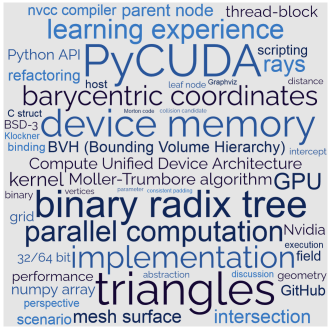
arXiv:2305.01867v2 [cs.DC] 4 May 2023

# PyCUDA 实践：重构现有的射线-曲面求交算法实现

**Raymond Leung**

澳大利亚现场机器人中心 悉尼大学工程学院




2023年5月3日

**摘要**

本文是《CUDA 中射线-曲面求交算法的 GPU 实现》（arXiv:2209.02878）[1]的续篇。本文重点介绍 PyCUDA，它是在统一计算设备架构（CUDA）框架中实现 GPU 运行时代码生成的一种 Python 脚本编程方案。本文配有在GitHub 上发布的开源代码，其中提供了基于 GPU 的线段与曲面三角形相交测试的 PyCUDA 实现。本文旨在与刚接触 PyCUDA 的读者分享学习 PyCUDA 的经验。我们以现有 CUDA 代码和 [1] 中的基础实现为起点，记录了为迁移到 PyCUDA 所做的关键修改。射线-曲面求交测试的 CUDA 源代码同时包含主机代码和设备代码，并使用多个内核函数，因此本文提供了一个内容充实的实例，让读者从实际角度了解使用 PyCUDA 的体验。文章还深入探讨了二进制基数树等自定义数据结构，并指出了一些可能的陷阱。案例研究介绍了一种调试策略，可使用标准 Python 工具检查设备内存中的复杂 C 结构体（例如三角形网格包围体层次结构），而无需使用 CUDA-GDB 调试器。

## 1 背景

PyCUDA 代表了一种基于脚本的 GPU 运行时代码生成方法。其设计理念和技术要素此前已由其架构师及合作者在 [2]和 [3]中介绍。本文围绕 PyCUDA 的实际应用展开讨论，并从解决一个特定几何问题的角度进行阐述：确定一组射线是否与三角网格曲面相交。为简单起见，我们用“射线”指代由起点和终点定义的线段；它可以写作 $l_i = (\mathbf{r}_i^{\text{start}}, \mathbf{r}_i^{\text{end}})$。典型问题涉及大量射线，例如 $N_\text{r} \in [10^6, 10^8]$ 条，因此并行计算非常有必要。网格曲面包含 $N_\text{t}$ 个三角形（通常为 $N_\text{t} \sim 10^4$），每个三角形由一个三元组 $\mathbf{t}_j = [t_{j,1}, t_{j,2}, t_{j,3}]$ 描述，该三元组引用了集合 $\{\mathbf{v}_n\}_{1 \le n \le N_\text{v}}$中的三个顶点 $\mathbf{v}(\mathbf{t}j) \equiv [\mathbf{v}_{t_{j,1}}, \mathbf{v}_{t_{j,2}}, \mathbf{v}_{t_{j,3}}]$。Möller–Trumbore 算法 [4]为这一问题提供了一种高效的解决方案。然而，如果不使用加速结构，穷举搜索 [将 $N_\text{t}$ 个三角形与 $N_\text{r}$ 条射线] 逐一比较，速度会非常慢，即使计算是在图形处理器上进行的也是如此（参见 [1]中的表 1）。

在现有的 CUDA实现中，使用莫顿码对所有网格三角形的位置进行编码。随后对这些三角形重新排序，以保持空间邻近性，并使用二进制基数树将其表示为包围体层次结构（BVH） [5]。这样便可通过树搜索，快速识别出一小部分三角形 [碰撞候选对象]，它们可能与每条射线相交。总体工作流程如图 2所示。该实现的核心要素在 [1] 的第 2 节和第 3 节中进行了介绍，也可在GitHub上交叉参阅。[1]

所需的输入和计算组件如示意图所示。并行进程（内核代码）用 // 符号标记。程序可配置为以不同模式运行，以支持三个用例。

[1]现有 CUDA 实现：https://github.com/raymondleung8/gpu-ray-CUDA 中的表面求交。

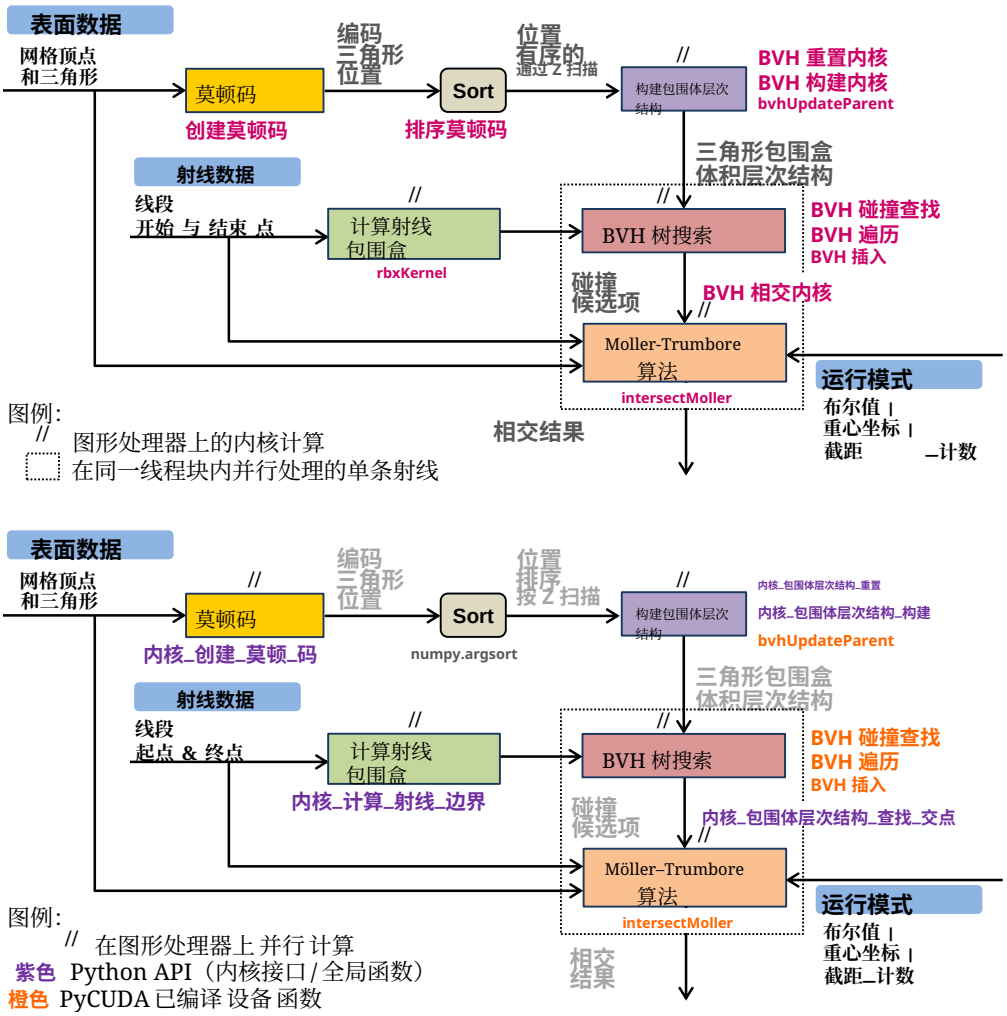



图 1：（上）CUDA 和（下）PyCUDA 实现中的射线与表面相交检测工作流程

- 标准用法（模式=布尔值）——返回一个（$N_r$,1）布尔数组，用于指示每条射线是否与表面相交。
- 扩展用法（模式=重心坐标）— 返回与三角形相交的射线，以及到最近交点的距离和该交点的位置。
- 实验性功能（模式=截距_计数）— 返回每条射线与表面的唯一交点数量。

本文其余部分安排如下。第 2 节概述 PyCUDA 的核心功能和安装步骤，并给出一个新 PyCUDA 库的用法示例。第 3 节聚焦于实现问题，并介绍为使现有 CUDA 代码能够在 PyCUDA 中运行而进行的修改。第 4 节总结经验教训，并介绍一种调试方法，可使用 Python 工具检查驻留在设备内存中的 C 结构体。

## 2 预备知识

### 2.1 PyCUDA：它是什么，为什么要使用它?

如引言中所述，PyCUDA 提供了一种集成于 Python 的方法，可在运行时生成图形处理器代码。PyCUDA 让程序员能够以符合 Python 风格的方式访问 Nvidia 的并行计算 API [6]。它允许将原始 CUDA 代码作为字符串嵌入s 到 Python 脚本中，再由 pycuda.compilerSourceModule解析，使用 nvcc（CUDA 编译器驱动程序）编译 编译器

SourceI 会返回一个模块对象 m，该对象通过 m.get_function(<cuda e>) 使用
CUDA 内核

> 在 CUDA 术语中，__全局 在 Nvidia GPU 上执行的内核代码。内核调用的所有依赖函数都使用 __设备__ 说明符声明。

根据 PyCUDA 文档，可以注意到几个特性

- 资源管理 与对象的生命周期绑定（类似于 C++ 中的 RAII 惯用法）。PyCUDA 了解依赖关系，不会在上下文中分配的所有内存释放之前与该上下文分离。
- 自动错误检查 ——PyCUDA 对参数类型以及用于内核 API 的 Boost Python 绑定要求严格。CUDA 错误会被转换为 Python 异常。
- 类似 *Numpy.array* 的抽象，使用 进行向量运算/基本算术运算。
- 易于访问——用户可以完全自由地使用所需的内核 API。
- 速度 — PyCUDA 的基础层是用 C++ 编写的，因此对吞吐量性能的影响微乎其微。

作者认为，PyCUDA 可以让设备代码更容易调试。更多内容请见第 4 节。

### 2.1.1 PyCUDA 与 CuPy 和 Numba CUDA 的比较

PyCUDA 专为希望将代码集成到 Python 中的 CUDA 开发者而设计 [现有的]CUDA 代码。对于只希望让基于 NumPy 的代码在 GPU 上运行的机器学习开发者，CuPy 则提供了另一种选择。它让用户无需学习 CUDA 语法即可受益于 GPU 的高速计算 [7]。

对于愿意拥抱 CUDA 且倾向于使用 Python 语法编程的人来说，还有 适用于 CUDA GPU 的 Numba，它保留了 CUDA 的概念，但会按照 CUDA 执行模型，将受限子集的 Python 代码转换为 CUDA 内核和设备函数。对于 PyCUDA，这些内核和设备函数是用 CUDA/C 语言编写的；因此，限制较少且可直接部署到 CUDA 中，可能是一项优势。

### 2.1.2 同时使用两个版本的源代码

要比较两个不同版本源代码产生的结果，我们可以同时使用这两个版本，而不必在 Git 分支之间切换，依次编译和测试代码。相反，可以在 Python 环境中使用以下工作流。[2] 当我们需要确保增量更改不会在开发过程中改变测试结果时，这可能会很有用。可以使用 NumPy 比较结果
函数及其差异都可以轻松识别。此情形如运行_示例3 见 pycuda/demo.py

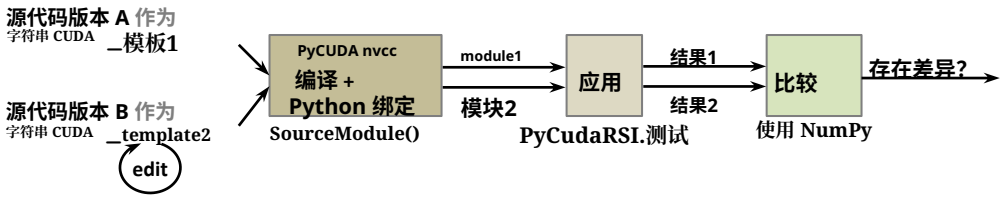



图 2：比较 PyCUDA 环境中不同源代码版本的结果 t

在 try 于 用于执行_s nel_相交_* 相关的条件语句则共同用于处理 码（CUDA 模块的 A 版本）中不存在的内核被加入了开发代码（B 版本）中。

## 2.2 PyCUDA 安装

前置条件已列于 PyCUDA 官方网站。总体而言，PyCUDA 需要 a

- 支持 CUDA 的 Nvidia GPU
- 可正常使用的 Python 安装（在我们的案例中为版本 3.8）

[2]为简洁起见，图中未显示编译器缓存。请参阅 [2] 中 GPU 二进制文件的缓存（见图 2）。

- C++ 编译器，最好是GCC和
- 访问 Nvidia 的 CUDA工具包

尝试 pip安装 之前，应先设置几个环境变量。对作者而言，执行以下命令后成功完成了安装。

```
#!/ bin / bash

$ export PATH =/ usr / local /cuda -11.2/ bin$ { PATH :+: ${ PATH }} $ export LD_LIBRARY_PATH =/ usr / local / cuda -
```

我的虚拟环境中的 Python 软件包列于 pycuda/requirements.txt。

### 2.3 PyCUDA 射线-表面相交库：用法

新的 PyCUDA 库（不要与 [1] 中现有的 CUDA实现 混淆）使用方式如下。模式 的有效选项为 '布尔值'、'重心坐标' 和 '截距_计数'。

```
from pycuda_射线_表面_intersect import PyCudaRSI

设计_参数 = 字典 ()
cfg ={'模式 ': 模式 }

使用 PyCudaRSI（design_params）作为 pycu:
    if mode != '重心坐标 ':
        射线_相交于 =       pycu . 测试 ( 顶点 ,       三角形 ,       射线起点 ,    射线终点 , 配置 )
    否则:
        （相交的_条射线,          距离 ,         命中_三角形,        命中_点 ) \
            = pycu . 测试 ( 顶点 ,       三角形 ,       射线起点 ,    射线终点 , 配置 )
```

请参阅 pycuda/demo.py 以获取更多注释和诊断功能。

### 2.4 PyCUDA 射线-表面相交：源代码

供参考，PyCUDA射线-表面相交库可在以下位置获取。

https://github.com/raymondleung8/gpu-ray-surface-intersection-in-cuda/tree/main/ pycuda/

它基于 BSD 三条款许可证以开源形式发布。

## 3 PyCUDA 实现

核心功能实现在 pycuda_ray_surface_inte t_cuda_template() 从 pycuda_source.py 获取 C++/CUDA 源 个模块字 码与 "rsi_geometry.h"、"morton3D.h" 和 "bvh_structure.h" 中的代 从 拆分 pycuda_ray_surface_intersect.py 中的 PyCudaRSI 类更易读。 RSI ray-surface_intersect.cu 中的代码（来自 [1] 的现有 CUDA实现），并提供支持三种主要用例的高级接口。

### 3.1 为 PyCUDA 重构现有 CUDA 代码

默认情况下，PyCUDA 会将 n 为 False。这样会将 `cuda_template` 中指定的整个源代码包裹在 `extern "C" {}` 中 名称修 所需的 API。这意味着，如果保留来自 `#include <vector>` `"bvh_structure.h"` 中的语句 [注意：标准模板库（STL） 码，从不用于设备代 样的链接错误

> include/c++/6.2.1/bits/stl_iterator_base_types.h(116)：错误：此
> 声明不能使用 extern "C" 链接。

PyCUDA 也不支持重载函数。虽然它允许使用模板化的 __设备__ 函数，但模[illegible]数（例如 模板 <类型名 M> __全局__ void bvhCons[illegible]码有所不同。

### 3.2 具体更改

- 来自“bvh_structure.h”的主机函数 void i[illegible]型[illegible]__ kernelMortonCode(const [illegible].)。

  –在 Python 中，通过绑定内核_create_morton_code =m.get_funct[illegible]可[illegible]置转换为 64 位 MORTON 码。

- 莫顿码的排序本质上是将三角形组织成局部簇，以保持空间邻近性，不再使用 C++主机函数执行。而是在 Python 中完成排序，随后将结果复制到设备内存 d_已排序的三角形 ID 。

- 从“morton3D.h”中移除了模[illegible]。取而代之的是，在由 pycuda.compilerSou[illegible]解析之前，使[illegible]_模板中的类型 MORTON、COORD 替换为具体类型（例如 {"MORTON":

- 最后，现有的 CUDA实现大量使用了自定义数据结构（例如轴对齐包围盒 AABB、BVH节点、CollisionList 和 InterceptDistances ）。
  对于 float* 这类原生类[illegible]以直接使用 d_vertices= cuda.mem_alloc(h_vertices.nbytes) [illegible]型为numpy.float32、形状[illegible]ices,3)的 numpy.array。
  [illegible]，则定义了[illegible]ruct_size(cuda_szQuery,d_answer)，[illegible]
  [illegible]给 Python。为了使用 PyCUDA 在图形处理器设备上分配全局内存，必须事先知道这一大小。

  – 这里，[illegible]指的是 Python 接口，它对应于诸如 __global__voidbytesIn[illegible]x)[illegible]- 不应对字节大小进行硬编码，因为编译时可能会引入内存填充，而且其范围可能因设备架构而异。建议使用 __al[illegible]编译器说明符（或其他变体），以确保 CUDA 内存对齐保持一致。

尽管存在这些差异，但 pycuda_ray_su[illegible]现有代码非常相似的 CUDA 实现。

### 3.3 性能

程序的速度和正确性始终是首要考量。本节首先考察速度，并找出可改进之处。下表比较了原始 CUDA 实现[1] 与 PyCUDA 的执行时间。对于 PyCUDA，经过的时间（参见 t_start 和 t_end，位于 pycuda_ray_surface_intersect.py中）还包括 I/O 开销（例如由 PyCUDA 协调执行的 memcpy_设备到主机操作），以及任[illegible]重[illegible]模式下运行时的 NumPy 处理。相比之下，原始 CUDA 程序会将结果写入二进制文件；不会计算派生量（例如交点）。总体而言，在 布尔值 模式和 截距_计数 模式下，[illegible]间相

测试数据由 " diagnostic_input . py " 包含 1000 万条射线和一个包含 14874 个顶点的表面，29260 个三角形。

| 运行模式 | 原始 CUDA 代码 | PyCUDA 实现 |
|---|---|---|
| 布尔值 | 300.166 ms | 355.094 ms |
| 重心坐标 | 308.625 ms | 909.376 ms |
| 截距_计数 | 334.192 ms | 383.117 ms |

下表对重心坐标模式进行了更详细的分解。

[3]相比之下，CuPy 支持 类型泛型内核以及 C++ 模板化内核，其中未修饰的内核名称和模板参数通过 name_表达式进行明确指定。名称修饰和反修饰均在底层自动处理。

| 组件 | 描述 | 持续时间 |
|---|---|---|
| 1 | Core 处理中 * | 138.513 ms |
| 2 | memcpy_设备到主机 (三角形相交, 重心坐标变换) | 231.357 ms |
| 3 | 正在计算 交点/距离 | 539.506 毫秒 |
| 3a | 识别 相交 rays (NumPy) | 34.562 ms |
| 3b | 计算 相交 距离 (主机) | 240.836 ms |
| 3c | 保留 相交 三角形 (主机) | 9.246 ms |
| 3d | 计算 交点 (主机) | 254.862 ms |

* 从 t_开始执行完成_包围体层次结构
这包括：配置_，分配_内存_，传输_数据_，
获取_最小值_最大值_表面的_范围_，内核_计算_射线_边界，
内核 -创建 -morton _code，内核 _bvh_重置 ，内核 _bvh_构造。

基于奥卡姆剃刀原理，组件 3b 和三维的主机代码被转换为内核代码，以提高效率。这些内核函数（kernelIntersectDistances 和kernelInters urce.py）从下面的运行时间著的改进。

| 组件 | 描述 | 持续时间 | |
|---|---|---|---|
| 3b | 计算 相交 距离 (图形处理器) | 18.039 ms | |
| 3d | 计算 交点 (图形处理器) | 24.629 ms | |
| 3 | 正在计算 交点/距离 | 86.476 毫秒 | （减少 84%） |
| * | 重心坐标总持续时间 | 456.346 毫秒 | (50% reduction) |

接下来，我们将深入探讨调试和验证工作。

## 4 条经验教训

PyCUDA 的一个显著优势是，它能让设备代码更容易调试。这在一定程度上得益于可以轻松地将设备内存复制到主机，以 numpy 数组的形式进行检查。不过，开发人员也需要留意一些陷阱。在接下来的讨论中，我们将探讨两种情况：一种涉及异常测试结果，另一种则因网格划分而导致程序崩溃（构建二进制基数树时的疏忽）。这两个问题都源于内存使用问题，可以通过检查三角形网格包围体层次结构的内容来诊断。

### 4.1 案例研究 1：调试来自 devicememory 的 C 结构体。CUDA 和 NumPy 中“整数类型”的含义

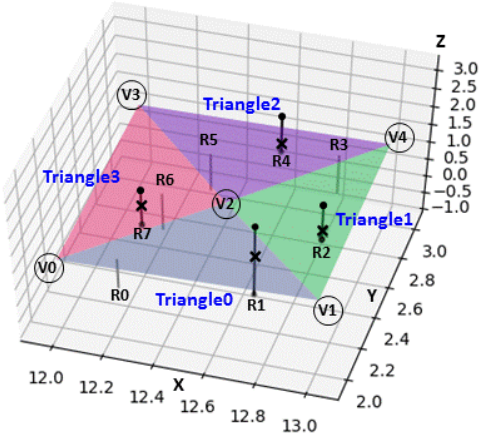



图 3：包含 8 条射线和 4 个三角形的简单测试配置

此图描绘了一个测试场景，其中网格曲面由 4 个三角形组成，并有 8 条射线（线段）。具体而言，$R_1$, $R_2$, $R_4$ 和 $R_7$ 与三角形 $T_0$, $T_1$, $T_2$ 和 $T_3$ 相交于 [12.7,2.2,1.14], [12.9,2.4,1.21], [12.6,2.9,1.23]

以及 [12.2、2.4、1.08], 。然而，由于一个错误，最初的 PyCUDA 实现仅报告 $R_1$ 和 $R_7$ 为相交射线。通过逐步排查，问题原因被缩小到 BVH 树的构建，以及对应莫顿码排序后对三角形 ID 的解读。

为诊断问题，我们检查了三角形包围体层次结构（BVH）的内容，旨在验证 BVH 的完整性并找出根本原因。具体分为以下两个步骤。

### 4.1.1 包围体层次结构（BVH）调试策略

1. 使用标准 PyCUDA I/O 将相关内容从设备内存传输到主机内存
2. 根据 `structBVHNode` 中给出的类型定义，解码每个 BVH 节点中的字段。

接口 s.

在当前情况下，我们的目标是收集所有信息，并检查图中的某些属性（例如连通性）。因此，这种策略提供了一种合适的替代方案，无需调用 `cuda-gdb`，因为它的许多丰富功能在这里并不需要。当然，这着它在一般意义上可以取代 `cuda-gdb`。




图4：一个包围体层次结构由 $N_t$ 个内部节点（最后一个内部节点仅保存指向根节点的指针）和 $N_t$ 个叶节点组成。每个内存数组都从图形处理器设备复制到主机（使用 32 位整数表示）。随后，根据每个字段的数据类型解码各个 BVH节点中的内容。

具体而言，我们创建两个 NumPy数组来保存内部节点和叶节点的内容。`cuda.memset_`设备到主机
换为 32 位整数 值。

```
h_叶节点  =np . zeros ( n_三角形 * sz_BVH节点 , dtype =np. 32 位整数 ) h_内部节点  =np.
```

为了操作字流（其中“字” = 4 是连续的字节），每个 BVH节点 的内存占用通过`struct_size` 宏 [确定，该宏定
`pycuda_射线_表面_intersect.py` 除以`numpy.int32` 中的字 l32 位整数元素数量。

```
sz_BVH节点  =整数类型( 结构体_大小 ( 字节_在_BVH节点 , d_szQuery ) / np . ones (1 ,
```

每个 BVH节点 （包 节点的块） isplay_node_contents方法逐个处理，该
pycuda/dia 结构体的定义解释字段值。

BVH节点。出于调试目的，参数['USE_EXT ) 指明左/右子 （左子节点 和右子节点 节点）是“内部”节点还是“叶节点” 节点； ) 包含父节点和当前节点自身的内存地址，从而更容易追踪二进制基数树结构。

```
for t in range ( n_三角形 ): words    =h_内部节点  [t* sz_BVH节点 :(t +1)* sz_BVH节点 ]显示
```

提炼出关键信息后，可以从上述代码看出，第一个三角形 $T_0$ 在叶节点中重复出现了。此外，除了 $T_3$, , $T_1$ 和 $T_2$ 的三角形 ID 也缺失了。

```
叶节点
[0] x : [12.0 ,13.0] , y: [2.0 ,2.5] , z : [1.0 ,1.2] , 三角形 ID : 0, 左范围 : 0, 右范围 : 0[1] x : [12.0 ,13.0] , y: [2.0 ,2.
```

### 4.1.2 问题根源

在 t 参数 sortedOb 整数类型* 。 UDA 中，这会被视为一个 整数类 组 .

```
__global__ void kernelBVHReset (const 浮点数*  --限制--  顶点,
                                const 整数类型 *  __restrict__  三角形,
                                BVH节点 *  --restrict  --  内部节点 ,
                                BVH节点 *  --限制--限制  --  叶节点 ,
                                整数类 --限制--  sortedObjectIDs,  整数类型 节点数量)
{
    无符号整数类型 i   =blockIdx.x * blockDim.x   +threadIdx .x;
    如果 (i   >=  节点数量)
        返回;
    // 在叶节点中设置三角形属性
    叶节点 [i]。三角形 ID   =t   =          sortedObjectIDs [i];
    :
}
```

这与.....不兼容

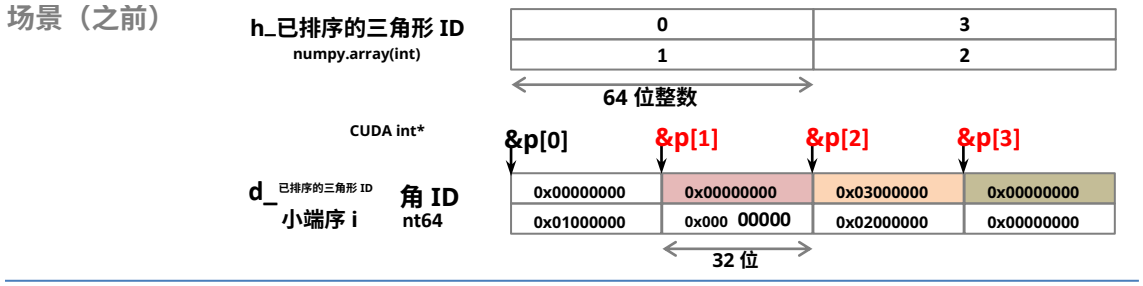
```
d_已排序三角形 ID   =cu      da . mem_alloc ( n_三角形 * np . ones (1 , dtype =np . 64 位整数 ). nbytes ) #(**)
```

它作为 sortedObjectI :数据类型遵循.....的约定

```
h_sortedTriangleIDs   =np. argsort ( h_morton ) #(**)
```

在 Python 3 中，返回的数组类型 整数类型 认为 64 位整数，适用于 64 位机器。

因此，将 p[i] 作为 tedObjectIDs[i] , 的 的 int32 指针递增速度只有所需速度的一半（每次前进 32 位）。在第三次“读取”时，p[2] 获取的是第二个 64 位整数的最低有效字，而该整数恰好引用 三角形 $T_3$。



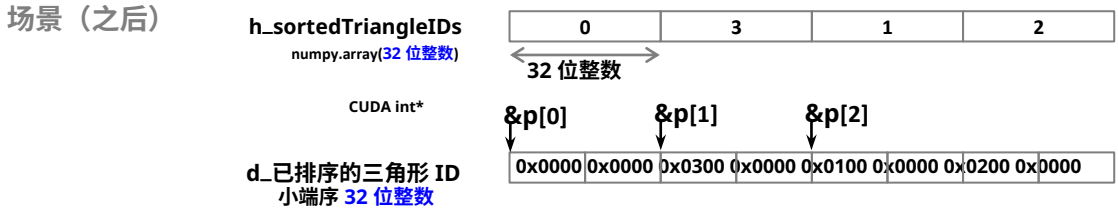



图 5：CUDA int* 从 64 位数组中读取 32 位整数时，内存访问表现出的不协调性。这种情况的出现，是因为设备数组的内存分配依据的是 numpy.argsort 返回的主机数组大小，而 numpy.argsort 将 int 解释为 int64。与此同时，在 CUDA 中，编译器将 int 视为 int32。

### 4.1.3 解决方案

要解决这个问题，（CUDA 中 32 位整数指针的）步幅需要与设备内存中整数元素的宽度一致。可以将上文中的 **(**)** 行替换为以下内容来实现：

```
h_排序后的三角形 ID  =np . argsort ( h_morton ). astype ( np. 32 位整数 ) d_排序后的三角形 ID  =cuda . mem_alloc ( n_三角形
```

最终的 PyCUDA 实现包含这些更改，并会生成调试输出，其中每个叶节点都对应一个不同的三角形。

```
BVH 树结构

内部节点
[0]  x :[12.0 ,13.0] ,    y: [2.0, 3.0] ,  z: [1.0, 1.2]    <<< 边界盒      体积
自身: 0 x66e1200,     父节点: 0 x66e1260
索引:        0（自身）, 0（左叶节点）, 1（右叶节点）
左子节点: 0x66e1000   ,  右子节点: 0 x66e1060
原子 : 2,        左范围 : 0,      右范围 : 1

[1]  x :[12.0 ,13.0] ,    y: [2.0, 3.0] ,  z: [1.0, 1.3]           ROOT NODE
自身: 0 x66e1260,     父节点:
索引: 1（自身）       , 0（L - 内部）       , 2（R - 内部）
左子节点 : 0x66e1200   ,  右子节点: 0 x66e12c0
原子: 2,        左范围 : 0,      右范围: 3

    [2] x :[12.0 ,13.0] , y :[2.0 ,3.0] , z :[1.1 ,1.3]自身 :




[3]  x :[0.0 ,0.0]   ,  y :[0.0 ,0.0]   ,  z: [0.0, 0.0]
自身: 0x0,     父节点: 0x0
索引: 3（自身）       , 1（L - 内部）       , 0(R - 内部 )
左子节点 : 0 x66e1260 (根节点)          ,  右子节点 : 0x0
原子: 0,        左范围 : 0,      右范围 : -1


叶节点
[0]  x :[12.0 ,13.0]   ,  y: [2.0, 2.5] ,  z: [1.0, 1.2]
自身: 0 x66e1000 , 父节点 : 0 x66e1200
三角形 ID:  0

[1]  x: [12.0, 12.5] ,  y: [2.0, 3.0] ,  z: [1.0, 1.2]
自身:  0 x66e1060,     父节点:  0 x66e1200
三角形 ID : 3

[2]  x :[12.5 ,13.0]   ,  y: [2.0, 3.0] ,  z :[1.1 ,1.3]
自身: 0 x66e10c0 ,     父节点: 0 x66e12c0
三角形 ID:  1
```

```
[3] x: [12.0, 13。0], y: [2.5, 3。0], z: [1.1, 1。3]
```

此外，程序在“布尔值”模式下正确报告了 $R_1$,$R_2$,$R_4$ 和 $R_7$ 为相交射线。

```
0: 0      1: 1      2: 1      3: 0      4: 1      5: 0      6: 0      7: 1
```

### 4.1.4 测试代码和 Graphviz

运行 `pycuda/demo.py` 即可复现这些结果。设 ualisatio 会 iz 为网格三角形二进制基数树生成一个图。

- 相关代码 bvh_gra cuda/diagnc

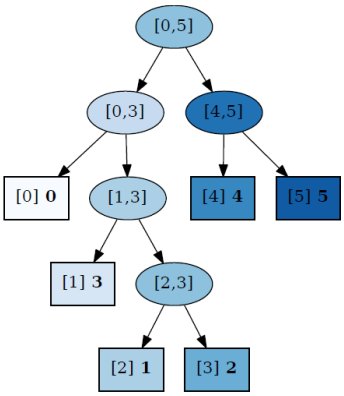



图 6：使用 Graphviz 生成的二进制基数树图

上面的图对应于截距_计数 模式中使用的测试表面。 的三角形 1 和 2 上方添加了两个面片，形成一个“顶篷”。对于根节点和内部节点， `[a,b]`表示节点索引范围。对于终端节点（正方形）， `[c]` 的c 表示叶节点索引，而d表示三角形 ID（计入使用莫顿码重新排序 况）。

- 此设置还会将一个图的描述以 DOT 语言写入文件（包围体层次结构_结 .

### 4.2 案例研究 2：未充分关注网格大小。未正确构建 BVH 结构时发生非法内存访问

使用 合成_数据 从 `pycuda/diagnostic_input.py` 创建了一个包含 ~30,000 ) 万条 的构造方式，在 布尔值 模式下 I.test API。产生 错误消息。

```
回溯（最近一次调用最后）： 文件 "/ opt / python3 .8/ lib / python3 .8/ runpy . py"，第 193 行，在 _run_
```

### 4.2.1 调查

为调查此问题，我们按照 BVH 调试策略再次检查了树的内容。通过设置 配置[“检查_bvh”] =True 暂时启用该选项后，即可获得打印输出。

In t在这种情况下，包围体层次结构要复杂得多。图 7 展示了该图的一段截断部分。

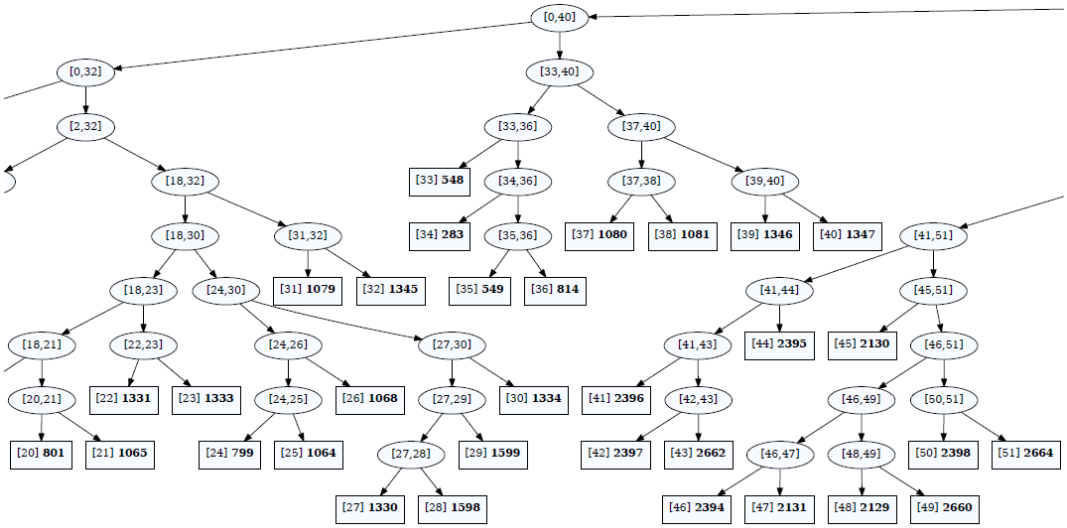



图 7：用于 cas 中的三角形网格二进制基数树的图的一段截断部分 e 2

下一张图片（图 8）展示了我们从树的顶部向下遍历时，包围体层次结构的样子。根节点视为第 0 层，其直接子节点构成第 -1 层，以此类推。

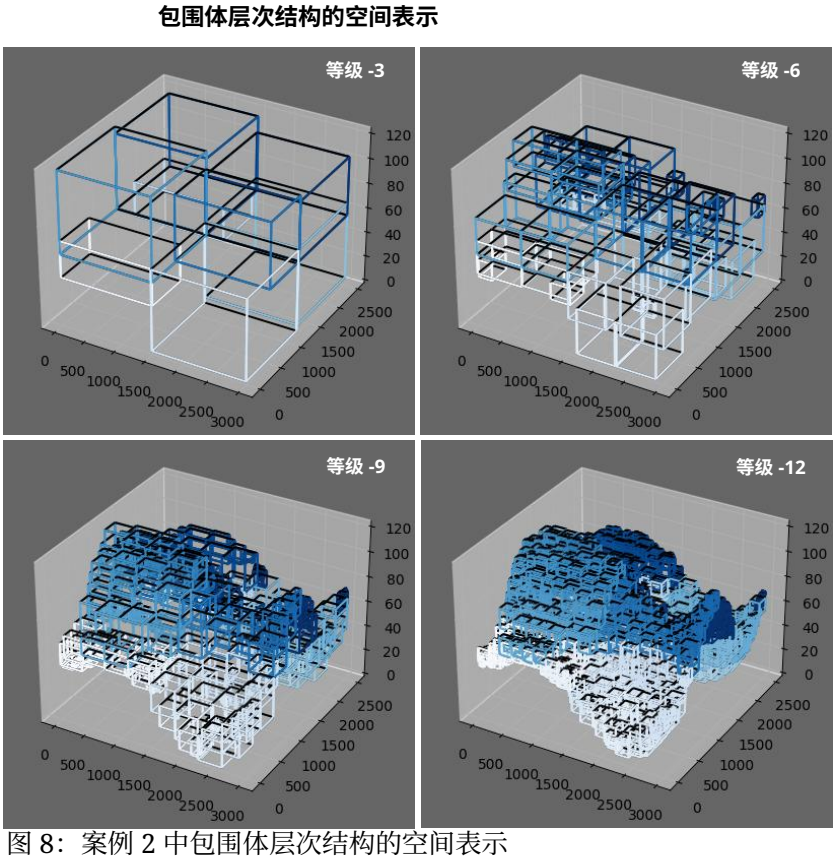



图 8：案例 2 中包围体层次结构的空间表示

### 4.2.2 观察

关键发现如下。

```
BVH 树结构

内部节点
    :
[16382] x:[992.584,1018.47], y:[363.936,387.745], z:[61.0106,62.1155]
自身: 0 xe2361ff40 , 父节点 : 0 xe2361ffa0
索引 :  16382( 自身) , 16382( 左叶节点 ), 16383( 右叶节点 )
左子节点 : 0xe2335ff40 , 右子节点: 0 xe2335ffa0
原子 : 2,  左范围 :  16382,  右范围:  16383

[16383] x: [0,0], y: [0,0], z: [0,0]
自身: 0x0,  父节点 : 0x0
索引 : 16383( 自身)  , 16382( L - 内部 )  , 0(R - 内部 )
左子节点 : 0xe2361ff40 , childR : 0x0
原子:  1, 左范围:  16382,  右范围: -1

[16384] x: [0,0], y: [0,0], z: [0,0]
自身: 0x0,  父节点: 0x0
索引: 16384（自身） , 0（L - 内部） , 0（R - 内部）
左子节点 : 0x0 , 右子节点 : 0x0
原子 : 0
    :
[29259] x:[0,0], y:[0,0], z:[0,0]
自身: 0x0,  父节点: 0x0
索引: 29259（自身） , 0（L - 内部） , 0（R - 内部）
左子节点: 0x0 , 右子节点: 0x0
原子: 0
```

对于内部节点，

- 存在诸如 16382 这样的节点，其所有属性都已定义。
- 然而，存在诸如 16383 这样的“半填充”节点。信息已从其左子节点向上传播，但尚未连接到其右子节点。因此，原子计数器的值为 1。
- 事实上，有些节点（例如 16384）尚未以任何方式填充。对于二进制基数树，`n_triangles- 1` 个内部节点应始终
- 最后，最后一个内部节点（索引为 `29259` `childL` 中保存根节点的地址。只有自底向上构建 顶端时，才可能出现这种情况。
- 如果根节点是空指针，BVH 将无法遍历。

```
叶节点: [16383] x: [992.584, 1018.47] , y: [363.936, 387.745] , z: [61.2898, 62.
```

对于叶节点，

- 存在诸如 16384 这样的节点，它们未与其父节点连接。
- 这意味着 __ le odes) 并未在这个叶节点上执行，尽管它已由 kernelBVHReset 初始化了三角形 ID。

### 4.2.3 问题根源

由于 只会递归调用自身或 kernelBVHConstr 出在 self.kernel_包围体层次结构_构建 的调用上。检查 p y 后发现，网格 配置有误。

```
自身 . 内核_包围体层次结构_construct (
                self . d _内部节点 , 自身 . d _叶节点 , 自身 . d _morton ,
                np . 32 位整数 ( 自身 . n_三角形 ), 块 =自身 . 块_维度 , 网格 = 自身 .网格_lambda)
```

应读取

```
自身 . 内核_包围体层次结构_构建 ( 自身 . d_内部节点 , 自身 . d_叶节点 , 自身 . d_morton , np. 32 位整数 ( 自身 . n_
```

根据定义

```
自身 . grid_lambda =( 自身 . grid_xLambda , 1) 自身 . grid_dimsT =( 整数类型 ( np. ceil ( 自身 . n_三
```

执行程序时，控制台还会显示

```
CUDA 分区：1024 个线程 / 块，网格： [射线：9766，包围体层次结构_构建：29，包
```

也就是 网格. 于内 格树构建相关，因此分区应取决于网格_dimsT （等价于n_三角形）。 应取决于网格_λ 与__设备__ voidbvhFindCollisions( ... , void kernelBVHIntersection 相关的 $N_{\text{grids}}$ 参数。这是一次

### 4.2.4 解决方案

对于这个测试用例 格_ 释了为什么某些叶节点未被处理，以及相关数据（体积边界和索引）为何没有向树的顶层传播。它执行的工作量少于预期。最终的 PyCUDA 实现包含了上方以蓝色标出的错误修复。

## 5 总结

这些示例表明，PyCUDA 可以让内核代码更易于调试。借助 PyCUDA 的抽象机制和标准 Python 工具，可以轻松地将数据从设备内存传输到主机，并检查对象。对于包围体层次结构等更复杂的 C 结构体，则需要一个解码层/方法，以便解析构成内部节点数组和叶节点数组的各个BHVNodes 中的混合类型内容。拥有如此强大的能力也意味着承担相应的责任。开发人员应当注意一些潜在的问题。在第一个案例研究中，我们注意到 整数类型存在不同的解释方式——Nvidia CUDA 编译器（nvcc）将 int 视为 i ¯2，而 numpy 使用 int64。这种差异会导致不匹配的情况（参见图 5），使得 int32* 指针落后于实际的 int64 数据。因此，网格三角形会从二进制基数树中“消失”，并且在测试场景中，某些射线-表面相交也会被错误检测。这一问题可能以其他方式表现出来，并在不同的上下文中引发其他异常行为。在第二个案例研究中，我们遇到了一个异常。我们采用了相同的调试策略，并成功找到了该异常的根本原因。我们发现，网格大小指定不正确是导致 BVH 构建不完整的原因之一。通过检查 BVH 结构，我们清楚地发现，某些叶节点属性（体积边界）未能传播到树的顶层。最终的 PyCUDA 射线-表面相交库已不存在这些问题。源代码遵循 BSD-3 条款许可证，可在 https://github.com/raymondleung8/ gpu-ray-surface-intersection-in-cuda/tree/main/pycuda 获取。

## 后记

目前，整个实现都使用 32 位浮点数。在 `intersectMoller` 函数中，[illegible]算相关的舍入误差可能会累积。对于射线平行或几乎平行于三角形面的边界情况，精度损失有时可能会产生影响。根据我们的经验，这种情况相对少见——发生率约为 $10^{-6}$ [这取决于近乎退化的交点测试数量]。如果这是一个问题，可以将 `PyCudaRSI.__init__(params)` 初始化为`params['USE_DOUBLE_PRECISION_MOL` [illegible] `cuda_source.py` 中启用 64 位浮[illegible]在 `__device__int in`

## 延伸阅读

Hwu 等人在 [8] 中介绍了 GPU 编程和 CUDA 概念。这本书是初学者的宝贵资源。 [2, 3] 讨论了基于脚本的 GPU 运行时代码生成方法（尤其是 PyCUDA）。Apetrei 在 [9] 中介绍了包围体层次结构和线性二进制基数树构建。Jiménez 等人在 [10, 11] 中综述并评估了射线-三角形相交测试。

## 致谢

本研究得到了力拓矿山自动化中心和澳大利亚现场机器人中心的支持。

## 参考资料

[1] Raymond Leung。CUDA 中射线-曲面求交算法的 GPU 实现。源代码可在 https://github.com/raymondleung8/gpu-ray-surface-intersection-in-cuda 获取，采用 BSD 3 条款许可证。文档地址：https://arxiv.org/abs/2209.02878。*arXiv* 预印本，第 1–11 页，2022 年。[2] Andreas Klöckner、Nicolas Pinto、Yunsup Lee、Bryan Catanzaro、Paul Ivanov 和 Ahmed Fasih。PyCUDA：面向高性能计算的 GPU 运行时代码生成。网址：http://arxiv.org/abs/0911.3456。*arXiv* 预印本 *0911.3456*，2009 年。[3] Andreas Klöckner、Nicolas Pinto、Bryan Catanzaro、Yunsup Lee、Paul Ivanov 和 Ahmed Fasih。使用 PyCUDA 进行 GPU 脚本编写和代码生成。网址：http://arxiv.org/abs/1304.5553。载于 Wen-Mei Hwu 编，*GPU Computing Gems*，第 *27* 章。爱思唯尔，2011 年。[4] Tomas Möller 和 Ben Trumbore。快速、低存储开销的射线-三角形相交。*Journal of GraphicsTools*，2(1)：21–28，1997 年。[5] Tero Karras。最大化包围体层次结构、八叉树和 k-d 树构建中的并行度。载于第四届 *ACM SIGGRAPH/Eurographics High-Performance Graphics* 会议论文集，第 33–37 页，2012 年。[6] Andreas Klöckner。PyCUDA 文档。网址：https://documen.tician.de/pycuda/，2022 年。[7] Ryosuke Okuta、Yuya Unno、Daisuke Nishino、Shohei Hido 和 Crissman Loomis。CuPy：兼容 NumPy 的 NVIDIA GPU 计算库。第 *31* 届神经信息处理系统会议，151(7)，2017 年。[8] Wen-Mei Hwu、David Kirk 和 Izzat El Hajj。大规模并行处理器编程：实践指南。Morgan Kaufmann，2022 年。[9]Ciprian Apetrei。快速简洁的凝聚式线性包围体层次结构构建。载于 Rita Borgo 和 Wen Tang 编，*EG UK* 计算机图形学与可视化计算。Eurographics 协会。网址：http://diglib.eg.org/bitstream/handle/10.2312/cgvc.20141206.041-044/041-044. pdf?sequence=1&isAllowed=y，2014 年。[10] Juan J Jiménez、Rafael J Segura 和 Francisco R Feito。用于干涉测试的稳健线段-三角形相交算法：效率研究。计算几何，43(5)：474–492，2010 年。[11] Juan J Jiménez、Carlos J Ogáyar、José M Noguera 和 Félix Paulano。基于 GPU 的射线-三角形算法性能分析。载于 *2014* 年国际计算机图形学理论与应用会议（*GRAPP*），第 1–8 页。IEEE，2014 年。