Large-Scale Spatial Data Processing on GPUs and GPU-Accelerated Clusters[*]

Jianting Zhang[1 2], Simin You[2],Le Gruenwald[3]
[1]Department of Computer Science, City College of New York, USA
[2]Department of Computer Science, CUNY Graduate Center, USA
[3]School of Computer Science, University of Oklahoma, USA

Abstract

The massive data parallel computing power provided by inexpensive commodity Graphics Processing Units(GPUs) makes large-scale spatial data processing on GPUs and GPU-accelerated clusters attractive from both a research and practical perspective. In this article, we report our works on data parallel designs of spatial indexing, spatial joins and several other spatial operations, including polygon rasterization, polygon decomposition and point interpolation. The data parallel designs are further scaled out to distributed computing nodes by integrating single-node GPU implementations with High-Performance Computing (HPC) toolset and the new generation in-memory Big Data systems such as Cloudera Impala. In addition to introducing GPGPU computing background and outlining data parallel designs for spatial operations, references to individual works are provided as a summary chart for interested readers to follow more details on designs, implementations and performance evaluations.

Keywords: Spatial Data, Large-Scale, Data Parallel Design, GPGPU

1 Introduction

Geospatial data is 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. Furthermore, polygons derived from point data clustering (e.g., lidar point clouds, GPS locations) and raster data segmentations (e.g., satellite and airborne remote sensing imagery) are likely to be even larger in volumes and computing-intensive. To efficiently process these large-scale, dynamic and diverse geospatial data and to effectively transform them into knowledge, a whole new set of data processing techniques are thus required.

Existing Big Data techniques include algorithmic improvements to reduce computation complexity, developing approximate algorithms to trade accuracy with efficiency and using parallel and distributed hardware and systems. As parallel hardware, such as multi-core CPUs and many-core Graphics Processing Units (GPUs), is now mainstream commodity [4], parallel and distributed computing techniques on top of the inexpensive commodity hardware are attractive, especially for applications that require exact computation while little room has been left for algorithmic improvements. In the past few years, the simplicity of the MapReduce computing model and its support in the open source Hadoop system have made

[*]Partially supported by NSF Grants IIS-1302423 and IIS-1302439.

it attractive to develop distributed geospatial computing techniques on top of MapReduce/Hadoop [2]. The success of SpatialHadoop [3] and HadoopGIS [1] has demonstrated the effectiveness of MapReduce-based techniques for large-scale geospatial data management where parallelisms are typically identified at the spatial partition level which allows adapting traditional serial algorithms and implementations within a partition.

While MapReduce/Hadoop based techniques are mostly designed for distributed computing nodes each with one or multiple CPU cores, the General Purpose computing on Graphics Processing Units (GPGPUs) techniques represent a significantly different parallel computing scheme. GPU hardware architectures adopt a shared-memory architecture closely resembles traditional supercomputers [5], which requires fine-grained thread level coordination for data parallelization. From a practical perspective, as the data communications are becoming increasingly expensive when compared with computation on modern processors/systems [4], GPU shared-memory architectures allow very fast data communications (currently up to 672 GB/s for Nvidia GTX Titan Z[1]) 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 (5,760 for Nvidia GTX Titan Z) makes it ideal for processing geospatial data which is typically both data-intensive and compute-intensive. Nevertheless, from a research perspective, techniques based on a single GPU device have limited scalability which makes it desirable to scale-out the techniques to cluster computers with multiple-nodes and multiple GPU devices.

In this paper, we report our work on data parallel designs for several geospatial data processing techniques. By further integrating these GPU-based techniques with distributed computing tools, including Message Passing Interface (MPI[2]) library in the traditional High-Performance Computing (HPC) clusters and newer generation of Big Data systems (such as Impala[3] and Spark[4]) for Cloud computing, we are able to scale the data parallel geospatial processing techniques to cluster computers with good scalability. While we are aware of the complexities in developing a full-fledged GIS and/or a Spatial Database on GPUs, our research bears three goals: 1) to demonstrate the feasibility and efficiency of GPU-based geospatial processing, especially for large-scale data, 2) to develop modules for major geospatial data types and operations that can be directly applied to popular practical applications, such as large-scale taxi trip data and trajectory data, and 3) to develop a framework to integrate multiple GPU-based geospatial processing modules into an open system that can be shared by the community. We have developed several modules (as summarized in Fig. 4 in Section 3), over the past few years. We are in the process of integrating these modules under a unified framework and developing new modules to further enhance functionality. Interested readers can follow the respective references for more details.

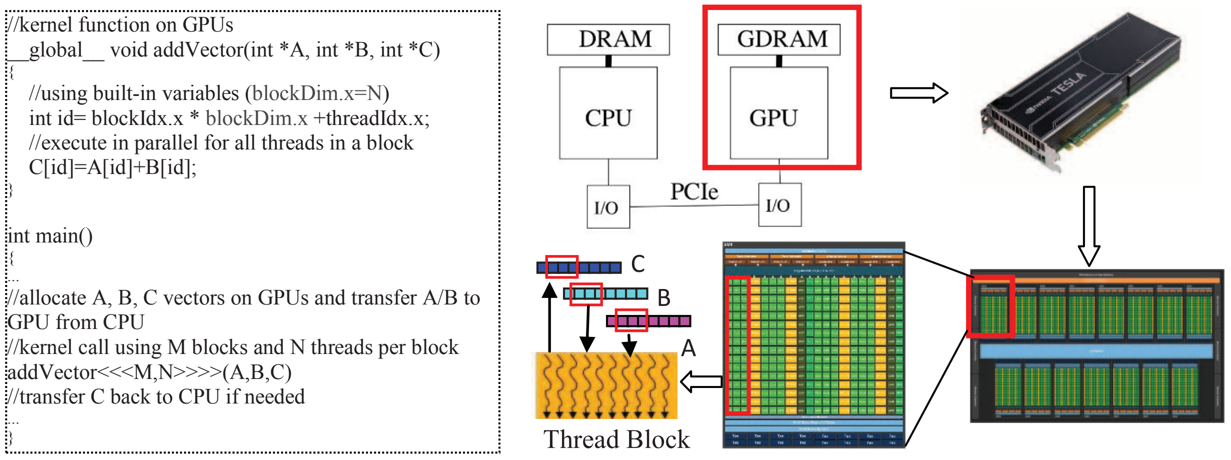
For the rest of the paper, Section 2 provides a brief introduction to GPGPU computing; Section 3 introduces our data parallel designs and GPU implementations; Section 4 presents the high-level designs and implementations on integrating single-node GPU techniques for scaling out geospatial processing on GPU-accelerated clusters; and finally Section 5 is the summary and future work directions.

2 GPGPU Computing

Modern GPUs are now capable of general computing [4]. Due to the popularity of the Compute Unified Device Architecture (CUDA) [6] on Nvidia GPUs, which can be considered as a C/C++ extension, we will mostly follow CUDA terminologies to introduce GPU computing. Current generations of GPUs are used as accelerators of CPUs and data are transferred between CPUs and GPUs through PCI-E buses. The Nvidia Tesla K10 GPU shown in the top-right side of Fig. 1 has 15 Streaming Multiprocesors (SMXs) with each SMX having 192 processing cores. Since 32 processing cores form a warp and warps are used as the basic units for scheduling, GPUs can be viewed as Single Instruction, Multiple Data (SIMD) devices [4]. A multiprocessor can accommodate multiple thread blocks with each thread block having one or more warps through time multiplexing to hide I/Os and other types of latencies. For example, Tesla K10 GPU supports up to 2048 concurrent threads (i.e., 64 warps) per SMX. All the 32 threads in a warp execute the same instruction and the performance is maximized when there are no code branches within the warp; otherwise the branches will be serialized and the performance can be poor.

GPUs that are capable of general computing are facilitated with Software Development Toolkits (SDKs) provided by hardware vendors. The left side of Fig. 1 shows a simple example on summing up two vectors (*A* and *B*) into a new vector

[1]http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-z
[2]http://en.wikipedia.org/wiki/Message_Passing_Interface
[3]http://impala.io
[4]http://spark.apache.org



```
//kernel function on GPUs
__global__ void addVector(int *A, int *B, int *C)
{
    //using built-in variables (blockDim.x=N)
    int id= blockIdx.x * blockDim.x +threadIdx.x;
    //execute in parallel for all threads in a block
    C[id]=A[id]+B[id];
}

int main()
{
...
//allocate A, B, C vectors on GPUs and transfer A/B to
GPU from CPU
//kernel call using M blocks and N threads per block
addVector<<<M,N>>>>(A,B,C)
//transfer C back to CPU if needed
...
}
```

Figure 1: Illustration of GPU hardware Architecture and Programming Model

(*C*) in CUDA. The top part of the code shows the kernel function to be invoked by the main function in the lower part of the code segment. The whole computing task is divided into *M* blocks with each being assigned to a thread block with *N* threads. Within a thread block, an index can be computed to address the relevant vector elements for inputs/outputs based on its thread identifier (*threadIdx.x*) and block identifier (*blockIdx.x*), which are automatically assigned by the hardware scheduler, and block dimension (*blockDim.x*) which is specified when the kernel is invoked. While we use a 1D example in Fig. 1, CUDA supports up to three dimensions.

Parallelism is fundamental to data processing on parallel hardware. While coarse-grained parallelization can be used to create parallel tasks and exploit existing scheduling algorithms for parallel execution, the reverse is not true. Roughly speaking, the CUDA computing model for GPUs supports both task parallelism at the thread block level and data parallelism at the thread level. For a single GPU kernel designed for solving a particular problem, the boundary between task and data parallelism can be configured when the kernel is invoked (the lower-left part of Fig. 1). However, to maximize performance, data items should be grouped into basic units that can be processed by a warp of threads (which are dynamically assigned to processor cores) without incurring significant divergence. Instead of accessing data items sequentially that exhibits significant temporal locality that is optimal on CPUs, when nearby threads in a warp access a continuous block of data items in GPU device memory, the individual GPU memory accesses by the warp of threads can be combined into fewer memory accesses (coalesced memory accesses). This GPU characteristic requires a careful design of the layouts of multi-dimensional spatial data structures and their access patterns when developing spatial algorithms.

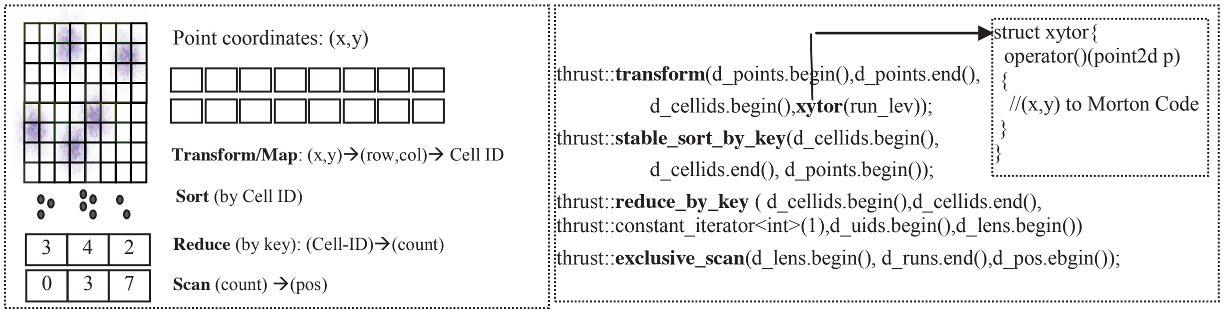
The unique hardware features and large tunable parameter space have made developing efficient GPU programs challenging. Using local, focal, zonal and global classification of geospatial operations [11] for both vector and raster data, as local operations only involve independent individual data items and focal operations mostly involve a bounded small number of neighboring items, they are relatively easy to be parallelized on GPUs. However, zonal operations (such as generating elevation distribution histograms for raster cells in polygons) and global operations (such as indexing vector geometry as trees) typically involve geometrical objects with variable numbers of vertices and may be spatially related to unbounded numbers of geometrical objects, such as joining two polygon datasets based on point-polygon test or two polyline datasets based on distance or similarity measures. The irregularities of data layout and data access patterns in such spatial operations have made it technically very challenging to design and implement efficient geospatial algorithms on GPU hardware.

While we have developed some geometrical algorithms on GPUs using CUDA directly at the beginning of our explorations of massive data parallel computing power for geospatial processing, we gradually realized that the straightforward approach is not productive. Instead, we have chosen to adopt a parallel primitive based approach whereas possible to reduce implementation complexity and improve development productivity. Parallel primitives refer to a collection of fundamental algorithms that can run on parallel machines [8]. The behaviors of popular parallel primitives on 1D arrays are well-understood. Parallel primitives usually are implemented on top of native parallel programming languages (such as CUDA) but provide a set of simple yet powerful interfaces (or APIs) to end users. Technical details are hidden from end users and many parameters that are required by native programming languages are fine-tuned for typical applications in parallel libraries so that users do not need to specify such parameters explicitly.

3 Data Parallel Designs and Single-Node GPU-Implementations

Due to space limit, we will use a grid-file based indexing as an example to illustrate the idea of parallel primitives based data parallel designs and their implementations on GPUs. We then provide a summary chart for our existing designs and implementations and refer the readers to respective references for details.

Consider indexing a large set of points using the classic grid-file structure [9]. While serial algorithms and their implementations loop through all the points and determine the grid cell that each point should be associated with, as shown in Fig. 2, we use four parallel primitives for this purpose: *transform*, *(stable) sort*, *reduce (by key)* and *(exclusive) scan*. The *transform* primitive (similar to the *Map* function in *MapReduce*) generates Morton codes [9] that are used as grid cell identifiers for all points at a pre-defined resolution level; the *sort* primitive sorts points based on the cell IDs; the *reduce (by key)* primitive counts the number of points within each grid cell; and finally the *(exclusive) scan* primitive computes the prefix-sums of the numbers of points in all grid cells which are the starting positions of the points in the sorted point data vector. The primitives are executed by GPU hardware in parallel using their most efficient implementations which are transparent to algorithm and application developers. In fact, the current Thrust[5] parallel library (which comes with CUDA SDK) uses radix sort for the sort primitive. Although quicksort is known to be efficient on CPUs, radix sort is considered to be more efficient on GPUs.


Point coordinates: (x,y)
3 4 2
0 3 7
Transform/Map: (x,y)→(row,col)→ Cell ID
Sort (by Cell ID)
Reduce (by key): (Cell-ID)→(count)
Scan (count) →(pos)

```
thrust::transform(d_points.begin(),d_points.end(),
            d_cellids.begin(),xytor(run_lev));
thrust::stable_sort_by_key(d_cellids.begin(),
            d_cellids.end(), d_points.begin());
thrust::reduce_by_key ( d_cellids.begin(),d_cellids.end(),
thrust::constant_iterator<int>(1),d_uids.begin(),d_lens.begin())
thrust::exclusive_scan(d_lens.begin(), d_runs.end(),d_pos.ebgin());
```

```
struct xytor{
  operator()(point2d p)
  {
   //(x,y) to Morton Code
  }
}
```

Figure 2: Data Parallel Design and Implementation of Grid-File Point Indexing on GPUs

We have designed indexing techniques for rasters [21, 22, 19], points [18, 26] and Minimum Bounding Boxes [26, 13] using Grid-Files [26], Quadtrees [18, 21, 22, 19] and R-Trees [13]. We have also developed a GPU-based spatial join framework to join two indexed spatial datasets based on point-in-polygon tests [18], point-to-polyline distance [26], polyline-to-polyline similarity [23] with applications to spatiotemporal aggregation of large-scale taxi-trip data [18], trip-purpose analysis [25], trajectory similarity query [23] and global biodiversity studies [20]. Fig. 3 illustrates our framework for spatial join processing on GPUs using grid-file indexing. After MBRs are rasterized into grid cells, the middle part of Fig. 3 illustrates how to use parallel primitives including *sort*, *binary searches* and *unique* to pair up polygons or polylines (i.e., spatial filtering) for the subsequent spatial refinement. While using quadtrees or R-trees for spatial filtering may require different parallel primitives (we refer to [18, 13] for details), the grid-file based spatial filtering essentially transforms a spatial query (filtering) problem into a relational equi-join problem which has been shown to be effective on GPUs [26]. The lower part of Fig. 3 shows four types of spatial refinement operations which can be realized efficiently on GPUs and we refer to [18, 26, 25] for details.

In addition to spatial indexing and query processing, which are important components in spatial databases, our research also involves several modules that are more related to pre-processing and post-processing as well as data conversions on GPUs, which are essential in a GIS environment. The work on natural neighbor based spatial interpolation for lidar data [12], although is implemented using CUDA directly for performance, also adopts a parallel primitive approach internally at the thread block level. The spatial interpolation module naturally bridges point data and raster data which makes it possible to apply existing techniques for rasters for point data. Similarly the GPU-based polygon rasterization technique in [16] bridges polygons and rasters. In observing that indexing polylines and polygons at MBR level might be limited by high false positives and result in low indexing power, we have developed a polygon decomposition technique which can decompose polygons into quadrants [24]. The decomposed polygons can be used for both indexing and approximating polygons in certain queries. We have also performed preliminary designs and implementations for polygon overlays [28], Geographical Weighted Regression (GWR) analysis [15] and map-matching for trajectories on GPUs. Fig. 4 provides

[5]https://thrust.github.io

a summary chart of our existing works where shaded rectangles represent indexing techniques, diamond-headed edges represent spatial applications and bracketed numbers represent publication sources for more details.

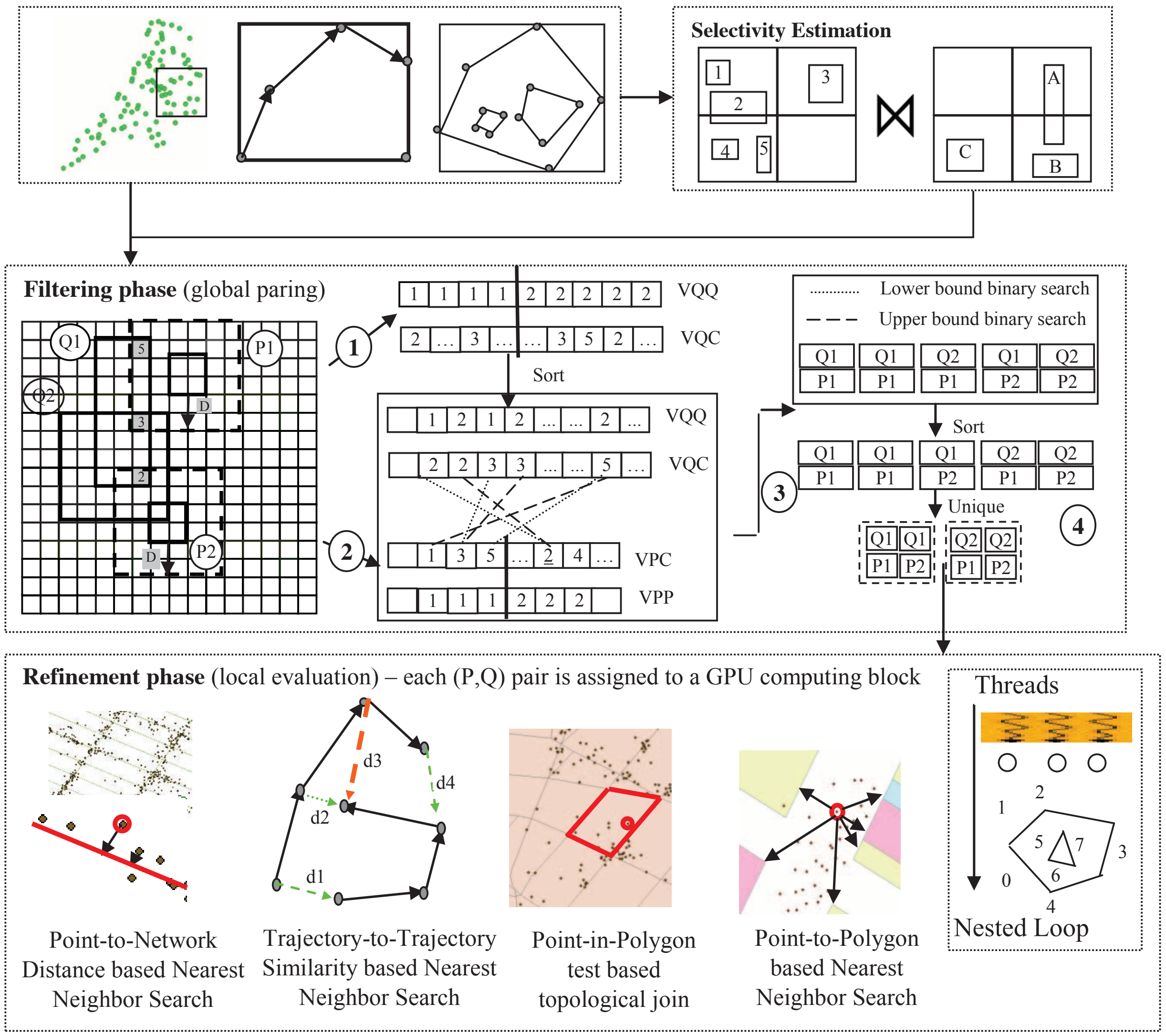

Selectivity Estimation
Filtering phase (global paring)
VQQ
VQC
Sort
VPC
VPP
Lower bound binary search
Upper bound binary search
Unique
Refinement phase (local evaluation) – each (P,Q) pair is assigned to a GPU computing block
Threads
Nested Loop
Point-to-Network Distance based Nearest Neighbor Search
Trajectory-to-Trajectory Similarity based Nearest Neighbor Search
Point-in-Polygon test based topological join
Point-to-Polygon based Nearest Neighbor Search

Figure 3: A Framework of Spatial Join Query Processing on GPUs using Grid-File Indexing

4 Scaling-out to GPU-Accelerated Clusters

To further improve the performance of large-scale geospatial data processing, it is essential to share workloads among distributed computing nodes that are equipped with GPUs for scalability. As discussed in Section 1, it is nontrivial to design and implement efficient distributed computing systems while existing Big Data systems typically do not support spatial data processing. In observing that improving single-node efficiency using GPUs can significantly reduce inter-node data communications [10], we believe that integrating our single-node GPU-based geospatial processing techniques with distributed computing techniques can be competitive with existing solutions (such as HadoopGIS [3] and SpaitalHadoop [1]).

Towards this goal, we have experimented two approaches: one using the MPI parallelization software stack available on the ORNL Titan supercomputer and one using the open source Cloudera Impala [7]. Fig. 5 illustrates the framework of the first approach where the NASA SRTM 30-meter DEM rasters with 20 billion raster cells are first divided into raster titles and the tiles are paired up with county MBRs. The pairs are partitioned and the partitions are then sent to Titan computing nodes through MPI APIs (Left Fig. 5). On each computing node, a raster tile is further divided into blocks and

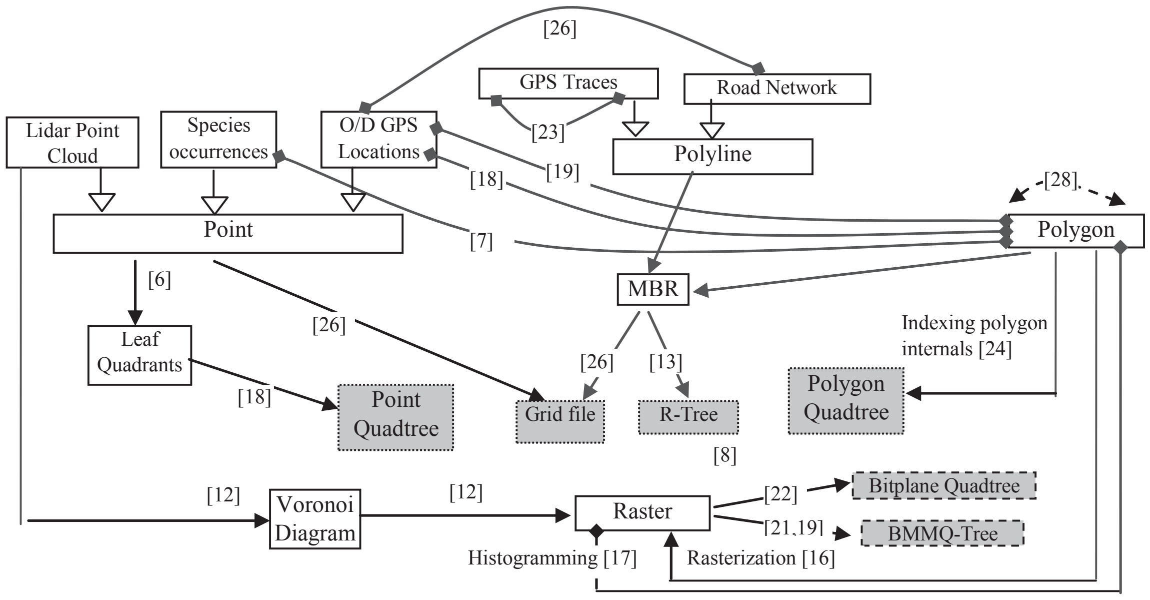


Figure 4: Summary Chart of GPU-based Spatial Operations for Different Spatial Data Types and Datasets

elevation histograms for the raster blocks can be efficiently computed on a GPU in parallel. Subsequently, raster blocks are tested whether they are completely inside a polygon or intersect with polygons after spatial filtering based on MBRs,. For raster blocks that are completely within a polygon, their histograms will be merged. For raster blocks that overlap with polygons, raster cells in the blocks are then treated as points and point-in-polygon tests are performed in parallel on GPUs as described in [18]. The per-polygon histograms are subsequently updated based on the test results. While we refer to [17] for details on the designs, implementations and performance evaluations, as a summary of results, we were able to generate elevation histograms for 3000+ US counties over 20 billion raster cells in about 10 seconds using 8 Titan nodes. We are in the process of generalizing the approach to support more general Zonal Statistics spatial operations which are popular in GIS applications [11].

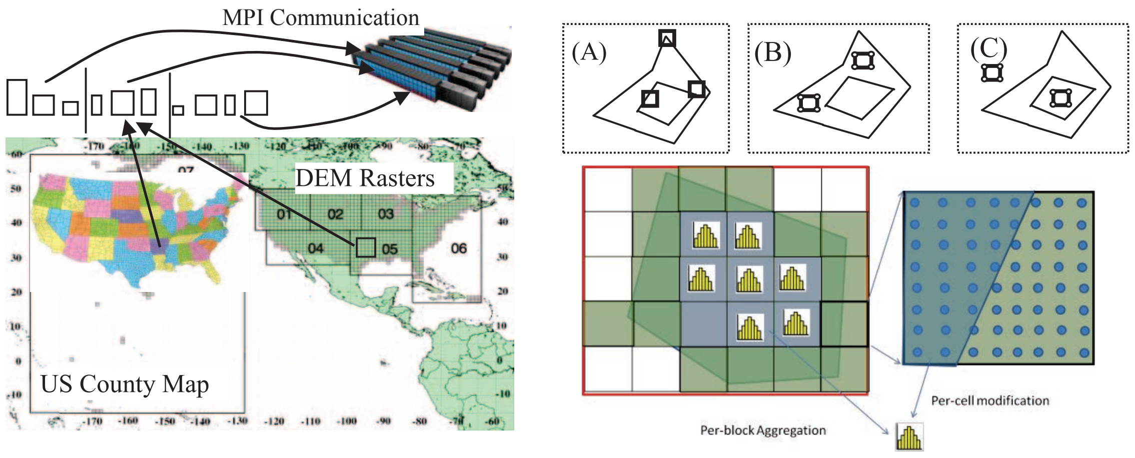


Figure 5: Zonal Statistics on US Counties over NASA SRTM DEM Rasters on Titan Supercomputer using MPI with GPU-based Histogramming, Point-in-Polygon Test and Box-in-Polygon Test

The second approach we have adopted is to extend Cloudera Impala to support spatial query in SQL. Different from traditional distributed computing that utilizes MPI, data communication in Impala is based on Apache Thrift[6] and is tightly embedded into SQL physical execution plan. As shown in Fig. 6, in the ISP prototype system we have developed, three additional extensions are implemented in order to reuse the Impala infrastructure for distributed spatial query processing.

[6]http://thrift.apache.org

First, we modify the Abstract Syntax Tree (AST) module of Impala frontend to support spatial query syntax. Second, we represent geometry of spatial datasets as strings to support spatial data accesses in Impala and prepare necessary data structures for GPU-based spatial query processing. Third, we have developed a *SpatialJoin* module as a subclass of *ExecNode* to extract data from both left and right sides in a spatial join in batches before the data is sent to GPU for query processing. We again refer to our technical report [27] for details while only provide a summary of performance evaluation here due to space limit.

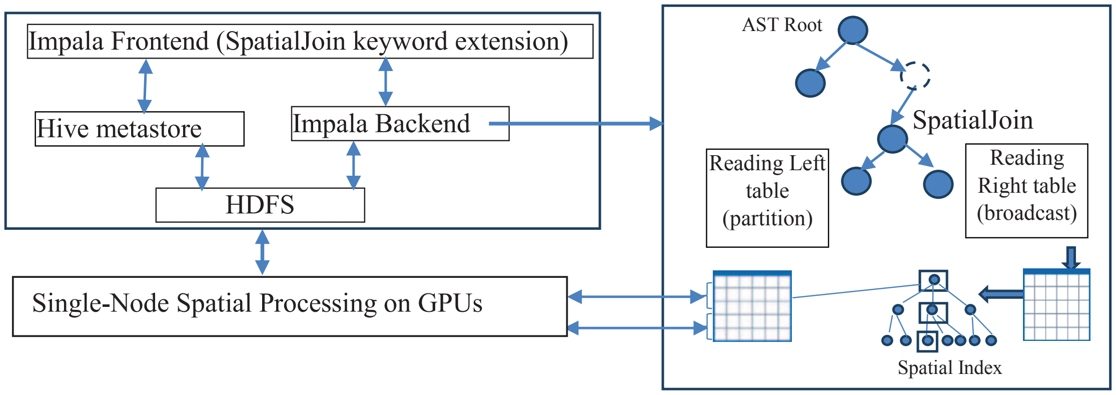

Impala Frontend (SpatialJoin keyword extension)
Hive metastore
Impala Backend
HDFS
Single-Node Spatial Processing on GPUs
AST Root
SpatialJoin
Reading Left table (partition)
Reading Right table (broadcast)
Spatial Index

Figure 6: ISP System Architecture and Components

The current GPU SDKs have limited support for JAVA, Scala and other languages other than C/C++, which makes it difficult to integrate our GPU-based implementations with Hadoop and Spark for scalability. However, we have observed that our data parallel designs and their implementations on top of the Thrust parallel library have strong connections with the built-in vector functions (e.g., *map*, *reduce* and *sort*) in Scala (and similarly Java 8). The connections have motivated us to develop SpatialSpark [14] to process spatial queries directly on Spark, a popular and high-performance in-memory Big Data system developed using Scala and Java. While the end-to-end performance of SpatialSpark is largely affected by the underlying geometry library (JTS[7] in this case) which dominates the spatial join query runtimes, the simple implementations and high-performance have made the implementation attractive for Cloud deployment [14]. This subsequently has motivated us to develop a data communication infrastructure similar to Spark (and Akka[8] that Spark depends on) to natively support large-scale geospatial processing on GPU-accelerated clusters. By developing more semantics-aware spatial data partition and communication primitives and extending the row-batch based asynchronous data processing framework in Impala [7] to semi-structure data (such as spatial data and trajectory data), we hope the new designs and implementations can bring higher efficiency and scalability for large-scale geospatial processing.

5 Conclusion and Future Work

Large-scale geospatial data in newly emerging applications require new techniques and systems for better scientific inquiries and decision making. Efficient and scalable processing of large-scale geospatial data on parallel and distributed platforms is an important aspect of Big Data research. In this paper, we present our work on parallel designs and implementations of geospatial processing algorithms and systems on GPUs and GPU-accelerated clusters for both efficiency and scalability. Experiments on several large-scale geospatial data have demonstrated orders of magnitude speedups when compared with traditional techniques on single CPU cores and have shown great potentials in significantly speeding up a wide range of geospatial applications.

For future work, first of all, we would like to investigate on both generic and spatial specific parallel primitives for multi-dimension data as most of existing primitives in parallel libraries (including Thrust) are designed for one dimensional vectors. Second, while we have successfully scaled out our data parallel designs from a single node to distributed nodes , there is still considerable room for optimizations to further improve scalability by incorporating spatial processing domain semantics. Finally, compared with existing spatial databases that provide declarative SQL interfaces and GIS that provide intuitive graphics interfaces, except for ISP, most of our prototypes are standalone command line programs. We plan to integrate both SQL and graphics interfaces with our prototypes for better usability.

[7]http://www.vividsolutions.com/jts/JTSHome.htm
[8]http://akka.io

References

[1] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop-gis: A high performance spatial data warehousing system over mapreduce. In *VLDB, 6(11)*, pages 1009–1020, 2013.

[2] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *SSDBM*, pages 302–319, 2009.

[3] A. Eldawy and M. Mokbel. A demonstration of spatialhadoop: an efficient mapreduce framework for spatial data. In *VLDB, 6(2)*, pages 1230–1233, 2013.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 5th ed.* Morgan Kaufmann, 2011.

[5] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. In *PPoPP*, pages 267–276, 2011.

[6] D. B. Kirk and W.-M. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach (2nd Ed.)*. Morgan Kaufmann, 2012.

[7] M. Kornacker and et al. Impala: A modern, open-source sql engine for hadoop. In *CIDR*, 2015.

[8] M. McCool, A. Robison, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[9] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2005.

[10] A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. In *VLDB*, pages 277–288, 2013.

[11] D. M. Theobald. *GIS Concepts and ArcGIS Methods, 2nd Ed.* Conservation Planning Technologies, Inc, 2005.

[12] S. You and J. Zhang. Constructing natural neighbor interpolation based grid dem using cuda. In *COM.Geo*, 2012.

[13] S. You, J. Zhang, and L. Gruenwald. Parallel spatial query processing on gpus using r-trees. In *ACM BigSpatial*, pages 23–31, 2013.

[14] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *IEEE CloudDM workshop (To Appear)* `http://www-cs.ccny.cuny.edu/~jzhang/papers/spatial_cc_tr.pdf`, 2015.

[15] J. Zhang. Towards personal high-performance geospatial computing (hpc-g): perspectives and a case study. In *HPDGIS*, pages 3–11, 2010.

[16] J. Zhang. Speeding up large-scale geospatial polygon rasterization on gpgpus. In *HPDGIS*, pages 10–17, 2011.

[17] J. Zhang and D. Wang. High-performance zonal histogramming on large-scale geospatial rasters using gpus and gpu-accelerated clusters. In *IEEE IPDPSW (AsHES)*, pages 993–1000, 2014.

[18] J. Zhang and S. You. Speeding up large-scale point-in-polygon test based spatial join on gpus. In *ACM BigSpatial*, pages 23–32, 2012.

[19] J. Zhang and S. You. High-performance quadtree constructions on large-scale geospatial rasters using gpgpu parallel primitives. *IJGIS*, 27(11):2207–2226, 2013.

[20] J. Zhang and S. You. Efficient and scalable parallel zonal statistics on large-scale species occurrence data on gpus. *Technical Report:* `http://www-cs.ccny.cuny.edu/~jzhang/papers/szs_gbif_tr.pdf`, 2014.

[21] J. Zhang, S. You, and L. Gruenwald. Indexing large-scale raster geospatial data using massively parallel gpgpu computing. In *ACM GIS*, pages 450–453, 2010.

[22] J. Zhang, S. You, and L. Gruenwald. Parallel quadtree coding of large-scale raster geospatial data on gpgpus. In *ACM GIS*, pages 457–460, 2011.

[23] J. Zhang, S. You, and L. Gruenwald. U2stra: high-performance data management of ubiquitous urban sensing trajectories on gpgpus. In *ACM CDMW Workshop*, pages 5–12, 2012.

[24] J. Zhang, S. You, and L. Gruenwald. Data parallel quadtree indexing and spatial query processing of complex polygon data on gpus. In *ADMS workshop*, pages 13–34, 2014.

[25] J. Zhang, S. You, and L. Gruenwald. High-performance spatial query processing on big taxi trip data using gpgpus. In *IEEE BigData Congress*, pages 72–79, 2014.

[26] J. Zhang, S. You, and L. Gruenwald. Parallel online spatial and temporal aggregations on multi-core cpus and many-core gpus. *Information Systems*, 4:134–154, 2014.

[27] J. Zhang, S. You, and L. Gruenwald. Scalable and efficient spatial data management on multi-core cpu and gpu clusters: A preliminary implementation based on impala. *Technical Report:* `http://www-cs.ccny.cuny.edu/~jzhang/papers/isp_gpu_tr.pdf`, 2015.

[28] J. Zhang, S. You, and K. Zhao. Polygon overlay operations in cudagis: Design draft. *Technical Report:* `http://www-cs.ccny.cuny.edu/~jzhang/PolyOvr.html`, 2012.