doi:10.3772/j.issn.1002-0470.2020.06.002

高性能 GPU 模拟器的实现①

张立志②[*,**,***] 赵士彭[*,**,***] 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 基于图形处理器(GPU)由固定功能管线与可编程流处理器单元构成的特点,设计了一种半周期精确的模拟器实现方法,并结合一种 GPU 结构进行了 GPU 模拟器的实现。GPU 模拟器支持 OpenGL 2.0 API,在实现了固定功能管线的基础上,支持统一着色器渲染架构。模拟器对 GPU 结构进行半周期精确模拟。固定功能管线部分实现为完全周期精确模拟,可以进行3D 图形算法的正确性验证以及性能评估。对可编程流处理器单元进行指令级功能模拟,在固定功能管线部分的配合下,可以快速轻量地对整个 GPU 结构进行结构与功能的验证。文章使用模拟器运行了 glmark2 测试集,并且根据测试结果对 GPU 结构进行3D 图形算法正确性验证与性能评估,结果表明基于本文设计方法的 GPU 模拟器可以快速轻量地实现3D 图形算法正确性验证与性能分析。

关键词 图形处理器(GPU);模拟器;半周期精确;glmark2;逆序流水

0 引 言

图形处理器(graphics processing unit,GPU)是计算机系统中处理3D 实时图形程序的专用加速芯片,目前广泛应用于个人电脑、工作站、嵌入式设备与智能手机中。在中央处理器(central processing unit,CPU)与 GPU 的芯片设计[1]中,当对某款处理器进行结构探索时,通常并不是直接使用硬件描述语言对芯片进行实现与验证,而是使用高级语言(如 C、C ++)编写对应的处理器模拟器,来进行结构探索与功能验证,因为高级语言编写的模拟器在功能实现与错误调试方面都更为简单快捷,同时可以对结构进行低成本的优化改进。

在 GPU 的架构演进[2,3]中,GPU 的功能不断增多。1992 年美国硅图公司将图形流水线应用于 GPU[4],GPU 的原始输入数据是由 CPU 预先处理好的三角形数据,GPU 只负责光栅化与纹理采样等少数几个操作,功能非常单一。在不断的发展过程中,GPU 逐渐开始负责原本由 CPU 负责的三角形的生成与剔除、顶点光照特效、顶点数据移动变换操作、像素点特效操作、曲面细分等功能。越来越复杂的 GPU 功能使得进行直接芯片实现的成本巨大,故模拟器验证变得越来越重要。随着 GPU 功能的丰富与性能的提升,GPU 结构变得越来越复杂,GPU 模拟器的代码规模也越来越庞大,灵活性越来越低,这使得 GPU 模拟器不易添加新的功能模块,也增加了测试验证的难度。

为解决周期精确模拟器 GPU 模拟器代码灵活度低的问题,本文提出并实现了一种半周期精确的

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,1992 年生,博士生;研究方向:计算机系统结构,GPU 体系结构;联系人,E-mail: zhanglizhi@loongson.cn

(收稿日期:2019-07-10)

GPU 模拟器实现方法,基于该方法设计的 GPU 模拟器可以快速地对 3D 图形算法进行正确性验证与性能分析。

本文的主要贡献是:(1)结合 GPU 结构特性,提出了半周期精确的 GPU 模拟器设计方法。该方法在保证功能验证与性能评估的基础上,降低了 GPU 模拟器内部功能模块之间的耦合度,增加了模拟器灵活性,同时降低了 GPU 模拟器使用者对模拟器的学习成本。它将 GPU 固定功能管线部分使用逆序流水方式实现完全的周期精确模拟,在可编程流处理器单元使用单周期处理器的功能模拟,各模块之间使用队列对周期精确进行解耦合。(2)将半周期精确模拟器设计方法应用于实际系统,实现一款 GPU 模拟器,并使用 glmark2 测试集对该模拟器进行功能正确性验证,同时对三角形光栅化算法进行了性能评估。

1 相关工作

现代 GPU 架构主要由 2 部分构成,即固定功能管线与可编程流处理器单元。固定功能管线主要负责 GPU 中图形算法的实现与 GPU 内各模块的调度;可编程流处理器单元负责执行图形程序中的顶点着色器与片段着色器代码,同时因为其可编程特性,近些年较为火热的 GPGPU[5](general-purpose computing on graphics processing unit)也由可编程流处理器单元实现。

目前国内外已经提出多种高级语言编写的 GPU 模拟器,这些模拟器也可依据 GPU 架构的特性分为 2 类:完整的 GPU 模拟器和只实现可编程流处理器单元的通用计算模拟器。其中通用计算模拟器涵盖了近些年大部分 GPU 模拟器,例如 Multi2Sim[6]、Multi2Sim Kepler[7]、MIAOW[8]、GPGPU-Sim[9]与基于 Gem5 CPU 模拟器[10]开发的 gem5-gpu[11]。这些模拟器只对可编程流处理器单元进行模拟仿真,可以运行完整的 GPGPU 程序,但不包含引言所提到的光栅化与纹理采样操作、三角形的生成与剔除、顶点光照特效、顶点数据移动变换操作、像素点特效操作、曲面细分等功能,无法执行任何完整的图形应用程序。

因为 GPU 技术中图形处理技术主要掌握在少数商业 GPU 厂商手中(如 Nvidia、AMD),他们因为商业原因,并没有将 GPU 技术对外公开,所以学术界所使用的 GPU 模拟器数目并不多。所公开源代码的 GPU 模拟器中最具代表性且广泛使用的模拟器是 ATTILA 模拟器[12]。ATTILA 模拟器是加泰罗尼亚理工大学于 2006 年与 Intel 公司共同开发的一款完全周期精确的 GPU 模拟器,模拟器一直维护至 2015 年。该模拟器为了实现完全的周期精确模拟,使用 Signal 数据结构详细模拟了 Verilog HDL 语言中的 Wire,同时模拟了时序延迟与传输带宽。这种处理方式虽然保证了模拟器的正确性与准确度,但是极大增加了模拟器的软件规模与软件结构复杂度,使得开发者难以在模拟器上增加新的功能模块,更加难以对所模拟的 GPU 结构进行较大规模的升级修改。在图形算法的模拟中,ATTILA 采用时序模拟与算法功能仿真分离的实现方式,分别使用不同的模块进行实现,这种方法再次增加了代码的可调试性与阅读难度。加泰罗尼亚理工大学于 2013 年再次与 Intel 公司合作开发了 Teapot[13]模拟器,迄今一直在维护使用,但该模拟器依然采用了完全周期精确的模拟方式,没有解决模拟器的灵活性问题。

为解决 GPU 模拟器软件结构过于复杂且不易进行结构更改与代码调试的问题,同时结合前述模拟器实现方法与 GPU 本身特性,本文提出了半周期精确的 GPU 模拟器实现方法,该方法在保证功能正确的基础上,降低了 GPU 模拟器软件结构复杂度,使得模拟器使用者可以快速地为 GPU 模拟器添加新功能,并使用 OpenGL 2.0 程序为该功能模块进行正确性验证与性能分析。

2 半周期精确模拟器

在完整的 GPU 设计路线中,模拟器只是一个阶段性产物,而不是像 ATTILA 模拟器与 Teapot 模拟器在其项目规划中将模拟器作为最终产物,简单高效的实现模拟器将极大影响 GPU 的整体研发效率。半周期精确模拟器的设计目的是在 GPU 结构设计

初期对结构进行功能正确性验证与性能分析。由于 GPU 最终结构设计代码将由 Verilog HDL 语言编写,用高级语言编写的模拟器在功能与结构上与最终所期待 GPU 有很大差距。所以在使用模拟器对 GPU 结构进行验证后,还需要使用现场可编程门阵列(field progrmmable gate array, FPGA)进行更深一步验证与性能评估。

GPU 一般由固定功能管线与可编程流处理器单元 2 部分构成。固定功能管线是一条数据驱动的图形算法流水线电路,驱动数据为顶点数据与像素点数据;可编程流处理器单元则是指令驱动处理器,可进行取指、译码、执行、访存操作,其指令为着色器程序的汇编指令。这 2 个面向不同处理对象的单元由先入先出(first input first output,FIFO)或随机存取存储器(random access memory, RAM)进行解耦合。在进行性能评估时[14,15],固定功能单元性能指标一般为每秒或每周期可处理的顶点数目与像素点数目;而可编程流处理器单元性能指标一般为每秒可处理浮点指令的数目。这是因为在实际 3D 图形应用中,即使使用相同的顶点数据与像素点数据,在调用不同着色器程序时,图形的显示效果也大不相同。一般增加着色器程序指令数目后,图形显示效果将更加逼真细腻,同时可编程流处理器单元执行的时间也将变长,进而影响到固定功能单元每秒可处理的顶点数目与像素点数目。这 2 个单元在性能评估阶段并没有使用相同的标准,而且这 2 个性能评估标准也无法进行进一步的统一衡量。

基于上述原因,本文基于“GPU 由固定功能单元与可编程流处理器单元共同组成”这一特性,将 GPU 模拟器实现为半周期精确的功能级模拟器。将固定功能管线进行周期精确的模拟,所有模块在模拟器中所执行时间与该模块在真实芯片中执行时间相同,例如 32 位乘法部件将进行 4 个时钟周期的全流水模拟,与真实芯片执行时间相同。而可编程流处理器单元则不进行周期精确的模拟,只进行指令级功能模拟,每周期处理一条完整指令。虽然在真实芯片中,每条指令将进行多周期全流水执行,但单周期功能级模拟也将满足程序正确性,并且极大降低了模拟器平台调试与更改的复杂度,同时节省了开发成本。不同于 ATTILA 模拟器,本文在对模拟器实现时,使用倒序流水与虚假执行的方式将功能模拟与时序模拟实现在同一模块,增加了代码的可读性与可调试性。

3 模拟器实现

GPU 模拟器主要由 3 部分构成,分别是模拟器驱动程序、GPU 固定功能单元和 GPU 可编程流处理器单元。其中 GPU 固定功能单元还包括光栅化单元。下面将依次对这些模块进行介绍。

3.1 GPU 模拟器驱动程序

为了快速方便地为 GPU 模拟器生成输入数据,GPU 模拟器配套开发一套基于 Mesa[16] 驱动程序的模拟器驱动程序。模拟器驱动程序可适配 OpenGL 等多款应用程序接口,可以将正在运行的 OpenGL 程序的信息抓取出来。所抓取信息包括顶点数据、纹理数据、OpenGL 流水线状态、GPU 命令、着色器代码。这些数据将作为 GPU 模拟器的输入数据。

3.2 模拟器具体实现

模拟器整体软件架构主要由 Module 和 Queue 2 个数据结构与 Clock 函数构成。Module 用于构建 GPU 中每一个功能单元,类似于 Verilog HDL 语言中关键词 module。Queue 用于实现各个 Module 间通讯功能,同时实现各 Module 的解耦合。Queue 在本质上是一个每周期一写一读的循环队列。具体实现方式如图1所示。其中方框表示Module,对应GPU中每一

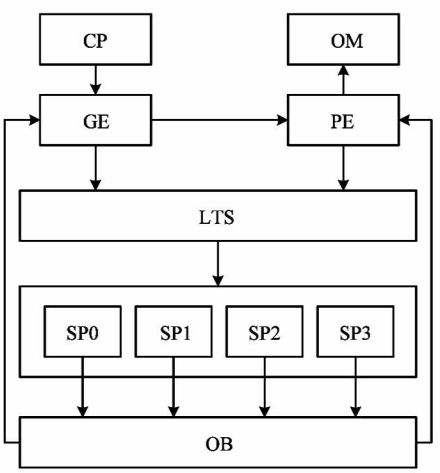

图 1 GPU 模拟器结构示意图

个部件。每个 Module 都具有自己的 Clock 函数，该函数实现了对应模块在每个时钟周期内应当执行的操作。连接线表示 Queue，各个模块通过 Queue 进行数据传递。

3.2.1　固定功能单元

固定功能单元包括 GPU 流水线中命令处理器（command processor，CP）、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 schedule，LTS）、输出合并单元（output merge unit，OMU）、输出总线（output bus，OB）。这些单元因为算法较为单一固定，可以通过对寄存器进行不同配置来实现不同的功能需求，并且其性能完全由结构设计体现。为了对这些部件进行完整的性能分析，本方法对这些单元使用周期精确的方式进行实现。

在对上述模块进行周期精确模拟时，由于有些处理单元具有多个流水级，且每个流水级又包括多级细分流水级，所以在进行模拟器实现时，本方法通过两层逆序流水的方式对图元处理引擎进行实现。下文使用图元处理引擎的实现方式进行说明。

图元处理引擎结构各流水级完成各种不同数据的乘加运算或其他数学运算，所有运算部件设计为全流水部件，流水级内部同样实现更细粒度的周期模拟。流水级内部通过虚假执行与逆序流水的方式实现细分流水。图 2 表示 2 个流水级之间的数据传递与数据计算，每一流水级包括 4 级细分流水级。如第 1 步所示，Clock 函数执行时，后序流水级首先判定其最后一级细分流水级是否为空，如果为空，则将其前 3 级细分流水数据传递至后一级细分流水级，同时将第 1 级细分流水级标为空。如第 2 步所示，当前序流水级最后 1 级细分流水级不为空，且后序流水级第 1 级细分流水级为空时，前序流水级将数据传递至后序流水级，后序流水级直接将数据进行计算，并将结果写入后序流水级第 1 级细分流水级。如第 3 步所示，前序流水级进行流水级内细分流水级数据传递。图元处理引擎将整条流水线逆向调用所有流水级的 Clock 操作，来实现整条流水线的并行周期执行。

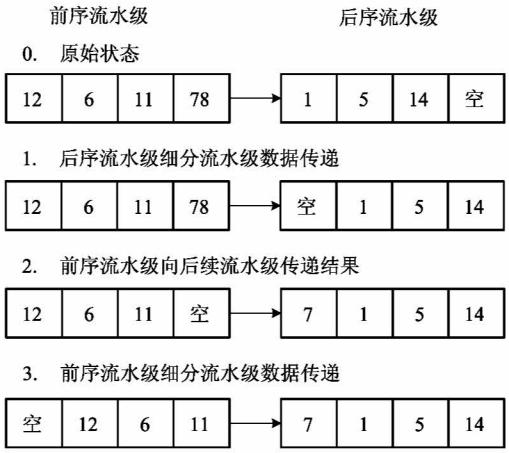

图 2　Clock 函数逆序流水实现方式

3.2.2　光栅化单元

光栅化单元是图元处理引擎中一个固定算法单元，本模拟器光栅化单元基于 Larrabee[17,18] 算法进行实现，该算法使用递归算法来实现。高级语言可以通过使用栈这一数据结构来保存递归算法的中间数据，但递归中无法预知递归调用的深度，继而无法预知中间结果数据量的大小。而且在硬件设计中，存储单元代价比较大，所以不便于使用单纯意义上的递归程序来实现光栅化算法。本工作中 GPU 模拟器通过将光栅化核心算法单元复制多份来实现不同层级的递归调用，光栅化单元分为 7 级。第 1 级负责根据接受到的数据生成光栅化所必需的数据结构，其余流水级将宽度为 2^n 正方形片段数据生成宽度为 $2^{(n-1)}$ 的正方形片段数据，然后将产生的结果传递至其后一级流水级。这种结构通过增大逻辑单元面积来减小存储单元的消耗，解决了递归深度不可预知的问题，同时稳定地保证了光栅化单元每周期产生 1 组 4×4 的像素点数据的速率，使光栅化部件不会成为整条流水线的瓶颈。

3.2.3　可编程流处理器单元

可编程流处理器单元支持 OpenGL 2.0 标准中的统一着色器功能，统一着色器可执行顶点着色器与片段着色器 2 种着色器程序，模拟器实现为 4 份，在图 1 中由 SP0、SP1、SP2、SP3 表示。可编程流处理器与传统较为简单多流水级 CPU 的结构类似，包括取指、译码、执行、写回等流水级，不易使用 C++ 串行语言进行模拟，所以可编程流处理器单元实现

为功能级模拟。GPU 模拟器配套开发环境下的编译器将 OpenGL 程序中编写的着色器程序编译为对应 GPU 模拟器流处理器的汇编代码,在功能级模拟下流处理器每拍直接完成一条指令。

4 实 验

为了验证本文提出模拟器的 3D 图形算法正确性验证的功能与性能分析的功能,本文采用 glmark2[19] 测试程序对 GPU 模拟器进行实验测试。glmark2 是一款由 Alexandros Frantzis 与 Jesse Barker 开发的 OpenGL API 程序测试集,包括涵盖了 OpenGL 2.0 API[20] 大部分功能的多个测试场景,这些测试场景所实现的图片效果较为简单,可以轻易判定功能实现的正确性。实验分为 3D 图形正确性验证与算法性能验证两部分展开。

4.1 3D 图形算法正确性验证

本文通过对 glmark2 中 build 测试项的正确性分析来表现模拟器 3D 图形算法正确性验证的功能。

在 3D 图形算法正确性验证实验中,本文将商业显卡 GPU 执行 OpenGL 程序渲染得到的图片与模拟器渲染得到的图片进行对比。为确保数据一致性,商业显卡 GPU 所执行的 OpenGL 程序输入数据也来自于本文中使用的模拟器输入数据。

GPU 模拟器绘制图片与商业显卡 GPU 绘制的图片存在差异与错误 2 种不同。差异是 GPU 进行具体实现时选择了不同实现方式而引起的不同,例如固定功能管线使用大量的数学运算实现复杂的图形算法,不同 GPU 设计人员在选择不同的数据计算精度与不同的数据传输精度时,将不可避免地引入差异。这种差异在商业显卡,如 AMD GPU 与 Nvidia GPU 之间同样有体现。它们通常表现为像素点的颜色值偏差与像素点的位置偏差,但这并不影响图形应用的效果展示,且肉眼无法分别。图形功能性错误则是由于固定功能管线中图形算法的错误选取或者错误实现引起的,也有可能是因为可编程单元中指令的错误实现而引起。这种错误通常表现为模型位置大幅偏移、像素点缺失、像素颜色值错误,肉眼可轻易辨别。

对比实验中,商业显卡 GPU 实验环境为操作系统是 Fedora 28,GPU 为 Nvidia 1660Ti,驱动程序为 NVIDIA-Linux-x86 _ 64-418.88。

图 3 是商业显卡 GPU 环境下 build 测试项的标准渲染结果,图 4 是 GPU 模拟器渲染结果。对比发现,肉眼无法看出 2 图之间的差别。对图 3 与图 4 进行逐像素对比,对比方式为对 2 图中每一相同位置像素值进行做差运算,差值结果为正数,则将差值数据保存至图 5,差值结果为负数,则将差值数据绝对值保存至图 6。对图 5 与图 6 进行对比,可以发现图 5 呈现出 build 测试场景的下边沿,图 6 则呈现出 build 测试场景的上边沿,表示图 3 与图 4 存在轻微的位置偏移,这是由于光栅化算法的实现不同所引起,属于差异而不是错误。而图5与图6中除了模

图 3 商业显卡 GPU 绘制效果图

图 4 GPU 模拟器绘制效果图

图 5　差异值正值效果图

图 6　差异值负值效果图

型边沿数据,不再有其他差别,表明 GPU 模拟器对 build 测试项绘制正确。

4.2　算法性能验证

本文选取 Intel 曾经在 Larrabee 论文中提到的结构光栅化算法来表现模拟器性能评估的功能。

结构光栅化算法根据待光栅化三角形的包围框数据,将屏幕空间对应的区域划分为多个正方形 Tile,通过检测每个 Tile 的 4 个顶点与三角形 3 条边的位置关系来判定当前 Tile 是否与三角形产生了覆盖关系。这一方法虽然效率比较高,但并不能保证所有被判定为覆盖的 Tile 是真正地与三角形产生覆盖关系,如图 7 所示,Tile 被判定为与黑色小三角形覆盖,但其实并没有覆盖。Larrabee 论文表示可以通过增加包围框测试对结果进行进一步确认,这样可以减少被误判定为与三角形虚假覆盖的Tile个数,但是该论文也表明,无法确认增加包围框测试是否会提升光栅化性能。

本文通过对 GPU 模拟器实现 2 种不同算法来对增加包围框是否对光栅化算法有明显提升这一结论进行验证。光栅化操作只与图形程序中待绘制模型的几何特性有关,在删去 glmark2 中重复模型后,选取了 build、shadow、refract、buffer、bump、loop、jellyfish、ideas、terrain 测试项进行实验。

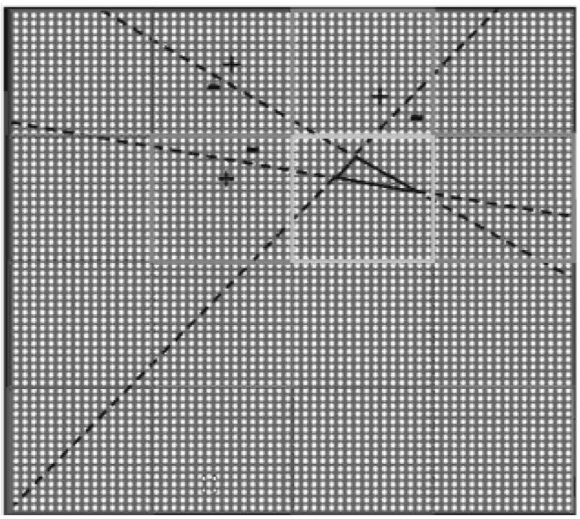

图 7　递归光栅化算法[19]

如 3.2 节与 Larrabee 论文所述,本文将光栅化算法实现 7 级处理。第 1 级接受 128 × 128 的 Tile 数据,将该 Tile 等分为 4 个 64 × 64 的 SubTile,通过接受点与拒绝点测试算法测试后输出每一个 SubTile 与三角形的覆盖状态,覆盖状态为 True 的 SubTile 将作为第 2 级的输入数据继续进行测试。图 8 表示算法测试结果,其中 build、shadow、refract、ideas、terrain 在第 6 级与第 7 级表现出超过 10% 的性能提升,refract 在第 6 级表现出最高 31.1% 的性能提升;buffer、loop、jellyfish 在第 5 级表现出超过 10% 的性能提升;bump 性能提升并不明显;而除了 bump 外,各个模型在第 1 级第 2 级第 3 级几乎没有性能提升。结合各个模型不同像素数目,综合计算得出这 9 个测试模型的加权平均性能提升了10.11%。结果表明在增加包围框测试后,光栅化算法得到了明显的性能提升。

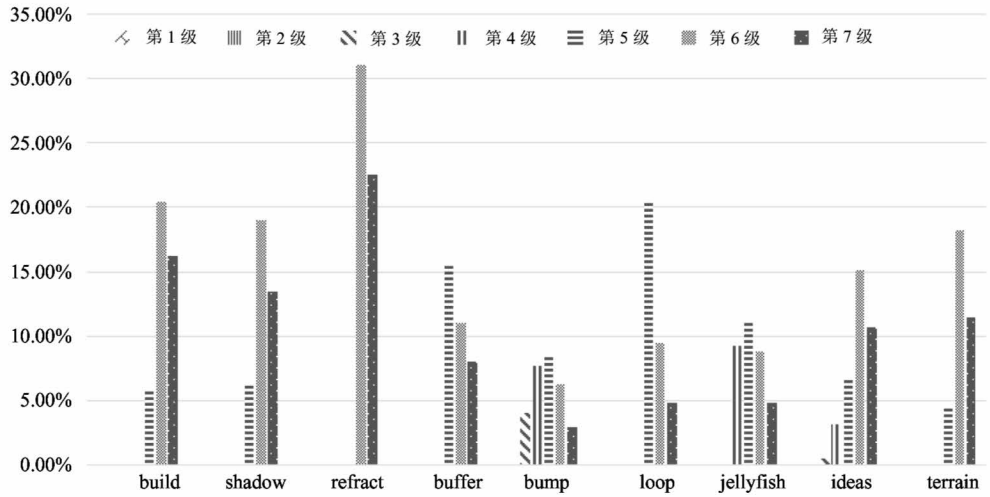

图 8　包围框算法优化前后对比图

5　结　论

GPU 模拟器通过使用高级语言模拟仿真实现硬件描述语言,在处理器结构设计前期起到重要的验证作用。本文基于 GPU 由固定功能管线与可编程流处理器单元构成这一特点,同时基于 GPU 各部件天然解耦合的性质,提出半周期精确的 GPU 模拟器设计方法。这一方法基于逆序流水与虚假执行的方式,将固定功能管线实现为周期精确模拟,将可编程流处理器单元实现为指令级模拟。该方法降低了 GPU 模拟器开发成本,同时也降低了 GPU 模拟器使用者对模拟器的学习成本。

本文基于此方法设计并实现了 GPU 模拟器。该模拟器实现了 OpenGL 2.0 API,在配套使用 GPU 模拟器驱动程序的情况下,可执行完整的 OpenGL 2.0 图形程序。本文选取了 glmark2 测试集对 GPU 模拟器进行验证,同时选取了 Larrabee 中结构光栅化算法进行算法性能评估。模拟器使用者可以基于模拟器驱动程序运行其他 OpenGL 2.0 程序,还可以对该模拟器增加新的功能与新的算法,同时也可以使用其他 OpenGL 2.0 程序作为模拟器的输入,进行不同算法的功能正确性验证与性能评估。

参考文献

[1] 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

[2] McClanahan C. History and evolution of GPU architecture [EB/OL]. http://mcclanahoochie. com/blog/wpcontent/uploads/2011/03/gpu - hist - paper. pdf: Mcclanahoochie, 2010

[3] Akenine-Moller T, Haines E, Hoffman N. Real-time rendering [M]. Natick, Massachusetts: AK Peters/CRC Press, 2019: 1024-1039

[4] Akeley K. Reality engine graphics [C] // Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, Anaheim, USA, 1993: 109-116

[5] Luebke D, Harris M, Govindaraju N, et al. GPGPU: general-purpose computation on graphics hardware [C] // Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, Tampa, USA, 2006: 208

[6] Ubal R, Jang B, Mistry P, et al. Multi2Sim: a simulation framework for CPU-GPU computing [C] // The 21st International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, USA, 2012: 335-344

[7] Gong X, Ubal R, Kaeli D. Multi2Sim Kepler: a detailed architectural GPU simulator [C] //2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Santa Rosa, USA, 2017: 269-278

[8] Balasubramanian R, Gangadhar V, Guo Z, et al. Miaowan open source rtl implementation of a GPGPU [C] // 2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII), Yokohama, Japan, 2015: 1-3

[9] The University of British Columbia. GPGPU-Sim [EB/

OL]. http://www.gpgpu-sim.org: GPGPU-Sim, 2019

[10] Binkert N, Beckmann B, Black G, et al. The gem5 simulator[J]. *ACM SIGARCH Computer Architecture News*, 2011, 39(2): 1-7

[11] Power J, Hestness J, Orr M S, et al. gem5-gpu: a heterogeneous CPU-GPU simulator[J]. *IEEE Computer Architecture Letters*, 2014, 14(1): 34-36

[12] Del Barrio V M, González C, Roca J, et al. ATTILA: a cycle-level execution-driven simulator for modern GPU architectures[C]//2006 IEEE International Symposium on Performance Analysis of Systems and Software, Austin, USA, 2006: 231-241

[13] Arnau J M, Parcerisa J M, Xekalakis P. Teapot: a toolset for evaluating performance, power and image quality on mobile graphics systems[C]//Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, Eugene, USA, 2013: 37-46

[14] Techpowerup. TechPowerUp GPU-Z[EB/OL]. https://www.techpowerup.com/gpuz: Techpowerup, 2019

[15] 3dmark. 3dmark[EB/OL]. https://www.3dmark.com: 3dmark, 2019

[16] Mesa3D. Mesa 3D graphics library[EB/OL]. https://www.mesa3d.org: Mesa 3D, 2008

[17] Abrash M. Rasterization on larrabee[EB/OL]. http://software.intel.com/en-us/articles/rasterization-on-larrabee: Intel, 2009

[18] Seiler L. Larrabee: a many-core x86 architecture for visual computing[J]. *ACM Transactions on Graphics*, 2008, 27(1):10-21

[19] Git Hub Inc. glmark2[EB/OL]. https://github.com/glmark2/glmark2: GitHub, 2017

[20] Rost R J, Licea-Kane B, Ginsburg D, et al. OpenGL Shading Language[M]. New York: Pearson Education, 2009: 1-382

Implementation of a high performance graphics processor simulator

Zhang Lizhi[*,**,***], Zhao Shipeng[*,**,***], 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

Based on the structural characteristics of graphics processing unit (GPU), a half-cycle accurate simulator implementation method is designed. The graphics processor simulator supports the OpenGL 2.0 API and supports a unified shader rendering architecture based on a fixed-function pipeline. The simulator performs a half-cycle accurate simulation of the graphics processor architecture. The fixed-function pipeline part is implemented as a full-cycle accurate simulation. The simulator completes the correctness verification of the graphics algorithm and performance evaluation. Programmable unit is implemented by instruction-level functional simulation. Together with the cooperation of the fixed function pipeline and programmable unit, the simulator can quickly and lightly verify the correctness of the architecture and function of the entire graphics processor architecture. At the end of the article, the simulator runs glmark2, and the performance of the graphics processor architecture is evaluated according to the results, and the feature set is analyzed. It proves that the simulator can achieve the function of correctness verification and architecture performance evaluation quickly and lightly.

Key words: graphics processing unit (GPU), simulator, half-cycle accurate, glmark2, reverse pipeline