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

  ➢ 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
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

Caveat emptor

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:

$$672 \text{ GB/s} \div 10 \text{ GRays/s} = 67.2 \text{ bytes / ray}$$

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)

“Mega” Kernel
RTX traversal: custom primitives

*NB: conceptual model of execution, not timing.*
RTX traversal: built-in triangles

Diagram showing the traversal of triangles with numbers indicating the sequence of triangles visited.
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 $t_{\text{max}}$
- 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
New! NSight Compute can profile OptiX 7 programs.
Visit the NVIDIA booth for a demo.
Comparing Optimized Profiler Run to the Original Baseline Run

![Image of NVIDIA Nightly Compute interface showing GPU speed of light and utilization analysis]

**recommendations**

- Compute Workload Analysis
  - Detailed analysis of the compute resources of the streaming multiprocessors (SMs), 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: L1 cache hits [instruction]: 8.15 (+43.89%), BR busy [Instr]: 2.16 (+44.45%).
  - Executed L2 cache hits: 121.80 (+34.15%), BR busy [Instr]: 23.99 (+39.93%).

- Memory Workload Analysis
  - 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 exceeding the throughput of memory instructions (Mem PPS Busy). Detailed chart of the memory units. Detailed failures with data for each memory unit.
  - Memory throughput [GB/sec]: 217.86 (+34.15%), Mem busy [Instr]: 23.99 (+39.93%).

- 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 Warps). 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: 1.26 (+3.53%), Instructions Per Active Issue Slot [Instruction]: 1 (-0.80%).
  - Eligible Warps Per Scheduler: 0.64 (+1.61%), No Eligible: 90.90 (-1.64%).
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?