Postgraduate Studies Programme Of The AUTH Informatics Department

"Information Systems"

Parallel Spatial Query Processing on GPUs Using R-Trees

Athanasiou Nikolaos
AEM 498

this page is intentionally left blank

1. Introduction

1.1 Object-Based Hierarchical Interior-Based Representations

A **location query** is a process where we take a location "a" as input and return the objects in which "a" is a member when using a representation that stores with each object the addresses of the cells it comprises (i.e., an explicit representation). The most natural hierarchy that can be imposed on the objects that would enable us to answer this query is one that **aggregates** every M objects (that are hopefully in close spatial proximity, although this is not a requirement) into larger objects. This process is repeated recursively until there is just one aggregated object left. Since the objects may have different sizes and shapes, it is not easy to compute and represent the aggregate object. Moreover, it is similarly difficult to test each one of them (and their aggregates) to determine if they contain "a" since each one may require a different test by virtue of the different shapes. Thus, it is useful to use a common aggregate shape and point inclusion test to prune the search.

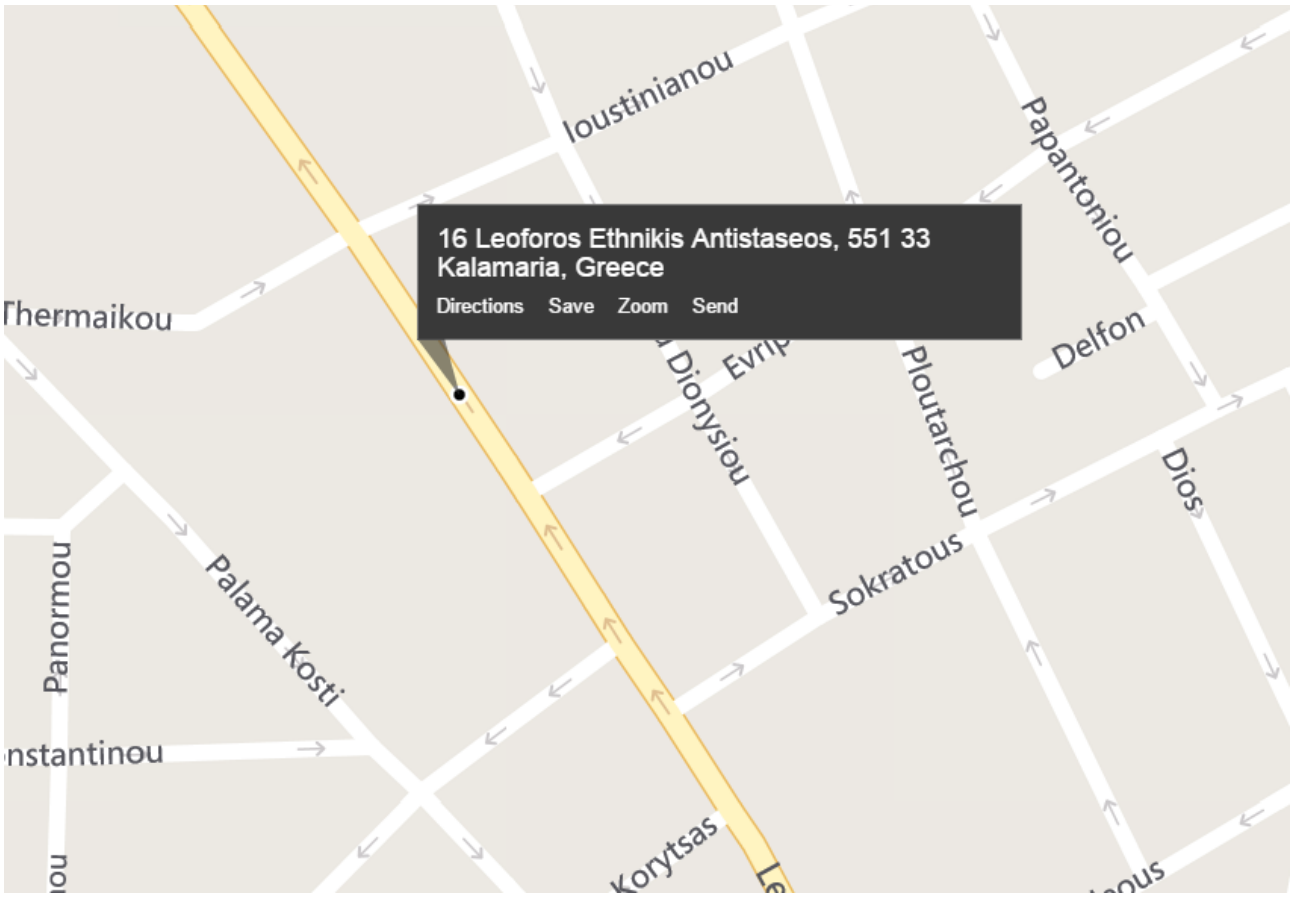


Illustration 1: example of a location query

The common aggregate shape and point inclusion test that we use assumes the existence of a minimum enclosing box (termed a bounding box) for each object. This **bounding box** is part of the data associated with each object and aggregate of objects. In this case, we reformulate our **object hierarchy** in terms of bounding boxes. In particular, we aggregate the bounding boxes of every M objects into a box (i.e., block) of minimum size that contains them. This process is repeated recursively until there is just one block left. The value associated with the bounding box b is its location (e.g., the coordinate values of its diagonally opposite corners for two-dimensional data). It should be clear that the bounding boxes serve as a filter to prune the search for an object that contains "a"and are not necessarily disjoint; in fact, the objects may be configured in space in such a way that no disjoint hierarchy is possible. By the same reasoning, the objects themselves need not be disjoint.

The process that we have just outlined can be described more formally as follows: *Assume that there are N objects in the space, and let n be the smallest power of M such that* $M^n \geq N$. *Assume that all aggregates contain M elements with the exception of the last one at each level, which may contain less than M as* M^n *is not necessarily equal to N. The hierarchy of objects consists of the set D of sets {Di}* $(0 \leq i \leq n)$, *where Dn corresponds to the set of bounding boxes of the individual objects;* D_{n-1} *corresponds to the result of the initial aggregation of the bounding boxes into N/M bounding boxes, each of which contains M bounding boxes of objects; and* D_0 *is a set containing just one element corresponding to the aggregations of all of the objects and is a bounding boxthat encloses all of the objects.*

We term the resulting hierarchy an **object pyramid**. What we have is a **multiresolution representation** as the original collection of objects is described at several levels of detail by virtue of the number of objects whose bounding boxes are grouped at each level.

1.2 Searching an object pyramid

Searching an object pyramid consisting of sets Di $(0 \leq i \leq n)$ for the object containing a particular location "a" (i.e., the location query) proceeds as follows:

We start with D_0, which consists of just one bounding box "b" and determine if "a" is inside b. If it is not, then we exit, and the answer is negative. If it is, then we examine the M elements in D_1 that are covered by "b" and repeat the test using their bounding boxes; we exit if "a" is not covered by at least one of the bounding boxes at this level. This process is applied recursively to all elements of Dj for $0 \leq j \leq n$ until all elements of D_n have been processed, at which time the process stops. The advantage of this method is that elements of D_j $(1 \leq j \leq n)$ are not examined unless "a" is guaranteed to be covered by at least one of the elements of D_{j-1}.

The bounding boxes serve to **distinguish between occupied and unoccupied space**, thereby indicating whether the search for the objects that contain a particular location (i.e., the location query) should proceed further. At a first glance, it would appear that the object pyramid is rather inefficient for responding to the location query because, in the worst case, all of the bounding boxes at all levels must be examined. However, the

maximum number of bounding boxes in the object pyramid, and hence the maximum number that will have to be inspected, is $\sum_{j=0}^{n} M^j \leq 2\mathrm{N}$.

Of course, we may also have to examine the actual sets of locations associated with each object when the bounding box does not result in any of the objects being pruned from further consideration since the objects are not necessarily rectangular in shape (i.e., boxes). Thus, using the hierarchy provided by the object pyramid results in at most an additional factor of 2 in terms of the number of bounding box tests while possibly saving many more tests. Therefore, the maximum amount of work to answer the location query with the hierarchy is of the same order of magnitude as that which would have been needed had the hierarchy not been introduced.

1.3 The object tree pyramid

As we can see, the way in which we introduced the hierarchy to form the object pyramid did not necessarily enable us to make more efficient use of the explicit interior based representation to respond to the location query. The problem was that once we determined that location a was covered by one of the bounding boxes, say b, in D_j , $(0 \leq j \leq n - 1)$, we had no way to access the bounding boxes making up b without examining all of the bounding boxes in D_{j+1}. This is easy to rectify by imposing an access structure in the form of a tree T on the elements of the hierarchy D. One possible implementation is a tree of fanout M, where the root T_0 corresponds to the bounding box in D_0. T_0 has M links to its M children $\{ T_{1k} \}$, which correspond to the M bounding boxes in D_1 that D_0 comprises. The set of nodes $\{ T_{ik} \}$ at depth i correspond to the bounding boxes in D_i $(0 \leq i \leq n)$, while the set of leaf nodes $\{ T_{nk} \}$ correspond to D_n. In particular, node t in the tree at depth j corresponds to bounding box b in D_j $(0 \leq j \leq n - 1)$, and t contains M pointers to its M children in T_{j+1} corresponding to the bounding boxes in Dj+1 that are contained in b. We use the term **object-tree pyramid** to describe this structure.

The object-tree pyramid that we have just described still has a worst case where we may have to examine all of the bounding boxes in D_j $(1 \leq j \leq n)$ when executing the location query or its variants (e.g., a window query). This is the case if query location a is contained in every bounding box in D_{j-1}. Such a situation, although rare, can arise in practice because "a" may be included in the bounding boxes of many objects, termed a **false hit,** as the bounding boxes are not disjoint, while a is contained in a much smaller number of objects. Equivalently, false hits are caused by the fact that a spatial object may be spatially contained in full or in part in several bounding boxes or nodes while being associated with just one node or bounding box.

However, unlike the object pyramid, the object-tree pyramid does guarantee that only the bounding boxes that contain a, and no others, will be examined. Thus, we have not improved on the worst case of the object pyramid although we have reduced its likelihood, because we may still have to examine 2N bounding boxes. It is interesting to observe that the object pyramid and the object-tree pyramid are instances of an **explicit interior-based**

representation since it is still the case that associated with each object "o" is a set containing the addresses of the cells that it comprises. Note also that the access structure facilitates only the determination of the object associated with a particular cell and not which cells are contiguous. Thus, the object-tree pyramid is not an instance of an implicit interior-based representation.

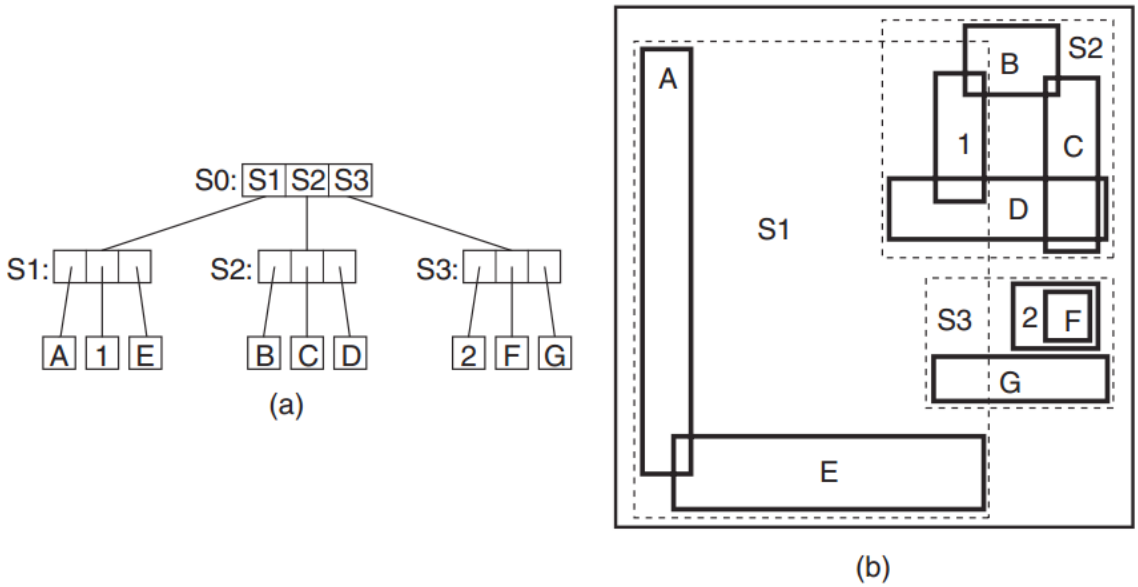


Illustration 2: Object tree pyramid and its spatial extends

1.4 Efficiency factors of the object tree pyramid

The decision as to which objects to aggregate is an important factor in the efficiency of the object-tree pyramid in responding to the location query. The efficiency of the object-tree pyramid for search operations depends on its abilities to distinguish between occupied space and unoccupied space and to prevent a node from being examined needlessly due to a false overlap with other nodes.

The extent to which these efficiencies are realized is a direct result of how well our aggregation policy is able to satisfy the following two goals. The first goal is to minimize the number of aggregated nodes that must be visited by the search. This goal is accomplished by minimizing the area common to sibling aggregated nodes (termed **overlap**). The second goal is to reduce the likelihood that sibling aggregated nodes are visited by the search. This is accomplished by minimizing the total area spanned by the bounding boxes of the sibling aggregated nodes (termed **coverage**). A related goal to that of minimizing the coverage is minimizing the area in sibling aggregated nodes that is not spanned by the bounding boxes of any of the children of the sibling aggregated nodes (termed **dead area**). Of course, at times, these goals may be contradictory.

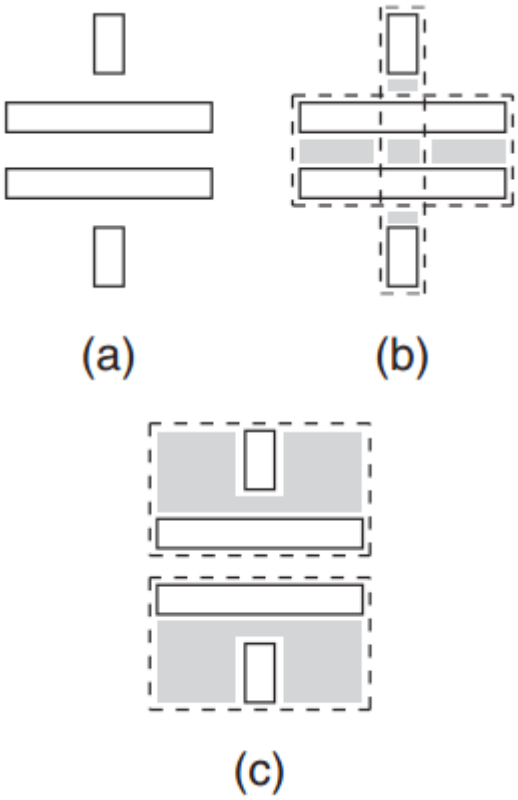


Illustration 3: (a) Four BBs (b) Min coverage (c) Min overlap (dead area in grey)

2. Aggregation Techniques

2.1 Ordering based aggregation techniques

The most frequently used ordering technique is based on mapping the bounding boxes of the objects to a representative point in a lower, the same, or a higher-dimensional space and then applying one of the space-ordering methods described here. We use the term **object number** to refer to the result of the application of space ordering. For example, twodimensional rectangle objects can be transformed into one of the following representative points

1. The centroid
2. The centroid and the horizontal and vertical extends
3. The x and y coordinate values of the two diagonally opposite corners of the rectangle
4. The (x,y) pair or the lower right corner of the rectangle and its height and width

Consider for example the following collection of rectangle objects in illustration 4, which are reduced to their centroid. The numbers associated with the rectangles denote the relative times at which they were created.

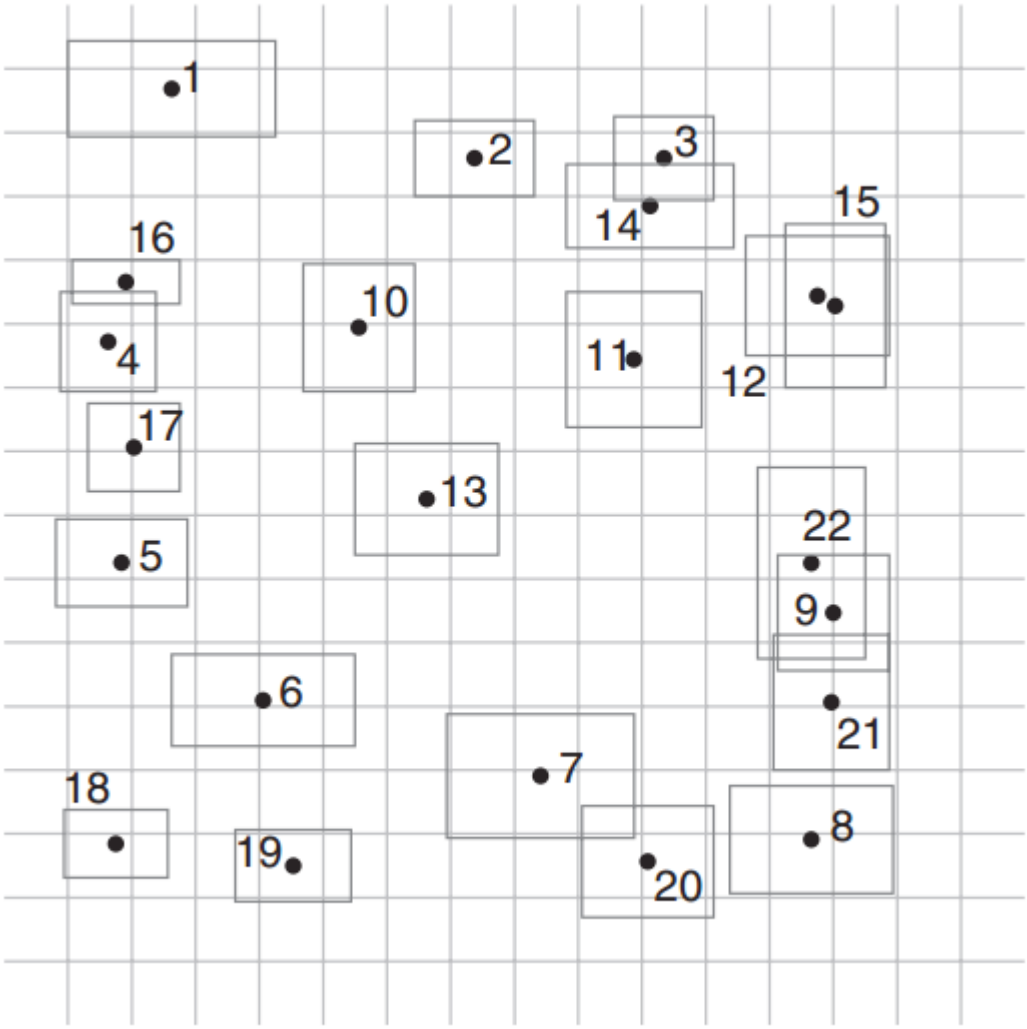


Illustration 4: A collection of rectangle objects

Illustration 5 shows the results of applying a Morton order (a) and a Peano Hilbert order (b) to the above collection.

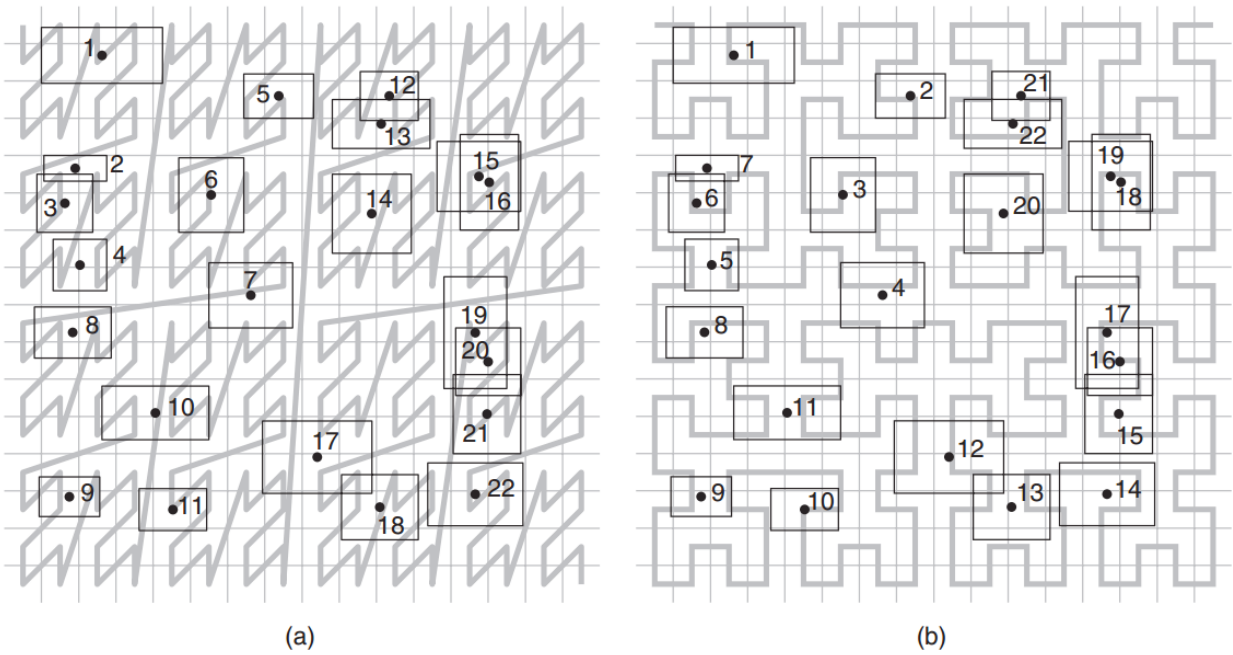


Illustration 5: Space ordering methods : (a) Morton (b) Peano Hilbert

Once the N objects have been ordered, the hierarchy D is built in the order D_n, D_{n-1}, ..., D_1, D_0, where n is the smallest power of M such that $M^n \geq N$. D_n consists of the set of original objects and their bounding boxes. There are two ways of grouping the items to form the hierarchy D: one-dimensional and multidimensional grouping. In the **one-dimensional grouping method**, D_{n-1} is formed as follows:

The first M objects and their corresponding bounding boxes form the first aggregate, the second M objects and their corresponding bounding boxes form the second aggregate, and so on. D_{n-2} is formed by applying this aggregation process again to the set D_{n-1} of N/M objects and their bounding boxes. This process is continued recursively until we obtain the set D_0 containing just one element corresponding to a bounding box that encloses all of the objects. Note, however, that when the process is continued recursively, the elements of the sets D_i $(0 \leq i \leq n-1)$ are not necessarily ordered in the same manner as the elements of D_n.

There are several implementations of the object-tree pyramid using the one-dimensional grouping methods. For example, the **Hilbert packed R-tree** [11] is an object-tree pyramid that makes use of a Peano-Hilbert order. It is important to note that only the leaf nodes of the Hilbert packed R-tree are ordered using the Peano-Hilbert order. The nodes at the remaining levels are ordered according to the time at which they were created.

A slightly different approach is employed in the **packed R-tree** [12], which is another instance of an object-tree pyramid. The packed R-tree is based on ordering the objects on the basis of some criterion, such as increasing value of the x coordinate or any of the space-ordering methods shown in Figure 5. Once this order has been obtained, the leaf nodes in the packed R-tree are filled by examining the objects in increasing order, where each leaf node is filled with the first unprocessed object and its M − 1 nearest neighbors that have not yet been inserted into other leaf nodes. Once an entire level of the packed R-tree has been obtained, the algorithm is reapplied to add nodes at the next level using the

same nearest neighbor criterion, terminating when a level contains just one node. The only difference between the ordering that is applied at the levels containing the nonleaf nodes from that used at the level of the leaf nodes is that, in the former case we are ordering the bounding boxes while, in the latter case, we are ordering the actual objects.

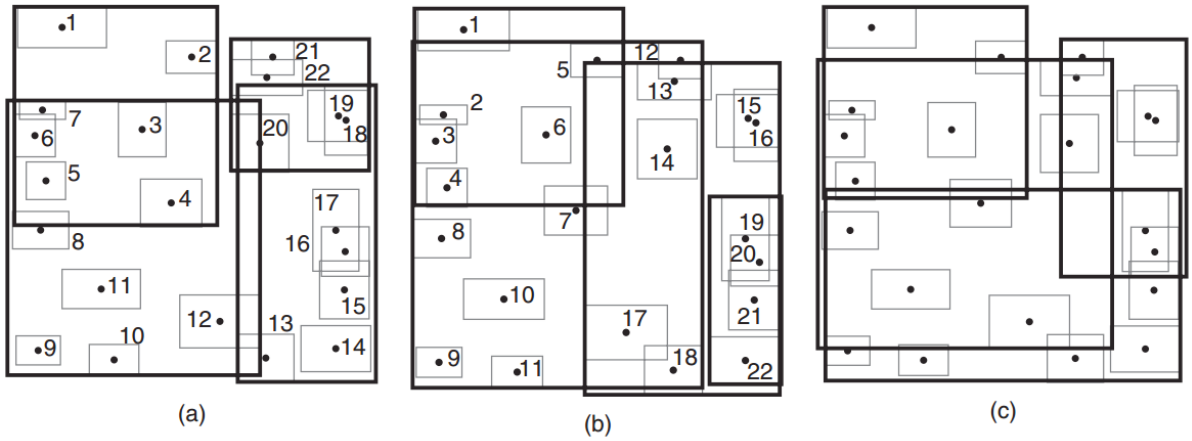


Illustration 6: The bounding boxes corresponding to the first level of aggregation for the (a) Hilbertpacked R-tree, (b) Morton packed R-tree, and (c) packed R-tree (using a Peano-Hilbertorder for the initial ordering) for the collection of 22 rectangle objects M = 6

2.1.2 Multidimensional grouping – The STR method

The **sort-tile-recurse** method (STR method) of Leutenegger, Lopez, and Edgington [7] is an example of the multidimensional grouping method. It is assumed, without loss of generality, that the underlying space is two-dimensional, although the extension of the method to higher dimensions is straightforward.

Assuming a total of N rectangles and a node capacity of M rectangles per leaf node, D_{n-1} is formed by constructing a tiling of the underlying space consisting of s vertical slabs, where each slab contains s tiles. Each tile corresponds to an object-tree pyramid leaf node that is filled to capacity. Note that the result of this process is that the underlying space is being tiled with rectangular tiles, thereby resembling a grid, but, most importantly, unlike a grid, the horizontal edges of horizontally adjacent tiles (i.e., with a common vertical edge) **do not form a straight line** (i.e., are not connected). Using this process means that the underlying space is tiled with approximately sqrt(N/M) × sqrt(N/M) tiles and results in approximately N/M object-tree pyramid leaf nodes. The tiling process is applied recursively to these N/M tiles to form D_{n-2}, D_{n-3},... until just one node is obtained.

The STR method builds the object-tree pyramid in a **bottom-up manner**. The actual mechanics of the STR method are as follows:

Sort the rectangles on the basis of one coordinate value of some easily identified point that is associated with them, say the x coordinate value of their centroid. Aggregate the sorted rectangles into sqrt(N/M) groups of sqrt(NM) rectangles, each of which forms a ***vertical***

slab *containing all rectangles whose centroid's x coordinate value lies in the slab. Next, for each vertical slab v, sort all rectangles in v on the basis of their centroid's y coordinate value. Aggregate the sqrt(NM) sorted rectangles in each slab v into sqrt(N/M) groups of M rectangles each. Recall that the elements of these groups form the leaf nodes of the object-tree pyramid. Notice that the minimum bounding boxes of the rectangles in each tile are usually larger than the tiles. The process of forming a gridlike tiling is now applied recursively to the N/M minimum bounding boxes of the tiles, with N taking on the value of N/M until the number of tiles is no larger than M, in which case all of the tiles fit in the root node, and we are done.*

A couple of items are worthy of further note. First, the minimum bounding boxes of the rectangles in each tile are usually larger than the tiles. This means that **the tiles at each level will overlap**. Thus, we do not have a true grid in the sense that the elements at each level of the object-tree pyramid are usually not disjoint. Second, the ordering that is applied is quite similar to a row order (actually column order to be precise), where the x coordinate value serves as a primary key to form the vertical slabs while the y coordinate value serves as the secondary key to form the tiles from the vertical slabs.

Notice that the STR method is a **bottom-up technique**. However, the same idea could also be applied in a top-down manner so that we originally start with M tiles that are then further partitioned. In other words, we start with sqrt(M) vertical slabs containing sqrt(M) tiles apiece. The disadvantage of the top-down method is that it requires that we make roughly 2logMN passes over all of the data, whereas the bottom-up method has the advantage of making just two passes over the data (one for the x coordinate value and one for the y coordinate value) since all recursive invocations of the algorithm deal with centroids of the tiles. Regardless of how the objects are aggregated, the object-tree pyramid is analogous to a height-balanced M-ary tree where only the leaf nodes contain data (objects in this case), and all of the leaf nodes are at the same level.

2.2 Extend based aggregation techniques – the R-Tree

When the objects are to be aggregated on the basis of their extent (i.e., the space occupied by their bounding boxes), then good dynamic behavior is achieved by making use of an **R-tree** [1]. **An R-tree is a generalization of the object-tree pyramid where, for an order (m,M) R-tree, the number of objects or bounding boxes that are aggregated in each node is permitted to range between m ≤ M/2 and M.** On the other hand, it is always M for the object-tree pyramid. The root node in an R-tree has at least two entries unless it is a leaf node, in which case it has just one entry corresponding to the bounding box of an object. **The R-tree is usually built as the objects are encountered** rather than after all objects have been input. Of the different variations on the object-tree pyramid that we have presented, the R-tree is the one used most frequently, especially in database applications. Note that the R-tree is not unique. Its structure depends heavily on the order in which the individual objects were inserted into (and possibly deleted from) the tree.

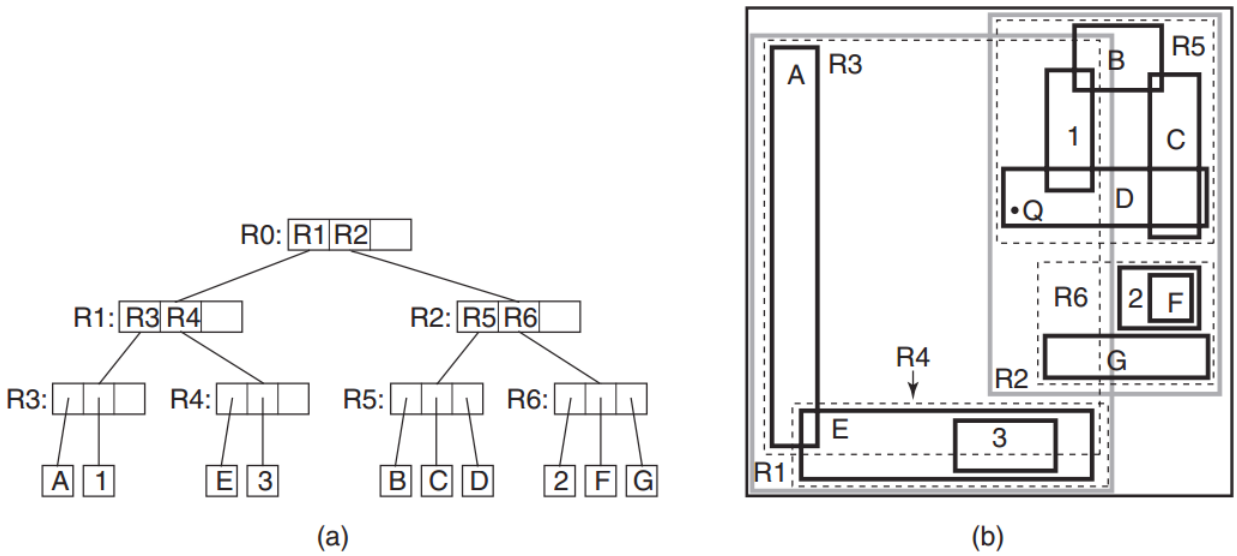


Illustration 7: example rTree (a) data structure (b) spatial extends

Given that each R-tree node can contain a varying number of objects or bounding boxes, it is not surprising that the R-tree was inspired by the B-tree. This means that nodes are viewed as analogous to disk pages. Thus, the parameters defining the tree (i.e., m and M) are chosen so that a small number of nodes is visited during a spatial query (i.e., variants of the location query), which means that m and M are usually quite large.

The need to minimize the number of disk accesses also **affects the format of each Rtree node**. Recall that in the definition of the object-tree pyramid, each node p contains M pointers to p's children and one bounding box corresponding to the union of the bounding boxes of p's children. This means that in order to decide which of node p's children should be descended, we must access the nodes corresponding to these children to perform the point-inclusion test. Each such access requires a disk I/O operation. In order to avoid these disk I/O operations, the format of R-tree node p is modified so that p contains k ($m \leq k \leq M$) pointers to p's children and the k bounding boxes of p's children, instead of containing just one bounding box corresponding to the union of the bounding boxes of p's children as is the case for the object-tree pyramid. Once again, we observe that the k point inclusion tests do not require any disk I/O operations at the cost of being able to aggregate a smaller number of objects in each node since m and M are now smaller, assuming that the page size is fixed.

As long as the number of objects in each R-tree leaf node is between m and M, no action needs to be taken on the R-tree structure other than adjusting the bounding boxes when inserting or deleting an object. If the number of objects in a leaf node decreases below m, then the node is said to **underflow**. In this case, the objects in the underflowing nodes must be reinserted, and bounding boxes in nonleaf nodes must be adjusted. If these nonleaf nodes also underflow, then the objects in their leaf nodes must also be reinserted. If the number of objects in a leaf node increases above M, then the node is said to **overflow**. In this case, it must be split, and the M + 1 objects that it contains must be distributed in the two resulting nodes. Splits are propagated up the tree.

Underflows in an R-tree are handled in an analogous manner to the way they are dealt

with in a B-tree. In contrast, the overflow situation points out a significant difference between an R-tree and a B-tree. In a B-tree, we usually do not have a choice as to the node p that is to contain t since the tree is ordered. Thus, once we determine that p is full, we must either split p or apply a rotation (also known as deferred splitting) process. On the other hand, **in an R-tree, we can insert t into any node p, as long as p is not full**. However, once t is inserted into p, we must expand the bounding box associated with p to include the space spanned by the bounding box b of t. Of course, we can also insert t in a full node p, in which case we must also split p.

The need to expand the bounding box of p has an effect on the future performance of the R-tree, and thus we must make a wise choice with respect to p. As in the case of the object-tree pyramid, the efficiency of the R-tree for search operations depends on its abilities to distinguish between occupied space and unoccupied space and to prevent a node from being examined needlessly due to a false overlap with other nodes. Again, as in the object-tree pyramid, the extent to which these efficiencies are realized is a direct result of how well we are able to satisfy our goals of minimizing **coverage** and **overlap**. These goals guide the initial R-tree creation process and are subject to the previously mentioned constraint that the R-tree is usually built as the objects are encountered rather than after all objects have been input.

3. R-Tree Construction : bulk insertion and bulk loading

At times, it is desired to update an existing object-tree pyramid with a large number of objects at once. Performing these updates one object at a time using the implementations of the dynamic methods described above can be expensive. The CPU and I/O costs can be lowered by grouping the input objects prior to the insertion. This technique is known as bulk insertion. It can also be used to build the object-tree pyramid from scratch, in which case it is also known as bulk loading.

A simple **bulk insertion** idea is to sort all of the m new objects to be inserted according to some order (e.g., Peano-Hilbert) and then insert them into an existing object-tree pyramid in this order. The rationale for sorting the new objects is to have each new object be relatively close to the previously inserted object so that, most of the time, the nodes on the insertion path are likely to be the same, which is even more likely to be the case if some caching mechanism is employed. Thus, the total number of I/O operations is reduced. This technique works fine when the number of objects being inserted is small relative to the total number of objects. Also, it may be the best choice when the collection of new objects is spread over a relatively large portion of the underlying space, as, in such cases, the use of other methods may lead to excessive overlap. It can be used with any of the methods of building an object-tree pyramid.

The **bulk loading** methods that we describe [14, 13] are quite different from the bulk insertion methods described above as the individual objects are inserted using the dynamic insertion rules. In particular, the objects are not preprocessed (e.g., via an explicit sorting step or aggregation into a distinct object-tree pyramid) prior to insertion as is the case for

the bulk insertion methods. In particular, the sorting can be viewed as a variant of a lazy evaluation method, where the sorting is deferred as much as possible. Nevertheless, at the end of the bulk loading process, the data is ordered on the basis of the underlying tree structure and hence can be considered sorted. These methods are general in that they are applicable to most balanced tree data structures that resemble a B-tree, including a large class of multidimensional index structures, such as the R-tree. They are based on the general concept of the buffer tree of Arge [15], in which case each internal node of the buffer tree structure contains a buffer of records stored on disk. A main memory buffer is employed when transferring records from the buffer of a node to the buffers of its child nodes.

In simpler terms, the process works as follows. We keep inserting into the buffer of the root node until its buffer is full. At this time, we distribute its content, as described above, and possibly reapply the process to the children if they are full, and so on, until encountering leaf nodes, at which time a node split is needed if the nodes contain more entries than the block size. Once all full buffers have been emptied, the root's buffer is filled with more data, and the insertion process is repeated. Once all of the objects in the dataset have been processed, a pass is made over the buffer associated with the root node, and the remaining entries are inserted into the appropriate children of the tree rooted at the node. The same process is applied to the buffers associated with the children of the root and their descendants so that all nonempty buffers in the tree have been emptied. At this point, the leaf nodes of the object-tree pyramid have been constructed. Next, apply the same building process to the minimum bounding boxes of the leaf nodes, thereby constructing the level of the object tree pyramid corresponding to the parents of the leaf nodes. This process is continued until the number of leaf nodes is less than the maximum capacity of an object-tree pyramid leaf node.

4. R-Trees on GPUs

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 the study by You et al [2] the potentials of accelerating both R-Tree bulk loading and spatial window query processing on GPUs using R-Trees are evaluated. In addition to designing an efficient data layout schema for R-Trees on GPUs, they implement 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.

4. 1 Node Layout

Simple **linear array based data structures** are used 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 this 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. 8

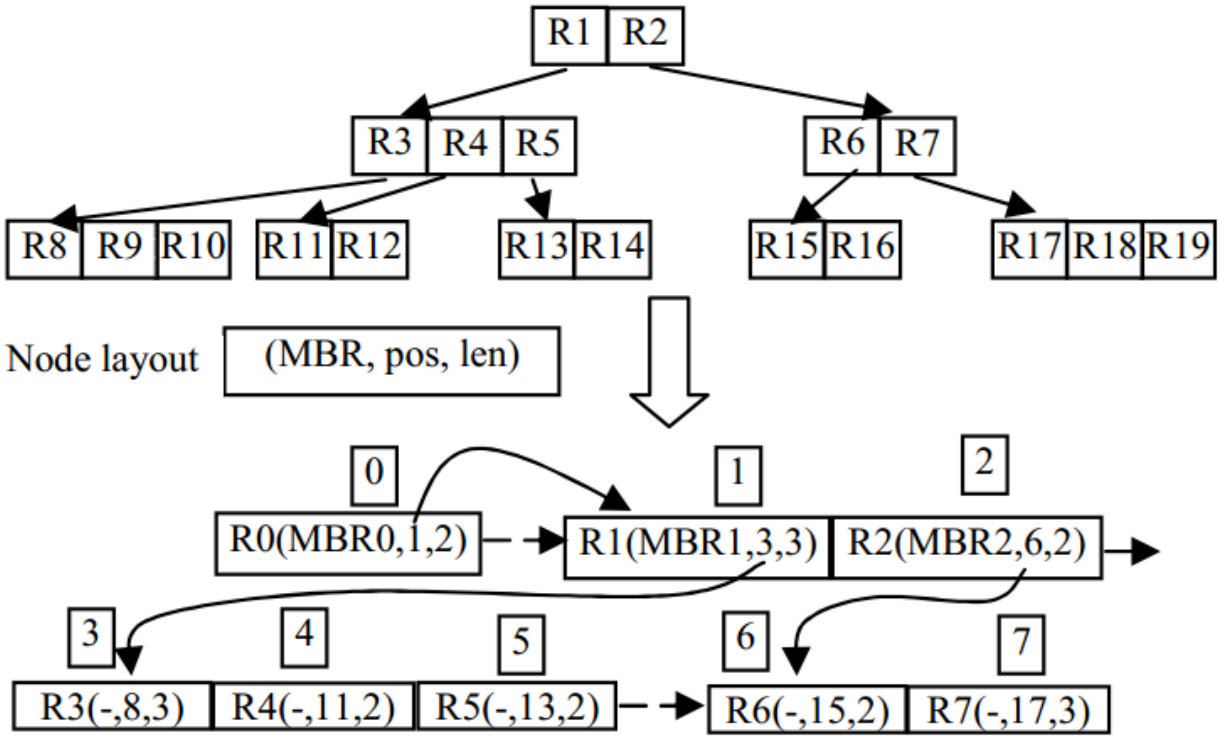


Illustration 8: Node layout of linearized R-Tree

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.

4. 2 Parallel Bulk Loading on GPUs

In the study at hand, both low-x packing [6] and STR packing [7] are implemented for bulk loading R-Trees. Instead of using native GPU programming languages (such as Nvidia CUDA3) directly, the implementations are built on top of several parallel primitives provided by the Thrust library [5].

To explain **low-x packing** two concepts must be presented :

- **reduce** : reduce is a generalization of summation. it computes the sum (or some other binary operation) of all the elements in the range [first, last). Reduce is similar to the C++ Standard Template Library's std::accumulate. The primary difference between the two functions is that std::accumulate guarantees the order of summation, while reduce requires associativity of the binary operation to parallelize the reduction.
- **reduce_by_key** is a generalization of reduce to key-value pairs. For each group of consecutive keys in the range [keys_first, keys_last) that are equal, reduce_by_key copies the first element of the group to the keys_output. The corresponding values in the range are reduced using the plus and the result copied to values_output.

The low-x packing approach is the parallel counterpart of one-dimensional grouping using low-x ordering. 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. 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 bottom-up using a reduce_by_key primitive. 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 (ie belonging to the same node), 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.

The parallel version of the **Sort-Tile-Recurse** R-Tree bulk loading algorithm on GPUs differs from the sequential one, in the application of a sort_by_key parallel primitive in the initial sorting step and the parallel sorting of the vertical slabs produced. To implement the second sort where each slice is sorted individually, an auxiliary array is used to identify items that belong to the same slice. This is achieved by assigning the same unique identifier for all items that belong to the same slice. 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.

5. GPU based R-Tree batched query

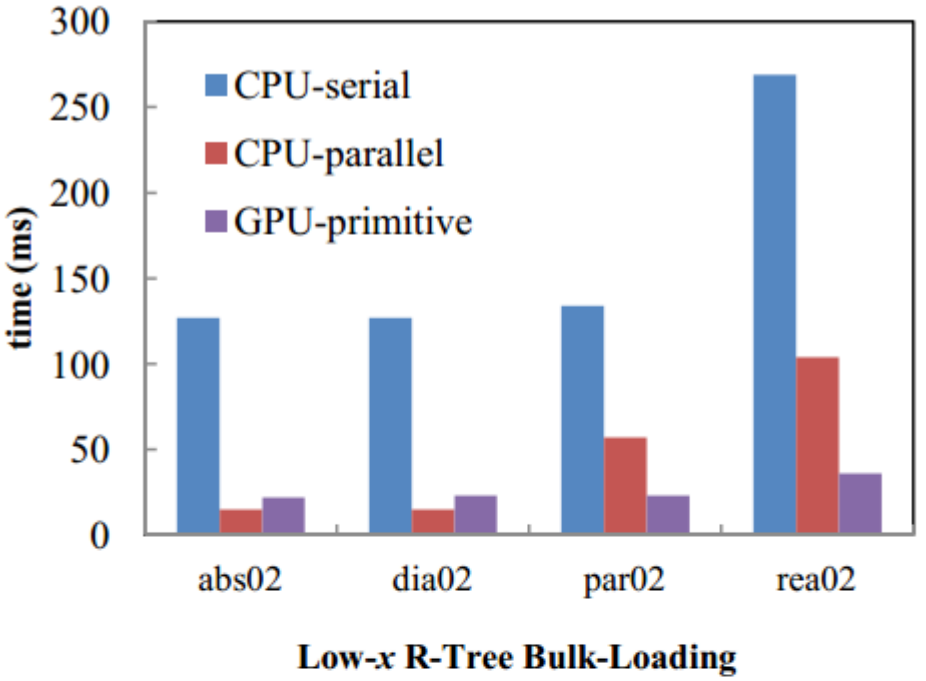
Instead of accelerating a single query, the goal of a parallel implementation as in [2] is to support efficient **batched query processing** on the GPU in parallel. To leverage massively parallel processing power of GPUs, we need to balance workload among all parallel processing units while minimizing expensive global memory operations on GPUs.

In the DFS approach, each thread processes a query in a **Depth-FirstSearch** (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 written but the global memory accesses are not coalesced in the naive implementation. To improve the performance, perblock shared memory is utilized for the stack structure instead. While it is well known that GPU shared memory is usually limited for many applications, 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.

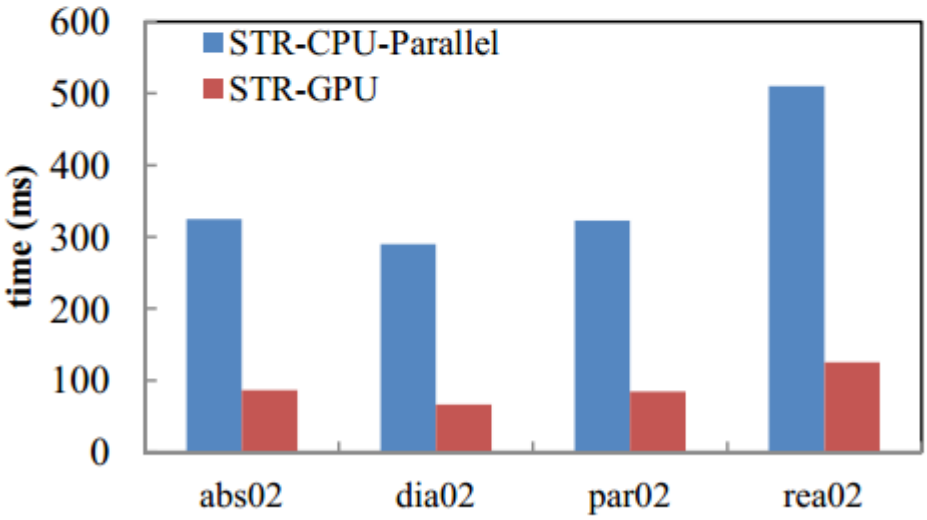
As an improvement to the DFS-based spatial window query processing technique, the **BFS-based** technique is developed to balance the workload within a GPU computing block. A queue is maintained for all the threads inside a computing block to process all the batched queries assigned to a computing block. 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.

6. Experiments and evaluation

All experiments are performed on a workstation with two Intel E5405 processors at 2.0GHz (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 [9]. As the baseline for CPU-based dynamic R-Tree implementation, [8] was used.



Low-*x* R-Tree Bulk-Loading



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

Illustration 9

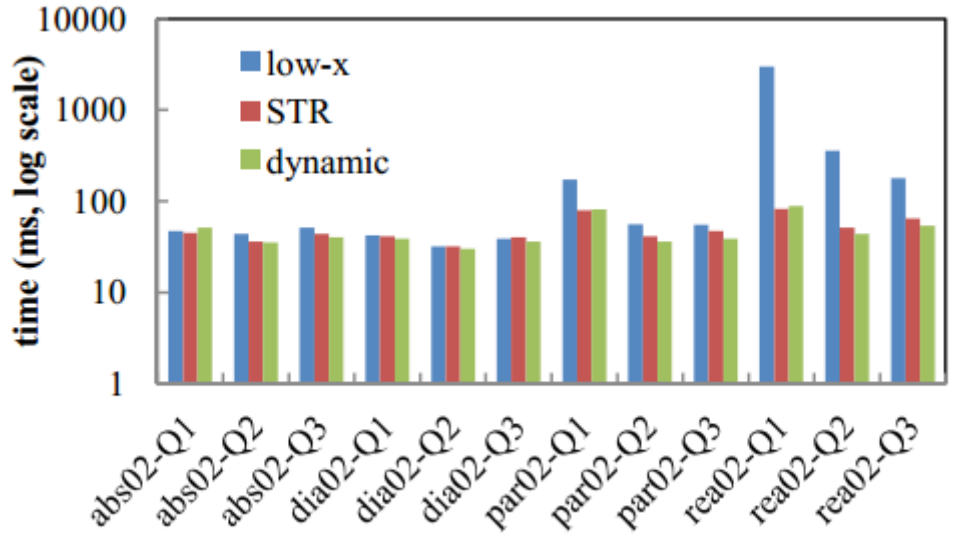


Illustration 10: query performance on differenct R-Trees

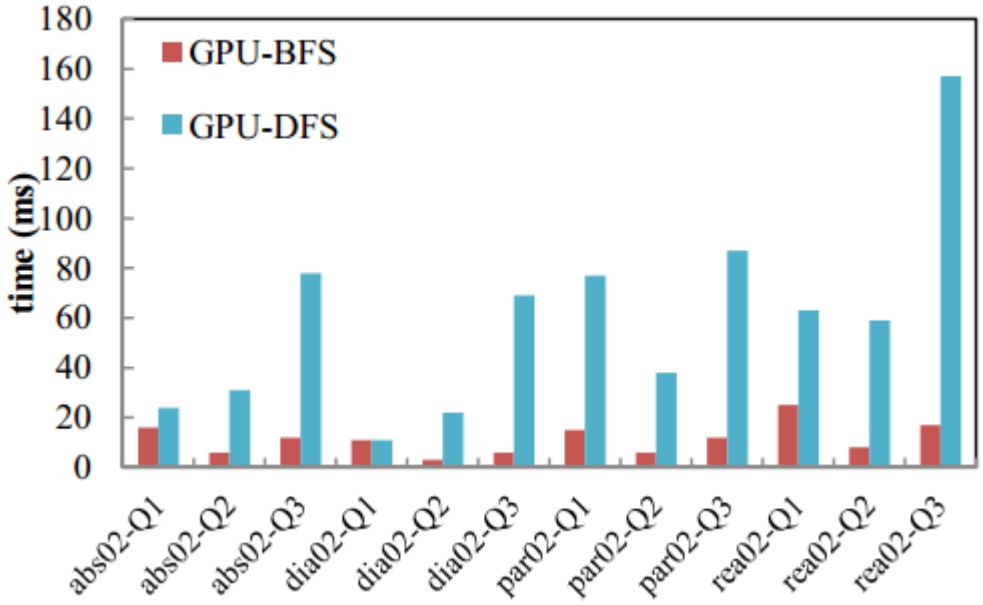


Illustration 11: Performance comparison between BFS and DFS based query processing

7. Conclusions

In the study at hand [2], parallel designs were implemented for R-Tree representation, bulk loading and query processing on GPUs. Extensive experiments have shown that the GPU parallel query implementations can provide great speed ups over multi-core CPU based implementations. It has also been 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 [10] 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. As future work plan, in addition to further investigating on GPU based bulk loading that have been discussed inline, it's also planned to compare R-Tree based indexing approaches with quadtree based ones on GPUs to further explore their respective advantages and disadvantages.

8. REFERENCES

1. **Antonin Guttman** – R-Trees, A dynamic index structure for spatial searching
2. **Simin You, Jianting Zhang, Le Gruenwald** - Parallel Spatial Query Processing on GPUs Using R-Trees
3. **Hanan_Samet** - Foundations of Multidimensional and metric data structures
4. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html
5. https://developer.nvidia.com/thrust
6. **Luo, L.** et al. 2012 - Parallel implementation of R-trees on the GPU
7. **Leutenegger,** S.T. et al. 1997 - STR: a simple and efficient algorithm for R-tree packing
8. http://superliminal.com/sources/sources.htm
9. A Benchmark for Multidimensional Index Structures:

   http://www.mathematik.uni-marburg.de/~rstar/benchmark/
10. **Hoel, E.G. and Samet, H.** - Performance of DataParallel Spatial Operations
11. **I. Kamel and C. Faloutsos** - On packing R-trees.
12. **N. Roussopoulos and D. Leifker** - Direct spatial search on pictorial databases using packed R-trees.
13. **L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter** - Efficient bulk operations on dynamic R-trees
14. **J. van den Bercken, B. Seeger, and P. Widmayer** - A generic approach to bulk loading multidimensional index structures.
15. **L. Arge** - Efficient external-memory data structures and applications - PhD thesis