2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

A Parallel Rectangle Intersection Algorithm on GPU+CPU

Shih-Hsiang Lo, Che-Rung Lee, Yeh-Ching Chung
Computer Science Department
National Tsing Hua University
Hsinchu, Taiwan
awatch@gmail.com, cherung@cs.nthu.edu.tw and
ychung@cs.nthu.edu.tw

I-Hsin Chung
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
ihchung@us.ibm.com

***Abstract*—In this paper, we investigate efficient algorithms and implementations using GPU plus CPU to solve the rectangle intersection problem on a plane. The problem is to report all intersecting pairs of iso-oriented rectangles, whose parallelization on GPUs poses two major computational challenges: data partition and the massive output. The algorithm we presented is called *PRI-GC*, Parallel Rectangle Intersection algorithm on GPU+CPU, which consists of two phases: mapping and intersection-checking. In the mapping phase, rectangles are hashed into different subspaces (called cells) to reduce the unnecessary intersection checking for far-apart rectangles. In the intersection-checking phase, pairs of rectangles within the same cell are examined in parallel, and the intersecting pairs of rectangles are reported. Several optimization techniques, including rectangles re-ordering, output data compressing/encoding, and the execution overlapping of GPU and CPU, are applied to enhance the performance. We had evaluated the performance of PRI-GC and the result shows over 30x speedup against two well-implemented sequential algorithms on single CPU. The effectiveness of each optimization technique for this problem was evaluated as well. Several parameters, including different degrees of rectangle coverage, different block sizes, and different cell sizes, were also experimented to explore their influences on the performance of PRI-GC.**

Keywords - Rectangle Intersection, CUDA, Parallel Algorithms

I. INTRODUCTION

Rectangle intersection is an important step in a variety of applications, such as motion simulations, computer graphics, and VLSI physical design. In this paper, we consider the iso-oriented rectangle intersection problem on a plane, which is defined as follows: Given a set of N rectangles with axis-parallel sides (iso-oriented rectangles) on a plane, report all intersecting pairs of rectangles. A rectangle in this definition is a set of closed line intervals and an intersecting pair of rectangles means the two rectangles share at least one common point.

The most naïve method to solve this problem is to search all pairs of N rectangles. Although it takes $O(N^2)$ time, it is the worst case lower bound to report all intersecting pairs of rectangles. In practical cases, the number of intersected rectangles is much fewer than $N^2/2$. Speedup methods had been widely studied for decades. Sequential algorithms, for example, can be found in [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12] and [13]. Common techniques used in those methods include plane sweep, spatial partition, and sorting. Parallel solutions to this problem for different architectures were also well investigated. Algorithms for PRAM models were presented in [14]; algorithms for multiprocessor systems could be found in [15] and [16].

This paper is concerned with the algorithms for solving the rectangle intersection problems on Graphics Process-ing Units (GPUs). Two important reasons motivate this direction. The first is the maturity of programmable graphic hardware, which possesses massively parallel processing units and allows user to program it for the purposes other than graphics related computation. Second, the computation of most algorithms is data parallelizable that fits naturally to the massive parallel computing environments like GPUs.

There are two major computational challenges to implement the rectangle intersection algorithms on GPUs. The first one is the data partition, which reduces the large problem into small ones. For the naïve method, one can simply divide the data to rectangle level. However, many unnecessary comparisons and data movements may significantly slow down the performance. Using the locality of rectangles to partition data is the most common and efficient method. Static space partition methods that divide data according to the evenly partitioned space face the load balance problem. Adaptive space partition methods that dynamically refine partition space to achieve better load balance need complicated data structures, which may not be really beneficial on GPUs.

The second challenge is to report intersecting pairs of rectangles, because the output data size can be large and unpredictable. The output data size can be as large as the number of rectangles or even more, which makes it the major performance bottleneck. In addition, the number of intersecting pairs of rectangles in each set of partition data cannot be known in advance. How to allocate the memory dynamically for the output also affects the performance significantly.

After surveying major methods, sequential or parallel, the algorithm we designed on GPU+CPU is based on static space partition method. Our algorithm consists of two phases: mapping and intersection-checking. In the mapping

978-0-7695-4395-6/11 $26.00 © 2011 IEEE
DOI 10.1109/CCGrid.2011.13

phase, rectangles are hashed into different subspaces (called cells) to reduce the unnecessary intersection checking for far-apart pairs of rectangles. In the intersection-checking phase, pairs of rectangles within the same cell are examined in parallel, and the intersecting pairs of rectangles are reported.

Several optimization techniques are applied to enable this algorithm to resolve the challenges mentioned above. First, although the mapping is done in parallel, rectangles in a cell are recorded in a continuous space to speed up the data access by memory coalescing and shared memory. In addition, all cells with nonzero rectangles resided are packed into one big continuous memory, based on their order, to avoid the access of empty cells. Second, to reduce the number of memory transmissions, intersection results, computed in GPU, are encoded/compressed in the hierarchical manner, and are decoded/decompressed by CPU. Third, to overlap the execution of GPU and CPU, the encoded data are stored in double buffers, and the decoding and encoding processes are done on each buffer alternatively.

The performance of the proposed algorithm was evaluated in contrast with one sequential cell-based algorithm (Cell-CPU) and one sequential tree-based algorithm (Tree-CPU) on single CPU. We implemented Cell-CPU, which uses the optimized cell size to reduce unnecessary intersections checking. We enable Cell-CPU to adjust the area of cells to be the average area of all rectangles. Tree-CPU is obtained from Computational Geometry Algorithms Library (CGAL) [17], which is the readily available implementation for researchers and industry customers. We will briefly explain Tree-CPU in Section II. For the problem with ten millions of rectangles, the proposed algorithm delivers up to around 30 times performance improvement comparing to the two sequential implementations. The optimization techniques enhance 3-5x performance improvement, compared with the proposed algorithm without any optimization. The comparisons were made for different problem settings, mainly for different degrees of rectangle coverage (DRC), which represents the crowdedness of rectangles. Last, the effectiveness of different parameter settings, such as the block size (i.e., the number of threads per GPU thread block) and the cell size (i.e., the number of cells in partitioned space), are examined empirically and theoretically.

The organization of this paper is as follows. In Section II, we briefly discuss the background, including the survey on rectangle intersection algorithms and the used GPU architecture, namely nVidia's CUDA. Section III introduces our algorithm in detail. The relation of cell size and performance will be discussed in Section IV. In Section V, the experiments and the results for performance evaluation are presented. We conclude this paper and describe future work in Section VI.

II. BACKGROUND

In this section, we give a briefly survey on the rectangle intersection problem and an illustration on the nVidia CUDA architecture.

A. Sequential Algorithms for the rectangle intersection problem

The algorithms proposed in [1], [2] and [3] for finding intersecting pairs of rectangles are based on the sweep-line algorithm proposed by Shamos and Hey [18]. An efficient sweep-based algorithm was presented by Six and Wood [3]. The authors used the sweep line technique to determine intersections of rectangles at 1^{st} dimension. During sweeping, it uses the interval tree or the range tree data structure (see [19] for details) to further determine intersections of rectangles at 2^{nd} dimension.

Vaishnavi and Wood [4] and Edelsbrunner [5] used layered segment tree and *d*-fold rectangle tree to directly obtain the intersection results in 2-D environments, respectively. An alternative tree-based algorithm was shown in [7]. It can be easily generalized to 3- or higher-D environments. For *d*-D environments, the algorithm builds one interval tree for each dimension. For a *d*-D rectangle query with *d* line interval queries, it merges the *d* intersection results of *d* queries to obtain a *d*-D rectangle intersection result. Regarding the space requirement in practice, Zomorodian and Edelsbrunner proposed a hybrid algorithm in [8]. The algorithm scans the end points of boxes and traverses segment and range trees in a hybrid fashion according to a cut-off value. The implementation of the algorithm (called Tree-CPU in this paper) can be found in Computational Geometry Algorithms Library (CGAL) [17].

Some rectangle intersection algorithms preferred to divide the space of an environment into subspaces (cells) by fixed or dynamic size. The authors in [6], [9] and [20] adopted fixed-size partition strategy. An environment is decomposed into cells and rectangles are mapped to cells. Rectangles are only compared with the other rectangles in the particular cell(s). This can reduce unnecessary comparisons for those rectangles at different cells. The major issue of algorithms using uniform cells is that the cell size is crucial to the performance of rectangle intersection [10]. However, cell-based algorithms using uniform cells are suitable for parallelization [21]. Van Hook et al. [6] and Eroglu et al. [13] used 2^d tree data structure to support dynamic cell size change. The idea is to split a cell (partition) into four equal areas of the partitioned cell until a stop condition is satisfied. Then, each rectangle in a cell is compared with other rectangles in the particular cell(s).

Some rectangle intersection algorithms in [10], [11] and [12] are based on sorting techniques. Essentially, the sort-based algorithms are similar to the sweep-based algorithms. An efficient sort-based algorithm proposed by Raczy et al. [12]. The algorithm first sorts end points of all rectangles for each dimension. It then scans all end points for obtaining the intersection result in each dimension. The overall

intersection result can be obtained by merging the intersection results of all dimensions. Pan et al. [11] proposed a dynamic sort-based algorithm to support scanning the end points within a dynamic range rather than scanning all end points. Gupta and Guha [12] proposed the P-Pruning algorithm with an another sorting technique (i.e., bucket sort). The algorithm performs rectangle intersection computations efficiently particularly for small-scale environments.

B. Parallel Algorithms for the rectangle intersection problem

In the early work, Chow [14] proposed three parallel algorithms for the rectangle intersection problem. The first algorithm is based on a concurrent read exclusive write (CREW) parallel random access machine (PRAM) model and the second and the third algorithms are based on a cube-connected cycles (CCC) model. In both models, low-level parallel architectures are ignored, such as synchronization and communication. The three algorithms are of theoretical interest. However, the first algorithm might run on share memory systems with many processors (e.g., GPUs) because PRAM is a share memory abstract machine. The time complexity of the first algorithm is $O(\log^2 N+I_{max})$ time with $O(N)$ processors, where N is the number of rectangles and I_{max} is the maximum number of intersections per rectangle.

A parallel interest matching algorithm for distributed virtual environments (DVEs) was presented by Liu and Theodoropoulos [15]. The algorithm performs interest matching on shared memory multiprocessor systems. The authors used flat subdivision technique [22] to divide the space of an environment into subspaces initially. Rectangles are then hashed into these subspaces according to the end points of rectangles. Each subspace is treated as a work unit. Work units are dispatched to processors and processed by a sort-based algorithm with insertion sort. The sort-based algorithm performs well based on the assumption that temporal and geometric coherence exists in DVEs.

Recently, Batista et al. [16] presented a parallel algorithm for *d*-D box (rectangle) intersection. The parallel algorithm is based on the sequential algorithm by Zomorodian and Edelsbrunner [8]. The central idea of the parallel algorithm is divide the sequence of intervals as subtasks and conquer subtasks by threads on OpenMP [23] framework.

C. Compute Unified Device Architecture (CUDA)

CUDA is a parallel computing architecture with an array of multiprocessors and various memory spaces for execution. In the CUDA architecture, threads are assigned to groups (called warps) and executed by multiprocessors of the device (i.e., the CUDA-enabled product). Since the CDUA architecture employs Single-Instruction Multiple-Threads (SIMT) paradigm, all threads in a warp execute one common instruction at a time. Each multiprocessor is capable of executing one or more warps concurrently. In the device, data for access by threads could be placed in different levels of device memory hierarchy, including registers, shared memory, cache, constant memory, texture memory and global memory.

To carry out tasks using CUDA, the host (i.e., the computer system consists of one or more CPUs and one or more CUDA-enabled products) copies data from the host memory to the device memory. Then, the host invokes a kernel function with the specified execution configuration. The execution configuration defines the organization of threads (i.e., the number of thread blocks in a kernel grid and the number of threads in a thread block). While running a kernel function, all thread blocks within a kernel grid will be distributed to multiprocessors. Each multiprocessor will arrange thread blocks into warps and schedule them for execution. When the kernel function is completed, the host copies data from the device memory to the host memory.

Two key aspects of performing tasks using CUDA with high performance need to be concerned. One is the degree of thread parallelism. In SIMT paradigm, if threads in a warp have different execution paths due to a data-dependent condition, threads need to take turns in performing instructions (i.e., if- and else-part). The behavior is called warp divergence. When encountering more branches, threads in a warp are serialized more. Warp divergence should be avoided. Another one is the memory access pattern by threads. Threads should access data in the low latency memory (e.g., shared memory or cache) rather than in the high latency memory (e.g., global memory). Even if threads needs to access data from the global memory, coalesced memory access can reduce the number of memory transactions.

D. The Potential of Rectangle Intersection Algorithms Using CUDA

To perform rectangle intersection computations on GPUs, we consider the intrinsic nature of the rectangle intersection algorithms reported in literature.

For the naïve algorithm, each rectangle is compared with all the other rectangles. The entire comparison operations can be executed in parallel. However, it is algorithm inefficient. For the cell-based algorithms, the operation of mapping rectangles can be executed in parallel as long as rectangles can be recorded in cells concurrently. Atomic instructions provided in the CUDA programming model are used to guarantee it. After the operation of mapping rectangles, the operation of matching rectangles is similar to that of the naïve algorithm.

The cell-based algorithms are variants of the brute-force algorithm. For the sweep-based and the sort-based algorithms, the end points of rectangles need to be sorted. It is possible to parallelize this sorting operation. However, the scanning range of a rectangle is usually different with those of other rectangles. While a thread scans the range of a rectangle, the other threads in the warp might be disabled. This leads to thread serialization in a warp. For tree-based algorithms, the tree traverse operation enables the nearby

rectangles of a rectangle be found efficiently. As stated above, due to varying ranges of rectangles, to find intersections of rectangles needs to traverse different paths. This results in different execution paths within the same warp. Another problem for the tree-based algorithm is the difficulties of dynamically tree construction in GPUs. As nodes change at run-time, data in nodes need to be altered appropriately and atomically.

In divide and conquer paradigm, the rectangle intersection algorithms can be parallelized in some way. The cell-based algorithms are efficiently feasible to fulfill performing rectangle intersection computations on GPUs, while the sweep-based, sort-based, and tree-based algorithms that depend on control flow, are suitable to be performed and parallelized on CPUs, like [15] and [16].

III. The Parallel Intersection Algorithm (PRI-GC)

Our algorithm can be divided into two phases: the mapping phase and the intersection-checking phase.

A. The Mapping Phase

This phase is to partition the entire space into small cells and to find the cells occupied by rectangles. We use static cell size to partition the space and order them in the row major. (The relation of cell size and performance will be discussed in Section IV.) With this data partition, the cells that each rectangle resides in can be calculated in parallel. However, the information we need in the intersection-checking phase is a list of rectangles resided in a cell for each cell. Since the number of rectangles in a cell varies, allocating a fixed-length array for each cell is infeasible. On the other hand, dynamic space allocation introduces synchronization overhead. To obtain the optimal performance, we need the lists of rectangles are placed in order in a continuous array.

To achieve this, we use three arrays as shown in Fig. 1. The first array, of the same length as the total number of cells, stores a counter for each cell that indicates the number of rectangles occupies the cell. The second array, also sized for the total number of cells, stores the offset of a cell, which indicates the beginning position of the rectangle list in the third array. The third array is a compact storage for the list of rectangles of all cells. For example, cell c_1 contains three rectangles and the offset of cell c_1 is 2. Rectangle r_3, r_9 and r_{10} are in the 3rd, 4th and 5th element in the rectangle list, respectively.

The algorithm of mapping is sketched as follows.

1. The information of rectangles, including coordinates and identifications, are blocked, and each block of data are read into the shared memory in parallel.
2. Each thread handles a constant number of rectangles. It first read the coordinate information of each rectangle and checks those cells it occupies.
3. Each thread increases the rectangle counter for the occupied cells. To avoid concurrent write, the atomic instruction is used.

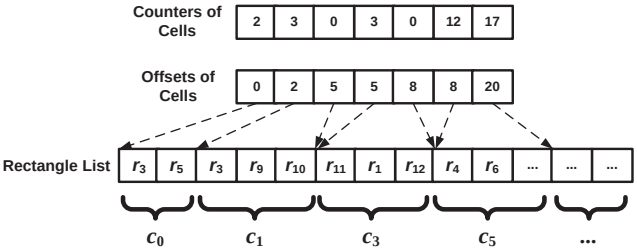

Figure 1. Example of data structures for cells

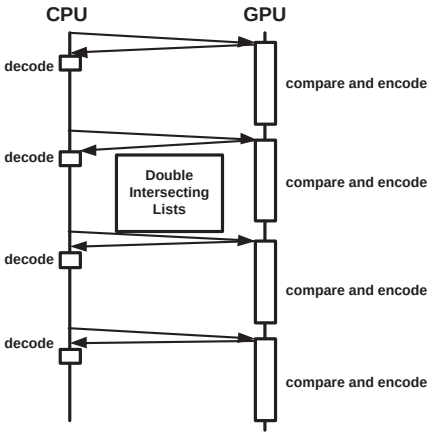

Figure 2. The co-operation between the GPU and CPU

4. Run the parallel prefix sum algorithm to obtain the offset array.
5. Based on the offset, each thread fills in the rectangles. The corresponding positions of a rectangle in the rectangle list can be calculated according to the offset of the cell and the order of executing the atomic instruction.

B. The Intersection-checking Phase

In this phase, the CPU and GPU co-operate in performing intersection checking and reporting intersection results. The CPU is to do block scheduling and to decode the encoded intersecting pairs of rectangles. Since both jobs need to do condition checking on two lists, counters of cells and the intersecting list (where the GPU stores the encoded intersecting pairs of rectangles), the CPU is suitable to perform both jobs. The GPU is responsible for performing comparisons and reporting the encoded intersecting pairs of rectangles. Fig. 2 illustrates the co-operation between the CPU and GPU. After performing block scheduling, the CPU invokes an intersecting kernel function and then decodes the encoded intersecting pairs of rectangles. By double buffering, the CPU and GPU can execute at the same time so that the time to perform the decoding process can be hid by the time to execute the kernel function. Before explaining the intersection-checking phase, we first define several notations used. Let T be the block size (i.e., the number of threads per

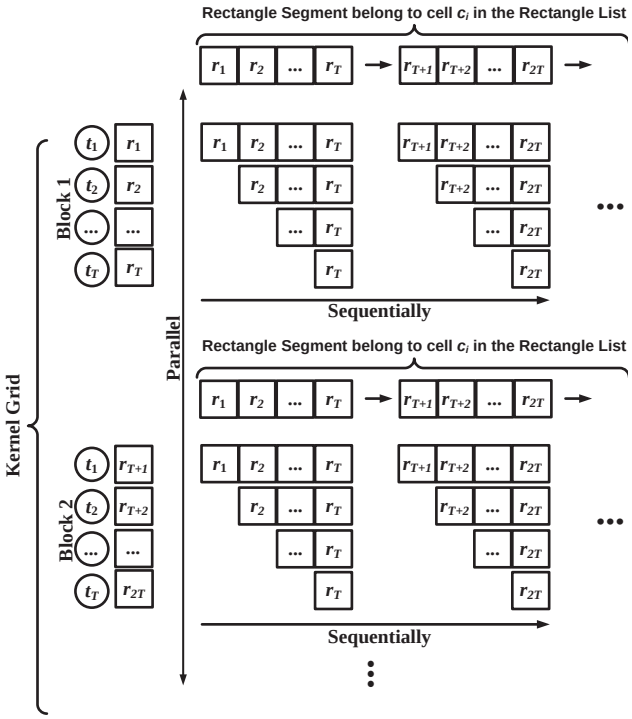

Figure 3. The execution diagram of the kernel function in the intersecting-checking phase

thread block). Let *CS* be the cell size and be defined as the number of cells in partitioned space. Let cnt_i be the value of the counter of cell c_i. Let $v(r_a,i,\text{low})$ be the coordinate of the lower end point of rectangle r along i^{th} dimension. Let $v(r_a,i,\text{up})$ be the coordinate of the upper end point of rectangle r along i^{th} dimension.

In the beginning of the intersection-checking phase, the CPU carries out block scheduling such that each thread block is scheduled to check intersections for at most T rectangles. That is, if cell c_i contains more than T rectangles, cnt_i/T thread blocks are used for cell c_i. More rectangles a cell contains, more threads are used to check intersections for all rectangles in the cell. If cell c_i contains less than or equal to T rectangles, cell c_i is processed by a thread block or several cells (including cell c_i) are grouped and handled by a thread block. In the following, we explain the detailed steps of the kernel function executed by the GPU.

In the kernel function, each thread compares one rectangle with all rectangles in the same cell. Fig. 3 shows the execution diagram when the GPU checks intersections for rectangles. The kernel function of the intersecting-checking phase consists of four steps as follows:

Step 1: Each thread obtains one rectangle from the rectangle list that its thread block needs to deal with. According to the identification of the rectangle, it reads its rectangle from the global memory.

Step 2: Each thread within a thread block loads a rectangle from the global memory into the shared memory. For example, in Fig. 3, thread t_1 in Block 1 gets the identification of a rectangle from the rectangle list first and then loads that rectangle from the global memory into the shared memory. Since each thread participates in loading T rectangles into the shared memory, the number of times a rectangle is accessed from the global memory is decreased from $T^2/2$ to T and each rectangle in the shared memory is reused for $T/2$ times averagely. Because each pair of rectangles has symmetric property, only half pairs of rectangles are compared. Only T rectangles are loaded into the shared memory at a time due to the limited amount of the shared memory available for a multiprocessor. Until T rectangles within the thread block have been loaded into the shared memory, all threads within the thread block proceed to *Step* 3.

Step 3: Each thread within the thread block sequentially compare its rectangle with the rectangles that have been loaded into the shared memory. Because of the symmetric property of pairs of rectangles, some pairs of rectangles are skipped. The principle of comparing two rectangles is that the lower and the upper end points of a rectangle are compared with those of the other rectangle along each dimension. Formally, rectangle r_a and r_b intersects such that for each dimension i

$$v(r_a,i,\text{low}) \le v(r_b,i,\text{up}) \wedge v(r_b,i,\text{low}) \le v(r_a,i,\text{up}). \quad (1)$$

Since rectangles could be mapped to one or more cells, two rectangles could be compared and decided as intersected in more than one cell. This results in redundant intersecting pairs of rectangles in the intersecting list. Fig. 4 illustrates such a case in a 2-D environment. In Fig. 4, rectangle r_1 and r_2 intersects in two cells. The pair of rectangle r_1 and r_2 will be recorded in the intersecting list twice. To avoid redundancy, we have a scheme to let threads insert intersecting pairs of rectangles into the intersecting list without doing synchronization. In this scheme, once a thread decides two rectangles as intersected, the thread needs to determine the right to write the intersecting pair to intersecting list. First, it finds the first cell of the intersecting area of two rectangles. If the first cell is the same as the cell where the thread is comparing rectangles, then the thread has the right to insert the rectangle to the intersecting list. Otherwise, the pair of rectangles should not be inserted into the intersecting list.

In order to find the first cell that the intersecting area of two rectangles, we calculate one particular intersecting point of two rectangles and calculate the cell the intersecting point intersects. The particular intersecting point of two rectangles is the first-considered point when performing mapping. The coordinate of the particular intersecting point of two rectangles r_a and r_b at i^{th} dimension is

$$max(v(r_a,i,\text{low}), v(r_b,i,\text{low})). \quad (2)$$

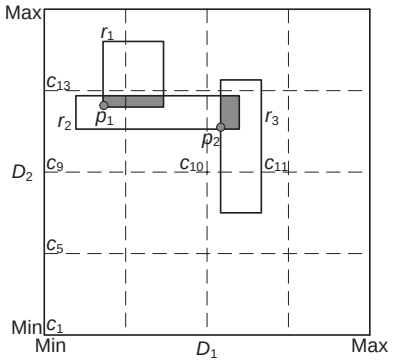

Figure 4. Example of redundant pairs: (r_1,r_2) in cell c_9 and (r_1,r_2) in cell c_{10}.

In Fig. 4, for example, p_1 and p_2 are the particular intersecting points of the pair (r_1 , r_2) and the pair (r_2 , r_3), respectively. With the intersecting points and the cell size, it is easy to calculate the cell a point intersects. Both calculations, to calculate the particular intersecting point of two rectangles and to calculate the cell the point intersects, can be done independently and efficiently.

Once two rectangles intersect, the pair of the two rectangles is encoded and then stored in thread local-cached memory. Since each thread compares one rectangle with other rectangles, each thread can encode the intersecting pairs by using the local indexes of rectangles in the cell. Specifically, the local memory of a thread records the elements as follows: the current cell, the number of intersecting pairs, the local index of its rectangle and the local indexes of other intersecting rectangles.

Step 4: When all threads within the thread block complete the comparison operations for *T* (or less than *T*) rectangles, all threads within the thread block write its encoded intersecting pairs of rectangles in the intersecting list (in the global memory). Concurrently, several thread blocks could write the intersecting pairs of rectangles to the global memory. To reduce the number of atomic instructions used, we use prefix sum mechanism to calculate the offsets by which threads can know where it can place the intersecting pairs of rectangles in the intersecting list safely. In this way, each thread block only performs one atomic instruction in order to get the global offset in the intersecting list. After this step, threads proceed to *Step* 2 for finding intersections for the next *T* rectangles if needed. *Steps* 2, 3 and 4 repeat until a thread block completes the comparison operations for all those rectangles in the same cell.

IV. Relation of Cell Size and Performance

In this section we discuss the cell size impact on the performance of PRI-GC. The cell size decides the performance of three tasks: calculate the rectangle offset array by prefix sum algorithm, block scheduling by CPU and intersecting kernel function by GPU. The first two tasks operate on the rectangle counter array. Since the cell size decides the length of the rectangle counter array, the time to perform the two tasks is affected by the cell size. The third task is most affected by the cell size because the cell size decides the fact of the number of rectangles in a cell, i.e., the number of comparisons performed. In the following, we focus on the discussion of the cell size related to the third task.

For PRI-GC, the comparisons carried out in the intersection-checking phase can be classified into three types: *effective comparison*, *unnecessary comparison* and *redundant comparison*. A comparison of two rectangles is an effective comparison (EC) if two rectangles are in a given cell and both rectangles are overlapped. A comparison of two rectangles is an unnecessary comparison (UC) if two rectangles are in a cell and they are not overlapped. A comparison is a redundant comparison (RC) if the match of two rectangles has been performed at other cells. In PRI-GC, the total number of ECs carried out is a constant. However, the numbers of UCs and RCs carried out are related to the chosen cell size. When the area of cells is larger than the average area of rectangles, the number of UCs will increase. Conversely, when the area of cells is smaller than the average area of rectangles, the number of RCs will increase. From a rectangle point of view, a rectangle is compared with those rectangles that are close to the rectangle in terms of the area of cells. If the area of cells is approximately equal to the area of the rectangle, the rectangle is compared with a very small set of rectangles only. Also, the comparisons carried out are likely effective comparisons. (That is why our sequential algorithm (Cell-CPU) sets the area of cells to be the average area of all rectangles.) If PRI-GC uses the average area of all rectangles as the area of cells, it can minimize the number of comparisons performed.

However, in the GPU computing environment, the performance of PRI-GC is not only related to algorithm efficiency but also to GPU hardware efficiency. Since the CDUA architecture employs SIMT computing model, a warp (i.e., 32 threads in the current CUDA architecture) executes one instruction concurrently. If a cell contains only few rectangles (<32) that mapped to this cell, to check intersections for the rectangles in this cell causes low utilization of hardware resources (including computing units and registers and shared memory).

As a result, areas of rectangles, the total number of rectangles, and the GPU characteristics should be considered. The choice of the cell size should make a cell to averagely contain more or less 32 rectangles, i.e.,

$$\frac{RA}{N} ws \cong \frac{EA}{CS}, \tag{3}$$

where *RA* is the sum of areas of all rectangles, *N* is the number of rectangles, *WS* is the warp size, *EA* is the area of an environment, *CS* is the cell size, i.e., the number of cells in partitioned environment. In (3), the area of cells is roughly

equal to 32 times of the average area of all rectangles. This minimizes the number of comparisons performed and also enables threads within a warp to do comparisons effectively.

V. PERFORMANCE EVALUATION

In this section, the performance of the proposed algorithm is evaluated by the comparisons to two efficient sequential implementations. The first one is a cell-based algorithm, which alters the area of cells as the current average rectangle area (Cell-CPU). The second one is a tree-based algorithm in CGAL (Tree-CPU), which is the widely available implementation for research. We take the 2-D box self-intersection implementation for reference.

The experimental platform is equipped with one Intel i7 processor 2.67 GHz, 6 GB DRAM, and a Geforce GTX 480 video card. The OS used is Linux of kernel version 2.6.27. Geforce GTX 480 contains 15 multiprocessors (480 CUDA cores in total) and 1.5 GB DRAM. We set L1 cache size to 16 KBytes and shared memory 48 Kbytes. For the host programs, we use GCC 4.3 and enable optimizations, –O3 and –DNDEBUG. For the device programs, we use CUDA compiler driver 3.2. The parallel algorithms run on the host and the device, while the sequential algorithm runs on the host. For the parallel algorithms, we measure the time to perform rectangle intersection computations for *N* rectangles, including the time to copy rectangles to the device, to invoke the kernel function(s), to copy the intersecting result to the host, and to decode the intersecting result. For the sequential algorithm, we measure the CPU time to perform rectangle intersection computations for N rectangles. The running time is averaged over 10 times for each test case.

In this paper, we use the *Degree of Rectangle Coverage* (DRC) to adjust the density of rectangles. DRC is defined as

$$DRC = \frac{RA}{EA}, \tag{4}$$

where *RA* is the sum of areas of all rectangles in an environment and *EA* is the area of the environment. The DRC value means the average number of rectangles per unit area. We generate the low DRC (10^{-7} ~ 10^{-3}), the medium DRC (10^{-5} ~ 10^{-1}) and the high DRC (10^{-3} ~ 10^{1}) environments for experiments. As the number of rectangles simulated increases (i.e., *RS* increases), the DRC value increases as well. In general, the DRC values for most applications are within the range 10^{-7} to 10^{1}. For number of rectangles equals to 10^{7}, the average numbers of intersections for the low DRC, medium DRC, and high DRC are about 0.002*N*, 0.2*N* and 20*N*, respectively.

A. Performance in Varying DRC Environments

We evaluate the performance of three algorithms in the low, the medium and the high DRC environments. The numbers of rectangles (*N*) are 10^{3}, 10^{4}, 10^{5}, 10^{6}, and 10^{7}. The block size for the parallel algorithms is 256. The number of cells in an environment for the proposed algorithm, PRI-GC, is 400×400.

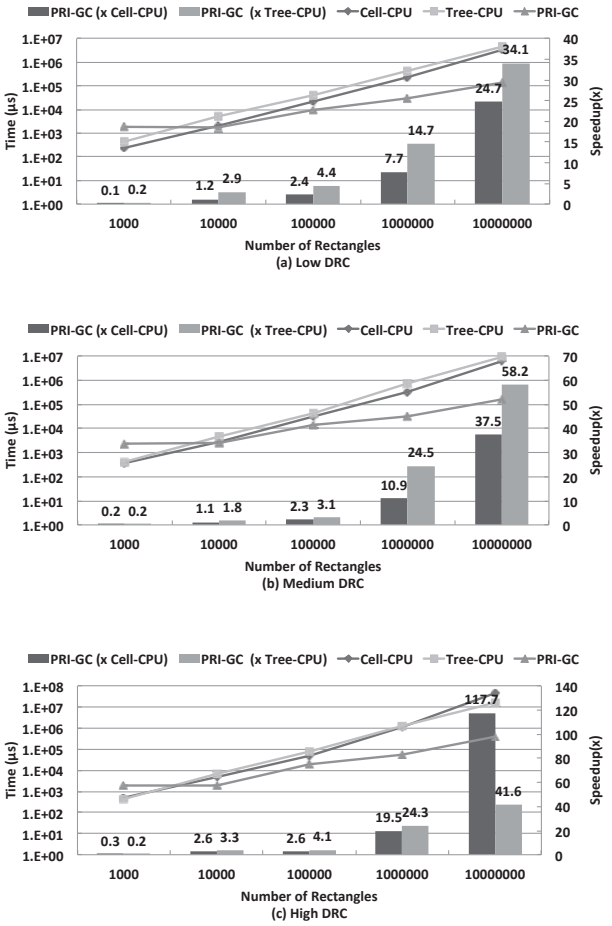

Figure 5. The execution time of Cell-CPU, Tree-CPU and PRI-GC and the speedup of PRI-GC compared with both sequential algorithms: (a) Low DRC (b) Medium DRC (c) High DRC.

Fig. 5 shows the execution time of the three algorithms for different DRC environments and the speedup of PRI-GC compared with both sequential algorithms. Fig. 6 shows the execution time (μs) taken by each task of PRI-GC in the high DRC environment and the percentage of the execution time for each task.

When *N* is small ($N=10^{3}$), PRI-GC is slower than both sequential algorithms owing to the overhead of parallelization. The major overheads are induced by block scheduling and prefix sum operations (see task Scan and Sched in figure 6), as are for $N=10^{4}$. The time by two tasks closely relate to the length of rectangle counter array (i.e., equal to 400×400).

As *N* increases, PRI-GC outperforms both sequential algorithms over 30x speedup for *N* equal to 10^{7}. In the high DRC environment, PRI-GC outperforms Cell-CPU over 100x speedup due to the significant performance degradation of Cell-CPU. The rectangle intersection time by PRI-GC

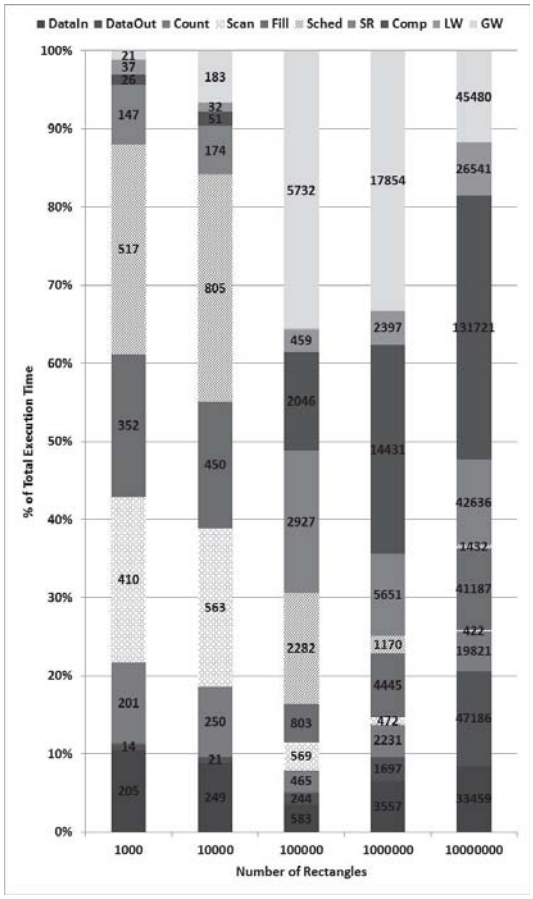

Figure 6. The execution time (μs) of PRI-GC in and the percentage of the execution time in the high DRC environment taken by the following tasks: DataIn: copy rectangle information, DataOut: copy encoded intersecting results, Count: read rectangle information and increase the cell counter, Scan: perform prefix scan, Fill: fill the rectangles into the rectangle list, Sched: do block scheduling, SR: load rectangle information into the shared memory, Comp: compare rectangles, LW: write encoded pairs of rectangles to local memory, GW: write the encoded results to the global memory.

exhibits linear growth with respect to *N*, while the rectangle intersection time by Cell-CPU exhibits near-quadratic growth with respect to *N*. Since PRI-GC and Cell-CPU are of cell-based algorithms, it proves that the design of PRI-GC is suitable (for such considerable number of rectangles) to perform rectangle intersection computations on GPUs.

Next, we present the performance of PRI-GC with varying cell sizes and varying block sizes in figure 7. The combinations of experiment configuration are listed below

1. The numbers of cells used are 25×25, 50×50, 100×100, 200×200 and 400×400.
2. The numbers of rectangles are 10^3, 10^5 and 10^7.
3. The block sizes used are 64, 128, 256, 512, and 1024.
4. Since we are interested in large output size, the high DRC environment is tested.

B. Performance with Varying Cell Sizes

In this section we present the performance change for different cell sizes in Fig 7. With different numbers of rectangles, the cell size effect on the performance is totally different.

When $N=10^3$, PRI-GC using 25×25 cells has better performance. As described in Section A, both tasks, the prefix sum operations and block scheduling, dominate the execution time. As a result, a smaller cell sizes implies less time in scanning the rectangle counter list.

When $N=10^5$, PRI-GC using 25×25 cells performs better than that using other numbers of cells. From our experimental data, with the setting T=256, PRI-GC using 25×25, 50×50, 100×100, 200×200 and 400×400 cells made a cell averagely contain 61, 37, 9, 2 and 1 rectangles, respectively. The performance results of PRI-GC using different cell sizes illustrated in Fig 7(b) match our analysis in Section IV. For better performance, a cell should contain at least 32 rectangles.

In contrast to the cases in Fig. 7(a)-(b), *N* is very large ($>10^7$) such that each cell contains at least 32 rectangles even though 400×400 cells are used. The experimental data showed that if T=256, PRI-GC using 25×25, 50×50, 100×100, 200×200 and 400×400 cells made a cell averagely contain 7498, 3794, 880, 158 and 82 rectangles, respectively. As the number of rectangles a cell contains increases, many unnecessary comparisons are performed in cells. The performance results match our analysis of cell size. PRI-GC using the 400×400 cells performs best, compared with other numbers of cells used.

C. Performance with Varying Block Sizes

Now, we investigate the block size impact on the performance of PRI-GC. The performance results are shown in Fig. 7.

The performance change shown in Fig 7(a) is not mainly determined by the block size. Most execution time is taken by prefix sum and block scheduling operations, which is related to the cell size used.

As shown in Section B, with $N=10^5$ and T=256, PRI-GC using 25×25, 50×50, 100×100, 200×200 and 400×400 cells made a cell averagely contain 61, 37, 9, 2 and 1 rectangles, respectively. Fig 7(b) shows that the performance of PRI-GC using T=64 outperforms that using other settings of T. Since the average number of rectangles a cell contains is less than the block size, only few threads within a thread block are active. If T=1024 is used, this results in poor performance due to low utilization of threads.

In Fig. 7(c), with 25×25 cells, PRI-GC using a larger block size (e.g., 1024) has better performance than that using a smaller block size (e.g., 64). In this case, *N* is large enough such that each cell, on average, contains 7498 rectangles (>1024 rectangles). Larger block size, more threads co-operate in loading rectangle information from the global memory to the shared memory. However, if 400×400 cells are used, the average numbers of rectangles (82) in a cell

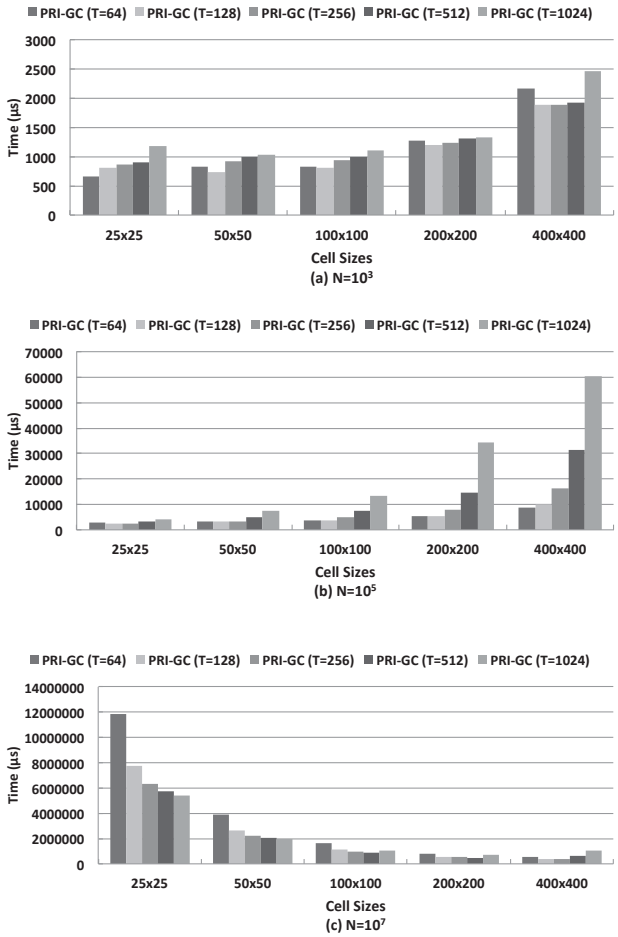

Figure 7. The execution time of PRI-GC with varying numbers of cells and varying block sizes: (a) N=10^3 (b) N=10^5 (c) N=10^7.

decreases (<256). PRI-GC using T=64 results in lower cooperativeness among threads, while PRI-GC setting T=1024 results in low utilization of threads. There is a performance tradeoff point in choosing block size. For example, if PRI-GC using 400×400 cells and T=128 or T=256, the performance is better than other parameter settings.

Consider the block size used in PRI-GC. The block size should be close to the average number of rectangles a cell contains. The block size 256 is suggested. According to our experimental results, it has in average performance in most cases and best performance in some cases.

D. Performance with optimization techniques

In this subsection, we show the performance of PRI-GC using different optimization techniques in Table I. From Table I, we can see that PRI-GC using shared memory in our algorithm deliveries up to 3-5x speedup, compared with the version without any optimization. The encoding and the overlap execution techniques can enhance about 20-40% performance gain. In the low and medium DRC environments, since few intersections are reported, little performance gain by the encoding and the overlap execution optimization is achieved. However, in the high DRC environment, the number of intersections reported is large (20 times of N). PRI-GC benefits from the encoding and the overlap execution techniques.

TABLE I. The execution time of PRI-GC using different optimization schemes in the low, medium and high DRC environments: Without any optimization, +SM: use of shared memory, +Encode: use of encoded scheme, +Overlap: use of overlap execution scheme.

PRI-GC	Low DRC	Medium DRC	High DRC
Without any optimization	452391 (1.0x)	532573 (1.0x)	2176840 (1.0x)
+SM	135876 (3.33x)	170908 (3.12x)	649523 (3.35x)
+Encode	379776 (1.19x)	438903 (1.21x)	1541160 (1.41x)
+Encode +Overlap	376801 (1.20x)	429387 (1.24x)	1470930 (1.48x)
+SM +Encode +Overlap	137435 (3.29x)	157654 (3.38x)	380694 (5.72x)

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we investigated the parallelization of rectangle intersection computations on the hybrid system: GPUs+CPU. Two major challenges for parallelizing this problem are data partition and the massive output. We presented an algorithm, called PRI-GC, which utilizes many optimization techniques for performance enhancement. First, to speed up the data access by memory coalescing and shared memory, rectangles in a cell are recorded in a continuous space. Second, to reduce the number of memory transmissions, the intersection results, computed by GPU, are encoded and compressed before writing into the global memory. Third, to overlap the execution of GPU and CPU, which is responsible for decoding and decompression, the encoded data are stored in double buffers, and the decoding and encoding processes are done on each buffer alternatively.

The performance of PRI-GC was evaluated with two well-implemented sequential algorithms, and an over 30 times speedup had been observed for ten millions rectangles. Experiments were also made to evaluate the influence of different parameters, such as the degrees of rectangle coverage (DRC), the block size, and the cell size. Finally, the effectiveness of the optimization techniques used in this algorithm was examined as well.

Owing to the importance of this problem in various applications, further investigations on scalability and the generalization of used techniques are required for developing practical packages. Other implementations of different styles of algorithms to have better load balance or processor utilization are also of our interests. Parallelizing the more

general problems, such as arbitrary-oriented rectangle intersection problem or the triangulation intersection problem in 3-D environments, are worth for further studies. Moreover, we feel that the techniques developed for this problem, such as data partition and data compression, can be used to parallelize other related problems using GPU, especially the problem with massive input and output data.

ACKNOWLEDGEMENT

The work of this paper is sponsored by National Science Council under contract NSC 100-2623-E-007-016-D. The authors would like to thank anonymous referees for their comments.

REFERENCES

[1] J. L. Bentley and T. A. Ottmann, "Algorithms for Reporting and Counting Geometric Intersections," Computers, IEEE Transactions on, vol. C-28, pp. 643-647, 1979.

[2] J. L. Bentley and D. Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," Computers, IEEE Transactions on, vol. C-29, pp. 571-577, 1980.

[3] H. W. Six and D. Wood, "The rectangle intersection problem revisited," BIT Numerical Mathematics, vol. 20, pp. 426-433, 1980.

[4] V. K. Vaishnavi and D. Wood, "Rectilinear line segment intersection, layered segment trees, and dynamization," Journal of Algorithms, vol. 3, pp. 160-176, 1982.

[5] H. Edelsbrunner, "A new approach to rectangle intersections part I," International Journal of Computer Mathematics, vol. 13, pp. 209 - 219, 1983.

[6] D. J. V. Hook, et al., "Approaches to RTI Implementation of HLA Data Distribution Management Services," in the 15th Distributed Iinteractive Simulation Workshop, 1996, pp. 96--14--084.

[7] M. D. Petty and A. Mukherjee, "Experimental Comparison of d-Rectangle Interection Algorithms Applied to HLA Data Distribution ", 1997.

[8] A. Zomorodian and H. Edelsbrunner, "Fast software for box intersections," presented at the Proceedings of the sixteenth annual symposium on Computational geometry, Clear Water Bay, Kowloon, Hong Kong, 2000.

[9] G. Tan, et al., "A hybrid approach to data distribution management," in the Fourth IEEE International Workshop on Distributed Simulation and Real-Time Applications, 2000, pp. 55-61.

[10] C. Raczy, et al., "A sort-based DDM matching algorithm for HLA," ACM Transactions on Modeling and Computer Simulation, vol. Volume 15 Issue 1, pp. 14-38, 2005.

[11] K. Pan, et al., "An Efficient Sort-Based DDM Matching Algorithm for HLA Applications with a Large Spatial Environment," in the 21st International Workshop on Principles of Advanced and Distributed Simulation, 2007, pp. 70-82.

[12] P. Gupta and R. K. Guha, "A Comparative Study of Data Distribution Management Algorithms," The Journal of Defense Modeling and Simulation on Applications, Methodology, Technology, vol. Volume 4 Issue 2, pp. 127-146, 2007.

[13] O. Eroglu, et al., "Quadtree-based approach to data distribution management for distributed simulations," presented at the the 2008 Spring simulation multiconference, Ottawa, Canada, 2008.

[14] A. L. Chow, "Parallel algorithms for geometric problems," University of Illinois at Urbana-Champaign, 1980.

[15] E. S. Liu and G. K. Theodoropoulos, "An Approach for Parallel Interest Matching in Distributed Virtual Environments," in Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, 2009, pp. 57-65.

[16] V. H. F. Batista, et al., "Parallel geometric algorithms for multi-core computers," Computational Geometry, vol. 43, pp. 663-677, 2010.

[17] L. Kettner, et al. Intersecting Sequences of dD Iso-oriented Boxes. Available: http://www.cgal.org/Manual/latest/doc_html/cgal_manual/Box_intersection_d/Chapter_main.html

[18] M. I. Shamos and D. Hoey, "Geometric intersection problems," in Foundations of Computer Science, 1976., 17th Annual Symposium on, 1976, pp. 208-215.

[19] M. d. Berg, et al., Computational geometry : algorithms and applications, 3rd ed. Berlin: Springer-Verlag, 2008.

[20] A. Boukerche, et al., "Grid-Filtered Region-Based Data Distribution Management in Large-Scale Distributed Simulation Systems," in the 38th Annual Simulation Symposium, 2005, pp. 259-266.

[21] C. L. Ming and G. Stefan, "Collision Detection Between Geometric Models: A Survey," in IMA Conference on Mathematics of Surfaces, San Diego, CA, 1998.

[22] M. H. Overmars, "Point location in fat subdivisions," Inf. Process. Lett., vol. 44, pp. 261-265, 1992.

[23] The OpenMP API Specification for Parallel Programming. Available: http://openmp.org/wp/