GPU Coprocessing for Wireless Network Simulation

Scott Bai
The MITRE Corporation
McLean, Virginia, USA
sbai@mitre.org

David M. Nicol
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
dmnicol@iti.uiuc.edu

***Abstract*—Site-specific modeling of wireless communications channels has historically been too computationally intensive to incorporate into commodity network simulators. Simulation cannot accurately predict the behavior of wireless networks in real-world environments without modeling the physical channel realistically. Realistic models typically involve large amounts of floating point computation, to which modern GPUs are well suited. In this paper we demonstrate parallel radio propagation prediction in a single machine using multiple GPUs and CPU cores. We explore the tradeoffs between model accuracy and performance, and use techniques from graphical raytracing to improve the speed with which radio path loss can be computed.**

GPUs; raytracing; geometric optics; propagation modeling; network simulation; parallelism; acceleration; wireless; radio; path loss; threads; CUDA; kd-trees; rays; performance

I. Introduction

Computer networks today are often comprised of thousands or millions of nodes, connected in complex dynamic topologies. The study of such systems is made difficult by their scale and complexity. Full-scale experiments on real systems are expensive, time-consuming, and difficult to repeat or verify. Simulation techniques allow researchers to represent network behavior in software models, which are executed on high-performance computing platforms. Many network simulators can emulate protocol stacks and host behavior down to the packet level. Wireless communications, however, cannot be adequately modeled without realistic models of the physical environment.

Network simulators currently used in academia, such as ns-2 [1], typically use simple propagation models to predict the path losses of radio signals propagating between pairs of nodes. These models require few computations but cannot account for the specific physical structure (i.e., the walls, furniture, terrain, etc.) of a scene. Realistic site-specific models require substantially more computation, and have until recently been infeasible for use with commodity computer hardware.

One class of models is based on raytracing, in which rays representing the directions(s) of wave propagation traverse the paths by which energy is transmitted from a transmitter to the receivers in the scene. This approximation is valid when the dimensions of physical features in the scene are much larger than the wavelength of the signal. Because rays can be traced in parallel, raytracing-based models can be implemented efficiently in Graphics Processing Units (GPUs), which are highly-parallel processors designed for high floating-point performance. Insights from recent research in graphical raytracing on GPUs [2][3][4] and the use of multiple GPUs and CPU cores together can improve performance of raytracing-based radio propagation models.

This work was supported in part by the National Science Foundation under Grant No. CNS-0524695.

II. Propagation Model

A wireless network simulator computes the signal to interference and noise ratio (SINR) at a receiver antenna to determine whether a transmission has been received from the transmitter. A propagation model determines the power of a signal at a receiver due to the transmitter. We use a simplified model based on geometric optics (GO).

GO models represent the geometry of the scene (walls, windows, floors, etc.) as piecewise planar surfaces, which rays may interact with as shown in Fig. 1. GO models have been used to simulate indoor pathloss in buildings as well as outdoor urban pathloss. In two dimensions, a point-source antenna emits a circular wavefront which can be approximated by a set of rays emanating in all directions away from the transmitter location at equal angular intervals [5]. In three dimensions, a spherical wavefront is used [6]. These initial (*primary*) rays then recursively generate new (*secondary*) rays by reflecting from, transmitting through, and/or diffracting at each surface, until their energy has decayed below some threshold or they have reached a set number of recursions.

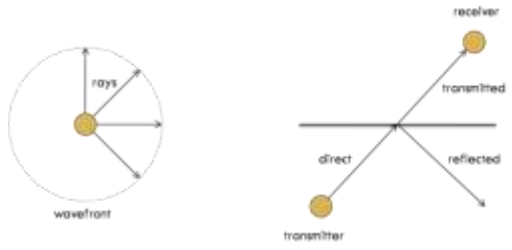

Figure 1. Rays emit from the transmitter and recursively generate new rays when they encounter surfaces.

A *reception circle* (in 2-d) or *sphere* (in 3-d) determines whether a ray has passed close enough to an antenna to be "received" by it. Rays intersecting the circle represent different paths by which the signal can travel from the transmitter to the receiver. The length of each path, as well as the properties of each material the ray path traverses, determine the strength of the received signal components. Received signal components contribute to the signal power reported by the simulator.

III. Parallelization

The wireless propagation problem in network simulation is inherently parallel. First, there are usually multiple, sometimes hundreds of wireless nodes in each wireless network, and potentially several wireless networks in the simulated scenario. Second, the brute-force GO raytracing process for each transmitter involves millions of independent rays, which can be traced in parallel.

To exploit this parallelism, we use a quad-core computer containing two NVIDIA GeForce 9800X2 cards for a total of four GPU devices. Each GPU is programmed using the Compute-Unified Device Architecture (CUDA) framework. A set of CUDA kernels (programs executed on the GPU) perform the raytracing algorithm to compute the signal strength between a given transmitter and all active receivers in the scene.

Our solution is implemented as a set of modifications to PRIME SSFNet [7], a discrete-event parallel network simulator. PRIME updates the locations of all mobile nodes at regular intervals. We maintain a cache of path loss values between each node pair, and values can be reused if multiple transmissions occur before one or both of the nodes change their location. When a node moves, previously computed path loss values for that node are invalidated. If the simulator needs a new path loss value for a given transmitter, it adds a query to a work queue which is served by all CUDA devices detected on the system.

Each GPU is managed by a CPU thread which checks the work queue when its corresponding GPU is idle. The GPUs may be heterogeneous and service the queue at different rates according to their computational power. The GPU assigns primary rays to each of its threads and performs raytracing to calculate the path loss to each receiver. We also use CPU raytracing threads to service the work queue. All raytracing threads become inactive when the work queue is fully emptied, and remain so until new queries are queued.

IV. Raytracing

Most research in GPU raytracing has been in the context of computer graphics, in which raytracing is a key rendering method. Several recently-developed techniques for graphical raytracing are also applicable to our geometric optics implementation. In particular, raytracing performance is greatly improved through the use of acceleration structures, which organize the scene geometry to improve the speed of detecting ray-object intersections, where “object” means a polygon in 3-d or a line segment in 2-d. In radio propagation modeling as in graphics, we must find the nearest object intersected by a ray to determine where new reflected and transmitted rays will be generated.

The number of intersection tests is $N_R \times N_O$, the number of rays multiplied by the number of objects, often billions of operations or more. Instead of testing a ray against each object to find the nearest intersection, an accelerated raytracing algorithm first recursively subdivides the scene as shown in Fig. 2. The ray then traverses the acceleration structure to quickly identify the nearest intersection, eliminating as many objects as possible from consideration. Recent work with *kd-trees* [2][3] and *bounding volume hierarchies* (BVHs) [4] show good performance on the GPU.

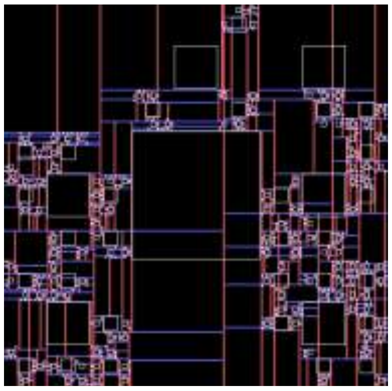

Figure 2. A 2-d scene recursively subdivided by a kd-tree, with splitting planes shown in color.

CPU-optimized recursive traversal algorithms do not work efficiently for acceleration structures on the GPU. We use instead the *short-stack* algorithim of [2] (originally implemented on ATI hardware) to traverse a kd-tree structure stored in global memory. A constant sized short stack is used to record a thread's state as it recursively traverses the kd-tree to find the nearest object. We choose the size of the stack so that all threads in a CUDA block can have one in shared memory. If traversal overflows the stack, the *short-stack* algorithm restarts from the kd-tree root but advances the ray's origin forward to skip over regions that have already been tested for intersections.

Table I shows the time in seconds our 2-d propagation model takes to compute a single query (i.e. to compute path loss between a transmitter and all receivers) with one GPU using the *short-stack* traversal method at various levels of scene geometry complexity. All scenes contain 16 randomly placed wireless nodes.

All results were obtained with 8192 primary rays emitted from the transmitter and a maximum recursion level of 9 for the raytracing algorithm. In the simplest scene, containing 7184 objects, we obtain 3.69 million rays per second. This is similar to results reported for 9-bounce specular rays in [2] using *short-stack* for rendering graphics (however, we used different test scenes and results are not directly comparable since the application areas differ). A query completes in 1.2 seconds. In our experiments, raytracer performs better in sparser, non-uniform scenes because such scenes contain large empty regions which can be represented using fewer kd-tree nodes, allowing rays to traverse the tree faster.

TABLE I. Single GPU performance

Number of objects	**Raytracing rate (Mrays/s)**	**Execution time (s)**
149,796	2.65	3.2
77,256	3.13	2.4
31,052	3.62	1.7
17,548	3.38	1.3
7184	3.69	1.2

V. Concurrency and Coherence

Each CUDA thread in is responsible for one primary ray and all secondary rays generated by that ray. Our raytracing kernel uses 36 registers per thread, meaning that a multiprocessor with 8192 registers can accomodate seven active warps of 32 threads each. Using the simpler but slower *kd-restart* [8] algorithm consumes 32 registers per thread. Using the stackless kd-tree traversal of [3] consumes over 40 registers.

We determined that modeling diffraction, which generates a cone of rays at the point of diffraction of a ray, would drastically increase the memory usage of the raytracer. Because each ray may generate several diffracted rays, the number of rays that may need to be stored per thread is quite large, limiting the number of threads that can be safely spawned without exceeding the available memory. By ignoring diffraction, we ensure that only two rays (a reflected and a transmitted ray) are generated per iteration of the raytracer by each thread, and can limit the amount of storage needed for a given number of threads by setting the maximum recursion level appropriately. An alternative would be to model diffraction only for primary rays, since secondary rays produce weak diffracted rays that are usually insignificant in power.

CUDA threads execute in warps of 32, which perform best when operating in lockstep. In raytracing, this coherence occurs when neighboring rays assigned to the same warp travel in similar directions and intersect the same objects in the same sequence. As rays travel away from their source, they separate from one another as the wavefront spreads out. Child rays are less likely to be coherent than their parents, so as recursive raytracing proceeds, neighboring threads are less likely to execute the same instructions in lockstep [2]. Furthermore, some rays exit the scene quickly while others bounce around many times, generating many children. In our tests at 9 levels of recursion, over half of all threads are idle when the longest-running thread reaches 50% of its execution time. There is a tradeoff between the length of the signal paths that can be captured by the model and the coherence of the rays and their threads – i.e., a tradeoff between accuracy and performance. In all our test scenes, increasing the maximum recursion level beyond 9 produces no appreciable difference in the total received signal strength predicted by the model. For simpler scenes with few obstacles, 4 or 5 levels of recursion are often sufficient to capture all significant paths.

VI. Results

Table II shows the query throughput for the same test scenes as in Table 1, but with all four GPUs and the quad-core CPU working simultaneously to service the queue. Each GPU alone traces rays approximately twice as quickly as the Intel Core 2 Quad processor. On average, the full system answers queries over five times faster than a single GPU. In the scene with 7184 objects, 4.4 queries are answered per second when a primary ray resolution of 8192 is used, as compared to 0.83 queries per second using a single GPU. Doubling the number of primary rays emitted from the transmitter increases execution time by somewhat less than a factor of 2, even though this approximately doubles the total number of rays traced. This implies that more closely-spaced rays are more coherent and produce better SIMD behavior among CUDA threads in each warp.

TABLE II. Total System Throughput in Queries per Second at Various Primary Ray Resolutions

Number of objects	2048	4096	8192	16384
149,796	5.8	3.4	1.9	1.1
77,256	7.9	4.7	2.6	1.5
31,052	9.4	5.4	3.3	1.9
17,548	10.7	6.8	4.1	2.5
7184	11.5	6.5	4.4	2.7

We have shown that GPUs can perform raytracing at high speeds to facilitate propagation modeling based on geometric optics. We have also shown that multiple GPUs can be combined with multicore CPUs to produce higher throughput in answering path loss queries. By using a centralized queue of work to be performed, the network simulator can distribute queries among the available computing devices in the system. As a self-contained library, our propagation modeling software can be linked into network simulators and used whenever GPU accelerators are present. GO-based propagation modeling is inherently more complex than simple models like free-space and two-ray, but raytracer performance on GPUs continues to improve thanks to strong interest in the graphics community. Wireless network simulation using non-trivial physical models is becoming more practical than ever thanks to the prevalence of high-performance parallel accelerators like GPUs.

References

[1] Information Sciences Institute, “The Network Simulator ns-2: Documentation,” June 2009. [Online]. Available: http://www.scalable-networks.com/publications/documentation.

[2] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, “Interactive k-d tree GPU raytracing,” in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, New York, NY, 2007, pp. 167–174.

[3] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, “Stackless kd-tree traversal for high performance GPU ray tracing,” *Computer Graphics Forum*, vol. 26, no. 3, pp. 415–424, Sep. 2007.

[4] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek, “Realtime ray tracing on GPU with BVH-based packet traversal,” *IEEE Symposium on Interactive Ray Tracing*, 2007, pp. 113–118, Sep. 2007.

[5] W. Honcharenko, H. Bertoni, J. Dailing, J. Qian, and H. Yee, “Mechanisms governing UHF propagation on single floors in modern office buildings,” *IEEE Transactions on Vehicular Technology*, vol. 41, no. 4, pp. 496–504, Nov. 1992.

[6] S. Seidel and T. Rappaport, “Site-specific propagation prediction for wireless in-building personal communication system design,” *IEEE Transactions on Vehicular Technology*, vol. 43, no. 4, pp. 879–891, Nov. 1994.

[7] J. Liu, S. Mann, N. Van Vorst, and K. Hellman, “An open and scalable emulation infrastructure for large-scale real-time network simulations,” *26th IEEE International Conference on Computer Communications*, pp. 2476–2480, May 2007.

[8] T. Foley and J. Sugerman, “Kd-tree acceleration structures for a GPU raytracer,” in *Proceedings of the ACM SIGGRAPH / EUROGRAPHICS Conference on Graphics Hardware*, New York, NY, 2005, pp. 15–22.