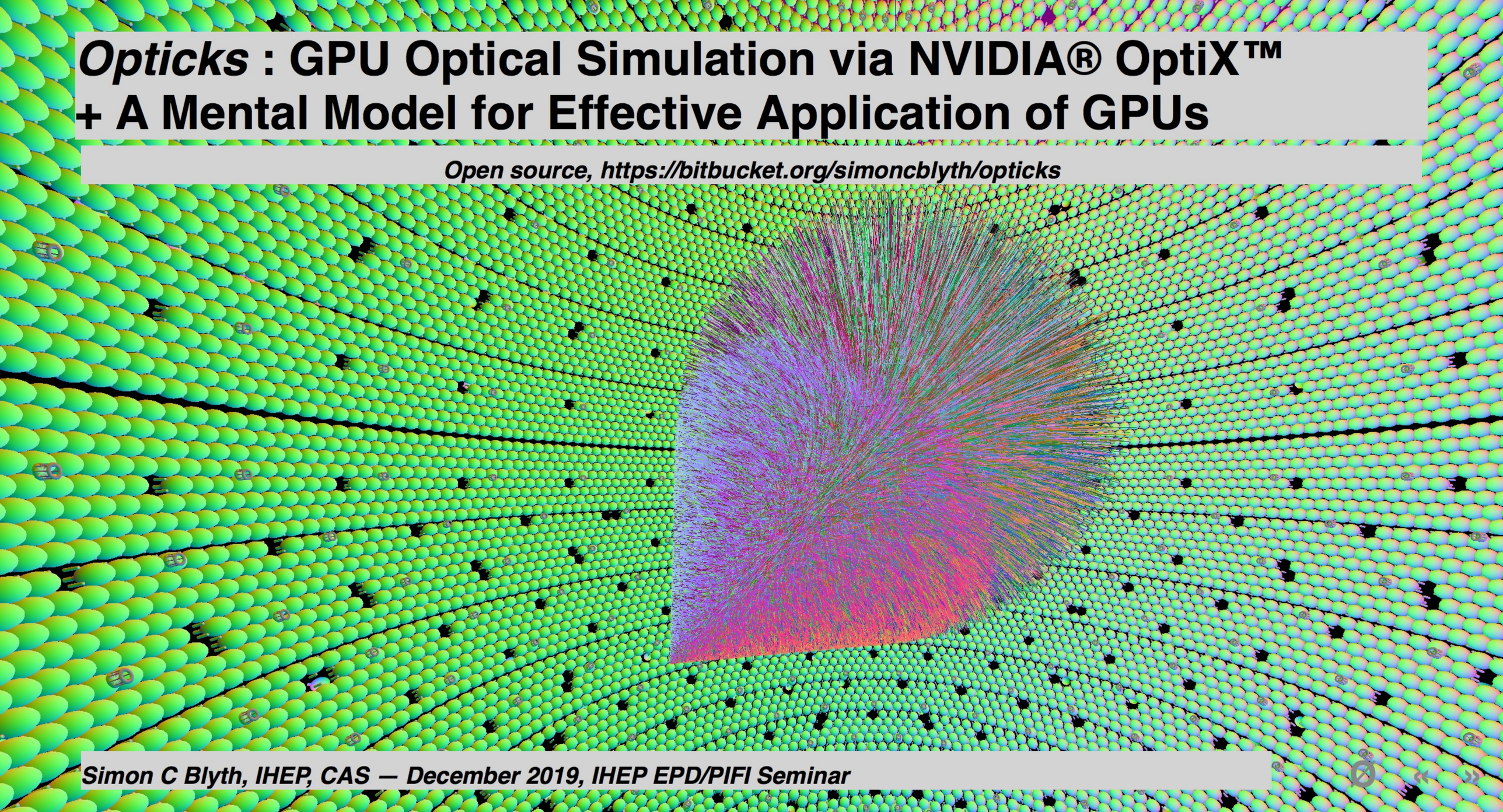
Opticks : GPU Optical Simulation via NVIDIA® OptiX™ + A Mental Model for Effective Application of GPUs

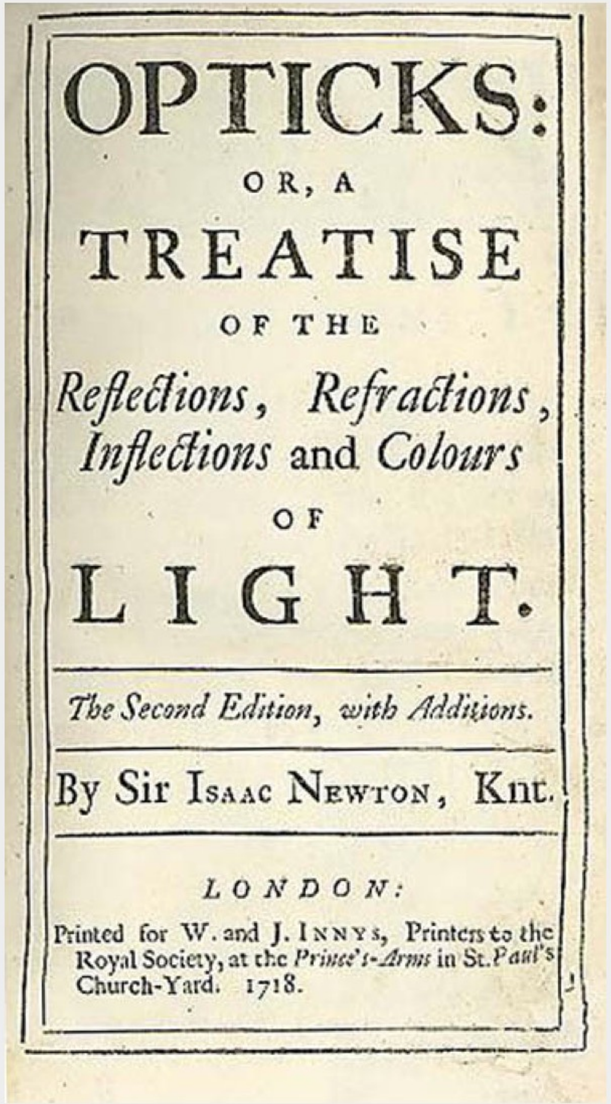
Open source, https://bitbucket.org/simoncblyth/opticks



Simon C Blyth, IHEP, CAS — December 2019, IHEP EPD/PIFI Seminar

Outline

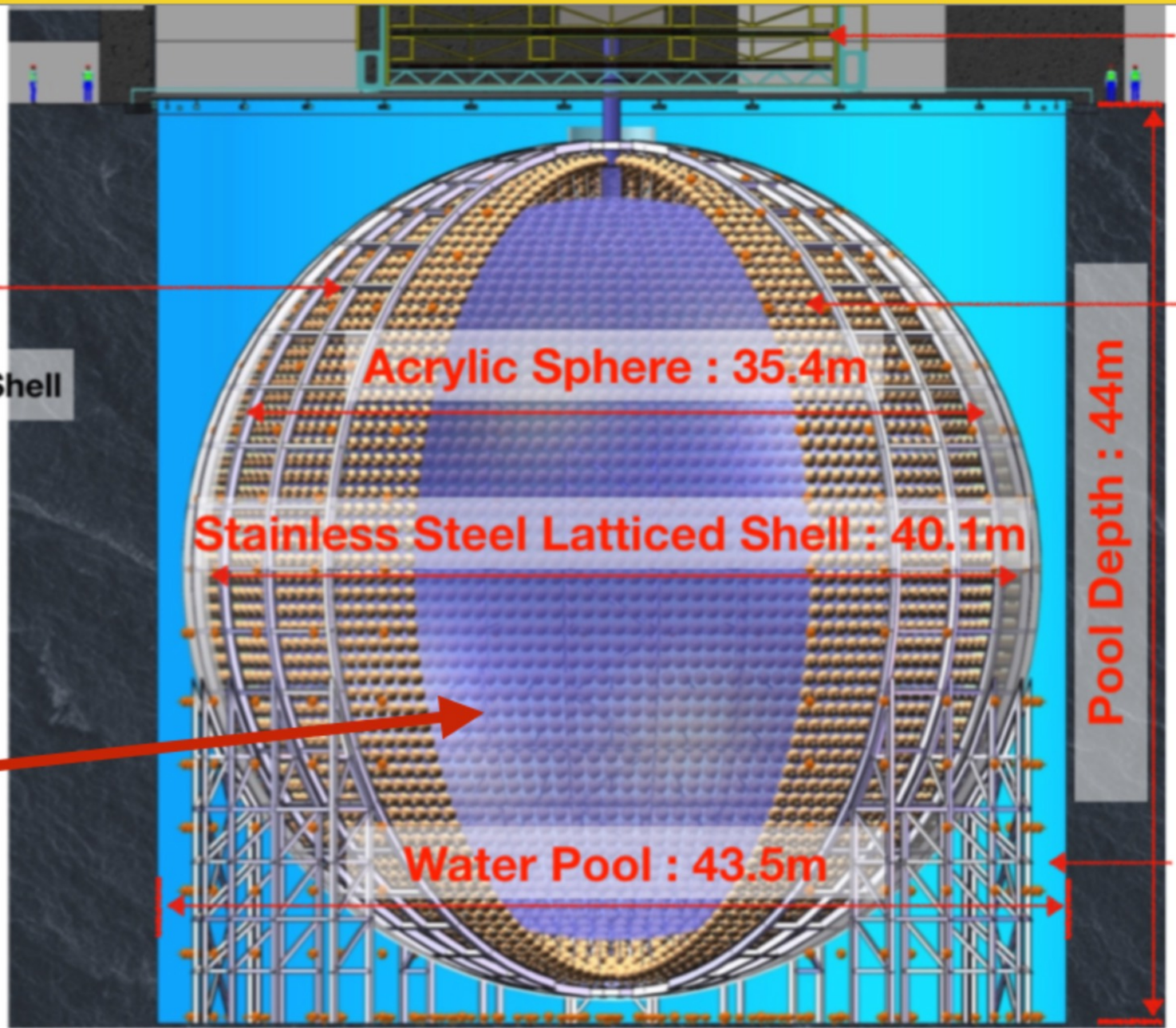
- Context and Problem
  - Jiangmen Underground Neutrino Observatory (JUNO)
  - Optical Photon Simulation Problem...
- Tools to create Solution
  - Optical Photon Simulation ≈ Ray Traced Image Rendering
  - Rasterization and Ray tracing
  - Turing Built for RTX
  - BVH : Bounding Volume Hierarchy
  - NVIDIA OptiX Ray Tracing Engine
- Opticks : The Solution
  - Geant4 + Opticks Hybrid Workflow : External Optical Photon Simulation
  - Opticks : Translates G4 Optical Physics to CUDA/OptiX
  - Opticks : Translates G4 Geometry to GPU, Without Approximation
  - CUDA/OptiX Intersection Functions for ~10 Primitives
  - CUDA/OptiX Intersection Functions for Arbitrarily Complex CSG Shapes
- Validation and Performance
  - Random Aligned Bi-Simulation -> Direct Array Comparison
  - Perfomance Scanning from 1M to 400M Photons
- Overview + Links


OPTICKS:
OR, A
TREATISE
OF THE
Reflections, Refractions, Inflections and *Colours*
OF
LIGHT.
The Second Edition, with Additions.
By Sir Isaac Newton, Knt.
LONDON:
Printed for W. and J. Innys, Printers to the Royal Society, at the *Prince's-Arms* in St. *Paul's* Church-Yard. 1718.

Opticks : GPU Optical Simulation via NVIDIA® OptiX™
+ A Mental Model for Effective Application of GPUs

- GPU Mental Model : Context and Constraints
  - Understanding GPU Graphical Origins -> Effective GPU Computation
  - GPU Demands Simplicity (Arrays) -> Big Benefits : NumPy + CuPy
- Parallel Processing 0th Step
  - Re-shape Data into "Parallel" Array Form
- High Level Tools
  - Survey of High Level General Purpose CUDA Packages
  - NumPy + CuPy
- Example 1 : NumPy/CuPy Python interface
  - Python vs NumPy vs CuPy : Python/NumPy ~ 10, NumPy/CuPy ~ 1300
- Array Mechanics
  - NP.hh header + union "trick"
- Example 2 : A Taste of Thrust C++
  - Photon History Indexing with CUDA Thrust
- Summary

JUNO : Liquid Scintillator, 18k 20-inch PMTs, 25k 3-inch PMTs



Top Tracker

Central Detector

Acrylic Sphere
Stainless Steel Latticed Shell

PMT

18,000 20-inch PMTs
25,000 3-inch PMTs
78% Coverage

Liquid Scintillator

20 kton

Water Cherenkov

35 kton pure water
2,000 20-inch veto PMTs

JUNO : A Multipurpose Neutrino Observatory

Solar ν's
(10-1000)/day

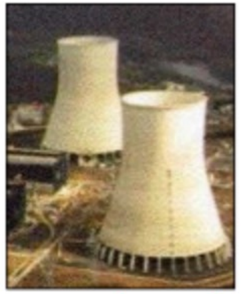
Supernova ν's
~10^4 in 10 s
for 10 kpc

Atmospheric ν's
several/day

700 m

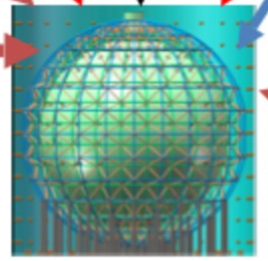
Cosmic muons
~ 250k/day

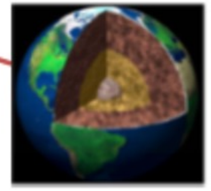
0.003 Hz/m², 215 GeV
10% multiple-muon



reactor ν's
~ 80/day

36 GWth, 53 km





Geo-ν's
1-2/day

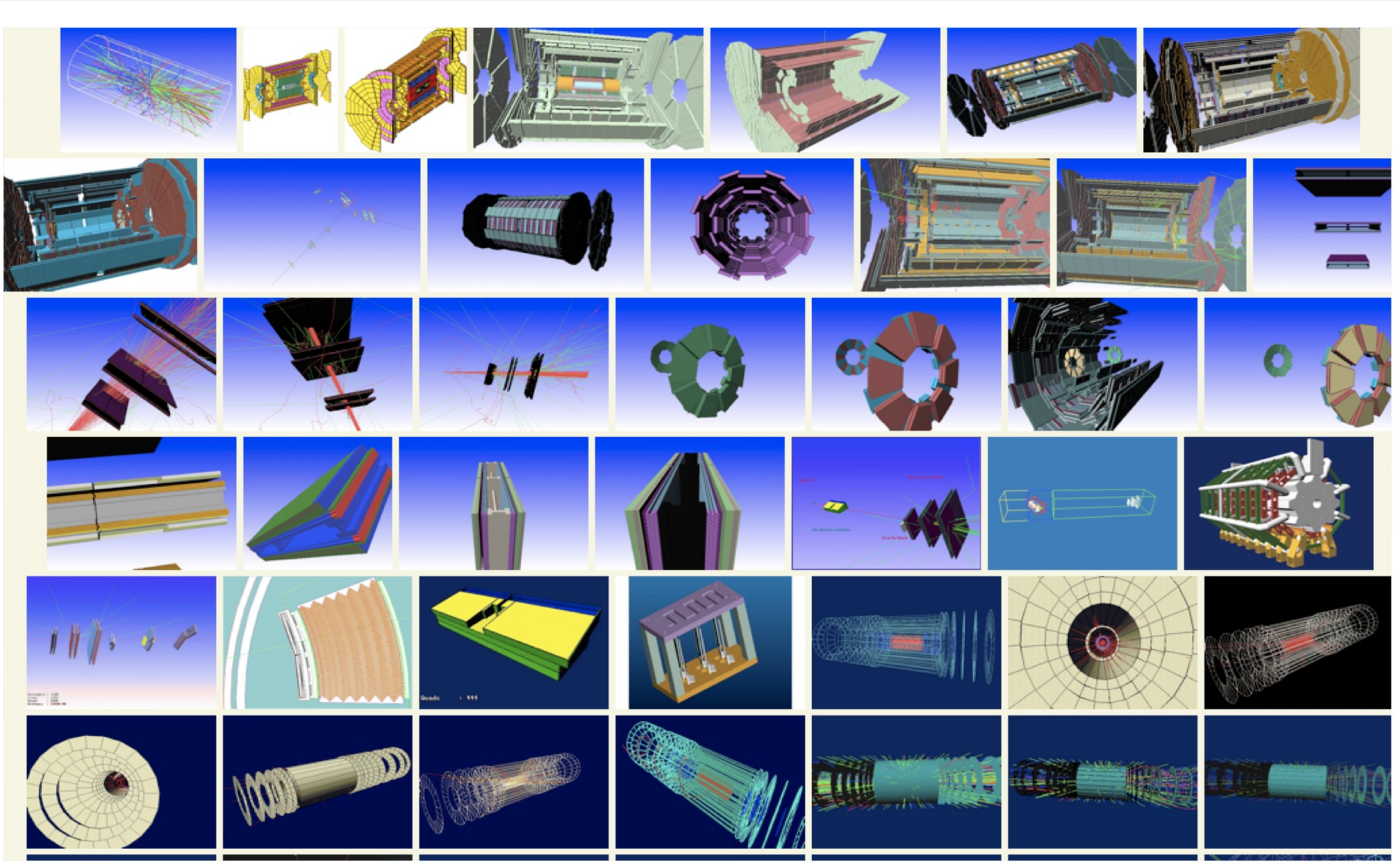
Main backgrounds are cosmic muon induced :

- underground site
- water shield
- muon veto system

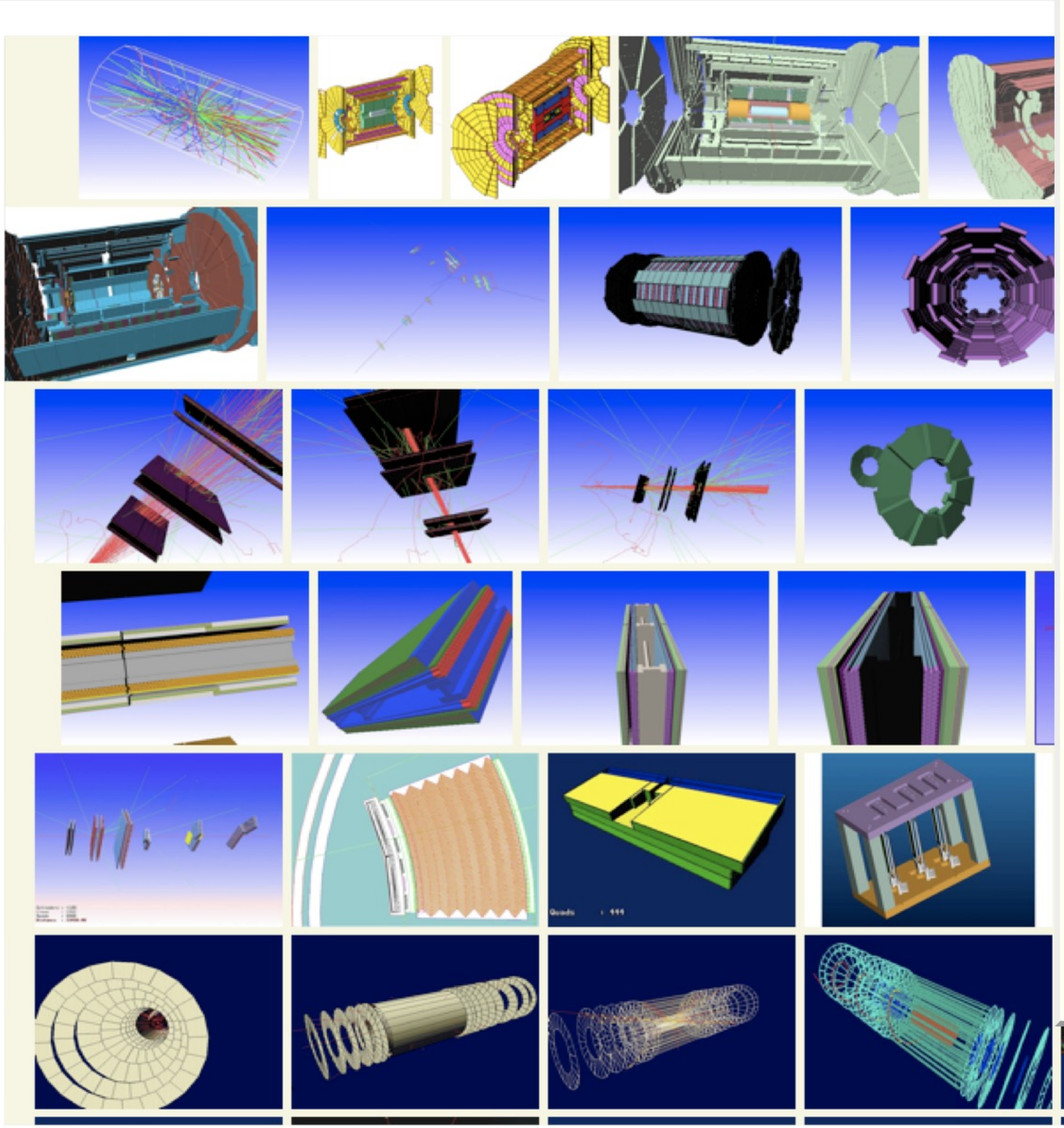
To minimize veto time/volume :

- precise muon reconstruction
- large simulated muon samples are essential for optimization

Geant4 : Monte Carlo Simulation Toolkit



Geant4 : Monte Carlo Simulation Toolkit Generality



Standard Simulation Tool of HEP

Geant4 simulates particles travelling through matter

- high energy, nuclear and accelerator physics
- medical physics : deciding radiotherapy doses/sources
- space engineering : satellites

Geant4 Approach

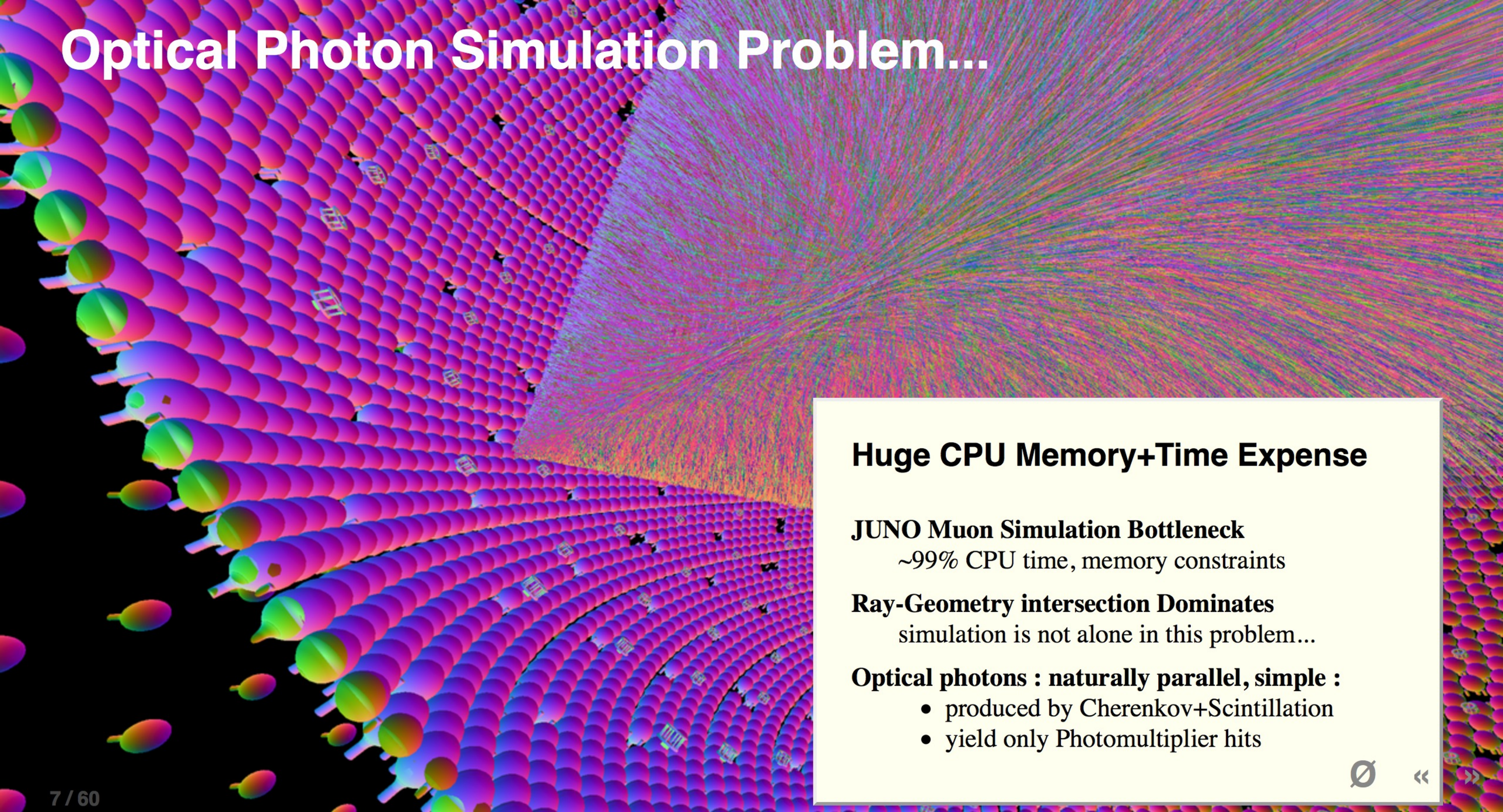
- geometry : **tree of CSG solids**
- particles : track position and time etc..
- processes : nuclear, EM, weak, **optical**

Very General and Capable Tool

- **mostly unused for optical photon propagation**

https://geant4.web.cern.ch

Optical Photon Simulation Problem...



Huge CPU Memory+Time Expense

JUNO Muon Simulation Bottleneck
~99% CPU time, memory constraints

Ray-Geometry intersection Dominates
simulation is not alone in this problem...

Optical photons : naturally parallel, simple :

- produced by Cherenkov+Scintillation
- yield only Photomultiplier hits

Optical Photon Simulation ≈ Ray Traced Image Rendering

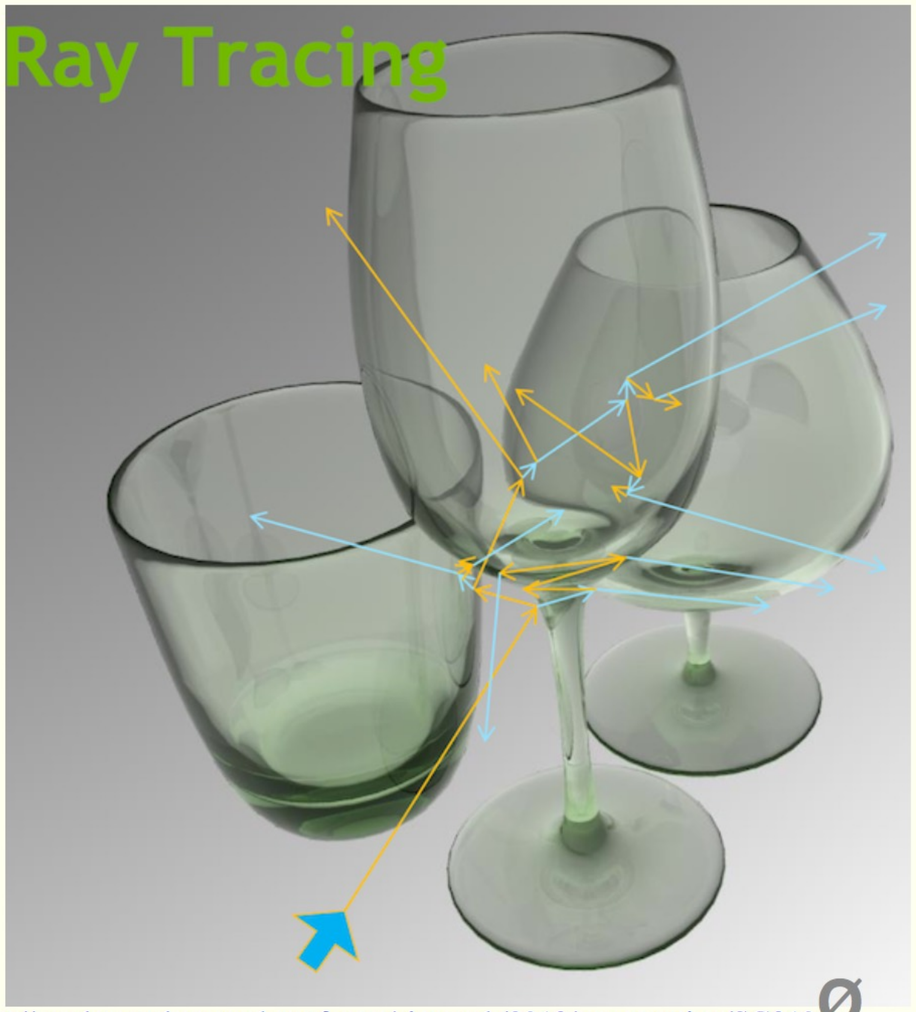
Much in common : geometry, light sources, optical physics

- simulation : photon parameters at PMT detectors
- rendering : pixel values at image plane
- both limited by ray geometry intersection, aka ray tracing

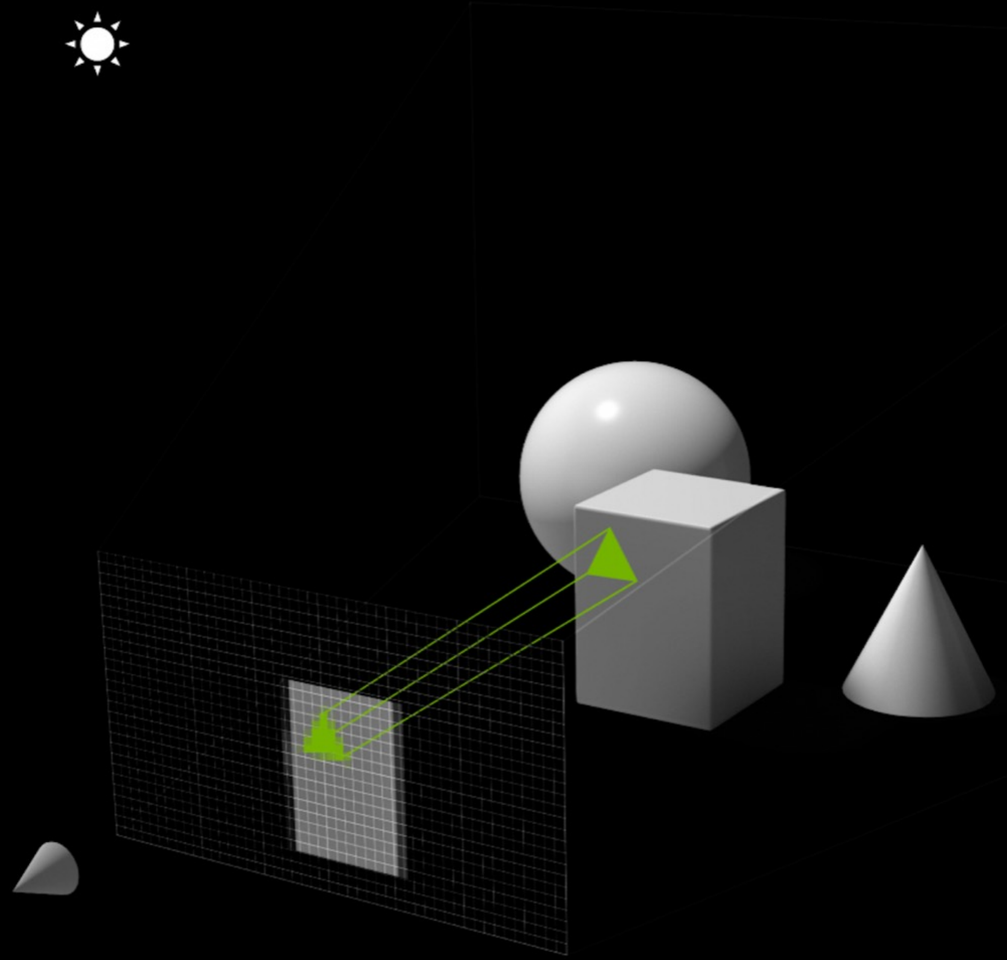
Many Applications of ray tracing :

- advertising, design, architecture, films, games,...
- -> huge efforts to improve hw+sw over 30 yrs

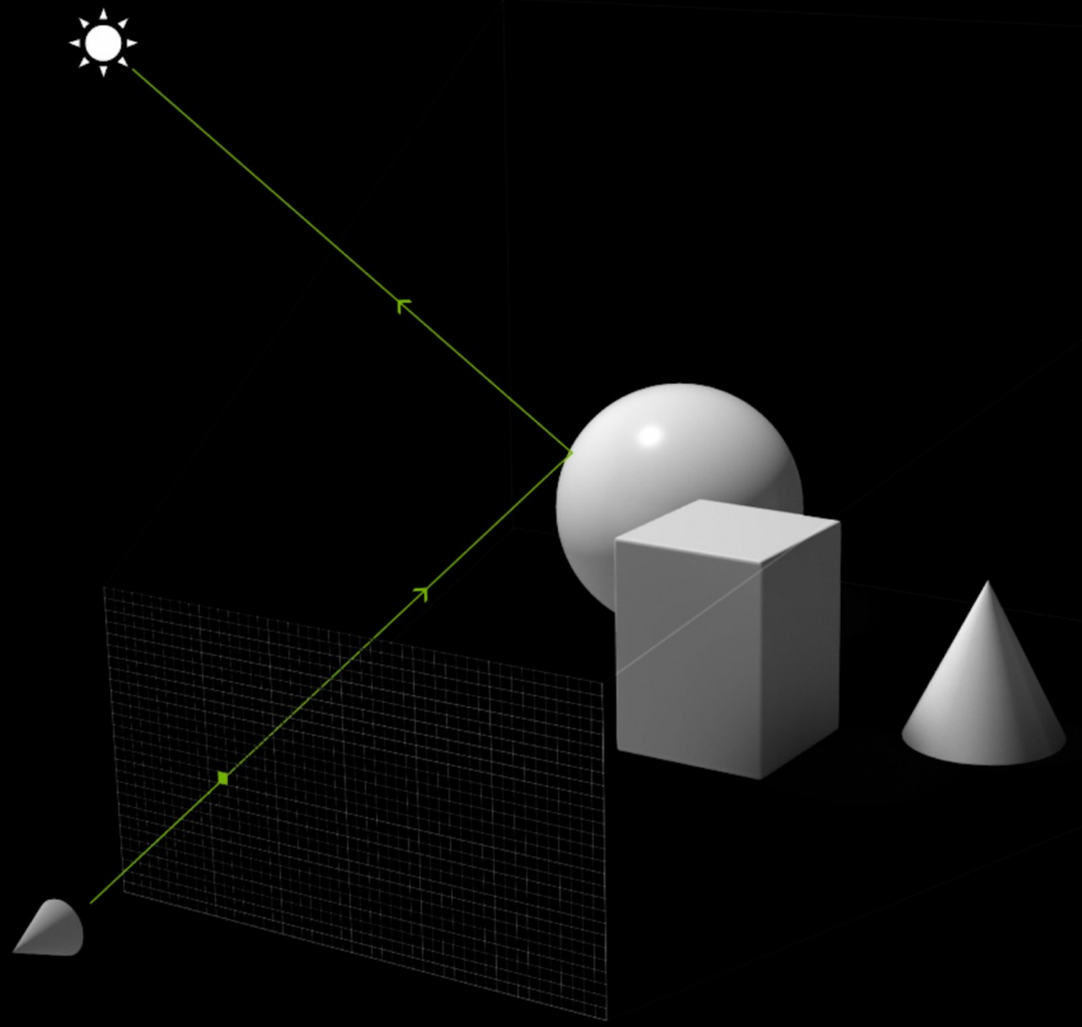
Not a Photo, a Calculation



http://on-demand.gputechconf.com/siggraph/2013/presentation/SG3106-Building-Ray-Tracing-Applications-OptiX.pdf



RASTERIZATION



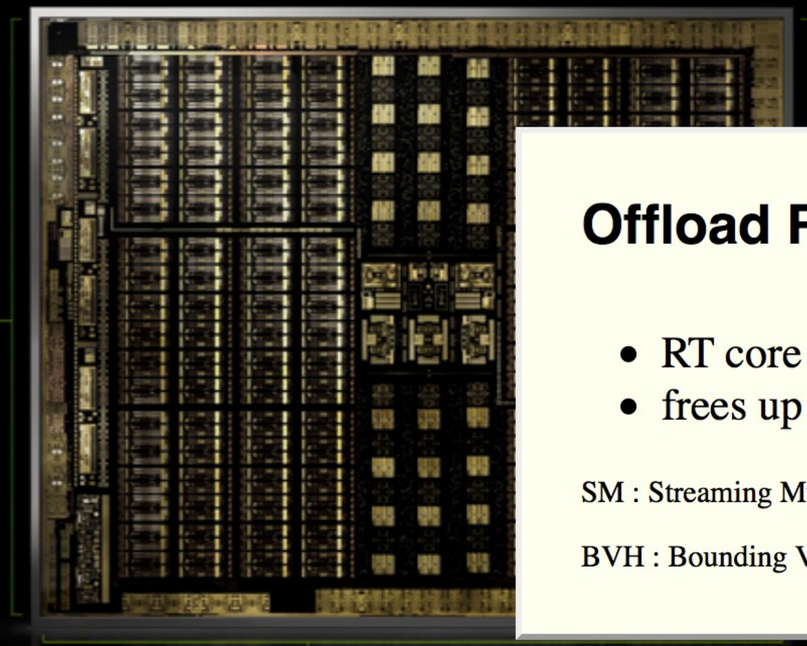
RAY TRACING

SIGGRAPH 2018

TURING BUILT FOR RTX

GREATEST LEAP SINCE 2006 CUDA GPU

Turing SM
14 TFLOPS + 14 TIPS
Concurrent FP & INT Execution
Variable Rate Shading



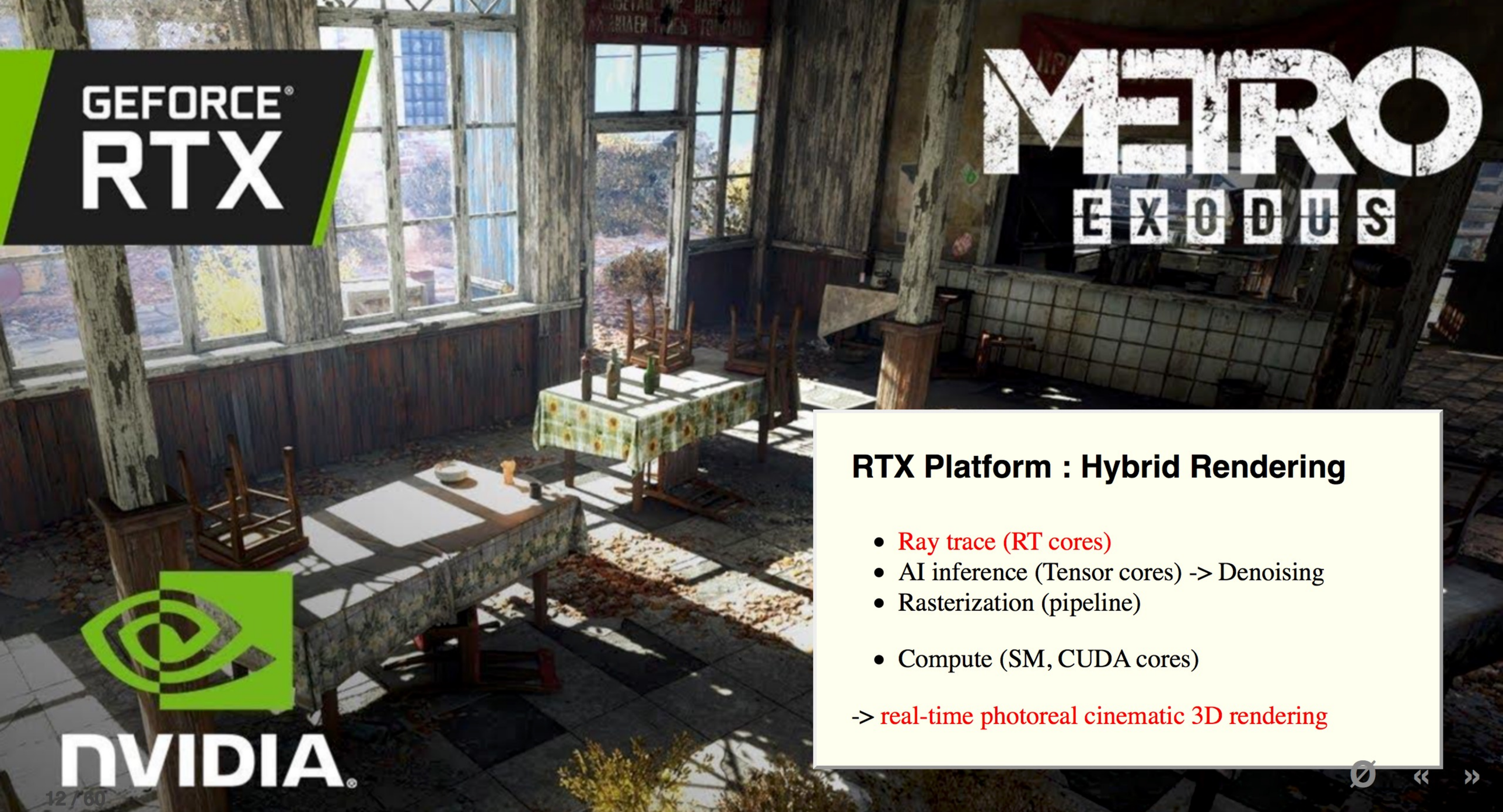
RT Core
10 Giga Rays/sec
Ray Triangle Intersection
BVH Traversal

Offload Ray Trace to Dedicated HW

- RT core : BVH traversal + ray tri. intersection
- frees up general purpose SM

SM : Streaming Multiprocessor

BVH : Bounding Volume Hierarchy



RTX Platform : Hybrid Rendering

- Ray trace (RT cores)
- AI inference (Tensor cores) -> Denoising
- Rasterization (pipeline)

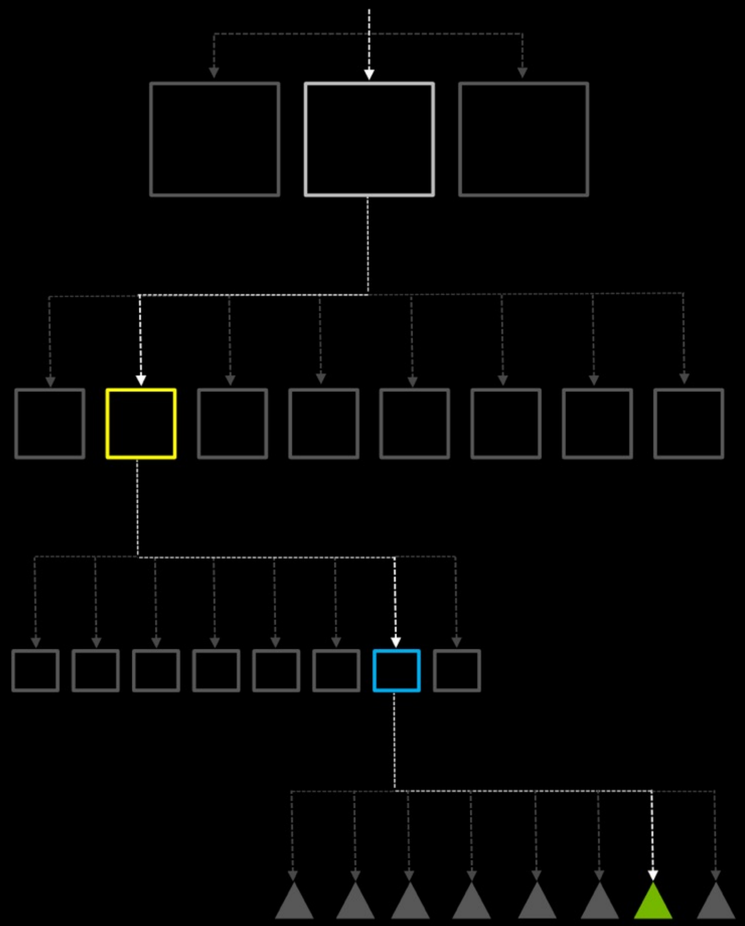
- Compute (SM, CUDA cores)

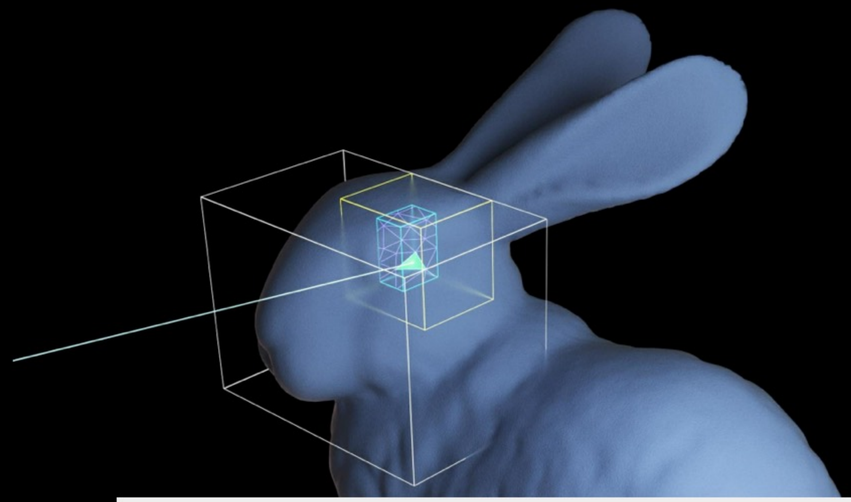
-> real-time photoreal cinematic 3D rendering

Spatial Index Acceleration Structure

BVH ALGORITHM

Massive Improvement in Search Efficiency





Tree of Bounding Boxes (bbox)

- aims to minimize bbox+primitive intersects
- accelerates ray-geometry intersection

NVIDIA® OptiX™ Ray Tracing Engine -- http://developer.nvidia.com/optix

OptiX makes GPU ray tracing accessible

- **accelerates** ray-geometry intersections
- simple : single-ray programming model
- "...free to use within any application..."
- access RT Cores[1] with OptiX 6.0.0+ via RTX™ mode

NVIDIA expertise:

- ~linear scaling up to 4 GPUs
- acceleration structure creation + traversal (Blue)
- instanced sharing of geometry + acceleration structures
- compiler optimized for GPU ray tracing

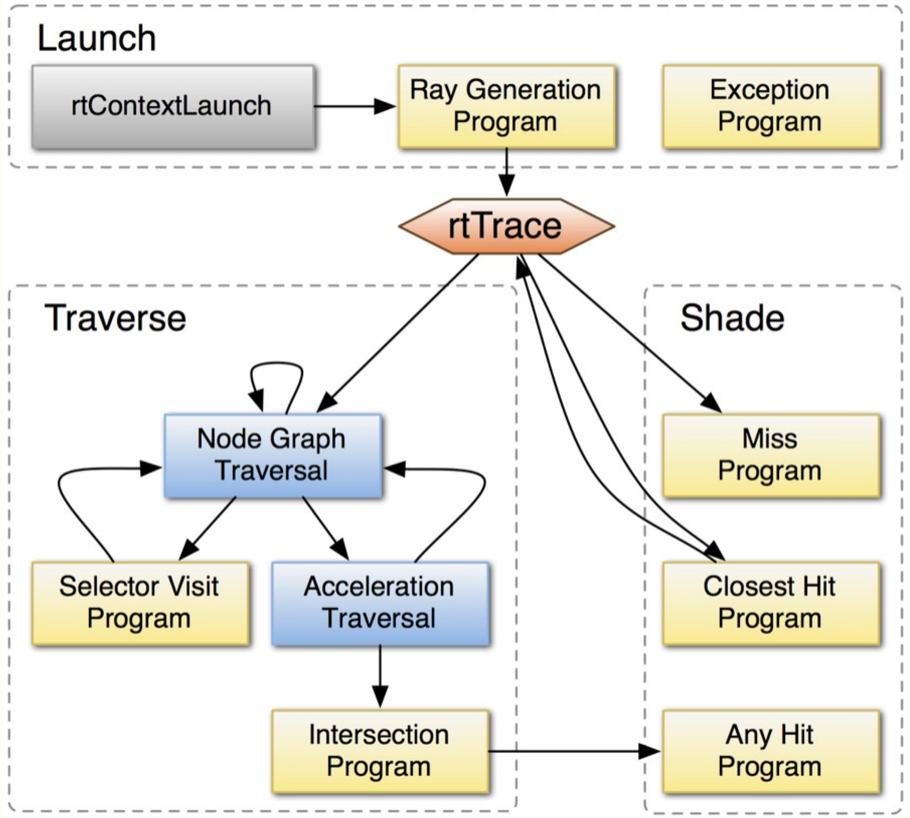
Opticks provides (Yellow):

- ray generation program
- ray geometry intersection+bbox programs

[1] Turing RTX GPUs

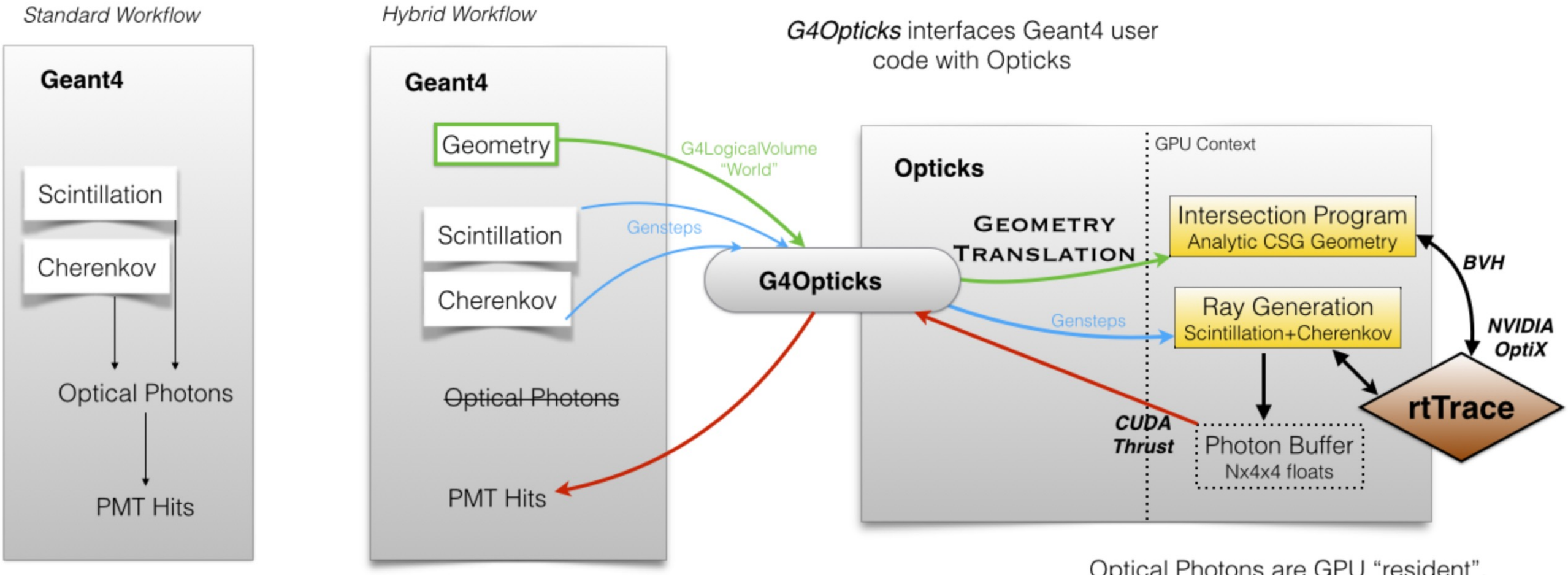
OptiX Raytracing Pipeline

Analogous to OpenGL rasterization pipeline:



Geant4 + Opticks Hybrid Workflow : External Optical Photon Simulation

https://bitbucket.org/simoncblyth/opticks



Optical Photons are GPU "resident", only hits are copied to CPU memory

Opticks : Translates G4 Optical Physics to CUDA/OptiX

OptiX : single-ray programming model -> line-by-line translation

CUDA Ports of Geant4 classes

- G4Cerenkov (only generation loop)
- G4Scintillation (only generation loop)
- G4OpAbsorption
- G4OpRayleigh
- G4OpBoundaryProcess (only a few surface types)

Modify Cherenkov + Scintillation Processes

- collect *genstep*, copy to GPU for generation
- avoids copying millions of photons to GPU

Scintillator Reemission

- fraction of bulk absorbed "reborn" within same thread
- wavelength generated by reemission texture lookup

Opticks (OptiX/Thrust GPU interoperation)

- **OptiX** : upload gensteps
- **Thrust** : seeding, distribute genstep indices to photons
- **OptiX** : launch photon generation and propagation
- **Thrust** : pullback photons that hit PMTs
- **Thrust** : index photon step sequences (optional)

GPU Resident Photons

Seeded on GPU

associate photons -> *gensteps* (via seed buffer)

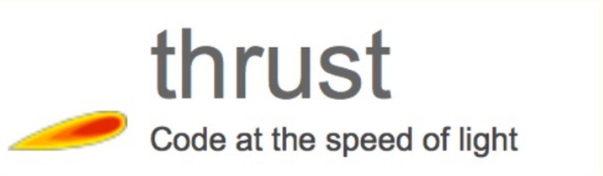
Generated on GPU, using genstep param:

- number of photons to generate
- start/end position of step

Propagated on GPU

Only photons hitting PMTs copied to CPU

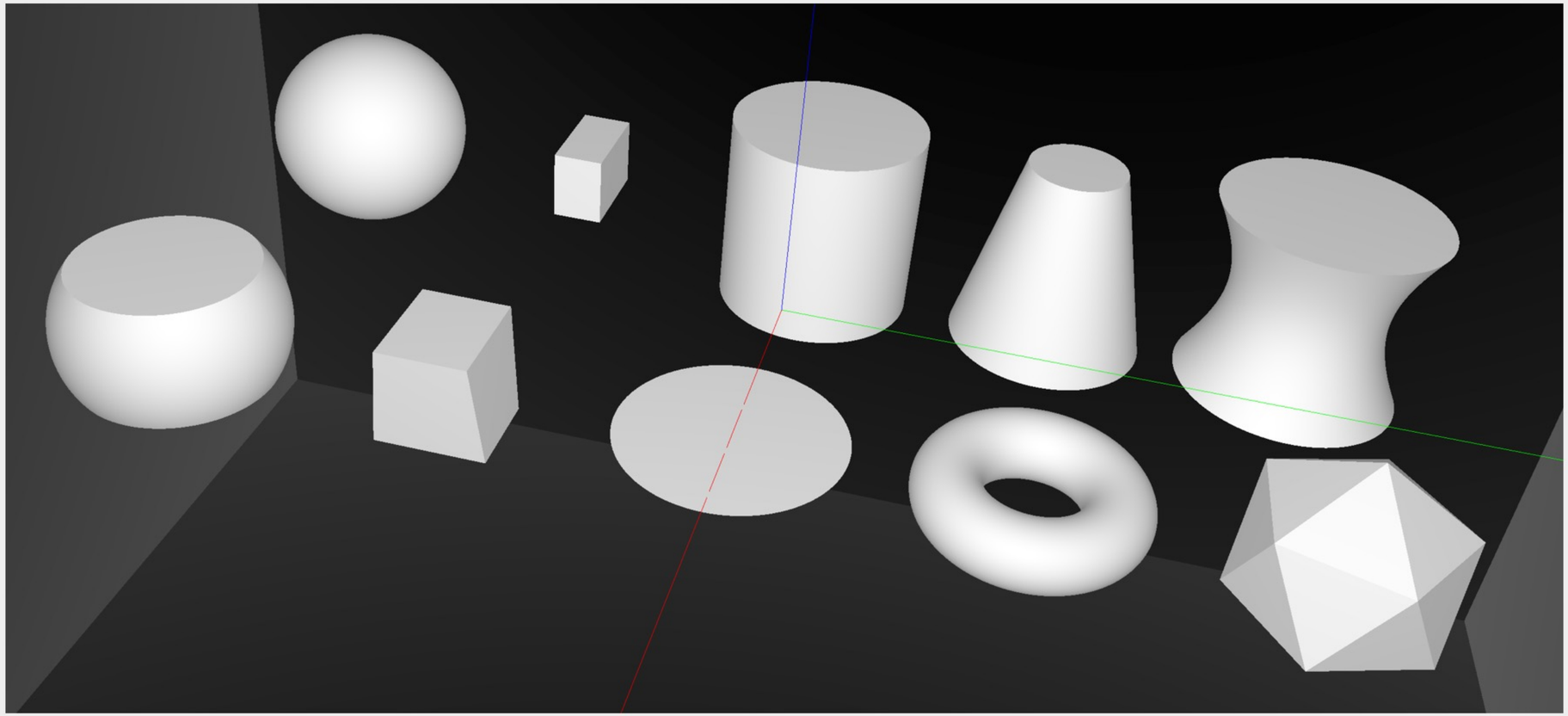
Thrust: **high level C++ access to CUDA**



- https://developer.nvidia.com/Thrust ❑

G4Solid -> CUDA Intersect Functions for ~10 Primitives

- 3D parametric ray : **ray(x,y,z;t) = rayOrigin + t * rayDirection**
- implicit equation of primitive : **f(x,y,z) = 0**
- -> polynomial in **t** , roots: **t > t_min** -> intersection positions + surface normals



Sphere, Cylinder, Disc, Cone, Convex Polyhedron, Hyperboloid, Torus, ...

G4Boolean -> CUDA/OptiX Intersection Program Implementing CSG

Complete Binary Tree, pick between pairs of nearest intersects:

UNION tA < tB	Enter B	Exit B	Miss B
Enter A	ReturnA	LoopA	ReturnA
Exit A	ReturnA	ReturnB	ReturnA
Miss A	ReturnB	ReturnB	ReturnMiss

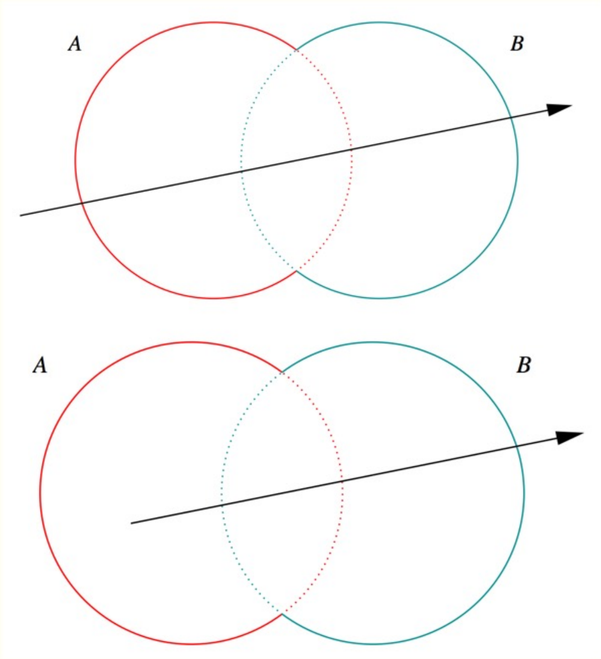
- *Nearest hit intersect algorithm* [1] avoids state
  - sometimes Loop : advance **t_min** , re-intersect both
  - classification shows if inside/outside
- *Evaluative* [2] implementation emulates recursion:
  - recursion not allowed in OptiX intersect programs
  - bit twiddle traversal of complete binary tree
  - stacks of postorder slices and intersects
- Identical geometry to Geant4
  - solving the same polynomials
  - near perfect intersection match

[1] Ray Tracing CSG Objects Using Single Hit Intersections, Andrew Kensler (2006)
with corrections by author of XRT Raytracer http://xrt.wikidot.com/doc:csg

[2] https://bitbucket.org/simoncblyth/opticks/src/tip/optixrap/cu/csg_intersect_boolean.h
Similar to binary expression tree evaluation using postorder traverse.

Outside/Inside Unions

dot(normal,rayDir) -> Enter/Exit



- **A + B** boundary not inside other
- **A * B** boundary inside other

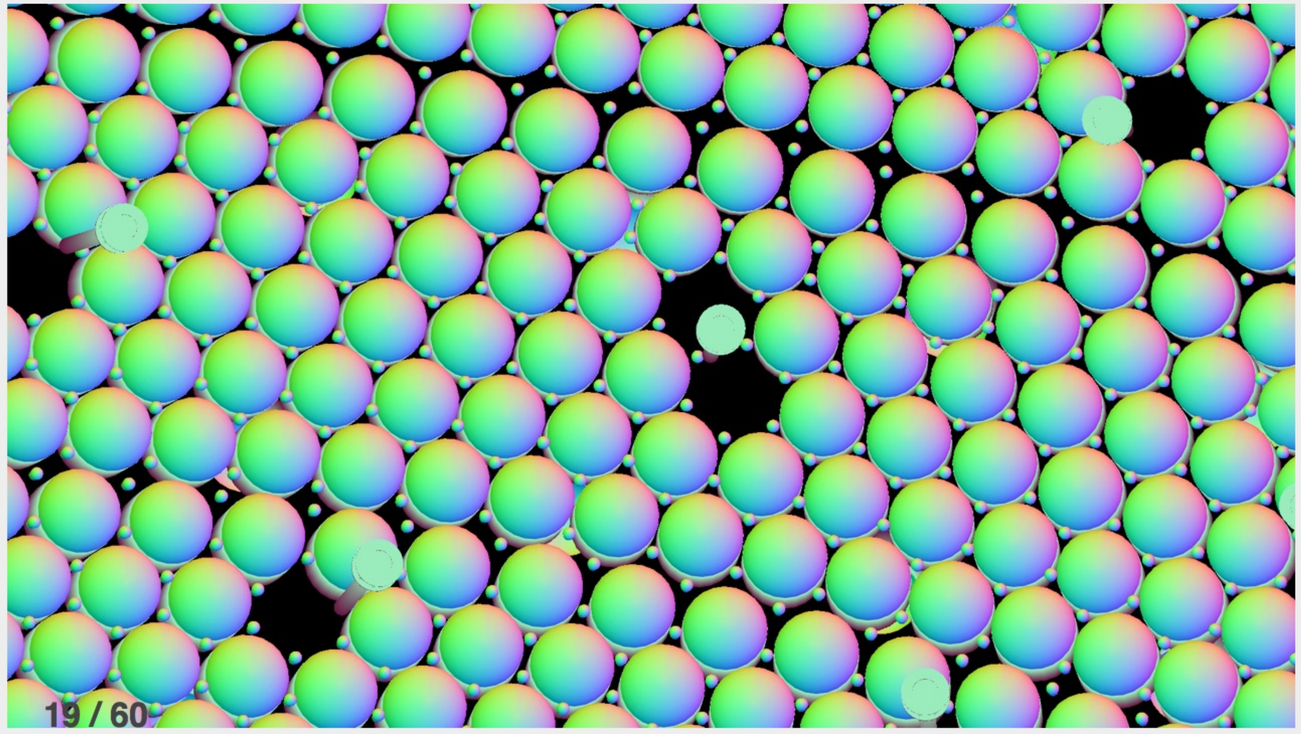
Opticks : Translates G4 Geometry to GPU, Without Approximation

G4 Structure Tree -> Instance+Global Arrays -> OptiX

Group structure into repeated instances + global remainder:

- auto-identify repeated geometry with "progeny digests"
  - JUNO : 5 distinct instances + 1 global
- instance transforms used in OptiX/OpenGL geometry

instancing -> huge memory savings for JUNO PMTs



Materials/Surfaces -> GPU Texture

Material/Surface/Scintillator properties

- interpolated to standard wavelength domain
- interleaved into "boundary" texture
- "reemission" texture for wavelength generation

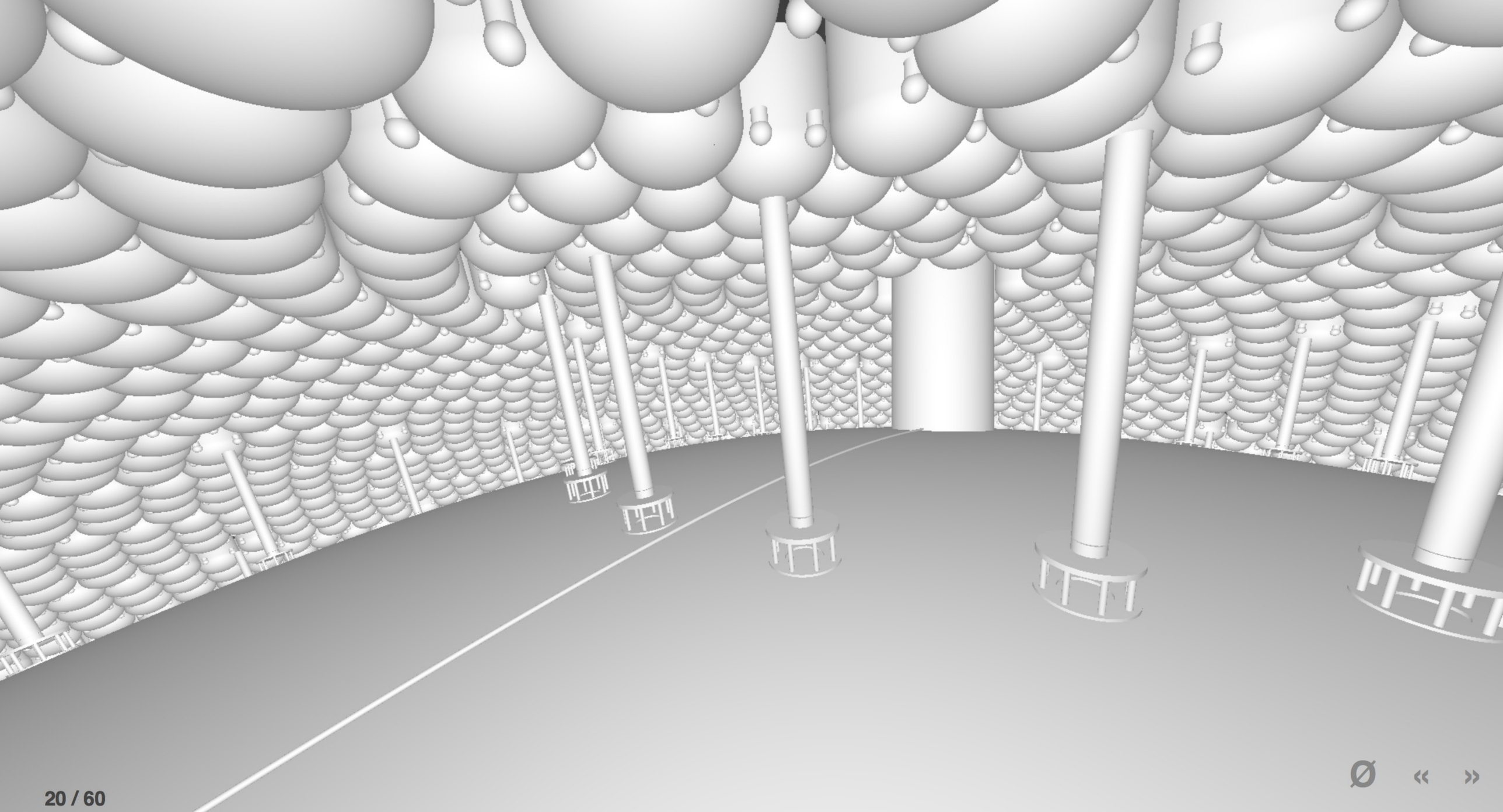
Material/surface boundary : 4 indices

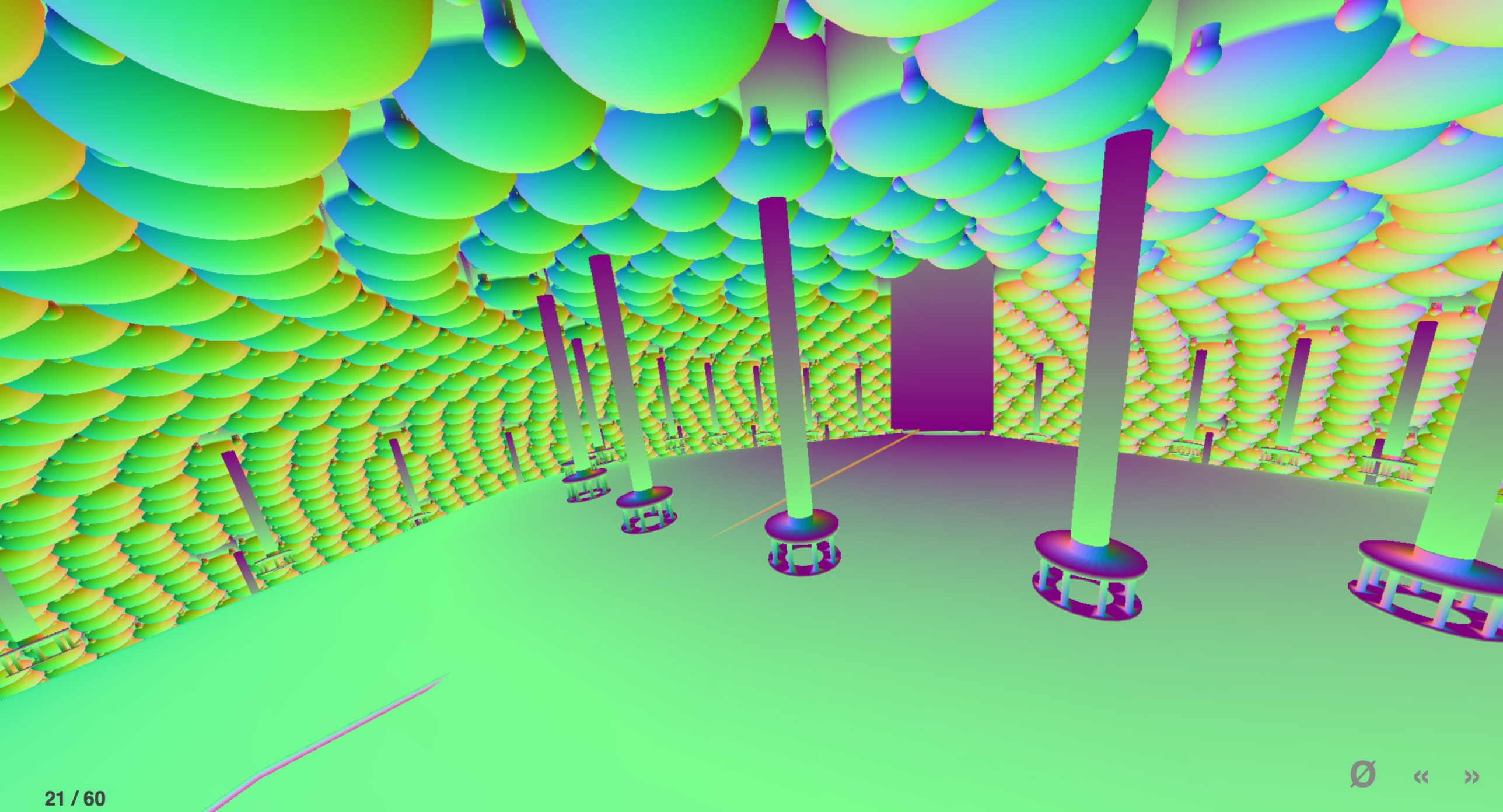
- outer material (parent)
- outer surface (inward photons, parent -> self)
- inner surface (outward photons, self -> parent)
- inner material (self)

Primitives labelled with unique boundary index

- ray primitive intersection -> boundary index
- texture lookup -> material/surface properties

simple/fast properties + reemission wavelength





Validation of Opticks Simulation by Comparison with Geant4

Bi-simulations of all JUNO solids, with millions of photons

mis-aligned histories
    mostly < 0.25%, < 0.50% for largest solids

deviant photons within matched history
    < 0.05% (500/1M)

Primary sources of problems

- grazing incidence, edge skimmers
- incidence at constituent solid boundaries

Primary cause : float vs double

Geant4 uses *double* everywhere, *Opticks* only sparingly (observed *double* costing 10x slowdown with RTX)

Conclude

- neatly oriented photons more prone to issues than realistic ones
- perfect "technical" matching not feasible
- instead shift validation to more realistic full detector "calibration" situation

Random Aligned Bi-Simulation

Same inputs to *Opticks* and *Geant4*:

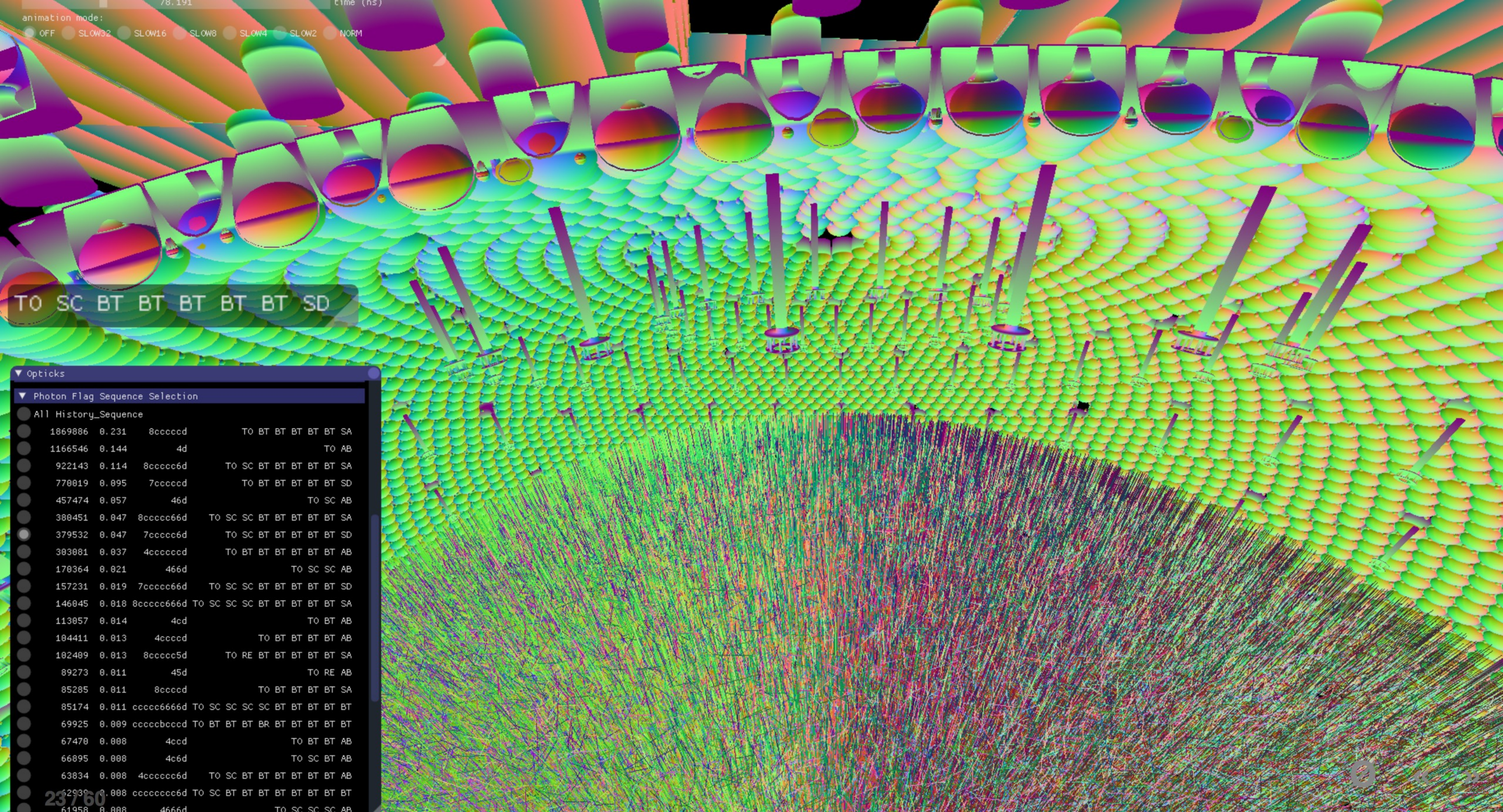
- CPU generated photons
- GPU generated randoms, fed to *Geant4*

Common recording into *OpticksEvents*:

- compressed photon step record, up to 16 steps
- persisted as *NumPy* arrays for python analysis

Aligned random consumption, direct comparison:

- ~every **scatter, absorb, reflect, transmit** at matched positions, times, polarization, wavlen


78.191
time (ns)
animation mode:
OFF
SLOW32
SLOW16
SLOW8
SLOW4
SLOW2
NORM
TO SC BT BT BT BT BT SD
Opticks
Photon Flag Sequence Selection
All History_Sequence
1869886 0.231 8ccccd TO BT BT BT BT BT SA
1166546 0.144 4d TO AB
922143 0.114 8ccccc6d TO SC BT BT BT BT BT SA
770019 0.095 7ccccd TO BT BT BT BT BT SD
457474 0.057 46d TO SC AB
380451 0.047 8ccccc66d TO SC SC BT BT BT BT BT SA
379532 0.047 7ccccc6d TO SC BT BT BT BT BT SD
303081 0.037 4cccccd TO BT BT BT BT BT BT AB
170364 0.021 466d TO SC SC AB
157231 0.019 7ccccc66d TO SC SC BT BT BT BT BT SD
146045 0.018 8ccccc666d TO SC SC SC BT BT BT BT BT SA
113057 0.014 4cd TO BT AB
104411 0.013 4ccccd TO BT BT BT BT AB
102409 0.013 8ccccc5d TO RE BT BT BT BT BT SA
89273 0.011 45d TO RE AB
85285 0.011 8ccccd TO BT BT BT BT BT SA
85174 0.011 ccccc6666d TO SC SC SC SC BT BT BT BT BT
69925 0.009 cccccbcccd TO BT BT BT BR BT BT BT BT BT
67470 0.008 4ccd TO BT BT AB
66895 0.008 4c6d TO SC BT AB
63834 0.008 4ccccccd TO SC BT BT BT BT BT BT AB
61958 0.008 4666d TO SC SC SC AB
23760

Recording the steps of Millions of Photons

Up to 16 steps of the photon propagation are recorded.

Photon Array : 4 * *float4* = 512 bits/photon

- *float4*: position, time [32 * 4 = 128 bits]
- *float4*: direction, weight
- *float4*: polarization, wavelength
- *float4*: flags: material, boundary, history

Step Record Array : 2 * *short4* = 2*16*4 = 128 bits/record

- *short4*: position, time (snorm compressed) [4*16 = 64 bits]
- *uchar4*: polarization, wavelength (uchar compressed) [4*8 = 32 bits]
- *uchar4*: material, history flags [4*8 = 32 bits]

Compression uses known domains of position (geometry center, extent), time (0:200ns), wavelength, polarization.

Compression Essential

Domain compression to fit in VRAM

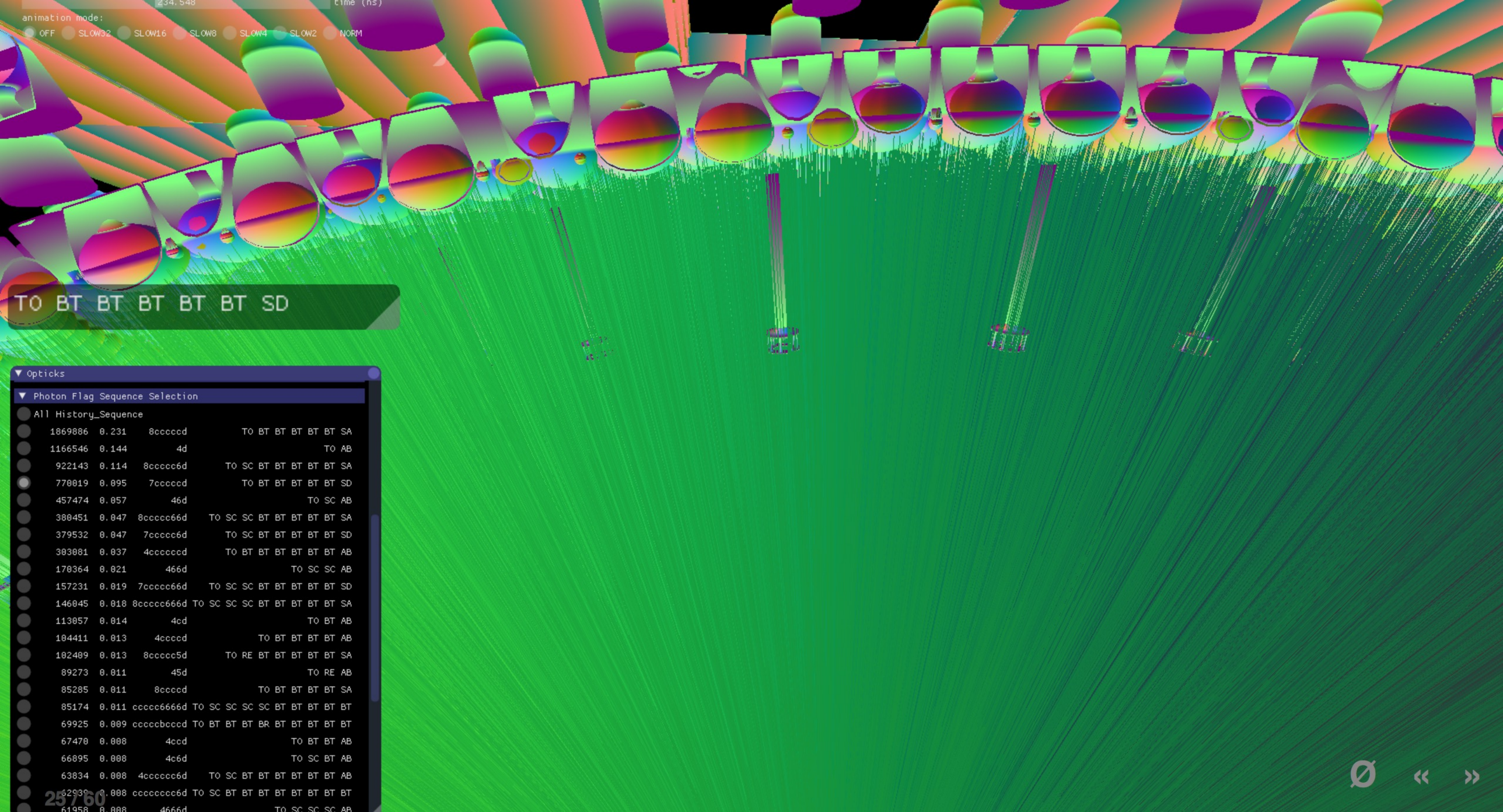
- 16 step records per photon -> 256 bytes/photon
- 10M photons -> 2.56 GB

4-bit History Flags at Each Step

```
BT : boundary
BR : boundary reflect
SC : bulk scatter
AB : bulk absorb
SD : surface detect
SA : surface absorb
```

seqhis

64-bit integer history sequence


234.548
time (ns)
animation mode:
OFF
SLOW32
SLOW16
SLOW8
SLOW4
SLOW2
NORM
TO BT BT BT BT BT SD
Opticks
Photon Flag Sequence Selection
All History_Sequence
1869886 0.231 8ccccd TO BT BT BT BT BT SA
1166546 0.144 4d TO AB
922143 0.114 8ccccc6d TO SC BT BT BT BT BT SA
770019 0.095 7ccccd TO BT BT BT BT BT SD
457474 0.057 46d TO SC AB
380451 0.047 8ccccc66d TO SC SC BT BT BT BT BT SA
379532 0.047 7ccccc6d TO SC BT BT BT BT BT SD
303081 0.037 4cccccd TO BT BT BT BT BT BT AB
170364 0.021 466d TO SC SC AB
157231 0.019 7ccccc66d TO SC SC BT BT BT BT BT SD
146045 0.018 8ccccc666d TO SC SC SC BT BT BT BT BT SA
113057 0.014 4cd TO BT AB
104411 0.013 4ccccd TO BT BT BT BT AB
102409 0.013 8ccccc5d TO RE BT BT BT BT BT SA
89273 0.011 45d TO RE AB
85285 0.011 8ccccd TO BT BT BT BT SA
85174 0.011 ccccc6666d TO SC SC SC SC BT BT BT BT BT
69925 0.009 cccccbcccd TO BT BT BT BR BT BT BT BT BT
67470 0.008 4ccd TO BT BT AB
66895 0.008 4c6d TO SC BT AB
63834 0.008 4ccccccc6d TO SC BT BT BT BT BT BT BT AB
61958 0.008 4666d TO SC SC SC AB
257760

Performance : Scanning from 1M to 400M Photons

Full JUNO Analytic Geometry j1808v5

- "calibration source" genstep at center of scintillator

Production Mode : does the minimum

- only saves hits
- skips : genstep, photon, source, record, sequence, index, ..
- no *Geant4* propagation (other than at 1M for extrapolation)

Multi-Event Running, Measure:

interval
: avg time between successive launches, including overheads: (upload gensteps + launch + download hits)

launch
: avg of 10 OptiX launches

- overheads < 10% beyond 20M photons

Test Hardware + Software

Workstation

- DELL Precision 7920T Workstation
- Intel Xeon Gold 5118, 2.3GHz, 48 cores, 62G
- NVIDIA Quadro RTX 8000 (48G)

Software

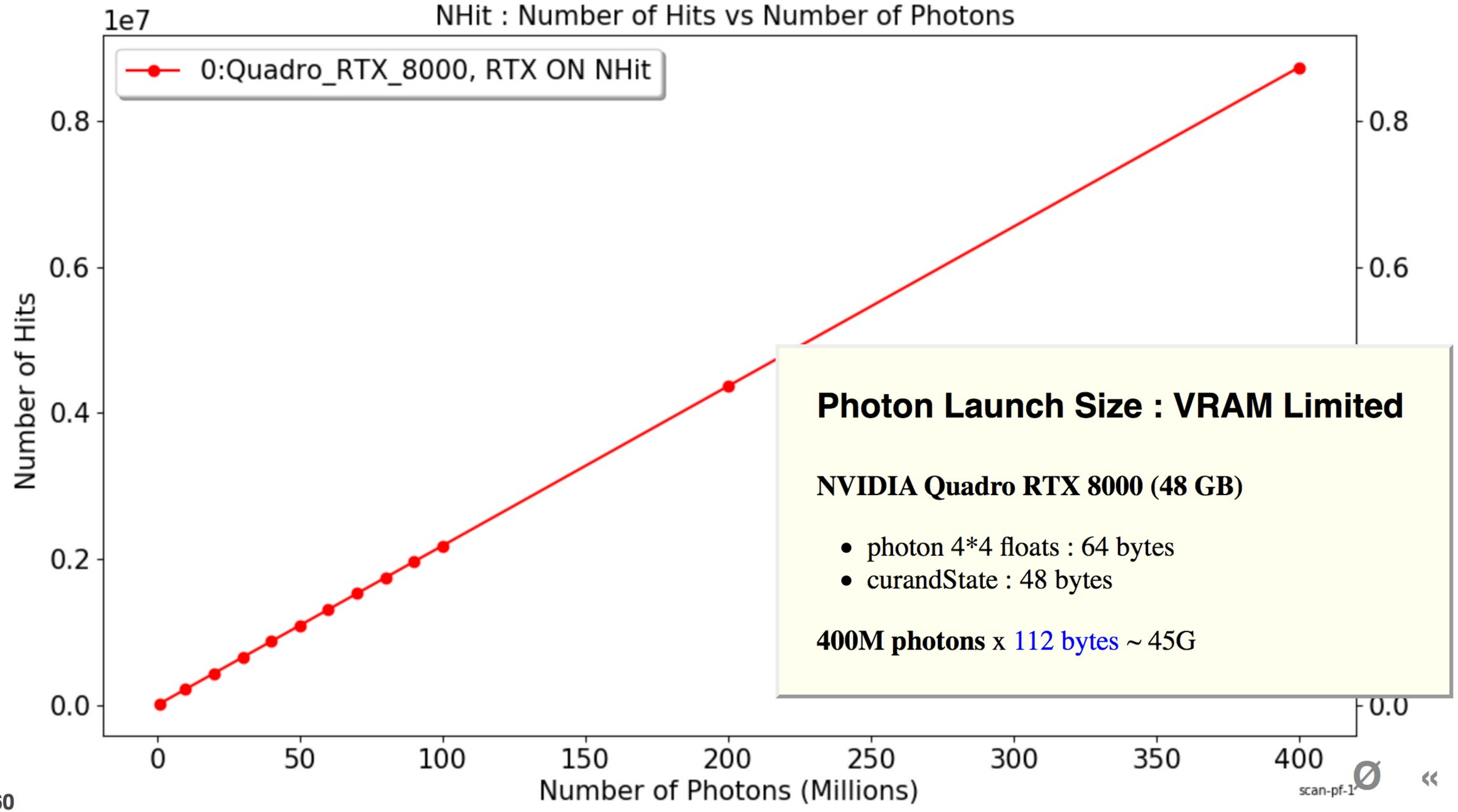
- Opticks 0.0.0 Alpha
- Geant4 10.4p2
- NVIDIA OptiX 6.5.0
- NVIDIA Driver 435.21
- CUDA 10.1

IHEP GPU Cluster

- 10 nodes of 8x NVIDIA Tesla GV100 (32G)

NVIDIA Quadro RTX 8000 (48G)

谢谢 NVIDIA China
for loaning the card

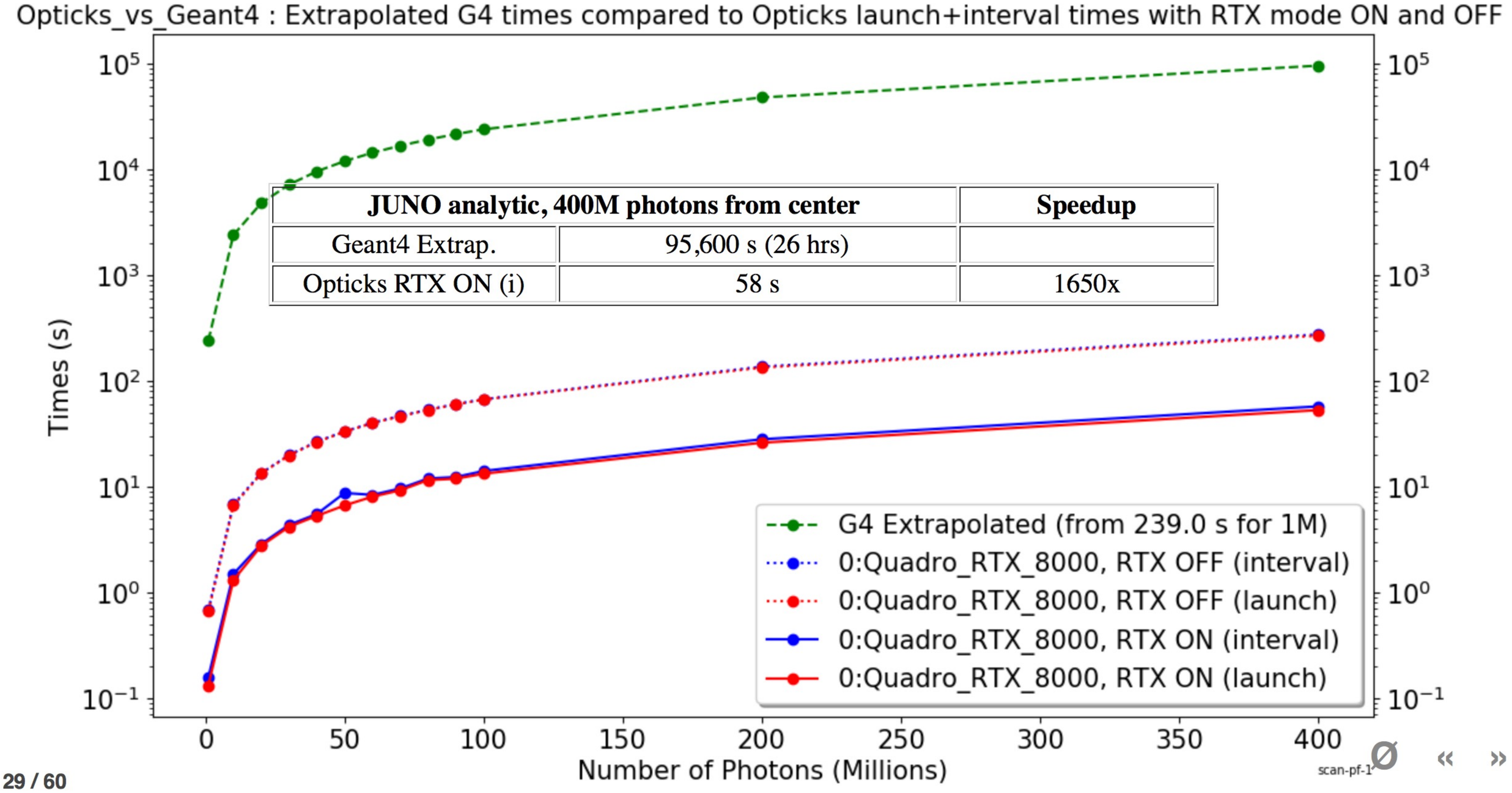


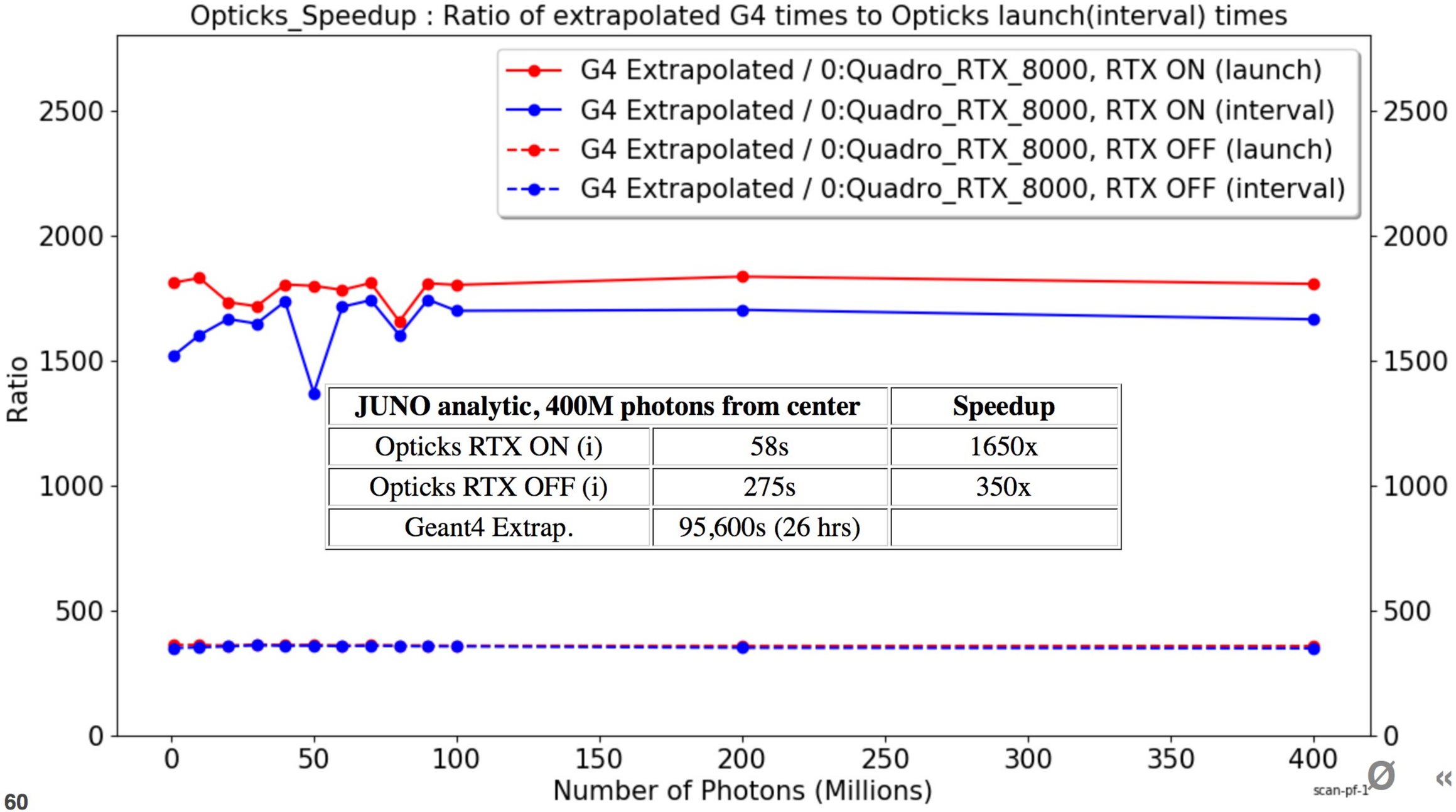
Photon Launch Size : VRAM Limited

NVIDIA Quadro RTX 8000 (48 GB)

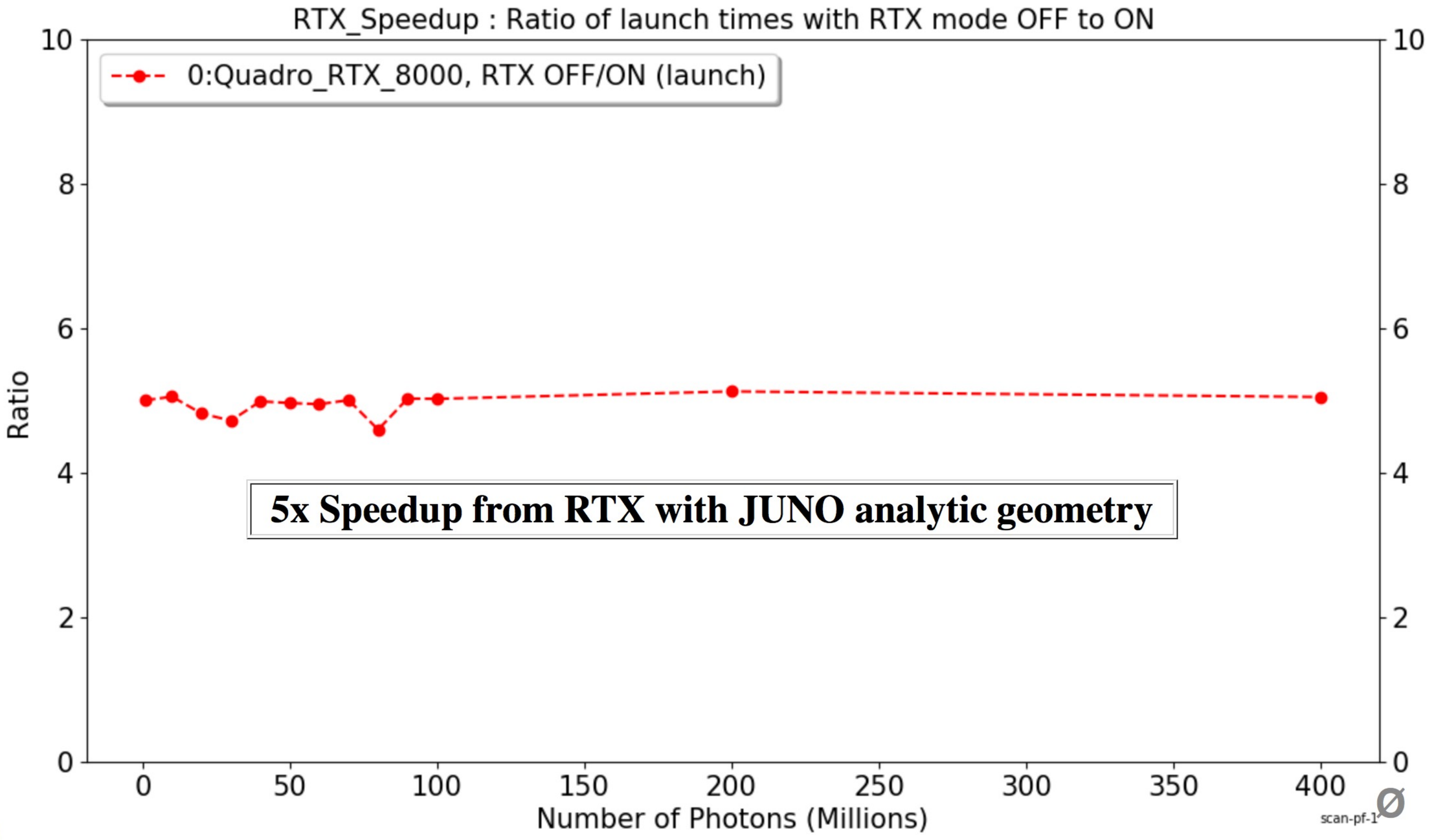
- photon 4*4 floats : 64 bytes
- curandState : 48 bytes

400M photons x 112 bytes ~ 45G


Opticks_vs_Geant4 : Extrapolated G4 times compared to Opticks launch+interval times with RTX mode ON and OFF
JUNO analytic, 400M photons from center
Speedup
Geant4 Extrap.
95,600 s (26 hrs)
Opticks RTX ON (i)
58 s
1650x
Times (s)
Number of Photons (Millions)
G4 Extrapolated (from 239.0 s for 1M)
0:Quadro_RTX_8000, RTX OFF (interval)
0:Quadro_RTX_8000, RTX OFF (launch)
0:Quadro_RTX_8000, RTX ON (interval)
0:Quadro_RTX_8000, RTX ON (launch)

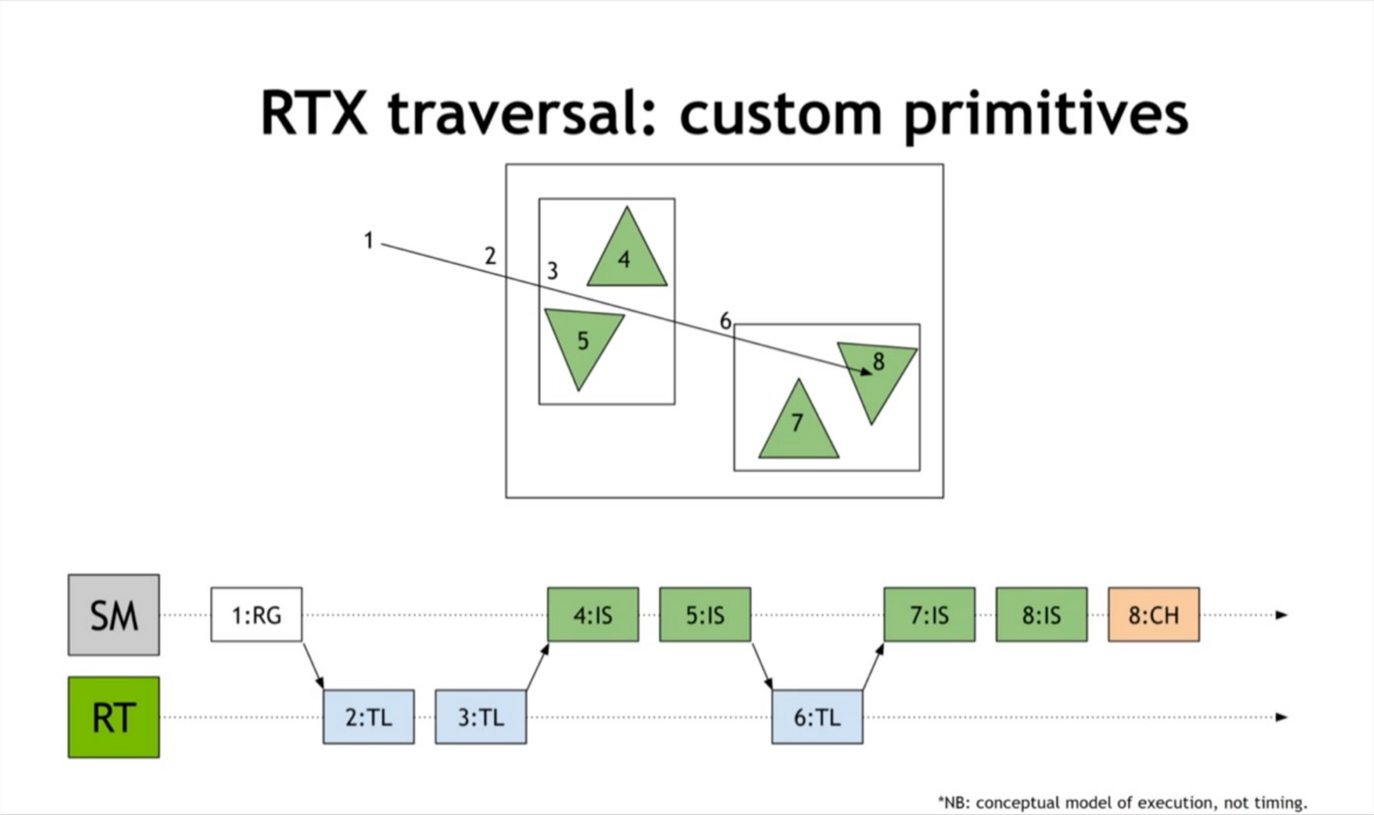


JUNO analytic, 400M photons from center		Speedup
Opticks RTX ON (i)	58s	1650x
Opticks RTX OFF (i)	275s	350x
Geant4 Extrap.	95,600s (26 hrs)	


RTX_Speedup : Ratio of launch times with RTX mode OFF to ON
0:Quadro_RTX_8000, RTX OFF/ON (launch)
Ratio
Number of Photons (Millions)
0
2
4
6
8
10
0
50
100
150
200
250
300
350
400
5x Speedup from RTX with JUNO analytic geometry

Useful Speedup > 1000x : But Why Not Giga Rays/s ? (1 Photon ~10 Rays)

- NVIDIA claim : 10 Giga Rays/s with RT Core
- -> **1 Billion photons per second**
- **RT cores : built-in triangle intersect + 1-level of instancing**
- flatten scene model to avoid SM<->RT roundtrips ?



OptiX Performance Tools and Tricks, David Hart, NVIDIA
https://developer.nvidia.com/siggraph/2019/video/sig915-vid ❏

100M photon RTX times, avg of 10

Launch times for various geometries			
Geometry	**Launch (s)**	**Giga Rays/s**	**Relative to ana**
JUNO ana	13.2	0.07	
JUNO tri.sw	6.9	0.14	1.9x
JUNO tri.hw	2.2	0.45	6.0x
Boxtest ana	0.59	1.7	
Boxtest tri.sw	0.62	1.6	
Boxtest tri.hw	0.30	3.3	1.9x

- ana : Opticks analytic CSG (SM)
- tri.sw : software triangle intersect (SM)
- tri.hw : hardware triangle intersect (RT)

JUNO 15k triangles, 132M without instancing

Simple Boxtest geometry gets into ballpark ∅

Where Next for Opticks ?

JUNO+Opticks into Production

- optimize geometry modelling for RTX
- full JUNO geometry validation iteration
- JUNO offline integration
- optimize GPU cluster throughput:
  - split/join events to fit VRAM
  - job/node/multi-GPU strategy
- support OptiX 7, find multi-GPU load balancing approach

Geant4+Opticks Integration : Work with Geant4 Collaboration

- finalize *Geant4+Opticks* extended example
  - aiming for *Geant4* distrib
- prototype *Genstep* interface inside *Geant4*
  - avoid customizing *G4Cerenkov G4Scintillation*

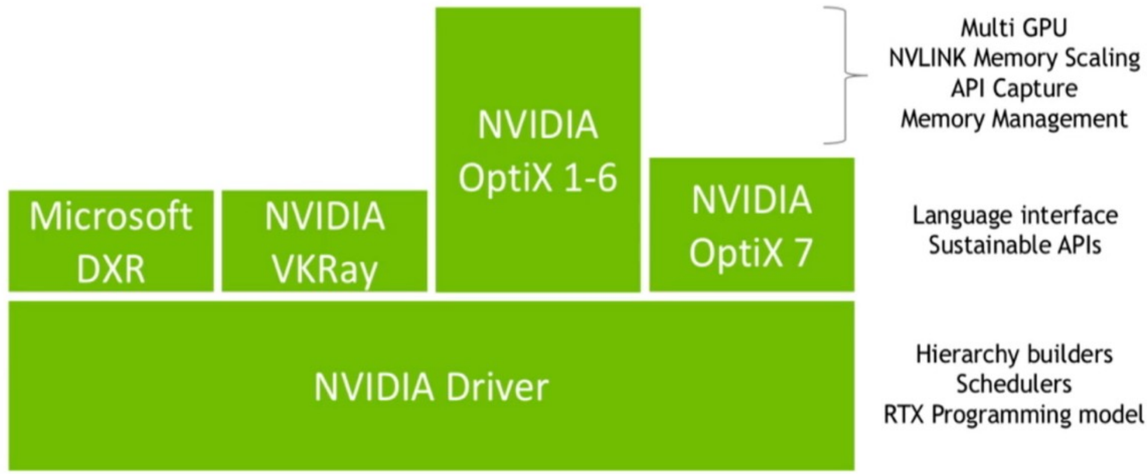
Alpha Development ------>----------> Robust Tool

- many more users+developers required (current ~10+1)
- if you have an optical photon simulation problem ...
  - start by joining : https://groups.io/g/opticks ❑

NVIDIA OptiX 7 : Entirely new API

- introduced August 2019
- low-level CUDA-centric thin API
- ~~near perfect scaling to 4 GPUs, for free~~

Introducing OptiX 7



Drastically Improved Optical Photon Simulation Performance...

Three revolutions reinforcing each other:

- games -> graphics revolution -> GPU -> cheap TFLOPS
- internet scale big datasets -> ML revolution
- computer vision revolution for autonomous vehicles

Deep rivers of development, ripe for re-purposing

- analogous problems -> solutions
- experience across fields essential to find+act on analogies

Example : DL denoising for faster ray trace convergence

- analogous to hit aggregation
- skip the hits, jump straight to DL smoothed probabilities
  - blurs the line between simulation and reconstruction

Re-evaluate long held practices in light of new realities:

- large ROOT format (C++ object) MC samples repeatedly converted+uploaded to GPU for DL training ... OR:
- small Genstep NumPy arrays uploaded, dynamically simulated into GPU hit arrays in fractions of a second

How is >1000x possible ?

Progress over 30 yrs, Billions of Dollars

- industry funded : game, film, design, ...
- re-purposed by translating geometry to GPU
  - tree of C++ objects -> arrays -> BVH

Photon Simulation ideally suited to GPU

- millions of photons -> abundantly parallel
- simple phys. -> small stack -> many in flight
- decoupled -> no synchronization

Dynamically generated simulation feasible ?

- current reconstruction -> custom simulation
- no more : limited MC stats in edge cases

Summary

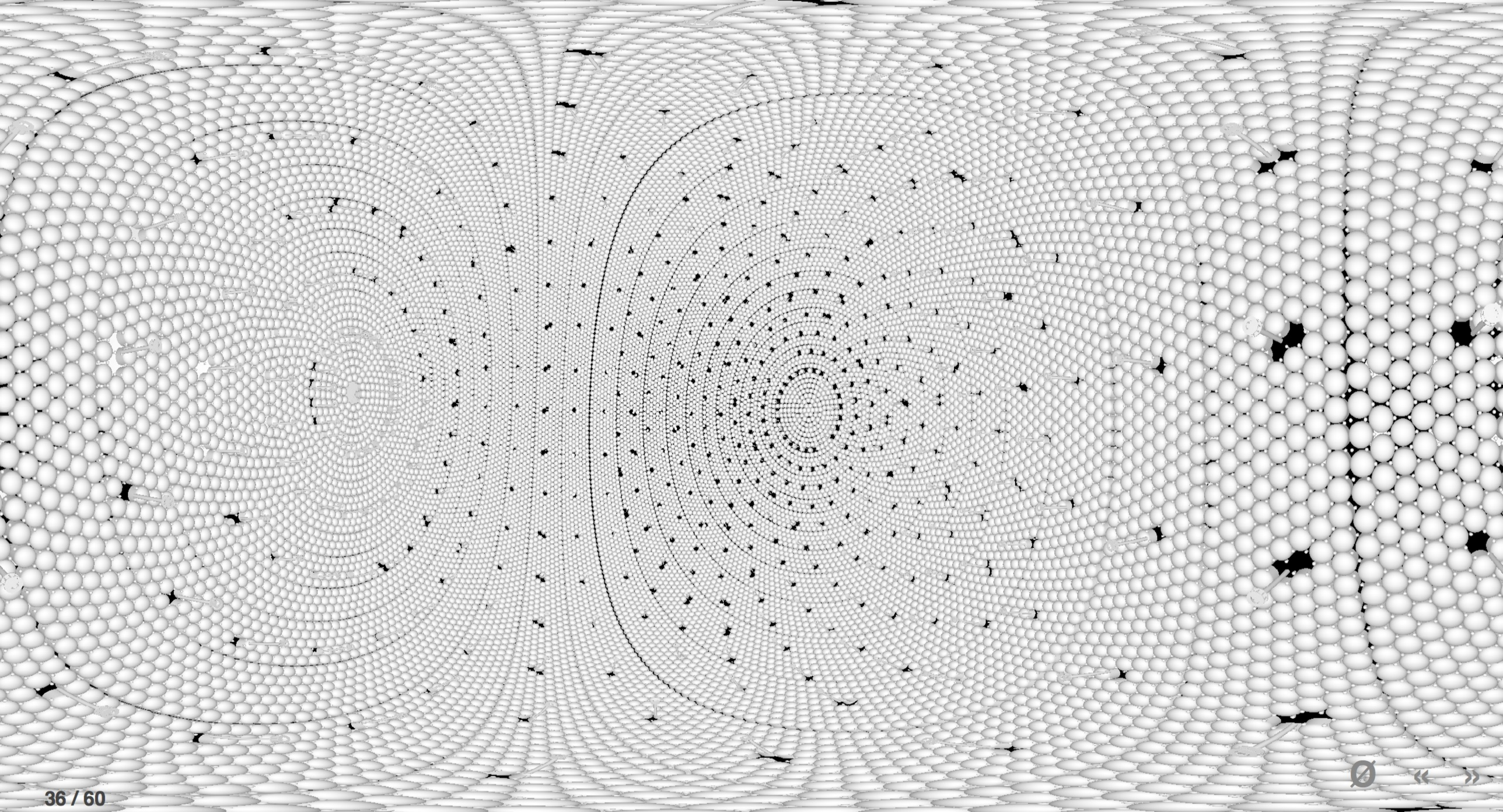
Opticks : state-of-the-art GPU ray tracing applied to optical photon simulation and integrated with *Geant4*, giving a leap in performance that eliminates memory and time bottlenecks.

Highlights 2019

- Benefit from hardware accelerated ray tracing
- **Opticks > 1000x Geant4** (one Turing GPU)

- Drastic speedup -> better detector understanding -> greater precision
  - **any simulation limited by optical photons can benefit**
  - more photon limited -> more overall speedup (99% -> 100x)

https://bitbucket.org/simoncblyth/opticks ❏	code repository
https://simoncblyth.bitbucket.io ❏	presentations and videos
https://groups.io/g/opticks ❏	forum/mailing list archive
email:opticks+subscribe@groups.io	subscribe to mailing list



Opticks : GPU Optical Simulation via NVIDIA® OptiX™
+ A Mental Model for Effective Application of GPUs

- GPU Mental Model : Context and Constraints
  - Understanding GPU Graphical Origins -> Effective GPU Computation
  - GPU Demands Simplicity (Arrays) -> Big Benefits : NumPy + CuPy
- Parallel Processing 0th Step
  - Re-shape Data into "Parallel" Array Form
- High Level Tools
  - Survey of High Level General Purpose CUDA Packages
  - NumPy + CuPy
- Example 1 : NumPy/CuPy Python interface
  - Python vs NumPy vs CuPy : Python/NumPy ~ 10, NumPy/CuPy ~ 1300
- Array Mechanics
  - NP.hh header + union "trick"
- Example 2 : A Taste of Thrust C++
  - Photon History Indexing with CUDA Thrust
- Summary

Amdahls "Law" : Expected Speedup Limited by Serial Processing

optical photon simulation, P ~ 99% of CPU time

- -> potential overall speedup S(n) is 100x
- even with parallel speedup factor >> 1000x

Must consider processing "big picture"

- remove bottlenecks one by one
- re-evaluate "big picture" after each

S(n) Expected Speedup

$$S(n) = \frac{1}{(1-P) + P/n}$$

P
: parallelizable proportion

1-P
: non-parallelizable portion

n
: parallel speedup factor

Understanding GPU Graphical Origins -> Effective GPU Computation

GPUs evolved to rasterize 3D graphics at 30/60 fps

- 30/60 "launches" per second, each handling millions of items
- literally billions of small "shader" programs run per second

Simple Array Data Structures (N-million,4)

- millions of vertices, millions of triangles
- vertex: **(x y z w)**
- colors: **(r g b a)**

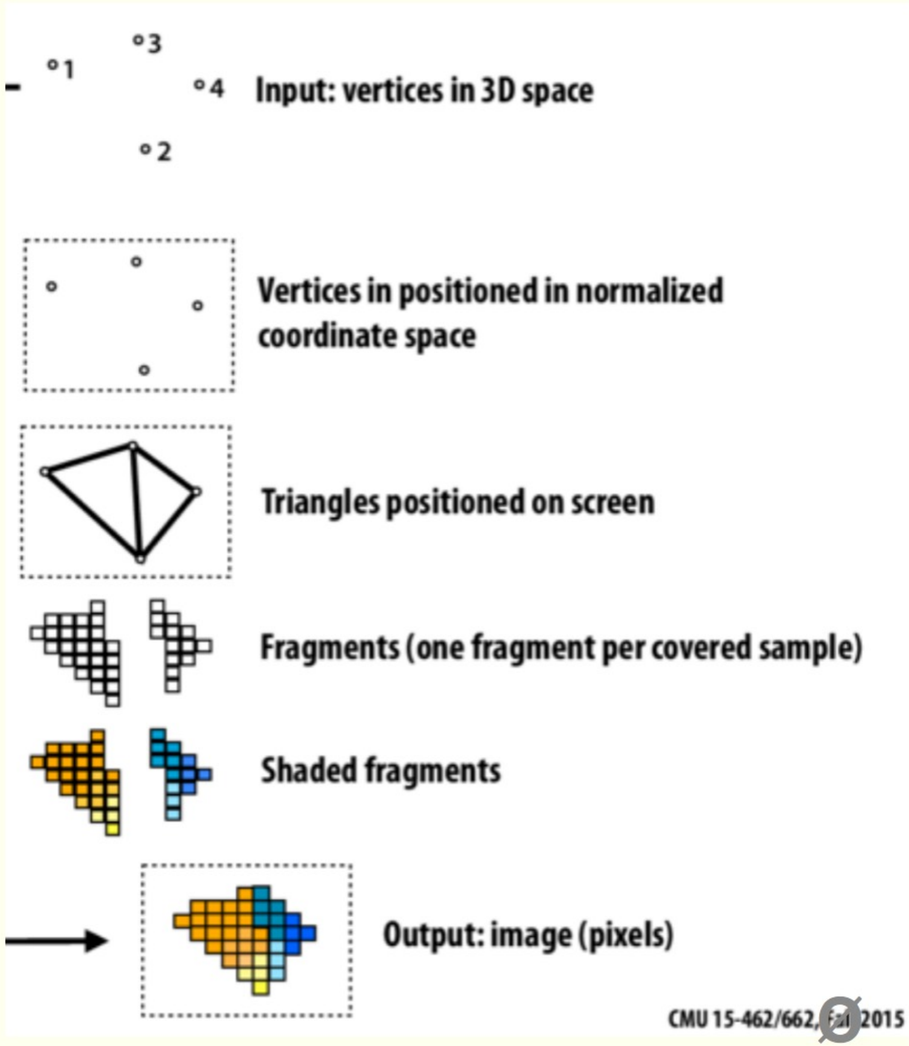
Constant "Uniform" 4x4 matrices : scaling+rotation+translation

- 4-component homogeneous coordinates -> easy projection

Graphical Experience Informs Fast Computation on GPUs

- array shapes similar to graphics ones are faster
  - "float4" 4*float(32bit) = 128 bit memory reads are favored
  - Opticks photons use "float4x4" just like 4x4 matrices
- GPU Launch frequency < ~30/60 per second
  - avoid copy+launch overheads becoming significant
  - ideally : handle millions of items in each launch

OpenGL Rasterization Pipeline



CPU Optimizes Latency, GPU Optimizes Throughput

Waiting for memory read/write, is major source of latency...

CPU : latency-oriented : Minimize time to complete single task : *avoid latency with caching*

- complex : caching system, branch prediction, speculative execution, ...

GPU : throughput-oriented : Maximize total work per unit time : *hide latency with parallelism*

- many simple processing cores, hardware multithreading, SIMD (single instruction multiple data)
- simpler : lots of compute (ALU), at expense of cache+control
- can tolerate latency, by **assuming** abundant other tasks to resume : **design assumes parallel workload**

Totally different processor architecture -> Total reorganization of data and computation

- major speedups typically require total rethink of data structures and computation

How to Make Effective Use of GPUs ? Parallel / Simple / Uncoupled

Abundant parallelism
- many thousands of tasks (ideally millions)

Low register usage : otherwise limits concurrent threads
- simple kernels, avoid branching

Little/No Synchronization
- avoid waiting, avoid complex code/debugging

Minimize CPU<->GPU copies
- reuse GPU buffers across multiple CUDA launches

How Many Threads to Launch ?

- can (and should) launch many millions of threads
  - largest Opticks launch : 400M threads, at VRAM limit
- maximum thread launch size : so large its irrelevant
- maximum threads inflight : #SM*2048 = 80*2048 ~ 160k
  - best latency hiding when launch > ~10x this ~ 1M

Optical Photon Simulation

Abundant parallelism
- Many millions of photons

Low register usage
- Simple optical physics, texture lookups

Little/No synchronization
- Independent photons -> None

Minimize CPU<->GPU copies
- geometry copied at initialization
- gensteps copied once per event
- only hits copied back

~perfect match for GPU acceleration

Understanding Throughput-oriented Architectures https://cacm.acm.org/magazines/2010/11/100622-understanding-throughput-oriented-architectures/fulltext

NVIDIA Titan V: 80 SM, 5120 CUDA cores

GPU Demands Simplicity (Arrays) -> Big Benefits : NumPy + CuPy

Separate address space -> cudaMemcpy -> Serialization

upload/download : host(CPU)<->device(GPU)

- Serialize everything -> Arrays
- Many small tasks -> Arrays
- Random Access/Order undefined -> Arrays

Object-oriented : mixes data and compute

- complicated serialization
- good for complex systems, up to ~1000 objects

Array-oriented : separate data from compute

- inherent serialization + simplicity
- good for millions of element systems

NumPy : standard array handling package

- simple .npy serialization
- read/write *NumPy* arrays from C++
  https://github.com/simoncblyth/np/blob/master/NP.hh ❏

https://realpython.com/numpy-array-programming/ ❏

Array Serialization Benefits

Persist everything to file -> fast development cycle

- data portability into any environment
- interactive debug/analysis : *NumPy,IPython*
- flexible testing

Can transport everything across network:

- production flexibility : distributed compute

Arrays for Everything -> direct access debug

- (num_photons,4,4) *float32*
- (num_photons,16,2,4) *int16* : step records
- (num_photons,2) *uint64* : history flags
- (num_gensteps,6,4) *float32*
- (num_csgnodes,4,4) *float32*
- (num_transforms,3,4,4) *float32*
- (num_planes,4) *float32*
- ...

Parallel Processing 0th Step : Re-shape Data into "Parallel" Array Form

Q1 : What Shape is Your Data ?

- longest "parallelization" axis first, eg:
  - photons **(400M,4,4)**
  - PMT waveforms **(20k,1000,4)**
  - SiPM time series **(24*60*60,40,4)**

  longer -> more abundantly parallel -> more speedup

- cross reference array elements with indices (not pointers)
  - every photon element has corresponding genstep element
  - works across address spaces

Look for "long" CPU loops

- loop -> CUDA launch
- For example : loops over photons, JUNO PMTs, time series

Q2 : What are Independent Chunks ?

CUDA Thread for each element:

- "owns" single array slot
- runs in undefined order -> no problem

Survey of High Level General Purpose CUDA Packages

- Learn CUDA basics (kernels, thread+memory hierarchy, ...)
  - BUT: base development on higher level libs -> faster start

C++ Based Interfaces to CUDA

- Thrust : https://developer.nvidia.com/Thrust ❏
  - C++ interface to CUDA performance
  - high-level abstraction : reduce, scan, sort
- CUB : http://nvlabs.github.io/cub/ ❏
  - CUDA C++ specific, GPU less hidden
- MGPU : https://github.com/moderngpu/moderngpu ❏
  - teaching tool : examples of CUDA algorithms

Mature NVIDIA Basis Libraries

- cuRAND, cuFFT, cuBLAS, cuSOLVER, cuTENSOR, ...
  - https://developer.nvidia.com/gpu-accelerated-libraries

RAPIDS : New NVIDIA "Suite" of open source data science libs

- GPU-accelerated open source data science suite
  - "... end-to-end data science workflows..." http://rapids.ai/ ❏
  - cuDF : GPU dataframe library, Pandas-on-GPU

CuPy : Simplest CUDA Interface

- https://cupy.chainer.org/ ❏
- NumPy API accelerated by CUDA stack
- plus some of SciPy API

- develop processing with NumPy on CPU
  - switch *numpy->cupy* to test on GPU
  - great for prototyping

"Production" CuPy ? Depends on requirements:

- integrations (eg Geant4, OpenGL, ...)
- control + performance

NumPy : Foundation of Python Data Ecosystem

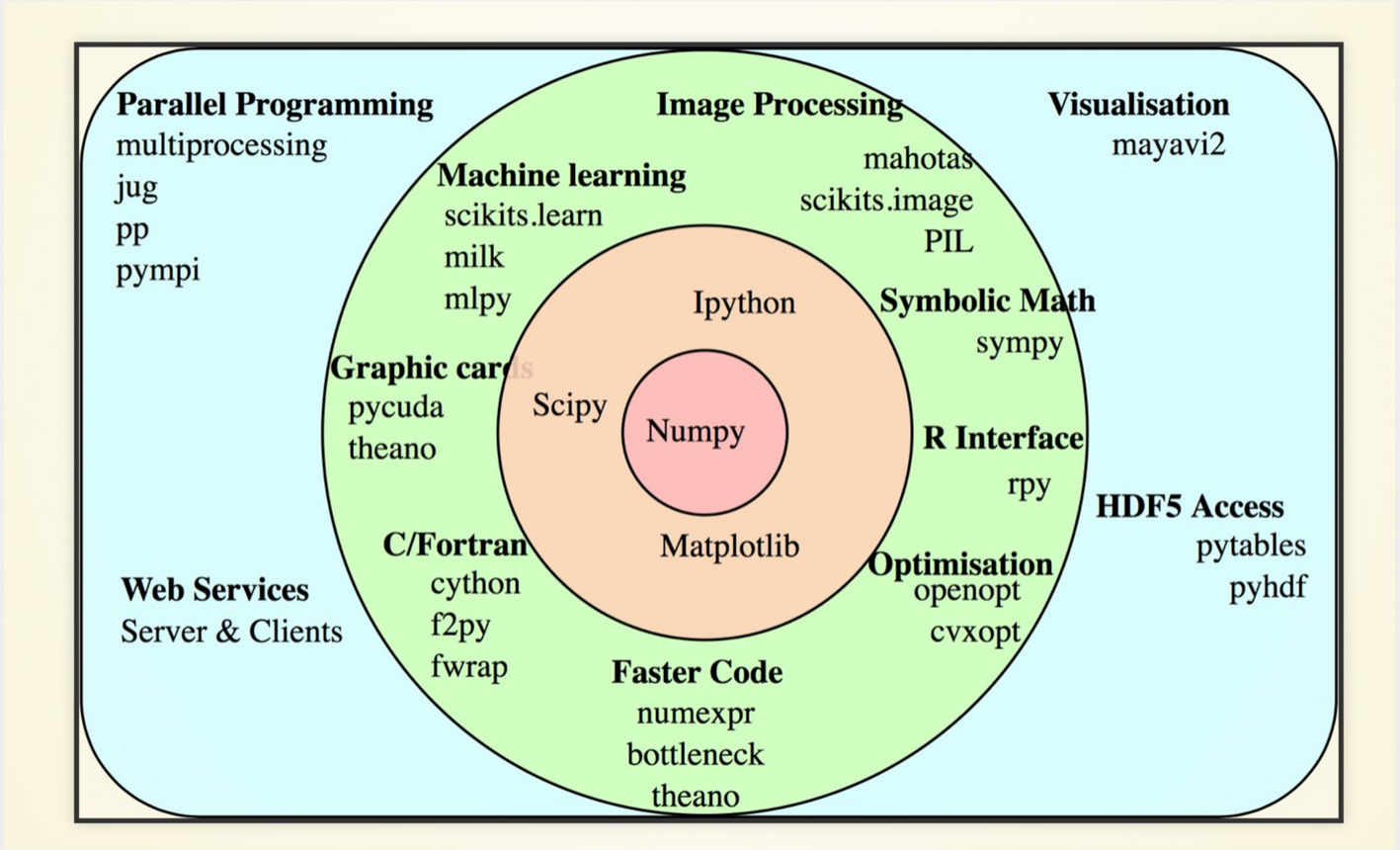
https://bitbucket.org/simoncblyth/intro_to_numpy ❏

Very terse, array-oriented (no-loop) python interface to C performance

- C speed, python brevity + ease
- **array-oriented**
- **vectorized** : no python loops
- efficiently work with large arrays

Recommended paper:

The NumPy array: a structure for efficient numerical computation
https://hal.inria.fr/inria-00564007 ❏

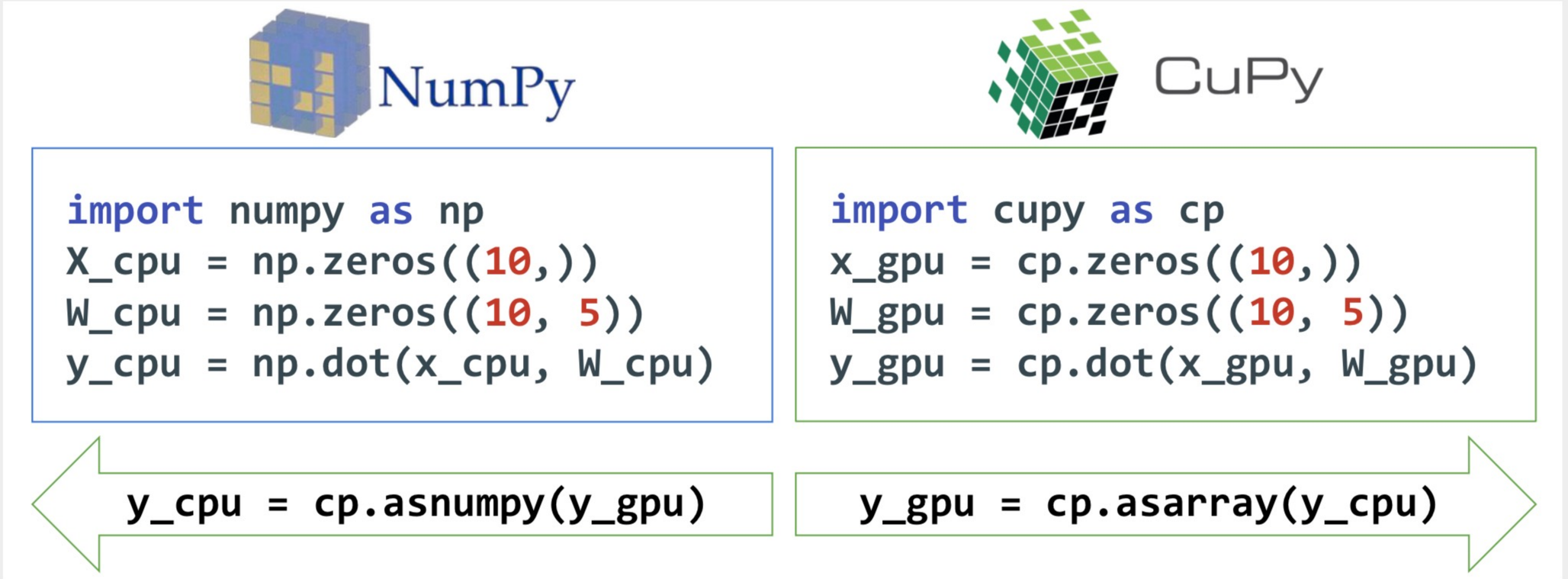


https://docs.scipy.org/doc/numpy/user/quickstart.html ❏

http://www.scipy-lectures.org/intro/index.html ❏

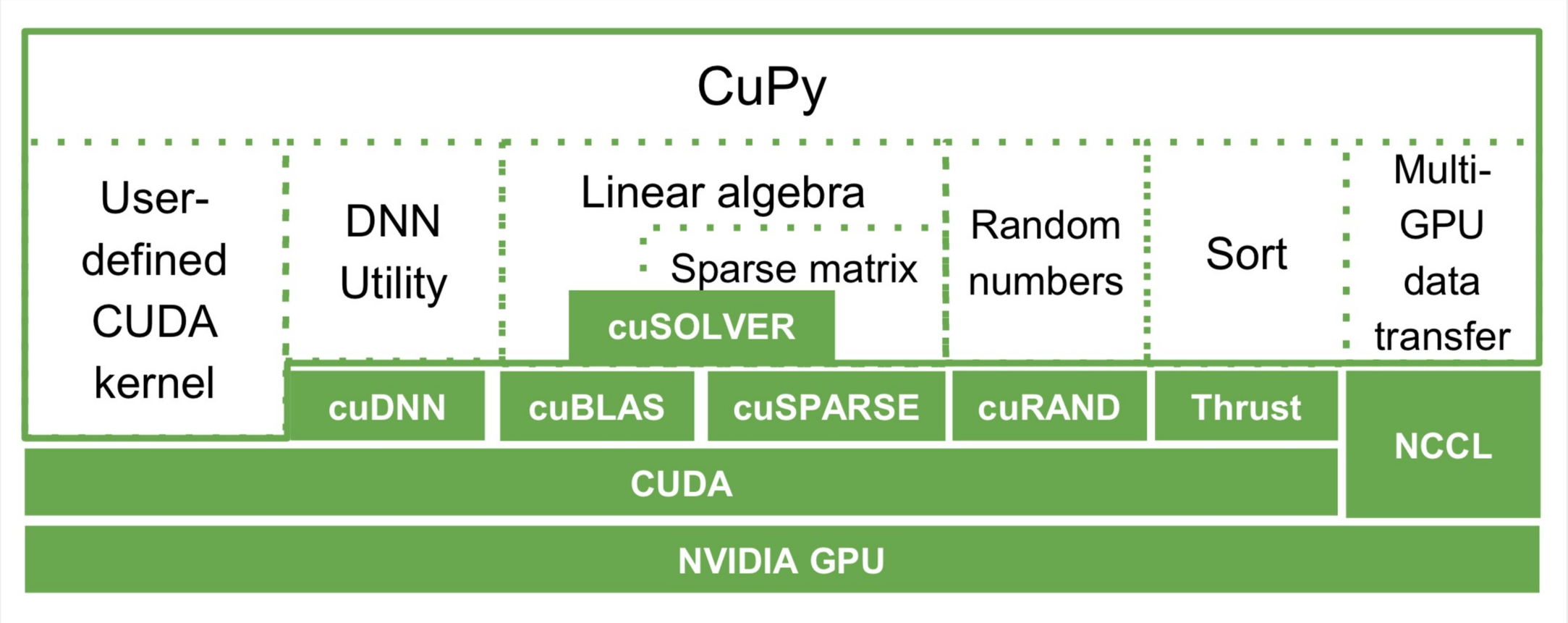
https://github.com/donnemartin/data-science-ipython-notebooks ❏

CuPy : NumPy API (+ some of SciPy) accelerated by NVIDIA CUDA : cuRAND/cuBLAS/cuSOLVER/cuSPARSE/Thrust/NCCL



- *CuPy* is GPU backend of *Chainer* : python deep learning framework developed by Preferred Networks (Japanese startup)
- https://cupy.chainer.org/ ❑ (**pip install cupy-cuda101** OR **conda install cupy**)
- https://github.com/cupy/cupy ❑

CuPy : Easy Interface to NVIDIA CUDA stack



- CuPy : potential for big speedups with little effort, if processing can adopt covered API:
  - https://docs-cupy.chainer.org/en/stable/reference/comparison.html ❏
  - easy introduction to CUDA libraries https://docs.nvidia.com/cuda-libraries/index.html ❏
  - source shows : CUB, cuTENSOR on the way, CuPy is based on Cython

NumPy + CuPy Example : closest approach of Ellipse to a Point

```
import numpy as np, cupy as cp

def ellipse_closest_approach_to_point( xp, ex, ez, _c, N ):
    """ex, ez: ellipse semi-axes, c: coordinates of point in ellipse frame"""
    c = xp.asarray( _c )  ; assert c.shape == (2,)

    t = xp.linspace( 0, 2*np.pi, N )      # t: array of N angles [0,2pi]
    e = xp.zeros( [len(t), 2] )
    e[:,0] = ex*xp.cos(t)
    e[:,1] = ez*xp.sin(t)                          # e: N parametric [x,z] points on the ellipse

    return  e[xp.sum((e-c)**2, axis=1).argmin()]    # point on ellipse closest to c
```

- first argument *xp* is python module : *np* or *cp* : switching between NumPy and CuPy, CPU and GPU implementations
- interactively debug numpy code in *ipython* : check array shapes at every stage

expression	shape	note
`e`	`(10000000,2)`	
`c`	`(2,)`	
`e-c`	`(10000000,2)`	*c* is **broadcast** over *e* : must be compatible shape
`np.sum((e-c)**2, 1)`	`(10000000,)`	`axis=1` : summing over the axis of length 2
`np.sum((e-c)**2, 0)`	`(2,)`	`axis=0` : summing over axis of length 10M
`np.sum((e-c)**2, None)`	`()`	`axis=None` : summing over all elements, yielding scalar

Python vs NumPy vs CuPy

https://bitbucket.org/simoncblyth/intro_to_numpy/src/default/python_vs_numpy_vs_cupy/ellipse_closest_approach_to_point.py ❏

```
17 import numpy as np, cupy as cp
...
67 if __name__ == '__main__':
68
69     N = 10000000     # 10M is large enough to make overheads negligible
70     M = 8            # repeat timings
71
72     ex,ey = 10,20      # parameters of ellipse
73     c = [100,100]       # point to check
74
75     r = {}               # python dict for results
76     t = np.zeros(M)      # numpy array for timings
77
78     for i in range(M):
79         if i == 0:
80             r[i],t[i] = timed(pure_python_ellipse_closest_approach_to_point_DO_NOT_DO_THIS)(ex,ey,c,N)
81         else:
82             xp = np if i < M/2 else cp   # switch between NumPy and CuPy implementations of same API
83             r[i],t[i] = timed(ellipse_closest_approach_to_point)( xp, ex,ey,c, N )
84         pass
85     pass
86     for k in r.keys():
87         if k == M/2: print("")
88         print(k, r[k], t[k])
89     pass
```

Python vs NumPy vs CuPy : Python/NumPy ~ 10, NumPy/CuPy ~ 1300 (CUDA 10.1, NVIDIA TITAN V)

```
[blyth@localhost python_vs_numpy_vs_cupy]$ ipython -i ellipse_closest_approach_to_point.py
                    # "ipython -i" : run python script leaving context for interactive examination

Python 2.7.15 |Anaconda, Inc.| (default, May  1 2018, 23:32:55)
IPython 5.7.0 -- An enhanced Interactive Python.
...
(0, (4.983054096206095, 17.34003135802051), '6.266726970672607')
(1, array([ 4.9830541 , 17.34003136]), '0.674299955368042')
(2, array([ 4.9830541 , 17.34003136]), '0.6548769474029541')
(3, array([ 4.9830541 , 17.34003136]), '0.6450159549713135')

(4, array([ 4.9830541 , 17.34003136]), '0.4854261875152588')     # 1st CuPy run CUDA compiles populating cache
(5, array([ 4.9830541 , 17.34003136]), '0.000415802001953125')
(6, array([ 4.9830541 , 17.34003136]), '0.0003409385681152344')
(7, array([ 4.9830541 , 17.34003136]), '0.000762939453125')

In [1]: tpy = t[0]
In [2]: tnp = np.average(t[1:M/2])
In [3]: tcp = np.average(t[M/2+1:])

In [4]: tpy/tnp
Out[4]: 9.522970786915796      #  NumPy ~ 10x Python

In [5]: tnp/tcp
Out[5]: 1299.0845622842799     #  CuPy > 1000x NumPy : with almost zero effort can apply the CUDA stack
```

- *CuPy* looks great for prototyping
- mirroring *NumPy* API is **extremely convenient/flexible** (unlike PyCUDA, Numba.cuda)
- **develop with NumPy without GPU, occasionally checking identical code gets expected CuPy speedup**

Persist NumPy Arrays to .npy Files from Python, Examine File Format

IPython persisting NumPy arrays:

```
In [1]: a = np.arange(10)            # array of 10 long (int64)
In [2]: a
Out[2]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [3]: np.save("/tmp/a.npy", a )    # serialize array to file

In [4]: a2 = np.load("/tmp/a.npy")   # load array into memory
In [5]: assert np.all( a == a2 )     # check all elements the same
```

IPython examine NPY file format:

```
In [6]: !xxd /tmp/a.npy              # xxd hexdump file contents
                                     # minimal header : type and shape

00000000: 934e 554d 5059 0100 7600 7b27 6465 7363  .NUMPY..v.{'desc
00000010: 7227 3a20 273c 6938 272c 2027 666f 7274  r': '<i8', 'fort
00000020: 7261 6e5f 6f72 6465 7227 3a20 4661 6c73  ran_order': Fals
00000030: 652c 2027 7368 6170 6527 3a20 2831 302c  e, 'shape': (10,
00000040: 292c 207d 2020 2020 2020 2020 2020 2020  ), }
00000050: 2020 2020 2020 2020 2020 2020 2020 2020
00000060: 2020 2020 2020 2020 2020 2020 2020 2020
00000070: 2020 2020 2020 2020 2020 2020 2020 200a                .
00000080: 0000 0000 0000 0000 0100 0000 0000 0000  ................
00000090: 0200 0000 0000 0000 0300 0000 0000 0000  ................
..
```

Load NumPy array into C/C++

Straightforward to parse NPY files

http://github.com/simoncblyth/np/ ❏

NP::Load

- parses file header, array shape + type
- reads array data into std::vector

```
// gcc NPMinimal.cc -lstdc++ && ./a.out /tmp/a.npy
#include "NP.hh"
int main(int argc, char** argv)
{
    assert( argc > 1 && argv[1] ) ;
    NP* a = NP::Load(argv[1]) ;
    a->dump();
    return 0 ;
}
```

Create and Write .npy Array from C/C++ with NP.hh header (No Python)

- **NP.hh** : https://github.com/simoncblyth/np/blob/master/NP.hh ❏

Check C++ created NumPy array from commandline:

```
$ python -c "import numpy as np ; print(np.load('/tmp/a.npy'))"
[[[ 0.  1.  2.  3.]
  [ 4.  5.  6.  7.]]

 [[ 8.  9. 10. 11.]
  [12. 13. 14. 15.]]

 [[16. 17. 18. 19.]
  [20. 21. 22. 23.]]]
```

Access/Create Array Data Anywhere

- Python : *np.load np.array np.save* ...
- C/C++ : NP.hh **NP struct** methods
- CUDA : *cudaMemcpy()* Arrays to/from GPU
  - OR via CuPy *cp.load cp.save cp.array*

google:"parse NumPy file with Java/Rust/R/Swift/Android/..."

Create 3D NumPy Array from C/C++

```
// gcc NPMiniMake.cc -lstdc++ && ./a.out 3 2 4
#include "NP.hh"
int main(int argc, char** argv)
{
    int ni = argc > 1 ? atoi(argv[1]) : 10  ;
    int nj = argc > 2 ? atoi(argv[2]) : 1   ;
    int nk = argc > 3 ? atoi(argv[3]) : 4   ;

    NP* a = new NP(ni,nj,nk) ;

    for(int i=0 ; i < ni ; i++ ){
    for(int j=0 ; j < nj ; j++ ){
    for(int k=0 ; k < nk ; k++ ){

        int index = i*nj*nk + j*nk + k ; // 3d -> 1d
        a->data[index] = float(index) ;  // dummy value
    }
    }
    }
    a->save("/tmp/a.npy") ;
    return 0 ;
}
```

http://github.com/simoncblyth/np/ ❏

C Union "Trick" Mixed Type Arrays -> Simpler + Faster CUDA Usage

http://bitbucket.org/simoncblyth/intro_to_numpy/src/tip/union_trick/

```
//uif.h
#pragma once
union uif_t {  // store three 32-bit types at same location
    unsigned u;
    int i ;
    float f;
} ;
```

```
In [1]: import numpy as np
In [2]: a = np.ones(3, dtype=np.float32)

In [3]: a.view(np.int32)[1] = -10   # int32 inside float32 array
 // NumPy view reinterprets same bits as different type

In [4]: a
Out[4]: array([ 1., nan,  1.], dtype=float32)

In [5]: a.view(np.int32)
Out[5]: array([1065353216,        -10, 1065353216], dtype=int32)
```

CUDA **reinterpret** device functions, also **__int_as_float**:

Store int/uint bits into float array

```
#include <iostream>
#include <cassert>
#include "uif.h"

int main(int argc, char** argv)
{
    int value = argc > 1 ? atoi(argv[1]) : -10 ;
    uif_t uif ;
    uif.i = value ;

    float a[3] = { 1.f, 1.f, 1.f } ;
    a[1] = uif.f ; // plant int bits into float array

    uif_t uif2 ;
    uif2.f = a[1] ; // float copy

    assert( uif2.i == value ) ; // recover int
    // std::cout << ... elided for brevity
    return 0 ;
}
```

```
$ gcc uifDemo.cc -lstdc++ && ./a.out -10
uif.u  4294967286 uif.i  -10 uif.f  nan
uif2.u 4294967286 uif2.i -10 uif2.f nan
```

- *Opticks* photon arrays : mostly float + int flags

Example 2 : OpenGL Interactive Photon History Selection

Photon record renderer, OpenGL geometry shader extracts:

```
01 #version 410 core
..
37 uniform ivec4 RecSelect ;  //  constant input to the "launch"
..
44 in ivec4 sel[];            //  input array
..
46
47 layout (lines) in;
48 layout (points, max_vertices = 1) out;
..
52 void main ()
53 {
54     uint seqhis = sel[0].x ;
55     uint seqmat = sel[0].y ;
56     if( RecSelect.x > 0 && RecSelect.x != seqhis )  return ;
57     if( RecSelect.y > 0 && RecSelect.y != seqmat )  return ;
..     // History and Material Selection
63     vec4 p0 = gl_in[0].gl_Position  ;
64     vec4 p1 = gl_in[1].gl_Position  ;
65     float tc = Param.w / TimeDomain.y ;
66     // for interpolation of photon position at uniform input time

..
```

- https://bitbucket.org/simoncblyth/opticks/src/default/oglrap/gl/rec/geom.glsl ❏

OpenGL Geometry Shaders

- typically : vertices -> primitives
- can also emit nothing : unlike other shaders
- use this flexibility for history selection

-> need "popularity" index for each photon

Photon History Sequence (64-bit int)

- eg: 0x7cc6d "TO SC BT BT SD"

Example 2 : Photon History Indexing with CUDA Thrust

https://bitbucket.org/simoncblyth/opticks/src/default/thrustrap/TSparse_.cu

```
089 template <typename T>
 90 void TSparse<T>::count_unique()
 91 {
 ///     preparation of src with strided range iterator elided for bevity
 97
 98     thrust::device_vector<T> data(src.begin(), src.end());  // GPU copy to avoid sorting original
 99
100     thrust::sort(data.begin(), data.end());  // GPU sort of millions of integers
101
102     // inner_product of sorted data with shifted by one self finds "edges" between values
103     m_num_unique = thrust::inner_product(
104                                    data.begin(),data.end() - 1,   // first1, last1
105                                               data.begin() + 1,   // first2
106                                                          int(1),   // output type init
107                                                 thrust::plus(),   // reduction operator
108                                      thrust::not_equal_to()    // pair-by-pair operator, returning 1 at edges
109                                        );
116     m_values.resize(m_num_unique);
117     m_counts.resize(m_num_unique);
118
119     // find all unique key values with their counts
120     thrust::reduce_by_key(
121                                    data.begin(),    // keys_first
122                                      data.end(),    // keys_last
123            thrust::constant_iterator(1),    // values_first
124                                  m_values.begin(),    // keys_output
125                                  m_counts.begin()     // values_output
126                             );
///     sort into descending key order elided for brevity
147 }
```

Using Thrust will improve your C++ skills

Applying CUDA Thrust -> Translate Task into Processing "Primitives"

Benefit from highly optimized GPU implementations

- number of unique values
  - -> **thrust::inner_product** with sorted self shifted by one
- find all unique keys with their counts
  - -> **thrust::reduce_by_key**
- sorted counts for each value
  - -> **thrust::sort_by_key**

Thrust Hides GPU Details

- helpful when starting, can reach thru to lower level
- easy interoperation with
  - CUDA, OpenGL, NVIDIA OptiX

Thrust : High level C++ interface to CUDA

Flavor of Thrust "Primitives"

```
adjacent_difference
copy_if
exclusive_scan
exclusive_scan_by_key
for_each
gather
generate
inclusive_scan
inner_product
max_element
merge_by_key
min_element
minmax_element
reduce
reduce_by_key
scatter
sort
tabulate
transform
transform_inclusive_scan
transform_reduce
unique_by_key
```

- https://thrust.github.io/doc/index.html ❑

thrust::inner_product

◆ inner_product() [4/4]

template<typename InputIterator1 , typename InputIterator2 , typename OutputType , typename BinaryFunction1 , typename BinaryFunction2 >

```
OutputType thrust::inner_product ( InputIterator1   first1,
                                   InputIterator1   last1,
                                   InputIterator2   first2,
                                   OutputType       init,
                                   BinaryFunction1  binary_op1,
                                   BinaryFunction2  binary_op2
                                 )
```

`inner_product` calculates an inner product of the ranges `[first1, last1)` and `[first2, first2 + (last1 - first1))`.

This version of `inner_product` is identical to the first, except that is uses two user-supplied function objects instead of `operator+` and `operator*`.

Specifically, this version of `inner_product` computes the sum `binary_op1( init, binary_op2(*first1, *first2) ), ...`

Unlike the C++ Standard Template Library function `std::inner_product`, this version offers no guarantee on order of execution.

An example of Thrust documentation

thrust::reduce_by_key

◆ reduce_by_key() [2/6]

```
template<typename InputIterator1 , typename InputIterator2 , typename OutputIterator1 , typename OutputIterator2 >
thrust::pair<OutputIterator1,OutputIterator2> thrust::reduce_by_key ( InputIterator1  keys_first,
                                                                      InputIterator1  keys_last,
                                                                      InputIterator2  values_first,
                                                                      OutputIterator1 keys_output,
                                                                      OutputIterator2 values_output
                                                                    )
```

`reduce_by_key` is a generalization of `reduce` to key-value pairs. For each group of consecutive keys in the range `[keys_first, keys_last)` that are equal, `reduce_by_key` copies the first element of the group to the `keys_output`. The corresponding values in the range are reduced using the `plus` and the result copied to `values_output`.

This version of `reduce_by_key` uses the function object **`equal_to`** to test for equality and `plus` to reduce values with equal keys.

Best way to learn Thrust API is exercise it by writing small tests

- https://bitbucket.org/simoncblyth/opticks/src/default/thrustrap/tests/ ❏
- Thrust implemented with template metaprogramming
  - -> quite slow compilation
  - thousands of lines of compilation errors when mis-using API

Summary of a Mental Model for Effective GPU Usage

- GPU Constraints -> Simplicity -> Arrays -> Advantages
  - Remember graphical origins
  - Easy Serialization : access data anywhere
  - Standard Tools : NumPy, CuPy
- Parallel Processing 0th Step : Re-shape Data into "Parallel" Array Form
  - most important thing for effective GPU usage :
    - shape of your data
- High Level Tools
  - CuPy + NumPy are the best way to prototype GPU processing
- Array Mechanics
  - array simplicity makes them easy to handle
    - once you know a few tricks
- When Python not appropriate:
  - try first CUDA Thrust