Efficient and Scalable Parallel Zonal Statistics on Large-Scale Species Occurrence Data on GPUs

Jianting Zhang[1,2] and Simin You[2]
1 Department of Computer Science, City College of New York, New York, NY, 10031
2 Department of Computer Science, CUNY Graduate Center, New York, NY, 10006
Correspondent author email: jzhang@cs.ccny.cuny.edu

Abstract

Analyzing how species are distributed on the Earth has been one of the fundamental questions in the intersections of environmental sciences, geosciences and biological sciences. With world-wide data contributions, more than 375 million species occurrence records for nearly 1.5 million species have been deposited to the Global Biodiversity Information Facility (GBIF) data portal. The sheer amounts of point and polygon data and the computation-intensive point-in-polygon tests for zonal statistics for biodiversity studies have imposed significant technical challenges. In this study, we have significantly extended our previous work on parallel primitives based spatial joins on commodity Graphics Processing Units (GPUs) and have developed new efficient and scalable techniques to enable parallel zonal statistics on the GBIF data completely on GPUs with limited memory capacity. Experiment results have shown that an impressive end-to-end response time under 100 seconds can be achieved for zonal statistics on the 375+ million species records over 15+ thousand global eco-regions with 4+ million vertices on a single Nvidia Quadro 6000 GPU device. The achieved high performance, which is several orders of magnitude faster than reference serial implementations using traditional open source geospatial techniques, not only demonstrates the potential of GPU computing for large scale geospatial processing, but also makes interactive query driven visual exploration of global biodiversity data possible.

1. Introduction

Quantifying species-environment relationships, i.e., analyzing how species are distributed on the Earth has been one of the fundamental questions studied by biogeographers and ecologists for a long time (Cox and Moore 2005). Several enabling technologies have made biodiversity data available at much finer scales in the past decade (Bisby 2000), including DNA barcoding for species identification, geo-referring for converting descriptive museum records to geographical coordinates, database technologies for managing species presence locations and related taxonomic and environmental data, and, Geographical Information System (GIS) for species distribution data modeling and analysis. The newly emerging cyberinfrastructure technologies (e.g., metadata, ontology, Web services and scientific workflow) have made exchanging and sharing species distribution data over the Web much easier. The currently largest species occurrence data repository might be the Global Biodiversity Information Facility (GBIF) which was established by governments in 2001 to encourage free and open access to biodiversity data via the Internet (GBIF 2014). Through a global network of countries and organizations, as of August 2012, the GBIF data portal has more than 375 million species occurrences records on 1,487,496 species. The majority of the records are geo-referenced which makes it possible to overlay species occurrence records with different types of raster and vector layers for exploring biodiversity patterns and their relationships with environments and human impacts at global and regional scales.

Given the virtually countless combinations of species taxa, geographical regions and ecosystems (Zhang 2012), many types of exploratory analysis on integrated taxonomic-geographical-environmental data can be investigated (Zhang and Gruenwald 2008). In this study, we will be focusing on a fundamental spatial operation for zone-based point location data summation, i.e., counting the numbers of points that fall within a set of polygons in a zonal dataset. The operation is closely related to point-in-polygon test based spatial joins (Jacox and Samet 2007, Zhang and You 2012a) and is well-supported in several leading GIS software known as Zonal Statistics (Theobald 2005). While both spatial databases and GIS have exploited optimization techniques, such as indexing and preprocessing, existing designs and implementations are mostly based on serial CPU computing models and usually incur significant delays when processing large scale datasets. Modern commodity personal computers are increasingly equipped with large memory and many-core accelerators, such as Nvidia GPUs that are capable of general computing based on the Compute Unified Device Architecture (CUDA) parallel programming model (Kirk and Hsu 2012). Unfortunately, many commercial and open source spatial databases and GIS are optimized for the previous generations of hardware based on outdated cost models and fail to make full use of the computing power provided by modern commodity hardware.

Built on top of our previous research and development efforts on spatial indexing and query processing on GPUs (Zhang and You 2012a, Zhang et al 2014), in this study, we aim at accelerating explorations of the GBIF global biodiversity data by designing and implementing efficient data parallel algorithms for scalable and high-performance zonal statistics on the hundreds of millions of species occurrences over tens of thousands of complex polygons on commodity GPUs with limited memory capacities. First of all, we have designed a framework to allow efficiently use mapped memory on CPUs as extended GPU memory automatically and support data parallel designs. Second, we have developed scalable point indexing technique to index large point datasets that are beyond GPU memory capacity by using mapped memory efficiently through batched processing. Third, we have extended our binary search based spatial filtering algorithm to work with the new point indexing technique. Fourth, a cell-in-polygon test based optimization technique for advanced spatial filtering is developed to allow assigning polygon identifiers to points if the grid cell that the points fall in is tested to be completely within a polygon without performing expensive point-in-polygon test for individual points. We have performed extensive experiments to demonstrate the efficiency of GPU-based massively data parallel zonal statistics technique and compare it with two reference serial implementations using traditional open source software packages. The scalability and performance of the data parallel framework and techniques as well as the effectiveness of the cell-in-polygon based optimization technique are tested under different experiment settings.

The rest of the paper is arranged as follows. Section 2 introduces background and motivation and briefly overviews related work. Section 3 provides details of the data parallel zonal statistics framework, the scalable point indexing and spatial filtering techniques, and the cell-in-polygon test based optimization technique for advanced spatial filtering. Section 4 presents the experiments and results. Finally Section 5 is the conclusion and future work directions.

2. Background, Motivation and Related Work

Given a point dataset T_O representing species occurrences with two attributes (sp_id, the_geom) and a polygon dataset T_Z representing zones also with two attributes (z_id, the_geom), the basic zonal statistics operation to count the number of occurrences of species in each polygon can be expressed as the following Structured Query Language (SQL) statement:

```
SELECT COUNT(*) from T_O, T_Z
WHERE ST_WITHIN (T_O.the_geom,T_Z.the_geom)
GROUP BY T_Z.z_id;
```

Here the_geom attributes in the two datasets represent geometry, i.e., points and polygons, respectively. Advanced zonal statistics operations likely involve species identifiers in additional clauses (such as WHERE and GROUP) to derive the occurrence counts for a single or a group of species. This is useful for biodiversity researchers that are interested in a particular group of species. The species occurrence counts can be used to compute abundance and richness measurements for a variety of types of biodiversity studies (Ricotta 2005). The GBIF data portal (GBIF 2014) has provided overview maps of species occurrences for different species groups as well as countries which is very useful in understanding the overall species distribution patterns. However, the maps are mostly for visualization purposes and are limited to a few fixed resolutions up to 0.1 by 0.1 degree which might be too coarse for many scientific inquiries. We note that, in addition to COUNT, additional statistics such as MIN, MAX, SUM, STD_DEV and MEDIAN are also possible, although not all of them are natively supported by SQL.

It is clear that the zonal statistics based data summation operations are closely related to point-in-polygon test used in the ST_WITHIN function. The function is defined by Open Geospatial Consortium (OGC) Simple Feature Specification (SFS) (OGC 2006) and has been implemented in several spatial databases and GIS, e.g., Java Topology Suit (JTS[1]), Geometry Engine - Open Source (GEOS)[2] and PostGIS/PostgreSQL (Obe and Hsu 2011). The Oracle Spatial development team recently proposed to build an in-memory R-Tree to speed up topological relationship query processing for complex regions (Hu et al 2012), including point-in-polygon test. Point-in-polygon test has been extensively investigated by the computational geometry and spatial databases research communities. While computational geometry research usually focuses on a single point and polygon pair, spatial databases research addresses the overall efficiency on testing a large set of points and polygons that can be abstracted as a special type of spatial joins.

[1] http://www.vividsolutions.com/jts/JTSHome.htm
[2] http://trac.osgeo.org/geos/

Spatial joins are typically divided into two phases, i.e., the filtering phase and the refinement phase (Jacox and Samet 2007). The filtering phase utilizes some pre-built or on-the-fly constructed spatial indices to pair subsets of points and subsets of polygons for further refinements. The refinement phase in the point-in-polygon test based spatial joins applies computational geometry algorithms to determine whether a point is within a polygon for paired points and polygons. Obviously, building indices incurs additional overheads but can significantly reduce the numbers of required point-in-polygon tests and improve spatial join efficiency. In addition to the classic ray-casting based point-in-polygon test algorithm with linear complexity with respect to the number of vertices in a polygon (which does not need preprocessing), several new algorithms have been proposed in recent years (Wang et al 2005, Li et al 2007, Jimenez et al 2009, Yang et al 2010). In some of these approaches, points and polygons that are indexed by a same grid tessellation can be paired directly without requiring tree traversals that typically involve irregular memory accesses which can be expensive on modern hardware (Hennessy and Patterson 2011).

One of the driving motivations for us to push the limits of parallel geospatial processing in personal computing environments is to enable interactive visual explorations on large scale geospatial data. Traditionally MapReduce/Hadoop based parallel processing techniques have been designed for offline data processing and are generally too slow for online interactive queries that require real time responses (Grochow et al 2010). On the other hand, while GPUs have been extensively used for speeding up rendering high-quality images based on sophisticated physics and/or statistics at interactive frame rates, only a few research works have exploited GPU's Single Instruction Multiple Data (SIMD) computing power (Kirk and Hsu 2012) to support query driven visual explorations (e.g. Gosink et al 2009). We hope the scalable data parallel zonal statistics techniques that we have developed in this study can accelerate the performance of the biodiversity data visualization and visual exploratory analysis systems, e.g. those reported in (Zhang et al 2007, Zhang and Gruenwald 2008, Zhang 2012), when applied to much finer resolution data at the global scale. The GBIF global species occurrence data, however, is significantly different from the tree species and bird species range map data that were used in these previous studies: the data volume is 1-2 orders of magnitude higher, the number of species is 2-3 orders of magnitude larger, and, more importantly point locations need to be aligned to zones for subsequent statistics which require computation intensive spatial operations.

The distinctions between data parallelisms and task parallelisms are well known in parallel computing (Hills and Steele 1986, McCool 2012). While parallelism is expressed using distinct tasks which may be different from each other significantly in task parallelisms, data parallelisms is driven by a collection of data which typically have regular data structures such as vectors or matrices. While it is easy to assign the work on processing a chunk of such homogenous data items as a task and use data parallelisms for task parallelisms, the reverse is generally not true. Traditionally task parallelisms are extensively explored in distributed computing and multi-core CPUs as processors can perform very different tasks without suffering performance degradation. However, data parallelisms are becoming increasingly important in making full use of the processing power of modern parallel hardware, including GPUs and Vector Processing Units (VPUs) that come with multi-core CPUs (Hennessy and Patterson 2011). While the benefits of designing data parallel algorithms was not significant enough to justify the cost of explicitly changing design patterns due to limited vector processing power on previous generations of CPUs, the widely adoptions of GPU computing techniques and the increasingly wider SIMD widths have made it necessary to adopt data parallel designs in order to make full utilization of SIMD computing power on GPUs and VPUs (Kirk and Hsu 2012, Hennessy and Patterson 2011). On the other hand, as modern CPUs heavily rely on the cache subsystem to bridge the increasingly larger gap between CPU speeds and memory access latencies and to improve overall system performance, very often good data parallel designs are naturally cache friendly and can significantly improve system performance even on multi-core CPUs. According to the CUDA programming model, the performance of GPUs is maximized when neighboring threads access continuous memory addresses (coalesced memory accesses) and follow a same execution path within a warp of threads (low control divergence). Data parallelisms typically satisfy such requirements very well on GPUs and are naturally cache friendly on both CPUs and GPUs. The excellent scalability of data parallelisms becomes crucially important for modern parallel hardware to reach its potential and deliver the desired high performance. Unfortunately, despite that there are some pioneering works on data parallel designs for geospatial processing, such as polygonization (Hoel and Samet 2003), quad-tree based spatial indexing (Hoel and Samet 1995) and spatial join (Hoel and Samet 1994), the potential of exploring data parallel designs in geospatial processing is still largely unclear, especially on modern commodity parallel hardware.

In our previous works, we have extensively investigated on the potentials of GPU-based spatial indexing and spatial joins and many of them are based on data parallel designs. In particular, we have developed data parallel designs for constructing quadtrees (Zhang and You 2012a) and grid-files (Zhang et al 2014) for large-scale point data and raster data (Zhang and You 2013a). We have also developed and evaluated multiple R-Tree implementations on GPUs for polyline and polygon data (Zhang and You 2013b). A parallel binary search based spatial join framework (Zhang et al 2014) is proposed for joining indexed point data (using quadtree or grid-files) and indexed polyline or polygon data (using grid-files). Several applications that demonstrate the effectiveness and efficiency of the indexing and spatial join techniques have been reported, including point-in-polygon test based spatial association between taxi pickup/drop-off locations and census tracks in the New York City (NYC) (Zhang and You 2012a), point-to-polyline nearest neighbor search based spatial associations between taxi pickup/drop-off locations and street network in NYC (Zhang et al 2014), and Hausdorff distance based trajectory similarity queries in Beijing (Zhang et al 2012). While the datasets (especially for the NYC taxi pickup location point dataset) used in these applications are considerably large, fortunately, they can be fit into high-end GPUs, such as Nvidia Quadro 6000 and GTX Titan with 6 GB memory. However, as reported in (Zhang et al 2014), the limited GPU memory has forced us to index large point dataset using grid-files in CPUs as the underlying radix sort algorithm implemented in the Thrust library[3] that comes with Nvidia CUDA SDK[4] has a large memory footprint and cannot be done in-place on the GPU device in our experiment machine (Quadro 6000, 6 GB). Indeed, while it is not absolutely necessary to materialize intermediate results as vectors in implementing data

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

[4] https://developer.nvidia.com/gpu-computing-sdk

parallel designs using parallel primitives, doing so typically makes programming much easier and implementations more interpretable. Otherwise, multi-level nested iterators will be needed to provide “virtual” vectors as the inputs of the subsequent parallel primitives. However, materializing intermediate results is at the cost of large runtime memory footprint which, unfortunately, is a bottleneck of GPU computing at present. This research aims at providing a more systematic solution to this outstanding research issue from an application perspective by using the GBIF data and its zonal statistics applications as a case study. While using GPU mapped memory on CPUs (see Section 3.1 for more details) has been supported by GPUs since Nvidia’s Fermi architecture, to the best of our knowledge, we are not aware of previous studies on GPU-based geospatial processing that make use of GPU mapped memory for large dataset that is beyond the GPU memory capacity.

As discussed previously, zonal statistics on GBIF species occurrence data is conceptually similar to the point-in-polygon test based spatial join which may suggest that we can simply apply the techniques we have developed in (Zhang and You 2012a) to this new dataset. However, first of all, the number of species occurrences in the dataset (375+ million) is more than two times larger than the number of taxi pickup locations we have processed previously (~170 million) and it is impossible to index all species occurrences on GPUs completely due to their memory capacity limit. Second, the polyline/polygon/trajectory data that we have used in our previous applications are considerably simpler than the World Wild Fund (WWF) ecoregion polygon data[5]. The average number of vertices per polygon in the WWF dataset (279) is nearly three times as large as that of the NYC census block dataset (108) which brings the expected computation intensity to be more than 6 times higher. Third, compared with taxi pickup locations that are mostly clustered in major street intersections, the distributions of species occurrences are much more dispersed which are likely to cause significant divergences on GPUs, a typical problem in degrading GPU computing efficiency (Kirk and Hsu 2012). As such, effective optimization strategies are keys to achieving high performance for large datasets at the scale in order to support interactive visual explorations. Finally, perhaps more importantly, while the recent Nvidia K40 GPUs set GPU memory capacity (12 GB) to a new level, from a research perspective, it is crucial to develop a scalable framework to support zonal statistics and other types of geospatial processing on large dataset that exceeds GPU memory capacity limit.

3. Efficient and Scalable Zonal Statistics on GPUs: Data Parallel Framework and Techniques

We propose to follow the GPU-based spatial join framework we have developed previously (Zhang et al 2014) and reuse existing components, e.g., point-in-polygon test GPU routine presented in (Zhang et al 2012a), whereas possible. Our new contributions in this study are four-fold: 1) a framework to allow efficiently use mapped CPU memory as extended GPU memory automatically and support data parallel designs, (2) a scalable and efficient point indexing technique to index large point dataset that is beyond GPU memory capacity, (3) an extended binary search based spatial filtering algorithm to work with the new point indexing technique, (4) a cell-in-polygon test based optimization

[5] http://worldwildlife.org/biomes

technique for advanced spatial filtering. The four new designs are highlighted and numbered in Fig. 1. We next introduce our data parallel framework as the motherboard for relevant techniques before the design and development details are presented in the following subsections.

3.1 The Data Parallel Framework for Zonal Statistics

The data parallel framework for scalable and high-performance zonal statistics is shown in Fig. 1. Note that we use solid arrows to show data processing steps and dashed arrows to show the correspondences among data used in different components in the framework. Following our previous study (Zhang and You 2012b), the point coordinates and polygon vertices are stored as arrays with each element has a fixed length, instead of storing them as objects that may have variable lengths. Although not shown in Fig. 1 due to space limit, a polygon index array is constructed to store the first vertex positions of the polygons to efficiently access polygon vertex arrays on both GPUs (for coalesced memory accesses) and CPUs (for cache-friendly memory accesses). Since the GPU-based zonal statistics technique is built on top of the point-in-polygon test based spatial joins, we reuse the relevant data parallel designs presented in (Zhang et al 2014) including sort-based point indexing, grid-file based polygon MBB (Minimum Bounding Box) rasterization and indexing and binary search based spatial filtering and nested-loop based spatial refinement. The GPU-based point-in-polygon test technique (Zhang and You 2012a) is plugged into spatial refinement to implement the required zonal statistics functionality. These designs are extended for scalability when necessary and will to be described in their respective subsections next. As both the previous implementations and the implementations for new extensions can be realized using either data parallel primitives supported by parallel libraries (e.g., Thrust) or nested loops with regular data access patterns and can be efficiently realized by using native GPU programming languages (e.g. CUDA), we consider both new designs for individual components and the overall framework data parallel.

Our framework utilizes the Unified Virtual Addressing (UVA) feature that is available in newer generations of GPUs (Kirk and Hsu 2012), which include both Fermi and Kepler based Nvidia GPUs, to allocate chunks of CPU memory and made them accessible to both CPUs and GPUs. We term such CPU memory chunks as GPU mapped memory on CPUs, or simply GPU mapped memory when there are no confusions. Using GPU mapped memory virtually extends GPU memory capacity by using CPU memory which can be two orders of magnitude larger (1-6 GB vs. 100-1000 GB). However, in a way similar to using disks as virtual CPU memory (Hennessy and Patterson 2011), using GPU mapped memory in a naive way may perform poorly. For example, our experiments show that simply applying the parallel sort primitive on GPUs (which is based on radix sort algorithm) for point indexing using mapped memory can result in a much inferior performance. Our data parallel framework allows effectively utilize GPU mapped memory for scalability without significant degrading overall performance when applied to larger scale data. While we currently focus on efficient single-node computing for interaction intensive applications (as discussed in Section 2), conceptually, it is possible to apply the same set of designs to larger but slower storage medium, such as local disks and distributed memory and disks, to achieve even larger scalability when necessary.

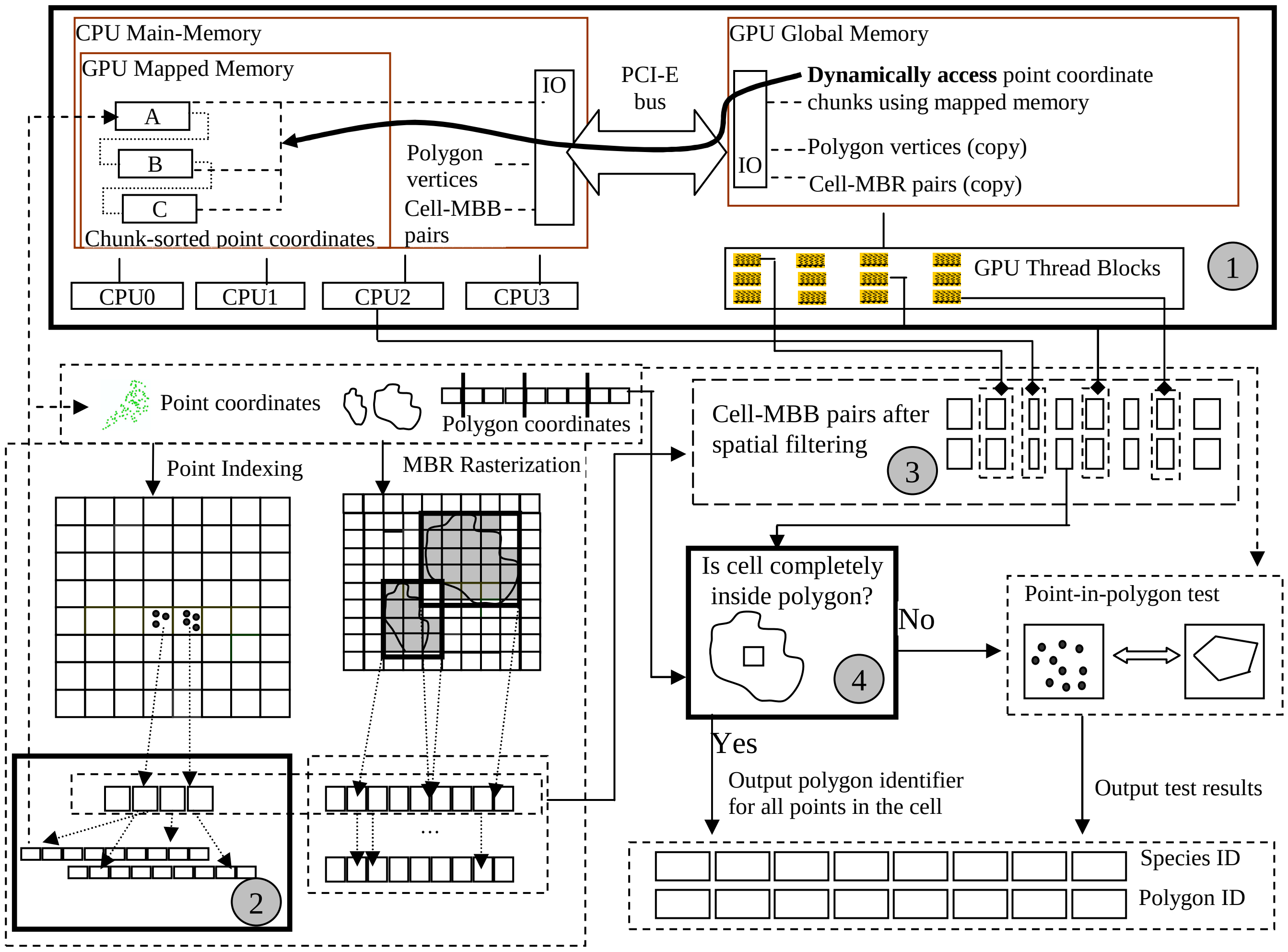


Fig. 1 Data Parallel Framework for Efficient Zonal Statistics on GPUs

We also would like to note that, while not realized in this study, the data parallel framework also naturally supports heterogeneous computing by integrating the parallel computing power of multi-core CPUs, GPUs as well as other types of hardware accelerators (e.g., Intel Xeon Phi devices[6]) that share a same address space. For example, in the context of zonal statistics, the vector of the cell-MBB pairs derived from spatial filtering can serve as a parallel workload queue to distribute workloads to different processing units as shown in the top-right part of Fig. 1 where the assignments are illustrated using solid lines with a diamond ending style. We leave this interesting study to future work.

[6] http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

3.2 Scalable Point Indexing on GPUs using Batched Processing

As reported in (Zhang et al 2014), the Flatly Structured Grid-file (FSG) approach is much simpler than the Multi-Level Quadrant (MLQ) based approach for point indexing from both a design and implementation perspective. The load balancing guarantee of the MLQ approach is not instrumental for large scale data to achieve good performance when the number of (point quadrants, polygon) pairs after spatial filtering is much larger than the number of processing units. The multi-core CPU implementation (using 8 Intel Xeon E5405 CPU cores) of the FSG approach actually has achieved much better performance than the MLQ approach on GPUs (using Nvidia Quadro 6000) despite that the GPU can achieve a much higher sorting rate which is a key to the performance of both MLQ and FSG implementations. The results suggest that the FSG approach is superior to the MLQ approach for spatially joining large scale datasets. Therefore, the FSG approach is adopted in this study for point indexing.

While it is interesting to implement the FSG approach on GPUs, it has a much larger memory footprint which limits the number of species occurrence point records to about 100 million when each record has a length of 12 bytes, i.e., 4-byte float for x/y coordinate and 4 byte integer for taxon identifier. This is also the reason that we were forced to index 170 million taxi trip records on multi-core CPUs as reported in (Zhang et al 2014). The scalability issue of the existing point indexing technique has motivated us to develop a more scalable parallel design for the FSG approach on GPUs. Since the total data volume of the longitude/latitude coordinates is about 4.2 GB for the 375 million point data records which is well below the CPU memory capacity of a reasonably up-to-date workstation, we assume CPU has sufficient memory to hold the raw point data and intermediate results.

Given a point dataset with N records with each record includes a longitude and latitude pair (optionally with some other attributes such as taxon identifier in the GBIF dataset), the dataset is stored as an array of records in a CPU memory block which is mapped by GPU through the UVA mechanism (Kirk and Hsu 2012). Both the CPU and the GPU in a computing node can access the memory block, not only for point indexing but also for point-in-polygon test in spatial refinement. When GPUs access the mapped memory in CPUs, as illustrated at the top of Fig. 1, they are required to transfer data in small units from the mapped memory in CPUs to their processors that need the data through a PCI-E bus dynamically. This is quite different from the conventional way that transfers data from CPUs to GPUs in large chunks before they are processed by GPUs. Clearly the flexibility of being able to utilize larger CPU memory is at the cost of lower efficiency in data transfer, in a way very similar to virtual memory in traditional CPU computing and buffer management in relational database systems.

One might attempt to apply the FSG design to GPU mapped memory to minimize the effort of reimplementation which can be costly. However, this will not work for two reasons. First, while the inputs and outputs of the FSG design can use GPU mapped memory, the implementations of many parallel primitives used in the design (including *sort* in Thrust which is used by FSG) may use temporal GPU memory storage for intermediate results which is typically proportional to the input sizes. The required temporal memory amounts are likely to exceed GPU memory capacity for large scale data and the process will fail due to out of memory. For example, the Nvidia Quadro 6000 GPU can only sort about 200 million records (including longitude/latitude and taxon identifier) which is well below our goal for a scalable solution. Second, even if little intermediate results are produced and the GPU is free from the memory capacity problem by putting both inputs and outputs in GPU mapped memory in CPUs, excessive accesses to the mapped host memory in an uncoordinated manner may significantly degrade performance and make GPU implementations unattractive. For example, sorting a subset of 125 million GBIF point data records in a Quadro 6000 GPU using mapped memory needs 23.867 seconds while only 0.683 second is required if the sorting is done completely in GPU memory. This represents a 34.7X slowdown which is not surprising, given that the underlying radix sort algorithm requires significant amount of data movements and PCI-E bus bandwidths are about 1-2 orders of magnitude slower than GPU memory bandwidths.

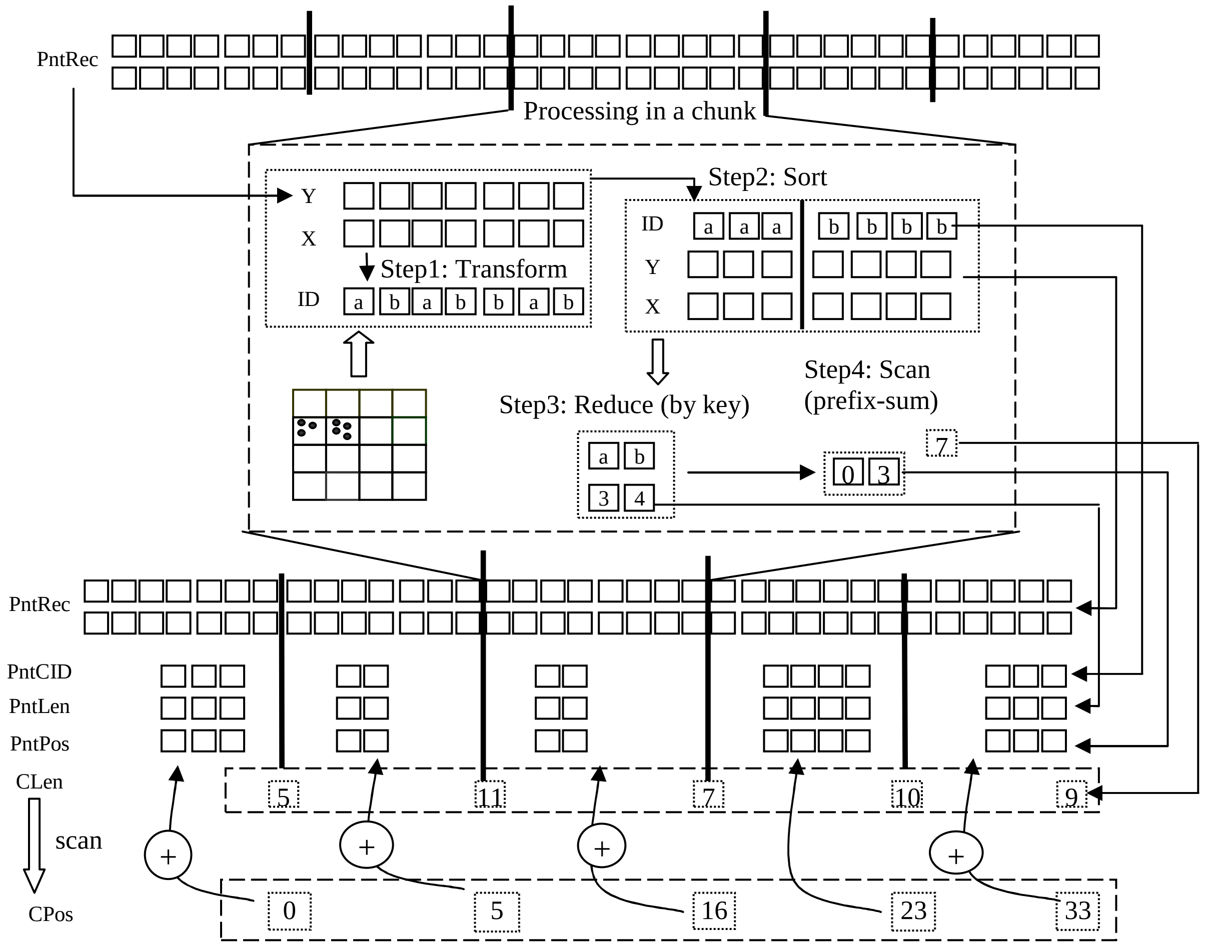


Fig. 2 Parallel Design for Indexing Point Data in Chunks

Our solution is to partition the input point data array into chunks and process the chunks in batches. While we refer to (Zhang et al 2014) for the detailed design of the original (i.e., single-chunk) FSG algorithm and a multi-core CPU implementation for reference, we next briefly repeat the key ideas of the design before presenting design details of the multi-chunk FSG algorithm and its GPU implementation for the purpose of being self-contained. As discussed earlier, the FSG algorithm for point indexing actually is much simpler than the MLQ algorithm presented in (Zhang et al 2012a) and requires a simple chaining of only four parallel primitives, i.e., *transform*, *sort*, *reduce* (by key) and *scan*. The transform primitive derives a cell identifier for each point based on its (longitude, latitude) pair. Given a grid cell size, row-major ordering is used to compute the cell identifier for easy calculation. The next step is to sort the points based on their cell identifiers to put all points that fall within a grid cell close to each other. Clearly, points within a grid cell are not sorted for performance concerns. A *reduce* (by key) primitive is used to count the numbers of points within all grid cells which are subsequently used to compute the positions of the first points among the points that are within the corresponding grid cells. As shown in the middle part of Fig. 2, given an input array *PntRec*, four arrays will be in the output list. In addition to the sorted *PntRec* array, we also have *PntCID* that stores grid cell identifiers, *PntLen* array that stores the numbers of points in cells and *PntPos* array that stores the positions of first points among the points in a grid cell in the sorted *PntRec* array.

When there are multiple chunks in a point data array, thanks to our data parallel design, each chunk can be processed independently, either using a single GPU where the chunks are processed sequentially, or using multiple GPUs where the chunks are processed in parallel, or in a way that combines the two options. For GPUs with smaller memory capacities, we can simply decrease batch sizes and make the technique scalable. The performance will degrade gracefully for very small GPU memory capacities but the tradeoff can be justified in this case. The design is similar to the mapping phase in the MapReduce computing model (Dean and Ghemawat 2010) in the sense that chunks are processed independently and no communications are required in this step.

While it seems that we will need to rearrange the sorted point array in multiple chunks to proceed to the spatial filtering step, our design avoids such data movements (which could be expensive for hundreds of millions records) by only manipulating the three arrays at the grid cell level, i.e., *PntCID, PntLen and PntPos* arrays. Since the number of the grid cells for indexing is typically much smaller than the number of point record, the costs for manipulating such arrays are much lower. This is the key to the scalability and efficiency of our new design for indexing point data. The steps are illustrated in the lower part of Fig. 2. First of all, the total number of points in each chunk is collected for all chunks and stored in the *CLen* array. Similar to computing the *PntPos* array from the *PntLen* array by using a *scan* (prefix-sum) primitive, we can compute the *CPos* array from the *CLen* array. Note that the lengths of the *CLen* and *CPos* arrays are the same as the number of chunks which are typically very small and the costs of this step are negligible. Next, the value of each *CPos* array element is added back to all the elements in the *PntPos* array within each chunk (bottom part of Fig. 2), so that the elements in the *PntPos* array correctly index points in grid cells after concatenating the *PntRec*, *PntLen* and *PntPos* arrays in all chunks. Again, since all the steps are implemented using parallel primitives, the design is highly data parallel and can be implemented on top of parallel libraries that support these fundamental primitives in a straightforward manner. Experiments on the GBIF pint data shows that, about 1/3 of the total processing time is spent on transferring data between GPUs and CPUs while the rest 2/3 of the time is spent on sorting among all batches. The runtimes of the rest steps (including *transform*, *reduce* and *scans*) are relatively insignificant. Given that GPUs have excellent performance on sorting (Merrill and Grimshaw, 2011), the new design, termed as Multi-Chunked FSG for point indexing, is expected to be not only scalable but also highly efficient.

Comparing with our previous design on point indexing that requires all point data fit in GPU memory capacity, our new design not only solves the outstanding scalability problem but also has the following two advantages that are worthy of mentioning. First, the inputs and outputs are stored in GPU mapped memory in CPUs and thus the memory pressure on GPUs is significantly reduced. The reduced memory requirement allows a larger chunk size in a batch and/or supports more sophisticated processing that requires more temporal GPU memory. Second, while not implemented in this study (as the end-to-end performance of point indexing is already satisfactory for the GBIF point data), it is possible to put the task of processing each batch into a GPU stream so that data transfer latency between CPUs and GPUs can be hidden by computing in multiple GPU streams on a single GPU device (Kirk and Hsu 2012). Based on our experiments, the batched GPU implementation is able to index the 375+ million species occurrence records with 3 batches in about 4.5 seconds. The performance amounts to an impressive throughput of 83 million records (about 1 GB data volume in total) per second.

3.3 Extending Spatial Filtering to Support Chunked Point Indexing

The binary search based spatial filtering design and its GPU-based implementation (Zhang and You 2012a, Zhang et al 2014) does not allow duplicated cell identifiers which means that the technique will not work for the multi-chunked point indices using the technique presented in Section 3.2. For a grid cell appeared in *K* chunks, there will be *K* duplicated cell identifiers in the *PntCID* array. We next present details on how binary search based spatial filtering can be extended to work for grid-file indexed point data in multiple chunks.

First, the *PntLen* and *PntPos* arrays derived from Multi-Chunked FSG point indexing approach are sorted by using the *PntCID* array as keys to make the same cell identifiers appear next to each other in the *PntCID* array. Note that the positions of the elements in the *PntLen* and *PntPos* arrays are changed according to the key-value based sorting. Next, as shown in Fig. 3, for each of the element in the MID array, our spatial filtering algorithm binary searches the *PntCID* array by using the corresponding element in the *MC* array as the key. Recall that the MID array and the MC array store the correspondences between polygon MBB identifiers and cell identifiers of rasterized polygon MBBs (Zhang et al 2014). The key extension is to match cell identifiers in the MC array and the sorted PntCID array by using three parallel primitives, i.e., *binary_search*, *lower_bound* and *upper_bound*, as a bundle for binary searches. While the *lower_bound* and *upper_bound* primitives returns the first and the last positions where values could be inserted without violating the ordering during binary searching, the *binary_search* primitive returns whether the values being searched are or are not in the array being searched. The resulting position vectors from *lower_bound* and *upper_bound* primitives need to be filtered out by the resulting boolean vector from the *binary_search* primitive to eliminate unsuccessful searches while keeping the upper bounds and lower bounds of successful searches. Note that it is not necessary to use *upper_bound* primitive if cell identifiers in the *PntCID* array are guaranteed to be unique, which is the case if the point dataset is not chunked. This is exactly the original FSG design for spatial filtering presented in (Zhang et al 2014). Finally, for each matched (MID_i, $lower\_bound_i$, $upper\_bound_i$) triple, we can use MID_i and $lower\_bound_i$ and $upper\_bound_i$ values to access polygon vertex arrays and point coordinate arrays as following. Assuming arrays that store vertex positions and the numbers of polygon vertices are *PlyPos* and *PlyLen*, respectively, then the polygon vertices will be at the position *PlyPos*[**idx**(MID_i)] .. *PlyPos*[***idx***(MID_i)+1]-1 with *PlyLen*[i] vertices. Function ***idx***(i) maps polygon identifier *i* to an index in the *PlyPos* or *PlyLen* array, which can be as simple as ***idx***(i)=*i*. Similarly points that fall within the grid cell whose identifier is being searched are distributed in $upper\_bound_i$ - $lower\_bound_i$ +1 blocks. Note that blocks are combinations of chunks and grid cells, i.e., a bock of points are within a grid cell in a chunk. For each *j*= $lower\_bound_i$ .. $upper\_bound_i$, the starting position and number of points in these blocks are recorded in *PntPos*[*j*] and *PntLen*[*j*], respectively. They can be used to access the *PntRec* array to retrieve point coordinates or other information for further processing. While supporting multiple data point chunks has added significant complexity to our original spatial filtering design, it eliminate the need to actually sort point records in multiple chunks as it would have been done for a single chunk. We note that data movements are typically expensive in various sorting implementations on both CPUs and GPUs and should be avoided as much as possible for large scale data.

To better illustrate our extended design, an example is provided in Fig. 3. In the top part of the figure, after binary searching each cell identifier in the *MC* array from the *PntCID* array, while there are two matched cell identifiers in the *PntCID* array (at position 1 and 2 and shaded with light and dark gray color, respectively) are paired with cell identifier 2 in the *MC* array, there is only one match for cell identifiers 6 and 8, respectively, and there are no match for cell identifiers 5, 4 and 1. As shown in the bottom part of Fig. 3, the three points in the first chunk and the four points in the second chunk in grid cell #2 can be accessed by combining the corresponding elements in the *PntPos* and the *PntLen* arrays. The point data records are colored in light and dark gray in a same way as the two matched elements in the *PntCID*, *Pntlen and PntPos* arrays are colored.

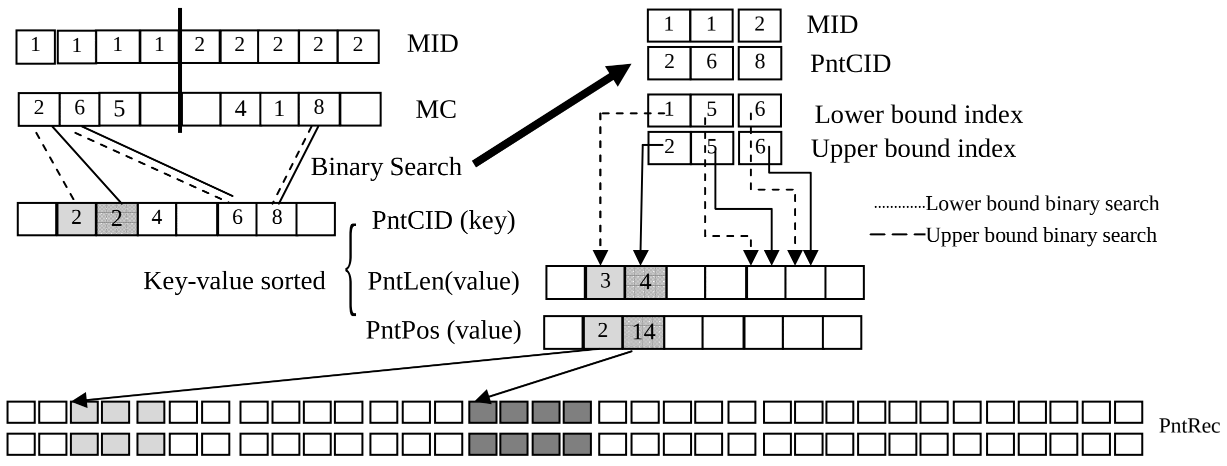


Fig. 3 Data Parallel Design for Spatial Filtering with Chunked Point Indexing

3.4 Parallel Cell-in-Polygon Test for Optimization

The tradeoffs between spatial filtering and spatial refinement in spatial joins are well studied in spatial databases (Jacox and Samet 2007). In our FSG approach, clearly, using a high resolution grid for point/polygon indexing will increase the amount of workload in indexing and spatial filtering but is likely to reduce the workload in the final spatial refinement phase. However, for heavily clustered regions, the numbers of points that fall within some grid cells are likely to be large. Assuming that there are *K* points in a grid cell, directly applying the point-in-polygon test would require O(*K*) tests, each requires O(*V*) operations where *V* is the number of vertices in the polygon to be tested. When *K* is large in such grid cells, directly performing point-in-polygon test can be very expensive.

By observing that if the grid cell is completely inside or outside a polygon, we can directly assign results to all points in the grid cell without requiring any point-in-polygon test. Although a cell-in-polygon test is generally more expensive than a cell-in-polygon test, when *K* is large, the optimization is likely to be beneficial. From a probabilistic perspective, if the probability that the grid cell is completely within or outside of a polygon is high, the overall computing cost can be significantly decreased by performing a single cell-in-polygon test instead of multiple point-in-polyline tests. We consider this optimization technique as part of spatial filtering and refer it as advanced spatial filtering in this study.

Several well-established computational geometry principles can be used to test the relationships between a rectangle (including a squared grid cell) and a polygon. Motivated by the procedure used in (Wang et al 2012), we have used the following two steps to determine whether a grid cell intersects, is within, or, is outside of a polygon. Note that multi-rings are allowed in our technique by separating rings with the origin of the underlying coordinate system. Our technique extends the work in (Wang et al 2012) that only supports single-ring polygons and the extension is necessary for WWF ecoregion data as polygons in this dataset are complex and many of them have multiple rings. As shown in Fig 4A, the first step for cell-in-polygon test is to check whether any of the grid cell's four edges intersect with any of the polygon edges, or, whether any of the polygon's vertices are within the cell, to determine whether the grid cell intersects with the polygon. If the grid cell does not intersect with the polygon, then it is either completely inside (Fig. 4B) or completely outside the polygon (Fig. 4C). We subsequently test whether any of the cell's corners are within the polygon. If the test is true then the grid cell is inside the polygon otherwise the grid cell is outside of the polygon.

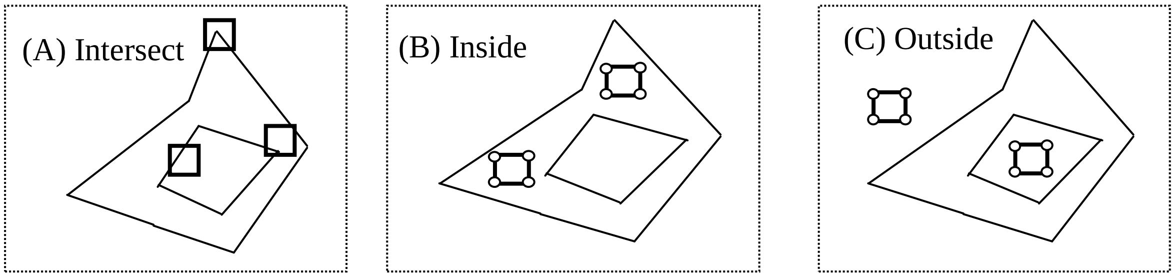


Fig. 4 Three Cases in Cell-in-Polygon Tests

Assuming that there are N+1 polygon vertices, then the total number of edge intersection tests between the 4 cell edges and the N polygon edges is 4*N and the total number of vertex-in-cell test is N+1. Our GPU implementation requires about $C1$=25 arithmetic operations for edge intersection test and about C2=20 arithmetic operations for vertex-in-cell test. As such, the total number of operations required for a cell-in-polygon test is around (4*$C1$+$C2$)*N. In contrast, a single point-in-polygon test requires about $C2*N$ operations while testing K points requires $C2*K*N$ operations. Assuming the probability of the grid cell that is completely within or outside of the polygon is p, then the expected number of operations for applying the optimization technique is $(4*C1+C2)*N*p+C2*K*N*(1-p)$. The simple cost model allows us to further explore the performance of the optimization technique with respect to K, p, $C1$ and $C2$.

First of all, in order for the optimization technique to be beneficial, the number of operations with the optimization should be less than the number of operations without the optimization, i.e., $(4*C1+C2)*N*p+ C2*K*N*(1-p)<C2*K*N$. A simple derivation shows that the necessary condition is $K>4*C1/C2+1$. The condition, which is surprisingly simple and is irrelevant to p, is fairly easy to achieve. This is because, as $C1$ and $C2$ are comparable, the condition can be further reduced to K>5, which should hold for most grid cells. Even if $C1$ is much larger than $C2$ for some hardware instruction sets, there is a high chance that K is still bounded by a relatively small number in order for the condition to hold. Second, we would like to compute the speedup due to the optimization and see how it changes with K, p, $C1$ *and* $C2$. Further assuming $C2=w*C1$, the speedup can be simply calculated as

$$S = \frac{\mathrm{C2*K*N}}{\mathrm{(4*C1+C2)*N*p+C2*K*N*(1-p)}}$$
$$= \frac{\mathrm{K*w*C1*N}}{\mathrm{4*C1+w*C1)*N*p+w*C1*K*N*(1-p)}}$$
$$= \frac{\mathrm{K*w}}{\mathrm{(4+w)*p+K*w*(1-p)}}$$

Although unlikely in real geospatial data, when p is close to 1, i.e., almost all grid cells completely fall within polygons, we can see that S becomes proportional to K which indicates a linear speedup with K. By setting w=20/25=0.8 and plugging p=0.85 and K=364 (measured values, see Section 4.4), the theoretical speedup S is about 6.1X, which agrees with the experiment results (6.4X on GPUs at the grid level 13) very well on an average basis at the dataset level (Section 4.4). When K is relatively large, the K*w*(1-p) part will dominate the denominator in the cost model. As such, the upper bound of S becomes S=1/(1-p) which is much easier to estimate. In this simplified case, S increases with p as expected. When plugging p=0.85, S becomes 6.7 which is still very close to the experiment results reported in Section 4.4. We expect that as the grid level increases and grid cells become smaller, p will increase while K will decrease. This makes it interesting to choose a grid level to maximize performance speedup.

4. Experiments

Our primary goal in this study is to develop high-performance computing tools for zonal statistics of large-scale species occurrence data. Our experiments in this section thus focus on the GBIF species occurrence dataset, although the designs and implementations can be extended to other types of data and additional geospatial operations. We will first provide a description of the datasets and the experiment settings in Section 4.1. As it is impossible to report the experiment results of all the proposed parallel techniques due to space limit, we will be focusing on the overall performance in Section 4.2, the performance of the cell-in-polygon optimization technique for advanced spatial filtering in Section4.3 and comparisons with serial implementations using traditional techniques in Section 4.4.

4.1 Data and Experiment Setting

The GBIF global species occurrence dataset has 375+ million species occurrences records as of 08/02/2012. Our preprocessing results have shown that the dataset contains 1,487,496 species, 168,280 genus, 1,142 families in 262 classes, 109 phyla and 9 kingdoms. The majority (95.7%) of the records is related to animals and plants. A large portion (74.1%) is geo-referenced (with latitude/longitude coordinates at different accuracy levels) and can be associated with terrestrial eco-regions. The WWF ecoregion dataset comes in ESRI shapefile format and has 14,458 polygons, 16,838 rings and 4,028,622 points. The ecoregion data volume is relatively small when compared to today's CPU memory capacities. However, the raw GBIF species occurrence data we received is in the form of a relational database dump with 35 columns and has a total data volume of 180 GB. Many of these columns use the variable character type which makes random accesses very difficult. We have extracted individual columns and converted them into binary format for further processing. In this study, we primarily focus on three attributes, i.e., latitude, longitude and taxon identifier. As the total data volume of the three attributes is less than 1/3 of the CPU memory in our experiment system (16 GB), hereafter we assume that all data involved are memory-resident.

We have empirically set the data grid resolution to 1 arc-minute (approximately 2 kilometers around the equator) primarily because this might be the finest resolution for global biodiversity studies and it may already be beyond the accuracy of some species occurrence records. The width and height of the resulting grid are 21,600 and 10,800, respectively. The gridded coordinates of a point location can be easily stored as a 2-byte short integer along both longitude and latitude dimensions. As the indexing grid resolutions are allowed to be coarser than the data grid resolution, we have chosen three grid resolutions for spatial indexing, i.e., 2^n*2^n for n=13, 14 and 15, to investigate how various performance measurements change with indexing grid resolutions.

All experiments are performed on a Dell Precision T5400 workstation equipped with 16 GB memory and a 500 GB 7200 RPM hard drive. The workstation has dual quad-core Intel E5405 CPUs (8 cores in total) running at 2.00 GHZ and with 6MB L2 cache per core pair, 128 KB L1 cache per core and 12.8 GB/s memory bandwidth per CPU. The workstation is also equipped with an Nvidia Quadra 6000 GPU device with 448 CUDA cores (1.15 GHz), 6 GB GDDR5 memory and 144 GB/s memory bandwidth. The sustainable disk I/O speed is about 100 MB/s while the theoretical data transfer speed between the CPU and the GPU is 8 GB/s through PCI-E. The relevant software installed on the workstation are Nvidia CDUA SDK 5.0 (with Thrust library 1.6), g++ 4.6.3 and Intel TBB 4.1. All programs, including the two serial implementations using traditional technologies (Section 4.3), are optimized with -O3 during compilations for fair comparisons.

4.2 Overall Experiment results

The runtimes of the four components in our GPU-based zonal statistics technique, i.e., point indexing, polygon MBB indexing, spatial filtering and spatial refinement, under the three grid resolutions are plotted in Fig. 5. We do not include polygon MBB indexing runtimes as they are negligible when compared to others (51, 197 and 787 milliseconds for the three grid cell levels). Note that the spatial filtering runtimes are measured with the optimization technique described in Section 3.4. The comparisons with non-optimized implementations are discussed separately in Section 4.3.

From Fig. 5 we can see that, the runtimes of spatial filtering and spatial refinement dominate the overall runtimes under all the three grid resolutions. From an application perspective, the most significant conclusion we can draw from the experiments is that, zonal statistics on the 375+ million species occurrences over the 15 thousand complex ecoregion polygons based on point-in-polygon test spatial relationship can be completed on a commodity workstation equipped with a single GPU device in the order of 100 seconds.

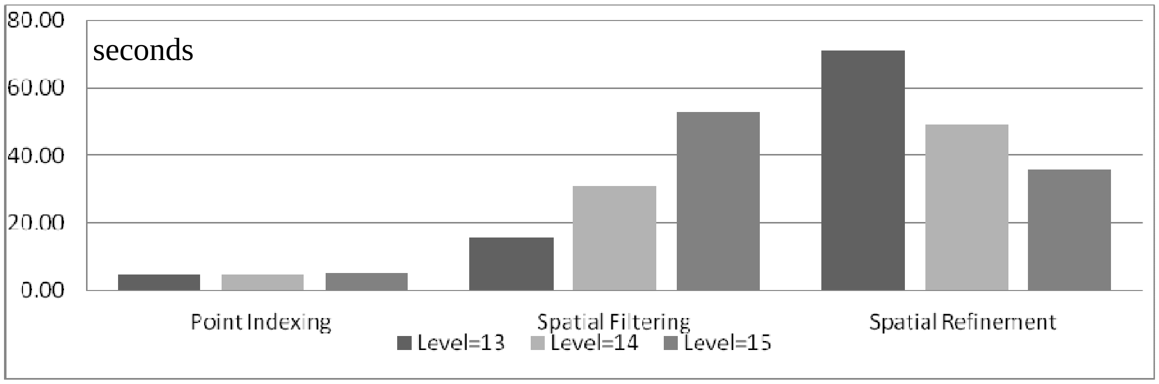


Fig. 5 Plots of runtimes of Point Indexing, Spatial Filtering and Spatial Refinement on GPUs using three grid resolutions

Our data parallel designs make it relatively easy to implement the designs in multiple parallel hardware platforms. For demonstration and comparison purposes, we have also implemented the designs on multi-core CPUs. To minimize the additional implementation efforts, since the Thrust parallel library also provides interfaces to Intel Thread Building Block (TBB[7]) library that is known to be efficient on multi-core CPUs, we recompile our GPU-based Thrust code to use TBB and link it with TBB runtime library to utilize multi-core CPUs, in a way similar to the work reported in (Zhang et al 2014) for point-to-polyline nearest neighbor search based spatial joins, but with two exceptions. The first exception is on point indexing where we have found that the GNU parallel mode library[8] is more efficient for multi-core CPU based sorting and we use it

[7] https://www.threadingbuildingblocks.org/
[8] http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html

instead for fair comparisons. The second exception is related to the native CUDA implementation of the point-in-polygon test module as reported in (Zhang and You 2012a). Also fore fair comparisons, we have implemented the point-in-polygon test module using the native TBB programming model by assigning a range of (polygon, block) pairs as a task and let a single CPU core loop through all the points in the polygon for point-in-polygon test.

As expected, the GPU-based implementations are significantly faster than their peer multi-core CPU implementations with speedups ranging from 2.7X to 4.7X for the three major components (point indexing, spatial filtering and spatial refinement) under the three grid resolutions, as shown in Fig. 6. The speedups are higher for spatial filtering and spatial refinement as they are more computing intensive and can better use GPU's massive floating point computing power better. Please note that the CPU performance is measured when all the 8 cores are fully utilized and the multi-core CPU implementations have been optimized as much as possible for fair comparisons. Our results agree with the rigorous performance analysis on quite a few non-geospatial benchmarks by Lee et al (2010) when comparing the performance of GPUs and multi-core CPUs. The comparisons may also suggest that our data parallel designs can achieve high efficiency on both GPUs and multi-core CPUs by using parallel primitives that are optimized for the respective hardware platforms. As such, they are less likely to depend on the programming skills of individual programmers and are more preferable from a software development perspective.

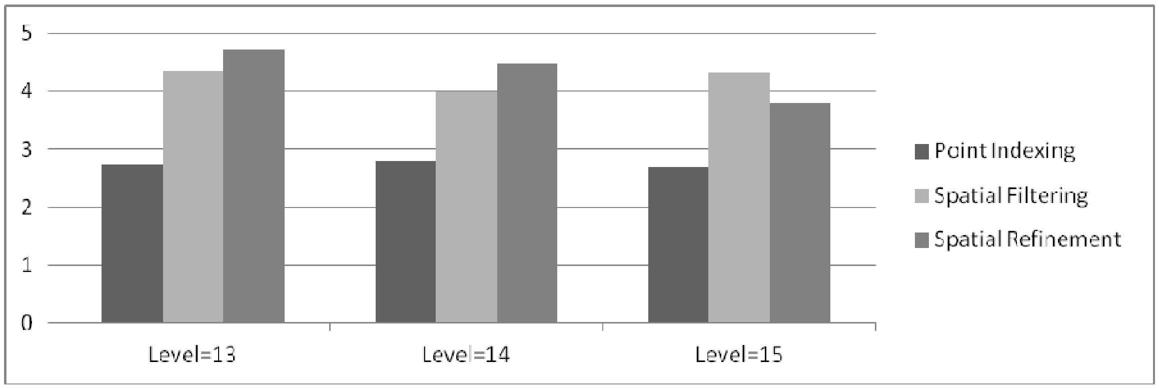


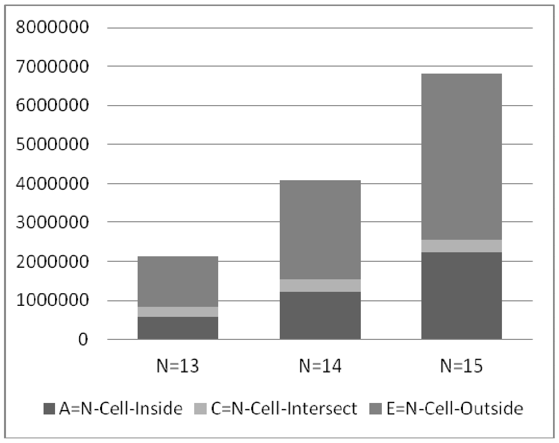
Fig. 6 Plots of GPU over multi-core CPU speedups of Point Indexing, Spatial Filtering and Spatial Refinement using three grid resolutions

After comparing with multi-core CPU implementations based on our data parallel designs, we would like to comment on the relationships between filtering and refinement using different grid resolutions in our GPU-based implementation as observed in the experiments before we move to experiments on the cell-in-polygon test optimizations in the next subsection. First of all, from Fig. 5, it is easy to see that the filtering runtimes increase with grid resolutions while the refinement runtimes decrease with grid resolutions for both CPU and GPU implementations. This is expected as using finer resolution grid for filtering reduces false positives and requires fewer point-in-polygon tests in the refinement phase. Since cell-in-polygon test is used in the filtering phase as an optimization technique, which is also computation intensive, the runtimes in the filtering phase are comparable with the runtimes in the refinement phase, although the computing workload for the basic spatial filtering design cam be quite light (Zhang et al 2014). While the runtime of spatial filtering is about 1/5 of the runtime of spatial refinement at the grid level 13, the ratio quickly increases to 1.6 at the grid level 15. The totals of the filtering and refinement runtimes (and hence the end-to-end runtimes) are minimized at the grid level 14. The results indicate that choosing proper grid level is important in improving the system performance and we leave a more comprehensive investigation for future work.

4.3 Experiments on Cell-in-Polygon Test based Optimization

Recall that when pairing grid cells for points with grid cells for polygon MBBs in the cell-in-polygon test based optimization for advanced spatial filtering (Section 3.3), there are three cases for non-empty point grid cells. The first case is that point grid cells are not in the polygons corresponding to the paired MBBs and the number of such grid cells is measured as E=*N-Cell-Outside*. The second case is that point cells complete fall within polygons and the number is measured as A=*N-Cell-Inside*. The rest of the grid cells belong to the third category whose number is measured as C=*N-Cell-Intersect*. Subsequently the total numbers of points that fall within these grid cells can be computed by summing up all the points in the respective types of grid cells and are referred as F=*N-Point-Outside*, B=*N-Point-Inside* and D=*N-Point-Intersect*, respectively. The numbers of grid cells and the numbers of points are plotted in Fig. 7. Clearly, C=*N-Cell-Intersect* is much smaller than both *A=N-Cell-Inside* and *E=N-Cell-Outside* (left of Fig. 7*)* and *D=N-Point-Intersect* is much smaller than *B=N-Point-Inside and F=N-Point-Outside* (right of Fig. 7*)*. This is the foundation of our optimization technique and will be further discussed from a probabilistic perspective shortly. An interesting observation is that, species occurrences that fall in the first category of grids (outside) are mostly for non-terrestrial species. While the numbers of occurrences are relatively small (*F=N-Point-Outside*), i.e. the species distributions in these grid cells are sparse, the number of such grid cells (E=*N-Cell-Outside*) is large. Fortunately, we can simply discard such grid cells and the associated species occurrence records after the advanced spatial filtering as they are deemed not to be associated with any polygons representing terrestrial ecoregions that are paired with as a result of the basic spatial filtering.

To support the analysis based on the cost model and help validate the optimization technique, we have also computed the *K* and *p* values (defined in Section 3.4) at the three levels as following. At the dataset level, *K* can be intuitively defined as the total number of points (i.e., NP=B+D+F) divided by the total number of cells (i.e., NC=A+C+E), where the values A through F are plotted in Fig. 7. We can compute *p* based on either cells (i.e., *p*-Cell=(A+C)/NP) or points (i.e., *p*-Point=(B+D)/NP). When points are uniformly distributed, the two measurements should be close. However, this is not the case for the GBIF data as the numbers of species occurrence records vary significantly across the world due to various ecological and geographical reasons and human factors in collecting the data. From the left plot in Fig. 8, we can see that *K* decreases from 364 at the level 13 to 112 at the level 15, which is expected. While *p*-Cell values are larger than *p*-Point values, all the ratios are above 0.8 as shown in the right part of Fig. 8. This clearly indicates the effectiveness of the optimization technique as only points in the "intersect" cells need to be actually tested while points in the "inside" and "outside" cells can be assigned polygon identifiers directly and simply discarded, respectively.



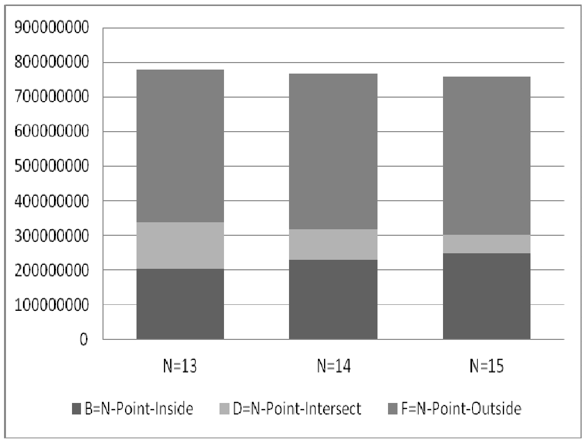
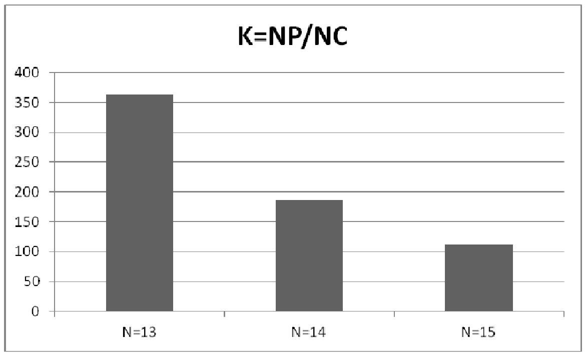


Fig. 7 Plots of numbers of grid cells (left) and numbers of points (right) in grid cells that are inside, intersect and outside of paired polygons at three grid resolutions



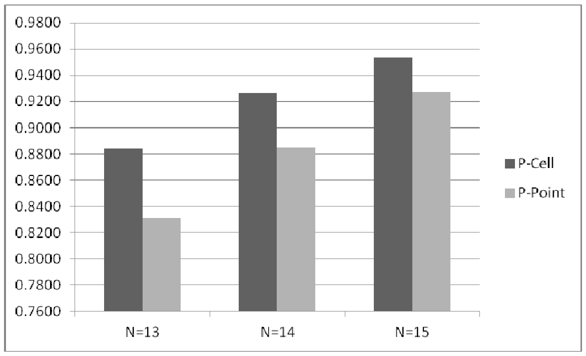
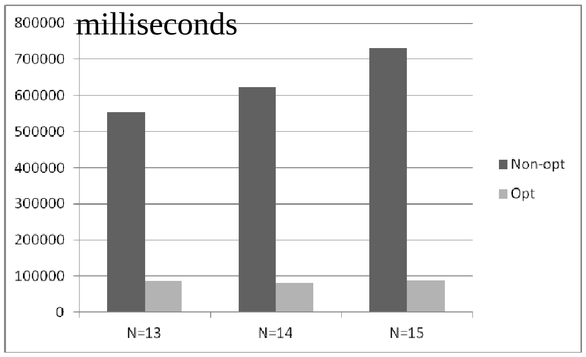


Fig. 8 Plots of *K* values (left) and *p*-Cell and *p*-Point (right) values at three grid resolutions

The left part of Fig. 9 plots the total runtimes of spatial filtering on GPUs with and without the optimization technique for advanced spatial filtering. To validate the effectiveness of the cost model presented in Section 3.4, we have used both *p*-Cell and *p*-Point as *p* values and plugged *K* values (Fig. 8) into the cost model and term the resulting speedups as *S*-Cell and *S*-Point (*w* has been set to 0.8 as explained in Section 3.4). From the results we can see that both the computed and measured speedups increase as the grids get finer. The measured speedup due to the optimization technique is labeled as *S*-GPU and they are plotted in the right part of Fig. 9. We can see that *S*-GPU agrees with *S*-Point pretty well although *S*-Cell is generally over estimated. The results clearly demonstrate the effectiveness of our cost model presented in Section 3.4 by using point level statistics. This can be useful for guiding query optimizations and we will explore it further in our future work.



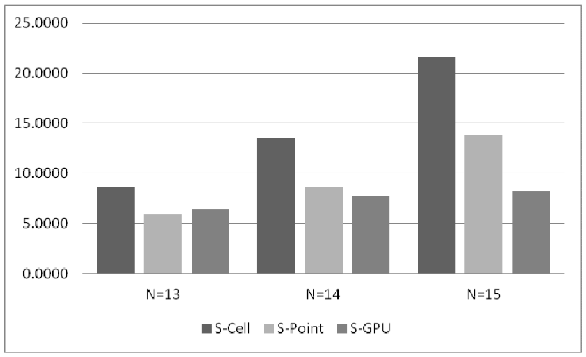


Fig. 9 Plots of total runtimes of optimized and non-optimized GPU implementations (left) and their relative speedups (right) at three grid resolutions

4.4 Comparisons with alternatives using traditional technologies

It is not our intention to directly compare our memory-resident massively data parallel technique with serial implementations using traditional geospatial software packages that are designed for uniprocessors and disk-resident systems. This is because the two techniques are developed for different applications targeting at different hardware. Nevertheless, we report performance comparisons with two serial implementations using libspatialindex[9] for R-Tree based polygon indexing and GDAL[10] (through GEOS) for point-in-polygon test for reference purposes. The comparisons can also help understand the level of performance that our technique has achieved due to data parallel designs and optimized implementations on GPUs.

The major difference between the two serial implementations is that the second implementation incorporates an optimization heuristic in hope to improve the overall performance while the first serial implementation simply querying polygon MBBs that intersect with each and every point before performing point-in-polygon test between the point and the polygons whose MBBs intersect with the querying point. Given that querying the polygon R-Tree for 375+ million points can be expensive when traversing the polygon R-Tree individually, the heuristic is to locate all the MBBs in the polygon R-Tree leaf nodes that intersect with grid cells of groups of points where only a single R-Tree query is needed for the groups of points within the grid cells. The second implementation clearly requires grid-based indexing of points but can potentially save R-Tree query time as the number of accesses to R-Tree nodes can be significantly reduced through point grouping. Although it is possible to use R-Tree to index points by treating each point as a degenerated MBB, the high index construction cost has led us to decide to either not index the point data (implementation 1) or re-use the results of our grid file based indexing (implementation 2).

We believe the first serial implementation represents a reasonably efficient implementation by apply spatial filtering before refinement as a typically trained geospatial programmer would do. We also expect the second serial implementation to be

[9] http://libspatialindex.github.io/
[10] http://www.gdal.org/

more efficient by incorporating the optimization heuristic. However, the results are quite the opposite as detailed below. The code for the two serial implementations and the three subsets of point data are publically available online[11] and we encourage interested readers to cross-examine the implementations, validate the experiment results and make independent comparisons.

First of all, neither implementation is as efficient as we have expected. It takes 18.77 hours to process a subset of approximately 10 million point records with a throughput in the order of 139 points per second. Additional experiments using two smaller subsets of species with 279,808 and 746,302 points result in similar performance, i.e., 138 points per second for both smaller datasets. By using a linear extrapolation, it would take 600+ hours to complete the 375 million points using the serial implementations, although the implementation does exhibit excellent scalability and is suitable for MapReduce/Hadoop systems. However, the performance is 4-5 orders of magnitude slower (138 points per second) than our GPU based implementation (375 million points in about 100 seconds) which is inferior from both a usability and a finical perspective.

Second, the experiments show that the optimization heuristic employed in the second serial implementation is largely ineffective. While measured accesses to the polygon R-Tree has been dramatically reduced by the optimization, the runtimes do not get improved noticeably. Further investigations have revealed that the polygon R-Tree is fairly small (a few megabytes) and can be completely cached in memory which makes reducing accesses to R-Tree insignificant as in disk-resident cases. Since querying cells boundaries against the polygon R-Tree will inevitably cause more false positives when compared with directly querying points and point-in-polygon test which is much more expensive than accessing memory-resident R-Tree nodes, the heuristic does not work as expected. Since point data are also made memory resident in both serial implementations, we can conclude that the low performance of the serial implementations is largely unrelated to disk I/Os in our experiments.

While we are still in the process of fully understanding the 4-5 orders of magnitude of performance differences, we believe that excessive memory allocation/deallocation to accommodate for low memory capacities, library overheads for generality (e.g., object-oriented abstractions) and mismatch between traditional data structures and algorithms with modern hardware architectures (e.g., cache unfriendliness in depth-first tree traversals) are among the factors that contribute to the low performance of the two serial implementations by using traditional geospatial techniques. Furthermore, it is interesting to observe that, even assuming that our multi-core CPU-based implementations have achieved perfect scalability (8X for 8 cores), the performance of the corresponding serial implementations of our data parallel designs (by multiple the number of cores with the measured runtimes) are still about three orders of magnitude faster than using traditional technologies. This may suggest that there is a huge room to improve traditional spatial data processing technologies by adopting data parallel designs and hardware architecture aware implementations. We leave this interesting interdisciplinary research topic for our future work.

[11] http://www-cs.ccny.cuny.edu/~jzhang/zs_gbif.html.

5 Summary and Conclusions

In this study, we have significantly extended our previous techniques for point-in-polygon test based spatial joins on GPUs for large scale data. The integrated scalable designs and their GPU implementations have successfully performed zonal statistics on 375+ million global species occurrence records over 15 thousand complex ecoregions in facilitating exploring global biodiversity explorations. First, we have developed a scalable data parallel framework by using GPU mapped memory in CPUs for large-scale data that may exceed GPU memory capacity. Second, we have further extended our point data indexing and binary search based spatial filtering designs to accommodate multi-chunked point data indexing while achieving much higher efficiency when compared with using mapped memory naively. Third, the cell-in-polygon test based optimization technique for advanced spatial filtering is highly effective and achieves 6.4-8.2x speedups. The measured speedups match with our cost model very well which opens the possibility for predictive optimization. The combined improvements have reduced the total runtime to about 100 seconds using a single GPU device. The performance is several orders of magnitude faster than two reference serial implementations using traditional open source geospatial techniques. The realized high performance on top of scalable designs is not only significant for practical applications in exploring increasingly larger global biodiversity data but also suggests that there are huge rooms to improve the performance of traditional geospatial technologies on modern parallel hardware.

For future work, first of all, we would like to integrate our technique with data management and visualization frontends for practical applications. Second, since the scalable data parallel framework is also applicable to other types of spatial processing, it is thus interesting to examine its scalability in additional applications with larger scale data. For example, spatially and temporally associating 2.7 billion GPS points deposited to Openstreetmap Planet[12] with global road networks by using the point-to-polyline nearest neighbor search based spatial joins (Zhang et al 2014). Third, as discussed in Section 3.1, our new data parallel framework allows integrate multi-core CPUs and multi-GPUs as well as other types of hardware accelerators that share a same address space to synergistically process large scale data by assigning chunks of array elements to multiple processors in a straightforward manner. We plan to materialize the design which essentially allows heterogeneous computing and implement it on a hybrid CPU-GPU system for performance evaluation using the GBIF data and the Openstreetmap Planet GPS location data. Finally, while it is certainly a challenging task that requires significant effort, we plan to investigate the mismatches between the designs and implementations of traditional geospatial processing software packages and the new generation of parallel hardware in a systematic manner. The findings may not only lead to improved performance but also may provide new insights on how to make better use of commodity parallel hardware and enable larger scale geospatial processing with higher efficiency and better scalability.

ACKNOWLEDGEMENT

This work is supported in part through PSC-CUNY Grants #65692-00 43 and #66724-00 44 and NSF Grants IIS-1302423.

[12] http://wiki.openstreetmap.org/wiki/Planet.osm

References

1. Bisby, F. A., 2000. The quiet revolution: Biodiversity informatics and the internet, Science, 289 (5488), pp. 2309-2312.
2. Cary, A., Sun, Z., Hristidis, V. and Rishe, N., 2009. Experiences on processing spatial data with MapReduce. Proceedings of the 21st International Conference on Scientific and Statistical Database Management (SSDBM'09), pp.302-319.
3. Cox, C. and Moore, P., 2005. Biogeography: An Ecological and Evolutionary Approach (7th Ed.), Wiley.
4. Dean, J. and Ghemawat, S., 2010. MapReduce: a flexible data processing tool. Communications of the ACM, 53(1), pp.72-77.
5. GBIF, 2014. Global biodiversity information facility (GBIF) data portal, online at http://data.gbif.org/.
6. Grochow, K., Howe, B. , Stoermer, M. , Barga, R. and Lazowska, E. , 2010. Client+Cloud: evaluating seamless architectures for visual data analytics in the ocean sciences, Proceedings of the 22st International Conference on Scientific and Statistical Database Management (SSDBM'10), pp. 114-131.
7. Gosink, L. J., Wu, K. et al. 2009. Data parallel bin-based indexing for answering queries on multi-core architectures, Proceedings of the 21st International Conference on Scientific and Statistical Database Management (SSDBM'09), pp. 110-129.
8. Hennessy, J. and Patterson, D. A., 2011. Computer Architecture: A Quantitative Approach (5th ed.), Morgan Kaufmann.
9. Hillis, W. D. and Steele, Jr., G. L., 1986. Data Parallel Algorithms. Communications the ACM (CACM), 29(12), pp.1170-1183
10. Hoel, E. G. and Samet, H., 1994. Performance of Data-Parallel Spatial Operations. Proceedings of the International Conference on Very Large Data Bases (VLDB'94), 156-167.
11. Hoel, E. G. and Samet, H., 1995. Data-parallel primitives for spatial operations using PM quadtrees. Proceedings of Computer Architectures for Machine Perception (CAMP'95), pp. 266-273.
12. Hoel, E. G. and Samet, H., 2003. Data-parallel polygonization. Parallel Computing 29(10), pp. 1381-1401.
13. Hu, Y., Ravada, S., Anderson, R. and Bamba, B., 2012. Topological relationship query processing for complex regions in oracle spatial, Proceedings of ACM international symposium on Advances in Geographic Information Systems (ACM-GIS'12), pp. 3-12.
14. Kirk, D. B. and Hwu,W.-M. W., 2012. Programming Massively Parallel Processors: A Hands-on Approach (2nd ed.), Morgan Kaufmann.
15. Lee, V.W. Kim, C. et al, 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10), pp. 451-460.
16. Li, J., Wang, W. and Wu, E., 2007. Point-in-polygon tests by convex decomposition, Computers & Graphics, 31 (4), pp. 636-648.
17. Jacox, E. H. and Samet, H., 2007. Spatial join techniques, ACM Transaction on Database System, 32 (1), Article #7.

18. Jimenez, J. J., Feito, F. R. and Segura, R. J. , 2009. A new hierarchical triangle-based point-in-polygon data structure, Computers & Geosciences, 35 (9), pp. 1843-1853.
19. Merrill, D. and Grimshaw, A. S., 2011. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing, Parallel Processing Letters, 21 (2), pp. 245-272.
20. McCool, M., Robison, A.D. and Reinders, J., 2012. Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann.
21. Obe, R. and Hsu, L., 2011. PostGIS in Action, Manning Publications.
22. OGC, 2006. Open Geospatial Consortium (OGC) Simple Feature Specification (SFS), online at http://www.opengeospatial.org/standards/sfs.
23. C. Ricotta, 2005. Through the jungle of biological diversity, Acta Biotheoretica, 53, pp. 29-38.
24. Theobald, D., 2005, GIS Concepts and ArcGIS Methods, 2nd Ed., Conservation Planning Technologies, Inc.
25. Wang, K., Huai, Y. et al., 2012. Accelerating pathology image data cross-comparison on CPU-GPU hybrid systems, Proceedings of the VLDB Endowment, 5 (11), pp. 1543-1554.
26. Wang, W., Li, J. and Wu, E., 2005. 2D point-in-polygon test by classifying edges into layers, Computers & Graphics 29 (3), pp. 427-439.
27. Yang, S., Yong, J.-H. et al., 2010. A point-in-polygon method based on a quasi-closest point, Computers & Geosciences, 36 (2), pp. 205-213.
28. Zhang, J., You, S. and Gruenwald, L., 2014. Parallel Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs. Information Systems, in press. (doi: 10.1016/j.is.2014.01.005).
29. Zhang, J. and You, S., 2013a. High-performance quadtree constructions on large-scale geospatial rasters using GPGPU parallel primitives. International Journal of Geographical Information Sciences (IJGIS), 27(11), pp. 2207-2226.
30. Zhang, J. and You, S., 2013b. GPU-based Spatial Indexing and Query Processing Using R-Trees. Proceedings of ACM SIGSPAIAL Proceedings of the ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data (BigSpatial’13).
31. Zhang, J., 2012. A high-performance web-based information system for publishing large-scale species range maps in support of biodiversity studies. Ecological Informatics 8, pp. 68-77.
32. Zhang, J. and You, S., 2012a. Speeding up large-scale point-in-polygon test based spatial join on GPUs, Proceedings of the ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data (BigSpatial’12), pp. 23-32.
33. Zhang, J. and You, S., 2012b. CudaGIS: Report on the design and realization of a massive data parallel GIS on GPUs. Proceedings of the ACM SIGSPATIAL Workshop on GeoStreaming (IWGS’12), pp. 101-108.
34. Zhang, J., You, S. and Gruenwald, L., 2012. U^2STRA: high-performance data management of ubiquitous urban sensing trajectories on GPGPUs. Proceedings of the 2012 ACM workshop on City data management workshop (CDMW’12), pp. 5-12.
35. Zhang, J. and Gruenwald, L., 2008. Embedding and extending GIS for exploratory analysis of large-scale species distribution data, Proceedings of ACM international symposium on Advances in Geographic Information Systems (ACM-GIS'08), #28.

36. Zhang, J., Pennington, D. D. and Liu, X., 2007. GDB-explorer: Extending open source Java GIS for exploring ecoregion-based biodiversity data. Ecological Informatics, 2 (2), pp. 94- 102.