mathematics

Article

Efficient Parallel Processing of R-Tree on GPUs

Jian Nong [1,2,3], Xi He [2,3,*], Jia Chen [2,3] and Yanyan Liang [1,*]

1 School of Computer Science and Engineering, Faculty of Innovation Engineering, Macau University of Science and Technology, Macau 999078, China; 2009853xii30001@student.must.edu.mo or nongjian@gxuwz.edu.cn
2 Guangxi Key Laboratory of Machine Vision and Intelligent Control, Wuzhou University, Wuzhou 543002, China; chenjia@gxuwz.edu.cn
3 High Performance Computing Laboratory, Wuzhou University, Wuzhou 543002, China
* Correspondence: hexi@gxuwz.edu.cn (X.H.); yyliang@must.edu.mo (Y.L.); Tel.: +86-150-7740-1591 (X.H.); +86-853-8897-1997 (Y.L.)

Abstract: R-tree is an important multi-dimensional data structure widely employed in many applications for storing and querying spatial data. As GPUs emerge as powerful computing hardware platforms, a GPU-based parallel R-tree becomes the key to efficiently port R-tree-related applications to GPUs. However, traditional tree-based data structures can hardly be directly ported to GPUs, and it is also a great challenge to develop highly efficient parallel tree-based data structures on GPUs. The difficulty mostly lies in the design of tree-based data structures and related operations in the context of many-core architecture that can facilitate parallel processing. We summarize our contributions as follows: (i) design a GPU-friendly data structure to store spatial data; (ii) present two parallel R-tree construction algorithms and one parallel R-tree query algorithm that can take the hardware characteristics of GPUs into consideration; and (iii) port the vector map overlay system from CPU to GPU to demonstrate the feasibility of parallel R-tree. Experimental results show that our parallel R-tree on GPU is efficient and practical. Compared with the traditional CPU-based sequential vector map overlay system, our vector map overlay system based on parallel R-tree can achieve nearly 10-fold speedup.

Keywords: graphics processing unit (GPU); parallel R-tree; parallel computing; parallel data structure; vector map overlay

MSC: 68W10; 68P05

check for updates

Citation: Nong, J.; He, X.; Chen, J.; Liang, Y. Efficient Parallel Processing of R-Tree on GPUs. *Mathematics* **2024**, *12*, 2115. https://doi.org/10.3390/math12132115

Academic Editors: Idelfonso B. R. Nogueira, Jozef Husar and Xiangmin Jiao

Received: 23 April 2024
Revised: 27 June 2024
Accepted: 4 July 2024
Published: 5 July 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

R-tree is an important multi-dimensional data structure used in storing and querying spatial data. It is widely used in various fields. One of R-tree related applications is map searching, in which spatial objects in a map, such as hospitals, schools, restaurants, etc., are stored in an R-tree, and then the R-tree is queried to answer questions such as “find hospitals or restaurants within three kilometers of the current location”. In an R-tree, each node can hold multiple entries, and the maximum number of entries that a node can contain is defined by a metric, typically denoted by M. Each entry in an R-tree is associated with a Minimal Bounding Rectangle or MBR. MBRs are the critical metadata for constructing an R-tree. On one hand, each entry in a leaf node corresponds to a spatial object and its MBR covers this spatial object. On the other hand, an entry in a non-leaf node aggregates all entries in its child node, and its MBR is the union of all the MBRs associated with the aggregated entries. Figure 1 shows an R-tree with M equal to 3. The R-tree have three leaf nodes, R2, R3, and R4, and stores a total of 11 spatial objects D∼N. The root node, R1, has three entries A∼C, with the MBR of each entry covering the MBRs of all the entries in its corresponding child nodes.

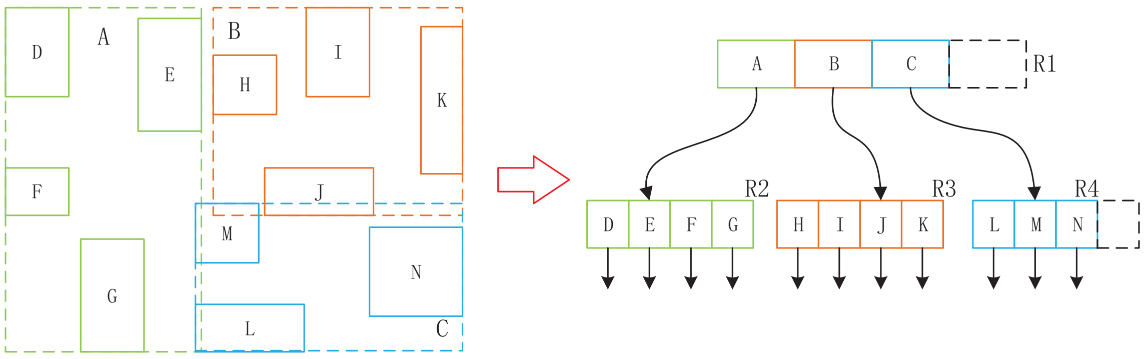

Figure 1. The layout of MBRs of R-tree entries and the corresponding R-tree.

The core idea of R-trees is to aggregate spatial objects that are close in space and minimize the overlap between the MBRs of different entries so as to reduce search paths and improve query performance. Ref. [1] is the first paper to propose the concept of R-tree. It provides a series of algorithms for insertion, deletion, query, and construction of R-trees. For example, when inserting a spatial object into an R-tree, the insertion algorithm finds the most suitable path from the root level to the leaf level to insert the spatial object so that the area increment of the related MBRs is minimized. The basic method of constructing an R-tree is to insert spatial objects into the R-tree continuously.

General-purpose graphics processing unit (GPGPU or GPU) has become an integral part of today's mainstream computing systems. The modern GPU is not only a powerful graphics engine but also a highly parallel programmable, many-core graphic processor featuring peak arithmetic and memory bandwidth. A GPU consists of an array of parallel processors, which are often referred to as streaming multiprocessors. Each streaming multiprocessor contains dozens or hundreds of streaming processors. The streaming multiprocessors run in a Single Instruction Multiple Thread (SIMD) mode in which a group of 32 threads called a warp is the minimum execution unit. Once scheduled on a streaming multiprocessor, the threads in a warp share the same instruction and execute in a fairly synchronous fashion. The GPU is configured with global memory accessible to all streaming multiprocessors and share memory used internally by each streaming multiprocessor.

For parallel tree-based data structures, several prerequisites must be taken into account to properly exploit the potential parallel performance benefits offered by the GPU. First, workloads need to be assigned to every streaming processor so as to make full use of the computing resources inside the GPU. Second, branch instructions should be avoided in the program. Assuming that there are 16 threads in a warp that satisfy the condition and another 16 threads that do not, half of the threads will execute the statements in the "if" block while the other half executes the statements in the "else" block. This appears to be a paradox, given that the threads in a warp can only execute the same instruction at the same time. In reality, when encountering a branch divergence, the GPU will execute each branch path sequentially, disabling the threads that are not on that path until all enabled branch paths have been executed, at which point the threads reconverge on the same execution path. Some threads will be idle on each branch, which results in a performance loss of at least half as much as the original branch split, or at worst, 1/32 of the peak performance if every thread executes the branch differently. Third, accessing data in the global memory is one of the most important dimensions of CUDA kernel performance. Coalesced memory access is a favorable data access pattern when designing kernel code. Such coalesced memory access allows the dynamic random access memories to deliver data at a rate close to the peak global memory bandwidth. Due to the powerful parallel computing capability, many traditional computing problems have been successfully ported to the GPUs, and the computing time has been shortened by more than ten times or even tens of times.

The design and development of parallel R-tree on GPUs is crucial to many R-tree based applications. The traditional R-tree structure, construction algorithm, and query algorithm are designed for the CPU and do not consider the characteristics of the GPUs. Thus the performance is inefficient when porting traditional R-tree data structure to GPUs directly. To address this issue, we redesign the R-tree structure and related operations and demonstrate its feasibility by employing it to help port the vector map overlay system to the GPU. Vector map overlay is one of the important spatial operations in GIS and is computationally intensive and time-consuming. The contributions of our work can be summarized as follows:

- We have designed a more efficient data structure for R-tree on GPUs.
- We present two parallel R-tree construction algorithms and one parallel R-tree query algorithm that can take the hardware characteristics of GPUs into consideration.
- We present the migration of a vector map overlay system from CPU to GPU to demonstrate the feasibility of parallel R-tree.

The rest of this paper is organized as follows: Section 2 briefly reviews the related work. Section 3 presents the ideas behind the algorithms of parallel R-tree construction and query. Detailed descriptions of these algorithms are also included in this section. Section 4 discusses the application of parallel R-tree in the context of vector map overlap problems. In Section 5, comprehensive experiments are conducted to evaluate the performance of parallel R-tree and vector map overlap systems. Analyses on these experiments are also included. Finally, we offer some conclusions and roadmaps for future work in Section 6.

2. Related Work

2.1. Related Variants of R-Trees

The traditional R-tree is a dynamic spatial index data structure that can be inserted, deleted, and queried simultaneously without the need for periodic reorganization. Since the MBRs of different entries in R-tree may overlap, there may be multiple search paths for R-tree query operations. As we introduced in the Section 1, an important goal of building an R-tree is to reduce the overlap between the MBRs and thus reduce the search path. In order to avoid the problem of multi-path query in R-tree, R+tree is designed by Sellis et al. [2]. R+tree uses object separation technology to separate and store objects across subspace in different nodes. Although R+tree can solve the problem of multi-path search, it also brings problems such as redundant storage and complicated update operations. R*tree is proposed by Beckmann et al. [3]. A series of node splitting optimization criteria and node forced reinsertion technology are designed to increase the space utilization of R-tree, which significantly improves the tree query performance. The Hilbert R-tree proposed by Kamel et al. [4] uses a Hilbert curve to sort one-dimensional linear data in K-dimensional space and then sorts tree nodes so as to obtain higher node storage utilization.

2.2. Parallel Tree-Based Data Structures on GPUs

Considering that this work deals with tree-based data structures on GPUs, here is a brief overview of this area. Tree-based data structures are often stored in linear arrays to make more efficient use of the coalesced memory access features of GPUs. GPUs are better suited for batch operations, such as constructions and batch queries. Delete and update operations on GPU should be avoided.

The traditional data structure based on CPU have been studied by researchers for many years. As GPUs are widely used in both industry and academics, research into parallel data structures on many-core architectures has become urgent. A number of attempts have been made to accelerate the processing of parallel data structures by using the massive parallelism of GPUs. Luo et al. [5] combines a queue with the conventional array of structures to parallelize the R-tree index search in the breadth-first-search fashion on GPUs. You et al. [6] proposed a GPU-based R-tree to perform the breadth-first search with a queue. It is hard to exploit the maximum parallelism potential of GPUs due to the high correlation of their hierarchical data and computational structures. Unfortunately, the highly dependent data and computation structure of this method is inadequate for maximizing the coalesced memory access of GPUs. Zhou et al. [7] constructed KD-tree in breadth-first order and adopted a novel construction algorithm for large nodes to make full use of the computing power of the GPU. To solve the problem of consuming too much memory of GPU in [7], Hou et al. [8] constructed KD-tree in partial breadth-first order and sacrificed some concurrency in exchange for large memory savings. Kim et al. [9] proposed a parallel KDB-tree to utilize GPUs for query processing using a hierarchical structure. This approach is effective in accelerating the retrieval of the R-tree, but it pays the high cost to find the leftmost child and the rightmost child on GPUs. An approach for kd-tree generation on GPU [10] is introduced to improve the performance when working with medium-size point datasets. This approach adapts a parallel sorting algorithm to sort sub-sections of the data independently. A GPU-aware parallel indexing method called G-tree [11] is presented to address the high-dimensional big data. It exploits the advantage of the traditional R-tree and the parallel computation benefits of GPUs. The rationale of the design is to combine the efficiency of the R-tree in lower-dimensional space with the parallel computing capability of GPUs in higher dimensionality. He et al. [12] use the strategy of large nodes and pipelines to realize concurrent heaps in GPUs, which gives a speed up of 20-fold. And the concurrent heap is successfully applied to the morphological reconstruction system [13]. Other tree-based data structures, such as Octree [14–18], decision tree [19–21], and bounding volume hierarchy [22,23], have been ported to GPUs. Besides the tree-based data structures, hash tables on GPUs have also been discussed in [24–26]. Kim et al. [27] presented the LBPG-tree, an efficient multi-GPU indexing scheme that integrates the advantages of CPU instruction pipelining with the parallel processing capabilities of GPUs. Through novel strategies, it optimizes the utilization of GPU L2 cache to accelerate index searching and node access on GPUs. Also, the LBPG-tree employs a hierarchical pipeline approach to maximize the utilization of streaming multiprocessors and introduces a compact-and-sort mechanism to enhance memory throughput. Xiao et al. [28] proposed a distributed parallel R-tree solution supported by RDMA on CPU clusters. This approach achieves low-latency and high-throughput parallel R-tree processing by adaptively utilizing network bandwidth and computing resources. These two methods proposed by Kim et al. and Xiao et al. are parallel schemes based on distributed computing resources, which are different from our scheme. We focus on parallel computing on a single GPU.

Despite the challenges of designing R-trees on GPUs, which have been primarily discussed in [5,6], more work is still needed for a better R-tree on GPUs. Ref. [5] presents a R-tree memory layout on GPU. But this layout has two problems: (1) It has a fixed size for every R-tree node, even though the R-tree node is not full. For large datasets, this design can waste many GPU memory spaces. (2) The design of this R-tree layout makes it very difficult to access R-tree nodes by levels. This is not good for parallel R-tree construction, R-tree query, and other R-tree operations because we need to access a level of R-tree nodes in parallel and take advantage of coalesced memory access. Essentially, this design of the R-tree memory layout still follows the idea of traditional R-tree design on the CPU and fails to think about R-tree design in a parallel way. So is its R-tree construction algorithm and query algorithm. It is hard to execute its construction algorithm and query algorithm in a way that can fully take advantage of the computing power of GPU. What is more, its query algorithm focuses on the performance of a single R-tree query instead of a batch of R-tree queries. This can prevent the potential collaboration of threads on GPU and thus hurt the performance of R-tree query batches. The R-tree memory layout presented in [6] is similar to the one in [5], and it also lacks the support for accessing a level of R-tree nodes in parallel. Therefore, its construction algorithm and query algorithm suffer the same problem as [5]. The paper proposes a more complicated R-tree query algorithm, yet this query algorithm is the direct port of the CPU query algorithm and may not be the most well-suited R-tree query algorithm on GPU. Our paper is aimed at developing a GPU-friendly R-tree data structure and tries to explore the balance between R-tree construction efficiency and resulting R-tree quality. We also design an R-tree query algorithm targeted at R-tree query batches.

3. Parallel R-Tree Processing on GPUs

3.1. Parallel R-Tree Construction Algorithm

To take full advantage of coalesced memory access of GPUs, the parallel R-tree proposed in this paper is stored in four one-dimensional arrays. With this design, coalesced memory access is promoted and scattered memory reads are reduced, thus enhancing the overall performance of R-tree access. Figure 2 shows the data structure of the R-tree introduced in Figure 1. The four arrays are the *level* array, *start* array, *end* array, and *entry* array. The *start*, *end* and *entry* arrays define the relationship between the R-tree nodes and their entries. As to the R-tree shown in Figure 2, the first element in the *start* array has a value of 1, and the first element in the *end* array has a value of 3. These indicate that the root node R1 contains three entries (i.e., *A*, *B*, and *C*) with the indexes from 1 to 3 in the *entry* array. Similarly, the second node *R*2 contains four entries (i.e., *D*, *E*, *F*, and *G*) with the indexes from 4 to 7 in the *entry* array. The third node R3 contains four entries (i.e., *H*, *I*, *J*, and *K*) with the indexes from 8 to 11 in the *entry* array. The fourth node R3 contains three entries (i.e., *L*, *M*, and *N*) with the indexes from 12 to 14 in the *entry* array. We can also observe that the number of R-tree nodes can be obtained from the size of the *start* array or the *end* array. The *level* array records the index of the first node at each level in the *start* or *end* array. Since the R-tree in this example has two levels, there are two elements in the *level* array. Specifically, the first node at the first level is *R*1, which is denoted with the first element in the *start* and *end* arrays. Therefore, the first element in the *level* array is 0. Similarly, the first node at the second level is *R*2, which is denoted with the second element in the *start* and *end* arrays. Thus the second element in the *level* array is 1. By making use of the structure of arrays, spatial object data of different regions can be loaded into threads in the same warps at one memory read; therefore, the parallel R-tree construction and query can perform efficiently.

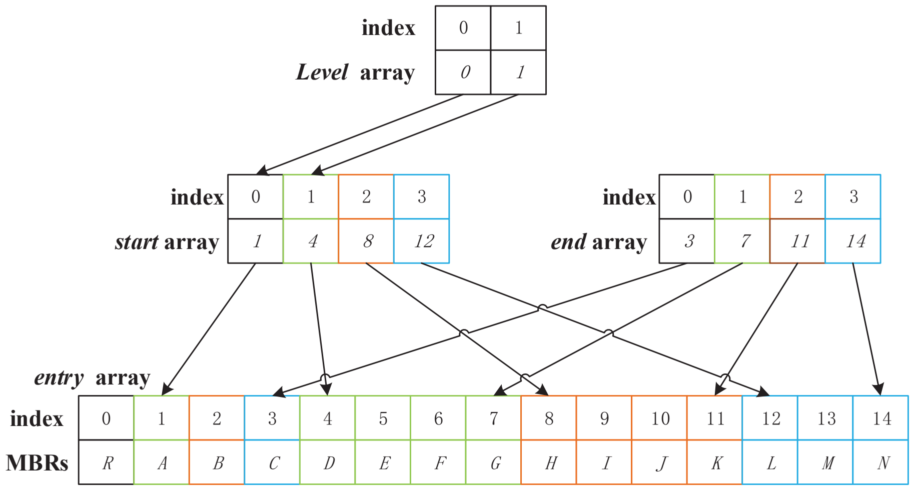

Figure 2. The memory layout of a parallel R-tree.

Generally, every thread needs to have sufficient computing tasks so as to better take advantage of the parallel computing capacity of GPUs. Traditional R-tree construction is performed by inserting spatial objects in succession. Essentially, it is performed sequentially. In addition, inserting spatial objects sequentially in an R-tree requires deciding whether the spatial object is inserted into the left subtree or the right subtree. In other words, branching is introduced. Therefore, the traditional construction algorithm is not well-suited for R-tree parallel construction on GPUs. In this work, we adopt a bottom-up or top-down parallel batch processing strategy to build R-tree level by level.

The bottom-up parallel construction algorithm is exhibited in Algorithm 1 (Bottom-up GPU). Its basic idea is to build each level of R-tree in a bottom-up fashion by merging entries at the lower level. Lines 3–9 are executed on CPU, while lines 10–16 are carried out on GPU. Line 3 reads the spatial objects from the input file and creates R-tree leaf level entries by computing the MBRs of these spatial objects. The entry size at R-tree leaf level is calculated at line 4, and lines 5–6 compute the total number of entries and the height of the R-tree. These computations are necessary for determining the memory space required by R-tree. GPU memory is then allocated for R-tree structure in line 7. Next, the entries at the R-tree leaf level are sorted by the space filling curve (SFC) values [29] of their MBRs. These sorted entries are copied to the *entry* array on GPU in line 9, and the *start*, *end*, and *level* arrays need to be updated accordingly in line 10. In lines 12–14, the algorithm loops over all of the R-tree levels to fill the four arrays with R-tree structure. At each iteration, entries at a R-tree level are generated in parallel by merging M entries at the lower R-tree level, and these newly created entries are also stored in the *entry* array. With the new entries written in the entry array, the *start*, *end*, and *entry* arrays are also needed to be updated to keep track of the relationships between R-tree levels and R-tree nodes and between R-tree nodes and R-tree entries. When this algorithm is executed, the entire R-tree structure is stored in the GPU memory.

Algorithm 1 Bottom-up parallel construction algorithm

```
Input: The spatial object dataset and the maximum number of entries $M$ in an R-tree node
Output: R-tree data structure in GPU memory
1: Begin
2: In CPU
3: $entries \leftarrow getEntries(dataset)$
4: $n_{entries} \leftarrow sizeof(entries)$
5: $total_{entries} \leftarrow sum(n_{entries}, M)$
6: $L \leftarrow getLevel(n_{entries}, M)$
7: $alloMem(L, M, n_{entries}, total_{entries})$
8: $sort(entries)$
9: $entry \leftarrow entries$
10: In GPU
11: $update(entry, start, end, level)$
12: for $i = L - 1$ to 1 do
13:     $createEntry(entry)$
14:     $update(entry, start, end, level)$
15: end for
16: End
```

A crucial problem in R-tree construction is how to guarantee the quality of the resulting R-tree and reduce the overlaps between entries to improve query performance. The quality of a bottom-up construction algorithm largely depends on the order of spatial objects. Therefore, it is necessary to preprocess spatial objects before building the R-tree. One of the solutions is to pre-sort spatial objects according to the horizontal coordinate of the bottom left points of the spatial objects' MBRs [5]. Unfortunately, this solution is not effective because it is simplistic and does not take into account two dimensions and above. It is not accurate to sort only by the coordinate value of a spatial object. Factors such as the size and location of the two-dimensional or multi-dimensional spatial object should be taken into consideration. The approach of this work (Line 8) is to first calculate the values of the center points of spatial objects' MBRs in the SFC and then sort the spatial objects according to these values. The basic idea is to transform spatial objects into multidimensional points and then use SFC to transform multi-dimensional objects into one-dimensional values so as to facilitate sorting with traditional sorting algorithms. Since the points of similar values in the SFC are also adjacent in space, adjacent objects in space will group together after sorting. Therefore, this algorithm can ensure the quality of R-tree construction to a certain extent.

The disadvantage of this method is that the spatial overlap between entries at non-leaf levels cannot be controlled. As a result, a top-down construction algorithm is designed.

Algorithm 2 (Top-down GPU) is designed to construct a R-tree in a top-down fashion, aiming at reducing the overlap between nodes at non-leaf level. Its idea is as follows: Initially, all the MBRs of entries obtained from the dataset are in a group. Then, starting from the root level, at each level, the MBRs in the same group are split into M sub-groups after they are sorted. The MBRs in each sub-group are merged into one entry at that level. Lines 3–8 are shared between Algorithms 1 and 2. These lines are used to compute the parameter of the R-tree so that memory space required by the R-tree can be determined and allocated. Also, the MBRs of entries created from the spatial data set are transferred to GPU. The loop from line 10 to line 19 is executed to build the R-tree level by level. According to the parity of the R-tree level, MBRs in each group are sorted according to either the x-coordinates of their bottom left corners (line 12) or the y-coordinates of their bottom left corners (line 14). Then MBRs in each group are divided into M sub-groups, and the MBRs in each group will be merged to create an entry at the current level (line 16). Similar to Algorithm 1, after the new entries are created and written to the *entry* array, the *start*, *end*, and *level* are also needed to be updated accordingly (line 17).

Algorithm 2 Top-down parallel construction algorithm

```
Input: The spatial object dataset and the maximum number of entries $M$ in an R-tree node
Output: R-tree data structure in GPU memory
1: Begin
2: In CPU
3: $entries \leftarrow getEntries(dataset)$
4: $n_{entries} \leftarrow sizeof(entries)$
5: $total_{entries} \leftarrow sum(n_{entries}, M)$
6: $L \leftarrow getLevel(n_{entries}, M)$
7: $alloMem(L, M, n_{entries}, total_{entries})$
8: $MBRs \leftarrow entries$
9: In GPU
10: for $i = 1$ to $L$ do
11:     if $i$ is $odd$ then
12:         $sortX(MBRs)$
13:     else if $i$ is $even$ then
14:         $sortY(MBRs)$
15:     end if
16:     $splitEntry(entry, start, end, level, MBRs)$
17:     $update(entry, start, end, level)$
18: end for
19: End
```

Compared with Algorithms 1 and 2 requires more computation and therefore has a larger execute time. Yet it can effectively reduce the space overlap between entries, thus the quality of the R-tree it builds should be better.

3.2. Parallel R-Tree Query Algorithm

R-tree applications, such as vector map overlay system, often generate multiple R-tree query requests simultaneously. Here we will discuss how to process these R-tree query requests efficiently on GPU. Generally, there are two approaches to execute R-tree queries: depth-first query and breadth-first query. Yet the depth-first query is not well suited for GPU execution. The execution mode of GPU is single instruction multiple data, or SIMD, which means all threads in a group (warp in Nvidia's term) always execute the same instructions. Since the workloads of different depth-first queries are usually different, and sometimes the difference may be quite large, the query thread with a small workload has to wait for the query thread in the same group with a large workload. These would result in the degradation of overall query performance. Also, depth-first queries would introduce branch divergence, thus causing performance loss (see Section 1 for more details). Moreover, it is difficult for multiple depth-first query threads to cooperate with each other, which makes it difficult to effectively coalesce memory access. For R-tree queries on GPU, their performance depends on the efficiency of memory access, and coalesced access of global memory, which can combine multiple memory access requests into one, can greatly improve the efficiency of memory access.

Therefore, R-tree queries are processed in parallel on GPU in the manner of breadth-first. The scheme is illustrated in Figure 3 where R-tree queries are executed in a level-by-level fashion. A R-tree query task in this scheme is composed of a R-tree query request and a R-tree node that this R-tree query is going to check. Initially every R-tree query request is wrapped up with the root node to be a R-tree query task. Then, starting from Level 0, these R-tree query tasks are processed in parallel, with each R-tree query task checking whether the query window of its R-tree query request intersects with any entry's MBR in the root node. If there is an intersection between a R-tree query request and an entry in the root node, then a new R-tree query task consisting of the R-tree query request and the R-tree node represented by the entry will be generated for the next level. Likewise at Level 1, all the R-tree query tasks at this level will be collected and processed in parallel. Then these R-tree query tasks will again generate new tasks for the next level. This loop will continue until the leaf level of the R-tree is visited and query results are found. There are two details worth discussing in the process. Firstly, it is needed to allocate global memory when storing the newly generated R-tree query tasks. However, the number of R-tree query tasks cannot be estimated in advance, so it is difficult to determine the memory space to be allocated. In some cases, the problem occurs when the allocated memory space is insufficient to store the R-tree query tasks. Ref. [6] proposed three kinds of data overflow solutions. Our solution is to avoid data overflow by counting the number of newly generated R-tree query tasks before allocating memory space to store them. Secondly, breadth-first query of R-tree requires a global synchronization of all threads between levels, and global synchronization is time consuming. However, there are not many entries in the first few levels of the R-tree, which cannot take advantage of the computational processing of the GPU, and each level needs global synchronization. Therefore, our R-tree breadth-first query ignores the first few levels and starts at the level with enough entries.

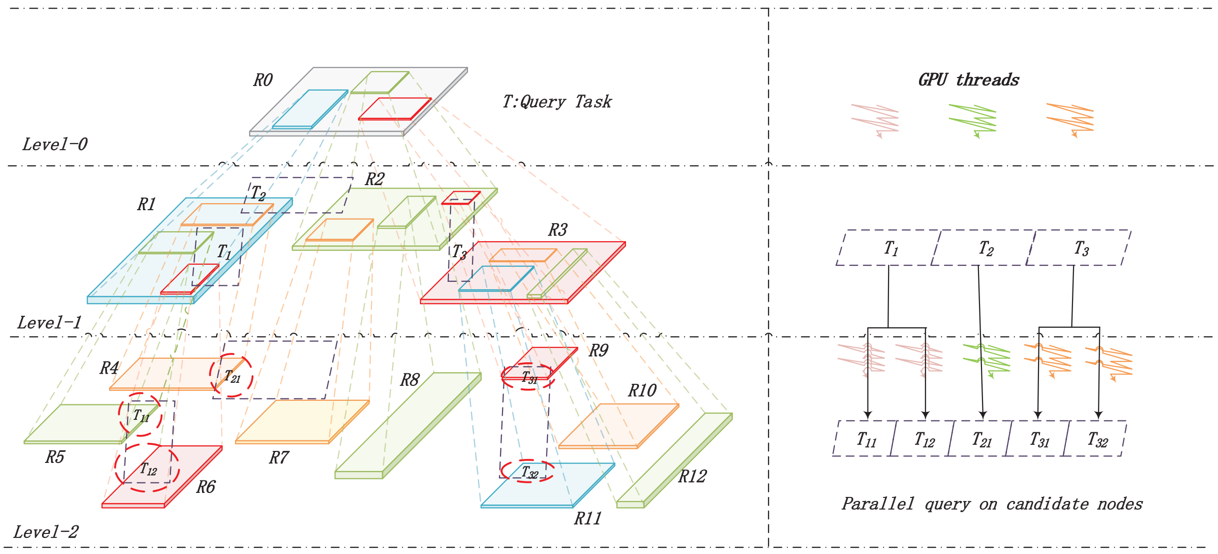

Figure 3. Breadth-first parallel R-tree queries.

The pseudo-code of Algorithm 3 sketches the process detail of breadth-first parallel query of R-tree. All the algorithm code is executed on GPU. Line 3 initializes the R-tree query tasks by wrapping the R-tree queries and R-tree nodes at level K to form R-tree query tasks. Assuming that the query starts from level K and L is the height of the R-tree, the loop in Line 4–8 searches the R-tree level by level. Line 5 processes the R-tree query tasks and uses prefix sum to count the number of overlaps between R-tree queries and the associated entries specified in R-tree query tasks. According to this number, line 6 can allocate memory space for storing newly generated tasks. Line 7 processes the R-tree query tasks again, and results are wrapped to become the new tasks. Once the loop is completed, line 10 returns the R-tree queries and their overlapping spatial objects.

Algorithm 3 Breadth-first parallel query algorithm

```
Input: R-tree data structure and R-tree query requests
Output: The spatial objects that overlap with the R-tree query
 1: Begin
 2: In GPU
 3: $tasks = initTask()$
 4: for $i = K$ to $L$ parallel do
 5:     $sum = countNewTasks(tasks, i)$
 6:     $newTasks = memAlloc(sum)$
 7:     $newTasks = processTasks(tasks)$
 8:     $tasks \leftarrow newTasks$
 9: end for
10: return tasks
11: End
```

3.3. R-Tree Deletion and Update

Parallel deletion and update operations are not suitable for R-tree running on GPUs because those operations involve concurrency and locking, which can affect the efficiency of the GPUs. A better strategy is to rebuild the R-tree if it needs to be deleted and updated. It is very efficient to construct R-tree in GPUs, especially using the bottom-up R-tree construction method. The source code of the parallel R-tree on GPU is available on GitHub at: https://github.com/83033183/ParlllelRtree (accessed on 20 April 2024).

4. Application of Parallel R-Tree

The vector map overlay problem in GIS [30] is mainly used to compute the overlay operations between a base polygon set and an overlay polygon set. The overlay operations include intersection, union, complement, and difference. This problem is computationally intensive and is ideal for running on GPUs. The parallel processing steps are as follows:

- Read the base polygon set and the overlay polygon set from the input file respectively.
- Create overlapping polygon pairs.
- Execute the overlay operations of overlapping polygon pairs in parallel and merge the results.
- Write the result to the output file.

Figure 4 shows the processing pipeline of the vector map overlay system. In Step 2, we create overlapping polygon pairs from the base polygon set and the overlay polygon set. The idea is to convert the computation of vector map overlay operation into the computation of overlay operations on each of these overlapping polygon pairs since the overlay operations of overlapping polygon pairs are independent of each other and they can be processed in parallel. The detailed process is as follows. First, we build an R-tree in parallel based on the base polygon set. Then, every polygon in the overlay polygon set is fed to the R-tree to find out all of the base polygons that overlap with the given overlay polygon. An overlay polygon and a base polygon that overlap with the overlay polygon comprise an overlapping polygon pair.

For the overlapping polygon pairs created in Step 2, the overlay operation of each pair is computed in parallel, and the resulting polygons will be collected in Step 3. The classic polygon clipping algorithm [31] is often employed to carry out the overlay operation between a pair of polygons. Yet this algorithm is designed to be executed sequentially in CPU. To speed up its performance, we designed the parallel version of the algorithm, which will be discussed in detail in another paper due to space limitations.

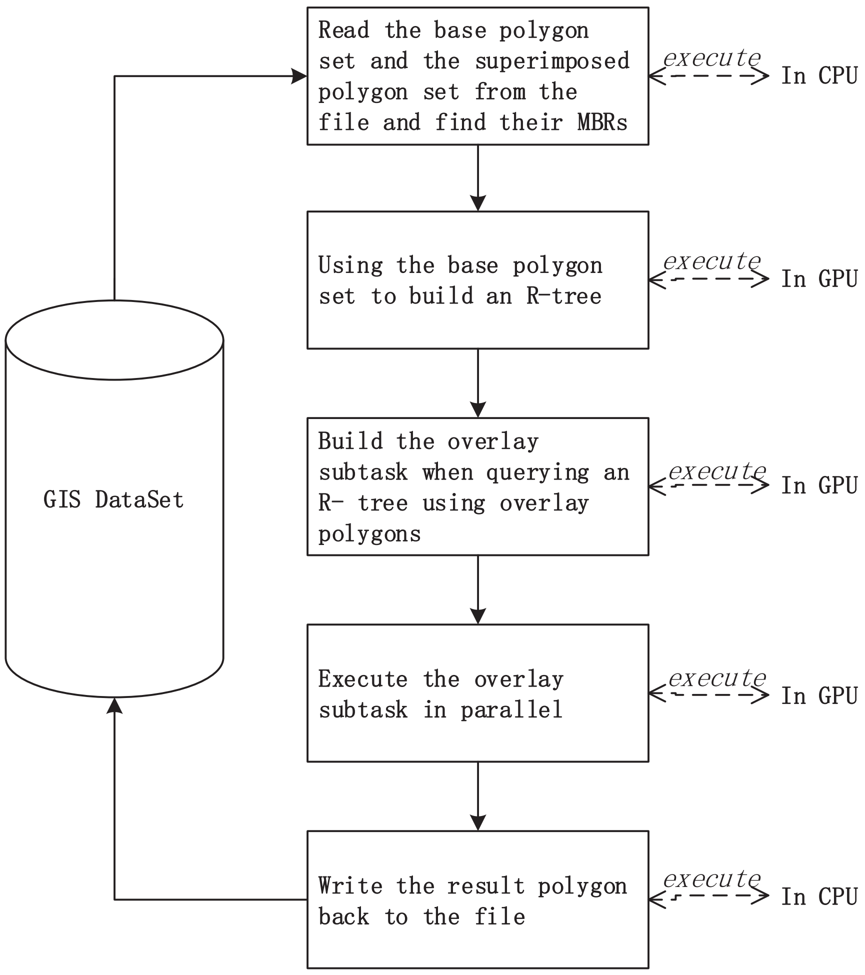
Read the base polygon set and the superimposed polygon set from the file and find their MBRs
execute
In CPU
Using the base polygon set to build an R-tree
execute
In GPU
GIS DataSet
Build the overlay subtask when querying an R- tree using overlay polygons
execute
In GPU
Execute the overlay subtask in parallel
execute
In GPU
Write the result polygon back to the file
execute
In CPU

Figure 4. Pipeline diagram of GPU based vector map overlay system.

5. Experimental Evaluation

In this section, we describe our experimental setup and provide comprehensive discussions based on our obtained experimental results.

5.1. Experimental Setup

All experiments are conducted on an Ubuntu server with one Intel Xeon Gold 5120 of 2.2 GHz consisting of 28 cores and 512 GB of memory. The graphic card is the NVIDIA GeForce RTX 2070 SUPER, consisting 2560 CUDA cores and 8 GB GDDR6 memory.

The purpose of the experiments is to measure and evaluate the performance of our parallel R-trees. We have three goals in our experiments. Firstly, we would like to measure the efficiency of our R-tree construction with different datasets and compare it with sequential R-tree construction and other GPU-based parallel R-tree construction. The quality of the resulting R-trees will be also compared and analyzed. Secondly, we want to evaluate the R-tree query performance of our parallel R-tree by comparing it with other R-tree query implementations. Lastly, we use different numbers of overlap polygons to perform overlap operations to evaluate the performance of our parallel R-trees in a practical application.

5.2. Performance Analysis

We have prepared two groups of test data for the R-tree construction and query experiments. The first group contains 5 data sets. Each data set contains rectangles, each of which stands for a spatial object. These data sets are generated randomly by computer programs. Table 1 describes the characteristics of each data set. The first column indicates the number of spatial objects in each data set. The second column shows the total number of overlapping pairs of spatial objects in each data set. The third and fourth columns indicate the average and maximum number of overlapping spatial objects that each spatial object in the data set has. The second group contains 3 data sets, namely rea02, abs02, and par02. These are the publicly available benchmark data sets from [32]. Table 2 describes the characteristics of each of these data sets.

Table 1. The characteristics of generated test data sets.

Data Size	Total Overlap Number	Average Overlap Number per Object	Maximum Overlap Number per Object
1024	5489	5.36	28
4096	80,287	19.6	109
16,384	126,199	77.03	399
65,536	20,108,655	306.83	1532
262,144	320,865,497	1224	6053

Table 2. The characteristics of benchmark data sets.

Name	Data Size	Total Overlap Number	Average Overlap Number per Object	Maximum Overlap Number per Object
rea02	1,888,012	4,892,099	2.59	317
abs02	1,000,000	950,474	0.95	7
par02	1,048,576	2,974,753	2.84	4727

The sequential R-tree implementation can be downloaded from [33] (denoted as SeqCPU). It will be used as the baseline for comparison. Also, we have implemented the parallel R-tree in [5] (denoted as ParallelRtree) and developed an openMP based parallel R-tree rooted in the idea from [34] (denoted as OpenMPRtree).

In the assessment of R-tree performance, the definition of speedup is defined as follows:

$$speedup = \frac{SeqCPU's\ execution\ time}{parallel\ Rtrees'\ execution\ time} \tag{1}$$

We first want to evaluate the quality of the resulting R-trees constructed by different algorithms. The basic idea is to execute the same R-tree queries on different R-trees, and then count the total number of R-tree nodes these R-tree queries have to deal with because the better the quality of an R-tree obtained, the fewer query paths there are and the fewer R-tree nodes a query has to deal with. The R-tree queries are generated with the exact spatial objects that are used to construct the R-tree. Table 3 shows the total number of R-tree nodes that the queries encounter for the R-trees constructed with the randomly generated data sets. The third, fourth, fifth, and sixth columns keep track of the average sequential query time on the R-tree constructed by [5,34], by our bottom-up algorithm, and by our top-down algorithm, respectively. Table 4 shows the similar statistical result for the R-trees built with the benchmark data sets. It is clear that our two algorithms can build R-trees with better quality.

Table 3. Average sequential query time for R-trees constructed with randomly generated data sets (ms).

Data Size	Query Size	ParallelRtree [5]	OpenMPRtree [34]	Bottom-Up	Top-Down
1024	1024	8.99	10.05	4.1	3.61
4096	4096	136.04	187.37	37.76	30.42
16,384	16,384	1634.71	2687.62	408.2	331.83
65,536	65,536	23,498.74	36,751.5	3568	4807.59
262,144	262,144	366,976.91	520,437.14	42,041.32	68,155.29

Table 4. Average sequential query time for R-trees constructed with benchmark data sets (ms).

Name	Data Size	Query Size	ParallelRtree [5]	OpenMPRtree [34]	Bottom-Up	Top-Down
rea02	1,888,012	1,888,012	127,578.54	158,742.22	12,919.29	6173.91
abs02	1,000,000	1,000,000	55,732.25	75,738.24	4022.59	2473.4
par02	1,048,576	1,048,576	173,510.33	235,942.18	8536.35	6275.69

Figure 5 shows the performance of different R-tree construction algorithms. It is clear that the bottom-up GPU R-tree construction algorithm, the top-down GPU R-tree construction algorithm, the ParallelRtree algorithm, and the OpenMPRtree algorithm have similar construction performance when the data set is relatively small. When the dataset increases exponentially, the construction performance of the ParallelRtree algorithm outperforms other algorithm. In this work, we focus on balancing between the efficiency of R-tree construction and the quality of the resulting R-tree construction. Although our construction algorithm is less efficient than the ParallelRtree construction algorithm, the quality of the resulting R-tree has improved a lot, making our construction algorithm worthy and useful. For example, for the abs02 data set, the construction time of our bottom-up algorithm is about 3.67 times slower than that of the ParallelRtree construction algorithm. Yet the quality of the resulting R-tree constructed by our bottom-up algorithm is 13.85 times better than that constructed by the ParallelRtree construction algorithm if we measure the quality of R-trees by their corresponding sequential query time.

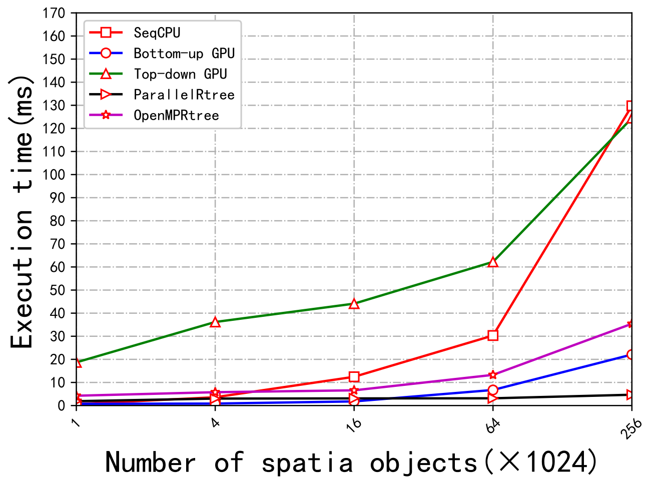

Figure 5. Execution time of different R-tree construction algorithms with randomly generated datasets.

In a real R-tree application such as a vector map overlay system, R-tree construction is usually carried out once, while R-tree queries may be executed many times, and a R-tree with good quality can save plenty of query time. Also, our data structure is relatively more complex compared to the one in [5] and thus more informative, making it more useful and practical in real applications. Figures 6 and 7 demonstrate the speedup of different parallel R-tree construction algorithms on the randomly generated datasets and benchmark datasets. *M* is set to 4 in the experiment.

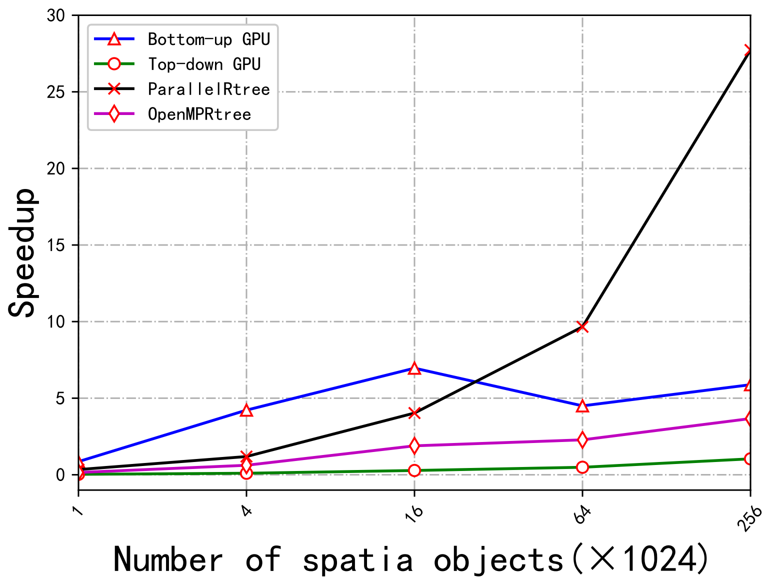

Figure 6. Speedup of different R-tree construction algorithms over sequential R-tree implementation with randomly generated datasets.

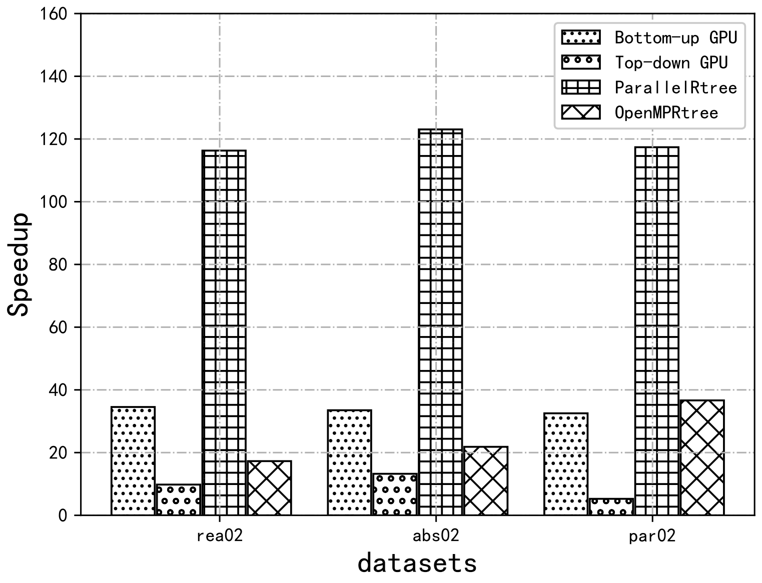

Figure 7. Speedup of different R-tree construction algorithms over sequential R-tree implementation with benchmark datasets.

Figure 8 shows the execution time of different query algorithms. Figure 9 shows the speedup of different parallel query algorithms over sequential query algorithms. These query requests are selected from the randomly generated datasets. Figure 10 exhibits the speedup of different parallel query algorithms over sequential query algorithms. The query requests are selected from the benchmark datasets. It is obvious that as the number of

R-tree query requests increases, the parallelism in the parallel query is enhanced, and thus the speedup increases. Compared with ParallelRtree query and OpenMPRtree query, our query algorithm also has a better speedup.

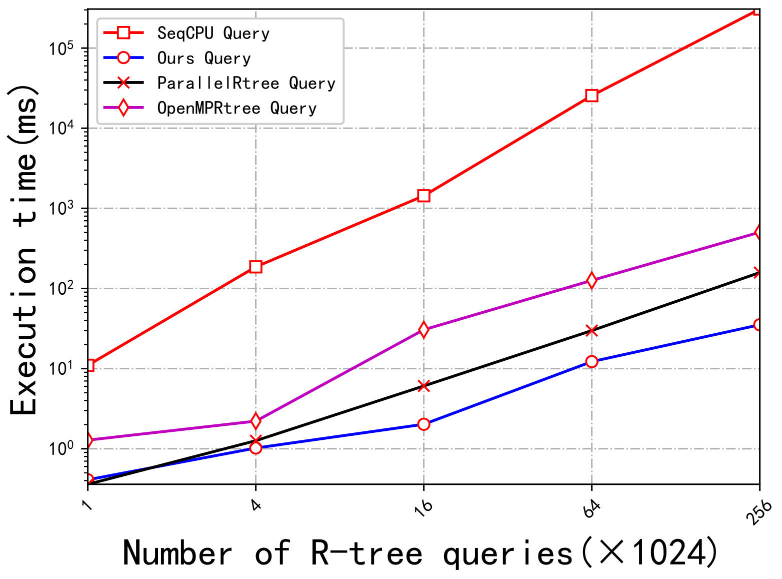

Figure 8. Execution time of different query algorithms with different number of R-tree query requests.

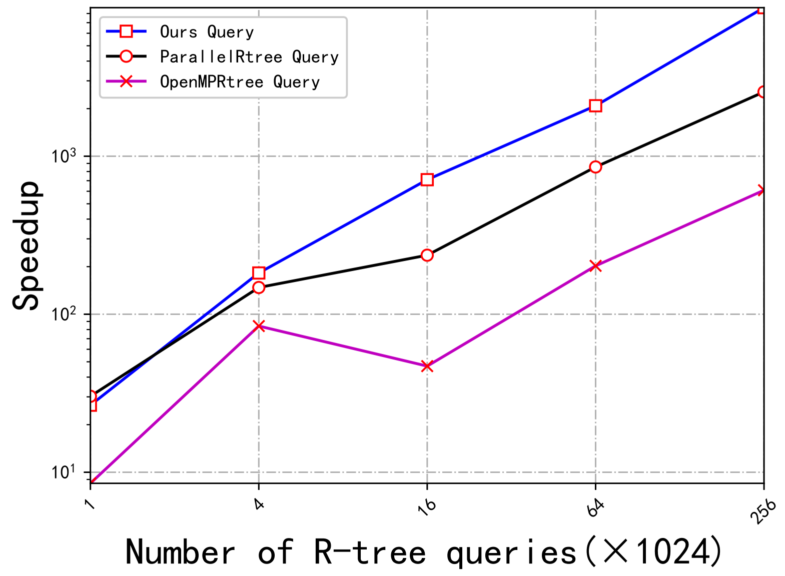

Figure 9. Speedup of different query algorithms over sequential query algorithm with different number of R-tree query requests.

Experiments of vector map overlay systems are carried out with a base polygon set containing 1000 polygons and an overlap polygon set containing various numbers of polygons. The sequential program on CPU is the optimized implementation of the Vatti algorithm [31]. Figure 11 shows the execution time of sequential version and parallel version of vector map overlay system. Figure 12 shows the speedup of a parallel vector map overlay system over other versions of vector map overlay systems. It is shown that, compared with the traditional CPU-based system, our parallel R-tree-based system can achieve nearly 10-fold speedup.

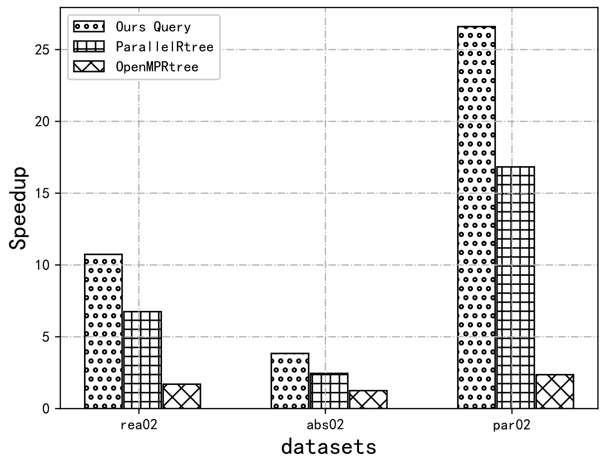

Figure 10. Speedups of different query algorithms over sequential query algorithm with benchmark datasets.

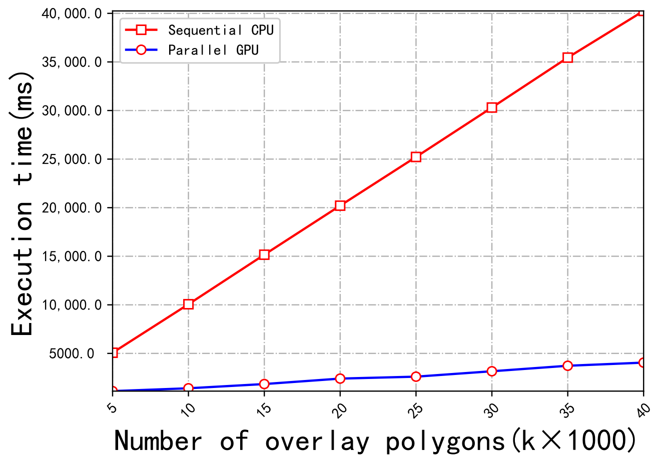

Figure 11. Execution time of sequential version and parallel version of vector map overlay system.

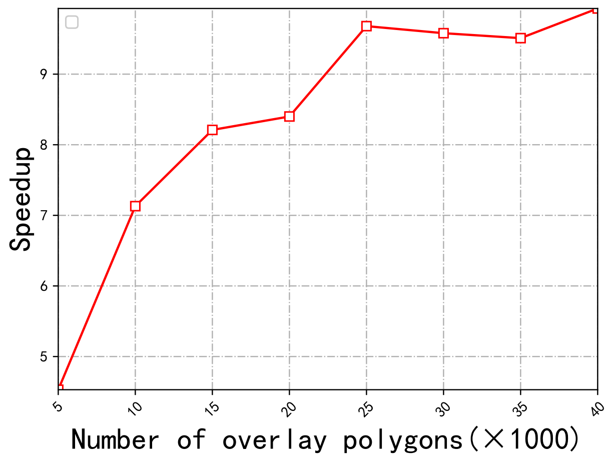

Figure 12. Speedup of parallel vector map overlay system with different number of overlay polygons.

6. Conclusions

This paper discusses the design and implementation of parallel R-tree on GPU. We are particularly interested in the design of R-tree construction and query that can make full use of GPU's computing power. Experiments results are promising, showing that, for example, our R-tree construction on GPU can achieve up to 10 fold speedup compared to the sequential R-tree construction. Based on parallel R-tree on GPU, we also port vector map overlay system from CPU to GPU. Our vector map overlay system on GPU can achieve nearly 10-fold speedup compared to the traditional vector map overlay systems on CPU. In the future, we are interested in exploring multi-GPU parallel schemes. The performance of our vector overlap system can be further improved by segmentation in that it can run efficiently on a cluster of GPUs. In addition, some operations of parallel R-tree on GPU, such as update operation and parallel join, have yet to be implemented and will become the direction of our future research.

Author Contributions: Conceptualization, Formal analysis, Writing—review & editing, J.N.; Methodology, Writing—original draft,Supervision, Funding acquisition, X.H.; Software, Data curation, J.C.; Supervision, Project administration, Funding acquisition, Y.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (Grant Nos. 62262059), the Natural Science Foundation of Guangxi Province (Grant Nos. 2021JJA170178), the Guangxi Innovation-Driven Development Special Fund Project (Guike AA18118036), Special funds for the introduction of high-level talents of Wuzhou University (WZUQDJJ17075).

Data Availability Statement: Data are contained within the article.

Acknowledgments: The authors would like to thank the anonymous reviewers for their contribution to this paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Guttman, A. R-trees: A dynamic index structure for spatial searching. In Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, MA, USA, 18–21 June 1984; pp. 47–57.
2. Sellis, T.; Roussopoulos, N.; Faloutsos, C. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In Proceedings of the 13th VLDB, Brighton, UK, 1–4 September 1987; pp. 507–518.
3. Beckmann, N.; Kriegel, H.; Schneider, R.; Seeger, B. The R*-tree: An efficient and robust access method for points and rectangles. In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, 23–25 May 1990; pp. 322–331.
4. Kamel, I.; Faloutsos, C. Hilbert R-tree: An improved R-tree using fractals. In Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), San Francisco, CA, USA, 12–15 September 1994; pp. 500–509.
5. Luo, L.; Wong, M.D.F.; Leong, L. Parallel implementation of R-trees on the GPU. In Proceedings of the 17th Asia and South Pacific Design Automation Conference, Sydney, Australia, 30 January–2 February 2012; pp. 353–358.
6. You, S.; Zhang, J.; Gruenwald, L. Parallel spatial query processing on gpus using r-trees. In Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, Orlando, FL, USA, 4 November 2013; pp. 23–31.
7. Zhou, K.; Hou, Q.; Wang, R.; Guo, B. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. TOG* **2008**, *27*, 1–11.
8. Hou, Q.; Sun, X.; Zhou, K.; Lauterbach, C.; Manocha, D. Memory-scalable GPU spatial hierarchy construction. *IEEE Trans. Vis. Comput. Graph.* **2010**, *17*, 466–474.
9. Kim, J.; Hong, S.; Nam, B. A performance study of traversing spatial indexing structures in parallel on GPU. In Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, Liverpool, UK, 25–27 June 2012; pp. 855–860.
10. Wehr, D.; Radkowski, R. Parallel kd-tree construction on the GPU with an adaptive split and sort strategy. *Int. J. Parallel Program.* **2018**, *46*, 1139–1156. [CrossRef]
11. Kim, M.; Liu, L.; Choi, W. A GPU-aware parallel index for processing high-dimensional big data. *IEEE Trans. Comput.* **2018**, *67*, 1388–1402. [CrossRef]
12. He, X.; Agarwal, D.; Prasad, S.K. Design and implementation of a parallel priority queue on many-core architectures. In Proceedings of the 2012 19th International Conference on High Performance Computing, Pune, India, 18–22 December 2012; pp. 1–10.

13. He, X.; Wu, Y.; Di, Z.; Chen, J. GPU-based morphological reconstruction system. *J. Comput. Appl.* **2019**, *39*, 2008–2013.
14. Benson, D.; Davis, J. Octree textures. *ACM Trans. Graph. TOG* **2002**, *21*, 785–790. [CrossRef]
15. Deng, Z.; Wang, L.; Han, W.; Ranjan, R.; Zomaya, A. G-ML-Octree: An update-efficient index structure for simulating 3D moving objects across GPUs. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *29*, 1075–1088. [CrossRef]
16. Young, G.; Krishnamurthy, A. GPU-accelerated generation and rendering of multi-level voxel representations of solid models. *Comput. Graph.* **2018**, *75*, 11–24. [CrossRef]
17. Zhou, K.; Gong, M.; Huang, X.; Guo, B. Data-parallel octrees for surface reconstruction. *IEEE Trans. Vis. Comput. Graph.* **2010**, *17*, 669–681. [CrossRef] [PubMed]
18. Vespa, E.; Nikolov, N.; Grimm, M.; Nardi, L.; Kelly, P.H.J.; Leutenegger, S. Efficient octree-based volumetric SLAM supporting signed-distance and occupancy mapping. *IEEE Robot. Autom. Lett.* **2018**, *3*, 1144–1151. [CrossRef]
19. Wen, Z.; He, B.; Kotagiri, R.; Lu, S.; Shi, J. Efficient gradient boosted decision tree training on GPUs. In Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, Canada, 21–25 May 2018; pp. 234–243.
20. Wen, Z.; Shi, J.; He, B.; Chen, J.; Ramamohanarao, K.; Li, Q. Exploiting GPUs for efficient gradient boosting decision tree training. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *30*, 2706–2717. [CrossRef]
21. Lettich, F.; Lucchese, C.; Nardini, F.M.; Orlando, S.; Perego, R.; Tonellotto, N.; Venturini, R. Parallel traversal of large ensembles of decision trees. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *30*, 2075–2089. [CrossRef]
22. Lauterbach, C.; Garland, M.; Sengupta, S.; Luebke, D.; Manocha, D. Fast BVH construction on GPUs. *Comput. Graph. Forum* **2009**, *28*, 375–384. [CrossRef]
23. Meister, D.; Bittner, J. Parallel reinsertion for bounding volume hierarchy optimization. *Comput. Graph. Forum* **2018**, *37*, 463–473. [CrossRef]
24. Ashkiani, S.; Farach-Colton, M.; Owens, J.D. A dynamic hash table for the GPU. In Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, Canada, 21–25 May 2018; pp. 419–429.
25. Alcantara, D.A.; Sharf, A.; Abbasinejad, F.; Sengupta, S.; Mitzenmacher, M.; Owens, J.D.; Amenta, N. Real-time parallel hashing on the GPU. *ACM Trans. Graph. TOG* **2009**, *28*, 1–9. [CrossRef]
26. Lessley, B.; Childs, H. Data-parallel hashing techniques for GPU architectures. *IEEE Trans. Parallel Distrib. Syst.* **2009**, *31*, 237–250. [CrossRef]
27. Kim, M.; Liu, L.; Choi, W. Multi-GPU efficient indexing for maximizing parallelism of high dimensional range query services. *IEEE Trans. Serv. Comput.* **2021**, *5*, 2910–2924. [CrossRef]
28. Xiao, M.; Wang, H.; Geng, L.; Lee, R.; Zhang, X. An RDMA-enabled In-memory Computing Platform for R-tree on Clusters. *ACM Trans. Spat. Algorithms Syst. TSAS* **2022**, *2*, 1–26. [CrossRef]
29. Kamel, I.; Faloutsos, C. On packing R-trees. In Proceedings of the Second International Conference on Information and Knowledge Management, Washington, DC, USA, 1–5 November 1993; pp. 490–499.
30. Healey, R.; Dowers, S.; Gittings, B.; Mineter, M.J. Vector polygon overlay. In *Parallel Processing Algorithms for GIS*; CRC Press: Padstow, UK, 2020; pp. 265–310.
31. Vatti, B.R. A generic solution to polygon clipping. *Commun. ACM* **1992**, *35*, 56–63. [CrossRef]
32. Beckmann, N.; Seeger, B. A Benchmark for Multidimensional Index Structures. Available online: https://www.mathematik.uni-marburg.de/~rstar/benchmark/ (accessed on 23 June 2024).
33. Gutman, T. Free Source Code: Implementations of the R-Tree Algorithm. Available online: https://superliminal.com/sources/ (accessed on 23 June 2024).
34. Malinova, M. Parallelization of the Tree Search Algorithm Using OpenMP. Available online: https://github.com/mmalinova/Parallel_TreeSearch_OpenMP (accessed on 31 May 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.