GPU-based Batched Spatial Query Processing on 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

ABSTRACT

R-trees are popular spatial indexing techniques that have been widely used in many geospatial applications. The increasingly available Graphics Processing Units (GPUs) resources for general computing have attracted considerable research interests in applying the massive data parallel technologies to index and query geospatial data based on R-trees. In this paper, we investigate on the potentials of accelerating both R-tree bulk loading construction and R-tree based spatial window query on GPUs. Experiments show that our proposed GPU-based parallel query processing implementation achieves 6x~18x speedup over serial CPU implementations and is 2X faster on average over 8-core CPU implementation using OpenMP. Our experiments also show that the speedups are significantly affected by R-tree qualities which warrants further investigations. Additional comparisons between the GPU R-tree implementation and a GPU single-level grid-file based indexing approach are performed to understand the relative advantages and disadvantages of the two popular spatial indexing approaches on GPUs.

Keywords

Spatial Indexing, R-trees, GPU

1. Introduction

R-trees [1-3] are well known spatial indexing techniques and have been widely adopted in many applications for indexing 2-D or higher dimensional data. Several parallel R-Tree construction and query processing algorithms [4-9] have been proposed. While early research works mostly focused on shared-nothing computer clusters [4-8], a few recent works (e.g., [9]) have implemented R-Tree based construction and query processing on GPUs based on the General Purpose computing on GPUs (GPGPU) technologies.

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, greater memory bandwidth and higher computing power with more affordable prices. For example, the Nvidia GTX 690 GPUs[2] has more than 3,000 processing cores, nearly 400 GB/s bandwidth, nearly 6 TFLOPS/s 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-based spatial data management in particular. Furthermore, GPUs have been extensively used to accelerating more computing intensive applications, such as nearest neighbor queries in databases and information retrieval and ray-tracing in computer graphics. Efficient indexing structures such as R-Trees are promising in speeding up such computing on GPUs are practically useful. As it is still quite expensive to transfer data between CPUs and GPUs through PCI-E buses (limited to 4 GB/s on PCI-E 2 devices), being able to directly construct and query R-Trees on GPUs to avoid or reduce data transfer overheads is beneficial, even if the speedups of index construction and query processing themselves are not significant from a practical perspective.

Towards this end, we have explored different design strategies on R-tree construction and query processing on GPUs and evaluated the performances using real datasets. Compared with the work reported in [9] that is most related to our work, in addition to the Breadth-First Search (BFS) tree traversal based query processing, we have also implemented the Depth-First Search (DFS) tree traversal based query processing which leads to a different overflow handling mechanism with certain advantages. While the R-Tree packing based bulk loading [10, 11] has been implemented in [9], the packed R-Tree was not used for query processing in [9]. In contrast, we have compared the performance of query processing using both bulk loaded R-Trees and dynamically generated R-Trees and revealed that the inferior quality of packed R-Trees (using different ordering) through bulk loading may contribute to poor performance of query processing on GPUs.

While some of our designs and implementations have resulted significant speedups over serial 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 and those over alternative choices (such as single-level grid-file) to provide insights and guidelines for more systematic and more efficient implementations. For testing purposes, we have experimented our implementations on the Minimum Bounding Boxes (MBRs) of two datasets that have 990,142 MBRs (taxi quadrants) and 735,488 MBRs (tax lots) in the New York City (NYC), respectively. 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 both a BFS and DFS R-Tree traversal based query processing designs on GPUs.

3) We have presented a new overflow handling mechanism by combining BFS and DFS based query processing with certain advantages over [9].
4) We have performed extensive experiments to compare the query processing performance among GPU-based R-Trees, CPU-based R-Trees and GPU-based single-level grid-file to support further investigations.

The rest of this paper is organized as follows. Section 2 briefly introduces background and related work. Section 3 provides our GPU R-tree designs and implementations on GPUs, including bulk-loading and BFS and DFS based query processing approaches. Section 4 presents experiments and results. Finally Section 5 is the conclusion and future work discussions.

2. Background and Related Work

R-Tree based indexing has been extensively studied in spatial databases and quite a few variants have been proposed over the past two decades [1-3]. Although most existing R-Tree implementations are serial on a single CPU, there are several previous studies on parallel R-Tree construction and query processing based on different parallel hardware architectures with the ones based on shared-nothing clusters dominate [4-8]. In this study, we focus on the R-Tree based spatial indexing using GPGPU computing technologies that have a quite different parallel computing model. However, we argue that our work is complementary to cluster computing based distributed and parallel spatial query processing provided that the 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 [9], has implemented a BFS traversal based query processing algorithm on modern GPUs. In addition to the differences that have been discussed in the introduction section, our implementations also different from the following aspects. First, as discussed in details in Section 3.1, 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 [13]), our node layout has a separate field to store the number of children of a R-Tree node. By using at most an 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 [9] focused on the performance of a single R-Tree query processing implementation, we have compared two different R-Tree query processing implementations and compared with single-level grid-file based query processing, all on GPUs.

An interesting observation in [5] pointed out that space-driven indexes (e.g., quadtree variants) worked better than data-driven indexes (e.g., R-Tree variants) in a parallel 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. Although we are only able to compare the R-Tree based implementations with a single-level grid-file based implementation on GPUs due to the complexities in implementing the PMR-quadtree used in [5] on GPUs, as quadtrees are closely related to grid-file, the comparison may shed some light on the comparisons between PMR-quadtree and R-Trees. A more comprehensive comparison is left for our future work.

While the original R-Tree construction algorithms use dynamic insertions, several bulk loading approaches have been proposed [10-12, 8]. Bulk loading approaches usually adopt a packing technique to construct R-trees which maximize space utilization and reduce height of the resulting trees as much as possible. Packed R-tree guarantees better space utilization and query responses have been reported to be at least as fast as R-trees built from dynamic insertion [10, 11]. Bulk loading methods can be classified into two categories, i.e., top-down and bottom-up. Alborzi and Samet [12] discussed the difference between top-down and bottom-up strategies. They claimed that the top-down method could potentially process queries faster but non-leaf nodes might be underpacked. On the other hand, R-trees constructed by bottom-up methods have fewer nodes than using top-down methods. On GPUs, Luo etc. argued that for both methods there are no fundamental differences as both of them were constructed by sorting and packing [9].

Bulk loading is more suitable for static read-only data in OLAP[3] (Online Analytic Processing) settings in many applications. Assuming that MBRs of geospatial data can fit into processor memory (which is increasingly becoming practical due to the decreasing prices of memories), the cost of bulk loading is largely determined by in-memory sorting in the order of O(nlogn). The required sorting for bulk loading can be significantly accelerated on GPUs by utilizing the parallel computing power which makes GPU implementations attractive. However, for MBRs with varies sizes of degrees of overlapping, the qualities of constructed R-Trees through bulk loading can be very different which may significantly affect query performance on both CPUs and GPUs. While a bulk loading algorithm has been implemented on GPUs, the bulk loaded R-Trees were not used for querying in [9]. In contrast, in this study, we compare the query performance of bulk loaded R-Trees using different orderings and provide discussions on the effects of query performance due to R-Tree qualities. A promising research idea is to exploit the parallel computing power of GPUs to improve the qualities during R-Tree bulk loading which is also left for future work.

We have implemented a single-level grid-file based spatial indexing for the filtering phase of spatial joins and used it extensive in several applications [14,15,17]. As spatial filtering is closely related to batched spatial query, the comparison among the two classic spatial indexing approaches [2, 3] on GPUs are interesting. To be self-contained, we next briefly introduce our implementation of the single-level grid-file based spatial indexing and query processing on GPUs. The indexing approach is based on the decomposition of two sets of MBRs according to a uniform grid space whose resolution is pre-defined by users. Using the example shown in Fig. 1, we assume that {P1, P2} and {Q1, Q2} are the query MBRs and data MBRs, respectively. To find the P-Q pairs that spatially intersect, first, Q1 and Q2 are decomposed onto the grid to generate two vectors, assuming they are VQQ and VQC, respectively. VQQ stores original identifiers of Q1 and Q2 and VQC stores the corresponding grid cell identifiers (derived from x, y coordinates in the grid space) decomposed from Q1 and Q2. VQQ are sorted in parallel by using VQC as the key so that Q MBRs that overlap with a same grid cell are stored consecutively in VQQ. Second, P1 and P2 are decomposed in the same way and VPC and VPP are generated. Here VPC stores cell identifiers and VPP stores the original identifiers of P1 and P2. A MBR decomposition kernel on GPUs is developed for the first two steps. Third, to query P MBRs over Q MBRs, two binary searches (one for lower bound search and one for upper bound search) of the elements in VPC over VPC are performed to match the MBRs stored in VPP and VQQ. Finally, duplicated pairs are removed by combining a parallel *sort* and a parallel *unique* operation. As all the involved operations except MBR decomposition, i.e., *sort*, *search*, *unique*, can be efficiently parallelized in quite a few parallel libraries including Thrust[4] that comes with CUDA SDK, the single-level grid-file based spatial query can be relatively easily implemented on GPUs. Despite that the implementation has been extensively used in our previous studies, it has not been compared with alternative implementations. The R-Tree implementations reported in this paper serve as a good opportunity.

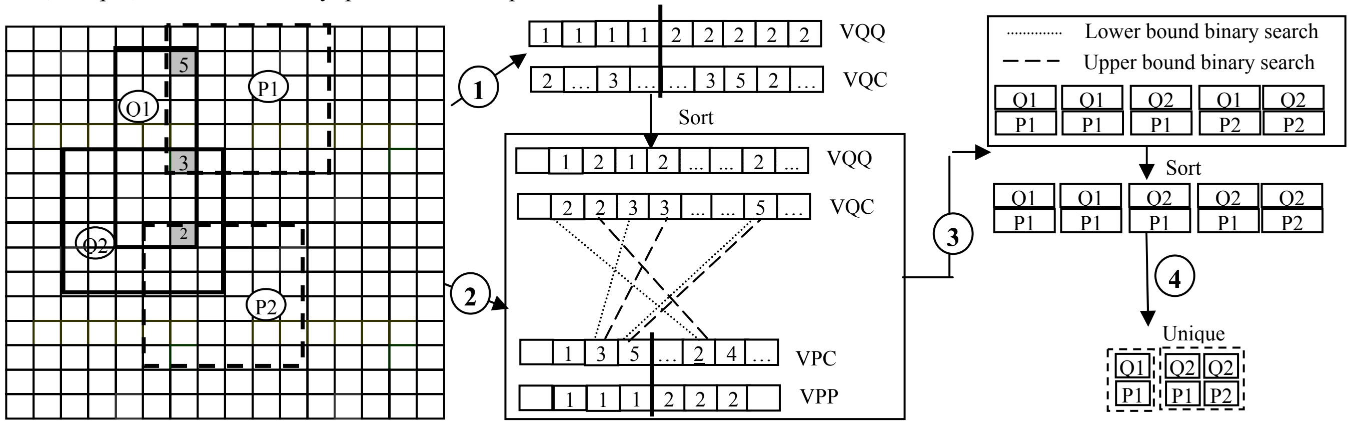

Fig. 1 Example of grid based index

3. GPU Based R-tree Indexing and Querying

In this section, we present our implementations of parallel construction of R-Trees and query processing based on R-tree indexing on GPUs. We will first introduce the node layout of our linearized R-tree design followed by the implementation details of R-Tree construction using parallel bulk loading. We then focus on both BFS and DFS R-Tree traversal based query processing on GPUs and discuss some of the design and implementation choices.

3.1 Node Layout of Linearized R-tree

We use simple array based linear data structures to represent an R-tree. Simple linear data structures can be easily streamed between CPU main memory and GPU memory without serialization and is 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 box of the corresponding node, *pos* and *len* are first child position and number of children, respectively. The node layout is illustrated in Fig. 2. The tree nodes are sequenced into an array based on the DFS ordering. The decision to record only the first child node position instead of recording the positions of all child nodes 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 Section 3.2. When the R-Tree is generated on CPUs, the two fields can be filled easily by sequentially looping through R-Tree nodes through pointer chasing.

Both Array of Structures (AoS) and Structure of Arrays (SoA) can be adopted to physically store R-Tree nodes. We choose SoA on GPUs in this study as it is more beneficial for coalesced memory accesses to GPU global memory. As an example, assuming, two threads read two consecutive R-Tree nodes from global memory. In AoS, two structures representing the two nodes are loaded. Since each structure has a size of 24 bytes (4 floats for MBR, 2 integers for *pos* and *len*), it will result in non-coalesced global memory access on the current GPU architecture. On the contrary, SoA splits the node structures into multiple single value arrays. With the same example, each time a single value (4 bytes for either floats or integers) is read and the accesses are coalesced.

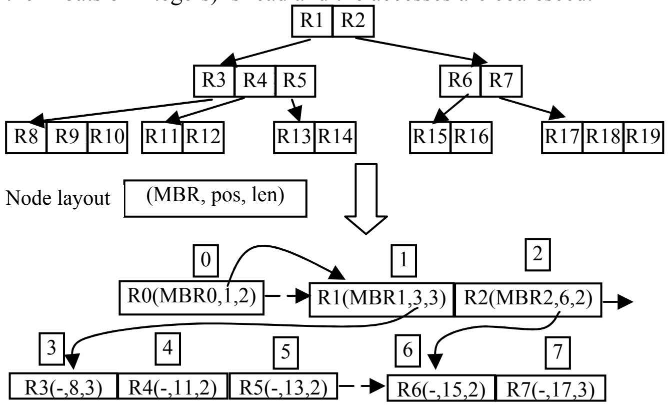

Fig. 2 Illustration of Linear R-tree Node layout

3.2 Parallel Bulk Loading R-Tree on GPUs

In this study, we have chosen to adopt the bottom-up approach that has also been used in [9] for bulk loading R-Tree with two phases: sorting and packing. However, instead of using CUDA programming directly, our implementation is built on top of the parallel primitives provided by Thrust that comes with CUDA SDK. The decision has significantly reduced coding complexity and improved portability. In the sorting stage, the original data (MBRs) is sorted by applying a linear ordering schema using a *sort_by_key* primitive. We note that the linear ordering schema will directly impact the qualities of constructed R-trees and subsequently impact the query performance on R-Trees [10,11].

This is because spatial adjacency in 2-D may not be well preserved in 1-D, an issue that has been well studied in spatial databases [16,2,3].

```
1. R_sz = 0; num_lev = 0;
2. while(length != 1)
3.   length = ceil(length/d);
4.   nodes_by_lev.push_back(num_nodes);
5.   num_lev++;
6.   R_sz += length;
7. for level = num_lev decreasae to 1
8.   if (num_lev == num_lev)
9.     reduce_by_key from original data
10.  else
11.    reduce_by_key from lower level
```

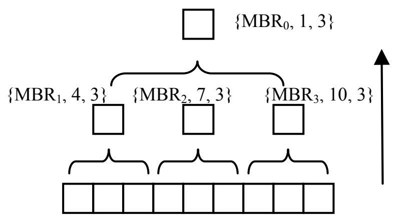

Fig. 3 Implemenation of Parallel R-tree Bulk Loading on GPUs with a Running Example

In the packing phase, as shown in Fig. 3, 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 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 from bottom up using a *reduce_by_key* primitive. For the algorithm shown in the top of Fig. 3, Steps 1 to 6 calculate the number of levels and the number of nodes at each level. Steps 7 to 11 construct the tree level by level. In step 9 and 11, same keys are need to be generated every d items for parallel reduction purpose. This can be achieved by combining *transform_iterator* and *counting_iterator* primitives provided by Thrust. 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* 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). A running example is included in bottom of Fig. 3.

3.3 Parallel Query on R-Trees

Different from bulk-loading, we have chosen to implement R-Tree based queries on GPUs using CUDA for both efficiency and flexibilities. This is partially because the difficulties in mapping irregular tree-traversals that are required for processing spatial queries that involve 2D operations while the parallel primitives in parallel libraries that predominately support 1D structures only. For GPUs that support Nvidia CUDA programming model, there are two levels of parallelism, i.e., computing block level and thread level. In a way similar to the strategy adopted in [9], queries are assigned to computing blocks to utilize the first level parallelism. Each computing block processes a batch of queries by coordinating the threads within the computing block to utilize the second level of parallelism. The query problem can be formulated as the following. Given a set of query rectangles $\boldsymbol{Q}$ and a set of MBRs $\boldsymbol{P}$ that has been indexed, a query on an R-Tree returns a list of pairs $\{(q, p) \mid q \in \boldsymbol{Q}, p \in \boldsymbol{P}$, q and p intersects. Without further optimization at the computing block level (which is left for future work), by sequentially assigning every S queries to a computing block, we next present two approaches, i.e., DFS and BFS based batched query processing algorithms on R-Trees within a computing block

3.3.1 *Depth First Search Based Query*

In this approach, each thread processes a query in a DFS manner and thus a stack is required to track visited nodes. A naïve implementation can be maintaining the stack on global memory and each thread does its own work. Observing that the stack is frequently read and write but the global memory access pattern is not coalesced, 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 has a depth of l, a stack with size larger than l is sufficient. Since our construction algorithm guarantees the depth of R-tree to be $log_d(n)$, an appropriate fanout d value will give a reasonable depth less than l. For example, with a fanout of 8, the R-tree of a dataset with 990,141 items (a dataset that will be used in the experiments) only has 7 levels.

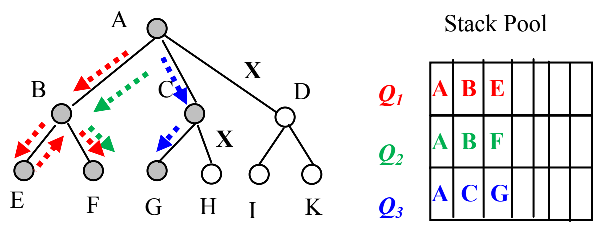

Figure 4 A Running Example of Batched DFS Queries

For batched query within a computing block, the number of batched queries m is determined by size of available shared memory (*sm_sz*) in a computing block using a stack size *stack_sz*, $m = floor(sm\_sz/stack\_sz)$. To keep track of visited information in DFS traversals, the data items in the stack are organized using two fields, *index* and *visit*. *index* is the index to the R-tree node array that provides access to the corresponding R-tree node. The other field *visit* is used for recording the number of visited children under the current R-Tree node. Fig. 4 (best viewed in color) is a running example of the batched DFS based query processing where three queries ($\boldsymbol{Q_1}$, $\boldsymbol{Q_2}$, $\boldsymbol{Q_3}$) are executed in parallel, each by a thread. In the example, grey nodes indicate nodes that have been visited by at least one thread while white nodes indicate pruned nodes. A stack pool is maintained in shared memory where each query/thread works on its own stack. Among the three queries, Q1 requires back tracing to B after visiting E before finally reaching F by using the stack of the corresponding thread. E and F are the leaf R-Tree nodes that should be returned. Differently, Q2 and Q3 only follow a single path and results the queries result in a single R-Tree leaf 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. 5 is the detailed implementation of the "count" phase. In addition to the stack pool structure in shared memory discussed before, an array *Pos* is allocated for 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 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 modification in Steps 16 and 19, i.e., instead of simply counting number of hits, the query results are output to the allocated array.

```
1. //count
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] = Hit;
```

Fig. 5 Implementation of the Counting Phase of the DFS based Query Processing on GPUs

As the alert readers might have observed, the duplicated processes may hurt the performance of the DFS based query processing implementation. The counting phase is essentially the overhead for thread coordination in parallel computing. In addition to the fact that the parallelism of DFS based query is limited by the size of shared memory in terms of the available number of threads in a computing block (as discussed earlier), workloads among the threads in a computing block may be imbalanced as threads work independently. When large query windows such as Q1 in Fig. 4 (which requires significant back tracking) and small query windows such as Q2 and Query 3 in Fig. 4 (which usually follow a single query path) are assigned to the same computing block, the imbalanced workloads may significantly hurt the performance.

3.3.2 Breadth First Search Based Query

Similar to [9], a global queue in a computing block is used for all threads inside the block to process all the batched queries assigned to a computing block. The queue node is represented in the form of {*index, qid*} where *index* is the index to the R-tree node array so that the corresponding R-Tree node can be retrieved (the same as in DFS based one). *qid* represents the identifier of the query that is being processed. R-Tree nodes whose MBRs intersect with any of the queries are expanded in parallel and stored in the queue level by level.

Unlike DFS based query processing where the sizes of outputs are computed in a separate phase for each query/thread, in BFS-based query processing a computing block has its own global memory space for writing out query results which are predefined. 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 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, a per-block flag is needed to indicate whether BFS based query processing in a computing block is successful. The flag will be set if an overflow happens and the query process in the computing block will stop.

If one or more overflows happens during BFS based query processing among all computing blocks, an overflow handling mechanism is needed to complete the query process. The solution adopted in [9] is to launch new kernels repetitively until no overflow happens. The queue size is increased for each successive round of kernel launch to reduce the probability of overflows while minimizing wasting global memory. In this study, we have adopted a different strategy to handle overflows by integrating DFS and BFS based query processing. When an overflow happens in a computing block, the overflow flag is set and the current BFS queue in the computing block is copied to global memory. After the BFS kernel is complete, a DFS kernel introduced in Section 3.3.1 is then started by assigning each queue node of blocks that overflow to a thread and the query process continues by switching to DFS based query processing. As DFS query processing adopts the "count-and-write" pattern, there will be no more overflow happens. As such, the combined BFS+DFS based query processing (herein referred as the hybrid approach) only needs two kernel launches. Correspondingly, only one additional global memory allocation is needed after the "count" phase is completed in the DFS based query processing. In contrast, the solution in [9] needs more kernel launches and more memory allocations whose numbers cannot predetermined.

```
__shared__ QUEUE[]
for each query i parallel do
  QUEUE[i] = {0, qid};
while (!done & !overflow)
  for each thread parallel do
    {index, qid} = QUEUE[threadIdx]
    hit = 0
    for each child i of R[index]
      if (intersect(MBR[qid], R[i].MBR))
        hit++
  pos = Prefix_Scan(hit)
  if (threadIdx == NUM_THREADS-1)
    if (pos+hit == 0) done = true
    if (pos+hit > Q_SZ) overflow = true
  if (!done & !overflow)
    for each thread parallel do
    {index, qid} = QUEUE[threadIdx]
    for each child i of R[index]
      if (intersect(MBR[qid], R[i].MBR))
        QUEUE[pos++] = {i, qid}
for each thread parallel do
  save QUEUE to global memory
```

Fig. 6 Implementation of the BFS based Query Processing on GPUs

The implementation of the BFS based query processing design is listed in Fig. 6. Note that the algorithm presented in Fig. 6 is only for BFS based query processing and a complete listing of the hybrid approach for end-to-end application would require combining the algorithms listed in Fig. 6 and Fig. 5. In Fig. 6, during the initialization phase (line 2~3), each query is loaded into the per-block queue in parallel with the *index* field set to 0 to start from the root of the R-tree. Line 4~20 process the batched queries in parallel on the R-Tree tree level by level with each thread works on an entry of the queue. Each thread first determines how many nodes need to be expanded for the next level by accessing the len field of the R-tree node whose array index is the value in the *index* field of the queue node. A computing block level prefix scan is then used to computes the positions for outputting the child nodes of the current R-Tree node whose MBRs intersect with the window of the query that is assigned to the thread. The process is repeated for all R-Tree levels until either the R-Tree leaf nodes are reached (with a *done* flag) or overflows are detected (with an *overflow* flag). Finally, in the normal termination case, the queue is saved from shared memory to global memory as the output (Line 21 and 22). In the overflow case, a DFS based query processing is started to continue the query processing as discussed previously.

To better illustrate the BFS based query processing and the hybrid approach, Fig. 7 provides a running example for the BFS based query processing with no overflows and Fig. 8 provides a running example for the hybrid BFS+DFS approach where an overflow happens. The two figures are best viewed in color. In both figures, grey nodes represent non-leaf R-Tree nodes whose MBRs intersect with query windows and their child nodes should be further testes for spatial intersection tests on the respective MBRs. The white nodes represent the opposite case and they should be pruned for further tests. In addition, black nodes represent the leaf nodes whose MBRs intersect with query windows and the pairs of the corresponding R-Tree node identifiers and the query window identifiers should be returned. In both examples, we assume there are two batched queries (whose query paths are symbolized using red and green arrows, respectively) in a computing block (with two threads) and the queue capacity is 3.

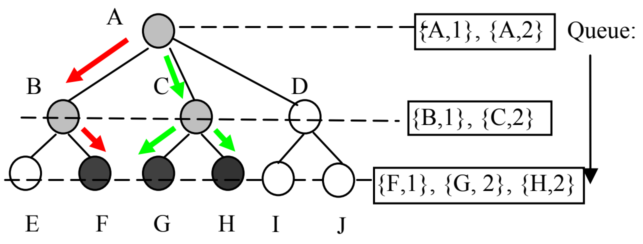

Fig. 7 Running Example for BFS based Query Processing without Overflows

In Fig. 7, as the MBRs of the two queries intersect with node A (root node) at the level 0, two queue nodes {A,1} and {A, 2} are enqueued. Thread 1 processes query 1 by dequeuing {A,1} and tests whether the MBRs of the three child nodes (B,C,D) intersect with the MBR of query 1. Assuming that only the MBRs of node B intersects with the MBR of query 1, {B,1} is thus enqueued. Similarly, thread 2 dequeues {A,2} and enqueus {C,2} at the level 1. Following the same procedure, thread 1 dequeues {B,1} and enqueues {F,1} and thread 2 dequeues {C,2} and enqueues {G,2} and {H,2}, respectively, at the level 2. Since the capacity of the queue is 3, no overflows happen.

In Fig. 8, following the same BFS based query processing procedure, the queue at the level 1 would be {B,1}, {C, 1} and (D,2). We have abbreviated {*index, qid*} as *index* $_{qid}$. in Fig. 8 due to space limit for presentation. For example, {B,1}is abbreviated as B_1. However, following the same procedure, there would be 4 nodes in the queue at the level 2 which is beyond the capacity of the queue size. As such, the previous queue state (B_1 C_1 D_2) is copied to GPU global memory and the BFS stage for the computing block terminates. When the DFS stage in the hybrid approach begins, the batched queries that have overflow flags are processed by first copying back the respective queue to shared memory and then using the values of the *index* field of the queue nodes as the stack tops (c.f. Fig. 4). For this particular example, the leaf nodes F and G are reached by an one-step node expansion from B and C, respectively. Differently, for node D, a back tracking process is needed to retrieve its two child nodes as both of their MBRs intersect with the MBR of query 2.

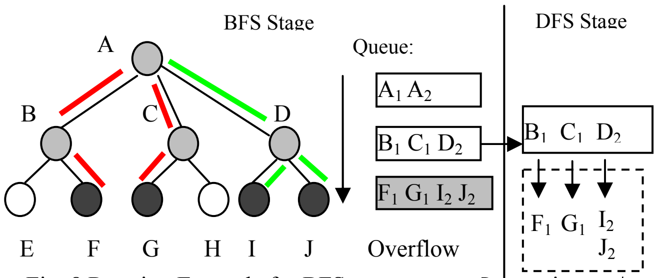

Fig. 8 Running Example for BFS based Query Processing with Overflows – the Hybrid Approach

3.3.3 *Discussions*

3.3.3.1 *Comparing DFS and BFS based Approaches*

For BFS based query processing, the memory access pattern is generally better than that of DFS based one. First, DFS traversal order does not match the level-by-level (BFS) based R-tree node sequence where BFS traversal is a better match. BFS based query processing may better utilize the L2 cache introduced in Fermi GPU architecture better. As BFS uses a global queue for all threads that process all batched queries simultaneously, workloads are better balanced among threads. At any tree level, if a queue has N nodes and there are K threads assigned to a computing block, then the N expansion tasks are almost evenly distributed to the K threads although the workload within an expansion might be uneven as R-Tree nodes may have different numbers of child nodes whose MBRs are needed to be checked with the MBR of the query window that is assigned to the corresponding thread. That being said, as the number of child nodes of an R-tree node is between M/2 and M, the degree of imbalance is well bounded which makes BFS based query processing desirable from a load balancing perspective which usually have a positive impact on overall performance. Furthermore, as neighboring threads access neighboring nodes in the queue, memory accesses are better coalesced for DFS based query processing.

On the other hand, as discussed earlier, DFS based query processing has a better memory utilization because the output memory is allocated after first counting the size of results. In contrast, DFS-based query processing is prone to either overflows or memory underutilization. For applications where GPU memory capacity is a limiting factor, DFS based method can be a choice.

However, due to the duplicated computing overheads, unbalanced workloads and limited scalability, the performance of DFS based query processing is expected to be inferior to BFS based ones which is supported by our experiments. This makes the hybrid approach attractive by balancing probabilities of memory underutilization and overflows. In general, when memory is not a limiting factor, BFS based query processing will have better performance and should be preferred.

3.3.3.2 Impacts of R-tree Quality

The qualities of R-trees are known to have great impacts on query performance on CPUs. The same effect remains true on GPUs. For BFS based method, R-trees with large overlapping MBRs cannot prune branches at the top levels. Thus, more nodes need to be loaded into the queue and tested for intersections. As the number of nodes that are loaded into the queue increases, more global memory accesses are needed which are quite expensive on GPUs. Even worse, the probability of queue overflowing will also increase as more nodes are loaded at each level. For the example shown in Fig. 9, assume two packed R-trees are constructed using the Z-order (left) and lowx ordering (right), respectively. Further assume that two queries whose MBRs are represented by the red and green rectangles, respectively, are provided. Also assume the queue capacity is 3. The BFS query processing on the R-Tree in the left of Fig. 9 will result in enqueuing 4 nodes (A_1, B_1, A_2, B_2) which leads to an overflow. On the contrary, the BFS based query processing using the R-Tree in the right part of Fig. 9 will only enqueue two entries (B_1, B_2) are and thus no overflow will occur. For DFS based query processing, more node examinations are needed when querying the low quality R-Tree (left) than the high-quality R-Tree (right). The analysis highlights the needs to construct high quality R-Trees to improve query performance and to make tradeoffs between R-Tree construction time and R-Tree query time which is left for future work.

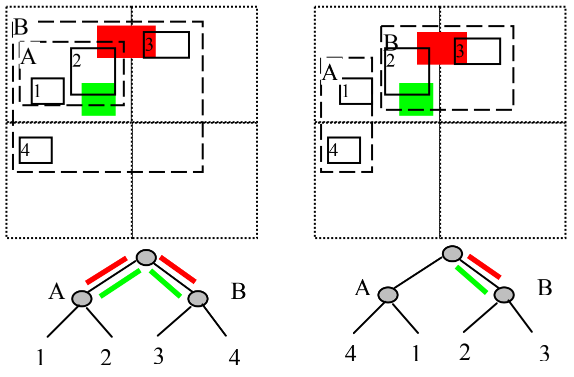

Fig. 9 Examples Showing the Impacts of R-Tree Qualities

4. Experiments

All experiments are performed on a workstation with two dual quadcore Intel E5405 processors at 2.0 GHz (8 cores in total) and a Nvidia Quadro 6000 GPU. For all experiments, -O2 flag is used for optimization. Two MBR datasets are used for evaluations. One (taxi quadrants or simply *taxi*) consists of 990,142 MBRs is derived from the quadrants of about 170 million taxi pickup locations in 2009 in NYC. The details of generating the quadrants are provided in [17]. The other dataset (pluto tax lots or simply *pluto*) has 735,488 MBRs, which comes from the NYC MapPluto[5] tax lot dataset. Because both *taxi* and *pluto* datasets are in the NYC area, experiments can be performed by using one as the query dataset the other and as the indexed dataset among the four possible combinations (*taxi-taxi*, *pluto-pluto*, *taxi-pluto*, and *pluto-taxi)*. In all experiments, the batch size *m* is empirically set to 16, i.e., 16 queries are processed within a computing block.

Table 1 Results of CPU and GPU queries

	CPU-1 (ms)	CPU-8 (ms)	GPU (ms)	Speedup
taxi-taxi	1925	290	105	18.33
pluto-taxi	833	220	130	6.41
taxi-pluto	1494	253	124	12.05
Pluto-pluto	1711	269	169	10.12

Our experiments include three groups. The first group experiments (Section 4.1) are designed to compare the performance of query processing on R-Trees using the DFS and BFS+DFS (or hybrid) approaches. We do not experiment on the BFS based query processing alone as the overflows need to be handled. The second group of experiments (Section 4.2) compares the many-core GPU based hybrid implementation with an open source serial CPU-based R-Tree implementation from [18] as well as its multicore CPU accelerated implementation based on OpenMP[6] directive based parallelization to compare the performance under different computing settings. We note that the R-Tree implementation from [18] was also used in [9] for comparison purposes. Finally, the third group experiments (Section 4.3) compare the GPU hybrid implementation with an alternative single-level grid-file based implementation. We note that while we have tested the performance of GPU based R-Tree bulk loading implementation, we have found that the query performance on bulk loaded R-Trees are much inferior to that on R-Trees using dynamic insertion on CPUs. As such, we will only use the dynamically generated R-Tree using [18] for experimenting the performance of query processing in the first two groups of experiments. The performance of queries on bulk loaded R-Trees based on two orderings are provided in the third experiment group.

4.1 Comparison between DFS and the Hybrid

Because the DFS approach has two phases, i.e., "count" and "write", we report the running times for both phases when DFS based query processing is used. The results are shown in Table 1. Clearly the hybrid BFS+DFS approach outperforms DFS approach significantly in all the four experiments.

Table 1 Runtimes of DFS and Hybrid Approaches of Four Tests

Runtimes (ms)	DFS		BFS+DFS		
	count	write	BFS	DFS count	DFS write
taxi-taxi	141	136	105	0	0
pluto-taxi	317	276	89	20	21
taxi-pluto	111	108	106	8	10
pluto-pluto	313	302	163	3	3

4.2 Comparisons between CPU and GPU Implementations

The experiment results are reported in Table 2 where "CPU-1" denotes the serial CPU implementation and "CPU-8" stands for 8-Core CPU implementation. The runtimes of the GPU implementation are based on the hybrid BFS+DFS implementation. As can be seen from Table 2, the hybrid approach has achieved 6X~18X speedups over serial CPU implementation and is about 2X faster over the 8-core CPU implementation. As one single E5405 has 820 million transistors and Fermi GPUs have about 3 billion transistors, we conclude that CPUs and GPUs have comparable per transistor performance as 3/(0.82*2)≈ 2.

Table 2 Runtimes of of CPU and GPU queries

Runtime (ms)	CPU-1	CPU-8	GPU	Speedup
taxi-taxi	1925	290	105	18.33
pluto-taxi	833	220	130	6.41
taxi-pluto	1494	253	124	12.05
pluto-pluto	1711	269	169	10.12

4.3 Comparison between R-trees and Single-Level Grid-file based Indexing on GPUs

The runtimes of query processing using the hybrid BFS+DFS approach on the dynamically generated R-Tree (column *R-tree-0*) and the single-level grid-file based indexing (column G*rid*) are shown in Table 3. Note that since the dynamic R-Tree is created on CPUs, we have included the data transfer time to the end-to-end query processing times for fair comparison. As such, the runtimes listed in the *R-tree-0* column of Table 3 are slightly larger than the runtimes listed in the *GPU* column of Table 2. From the results we can see that R-Tree based query processing is about an order (10X) times faster than the single-level grid-file which makes it attractive in many applications.

Table 3 Runtimes of Query Processing on Dynamically Generated R-Tree, Single-Level Grid-File and Two Bulk Loaded R-Trees

Runtime (ms)	R-tree-0	Grid	R-Tree-1	R-Tree-2
taxi-taxi	133	1544	5722	1768
pluto-taxi	145	1007	8453	1048
taxi-pluto	143	1420	41218	6247
pluto-pluto	185	1440	72322	18279

R-Tree-0: Dynamically Generated R-Tree on CPUs
R-Tree-1: Bulk Loaded R-tree using Z-order
R-Tree-2: Bulk Loaded R-tree using lowx ordering

To better understand the impacts of R-Tree qualities on query processing on GPUs, we have also included the runtimes of query processing on two bulk loaded R-Trees, one is based on Z-ordering and the other is based on lowx ordering where MBRs are sorted based on x1 values of MBRs defined as (x1,y1,x2,y2) tuples. From table 3, we can see that the performance of the two bulk loaded R-Trees are significantly worse than the dynamically generated R-Tree which warrants further research.

5. Conclusion and Future Work

In this study, we have implemented designs to bulk load R-Trees and query constructed R-Trees on GPUs based on different strategies. Our extensive experiments have shown that the hybrid BFS+DFS approach can achieve 6-18X speedup over a main-memory based serial CPU R-Tree implementation which makes it attractive for many real world applications. Our experiments also have shown that R-Tree qualities can have signficant impacts on query performance. The query performance of bulk loaded R-Trees on GPUs are far inferior to the serial CPU implementation which necessitates the need to seek approaches to building high-quality R-Trees. Intelligently using massively data parallel computing power for R-Tree bulk loading to generate high-quality R-Trees and support efficient query processing on GPUs is an interesting research topic.

For 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 explore their advantages and disadvantages. Another research direction is how to reorder or index the query windows for more efficient parallel query processing on GPUs. Finally, as the batched queries on R-Trees are closely related to spatial joins, we also would like to incorporate the R-tree implementations into spatial joins.

6. References

[1] Guttman, A. (1984) R-trees: a dynamic index structure for spatial searching. Proc. ACM SIGMOD Conference, 47-57.

[2] Gaede, V. and O. Gunther (1998). Multidimensional access methods. ACM Computing Surveys **30**(2): 170-231.

[3] Samet, H. (2005). Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann.

[4] Kamel, I. and C. Faloutsos (1992). Parallel R-trees. Proc. of ACM SIGMOD Conference, 195-204.

[5] Hoel, E.G. and H. Samet (1994), Performance of Data-Parallel Spatial Operations, Proc. of VLDB Conference, 156-167.

[6] Schnitzer, B. and S.T. Leutenegger (1999), Master-Client R-Trees: A New Parallel R-Tree Architecture, Proc. of SSDBM Conference, 68-77

[7] Wang, B., et al. (1999), Parallel R-Tree Search Algorithm on DSVM. Proc. of DAFSAA.

[8] Apostolos, P. and Yannis, M. (2003). Parallel bulk-loading of spatial data. Journal of Parallel Computing. 29(10): 1419-1444.

[9] Luo, L., Wong, M.D.F. and Leong, L. (2012). Parallel implementation of R-trees on the GPU. Proc. ASP-DAC

[10] Kamel, I. and Faloutsos, C. (1993) On packing R-trees. Proc. CIKM Conference, 490-499.

[11] Garcia, Y.J., M.A. Lopez, and S.T. Leutenegger (1998) A greedy algorithm for bulk loading R-trees, Proc. ACM GIS, 163-164

[12] Alborzi, H. and H. Samet (2007). Execution time analysis of a top-down R-tree construction algorithm. Information Processing letters, 101(1): p. 6-12.

[13] Zhang, J., You, S. and Gruenwald, L. (2011) Parallel Quadtree Coding of Large-Scale Raster Geospatial Data on Multicore CPUs and GPGPUs. Proc. ACM-GIS Conference.

[14] Zhang, J., You., S. and Gruenwald, (2012). U^{2}STRA: High-Performance Data Management of Ubiquitous Urban Sensing Trajectories on GPGPUs. To appear in Proc. ACM CDMW workshop. http://geoteci.engr.ccny.cuny.edu/pub/u2stra_tr.pdf

[15] Zhang, J., You., S. and Gruenwald, L (2012). High-Performance Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs. To appear in Proc. ACM DOLAP workshop. http://www-cs.ccny.cuny.edu/~jzhang/papers/aggr_tr.pdf

[16] Mokbel, M. and Aref, W. (2011). Irregularity in high-dimensional space-filling curves. Distributed and Parallel Databases 29(3): 217-238.

[17] Zhang, J. and You., S. (2012). Speeding up Large-Scale Point-in-Polygon Test Based Spatial Join on GPUs. Technical report online at http://geoteci.engr.ccny.cuny.edu/pub/pipsp_tr.pdf

[18] Open source R-Tree Implmenenation available at http://superliminal.com/sources/sources.htm.

[1] http://en.wikipedia.org/wiki/Parallel_Random_Access_Machine
[2] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-690/specifications
[3] http://en.wikipedia.org/wiki/Online_analytical_processing
[4] http://thrust.github.com/
[5] http://www.nyc.gov/html/dcp/html/bytes/applbyte.shtml
[6] http://en.wikipedia.org/wiki/OpenMP