doi:10.3772/j.issn.1002-0470.2020.05.001

高性能 GPU 模拟器驱动设计研究①

赵士彭②[*,**,***] 张立志[*,**,***] 赵皓宇[****] 苏孟豪[*****] 刘 苏[*****]

([*]计算机体系结构国家重点实验室(中国科学院计算技术研究所) 北京 100190)

([**]中国科学院计算技术研究所 北京 100190)

([***]中国科学院大学 北京 100049)

([****]中国科学技术大学计算机科学与技术学院 合肥 230026)

([*****]龙芯中科技术有限公司 北京 100190)

摘 要 在研究 GPU 驱动设计时,考虑到驱动的开发量较大,提出了一种基于 Mesa 开源驱动框架的 GPU 模拟器驱动设计方法,在 Mesa 开源驱动的开发框架下,实现了一整套高性能 GPU 模拟器的驱动设计。本设计可适配 OpenGL 等多款应用程序接口(API),同时基于开源框架,大幅减小了开发难度。为了适配 GPU 模拟器的可编程性,在驱动设计中还集成了 GPU 的编译器设计,可将 GLSL 等编程语言转换为 GPU 中着色器的汇编代码,由着色器进行计算。根据模拟器驱动设计,提出了一套高性能 GPU 模拟器的接口设计,为模拟器各个模块提供了一套可读的驱动接口,指导了模拟器的结构设计。

关键词 开源驱动;应用程序接口(API);驱动设计;GPU 模拟器;模拟器接口

0 引 言

目前,图形处理单元(graphics processing unit,GPU)已经广泛存在于个人电脑、移动设备之中。GPU 作为图形加速器,在游戏、通用计算、图像处理等领域都起到了十分显著的作用。在现代个人计算机系统中,GPU 已经变得越来越不可或缺[1],且变得越来越复杂[2-4]。中央处理器(central processing unit,CPU)[5]不再独立处理图形工作,将需要处理的图形工作转交给 GPU 执行,从而大幅提高系统的整体性能。GPU 一般以插卡的方式,通过主板上的图形加速接口(accelerated graphics port,AGP)或高速串行计算机扩展总线标准(peripheral component interconnect express,PCIe)插槽与 CPU 进行通信。GPU 目前已经逐渐面向通用,通用 GPU(GPGPU)[6]已经在通用计算、机器学习、人工智能等相关领域发挥不可替代的作用。当前的 GPU 支持 DirectX、OpenGL、OpenGL ES 等应用程序接口(application programming interface,API)。DirectX[7]用于 Windows 操作系统,OpenGL[8]用于 Linux 操作系统,OpenGL ES[9]用于安卓和 IOS 操作系统。这些 API 都需要 GPU 的驱动进行处理,转换为 GPU 硬件可识别的状态和命令等。随着 GPU 越来越通用,当前的 GPU 已经拥有了统一的可编程着色器(shader),这使 GPU 已经成为一种可编程处理器。统一的可编程 shader 使 GPU 的可编程性和灵活性大大提高,与此同时,驱动也自然承担起 GPU 编译器的任务[10,11]。GPU 其他功能部件的可配置性也越来越高,这些都需要驱动负责处理。作为 CPU 与 GPU 之间交互的桥梁,驱动还负责 GPU 的任务管理调度等相关工作,这在 GPU 研发工作中必不可少。目前 GPU 的驱动多为闭源驱动,各硬件厂商根据自身 GPU 硬件进行设计。鉴于驱动的代码量巨大,难度

① 国家自然科学基金(61521092,61432016)和中国科学院重点部署(ZDRW-XH-2017-1)资助项目。

② 男,1993 年生,博士生;研究方向:计算机系统结构,处理器设计;联系人,E-mail:zhaoshipeng@ict.ac.cn

(收稿日期:2019-06-10)

很高,本文提出了一种基于 Mesa 开源驱动框架的驱动设计方法,本方法可以借用 Mesa 开源驱动,实现模拟器的驱动设计,大幅度减小开发难度并减少代码量。

1 研究背景

1.1 3D 图形流水线

3D 图形流水线是根据 OpenGL、DirectX 等 API 给出的图形渲染流程,它们指导了渲染的每一个步骤。流水线涉及 GPU 中的渲染算法,描述了从图形 API 到点,再到图元,最后变为 2D 图像输出到帧缓冲区的整体过程。首先,从图形 API 中获得顶点数据流,之后,将这些数据转换为相应的图元,并通过光栅化阶段变为 2D 像素块用做最终显示。文献[12-14]对 GPU 的结构和流水线做了更为详细的介绍。

图形流水线的渲染包括以下阶段。

顶点获取 顶点获取阶段是从图形 API 中获取包括顶点的位置、属性以及纹理等相关信息,并将相关信息传递给后面的流水线。

顶点着色器 顶点着色器阶段是将顶点的位置、属性等信息通过矩阵运算等方式转换为对应的屏幕坐标和需要的属性,并交给图元组装阶段。

图元组装 图元组装将顶点着色器输出的位置信息与图元连接信息对应,将顶点组装成对应的图元,例如点、线或三角形等。并将图元交给光栅化阶段。

光栅化 光栅化阶段是将图元变为独立的像素点,即将 3D 的图元变为 2D 的像素块,同时计算每个像素块的插值系数,并将这些信息发送给片段着色器。

片段着色器 片段着色器为每个像素针对插值系数进行插值运算,使每个像素块都拥有自己的属性等信息。

输出混合 输出混合阶段是将每个像素块进行透明度、模板和深度等相干测试,剔除掉不需要显示的像素块。之后经过混合后输出到帧缓冲区进行显示。

1.2 Mesa 开源驱动架构

Mesa[15]是一款开源的用户级驱动,经过多年的发展,已经支持 OpenGL、OpenGL ES、OpenCL[16]、Vulkan 等多种 API。在硬件驱动方面,Intel、Nvidia、AMD 等厂商的多款 GPU 已经适配。如今,Mesa 已经成为 Linux 操作系统中使用较为广泛的用户级显卡驱动之一。

Mesa 的主要任务是将 API 的数据及状态处理为硬件驱动需要的命令、状态、指令等。当前的 Mesa 为了方便驱动开发者对不同的设备进行驱动开发,在原驱动架构中抽象出了 Gallium[17]架构。Gallium 架构是在原 Mesa 驱动中抽象出与硬件相关的部分。基于 Gallium 架构,驱动开发者可无需重复开发与硬件无关的代码,大量减少了开发驱动的代码量及开发难度。

目前,Mesa 驱动的架构主要分为 3 大部分,即 API 层,硬件层和系统层。图 1 是 Mesa 开源驱动架构的结构图。API 层是 Mesa 驱动中负责处理 API 接口的位置,包含 Mesa 和 state tracker 两部分,这两部分的实现均与硬件无关。经过 API 层后,API 的数据和状态会被转换为硬件层中与硬件无关的接口。硬件层的主体是 Gallium 架构。经过了硬件层的处理,需要渲染的数据和状态已经转换为硬件可识别的命令、状态和指令等。这些数据会经由系统层与内核级驱动(direct rendering manager,DRM)交互,将有关数据传入 CPU 与 GPU 的共享内存上。这样,GPU 就可以通过命令处理器(command processor,CP)获取需要的数据。

图 1 Mesa 开源驱动架构

2 高性能 GPU 模拟器

本文使用的高性能 GPU 模拟器主要由命令处理器(command processor, CP)、全局任务调度器(global task scheduler, GTS)、图形处理集群(graphics processing cluster, GPC)、二级静态缓存(L2 Scache)、内存控制器(memory controller, MC)5 部分组成。其中图形处理集群又由计算处理引擎(compute engine, CE)、几何处理引擎(geometry engine, GE)、图元处理引擎(primitive engine, PE)、局部任务调度器(local task scheduler, LTS)、流处理器集群(stream processor cluster, SPC)、输出合并单元(output merge unit, OMU)6 部分组成。整体结构如图 2 所示。

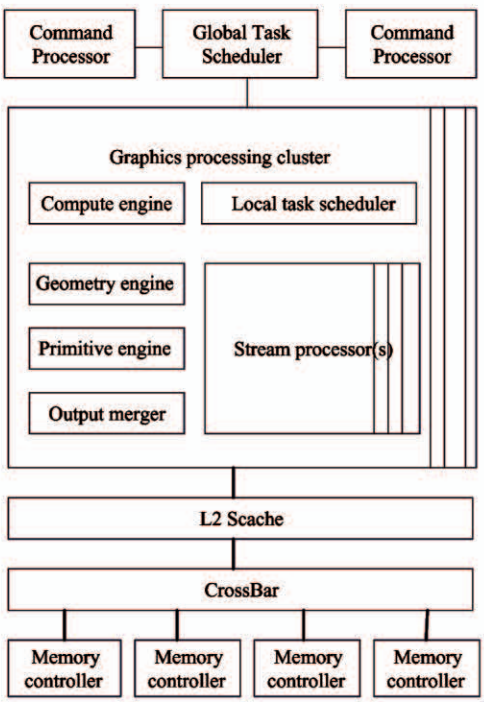

图 2 高性能 GPU 模拟器的顶层结构

本文使用的高性能 GPU 模拟器是一个由 C++ 语言编写的 GPU 模拟器。其设计目的是对 gsgpu 结构设计做功能验证与一定程度的性能分析,该结构后续还将使用现场可编程门阵列(field progrmmable gate arry, FPGA)进行更深一步功能验证,最终流片,所以在整个 GPU 设计计划中,模拟器只是一个阶段性产物,而不像 ATTILA 模拟器作为最终实验平台,所以简单高效地实现 GPU 模拟器是当前计划中很重要的标准。GPU 模拟器整体软件架构由 Module 与 Queue 2 个数据结构搭建。Module 构建每一个功能模块,类似于 Verilog 语言中 module 模块。Queue 构建各个 module 间通讯模块,同时实现各 module 的解耦合,Queue 本质是一个每周期一写一读的循环队列。

3 高性能 GPU 模拟器驱动

3.1 模拟器驱动架构

高性能 GPU 模拟器的驱动需要为模拟器提供数据、命令、状态以及 shader 需要的指令等信息。这些信息是驱动从 API 中获得并通过处理后得到的。驱动的设计共分为 3 个阶段。首先,API 经过 Mesa 的 API 层转化为 Gallium 架构下与硬件无关的状态,同时将 OpenGL 着色器语言(OpenGL shading language, GLSL)转化为 TGSI(Tungsten graphics shader infrastructure)[18] 中间语言;然后,硬件层将硬件无关状态转化为与硬件相关的状态,同时产生硬件执行所需的命令等信息,并将 TGSI 中间语言转化为 GPU 模拟器的汇编;最后,将每一帧的信息处理为硬件需要的可读的接口信息,同时将信息传入模拟器的显存中等待执行渲染。

驱动设计中,Mesa 的硬件层是设计的重点部分,硬件层的设计也同时指导了模拟器接口的设计,对模拟器的设计起到了指导作用。Mesa 的 API 层是与硬件无关的部分,是可重用的部分,所以在设计中采用了 API 层的结构。Mesa 驱动的系统层中与 DRM 交互的部分,在 GPU 模拟器中也无需实现,所以没有采用。

3.2 硬件驱动设计

硬件驱动的设计主要是在 Mesa 驱动的 Gallium 架构下实现的。在 Mesa 的 API 层中,API 的数据及状态已经转换成了 Gallium 架构中与硬件无关的接口。在硬件驱动的设计中,需要考虑绘制进入接口、状态对象接口、缓存及中断接口和刷新接口等。

首先,驱动为 GPU 模拟器中的命令处理器准备命令队列,命令队列是用于控制 GPU 模拟器执行整

体渲染的一组命令。驱动为 GPU 模拟器准备的命令队列设计为 draw、sync、dump 和 halt 4 个主要命令。

Draw 命令是一条渲染命令,也是整个渲染的核心命令。当执行 draw 命令时,GPU 模拟器会调用已经准备在显存中的 draw _ info、pipe _ state 和 shader _ state 等一些必要的状态。命令处理器准备好各状态后,会将顶点、流水线状态等信息通过全局任务调度器逐个发给几何计算引擎。几何计算引擎执行顶点子块处理任务后,LTS 做 SPC 资源管理,查找合适的 SPC,并把线程组任务发过去。然后,SPC 执行顶点着色器(vertex shader,VS)程序,GE 跟踪 VS 的位置输出,按原始顶点顺序对应的微块顺序,读出顶点位置数据,并进行广播。之后,PE 对图元进行光栅化,LTS 根据资源情况,查找合适的 SPC,把线程组任务发送过去,SPC 执行片段着色器(fragment shader,FS)程序。最后,PE 将排完序的片段结果送到 OMU,OMU 进行后处理。

Sync 命令是一条同步命令,用于同步 GPU 模拟器中的各个模块。在整条流水线中,各模块之间会依次传递一个 last 状态。当模块收到 last 状态,则说明本次 draw _ call 渲染已经结束,模块会向命令处理器返回已经处理结束的信号。当执行 sync 命令时,命令处理器会进入等待状态,不会继续执行命令队列中的其他命令。只有当收到所有模块处理结束的信号后,才继续执行命令队列中的其他命令。

Dump 命令是一条显示命令,用于将已经渲染完成的 draw _ call 显示出来。当命令处理器执行 dump 命令时,命令处理器会调用显存控制器,将显存中帧缓冲区渲染好的像素显示到屏幕上。

Halt 命令是停机命令,用于停止 GPU 模拟器的所有工作。当命令处理器执行 halt 命令时,整个 GPU 模拟器将停止所有模块的执行,GPU 模拟器进入停机状态。

其次,为了使几何计算引擎和顶点着色器准确获取顶点和属性信息,驱动需要配置与顶点和属性相关的状态。Draw _ info 是驱动配置的一次 draw _ call 的相关信息,包括:起始顶点 start,用于索引提前准备在显存顶点区域的第一个顶点位置;本次 draw _ call 的顶点数量 count,用于索引显存中顶点区域用于本次渲染的顶点。每个图元的顶点数,用于图元处理引擎进行图元组装;index _ buffer 用于顶点使用索引缓冲的情况下进行顶点在显存中位置的描述。

为了增强 GPU 模拟器固定管线的可配置性,驱动需要配置相应状态给对应的固定管线模块。这些配置信息就是 API 层传下来与硬件无关的状态。这些状态的接口是 Gallium 架构中的 pipe 层。在高性能图形 GPU 中,有 2 个主要的固定管线,一个是图元处理引擎,用于图元的组装以及光栅化;另一个是输出合并单元,用于像素的测试和混合。本设计将 pipe 层的状态处理为供图元处理引擎使用的光栅化及裁剪等状态,以及供输出合并单元使用的深度测试状态、模板测试状态和混合状态。这些状态被转换成 32 位的二进制,传入 GPU 模拟器的显存中。

Rasterizer _ state 是用于 PE 使用的状态,包含了光栅化所需要的点大小、线宽、多重采样等所有配置信息。viewport _ state 和 clip _ state 也是用于 PE 使用的状态,告知 PE 视口的大小以及裁剪的大小,以便光栅化使用。Alpha _ state、depth _ state 和 stencil _ state 是 OMU 使用的状态,分别用于 OMU 的透明度测试、深度测试和模板测试。驱动将对应的测试算法告知 OMU,由 OMU 调用深度和模板的缓冲完成所有测试,3 种测试分别有 7 种不同的测试算法可以配置。Blend _ state 也是 OMU 使用的状态,驱动将混合算法告知 OMU 后,OMU 调用渲染目标(render target)进行混合后,将结果写入帧缓冲区(framebuffer)等待最后的显示,其中,混合共有 19 种不同算法。

为了进一步提高 GPU 模拟器的可编程性,GPU 模拟器采用了统一的可编程 shader。这就需要驱动为可编程的 shader 准备相应的状态以及所需寄存器个数等信息。本设计中,shader _ state 对标量通用寄存器和向量通用寄存器的数量等信息进行了描述。这些信息由驱动根据 shader 执行的汇编程序及其状态产生。

3.3 编译器设计

随着 GPU 的可编程性越来越高,驱动中对于

GPU 编译器的设计也越来越重要。在 Mesa 驱动架构中,Mesa 提供了 GPU 编译器的前端代码。通过 Mesa 驱动中的编译器前端,可以将 GLSL 等编程语言翻译为一种叫 TGSI 的中间语言。TGSI 语言是 Mesa 为所有的 GPU 提供的唯一中间语言。本设计主要针对 TGSI 的中间语言进行进一步的编译器后端设计。表 1 中是一个彩色正方形程序,通过 Mesa 的 API 层编译成的 TGSI 中间语言。表中,VERT 代表当前是顶点着色器对应的程序,PROPERTY NEXT _ SHADER FRAG 代表对应的下一个 shader 程序是片段着色器程序。DCL 声明了有 2 个输入,IN0 是顶点位置信息,IN1 是顶点属性信息。同时还声明了 2 个输出 OUT0、OUT1,一个常量缓存 CONST0 和一个临时缓存 TEMP0。之后进行 4 ×4 的矩阵乘法,MUL 表示乘法运算,MAD 表示乘加运算,之后将结果输出到 OUT0。IN1 直接输出给 OUT1。基于 Mesa 框架,模拟器驱动将 TGSI 中间语言根据语法语义进一步编译为 GPU 模拟器的汇编代码。

表 1　正方形在 Mesa 中翻译为 TGSI 中间语言

正方形在 Mesa 中的 TGSI 中间语言
VERT
PROPERTY NEXT _ SHADER FRAG
DCL IN [0]
DCL IN [1]
DCL OUT[0], POSITION
DCL OUT[1], COLOR
DCL CONST[0][0..3]
DCL TEMP[0]
MUL TEMP[0], IN[0].xxxx, CONST[0][0]
MAD TEMP[0], IN[0].yyyy, CONST[0][1], TEMP[0]
MAD TEMP[0], IN[0].zzzz, CONST[0][2], TEMP[0]
MAD OUT[0], IN[0].wwww, CONST[0][3], TEMP[0]
MOV OUT[1], IN[1]
END

表 2 是将 TGSI 中间语言的 4 ×4 矩阵乘法部分转换为 GPU 模拟器汇编后的代码部分。这部分是将原 TGSI 的 MUL 和 MAD 构成的矩阵乘法变为 GPU 模拟器的汇编语言,可编程的 shader 将执行这段汇编程序完成矩阵的乘法运算。其中 MUL 是进行乘法运算,MAD 是进行乘加运算。S 代表标量通用寄存器,是从常量缓存 CONST0 取出的数据,是驱动准备好的 4 ×4 矩阵。V 代表向量通用寄存器,是顶点的位置信息,由输出传入。

表 2　4X4 的矩阵乘法翻译后的 GPU 汇编

翻译后的 GPU 汇编语言
MUL V8,S0, V4
MUL V9, S1, V4
MUL V10, S2, V4
MUL V4, S3,V4
MAD V8, S8, V5
MAD V9,S9,V5
MAD V10, S10,V5
MAD V4, S11,V5
MAD V8, S12, V6
MAD V9, S13,V6
MAD V10, S14, V6
MAD V4, S15, V6
MAD V8, S4, V7
MAD V9,S5,V7
MAD V10, S6, V7
MAD V4, S7,V7

3.4　模拟器的驱动接口设计

当所有数据、状态、命令和 shader 指令准备好之后,都将传入到显存之中,以便 GPU 模拟器根据命令队列进行执行。根据前述的驱动设计,本文提出了高性能 GPU 模拟器的接口设计,用于指导模拟器的结构设计。该接口可将每一帧中驱动写入 GPU 模拟器的所有信息变为可读信息,更加方便了后续的调试。同时该接口还可对驱动传入 GPU 模拟器的信息进行修改,也可提高调试模拟器的灵活性,更加方便地找出问题所在。

该接口将所有信息分为命令、绘制和数据 3 大部分。命令部分反映 GPU 的命令处理器中命令队列,是 GPU 流水线的执行状态。绘制部分包括 draw _ info、pipe _ state 和 shader _ state 3 部分。draw _ info 反映的是模拟器中 GE 的执行状态,pipe _ state 反映的是 PE 和 OMU 的执行状态,shader _ state 反映的是模拟器中 SPC 的执行状态。数据部分则包括了顶点数据、纹理数据和 shader 指令等。Shader 的指令在显存中是二进制的形式存在,本接口还将指

令进行了反汇编，更加增强了调试者的可读性。

通过这些可读的信息，在调试模拟器的过程中可以更加方便地知道流水线中各级的执行状态。这些信息以 draw _ call 为单位，每次可显示一帧的数据。同时，通过修改这些接口信息，也可以直接改变模拟器的执行状态，更加方便调试人员调试模拟器。

4　实　验

本实验使用的操作系统为 Fedora 28. X86 _ 64，采用的 CPU 为 Intel Core i7 3770，主频 3. 4 GHz。GPU 实验平台采用一种高性能 GPU 模拟器，Mesa 版本为 18. 0. 5，可支持图形 API 的 OpenGL ES 3. 0 版本。

实验所用基准测试集为 GL _ mark。在 Linux 操作系统上，GPU 的图形基准测试集并不多。GL _ mark 是由 Linaro 发行的一款图形基准测试集，使用 OpenGL-ES 进行开发，提供了一系列丰富的图形测试，涉及图形单元性能的各个方面，涵盖光照、阴影、超多图元、简单 2D 等多种类型的测试，是目前 Linux 操作系统上较为全面的测试集之一。

图 3 是实验的测试步骤，Mesa 开源驱动将 GL _ mark 基准测试集所使用的 OpenGL-ES 的 API 转化为相应的中间状态，同时将 GLSL 的 shader 编程语言转换为 TGSI 中间语言。GPU 驱动将中间状态及中间语言转化为 GPU 模拟器的接口信息和底层汇编语言传递给 GPU 模拟器驱动。经过 GPU 模拟器的一系列处理，最终以帧的形式显示出来。

图 3　GPU 驱动验证的实验过程

本文在 GL _ mark 中选择的基准测试描述如表 3所示，从 GL _ mark 基准测试集中选择了 4 个阴影测试，分别用于测试 2D 显示、反射及阴影测试、折射测试以及复杂 shader 测试。Draw _ call 数量包括清除深度缓存和模板缓存、多重采样等操作。图 4所示是 2D 显示测试的效果图。2D 显示时，顶点数为 4 个，即正方形的 4 个顶点。图中图案是使用纹理贴图实现的。通过该测试表明 GPU 模拟器可以利用 3D 图形驱动配合 3D 图形模拟器完成 2D 的显示。图 5 所示是反射及阴影绘制的测试，通过该测试表明 GPU 模拟器驱动可准确完成反射及阴影的绘制。图6所示是折射绘制的测试，通过该测

表 3　选取的基准测试集

GL _ MARK 测试用例	测试用例特点	顶点数	每帧 draw _ call 数量
Effect2D	3D 模拟器渲染 2D 测试	4	3
Shadow	反射及阴影渲染的测试	21 516	6
Refract	折射渲染的测试	209 889	5
Jellyfish	复杂 shader 处理的测试	13 200	4

图 4　Effect 2D GPU 模拟器效果图

图 5　Shadow GPU 模拟器效果图

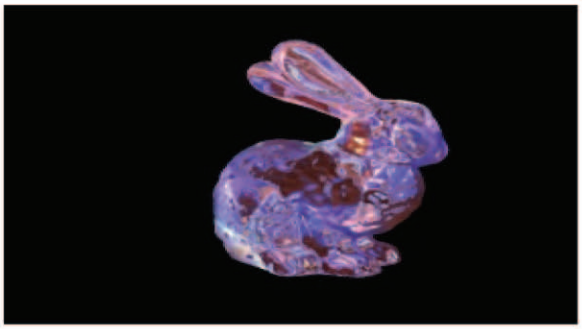

图 6　Refract GPU 模拟器效果图

图 7　Jellyfish GPU 模拟器效果图

试表明 GPU 模拟器驱动可准确完成折射的绘制。图 7 所示是复杂 shader 的测试,通过该测试表明,利用 GPU 模拟器驱动中的编译器可以正确绘制复杂的 shader。测试中,2D 图形绘制顶点及图元数目较少,每帧虽需 3 次 drawcall,但包含了 clear 等操作,实际绘制 1 次 drawcall 即可完成,具有很好的图形绘制加速效果。其余测试用例顶点及图元数目较多,每帧 drawcall 数量也相对较多,说明本驱动配合 GPU 模拟器可准确无误地完成复杂图形绘制。

5　结　论

GPU 驱动是 GPU 与 CPU 之间交互的桥梁,在 GPU 开发中起到了不可忽视的作用。本文在 Mesa 开源驱动的框架下进行 GPU 的驱动开发工作,可以大幅减少 GPU 驱动的开发时间和开发难度。同时,为了提高 GPU 模拟器的可编程性,在 Mesa 驱动中集成了编译器的设计,可将 Mesa 中的通用中间语言 TGSI 编译为 GPU 模拟器的汇编语言。本驱动还适配 OpenGL 等多款 API 以及 GLSL 等 shader 编程语言。并且,根据 Mesa 开源驱动驱动,设计出一套完整的可读 GPU 模拟器接口,方便调试人员进行驱动及 GPU 结构的调试工作。通过设计 GPU 的驱动和模拟器的接口,对高性能 GPU 模拟器的结构设计起到了指导作用。在 GPU 模拟器中设计的各个模块均参考该套接口,在本接口的基础上进行结构设计。本设计已经和 GPU 模拟器进行了 OpenGL 等 API 的验证,在 GL _ mark 的验证测试中,可以正常完整地绘制出 GL _ mark 测试集的每一帧图像。本接口设计后续可继续用于高性能 GPU 的结构设计中,驱动设计后续还可继续用于 GPU 的用户态驱动中,通过实现内核态驱动 DRM,并编写 libDRM 接口与本设计进行对接,便可实现高性能 GPU 的一整套完整驱动。

参考文献

[1] Das P K, Deka G C. History and evolution of GPU architecture[EB/OL]. https://www.igiglobal.com/chapter/history-and-evolution-of-gpuarchitec-ture/139841: IGI Global, 2016

[2] AMD Corporation. RDNA architecture presentation[EB/OL]. https://gpuopen.com/wp-content/uploads/2019/08/RDNA _ Architecture _ public.pdf: AMD Corporation, 2019

[3] AMD Corporation. "RDNA 1.0" instruction set architecture[EB/OL]. https://gpuopen.com/wp-content/uploads/2019/08/RDNA _ Shader _ ISA _ 5August2019.pdf: AMD Corporation, 2019

[4] AMD Corporation. RDNA architecture whitepaper[EB/OL]. https://www.amd.com/system/files/documents/rdna _ whitepaper.pdf: AMD Corporation, 2019

[5] 胡伟武. 自主 CPU 发展道路及在航天领域应用[J]. 上海航天, 2019(1):1-9

[6] 林一松,唐玉华,唐滔. GPGPU 技术研究与发展[J]. 计算机工程与科学, 2011, 33(10):85-92

[7] Blythe D. The Direct3D 10 system[J]. *ACM Transactions on Graphics*, 2006, 25(3):724-734

[8] Wright R S, Haemel N, Sellers G 著, 付飞, 李艳辉译. OpenGL 超级宝典(第 5 版)[M]. Addison-Wesley Longman, Amsterdam, 北京:人民邮电出版社, 2012

[9] Ginsburg D, Purnomo B, Shreiner D 著. OpenGL ES 3.0 编程指南[M]. 北京: 机械工业出版社, 2015

[10] Kuo L W, Yang C C, Lee J K, et al. The design of LLVM-based shader compiler for embedded architecture [C]// IEEE International Conference on Parallel & Distributed Systems, Xinzhu, China, 2015

[11] AMD Corporation. A detailed look at the R600 backend-LLVM [EB/OL]. http://llvm.org/devmtg/2013-11/slides/Stellard-R600.pdf: AMD Corporation, 2013

[12] Mantor M. AMD Radeon™ HD 7970 with graphics core next (GCN) architecture[C]// Hot Chips 24 Symposium, Cupertino, USA, 2012

[13] Wittenbrink C M, Kilgariff E, Prabhu A. Fermi GF100 GPU architecture[J]. *IEEE Micro*, 2011, 31(2):50-59

[14] Blinzer P. The heterogeneous system architecture: it's beyond the GPU[C]//2014 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, Agios Konstantinos, Greece, 2014

[15] The Mesa 3D Graphics Library, release 17.0.3 notes [EB/OL]. https://www.mesa3d.org/relnotes/17.0.3html: VMware, 2017

[16] Shreiner D, Sellers G, Kessen J, 著. OpenGL 编程指南 [M]. 北京: 机械工业出版社, 2014

[17] VMware, X.org. Gallium's documentation[EB/OL]. https://gallium.readthedoc s.io/en/latest/: VMware, X.org: Nouveau Revisione64091eb, 2011

[18] VMware, X.org. TG shader infrastructure[EB/OL]. https://www.freedeskto p.org/wiki/Software/gallium/tgsi-specification.pdf: VMware, X.org, NouveauRevisione64091eb, 2011

The driver design reserch for high-performance GPU simulater

Zhao Shipeng[*,**,***], Zhang Lizhi[*,**,***], Zhao Haoyu[****], Su Menghao[*****], Liu Su[*****]

([*] State Key Laboratory of Computer Architecture, Institute of Computer Technology, Chinese Academy of Sciences, Beijing 100190)

([**] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([***] University of Chinese Academy of Sciences, Beijing 100049)

([****] School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026)

([*****] Loongson Technology Corporation Limited, Beijing 100190)

Abstract

The GPU driver design are studied. Considering the large amount of driver development, a design method of high-performance GPU simulator driver based on Mesa open source driver framework is proposed. Under the development framework of Mesa open source driver, a complete set of GPU simulator driver is realized. This design can be adapted to a variety of application programming interfaces (APIs), such as OpenGL. Based on the open source framework, the development difficulty is greatly reduced. In order to adapt to the programmability of the high-performance GPU simulator, the GPU compiler design is also integrated in the driver design, and the programming language, such as GLSL, can be changed into the assembly code of the shader in the GPU, which is calculated by the shader. According to the simulator driver design, a set of interface design of high-performance GPU simulator is proposed, which provides a set of readable drive interface for each module of the simulator and guides the structural design of the simulator.

Key words: open source driver, application programming interfaces (API), driver design, GPU simulator, simulator interface