CudaGIS: Report on the Design and Realization of a Massive Data Parallel GIS on GPUs

Jianting Zhang
Department of Computer Science
The City College of the City University of New York
New York, NY, 10031
jzhang@cs.ccny.cuny.edu

Simin You
Dept. of Computer Science
CUNY Graduate Center
New York, NY, 10016
syou@gc.cuny.edu

ABSTRACT

We report the preliminary design and realization of a high-performance, general purposed, parallel GIS (CudaGIS), based on the General Purpose computing on Graphics Processing Units (GPGPU) technologies. Still under active developments, CudaGIS currently supports major types of geospatial data (point, polyline, polygon and raster) and provides modules for spatial indexing, spatial join and other types of geospatial operations on such geospatial data types. Experiments have demonstrated 10-40X on main-memory systems due to GPU accelerations and 1000-10000X speedups over serial CPU implementations and disk-resident systems by integrating additional performance boosting techniques, such as efficient in-memory data structures and algorithmic engineering.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database applications -Spatial databases and GIS

General Terms

Management, Design

Keywords

GIS, Spatial Databases, GPGPU, Data Parallelism

1. INTRODUCTION

The increasingly larger data volumes and more complex semantics of geospatial information never cease to request more computing power to turn such data and information into knowledge and facilitate decision support, ranging from global change research to personal travel planning. While parallel processing has been considered as an important component in achieving high-performance in geospatial computing [1], it was not until the General Purpose computing on Graphics Processing Units (GPGPU) technologies appeared in 2007 that large-scale parallel geospatial computing on commodity hardware become a reality, both technologically and economically. Indeed, while supercomputers and parallel computing resources had been made available to highly selective research groups in the past, the inexpensive commodity GPUs, whose architectures closely resemble supercomputers as argued in [2], are affordable to virtually all research groups and individuals. As such, there are significant potentials for GPU accelerated GIS to boost the performance of geospatial computing in a personal computing environment for larger and more complex datasets.

ACM SIGSPATIAL IWGS'12, November 6, 2012. Redondo Beach, CA, USA Copyright (c) 2012 ACM ISBN 978-1-4503-1695-8/12/11...$15.00

There are quite a few technical challenges in designing and realizing such a GPU based parallel GIS. First of all, while many geospatial computing tasks (especially local and focal based operations) are inherently data parallel, there are also quite some tasks (e.g., zonal and global operations) require more sophisticated mappings among conceptual designs and parallel computing models and hardware architectures. Second, GPGPU technologies are relatively new and existing programming languages and software tools may not be sufficient for cost-effective application developments. General purposed software packages and application development platforms that can bridge between geospatial applications and GPU hardware specific computing models are essential but insufficiently supported at present. In this paper, we introduce our research and development efforts in designing and implementing a GPU based general purposed parallel GIS targeting at typical personal computing environments for a wide range of geospatial applications. Since the prototype system was developed using Nvidia Compute Unified Device Architecture (CUDA) parallel programming language and its libraries [3], we tentatively call the collection of modules that we have developed as CudaGIS. While still under active development and new modules are being added, currently CudaGIS is able to handle major geospatial data types such as raster, point, polyline and polygon and support efficient indexing and query processing on them. Given that geospatial data are both computing and I/O intensive and usually requires extensive visualization and interaction, high-performance GIS modules that can run on personal computers can be valuable to many geospatial applications [4]. Furthermore, by processing larger scale geospatial data on a single computing node faster, the communication overheads in distributing the computing tasks over a set of independent computing nodes in grid or cloud computing environments can be significantly reduced [5] which in turn allows solve even larger scale problems on cluster computers more efficiently.

While this paper focuses on high-level system design and realizations of key components (we refer to relevant publications and technical reports [6-16] for performance of individual operations), preliminary results have shown that CudaGIS is able to achieve signficant speedups over serial CPU implementations. For main-memory based implementations that have already made full use of large-memory capacities of modern hardware, the speedups are in the range of 10-40X. When compared with open source implementations that typically adopt a disk-resident architecture, the speedups can be can be 3-4 orders (1000X-10,000X) where cache-friendly main-memory data structures also have contributed significantly to the overall performance. Since CudaGIS is a main-memory based system, in-

memory data structures are prerequisites for GPU accelerations. To allow CudaGIS to solve larger scale problems that are beyond the memory capacities of CPUs/GPUs (in a way similar to disk-resident systems), we are in the process of investigating a batch-process based approach which is different from buffer management techniques in traditional spatial databases and the relevant techniques will be reported separately.

The rest of the paper is arranged as follows. Section 2 introduces background, motivations and related work. Section 3 presents the designs and implementation details of key system components. Section 4 discusses several high-level design and implementation issues. Finally Section 5 is the summary and the future work plan.

2. BACKGROUND AND RELATED WORK

Geospatial data is among one of the fast growing types of data due to the advances of sensing and navigation technologies and newly emerging applications. First of all, the ever increasing spatial, temporal and spectral resolutions of satellite imagery data have led to exponential growth of data volumes. Second, both airborne and mobile radar/lidar sensors have generated huge amounts of point-cloud data with rich structural information embedded. Third, many mobile devices are now equipped with locating and navigation capabilities by using GPS, cellular and wifi network technologies or their combinations. Considering the large amounts of mobile devices and their users, the accumulated GPS traces, which are essential to understand human mobility, urban dynamics and social interactions, can be equally computing demanding when compared with satellite imagery data and lidar point cloud data. While the traditional infrastructure data, such as administrative regions, census blocks and transportation networks, remain relatively stable in growth when compared with the new types of geospatial data, quite often the new sensing and location data need to be related to the infrastructure data in order to make sense out of them. At the first glance it seems that polygonal data may be the one that has least growth potential from a data acquisition perspective, we argue that the derived data from point (e.g., lidar point clouds, GPS locations), raster (satellite and airborne remote sensing imagery) and polyline (GPS traces) data are best represented as polygons for subsequent analysis. Given the diverse interests of human societies, it is conceivable that the data volumes of polygonal data will also grow fast, if not faster than the other types of geospatial data.Despite certain types of traditional geospatial data remain relatively stable, the newly emerged geospatial data and their applications have imposed signficant computing challenges. Indeed, the gap between the desired computing capabilities and the available ones is increasing rather than decreasing.

There are several noticeable efforts in developing parallel GIS on different types of hardware architectures over the past few decades, such as shared-disk [17], shared-memory [18] and shared-nothing architectures [19-22]. A major research effort in shared disk and shared-nothing based parallel geospatial processing is data partition, replication and declustering to improve I/O efficiency. In recent years, several research works [23-25] on parallel geospatial data processing on shared-nothing clusters follow the MapReduce parallelization model to simplify algorithmic designs and data communications. Another trend we can see is that existing works on parallel geospatial data processing are primarily research driven which little involvement from industries and no commercial products are currently available from leading GIS or spatial database vendors. Indeed, given that Moore's law has held from 1986 to 2002 with processor clock rates double very 18 months, vendors can simply rely on the clock rate improvements of uniprocessors for higher performance without resorting to parallelization.

However, due to the physical limits of CMOS technologies and the power constraints, the growth rates of the processing speed of uniprocessors are decreasing (instead of increasing), especially after 2006 [26]. Given that parallel processors are becoming mainstream, it is natural to seek parallel computing technologies to achieve the desired computing capabilities for large-scale geospatial computing. Currently there are three major leading technologies available, i.e., multi-core CPUs, many-core GPUs and cluster computing (in both grid and cloud settings) and all of them have been applied to geospatial data processing. While we refer to [4] for more details on these three technologies in geospatial applications, we want to stress that they are technologically complementary in nature although they do compete with each other when it comes to system designs for specific applications. We are particularly interested in massively data parallel GPGPU technologies as they represent a radical change to traditional serial computing paradigm which is still the basis for multi-core CPUs and cluster computing where parallelization can be realized at coarse grains.

GPU hardware architectures closely resemble supercomputers[2] which allows/requires fine-grained thread level coordination for data parallelization. This is drastically different from task parallelization that is more suitable for multi-core CPUs and cluster computers. Furthermore, from a practical perspective, as the data communications are becoming increasingly expensive when compared with computation on modern processors/systems [26], GPU's shared-memory architectures allow very fast data communications (up to 400 GB/s) among processing units when compared with cluster computing (~50 MB/s in cloud computing and a few GB/s in grid computing with dedicated high-speed interconnection networks) and multi-core CPUs (a few tens of GB/s), which is desirable for data intensive computing. Finally, in addition to fast floating point computing power and energy efficiency, the large number of processing cores on a single GPU device (3,072 for Nvidia GTX 690 that currently available from the market under $1,000) makes it ideal to solve certain large-scale geospatial problems in a personal computing environment, especially for those that requires extensive visual explorations and user interactions. As discussed in [5], solving larger sub-problems on a single processing unit (a GPU device in this case) will significantly reduce the communication cost in solving a large problem using the MapReduce parallel computing model.

The high-throughput and energy efficiency of GPU computing are largely due to the SIMD (single-instruction-multiple-data) architectural design. To achieve high performance on GPUs, threads or thread groups must be carefully coordinated by ensuring coalesced memory accesses and minimizing branches in code execution. This in turn requires geospatial data structures and algorithms to be parallelization friendly. In the context of spatial query processing for the data management aspect of geospatial computing, hundreds of indexing structures and query algorithms on vector and raster geospatial data have been

proposed over the past few decades [27]. Unfortunately, the majority of them are designed for serial CPU implementations which left their suitability of fine-grained parallelization largely unknown. The CudaGIS system, to the best of our knowledge, is the first in its kind to systematically address the technical challenges in large-scale geospatial data management and processing on GPUs and provide end-to-end solutions for practical geospatial applications.

3. System Design and Implementation

CudaGIS aims at supporting major geospatial data types and a subset of carefully selected operations on these geospatial data types as separate modules. Due to time and resource constraints, each module in CudaGIS is derived from a real application that we have developed over the past few years but we have kept general applicability in mind when they were designed and implemented. In this section, we first provide an overview of the system architecture and highlight the connections between different components before we present the details of the designs and implementations. In a way similar to many modern GIS, these components can be chained to solve a variety of real world geospatial problems. We note that CudaGIS is designed to process large-scale geospatial data that exhibits signficant data parallelism and fits SIMD parallel computing models. As such, it is not our intension to support all geospatial operations as we believe some of them could be more efficiently implemented by using serial or coarse-grained parallelism on multi-core CPUs (or cluster computers). We also note that a system approach has been adopted with a focus on achievable efficiencies. Some of the designs and implementations may not be optimal at present from a research perspective and some are still under developments. Fig. 1 shows the relationships among different geospatial data types and the realized or planned implementations on them in CudaGIS. Numbers within square brackets (e.g., [12]) indicate the references to our publications or technical reports that have the details of the designs, implementations, applications and experiments. Letters within the curly brackets (e.g., {A}) indicate the planned implementations. Some of these components (e.g., polygon overlay) have already been implemented but are skipped in this paper due to space limit. Index structures that have been developed for each data type are listed in the boxes next to the data types. While these index structures can be used to process a single query, to make full use of data parallel computing power on GPUs, we are more interested in batch processing a large number of structurally similar queries and/or large-scale spatial joins [28].

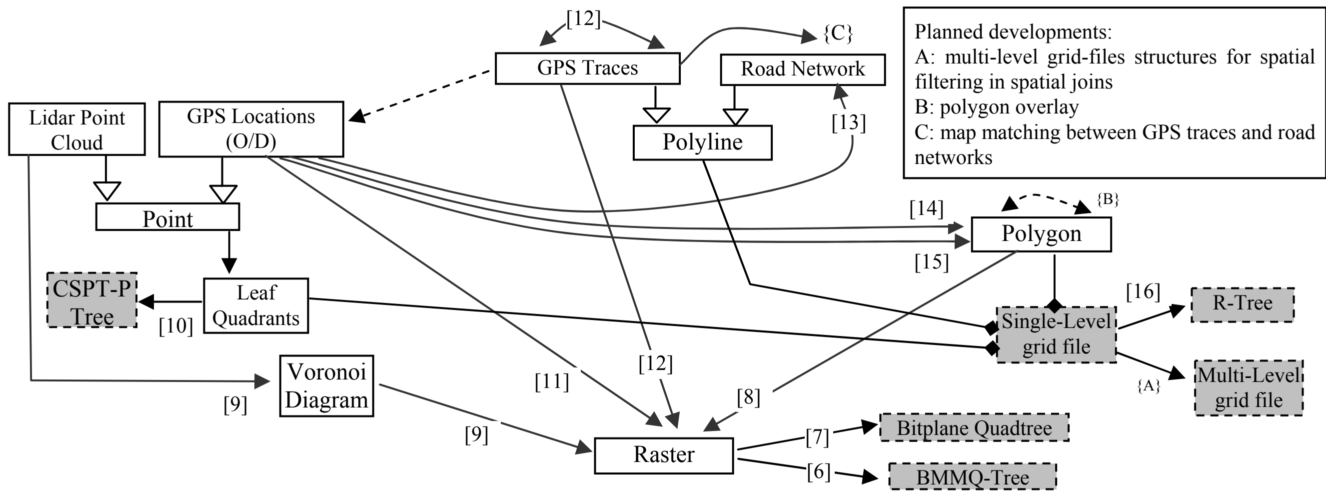


Fig. 1 Supported Geospatial Data Types and Operations in CudaGIS

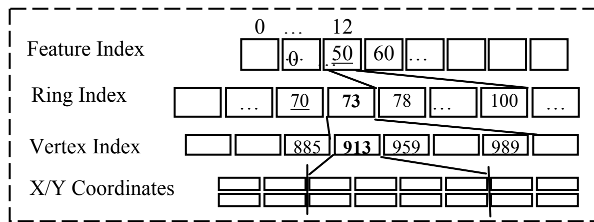
3.1 Efficient In-memory data structures

While points and rasters usually have fixed lengths, polylines and polygons consist of variable length points, which makes their structures irregular. Modern GIS and spatial databases typically adopt an object-relational model and treat polyline and polygons as objects that are used as the minimum units for data accesses and operations. These objects are stored as BLOBs (Binary Large Objects) on disks and the structures of the objects need to be reconstructed when they are streamed from disks to CPU memories. While the design is convenient for page-based buffer management in both an operating system and a database setting, reconstructing variable-length data structures dynamically require extensive memory allocation and deallocaitons which are quite expensive on modern hardware. In contrast, simple arrays are naturally cache friendly and operations based on array scanning can be easier to parallelize. Array based data structures are especially suitable for read-only data in an OLAP (Online Analytical Processing) setting where analytical tasks usually do not need share resources with other tasks that have unpredictable memory requirements (where dynamic resource allocations are more important). As reasonably current desktop computers usually have large memory capacity (4+ GB) which is orders larger than those that are 10 or 20 years ago [26], the ability to use large arrays can signficant reduce the overheads of disk I/Os and improve cache hit rates which will subsequently improve the overall system performance.

Towards this end, we have developed an array-based data layout schema for both polygon and polyline data. According to the Open Geospatial Consortium (OGC) Simple Feature Specification (SFS), a polygonal feature may have multiple rings and each ring consists of multiple vertices. As such, we can form a four level hierarchy from a data collection to vertices, i.e., dataset → feature → ring→ vertex. The hierarchy can be easily extended to polyline data by not requiring the first and the last vertex in a ring to be the same

(enclosed). Five arrays are used for a large polygon/polyline collection. Besides the x and y coordinate arrays, three auxiliary arrays are used to maintain the position boundaries of the aforementioned hierarchy. As shown in Fig. 2, given a dataset ID (0..N-1), the starting position and the ending position of features in the dataset can be looked up in the feature index array. For a feature within a dataset, the starting position and the ending position of rings in the feature can be looked up in the ring index array. Similarly, for a ring within a feature, the starting position and the ending position of vertices belong to the ring can be looked up in the vertex index array. Finally, the coordinates of the ring can be retrieved by accessing the x and y coordinate arrays and the optional timestamp array. While an array of bounding boxes can be associated with the hierarchy at the dataset, feature or ring levels, currently CudaGIS associates the bounding box array at the feature level as bounding boxes at this level can potentially provide better filtering power.

It is easy to see that retrieving coordinates of single or a range of datasets, features and rings can all be done by scanning the five arrays in a cache friendly manner. It is also clear that the number of features in a dataset, the number of rings in a feature and the number of vertices in a ring can be easily calculated by subtracting the two neighboring positions in the respective index array. As such, the array representation is also space efficient. We also note that the data structures and the hierarchy also make it easy for parallelization for a variety of operations. For the example at the right of Fig. 2, given the feature indices of three datasets, assuming each feature is associated with a Minimum Bounding Rectangles (MBR), by chaining a *scatter* primitive, a *max-scan* primitive and a *reduce-by-key* primitive, the MBRs of these three datasets can be derived in parallel. Here a parallel primitive refers to a fundamental algorithm whose behavior is well-understood and can be efficiently executed on parallel hardware. The three parallel primitives we have used in the example are among the many that are supported in quite a few parallel libraries including Thrust that comes with CUDA SDK [3].



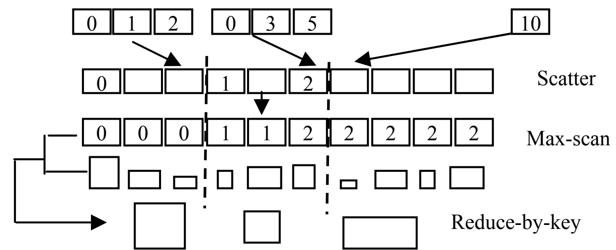


Fig. 2 Array Based In-Memory Data Structures for Polyline/Polygon Data and an Example on Efficient Aggregation Using the Data Structures and Parallel Primitives

3.2 Indexing and Spatial Joins on Vector Data

Indexing and joining vector data are the cornerstones of GIS and spatial databases and we refer to [27, 28] for in-depth surveys on data structures and algorithms designed for CPU based serial implementations. In this subsection, we will be focusing on the designs and implementations of parallel indexing and join processing on GPUs. We are particularly interested in supporting spatial joins in CudaGIS as batched query processing can be treated as spatial joins in a certain way. For single query processing, unless the query covers a large portion of records in the dataset to be queried, serial CPU processing probably can have better performance.

For point data, we have developed an indexing structure called Constrained Spatial Partition Tree for Point data (CSPT-P) on GPUs [10]. We first transform points into Morton codes by following the Z-order and then identify leaf quadrants with a maximum of *K* points in a level-wise top-down manner before a CSPT-P tree is constructed in a level-wise bottom-up manner from the identified leaf quadrants. Both the leaf identification and CSPT-P tree construction processes can be implemented by a few parallel primitives [10].

Besides quadtree indexing for point quadrants, CudaGIS currently supports bulk-loading R-Trees through packing [16] by utilizing GPU's excellent sorting capabilities. Assuming *m* queries are batched in a GPU computing block, two variants of batched R-Tree are supported in CudaGIS, i.e., Depth-First-Search (DFS) based and Breadth-First-Search (BFS) based [16]. The DFS based query adopts a *Count-Then-Move* parallelization strategy (c.f. Section 4.1) by having each thread traverse the query R-Tree independently in a depth-first manner, counting the number of intersections for all the queries in a batch, allocating appropriate output memory space before actually outputting the query results in parallel. The strategy essentially requires two passes of query processing with the first pass for "counting" and the second pass for "outputting". While obviously a query is processed twice which brings some computation redundancy, the DFS strategy does not require any extra memory beyond the output size due to the "counting" pass. The BFS based query has better parallelism in the sense that threads within a computing block are coordinated through a queue. R-Tree nodes that intersect with query MBRs are pushed into the queue and each thread is responsible for processing a single R-Tree node in the queue by examining whether the child nodes intersect with the query MBRs. Different from DFS based query where a thread processes only a single query, in BFS based query, child nodes that intersect with a same query can be processed by multiple threads to achieve higher parallelism. Since the output sizes of individual queries can exceed the pre-allocated queue capacity, an overflow handling mechanism is required for this purpose. Currently CudaGIS simply sets an overflow flag and resorts to DFS based queries for overflowed cases but other mechanisms (such as dynamic memory allocations or buffer page redistributions) are also possible.

There are many types of spatial joins among different types of geospatial data based on different criteria. Take the point data for example, after points are grouped into non-

overlapping quadrants, CudaGIS can join point data with polyline or polygon data based on different distance measurements, point-in-polygon test or nearest neighbor search principles. Similar to serial spatial joins on CPUs [28], spatial join in CudaGIS also has two phases, i.e., filtering phase and refinement phase. In the filtering phase, CudaGIS pairs point quadrants with polylines and polygons where the points and polylines/polygons can potentially satisfy the join criteria. In the refinement phase, for each of such matched pairs, in parallel, each point will be evaluated against the vertices of polylines or polygons based on the join criteria. The evaluated results will have fixed lengths to facilitate parallelization – the threads will need to know exactly where to write out the results. The condition actually can be satisfied in many cases: a boolean variable for point-in-polygon test, a double variable for distance between a point and a polygon/polyline, a (double, integer) pair for nearest-neighbor. The nearest-neighbor search can be extended to K nearest neighbor (KNN) search where the results will be K (double, integer) pairs.

The framework can also be extended for point-to-point, polyline-to-polyline and polygon-to-polygon spatial joins. For example, by treating a segment of GPS trace as a trajectory, we are able to perform trajectory similarity join on GPUs to find two trajectories that are most similar [12] based on the Hausdorff distance. An interesting application of point-to-point join might be Geographically Weighted Regression (GWR) on GPUs [4]. By expanding the leaf quadrants of a point dataset with the GWR bandwidth (W,H), the quadrants can be paired up which guarantees that all the points that are within the (W,H) window of a focal point can be located in one or more of the quadrant pairs in the filtering phase. In the refinement phase, in parallel, each point locates all the points in a (W,H) window and compute its GWR coefficient.

While the join criteria in the refinement phase differ significantly among applications, the criteria for pairing up point quadrants, polylines and polygons based on their MBRs in the filtering phase are relatively uniform which is primarily based on the simple spatial intersection test. Very often we are only interested in geographical features (or geospatial objects) that are no more than distance *D* away. If the expanded MBR of an object with expansion *D* does not intersect with the MBR of another object, then the two objects are at least *D* way and should not be paired. While we are still in the process of designing and implementing efficient data structures for filtering in spatial joins and other types of spatial queries, current CudaGIS utilizes a simple grid-file based data structure for filtering and we would like to provide more details.

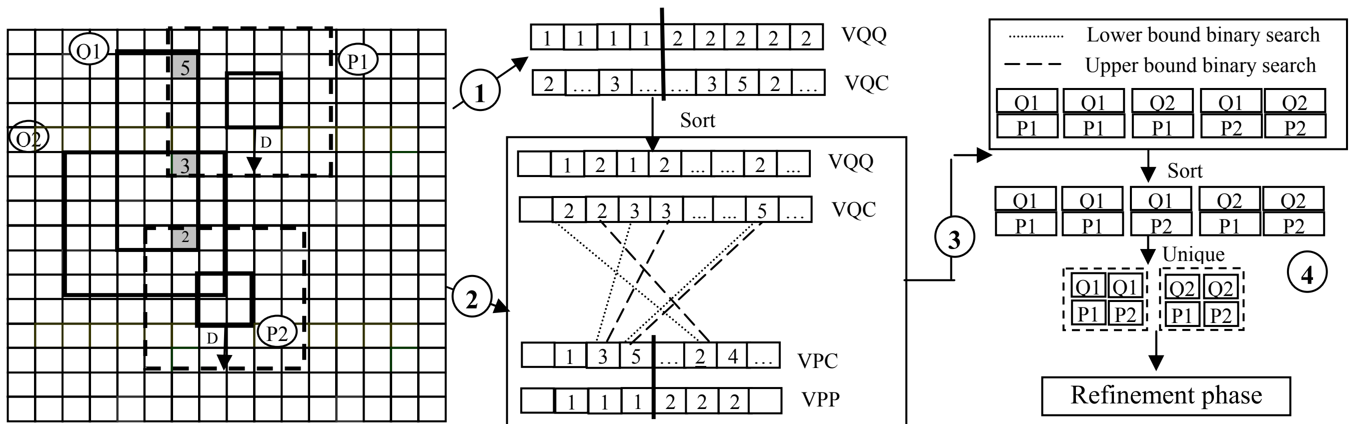


Fig. 3 Illustrate the Single-Level Grid-file based Spatial Filtering in CudaGIS

Given a user-specified grid, the MBRs of the two datasets that participate a spatial join are rasterized to grid cells with each grid cell associated with any (0, 1 or many) numbers of the MBRs. After the rasterization, the pairing process can be reduced to a binary search process as shown in Fig. 3. Assuming *P* represents the pairing dataset and Q represents the dataset to be paired and the rasterization results (after Step 1 and Step 2) are stored in VPC and VQC, respectively. The filtering algorithm in CudaGIS first sorts VQC based on their feature/object identifiers (stored in VQQ) so that feature/object identifiers correspond to a cell identifiers appear consecutively in VQQ. For all the elements in VPC (whose feature/object identifier is stored in VPP), a lower bound binary search and an upper bound binary search are performed in parallel to locate the starting and ending positions of the cell in VQC. If there is a hit, all the elements in VQQ between the starting and ending positions will be paired with the corresponding element in VPP (Step 3). It is clear that there will be duplicates after the initial pairing process and they need to be removed. This can be done by chaining a *sort* and a *unique* primitive using the combination of the P and Q identifiers. Since *sort*, *binary search* and *unique* primitives are supported by parallel library such as Thrust that comes with CUDA SDK, the filtering phase can be efficiently implemented on top of these primitives.

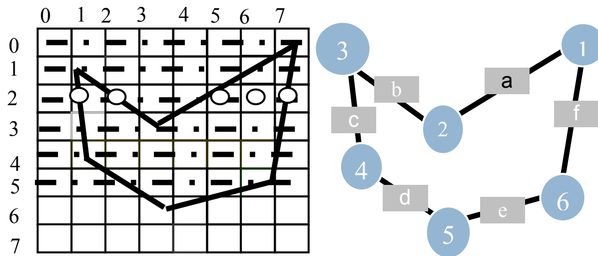
While the design and the implementation are relatively simple and straightforward, a drawback is large memory footprints. It is conceivable that the finer the grid used for rasterization and the larger the MBRs, the greater numbers of the matched pair before Step 4. Our experiments have shown that, when the size of the grid used for rasterization is not properly set, the number of pairs after the initial pairing process can be very large and the GPU memory can be depleted. We are currently extending the single-level grid files data structure to multiple-level one. In multi-level grid file based spatial filtering, MBRs are decomposed to the highest levels of quadrants possible so that O(max(W,H)) cells instead of O(W*H) cells are generated [29] which will also reduce the numbers of duplicated

pairs in spatial joins. As an alternative to grid-file based indexing structures, we are also in the process of extending R-Tree indexing for spatial join [16].

3.3 Raster Indexing and Vector Data Rasterization

For raster data, two indexing approaches have been developed. The Binned Min-Max Quadtree (BMMQ-Tree) is proposed to associate statistics such as minimum and maximum values with quadtree nodes to speed up spatial window queries and Region of Interests (ROIs) types of queries on raster data [6]. The Bitplane Quadtree (BQ-Tree) focuses on efficient coding of large-scale rasters for simultaneously data compression and indexing at the bitplane level [7]. Despite the resulting trees are irregular, both the inputs of BMMQ-Tree construction and BQ-Tree encoding are regular which makes it easier to process. Our experiments have shown that the CudaGIS implementations of both trees have achieved more than 1 billion raster cell per second rate on a single GPU card which has positive implications for practical applications. For BQ-Tree, while still remains to be implemented, dynamically generated binary rasters can be used to efficiently support different types or range queries on raster values in a way similar to bitmap indexing in relational database (see more discussions in the technical report at [7]).

Besides indexing and query processing on rasters themselves, like many GIS, it is important to allow conversions between vector and raster data. We note that the image processing and computer vision community have already developed techniques to achieve the similar goal on GPUs (e.g., region labeling [31]). Currently CudaGIS focuses on rasterization of vector data. It is relatively straightforward to convert points to rasters where a *scatter* primitive can serve the right purpose after converting the point coordinate to a position in a row-major order using a *transform* primitive. However, when more than two points have a same coordinate which will cause a write conflict when executed in parallel, the results are often undefined unless an atomic operation is applied, which can be costly if the resulting raster are dense. Given that many applications actually require deriving statistics (count, average, min/max, standard deviation etc.) for points that fall within raster cells, CudaGIS sorts the point dataset to be rasterized based on the desired raster dimension and uses the non-empty cell positions as the keys to derive the required statistics by using one or more *reduce* parallel primitives. The derived statistics are then associated with corresponding non-empty grid cells while leaving the empty cells with default values [11, 12].



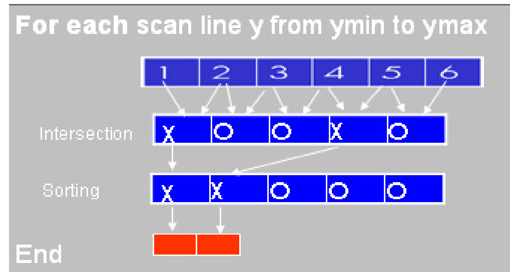


Fig. 4 Illustration of Polygon Rasterization on GPUs in CudaGIS

We have not implemented the rasterization module for polyline data but we believe it can be realized in a similar way as the point data. On the other hand, rasterizing polygon data on GPUs seems to be technically challenging. The polygon rasterization module currently in CudaGIS is based on our preliminary work reported in [8]. The implementation is based on the classic scan line fill algorithm that has also been adopted by the open source GRASS GIS. A polygon is assigned to a computing block and each thread is responsible for a polygon vertex. As illustrated in Fig. 4, all threads, in parallel, loop through the (ymax-ymin+1) scan lines to compute the intersection points and compact them by removing (scan-line, edge) pairs that do not intersect before outputting the intersected points to GPU device memory for actually filling raster cells. To optimize the performance, all threads collaboratively upload the polygon vertices from GPU global memory to shared memory and collaboratively output the computed intersection points to global memory. The vertices in shared memory are re-used (ymax-ymin+1) times to achieve the desired efficiency.

CudaGIS also provides a module to compute a discrete Voronoi diagram from a point dataset and perform Natural Neighborhood Interpolation (NNI) to create a raster based on the Voronoi diagram [9]. The functionality can be considered as a different type of rasterization for points and polygons (the constructed Voronoi diagram). The implementation is motivated by the recent work on Parallel Banding Algorithm (PBA) [31] in computer graphics applications by adding an efficient NNI module on top of the discrete Voronoi diagram generated by PBA. Our preliminary results show that our implementation is 10X+ faster than the state-of-art NNI implementation using OpenGL based GPU graphics APIs [32]. Similar to polygon rasterization, computing Voronoi diagrams and rasterizing Voronoi polygons using GPGPU technologies eliminates the need for graphics hardware context and is more suitable for data management [8].

4. Discussions

Designing and implementing a GIS on a completely new parallel hardware architecture is technically challenging. It is beyond our scope to design and implement all of geospatial operations. Instead, our focus is to understand how commodity parallel hardware can potentially affect geospatial computing, its applications and user communities by providing a prototypical reference implementation with a framework and infrastructure to allow extension, modification or simply testing. We encourage interested readers to join our research and development efforts on exploring the potentials of parallel GIS on GPUs. In addition to sharing the concrete design and implementation experiences detailed in Section 3, we would also like to discuss several more general methodological issues

which can potentially help catalyzing more efficient and effective designs and implementations.

4.1 What are the good parallelization strategies on GPUs?

While the Hadoop implementation of MapReduce represents a simplified yet effective parallelization strategy by regularizing materialized intermediate data to be (key, value) pairs, shared-memory parallel architectures allow more diverse data organizations. Four parallelization techniques have been discussed on multicore CPUs, namely *Independent Output*, *Concurrent Output*, *Count-Then-Move* and *Parallel Buffers* [33]. For the *Independent Output* technique, each thread has a private output buffer for each data partition assigned to it. For the *Concurrent Output* technique, all threads share a same output buffer and threads are coordinated through atomic operations or locks. The *Count-Then-Move* technique has two passes over data partitions with the first pass calculates the positions to which a thread should output its data and the second pass actually outputs the results. The *Parallel Buffer* technique can be considered as a hybridization of the *Independent Output* and *Concurrent Output techniques* where each thread is assigned a private chunk and more chunks are allocated dynamically by coordinating with other threads (concurrent output using atomic operations or locks). While these four techniques are applicable to GPUs in general, since GPUs usually have a much larger number of processing cores and even larger numbers of concurrent threads, the first two techniques are generally not scalable on GPUs. The implementation of CudaGIS has used *Count-Then-Move* technique extensively to achieve better performance while minimizing coding complexity, mostly within computing blocks. For example, BFS based R-Tree query processing (Section 3.2 and [16]), rasterizing MBRs for grid-file based filtering in spatial join (Section 3.2 and [12-15]) and computing the intersections between polygon edges and scan lines in polygon rasterization (Section 3.3 and [8]).

Over the course of designing and implementing CudaGIS, we have also developed a novel parallelization strategy by adopting a transformation approach. The idea is to decompose spatial objects into smallest units at the appropriate granularities to transform a spatial problem into a non-spatial problem and apply parallel primitives, which are often highly parallelizable in implementation, to solve the original spatial problem (e.g., spatial joins). The single-level grid-file based spatial filtering framework (Section 3.2) is a typical example of the strategy. The advantage is that, non-spatial primitive data types (such as integer identifiers) allows only a few limited operations (e.g., equality test) and usually follow a single path when traversing complex data structures (e.g., tree indices). This in turn produces a fixed output size (typically small) and is suitable for parallel executions. Take the single-level grid file based spatial filtering for example, after the two input MBR sets are rasterized into grid cells, binary searches (based on equality test of the cell identifiers) on the grid cells can be used for pairing identifiers in the joining datasets. Unlike search on an R-Tree, a binary search on a sorted integer vector follows a single path and outputs whether there is a hit and the position where the corresponding element is closest to the value being searched.

While parallelization and parallel computing is not a completely new concept from a research perspective, given that it has been only a few years when commodity multi-core CPUs and many-core GPUs became the mainstream processors in the market and there are even fewer research and developments on paralleling geospatial computing (other than local and focal operations on rasters), there is still considerable amount work to understand both the geospatial problems and parallel hardware characteristics and design better parallelization strategies for domain-specific problems.

4.2 Geospatial-specific parallel primitives: essential or cosmetic?

The design and implementation of CudaGIS benefits from the Thrust parallel primitives that come with CUDA SDK significantly. While our initial developments are purely based on the native parallel programming language (CUDA), as the project goes on, we have been using parallel primitives more frequently as the targeted geospatial operations and their implementation get more complex. Using parallel primitives not only reduces coding time and code complexity but also improves the maintainability of the increasingly larger CudaGIS codebase. Furthermore, as discussed previously, the experiences that we have learnt from using parallel primitives actually have motivated us to develop spatial data structures and query processing algorithms that are more friendly to using parallel primitives. On the other hand, the primitives that are available in most parallel libraries (including Thrust) are predominately designed for one dimensional vectors or arrays. Although there are quite a few Space Filling Curves (SFC) based orderings available to transform multi-dimensional geospatial data to one-dimensional data with different characteristics [34] in order to apply such 1D parallel primitives, the semantics need to be maintained by developers can be cumbersome in practical developments. We are considering reorganize the modules that have been implemented in CudaGIS as a set of geospatial-specific parallel primitives so that they can be invoked in a way similar to the 1D primitives.

5. Future Work Plans

In this report, we have adopted a system oriented approach to introducing the designs and the implementations of several key components in CudaGIS, the first general purposed parallel GIS on commodity GPUs. The realized modules include indexing of point, raster and vector (polyline/polygon) data, single-level grid file based and R-Tree based spatial join frameworks and their applications to different types of spatial joins, and, the conversion from point/polygon data to rasters through rasterization and natural neighbor interpolation.

For the future work, we would like to improve the existing design and implementations and efficiently implement the designs for planned modules, including the increasing refining grid file based spatial join framework, polygon overlay operations and indexing the internals of complex polygons for efficient spatial queries and joins. We will also add new modules that are essential for geospatial computing. Finally we plan to develop data generators to produce synthetic data for testing purpose, and, add a front module to support SQL query statements to make CudaGIS more database friendly.

References

1. A. Clematis, M. Mineter, and R. Marciano. High performance computing with geographical data. Parallel Computing, 29(10):1275–1279, 2003.
2. Hong, S., Kim, S. K., et al., 2011. Accelerating CUDA graph algorithms at maximum warp. Proceedings of the 16th ACM symposium on Principles and practice of parallel programming.
3. http://www.nvidia.com/object/cuda_home_new.html
4. Zhang, J. 2010. Towards Personal High-Performance Geospatial Computing (HPC-G): Perspectives and a Case Study. Proceedings of ACM HPDGIS workshop.
5. Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, Jeffrey D. Ullman: Upper and Lower Bounds on the Cost of a Map-Reduce Computation CoRR abs/1206.4377: (2012)
6. Zhang, J., You., S. and Gruenwald, (2010). Indexing Large-Scale Raster Geospatial Data Using Massively Parallel GPGPU Computing. Proceedings of ACMGIS 2010.
7. Zhang, J., You., S. and Gruenwald, (2011). Parallel Quadtree Coding of Large-Scale Raster Geospatial Data on GPGPUs. Proceedings of ACMGIS 2010. Expanded version at http://www-cs.ccny.cuny.edu/~jzhang/papers/ACMGIS11_Extended.pdf
8. Zhang J. (2011). Speeding Up Large-Scale Geospatial Polygon Rasterization on GPGPUs. Proceedings of ACM HPDGIS Workshop.
9. You, S., Zhang, J. (2012). Constructing Natural Neighbor Interpolation Based Grid DEM Using CUDA. Proceedings of COM.Geo Conference.
10. Zhang, J. and Gruenwald, (2012). Spatial Indexing of Large-Scale Geo-Referenced Point Data on GPGPUs Using Parallel Primitives. http://geoteci.engr.ccny.cuny.edu/primcsptp/CSPTP_tr.pdf.
11. Zhang, J., Gong, H., Kamga, C. and Gruenwald L. (2012). U^{2}SOD-DB: A Database System to Manage Large-Scale Ubiquitous Urban Sensing Origin-Destination Data. Proceedings of ACM SIGKDD UrbComp workshop
12. Zhang, J., You., S. and Gruenwald, (2012). U^{2}STRA: High-Performance Data Management of Ubiquitous Urban Sensing Trajectories on GPGPUs. To appear in Proceedings of ACM CDMW workshop. http://geoteci.engr.ccny.cuny.edu/pub/u2stra_tr.pdf
13. Zhang, J., You., S. and Gruenwald, (2012). High-Performance Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs. To appear in Proceedings of ACM DOLAP workshop. http://www-cs.ccny.cuny.edu/~jzhang/papers/aggr_tr.pdf
14. Zhang, J., You., S. and Gruenwald, (2012). High-Performance Spatial Join Processing on GPGPUs with Applications to Large-Scale Taxi Trip Data. http://www-cs.ccny.cuny.edu/~jzhang/papers/nnsp_tr.pdf
15. Zhang, J. and You., S. (2012). Speeding up Large-Scale Point-in-Polygon Test Based Spatial Join on GPUs. Technical report online at http://geoteci.engr.ccny.cuny.edu/pub/pipsp_tr.pdf
16. You., S. and Zhang, J. (2012). Batched R-Tree Queries on GPUs. Technical report at http://geoteci.engr.ccny.cuny.edu/pub/rtree_tr.pdf.
17. Patel, J. M. and DeWitt, D. J. (2000). Clone join and shadow join: two parallel spatial join algorithms. Proceedings of ACMGIS.
18. Shekhar, S., Ravada, S., Kumar, V., Chubb, D. and Turner, G. (1996). Parallelizing a GIS on a Shared Address Space Architecture. IEEE Computer 29(12): 42-48.
19. Hoel, E. G. and Samet, H., 1994. Performance of Data-Parallel Spatial Operations. Proceedings of VLDB Conference.
20. Brinkhoff, T., Kriegel, H.-P. and Seeger, B. (1996). Parallel Processing of Spatial Joins Using R-trees. Proceedings of IEEE ICDE Conference.
21. Patel, J., Yu, J., et al (1997). Building a scalable geo-spatial DBMS: technology, implementation, and evaluation. Proceedings of SIGMOD conference.
22. Zhou, X., Abel, D. J. and Truffet, D. (1998). Data Partitioning for Parallel Spatial Join Processing. Geoinformatica 2(2): 175-204.
23. Zhang, S., Han, J., Liu, Z., Wang, K. and Xu, Z. (2009). SJMR: Parallelizing spatial join with MapReduce on clusters. Proceedings of IEEE International Conference on Cluster Computing.
24. Liu, Y., Wu, K., Wang, S., Zhao, Y. and Huang, Q. (2010). A MapReduce approach to Gi*(d) spatial statistic. Proceedings of ACM HPDGIS Workshop.
25. Zhang, C., Li, F. and Jestes, J. (2012). Efficient parallel kNN joins for large data in MapReduce. Proceedings of EDBT Conference
26. Hennessy, J.L. and Patterson, D. A, 2011. Computer Architecture: A Quantitative Approach (5th ed.). Morgan Kaufmann.
27. Gaede V. and Gunther O., 1998. Multidimensional access methods. ACM Computing Surveys 30(2), 170-231
28. Jacox, E. H. and Samet, H. (2007). Spatial join techniques. ACM Transaction on Database System 32(1).
29. Tsai, Y. H., Chung, K. and Chen, W. Y. (2004). A strip-splitting-based optimal algorithm for decomposing a query window into maximal quadtree blocks. IEEE TKDE 16(4): 519-523.
30. Stava, O. and Benes, B (2010). Connected component labeling in CUDA. in Hwu.,W. W. (Ed.), GPU Computing Gems.
31. Cao, T.-T., Tang, K., Mohamed, A. and Tan, T.-S. (2010). Parallel Banding Algorithm to compute exact distance transform with the GPU. Proceedings of ACM I3D Workshop.
32. Beutel, A., Molhave, T. et al, (2011). TerraNNI: natural neighbor interpolation on a 3D grid using a GPU. Proceedings of ACMGIS.
33. Cieslewicz, J. and Ross, K. A. (2008). Data partitioning on chip multiprocessors." Proceedings of DaMoN Workshop.
34. Mokbel, M. and Aref, W. (2011). Irregularity in high-dimensional space-filling curves. Distributed and Parallel Databases 29(3): 217-238.