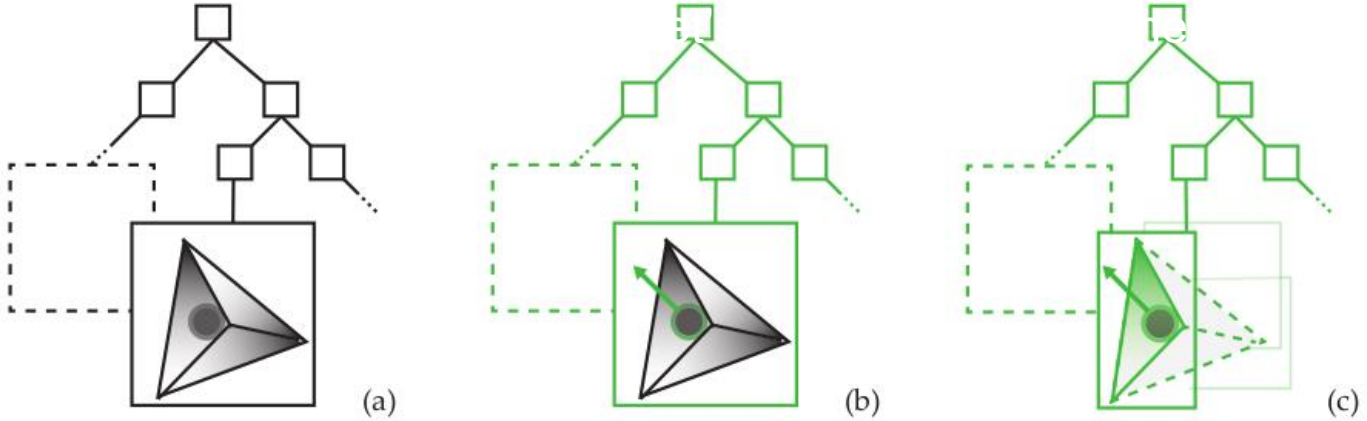
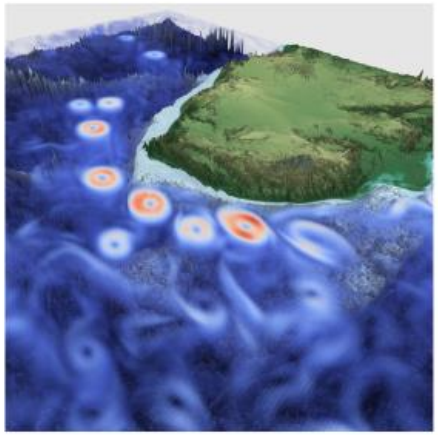
RTX Beyond Ray Tracing

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





I Wald (NVIDIA),
W Usher, N Morrical, L Lediaev, V Pascucci (University of Utah)

Motivation – What this is about

- In this paper: We accelerate Unstructured-Data (Tet Mesh) Volume Ray Casting…

Motivation – What this is about

- **In this paper: We accelerate Unstructured-Data (Tet Mesh) Volume Ray Casting…**

- **But: This is *not* what this is (primarily) about**
  - Volume rendering is just a “proof of concept”.

- **Original question: “What *else*” can you do with RTX?**
  - Remember the early 2000’s (e.g., “register combiners”): Lots of innovation around “using graphics hardware for non-graphics problems”.
  - Since CUDA: Much of that has been subsumed through CUDA
  - Today: Now that we have *new* hardware units (RTX, Tensor Cores), what else could we (ab-)use *those* for? (“(ab-)use” as in “use for something that it wasn’t intended for”)

Motivation – What this is about

- **In this paper: We accelerate Unstructured-Data (Tet Mesh) Volume Ray Casting…**

- **But: This is *not* what this is (primarily) about**
  - Volume rendering is just a “proof of concept”.

- **Original question: “What *else*” can you do with RTX?**
  - Remember the early 2000’s (e.g., “register combiners”):

→ Two main goal(s) of this paper:
a) Get readers to think about the “what else”s…
b) Show one first proof-of-concept
(for this paper, tet-mesh volume rendering)

Background: Volume Rendering Unstructured Data in OSPRay

Background: Volume Rendering Unstructured Data in OSPRay

Two key components:

***A) Renderer* operates on abstract volume data type**

- All any volume offers is a method to “sample(pos) → scalar”
- Renderer does ray marching; sample()’s the volume, integrates

Background: Volume Rendering Unstructured Data in OSPRay

Two key components:

***A) Renderer* operates on abstract volume data type**

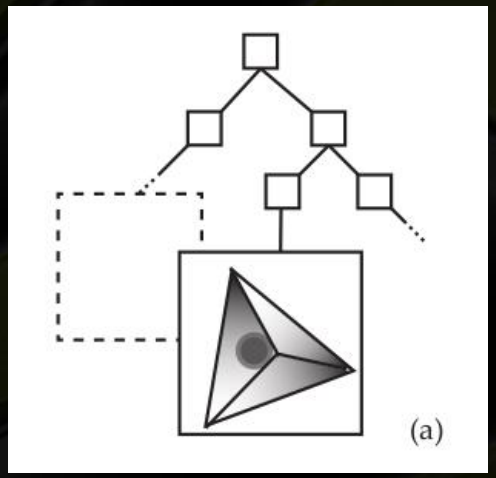
- All any volume offers is a method to "sample(pos) → scalar"
- Renderer does ray marching; sample()'s the volume, integrates

B) Specifically for *tet* mesh volumes:

- Have Bounding Volume Hierarchy (BVH) over tet prims
- Sample() traverses sample point down this BVH
  - Reject subtrees that do not contain sample pos
  - "Recursively" traverse those that do
  - Perform point-in-tet tests when reaching leaves
  - Back-track …

Reference Method: “CUDA BVH”

- **Reference: Exactly that method… just in CUDA**
  - Build BVH over tets (on the host), then upload to GPU
  - Use quad-BVH, with node quantization [Benthin et al’18]
    - No particular reason – that code was just available
  - Sample() method traverses BVH and performs point-tet tests
  - Renderer ray marches, calls “sample()”, calls XF, integrates, …



Reference Method: “CUDA BVH”

- **Pro:**
  - Conceptually simple. Same as reference method, done.

- **Con:**
  - Lots of code for BVH construction & traversal
    (actually pretty expensive code, too)
  - Doesn’t use RT Cores at all

    →No acceleration beyond CUDA

    →No benefit from RTX at all ☹

→Question: Can we do more? Can use use RTX? How?

Method #1: “OptiX-BVH”

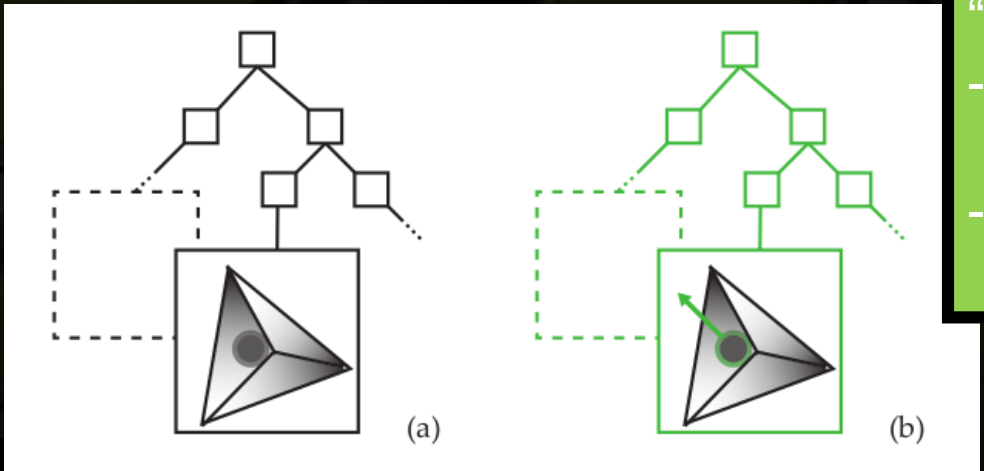
- **Observation: sample() very similar to ray tracing ….**
  - BVH, BVH traversal, prim intersection, ….

→**“Make it fit” by viewing samples as zero-length “ray”s**

  - Traversing a ray will automatically traverse any point *on* that ray
    → For any sample “position”, just “trace” a zero-length ray
    → Will do *more* than required (eg, order…), but will be correct!
  - Direction of ray doesn’t matter – pick (1,1,1)
    - Implementation note: Have to use Epsilon-length, not zero …

Method #1: “OptiX-BVH”

- **Once re-formulated as a ray problem: Use OptiX**
  - Create optix::Geometry(Instance) over tets
    - Primitive count = #tets
  - Write bounding box program for tets (trivial)
  - Write intersection program that performs “origin-in-tet” test
    - (just ignore the ray direction, it doesn’t matter!)
  - Trigger traversal by calling rtTrace() ***at each sample location***
  - Same raygen program as reference method, just different sample()



“OptiX-BVH” method:
- Still using same CUDA point-in-prim test
- Replaced CUDA BVH trav with OptiX/RTX BVH trav.

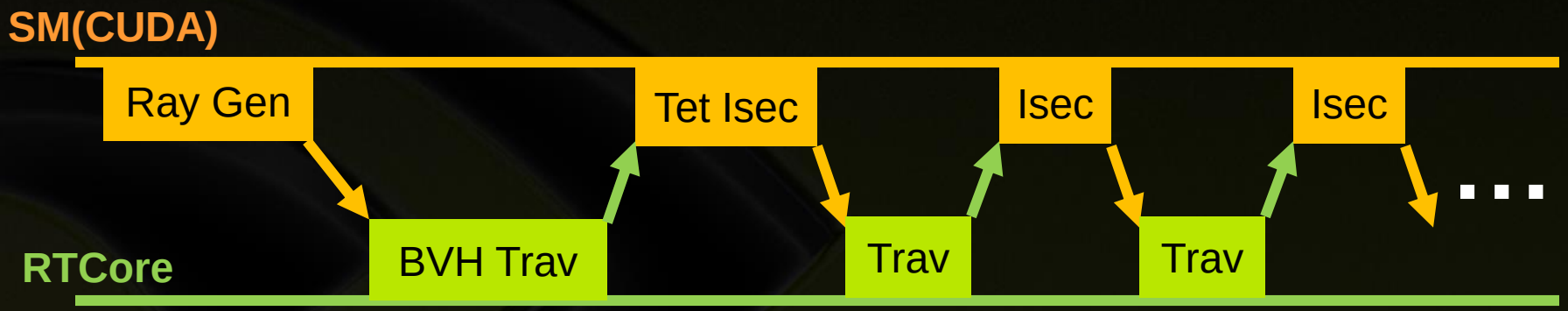
Method #1: “OptiX-BVH”

- **Pro:**
  - *Much* simpler: No longer need to deal with BVH build / traversal … (let OptiX deal with that …)
  - Automatically use RT Cores when available
- **Con:**
  - “Ray traversal” *a*ctually does *more* work than “point location”!
  - RT Cores can *only* accelerate BVH traversal, not intersection ☹

    →Limits potential speedup (Amdahl’s Law)

Method #2: “RTX (replicated) Faces”

- **Problem: Can only accelerate traversal. (Illustration)**



→ **Q: How can we use even *more* of RTX HW?**

- How can we et rid of CUDA intersection?
- How can we get rid of alternating HW-trav / SW isec?

Method #2: “RTX (replicated) Faces”

- **Core observations**
  a) Any ray *leaving* sample will have to hit one of enclosing tet’s faces
  b) Faces are triangles (and RTX supports triangles!)

→Trace rays against tet *faces*, use RTX Triangle HW

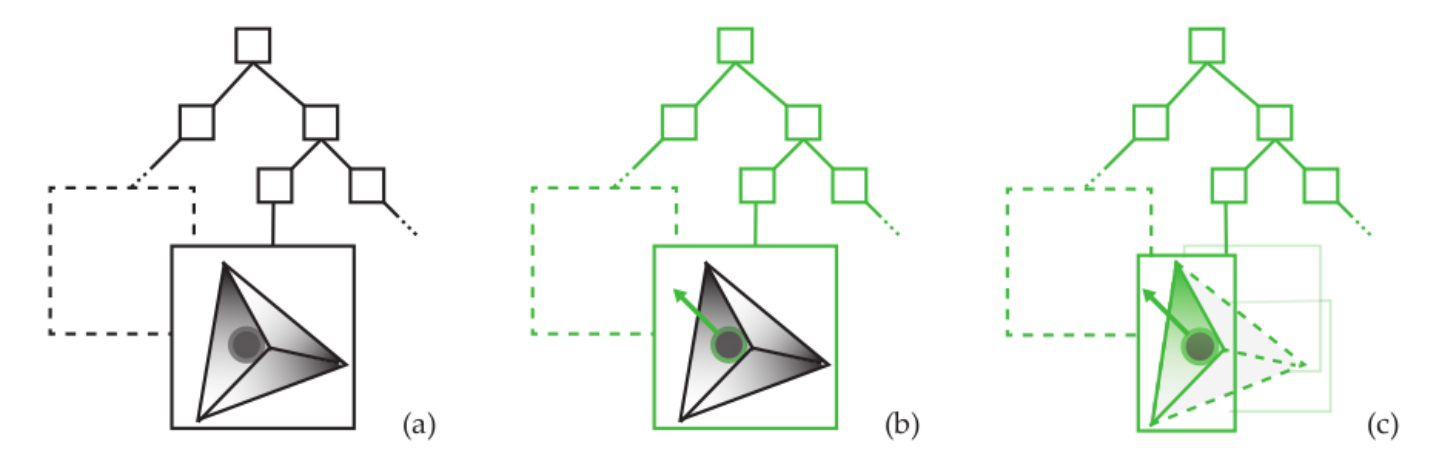
- Represent each tet through its four face triangles
- Tag each face with ID of its tet ID (index.w=tetID)
- Trace *non-zero length* ray (arbitrary direction), find closest face
  →Closest-hit program reads index.w … there’s your tet!

Method #2: “RTX (replicated) Faces”

- **Core observations**
  a) Any ray *leaving* sample will have to hit one of enclosing tet’s faces
  b) Faces are triangles (and RTX supports triangles!)

→Trace rays against tet *faces*, use RTX Triangle HW

- Represent each tet through its four face triangles
- Tag each face with ID of its tet ID (index.w=tetID)
- Trace *non-zero length* ray (arbitrary direction), find closest face
  →Closest-hit program reads index.w … there’s your tet!



Method #2: “RTX (replicated) Faces”

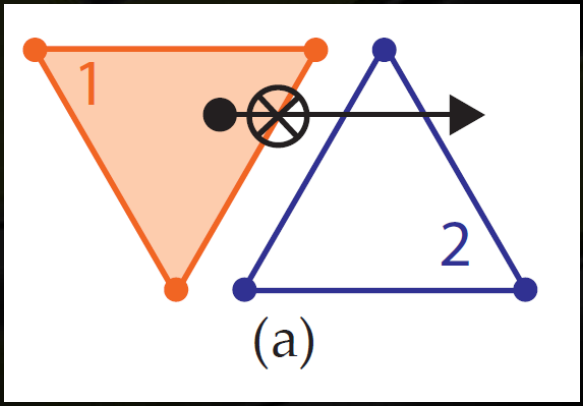
- **Core observations**
  a) Any ray *leaving* sample will have to hit one of enclosing tet’s faces
  b) Faces are triangles (and RTX supports triangles!)

→Trace rays against tet *faces*, use RTX Triangle HW

- **In theory, now *much* more work than point location**
  - 4x num prims
  - BVH over triangles, not boxes
  - Non-infinitesimal rays: Actual traversal
  - Ray-triangle tests
- **But: With hardware support**
  - All of traversal & isec, SM only gets final hitpoint

Method #2: “RTX (replicated) Faces”

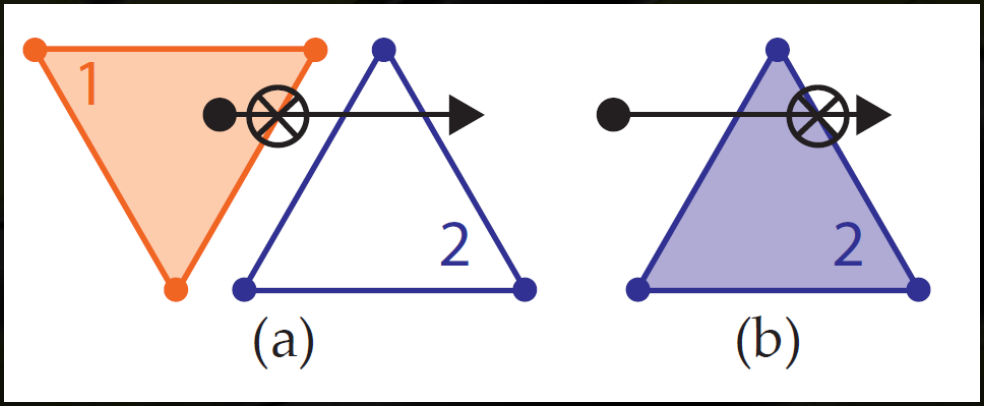
- **Caveat #1: Have to find the *right* faces**
  - Adjacent triangles *each* produce same face
    - Have to make sure we report the “right” one!
  - Solution: Inward-facing triangles + “Back-face” Culling
    - Supported by RTX → “free”



Method #2: “RTX (replicated) Faces”

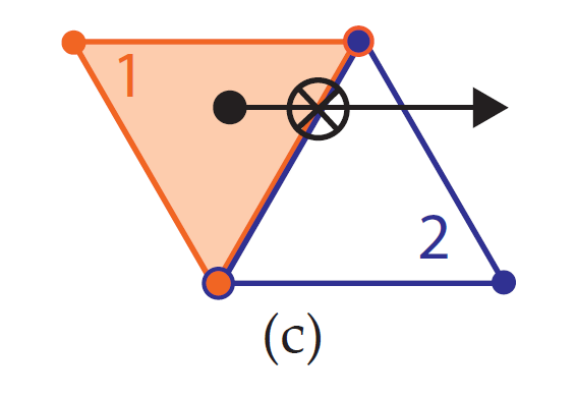
- **(Unexpected) Caveat #2: Query points *outside* any tet**
  - May report “false positive”

  →Require additional (software-)point-in-tet test on “final” closest hit
    - Only one such test per sample … but still expensive ☹



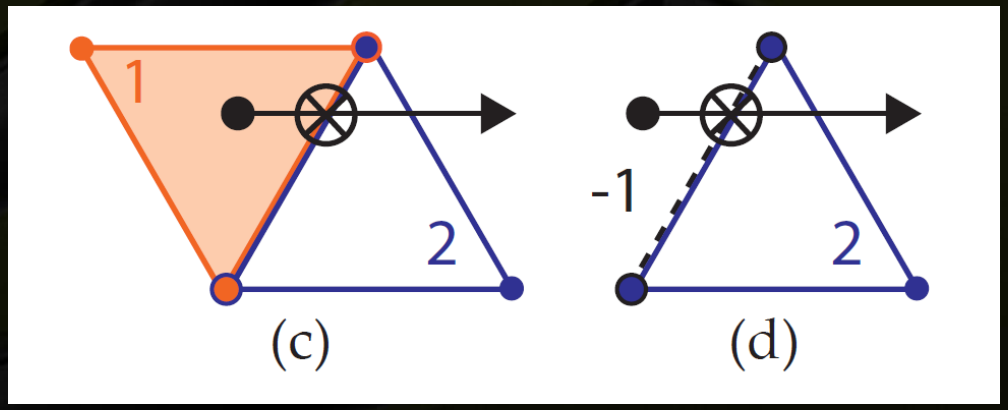
Method #3: “RTX (shared) Faces”

- **Q: Can we avoid this final software test?**
- **Idea: Store *shared* faces**
  - Shared faces only stored once
    - Tagged with ID of both tet on front, *and* that on back (-1 = none)
    - In closest hit program: → rtIsTriangleHitBackFace() (cheap)



Method #3: “RTX (shared) Faces”

- **Q: Can we avoid this final software test?**
- **Idea: Store *shared* faces**
- **Con:**
  - Required precompute pass (host, tricky ☹)
- **Pro:**
  - 2x fewer triangles, cheaper BVH
  - Automatically fixes “outside any tet” case (tetID=“-1”) → faster



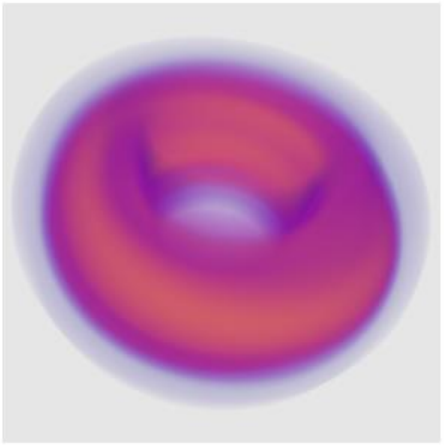
Evaluation

Evaluation: HW/SW Setup

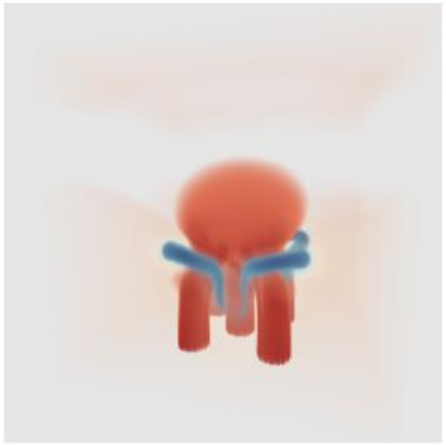
- **Host: Regular PC**
  - Intel Core i7–5930k CPU, 32 GB Ram
- **Software Stack**
  - Linux (Ubuntu 18)
  - Nvidia Driver 418.35 (regular, public driver)
  - OptiX version: initially 5.5, then 6.0 after release
- **Three GPUs: Both pre- and post-RTX**
  - V100 (Volta) : Pre-RTX, high-end, for reference
    - 5120 cores @ 1.2Ghz , 12 GB HBM2 Memory, no dedicated RT Cores
  - RTX 2080 FE: Turing w/ RTX support
    - 2944 cores @ 1.8GHz, 46 RT Cores, 8 GB DDR6 RAM
  - Titan RTX: Turing w/ RTX
    - 4608 cores @ 1.35 GHz, 72 RT Cores, 24 GB DDR6 RAM

Evaluation: Data Sets

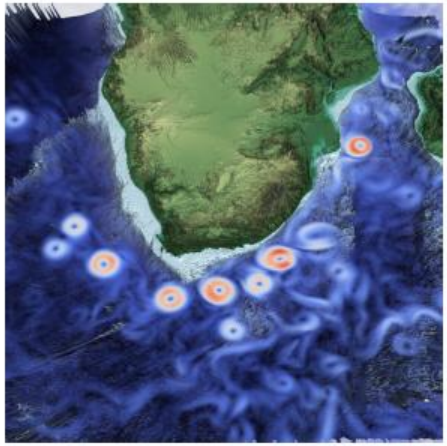
- **Four different tet data sets, varying complexity**
  - Only tets for now
    - (generalization “in the works”)
  - Only focused on “realistically large” data sets
  - Some data sets also contain triangles (“bathymetry”, outlines, …)
  - Some require interpolation (per vertex data) others not (per cell)



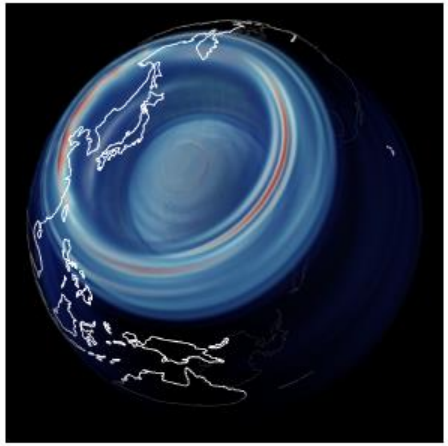
Fusion
3 M tets, 622 K verts
scalar per-vertex



Jets
12 M tets, 2.1 M verts
scalar per-vertex



Agulhas Current
35.7 M tets, 6.2 M verts
scalar per-cell, plus 20.1 M tris



Japan Earthquake
62.2 M tets, 15 M verts
scalar per-vertex, plus 257 K tris

Memory …

	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

Memory: RTX vs pre-RTX

	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	
peak (w/ p/s)	725M	1.6G	3.9G	6.5G	504M	1.1G	2.1G	
final	717M	1.6G	3.8G	6.1G				3.1G
`rtx-rep-faces` (Section 5.1)								
#faces	11.9M	49.1M	143M	249M	11.9M	49		249M
peak (no p/s)	2.5G	9.0G	(oom)	(oom)	1.6G	5		(oom)
peak (w/ p/s)	2.1G	7.3G	(oom)	(oom)	1.2G	2.5G	6.1G	11.0G
final	2.1G	7.2G	(oom)	(oom)	770M	1.8G	5.4G	10.7G
ces (Section 5.2)								
				134M	5.99M	24.7M		
				(oom)	960M	3.3G	9	
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)	645M	1.4G	3.9G	6.8G

~2x

~4x

~3x

Significantly lower memory usage on RTX-enabled HW

Memory: Pre-Splitting vs Naive

model	Volta, no RTX				Turing, with RTX			
	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	[illegible]	
final	717M	1.6G	3.8G	6.1G	464M	754M	1.7G	
`-faces` (Section 5.1)								
			3M	249M	11.9M	49.1M	143M	249M
			om)	(oom)	1.6G	5.9G	16.9G	(oom)
			om)	(oom)	1.2G			[illegible]
final	2.1G	7.2G	(oom)	(oom)	770M			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)			[illegible]	(oom)	846M			[illegible]
final			11.3G	(oom)	643M			6.8G

Pre-splitting reduced peak mem usage by ~2x (~same final mem & perf)

-70%

<inf>

<inf>

-60%

Memory: Per Method …

model	Volta, no RTX				Turing, with RTX			
	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				9.6G
peak (w/ p/s)	725M	1.6G	3.9G	6.5G				4.4G
final	717M	1.6G	3.8G	6.1G				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)				(oom)
								11.0G
								10.7G
Section 5.2)								
					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)				7.2G
final	1.3G	4.0G	11.3G	(oom)				6.8G

~same

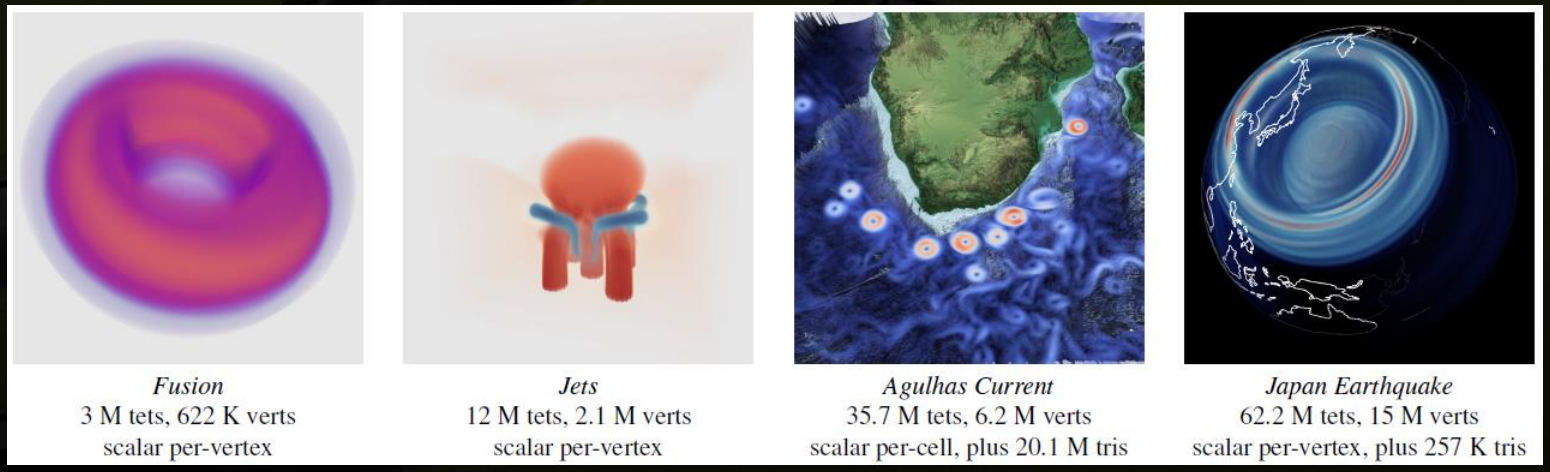
3x worse

2+x worse

Adding (many) triangles creates significant mem overhead… surprise! ☹

Performance

- **Let's run some *more* experiments …**
  a) "Synthetic" cost-per-sample
  b) "Integrated" volume render performance



Fusion
3 M tets, 622 K verts
scalar per-vertex

Jets
12 M tets, 2.1 M verts
scalar per-vertex

Agulhas Current
35.7 M tets, 6.2 M verts
scalar per-cell, plus 20.1 M tris

Japan Earthquake
62.2 M tets, 15 M verts
scalar per-vertex, plus 257 K tris

Performance - Synthetic

	Synthetic Uniform (samples/sec)				Synthetic Random (samples/sec)			
	fusion	jets	agulh	jpn-qk	fusion	jets	agulh	jpn-qk
#tets	(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%
Titan V (Volta, no RTX)								
cuda-bvh	89.7M	1.55G	971M	461M	36.4M	82.4M	83.8M	70.3M
rtx-bvh	91.8M	1.05G	741M	373M	30.2M	108M	83.6M	68.6M
rtx-rep-faces	34.7M	407M	(oom)	(oom)	23.7M	81.5M	(oom)	(oom)
rtx-shrd-faces	59.7M	689M	397M	(oom)	35.1M	101M	63.6M	(oom)
RTX 2080 (Turing, with RTX)								
cuda-bvh	53M	996M	563M	263M	19.7M	60.5M	53.3M	44.1M
rtx-bvh	98.2M	1.17G	1.03G	525M	24.7M	74.7M	69M	59.6M
rtx-rep-faces	253M	1.23G	1.11G	(oom)	65.2M	159M	126M	(oom)
rtx-shrd-faces	354M	1.62G	1.58G	1.28G	76.1M	175M	130M	100M
Titan RTX (Turing, with RTX)								
cuda-bvh	82.5M	1.39G	799M	384M	30.4M	88.7M	76.5M	62.8M
rtx-bvh	145M	1.67G	1.43G	736M	37.1M	111M	99.5M	83.9M
rtx-rep-faces	377M	1.78G	1.67G	1.36G	97M	234M	182M	133M
rtx-shrd-faces	537M	2.39G	2.31G	1.89G	112M	258M	189M	145M

Performance - Synthetic

	Synthetic Uniform (samples/sec)				Synthetic Random (samples/sec)			
	fusion	jets	agulh	jpn-qk	fusion	jets	agulh	jpn-qk
%of bb								5%
cuda-b								3M
rtx-bv								6M
rtx-rep-faces	34.7M	407M	(oom)	(oom)	23.7M	81.5M		(oom)
rtx-shrd-faces	59.7M	689M	397M	(oom)	35.1M	101M		(oom)
RTX 2080 (Turing, with RTX)								
cuda-bvh	53M	996M	563M	263M	19.7M	60.5M	53.3M	44.1M
rtx-bvh	98.2M	1.17G	1.03G	525M	24.7M	74.7M	69M	59.6M
rtx-rep-faces	253M	1.23G	1.11G	(oom)	65.2M	159M	126M	
rtx-shrd-faces	354M	1.62G	1.58G	1.28G	76.1M	175M	130M	
Titan RTX (Turing, with RTX)								
cuda-bvh	82.5M	1.39G	799M	384M	30.4M	76.5M		62.8M
rtx-bvh	145M	1.67G	1.43G	736M	37.1M	111M	99.5M	83.9M
rtx-rep-faces	377M	1.78G	1.67G	1.36G	97M	234M	182M	133M
rtx-shrd-faces	537M	2.39G	2.31G	1.89G	112M	258M	189M	145M

Note: “Random” is *designed* to be bad.
(*guaranteed* to diverge, for *every* sample)
Across all experiments, see perf drop by ½ to full OOM

~7x

11x

~2.5x

Performance - Synthetic

	Synthetic Uniform (samples/sec)				Synthetic Random (samples/sec)			
	fusion	jets	agulh	jpn-qk	fusion	jets	agulh	jpn-qk
#tets	(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%
				Titan V (Volta, no RTX)				
`cuda-bvh`	89.7M	1.55G	971M	461M	36.4M	82.4M	83.8M	70.3M
`rtx-bvh`	91.8M	1.05G	741M	373M	30.2M	108M	83.6M	68.6M
`rtx-rep-faces`	34.7M	407M	(oom)	(oom)	23.7M	81.5M	(oom)	(oom)
`rtx-shrd-faces`	59.7M	689M	397M	(oom)	35.1M	101M	63.6M	(oom)

More interesting: How do *methods* compare?
Start with pre-RTX….

Performance - Synthetic

	Synthetic Uniform (samples/sec)				Synthetic Random (samples/sec)			
	fusion	jets	agulh	jpn-qk	fusion	jets	agulh	jpn-qk
#tets	(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%
					Titan V (Volta, no RTX)			
cuda-bvh	89.7M	1.55G	971M	461M	36.4M	82.4M	83.8M	70.3M
rtx-bvh	91.8M	1.05G	741M	373M	30.2M	108M	83.6M	68.6M
rtx-rep-faces	34.7M	407M	(oom)	(oom)	23.7M	81.5M	(oom)	(oom)
rtx-shrd-faces	59.7M	689M	397M	(oom)	35.1M	101M	63.6M	(oom)

Pre RTX Hardware:
Perf at best stays about same (random samples)…

Performance - Synthetic

	Synthetic Uniform (samples/sec)				Synthetic Random (samples/sec)			
	fusion	jets	agulh	jpn-qk	fusion	jets	agulh	jpn-qk
#tets	(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%
				Titan V (Volta, no RTX)				
`cuda-bvh`	89.7M	1.55G	971M	461M	36.4M	82.4M	83.8M	70.3M
`rtx-bvh`	91.8M	1.05G	741M	373M	30.2M	108M	83.6M	68.6M
`rtx-rep-faces`	34.7M	407M	(oom)	(oom)	23.7M	81.5M	(oom)	(oom)
`rtx-shrd-faces`	59.7M	689M	397M	(oom)	35.1M	101M	63.6M	(oom)

Pre RTX Hardware:
Perf at best stays about same (random samples)…
… and for uniform samples: more than 2x ***worse***!

(yes, if both are in software, “ray trace” and “triangles” are more expensive than “point query” and “points”!)

Performance - Synthetic

	Synthetic Uniform (samples/sec)				Synthetic Random (samples/sec)			
	fusion	jets	agulh	jpn-qk	fusion	jets	agulh	jpn-qk
#tets	(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%
Titan V (Volta, no RTX)								
cuda-bvh								
rtx-bvh								
rtx-rep-fac								
rtx-shrd-fa								
RTX 2080 (Turing, with RTX)								
cuda-bvh	53M	996M	563M	263M	19.7M	60.5M	53.3M	44.1M
rtx-bvh	98.2M	1.17G	1.03G	525M	24.7M	74.7M	69M	59.6M
rtx-rep-faces	253M	1.23G	1.11G	(oom)	65.2M	159M	126M	(oom)
rtx-shrd-faces	354M	1.62G	1.58G	1.28G	76.1M	175M	130M	100M
Titan RTX (Turing, with RTX)								
cuda-bvh	82.5M	1.39G	799M	384M	30.4M	88.7M	76.5M	62.8M
rtx-bvh	145M	1.67G	1.43G	736M	37.1M	111M	99.5M	83.9M
rtx-rep-faces	377M	1.78G	1.67G	1.36G	97M	234M	182M	133M
rtx-shrd-faces	537M	2.39G	2.31G	1.89G	112M	258M	189M	145M

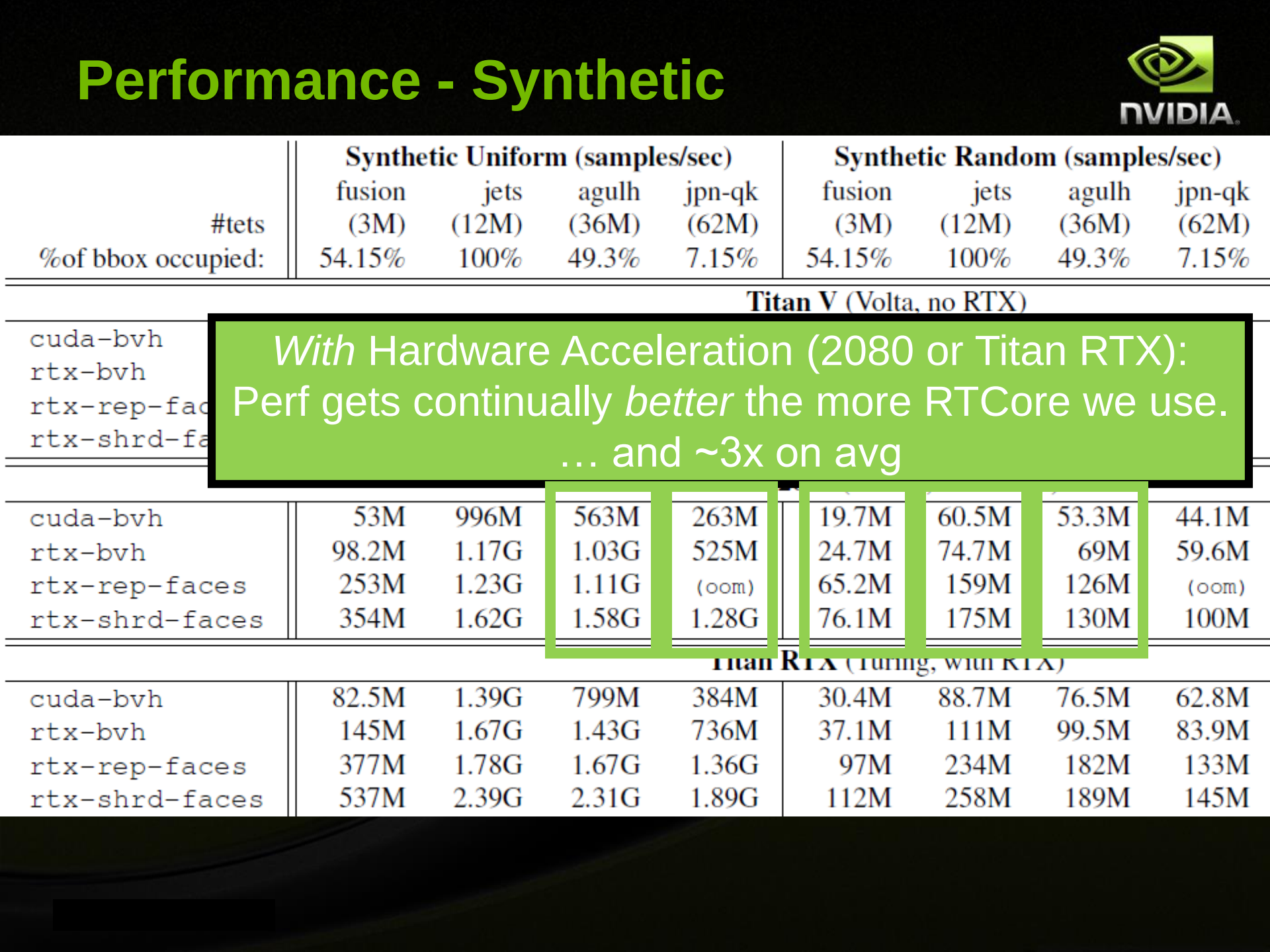
With Hardware Acceleration (2080 or Titan RTX):
Perf gets continually *better* the more RTCore we use.
From ~1.5x ...

Performance - Synthetic

	Synthetic Uniform (samples/sec)				Synthetic Random (samples/sec)			
	fusion	jets	agulh	jpn-qk	fusion	jets	agulh	jpn-qk
#tets	(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%
Titan V (Volta, no RTX)								
cuda-bvh	53M	996M	563M	263M	19.7M	60.5M	53.3M	44.1M
rtx-bvh	98.2M	1.17G	1.03G	525M	24.7M	74.7M	69M	59.6M
rtx-rep-faces	253M	1.23G	1.11G	(oom)	65.2M	159M	126M	(oom)
rtx-shrd-faces	354M	1.62G	1.58G	1.28G	76.1M	175M	130M	100M
Titan RTX (Turing, with RTX)								
cuda-bvh	82.5M	1.39G	799M	384M	30.4M	88.7M	76.5M	62.8M
rtx-bvh	145M	1.67G	1.43G	736M	37.1M	111M	99.5M	83.9M
rtx-rep-faces	377M	1.78G	1.67G	1.36G	97M	234M	182M	133M
rtx-shrd-faces	537M	2.39G	2.31G	1.89G	112M	258M	189M	145M

With Hardware Acceleration (2080 or Titan RTX):
Perf gets continually *better* the more RTCore we use.
… to up to almost 7x ...

Performance - Synthetic

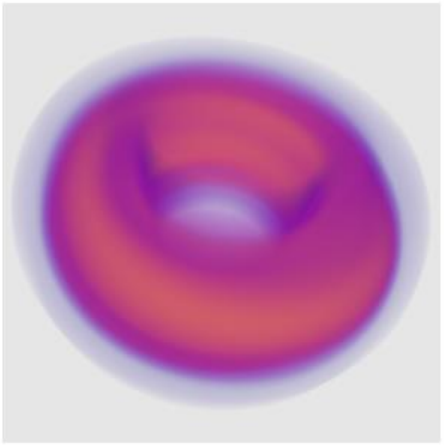


	Synthetic Uniform (samples/sec)				Synthetic Random (samples/sec)			
	fusion	jets	agulh	jpn-qk	fusion	jets	agulh	jpn-qk
#tets	(3M)	(12M)	(36M)	(62M)	(3M)	(12M)	(36M)	(62M)
%of bbox occupied:	54.15%	100%	49.3%	7.15%	54.15%	100%	49.3%	7.15%
Titan V (Volta, no RTX)								
cuda-bvh	53M	996M	563M	263M	19.7M	60.5M	53.3M	44.1M
rtx-bvh	98.2M	1.17G	1.03G	525M	24.7M	74.7M	69M	59.6M
rtx-rep-faces	253M	1.23G	1.11G	(oom)	65.2M	159M	126M	(oom)
rtx-shrd-faces	354M	1.62G	1.58G	1.28G	76.1M	175M	130M	100M
Titan RTX (Turing, with RTX)								
cuda-bvh	82.5M	1.39G	799M	384M	30.4M	88.7M	76.5M	62.8M
rtx-bvh	145M	1.67G	1.43G	736M	37.1M	111M	99.5M	83.9M
rtx-rep-faces	377M	1.78G	1.67G	1.36G	97M	234M	182M	133M
rtx-shrd-faces	537M	2.39G	2.31G	1.89G	112M	258M	189M	145M

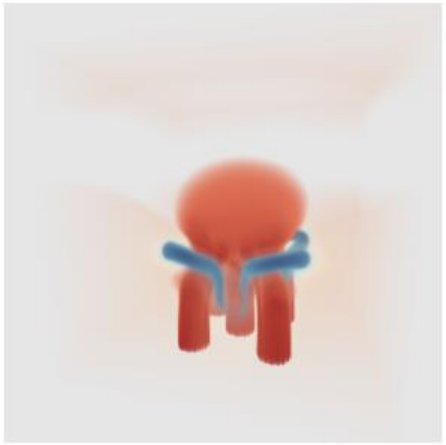
With Hardware Acceleration (2080 or Titan RTX):
Perf gets continually *better* the more RTCore we use.
… and ~3x on avg

Performance – Volume Ray Marching

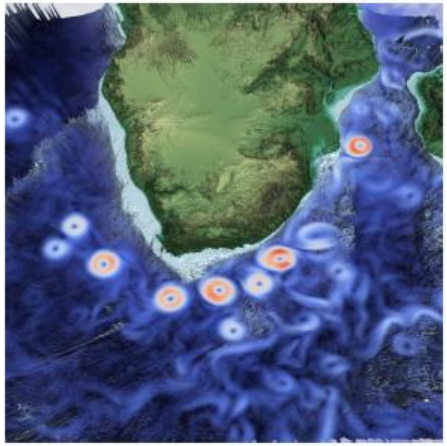
- **Integrated Variants into Actual Volume Ray Marcher**
  - Important: Sample-based! (***not*** tet marching, projected tets, etc)
    - Ie, one ray ***per sample – NOT*** per *pixel*
  - Also trace “some” (geometry) rays for bathymetry (with AO)



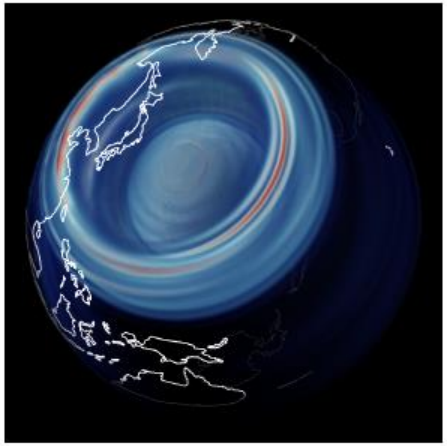
Fusion
3 M tets, 622 K verts
scalar per-vertex



Jets
12 M tets, 2.1 M verts
scalar per-vertex



Agulhas Current
35.7 M tets, 6.2 M verts
scalar per-cell, plus 20.1 M tris



Japan Earthquake
62.2 M tets, 15 M verts
scalar per-vertex, plus 257 K tris

Performance – Volume Ray Marching

Roughly same results as for synthetic:

	Volume Rendering (FPS, 1024^2 pix)			
#tets	(5M)	(12M)	(36M)	(82M)
Titan V (Volta, no RTX)				
cuda-bvh	13.98	27.64	24.62	5.15
rtx-bvh	5.74	13.7	17.3	3.07
rtx-rep-faces	5.82	8.79	(oom)	(oom)
rtx-shrd-faces	9.4	13.2	(oom)	(oom)
RTX 2080 (Turing, with RTX)				
cuda-bvh	8.85	17.2	19.6	3.18
rtx-bvh	6.45	9.78	13.1	3
rtx-rep-faces	21.6	22.5	27.9	(oom)
rtx-shrd-faces	33.7	27.5	35.4	5.53
Titan RTX (Turing, with RTX)				
cuda-bvh	12.2	22	24	4.44
rtx-bvh	7.44	11	16.2	3.57
rtx-rep-faces	27.7	27.1	31.9	7.89
rtx-shrd-faces	41	32.1	40.1	7.22

Performance – Volume Ray Marching

Roughly same results as for synthetic:
- Perf *drops* for pre-RTX HW …

Volume Rendering (FPS, 1024^2 pix)				
cuda-bvh	13.98	27.64	24.62	5.15
rtx-bvh	5.74	13.7	17.3	3.07
rtx-rep-faces	5.82	8.79	(oom)	(oom)
rtx-shrd-faces	9.4	13.2	(oom)	(oom)
RTX 2080 (Turing, with RTX)				
cuda-bvh	8.85	17.2	19.6	3.18
rtx-bvh	6.45	9.78	13.1	3
rtx-rep-faces	21.6	22.5	27.9	(oom)
rtx-shrd-faces	33.7	27.5	35.4	5.53
Titan RTX (Turing, with RTX)				
cuda-bvh	12.2	22	24	4.44
rtx-bvh	7.44	11	16.2	3.57
rtx-rep-faces	27.7	27.1	31.9	7.89
rtx-shrd-faces	41	32.1	40.1	7.22

Performance – Volume Ray Marching

Roughly same results as for synthetic:
- Perf *drops* for pre-RTX HW …
- … but it *does* pay off w/ HW Accel (by ~1.5-3x)

Volume Rendering (FPS, 1024^2 pix)				
rtx-bvh	5.74	13.7	17.3	3.07
rtx-rep-faces	5.82	8.79	(oom)	(oom)
rtx-shrd-faces	9.4	13.2	(oom)	(oom)
RTX 2080 (Turing, with RTX)				
cuda-bvh	8.85	17.2	19.6	3.18
rtx-bvh	6.45	9.78	13.1	3
rtx-rep-faces	21.6	22.5	27.9	(oom)
rtx-shrd-faces	33.7	27.5	35.4	5.53
Titan RTX (Turing, with RTX)				
cuda-bvh	12.2	22	24	4.44
rtx-bvh	7.44	11	16.2	3.57
rtx-rep-faces	27.7	27.1	31.9	7.89
rtx-shrd-faces	41	32.1	40.1	7.22

Performance – Volume Ray Marching

#tets	Volume Rendering (FPS, 1024^2 pix) fusion (3M)	jets (12M)	agulh (36M)	jpn-qk (62M)
Titan V (Volta, no RTX)				
cuda-bvh	13.98	27.64	24.62	5.15
rtx-bvh	5.74	13.7	17.3	3.07
rtx-rep-faces	5.82	8.79		

Eventually: Speedup for *every* dataset.

Bottom line: Re-formulating point location as a ray tracing problem does *more* work … but still makes it faster because of hardware acceleration!

cuda-bvh	12.2	22	24	4.44
rtx-bvh	7.44	11	16.2	3.57
rtx-rep-faces	27.7	27.1	31.9	7.89
rtx-shrd-faces	41	32.1	40.1	7.22

Summary

- **In this paper**
  - Encouraged to look at RT Cores “beyond ray tracing”
  - Our example: Tet-mesh point location / volume sampling
- **Proposed four different methods**
  - With increasing degrees of using RTX
  - General scheme: do *more* work - but get it done faster through HW

→Successful proof-of-concept

- Early work, but already ~2x speedup in actual unstructured-data volume renderer

Future Work

- **Lots of opportunities for rendering / visualization**
  - Biggest-ticket item: Space skipping / adaptive sampling (already working on that)
  - More general: Now that RT is *that* fast, what else can we use it for?

Future Work

- **Lots of opportunities for rendering / visualization**
  - Biggest-ticket item: Space skipping / adaptive sampling (already working on that)
  - More general: Now that RT is *that* fast, what else can we use it for?
- **Interesting questions for non-gfx CS practitioners**
  - What *else* could we use RT Cores for? *Beyond* rendering!?

Future Work

- **Lots of opportunities for rendering / visualization**
  - Biggest-ticket item: Space skipping / adaptive sampling (already working on that)
  - More general: Now that RT is *that* fast, what else can we use it for?
- **Interesting questions for non-gfx CS practitioners**
  - What *else* could we use RT Cores for? *Beyond* rendering!?
- **Interesting questions for HW designers**
  - What if we could trace things *other* than rays?
  - What if we had prims *other* than triangles?
  - What kind of workloads could we accelerate then?

Questions…

Reference Method: “CUDA BVH”

- **Reference: Exactly that method… just in CUDA**
  - Build BVH over tets (on the host), then upload to GPU
  - Use quad-BVH, with node quantization [Benthin et al’18]
    - No particular reason – that code was just available
  - Sample() method traverses BVH and performs point-tet tests
  - Renderer ray marches, calls “sample()”, calls XF, integrates, …

- **Implementation Notes**
  - Reasonably complete volume ray tracer, but not “crazy” advanced
  - “Plain” CUDA implementation, but within OptiX
    - Ie, use OptiX frame buffer, multi-GPU, “raygen” program etc
      → Allowed for sharing code infrastructure w/ other kernels
    - But: Plain CUDA code in raygen program, no rtTrace() etc.

Memory: Per Method …

	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.[illegible]G
peak (w/ p/s)	725M	1.6G	3.9G	6.5G	504M	1.1G	2.1G	4.[illegible]G
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	24[illegible]M
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.[illegible]G
final	2.1G	7.2G	(oom)	(oom)	770M	1.8G	5.4G	10.7G
[illegible]tion 5.2)								
#fa[illegible]					5.9[illegible]			[illegible]M
pe[illegible]					96[illegible]			[illegible]G
peak (w/ p/s)	1.3G	4.1G	11.3G	(oom)	846M	1.7G	4.4G	7.[illegible]G
final	1.3G	4.0G	11.3G	(oom)	643M	1.4G	3.9G	6.8G

But shared-faces method *much* better than replicated

-~40%

Implementation Notes

Ray Length

- **For face-methods: Can no longer use 0-length rays**
  - Infinite-length is correct, but expensive
- **Solution: Precompute maximum edge length**
  - Guaranteed long enough to reach faces → correct
  - Much faster (avoids traversing far-away regions)

Disabling Any-Hit

- **AnyHit Programs in OptiX**
  - Called on *any* valid intersection during a traversal (ie, *not* just the final, closest one)
  - Eg, useful for accumulating opacity for shadow rays
- **But: Can lead to overhead when *not* using them**
  - "Not specifying AH Program" != "not having one"
    - By default, OptiX assigns "empty" AnyHit program
  - Even an "empty" program must run on the SM
    - →Switch from HW traversal to "empty" CUDA prog for every isec!
    - →Significant call overhead, even when doing "nothing"
- **Tip: If you don't need them, *disable* them!**
  - E.g., rtTrace(…RT_RAY_FLAG_DISABLE_ANYHIT…);
  - In our implementation, at times seen ~10% final frame difference

Pre-Splitting

- **Problem(s): BVH Peak Mem limits max model size**
  a) BVH builder needs (non-trivial) temp memory during build
  → "peak" mem (during build) much higher than "final" mem (render)
  b) No "graceful degradation" if peak mem > available GPU mem
  → Peak Mem > Available GPU Mem: "Allocation Error" …
  →Several tested data sets initially didn't work….
  … even if final memory was << GPU memory

Pre-Splitting

- **Problem(s): BVH Peak Mem limits max model size**
- **Suggestion: "Pre-split"**
  - Don't create *one* big optix::GeometryGroup over all prims/tris
  - Instead: on host, pre-split prims into multiple (T)GGs ….
    - E.g., "chunks" of max 1M prims/tris
    - Using simple spatial median parititoning seems fine
  - …. then create second-level group over those (T)GGs
    - Ideally, "force" a build with a rtLaunch(0,0,0) for each (T)GG

→ **Significantly limits peak mem usage**

  - … to peak mem of any *chunk*

- **(Apparently) negligible performance impact**
  - *Does* force a two-level BVH, but since RTX supports that ….