A GPU-Based System for GIS Polygonal Data Overlay Processing

Xi He, Dinesh Agarwal, Satish Puri and Sushil K. Prasad
Department of Computer Science
Georgia State University
Email: xhe8, dagarwal2, spuri2@student.gsu.edu, sprasad@gsu.edu

***Abstract*—Overlay processing of two or more thematic layers of Geoscience polygonal data is a frequently employed fundamental operation. As opposed to raster data, polygonal data overlaying is irregular and both computation and data intensive. Existing parallelizations have exclusively been over traditional hardware platforms. This work presents the first Geographic Information System (GIS) for overlay processing over GPGPU platforms. This work also has other firsts: To enable efficient identification of the potentially intersecting set of polygons across two input GIS layers, we show how to construct a parallel R-Tree on CUDA architecture and how to efficiently search over it concurrently for thousands of bounding boxes. This involved employing space filling curves for bottom-up construction of R-Tree and non-trivial query bundling and CUDA shared-memory (as opposed to global memory) management for concurrent search. A complex coordination between CPU and GPU was needed to manage dynamic memory allocation for R-Tree construction, search, and for the variable number and size of individual output polygons. We present detailed algorithms and how these were engineered for CUDA . Our experiments also demonstrate good speedups over nVidia's CUDA 1.x and Fermi architectures.**

I. Introduction

In Geographic Information System (GIS), geographic features are normally captured by a specific theme and thus can be organized as a series of thematic layer. To analyze the spatial relationships between sets of geographic features within the same spatial scope, all types of geographic features in layered spatial data-sets are overlaid, which is one of the generic functions in GIS. GIS vector data overlay processing is time-consuming, and in many cases time-sensitive. For emergency response in the US, for example, disaster-based consequence modeling is predominantly performed using HAZUS-MH, a FEMA-developed application that integrates current scientific and engineering disaster modeling knowledge with inventory data in a GIS framework [1]. Depending on the extent of the hazard coverage, datasets used by HAZUS-MH have the potential to become very large, and often beyond the capacity of standard desktops for comprehensive analysis, and it may take several hours to obtain the analytical results.

A parallel system for vector overlay processing is necessary due to its tremendous computation. Although there are some background literatures on parallel/distributed algorithms for vector overlay computation, very little of implementation projects has been done even on traditional parallel/distributed machines. With the emergence of GPGPU (General Purpose Graphics Processing Unit), a powerful computing platform is readily available to scientists and engineers. However, no vector overlay applications on GPGPUs have been ever reported. The SIMT architecture and memory hierarchy on GPGPUs make most of parallel overlay algorithms hard to port, and even though the parallel algorithm can be ported to GPGPUs, the performance is not satisfactory. The motivation of our framework is to provide a complete and efficient solution for vector overlay processing on GPGPUs, and enable other vector overlay related applications on GPGPUs.

The paper's organization is as follows: Section II briefly reviews the related work. After overviewing the framework and briefly introducing its components in Section III, design and implementation details of important components are present in Section IV, V and VI. In Section VII, comprehensive experiments conducted to evaluate the system performance are described. We offer some conclusions and future roadmap in Section VIII.

II. Relate Work

A. Nvidia GPGPUs and CUDA

Modern Nvidia GPGPUs are fully programmable graphic processing units. A GPGPU consists of an array of parallel processors which are often referred to as streaming multiprocessors (SM). For example, in Nvidia's 480 GTX chip, a single SM consists of 32 scalar processors with each scalar processor equipped with certain amounts of registers. Each SM also has 48 KB on-chip memory which has lower access latency and higher bandwidth compared to the global memory which is accessible to every SM, and has larger size. The SMs employ a SIMT (Single Instruction Multiple Thread) architecture. A group of 32 threads called a warp is the minimum execution unit. Once scheduled on a SM, the threads in a warp share the same instruction and can execute in a fairly synchronous fashion.

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed for parallel processing and enables the programmers to access to the instruction sets and memory in the Nvidia GPGPUs. A typical CUDA program is organized into host programs and one or more parallel kernel programs. The host programs are executed on CPU while the parallel kernel programs run on GPGPUs.

There are some performance considerations while programming using CUDA. The first one is the number of threads spawned in a CUDA program. When the instructions executed

by the threads in a warp must wait for a long-latency operation such as the global memory read, other warps are scheduled to be executed. The mechanism of tolerating the latency of expensive operations using work from other warps is often referred to as latency hiding. The mechanism can work due to the fact that the thread management in GPGPUs is implemented in hardware and is extremely efficient. Therefore, a CUDA program should be designed to spawn as many threads as possible so that enough threads are available to hide the latency of expensive operations. The second consideration is the on-chip shared memory provided by the SM. Considering that the shared memory is much faster than the global memory but smaller, it is advantageous if a CUDA program can partition the data in the global memory into pieces that can fit into the shared memory, load these pieces into the shared memory one by one, process them, and then write the results back to the global memory.

B. R-tree construction

R-tree is an important data structures for handling spatial data. It can be used, for example, for range query which retrieve all records whose attributes represent a d-dimensional point located in a given d-dimensional query box. A R-tree node contains certain number of index entries, each of which consists of a MBR (Minimal Bounding Rectangle) and a pointer to spatial objects if it is a leaf node or a pointer to its child node if it is a non-leaf node. An MBR of a two- dimensionalspatial object can be represented by its co-ordinates of top-left corner and bottom-right corner.

In the seminal R-tree paper [2], a R-tree is built by inserting new items iteratively as they arrive. Each insert operation firstly needs to choose a path in accordance with certain R-tree rules and go down to the R-tree leaf-level along this path. Then the new item is added to a R-tree leaf node. If the R-tree leaf node does not have enough space for the new item, it will be spilt into two nodes which will insert a new item into its parent node. Such change might recursively propagate upward along the same path till the root node. If the root node is full, it will be split into two nodes, and a new root node and a new level will be created. To optimize R-tree's search performance, two goals are considered during the construction of R-tree. The first goal is to minimize the area covered by a MBR. The second goal is to minimize the overlap between MBRs. Some R-tree variations, such as R* tree [3], and Hilbert R-tree [4], are designed towards these two goals by employing another heuristic-based splitting algorithms. R+ tree, another R-tree variation clips to-be-inserted rectangle-bounded spatial objects into two and avoid possible overlay between MBRs on the same level. Yet these R-tree construction methods are inherently sequential because these insert operations need to compete to access some R-tree nodess, and the split operation may require the locking of the whole R-tree and prevent concurrent insertions.

Concurrent construction and search over R-tree is an interesting research problem. Some research has been done on concurrent R-tree on multi-core architecture [5]–[8] and distributed systems [9]–[11]. The basic idea for concurrent R-tree on multi-core architecture is that with the help of concurrent control techniques such as lock coupling [6], linking [5] and retrying [12], certain insertion, deletion or search operations can be performed concurrently. Thus the construction of R-tree and its search operation can be accelerated. On the other hand, R-tree on distributed systems aim to improve R-tree's scalability by de-clustering the R-tree nodes on independent computer nodes. Unfortunately, these concurrent R-tree construction and search schemes are not well suited for GPUPU due to the lack of efficient computer locking implementations on GPGPU and GPGPUs' SIMT architecture. The more common way for building R-tree on GPGPU is to built a R-tree on CPU and then copy it to global memory.

In case of analytical application in which all of new items are available a priori, R-tree can also be built by packing techniques. Existing R-tree packing can be classified into two categories: bottom-up packing [13]–[15] and top-down packing [16], [17]. As their names imply, bottom-up packing builds a R-tree from leaf-level to root-level by merging while top-down packing packs a R-tree in a top-down fashion by spliting. These R-tree packing methods motivated our parallel R-tree construction on GPGPU.

Let us suppose we have n spatial objects in a 2-D plane. Let m be the maximum number of MBR entries in an R-tree node and k be the minimum number of MBR entries per node. Then, the number of levels at maximum will be l = $\lceil log_k n \rceil$. Since the number of MBRs per node is bounded, the number of nodes at a given level i can be computed as $\lfloor \frac{n}{k^{l-i}} \rfloor$ assuming root node is at level 0. For example, lets say, we need to construct an R-tree from 100 spatial objects that are approximated by their corresponding minimum bounding rectangles. Let the maximum number of MBR entries in an R-tree node to be 5 and minimum number of entries per node be 3. As such there will be ceiling $\lfloor \log_3 100 \rfloor$ levels. In a bottom up construction approach, the leaf level will be constructed first which is followed by the construction of subsequent non-leaf levels. In this example, the leaf level consists of 100 MBRs. The number of R-tree nodes in the immediate non-leaf level can be calculated as $\lfloor 100/3 \rfloor$ which is equal to 33. Since there are 33 nodes, each of the first 32 nodes will have 3 children MBRs except the last one which will have 4 children MBRs. Similarly, the preceding non-leaf level will have 11 (33/3) nodes. Similarly, the number of nodes in the remaining levels can be calculated.

C. Spatial Search

One of R-tree's functionalities is to support spatial search. Other data structures that can support spatial search include extended K-D-tree [18] and multi-layer grid file [19]. The research of spatial search on GPGPU is a new research area where there are many potential research problems to be addressed. In [20], the author introduces a parallel R-tree search algorithm on CUDA. But the algorithm has some drawbacks. First, one important assumptions in the search algorithm is that the whole R-tree and another two helper tables can be contained in the shared memory. But this assumption is not practical since the size of GPU's shared

memory is very limited, and it is almost impossible to contain a R-tree composed of real data. Second, the algorithm requires a lot of unnecessary work. The time complexity of the search algorithm is $n * logn$, where n is the number of r-tree nodes and $logn$ is the number of r-tree levels. Third, the algorithm is designed for one search operation rather than batch queries, and does not well fit into GPGPU's architecture. [21] is another paper aiming to spatial search on GPGPU. The basic idea of the algorithm for spatial data search in this paper is to scan every spatial object, and compare its bounding box with the query region. Therefore time complexity for a search operation is O(n), which is not good comparing to other O(logn) search operation by R-tree and other methods. The author use one bit as a flag to indicate if a spatial object is within the query region or not, aiming to reduce the size of the output. In other words, a byte with eight bits is used as indicators for eight spatial objects. However, multiple threads might write their results to the same byte of the result flag set array, and cause synchronization problems.

D. Parallel spatial overlay operations

Spatial vector data processing routines are widely used in geospatial analysis. There is only a little research reported in literature on high volume vector-vector or vector-raster overlay processing [**?**]. Since spatial overlay processing depends on the implementations of suboptimal algorithms [22]–[24], the processing costs can vary significantly based on number, size, and geometric complexity of the features being processed [**?**]. There has been extensive research in computational geometry that addressed scalability and parallel or out-of-core computation [25], [26]. Nevertheless, the application of this research in mainstream GIS has been limited [23], [24]. Some research exists for parallel implementations of vector analysis, showing gains in performance over sequential techniques [27]–[29] on classic parallel architectures and models, but none on the modern platforms such as clouds and GPGPUs.

1) Crayons system for spatial overlay operations: With the Crayons system, we have introduced the end-to-end parallel spatial overlay processing over vector data, for the first time. Our goal is to make Crayons available at all possible platforms, and eventually on hybrid and/or heterogenous architectures so that the GIS scientists can leverage the best of our work without any need for system upgrade. In addition to our implementation of Crayons system on GPGPUs, we have already implemented the Crayons system on a variety of other platforms, such as Microsoft's Azure cloud platform [**?**], [**?**], a Linux cluster [**?**] using message passing interface (MPI), and the work is under progress on porting it using Hadoop. Our results for each and every platform have been promising when compared with the state-of-the-art commercial overlay processing systems.

III. An Overview of the Framework

As a CUDA-based application, our framework heavily relies on efficient collaboration of CPU and GPU. Considerable effort is spent on determining the assignment of tasks on CPU or GPGPU and therefore the storage location of related data structures when designing the framework. In general, GPU is well suited to parallel tasks while CPU is designed to run sequential or control tasks. CPU is responsible for controlling the program flow, and for coordinating the host and kernel functions to form a complete application. For our GIS overlay processing framework, there are basically two tasks. One is to identify each pair of potentially intersecting polygons and the other is to perform GIS overlay operations on each of these pairs. Obviously the latter task can be easily parallelized and should be executed on GPGPU side. The first task, however, is worthy of more discussion. On one hand, the sorting-based algorithm (Algorithm 1) for this task can be readily executed in parallel, but its cost is $O(nlog^2n)$. On the other hand, the R-tree based algorithm (Algorithm 2) has a better cost of $O(logn)$. However, it is tricky to parallelize this algorithm and there are no existing CUDA implementation.

Algorithm 1 Create intersection graph by sorting [30]

Input: S_o: overlay polygons; S_b: base polygons
Output: Intersection graph;

```
 1: procedure CREATEINTERSGRAPH_A
 2:     Sort S_o based on X co-ordinates of bounding boxes in parallel
 3:     parfor each base polygon B_i in set S_b do
 4:         Find S_x ⊂ S_o such that B_i intersects with all
 5:         elements of Sx over X co-ordinate using binary search and sorting.
 6:         for each overlay polygon O_j in S_x do
 7:             if B_i intersects O_j over Y co-ordinate then
 8:                 Create Link between O_j and B_i
 9:             end if
10:         end for
11:     end parfor
12: end procedure
```

Algorithm 2 Create intersection graph by R-tree [2]

Input: S_o: overlay polygons; S_b: base polygons
Output: Intersection graph;

```
 1: procedure CREATEINTERSGRAPH_B
 2:     Create a R-tree of S_o in parallel
 3:     parfor each base polygon B_i in set S_b do
 4:         Find S_x ⊂ S_o such that B_i intersects with all
 5:         elements of Sx over X and Y co-ordinate by searching on the R-tree.
 6:         for each overlay polygon O_j in S_x do
 7:             Create Link between O_j and B_i
 8:         end for
 9:     end parfor
10: end procedure
```

Algorithm 1 and 2 are two alternative algorithms to create an intersection graph. The sequential time complexity of Algorithm 1 is $O(n^2 * logn)$ while Algorithm 2 have time complexity of $O(n * logn)$. Based on these comparison, Algorithm 2 is employed although R-tree construction algorithms on many-core architecture have not been reported. Figure 1 is the architecture of the framework. The framework consists

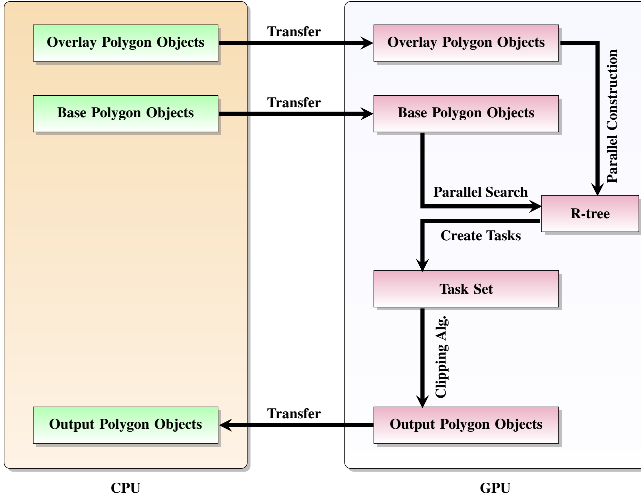

Fig. 1: Architecture of the framewok

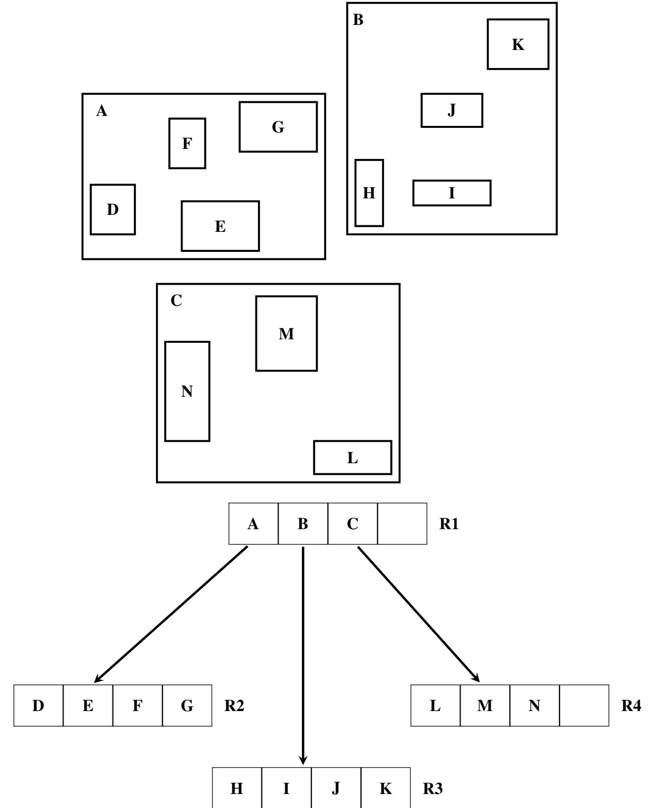

Fig. 2: A two-level R-tree

of three major components: components for parallel R-tree construction, parallel R-tree search and polygon clipping. Firstly, a CPU program reads GIS map data, i.e base polygons and overlay polygons from GML files or shape files, and transfers these data from CPU to GPGPU. Then a concurrent R-tree is constructed on GPU based on overlay polygons. For each base polygon, we will conduct a concurrent R-tree search over these overlay polygons, and find out the overlay polygons that will potentially intersect with the given base polygon. We identify a pair of such base polygon and overlay polygon as a task. Finally, for each task we run a classic overlay algorithm and calculate the output polygons. These output polygons are transferred back to CPU and written to GML files or shape files. We now describe our key contribution on how to construct a R-Tree in parallel, followed by how to search into it concurrently.

IV. Concurrent R-tree Construction

In a R-tree, the *i*th MBR of a non-leaf node is the union of all MBRs of the node's *i* child. The union of MBRs is done in each node by computing the minimum bounding rectangle that spatially contains its children. Since, we know the left top and right bottom coordinates of each child, we can use a naive sequential method to merge children MBRs in which we take two children MBRs at a time and replace it by a smallest rectangle which spatially contains both of them and we repeat this step for all the children. In other words, the construction of parent nodes relies on that of child nodes. This dependence determines that the parallelism we can exploit is on the construction of R-tree nodes on the same R-tree level. For example, Figure 3 is a simple two-level R-tree in which MBR *A, B, C* contains MBR *D, E, F, G*, MBR *H, I, J, K* and MBR *L, M, N*, respectively, and Nodes *R1, R2, R3* need to built prior to node *R0*. The concurrent R-tree construction algorithm we present in this paper is a bottom-up construction, building a R-tree from leaf-level nodes to root node.

One of our considerations for building a R-tree is the quality of the resulting R-tree. As discussed in the above section, to improve R-tree search performance, it is crucial to minimize the area covered by MBRs and the overlap between MBRs. The basic idea in our R-tree construction is based on the heuristic that if R-tree can be built by grouping spatially neighboring MBRs, the area of resulting MBRs and overlap between these MBRs can be greatly minimized, and a R-tree with good search performance can be built. In mathematical analysis, a space-filling curve is a curve whose range contains an D-dimensional hypercube. A one-to-one relationship can be developed between every point in a space-filling curve and every hypercube in D-dimension space. In other words, a space-filling curve can be used to “linearize” a D-dimension space. Considering the locality property of the space-filling curve, we can determine the neighborhood of a D-dimensional hypercube by its corresponding point in the space-filling curve. If we consider a MBR as a D-dimension point or a D-dimensional hypercube, the space-filling curve is well suited for identifying the neighboring MBRs.

The R-tree in global memory is stored in array-based structures instead of traditional pointer-based structures in order to benefit from GPGPUs' characteristics such as global memory coalescing. As shown in Figure 3, we have four arrays *left, right, bottom, top* for collecting all of the MBRs in either leaf nodes or non-leaf nodes. These MBRs are arranged in the order of their enclosing nodes. We also designed two array *start, end* to record the starting index and ending index of MBRs of each R-tree node. For the R-tree in Figure3, MBRs *A–N* are stored in arrays *left, right, bottom* and *top*. The *i*th

Algorithm 3 Space-filling Curve Based R-tree Construction

Input: Filename;
Output: R-tree;

1: **procedure** CONCURRENTCONSTRUCTION(Filename)
2: Read data objects' bounding boxes from a file.
3: Copy these bounding boxes to global memory
4: Compute the number of MBRs for each R-tree level.
5: **parfor** $i = 1 \rightarrow num1$ **do** ▷ $num1$ is the num1 of bounding boxes
6: Compute the ith bounding box's Z-order value.
7: **end parfor**
8: Sort these bounding boxes on their Z-order in parallel.
9: **parfor** $i = 1 \rightarrow num2$ **do** ▷ $num2$ is the num of R-tree levels
10: Compute the MBRs of R-tree nodes in the ith level.
11: **end parfor**
12: **end procedure**

Algorithm 4 Sequential R-tree search algorithm

Input: A query rectangle R: Rect; A R-tree node T: Rnode;
Output: Rectangle objects: an array of Rectangle.

1: **procedure** SEQSEARCH(R, T)
2: **if** $T.ifLeaf$=false **then**
3: **for** $i = 1 \rightarrow T.numOfRects$ **do**
4: **if** $T.rects[i]$ overlaps with R **then**
5: SEQSEARCH($R, T.children[i]$)
6: **end if**
7: **end for**
8: **else**
9: **for** $i = 1 \rightarrow T.numOfRects$ **do**
10: **if** $T.rects[i]$ overlaps with R **then**
11: Output $T.rects[i]$
12: **end if**
13: **end for**
14: **end if**
15: **end procedure**

element in the *start, end* array represents the *i*th R-tree node. For example, the second elements in the *start, end* array is 4 and 7, which indicates that the R-tree node R2 contains the *4*th to the *7*th MBRs.

Besides, we have two arrays on CPU side to facilitate the R-tree construction. These two arrays are used to store the number of R-tree nodes in each R-tree level.

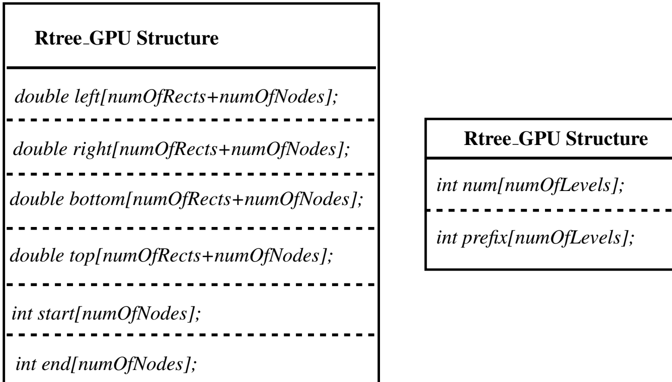

Fig. 3: Data structures for R-tree on CUDA

Left, right, bottom, top array for storing MBRs

1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*	*N*

Start, end array

1	2	3	4
1	*4*	*8*	*12*
3	*7*	*11*	*14*

Fig. 4: Data structures for the sample R-tree

Algorithm 3 shows the space-filling curve based R-tree construction. The algorithm first determines the number of R-tree levels and the number of MBRs in each R-tree level, which in turn determine the number of kernel calls the algorithm will then invoke. Then the algorithm reads bounding boxes of polygons, and calculate the Z-order value (or morton number) for each bounding box in parallel. The technique for efficiently calculating morton number is described in [31]. Next, all of bounding boxes are sorted in parallel on their morton number. Once done, the algorithm starts to built the R-tree layer by layer.

V. PARALLEL R-TREE SEARCH

Tree based structures can be traversed either in the breadth-first manner or the depth-first manner. The sequential search algorithm for R-Tree shown in Algorithm 4 and Figure 5 starts the traversal at the root and in a depth-first manner visits all the leaf nodes during traversing. The result of traversal is a list of those spatial objects that have MBRs that can overlap with the search rectangle specified as input. Since the route of the algorithm is data-dependent and hence sequential, parallelizing the search algorithm is a non-trivial task.

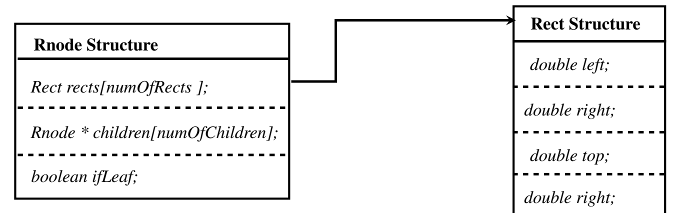

Fig. 5: Data structures for the sequential R-tree.

In addition to the irregular computations for search, the architectural characterisitcs of GPGPUs increase the complexities of the parallel R-Tree search algorithm. Based on the size of the input search rectangle, in the worst case, R-Tree might return all of the records as result. However, the resources such

Algorithm 5 Parallel R-tree search algorithm

Input: Query rectangles $Q[numOfQueries]$: Rect; R-tree root node R: Rtree_GPU;
Output: An array of Rect type objects.

1: **procedure** PARALLELSEARCH(Q, numOfQuerys, R)
2: PARALLELSEARCHA($Q, numOfQuerys, R$)
3: **for** $i = 1 \to 2$ **do**
4: PARALLELSEARCHB($Rect, numOfQuerys, T$)
5: **end for**
6: **for** $i = 3 \to numOfLevels$ **do**
7: PARALLELSEARCHC($Rect, numOfQuerys, T$)
8: **end for**
9: **end procedure**

as device's global memory are limited and thus cannot be reserved for the worst case. Dynamic memory allocation is the traditional way out of this problem but for GPGPUs the dynamic memory allocation process adds significant overhead.

In order to introduce parallelism while tolerating the overheads associated with parallelizing the search process in the R-tree search algorithm, instead of a single query request multiple requests are executed at any given time. Considering the dependency between a R-tree node and its children nodes, a breadth-first search is preferred. At each level multiple threads search for the winning routes that are yet to be traversed at the levels below. **However, threads need to synchronize after finishing the process at current level and before moving to the next level. Considering that our search algorithm is carried on each level of the R-tree, the memory usage for the next level of the R-tree thus can be bounded by computing the search results of current level.**

In Nvidia GPGPUs, the access to data in shared memory is about 10 times faster than that in global memory. It will greatly boost performance if we can refine the data access pattern and cache frequently used data in shared memory. For the upper levels of a R-tree, it is quite possible that a R-tree node will be accessed by quite a few queries. So it is worthy to load a R-tree node to shared memory, cluster the queries that will be performed on the same node and concurrently perform these queries. However, for the lower levels of a R-tree where the number of R-tree nodes become large, loading the R-tree node to shared memory does not necessarily improve search performance. Based on these analysis, we design three search routines aiming to root level, the first two level and the other levels of R-trees, respectively. Algorithm 5 is the main routine of our parallel search algorithm, in which Algorithm 6, 7, 8 will be executed sequentially.

Algorithm 6 is designed for the query on the root level. It first loads the MBRs in the root node into shared memory, then test the query rectangles with each of these MBRs. If one query rectangle overlaps with any of these MBRs, the index of the query rectangle and MBRs will be wrapped and stored in a Tmp_Output structure. (See Figure 6) The output of Algorithm 6 is input to Algorithm 7 in which the Tmp_Output objects are sorted on the query index, and are merged into

Algorithm 6 Parallel R-tree search algorithm A

Input: Query rectangles $Q[numOfQueries]$: Rect; A R-tree node R_0: Rtree_GPU;
Output: An array of Tmp_Output objects.

1: **procedure** PARALLELSEARCHA(Q, R)
2: Load R into shared memory
3: **parfor** $i = 1 \to$ numOfQueries in parallel **do**
4: **parfor** $j = 1 \to (end[R0] - start[R0] + 1)$ in parallel **do**
5: **if** jth MBR overlaps with $Rect[i]$ **then**
6: Output Tmp_Output(i,j);
7: **end if**
8: **end parfor**
9: **end parfor**
10: **end procedure**

Algorithm 7 Parallel R-tree search algorithm B

Input: An array of Tmp_Output objects, arr1; A R-tree node R: Rtree_GPU
Output: An array of Tmp_Output objects, arr2;

1: **procedure** PARALLELSEARCHB(arr1, arr2, R)
2: Sort arr1 on the query index in parallel
3: Combine elements with the same query index in arr1 into one Tmp_Cluster Object.
4: **parfor each** Tmp_Cluster object **do**
5: $index \leftarrow$ Tmp_Cluster.indexOfRnodes
6: Load R[index] into shared memory
7: **parfor** $i = 1 \to$ numOfQueries in parallel **do**
8: **parfor** $j = 1 \to (end[R[index]] - start[R[index]] + 1)$ in parallel **do**
9: **if** jth MBR overlaps with $Rect[i]$ **then**
10: Output Tmp_Output(i,j);
11: **end if**
12: **end parfor**
13: **end parfor**
14: **end parfor**
15: **end procedure**

Tmp_Cluster objects. The R-tree node specified in each of Tmp_cluster object will be loaded into shared memory, and be tested with each query. In Algorithm 8, the query specified in each Tmp_Output object is tested with the R-tree node in the same Tmp_Output object.

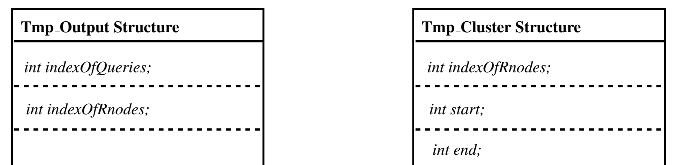

Fig. 6: Data structures for the parallel search.

Algorithm 8 Parallel R-tree search algorithm C

Input: An array of Tmp_Output objects, arr1; A R-tree node R: Rtree_GPU
Output: An array of Tmp_Output objects, arr2;

```
1: procedure ParallelSearchC(arr1, arr2, R)
2:     parfor each element in arr1 do
3:         index ← arr1.indexOfRnodes
4:         parfor i = 1 → (end[R[index]] − start[R[index]] + 1) in parallel do
5:             if ith MBR overlaps with Rect[i] then
6:                 Output Tmp_Output(i,j);
7:             end if
8:         end parfor
9:     end parfor
10: end procedure
```

VI. Polygon Clipper

Another important component in our framework is the polygon clipper module that takes a pair of polygons and the overlay operation as input, overlays them based on the operation, and finally returns a set of resulting polygons. The polygon clipping algorithm being employed by the polygon clipper is based on the algorithm by Vatti [32]. Vatti's algorithm can handle general as well as complex polygons such as concave and self-intersecting polygons. The gist of Vatti's algorithm is to scan every vertex in a set of polygons, starting at the bottom to the top, and compute potential intersection points between the sets of base polygons and overlay polygons. Meanwhile, the scanned vertices and identified intersection points are determined based on the fact that they will contribute to output polygons according to their relative positions. Contributing vertices and intersection points are connected together to form the output polygons. The time complexity of this algorithm depends on the number of total vertices in the sets of base polygons and overlay polygons. Since the algorithm relies on bottom-up scanning, it is sequential in nature. General Polygon Clipper library (GPC library) [33] is an open source software written in C that implements and extends Vatti's algorithm. It is employed as the fundamental processing component in the polygon clipper component.

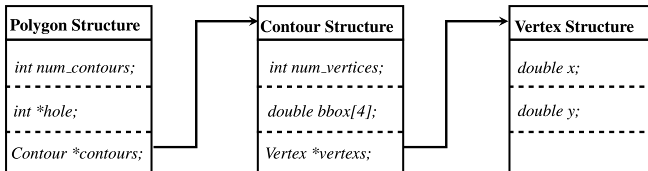

Fig. 7: Data structures used in GPC library.

Figure 7 shows the data structures for the polygons used as the interfaces for the polygon clipper component. These data structures are organized hierarchically, with each polygon containing a set of contours and each contour containing a set of vertices. Notice that the input polygons usually contain only one contour. Table I lists important functions in GPC library. While GPC library is well designed and carefully documented, considerable effort was required to port it to CUDA architecture. GPC library is written in C language, so the functions in the library have to be reengineered to CUDA functions to execute them on CUDA architecture. NVidia GPUs with Compute capability of 1.x do not support recursion, while functions such as *add_to_sbtree()*, *add_edge_to_aet()*, *build_sbt()* employ recursion heavily. We had to rewrite those functions to convert recursion into multiple iterative calls. Moreover, we do not have the luxury of memory allocation routines such as *malloc*, *calloc*, and *realloc* in CUDA. Although in-kernel dynamic memory allocation is supported in Fermi-based GPUS, the built-in memory allocator is not efficient enough for applications that invoke memory allocation routines frequently. After comparing a few parallel memory allocators, we chose Xmalloc [34]—an in-kernel dynamic memory allocator for GPUs–for our framework. Xmalloc employs lock-free algorithms and hierarchical cache-like buffer. Xmalloc scales well with growth in both the number of processors and the number of vector units in each SIMD processor.

Function name	Description
gpc_polygon_clip	Main function
build_lmt	Build a local minima list to store bounds for every polygon
insert_bound	Insert bounds to the local minima list
add_to_sbtree	Insert edges to the scan-beam tree
build_sbt	Build a scan-beam table
add_edge_to_aet	Add edges starting at the local minimum to the active edge list
build_intersection_table	Build intersection table for the current scan-beam

TABLE I: Important functions in GPC library

VII. Experiment

Performance with different data set, and comparison with other implementation, such as cloud platform implementation and MPI implementation.

VIII. Conclusion

References

[1] "Hazus website." [Online]. Available: http://www.fema.gov/plan/prevent/hazus/
[2] A. Guttman, *R-trees: A dynamic index structure for spatial searching*. ACM, 1984, vol. 14, no. 2.
[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, *The R*-tree: an efficient and robust access method for points and rectangles*. ACM, 1990, vol. 19, no. 2.
[4] I. Kamel and C. Faloutsos, "Hilbert r-tree: An improved r-tree using fractals," 1993.
[5] V. Ng and T. Kameda, "The r-link tree: A recoverable index structure for spatial data," in *Database and Expert Systems Applications*. Springer, 1994, pp. 163–172.
[6] Y. Chen *et al.*, "A study of concurrent operations on r-trees," *Information sciences*, vol. 98, no. 1-4, pp. 263–300, 1997.
[7] V. Ng and T. Kameda, "Concurrent access to r-trees," in *Proceedings of the Third International Symposium on Advances in Spatial Databases*. Springer-Verlag, 1993, pp. 142–161.
[8] M. Kornacker and D. Banks, "High-concurrency locking in r-trees," in *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*. Citeseer, 1995, pp. 134–145.

[9] B. Schnitzer and S. Leutenegger, "Master-client r-trees: A new parallel r-tree architecture," in *ssdbm*. Published by the IEEE Computer Society, 1999, p. 68.

[10] X. Fu, D. Wang, W. Zheng, and M. Sheng, "Gpr-tree: a global parallel index structure for multiattribute declustering on cluster of workstations," in *Advances in Parallel and Distributed Computing, 1997. Proceedings*. IEEE, 1997, pp. 300–306.

[11] S. Lai, F. Zhu, and Y. Sun, "A design of parallel r-tree on cluster of workstations," *Databases in networked information systems: proceedings. Aizu, Japan, December 4-6, 2000*, vol. 1, p. 119, 2000.

[12] A. Biliris, "Operation-specific locking in balanced structures," *Information Sciences*, vol. 48, no. 1, pp. 27–51, 1989.

[13] S. Leutenegger, M. Lopez, and J. Edgington, "Str: A simple and efficient algorithm for r-tree packing," in *Data Engineering, 1997. Proceedings. 13th International Conference on*. IEEE, 1997, pp. 497–506.

[14] N. Roussopoulos and D. Leifker, "Direct spatial search on pictorial databases using packed r-trees," in *ACM SIGMOD Record*, vol. 14, no. 4. ACM, 1985, pp. 17–31.

[15] I. Kamel and C. Faloutsos, "On packing r-trees," in *Proceedings of the second international conference on Information and knowledge management*. ACM, 1993, pp. 490–499.

[16] L. Chen, R. Choubey, and E. Rundensteiner, "Merging r-trees: Efficient strategies for local bulk insertion," *GeoInformatica*, vol. 6, no. 1, pp. 7–34, 2002.

[17] Y. García R, M. López, and S. Leutenegger, "A greedy algorithm for bulk loading r-trees," in *Proceedings of the 6th ACM international symposium on Advances in geographic information systems*. ACM, 1998, pp. 163–164.

[18] J. Chang and K. Fu, "Extended kd tree database organization: A dynamic multiattribute clustering method," *Software Engineering, IEEE Transactions on*, no. 3, pp. 284–290, 1981.

[19] K. Sevcik and N. Koudas, "Filter trees for managing spatial data over a range of size granularities," in *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1996, pp. 16–27.

[20] M. Kunjir and A. Manthramurthy, "Using graphics processing in spatial indexing algorithms," *Research report, Indian Institute of Science, Database Systems Lab*, 2009.

[21] B. Oh, "A parallel access method for spatial data using gpu," *International Journal*, vol. 4, 2012.

[22] J. Bentley and T. Ottmann, "Algorithms for Reporting and Counting Geometric Intersections," *Computers, IEEE Transactions on*, vol. C-28, no. 9, pp. 643 –647, sept. 1979.

[23] B. Chazelle and H. Edelsbrunner, "An optimal algorithm for intersecting line segments in the plane," *J. ACM*, vol. 39, pp. 1–54, January 1992. [Online]. Available: http://doi.acm.org/10.1145/147508.147511

[24] T. M. Chan, "A Simple Trapezoid Sweep Algorithm for Reporting Red/Blue Segment Intersections," in *In Proc. 6th Canad. Conf. Comput. Geom*, 1994, pp. 263–268.

[25] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005.

[26] P. K. Agarwal, L. Arge, T. Mølhave, and B. Sadri, "I/o-efficient efficient algorithms for computing contours on a terrain," in *Proceedings of the twenty-fourth annual symposium on Computational geometry*, ser. SCG '08. New York, NY, USA: ACM, 2008, pp. 129–138.

[27] F. Wang, "A parallel intersection algorithm for vector polygon overlay," *Computer Graphics and Applications, IEEE*, vol. 13, no. 2, pp. 74 –81, mar 1993.

[28] R. G. Healey, M. J. Minetar, and S. Dowers, Eds., *Parallel Processing Algorithms for GIS*. Bristol, PA, USA: Taylor & Francis, Inc., 1997.

[29] J. D. Hobby, "Practical segment intersection with finite precision output," *Computational Geometry*, vol. 13, no. 4, pp. 199 – 214, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925772199000218

[30] D. Agarwal, S. Puri, X. He, and S. K. Prasad, "A system for GIS polygonal overlay computation on linux cluster - an experience and performance report," in *IEEE International Parallel and Distributed Processing Symposium workshops, to appear*, Shanghai, China, May 2012.

[31] "Interleave bits by Binary Magic Numbers," Website, 1997. [Online]. Available: http://graphics.stanford.edu/~seander/bithacks.html#InterleaveBMN

[32] B. Vatti, "A generic solution to polygon clipping," *Communications of the ACM*, vol. 35, no. 7, pp. 56–63, 1992.

[33] Alan Murta, " General Polygon Clipper library." [Online]. Available: http://www.cs.man.ac.uk/~toby/gpc/

[34] X. Huang, C. Rodrigues, S. Jones, I. Buck, and W. Hwu, "Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 1134–1139.