Optimizing Pairwise Box Intersection Checking on GPUs for Large-Scale Simulations

SHIH-HSIANG LO and CHE-RUNG LEE, National Tsing Hua University
I-HSIN CHUNG, IBM T.J. Watson Research Center
YEH-CHING CHUNG, National Tsing Hua University

Box intersection checking is a common task used in many large-scale simulations. Traditional methods cannot provide fast box intersection checking with large-scale datasets. This article presents a parallel algorithm to perform Pairwise Box Intersection checking on Graphics processing units (PBIG). The PBIG algorithm consists of three phases: planning, mapping and checking. The planning phase partitions the space into small cells, the sizes of which are determined to optimize performance. The mapping phase maps the boxes into the cells. The checking phase examines the box intersections in the same cell. Several performance optimizations, including load-balancing, output data compression/encoding, and pipelined execution, are presented for the PBIG algorithm. The experimental results show that the PBIG algorithm can process large-scale datasets and outperforms three well-performing algorithms.

Categories and Subject Descriptors: I.3.1 [**Computer Graphics**]: Hardware Architecture—*Parallel processing*; I.6.7 [**Simulation and Modeling**]: Simulation Support Systems

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Box intersection checking, load-balancing, data compression, pipelined execution

ACM Reference Format:
Lo, S.-H., Lee, C.-R., Chung, I.-H., and Chung, Y.-C. 2013. Optimizing pairwise box intersection checking on GPUs for large-scale simulations. *ACM Trans. Model. Comput. Simul.* 23, 3, Article 19 (July 2013), 22 pages.
DOI: http://dx.doi.org/10.1145/2499913.2499918

1. INTRODUCTION

Box intersection checking is the process of finding all intersecting pairs of iso-oriented boxes in a space. It has many applications in computer simulations, including motion simulation, geometry modeling, computer games, many-body dynamics analysis and granular dynamics. In motion simulations, an intersection query is a mechanism used to determine the movement and interaction of entities with one another. In particle simulations, the distinct-element method (DEM) requires millions of particles for realistic results of fluids and granular materials [Harada 2007]. In massively multiplayer online role-playing games [EVE Online 2003], tens of thousands of players, entering the same server, interact with other players and many computer-generated objects. In high-level architecture (HLA) compliant simulations, the intersection checking is

The authors would like to thank the CUDA Center of Excellence (CCOE) at National Tsing Hua University and National Science Council of Taiwan (under Contract No. 101-2115-M-007-004-MY2) for financially/partially supporting this research.
Authors' addresses: S.-H. Lo, C.-R. Lee, and Y.-C. Chung, Computer Science Department, National Tsing Hua University, Hsinchu, Taiwan; email: albert@sslab.cs.nthu.edu.tw; I.-H. Chung, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

DOI: http://dx.doi.org/10.1145/2499913.2499918

performed to reduce unnecessary communication between simulation federates [Liu and Theodoropoulos 2009; Santoro and Fujimoto 2008]. These applications require an efficient box intersection checking algorithm for large-scale datasets.

This article presents an algorithm, called PBIG, to optimize iso-oriented box intersection checking using graphics processing units (GPUs). Three specific reasons motivate this direction of research. First, when the number of entities scales substantially, the intersection calculation becomes one of performance bottlenecks [Avril et al. 2010], which makes real-time simulation difficult. Second, pairwise intersection checking falls under the scope of data parallelism, which fits naturally into the massive parallel computing environments such as GPUs. Third, the maturity of programmable graphic hardware allows users to utilize massively parallel processing units in GPUs based on a single-instruction multiple-thread (SIMT) programming model.

Using GPUs to perform box intersection checking is not new. Several GPU-based algorithms [BULLET 2011; Le Grand 2007; Liu et al. 2010] have been proposed to accelerate broad-phase collision detection for graphics applications. These algorithms adopt the uniform partition strategy to fit the GPU execution; however, they do not address the issues that have arisen for large-scale datasets, such as the large output problem. Although the GPU-based algorithm [Lo et al. 2011] can handle large datasets for 2-D rectangle intersection checking on GPUs, it causes workload imbalances among threads, and its applicability is limited to certain types of datasets.

The proposed algorithm consists of three phases: planning, mapping and checking. The planning phase determines an optimal cell size, which is the key factor in reducing unnecessary computations. Based on the cell size, the space is partitioned into smaller cells. The mapping phase finds the cells that the boxes occupy and stores the boxes in a continuous space for better memory access. The checking phase performs pairwise checks of the boxes in the same cell and employ a load-balancing scheme to equalize the workload among threads. To attack the large output problem, the GPU encodes the intersection results using a 3-level compression scheme, which the CPU reports (decodes). In addition, to hide the data transmission and decoding overheads in the GPU execution, the proposed algorithm enables a pipelined execution of the GPU execution, data transmission and CPU execution.

We compare the PBIG implementation with one sequential implementation and two GPU-accelerating implementations. The sequential implementation [Kettner et al. 2011] from the computational geometry algorithms library [CGAL 2011] is proven to be one of the fastest implementations in practice. The first GPU-based implementation, called BULLET, is from the bullet physics engine library [BULLET 2011]; the second implementation, called PRI, extends the implementation in a previous work [Lo et al. 2011] for the 3-D environment. The comparisons were performed on different problem settings, including the number, density, and distribution of boxes. The experimental results show that the proposed algorithm has up to 110× faster performance than the sequential algorithm and up to 10× and 3× performance improvements over the BULLET and PRI, respectively.

This article makes two major contributions. First, we propose an efficient implementation to optimize box intersection checking on GPUs, which can be used in large-scale, real-time simulations. The implementation is available as open-source for download at Google-Code [PBIG 2013]. Second, we present analyses and techniques to address the performance challenges in implementing box intersection checking algorithms on GPUs. Those analyses and techniques can be used for box intersection checking and many related problems implemented on GPUs.

The remainder of this article is organized as follows. In Section 2, we survey the sequential and parallel box intersection checking algorithms reported in the literature. In Section 3, we introduce the GPU architecture used, CUDA, and the challenges in

implementing box intersection checking algorithms on GPUs. In Section 4, we present our algorithm PBIG in detail. In Section 5, we explain the performance optimization methods developed for the PBIG algorithm. In Section 6, we discuss the experimental results for performance evaluation. Section 7 presents our conclusion and directions for future work.

2. RELATED WORK

The box intersection checking problem has received many investigations, including analyses on its time complexity [Guting and Wood 1984; Petty and Morse 2004] and algorithms to solve it. In this section, we provide a brief survey of sequential and parallel algorithms for box intersection checking.

The sequential algorithms can be roughly classified into the following categories according to the techniques used.

— sweep-line-based algorithms: Bentley and Ottmann [1979], Bentley and Wood [1980], Six and Wood [1980];
— tree-based algorithms: Vaishnavi and Wood [1982], Edelsbrunner [1983], Petty and Mukherjee [1997], Zomorodian and Edelsbrunner [2002];
— uniform partition-based algorithms: Van Hook et al. [1996], Tan et al. [2000], Boukerche et al. [2005];
— sort-based algorithms: Raczy et al. [2005], Gupta and Guha [2007], Pan et al. [2011].

Among these sequential algorithms, Zomorodian and Edelsbrunner [2002] presented a fast algorithm in practice and its implementation [Kettner et al. 2011] is widely available for research.

The parallel box intersection checking algorithms have also been widely studied. Chow [1980] proposed three parallel algorithms using two theoretical models: the parallel random access machine (PRAM) and cube-connected cycles (CCC) models. For shared memory multiprocessor systems, Overmars [1992] used the flat subdivision and insertion-sort techniques to find box intersections. Liu and Theodoropoulos presented parallel matching algorithms using the insertion-sort technique on a multicore platform [Liu and Theodoropoulos 2009] and on distributed systems [Liu and Theodoropoulos 2011]. Batista et al. [2010] also presented a parallel algorithm for d-D box intersection checking, which is based on the efficient sequential algorithm [Zomorodian and Edelsbrunner 2002]. Their parallel algorithm divides box interval sequences into subtasks and handles the subtasks using the OpenMP framework.

To handle large datasets, the GPU-accelerating algorithms have been proposed in the literature [BULLET 2011; Le Grand 2007; Liu et al. 2010; Lo et al. 2011]. Le Grand [2007] employed a uniform partition strategy to separate boxes into different cells and sorted the cells to filter out empty ones. The algorithm uses a static cell size, such that a box crosses at most 2^d cells, where d is the number of dimensions. Such a cell size introduces unnecessary computations when box sizes vary. The bullet physics library [BULLET 2011] provides a similar algorithm to perform broad-phase collision detection for graphics applications on GPUs. The boxes that cross a certain number of cells, which are considered large boxes, are directly matched with one another without mapping them into cells. Liu et al. [2010] presented an algorithm that applies a coarse partition strategy, which divides the space into subdivisions and then employs the sweep-line technique to find box intersections for each subdivision. Lo et al. [2011] proposed a 2-D rectangle intersection algorithm on GPUs for overcrowded situations, in which many intersecting box pairs are reported. However, the presented algorithm introduced load imbalances among computing units, and the compression scheme used was ineffective in handling nonuniform datasets.

3. BOX INTERSECTION CHECKING ON GPUS

3.1. CUDA Overview

Compute unified device architecture (CUDA) is a parallel computing architecture with an array of streaming multiprocessors (SMs) and various memory spaces for execution. In the CUDA architecture, threads are assigned to groups (called warps) and executed by SMs in the device (i.e., the CUDA-enabled product). Because the CUDA architecture employs single-instruction multiple-threads (SIMT) paradigm, all the threads in a warp synchronously perform one common instruction. Each SM can execute one or more warps concurrently. For memory access, threads can access data in different levels of the device memory hierarchy, including registers, shared memory, cache, constant memory, texture memory and global memory.

To perform tasks using CUDA, the host (i.e., the computer system) copies data from the host memory to the device memory. The host then invokes a GPU kernel function with a specified execution configuration. The execution configuration defines the thread organization (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, each SM receives a certain number of thread blocks and arranges those thread blocks into warps for execution. The result is subsequently copied from the device memory to the host memory.

Two key issues concern application performance on CUDA. One is branch divergence. If the threads in a warp have different execution paths because of a data-dependent condition, the threads must take turns performing instructions (i.e., if- and else-parts). As the threads encounter more branch divergences, the execution becomes more serialized. Another issue is memory access. Threads should reuse the data in the low-latency memory (e.g., shared memory or cache) as opposed to the high-latency memory (e.g., global memory). If accessing global memory is required, coalescing memory accesses can reduce the number of memory transactions.

3.2. Computational Challenges

There are two major computational challenges in implementing box intersection checking algorithms on GPUs.

(1) *Data partitioning.* A large dataset must be divided into small datasets for parallel processing. Traditional adaptive partition techniques, such as binary space partitioning (BSP), divide the space into subspaces and let each subspace contain an equal number of boxes. However, such adaptive techniques favor and fit CPU execution but not GPU execution. Static partition techniques have better parallelism for GPU [Ming and Stefan 1998] but could cause load imbalances among the partitions. If few threads within a thread block perform tasks or if certain of the threads have heavy workloads, this situation causes GPU resources to idle. The load imbalances situation worsens for box intersection checking because it has the symmetric property and performs symmetric data operations.

(2) *Large output processing.* The number of intersecting box pairs can be large and unpredictable. Outputting all intersecting box pairs becomes a crucial factor for performance. For N boxes, the output size is generally $O(N)$ [Zomorodian and Edelsbrunner 2002]. When N is large, the output size also grows, which degrades the overall performance on GPUs. Allocating the memory for the output also affects the performance. A common allocation method is to create a designated space for each box. This method wastes memory space and requires post-processing when an application has few intersecting pairs or features a non-uniform distribution of boxes. When N is large, the required memory space cannot fit into the GPU memory.

Table I. List of Symbols

Symbol	Meaning
N	number of boxes
N_c	number of boxes in a cell c
M	number of cells
α	average number of cells a box occupies
f	objective function to minimize the number of intersection checks
W	number of threads per warp
T	number of threads per thread block
ℓ_s^i	length of a space s in the ith dimension
ℓ_c^i	length of a cell c in the ith dimension
ℓ_b^i	length of a box b in the ith dimension
p_b^i	lower coordinate of a box b in the ith dimension
q_b^i	upper coordinate of a box b in the ith dimension

4. PBIG ALGORITHM

This paper considers the box intersection checking problem: given N iso-oriented boxes in a 3-D space, find all the intersecting box pairs. Each box is numbered by a unique identification (box ID). The position of a box b is defined by three pairs of real numbers, (p_b^1, q_b^1), (p_b^2, q_b^2), and (p_b^3, q_b^3), which denote the lower and upper coordinates of box b in each dimension. Two boxes b_1 and b_2 intersect if and only if for $i = 1, 2, 3$,

$$p_{b_1}^i \le q_{b_2}^i \text{ and } p_{b_2}^i \le q_{b_1}^i. \tag{1}$$

This section presents the PBIG algorithm to efficiently perform box intersection checking on GPUs. The presented algorithm requires all the information of boxes in the GPU memory and comprises three phases to obtain the intersection results (i.e., intersecting box IDs) in an intersection list. The intersection list is an array in the GPU global memory. The following describes the three phases, planning, mapping and checking. To clarify the description, Table I lists the symbols used in this section.

4.1. Planning Phase

The planning phase determines a cell size for space partitioning. The space partitioning allows comparing boxes with other boxes in the same cell only, which reduces unnecessary computations in the checking phase. Cell size is thus crucial to intersection checking performance. If the cell size is too large, the number of unnecessary intersection checks (UCs), which compare pairs of distant boxes, becomes large. Conversely, if the cell size is too small, many redundant intersection checks (RCs) are performed. That is, the same box pair, intersecting or not, is checked many times in different cells. The cell size thus significantly influences the number of intersection checks performed.

In this article, the 3-D space is partitioned into uniform cells, and the size of each cell c is $\ell_c^1 \times \ell_c^2 \times \ell_c^3$, where ℓ_c^i is the length of c in the ith dimension and symbol $\times$ stands for the multiplication sign between scalars. Finding the optimal cell size that minimizes the number of unwanted intersection checks, UCs and RCs, can be formulated as an optimization problem,

$$\min_{\ell_c^1, \ell_c^2, \ell_c^3} f = \frac{N\alpha}{M}, \tag{2}$$

where f is the objective function, N is the number of boxes, M is the number of cells, and α is the average number of cells that a box occupies. The value of N/M means

the average number of boxes per cell. A larger N/M indicates that more unnecessary checks (UCs) can occur. On the other hand, a larger α implies that more redundant checks (RCs) can occur. The objective function f is therefore designed to minimize both unwanted checks.

In a 3-D space s of size $\ell_s^1 \times \ell_s^2 \times \ell_s^3$, where ℓ_s^i is the length of s in the ith dimension and symbol $\times$ stands for the multiplication sign between scalars, M can be calculated as

$$M = \left\lceil \frac{\ell_s^1}{\ell_c^1} \right\rceil \left\lceil \frac{\ell_s^2}{\ell_c^2} \right\rceil \left\lceil \frac{\ell_s^3}{\ell_c^3} \right\rceil. \tag{3}$$

Similarly, given the size of a box b, $\ell_b^1 \times \ell_b^2 \times \ell_b^3$, where ℓ_b^i is the length of b in the ith dimension and symbol $\times$ stands for the multiplication sign between scalars, the number of cells occupied by b can be estimated using

$$\left\lceil \frac{\ell_b^1}{\ell_c^1} \right\rceil \left\lceil \frac{\ell_b^2}{\ell_c^2} \right\rceil \left\lceil \frac{\ell_b^3}{\ell_c^3} \right\rceil. \tag{4}$$

Thus, α is defined as

$$\alpha = \frac{1}{N} \sum_{b \in B} \left(\left\lceil \frac{\ell_b^1}{\ell_c^1} \right\rceil \left\lceil \frac{\ell_b^2}{\ell_c^2} \right\rceil \left\lceil \frac{\ell_b^3}{\ell_c^3} \right\rceil \right), \tag{5}$$

where B denotes the set of N boxes in a space.

The relationship between cell size and box intersection checking performance is even more complicated on the GPU because GPU hardware efficiency must also be considered. The CUDA architecture employs the SIMT computing model, in which a warp (i.e., a group of 32 threads) executes one common instruction synchronously. If a cell contains fewer boxes than the number of threads in a warp W, hardware resource utilization is low. The problem in Eq. (2) can be therefore reformatted as

$$\min_{\ell_c^1, \ell_c^2, \ell_c^3} f = \frac{N\alpha}{M}, \text{ subject to } : f \geq W. \tag{6}$$

If we set all ℓ_c^i values to be equal, we can solve this optimization problem using a binary search on ℓ_c^i. The most time-consuming task in Eq. (6) is computing α, the time complexity of which is $O(N)$ per trial of ℓ_c^i. To reduce the computational cost, we take a simple random sample of size 1000 from N boxes to estimate α and offload computing α onto the GPU.

4.2. Mapping Phase

The mapping phase determines a list of occupied cells for each box and generates a list of boxes for each cell c to store the boxes in c. Since all of the cells have equal and fixed size, the mapping phase can determine the cells that the boxes occupy in parallel. The difficulty is efficiently converting the lists of cells for all boxes to the lists of boxes for all cells on GPUs. The lists of boxes are concatenated in a continuous memory space for performance, which is discussed in this article.

To achieve these goals (i.e., parallelization, one box list per cell, and a concatenated storage), three arrays are used. First, the *box array* stores the concatenated box lists for all cells. Second, the *offset array* keeps the beginning address of each box list in the box array. Third, the *counter array*, which is an auxiliary array, records the number

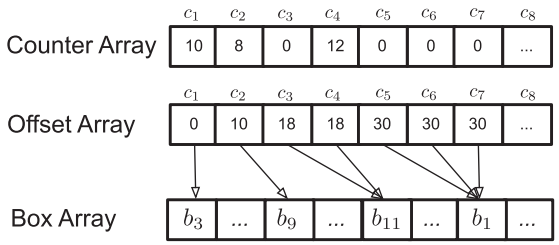

Fig. 1. Example of the counter, offset, and box arrays. Boxes are concatenated in the box array according to the cell order.

of boxes in each cell. Figure 1 shows an example: cell c_1 contains 10 boxes and the beginning address (i.e., the offset) of its box list in the box array is 0; cell c_2 contains 8 boxes and its offset is 10; cell c_3 has no boxes in it, giving cells c_3 and c_4 the same offset.

The mapping phase uses three GPU kernels to produce the desired arrays.

Box Counting Kernel. The box counting kernel generates the counter array. Each thread handles a box and computes the cells it occupies. For an occupied cell of index i, the thread increases the ith element in the counter array. This kernel uses one atomic instruction to guarantee counter array correctness because it is possible for multiple threads to simultaneously access the same counter.

Offset Calculating Kernel. This kernel generates the offset array by performing the parallel prefix sum algorithm [Harris et al. 2007] on the counter array.

Box Mapping Kernel. The box mapping kernel fills the box array with the boxes. The position of a box in the box array corresponds to the offset of the cell (in the offset array) and the order of executing one atomic instruction. Each thread performs one atomic instruction to acquire the order instead of recording the order obtained in the box counting kernel.

We briefly discuss two implementation decisions made for this phase. The first decision is concerned with the data structure of box lists. For better memory access, all the box lists are concatenated in a continuous array, which is statically allocated.[1] Although CUDA provides a dynamical allocation method[2] for GPU kernel functions, simultaneously creating dynamic arrays introduces a heavy overhead [Huang et al. 2010]. We further conduct an experimental test to measure the overhead. In the test case, the number of boxes is 10^7 and other settings are the same as in Section 6.1. The result shows the dynamic allocation method takes 95 seconds for memory allocation, which is extremely slow in comparison with 1 millisecond by the static allocation method.

The second decision is the use of atomic instructions in the box counting and box mapping kernels. It is generally believed that atomic instructions harm performance because they serialize the memory accesses to the same address. However, using atomic instructions is a better choice. An alternative approach that avoids atomic

[1]Before the box mapping kernel execution, the host uses cudaMalloc() function [CUDA 2011] to allocate a device memory space.

[2]In the CUDA programming model, a GPU thread can dynamically allocate a device memory space from a fixed-size heap using the device malloc() function.

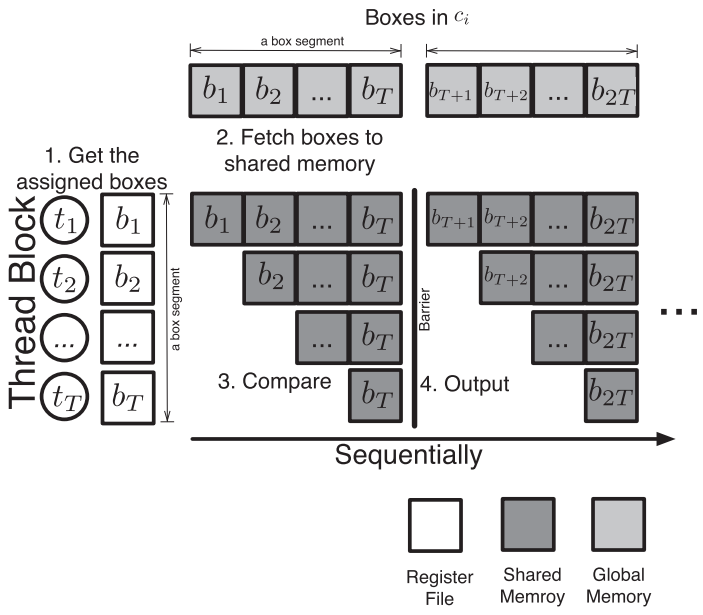

Fig. 2. Execution diagram of parallel box intersection checking.

instructions is the scan technique [Merrill and Grimshaw 2011], in which each thread uses a temporary space of size O(M) to record the number of boxes in each cell for M cells. Since M is large, the total temporary space must reside in the global (off-chip) memory, which causes great performance degradation. On the other hand, if atomic instructions are used, there is a small chance that there are simultaneous memory accesses to the same address. The reason is that the cell size derived from the planning phase controls the number of boxes per cell. The performance slowdown caused by atomic instructions is relatively small.

4.3. Checking Phase

This phase uses a checking kernel to perform pairwise box intersection checks for a set of cells C. Each cell c in C is assigned to a certain number of thread blocks for the checking kernel execution. If cell c contains N_c boxes, the boxes in c are divided into $\lceil N_c/T \rceil$ box segments. The size of a box segment equals the number of threads per thread block T. Each box segment is assigned to a thread block such that each box is assigned to one thread. This assignment is achieved using dynamic thread block scheduling. Figure 2 illustrates an execution diagram of the checking kernel, and Algorithm 1 presents the pseudocode of the checking kernel.

In the checking kernel, each thread block compares the assigned box segment with all the box segments in the same cell. Each thread first receives the assigned cell c and fetches the assigned box in cell c from the global memory. Threads cooperatively load all of the boxes within a box segment into the shared memory. (Loading T boxes into the shared memory enables the threads to compare approximately $T^2/2$ box pairs with fast memory access.) Block-level barrier synchronization is required to wait for all of the boxes within a box segment to be loaded into the shared memory. Each thread then compares the assigned box against the boxes residing in the shared memory. Equation (1) is used to check the intersection of two boxes. When one intersecting box pair is found, the box pair is temporarily stored in the thread-local memory.

ALGORITHM 1: Parallel Box Intersection Checking Kernel

```
Data: C: a set of cells to be checked
Data: c: the assigned cell c, where c ∈ C
Data: seg_v: the assigned box segment seg_v in c
Data: tid: the thread index in a thread block
Data: cnt: a temporary variable indicating the number of intersecting box pairs
Data: pairs: a temporary array to hold intersecting box IDs in the thread-local memory
Data: shbox: a temporary array to hold boxes in the shared memory
Data: shcnt: a temporary array to hold counter values in the shared memory
Result: IDs of intersecting box pairs get recorded in the intersection list L
1  i ← tid
2  b_i ← seg_v[i] // load the assigned box
3  for each box segment seg_h in c do
4      n ← the number of boxes in seg_h
5      cnt ← 0
6      shbox[tid] ← seg_h[tid] // load one box into the shared memory
7      Perform block-level barrier synchronization
8      j ← tid
9      while j < n do
10         b_j ← shbox[j]
11         Compare b_i and b_j using Equation (1)
12         Calculate the cell ĉ where the box pair (b_i,b_j) can be output
13         if b_i is intersected by b_j, b_i ≠ b_j, and c = ĉ then
14             Output IDs of b_i and b_j to pairs
15             cnt ← cnt + 1
16         end
17         j ← j + 1
18     end
19     shcnt[tid] ← cnt
20     Perform block-level barrier synchronization
21     Get the thread's output offset pos_t by computing a prefix sum on shcnt
22     Get the thread block's output offset pos_b in L using an atomic instruction
23     Output pairs to L from the offset pos = pos_t + pos_b
24 end
```

To avoid redundant box intersection output, the checking kernel must require two procedures. First, the box intersection checking has the symmetric property, which means that if box b_1 intersects with box b_2, box b_2 must intersect with box b_1. Only half of box pairs in each cell must therefore be checked. Threads compare box pairs (b_i,b_j) for $i \leq j$, where i and j are the box indices within a box segment, as in Figure 2. (Note that boxes b_i and b_j could be in different box segments.) Second, because boxes could reside in one or more cells, different thread blocks could find the same intersecting pair. Only the thread in the output cell (denoted $\hat{c}$ in Algorithm 1), where the corner of the intersection of two boxes with the largest coordinates resides, must output the intersecting pair. All other threads could compare the same pair without outputting it.

When completing the checks for one box segment, all of the threads within a thread block output the intersection results to the intersection list. To parallelize the output process, each thread must know the output address (offset) of the intersection list for outputting its intersection result. An output offset comprises two parts: the thread offset in a thread block and the thread block offset in a kernel grid. Threads' offsets are obtained by computing prefix sums, and thread blocks' offsets are obtained using atomic instructions by the first thread within a thread block.

This difference in how the offsets are calculated is a result of the different parallelism granularities. For the threads, the number of their offsets is massive and their action is more synchronous. For the thread blocks, the number of their offsets is much smaller and their execution is more asynchronous. If the checking kernel obtains the offset of each thread block by computing prefix sums, the output process must wait for the checks performed by all thread blocks to finish, which is less efficient than the atomic approach.

4.4. Application Considerations

Although box intersection checking is a common task in applications, the PBIG algorithm has three specific considerations when applied to various applications.

First, some applications exhibit a strong temporal consistency, as described by Cohen et al. [1995]. Exploiting the temporal consistency of objects' locations can usually accelerate box intersection checking. For instance, Liu and Theodoropoulos [2009] cache the intersection results of the previous iteration and use the insertion-sort technique to reduce the computational complexity. To integrate this idea, a dynamic data structure (i.e., a container) on GPUs is required for updating the intersection results from iteration to iteration. However, GPU implementations of such a structure are expensive because updating the structure involves non-uniform memory accesses [Lefohn et al. 2005] and branch divergences. The PBIG algorithm does not exploit the temporal consistency property of applications on current GPUs.

Second, the PBIG algorithm can be performed in either a centralized platform or distributed platforms. The algorithm requirement is the GPU hardware. In a distributed simulation, such as an HLA federation, the box intersection checking can be performed in either the Central RTI Component (CRC) or the Local RTI Components (LRCs). If the PBIG algorithm is performed in the CRC, a single GPU device in the CRC is required to perform intersection checking. With multiple GPU devices, the PBIG algorithm can be performed in multiple LRCs.

Finally, PBIG can be easily generalized to perform the box intersection checking in d-dimension. These are the required modifications.

— The planning phase considers d dimensions to compute Eqs. (3), (5), and (6).
— The mapping phase considers d coordinate pairs of a box to determine the occupied cells of the box.
— The checking phase considers d dimensions in Eq. (1) to perform an intersection check of two boxes.

5. PERFORMANCE OPTIMIZATIONS

We present three optimization techniques to further enhance the PBIG performance. The first technique handles a load-balancing issue for threads and warps. The second technique uses data compression to reduce the communication cost of outputting intersecting pairs. The third technique enables a 3-stage pipelined execution to overlap communication and computation.

5.1. Load-Balancing

Because of the symmetric property of box intersection checking, threads only compare box pairs (b_i,b_j) for $i \leq j$, where i and j are the box indices within a box segment. As in Figure 2, only the box pairs in the top-right part are checked. The workload assignment presented in Section 4.3 is to simply assign box pairs in one row to one thread. The advantage of the assignment is that all the required data can be preloaded into the shared memory and reused many times. The assignment, however,

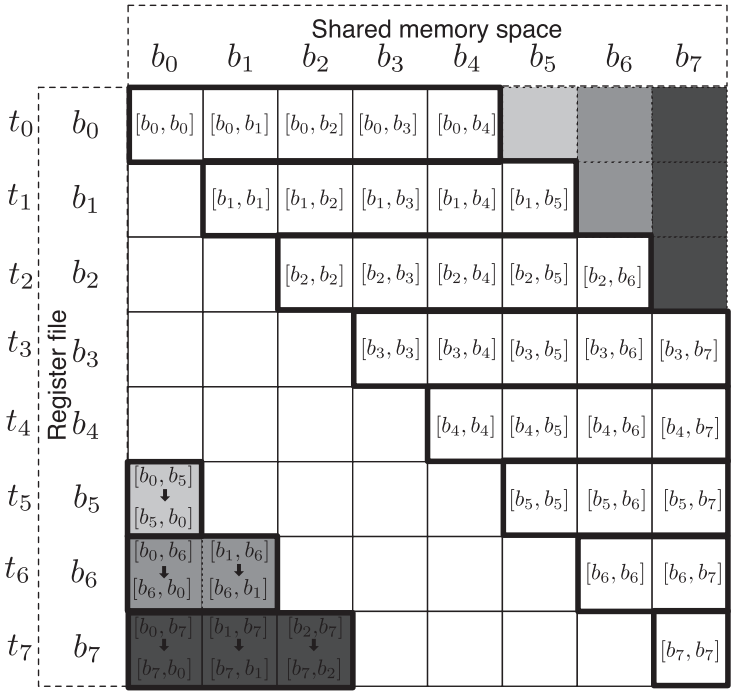

Fig. 3. Example of the load-balancing assignment for an 8×8 matrix. Eight GPU threads compare the assigned box segment with one box segment. Each of the top-half threads compares 5 pairs; each of the bottom-half threads compares 4 pairs.

causes load imbalances among threads because some threads check more box pairs than other threads.

To achieve better load-balancing among threads, we present a new workload assignment. In the new assignment, pairwise checks for two box segments of sizes m and n are represented as an $m \times n$ matrix, in which each element is a box pair. Each thread handles one row of a matrix, but some top-right matrix elements are shifted to the bottom-left part. Figure 3 demonstrates this new assignment for an 8×8 matrix. In Figure 3, the elements $(0,5), (0,6), (0,7), (1,6), (1,7)$, and $(2,7)$ are shifted to $(5,0), (6,0), (7,0), (6,1), (7,1)$, and $(7,2)$, respectively.

Given an $m \times n$ matrix, the number x of matrix elements to be checked in ith row is

$$x = \begin{cases} (n-m) + \lceil (m+1)/2 \rceil & \text{if } i \leq \lceil m/2 \rceil \\ (n-m) + \lfloor (m+1)/2 \rfloor & \text{otherwise} \end{cases}, \tag{7}$$

where m is the number of rows of the matrix, n is the number of columns of the matrix, and $m \leq n$. The reason why $m \leq n$ is that each thread only compares box pairs from the diagonal element, as in Figure 3. The number of box pairs assigned to a thread depends on the row the thread handles. This new assignment equalizes the box pairs among threads. In addition, since each thread still handles one row of the matrix, all the required data also can be preloaded into the shared memory and reused many times, as in the original assignment.

Algorithm 2 presents the details of the checking kernel with the proposed workload assignment. The workload assignment requires two operations. First, each thread compares box pairs from the diagonal box pairs until it finishes comparing x box pairs, that

ALGORITHM 2: Parallel Box Intersection Checking Kernel with Load-Balancing

Data: C: a set of cells to be checked
Data: c: the assigned cell c, where $c \in C$
Data: seg_v: the assigned box segment seg_v in c
Data: tid: the thread index in a thread block
Data: cnt: a temporary variable indicating the number of intersecting box pairs
Data: $pairs$: a temporary array to hold intersecting box IDs in the thread-local memory
Data: $shbox$: a temporary array to hold boxes in the shared memory
Data: $shcnt$: a temporary array to hold counter values in the shared memory
Result: IDs of intersecting box pairs get recorded in the intersection list L

```
1  i ← tid
2  b_i ← seg_v[i] // load the assigned box
3  for each box segment seg_h in c do
4      n ← the number of boxes in seg_h
5      m ← min(the number of boxes in seg_v, n)
6      cnt ← 0
7      j ← tid, k ← 0
8      shbox[tid] ← seg_h[tid] // load one box into the shared memory
9      Perform block-level barrier synchronization
       // Compare x box pairs
10     Compute the number x of box pairs assigned for an m × n matrix using Equation (7)
11     while i < n and k < x do
12         if j = n then
13             j ← 0 // shift to the first box in seg_h
14         end
15         b_j ← shbox[j]
16         Compare b_i and b_j using Equation (1)
17         Calculate the cell ĉ where the box pair (b_i,b_j) can be output
18         if b_i is intersected by b_j, b_i ≠ b_j, and c = ĉ then
19             Output IDs of b_i and b_j to pairs
20             cnt ← cnt + 1
21         end
22         j ← j + 1, k ← k + 1
23     end
24     shcnt[tid] ← cnt
25     Perform block-level barrier synchronization
26     Get the thread's output offset pos_t by computing a prefix sum on shcnt
27     Get the thread block's output offset pos_b in L using an atomic instruction
28     Output pairs to L from the offset pos = pos_t + pos_b
29 end
```

is, lines 10–11 in Algorithm 2. Second, when reaching the boundary of a box segment, threads continue to compare other box pairs by shifting to the first box in the box segment, that is, lines 12–14 in Algorithm 2.

5.2. Data Compression

To reduce the output size, we design a hierarchical coding scheme for box IDs, under which each box ID can be represented by a three-tuple

$$\text{box ID} = (\text{cell ID, segment ID, local ID}). \tag{8}$$

The first level ID is the cell ID, obtained after the mapping phase. In each cell, we use the box indices according to the order in which they are being mapped into cells.

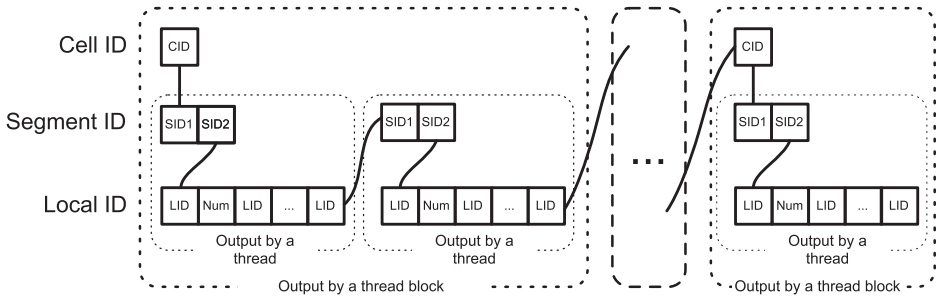

Fig. 4. The encoded intersection list. Each box ID is represented by a three-tuple (cell ID, segment ID, and local ID).

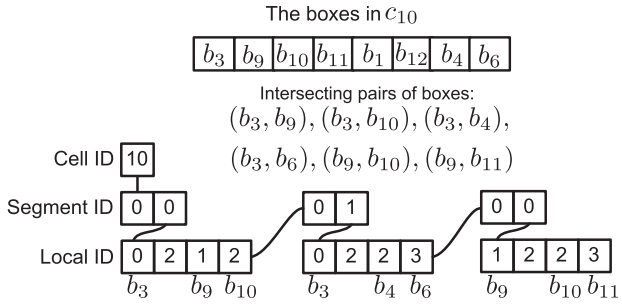

Fig. 5. Example of the encoded intersection list. Six intersecting box pairs are encoded, and the length of a box segment is 4.

Because the number of boxes in a cell can be large and varied, we further partition the boxes into segments. (The length of a box segment can be equal to or smaller than the length of a box segment used in Section 4.3. In this article, we set them to be equal.) Each segment can contain only a fixed number of boxes, for example, 256. For a box with a new index *idx* in a cell, its segment ID is $\lfloor idx/256 \rfloor$, and its local ID is $\text{mod}(idx, 256)$. Each box in a cell thus has a unique pair of segment ID and local ID.

Figure 4 shows how box IDs are encoded using the hierarchical IDs. First, a thread block performs intersection checks for a box segment in a cell at a time, and therefore only one thread in a thread block outputs the cell ID. Second, because each thread compares one assigned box with the other boxes, it can output the segment IDs of the assigned box and intersecting boxes within a box segment only once. Finally, we combine the local ID of the assigned box, the number of intersecting boxes and the local IDs of the intersecting boxes. The number of intersecting boxes is output for decompression.

Figure 5 shows an example of encoding box IDs. Suppose that the length of a box segment is 4; there are 8 boxes in cell c_{10} and the intersecting box pairs in c_{10} are as shown in Figure 5. When a thread finds an intersecting box pair, for example, (b_3, b_4), it stores the segment IDs of b_3 and b_4 (i.e., SID1=0 and SID2=1 in Figure 5), the local IDs of b_3 and b_4 (i.e., 0 and 2), and the number of intersecting pairs for b_3 in the thread-local memory. The encoded intersection results in the thread-local memory are output to the intersection list, which is attached to the cell ID of the current cell (i.e., 10 in this example).

The CPU is responsible for decoding the encoded intersection results. After the mapping phase, the CPU obtains (duplicates) the decoding information, including the offset

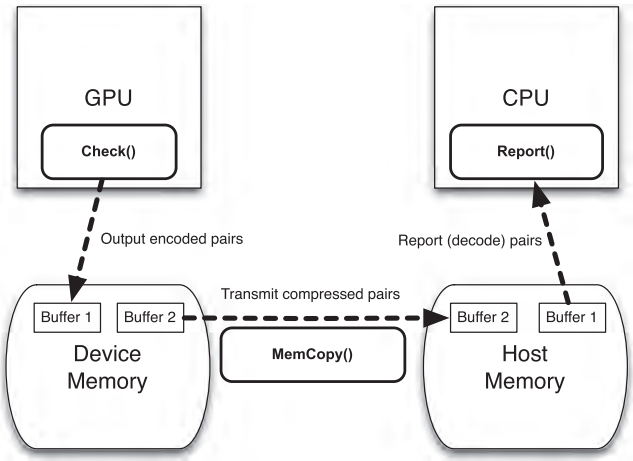

Fig. 6. Double-buffering. Two device buffers and two host buffers enable three tasks (the checking kernel, data transmission and reporting (including decoding)) to access different buffers simultaneously.

and box arrays defined in Section 4.2. Combing with the segment ID, box segment size, and local ID, the CPU can locate the original box IDs of intersecting boxes.

Suppose that a box ID uses x memory bits, a cell ID uses y memory bits, and a segment ID uses z memory bits. Let L be the maximum number of boxes a segment can keep. One local ID requires $\lceil \log_2 L \rceil$ memory bits. In a segment, the maximum number of intersecting pairs is L. The number of intersecting pairs of a box is thus encoded into $\lceil \log_2 L \rceil$ memory bits. To output k intersecting box pairs in a segment, the compression ratio of the compressed to uncompressed output size is

$$\frac{y + 2z + 2\lceil \log_2 L \rceil + k\lceil \log_2 L \rceil}{2xk}. \tag{9}$$

When the number of intersecting pairs is large, the cell and segment IDs appear in the intersection list occasionally. The asymptotical compression ratio is

$$\frac{\lceil \log_2 L \rceil}{2x}. \tag{10}$$

5.3. Pipelined Execution

After the GPU outputs the intersecting box pairs to the encoded intersection list, the encoded intersection list must be transmitted from the device to the host, and the CPU is then responsible for reporting (including decoding) the encoded information. To reduce the communication and decoding costs, we divide the computation into stages and overlap the execution of the following three tasks.

(1) The GPU executes the checking kernel and encodes/compresses the intersecting box pairs.
(2) The encoded intersection list is transmitted from the device to the host.
(3) The CPU decodes the encoded intersection list and reports the intersecting box pairs.

To relieve the data dependency of the three tasks, the double-buffering technique is employed such that the three tasks can simultaneously access data in different buffers. Figure 6 illustrates the double-buffering idea. First, we create two device buffers for

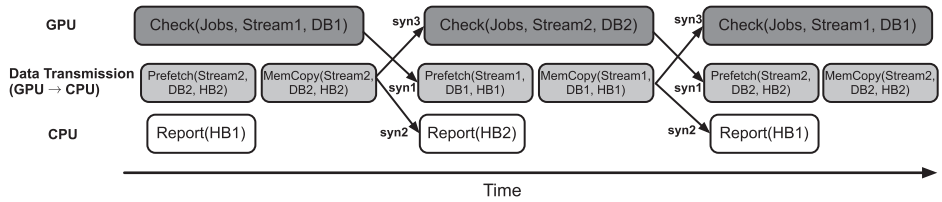

Fig. 7. Pipelined execution of the checking kernel, data transmission and reporting (and decoding) tasks.

Table II. Specifications of the Experimental Platform

Host	
Processor	Intel i7 processor 2.67 GHz
Memory	6 GB DRAM
Operating System	openSUSE 11.1 with kernel version 2.6.27
Compiler	GCC 4.3 with optimizations –O3 and –DNDEBUG
Device	
GPU Device	NVIDIA Tesla C2070
Global Memory	6 GB DRAM
Shared Memory	48 KB
Compiler	CUDA 4.0

the GPU and two host buffers for the CPU. When the GPU outputs data into the first device buffer, the data in the second device buffer is transmitted to the second host buffer. Meanwhile, the CPU can access the compressed data in the first host buffer. In the next iteration, the GPU and the CPU can work on the data in the second device buffer and the second host buffer respectively, and the data in the first device buffer are transmitted to the first host buffer.

The CUDA Stream technology [CUDA 2011] is required to enable executing multiple streams independently. A stream means an execution of the three tasks discussed above. Using the double-buffering and CUDA Stream, the three tasks become a 3-stage pipelined execution. Figure 7 shows the 3-stage pipelined execution using two streams (Stream1 and Stream2), two device buffers (DB1 and DB2) and two host buffers (HB1 and HB2).

In Figure 7, the pipelined execution effectiveness requires two mechanisms. First, the exact number of intersecting box pairs is available only when finishing the checking kernel. Data are prefetched to hide the time required to transmit data from the device to the host in the time to perform the checking kernel. The prefetching size equals the number of intersecting box pairs found in the previous kernel execution. Second, three synchronizations (as in Figure 7) are required to guarantee the execution correctness of the three tasks. The first synchronization waits for the data (in DB1) produced by the checking kernel before transmitting the data. The second synchronization waits for the data to be copied from the device (DB2) to the host (HB2) before reporting the encoded intersecting box pairs. The last synchronization waits for the data to be copied from the device (DB2) to the host (HB2) before the next checking kernel execution.

6. PERFORMANCE EVALUATION

We evaluate the performance of the proposed algorithm on a workstation with one GPU device. Table II lists the details of our experimental platform. We measure the time to perform box intersection checking for N boxes in 3-D space, including the data transmission.

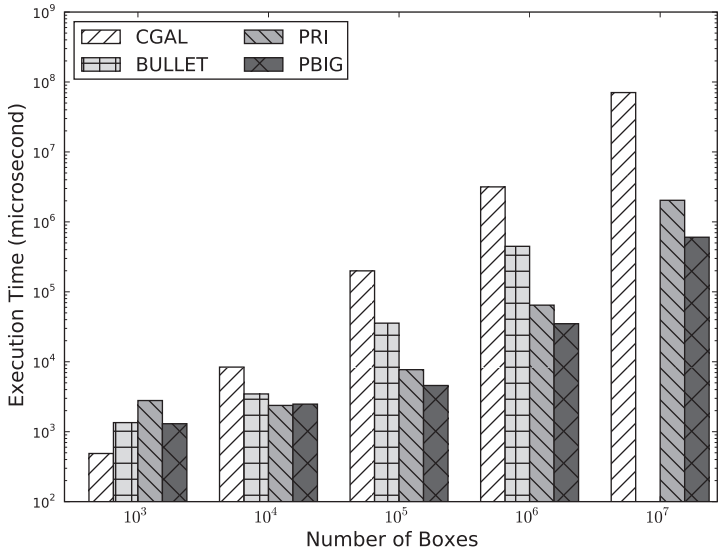

Fig. 8. Performance of the four algorithms with various numbers of boxes.

We compare the performance of our algorithm with those of one sequential and two parallel (GPU-based) algorithms. The sequential algorithm (called CGAL) is a fast algorithm in practice [Kettner et al. 2011]. The first parallel algorithm (called PRI) is an extension of the algorithm [Lo et al. 2011]; the second one (called BULLET) is the btCudaBroadphase module in the Bullet Physics Engine library [BULLET 2011]. The thread block size for the GPU-based implementations is 256; the cell sizes of the three algorithms, BULLET, PRI and PBIG, are the same, which are derived from Eq. (6). We use these algorithms for reference.

We present four sets of experimental results, each of which addresses one of the following performance factors:

— Number of boxes,
— Density of boxes,
— Distribution of boxes, and
— Optimization techniques.

6.1. Number of Boxes

We evaluate the performance of the four algorithms for various numbers of boxes. Boxes are randomly distributed in a 3-D space of size $10^4 \times 10^4 \times 10^4$. Specifically, the coordinate of the center point of a box in each dimension is a uniform random variable within the interval [0,10000], and the length of a box in each dimension is a uniform random variable within the interval [1,100]. The numbers of boxes tested are 10^3, 10^4, 10^5, 10^6 and 10^7.

Figure 8 shows the execution times of four implementations with various numbers of boxes. When $N = 10^3$, the performance of PBIG is slower than that of CGAL and BULLET because PBIG must perform additional tasks (e.g., CPU-GPU data communication and thread block scheduling), which results in a computational overhead for PBIG. Our profiling result shows that those tasks account for more than 50% of the total execution time of PBIG in this case. However, as N grows to 10^7, PBIG outperforms CGAL and PRI by 116× and 3.3× performance, respectively. BULLET cannot handle

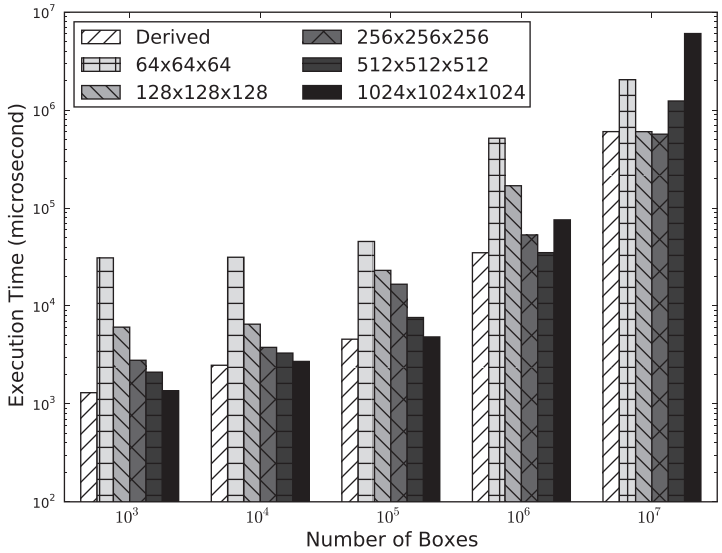

Fig. 9. Performance of PBIG using various cell sizes with various numbers of boxes.

the largest problem size, that is, 10^7 boxes in this experiment, because it allocates a fixed memory space for each box to record the output per box and a fixed capacity for each cell to record the boxes in a cell. For a problem of size 10^7, BULLET requires more memory space than the GPU Tesla C2070 has available.

Figure 9 shows the performance differences for various cell sizes. The cell sizes tested include the one derived from Equation (6), $64\times64\times64$, $128\times128\times128$, $256\times256\times256$, $512\times512\times512$ and $1024\times1024\times1024$. Figure 9 demonstrates that cell size affects performance significantly. When $N = 10^7$, the performance for cell size $64\times64\times64$ or $1024\times1024\times1024$ is worsen than that for cell size $256\times256\times256$. PBIG uses Eq. (6) to determine the cell size, the performance of which, as in Figure 9, is close to optimal in all cases.

6.2. Density of Boxes

In this paper, we use the Degree of Coverage (DoC) to adjust the density of boxes. The DoC is defined as

$$\text{DoC} = \frac{\text{The sum of volumes of all boxes}}{\text{The volume of the domain space}}. \tag{11}$$

The DoC value means the average number of boxes per unit area.

The DoC settings include 10^{-3}, 10^{-2}, 10^{-1}, 1 and 10. Figure 10 shows the execution times of the four algorithms for density of boxes. PBIG can perform intersection checking for 10^7 boxes with various DoC settings within one second, which outperforms the other three algorithms in all cases. For the greatest density of boxes, the number of intersecting pairs reported approximates 5×10^7, and the uncompressed and compressed output sizes are 377.76 MB and 44.73 MB, respectively.

Table III lists the PBIG statistics for various DoC settings. As the density of boxes grows, the number of intersecting pairs also increases, as in the second column of Table III. However, the number of intersection checks (reported in the third column of Table III) does not grow with the number of intersecting pairs. The number of

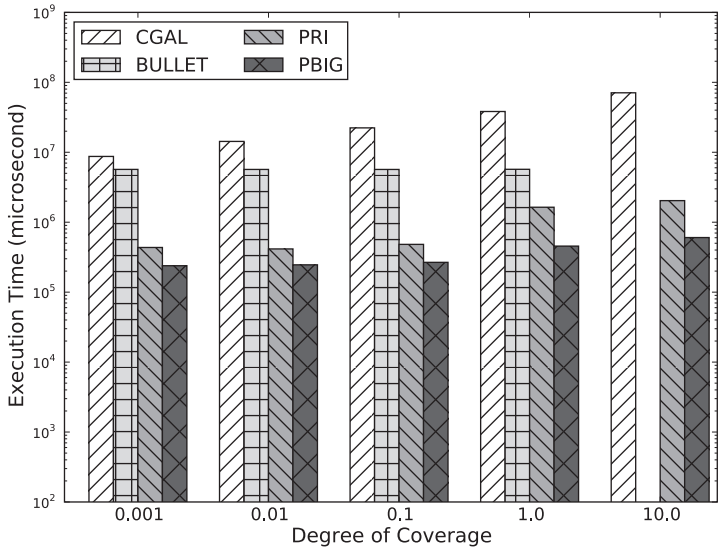

Fig. 10. Performance of the four algorithms with various DoC settings.

Table III. Statistics of the PBIG in Various Values of DoC for $N = 10^7$

DoC	Total Number of Intersecting Pairs	Total Number of Intersection Checks	Input Size (MB)	Output Size (MB)[a]
0.001	6,234	879×10^6	280	0.024
0.01	50,064	930×10^6	280	0.183
0.1	530,689	1062×10^6	280	1.334
1	5,165,837	286×10^6	280	9.369
10	49,514,252	744×10^6	280	44.734

[a]This is the compressed size.

intersection checks becomes large for DoC = 0.001, 0.01, and 0.1, whereas the number drops at DoC = 1. This phenomenon results from the cell size adjustment. For various DoC settings, the cell size is adjusted to reduce unnecessary checks and to achieve better hardware efficiency. For example, PBIG uses a larger cell size (i.e., $256\times256\times256$) to make cells contain at least 32 boxes when DoC = 0.1, but it uses a smaller cell size (i.e., $128\times128\times128$) to reduce unnecessary intersection checks when DoC = 1 or DoC = 10.

6.3. Distribution of Boxes

We further investigate the performance changes in a nonuniform distribution. The coordinates of the center points of boxes follow a Gaussian distribution, whose mean is (5000,5000,5000) with a standard deviation of 1000. Boxes are distributed in a 3-D space of size $10^4\times10^4\times10^4$. The numbers of boxes tested are 10^3, 10^4, 10^5, 10^6, and 10^7. For 10^7 boxes, this box distribution generates approximately 112×10^7 intersecting pairs. Other parameter settings are the same as in Section 6.1. Figure 11 shows the execution times of the four algorithms. Overall, all the execution times for the four algorithms in the nonuniform distribution are greater than those in the uniform distribution in Section 6.1.

Although PBIG is the fastest algorithm in most cases, its performance increase comparing the sequential algorithm CGAL for the non-uniformly distributed data is not

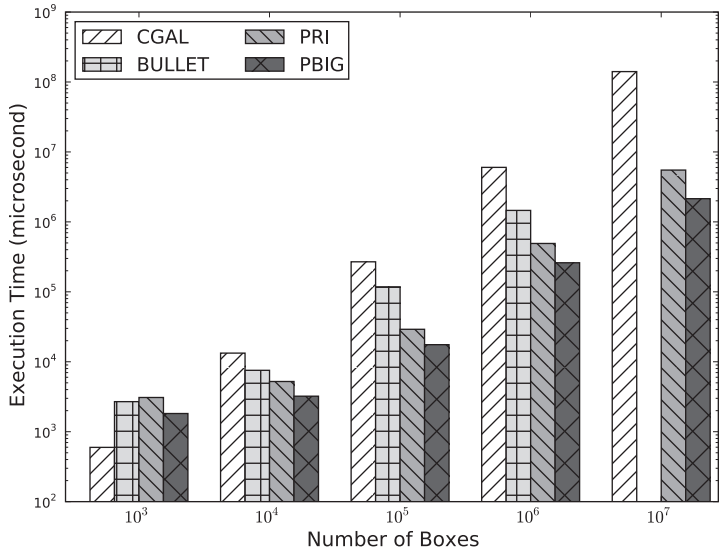

Fig. 11. Performance of the four algorithms with various numbers of boxes in a non-uniform distribution of boxes.

Table IV. Speedups of the PBIG Algorithm with Various Optimizations

Combinations of Optimizations	**Speedup**
Data Compression	1.86
Pipelined Execution	1.98
Load-Balancing	1.01
Data Compression and Pipelined Execution	2.63
Data Compression and Load-Balancing	2.88
Pipelined Execution and Load-Balancing	1.92
Compression, Pipelined Execution and Load-Balancing	4.81

comparable to that for the uniformly distributed data. For instance, when $N = 10^7$, the performance increase is 116× for uniformly distributed data, but only 65.66× for the non-uniformly distributed data. The major reason is the rapid increases of intersecting pairs in the non-uniformly distributed data. PBIG requires more memory operations and copies for data output, whereas CGAL does not incur the transmission overhead.

The PBIG algorithm still achieves a substantial performance increase in the nonuniformly distributed data. The data compression and pipelined execution schemes can help the box intersection checking performance in the simulations with a nonuniform distribution of entities.

6.4. Optimization Techniques

We demonstrate the effectiveness of three performance optimization techniques on our algorithm: load-balancing, data compression and pipelined execution. The number of boxes evaluated is 10^7. Other settings are the same as in Section 6.3. The baseline configuration is the PBIG algorithm without any optimization. Table IV lists the PBIG performance speedups using different optimization techniques.

We further analyze the performance speedups by breaking the PBIG execution into tasks, listed in Table V. Table VI shows the execution time breakdown. In Table VI, when the pipelined execution is activated, the CHECK, DataOut and REPORT tasks

Table V. List of Tasks in the PBIG Algorithm According to the Execution Order

Tasks	Description
DataIn	Copy boxes from the host to the device.
PLAN	Determine a cell size.
MAP	Map boxes into cells.
SCHED	Do thread block scheduling.
CHECK	Perform box intersection checks and output the intersecting pairs to global memory.
DataOut	Copy the encoded intersection list from the device to the host.
REPORT	Report (including the decoding) the intersecting pairs.

Table VI. Execution Time Breakdown of the PBIG Algorithm

PBIG	Total	DataIn	PLAN	MAP	SCHED	CHECK	DataOut	REPORT
Baseline	10264940	50423	30	137909	2644	4970496	1740208	3312812
Baseline+C[a]	5512054	52686	35	137713	2709	3806897	473857	995124
Baseline+P[b]	5181014	48470	28	137649	2753	4983314		
Baseline+C+P	3891116	49243	30	137845	2684	3697194		
Baseline+B[c]	10167360	49926	31	137854	2701	5039146	1696377	3191348
Baseline+C+B	3562590	54857	39	137996	3193	2097496	483593	740671
Baseline+P+B	5337236	48115	41	137856	2837	5139588		
Baseline+C+P+B	2132998	47760	35	137840	2537	1941184		

[a]C: Data compression is activated.
[b]P: Pipelined execution is activated.
[c]B: Load-balancing is activated.

overlap. Table VI present the total execution time of the three tasks. The three tasks occupy more than 90% of the execution time. We use the three optimizations to handle the three PBIG tasks.

Among the three optimizations, data compression and pipelined execution can increase performance by factors of 1.86× and 1.98×, respectively. The load-balancing-only optimization contributes much less performance than the other two optimizations. Applying all of the optimizations, however, multiplies the effect on performance by a factor of 4.81×. As in Table VI, the PBIG algorithm with the three optimizations can reduce the execution time of the three tasks to 1.941 seconds, compared with that for only the data compression and pipelined execution, that is, 3.697 seconds.

The effect of the load-balancing optimization is constrained by the performance of outputting intersecting pairs to the intersection list. This test case generates many intersecting pairs (i.e., 112×10^7 pairs), which dominates the overall execution time. The algorithm without compression produces approximately 8.968 GB of data, whereas the algorithm with data compression reduces the output size to approximately 1.824 GB. The load-balancing method is effective in reducing the execution time when the data compression is first applied to handle the large output issue.

7. CONCLUSIONS AND FUTURE WORK

For large-scale, real-time computer simulations, pairwise intersection checking is essential to prune unwanted interactions among millions of objects. This paper investigated pairwise box intersection checking on GPUs for large-scale datasets. Two major challenges for parallelizing box intersection checking are data partitioning and large output processing. The proposed algorithm utilizes several optimization techniques for performance enhancement. The PBIG algorithm was compared with one well-implemented sequential algorithm and two GPU-based algorithms: up to 110×, 10× and 3× performance increases, respectively, were observed.

The optimization techniques and analyses developed to improve the performance can be applied to a broad range of applications implemented on GPUs. First, problems

with random access behaviors can utilize the three-phase strategy to reduce the random access cost and enhance scalability. Second, problems with massive data output can utilize the data compression and pipelined execution to reduce the I/O time. Third, applications with symmetric properties can apply the load-balancing method to equalize the workload among computing threads.

There are several directions for future work. First, parallelizing the more general computations, such as arbitrary-oriented box intersection checking or triangulation intersection checking in 3-D environments, will be our future studies. Second, the possibility of efficient implementations exploiting the temporal consistency property of applications requires further study. Third, utilizing different computational architectures, including multiple GPUs or multicore CPUs with GPUs, to enhance scalability and performance also holds great interest. Because of the importance of box intersection checking in various applications, further investigations addressed to generalizing the techniques used are also required to develop practical packages.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their valuable comments and suggestions.

REFERENCES

Avril, Q., Gouranton, V., and Arnaldi, B. 2010. A broad phase collision detection algorithm adapted to multi-cores architectures. In *Proceedings of the IEEE Virtual Reality International Conference*. IEEE, Los Alamitos, CA, 95–100.

Batista, V. H. F., Millman, D. L., Pion, S., and Singler, J. 2010. Parallel geometric algorithms for multi-core computers. *Computat. Geometry* 43, 8, 663–677.

Bentley, J. L. and Ottmann, T. A. 1979. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput. C-28*, 9, 643–647.

Bentley, J. L. and Wood, D. 1980. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Trans. Comput. C-29*, 7, 571–577.

Boukerche, A., McGraw, N., Dzermajko, C., and Lu, K. 2005. Grid-filtered region-based data distribution management in large-scale distributed simulation systems. In *Proceedings of the 38th Annual Simulation Symposium*. IEEE, Los Alamitos, CA, 259–266.

BULLET. 2011. Bullet Physics Library. (2011). http://bulletphysics.org/wordpress/.

CGAL. 2011. Computational geometry algorithms library. (2011). http://www.cgal.org.

Chow, A. L. 1980. Parallel algorithms for geometric problems. Ph.D. dissertation, University of Illinois at Urbana-Champaign.

Cohen, J. D., Lin, M. C., Manocha, D., and Ponamgi, M. 1995. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the Symposium on Interactive 3D Graphics*. ACM, 189–196.

CUDA. 2011. CUDA Programming Guide, 4.0, NVIDIA. (2011). http://developer.nvidia.com/cuda-downloads.

Edelsbrunner, H. 1983. A new approach to rectangle intersections part I. *Int. J. Comput. Math. 13*, 3, 209–219.

EVE Online. 2003. EVE online game. (2003). http://www.eveonline.com/.

Gupta, P. and Guha, R. K. 2007. A comparative study of data distribution management algorithms. *J. Defen. Model. Simul. Appl. Method. Tech. 4*, 2, 127–146.

Guting, R. H. and Wood, D. 1984. Finding rectangle intersections by divide-and-conquer. *IEEE Trans. Comput. C-33*, 7, 671–675.

Harada, T. 2007. Real-time rigid body simulation on GPUs. In *GPU Gems 3*, Addison-Wesley, Boston MA, 611–632.

Harris, M., Shubhabrata, S., and Owens, J. D. 2007. Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*, Addison-Wesley, Boston MA, 851–876.

Huang, X., Rodrigues, C. I., Jones, S., Buck, I., and Hwu, W. M. 2010. XMalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *Proceedings of the IEEE 10th International Conference on Computer and Information Technology (CIT)*. IEEE, Los Alamitos, CA, 1134–1139.

Kettner, L., Meyer, A., and Zomorodian, A. 2011. Intersecting sequences of dD iso-oriented boxes. http://www.cgal.org.

Lefohn, A., Kniss, J., and Owens, J. 2005. Implementing efficient parallel data structures on GPUs. In *GPU Gems 2*, Addison-Wesley, Boston MA, 521–545.

Le Grand, S. 2007. Broad-phase collision detection with CUDA. In *GPU Gems 3*, Addison-Wesley, Boston MA, 697–721.

Liu, E. S. and Theodoropoulos, G. K. 2009. An approach for parallel interest matching in distributed virtual environments. In *Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*. IEEE, Los Alamitos, CA, 57–65.

Liu, E. S. and Theodoropoulos, G. K. 2011. A parallel interest matching algorithm for distributed-memory systems. In *Proceedings of the IEEE/ACM 15th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, Los Alamitos, CA, 36–43.

Liu, F., Harada, T., Lee, Y., and Kim, Y. J. 2010. Real-time collision culling of a million bodies on graphics processing units. In *Proceedings of ACM SIGGRAPH Asia 2010*. ACM, New York, 154:1–154:8.

Lo, S. H., Lee, C. R., Chung, I. H., and Chung, Y. C. 2011. A parallel rectangle intersection algorithm on GPU+CPU. In *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, Los Alamitos, CA, 43–52.

Merrill, D. and Grimshaw, A. 2011. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Paral. Proc. Lett. 21*, 02, 245–272. DOI:http://dx.doi.org/10.1142/S0129626411000187.

Ming, C. L. and Gottschalk, S. 1998. Collision detection between geometric models: A survey. In *IMA Conference on Mathematics of Surfaces*, Vol. 1. Springer, New York, NY 37–56.

Overmars, M. H. 1992. Point location in fat subdivisions. *Inform. Process. Lett. 44*, 5, 261–265.

Pan, K., Turner, S. J., Cai, W., and Li, Z. 2011. A dynamic sort-based DDM matching algorithm for HLA applications. *ACM Trans. Model. Comput. Simul. 21*, 3, Article 17. DOI:http://dx.doi.org/10.1145/1921598.1921601.

PBIG. 2013. Implementation of pairwise box intersection checking on graphics processing units. (2013). http://code.google.com/p/pbig/.

Petty, M. D. and Mukherjee, A. 1997. Experimental comparison of d-rectangle interection algorithms applied to HLA data distribution. In *Proceedings of the Distributed Simulation Symposium*. 13–26.

Petty, M. D. and Morse, K. L. 2004. The computational complexity of the high level architecture data distribution management matching and connecting processes. *Simul. Model. Pract. Theory 12*, 3–4, 217–237. http://www.sciencedirect.com/science/article/pii/S1569190X04000395.

Raczy, C., Tan, G., and Yu, J. 2005. A sort-based DDM matching algorithm for HLA. *ACM Trans. Model. Comput. Simulat. 15*, 1, 14–38.

Santoro, A. and Fujimoto, R. M. 2008. Offloading data distribution management to network processors in HLA-based distributed simulations. *IEEE Trans. Paral. Distrib. Syst. 19*, 3, 289–298.

Six, H. W. and Wood, D. 1980. The rectangle intersection problem revisited. *BIT Numer. Math. 20*, 4, 426–433.

Tan, G., Yusong, Z., and Ayani, R. 2000. A hybrid approach to data distribution management. In *Proceedings of the 4th IEEE International Workshop on Distributed Simulation and Real-Time Applications*. IEEE, Los Alamitos, CA, 55–61.

Vaishnavi, V. K. and Wood, D. 1982. Rectilinear line segment intersection, layered segment trees, and dynamization. *J. Algor. 3*, 2, 160–176.

Van Hook, D. J., Rak, S. J., and Calvin, J. O. 1996. Approaches to RTI implementation of HLA data distribution management services. In *Proceedings of the 15th Distributed Iinteractive Simulation Workshop*. IEEE, Los Alamitos, CA, 96–14–084.

Zomorodian, A. and Edelsbrunner, H. 2002. Fast software for box intersections. *Int. J. Computat. Geom. Appl. 12*, 1–2, 143–172.

Received January 2012; revised July 2012, January 2013; accepted February 2013