# Data Parallel Quadtree Indexing and Spatial Query Processing of Complex Polygon Data on GPUs

Jianting Zhang
Department of Computer Science
The City College of New York
New York, NY, USA
jzhang@cs.ccny.cuny.edu

Simin You
Dept. of Computer Science
CUNY Graduate Center
New York, NY, USA
syou@gc.cuny.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK, USA
ggruenwald@ou.edu

## ABSTRACT

Fast growing computing power on commodity parallel hardware makes it both an opportunity and a challenge to use modern hardware for large-scale data management. While GPU (Graphics Processing Unit) computing is conceptually an excellent match for spatial data management which is both data and computing intensive, the complexity of multi-dimensional spatial indexing and query processing techniques has made it difficult to port existing serial algorithms to GPUs. In this study, we propose a parallel primitives based strategy for spatial data management. We present data parallel designs for polygon decomposition, quadtree construction and spatial query processing. These designs can be realized on both GPUs and multi-core CPUs as well as future generation hardware when parallel libraries that support the primitives are available. Using a large-scale geo-referenced species distribution dataset as an example, the GPU-based implementations can achieve up to 190X speedups over serial CPU implementations and 14X speedups over 16-core CPU implementations for polygon decomposition, which is the most computing intensive module in the end-to-end spatial data management solution we have provided. For quadtree constructions and spatial range/polygon query modules, which are more data intensive, the speedups over single and multi-core CPUs are up to 27X and 2X, respectively, depending on workloads. Comparing with a similar technique on polygon decomposition that is realized using a native parallel programming language, our parallel primitives based implementation is up to 3X faster on the species distribution dataset. The results may suggest that simplicity and efficiency can be achieved simultaneously using the data parallel design strategy by identifying the inherent data parallelisms in application domains.

## 1. INTRODUCTION

Multi-dimensional indexing is crucial in speeding up spatial query processing. Hundreds of indexing structures have been proposed in the past few decades [1]. However, the majority of existing indexing structures are designed for traditional computing models, i.e., serial algorithms targeted for uniprocessors in a disk-resident system. Meanwhile, as argued in [2], due to the increasing diversity and heterogeneity of the mainstream hardware, it is no longer possible to simply work on top of abstractions provided by either the operating system or by system libraries and hope to achieve high performance automatically. However, hardware sensitive designs are fairly costly and it is very difficult to provide an optimized design for each and every of hardware architectures.

In this study, we aim at exploiting the inherent data parallelisms in processing multi-dimensional spatial data and exploring data parallel designs to achieve high-performance across multiple parallel hardware platforms. We use quadtree indexing and querying on complex polygons, which has many real world applications in Geographical Information Systems (GIS) and Spatial Databases, as a case study to demonstrate the feasibility and efficiency of the proposed techniques. Using real world data and targeting at real world applications of large-scale species distribution data, we evaluate the realizations of our data parallel designs on GPUs.

Our technical contributions are three-fold. First, we develop data parallel designs for indexing and querying complex polygons in multiple overlapped polygonal datasets that can be efficiently realized on GPUs using well-understood and well-supported parallel primitives [3]. Second, we compare our technique on polygon decomposition with a native parallel implementation and demonstrate that our parallel primitives based implementation can achieve both simplicity and efficiency. Third, we apply our technique to a large-scale global species distribution dataset and have achieved considerable speedups over serial implementations on CPUs, which is signficant from an application perspective.

The rest of the paper is arranged as follows. Section 2 introduces background and related work on indexing multi-dimensional spatial data on new hardware, including GPUs. Section 3 presents the application context of our technique and proposes a spatial database approach to managing large-scale species distribution data. Section 4 provides details on data parallel designs of spatial indexing and query processing techniques on GPUs. Section 5 presents experiment results on the proposed techniques using the 4000+ bird distribution range maps in the West hemisphere [4] on GPUs. Section 6 provides comparison with the PixelBox algorithm [5] that targets at a similar technical context but uses a different parallelization strategy on GPUs. Section 7 discusses several high-level design, implementation and application issues. Finally, Section 8 is conclusion and future work directions.

## 2. BACKGROUND AND RELATED WORK

Spatial data processing is known to be both data and computing intensive [6]. Various techniques, such as minimizing disk I/O overheads in spatial indexing [1] and the two phase filter-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at ADMS'14, a workshop co-located with The 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.

refinement strategy in spatial joins have been proposed [7]. The increasingly available new hardware, such as inexpensive Solid State Drives (SSDs), large memory capacities, multi-core CPUs, and many-core GPUs, have significantly changed the cost models on which traditional spatial data processing techniques are based. Developing new indexing and query processing techniques and adapting traditional ones to make full use of new hardware features are active research topics in spatial data processing in the last few years. A generic framework for flash aware trees is proposed in [8]. The TOUCH technique performs in-memory spatial join by developing a hierarchical data oriented partitioning [9]. Furthermore, a comprehensive analysis of iterated spatial joins in main memory has been provided through extensive experiments [10]. MapReduce/Hadoop based techniques have been proposed to achieve higher scalability for spatial warehousing [11] and geometry computation [12].

Compared with techniques for processing point data, it is technically more challenging to efficiently index and query "complex" polygons. Real world polygons, even for those that are defined as mathematically "simple" polygons, can have complex data structures. For example, according to Open Geospatial Consortium (OGC) Simple Feature Specification (SFS) [13], a simple polygon may have one outer ring and many (including 0, 1 or 1+) inner rings. Determining the spatial relationships between a quadrant and a polygon with multiple rings, which is fundamental in quadtree-based indexing, is much more complex than processing polygons with a single ring. Furthermore, complex polygons may overlap and there might be multiple polygons intersect with a single quadrant. Traditional Minimum Bounding Rectangle (MBR) based spatial indexing techniques are not likely to be efficient for significantly overlapped polygons due to decreased spatial discrimination power. This is because MBRs of overlapped polygons are likely to have higher degrees of overlap. In addition, many geometric algorithms that are used in the filtering phase of spatial join processing [7] have at least linear time complexity with respect to the number of polygon vertices. As real world polygons can have large numbers of vertices and a few of them in a dataset may have extremely large numbers of vertices, the filtering phase can be computing intensive, incurs long runtimes and difficult to parallelize due to load unbalancing.

In this study, we refer multi-ring and potentially highly overlapping polygons as "complex" polygons although they are still considered "simple" mathematically. It is intuitive to rasterize each polygon as a binary raster to speed up spatial queries. The QUILT geographical information system [14] developed more than two decades ago was based on region quadtrees where linear quadtree nodes are used to represent polygons after rasterization and support various queries. Linear quadtrees are also used to index polygon MBRs so that leaf quadtree nodes can also be indexed by B+ trees based on their Space Filling Curve (SFC [1]) codes in disk-resident databases. However, while the computing overhead to generate linear quadtrees from binary rasters and MBRs are light, directly generating quadrants from polygons can be expensive which makes it desirable to utilize parallel hardware to speed up the process. In this study, we aim at indexing polygon internals in-memory and utilizing parallel processing units for high performance. Our techniques represent complex polygon as sets of independent quadrants that can be manipulated collectively in parallel at different granularities (quadtree levels). By decomposing complex polygons into large numbers of simple quadrants (squares in geometry), the sharp boundary between spatial filtering and spatial refinement using MBRs in traditional spatial joins is now multi-level and can be easily adjusted based on applications and/or system resources at runtime. The increased data parallelisms make our techniques more parallelization friendly on massively data parallel GPUs.

In this study, we extensively utilize parallel primitives wherever possible to exploit data parallelisms and achieve portability among multiple hardware platforms, including GPUs. The strategy is significantly different from traditional approaches that program parallel hardware using their native programming languages directly. Here parallel primitives refer to a collection of fundamental algorithms that can be run on parallel machines, such as map/transform, sort, scan, and reduction [3]. The behaviors of popular parallel primitives on one dimensional (1D) arrays or vectors are well-understood and are well-supported in multiple parallel platforms. Despite there are inevitable parallel library overheads, very often using parallel primitives that have been highly tuned for different hardware achieves better performance than native parallel programs. In our previous works on GPU-based spatial data management, including grid-file based point data indexing [15], min-max quadtree based raster data indexing [16], point-to-polyline distance based Nearest Neighbor (NN) spatial join [15] and point-in-polyline test based spatial join [17], we have extensively explored parallel primitives based designs and implementations with encouraging good performance. However, they have not been compared with native parallel implementations. This study targets at a more complex spatial data management problem, i.e., indexing complex polygon internals and speeding up spatial queries (Section 4 and 5). We also compare our data parallel technique for the most computing intensive step in indexing with a similar published technique using a native parallel implementation (PixelBox, [5]) (Section 6). We hope our study can stimulate the discussions on seeking effective ways, with respect to both efficiency and productivity, in utilizing GPUs for domain specific applications.

In the context of indexing polygon internals to speed up spatial query processing, we note that Microsoft SQL Server Spatial adopted a similar hierarchical decomposition of space strategy and used B+ tree to index the decomposed polygons [18]. Spatial query processing is then based on the symbolic ancestor-descendent relationships of the identifiers of decomposed quadrants which is typically much faster than testing spatial relationships based on geometric computation on polygon vertices. However, the algorithm in tessellating polygons into quadrants, which is the key to the performance of polygon indexing, was not well-documented. Although our experiments have shown that the polygon indexing module in SQL Server 2012 release is able to utilize multiple CPU cores, it is unclear how the parallelization is achieved and whether it is possible to extend it to many-core GPUs efficiently.

Another closely related work is the GPU-based PixelBox algorithm on intersecting two polygons derived from high resolution biomedical images [5]. Although similar geometrical principles in determining whether a box or quadrant intersects with a polygon are used in both PixelBox and our technique on polygon decomposition for spatial indexing, PixelBox intersects two polygons and computes their intersection area at the same time while our technique is designed to decompose individual polygons. There are also several additional key differences between the two techniques. First, when multiple polygons are involved, our technique decomposes each polygon exactly once and can reuse the resulting quadrants whereas needed. In contrast, PixelBox would require pair-wise

intersections among multiple polygons. Although PixelBox meets its design requirement for its targeted application domain well, where only pair-wise intersections on image-derived single-ring polygons are needed, it is not efficient in more general cases (such as our species distribution data management) when multiple polygons are involved in intersections simultaneously. Second, PixelBox is implemented natively using CUDA on Nvidia GPUs while our technique is based on parallel primitives. Despite that the CUDA implementation has been extensively optimized as reported in [5], experiments have shown that our technique is not only simpler in design but also performs up to 3X better. The comparisons are provided in Section 6.

In addition to adapting traditional linear quadtree techniques [1] to parallel computing on GPUs, our technique is also related to rasterization on parallel hardware for rendering purposes in computer graphics. Efficient parallel techniques to rasterize triangles into pixels are cornerstones of high-performance computer graphics and have been extensively researched. A recent study by Nvidia researchers has shown that software rasterization is within a factor of 2-8X compared to the hardware graphics pipeline on a high-end GPU [19]. While it is interesting to apply these software rasterization techniques for spatial indexing and query processing in a data management context, we argue that there are mismatches between the two application domains which may render the software rasterization techniques less applicable in spatial data processing. First and foremost, rasterization techniques for computer graphics are optimized for triangles and cannot be used to process real world complex polygons directly. In fact, the GL_POLYGON primitive defined by OpenGL does not guarantee the correctness of rendering concave polygons, in addition to being much slower than GL_TRIANGLES. While some tessellation and triangulation algorithms and packages are available to decompose complex polygons to simple polygons or triangles, they may not be supported by hardware and are left for serial software implementations. It is non-trivial to parallelize such implementations on GPUs with high performance. Furthermore, while it is possible to identify uniform quadrants from rasterized pixels efficiently, the ultimate goal of software rasterization in computer graphics is to generate pixel values for triangles that are visible from current views with visually acceptable resolution. In contrast, our goal is to decompose polygons in a spatial database to speed up query processing that guarantees pre-defined numeric accuracy.

## 3. A Spatial Database Approach to Managing Large-Scale Species Distribution Data

Historically, species range maps represent fundamental understanding of the observed and/or projected distributions of species. While only a limited number of species are documented with reasonably accurate range maps throughout human history, several enabling technologies have made biodiversity data available at much finer scales in the past decade [20], including DNA barcoding for species identification and geo-referring for converting descriptive museum records to geographical coordinates. The increasingly richer biodiversity data has enabled ecologists, biogeography researchers and biodiversity conservation practitioners to compile species range maps from multiple sources with increasing accuracies. For example, NatureServ has published range maps of 4000+ birds in the west hemisphere [4] with 700+ thousand complex polygons and more than 77 million vertices in ESRI Shapefile format [21]. While these datasets are useful for visualization purposes and for scientists that are specialized in a small subset of species to examine the data manually, it is highly desirable to manage such data in a database environment to allow queries across a large number of species and understand the relationships between global and regional biodiversity patterns and their underlying environments. Figure 1 illustrates the potential queries among taxonomy, geography and environment [22].

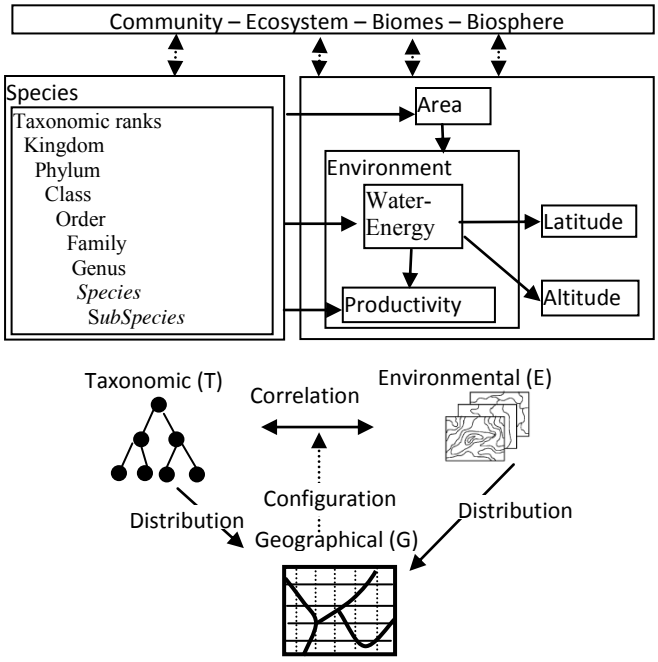


Figure 1 Illustration of Potential Queries on Species Distribution Data

```
SELECT aoi_id, sp_id, SUM (ST_AREA (inter_geom))
FROM (
      SELECT aoi_id, sp_id,
            ST_INTERSECTION (sp_geom,qw_geom) AS inter_geom
      FROM SP_TB, QW_TB
      WHERE ST_INTERSECTS (sp_geometry, qw_geom)
       )
GROUP BY aoi_id, sp_id
HAVING SUM(ST_AREA(inter_geom)) >T;
```

Figure 2 A Spatial Query on Species Distribution Data in SQL

Given the numbers of species, ecological zones and environmental variables, among the virtually countless queries, a fundamental one is to retrieve the list of species and their distribution areas within Region of Interests (ROIs). Given the species range map stored in table SP_TB (sp_id, sp_geom) and the ROIs stored in table QW_TB (roi_id, roi_geom), where sp_geom and roi_geom represent geometrical objects (polygons and rectangles, respectively), the spatial query can be formulated as the SQL statement listed in Figure 2 according to OGC SFS specification [13]. We note that the OGC SFS specification has been largely adopted by SQL/MM and implemented in major commercial and open source spatial databases, e.g., Microsoft SQL Server Spatial [18] and PostgreSQL/PostGIS [23]. The WHERE clause (using ST_INTERSETS boolean function) in Figure 2 serves as an optimization trick to reduce the number of calls to the ST_INTERSECTION function for intersecting two

polygons which is very expensive for complex polygons. The optional HAVING clause can be used to set the threshold to prevent from including resulting intersected polygons that are too smaller, such as "sliver polygons" that appear along the borders of the two intersecting polygons. Similarly, species distribution polygons can also be used as ROIs to query environmental variables. Although the exact query syntax may be different for spatially querying environmental variables in vector format (e.g., rain gauge observations) and in raster format (e.g., satellite imagery), they can be formulated in a similar spatial query processing framework [23].

For complex polygons with large number of vertices and multiple holes, the query shown in Figure 2 may incur very long response times. Our previous experiments have shown that even a single simple rectangular ROI (e.g., spatial range/window query), when used to query against the bird range map in a PostgreSQL/PostGIS database, may incur more than 100 seconds [24]. This makes interactive explorations of the dataset impractical. We have developed techniques to decompose polygons into quadrants in an offline manner in order to speed up online query processing in both a disk-resident database (PostgreSQL/PostGIS) [24] and a memory-resident database environment [25]. While the online query performance is satisfactory for both systems (with the help of parallelization on the disk-resident system through query window decomposition [24]), it took a long time for offline processing using sophisticated geospatial software (e.g., GDAL [26]) which is too slow for large-scale data. In this study, we aim at speeding up polygon decomposition by utilizing the increasing computing power on parallel hardware. Different from previous techniques that extensively use recursions and dynamic memory for both polygon decompositions and quadtree constructions which make them very difficult to parallelize, as detailed in the next section, our new data parallel designs using parallel primitives make them portable across multiple hardware platforms and easy to scale to large numbers of processing units. Experiment results are provided in Section 5 and comparisons with a similar technique are reported in Section 6.

# 4. Data Parallel Designs on Quadtree Indexing and Spatial Query Processing

Compared with manipulating data structures with regular access patterns (such as arrays and matrices), deriving irregular data structures (such as quadtrees) from complex polygons with multiple layers of variable structures on GPUs is technically challenging. As discussed in Section 2, PixelBox [5] provides a native CUDA based design for computing the area of the intersection of two single-ring polygons. Despite that multi-ring polygons are not supported and the output is only a scalar value (area), the implementation is already very sophisticated. Our technique aims at supporting multi-ring polygons (which is mandatory in our applications) and deriving quadtree structures for indexing complex polygons. We next present our data parallel designs for the three major modules in our technique, i.e., decomposing polygon into quadrants at multiple levels, constructing Multiple Attribute Quadtrees (MAQ-Tree [25]) and spatial queries on quadtrees, by identifying data parallelisms in each module. Steps in each parallel design are then mapped to well-understood and well-supported parallel primitives. Simple loops on top of these parallel primitives may be needed to handle multiple tree levels, which are supported by the host language of the parallel library being used. We refer to [3] for excellent introductions to parallel primitives and the Thrust library website[1] for the exact syntax of parallel primitives that are being used in this study.

## 4.1 Polygon Decomposition

While it is possible to rasterize polygons into binary rasters and then use the technique similar to our previous work reported in [25] to identify linear quadtree nodes and construct quadtrees, rasterizing complex polygons to fine resolution grid cells incur significant computing and storage burden. Furthermore, it is very difficult if not impossible to parallelize polygon rasterization using fine-grained data parallelisms on GPUs. Our technique adopts a top-down approach by performing quadrant-polygon intersection tests in a level-wise manner. The top-down approach allows stop at any level based on user specification and/or available system resources. The level-wise processing can accumulate sufficient quadrant-polygon pairs across multiple polygons and utilize large number of processing units (e.g., GPU cores) more effectively. While our data parallel designs using parallel primitives are significantly different from PixelBox [5] as discussed previously, we apply a similar set of computational geometry principles for quadrant-polygon test which is the building block for polygon decomposition on a polygonal dataset with a large number of polygons. We next present the procedure (Figure 4) for testing the relationship (left of Figure 3) between a single quadrant-polygon pair before we introduce a data parallel procedure (Figure 5) to decompose multiple polygons in a dataset into multiple quadrants at different levels. An example illustrates a decomposed polygon is shown in the right part of Figure 3.

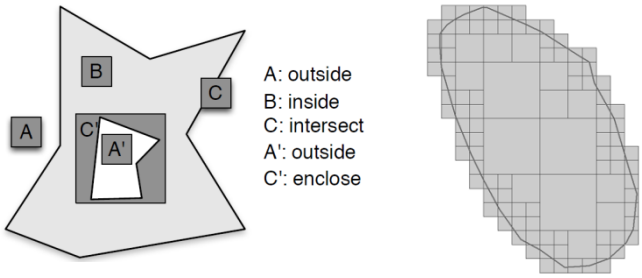


Figure 3 Five Relationships between a Quadrant and a Polygon with a Hole (left) and An Example of Decomposed Polygon

As shown in Figure 3, among the three possible relationships between a quadrant and a polygon (inside/intersect/outside), when a polygon has holes, a quadrant can be outside the polygon if it is inside one of the holes. In Figure 4, function *isEdgeIntersect* (line 2) is first used for testing whether any of the polygon edges intersects with the quadrant (type C) by checking whether any of polygon edges (lines) intersect with the quadrant. This is equivalent to line-rectangle intersection test which is well defined in computational geometry. If none of the polygon's edges intersects with the quadrant, function *pointInRing is* applied to both the outer ring and all the inner rings for further tests. The classic ray-tracing algorithm for point-in-polygon test [27] can be used for implementing function *pointInRing*. We first test whether there is any edge of the outer ring intersects with the input quadrant; if so, the procedure immediately terminates with type C (Line 3). If none of them intersects with the quadrant, vertices of the quadrant should be either all inside or outside the ring.

[1] https://github.com/thrust/thrust

Therefore, by performing a point-in-polygon test on one vertex of the quadrant and the outer ring (Line 4), we will know whether all vertices of the quadrant are inside/outside the outer ring. Notice that we start from the minimum enclosing quadrant of the polygon being processed to ensure that there is no chance for subsequent lower level quadrants to enclose the outer ring. As such, if the point in polygon test at Line 4 is negative, the quadrant must be outside the polygon (type A). If a quadrant is completely inside the outer ring, we still need to verify its relationship with each inner ring (Line 6~11). Recall that there may be 0, 1 or 1+ inner rings in a complex polygon (c.f. Section 2). When an inner ring encloses the quadrant, such quadrant is then outside the polygon (type A'). On the other hand, even if all the vertices of the quadrant are outside of all inner rings, the quadrant can be either complete inside the polygon (type B) or intersect with the polygon (type C'). To distinguish these two cases, we can simply test whether any point of an inner ring falls within the quadrant (Line 9). Since quadrants are special rectangles (squares), testing whether a point is in a quadrant is fairly straightforward. Assuming a polygon has *N* points, the complexity of determine whether any edge intersects with the quadrant is O(*N*). Meanwhile, the point-in-polygon test using the ray-tracing algorithm is also O(*N*) [27]. Thus, the total complexity of function *QPRelationTest* is O(*N*).

```
Input: quadrant Q, polygon P
Output: relation type A, B or C
QPRelationTest(Q, P)
1.  V = vertex(Q) //get one of the vertex of Q
2.  intersect = isEdgeIntersect(Q, P) //quadrant intersection test
3.  if (intersect)  return C;
4.  inOuterRing = pointInRing(V, P.outer_ring)
5.  if(!inOuterRing) return A; //outside of outer ring
6.  for each IR in P.inner_rings do:
7.      inInnerRing = pointInRing(V, IR)
8.      if (inInnerRing) return A //inside a inner ring
9.      if (any point of IR is in Q)
10.         return C //quadrant encloses a ring
11.  end for
12.  return B //inside
```

Figure 4 Algorithm on Testing Relationship between Quadrant and Polygon

Based on the relationships we have defined previously (Figure 3), quadrants of a polygon can be generated in a top-down manner by iteratively testing relationship among refined quadrants and the polygon. The procedure is illustrated in Figure 5. We first generate the minimum enclosing quadrant based on the MBR of the polygon and then split the quadrant into four child quadrants. The child quadrants are subsequently tested against the polygon. If a quadrant is completely outside or within the polygon (e.g., type A and B in Figure 5), we stop the decomposition process on such quadrant and output a leaf quadrant for the polygon. Otherwise, the quadrant either intersects with the polygon or encloses a ring of the polygon (type C). We then test whether such quadrant reaches the predefined maximum level (MAX_LEV in Figure 6) and decide whether it needs to be further processed.

The parallel primitives based GPU implementation is presented in Figure 6 where the input is a vector of complex polygons (for parallel processing) and the output is a vector of quadrants. Note that some parallel primitives may take functional objects (functors) as parameters. All threads that are assigned to process elements in input polygon vector will execute the functor in parallel, where parameters of the functors are extracted from the input vector by the underlying parallel library dispatcher.

In Line 1, from a vector of polygons, we generate their corresponding minimum enclosing quadrants in parallel and store them in a vector **VQ**. Each element of **VQ** is a tuple in the format of (*polygon_id*, *z_val*, *lev*), where *polygon_id* is an identifier to locate the input polygon, *z_val* and *lev* represent the Morton code [1] and the quadrant level, respectively. Given an MBR, the level of the quadrant being processed can be calculated as *lev*=__clz(z1^z2)/d where z1 and z2 are the Morton codes of the top-left and lower-right corners, ^ is the bit-wise XOR operation, d=2 is the number of dimensions and *__clz* is the intrinsic function available on CUDA-enabled GPUs for calculating the number of leading zeros in an integer (similar intrinsic function is available for other hardware architectures). The Morton code of the quadrant code (*z_val*) is calculated as z1 with the lower 2**lev* bits set to zeros if *lev* is larger than 0. In particular, *lev* =0 denotes the minimum enclosing quadrant is the root of the quadtree being constructed. Examples for computing the *lev* value and the Morton codes for two MBRs are shown in Figure 7. Note that we use 4-bit words (W=4) in the __clz function in the figure for illustration purpose while our GPU implementation uses 32-bit words (W=32). The design is loop free and can be implemented as a functor to work with a map/transform parallel primitive.

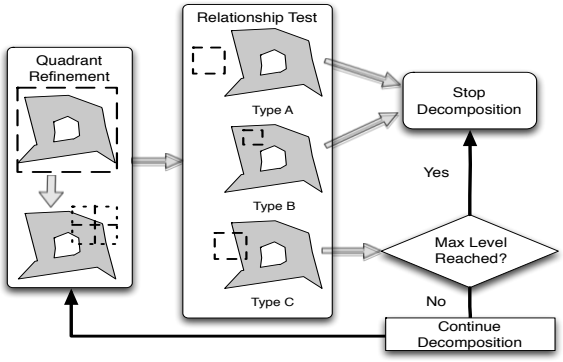


Figure 5 Illustration of Indexing Polygon Internals through Iterative Polygon Decomposition

```
Input: vector of polygons Ps
Output: vector of quadrant pairs Qs
ParallelPolygonDecomposition(Ps, Qs):
1.  VQ = transform(Ps) //generate minimum enclosing
                       quadrants from MBRs of Ps
2.  While (VQ.size() > 0 or TempQ.size() > 0):
3.    if (VQ.size() > MAX_CAPACITY):
4.      copy out-of-capacity items from VQ to TempQ
5.    if (VQ.size() == 0 and TempQ.size() > 0):
6.      copy  items from TempQ to VQ
7.    NextVQ = split(VQ) //split is a combination of scatter,
                          scan and transform primitives
8.    Status = transform(NextVQ, QPRelationTest) //for each
                       quadrant, a QPRelationTest is performed
9.    sort(Status, NextVQ) // sort NextVQ based on Status
// if Status is set to either leaf node or MAX_LEV is reached
10.   Qs = copy_if(NextVQ, Status, MAX_LEV)
//Otherwise
11.   VQ = copy_if(NextVQ, Status, MAX_LEV)
12.  return Qs
```

Figure 6 Algorithm Polygon Decomposition

Lines 2~11 in Figure 6 consist the major part of the whole procedure of polygon decomposition. Since that GPU memory is limited comparing with CPU memory, we use a temporary vector TempQ in CPU memory to hold workload when it exceeds the predefined MAX_CAPACITY threshold (Lines 2-6). The threshold is set based on the size of available GPU memory. In Line 7, **VQ** is split into four sub-quadrants at the next level and saved to **NextVQ** (to be detailed next). After the split, each new quadrant is tested with its corresponding polygon, and the relationship test results are saved in a vector called **Status** (Line 8). Line 9 sorts **NextVQ** based on **Status** before we can copy the quadrants to **Qs** which stores the output quadrant (Line 10) or to **VQ** for the next iteration (Line 11).

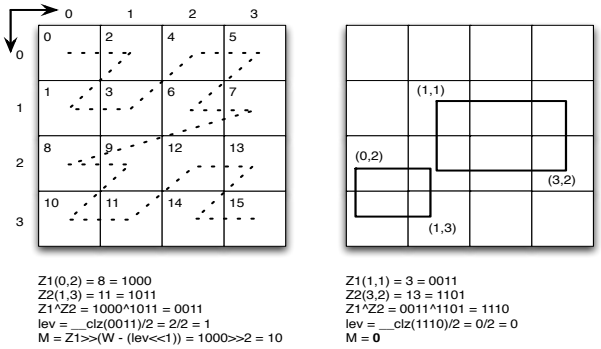


Figure 7 Examples of Extracting Minimum Enclosing Quadrants from Polygon MBRs

The split procedure used in Line 7 of Figure 6 is illustrated in Figure 8 where the two upper shaded quadrants need to be split. At the first step, a vector of 4s is set up and an exclusive scan is used to generate write positions for input data. The second step writes input data in **VQ** to **NextVQ** using a scatter primitive according to the previously generated positions followed by an inclusive scan primitive to fill the rest of the vector. The scan is implemented by using "maximum" as its functor and we call it as "inclusive maximum scan". A transform primitive is lastly used to generate the Morton codes ($z\_val'=4*z\_val+\{0,1,2,3\}$) and levels ($lev'=lev+1$) in parallel. The offset (0-3) for each quadrant in Morton code calculation can be easily derived from CUDA thread identifiers, which does not require additional space in the GPU implementation.

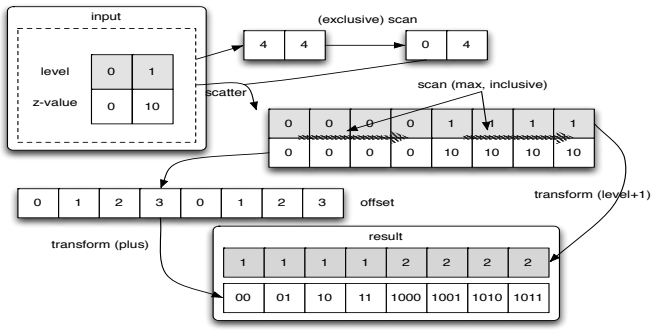


Figure 8 Illustration of Data-Parallel Split Procedure Using Parallel Primitives

## 4.2 Quadtree Construction

We do not keep the parent-child relationships among quadrants in the polygon decomposition module although we could have done so. The most important reason in the decision is that keeping such relationships in a data parallel computing setting is much more cumbersome than in a serial computing setting and we want to simplify the implementation of polygon decomposition module as much as possible. Furthermore, a quadrant may be covered by multiple polygons and we would like to group polygons based on quadrant identifiers. This is not possible during polygon decomposition as quadrants are grouped based on polygon identifiers there. However, after quadrants corresponding to leaf nodes are identified by the polygon decomposition module, constructing a quadtree from quadrants of a set of polygons can be accomplished by chaining parallel primitives as in the polygon decomposition module (Section 4.1). We note that, while quadrants identified from a single polygon do not overlap (classic quadtree structures where typically there is no information to be associated with intermediate nodes), quadrants identified from a set of polygons may overlap and polygon identifiers may be associated with intermediate tree nodes. Our previous work on constructing such an extended quadtree from a large number of overlapped polygon datasets on CPUs, termed as Multi-Attribute Quadtree or MAQ-Tree [25] is illustrated in the top of Figure 9 where the tree is constructed dynamically in CPU memory. Experiments have shown that storage overhead of a MAQ-Tree can be much smaller than storing individual quadtree or combined quadtree using classic quadtree representation by pushing down identifiers to leaf nodes [25]. In addition, window query on MAQ-Trees is more efficient than traversing multiple individual classic quadtrees each representing a polygon datasets with non-overlapping polygons.

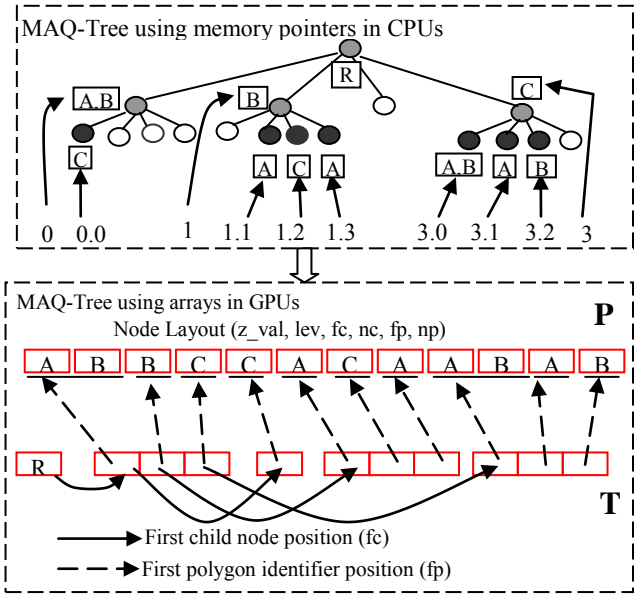


Figure 9 Illustration of MAQ-Tree Structure Using Memory Pointer (Top) and Array (Bottom) Representations

We have developed an array representation that is suitable for GPUs by extending the GPU-based BMMQ-Tree proposed in our previous work [16]. As illustrated in the bottom of Figure 9, for each node in the GPU-based MAQ-Tree, in addition to the quadrant identifier (*z_val*), level (*lev*), First Child Position (*fc*) and Number of Child Nodes (*nc*) fields as in BMMQ-Tree, two additional fields, i.e., First Polygon Identifier Position (*fp*) and Number of Polygon Identifier Position (*np*) are

added. Note that *fc* and *fp* are shown in all the tree nodes at the bottom part of Figure 9 while *z_val*, *lev*, *nc* and *np* are not shown due to space limit. The functionality of the two array offsets fields, i.e., *fc* and *fp*, are equivalent to memory pointers in CPUs. However, they can be computed in parallel (to be detailed next) and do not need memory allocations which are expensive on GPUs. The algorithm to construct a MAQ-Tree using parallel primitives is listed in Figure 10. The input (**Qs**) is a vector of leaf quadrants with their corresponding polygon identifiers that are generated in the previous module. We use tuple (*z_val*, *lev*, *p_id*) to represent a leaf quadrant. The output of the algorithm will be a GPU-based MAQ-Tree that consists of a tree structure array (**T**) and a polygon identifier array (**P**), as illustrated in the bottom of Figure 9.

Input: leaf quadrants **Qs** where each element is **(z_val, lev, p_id)**
Output: MAQ-tree (**T, P**) where **T** is in the format of **(z_val, lev, fc, nc, fp, np)** and **P** is a vector of polygon identifiers

ParallelConstructMAQTree(**Qs**):
1. *stable_sort* **Qs** by *z_val*
2. *stable_sort* **Qs** by *lev*
//**UQs** is in the format of (*z_val*, *lev*, *np*, *fp*)
3. (**UQs**.*z_val*, **UQs**.*lev*, **UQs**.*np*) = *reduce* **Qs** by (*z_val*, *lev*)
4. **UQs**.*fp* = *exlusive_scan*(**UQs**.*np*)
5. *copy* **Qs**.*p_id* to **P**
//count the size of quadrants at each level
6. (*lev*, *lev_size*) = *reduce* **UQs** by *lev*
//compute the start position for each level
7. *lev_pos* = *exlusive_scan*(*lev_size*)
8. *copy* last level quadrants from **UQs** to **T**
9. *current_lev* = MAX_LEV
//level-wise iteration starts
**10. while (*current_lev* > 0)**
11. *current_lev* = *current_lev* – 1
12. *transform* and *reduce* quadrants in **T** at *current_lev*+1 to *current_lev* and save in **TempQs**
13. *copy (append)* quadrants at *current_lev* from **UQs** to **TempQs**
14. *sort* and *unique* **TempQs**
15. *reduce (by key)* using *z_val* as the key to compute **TempQs** ***nc***
16. scan on **TempQs.nc** to compute **TempQs.fc**
17. *copy* **TempQs** to **T**
18. return (**T, P**)

Figure 10 Algorithm MAQ-Tree Construction

Lines 1-4 in Figure 10 group polygon identifiers that are associated with quadrants, and compute the positions of the first polygon identifiers (*fp*) associated with each unique quadrant identifier based on (*z_val*, *lev*). Note that quadrants at different levels may have the same Morton codes based on the algorithms discussed in polygon decomposition module. This is also the reason that we use the combination of Morton code and quadrant level as the key in Line 3. After this step, we can get a vector of unique quadrants (**UQs**) where each item contains the Morton code (*z_val*), level (*lev*), the first polygon identifier position (*fp*) and the number of polygon identifiers (*np*). As indicated in Line 4, *fp* can be computed from *np* by using an exclusive scan parallel primitive, as *fp*[i]=sum(*np*[j]) for j=0..(i-1) by the definition of a scan primitive [3].

Next, the polygon identifiers array is then saved to **P** (Line 5), so that a quadrant in the tree can easily look up its related polygons by using *fp* and *np*, i.e., all polygon identifiers that are associated with the quadrant are stored at the position *fp*..(*fp*+*np*-1) of array **P**. Lines 6-7 sort quadrants by levels and generate level boundaries to keep track of the number of quadrants at each level. We first copy the last level quadrants to the tree (Line 8) and process tree nodes in a bottom up manner (Line 10-15). To generate a new level, say *current_lev*, there are two major components. The first component directly comes from the Morton codes of leaf quadrants generated during polygon decomposition. With the level information derived at Line 6 and 8, we can easily locate leaf quadrants at *current_lev* and copy them to a temporary space (**TempQs**). The other component comes from the reduction of lower level quadrants, i.e., the quadrants at *current_lev* + 1. Those quadrants are reduced to remove duplications before they are appended to **TempQs** (Line 12-13).

To maintain the parent-child links between two consecutive levels in a quadtree using an array representation, *fc* (first child position) and *nc* (number of children) fields of all tree nodes need to be set appropriately. Similar to computing *np* and *fp* as discussed above, computing *nc* and *fc* can be realized by chaining a sort, a segmented reduction and an exclusive scan parallel primitive (Line 14-16). The last step during an iteration is to append **TempQs** (using a copy primitive) to the tree structure **T** (Line 17 in Figure 10). The iteration will continue at a higher level until the root of the tree is constructed. The alert reader might ask what would be the *np* and *fp* values of the non-leaf nodes as they may be created level-wise bottom-up in the loop. The answer is that, for a non-leaf tree node created in Line 12, we check whether the corresponding quadrant is already in **Qs**. If not, then the non-leaf node is just a "via node" in the tree and is not associated with any polygon identifiers. We set both *np* and *fp* to a negative number as an indication. If the quadrant corresponding to the none-leaf node is already in **Qs**, our algorithm makes sure that, the combination functor in the *reduce* primitive in Line 12 only updates *nc* and *fc* while keep *np* and *fp* unchanged. Actually, the check logic can be easily implemented in the combination functor by checking the signs of *np* and *fp* of the two input tree nodes to be combined when the *reduce* primitive is invoked to process all tree nodes at the level in parallel.

## 4.3 Spatial Query Processing

Once a MAQ-Tree is constructed, it can be used to speed up spatial queries by traversing the tree in either a breadth-first search (BFS) or a depth-first search (DFS) manner. As processing a single query on trees with limited depth on modern hardware (including both CPUs and GPUs) are typically fast, it is more beneficial to process multiple queries on GPUs to make full use of its computing power. Our previous work on parallel R-tree based batched queries on GPUs [28] showed that BFS generally performs better than DFS on GPUs. As such, in this work, we have chosen to implement parallel quadtree based batched queries on GPUs using BFS. We propose techniques for two types of spatial queries, including batched range (or window) query and polygon query where a query is defined by a polygon. Clearly, polygon queries are more generic but are more complex. Using the polygon decomposition techniques discussed in Section 4.1, complex query polygons can be decomposed into quadrants with different sizes. As such, both types of queries can be supported using a unified design and implementation.

### *4.3.1 Parallel Batched Range Query*

The problem of batched range query is to answer a set of range queries in parallel and locate all intersecting quadrants for each individual range query. A naïve approach to parallelize batched query is to assign a thread to process a range query and queries are processed independently. However, such design can easily incur significant load unbalancing and uncoalesced memory accesses on GPUs, which is likely to result in poor performance.

The key idea of our fine-grained data parallel design is to process a query batch using BFS and redistribute workload within an iteration. As shown in Figure 11, the workload is represented as query pairs where each pair consists of a query id (*query_id*) and a quadtree node in quadtree **T**. The main process of the query algorithm in Figure 11 is from Line 3 to Line 15. For all pairs, rectangle-quadrant intersection tests are performed in parallel and the results are saved to **Status** (Line 8). In Line 9, **W** is reordered based on **Status**, where the first part of **W** contains pairs that need to be processed in the next iteration and the size is denoted as *new_size*. Based on **Status**, intersected pairs in **W** that need to be output will be copied to the result vector (Line 11). The next step is to expand the first *new_size* pairs of **W** that need be processed in the next iteration. Note that the "expand" operation in Line 13 is almost identical to the "split" operation first introduced in polygon decomposition (Section 4.1, c.f. Figure 8) with a slight difference, i.e., the number of items to be expanded at next level. The number is always 4 in polygon decomposition but varies based on *nc* (number of children of the tree node) in batched query processing. During the process, the memory consumed by workload **W** might exceed the GPU device memory capacity. Our solution is to use a temporary space (**TempW**) allocated in CPU memory to offload out-of-capacity pairs (Line 5), which will be copied back to GPU when needed (Line 7).

```
Input: Query windows Q, Quadtree T
Output: intersected pairs (query_id, quadrant_id)
ParallelRangeQuery(Q, T):
1.    generate query pairs W = (query_id, T.root)
2.    size = Q.size()
3.    while (size > 0 or TempW.size() > 0):
4.      if (size > MAX_CAPACITY):
5.        copy out-of-capacity pairs to TempW
6.      if (size == 0):
7.        copy workload from TempW to W
8.      Status = transform(W, IntersectionTest)
        //first new_size pairs will be further processed
9.      sort W according to Status
10.     size = new_size
11.     copy_if W to Result based on Status
12.     if (size == 0) continue;
        //quadrants are expanded for next iteration
13.     NextW = expand(W, size)
14.     W = NextW
15.     size = W.size()
```

Figure 11 Parallel Batched Range Query

### *4.3.2 Parallel Polygon Query*

In addition to range query, we also support queries that are defined by polygons instead of rectangular windows, which we call *polygon query*. Such types of query are very useful in two scenarios. Firstly, visual analytics, where user defined a query by drawing a polygon, and secondly, to serve as an advanced spatial filtering for spatial joins on polygons. Instead of filtering based on rectangular MBRs of polygons, we may build a MAQ-Tree on one polygon dataset and use the other one as query polygons. As the MAQ-Tree represents polygons being queried more accurately than the MBRs of the polygons, it can be more effective in spatial filtering with fewer false positives that need to be refined in the refinement phase in spatial joins [7]. Since polygons are decomposed into quadrants rather than arbitrary rectangles as in range queries discussed in Section 4.1, an optimization can be done is to replace intersection test of two rectangles with bit operations over the Morton codes of two quadrants (c.f., Figure 7).

# 5. EXPERIMENTS

We use a real large-scale dataset to validate the designs and test the efficiency of the implementations. The dataset consists of 708,509 polygons of 4062 bird species distribution range maps in the West Hemisphere [4]. The total number of polygon vertices in the dataset is 77,699,991, i.e., roughly 110 vertices per polygon. We divide the original dataset into four groups based on numbers of vertices as shown in Table 1. Here we essentially treat the four groups of datasets as four separate datasets to test the scalability of our proposed techniques. All the experiments are performed on a workstation equipped with two Intel Xeon CPUs (at 2.60 GHz, 16 physical cores in total), 128 GB DDR3 memory and an Nvidia GTX Titan GPU. The operating system is CentOS 6.4 with GCC 4.7.2, TBB 4.2 and CUDA 5.5 installed. All the codes are compiled using O3 optimization. We use the Thrust parallel library that comes with Nvidia CUDA SDK when parallel primitives are used in our GPU implementations. All runtimes are reported in milliseconds and are based on the average of 5 runs, unless otherwise stated.

Table 1 Statistics of Bird Species Range Map Datasets

| Polygon Group | num of vertices range | total num of polygons | total num of points |
|---|---|---|---|
| 1 | 10-100 | 497,559 | 11,961,389 |
| 2 | 100-1,000 | 33,374 | 8,652,278 |
| 3 | 1,000-10,000 | 6,719 | 20,436,931 |
| 4 | 10,000-100,000 | 1,213 | 33,336,083 |

## 5.1 Performance on Polygon Decomposition

We implemented our proposed parallel decomposition algorithm described in Section 4.1 on Nvidia GPUs using Thrust library. A serial implementation using only one CPU core is adopted as the baseline (termed CPU-Serial) where polygons are decomposed iteratively. Since Thrust allows compiling its code to multi-core CPUs using TBB [3] as the backend, we use the TBB implementation for multi-core CPUs (termed as CPU-TBB). We performed experiments on polygon decomposition using different maximum quadtree levels (i.e., MAX_LEV), ranging from 12 to 15, to understand how the implementations perform under different workloads. Since data transfer times are insignificant, they are excluded from reporting.

The runtimes of polygon decompositions using different experiment settings are plotted in Figure 12. In subplots for dataset group 3 and 4, some CPU implementations cannot complete in reasonable time and are excluded. Figure 12 shows that our GPU implementations outperform all CPU counterparts at all quadtree levels. For dataset group 1, 43X-75X speedups are measured over the serial CPU implementations and 3.5X-7.1X speedups are measured over the multi-core CPU implementations (TBB, 16 CPU cores). The speedups are higher for dataset groups 2, 3 and 4, which are 107X-190X and 11.1X-14.6X, respectively. The speedups for dataset group 1 are lower than the other three dataset groups might be due to the fact that the dataset group has a large number of small polygons and the ratio of data accesses to computation may be too high to saturate GPU computing power as the other three groups do. The speedups clearly demonstrate the efficiency of polygon decompositions on GPUs by taking advantages of their excellent floating point computing power as well as high memory bandwidth. Figure 12 also suggests that, while CPU serial implementation for dataset group 4 already takes more than four hours at quadtree level 13 and becomes infeasible

for higher quadtree levels, our GPU implementation only takes about 7 minutes at quadtree level 15. The high efficiency is desirable for indexing complex polygons with large numbers of vertices, such as those in group dataset 4.

To help better understand the scalability of the data parallel design for polygon decomposition, the numbers of the resulting quadrants at the four quadtree levels in the four groups of datasets are plotted in Figure 13. As expected, the numbers of the resulting quadrants grow exponentially as the quadtree levels (MAX_LEV) increase which also explains that the runtimes of the three categories of implementations (CPU-serial, CPU-TBB and GPU) increase exponentially with the quadtree levels as observed in Figure 12. Note that the Y-Axis in both Figure 12 and Figure 13 uses a logarithmic scale. From Figure 13 we can also see that, while dataset group 1 has larger number of polygons and larger number of total points than group 2 (Table 1), the resulting numbers of quadrants in group 2 is much larger than those of dataset group 1. The runtimes are largely determined by the resulting numbers of quadrants, not the numbers of input polygons or their total numbers of vertices. The results are consistent with our design where each quadrant needs to test its relationship with the polygon that its parent quadrant intersects. The number of tests and hence the runtimes are generally proportional to the total numbers of quadrants that are being tested at each quadrant level ranging from level 0 (root) to MAX_LEV.

## 5.3 Performance on Quadtree Construction

Since the four groups of polygon datasets used in the polygon decomposition module may produce similar numbers of quadrants, it is not suitable to use the same data to test the scalability of the design and implementation of quadtree construction. As such, we have combined all the computed quadrants and randomly select $2^{16}$ to $2^{21}$ quadrants for testing purposes. We repeat the random sampling process four times and report average runtimes for the 6 sampling tests. Note that the reported runtimes are end-to-end and include times to transfer data among CPUs and GPUs for GPU-based implementations. The results are plotted in Figure 14.

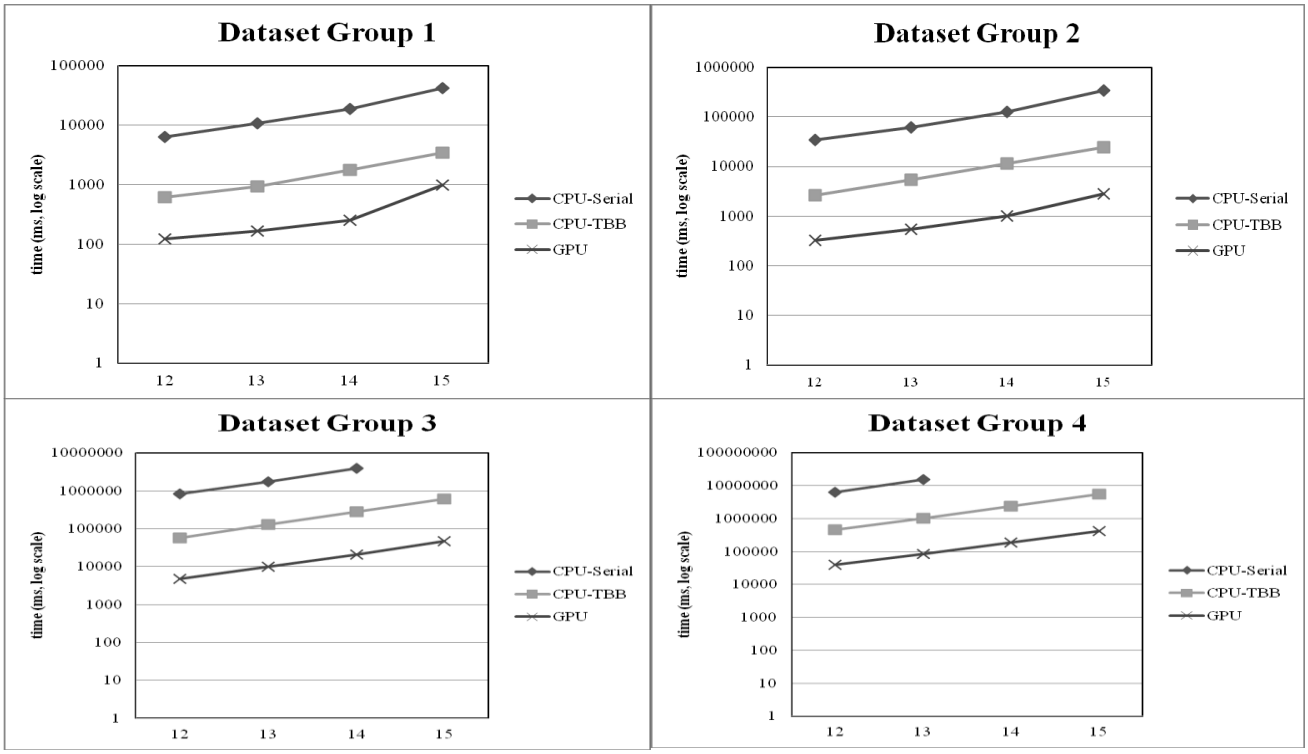


Figure 12 Runtimes of the Four Dataset Groups Using Four Quadtree Levels

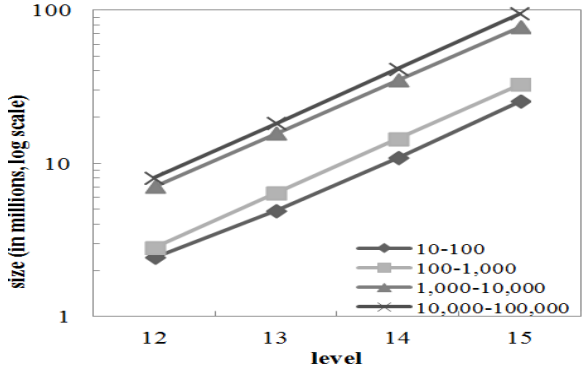


Figure 13 Sizes of Generated Quadrants in Four Groups

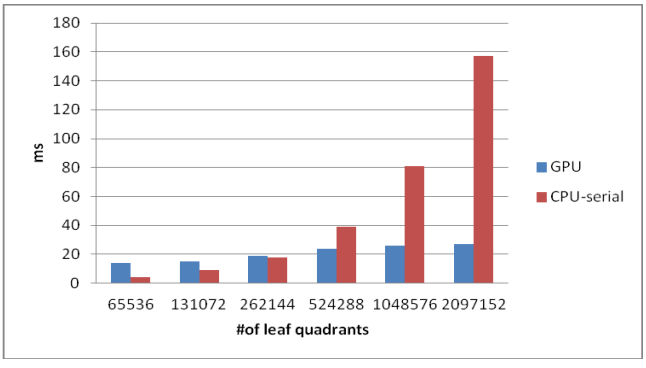


Fig. 14 Runtime Comparison on Quadtree Constructions

From the figure, we can see that the serial implementation actually performs better on single-core CPUs when the number of quadrants is below $2^{18}$. This is not surprising due to the overheads in data transfers and kernel invocations. However, when the number of quadrants is above $2^{20}$ (~1 million), the speedup is increased to 15.2X. The speedup is further increased to 27.3X when the number of quadrants researches 2 million. Different from the CPU-serial implementation whose runtimes grow almost linearly with the number of quadrants, the runtimes of the GPU implementation increase only 0-5 milliseconds when the numbers of quadrant double for the number of quadrants up to 2 million, which is already the largest number of quadrants in our tests. We believe further speedups are achievable for larger datasets and/or using higher maximum quadtree levels (MAX_LEV) which is beyond the scope of our current applications. The results clearly indicate the efficiency of our parallel primitives based design and its GPU implementation. On the other hand, the runtimes for quadtree constructions are relatively insignificant when compared with those of polygon decompositions. As such, the significance of further performance improvement of the module is relatively low. Nevertheless, we consider our parallel primitives based design and implementation of MAQ-Tree using simple vector structures a novel and efficient technique when compared with traditional tree construction techniques that adopt DFS traversals and rely on intensive dynamic memory operations which are becoming increasingly expensive on modern hardware. We plan to perform direct comparisons in our future work.

## 5.3 Performance on Spatial Queries

We have generated five groups of random spatial window/range queries to test the scalability of the proposed parallel design and implementations. The numbers of queries in the four groups are 1,000, 5,000, 10,000, 50,000 and 100,000, respectively. The window is first generated by randomly picking a center point (x/y) and then randomly picking a width and a height. The runtimes of the CPU-Serial, CPU-TBB and GPU implementations using quadtree level 12 are plotted in Figure 15 (using other levels shows similar results and are skipped). In a way similar to the results in quadtree constructions, the GPU implementation is only superior to the CPU-TBB implementation when there are sufficient numbers of queries to saturate GPU hardware. For the largest test set, the GPU implementation is about 10-20X faster than CPU-serial and about 2X faster than CPU-TBB when all the 16 CPU cores are used.

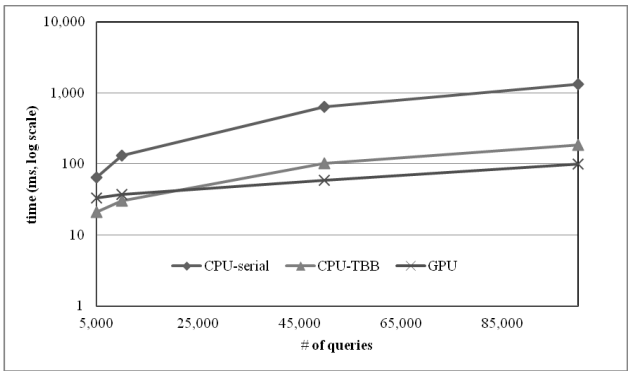


Figure 15 Runtime Comparison on Parallel Range Queries

We have also performed experiments on parallel polygon queries by using the boundary of USA (labeled as "USA" in Figure 16) and boundaries of a set of countries in South America (labeled as "countries" in Figure 16) as our test data to query against the quadtree derived from the species range data at the four different quadtree levels and the results are plotted in Figure 16. As expected, the results are similar to range queries where GPU implementation is only faster when both the query polygons and the quadtrees are sufficiently large. The speedup of the GPU implementation is up to 2X faster over CPU-TBB implementation.

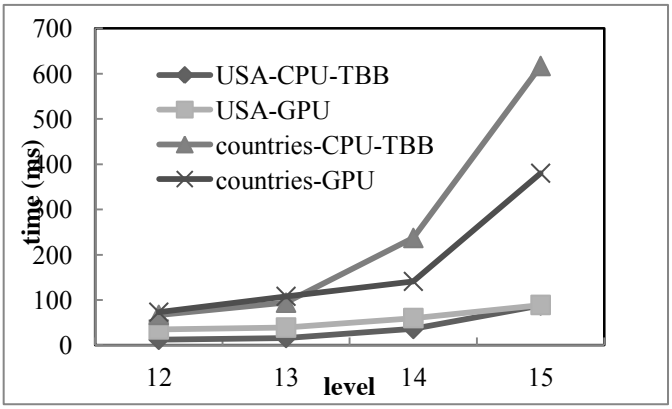


Figure 16 Runtime Comparison on Parallel Polygon Queries

# 6. Comparisons with PixelBox* on Polygon Decomposition

Despite that PixelBox proposed in [5] is designed for computing the area of intersection between two polygons rather than indexing a single polygon as in our work, they share the commonality on top-down and level-wise polygon decomposition. We obtained the source code of PixelBox from their authors and provided an interface for constructing quadtree from multiple complex polygons on top of the SubSampBox routine in PixelBox. We call the resulting hybrid technique as PixelBox*. Similar to PixelBox, PixelBox* also maintains a stack in the shared memory of a thread block that is assigned to decompose a polygon. Each thread in the tread block is assigned to decompose a single box/quadrant and the decomposed quadrants are pushed onto the stack for further decompositions in the next round (level) if they are qualified. Due to the last-in-first-out nature of stacks, PixelBox* inherits the DFS order when decomposing polygons. We also note that, different from the original PixelBox that only supports single-ring polygons, PixelBox* supports complex polygons with holes after extension, which is a must in our applications.

Different from PixelBox* (and hence PixelBox), our parallel primitives based design adopts a BFS order where the polygons are decomposed level by level. In addition, we do not use a private queue for each polygon at the thread block level. Instead, a global queue is maintained for all polygons due to the parallel primitives based design as the underlying parallel primitives support only element-wise operations defined in their functors. While it is generally believed that implementations using native programming languages such as CUDA and using shared memory can significantly improve overall performance, we next show empirically that our parallel primitives based design and implementation is more efficient than PixelBox*, which is implemented in CUDA and optimized for GPUs (see [5] for design considerations and optimization details). First, similar to our experiments on R-Tree traversals, BFS is more efficient than

DFS on GPUs as there are much higher degrees of coalesced memory accesses using BFS despite that accessing the stack on shared memory is faster than accessing the queue structure in GPU device memory in our technique. Second, to make full use of the GPU hardware capacity, the number of split factor should be at least the same as the warp size (32) in PixelBox* (and PixelBox). Take the N=8*8=64 decomposition pattern for example, when traversing along the polygon boundary, the number of expensive tests on the relationship between boxes/quadrants and polygons can be significantly smaller than T=64 when using a multi-level N=2*2=4 decomposition pattern. As our parallel primitives based approach exploits fine-grained data parallelisms and is agnostic to the thread block boundary (which is determined by parallel primitives and invisible to users), it does not suffer from GPU resource utilization constraints as PixelBox and PixelBox*.

For fair comparisons, we force PixelBox* to use N=64 (which is suitable for PixelBox) instead of N=4 in each iteration (as in our original design), in order to improve GPU utilization as in PixelBox. The configuration is termed as PixelBox*-shared. Furthermore, to get rid of the shared memory limit in PixelBox* and potentially achieve better performance, we have modified PixelBox* to use GPU global memory for the stack. The modification allows experiment with different N sizes without worrying about overflowing the stack due to limited per-thread block shared memory capacity. We term the new implementation as PixelBox*-global. We compare our proposed parallel primitives based technique with both PixelBox*-shared and PixelBox*-global. The runtimes on the four dataset groups are plotted in Figure 17.

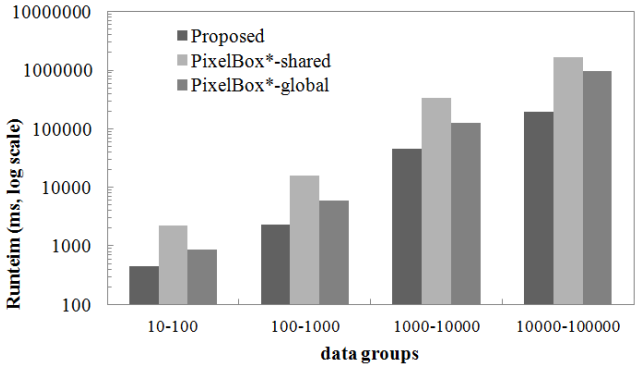


**Figure 17 Runtime Comparisons among the Proposed technique and PixelBox* Variations**

The fact that our parallel primitives based technique is significantly faster (~3X) than PixelBox*-global, which is the best among different PixelBox* variations, can be explained by the previous discussions. However, the observation that PixelBox*-global is about ~2X faster than PixelBox*-shared across the four dataset groups is somewhat surprising, given the common belief that using shared memory can boost GPU performance significantly. One explanation is that, the reported PixelBox*-global runtimes are the best among all configurations using different N sizes. While N can be neither too big nor too small to meet shared memory capacity constraints in PixelBox*-shared, using global memory allows search a much larger parameter space and get better performance in PixelBox*-global. For example, while using a large N may reduce the GPU occupancy in PixelBox*-shared, it may actually improve the overall performance in PixelBox*-global due to better warp scheduling opportunities when there are a larger number of warps in a thread block can be selected for execution. In addition, coalesced global memory accesses to the stack may render the advantages of using shared memory less signficant in this particular application.

In summary, while more thorough investigations are needed to fully understand the advantages of our parallel primitives based design and implementation for polygon decomposition in our particular application, our experiments have shown that, using high level parallel tools, such as parallel primitives, may not necessarily lead to inferior performance. Both efficiency and simplicity can be achieved simultaneously by identifying the inherent data parallelisms in applications, map them to parallel primitives and chain the parallel primitives to develop end-to-end, high-performance applications.

## 7. Summary and Discussions

Our research and development effort on quadtree indexing and spatial query processing are motivated by the practical needs in efficiently managing large-scale species distribution data, in a way similar to several recent works on managing spatial data in high-resolution biomedical images using new hardware, ranging from multi-core CPUs [9], GPUs [5] to Hadoop-alike distributed systems [11]. Given the ubiquitous nature of spatial data, it is important to research and develop a set of high-performance and scalable spatial data management tools across multiple commodity parallel hardware platforms and are applicable to multiple domains.

While previous research works have explored different parallelization techniques that are popular to their respective hardware platforms, in this study, we have investigated a different parallelization strategy in hope to achieve both simplicity in design/implementation and efficiency in execution in the context of spatial data management. Our case study on quadtree indexing and spatial query processing based on the quadtree indexing structure has demonstrated the feasibility of the proposed strategy. Our primitives based parallel designs, although originally designed for GPUs, can be easily ported to multi-core CPUs and achieve high performance.

Although the parallel primitives based techniques may not always bring the best performance due to the inevitable library overheads, we believe that the process in seeking data parallel designs helps understand the inherent parallelisms in processing large-scale spatial data. Different from hardware specific designs, fine-grained data parallel designs on top of parallel primitives may both scale up and scale out (automatically) across multiple hardware generations. This may also simplify integration of multiple hardware platforms, as only a single codebase needs to be maintained as long as the parallel primitives are supported by the underlying hardware platforms. We believe the feature is desirable from an application perspective and the approach merits further research

## 8. Conclusions and Future Work

Motivated by the practical needs in efficiently managing large-scale geo-referenced species distributed data on new hardware and the difficulties in developing hardware specific techniques for such complex applications, we have proposed a parallel primitives based approach to spatial data management. Using quadtree indexing and spatial query processing on complex polygons as a case study, we have developed data parallel designs for polygon

decomposition, quadtree construction and both range and polygon based spatial query processing. We implemented the designs on both GPUs and multi-core CPUs. Experiments have demonstrated that, GPU implementations can achieve 100X+ speedups over serial CPU implementations and 10X+ speedups over multi-core CPU implementations for computationally intensive tasks, such as polygon decompositions for dataset groups 2-4. While the speedups over single and 16 CPU cores drop to 10-27X and less than 2X for quadtree construction and spatial query processing, respectively, they are still significant from an application perspective.

For future work, first of all, we would like to apply the primitives based parallel design strategy to additional spatial data management tasks and develop a comprehensive set of tools to support spatial data management on modern parallel hardware. Second, we plan to provide an integrated frontend with SQL interface to our existing toolset to help using commodity parallel hardware more effectively. Finally, as discussed, we would like to investigate how the data parallel designs may help efficient scheduling across multiple hardware platforms both within and across computing nodes for larger scale data processing.

**Acknowledgement**: This work is supported in part by NSF Grants IIS-1302423 and IIS-1302439 Medium Collaborative Research project "Spatial Data and Trajectory Data Management on GPUs". We would like to thank Kaibo Wang for sharing the PixelBox CUDA source code.

# 9. REFERENCES

[1] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann Publishers Inc., 2005.

[2] G. Alonso, "Hardware Killed the Software Star," in IEEE 29th International Conference on Data Engineering (ICDE), Brisbane, 2013.

[3] M. McCool and J. R. J. Reinders, Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann, 2012.

[4] R. Ridgely, T. Allnutt, T. Brooks, D. McNicol, D. Mehlman, B. Young and J. Zook, "Digital Distribution Maps of the Birds of the Western Hemisphere, version 1.0.," NatureServe, Arlington, Virginia, USA, 2003.

[5] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang and J. H. Saltz, "Accelerating Pathology Image Data Cross-comparison on CPU-GPU Hybrid Systems," Proc. VLDB Endow., vol. 5, no. 11, pp. 1543--1554, 2012.

[6] S. Shekhar and S. Chawla, Spatial Databases: A Tour, Prentice Hall, 2003.

[7] E. H. Jacox and H. Samet, "Spatial join techniques," ACM Transaction on Database Systems, vol. 32, no. 1, pp. 7-24, 2007.

[8] M. Sarwat, M. F. Mokbel, X. Zhou and S. Nath, "Generic and efficient framework for search trees on flash memory storage systems," GeoInformatica , vol. 13, no. 3, pp. 417-448 , 2013.

[9] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan and A. Ailamaki, "TOUCH: in-memory spatial join by hierarchical data-oriented partitioning," in SIGMOD Conference, 2013.

[10] B. Sowell, M. A. V. Salles, T. Cao, A. J. Demers and J. Gehrke, "An Experimental Analysis of Iterated Spatial Joins in Main Memory," Proceedings of the VLDB Endowment , vol. 6, no. 4, pp. 1882-1893 , 2013.

[11] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang and J. H. Saltz, "Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce," Proceedings of the VLDB Endowment , vol. 6, no. 11, pp. 1009-1020, 2013.

[12] A. Eldawy, Y. Li, M. F. Mokbel and R. Janardan, "CG_Hadoop: Computational Geometry in MapReduce," in ACM-GIS Conference, 2013.

[13] OGC, OpenGIS Simple Feature Specificaiton for SQL, 2006.

[14] C. A. Shaffer, H. Samet and R.C. Nelson, "QUILT: a geographic information system based on quadtrees," International Journal of Geographical Information Systems, vol. 4, no. 2, pp. 103-131, 1990.

[15] J. Zhang, S. You and L. Gruenwald, "Parallel Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs," Information Systems, vol. 4, p. 134–154, 2014.

[16] J. Zhang and S. You, "High-performance quadtree constructions on large-scale geospatial rasters using GPGPU parallel primitives," International Journal of Geographical Information Sciences (IJGIS), vol. 27, no. 11, pp. 2207-2226, 2013.

[17] J. Zhang and S. You, "Speeding up large-scale point-in-polygon test based spatial join on GPUs," in Proceedings of the ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data (BigSpatial'12), 23-32, 2012.

[18] Y. Fang, M. Friedman, G. Nair, M. Rys and A.-E. Schmid, "Spatial indexing in microsoft SQL server 2008," in SIGMOD Conference 2008, 2008.

[19] S. Laine and T. Karras, "High-Performance Software Rasterization on GPUs," in High-Performance Graphics, 2011.

[20] F. A. Bisby, "The quiet revolution: Biodiversity informatics and the internet," Science, vol. 289, no. 5488, pp. 2309-2312, 2000.

[21] ESRI, Shapefile Technical Description, 1998. http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

[22] J. Zhang and L. Gruenwald, "Embedding and extending GIS for exploratory analysis of large-scale species distribution data," in ACM-GIS Conference, 2008.

[23] R. Obe and L. Hsu, PostGIS in Action, Manning Publications, 2011.

[24] J. Zhang, M. Gertz and L. Gruenwald, "Efficiently Managing Large-scale Raster Species Distribution Data in PostgreSQL," in ACM-GIS Conference, 2009.

[25] J. Zhang, "A high-performance web-based information system for publishing large-scale species range maps in support of biodiversity studies," Ecological Informatics, vol. 8, pp. 68-77, 2012.

[26] GDAL, Geospatial Data Abstraction Library. http://www.gdal.org/

[27] J. O'Rourke, Computational geometry in C., Cambridge University Press, 1998.

[28] J. Zhang and S. You, "GPU-based Spatial Indexing and Query Processing Using R-Trees," in ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data (BigSpatial'13), 2013