# GPU Coprocessing for Wireless Network Simulation

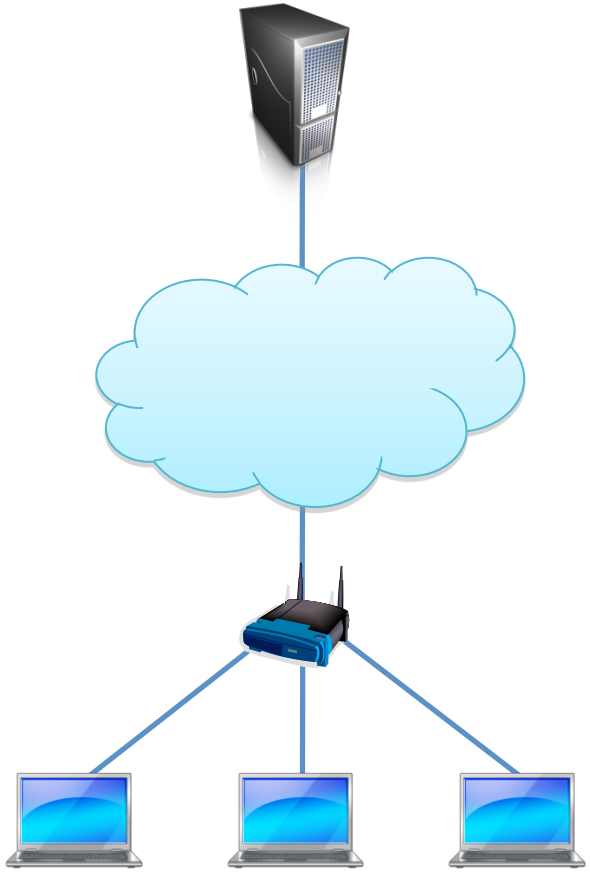
Scott Bai
The MITRE Corporation
sbai@mitre.org

David M. Nicol
University of Illinois at Urbana-Champaign
dmnicol@iti.uiuc.edu



# Network Simulation

- *Goal*: Predict behavior of computer networks
  - Protocols (TCP/IP, 802.11, FTP, etc.)
  - Traffic patterns (web browsing, P2P, VOIP, etc.)
  - Security (attacks and countermeasures)
  - Performance of protocols and topologies under various conditions





# Wireless Network Simulation

- *Problem*: behavior of wireless networks depends on the physical environment
  - Obstacles affect signal transmission
  - Nodes may be mobile
  - Typically looking to compute signal strength and Signal/Noise
    - Decide whether signal is received
- Physical phenomena must be modeled to accurately predict MAC and PHY activity of nodes
- *Problem*: realistic models are computationally infeasible
  - Large-scale simulations use empirical and/or stochastic models
  - Physically realistic models used at smaller scales, e.g. in antenna design

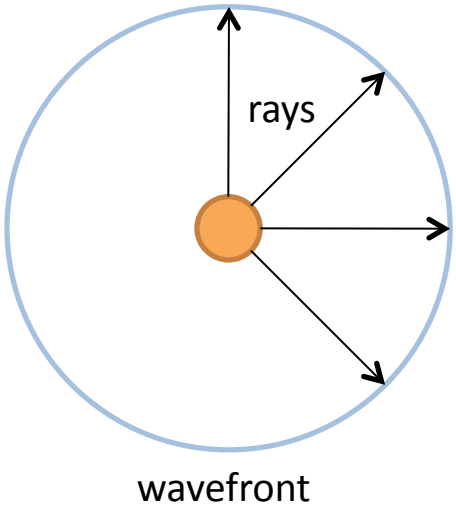


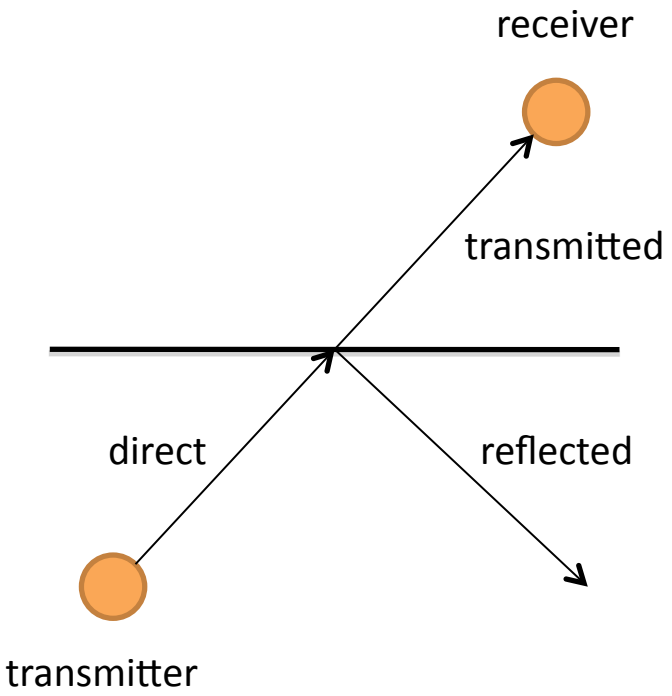
# Modeling the Wireless Channel

- Raytracing can be used to model radio signal propagation
  - Similar to optics: when features in environment are much larger than wavelength, radio wave behavior is quasi-optical
    - High frequency approximation
  - Rays emit from a source, represent directions of radio wave propagation
  - Geometric optics (GO): represent obstacles as piecewise planar surfaces with effective reflection and transmission coefficients $R$ and $T$
  - Rays reflect off or transmit through surfaces
    - Reflections and transmissions recursively generate new rays
    - Reflected and transmitted rays have fraction of incident ray's power, calculated using $R$ and $T$ of surface
  - Can be extended to account for diffraction
    - Geometric Theory of Diffraction (GTD), Uniform Theory of Diffraction (UTD)



# Modeling the Wireless Channel



# Shooting and Bouncing Rays (SBR)

- A brute-force raytracing method
  - *Primary* (direct) rays emit in all directions from point-source transmitter with constant angular separation $\phi$
  - Rays recursively bounce and generate new *secondary* rays using GO or other model
    - Recursion terminates when ray exits scene, reaches a set number of recursions, or becomes weaker than some threshold
  - Ray sequences recursively generated from a primary ray represent paths by which wavefront propagates from transmitter
  - Ray is received (i.e. contributes energy at receiver) when passing through reception circle (with radius determined by $\phi$) around receiver antenna (multiple received rays -> *multipath*)
    - Total distance of ray sequence used to calculate free space path loss, with other losses from reflection or transmission



# Raytracing on GPU

- GPUs already fast for raytracing in graphics
  - Parallel hardware processes thousands of rays concurrently
  - Millions of rays/s, tens of millions for some scenes
  - Need good acceleration structure for speed
- Additional parallelism in wireless network simulation:
  - Typically mobile node positions updated at regular intervals
    - Must recalculate path loss between each active transmitter and each potential receiver
  - Raytracing for each transmitter can be done in parallel
    - Use multiple GPUs and GPU-optimized raytracing algorithms
    - Also use CPUs with traditional raytracing algorithms
    - Assign a transmitter location to each device to raytrace from

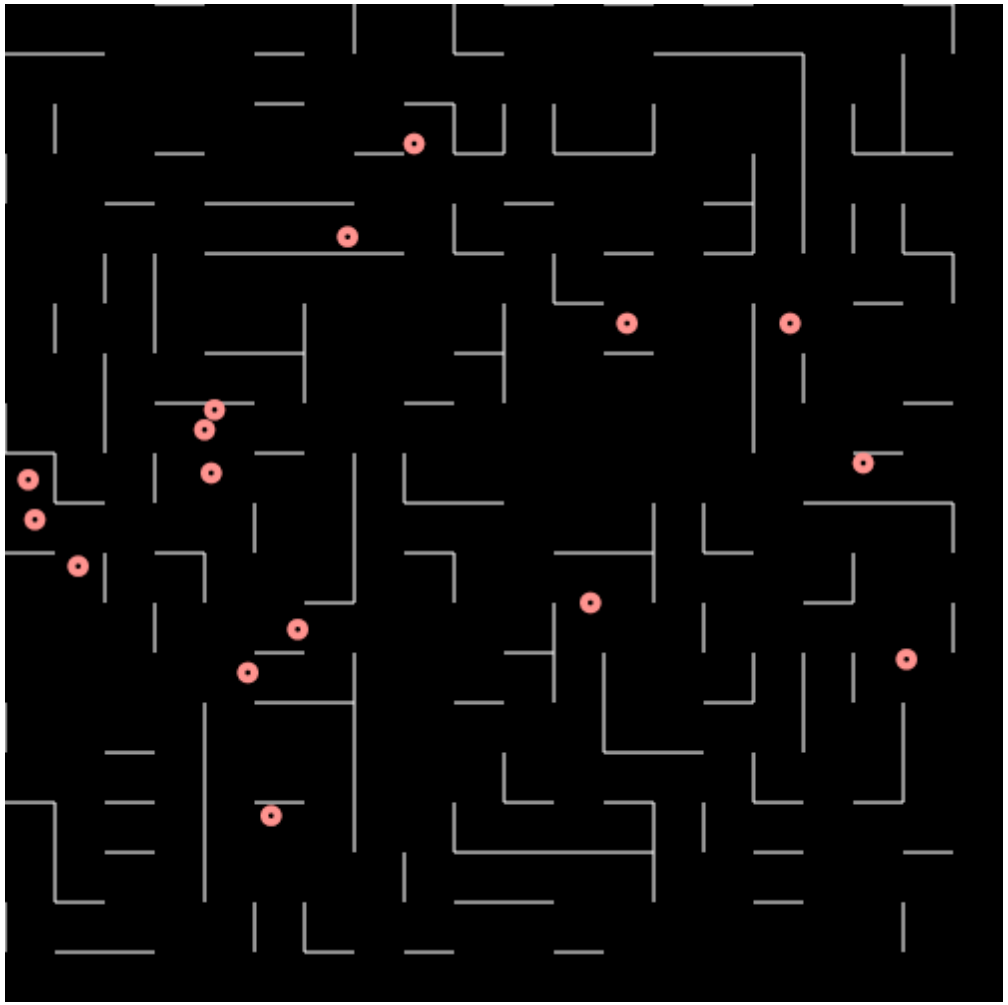


# 2-d SBR on GPU

- Scene geometry (indoor scenario):
  - Represent as line segments (walls, partitions, doors, etc.)
  - Each has *effective* $R$ and $T$ values
    - For real buildings, precise information about building materials, internal structure of walls etc. often unavailable
- Primary rays:
  - Launch in parallel, one per CUDA thread (thousands in total)
  - Allocate global memory for potential child rays
    - Up to two children (one reflected, one transmitted) per parent ray
    - Max amount of memory needed per primary ray determined by max level of recursion
    - One child ray stored for each recursive step (other child processed immediately)



# Illustrative Topology



Uniform density of surfaces

Much smaller than topologies in study



# Multi-GPU

- At simulation initialization, all GPUs receive description of scene geometry (the *environment*)
  - Geometry assumed to remain static throughout simulation
- Each GPU managed by a CPU thread which accesses a *work queue* containing queries from network simulator
  - Each query requests prediction of path loss between a given pair of nodes
  - Prediction results cached until invalidated by node movement
  - Since raytracing from a node calculates received signal at all other nodes, cache entries may be updated opportunistically when not requested by simulator



# Work Assignment

- GPUs and CPU cores service queue concurrently for maximum throughput
  - Threads check for new work when done with current task, sleep on empty queue, wake when new work arrives
  - Heterogeneous devices service queue at different rates according to relative computing capacity
    - Assuming long queue lengths, sufficient to keep all devices busy
    - If queue lengths short (few queries submitted at a time), may be preferrable to assign higher priority to more powerful devices
  - Raytracing task takes variable amount of time to complete
    - Some node locations have complex scenery, others relatively empty
  - Work assignment is coarse – rays from same query/transmitter all traced by same device



# Acceleration Structures

- Naïve raytracing: most time spent testing for ray-obstacle intersections
  - Millions of rays, each tested against thousands to millions of polygons (3d) or line segments (2d)
- Use *k-d tree* or *BVH* to recursively subdivide scene into smaller segments
  - Traverse structure to find nearest object - O(log N)
  - Test only objects in leaf nodes ray passes through
  - Avoids testing most objects
  - Static geometry? *Build acceleration structure only once per simulation*



# k-d Tree Traversal

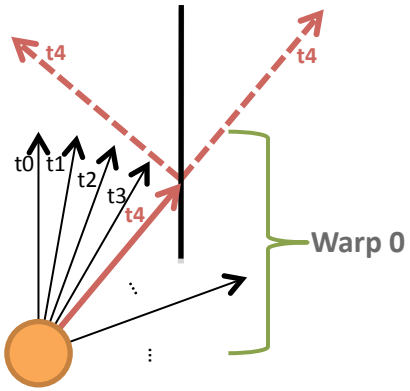
- Traditional method on CPU: use traversal stack
  - Keeps track of which subtrees for ray to recursively traverse next
  - *Problem*: not enough shared memory on GPU to keep stack for each thread
- GPU approaches:
  - Stackless
    - Using *ropes* between nodes to traverse neighboring regions (Popov et al. 2007), fast but uses too many registers for our application (was also an issue for the authors)
    - *kd-restart*: when ray exits a region represented by a leaf node without intersecting any object, move ray origin to region boundary and restart k-d tree traversal – next node traversed will be on other side of boundary, skipping already-visited nodes (Foley et al. 2005)
  - *short-stack*:
    - Constant-sized stack for traversal, fallback to *kd-restart* when stack overflows (Horn et al. 2007), used in our design



# Coherence and Divergence

- CUDA threads organized in warps
  - Best performance when all threads in warp operating in SIMD fashion
- Neighboring threads responsible for neighboring rays
  - Neighboring primary rays very *coherent*:
    - Share same origin
    - Propagate in nearly identical directions
    - Likely to intersect the same objects in scene, traverse k-d tree identically: SIMD
  - Secondary rays less coherent:
    - Different origins
    - Propagate in different directions
    - More incoherent at each recursive step: poor SIMD, different thread lifetimes
    - Performace (rays/s) suffers as max number of recursive bounces increases

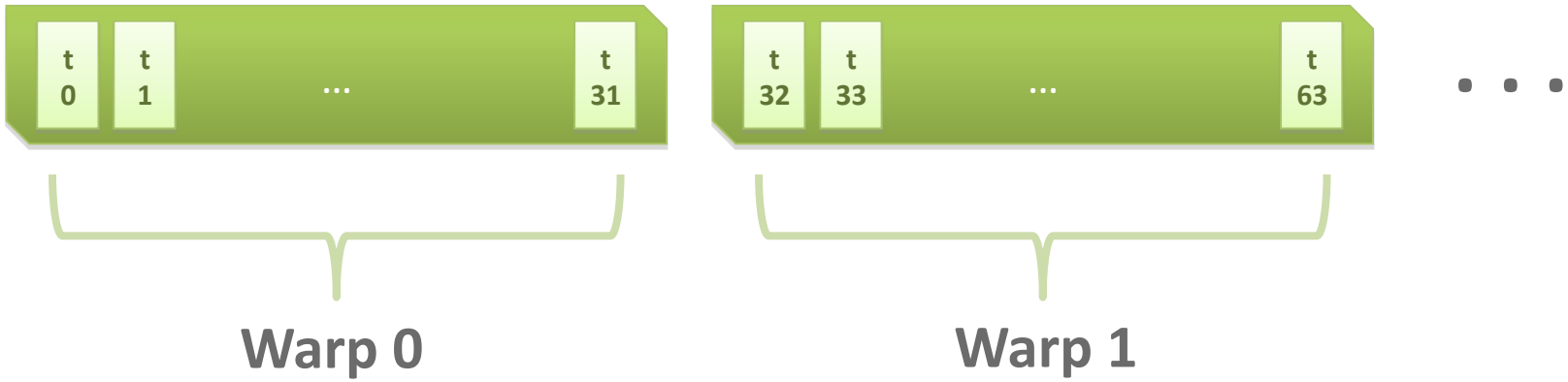





Thread $t_i$ traces the $i^{th}$ primary ray and any children generated from it

Thread $t_4$ traces the $4^{th}$ primary ray and any children generated from it

For best performance, threads in same warp should execute same instructions -- *i.e., their rays should traverse same parts of scene in same order*



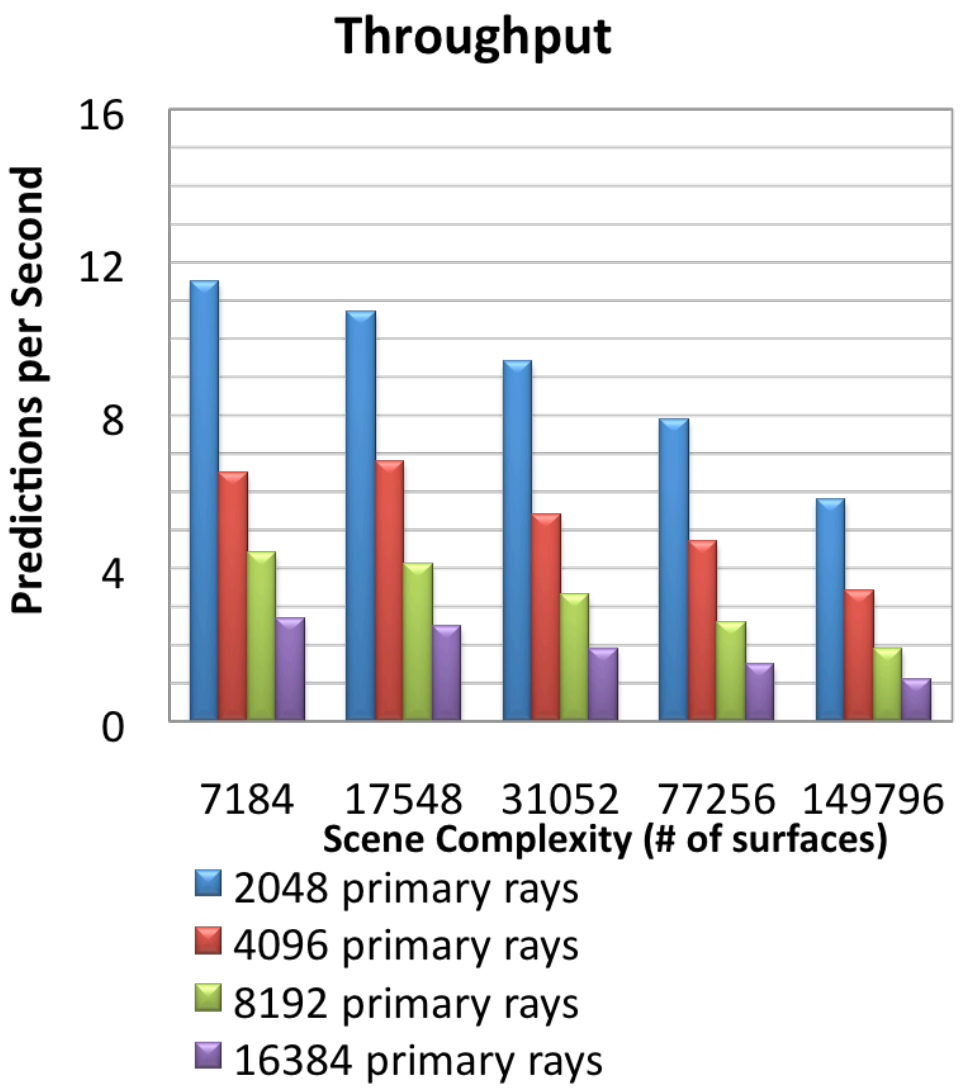



# Coherence and Divergence

- Tradeoff: more recursion -> find more paths but less SIMD
  - Few significant-energy paths found beyond ~8 bounces in our tests
  - At 9 bounces, more than half of all threads already done when longest-running thread is 50% done
  - May be possible to redistribute rays from heavily-loaded threads to ensure concurrency



# Performance




System hardware:

— Intel Core 2 Quad Q9300

— 2x NVIDIA GeForce 9800X2

Using both 9800X2 cards (4 GPUs total) and the quad-core CPU simultaneously, we achieve up to **17 Mrays/s** at 9 bounces

16 transmitters

Each prediction computes the signal strength between one pair of nodes

Doubling primary rays “doubles” resolution

- basic accuracy/speed tradeoff



# Performance

Table I: Performance in Mrays/s of the raytracer in various system configurations

| walls | 1gpu | 4core | 4gpu | 4gpu+4core |
|---|---|---|---|---|
| 149 796 | 2.65 | 1.52 | 9.64 | 10.9 |
| 77 256 | 3.13 | 1.61 | 10.8 | 12.4 |
| 31 052 | 3.62 | 1.71 | 12.0 | 13.6 |
| 17 548 | 3.38 | 1.79 | 12.7 | 14.2 |
| 7184 | 3.69 | 1.97 | 12.8 | 14.4 |

Scene complexity impacts throughput

GPUs significantly accelerate performance

Marginal gain of using multiple CPUs with GPUs



# Questions?



# Backup Slides



# Diffracted Rays

- Needed to avoid discontinuous transition between direct LOS and shadowed regions
- Generate cone of child rays from point of incidence
  - Drastically increases potential memory requirement
- Secondary rays generate weak diffracted rays
  - So diffract only primary (direct) rays
    - Reduces memory footprint
- Increases kernel complexity, register use
  - Reduces concurrency