Parallel Spatial Query Processing on GPUs Using R-Trees

Simin You
Dept. of Computer Science
CUNY Graduate Center
New York, NY, 10016
syou@gc.cuny.edu

Jianting Zhang
Dept. of Computer Science
City College of New York
New York, NY, 10031
jzhang@cs.ccny.cuny.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK 73071
ggruenwald@ou.edu

ABSTRACT
R-Trees are popular spatial indexing techniques that have been widely adopted in many geospatial applications. As commodity GPUs (Graphics Processing Units) are increasingly becoming available on personal workstations and cluster computers, there are considerable research interests in applying the massive data parallel GPGPU (General Purpose computing on GPUs) technologies to index and query large-scale geospatial data on GPUs using R-Trees. In this study, we aim at evaluating the potentials of accelerating both R-Tree bulk loading and spatial window query processing on GPUs using R-Trees. In addition to designing an efficient data layout schema for R-Trees on GPUs, we have implemented several parallel spatial window query processing techniques on GPUs using both dynamically generated R-Trees constructed on CPUs and bulk loaded R-Trees constructed on GPUs. Extensive experiments using both synthetic and real-world datasets have shown that our GPU based parallel query processing techniques using R-Trees can achieve about 10X speedups on average over 8-core CPU parallel implementations by effectively utilizing large numbers of processors and high memory bandwidth on GPUs.

Categories and Subject Descriptors
H.2.8 [**Database Management**]: Database Application – *spatial databases and GIS*; I.3.1 [**Computer Graphics**]: Hardware Architecture – *Graphics processors*

General Terms
Algorithms, Performance

Keywords
Spatial Indexing, R-Tree, GPU

1. INTRODUCTION
R-Trees [4, 6, 15] are well known spatial indexing techniques and have been widely adopted in many applications for indexing 2-D or higher dimensional spatial data. Many techniques have been proposed over the past three decades to improve efficiency and performance of R-Tree construction and R-Tree based spatial query processing. As R-Tree construction typically incur super-linear complexity, it is desirable to parallelize R-Tree constructions on parallel platforms, especially when processing large-scale geospatial data. While processing a single spatial query on an R-Tree is typically sub-linear, there are many applications that involve batched queries on R-Trees where parallelization is obviously beneficial. Quite some parallel R-Tree construction and query processing algorithms have been proposed for different parallel architectures [9, 10, 12, 14, 16, 17]. Early research mostly focused on shared-nothing computer clusters that are made of identical computing nodes equipped with uniprocessors [9, 14, 16, 17].

Modern GPU architectures closely resemble supercomputers as both implement the Primary Parallel Random Access Machine (PRAM[1]) characteristic of utilizing a very large number of threads with uniform memory latency. Compared to modern CPUs, GPU devices usually have larger numbers of processing cores, higher memory bandwidths with more affordable prices. For example, the Nvidia GTX Titan GPUs[2] have nearly 3,000 processing cores, 300 GB/s bandwidth, 6 GB memory and can be purchased from the market around $1,000. As many commodity desktop computers have already been equipped with GPU devices that are capable of general computing, it is desirable to use GPUs to accelerate geospatial computing in general and R-Tree based spatial data management in particular. Furthermore, GPUs have been extensively used to accelerate many computing intensive applications, such as nearest neighbor queries in databases and ray-tracing in computer graphics. Efficient indexing structures such as R-Trees are promising in speeding up such computing on GPUs that are practically useful. As it is still quite expensive to transfer data between CPUs and GPUs through PCI-E buses (limited to 8-32 GB/s on PCI-E devices with 16 lanes), being able to directly construct and query R-Trees on GPUs to avoid or reduce data transfer overheads is beneficial, which motivates us to implement both R-Tree construction and query processing on GPUs.

In this study, we aim at exploring different design strategies on R-Tree construction and query processing on GPUs. We have evaluated the performance of several designs and implementations using both synthetic and real-world datasets. First, in addition to the re-launching overflow handling strategy (details in Section 3.2.2) for batched spatial queries implemented in [12], we have also designed and implemented several additional overflow handling techniques. Second, while a GPU based R-Tree bulk loading technique called low-*x* packed R-Tree has been implemented in [12], the bulk loaded R-Tree was not used for query processing in the study. This left the performance of spatial query processing on bulk loaded R-Trees, which typically are faster in construction but have lower qualities, largely unknown. In this study, we have considered both low-*x* packed R-Tree based bulk loading [12] and Sort-Tile-Recursive (STR) based R-Tree bulk loading [11]. While some of our designs and implementations have demonstrated significant speedups over CPU implementations on R-Tree based query processing (as reported in Section 4), we are more interested in understanding the relative advantages and disadvantages of GPU-based R-Tree construction and query processing over CPU-based ones as well as the impacts of R-Tree quality to provide insights and guidelines for more systematic and more efficient implementations. For testing purposes, we have run our implementations on a spatial query benchmark including synthetic and realistic datasets. We report our experiment results and provide discussions on our findings. Our technical contributions in this paper can be summarized as follows:

1) We have provided an improved R-Tree node layout on GPUs which has lower memory footprint.
2) We have implemented alternative R-Tree bulk loading and spatial window query processing strategies on GPUs using R-Tree which are not covered by previous works [10, 12].
3) We have performed extensive experiments using different R-Trees, including bulk loaded ones and dynamic inserted ones, and evaluated different R-Tree based spatial window query processing designs on GPUs using both synthetic datasets and real datasets. To the best of our knowledge, this is the most comprehensive set of evaluations of R-Tree bulk loading and spatial query processing on GPUs.

The rest of this paper is organized as follows. Section 2 introduces background and related work. Section 3 provides our R-Tree designs and implementations on GPUs, including bulk loading and batched spatial query processing techniques. Section 4 presents experiments and results. Finally Section 5 is the conclusion and future work.

2. BACKGROUND AND RELATED WORK

R-Tree based indexing techniques have been extensively studied in spatial databases and quite a few variants have been proposed over the past three decades [4, 6, 15]. Although most existing R-Tree implementations are serial on a single CPU (uniprocessor), there are several previous studies on parallel R-Tree construction and query processing based on different parallel hardware architectures although those based on shared-nothing clusters clearly dominate [7, 9, 14, 16, 17]. In this study, we focus on R-Tree based spatial indexing using GPGPU computing technologies that have quite a different parallel computing model. We also argue that our work is complementary to cluster computing based distributed and parallel spatial query processing provided that computing nodes are equipped with GPU devices which is becoming increasingly popular in both institutional grid computing resources and commercial cloud computing resources.

The most related work to ours, which is reported in [12], has implemented a Breadth-First-Search (BFS) traversal based query processing algorithm using R-Trees on modern GPUs. In addition to the differences that have been discussed in the introduction section, our designs and implementations are also different from the following aspects. First, as discussed in details in Section 3.1.1, while both implementations use linear array structures to store R-Tree nodes in a BFS order and has a node field to indicate the array position of the first child node (in a way similar to the design of our GPU-based Binned Min-Max Quadtree BMMQ-Tree [18]), our node layout has a separate field to store the number of children of an R-Tree node. By using at most one extra byte (which can represent 256 children) in most cases, we are free of having to store non-exist child nodes which can save up to 50% of required memory for a whole R-Tree. Second, while the work in [12] proposed and evaluated only one overflow handling mechanism (see details in Section 3.2.2), we introduce two new overflow handling mechanisms as well as a pure parallel primitive based BFS query implementation which does not require overflow handling. We have compared query performance based on different parallel spatial query processing strategies. Third, while [12] only performed experiments on dynamic inserted R-Trees, we have evaluated query processing on bulk loaded R-Trees using different R-Tree bulk loading strategies.

A GPU-based R-Tree query processing algorithm termed Massively Parallel Three-phase Scanning (MPTS) is proposed in [10]. The key idea of MPTS is to minimize irregularly memory accesses during R-Tree traversals when processing spatial queries. During the traversals, left- and right- most nodes are identified and a parallel scan is performed from the left-most node to the right-most node to search for intersected MBRs. However, MPTS is mostly designed for optimizing single spatial window query processing instead of processing multiple queries in parallel. Rather than further improving query processing times for single queries, which are typically already fast enough even for large datasets with reasonable degrees of selectivity (in the order of a fraction of a second), we focus on parallel query processing for a large number of independent queries by fully utilizing massively data parallel computing power on GPUs, which we believe is more cost-effective for practical applications.

While the original R-Tree construction algorithms use dynamic insertions, several bulk loading approaches have been proposed [3, 5, 8, 11, 14]. Bulk loading approaches usually adopt a packing technique to construct R-Trees that maximizes space utilization and reduces the height of the resulting tree as much as possible. Packed R-Tree guarantees better space utilization and query responses based on packed R-Trees have been reported to be comparable with R-Trees built from dynamic insertions [5, 8]. Bulk loading methods can be classified into two categories, i.e., top-down and bottom-up. Alborzi and Samet [3] discussed the difference between top-down and bottom-up methods. They argued that the top-down method could potentially process queries faster but non-leaf nodes might be under-packed. On the other hand, R-Trees constructed by bottom-up methods may have fewer nodes than using top-down methods. Sort-Tile-Recursive (STR) is a simple yet efficient R-Tree packing method proposed in [11]. In the STR approach, R-Tree is constructed by recursively sorting and packing in a bottom-up manner. As both sorting and packing can be easily mapped to parallel primitives (i.e., *sort* and *reduce*, respectively), it is attractive to implement R-Tree bulk loading algorithms on GPUs directly. Based on the results discussed in [11], low-*x* packing is not competitive with STR based packing. The results have motivated us to implement STR R-Tree bulk loading on GPUs and compare it with the low-*x* R-Tree bulk loading that has been implemented in [12].

3. METHODOLOGY

3.1 Parallel R-Tree Bulk-loading

3.1.1 Node Layout of Linearized R-Tree

We use simple linear array based data structures to represent an R-Tree. Simple linear data structures can be easily streamed between CPU main memory and GPU device memory without serialization and are also cache friendly on both CPUs and GPUs. In our design, each non-leaf node is represented as a tuple {*MBR*, *pos*, *len*}, where *MBR* is the minimum bounding rectangle of the corresponding node, *pos* and *len* are the first child position and the number of children, respectively, as illustrated in Fig. 1. In contrast, previous work [12] stored entries for all children in non-leaf nodes, which will use more memory than our method. The tree nodes are serialized into an array based on the Breadth-First-Search (BFS) ordering. The decision to record only the first child node position instead of recording the positions of all child nodes in our approach is to reduce memory footprint. Since sibling nodes are stored sequentially, their positions can be easily calculated by adding the offsets back to the first child node position. In addition to memory efficiency, the feature is desirable on GPUs as it facilitates parallelization by using thread identifiers as the offsets. In this study, we have used R-Trees constructed from two approaches: bulk loading on GPUs and dynamic insertions on CPUs. The algorithm to fill the *pos* and *len* fields using bulk loading on GPUs are discussed in the next section. When an R-Tree is generated on CPUs, the two fields can be filled easily by sequentially looping through R-Tree nodes through pointer chasing.

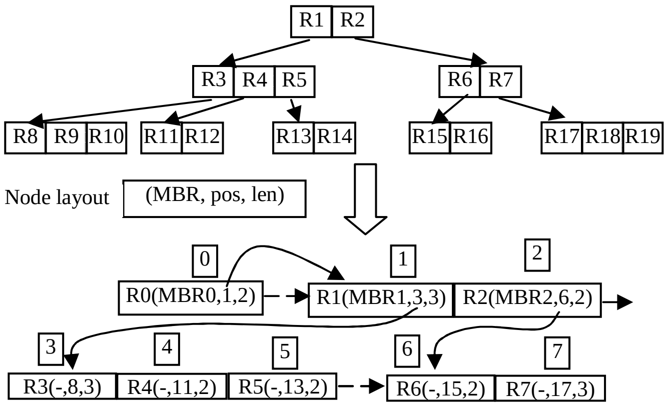

Fig. 1 Illustration of Linear R-Tree Node layout

3.1.2 Parallel R-Tree Bulk Loading on GPUs

In this study, we implement both low-*x* packing (used in [12]) and STR packing [11] for bulk loading R-Trees. Instead of using native GPU programming languages (such as Nvidia CUDA[3]) directly, our implementations are built on top of several parallel primitives provided by the Thrust library that comes with CUDA SDK. The decision has significantly reduced coding complexity and improved portability.

For the low-*x* packing approach, in the sorting stage, the original data (MBRs) is sorted by applying a linear ordering schema (low-*x* in this case) using a *sort_by_key* parallel primitive. The algorithm is shown in Fig. 2 where the R-Tree is constructed by packing MBRs bottom-up. Every *d* items are packed into one node at the upper level until the root is created. We first calculate the number of levels (num_{level}) and the number of nodes at each level for memory allocation and addressing during the packing iteration. We then construct the R-Tree level by level bottom-up using a *reduce_by_key* primitive. Step 1 sorts the original dataset using any 1-D ordering (low-*x* in this case). From steps 2 to 6, R-Tree is iteratively packed from lower levels. In step 4 and 6, same keys need to be generated every *d* items for parallel reduction purpose. This can be done by combining *transform_iterator* and *counting_iterator* iterators provided by the Thrust parallel library. The MBRs, first child positions and numbers of children are evaluated from the data items at the lower levels as follows. For the *d* items with a same key, the MBR for the parent node is the union of MBRs of the children nodes. For each R-Tree node, the first child position (*pos*) is computed as the minimum sequential index of lower level nodes (by using a *counting_iterator*) and the length (*len*) is calculated as the sum of 1s (by using a *const_iterator* initially set to 1) for each child node. While we have skipped the details of the auxiliary iterators (which are nonessential to understanding the implementation of the construction process) for the interests of space, we would like to note that *reduce_by_key* and min/sum based *scans* are well supported by parallel libraries (e.g. Thrust).

We also implement the Sort-Tile-Recursive (STR) R-Tree bulk loading algorithm on GPUs using parallel primitives as follows. First, MBRs are sorted along one direction, i.e., using *x* coordinates from lower left corners, which is implemented by using *sort_by_key*. Then the space is divided into slices according to the predefined fanout *d*, and each slice is sorted along the other direction, such as *y*-coordinates. Finally every *d* MBRs in a slice are packed as parent nodes which will be used as the input for the next iteration. This process is iteratively executed until the root of the tree is constructed. Fig. 3 outlines the STR R-Tree construction algorithm. Steps 2 to 4 check whether the number of MBRs is smaller than the fanout *d*. If this is the case, the MBRs will be packed as root and the iteration is terminated. Otherwise, the MBRs are first sorted using low-*x* coordinates (Step 6), and *N* MBRs are divided into $\sqrt{N/d}$ slices where each slice is sorted according to low *y*-coordinates (Step 7). After sorting on each slice, parent nodes are generated via packing every *d* MBRs (Step 8). Finally, N/d nodes are used as input for the next iteration (Step 9). The first sort can be easily implemented by using *sort_by_key* where *x*-coordinates are used as the key. To implement the second sort where each slice is sorted individually, we use an auxiliary array to identify items that belong to the same slice. This is achieved by assigning the same unique identifier for all items belong to the same slice, i.e., a sequence identifier is assigned for each slice and stored in the auxiliary array. With the help of the auxiliary array, Step 7 can be accomplished by invoking *sort_by_key* twice, where the first sort is on *y*-coordinates and the second sort uses the unique identifiers in the auxiliary array. Step 8 is the same as the packing phase introduced previously (steps 4 and 6 in Fig. 2). The difference between the two packing algorithms is that the low-*x* packing algorithm only sorts once while the STR packing algorithm requires multiple sorts at each level.

```
Input: fanout d; dataset D
Output: packed R-Tree
1. sort D using 1-D ordering (e.g. low-x)
2. for level ← $num_{level}$ decrease to 1
3.   if (level is last level)
4.     reduce_by_key from original data D
5.   else
6.     reduce_by_key from lower level
```

Fig. 2 Low-*x* R-Tree Bulk Loading on GPUs

```
Input: fanout d; dataset D
Output: packed R-Tree
1. while (true)
2.   if ($N \le d$)
3.     root ←pack N MBRs
4.     break;
5.   else
6.     sort_by_key  on x-coordinates
7.     sort_by_key  on y-coordinates for each slice
8.     reduce_by_key packed every d MBRs
9.     $N \leftarrow N/d$
```

Fig. 3 STR R-Tree Bulk Loading on GPUs

```
1. //DFS count kernel
2. __shared__ STACK_POOL[]
3. i = get_thread_index();
4. STACK[] = &STACK_POOL[i*STACK_SZ];
5. Push(STACK, {0, 0}); //push root to stack
6. Hit = 0
7. while (Size(STACK)>0)
8.   {index, visit} = Pop(STACK)
9.   if (R[index].len == visit)
10.     continue; //all children are visited
11.   next = R[index].pos + visit;
12.   visit++;
13.   Push(STACK, {index, visit});
14.   if (Intersect(MBR[i], R[next].MBR))
15.     if (Leaf(R[next]))
16.       Hit++;
17.     else
18.       Push(STACK, {next, 0});
19. Pos[i+block_offset] = Hit;
```

Fig. 4 Implementation of the Counting Phase of the DFS based Query Processing on GPUs

3.2 GPU based R-Tree batched query

Instead of accelerating a single query, our goal is to support efficient batched query processing on the GPU in parallel. To leverage massively parallel processing power of GPUs, we need to balance workload among all parallel processing units while minimizing expensive global memory operations on GPUs. In this section we will present different approaches of utilizing GPUs to speed up batched spatial window query processing.

3.2.1 Depth-First-Search based Method

In this approach, each thread processes a query in a Depth-First-Search (DFS) manner and thus a stack is required to track visited nodes for each query. A naïve implementation can be maintaining the stack on GPU global memory and each thread does its own work. Note that the stack is frequently read and write but the global memory accesses are not coalesced in the naive implementation. To improve the performance, we utilize per-block shared memory for the stack structure instead. While it is well known that GPU shared memory is usually limited for many applications, we show that this is not a disabling factor for DFS based R-Tree query processing although it does affect the scalability of the approach. For an R-Tree with a depth of h, which is typically in the order of a few tens, a stack of size larger than h is sufficient for DFS-based queries. As we assign a thread to a query in a batch, the total required shared memory M is in the order of $h*t$, where t is the number of threads in a computing block (or the number of queries in a batch). Even for t as large as 256, M is still significantly less than the typical 16 KB or 48 KB limit. To keep track of visited information in DFS traversals, the data items in the stack are organized using two fields, *index* and *visit*. The *index* field is the position to the R-Tree node array that provides access to the corresponding R-Tree node. The *visit* field is used for recording the number of visited children under the current R-Tree node.

The DFS-based query is divided into two phases which follows the "count and write" pattern. Two kernels are launched during the query process. The first one, termed as "count", is to count the numbers of hits (leaf R-Tree nodes whose MBRs intersect with query windows) for all individual queries in order to output query results in parallel. Fig. 3 shows the implementation of the "count" phase. In addition to the stack pool structure in shared memory discussed before, an array *Pos* is allocated for storing counting results. After the counting phase completes, a parallel prefix scan is performed on the *Pos* array to compute the output positions for the second phase which actually outputs the query results in parallel based on the computed positions. Since the length of the output array can be derived by the prefix scan results before memory allocation, no memory space is wasted in the DFS query approach which is a desirable feature. The implementation of the "write" phase is almost identical to the "count" phase with some modifications in Steps 16 and 19 in Fig. 4; i.e., instead of simply counting the number of hits, the query results are output to the allocated array.

Despite that the DFS-based query processing technique has a low shared memory footprint on GPUs, the nearly duplicated count/write phases may hurt the performance of the DFS based query processing implementation. The counting phase is essentially the overhead for thread coordination in parallel computing. Another disadvantage of the DFS-based query processing technique is that, workloads among the threads in a computing block may be imbalanced as threads work independently. As shown in Section 4, it is not surprising that the DFS-based technique has poor performance when compared to alternatives to be presented next.

3.2.2 Breadth-First-Search based Method

As an improvement to the DFS-based spatial window query processing technique, the BFS-based technique is developed to balance the workload within a GPU computing block. A queue is maintained for all the threads inside a computing block to process all the batched queries assigned to a computing block. Each element of the queue is represented in the form of {*index*, *qid*} where *index* is the position to the R-Tree node array so that the corresponding R-Tree node can be retrieved (the same as in DFS based one). The *qid* field represents the identifier of the query that is being processed. In the BFS-based technique, R-Tree nodes whose MBRs intersect with any of the query windows are expanded in parallel and stored in the queue level by level.

Unlike the DFS-based query processing where the sizes of outputs are computed in a separate phase for each query, in the BFS-based query processing, a computing block has its own global memory space for writing out query results which are pre-allocated. In our implementation, the size is set to the same as the queue capacity in shared memory so that computing blocks that successfully complete their BFS-based queries can easily copy the queue, which represent the query results, to global memory by synchronizing all the threads assigned to the computing block. Since the memory accesses are coalesced, the cost of copying the query results to global memory is minimized. However, as the queries may vary in window sizes and large query windows may intersect with a large number of R-Tree nodes, during level-wise query expansions, there are chances that the pre-allocated memory space to a computing block may overflow. As such, overflow handling must be used to correctly report query results in overflowed blocks. The essential idea of overflow handling is to complete query processing even when overflow happens. We will introduce three different strategies in the following subsections.

3.2.2.1 Kernel re-launching

In the previous work presented in [12], the authors suggested using additional resolving kernels to complete queries in overflow blocks. While an overflow happens during the process, a per-block overflow tag is set and all elements of the queue at the level before the overflow happens are copied to GPU global memory. The resolving kernel is then launched to continue BFS-based queries on the saved queues in GPU global memory. In order to complete the queries, multiple blocks are assigned to the batched queries that are originally dedicated to one block. During the execution of resolving kernel, overflow may happen again. In such a case, new kernels are repetitively launched until all queries have been successfully processed. This process of addressing overflow cases is termed as kernel re-launching. A major drawback of this approach is that additional kernel invocation overhead is imposed when iteratively launching the kernel. The imposed overhead will hurt the overall performance especially when overflow cases are frequent. Furthermore, [12] used shared memory for the per-block queue and the size of shared memory queue is thus another limiting factor.

3.2.2.2 DFS based overflow handling

To minimize the number of kernel launches, we propose a new approach to addressing the overflow issue by using DFS batched queries on the saved queues for BFS batched queries. As such, we term the approach as BFS-DFS Hybrid or simply Hybrid for short. In this technique, each data element is assigned to a thread and the DFS-based query processing technique is used to continue searching on the saved queue. Because the stack in DFS will not overflow as we discussed previously, only two kernels launches are required. An example is given in Fig. 5 where two queries are showing as red (query 1) and green (query 2), respectively. The queue capacity in this example is 3. While an overflow happens in the BFS stage, elements (B_1, C_1, D_2) in the queue are saved to GPU global memory. The DFS stage subsequently takes the saved queue as its input. For each element in the queue, a DFS query is performed to complete the query. However, in cases when DFS-based queries are highly unbalanced, the overall performance will be dominated by the overflow handling module. Thus, this approach is only useful when overflow cases are infrequent.

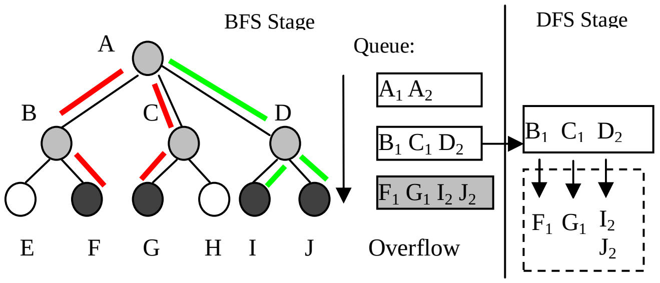

Fig. 5 Running Example for BFS based Query Processing with Overflows – the Hybrid Approach

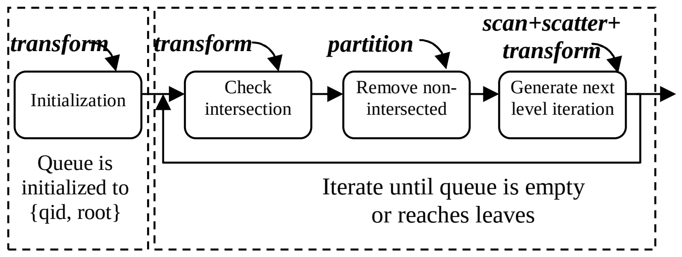
transform
transform
partition
scan+scatter+ transform
Initialization
Check intersection
Remove non-intersected
Generate next level iteration
Queue is initialized to {qid, root}
Iterate until queue is empty or reaches leaves

Fig. 6 Parallel primitive based BFS batch query

3.2.2.3 Dynamic Memory Allocation

Both methods for handling overflow cases that are introduced previously require invoking new kernels. In other words, when an overflow happens, the query kernel is terminated with saved overflowed queue written into GPU global memory, and, overflow handling kernels are launched to continue process overflow blocks. Here we propose a new method that is capable of performing query within a single kernel execution. Instead of using additional resolving kernels, this new method handles overflow cases without terminating a query kernel. In order to achieve this goal, additional global memory space is allocated to hold overflow elements. These elements will be stored in GPU global memory when an overflow happens and dynamically loaded into GPU shared memory when being processed. Note that the query processing kernel is not interrupted and no additional resolving kernels are needed. To efficiently manage the memory during kernel execution, a global memory pool is pre-allocated before kernel launches. The pool is organized as a group of memory chunks. The size of the chunks is set to the same as the queue capacity so that each chunk can be directly loaded into the queue in shared memory for subsequent BFS query processing. When a query queue is overflowed, memory chunks are dynamically allocated at runtime for the overflowed block with the help of GPU atomic operations. Overflowed elements are written into the newly allocated chunks so that the current GPU computing block can continue BFS query using the current queue. Once a chunk is processed, another chunk is loaded into the shared memory queue iteratively. Although current GPUs support dynamic allocation during the kernel execution, we still choose to maintain our own memory pool because our light-weighted memory management is more efficient for this particular application.

3.2.3 BFS based query using parallel primitives

The methods discussed in the previous subsections group a batch of queries and assign them to a computing block. Therefore, the workload is limited within a computing block. Instead of maintaining a queue for each computing block, a global queue for all queries can be maintained to maximize workload balancing in order to achieve maximum parallelism. In this new approach, an R-Tree is traversed level by level for all queries. Only intersected R-Tree nodes are expanded for the next iteration. In other words, the global queue contains children of intersected R-Tree nodes in a previous iteration. Since R-Trees are balanced search trees (all leaves are at the same height), intersected nodes at the last iteration of the queue can be output as the query results. As this method only maintains a single global queue, there is no need of overflow handling. Instead of using native languages such as CUDA, we have developed a parallel primitive based implementation that not only works on GPUs but also can be easily ported to other parallel platforms such as multi-core CPUs.

Fig 6 outlines the parallel primitives based technique. First, a global queue is maintained and it is initialized using the root of the R-Tree being queried for each query. Second, during each iteration, all queries are checked for intersection with its corresponding R-Tree node in parallel using a *transform* primitive which applies the intersection test operator for all the queries. Third, during each iteration, non-intersected pairs are removed from the queue and the queue is compacted. Fourth, intersected nodes then need to be expanded for the next iteration. This step is a combination of several parallel primitives such as *scan, scatter* and *transform*. The iteration terminates when the queue is empty or the query processing procedure reaches the last level of the R-Tree that is being queried. Finally query results are copied from the queue to an output array.

We note that there are two weak points for this new method. Firstly, the queue is maintained on global memory which cannot fully take advantage of fast shared memory on GPUs. At each level, the expansion of the current nodes incurs expensive global memory accesses, which may hurt performance. Secondly, the parallel primitives-based implementation imposes additional overhead by primitive libraries when compared with using native parallel programming languages such as CUDA.

Table 1 Datasets Sizes

Dataset	Size
abs02	1,000,000
dia02	1,000,000
par02	1,048,576
rea02	1,888,012

4. EXPERIMENTS AND EVALUATION

All experiments are performed on a workstation with two Intel E5405 processors at 2.0 GHz (8 cores in total) and one Nvidia Quadro 6000 GPU with CUDA 5.0 installed. For all experiments, -O3 flag is used for optimization. To evaluate the performance of the proposed techniques, we use benchmark datasets from R-Tree benchmark [1]. The specifications of the datasets are listed in Table 1 and Table 2. We use [13] as the baseline for CPU-based dynamic R-Tree implementation.

Table 2 Specs of Queries

	Query size	Min # of answers	Max # of answers	Avg # of answers
abs02-Q1	1,000,000	1	1	1
abs02-Q2	10,000	50	150	99.8
abs02-Q3	3,164	500	1,500	992
dia02-Q1	1,000,000	1	4	1.26
dia02-Q2	10,000	50	150	99.8
dia02-Q3	3,164	500	1,500	992
par02-Q1	1,048,576	1	10	2.11
par02-Q2	10,485	50	150	99.8
par02-Q3	3,318	500	1,500	992
rea02-Q1	1,888,012	1	9	1.2
rea02-Q2	18,880	50	162	101
rea02-Q3	5,974	501	1,514	999

4.1 Experiments on R-Tree Bulk-loading using Synthetic Datasets

The major component in R-Tree construction that dominates the overall performance is the sorting phase. We used sort implementations in existing libraries such as STL[4], TBB[5] and Thrust. In this set of experiments, we empirically set R-Tree fanout to 4 and use *x*-coordinates of MBR centroids as keys for sorting. The experiment results are given in Fig. 7, where "*CPU-serial*" denotes CPU serial implementation, "*CPU-parallel*" denotes the CPU parallel implementation, and, "*GPU-primitive*" denotes the GPU implementation based on parallel primitives. From Fig. 7 we can see that, when datasets are relatively small, parallel CPU implementations outperform GPU implementations. One explanation is that GPU parallel processing power is not fully exploited for small datasets and the overheads of utilizing parallel library cannot be hidden. We also observe that the runtimes for GPU implementations increase much slower than those of parallel CPU implementations which might indicate better scalability of the GPU implementations. In particular, when datasets become large enough that can hide library overheads, GPU implementations are several times faster than parallel CPU implementations. Following this trend, we might be able to predict that GPUs are capable of achieving better performance when bulk loading larger datasets. However, we should be aware that GPU memory capacities are usually limited when compared with CPU memory capacities. Therefore large datasets might not be able to completely reside in GPU memory. In that case, however, we still can process such large dataset using data partition techniques which are left for future work.

We have implemented and evaluated the STR R-Tree bulk loading algorithm on multi-core CPUs and the GPUs. The results are given in Fig. 8 where "*STR-CPU-Parallel*" denotes the multi-core CPU implementation and "*STR-GPU*" denotes the GPU implementation. From the results, our GPU implementation has achieved about 4X speedup over the multi-core CPU implementation. Based on the results shown in Fig. 7 and Fig. 8, low-*x* bulk loading is faster than STR bulk loading for both CPU and GPU implementations. The STR R-Tree bulk loading, as we discussed in Section 3.1, requires multiple sorts at each level. Thus, it is understandable that the overall performance of the STR R-Tree bulk loading technique is not as fast as the low-*x* bulk loading technique that only sorts once. However, as we shall show in Section 4.2.1, R-tree generated by the STR bulk loading technique usually has better quality comparing with low-*x* bulk loading and results in faster query processing, a feature that is desirable.

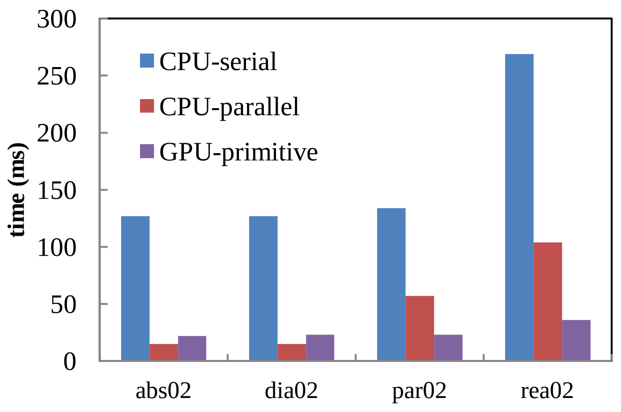

Fig. 7 Low-*x* R-Tree Bulk-Loading

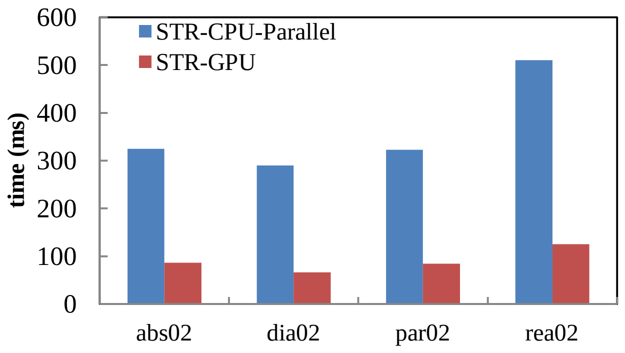

Fig. 8 Performance of STR R-Tree Bulk-Loading on Multi-Core CPUs and GPUs

4.2 Spatial Query Processing Performance using Synthetic Datasets

In this section, we conduct a set of comparisons to evaluate R-Tree based spatial window query processing performance on GPUs. We first present query performance results on R-Trees generated from different approaches and then discuss the impact of R-Tree quality in query processing which was not addressed in [12]. We also compare DFS-based and BFS-based query processing techniques. For the BFS-based technique, we evaluated different overflow handling approaches.

4.2.1 Comparisons on using different R-Trees

As discussed previously, the quality of R-Trees will impact the query performance. In this set of experiments, we have used three different R-Trees, including low-*x* packed R-Tree (low-*x*), STR R-Tree (STR) and R-Tree generated via dynamic insertion (dynamic), and the results are presented in Fig. 9. From the results we can see that, for small datasets such as abs02 and dia02, the difference of the three R-Trees is negligible. However, when the size of dataset increases, the impact of R-Tree quality is significant, especially for the largest dataset rea02. We can also observe that STR R-Tree almost always has better query performance than R-Tree generated via dynamic insertion in this set of experiments. We note that, while STR R-Tree bulk loading has been implemented on the GPU, dynamic insertion on R-Tree is still difficult to parallelize. Furthermore, as both R-Tree construction and query processing can be accomplished on the GPU without transferring data back and forth between CPU memory and GPU memory, STR R-Tree is more preferable. Due to its excellent performance and the desirable features, for the rest of our experiments, we will use STR R-Tree as the default R-Tree construction algorithm unless otherwise explicitly stated.

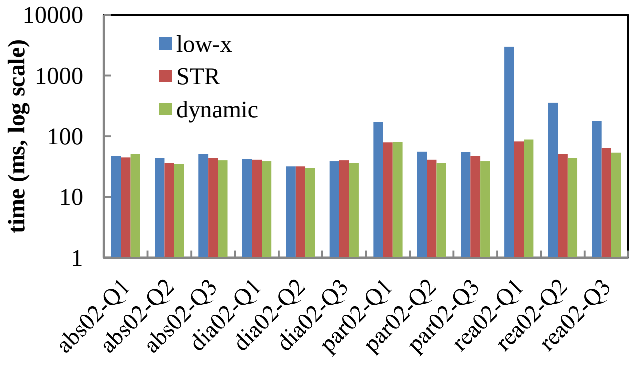

Fig. 9 Query Performance on Different R-Trees

4.2.2 Comparison between DFS and BFS based approaches

As shall show in Section 4.2.3, the BFS-based query processing technique with dynamic memory allocation based overflow handling achieves the best performance. As such, we use it as the baseline to compare against the DFS-based query processing implementation. Based on the results shown in Fig. 10, the BFS implementation outperforms its counterpart in all queries significantly. As discussed previously, DFS batched query suffers from two disadvantages. First, the "count and write" pattern in DFS actually requires two nearly identical kernel executions which nearly doubles the running time. Second, when the size of returning results for each query increases (e.g. Q3), it is likely that global memory accesses in "write" kernel are not coalesced because each thread writes its own results individually.

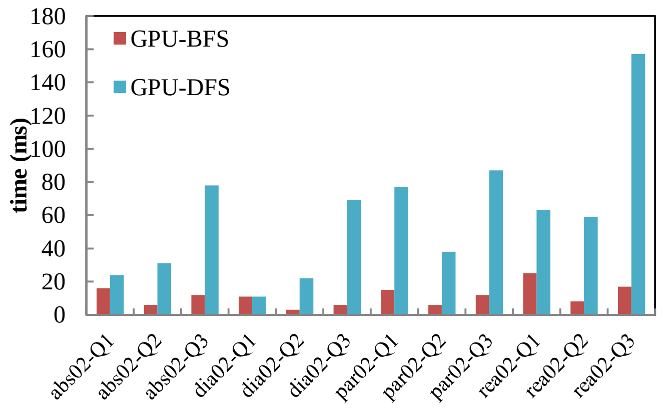

Fig. 10 Performance Comparisons among DFS- and BFS-Based Query Processing

4.2.3 Comparisons of different overflow handling mechanisms in BFS-based query processing

We use STR R-Tree as the spatial index in this set of experiments. The experiment results of using different overflow handling approaches are given in Fig. 11 where "*GPU-hybrid*" denotes DFS based overflow handling, "*GPU-memory*" denotes dynamic memory overflow handling, "*GPU-relaunch*" denotes using the re-launching mechanism, and, "*GPU-primitive*" denotes the parallel primitive based implementation on GPUs. For all experiments, the number of queries in a block and the size of queue are empirically set. As mentioned before, the DFS-based overflow handling mechanism fails in some scenarios especially when queries generate large number of elements in results. For example, Q3 queries return approximately 1,000 results on average and have very bad performance when compared with other techniques. In contrast, the dynamic memory allocation approach is almost the best in all queries. The re-launching method suffers from queries that have large results when overflow happens frequently and multiple rounds of invocations of overflow handling kernels are required in those queries, which hurt the overall performance. We also observe that parallel primitive based implementation is not as fast as the others, especially when query results are small in sizes as parallel library overheads may dominate. However, parallel primitives-based implementation is more scalable and no explicit parameter tuning is required which is desirable. For instance, in the hybrid query processing strategy, the queue size for a batched query in a computing block has to be carefully chosen. For all BFS-based query processing techniques, the number of queries assigned to a block also needs to be tuned. Furthermore, the parallel primitive based implementation makes the design easily applicable to other parallel environments such as multi-core CPUs, which is also desirable.

4.2.4 Comparisons on query processing performance on multi-core CPUs and GPUs

In this section, we compare the performance of batched query processing on the GPU with multi-core CPU implementations. The multi-core CPU implementations utilize all available cores (8 cores in total) in the system based on OpenMP[6] where each core is responsible for a single query. As can be seen from Fig. 12, our GPU implementations have achieved about 10X speedup on average when compared with multi-core CPU implementations.

For small queries such Q1 queries, GPU implementations did not show advantages over multi-core CPU implementations. However, as the size of query results in each query window increases, GPU based implementations outperform their counterparts significantly.

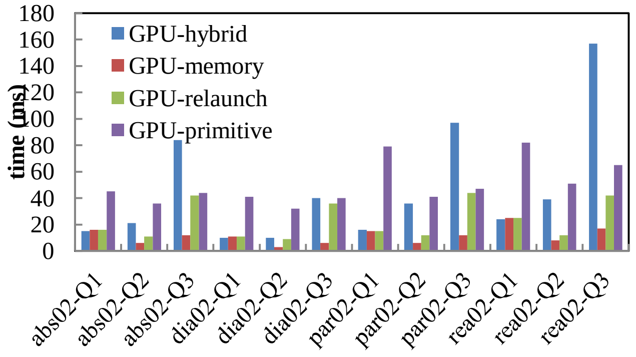

Fig. 11 Query results on the GPU

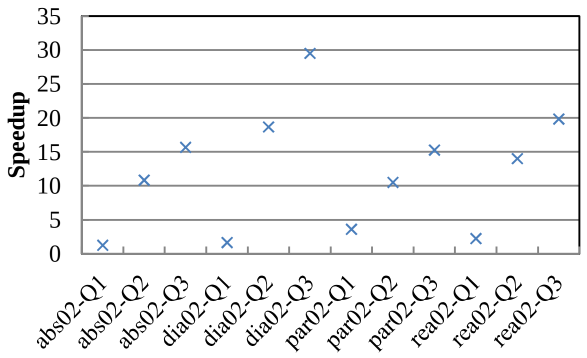

Fig. 12 Speedups of GPU-based Implementations over Multi-Core CPU-based Implementations for Spatial Window Query Processing on Synthetic Data

4.3 Experiments using Real-world Datasets

In addition to performing evaluations on benchmark datasets, we also conduct experiments using datasets from real-world applications. Two real-world datasets are extracted from a point-in-polygon test based spatial join application. Details of the application and the datasets are provided in our previous work [19]. The first dataset, termed as *taxi*, is derived from 170 million taxi pickup location in 2009 in NYC that contains 3,415,981 MBRs. The second dataset, called *pluto*, comes from NYC tax lot dataset that consists 735,488 MBRs. The fanout of R-Trees in the set of experiments is empirically set to 8.

The first experiment is to compare R-Tree bulk loading performance between the CPU and GPU implementations. As discussed earlier, both implementations adopt the STR R-Tree bulk loading algorithm. The CPU implementation also utilizes all available cores in the system (8 cores in total). Based on the results shown in Table 3, GPU implementation achieves about 4X speedup against CPU parallel implementation which is similar to our findings in Section 4.1.

For the experiments on query processing, one dataset is chosen as indexed dataset and the other is considered as query windows for both datasets. The results of the four experiments are listed in Table 4. From the results we can see that GPU implementations have achieved about 8X speedups over parallel CPU implementations, which is similar to the results reported in Section 4.2.

Table 3 Runtimes of R-Tree Bulk Loading on Real Datasets

Dataset	GPU (ms)	CPU (ms)
Taxi	207	1,111
Pluto	70	247

Table 4 Query performance on Real Datasets

Indexed dataset	Query dataset	GPU (ms)	CPU (ms)
taxi	pluto	338	2,974
pluto	taxi	1,191	10,538
taxi	taxi	1,414	12,255
pluto	pluto	322	2,736

5. CONCLUSIONS AND FUTURE WORK

In this study, we have implemented parallel designs of bulk loading R-Trees and several parallel query processing techniques on GPUs using R-Trees. Our extensive experiments have shown that the GPU parallel query implementations can achieve significant speedups over multi-core CPU based implementations which makes the GPU-based R-Tree construction and query processing techniques attractive for many real world applications. Our experiments also have shown that R-Tree qualities can have significant impacts on query performance on GPUs. Building high quality R-Tree on the GPU is crucial to achieve high performance in query processing.

An interesting observation in [7] pointed out that space-driven indexes (e.g., quadtree variants) worked better than data-driven indexes (e.g., R-Tree variants) in a parallel computing context (e.g., the Thinking Machine CM-5 used in the experiments). However, it is unclear to what degree the observation still holds on modern GPUs which have a quite different parallel hardware architecture. For our future work, in addition to further investigations on GPU based bulk loading that have been discussed inline, we also plan to compare R-Tree based indexing approaches with quadtree based ones on GPUs to further explore their respective advantages and disadvantages. Another research direction we plan to follow is to investigate on how to reorder or index query windows for more efficient parallel query processing on GPUs. Finally, we plan to evaluate other R-tree bulk loading heuristics on the GPU, such as adopting dynamic programming based optimization proposed in [2].

6. ACKNOWLEDGEMENT

This work is supported in part by NSF Grant IIS-1302423 and by the Doctoral Student Research Grant program from CUNY Graduate Center.

7. REFERENCES

[1] A Benchmark for Multidimensional Index Structures: *http://www.mathematik.uni-marburg.de/~rstar/benchmark/*.

[2] Achakeev, D. et al. 2012. Sort-based parallel loading of R-trees. *Proceedings of the 1st ACM SIGSPATIAL*

International Workshop on Analytics for Big Geospatial Data - BigSpatial '12, 62–70.

[3] Alborzi, H. and Samet, H. 2007. Execution time analysis of a top-down R-tree construction algorithm. *Information Processing Letters*. 101, 1 (Jan. 2007), 6–12.

[4] Gaede, V. and Günther, O. 1998. Multidimensional access methods. *ACM Computing Surveys*. 30, 2 (Jun. 1998), 170–231.

[5] García R, Y.J. et al. 1998. A greedy algorithm for bulk loading r-trees. *Proceedings of the sixth ACM international symposium on Advances in geographic information systems - GIS '98*, 163–164.

[6] Guttman, A. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84*, 47.

[7] Hoel, E.G. and Samet, H. 1994. Performance of Data-Parallel Spatial Operations. *VLDB '94 Proceedings of the 20th International Conference on Very Large Data Bases* (Sep. 1994), 156–167.

[8] Kamel, I. and Faloutsos, C. 1993. On packing R-trees. *Proceedings of the second international conference on Information and knowledge management - CIKM '93*, 490–499.

[9] Kamel, I. and Faloutsos, C. 1992. Parallel R-trees. *Proceedings of the 1992 ACM SIGMOD international conference on Management of data - SIGMOD '92*, 195–204.

[10] Kim, J. et al. 2013. Parallel multi-dimensional range query processing with R-trees on GPU. *Journal of Parallel and Distributed Computing*. 73, 8 (Apr. 2013), 1195–1207.

[11] Leutenegger, S.T. et al. 1997. STR: a simple and efficient algorithm for R-tree packing. *Proceedings 13th International Conference on Data Engineering* (1997), 497–506.

[12] Luo, L. et al. 2012. Parallel implementation of R-trees on the GPU. *17th Asia and South Pacific Design Automation Conference* (Jan. 2012), 353–358.

[13] Open source R-Tree Implmenenation: *http://superliminal.com/sources/sources.htm*.

[14] Papadopoulos, A. and Manolopoulos, Y. 2003. Parallel bulk-loading of spatial data. *Parallel Computing*. 29, 10 (Oct. 2003), 1419–1444.

[15] Samet, H. 2005. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc.

[16] Schnitzer, B. and Leutenegger, S.T. 1998. Master-client R-trees: a new parallel R-tree architecture. *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management* (Jul. 1998), 68–77.

[17] Wang, B. et al. Parallel R-tree search algorithm on DSVM. *Proceedings. 6th International Conference on Advanced Systems for Advanced Applications* 237–244.

[18] Zhang, J. et al. 2011. Parallel quadtree coding of large-scale raster geospatial data on GPGPUs. *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS '11*, 457.

[19] Zhang, J. and You, S. 2012. Speeding up large-scale point-in-polygon test based spatial join on GPUs. *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data - BigSpatial '12*, 23–32.

[1] http://en.wikipedia.org/wiki/Parallel_Random_Access_Machine
[2] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications
[3] https://developer.nvidia.com/category/zone/cuda-zone
[4] http://www.sgi.com/tech/stl/
[5] http://www.threadingbuildingblocks.org/
[6] http://openmp.org/