基于 Dr.Jit 的可微分路径追踪渲染器

摘要

Dr.Jit [1] 是一款专门为实现基于物理的可微分渲染而设计的编译器，使用 Dr.Jit 编写的渲染算法会被编译为一个支持数据并行的大型核函数，其在 GPU 端的渲染速度超过了 PBRT 4 的两倍，且 Dr.Jit 支持自动微分，提供前向和反向两种模式。Mitsuba 3 是基于 Dr.Jit 的可微分渲染器，由于其使用 C++ 编写且代码规模较大，在科研工作中快速验证想法变得困难且繁琐。可微渲染领域的研究大多采用 Python 作为主要编程语言，而 Dr.Jit 也提供了 Python 接口，因此，本次复现旨在使用 Python 代码实现一个小型的可微分渲染器框架，为后续科研工作提供一个易于上手且高度自定义的平台。

关键词： Dr.Jit；可微分渲染

1 引言

渲染的过程可以描述为一个函数，它接收场景中的参数，包括物体的几何信息，材质，场景光照，相机参数等，通过一些基于物理的运算，此函数输出一张 RGB 图像，使这张图像尽可能还原出真实世界的光影。随着机器学习与深度学习在近十年的快速发展，近年来，可微分渲染也逐渐成为了研究者关注的方向，在一些任务中，我们已知 RGB 图像，希望通过图像求出场景中的参数，一般此类病态 (ill-posed) 问题无法直接求解，但是可以使用随机梯度下降法对场景参数进行优化，这和深度学习的优化过程类似，只是这个过程中不存在神经网络这样的黑盒。基于物理的渲染一般指使用路径追踪方法的渲染，也被人称作光线追踪。

Dr.Jit [1] 是一个专为可微渲染任务设计的运行时编译器，它提供了 C++ 和 python 接口，可以根据用户编写的代码生成高度优化的可在 GPU 或 CPU 端高效并行执行的程序，并且和 pytorch 一样，支持自动微分。

开发 Dr.Jit 的团队也是渲染器 Mitsuba 3 的开发人员，这是一款以研究为导向的可微分渲染器，后端依赖于 Dr.Jit，很多论文在此渲染器中都有正确的实现，然而此渲染器以 C++ 编写，项目代码较为庞大，要修改此渲染器并实现一些自定义的功能是一件较为困难的事，非专业人员很难上手。Dr.Jit 同样提供了 Python API，因此理论上 Mitsuba 可以实现的功能使用 Python 编写的渲染器同样可以实现，使用 Dr.Jit 编写的算法会由此编译器进行优化，因此仅对于渲染程序来说，不管使用什么编程语言，程序的效率不会有太大变化。

2 相关工作

此部分对课题内容相关的工作进行简要的分类概括与描述，二级标题中的内容为示意，可按照行文内容进行增删与更改，若二级标题无法对描述内容进行概括，可自行增加三级标题，

后面内容同样如此，引文的 bib 文件统一粘贴到 **refs.bib** 中并采用如下引用方式 [?]。

2.1 数组编程

今年来机器学习的热度逐渐升高，促使许多框架将自动微分 (AD) 与 n 维数组表示结合在一起，这些框架使用类似 XLA 的即时编译 (JIT) 后端，将操作融合为高效的内核，鉴于这些框架的广泛成功和与 Dr.Jit 的明显相似性，自然便产生出一个疑问，是否可以通过使用此类框架实现基于物理的直接渲染 (Physically based direct rendering, PBDR)，遗憾的是此类框架在实现 PBDR 任务时存在一些缺陷，致使它们成为了低效的解决方案。

2.2 自动微分

对数学表达式进行手动微分是一项机械化的活动且很容易出错，自上世纪 50 年代至 70 年代起，Nolan 1953, Wengert 1964, Linnainmaa 1976 等人提出一系列开创性工作，希望将微分的工作委托给计算机执行，Griewank 和 Walther 2008 的书回顾了几十年来对自动微分 (AD) 的研究成果。

基于对导数传播策略的数学结构和渐进复杂性的全面理解，在向用户展示自动微分时，存在多种方式，包括跟踪、抽象语法树或中间表示 (IR) 的源到源转换，以及结合跟踪与转换的混合方法。微分可以针对前向、反向和混合模式下的标量、密集或稀疏数组，计算纯或不纯函数 (pure/ impure function) 的一阶或高阶导数，Baidin 2018 等人对此进行了综述，Dr.Jit 的工作仅关心一阶导数的自动计算。

3 本文方法

3.1 可微渲染的动态特性

Dr.Jit 是一个运行时编译器，这涉及到可微分渲染的动态特性：我们使用各种各样的参数定义场景，包括采样器，相机，光源，模型，材质，贴图，如果对所有参数都计算微分，性能开销会很大，实际情况是，用户只需要对其中的一小部分参数计算微分，也就是图 1中使用淡蓝色标记出来的这部分，因此整个系统中哪些参数需要参与到自动微分中，哪些参数可以从自动微分中剔除，只有运行一遍程序才能知道，所以 Dr.Jit 是一个运行时编译器，它会记录一遍代码的执行流程，然后在运行时编译出真正要执行的程序。

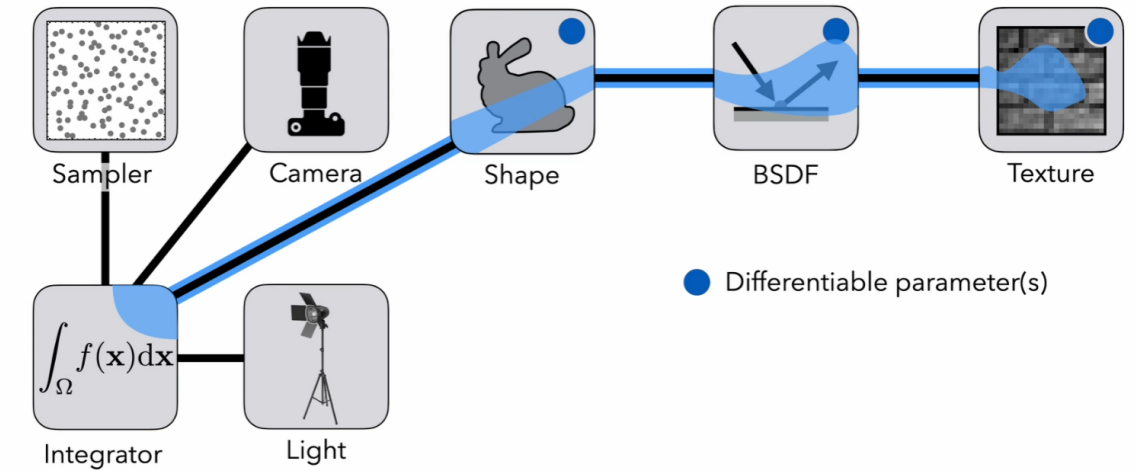

图 1. 可微渲染的动态特性

3.2 追踪

如果使用 Pytorch 渲染康奈尔盒子，会生成异常庞大的计算图，因为 Pytorch 会把每个计算步骤储存到计算图中，对于渲染这类算法，存在大量的循环以及嵌套的循环，因此循环展开后的计算步骤尤其庞大。使用 Dr.Jit 则可以解决这个问题，Dr.Jit 的 Tracing 功能也会记录计算图，为后续的编译做准备，但它可以把循环体和循环条件也记录到计算图中，不会像 Pytorch 那样在计算图中把循环展开，因此计算图的大小就被大大压缩了，此外，渲染算法中存在很多虚函数调用，只有在运行时才能确定执行哪个分支，Dr.Jit 会把可能的分支全都记录到计算图中 2，这就是它关于 Tracing 的两大特性。

但这样的功能也会带来一些麻烦，比如说编程时需要遵循更多的规范才能确保计算图被正确地记录，而且使用此功能会导致调试困难，报错信息常常无法给出错误的准确位置，聊胜于无，第三点是对于递归，Dr.Jit 也无能为力，展开递归会导致编译后的指令数量骤增，所以程序中所有的递归都要由循环代替。

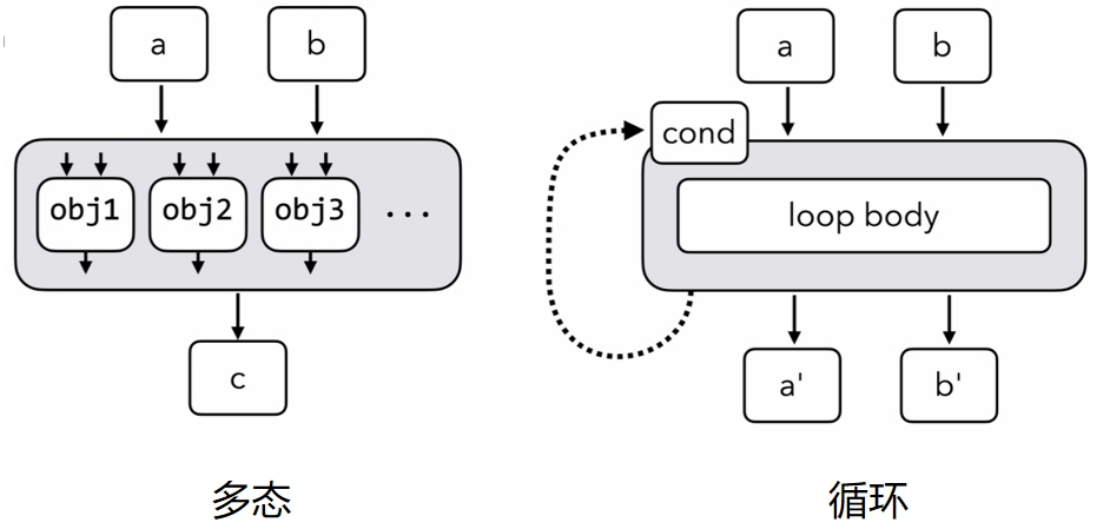

图 2. 追踪 (tracing)

3.3 优化原理

本节介绍 Dr.Jit 如何生成高效的程序。首先，Dr.Jit 会把整个渲染过程记录到一张巨大的计算图中，然后对计算图执行优化，生成一个大体量的 Kernel Function，也叫做 Megakernel，再次和 Pytorch 做一下对比，执行一遍 pytorch 程序通常会涉及多个 kernel function 的调用，它们之间存在一些依赖关系，使用这种策略的好处是每个 kernel 内部的并行度很高，但它的瓶颈在于每执行完一个 kernel 都要将中间结果写入 GPU 的显存，再由下一个 kernel 读取，这会占用大量带宽，使程序性能降低。对于渲染这类算法，每次采样涉及到上百万条光线的并行计算，一个 kernel 输出的中间数据通常是 GB 量级的，在显存上读写 GB 量级的数据成为影响渲染程序性能的主要因素，实验表明，把所有 kernel 合并为一个 MegaKernel 对渲染任务来说是比较合适的，即使这会降低 kernel 内部的并行度。

Dr.Jit 将所有渲染指令合并为 Megakernel 的计算图后，还会对计算图执行一些传统的优化方法，例如：消去公共子表达式，常量传递，删除无效的指令。

4 复现细节

4.1 与已有开源代码对比

本次复现参考了开源项目 Ray tracing in one weekend [2] 的代码，此项目包含 3 本在线书籍和对应的代码，是一个使用 C++ 实现的轻量级单线程路径追踪渲染器，项目中充分使用了递归和多态简化了代码逻辑，在将此项目转化为使用 Dr.Jit 的向量化程序时，面临的困难主要有以下几点：所有数据的向量化，递归转为循环，重写多态。

要使整个渲染算法可以并行执行，要将所有需要并行运算的数据向量化，包括每条采样射线，场景中所有的图元，所有的材质等；此外，Dr.Jit 无法追踪递归，仅能追踪循环，为了减小编译后的指令数量，要将所有的递归程序转换为循环程序，如有必要，需要利用栈来记录递归的中间状态；最后，C++ 的多态特性可以大大降低编程的难度，简化代码，然而在使用 Dr.Jit 时无法直接使用虚函数，因此用到虚函数调用的代码需要重构。

4.2 实验环境搭建

如果使用 anaconda，仅需打开一个安装了 python 的虚拟环境，然后执行 ‘pip install drjit‘ 安装 drjit 库即可。项目根目录中的脚本展示了复现的渲染器的基本使用方法，cornell_box.py 展示了如何渲染一个简单的康奈尔盒子，differential_test.py 展示了如何利用自动微分优化材质颜色，demo_scene.py 3 演示了如何导入 obj 模型文件，使用棋盘格纹理并开启景深效果。

图 3. 渲染效果演示

5 实验结果分析

5.1 定量分析

图 4 中展示了使用 Lambert 漫反射材质，发光材质和长方形图元搭建的康奈尔盒子，这是一个经典的测试场景，整个场景仅由最上方的一小块面光源点亮，中间的两个长方体为白色，但是光线经过左边的绿墙和右边的红墙反弹后，会将长方体的两侧染成绿色和红色，这就是全局光照，光栅化渲染器是很难获得这种效果的。

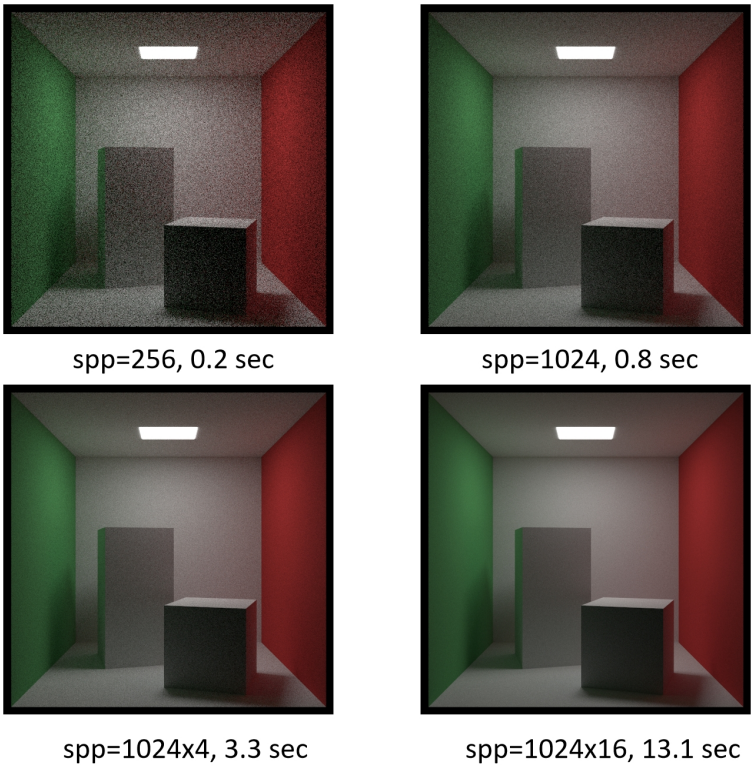

图 4. 渲染效果演示

由于光源面积过小，在进行随机采样时，只有很少一部分光线可以最终弹射到光源，导致采样数较小时场景的噪声比较大，通常这可以通过一些去噪算法和更好的采样方法缓解，例如直接对光源进行采样。

在 600 分辨率下，当光线弹射 50 次，每像素采样 2^14 次时，由 Dr.Jit 编译的核函数的指令数仅有 581 条，因为出现循环时，计算图只需要追踪一次循环，得益于极低的指令数，编译时间也仅有 20 ms，几乎可以忽略，而且编译过程只需执行一次，之后可以重复使用编译好的核函数，对于使用更复杂算法的 mistuba 3 渲染器，渲染此场景的核函数指令数约为 1600。

5.2 与 Mitsuba 3 对比

Mitsuba 3 是基于 Dr.Jit 的渲染器， 5 给出了使用 Mitsuba 渲染康奈尔盒子场景时渲染时间的对比，表格中的 spp 指每个像素的采样数，可以看到不论采样数为多少，本次复现实现的渲染器的渲染速度都会更快一点，不过这样的对比对于 Mitsuba 来说其实是不公平的，因为它使用了更复杂，但渲染质量更高的一些算法，但至少这个对比可以说明使用 python 编写的渲染器也是有实用价值的。

另外 Mitsuba 会为每根光线创建一个线程，而本次复现则是为每个像素创建一个线程，所以当每像素采样数量增加时，渲染器不会出现数组长度溢出这类问题，但 Mitsuba 无法应对这类情况，当采样数来到 4096×4 时，Mitsuba 就无法工作了，当然一般不会使用这么多的线程，除非是要渲染高清 4K 或 8K 图像。

spp	256	1024	4096	4096 * 4
Mitsuba 3	0.43 sec	1.3 sec	4.6 sec	✕
Mine	0.23 sec	0.88 sec	3.3 sec	13 sec

图 5. 与 mitsuba3 渲染时间对比

5.3 可微渲染测试

本节将利用 Dr.Jit 的自动微分功能优化场景中材质的颜色。在初始化场景时，首先为左右两面墙和中间的两个长方体随机设置了四种颜色，然后将渲染的图像与参考图像计算 L1 损失，利用梯度下降法更新这四种颜色，共计迭代 100 次，基本上迭代 50 轮就可收敛。在图 6 第二行的示例中，相机仅能看到中间的两个长方体，看不到两侧墙面，但仅依靠墙面产生的间接光，也可以正确优化出墙面的颜色，优化过程中，长方体的颜色在前几个迭代时迅速被优化为白色，之后借助参考图像中两侧墙面产生的绿色和红色的间接光，视野之外的红墙和绿墙的颜色也可以被慢慢还原出来，右侧两列展示了优化结果和参考图像。这就是光线追踪相较于光栅化可微渲染器的优势：光追渲染器支持全局光照。某些相关工作甚至仅利用物体产生的影子，就可以把物体的形状还原出来。

然而没有免费的午餐，虽然光线追踪可微渲染器的功能更加强大，但它比基于光栅化的渲染器速度更慢，而且占用的显存也更多，尤其是采样数量和光线弹射次数增加时，显存占用也会线性增长。

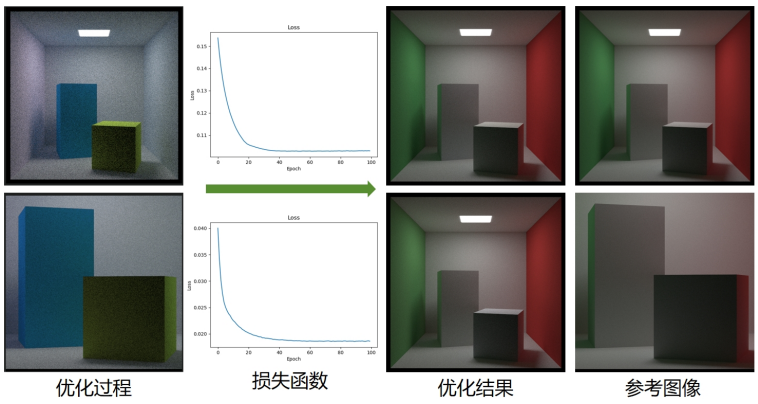

图 6. 与 mitsuba3 渲染时间对比

6 总结与展望

本次复现工作实现了一个完全以 Python 编写的基于物理的可微分渲染器的简单框架，此渲染器支持四种材质类型：漫反射、高光、光滑介质、发光材质；支持三种图元：长方形、三角形、球形。可以模拟相机的景深效果，支持导入 obj 格式的网格模型，可为模型添加棋盘格纹理，并且支持可微分渲染，可以将材质颜色设置为可优化的参数。

编写一个功能完备的可微分渲染器是一件较为庞大且复杂的工程，目前完成的渲染器仅是一个雏形，不论是传统的渲染技术，还是可微分渲染技术，都有很多重要的算法有待添加到此渲染器中。例如对于可微分渲染，有一些工作队路径追踪的梯度传递方法做出了优化，可以提高反向模式的性能，此外优化物体的形状也是一个比较普遍的需求，有多种方法可以实现。对于前向渲染部分，亟须实现的就是空间划分结构，例如 BVH 树，此数据结构可以大大减少复杂场景的渲染时间，使用硬件光追加速也可以大幅提升程序性能，复杂模型渲染是目前此渲染器的最大性能瓶颈，此外还要实现：纹理贴图，重要性采样，更多基于物理的材质，体渲染等功能。

此项目的另一个潜力是可以应用到 NeRF 和 3D 高斯溅射中，对于这两种渲染方式，科研工作者多使用 CUDA 程序编写前向渲染和反向传播过程，没有使用深度学习框架的自动微分功能，人为编写梯度的计算方式很容易出错，开发效率也很低，使用 Dr.Jit 理论上也可以高效地实现这些项目，而且可利用它的自动微分功能降低出错的可能性，以此项目为基石，未来甚至可以尝试在 3D 高斯或 2D 高斯上做路径追踪。

参考文献

[1] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. Dr.jit: A just-in-time compiler for differentiable rendering. *Transactions on Graphics (Proceedings of SIGGRAPH)*, 41(4), July 2022.

[2] Steve Hollasch Peter Shirley, Trevor David Black. Ray tracing in one weekend, August 2024.