High-Performance Graphics (2019)
T. Foley and M. Steinberger (Editors)

Short Paper

RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location

Ingo Wald[†] Will Usher[‡] Nate Morrical[‡] Laura Lediaev[‡] Valerio Pascucci[‡]
[†]NVIDIA [‡]SCI Institute, University of Utah

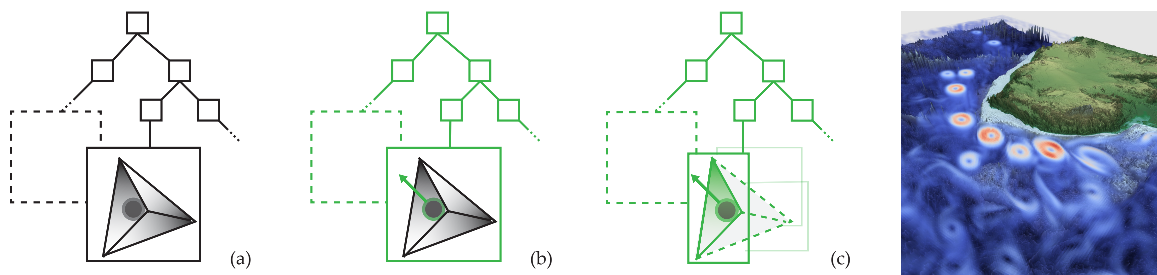


Figure 1: *a-c) Illustrations of the tetrahedral mesh point location kernels evaluated in this paper. a) Our* reference *method builds a BVH over the tets and performs both BVH traversal and point-in-tet tests in software (black) using CUDA. b)* `rtx-bvh` *uses an RTX-accelerated BVH over tets and triggers hardware BVH traversal (green) by tracing infinitesimal rays at the sample points, while still performing point-tet tests in software (black). c)* `rtx-rep-faces` *and* `rtx-shrd-faces` *use both hardware BVH traversal and triangle intersection (green) by tracing rays against the tetrahedras' faces. d) An image from the unstructured-data volume ray marcher used to evaluate our point location kernels, showing the 35.7M tet* Agulhas Current *data set rendered interactively on an NVIDIA TITAN RTX (34 FPS at* 1024^2 *pixels).*

Abstract

We explore a first proof-of-concept example of creatively using the Turing generation's hardware ray tracing cores to solve a problem other than classical ray tracing, specifically, point location in unstructured tetrahedral meshes. Starting with a CUDA reference method, we describe and evaluate three different approaches to reformulate this problem in a manner that allows it to be mapped to these new hardware units. Each variant replaces the simpler problem of point queries with the more complex one of ray queries; however, thanks to hardware acceleration, these approaches are actually faster than the reference method.

1. Introduction

GPUs started out as relatively simple devices which only accelerated certain parts of the triangle rasterization pipeline, but have since evolved into massively parallel processors with a wide range of applications. The primary driver behind these rapid architectural advancements was—and still is today—graphics, and in particular gaming. However, the raw computational capability available on GPUs has always attracted researchers to look at creative ways of harnessing these capabilities outside of graphics. In the early days of (GP)GPU compute this was done by re-formulating the problem at hand into a rendering problem, in a way that it could make use of the respective GPU generation's most powerful units. On prior GPU generations, work has leveraged the texture units, register combiners, early shader cores, etc. (see, e.g., [LM01, RS01b, RS01a]).

Research in this area was very active in the early 2000's, but calmed down once GPUs became freely programmable through high-level languages such as CUDA and OpenCL. When most of a GPU's computational power is available through programmable shader cores, CUDA and OpenCL provided the easiest way to leverage the hardware's capabilities to solve general problems.

Nevertheless, GPUs still contain a significant amount of dedicated hardware resources which offer the potential to accelerate workloads beyond general purpose CUDA compute. Specifically, NVidia's latest GPU architecture, Turing [NVI18b], adds two new hardware units which fit this description: Tensor Cores for AI and machine learning; and ray tracing cores for bounding volume hierarchy (BVH) traversal and ray-triangle intersection. A first proof of concept using the Tensor Cores to accelerate general computation was recently shown by Haidar et al. [HTDH18].

In this paper, we investigate and evaluate using Turing's other new hardware unit—the ray tracing cores—for an application beyond rendering. Specifically, we consider how these can be used to accelerate the task of point location in tetrahedral meshes (i.e., given a point p and a tetrahedral mesh m, find the tetrahedron t containing p). We evaluate four different strategies with varying degrees of RTX hardware acceleration, and show that it is possible to effectively use the ray tracing cores for this problem. Though this problem has applica-

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

DOI: 10.2312/hpg.20191189

tions well beyond interactive volume visualization, we demonstrate a first example use case of our RTX-accelerated point location kernels by using them to accelerate a prototype sample-based volume ray marcher for unstructured meshes.

2. RTX Beyond Ray Tracing

The overarching goal of this paper is to investigate and evaluate different ways of leveraging the RTX hardware to accelerate tetrahedral mesh point location kernels. In the following sections, we will describe four different kernels (also see Figure 1), starting with a reference CUDA kernel which traverses a BVH without any RTX acceleration at all (Section 3); followed by a kernel that uses the RTX hardware for BVH traversal (Section 4); and two different kernels (Section 5) that exploit both hardware BVH traversal and hardware ray-triangle intersection. In Section 6 we evaluate these kernels on artificial benchmarks and a proof-of-concept volume ray marcher for unstructured meshes.

All four kernels use the same interface: Given a tetrahedral mesh and an arbitrary 3D point, determine which tetrahedron that point is in and return it to the user. For each kernel we look at two variants: one that returns just the ID of the tet containing the point, and one that returns a scalar field value for the query point (either by interpolating a per-vertex scalar, or looking up a per-cell scalar, as provided by the data set). If the point is not contained in a tet, the kernels return -1 or $-\infty$, respectively.

Similarly, the input to each kernel is the same. The tetrahedral mesh consists of an array of `float3` vertices and an array of `int4` tetrahedra indices. For the scalar field kernel, an additional `float` array of per-vertex or per-cell scalars is also provided. Extension to non-tetrahedral primitives should be possible, but for the sake of brevity will not be investigated in this paper.

We implement our kernels within OptiX [PBD*10], which added support for Turing's RTX capabilities in version 6. We do assume basic familiarity with both OptiX and RTX, and refer the user to the latest OptiX Programming Guide [NVI18a] and the Turing whitepaper [NVI18b] for reference. Other than portability, one advantage of OptiX is that it uses CUDA under the hood, which allows us to evaluate our CUDA-only reference method within the same framework as our RTX optimized methods. We also make use of OptiX's template support to guarantee code consistency (i.e., point-in-tet testing, the volume renderer's ray marching, integration, and transfer function lookup code, etc.) across our kernels.

For the point-in-tet test we use the 3D version of Cramer's method to compute the four barycentric coordinates of p, and test if they are all nonnegative. If all are positive the four values can then, if desired, be used for interpolating the per-vertex scalar values.

3. Non-RTX Reference: `cuda-bvh`

To provide a non-RTX reference method we first implemented a software-based tet-mesh point query in CUDA. Our implementation is similar to how such queries are done in OSPRay [WJA*17], using the method described by Rathke et al. [RWCB15]. Similar to Rathke et al., we build a BVH over the tetrahedra; however, instead of their uncompressed binary BVH, we use a four-wide BVH with quantized child node bounds, similar to Embree's QBVH8 layout [WWB*14]. We note that this choice of BVH was not motivated by any expected performance gain or memory use optimization, but simply because an easy-to-integrate library for this BVH was readily available. The BVH is built on the host using this library, after which it is uploaded to the GPU by loading it into an OptiX buffer.

Though we use this reference method within our OptiX framework, the kernel itself does not use any OptiX constructs whatsoever, and could be used from arbitrary CUDA programs. To find the tetrahedron containing the query point the kernel performs a depth first traversal using a software managed stack of BVH node references, immediately returning the tet once it is found. Our implementation is similar to the following pseudocode:

```
int pointLoc_reference(vec3f P)
  stack = { rootNode };
  while (!stack.empty())
     nodeRef = stack.pop();
     if (nodeRef is leaf)
        if (pointInTet(P, (tetID=nodeRef.getChild())
           return tetID;
     else foreach child : 0..4
        if (pointInBox(P, nodeRef.getBounds(child))
           stack.push(nodeRef.getChildRef(child));
  return -1; /* no containing tet */
```

4. `rtx-bvh`: Exploiting RTX for BVH Traversal

While the reference method is reasonably efficient, it does not use the RTX hardware at all. To do so, we first have to re-formulate our problem in a way that it fits the hardware; i.e., we have to express point location as a ray tracing problem.

Staying conceptually close to our reference implementation, we can use OptiX to build an RTX BVH over the tetrahedra by creating an `optix::Geometry` with the given number of tets and a *bounding box program* that returns the respective tet's bounding box. Once we put this geometry into an `optix::GeometryInstance` (GI) and attach an `optix::Acceleration` we know that OptiX will build an RTX acceleration structure over the tets.

Although now armed with a hardware accelerated BVH, one problem remains: the hardware only knows about tracing rays, not points! Thus, we must find a way to express our query points as "rays". Fortunately, we can simply view each query point as an infinitesimally short ray, and use an arbitrary direction (e.g., $(1,1,1)$) and vanishingly small ray interval ($ray.tmax = 1e-10f$) to express this to OptiX. When we call `rtTrace` on such a "ray" the hardware will traverse the BVH and must visit the tetrahedra potentially overlapping the ray to find an intersection. To find the tetrahedra containing the query point, we attach an *intersection program* to our geometry which executes our point-in-tet test and, when the containing tet is found, stores the result in the per-ray data (Figure 2).

As the rays traced are vanishingly short we can expect the traversal to visit roughly the same BVH nodes as our reference implementation, though there is no guarantee that the hardware will visit *only* those nodes overlapping the point. Once the containing tet is found we tell OptiX to report the hit, allowing the hardware to immediately terminate BVH traversal regardless of what else might be on the traversal stack, as done in the reference implementation. For

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

```
rtDeclareVariable(Ray, ray, rtCurrentRay, );
rtDeclareVariable(float, prd, rtPayload, );
rtDeclareVariable(rtObject, world, , );
RT_PROGRAM void bounds(float *bounds, int tetID)
{ *bounds = box3f(vertex[index[tetID].x],...); }
RT_PROGRAM void intersect(int tetID) {
  if (intersectTet(ray.origin,tetID,result) &&
      rtPotentialIntersection(1e-10f)) {
    prd = result;
    rtReportIntersection(0);
}}
__device__ float getSample(const vec3f P) {
  Ray ray(P, vec3f(1), 0, 0.f, 2e-10f);
  float prd_result = negInf;
  rtTrace(world, ray, prd_result,
     RT_VISIBILITY_ALL,
     RT_RAY_FLAG_TERMINATE_ON_FIRST_HIT
     |RT_RAY_FLAG_DISABLE_ANYHIT
     |RT_RAY_FLAG_DISABLE_CLOSESTHIT);
  return prd_result;
}
```

Figure 2: *Pseudocode for our `rtx-bvh` method, which performs the point query by launching an infinitesimal ray from the point and performs the point-in-tet tests in the intersection program.*

performance reasons, we explicitly disable the anyhit and closest hit programs to save the overhead of calling empty functions.

The `rtx-bvh` kernel leaves the actual BVH construction and traversal to OptiX, which will automatically use hardware accelerated BVH traversal if available, and fall back to its own software traversal if not. Compared to the CUDA reference method, `rtx-bvh` leverages the RTX hardware to accelerate BVH traversal, though it still performs the point-in-tet tests in software (Figure 1b). Although traversing a ray is *more* expensive than traversing a point, the ray traversal is now hardware accelerated, and we can expect to observe a performance gain over the reference method.

5. Full Hardware Acceleration with RTX Triangles

Though the `rtx-bvh` method uses hardware accelerated BVH traversal, it still relies on a software point-in-tet test, limiting the potential speedup it can achieve. To improve performance further, we must reduce these tests and eliminate the back-and-forth between the hardware traversal units and the programmable cores running the software point-tet test. Our goal is to be able to make a single call to `rtTrace` and immediately get back just the ID of the tet containing the point, with no software execution required in between.

To achieve this, we first note that each tetrahedron is enclosed by four faces, meaning that any non-infinitesimal ray traced from

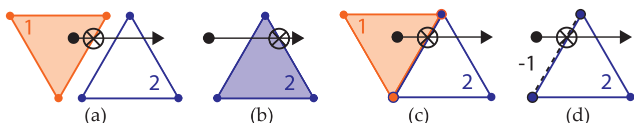


Figure 3: *(a) `rtx-rep-faces` uses backface culling to avoid reporting co-planar neighboring faces; (b) however, points outside any tet can return false positives, as the exterior faces are hidden, requiring an extra point in tet test for correctness. (c,d) `rtx-shrd-faces` does not duplicate faces and thus does not need backface culling, giving the correct result in both cases.*

a point within the tet will hit one of these faces. Furthermore, each face is a triangle, and ray-triangle intersection is accelerated by RTX. Thus, we can represent the tetrahedra by their faces, and instead of going back and forth between hardware BVH traversal and software intersection, can let the hardware perform both the BVH traversal and ray-triangle intersection. When an intersection is found we will be given the intersected triangle ID, which we can use to determine the containing tet.

5.1. Variant 1: `RTX-Replicated-Faces`

The most straightforward way to implement this idea is to create a triangle for each face of each tetrahedron, and use a *closest-hit* program which looks up which tet the hit triangle belongs to.

This technique is easy to implement, but in practice has some caveats. First, interior faces are now represented twice, and we need a way to ensure that the ray only reports the current tet's face, and not the co-planar face from its neighbor. We solve this by constructing the triangles such that they always face inward, and trace the ray with backface culling enabled (Figure 3a, Figure 5). Ray traversal, intersection, and backface culling are now all performed in hardware, and we can simply trace a ray and let the hardware do the work until the right face is found, eliminating the back and forth between hardware and software in the previous methods.

Though the method as described so far works perfectly well for any query point inside a tet, without further care it may return false positives for points outside. As shown in Figure 3b, a ray from a point outside the mesh can travel into a tet and, with backface culling enabled, will not intersect the boundary face, but rather the next interior face, incorrectly marking the point as contained in the boundary tet. To ensure correctness in all cases we perform an additional point-in-tet test inside the GI's *closest-hit* program. Unlike the reference and `rtx-bvh` methods this test needs to be done only once per ray, and thus is relatively cheap, more so when per-vertex scalar interpolation is desired, as these values will also be needed for interpolation.

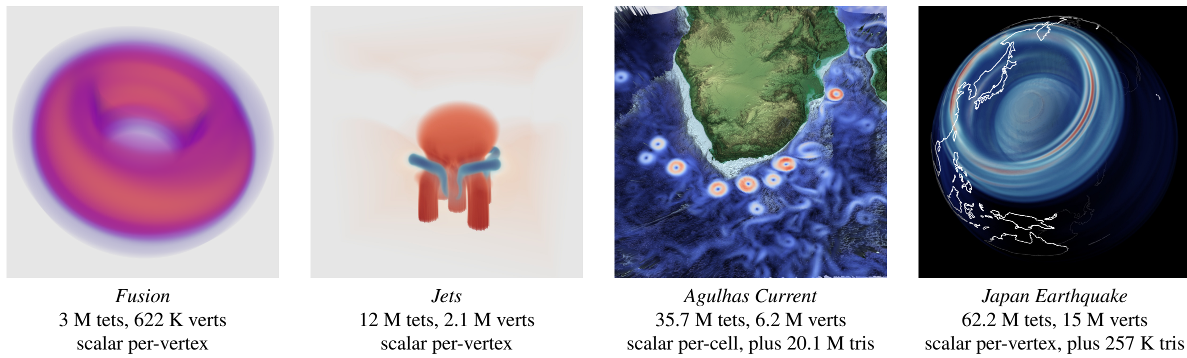
5.2. Variant 2: `RTX-Shared-Faces`

Instead of replicating shared faces, the obvious alternative is to find faces shared by neighboring tetrahedra and merge them. Although this preprocessing step is expensive, the benefits are significant. The resulting output triangle mesh is much smaller, and no longer requires special treatment to cull co-planar duplicate faces.

For each face we now store two integers, which specify the IDs of the tets on its front and back side (or -1 if no tet exists on that side). In the *closest-hit* program we use OptiX's `rtIsTriangleHitBackFace()` to return the correct tet ID based on which side of the face was hit (Figure 6). As backface culling is no longer needed to hide co-planar faces, the `rtx-shrd-faces` method eliminates the caveats of the `rtx-rep-faces` method discussed above.

To compute the set of unique faces we define a *unique representation* of a given face by sorting its vertex indices. We then use a `std::vector` to store the unique faces and a `std::map` to map from the unique face representation to a unique face index. In a second pass we go over faces of each tet to find their unique representation,

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.



Fusion	*Jets*	*Agulhas Current*	*Japan Earthquake*
3 M tets, 622 K verts	12 M tets, 2.1 M verts	35.7 M tets, 6.2 M verts	62.2 M tets, 15 M verts
scalar per-vertex	scalar per-vertex	scalar per-cell, plus 20.1 M tris	scalar per-vertex, plus 257 K tris

Figure 4: *Sample images and statistical data for the data sets used in evaluating our kernels.* Agulhas *and* Japan *include triangle meshes for the bathymetry and continent outlines, respectively. These are used during rendering but do not effect the point query kernels.*

```
struct Face { int3 index; int tetID; };
rtBuffer<Face, 1> faceBuffer;
rtDeclareVariable(float, prd, rtPayload, );
rtDeclareVariable(rtObject, world, , );
rtDeclareVariable(float, maxEdgeLength, , );
RT_PROGRAM void closest_hit() {
  const int faceID = rtGetPrimitiveIndex();
  const int tetID  = faceBuffer[faceID].tetID;
  float fieldValue;
  if (interpolateTet(tetID, ray.origin, fieldValue))
    prd = fieldValue;
}
__device__ float getSample(const vec3f P) {
    Ray ray(P, vec3f(1), 0, 0.f, maxEdgeLength);
    float fieldValue = negInf;
    rtTrace(world, ray, fieldValue,
            RT_VISIBILITY_ALL,
            RT_RAY_FLAG_CULL_BACK_FACING_TRIANGLES
            |RT_RAY_FLAG_DISABLE_ANYHIT);
    return fieldValue;
}
```

Figure 5: *Pseudocode kernels for our `rtx-rep-faces` method.*

```
struct Face { int3 index; int2 tetIDs; };
rtBuffer<Face, 1> faceBuffer;
RT_PROGRAM void closest_hit() {
  int   faceID = rtGetPrimitiveIndex();
  int2 tetIDs = faceBuffer[faceID].tetIDs;
  int   tetID  = rtIsTriangleHitBackFace() ?
                 tetID.y : tetID.x;
  if (tetID < 0) return;
  // store ID or compute scalar field ...
}
```

Figure 6: *The* closest_hit *program for `rtx-shrd-faces` . (*getSample() *is the same as in Figure 5).*

and store the tet as the front or back side tet for the unique face, depending on the face's orientation.

5.3. Common Implementation Details

Once the set of triangles is generated for each method, the actual OptiX set-up code is almost identical. We create an `optix::GeometryTriangles` (GT) for the triangle mesh and assign the triangle vertices and indices using the GT's `setVertices` and `setTriangleIndices` methods. The GT is then placed in an `optix::GeometryInstance` with the respective *closest-hit* program, and assigned the buffer of tet IDs computed by the kernel, with either one or two ints per-triangle. As an optimization, for both methods we explicitly disable the *any-hit* program. This guarantees to the ray tracer that it can skip calling the *any-hit* program, avoiding any back-and-forth between hardware traversal and an empty software *any-hit* program.

A key difference of both methods when compared to the `rtx-bvh` kernel is that we can no longer use an infinitesimal ray length, since such short rays would not reach the faces. Although infinite length rays would intersect the faces, this would necessarily require the hardware to perform more traversal operations. Even with hardware acceleration, this is expensive. To address this, we compute the maximum edge length of any tet in the data set, and use this as the ray's `ray.tmax` value. This ensures that rays can reach the right faces, while limiting the traversal distance.

6. Evaluation

Given these four kernels, we can now evaluate their relative performance. All experiments are run on a mid-range workstation with an Intel Core i7–5930k CPU, 32 GBs of RAM, and one or more of the GPUs listed in Table 1. In particular, we evaluate on both a consumer and high-end RTX-enabled card (RTX 2080 and Titan RTX respectively) and, for reference, a pre-Turing Titan V.

Our experiments are run on Ubuntu 18.04 using OptiX 6.0, with NVIDIA driver version 418.43 and CUDA 10.1. The data sets used for evaluation cover a range of shapes and sizes (see Figure 4), from 3 to nearly 63 million tets. All but the *Jets* data set are sparse, in that only part of the data's bounding box is covered by tets. For *Fusion*, only the torus is covered; in *Agulhas*, tets only cover "wet" cells (roughly 50% of the bounding box); and for *Japan*, only nonzero cells are included, covering just 7.15% of the bounding box.

6.1. Memory Usage

We first measured the total memory usage for the various methods, listed in Table 2. We observe that on Turing our kernels require significantly less memory than on Volta, especially for the triangle-based variants. Irrespective of the GPU architecture, we found that OptiX 6.0 exhibited a significant difference between its *final* memory usage, after all data structures had been built, and its *peak* memory usage, while these data structures were being built. Although this overhead is temporary, it was significant enough that some of our experiments initially ran out of memory on the RTX 2080.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

	Synthetic Uniform (samples/sec)				**Synthetic Random (samples/sec)**				**Volume Rendering (FPS, 1024^2 pix)**			
	fusion	jets	agulh	jpn-qk	fusion	jets	agulh	jpn-qk	fusion	jets	agulh	jpn-qk
#tets	(3M)	(12M)	(36M)	(62M)	(3M)	(12M)	(36M)	(62M)	(3M)	(12M)	(36M)	(62M)
%of bbox occupied:	54.15%	100%	49.3%	7.15%	54.15%	100%	49.3%	7.15%	54.15%	100%	49.3%	7.15%
Titan V (Volta, no RTX, 5120 cores@1.2 GHz, 12 GB HBM2 RAM)												
`cuda-bvh`	89.7M	1.55G	971M	461M	36.4M	82.4M	83.8M	70.3M	13.98	27.64	24.62	5.15
`rtx-bvh`	91.8M	1.05G	741M	373M	30.2M	108M	83.6M	68.6M	5.74	13.7	17.3	3.07
`rtx-rep-faces`	34.7M	407M	`(oom)`	`(oom)`	23.7M	81.5M	`(oom)`	`(oom)`	5.82	8.79	`(oom)`	`(oom)`
`rtx-shrd-faces`	59.7M	689M	397M	`(oom)`	35.1M	101M	63.6M	`(oom)`	9.4	13.2	`(oom)`	`(oom)`
RTX 2080 (Turing, with RTX, 2944 cores@1.8 GHz, 46 RT Cores, and 8 GB GDDR6 RAM)												
`cuda-bvh`	53M	996M	563M	263M	19.7M	60.5M	53.3M	44.1M	8.85	17.2	19.6	3.18
`rtx-bvh`	98.2M	1.17G	1.03G	525M	24.7M	74.7M	69M	59.6M	6.45	9.78	13.1	3
`rtx-rep-faces`	253M	1.23G	1.11G	`(oom)`	65.2M	159M	126M	`(oom)`	21.6	22.5	27.9	`(oom)`
`rtx-shrd-faces`	354M	1.62G	1.58G	1.28G	76.1M	175M	130M	100M	33.7	27.5	35.4	5.53
Titan RTX (Turing, with RTX, 4608 cores@1.35 GHz, 72 RT Cores, and 24 GB GDDR6 RAM)												
`cuda-bvh`	82.5M	1.39G	799M	384M	30.4M	88.7M	76.5M	62.8M	12.2	22	24	4.44
`rtx-bvh`	145M	1.67G	1.43G	736M	37.1M	111M	99.5M	83.9M	7.44	11	16.2	3.57
`rtx-rep-faces`	377M	1.78G	1.67G	1.36G	97M	234M	182M	133M	27.7	27.1	31.9	7.89
`rtx-shrd-faces`	537M	2.39G	2.31G	1.89G	112M	258M	189M	145M	41	32.1	40.1	7.22

Table 1: *Performance results for all our kernels, across all data sets, and across different benchmarks. All experiments use pre-splitting (see Section 6.1), and are averaged across several runs to reduce launch overhead.* `(oom)` *indicates OptiX ran out of memory during the BVH build.*

While we expect upcoming versions of OptiX to reduce this build-time overhead, we have implemented a workaround to reduce it. We first spatially partition the set of primitives into groups of at most 1 million each, then put each group into its own geometry instance with its own acceleration structure. These GIs are then put into a "top-level" `optix::Group`. As each individual BVH is much smaller the peak memory usage is lower, allowing even the 8 GB card to fit all but one experiment. The resulting two-level data structure is fully supported by RTX, and does not significantly impact performance (some experiments even performed marginally better).

6.2. Benchmark Performance

To measure just the raw query performance of our kernels we conducted a set of synthetic benchmarks (Table 1). We first performed these benchmarks by taking uniformly and randomly distributed sample points within the volume's bounding box; however, as most models are sparse many of these samples will not be inside any tet, making them artificially less expensive to compute. This led to an unrealistically high average sampling rate for the kernels.

	Volta, no RTX				**Turing, with RTX**			
model	fusion	jets	agulh	jpn-qk	fusion	jets	agulh	jpn-qk
#tets	3M	12M	36M	62M	3M	12M	36M	62M
`cuda-bvh` (Section 3)								
final	725M	921M	2.0G	3.2G	466M	844M	1.9G	3.1G
`rtx-bvh` (Section 4)								
peak (no p/s)	837M	2.4G	6.3G	10.6G	656M	2.1G	5.7G	9.6G
peak (w/ p/s)	725M	1.6G	3.9G	6.5G	504M	1.1G	2.1G	4.4G
final	717M	1.6G	3.8G	6.1G	464M	754M	1.7G	3.1G
`rtx-rep-faces` (Section 5.1)								
#faces	11.9M	49.1M	143M	249M	11.9M	49.1M	143M	249M
peak (no p/s)	2.5G	9.0G	`(oom)`	`(oom)`	1.6G	5.9G	16.9G	`(oom)`
peak (w/ p/s)	2.1G	7.3G	`(oom)`	`(oom)`	1.2G	2.3G	6.1G	11.0G
final	2.1G	7.2G	`(oom)`	`(oom)`	770M	1.8G	5.4G	10.7G
`rtx-shrd-faces` (Section 5.2)								
#faces	5.99M	24.7M	72M	134M	5.99M	24.7M	72M	134M
peak (no p/s)	1.5G	4.9G	`(oom)`	`(oom)`	960M	3.3G	9.3G	16.9G
peak (w/ p/s)	1.3G	4.1G	11.3G	`(oom)`	846M	1.7G	4.4G	7.2G
final	1.3G	4.0G	11.3G	`(oom)`	643M	1.4G	3.9G	6.8G

Table 2: *GPU memory cost for our four kernels. "Peak" is the peak memory used by OptiX during the BVH build (with and without pre-splitting); "final" is the total memory required after BVH construction. Additional non-volume data, e.g., framebuffer and surface meshes, are not included.*

Such a purely spatial sample distribution is not entirely unrealistic; in fact, it is exactly what our prototype volume renderer in the next section will generate. Nevertheless, we felt these numbers to be artificially inflated, and changed to a method where samples are always placed within valid tets. The *uniform* benchmark launches one thread per tet, and takes a sample at its center. The *random* benchmark has each thread select a random tet, and sample a random position within its tet.

On the pre-Turing Titan V (i.e., without RTX acceleration) we see that, as expected, performance decreases as we increasingly use more ray tracing. Tracing a ray is inherently more expensive than querying a point, and without hardware acceleration it will be slower. Despite this theoretically higher cost, when run with RTX acceleration, our kernels not only perform well, but in fact outperform the reference method significantly—by $1.7-6.5\times$ on the *uniform* benchmark and $2.3-3.7\times$ on the *random* benchmark.

An interesting outlier in these results is our smallest data set, *Fusion*, which sees the worst absolute performance on the synthetic benchmarks. The tets in the *Fusion* data set have a much larger difference between the min and max edge lengths, and are densely packed around the center line of the torus. As the RTX based methods use the max edge length as the ray query distance, they will traverse many more BVH nodes than required on the *Fusion* compared to the other data sets, impacting performance on the synthetic benchmarks.

On the *random* benchmark we find that, as expected, the poor coherence of the query points impacts performance on all the kernels evaluated. Across all the methods we see a decrease on the order of $5-10\times$; however our RTX accelerated kernels continue to outperform the reference. The *uniform* benchmarks achieve much higher sample rates on all methods, with our `rtx-shrd-faces` kernel achieving on the order of $1-2$ billion samples per-second. This translates to $1-2$ billion rays per-second, which far exceeded our

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

expectations, as the rays are by no means coherent from a rendering sense. Our experimental setup actually guarantees that even in the *uniform* case no two rays will ever hit the same face.

6.3. Unstructured Volume Ray Marching

To see how these speedups translate to a more challenging application, we implemented a prototype volume ray marcher similar to that presented by Rathke et al. [RWCB15]. For each pixel we march a ray through the volume's bounding box and sample it at a fixed step size, sampling approximately once per-tet. At each sample point the renderer uses the respective kernel to compute the scalar field value. The field value is then assigned a color and opacity from a transfer function stored in a 1D texture, and accumulated along the ray until the ray's opacity exceeds 99%. Samples which are not in a tet are treated as fully transparent. If surface geometry is provided it is put into an `optix::TrianglesGeometry`, the renderer then traces a ray to find the nearest surface intersection and performs volume integration only up to the surface.

The speedups achieved by our kernels on the synthetic benchmarks carry over to rendering, with our fastest method achieving a $1.5-4\times$ speedup over the reference (Table 1). The frame rate is more dependent on the data size, and in absolute terms decreases as the data size grows. We note that this is not due to an increase in cost per-sample, but rather due to our relatively naive volume ray marcher. The ray marcher uses a fixed step size and does not implement empty space skipping, thus for large but sparse data sets it will take a large number of samples which are not contained in a tet. While each such sample is relatively cheap, in aggregate they are not. This could be alleviated by adding support for space skipping or adaptive sampling to our renderer.

6.4. Power Draw

Another noteworthy experiment is to compare the different methods in terms of power draw. According to `nvidia-smi`, both `cuda-bvh` and `rtx-bvh` always reach roughly the card's maximal power draw (225W for the RTX 2080, 280W for the TITAN RTX). However, the RTX triangle based kernels consistently draw less power, averaging around 170W on the RTX 2080 and 230W on the TITAN RTX. By leveraging new hardware capabilities, our kernels achieve a $2\times$ or higher performance improvement, while using around 20% less power.

7. Discussion and Conclusion

Though our results are promising, more work remains to be done. We have shown *one* application where the RTX cores can be used for other purposes; however, we do not know how far this idea will carry beyond sampling tetrahedral meshes. Some initial results for other data types appear promising, but require further investigation. As for the kernels presented in this paper, we have yet to test them in a real non-rendering application, such as a simulation.

Another caveat is we have only investigated taking individual samples. Sampling is the most general approach for dealing with unstructured data and applicable to both rendering and simulations; however, for rendering specifically, other techniques may be more efficient (e.g., [Gri19, NLKH12, MHDG11]). Some of our general ideas may apply to such techniques as well, though this requires further investigation. Similarly, even in a sample based renderer the RTX cores may be better used for tasks besides sampling, e.g., empty space skipping or rendering surface effects.

Nevertheless, our results are encouraging. Not only did our first attempt to use the RTX cores for something beyond classical ray tracing work at all, each of the three kernels evaluated provided improvement over the reference, with the fastest methods far exceeding our expectations. This raises two related but different questions for current and future hardware architectures. First, what other more general tree traversal problems might the current iteration of these hardware units be able to accelerate? And second, how might hypothetical changes to future iterations of these or similar hardware units change the answer to that question? Our encouraging results in this work serve to motivate further investigation into general applications of the RTX hardware.

Acknowledgments

The Agulhas data set is courtesy Dr. Niklas Röber (DKRZ); the Japan Earthquake data set is courtesy of Carsten Burstedde, Omar Ghattas, James R. Martin, Georg Stadler, and Lucas C. Wilcox (ICES, the University of Texas at Austin) and Paul Navrátil and Greg Abram (TACC). Hardware for development and testing was graciously provided by NVIDIA Corp. This work is supported in part by NSF: CGV Award: 1314896, NSF:IIP Award: 1602127, NSF:ACI Award: 1649923, DOE/SciDAC DESC0007446, CCMSC DE-NA0002375 and NSF:OAC Award: 1842042.

References

[Gri19] GRIBBLE C.: Multi-Hit Ray Tracing in DXR. In *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, Haines E., Akenine-Möller T., (Eds.). 2019.

[HTDH18] HAIDAR A., TOMOV S., DONGARRA J., HIGHAM N. J.: Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed Up Mixed-precision Iterative Refinement Solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (Proceedings of Supercomputing '18)* (2018).

[LM01] LARSEN E. S., MCALLISTER D.: Fast Matrix Multiplies Using Graphics Hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (2001).

[MHDG11] MUIGG P., HADWIGER M., DOLEISCH H., GROLLER E.: Interactive Volume Visualization of General Polyhedral Grids. *IEEE Transactions on Visualization and Computer Graphics* (2011).

[NLKH12] NELSON B., LIU E., KIRBY R. M., HAIMES R.: ElVis: A System for the Accurate and Interactive Visualization of High-Order Finite Element Solutions. *IEEE Transactions on Visualization and Computer Graphics* (2012).

[NVI18a] NVIDIA: NVIDIA OptiX 6.0–Programming Guide. `https://bit.ly/2ErCDti`, 2018.

[NVI18b] NVIDIA: NVIDIA TURING GPU ARCHITECTURE. `https://bit.ly/2NGLr5t`, 2018.

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A.: OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)* (2010).

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

[RS01a] RUMPF M., STRZODKA R.: Level Set Segmentation in Graphics Hardware. In *Proceedings of the 2001 International Conference on Image Processing* (2001).

[RS01b] RUMPF M., STRZODKA R.: Using Graphics Cards for Quantized FEM Computations. In *Proceedings of the VIIP Conference on Visualization and Image Processing* (2001).

[RWCB15] RATHKE B., WALD I., CHIU K., BROWNLEE C.: SIMD Parallel Ray Tracing of Homogeneous Polyhedral Grids. In *Eurographics Symposium on Parallel Graphics and Visualization* (2015).

[WJA*17] WALD I., JOHNSON G. P., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J., GÜNTHER J., NAVRÁTIL P.: OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics* (2017).

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree - A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics* (2014).

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.