OptiX Performance Tools and Tricks [35 mins]

Siggraph 2019

AGENDA

How do I make it **work**?

Debugging

How do I make it **fast**?

Architecting & Profiling

Summary / Q&A

How do I make it work?

OptiX 7 has a NEW! learning curve

Start from a working sample

⇒ Take time to understand the Shader Binding Table (SBT)

Primary tools:

- Bootstrap: OptiX logging & exceptions
- Debug: NSight VSE / cuda-gdb
- DIY: visual debugging, printf

OptiX Logging & Exceptions

- See **`logString[Size]`** params to:
  - **`optixPipelineCreate()`, `optixModuleCreateFromPTX()`, `optixProgramGroupCreate()`**
- See **`optixDeviceContextSetLogCallback()`**

USE BOTH ^^

- See **`OptixPipelineCompileOptions`** for exceptions:

  stack overflow, trace depth, debug, **user**

New Tools!

CUDA tools get new OptiX skills this August

- Debug in Windows: Nsight Visual Studio Edition 2019.3
  - Break & step in code called by OptiX
  - See Source/SASS/PTX
  - Inspect GPU State, analyze GPU core dumps
- Debug in Linux: cuda-gdb
  - Break & step in SASS called by OptiX
  - Inspect GPU State, analyze GPU core dumps
- Nsight Compute: profiling...

NsightVSE Debugging OptiX Application Code

OptiX-Samples (Debugging) - Microsoft Visual Studio

File Edit View Project Build Debug Team Nsight Tools Test Analyze Window Help

Process: [25892] optixPathTracer.exe Thread: [1] CUDA Thread Stack Frame: closesthit__Z7diffusev_ptx0x66d10b554b9

sutil.cpp | optixPathTracer.cu

optixPathTracer | (Global Scope) | diffuse()

```
196  RT_PROGRAM void diffuse()
197  {
198      float3 world_shading_normal   = normalize( rtT
199      float3 world_geometric_normal = normalize( rtT
200      float3 ffnormal = faceforward( world_shading_n
201
202      float3 hitpoint = ray.origin + t_hit * ray.dir
203
204      //
205      // Generate a reflection ray.  This will be tr
206      //
207      current_prd.origin = hitpoint;
208
209      float z1=rnd(current_prd.seed);
210      float z2=rnd(current_prd.seed);
211      float3 p;
212      cosine_sample_hemisphere(z1, z2, p);
213      optix::Onb onb( ffnormal );
214      onb.inverse_transform( p );
215      current_prd.direction = p;
216
217      // NOTE: f/pdf = 1 since we are perfectly impo
218      // with cosine density.
219      current_prd.attenuation = current_prd.attenuat
220      current_prd.countEmitted = false;
221
222      //
223      // Next event estimation (compute direct light
224      //
225      unsigned int num_lights = lights.size();
226      float3 result = make_float3(0.0f);
227
228      for(int i = 0; i < num_lights; ++i)
```

110 %

optix_internal.h | Disassembly

Address: esthit__Z7diffusev_ptx0x66d10b554b965e4a_ss_0 CUDA Disassembly: PTX and SASS

Viewing Options

```
0x000001b3f4d00530                LDG.E.128.CONSTANT.SYS R16, [R30+0x40]
0x000001b3f4d00540  [130] ld.global.nc.v4.u32 { %r152,%r153,%r154,%r155 }, [%r
0x000001b3f4d00540                LDG.E.128.CONSTANT.SYS R20, [R20]
0x000001b3f4d00550  [146] ld.global.nc.v4.u32 { %r160,%r161,%r162,%r163 }, [%r
0x000001b3f4d00550                LDG.E.128.CONSTANT.SYS R4, [R30+0x50]
--- C:/ProgramData/NVIDIA Corporation/OptiX SDK 6.6.0/include/optixu/optixu_ma
   643:   return make_float3(a.x + b.x, a.y + b.y, a.z + b.z);
0x000001b3f4d00560  [266] fma.rn.ftz.f32    %f4, %f116, %f108, %f105;
0x000001b3f4d00560                FFMA.FTZ R8, R15, R12, R8
0x000001b3f4d00570  [267] fma.rn.ftz.f32    %f5, %f116, %f109, %f106;
0x000001b3f4d00570                FFMA.FTZ R9, R15, R13, R9
0x000001b3f4d00580  [268] fma.rn.ftz.f32    %f6, %f116, %f110, %f107;
0x000001b3f4d00580                FFMA.FTZ R10, R15, R14, R10
--- C:/ProgramData/NVIDIA Corporation/OptiX SDK 6.6.0/SDK/optixPathTracer/opti
   220:     current_prd.countEmitted = false;
0x000001b3f4d00590  [275] st.u32    [%rd4+68], %r327;
0x000001b3f4d00590                ST.E.SYS [R32+0x44], RZ
   207:     current_prd.origin = hitpoint;
0x000001b3f4d005a0  [271] st.f32    [%rd4+36], %f4;
0x000001b3f4d005a0                ST.E.SYS [R32+0x24], R8
0x000001b3f4d005b0  [272] st.f32    [%rd4+40], %f5;
0x000001b3f4d005b0                ST.E.SYS [R32+0x28], R9
0x000001b3f4d005c0  [273] st.f32    [%rd4+44], %f6;
0x000001b3f4d005c0                ST.E.SYS [R32+0x2c], R10
--- C:/ProgramData/NVIDIA Corporation/OptiX SDK 6.6.0/include/internal/optix_i
   652:     asm volatile("call (%0, %1, %2, %3), _rt_transform_tuple, (%4, %5,
0x000001b3f4d005d0  [217] mul.rn.f32    %f87, %f75, %f30;
0x000001b3f4d005d0                FMUL R38, R41, R17
0x000001b3f4d005e0  [216] mul.rn.f32    %f86, %f72, %f29;
0x000001b3f4d005e0                FMUL R19, R39, R21
0x000001b3f4d005f0  [218] add.f32   %f88, %f86, %f87;
0x000001b3f4d005f0                FADD R23, R19, R38
0x000001b3f4d00600  [219] mul.rn.f32    %f89, %f78, %f31;
0x000001b3f4d00600                FMUL R30, R43, R5
0x000001b3f4d00610  [211] mul.rn.f32    %f81, %f74, %f30;
0x000001b3f4d00610                FMUL R38, R41, R16
0x000001b3f4d00620  [210] mul.rn.f32    %f80, %f71, %f29;
0x000001b3f4d00620                FMUL R19, R39, R20
```

Warp Info

Enter filter Viewing 1701/1701

Context	SM Version	Grid ID	Shader Info	Threads
1b36b0ddcd0	00070005	00000111	CTA: (0, 0, 0), Thread: (0, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 0, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 1, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 2, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 3, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 4, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 5, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 6, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 7, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 8, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 9, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 10, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 11, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 12, 0), Thread: (32, 0, 0)	
1b36b0ddcd0	00070005	00000111	CTA: (0, 13, 0), Thread: (32, 0, 0)	

GPU Registers

SASS

```
R0  = 01000000  R1  = 00fffc80  R2  = 09c80a80  R3  = 779db190  R4  = 00000000  R5  = 00000000
R6  = 3f800000  R7  = 00000000  R8  = 440b0000  R9  = 3fb810e1  R10 = 3f2907bd  R11 = 00000000
R12 = 3e911a7e  R13 = be8dbe3e  R14 = 3f6b0d54  R15 = 44753b5c  R16 = 00000000  R17 = 3f800000
R18 = 00000000  R19 = 00000000  R20 = 3f800000  R21 = 00000000  R22 = 00000000  R23 = 00000000
R24 = 09e005c0  R25 = 0000000b  R26 = 0000000e  R27 = 00001605  R28 = 7ce20011  R29 = 0100ff02
R30 = 09e13400  R31 = 0000000b  R32 = 6ffffbe0  R33 = 000001b3  R34 = 6ffffbb0  R35 = 000001b3
R36 = 6ffffb00  R37 = 000001b3  R38 = 00000000  R39 = bf7fffff  R40 = 09e11020  R41 = 00000000
R42 = 00400444  R43 = 00000000  R44 = 00000000  R45 = bf7fffff  R46 = 00000000  R47 = 00000000
R48 = 00000000  R49 = 00000000  R50 = 00000000  R51 = 00000000  R52 = 00000000  R53 = 00000000
R54 = 00000000  R55 = 00000000  R56 = fffffffd  R57 = 3f800000  R58 = 447687cc  R59 = 00000000
R60 = 00000f30  R61 = 00000000  R62 = 00000000  R63 = 3b2bb92b  R64 = 6ffffbe0  R65 = 000001b3
R66 = 3f2907bd  R67 = 3f2907bd  R68 = 3a9ac355  R69 = 00000000  R70 = 00000f20  R71 = 3aea6459
R72 = 00000f10  R73 = 3fb810e1
```

Registers | GPU Registers

Locals | Call Stack | Breakpoints

Solution Explorer | Team Explorer

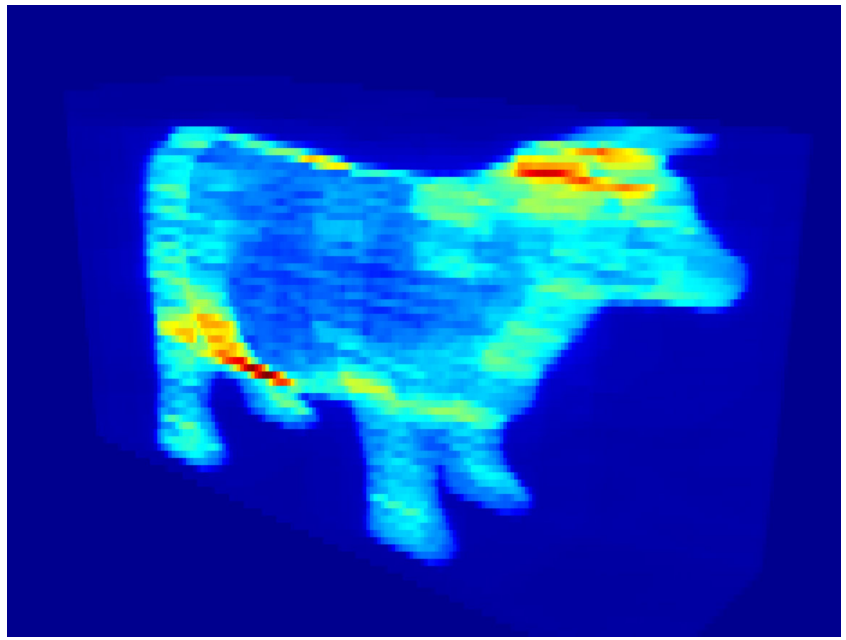
Visual output & printf

Assign pixel color to a variable

Good for debugging & profiling

Example: “time view”

see: optixMotionBlur in OptiX 6 SDK

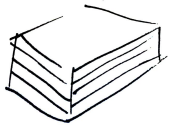


Last, not least: printf

Easy, unsophisticated & effective

Useful to filter w/ launch index (mouse coords)

Stack sizes



In optix_stack_size.h:

```
OptixStackSizes
optixUtilAccumulateStackSizes()
optixUtilComputeStackSizes()
optixPipelineSetStackSize()
```

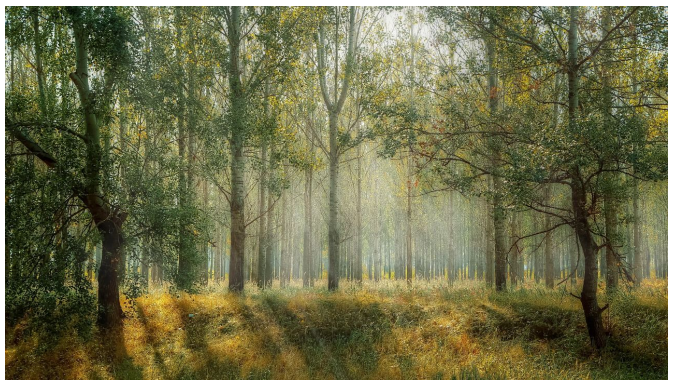
These analyze your shaders. Source code included!

Detailed control over stack size
See optixPathTracer for example

How do I make it fast?

Philosophy

- Macro View:
  - Understand & architect
  - Know the machine
  - Be aware of design goals
- Micro View:
  - Profile & optimize, when ready
  - Use to uncover unexpected bottlenecks
  - Invaluable (but also not a magic bullet)



How do I make it fast?

Understand & architect

Go async with CPU threads & CUDA streams

Use OptiX “built-in” triangles

Keep GPU thread workload locally similar

Avoid unnecessary round trips from RT cores to SM

Reduce memory use

Flatten scene, use BVH refit

How do I make it fast?

Understand & architect

<u>Caveat emptor</u>

Goal is understanding. Know which tradeoffs to make.

There is no right answer.

None of these suggestions are absolute.

Go async

Use CPU threads & CUDA streams

New OptiX 7 API is *almost* completely thread safe

OptiX uses CUDA resources natively; CUDA interop API no longer needed

CUDA streams allow concurrent kernels

CUDA events allow synchronizing

⇒ Example: concurrent buffer copies w/ **`cudaMemcpyAsync()`**

GPU workload: keep it locally similar

Understanding warps

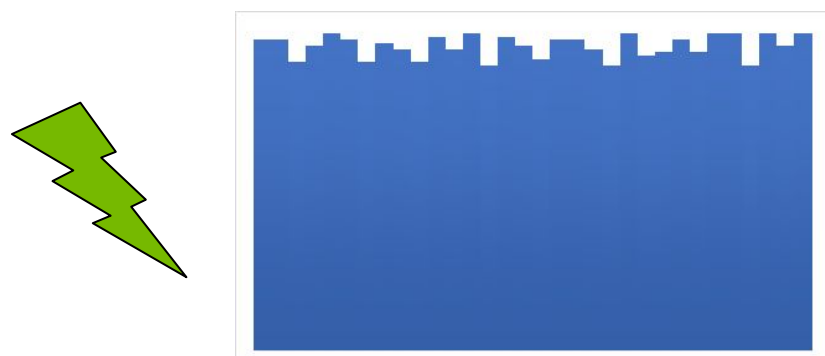
Remember GPUs are SIMD

A warp is a group of 32 threads

When one thread diverges***, the others wait

Example 1: # optixTrace() calls / thread

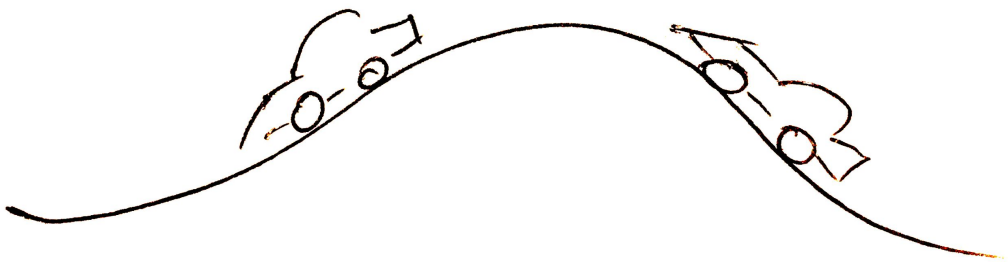
Example 2: 1 Uber shader vs many specialized

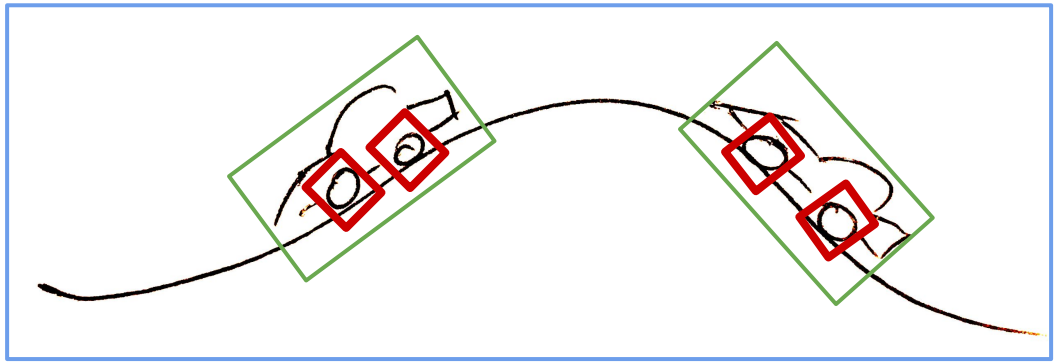


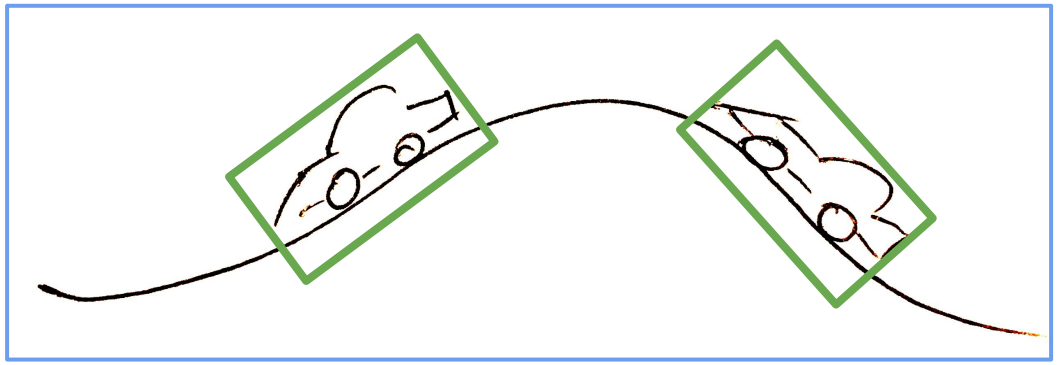
BVH: flatten scene

Hardware instancing: your first level is free!

Flatten if you can



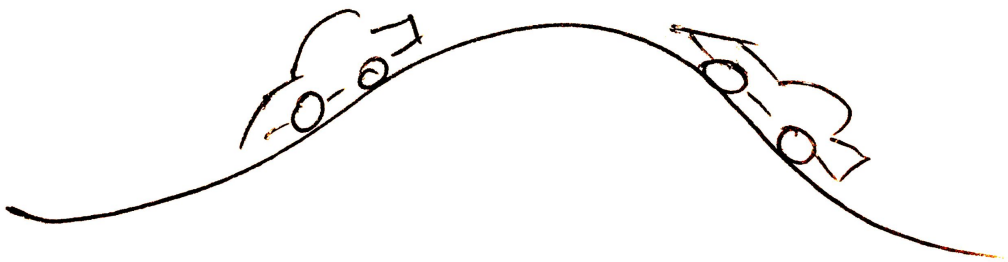


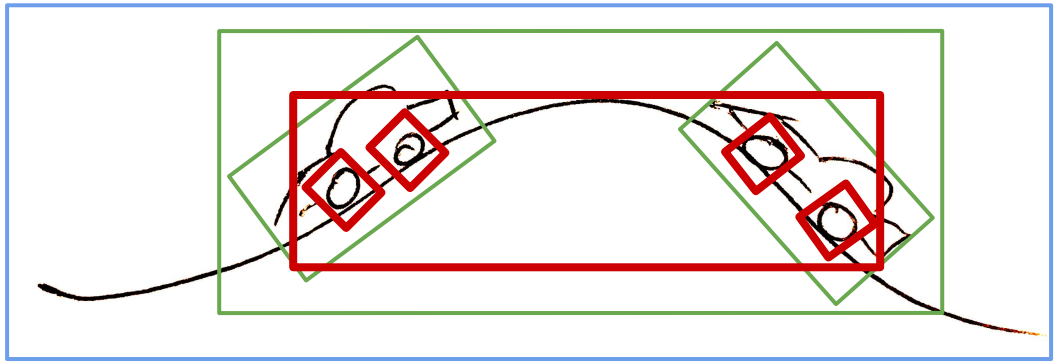


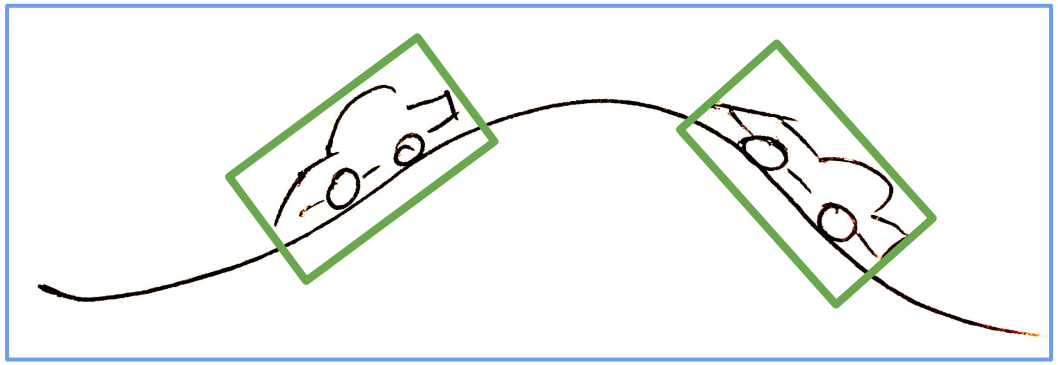
BVH: flatten scene

Hardware instancing: your first level is free!

Merge if you can







BVH: use refit [dynamic scenes]

Rebuild: start over

Required when topology changes

Refit: adjust positions

~10x faster than rebuild

Tree does not change (not very compatible with flattening)

Large moves may cause excess node overlap (slower trace)

⇒ Evaluate the trade-off

Reduce memory access

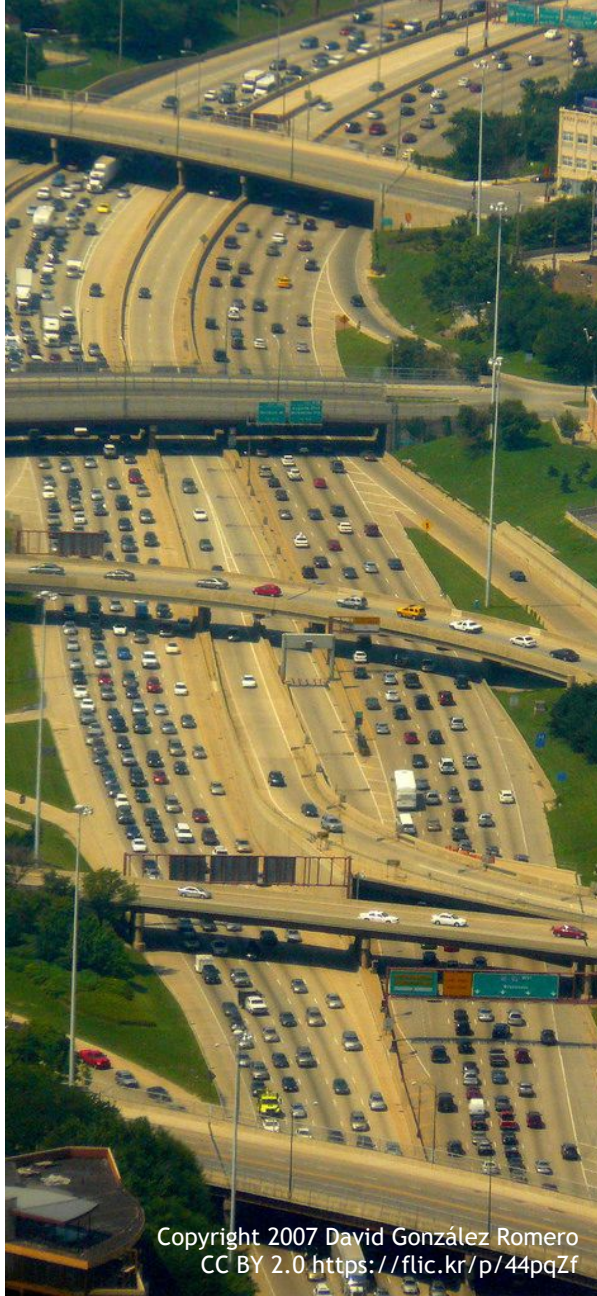
Quadro RTX 6000 memory bandwidth: 672 GB/s

IF 10 GRays/s is your goal, then:

672 GB/s ÷ 10 GRays/s = 67.2 bytes / ray

N.B.: includes traversal

⇒ Care must be taken to stay in this budget



Ways to reduce memory access

There are 8 32-bit registers for payload & attribs

- Trim to bare minimum
- Using 8 or less? Stick to registers, no pointer needed
- Using more than 8? *May* help to pass 6 most used + 64b pointer

Use iteration instead of recursion

Share BVH for identical instances

BVH compaction

Plus the usual: your data format, quantization, compression, etc.

RT Cores: avoid unnecessary round trips

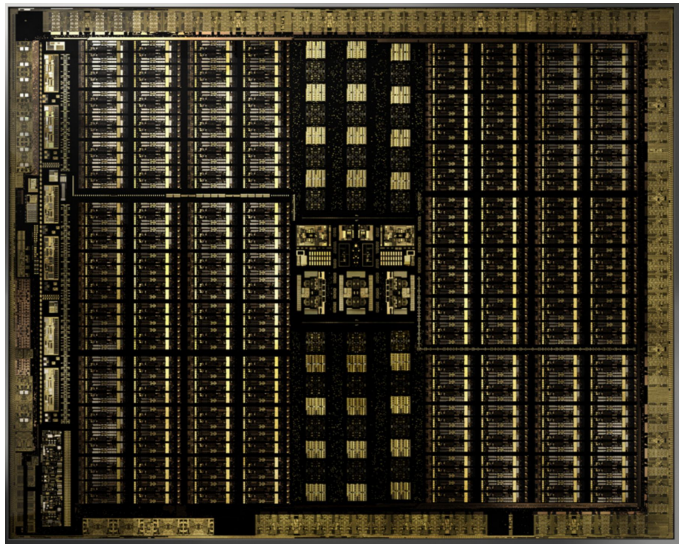
Don’t forget **Amdahl's Law**

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

Biggest speedup when traversal bound

Both obvious and easy to forget

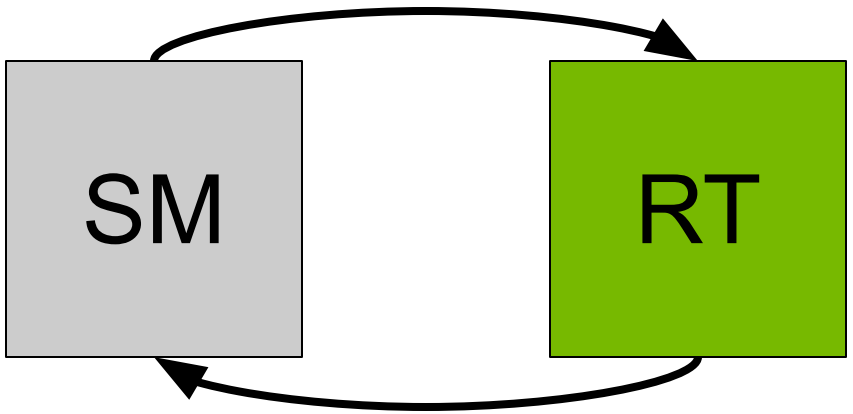
Imagine 50/50 shade vs traverse+intersect



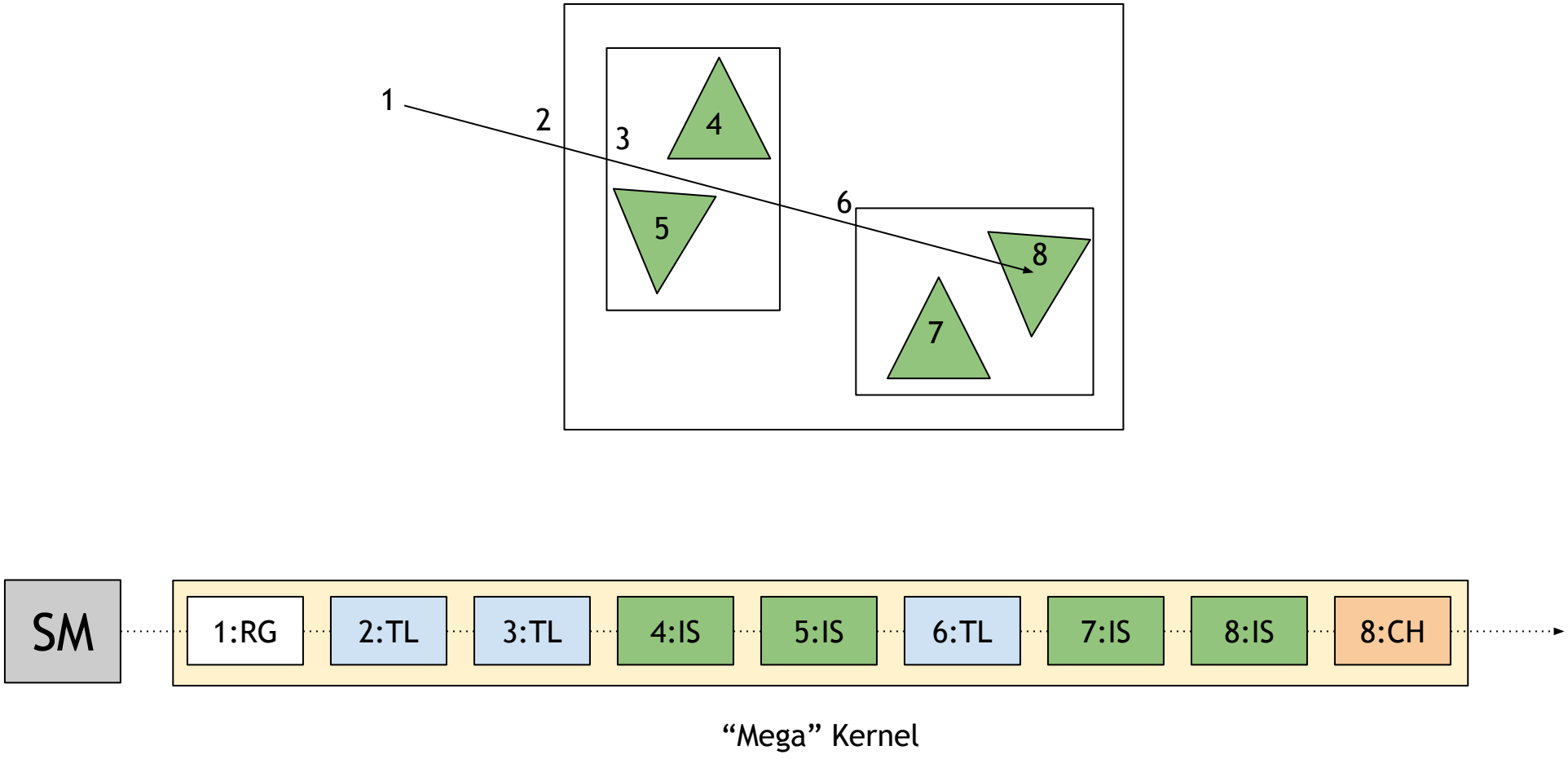
SM + RT Core round-trips

Some features run CUDA code (your programs) mid-traversal

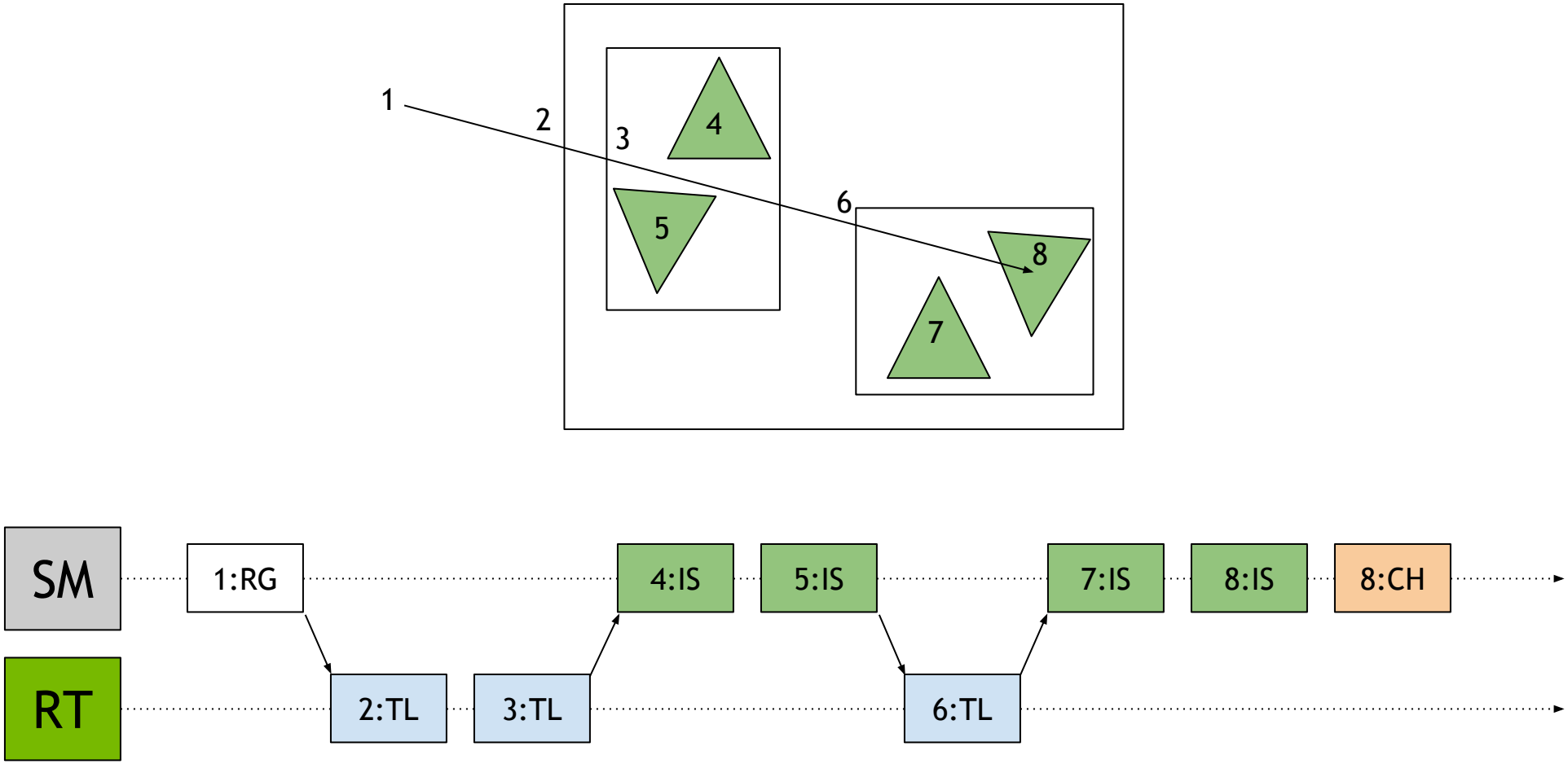
Let’s build a mental model of how the SMs & RT Cores interact



OptiX 5 traversal (Pre-RTX)

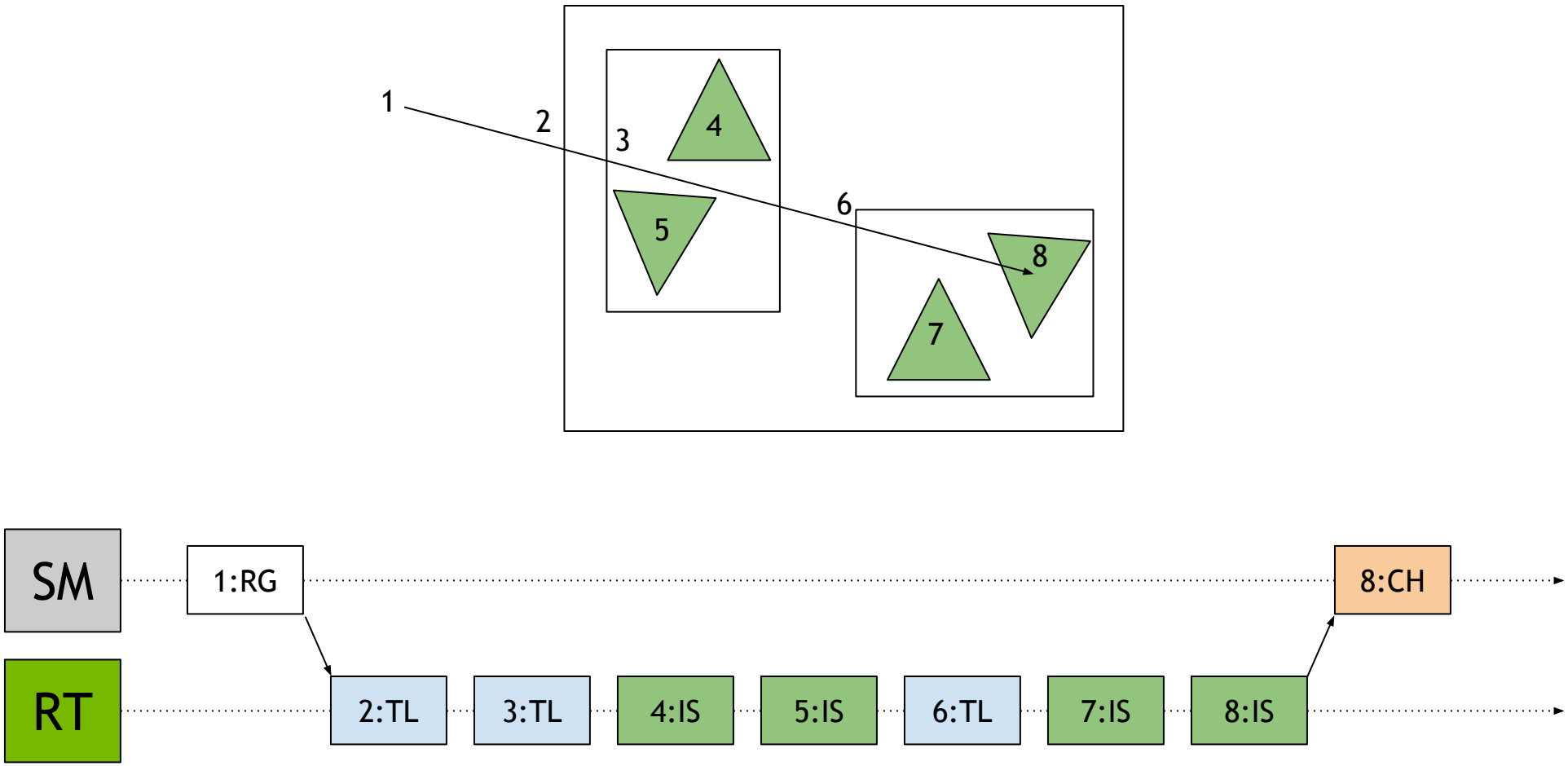


RTX traversal: custom primitives

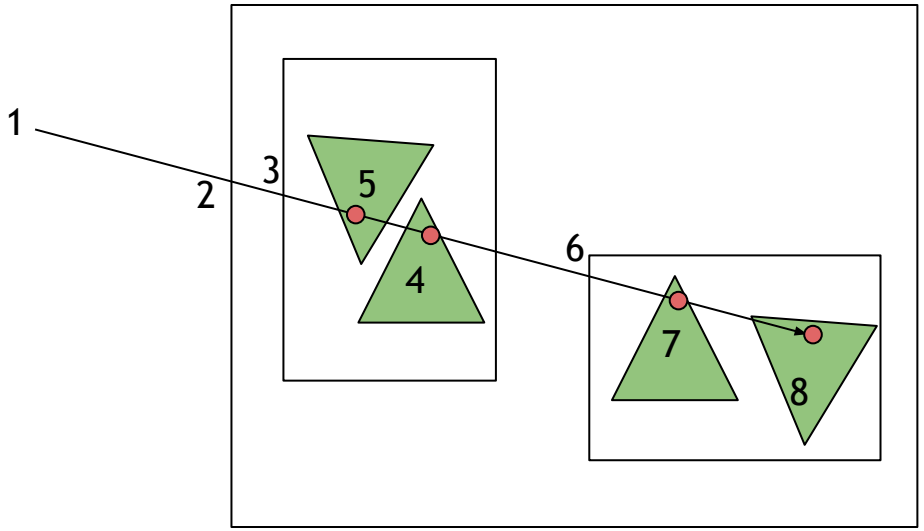


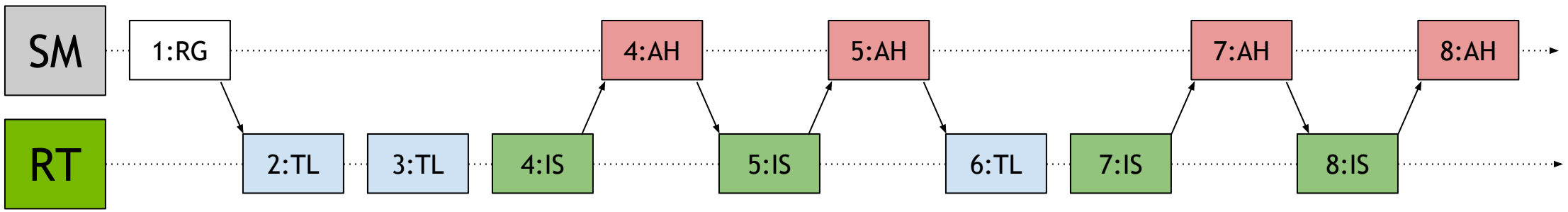
*NB: conceptual model of execution, not timing.

RTX traversal: built-in triangles

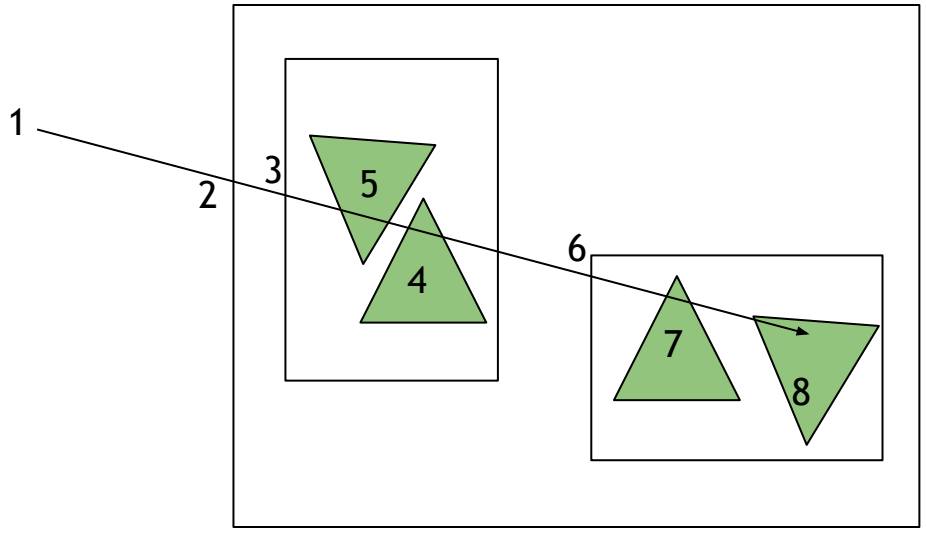


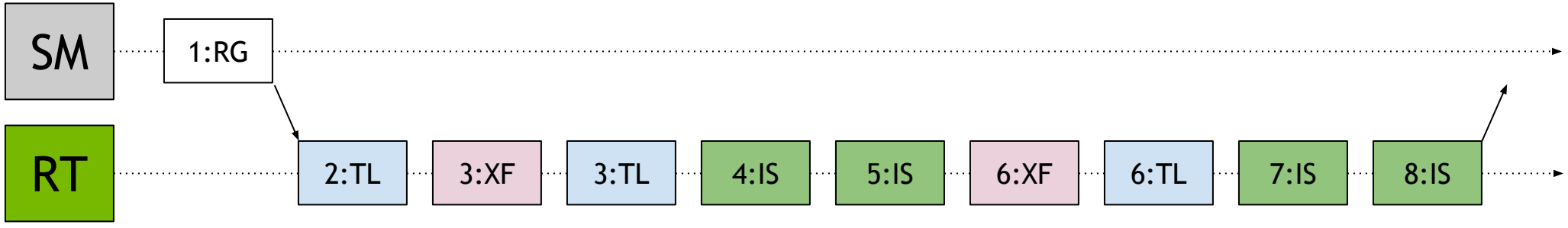
RTX traversal: any-hit



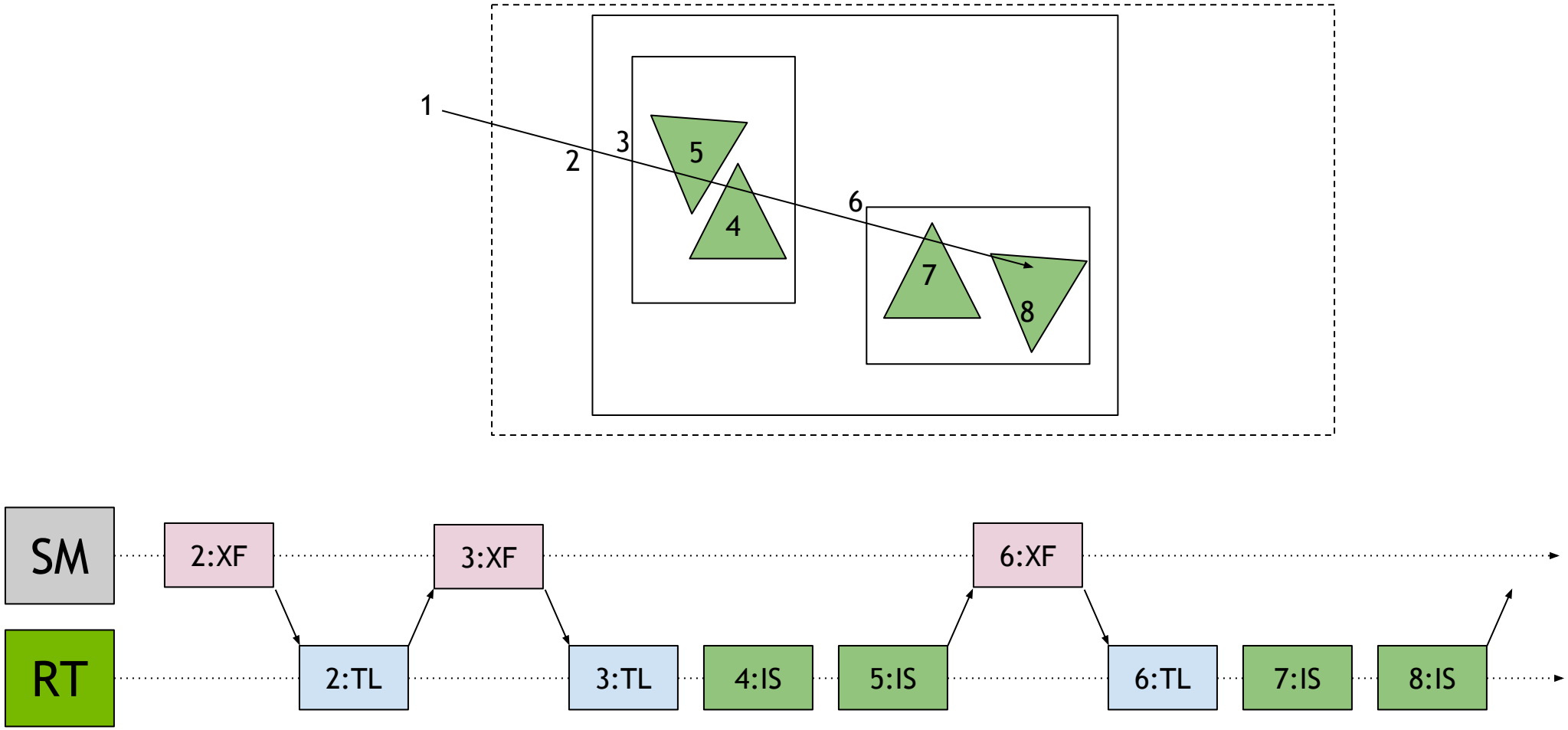


RTX traversal: 1 level instancing (2 lvl scene)

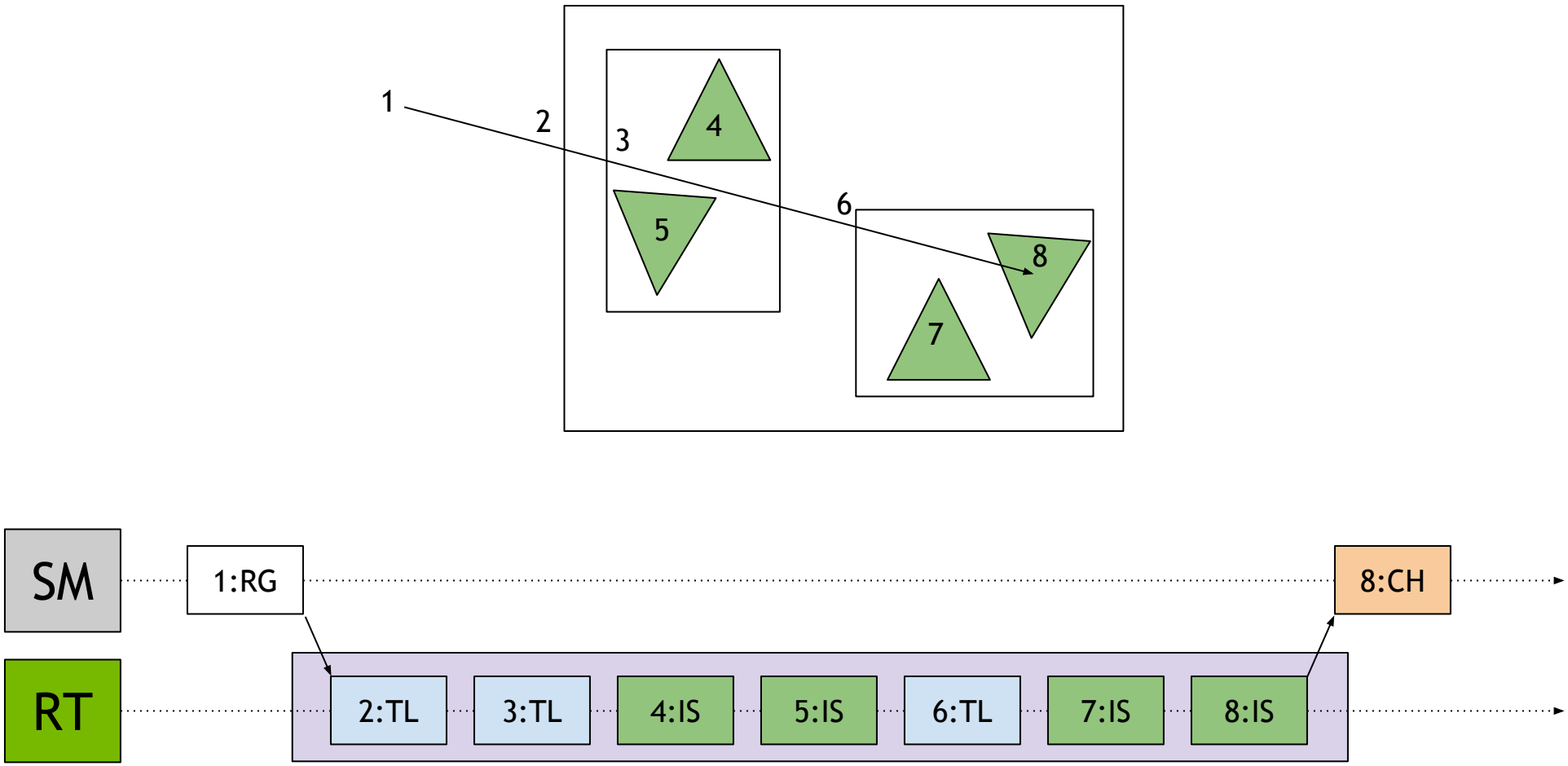




RTX traversal: 2 level instancing (3 lvl scene)



RTX traversal: the ideal



The goal: uninterrupted traversal on RT core

RTX traversal: summary

Built-in triangles + RTX => 2x-10x faster -- first step to gigarays

Features that can impact traversal:

- intersection
- anyhit
- 2+ levels of instancing
- motion blur

Shadow rays tend to be faster. (re-evaluate balance)

Any-hit is opt-out

Disable any-hit, whenever you don't need it

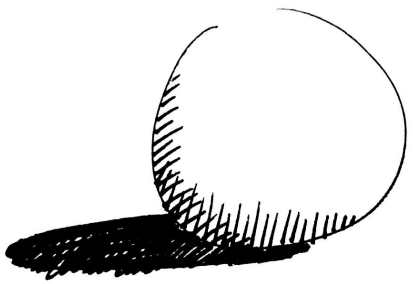
- **`*_DISABLE_ANYHIT`** {instance, geometry, ray}

For shadow-like rays and built-in triangles

- closesthit + **`optixTrace(..., OPTIX_RAY_FLAG_TERMINATE_ON_FIRST_HIT)`**

For shadow-like rays and custom intersection

- anyhit + **`optixTerminateRay()`**



Highest Performance Workloads

Coherent / primary rays

Short rays / small t_max

Simple shading

Big batches

Achieving High Performance in OptiX

Peak RT Core throughput depends on your workload

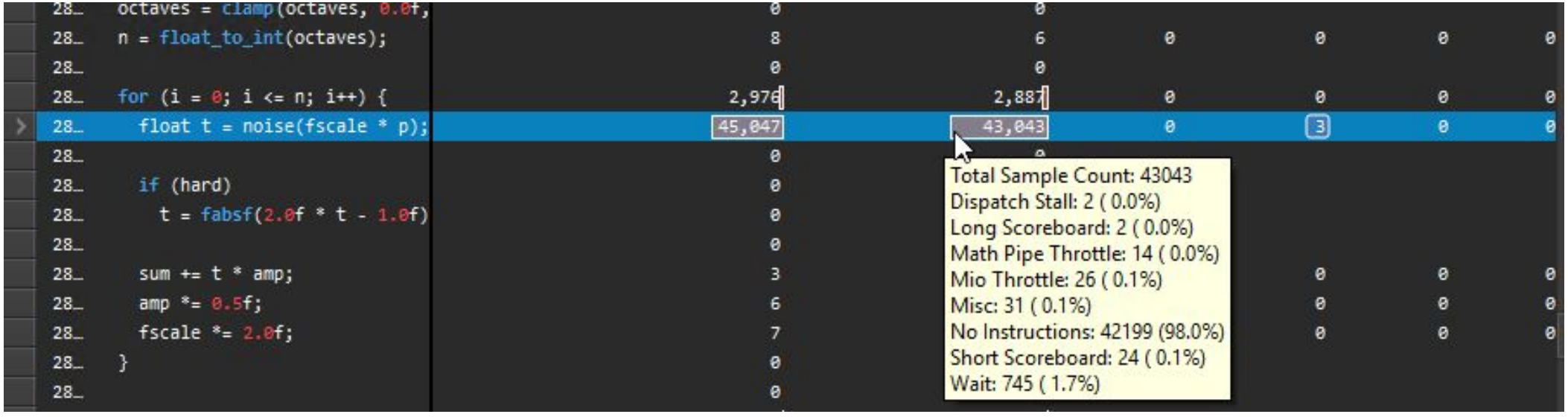
Be judicious with:

- instancing
- shading
- anyhit
- motion blur
- memory

NSight Compute + OptiX

New! NSight Compute can profile OptiX 7 programs.

Visit the NVIDIA booth for a demo.



Comparing Optimized Profiler Run to the Original Baseline Run

SIGGRAPH 19 Blender Optimization Demo - NVIDIA Nsight Compute

File Connection Debug Profile Tools Window Help

Connect Disconnect Terminate Profile Kernel

API Stream

12404 > __raygen__kernel_optix_path_trace_0x43d0f87eb8eca13d_ss_

Next Trigger: raygen

ID	API Name	Detail
6254	cuCtxPushCurrent_v2	
6255	cuMemAlloc_v2	
6256	cuCtxPopCurrent_v2	
6257	cuCtxPushCurrent_v2	
6258	cuMemsetD8_v2	
6259	cuCtxPopCurrent_v2	
6260	cuCtxPushCurrent_v2	
6261	cuMemcpyHtoDAsync_v2	
6262	cuMemcpyDtoDAsync_v2	
6263	cuStreamWaitEvent	
6264	cuOccupancyMaxPotentialBlockSizeWithFlags	
6265	cuOccupancyMaxActiveBlocksPerMultiprocessor	
6266	cuOccupancyMaxActiveBlocksPerMultiprocessor	
6267	cuMemcpyAsync	
6268	cuMemcpyDtoDAsync_v2	
6269	cuCtxGetLimit	
6270	cuCtxSetLimit	
6271	cuMemcpyDtoDAsync_v2	
6272	cuLaunchKernel	
6273	__raygen__kernel_optix_path_trace_0x43d0f87eb8eca13d_	__ray

Untitled 3 * | BlenderBmwOptimized.nsight-cuprof-report

Page: Details Process: All Launch: path_trace_0x3b31debfb7a7d4be_ss_3 Add Baseline Apply Rules Copy as Image

Current 6273 - __raygen__kernel_optix_path_trace_0x3b31debfb7a7d4be_ss_3 (16384, ... **Time:** 34.43 msecond **Cycles:** 49,648,109 **Regs:** 128 **GPU:** Quadro RTX 6000 **SM Frequency:** 1.44 cycle/nsecond **CC:** 7.5 **Process:** [11736] blender.exe

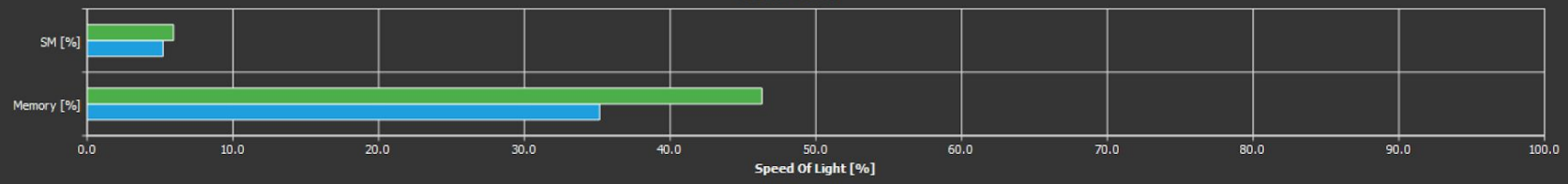
Baseline 1 6273 - __raygen__kernel_optix_path_trace_0x43d0f87eb8eca13d_ss_3 (16384, ... **Time:** 70.50 msecond **Cycles:** 102,394,759 **Regs:** 128 **GPU:** Quadro RTX 6000 **SM Frequency:** 1.45 cycle/nsecond **CC:** 7.5 **Process:** [10640] blender.exe

GPU Speed Of Light

High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.

SOL SM [%]	5.20	(-12.24%)	Duration [msecond]	34.43	(-51.17%)
SOL Memory [%]	35.16	(-24.05%)	Elapsed Cycles [cycle]	49,648,109	(-51.51%)
SOL TEX [%]	5.30	(-30.17%)	SM Active Cycles [cycle]	48,955,574.82	(-51.76%)
SOL L2 [%]	23.99	(+7.91%)	SM Frequency [cycle/nsecond]	1.44	(-0.79%)
SOL FB [%]	35.16	(-24.05%)	Memory Frequency [cycle/nsecond]	1.61	(-1.01%)

GPU Utilization



Recommendations

Bottleneck [Warning] This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at `Scheduler Statistics` and `Warp State Statistics` for potential reasons.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	0.15	(+63.09%)	SM Busy [%]	3.86	(+64.45%)
Executed Ipc Active [inst/cycle]	0.15	(+63.80%)	Issue Slots Busy [%]	3.86	(+64.45%)
Issued Ipc Active [inst/cycle]	0.15	(+64.45%)	-		-

Memory Workload Analysis

All

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/second]	217.06	(-24.81%)	Mem Busy [%]	23.99	(+7.91%)
L1 Hit Rate [%]	24.60	(+162.27%)	Max Bandwidth [%]	35.16	(-24.05%)
L2 Hit Rate [%]	80.51	(+2.34%)	Mem Pipes Busy [%]	5.20	(-12.24%)

Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	3.36	(-3.43%)	Instructions Per Active Issue Slot [inst/cycle]	1	(+0.00%)
Eligible Warps Per Scheduler [warp]	0.04	(+63.53%)	No Eligible [%]	96.10	(-1.58%)
Issued Warp Per Scheduler	0.04	(+65.52%)	One or More Eligible [%]	3.90	(+65.52%)

Best practices in OptiX

When possible…

- Go async, queue up multiple concurrent batches
- Prefer built-in triangles over custom bounds & intersection
- Keep thread workloads similarly sized
- For BVH, try to flatten, merge, compact, refit
- Avoid unneeded SM ⇐ ⇒ RT core round trips
  - Use closesthit (instead of anyhit) with built-in triangles
  - *do* use anyhit for cut-outs
- Pay attention to memory use
  - Minimize memory, payload, attributes, & trace depth
  - esp. dependent memory access

Questions?