Architecture Considerations for Tracing Incoherent Rays

Timo Aila, Tero Karras
NVIDIA Research
Outline

• Our research question:
  • What can be done if memory bandwidth becomes the primary bottleneck in ray tracing?

• Test setup
• Architecture overview
• Optimizing stack traffic
• Optimizing scene traffic
• Results
• Future work
Test setup – methodology

• Hypothetical parallel architecture
  • All measurements done on custom simulator

• Assumptions
  • Processors and L1 are fast (not bottleneck)
  • L1s ↔ Last-level cache, LLC, may be a bottleneck
  • LLC ↔ DRAM assumed primary bottleneck
    • Minimum transfer size 32 bytes (DRAM atom)

• Measurements include all memory traffic
Test setup – scenes

- Simulator cannot deal with large scenes
- Two organic scenes with difficult structure
- One car interior with simple structure
- BVH, 32 bytes per node/triangle

Vegetation
- 1.1M tris
- 629K BVH nodes
- 86Mbytes

Hairball
- 2.8M tris
- 1089K BVH nodes
- 209Mbytes

Veyron
- 1.3M tris
- 751 BVH nodes
- 47Mbytes
Test setup – rays

• In global illumination rays typically
  • Start from surface
  • Need closest intersection
  • Are not coherent

• We used diffuse interreflection rays
  • 16 rays per primary hit point, 3M rays in total
  • Submitted to simulator as batches of 1M rays

• Ray ordering
  • Random shuffle, ~worst possible order
  • Morton (6D space-filling curve), ~best possible order
  • Ideally ray ordering wouldn’t matter
Architecture (1/2)

- We copy several parameters from Fermi:
  - 16 processors, each with private L1 (48KB, 128B lines, 6-way)
  - Shared L2 (768KB, 128-byte lines, 16-way)
  - Otherwise our architecture is not Fermi

- Additionally
  - Write-back caches with LRU eviction policy

- Processors
  - 32-wide SIMD, 32 warps** for latency hiding
  - Round robin warp scheduling
  - Fast. Fixed function or programmable, we don’t care

** Warp = static collection of threads that execute together in SIMD fashion
Architecture (2/2)

- Each processor is bound to an *input queue*
  - Launcher fetches work
- Compaction
  - When warp has <50% threads alive, terminate warp, re-launch
  - Improves SIMD utilization from 25% to 60%
  - Enabled in all tests
Outline

- Test setup
- Architecture overview
- Optimizing stack traffic
  - Baseline ray tracer and how to reduce its stack traffic
- Optimizing scene traffic
- Results
- Future work
Stack traffic – baseline method

• While-while CUDA kernel [Aila & Laine 2009]
  • One-to-one mapping between threads and rays
  • Stacks interleaved in memory (CUDA local memory)
    • 1\textsuperscript{st} stack entry from 32 rays, 2\textsuperscript{nd} stack entry from 32 rays,…
    • Good for coherent rays, less so for incoherent
  • 50% of traffic caused by traversal stacks with random sort!

![Bar chart showing stack traffic for different scenes]
Stack traffic – stacktop caching

- Non-interleaved stacks, cached in L1
  - Requires 128KB of L1 (32x32x128B), severe thrashing
- Keep N latest entries in registers [Horn07]
  - Rest in DRAM, optimized direct DRAM communication
  - N=4 eliminates almost all stack-related traffic
  - 16KB of RF (1/8th of L1 requirements…)

![Graph showing stack traffic comparison for Vegetation, Hairball, and Veyron]
Outline

- Test setup
- Architecture overview
- Optimizing stack traffic
- Optimizing scene traffic
  - Treelets
  - Treelet assignment
  - Queues
  - Scheduling
- Results
- Future work
Scene traffic – treelets (1/2)

- Scene traffic about 100X theoretical minimum
  - Each ray traverses independently
  - Concurrent working set is large
  - Quite heavily dependent on ray ordering
Scene traffic – Treelets (2/2)

• Divide tree into *treelets*
  • Extends [Pharr97, Navratil07]
  • Each treelet fits into cache (nodes, geometry)
  • Assign one *queue* per treelet
  • Enqueue a ray that enters another treelet (red), suspend
    • Encoded to node index

• When many rays collected
  • Bind treelet/queue to processor(s)
  • Amortizes scene transfers
  • Repeat until done

• Ray in 1 treelet at a time
  • Can go up as well
Treelet assignment

- Done when BVH constructed
  - Treelet index encoded into node index

- Tradeoff
  - Treelets should fit into cache; we set max mem footprint
  - Treelet transitions cause non-negligible memory traffic

- Minimize total surface area of treelets
  - Probability to hit a treelet proportional to surface area
  - Optimization done using dynamic programming
  - More details in paper
  - E.g. 15000 treelets for Hairball (max footprint 48KB)
Queues (1/2)

• Queues contain ray states (16B, current hit, …)
  • Stacktop flushed on push, Ray (32B) re-fetched on pop

• Queue traffic not cached
  • Do not expect to need a ray for a while when postponed

• Bypassing
  • Target queue already bound to some processor?
  • Forward ray + ray state + stacktop directly to that processor
  • Reduces DRAM traffic
Queues (2/2)

• Static or dynamic memory allocation?

• Static
  • Simple to implement
  • Memory consumption proportional to scene size
  • Queue can get full, must pre-empt to avoid deadlocks

• Dynamic
  • Need a fast pool allocator
  • Memory consumption proportional to ray batch size
  • Queues never get full, no pre-emption

• We implemented both, used dynamic
Scheduling (1/2)

- Task: Bind processors to queues
- Goal: Minimize binding changes

- Lazy
  - Input queue gets empty → bind to the queue that has most rays
  - Optimal with one processor…
  - Binds many processors to the same queue
    - Prefers L2-sized treelets
    - Expects very fast L1↔L2
    - Unrealistic?
Scheduling (2/2)

- Balanced
  - Queues request #processors
  - Granted based on “who needs most”
  - Processors (often) bound to different queues $\rightarrow$ more bypassing
  - Prefers L1-sized treelets
  - Used in results
Treelet results

- Scene traffic reduced ~90%
  - Unfortunately aux traffic (queues + rays + stacks) dominates
- Scales well with #processors
- Virtually independent of ray ordering
  - 2-5X difference for baseline, now <10%
Conclusions

• Scene traffic mostly solved
  • Open question: how to reduce auxiliary traffic?

• Necessary features generally useful
  • Queues [Sugerman2009]
  • Pool allocation [Lalonde2009]
  • Compaction

• Today memory bw perhaps not #1 bottleneck, but likely to become one
  • Instruction set improvements
  • Custom units [RPU, SaarCOR]
  • Flops still scaling faster than bandwidth
  • Bandwidth is expensive to build, consumes power
Future work

• Complementary memory traffic reduction
  • Wide trees
  • Multiple threads per ray? Reduces #rays in flight
  • Compression?

• Batch processing vs. continuous flow of rays
  • Guaranteeing fairness?
  • Memory allocation?
Thank you for listening!

- Acknowledgements
  - Samuli Laine for Vegetation and Hairball
  - Peter Shirley and Lauri Savioja for proofreading
  - Jacopo Pantaleoni, Martin Stich, Alex Keller, Samuli Laine, David Luebke for discussions