轻松获知用户空间中暴露的缓冲区之间的关系。

在图 14 中，虚线表示通过目标端安装的驱动程序查询唯一的内核缓冲区 ID。这样，我们就能利用该唯一 ID 检查目标端分配的缓冲区之间的关系，进而判断是否确实需要将原生缓冲区从客户端复制到服务器端。如果发现服务器端已有缓冲区缓存，就无需这样做，相应地，也不必将缓冲区从服务器端复制回客户端。服务器端的所有渲染工作都将基于缓冲区缓存进行，我们只会将专门映射到客户端设备 fb0 的帧缓冲区写回。因此，可以节省大量内存带宽。

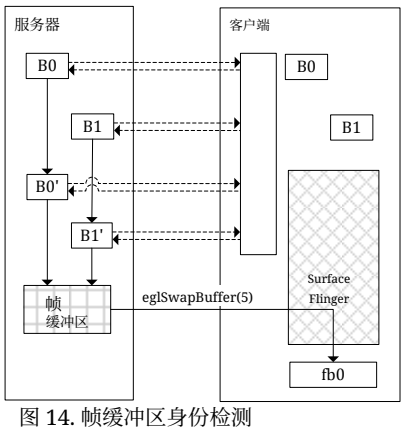


图 14. 帧缓冲区身份检测

6.3 多媒体开发与测试 支持

And 由以及然 Google 而的视频播放器内容流媒体服务（例如 Netflix）。这些视频播放应用大多使用由 Stagefright 支持的 Android Java MediaPlayer API。Stagefright 能够通过 OpenMAX IL（集成层）将视频解码卸载到专用硬件模块，并执行会话管理、时间同步渲染和数字版权管理等功能。与 QEMU 内置的 GLES CM 1.0 软件渲染相比，GGA 可以大幅加快渲染工作。更重要的是，供应商通常会使用 OpenMAX IL 标准设计自己的硬件编解码器，该标准为 Stagefright 识别和使用定制的基于硬件的多媒体编解码器提供了

一种标准化方式。所提出的方法有助于在计算子系统（CSS）中整体验证与 Stagefright 的集成，并更好地支持 Android 媒体兼容性测试套件（CTS）提供的自动化测试框架。

6.4 开启 GRM：即时执行 GRM 总是很慢

这会降低渲染加速，因为我们必须与模型上的 API 执行同步。纯 GGA 模式是在 Fast Model 中验证图形软件栈时所能达到的最高速度。如果能够动态开启或关闭 GRM 执行，就会好得多；最终用户只需对自己感兴趣的执行开启 GRM。如果在任意时刻开启，图形驱动程序可能会漏掉某些 API 调用，导致驱动程序中的状态与主机上的 GGA 开始不一致。例如，如果驱动程序没有看到 glGenTexture 调用，却随后看到了 glTexImage2D 调用，就会报错。不过，仍然可以只在进程生命周期范围的边界处开启或关闭 GRM 执行。通常，目标端的不同进程会进行初始化以创建不同的 GLES 上下文，因此我们可以轻松地在目标端和主机上分别做出调整，根据进程 ID 动态决定是否需要执行 GRM。只要确保单个上下文内的渲染完整性，GLES 状态就不会被破坏。

6.5 Gralloc-ION 相关的未来工作　在典

型的运行安卓的片上系统中，有许多硬件模块需要管理缓冲区，并在驱动程序和进程之间共享缓冲区。Gralloc 和 Android ION 框架定义了相关准则，可据此灵活定制各内核驱动程序之间的缓冲区共享方式。GPU 的 GGA 和 GRM 执行可帮助这些多媒体内核驱动程序顺利运行，并更系统地测试软件流水线，进而支持在 Fast Model 中进行完整的系统验证。此外，基于 Android fence 的同步机制是协调媒体设备协作的关键，其中包括视频解码器、显示控制器、HWComposer 使用的任何硬件，以及 GPU 驱动程序。GGA 和 GRM 执行可从用户空间和内核空间的角度，帮助确定执行此类同步的关键位置。这对于希望集成多个媒体设备或调整跨设备技术（例如，利用 ARM 帧缓冲压缩（AFBC）协议，尽量减少片上系统内 IP 模块之间传输的数据量）的供应商而言，极具价值。

VII.　致谢

本文所述工作得到了 ARM Ltd. 赞助的 Fast Model 项目支持。作者还要感谢 Jesse Zhou（ARM）对这项工作提出的建议和反馈。

VIII.　参考文献

[1]　Arm 建模与 ESL. http://infocenter.arm.com/help/topic/com.arm.doc.set.models/index.html[2]　Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, James T. Klosowski, “Chromium：一种用于集群交互式渲染的流处理框架” SIGGRAPH '02 第29届计算机图形与交互技术年会论文集，第693-702页[3]　Khronos 组织发布的 OpenGL ES 规范。 https://www.khronos.org/registry/gles/[4]　Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, David A. Wood, “gem5-gpu：一种异构 CPU-GPU 模拟器”，《计算机体系结构快报》第13卷第1期，2014年1月[5]　Ashu Rege, 《现代 GPU 架构导论》，ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf[6]　Shivani Raghav, Christian Pinto, Martino Ruggiero, “使用 GPU 和 QEMU 半主机对多核异构片上系统进行全系统仿真。” GPGPU-5，2012年3月3日，ACM。 [7]　Youngsub Ko, Youngmin Yi, Joongbaik Kim 和 Soonhoi Ha, “面向安卓图形应用的 OpenGL ES API 级快速 GPU 闭环仿真技术。” 快速系统原型设计（RSP），2015年国际研讨会 [8]　ARM 周期模型。 https://developer.arm.com/products/system-design/cycle-models[9]　Andreas Sandberg, “NoMali：使用桩 GPU 理解软件渲染的影响”，http://www.gem5.org/wiki/images/5/53/2015_ws_04_ISCA_2015_NoMali.pdf[10]　Rafael Ubal, Byunghyun Jang, Perhaad Mistry, “Multi2Sim：一种用于 CPU-GPU 计算的模拟框架。” PACT’12，2012年9月19日至23日，ACM。[11]　Android 图形。 https://source.android.com/devices/graphics/[12]　ARM Mali 仿真器。 https://developer.arm.com/products/software-development-tools/graphics-development-tools/opengl-es-emulator