PAPER

Accelerating Range Query Processing on R-Tree Using Graphics Processing Units

Boseon YU[†], Hyunduk KIM[†], *Nonmembers*, Wonik CHOI^{†a)}, and Dongseop KWON^{††}, *Members*

SUMMARY Recently, various research efforts have been conducted to develop strategies for accelerating multi-dimensional query processing using the graphics processing units (GPUs). However, well-known multi-dimensional access methods such as the R-tree, B-tree, and their variants are hardly applicable to GPUs in practice, mainly due to the characteristics of a hierarchical index structure. More specifically, the hierarchical structure not only causes frequent transfers of small volumes of data but also provides limited opportunity to exploit the advanced data parallelism of GPUs. To address these problems, we propose an approach that uses GPUs as a buffer. The main idea is that object entries in recently visited leaf nodes are buffered in the global memory of GPUs and processed by massive parallel threads of the GPUs. Through extensive performance studies, we observed that the proposed approach achieved query performance up to five times higher than that of the original R-tree.

key words: database, multi-dimensional index structure, GPUs

1. Introduction

Since Guttman's paper [1] in 1984, there have been numerous research efforts to improve the range query performance of the R-tree for vast and rapidly growing amounts of data in multi-dimensional spaces. Index structures commonly used in various areas such as geographical information systems, CAD, spatial databases and Web information systems include the R-tree [1], grid file [5] and variants such as the R⁺-tree [2], R*-tree [3], Quad-tree [4], Multi-R tree [30] and HG-tree [31]. In particular, the R-tree index structure is one of the most widely used structures in spatial database systems due to its efficiency and simplicity. However, there has been growing interest in accelerating the query performance because R-tree query performance degrades drastically as the volume of data grows. To address this problem, many approaches have been proposed. The buffer management systems have been suggested to reduce disk I/O [6], [7]. Since query operations on multi-dimensional index structures are composed of many disk access operations and few computations, reducing disk access has a critical impact on the overall query performance. With the advent of multicore processors, parallelizing index structures have been studied widely to improve the query performances. As proposed in many studies [10]–[16], [32], the recent trend in parallelization is to exploit the SIMD (Single Instruction Multiple Data) architecture, on which considerable interest has recently been focused due to its low cost, massive data

parallelism, high memory bandwidth and improved general purpose programming interface [8]. The SIMD architecture enables parallel execution of the same instructions on hundreds or thousands of data for simple and intuitive parallelization. In this paper, GPUs that have massively parallel SIMD architecture are used to implement the proposed method.

However, direct exploitation of GPUs is not suitable for disk-based hierarchical index structures due to the structural characteristics of both GPUs and index algorithms. More specifically, query processing for the structures needs to traverse the trees in a depth-first order. This reflects the fact that high data-dependency exists between two result sets from branch tests at each level. Furthermore, the limited capacity of a node makes parallelization of query processing more difficult since it leads to frequent memory transfers between the GPU memory (device memory) and the CPU memory (host memory). To address these problems, we propose a novel approach that uses GPUs as a buffer. The difficulties encountered in utilizing GPUs to accelerate hierarchical index structures are discussed in more detail in Sect. 2. The main idea behind the proposed method is that every leaf node recently visited is buffered as much as possible in the global memory of GPUs so that the range query processing can benefit from the massive resources of the GPUs. Note that this paper is an extended version of [35]. The main contributions of our work are as follows:

- To the best of our knowledge, this work is the first attempt to accelerate the performance of range query on R-tree by using GPUs as a buffer.
- We propose a novel approach that uses the global memory of GPUs as a buffer to efficiently accelerate the range query processing of the hierarchical index structures. It is also noteworthy that our approach is orthogonal to existing hierarchical index structures, which suggests that all existing algorithms can benefit from our proposed approach during their own query processing.
- With a minimum of additional changes in the R-tree, we achieved query performance up to 5 times higher than that of the original R-tree.

The rest of this paper is organized as follows. In Sect. 2, we describe utilizing GPUs for accelerating query performances on hierarchical index structures. We briefly describe overheads caused by the use of GPUs in Sect. 3. Section 4 reviews the related work on existing GPU-based index

Manuscript received April 17, 2013.

Manuscript revised August 13, 2013.

[†]The authors are with Inha University, Incheon, Korea.

^{††}The author is with Myongji University, Yongin, Korea.

a) E-mail: wchoi@inha.ac.kr

DOI: 10.1587/transinf.E96.D.2776

structures and on buffering mechanisms. We introduce our approach for an efficient adaptation of GPUs on range query processing on R-tree in Sect. 5. Section 6 presents buffer updating strategies. In Sect. 7, we evaluate our approach by comparing it against the original R-tree. Finally, Sect. 8 concludes our work.

2. Utilization of GPUs for Hierarchical Index Structures

Hierarchical index structures such as R-trees or B-trees are inadequate for integration with GPUs due to the nature of both the hierarchical index structures and the GPUs. Nodes in index structures usually contain a limited number of object keys (e.g., 200-300) to fit those into a disk page (e.g., 4KB), whereas GPUs are based on a massively parallel SIMD architecture and thus not considering the organization of disk. More specifically, it is recommended that a GPU thread block contains as many threads as possible to replace threads stalled by memory access operations. A thread block usually has 256 or 512 threads to hide memory access latency, and GPUs are capable of running hundreds of thread blocks. Therefore, the limited number of objects in a node affords little opportunity to exploit the parallel computing capability of the GPUs. Moreover, the fanout tree structure leads to frequent transfers of small volume of memory, which is one of the most critical overheads from utilizing GPUs. In addition, complex data and control dependencies between two levels make the utilization of GPUs for hierarchical index structures more difficult. In searching the trees, multiple branch tests must be performed in a depth-first order. This reflects the fact that the results of branch tests at the previous level affect branch tests at the next level. This characteristic makes it difficult to gain the maximum benefit from the massively parallel SIMD architecture of GPUs.

Nevertheless, accelerating the query processing for hierarchical index structures using GPUs is valuable and necessary, considering that R-trees and B-trees are index structures widely used in modern database systems, and still need improvement for their practical applications in various fields.

3. GPU Overhead

GPUs are promising hardware platforms for data parallelism, mainly due to the excellent parallel computing capability and high memory bandwidth. However, substantial overheads are caused by hardware characteristics of GPUs. Figure 1 depicts how a CPU and a GPU are interconnected. Since GPUs cannot directly access data on a disk, memory transfers between the global memory (or device memory) and the host memory are necessary to load data from disks and to return a result of parallel processing in the GPUs. Although GPUs have high memory bandwidths (e.g. the GeForce GTX580 has a memory bandwidth of approximately 200GB/s.), the cost for memory transfers is a major factor that deteriorates the overall performance

Fig. 1 Interconnection between CPU and GPUs.

Fig. 2 Elapsed time for Memory exchange.

Fig. 3 Memory latency Overhead.

and sacrifices the computing capability of GPUs.

To provide a better view of this cost, Fig. 2 illustrates the elapsed time for transferring data between the global memory and the host memory. As shown in Fig. 2, the overhead measured for memory transfers between CPU and GPU is proportional to the volume of data. It is also observed that the cost for memory transfers is asymmetrical. Specifically, the cost for memory transfers from the global memory to the host memory (GPU→CPU in Fig. 2) is much greater than for those from the host memory to the global memory (CPU→GPU in Fig. 2). Another major factor is global memory access latency. According to [9], any opportunity to replace global memory access by the shared memory access of GPUs should be exploited, since the shared memory space is much faster than the global memory space. However, since there is little chance to share objects while traversing the trees of a hierarchical index structure in range query processing, the performance benefits from exploiting the shared memory space are negligible in the overall performance of range query processing.

Figure 3 illustrates the computing overhead as the number of MBRs increases. In the experiments, we measure the elapsed time to show the effects of the latency. As shown in Fig. 3, if the number of MBRs compared to the given query is smaller than a threshold value, T , the GPUs consume more time than the CPU in spite of the advanced parallelism of the GPUs. This is because the computational time of the GPUs includes significant global memory access latencies. The gain from parallel computation cannot compensate for the latencies when the number of MBRs is less than T . In our experiments, the threshold value, T , is measured at approximately 1,500 MBRs. Consequently, it is hard to expect

any performance improvements if we simply port R-trees or B-trees to GPUs without any structural changes.

To minimize such overheads, memory coalescing techniques must be exploited. These techniques are based on the fact that threads execute the same instruction at any given point in time. When all of these threads execute a load instruction, the GPUs detect whether the threads access consecutive global memory locations at the same time. In this case, the GPU coalesces all of these read operations into a combined access to the global memory [9]. In addition, the developers of the CUDA platform recommend exploiting page-locked host memory allocations [9]. The page-locked memory offers asynchronous transfers and thus bandwidth superior to non-page-locked memory. Using GPU streams, commands for memory transfers and kernel function executions that belong to different streams can be overlapped. In our implementations, we store data in the *structure-of-array* (SOA) format [28] in order to maximize the possibility of using coalesced memory transactions. In the SOA format, the individual attributes of each record are stored contiguously so that component-wise memory access by threads is possible regardless of the size of a record.

4. Related Work

4.1 In-memory Grid File

To the best of our knowledge, the in-memory grid file [17] is the first study to develop multi-dimensional index structures on GPUs. The researchers explored the use of GPUs for a grid file [5], a traditional multi-dimensional access method. They designed a massively multi-threaded GPU-based grid file for static, memory-resident multi-dimensional point data. In addition, they proposed a hierarchical grid file variant to handle data skews efficiently. Hash-based index structures such as grid files have advantages for adaptations to GPUs, owing to the characteristics of hash-based structures. For instance, operations such as the GPU-based sort, scan, and prefix sum, which are used to construct the structure, yield performance superior to that of the original CPU-based operations.

4.2 GPU-CSStree

The GPU-CSStree has been proposed in [29]. The CSS-tree, which is a static, in-memory, cache-sensitive variant of the B⁺-tree, has been proposed to support decision making within the context of a main memory database system [18]. The main feature of the CSS-tree is that the tree nodes are physically aligned as an array without any pointers. Therefore, a search of a CSS-tree can be performed without address computation. The CSS-tree can be applied to GPUs, since its array-based structure enables parallel execution of operations in query processing such as for address computations, memory accesses, and overlap tests. The GPU-CSStree takes a sorted set as input and separates the set into groups of four records. These grouped records are stored in

a texel data array representing leaf nodes. Internal nodes, which constitute the directory structure, are computed and stored in a separate array, *dir*.

4.3 KD-Tree Traversal on GPUs

The KD-Tree [19] is one of the most well-known structures for accelerating ray-tracing of static scenes. A search of the KD-tree is very similar to a search of the binary tree except for the traversal order. Several attempts have been made to implement the KD-tree structure on GPUs. An implementation of a parallel stack for KD-tree traversal using several kernels has been presented in [20]. However, the study encountered a problem in storing intermediate results, due to frequent kernel switches. Therefore, that study is not suitable for application to interactive ray tracing. Several implementations of GPU-based stackless KD-tree traversal algorithms have been proposed in [21]. Those implementations outperformed regular grids on GPUs but were still not able to outperform the CPU implementations. An interactive GPU ray tracer that achieved high performance of 15-18 million rays/s by the addition of a short stack has been proposed in [22]. Although the previous work has demonstrated some performance improvements, only static in-memory index structures were targeted to avoid the overheads mentioned in Sect. 3.

4.4 Parallel Implementation of R-Trees

GPU-based parallel implementation of the R-tree and query processing has been proposed in [33]. For a given R-tree, the approach uses two arrays to store the node structures and rectangle coordinates of the R-tree in the global memory of the GPUs. To handle n queries, it launches n thread blocks. Then, each block of threads is dedicated to solving one query by using a breadth-first approach. Although this work has improved the query performance, the approach is not suitable for dynamic database systems, since the entire node structure resides in the global memory of the GPUs.

4.5 CCQ-Tree

CCQ-tree [34] has been proposed for fast indexing of large-scale raster geospatial data. CCQ-tree is similar to the CSS-tree in terms of that it places all nodes in a one-dimensional array to completely continuous memory allocation but is more memory efficient. Specifically, the CCQ-tree utilizes multiple pyramid data structures and Z-order based prefix sums to build the CCQ-tree on a GPU. The CCQ-tree first builds a pyramid of matrices from raw raster data and determines the corresponding node of each element of the matrices that should be pruned. By traversing the pyramid, the position of each node can be computed.

4.6 Buffer Management

The buffer management system that has been suggested to

reduce the disk I/O has become an ever more important component for database systems as internal computation gets faster and parallel computing gains popularity. The main idea is to retain for a period of time in the main memory buffer disk pages that have a great possibility of being referred to again. Therefore, the performance of each buffer algorithm is highly dependent on the disk page replacement policy, which is based on user access patterns, as well as spatial and temporal localities. The most well-known algorithms include the LRU [26] algorithm and the LRU-K [27] algorithm. The LRU algorithm replaces the disk pages that were least recently used. The LRU-K algorithm uses the latest reference times and reference frequencies of pages to choose pages for replacement. This algorithm yields better performance than the LRU algorithm but still imposes a high cost, since the algorithm has to manage a priority queue whenever it accesses data. The replacement policy used in our proposed approach exploits the localities in order to achieve a greater performance gain by attempting to maximize the *GPU buffer* hit ratio.

5. GPU-Based Range Query

In this section, we introduce our proposed approach that uses the global memory as a buffer. Those approaches mentioned in Sect. 4 focus on how to efficiently parallelize their index structures on the global memory of GPUs to speed up query performances, whereas our approach aims at accelerating the range query performance on hierarchical index structures by applying a GPU-based buffer management system. It is also noteworthy that all index structures can benefit from our approach during their own query processing.

The main idea is to retain the objects of recently visited leaf nodes on the global memory of the GPUs and to perform the overlap tests through a *kernel* function. Figure 4 illustrates our proposed structure. Our approach uses the global memory as a buffer, called *GPU buffer*, which is an array of objects of leaf nodes to avoid the problem caused by data dependency. More specifically, the *GPU buffer* is based on the SOA format with an array for each dimension of a data object. In the case of the original memory format, *i.e.*, the AOS (*array-of-structures*) format, the threads of the warp read one attribute of the object at a time. This results in as many uncoalesced memory reads from the global memory as there are the dimensions of data whenever the elements of such a structure must be accessed by a thread. Conversely, the SOA format guarantees that all reads from global memory are coalesced, regardless of the number of dimensions of data, since all threads of the same warp-half access consecutive single values in the global memory [28]. An in-memory R-tree called *Q-tree*, which means ‘Query-tree’ is used for managing the *GPU buffer*. As shown in Fig. 5, the *Q-tree* indexes queries instead of objects and thus plays a crucial role in checking whether the area that overlaps with a given query is already in the *GPU buffer*. Moreover, based on the results from query processing on the *Q-tree*, we can significantly narrow down the search-space

Fig. 4 The proposed structure.

Fig. 5 Q-tree and GPU buffer.

of the *GPU buffer*.

For example, as shown in Fig. 5, if given a query Q that intersects with $Q5$ and $Q6$, we can search only the part designated by *start_point* and *end_point* instead of the whole *GPU buffer*. The *leaf table* represented in Fig. 4 is used for managing buffered leaf nodes. It contains the identifiers of leaf nodes in the *GPU buffer*. Whenever a leaf node is reached while query processing on the R-tree, we check whether the node is already buffered using the *leaf table* so that we can avoid disk reads and computations for buffered nodes. The *Q-tree* and *leaf table* are vital components for buffer management. All updating operations for the *GPU buffer* are performed after searching the *Q-tree* and *leaf table*. In particular, for a deletion of a specific part of the *GPU buffer*, we first find the part through a *Q-tree* search. Then, the corresponding leaf identifiers in the *leaf table* and the *Q-tree* nodes containing relevant query information must be removed. These areas that are removed from the *Q-tree* and *leaf table* are searched through the R-tree for subsequent queries. We will discuss the management of the *GPU buffer* in more detail in Sect. 6.

5.1 Search

Algorithm 1 gives a sketch of the pseudo code of range query processing on our proposed approach. When a range

Algorithm 1. RangeQuery

```

Begin
1:  start_point = 0;
2:  end_point = 0;
3:  Q_Rtree_Search(query, start_point, end_point);
4:  if Q_Rtree is overlap with query range then
5:    IsOverlap = true;
6:    SearchOnGPU(query, *start_point, *end_point);
7:  end if
8:  RangeQuery_On_R_tree(query);
9:  if IsOverlap is true then
10:    Get result back from GPU;
11:  end if
12:  Update GPU_buffer;
End

```

Algorithm 2. *Q_tree_Search* (*n*, *q*, *s*, *e*)

```

n = head node of Q_tree;      s = 0; //start_point
q = query;                    e = 0; //end_point
Begin
1:  if n is not leaf node then
2:    for i := 1 to n->count
3:      if n->child[i] is overlap with q then
4:        Q_tree_Search(n->child[i], q, s, e);
5:      end if
6:    end for
7:  else
8:    for i := 1 to n->count
9:      if n->child[i] is overlap then
10:       if s > n->child[i].start_point then
11:         s = n->child[i].start_point;
12:       end if
13:       if e < n->child[i].end_point then
14:         e = n->child[i].end_point;
15:       end if
16:     end if
17:   end for
18: end if
End

```

query arrives, the search algorithm begins with the search on the *Q-tree* first (line 3). If overlapping areas exist, both the R-tree and the *GPU buffer* are needed for query processing. Specifically, the *Q-tree* searches the specific part of the *GPU buffer* designated by the pair *start_point* and *end_point*, as shown in Fig. 5. Then, kernel functions are executed to run overlap tests in parallel (line 6). While the kernel functions are running, the query is processed also on the R-tree. When a leaf child pointer is encountered during the R-tree search process, the *leaf table* is used to check whether the leaf node is buffered. If the node exists in the *GPU buffer*, the leaf node is simply ignored. Note that disk accesses and computations are not required for such nodes. Conversely, in the case where overlapping areas do not exist, the query is processed on the R-tree only.

Once the R-tree range search has been completed, the search algorithm issues a memory transaction to obtain the result of the *GPU buffer* search, if the *GPU buffer* search is invoked (lines 9 and 10). Then, the updating procedure is invoked. Algorithm 2 gives a sketch of the range query algorithm on the *Q-tree*. The algorithm descends the *Q-tree* in the same manner of the original R-tree. When it

Algorithm 3. *GPU buffer search* (*Q*, *GPU buffer*, *s*, *e*, *flag*)

```

Begin
// s = start_point, e = end_point
1:  cur := blockIdx.x * blockDim.x + threadIdx.x + s;
2:  num_threads := blockDim.x * gridDim.x;
3:  for i := cur; i < e; i += num_threads
4:    if Intersect(Q, GPU buffer[i]) is true then
5:      flag[i] ← OVERLAPPED;
6:    end if
7:  end for
End

```

reaches a leaf node (at line 7), it finds the smallest value of *start_point* and the largest value of *end_point* in line 8~17 to determine which part of the *GPU buffer* must be searched. In Fig. 5, both *Q5* and *Q6* intersect with the given query *Q*. Therefore, the *Q-tree* search results in a pair that comprises the *start_point* of *Q5* and the *end_point* of *Q6*. Once the *Q-tree* search has been completed, the GPU threads execute kernel functions in order to scan the corresponding part of the *GPU buffer* specified by the *start_point* and the *end_point*. For the *GPU buffer* search in Algorithm 3, we use a pool of threads, which is tuned to fully utilize the advantages of the GPUs to handle a large number of objects in parallel. For a given query, each thread checks whether an object intersects with the query range. After finishing one check, a thread handles another. Specifically, the thread with a given unique identifier *t* starts its *i*th overlap test by reading in the (*t* + *iT*)th object from the *GPU buffer* until the *end_point* is encountered, where *T* is the number of threads.

5.2 Execution Configuration

The execution configuration defines the dimension of the grid and blocks for GPU threads. The execution configuration has a critical impact on maximizing the performance, since the number of blocks and warps residing on each multiprocessor for a given kernel depends on the execution configuration. To fully utilize the resources of GPUs, it is recommended that a thread block contains at least two warps (1 warp = 32 threads) to replace threads stalled by memory access operations. Usually, a thread block has 256 or 512 threads to hide memory access latency. It is also recommended that at least two thread blocks are assigned to each streaming multiprocessor to minimize the synchronization latency.

The occupancy, the ratio of the number of resident warps to the maximum number of resident warps, also has a critical impact on overall performances. Higher occupancy provides more chances to keep the streaming multiprocessors busy. The occupancy varies according to the number of registers and the amount of the shared memory allocated to a thread. For example, in the case of Fermi devices, a shared memory of 16K is available and the maximum number of warp is 48. If the kernel uses 32 bytes of the shared memory per thread, only 16 warps can be executed concurrently and thus the occupancy is 1/3. In our approach, we specify the execution configuration to meet the above conditions.

The GPUs used in our experiments have eight multiprocessors and 24 warps can reside in each multiprocessor. For the proposed approach, one of possible execution configurations is 16 blocks and 128 threads per block considering the shared memory and registers used by a thread. To achieve better performance, we use an execution configuration of 256 blocks and 512 threads per block.

6. Update Strategies

Like that of any other system using buffers, the performance of our proposed approach mainly depends on the *GPU buffer* hit ratio, which is significantly affected by the buffer updating strategy. In the following two cases, the *GPU buffer* needs to be updated:

- 1) When objects are transferred to the *GPU buffer*.
- 2) When objects in the *GPU buffer* are modified by operations such as insert, delete, split and merge.

In managing the *GPU buffer*, however, the updating procedure is challenging. Concretely, the updating procedure starts with a search of the *Q-tree*. In the first case, if the *GPU buffer* has memory space enough to retain the new objects, the updating procedure is straightforward and is completed by appending new objects to the *GPU buffer*. Otherwise, some objects must be removed from the *GPU buffer* to secure enough memory space for the new objects. Therefore, we first need to decide which part of the *GPU buffer* should be replaced. To preserve the spatial locality [23]–[25], we use the *Q-tree* to search the node farthest from the most recent query. While the search algorithm descends the *Q-tree*, every time that it visits a node it finds the branch farthest from the recent query. If it reaches the farthest leaf node, it simply deletes the node. Then, the corresponding buffer area and leaf identifiers are deleted. The procedure is repeated until enough memory space is secured. However, this strategy has various defects. The strategy cannot guarantee that the spatial locality of the *GPU buffer* is preserved, since it simply appends objects to the *GPU buffer* in a consecutive manner. As a result, the strategy may significantly enlarge the search-space and increase the data transfer cost.

In the second case, the first step is also the *Q-tree* search for the modified node. Once the modified node is found, we simply delete the object, *GPU buffer* area, and corresponding leaf identifiers. However, this update strategy severely damages the spatial locality, since it eliminates objects that will be needed for processing queries in the near future.

7. Experiments

In this section, we verify the effectiveness of our approach by comparing it against the original R-tree. We demonstrate that our proposed approach can significantly accelerate the range query processing of a tree-based index structure in spite of the disadvantages discussed in Sect. 2. All experiments are conducted on a machine with an Intel Core2 Quad

CPU Q6600 and GeForce 8600GT by processing range queries on both the original R-tree and the proposed approach. The reason why we choose the original R-tree instead of those approaches mentioned in Sect. 4 for comparison is due to the fact that they focus on parallelizing index structures by storing the whole nodes to the global memory of GPUs. On the other hand, our approach is by design to apply a GPU-based buffer mechanism to index structures such as the original R-tree while maintaining index structures on a disk. Also it is noteworthy that any index structures can benefit from our approach during their own query processing. For the evaluation, we employ datasets listed in Table 1. We generate three synthetic datasets of 100,000 MBRs each, according to uniform, skewed, and Gaussian distributions.

In addition, three real data sets from the TIGER/Line datasets are employed. In our implementation, we use a configuration of 256 thread blocks and each block contained 512 threads so as to have full warp occupancy. We assume that the maximum node capacity M is 200, and the *GPU buffer* size is set to 128KB. It is obvious that the *GPU buffer* size may have a non-trivial effect on the overall performance, since the *GPU buffer* hit ratio increases with the *GPU buffer* size. However, once the *GPU buffer* hit ratio is fixed, the performance gain (response time of R-tree / response time of proposed approach) of the proposed approach is nearly independent of the *GPU buffer* size since most of the performance gain comes from reduced I/O cost.

To further illustrate this fact, we measure the elapsed times of the proposed approach for *GPU buffer* sizes 64, 128, and 256KB as shown in Fig. 6. Since the *GPU buffer* size has little effect on the overall performance for a given *GPU buffer* hit ratio, we focus on evaluating the

Table 1 Datasets.

Dataset	Type	Dimension	Number of tuples
California	MBR	2	2,249,727
Iowa	MBR	2	556,696
Germany	MBR	2	161,797
Uniform distribution	MBR	2	100,000
Uniform distribution	Point	3,4,5,6	100,000
Skewed distribution	MBR	2	100,000
Gaussian distribution	MBR	2	100,000

Fig. 6 Effect of *GPU buffer* size.

Fig. 7 Elapsed time vs. *GPU buffer* hit ratio for generated datasets.

Fig. 8 Elapsed time vs. *GPU buffer* hit ratio for real datasets.

Table 2 Notations and Descriptions.

Notation	Description
<i>GPU buffer</i> hit ratio	Hits in <i>GPU buffer</i> / Number of result tuples
CPU time	R-tree search time.
GPU time	<i>Q-tree</i> search time + buffer search time in GPU.
<i>memcpy</i> time	Time consumed for transferring new objects and result set.

performance of our approach with a certain buffer size (128KB). It is noteworthy that the results of experiments with different buffer sizes show similar patterns to those in Figs. 7 and 8. We summarize all the notations used in our experiments in Table 2. The most important notation is the *GPU buffer* hit ratio that is defined to observe the effect of the *GPU buffer* for the overall performance. We measure the response time of both algorithms as the *GPU buffer* hit ratio increases to figure out how many disk I/O have to be reduced to improve the range query performance. To achieve this, we gradually increase the *GPU buffer* hit ratio while maintaining the selectivity of queries constant.

The overall cost is the summation of CPU time, GPU time and *memcpy* time. Figures 7 and 8 show the performance evaluations on the two-dimensional datasets listed in

Table 1. This set of experiments shows similar patterns to each other as shown in Figs. 7 and 8. We observe that the proposed approach starts to surpass the original R-tree when the *GPU buffer* hit ratio exceeds 0.15. Figure 9 shows the elapsed times when the dimensionality is varied from three to six. Regardless of the dimensionality of the dataset, the proposed approach begins to outperform the original R-tree when the *GPU buffer* hit ratio is about 0.15.

To provide a better view of the performance gain from the coalesced memory access mentioned in Sect. 3, we compare the proposed approach (*i.e.*, SOA-based *GPU buffer*) with the *array-of-structure* (*i.e.*, AOS-based *GPU buffer*), which is exactly the same as our approach except for the fact that this approach stores nodes in the AOS format. For this experiment, we employ uniform dataset with dimensionality $d = 2$. We measure the response time when the number of MBRs that are compared to a query range varies. As shown in Fig. 10, our SOA-based *GPU buffer* continuously shows much faster response time than an AOS-based *GPU buffer*. As mentioned in Sect. 3, this performance gain results from an optimized memory access pattern that maximizes the possibility of coalesced memory transactions being occurred.

To further understand the characteristics of the pro-

Fig. 9 Varying dimensionality d .

Fig. 10 SOA vs. AOS.

Fig. 11 Memory usage.

posed approach, in the next set of experiments, we measure the memory consumption of our approach and the original R-tree. As shown in Fig. 11, our approach requires slightly more memory than the original R-tree. Concretely, even though our approach uses up to 10% more memory, our approach achieves query performance up to five times higher than that of the original R-tree.

8. Conclusions

The rapid growth in data volume over the past few decades has intensified the need for faster query processing. There has been much research on exploiting parallel computation environments to accelerate query processing for traditional multi-dimensional access methods. Recently, GPUs have become a promising alternative to such environments due to massive data parallelism and high memory bandwidth. Recent studies have proposed flat structures based on GPU memory to avoid the data dependency problem and minimize the memory transfer cost. In this paper, we explained the reasons that GPU-based adaptations of hierarchical index structures involve many difficulties such as data dependencies and capacity limitations. Furthermore, we proposed a novel buffer approach for an efficient use of the GPU for hierarchical index structures. In most cases where GPUs are utilized to accelerate conventional systems, the memory transaction cost is one of the most critical performance factors. Moreover, especially for tree-based index structures, the limitation of node capacity permits little opportunity to accelerate range query processing through parallel computations. The proposed *GPU buffer* approach, however, provides an efficient way not only to minimize data transfer costs but also to maximize the advantages of the advanced data parallelism of GPUs. The proposed approach based on the *GPU buffer* guarantees enhanced query performance as long as the *GPU buffer* hit ratio is greater than 0.15. Through the proposed approach, with a minimum of additional changes in the R-tree, we achieved query performance up to five times higher than that of the original R-tree. For future work, we plan to augment our buffer approach with more efficient updating strategies for dynamic insertion and deletion operations.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0024362). This work was also partially supported by INHA UNIVERSITY.

References

- [1] A. Guttman, "R-trees: A dynamic index structure for spatial searching," Proc. SIGMOD84, 1984.
- [2] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+ tree: A dynamic index for multidimensional objects," Proc. 13th International Conference on Very Large Databases, pp.507–518, Sept. 1987.
- [3] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger, "The

- r*-tree: An efficient and robust access method for points and rectangles," *Proc. ACM SIGMOD*, pp.323–331, May 1990.
- [4] R. Finkel and J.L. Bentley "Quad trees: A data structure for retrieval on composite keys," *Acta Informatica*, vol.4, no.1, pp.1–9, 1974.
- [5] J. Nievergelt, H. Hinterberger, and K. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," A. Duijvestijn and P. Lockemann (Eds.), *Proc. 3rd ECI Conf.*, Number 123 in LNCS, pp.236–251, Berlin/Heidelberg/New York, 1981.
- [6] Y. Theodoridis and T. Sellis, "A model for the prediction of R-tree performance," *Proc. 8th Symposium on Principles of Database Systems (PODS)*, May 1996.
- [7] T. Scott, Leutenegger and M.A. Lopez, "The effect of buffering on the performance of R-trees," *IEEE Trans. Knowl. Data Eng.*, vol.12, no.1, pp.33–44, Jan. 2000.
- [8] J.D. Owens, D. Luebke, N. Govindaraju, et al., "A survey of general-purpose computation on graphics hardware," *Proc. Eurographics*, 2005.
- [9] CUDA programming Guide 4.0
- [10] C.F.I. Kamel, "Parallel r-trees," *Proc. SIGMOD92*, 1992.
- [11] X. Fu, D. Wang, W. Zheng, et al., "GPR-tree: A global parallel index structure for multiattribute declustering on cluster of workstations," *Proc. APDC*, pp.300–306, 1997.
- [12] H. Ferhatsmanoglu, A.S. Tosun, G. Canahuate, and A. Ramachandran, "Efficient parallel processing of range queries through replicated declustering," *J. Distrib. Parallel Databases*, vol.20, pp.117–147, 2006.
- [13] N. Koudas, C. Faloutsos, and I. Kamel, "Declustering spatial databases on a multi-computer architecture," *Proc. EDBT*, 1996.
- [14] B. Schnitzer and S.T. Leutenegger, "Master-client R-trees: A new parallel R-tree architecture," *Scientific and Statistical Database Management*, pp.68–77, 1999.
- [15] S. Lai, F. Zhu, and Y. Sun, "A design of parallel R-tree on cluster of workstations," *Proc. IWDNIS 2000*, pp.119–133, 2000.
- [16] B. Wang, H. Horinokuchi, J. Kaneko, and A. Makinouchi, "Parallel R-tree search algorithm on DSVM. database systems for advanced applications," *Proc. SICDSAA*, Hsinchu, Taiwan, 1999.
- [17] K. Yang, B. He, R. Fang, M. Lu, N. Govindaraju, Q. Luo, P. Sander, and J. Shi, "In-memory grid files on graphics processors," *Proc. 3rd International Workshop on Data Management on New Hardware*, 2007.
- [18] J. Rao and K.A. Ross, "Cache conscious indexing for decision-support in main memory," *Proc. VLDB*, pp.78–89, 1999.
- [19] J.L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol.18 no.9, pp.509–517, Sept. 1975.
- [20] M. Ernst, C. Vogelgsang, and G. Greiner, "Stack implementation on programmable graphics hardware," *Proc. Vision, Modeling, and Visualization Conference*, 2004.
- [21] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer," *Proc. HWS '05*, pp.15–22, 2005.
- [22] D. Reiter, H.J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing," *Proc. 2007 Symposium on Interactive 3D Graphics and Games*, pp.167–174, 2007.
- [23] L. Ki-Joune and L. Robert, "The spatial locality and a spatial indexing method by dynamic clustering in hypermap system," *Advances in Spatial Databases*, pp.207–223, 1990.
- [24] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Parallel processing of spatial joins using R-trees," *Proc. Twelfth International Conference on Data Engineering*, pp.258–265, 1996.
- [25] T. Brinkhoff, H. Horn, H. Kriegel, and R. Schneider, "A storage and access architecture for efficient query processing in spatial database systems," *Symposium on Large Spatial Databases*, pp.357–376, 1993.
- [26] W. Effelsberg and T. Harder, "Principles of database buffer management," *ACM Trans. Database Syst.*, vol.9, no.4, pp.560–595, Dec. 1984.
- [27] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *ACM SIGMOD Conference*, pp.297–306, May 1993.
- [28] J. Siegel, J. Ributzka, and X. Li, "CUDA memory optimizations for large data-structures in the gravit simulator," *Proc. ICPPW*, 2009.
- [29] R. Fang, B. He, M. Lu, K. Yang, N.K. Govindaraju, Q. Luo, and P.V. Sander, "GPUQP: Query co-processing using graphics processors," *Proc. SIGMOD07*, 2007.
- [30] K.S. Bang and H. LU, "An efficient index structure for spatial databases," *J. Database Management*, vol.7, pp.3–16, 1996.
- [31] G.-H. Cha and C.-W. Chung, "An efficient data structure for organizing multidimensional data," *J. Database Management*, 1997.
- [32] E. Tousidou, M. Vassilakopoulos, and Y. Manolopoulos, "Performance evaluation of parallel S-trees," *J. Database Management*, vol.11, pp.28–34, 2000.
- [33] L. Luo, and M.D.F. Wong, "Parallel implementation of R-trees on the GPU," *In Proc. Of ASP-DAC*, 2012.
- [34] J. Zhang, S. You, and L. Gruenwald, "Indexing large-scale raster geospatial data using massively parallel GPGPU computing," *Proc. 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2010.
- [35] B. Yu, et al., "Parallel range query processing on r-tree with graphics processing unit," *IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC)*, 2011, 2011.

Beseon Yu received the B.S and M.S degree in Information and Communication Engineering from Inha University, Incheon 402-751, Korea in 2008 and 2010, respectively. Currently, he is a third year Ph.D student in the School of Information and Communication Engineering of the Inha University. His research interests include database system, GPU and sensor network system.

Hyunduk Kim received the Master of Engineering degree in the School of Information and Communication Engineering from Inha University, South Korea, in 2009. He is currently a Ph.D. candidate in Information and Communication Engineering of Inha University, Korea. His research interests include sensor network routing algorithm, network traffic management, and ad-hoc network.

Wonik Choi is an Associate Professor in the School of Information and Communication Engineering at Inha University, where he runs Ubiquitous Computing Lab. He received his PhD in Computer Engineering from Seoul National University, Korea. His research interests include spatio-temporal databases, sensor network topology, telematics and GIS/LBS. He was a visiting scholar in the school of Computer Science at Georgia Institute of Technology in 2012. He is a member of IEEE since 2006.

Dongseop Kwon received his M.S. and B.S. degrees in the Department of Computer Engineering from Seoul National University, Seoul, Korea, in 1998 and 2000, respectively and Ph.D. in the School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea in 2005. He is currently an Associate professor of the Department of Computer Software, Myongji University, Gyeonggi, Korea. His current research interests include spatio-temporal databases, high dimensional index

structures, mobile data managements, and time series databases.