Ad Hoc & Sensor Wireless Networks, Vol. 24, pp. 109–134
Reprints available directly from the publisher
Photocopying permitted by license only

CU-Simulator: A Parallel Scalable Simulation Platform for Radio Channel in Wireless Sensor Networks

LIHENG JIAN[1], YING LIU[2,3] AND WEIDONG YI[1]

[1]*School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, 100049, Beijing, China*
[2]*School of Computer and Control, University of Chinese Academy of Sciences, 100049, Beijing, China*
[3]*Research Center on Fictitious Economy and Data Sciences, Chinese Academy of Sciences, 100080, Beijing, China*
Email: jian.9613@gmail.com, {yingliu, weidong}@ucas.ac.cn

Received: January 29, 2013. Accepted: September 5, 2013.

Due to the computational intensive nature, the current available WSN simulators, which are based on the traditional CPU computing architecture, cannot run in a linear scalability. In this paper, we propose and set up *CU-Simulator,* a parallel radio channel simulator to enhance the performance for simulating data packet transmission in WSNs using NVIDIA's CUDA-enabled GPU parallel computing architecture. First, the node positions are simulated on GPU. Second, we propose an efficient data structure for acceleration, called *CUDA-quad-trees*, residing in the fast on-chip memory of GPU, to organize sensor nodes in such a manner that the detection of possible transmitters is facilitated. Third, a CUDA parallel radio channel simulating engine is established. Experimental results show that *CU-Simulator* has a super-linear scalability and greatly outperforms a CPU implementation with up to 452.07-times speedup on an HP Z800 workstation with a NVIDIA Tesla C2070 card and an Intel Xeon Core-quad CPU.

Keywords: Wireless sensor network, radio channel simulation, node position, quad-tree, CUDA, parallel computing, scalability.

1 INTRODUCTION

A wireless sensor network (WSN) [1] is composed of small and independent sensor nodes, which collect information from the environment where they are deployed. Each node is equipped with a short-range radio transceiver. A network is formed through the communication between the nodes. Communication protocols, OS and applications run on the nodes to make a WSN work efficiently. WSNs have broad applications, such as military target tracking and surveillance, natural environment exploration, animal tracking, health monitoring, and business inventory monitoring. WSNs are becoming more and more popular in the recent years, because nodes are smaller, cheaper, and more powerful than ever.

Software simulation is a popular auxiliary approach for on-node software development in the study of WSNs, similar to the studies in other wireless networks. Simulation of data packet transmission in wireless medium (radio channel) is the most restrictive factors in simulation [2]. The computation of determining the possible receptors for each packet transmission of a node is costly, and it repeats endlessly during the simulation, which takes most of the computation. Moreover, the more accurate the radio model is, the more cost it takes. In addition, the number of nodes in a real-world deployment has been up to 2900 [3]. A large number of nodes aggravate the burden of simulation seriously, which leads to a poor performance, not to mention scalability.

In sequential simulators, two approaches are adopted to improve the performance. One approach is to eliminate the simulation of data packet transmission in radio channel partially or completely, because high-level protocols only care the output of lower layers, and radio channel will not produce a direct effect on them [4-6]. The other approach is to develop optimization techniques. To accelerate the computation in simulation, the computation for data packet transmission is cut down by reducing the search for possible receptors [7]. However, the compute-intensive and memory-hungry nature doesn't change. When the network is large, the performance of the simulation is not tolerable at all. As a result, neither of the above solutions provides a scalable solution. Parallel and distributed simulation techniques have been discussed, however, there is still no parallel WSN simulator yet, due to the complexity of data exchange between different processors.

Graphics Processing Unit (GPU) is now a flourishing solution for parallelism. It is originally a kind of highly specialized processors designed for graphics rendering. Nowadays, its architecture has evolved towards general-purpose parallel computing, and high-level languages have also emerged to support easy programming on GPUs. NVIDIA provides Compute Unified Device Architecture (CUDA) architecture with standard C-like interface to manipulate its GPUs [8]. GPU now provide tremendous memory bandwidth and computing power. For example, Tesla C2070 can achieve a bandwidth of 144 GB/s, a double-precision peak performance of 515 Gflops/s and a single-precision peak performance of 1.03 Tflops/s [9]. Low cost is another highlight of GPU, only around $1,900 for a Tesla C2070. Although it is low in cost, its computing power is equivalent to a medium-sized supercomputer which is orders of magnitude more costly. It provides the medium-sized business and individuals great opportunities to afford supercomputing facilities. As a result, there has been an evident trend to accelerate computational intensive applications on a GPU+CPU heterogeneous system, where the GPU acts as the computation accelerator; and this architecture has been widely used in communication, military, business, medical and other domains.

In this paper, we propose and implement *CU-Simulator*, a parallel scalable radio channel simulation platform based on the CUDA-enabled GPU parallel computing architecture. A novel data structure, called *CUDA-quad-trees*, is developed to accelerate the detection of possible transmitters, which takes advantage of the fast on-chip shared memory of GPU. A statistical model for data packet transmission in WSNs is applied to determine data packet reception by inspecting the found possible transmitters. *CU-Simulator* runs on GPU, and interacts with the simulation of on-node native code on CPU through limited data exchange. Experiments are performed on an HP Z800 workstation with a NVIDIA Tesla C2070 card and an Intel Xeon Core-quad 2.93 GHz CPU. Compared with the radio channel simulation of a CPU-based serial WSN simulator, our proposed *CU-Simulator* achieves up to 452.07-times speedup, showing good performance with super-linear complexity.

The contribution of this paper can be summarized in two aspects: 1) An in-memory tree structure on GPU, called *CUDA-quad-trees,* is proposed, which resides in the high-speed on-chip memory of a GPU not only in the construction stage but also in the search stage; 2) Different with existing simulation frameworks of radio channel, a parallel search-based computing engine is proposed, which inspects the possible transmitters for each receptor to accomplish simulation of data packet transmission in a WSN.

The rest of this paper is organized as follows: Section 2 gives a brief introduction to the GPU architecture and CUDA programming model, then presents the related research works. Section 3 outlines our parallel simulation platform for radio channel in WSNs, which is based on the GPU+CUDA parallel computing architecture. In Section 4, CUDA-based random number generation and nodes simulation are presented. In Section 5, we propose a tree-based data structure for acceleration, *CUDA-quad-trees*. In Section 6, our parallel search-based simulation engine for radio channel is described. Scalability analysis and experimental results are presented in Section 7 and we summarize our work in Section 8.

2 BACKGROUND KNOWLEDGE AND RELATED WORKS

2.1 NVIDIA's CUDA-enabled Parallel Computing

Nowadays, GPUs have evolved into a highly parallel, many-core processor. The peak floating-point capability of GPU is an order of magnitude higher than that of CPU, as well as the memory bandwidth. In addition to rendering process, they are also suitable for general compute-intensive, highly parallel computation. In NVIDIA's nomenclature, CPU is referred as the "host", CUDA-enabled GPU is referred as the "device". The host sees a CUDA device as a many-core co-processor, and controls it to accomplish the dispatched computational tasks.

2.1.1 GPU Architecture

At hardware level, a CUDA-enabled GPU consists of a set of single instruction multiple data (SIMD) stream multiprocessors (SMs) with 32 stream processors (SPs) each. For example, a Tesla C2070 has 14 SMs, totally 448 SPs. Each SM contains a small but fast on-chip *shared memory*, which has very low access latency (1-2 clock cycles) and is shared by all of its SPs as shown in Figure 1. Shared memory is managed explicitly by the programmers. It also contains a read-only *constant cache* which is shared by its SPs. A set of local 32-bit *registers* is available for each SP. The SMs communicate through *global/device* memory. Global memory is large but has high access latency (400-800 clock cycles). It can be read or written by the host, and is persistent across kernel launches by the same application.

2.1.2 CUDA Programming Model

At software level, CUDA model is a collection of threads running in parallel. A unit of work issued by the host to the device is called a *kernel.* A CUDA program is running in thread-parallel fashion. Computation is organized as a *grid* of *thread blocks* which consists of a set of threads as shown in Figure 2. A thread block is a batch of SIMD-parallel threads, which runs on the same SM at a given moment. Each SM executes one or more thread blocks concurrently. At instructional level, 32 consecutive threads in a *thread block* make up the minimum unit of execution, called a *thread warp*. For a given thread,

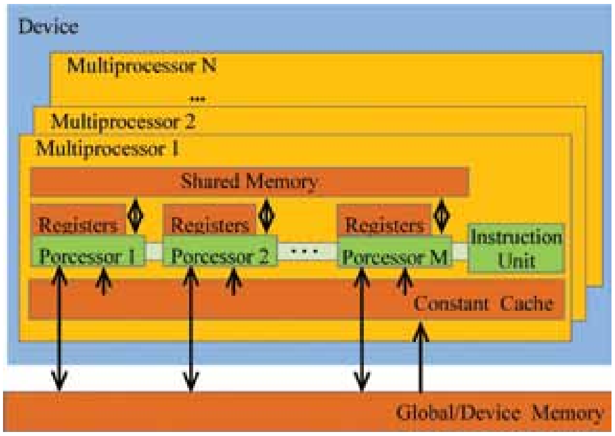

FIGURE 1
A set of SIMD stream multiprocessors with memory hierarchy.

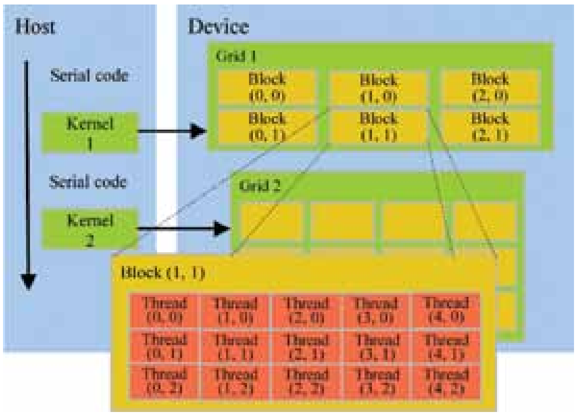

FIGURE 2
Serial execution on the host and parallel execution on the device.

its index determines the portion of data to be processed. Threads in a single block communicate through the shared memory.

CUDA consists of a set of C language extensions and a runtime library that provides rich APIs for non-graphical applications. Thus, CUDA programming model allows the programmers to better exploit the parallel power of GPU for general-purpose computing using the infrastructure.

2.2 Traditional WSN Radio Channel Simulation

In radio propagation of a WSN, wireless electromagnetic waves carry data packets which are the information a node sends to other nodes. Since signal strength of radio waves attenuates during radiation, a data packet can be received by a node only if the strength of the radio waves arrives at the transceiver is larger than a reception threshold. We refer this kind of nodes as *communication reachable* nodes in this article.

The simulators for regular wireless networks, such as NS-2 [10], OMNeT++ [11], cannot simulate native code, and their radio models may not be accurate [12]. In most of WSN oriented simulators, such as TOSSIM [4], ATEMU [5], Avrora [6], VisualSense [13], the size of a network can be up to 10,000, where the simulation of radio channel is omitted or very simple, however. Naoumov et al. [7] proposed two partitioning-based approaches to reduce the search for possible receptors in simulation. One is to divide the simulation area into a grid of cells. Only the cells spatially close to the transmitters take part in computation. The other is similar to the former in principle, but the neighbors of the transmitters are firstly searched by X-coordinate or Y-coordinate of the nodes, whereby two rough neighbor sets are produced, then the possible receptors is determined by taking intersection of the two sets. However, the performance is sensitive to full interference and the mobility of nodes. Eseban et al. [2] investigated WSN simulators and elaborated the scalability issue in WSNs. For each transmission of a transmitter, the relationship between the transmitter and the possible receivers (usually all the nodes in a network) is maintained for reception checking, which consumes excessive computational and memory resource. Since simulation almost involves the computation of the relationships between each pair of nodes in a network, the large number of nodes and requirement of credible results result in long execution time and huge memory cost.

COOJA [14] is one of the mainstream WSN simulators. The on-node native code can execute in COOJA, and communicate with the simulator through java JNI calls. Four models are available, from Unit Dist Graph Medium model (the standard model) to Multi-path Ray-tracer Medium model (a more complicated one with the consideration of full interference). Up to now, it is the most detailed radio channel simulation in WSNs. A simulation can be configured with a fast but simple or slow but precise model as needed.

2.3 Statistical Models for Radio Channel in WSNs

A statistical model is a good approach to study radio channel, and widely adopted in WSNs. It synthesizes large-scale radio variations (*distance attenuation* and *shadow fading*) and small-scale radio variation (*multi-path fading*) into mathematical expressions, concentrating on the effect of radio propagation on upper communication protocols. Since the parameters are obtained from a real scenario and it is validated by in-situ measurement, it is considerably accurate. Moreover, its corresponding computation is little or medium and acceptable. Baccour et al. [15] gave a survey of the current development of statistics-based radio channel models in WSNs, and classified the existing research into three categories, PRR-based [16, 17], which counts Packet Reception Rate, RNP-based [18, 19], which counts Required Number of Packet retransmissions, and Score-based [20, 21], respectively.

2.4 Simulation of Radio Propagation in Wireless Networks using GPUs

There are some works on simulating radio channel in regular wireless networks with a physical model using GPUs, where radio signal is far more stronger than that in WSNs. Abdelrazek et al. [22] developed an architecture to accelerate simulation of multi-path fast fading channels. The extensive channel simulation is offloaded to GPU, and CPU is responsible for advancing simulation time and processing node events. It obtained about 30 times speedup compared to a regular implementation on an Intel Core-duo CPU. Scott et al. [23] simulates radio propagation in a floor-plan scene using a system with multiple GPUs and a multi-core CPU. Similar to [22], GPUs take the computation task of ray tracing-based radio propagation. A KD-tree is used to manage the objects in a simulation scene and eliminate the unnecessary inspection of ray intersection. Experiments showed a 17-times speedup over a quad-core CPU implementation. The simulation system is encapsulated into a library and can be linked into an existing discrete-event network simulator. It is close to reality to model radio channel physically in the form of radiation, reflection, refraction, diffraction, and scattering. However, it is computation-hungry to obtain fidelity, as shown in [22, 23] for a single transmitter.

2.5 CUDA-based Trees

Quad-tree [24] is a classical hierarchical data structure, where each non-leaf node has at most four child nodes. It is an efficient approach for spatial data organization, and is widely used to search for neighbors in image processing, geographic information systems, robotics, etc.

CUDA computing model is announced to be not suitable for irregular computation [8]. Thus, little work has been conducted to build trees on GPU using CUDA. In a hybrid approach for quad-tree construction [25], the top levels of a tree are built on CPU, and then data is transferred to GPU to construct the remaining levels. Best performance is 1.0038 seconds when the number of point is 1,000,000. However, no formal publication or code is available. Zhou et al. [26] proposed an algorithm for constructing KD-tree on GPU, which builds the tree nodes in a breadth-first manner to fully exploit the fine-grained parallelism of GPU. Global memory is allocated for tree nodes. Experiments on ray tracing and K-nearest neighbors search showed that the algorithms are up to 10 times faster than their corresponding CPU implementations. KD-tree in [23] also resides in global memory, and a short-stack residing in on-chip memory is used to maintain the traversal path of each thread. It is not suitable for a large network since the size of the traversal stack is limited by the size of shared memory.

In simulation of the evolution of galaxies, the computation involves finding a large number of neighbors, which is similar to radio channel simulation in WSNs. In the worst case, it has to consider all the cosmic bodies, which will incur excessive computation. Martin and Keshav [27] built an octree on NVIDIA's GPU in simulation. Their Barnes Hut n-body simulation algorithm spent 5.2 seconds in simulating one time step with 5,000,000 bodies on a 1.3 GHz GPU, which is 74 times faster than an optimized serial implementation on a 2.53 GHz Xeon Core-quad CPU. The tree resides in global memory. Since trees are irregular data structures in terms of CUDA's memory access pattern, and the access to global memory incurs high latency, neither building nor searching a tree residing in global memory can obtain efficient memory access.

3 SYSTEM FRAMEWORK

In this section, we will describe our proposed *CU-Simulator*, including what components it contains, how it accomplishes the radio channel simulation, and how it interacts with the simulation of on-node native code.

3.1 Overview of CU-Simulator

Since different WSNs deployment environments put different effects on wireless communication between nodes, our proposed radio channel simulation is oriented toward typical WSNs scenarios. In our simulator, we target the simulation of a WSN scenario as an abstract 2-dimensional field. The nodes in a network are deployed randomly or customized throughout the simulation area, static or moving as requested.

Unlike traditional sequential simulators, *CU-Simulator* places most computation task of a simulation on GPU to achieve performance enhancement. There are two functional components in our proposed simulator. 1) Position simulation generates positions of nodes in the first run of *CU-Simulator* and is able to adjust the positions dynamically in the subsequent simulation. 2) Data transmission simulation simulates data transmission between the nodes for a short time interval in parallel using a search-based inspection, which is different from existing simulations [14] [23]. We propose *CUDA-quad-trees*, which can reside in the fast on-chip memory of GPU, to organize the nodes in the former and speed up the detection of possible transmitters in the latter.

As shown in Figure 3, our proposed process is straightforward. First, the positions of the nodes in a network are initialized and then used to construct

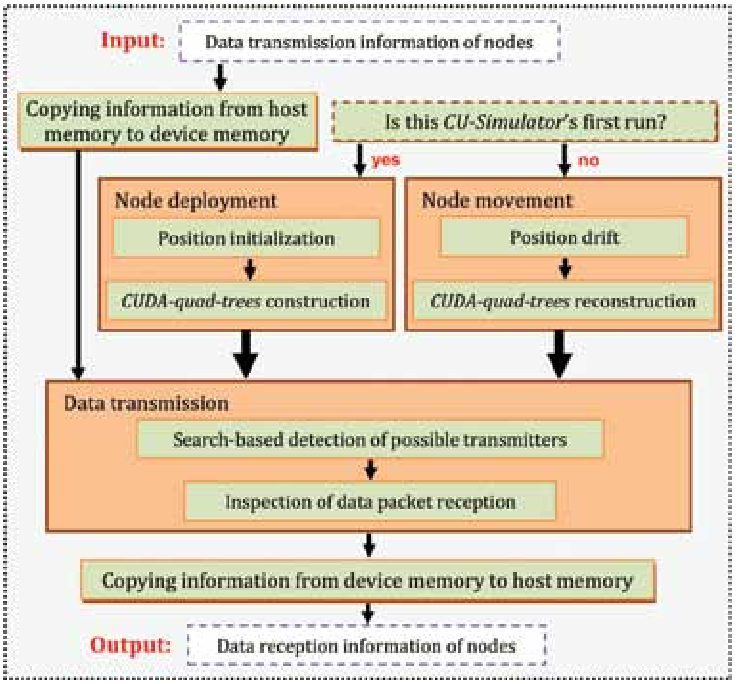
Input:
Data transmission information of nodes
Copying information from host memory to device memory
Is this CU-Simulator's first run?
yes
no
Node deployment
Position initialization
CUDA-quad-trees construction
Node movement
Position drift
CUDA-quad-trees reconstruction
Data transmission
Search-based detection of possible transmitters
Inspection of data packet reception
Copying information from device memory to host memory
Output:
Data reception information of nodes

FIGURE 3
Framework and flow diagram of *CU-Simulator*.

CUDA-quad-trees if *CU-Simulator* is invoked for the first time; if a node movement is requested to be simulated, the node movement module will be invoked in later runs, and then *CUDA-random-bit-moving* is used to simulate the moving behavior of the nodes, and the newly generated positions will be used to reconstruct the CUDA-quad-trees. Second, for each node as a receptor, detect the spatially possible *communication reachable* transmitters by searching *CUDA-quad-trees*, and then the communication between each found potential transmitter and this node is inspected to determine its final data reception. Each CUDA thread is responsible for a node, not only generating its position and inserting it into *CUDA-quad-trees* but also determining a receptor's data reception from possible transmitters. For each node, *CU-Simulator* interacts with the simulation of on-node native code by means of data transmission between them.

To simulate and evaluate the communication between a pair of sensor nodes, a statistical model for data packet transmission, which has configurable and scene-specific parameters, guarantees that a simulation in *CU-Simulator* will approximate the reality of a WSN in some scenario with acceptable computation.

Wireless communication in a WSN is a course of repeated data transmission between the nodes. *CU-Simulator* simulates a short time interval of such a process, and is invoked repeatedly to accomplish continuous in-network communication simulation.

3.2 Interaction with On-node Native Code Simulation

A complete WSN run consists of wireless communication between the nodes and the execution of native code on each node. Our parallel radio channel simulator aims to strengthen a WSN simulation by accelerating the former simulation, which is a performance bottleneck. It can be combined with the simulation of on-node native code in other WSN simulators, which runs on the CPU computing platform, producing new data packets that will be transmitted or processing received data packets for each node. There is an interaction between them as shown in Figure 3.

Information about data transmission of each node includes the following items: 1) The channel state of the transceiver, which is kept or changed as the communication protocol and applications requires; 2) The time stamp of a transmitted data packet, which has appropriate values if the node is transmitting a data packet; or else, the starting point of transmitting time is set with the upper extreme, the duration of data transmission is set with zero, and then such a data packet won't be processed. 3) The number of bits transmitted in a data packet. 4) The output power of the transceiver when the node transmits a data packet. The information is set by the simulators for on-node native code before each time of radio channel simulation, and then copied to *CU-Simulator* as its input.

Information about data reception of each node is as follows: 1) Whether a node receives a data packet or not; 2) The transmitter, whose transmitted data packet is received by this receptor; 3) The signal strength of the reception; 4) Whether the received data is interfered or not. The information is output by *CU-Simulator*, and then copied back and processed by the simulators for on-node native code after an iteration of radio channel simulation in *CU-Simulator*.

4 RANDOM NUMBER GENERATION AND NODE POSITION SIMULATION

4.1 CUDA-based Random Number Generation

Since the position of a node is stochastic in a real scene and radio propagation as well, effort has been made to simulate the characteristics in *CU-Simulator*. Similar to other simulations such as Monte Carlo algorithms, we use random numbers to provide a good approximation to reality, where a number of random numbers are used in deployment of the nodes, movement of the nodes, and the simulation of data packet transmission. Parallel generation of random numbers with CUDA has been widely studied and there are a few well-developed implementations [28, 29]. Therefore, we adopt [29] as our random number generation method, where different seeds are used to generate multiple batches of random numbers for different purpose.

4.2 CUDA-based Node Position Generation

Due to the weak energy supply of radio transceivers, only the neighboring nodes of a node can receive its data packets in a WSN. Thus, the spatial relationship between every pair of nodes is an important factor that determines whether the communication between these two nodes will succeed or fail. To achieve a good simulation performance, we implement the node position simulation on GPU. This part of simulation includes deployment of nodes and movement of nodes, which are suitable to be parallelized by CUDA where each CUDA thread is responsible for a node. Similar to the 2-dimensional deployment of the nodes, CUDA threads are configured as a 2-dimensional grid. Meanwhile, they are processed in the fast on-chip memory of GPU.

Node deployment. Three approaches are provided to produce positions for all nodes. 1) Load the positions from a pre-specified file; 2) Specify the positions as a 2-dimension regular grid with an equal interval between neighboring nodes. The indices of CUDA threads and a predefined parameter for the interval between two adjacent nodes are combined to generate the coordinates of nodes in parallel. 3) Initialize the positions in a random manner, making a grid with different intervals between neighboring nodes. In such a deployment, coarse-grained coordinates of nodes are firstly produced as in (2). Then, the predefined interval and random numbers are used to drift the rough coordinates a little bit to generate the final coordinates of the nodes. In

```
(1) if (random_number & 0x1) {  // move forward
(2)     new_coord_x = old_coord_x + (random_number & 0x7);
(3) } else {                    // move backward
(4)     new_coord_x = old_coord_x - (random_number & 0x7);
(5) }
```

TABLE 1
Pseudo code for the movement of a node in *x* coordinate.

order to make node deployment more "random", we let some positions generated by some threads invalid as long as the corresponding random number is equal to a pre-specified value.

Node movement. We propose an approach, called *CUDA-random-bit-moving*, to simulate the movement of each node in a deployed area, where random numbers are used to control directions and offsets of the movement of a node. Table 1 demonstrates how the *x* coordinate of a node is updated after moving from its original position. A *bitwise and* operation of a random number with *0x1* determines the moving direction in the *x* coordinate, that is, forward or backward; a *bitwise and* operation of the random number with *0x7* determines the moving distance in *x* coordinate. Moving speed can be adjusted by replacing *0x7* with another constant value. If a node close to the boundary of the deployed area moves to a position beyond the boundary, the updated coordinate will be switched to a valid value by adding or by subtracting the span of the deployed area in the corresponding coordinate. For example, if the updated value in *x* coordinate is less than 0, it will be adjusted by adding the span of the deployed area, then the new value of *x* coordinate will be a value close to the maximum valid value in *x* coordinate.

5 CUDA-QUAD-TREES FOR ACCELERATING NODE DETECTION

In order to cut down the cost for possible transmitter detection, we propose a novel data structure, called *CUDA-quad-trees*, which resides in the fast on-chip memory of GPU, and is used to organize the nodes in network. Our proposed data structure facilitates the search for potential transmitters.

5.1 Organization of Nodes in a WSN

CUDA-quad-trees are like a forest, consisting of many small-scale quad-trees. The entire node deployment area is partitioned into many equal sized square regions, each of which contains a small number of nodes (In our experiment, the threshold is set as 64, that is, at most 64 nodes are allowed in a square region). A *CUDA-quad-tree* is used to organize the nodes in a given square region. During its construction, as shown in Figure 4, the square region is firstly divided into four equal sized sub-squares region, and then, each sub-square region, which contains more than one sensor node, is split into four sub-squares region recursively. In this way, spatially closed sensor nodes finally locate in sibling nodes of a *CUDA-quad-tree*. For example, Figure 5 shows a built *CUDA-quad-tree* for the square region in Figure 4.

The nodes in a WSN are usually deployed in a restricted area. Although small drifts in altitude may happen, a 2-dimensional coordinate is precise enough to distinguish the sensor nodes in space. Therefore, a 2-dimensional quad-tree structure is adopted in this paper. This decision not only complies with the current characteristic of node deployment in WSNs, but also minimizes the cost for building and searching the trees in terms of both memory and computation. For an unusual stereoscopic deployment, the quality of

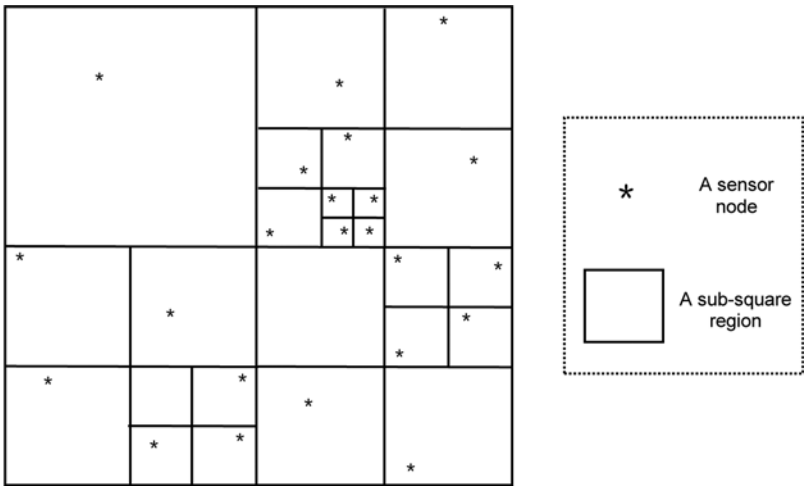

FIGURE 4
Iterative 2-dimensional partition of a square region.

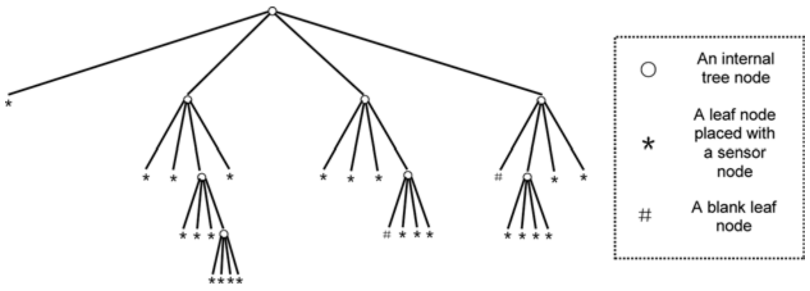

FIGURE 5
A *CUDA-quad-tree* for the nodes in Figure 4.

search in *CUDA-quad-trees* will be remedied by inspection of data packet reception between a pair of nodes later. Meanwhile, compared with a single and big tree, a collection of small trees maximizes the parallelism of CUDA threads in tree construction stage, a *CUDA-quad-tree* is small enough to reside in the fast on-chip memory of GPU. Thus, high latency of access to global memory is eliminated, leading to a considerable performance improvement.

5.2 On-chip Quad-tree Construction

In the *CUDA-quad-trees* construction stage, each thread block builds a *CUDA-quad-tree* for a square region, where each thread holds a sensor node and performs an insertion. To facilitate neighborhood searching in a *CUDA-quad-tree*, we refer a non-leaf node in the tree as a *virtual node*, which is a sub-square region. For example, in Figure 4 *root virtual node* represents the entire square region. To store a *CUDA-quad-tree*, we allocate an array with pre-defined size in shared memory to maintain the tree structure. When a new sensor node is inserted into the CUDA-quad-tree, a non-occupied position of the array backward will be used to store the information of the current node. Every tree is built in shared memory. Then, each built CUDA-quad-tree is stored into global memory in a memory coalescing access manner through the cooperation of all the threads in the working thread block.

As shown in Figure 6, it is a course of repeated attempts for a thread to insert a sensor node to construct a *CUDA-quad-tree*, and each thread competes with others for a chance to insert its node. First of all, thread 0 initial-

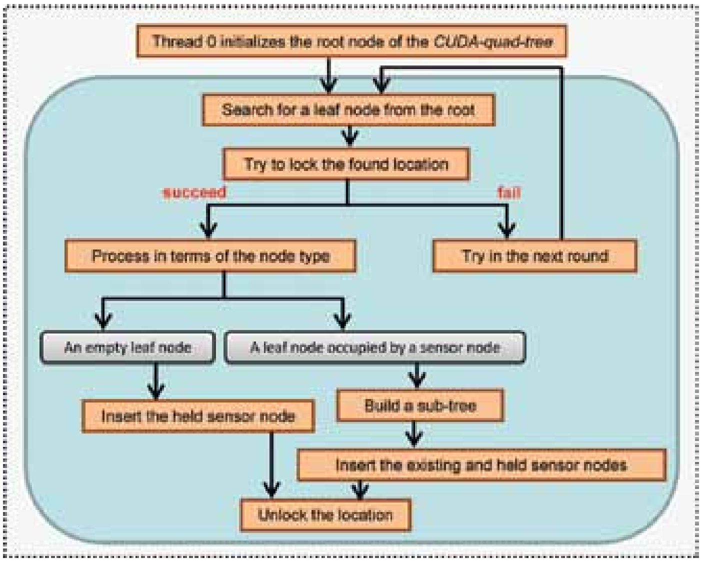
Thread 0 initializes the root node of the CUDA-quad-tree
Search for a leaf node from the root
Try to lock the found location
succeed
fail
Process in terms of the node type
Try in the next round
An empty leaf node
A leaf node occupied by a sensor node
Insert the held sensor node
Build a sub-tree
Insert the existing and held sensor nodes
Unlock the location

FIGURE 6
A thread's insertion of a sensor node for a *CUDA-quad-tree* construction.

izes the *root virtual node* with four non-occupied child tree nodes. Then, each thread looks for an appropriate leaf tree node for insertion. When such a position is found, one thread locks this position since some other threads are also trying to insert their sensor nodes at this position at this moment. Then, the held sensor node is placed into this position; that is, the node is written into the information array. At last, the thread unlocks this position, and quits insertion. If a thread fails to lock a position, it will wait for the next chance to insert. If the position, which is found and locked by a thread in the second round, has been occupied by a sensor node, the thread will have to replace the existing tree node with a new sub-tree and four non-occupied child tree nodes are created at this position, and then insert the existing and held sensor node into this sub-tree, unlocks this position, and quits insertion. The rest of the threads in the thread block keep trying repeatedly until all held sensor nodes have been placed in a leaf tree node of the *CUDA-quad-tree* as illustrated in Figure 6.

5.3 Searching On-chip Quad-trees

Due to the limited communication range of sensor nodes, the potential *communication reachable* nodes of a node must locate in its neighborhood. Since *CUDA-quad-trees* organize sensor nodes in terms of spatial closeness, the possible transmitters of a node can be found with a low computational cost. According to the communication range of radio transceivers, the superposition of the communication areas of all the sensor nodes held in a thread block will be determined, that is, a subset of a number of square regions. Then, the corresponding *CUDA-quad-trees* are looked through one by one to find all the *communication reachable* nodes for each node held in the thread block concurrently. The deeper the search in a tree is, the smaller the search area is. In order to optimize the memory access during the tree search, we firstly make all the threads in a block cooperate to load a *CUDA-quad-tree* from global memory to shared memory, and then, process the tree in the shared memory.

Figure 7 illustrates how a thread warp in a thread block performs a depth-first search in a *CUDA-quad-tree*. We make all the threads in a warp follow the same tracing path, thus, the warp maintains only two drilling information from the root to the leaves. One is the depth from the root node to the current visiting one, the other is a stack storing the indices of each being visited node among its siblings at different depth. When a *CUDA-quad-tree* is in shared memory, the warp traverses from the root node downward. Three kinds of nodes will be visited:

(1) A non-leaf node (*a virtual node*). Each thread in a warp tests whether the communication range of its held sensor node overlaps with the covered area of the *virtual node* or not. If there is an overlap, no matter how many threads meet this condition, the sub-tree of this *virtual node* will

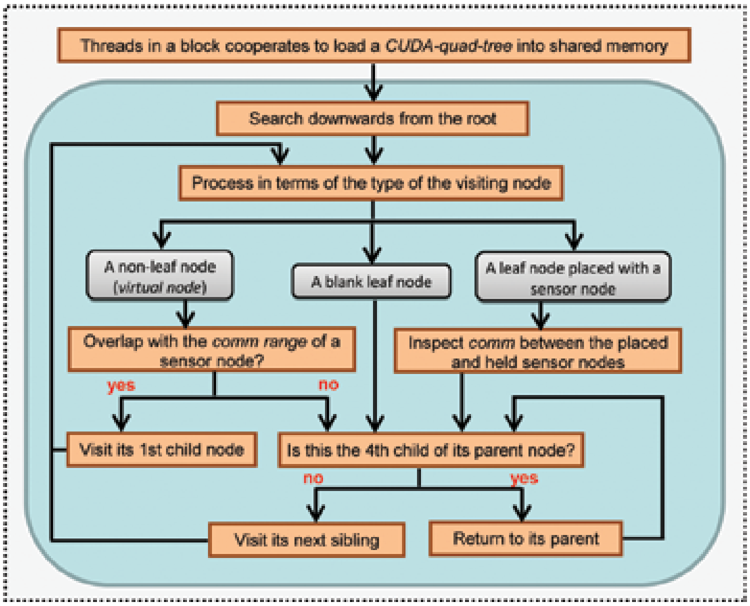
Threads in a block cooperates to load a CUDA-quad-tree into shared memory
Search downwards from the root
Process in terms of the type of the visiting node
A non-leaf node (virtual node)
A blank leaf node
A leaf node placed with a sensor node
Overlap with the comm range of a sensor node?
Inspect comm between the placed and held sensor nodes
yes
no
Visit its 1st child node
Is this the 4th child of its parent node?
no
yes
Visit its next sibling
Return to its parent

FIGURE 7
A thread warp's search in a *CUDA-quad-tree.*

be visited; otherwise, the warp skips this node and its sub-trees, and turns to its next sibling.

(2) A leaf node placed with a sensor node. Each thread in a warp tests whether the held sensor node can communicate with this sensor node in terms of distance or not, then determines if data transmission inspection should be invoked.

(3) A blank leaf node. The warp turns to its next sibling node in the current sub-tree.

Different with a single big tree maintained in global memory in the existing researches, the small on-chip *CUDA-quad-trees* allow an efficient memory access, which is critical in tree search. The CUDA threads work as a warp unit so as to prevent branch divergence.

6 SIMULATION OF DATA PACKET TRANSMISSION

In this section, we firstly present the type of radio channel models suitable for *CU-Simulator*; then, a PRR-based channel model, which has been implemented in *CU-Simulator*, is introduced; finally, we present how our search-based transmitter inspection accomplishes a simulation in parallel.

6.1 Statistical Radio Channel Models

Most statistical radio channel models are derived from a combination of analytical and empirical methods. They implicitly take into account all related factors from both wireless environment and radio hardware, known or unknown, by actual field measurements. Mathematical expression with configurable parameters also makes it easy to be implemented at low or medium computational cost in simulation. Thus, we adopt such a kind of available models in *CU-Simulator* to simulate and determine data packet transmission between a pair of sensor nodes. Common WSN deployment scenarios can be simulated with appropriate configuration, which can refer to some existing research [15] or manual measurements in a particular scenario. What's more, the radio propagation characterization of a target deployment scenario can be approximately simulated by implementing a customized model with more measurement effort to get an accurate simulation.

6.2 A PRR-based Channel Model

The radio model developed in [16] is a popular and widely applied one, which is used to estimate radio link quality in a WSN. This abstraction combines channel model with radio-receiver model, which indicates the effect of radio hardware on radio propagation. In this model, the received signal strength (RSS) is expressed as a function of distance, and so does the packet reception rate (PRR).

Received Signal Strength: When an electromagnetic signal propagates, it may be diffracted, reflected, and scattered. These effects have two important consequences on the signal strength. First, the signal strength decays exponentially with respect to distance. Second, for a given distance *d*, the signal strength is random and log-normally distributed about the mean distance-dependent value. The received power (*Pr*) in dBm is given by

$$P_r(d) = P_t - PL(d_0) - 10\eta \log_{10}\left(\frac{d}{d_0}\right) + \mathcal{N}(0,\sigma) \tag{1}$$

where d is the transmitter-receiver distance, d_0 is a reference distance, η is the path loss exponent that captures the rate at which the signal decays with respect to distance, $\mathcal{N}(0, \sigma)$ is a Gaussian random variable with mean 0 and variance σ (standard deviation due to multi-path effects), P_t is the output power, and $PL(d_0)$ is the power decay for the reference distance d_0.

Packet Reception Rate: The receiver response denotes the ratio of the number of successfully received packets to the number of transmitted packets. It is derived from a serial of equations, which depends on the modulation schemes of a radio that are widely available in the wireless communication literature, and finally described as a function of distance *d*.

The packet reception rate is derived from the bit error rate (β_E), the expression is as follows:

$$\Psi = (1-\beta_E)^{8f} \tag{2}$$

Where f is the number of bits transmitted in a data packet.

The bit error rate expression β_E is a function of α, the $\frac{E_b}{N_0}$ ratio, and its definition depends on the adopted modulation scheme. Regarding FSK non-coherent modulation scheme, for example, β_E is defined as:

$$\beta_E = \frac{1}{2} exp^{-\frac{\alpha}{2}} \tag{3}$$

The relation between signal to noise ratio (SNR) γ and $\frac{E_b}{N_0}$ is given by:

$$\gamma = \frac{E_b}{N_0}\frac{R}{B_N} \tag{4}$$

Where R is the data rate in bits, and B_N is the noise bandwidth. So, α can be derived as a function of γ from this expression.

Given a transmitting power P_t, the SNR γ at a distance d is:

$$\gamma(d) = P_r(d) - P_n \tag{5}$$

Where P_r (d) can be obtained from Eq. (1) and P_n is the noise floor.

Providing a simulation is positioned to a deployment scenario and all the corresponding parameters are determined according to literature or measurements, there are two mathematical expressions as functions of distance to be processed in simulation.

To calculate the bit error rate (β_E), we implement the mathematical expression with respect to FSK non-coherent modulation, which is typical for sub-GHz bands. According to IEEE 802.15.4-2006 standard [30], we also implement the following equation:

$$\beta_E = \frac{8}{15} \times \frac{1}{16} \times \sum_{k=2}^{16} -1^k \begin{pmatrix} 16 \\ k \end{pmatrix} e^{20 \times \gamma(d) \times \left(\frac{1}{k}-1\right)}, \tag{6}$$

which is typical for the widely used 2.4 GHz band. For those parameters whose values in a sensor node may be different with those in other sensor nodes, such as P_r (d), the parameters are maintained on node. For the parameters which are consistent in all the sensor nodes in a network, such as η, only the global variables are maintained.

6.3 Search-based Transmitter Inspection

Different from traditional sequential wireless channel simulation, our proposed simulation framework creates one module, search-based transmitter inspection using *CUDA-quad-trees*. Each sensor node is held by a CUDA thread as a receptor, and its possible transmitters are inspected. All CUDA threads work simultaneously to accomplish the simulation of all in-network data transmission for a short time interval. The search based approach benefits from the computational power of GPU and *CUDA-quad-trees* for acceleration.

In this module, each thread block takes charge of a square region, a thread holds a node. The threads in the block look up a number of neighboring *CUDA-quad-trees*, using the approach described in sub-section 5.3, to obtain the possible communication reachable nodes for the held nodes. As a receptor, all the possible transmitters discovered are inspected one by one to determine its data reception. We analyze data transmission between the receptor and a possible transmitter in terms of the frequencies of the transceivers, the distance between the nodes, and the time stamps of the data packets. Based on the fact that the radio medium is shared by sensor nodes, the possible data packets from different transmitters are compared to find out whether the receptor receives a data packet or not, if a packet is received which transmitter's data packet is received and whether the reception is interfered or not.

When a possible transmitter is detected, the communication between the receptor and this transmitter will be examined to determine whether there should be a data transmission between them. First of all, consistency of the two channels and whether the transmitter is sending a data packet are checked to determine whether there is a probability of data transmission between them. Next, the distance between these two nodes is used to calculate and determine whether the receptor can receive the data and its reception quality. (A percentage generated from a random number is compared with the calculated PRR to determine whether the transmitted data packet should be received.) If they are on the same channel, the transmitter is sending data, the receptor is within the *communication reachable* range of the detected transmitter, then, time stamps are used to conclude the comparison. There are four possible cases:

(1) It is the first data packet detected, and thus, the information of this transmission will be stored;
(2) The detected data packet is earlier than the stored one, so, discard the old one and store the newly detected one;

(3) The time of the detected data packets overlaps that of the stored one, then, keep the one which has an earlier beginning time, and the received packet is considered as being interfered;

(4) The detected data packet is later than the stored one, then, the detected data packet will be discarded.

The received data packet of the receptor will be determined by the inspection of the communication between the receptor and each found possible transmitter, and the stored data transmission will be the right one.

7 PERFORMANCE EVALUATION

7.1 Scalability Analysis

7.1.2 Space Complexity

Different from frameworks in traditional simulators, our simulation framework adopts a search-based strategy to eliminate the memory bottleneck in simulation. A small set of data for each node and a moderate set of data for each *CUDA-quad-tree*, are maintained by *CU-Simulator*. The memory overhead of a simulation in *CU-Simulator* is negligible.

Table 2 shows the data parameters used in our simulator and their sizes and the memory locations as well. The buffer size of a node is set at 128 bytes according to the hardware specification of mainstream sensor nodes. The memory space needed by a node on the host memory adds up to 281 bytes, where the data buffers dominate. The memory space needed by a *CUDA-quad-tree* is pre-specified according to the maximum number of nodes in a square region. Assuming that every square region has such number of nodes, the memory cost of a *CUDA-quad-tree* can be transformed and merged into that of the nodes in its corresponding square region when divided by this threshold, MOST_NODES (64 in our experiment). Thus, the average memory cost per node on the device is 113.1875 bytes, including the cost for a sensor node and transformation from a *CUDA-quad-tree*. For a device with 4 GB memory, a network with 35.3396 million nodes can be simulated in *CU-Simulator* as long as 9.93 GB memory on a host is available.

The data transmission between the host and the device is also small as shown in Figure 8. Before a simulation begins, the communication channel set by on-node software, the time stamp and frame size of a data packet, and the output power of a transmitter, are copied from the host to the device. When a simulation is completed, the signal strength and the index of the received data packet are copied back and processed by on-node software. There is only 25 bytes transmitted per node, and such a small transmission is negligible, which is verified by our experiment.

Type	Parameter	Size	Memory location
A sensor node	Communication channel of a node	1 char	Host & device
	Output power of a transmitter	1 float	Host & device
	Frame size of a packet in bits	1 short	Host & device
	Transmission starting time of a packet	1 long	Host & device
	Transmission duration of a packet	1 short	Host & device
	Signal strength of a received packet	1 float	Host & device
	Index of a receive packet	1 int	Host & device
	Transmission buffer per node	128 bytes	Host
	Reception buffer per node	128 bytes	Host
	Coordinates of a node	2 floats	Device
	Random numbers for node deployment	2 floats	Device
	Random numbers for node movement	2 floats	Device
	Random number generation for inspection of data packet reception	40 bytes	Device
A CUDA-*quad-tree*	Coordinates of non-leaf nodes	MOST_NODES * 2 floats	Device
	Array for node organization	(MOST_NODES * 4 + 1) ints	Device
	Bottom and depth of a CUDA-quad-tree	2 ints	Device

TABLE 2
Parameter summary and memory cost in a simulation.

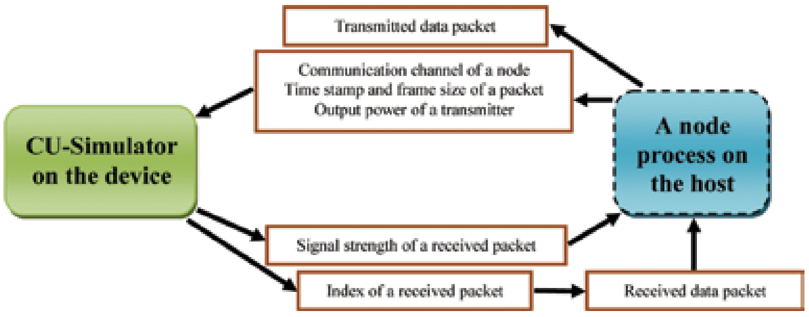
Transmitted data packet
Communication channel of a node
Time stamp and frame size of a packet
Output power of a transmitter
CU-Simulator on the device
A node process on the host
Signal strength of a received packet
Index of a received packet
Received data packet

FIGURE 8
Data exchange between the host and the device.

7.1.2 *Time Complexity*

To minimize the effect of memory access on performance, we perform computation in the manner recommended by NVIDIA. Firstly, copy data from global memory to shared memory in an efficient memory-coalescing way;

next, perform the computation in shared memory; finally, store the results back to global memory in an efficient memory-coalescing way. All kernels in *CU-Simulator* follow such a data access pattern so as to let the computation of a thread block hide memory access latency from other thread blocks.

There are four core computations in CU-Simulator, where each thread holds a sensor node and perform some computation. The time complexity of a thread in each computation is as follows:

(1) Node deployment or movement is straightforward. Coordinates generation or regeneration using random numbers and several basic mathematical computations is trivial, as illustrated in Table 1.

(2) *CUDA-quad-trees* construction or reconstruction inserts nodes into a tree. Time complexity on each node is $O(ht)$, where h is the height of a *CUDA-quad-tree*, t is the number of insertions. The worst case happens when a *CUDA-quad-tree* is something like a link list. At that time, both of these variables are no more than the pre-defined threshold.

(3) Searching *CUDA-quad-trees* is to find the neighboring nodes for the held node in several *CUDA-quad-trees*. The time complexity is $O(mn)$, where m is the number of trees inspected and n is the number of the average visited nodes in each *CUDA-quad-tree*. The transmitting range of a transceiver is constant, thus m is fixed. In the worst case, n is the number of the nodes in a *CUDA-quad-tree* where it is no more than the threshold.

(4) The inspection of data packet reception between the held node and a possible neighbor node, as in (3), is straightforward with comparison of the fixed number of parameters, in which there are the calculations for distance, a log-normally distributed variance, the received signal strength, a uniformly distributed random number and the packet reception rate. So, each thread has a constant computation.

The time complexity of *CU-Simulator* is $O(N/SPs)$, where N denotes the number of threads in any kernel (approximating to the number of nodes deployed in a simulation), and *SPs* denotes the number of stream processors in a GPU which is a constant value. Thus, *CU-Simulator* has a linear scalability.

7.2 Experimental Results

7.2.1 Experimental Setup

The device we used in our experiment is a NVIDIA's Tesla C2070 card, which is a dedicated general-purpose computing GPU with 448 stream processors (1.15 GHz) and 6 GB global/device memory. NVIDIA driver 270.41.19, CUDA Toolkit and software development kit of version 4.0 are installed. All the experiments are performed on an HP Z800 workstation with a Core-quad 2.93 GHz Intel Xeon CPU and 8 GB main memory, running Red Hat Enterprise Linux WS 6.0 operating system. Traditional programs on CPU are compiled by *java 1.6.0* or *gcc 4.4.4*.

We measured the total execution time of data transmission simulation for all nodes in milliseconds or microsecond, abbreviated as *ms* and *μs*, respectively. The time of *CU-Simulator* includes the kernel cost, as well as the cost of memory copying between the host and device. In the scalability analysis, node movement is taken into account to demonstrate the performance of *CU-Simulator* in the worst case. For a fair play, however, node movement is removed from our simulation when comparing with COOJA since COOJA doesn't support such operation. To obtain a reliable result, the execution time of each experiment is measured as an average of 10 runs.

7.2.2 Scalability

We make our simulation configuration as the experiment in [16]. An outdoor environment (football field) and mica2 motes, which use FSK non-coherent modulation at 915 MHz, are simulated. The output power P_t is set as 0 dBm, the reference distance (d_0) of the log-normal model is set at 1 m and its corresponding power decay is 55 dB. Following the measurement in [16], the path loss exponent η is 4.7, the standard deviations σ is 3.2, the noise floor Pn is −105 dBm. For mica2 motes, R = 19.2 kbps (data rate) and B_N = 30 kHz (noise bandwidth).

The execution time of simulations in CU-Simulator with varying network size is shown in Table 3. When the number of nodes is small, the execution time is quite close. The reason is that the size of a network is not large enough to make all the stream processors in the GPU work at full speed and the access latency to device memory is not well hidden by computation. Once the number of nodes in a network is over 14,400, the average execution time per node decreases not too dramatically when increasing the nodes in the network. Generally speaking, the scalability of *CU-Simulator* is super-linear. That is, the more nodes, the less execution time per node *CU-Simulator* consumes.

Table 4 presents the execution time of the main functional components in *CU-Simulator* when varying the size of a network. The characteristic is similar to what we see in Table 3. The experiments show that time consumed per node in all components drops, and data transmission simulation dominates each simulation when increasing the number of nodes in a network, even takes most of the execution time of the entire simulation when a network is large. Node movement spends a little more time than node deployment since some nodes move between different square regions, which incurs an expensive storage cost of global memory. In fact, *CUDA-quad-trees* construction is very fast, whose effect on performance is negligible.

Deployed nodes	Execution time (ms)	Time per node (μs)
1,000,000	114.86	0.1149
640,000	73.98	0.1156
160,000	19.81	0.1238
40,000	6.09	0.1524
14,400	3.03	0.2103
9,216	2.71	0.2942
6,400	2.07	0.3231
4,096	1.96	0.4773
2,304	1.92	0.8342

TABLE 3
Execution time of simulations with varying network size.

Number of nodes in a network	Node deployment (ms)	Node movement (ms)	Data transmission (ms)	Memory copy between host and device (ms)
640,000	3.809	6.783	61.050	6.150
160,000	1.300	2.035	15.637	2.141
40,000	0.803	1.018	4.361	0.715
14,400	0.709	0.782	1.862	0.384
9,216	0.687	0.732	1.641	0.338
6,400	0.661	0.685	1.099	0.284
4,096	0.659	0.674	1.031	0.250
2,304	0.656	0.670	1.020	0.211

TABLE 4
Comparison of the execution time of different components in *CU-Simulator*.

7.2.3 Compared with COOJA

COOJA with its standard channel model, *Unit Disk Graph Medium: Distance Loss (UDGM)*, was adopted as the testing counterpart in our experiments. To be consistent with UDGM, we implement the above model in our parallel radio channel simulator. Since broadcast is the most common and basic operation in wireless communication, we choose it as the benchmark and test one broadcast. While COOJA runs a Contiki [31] program, *example-broadcast.c*, *CU-Simulator* performs one inspection of data transmission for all the nodes in the network. The number of nodes in a network is varied in our experiments. Consistent with COOJA's setting, the reception threshold of a transceiver is 200 meters, and the interference threshold of a transceiver is 400 meters.

Deployed nodes	CU-Simulator (ms)	COOJA (ms)
40,000	5.38	2432.15
14,400	2.67	143.22
9,216	2.58	46.03
6,400	2.01	20.67
4,096	1.85	16.19
2,304	1.78	2.51
1,024	1.73	0.49
256	1.46	0.04

TABLE 5
Execution time with varying number of nodes in a network.

We present a comparison of the execution time between *CU-Simulator* and COOJA, shown in Table 5. When the number of nodes in a network is small, COOJA behaves better than *CU-Simulator*. This is because CUDA programs run on many-core processors in a thread-parallel way. When the number of nodes is large, the parallel superiority of GPU becomes evident. Along with the growth of network scale, the execution time of CU-Simulator increases slowly, while the execution time of COOJA increases far faster than linear complexity. When over 40,000 nodes are in the network, *CU-Simulator* achieves 452.07 times speedup.

Apart from radio channel, COOJA can simulate the execution of on-node native code, and a great effort has been made to handle the interaction with users. A run of COOJA consumes much more time and memory than the radio channel simulation. When the number of nodes is over 40,000 in a network, COOJA costs more than 2.5GB memory and more than 24 hours. Thus, we didn't measure the cost when the size of a network is over 40,000 in COOJA.

8 CONCLUSIONS

In *CU-Simulator*, the parallel search-based inspection of possible transmitters, not only brings GPU's enormous computational power into full play, but also eliminates the memory bottleneck of radio channel simulation. A group of small-scale quad-trees makes good use of fast on-chip memory to improve the efficiency of tree access, and minimizes the cost for search in simulation. Furthermore, the limited data exchange ensures that *CU-Simulator* can cooperate with a CPU-based simulation of on-node native code efficiently to accomplish a comprehensive and detailed simulation of a WSN. Experiments show that *CU-Simulator* is superior to a mainstream CPU-based implementation of radio channel simulation, where 452.07-times speedup is observed in the best case. Super-linear scalability is also observed in our experiments. Therefore, we believe CUDA-enabled GPU parallel computing architecture is able to provide a great benefit to simulation of radio channel in WSNs.

ACKNOWLEDGEMENT

This research has been supported by the Key Projects of Chinese National Science and Technology No.2010ZX03006-001, 100 Talents Program of Chinese Academic of Sciences No. 99T300CEA2 and 2009 NVIDIA's Professor Partnership.

REFERENCES

[1] Jennifer Y., Biswanath M. and Dipak G. (2008). Wireless sensor network survey. Computer Networks, 52(12), 2292–2330.
[2] Egea-Lopez E., Vales-Alonso J., Martinez-Sala A., Pavon-Mari P. and Garcia-Haro J. (2006). Simulation scalability issues in wireless sensor networks. IEEE Communications Magazine, 44(7), 64–73.
[3] Raghunathan V., Schurgers C., Park S. and Srivastava, M. B. (2002). Energy-aware wireless microsensor networks. IEEE Signal Processing Magazine, 19(2), 40–50.
[4] Levis P., Lee N., Welsh M. and Culler D. (2003). TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. ACM SenSys '03 Proceedings of the 1st international conference on Embedded networked sensor systems, 126–137.
[5] Polley J., Blazakis D., McGee J., Rusk D., Baras J. S. and Karir M. (2004). ATEMU: A Fine-grained Sensor Network Simulator. 1st Annual IEEE-Communications-Society Conference on Sensor and Ad Hoc Communications and Networks (SECON 2004).
[6] Titzer B. L. and Lee D. K. (2005). Avrora: Scalable Sensor Network Simulation with Precise Timing. 4th International Symposium on Information Processing in Sensor Networks.
[7] Naoumov V. and Gross T. (2003). Simulation of Large Ad Hoc Networks. Proceedings of the Sixth ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2003).
[8] NVIDIA CUDA C programming guide 4.0. (2011). http://developer.nvidia.com/cuda-toolkit-40.
[9] Tesla C2050 and Tesla C2070 Computing Processing Board. (2010). http://www.nvidia.com/docs/IO/43395/Tesla_C2050_Board_Specification.pdf.
[10] The Network Simulator Ns-2. (2002). http://www.isi.edu/nsnam/ns/ns-documentation.html.
[11] OMNeT++ Network Simulation Framework. (2012). http://www.omnetpp.org/.
[12] Said A. M. and Hasbullah H. (2010). A Survey of Simulators, Emulators and Testbeds for Wireless Sensor Networks. 2010 International Symposium in Information Technology (ITSim), 897–902.
[13] Baldwin P., Kohli S., Lee E. A., Liu X. and Zhao Y. (2004). VisualSense: Visual Modeling for Wireless and Sensor Network Systems. Technical Memorandum UCB/ERL M04/08, University of California, Berkeley.
[14] Osterlind F., Dunkels A., Eriksson J., Finne N. and Voigt T. (2006). Cross-level sensor network simulation with cooja. Proceedings of 31st IEEE Conference on Local Computer Networks.

[15] Baccour N., Koubaa A., Mottola L., Zuniga M., Youssef H., Boano C., and Alves M. (2012). Radio Link Quality Estimation in Wireless Sensor Networks: a Survey. ACM Transactions on Sensor Networks (TOSN), (8)4, Article Number: 34, DOI: 10.1145/2240116.2240123.

[16] Zuniga M. and Krishnamachari B. (2007). An Analysis of Unreliability and Asymmetry in Low-Power Wireless Links. ACM Transactions on Sensor Networks (TOSN), 3(2), Article Number: 7, DOI: 10.1145/1240226.1240227.

[17] Senel M., Chintalapudi K., Lal D., Keshavarzian A., and Coyle E. J. (2007). A Kalman filter based link quality estimation scheme for wireless sensor networks. Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM '07), 875–880.

[18] Fonseca R., Gnawali O., Jamieson K., and Levis P. (2007). Four bit wireless link estimation. Proceedings of the 6th International Workshop on Hot Topics in Networks (HotNets VI), ACM SIGCOMM.

[19] Zhang H., Sang L., and Arora A. (2010). Comparison of data-driven link estimation methods in low-power wireless networks. IEEE Transactions on Mobile Computing, 9(11), 1634–1648.

[20] Puccinelli D. and Haenggi M. (2008). DUCHY: Double cost field hybrid link estimation for low-power wireless sensor networks. Proceedings of the 5th Workshop on Embedded Networked Sensors.

[21] Baccour N., Koubaa A., Ben J. M.,Youssef H., Zuniga M., and Alves M. (2009). A comparative simulation study of link quality estimators in wireless sensor networks. Proceedings of the 17th IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '09), 301–310.

[22] Fathy A. A., Matthias K. and Christian B. (2009). A Novel Architecture using NVIDIA CUDA to speed up Simulation of Multi-Path Fast Fading Channels. 69th IEEE Vehicular Technology Conference.

[23] Bai S. and Nicol D. M. (2010). Acceleration of wireless channel simulation using GPUs. Proceedings of European Wireless 2010, 841–848.

[24] Samet H. (1984). The Quadtree and Related Hierarchical Data Structures. Computing Surveys, 16(2), 187–260.

[25] Kelly M. and Breslow A. (2010). Parallel computing final project-Quad-tree Construction on the GPU: A Hybrid CPU-GPU Approach. http://www.sccs.swarthmore.edu/users/10/mkelly1/courses.html.

[26] Zhou K., Hou Q. M., Wang R. and Guo B. N. (2008). Real-Time KD-Tree Construction on Graphics Hardware. ACM Transactions on Graphics.

[27] Burtscher M. and Pingali K. (2011). An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. GPU Computing Gems: Emerald Edition, 75-92, Morgan Kaufmann.

[28] van Meel J. A., Arnold A., Frenkel D., Zwart S. F. Portegies and Belleman R. G. (2008). Harvesting graphics power for MD simulations. Molecular Simulation, 34(3), 259–266.

[29] NVIDIA CUDA Random Number Generation library – cuRAND. (2012). http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CURAND_Library.pdf

[30] IEEE standard 802.15.4-2006. (2006). http://standards.ieee.org/getieee802/download/802.15.4–2006.pdf.

[31] Dunkels A., Grönvall B., and Voigt T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. Proceedings of the 1st IEEE Workshop on Embedded Networked Sensors.