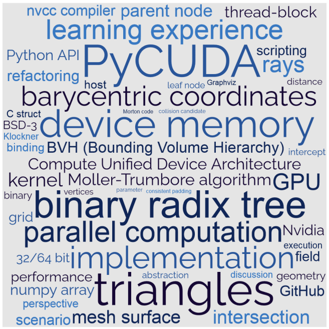
An experience with PyCUDA: Refactoring an existing implementation of a ray-surface intersection algorithm



Raymond Leung

Australian Centre for Field Robotics
Faculty of Engineering
The University of Sydney

May 3, 2023

Abstract

This article is a sequel to "GPU implementation of a ray-surface intersection algorithm in CUDA" (arXiv:2209.02878) [1]. Its main focus is PyCUDA which represents a Python scripting approach to GPU run-time code generation in the Compute Unified Device Architecture (CUDA) framework. It accompanies the open-source code distributed in GitHub which provides a PyCUDA implementation of a GPU-based line-segment, surface-triangle intersection test. The objective is to share a PyCUDA learning experience with people who are new to PyCUDA. Using the existing CUDA code and foundation from [1] as the starting point, we document the key changes made to facilitate a transition to PyCUDA. As the CUDA source for the ray-surface intersection test contains both host and device code and uses multiple kernel functions, these notes offer a substantive example and real-world perspective of what it is like to utilise PyCUDA. It delves into custom data structures such as binary radix tree and highlights some possible pitfalls. The case studies present a debugging strategy which may be used to examine complex C structures (such as a triangle-mesh bounding volume hierarchy) in device memory using standard Python tools without the CUDA-GDB debugger.

1 Background

PyCUDA represents a scripting-based approach to GPU run-time code generation. Its design philosophy and technical elements have previously been covered by its architect and collaborators in [2] and [3]. In this paper, PyCUDA discussion is centered around practicalities, and it is told from the perspective of solving a specific geometry problem, where the goal is to determine if a set of rays intersect a triangle mesh surface. For simplicity, we use the term 'ray' to refer to a line segment specified by a start and end point; this may be written as $l_i = (\mathbf{r}_i^{\text{start}}, \mathbf{r}_i^{\text{end}})$. A typical problem involves a large number of rays, say $N_\text{r} \in [10^6, 10^8]$, which makes parallel computation highly desirable. The mesh surface contains N_t triangles (typically $N_\text{t} \sim 10^4$), each described by a triplet $\mathbf{t}_j = [t_{j,1}, t_{j,2}, t_{j,3}]$ which references the three vertices $\mathbf{v}(\mathbf{t}_j) \equiv [\mathbf{v}_{t_{j,1}}, \mathbf{v}_{t_{j,2}}, \mathbf{v}_{t_{j,3}}]$ from the collection $\{\mathbf{v}_n\}_{1 \leq n \leq N_\text{v}}$. The Moller-Trumbore algorithm [4] provides an efficient solution to this problem. However, an exhaustive search [comparing N_t triangles with N_r rays] is very slow if acceleration structures are not used even when computation is done on a GPU (see Table 1 in [1]).

In the existing CUDA implementation, Morton code is used to encode the location of all mesh triangles. These are subsequently reordered to preserve spatial proximity and represented by a bounding volume hierarchy (BVH) using a binary radix tree [5]. This allows a small subset of triangles [collision candidates] that could possibly intersect with each ray to be quickly identified via tree search. As an overview, the general work flow is shown in Figure 2. Core elements of the implementation are described in Sections 2 and 3 in [1] and may be cross-referenced in GitHub.[1]

The required inputs and computational components are depicted in the diagram. Parallel processes (kernel codes) are labelled with the // symbol. The program may be configured to operate in different modes to support three use cases.

[1] Existing CUDA implementation: `https://github.com/raymondleung8/gpu-ray-surface-intersection-in-cuda`.

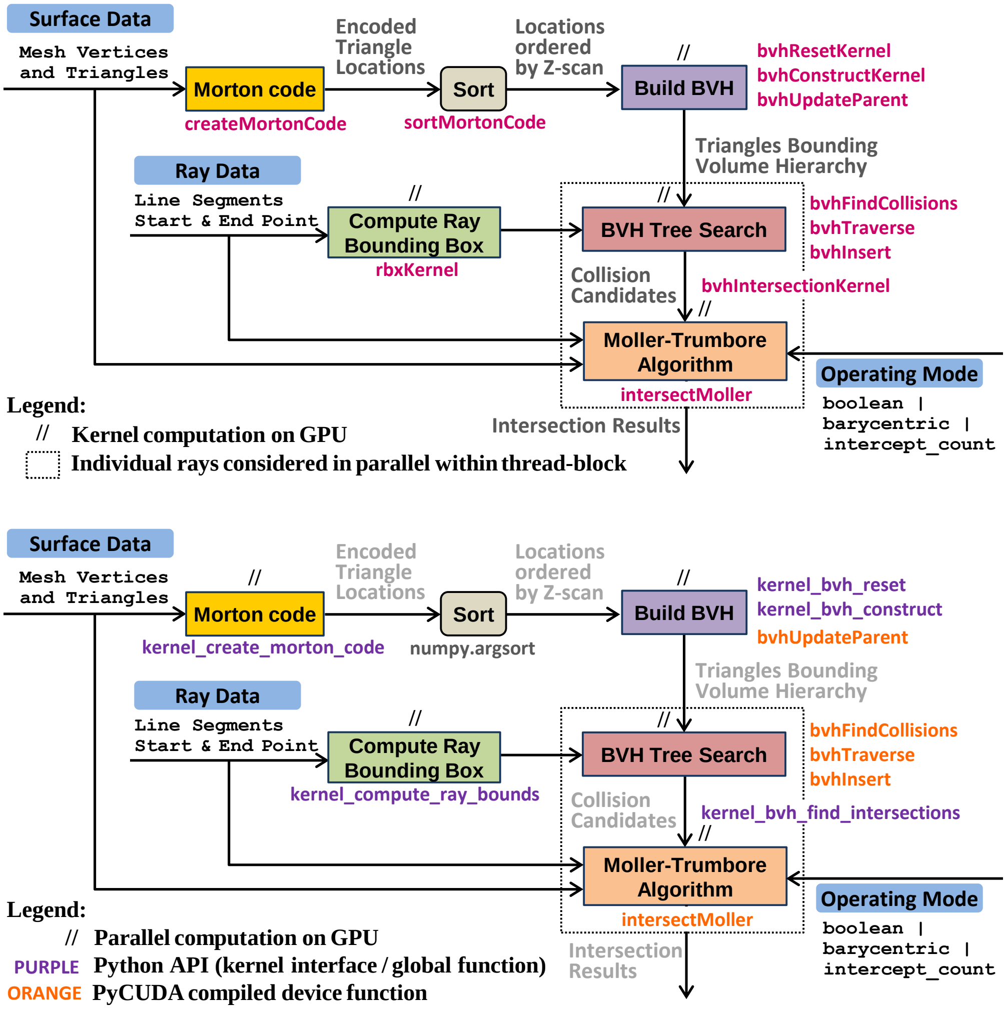

Surface Data
Mesh Vertices and Triangles
Morton code
createMortonCode
Encoded Triangle Locations
Sort
sortMortonCode
Locations ordered by Z-scan
//
Build BVH
bvhResetKernel
bvhConstructKernel
bvhUpdateParent
Triangles Bounding Volume Hierarchy
Ray Data
Line Segments Start & End Point
//
Compute Ray Bounding Box
rbxKernel
//
BVH Tree Search
bvhFindCollisions
bvhTraverse
bvhInsert
Collision Candidates
bvhIntersectionKernel
//
Moller-Trumbore Algorithm
intersectMoller
Operating Mode
boolean | barycentric | intercept_count
Intersection Results
Legend:
// Kernel computation on GPU
Individual rays considered in parallel within thread-block
Surface Data
Mesh Vertices and Triangles
//
Morton code
kernel_create_morton_code
Encoded Triangle Locations
Sort
numpy.argsort
Locations ordered by Z-scan
//
Build BVH
kernel_bvh_reset
kernel_bvh_construct
bvhUpdateParent
Triangles Bounding Volume Hierarchy
Ray Data
Line Segments Start & End Point
//
Compute Ray Bounding Box
kernel_compute_ray_bounds
//
BVH Tree Search
bvhFindCollisions
bvhTraverse
bvhInsert
Collision Candidates
kernel_bvh_find_intersections
//
Moller-Trumbore Algorithm
intersectMoller
Operating Mode
boolean | barycentric | intercept_count
Intersection Results
Legend:
// Parallel computation on GPU
PURPLE Python API (kernel interface / global function)
ORANGE PyCUDA compiled device function

Figure 1: Ray-surface intersection detection work flow in the (top) CUDA and (bottom) PyCUDA implementation

- Standard usage (`mode=boolean`) — Return results as a $(N_r, 1)$ boolean array indicating whether or not each ray intersects with the surface.
- Extended usage (`mode=barycentric`) — Returns the intersecting rays with the intersecting triangle, the distance to and location of the nearest intersecting point.
- Experimental feature (`mode=intercept_count`) — Returns the number of unique intersections with the surface for each ray.

The rest of this article is organised as follows. Section 2 outlines PyCUDA's core features and installation steps. It includes a usage example of the new PyCUDA library. Section 3 focuses on implementation issues and describes the changes that made the existing CUDA code work in PyCUDA. Section 4 presents the lessons learned and describes a debugging approach which may be used to inspect C structures that reside in device memory using Python tools.

2 Preliminaries

2.1 PyCUDA: What is it and why use it?

As mentioned in the introduction, PyCUDA provides a Python integrated approach to GPU code generation at runtime. PyCUDA gives the programmer Pythonic access to Nvidia's parallel computation API [6]. It allows raw CUDA code to be embedded as a string `s` within a Python script, parsed by `pycuda.compilerSourceModule`, compiled using nvcc (the CUDA compiler driver) and linked against the CUDA runtime library. The command `compiler`

`SourceModule(s)` returns a module object `m` which exposes CUDA kernels (`__global__` functions) to the user via `m.get_function(<cuda_kernel_name>)` using the `cuModuleGetFunction` API.

> In CUDA terminology, a `__global__` function represents kernel code that is executed in an Nvidia GPU. All dependent functions called by the kernel are declared with the `__device__` specifier.

Several features may be noted based on the PyCUDA documentation

- *Resource management* is tied to lifetime of objects (akin to RAII idiom in C++). PyCUDA knows about dependencies, it won't detach from a context before all memory allocated in it is also freed.
- *Automatic error checking* — PyCUDA is strict about argument types and Boost python bindings for kernel APIs. CUDA errors are translated into Python exceptions.
- *Numpy.array like abstraction* using `pycuda.gpuarray.GPUArray` for vector operations / basic arithmetic.
- *Accessible* — The kernel APIs of interest are completely at the user's disposal.
- *Speed* — PyCUDA's base layer is written in C++, so it has negligible impact on throughput performance.

In the writer's opinion, PyCUDA can make device code easier to debug. More about this in Section 4.

2.1.1 PyCUDA versus CuPy and Numba CUDA

PyCUDA is designed for CUDA developers who want to integrate code [already] written in CUDA with Python. For machine learning developers who simply want their NumPy-based code to run on GPUs, CuPy offers an alternative. It allows users to benefit from fast GPU computation without learning CUDA syntax [7].

For people willing to embrace CUDA who prefer programming using Python constructs, there is Numba for CUDA GPUs which retains CUDA concepts, but translates a restricted subset of Python code into CUDA kernels and device functions following the CUDA execution model. In the case of PyCUDA, these kernels and device functions are written in the CUDA/C language; so being less restrictive and readily deployable in CUDA might be an advantage.

2.1.2 Working with two versions of the source at the same time

To compare the results produced by two different versions of the source, we can work with both at the same time without needing to switch between git branches, compiling and testing the code one after the other. Instead, the following workflow may be used within a Python environment.[2] This could be useful when we need assurance that incremental changes do not alter the test outcomes during development. The results may be compared using numpy functions and any differences can be easily identified. This scenario is shown in `run_example3` in `pycuda/demo.py`

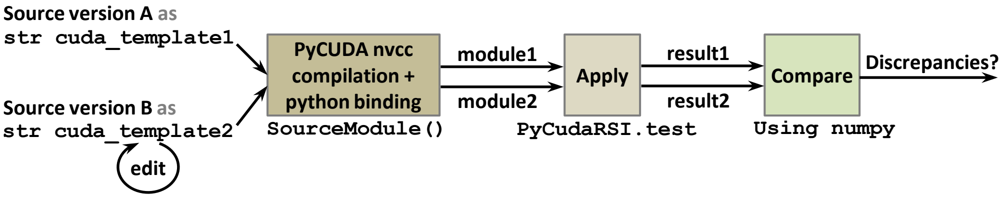


Figure 2: Comparing results from different source versions within the PyCUDA environment

In `pycuda_ray_surface_intersect.py`, the `try` clause in `perform_bindings_` and conditional statements associated with `self.kernel_intersect_*` are used jointly to manage an asymmetric situation where kernels absent from the legacy code (version A of the CUDA module) are introduced in the `development` code (version B).

2.2 PyCUDA installation

The prerequisites are listed in the official PyCUDA website. In essence, PyCUDA requires a

- CUDA-capable Nvidia GPU
- Working Python installation (in our case, version 3.8)

[2] For simplicity, the compiler cache is not shown in this figure. Refer to caching of GPU binaries (see Fig. 2) in [2].

- C++ compiler preferrably GCC and
- Access to Nvidia's CUDA toolkit

Several ENVIRONMENT variables should be set before pip install is attempted. For the author, the following commands resulted in a successful installation.

```
#!/bin/bash

$ export PATH=/usr/local/cuda-11.2/bin${PATH:+:${PATH}}
$ export LD_LIBRARY_PATH=/usr/local/cuda-11.2/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
$ export CUDA_INC_DIR=/usr/local/cuda-11.2/include
$ pip3 install pycuda
```

The Python packages in my virtual environment are shown in pycuda/requirements.txt.

2.3 PyCUDA ray-surface intersection library: Usage

The new PyCUDA library (not to be confused with the existing CUDA implementation in [1]) is used in the following manner. The valid options for `mode` are `'boolean'`, `'barycentric'` and `'intercept_count'`.

```
from pycuda_ray_surface_intersect import PyCudaRSI

design_params = dict()
cfg = {'mode': mode}

with PyCudaRSI(design_params) as pycu:
    if mode != 'barycentric':
        ray_intersects = pycu.test(vertices, triangles, raysFrom, raysTo, cfg)
    else:
        (intersecting_rays, distances, hit_triangles, hit_points) \
            = pycu.test(vertices, triangles, raysFrom, raysTo, cfg)
```

Refer to pycuda/demo.py for further comments and diagnostic features.

2.4 PyCUDA ray-surface intersection: Source Code

For reference, the PyCUDA ray-surface intersection library is available at

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

It is distributed open-source under the BSD 3-clause license.

3 PyCUDA implementation

The core functionalities are implemented in `pycuda_ray_surface_intersect.py`. Within this script, the C++/CUDA source code is retrieved from `pycuda_source.py` using `get_cuda_template()`. It returns a module string which contains nearly the same code as found in `"rsi_geometry.h"`, `"morton3D.h"` and `"bvh_structure.h"`. This separation is not technically necessary, but it makes the the class `PyCudaRSI` in `pycuda_ray_surface_intersect.py` more readable. `PyCudaRSI` (more or less) replicates the code found in `gpu_ray_surface_intersect.cu` (the existing CUDA implementation from [1]) and provides user with high-level interfaces that support the three primary use cases.

3.1 Refactoring existing CUDA code for PyCUDA

By default, PyCUDA sets `no_extern_c = False`. This has the effect of wrapping the entire source code specified in the `cuda_template` with `extern "C" {}` which prevents C++ name mangling and allows the APIs of interest to be identified. This means, if the `#include <vector>` statement from `"bvh_structure.h"` is retained [note: STL is only permitted in host code, it is never used in device code], it would trigger a linkage error like

> include/c++/6.2.1/bits/stl_iterator_base_types.h(116):
> Error: this declaration may not have extern "C" linkage.

PyCUDA also does not support overloaded functions. Although it allows templated `__device__` functions to be used, templated kernel functions (like `template <typename M> __global__ void bvhConstruct`) appear to be unsupported.[3] These resulted in some changes that differ from the original CUDA code.

3.2 Specific changes

- From "bvh_structure.h", the host function `void inline createMortonCode(const vector<T> &vertices, const vector<int> &triangles,...)` is converted into a kernel `__global__ kernelMortonCode(const float* vertices, const int* triangles,...)` .
  - This becomes accessible in Python with the binding `kernel_create_morton_code = m.get_function("kernelMortonCode")` which allows the location of mesh triangles to be transformed into 64-bit Morton codes.
- Sorting of the Morton codes, which essentially organises the triangles into local clusters to preserve spatial proximity, is no longer performed using a C++ host function. Instead, it is done in python, and the result is subsequently copied to device memory `d_sortedTriangleIDs` .
- From "morton3D.h", the `template<typename morton, typename coord>` parameters are removed. Instead, the types `MORTON, COORD` in the `cuda_template` are substituted with concrete types using a dictionary (e.g. `{"MORTON": "uint64_t", "COORD": "unsigned int"}`) before being parsed by `pycuda.compilerSourceModule(...)` .
- Finally, the existing CUDA implementation makes extensive use of custom data structures (such as axes-aligned bounding box `AABB` , `BVHNode` , `CollisionList` and `InterceptDistances`).
  For native types like `float*` , device memory allocation can be done simply using `d_vertices = cuda.mem_alloc(h_vertices.nbytes)` for instance, where `h_vertices` is a numpy.array of type `numpy.float32` and shape `(n_vertices,3)` . For C structures, a helper method `struct_size(cuda_szQuery, d_answer)` is defined to convey the bytesize of the relevant structure to Python. This must be known in order to allocate global memory on the GPU device using PyCUDA.
  - Here, `cuda_szQuery` refers to the Python interface to a getSize function such as `__global__ void bytesInBVHNode(int &x){ x = sizeof(BVHNode); }` .
  - The bytesize should not be hardcoded because memory padding may be introduced at compile time, and the extent may vary depending on the device architecture. Using the `__align__` compiler specifier (or some other variant) is encouraged to ensure CUDA memory alignment is consistent.

Despite these differences, the new code in `pycuda_ray_surface_intersect.py` is remarkably similar to the existing CUDA implementation.

3.3 Performance

Speed and correctness of the program are always front of mind. In this section, we first look at speed and identify areas where it can be improved. The following table compares the execution time for the original CUDA implementation [1] with PyCUDA. For PyCUDA, the elapsed time (see `t_start` and `t_end` in `pycuda_ray_surface_intersect.py`) also includes I/O overhead (e.g. `memcpy_dtoh` operations orchestrated by PyCUDA) and numpy processing when run in `barycentric` mode. In contrast, the original CUDA program writes results to binary files; derived quantities (e.g. intersecting points) are not computed. Overall, run times are comparable in `boolean` and `intercept_count` mode.

```
Test data generated by "diagnostic_input.py" contains
10M rays and a surface with 14874 vertices, 29260 triangles.

| Operating mode | Original CUDA code | PyCUDA implementation |
|----------------|--------------------|-----------------------|
| boolean        |     300.166 ms     |      355.094 ms       |
| barycentric    |     308.625 ms     |      909.376 ms       |
| intercept_count|     334.192 ms     |      383.117 ms       |
```

The next table provides a more detailed breakdown in `barycentric` mode.

[3] In contrast, CuPy supports type-generic kernels and C++ templated kernels where the un-mangled kernel name and template parameters are unambigously specified using `name_expressions`. Name mangling and demangling are handled under the hood.

```
| Component   Description                              | Duration     |
|----------------------------------------------------|-------------|
| 1 | Core processing*                               | 138.513 ms  |
| 2 | memcpy_dtoh (intersectTriangles, baryT)        | 231.357 ms  |
| 3 | Calculating intersecting points/distances      | 539.506 ms  |
|------------------------------------------------------------------|
| 3a| Identify intersecting rays (numpy)             |  34.562 ms  |
| 3b| Calculate intersecting distances (host)        | 240.836 ms  |
| 3c| Retain intersecting triangles    (host)        |   9.246 ms  |
| 3d| Calculate intersecting points    (host)        | 254.862 ms  |

* From t_start to completion of kernel_bvh_find_intersections2 .
  This includes: configure_, allocate_memory_, transfer_data_,
  get_min_max_extent_of_surface, kernel_compute_ray_bounds,
  kernel_create_morton_code, kernel_bvh_reset, kernel_bvh_construct.
```

Based on Occam's principle, the host code for components 3b and 3d were converted to kernel code to increase efficiency. These kernel functions (`kernelIntersectDistances` and `kernelIntersectPoints`) were added to `pycuda_source.py` . With minimal effort, significant improvement can be seen from the run time figures below.

```
| Component   Description                              | Duration     |
|----------------------------------------------------|-------------|
| 3b| Calculate intersecting distances (gpu)         |  18.039 ms  |
| 3d| Calculate intersecting points    (gpu)         |  24.629 ms  |
|------------------------------------------------------------------|
| 3 | Calculating intersecting points/distances      |  86.476 ms  | (84% reduction)
| * | Total duration for barycentric                 | 456.346 ms  | (50% reduction)
```

Next, we will take a deep-dive and discuss debugging and verification efforts.

4 Lessons learned

A significant advantage is that PyCUDA can make device code easier to debug. This in part is due to the ease with which device memory can be copied to the host for inspection as numpy.arrays. However, there are also pitfalls that developers need to be mindful of. In the ensuing discussion, we examine two situations: one concerns abnormal test results, another resulted in the program crashing due to grid partitioning (carelessness in the binary radix tree construction). Both issues had their origins in memory use, which may be diagnosed by examining contents in the triangle-mesh bounding volume hierarchy.

4.1 Case Study 1: Debugging C struct from device memory. Meaning of 'int' in CUDA and numpy

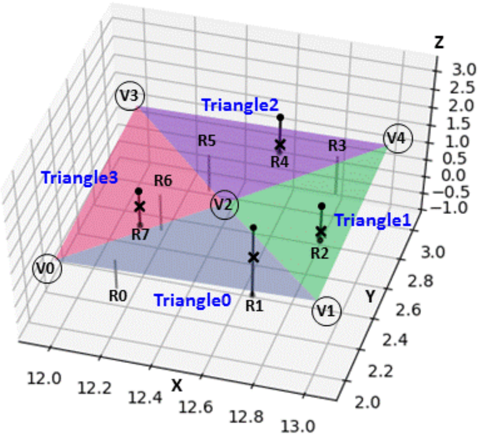


Figure 3: A simple test configuration with 8 rays and 4 triangles

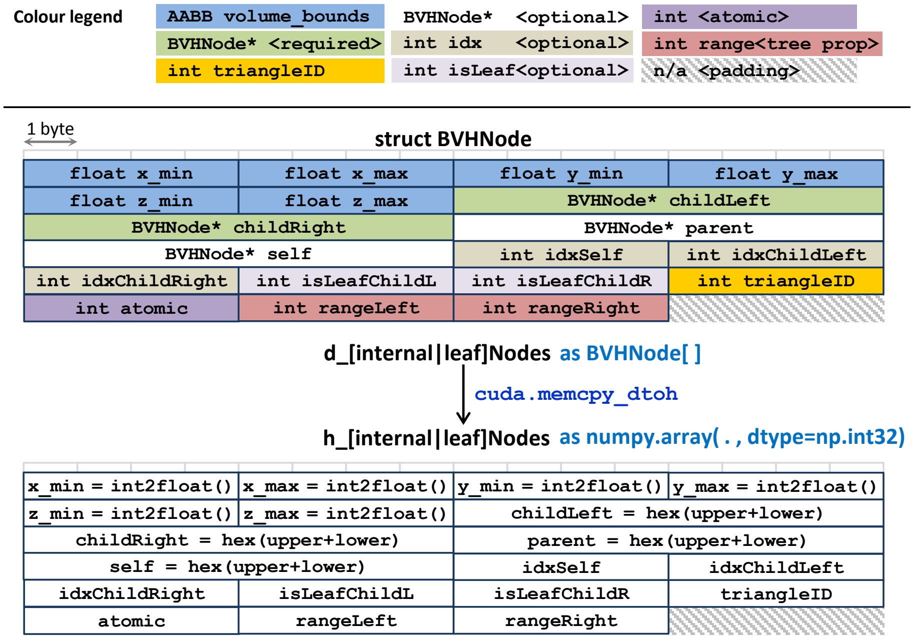
This picture depicts a test scenario where the mesh surface consists of 4 triangles and there are 8 rays (line segments). In particular, R_1, R_2, R_4 and R_7 intersect triangle T_0, T_1, T_2 and T_3 at [12.7,2.2,1.14], [12.9,2.4,1.21], [12.6,2.9,1.23] and [12.2,2.4,1.08], respectively. However, due to a bug, the initial PyCUDA implementation reported only R_1 and R_7 as the intersecting rays. Through a process of elimination, the cause was narrowed down to BVH tree construction and interpretation of the triangle IDs after the corresponding Morton codes are sorted.

To diagnose the problem, the contents of the triangle bounding volume hierarchy (BVH) were examined. The aim is to verify the integrity of the BVH and find the root cause. This involves two steps as listed below.

4.1.1 Bounding Volume Hierarchy (BVH) Debugging Strategy

1. Transfer the relevant content from device memory to host memory using standard PyCUDA I/O interfaces.
2. Decode the fields within each BVH node based on the type definitions given in `struct BVHNode` .

In our current circumstances, the goal is to flush out all information and examine some properties (e.g. connectivity) within a graph. So, this strategy offers a suitable alternative to invoking `cuda-gdb` as many of its rich features are not needed. It is certainly not intended as a replacement of `cuda-gdb` in the general sense.



float x_min	float x_max	float y_min	float y_max
float z_min	float z_max	BVHNode* childLeft	
BVHNode* childRight		BVHNode* parent	
BVHNode* self		int idxSelf	int idxChildLeft
int idxChildRight	int isLeafChildL	int isLeafChildR	int triangleID
int atomic	int rangeLeft	int rangeRight	▨

x_min = int2float()	x_max = int2float()	y_min = int2float()	y_max = int2float()
z_min = int2float()	z_max = int2float()	childLeft = hex(upper+lower)	
childRight = hex(upper+lower)		parent = hex(upper+lower)	
self = hex(upper+lower)		idxSelf	idxChildLeft
idxChildRight	isLeafChildL	isLeafChildR	triangleID
atomic	rangeLeft	rangeRight	▨

Figure 4: A bounding volume hierarchy consists of N_t internal nodes (the last one only holds a pointer to the root node) and N_t leaf nodes. Each memory array is copied from GPU device to host (using int32 representation). The contents within individual BVHNodes are subsequently decoded according to the data type of each field.

Specifically, two numpy arrays are created to hold the content of the internal and leaf nodes. The `cuda.memset_dtoh` operation converts the bitstream into `int32` values.

```
h_leafNodes = np.zeros(n_triangles * sz_BVHNode, dtype=np.int32)
h_internalNodes = np.zeros(n_triangles * sz_BVHNode, dtype=np.int32)
cuda.memcpy_dtoh(h_leafNodes, d_leafNodes)
cuda.memcpy_dtoh(h_internalNodes, d_internalNodes)
```

To manipulate the word-stream (where ‘word’ = 4 contiguous bytes), the memory footprint of each `BVHNode` is determined using the helpful `struct_size` macro [defined in the `pycuda_ray_surface_intersect.py` module] and divided by the number of bytes in `numpy.int32` to calculate the number of `int32` elements per `BVHNode` .

```
sz_BVHNode = int(struct_size(bytes_in_BVHNode, d_szQuery)
             / np.ones(1, dtype=np.int32).nbytes)
```

Content within each `BVHNode` (a chunk of `h_*Nodes`) is processed one node at a time using the `display_node_contents` method in `pycuda/diagnostic_utils.py` which interprets the field values according to the definition of `struct`

`BVHNode`. For debugging purpose, `params['USE_EXTRA_BVH_FIELDS']` is set to `True` to (i) indicate whether the Left/Right descendants (`childL` and `childR` nodes) are "internal" or "leaf" (terminal) nodes; and (ii) include the memory address of the parent and current node itself, to make the binary radix tree structure easier to trace.

```
for t in range(n_triangles):
    words = h_internalNodes[t*sz_BVHNode:(t+1)*sz_BVHNode]
    display_node_contents(words, t, params['USE_EXTRA_BVH_FIELDS'])

for t in range(n_triangles):
    words = h_leafNodes[t*sz_BVHNode:(t+1)*sz_BVHNode]
    display_node_contents(words, t, params['USE_EXTRA_BVH_FIELDS'])
```

Distilling the critical information, the code above reveals that the first triangle T_0 was repeated in the leaf nodes. Furthermore, aside from T_3, the triangle IDs for T_1 and T_2 were also missing.

```
Leaf nodes
[0] x:[12.0,13.0], y:[2.0,2.5], z:[1.0,1.2], triangleID: 0, rangeL: 0, rangeR: 0
[1] x:[12.0,13.0], y:[2.0,2.5], z:[1.0,1.2], triangleID: 0, rangeL: 1, rangeR: 1
[2] x:[12.0,12.5], y:[2.0,3.0], z:[1.0,1.2], triangleID: 3, rangeL: 2, rangeR: 2
[3] x:[12.0,13.0], y:[2.0,2.5], z:[1.0,1.2], triangleID: 0, rangeL: 3, rangeR: 3
```

4.1.2 The culprit

In `kernelBVHReset`, the argument `sortedObjectIDs` is of type `int*`. In CUDA, this is treated as an `int32[]` array.

```
__global__ void kernelBVHReset(const float* __restrict__ vertices,
                               const int* __restrict__ triangles,
                               BVHNode* __restrict__ internalNodes,
                               BVHNode* __restrict__ leafNodes,
                               int* __restrict__ sortedObjectIDs, int nNodes)
{
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= nNodes)
        return;
    //set triangle attributes in leaf
    leafNodes[i].triangleID = t = sortedObjectIDs[i];
    :
}
```

This is incompatible with

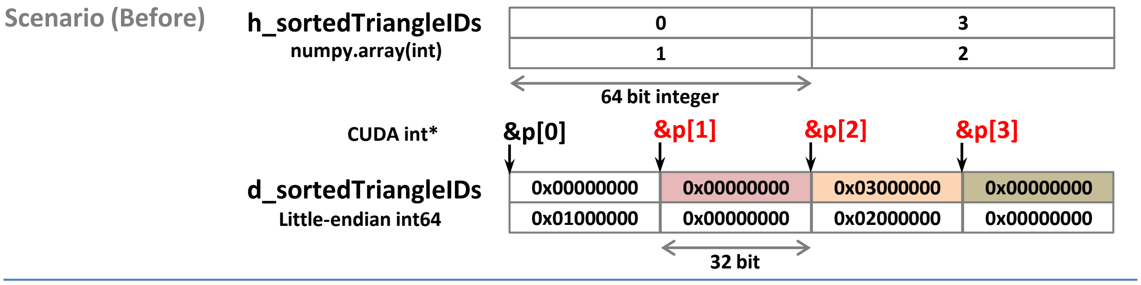
```
d_sortedTriangleIDs = cuda.mem_alloc(n_triangles * np.ones(1, dtype=np.int64).nbytes) #(**)
```

which is passed as the `sortedObjectIDs` argument. Its dtype follows the convention of

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

which under Python3, the returned array type `int` is defaulted to `int64` on 64-bit machines.

As a result, using `p[i]` as a short-hand for `d_sortedObjectIDs[i]`, the figure below shows that the CUDA int32 pointer is incremented at half the required rate (advancing by 32-bit each time). For the third "read", `p[2]` is fetching the least-significant word of the second 64-bit integer which happens to reference triangle T_3.



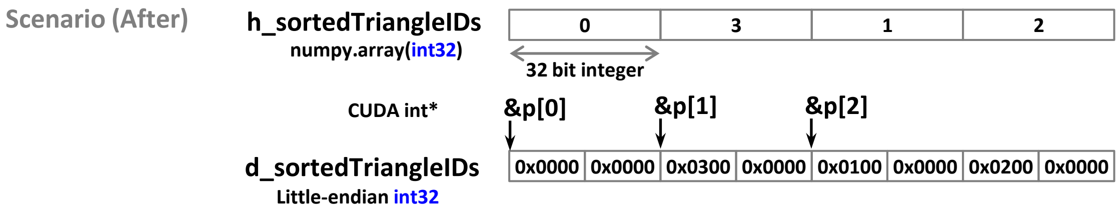


Figure 5: The incongruous nature of memory access when CUDA int* reads off 32-bit integers from a 64-bit array. This situation arised because memory for the device array was allocated based on the size of the host array returned by numpy.argsort which interprets int as int64. Meanwhile in CUDA, int is treated as int32 by the compiler .

4.1.3 Resolution

To fix this, the stride (for a 32-bit integer pointer in CUDA) needs to be consistent with the width of an integer element in device memory. This may be enforced by replacing the lines (**) above with

```
h_sortedTriangleIDs = np.argsort(h_morton).astype(np.int32)
d_sortedTriangleIDs = cuda.mem_alloc(n_triangles * np.ones(1,dtype=np.int32).nbytes)
```

The final PyCUDA implementation contains these changes and produces debugging output where each of the LEAF nodes corresponds to a different triangle.

```
BVH tree structure
---------------------------
Internal nodes
[0] x:[12.0,13.0], y:[2.0,3.0], z:[1.0,1.2]  <<< Bounding Volume
self: 0x66e1200, parent: 0x66e1260
indices: 0(self), 0(L-leaf), 1(R-leaf)
childL: 0x66e1000, childR: 0x66e1060
atomic: 2, rangeL: 0, rangeR: 1

[1] x:[12.0,13.0], y:[2.0,3.0], z:[1.0,1.3]  ------ ROOT NODE
self: 0x66e1260, parent:
indices: 1(self), 0(L-internal), 2(R-internal)
childL: 0x66e1200, childR: 0x66e12c0
atomic: 2, rangeL: 0, rangeR: 3

[2] x:[12.0,13.0], y:[2.0,3.0], z:[1.1,1.3]
self: 0x66e12c0, parent: 0x66e1260
indices: 2(self), 2(L-leaf), 3(R-leaf)
childL: 0x66e10c0, childR: 0x66e1120
atomic: 2, rangeL: 2, rangeR: 3

[3] x:[0.0,0.0], y:[0.0,0.0], z:[0.0,0.0]
self: 0x0, parent: 0x0
indices: 3(self), 1(L-internal), 0(R-internal)
childL: 0x66e1260(root node), childR: 0x0
atomic: 0, rangeL: 0, rangeR: -1

---------------------------
Leaf nodes
[0] x:[12.0,13.0], y:[2.0,2.5], z:[1.0,1.2]
self: 0x66e1000, parent: 0x66e1200
triangleID: 0

[1] x:[12.0,12.5], y:[2.0,3.0], z:[1.0,1.2]
self: 0x66e1060, parent: 0x66e1200
triangleID: 3

[2] x:[12.5,13.0], y:[2.0,3.0], z:[1.1,1.3]
self: 0x66e10c0, parent: 0x66e12c0
triangleID: 1
```

```
[3] x:[12.0,13.0], y:[2.5,3.0], z:[1.1,1.3]
self: 0x66e1120, parent: 0x66e12c0
triangleID: 2
```

Furthermore, the program correctly reports R_1, R_2, R_4 and R_7 as the intersecting rays under 'boolean' mode.

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

4.1.4 Test code and Graphviz

These results may be replicated by running `pycuda/demo.py`. Setting `cfg['bvh_visualisation'] = ['graph']` in `run_example1`, it will also generate a graph for the mesh-triangle binary radix tree using graphviz.

- The relevant code `bvh_graphviz` is found in `pycuda/diagnostic_graphics.py`.

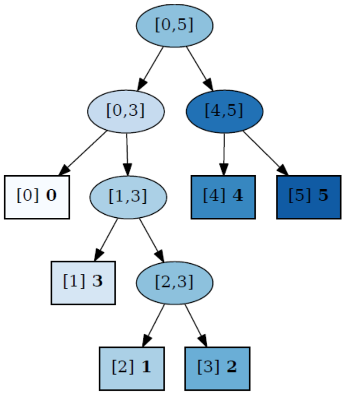


Figure 6: A graph for the binary radix tree generated using Graphviz

The graph above corresponds to the test surface used in `intercept_count` mode, where two patches have been added above triangles 1 and 2 in the simple test surface to form a "canopy". For the root node and internal nodes, `[a,b]` represents the node index range. For terminal nodes (squares), the `c` in `[c] d` represents the leaf node index and `d` represents the triangle ID (taking into account reordering using the Morton codes).

- This setting also writes a graph description in the DOT language to a file (`bvh_structure.gv`).

4.2 Case Study 2: Not paying enough attention to grid size. Illegal memory access when BVH structure is not properly constructed

A large surface with ∼30,000 triangles and 10 million rays were created using `synthesize_data` from `pycuda/diagnostic_input.py`. Using similar construction, the `PyCudaRSI.test` API was called in `boolean` mode. The following error messages were produced.

```
Traceback (most recent call last):
  File "/opt/python3.8/lib/python3.8/runpy.py", line 193, in _run_module_as_main
    return _run_code(code, main_globals, None,
  File "/opt/python3.8/lib/python3.8/runpy.py", line 86, in _run_code
    exec(code, run_globals)
  File "<git_repo>/pycuda/demo.py", line 182, in <module>
    run_example2(mode)
  File "<git_repo>/pycuda/demo.py", line 140, in run_example2
    ray_intersects = pycu.test(vertices, triangles, raysFrom, raysTo, cfg)
  File "<git_repo>/pycuda/pycuda_ray_surface_intersect.py", line 313, in test
    cuda.memcpy_dtoh(self.h_crossingDetected, self.d_crossingDetected)
pycuda._driver.LogicError: cuMemcpyDtoH failed:
                           an illegal memory access was encountered
PyCUDA WARNING: a clean-up operation failed (dead context maybe?)
cuModuleUnload failed: an illegal memory access was encountered
```

4.2.1 Investigation

To investigate this issue, we examined the tree content once again following the BVH debugging strategy. A print-out is obtained by setting `cfg['examine_bvh'] = True` temporarily.

In this case, the BVH is much more complex. A truncated portion of the graph is shown in Fig. 7.

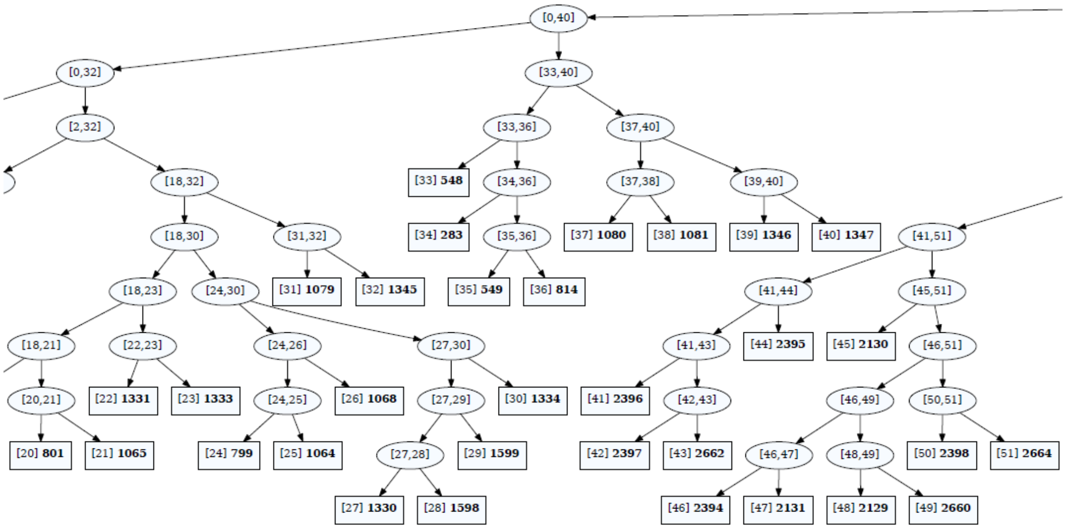


Figure 7: A truncated portion of the graph for the triangle-mesh binary radix tree used in case 2

The next picture (Fig. 8) shows what the bounding volume hierarchy looks like as we descend from the top of the tree. The root node is considered level 0, its immediate descendants constitute level -1 and so forth.

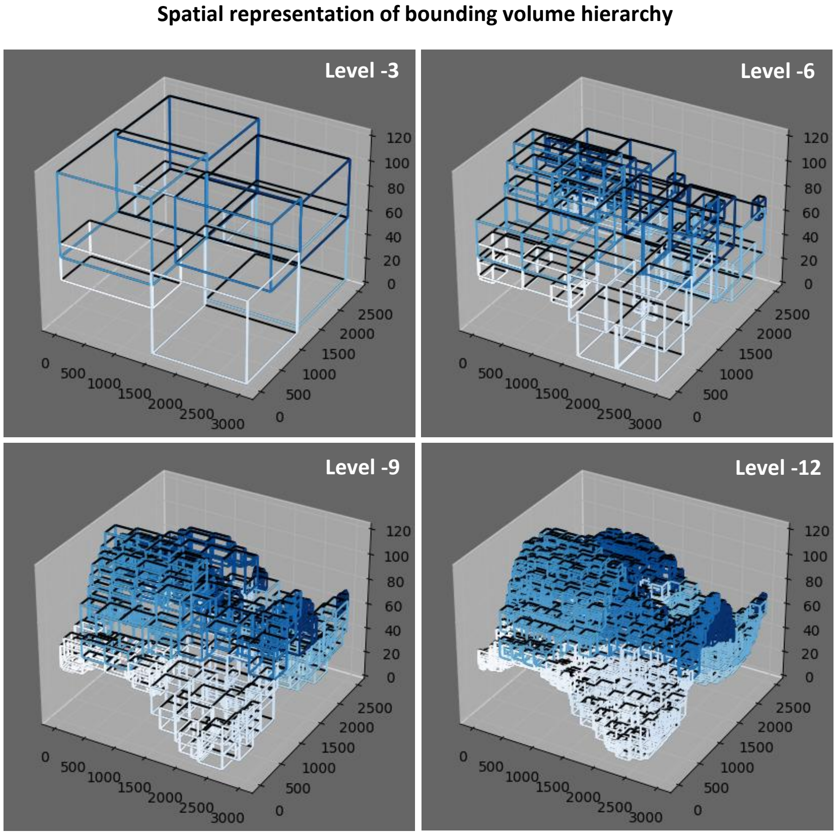


Figure 8: A spatial representation of the bounding volume hierarchy in case 2

4.2.2 Observations

The critical findings are included below.

```
BVH tree structure
---------------------------
Internal nodes
   :
[16382] x:[992.584,1018.47], y:[363.936,387.745], z:[61.0106,62.1155]
self: 0xe2361ff40, parent: 0xe2361ffa0
indices: 16382(self), 16382(L-leaf), 16383(R-leaf)
childL: 0xe2335ff40, childR: 0xe2335ffa0
atomic: 2, rangeL: 16382, rangeR: 16383

[16383] x:[0,0], y:[0,0], z:[0,0]
self: 0x0, parent: 0x0
indices: 16383(self), 16382(L-internal), 0(R-internal)
childL: 0xe2361ff40, childR: 0x0
atomic: 1, rangeL: 16382, rangeR: -1

[16384] x:[0,0], y:[0,0], z:[0,0]
self: 0x0, parent: 0x0
indices: 16384(self), 0(L-internal), 0(R-internal)
childL: 0x0, childR: 0x0
atomic: 0
   :
[29259] x:[0,0], y:[0,0], z:[0,0]
self: 0x0, parent: 0x0
indices: 29259(self), 0(L-internal), 0(R-internal)
childL: 0x0, childR: 0x0
atomic: 0
---------------------------
```

For internal nodes,

- There are nodes such as 16382 where all attributes are defined.
- However, there are nodes such as 16383 which are "half-filled". Information has propagated up from its left child, however it is not yet connected to its right child. Hence, the atomic counter has value 1.
- In fact, there are nodes such as 16384 which are yet to be populated in any way. For a binary radix tree, `n_triangles - 1` internal nodes should always be used.
- Finally, the last internal node (with index `29259 = n_triangles - 1`) should contain the address of the root node in `childL`. This can only happen if bottom-up construction reaches the top.
- Having a NULL pointer for the root node would prevent the BVH from being traversed.

```
Leaf nodes
   :
[16383] x:[992.584,1018.47], y:[363.936,387.745], z:[61.2898,62.1155]
self: 0xe2335ffa0, parent: 0xe2361ff40
triangleID: 4345

[16384] x:[968.359,992.584], y:[341.158,363.936], z:[61.0106,63.3395]
self: 0x0, parent: 0x0
triangleID: 4077
   :
```

For leaf nodes,

- There are nodes such as 16384 that are not connected with its parent.
- This means `__device__ void bvhUpdateParent(BVHNode* node, BVHNode* internalNodes, BVHNode* leafNodes, MORTON* morton, int nNodes)` has not been performed on this leaf node, even though it has been initialised with a triangleID by `kernelBVHReset`.

4.2.3 The culprit

As `bvhUpdateParent` is only recursively called by itself or `kernelBVHConstruct`, the evidence points toward a bug with the invocation of `self.kernel_bvh_construct`. Inspection of `pycuda_ray_surface_intersect.py` reveals the grid argument (number of thread-blocks) was incorrectly configured for this kernel.

```
self.kernel_bvh_construct(
            self.d_internalNodes, self.d_leafNodes, self.d_morton,
            np.int32(self.n_triangles), block=self.block_dims, grid=self.grid_lambda)
```

It should read

```
self.kernel_bvh_construct(
            self.d_internalNodes, self.d_leafNodes, self.d_morton,
            np.int32(self.n_triangles), block=self.block_dims, grid=self.grid_dimsT)
```

based on the definitions

```
self.grid_lambda = (self.grid_xLambda, 1)
self.grid_dimsT = (int(np.ceil(self.n_triangles / self.block_x)), 1)
```

When the program is executed, the console also shows

```
CUDA partitions: 1024 threads/block,
grids: [rays: 9766, bvh_construct: 29, bvh_intersect: 16]
```

That is, `grid_lambda = (16, 1)` and `grid_dimsT = (29, 1)`. As `kernel_bvh_construct` relates to triangle-mesh tree construction, the partitions should depend on `grid_dimsT` (equivalently, `n_triangles`). The number of partitions should not depend on `grid_lambda` which corresponds to the N_{grids} parameter which relates to `__device__ void bvhFindCollisions` and `__global__ void kernelBVHIntersection`. This was a copy-and-paste error.

4.2.4 Resolution

For this test case, `grid_lambda` < `grid_dimsT`. This explains why some leaf nodes were untouched and the associated data (volume bounds and indices) did not propagate to the top of the tree. It was doing less work than it was supposed to. The final PyCUDA implementation contains the bug-fix indicated in blue above.

5 Summary

These examples demonstrate that PyCUDA can make kernel code easier to debug. It is possible to transfer data from device memory to host with ease and inspect objects using PyCUDA abstraction and standard Python tools. For more complex C structures such as the bounding volume hierarchy, a decoding layer/method is required to interpret mixed-type content in individual `BHVNodes` that constitute the internal and leaf node arrays. With such power also comes responsibility. There are certain pitfalls that developers ought to be mindful of. In the first case study, one is reminded of the different interpretations of `int` — the Nvidia CUDA compiler (nvcc) treats int as int32 whereas numpy uses int64. This discrepancy creates a mismatch situation (see Fig. 5) where the int32* pointer lags behind the actual int64 data. This resulted in the "disappearance" of mesh triangles from the binary radix tree and misdetection of certain ray-surface intersections in the test scenario. This problem could have manifested in other ways and led to other strange behaviour in a different context. In the second case study, we encountered an exception. The same debugging strategy was employed and we managed to find the root cause of the exception. We found that an incorrect specification of the grid size contributed to an incomplete construction of the BVH. Inspection of the BVH structure clearly revealed some leaf node properties (volumetric bounds) had not propagated to the top of the tree. The final PyCUDA ray-surface intersection library is free of these issues. The source code is available at `https://github.com/raymondleung8/gpu-ray-surface-intersection-in-cuda/tree/main/pycuda` under a BSD-3 clause license.

Epilogue

Currently, the entire implementation uses 32-bit float. Inside the `intersectMoller` function, rounding errors associated with the dot and cross-product calculations can accumulate. For edge cases where the ray runs parallel or almost parallel to the face of a triangle, the loss of precision can sometimes make a difference. In our experience, this is relatively uncommon—the incidence rate is about 10^{-6} [this depends on the number of intersection tests that are nearly degenerate]. If this is an issue, `PyCudaRSI.__init__(params)` may be initialised with `params['USE_DOUBLE_PRECISION_MOLLER']` set to `True`. This will activate `COMPILE_DOUBLE_PRECISION_MOLLER` in `pycuda_source.py` and store intermediate quantities within `__device__ int intersectMoller` using 64-bit floats.

Further reading

GPU programming and CUDA concepts are covered by Hwu et al. in [8]. This book is a valuable resource for beginners. Scripting-based approach to GPU run-time code generation (particularly PyCUDA) is discussed in [2, 3]. Bounding volume hierarchies and linear binary radix tree construction are described by Apetrei in [9]. Ray-triangle intersection tests are surveyed and evaluated by Jiménez et al. in [10, 11].

Acknowledgements

This work has been supported by the Rio Tinto Centre for Mine Automation and Australian Centre for Field Robotics.

References

[1] Raymond Leung. GPU implementation of a ray-surface intersection algorithm in CUDA. Source code available at `https://github.com/raymondleung8/gpu-ray-surface-intersection-in-cuda` under the BSD 3-clause license. Documentation available at: `https://arxiv.org/abs/2209.02878`. *arXiv e-print*, pages 1–11, 2022.

[2] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda: GPU run-time code generation for high-performance computing. Available at `http://arxiv.org/abs/0911.3456`. *arXiv e-print 0911.3456*, 2009.

[3] Andreas Klöckner, Nicolas Pinto, Bryan Catanzaro, Yunsup Lee, Paul Ivanov, and Ahmed Fasih. GPU scripting and code generation with PyCUDA. Available at `http://arxiv.org/abs/1304.5553`. In Wen-mei Hwu, editor, *GPU Computing Gems, Chapter 27*. Elsevier, 2011.

[4] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[5] Tero Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37, 2012.

[6] Andreas Klöckner. PyCUDA documentation. Available at: https://documen.tician.de/pycuda/, 2022.

[7] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. CuPy: A NumPy-compatible library for NVIDIA GPU calculations. *31st Confernce on Neural Information Processing Systems*, 151(7), 2017.

[8] Wen-Mei Hwu, David Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.

[9] Ciprian Apetrei. Fast and simple agglomerative LBVH construction. In Rita Borgo and Wen Tang, editors, *EG UK Computer Graphics & Visual Computing*. The Eurographics Association. Available at: `http://diglib.eg.org/bitstream/handle/10.2312/cgvc.20141206.041-044/041-044.pdf?sequence=1&isAllowed=y`, 2014.

[10] Juan J Jiménez, Rafael J Segura, and Francisco R Feito. A robust segment/triangle intersection algorithm for interference tests. Efficiency study. *Computational Geometry*, 43(5):474–492, 2010.

[11] Juan J Jiménez, Carlos J Ogáyar, José M Noguera, and Félix Paulano. Performance analysis for GPU-based ray-triangle algorithms. In *2014 International Conference on Computer Graphics Theory and Applications (GRAPP)*, pages 1–8. IEEE, 2014.